Fundamentals for Computer Science

Master of Engineering: Computer Science

July 8, 2018

B. Demoen
KU Leuven
Departement Computerwetenschappen

Contents

1 Foreword 1

2 Introduction to graph theory 2

2.1 Introduction . . . 2
2.2 Graphs . . . 7
2.3 Trees . . . 29
2.4 Network models . . . 44
2.5 References . . . 59

3 Languages and Automata 60

3.1 Introduction . . . 60
3.2 What is a language? . . . 62
3.3 An algebra of languages . . . 64
3.4 Language descriptions . . . 65
3.5 Regular expressions en regular languages . . . 66
3.6 The subalgebra of regular languages . . . 69
3.7 Finite automata . . . 70
3.8 The transition table . . . 73
3.9 An algebra of NFAs . . . 74
3.10 From regular expression to NFA . . . 77
3.11 From NFA to regular expression . . . 78
3.12 Deterministic finite state machines . . . 81
3.13 The minimal DFA . . . 85
3.14 The pumping lemma for regular languages . . . 89
3.15 Intersection, difference and complement of two DFAs . . . 92
3.16 Regular expressions and lexical analysis . . . 93
3.17 Variants of finite state automata . . . 95
3.18 References . . . 99
3.19 Context free languages and their grammar . . . 101
3.20 The pushdown automaton . . . 109
3.21 Equivalence of CFG and PDA . . . 112

3.22 A pumping lemma for Context Free Languages . . . 117
3.23 An algebra of CFLs? . . . 119
3.24 Ambiguity and determinism . . . 120
3.25 Practical parsing techniques . . . 122
3.26 Context Sensitive Grammar . . . 124

4 Languages and Computability 126
4.1 The Turing Machine as a Decider or Recognizer . . . 126
4.2 A graphical representation of Turing Machines . . . 130
4.3 Representation and simulation of TM computations . . . 132
4.4 Non-deterministic Turing Machines . . . 134
4.5 Languages, $\mathcal{P}(\mathbb{N})$, properties and terminology . . . 134
4.6 Encoding . . . 134
4.7 Universal Turing Machines . . . 136
4.8 The Halting problem . . . 138
4.9 The enumerator machine . . . 140
4.10 Decidable languages . . . 142
4.11 Undecidable languages . . . 145
4.12 Some trivia on languages . . . 149
4.13 Countable . . . 150
4.14 How to kill N birds with one stone: Rice's Theorem . . . 151
4.15 The *Post Correspondence Problem* . . . 153
4.16 Many-one reduction . . . 156
4.17 Oracle machines and a hierarchy of decidability . . . 158
4.18 The computable functions and the recursive functions . . . 160
4.19 The busy beaver and fast increasing functions . . . 164

5 Introduction to Complexity Theory 166
5.1 Time complexity of algorithms . . . 167
5.2 Time complexity of languages or decision problems . . . 170
5.3 The time hierarchy . . . 184
5.4 co-**NP** . . . 184
5.5 **NP**–complete or **P**? . . . 185
5.6 Space complexity . . . 186
5.7 A sequence of inclusions . . . 189
5.8 Complexity and Chomsky hierarchy . . . 189
5.9 Conclusion . . . 189
5.10 References . . . 190

Chapter 1

Foreword

Algorithms are at the heart of computer science. Effective algorithms for concrete problems are always based on a goed choice of the abstraction and a formal insight in that abstraction. Apparently, *graphs* offer an often useful abstraction context, so we start this course with an introduction to graph theory: basic concepts, a few theorems, and some algorithms that are based on the theorems. Graphs will be useful in the later chapters. After graph theory follow chapters on *languages*: we build up a hierarchy of languages, the necessary machinery to decide those languages, and the formalisms to specify them. We pay particular attention to non-decidable languages, because they define the limits of what can be achieved by means of an algorithm. The course notes end with an introduction to complexity theory: one of the central problems of computer science is defined precisely, i.e. the question whether **P** equals **NP**. Besides time complexity, we also discuss space complexity.

This text is a translation of the course notes for *Fundamenten voor de Computerwetenschappen*, and it contains pieces of course notes from other courses, in particular

* *Fundamenten voor de Informatica* 1ste Bachelor Informatica, written with my colleague K. De Kimpe in 1997,
* *Automaten en Berekenbaarheid* 3de Bachelor Informatica, written in 2004.

This explains the non-uniform style.

Sources and extra material are mentioned at the end of a chapter, the end of a subject, or the end of the course notes. On purpose, there are no exercises in this text, but often, the student is invited to do certain things themselves.

Chapter 2

Introduction to graph theory

Graphs can often be used as an abstraction for solving a concrete problem. They are a context in which many algorithms were (and are) developed): shortest-path algorithms (think of GPS), spanning trees (optimalization of networks), graph coloring (register allocation by a compiler), maximal flow in networks, cycles of all kinds ... Therefore, it pays of to study formally the abstract context of graphs. This chapter deals with definitions related to graphs, and examples of some particular graphs, planar graphs - Euler's formula - K_3 en $K_{2,2}$ as minimal non-planar graphs, Eulerian and Hamiltonian cycles, weighted graphs, minimal spanning tree (Kruskal and Prim), a maximal flow algorithm and the SCCS algorithm, graph coloring, problem modeling with graphs.

Introduction

Informally, a graph is picture with dots (we name them vertices) connected by lines (edges). We will make the definitions more formal later.

Three coins

Three coins are on a table heads (H) up, tails (T) down. You are allowed to perform the following action as often as you want: flip any two coins. Question: is it possible that after some actions, all coins are tails up?

One way to solve this (type of) problem goes as follows: draw a graph with each of the possible configurations of the 3 coins as a vertex, and an edge between two vertices if the allowed action transform one configuration into the other. We get the graph in Figure 2.1.

Just looking at the graph shows us that it is impossible to get from the HHH configuration to the TTT configuration: they belong to a different component of the graph. The property

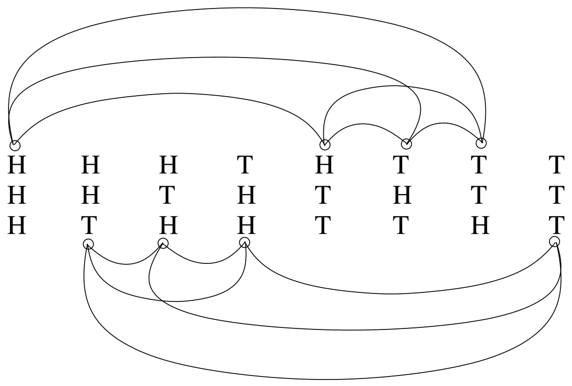


Figure 2.1: The 3 coins graph

used here is *reachability*.

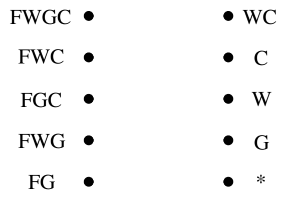
The wolf, the goat, the cabbage, and the farmer

A farmer owns a wolf, a goat and a cabbage. He lives at the left bank of a river, and he wants to move to the right bank. He also has a small boat: it is too small to put all his belongings in it at the same time, in fact, it is so small he can transport at most one item at a time. He could transport one by one his wolf, goat, and cabbage to the right bank, in some particular order, but the problem is that when the wolf and the goat are left alone, the wolf will eat the goat. Similarly for the goat and the cabbage: they can't be left alone. So this simple plan does not work. Can the farmer transport all his belongings to the other side so that nothing gets eaten?

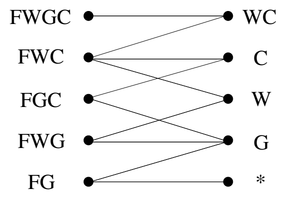
You could of course try out all possibilities - why not try it now! You notice that there is a problem of bookkeeping while doing that. A better way is note down all possible situations: a situation describes who/what is on which side of the river, and is characterized completely by what is on the right side. So, the situation FGC means that the farmer, the goat and the cabbage are on the left bank, while the wolf is at the right bank. Some situations - like GC - are a priori forbidden. All situations can be described by the dots in Figure 2.2(a): we use $*$ to indicate that nothing is at the left bank: it is the situation the farmer really wants.

We can now connect any two situations α and β if the farmer can cross the river - possibly with one of his belongings - and by doing so transforms α into β. We get the graph in Figure 2.2(b). The problem is now reduced to the question: is there a path from vertex FWGC to $*$ using the edges in Figure 2.2(b)?

Again, reachability is the key concept, and the details of the problem have been abstracted



(a) All possible situations for the farmer, wolf, goat, and cabbage



(b) The graph describing the situations and their allowed transitions

Figure 2.2: The farmer, wolf, goat and cabbage problem

away.

The bridges in Königsberg

18th century, Königsberg (now Kaliningrad in Russia): the river Pregel crosses the city. It contains two islands, connected by a total of 7 bridges with each other and the river banks: see Figure 2.3. The citizens if Königsberg used to make walks during the weekend, and they wondered: is it possible to make a walk (starting at home and arriving at home of course) so that every bridge is crossed exactly once? They couldn't solve it. When the swiss mathematician Leonhard Euler (1707-1783) visits the town, he hears about the problem, solves it (negatively), and publishes the first paper ever on graph theory.

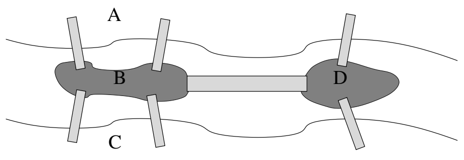


Figure 2.3: The bridges on the river Pregel

How is this problem connected to graph theory? The graph representation of the bridges and river banks can be seen in Figure 2.4. The essence of the problem is now: does the graph have a cycle (a closed path) that uses every edge exactly once? This kind of cycle is now names an Eulerian cycle. We will later see that the characterization of graphs with an Eulerian cycle is simple, as well as finding an Eulerian cycle.

You might know the problem of finding an Eulerian cycle (or path) in a different context: you are given a drawing in which points are connected by lines; you are asked to draw it by putting your pen at one point, never lifting it, following all lines not more than once.

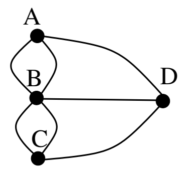


Figure 2.4: The graph modeling the bridges over the Pregel

Finding Eulerian cycles is important in real life: suppose you need to check the state of the median strip of the highways. You need to traverse all highways, but preferable only once.

Hamilton's toy

Sir William Rowan Hamilton tried to commercialize circa 1850 a 3-D puzzle in the form of a dodecahedron (it has 12 pentagons): Figure 2.5 shows a planar representation of this spatial object. Each corner had the name of a city and the problem consisted in finding a closed path (a cycle) starting at a particular city so that every city is visited exactly once. Such path is named a Hamiltonian cycle. It turns out that finding such a cycle (or proving it exists) is much more difficult than for Eulerian cycles.

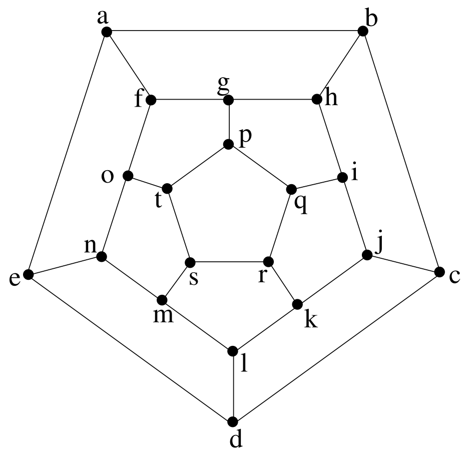


Figure 2.5: Hamilton's puzzle

The puzzle was a commercial disaster, but it is at the heart of the Travelling Salesman Problem we will encounter in complexity theory.

The Postman Problem has a similar flavor: during his tour, a postman wants to visit each street exactly twice (once at each side of the street) ... is that always possible? . . .

Figure 2.6 shows a Hamiltonian cycle for Hamilton's puzzle.

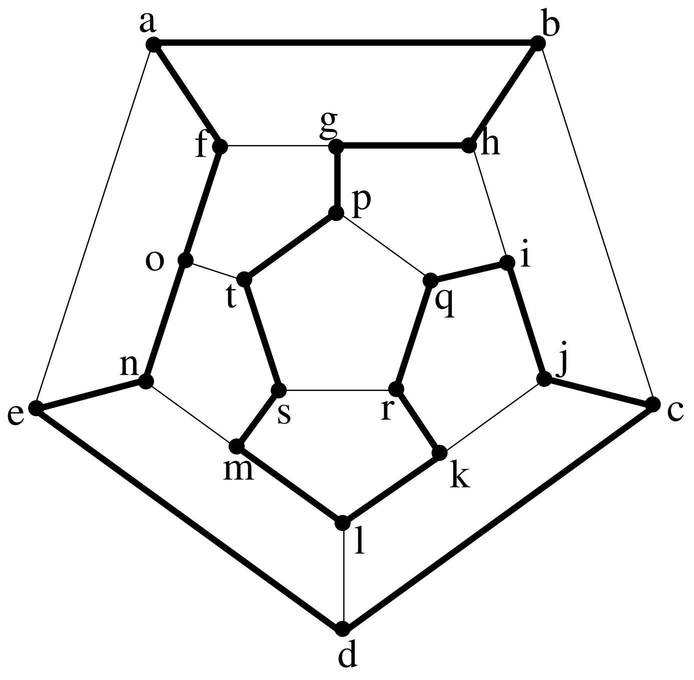


Figure 2.6: A solution for Hamilton's puzzle

Graphs

Definitions and all kinds of paths

Definition 2.2.1. *Graph*
A (undirected) **graph** G is tuple (V, E) in which V is a set of **vertices** and E a (multi-) set [1] of **edges**; each edge $e \in E$ is a unordered pair $(v, w) \in V \times V$; we write $e = (v, w)$ or $e = (w, v)$.

We allow some freedom of notation and speech:

- we write $G(V, E)$ for the graph G with vertices V and edges E
- we write $e \in G$ when $e \in E$ and E is a subset of the edges of G
- we write $v \in G$ when $v \in V$ and V is a subset of the vertices of G
- let the edge $e = (x, y)$, and $G(V, E)$, then $G \cup \{e\}$ denotes the graph $(V \cup \{x, y\}, E \cup \{e\})$; we say *add e to G*
- let b be an vertex, and $G(V, E)$, then $G \cup \{b\}$ denotes the graph $(V \cup \{b\}, E)$; we say *add b to G*

We assume sometimes that the vertices are numbered from 1 to $n = |V|$. We will deal only with finite graphs.

Edge is also used in the context of polyhedra, and that is no surprise: the study of polyhedra and graphs have common points.

Note that E can contain two edges (v, w): we name those edges *parallel*.

An edge (v, w) is *incident on* the vertex v (and on w); the vertices v and w are *adjacent to* the edge (v, w).

Definition 2.2.2. *Directed graph (digraph)*
A (undirected) **graph** G is tuple (V, E) in which V is a set of **vertices** and E a (multi-) set of **edges**; each edge $e \in E$ is an ordered pair $(v, w) \in V \times V$; we write $e = (v, w)$.

You can think of a directed graph as a graph in which each edge has an arrow, pointing from one vertex to the other.

[1] A *$multi - set$* is similar to a set, but an element can occur more than once; $\{1, 1, 1, 2, 2, 3, 4, 5, 5\}$ is multi-set.

Definition 2.2.3. *Loop*
A **loop** in a graph is an edge (v, v).

Definition 2.2.4. *Simple graph*
A graph is **simple** if the graph has no parallel edges neither loops.

Definition 2.2.5. *Degree of a vertex*
The **degree** $\delta(v)$ **of a vertex** v of graph (V, E) is the number of edges $(v, w) \in E$, where a loop counts for 2.

Definition 2.2.6. *Isolated vertex or point*
A vertex v in a graph (V, E) is **isolated** *if $\delta(v) = 0$.*

Theorem 2.2.7. *Sum of vertex degrees*
The sum of all vertex degrees is even.

Proof. Since each edge (v, w) contributes 1 to the degree of both v as w, each edge contributes 2 to the sum of all vertex degrees, hence the conclusion. ■

Theorem 2.2.8. *Number of vertices with odd degree*
The number of vertices of odd degree is even.

Proof. Partition V in the vertices with even degree $\{a_i | i = 1, \ldots, n\}$, and the vertices with odd degree $\{b_i | i = 1, \ldots, m\}$. Then, because of Theorem 2.2.7

$$0 = \left(\sum_{i=1}^{n} \delta(a_i) + \sum_{i=1}^{m} \delta(b_i) \right) \bmod 2 = m \bmod 2$$

and the conclusion follows. ■

Definition 2.2.9. *Path*
A **path** (with length n) in a graph (V, E) is a sequence of edges

$$(e_1 = (v_1, v_2),\ e_2 = (v_2, v_3),\ \dots, e_n = (v_n, v_{n+1})).$$

We assume that it is clear which of the parallel edges is used (if at all important).
In a simple graph, a path can also be characterized by a sequence of vertices $(v_1, \dots, v_{n+1})$

Definition 2.2.10. *Simple path*
A **simple path** $(v_1, \dots, v_{n+1})$ is a path with the property that $\forall i, j : i \neq j \Rightarrow v_i \neq v_j$

Definition 2.2.11. *Cycle*
A **cycle** is a path $((v_1, v_2), ..., (v_n, v_{n+1}))$ such that all edges in it are different and $v_1 = v_{n+1}$.

A cycle is sometimes name a circuit.

Definition 2.2.12. *Eulerian cycle (path)*
An **Eulerian cycle (path)** is a cycle (path) containing all edges exactly once, and all vertices.

Definition 2.2.13. *Hamiltonian cycle*
A **Hamiltonian cycle** is a cycle containing all vertices of the graph exactly once.

Definition 2.2.14. *Connected graph*
A graph is **connected** if for every two different vertices v and w, there is a path from v to w.

Theorem 2.2.15. *Existence of an Eulerian cycle.*
A graph $G(V, E)$ has an Eulerian cycle if and only of G is connected and the degree of each vertex is even.

The intuition behind the proof is that when you arrive in a vertex, you can leave again because the degree is even. It therefore does not matter which path you follow, as long as you don't return to the starting point too soon. Even if you did, you can still mend the path.

Proof.

- A graph with an Eulerian cycle is connected: the cycle connects all vertices, so there is a path from each vertex to each other vertex. Wile traversing the cycle, we leave every vertex in which we arrive, and since all edges are traversed by the cycle, the degree of each vertex is even.
- Construct a cycle P: start from any vertex s, follow any edge incident on s; from then on extend the partial path with any edge incident on the most recently added vertex, never using a previously used edge. Repeat until there is no more edge to choose from. Since the degree of each vertex is even, the just constructed P is a cycle starting in s and arriving at s. If P contains all edges from G, the theorem is proven. Otherwise, there exists a vertex s' on the path P, from which an unused edge leaves (explain why !). Construct a cycle P' starting at s' not using any edges from P (why is this possible?). Then add P' to P: you have now a larger cycle. One can repeat the procedure until there are no more unused edges. Figure 2.7 illustrates the construction.

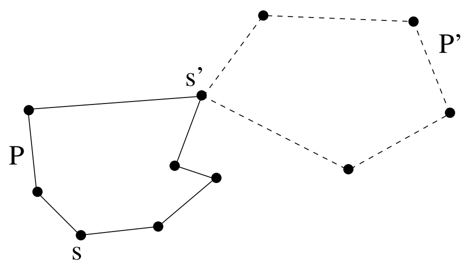


Figure 2.7: P and the extension P'

■

Theorem 2.2.16. *Existence of an Eulerian path*
A connected graph G has an Eulerian path from vertex v to w ($v \neq w$) if v and w are the only vertices with odd degree.

Proof. Consider the graph G' obtained by adding the edge (w, v) to G. G' is connected and each vertex has even degree, so it has an Eulerian cycle $(w, v, \ldots, w)$; delete from this cycle the first edge to obtain the Eulerian path $(v, \ldots, w)$ in G. ■

Looking back at Figure 2.4 (the graph modeling the bridges in Königsberg), you notice that there are four vertices with odd degree. It follows there is no Eulerian cycle, neither an Eulerian path.

Definition 2.2.17. *Subgraph*
A graph (V_1, E_1) is a **subgraph** of (V, E) if $V_1 \subseteq V$ en $E_1 \subseteq E$.

Definition 2.2.18. *Component of a graph*
A **component** C of a graph G is a maximal connected subgraph of G, i.e. $\forall C' \subseteq G$: $C \subset C' \Rightarrow C'$ is not connected.

Theorem 2.2.19. *Partition of a graph*
The components (V_i, E_i) $(i = 1, \dots, n)$ of a graph (V, E) form a partition, i.e.
$(V, E) = (\cup_{i=1}^{n} V_i, \cup_{i=1}^{n} E_i)$ *and for any $i \neq j$, $V_i \cap V_j = \emptyset$ en $E_i \cap E_j = \emptyset$*

Proof. Clearly every vertex and edge belongs to at least one component. Suppose a vertex belongs to two components α en β, then the union of α and β is a connected subgraph of (V, E) so $\alpha = \beta$, so $V_i \cap V_j = \emptyset$ for $i \neq j$. Since an edge belongs to the component of its incident vertices, the proof can be completed easily. ■

Theorem 2.2.19 is important because it is often easier to prove properties for connected graphs: Theorem 2.2.19 tells us that a non-connected graph can be uniquely divided in connected components.

Graph representation

Until now we have just drawn graphs. However, we need often a more formal representation of graphs, e.g. because we need to write programs dealing with graphs. There is no single best representation: it depends on what we want to compute. So we introduce several representations, first for undirected graphs, later for digraphs.

Let us number the $n = |V|$ vertices of $G(V, E)$ from 1 to n. Construct an $n \times n$ matrix A with $A(i, j)$ equal to 1 if $(i, j) \in E$ and 0 otherwise. A is called the adjacency matrix G: it readily shows which vertices are adjacent.

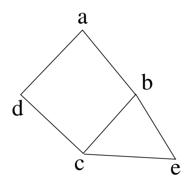


Figure 2.8: Some graph

The adjacency matrix A of the graph in Figure 2.8 is

$$\begin{array}{c|ccccc} & a & b & c & d & e \\ a & 0 & 1 & 0 & 1 & 0 \\ b & 1 & 0 & 1 & 0 & 1 \\ c & 0 & 1 & 0 & 1 & 1 \\ d & 1 & 0 & 1 & 0 & 0 \\ e & 0 & 1 & 1 & 0 & 0 \end{array}$$

The adjacency matrix of a simple graph has only zeros on the main diagonal. The adjacency matrix does not show whether a graph has parallel edges. The adjacency matrix is symmetric and hence not a very efficient representation of a graph. Still, it has interesting properties.

Compute A^2 for the example above. We get

$$A^2 = \begin{pmatrix} 2 & 0 & 2 & 0 & 1 \\ 0 & 3 & 1 & 2 & 1 \\ 2 & 1 & 3 & 0 & 1 \\ 0 & 2 & 0 & 2 & 1 \\ 1 & 1 & 1 & 1 & 2 \end{pmatrix}$$

We got the (a, c)-the element van A^2 by:

$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} = 0 \times 0 + 1 \times 1 + 0 \times 0 + 1 \times 1 + 0 \times 1 = 2$$

The positive contributions in the result result from the paths $(a, b)-(b, c)$ and $(a, d)-(d, c)$. This shows that $A^2[i, j]$ = the number of paths vertex i to vertex j with length 2. But we need to be careful: the adjacency matrix does not show parallel edges, and parallel edges increase the number of paths between two vertices. As a consequence, the next theorem is about simple graphs.

Theorem 2.2.20.
Let A be the adjacency matrix of a simple graph $G(V, E)$,
then $A^n[i, j]$ = the number of paths with length n from vertex i to vertex j.

Proof. We use induction on n. For $n = 1$, the theorem is true because of the definition of adjacency matrix.

Suppose the theorem is true for n; we prove that the theorem is true for $(n+1)$: $A^{n+1} = A^n * A$, so $A^{n+1}[i,j] = \sum_{k=1}^{\#V} A^n[i,k] * A[k,j]$. By the induction hypothesis, $A^n[i,k]$ is the number of paths from i to k, and if there is an edge from k to j, (i.e. if $A[k,j] = 1$), then there are also $A^n[i,k]$ paths from i to j passing through k just before arriving at j. Since no two paths are the same (why not?), the result for $(n+1)$ follows. ■

For the graph in Figure 2.8, we have

$$A^4 = \begin{pmatrix} 9 & 3 & 11 & 1 & 6 \\ 3 & 15 & 7 & 11 & 8 \\ 11 & 7 & 15 & 3 & 8 \\ 1 & 11 & 3 & 9 & 6 \\ 6 & 8 & 8 & 6 & 8 \end{pmatrix}$$

You can check manually that the theorem is true.

Making use of the previous result, we can derive that $(\sum_{k=1}^{n} A^k)[i,j]$ equals the number of paths from i to j and equal to or shorter than n edges.

The adjacency matrix can also be used in a different way: interpret the numbers 0 and 1 as the boolean values $false$ and $true$, and define boolean matrix multiplication as

$$(A * B)[i,j] = (A[i,1] \wedge B[1,j]) \vee (A[i,2] \wedge B[2,j]) \vee \ldots \vee (A[i,n] \wedge B[n,j])$$

If B is the boolean adjacency matrix of G, then $B^n[i,j]$ equals the truth value of the sentence *there exists a path of length n from i to j.*

Analogously, $(\sum_{k=1}^{n} B^k)[i,j]$ (sum means boolean sum, i.e. $\vee$) is the truth value of *there exists a path from i to j of length smaller than or equal to n.*

For the usual adjacency matrix, the entries in this sum, for successive values of n, grow indefinitely, but for the boolean adjacency matrix, the entries can only be $false$ or $true$. Moreover, they are monotonically increasing, so the limit exists. In fact, the limit is reached after at most $|V|$ steps, and it is one way to compute the transitive closure of the relation *is connected by an edge.*

Another graph representation consists of the incidence matrix I: it has a row for each vertex and a column for each edge. $I[i,j] = 1$ iff the j-de edge is incident on i and 0 otherwise. The incidence matrix of the graph in Figure 2.8 is

$$\begin{array}{c|cccccc} & ab & ad & cd & bc & ce & be \\ \hline a & 1 & 1 & 0 & 0 & 0 & 0 \\ b & 1 & 0 & 0 & 1 & 0 & 1 \\ c & 0 & 0 & 1 & 1 & 1 & 0 \\ d & 0 & 1 & 1 & 0 & 0 & 0 \\ e & 0 & 0 & 0 & 0 & 1 & 1 \end{array}$$

The incidence matrix shows loops and parallel edges.

Instead of using a adjacency matrix, one can also use an *adjacency list*: it is just a list of all edges.

The representation of digraphs can be done as for non-directed graphs: the adjacency matrix has a 1 on the $(i, j)^{th}$ entry if there is a directed edge from i to j, and otherwise a 0. Usually, this matrix is not symmetric, but since we never used that fact for non-directed graphs, Theorem 2.2.20 applies also to digraphs, and the idea of the boolean adjacency matrix does too.

One use of the boolean adjacency matrix is in finding out the functions that are called (directly or indirectly) from a given function in a given program: you can work that out yourself, and also discover how you can detect *dead code* (compiler jargon), i.e. functions that are not called by anyone. In Section 2.2.7 op pagina 26, this application example is used in an interesting graph algorithm.

Graph Isomorphism

We do the following experiment: all of you take a pen and paper, and without looking at what your neighbour is doing, you follow the following instructions:

- draw 5 points and put a name next to each of them
- connect the first point you drew with the second point
- connect the second with the third
- connect the third with the fourth
- connect the fourth with the fifth
- connect the fifth with the first

There is a chance that some students come up with one of the graphs in Figure 2.9.

All these graphs look different, but they were drawn following the same instructions: there is a good reason for considering them *the same* in some sense, hence the following definition:

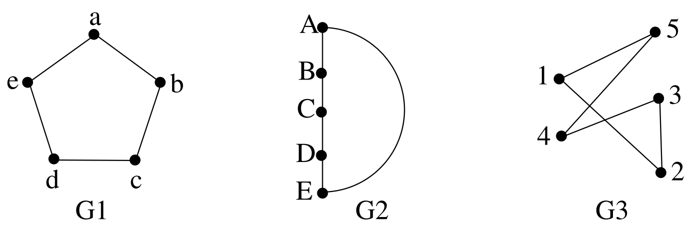


Figure 2.9: Three different graphs, or are they equal?

Definition 2.2.21. *Graph Isomorphism*
The graphs $G_i(V_i, E_i)$ (i = 1,2) are **isomorphic**, if there exists a bijection $f : V_1 \rightarrow V_2$ such that $g : E_1 \rightarrow E_2$ specified by $g((v,w)) = (f(v), f(w))$ for all $v, w \in E_1$ is well defined, i.e. $g((v,w)) \in E_2$ and a bijection. Such f is named a graph isomorphism.

Between G_1 and G_2 of Figure 2.9, there exists the isomorphism f:

f(a) = B
f(b) = C
f(c) = D
f(d) = E
f(e) = A

You can check that this is a graph isomorphism between G_1 and G_2. Another characterisation of isomorphic graphs relies on the following theorem:

Theorem 2.2.22. *Characterisation of isomorphic graphs with the incidence matrix*
Graphs G_1 and G_2 are isomorphic if and only if there is an ordering of the vertices and edges so that the incidence matrices of G_1 and G_2 are equal.

Proof.
- Let G_1 and G_2 be isomorphic, with isomorphism f. Choose any order of the vertices of G_1. Now order the vertices of G_2 by: $f(v) < f(w) \Leftrightarrow v < w$; the order on the edges of G_2 is induced in the same way by any order on the edges of G_1 and the bijection on the edges. The incidence matrices are equal.
- If the incidence matrices are equal, the isomorphism f is trivial to specify.

■

We know that the adjacency matrix does not fully determine a graph, since parallel edges and loops are not visible in the adjacency matrix. Consequently the following theorem is limited to simple graphs.

Theorem 2.2.23. *Characterisation of isomorphic simple graphs with the adjacency matrix*
The simple graphs G_1 and G_2 are isomorphic if and only if there is an ordering of the vertices so that the adjacency there is an ordering of the vertices and edges so that the incidence matrices of G_1 and G_2 are equal.

Proof. Please prove this yourself. ■

The theorems on graph isomorphism provide a algorithms using the concrete representation of the graphs. However, every currently known algorithm for testing whether two graphs are isomorphic, is at least exponential. On the other hand, there exist very efficient tests that can show that two graphs are *not* isomorphism: one checks whether some graph properties that is *invariant* under graph isomorphism holds for both graphs. Such property can be based on the number of vertices, the number of edges, the degrees of the vertices ... We will later see more of such properties. But remember that these simple tests never tell you that two graphs are isomorphic, only that two graphs are not isomorphic.

Weighted graphs

Definition 2.2.24. *Weighted graph*
A **weighted graph** (V, E) is a graph in which each edge $e \in E$ has a **weight** $w(e) \in \mathbb{R}_0^+$. The weight $w(G)$ of a weighted graph G, is the sum of the weights of all the edges of G.

The weight of a of an edge can be interpreted as the length of an edge, or some cost associated with the use of the edge in a path: if the graph models a road network, then the weight can be a distance between cities, or a tol to be payed to use the road. The *weight of a path* is the sum of the weights of the edges in the path. It is clear that in the case of road networks, one would be naturally interested in minimal weight paths connecting two given nodes, or put differently, one is interested in *shortest* paths. Since two such nodes are necessarily in the same component, this section deals with connected graphs.

The *existence* of a shortest path in a connected (and finite) graph is easy to prove (do it anyway). We are now interested in *constructing* a shortest path. We restrict ourselves to simple graphs: loops can't occur in a shortest path, and of the parallel edges between two nodes, we need only the shortest one.

Algorithms in this chapter consist of a sequence of instructions, numbered 1,2, ... At the end of instruction n, there is an implicit **Go to instruction** $n + 1$. *Stop* means the algorithm finishes.

Planar graphs

Informally, a planar graph is a graph one can draw on a sheet of paper so that no two edges cross: there is no need for the third dimension to achieve this non-crossing. There is a formal definition as well: look it up. We will use the informal definition in proofs.

Another important class of graphs is the class of *bipartite* graphs.

Definition 2.2.25. *Bipartite graph*
A **bipartite graph** is a graph (V, E) with $V = V_1 \cup V_2$ so that $V_1 \cap V_2 = \emptyset$ en $E \subseteq V_1 \times V_2$

In words: in a bipartite graph, one can identify two disjoint subsets of vertices, so that all the edges have an element of each subset as an endpoint (and consequently, there are no edges that connect nodes from the same subset).

Figure 2.10 shows one such graph at the right.

$K_{n,m}$ denotes the bipartite graph in which $|V_1| = n$ and $|V_2| = m$, and such that every vertex in V_1 is connected to every vertex of V_2. $K_{n,m}$ occurs naturally when one considers the problem of connecting n houses with m utilities (water, gas, electricity, internet …).

K_n denotes the graph with n vertices, in which there is an edge between every two vertices. Such graphs are *completely connected*, and we name them cliques.

The K is in honour of Kazimierz Kuratowski, a famous graph theorist.

Figure 2.10 shows K_4 and $K_{2,2}$.

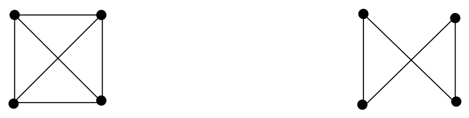


Figure 2.10: K_4 en $K_{2,2}$

When you try drawing K_n on paper for successive values of n, and such that no two edges cross, you notice that from $n = 5$ on, it is no longer possible. Similarly, trying this for successive values of (n, m) and $K_{n,m}$, it fails for all $n, m > 2$.

As we said before, graphs one can draw in the plane without crossing edges are named **planar**. As an application: if the graph of a road network is planar, there is no need for tunnels or bridges. Euler proved in 1752 the following formula[2]:

[2]Descartes (± 1600) knew the formula and most probably so did Archimedes (± –250)

Theorem 2.2.26. *Euler's formula for planar graphs:*
If G is a connected planar graph with e edges, v vertices and f faces *then $v - e + f = 2$.*

We haven't defined a face yet: informally, it is a piece of the plane enclosed by a minimal cycle, and also the infinite part of the plane that lies outside of a (drawing of a) graph counts as a face. The origin of the word *face* is in the study of polyhedra.

Proof. We use induction on the number of edges. Suppose $e = 1$. Then G is one of the graphs in Figure 2.11. In both cases, the Euler's formula is correct. Suppose the formula is correct for all graphs with n edges. Let G be a graph with $(n + 1)$ edges.

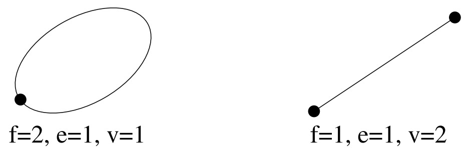


Figure 2.11: The two graphs with $e = 1$

1) First assume that G does not contain a cycle. Consider a maximal simple path in G; that path contains a vertex a with $\delta(a) = 1$: now consider the graph G' which you obtain by removing from G the vertex a and the only edge arriving there; G' has 1 vertex and 1 edge less than G, the same number of faces, and G' is also connected (why?), so Euler's formula is valid for G', and consequently also for G (see e.g. Figure 2.12).

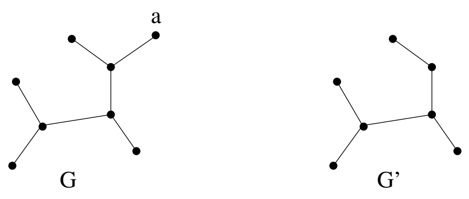


Figure 2.12: G without a cycle

2) Suppose now that G contains a cycle: take away any edge x from this cycle, but not its endpoints; the result is G' (see e.g. Figure 2.13). G' has n edges, the same number of vertices as G, and one less face than G; G' is connected (why?); so Euler's formula is valid for G' (by the induction hypothesis), and it follows that Euler's formula is also valid for G. ■

Note that the connectedness condition is essential: the graph with two vertices and no edge, has $f = 1, e = 0, v = 2$, so Euler's formula is not valid. You should be able to

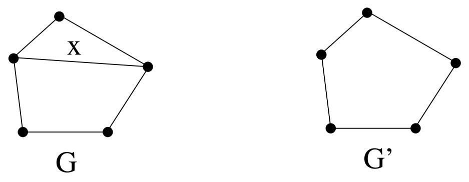


Figure 2.13: G with a cycle

generalize Euler's formula to non-connected graphs, taking into account the number of components.

The origin of Euler's interest in this formula lies in the study of spatial objects, more specifically the polyhedra. This interest existed already since Greek ancient history: regular polyhedra were supposed to be the building blocks of the world. The formula expresses the relation between the number of faces, edges and vertices of a polyhedron. The transformation from a polyhedron and a planar graph can be visualised as follows: image the polyhedron is drawn on a balloon. Carefully make a hole in the middle of any face, and stretch the hole until all the material is in a plane. The resulting picture of edges and vertices form planar graph. The face that was punctured corresponds to the infinite face.

Theorem 2.2.27. *$K_{3,3}$ and K_5 are not planar.*

Proof. Suppose $K_{3,3}$ is a planar graph. Let β be the total number of edges in all faces: an edge counts n times of that edge occurs in n. Since (in a planar drawing) every edge can belong to at most 2 faces, we know that $2 * e \geq \beta$.

Moreover, every cycle in $K_{3,3}$ has at least 4 edges (why?) from which it follows that $\beta \geq 4 * f$. Together with the previous inequality, we derive $e \geq 2 * f$. Now use Euler's formula to eliminate f from the right hand side. We get $e \geq 2 * (e - v + 2)$. Fill out the values for e and v for $K_{3,3}$, and get $9 \geq 10$, which is not true. So $K_{3,3}$ is not planar.

Every cycle in K_5 has minimally 3 edges ... and the rest of the proof is like above. ■

When two graphs are isomorphic, they are basically the same up to a renaming of the vertices. But graphs sometimes look similar even though they they do not have the same number of edges or vertices. E.g. if we add a vertex in the middle of an edge, it feels like nothing essential has changed to the graph. For one thing, its planarity has not changed, neither its connectivity (and you may perhaps think of other properties that remain invariant). The operation of adding the extra vertex is named subdivision or expansion of the graph. The inverse operation is named *smoothing (out)* the graph. The latter consists of replacing two edges (a, b) and (b, c) of which $\delta(b) = 2$ by one edge (a, c). See Figure 2.14.

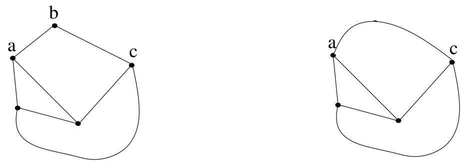


Figure 2.14: Smoothing a graph

Definition 2.2.28. Two graphs G_1 and G_2 are **homeomorphic** if by applying smoothing to both graphs, one obtains isomorphic graphs.

Theorem 2.2.29. *Kuratowski's Theorem*
A graph is planar if and only if it does not contain a subgraph homeomorphic to K_5 nor $K_{3,3}$.

Proof. One direction is obvious (but please give arguments!). The other direction is out of the scope of this course. ■

The above theorem basically says that K_5 and $K_{3,3}$ are in some sense the smallest non-planar graphs.

Graph coloring

Definition 2.2.30. A **coloring** of a graph (V, E) is an assignment of a color to each $v \in V$ such that the colors of v and w differ if $(v, w) \in E$. An **n-coloring** is a coloring with n or less (different) colors. A **minimal** coloring is an n-coloring with minimal n.

There are lots of practical applications of (minimal) graph colorings. Just a few examples:

- Suppose you need to plan 4 meetings, involving persons A,B,C en D. The first meeting must be attended by A and B; the second by A and C; the third by B, C and D, and the fourth one by C and D. What is the minimal number of time slots you can make all these meetings happen? You may assume you have enough meeting places, and that each meeting takes the same time. Of course, a person can't attend two meetings at the same time.

  You can represent this problem by the graph in Figure 2.15: each vertex corresponds to a meeting, and two meetings are connected if they can't be at the same moment (because some person needs to attend both). Now find a minimal coloring of this graph, et voilá! Each color corresponds to a new time slot.

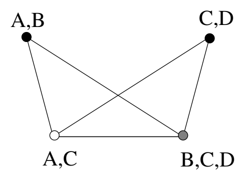


Figure 2.15: The meeting graph with a minimal coloring

- You need to organize the goods in a supermarket in the racks, but some goods can not be put next to each other, e.g. fuel can't be next to bread, porn not next to cleaning powder, etc. Once more, you can construct a graph representing the constraints, and a coloring of the graph solves your problem.
- A compiler tries to put integer variables in machine registers instead of in memory (why?). There are typically more variables than available registers, but the same register can be used for more than one variable. For example, consider the following code:

```
{
   int i,j,k,l,m;

   i = 1;
   j = 2;
   k = i+j;
   i = 3;
   j = 4;
   l = i+j;
   m = k+l;
}
```

  i and *l* can be assigned the same register, and likewise for *k* and *m*. You can find that out by constructing the inference graph of this piece of code: the graph has as vertices the variables, and an edge between two variables whose value is *alive* at the same moment. Figure 2.16 shows the corresponding graph.

  A minimal coloring determines the minimal number of registers needed to assign each variable in a register, and also an optimal assignment.

A problem with practical implications, and historically important, is the 4-color problem:

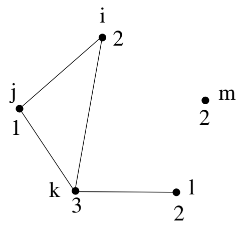


Figure 2.16: The interference graph and a coloring (with numbers)

is it possible to color each map with four colors, so that no two neighboring countries have the same color? This problem was set up by Francis Guthrie around 1850, and conjectured to have a positive answer. Despite several attempts, the conjecture remained unproven until 1976: K. Appel and W. Haken proved that there exist 1936 graphs of which at least one must be in any 4-colorable graph, and then proved that such graphs are not minimal ... The proof was performed by a computer program.

