Algorithms and Data Structures II.
Lecture Notes:

Elementary graph algorithms

Ásványi Tibor – asvanyi@inf.elte.hu

October 2, 2021

Contents

References

[1] Ásványi, T, Algorithms and Data Structures I. Lecture Notes
http://aszt.inf.elte.hu/~asvanyi/ds/AlgDs1/AlgDs1LectureNotes.pdf

[2] Burch, Carl, B+ trees
(See http://aszt.inf.elte.hu/~asvanyi/ds/AlgDs2/B+trees.pdf)

[3] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.,
Introduction to Algorithms (Third Edititon), *The MIT Press*, 2009.

[4] Cormen, Thomas H., Algorithms Unlocked, *The MIT Press*, 2013.

[5] Narashima Karumanchi,
Data Structures and Algorithms Made Easy, *CareerMonk Publication*, 2016.

[6] Neapolitan, Richard E., Foundations of algorithms (Fifth edition), *Jones & Bartlett Learning*, 2015. ISBN 978-1-284-04919-0 (pbk.)

[7] Shaffer, Clifford A.,
A Practical Introduction to Data Structures and Algorithm Analysis, Edition 3.1 (C++ Version), 2011
(See http://aszt.inf.elte.hu/~asvanyi/ds/C++3e20110103.pdf)

[8] Tarjan, Robert Endre, Data Structures and Network Algorithms, *CBMS-NSF Regional Conference Series in Applied Mathematics*, 1987.

[9] Weiss, Mark Allen, Data Structures and Algorithm Analysis in C++ (Fourth Edition),
Pearson, 2014.

1 Simple graphs and their representations ([3] 22, [7] 11)

Graphs can model networks, complex processes etc. The models must be represented in the computer, in a mathematical way, or in an intuitive manner for better understanding.

1.1 Basic notions of graph theory

Definition 1.1 *A* graph *is a* $G = (V, E)$ *ordered pair where* V *is the finite set of vertices,* $E \subseteq V \times V \setminus \{(u, u) : u \in V\}$ *is the set of edges. If* $V = \{\}$*, then the graph is empty. If* $V \neq \{\}$*, then the graph is nonempty. (The* vertices *of graphs are also called* nodes*.)*

This definition excludes parallel edges (we cannot distinguish two (u, v) edges) and self-loops like edge (u, u).

In these lecture notes *graph* means *simple graph* (without parallel edges and self-loops).

Definition 1.2 *Graph* $G = (V, E)$ *is* undirected*, if for each edge* $(u, v) \in E$ *:* $(u, v) = (v, u)$*.*

Definition 1.3 *Graph* $G = (V, E)$ *is* directed *or* digraph*, if for each pair of edges* $(u, v), (v, u) \in E$*:* $(u, v) \neq (v, u)$*.*

Definition 1.4 *Given graph* $G = (V, E)$*,* $\langle u_0, u_1, \ldots u_n \rangle$ *(*$n \in \mathbb{N}$*) is a* path*, if for each* $i \in 1..n$*:* $(u_{i-1}, u_i) \in E$*. These* (u_{i-1}, u_i) *edges are the edges of the path. The* length *of this path is* n*, i.e. equal to the number of edges of the path.*

Definition 1.5 *Given path* $\langle u_0, u_1, \cdots, u_n \rangle$*, its* subpath *is path* $\langle u_i, u_{i+1}, \cdots, u_j \rangle$ *where* $0 \leq i \leq j \leq n$*.*

A path $\langle u_0, u_1, \ldots u_n \rangle$ *is* loop*, if* $u_0 = u_n$*, and the edges of the path are pairwise distinct.*

A loop $\langle u_0, u_1, \ldots u_{n-1}, u_0 \rangle$ *is* simple loop*, if vertices* $u_0, u_1, \ldots u_{n-1}$ *are pairwise distinct.*

A path contains a loop*, if it has some subpath which is a loop.*

A path is acyclic*, if it does not contain loop.*

A graph is acyclic*, if all the paths of the graph are acyclic.*

Definition 1.6 *A* DAG *is a directed acyclic graph.*

Definition 1.7 *Given digraph $G = (V, E)$, its* undirected pair *is undirected graph $G' = (V, E')$ where $E' = \{(u, v) : (u, v) \in E \vee (v, u) \in E\}$.*

Definition 1.8 *An undirected graph is* connected*, if there is some path between each pair of its vertices.*

A digraph is connected*, if its undirected pair is connected.*

Definition 1.9 *An* undirected tree *is an undirected, acyclic, connected graph.*

Definition 1.10 *Given a digraph, its vertex u is* generator vertex *of the graph, if each vertex v of this graph is available form u, i.e. there is some path $u \rightsquigarrow v$.*

Property 1.11 *If a digraph has generator node, then it is connected. But there are connected digraphs without generator node.*

Definition 1.12 *T is a* rooted tree*, if it is a digraph with generator node, and its undirected pair is an acyclic graph.*

The generator vertex of a rooted tree is called its root*.*

Digraph T is directed tree*, if it is* rooted tree*, or it is empty.*

Property 1.13 *Given a (directed or undirected) nonempty tree with n vertices, it has $n - 1$ edges.*

Definition 1.14 *Graph $G' = (V', E')$ is* subgraph *of graph $G = (V, E)$, if $V' \subseteq V \wedge E' \subseteq E$, and both graphs are directed or both graphs are undirected.*

Graph G' is proper subgraph *of graph G, if graph G' is subgraph of graph G, but $G' \neq G$, and G' is not empty.*

Definition 1.15 *Two (sub)graphs are* disjunct*, if they have no common vertex.*

Definition 1.16 *Nonempty graph G' is* connected component *of graph G, if G' is a connected subgraph of G, but there is no connected subgraph G'' of G that G' is proper subgraph of G''.*

Property 1.17 *A graph is connected, or it consists of pairwise disjunct connected components (which together cover the whole graph).*

Definition 1.18 *A graph is* forest*, if its connected components are trees (or it is a tree).*

Property 1.19 *An undirected graph is forest $\iff$ it is acyclic.*

A directed graph G is forest $\iff$ its undirected pair is acyclic, and each connected component of G has generator vertex.

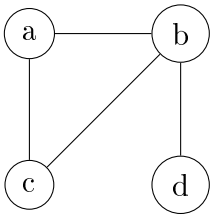
1.2 Introduction to graph representations

When we represent graph $G = (V, E)$, we suppose that $V = \{v_1, \cdots, v_n\}$ where $n \in \mathbb{N}$, i.e. the vertices of the graph can be identified with the sequence numbers or labels $1..n$.

In *graphical* and *textual* representations (see below) the indices of the vertices are often given as the lower case letters of the English alphabet where $a = 1, b = 2, \cdots, z = 26$.

1.3 Graphical representation

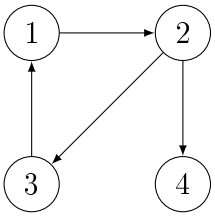
The vertices are represented with small circles. The edges are given as arrows (in case of digraphs) or simple lines (in case of undirected graphs). The labels or indices of the vertices are written into the corresponding circles.



a – b ; c.
b – c ; d.

Figure 1: The same undirected graph in graphical (on the left) and textual (on the right) representations. The vertices are labeled with letters.

Note 1.20 *We can label the vertices of undirected graphs also with index numbers and those of digraphs with letters.*



$1 \to 2$.
$2 \to 3$; 4.
$3 \to 1$.

Figure 2: The same digraph in graphical (on the left) and textual (on the right) representations. The vertices are labeled with indices from 1 to 4.

1.4 Textual representation

In case of undirected graphs "$u - v_{u_1}; \cdots ; v_{u_k}$." means that the neighbors of vertex u are vertices $v_{u_1}, \cdots , v_{u_k}$, i.e. $(u, v_{u_1}), \cdots , (u, v_{u_k})$ are edges of the graph. (See Figure 1.)

In case of digraphs "$u \to v_{u_1}; \cdots ; v_{u_k}$." means that from vertex u come out directed edges $(u, v_{u_1}), \cdots , (u, v_{u_k})$, i.e. vertices $v_{u_1}, \cdots , v_{u_k}$ are the immediate successors or children of vertex u. (See Figure 2.)

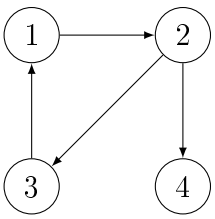
1.5 Adjacency matrix representation

In the *adjacency matrix representation*, graph $G = (V, E)$ ($V = \{v_1, \cdots , v_n\}$) is represented with bit matrix $A/1 : bit[n, n]$ where $n = |V|$ is the number of vertices, $1..n$ are the indices or identifiers of the vertices, **type** *bit* **is** $\{0, 1\}$; and for each indices $i, j \in 1..n$:

$$A[i, j] = 1 \iff (v_i, v_j) \in E$$

$$A[i, j] = 0 \iff (v_i, v_j) \notin E.$$

See for example, Figure 3.



A	1	2	3	4
1	0	1	0	0
2	0	0	1	1
3	1	0	0	0
4	0	0	0	0

Figure 3: The same digraph in graphical (on the left) and adjacency matrix (on the right) representations.

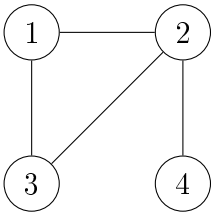
In the main diagonal, there are always zero values and the nonzero elements have value one, because we consider only simple graphs (i.e. graphs without self-loops and parallel edges).

Let us notice that the adjacency matrix of an undirected graph is always symmetrical, because $(v_i, v_j) \in E \iff (v_j, v_i) \in E$. See Figure 4.

This means that in case of undirected graphs, for each pair v_i és v_j of vertices $A[i, j] = A[j, i]$, and $A[i, i] = 0$.

Thus it is enough to represent the triangle under[1] the main diagonal. This lower triangular matrix (without the main diagonal) contains no element in

[1] or above



A	1	2	3	4
1	0	1	1	0
2	1	0	1	1
3	1	1	0	0
4	0	1	0	0

Figure 4: The same undirected graph in graphical (on the left) and adjacency matrix (on the right) representations.

the first row, a single element in the second row, two elements in the third row, and so on, $(n-1)$ elements in the last row. Consequently, instead of

$$n^2 \text{ bits, it consists only of } 1+2+\cdots+(n-1) = n*(n-1)/2 \text{ bits.}$$

Consequently, instead of matrix A we can use array $B : bit[n*(n-1)/2]$ which – using notation $a_{ij} = A[i,j]$ – contains the following sequence.

$$\langle a_{21}, a_{31}, a_{32}, a_{41}, a_{42}, a_{43}, \ldots, a_{n1}, \ldots, a_{n(n-1)} \rangle$$

Thus
$A[i,j] = B[(i-1)*(i-2)/2 + (j-1)]$ if $i > j$ (in the lower triangle)
$A[i,j] = A[j,i]$ if $i < j$ ($A[i,j]$ in the upper triangular matrix)
$A[i,i] = 0$. ($A[i,i]$ is on the main diagonal)

Explanation: If we want to determine the index of item $a_{ij} = A[i,j]$ of the abstract lower triangular matrix in its representation, namely in array B, we have to count the number of items preceding a_{ij} in array B, because array B is indexed from zero. Item a_{ij} of the abstract lower triangular matrix is preceded by the following elements in its representation, namely in array B.

a_{21}
a_{31}, a_{32}
a_{41}, a_{42}, a_{43}
$\vdots$
$a_{(i-1)1}, a_{(i-1)2}, \cdots, a_{(i-1)(i-2)}$
$a_{i1}, a_{i2}, \cdots, a_{i(j-1)}$

These are $(1+2+3+\cdots+(i-2))+(j-1) = (i-1)*(i-2)/2+(j-1)$ elements.

In an adjacency matrix property $(v_i, v_j) \in E$ can be checked in $\Theta(1)$ time. Thus this representation may be a good choice, if our algorithm uses this operation frequently.

On the contrary, enumerating the children (in a digraph) or neighbors (in an undirected graph) of a vertex needs n steps which is typically much more than the actual number of children or neighbors. In case of an algorithm intensively using such enumerations adjacency lists may be a better choice.

1.6 Adjacency list representation

The adjacency list representation is similar to the textual representation.

Graph $G = (V, E)$ ($V = \{v_1, \cdots, v_n\}$) is represented with pointer array $A/1 : Edge*[n]$ where

Edge
$+v : \mathbb{N}$
$+next : Edge*$

In case of an undirected graph, S1L $A[i]$ contains the indices of the neighbors of vertex v_i ($i \in 1..n$). Thus in case of an undirected graph, each edge is represented twice: if vertex v_j is neighbor of vertex v_i, then vertex v_i is also neighbor of vertex v_j.

In case of a digraph, S1L $A[i]$ contains the indices of the children (i.e. immediate successors) of vertex v_i ($i \in 1..n$). Thus in case of an digraph, each edge is represented only once. See an example on Figure 5.

One might use other kind of lists (C2Ls, arrays etc.) instead of S1Ls here. These are also adjacency list representations. However, the representation detailed above is our default

Using adjacency lists, in order to decide, if $(v_i, v_j) \in E$, we have to search for index j on list $A[i]$. Consequently, if this operation is frequent in an algorithm, adjacency matrix representation of the graph may be a better choice.

On the contrary, given a vertex, enumerating its neighbors (in an undirected graph) or children (in a digraph) needs as many steps as the number of its neighbors or children. Regarding the graph algorithms of these notes we find that such enumerations are the most intensively used operations of most of these algorithms. Therefore we usually prefer adjacency lists to matrices.

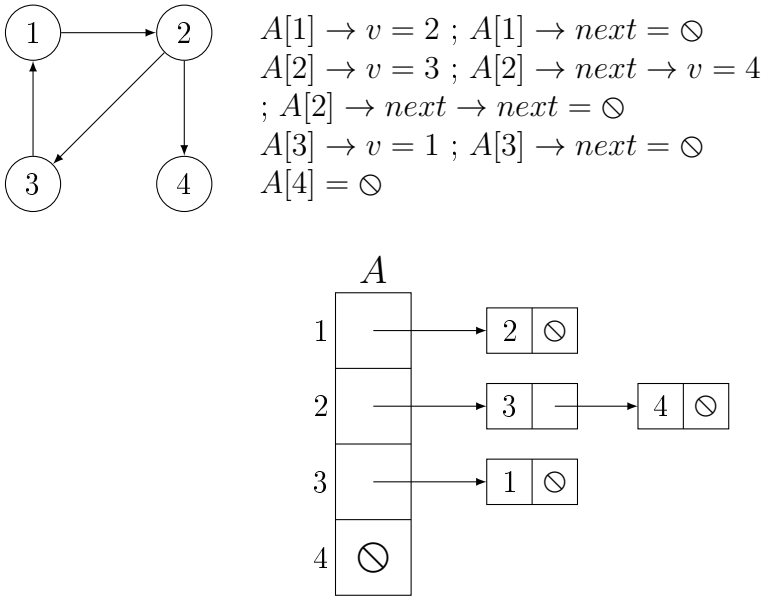


Figure 5: The same digraph is given with graphical representation on the left, and with adjacency list representation on the right.

1.7 Space complexity of representing graphs

Given graph $G = (V, E)$, $n := |V|$, $m := |E|$, i.e. in these lecture notes, we are going to use n and m for the number of vertices and edges of a graph. Clearly $0 \leq m \leq n * (n-1) \leq n^2$, thus $m \in O(n^2)$.

The *sparse graphs* are characterized with $m \in O(n)$, while the *dense graphs* are characterized with $m \in \Theta(n^2)$.

1.7.1 Adjacency matrices

The adjacency matrix, i.e. $A/1 : bit[n, n]$ needs n^2 bits by default.

In case of undirected graphs the abstract matrix above can be represented with array $B : bit[n * (n-1)/2]$ (See subsection 1.5 for the details.)

For this reason the space complexity of the graph is $\Theta(n^2)$ in both cases.

1.7.2 Adjacency lists

In case of adjacency list representation we have pointer array $A/1 : Edge^*[n]$ and m or $2m$ elements of the n adjacency lists together. (See subsection 1.6 for the details: m elements for digraphs and $2m$ elements for undirected

graphs.) For this reason the space complexity of the graph is $\Theta(n + m)$ in both cases.

- In case of *sparse graphs* $m \in O(n)$. Thus the space complexity of the adjacency list representation of these graphs is $\Theta(n+m) = \Theta(n)$. This is asymptotically smaller than $\Theta(n^2)$, which is a space complexity of the adjacency matrix representations of all the graphs. Considering that sparse graphs model most of the different networks, this representation may most often be a good choice.

- In case of *dense graphs* $m \in \Theta(n^2)$. Consequently $\Theta(n + m) = \Theta(n^2)$. Therefore the space complexities of the adjacency list and matrix representations are asymptotically equivalent.

- In case of *complete graphs* the adjacency lists together contain $n * (n - 1)$ elements, and each element consists of many bits (typically one word for the index of the neighbor or child of the actual vertex and another word for the pointer referring to the next element of the list of edges, which means that a single element consists of 128 bits in a 64 bit architecture). As a result, the *actual storage requirements* of the adjacency list representation of a (nearly) complete graph can be significantly greater than *those* of the adjacency matrix representation of the same graph where one element of the matrix can be stored in a single bit.

2 Abstract types *set*, *sequence* and *graph*

These types will be useful in the subsequent abstract algorithms of graphs etc. Given some type $\mathcal{T}$, let

- $\mathcal{T}\{\}$ denote a finite set of items with element type $\mathcal{T}$
- $\{\}$ denote the empty set
- $\mathcal{T}\langle\rangle$ denote a finite sequence of items with type $\mathcal{T}$
- $\langle\rangle$ denote the empty sequence.

We will use the usual operations of sets, plus operation u **from** S where S is a nonempty set. This statement selects a random element of S, assigns its value to variable u, and removes it from set S.

A sequence is indexed from 1. If s is a sequence, s_i denotes its ith element. If $u, v : \mathcal{T}\langle\rangle$, then quad $u + v$ is their concatenation.

Variables of set and variables of sequence types must be declared (like arrays). We suppose that declaration $s : \mathcal{T}\langle\rangle$ initializes s with an empty sequence, and declaration $h : \mathcal{T}\{\}$ initializes h with an empty set.

In order to describe the type of abstract graphs, first we introduce the elementary type $\mathcal{V}$ which will be the abstract type of the vertices of graphs. We suppose that each vertex can labeled by any number named values where each label of the vertex has a value.

These labels are partial functions. The domain of each label is a subset of vertices. They can be created and modified with assignment statements. If a label exists, it is visible and it is effective in the whole program, and it lives while the program runs.

If $v : \mathcal{V}$ and *name* is a label of vertex v, then $name(v)$ denotes the value of label *name* of vertex v. As a result, performing statement $name(v) := x$ assigns value x to label *name* of vertex v. If the label does not exists, $name(v) := x$ creates label *name* of vertex v with value x.

Set $\mathcal{V}$ is typically represented by set $\mathbb{N}$. Similarly, the vertices of a graph with n vertices are typically represented by set $1..n$ if we index the array representing the graph from 1 (or $0..(n-1)$ if we index this array from 0). The labels of the vertices can also be represented with arrays. Thus their visibility, scope and lifetime are bounded in the implementation of the graph algorithm. The solution of the problems arising from this boundaries is part of the implementation process.

Now we describe `type` edge ($\mathcal{E}$) and `type` unweighted abstract graph ($\mathcal{G}$). We emphasize again that for practical reasons we exclude graphs with looping and/or parallel edges, i.e. our *graph* notion means *simple graph*.

$\mathcal{E}$
$+\ u, v : \mathcal{V}$

$\mathcal{G}$
$+\ V : \mathcal{V}\{\}$ // vertices
$+\ E : \mathcal{E}\{\}$ // $E \subseteq V \times V \setminus \{(u,u) : u \in V\}$ // edges

3 Elementary graph algorithms ([3] 22)

The elementary graph algorithms are algorithms on unweighted graphs. In an unweighted graph the *length of a path* is simply the *number of edges* on this path. A *shortest path* between two vertices is also called *optimal path* or *distance* between them. A path may include loop. Clearly, an optimal path never includes loop. In *directed graphs* or *digraphs* edge (u, v) is different from edge (v, u). In *undirected graphs* edge (u, v) is equal to edge (v, u) by definition.

Notation 3.1 *Given vertices u and v of a graph, $u \rightsquigarrow v$ means a path from u to v.*

Definition 3.2 *Given vertices u and v of a graph G, v is available from u means that there is some $u \rightsquigarrow v$ in G.*

In this chapter we consider two basic algorithms on unweighted graphs, i.e. *breadth-first search* (BFS) and *depth-first search* (DFS). The later one is also called *depth-first traversal*. We also discuss two applications of DFS.

3.1 Breadth-first Search (BFS)

We consider BFS on digraphs and also on undirected graphs. Given a graph $G : \mathcal{G}$, we fix a source vertex $s \in G.V$ which can be any vertex of G. We find the shortest paths to each other vertex available form s. If there are more than one shortest paths to a vertex, we compute only one of them.

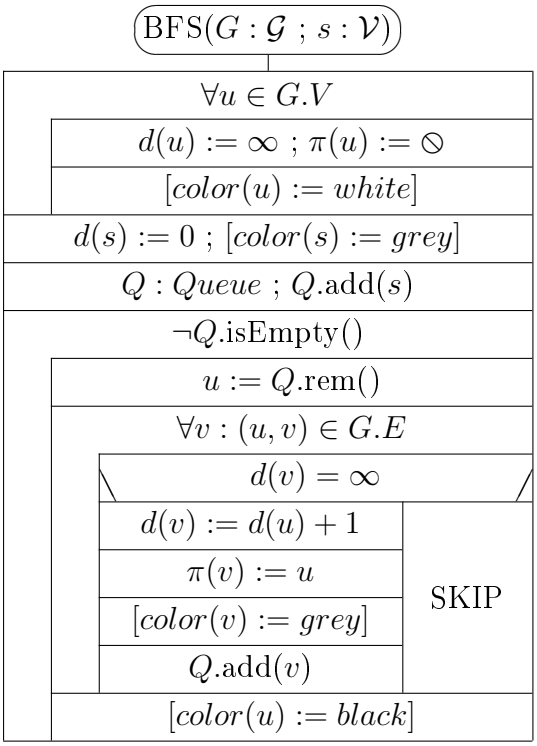
Let u be a vertex of G. The most important labels of u:

- $d(u)$ = the length of $s \rightsquigarrow u$ found. $d(u) = \infty$ means that (still) we have not found any $s \rightsquigarrow u$. Clearly, $d(u) = 0$, **iff** $u = s$.
- $pi(u)$ = the parent of vertex u on $s \rightsquigarrow u$ found. $pi(u) = \oslash$ means that (still) we have not found any $s \rightsquigarrow u$ or $u = s$.

If vertex u is unavailable form s, $d(u) = \infty$ and $\pi(u) = \oslash$ remains even when the algorithm terminates. Otherwise BFS calculates a shortest (i.e. optimal) $s \rightsquigarrow u$. $\pi(s) = \oslash$ also remains true because the optimal $s \rightsquigarrow s$ consists of only s, it contains no edge, i.e. s has no parent on this path.

We will also use a label called *color*. Its value does not influence the run of the program. Consequently the statements referring to it can be safely omitted form BFS. (Therefore they are put between square brackets in the structogram of BFS.) They are useful only for illustration:

- The *white* vertices have not yet been found by the graph search (or traversal).
- The *grey* vertices have already been found but not processed yet.
- The *black* vertices have already been processed. The algorithm has nothing to do with them.



Performing the first loop, $d(u) = \infty$ becomes true for each vertex of the graph, which means that no vertex have been found by the algorithm. As a result, $\pi(u) = \oslash$ for each vertex[2].

We know that $d(s) = 0$ is the length of the shortest $s \rightsquigarrow s$.[3] The vertices found but not yet processed are put into the queue, i.e. into Q. For this reason, s is already put into Q. Processing vertex u means that we remove u form Q, plus *expand* u, i.e. enumerate and consider the *children* or *neighbors* of u (*children* in digraphs and *neighbors* is undirected graphs). This means that all the time we process the first element of Q.

The second, main loop runs while there is any vertex which we have found but not yet processed, i.e. Q is not empty. The main loop first removes vertex

[2] [and also $color(u) = white$]

[3] [Performing $d(s) := 0$ we have started the processing of s. This has the effect of $color(s) := grey$.]

$u = s$ form Q. Then for each child/neighbor of s its d value will be 1 and its π value will be s because the optimal path to it consist of the edge (s, u). The children/neighbors of s are also put into Q so that they can be processed later.[4]

Now Q contains the vertices available from s in a single step. The main loop removes them from Q one by one. It finds the vertices available from s in two steps, i.e. minimum through two edges, because these vertices are the **newly found** *children* or *neighbors* of those available in a single step. Undoubtedly, the **newly found** vertices are those with $d(v) = \infty$. The main loop also assigns the $d(v) = 2$ and $\pi(v) = u$ values according to the parents of these vertices[5], and they are added to the end of Q[6]. Provided that for some vertex v, $d(v) \neq \infty$ when we process edge (u, v), this vertex is unquestionably known form earlier[7], consequently $d(v) \in \{0, 1, 2\}$, which means that the newly found path to v is not shorter than the old path found it. For this reason, in this case edge (u, v) is omitted by BFS. (See the conditional statement in the abstract code of BFS above.)

By the time BFS has processed all the vertices at distance 1 from s, i.e. it has removed them from Q and it has *expanded* them (i.e. processed the edges going out of them), by this time Q contains the vertices at distance 2 form s.

While BFS is processing the vertices at distance 2, i.e. is it removes them from Q and *expands* them, the vertices at distance 3 are newly found and put at the end of Q with the appropriate d and π values. As a result of this, by the time BFS has processed all the vertices at distance 2, Q contains all the vertices at distance 3, and so on.

Generally speaking, when Q consists of the vertices at distance k from s, BFS starts to process them. While processing them, it newly finds the vertices at distance $k+1$, assigns the appropriate values to their labels and puts them at the end of Q. By the time all the vertices at distance k have been processed, Q consists of the vertices at distance $k + 1$, and so on.

We say that the vertices at distance k form s are at level k of the graph. As a result, BFS traverses the graph level by level. It starts with level 0. Next it goes to level 1. Then it follows with level 2, and so on. Each level is completely processed before BFS goes on to the next level: While BFS is processing a level, it newly finds the vertices at the next level, and puts them

[4] [They also receive *grey* color, and finally s is colored black, because this vertex has been finished.]

[5] [they also receive *grey* color]

[6] [then their parent is colored *black* because it has been finished]

[7] [already it is not *white*, but *grey* or *black*]

into Q. Then it finds the vertices of the next level in Q. For this reason, it is called *breadth-first search* (or breadth-first traversal).

Because the graph is finite, finally there will be no vertex at the next level, i.e. at distance $k+1$. Q becomes empty, and BFS stops. The vertices available from s have been found at some level. For each of them, its d value contains its distance from s, and its π value refers to its parent on the optimal path found to it (with the exception that $\pi(s) = \oslash$ remains true, because s has no parent). Therefore all the other vertices are unavailable from s: for each of them $d(v) = \infty$ and $\pi(v) = \oslash$ remain true, because it is not found by BFS.[8]

Exercise 3.3 *In the algorithm of BFS, condition "$d(v) = \infty$" can be replaced by another condition equivalent to it. Which is this condition? Explain your decision.*

3.1.1 Breadth-first Tree

Let us suppose that we have run procedure call BFS(G, s). Let $v \neq s$ be a vertex available form s. Then BFS finds an optimal path $s \rightsquigarrow v$, and $\pi(v)$ refers to the parent of v on this path. Unquestionably many vertices may have the same parent on the optimal paths found to them, but the parent of each vertex similar to v has a single parent.

Consequently – as the result of BFS(G, s) – the π values of the vertices available from s define a general tree. Its root is s, and accordingly $\pi(s) = \oslash$. This tree is called *breadth-first tree* or *shortest-paths tree*. For each vertex v available from s, this tree contains a shortest path $s \rightsquigarrow v$ where this optimal path has been computed by procedure call BFS(G, s).

Obviously this reversed representation of the *shortest-paths tree* is space efficient because each vertex has at most one parent but may have many children in the tree.

Exercise 3.4 *Let us suppose that we have run procedure call BFS(G, s), and vertex v is available from s. Write recursive procedure printShortestPathTo(v) which prints the shortest path $s \rightsquigarrow v$ calculated by BFS(G, s).*

Notice that parameter s is not needed, if v is available from s. This algorithm should not build any new data structure.

$MT(d) \in \Theta(d)$ where $d = d(v)$.

[8] [Thus the vertices available from s become black, while the others remain white.]

Exercise 3.5 *Let us suppose that we have run procedure call BFS(G, s), and vertex v is available from s. Write nonrecursive procedure printShortestPathTo(v) which prints the shortest path $s \rightsquigarrow v$ calculated by BFS(G, s).*

Notice that parameter s is not needed, if v is available from s. Explain your data structure needed for avoiding recursive code.

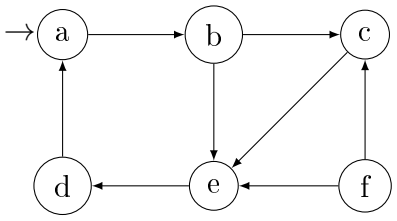
$MT(d) \in \Theta(d)$ where $d = d(v)$.

Exercise 3.6 *Let us suppose that we have run procedure call BFS(G, s). Write procedure printShortestPathTo(s, v) which*

- *prints the shortest path $s \rightsquigarrow v$ calculated by BFS(G, s), provided that vertex v is available from s.*
- *prints text "There is no path from s to v" with the appropriate substitutions for s and v, otherwise.*

$MT(d) \in \Theta(d)$, where $d = d(v)$, if v is available from s; and $d = 1$, otherwise.

3.1.2 Illustrations of BFS



a → b.
b → c ; e.
c → e.
d → a.
e → d.
f → c ; e.

Figure 6: The same digraph in graphical (on the left) and textual (on the right) representation (s = a).

We illustrate BFS on the graph of Figure 6. We use the table below where "a" is the source vertex. Obviously the new vertices always go to the end of queue Q.

We have a convention that in *nondeterministic* cases we prefer the vertex with lower index. This convention will be applied at the illustration of each graph algorithm. (This is the weakest "rule" but you should follow it at tests and exams.) For example, in the following illustration it is used when the children of vertex "b" are put into the queue in order "c,e".

changes of d						ex-	Q :	changes of π					
a	b	c	d	e	f	panded	Queue	a	b	c	d	e	f
0	∞	∞	∞	∞	∞	vertex : d	$\langle$ a $\rangle$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$
	1					a:0	$\langle$ b $\rangle$		a				
		2		2		b:1	$\langle$ c, e $\rangle$			b		b	
						c:2	$\langle$ e $\rangle$						
			3			e:2	$\langle$ d $\rangle$				e		
						d:3	$\langle$ $\rangle$						
0	1	2	3	2	∞	final d and π values		$\oslash$	a	b	e	b	$\oslash$

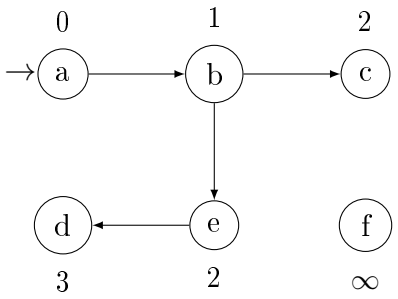
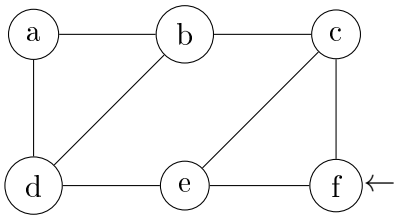


Figure 7: The breadth-first tree of BFS on the graph familiar from Figure 6 provided that s = a.

Now we illustrate BFS on the graph of Figure 8 where "f" is the source vertex.



a – b ; d.
b – c ; d.
c – e ; f.
d – e.
e – f.

