8. Basic algorithmic strategies

8.1 Self-referential problems: Recursion

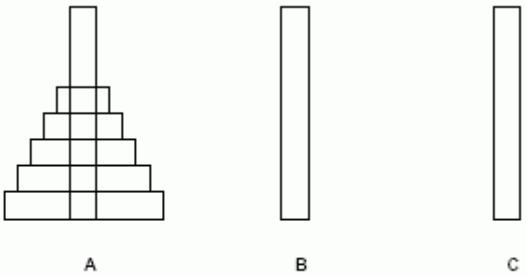
recursive function: calls itself, see "factorial" example in the Java chapter

recursive definition: the notion to be defined appears in the definition (but with some sort of simplification), see definition of the semantics of a propositional formula

recursive solution of problems: sometimes possible if the problem involves some self-reference
– careful analysis of the problem necessary!

Example: The "Tower of Hanoi" game

Three pegs; two of them empty (B and C); one (A) with a tower of n disks of decreasing sizes.



Goal configuration: Pegs A and B empty, all disks on C with decreasing sizes.

A move: consists of the transfer of the topmost disk from some peg to another peg.

Restriction: A larger disk may never be put on a smaller disk.

Solution strategy:

To transfer a tower of n disks from any peg X to another peg Y, using a third peg Z for help, do the following:

1. If $n = 0$, nothing is to be done.

2. If $n = 1$, just move the single disk from X to Y.

3. Otherwise, i.e. if $n > 1$, do the following in sequence:

3.1 Transfer the tower of the $n - 1$ topmost disks from X to Z, using Y as help peg.

3.2 Move the bottom disk of the original n-disk tower, now lying on top on peg X, from peg X to peg Y.

3.3 Move the tower of $n - 1$ disks from peg Z to peg Y, using X as help peg.

Recursive part: in steps 3.1 and 3.3 !

How do we make plausible that this strategy solves the problem?

To be shown: that in the process, a larger disk is never put on top of a smaller one.

Proof: Assuming that steps 3.1 and 3.3 work, 3.2 is the only possibly problematic step. But all disks smaller than the one moved in 3.2 are on the help-peg. QED

The solution seems to use a “trick”: It somehow seems **to use itself**! This is called **RECURSION**. Why or in which cases is this allowed?

Three principles are necessary for this to be allowed:

1. Each problem has a “size”; sizes are ordered, and the set of sizes is “well-founded”, i.e. for each size, there is only a finite number of smaller sizes.

2. For all cases which are small enough, the algorithm does NOT “use itself”.

3. For inputs for which the algorithm “uses itself”, the problem to be dealt with in the self-use is “smaller” than the input problem.

Are the three principles for the Tower-of-Hanoi problem fulfilled?

1. **“Size”:** natural numbers, the height of the tower to be moved. We will use the letter n for the problem size.

2. **Non-recursive cases:** Problems with sizes $n = 0$ and $n = 1$ can be solved without recursive invocation.

3. **Recursive cases:** An input problem of size n (with $n > 1$) is reduced to a partial problem of size $n - 1$.

Another recursive algorithm: “Quick sort”. Three principles:

1. **Size:** length of array to be sorted.

2. **Non-recursive cases:** lengths 0 and 1.

3. **Recursive cases:** length $n > 1$, reduced to two problems of same kind, with sizes

– $n - 1$ and 0 (worst case), or

– two problems of (approximately) the same size of (approximately) $(n - 1)/2$

Strategy for application of recursion in general:

1. **Reduce the original version of the problem** to a finite number of smaller problems.

2. **Solve the smaller problems recursively**. This results in some partial solutions.

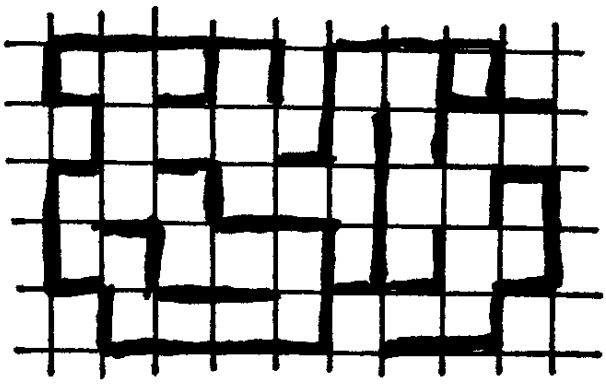
3. **Put together the partial solutions** to the full solution of the original problem.

4. **Check that the recursion-principles are fulfilled;** especially: ensure that recursive calls are always done with smaller problem sizes.