The connection between map coloring and graph coloring is rather easy for planar graphs. A map is a planar graph G, and can be turned into another planar graph - its *dual* - G' as follows: assume for the sake of simplicity that each region in the map is a country. Each country has a capital inside that country. Now construct the graph with as nodes the capitals, and define an edge between two capitals if their countries share a border. Figure 2.17 shows some examples.

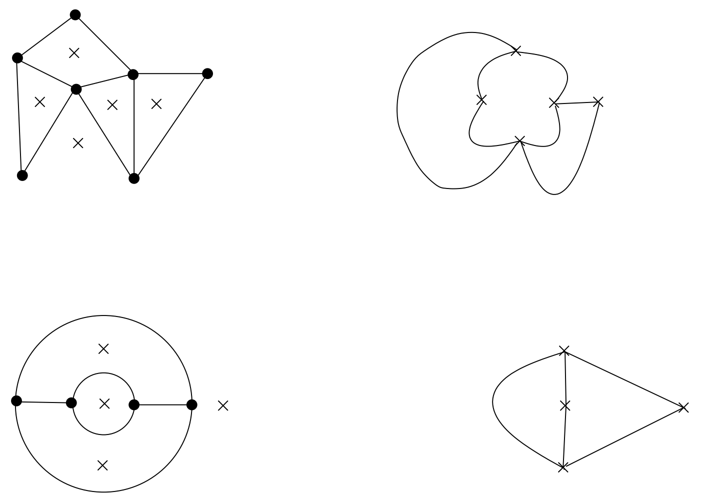


Figure 2.17: Two planar maps and their dual graphs

Note that the dual graph of a planar map is planar, and simple. A coloring of the dual graph, corresponds to a coloring of the map - and vice versa.

In the rest of this section, we prove the 5-color theorem: every planar graph can be colored with 5 colors.

Theorem 2.2.31. $e \leq 3 * v - 6$ *for every planar, simple graph* G *with more than one edge.*

Proof. We prove the theorem only for connected graphs: first remove from G every vertex with degree equal to 1 (and the corresponding edge), until no more such vertices exist. The resulting graph G' is still planar, simple and connected. Denote by e' the number of its edges, and v' the number of its vertices; the number of faces is f because removing the particular edges did not change that. So we have $e - e' = v - v'$ which is equal to the number of removed edges (or vertices). Denote that quantity by t. We now make a case distinction:

1) if $e' = 0$ then $v' = 1$; since $e > 1$ we also have $t > 1$; so we know: $e = t$ and $3 * v - 6 = 3 * (v' + t) - 6 = 3 * t - 3$; since $t \leq 3 * t - 3$, we get $e \leq 3 * v - 6$.

2) e' can't be equal to 1 or 2 (why not?)

3) $e' > 2$; every face is delimited by at least 3 edges (because there are no loops or parallel edges); let $\sum$ denote the sum of the number of edges in all faces. then $\sum \geq 3 * f$; since each edge borders exactly 2 faces (why?), we also have $\sum = 2 * e'$, so $2 * e' \geq 3 * f$. Now use Euler's formula for G' to eliminate f from this inequality, and you get: $e' \leq 3 * v' - 6$ and thus $e' + t \leq 3 * (v' + t) - 6$ and $e \leq 3 * v - 6$. ■

Theorem 2.2.32. *A planar, simple graph has at least one vertex - say* v *- with* $\delta(v) \leq 5$.

Proof. This is clearly tru for graphs with 1 or 2 vertices. Suppose the theorem is not tru for a certain graph with at least 3 vertices, i.e. this graph has degree at least 6 for each vertex. Then the sum of the degrees of all the vertices is at least $6 * v$, so, $e \geq 3 * v$, in contradiction with the $e \leq 3 * v - 6$ from the previous theorem. ■

We are now ready to state and prove

Theorem 2.2.33. *5 colors is enough to color any simple, planar graph* $G(V, E)$.

Proof. (the proof follows Townsend)

We use induction of the number of vertices: a G with just one vertex, can clearly be colored with 5 colors. Also, assume G is connected: if not, the theorem will be proven true for its components, and carry over to the whole graph. So, assume that the theorem is true for graphs with n vertices; let G have $n+1$ vertices. Then there is at least one vertex v with degree 5 or less.

1) If v has degree 4 or less, then v occurs in G as in Figure 2.18.

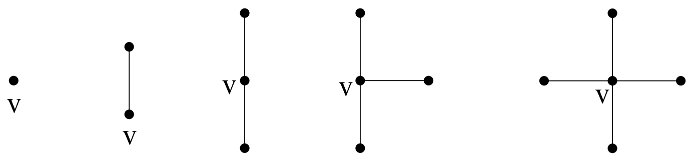


Figure 2.18: The 5 possibilities in which $\delta(v) < 5$

Consider the G' obtained by deleting v from G (and the edges arriving in v). G' has the required properties for the theorem, and it has only n vertices, so by the induction hypothesis, it has a 5-coloring. Since v has at most 4 neighbors, v can get a color different from the color of all its neighbors, resulting in a 5-coloring of G.

2) If $\delta(v) = 5$, then v occurs in G as in Figure 2.19: each neighbor w_i of v has a color c_i $(i = 1, \ldots, 5)$ as obtained by a 5-coloring of G'.

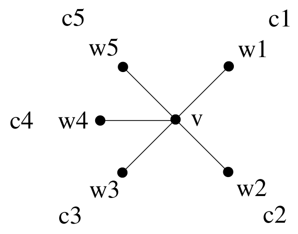


Figure 2.19: $\delta(v) = 5$

We still need to determine a color for v. If the neighbors of v do not use all 5 colors, we can give v any remaining color.

So, assume that all five c_i are different. Consider the set $P_{1,3}$ of all paths in G' starting at w_1 and whose vertices have alternating colors c_1 and c_3. We name such paths $c_1 - c_3$ paths. There are two possibilities:

$P_{1,3}$ **does not contain a path that contains** w_3**:** for each vertex in the paths $P_{1,3}$, change color c_1 in c_3 and vice versa. We obtain a different 5-coloring of G' (why?) and now vertices w_3 and w_1 have the same color c_3, so v has no neighbor with color c_1. This means the new 5-coloring of G' can be extended to a 5-kleuring of G, by assigning v the color c_1.

$P_{1,3}$ contains a path $p_{1,3}$ that contains w_3: construct the path set $P_{2,5}$; as you can see in Figure 2.20, $P_{2,5}$ cannot contain a path that contains w_5; indeed, if such a path $p_{2,5}$ exists, then both $p_{1,3}$ and $p_{2,5}$ can be extended with the vertex v so that they form cycles in G; these cycles must intersect, but not in a vertex, because of the different colors. But we have started from a plane drawing of the graph, so no intersection is possible. This means that no path in $P_{2,5}$ arrives at w_5, and we can use the trick.

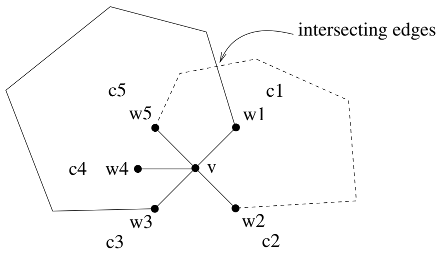


Figure 2.20: $p_{1,3}$ en $p_{2,5}$

■

Appel and Haken *proved* that every planar graph has a 4-coloring, so they *proved* the 4-color conjecture for planar maps.

This is a good moment to reflect on the concept of a *computer proof*. Whether something is accepted as a proof or not, is mainly a social given. It feels true that the mathematical correctness of a proof is independent of people agree with it, but a proof exists mainly by the grace of the community accepting the proof, even of the proof turns out to be incorrect later. So it is up to the community (only to the mathematical community?) to accept or reject computer proofs: one could imagine a branch of science that accepts only hand written proofs on paper, just as there is a branch of mathematics that accepts only constructive proofs, and no existential proofs. It is not possible to decide univocally whether to accept computer proofs just based on scientific arguments. But the fact is that the proofs are only possible by computer, the more we are inclined to accept such proofs, especially if the theorems so proven have useful consequences.

One could argue that it is in principle possible to check a computer proof by hand, or proof (by hand) that the computer program generating the proof is correct. That is currently not possible for larger programs. Moreover, one would also have to prove the correctness of the machine executing the program . . . something we might want to leave to another computer program?

Returning to the coloring of graphs: there are non-planar graphs with no 4-coloring, e.g. K_n with $n > 4$ needs at least n colors, and the same holds for graphs containing the clique K_n. But there exist also graphs no containing K_4 (or larger) with no 3-coloring. Finally, lots of non-planar graphs have a 4-coloring. More specifically, every bipartite graph (e.g. $K_{3,3}$) has a 2-coloring.

Strongly connected components: an algorithm by Tarjan

We abbreviate *strongly connected component* to SCC. Robert Endre Tarjan [3] - the master algorithmist - designed in 1972 a very nice algorithm that partitions the vertices of a digraph into SCCs. An SCC is a (maximal) set of vertices in which every node is connected to every other node of the same set. We first look at an application. Below is a program in pseudo-code, from which everything is stripped except the procedure call and definitions.

```
main()              venus()             aurochs()           tarpan()
{ venus();          { pluto();          { dodo();           { aurochs();
  aurochs();          venus();            tarpan();         }
}                   }                   }

                    pluto()             dodo()              mamoet()
                    { venus();          { tarpan();         { mamoet();
                    }                     mamoet();         }
                                        }
```

The *call graph* of this program can be seen in Figure 2.21: the vertices are the procedures; there is an directed edge (p_1, p_2) if and only if p_1 calls p_2. We have left out the self-calls.

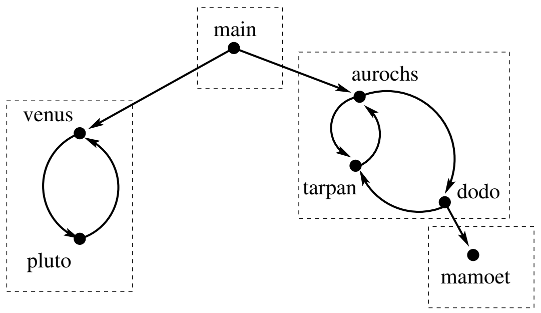


Figure 2.21: The directed call graph

SCCs partition the program in smaller parts of which the analysis can be performed independently, and/or sequentialized appropriately. The analysis could concern complexity,

[3]Turing award 1986

invariants (for correctness), data-flow, liveness analysis (for optimization) ... One finds uses of SCCs in very different contexts.

The SCCs are indicated in Figure 2.21 with a rectangle: each vertex belongs to exactly one SCC. Try understanding and proving this.

Algorithm 1 is an adapted version from Wikipedia[4]. Each vertex has 4 fields: the booleans *instack* and *visited* are initially FALSE; the integers *index* and *lowlink* are initialized in time. Their selection is denoted as e.g. $v.visited$.

Algorithm 1 Tarjan's algorithm computes the SCCs

```
function TARJAN
    index := 1;
    for all $v \in V \wedge (\neg v.visited)$ do
        strongconnect(v);
    end for
end function

function STRONGCONNECT(v)
    v.visited := TRUE;
    v.index := v.lowlink := index++;
    push(v); v.instack := TRUE;

    for all $(v, w) \in E$ do
        if $\neg w.visited$ then;
            strongconnect(w);
            v.lowlink := min(v.lowlink, w.lowlink);
        else if w.instack then
            v.lowlink := min(v.lowlink, w.index)
        end if
    end for
    if v.lowlink == v.index then
        start a new SCC
        repeat
            w := pop(); w.instack := FALSE;
            add w to current SCC
        until w == v
        output current SCC
    end if
end function
```

Give arguments why this algorithm ends and is correct.

[4] http://en.wikipedia.org/wiki/Tarjan's_strongly_connected_components_algorithm, 28-7-2014

Donald E. Knuth[5] says[6] that Tarjan's algorithm is his favorite one, also because it sorts topologically as a byproduct. Can you see that in the algorithm?

[5]Turing award in 1974

[6]**Twenty Questions for Donald Knuth** at `http://www.informit.com/articles/article.aspx?p=2213858`

Trees

Introduction

Definition 2.3.1. *Tree*
A **tree** is a simple graph with the property that there is a unique simple path between any two different points.

Any vertex of a tree could be designated as a root of the tree.

Definition 2.3.2. *Height of a vertex*
The **height of a vertex** v in a tree with root w, is the number of edges in the path from w to v.

Definition 2.3.3. *Height of a tree*
The **height of a tree** is the maximum of the heights of its vertices.

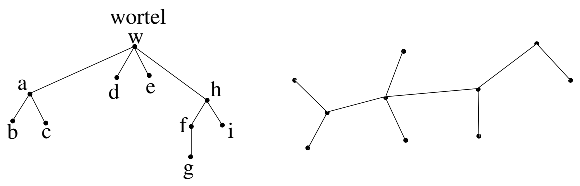


Figure 2.22: Examples of trees

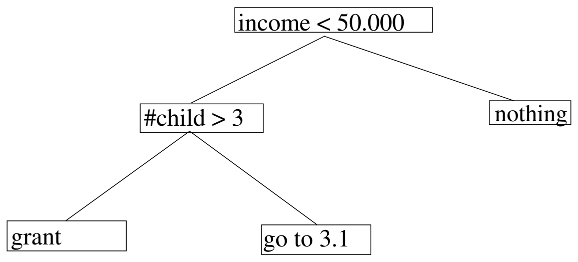
Figure 2.22 shows a tree with root w and a tree without a root. The heights of $a, b, c, d, e, f, g, h, i$ en w are 1,2,2,1,1,2,3,1,2 en 0.

Figure 2.23(a) shows a decision tree: each vertex contains a test. The idea is that you start at the root of the tree, and depending on the answer in the test, you move down left or right. In this way, you could get advice on whether you are eligible for financial support.

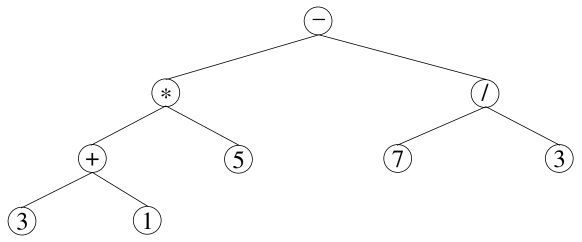
Figure 2.23(b) shows a tree representation of the arithmetic expression $(3 + 1) * 5 - 7/3$.

Properties of trees

The following theorem gives 4 equivalent characterisations of trees:



(a) Decision tree



(b) Representation of $(3 + 1) * 5 - 7/3$

Figure 2.23: Useful trees

Theorem 2.3.4. *For a simple graph T with n vertices $(a) \Leftrightarrow (b) \Leftrightarrow (c) \Leftrightarrow (d)$ with*

(a) T is a tree
(b) T is connected and cycle free
(c) T is connected and has exactly $(n-1)$ edges
(d) T is cycle free and has exactly $(n-1)$ edges

Proof. We prove that $(a) \Rightarrow (b)$, $(b) \Rightarrow (c)$, $(c) \Rightarrow (d)$ and $(d) \Rightarrow (a)$

- $(a) \Rightarrow (b)$: since T is a tree, there is a path between any ttwo vertices, so T is connected; if T contained cycle, some vertices would be connected by more than one path, so T is cycle free

- $(b) \Rightarrow (c)$: we prove this by induction on n that T has $(n-1)$ edges; for $n = 1$ T has zero edges, so the basis for the induction is true; suppose that every connected cycle free graph with n vertices has $(n-1)$ edges; consider a connected cycle free graph S with $(n+1)$ vertices; choose a simple path in P of maximal length in S (convince yourself this is possible); P contains a vertex v with $\delta(v) = 1$, because there are no cycles in S: indeed, when $\delta(v) > 1$ then one can extend P; consider $S \setminus v$ - the graph S from which v is removed; $S \setminus v$ has n vertices, is cycle free and connected, so by induction $S \setminus v$ has S_* $(n-1)$ edges; consequently, S has n edges

- $(c) \Rightarrow (d)$: suppose T contains a cycle; we can now remove at least one edge (and no vertices) from T to obtain a connected, cycle free graph T_*; so we can use the previous result (from $(b) \Rightarrow (c)$, meaning T_* has $(n-1)$ edges; this means that T has strictly more than $(n-1)$ edges, so we get a contradiction, from which we must conclude that T is cycle free

- $(d) \Rightarrow (a)$: we need to prove that T is connected, and that there is a unique path between every two vertices; consider the partition of T in its connected components $\{T_i\}_{i=1}^{k}$; let the number of vertices in T_i be equal to n_i; each of the T_i is connected and cycle free, so T_i has $(n_i - 1)$ edges; summing these we get: $(n-1) = \sum_{i=1}^{k}(n_i - 1) = \sum_{i=1}^{k}(n_i) - k = n - k$; from this it follows that $k = 1$, so T is connected; now suppose that there exist two different paths from a to b: those two paths form a cycle, but T is cycle free; we conclude that there is a unique path between every two nodes; so T is a tree

■

One reason why this theorem is important is this: to walk around in a general graph is dangerous, because you run the risk to run in circles (cycles), and your program could easily end up in an infinite loop. To prevent that, you need to do extra work, for instance keep track of all the nodes you have visited already. The theorem above tells us that in a tree, your walk will end if you always go *forward*, i.e. never go back to the (one) previous node. The reason is that a tree has no cycles. So, if you know that you are dealing with trees, you can and should use that fact in your program.

Definition 2.3.5. *Terminology related to trees*
For a tree T with root v_0, let x, y and z be vertices in T and $(v_0, v_1, \ldots, v_n)$ a path.

- v_{n-1} is the **parent** of v_n; we often say father (or mother)
- $v_0 \ldots v_{n-1}$ are all **ancestors** of v_n, and v_n is a **descendant** of $v_0 \ldots v_{n-1}$
- v_n is a **child** of v_{n-1}
- if x and y have the same parent, then x and y are siblings
- if x has no child, x is a **leaf**, also called **external node**
- if x is not a leaf, x is an **internal node**
- a subgraph of T with vertex x and all descendants of x, is a **subtree** of T rooted at x[7]

Relating this to Figure 2.24:

- c, d, e are leaves
- a is the parent of d
- b, c, f are descendants of a

[7]Actually, you should prove that this subgraph is a tree!

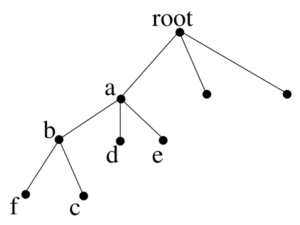


Figure 2.24: Example tree

- a and *root* are the ancestors of b
- b, d, e are siblings
- a, b are internal nodes
- the subtree rooted in a contains the vertices a, b, c, d, e, f

Often, the order of the siblings is important, especially with binary trees.

Definition 2.3.6. *Binary tree*
A **binary tree** is a rooted tree in which every node has 0, 1 or 2 children; because of the way binary trees are drawn and used, we speak of a left child (connected to its parent by the left branch) and a right kind (...); this makes an order on the branches explicit.

You already saw a binary tree in Figure 2.23(a): it is important to know whether to go left or right on a positive answer to the test in an internal node.

Definition 2.3.7. *Full binary tree*
A **full binary tree** is a binary tree in which every internal node has exactly two children.

Figure 2.25 shows a tournament tree: it is a full binary tree.

Theorem 2.3.8. *A full binary tree T with i internal vertices has T $(i + 1)$ leaves and $(2i + 1)$ vertices.*

Proof. Each internal vertex has 2 children, so there are $2i$ children in T; there is exactly one vertex that is not a child: the root; so there are $2i + 1$ vertices. Each vertex is either a leaf or internal, so the number of leaves equals $2i + 1 - i = i + 1$ ∎

In a tournament with n participants, and direct elimination, you could wonder how many

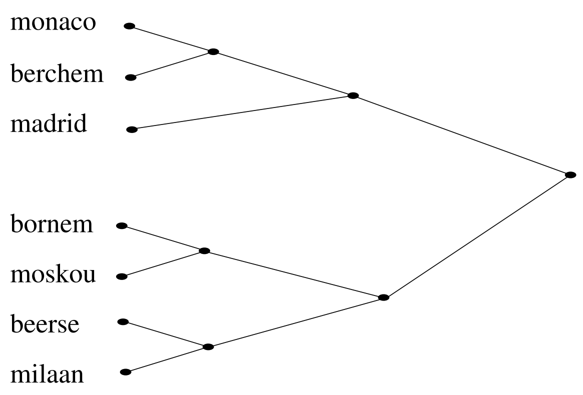


Figure 2.25: Tournament tree

matches must be played before the winner is known. Since the tournament tree has n leaves, the number of internal nodes is $n-1$, and that is the number of matches.

Theorem 2.3.9. *For a binary tree T with height h and t leaves $\log_2(t) \leq h$ (of $t \leq 2^h$).*

Proof. We prove the theorem by induction on h and by considering the subtrees of T: for $h = 0$, there is one leaf (the root), so $t = 1 = 2^h = 2^0$ this provides the base case.

Assume $h > 0$; consider T_l and T_r, the left and right subtree of the root of T. T_l or T_r can be empty, but not both. Assume T_r is empty: then we have $h_l = h - 1$, and by induction $t_l \leq 2^{h_l}$; since $t_l = t$ we get $t \leq 2^{h-1} \leq 2^h$.

If T_l is empty, we can reason analogously.

Suppose neither T_l nor T_r is empty; so $h_l \leq h - 1$ and by induction $t_l \leq 2^{h_l}$ and similarly $t_r \leq 2^{h_r}$, so $t = t_l + t_r \leq 2^{h_l} + 2^{h_r} \leq 2^{h-1} + 2^{h-1} = 2^h$. ■

Theorem 2.3.10. *For a binary tree T with n vertices and height h $\log_2(n+1) \leq h+1$*

Proof. Extend the tree T to a tree T' as follows:

- add a left and right branch to every leaf
- add a left branch to each internal node missing a left branch
- add a right branch to each internal node missing a right branch

T' has n internal vertices (all the vertices of T) and height $(h+1)$; T' is full and binary, so we can apply Theorem 2.3.8 and conclude that T' has $(n+1)$ leaves, and by Theorem 2.3.9 $lg(n+1) \leq h+1$. ■

Figure 2.26 shows a tree and its extension as defined in Theorem 2.3.10: the added vertices are drawn by a black rectangle. Note that if the original tree is already full, its extension is different from the original.

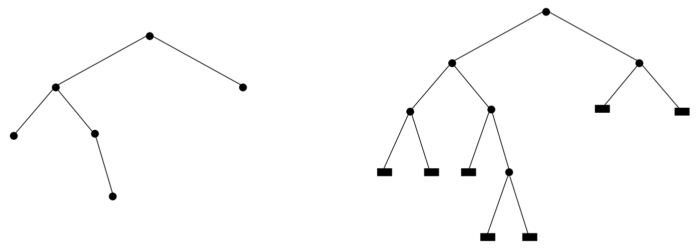


Figure 2.26: A binary tree and its extension to a full binary tree

Definition 2.3.11. *Binary search tree*
A **binary search tree** is a binary tree in which every vertex v has a value $w(v)$ (e.g. a number or a string) so that if l belongs to the left subtree of v and r to the right subtree of v, then $w(l) < w(v) < w(r)$

A binary search tree is also named a sorted binary tree.

Figure 2.27 shows a binary search tree in which the values of the vertices are words; the order is alphabetic.

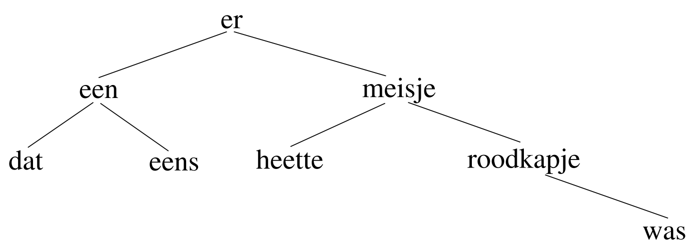


Figure 2.27: A binary search tree for the words from the sentence *Er was eens een meisje dat Roodkapje heette*

The next algorithm searches a given value in a binary search tree: the algorithm is written as a procedure returning TRUE if the value is found, and FALSE otherwise.

```
boolean search(T tree, W value);
{
    P = root(T);
    while (! empty(P))
    {
        if (value(P) == W)
                return(TRUE);
        else
        if (value(P) < W)
            P = rightchild(P);
        else P = leftchild(P);
    }
    return(FALSE);
}
```

The complexity of this algorithm can be expressed in the number of times the body of the while-loop is executed. In the worst case, the value is not in the tree, and we search along the longest path starting at the root: that path has length equal to the height h of the tree, so the loop is executed $h+1$ times. From Theorem 2.3.10 we know that $lg(n+1) \leq h+1$, so for fixed n, the worst case is not less than $lg(n+1)$. By balancing the tree as well as possible, we can achieve $\lceil lg(n+1) \rceil$. Figure 2.28 shows two binary search trees with the same values: the one at the right is balanced better than the one on the left, and has a smaller height.

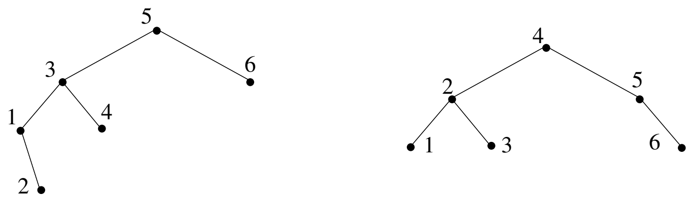
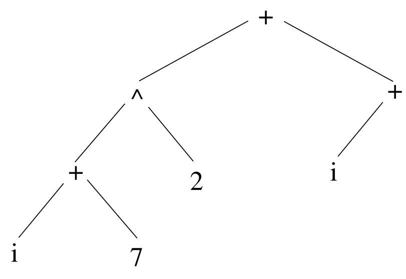


Figure 2.28: Two trees with the same values and different height

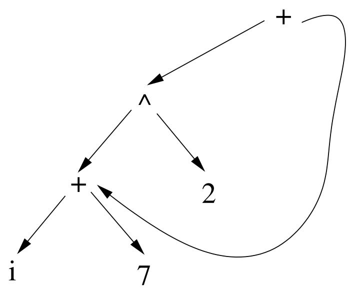
A more compact representation of (some) trees

A tree is often represented with *directed* edges. This representation stresses the fact that often the two functions *leftchild* en *rightchild* are explicitly available for use in a program, but not the function *parent*.

It can also be useful to represent a tree as a digraph without cycles that is not necessarily a tree. As an example take the tree representation of the expression $(i+7)^2+i+7$ in Figure 2.29(a). Two subtrees are equal, namely for the subexpression $i + 7$ that occurs twice. The corresponding more compact representation as digraph can be seen in Figure 2.29(b): the representation of $i + 7$ is now shared.



(a) Tree representation



(b) Graph representation with sharing

Figure 2.29: Two representations of $(i + 7)^2 + i + 7$

Sharing subtrees has its dangers: when the shared subtree is changed, *both* occurrences change. If that is not intended, one should not use the more compact representation.

Spanning trees

In this section, we only consider simple graphs: convince yourself (later) that this is not a severe restriction.

Definition 2.3.12. *Spanning tree*
T is a **spanning tree** of a graph G, if T is a subgraph of G containing all vertices of G.

A spanning tree covers all vertices of a graph, and it is the smallest connected subgraph doing so: if you take away an edge from a spanning tree, you end up with a subgraph that is not a tree.

Theorem 2.3.13. *A graph G has a spanning tree T if and only if G is connected.*

Proof.

- If G has a spanning tree T, then G is connected: a path between any two edges can use the edges of T.
- If G is connected and cycle free, then G is a tree and a spanning tree of itself.

  If G is connected and has a cycle, remove one edge from that cycle (but not the vertices); the resulting graph is still connected and has one cycle less than G; the theorem is now proved by induction on the number of cycles.

■

The above theorem does more than prove the existence of a spanning tree: it gives a constructive method (and algorithm) for finding a spanning tree: remove an edge from every cycle and you end up with a spanning tree. An algorithm based on this method is not very efficient, because we must find cycles and it might be possible to do better.

Theorem 2.3.14. *Characterisation of spanning tree*
A cycle free subgraph T of a connected graph G that contains a maximal number of edges of G, is a spanning tree of G.

Proof. Assume that G has at least 2 vertices.

The fact that T contains a maximal number of edges, means that that one can't add an edge without introducing a cycle.

We must prove two things: (1) T is connected (which makes it into a tree); (2) T covers all vertices (making it spanning).

(2) Suppose there exists a vertex $v \in G \backslash T$; since G is connected is, there exits an edge b arriving in v; $b \notin T$, since otherwise $v \in T$; since T is maximal, the graph $T \cup \{b\}$ contains a cycle that contains b. But this means that $v \in T$, which contradicts the assumption. So T contains all vertices of G.

(1) Suppose that T is not connected; consider the partition of its components $\{T_i\}_{i=1}^{n}$. Figure 2.30 shows T_1 and T_2. Since G itself is connected, there exist vertices $v_1 \in T_1$ and $v_2 \in T_2$ connected by a simple path P in G and this path has no other vertices in T_1 neither T_2 (see the dotted line in Figure 2.30). Suppose P has only one edge b. Then $T \cup \{b\}$ contains a cycle through v_1 and v_2; but that means there exists already a path from v_1 to v_2 in T, which contradicts the assumption that T_1 and T_2 are two different components.

We still need to prove that in a partition $\{V_k\}_{k=1}^{n}$ vertices of a connected graph G, there exist always a V_i and V_j so that there exist $v_i \in V_i$, $v_j \in V_j$, and an edge $(v_i, v_j) \in G$: consider V_1 and any other V_k; because G is connected, there is a path from a vertex $v_1 \in V_1$ to a vertex of V_k; we can choose v_1 so that the first edge arrives in a vertex $v \notin V_1$. If $v \in V_k$, then we are done. If not, v belongs to another V_j and with $j \neq 1$. ■

Here is an additional characterisation of spanning trees:

Property 2.3.15. *A subgraph of a simple connected graph G with n vertices, is a spanning tree of G id T is cycle free and has $n - 1$ edges.*

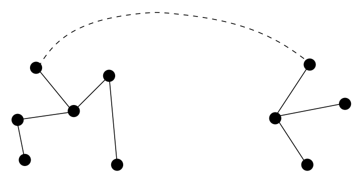


Figure 2.30: Two components of T connected by a path in G

Theorem 2.3.14 leads to a a general (non-deterministic) algorithm for constructing a spanning tree for a graph G: start with the empty tree T; repeatedly add an edge to T so that no cycle is introduced, until this is no longer possible: T is now a spanning tree.

Note that (1) each edge needs to be considered only once (why?); (2) the order in which the edges are considered does not matter; (3) eventually T is a tree, but at intermediate stages, T does not need to be connected: it is a forest; (4) one can stop trying to add edges as soon as T has $n-1$ edges.

Two orders for choosing edges are related to general strategies for tree traversal: *depth-first* and *breadth-first*. They are described informally here:

Depth-first construction of a spanning tree: choose a vertex and a construct a simple path starting at this vertex with maximal length: this path belongs to the spanning tree; back up one step along that path and start at that node constructing a maximal simple path; repeat backing up and constructing paths ... until you are back at the initial vertex and can't construct a path anymore; Figure 2.31 illustrates the method starting from the top vertex: the dotted edges belong to the spanning tree.

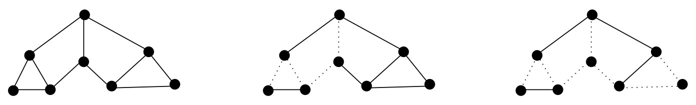


Figure 2.31: 3 phases in the depth-first construction of a spanning tree

Breadth-first construction of a spanning tree: choose a vertex and add all edges starting at this vertex make sure no cycle is made: these edges belong to the spanning tree; repeat this for all new vertices and keep repeating until no more edge can be added; Figure 2.32 illustrates the method starting from the top vertex.

The correctness of both methods results from the fact that all edges are eventually considered for adding to the growing tree, and on Theorem 2.3.14. In both methods, the intermediate graph is a tree, because it is connected and cycle free.

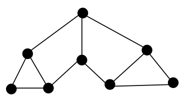


Figure 2.32: 3 phases in the breadth-first construction of a spanning tree

Figure 2.33 shows for K_4 that depth-first and breadth-first constructed spanning trees do not need to be isomorphic.

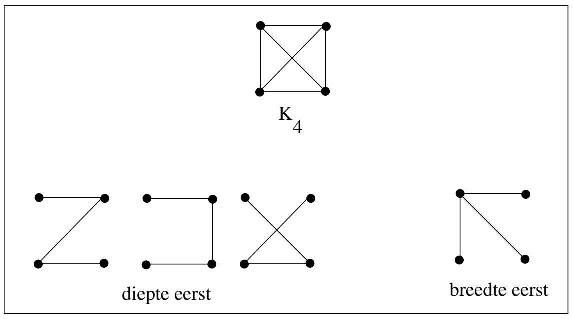


Figure 2.33: Spanning trees for K_4

Maybe you think that **all** spanning trees can be obtained by either method, but Figure 2.34 shows a graph with a spanning tree that neither of the methods can compute.

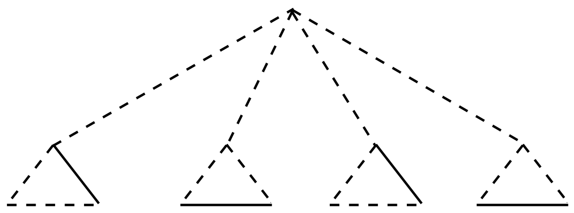


Figure 2.34: Graph with a hybrid spanning tree

Minimal spanning trees

Consider the following problem: there are a number of cities between which a road network must be build. The cost of building any road between two cities is known (it is always strictly positive). We want to choose which cities to connect by a road, while satisfying two criteria: (1) the total cost must be minimal, (2) each city must be reachable from any other city.

Clearly, the network has to be a tree because it can't have cycles (otherwise the cost would not be minimal), and it must be connected. This kind of trees is defined as:

Definition 2.3.16. *Minimal spanning tree (MST)*
T is a **minimal spanning tree** of a weighted graph G if T is a spanning tree of G with minimal weight.

Figure 2.35 shows a graph, one of its spanning trees, and its minimal spanning tree.

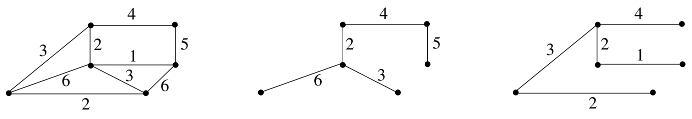


Figure 2.35: A graph with spanning trees

Can a graph have more than one MST? Does a graph have always at least one MST?

Efficient algorithms for constructing a MST are based on the following theorem:

Theorem 2.3.17. *An edge that belongs to an MST*
Let (V, E) be a connected graph (V, E), $U \subset V$ and $e \in E$ so that e has minimal length of all edges between U and $V \backslash U$. Then e belongs to some minimal spanning tree T of (V, E).

Proof. Let T_0 be any MST of (V, E). If $e \in T_0$, then we are done. If not, add e to T_0, resulting in T_1 which contains a cycle (Theorem 2.3.14). This cycle contains e and also another edge $e' = (u, v)$ so that $u \in U$ and $v \in V \backslash U$. Removing e' from T_1 results in a spanning tree T (whu is T a tree?) and moreover $w(T) \leq w(T_0)$ since $w(e) \leq w(e')$. So, T is a minimal spanning tree containing e. ■

Algorithm 2.3.18. *Prim*
Let $G(V,E)$ be a connected weighted graph $G(V,E)$ where the vertices are numbered: $V = \{v_1, v_2, \ldots, v_n\}$. The following procedure constructs a minimal spanning tree T for G.

1. ***Initialisation**: $T := (\{v_1\}, \emptyset)$*
2. ***Stop?**: If T has $(n-1)$ edges, stop.*
3. ***Add edge**: Define $S = \{e | e = (u,v), u \in T, v \notin T\}$ and choose an edge b from S so that $\forall e \in S : w(b) \leq w(e)$. Add b to T. The result is still connected and cycle free. Go to **Stop?**.*

Proof. At the end of of the algorithm T is a tree with n vertices edges $(n-1)$ edges, and cycle free, so T is a spanning tree G. We still need to prove termination of the algorithm and the minimality of T.

Termination is easy: **Add edge** is executed $(n-1)$ times, and then the algorithm stops. Maybe you should convince yourself that **Add edge** is always possible as long as T has less than $n-1$ edges.

Figure 2.36 shows the reasoning below.

After the **Initialisation**, T consists of one vertex, so at that moment T is part of a MOB. We now prove that this property is invariant under the application of **Add edge**:

denote by W the vertices of T and suppose that $T \subset T'$ with T' a MOB. Denote by B the set $\{(x,y) \mid x \in W,\ y \in V \backslash W,\ (x,y) \in E\}$. Consider a shortest edge (i,j) in B that does not cause a cycle when added to T. If $(i,j) \in T'$ then clearly $T \cup \{(i,j)\} \subseteq T' =$ MOB. If $(i,j) \notin T'$ then $T' \cup \{(i,j)\}$ contains a cycle which contains (i,j). This cycle contains another edge $(x,y) \in B$ and we can take it out of $T' \cup \{(i,j)\}$ so that we get a new spanning tree T'' (it contains all vertices and is connected and cycle free). So the question is: is T'' minimal? Since $w(i,j) \leq w(x,y)$ (that is how (i,j) was chosen), we have $w(T'') \leq w(T')$ so T'' is a MOB. We can conclude that $T \cup \{(i,j)\}$ is part of a MOB.

As a result, the spanning tree constructed by Prim is a minimal spanning tree. ■

Prim's algorithm is a classic example of a *greedy* algorithm: any time a choice needs to be made, the decision is made taking very little into account, and definitely not the future choices. Exactly because of Theorem 2.3.17, this greedy algorithm delivers the optimal solution. That is not true for every greedy algorithm: a shortest path algorithm that would choose at any junction the shortest edge leaving that junction, often does not give an overall shortest path; also, a greedy chess player looses more often than they win.

Prim's algorithm is illustrated in Figure 2.37: the initial vertex is A; the order in which the edges are added, is indicated as a,b,c . . . h.

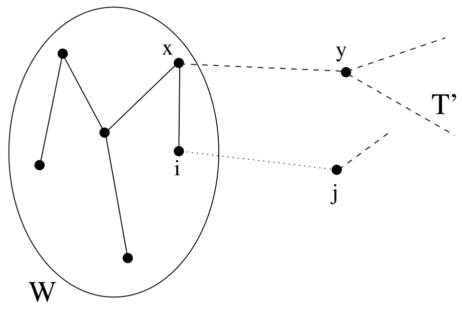


Figure 2.36: Illustration of the Algorithm 2.3.18

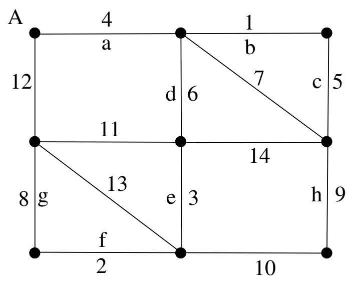


Figure 2.37: Prim's algorithm executed 2.3.18

Prim's algorithm builds the MOB incrementally: at each moment during the execution of the algorithm, T is a MOB of a subgraph of G. There is also a variant of Prim's algorithm without this property:

Algorithm 2.3.19. *Kruskal*
Let $G(V,E)$ be a connected weighted graph $G(V,E)$ where the vertices are numbered: $V = \{v_1, v_2, \ldots, v_n\}$. The following procedure constructs a minimal spanning tree T for G.

1. ***Initialisation**: $T := \emptyset$*
2. ***Stop?**: If T has $(n-1)$ edges, stop.*
3. ***Add edge**: Add to T an edge b with minimal weight that does not introduce a cycle in the result. Go to* **Stop?**.

Proof. Termination of Kruskal is proven as for Prim.

The proof that T is a spanning tree at the end, is also similar.

We prove minimality of T. Assume T is not a MOB. Name the edges of T $b_1, b_2, \ldots, b_{n-1}$ in the order as added by the algorithm. Let S be a MOB of G so that $\{b_1, b_2, \ldots, b_i\} \subseteq S$

and with maximal i (such an S exists and by Theorem 2.3.17 $i \geq 1$!). There are now two possibilities:

- $i = n - 1$: this means that $T = S$ so T is a MOB.
- $i < n - 1$: now $b_{i+1} \notin S$; let H be the graph with the edges $\{b_1, b_2, \ldots, b_i\}$ (and their vertices). Consider the graph $S \cup \{b_{i+1}\} = S'$. S' has a cycle with at least one edge b not belonging to T (because T is itself cycle free). So, $b \in S$. $H \cup \{b\}$ is cycle free, because $H \cup \{b\} \subseteq S$ and since Kruskal added edge b_{i+1} in the $(i+1)$-de **Add edge** (for addition to H), we know that $w(b_{i+1}) \leq w(b)$. Consequently $S' \backslash \{b\} = S''$ is a MOB. But that makes S'' into a MOB that contains $\{b_1, b_2, \ldots, b_{i+1}\}$, in contradiction with the maximality of S. It follows that $i < n - 1$ is not possible.

■

Note that during Kruskal, T does not need to be a tree all the time. The execution if Kruskal's algorithm is illustrated in Figure 2.38 on the same graph as in Figure 2.37.

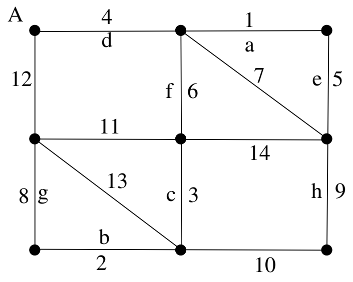


Figure 2.38: Kruskal's algorithm executed

Network models

A network of connections with each its own capacity can be modelled by a directed, weighted graph. The following are examples of such networks: a road network, an electrical network, oil pipes ... . An interesting and important problem in this area is the optimalization of the flow, while respecting the capacities. We will solve this optimization problem with some graph theory. Problems that are seemingly unrelated to flow optimization can often be modelled as a network problem as well: personnel assignment, resource allocation and even partner choice - the latter is know as the *marriage problem*.