Figure 8: An undirected graph in graphical (on the left) and in textual (on the right) representation (s = f).

changes of d						ex-	Q :	changes of π					
a	b	c	d	e	f	panded	Queue	a	b	c	d	e	f
∞	∞	∞	∞	∞	0	vertex : d	$\langle$ f $\rangle$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$
		1		1		f:0	$\langle$ c, e $\rangle$			f		f	
	2					c:1	$\langle$ e, b $\rangle$		c				
			2			e:1	$\langle$ b, d $\rangle$				e		
3						b:2	$\langle$ d, a $\rangle$	b					
						d:2	$\langle$ a $\rangle$						
						a:3	$\langle$ $\rangle$						
3	2	1	2	1	0	final d and π values		b	c	f	e	f	$\oslash$

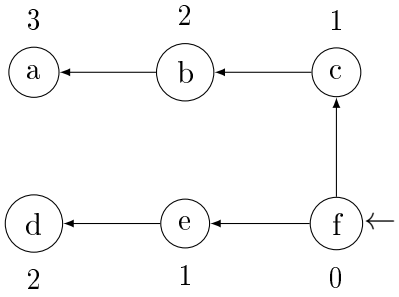


Figure 9: The breadth-first tree of BFS on the graph familiar from Figure 8 provided that $s = \text{f}$.

3.1.3 Efficiency of BFS

Remember that we use the following traditional notations at graph algorithms: $n = |G.V|$ and $m = |G.E|$.

The first, the initializing loop of BFS(G, s) iterates n times.
The second, the main loop of it iterates as much as the number of vertices available from s (counting also s itself): this is maximum n, minimum 1.

As a result of this, the number of iterations of the inner loop is maximum m or $2m$ on directed/undirected graphs (when all the vertices are available form s). And it is minimum zero (provided that no edge goes out form s).

For this reason, $MT(n, m) \in \Theta(n + m)$, and $mT(n, m) \in \Theta(n)$.

3.1.4 The implementations of BFS in case of adjacency list and adjacency matrix representations

We suppose that in graph $G = (V, E)$, $V = \{v_1, \cdots, v_n\}$ where $n \in \mathbb{N}$, i.e. the vertices of the graph can be identified by the indices $1..n$. Labels d and

π of the vertices are represented by arrays $d/1, \pi/1 : \mathbb{N}[n]$ where $d(v_i)$ is represented by $d[i]$ and $\pi(v_i)$ is represented by $\pi[i]$. The representation of the *color* labels of the vertices is superfluous. The representation of $\oslash$ can be number 0, for example $\pi[s] = 0$ means $\pi(v_s) = \oslash$. Undoubtedly the length of a shortest path between two vertices is maximum $n-1$. And this is also the maximum of the finite d-values in BFS. Thus we can use number n instead of ∞.

Exercise 3.7 *Let us suppose that we represent the abstract graph of BFS with adjacency lists. (See section 1.6.)*

Write procedure BFS($A/1 : Edge*[n]$; $s : 1..n$; $d/1, \pi/1 : \mathbb{N}[n]$) implementing the appropriate abstract algorithm in this case. Make sure that the the worst-case and the best-case operational complexities of the implementation of BFS remain $MT(n,m) \in \Theta(n+m)$, and $mT(n,m) \in \Theta(n)$, respectively.

Exercise 3.8 *Let us suppose that we represent the abstract graph of BFS with adjacency matrix. (See section 1.5.)*

Write procedure BFS($A/1 : bit[n,n]$; $s : 1..n$; $d/1, \pi/1 : \mathbb{N}[n]$) implementing the appropriate abstract algorithm in this case.

Can we retain the worst-case and the best-case operational complexities of the abstract BFS? If not, how do they change?

3.2 Depth-first Search (DFS)

DFS is also called *depth-first traversal*, because it touches all the vertices and edges of the graph.

In these lecture notes we consider DFS on digraphs only. Unlike in BFS, in DFS the colors of the vertices are essential. Unlike BFS, DFS goes on in one direction in the graph while it finds undiscovered, i.e. white vertices. When DFS discovers a vertex, it becomes grey.

When DFS does not find any white vertex as a child of the actual vertex, it colors the actual vertex black, and backtracks to its *parent*, i.e. to the vertex it was discovered from. Then this parent becomes the actual vertex again. And BFS tries to go through another, still unprocessed edge to another white vertex, and so on.

Unquestionably DFS is highly non-deterministic. It may select any unprocessed edge going out from the actual vertex. (While illustrating DFS, we will resolve this non-determinism: we prefer the edge going to the vertex with the lowest index.) DFS can backtrack recursively, if needed.

The classical form of DFS does not have any source vertex. The *depth-first traversal* of the graph consists of *depth-first visits* (DFvisits). A DFvisit

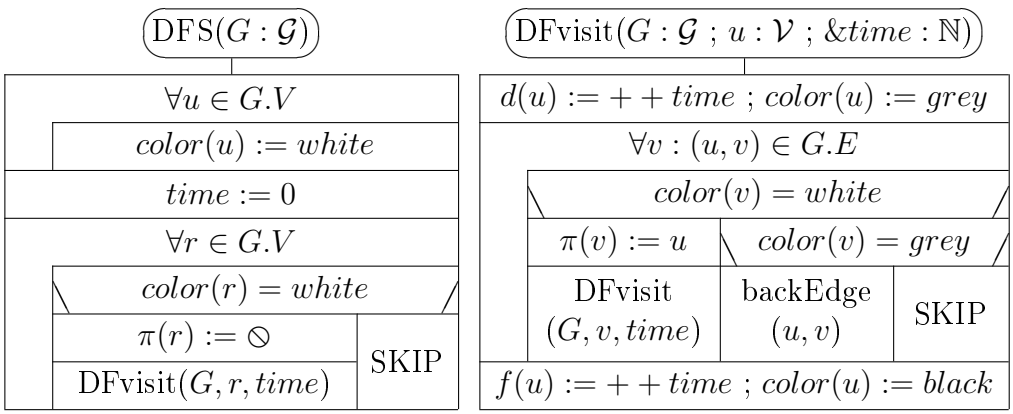
may start from any white vertex of the graph. (While illustrating DFS, we will resolve this non-determinism: we prefer the white vertex with the lowest index.) The starting point of a DFvisit is its root, because each DFvisit builds up a *depth-first tree* with this starting point as root. In a depth-first tree, the *parent* of a non-root vertex is the vertex it was discovered form.

A DFvisit ends when we color its root black, and we cannot backtrack from it, because it does not have any parent. Then we try to start another DFvisit form a white vertex. When no white vertex remains, DFS stops. It is certainly the case that a vertex may be grey only during a DFvisit. Before a DFvisit all the vertices are white or black. As a result, at the end of the program all of them are black.

DFS also has variable $time : \mathbb{N}$. It starts from zero and it is increased when a vertex is *discovered* or *finished*.

For each vertex of v of the graph, DFS assigns value to the following labels of v.

- $color(v) \in \{white, grey, black\}$ where $color(v) = white$ means that v is still undiscovered, i.e. no DFvisit has touched it; $color(v) = grey$ means that v has been discovered, but it is still unfinished, i.e. DFS still has not tried to backtrack from it; and $color(v) = black$ means that v has been finished.
- $d(v) \in \{1, 2, 3, ...\}$ is the *discovery time* of v.
- $f(v) \in \{2, 3, 4, ...\}$ is the *finishing time* of v.
- $\pi(v) : \mathcal{V}$ is the *parent vertex* of v. If v does not have parent (i.e. it is the root of a DFvisit), then $\pi(v) = \oslash$.



3.2.1 Depth-first forest

Each DFvisit (started from DFS) computes a *depth-first tree*. The *depth-first forest* consists of these depth-first trees.

$r \in G.V$ is the root of a depth-first tree $\iff \pi(r) = \oslash$
$(u, v) \in G.E$ is the edge of a depth-first tree $\iff u = \pi(v)$

3.2.2 Classification of edges

Definition 3.9 *The classification of the edges of the graph:*

(u, v) *is* tree edge $\iff$ (u, v) *is the edge of some* depth-first tree.
(We traverse the graph through the tree edges.*)*

(u, v) *is* back edge $\iff$ v *is ancestor of* u *in a* depth-first tree.

(u, v) *is* forward edge $\iff$ (u, v) *is not tree edge, but* v *is a descendant of* u *in a* depth-first tree.

(u, v) *is* cross edge $\iff$ u *and* v *are vertices on different branches of the same* depth-first tree*, or they are in two different* depth-first trees.

Theorem 3.10 *Recognizing the edges of a graph:*
When we process edge (u, v) *during a DFvisit, this edge can be classified according to the following criteria.*

(u, v) *is* tree edge $\iff$ *vertex* v *is still white.*

(u, v) *is* back edge $\iff$ *vertex* v *is just grey.*

(u, v) *is* forward edge $\iff$ *vertex* v *is already black* $\wedge\ d(u) < d(v)$.

(u, v) *is* cross edge $\iff$ *vertex* v *is already black* $\wedge\ d(u) > d(v)$.

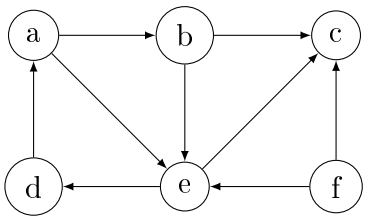
Exercise 3.11 *We are just processing edge* (u, v) *during DFS. Why* $d(u) = d(v)$ *cannot happen? In which case could it happen? In this case, how should we modify* **Definition 3.9** *so that we do not have to modify* **Theorem 3.10***?*

3.2.3 Illustration of DFS

We illustrate DFS on Figure 10. DFS is non-deterministic in two aspects. (1) Which white certex is selected as root of a DFvisit. (2) Which unprocessed edge going out from the actual vertex is selected to be processed: these edges can be ordered according to the vertices they point to.

At this course, when we illustrate a graph algorithm, its non-determinism is resolved by considering the possible vertices in alphabetical order, i.e. the vertex with lowest index is preferred. Note that there could be another convention or we could select randomly from the set of possible vertices. However, this alphabetical convention must be followed at our tests and exams.

On Figure 10 we can see a digraph.



a → b ; e.
b → c ; e.
d → a.
e → c ; d.
f → c ; e.

Figure 10: The same digraph in graphical (on the left) textual (on the right) representation.

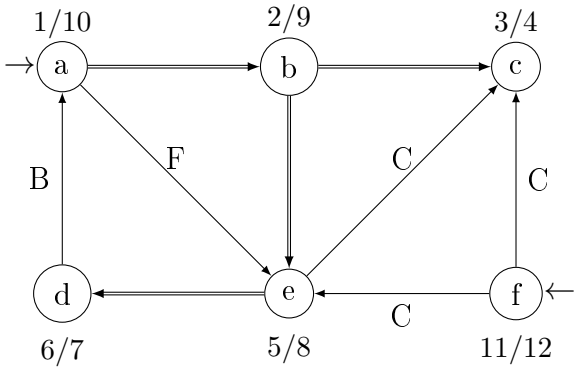


Figure 11: DFS of graph on Figure 10.

On Figure 11 we can see the result of DFS on the same digraph. Labels of the form d/f of the vertices display the discovery and finishing times of them. Small arrows pointing to vertices "a" and "f" show the roots of the DFvisits. Double arrows identify the edges of the depth-first trees. The classification of the other edges is made clear by the following labels of them. "B" means *back edge*, "F" means *forward edge*, and "C" means *crossing edge* .

3.2.4 The running time of DFS

We use the same notations as before: $n = |G.V|$ and $m = |G.E|$. Procedure DFS is called only once. Both loops of DFS iterares n times. We have $2n+1$ steps without DFvisits.

Procedure DFvisit is also called n times, once for each vertex: when we achieve it first, and it is still white, recursive procedure DFvisit is called for it. The loop of DFvisit iterates as much as the number of edges going out from vertex u. This loop processes the edges of the graph, and each edge is processed by one iteration of this loop. As a result of this, considering all the calls of DFvisit together, the loop of DFvisit iterates m times. We suppose here that a call to procedure backEdge just labels a back edge[9], thus its operational complexity is $\Theta(1)$, and its running time does not modify the asymptotic order of the loop iteration invoking it. Altogether we have $n+m$ steps in DFvisits.

We have counted $3n + m + 1$ steps.

Consequently $MT(n), mT(n) \in \Theta(n + m)$ for DFS.

3.2.5 Checking the DAG property

Definition 3.12 *Digraph G is DAG (Directed Acyclic Graph),* **iff** *it does not contain directed loop.*

DAGs are an important class of graph. Many useful algorithms is defined on them. For this reason, checking their input can be crucial. (For example, see algorithms *topological sort* and *DAG shortest paths* later.

It is certainly the case that when DFS finds back edge (u, v), then – according to the definition of *back edge* – it also finds a directed loop in the graph, because v is an ancestor of u in the depth-first tree which the actual DFvisit is building. Therefore the path consisting of the tree edges and leading from v to u concatenated with edge (u, v) form a directed loop.

It can be proved that if digraph G in not DAG, i.e. it contains directed loop, then DFS finds back edge, and thus it finds a directed loop containing this back edge [3]. [However, DFS often does not find all the directed loops, because a back edge can be part of many directed loops: provided that it processes a back edge, it will never process it again.]

Exercise 3.13 *Draw a digraph with three vertices and four edges which contains too simple directed loops, and maybe DFS finds both of them, maybe*

[9]Notice that we omitted the explicit labeling of the non-back edges in our structogram, although each tree edge (u, v) is booked by the assignment statement $\pi(v) := u$. We are going to see that the forward and cross edges do not have significance in our applications.

not, depending on the resolution of its inherent non-determinism. Illustrate both cases.

Summarizing all these we receive the following theorem.

Theorem 3.14 *Digraph G is DAG $\iff$ DFS does not find back edge.*
If DFS finds back edge (u, v), then sequence $\langle v, u, \pi(u), \pi(\pi(u)), \ldots, v\rangle$ of vertices read in backward direction forms a simple directed loop.

Based on this theorem, the run of procedure call backEdge(u, v) of structogram DFvisit() (see section 3.2) is able to print the directed loop found. In this case its maximal running time is clearly $\Theta(n)$. (And in some cases the printing of all the directed loops found can increase even the asymptotic running time of DFS.)

Exercise 3.15 *Write the structogram of procedure backEdge(u, v) provided that it must print all the loops found in an easy to read manner. Make sure that $MT(n) \in \Theta(n)$ is satisfied.*

Exercise 3.16 *Define an infinite sequence of digraphs where $m \in O(n)$, consequently $MT_{DFS}(n, m) \in \Theta(n)$, but if procedure backEdge(u, v) must print all the directed loops found, then even the minimal running time of DFS is $\Omega(n^2)$. Explain why these properties are satisfied.*

Exercise 3.17 *Modify DFS so that it can search loops on undirected graphs. How can we recognize back edges? [Undoubtedly, in an undirected graph, if (u, v) is tree edge, then (v, u) is not back edge because $(v, u) = (u, v)$.] What about forward and cross edges?*

3.2.6 Topological sort

Definition 3.18 *A topological order of digraph G is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in this ordering.*

A graph may have more than one topological order. See for example Figure 12.

Theorem 3.19 *A digraph has topological order $\iff$ it is DAG (i.e. it does not contain directed loop.)*

Proof.

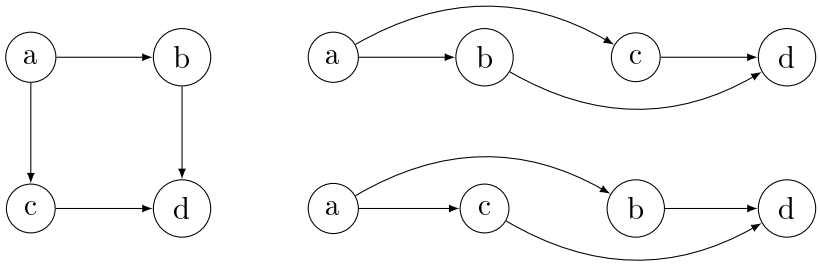


Figure 12: A DAG drawn in three different ways: it is drawn in a traditional way on the left. On the right the vertices are in two topological orders.

$\Rightarrow$ If there is a directed loop in the graph, let it be $\langle u_1, u_2, \ldots, u_k, u_1 \rangle$. As a result, u_1 is followed directly or indirectly by u_2, then somewhat later by u_3, and so on. Even later u_k comes, and then u_1 again. This is contradiction. Unquestionably, there is no topological order of this graph.

$\Leftarrow$ If there is no directed loop in the graph, undeniably it has some vertex without parent. If we delete such a vertex from the graph (together with its edges), then the no cycle is generated in the graph remaining. Again we have some vertex without parent, we can delete it, and so on. In order the deleted vertices form a topological order.

□

A *topological sort* is a graph algorithm generating a topological order of the graph. This process of generating a topological order is also called *topological sort*.

We can sort a graph topologically with DFS, for example.

Topological sort of a DAG with DFS:

1. We create an empty stack.
2. Perform DFS on the digraph: When a vertex is finished we put it on the top of the stack.
3. If DFS finds a back-edge, then the graph is not a DAG, there is no topological ordering, and the content of the stack is undefined.
4. Otherwise, finally the content of the stack in top-down order is the topological order of the DAG.

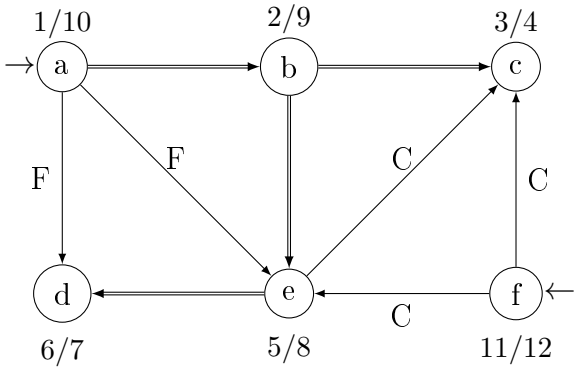


Figure 13: The vertices of a DAG sorted strictly decreasingly according to the finishing times of DFS form a topological order of the DAG. And this order can be received efficiently with pushing each vertex at the beginning of the sequence when it is finished by DFS. As a result, $\langle f, a, b, e, d, c\rangle$ is a topological order of the DAG above. Let us notice, if we resolve the non-determinism of DFS differently, the topological order received may also be different.

An illustration of topological sort is found on Figure 13.

A natural application of topological sort is a solution of a *single-machine scheduling* problem: The vertices of the DAG are jobs, and an edge (u, v) of the graph means that job u must be performed before job v. The jobs must be sorted according to these constraints represented by the edges.

Exercise 3.20 *Write the structograms of (1) DFS and (2) topological sort in cases of (A) adjacency list and (B) adjacency matrix representations of the graph. What can you say about the efficiencies of the four implementations?*

Exercise 3.21 *Notice that the second half of the proof of Theorem 3.19 can be considered an algorithm. Write its structogram, which is independent of DFS. How can you avoid the destroying of the input graph using just $O(n)$ working memory? What about efficiency? How can you handle cyclic digraphs in this algorithm?*

Algorithms and Data Structures II.
Lecture Notes:
Algorithms on weighted graphs

Ásványi Tibor – asvanyi@inf.elte.hu

November 6, 2022

Contents

References

[1] Ásványi, T, Algorithms and Data Structures I. Lecture Notes
http://aszt.inf.elte.hu/∼asvanyi/ds/AlgDs1/AlgDs1LectureNotes.pdf

[2] Ásványi, T, Algorithms and Data Structures II. Lecture Notes: Elementary graph algorithms
http://aszt.inf.elte.hu/∼asvanyi/ds/AlgDs1/AlgDS2graphs1.pdf

[3] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.,
Introduction to Algorithms (Third Edititon), *The MIT Press*, 2009.

[4] Cormen, Thomas H., Algorithms Unlocked, *The MIT Press*, 2013.

[5] Narashima Karumanchi,
Data Structures and Algorithms Made Easy, *CareerMonk Publication*, 2016.

[6] Neapolitan, Richard E., Foundations of algorithms (Fifth edition), *Jones & Bartlett Learning*, 2015. ISBN 978-1-284-04919-0 (pbk.)

[7] Shaffer, Clifford A.,
A Practical Introduction to Data Structures and Algorithm Analysis, Edition 3.1 (C++ Version), 2011
(See http://aszt.inf.elte.hu/∼asvanyi/ds/C++3e20110103.pdf)

[8] Tarjan, Robert Endre, Data Structures and Network Algorithms, *CBMS-NSF Regional Conference Series in Applied Mathematics*, 1987.

[9] Weiss, Mark Allen, Data Structures and Algorithm Analysis in C++ (Fourth Edition),
Pearson, 2014.

[10] Ásványi Tibor, Detecting negative cycles with Tarjan's breadth-first scanning algorithm (2016)
http://ceur-ws.org/Vol-2046/asvanyi.pdf

1 Weighted graphs and their representations ([3] 22)

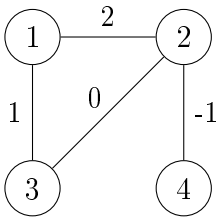
Definition 1.1 *A* weighted graph *is a graph* $G = (V, E)$ *with the weight function* $w : E \to \mathbb{R}$ *where* V *is the finite set of vertices, and* $E \subseteq V \times V \setminus \{(u, u) : u \in V\}$ *is the set of edges.*

Provided that $(u, v) \in E$, $w(u, v)$ *is its* weight *or* length *or* cost *(*weight, length *and* cost *are synonyms).*

Definition 1.2 *Given a weighted graph, the* weight *or* length *or* cost *of a path is the sum of the weights of the edges along the path.*

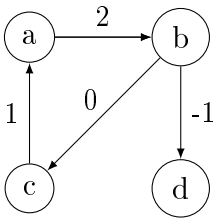
1.1 Graphical representation

The edges are labeled with their weights.



1 – 2, 2 ; 3, 1.
2 – 3, 0 ; 4, -1.

Figure 1: A weighted, undirected graph in graphical representation (on the left) and in textual representation.



a → b, 2.
b → c, 0 ; d, -1.
c → a, 1.

Figure 2: A weighted digraph in graphical representation (on the left) and in textual representation (on the right).

1.2 Textual representation

In case of undirected graphs "$u - v_{u_1}, w_{u_1}; \cdots ; v_{u_k}, w_{u_k}$." means that $(u, v_{u_1}), \cdots , (u, v_{u_k})$ are edges of the graph, in order with weights $w(u, v_{u_1}) = w_{u_1}, \cdots , w(u, v_{u_k}) = w_{u_k}$. (See Figure 1.)

In case of digraphs "$u \to v_{u_1}, w_{u_1}; \cdots ; v_{u_k}, w_{u_k}$." means that from vertex u come out directed edges $(u, v_{u_1}), \cdots , (u, v_{u_k})$, in order with weights $w(u, v_{u_1}) = w_{u_1}, \cdots , w(u, v_{u_k}) = w_{u_k}$. (See Figure 2.)

1.3 Adjacency matrix representation

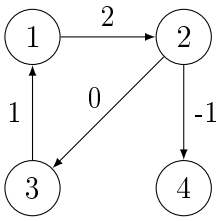
In the *adjacency matrix representation*, graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$ ($V = \{v_1, \cdots , v_n\}$) is represented with matrix $A/1 : \mathbb{R}_\infty[n, n]$ where $n = |V|$ is the number of vertices, $1..n$ are their sequence numbers, and for sequence numbers $i, j \in 1..n$,

$$A[i, j] = w(v_i, v_j) \iff (v_i, v_j) \in E$$

$$A[i, i] = 0$$

$$A[i, j] = \infty \iff (v_i, v_j) \notin E \wedge i \neq j$$

For example, on Figure 3, the digraph is represented with the adjacency matrix next to it.

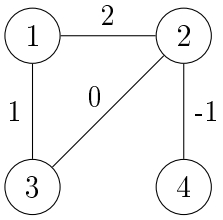


A	1	2	3	4
1	0	2	∞	∞
2	∞	0	0	-1
3	1	∞	0	∞
4	∞	∞	∞	0

Figure 3: A weighted digraph in graphical representation (on the left) and in adjacency matrix representation (on the right).

There are always zeros in the main diagonal, because we study only simple graphs (with no looping edge), and a vertex is available from itself on a path of zero length. See Figure 4.

Notice that the adjacency matrix is always symmetrical, if the graph is undirected, because in case $(v_i, v_j) \in E$, $(v_j, v_i) = (v_i, v_j) \in E$.



A	1	2	3	4
1	0	2	1	∞
2	2	0	0	-1
3	1	0	0	∞
4	∞	-1	∞	0

Figure 4: A weighted undirected graph in graphical representation (on the left) and in adjacency matrix representation (on the right).

1.4 Adjacency list representation

The adjacency list representation is similar to the textual one. Graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$ ($V = \{v_1, \cdots, v_n\}$) is represented with pointer array $A : Edge*[n]$ where type $Edge$ is the following one.

$Edge$
$+v : \mathbb{N}$
$+w : \mathbb{R}$
$+next : Edge*$

The roles of attributes v and $next$ are the same as in case unweighted graphs, but w is the weight (i.e. length) of the appropriate edge. In case of undirected graphs, each edge is represented twice, but in case of digraphs, each edge is represented only once. (See Figure 5.)

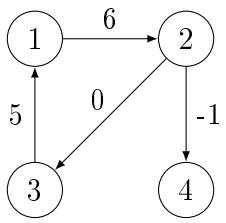
1.5 Space complexity of representing graphs

1.5.1 Adjacency matrices

Their space requirements can be calculated similarly as in the unweighted case. Provided that a float is represented in one word, the storage requirement of adjacency matrix representation is n^2 words in the general case. In case of undirected graphs, storing just the lower triangle matrix, we need $n*(n-1)/2$ words. $n*(n-1)/2 \in \Theta(n^2)$, consequently the space complexity of this representation is $\Theta(n^2)$ in both cases.

1.5.2 Adjacency lists

Each "Edge" contains an extra data member w compared to the unweighted case. Clearly, this fact does not influence the asymptotic storage requirements.



$A[1] \to v = 2$; $A[1] \to w = 6$; $A[1] \to next = \oslash$
$A[2] \to v = 3$; $A[2] \to w = 0$; $A[2] \to next \to v = 4$
; $A[2] \to next \to w = -1$; $A[2] \to next \to next = \oslash$
$A[3] \to v = 1$; $A[3] \to w = 5$; $A[3] \to next = \oslash$
$A[4] = \oslash$

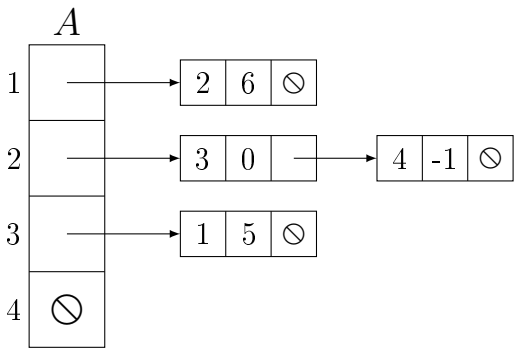
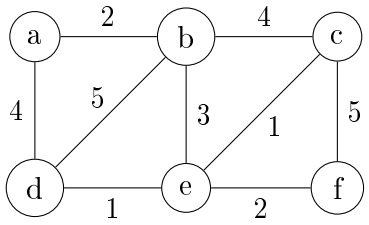


Figure 5: A weighted graph (on the left) in adjacency list representation (on the rignt).

1.6 The abstract class of weighted graphs

$\mathcal{G}_w$
$+\ V : \mathcal{V}\{\}$
$+\ E : \mathcal{E}\{\}\ //\ E \subseteq V \times V \setminus \{(u,u) : u \in V\}$
$+\ w : E \to \mathbb{R}\ //$ weights of edges

2 Minimum spanning trees (MSTs) ([3] 23)



a – b, 2 ; d, 4.
b – c, 4 ; d, 5 ; e, 3.
c – e, 1 ; f, 5.
d – e, 1.
e – f, 2.

Figure 6: Connected weighted undirected graph. The vertices can be cities, the edges are possible routes with the costs of building them. We want to ensure that we can go form each city to each other city while minimizing the costs of building the network.

In order to solve problems similar the one illustrated on Figure 6, we define the notion of *MST = Minimum Spanning Tree*. In this chapter we discuss algorithms computing the MSTs of connected weighted undirected graphs. (The weights of the edges may be negative.)

Definition 2.1 *Given undirected graph $G = (V, E)$, its* spanning forest *is graph $T = (V, F)$, if $F \subseteq E$, and T is (undirected) forest (i.e. T is an undirected graph, consisting of undirected tree components where the trees are pairwise disjunctive, and they cover V together).*

Definition 2.2 *Given undirected connected graph $G = (V, E)$, graph $T = (V, F)$ is its* spanning tree*, if $F \subseteq E$, and T is (undirected) tree.*

Definition 2.3 *Provided that $G = (V, E)$ is a weighted graph (tree, forest, etc.) with weight function $w : E \to \mathbb{R}$, then the weight of G is the sum of the weights of its edges:*

$$w(G) = \sum_{e \in E} w(e)$$

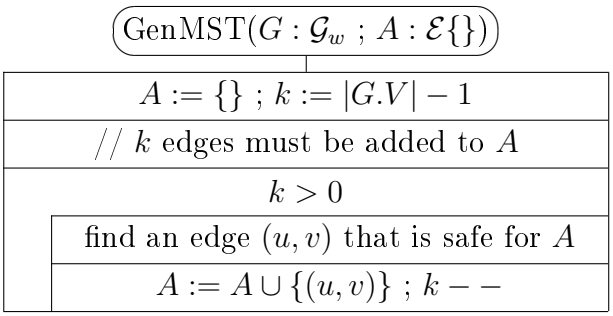
Definition 2.4 *Given undirected connected weighted graph G, T is the* minimum spanning tree*, i.e. MST of G, if*

- *T is spanning tree of G, and*
- *for each other spanning tree T' of G, $w(T) \leq w(T')$.*

Definition 2.5
We say that an A set of edges is a subset of the graph $G = (V, E)$, **iff** $A \subseteq E$.

2.1 A general method

The general algorithm below starts with the empty set $A = \{\}$. In each iteration, it adds a new edge to this set of edges. It has an invariant property: *Set A is a subset of some MST of G.* Consequently, A becomes the set of edges of an MST of G, when the size of A becomes $|G.V|-1$ (because the number of edges of a tree is always its number of vertices minus one).



Definition 2.6 *Let us suppose that $G = (V, E)$ is an undirected connected weighted graph, and the edge set A is a subset of some MST of G.*

Edge $(u, v) \in E$ is safe for A, **iff** $(u, v) \notin A$, *and $A \cup \{(u, v)\}$ is still a subset of some MST of G. (The two MSTs may be different.)*

Consequence 2.7 *Let us suppose that $G = (V, E)$ is an undirected connected weighted graph. Provided that we have the initially empty set A of edges, and in each step we add an edge to A which is safe for A, then after $|V|-1$ steps we receive an MST of G.*

Now we have the following question. *How to find an edge which is safe for A?* In order to answer it, we present the following notions and theorems.

Definition 2.8 *$(S, V \setminus S)$ is a* cut *of the graph $G = (V, E)$,* **iff** $\{\} \subsetneq S \subsetneq V$.