8.2 Search problems: Backtracking

Fundamental idea:
If you look for something in an unknown environment, first go *somewhere*. If that what you are looking for is not there, ***go back*** and try it in another location.
(Always keep book about what places you have already inspected!)

Example: Mastering a maze



Given data and problem: A maze of unknown size on a grid with quadratic cells. Sometimes, there is a wall between two cells, and sometimes, the way is open. Given two positions in the maze, is there a way from one to the other?

Strategy: Mark all cells reachable from the first position given. If during the marking process, the second position given is reached, stop and output “yes”. If not, output “no”.

More specific search algorithm

Each cell will either (1) be unmarked, or (2) carry one of the marks "n, e, s, w, visited".

– For n, e, s, w, this denotes the direction which has been used the last time the cell has been left,

– "visited" means that all directions have been tried.

Start with current position at pos1, then repeat the following until the process stops:

1. If the current position is pos2, output "yes" and stop.

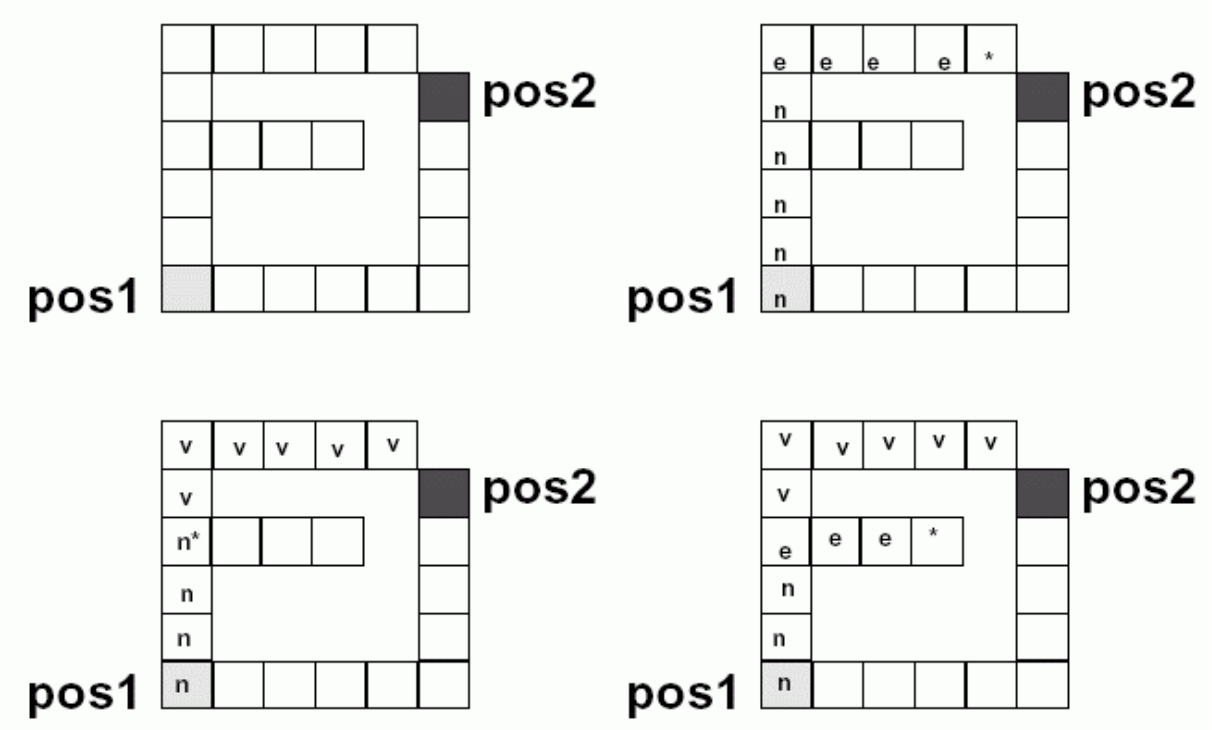
2. If the current position is pos1 and all reachable neighbors are marked, output "no" and stop.

3. Check if there is a reachable unmarked neighbor in some direction $d \in \{n, e, s, w\}$.

   3.1 **If yes**, mark current cell with d and move current cell to that neighbor.

   3.2 **If no**, mark the current cell as "visited" and go back to the cell from which the current cell has been entered for the first time. This is the only reachable cell in a direction d which is marked with $-d$. ($-d$ is just the opposite direction of d).

In the picture, four situations of the algorithm are shown. The star marks the currently visited cell.
"*v*" = visited



What happens if the search process reaches a dead end?

In this case, **step 3.2 is taken repeatedly, going back the way** in which the cell was reached before, until either
(1) a position with an untried direction is reached, or
(2) pos1 is reached.