Transport network

Definition 2.4.1. *Transport network*
A **transport network** (or simply **network**) is a simple, weighted, directed graph G with the following properties:

1. there is exactly one vertex in G without incoming edges; this vertex is called the **source**
2. there is exactly one vertex in G without outgoing edges; this vertex is called the **sink**
3. the weight $C_{i,j}$ of the (directed) edge (i, j) is positive and is named the **capacity** of the edge
4. disregarding the direction of the edges, G is connected

Figure 2.39 shows a network: the source is vertex a and the sink is z; the capacity of an edge is written next to the edge. The network could model a number of one way streets in a city with a railway station a and a market place z; the capacity could be the number of vehicles that can pass through the street per minute.

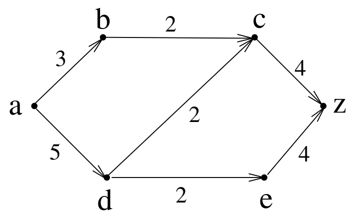


Figure 2.39: A transport network

The restriction to simple graphs is not so bad: loops and parallel edges can be simply removed by putting an extra vertex on such edges. This changes nothing essential as far as the problems studied later in this section is concerned.

Definition 2.4.2. *Network flow*
A **network flow** F in a network $G(V,E)$ with capacities $C_{i,j}, \quad i,j \in V$[8]is a mapping from E to $\mathbb{R}^+$ so that

1. $F(i,j) \leq C_{i,j}$

2. for each vertex j different from source and sink:

$$\sum_{i \in V} F(i,j) = \sum_{i \in V} F(j,i)$$

We name $F(i,j)$ the flow in edge (i,j). For a vertex j, we name the quantity $\sum_{i \in V} F(i,j)$ the incoming flow in j, and $\sum_{i \in V} F(j,i)$ the outgoing flow out of j.

Formula 2 in Definition 2.4.2 expresses the law of conservation of goods: everything that comes into an edge must get out as well. That prevents accumulation of goods in a vertex, and production of goods in a vertex. Think of Kirchoff's law.

Figure 2.40 shows a flow for the network of Figure 2.39; the flow is defined by:

$$\begin{aligned} F(a,b) &= 2 \\ F(b,c) &= 2 \\ F(c,z) &= 3 \\ F(a,d) &= 3 \\ F(d,c) &= 1 \\ F(d,e) &= 2 \\ F(e,z) &= 2 \end{aligned}$$

It is marked next to the capacity of the corresponding edge.

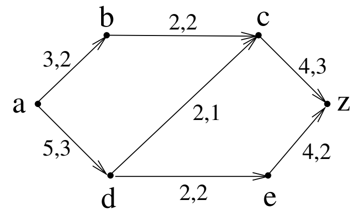


Figure 2.40: A transport network flow

[8]if there is no edge (i,j), we assume that $C_{i,j} = 0$

You can check that Formula 2 in Definition 2.4.2 is satisfied for each vertex, except the source and the sink.

You can also check that the flow coming out of the source equals the flow coming into the sink.

$$F(a, b) + F(a, d) = F(c, z) + F(e, z).$$

This equality is more generally true:

Theorem 2.4.3. *Source-out = sink-in*
For any flow F in a network $G(V, E)$, the flow coming out of the source equals the flow going into the sink, or more formally:

$$\sum_{i\in V} F(a, i) = \sum_{i\in V} F(i, z).$$

Proof. It should be clear that

$$\textstyle\sum_{j\in V}(\sum_{i\in V} F(i, j)) = \sum_{i\in V}(\sum_{j\in V} F(i, j)) = \sum_{j\in V}(\sum_{i\in V} F(j, i))$$

Interchanging the $\sum$'s is allowed as we are dealing with finite graphs. The second equality holds because we only renamed i and j.

It follows that

$$\begin{aligned} 0 &= \textstyle\sum_{j\in V}\left(\sum_{i\in V} F(i, j) - \sum_{i\in V} F(j, i)\right) \\ &= \left(\textstyle\sum_{i\in V} F(i, z) - \sum_{i\in V} F(z, i)\right) + \left(\sum_{i\in V} F(i, a) - \sum_{i\in V} F(a, i)\right) \\ &\qquad + \textstyle\sum_{j\in V\setminus\{a,z\}}\left(\sum_{i\in V} F(i, j) - \sum_{i\in V} F(j, i)\right) \\ &= \textstyle\sum_{i\in V} F(i, z) - \sum_{i\in V} F(a, i) \end{aligned}$$

since $F(z, i) = 0 = F(i, a)$ voor $\forall i \in V$ (by Definition 2.4.2)

and $\sum_{i\in V} F(i, j) = \sum_{i\in V} F(j, i)$ $\forall j \in (V\setminus\{a, z\})$ (by Formula 2 in Definition 2.4.2) ■

Theorem 2.4.3 justifies the following:

Definition 2.4.4. *Value of a flow*
The **value of a flow** F in a network $G(V, E)$ with source a and sink z is $\sum_{i\in V} F(a, i)$ or $\sum_{i\in V} F(i, z)$

The value of the flow in Figure 2.40 is 5.

The network problem can now be formulated as: for a given network G, find a maximal flow, i.e. a flow with maximal value.

Before we go into that ... networks could have more than one source and/or sink: the network in Figure 2.41 could represent the water supply of cities A and B, from sources X, Y and Z, and with intermediate distribution centers b, c and d.

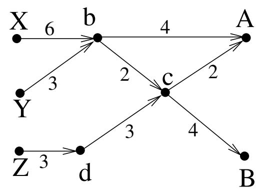


Figure 2.41: A transport network with more than one source and sink

One can add a super source a and a super sink z to the network, an edge from a to each original source with infinite capacity, and an edge from each sink to the super sink, also with infinite capacity: we get the usual network.

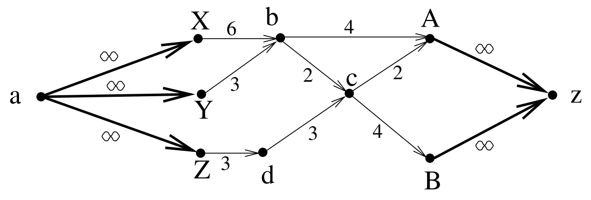


Figure 2.42: The same transport network with a super source and a super sink

A maximal flow in the new network corresponds to a maximal flow in the original network: convince yourself about that.

Finally, a maximal flow is often not unique: Figure 2.43 shows a network with infinitely many maximal flows. Indeed, for each i and j satisfying $i + j = 5$ en $0 \leq i, j \leq 3$, the flow is maximal.

Maximal flow

The idea behind the next algorithm for computing a maximal flow is as follows: start from any flow and improve it until no longer possible - you end up with a maximal flow. The informal description of how to improve a flow is:

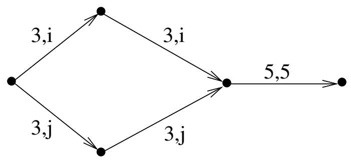
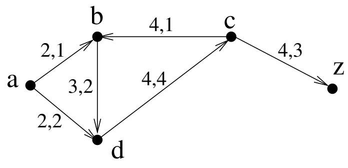


Figure 2.43: A network with infinitely many maximal flows

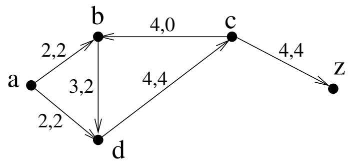
- consider a path P from the source a to the sink z
- find the minimum Δ of $C_b - F(b)$ over all edges $b \in P$
- construct the new flow along path P by adding Δ to each $F(b)$

A few remarks about this recipe:

- the operation increases the flow if and only if $\Delta > 0$ along that chosen path P
- the above only works if all edges in P have the correct (the *good*) direction, but
- limiting ourselves to paths from a to z along the directed edges is not enough: look at Figure 2.44(a); no path with only good edges can be improved with the above method; but the flow along the path (a, b, c, z) can be improved: the resulting flow is shown in Figure 2.44(b);



(a) A non-maximal flow



(b) A better flow along a path with a *bad* edge

Figure 2.44: Improving a flow

It follows that we also need to consider paths with inverted edges, and we don't add a Δ to them, we subtract it: if an inverted edge has a non-zero flow, it is possible that some flow is running in cycles without ever reaching the sink, while still using up the capacity of some edges. We formalize this now.

Definition 2.4.5. *Good and bad edges*
In a digraph $G(V,E)$ with path $(v_1, v_2, \ldots, v_n)$ we name an edge (v_i, v_{i+1}) **good** if $(v_i, v_{i+1}) \in E$, and otherwise **bad**

In a path P , we denote the good edges by P_+, the bad edges by P_-.

Theorem 2.4.6. *Improving a flow*
Let P be a path from a to z in a network $G(V,E)$, so that

1. $\forall (i,j) \in P_+$: $F(i,j) < C_{i,j}$
2. $\forall (i,j) \in P_-$: $0 < F(i,j)$

Define Δ as $min(min_{(i,j)\in P_+}\{C_{i,j} - F(i,j)\}, min_{(i,j)\in P_-}\{F(i,j)\})$, and the function F' as:

$$\begin{aligned} F'(i,j) &= F(i,j) \quad \forall (i,j) \notin P \\ &= F(i,j) + \Delta \quad \forall (i,j) \in P_+ \\ &= F(i,j) - \Delta \quad \forall (i,j) \in P_- \end{aligned}$$

The F' is a flow and it is (strictly) larger than F by Δ.

Proof. In order to prove that F' is a flow, we must check 1 and 2 from Definition 2.4.2: neither is difficult.

The fact that the flow is improved by Δ, follows from the fact that the flow over the edge $(a, \_) \in P$ (a good edge! why?) is increased by Δ, while the flow through the other edges starting at a has not changed. ■

One of the consequences of this theorem, is that in a maximal flow F each path from a to z contains at least one good edge with $C_{i,j} = F(i,j)$ or one bad edge with $F(i,j) = 0$. Otherwise, the flow can be improved along that path.

We would also like the following properties:

- if a flow cannot be improved along any path from a to z (according to the method of Theorem 2.4.6) then the flow is maximal - that might look trivial, but remains to be proven!
- an algorithm to maximize the flow consists of repeatedly looking for a path that satisfies the conditions of Theorem 2.4.6, and improving the flow over that path, until no more such paths exist; here we need a systematic way to find relevant paths

Both properties will be proven formally. We first describe a classical algorithm that is actually a *labeling procedure*: each vertex gets a label with information during the execution of the algorithm. We use a label with two pieces of information: the first component is *the vertex you came from*; the second component is the Δ from Theorem 2.4.6 in the path up to that vertex. The algorithm makes this more precise.

Algorithm 2.4.7. *Construction of a maximal flow.*
Let $G(V, E)$ be a network with source a, sink z and capacity C, and let all the capacities be a positive integer. Define any order on the vertices: $a = v_0, v_1, \ldots, v_n = z$.

We use $\mathcal{C}$ to denote the set of considered vertices, $\mathcal{L}$ for the labeled vertices. We will make the operations on $\mathcal{C}$ explicit and leave the operations on $\mathcal{L}$ implicit.

1. ***Initialisation**: Define $\forall (i, j) \in E : F(i, j) = 0$*
2. ***Label the source**: Give a the label $(\_, \infty)$; set $\mathcal{C} = \emptyset$*
3. ***Arrived?**: If z has a label, improve the flow and go back to **Label the source**.*

   Improving the flow works as follows: there is exactly one path P from a to z that can be found by constructed backward starting at z and following repeatedly the first component of the label of the vertex. The second component of the z's label is a quantity Δ that is added to the flow in the good edges of P, and subtracted from the flow in the bad edges. Finally, erase all labels in the network.
4. ***Choose the next vertex**: If $\mathcal{L} \setminus \mathcal{C} = \emptyset$, **stop**: the flow F is maximal.*

   Let v be the $v_i \notin \mathcal{C}$ with a label and with minimal index i.
5. ***Label the neighbours**: Suppose the label of v equals (α, Δ). Treat any edge of the form (v, w) or (w, v) in the order (v, v_0), (v_0, v), (v, v_1), (v_1, v), $\ldots$ and such that w has no label yet.*
   - *for an edge (v, w) (i.e. outgoing of v): if $F(v, w) < C_{v,w}$ than give w the label $(v, min\{\Delta, C_{v,w} - F(v, w)\})$ otherwise, do not label w*
   - *for an edge (w, v) (i.e. arriving in v): if $F(w, v) > 0$ than give w the label $(v, min\{\Delta, F(w, v)\})$ otherwise, do not label w*

   *Go to **Arrived?***

Proof.

- **Termination**:

  Point 3 in the algorithm is crucial: point 3 is repeated as long as the exit **stop** is not taken. From point 3, execution goes to point 2 or 4. The transition from point

3 to point 2 can occur only a finite number of times, as each transition implies that the flow improved with $\Delta > 0$; moreover, Δ is an integer. So after a finite number of transitions from point 3 to point 2, the transitions from point 3 must go to point 4, in which a vertex is added to $\mathcal{C}$ (it is never reset to $\emptyset$!). So, after a while $\mathcal{L} = \mathcal{C}$ and the algorithm stops.

- **Maximality**: we postpone the proof until after Theorem 2.4.11.

■

Note that not all maximal flows are found by Algorithm 2.4.7: see Figure 2.43.

Definition 2.4.8. *Cut in a network*
A **cut** in a network $G(V, E)$ with source a and sink z, is a tuple $(P, \overline{P})$ so that $a \in P$, $z \in \overline{P}$, $P \cup \overline{P} = V$ and $P \cap \overline{P} = \emptyset$

We can draw a cut by a line that separates the vertices of the network in two sets: in Figure 2.45, we have drawn the cut with a dotted line.

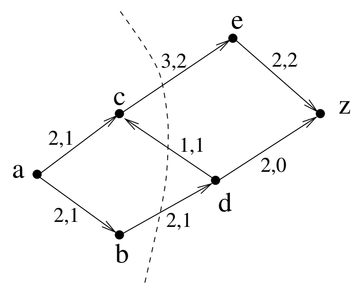


Figure 2.45: A network with a flow and a cut

We can check how much of the flow traverses the cut, i.e. from left to right in the figure. While doing that, an edge from P to $\overline{P}$ should be counted as contributing to that quantity, while the flow of the other edges must be subtracted. For the network in Figure 2.45 we get: $F(c, e) + F(b, d) - F(d, c) = 2 + 1 - 1 = 2$

Comparing this number with the flow (in a or z), we notice that these are also equal to 2! Coincidence?

We can also try to make an optimistic estimate of how much flow there could maximally be over a cut: maybe there exists a flow that uses all the capacity of the edges from P to $\overline{P}$, and nothing is flowing back over the other edged. We name this quantity the *capacity of the cut*. In the example, we get $C_{c,e} + C_{b,d} = 2 + 3 = 5$

The capacity of a cut seems larger than the flow over the cut, and larger than the maximal flow (you can check that the maximal flow in the example is 4). Is this a coincidence? We study this more formally.

Definition 2.4.9. *Capacity $C(P,\overline{P})$ of a cut*
The **capacity of a cut** $(P,\overline{P})$ is $C(P,\overline{P}) = \sum_{i\in P}\sum_{j\in\overline{P}} C_{i,j}$

Theorem 2.4.10. *The capacity of a cut is not smaller than a flow, i.e.*

$$\sum_{i\in P}\sum_{j\in\overline{P}} C_{i,j} \geq \sum_{i\in V} F(a,i).$$

Proof.

$$\begin{aligned}
\sum_{i\in V} F(a,i) &= \sum_{i\in V}(F(a,i)-F(i,a)) + \sum_{j\in P\setminus\{a\}}\sum_{i\in V}(F(j,i)-F(i,j)) \\
&= \sum_{j\in P}\sum_{i\in V}(F(j,i)-F(i,j)) \\
&= \sum_{j\in P}\sum_{i\in P} F(j,i) + \sum_{j\in P}\sum_{i\in\overline{P}} F(j,i) - \sum_{j\in P}\sum_{i\in P} F(i,j) - \sum_{j\in P}\sum_{i\in\overline{P}} F(i,j) \\
&= \sum_{j\in P}\sum_{i\in\overline{P}} F(j,i) - \sum_{j\in P}\sum_{i\in\overline{P}} F(i,j) \\
&\leq \sum_{j\in P}\sum_{i\in\overline{P}} F(j,i) \\
&\leq C(P,\overline{P})
\end{aligned}$$

■

Note that from the last but two lines, you see that the flow equals the flow over the cut.

A **minimal** cut is a cut with minimal capacity. The previous theorem implies directly that any maximal flow is smaller than or equal to the capacity of the minimal cut. The connection between the two is even stronger:

Theorem 2.4.11. *Max flow - min cut*
Let $(P, \overline{P})$ be a cut and F a flow in a network $G(V, E)$, if

$$C(P, \overline{P}) = F \quad \textit{(or written out } \sum_{i \in P} \sum_{j \in \overline{P}} C_{i,j} = \sum_{i \in V} F(a, i) \textit{)}$$

then F is maximal and the cut is minimal.
Moreover, the equality is equivalent with

1. *$\forall i \in P, j \in \overline{P} : F(i, j) = C_{i,j}$ and*
2. *$\forall i \in \overline{P}, j \in P : F(i, j) = 0$*

i.e. all good edges in the cut have a flow equal to their capacity and all bad edges have zero flow.

Proof. All this follows from the inequalities in Theorem 2.4.10 ■

Note that we have not proven that for each maximal flow there is a minimal cut, neither the other way around.

Figure 2.46 shows a maximal flow with its minimal cut.

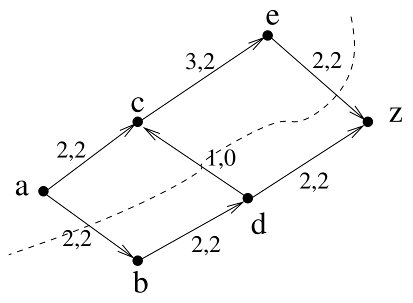


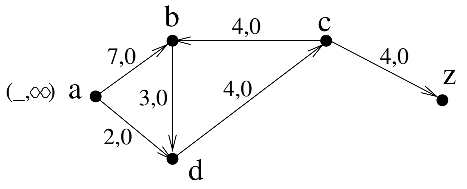
Figure 2.46: A network with a maximal flow and its minimal cut

We are now ready to prove that Algorithm 2.4.7 ends with a maximal flow.

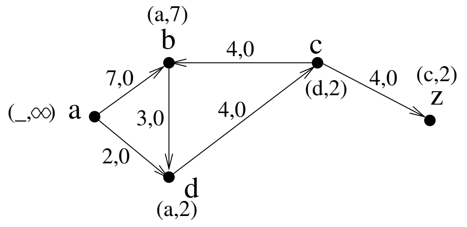
Proof. (of Theorem 2.4.7)
At the moment the algorithm ends, let P be the set of vertices with a label (note: $a \in P$) and $\overline{P}$ the set of vertices without a label ($z \in \overline{P}$). $(P, \overline{P})$ is a cut. Consider an (directed) edge (i, j) with $i \in P$ and $j \in \overline{P}$. Since i has a label $F(i, j)$ must be equal to $C_{i,j}$, otherwise j would have gotten a label. Now consider and edge (j, i) with $i \in P$ and $j \in \overline{P}$. Since i

has a label, $F(j,i)$ must be equal to 0, otherwise j would have gotten a label. Together with Theorem 2.4.11, we derive that F is maximal. ■

You see that the algorithm constructs at the same time a maximal flow and a minimal cut. Some of the steps in Algorithm 2.4.7 are shown in the Figure 2.47(a) and 2.47(b):



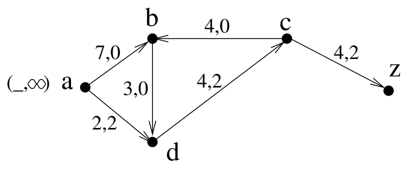
(a) After initialisation and labeling source a



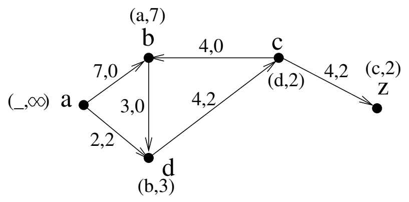
(b) After a labeling that reached z

Figure 2.47: Illustration 1 for Algorithm 2.4.7

Choose the order on the vertices as (a, c, d, b, z). Starting from a, d and b get a label. Then c is labeled from d; b is not labeled from d because b has already a label. Then z is labeled from c: we have found a path from a to z as you can see in Figure 2.47(b). The flow is improved and the result is in Figure 2.48(a).



(a) After improving the flow the first time

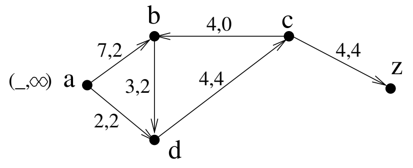


(b) A second labeling has reached z

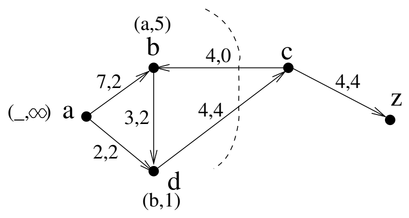
Figure 2.48: Illustration 2 for Algorithm 2.4.7

The labeling restarts in the situation ofFigure 2.48(a). From a, d is not labeled now: the flow on edge (a,d) is already maximal. c does not get a label from b, because the edge (c,b) is *bad* and its flow is zero, so it cannot be decreased. But d gets a label from b, and since the capacity of edge (b,d) (3) is smaller than the Δ of b (7 at that moment), we give d a $\Delta = 3$. c gets a label from d and since the unused capacity (d,c) equals 2, we give c

as Δ 2. From c, only z can be labeled: the resulting situation is in Figure 2.48(b). The flow can be improved along the obtained path: the result is in Figure 2.49(a).



(a) After improving the flow for the second time



(b) The labeling does not reach z: the cut is minimal, the flow is maximal

Figure 2.49: Illustration 3 for Algorithm 2.4.7

At this moment, the flow is already maximal, but the algorithm has not found that out: another attempt to construct a path from a to z is needed. The last round of labeling start in the situation in Figure 2.49(a): from a we can label b (the capacity of the edge (a, d) is already fully used) and also edge (b, d) has some spare capacity, but then we are stuck: the labeling stops at Figure 2.49(b). The minimal cut is defined by $P = \{a, b, d\}$ and $\overline{P} = \{c, z\}$.

Run the algorithm (manually) for the networks in in Figure 2.50, with order (a, b, c, d, e, z) on the vertices.

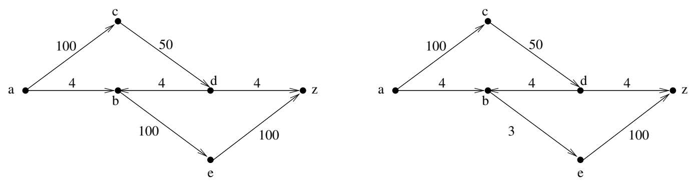


Figure 2.50: Two networks for practising

Finding the maximal flow in a network is equivalent to finding (integer) values $F(i, j)$ so that

- $\sum_j F(a, j)$ is maximal
- $0 \leq F(i, j) \leq C_{i,j}$ for the given $C_{i,j}$
- $\forall j : \sum_i F(i, j) = \sum_i F(j, i)$

This type of problems can also be solved by linear programming, e.g. by the simplex algorithm.

Matching

Consider the following problem: 4 students (A, B, C and D) want to make a separate appointment with an assistant who is available on 5 different days (a, b, c, d and e). Each student has their own preference for one or more of the days. The assistant must now decide who comes when. Clearly, this is not always possible: e.g. if both A and B have as their only preference day a, then it is impossible. Even if it is possible, it is not trivial to solve such a problem, especially not when there are many more students and days involved.

At first sight, this problem is unrelated to graphs, but we can represent it as a graph G and work from there: the students and the days play the role of the vertices of G. We draw a directed edge between a student and a day, if that student has included that day in their preference list. As an example: A prefers $\{b, c\}$, B prefers $\{a, b\}$, $C\{d, e\}$ and D $\{b, c, e\}$. We get the graph in Figure 2.51

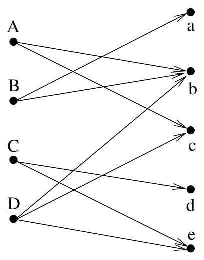


Figure 2.51: The graph for the matching problem

This graph is bipartite and directed. The connection with the previous section becomes clear if we add a source and a sink, and give each edge 1 as capacity: see Figure 2.52.

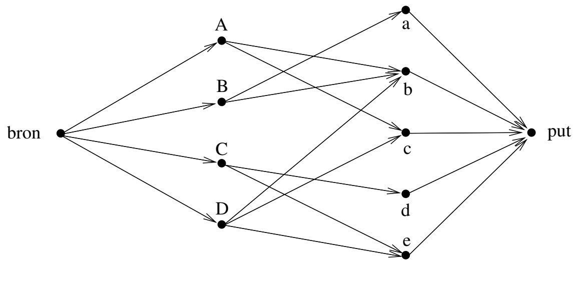


Figure 2.52: Network for the matching problem: all edges have capacity = 1

We can now construct a maximal (integer) flow F in in this network: the value of this maximal flow is 4 (the number of students). We can derive an assignment of students to days (or a matching of the students with the days) by interpreting an edge (s, d) so that $F((s, d)) = 1$ as: student s can see the assistant on day d. Figure 2.53 shows a maximal flow with dotted lines. We can read out from it the assignment (A, b), (B, a), (C, d), (D, e). The assignment is complete, and we have a complete (or perfect) matching, since every student is matched up with a day.

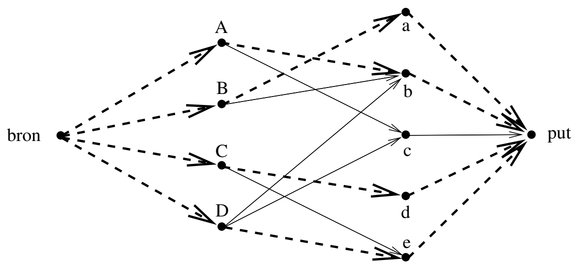


Figure 2.53: A solution for the assignment problem

The maximal flow could have been smaller than the number of students (for a different set of preferences of course): that maximal flow gives us a maximal matching, but not a complete one. More formally:

Definition 2.4.12. Let $G(V \cup W, E)$ be a directed, bipartite graph $G(V \cup W, E)$ with $V \cap W = \emptyset$ and $E \subseteq V \times W$, then M is a **matching** if

- $M \subseteq E$ and
- $\forall (x, y),\ (i, j) \in M$: if $(i, j) \neq (x, y)$ then $i \neq x$ and $j \neq y$ (i.e. at most one edge arrives and leaves any vertex)

A **maximal** matching has a maximal number of edges in M. A matching is **complete** if $\forall v \in V,\ \exists w \in W :\ (v, w) \in M$.

A directed, bipartite graph $G(V \cup W, E)$ to which a source and sink are added, edges from the source all vertices in V and edges from the vertices in W to the sink, and with capacity one for every edge, is named the **matching network** derived from G.

The next theorem formalises the correspondence between a a maximal flow in a matching network and a maximal matching.

Theorem 2.4.13. *Let $G(V \cup W, E)$ be a directed, bipartite graph $G(V \cup W, E)$ with $V \cap W = \emptyset$ en $E \subseteq V \times W$, then the following is true:*

- *An integer flow in F in the corresponding matching network, gives a matching in G: $v \in V$ corresponds to $w \in W$ if and only if $F(v, w) = 1$*
- *A maximal integer flow corresponds to a maximal matching.*
- *An integer flow with value $\#V$ corresponds to a perfect matching.*

Proof. You can work out the proof yourself. ■

It is sometimes possible to quickly test that no perfect matching exists: clearly, when n students together prefer strictly less than n days, they can all get one of their preferred days. We generalize this:

- define the mapping [9]

$$\begin{array}{rccl} R: & \mathcal{P}(V) & \rightarrow & \mathcal{P}(W) \\ & S & \mapsto & \{w \in W \mid \exists v \in S \text{ with } (v, w) \in E\} \end{array}$$

- If G has a perfect match, then $\#S \leq \#R(S)$ holds $\forall S \subseteq V$

The English mathematician Philip Hall proved the converse in 1935:

Theorem 2.4.14. *Hall's marriage theorem*

A directed, bipartite graph $G(V \cup W, E)$ with $V \cap W = \emptyset$ and $E \subseteq V \times W$, has a perfect match if and only if $\#S \leq \#R(S)$, $\forall S \subseteq V$

Proof. We have already argued one direction: if G has a perfect match, then $\#S \leq \#R(S), \forall S \subseteq V$. Now the converse:

Let $m = |V|$. We prove by induction on m that the condition is sufficient. The base case is $m = 1$: it is clear that a perfect match exists. So assume $m \geq 2$. We distinguish between two cases:

1. Suppose that for all S such that $\emptyset \neq S \subsetneq V$ it is true that $|S| + 1 \leq |R(S)|$. Take any edge (v, w) met $v \in V$ and consider the $G' = G \setminus \{v, w\}$. G' fulfills the conditions of Hall and V' is strictly smaller than m: by the induction hypothesis there exists a perfect matching of G'. Add the edge (v, w) to it and obtain a perfect match of G.

[9] $\mathcal{P}(S)$ denotes the power set of S

2. Now assume the existence of a set S with $\emptyset \neq S \subsetneq V$ and $|S| = |R(S)|$. Consider the subgraph G_1 induced by the vertices $S \cup R(S)$, and the subgraph G_2 induced by $(V \setminus S) \cup (W \setminus R(S)$. Now prove that G_1 and G_2 fulfill the conditions of Hall, so they both have a perfect matching. Take the union of these perfect matchings to obtain a perfect matching of G. Figure 2.54 illustrates the construction.

■

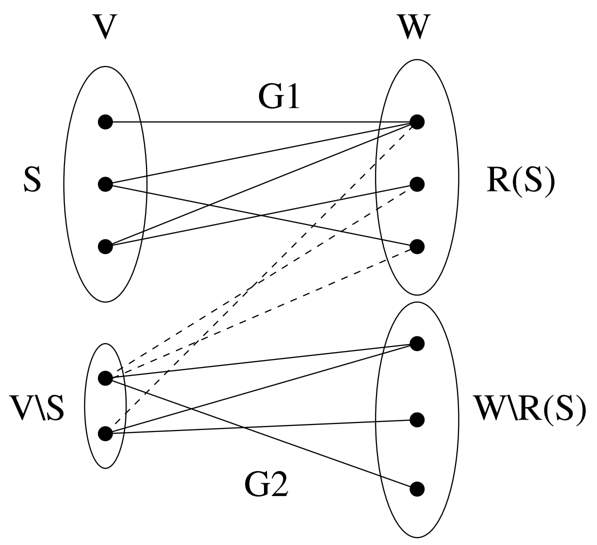


Figure 2.54: The dashed lines belong to the original graph, but not to G_1 and G_2

Hall's Theorem 2.4.14 can be used in partner choice problems, hence its name.

References

- Richard Johnsonbaugh "Discrete Mathematics", MacMillan, 1984
- Shimon Even "Graph Algorithms", Pitman, 1979
- Ralph P. Grimaldi "Discrete and Combinatorial Mathematics"
- William Barnier, Jean B. Chan "Discrete Mathematics"
- Michael Townsend "Discrete Mathematics: Applied combinatorics and graph theory"

Chapter 3

Languages and Automata

Introduction

Computer programs are central in the activity of the computer scientist. Computer programs transform a given input into a desired output. The input could be an electric pulse indicating a temperature change, a character from the keyboard, another program, a database ... The output could be a sequence of signals controlling some valves, a new version of a document, an error message, the display of a graph ...

From a practical point of view, the above examples are quite different, but when we focus too much on the details, we miss a lot. We can abstract away many details and focus on the essence: in every example, there is a mapping from elements in an input domain to elements in an output domain. These domains are always discrete, and sometimes infinite. This means that from a theoretical point of view, programs are just functions from (a part of) $\mathbb{N}$ to (a part of) $\mathbb{N}$.

In case the input domain is small, that function is not very interesting: one could just build a table with all precomputed values, and then the computation is just a simple lookup.[1]

In case the input domain is large, or not a priori known, then it is impossible to tabulate the function, so it is better to assume the input domain is infinite.

Now about the output domain: it needs to contain at least two values to be interesting. These two elements could be *yes, no* and that is enough to describe *decision problems*: you want to know whether a given input has a certain property, e.g. *is this program syntactically correct?* or *is this a good day to invest time in FCW?* [2] ...

[1] Tabulation is an interesting dynamic implementation technique, whichever the programming paradigm. Ask if you want to know more.

[2] Yes !

If the output domain contains more than two elements, but a finite number of them, then one can easily compute the correct output by a finite sequence of yes/no questions. Suppose you want to know which is the best day of the week for investing time in FCW (the output domain has 7 values), you could ask the following yes/no questions *is monday the best day for investing time in FCW?*, *is tuesday the best day for investing time in FCW?* ... So, it seems that of all finite output domains, the only really interesting one has exactly two values.

If the output domain is infinite, the function can not be reduced easily to a finite sequence of functions with a finite output domain.

We can conclude that there exist only two kinds of interesting functions. They have signature $\mathbb{N} \to \{yes, no\}$ and $\mathbb{N} \to \mathbb{N}$.

We focus for now on the first kind: it is easy to associate a certain structure and semantics with the output domain, but the input domain is very generic. $\mathbb{N}$ could indeed be replaced by any other countably infinite set. Moreover, when we note down elements of $\mathbb{N}$, we can use decimal digits: we form strings (sequences) of such digits. We could have used another basis for the representation - binary, hexadecimal, or even more exotic excess-3, ... These representations have something in common: a finite set of symbols is used (we name it an alphabet), a sequence of such symbols (we name it a string) has a unique meaning, and one can concatenate strings. Let Σ denote a finite alphabet, and let Σ^* denote all finite sequences of symbols from Σ. Then we have just argued that the interesting functions have signature $\Sigma^* \to \{yes, no\}$. A function F in this class is totally determined by the set of strings s for which $F(s) = yes$, so instead of studying such functions, we can just as well study subsets V of Σ^* together with the associated question: *does a given s belong to V*. It is important to understand that we are still considering the same class of functions $\mathbb{N} \to \{yes, no\}$, but also that it is often more natural and insightful to describe decision problems starting from a particular Σ^*. Almost all of computability theory uses this framework.

The second kind of functions - the ones with signature $\mathbb{N} \to \mathbb{N}$ - are of course very important as well, but will not be the focus in this course.

Selfie:

> The introduction simplifies matters, and makes assumptions that you might disagree with, or that require a bit more argument ... Find *holes* in the introduction, think of alternatives, argue in favor and against ...
>
> Describe a decision problem already known to you, but now by choosing an alphabet Σ and defining a subset *Sol* of Σ^* that is the solution to the problem. Do that for different kinds of problems: with numbers, words, graphs, ... Make sure to construct an example in which *Sol* is finite and one in which it is infinite. What happens if you interchange *yes* and *no*?

What is a language?

The introduction gives the motivation for studying decision problems. A decision problem corresponds to a subset of all strings (or words) consisting of elements of a finite alphabet Σ. Any such subset is a *language* over Σ.

Definition 3.2.1. *String over an alphabet* Σ
A **string** over an alphabet Σ is a finite sequence of zero, one or more elements of Σ

Clearly, if we put two strings x and y one after the other, we get a new string denoted xy.

Definition 3.2.2. *Language over an alphabet* Σ
A **language** over an alphabet Σ is a set of (finite) strings over Σ

A language can be finite or infinite. If the language L is infinite, is it countably infinite?[3]

In order to fix a language, one must give a description for every element of the language, and that description should not fit any string not in the language. E.g. the language of odd numbers (over the alphabet of decimal digits), or the set of strings ending on 1,3,5,7,9. Another example: the English words rhyming with automata.

The description of a language is preferably finite, even if the language itself is infinite.

The description of the language can *often* be used to check whether a given string belongs to the language, and even to generate or construct each element of the language. But be careful with the *often*: we will need to get back to it.

If the description of a language is simple, we expect the language to be simple, but we don't really have a good idea what *simple* means here.

Here are some questions for you to think about: this course provides you with material to answer them.

- Do formalisms exist that can denote language descriptions?
- Does such a formalism allow to derive (automatically) testers and generators for the language?
- Are there languages that do not fit the formalism?
- What is a good notion of testing/generating?
- Are some languages inherently easier to describe, test, generate?

[3]If you do not remember what that means, now is the time to look it up

And finally:

Why should an computer scientist know all this?

Selfie:

Can you use the description of the set of even numbers to generate all even numbers?

Same question but for the English words rhyming with automata.

Can you use them for testing?

Invent a new very extravagant language, give its description and use it to test whether a given string belongs to it, and use it to generate all strings belonging to the language.

Can you see from the examples a connection between the ease of description, the testing and the generation?

An algebra of languages

An algebra (or algebraic structure) is a set with some internal operations, often binary operations, but also unary operations are common, and even opreations with larger arity. The set of languages over the alphabet Σ becomes an algebra by defining operations like union, intersection, complement ... More concrete: let L_1 and L_2 be two languages, then

- their union is a language: $L_1 \cup L_2$
- their intersection is a language: $L_1 \cap L_2$
- the complement is a language: $\overline{L_1}$

You can construct other operations on languages by composing the three above, just like with any sets.

There is also a new way - not directly linked to set theory - to make a new language out of two given languages: *concatenation.*

Definition 3.3.1. *Concatenation of two languages*
Let L_1 and L_2 be two languages over the same alphabet Σ, then we denote the concatenation of L_1 and L_2 by L_1L_2 and we define: $L_1L_2 = \{xy|x \in L_1, y \in L_2\}$

When we concatenate three languages, we do not need brackets, because clearly, concatenation is associative: $(L_1L_2)L_3 = L_1(L_2L_3)$

Concatenating L n times with itself is denoted by L^n. L^0 contains only the empty string, and we denote that string by ϵ.

Finally, we also define an operation that allows to construct an infinite language from a finite one:

Definition 3.3.2. L^* *- the Kleene* **star** *of* L

$$L^* = \cup_{n=0}^{\infty} L^n$$

As an abbreviation of LL^* one uses L^+.

Now we have all this notation, we can say that L is a language over Σ if $L \subseteq \Sigma^*$, or equivalently $L \in \mathcal{P}(\Sigma^*)$.

Selfie: Define new operations on languages.

Language descriptions

In maths, one uses the set notation, e.g.

Example 3.4.1. *Let* $\Sigma = \{x, y, z\}$*:*

- $\Sigma^* = \{a_1a_2a_3...a_n |\ a_i \in \Sigma, n \in \mathbb{N}\}$
- $L = \{a_1a_2a_3...a_n |\ a_1 = y, n \in \mathbb{N}, \forall i > 1 : a_i \in \Sigma\}$
  or in words: L *is the set of strings starting with* y

Informal descriptions are possible as well, as long as they are not ambiguous about membership

Example 3.4.2.

- $H(n) = \{P | P\ is\ a\ Java\ program\ and\ P\ stops\ after\ at\ most\ \ n\ seconds\ on\ input\ n\}$
- $Prime = \{n | n \in \mathbb{N}, n\ is\ prime\}$

While the last one is an informal description, it hides the formal description of *prime*, so we could also have specified $Prime$ as

$$Prime = \{n | n \in \mathbb{N}, \forall i : 1 < i < n \rightarrow n\ mod\ i \neq 0\}$$

This kind of description is ok for doing maths, but computer scientists need a different formalism: it should allow (more easily) to convert a description into a generator and a tester. For natural languages, linguists have grammars: they describe the structure of a sentence. Natural languages are very complicated, with lots of exceptions to rules, and exceptions to exceptions ... It makes sense to study classes of languages with a simpler structure. We aim therefore at

a hierarchy of languages

an accompanying hierarchy of description formalisms, or grammars

an accompanying hierarchy of*test* and *generate* procedures

The hierarchy we study is named the *Chomsky-hierarchy*[4]. We start at the bottom of this hierarchy, i.e. with the easiest languages.

[4]Noam Chomsky

Regular expressions en regular languages

Definition 3.5.1. *Regular Expression (RE) over an alphabet* Σ
E is a **regular expression over alphabet** Σ if E has the form

- ϵ
- ϕ
- a for $a \in \Sigma$
- (E_1E_2) for regular expressions E_1 and E_2 over Σ
- $(E_1)^*$ regular expression E_1 over Σ
- $(E_1|E_2)$ for regular expressions E_1 and E_2 over Σ

In this way, the set set of regular expressions *RegExps* (over an alphabet Σ) is defined inductively. It implies that something that does not complies to the definition, is not a regular expression, and that every regular expression can be constructed using only the rules above. It is often implicitly clear which alphabet is used, and we often do not mention it. We use the brackets to avoid ambiguity: the brackets to not belong to the alphabet. In the next examples, $\Sigma = \{a, b, c\}$.

Example 3.5.2. *Regular expressions over* Σ*:*

- b
- $a(\epsilon c)$
- $(((ab))^*c \mid (bc))$

Are we using too many brackets, too few? Why?

Definition 3.5.3. *A regular expression E* **determines** *a language L_E over its alphabet Σas follows:*

- if E = a ($a \in \Sigma$) then $L_E = \{a\}$ (the langusge with just one string, consisting of the character a)
- if E = ϵ then $L_E = \{\epsilon\}$ (the empty string)
- if E = ϕ then $L_E = \emptyset$ (the empty set)
- if E = (E_1E_2) then $L_E = L_{E_1}L_{E_2}$
- if E = $(E_1)^*$ then $L_E = L_{E_1}^*$
- if E = $(E_1|E_2)$ then $L_E = L_{E_1} \cup L_{E_2}$

This similarity between the structure of the regular expressions and the structure of the languages they determine is not a coincidence, and we can exploit it to use less brackets in regular expressions. We just agree that * takes precedence over concatenation, which takes precedence over union. Note also that concatenation is associative. So we can agree that $L_{(((a)^*(b))c)} = L_{a^*bc}$. You should however never say that the regular expression $(((a)^*(b))c)$ equals the regular expression a^*bc: the correct wording is *the language (determined by)* $(((a)^*(b))c)$ *equals the language (determined by)* a^*bc.

