Advanced algorithms for data science

Gregory Kucherov

Gregory.Kucherov@univ-mlv.fr

Innopolis University / Université de Marne-la-Vallée

Course

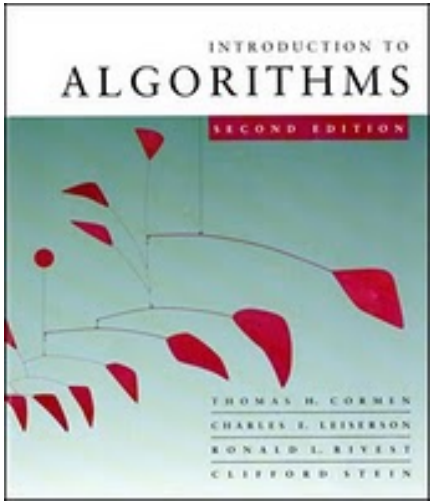
- *Purpose: a rigorous introduction to the design and analysis of algorithms*
    - Not a lab or programming course
    - Not a math course, either

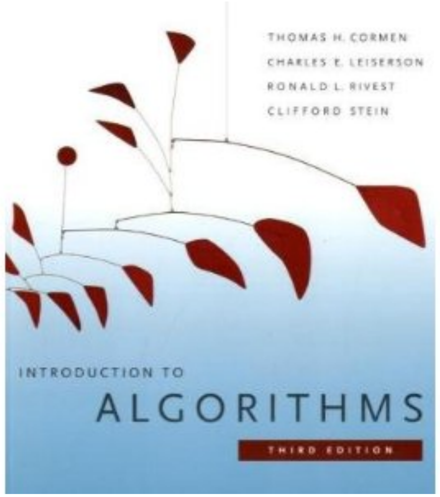
- *Prerequisites:*
    - imperative programming (C, C++, Java, …)
    - Basic data structures: lists, arrays, stacks, queues
    - Recursion
    - Big-Oh notation?
    - Sorting

- *"Free-style" pseudo-code*

Grading

- script 30%
- homework (each month) 40%
- class participation 30%





CLRS = Cormen & Leiserson & Rivest &Stein

Some other good algorithm textbook:

- *Steven Skiena*, The Algorithm Design Manual, 2nd Edition, Springer, 2008 [a bit advanced?]
- *Jon Kleinberg* and *Éva Tardos*, Algorithm Design, MIT Press 2005
- *Robert Sedgewick* and *Kevin Wayne*, Algorithms, Addison-Wesley, 4th Edition, 2011 [for beginners, Java-oriented]
- А.Шень, Программирование: теоремы и задачи, 2е изд, МЦНМО, 2004

Some topics addressed in the course

- **Graphs**
  - Shortest paths
  - Spanning trees
  - Flows
- **Search trees**
  - Arbres rouges-noirs
- **Sequence algorithms**
  - String matching
  - Suffix trees
  - Text compression
- **Dynamic programming**
  - Sequence alignment
  - Hidden Markov models
- **Advanced data structures**
  - Union-Find, Bloom filters…
- **NP-completeness**
  - P and NP
  - NP-complete problems

How to measure the efficiency of algorithms?

- **Efficiency (*in this course*) = TIME and SPACE**
  - other possible measures of efficiency:
    - accuracy, precision

- ***In this course*: RAM model of computation**
    - all memory accesses have equal unit cost
    - no parallel execution
    - unit cost (O(1)) basic operations (unless bits are explicitly manipulated)
  - time = # of RAM operations
  - space = # of computer words
  - other possible parameters: disk accesses, cache misses, probe model, query complexity

How to measure the efficiency of algorithms? (cont)

- **Algorithms solve *mass problems***
  - *n*: input size (in computer words or bits)
  - time/space as **a function of *n***

- ***In this course*: WORST-CASE complexity**
  - other possibility: average-case complexity