This process is called **backtracking**: a general principle of different algorithms based on searching.

Conditions for backtracking are:

- The search space consists of a **finite set of situations**, and for each situation, there is a **subset of neighbouring situations**.

- In each situation, it is clear which of the neighbouring situations **have already been visited**, and from which of them the current situation has been entered **for the first time**.

If these conditions are fulfilled, a search strategy like for the grid can be used.

Another example of a problem solvable by backtracking:

Given: A positive integer n and a finite list L of positive integers.

Problem: Is there a set S of list elements of L such that the sum of the elements of S is n?

(Similar problem: Cutting a large board without rest into some smaller boards of given lengths.)

Situations **are represented by lists of strictly increasing indices into** L, starting with the empty list. Each such list represents the subset of selected elements of L.

If the sum of these elements is larger than n, we have run into a dead end: we do not have to check extensions of the list.

Backtracking means in this case: Shorten the index list and try another extension not yet tried out.

The **next extension** to be tried is the one where the last index in the current list is incremented by one.

If there is **no such next index**, we have also run into a dead end, and the next index **for the next-to-last element** is tried out.

Example: $n = 10$, $L = [6, 5, 3, 5, 3, 2]$

Start with empty sequence. Next is sequence 1_6 (indices give the associated values).

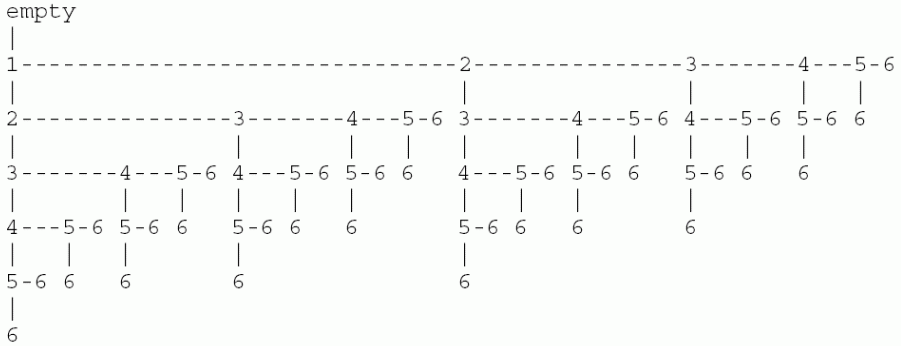
1_6
- $1_6 2_5$ (too long)
- $1_6 3_3$
  - $1_6 3_3 4_5$ (too long)
  - $1_6 3_3 5_3$
  - $1_6 3_3 6_2$ (no alternative for last index: update next-to-last)
- $1_6 4_5$
- $1_6 5_3$
  - $1_6 5_3 6_2$
- $1_6 6_2$ (no alternative for last index: update next-to-last)

2_5
- $2_5 3_3$
  - $2_5 3_3 4_5$ (too long)
  - $2_5 3_3 5_3$
  - $2_5 3_3 6_2$ – **solved!**

Backtracking as search in a tree (depth-first search)

– Each node represents an index sequence.

– Search stops as soon as the sum is too large.



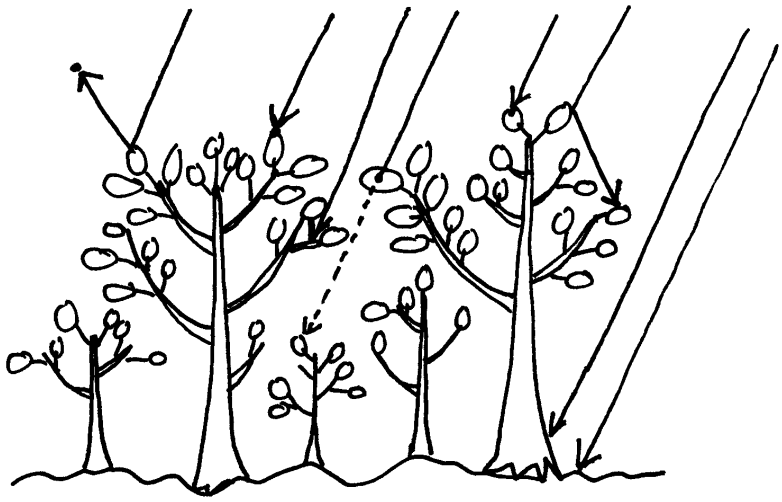
8.3 Simulation of discrete events: The Monte-Carlo method

In natural processes, often a very large number of objects or agents is involved.
Examples:

- sand particles in a sandstorm
- individual fishes in a maritime ecosystem
- cars in traffic flow
- light particles (photons) in an optical system
- speculators at the stock exchange ...