Selfie: Formulate a verbal description of the languages determined by the REs over $\{a, b\}$ below - the first one is an example:

$(ab)^*$: every a is followed immediately by a b, and there are as many a's as b's

$(aba)^*$

$(a|b)^*$

$(a|b)^*\phi$

$a\epsilon b$

Selfie: Prove the following statements, or give a counterexample:

if a regular expression E does not contain a *, then L_E is finite

if a regular expression E contains a *, then L_E is infinite

$L_E \subseteq L_{(E|F)}$ for all RE's E and F

the set of regular expressions (over a given alphabet) is itself a language (over which alphabet?)

Definition 3.5.4. *Regular Language*
A language determined by a regular expression is named a **regular language**.

We denote the set of regular languages by $RegLan$. Check which of the formulas below mae sense, and which are true.

1. $RegLan \subseteq \Sigma$
2. $RegLan \subseteq \Sigma^*$
3. $RegLan \subseteq \mathcal{P}(\Sigma)$
4. $RegLan \subseteq \mathcal{P}(\Sigma^*)$
5. $RegLan \subseteq \mathcal{P}(\mathcal{P}(\Sigma^*))$
6. if $x \in RegLan$ then $x \in \Sigma$
7. if $x \in RegLan$ then $x \in \Sigma^*$
8. if $x \in RegLan$ then $x \in \mathcal{P}(\Sigma)$
9. if $x \in RegLan$ then $x \in \mathcal{P}(\Sigma^*)$
10. if $x \in RegLan$ and $y \in x$ then $y \in \Sigma$
11. if $x \in RegLan$ and $y \in x$ then $y \in \Sigma^*$
12. if $x \in RegLan$ and $y \in x$ then $y \in \mathcal{P}(\Sigma)$
13. if $x \in RegLan$ and $y \in x$ then $y \in \mathcal{P}(\Sigma^*)$

Selfie:

Is true that *for every regular language L, there exists a regular expression E so that* $L_E = L$.

Is it clear that there exist languages that are NOT regular?

Is every finite language regular?

Is every infinite regular language countable?

Given a regular language, can you construct (one of) its regular expression(s)?

For given string s and regular expression E, can you determine whether $s \in L_E$?

Can you generate all strings in L_E once you have E?

The subalgebra of regular languages

We denote the set of languages over alphabet Σ by L_Σ. It is an algebra for certain operations, and $L_\Sigma = \mathcal{P}(\Sigma^*)$.

For $L_1, L_2 \in L_\Sigma$, we can consider the language L_1L_2 (concatenation), $L_1 \cup L_2$ (union), L_1^* (repetition), and $\overline{L_1}$ (complement). The result is always a member of L_Σ. We say: L_Σ is an algebra with (at least) four internal operations.

Since *RegLan* is a subset of L_Σ, it makes sense to ask whether RegLan is a subalgebra of L_Σ. In other words: are the operations internal to RegLan?

Selfie: Formulate a theorem stating that RegLan is a subalgebra of L_Σ (for the four operations above), and prove it in a constructive way, i.e. (for union) construct an E so that $L_E = L_{E_1} \cup L_{E_2}$. Did you succeed for complement?

Finite automata

Finite state machines (FSM) are used to describe languages: testing and generating strings is part of that. We start with a graphical representation of an FSM. The formal definition follows later. Figure 3.1 shows a first example of an *NFA* [5] over the alphabet $\{a, b, c\}$.

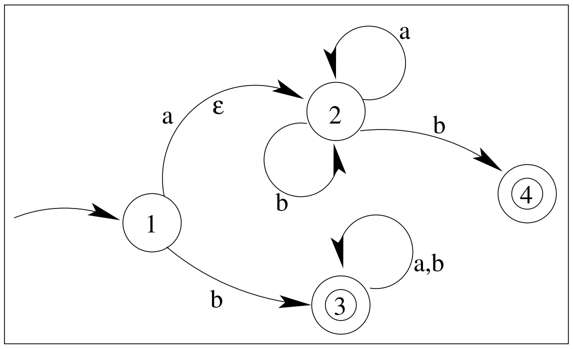


Figure 3.1: A finite state machine

The most important characteristics are:

- we see a directed graph
- the vertices have a name (in this case an number) - the vertices are named *states*
- there are two kinds of states: states with a double circle are named accepting states (sometimes also named end states or final states)
- loops are allowed
- the edges are labelled with zero, one or more symbols from the alphabet, and/or ϵ
- there is exactly one edge not starting from a vertex: the vertex in which it arrives is named the start state

We can use this graphical representation as follows:

1. you get a string *s* over the alphabet (it is now your current string) and you start in the start state

[5]The explanation of this acronym follows later.

2. you can now go from one state to the other along any given edge but you must pay a price for that: you spend from the beginning of your current string a symbol that appears on the edge (meaning that your string shortens); in case the edge contains ϵ, you do not need to spend a symbol from your string
3. if you manage to arrive in a final state with an empty string, we say *the NFA accepts the initial string s*

This constitues only an informal definition of which strings are accepted by an NFA.

Here is a different use of the graphical representation:

1. start with an empty string in the start state: that is your current string
2. now follow any edges: if the edge has a label with symbols from the alphabet, add any of these symbols to you string; if the edge contains ϵ you don't have to add any symbol; keep doing that
3. any time you get at a final state, you may say *this machine generates the current string*

Selfie: Answer the following questions:

does the NFA in Figure 3.1 accept ac?

does the NFA in Figure 3.1 accept bbb?

is there more than one way to accept bbb?

is it possible that you get stuck with bbb and what does that mean for bbb?

can you give some strings not accepted by this NFA?

construct an NFA with a circuit - now you can get into an endless loop? is that a problem?

can you comment on *the set of strings generated by the NFA equals the set of strings accepted by the NFA* ?

Informal definition 3.7.1. *An NFA M determines a language ...*
A language L is determined by the NFA M, if M accepts every string from L, and no other strings. We denote this language by L_M.

The alphabets of the NFA and the language do not need to be the same, but it is convenient to assume they are.

Definition 3.7.2. *Equivalence of NFAs*
Two NFAs are **equivalent** if they determine the same language.

The relation *NFA-equivalence* defines an equivalence relation on NFAs: each equivalence class corresponds to one language.

Selfie: think about the following

there exists a procedure for deciding whether two given NFAs are equivalent

for each NFA there exists an equivalent one with at most one final state

for each NFA there exists an equivalent one in which *you can't get stuck*

We need often the set $\Sigma \cup \{\epsilon\}$: we abbreviate it by Σ_ϵ.

Definition 3.7.3. *Non-deterministic finite state automaton*
A **non-deterministic finite state automaton** is a 5-tuple $(Q, \Sigma, \delta, q_s, F)$ with

- Q a finite set of states
- Σ a finite alphabet
- δ a transition function, i.e. $\delta : Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$
- q_s is the start state and an element of Q
- $F \subseteq Q$: F is the set of final states

As you guessed by now: NFA is acronym if *Non-deterministic Finite Automaton.*

We define formally what it means for a string s to be accepted by an NFA.

Definition 3.7.4. *A string accepted by an NFA*
A string s is accepted by an NFA $(Q, \Sigma, \delta, q_s, F)$ if s can be written as $a_1a_2a_3...a_n$ with $a_i \in \Sigma_\epsilon$, and there exist a sequence of states $t_1t_2t_2t_3...t_{n+1}$ so that

- $t_1 = q_s$
- $t_{i+1} \in \delta(t_i, a_i)$
- $t_{n+1} \in F$

Selfie:

> You now have both an intuitive notion of acceptance by an NFA (through its graphical representation) and a formal definition: make sure that your intuition matches the definition.

Up to now, we have defined only non-deterministic automata: it is possible that at a particular state, there is more than one outgoing edge with the same symbol. This leaves a potential choice. But some of those NFAs are deterministic: there is no choice. Such an automaton is a DFA.

The next section shows the connection between regular expressions and NFAs: we first construct an NFA from a RE so that $L_{RE} = L_{NFA}$, and later we do the reverse. Together, this will prove that the two formalisms are (in some sense) equivalent.

The transition table

The δ of an NFA is a function with a finite domain. It can be represented in the form of a table named the *transition table*: it shows which transitions are possible in the NFA. As an example:

Q	Σ_ϵ	$\mathcal{P}(Q)$
1	a	$\{2\}$
1	b	$\{3\}$
1	ϵ	$\{2\}$
2	a	$\{2\}$
2	b	$\{2,4\}$
3	a	$\{3\}$
3	b	$\{3\}$
2	ϵ	$\emptyset$
3	ϵ	$\emptyset$
4	a	$\emptyset$
4	b	$\emptyset$
4	ϵ	$\emptyset$
1,2,3,4	c	$\emptyset$

Table 3.1: The transition table of the NFA in Figure 3.1

For combination of a state in the NFA ad a symbol in Σ_ϵ for which there is an edge in the graphical representation, there is a set of states to which transition is possible. When there is no edge, one can add the entry with an empty set of states.

The transition table definitely looks usable in a program that implements the NFA. The non-determinism is perhaps a bit scary, but don't panic: we will get rid of it.

An algebra of NFAs

We fix the alphabet: we might relax this later. The set of NFAs over this alphabet is well defined: use the definition on page 72. We will show that on this set, we can define three internal operations: two binary ones (union and concatenation) and a unary one (the star). In the mean time, you know that an NFA can always be transformed to an equivalent one with exactly one accepting state from which no edges leave. That will make things a little easier.

The union of two NFAs: Figure 3.2 shows the intuition about the union of two NFAs: construct a new accepting state and draw an ϵ-edge from the old end states to the new one; the old end states become non-accepeting. Construct a new starting state, with ϵ-edges to the old starting states.

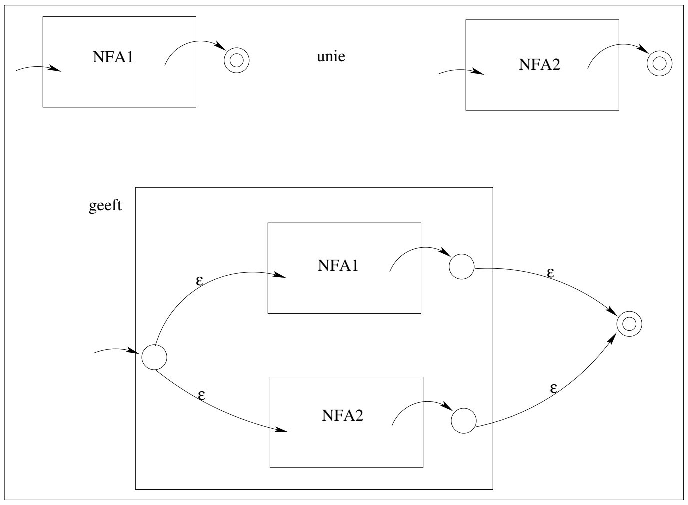


Figure 3.2: The union of two NFAs

We can write formally:

Given $NFA_1 = (Q_1, \Sigma, \delta_1, q_{s1}, \{q_{f1}\})$ en $NFA_2 = (Q_2, \Sigma, \delta_2, q_{s2}, \{q_{f2}\})$.

The union $NFA_1 \cup NFA_2$ is an $NFA = (Q, \Sigma, \delta, q_s, F)$ in which

- $Q = Q_1 \cup Q_2 \cup \{q_s, q_f\}$ with q_s and q_f new states

- $F = \{q_f\}$
- δ is defined as

  $\delta(q, x) = \delta_i(q, x) \ \ \forall q \in Q_i \backslash \{q_{fi}\}, x \in \Sigma_\epsilon$ for i=1,2
  $\delta(q_s, \epsilon) = \{q_{s1}, q_{s2}\}$
  $\delta(q_s, x) = \emptyset \ \ \forall x \in \Sigma$
  $\delta(q_{fi}, \epsilon) = \{q_f\}$ for i = 1,2
  $\delta(q_{fi}, x) = \emptyset \ \ \forall x \in \Sigma$ en for i = 1,2

The concatenation of two NFAs: this time we just give the graphical representation of concatenation in Figure 3.3:

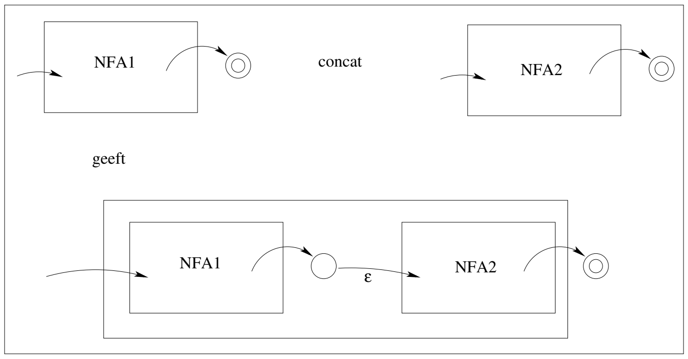


Figure 3.3: Concatenation of two NFAs

The star of an NFA: see Figure 3.4.

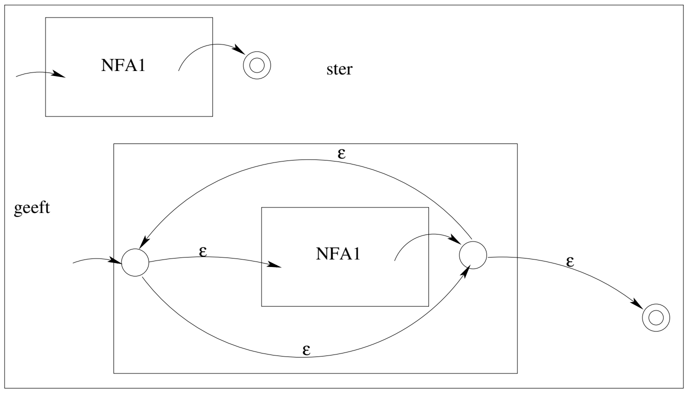


Figure 3.4: The star of an NFA

You should work out the formal description of concatenation and star yourself.

Selfie:

The concatenation of NFA_1 and NFA_2 determines $L_{NFA_1}L_{NFA_2}$: prove this.

Formulate a similar statement for star and union.

What does this prove about the algebraic isomorphism between ... and ... ?

From regular expression to NFA

We have now all necessary ingredients to construct from a regular expression RE an NFA such that $L_{RE} = L_{NFA}$. Since regular expressions are defined inductively (see definition page 66) we define for each case in the definition a corresponding NFA. NFA_{RE} denotes the NFA corresponding to a regular expression RE.

Figure 3.5 shows the NFA for the first three bases cases in the definition on page 66:

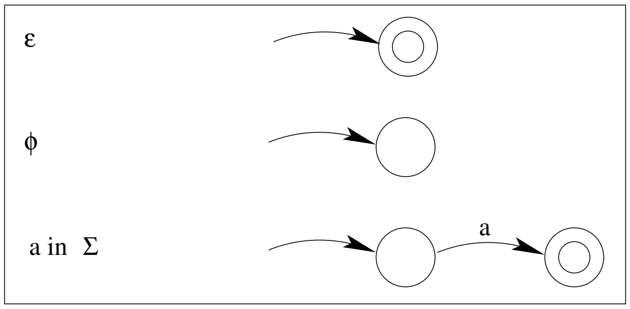


Figure 3.5: An NFA for the three base cases

The three *recursive* cases are described as follows: let E_1 and E_2 be two regular expressions, then

- $NFA_{E_1E_2} = concat(NFA_{E_1}, NFA_{E_2})$
- $NFA_{E_1^*} = star(NFA_{E_1})$
- $NFA_{E_1|E_2} = union(NFA_{E_1}, NFA_{E_2})$

Theorem 3.10.1. *The construction above preserves the language, i.e.*
$$L_{NFA_E} = L_E.$$

Proof. Prove this yourself - use structural induction. ■

From NFA to regular expression

The opposite way is a bit more complicated: we first introduce a new kind of finite automata, the *Generalized Non-deterministic Finite Automata* or GNFAs. We will then follow the path:

$NFA \rightarrow GNFA \rightarrow GNFA \text{ with } 2 \text{ states} \rightarrow \text{regular expression}$

In each of the steps, we must prove that the language stays the same.

Informal definition 3.11.1. *GNFA*
A **GNFA** is a finite state machine with the following modifications and restrictions:

- there is exactly one end state and it differs from the start state
- there is exactly one edge from the start state to any other state, but the start state has no incoming edge (except for the start arrow)
- there is exactly one edge from each state to the end state, but the end state has no outgoing edge
- there is exactly one edge between every other two state (in both directions) and an edge from each state to itself
- the edges carry a regular expression as a label

Figure 3.6 shows a GNFA.

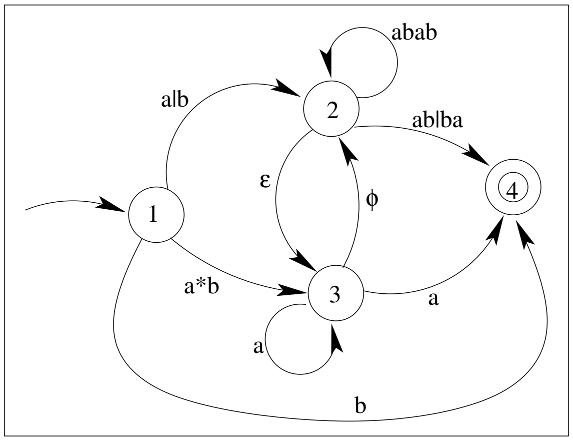


Figure 3.6: Een GNFA

We use the (graphical representation of the) GNFA as follows:

1. you start off with a string s in the start state
2. you may now move to any state (the same or another one) by following an edge as long as your string starts with a substring that matches the regular expression on the edge: during the transition, you chop off that substring; if the regular expression equals ϵ, the substring has lenght zero; if the edge just contains ϕ, you can't take that edge
3. keep making transitions: if you arrive in the end state with an empty string, we say *the GNFA has accepted the initial string s*

Selfie:

For the GNFA in Figure 3.6, find strings accepted by it and strings not accepted by it.

Now is the time to describe an algorithm for constructing an NFA from an RE.

Step 1: *Make a GNFA from an NFA*

Introduce a new start state and a new unique end state. Draw an ϵ-edge from the new start state to the old start state. and from the old end states to the new end state. Draw all missing edges with label ϕ. Replace parallel edges into a new edge with as label the union of the labels of the parallel edges.

Step 2: *Reduce the GNFA to a GNFA with two states*

Choose any state X different from start and end state: if there is none, go to step 3. Remove X and for each pair of states A and B with an edge A to X with E_1, from X to itself with E_2, from X to B with E_3, from A to B with E_4, do replace the label on the edge from A to B by $E_4 \mid E_1 E_2^* E_3$. Finally, remove all edges connected to X.

This base step is illustrated in Figure 3.7.

Repeat step 2.

Step 3: *Determine the RE*

The GNFA has exactly two state: the start and the accepting state. There is only one edge and it has RE as a label.

Example: Figures 3.8 and 3.9 illustrate this on the GNFA of Figure 3.6,

The main thing to prove is that the elimination of X in Step 2 does not change the set of accepted strings. We need to prove two things: (1) if a string s was accepted before

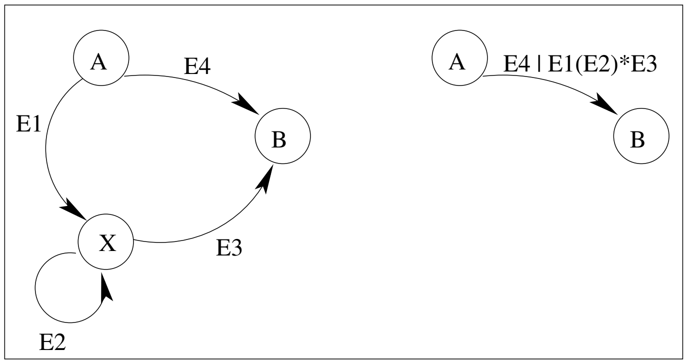


Figure 3.7: Removal of one state from the GNFA

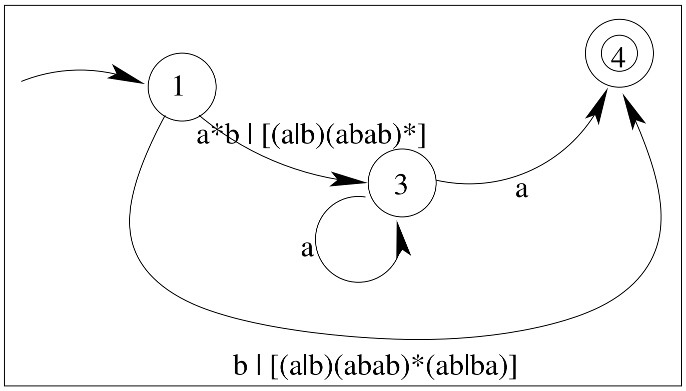


Figure 3.8: State 2 was removed

the elimination, it is also accepted after elimination; (2) if a string is not accepted before elimination, it is not accepted afterwards.

We use the notion of a path through the GNFA that can be followed to accept a string - the related notion for NFAs can be found on page 72: write it out for a GNFA. Such an acceptance path is a sequence of states. We denote the GNFA before the reduction by $GNFA_{before}$ and the machine after the reduction by $GNFA_{after}$.

1. If s is accepted by $GNFA_{before}$ with a path that does not contain X, then s is accepted by $GNFA_{after}$ with the same path.

   If the accepting path contains X, then there are states A and B so that for some $n > 0$, AX^{n}B ($n > 0$) is a subsequence in the path. The regular expressions on the edges AX, XX, XB are E1, E2, E3 and therefore, to go from A to B through X *costs* a piece of string that matches $E1(E2)^*E3$: that regular expression is part of the

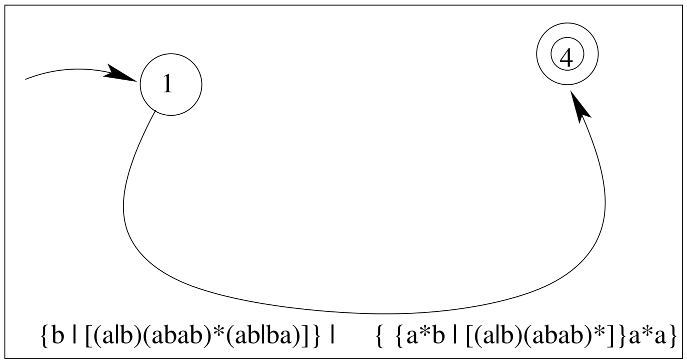


Figure 3.9: State 3 was removed

regular expression on the edge AB in $GNFA_{after}$, so ...

2. If s is accepted by $GNFA_{after}$ then an accepting path cannot contain X. Let AB be a subsequence of the path: on the edged from A to B, part of the regular expression is $E4|E1(E2)^*E3$: using this while going from A to B, means in $GNFA_{before}$ following the original edge from A to B (with E4), or following AX, XX (as often as needed) and then XB, i.e. $E1(E2)^*E3$. It follows that a string accepted by $GNFA_{after}$ is also accepted by $GNFA_{before}$.

We also need to justify that the GNFA obtained in step 1 determines the same language as the NFA we started from: please do that.

Conclusion: the two formalisms NFA and RE determine exactly the same class of languages, i.e. the regular languages. Our proofs were constructive and can be transformed into a program in Java, Prolog ... that compute from an RE an NFA, and vice versa. We will go two steps further: we will get rid of the non-determinism in Section 3.12, and we will build the smallest possible machine in Section 3.13.

Deterministic finite state machines

The definition of a (non-deterministic) finite state machine NFA allows for more than one edge leaving a state with the same symbol, and also with ϵ: both are a source of non-determinism. Indeed, one has the choice to shorten the current string or not, and one has the choice to which state to move. Implementing this (on the basis of the transition table for instance) is not so difficult, but it is clear that a program that is based on taking steps that might have to backtracked over cannot be optimal. It would be more efficient if there are no ϵ-edges, and if for each symbol in the alphabet, there were only one edge with that label from each state. Such an automaton would be called a deterministic finite state machine, abbreviated as DFA.

Formally, we restrict the transition function to the signature $\delta : Q \times \Sigma \rightarrow Q$, but we allow it to be a partial function. It should be clear that DFAs determine regular languages. So we are left with the question: is every regular language determined by a DFA? Or, equivalently, we can wonder whether every NFA can be transformed to an equivalent DFA. We describe this transformation in general.

Given: a NFA = $(Q_n, \Sigma, \delta_n, q_{sn}, F_n)$

To construct: a DFA = $(Q_d, \Sigma, \delta_d, q_{sd}, F_d)$ such that xs$L_{NFA} = L_{DFA}$

Construction: $Q_d = \mathcal{P}(Q_n)$: one state in the DFA is a set of NFA states

$F_d = \{S|S \in Q_d$,s$\cap F_n \neq \emptyset\}$: an end state in the DFA is any state that contains an NFA end state

δ_d has signature $\delta_d : (\mathcal{P}(Q_n) \times \Sigma) \rightarrow \mathcal{P}(Q_n)$

We first define $er : Q_n \rightarrow \mathcal{P}(Q_n)$ (er means **e**psilon-**r**eachable):

- $er(q)$ is the set of NFA states that can be reached from q using zero, one or more ϵ-edges
- we lift the definition of er to $\mathcal{P}(Q_n)$ in the usual way: for $\mathcal{Q} \in \mathcal{P}(Q_n)$

  $$er(\mathcal{Q}) = \cup_{q \in \mathcal{Q}} er(q)$$
- we lift δ_n in a similar way.

We can now define δ_d as:

- $\delta_d(\mathcal{Q}, a) = er(\delta_n(\mathcal{Q}, a))$ [6] for $\mathcal{Q} \in Q_d$
- in words: from a DFA state $\mathcal{Q}$, you move to the next DFA state by first using the symbol a an each NFA state in $\mathcal{Q}$ using the NFA transition function, and then following as many ϵ-edges as possible; then take the union of all the NFA states you could reach in this way.

Finally, we define

$$q_{sd} = er(q_{sn}).$$

End

We use this construction on the NFA of Figure 3.10. This NFA has 3 states, so we might need 8 DFA states.

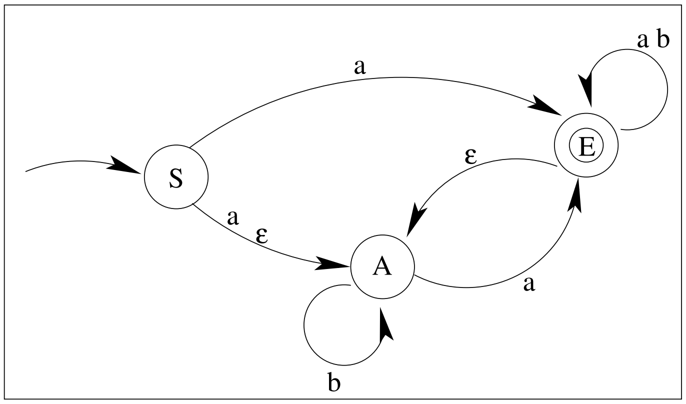


Figure 3.10: An NFA

$q \in Q_d$	$er(q)$	$\delta_n(q, a)$	$\delta_n(q, b)$	$\delta_d(q, a)$	$\delta_d(q, b)$
{}	{}	{}	{}	{}	{}
{S}	{S,A}	{A,E}	{}	{A,E}	{}
{A}	{A}	{E}	{A}	{A,E}	{A}
{E}	{A,E}	{E}	{E}	{A,E}	{A,E}
{S,A}	{S,A}	{A,E}	{A}	{A,E}	{A}
{S,E}	{S,A,E}	{A,E}	{E}	{A,E}	{A,E}
{A,E}	{A,E}	{E}	{A,E}	{A,E}	{A,E}
{S,A,E}	{S,A,E}	{A,E}	{A,E}	{A,E}	{A,E}

Table 3.2: The DFA obtained from the NFA in Figure 3.10

Table 3.2 shows the resulting transition table, and er.

Some states are not reachable from the start state $\{S, A\}$. It suffices to represent only the reachable states, as in the graphical representation of the DFA in Figure 3.11.

[6]Check the signature!

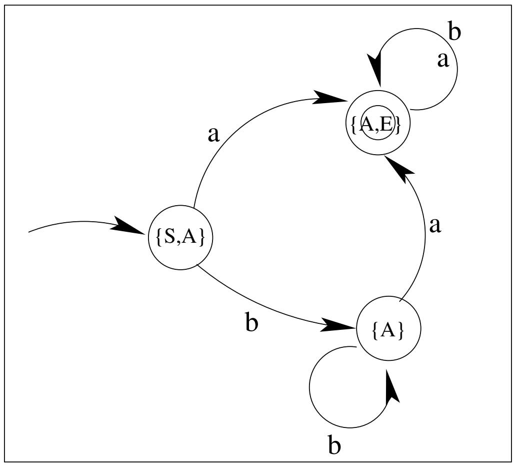


Figure 3.11: The resulting DFA

Selfie:

Which language does the DFA determine? Give an RE for it, or express in words which strings are accepted. Is this the *smallest* DFA accepting the same language? What would be a good notion of *size* of a DFA?

The construction could in the worst case require $2^{\#Q_n}$ DFA states, but the example shows that Q_d does not need to be larger than Q_n if we do not use the unreachable states. Is it possible that the DFA has less states than the NFA you start from?

Extending δ to strings: the domain of δ in a DFA is $Q \times \Sigma$. It is useful to extend δ to a function δ^* on the domain $Q \times \Sigma^*$ as follows:

- $\delta^*(q, \epsilon) = q$
- $\delta^*(q, aw) = \delta^*(\delta(q, a), w)$ if $\delta(q, a)$ exists - $a \in \Sigma$ and $w \in \Sigma^*$.

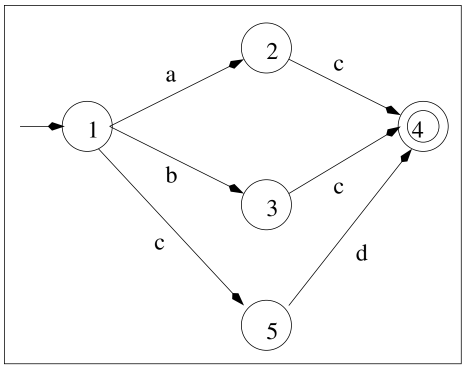
Selfie:

Prove that $\delta^*(q, wa) = \delta(\delta^*(q, w), a)$ for $a \in \Sigma$ and $w \in \Sigma_\epsilon^*$

The minimal DFA

For any regular language L, there exist many DFAs - how many? It is important to construct small machines: in an application in which a DFA is needed, you need to represent the transition table in one way or another, so you might want to keep the number of states low. It is clear that for a given regular language, there exists a minimal DFA, i.e. a DFA with the smallest number of states[7]. We will construct such a minimal (equivalent) DFA by removing states from a given DFA. We will prove minimality and even uniqueness.

States that are unreachable from q_s can be removed: that does not change the language. There can be another way to remove states: Figure 3.12 shows at the left a DFA with 5 states.



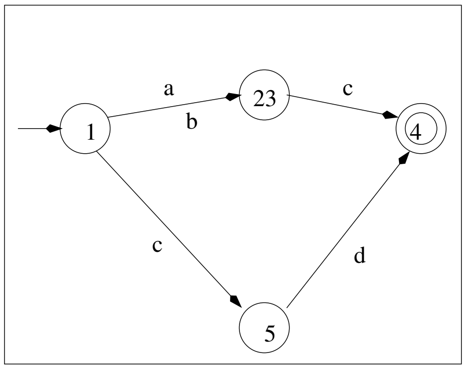


Figure 3.12: At the left a DFA with 2 *equivalent* states; thet were collapsed at the right.

From states 2 and 3, we see leaving edges with the same label and arriving in an end state. This shows that the two states are in some sense indistinguishable: from state 2 and 3, the same strings lead to F. On the other hand, states 3 and 5 do not have this property: the same string can lead to F and not. So, states 3 and 5 are distinguishable. The minimalisation idea consists in: identify indistinguishable states and collapse them.

We will assume that δ is total, i.e. that from each state there is an outgoing edge for every symbol in the alphabet. A DFA without that property can be easily transformed to a DFA complying to it: convince yourself you need at most one extra state.

Definition 3.13.1. *Indistinguishable states*
Two states p and q are **indistinguishable** if

$$\forall w \in \Sigma^* : \delta^*(p, w) \in F \Longleftrightarrow \delta^*(q, w) \in F$$

Two states are **distinguishable** if they are not indistinguishable.

[7]Why is that clear?

If p and q distinguishable, then there exists a word w so that

$$\delta^*(p,w) \in F \ \ \textit{and} \ \ \delta^*(q,w) \notin F \text{ or vice versa.}$$

We describe first informally an algorithm to compute sets of indistinguishable states, and what to do with them.

Init: a state $p \notin F$ is for sure distinguishable from any state in F; at this moment, we have not taken a decision about whether any two other states are distinguishable.

Repeat: take any pair of states p and q about which we have not taken a decision yet: suppose there is a symbol a so that with a you go fro p and q to distinguishable states; now mark p and q as distinguishable; repeat until no more such pair

Consolidate: for each pair of states p and q without a decision, decide that p and q are indistinguishable; indistinguishable is an equivalence relation, so consider the equivalence classes Q_i within Q; these Q_i will be the states in the minimal DFA; δ will be shown later

To save on writing, we use p_a as a shorthand for $\delta(p,a)$.

Algorithm 3.13.2. *Indistinguishable states*

Init: *Consider the graph V with as vertices the states of the DFA; add an edge between any two vertices of which exactly one belongs to F and label these edges with ϵ ; an edge between x and y with label l will be denoted by (x,y,l)*

Repeat: *If there exist verticesp and q so that*

- *there is no edge between p and q*
- $\exists a \in \Sigma : \exists (p_a, q_a, \_) \in V$

then choose an a with $(p_a, q_a, \_) \in V$ and add (p,q,a) to V; go back to the beginning of Repeat;

otherwise: go to Equal;

Equal: *Consider the complement of G: each component is a clique; let Q_i be the set of vertices in the i^{th} component; all states in Q_i are indistinguishable; every state in Q_i is distinguishable from any other state not in Q_i*

Proof. The termination of the algorithm is obvious: the maximal number of edges in V is $N(N-1)/2$ with $N = |Q|$; one such edge is added in Repeat each time the condition is fulfilled.

we prve that

$$(p, q, \_) \text{ is an edge in } V \Longleftrightarrow p \text{ and } q \text{ are distinguishable}$$

$\underline{\Longrightarrow}$: if (p, q, X) is an edge in V, then there are two possibilities: $X = \epsilon$ or $X = a \in \Sigma$; in the former case, we know that p and q are distinguishable (take $w = \epsilon$ in the conclusion after the definition of distinguishable on page 85); in the latter case, there exists an edge of the form $(p_a, q_a, \_)$ in V: you can use the label to arrive (eventually) at the former base case, i.e. $\exists w \in \Sigma^* : \exists(p_w, q_w, \epsilon) \in V$, and we conclude that p and q distinguishable.

$\underline{\Longleftarrow}$: if p and q distinguishable, then there exists a w so that $\delta^*(p, w) \in F$ *and* $\delta^*(q, w) \notin F$ or vice versa; if w is the empty string, the conclusion holds; if not, then w has a last symbol $z \in \Sigma$ and can be written as $w = vz$; we then know that $\delta^*(p, w) = \delta(\delta^*(p, v), z)$ so, between states $\delta^*(p, v)$ and $\delta^*(q, v)$ there is an edge; we can now get rid of the last symbol of v etc. until we derive that there is an edge between $\delta^*(p, \epsilon)$ and $\delta^*(q, \epsilon)$, and we are done. ■

We have now constructed the equivalence classes of indistinguishable states - the Q_i at the end of the algorithm: we are now ready to define the components of the DFA_{min}, starting from a DFA $(Q, \Sigma, \delta, q_s, F)$ all of whose states are reachable.

DFA_{min} consists of $(\tilde{Q}, \Sigma, \tilde{\delta}, \tilde{q_s}, \tilde{F})$ with

- $\tilde{Q} = \{Q_1, Q_2, ...\}$ in which the Q_i are as computed by the algorithm
- $\tilde{\delta}(Q_i, a) = Q_j$ in which Q_j is obtained by taking any $q \in Q_i$ (*) and then taking the Q_j containing $\delta(q, a)$
- $\tilde{q_s}$ is the Q_i for which $q_s \in Q_i$
- $\tilde{F}$ is the set of Q_i for which $Q_i \cap F \neq \emptyset$

Figure 3.13 illustrates how V evolves for the DFA in Figure 3.12, and the complement of the final graph.

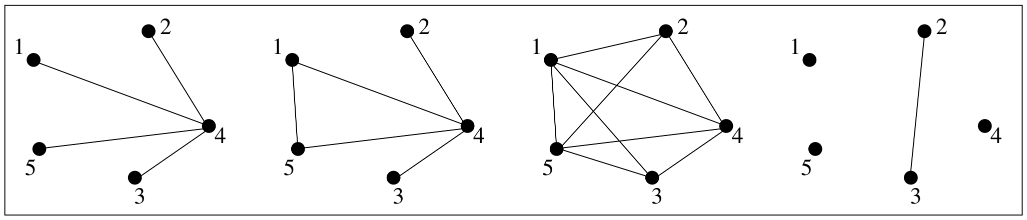


Figure 3.13: 4 steps in the algorithm

Selfie:

> In (*) above, really any $q \in Q_i$ can be chosen: prove that this is indeed true.
>
> Prove that $Q_i \cap F \neq \emptyset$ implies that $Q_i \cap F = Q_i$, or in words: every element of Q_i belongs to F.
>
> Prove that in DFA_{min} every pair of states is distinguishable.
>
> Finally, prove that $L_{DFA} = L_{DFA_{min}}$
>
> What happens when the algorithm is given a DFA with unreachable states ?

We still need to prove that the just constructed DFA_{min} has the minimum number of states. We prove a slightly stronger theorem:

Theorem 3.13.3. *If $DFA_1 = (Q_1, \Sigma, \delta_1, q_s, F_1)$ is a machine all of whose states are reachable, and with every pair of states distinguishable, then there is no equivalent DFA with strictly less states.*

Proof. Let DFA_1 have states $\{q_s, q_1, ..., q_n\}$: q_s is the starting state. Assume that $DFA_2 = (Q_2, \Sigma, \delta_2, p_s, F_2)$ is a DFA with strictly less states $\{p_s, p_1, ..., p_m\}$ than DFA_1.

Since in DFA_1 every state is reachable, there exist strings $s_i, i = 1..n$ such that $\delta_1^*(q_s, s_i) = q_i$.

Since DFA_2 has less states, there must exist $i \neq j$ so that

$$\delta_2^*(p_s, s_i) = \delta_2^*(p_s, s_j).$$

Since q_i and q_j are distinguishable, there exists a string v so that

$$\delta_1^*(q_i, v) \in F_1 \wedge \delta_1^*(q_j, v) \notin F_1 \text{ or vice versa.}$$

Consequently, $\delta_1^*(q_s, s_i v) \in F_1 \wedge \delta_1^*(q_s, s_j v) \notin F_1$ or vice versa, meaning that DFA_1 accepts $s_i v$ or $s_j v$ but not both.

On the other hand, in DFA_2 we have $\delta_2^*(p_s, s_i v) = \delta_2^*(\delta_2^*(p_s, s_i), v) = \delta_2^*(\delta_2^*(p_s, s_j), v) = \delta_2^*(p_s, s_j v)$ which means that DFA_2 accepts both string $s_i v$ and $s_j v$, or rejects both.

It follows that DFA_1 and DFA_2 are not equivalent. ∎

All states of the constructed DFA_{min} are reachable, and they are pairwise distinguishable, so our DFA_{min} has the minimum possible number of states.

Two DFAs can only be really the same if their states are the same, or otherwise said, if their states have the same name. Also, their δ, final states and start state must be the same. Still, two DFAs (on the same alphabet) can be similar except for the names of the states. We express that by the notion of a DFA isomorphism.

Definition 3.13.4. *Isomorphic DFAs*
$DFA_1 = (Q_1, \Sigma, \delta_1, q_{s1}, F_1)$ is **isomorphic** to $DFA_2 = (Q_2, \Sigma, q_{s2}, \delta_2, F_2)$ if there exists a bijection $b : Q_1 \to Q_2$ so that

- $b(F_1) = F_2$
- $b(q_{s1}) = q_{s2}$
- $b(\delta_1(q, a)) = \delta_2(b(q), a)$ (see Figure 3.14)

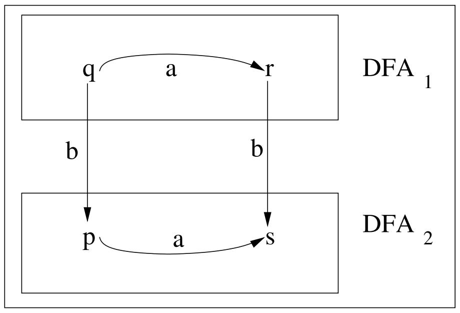


Figure 3.14: Commutative diagram for b and δ_i

You should be able to prove that two isomorphic DFAs are equivalent (but not necessarily the other way around).

We can now prove that the minimal DFA is unique up to isomorphism - give it a try! You can start from the observation that two equivalent DFAs whose states are reachable and distinguishable must have the same number of states ...

The pumping lemma for regular languages

In this section, we get acquainted with a method to prove that a given language is not regular. Before doing that, here are two arguments why there exist non-regular languages: it is up to you to make those arguments hard, if not now, certainly later:

1. the number of DFAs (over a fixed alphabet) is countably infinite; the number if languages is uncountably infinite ...
2. the complexity of deciding whether a string s belongs to a given regular language, is linear in the size of s; since there are decision problems with a higher complexity ...