Definition 2.9 *An edge $(u, v) \in E$ of graph $G = (V, E)$* crosses *cut $(S, V \setminus S)$,* **iff** *one endpoint of edge (u, v) is in S and the other is in $V \setminus S$.*

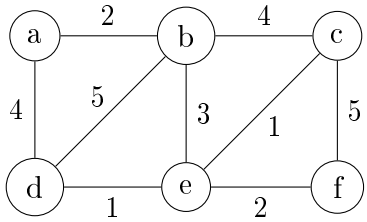
Definition 2.10 *An edge is a* light edge crossing a cut, **iff** *it crosses the cut and its weight is the minimum of the weights of the edges crossing the cut. (The expressions "*light edge crossing the cut*" and "*light edge in the cut*" are synonyms in this topic.)*

Definition 2.11 *A cut* respects *a set A of edges,* **iff** *no edge of A crosses the cut. (The verbs "*respects*" and "*avoids*" are synonyms in this topic.)*

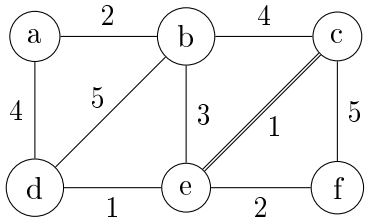
Theorem 2.12
Provided that $G = (V, E)$ is an undirected connected weighted graph, and
(1) the A set of edges is a subset *of some MST of G, and*
(2) the cut $(S, V \setminus S)$ respects *(i.e. avoids) the A set of edges, and*
(3) the edge $(u, v) \in E$ is a light edge crossing the cut $(S, V \setminus S)$
$\Longrightarrow$ *the edge (u, v) is* safe for *the A set of edges.*

Proof. $(u, v) \notin A$, because (u, v) crosses the cut, which respects A.
Let $T = (V, T_E)$ be an MST with the property $A \subseteq T_E$. There are two cases. (1) If $(u, v) \in T_E$, we are ready. (2) For $(u, v) \notin T_E$, we create a T' MST for which $A \cup \{(u, v)\}$ is a subset of T'. We note that T is a spanning tree, so in T there is exactly one path form u to v. Thus on this path there is an edge crossing the cut $(S, V \setminus S)$. Let (p, q) be such an edge. In this case, $w(p, q) \geq w(u, v) \wedge (p, q) \notin A$. Let us delete edge (p, q) from T. Then tree T becomes a spanning forest T'' of two trees. One of these trees contains vertex u, the other contains vertex v. Let us add edge (u, v) to T''. We receive a spanning tree again. Let us call it T'. As a result, $w(T') = w(T) - w(p, q) + w(u, v) \leq w(T)$. Remember that T was an MST of G. Thus $w(T') \geq w(T)$. Therefore $w(T') = w(T)$, and T' is also an MST of G. $\square$

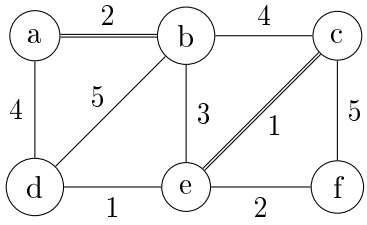
Based on the previous theorem, we have a method for building an MST. Let us denote the edges of the A set of edges with double lines. In the following example, in each step, we select a cut respecting A, we find a light edge crossing the cut, and add this edge safely to A. The example graph has 6 vertices. Thus we find an MST in 5 steps.



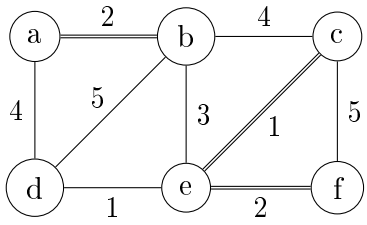
Initially $A = \{\}$, thus each cut is appropriate. Select cut $(\{a, b, c\}, \{d, e, f\})$. (c, e) is a light edge crossing this cut, consequentially it is safe for A. Add it to A.



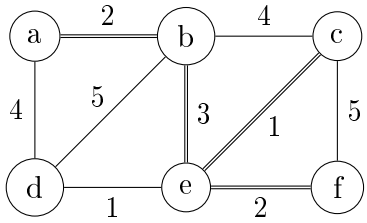
Now $A = \{(c, e)\}$. For example, cut $(\{a, f\}, \{b, c, d, e\})$ respects A. Edges (a, b) and (e, f) are the light edges crossing this cut. Both of them are safe for A. Select (a, b). Add it to A.



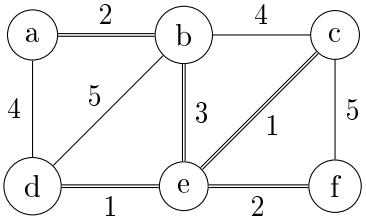
Now $A = \{(a, b), (c, e)\}$. For example, cut $(\{a, b, c, d, e\}, \{f\})$ respects A. Edge (e, f) is the light edge crossing this cut. Thus it is safe for A. Add it to A.



Now $A = \{(a, b), (c, e), (e, f)\}$. For example, cut $(\{a, b\}, \{c, d, e, f\})$ respects A. Edge (b, e) is the light edge crossing this cut. Thus it is safe for A. Add it to A.

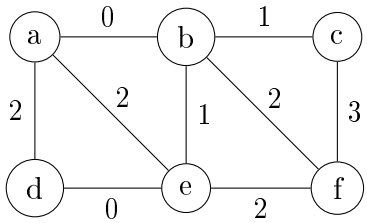


$A = \{(a, b), (b, e), (c, e), (e, f)\}$. Cut $(\{a, b, c, e, f\}, \{d\})$ respects A. Edge (d, e) is the light edge crossing this cut. Thus it is safe for A. Add it to A.



$A = \{(a, b), (b, e), (c, e), (d, e), (e, f)\}$. $|A| = 5 = |G.V| - 1$, thus A is the set of the edges of a *minimum spanning tree* (MST).

2.2 Algorithm of Kruskal



a – b, 0 ; d, 2 ; e, 2.
b – c, 1 ; e, 1 ; f, 2.
c – f, 3.
d – e, 0.
e – f, 2.

Figure 7: Connected weighted undirected graph

Components	edge	its weight	is it safe?
a, b, c, d, e, f	(a,b)	0	+
ab, c, d, e, f	(d,e)	0	+
ab, c, de, f	(b,c)	1	+
abc, de, f	(b,e)	1	+
abcde, f	(a,d)	2	-
abcde, f	(a,e)	2	-
abcde, f	(b,f)	2	+
abcdef	-	-	-

;

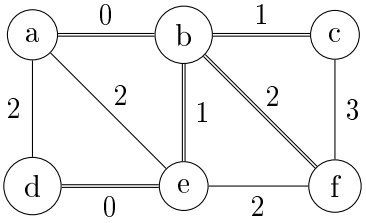
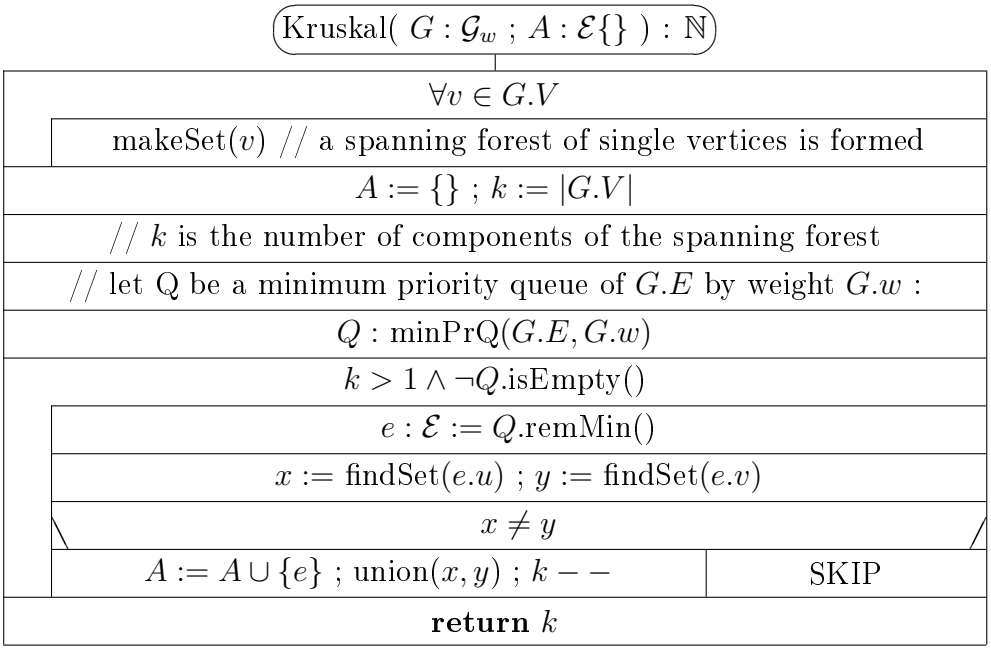


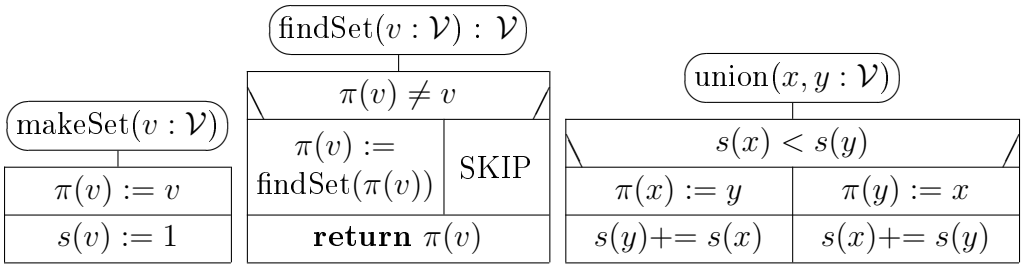
Figure 8: 7. Double lines show the edges of the MST found.



This algorithm checks whether G is connected. If so, it returns $k = 1$. Otherwise $k > 1$.

$MT(n, m) \in O(m * \lg n)$

2.2.1 The set operations of the Kruskal algorithm

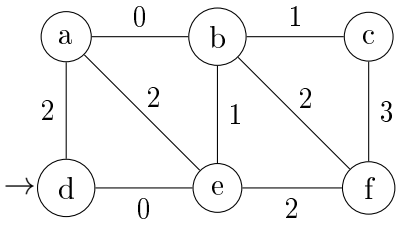


2.3 Algorithm of Prim

Prim($G : \mathcal{G}_w$; $r : \mathcal{V}$)

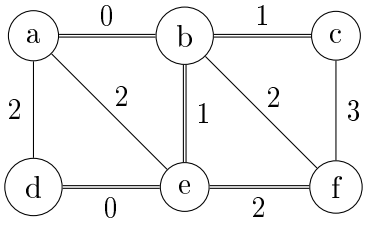
- $\forall v \in G.V$
  - $c(v) := \infty$; $p(v) := \oslash$ // costs and parents still undefined
  - // edge $(p(v), v)$ will be in the MST where $c(v) = G.w(p(v), v)$
- $c(r) := 0$ // r is the root of the MST where $p(r)$ remains $\oslash$
- // let Q be a minimum priority queue of $G.V \setminus \{r\}$ by label values $c(v)$:
- Q : minPrQ($G.V \setminus \{r\}, c$) // $c(v)$ = cost of light edge to (partial) MST
- $u := r$ // vertex $u = r$ has become the first node of the (partial) MST
- $\neg Q$.isEmpty()
  - // neighbors of u may have come closer to the partial MST
  - $\forall v : (u, v) \in G.E \wedge v \in Q \wedge c(v) > G.w(u, v)$
    - $p(v) := u$; $c(v) := G.w(u, v)$; Q.adjust(v)
  - $u := Q$.remMin() // $(p(u), u)$ is a new edge of the MST

$$MT_{\mathrm{P}rim}(n, m) \in O(m * \log n).$$



a – b, 0 ; d, 2 ; e, 2.
b – c, 1 ; e, 1 ; f, 2.
c – f, 3.
d – e, 0.
e – f, 2.

c values of nodes in Q						vertex	changes of labels p					
a	b	c	d	e	f	into	a	b	c	d	e	f
∞	∞	∞	0	∞	∞	MST	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$
2	∞	∞		0	∞	d	d				d	
2	1	∞			2	e		e				e
0		1			2	b	b		b			
		1			2	a						
					2	c						
						f						
0	1	1	0	0	2	result	b	e	b	$\oslash$	d	e



Compared to the algorithm of Kruskal, using the algorithm of Prim, we have received another MST of the same graph.

3 Single-Source Shortest Paths ([3] 24)

Problem 3.1 *Given graph $G : \mathcal{G}_w$ where $s \in G.V$ is the source vertex. We search for a shortest path to each vertex available from s in G. This is called the* single-source shortest paths *problem.*

Note 3.2 *In the following algorithms the undirected graphs will be modeled with digraphs where for each edge (u, v) of the graph, (v, u) is also edge of the graph, and $w(u, v) = w(v, u)$.*

This model simplifies the forthcoming discussions. In the rest of these lecture notes, graph means digraph by default.

Definition 3.3 *A loop of a digraph is* negative, **iff** *the sum of the weights of its edges is negative.*

Consequence 3.4 *Problem 3.1 can be solved,* **iff** *there is no negative cycle available from s.*

Note 3.5 *Remember that undirected graphs are modeled with digraphs here. This simplification has a consequence: if an undirected graph contains a negative edge, it also contains a negative cycle in this model, according to definition 3.3. And in case of undirected graphs, we can solve the* single-source shortest paths *problem (i.e. 3.1),* **iff** *there is no negative edge available from s.*

Provided that problem 3.1 can be solved, we consider the result of path-finding now. We have two possibilities for each vertex $v \in G.V \setminus \{s\}$:
- If there is some path form s to v, $d(v)$ is the length of the shortest (i.e. optimal) path, and $\pi(v)$ is the parent of vertex v on such a path.
- If there is no path from s to v, $d(v) = \infty$ and $\pi(v) = \oslash$.

Considering vertex s, $d(s) = 0$ and $\pi(s) = \oslash$ are the appropriate results, because the optimal path consists of only vertex s.

The shortest path finding algorithms approximate the optimal paths step-by-step.

Let us consider a given moment of the running algorithm. Let the shortest path found form s to v be denoted by $s \rightsquigarrow v$. Let the length of this path be denoted by $w(s \rightsquigarrow v)$. Then we have a common invariant of the forthcoming algorithms solving problem 3.1, and this invariant becomes true when we initialize the algorithm:
- Provided that we already have found some path from s to v, $d(v) = w(s \rightsquigarrow v)$ and $\pi(v)$ is the parent of $v \neq s$ on $s \rightsquigarrow v$.

- If still there is no $s \rightsquigarrow v$, i.e. still we have not found any path from s to v, then $d(v) = \infty$ and $\pi(v) = \oslash$.

In order to approximate the optimal paths step-by-step, edges (u, v), i.e. $u \rightarrow v$ of the graph are considered systematically. Provided that path $s \rightsquigarrow u \rightarrow v$ turns out shorter than $s \rightsquigarrow v$, $s \rightsquigarrow u \rightarrow v$ becomes the new $s \rightsquigarrow v$. At code level this means that we perform the following conditional, which is called *relaxation*. (Because of the special features of the different algorithms, this relaxation may contain some additional statements.)

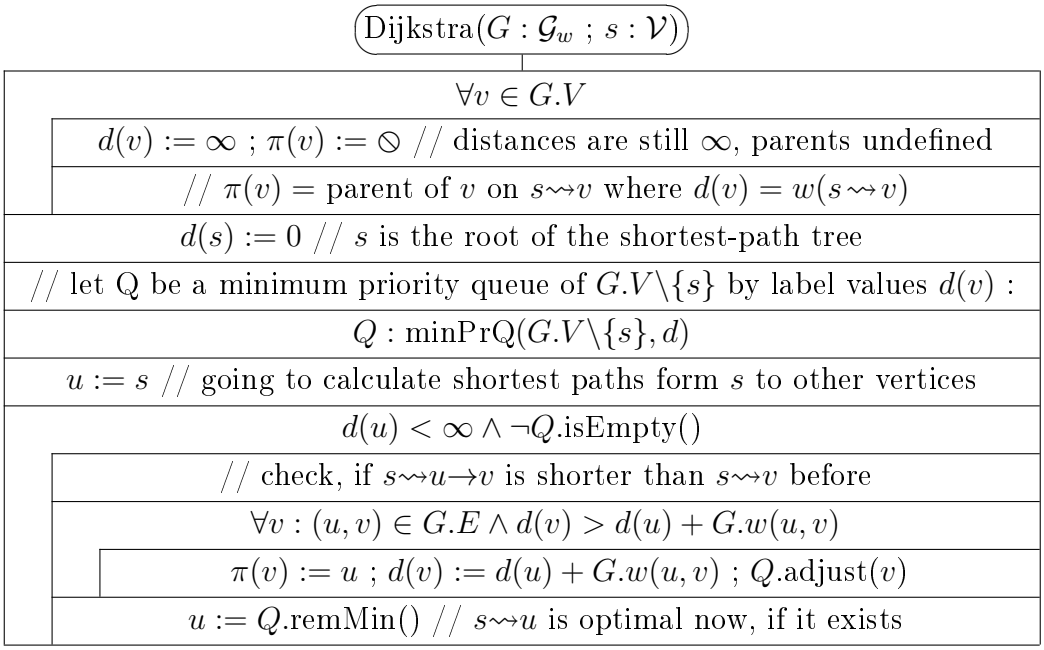
$d(v) > d(u) + G.w(u, v)$	
$\pi(v) := u \ ; \ d(v) := d(u) + G.w(u, v)$	SKIP

Many of the *single-source shortest paths* algorithms have an explicit set of the vertices *to be processed*. They repeatedly select and remove a vertex form this set, then perform relaxation on each edge going out from this vertex. All these relaxations together are called the *expansion* of this vertex. Processing a vertex means its selection + removal + expansion.

3.1 Dijkstra's algorithm

Precondition: In graph $G : \mathcal{G}_w$, $\forall (u, v) \in G.E$: $G.w(u, v) \geq 0$, i.e. each edge of the graph has non-negative weight.

In this case there is no negative cycle. Thus problem 3.1 (the single-source shortest paths problem) can be solved.



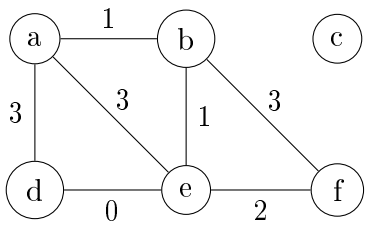
$MT_{\mathrm{Dijkstra}}(n, m) \in O((n + m) * \lg n)$. This result can be received similarly to that of algorithm Prim, because of the striking similarities between the two algorithms, although they solve completely different problems, and the interpretations of their results are also totally different. An important technical difference: here we sum up the weights of the edges along a path, but we have nothing to do with paths there.

$mT_{\mathrm{Dijkstra}}(n, m) \in \Theta(n)$, which is realized when s, i.e. the source vertex has no successor.

Exercise 3.6 *Implement Dijkstra's algorithm in case of adjacency matrix representation of the graph. Represent the priority queue with an unsorted array. What can we say about asymptotic run time efficiency (i.e. operational complexity)? Can we develop it with a more sophisticated representation of the priority queue?*

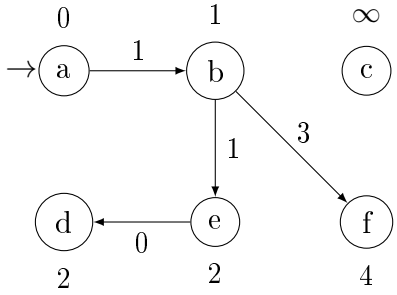
Exercise 3.7 *Implement Dijkstra's algorithm in case of adjacency lists representation of the graph. Be careful about keeping the theoretical* $MT_{\mathrm{Dijkstra}}(n, m) \in O((n + m) * \lg n)$ *and* $mT_{\mathrm{Dijkstra}}(n, m) \in \Theta(n)$. *(The appropriate representation and implementation of the priority queue is the key.)*

We illustrate Dijkstra's algorithm on the next graph where vertex **a** is the source node.



a – b, 1 ; d, 3 ; e, 3.
b – e, 1 ; f, 3.
c.
d – e, 0.
e – f, 2.

d values of nodes in Q						expanded	changes of π values					
a	b	c	d	e	f	vertex	a	b	c	d	e	f
0	∞	∞	∞	∞	∞	: d	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$
	1	∞	3	3	∞	a : 0		a		a	a	
		∞	3	2	4	b : 1					b	b
		∞	2		4	e : 2				e		
		∞			4	d : 2						
		∞				f : 4						
0	1	∞	2	2	4	result	$\oslash$	a	$\oslash$	e	b	b



The shortest paths tree in case of $s = \mathrm{a}$. We display also the unavailable node/nodes with ∞ d value.

Theorem 3.8 *When we select vertex u for expansion (first with statement $u := s$, later with $u := Q.remMin()$), then we have already found optimal path to u, provided that $d(u) < \infty$.*

Theorem 3.9 *If vertex u is selected for expansion and $d(u) = \infty$, then no element of $Q \cup \{u\}$ is available form s.*

Consequence 3.10 *While Dijkstra's algorithm selects a vertex with finite d value for expansion, we already have the optimal path to this vertex.*

If $d(u) < \infty$ for each vertex selected, then the second loop of the algorithm makes Q empty (with $n-1$ iterations), and stops when it has found optimal path to each vertex of the graph.

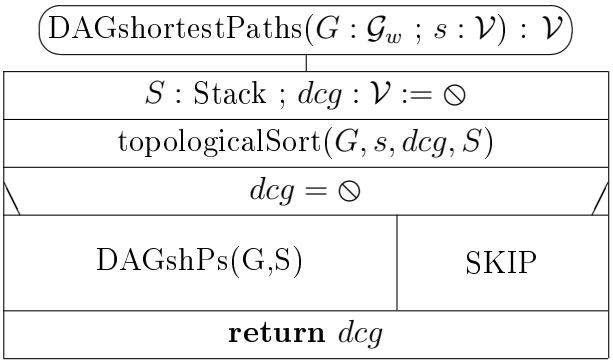
If some time it selects vertex u with $d(u) = \infty$, then no element of $Q \cup \{u\}$ is available from s. We stop, because $d(u) = \infty$. Each element of $Q \cup \{u\}$ has ∞ d value and $\oslash$ π value, while the vertices selected earlier are those available from s. And we have found optimal path to them.

3.2 Single-source shortest paths in DAGs

Precondition: Graph $G : \mathcal{G}_w$ is directed, and there is no loop available from s.

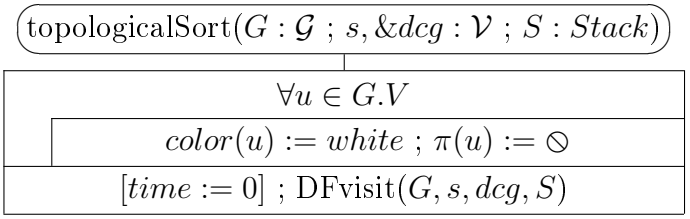
In this case there is no negative cycle available from s, and the *single-source shortest paths* problem has solution.

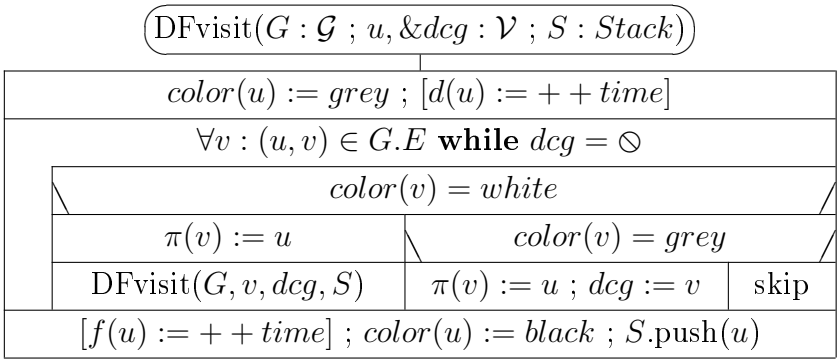
This algorithm checks its precondition. If it is satisfied, function DAGshortestPaths() returns $\oslash$. Otherwise it finds a directed loop, and it returns with some node of this loop. Starting from this vertex, and going through the π labels of the nodes. the loop can be traversed in reversed direction.

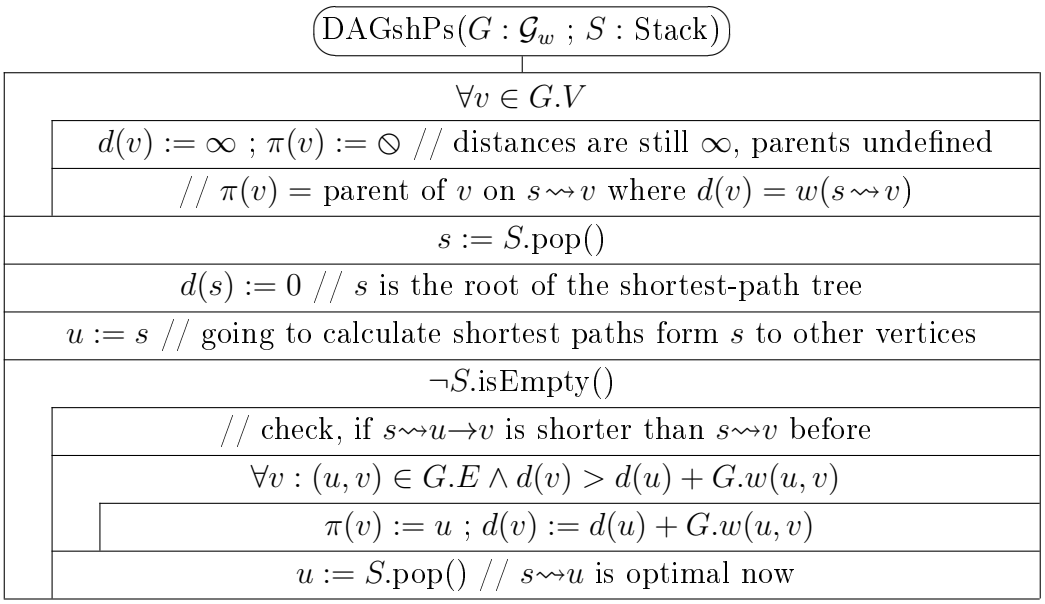


Procedure topologicalSort() tries to make a topological order of the vertices available from s.

Variable *time* (which is supposed to be global here) is needed only for the presentation of the run of the algorithm. Each statement corresponding to *time* can be omitted from an implementation. Thus these statements are put into square brackets in the structograms.





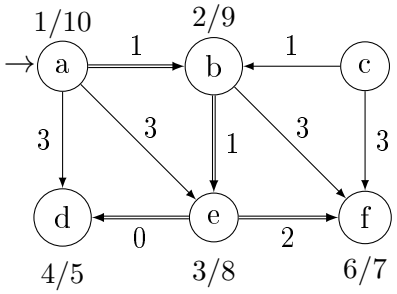


$$MT(n, m) \in \Theta(n + m) \quad \wedge \quad mT(n, m) \in \Theta(n)$$

Exercise 3.11 *When the operational complexity of this algorithm is the smallest? When it is the greatest? Why?*

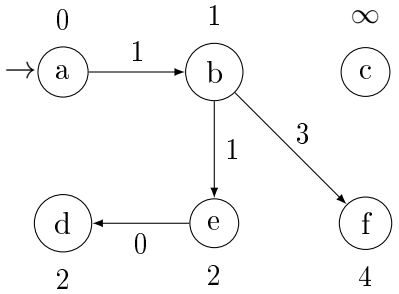
Theorem 3.12 *Provided that there is no loop available from s,*
- algorithm DAGshortestPaths() finds optimal path to the vertices available form s, and
- for each vertex x unavailable from s, it terminates with $d(x) = \infty$ and $\pi(x) = \oslash$.

We illustrate the algorithm on the DAG below where s = a. First we sort topologically the subgraph available from s. Then – after the usual initialization –, the vertices are expanded according to their topological order.



a → b, 1 ; d, 3 ; e, 3.
b → e, 1 ; f, 3.
c → b, 1 ; f, 3.
e → d, 0 ; f, 2.
$S = \langle$ a, b, e, f, d $\rangle$

changes of d values						expanded	changes of π labels					
a	b	c	d	e	f	vertex	a	b	c	d	e	f
0	∞	∞	∞	∞	∞	: d	⊘	⊘	⊘	⊘	⊘	⊘
	1		3	3		a : 0		a		a	a	
				2	4	b : 1					b	b
			2			e : 2				e		
						f : 4						
						d : 2						
0	1	∞	2	2	4	result	⊘	a	⊘	e	b	b



The shortest paths tree in case of $s =$ a. We display also the unavailable node/nodes with ∞ d value.

Exercise 3.13 *Implement the* DAG single source shortest paths *algorithm in case of* adjacency matrix *representation of the graph. What can you say about the asymptotic run time efficiency (i.e. operational complexity) of this implementation?*

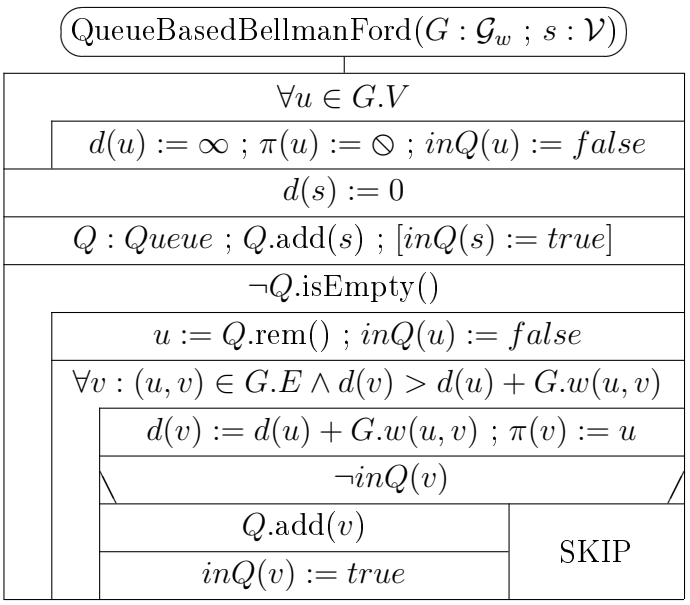
Exercise 3.14 *Implement the* DAG single source shortest paths *algorithm in case of* adjacency lists *representation of the graph. What can you say about the asymptotic run time efficiency (i.e. operational complexity) of this implementation?*

3.3 Queue-based Bellman-Ford algorithm

Precondition: There is no negative cycle available from s. (In case of directed graph there can be negative edges, i.e. edges with negative weights.)

The *Queue-based Bellman-Ford algorithm (QBF)* has unknown author (with neither theoretical analysis nor checking negative cycles). According to our best knowledge it was analyzed and published by professor Robert Endre Tarjan. He calls it *Breadth-first Scanning* in [8].[1]

QBF is similar to *breadth-first search* but the length of a path is calculated as the sum of the weights of the edges along the path. First it puts only s into the queue. Similarly to BFS, after the usual initialization, in the main loop, it repeatedly removes the first element of the queue, and expands it. Unlike BFS, QBF makes relaxation for each neighbor of this *actual node*. Provided that through the actual node, QBF finds a shorter path to a neighbor of it, QBF modifies the d and π labels of this neighbor accordingly. If this neighbor is not in the queue, QBF adds it to the end of the queue.



Because there is no operation $v \in Q$ for type Queue, we could introduce new type *Transparent_ queue*. We prefer introducing logic label $inQ(v)$ which is true $\iff$ $v \in Q$. (If we index the vertices from 1 to n, at implementation

[1] Tarjan received the Turing Award in 1986. The citation for the award states that it was: "For fundamental achievements in the design and analysis of algorithms and data structures." He has Hungarian roots.

level labels $inQ(v)$ can be represented by array $inQ/1 : \mathbb{B}[n]$. Undoubtedly this array could be part of the implementation of type *Transparent_ queue*.)

Provided that there is no negative cycle available from s, this simple version of QBF computes a path $s \overset{opt}{\rightsquigarrow} v$ for each vertex v available from s, and it stops with empty queue. If there is some negative cycle available from s, the expansions *go around* along such a negative cycle, and the algorithm gets into infinite loop.

3.3.1 Handling negative cycles

In order to handle this later case, Tarjan defined *passes* of QBF (see 3.3.5), and he proved that the algorithm stops in n passes, **iff** there is no negative cycle available from s. He also elaborated other criteria [8]. Here we use a relatively simple, easy-to-check criterion of the author of these lecture notes [10].

We introduce label $e(v)$ for each vertex v of the graph where $e(v)$ is the number of edges along $s \rightsquigarrow v$. We can prove that there is no negative cycle available from $s \iff$ during the run of QBF, for each vertex v accessed by its main loop, $e(v) < n$. (See [10] for the details.)

The statement above is clearly equivalent to the following one. There is some negative cycle available from $s \iff$ during the run of QBF, for some vertex v accessed by its main loop, $e(v) \geq n$ becomes true. In this case vertex v is part of some negative cycle which can be identified by the π labels of the vertices of this cycle, or starting from vertex v, the π labels of the vertices lead us into such a negative cycle. Obviously, the whole process of identifying this negative cycle needs maximum n steps where n is the number of the vertices of the graph.

Hereinafter, for each vertex v reached from s, we record label $e(v)$ besides labels $d(v)$ and $\pi(v)$.