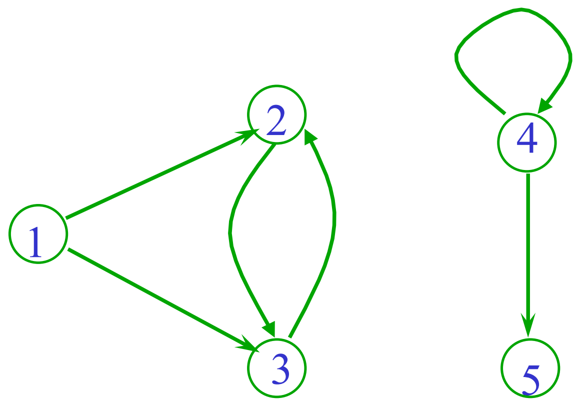
Graphs

Directed graph $G = (S, A)$

S finite set of *nodes (vertices)*

$A \subseteq S \times S$ set of *edges (arcs)*,

i.e., a relation on S



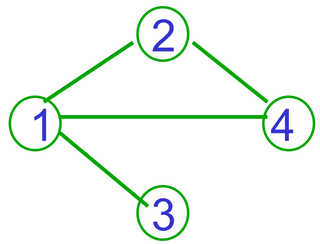
$S = \{ 1, 2, 3, 4, 5 \}$

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

Undirected graph $G = (S, A)$

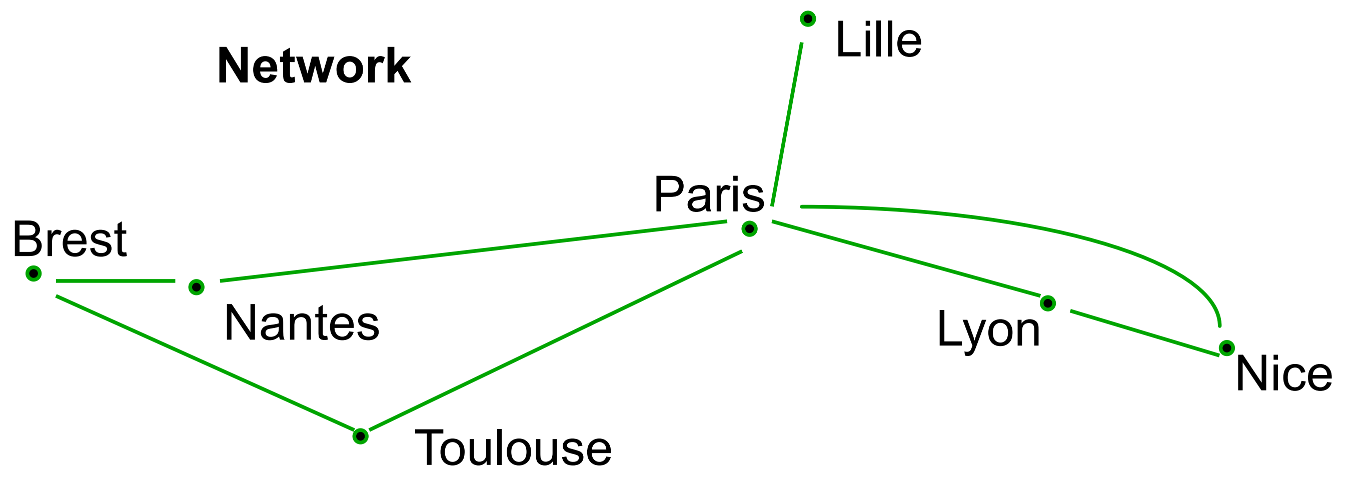
A set of *edges (arcs)*,

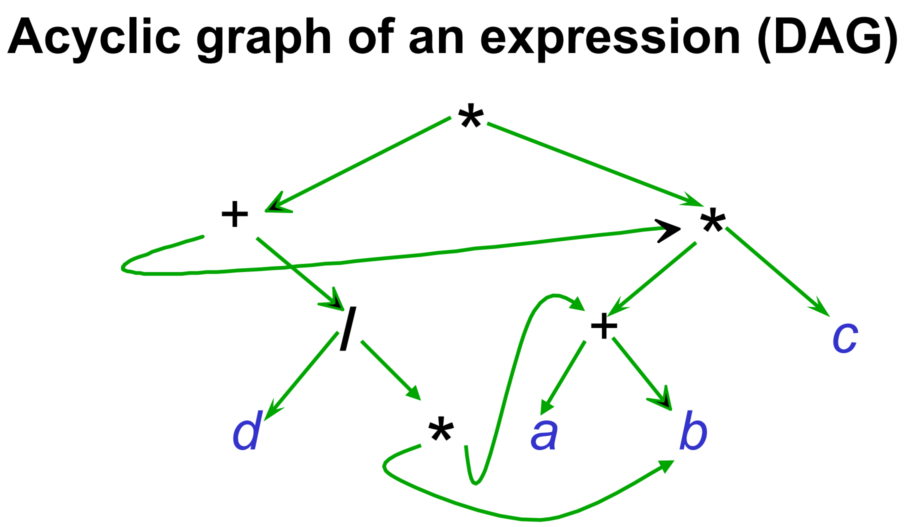
symmetric relation



$S = \{ 1, 2, 3, 4 \}$

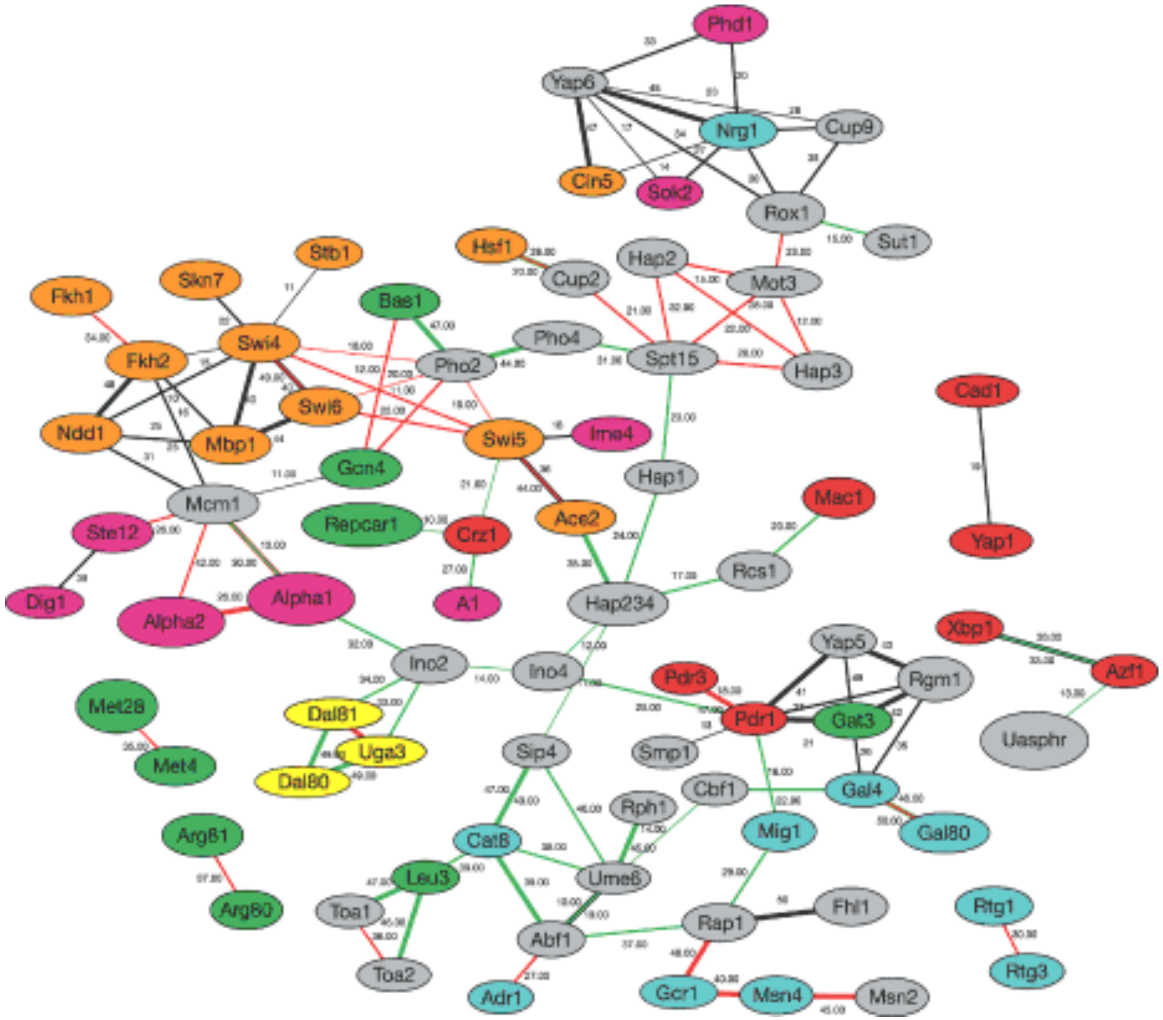
$A = \{ \{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 4\} \}$