Suppose we have an infinite regular language L. L has a DFA deciding it. This DFA has $N = \#Q$ states. Consider a string $s \in L$ so that $|s| \geq N$, and follow the acceptance path through the DFA with that string until you reach F. Since the string is longer than the number of states, at least one state occurs twice in that path: the path contains a cycle. That cycle *consumes* a part y of the string s, and $s = xyz$ where x, y and z. Convince yourself that with the string $xyyz$ you could also reach F, and more generally with xy^iz for any $i \geq 0$.

More formally

Theorem 3.14.1. *The pumping lemma for regular languages*
For every regular language L, there exists a pumping length d, so that for all $s \in L$ with $|s| \geq d$, there exists a division of s in three parts $s = xyz$ and

1. $\forall i \geq 0 : xy^iz \in L$
2. $|y| > 0$
3. $|xy| \leq d$

Proof. Take any DFA that determines L. Take $d = \#Q$.

Let $s \in L$ and $s = a_1a_2...a_n$ with $n \geq d$. Consider the sequence of states along the accepting path of s: $(q_s = q_1, q_2, ..., q_{n+1})$: its lengths is strictly larger that d, so in the first d elements of this sequence, there must be a repeated state. Assume that q_i equals q_j, with $i < j \leq d$, then define $x = a_1a_2...a_i$, $y = a_{i+1}...a_j$, and z the rest of the string. All statements now follow easily. ■

Figure 3.15 illustrates the division of s as xyz.

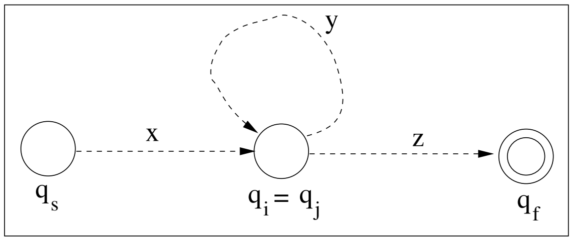


Figure 3.15: $s = xyz$

Using the pumping lemma

First convince yourself that you cannot use the pumping lemma for proving that a given language is regular. In fact, there are languages whose strings can be pumped, but which are not regular.

The classical example for applying the pumping lemma is the language $L = \{a^n b^n | n \in \mathbb{N}\}$ over the alphabet $\{a, b\}$.

Suppose that language has a pumping length d; consider the string $s = a^d b^d$. Take any division of $s = xyz$ with $|y| > 0$. There are now three possibilities:

- y contains only a's: then xyyz contains more a's than b's and does not belong to L
- y is of the form $a^i b^j$ with $i \neq 0, j \neq 0$: then xyyz has some a's and b's in the wrong order, so that string is not in L
- y contains only b's: then xyyz contains more b's than a's and does not belong to L

As a consequence, L is not regular.

We have not used point 3 in the theorem. If we do, the proof becomes shorter and easier. Suppose that language has a pumping length d; consider the string $s = a^d b^d$. Take any division of $s = xyz$ with $|y| > 0$ and $|xy| \leq d$. Now y contains only a's and ...

Note: Using the pumping lemma to prove that L is not in RegLan, needs the following:

- for any number d chosen as pumping length
- there exists a string $s \in L$ longer than d
- for which **every** division xyz prevents pumping (i.e. there is an i for which $xy^i z$ is not in L)

Especially the latter is important - here is an example of how not to use the lemma. Take as L the language generated by the regular expression ab^*c. We prove (wrongly !) that L is not regular: take any pumping length $d > 0$ and take the string $ab^d c$. Divide that string as $abc = xyz$ with $x = \epsilon$, $y = a$ en $z = bc$. Clearly, $xyyz$ is not in L so L is not regular ... or is it?

Selfie:

Which error was made above?

Define some languages and try to use the pumping lemma to prove they are not regular.

Is the language of the regular expressions (over a fixed alphabet) regular itself?

Prove: every regular language has a minimal pumping length. Is it related to the minimal DFA for that language?

Intersection, difference and complement of two DFAs

We have considered the union of regular languages before. Now it is time to study the other usual set operations: intersection, (symmetric) difference, and complement. We assume given two DFAs $(Q_i, \Sigma, \delta_i, q_{si}, F_i)$ for i=1,2. We construct a generic product DFA $(Q, \Sigma, \delta, q_s, F)$ as follows:

- $Q = Q_1 \times Q_2$
- $\delta(p \times q, x) = \delta_1(p, x) \times \delta_2(q, x)$
- $q_s = q_{s1} \times q_{s2}$

We still have some freedom in defining F:

- $F = F_1 \times F_2$: the new DFA defines the intersection of the two given languages
- $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$: ... union ... van de twee languages
- $F = F_1 \times (Q_2 \setminus F_2)$: the new DFA determines $L_1 \setminus L_2$
- $F = (Q_1 \setminus F_1) \times (Q_2 \setminus F_2)$: the new DFA determines $L_1 \triangle L_2$

The above constructions show that the union, the intersection, and the (symmetric) difference of two regular languages is also regular. It follows that also the complement is regular, since $\overline{L} = \Sigma^* \setminus L$.

Selfie:

Find a simpler construction of the complement of a given DFA.

Now use that same construction on an NFA; do you get what you wanted? Why (not)?

Regular expressions and lexical analysis

It is often easy to use a regular expression to specify which *input* is allowed within a certain context. For instance

$$20(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)$$

indicates that any year in this century may be input.

Also, the lexical tokens of a programming language are often specified by REs. The following could be a small part of the Java lexicon: $(a|b|c)(a|b|c|0|1|2)^* \mid (-|\epsilon)(1|2)(0|1|2)^*$ which describes identifiers that start with the letter a,b or c, after which are followed by any number of those three letters and digits (only 0,1,2), and integers that can start with a minus sign.

Such a specification becomes clumsy without the use of abbreviations. So one typically specifies a lexicon as follows:

$$PosDigit \leftarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

$$Digit \leftarrow PosDigit \mid 0$$

$$ThisCentury \leftarrow 20DigitDigit$$

and the general form of an integers as $(+|-|\epsilon)PosDigit\ Digit^* \mid 0$

or the general form of a bank account number:

$$BANKNR \leftarrow Digit^3(- \mid \epsilon)Digit^7(- \mid \epsilon)Digit^2$$

As for the Java lexicon:

$$JavaProgr \leftarrow (Id|Int|Float|Op|Delimiter)^*$$

$$Id \leftarrow Letter\ (Letter|Digit)^*$$

$$Digit \leftarrow PosDigit \mid 0$$

$$PosDigit \leftarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

$$Sign \leftarrow (+|-|\epsilon)$$

$$Unsigned \leftarrow PosDigit\ Digit^*$$

$$Int \leftarrow Sign\ Unsigned$$

$$Float \leftarrow Int\ .\ Unsigned$$

...

The description of a lexicon can be used to construct automatically a *lexer* for Java, i.e. a program that takes a Java program as input, and breaks it up in lexical units. You are at this moment actually capable of doing that: you got all the theory to make a good lexer!

There exist tools that do that for you: e.g. flex `http://flex.sourceforge.net/`) generates C-code: the C-code implements the DFAs (and necessary glue-code, and even error handling). jflex (`http://www.jflex.de/`) is based on the same principle and generates Java code. flex en jflex are both lexical analyse generators.

Selfie:

> Read more about (j)flex.
>
> Wite a program (pick your language) that takes as input a regular expression E, and produces a Prolog program with a predicate lex/1: a query like ?- lex(*string*). should succeed if and only if *string* is in L_E. You can represent a string like "abc" as the list $[a, b, c]$. An RE like $a^*b \mid c$ can be represented as the term $of([star(a), b], [c])$ (but you are welcome to try another one).

The description above of a $JavaProgr$, is actually a BNF-grammar: it is in *Backus-Nauer form*: this form is actually meant for context free languages. BNF can be used also for regular languages, because they are context free as well: see Section 3.19.

Variants of finite state automata

The transducer

A transducer transforms an input string into an output string: we adapt the definition of a DFA, so that it can produce output. One figure is almost worth a definition.

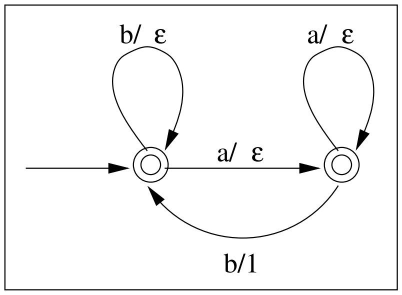


Figure 3.16: A simple transducer

The labels on the arcs have the form a/x: a belongs to the input alphabet, x to the output alphabet. x can also be the empty string. The a is used for finding a path in the transducer as if it were a DFA. The x is output when the edge is used. The above transducer accepts every string and outputs a 1 for every b that follows an a.

A DFA that can check an addition

Since a DFA can only be used for deciding whether a string belongs to a language, we define the language of correct additions. If that language is regular, we should be able to build a DFA for it. Let's do it as follows: $\Sigma = \{0,1\}$. The two numbers we want to add and the result are represented in binary, in reverse order. We also add leading zeros so that all three numbers have the same length. As an example: adding 3 to 13 with result 16 gives us the strings 11000 10110 and 00001. We merge them systematically, i.e. we make groups of 3 bits occurring in the i-th place, and write those groeps in a sequence. We obtain so

110 100 010 010 001

(the spaces are there only for showing the groups of three) As a result, the string 110100010010001 represents the correct addition 3+13=16. A DFA for this language is in Figure 3.17.

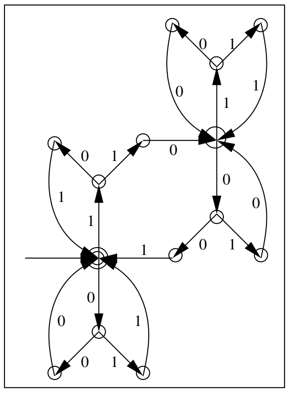


Figure 3.17: An addition checker

Adding by using a transducer

Addition is: given two numbers as input, output their sum. This is possible with a transducer: use the same representation as before for the two numbers to be added, and merge them in the same way. So 3+13 is represented by the string 1110010100. Figure 3.18 shows two addition transducers that are derived from the checker.

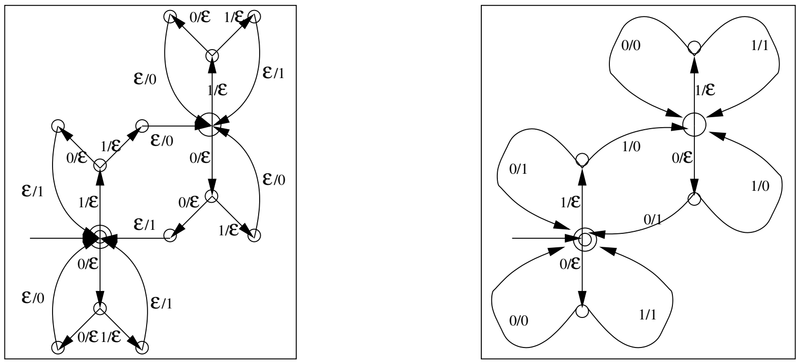


Figure 3.18: Two transducers that produce the sum of two numbers as output

Selfie:

Can a transducer also perform other arithmetic operations, like subtraction or multiplication? Why or why not?

Is the transducer output language regular?

Can you write a transducer that outputs every third character of the input string?

Can you write a transducer that outputs all but every third character of the input string?

Can you write a transducer that outputs the input string in reverse order?

Can every regular language be the output language of a transducer?

Two-way finite automata

Sometimes, one name a DFA a *one-way finite automaton*. The reason is that one describe the DFA as a machine with an input tape on which the input string resides: the machine has a reading head which is initially above the first character of the string. At each transition, the reading head moves one position to the right: always towards the end of the string.

A small adaptation of this automaton allows it to move in both directions: you can adapt the definition of δ (see for instance Chapter 4: there, it is done for Turing machines). What you get is a two-way finite automaton, or a 2DFA.

About other classes of automata, we know that more freedom in how the *memory* can be manipulated, leads to more powerful automata: to fully appreciate this, you might have to wait until you learn about PDA and PBA, but keep this in mind.

Therefore, the question *is a 2DFA more powerful than a DFA?*, or otherwise said *can a 2DFA recognise some non-regular languages?* is relevant. What do you think?

Büchi automata

Büchi[8] automata are like NFAs, only you feed them infinite strings: at no moment is the string finished. That means that acceptance of a string by such an automata is different from acceptance by an NFA:

Definition 3.17.1. The infinite string is accepted by a Büchi automaton if the sequence of states followed by the string passes by an accepting state infinitely many times.

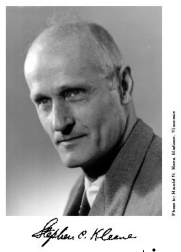
Clearly that is something one can't try out or implement, but that is not the purpose of Büchi automata: you use them to model a problem (e.g. an infinitely repeating process, a protocol ...) and then prove which strings satisfy the acceptance condition.

Non-deterministic Büchi automata are strictly stronger than deterministic Büchi automata: as an example, you could try to build a Büchi automaton for the language $(a|b)^*b^\omega$.

[8]Julius Büchi

References

A lot of what you learned in this chapter has been developed by Stephen Kleene: you might know him from fixpoint theorems. He made also contributions to logic, recursion theory, and intuitionism.



Modern texts on regular languages/expressions/machines do not always follow the same order of presentation: in the previous sections, we have shows that there is no chicken-and-egg problem, since all three are equivalent. There is also a variety of small differences in the basic definitions (e.g. whether δ needs to be total, whether there can be more than one accepting end state for the NFA ...): these do not matter either.

The same diversity in definitions and approaches exists for other languages and machines: we keep our mind open, focus on the essentials, and make sure we know when things are basically equivalent.

References:

- Dexter C. Kozen *Automata and Computability*
- Peter Linz *An Introduction to Formal Languages and Automata*
- Michael Sipser *Introduction to the Theory of Computation*
- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman *Introduction to Automata Theory, Languages, and Computation*
- Marvin Lee Minsky *Computation: Finite and Infinite Machines (Automatic Computation)*

If all this gives you the hots, do not forget another important book: *Regular algebra and finite machines*[9] by John Horton Conway - the same person of *The Game of Life* (it appears later in this text) and other interesting issues (a.o. the *Angels and Devils game*) that affect directly our understanding of algorithmics.

[9]The book is not in the library, but if it interests you, pass by my office

Selfie: Below, we use the same alphabet for all languages, and it contains at least a and b. We use the notation $\hat{s}$ to denote the string with the same characters as s but in reverse order. In each case, write down formally the definition of L3. The question is always the same: suppose L1 and L2 are regular languages, is L3 also regular?

L3 consists of strings obtained by merging strings of the same length from L1 and L2 (also write down formally what merging means!)

L3 = $\widehat{L1}$

L3 consists of the strings with even length from L1

L3 consists of strings s obtained by taking strings of even length from L1, chopping such strings in two parts of equal length as s_1s_2 and then concatenating $\widehat{s_1}$ with s_2

L3 consists of strings obtained by taking a string s from L1 and concatenating s with $\hat{s}$

L3 has the strings from L1 in which every occurrence of a is replaced by b

L3 has the strings from L1 not in L2

L3 consists of the string from L1 from which the symbols on the even places are taken out

$L3 = \{x|\exists y \in L2, xy \in L1\}$

$L3 = \{x|\exists y \in L2, yx \in L1\}$

Context free languages and their grammar

Regular expressions provide one way to determine a language. One can extend REs a little by allowing *abbreviations* for REs, so that shorter descriptions of the language can be obtained. An abbreviation gives a name to something, and when that name appears somewhere, you can replace it with the the thing it abbreviates (or a copy of it): you now get an expression that does no longer contain the name, and has the intended meaning. This means that

$$\text{Brackets} \rightarrow \text{BracketsBrackets} \mid [\text{Brackets}] \mid \epsilon$$

does not define an abbreviation: by replacing the name Brackets with its meaning (the right side), you can not get rid of the name. However, we can still use the above *rule* to define a language. Such rules form a *context free grammar*. Before we define that formally, here are a few examples:

Example 3.19.1.

- $S \rightarrow aSb$

  $S \rightarrow \epsilon$

  This grammar describes the strings of the form $a^n b^n$

- $S \rightarrow PQ$

  $P \rightarrow aPb$

  $P \rightarrow \epsilon$

  $Q \rightarrow cQd$

  $Q \rightarrow \epsilon$

  This grammar describes the strings of the form $a^n b^n c^m d^m$

- $Stat \rightarrow Assign$

  $Stat \rightarrow ITE$

  $ITE \rightarrow If\ Cond\ Then\ Stat\ Else\ Stat$

  $ITE \rightarrow If\ Cond\ Then\ Stat$

  $Cond \rightarrow Id == Id$

  $Assign \rightarrow Id := Id$

  $Id \rightarrow a$

  $Id \rightarrow b$

  $Id \rightarrow c$

Informally, a context free grammar (CFG) consists of rules; at the left side in a rule, we see a nonterminal symbol; at the right side we see a sequence of terminal and nonterminal symbols, and ϵ . There is also a start symbol.

One can use a CFG for generating strings, and also for checking whether a string complies to the grammar. Generation is done by making a derivation starting with the start symbol. The following is a derivation using the first CFG example:

$$S \rightarrow aSb \rightarrow aaSbb \rightarrow aabb$$

The idea is: if a string contains a nonterminal, choose any nonterminal X and replace it by the right hand side of a grammar rule with X at the left. Start with the start symbol and continue until there are only terminals.

We can represent essence of a derivation as a *syntax tree* or *parse tree*: see Figure 3.19.

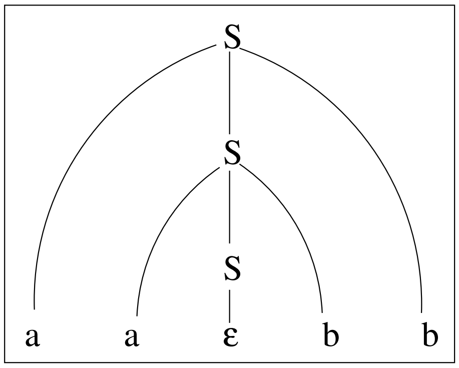


Figure 3.19: Syntax tree of aabb

Clearly, the parse tree hides the order in which rules are applied during the derivation.

We are now ready for more formal definitions.

Definition 3.19.2. *Context Free grammar - CFG*
A context free grammar is a 4-tuple (V, Σ, R, S) in which

- V is a finite set of nonterminal symbols (also named variables or nonterminals)
- Σ is a finite alphabet of terminal symbols (terminals); $\Sigma \cap V = \emptyset$
- R is a finite set of rules (or productions); a rule is a tuple consisting of one nonterminal and a string of elements from $V \cup \Sigma_\epsilon$;
  we write the two parts of a rule with a $\rightarrow$ in between
- S is the start symbol and $S \in V$

It is often clear which are the terminals and which symbol is the start symbol, and in such cases, we just mention the rules.

Definition 3.19.3. *Derivation of a string, based on a CFG*
Let (V, Σ, R, S) be a CFG. A string f over the alphabet $V \cup \Sigma_\epsilon$ is derived from a string b over $V \cup \Sigma_\epsilon$ using the CFG, if there exists a finite sequence of strings $s_0, s_1, ..., s_n$ so that

- $s_0 = b$
- $s_n = f$
- s_{i+1} is obtained from s_i (voor $i < n$) by replacing in s_i a nonterminal X by the right hand side of a rule in which X occurs at the left.

We use the notation: $s_i \Rightarrow s_{i+1}$ en $b \Rightarrow^* f$

Definition 3.19.4. *The language determined by a CFG*
The language L_{CFG} determined by the CFG (V, Σ, R, S) is the set of string over Σ that can be derived from S; or more formally: $L_{CFG} = \{s \in \Sigma^* | S \Rightarrow^* s\}$.

Definition 3.19.5. *Context Free language - CFL*
A language L is **context free** if there exists a CFG so that $L = L_{CFG}$

Regarding CFLs, we intend to follow a similar path as for RegLans: we define a new kind of machine and prove that the set of its accepted languages coincides with the CFLs defined by CFGs. We study the difference between deterministic and non-deterministic versions of these machines; a pumping lemma for CFLs will be established; the algebraic operations on CFLs are studied. We do not deal with minimization: the reason why becomes clear in a later chapter (remember this and get back to it !). We also treat an issue that was a non-issue for RegLans: ambiguity.

Ambiguity

Consider the CFG Arit1

- Expr $\rightarrow$ Expr + Expr
- Expr $\rightarrow$ Expr * Expr
- Expr $\rightarrow$ a

Expr is the start symbol. Consider the string $a + a * a$: it belongs to the language determined by Arit1. Below are two derivations of that string (the substituted nonterminal is underlined whenever there is a possible choice)

$$Expr \Rightarrow \underline{Expr} + Expr \Rightarrow a + Expr \Rightarrow a + \underline{Expr} * Expr$$
$$\Rightarrow a + a * Expr \Rightarrow a + a * a$$

and

$$Expr \Rightarrow Expr + \underline{Expr} \Rightarrow Expr + Expr * \underline{Expr} \Rightarrow Expr + \underline{Expr} * a$$
$$\Rightarrow Expr + a * a \Rightarrow a + a * a$$

In the former derivation, we always substituted the leftmost nonterminal. In the latter always the rightmost one. But the parse tree is in both cases the same (check that !). Those two derivations are essentially the same, and this justifies that we consider only leftmost derivations.

Consider now the following two leftmost derivations of $a + a * a$:

$$Expr \Rightarrow Expr + Expr \Rightarrow a + Expr \Rightarrow a + Expr * Expr$$
$$\Rightarrow^* a + a * a$$

and

$$Expr \Rightarrow Expr * Expr \Rightarrow Expr + Expr * Expr \Rightarrow a + Expr * Expr$$
$$\Rightarrow^* a + a * a$$

or in terms of their parse trees: see Figure 3.20:

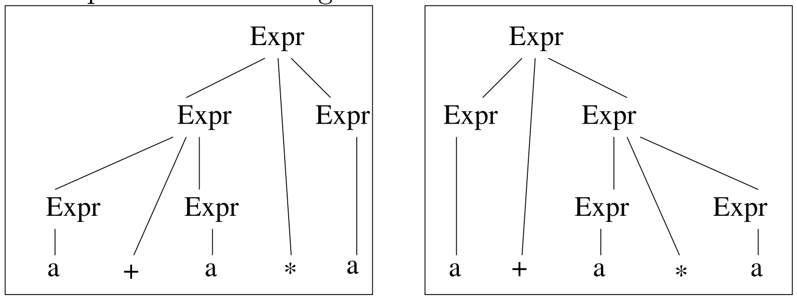


Figure 3.20: Two parse trees for $a + a * a$

The string a+a*a seems to have more than one leftmost derivations and more than one parse tree: the (meaning of the) string is ambiguous. We say that grammar Arit1 is ambiguous because there exist ambiguous strings in L_{Arit1} w.r.t. Arit1. A priori, it is unclear whether for the same language there exists also a non-ambiguous grammar.

Definition 3.19.6. *Equivalence of CFGs*
Two context free grammars $CFG1$ and $CFG2$ are equivalent if
$L_{CFG1} = L_{CFG2}$

You should be able to prove that this defines an equivalence relation on the CFGs.

We define a different CFG Arit2:

- Expr $\rightarrow$ Expr + Term
- Expr $\rightarrow$ Term
- Term $\rightarrow$ Term $*$ a
- Term $\rightarrow$ a

Convince yourself that Arit1 and Arit2 are equivalent.

You can also check that $a + a * a$ has only one leftmost derivation:

$Expr \Rightarrow Expr + Term \Rightarrow Term + Term \Rightarrow a + Term$

$\Rightarrow a + Term * a \Rightarrow a + a * a$

and even betterL every string in its language has exactly one parse tree for Arit2. We say: Arit2 is a non-ambiguous grammar.

Not every CFGL has a non-ambiguous CFG: such a CFL is named *inherently ambiguous*. Here is an example of such a language: $\{a^n b^n c^m, a^n b^m c^m | n, m \geq 0\}$. The proof that it is inherently ambiguous can be found in the Hopcroft-Motwani-Ullman book, but you should be able to prove that the language is context free. We get back to ambiguity later.

A final note on notation: if a nonterminal occurs at the left side of more than one rule, we can cluster those rules. For instance, grammar Arit2 can be written as:

- Expr $\rightarrow$ Expr + Term | Term
- Term $\rightarrow$ Term $*$ a | a

Special forms of CFGs

Often, it is convenient to have grammars of a restricted form, but without excluding any CFLs. Here is such a form.

Definition 3.19.7. *Chomsky Normal Form*
A CFG is in Chomsky Normal Form if every rule has one of the following forms

1. A $\rightarrow$ BC (with A,B,C nonterminals; B,C are different from the start symbol)
2. A $\rightarrow$ α (with α a terminal)
3. $S \rightarrow \epsilon$ (with S the start symbol)

Some textbooks do not the rule $S \rightarrow \epsilon$: such a grammar cannot derive the empty string.

Theorem 3.19.8. *For each CFG there exists an equivalent CFG in Chomsky Normal Form.*

Proof. The proof is constructive, a bit long but not difficult. We start from a given CFG and transform it in little steps to Chomsky Normal Form while retaining equivalence.

1. We start by making sure that there is a start symbol that never occurs at the right of a rule: if S is the start symbol in the given grammar, replace it everywhere by a new nonterminal (say X) and add the rule $S \rightarrow X$

2. We now try to satisfy the third requirement of the definition of Chomsky Normal Form:

suppose there exist rules $\mathcal{E} = A \rightarrow \epsilon$ and $\mathcal{R} = B \rightarrow \gamma$ with A occurring in γ; we define the set of rules $V(\mathcal{E}, \mathcal{R})$ of the form $B \rightarrow \eta$ in which η is obtained from γ by deleting any combination of occurrences of A in γ.

The transform the grammar as follows:

> while there exist rules $\mathcal{E} = A \rightarrow \epsilon$ and $\mathcal{R} = B \rightarrow \gamma$ with A occurring in γ and such that $V(\mathcal{E}, \mathcal{R})$ contains *new* rules, add $V(\mathcal{E}, \mathcal{R})$ to the grammar

Make sure you understand that this ends!

Then delete from the grammar all rules of the form $A \rightarrow \epsilon$, except if $A = S$: that is potentially the only rule deriving ϵ .

Make sure you understand that the resulting grammar still defines the same language: reasoning on the derivations can help.

3. We now want to get rid of the rules of the form $A \to B$ with A and B nonterminals. For a rule of the form $\mathcal{E} = A \to B$ and a rule of the form $\mathcal{R} = B \to \gamma$, define the rule $U(\mathcal{E}, \mathcal{R}) = A \to \gamma$.

> while there exists rules of the form $\mathcal{E} = A \to B$ and $\mathcal{R} = B \to \gamma$, and $U(\mathcal{E}, \mathcal{R})$ is a new rule, add $U(\mathcal{E}, \mathcal{R})$ to the grammar

Make sure you understand that this ends!

Then remove from the grammar all rules of the form $A \to B$.

Make sure you understand that the resulting grammar still defines the same language: reasoning on the derivations can help.

4. There are now 3 types of rules:

(a) $A \to \gamma$ in which γ has exactly two nonterminals: they can stay

(b) $A \to \gamma$ in which γ has at least two symbols; replace each terminal a by a new nonterminal A_a, and add the rule $A_a \to a$

(c) possible $S \to \epsilon$: it can stay

Once more, make sure you understand that this ends and that the language remains the same!

5. rules of the form $A \to X_1X_2...X_n$ with $n > 2$ are now replaced by
$A \to X_1Y_1, \quad Y_1 \to X_2Y_2, \quad ..., \quad Y_{n-2} \to X_{n-1}X_n$

... and by now, you know what to do.

Do we have a grammar in Chomsky Normal Form now? ■

The Chomsky Normal Form has several advantages: we can see immediately whether the language contains the empty string; every parse tree is (almost) a full binary tree (that will turn out to be handy in the pumping lemma). On top of that, every derivations of a string of length $n > 0$ has length $2n - 1$: that will be useful when we study decision problems related to CFLs.

There exist other normal forms for CFGs. As an example, the Greibach[10] normal form restricts grammars to rules of the form

- A $\to$ aX
- $S \to \epsilon$

[10]Sheila Greibach

in which X is a (possibly empty) sequence of nonterminals and a a terminal. Now, derivations are no longer than the string![11] Try proving the analogue of the theorem on page 106 but now for the Greibach normal form.

Selfie:

Prove that a derivation of a string of length $n > 0$ from a grammar in Chomsky normal form has length $2n - 1$.

Prove that a derivation of a string of length $n > 0$ from a grammar in Greibach normal form has length n.

Can you use the transformation to Chomsky normal form to understand that every (pure) Prolog program can be transformed to an equivalent Prologprogramma with in each clause body zero or two goals?

[11] Careful, do not conclude that one can make the derivation of a given string in linear time!

The pushdown automaton

An FSA has no memory besides its state, and since there is only a finite number of them, an FSA can not count in an unlimited way. As a result, it is incapable to determine lots of interesting languages. The next machine we introduce - the pushdown automaton (PDA) - has an unlimited memory, but it can use that memory in a very limited way only: the memory is organised as a stack, and at any given moment, only the top of the stack can be inspected or changed. The latter happens by adding an element (push) or deleting it (pop). As with the regular automata, we use PDAs as a means to decide whether a string is accepted: successive characters from the string are consumed, and based on the current state and the current top of the stack, the machine goes to a new state, and possibly changes the top of the stack. Just as with regular automata, we define the PDA from the start as non-deterministic.

Definition 3.20.1. *The pushdown automaton*
A pushdown automaton is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_s, F)$ in which

- Q is a finite kset of states
- Σ is a fine inputal phabet
- Γ is a finite stack alphabet
- δ is a transition function with signature $Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \mathcal{P}(Q \times \Gamma_\epsilon)$
- q_s is the start state
- $F \subseteq Q$ is the set of (accepting) end states

This definition does not specify how a PDA works. Let us first look at a graphical representation of a PDA, similar to the graphical representation for an FSA. As an example, we take PDA $(Q, \Sigma, \Gamma, \delta, q_s, F)$ with

- $Q = \{q_s, q_f, x, y\}$
- $F = \{q_f\}$
- $\Sigma = \{a, b, c\}$
- $\Gamma = \{m, \$\}$

The δ can be seen in Figure 3.21.

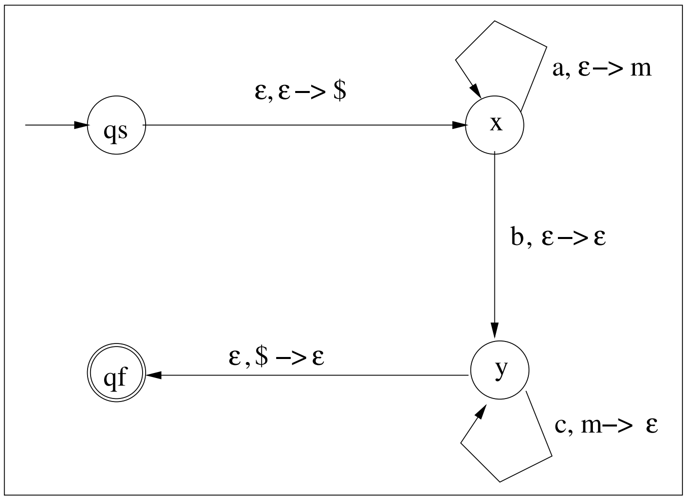


Figure 3.21: A pushdown automaton

The meaning of an edge label like $\alpha, \beta \to \gamma$ is

- if α is the first symbol of the current string
- and β is at the top of the stack
- then follow the edge and
  - remove α from the string
  - remove β from the stack
  - put γ on the stack

α, β and γ may be ϵ. For α this means: do not take the beginning of the string into account; for β it means: do not take top of the stack into account; for γ: do not push anything.

As you see, the labels specify δ.

The first time we arrive at y after starting in q_s with string a^2bc^2 and empty stack, the stack contains $mm\$$, and the string is reduced to c^2. Applying δ two more time exhausts the string and a final application of δ brings us in the end state. We say: a^2bc^2 is accepted by the PDA, or a^2bc^2 is in the language determined by the PDA. Here is the formal definition:

Definition 3.20.2. *A string accepted by a PDA*
A string s is accepted by a PDA if s can be split in parts w_i, i= 1..m ($w_i \in \Sigma_\epsilon$), and there exisat states q_j, j= 0..m, and stacks $stack_k$, k=0..m ($stack_k \in \Gamma^*$), so that

- $stack_0 = \epsilon$ (the stack is empty at the start)
- $q_0 = q_s$ (the first state is the start state)
- $q_m \in F$ (we arrive in an end state with an empty string)
- $(q_{i+1}, y) \in \delta(q_i, w_{i+1}, x)$ in which $x, y \in \Gamma_\epsilon$ and
  $stack_i = xt, \quad stack_{i+1} = yt$ with $t \in \Gamma^*$

The last bullet indicates that the transitions follow δ.

Caveat: Acceptance as defined above, does not take the contents of the stack into account at the moment of arriving in an end state: the string being empty is enough. There exist many alternative definitions for the PDA: some allow to push more than one symbol during a transition; some define acceptance as *the string and the stack are both empty*, in which case the notion of end state is not even needed; some define acceptance as *the string and the stack are both empty and we are in F*; some require that at each transition either a symbol is pushed or popped, but not both; some have only one end state ... All these are equivalent in terms of the set of languages that can be determined by these machines. So, do not get to much hung up by the exact definition, but better stick to a particular one while doing a proof or an exercise.

Definition 3.20.3. *The language determined by a PDA*
The language determined by a PDA is the set of strings accepted by the PDA.

Clearly, the language $\{a^n b c^n | n \geq 0\}$ is determined by a PDA: Figure 3.21 shows the implementation. Strings like aabc are not accepted.

Switching between alternative definitions of the PDA can be tricky. Look at the PDA defined in Figure 3.22: at first sight you might think it accepts the same language as in Figure 3.21, but is that really so?

It is only true of for this PDA you define acceptance as *arrive in the end state with an empty stack and string*. So you see that the details of the definition matter for a given PDA. On the other hand, the details of the definition change nothing to the class of languages determined by PDAs.

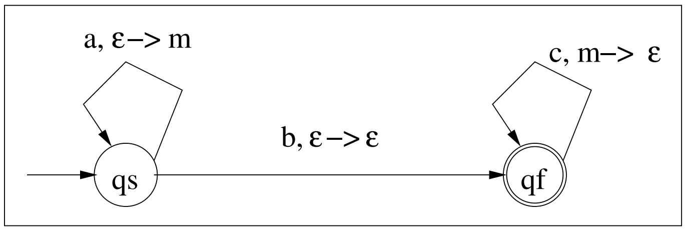


Figure 3.22: Another pushdown automaton for the same language?

Equivalence of CFG and PDA

We want to prove

Theorem 3.21.1. *Every PDA determines a CFL and every CFL is determined by a PDA.*

Proof. There are clearly two parts in this theorem. We prove the first part in the lemma on page 114, and the second part in the lemma on page 115. ■

We first need some preparatory work and an example.

We mentioned before that allowing PDAs to push more than one symbol during a transition does not change the power of PDAs: if you haven't done that yet, this is a good moment to prove it! We use this feature because it makes the description of the example much shorter.

Example 3.21.2. *Consider again the CFG Arit1 of page 103*

- $E \to\ E + E \qquad E \to\ E * E \qquad E \to\ a$

with E the start symbol. Now look at the PDA in Figure 3.23.

This PDA has a bit more symmetry than you would get starting from another CFG. Still, it is constructed in a systematic way as follows:

- *it has 3 states: the start state q_s, the end state q_f, and one* helper *state x*
- *there is one edge from q_s to x: it ignores the string and the stack; it pushes a marker \$ and the start symbol on the stack*
- *there is one edge from x to q_f: it consumes nothing from the string and it takes the marker \$ from the stack*

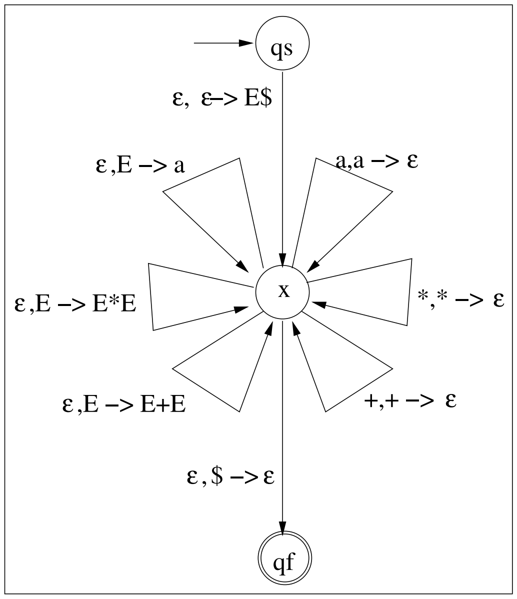


Figure 3.23: A PDA derived from Arit1

- *all other edges connect x with x; their labels correspond to*
  - *the symbols of Σ: for each $\alpha \in \Sigma$, there is an edge with label $\alpha, \alpha \rightarrow \epsilon$; the meaning of these edges is: if the top of the stack is the same symbol as the first symbol of the string, consume both*
  - *the grammar rules: for each rule $X \rightarrow \gamma$ there is an edge with label $\epsilon, X \rightarrow \gamma$; the meaning of these edges is: if the top of the stack is a nonterminal X, replace it by γ; γ is sequence of terminals and nonterminals*

Selfie: Describe exactly Σ and Γ for this PDA.

We use the PDA to parse string $a + a * a$: Table 3.3 shows the stack and the string during during parsing. Note that when we push the sequence of symbols XYZ, we actually want them in reverse order, so the representation of the stack is a string to which we add at the front (without reversing the XYZ) and from which we pop from the left.

The above parse is a leftmost derivation. It is not necessarily unique: choose E+E instead of E∗E at the second step, and you will still accept.

Is it possible that you get stuck? Try E*E instead of E+E at the 3th step, and you will see. Does that indicate a problem?

Convince yourself that L_{Arit1} equals the set of strings accepted by the PDA in Figure 3.23.

string	stack	state
a+a∗a	ϵ	q_s
a+a∗a	E\$	x
a+a∗a	E∗E\$	x
a+a∗a	E+E∗E\$	x
a+a∗a	a+E∗E\$	x
+a∗a	+E∗E\$	x
a∗a	E∗E\$	x
a∗a	a∗E\$	x
∗a	∗E\$	x
a	a\$	x
ϵ	\$	x
ϵ	ϵ	q_f

Table 3.3: Parsing $a + a * a$

We can generalize the example to the following lemma:

Lemma 3.21.3. *The construction of a PDA from a CFG as described above, results in a PDA that accepts the language L_{CFG}.*

Proof. You can check that there is a one-to-one correspondence between a derivation of string s using the CFG, and an accepting execution of the PDA for s. ■

The above construction results in a nondeterministic PDA whenever the CFG has two (or more) rules for a particular nonterminal. This might give the impression that the grammar is ambiguous, but is not true: grammar Arit2 on page 105 also results in a nondeterministic PDA, but Arit2 is not ambiguous. Still, you could wonder about the following questions

- if L has a deterministic PDA, then L is not ambiguous
- a nondeterministic PDA can be transformed to an equivalent deterministic one

The construction of a PDA from a CFG is so easy: you need only 3 states and there is a very uniform description of its transitions. The reverse direction - from a PDA to a CFG - is disappointingly more complicated, partly because not every PDA has 3 states. Also, our DFA-to-RE method using the GNFA seems not to work here (why not?).

We will assume that the PDA has a particular form

- there is only one end state
- the stack is being emptied before we arrive there
- each transition pops one symbol from the stack, or pushes one symbol onto the stack, but not both

We mentioned this form earlier - now is a good time to proof that it is not restrictive.

Construction of a CFG (V, Σ, R, S) from a PDA $(Q, \Sigma, \Gamma, \delta, q_s, \{q_f\})$:

- $V = A_{p,q}$ with $p, q \in Q$
- $S = A_{q_s,q_f}$
- R has 3 parts
  - rules of the form $A_{p,p} \to \epsilon$ for each $p \in Q$
  - rules of the form $A_{p,q} \to A_{p,r}A_{r,q}$ for each $p, q, r \in Q$
  - rules of the form $A_{p,q} \to aA_{r,s}b$ in which
    $p, q, r, s \in Q, \quad a, b \in \Sigma_\epsilon, \quad t \in \Gamma, \quad (r, t) \in \delta(p, a, \epsilon), \quad (q, \epsilon) \in \delta(s, b, t)$

The intuition behind this construction is this: the strings that bring you in the PDA from state p with empty stack to state q with empty stack, are exactly the strings generated by the nonterminal $A_{p,q}$.

Lemma 3.21.4. *The above construction of a CFG from a PDA preserves the language.*

Proof. We don't do the proof this year :-) ■

Consequence I: a language accepted by a PDA is context free.

Consequence II: every regular language is context free. The reason is that an FSA is a also a PDA: it just ignores the stack.

Selfie: Another way to understand that regular languages are context free: construct a CFG for the given regular language.

To conclude (for now) we mention two more theorems that provide better understanding of the structure of CFLs.

A theorem by Chomsky-Schützenberger states that every CFL is essentially the intersection of a regular language with a *Dyck*[12] *language*: a Dyck language consists of balanced strings of parentheses, possibly more than one.

Parikh's theorem can best be formulated using the transformation Ord of a language: you obtain Ord(s) by putting the symbols of s in alphabetic order. As an example $Ord(bacabbc) = aabbbcc$. Parikh states: for each CFL L, there exists a regular language R so that $Ord(L) = Ord(R)$. I.e. apart from the order of the symbols in their strings, regular and context free languages are the same!