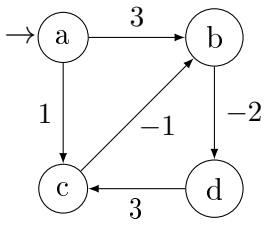
When we make a relaxation through edge (u, v), and we find that $e(v) \geq n$, then we identify some vertex of some negative cycle according to the previous method, and our version of algorithm QBF returns with this vertex.

Provided that during the run of QBF, after each relaxation $e(v) < n$, we stop with empty queue and return with $\oslash$.

It can be proved that our version of QBF stops in $O(n * m)$ time in both cases (where n is the number of vertices and m is the number of edges of the graph).

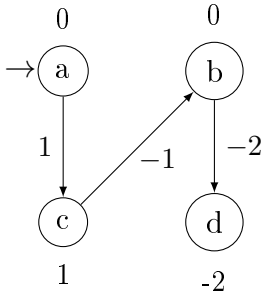
3.3.2 Illustration of finding optimal paths

Here we find a path $s \overset{opt}{\rightsquigarrow} v$ to each vertex v available from s.



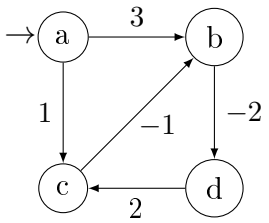
a → b, 3 ; c, 1.
b → d, −2.
c → b, −1.
d → c, 3.

changes of $d; e$				expanded	Q :	changes of π			
a	b	c	d	vertex	Queue	a	b	c	d
$0; 0$	∞	∞	∞	: $d; e$	$\langle$ a $\rangle$	⊘	⊘	⊘	⊘
	$3; 1$	$1; 1$		a: $0; 0$	$\langle$ b, c $\rangle$		a	a	
			$1; 2$	b: $3; 1$	$\langle$ c, d $\rangle$				b
	$0; 2$			c: $1; 1$	$\langle$ d, b $\rangle$		c		
				d: $1; 2$	$\langle$ b $\rangle$				
			$-2; 3$	b: $0; 2$	$\langle$ d $\rangle$				b
				d: $-2; 3$	$\langle\ \rangle$				
0	0	1	-2	final d and π values		⊘	c	a	b



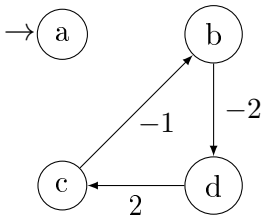
The shortest paths tree where $s = \text{a}$. We indicated only the d values of the nodes.

3.3.3 Illustration of handling negative cycles



a → b, 3 ; c, 1.
b → d, −2.
c → b, −1.
d → c, 2.

changes of $d; e$				expanded	Q :	changes of π			
a	b	c	d	vertex	Queue	a	b	c	d
$0; 0$	∞	∞	∞	: $d; e$	⟨ a ⟩	⊘	⊘	⊘	⊘
	$3; 1$	$1; 1$		a: $0; 0$	⟨ b, c ⟩		a	a	
			$1; 2$	b: $3; 1$	⟨ c, d ⟩				b
	$0; 2$			c: $1; 1$	⟨ d, b ⟩		c		
				d: $1; 2$	⟨ b ⟩				
			$-2; 3$	b: $0; 2$	⟨ d ⟩				b
		$0; 4$		d: $-2; 3$	⟨ ⟩			d	

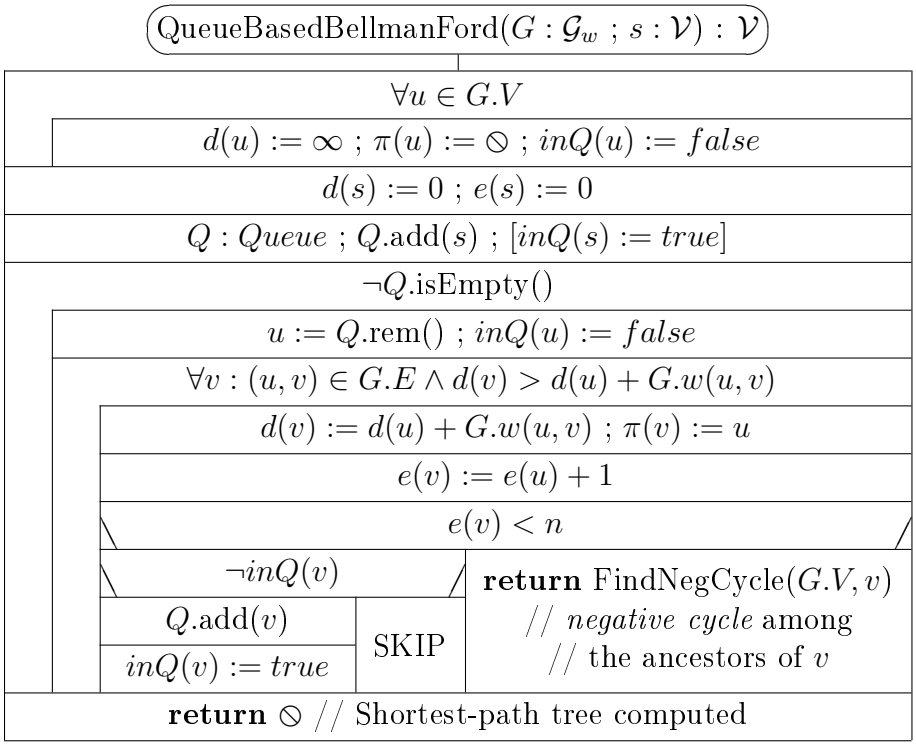


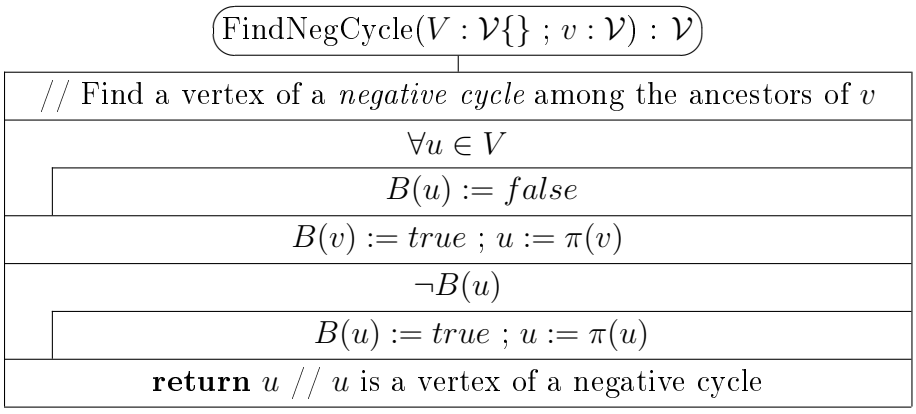
We stop here because $e(\mathrm{c}) = 4 = n$, which means that there is a negative cycle among the ancestors of vertex "c". Going back according to the π values we find the negative cycle.

3.3.4 Our version of Queue-based Bellman-Ford algorithm (QBF)

Here we give the structograms of the our version of QBF. Do not forget that when we have modified labels $d(v)$, $\pi(v)$ and $e(v)$ of vertex v, we add v to the end of the queue $\iff$ $e(v) < n$, and v is not currently in the queue.

If we know in advance that there is no negative cycle available from s, then those parts of the subsequent functions can be omitted which correspond to labels $e(v)$ or negative cycles (see the previous version of QBF).





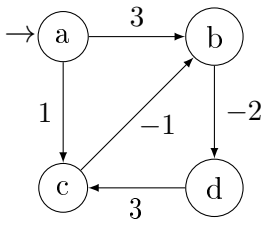
3.3.5 Analyzing QBF

We analyze only those cases when there is no negative cycle available from s.

In order to prove the correctness of QBF and to analyze its efficiency it is fundamental to divide its run into passes.

Definition 3.15 *Recursive definition of passes:*
– Pass 0: processing the source vertex (s).
– Pass (i+1): processing the vertices in the queue at the end of pass i.

For example, let us consider the illustration of our algorithm together with pass counting. **(Illustration of finding paths $s \overset{opt}{\rightsquigarrow} v$.)**



a → b, 3 ; c, 1.
b → d, −2.
c → b, −1.
d → c, 3.

changes of $d; e$				expanded	Q :	changes of π				
a	b	c	d	vertex	Queue	a	b	c	d	Pass
$0; 0$	∞	∞	∞	: $d; e$	$\langle$ a $\rangle$	⊘	⊘	⊘	⊘	
	$3; 1$	$1; 1$		a: $0; 0$	$\langle$ b, c $\rangle$		a	a		0.
			$1; 2$	b: $3; 1$	$\langle$ c, d $\rangle$				b	1.
	$0; 2$			c: $1; 1$	$\langle$ d, b $\rangle$		c			1.
				d: $1; 2$	$\langle$ b $\rangle$					2.
			$-2; 3$	b: $0; 2$	$\langle$ d $\rangle$				b	2.
				d: $-2; 3$	$\langle\ \rangle$					3.
0	0	1	-2	final d and π values		⊘	c	a	b	-

Property 3.16 *For each vertex u of the graph, if there is no negative cycle available from s, and there is some path $s \overset{opt}{\rightsquigarrow} u$ consisting of k edges, then by the beginning of pass k, $d(u) = w(s \overset{opt}{\rightsquigarrow} u)$, and $\langle s, \ldots, \pi^2(u), \pi(u), u\rangle$ is an optimal path.*

Proof. Use mathematical induction according to k. □

Lemma 3.17 *If there is no negative cycle available from s, then for each vertex u available from s, there is some path $s \overset{opt}{\rightsquigarrow} u$ consisting of maximum $n-1$ edges.*

Proof. In this case the paths containing no cycle are not longer then those containing cycle. A path containing no cycle has at most $n-1$ edges. Consequently there are finite number of paths containing no cycle. As a result, for each vertex u, there are finite number of paths from s to u which contain no cycle. Therefore there is optimal one among them. □

Theorem 3.18 *(Consequence of the Property and Lemma above)*
If there is no negative cycle available from $s \Rightarrow$ for each vertex u available

from s, there is some path $s \overset{opt}{\rightsquigarrow} u$ computed by the beginning of pass $n-1 \Rightarrow$ by the end of pass $n-1$ the queue becomes empty and the algortihm stops in $O(n*m)$ running time, i.e.

$$MT(n,m) \in O(n*m).$$

This theoretical time complexity does not guarantee efficiency. Fortunately practical tests support that in case of large, randomly generated sparse graphs ($m \in O(n)$) with positive edge-weights where the vertices of the graph are available from s, the average running time of QBF is statistically $\Theta(n)$, provided that we represent the graph with adjacency lists. And time complexity $\Theta(n)$ is the theoretical minimum. (Most of the networks can be modeled with large, sparse graphs. It is not accidental that this simple but general algorithm is often used for finding optimal paths in networks.)

4 All-Pairs Shortest Paths ([3] 25)

In this chapter, we consider how to compute the *transitive closure* of a finite binary relation. Next we introduce algorithm *Floyd-Warshall* that searches for shortest paths between each pairs of vertices of a weighted graph. Both algorithms

- are based on the adjacency representation of graphs,
- have computation complexity $\Theta(n^3)$,
- are classical examples of *dynamic programming* [3].

Floyd's *Floyd-Warshall algorithm* solves more general problem than the *Transitive Closure algorithm* of Roy and Warshall, and it is later than the other one.

Before going to the details of these algorithms, we shortly introduce dynamic programming.

4.1 Dynamic programming

Dynamic programming is similar to the *divide-and-conquer* method. It also has some trivial base case or cases which can be solved directly. It also divides more complex or larger problems into smaller subproblems, solves these subproblems and combines their solutions in order to solve the original problem. The main difference is that the *divide-and-conquer* method solves the subproblems independently from each other, like in case of *merge sort*. Thus it is effective when the subpromlems and subsubproblems etc. are typically independent from each other. It becomes inefficient when a larger problem has common subproblems and subsubproblems recursively. In *dynamic programming*, we solve each subproblem only once, and remember its solution whenever we meet that subproblem again.

For example, consider the Fibonacci function.
$F : \mathbb{N} \to \mathbb{N}$
$F_n = F_{n-1} + F_{n-2} \quad$ if $n > 1$
$F_1 = 1 \quad$ and $\quad F_0 = 0.$

Provided that $n > 1$, can divide computing F_n into computing F_{n-1} and F_{n-2}, and conquer with adding the results of the two recursive calls. As a result, for example
$F_9 = F_8 + F_7 = (F_7 + F_6) + F_7 = ([F_6 + F_5] + F_6) + (F_6 + F_5) =$
$([\{F_5 + F_4\} + F_5] + \{F_5 + F_4\}) + (\{F_5 + F_4\} + F_5) = \ldots$

Thus F_9 and F_8 are computed only once, F_7 is computed twice, F_6 is computed 3 times, F_5 is computed 5 times, and so on. In general, F_{n-k} is computed F_{k+1} times, and it follows that computing F_n needs exponential time of n.

On the contrary F_n can be computed in linear time, if we start with computing with F_2, F_3, F_4 in this order, and so on, always remembering the last two partial results. This is a trivial case of dynamic programming. In the subsequent two algorithms we use it in a much more complex way, and the partial results will be stored in matrices.

4.2 Transitive closure of a directed graph (TC)

In this subsection, for each pair of vertices (u, v) of a network/graph, we want to determine whether there is any path form u to v or not. We are interested neither in the path nor in its length. For this purpose we introduce the following notion.

Definition 4.1 *Given graph $G = (V, E)$, its transitive closure is relation $T \subseteq V \times V$ where*

$(u, v) \in T \iff$ *there is some path from vertex u to vertex v in graph G.*

Notation 4.2 $\mathbb{B} = \{0; 1\}$

Provided that the vertices of a graph can be identified by indices $1..n$, we can represent the graph with adjacency matrix $A/1 : \mathbb{B}[n, n]$. And we can represent its transitive closure with matrix $T/1 : \mathbb{B}[n, n]$ where

$T[i, j] \iff$ there is some path from vertex i to vertex j in the graph represented with matrix A.

In order to compute matrix T, let us define matrix sequence $\langle T^{(0)}, T^{(1)}, T^{(n)} \rangle$ where $T^{(n)} = T$.

Notation 4.3 *$i \overset{k}{\rightsquigarrow} j$ is a path from vertex i to vertex j where the indices of the vertices between i and j are $\leq k$ ($i > k$ and $j > k$ is possible, where $i, j \in 1..n$ and $k \in 0..n$).*

Definition 4.4 $T_{ij}^{(k)} \iff \exists\, i \overset{k}{\rightsquigarrow} j \quad (k \in 0..n \ \wedge \ i, j \in 1..n)$

Property 4.5 *(Recursive relation among the T matrices.)*

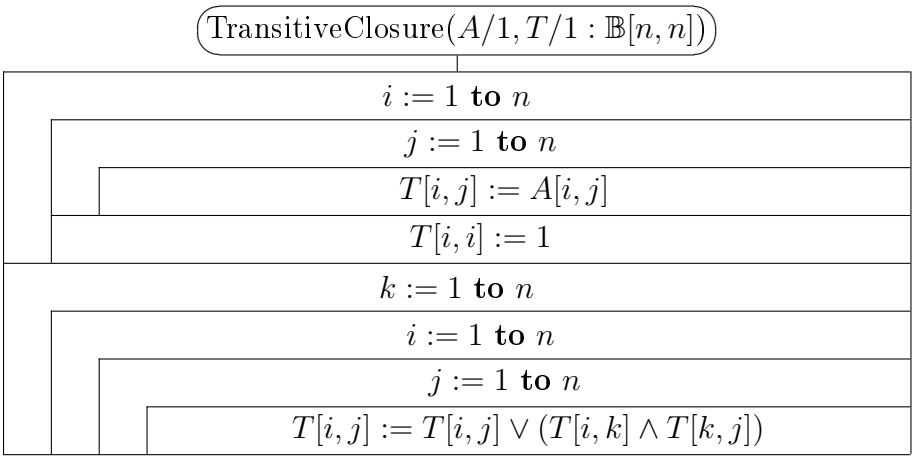
$T_{ij}^{(0)} = A[i, j] \vee (i = j) \qquad (i, j \in 1..n)$

$T_{ij}^{(k)} = T_{ij}^{(k-1)} \vee T_{ik}^{(k-1)} \wedge T_{kj}^{(k-1)} \quad (k \in 1..n \ \wedge \ i, j \in 1..n)$

Consequence 4.6
$T_{ik}^{(k)} = T_{ik}^{(k-1)} \quad \wedge \quad T_{kj}^{(k)} = T_{kj}^{(k-1)} \quad (k \in 1..n \ \wedge \ i, j \in 1..n)$

This means that column k and row k of matrix $T^{(k)}$ are the same as the appropriate column and row of matrix $T^{(k-1)}$. $(k \in 1..n)$

Let us notice that a single matrix T is enough for the whole computation, because $T_{ij}^{(k)}$ depends only on $T_{ij}^{(k-1)}$, $T_{ik}^{(k-1)}$ and $T_{kj}^{(k-1)}$, where $T_{ik}^{(k)} = T_{ik}^{(k-1)} \wedge T_{kj}^{(k)} = T_{kj}^{(k-1)}$



Clearly $T(n) \in \Theta(n^3)$ for procedure TransitiveClosure.

4.2.1 Computing transitive closure with breadth-first search

When BFS has been completed with $s = i$, the black vertices are those available from vertex i, and the white vertices are those unavailable from vertex i.

As a result, we can simplify BFS. We do not count d and π values just a boolean label $nonwhite(j)$ for each vertex j. Then $T[i, j] \iff nonwhite(j)$ where $j \in 1..n$. And this is row i of matrix T where $i \in 1..n$. Thus $MT(n, m) \in \Theta(n + m)$ for a single row of T.

Thus the whole matrix is computed with $\Theta(n*(n+m))$ maximal running time. This is $\Theta(n^2)$ on sparse graphs which is asymptotically better than algorithm TC.

But on dense graphs, this is asymptotically $\Theta(n^3)$ which means practically longer running time than that of the extremely simple TC algorithm.

4.3 The Floyd-Warshall algorithm (FW)

Notation 4.7 $\mathbb{R}_\infty = \mathbb{R} \cup \{\infty\}$

Given a weighted graph with adjacency matrix $A/1 : \mathbb{R}_\infty[n,n]$. FW computes matrix $D/1 : \mathbb{R}_\infty[n,n]$ where $D[i,j]$ is the length of an optimal path from vertex i to vertex j; or $D[i,j] = \infty$, if there is no path from i to j. FW also computes matrix $\pi/1 : \mathbb{N}[n,n]$ where $\pi[i,j]$ is the parent node of vertex j on an optimal path from i to j, if $i \neq j$ and there is some path from i to j. Otherwise $\pi[i,j] = 0$.

Precondition: There is no negative cycle in the graph. (This condition is checked by the algorithm.)
Task: FW constructs the following sequence of matrix pairs:
$\langle (D^{(0)}, \pi^{(0)}), (D^{(1)}, \pi^{(1)}), \ldots, (D^{(n)}, \pi^{(n)}) \rangle$ where $D^{(0)} = A$, $\pi^{(0)}$ and matrix pairs $(D^{(k)}, \pi^{(k)})$ $[k \in 1..n]$ can be constructed according to their properties which we are going to present, $D = D^{(n)} \wedge \pi = \pi^{(n)}$.

Notation 4.8 $i \underset{opt}{\overset{k}{\rightsquigarrow}} j$ *($k \in 1..n$) is a shortest path from vertex i to vertex j with two constraints:*

- *On this path, the indices of the vertices between vertex i and vertex j are $\leq k$.*
- *This path contains no cycle.*

Note 4.9

- *If $i = j$ then $i \underset{opt}{\overset{k}{\rightsquigarrow}} j = \langle i \rangle$.*
- *If $i \neq j$ then $\exists\, i \underset{opt}{\overset{0}{\rightsquigarrow}} j = \langle i, j \rangle \iff (i,j)$ is an edge of the graph.*
- *If $i \neq j \wedge k \in 1..n$, there are two possibilities about path $i \underset{opt}{\overset{k}{\rightsquigarrow}} j$:*
  $i \underset{opt}{\overset{k}{\rightsquigarrow}} j = i \underset{opt}{\overset{k-1}{\rightsquigarrow}} j \quad \vee \quad i \underset{opt}{\overset{k}{\rightsquigarrow}} j = i \underset{opt}{\overset{k-1}{\rightsquigarrow}} k \underset{opt}{\overset{k-1}{\rightsquigarrow}} j.$

Definition 4.10 $D_{ij}^{(k)} = \begin{cases} w(i \underset{opt}{\overset{k}{\rightsquigarrow}} j) & \text{if } i \overset{k}{\rightsquigarrow} j \text{ exists} \\ \infty & \text{if } i \overset{k}{\rightsquigarrow} j \text{ does not exist} \end{cases}$

Definition 4.11

$$\pi_{ij}^{(k)} = \begin{cases} \textit{the parent of vertex } j \textit{ on a path } \underset{opt}{i \overset{k}{\rightsquigarrow} j}, & \textit{if } i \neq j \wedge i \overset{k}{\rightsquigarrow} j \textit{ exists} \\ 0 \qquad \textit{if } i = j \vee i \overset{k}{\not\rightsquigarrow} j \textit{ does not exist} \end{cases}$$

Property 4.12 *Matrix $D^{(0)}$ is equal to adjacency matrix A of the graph, and matrix $D^{(n)}$ is equal to matrix D to be computed.*

Property 4.13

$$\pi_{ij}^{(0)} = \begin{cases} i & \textit{if } i \neq j \wedge (i,j) \textit{ is an edge of the graph} \\ 0 & \textit{if } i = j \vee (i,j) \textit{ is not edge of the graph} \end{cases}$$

And matrix $\pi^{(n)}$ is equal to matrix π to be computed.

Property 4.14 *based on note 4.9, provided that $k \in 1..n$:*
If $D_{ij}^{(k-1)} > D_{ik}^{(k-1)} + D_{kj}^{(k-1)}$
then $D_{ij}^{(k)} = D_{ik}^{(k-1)} + D_{kj}^{(k-1)} \wedge \pi_{ij}^{(k)} = \pi_{kj}^{(k-1)}$
else $D_{ij}^{(k)} = D_{ij}^{(k-1)} \wedge \pi_{ij}^{(k)} = \pi_{ij}^{(k-1)}$

Consequence 4.15 *Provided that $k \in 1..n$:*
$D_{ik}^{(k)} = D_{ik}^{(k-1)} \wedge \pi_{ik}^{(k)} = \pi_{ik}^{(k-1)} \quad \wedge \quad D_{kj}^{(k)} = D_{kj}^{(k-1)} \wedge \pi_{kj}^{(k)} = \pi_{kj}^{(k-1)}$

Note 4.16 *about the correctness of function* FloydWarshall(A, D, π) *on Figure 9. (We suppose here that the precondition is satisfied, i.e. there is no negative loop in the graph.)* We prove that $D = D^{(n)} \wedge \pi = \pi^{(n)}$ at the end of the function, where D and π are the matrices of the function, while $D^{(n)}$ and $\pi^{(n)}$ are the final theoretical matrices defined in 4.10 and 4.11.

Let us consider matrices D, π of the function and theoretical matrices $D^{(k)}$, $\pi^{(k)}$ where $k \in 1..n$.

The first double for-loop above initializes the matrices of the function as $D = D^{(0)} \wedge \pi = \pi^{(0)}$.

*A single iteration of the main loop ($k := 1$ **to** n) computes the matrix pair $(D^{(k)}, \pi^{(k)})$ from $(D^{(k-1)}, \pi^{(k-1)})$. And this computation is done in the matrix pair (D, π) of the function in the following way.*

Let us suppose that we are at the beginning of iteration k of the main loop where $D = D^{(k-1)} \wedge \pi = \pi^{(k-1)}$. (We have seen that it is true for $k = 1$.)

Now let us suppose that we arrive at condition $D[i,j] > D[i,k] + D[k,j]$ where $i, j \in 1..n$, and for the earlier iterations $i = i', j = j'$ of the inner, double loop $D[i',j'] = D_{i'j'}^{(k)} \wedge \pi[i',j'] = \pi_{i'j'}^{(k)}$ has been ensured.

Clearly $D[i,j] = D_{ij}^{(k-1)} \wedge \pi[i,j] = \pi_{ij}^{(k-1)}$, because $D[i,j]$ and *$\pi[i,j]$ has not been updated yet. And $D[i,k] = D_{ik}^{(k-1)} = D_{ik}^{(k)} \wedge D[k,j] = D_{kj}^{(k-1)} = D_{kj}^{(k)}$*

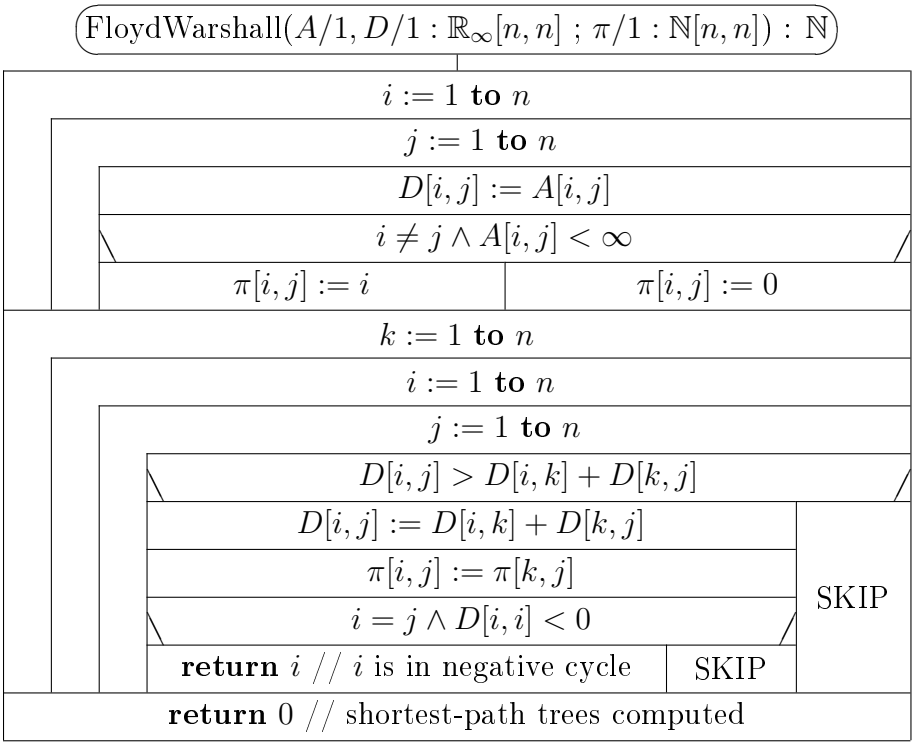


Figure 9: Algorithm FW – $MT(n) \in \Theta(n^3) \ \wedge \ mT(n) \in \Theta(n^2)$

$\wedge\ \pi[k,j] = \pi_{kj}^{(k-1)} = \pi_{kj}^{(k)}$ *according to Consequence 4.15. Performing this conditional statement* $D[i,j] = D_{ij}^{(k)} \wedge \pi[i,j] = \pi_{ij}^{(k)}$ *becomes true according to Property 4.14. (Clearly,* $D[i,i] < 0$ *becomes never true if there is no negative cycle in the graph.)*

Thus performing all the iterations of the inner double loop $D[i,j] = D_{ij}^{(k)} \wedge \pi[i,j] = \pi_{ij}^{(k)}$ *becomes true for all* $i,j \in 1..n$*. This means that* $D = D^{(k)} \wedge \pi = \pi^{(k)}$ *at the end of the actual iteration of the main loop. Therefore* $D = D^{(k-1)} \wedge \pi = \pi^{(k-1)}$ *will be true at the beginning of the next iteration for* k*.*

Consequently (by mathematical induction), $D = D^{(n)} \wedge \pi = \pi^{(n)}$ *will be true after the last iteration of the main loop, where* $k = n$*.*

Note 4.17 *If a negative value appears in the main diagonal of matrix* D *of the following program, then we have found a negative cycle. (Let the reader argue about this statement.)*

Note 4.18 *As a result of algorithm Transitive Closure,* $T[i,j] \iff D[i,j] < \infty$ *when algorithm FW has been finished* $(i,j \in 1..n)$*.*

Practically speaking, algorithm Transitive Closure solves its own task more efficiently than algorithm FW its own one, because of the simpler bodies of the loops.

4.3.1 Solving the All-Pairs Shortest Paths problem with the Single-Source Shortest Paths algorithms

Let us notice that the solution of the Floyd-Warshall (FW) algorithm, namely the ith rows of matrices D and π computed with it supply a solution of the Single-Source Shortest Paths problem for $s = i$, provided that there is no negative loop in the input graph.

Similarly, solving the Single-Source Shortest Paths problem for each $s \in 1..n$, we also receive a solution of the All-Pairs Shortest Paths problem.

Below we consider some cases.

Provided that our graph is a DAG, we can call the DAG Shortest Paths algorithm for each $s \in 1..n$, and a solution of the All-Pairs Shortest Paths problem can be computed with worst-case time complexity $MT(n, m) \in \Theta(n * (n + m))$.

For sparse graphs, where $m \in O(n)$, $\Theta(n * (n + m)) = \Theta(n^2)$, thus $MT(n, m) \in \Theta(n^2)$. This means that on sparse graphs, performing n times the DAG Shortest Paths algorithm is faster by an order of magnitude than performing the FW algorithm which runs in $MT(n), mT(n) \in \Theta(n^3)$ on DAGs.

For dense graphs, where $m \in \Theta(n^2)$, $\Theta(n * (n + m)) = \Theta(n^3)$, thus $MT(n, m) \in \Theta(n^3)$ which may cause slower run than that of FW, because of the higher constants hidden in the Θ-notation.

We receive similar results, if we apply Breadth-First Search versus FW to unweighted graphs.

Provided that our graph has no negative edge, we can call the Dijkstra algorithm for each $s \in 1..n$, and a solution of the All-Pairs Shortest Paths problem can be computed with worst-case time complexity $MT(n, m) \in O(n * (n + m) * \log n)$.

For sparse graphs, where $m \in O(n)$, $O(n*(n+m)*\log n) = O(n^2 * \log n)$, thus $MT(n, m) \in O(n^2 * \log n)$. This means that on sparse graphs, performing n times the Dijkstra algorithm is also faster by an order of magnitude than performing the FW algorithm which runs in $MT(n), mT(n) \in \Theta(n^3)$ on graphs with non-negative edges.

For dense graphs, where $m \in \Theta(n^2)$, $O(n*(n+m)*\log n) = O(n^3 * \log n)$, thus $MT(n, m) \in O(n^3 * \log n)$ which may cause slower run than that of FW, because $n^3 * \log n$ is asymptotically greater than n^3.

If we know only that the input graph contains no negative cycle, performing QBF for each $s \in 1..n$, $MT(n,m) \in O(n * n * m) = O(n^2 * m)$. Provided that our graph is **not** extremely sparse, i.e. we can suppose that $m \in \Omega(n)$, the upper estimate of the asymptotic running time of $n*$QBF is never better than the asymptotic running time of the FW algorithm which runs in $MT(n), mT(n) \in \Theta(n^3)$ on graphs with no negative cycle.

Algorithms and Data Structures II.
Lecture Notes:
String Matching, Data Compression

Ásványi Tibor – asvanyi@inf.elte.hu

August 27, 2022

Contents

References

[1] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C., Introduction to Algorithms (Fourth Edititon), *The MIT Press*, 2022

[2] Cormen, Thomas H., Algorithms Unlocked, *The MIT Press*, 2013.

[3] Ásványi, Tibor Algorithms and Data Structures I. Lecture Notes, `http://aszt.inf.elte.hu/~asvanyi/ds/AlgDs1/`, 2022

1 String Matching ([1] 32)

Given alphabet $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_d\}$ ($1 \leq d < \infty$ integer constant), text $T : \Sigma[n]$, pattern $P : \Sigma[m]$, we search for all the occurrences of $P[0\,..\,m)$ in $T[0\,..\,n)$, provided that $0 < m \leq n$. (These symbols will be used in this way in this chapter.) The elements of the alphabet will be called letters.

Definition 1.1
$s \in 0\,..\,(n-m)$ *is a valid shift of* P *on* T, **iff** $T[s\,..\,s+m) = P[0\,..\,m)$.

We will compute the set of valid shifts of P on T, i.e. set
$S = \{\, s \in 0\,..\,(n-m) \mid T[s\,..\,s+m) = P[0\,..\,m) \,\}$.