Network
Lille
Paris
Brest
Nantes
Lyon
Nice
Toulouse

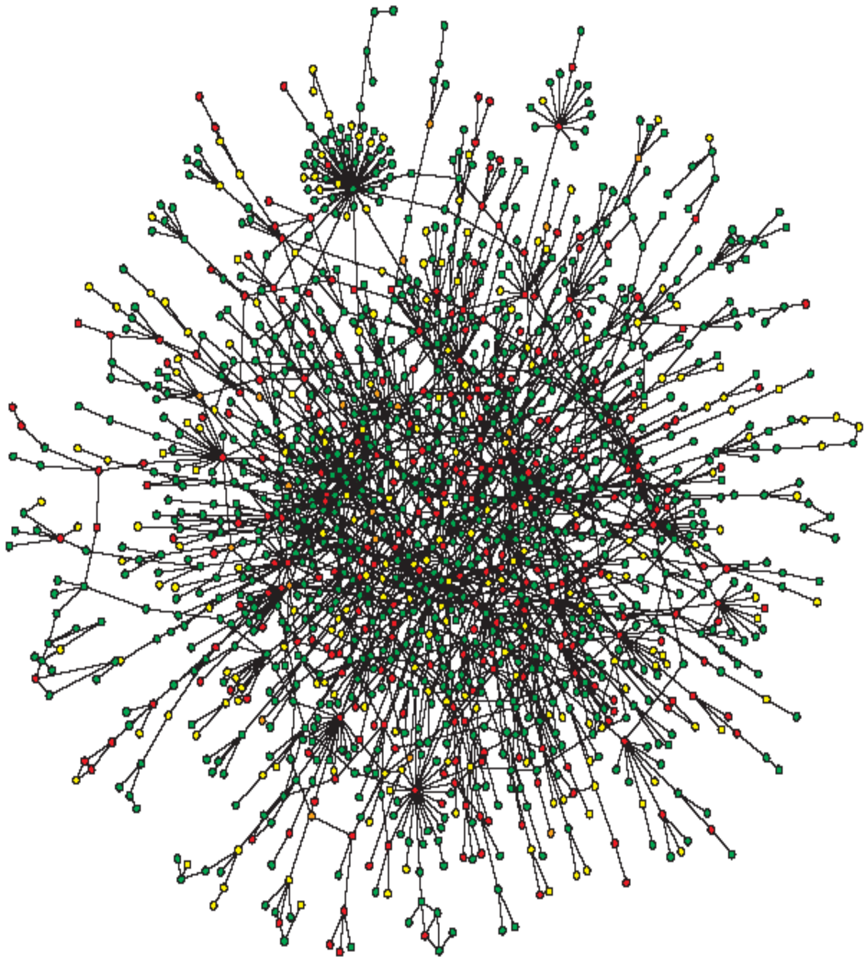

Acyclic graph of an expression (DAG)
*
+
*
/
+
c
d
*
a
b

((a+b)*c+d/(b*(a+b))) * (a+b)*c

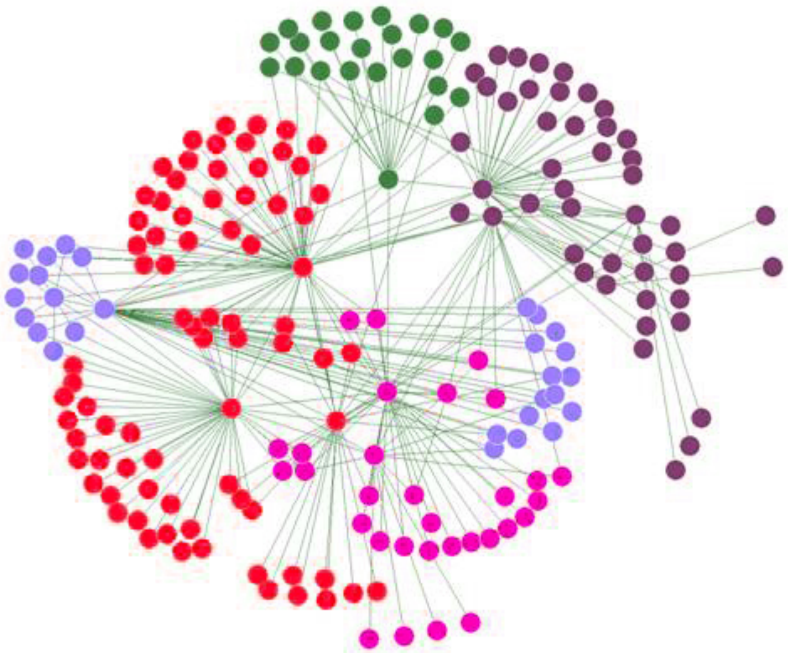
Gene regulation network in biology



Protein-protein interaction network (in yeast)



Social networks



Algorithms

Exploration

- Depth-first or breadth-first traversal
- Topological sorting
- Strongly connected components, ...

Path computation

- Transitive closure
- Minimal cost path
- Eulerian and Hamiltonian paths, ...

Spanning trees

- Kruskal and Prim algorithms

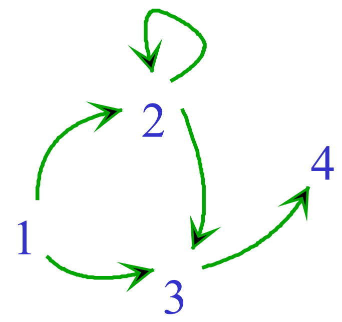
Networks

- Maximal flow

Others

- Graph coloring
- Planarity testing, ...

Terminology



Graph : $G = (S, A)$
Edge : $(s, t) \in A$ t adjacent to s, t successor of s
Successors of s : $A(s) = \{ t \mid (s, t) \in A \}$
(Self-)loop : $(t, t) \in A$

Paths

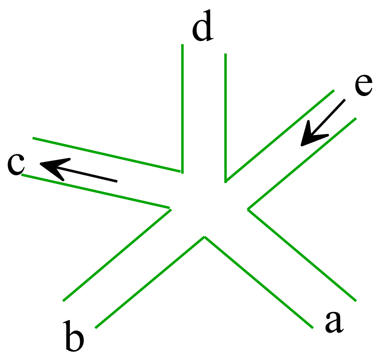
Path : $c = ((s_0,s_1), (s_1,s_2), \ldots, (s_{k-1},s_k))$ where $(s_{i-1},s_i) \in A$
source = s_0
end = s_k
length = k

((1,2), (2,2), (2,3), (3,4))

Cycle : path where source and end nodes coincide

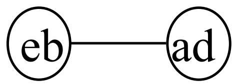
Traffic light problem

Graph to model a problem

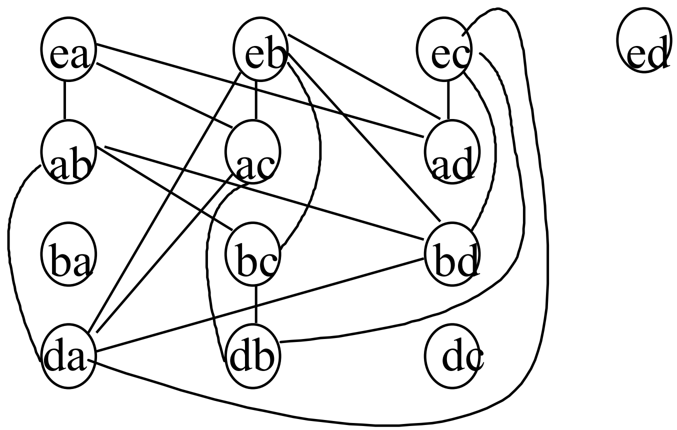


 one-way traffic

possible crossing



conflicting crossings



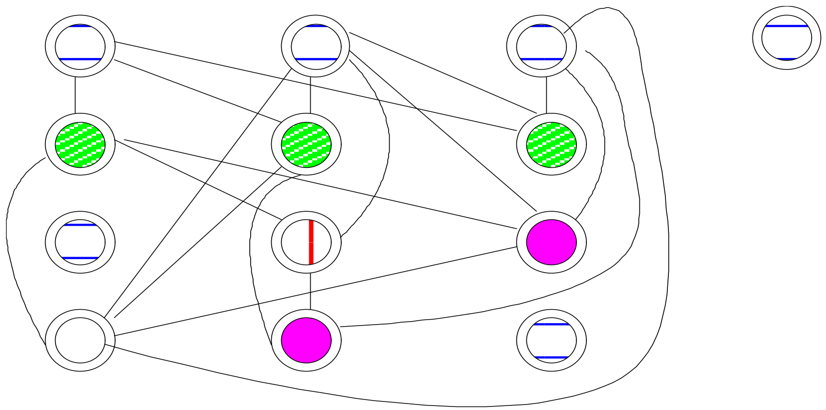
Coloring

$G = (S,A)$

coloring $f : S \rightarrow C$ such that $(s,t) \in A \Rightarrow f(s) \neq f(t)$

$\mathrm{Chr}(G) = \min_{f} |f(S)|$, chromatic number of G

$\mathrm{Chr}(G) = 4$



color = set of compatible crossings

Coloring algorithm

$G = (S, A)$ $S = \{ s_1, s_2, \ldots, s_n \}$
G without loops !

fonction sequential-coloring (G graph) : int ;
begin
  for $i \leftarrow 1$ **to** n **do {**
    $c \leftarrow 1$;
    while there exists t adjacent to s_i with $f(t) = c$ **do**
      $c \leftarrow c + 1$;
    $f(s_i) \leftarrow c$;
  }
  return max ($f(s_i)$, $i = 1, \ldots, n$) ;
end

Running time: $O(n^2)$ Computing Chr (G) : $O(n^2\, n!)$
(apply sequential-coloring to all permutations of S)
No known polynomial-time algorithm !