[12]Walther von Dyck

A pumping lemma for Context Free Languages

Theorem 3.22.1. *For every context free language L, there exists a number p (the pumping length), so that for each string $s \in L$ with $|s| \geq p$, there exists a partition of s in 5 pieces u,v,x,y and z from Σ^* so that $s = uvxyz$*

1. $\forall i \geq 0 : uv^ixy^iz \in L$
2. $|vy| > 0$
3. $|vxy| \leq p$

Proof. We use once more the pigeon hole principle, nnow on the parse tree for strings that are long enough. Consider a CFG in Chomsky normal form for L. Let n be the number of nonterminals. ■

If $d \in L$, it has a parse tree. By deleting from that tree the leaves, we get a full binary tree - because of the Chomsky normal form. The height of this tree is at least $log_2(|s|)$, so the longest (simple) path from the root has at least $log_2(|s|) + 1$ vertices. If we chose s long enough, $log_2(|s|)+1 > n$ and consequently, on this longest path at least one nonterminal - say X - is repeated. Consider the occurrence of X that is closest to the root, and denote it by X_2; denote by X_1 the closest re-occurrence on that path. Note that X does not equal S (why?). Figure 3.24 illustrates the situation. From the parse tree we can construct a derivation of which we show only some steps:

$S \Rightarrow^* uX_2z \Rightarrow^* uvX_1yz \Rightarrow^* uvxyz$ (a)

In this derivation u,v,x,y,z are strings from Σ^* and moreover v and y are not both empty, because that would mean that X could derive itself, which is impossible because of the Chomsky normal form.

Since (a) is a good derivation, also

$S \Rightarrow^* uX_2z \Rightarrow^* uxz$

is, and

$S \Rightarrow^* uXz \Rightarrow^* uvXyz \Rightarrow^* uvvxyyz$

as well and ...

So we already have obtained (1) and (2) from the theorem if we take strings longer than $2^{(n-1)}$: that is our pumping length p.

We now prove (3): vxy is derived from X with a parse tree smaller than n, so it has at ost 2^{n-1} leaves which correspond exactly with vxy. Figure 3.24 once more illustrates this

■

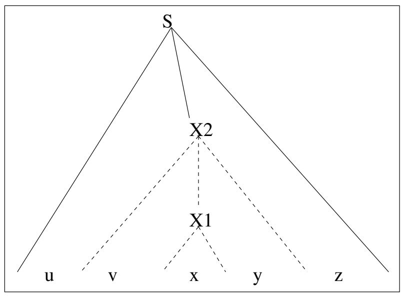


Figure 3.24: The parse tree with the repeating nonterminal X

Using the pumping lemma for CFLs

Consider the language $L = \{a^n b^n c^n | n \geq 0\}$. Suppose that there exists a pumping length p. We must take a string s longer than p, say $s = a^p b^p c^p$. Suppose that $s = uvxyz$ with $|vy| > 0$. Then there are two possibilities:

1. v has the form α^k and y has the form β^l and α and β are in $\{a, b, c\}$; in this case we have that $k + l > 0$. Now, uv^2xy^2z does not have the same number of a's, b's en c's
2. v or y contains more than one symbol from $\{a, b, c\}$; then v^2 or y^2 contains those symbols in the wrong order, so uv^2xy^2z does not belong to the language

As a consequence, s can not be pumped and so L is not context free.

An algebra of CFLs?

The union of two CFLs determined by CFGs CFG_1 and CFG_2 can be described easily by the following construction: first rename apart the nonterminals of the two given CFGs. Let the start symbols of the two grammars be S_1 and S_2. Construct the grammar for the union by the union of the two rule sets, adding the rule $S_{new} \to S_1|S_2$, and consider S_{new} as the start symbol of the union grammar. We can conclude that CFL is closed under union.

How about the intersection of two CFLs? We try an example: take as terminals $\{a, b, c\}$ and define $L_1 = \{a^n b^n c^m | n, m \geq 0\}$, $L_2 = \{a^n b^m c^m | n, m \geq 0\}$. Clearly, the L_i are context free[13]. The intersection of the L_i is the language $\{a^n b^n c^n | n \geq 0\}$ and we proved earlier in Section 3.22.1 that this language is not context free. So, the intersection of context free languages is not necessarily context free.

We can now also conclude that the complement of a CFL need not be context free, because $A \cap B = \overline{(\overline{A} \cup \overline{B})}$.

Finally, let L be context free and A regular, then we can wonder

- is $L \cup A$ context free/regular?
- is $L \cap A$ context free/regular?

What do you think? What can you prove or show with examples?

Selfie: Let $\Sigma = \{a, b\}$. The language $\{ss|s \in \Sigma^*\}$ is not context free: give a proof. Also show that the complement of this language is generated by the following context free grammar:

- $s \to$ AB | BA | A | B
- A $\to$ CAC | a
- B $\to$ CBC | b
- C $\to$ a | b

[13]Make a CFG for them!

Ambiguity and determinism

A bit more detail about the link between inherent ambiguity of a context free language and its determinism. We denote by DCFL the set of context free languages that have a deterministic PDA (DPDA).

An ambiguous language cannot be deterministic: there is a strong connection between a derivation and an accepting path in a PDA. It follows that the members of DCFL have a non-ambiguous grammar.

The converse it not true: there exist non-ambiguous non-deterministic languages. One standard example is the language $\{s\hat{s}|\text{s}\in \{a,b\}^*\}$ ($\hat{s}$ means the reversed string). A non-ambiguous grammar for this language is

$$s \rightarrow \text{ aSa} \mid \text{bSb} \mid \epsilon$$

but a PDA does not *know* in advance where the middle of the string is, so it must *guess* that. That is exactly the essence of the non-determinism needed to parse this language with a PDA. That does not imply that there is no deterministic parser at all for this language: you can write one easily in a language like Java. Only, it cannot be done with a deterministic PDA.

There is another way to construct examples of non-deterministic languages: one uses the fact that the complement operation is internal to DCFL[14]. Now consider $L = \{ss|\text{s}\in \{a,b\}^*\}$. Use the pumping lemma to prove that L is not in CFL. But is context free: page 119 contains a CFG for $\overline{L}$. Consequently, $\overline{L}$ is not deterministic.

Examples of non-deterministic languages are often obtained by taking the union of two partially overlapping CFLs. As an example:

$L = \{a^m b^n c^k | m \neq n\} \cup \{a^m b^n c^k | n \neq k\}$ is the union of two DCFLs.

Suppose that $L \in DCFL$, then its complement would be in DCFL, and also the intersection of $\overline{L}$ with the regular language $\{a^*b^*c^*\}$. But that results in $\{a^n b^n c^n | n \in \mathbb{N}\}$ and that is not even context free!

One more example of a non-deterministic language and a proof sketch that shows it: $L = \{a^n b^n | n \in \mathbb{N}\} \cup \{a^n b^{2n} | n \in \mathbb{N}\}$.

[14]For a proof, see for instance the book by Kozen

Suppose that a DPDA M determines L. M has a structure as in Figure 3.25.

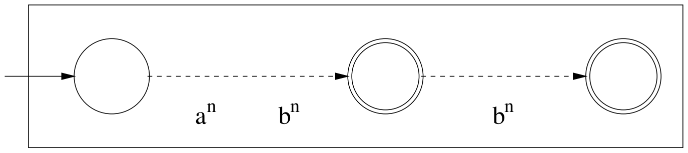


Figure 3.25: The DPDA M for L

Replace in the last part of M every b by a c, and change the left accepting state into an ordinary state. You obtain the machine in Figure 3.26.

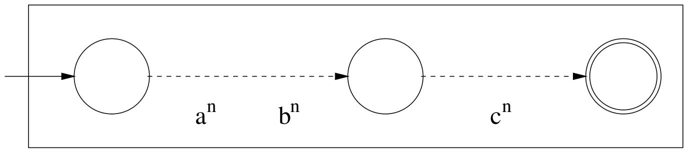


Figure 3.26: The DPDA for L'

This PDA accepts $L' = \{a^n b^n c^n | n \in \mathbb{N}\}$ but we know already that this language is not context free!

The book by Linz contains a variant with more details.

The main reason for going into this, is that unlike for regular languages and their FSAs, determinism makes a huge difference for CFLs and their PDAs: one step higher in the Chomsky hierarchy, non-determinism is suddenly very important.

Practical parsing techniques

We could be confronted with the following problem: given the specification of a language, construct a parser for it. Perhaps this language is the allowed input in a form to be filled out, and possibly what you are allowed to input in a particular place depends on what has been filled out earlier in the form. For instance, if first you filled out you have 2 children, you should not give the names of 3 children later on. Or the language could be something with an intricate structure, like for instance a programming language: nested if-then-else, statement blocks, arithmetic expressions, package qualifications ... Often this programming language is context free, and one usually gets (or makes) the description of this language in two levels: the first level is lexical (see page 93), the second level is syntactic. One uses the BNF-notation: BNF is for Backus[15]-Naur[16] form. It was used for the first time for Algol[17] [18]. BNF is very close to CFG and we already used it for the Stat grammar on page 101.

Just as flex can produce an efficient lexer starting from a regular expression, other tools convert a CFG to an efficient syntax analyser. The general construction of a PDA from a CFG is not good enough: it almost invariably results in a non-deterministic parser. Making it deterministic is not easy, and in general even impossible as we know in the mean time. Therefore, these parser generators impose some restrictions on the kind of languages/grammars they want to deal with.

Well known parser generators are Bison (it followed Yacc) that produces C, and ANTLER (Java). The practical use of those tools is beyond the scope of this course, but you should know they exist, so that if you even need a PDA, you remember that tools can do it for you.

The 3-state PDA we constructed from a CFG gave us leftmost topdown derivations: the parse tree is buildstarting from the start symbol downwards. One could realise that also as a deterministic program with a *recursive descent parser*. It contains a number of mutually recursive procedures that correspond to the nonterminals of the grammar. That works only for grammars of the type *LL(k)*: the L means that the input is read from Left to right; the second L tells us that it allows a Leftmost derivation, with k symbols as *look-ahead*.

There are other ways to implement a parser. Let us still read from left to right, and stack what we read. As soon as we have read something that matches the right side of a rule, we use that rule. Hier is a worked out example for the input $a + a * a$.

The actions correspond to what a shift-reduce parser does: either it pushes the first input symbol on the stack and shifts the input pointer one position, or it reduces what is on

[15]Turing award in 1977
[16]Turing award in 2005
[17]Look up what Algol is and how important it was and still is for programming language design!
[18]Lots of historical commotion: who deserves credit ... wikipedia tells you a lot

seen	to be read	action to take
	a+a∗a	shift
a	+a∗a	reduce
E	+a∗a	shift
E+	a∗a	shift
E+a	∗a	reduce
E+E	∗a	reduce
E	∗a	shift
E∗	a	shift
E∗a		reduce
E∗E		reduce
E		stop

Table 3.4: Bottom-up parsing of $a + a * a$

top of the stack, using a grammar rule. Such a machine needs also an internal state (not shown here) used to decide at each point which action to take. The decision can depend on the first k non-shifted input symbols as well. Such parser constructs the rightmost parse. The corresponding grammar must be LR(k).

Lots of research has focused on tools for the construction of efficient parsers, e.g. for compiler front ends, for XML document analysis ... Additionally, these tools are capable of error correction/recovery/reporting, generation of syntax directed editors, ... all on the basis of grammars.

Context Sensitive Grammar

The step from regular expressions to context free grammars involved recursion of the nonterminals. The main restriction on a CFG was that the lef hand side of a rule could have only one symbol, a nonterminal.

The next layer of the Chomsky hierarchy belongs to the context sensitive languages, generated by context sensitive grammars: at the left in a rule, one may put an arbitrary string of terminals and nonterminals: one of the nonterminals is rewritten. As an example:

$$a\underline{X}Yz \rightarrow a\underline{bCD}Yz$$

is a context sensitive grammar rule: X can be rewritten to bCD only in the context of an a (at its left) and Yz (at its right). There are alternative, equivalent, definitions of when a rule is context sensitive, like for instance: $\alpha \rightarrow \beta$ only if $|\alpha| \leq |\beta|$.

At first, it is not clear whether there exist context sensitive languages that are not context free. Here is an example we studied before: let $L = \{a^n b^n c^n | n \in \mathbb{N}\}$. We know already that L is not context free. But it has a context sensitive grammar (according to the equivalent definition).

$s \rightarrow$ abc

$s \rightarrow$ aSBc

cB $\rightarrow$ Bc

bB $\rightarrow$ bb

Selfie : prove that the above grammar generates the above L.

There exists a normal form for context sensitive grammars: the Kuroda normal form: each rule has one of the forms below.

AB → CD

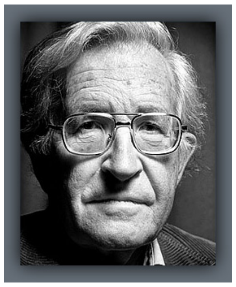
A → BC

A → B

A → a

A,B,C, and D are nonterminals; *a* is any terminal.

A CSL can be parser by a non-deterministic lineair bounded automaton (LBA). We introduce this class later: it is strictly stronger than the PDAs. LBAs are important because they solve decision problems in $O(n)$-space! This also indicates that there exists machinery even stronger than LBAs!



We have almost arrived at the top of the Chomsky hierarchy: by taking away the last restriction on the left hand side of the grammar rule, we get the *unrestricted grammars*. They generate languages that need Turing machines to be decided. They are introduced in the next chapter.

The hierarchy can be found schematically in Table 3.5: there exist quite a few refinements (for instance related to determinism), but they are not shown.

	Grammar	Language	Automaton
Type-0	unrestricted	recognisable	Turing machine
Type-1	context sensitive	context sensitive	lineair bounded
Type-2	context free	context free	pushdown
Type-3	regular	regular	finite

Table 3.5: The Chomsky hierarchy

Chomsky is also known for his political activism - it is worth to read about it.

Chapter 4

Languages and Computability

The Turing Machine as a Decider or Recognizer

In the previous sections, we looked at two classes from the Chomsky hierarchy. In particular we studied the machines that decide the membership problem for these languages: does a given string belong to a given language. We already understand that some (many?) languages are beyond the decision power of DFAs, PDAs, and even LBAs. So it is time to start studying machines that can do more. We use Turing machines.

Roughly speaking, a TM has a two infinite (unlimited would be enough) tape consisting of squares (or cells) with one symbol each, a control unit, a tape head that can read the symbol in a square, and write a new one. The action of the machine depends on the internal state of its control unit and the symbol being read. Possible actions are: writing a symbol and moving the tape head. Figure 4.1 shows the parts of the machine and the actions that are described by a function δ.

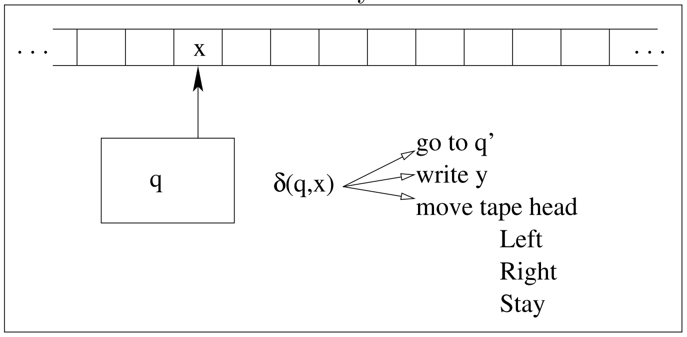


Figure 4.1: Schema of a Turing machine

More formally:

Definition 4.1.1. *Turing Machine*
A **Turing Machine** is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_s, q_a, q_r)$ with Q, Σ, Γ finite sets:

- Q is the set of states
- Σ is an input alphabet not containing #
- Γ is the tape alphabet; $\# \in \Gamma$ and $\Sigma \subset \Gamma$
- q_s is the start state
- q_a is the accepting end state
- q_r is the rejecting end state and different from q_a
- δ is the transition function: it is a total function with signature
  $$Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}$$

The machine is initialized as follows:

- the symbols of a string from Σ^* is put in subsequent tape cells: they constitute the input string; all other cells contain #
- the machine is put in start state q_s
- the tape head is positioned at the leftmost symbol of the input string; if the input string is empty, it can be positioned anywhere

The machine works as follows: on the basis of the state of the machine, and the symbol under its head, δ determines the next state of the machine, which symbol needs to be written, and the direction of the head movement. This is repeated until

- the machine arrives in the state q_a: the input string is accepted and sometimes, the contents of the tape at this moment is considered the result of the computation
- the machine arrives in the state q_r: the input string is rejected
- neither of the above, and it continues ... the machine *loops* or is in an infinite loop

As with DFA and PDAs, there are alternative definitions of TMs, or variants. Here are a few that do not seem to matter: they define *equivalent* machines. You can prove that, but you must start by defining precisely what you mean by equivalent.

- the tape head can not stay at the same place: it must at every step move left of right

- there can be more than one accepting or rejecting state
- the tape is unlimited in only one direction (it is half-infinite)
- there is more than one tape
- the input alphabet has only two symbols
- there are only three states
- the machine has two stacks instead of a tape

About the symbol #: initially it indicates that a tape cell has not been visited or written before, but the machine may write that symbol any time. Often, texts refer to it as the blank symbol and use ⊔. We often refer to # as the blank symbol as well.

For a given TM, Σ^* consists of three disjunct subsets:

1. the strings accepted by TM: L_{TM}
2. the strings for which TM loops: ∞_{TM}
3. all other strings (they are rejected)

These subsets are used in the following definitions.

Definition 4.1.2. *To recognize*
A Turing machine TM recognizes L_{TM}

Its complementary definition is

Definition 4.1.3. *Turing recognizable language*
A language L is Turing recognizable if there exists a Turing Machine TM so that TM recognizes L or alternatively, $L = L_{TM}$

Here is an example of a recognizable language L and a TM that recognizes L: $\Sigma = \{a, b\}$, $L = \{a\}^*$. The next table describes δ:

∞_{TM} is not empty: every string not in L brings the machine in an infinite loop. But we can also construct a TM for L that always halts:

This distinction is captured in the following definitions:

Definition 4.1.4. *To decide*
A Turing Machine TM decides a language L, if TM recognizes L and $\infty_{TM} = \emptyset$

Q	Γ	Q	Γ	LRS
q_s	a	q_s	#	R
q_s	b	x	b	S
q_s	#	q_a	-	-
x	a	x	a	S
x	b	x	b	S
x	#	x	#	S

Table 4.1: A TM recognizing $\{a\}^*$

Q	Γ	Q	Γ	LRS
q_s	a	q_s	#	R
q_s	b	q_r	-	-
q_s	#	q_a	-	-

Table 4.2: A TM that decides $\{a\}^*$

Definition 4.1.5. *Turing decidable language*
A language L is Turing decidable if there exists a Turing Machine that decides L.

It might not be clear at this point that to recognize and to decide are two very different things, but we will work towards this understanding. Still, it should be clear that a decidable language is also recognizable.

Other terminology for *decidable language* is *recursive language*, and for *recognizable language*, one uses *recursively enumerable*, or sometimes *semidecidable*.

Definition 4.1.6. *Co-recognizable/co-decidable*
A language L is **co-recognizable/co-decidable** if $\overline{L}$ recognizable/decidable is.

Theorem 4.1.7. *If L is decidable, then L is also co-decidable.*

Proof. Let M be the TM deciding L; now switch the roles of q_a and q_r. ■

Theorem 4.1.8. *If L is recognizable and co-recognizable, then L is decidable.*

Proof. Let M1 be the machine that recognizes L, and M2 the machine recognizing $\overline{L}$. The idea is to have a machine M that lets M1 and M2 run in parallel: as soon as M1 accepts, accept; as soon as M2 accepts, reject. M1 and M2 can not accept both, and for any given string, at least one of them halts in the accepting state. This M decides L.

The definition of M is informal, and we should make it more formal: define M as a 2-tape machine and work out the details. ■

Theorem 4.1.9. *Some languages are not recognizable.*

Proof. The set of Turing Machines is countably infinite, so the set of recognizable languages can be at most countably infinite. On the other hand, the set of languages (being $\mathcal{P}(\Sigma^*)$) is uncountably infinite. So there must exist languages that are not recognizable. In fact, most languages are not recognizable. ■

What is our plan with Turing Machines?

- we want to explore further the difference between deciding and recognizing
- fit into this picture the regular and context free languages
- study examples of languages that can not be decided or recognized by a TM and general techniques for proving this

Selfie: Comment on the following sentences:

- every string is recognizable
- every string is decidable
- every finite language is decidable/recognizable
- the union/intersection of two recognizable languages is recognizable
- the intersection of a recognizable and a decidable language is decidable

A graphical representation of Turing Machines

Just as FSAs and PDAs, TMs can be visualized by a graph: the vertices are the states. Figure 4.2 illustrates it for the TM deciding the language $\{0^n1^n | n \geq 0\}$.

Each state has as many outgoing edges as elements in Γ (some parallel edges are treated together) - except for the start and end state of course. The label represents the δ of the

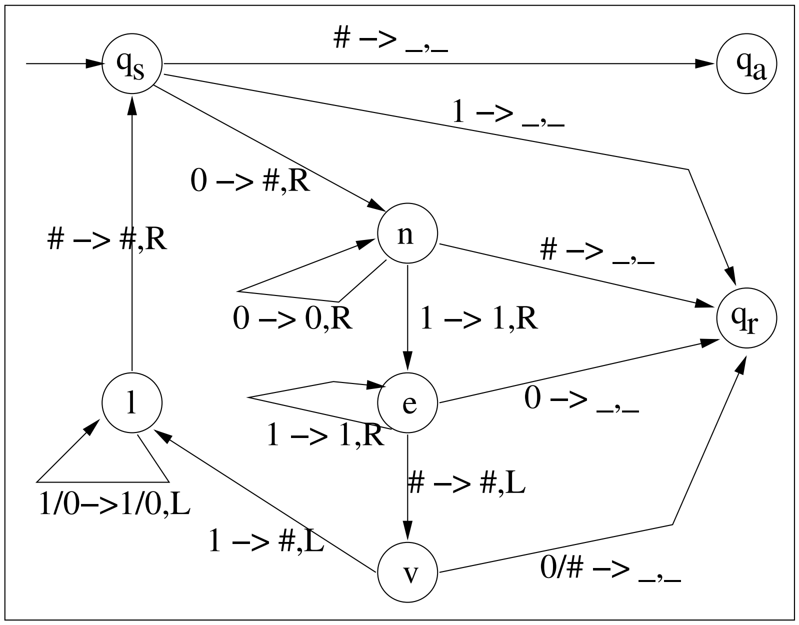


Figure 4.2: Graphical representation of a Turing Machine

machine. We use _ to indicate that the value at this place is irrelevant: this is typical for transitions to a halting state.

One can describe in words what the machine does with an input consisting of only zeroes and ones.

- if in the start state, the head sees a
  - #: accept
  - 1: reject
  - 0: wipe it out, go to the right, and switch to state n; it will skip all zeroes
- if in state n, the head sees a
  - #: reject
  - 1: go right to state e; it will skip all ones
  - 0: go right
- in state e
  - 1: go right

  - 0: reject
  - #: go left, switch to v; it will erase one 1

- in state v
  - 0: reject (but actually can't occur)
  - #: reject (but actually can't occur)
  - 1: erase the 1 (overwrite it by #); go left on the tape and to state l; it will find the leftmost symbol of the string

- in l
  - 0/1: go left;
  - #: go right and to the start state

As you see, every state has its little something to contribute to the whole program.

The TM in Figure 4.2 proves that $\{0^n 1^n | n \geq 0\}$ is a decidable language.

Representation and simulation of TM computations

It is often handy to be able to compactly represent a *configuration* of a TM: it is like a computer memory dump, together with all registers and the hardware state, so that execution could restart from that information. For a TM, we can do that because at each moment, only a finite part of the tape is in use, i.e. is different from #. So, a string of the form

$$\alpha q \beta$$

can represent the tape containing $\alpha\beta$ with to its left and right only #; moreover, the machine is in state q and the head is positioned under the leftmost symbol of β. α and β may contain #.

A start configuration is always of the form $q_s\alpha$, and an end configuration is either $\alpha q_a \beta$ or $\alpha q_r \beta$.

During the execution of the TM, two successive configurations are related by δ as follows:

$$\alpha q b \beta \rightarrow \ \alpha p c \beta \text{ if } \delta(q, b) = (p, c, S)$$

$$\alpha a q b \beta \rightarrow \ \alpha p a c \beta \text{ if } \delta(q, b) = (p, c, L)$$

$$\alpha q b \beta \rightarrow \ \alpha c p \beta \text{ if } \delta(q, b) = (p, c, R)$$

You can add some transitions for the case that α or β is empty: you might need to generate extra occurrences of #.

A sequence of configurations from a begin to an end configuration - following δ - is called a *computation history.*

It is reasonably easy to see that the computation of any TM can be simulated by a program, for instance in Prolog. To make that more concrete, here is some Prolog code. A configuration $\alpha q \beta$ is represented as a term with principal functor conf/3, as follows:

$conf([a_n, ..., a_2, a_1], q, [b_1, b_2, ..., b_m])$, where $a_1 a_2 ... a_n = \alpha$ en $b_1 b_2 ... b_m = \beta$ and $a_i, b_j \in \Gamma$.[1]

Here is part of the Prolog program working with configurations:

```
onestep(conf(Alpha,Q,[]), NewConf) :-
          onestep(conf(Alpha,Q,[#]), NewConf).
onestep(conf(Alpha,Q,[B|Beta]), conf(Alpha,P,[C|Beta])) :-
          delta(Q,B,P,C,stop).
```

Selfie: Extend the above program, and explain in words the function of each clause.

We should also understand that a TM can simulate a Prolog program: to do that directly would be a lot of work, so here is a classical road towards the same goal. First prove that Prolog can be implemented on an Intel machine - that has been done many times, e.g. by the SWI Prolog implementation. Next prove that an Intel machine can be simulated by a register machine; the register machine is in the first place a theoretical construct, but only a small step away from the TM.

That closes the circle: if a TM can simulate any X-program, and an X-program can simulate any TM, it means that X and the TM have the same computational power. Careful though: it says nothing about complexity, just about which languages can be recognized or decided!

We can often use the equivalence between TMs and more recent programming formalisms.

Selfie: Composition of Turing Machines We often need to combine a number of Turing Machines $TM_1, TM_2 ... TM_n$ into a new Turing Machine TM. We use words like

TM calls TM$_1$ as a subroutine

en

when TM_1 stops in its accepting end state, let TM_2 run on s

Make sure you know what this means, i.e. formalize it to some extent.

[1]Look up *Huet's Zipper*

Non-deterministic Turing Machines

We mention non-deterministic TMs for no other reason than that every book does it: as far as computability goes, non-determinism for TMs has little interest. They are however important in the context of complexity.

The signature of δ is adapted for non-determinism to

$$Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R, S\}),$$

i.e. from a given configuration possibly more than one new configuration can be reached. Two would have been enough actually - try to understand why. A string can have more than one computation history. A string is accepted if there exists an accepting computation history. That is similar to the notion of acceptance in an NFA: one accepting path is enough to be accepted.

A non-deterministic Turing Machine can be simulated by a deterministic one. Some books actually describe such a simulation. For us, it is more natural to reason as follows: the Prolog program given earlier can be meta-interpreted according to iterative deepening (of breadth-first), and as soon as it arrives in an end state, it stops with the same result. So, if we can simulate a NDTM in Prolog, we can with a DTM.

Languages, $\mathcal{P}(\mathbb{N})$, properties and terminology

We have defined the notions recognizable/decidable in terms of languages over an alphabet. It is equally common to define those concepts in the context of subsets if $\mathbb{N}$. Those two views are equivalent, because $\mathbb{N}$ can be seen as $\{0, 1\}^*$: it would be a binary notation of natural numbers (but another one would be ok as well). So, a part of $\mathbb{N}$ is a language. Commonly, one reserves the term *(recursive) enumerable* for such sets (or languages), and the terms *decidable* and *recognizable* for properties or decision problems. The distinction is not so important: let P be a property that elements of a set V might or might not have; we can define the subsets $\{x | x \in V, x\ has\ property\ P\}$. It now becomes apparent that properties and subsets give an equivalent view on the issues.

The word *recursive* triggers often the image of a function (method, predicate, procedure ...) activating itself, or a data type defined in terms of itself (a list, a tree, ...). But the *recursive* in term *recursive enumerable*, it is not related to that.

Encoding

When we describe a language informally, there is no need to bother about an encoding. We can talk about *planar graphs* for instance, without thinking in terms of a particular alphabet. Still, when we want to describe an algorithm dealing with planar graphs, then we need an alphabet and a mapping from graphs to strings over that alphabet. Such a mapping is an encoding. Encodings need to be *reasonable*. An encoding must have at least the following three properties (related to encoding of graphs):

- one graph encodes to one string
- two *different* graphs encode to two different strings
- it must be decidable whether a string is the encoding of a graph and of which one

A reasonable encoding should not introduce extra information.

Some examples:

1. You are studying prime numbers. You chose a unary representation: the alphabet is $\{@\}$. You represent 7 by 7 times the symbol @, i.e. by @@@@@@@

   That is reasonable.

2. You are studying prime numbers. You chose a binary representation: the alphabet is $\{@, !\}$. You represent 9 by !@@!

   That is reasonable.

3. You are studying prime numbers. You chose a two-part representation of numbers: the first part indicates whether the number is prime or not; if it is, this first part equals p, otherwise n. The second part of the representation is the unary representation. Som 7 is represented as p@@@@@@@ and 9 as n@@@@@@@@@

   That is not reasonable: your encoding has extra information.

   Maybe you think that having more information is a good. For instance, you can now decide whether a number is prime by inspecting its first symbol only. You have a constant time check for primality!

   However, consider the input string p@@@@. The first letter tells you it is prime, but the rest of the string denies that ... In fact, the string does not represent a number, but you can't tell unless you check whether the sequence of @ has prime length ... So in fact, the extra information that seems to be there, needs to be checked. This check has the same price as if the extra information would not have been there.

As far as computability is concerned, the encodings in (1) and (2) above are equivalent: you can transform one into the other with an algorithm implemented on a TM.

As far as complexity theory is concerned, the two encodings are not equivalent! E.g. adding two number in unary notation is O(n) (with n the size of the numbers), while in binary encoding, it is O(log(n)).

We denote the encoding of an object S by $\langle S \rangle$. When we encode a sequence if objects S_i, we use $\langle S_1, S_2, ... \rangle$.

Selfie:

Find reasonable encodings for trees, XML-document, ...

Find some unreasonable encodings for the same objects, or for others.

For a function, say from $\mathbb{N}$ to $\mathbb{N}$, one can also encode the output. Your professor gave you as project to compute, on input i, the i^{th} prime number. You finished very quickly: at input i, you produced output i, arguing that i in the output is just the encoding of the i^{th} prime number. Your professor was not happy. Was that reasonable?

Universal Turing Machines

A TM can be described completely by its transition table (plus some conventions to identify the start and end states). One can encode the transition table, by using numbers for the states (from 1 to ...), numbers for the symbols, and a number for the L,R,S. These numbers would then be encoded in unary say, and one would just need a new symbol for delimiting the different numbers. We put the encoding of the transition table on the first tape of a UTM (the program tape). On its second tape (the memory), we write an input string (using the same encoding for the symbols). UTM may have even more tapes if that makes life easier.

UTM has its own program: it uses the program and memory tape as input, it simulates the behavior of the TM (whose δ is input) on the given string. As soon as TM would stop in its q_a or q_r, UTM makes a transition to its q'_a or q'_r.

We could provide more detail about the construction above, but for us, it is detailed enough: the Universal Turing Machine can simulate any (properly encoded) TM: UTM could even start by checking whether the tapes contain a proper encoding.

Do not get confused by one of the previous sections in which you were asked to think about a TM_1 that uses another TM_2 as a subroutine: in that case, the states of TM_2 are a subset of the states of TM_1. This is not the case in the UTM: the UTM has a fixed number of states, independent of the TM it is simulating, and the encoded states of the TM are not states of the UTM. As Guy L. Steele, Jr. said: *One person's data is another person's program.*

Is a UTM *big*? Claude Shannon showed one can trade states for symbols in a TM - see the exercise page 128. So it has become a bit of a habit to express the size of a TM as

$(|Q|, |\Gamma|)$, or even just the product $|Q| \times |\Gamma|$. In 1956 Marvin Minsky[2] constructed (7,4) UTM. In the mean time, it has been proven that a particular (2,3) machine is universal - but it is a bit controversial. BTW, (2,2) TMs cannot be universal.

[2]Turing award 1969

The Halting problem

Informally, the question is this: given a Turing Machine M, and a string s, is it possible to determine whether M stops on input s. We don't care whether M accepts or rejects s: in both cases, M stops. So, the question asks whether M goes into a loop or not. We want this question to be solved by an algorithm, so we actually want a Turing Machine H that **decides** the language $\{\langle M, s\rangle | M\ is\ a\ Turing\ Machine\ that\ stops\ on\ input\ s\}$. We denote that language by H_{TM}.

We will show that such a machine H does not exist, by showing that H_{TM} is undecidable. We already knew that such languages exist, but H_{TM} is our first concrete example.

A related problem is the acceptance problem for Turing Machines. The associated language is

$$A_{TM} = \{\langle M, s\rangle | M\ is\ a\ Turing\ Machine\ and\ s \in L_M\}$$

A_{TM} is the first language for which we prove that it is undecidable.

Theorem 4.8.1. *A_{TM} is not decidable.*

Proof. We prove this *by contradiction.*

Assume the existence of a decider B for A_{TM}. This means that on input $\langle M, s\rangle$, B accepts if M on input s stops in its q_a and rejects if M on input s stops in its q_r or loops. We write:

$B(\langle M, s\rangle)$ is accept if M accepts s and otherwise reject

We now construct a contraction machine C defined as

$C(\langle M\rangle) = \mathit{opposite}(B(\langle M, M\rangle))$ for any Turing Machine M.

We used: $\mathit{opposite}(\mathit{accept}) = reject$ and $\mathit{opposite}(reject) = \mathit{accept}$.

In the above definition, one of the possible M is C itself, and we get

$C(\langle C\rangle) = \mathit{opposite}(B(\langle C, C\rangle))$

Now, $C(\langle C\rangle) = \mathit{accept} \Leftrightarrow B(\langle C, C\rangle) = \mathit{accept} \Leftrightarrow \mathit{opposite}(B(\langle C, C\rangle)) = reject$

$\Leftrightarrow C(\langle C\rangle) = reject$

(make sure you understand and can explain every step above)

As a conclusion, C does not exist, and likewise for B. So, A_{TM} is undecidable. ∎

Theorem 4.8.2. *H_{TM} is undecidable.*

Proof. Suppose H_{TM} is decidable by the H. We define a decider B for A_{TM} as follows: on input $\langle M, s\rangle$:

- B runs H on $\langle M, s\rangle$
- if $H(\langle M, s\rangle) = accept$, then B lets M run on s and gives as result what M gives
- if $H(\langle M, s\rangle) = reject$ then B rejects the string $\langle M, s\rangle$.

Convince yourself that B decides A_{TM}, in particular that it does not loop.

Since a decider for A_{TM} is non-existent, H does not exist, so H_{TM} is undecidable. ■

Theorem 4.8.3. *H_{TM} is recognizable.*

Proof. On input $\langle M, s\rangle$, the recognizer H for H_{TM} simply runs M on s: if it stops (either end state), H accepts its input; otherwise it keeps running ... and that is ok. ■

Quite often, when we construct or define a recognizer, it has an accept, but no point at which it rejects.

Theorem 4.8.4. *A_{TM} is recognizable.*

Proof. Similar to the recognizer for H_{TM}. ■

As a direct consequence we have

Consequence 4.8.5. *$\overline{A_{TM}}$ and $\overline{H_{TM}}$ are not recognizable.*

Proof. This follows from Theorem 4.1.8 and the fact that A_{TM} is recognizable and not decidable. Same for H_{TM}. ■

The enumerator machine

The enumerator machine seems to be the machine as originally described by Alan Turing in his 1936 publication: he was interested in generating decimal expansions of em computable real numbers. The link with recognizable talen is quite direct. We had to wait introducing the enumerator until we dealt with the Halting problem: we will need it.

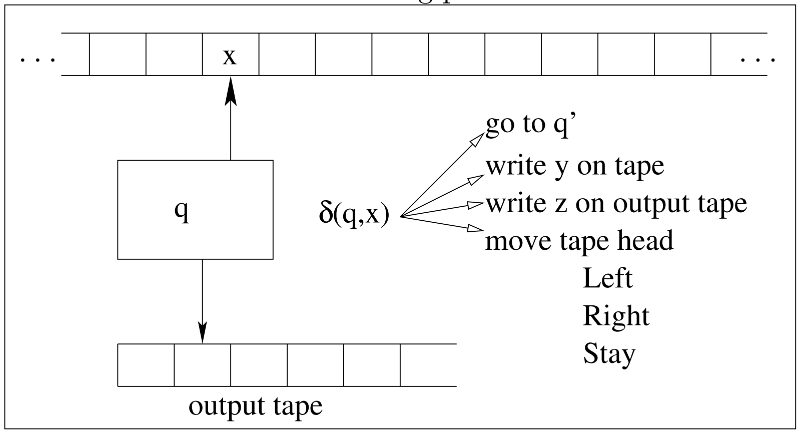


Figure 4.3: Schema of an enumerator machine

In Figure 4.3 you can see that the enumerator is like a Turing Machine with some extra's

- an enumerator state q_e
- an output tape
- an output marker

The signature of its δ is $Q \times \Gamma \rightarrow Q \times \Gamma \times \Gamma_\epsilon \times \{L, R, S\}$. The last Γ_ϵ is meant to be the symbol written at that transition to the output tape.

The machine start with an empty tape and empty output tape, in the usual q_s and starts working. Whenever something is written on the output tape, the write head moves one position to the right. Whenever the machine gets in state q_e, the output marker is written on the output, the machine goes to state q_s and the machine continues.

It is possible that the enumerator does not stop, and outputs one after another from a set of strings (separated by output markers): this set could be finite, or infinite (can it be uncountable infinite?). It is possible that the enumerator does not stop, and keeps working on a particular output string, maybe even the first one! It is also possible that the machine after a while stops and leaves on the output a (finite!) set of strings separated by output markers.

Whichever of the above scenarios, it makes sense to talk about the set of (finite) output strings produced by the enumerator: that is the language determined by the enumerator, or enumerated by the enumerator. The enumerator is allowed to produce the same output string multiple times. We can now prove:

Theorem 4.9.1. *The language enumerated by an enumerator is recognizable, and every recognizable language can be enumerated by an enumerator.*

Proof. (1) Given an enumerator Enu for L, we describe informally a recognizer M for L: M uses Enu as a subroutine as follows:

> On input s, M starts Enu. Any time Enu is in its q_e state, M inspects the most recently produced string on the output tape of Enu. If it equals s, M accepts s. Otherwise Enu continues.

(2) Suppose M recognizes L. We construct an enumerator Enu for L with the following ingredients:

- a TM_{gen} that given a number n puts on a tape the first n strings from Σ^*: $s_1, s_2, ...s_n$
- a TM_n that executes on each of the n strings, n transitions of M: if a string s_i is accepted within that budget, it is written on the output tape of Enu
- a TM_{driver} that generates the numbers 1,2,... one after the other and for each of them calls TM_{gen} and TM_n

■

Why did we need acquaintance with the Halting problem? Naively, we would have proposed the following procedure to make an enumerator Enu out of a Turing Machine TM:

> generate the strings from Σ^* in any order $s_1, s_2...$
>
> give each s_i as input to M and if M accepts, output s_i

That does not work, because M might loop on some s_i, and thanks to the Halting problem, we understand there is no way to know this in advance. Suppose s_{i+1} belongs to L_M, then our thus constructed Enu would not enumerate it.

Decidable languages

Related to regular languages

The sentence *regular languages are decidable* might be ambiguous: we must specify precisely what is input to the decider. So we refine that sentence, depending on how the regular language itself is given.

- $A_{DFA} = \{\langle D, s\rangle | D \textit{ is a } DFA, \textit{ and } D \textit{ accepts } s\}$
- $A_{NFA} = \{\langle N, s\rangle | N \textit{ is a } NFA, \textit{ and } N \textit{ accepts } s\}$
- $A_{RegExp} = \{\langle RE, s\rangle | RE \textit{ is a regular expression, and } RE \textit{ generates } s\}$

Theorem 4.10.1. *A_{DFA}, A_{NFA} and A_{RegExp} are decidable.*

Proof. The proof is constructive.

- The decider B has input $\langle D, s\rangle$. B simulates D on s. If D accepts s, B stops in its q_a. If D rejects s, B stops in its q_r. Looping does not occur.
- The simulation of an NFA can loop, unless one performs loop detection ... We chose an easier path: the decider B has input $\langle D, s\rangle$. B transforms the NFA D to a DFA (see the algorithm in Section 3.12). Now use the above simulation.
- Given input $\langle RE, s\rangle$, first transform the RE to an NFA and proceed as above.

■

The technicalities of reductions are introduced only later, but above, we used the principle of reduction twice.

We can also prove that any regular language is decidable. Try to understand the subtle difference with the statement in Theorem 4.10.1. (there is a hint in the proof)

The following three questions about a language are quite popular:

- does the language contain the empty string?
- is the language empty?
- are two given languages equal?

Pertaining to regular languages, we can construct a decider for each of these questions, but once more, we need to specify what is the input to the decider. Instead of constructing the decider as a TM, we will informally explain its working. Make sure that all steps are realizable as a TM.

The first question is trivial, because A_{DFA} is decidable.

Theorem 4.10.2. $E_{DFA} = \{\langle DFA\rangle | L_{DFA} = \phi\}$ *is decidable.*

Proof. There are many ways to prove this. Here is one that uses some theory we studied before, but is also a bit of an overkill: you are invited to find shorter and/or more elegant ways.

Transform the DFA to a minimal DFA_{min} accepting the same language. If $L_{DFA} = \phi$ then DFA_{min} is isomorphic with ... (draw that machine). Deciding that is easy. ∎

Theorem 4.10.3. $EQ_{DFA} = \{\langle DFA_1, DFA_2\rangle | L_{DFA_1} = L_{DFA_2}\}$ *is decidable.*

Proof. We use some algebraic properties of the set of DFA's.

From DFA_1 and DFA_2, construct DFA_Δ that accepts the symmetric difference between L_{DFA_1} and L_{DFA_2}. Then decide whether DFA_Δ accepts the empty language using the previous theorem. (you saw the reduction?) ∎

Selfie: Find other proofs for the theorems above.