If these processes are to be simulated, it makes often sense to follow the fates of a much lower number of ***randomly*** selected representatives.
If the distribution of their parameters is the same (or similar) to that of the total set of objects or agents, conclusions can be drawn to the outcome of the process.
Because random parameters are involved: name "Monte-Carlo method" after the famous casino.

Example: Simulation of light interception and reflection in a virtual forest stand
- generate representative photons with random initial position and follow them through the stand ("photon-tracing").



Attention:
initial distribution of position in the sky (solar positions) and direction (inclusion of diffuse radiation?) should be realistic
– not a trivial task
(but more a question of astronomy, geography and meteorology than computer science)

Technical question: How to generate *random* positions or directions?

More simple version of this problem:

Problem: Simulate the **throw of a dice**. How can this be done in a computer?

Typical way: **Pseudo-random numbers,** or: random number generators (RNG)

– An algorithm which, after having been **initialized with a "seed" (start number)**, produces a list of n-bit integers in which no pattern can be recognized.

– There is **no true "randomness"**: started with the same seed, the algorithm always produces the same sequence.

– After at most 2^n elements, the sequence **enters a cycle.**

A well-tested RNG is the one based on the following formula:

$$r_{i+1} = (r_i * 16807) \bmod 2147483647$$

In systems with 32-bit integers, this formula cannot be used directly because the intermediate value $2147483646 * 16807$ can occur in the computation.

Access functions of RNGs

An RNG typically contains at least two access functions:

- One access function is used for **setting the seed**.
- Another access function is used for **getting the next number from the sequence**.
- Often, random **floating-point numbers** between 0 and 1 are needed. This can be had by a floating-point division of the current number by 2^n. Many libraries provide an access function which yields this value.
- Sometimes, further access functions are provided, e.g. for different representations of floating-point numbers.

in Java: see `http://www.cs.geneseo.edu/~baldwin/reference/random.html`.

Problems to look out for when using RNGs

The main problem is: What does it mean that **"no pattern can be recognized"**?

This can not be checked - there is **no complete list of possible patterns**!

Example: the lower bits are often "less random" than the higher bits.

Consequence: For getting a random sequence of bits, do not use just "mod 2" on the number sequence!

Designing random-number generators is a "black art": **Never use home-growns.**

8.4 Graph algorithms

What is a *graph* in computer science?

– a structure consisting of nodes and arcs which connect some of these nodes

More precisely:

Finite directed graph: (V, E), with:

– V: Finite set of vertices, and

– $E \subseteq V \times V$: finite set of edges

Mathematical model for a **finite number of elements**, with **directed connections** between them.

Examples for real-world structures representable by graphs

1. Topology of a town; V: Places in a town; $(a, b) \in E$ iff there is a direct connection between places a and b.

2. Direct connections in railway network; V: Railway stations; $(a, b) \in E$ iff there is a train going directly from station a to station b.

3. Acquaintances; V: Humans; $(a, b) \in E$ iff person a knows person b.

4. Parents; V: Humans; $(a, b) \in E$ iff person a is a parent of person b

5. Food web; *V*: animal species, (*a*,*b*) ∈ *E* iff *a* eats *b*

6. Gene regulation network; *V*: genes and enzymes,
   E: activation / inhibition relations between them.

Graph-Related Concepts

An edge of the form (a, a) is called a **cycle**.

A graph is **non-directed** if $\forall (a, b) \in E : (b, a) \in E$. Otherwise the graph is **directed**.

A **clique** of a graph (V, E) is a set $C \subseteq V$ s.t. $a, b \in C \Rightarrow (a, b) \in E$.

A **follower** of a vertex a is a vertex b s.t. $(a, b) \in E$.

A **precursor** of a vertex a is a vertex b s.t. $(b, a) \in E$.

An **incident edge** of a vertex a is an edge of the form (b, a).

Two vertices a and b are called **adjacent** iff $(a, b) \in E \vee (b, a) \in E$.

A finite **path** in a graph (V, E) is a finite sequence $v_0, v_1, \ldots, v_n$ of vertices s.t.

$$\forall i \in \{0, \ldots, n-1\} : (v_i, v_{i+1}) \in E$$

A path is called **proper** if it contains more than one vertex.

An **acyclic graph** is a graph such that there is no vertex a for which a proper path exists starting in a and ending in a.

A graph is **connected** if for all $a, b \in V$, there is a path from a to b or a path from b to a.