Representations

$G = (S, A)$ $S = \{1, 2, ..., n\}$

List of edges

compact representation

indexing (hashing) by edge source (cf below)

Adjacency matrix

using matrix operations

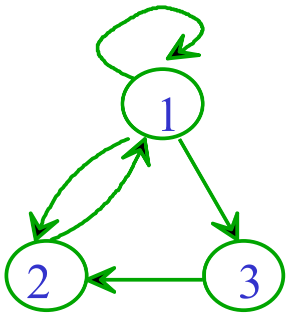
usually quadratic processing time

Adjacency list

reduces the size if $|A| << (|S|)^2$

usual processing time : $O(|S| + |A|)$

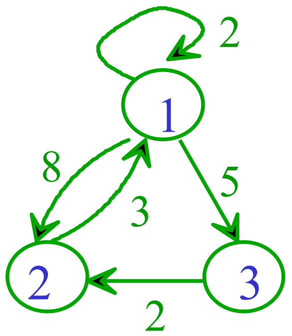
Adjacency matrix



$S = \{ 1, 2, 3 \}$
$A = \{ (1,1), (1, 2), (1, 3), (2, 1), (3, 2)\}$

$$M = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

$M [i, j] = 1$ iff j is adjacent to i



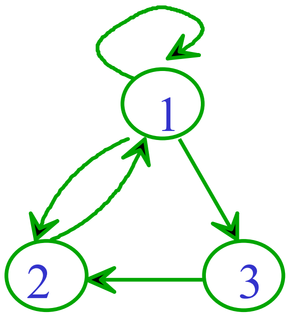
$$V = \begin{pmatrix} 2 & 8 & 5 \\ 3 & 0 & 0 \\ 0 & 2 & 0 \end{pmatrix}$$

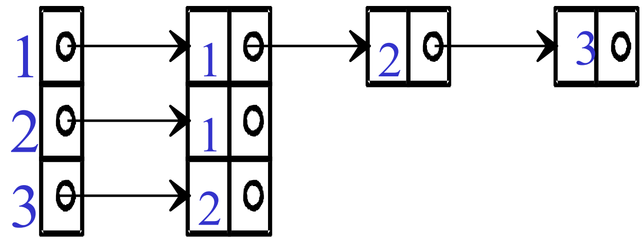
weight: $v : A \longrightarrow X$

Adjacency lists

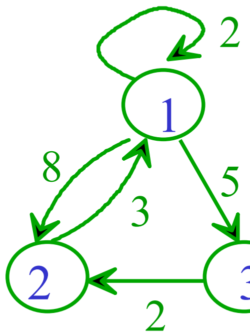
$S = \{\ 1,\ 2,\ 3\ \}$

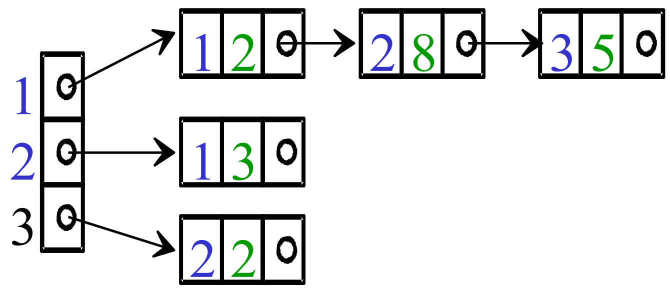
$A = \{\ (1,1),\ (1,\ 2),\ (1,\ 3),\ (2,\ 1),\ (3,\ 2)\}$





Lists of $A(s)$





weight: $v : A \longrightarrow X$

Graph traversals

$G = (S, A)$
Traverse G = visit all nodes (or all edges)

Used in

- cycle search
- topological sorting
- search for connected components
- processing nodes (coloring, ...)
  or edges (weighing, ...)

Depth-first or breadth-first traversals

- extensions of tree traversals

Depth-first traversal

Node marking

for each node *s* of *G* **do**

  visited[*s*] $\longleftarrow$ false ; //*s* is *white*

for each node s of G **do**

  if not visited [*s*] **then** DFT(*s*) ;

procedure DFT(*s* node of *G*) ;

begin

  opening action on *s* ;

  visited[*s*] $\longleftarrow$ true ; //*s* becomes *yellow*

  for each *t* successor of *s* **do {**

    processing edge (*s*,*t*) ;

    if not visited[*t*] **then** DFT(*t*) ;