Related to context free languages

The questions are similar to the ones in the previous section, and can be formulated either by giving the CFG or the PDA for the context free language at hand. We show it for the grammar only.

Theorem 4.10.4. $A_{CFG} = \{\langle G, s\rangle | G \textit{ is a } CFG, \textit{ and } s \in L_G\}$ *is decidable:* acceptance *of a string by a CFG is decidable.*

Proof. A naive proof idea consists in using the CFG to generate strings, and as soon s is generated, to accept. You can see why this does not work: when do you reject? We have a recognizer, not a decider.

It would be convenient if the CFG were in Chomsky Normal Form: we would then have an upper bound on the length of the derivation for a given string.

So, at input $\langle G, s\rangle$, first convert G to its Chomsky Normal Form. Generate all possible strings with a derivation length $2|s| - 1$: there are only finitely many of them. If s is one of them, accept, otherwise reject. ■

Theorem 4.10.5. *$E_{CFG} = \{\langle G\rangle | G\ is\ a\ CFG,\ and\ L_G = \phi\}$ is decidable:* emptiness *of a CFL is decidable.*

Proof. We describe informally an algorithm transforming G to a form in which taking the decision is easier.

- for a rule $A \rightarrow \alpha$ with α only terminal symbols
  - remove all rules with A at the left side
  - replace the occurrences of A in any right side by α
- keep doing this until
  - the start symbol is removed: reject, because the start symbol can derive a string
  - there are no rules of the required form: accept, because the language is empty

■

Be careful with the above proof: the grammar is transformed to a non-equivalent one!

Selfie: Write down a grammar that determines the empty language and apply th above procedure to it. Did you learn something about Prolog programs that have no answers?

Theorem 4.10.6. *$ES_{CFG} = \{\langle G\rangle | G\ is\ a\ CFG,\ and\ \epsilon \in L_G\}$ is decidable: it is decidable whether a CFG generates the empty string*

Proof. Transform the CFG to its Chomsky Normal Form. If it now contains the rule $S \rightarrow \epsilon$, accept, otherwise reject. ■

Theorem 4.10.7. *Every CFL is decidable.*

Proof. Here, the question is to prove the existence of a decider B_G that can decide for each string whether it belongs to G or not. Appreciate the difference with A_{CFG}. Work out the details yourself. ∎

The remaining one is EQ_{CFG}, i.e. if we get two CFGs, can we decide whether they have the same language? The decider for EQ_{DFA} relied on the symmetric difference of regular languages. We can not use the same idea for CFLs, because they are not closed under complement or intersection.

Selfie:

Prove that $\overline{EQ_{CFG}}$ is recognizable.

What does that mean for EQ_{CFG}?

Prove that Theorem 4.10.6 follows directly from Theorem 4.10.4.

Undecidable languages

In Section 4.8 we proved that A_{TM} and H_{TM} are not decidable. Here are some more of the same kind.

Theorem 4.11.1. *$E_{TM} = \{\langle M\rangle | M \text{ is a } TM, \text{ and } L_M = \phi\}$ is not decidable: it is not decidable whether a Turing Machine accepts no input.*

Proof. Assume that E_{TM} is decidable, then there exists a decider E for it. We describe now a decider B for A_{TM} using E as follows: B receives as input $\langle M, s\rangle$ and works as follows

- it constructs a machine $M_{M,s}$ whose working is given by: on input w, M_s does the following
  - if $w \neq s$, reject
  - else run M on w (or s) en return the same result
- run E on input $\langle M_{M,s}\rangle$
  - if E accepts $\langle M_{M,s}\rangle$, then B rejects its input $\langle M, s\rangle$; indeed: $M_{M,s}$ determines the empty language, so M has not accepted s
  - if E rejects $\langle M_{M,s}\rangle$, B accepts its input: since $M_{M,s}$ is not empty, M accepts s

It follows that B is a decider for A_{TM}, which is impossible, so E does not exist, so E_{TM} is not decidable. ■

We can define the E_{TM} problem slightly different, as an instance of a more general problem: does a given machine M determine a language from a given set? So we have

$$IsIn_{TM,S} = \{\langle M\rangle | L_M \in S\}.$$

Then $E_{TM} = IsIn_{TM,\{\phi\}}$

Another instance of the general problem:

> does a given Turing Machine M determine a regular language, i.e. is $IsIn_{TM,RegLan}$ decidable? We denote that problem as $REGULAR_{TM}$

Theorem 4.11.2. *$REGULAR_{TM}$ is not decidable.*

Proof. Suppose Turing Machine R decides $REGULAR_{TM}$. Fix two symbols from the alphabet, say 0 and 1. Make a decider B for A_{TM} as follows: on input $\langle M, s\rangle$ B does the following

- it constructs a helper machine $H_{M,s}$ that on input x does the following
  - if x is of the form 0^n1^n, accept
  - else let M run on s; return the same answer
- then B lets R run on $\langle H_{M,s}\rangle$
- if R accepts, accept; if R rejects, reject

First note that $H_{M,s}$ never runs: it is there only as input for R. Finishing the proof:

$H_{M.s}$ either accepts the non-regular language $\{0^n1^n\}$ or all of the regular language Σ^*. So, B accepts $\langle M, s\rangle$ iff R accepts $\langle H_{M,s}\rangle$, iff $H_{M,s}$ accepts Σ^*, iff M accepts s. It follows that B is a decider for A_{TM}, which is impossible, so R does not exist, so $REGULAR_{TM}$ is not decidable. ■

Theorem 4.11.3. *EQ_{TM} is not decidable.*

Proof. We already know that E_{TM} is not decidable. E_{TM} is a special case of EQ_{TM}, in which the second machine is M_ϕ. It is clear that if we could decide about two arbitrary languages that they equal, we would also be able to decide about one arbitrary language and the empty one. So, we arrive at a contradiction by assuming that EQ_{TM} is decidable. ■

The above proof technique is studied in more detail in Section 4.16.

The following is about context sensitive languages. We first define the machinery needed to decide them:

Definition 4.11.4. *Linear Bounded Automaton*
A Linear Bounded Automaton is a (non-deterministic) Turing Machine that read and writes only on the part of the tape that contained the initial input.

The name of the automaton might strike you as weird. An equivalent definitions allows the machine to use a portion of the tape that is larger by a constant factor f than the input: this f is independent of the input. We mentioned earlier that such machines can decide context sensitive languages. You can intuitively work that out by adhering to the alternative definition of context sensitive grammar on page 124, trying out a bottom-up parse, and notice that you do not need extra space.

There is no decision procedure for deciding whether a given Turing Machine is actually an LBA (adapt the proof for $REGULAR_{TM}$). But a TM can be turned into an LBA by adding a marker at the left and right of the input, and adapting the transition table so that this marker is never crossed.[3]

The acceptance problem for LBAs is

$$A_{LBA} = \{\langle M, s\rangle | M \textit{ is an LBA and } s \in L_{LBA}\}.$$

The following might surprise you ...

Theorem 4.11.5. *A_{LBA} is decidable*

Proof. Consider the configurations that can occur during the execution of an LBA on an input with length n. Using q for the number of states of the LBA, and b for the size of the tape alphabet, we can quickly compute the number of possible tape contents: it is bounded by b^n. The tape head can be on any cell of the allowed portion of the tape, so the upper bound for the number of configurations is qnb^n.

A decider B for A_{LBA} can now be constructed as follows: on input $\langle M, s\rangle$, B performs the following

- it computes $Max = qnb^n$

[3]**Selfie**: show with an example that in general you have changed the language!

- it simulates M on s for at most Max steps
- if M accepted in the mean time, accept
- if M rejected in the mean time, reject
- if M has not yet stopped, it means that M loops and does not accept: reject

■

Theorem 4.11.6. *$E_{LBA} = \{M|M$ is an LBA with $L_M = \emptyset\}$ is not decidable*

Proof. We first show that for a given Turing Machine M and string s, we can construct an LBA that can decide about a finite sequence of configurations of M, whether it is an accepting computation history for s. A sequence of configurations can be put on a tape easily as in the figure below

\$	a	q_4	c	d	\$	a	b	q_7	d	\$

It represents a transition for which $\delta(q_4, c) = (q_7, b, R)$.

To check whether a sequence of configurations is an accepting computation history for s, the following is needed:

- check that two subsequent configurations are connected through δ
- check that the first configuration is of the form $q_s s$
- check that the last configuration contains q_a

It should be clear - at least intuitively - that these checks need at most a constant amount of extra space, meaning that the decision kan be taken by an LBA. We construct the LBA in such a way that it accepts the accepting computation history for s, and rejects every other input. We can now start the proof.

Suppose we have a decider E for E_{LBA}. We construct a decider B for A_{TM} as follows. On input $\langle M, s\rangle$ B performs the following actions:

- it constructs the LBA $A_{M,s}$ as above described: this LBA can decide whether a given string is an accepting computation history for M on input s
- give $\langle A_{M,s}\rangle$ to E: if E accepts, reject; otherwise accept

B decides A_{TM} because B accepts $\langle M, s\rangle$ iff E rejects $\langle A_{M,s}\rangle$, iff $A_{M,s}$ accepts at least one string, iff there exists an accepting computation history for M on s. The latter is equivalent with M accepts s.

So, B does not exist, so neither can E, and as a conclusion: E_{LBA} is not decidable. ■

A small summary of the differences and similarities between PDAs, LBAs en TMs follows:

- acceptance en halting
  - PDA and LBA are similar: acceptance and halting are decidable
  - TM is different: acceptance and halting are not decidable
- emptiness
  - LBA are TM are similar: emptiness is not decidable
  - PDA differs: emptiness is decidable

To conclude for now: it is not decidable whether a CFG generates all strings from Σ^*, i.e. $ALL_{CFG} = \{\langle G\rangle | L_G = \Sigma^*\}$ is not decidable. That is a bit weird, as E_{CFG} is decidable. Then again, complement is not internal for CFLs ...

Theorem 4.11.7. *ALL_{CFG} is not decidable.*

Proof. No proof this year. ■

Selfie: Prove that ALL_{RegExp} is decidable.

Some trivia on languages

Let Dec denote the set of decidable languages, and Rec the recognizable languages. First some inclusions, all strict:

$$RegLan \subset DCFL \subset CFL \subset Dec \subset Rec \subset \mathcal{P}(\Sigma^*)$$

Some properties of operations on languages:

$RegLan$ is closed under union, intersection and complement

$DCFL$ is closed under complement, but CFL is not

> CFL is closed under union, but $DCFL$ is not
>
> Dec is closed under union and complement
>
> Rec is closed under union

Let us also have a look at a somewhat unusual operation on languages: inversion. Denote by $\hat{s}$ the string obtained by writing the symbols of s in reverse order. We define $\widehat{L} = \{\hat{s} | s \in L\}$.

> $RegLan$, CFL, Dec en Rec are closed under inversion
>
> $DCFL$ is NOT closed under inversion

Example: The language $L = \{ba^m b^n c^k | m \neq n\} \cup \{ca^m b^n c^k | n \neq k\}$ is deterministic context free (construct its DCFG) but $\widehat{L}$ is not.

Countable

A set V is countable if there exists a bijection between V and (a part of) $\mathbb{N}$. This definition is about *existence* and it does not care about the ease with which the bijection can be computed: as far as the definition goes, there is no need for that.

Still, in the context of computability, we care about such bijection being computable by a Turing Machine. We have indeed used constructions that use a TM that generates the strings of Σ^* one by one. It is thus not enough that Σ^* is countable, it needs to be (effectively) enumerable (by a TM). You should be able at this point to prove that there exist subsets if Σ^* that are not enumerable, and give concrete examples as well.

So, what is the connection between being generated by an enumerator machine, and effectively enumerable? The enumerator is allowed to generate duplicates. We can prevent it from doing so by modifying it a little: any time the enumerator arrives in its enumerator state, we check whether the most recently generated string is really new: if not, we erase it. Now the enumerator effectively enumerates all its strings. Since every language generated by an enumerator recognizable, and since we proved that there exists a non-recognizable language L, it follows that this L can not be enumerated, although it is countable.

Selfie: Construct languages that are countable, but not enumerable.

How to kill N birds with one stone: Rice's Theorem

We have seen some concrete examples of undecidable languages, often by reduction to A_{TM}: every time, we found a new little trick so that a supposed decider for a language could be used in a decider for A_{TM} and lead to contradiction. So, Rice's theorem comes as a relief: in one go, it proves that a whole lot (how many?) languages are undecidable.

Consider the set of Turing Machines (for convenience over a fixed alphabet). A property P of the Turing Machines splits the set in two parts: the machines that have property P, denoted $Pos_P = \{M|P(M) = true\}$, and the machines that do not have the property, denoted $Neg_P = \{M|P(M) = false\}$.

Definition 4.14.1. *Non-trivial property*
A property P of TMs is *non-trivial* if
$Pos_P \neq \emptyset$ and also $Neg_P \neq \emptyset$.

Definition 4.14.2. *Language invariant property*
The property P is *language invariant* if
$L_{M_1} = L_{M_2} \Longrightarrow P(M_1) = P(M_2)$
or in words: *machines recognizing the same language either all have Pm or none has P.*

Selfie:

Invent some examples of language invariant, non-trivial properties of TMs.

Invent some examples of properties of TMs that are not language invariant.

For a given non-trivial and language invariant property P, how many machines have P? How many don't?

Theorem 4.14.3. *Stelling I by Rice*
Let P be a non-trivial, language invariant property of the Turing Machines. Pos_P (and Neg_P) is undecidable.

Proof. Suppose $M_\emptyset$ (a machine deciding the empty language) does not have property P - if not, replace P by its negation. Since P is non-trivial, there exists also a machine X having P - denote its language by L_X. Suppose Pos_P were decidable by a decider B. We will use B to construct a decider A for A_{TM}.

A has as input $\langle M, s\rangle$ and does the following:

- it constructs a helper machine $H_{M,s}$ that on input x:
    - runs M on s
    - if M accepts s, lets X run on x and returns its result
- now give $H_{M,s}$ to B
- if B accepts $H_{M,s}$, accept, else reject

First check that $H_{M,s}$ accepts either the empty language, or L_X. This is used in the following:

A accepts $\langle M, s\rangle$ iff B $H_{M,s}$ accepts, iff $H_{M,s}$ has property P, iff $H_{M,s}$ accepts L_X, iff M accepts s.

So, A is a decider for A_{TM}, clearly impossible, so B does not exist, and Pos_P is not decidable. ■

Rice also has a second theorem: every non-monotone property is not recognizable; it is worth having a look at it!

Selfie:

What can you conclude regarding whether Pos_P and/or Neg_P are recognizable?

Use Rice for new proofs of old theorems.

The *Post Correspondence Problem*

Emile Post was one of the founding fathers of recursion theory, and he made contributions to universal of computing mechanisms. The word *correspondence* is related to *similarity, agreement*, not to sending mail by post. Emile Post developed the following game:

You have a finite collection of dominos, each with two strings (instead of the usual numbers). Of each type, there is an unlimited supply for this game. If you put some dominos next to each other, you see an upper string and a lower string. The question is: is it possible to lay out a finite number of dominos to that the upper and lower string are the same?

An example:

ab	ab	b	b
a	ba	bb	b

ab	ab	ab	b
a	ba	ba	bb

Figure 4.4: Four given stones and a solution

Figure 4.4 shows that you don't need to use all dominos and that one type of domino can be used more than once.

We are dealing here with a decision problem: it is enough to answer yes or no to the question, there is no need to construct the corresponding sequence of dominos.

This simple game (1) undecidable and (2) powerful enough to mimic all computations by all Turing Machines. Actually, the former follows form the latter, as soon as we have shown that the halting of a TM can be reduced to existence of a solution to the PCP. We show this by example: the books by Sipser or Minsky contain a general construction.

Turing a Turing Machine in a PCP game

In this example we use X for the start state, A for the accepting state, Z for the reject state. The alphabet is $\{a, b\}$, and the transition table looks like

1	X	a	B	a	R
2	X	b	Z	_	_
3	X	#	A	#	S
4	B	a	Z	_	_
5	B	b	X	b	R
6	B	#	Z	_	_

The numbers at the left are there for ease of reference. Check that this TM accepts the language $(ab)^*$. Take as input string *ab*. We introduce an extra symbol $. We now define the dominos needed in the Post game for imitating the above Turing Machine:

- for each symbol x create a domino with upper and lower string equal to x, i.e. 4 dominoes:

a	b	$	#
a	b	$	#

  and dominoes with Ax or xA as the upper string, and A as lower string, i.e. 8 dominoes:

aA	bA	$A	#A	Aa	Ab	A$	A#
A	A	A	A	A	A	A	A

- rule 1 in the transitie table results in a domino

Xa
aB

- from rule 5 we get

Bb
bX

- from rule 3

X#
A#

- the following dominoes do not depend on the TM

  the end domino

A$$
$

  and the blank generators left and right

$	$
$#	#$

- finally, the input *ab* turns into a starting domino

$
Xab

The question is now: construct a correspondence with these dominoes, starting from the input domino.[4] Here is the solution:

$	Xa	b	$	a	Bb	#	$	a	b	X#	$	a	bA	#	$	aA	#	$	A#	$	A$$
Xab	aB	b	#$	a	bX	#	$	a	b	A#	$	a	A	#	$	A	#	$	A	$	$

In the lower half, you see between any two occurrences of $ a configuration; two subsequent configurations are linked by the transition function, until the accepting state appears. After that, the configuration is emptied until only the accepting state and the end domino appears. Now, upper and lower are equal.

[4]The general PCP allows to start from an arbitrary domino: one problem can be reduced to the other.

Selfie:

> Convince yourself that by starting with an input string not in the language $(ab)^*$, no correspondence can be found.
>
> Find out about the general correspondence between δ and the dominoes: our δ did not have movements to the left!
>
> Formulate more precisely the link between PCP and H_{TM}.

An extra: the Post tag machine E. Post is also known for his *tag systems* or *Post tag machines*. Just a small example of a 2-tag system:

- the alphabet is $\{a, b, c\}$ and there is also a halt symbol H
- the rules are
  - a $\rightarrow$ aa
  - b $\rightarrow$ accH
  - c $\rightarrow$ a
- the initial word is aab
- and here is a derivation starting from the initial word

  aab $\rightarrow$ baa $\rightarrow$ aaccH $\rightarrow$ ccHaa $\rightarrow$ Haaa

The general rule for rewriting is: the first letter of the word decides which rule is used (this could be the source of non-determinism). Now add the right side of such a rule at the end of the string, and erase 2 symbols at the front (the same 2 as in 2-tag system).

The derivation ends when H appears at the front of the string: the rest of the string can be considered the result of a computation starting with the initial string.

This rewriting system is also Turing complete.

2-tag systems have been used to simulate small UTMs.

Many-one reduction

A reduction maps one problem L1 to another L2: it seems a bit counter intuitive, but L2 is in some sense *more difficult* than L1, because a method to solve L1 is (together with the reduction) powerful enough solve L2. We will make all this very precise. It is important to get a good grasp on the reductions in this section: in complexity theory, we use similar reductions, so it is a recurring concept.

Definition 4.16.1. *(Turing) computable function*
A function f is (Turing) computable if there exists a Turing Machine that on input s eventually halts with $f(s)$ on the tape.

We do not care whether the machine stops in an accepting or a rejecting end state.

Definition 4.16.2. *Many-one Reduction of languages*
A many-one reduction of language L_1 (over Σ_1) to language L_2 (over Σ_2) is a total computable function f from Σ_1^* to Σ_2^* so that $f(L_1) \subseteq L_2$ and $f(\overline{L_1}) \subseteq \overline{L_2}$.
We denote this by $L_1 \leq_m L_2$, and we can also say L_1 *reduces to* L_2, or L_1 *is mapping reducible to* L_2.

Figure 4.5 shows the condition on f.

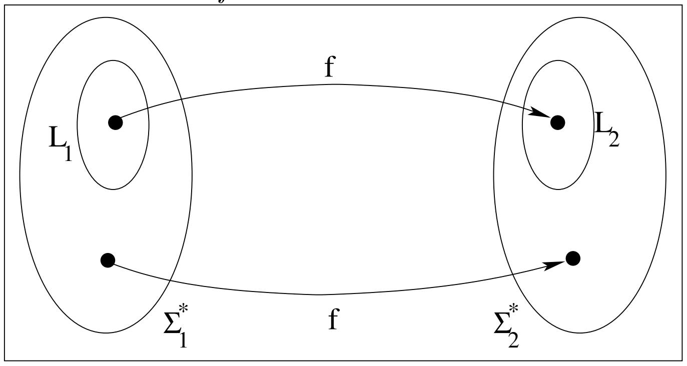


Figure 4.5: Schematic representation of a reduction

A reduction f offers a method to transform the decision question about L_1 into a decision question about L_2, indeed, $s \in L_1$? can be answered by deciding about $f(s) \in L_2$. The next theorems make that more precise: you must be able to prove them.

Theorem 4.16.3. *If $L_1 \leq_m L_2$ and L_2 is decidable, then L_1 is decidable.*

Theorem 4.16.4. *If $L_1 \leq_m L_2$ and L_2 is recognizable, then L_1 is recognizable.*

Consequence 4.16.5. *if $L_1 \leq_m L_2$ andn L_1 is not recognizable, then L_2 is not recognizable. If $L_1 \leq_m L_2$ and L_1 is not decidable, then L_2 is not decidable.*

We have used reductions informally in previous theorems, e.g. on page 146 when we proved that EQ_{TM} is not decidable. Let's do it now based on the definitions.

Theorem 4.16.6. *EQ_{TM} is not decidable.*

Proof. We reduce E_{TM} to EQ_{TM}: f maps input $\langle M \rangle$ to $\langle M, M_\phi \rangle$; M_ϕ a Turing Machine accepting the empty language. Clearly, f is Turing computable.

So, $E_{TM} \leq_m EQ_{TM}$, and since we know already that E_{TM} is not decidable, EQ_{TM} is not decidable either. ■

Theorem 4.16.7. *If $A \leq_m B$ then $\overline{A} \leq_m \overline{B}$.*

Proof. Selfie! ■

Also on page 145, we used a reduction from $\overline{A_{TM}}$ to E_{TM}: we mapped $\langle M, s \rangle$ to $\langle M_s \rangle$ ... Note that a mapping reduction from A_{TM} to E_{TM} does not exist - can you explain why?

Theorem 4.16.8. *EQ_{TM} is not recognizable and not co-recognizable.*

Proof. We construct two mapping reductions: $A_{TM} \leq_m \overline{EQ_{TM}}$ and $A_{TM} \leq_m EQ_{TM}$. Since $\overline{A_{TM}}$ is not recognizable, the result follows.

1. f maps $\langle M, s \rangle$ to $\langle M_s, M_\phi \rangle$; M_s is a machine that accepts every string if M accepts s; clearly f is computable; we should also prove the other conditions:
   - if M accepts s, then the languages of M_s and M_ϕ are different
   - if M does not accept s, then the languages of M_s and M_ϕ are equal
2. now, f maps $\langle M, s \rangle$ to $\langle M_s, M_{\Sigma^*} \rangle$; M_{Σ^*} is a machine accepting all strings; M_s is like before; checking the conditions on f should be straightforward.

■

Oracle machines and a hierarchy of decidability

Suppose we had another way to decide A_{TM} - not a TM of course - would there be a way to use that power to decide everything? Let's make the question a bit more concrete:

- since no TM can decide A_{TM}, we must name this device differently: it is an *oracle*; we get back to the implementation of an oracle later ...
- a TM must be able to consult the oracle, i.e. the oracle for A_{TM} can be called by a TM as a subroutine, with a string s as input to the oracle; the oracle needs only a finite number of steps to give its decision (is $s \in A_{TM}$?) to the TM
- in this way, we build a so called oracle machine $O^{A_{TM}}$: it is a TM that can ask questions to the oracle for A_{TM}

It is clear that we can make an $O^{A_{TM}}$ that decides A_{TM}: give the input $\langle M, s\rangle$ to the oracle, and return its answer. So, the set of oracle machines with oracle A_{TM} is strictly stronger than the set of Turing Machines. Here comes another example:

Theorem 4.17.1. *There exists an $O^{A_{TM}}$ that decides E_{TM}.*

Proof. We construct $O^{A_{TM}}$ as follows: on input $\langle M\rangle$ $O^{A_{TM}}$ performs the following actions

- it constructs a Turing Machine P that on input w performs the following actions
  - run M on all strings of Σ^* (*)
  - if M accepts a string, accept
- ask the oracle for A_{TM} whether $\langle P, x\rangle \in A_{TM}$
- if the oracle answers **yes**, reject; otherwise accept

If $L_M \neq \emptyset$, then P accepts every input, and for sure also input x; so, the oracle answers **yes** and $O^{A_{TM}}$ must reject. Vice versa: if $L_M = \emptyset$, then $O^{A_{TM}}$ accepts. We conclude that $O^{A_{TM}}$ decides the language E_{TM}. ■

The theorem proves that E_{TM} is decidable relative to A_{TM}. The relevant definition is:

Definition 4.17.2. *Turing reducible*
A language A is Turing reducible to language B, if A is decidable relative to B, i.e. there exists an oracle machine O^B that decides A. We write this as $A \leq_T B$.

The definition supports our intuition about what it means for one language to be reducible to another:

Theorem 4.17.3. *If $A \leq_T B$ and B is decidable, then A is decidable.*

The previous and following theorems are at your mercy!

Theorem 4.17.4. *If $A \leq_m B$ then also $A \leq_T B$.*
In other words: $\leq_m$ is finer than $\leq_T$.

Time for a description of an oracle for any language L: each string has a running number in the lexicographic order on strings (shorter before longer, and alphabetic). That means that the language can be represented as an infinite bitmap: bit i is one, if the i^{th} string is in L, and otherwise zero. That bitmap could be on a tape. When an oracle gets a membership question about s, it first computes its corresponding number, and then looks up the right bit ... So, do oracles exist or not?

Another question: can the class of oracle machines with an oracle for A_{TM} decide all languages? Certainly not, as a cardinality argument can show you! So there exists a language X that is not decidable with an oracle for A_{TM}. You can now repeat the argument for the X oracle ... and you soon note that there is an infinite hierarchy of ever more difficult languages. As a sidenote: also complexity theory has its oracle machines and a nice hierarchy. The difference is: the particular complexity hierarchy would still collapse in case it turns out that $NP = P$. Our compatibility hierarchy however stands strong!

Selfie: In the theorem on page 158, a statement is marked by a (*): how can we let M run on all strings? There are infinitely many of them ...

The computable functions and the recursive functions

We gave a definition of Turing computable function in the previous section. It was rather abstract and *existential*. We can also get a grip on which functions are computable by a Turing Machine, by working *bottom-up*: we start with very simple functions, and compose them with simple operators. That is how Kurt Goedel and Jacques Herbrand did it.

The primitive recursive functions

The basic functions

- the null function: $null : \mathbb{N} \to \mathbb{N}$
  $$null(x) = 0$$
- the successor function: $succ : \mathbb{N} \to \mathbb{N}$
  $$succ(x) = x + 1$$
- the projections: ${p_i}^n : \mathbb{N}^n \to \mathbb{N}$
  $${p_i}^n(x_1, x_2, ..., x_n) = x_i$$

Composition

Given:

- $g_1, g_2, ..., g_m$ functions $\mathbb{N}^k \to \mathbb{N}$
- f function $\mathbb{N}^m \to \mathbb{N}$

construct by composition the function $h : \mathbb{N}^k \to \mathbb{N}$:

$$h(\overline{x}) = f(g_1(\overline{x}), g_2(\overline{x}), ..., g_m(\overline{x}))^5$$

Notation: $h = Cn[f, g_1, \ldots, g_m]$

Primitive recursion

Given:

- $f : \mathbb{N}^k \to \mathbb{N}$

5 $(\overline{x} = x_1, x_2, ..., x_k)$

- $g : \mathbb{N}^{k+2} \to \mathbb{N}$

construct using primitive recursion $h : \mathbb{N}^{k+1} \to \mathbb{N}$:

$$h(\overline{x}, 0) = f(\overline{x})$$
$$h(\overline{x}, y+1) = g(\overline{x}, y, h(\overline{x}, y))$$

Notation: $h = Pr[f, g]$

The above is valid for $k > 0$. For $k = 0$ we have: $h : \mathbb{N} \to \mathbb{N}$:

$$h(0) = c \text{ in which } c \text{ is a number}$$
$$h(y+1) = g(y, h(y))$$

Definition 4.18.1. *Primitive recursive function*
All functions that can be constructed from the basic functions and by using composition and primitive recursion, are named **primitive recursive**.

Primitive recursive functions are total. They can be computed by FOR-programs.

Examples

- $Cn[succ, null]$ is the constant function 1
- $Cn[succ, Cn[succ, Cn[succ, null]]]$ is the constant function 3
- $Pr[{p_1}^1, Cn[succ, {p_3}^3]]$ = sum of 2 inputs ($\mathbb{N}^2 \to \mathbb{N}$)
  write out the definition and see the analogy with the Haskell program:
  sum(x,0) = x
  sum(x,y+1) = sum(x,y) + 1
- $Pr[null, Cn[som, {p_1}^3, {p_3}^3]]$ = product of 2 inputs ($\mathbb{N}^2 \to \mathbb{N}$)
- factorial, minus1 (or predecessor), ...

The recursive functions

Existence of a non primitive recursive function: the Ackermann function

Initially, Goedel stopped at the primitive recursive functions when he was trying to define all *computable functions*: he was doing that without referring to TMs. However, in 1928, Wilhelm Ackermann, a student van David Hilbert, published what we now know as the

Ackermann function: this function is clearly *computable* in an intuitive sense and understanding of computability (and also by a TM, but they did not exist in 1928!), still, the Ackermann function is not primitive recursive.[6] Here follows the definition of Ackermann's function with two arguments:

$$\text{Ack}(0, y) = y + 1$$

$$\text{Ack}(x + 1, 0) = \text{Ack}(x, 1)$$

$$\text{Ack}(x + 1, y + 1) = \text{Ack}(x, \text{Ack}(x + 1, y))$$

This function is total (try to understand this!) and increases faster than any primitive recursive function: that's why it is not primitive recursive.

Often one uses the words *Ackermann's function* to mean a function with one argument only. It is defined as `Ack(n) = Ack(n,n)`. Also this function increases faster than any primitive recursive function. Its inverse is important in complexity analysis of algorithms, a.o. for the Union-Find algorithm.

Unbounded minimization

Goedel had to enlarge his class of functions, so that also the Ackermann function was included. He introduced *unbounded minimization.*

Given

- $f : \mathbb{N}^{k+1} \to \mathbb{N}$

Construct by unbounded minimization:

$g : \mathbb{N}^k \to \mathbb{N}$ as follows:

$g(\overline{x}) = y$ if

$f(\overline{x}, y) = 0$ and
$f(\overline{x}, z)$ is defined for all $z < y$ and $f(\overline{x}, z) \neq 0$

otherwise $g(\overline{x})$ is not defined

Notation: $g = Mn[f]$

Loosely speaking, *g gives the minimal roots of f*

[6]Actually, another student of Hilbert, Gabriel Sudan, was the first to establish a computable, but not primitive recursive function.

Code to compute Mn[f](x)

```
y = 0;
while (f(x,y) != 0)
   y++;
return y;
```

There are two ways you can end up in a loop ...

Definition 4.18.2. *Recursive functions*
Recursive functions are constructed from the basic functions, and by applying Pr, Cn and Mn. One also names these functions *μ-recursive.* μ is the minimalization operator.

The recursive functions can be computed with WHILE-programs. More specifically, these functions can be computed by a Turing Machine, and vice versa. It means that the uring computable functions coincide with the recursive functions. It is clear from the definition of unbounded minimization that recursive functions can be partial. That is consistent with the notion of Turing computable: a TM often defines just a partial function. But the Ackermann function shows that really recursive functions can also be total.

Selfie:

Understand and/or prove: if the domain of a partial recursive function is decidable, then it can be extended to a total function that is also recursive.

Do partial recursive functions exist whose domain is not decidable?

Is the domain of a (partial) recursive function recognizable?

Is the range of a (partial) recursive function recognizable?

The busy beaver and fast increasing functions

The Ackermann function showed us that the fact that a function increases fast, can be an indication that extra machinery is needed to compute its values. For the Ackermann function, that worked with Turing Machines: TMs can implement unbounded minimization. The essence is *find the* **smallest** *number with a particular property V*. This can be implemented as

```
$i = 0;$
$while\ not(V(i))\ i++;$
```

and you already know that it does not finish if no i satisfies V.

We here introduce another novelty, just for trying it out: *unbounded maximalization*, i.e. *find the* **largest** *number with a particular property V*. How could we implement that? Here is a first attempt:

```
$i = \infty;$
$while\ not(V(i))\ i--;$
```

That has no chance of working - please argue.

Here is a second attempt:

```
$i = 0;$
$while\ i < \infty$
    $if\ V(i)\ max = i;$
    $i++;$
```

and you can certainly spot the weakness here as well ...

This shows that the introduction of a method to construct new functions from old ones, has its pitfalls.

The busy beaver

IN 1962, Tibor Radó invented the following function S: consider the class of all TMs with alphabet $\{0, 1\}$ and n states. There are only a finite number of them. We can let these TMs run with an initial empty tape - say all zeros. We then count for each machine how many transitions it takes until it stops (in q_a or q_r). A busy beaver is a champion in its class, i.e. for a given n, no other machine in its class needs more transitions before it stops. We define $S(n)$ as the number of transitions the busy beaver in class n makes. Tibor Radó restricted the TMs to make a head movement at each transition, and in terms of how often the head moves, but that does not matter.

Clearly, S is defined as a total function with signature $\mathbb{N} \to \mathbb{N}$. The question *is S Turing computable* is relevant.

It feels possible to construct a busy beaver for small n: construct exhaustively all TMs with n states, start them with an empty tape and start counting.

There is however a problem: because of the Halting problem, we know we do not always know in advance whether a machine eventually stops on the empty input. That means that when a machine runs very long, and we are losing patience, we better come up with a proof that the machine is actually in a loop, or we must accept that we can only can construct lower bounds for $S(n)$. This is really a problem, even for small n, because there exist very small universal TMs!

Nevertheless, the values for $S(n)$ are known exactly for $n < 5$. For $n = 5$ the current record is 47 176 870, but it is not know about some machines whether they are in a loop, so that record could still be broken. The record for $n = 6$ is 10^{2879}.

S is obtained by *bounded* maximalization, but over a non-computable set: the set of halting TMs with n states. That is the inherent reason why S is not computable, even though the function is total and in the abstract sense well defined.

Lots of open problems in mathematics could be solved *easily* if we knew S for some n.

Selfie: Which sentences below are true? Please give good arguments for your answer.

$S(n)$ can be determined for every n, but it is ∞ for n large enough.

It is possible that for n large enough, $Ack(n) > S(n)$.

Every function that increases slower than a given primitive recursive function is primitive recursive.

Chapter 5

Introduction to Complexity Theory

Chapter 3 studies the decidability of languages, and which machinery is needed to do that. Once it is clear that a language L is decidable, the game changes: we know there exists algorithms for deciding whether a string belongs to L, and we are now interested in the *cost* of a particular algorithm, the inherent cost of the problem, and even the *best* algorithm. All this must be made more precise.

Cost is always expressed in units of a particular type. We have two natural types for the cost of an algorithm: its time and its space usage. In this chapter, time is dominant. In daily life, we measure time in (multiples of) seconds. That time unit is difficult to formalize. Also, we would prefer not to depend on the technology at hand, because technology changes. We are lucky: there exists a universal technology: the Turing machine. Time means nothing to a TM, but we have the notion of an elementary operation that covers the execution of TM: one application of δ. That unit is used as *time unit*. We will at some point criticize this model, but it stands very firm. To be more precise on the model: the TM has one tape that is unbounded in both directions; the read/write head can move to the left, to the right, or stay on the same cell at each transition.

To get insight in the performance of an algorithm, someone could prepare for you a tabel with for a number of input strings s the time $time_A(s)$ the algorithm A uses for deciding about s. It is difficult to do that for enough relevant strings, and it is difficult to interpret correctly such a tabel. We must find a better way to play this game. It is reasonable to expect, and this is true for most reasonable/interesting algorithms, that (roughly) *the longer s, the higher $time_A(s)$*. It is therefore reasonable to express the performance of A as a function $time_A$ that maps the length n of an input string on one of the following quantities:

- the minimal time A uses to decide a string s with $|s| = n$
- the average time A uses to decide a string s with $|s| = n$
- the maximal time A uses to decide a string s with $|s| = n$

We use the latter: it s known as the *worst case time complexity* of A. A formal definition follows soon.

In this chapter the following topics are treated:

- time complexity of a Turing machine
- the class **P** and its robustness
- certificate and verifier
- two definitions of the class **NP**
- polynomial reduction and $\mathbb{N}$P-completeness
- examples of **NP**-complete problems: SAT, HAMPATH ...
- the time hierarchy
- co-**NP** and $\overline{\mathbf{NP}}$
- en introduction to space complexity complexity

Time complexity of algorithms

Definition 5.1.1. *Time complexity of an algorithm*
The **time complexity** *of an algorithm A is a function $time_A(n) : \mathbb{N} \rightarrow \mathbb{N}$ that maps an input size n to the maximally needed steps performed by A on any input string of size n:*

$$time_A(n) = max(\{time_A(s) |\ |s| = n\})$$

$time_A(s)$ is the number of steps A performs to decide s.

The definition shows that $time_A(n)$ is a **worst case** measure. It is quite possible that the algorithm needs much less time than $time_A(n)$ for most strings with length n and that only a few exceptional strings s with that length really need $time_A(n)$.

In general, complexity theory emphasizes the performance of an algorithm on large input. Since the exact time needed for one step in a TM is not clearly defined, the time complexity of an algorithm is relevant only up to a constant[1]. Therefore, the following definition comes in handy:

[1] And don't forget the linear speedup theorem!

Definition 5.1.2. *The big O-notation (pronounce as* big oh*)*
let f, g be functions with signature $\mathbb{N}$ to $\mathbb{R}^+$. We say $f(n)$ is $O(g(n))$ (or f is $O(g)$, and we write also $f(n) = O(g(n))$) if

$$\exists c \in \mathbb{R}_0^+,\ \exists N \in \mathbb{N},\ \forall n \in \mathbb{N}:\ n \geq N \Rightarrow f(n) \leq c.g(n)$$

We also say: f is of order g, and f is asymptotically dominated by g.

Note: it is possible that for particular f and g, both f is $O(g)$ and g is $O(f)$ are true, but that does not mean they are equal.

Example 5.1.3. *The function mapping n on $3n^2 + 4n + 3$ is $O(n^2)$, because*

$$3n^2 + 4n + 3 \leq 4n^2,\ \ \forall n \geq 5\ \ (c = 4,\ \ N = 5).$$

On the other hand, n^2 is also $O(3n^2 + 4n + 3)$ because

$$n^2 \leq 3n^2 + 4n + 3,\ \ \forall n \geq 0\ \ (c = 1,\ \ N = 0).$$

A selfie: prove that a positive polynomial with degree k is $O(n^{k+i})$ if $i \geq 0$, but not if $i < 0$; prove that the relation *dominates asymptotically* is transitive; provide two functions f and g so that f is not $O(g)$ and (at the same time) g is not $O(f)$.

Definition 5.1.4. *Asymptotic equivalence of functions.*
Two functions $f, g : \mathbb{N} \to \mathbb{R}^+$ are asymptotically equivalent if

$$f \text{ is } O(g) \text{ and } g \text{ is } O(f)$$

We denote this by f is $\theta(g)$ (or g is $\theta(f)$).

A selfie: *asymptotically equivalent* is an equivalence relation; two positive polynomials are asymptotically equivalent if and only if they have the same degree.

The O-concept provides a means to compare algorithms. Suppose two algorithms A and B solve the same problem. Let $time_A(n)$ and $time_B(n)$ their complexity functions. A is better (for large input) than B if

1. $time_A(n)$ is $O(time_B(n))$, and
2. $time_B(n)$ is not $O(time_A(n))$

So, if we have an algorithm A with linear complexity (i.e. $time_A(n)$ is $O(n)$) and an algorithm B with quadratics complexity function ($time_B(n)$ is $O(n^2)$), we consider A the better algorithm. One should however not conclude that algorithm A is always to be preferred over B: the constant factor c in the definition of big oh could be very large for A and small for B, meaning that if n remains small, B could be more efficient.

To get a bit more insight in the notion of complexity, it pays off to know the asymptotic behavior of some functions. The sequence below gives the asymptotic behavior of some important functions: a function $f(n)$ is to the left of $g(n)$ if $f(n)$ is $O(g(n))$ (and not the other way around).

$$\log_2 n \;\; n \;\; (n \log_2 n) \;\; n^2 \;\; \cdots \;\; n^k \;\; \cdots \;\; 2^n \;\; n!$$

Definition 5.1.5. *Let T be a function with signature$\mathbb{N} \to \mathbb{R}^+$. We say that a Turing machine M runs in time T if M stops on every input s after at most $T(|s|)$ steps.*

We also say M *is a T-time Turing machine*. The algorithm implemented by M is of course $O(T)$, but the converse is not true. (Why not?)

Definition 5.1.6. *A polynomial algorithm*
An algorithm is **polynomial** *if its time complexity is $O(n^k)$ for some $k \in \mathbb{N}$.*

Definition 5.1.7. *An exponential algorithm*
An algorithm is **exponential** *if its time complexity is $\theta(c^n)$ for some real number $c > 1$.*

A quick comparison shows that exponential algorithms are eventually less efficient than polynomial algorithms. Suppose that a particular problem can be solved by algorithm A with $time_A(n) = n^5$, and also by B with $time_B(n) = 2^n$. On a machine that can execute 10 million instructions each second, A solves a problem of size $n = 60$ in less than 2 minutes, while B needs more than 3500 years. We say B *scales badly.*

It feels like it is in our interest to have polynomial algorithms for particular problems. That is however not always possible, and even it is, the degree of the polynomial might be too high for practical purposes. Other methods will need to be employed in that case: these methods are not part of this course.