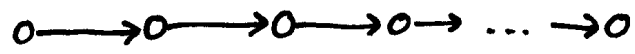
A vertex b is **reachable** from a vertex a if there is a path in the graph starting in a and ending in b.

Special forms of graphs:

Linear Lists

A **linear list** is a (special) graph with the following properties:

- Each vertex has **at most one precursor,**
- each vertex has **at most one follower,**
- there is exactly one vertex **with no precursor,** and
- there is exactly one vertex **with no follower,** and
- for different vertices v and v', there is **exactly one path,** either from v to v' or from v' to v.



Linear lists are important data structures.

Senseful operations for lists:

- append an element (at the end of the list)
- remove an element (from the end)
- insert an element
- delete an element (from an arbitrary position)
- concatenate two lists
- count the number of elements
- revert the order of the elements

...

Trees

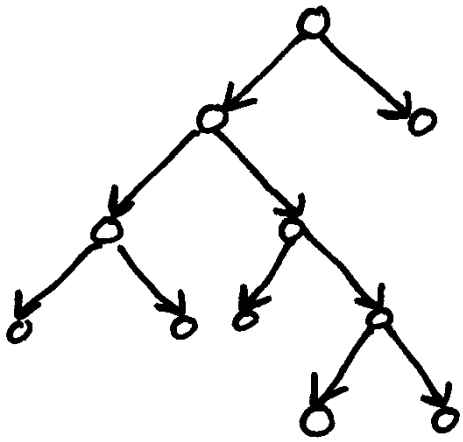
A **tree** is an acyclic graph such that

- there is at most one path from each element to another, and
- there is exactly one vertex without a precursor (the **root**).

Vertices in a tree without followers are called **leaves** of the tree.

Trees in computer science are normally drawn with the **root at the top,** spreading down.

A **binary tree** is a tree in which each vertex has at most two followers.



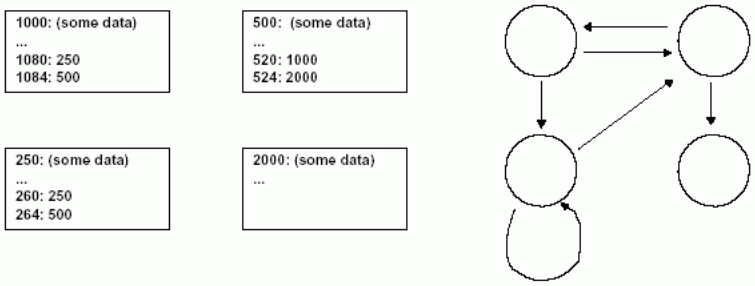
Trees are also important data structures.
Examples:
- decision trees (make a decision in each interior node!)
- space partitioning trees
- phylogenetic trees
- hierachical clustering of data (according to similarity)
- hierarchy of directories and files

...

Computer Representation of Finite Graphs

Two often used ways: **memory-block** representation, or **adjacency matrix** (we look into this in more detail)

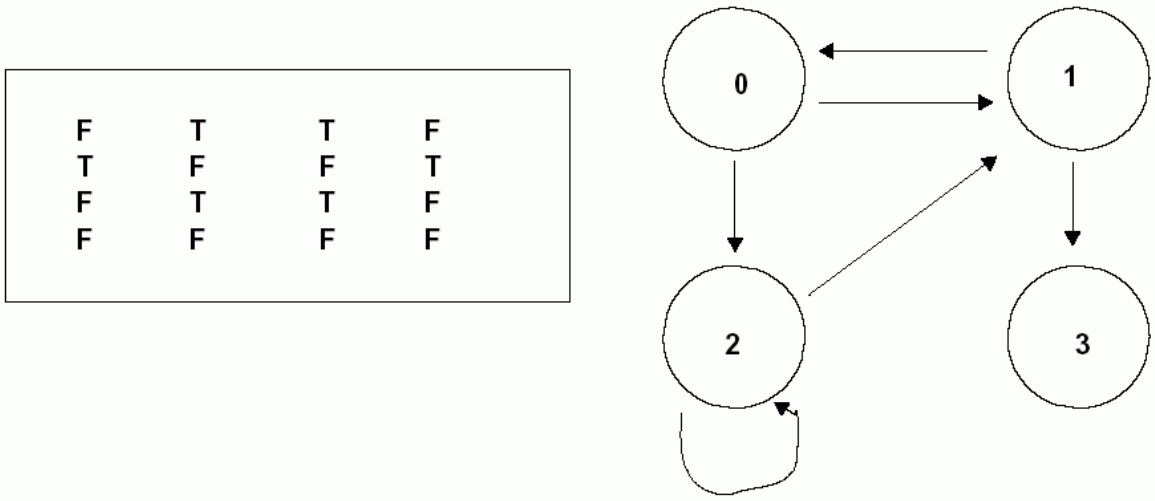
Memory-block representation: vertices as **blocks of memory**; edges as **memory addresses** in the memory block of the pre-node of the edge.