1.1 The naive string-matching (Brute-Force) algorithm

As an introduction, consider the following example. We search for pattern $P[0\,..\,4) = BABA$ in text $T[0\,..\,11) = ABABBABABAB$. (Notation: $\underline{B}$: letter B has been matched successfully against the appropriate letter of the text; $\not{B}$: it has been matched unsuccessfully.)

$i=$	0	1	2	3	4	5	6	7	8	9	10
$T[i]=$	A	B	A	B	B	A	B	A	B	A	B
	$\not{B}$	A	B	A							
		$\underline{B}$	$\underline{A}$	$\underline{B}$	$\not{A}$						
			$\not{B}$	A	B	A					
				$\underline{B}$	$\not{A}$	B	A				
$s=4$					$\underline{B}$	$\underline{A}$	$\underline{B}$	$\underline{A}$			
						$\not{B}$	A	B	A		
$s=6$							$\underline{B}$	$\underline{A}$	$\underline{B}$	$\underline{A}$	
								$\not{B}$	A	B	A

$$S = \{\, 4; 6 \,\}$$

In general, we have a window, i.e $T[s\,..\,s+m)$ on the text. The size of the window is equal to the length of the pattern. We perform the following algorithm.

(0) In the beginning, the window is at the beginning of the text, i.e. $s = 0$, and the set S of valid shifts is empty.

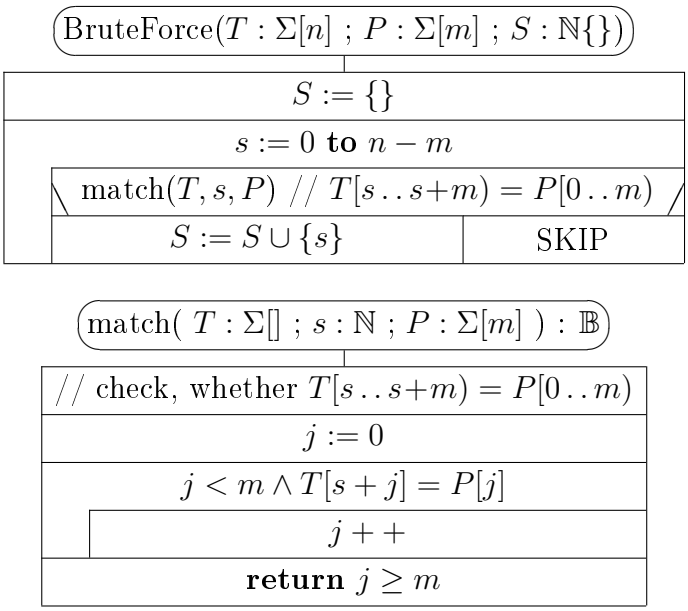
(1) We check, whether we see the pattern, i.e. $P[0\,..\,m)$ in the window.

(2) If so, we add the actual shift s of the window to the set S of valid shifts.

(3) We perform $s := s + 1$, i.e., we slide the window to the right by one.

(4) If the window is still in the text, we go to step (1).

(5) We return the set S of valid shifts.



Property 1.2 *For the naive string-matching algorithm,*
$MT(n,m) \in \Theta((n{-}m{+}1) * m) \quad \wedge \quad mT(n,m) \in \Theta(n{-}m{+}1)$

Proof. First we take some notes on the number of loop iterations + subroutine calls, to prove both statements.
– The main loop of the Brute-Force algorithm iterates $n{-}m{+}1$ times.
– The loop of equality test $T[s\,..\,s{+}m) = P[0\,..\,m)$ iterates 0 to m times.
– Considering all the iterations of the Brute-Force algorithm, this equality test makes $(n{-}m{+}1) * m$ loop iterations in the worst case (when $P[0\,..\,m)$ occurs "everywhere" in $T[0\,..\,n)$, i.e., both of them have the form "$\sigma\sigma\ldots\sigma$"), and no iteration in the best case (when $P[0] \notin T[0\,..\,n)$).
– There are $1 + (n{-}m{+}1) = n{-}m{+}2$ subroutine calls.

Based on these notes, first we prove that $MT(n,m) \in \Theta((n{-}m{+}1) * m)$.
– In the worst case, there are $(n{-}m{+}1){+}(n{-}m{+}1){*}m{+}(n{-}m{+}2)$ steps, i.e. loop iterations + subroutine calls, which means $MT(n,m) = (n{-}m{+}1){*}(m{+}2){+}1$ steps altogether, where $n \geq m$.
– Thus $MT(n,m) > (n{-}m{+}1){*}m$. Therefore $MT(n,m) \in \Omega((n{-}m{+}1){*}m)$.
– In order to prove $MT(n,m) \in O((n{-}m{+}1){*}m)$, we can solve the following

inequality.
$(n-m+1)*(m+2)+1 \leq 2*(n-m+1)*m$
$(n-m+1)*m+(n-m+1)*2+1 \leq 2*(n-m+1)*m$
$(n-m+1)*2+1 \leq (n-m+1)*m$
$1 \leq (n-m+1)*(m-2)$
And this is true if $m \geq 3$. (In this case, $m-2 \geq 1$. Thus $(n-m+1)*(m-2) \geq (n-m+1)*1 \geq 1$ because $n \geq m$ in general in this topic.)

Finally we prove that $mT(n,m) \in \Theta(n-m+1)$.
– In the best case, there are $(n-m+1)+(n-m+2)$ steps, i.e. loop iterations + subroutine calls, which means $mT(n,m) = 2*(n-m+1)+1$ steps altogether, where $n \geq m$.
– Thus $mT(n,m) > (n-m+1)$. Consequently $mT(n,m) \in \Omega(n-m+1)$.
– In order to prove $mT(n,m) \in O(n-m+1)$, we can solve the following inequality.
$2*(n-m+1)+1 \leq 3*(n-m+1)$
$1 \leq n-m+1$
$0 \leq n-m$
$m \leq n$ which is true in general in this topic. □

Property 1.3 *Provided that on a class of pattern matching problems there is some constant $c \in (0;1)$ so that $m \leq c*n$, we have the following asymptotic efficiency for the naive algorithm above.*

$$MT(n,m) \in \Theta(n*m) \quad \wedge \quad mT(n,m) \in \Theta(n)$$

Proof. $1*n \geq n-m+1 > n-c*n = (1-c)*n$ where $1-c \in (0;1)$ constant.
Based on the definition of $\Theta(\cdot)$, $n-m+1 \in \Theta(n)$.
Therefore $(n-m+1)*m \in \Theta(n*m)$.
Considering Property 1.2 and the transitivity of relation $\cdot \in \Theta(\cdot)$, we have
$0 < c < 1 \wedge m \leq c*n \Rightarrow mT(n,m) \in \Theta(n) \wedge MT(n,m) \in \Theta(n*m)$. □

Property 1.4 *Provided that on a class of pattern matching problems there are some constants $0 < \varepsilon \leq c < 1$ so that $\varepsilon*n \leq m \leq c*n$, we have the following worst-case asymptotic efficiency for the naive algorithm above.*

$$MT(n,m) \in \Theta(n^2)$$

Proof. Clearly $\varepsilon*n*n \leq n*m \leq n*n$. Thus $n*m \in \Theta(n^2)$. Considering $MT(n,m) \in \Theta(n*m)$ from Property 1.3, and the transitivity of relation $\cdot \in \Theta(\cdot)$, we have $MT(n,m) \in \Theta(n^2)$ □

1.2 Quick Search

Quick Search is a simplified version of the Boyer-Moore algorithm. Boyer-Moore and its variants are considered extremely efficient string-matching algorithms in usual applications. Quick Search (or some other variant of Boyer-Moore) is often implemented in text editors for the *search* and *substitute* commands.

In Quick Search, similarly to the naive string-matching algorithm, we have a window ($T[s\,..\,s+m)$) with the size of the pattern ($P[0\,..\,m)$). We start with $s = 0$, i.e., we start with the window at the beginning of the text, and we make the $T[s\,..\,s+m) = P[0\,..\,m)$ comparisons repeatedly, i.e., we check repeatedly whether we see the pattern in the window. Between two comparisons/checks we slide the window to the right. The naive method always slides the window by one, i.e., it increases the actual shift of the window by one. The speedup of Quick Search (and of many other efficient string-matching algorithms) comes from a typically greater increase of shift s. However, we must ensure that while sliding the window to right, we do not jump over any valid shift, i.e., we find each substring of $T[0\,..\,n)$ which matches $P[0\,..\,m)$.

In these efficient string-matching algorithms, we typically make some preparations before we start the actual search: Based (only) on the pattern $P[0\,..\,m)$, we generate a table. And from this table, after a successful or unsuccessful matching, we can determine in $\Theta(1)$ time, how to go on.

In the case of Quick Search, in this preparation phase, we consider each element of the alphabet Σ. We add a label $shift(\sigma) \in 1\,..\,m+1$ to each $\sigma \in \Sigma$.

Let us suppose that $T[s\,..\,s+m)$ (the window) has just been matched against $P[0\,..\,m)$; $over := s + m$; $\sigma := T[over]$. This means that $T[over]$ is just over the actual window. Then the $shift(\sigma)$ value shows how much the window should be moved (to the right) above the text so that we have some chance to see the pattern through the window. To decide about the chance, we consider only this $\sigma = T[over]$ character of the text.

We have two cases.

1. Provided that $\sigma \in P[0\,..\,m)$, $shift(\sigma) \in 1\,..\,m$ shows how much the window should be moved above the text, so that the old $T[over]$ can be seen as the rightmost occurrence of σ in $P[0\,..\,m)$. This rightmost occurrence of σ in $P[0\,..\,m)$ corresponds to the smallest movement of the window. (The other occurrences of σ in $P[0\,..\,m)$ corresponds to bigger movements of the window.)

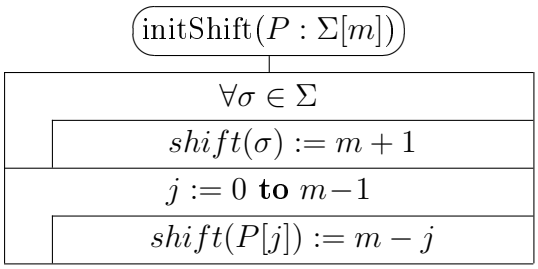
2. Provided that $\sigma \notin P[0\,..\,m)$, $shift(\sigma) = m + 1$, i.e., the window jumps over this σ character of the text. (With a smaller movement of the window, we have no chance to see the pattern through it.)

For example, let the alphabet be Σ = {A,B,C,D}, and let the pattern be $P[0\,..\,4)$ = CADA. In the following examples, xxxx shows the window's position before sliding it to the right, and the pattern CADA displays the window's position after sliding it to the right.

```
   Text: ...xxxxA......xxxxB......xxxxC......xxxxD...
Pattern:      CADA            CADA       CADA      CADA
```

The appropriate $shift$ values are given in the following table.

σ	A	B	C	D
$shift(\sigma)$	1	5	4	2



Considering the size of the alphabet as a constant, we receive

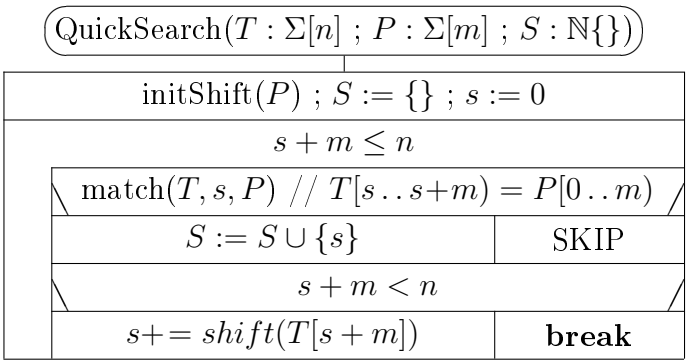
$$T_{\text{initShift}}(m) \in \Theta(m).$$

With the previous pattern $P[0\,..\,4) = CADA$, the illustration of initShift() and that of Quick Search follows.

σ	A	B	C	D
initial $shift(\sigma)$	5	5	5	5
C			4	
A	3			
D				2
A	1			
final $shift(\sigma)$	1	5	4	2

$i=$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
$T[i]=$	A	D	A	B	A	B	C	A	D	A	B	C	A	B	A	D	A	C	A	D	A	D	A
	$\not C$	A	D	A																			
		$\not C$	A	D	A																		
$s=6$							$\underline{C}$	$\underline{A}$	$\underline{D}$	$\underline{A}$													
												$\underline{C}$	$\underline{A}$	$\not D$	A								
														$\not C$	A	D	A						
$s=17$																		$\underline{C}$	$\underline{A}$	$\underline{D}$	$\underline{A}$		
																				$\not C$	A	D	A

$$S = \{\, 6; 17 \,\}$$



$mT(n,m) \in \Theta\left(\frac{n}{m+1} + m\right)$ (e.g. if $T[0\,..\,n)$ and $P[0\,..\,m)$ are disjunct)
$MT(n,m) \in \Theta((n - m + 1) * m)$ (e.g. if $T = \sigma\sigma\ldots\sigma$ és $P = \sigma\ldots\sigma$)

The best-case performance of Quick Search is an order of magnitude better than that of the naive string-matching algorithm. The worst-case performance is a bit worse than that of Brute-Force because of the running time of initShift(P), although this does not influence the asymptotic measure.

Fortunately, according to experimental studies, the average performance is much closer to the best case than to the worst case. As a result, in many practical applications, Quick Search is one of the best choices. However, if we want to optimize for the worst case, we need another algorithm, for example, Knuth-Morris-Pratt.

1.3 String Matching in Linear time (Knuth-Morris-Pratt, i.e. KMP algorithm)

As an introduction, consider the following example. We search for the occurrences of pattern $P[0\,..\,8) = BABABBAB$ in text a $T[0\,..\,18) = ABABABABBABABABBAB$. (Notation: The algorithm knows "even

without matching" that the unmarked letters at the beginning of the text are the same as the corresponding letters in the text. $\underline{B}$: letter B has been matched successfully against the appropriate letter of the text; a $\not{B}$: it has been matched unsuccessfully. [Listen to the explanation at the lecture.])

$i=$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$T[i]=$	A	B	A	B	A	B	A	B	B	A	B	A	B	A	B	B	A	B
	$\not{B}$																	
		$\underline{B}$	$\underline{A}$	$\underline{B}$	$\underline{A}$	$\underline{B}$	$\not{B}$											
$s=3$				B	A	B	$\underline{A}$	$\underline{B}$	$\underline{B}$	$\underline{A}$	$\underline{B}$							
									B	A	B	$\underline{A}$	$\underline{B}$	$\not{B}$				
$s=10$											B	A	B	$\underline{A}$	$\underline{B}$	$\underline{B}$	$\underline{A}$	$\underline{B}$
																B	A	B

$$S = \{\, 3; 10 \,\}$$

Notations 1.5

- *Let ε denote the empty string.*
- *If x and y are two strings, then $x + y$ is their concatenation. (For example, if $x = ABA$ and $y = BA$, then $x + y = ABABA$ and $y + x = BAABA$. If $y = \varepsilon$, then $x + y = x = y + x$.)*
- *If x and y are two strings, then $x \sqsubseteq y$ (x is* possibly full prefix of y*) means that $\exists z$ string, so that $x + z = y$. (For example, if $y = BABA$, then its possibly full prefixes are $\{\varepsilon, B, BA, BAB, BABA\}$.)*
- *If x and y are two strings, then $x \sqsubset y$ (x is* prefix of y*) means that $x \sqsubseteq y \wedge x \neq y$. (For example, if $y = BABA$, then its prefixes are $\{\varepsilon, B, BA, BAB\}$.)*
- *If x and y are two strings, then $x \sqsupseteq y$ (x is* possibly full suffix of y*) means that $\exists z$ string, so that $z + x = y$. (For example, if $y = BABA$, then its possibly full suffixes are $\{BABA, ABA, BA, A, \varepsilon\}$.)*
- *If x and y are two strings, then $x \sqsupset y$ (x is* suffix of y*) means that $x \sqsupseteq y \wedge x \neq y$. (For example, if $y = BABA$, then its suffixes are $\{ABA, BA, A, \varepsilon\}$.)*
- *$P_j = P[0\,..\,j)$ $(j \in 0\,..\,m)$ string P_j with length j is a possibly full prefix of string P. P_0 is the empty prefix of P. Similarly $T_i = T[0\,..\,i)$. [Similarly $P_0 \sqsupset P_j \quad (j \in 1\,..\,m)$.]*

- $x \sqsupset y$ (x *is* prefix-suffix *of* y) *means that* $x \sqsubset y \wedge x \sqsupset y$. *(For example, if* $y = BABA$, *then its prefix-suffixes are* $\{\varepsilon, BA\}$. *If* $y = BABAB$, *then its prefix-suffixes are* $\{\varepsilon, B, BAB\}$. *If* $y = ABC$, *its only prefix-suffix is* ε.*)*

- $\max_i H$ *is the* i*th greatest element of set* $H \quad (i \in 1\,..\,|H|)$.
  [*Consequently* $\max_1 H = \max H$. *Provided that set* H *is finite,* $\max_{|H|} H = \min H$.]

- $H(j) = \{\, h \in 0\,..\,j-1 \mid P_h \sqsupset P_j \,\} \quad (j \in 1\,..\,m)$
  (For example, if $P_5 = BABAB$, *then* $H(5) = \{0,1,3\}, H(4) = \{0,2\}, H(3) = \{0,1\}, H(2) = \{0\} = H(1)$.*)*
  [$0 \in H(j)$, $\max_1 H(j) = \max H(j)$, $\max_{|H(j)|} H(j) = \min H(j) = 0$.]
  [*Equivalent definition:* $H(j) = \{\, |x| : x \sqsupset P_j \,\} \quad (j \in 1\,..\,m)$]

- $next(j) = \max H(j) \quad (j \in 1\,..\,m)$
  (In the previous example, $next(5) = 3$. *And* $next(1) = 0$ *in general.)*

Properties 1.6 *(Similarly for prefixes)*
$x \sqsupseteq y \wedge y \sqsupseteq z \Rightarrow x \sqsupseteq z \qquad x \sqsupset y \wedge y \sqsupset z \Rightarrow x \sqsupset z$
$x \sqsupseteq y \wedge y \sqsupset z \Rightarrow x \sqsupset z \qquad x \sqsupset y \wedge y \sqsupseteq z \Rightarrow x \sqsupset z$

Property 1.7 *(Similarly for prefixes)* $x \sqsupset z \wedge y \sqsupset z \wedge |x| < |y| \Rightarrow x \sqsupset y$

Property 1.8 *Provided that* $i, j \in 0\,..\,m, \quad P_i \sqsupset P_j \iff P_i \sqsupset P_j$.

Property 1.9 *Provided that* $0 \le h < j \le m$ *and* $P_j \sqsupseteq T_i$,
$P_h \sqsupset T_i \iff P_h \sqsupset P_j$.

Properties 1.10
$P_h \sqsupseteq T_i \wedge P[h] = T[i] \iff P_{h+1} \sqsupseteq T_{i+1}$
$P_h \sqsupset P_j \wedge P[h] = P[j] \iff P_{h+1} \sqsupset P_{j+1}$
$P_h \sqsupset P_j \wedge P[h] = P[j] \iff P_{h+1} \sqsupset P_{j+1}$

Property 1.11 $next(j) \in 0\,..\,(j-1) \quad (j \in 1\,..\,m)$

Property 1.12 $next(j+1) \le next(j) + 1 \quad (j \in 1\,..\,m-1)$

Example 1.13

$P[j-1] =$	B	A	B	A	B	B	A	B
$j =$	1	2	3	4	5	6	7	8
$next(j) =$	0	0	1	2	3	1	2	3

Proof. [of Property 1.12: $next(j+1) \le next(j) + 1 \quad (j \in 1\,..\,m-1)$]

- If $next(j+1) = 0$, then $next(j+1) = 0 \leq 1 \leq next(j) + 1$.
- If $next(j+1) > 0$, consider $next(j+1) = \max H(j+1)$. Clearly, $next(j+1) \in H(j+1)$. Because $H(j+1) = \{\, h \in 0\,..\,j \mid P_h \sqsupset P_{j+1} \,\}$ we have $P_{(next(j+1)-1)+1} = P_{next(j+1)} \sqsupset P_{j+1}$. Based on property 1.10, $P_{next(j+1)-1} \sqsupset P_j$. Considering $next(j) = \max\{\, h \in 0\,..\,j-1 \mid P_h \sqsupset P_j \,\}$ we receive $next(j+1) - 1 \leq next(j)$, and $next(j+1) \leq next(j) + 1$.

□

Lemma 1.14 $next(max_l H(j)) \in H(j) \quad (j \in 1\,..\,m, l \in 1\,..\,|H(j)|-1)$

Proof.
$P_{next(\max_l H(j))} \sqsupset P_{\max_l H(j)}$ because $P_{next(i)} \sqsupset P_i \quad (i \in 1\,..\,m)$
$P_{\max_l H(j)} \sqsupset P_j$. Considering the transitivity of relation $\sqsupset$ (1.6) we receive $P_{next(\max_l H(j))} \sqsupset P_j$. As a result, $next(max_l H(j)) \in H(j)$. □

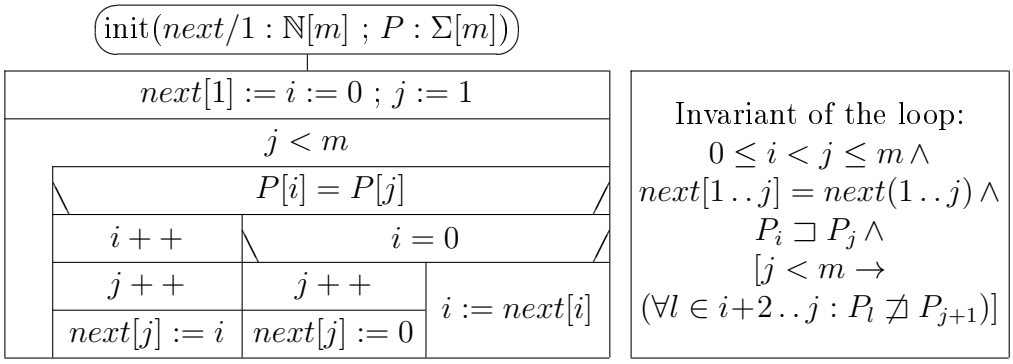
Property 1.15
$\max_{l+1} H(j) = next(\max_l H(j)) \quad (j \in 1\,..\,m, l \in 1\,..\,|H(j)|-1)$

Proof.

- First we prove that $\max_{l+1} H(j) \leq next(\max_l H(j))$.
  Clearly $P_{\max_{l+1} H(j)} \sqsupset P_j \wedge P_{\max_l H(j)} \sqsupset P_j \wedge \max_{l+1} H(j) < \max_l H(j)$. Considering 1.7, i.e. $x \sqsupset z \wedge y \sqsupset z \wedge |x| < |y| \Rightarrow x \sqsupset y$, we have $P_{\max_{l+1} H(j)} \sqsupset P_{\max_l H(j)}$. Thus $P_{\max_{l+1} H(j)} \sqsupset P_{\max_l H(j)}$. Considering the definition of function $next$, $P_{next(\max_l H(j))} \sqsupset P_{\max_l H(j)}$, and this is the longest one among the strings with this property. Consequently $\max_{l+1} H(j) \leq next(\max_l H(j))$.
- On the other hand, $next(\max_l H(j)) \leq \max_{l+1} H(j)$, because from the definition of function $next$, $next(\max_l H(j)) < \max_l H(j)$, and according to Lemma 1.14, $next(\max_l H(j)) \in H(j)$.

□

Now we define array $next/1 : \mathbb{N}[n]$. The following procedure, $\mathrm{init}(next, P)$ initializes it so that it contains the values of function $next$, i.e., $\forall j \in 1\,..\,m : next[j] = next(j)$, or in short: $next[1\,..\,m] = next(1\,..\,m)$. Procedure $\mathrm{init}(next, P)$ fills array $next/1 : \mathbb{N}[n]$ based on array $P : \Sigma[m]$.



Exercise 1.16 *Prove that the invariant of the loop of procedure* init$(next, P)$ *is correct. Why does this invariant imply that* $next[1\,..\,m] = next(1\,..\,m)$ *is satisfied when the procedure returns?*

Property 1.17 *For* init$(next/1{:}\mathbb{N}[m]; P{:}\Sigma[m])$: $MT(m), mT(m) \in \Theta(m)$.

Proof. It is enough to see that the loop of procedure init$(next, P)$ iterates minimum $m{-}1$ times, and maximum $2m{-}2$ times. Remember that according to the invariant of this loop $0 \leq i < j \leq m$.

- Before the first iteration $j = 1$, and each iteration increases j at most by one. Thus it needs at least $m{-}1$ iteration to achieve $j = m$, to finish the loop and the procedure as well.
- $t(i, j) := 2j - i$. Clearly, $0 \leq i < j \leq m$ implies $t(i, j) \in 2\,..\,2m$. Before the first iteration $t(i, j) = 2$. And $t(i, j)$ strictly increases with each iteration (on any branch of the core of the loop). Thus the loop stops after at most $2m{-}2$ iterations.

□

The illustration of procedure init$(next, P)$ on pattern $ABABBABA$:
(We start a new line at the beginning of a branch of the loop.)

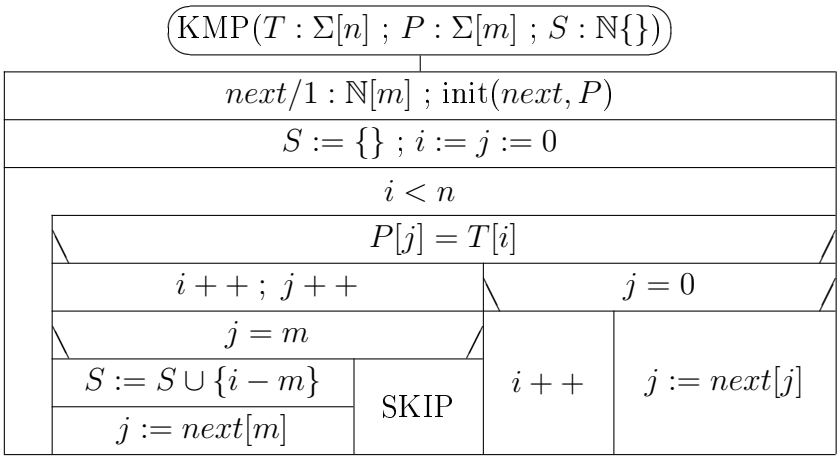
i	j	$next[j]$	$\overset{0}{A}$	$\overset{1}{B}$	$\overset{2}{A}$	$\overset{3}{B}$	$\overset{4}{B}$	$\overset{5}{A}$	$\overset{6}{B}$	$\overset{7}{A}$
0	1	0		~~A~~						
0	2	0			$\underline{A}$					
1	3	1			A	$\underline{B}$				
2	4	2			A	B	~~A~~			
0	4	2					~~A~~			
0	5	0						$\underline{A}$		
1	6	1						A	$\underline{B}$	
2	7	2						A	B	$\underline{A}$
3	8	3								

The result:

$P[j-1] =$	A	B	A	B	B	A	B	A
$j =$	1	2	3	4	5	6	7	8
$next[j] =$	0	0	1	2	0	1	2	3

Based on the properties of prefixes and suffixes which we have seen above, the algorithm Knuth-Morris-Pratt (KMP) solves the string matching problem in linear, i.e. $\Theta(n)$ time where n is the length of the text.

First we initialize array $next/1 : \mathbb{N}[m]$ with procedure $\text{init}(next, P)$. Next, with the help of this, we determine the valid shifts of the pattern:



Before discussing KMP in general, we explain this procedure through the following example. We search for the occurrences, i.e. valid shifts of pattern $P[0\,..\,8) = ABABBABA$ (see the computation of its $next$ array above) in text $T[0\,..\,17) = ABABABBABABBABABA$.

$P[j-1] =$	A	B	A	B	B	A	B	A
$j =$	1	2	3	4	5	6	7	8
$next[j] =$	0	0	1	2	0	1	2	3

The search:

$i =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$T[i] =$	A	B	A	B	A	B	B	A	B	A	B	B	A	B	A	B	A
	$\underline{A}$	$\underline{B}$	$\underline{A}$	$\underline{B}$	$\not{B}$												
$s=2$			A	B	$\underline{A}$	$\underline{B}$	$\underline{B}$	$\underline{A}$	$\underline{B}$	$\underline{A}$							
$s=7$								A	B	A	$\underline{B}$	$\underline{B}$	$\underline{A}$	$\underline{B}$	$\underline{A}$		
													A	B	A	$\underline{B}$	$\not{B}$
															A	B	$\underline{A}$

$$S = \{\, 2; 7 \,\}$$

$s=0$ We start with $s=0$. We find that $P[0]=T[0],\ldots,P[3]=T[3]$, but $P[4]\neq T[4]$. The longest prefix of $P_8=P[0\,..\,8)$ which matches the beginning of the text is $P_4=P[0\,..\,4)$. The problem: where to shift P so that we have a chance to find a valid shift, but we do not jump over a solution. $P_4=T[0\,..\,4)$. Therefore, if we find a prefix of P_4 which is also a suffix of it, it will also be a suffix of $T[0\,..\,4)$. We know that $next[4]=2$, i.e., $P_2 \sqsupset P_4$, and this is the longest one.

$s=2$ – If we shift P, so that $P[0\,..\,2)$ is under initial position of $P[2\,..\,4)$ which is equal to $T[2\,..\,4)$, then $P[0\,..\,2)$ automatically matches with $T[2\,..\,4)$, and we can go on with comparisons $P[2]=T[4], P[3]=T[5],\ldots,P[7]=T[9]$. We have found a valid shift ($s=2$) here.
– One may say that also $P_0 \sqsupset P_4$, and we could go on with comparison $P[0]=T[4]$, but in this case, we would have jumped over a solution. The smallest shift corresponds to the longest prefix-suffix, and we have to use it, to avoid jumping over a possible solution.

Therefore we selected the longest prefix-suffix of P_4, went on with $P[2]=T[4]\ldots$ and finally found valid shift $s=2$. (Note that the length of the longest prefix-suffix is most often different from the valid shift found.) Now we find that $next[8]=3$ which means that the longest prefix-suffix of P_8 is P_3. Thus we shift P so that $P[0\,..\,3)$ is under the end of the previous position of $P[0\,..\,8)=T[2\,..\,10)$ because this shift corresponds to the smallest one where we have chance to find an occurrence of the pattern.

$s=7$ Again, automatically $P_3 \sqsupset T_{10}$. Thus we go on with checking $P[3]=T[10]$, and so on. We are lucky again because we find another valid shift $s=7$. Again we find that $next[8]=3$ which means that the longest prefix-suffix of P_8 is P_3. Thus we shift P so that $P[0\,..\,3)$ is under the end of the previous position of $P[0\,..\,8)=T[7\,..\,15)$ because this shift corresponds to the smallest one where we have chance to find an occurrence of the pattern.

$s=12$ Now the end of the pattern is over the text, but the KMP algorithm does not check this (to reduce the overall running time despite a longer "end of the game"). Still it finds $P[3]=T[15]$ and $P[4]\neq T[16]$. We have found that $P_4 \sqsupset T_{16}$. Because $next[4]=2$ which means that P_2 is the longest prefix-suffix of P_4, we can shift pattern P so that P_2 is under the previous position of $P[2\,..\,4)=T[14\,..\,16)$.

$s=14$ We have found that $P_2 \sqsupset T_{16}$. Thus we check $P[2]=T[16]$. This is true, but with the next check, i.e. $P[3]=T[17]$ we would over-index

the text, so we stop.

After all, we received that the set of valid shifts is $S = \{\, 2; 7 \,\}$.

In general, we search for the valid shifts of P_m in T_n. We can start from P_0 and T_0 because $P_0 \sqsupseteq T_0$. In general, we have $P_j \sqsupseteq T_i$ for some i and j.

When the KMP algorithm finds that $P_j \sqsupseteq T_i \wedge j = m$, then it has found a valid shift which is $s = i - m$. When it finds that $P_j \sqsupseteq T_i \wedge j < m \wedge P[j] \neq T[i]$, then it has not found a valid shift.

