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.

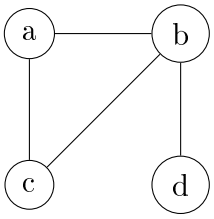
1.2 Introduction to graph representations

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

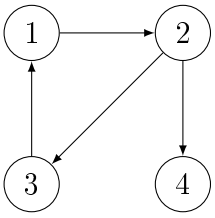
The vertices are represented with small circles. The edges are given as arrows (in case of digraphs) or simple lines (in case of undirected graphs). The labels or indices of the vertices are written into the corresponding circles.



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.)

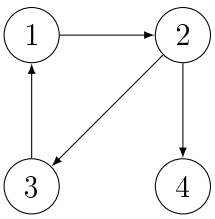
1.5 Adjacency matrix representation

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.

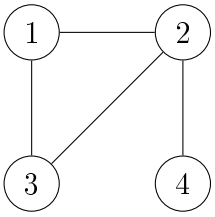
In the main diagonal, there are always zero values and the nonzero elements have value one, because we consider only simple graphs (i.e. graphs without self-loops and parallel edges).

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.

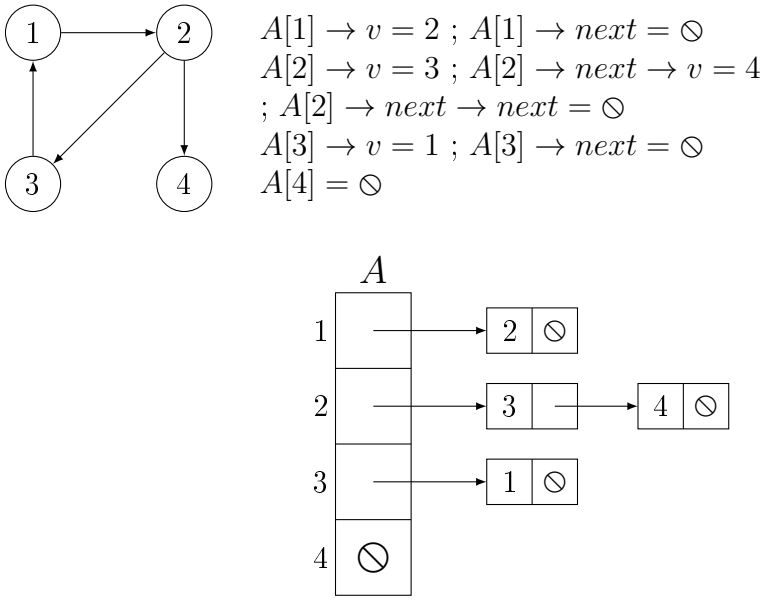


Figure 5: The same digraph is given with graphical representation on the left, and with adjacency list representation on the right.

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.

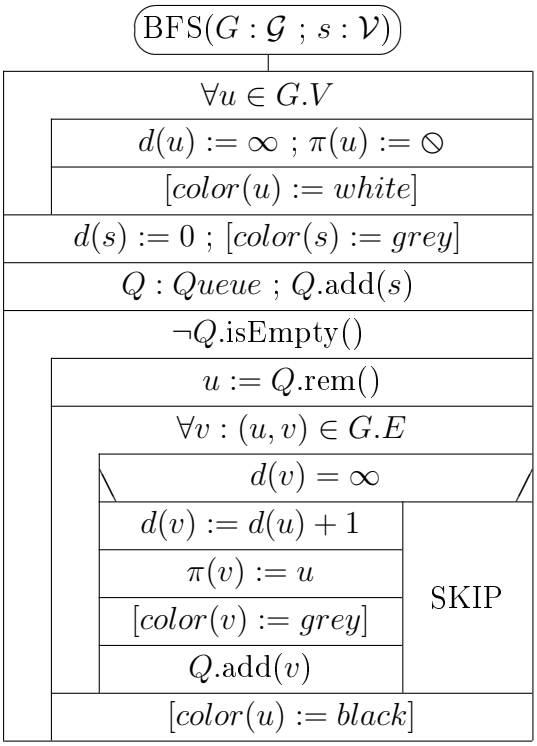
Let u be a vertex of G. The most important labels of u:

- $d(u)$ = 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.

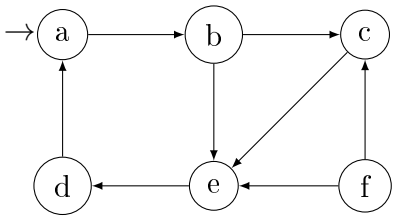
$MT(d) \in \Theta(d)$ where $d = d(v)$.

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	$\oslash$

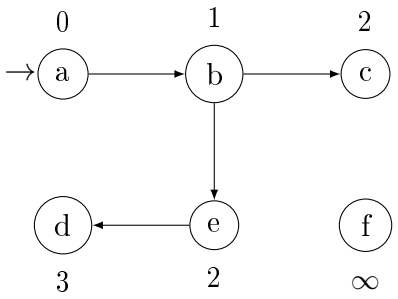
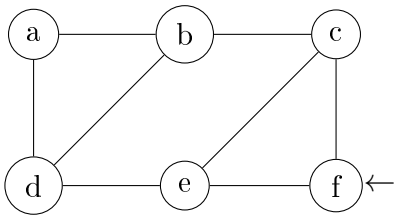


Figure 7: The breadth-first tree of BFS on the graph familiar from Figure 6 provided that s = a.