Representing Finite Graphs in an Adjacency Matrix

Elements of V are numbered through from 0 to $|V| - 1$. We denote the vertex with number i by v_i.

The adjacency matrix M is a boolean square matrix with $|V| \times |V|$ elements, in which $M(i, j)$ (row i, column j) is true iff $(v_i, v_j) \in E$.



Reachability Problem

Let (V, E) be a graph. How do you check if vertex v_j is reachable from vertex v_i?

Idea: we construct, from M, another Boolean matrix N in which $M^{\infty}(i, j)$ represents that there is a path from v_i to v_j. This would solve the reachability problem.

We use the **Warshall algorithm**:

Step 1: We define a special type of matrix multiplication for quadratic Boolean matrices. Let P and Q be quadratic Boolean matrices of the same size n. Then $P \times Q$ is defined by

$$(P \times Q)(i, j) = \bigvee_{0 \leq k < n} (P(i, k) \wedge Q(k, j))$$

(Disjunction for addition, conjunction for multiplication)

Verify that the following holds:

- **If** $P(i, k)$ represents if there is a path of up to x edges from v_i to v_k, and
- **if** $Q(k, j)$ represents if there is a path of up to y edges from v_k to v_j,
- **then** $(P \times Q)(i, j)$ represents if there is a path with up to $x + y$ from v_i to v_j.

Step 2: Define $N(i,j)$ as $M(i,j) \vee i = j$.

Verify that $N(i,j)$ represents if there is a path of up to one edge from v_i to v_j.

Step 3: Repeat $N := N \times N$ until the value does not change any more.

Verify that after n repetitions, $N(i,j)$ represents that there is a path with up to 2^n edges from v_i to v_j (use induction).

Verify that after a finite number of steps (logarithmic in the number of vertices), the value stops changing.

Step 4: In the result of step 3, $N(i,j)$ represents that there is a path from v_i to v_j.

Verify this.

Weighted Graphs and the Shortest Path Problem

Weighted graphs: A graph (V,E) with a function $w : E \to \mathbb{R}$ which assigns a **weight** to each edge.

Weights of a graph can be used, for example, to model **distances between vertices**.

Now, each **path** can be assigned a weight: The **sum** of the weights of its edges.

Representation of weighted graphs: with the **weights** in the array instead of just Boolean values.

Shortest Path Problem

Given: The distance matrix of a weighted graph, i.e. a matrix M in which $M(i,j)$ represents the weight of the edge between v_i and v_j. If there is no edge from v_i to v_j, the value is ∞.

Wanted: A matrix N in which $N(i,j)$ represents the length of the shortest path between v_i and v_j.

Algorithm: Similar to Warshall's:

- Define $P \times Q$ for quadratic matrices with n rows via

$$(P \times Q)(i,j) = \min_{0 \leq k < n} P(i,k) + Q(k,j)$$

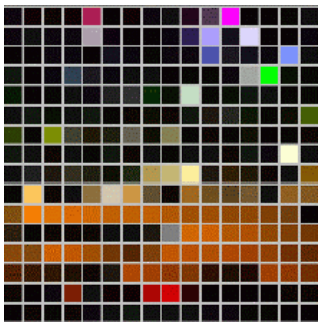
- Define $N(i,j)$ as $M(i,j)$ for $i \neq j$, and as 0 otherwise.

- Iterate $N := N \times N$ until the value does not change any more.

Verify this algorithm! Use steps as for Warshall's.