  }

  closing action on *s* ; //*s* becomes *red*

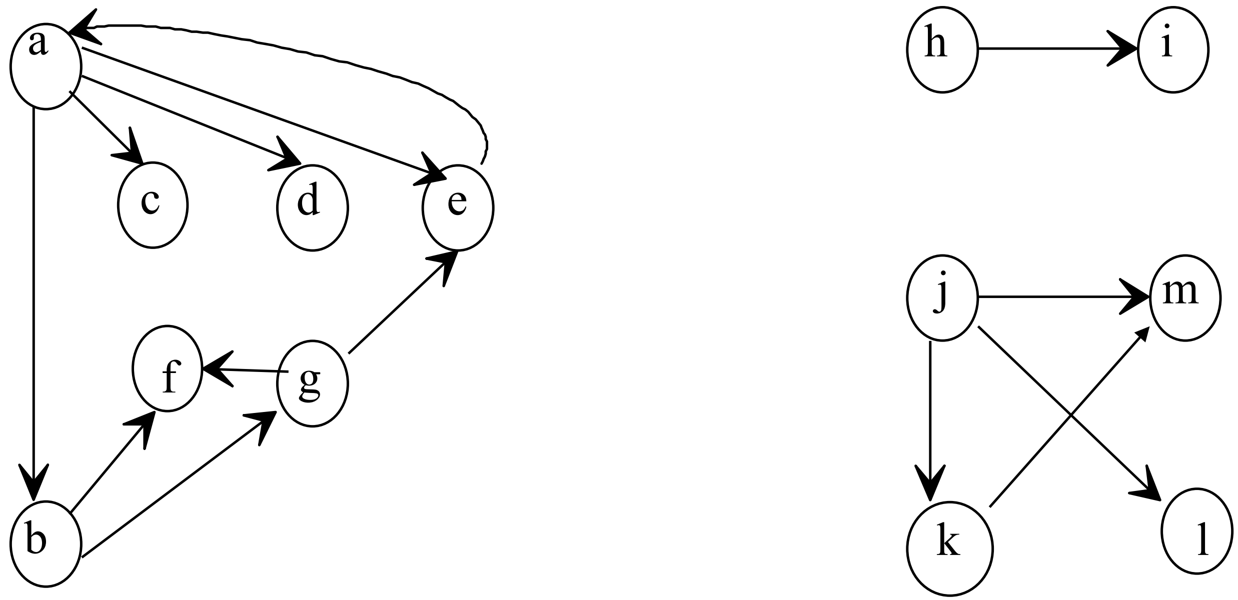
end

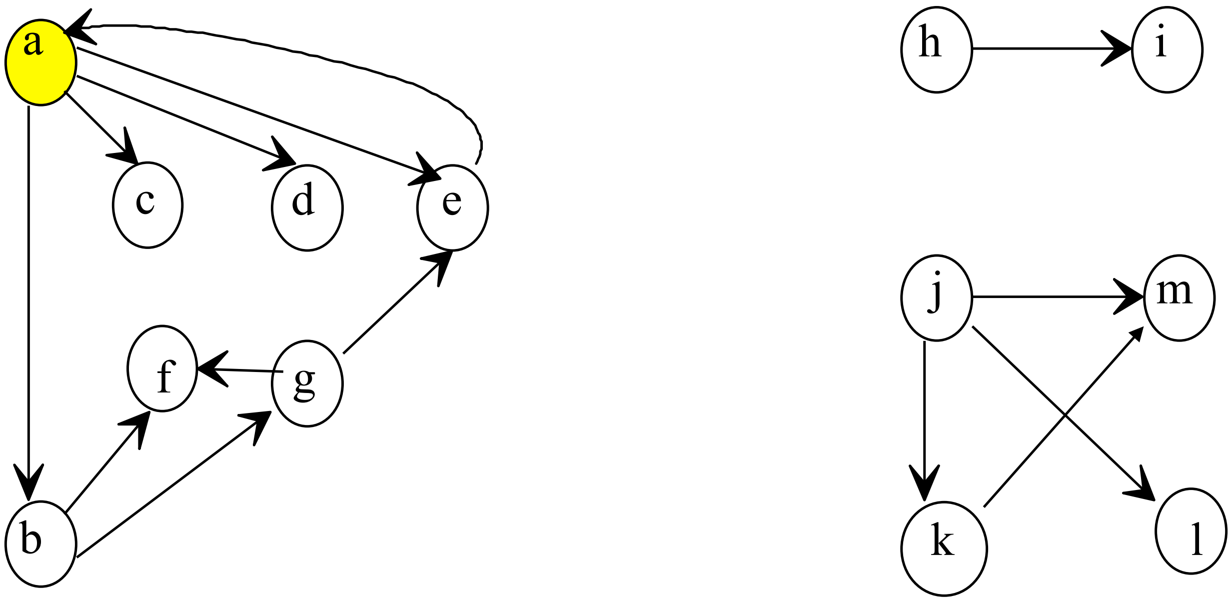
Three states of a node

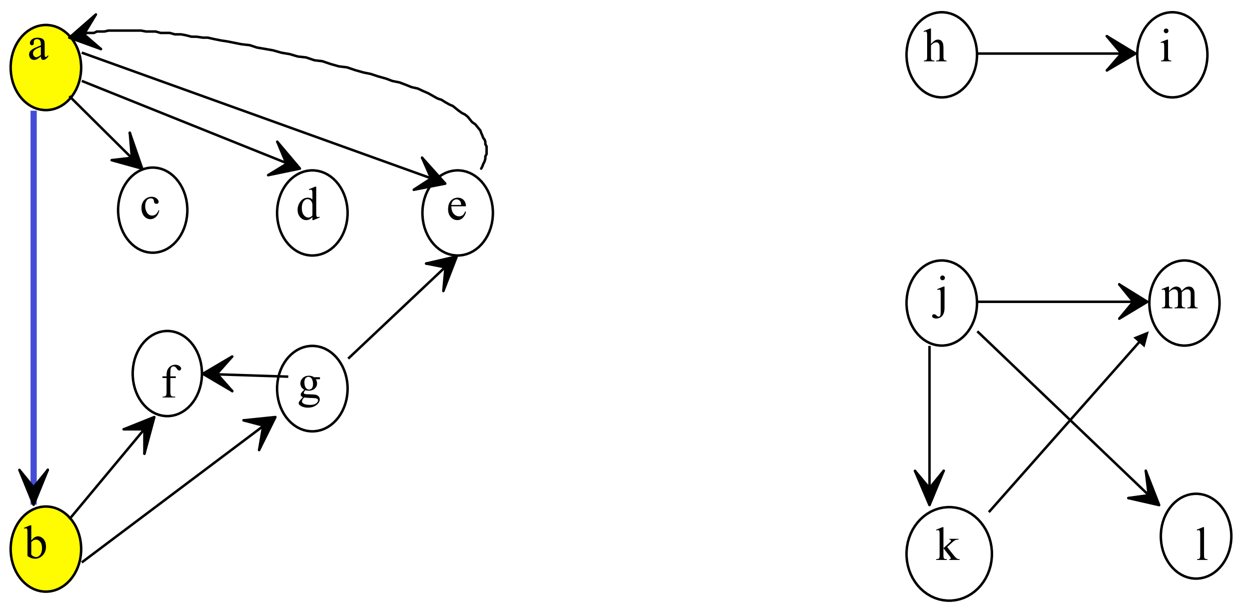
In the course of the traversal:

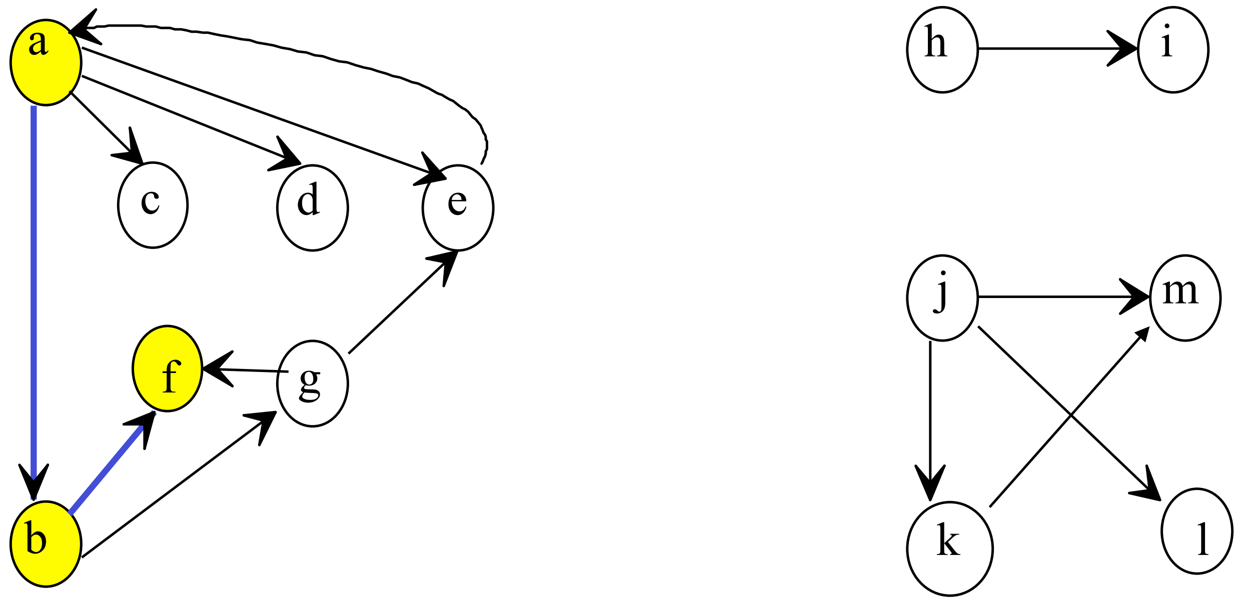
state [s] = white	s : has not yet been discovered
state [s] = yellow	s : under processing
state [s] = red	s : processing finished