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	$\oslash$

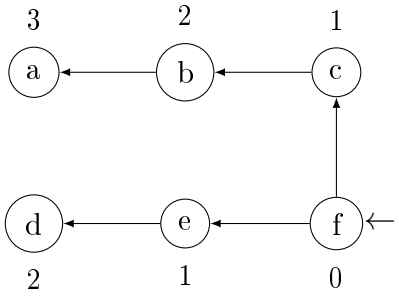


Figure 9: The breadth-first tree of BFS on the graph familiar from Figure 8 provided that $s = \text{f}$.

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

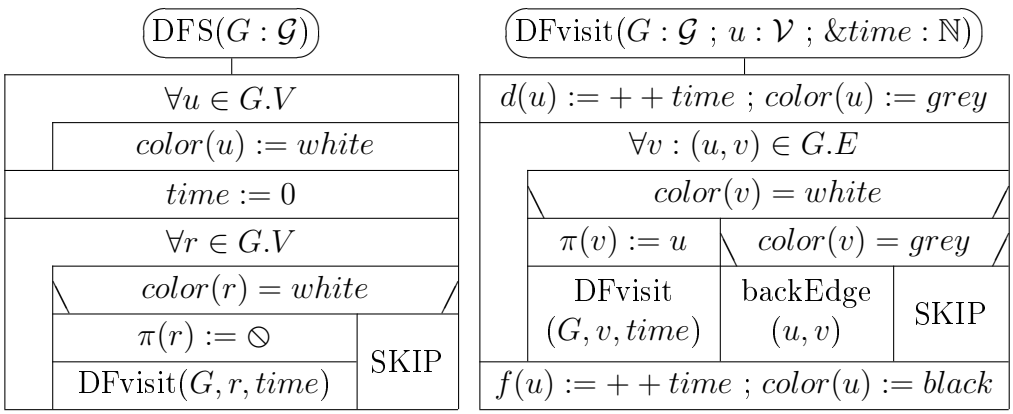
may start from any white vertex of the graph. (While illustrating DFS, we will resolve this non-determinism: we prefer the white vertex with the lowest index.) The starting point of a DFvisit is its root, because each DFvisit builds up a *depth-first tree* with this starting point as root. In a depth-first 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* $\wedge\ d(u) > d(v)$.

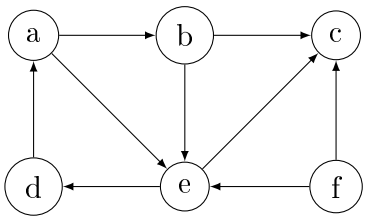
Exercise 3.11 *We are just processing edge* (u, v) *during DFS. Why* $d(u) = d(v)$ *cannot happen? In which case could it happen? In this case, how should we modify* **Definition 3.9** *so that we do not have to modify* **Theorem 3.10***?*

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.

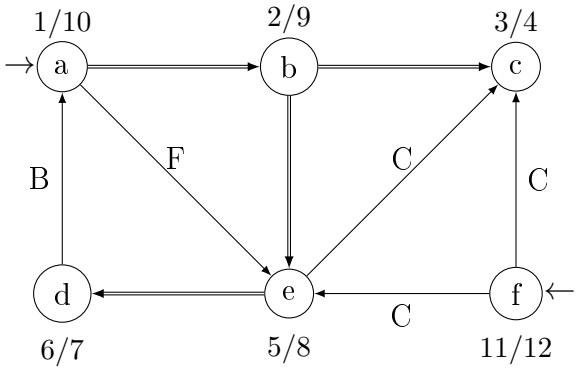


Figure 11: DFS of graph on Figure 10.

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.

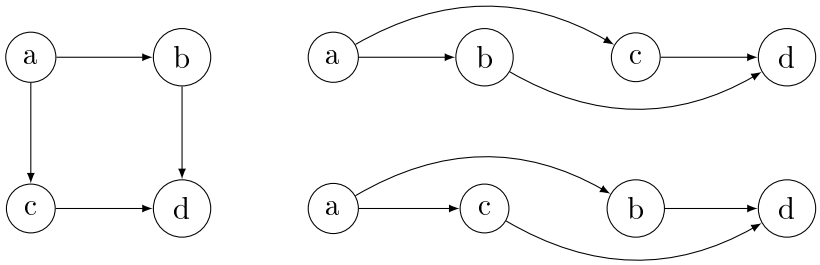


Figure 12: A DAG drawn in three different ways: it is drawn in a traditional way on the left. On the right the vertices are in two topological orders.

$\Rightarrow$ 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.

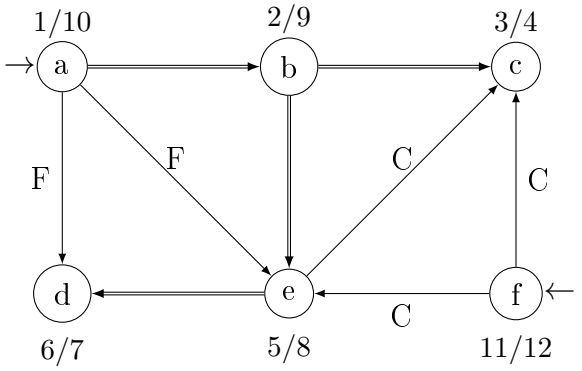


Figure 13: The vertices of a DAG sorted strictly decreasingly according to the finishing times of DFS form a topological order of the DAG. And this order can be received efficiently with pushing each vertex at the beginning of the sequence when it is finished by DFS. As a result, $\langle f, a, b, e, d, c\rangle$ 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?*