8.5 Image processing

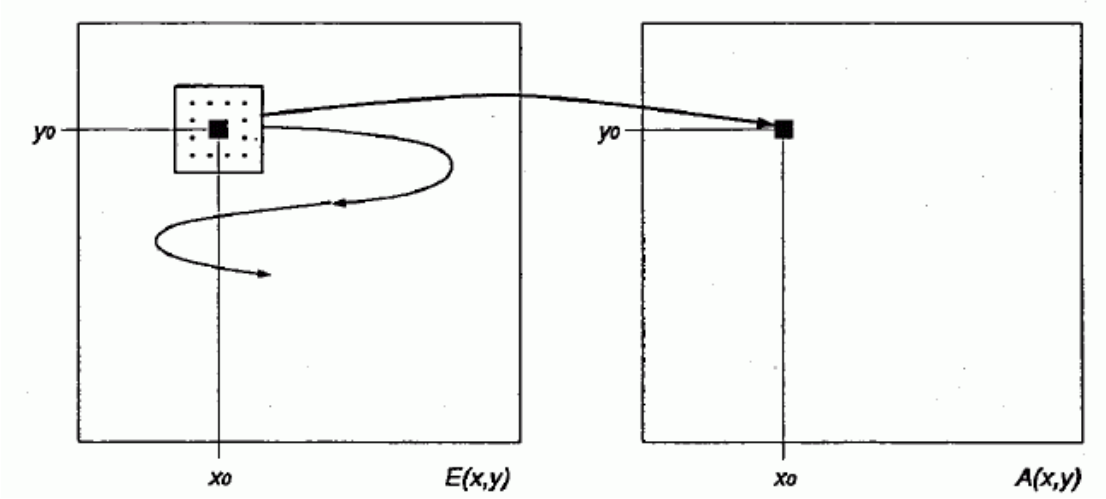
Images in computer graphics:
usually large rectangular matrices
of grey values (integers) or colour specifications (consisting usually of 3 integer values).



Basic tasks in the processing of images by computers:

- removal of "noise", smoothening
- edge detection
- segmentation (to distinguish objects from background and from each other)
- feature extraction (e.g., size of objects)
- classification and pattern recognition

Simple algorithm for smoothening a (greyscale) image:
Calculate the mean value for a block of pixels, use this value as the new grey value for the central pixel
iterate this for all pixels



mathematically: "convolution" of the image matrix with a (small) matrix of weights

original image ↓ convoluted images

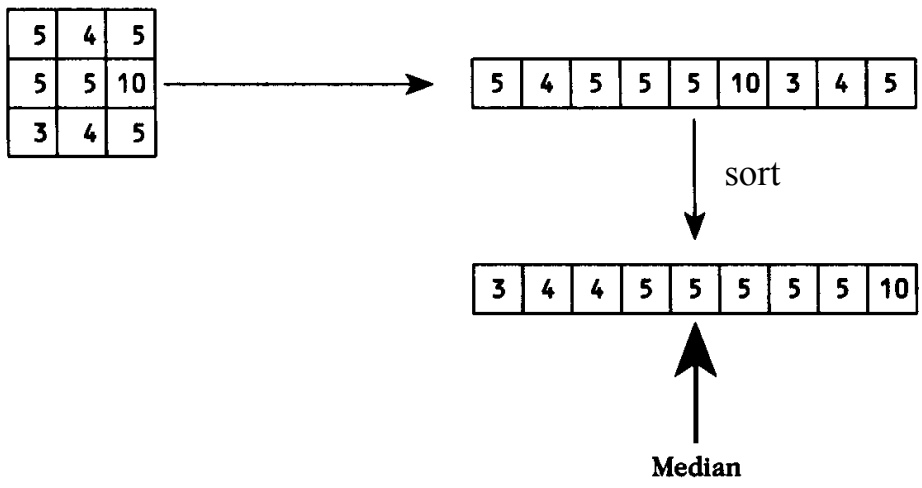
$$h = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$$h = \frac{1}{49}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

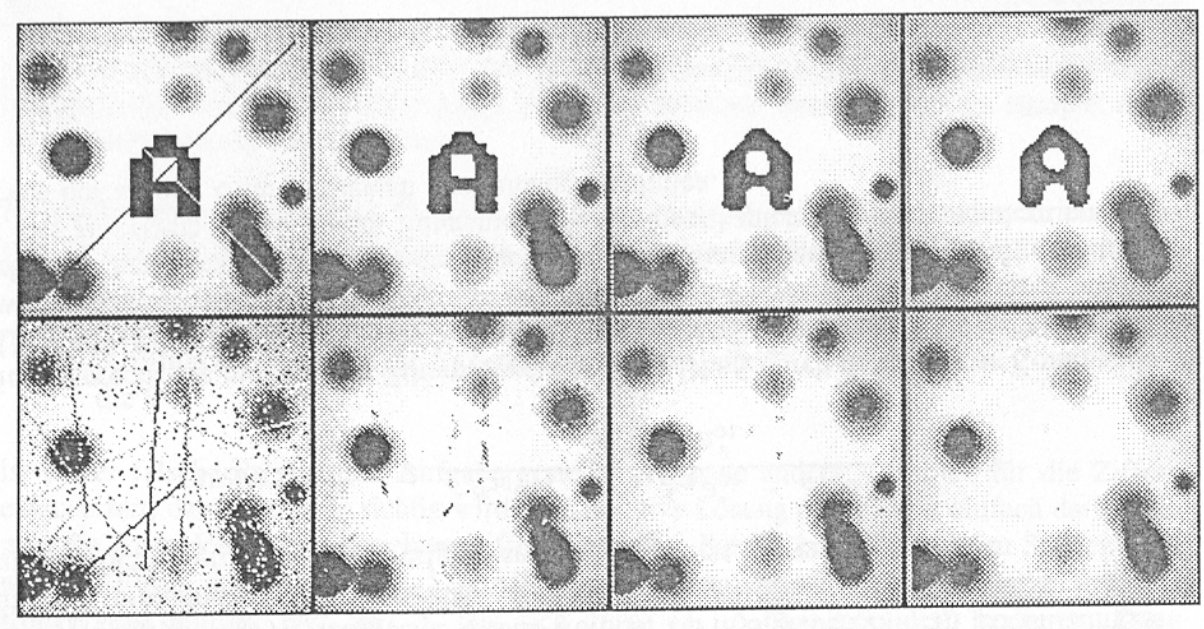
These are examples of *linear filters*.