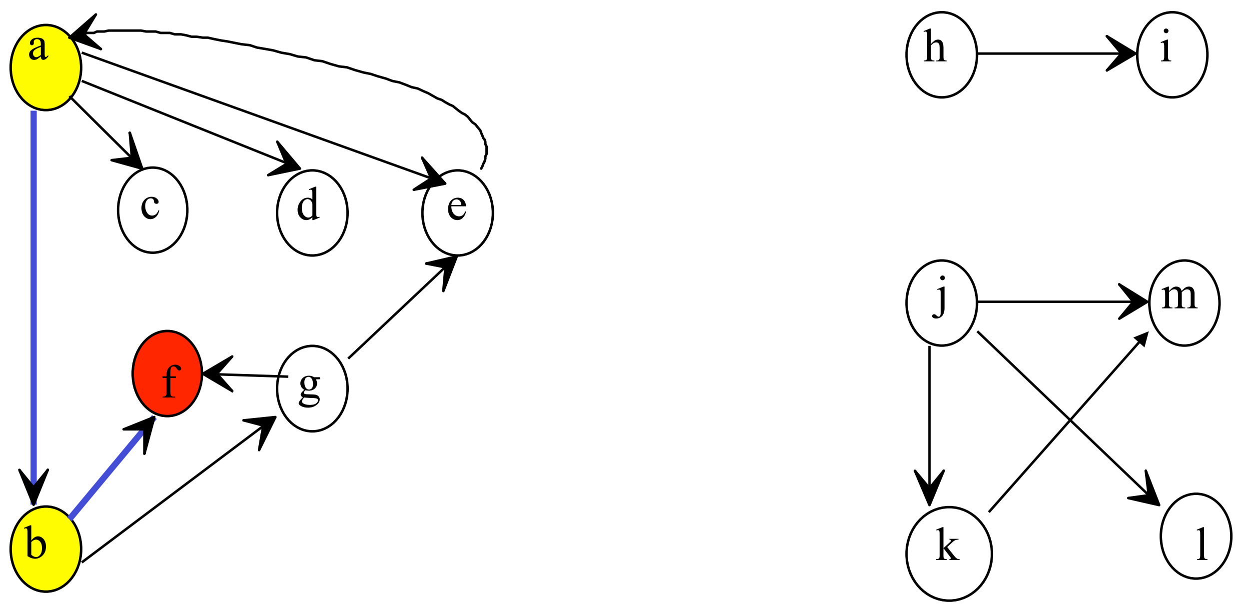
UMLV ©



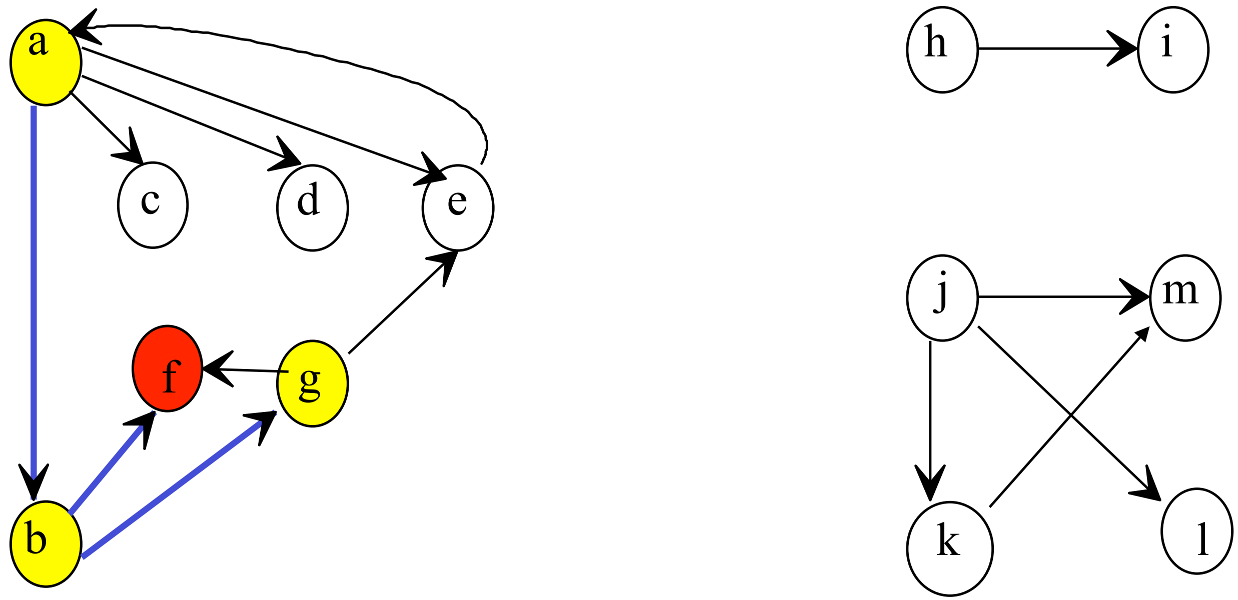

a
b
c
d
e
f
g
h
i
j
k
l
m

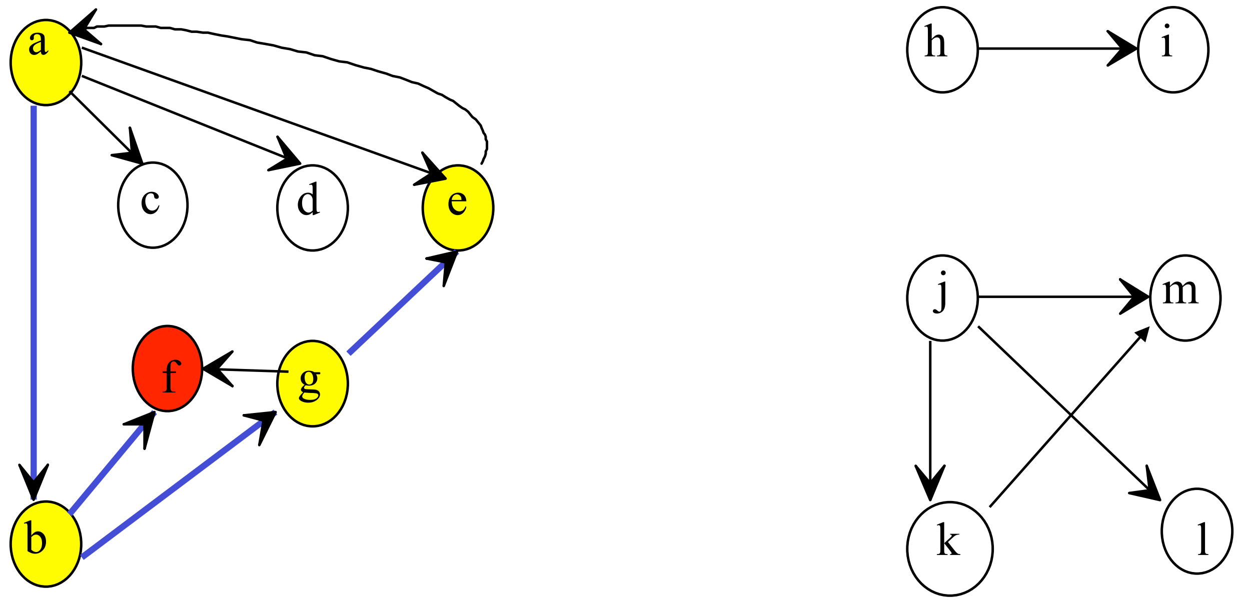

a
b
c
d
e
f
g
h
i
j
k
l
m

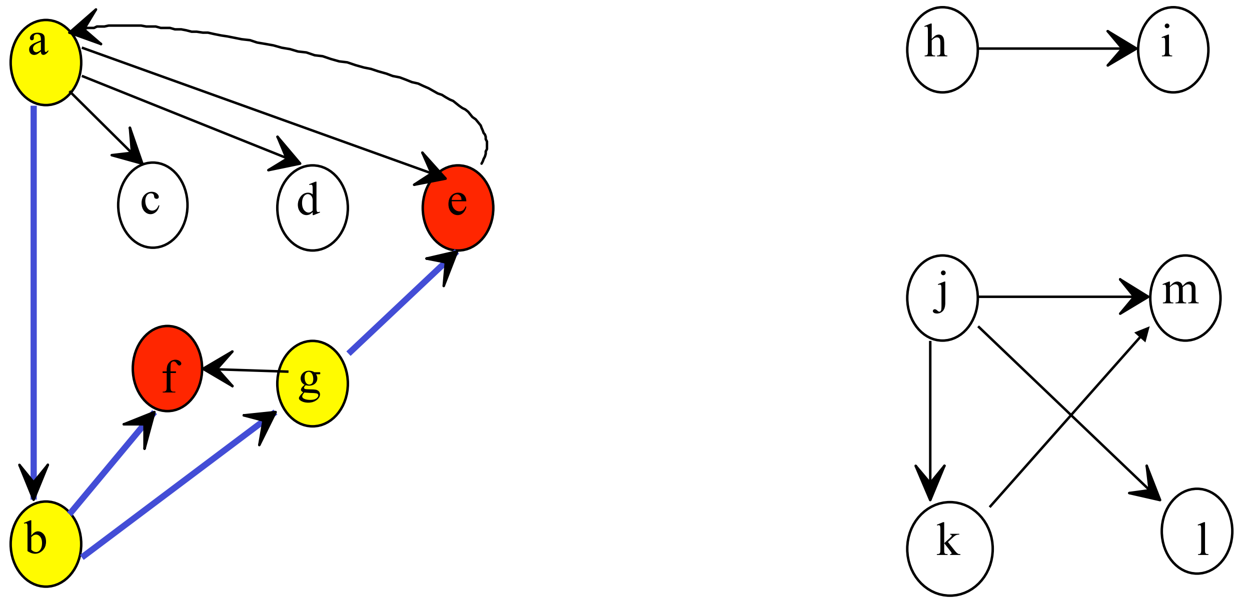

a
b