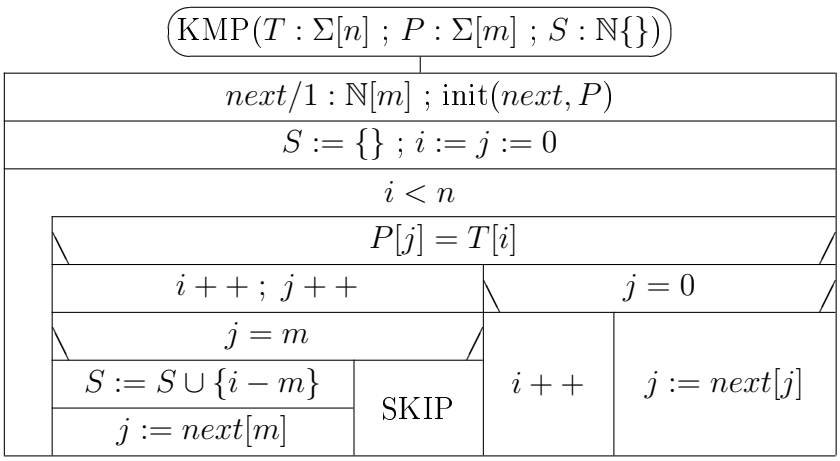
But in both cases – provided that $j > 0$ –, in order to find a new valid shift, it determines the longest prefix-suffix of P_j which is $P_{next[j]}$. This is useful because $P_{next[j]} \sqsupseteq T_i$ and moving $P_{next[j]}$ under the end of T_i, this is the smallest shift with which we have chance to find a new valid shift. To make this movement of the pattern, we perform the assignment statement $j := next[j]$, and then $P_j \sqsupseteq T_i$ is satisfied again.

Provided that $i < n$, we can go on with checking $P[j] = T[i]$. (1) If $P[j] = T[i]$, then $P_{j+1} \sqsupseteq T_{i+1}$, and we increment i and j. Then $P_j \sqsupseteq T_i$ is satisfied again, and so on. (2) If $P[j] \neq T[i]$, again we have to determine the longest prefix-suffix of P_j, and so on.

Still there is the case when $j = 0 \wedge P[j] \neq T[i]$. Then we make the minimal possible shift, i.e., we increment i and go on.

The whole process can continue while $i < n$ which means that $T[i]$ exists.

Now we verify that $MT(n), mT(n) \in \Theta(n)$ for procedure KMP().



Property 1.18 *A trivial invariant of the loop of algorithm KMP:*

$$i \in 0\,..\,n \wedge j \in [0\,..\,m) \wedge j \leq i$$

To prove that the time complexity of procedure KMP() is linear, it is enough to verify that the running time of its main loop is $\Theta(n)$, because $m \in 1\,..\,n$, consequently the $\Theta(m)$ time needed for the init$(next, P)$ call and other initialization do not modify this asymptotic order. And to verify the $\Theta(n)$ operational complexity of this main loop, it is sufficient to prove that it performs at least n and at most $2n$ iterations. For this purpose, we can use invariant 1.18: $i \in 0\,..\,n \wedge j \in 0\,..\,(m-1) \wedge j \leq i$.

- Before the first iteration, $i = 0$. Because each iteration increases i by at most one, and the condition of the loop is $i < n$, we need at least n iterations to finish the loop.

- To prove the upper bound $2n$ of the number of iterations, $t(i, j) := 2i - j$. Based on invariant $i \in 0\,..\,n \wedge j \in 0\,..\,(m-1) \wedge j \leq i$, we have $t(i, j) \in 0\,..\,2n$. Before the first iteration, $t(i, j) = 0$. Because $t(i, j) = 2i - j$ strictly increases on each of the four branches of the body of the loop, and $t(i, j) \leq 2n$, the loop stops after at most $2n$ iterations.

One can see that variable i never decreases during the run of procedure KMP() (i.e., we never backtrack on the text when we search it for the pattern). Consequently, the algorithm Knuth-Morris-Pratt can be implemented easily, even if this text is in a sequential file.

This is not the case with the Brute-force and Quick Search algorithms. In these algorithms, sometimes we have to backtrack even $m-2$ characters on the text. Provided that the text is in a sequential file, this means that during the run of the implementations of these algorithms, the last $m-1$ characters of the text must be stored in a buffer.

Algorithms and Data Structures II. Lecture Notes: Trees

Tibor Ásványi
Department of Computer Science
Eötvös Loránd University, Budapest
asvanyi@inf.elte.hu

August 27, 2021

Contents

References

[1] Ásványi, T, Algorithms and Data Structures I. Lecture Notes
http://aszt.inf.elte.hu/~asvanyi/ds/AlgDs1/AlgDs1LectureNotes.pdf

[2] Burch, Carl, B+ trees
(See http://aszt.inf.elte.hu/~asvanyi/ds/AlgDs2/B+trees.pdf)

[3] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.,
Introduction to Algorithms (Third Edititon), *The MIT Press*, 2009.

[4] Cormen, Thomas H., Algorithms Unlocked, *The MIT Press*, 2013.

[5] Narashima Karumanchi,
Data Structures and Algorithms Made Easy, *CareerMonk Publication*, 2016.

[6] Neapolitan, Richard E., Foundations of algorithms (Fifth edition),
Jones & Bartlett Learning, 2015. ISBN 978-1-284-04919-0 (pbk.)

[7] Shaffer, Clifford A.,
A Practical Introduction to Data Structures and Algorithm Analysis,
Edition 3.1 (C++ Version), 2011
(See http://aszt.inf.elte.hu/~asvanyi/ds/C++3e20110103.pdf)

[8] Weiss, Mark Allen, Data Structures and Algorithm Analysis in C++
(Fourth Edition),
Pearson, 2014.

[9] Wirth, N., Algorithms and Data Structures,
Prentice-Hall Inc., 1976, 1985, 2004.
(See http://aszt.inf.elte.hu/~asvanyi/ds/AD.pdf)

1 Introduction

We need *dictionaries*, i.e. large sets of data with efficient insertion, search and deletion operations (where removing the maximal or minimal element of the dictionary and removing an element with a given key are the cases of deletion). We already have two solutions where the average performance (AT) of these operation is good but the worst case performance (MT) is too slow (n is the size of the set / data structure):

- Binary search trees (BSTs): $AT(n) \in \Theta(\log n), MT(n) \in \Theta(n)$.
- Hash tables: $AT(n) \in \Theta(1)$ (if the load factor is not too high), $MT(n) \in \Theta(n)$. (Removing the maximal and minimal elements is not supported.)

We are going to discuss some kinds of *balanced search trees* where we can guarantee $MT(n) \in \Theta(\log n)$ for each dictionary operation.

- *AVL trees*: balanced BSTs for storing data in the central memory.[1]
- *B+ trees*: perfectly balanced multiway search trees optimized for storing data on hard disks.[2]

In the next section we consider *AVL trees*. In the last one we describe *B+ trees*.

2 AVL Trees

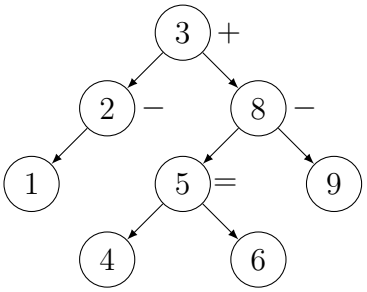
Being BSTs, AVL trees have linked representation typically. In these notes, we suppose that they have linked representation. Everywhere letter n denotes the size (number of nodes) of the tree, and h denotes the height of the tree.

Definition 2.1 *The* balance *of node* $(*p)$ *of a binary tree is* $p \to b = h(p \to right) - h(p \to left)$. *This means that the* balance *of a node is the height of its right subtree* minus *the height of its left subtree.*[3]

[1] *Red-black trees* are also specially balanced BSTs and can provide another efficient solution.

[2] *B trees* are an older solution.

[3] The *balance* notion above is sometimes called *height-balance*, in order to distinguish it from other kinds of *balances*. For example, the *size-balance* of a node is the size of its right subtree *minus* the size of its left subtree. A node is *size-balanced*, **iff** its *size-balance* is in $\{-1, 0, 1\}$. A tree is *size-balanced*, **iff** all of its nodes are *size-balanced*. In these notes, *balance* means *height-balance* by default.



The binary tree on the left in

- simple textual form:
  (((1) 2−) 3+ (((4) 5= (6)) 8− (9)))
- elegant parenthesized form:
  { [(1) 2−] 3+ [(⟨4⟩ 5= ⟨6⟩) 8− (9)] }

Figure 1: The same binary tree in graphical and textual representations. The balance of each leaf is zero, so we omit the balances of the leaves. We show the balance of each internal node next to it (or next to the key identifying it). In the example trees, we use the following notation for the balances: $0 \sim =$; $1 \sim +$; $2 \sim ++$; $-1 \sim -$; $-2 \sim --$

Example 2.2 *See the BST on* Figure 1. *Deleting node* ① *from it, we receive BST* { [2] 3++ [(⟨4⟩ 5= ⟨6⟩) 8− (9)] }. *And inserting node* ⑦ *into the BST on* Figure 1, *we receive the following BST.*
{ [(1) 2−] 3++ [(⟨4⟩ 5+ ⟨ 6+ {7} ⟩) 8−− (9)] }.

Exercise 2.3 ***Draw each tree which is given in textual representation in these lecture notes. Drawing them provides easier and deeper understanding, and better long-term knowledge. One can see a pattern above, at*** **Figure 1.**

Definition 2.4 *Node* $(*p)$ *is* perfectly balanced, **iff** *its balance,* $p \to b = 0$.[4]

Definition 2.5 *Node* $(*p)$ *is* balanced, **iff** *its balance, i.e.* $p \to b \in \{-1, 0, 1\}$. *Otherwise it is* imbalanced.

Consequence 2.6 *The leaves of a binary tree are* perfectly balanced, *because both of the right and left subtrees of a leaf are empty.*

Definition 2.7 *A binary tree is* balanced, **iff** *each (internal) node of it is balanced. It is* imbalanced, **iff** *at least one of its nodes is imbalanced.*

Definition 2.8 *A binary tree is* perfectly balanced, **iff** *each (internal) node of it is perfectly balanced.*

Consequence 2.9 *A binary tree is* perfectly balanced, **iff** *it is* complete.

[4] **iff** means **if and only if**.

Definition 2.10 *An* AVL tree *is a* balanced BST.[5]

Remember that $MT(h) \in \Theta(h)$ for the basic operations of BSTs (insertion, search and deletion). Fortunately, the BST and the balanced tree properties can be maintained efficiently throughout insertions and deletions on balanced BSTs, and the next theorem guarantees that the height of an AVL tree never goes far form the ideal case. Thus we can guarantee the high efficiency of these operations on AVL trees.

Theorem 2.11 *Given a nonempty balanced binary tree,*

$$\lfloor \log n \rfloor \leq h \leq 1.45 \log n, \quad \text{i.e.} \quad h \in \Theta(\log n)$$

We provide an outline of the proof of this theorem. The lower bound given there is the lower bound of the height of all the binary trees, thus it is also true for balanced binary trees. The upper bound given here follows from the properties of *Fibonacci trees*.

Definition 2.12 *A binary tree is* Fibonacci tree, **iff** *each* internal *node of it is balanced, but not perfectly balanced.*

These are called *Fibonacci trees*, because a Fibonacci tree with height $h > 0$ consists of a root *plus* a left subtree with height $h-1$ *and* a right subtree with height $h-2$ or vice versa where both subtrees are Fibonacci trees. And this recursive description is similar to the recursive definition of *Fibonacci numbers*.

It is easy to prove that among the balanced binary trees with a given height, the Fibonacci trees have the smallest size. Let f_h be the size of a nonempty Fibonacci tree with height h. Clearly
$f_0 = 1$, $f_1 = 2$, $f_h = 1 + f_{h-1} + f_{h-2} \quad (h \geq 2)$.

[5]It is easy to prove that the *size-balanced binary trees* are special cases of the *nearly complete binary trees*. And we know that the height h of a nearly complete binary tree is minimal among the binary trees of a given size n, i.e. $h = \lfloor \log n \rfloor$, plus $MT(h) \in \Theta(h)$ for the basic operations of BSTs (insertion, search and deletion), consequently these operations are most efficient on nearly complete and especially size-balanced BSTs. Thus one may ask, why we do not use *size-balanced* or *nearly complete* BSTs instead of AVL trees. Well, the *size-balanced* and the *nearly complete* properties turn out too strong requirements in most cases, because we cannot maintain these properties efficiently throughout insertions and deletions on BSTs. Therefore these properties are often lost throughout insertions and deletions, and the height of a resulting BST may go far from the ideal case. Thus we use AVL trees, because the AVL tree property can be maintained extremely efficiently throughout insertions and deletions, and the height of an AVL tree is quite close to the minimal height $h = \lfloor \log n \rfloor$ even in the worst case.

These formulas are similar to the definition of the Fibonacci sequence: $F_0 = 0$, $F_1 = 1$, $F_h = F_{h-1} + F_{h-2} \quad (h \geq 2)$.

It can be proved with mathematical induction that $f_h = F_{h+3} - 1$ where h is a natural number. Clearly $f_h \leq n$ where n is the size of a balanced binary tree with height h. With some mathematical skill, we can get the upper bound given in the previous theorem. (We omit the details.)

From the previous results, it can be proved that $MT(n) \in \Theta(\log n)$ for the basic operations of AVL trees (insertion, search and deletion). In order to see it, still we have to see that $MT(h) \in \Theta(h)$ even for the AVL tree version of these operations where each operation keeps the AVL tree property.

In order to achieve this aim, first we describe our representation of the nodes of AVL trees. Then we develop the algorithms of the basic operations of the AVL trees. It will be clear that these operations go down and possibly up only once in the tree, and the number of elementary operations can be limited with the same constant at each level, so $MT(h) \in \Theta(h)$ is true. See the representation first.

Node
+ $key : \mathcal{T}$ // $\mathcal{T}$ is some known type + $b : -1..1$ // the balance of the node + $left, right$: Node*
+ Node() { $left := right := \oslash$; $b := 0$ } // create a tree of a single node + Node(x:$\mathcal{T}$) { $left := right := \oslash$; $b := 0$; $key := x$ }

We can see that the balance of a node is stored explicitly in the node. When a tree is modified, clearly the balances of some nodes change. Thus we have to adjust the data members b of these nodes. This process will be called *rebalancing* in these notes. When we illustrate an operation on a BST, we will show the balance values stored in the node objects. (Although a stored balance value and the corresponding proper one must be equal before the operation, and also after it, they may be different during the transformation.)

Let us consider the efficiency and some details of the basic operations of AVL trees. Remember that the height of an AVL tree is $\Theta(\log n)$. Thus the functions search(t, k), min(t) and max(t) of BSTs [1] (which do not modify the tree and just go down in the tree once) can be applied to AVL trees with $MT(n) \in \Theta(\log n)$ efficiency.

Procedures insert(t, k), del(t, k), remMin$(t, minp)$ and remMax$(t, maxp)$ of BSTs [1] applied to AVL trees also run on AVL trees with $MT(n) \in \Theta(\log n)$ efficiency, and the result will be a BST, but possibly imbalanced. (See Example 2.2. The input of both operations is the same AVL tree, but

the outputs are imbalanced BSTs.) Clearly, after many modifications the resulting tree may become too high for efficient run of the operations of BSTs. Therefore we need some modifications on these recursive procedures: When we return from a recursive call, we check whether the actual node of the tree became imbalanced, and if so, we perform the appropriate *rotations* in order to make it balanced. The extra elementary operations needed can be limited by the same constant at each level, so efficiency $MT(n) \in \Theta(\log n)$ remains true.

The *rules of rotations* of AVL trees can be found on Figures 2-7. One can see that *each rotation* keeps the original inorder traversal of the tree. Consequently, if the input of a rotation is a BST, its output is also a BST.

And after an addition or deletion, we consider the smallest imbalanced subtree containing the position of the insertion or deletion, and apply the appropriate rotation to it. Thus its subtrees are balanced, and the result of the rotation is a balanced subtree. Sometimes some rotations must be done also at higher levels of the tree afterwards.

2.1 AVL trees: insertion

First we consider procedure insert(t, k) of BSTs [1].

For example, given AVL tree { [2] 4+ [(6) 8= (10)] }.

- Inserting key 1, we receive AVL tree
  { [(1) 2−] 4= [(6) 8= (10)] }.
- Alternatively, inserting key 3, we receive AVL tree
  { [2+ (3)] 4= [(6) 8= (10)] }.
- Alternatively, inserting key 9, we receive imbalanced BST
  { [2] 4++ [(6) 8+ ({9} 10−)] }.
- Alternatively, inserting key 7, we receive imbalanced BST
  { [2] 4++ [(6+ {7}) 8− (10)] }.

We can see that in the last 2 cases the tree became imbalanced. If we want to receive an AVL tree, we have to make the appropriate rotations on the tree. Let us see some detailed examples.

Example 2.13 *Insert key* 3 *into AVL tree* { [2] 4+ [(6) 8= (10)] }.

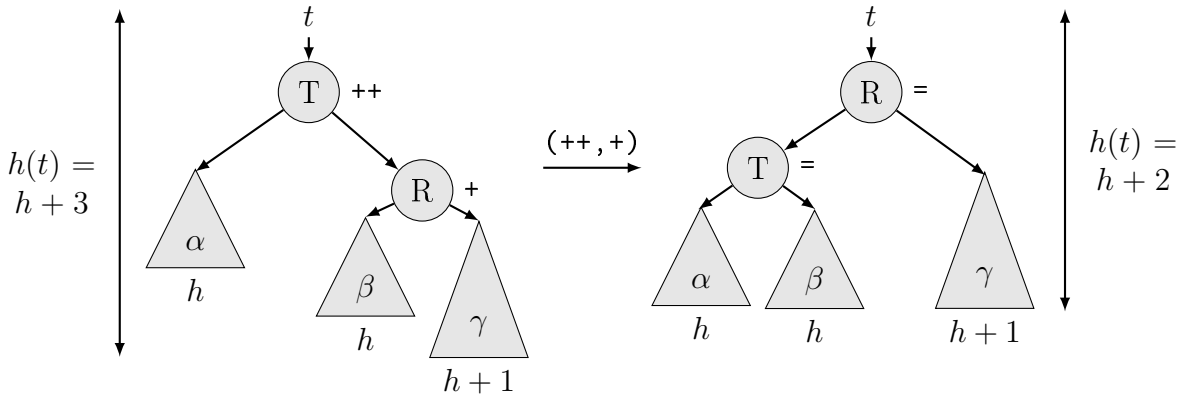


Figure 2: rotation $(++, +)$.

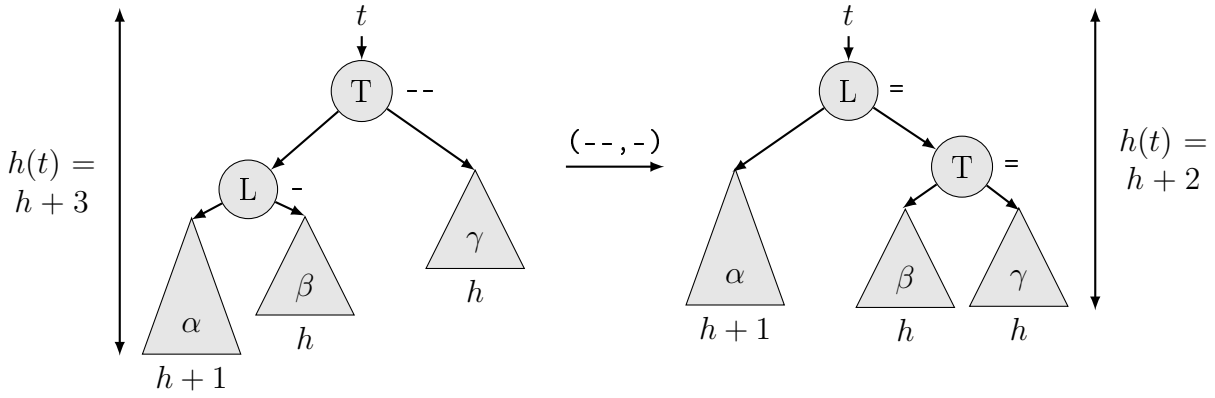


Figure 3: rotation $(--, -)$.

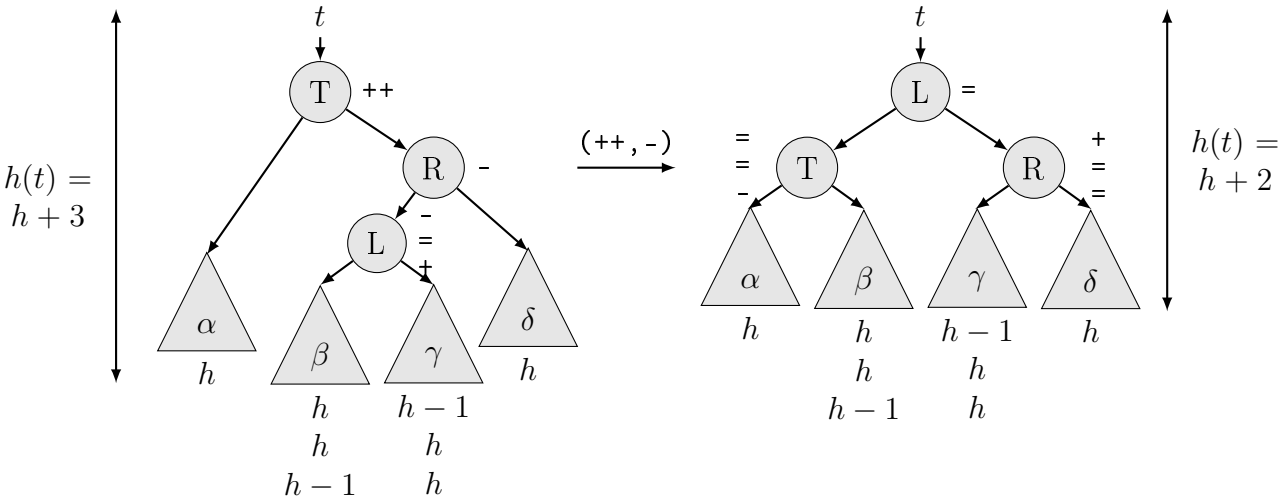


Figure 4: rotation $(++, -)$.

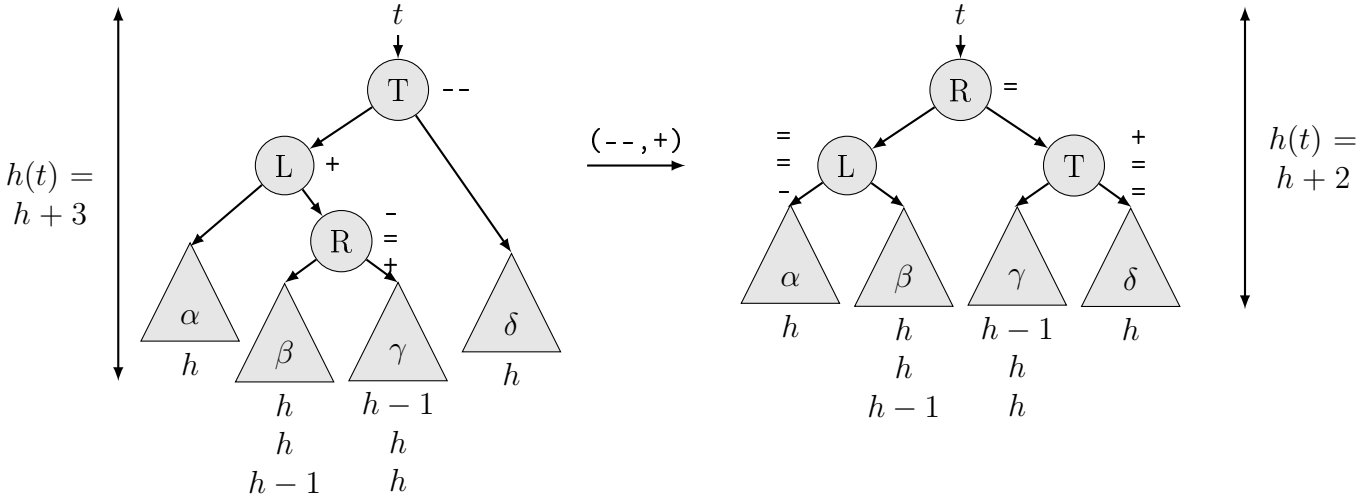


Figure 5: rotation $(--, +)$.

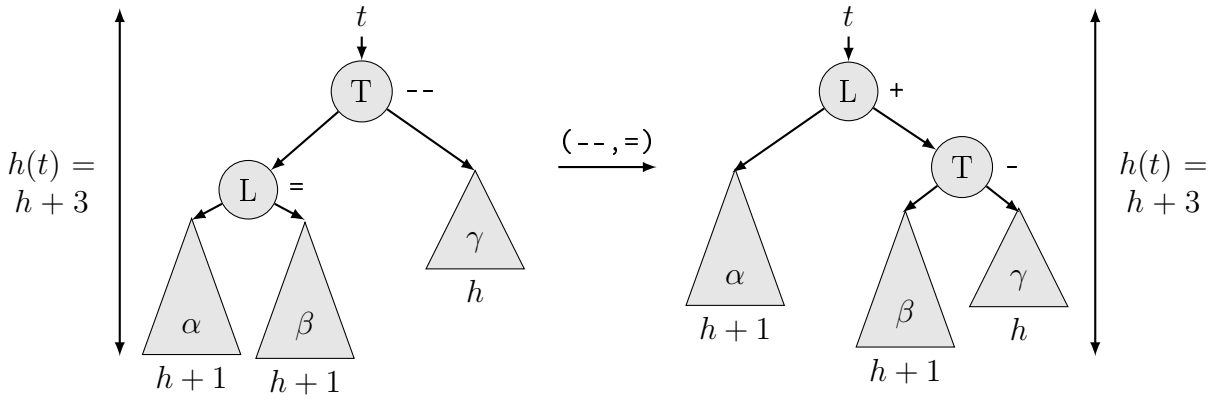


Figure 6: rotation $(--, =)$.

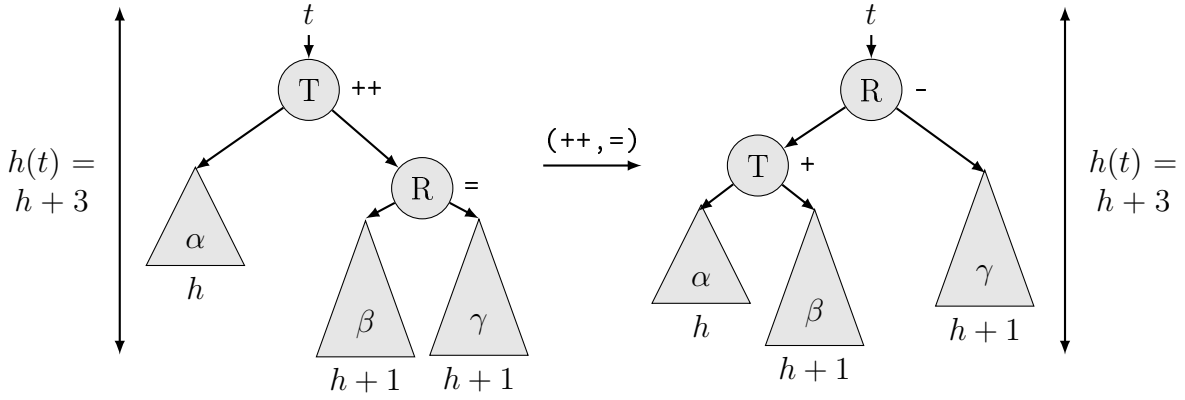


Figure 7: rotation $(++, =)$.

Solution: The key of the root of { [2] 4+ [(6) 8= (10)] } is 4.
3<4, so we go to the left subtree: [2].
3>2, so a new leaf ③ is inserted into the right, empty subtree of node ②.
Now we are at the new leaf ③. The actual subtree was empty. After inserting key 3, The actual subtree is (3). It has become higher.
Considering the balance information explicitly stored, this is the whole tree: { [2= (3)] 4+ [(6) 8= (10)] }. It is not rebalanced yet.
(Some of) the ancestors of the new leaf must be rebalanced.
We go up in the tree. First we arrive at node ②.
Its right subtree became higher, so its balance is increased by one.
Now, this is the whole tree: { [2+ (3)] 4+ [(6) 8= (10)] }.
The actual subtree is [2+ (3)], and it has become higher.
We step up in the tree. We arrive at the root node: ④.
Its left subtree has become higher, so its balance is reduced by one.
The actual subtree is the whole tree: { [2+ (3)] 4= [(6) 8= (10)] }.
We have finished rebalancing, and we can see that the result of insertion is a balanced BST, i.e. an AVL tree. Thus we have also finished insertion. We do not need any rotation here.

Exercise 2.14 *Insert key* 1 *into AVL tree* { [2] 4+ [(6) 8= (10)] }, *and show the details.*

Example 2.15 *Insert key* 9 *into AVL tree* { [2] 4+ [(6) 8= (10)] }.

Solution: We insert key 9. The key of the root is 4.
9>4, so we go to the right subtree: [(6) 8= (10)].
9>8, so we go to the right subtree: (10).
9<10, so we go to its left subtree, which is empty.
Thus we put the new node here: { [2] 4+ [(6) 8= (⟨9⟩ 10=)] }.
Then we go up and rebalance the ancestors at each level of the tree, but we stop when the first imbalanced node is found.
The subtree corresponding to this imbalanced node is the whole tree now: { [2] 4++ [(6) 8+ (⟨9⟩ 10−)] }.
We can use rotation (++, +) now (see also Figure 2):

$$[\ \alpha\ \mathrm{T}{+}{+}\ (\beta\ \mathrm{R}{+}\ \gamma)\ ] \to [\ (\alpha\ \mathrm{T}{=}\ \beta)\ \mathrm{R}{=}\ \gamma\ ]$$

Notice that in these rotation schemes the Greek letters are the subtrees and the English uppercase letters followed by the balance signs (like ++, +, = etc.) are the keys of the nodes.

We use the rotation scheme above on the imbalanced BST above where α=[2]=(2) ; T=4 ; β=(6) ; R=8 ; γ = (⟨9⟩ 10−) = [(9) 10−].
Finally we receive AVL tree { [(2) 4= (6)] 8= [(9) 10−] }.

Example 2.16 *Let us see an example of using rotation* $(++,-)$. *Insert key* 7 *into AVL tree* { [2] 4+ [(6) 8= (10)] }.

Solution: In order to keep the BST property, we find the appropriate empty subtree and insert node ⑦ there.
We receive tree { [2] 4+ [(6= ⟨7⟩) 8= (10)] }. Then we go up and rebalance the ancestors of the new leaf ⑦ at each level of the tree, but we stop when the first imbalanced node is found. We receive the following imbalanced BST:
{ [2] 4++ [(6+ ⟨7⟩) 8− (10)] }.
Rotation$(++,-)$ is needed (see also Figure 4):

$$\{\ \alpha\ \mathrm{T}{+}{+}\ [(\beta\ \mathrm{L}{-}{=}{+}\ \gamma)\ \mathrm{R}{-}\ \delta]\ \} \to \{\ [\alpha\ \mathrm{T}{=}{=}{-}\ \beta]\ \mathrm{L}{=}\ [\gamma\ \mathrm{R}{+}{=}{=}\ \delta]\ \}$$

In this $(++,-)$ rotation scheme, T++ is the root of the imbalanced BST. Its right child is R−. The left child of R is L.

L is the right-left grandchild of the imbalanced node T++. After the rotation, L becomes the new root of this (sub)tree. L's parent and grandparent become its two children. and the four subtrees: α, β, γ *and* δ *remain in their original order. The inorder traversal of the whole (sub)tree remains the same.*

L may have balance −, =, or +. After rotation, the new balance of T and R depends on the old balance of L. If the old balance of L was −, the new balance of T will be =, and that of R will be +, etc. (See Figure 4 for more details.) We use this later rotation scheme on the previous imbalanced BST { [2] 4++ [(6+ ⟨7⟩) 8− (10)] } where α=[2]=(2) ; T=4 ; β=⦸ ; L=6 ; γ=⟨7⟩=(7) ; R=8 ; δ=(10). The old balance of L=6 was +, so the new balance of T=4 will be −, and the new balance of R=8 will be =. Applying this rotation rule, we receive AVL tree
{ [(2) 4−] 6= [(7) 8= (10)] }.