Better suited for removal of erroneous pixels ("noise") is a nonlinear filter, the *median filter*:

First sort the entries in a window, then take the middle value as the new pixel value. Iterate this for all positions.



Application of the median filter (window lengths 3, 5, 7):
original images ↓



Edge detection:
where is a large *difference* between neighbour pixels?

Use linear filters again:

Roberts filter

$$h_1 = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \quad h_2 = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

"edge image": add the absolute values of the results of the convolutions with both matrices

h1 h2 edge image

other edge detection filter:
Sobel filter

$$h_1 = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \qquad h_2 = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Next steps:

- thinning the edges
- closing them to make contours
- approximation of contours by line segments or spline curves

→ segmentation by contours

Alternative: *Region-based segmentation*

Region growing (floodfill algorithm):

- set a start pixel in the interior of the region
- inspect all neighbour pixels if they have the same (or a very similar) colour
- if so, mark them as belonging to the region
- continue until no neighbour pixels remain
- do the same for other regions

Feature extraction
Typical features of regions / objects in images:

- area *A*
- longest diameter
- circumference *c*
- form factor ($c^2/4\pi A$) – is 1 for a circle
- center of gravity
- average grey value ...

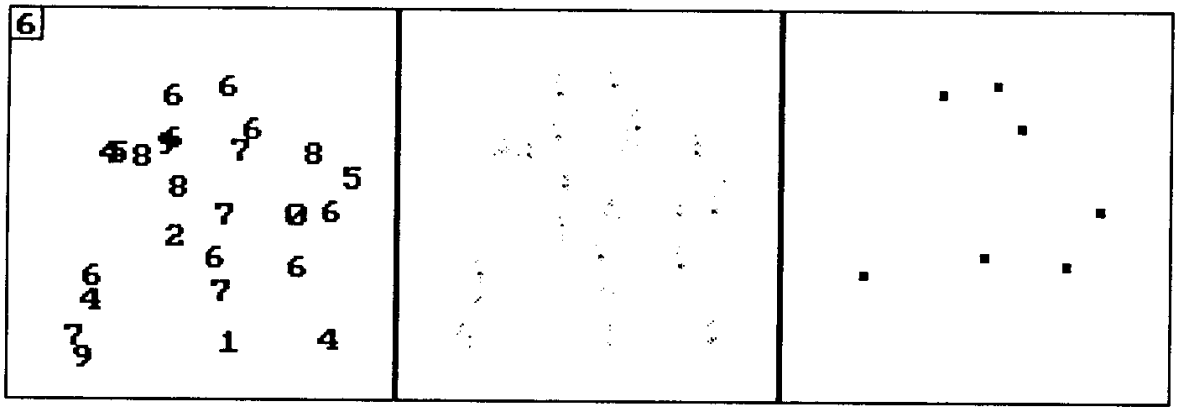
Pattern recognition

Simple method:
using the statistical parameter "correlation coefficient"

$$\frac{\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})}{\sqrt{(\sum_{i=1}^{n}(x_i-\bar{x})^2)(\sum_{i=1}^{n}(y_i-\bar{y})^2)}}$$

insert for x_i the pixel grey values of the picture and for y_i those of the pattern

Example: Search for the pattern "6" in an image



given image and given pattern (upper left corner) | correlation coeff. for all possible positions of the pattern | clusters of corr.coeff.

Problem:
The calculation of all the correlation coefficients costs much time

- make first segmentation of the picture to distinguish the characters from the background, then restrict search to the positions of the characters
- use other measures of similarity ...