What if we use other types of machines? Many other computing formalisms are Turing-complete, so it feels a bit arbitrary to build complexity theory starting from TMs. We will later see why it works anyway. Another attack on TMs would be that one could express the cost of an algorithm in terms of the number of elementary operations needed, as for instance the comparison of swapping of two array elements. One can read sentences like *quicksort is* $O(n^2)$, but the n is the number of items to sort, not the size of the input.

The n^2 is a (worst case) measure for the number of comparisons, and indeed, for quicksort that is $O(n^2)$. But one comparison takes many steps on a TM, and moreover depends on the size of the representation of the numbers to compare. On top of that, *sorting* is not a decision problem ...

To decide or to compute? A decider behaves like a function whose range has only two values: accept and reject. On the other hand, we can interpret the contents of the tape at the end of a decision process as the result of a computation. If we look at it that way, the TM M_f implements a function f with range Σ^*. You can construct a TM M_{fbit} with input a tuple $\langle s, i, b\rangle$ - the TM accepts if the i-th bit of $f(s)$ equals b: M_{fbit} uses M_f as a *subroutine*[2]. This shows a strong link between deciding and computing. It also works the other way around: if you get M_{fbit}, you can build M_f (even if M_{fbit} does not have M_f inside) ... well, almost ... What is missing?

Non-deterministic time complexity classes We later need a complexity measure for non-deterministic Turing machines (NDTM). The non-deterministic analogue of Definition 5.1.5 is

Definition 5.1.8. *Let T be a function with signature $\mathbb{N} \to \mathbb{R}^+$. We say that a non-deterministic Turing machine M runs in time T if M stops on every input s after at most $T(|s|)$ steps for each choice of the transition.*

Time complexity of languages or decision problems

Now we know about the time complexity of algorithms, we could try to determine the *best* algorithm for a given language L. That would give a characterization of the inherent difficulty of the problem. That approach fails in most cases. Still, there is a way to get some understanding of the difficulty of a problem.

Time complexity classes

Definition 5.2.1. *DTIME(T)*
DTIME(T) is the set of languages that can be decided by a (deterministic) $c.T$-time Turing machine, with $c > 0$.

$NDTIME(T)$ is similar but for a decision by a $c.T$-time NDTM.

[2]Please make the construction more explicit

A selfie:

if $f = O(g)$, then $DTIME(f) \subseteq DTIME(g)$
$DTIME(T) \subseteq NDTIME(T)$

The next definition captures the class of languages that can be decided by a polynomial Turing machine:

Definition 5.2.2. **P** $= \cup_{k\geq 1} DTIME(n^k)$

Examples of languages in P

- $Sum = \{\langle a, b, c\rangle | a, b, c \in \mathbb{N} \wedge a + b = c\}$
- $Sorted = \{[a_1, a_2, ..., a_n] | a_i \in \mathbb{N} \wedge (a_i \leq a_{i+1})\}$
- $Connected = \{\langle G\rangle | G \text{ is a connected graph}\}$

The definition of P is robust. We previously committed to expressing the time complexity function on a Turing machine with one read/write two-sided tape, and with the possibility for the head to not move. What happens to **P** if we use another machine ? The answer is ...*nothing* on condition we keep it reasonable: more than one tape, one-side tape ... it does not matter. We can even use random-access machines, n-dimensional tapes ... Each of those variant TMs can be simulated by the type of TMs we committed to, with only a polynomial overhead. It because tricky when one would allow one memory cell to contain an arbitrarily large number, or count some arithmetic operations on such numbers as unit cost. For instance, it would be weird (but scientifically valid) to define the time cost of the n^{th} step during a T execution as 2^{-n}[3]: each algorithm takes at most one time unit ... and we can fold the complexity theory books.

Outside of P ... Some problems cannot be solved in polynomial time. The class of languages that need at most exponential time contains such languages:

Definition 5.2.3. **EXP** $= \cup_{k\geq 1} DTIME(2^{n^k})$

It should be clear that **P** $\subseteq$ **EXP**. The inclusion is even strict: see Section 5.3. The complement of **EXP** is is not empty: Presburger arithmetic provides an example.

[3]This machine is known as a Zeno-machine.

How about the encoding? It is time to worry about the encoding of a problem, or of a language. A decision problem is usually formulated without direct reference to its possible encoding. For instance: we can talk about the language of even numbers, or the problem to decide whether a number is odd or even. Once the encoding is fixed, we can construct an algorithm, and then perhaps determine its complexity.

For numbers, we have (at least) two natural encodings: we can use a unary alphabet, or a binary one. In the unary one, the number 2048 has representation length 2048, in binary one, it has length 12. Let us imagine a TM_u and a TM_b for either encoding. The best possible TM_u needs to count (modulo 2) the number of characters in s, so it is $O(|s|)$. The best TM_b needs to skip all characters until it finds the last one, and then takes the decision: it is $O(|s|)$ as well. But in terms of the size of the number represented by s, the binary representation is superior: for deciding about 2048, TM_b needs a little more than $log_2(2048)$ steps, while TM_u needs about 2048. On the one hand, both algorithms have linear complexity, on the other hand, the unary encoding needs exponential more time for deciding the same number.

A second example makes it worse: consider a naive algorithm to decide whether a number n is prime. It has a loop of the form

```
i := 2
while $i \leq n$ do
    if i divides n then
        REJECT;
        i += 2;
    end if
end while
ACCEPT
```

In both encodings, the loop is executed n times, but in terms of the size of the encoding, that is exponentially more with the binary representation than with the unary one. In one case we could think that the algorithm is polynomial, in the other case that it is exponential.

Clearly, we need to agree on the encoding in order to talk unambiguously about the complexity of a problem and an algorithm. Here is rule number one: for numbers do not use the unary encoding. You may use binary, ternary or decimal ...: that makes no real difference since all these representations can be transformed into each other at only a polynomial cost. For other objects, we might have to find an ad-hoc agreement, or sometimes, it is clear that some encodings are bad. For graphs, we should use the adjacency matrix. Think up some unreasonable representation for graphs, and explain why it is.

The complexity of multiplication: an excursion

For convenience, we use the decimal representation of numbers. We can consider the multiplication of two numbers in the range 0..9 as elementary. How many elementary multiplications do we need to multiply two numbers? For simplicity, we consider numbers of the same length (add zeros if needed):$a = a_n a_{n-1} ... a_1$, and $b = b_n b_{n-1} ... b_1$. We want to compute $a \times b$. At school, we learned to do this with n^2 elementary multiplications,and some additions. People have thought for a long time that it could no be done with essentially less: the grat A. Kolmogorov even has put this forward as a conjecture he strongly believed in. A. Karatsuba showed that it is possible anyway.

Again for simplicity, assume that n is even and equal to $2m$. Then one can write a and b as $a = A_2 10^m + A_1$ en $b = B_2 10^m + B_1$. Then

$$a \times b = (A_2 \times B_2)10^{2m} + (A_2 \times B_1 + A_1 \times B_2)10^m + A_1 \times B_1$$

in which you can see 4 multiplications.

A. Karatsuba noticed that the coefficient of 10^m can be computed in a different way:

$$(A_2 \times B_1 + A_1 \times B_2) = (A_2 + A_1) \times (B_2 + B_1) - (A_2 \times B_2) - (A_1 \times B_1).$$

In the last two multiplications give the coefficients of 10^{2m} and 10^0. By remembering the result of the last two multiplications, and reusing it, one now gets $a \times b$ in only 3 multiplications. By applying this principle recursively (divide and conquer) the number of elementary multiplications is $O(n^{log_2(3)})$. In the mean time, algorithms with an even lower complexity have been discovered.

What did we learn from this excursion? Sometimes, intuition tells us that there is only one possible way to compute a result, and later we learn we were wrong: there is another (better) way, perhaps many other ways.

Certificates and verifiers

The concepts of certificate and verifier are introduced here independent of their use in complexity: you must be able to understand and interpret their definitions (and properties) without referring to complexity.

Definition 5.2.4. *A* **verifier** *for a language L is a deterministic TM V such that*

$$\forall s \in \Sigma^* : (s \in L \leftrightarrow \exists c \in \Sigma^* : V(\langle s, c \rangle) = accept)$$

c is named the **certificate** *for s*

Some examples:

- let L be the set of composite numbers; there exists of course a TM M that decides that language: a typical algorithm would try to find a divisor for a given number.

However, if a number $n \in L$ than there exists a divisor d, and we could give the numbers n and d to a TM V that verifies thatr d is indeed a divisor of n: V has less work to do than M, and according to the definition, it is a verifier for L, while the given divisor d is the certificate. Note that it impossible to fool or mislead the verifier: you cannot make it accept a prime as a composite

- let L be the language of tuples of the form $\langle G, a, b\rangle$ in which G is a graph, and a and b are vertices connected by a path in G; once more, there exists a TM M that decides exactly this language, by just trying to construct a path; a verifier V can be easier: a possible certificate is a path between a and b; V merely needs to check that this certificate is really a path in G, starting with a and ending with b; verifying something is a path is easier than constructing it: V needs less work than M; try to fool V ... it is impossible
- let L be the set of TMs M for which L_M is not empty; we showed earlier that this language is not decidable; still, for every TM M with $L_M \neq \emptyset$, there is a certificate that it is in L, namely any $c \in L_M$; the verifier just runs M on c

The first two examples give the intuition that the job of a verifier for L is easier than the job of a decider for L. The last example makes it blatantly clear: the language does not need to be decidable in order to have a verifier/certificate.

Confused? Think about it, reread Definition 5.2.4.

It might have escaped your attention: Definition 5.2.4 does not mention strings that do **not** belong to the language. You get to know something about such strings by juxtaposition. You get: for $s \notin L$, every c is **not** a certificate.

Think about this: the definition of verifier/certificate says nothing about strings **not** in the language.

The intuition holds up: a verifier must work less than a decider. Some questions remain:

- does every language have a verifier and a certificate for each string in the language?
- can a string have more than one certificate?
- how difficult is it to construct (by an algorithm) a certificate? is that always possible?

The second question above has *yes* as answer for each of the three examples: every divisor is a certificate for a composite number; every path is a certificate for the fact that a and b are connected; every $s \in L_M$ is a certificate.

As for the last question: for the first two examples, the certificate can be generated by an algorithm (although nobody knows whether it can be done in polynomial time in the first example); in the third example, no algorithm can generate a certificate for each string. Think about this: it might be a bit more subtle than you think at first.

An additional question can be asked about the length of the certificate:

- is there a relation between the length of $s \in L$ and the length of its certificate, and the complexity (or decidability) class of L?

Consider all three examples and comment on that question.

Suppose there is a function f so that for a verifier V and certificate c for s, the length of s is bounded above by $f(|s|)$, or: $|c| \leq f(|s|)$. Now we can generate a certificate as follows for any $s \in L$ as follows:

```
for all $(i \leq f(|s|) \wedge (c \in \Sigma^i)$ do
    if $V(\langle s, c \rangle)$ accepts then
        ACCEPT
    end if
end for
REJECT
```

This deterministic algorithm[4] tries all potential certificate candidates. Can we say something about its complexity?

- in the worst case, the for-loop is executed $O(2^{f(|s|)})$ (we assume the alphabet has two elements)
- if M is a g-time machine, the cost of the body of the for-loop is $O(g(|s| + |c|))$, or $O(g(|s| + f(|s|)))$
- this results in an overall complexity $O(g(n + f(n)) \times 2^{f(n)})$ where n is the size of the input

If f and g are at least linear, the complexity of this algorithm is exponential or worse, even if g is not. You might feel like arguing that for some problems there is a less naive way to generate certificates (think about the connected vertices), and you are right ... but there are quite a lot of problems for which science has not achieved that. We will see more about this later.

The algorithm above *guesses systematically* for a certificate. One can model that with a non-deterministic procedure, e.g. by a Prolog predicate. It can also be modelled with a non-deterministic Turing machine $NDTM_V$ with just two transitions at each stage. On input s, $NDTM_V$ writes in its first $f(|s|)$ steps non-deterministically a string from $\Sigma^{f(|s|)}$ on the tape. It then calls the deterministic verifier V. If one of the branches accepts, s is accepted. Figure 5.1 illustrates this: the length of the certificate is 2.

[4] Is it an algorithm or can you turn it into one?

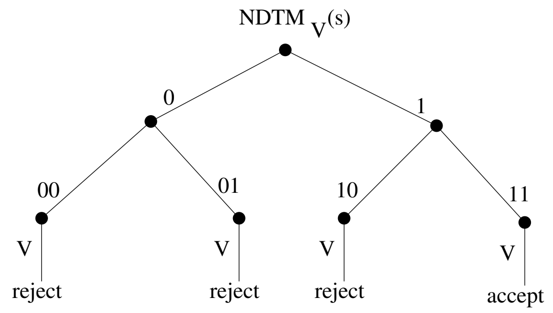


Figure 5.1: Generation of a certificate when the length is known

Vice versa, we can transform this $NDTM_V$ into a verifier: it needs a certificate. Give as certificate a string representing the choices needed for arriving at an accept state. Now simulate simulate $NDTM_V$ with those choices, of course with a deterministic TM ...

We have just (informally) shown that one can go back and forth between a verifier+certificate (with known length) and a non-deterministic Turing machine.

Two definitions of NP

The class **NP** can be defined in two way: the material in the previous section contains the ingredients to prove their equivalence.

Definition 5.2.5. NP *with verifiers*
NP *is the set of languages L for which there exists a polynomial verifier and a polynomial p such that for each $s \in L$, there exists a certificate c vso that $|c| \leq p(|s|)$.*

*More formally: $L \in$ **NP** if and only if*
$\exists$ a polynomial TM M and a polynomial $p : \mathbb{N} \rightarrow \mathbb{R}^+$, so that
$s \in L \Leftrightarrow \exists c \in \Sigma^{p(|s|)}$ met $M(\langle s, c \rangle) = accept$

The alternative definition of **NP** uses non-deterministic Turing machines:

Definition 5.2.6. NP *with non-determinism*
NP *is the set of languages accepted by a polynomial non-deterministic Turing machine.*

You should be able to argue that these two definitions define the same set. Also, you must be able to argue that **NP** is robust with respect to the execution model.

Some examples of languages in **NP** :

- since **P** $\subseteq$ **NP** (why?) each language in **P** can serve as an example ...
- pairs of isomorphic graphs (*)
- graphs with a Hamiltonian cycle (*)
- non-connected graphs
- weighted graphs with between two given nodes a simple pad shorter than a given number (*)
- Come up with something yourself!

Give for each example a description of a verifier and a certificate for a string belonging to the language. Argue that it is polynomial. Note that for the examples marked with (*), no polynomial algorithm is known, neither whether one exists.

Careful: **NP** does means *non-deterministic polynomial*, and **not** *non-polynomial.*

Polynomial reduction

In a previous chapter, we have reduced languages to each other: the many-one reduction $\leq_m$ is an example (See Page 156). The reduction below takes into account complexity.

Definition 5.2.7. *Polynomial reduction*
Given two languages $L_1 \subseteq \Sigma_1^*$ (over alphabet T_1) and $L_2 \subseteq \Sigma_2^*$ (over T_2). We say that L_1 reduces polynomially to L_2 if there exists a mapping $f : \Sigma_1^* \to \Sigma_2^*$ so that:

1. *$\forall x \in \Sigma_1^*$: $\quad x \in L_1 \Leftrightarrow f(x) \in L_2$*
2. *There exists a deterministic TM that computes f in polynomial time.*

We denote this by $L_1 \leq_p L_2$.

$\leq_p$ differs from $\leq_m$ only in the fact that the mapping must be computable in poly-time. This makes $\leq_p$ finer than $\leq_m$: if $A \leq_p B$ then $A \leq_m B$. Check whether some of the examples of $\leq_m$ earlier in this course notes, were actually $\leq_p$?

Theorem 5.2.8. *The relation $\leq_p$ is transitive, or more explicitly:*
if $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$ then $L_1 \leq_p L_3$.

Proof. Let f be the polynomial computable functions belonging to $L_1 \leq_p L_2$, and g to $L_2 \leq_p L_3$; M_f and M_g are the corresponding TMs: the first is a T_f-machine, the second a

T_g-machine, and T_f and T_g are polynomials. We prove that the function $h = g \circ f$ is good for $L_1 \leq_p L_3$.

Clearly, $s \in L_1 \Leftrightarrow h(s) \in L_3$, and h is computable by a TM M_h: first let M_f run on s, and when it is finished, move the head to the left, and then let M_g. We just need to show that M_h is polynomial.

Let s be a string with *a large enough* length be the input for M_h. When M_f is finished, the tape contains a string s_f with length at most $T_f(|s|)$. So, moving the head to the left costs at most a polynomial (in $|s|$) amount of work. Executing M_g on s_f takes at most $T_g(T_f(|s|)$ time, so together we have as an upper bound for the total time of M_h: $T_f(|s|) + T_f(|s|) + (T_g \circ T_f)(|s|)$ which is polynomial in $|s|$. ■

The next theorem has the same feel as an earlier theorem about $\leq_m$.

Theorem 5.2.9. *If $L_1 \leq_p L_2$ and $L_2 \in \mathbf{P}$ then $L_1 \in \mathbf{P}$.*

Proof. Let $L_1 \subseteq T_1^*$, $L_2 \subseteq T_2^*$, and $L_1 \leq_p L_2$ via the function $f : T_1^* \to T_2^*$ that is polynomial computable by TM M_f. Since $L_2 \in \mathbf{P}$, there exists a TM M_2 that decides L_2 in polynomial time. Construct TM M: it executes M_f and M_2 in sequence. Just as in the previous theorem, we can prove that M has polynomial time complexity. Moreover, M decides L_1, so $L_1 \in \mathbf{P}$. ■

We name two languages equivalent if each can be polynomially reduced to the other.

Definition 5.2.10. *Polynomial equivalence of languages*
Two languages L_1 and L_2 are polynomial equivalent (denoted by $L_1 \sim_p L_2$) if

$$L_1 \leq_p L_2 \;\; \textit{and} \;\; L_2 \leq_p L_1$$

The wording in this definition is justified by

Property 5.2.11. *The relation $\sim_p$ is an equivalence relation.*

Proof. We need to show that $\sim_p$ is reflexive, symmetric and transitive. Reflexivity is trivial, symmetry follows directly from the definition of $\sim_p$, and transitivity was already proven in Theorem 5.2.8. ■

As with any equivalence relation, we van consider the equivalence classes induced by the relation. They form a partition. $\mathbf{P}$ is consists of exactly three equivalence classes. Make sure you understand this.

NP–complete

Within **NP**, there exists a special class of languages: de most difficult of all in **NP** !

Definition 5.2.12. *The class* **NP**–*complete*
A language L is **NP**–*complete if and only if*

1. $L \in \mathbf{NP}$ *and*
2. *for each language $L' \in \mathbf{NP}$, it is true that $L' \leq_p L$.*

We denote the class of **NP**–*complete languages by* **NPC**.

Informally, an **NP**–complete language is so difficult that any other language can be translated to it in polynomial. In terms of problems: a problem is **NP**–complete if it has an solution (in the form of an algorithm) so that every other **NP** problem has a solution that can be deduced from the **NP**–complete solution by a polynomial reduction van de **NP**–complete. A priori, it is not clear whether **NP**–complete languages exist at all, neither whether **NPC** coincides with **NP** or **P**. However, the relevance of this class is captured in the following theorem:

Theorem 5.2.13.

1. **NPC** *is an equivalence class for* $\sim_p$.
2. *If* $\mathbf{NPC} \cap \mathbf{P} \neq \emptyset$ *then* $\mathbf{NP} = \mathbf{P}$.

Proof.

1. Suppose $L_1, L_2 \in \mathbf{NPC}$. By the definition of **NP**–completeness, both $L_1 \leq_p L_2$ and $L_2 \leq_p L_1$. So,L_1 and L_2 are polynomial equivalent. The class **NPC** is therefore part of one equivalence class C. Moreover, suppose that $L' \in C \setminus \mathbf{NPC}$, then for every $L \in \mathbf{NP}$, $L \leq_p L'$, meaning $L' \in \mathbf{NPC}$, so $C = \mathbf{NPC}$.
2. Suppose $L \in \mathbf{P} \cap \mathbf{NPC}$. Consider any other language $L' \in \mathbf{NP}$. Since $L \in \mathbf{NPC}$, we know that $L' \leq_p L$. Since also $L \in \mathbf{P}$, it follows from Theorem 5.2.9 that $L' \in \mathbf{P}$. So, $\mathbf{NP} = \mathbf{P}$.

■

The previous previous theorem shows that if one **NP**–complete problem can be solved in polynomial time, all problem in **NP** can be solved in polynomial time. There is however lots of evidence that **NP** does not equal **P**.[5]

Figure 5.2 shows the two possibilities: either $\mathbf{P} \neq \mathbf{NP}$ or $\mathbf{P} = \mathbf{NP}$.

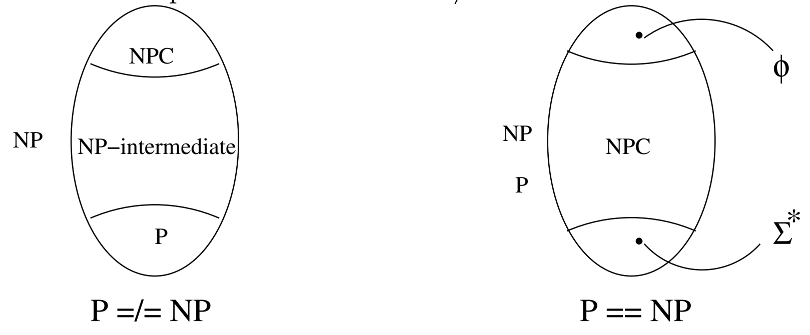


Figure 5.2: The possible relation between **P**, **NP** and **NPC**

The set **NP**-intermediate is not empty if $\mathbf{P} \neq \mathbf{NP}$. A candidate **NP**-intermediate problem is *graph isomorphism*.

The first problem proven to be in **NPC** is the **satisfiability** problem, often abbreviated by SAT. We need some definitions to specify the SAT language. Let $U = \{u_1, u_2, \ldots, u_n\}$ a finite set of boolean variables, i.e. each variable u_i can take the values *true* or *false*. A literal (or atom) from U we mean either one of the variables u_i, or its negation denoted by $\neg u_i$ (read this as *not* u_i). Consider a formula in Conjunctive Normal Form (CNF) over U, i.e. a conjunction of disjunctions of literals over U as in the following example: $C = (u_1 \vee u_2) \wedge (u_2 \vee \neg u_4 \vee u_7) \wedge (\neg u_2 \vee u_3)$.

The SAT problem is: given a formula in DNF, does there exist a truth assignment to the variables so that the formula is true?

More precisely, SAT is the language of DNF formulas that have a satisfying assignment.

Stephen A. Cook[6] proved in 1971

Theorem 5.2.14. *The Cook-Levin Theorem.* $SAT \in \mathbf{NPC}$

We do not study the proof of this theorem in this course. What you should be able to do is: show that $SAT \in \mathbf{NP}$, once using the non-deterministic definition of **NP** , and once using the verifier definition. What is the size of the certificate? How did you encode the language?

Once the first **NP**–complete problem was know, many more **NP**–complete were found. Here are just a few examples: Hamiltonian graphs, N-colorable graphs, subgraph isomorphism, the knapsack problem, pancake sorting, 0-1 integer programming, lots of combina-

[5]On the other hand ... Donald Knuth believes in $\mathbf{P} = \mathbf{NP}$.

[6]Turing award 1982

torial problems and even puzzles like Sudoku.

A language L that fulfills only the second condition of Definition 5.2.12, is named *NP-hard*. One also uses **NP**–hard for optimization problems whose decision version is **NP**–complete. There exist **NP**–hard problems that are not **NP**–complete (i.e. they are not in **NP**): SAT even has a polynomial reduction to H_{TM}, and so has every **NP**. Work this one out!

Examples of polynomial reductions

SAT to 3-SAT

3-SAT is SAT with the restriction that each disjunction in the DNF has exactly three literals. One formula in 3-SAT is $(a \vee b \vee \neg c) \wedge (\neg a \vee b \vee c)$. Every formula in DNF can be transformed to that format. Consider each disjunction in the formula. There are three cases:

- the disjunction contains less than three literals: repeat one of the literals until there are three
- the disjunction contains exactly three literals: just leave them unchanged
- the disjunction contains strictly more than three literals: the disjunction is of the form $a_1 \vee a_2 \vee ... \vee a_n$ met $n > 3$.

  Take a new boolean variable, say b, and replace this one disjunction by the two disjunctions

  (1) $b \vee a_{n-1} \vee a_n$ and
  (2) $\neg b \vee a_1 \vee a_2 \vee ... \vee a_{n-2}$

  Repeat all this until no disjunction contains more than three literals.

Now check the following about the transformation:

* a formula $\in SAT$ is mapped to a $\in$ 3-SAT
* a string $\notin SAT$ is mapped to a string $\notin$ 3-SAT
* the transformation can be implemented on a polynomial TM

3-SAT $\in$ **NP**: members of 3-SAT have a polynomial certificate and a verifier that uses it. Putting things together: 3-SAT $\in$ **NPC**. You should now be able to show that n-SAT $\in$ **NPC** whenever $n \geq 3$, but ...

2-SAT has a polynomial[7] algorithm. One find in literature even that 2-SAT has a linear algorithm, but that is on a random access machine, not on a TM. In any case, 2-$SAT \notin$ **NPC** ... or is it?

reducing 3-SAT to k-CLIQUE

The language k-CLIQUE consists of tuples $\langle G, i\rangle$ in which G is a graph with a clique of size i, or otherwise said: G contains a subgraph isomorphic to K_i. Two examples show how the reduction works:

Example 5.2.15. *The formula in 3-SAT is* $(a \vee b \vee c) \wedge (a \vee \neg b \vee \neg c) \wedge (\neg a \vee b \vee \neg c) \wedge (\neg a \vee \neg b \vee c)$

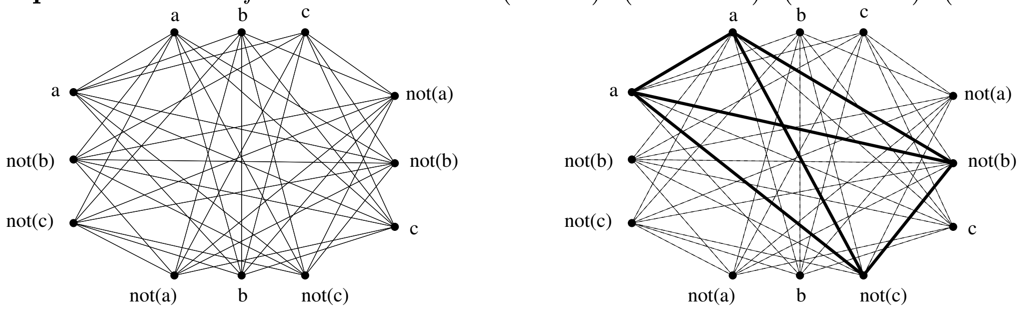


Figure 5.3: The graph corresponding to the first formula, and a 4-clique

Figure 5.3 shows at the left the graph resulting from the reduction:

- *each occurrence of a literal corresponds to one vertex with as its name the literal*
- *there is an edge between two vertices that are* compatible*: p and ¬p are not compatible and two vertices from the same disjunction are not compatible either*

The incidence matrix of this graph can be computed in polynomial time. The required clique size (4 in the example) is the number of disjunctions in the formula.

Convince yourself that the graph has a clique of the required size if and only of the formula belongs to SAT.

Example 5.2.16. *The reduction can be applied to smaller disjunctions as well. We take as formula:* $(a \vee b) \wedge (\neg a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee \neg b)$

Because the graph has no 4-clique, the formula does not belong to SAT. Of course, the graph does have k-cliques: the maximal k gives us the maximal number of disjunctions

[7]in the number of occurring literals

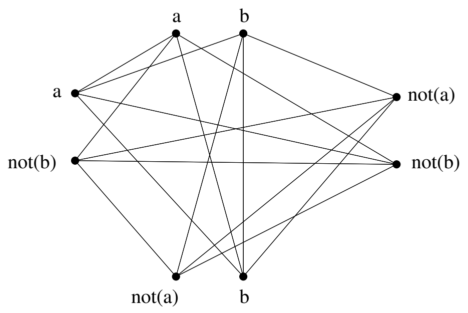


Figure 5.4: The graph corresponding to the second formula: there is no 4-clique

that can be true at the same time. In this way, two optimization problems are connected by a polynomial reduction.

Selfie:

- did you see a correspondence between the reduction from SAT to 3-SAT and how we put a CFG into Chomsky Normal Form?
- reduce 3-SAT to SAT
- which was easy because 3-SAT $\subsetneq SAT$, so here are some general questions:

  * does $A \subseteq B$ and $A \in \mathbf{P}$ imply that $B \in \mathbf{P}$?
  * does $A \subseteq B$ and $A \in \mathbf{NP}$ imply that $B \in \mathbf{NP}$?
  * does $A \subseteq B$ and $A \in \mathbf{NPC}$ imply that $B \in \mathbf{NPC}$?
  * does $A \subseteq B$ and $B \in \mathbf{P}$ imply that $A \in \mathbf{P}$?
  * does $A \subseteq B$ and $B \in \mathbf{NP}$ imply that $A \in \mathbf{NP}$?
  * does $A \subseteq B$ and $B \in \mathbf{NPC}$ imply that $A \in \mathbf{NPC}$?

Why are some problems to difficult?

For SAT, one could argue as follows: there are 2^n possible assignments (n is the number of boolean variables), and our intuition tells us that we need to check all of them to find the good one (or on average half of them) ...

For Hamiltonian cycles, we would argue: the number of simple cycles in a graph is exponential in the number of vertices, and our intuition tells us ...

For Minimal Spanning Tree, we would argue: the number of spanning trees for a graph with n vertices can be at worst n^{n-2} - that is even worse than exponential, so our intuition ... would be totally wrong. Indeed, we saw a greedy polynomial algorithm for this problem:

Prim and Kruskal. For MST, we relied on a graph theorem and that guaranteed the correctness of a polynomial algorithm without the need to check all possibilities.

Perhaps we have not yet proven the the right theorem for SAT, the theorem that results in a polynomial algorithm for SAT, but if we do, we will need to adjust our intuition just as with Karatsuba and multiplication.

The time hierarchy

As far as we discussed it, the structure of **P** is rather uniform. The time hierarchy theorem changes that: below is a correct version, but not the strongest possible one:

Theorem 5.3.1. $DTIME(f) \subsetneq DTIME(f^2)$ *for every* time-constructible f.

Loosely speaking, a function f is time-constructible if f can be computed on a TM in $O(f)$ steps and $f(n) \geq n$. Most probably, you will never during your life meet a (non-trivial) non-time-constructible function :-)

R. Stearns and J. Hartmanis proved in 1965 the first version of the time hierarchy theorem, quite a few years before the notion of **NP**–completeness came to life. For that - and other things - they got the Turing award in 1993.

It is important to realize that strictly more problems can be solved with a $O(n^4)$ algorithm than with $O(n^2)$: time is a resource, and it is limited. This is a good moment to think about the following:

- how many languages are in $DTIME(n^k)$?
- how many languages are in **P** ?
- how many languages are in **NP** ?
- if you answered *countable infinite* to one or more of these questions, would that also be effectively enumerable?

co-NP

Since the definition of **NP** treats elements of the language differently from the one not in the language, it is worthwhile investigating the complement of languages in **NP** :

Definition 5.4.1. *co*-**NP** *: co*-**NP** $= \{L|\overline{L} \in \textbf{NP}\}$

Be careful: co-**NP** does not equal $\overline{\mathbf{NP}}$. The latter contains non-decidable languages like A_{TM}, but every language in co-**NP** is decidable (why?). Some examples provide the intuition that co-**NP** might be different from **NP** .

- $SAT \in \mathbf{NP}$ because we have a short (polynomial) certificate for satisfiable formulas; $\overline{SAT} \in co\text{-}\mathbf{NP}$; how about certificates for elements in $\overline{SAT}$; $\overline{SAT}$ contains the formulas for which no assignment is satisfying ... could that have a short certificate? nobody knows
- consider $TAUTOLOGY$; it is the language of formulas that are true for **every** assignment to the variables; $\overline{TAUTOLOGY}$ is definitely in **NP**: a short certificate would be an assignment that makes the formula false; but a short certificate that can be verified in polynomial time so in case the formula is a tautology ... nobody knows
- the set of sorted sequences is a language in **NP**; a certificate for a non-sorted sequence would be an index i so that the ith element is larger than the (i+1)th element; so the set of sorted sequences belongs to co-**NP** as well; yu could have argued: the set of sorted sequences is in **P** , so also in co-**NP**; that is right, but you would have missed thinking up a non-trivial polynomial certificate
- COMPOSITE (the set of composite numbers) has a short certificate: a divisor of the number; it is difficult to think of a short certificate for PRIME (which is $\overline{COMPOSITE}$; on the other hand, since 2004 we know that PRIME $\in$ **P**, so this short certificate indeed exists ... (can you think of one?)

After these examples, you should understand that $\mathbf{P} \subseteq \mathbf{NP} \cap$ co-**NP** , and that **NP** =co-**NP** is not a trivial question. You must also be able to argue that $\mathbf{P} = \mathbf{NP}$ implies **NP** = co-**NP** . And what if $SAT \in$ co-**NP**?

NP–complete or P?

In this section, we merely want to point out that the details how a problem is specified really matter. Consider the following two languages:

- k-CLIQUE = $\{\langle G, i\rangle|$ i an integer, G a graph with an i-clique$\}$
- 7-CLIQUE = $\{\langle G\rangle|$ G a graph with an 7-clique$\}$

We showed that k-CLIQUE is in **NPC** (we reduced SAT to it), but 7-CLIQUE is polynomial: let n be the number of vertices in G; there are C_n^7 subsets of vertices that potentially form a 7-clique. Checking one of them is polynomial in the size of G, say $O(n^c)$ for some c. Moreover, $C_k^7 = O(n^7)$ so we have a naive algorithm with time complexity $O(n^{c+7})$. The encoding of G has size $O(n^2)$, so the 7-CLIQUE is in **P** .

This phenomenon shows up quite often: by keeping one or more parameters of the problem constant, we get a polynomial version of an **NP** -complete problem.

Reconsider the following two problems:

- $A_{TM} = \{\langle M, w\rangle | w \in L_M\}$
- $E_{TM} = \{\langle M\rangle | \epsilon \in L_M\}$

By keeping one parameter in A_{TM} fixed, we get E_{TM}. Did that pay off in this case?

Space complexity

Space is a resource: has your Java program ever stopped unexpectedly because of stack overflow? Or has the progress of your program ever become unacceptably slow because of excessive swapping or garbage collection? If so, you know that space is a (limited) resource, and space complexity deals with it. One could try to define the space complexity of an algorithm (on a TM) as the number of cells that are read/written on the tape. That would not be so useful:

- for non-trivial problems, the input needs to be read completely, so space complexity could not be sublinear: the space cost of the input is indeed unavoidable
- on the other hand, many algorithms need very little extra **extra** space (besides the input)

The latter should be familiar. E.g. testing whether a sequence is sorted needs an index and a fixed amount of space for comparing the two consecutive cells in the sequence: a constant space overhead.

Before we start with the definitions, think about this: one can reuse space, one cannot reuse time! So one expects to need *less* space than time for solving a given problem.

The definition of space complexity is based on a Turing machine model with

- one read-only tape containing the input string
- one read-write tape

The space cost of an algorithm is equal to the number of cells used on the R/W tape.

Definition 5.6.1. *DSPACE(f) is the set of languages decided by a deterministic Turing machine using $O(f)$ cells on the R/W-tape.*

Property 5.6.2. *If the function f is computed on a T-time TM, then $|f(s)| \leq T(|s|)$.*

This essentially says that a TM cannot write on more tape cells than it makes steps.

Now, you argue that $DTIME(f) \subseteq DSPACE(f)$.

Definition 5.6.3. **PSPACE** $= \cup_{k\geq 1} DSPACE(n^k)$

It should be clear now that **P** $\subseteq$ **PSPACE**.

Just as for time, there exists a space hierarchy theorem. A.o. $DSPACE(f) \subsetneq DSPACE(f^2)$ for *decent* f.

Definition 5.6.4. **NPSPACE** *is like* **PSPACE** *, but with non-deterministic Turing machines.*

The inclusion **PSPACE** $\subseteq$ **NPSPACE** is easy to prove, and one might suspect that **PSPACE** $\neq$ **NPSPACE**, or that it is unknown. Surprise ... **PSPACE** $=$ **NPSPACE** $=$ co-**PSPACE**. The fact that non-determinism does not change **PSPACE** is related to the fact that space can be reused: a non-deterministic TM can be simulated deterministically with little space overhead. That provides the intuition that co-**NPSPACE** must equal **NPSPACE**.

Less than linear space? This is also possible. Consider the language of even numbers: one can decide it with no extra space at all! This works actually for all regular languages. There exist also languages beyond RegLan that can be decided in sublinear space. As an example, we consider the problem to decide whether there exists a path between two given vertices in a given directed graph (V, E). We describe (in pseudo-code) a $log(n)^2$-space algorithm: n is the number of vertices.

```
function EXISTSPATH(a,z,l)
    if ($l = 0$) then
        return(a = z)
    end if
    if $(a, z) \in E$ then
        return(TRUE)
    end if
    for all ($v \in V$ do
        if EXISTSPATH$(a, v, l/2)$ $\wedge$ EXISTSPATH$(v, b, l - l/2)$ then
            return(TRUE)
        end if
    end for
    return(FALSE)
end function
```

EXISTSPATH(a, z, n) returns TRUE if there exists a path from a to z with length at more l. Its time complexity is horrible. If you imagine it is executed on a classical machine (your laptop), you see that no more than $\log(n)$ stack frames are needed, and that each stack frame has size at most $\log(n)$ bits. The algorithm can be implemented on a DTM with the same space complexity.

It is also interesting that one can trade space for time: you can make a version of EXISTSPATH that uses less time, but more space, e.g. by using a depth-first backtracking search through the graph, while you remember the visited nodes: that is $O(n)$.

Algorithms (and problems requiring them) that use log-space are interesting enough to warrant their own class:

Definition 5.6.5. $\mathbf{L} = DSPACE(\log)$

Log-space algorithms are important, e.g. in the context of large databases, or long DNA-sequences: with the log of the space taken by a database, you can keep a fixed number of pointers (in binary) and often that is often enough to traverse the database and answer a set of queries.

A sequence of inclusions

We know and understand enough at this point to justify the following sequence of inclusions:

$$\mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} = \mathbf{NPSPACE} \subseteq \mathbf{EXP}$$

Thanks to the hierarchy theorems, we know for sure that $\mathbf{L} \neq \mathbf{PSPACE}$ and $\mathbf{P} \neq \mathbf{EXP}$. This means that some of the inclusions are strict, but currently nobody knows which ones.

Complexity and Chomsky hierarchy

What is the connection between the complexity of a language and its place in the Chomsky hierarchy?

It is easy to see that a regular language can be decided in $O(n)$ steps if $n = |s|$ on an FSA. Whether this is also possible on a DTM remains to be proven ... by you. One needs very little extra space to decide a regular language, actually none at all. Besides, only regular languages can be decided with strictly less than log-SPACE.

Context free languages can be recognized by a general $O(n^3)$ algorithm (on a 3-tape TM) algorithm known by the names of the authors as Cocke-Younger-Kasami. It is a form of *chart* or *tabular parsing*, or dynamic programming. Moreover, you only need linear space on the stack of the PDA, so $CFL \subset SPACE(n)$.

The class of non-deterministic LBA decides languages by using no more than the space occupied by the input, so context-sensitive languages constitute the class **NSPACE**(n).

Conclusion

A basic concern in the study of algorithm complexity [8] is the characterization of *fast* algorithms in a formal way, and also of problems allowing a fast algorithm. Very soon it was realized that the size of the input to the algorithm has an important role in this characterization.

Only in 1965 did Jack Edmonds propose to define *fast* as *polynomial in the input length*: this followed the general feeling that a polynomial algorithm might be fast enough, but an exponential one certainly is not. The (decision) problems with a fast solution form together the class **P**. Problems not in **P** are considered *intractable* - at least for practical means.

[8] Its roots are at least 150 years old.

By the end of the 1960's, it became clear that for some seemingly simple problems (like SAT) nobody was able to construct a polynomial algorithm. Steve Cook cooked up the notion of *verifiable in polynomial time* and the class **NP** was born. In 1971 Cook proved that within **NP** there exists a class of *most difficult* problems, i.e. the class **NP**-complete, and that **NP**-complete is not empty. Leonid Levin (USSR) proved the same result almost simultaneously and independently: he published in Russian, and - also because of the cold war - his result was not well-known in the West. The finishing touch was the understanding that one problem in **NP**-complete $\cap$**P** is enough to collapse **P** and **NP**, or to put it in words: verifying [a solution] would be as difficult as computing [it] (module the exponent in the polynomial of course). However, the strict inclusion of **P** in **NP** is an open problem. It feels like betting on the opposite is a bad choice, because there are so many NP-complete problems for which researchers have tried to find a polynomial algorithm ... and they failed. Still, you should make up your own mind about this fundamental problem in CS!

Currently, the separation between **NP**-complete and **P** is important because one should not hope for an efficient solver for intractable problem (for ever growing input size). Most everyday real problems are **NP**-complete (vehicle routing, exam rostering ...) or they have algorithms with an impractical exponent (PRIME). Fortunately, efficient solvers based on different principles have been developed, as well as the theory on which they are based. We have very fast probabilistic algorithms for checking whether a number is prime (they fail almost never), we have heuristics and approximation schemes for optimization problems for which we have no polynomial algorithm (at this moment). Sometimes we can fix a parameter and that results in a tractable problem. Complexity theory has ramifications in other scientific activities like cryptography, quantum computing, information theory, circuit-design ... , but for now, the central question is **P** =?**NP**.

References

- Sanjeev Arora and Boaz Barak, “Computational Complexity: A Modern Approach”, Cambridge University Press