c
d
e
f
g
h
i
j
k
l
m

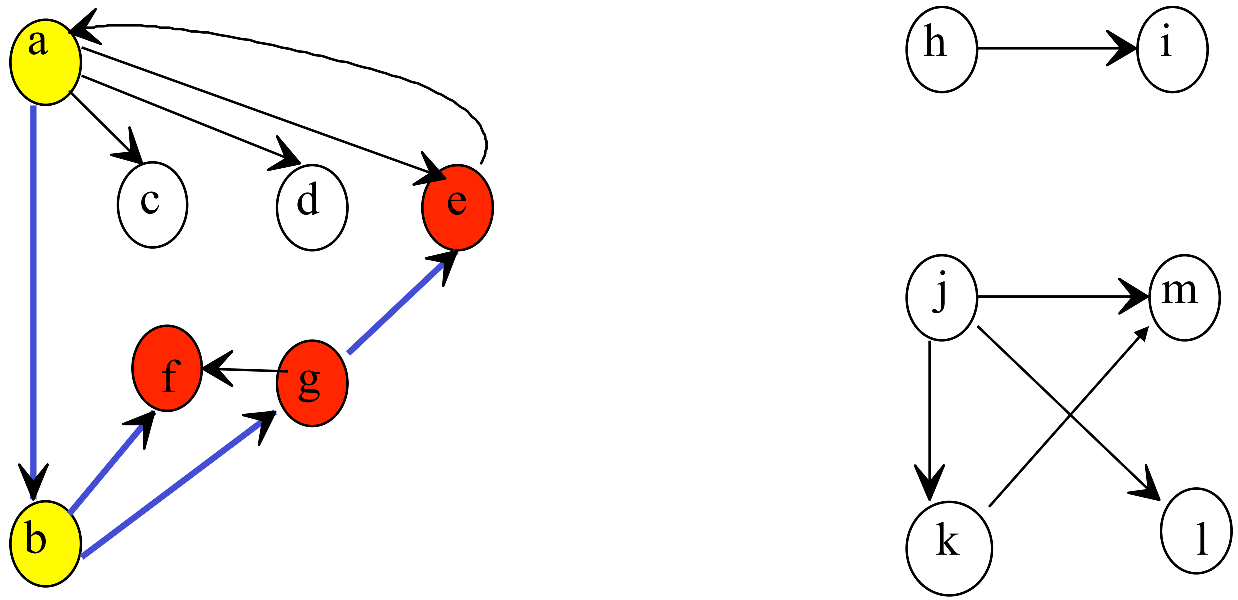

a
c
d
e
f
g
b
h
i
j
m
k
l

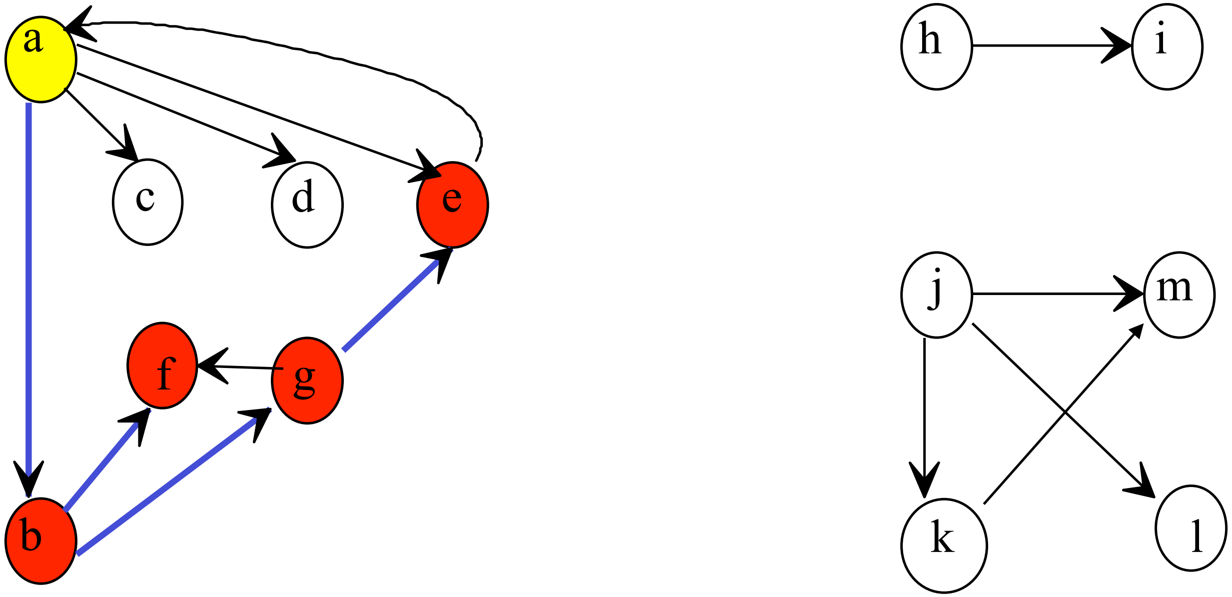
UMLV ©



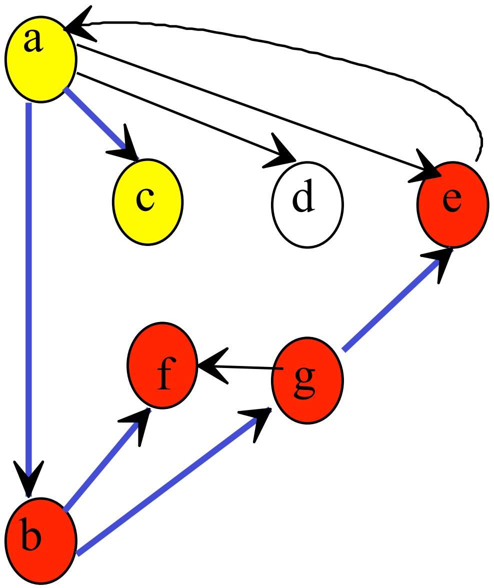

a
b
c
d
e
f
g
h
i
j
k
l
m

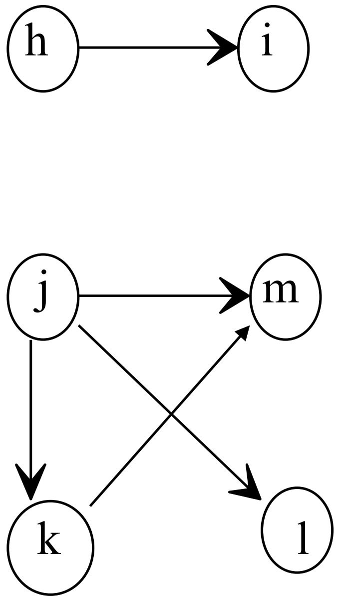