Let us notice, if the balance of node L was b before applying this $(++,-)$ rotation rule, then (after applying this rotation rule) the new balance of T will be b_t, and that of R will be b_r where

$$b_t = -\lfloor (b+1)/2 \rfloor \quad \text{and} \quad b_r = \lfloor (1-b)/2 \rfloor .$$

We get similarly the rotation rules of the cases when the balance of the root of the smallest imbalanced subtree is −− (see also Figures 3 and 5.):

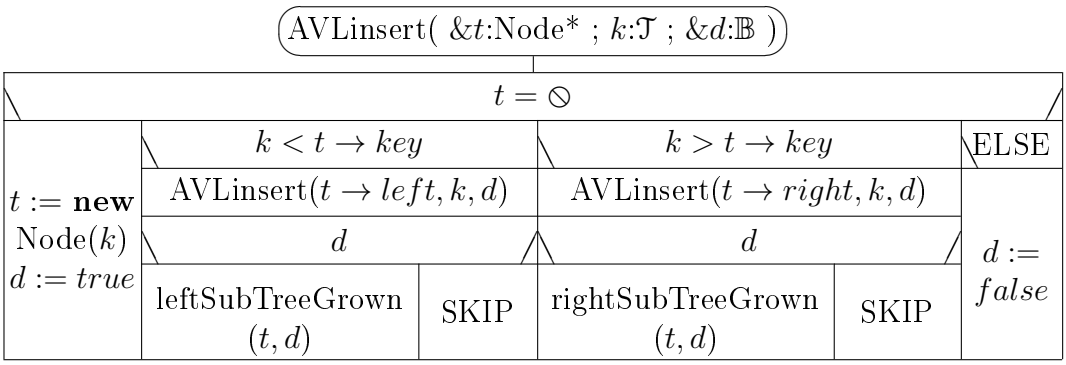
$$[\ (\alpha\ \mathrm{L}{-}\ \beta)\ \mathrm{T}{-}{-}\ \gamma\ ] \to [\ \alpha\ \mathrm{L}{=}\ (\beta\ \mathrm{T}{=}\ \gamma)\ ]$$
$$\{\ [\alpha\ \mathrm{L}{+}\ (\beta\ \mathrm{R}{-}{=}{+}\ \gamma)]\ \mathrm{T}{-}{-}\ \delta\ \} \to [\ (\alpha\ \mathrm{L}{=}{=}{-}\ \beta)\ \mathrm{R}{=}\ (\gamma\ \mathrm{T}{+}{=}{=}\ \delta)\ ]$$
$$b_l = -\lfloor (b+1)/2 \rfloor \quad \text{and} \quad b_t = \lfloor (1-b)/2 \rfloor$$

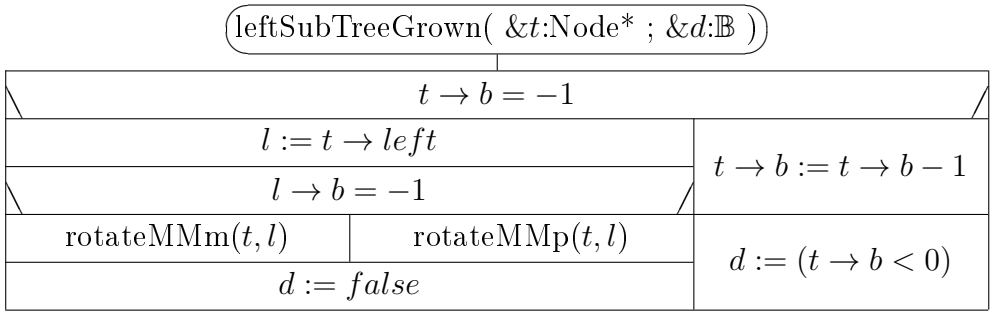
where b was the balance of node R before applying the $(--,+)$ rotation rule; but after applying this rule the new balance of L will be b_l, and that of T will be b_t.

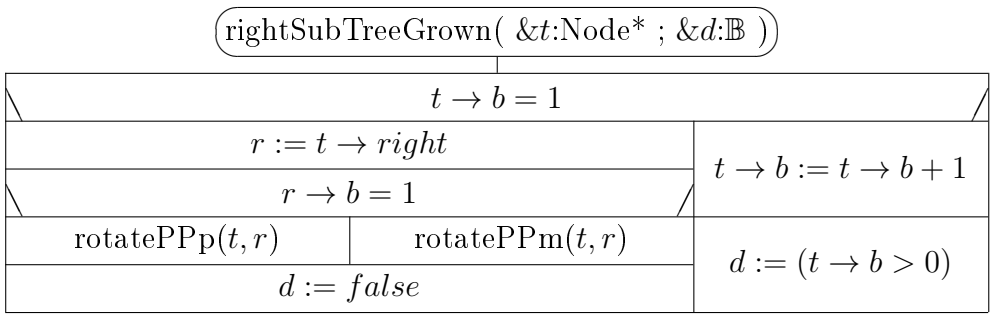
In the $(--,+)$ rotation scheme, T$--$ is the root of the imbalanced BST. Its left child is L$+$. The right child of L is R:

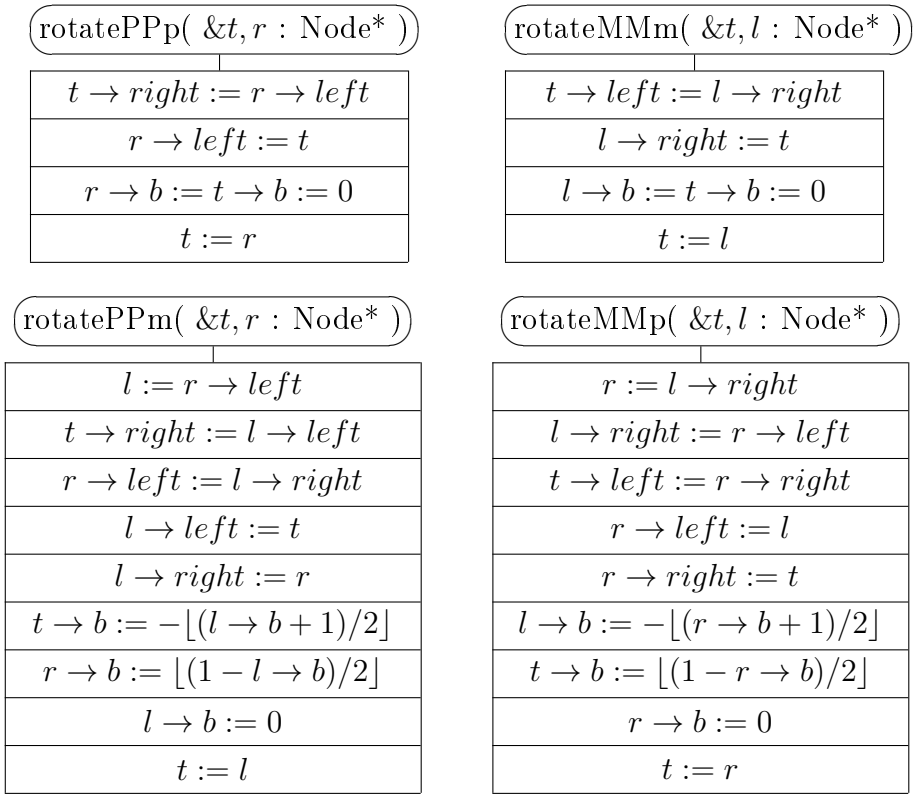
R is the left-right grandchild of the imbalanced node T$--$. After the rotation, R becomes the new root of this (sub)tree. R's parent and grandparent become its two children. and the four subtrees: α, β, γ and δ remain in their original order. The inorder traversal of the whole (sub)tree remains the same.

Let us see the detailed codes of insertion. Compared to the insert procedure of BSTs, we use an extra parameter here: d is a reference parameter of Boolean type. After the call, it is true, if the height of the actual subtree increased by one, and it is false, if its height did not change.









Notice that the rotation rules $(--,=)$ and $(++,=)$ on Figures 6 and 7 are never applied in insertion. (They will be applied in deletion.)

And one insertion applies maximum one of the rotation rules on Figures 2-5 and maximum once, because a successful insertion creates a new leaf, and then we go up level by level in the tree rebalancing the ancestors of the new leaf until

- we find an imbalanced node. In this case, the height of the actual subtree has been increased by one. Then we apply the appropriate rule, i.e. one of those on Figures 2-5. And each of them decreases the height of the actual subtree by one. (See Figures 2-5 for the details.) Thus the original height of this subtree has been restored, and we just leave insertion. Consequently, **in insertion, we never modify the balances of the nodes outside of the smallest imbalanced subtree.**

- rebalancing a node we find that it is perfectly balanced. This means that one of its direct subtrees has grown to the height of the other. Therefore the height of the subtree corresponding to the actual node has not been changed by the insertion. Thus we have finished rebalancing.

We did not find imbalanced node, and the insertion has been finished without rotation.

- we arrive at the root of the whole tree but do not find imbalanced node. This case is similar to the previous one: no rotation is needed, because the BST remained balanced.

Example 2.17 *Let us see an example of the first case above, especially when the imbalanced node is not the root of the whole tree. Insert key* 7 *into AVL tree* { [(1) 2−] 3+ [(⟨4⟩ 5= ⟨6⟩) 8− (9)] }.

Solution: After inserting key 7, but still before rebalancing we have the following tree: { [(1) 2−] 3+ [(⟨4⟩ 5= ⟨ 6= {7} ⟩) 8− (9)] }.
We rebalance nodes ⑥, ⑤, and ⑧. We stop rebalancing here, because node ⑧−− has become imbalanced (thus node ③+ is not rebalanced):
{ [(1) 2−] 3+ [(⟨4⟩ 5+ ⟨ 6+ {7} ⟩) 8−− (9)] }
Next we apply rotation $(--, +)$ at node ⑧−−.
The actual rule of rotation is
[(α L+ ⟨β R+ γ⟩) T−− δ] → [(α L− β) R= (γ T= δ)]
with α=⟨4⟩, L=5, β=⦸, R=6, γ={7}=⟨7⟩, T=8, δ=(9)=⟨9⟩
(see also Figure 5).
The result: { [(1) 2−] 3+ [(⟨4⟩ 5−) 6= (⟨7⟩ 8= ⟨9⟩)] }.

Exercise 2.18 *Let us modify the* Node *type of* AVL trees *so that we store the height of the corresponding subtree in each node, but we do not store the balances.*

The four simpler rules of rotations on figures 2, 3, 6, 7 can be reduced to two rules + recalculating the heights of the modified subtrees. The two simple rotations:
Right-to-left rotation: $[\alpha\ T\ (\beta\ R\ \gamma)] \to [(\alpha\ T\ \beta)\ R\ \gamma]$
Left-to-right rotation: $[(\alpha\ L\ \beta)\ T\ \gamma] \to [\alpha\ L\ (\beta\ T\ \gamma)]$
Which one should be used in the different cases?

The more complex rules of rotations which correspond to figures 4 and 5 can be considered double rotations *now, because we can get them as two simple rotations applied at the appropriate points of the the tree + recalculating the heights of the modified subtrees.*

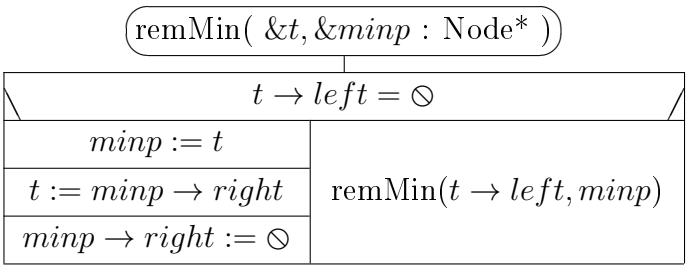
How to use the simple rotations above in the different cases?

Exercise 2.19 *At structogram (and/or C/C++) level, implement insertion using the new rules of rotations developed in Exercise 2.18.*

Compare your implementation of insertion to the corresponding structograms given above. What can we say about the complexity of the code? What can we tell about efficiency issues?

2.2 AVL trees: removing the minimal (maximal) node

On nonempty BSTs, procedure remMin removes the minimal node (i.e. the node with minimal key):



Applying this to AVL tree { [2] 4+ [(6) 8= (10)] }, we receive the imbalanced BST { 4++ [(6) 8= (10)] } while *minp* refers to the minimal node of the original tree where $minp \to key = 2 \wedge minp \to left = minp \to right = \oslash$. Notice that we show the theoretical balances above. At representation level, the code above does not do anything with the balances (b attributes). After removing the minimal node, we have to go up from the place of it, and level by level rebalance the ancestors of it. If we find a node with "++" balance, the appropriate rotation must be applied. (Notice that balance "−−" is not possible when we remove the minimal node.)

Rotation $(++, =)$ is needed here (see Figure 7 for the details):

$$\{\ \alpha\ \mathrm{T}{+}{+}[\beta\ \mathrm{R}{=}\ \gamma]\ \} \to \{\ [\alpha\ \mathrm{T}{+}\ \beta]\ \mathrm{R}{-}\ \gamma\ \}.$$

Notice that this rotation does not modify the height of the actual subtree. Therefore, neither more rebalancing nor more rotations are needed after this rotation, even if this rotation is applied to a proper subtree of the whole BST. The final result is { [4+ (6)] 8− (10) } here.

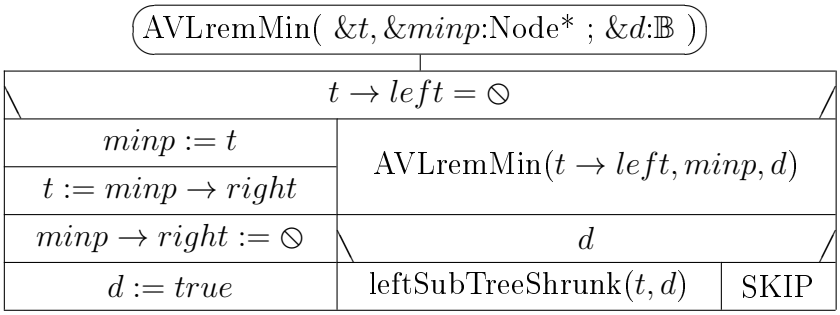
Notwithstanding, when rotation $(++, +)$ or $(++, -)$ is needed, then the rotated subtree becomes lower, and after the rotation, rebalancing goes on. Then it is possible that two or more rotations are needed at different levels. However, each rotation needs constant time, and the number of levels is not more than $1.45 \log n$. Thus the maximal runtime is still $\Theta(\log n)$.

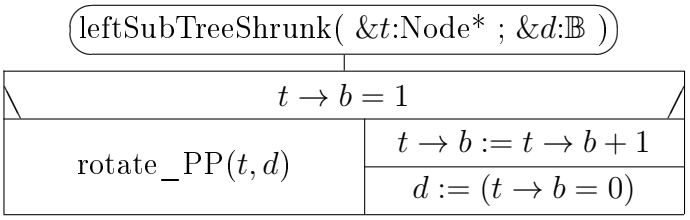
Example 2.20 *Remove the minimal node from AVL tree*
{ [(1) 2+ (3+ ⟨4⟩)] 5+ [(⟨6⟩ 7+ ⟨ {8} 9− ⟩) 10− (11+ ⟨12⟩)] } .
We need two rotations here.

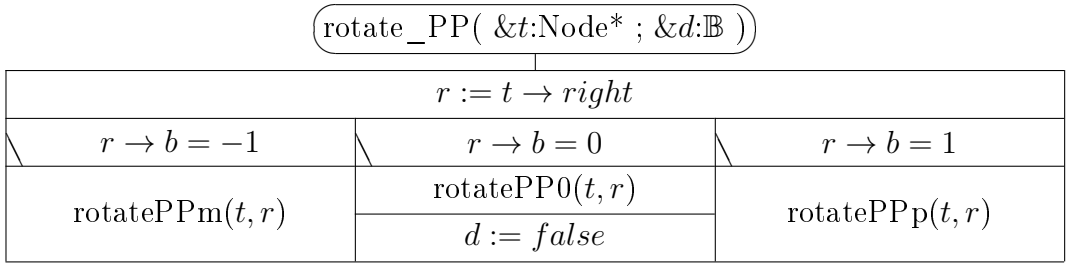
Solution: We remove the leftmost node, i.e. node ①. The appropriate subtree becomes empty. Its height is decreased by one. Then the balance of node ② becomes "++". The subtree rooted by node ② needs rotation

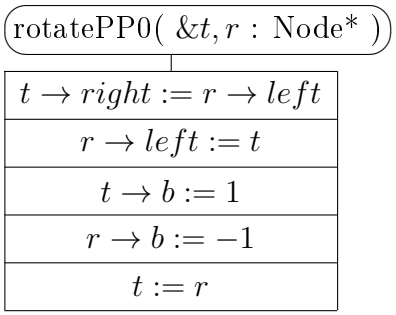
(++,+): [2++ (3+ ⟨4⟩)] → [(2) 3= (4)]. The height of this subtree is decreased by one. (Rotation (++,+) always decreases the height of the rotated subtree by one.) Consequently the balance of node ⑤ is increased by one. The whole tree now:
{ [(2) 3= (4)] 5++ [(⟨6⟩ 7+ ⟨ {8} 9− ⟩) 10− (11+ ⟨12⟩)] }.
The right child of node ⑤++ is ⑩−. Thus we need rotation (++, −). The left child of ⑩− is ⑦+. After rotation (++, −) (see Figure 4 for the details), always the right-left grandchild of the root of the rotated subtree becomes its root. This is ⑦+ now. Its original parent and grandparent becomes its two children, and the four subtrees are arranged in order:
{ [(⟨2⟩ 3= ⟨4⟩) 5− (6)] 7= [(⟨8⟩ 9−) 10= (11+ ⟨12⟩)] }.
The balances of the three nodes are determined by the old balance of the original grandson (⑦+ here), according to Figure 4. The four subtrees are not changed. (Just the brackets were rewritten here for better reading.)

Based on the rules discussed above, we can complete the code of procedure remMin($t, minp$) above. For AVL trees, we need an extra Boolean parameter d which is *true*, **iff** the height of the appropriate subtree has been decreased by one. (It cannot decrease by two or more.) When we return from a recursive call, we check d. If it is *true*, we have to go on with rebalancing and possibly rotations. If d is *false*, neither subsequent rebalancing nor rotations are needed.





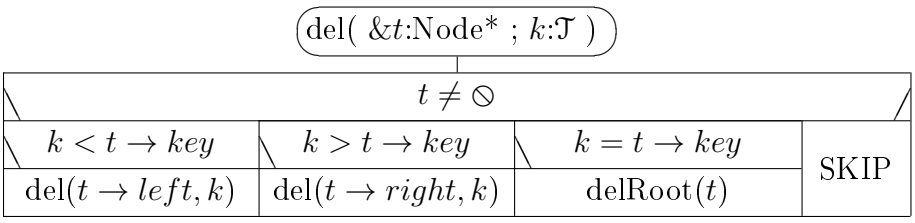




Exercise 2.21 *Based on procedure* AVLremMin$(t, minp, d)$ *above, write the structograms of procedure* AVLremMax$(t, maxp, d)$. *You will need the code of rotation rules* $(--, -)$, $(--, +)$ *and* $(--, =)$ *defined on Figures 3, 5 and 6. You should write new code for rotation rule* $(--, =)$ *defined on Figure 6.*

2.3 AVL trees: deletion

Procedures del(t, k) and delRoot(t) for BSTs:



delRoot(&t:Node*)

$t \to left = \oslash$	$t \to right = \oslash$	$t \to left \neq \oslash \wedge t \to right \neq \oslash$
$p := t$	$p := t$	remMin$(t \to right, p)$
$t := p \to right$	$t := p \to left$	$p \to left := t \to left$
		$p \to right := t \to right$
delete p	**delete** p	**delete** t ; $t := p$

These are completed below as it was done in case of procedure AVLremMin$(t, minp, d)$ in section 2.2. We add Boolean parameter d which is *true*, **iff** the height of the appropriate subtree has been decreased by one. (It cannot decrease by two or more.) When we return from a recursive call, we check d. If it is *true*, we have to go on with rebalancing and possibly rotations. If d is $false$, neither subsequent rebalancing nor rotations are needed.

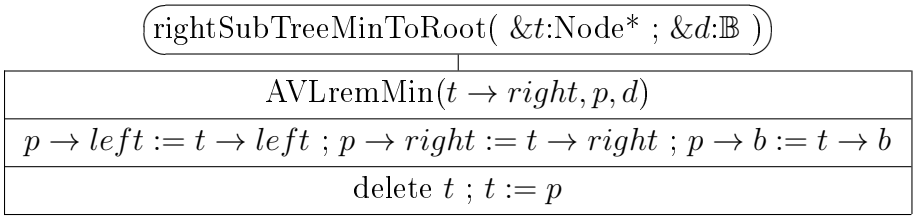
AVLdel(&t:Node* ; k:$\mathcal{T}$; &d:$\mathbb{B}$)

<table>
<tr><td colspan="5">$t \neq \oslash$</td></tr>
<tr><td colspan="2">$k < t \to key$</td><td colspan="2">$k > t \to key$</td><td>$k = t \to key$</td></tr>
<tr><td colspan="2">AVLdel$(t \to left, k, d)$</td><td colspan="2">AVLdel$(t \to right, k, d)$</td><td rowspan="3">AVLdelRoot (t, d)</td></tr>
<tr><td colspan="2">d</td><td colspan="2">d</td></tr>
<tr><td>leftSubTreeShrunk (t, d)</td><td>SKIP</td><td>rightSubTreeShrunk (t, d)</td><td>SKIP</td></tr>
</table>

$d :=$ $false$

AVLdelRoot(&t:Node* ; &d:$\mathbb{B}$)

<table>
<tr><td>$t \to left = \oslash$</td><td>$t \to right = \oslash$</td><td colspan="2">$t \to left \neq \oslash \wedge t \to right \neq \oslash$</td></tr>
<tr><td>$p := t$</td><td>$p := t$</td><td colspan="2" rowspan="2">rightSubTreeMinToRoot(t, d)</td></tr>
<tr><td>$t := p \to right$</td><td>$t := p \to left$</td></tr>
<tr><td>**delete** p</td><td>**delete** p</td><td colspan="2">d</td></tr>
<tr><td>$d := true$</td><td>$d := true$</td><td>rightSubTreeShrunk(t, d)</td><td>SKIP</td></tr>
</table>



When rotation $(++,=)$ or $(--,=)$ is performed, it does not modify the height of the actual subtree. Therefore, neither more rebalancing nor more rotations are needed after one of these rotations, even if the rotation is applied to a proper subtree of the whole BST.

Notwithstanding, when rotation $(++,+)$, $(--,-)$, $(++,-)$ or $(--,+)$ is needed (see Figures 2-5), then the rotated subtree becomes lower, and after the rotation, rebalancing goes on. Then it is possible that two or more rotations are needed at different levels. However, each rotation needs constant time, and the number of levels is not more than $1.45 \log n$. Thus the maximal runtime of deletion is also $\Theta(\log n)$.

Exercise 2.22 *Based on procedure* leftSubTreeShrunk(t, d) *write procedure* rightSubTreeShrunk(t, d) *together with its auxiliary procedures. (See rotation rule* $(--,=)$ *given on Figure 6.)*

Example 2.23 *Delete key 2 from AVL tree*
{ [(1) 2+ (3+ ⟨4⟩)] 5+ [(⟨6⟩ 7+ ⟨ {8} 9− ⟩) 10− (11+ ⟨12⟩)] } .

Solution: We delete node ②. AVLremMin is called on its right subtree, and node ③ is removed from it, so it will be AVL tree (4). No rotation is needed during AVLremMin, but the height of this right subtree is decreased by one, so AVLremMin returns with $d = true$. Node ② is substituted by node ③ which inherits the balance of node ②. Then its balance is decreased by one because $d = true$. Thus the actual subtree becomes [(1) 3= (4)]. The height of this subtree has been decreased by one. Consequently the balance of node ⑤ is increased by one. The whole tree now:
{ [(1) 3= (4)] 5++ [(⟨6⟩ 7+ ⟨ {8} 9− ⟩) 10− (11+ ⟨12⟩)] }.
The right child of node ⑤++ is ⑩−. Thus we need rotation $(++,-)$. The left child of ⑩− is ⑦+. After rotation $(++,-)$ (see Figure 4 for the details), always the right-left grandchild of the root of the rotated subtree becomes its root. This is ⑦+ now. Its original parent and grandparent becomes its two children, and the four subtrees are arranged in order:
{ [(⟨1⟩ 3= ⟨4⟩) 5− (6)] 7= [(⟨8⟩ 9−) 10= (11+ ⟨12⟩)] }.

The balances of the three nodes are determined by the old balance of the original grandchild (⑦+ here), according to Figure 4. The four subtrees are not changed. (Just the brackets were rewritten here for better reading.)

Example 2.24 *Delete key 5 from AVL tree*
{ [(1) 2+ (3+ ⟨4⟩)] 5+ [(⟨6⟩ 7+ ⟨ {8} 9− ⟩) 10− (11+ ⟨12⟩)] } .

Solution: In order to delete node ⑤, first AVLremMin is called on its right subtree: [(⟨6⟩ 7+ ⟨ {8} 9− ⟩) 10− (11+ ⟨12⟩)]. AVLremMin is called recursively on its left subtree: (⟨6⟩ 7+ ⟨ {8} 9− ⟩). We remove its lefmost node ⑥. The balance of its parent ⑦+ is increased by one. The corresponding subtree is (7++ ⟨ {8} 9− ⟩). Rotation (++, −) is performed on it. The result is (⟨7⟩ 8= ⟨9⟩). Its height is decreased by one, so the recursive call to AVLremMin returns with $d = true$, and the balance of ⑩− is increased by one. The corresponding subtree is
[(⟨7⟩ 8= ⟨9⟩) 10= (11+ ⟨12⟩)].
Its height is decreased by one, so the first call to AVLremMin returns with $d = true$. After the run of AVLremMin the whole tree is
{ [(1) 2+ (3+ ⟨4⟩)] 5+ [(⟨7⟩ 8= ⟨9⟩) 10= (11+ ⟨12⟩)] }.
Node ⑤ is substituted by node ⑥:
{ [(1) 2+ (3+ ⟨4⟩)] 6+ [(⟨7⟩ 8= ⟨9⟩) 10= (11+ ⟨12⟩)].
$d = true$ still shows that height of its right subtree was decreased by one, so the balance of ⑥+ is also decreased by one. The final AVL tree is
{ [(1) 2+ (3+ ⟨4⟩)] 6= [(⟨7⟩ 8= ⟨9⟩) 10= (11+ ⟨12⟩)].

Exercise 2.25 *Write an alternative code of procedure* AVLdel(t, k, d). *It shall be similar to the previous one, but if key k is in a node with two children, it shall substitute this node with the maximal node of its left subtree.*

3 General trees

Compared to binary trees, a node of a *rooted tree* may have unlimited (although finite) number of children. A rooted tree is *ordered* if the order of the children of nodes is important. An ordered rooted tree is also called *general tree*. (See Figure 8.) We will not define empty subtrees of *general trees*.

Using *general trees* we can model hierarchical structures like

- directory hierarchies in computers
- (block) structure of programs

- different kinds of expression in mathematics and computer science
- family trees etc.

General trees can be represented with linked binary trees in a natural way. Type Node of the nodes of general trees is given below.

- Pointer $firstChild$ refers to the first child of a node.
  $p \to firstChild = \oslash$, **iff** node $(*p)$ is a leaf.
- Pointer *nextSibling* refers to the next sibling in a list of children.
  $p \to nextSibling = \oslash$, **iff** $(*p)$ is the root node of the tree, or it is the last sibling in a list of children.

Node
+ $firstChild, nextSibling$: Node*   + key : $\mathcal{T}$
+ Node() { $firstChild := nextSibling := \oslash$ }   + Node(x:$\mathcal{T}$) { $firstChild := nextSibling := \oslash$; $key := x$ }

In the nodes, there might be *parent* pointers referring to the parent of the children. We will not use parent pointers here.

Exercise 3.1 *Try to invent alternative (linked) representations of general trees. Compare your representations to the linked binary representation above. Consider memory needs and flexibility.*

In the textual or parenthesized representation of general trees we start with the root of the actual (sub)tree. Thus a nonempty tree fits the general scheme

$$(R \;\; t_1 \ldots t_n)$$

where R is the content of the root node and $t_1 \ldots t_n$ are the direct subtrees of node R.

Example 3.2 *In general tree* { 1 [2 (5)] (3) [4 (6) (7)] }, 1 *is in the root. Its children are the nodes with keys* 2, 3 *and* 4. *The corresponding subtrees are* [2 (5)], (3) *and* [4 (6) (7)] *in order. The leaves of the tree are the nodes with keys* 5, 3, 6 *and* 7. *(See* Figure 8.*)*

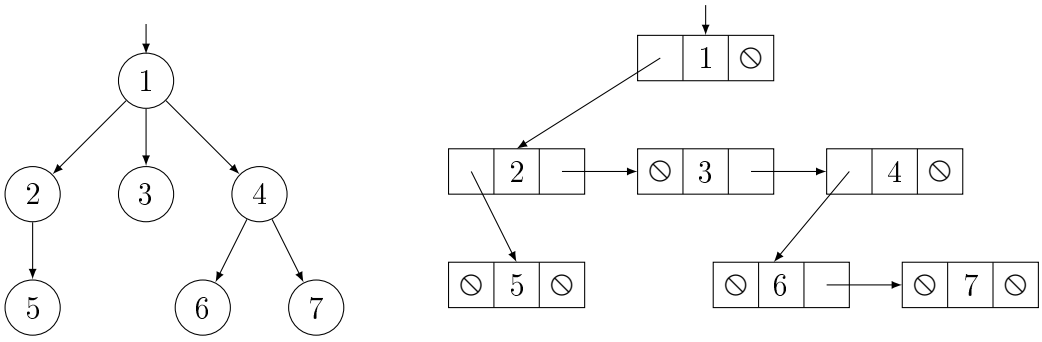


Figure 8: The abstract structure of a general tree on the left and its linked binary representation on the right. Its textual or parenthesized representation is { 1 [2 (5)] (3) [4 (6) (7)] }. (The different kinds of brackets may vary.)

Exercise 3.3 *Write a procedure printing a* general tree *given in* linked binary representation. *The result should be in textual representation.*

Write another procedure which can build a general tree *in* linked binary representation. *Its input is a textfile where the tree is given in parenthesized form.*

Write these printing and reading procedures for the linked representations you invented in Exercise 3.1.

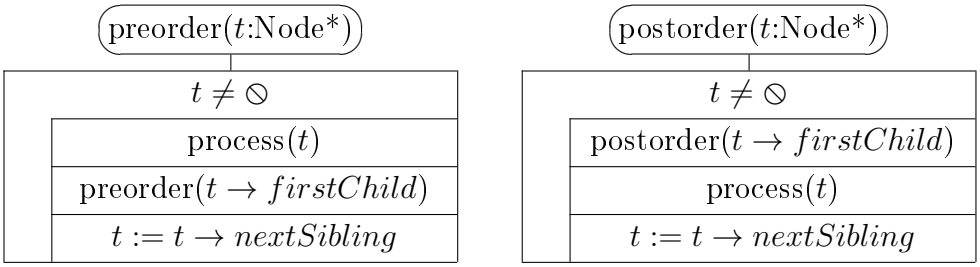
(The printing procedures are usually more straightforward than those which build up the linked representation from the textual form.)

In order to make the *preorder* traversal of a general tree, we have to traverse its binary representation also with *preorder* traversal where $firstChild$ corresponds to $left$ and *nextSibling* corresponds to *right*.

But the the *postorder* traversal of a general tree requires the *inorder* traversal of its binary representation.[6]

The *preorder* and *postorder* traversals of a general tree are given below, provided that it is given in linked binary representation.

[6] A search in the file system of a computer requires *preorder* traversal, while evaluating an expression requires *postorder* traversal of the general (i.e. theoretical) expression tree.



Exercise 3.4 *Write the traversals of general trees above for the representations you invented while solving* Exercise 3.1.

Given a general tree *in some representation, write a procedure copying it into another given representation.*

4 B+ trees and their basic operations

See file **http://aszt.inf.elte.hu/~asvanyi/ds/AlgDs2/B+trees.pdf**.

B+-trees

Contents:

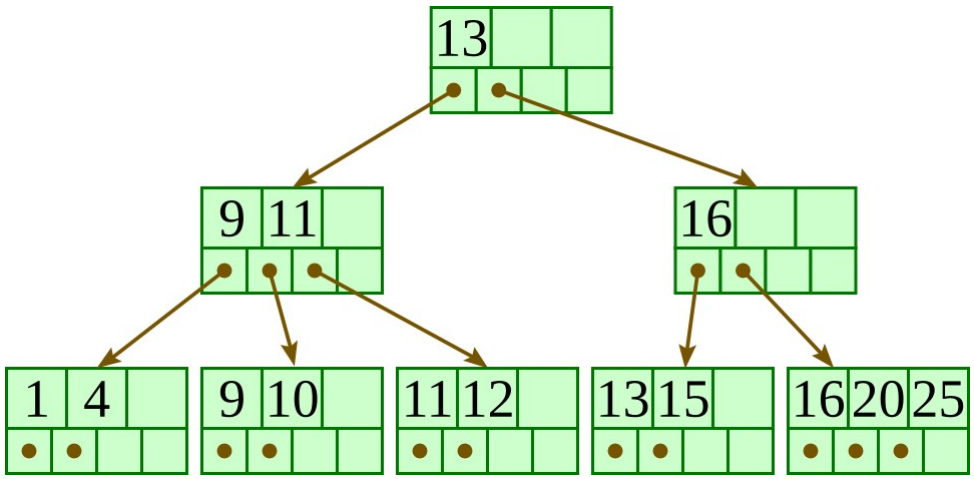
1. What is a B+-tree?

Most queries can be executed more quickly if the values are stored in order. But it's not practical to hope to store all the rows in the table one after another, in sorted order, because this requires rewriting the entire table with each insertion or deletion of a row.

This leads us to instead imagine storing our rows in a tree structure. Our first instinct would be a balanced binary search tree like a red-black tree, but this really doesn't make much sense for a database since it is stored on disk. You see, disks work by reading and writing whole **blocks** of data at once — typically 512 bytes or four kilobytes. A node of a binary search tree uses a small fraction of that, so it makes sense to look for a structure that fits more neatly into a disk block.

Hence the B+-tree, in which each node stores up to d references to children and up to $d - 1$ keys. Each reference is considered "between" two of the node's keys; it references the root of a subtree for which all values are between these two keys.

Here is a fairly small tree using 4 as our value for d.



A B+-tree requires that each leaf be the same distance from the root, as in this picture, where searching for any of the 11 values (all listed on the bottom level) will involve loading three nodes from the disk (the root block, a second-level block, and a leaf).

In practice, d will be larger — as large, in fact, as it takes to fill a disk block. Suppose a block is 4KB, our keys are 4-byte integers, and each reference is a 6-byte file offset. Then we'd choose d to be the largest value so that $4\ (d - 1) + 6\ d \leq 4096$; solving this inequality for d, we end up with $d \leq 410$, so we'd use 410 for d. As you can see, d can be large.

A B+-tree maintains the following invariants:

- Every node has one more references than it has keys.
- All leaves are at the same distance from the root.
- For every non-leaf node N with k being the number of keys in N: all keys in the first child's subtree are less than N's first key; and all keys in the ith child's subtree ($2 \leq i \leq k$) are between the $(i - 1)$th key of n and the ith key of n.
- The root has at least two children.
- Every non-leaf, non-root node has at least $floor(d / 2)$ children.
- Each leaf contains at least $floor(d / 2)$ keys.
- Every key from the table appears in a leaf, in left-to-right sorted order.

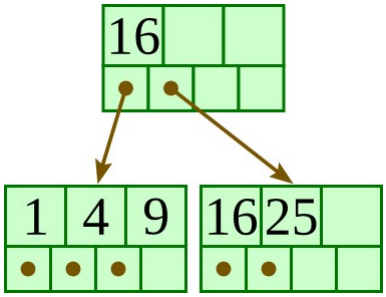
In our examples, we'll continue to use 4 for d. Looking at our invariants, this requires that each leaf have at least two keys, and each internal node to have at least two children (and thus at least one key).

2. Insertion algorithm

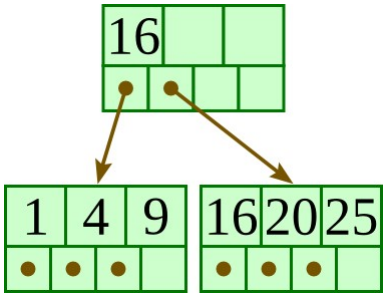
Descend to the leaf where the key fits.

1. If the node has an empty space, insert the key/reference pair into the node.
2. If the node is already full, split it into two nodes, distributing the keys evenly between the two nodes. If the node is a leaf, take a copy of the minimum value in the second of these two nodes and repeat this insertion algorithm to insert it into the parent node. If the node is a non-leaf, exclude the middle value during the split and repeat this insertion algorithm to insert this excluded value into the parent node.

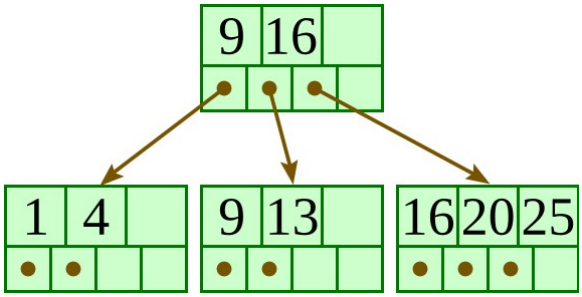
Initial:



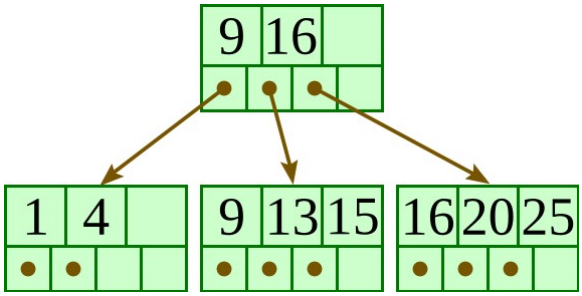
Insert 20:



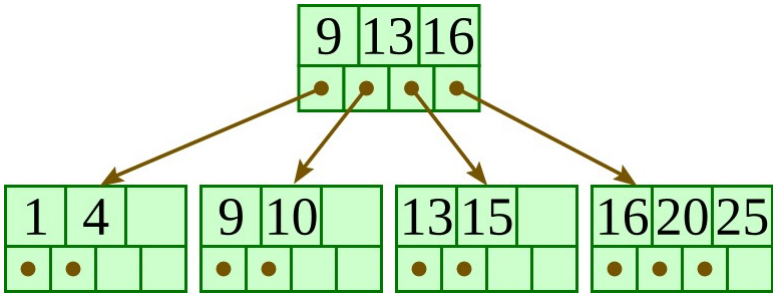
Insert 13:



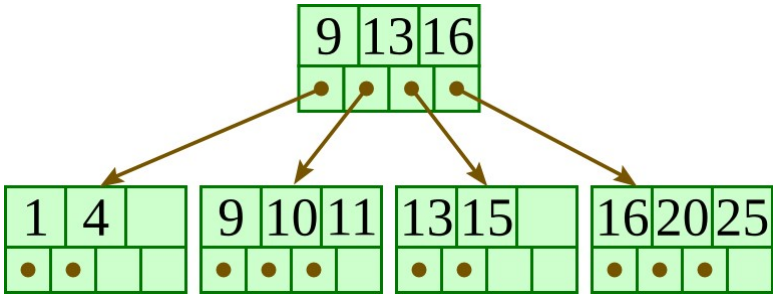
Insert 15:



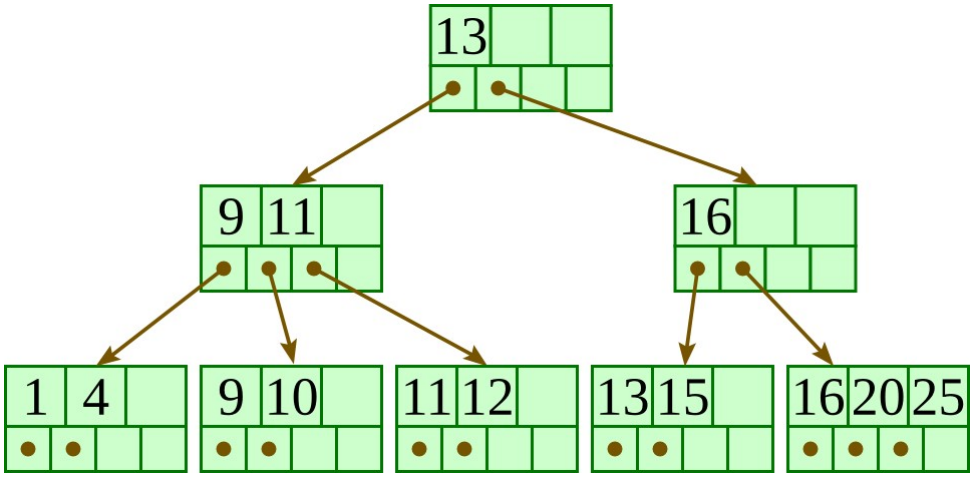
Insert 10:



Insert 11:



Insert 12:



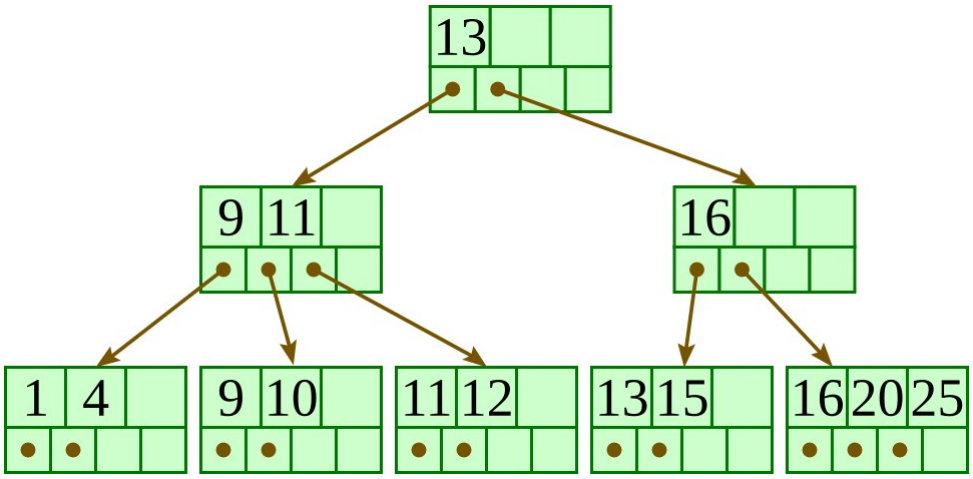
3. Deletion algorithm

Descend to the leaf where the key exists.

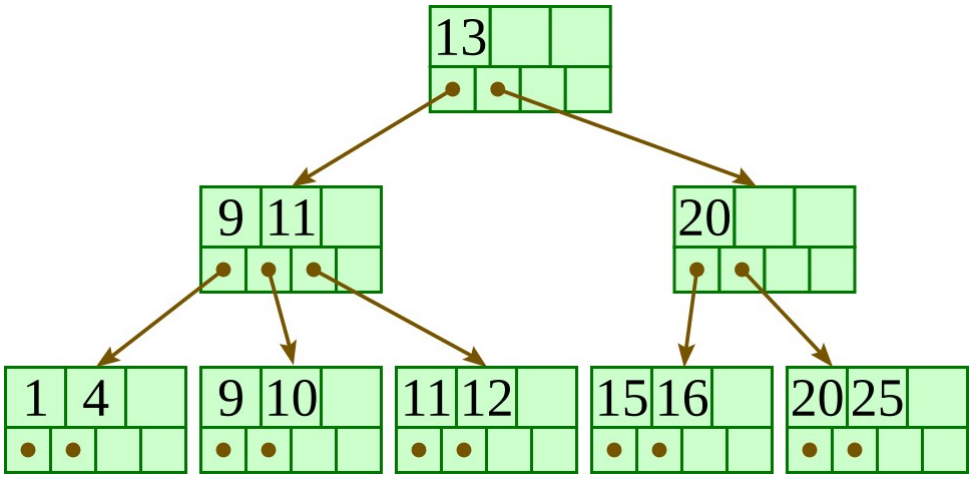
1. Remove the required key and associated reference from the node.
2. If the node still has enough keys and references to satisfy the invariants, stop.
3. If the node has too few keys to satisfy the invariants, but its next oldest or next youngest sibling at the same level has more than necessary, distribute the keys between this node and the neighbor. Repair the keys in the level above to represent that these nodes now have a different “split point” between them; this involves simply changing a key in the levels above, without deletion or insertion.
4. If the node has too few keys to satisfy the invariant, and the next oldest or next youngest sibling is at the minimum for the invariant, then merge the node with its sibling; if the node is a non-leaf, we will need to incorporate the “split key” from the parent into our merging. In either case, we will need to repeat the removal algorithm on the parent node to remove the “split key” that previously

separated these merged nodes — unless the parent is the root and we are removing the final key from the root, in which case the merged node becomes the new root (and the tree has become one level shorter than before).

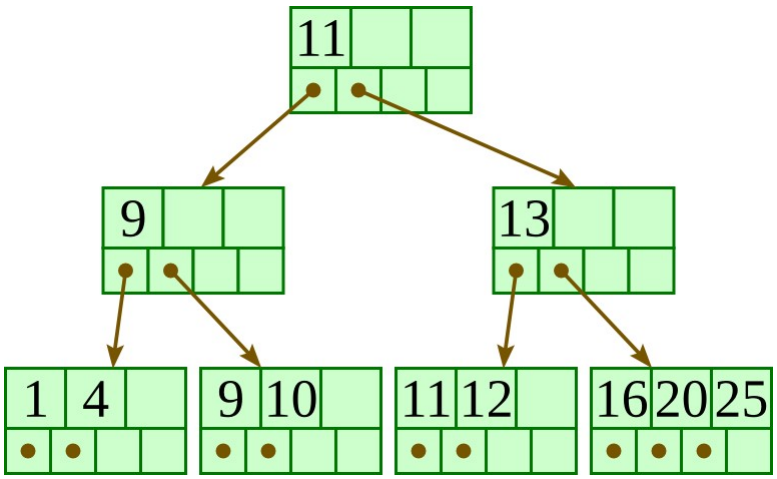
Initial:



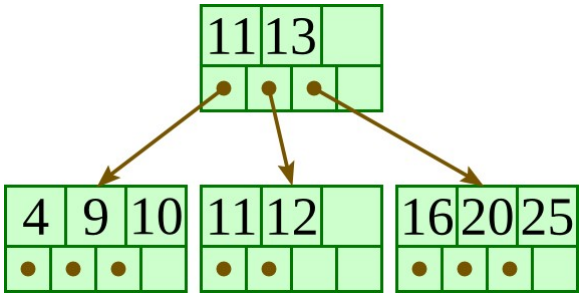
Delete 13:



Delete 15:

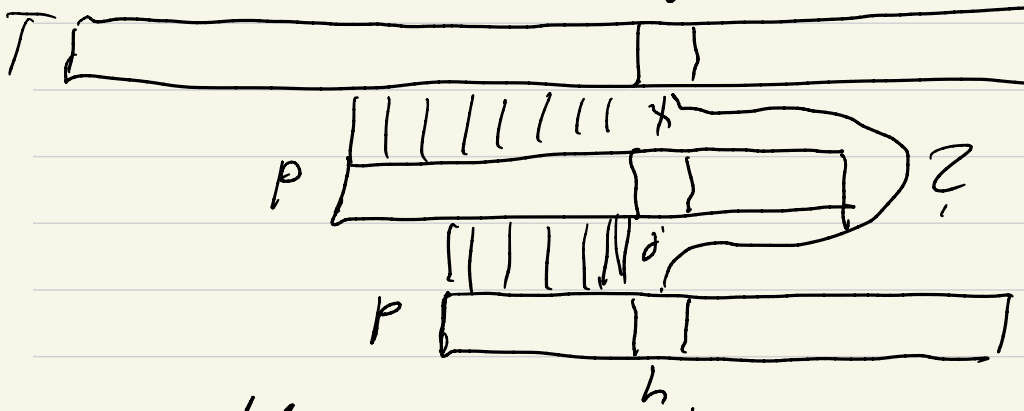


Delete 1:



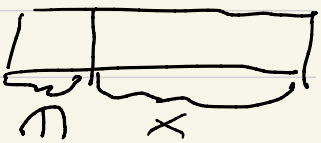
KMP algorithm

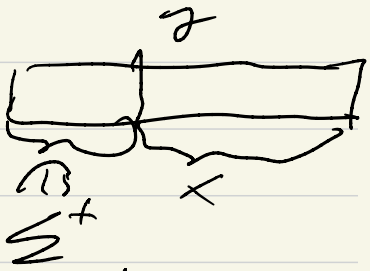
$T : \Sigma[n]$; $P : \Sigma[m]$



$\max\_h : P[0..h) \sqsupset P[0..j)$

$P_j := P[0..j) \;//\; P_h \sqsupset P_j$





$x \sqsupset y \;//\; x \sqsubset y$

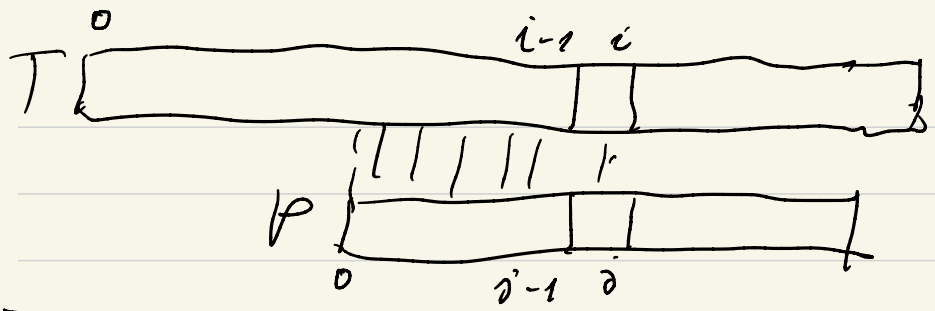
D. $H(j) \overset{def}{=} \{ h \in 0..j-1 \mid P_h \sqsupset P_j \}$ $(j \in 1..m)$

$x \square y \overset{def}{\iff} x \sqsubset y \wedge x \sqsupset y$

$H(j) = \{ |x| \mid x \square P_j \}$

D. $next : 1..m \longrightarrow 0..m-1$

$next(j) = \max H(j)$



$P_{j+1} \sqsupseteq T_{i+1}$

$\Leftrightarrow$ (II)

$P_j \sqsupseteq T_i \wedge P[j] = T[i]$

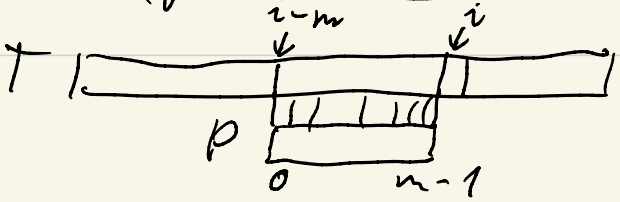
Tut

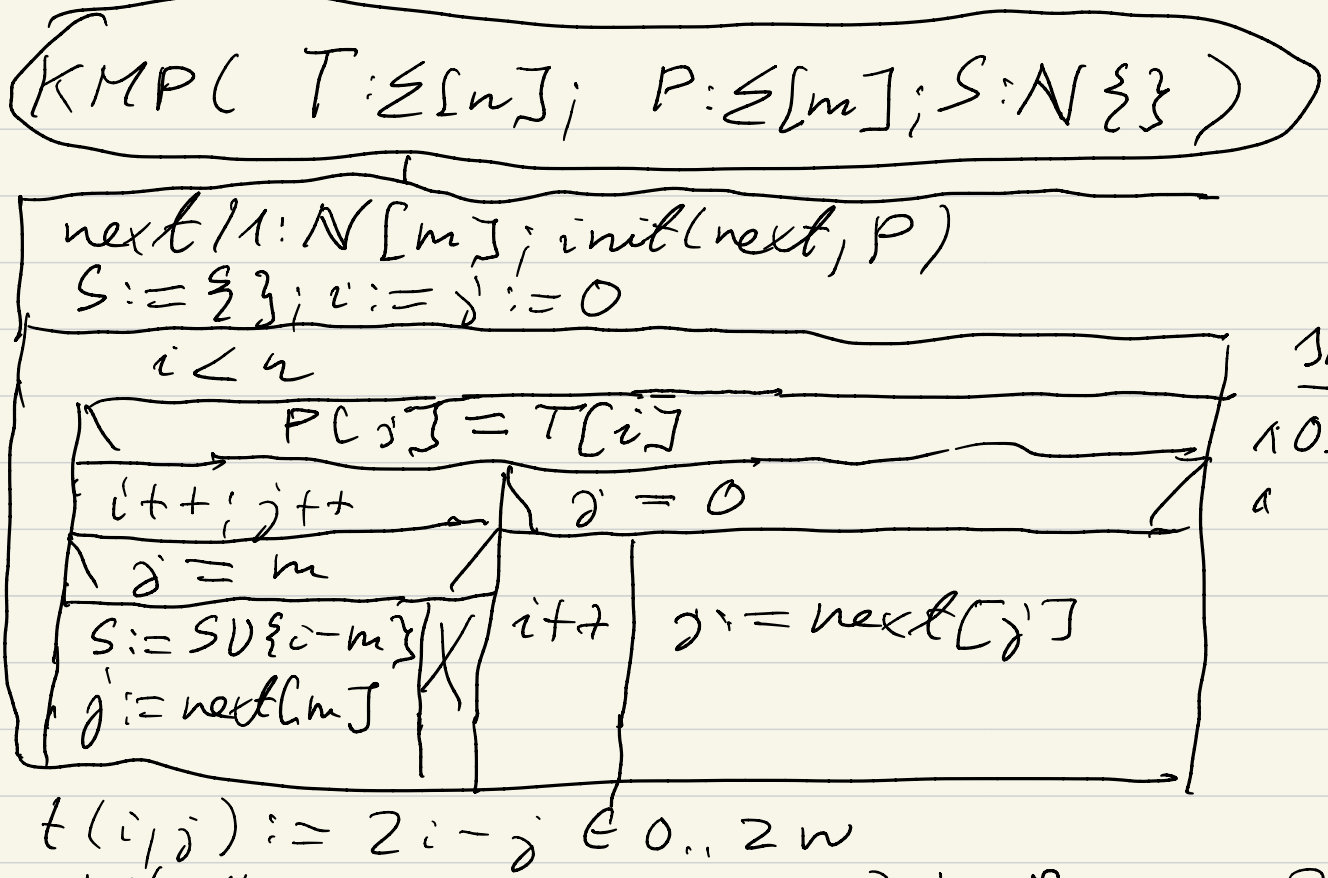
$P_{j+1} \sqsupseteq T_{i+1} \Leftrightarrow P_j \sqsupseteq T_i \wedge P[j] = T[i]$

$next(j) \in 0..j-1 \quad (j \in 1..m]$

$next(j+1) \leq next(j) + 1 \quad (j \in 1..m-1)$

$P[j-1] =$	B	A	B	A	B	B	A	B
$j =$	1	2	3	4	5	6	7	8
$next(j) =$	0	0	1	2	3	1	2	3



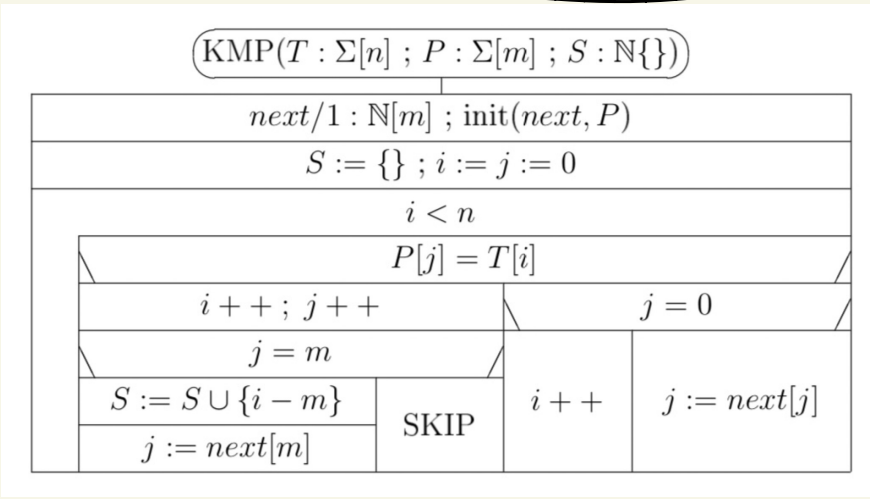

KMP(T:Σ[n]; P:Σ[m]; S:ℕ{})
next[1:ℕ[m]; init(next, P)
S := {}; i := j := 0
i < n
P[j] = T[i]
i++; j++
j = 0
j = m
S := S ∪ {i-m}
j := next[m]
i++
j := next[j]
Inv P_j ≅ T_i
∧ 0 ≤ j ≤ i ≤ n
∧ j < m
t(i,j) := 2i - j ∈ 0..2n
Initially: t(i,j) = 0
with each iteration: t(i,j) strictly increases
KMP: max. 2n iteration
KMP: max. n iteration
(initially: i = 0 ∧ finally: i = n
i increases max. by 1 in each iteration)

$mT(n,m), MT(n,m) \in \Theta(n) + \Theta(m)$

$mT_{init}(m), MT_{init}(m) \in \Theta(m)$

$m \leq n$

$mT(n,m), MT(n,m) \in \Theta(n)$



$next(1) = 0 \qquad (1 \le j < m)$

$next(j+1) = \max H(j+1) =$

$= \max \{ b \in 0..j \mid P_b \sqsupset P_{j+1} \} =$ // $i := b-1$

$= \max \{ i+1 \in 0..j \mid P_{i+1} \sqsupset P_{j+1} \} =$

$= \max(\{ i+1 \in 1..j \mid P_{i+1} \sqsupset P_{j+1} \} \cup \{0\}) =$

$= \max(\{ i+1 \in 1..j \mid \underbrace{P_i \sqsupset P_j}_{i \in H(j)} \wedge P[i] = P[j] \} \cup \{0\})$

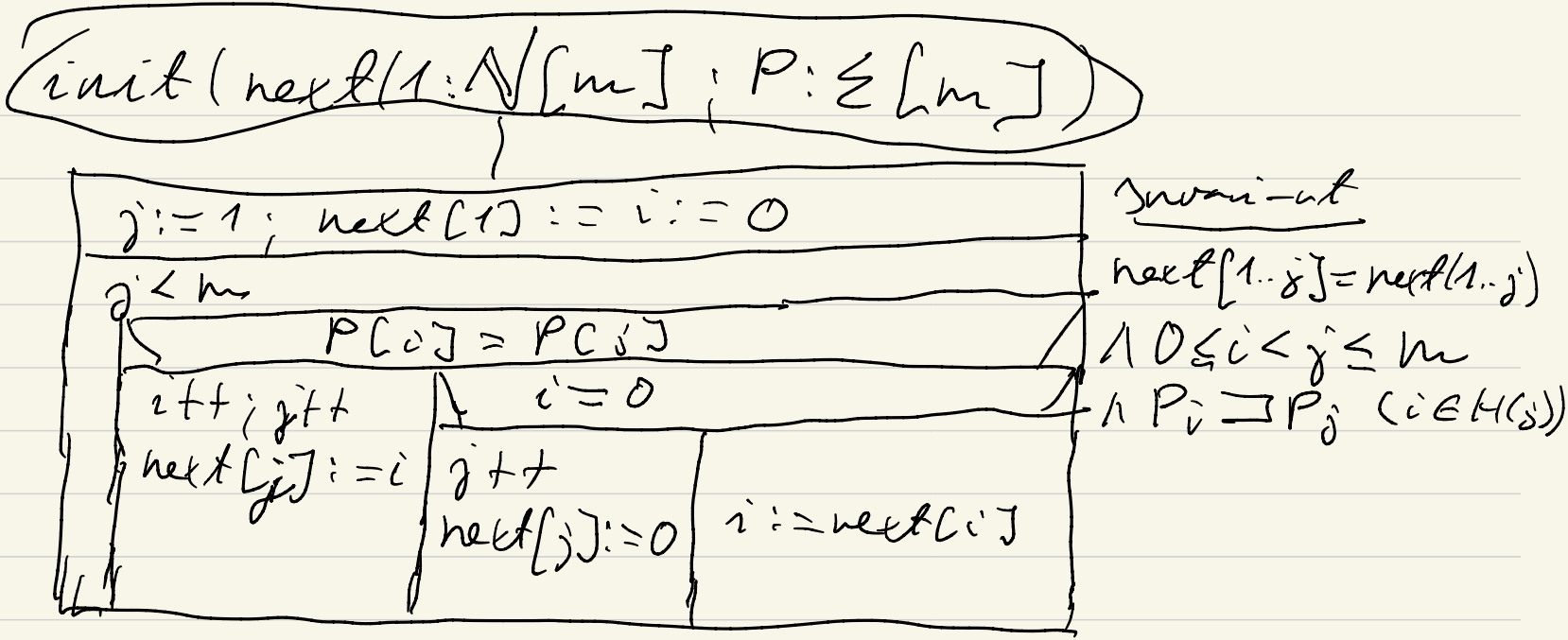
$h(j) := \{ i \in 0..j-1 \mid i \in H(j) \wedge P[i] = P[j] \}$

$h(j) \subseteq H(j)$

$$next(j+1) = \begin{cases} \max h(j) + 1 & \text{if } h(j) \neq \{\} \\ 0 & \text{if } h(j) = \{\} \end{cases}$$

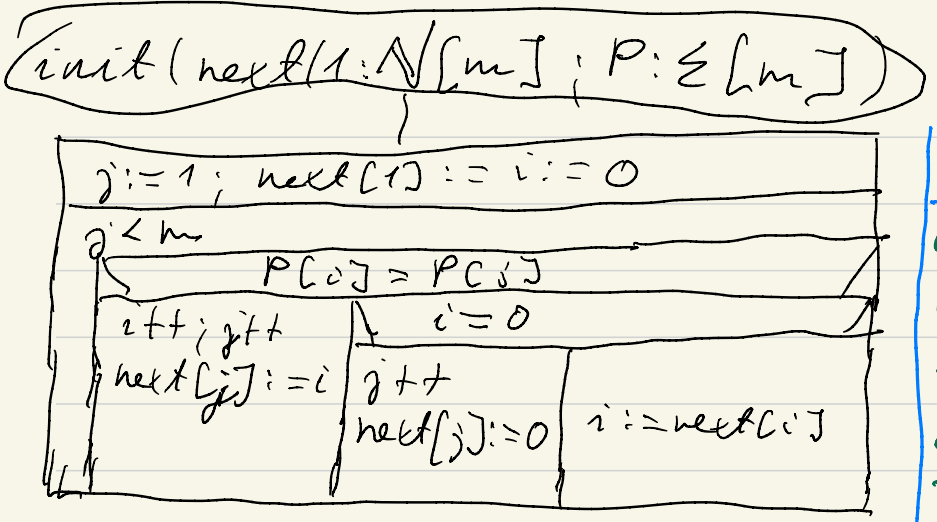
$\underline{I_+}$ $\max_{\ell+1} H(j) = next(\max_\ell H(j))$

$(\max_1 H(j) = next(j)$



to prove: $mT_{init}(m)$, $MT_{init}(m) \in \Theta(m)$

See in the printed Lecture notes !!!



P[j-1] =	B	A	B	A	B	B	A	B
j =	1	2	3	4	5	6	7	8
next[j] =	0	0	1	2	3	1	2	3

i	j	next[j]
0	1	0
0	2	0
1	3	1
2	4	2
3	5	3
1	5	3
0	5	3
1	6	1
2	7	2
3	8	3

next[j]	0	1	2	3	4	5	6	7
	B	A	B	A	B	B	A	B
0	B							
0		B						
1		B	A					
2		B	A	B				
3		B	A	B	A			
1				B	A			
0					B			
1					B	A		
2					B	A	B	