a
c
d
e
f
g
b
h
i
j
m
k
l

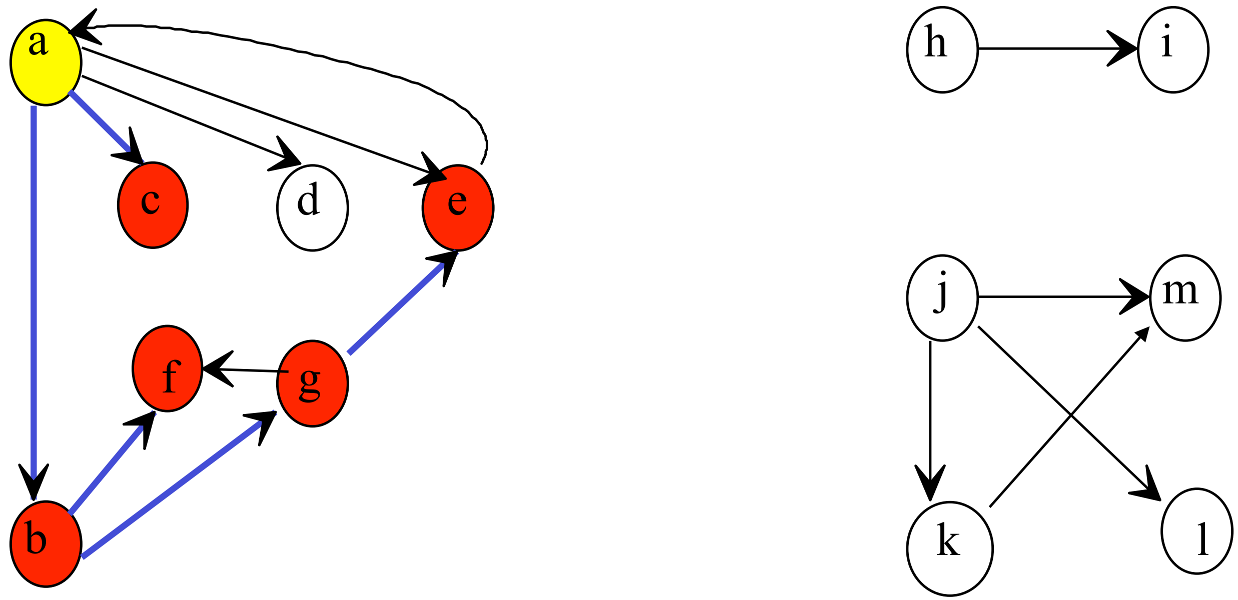

a
b
c
d
e
f
g
h
i
j
k
l
m

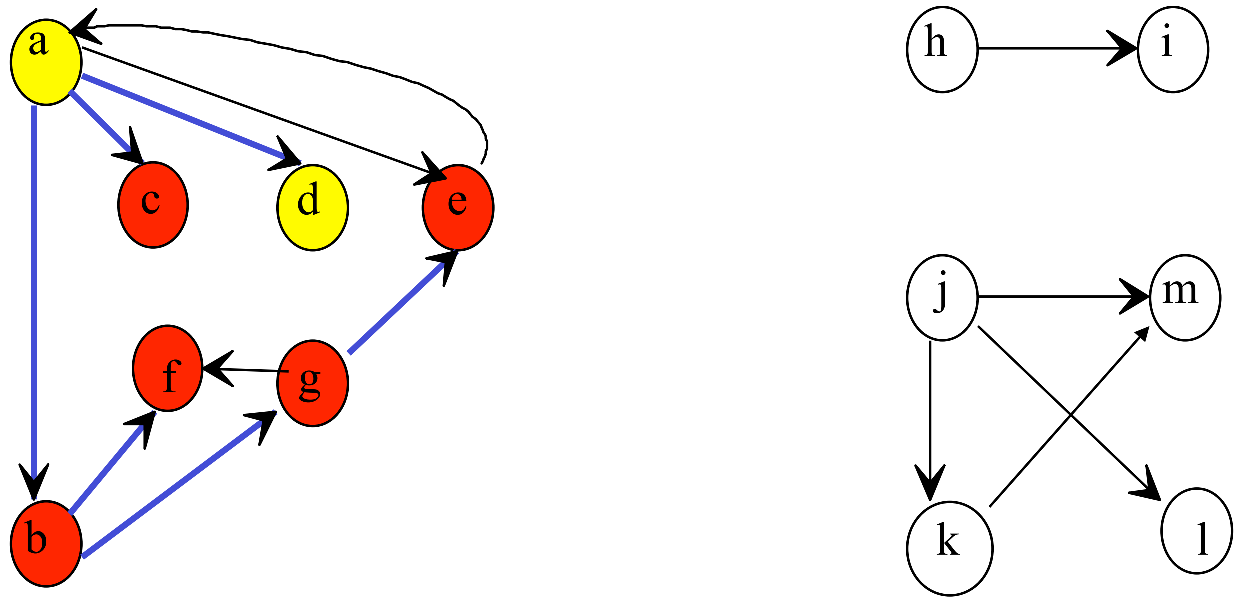

a
b
c
d
e
f
g
h
i
j
k
l
m

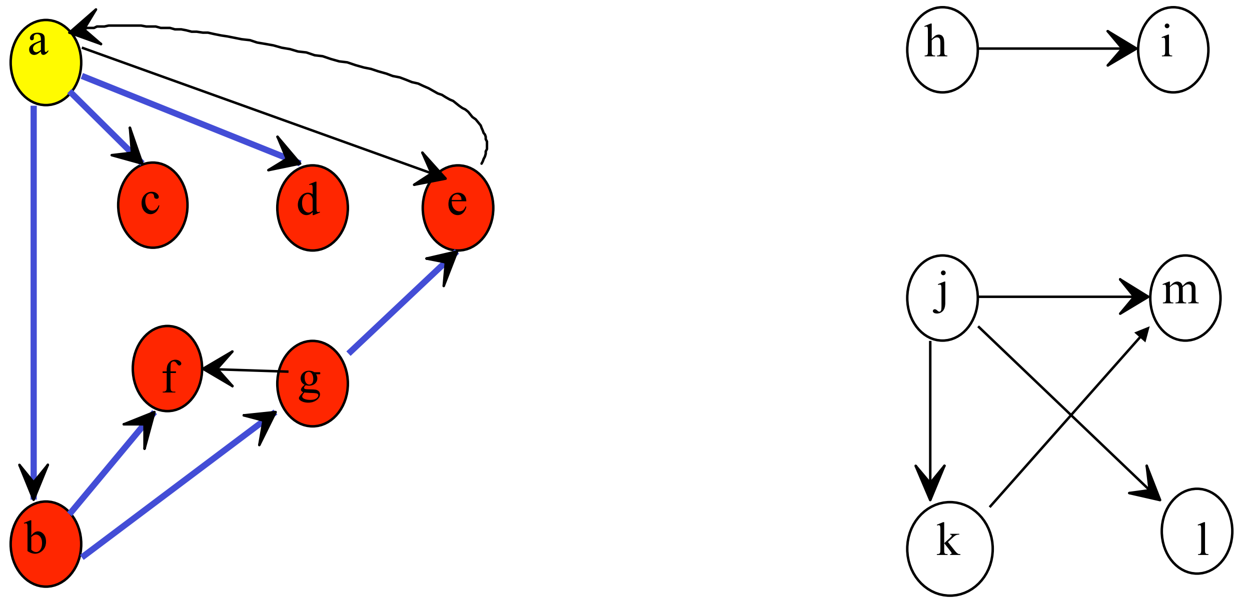
UMLV ©



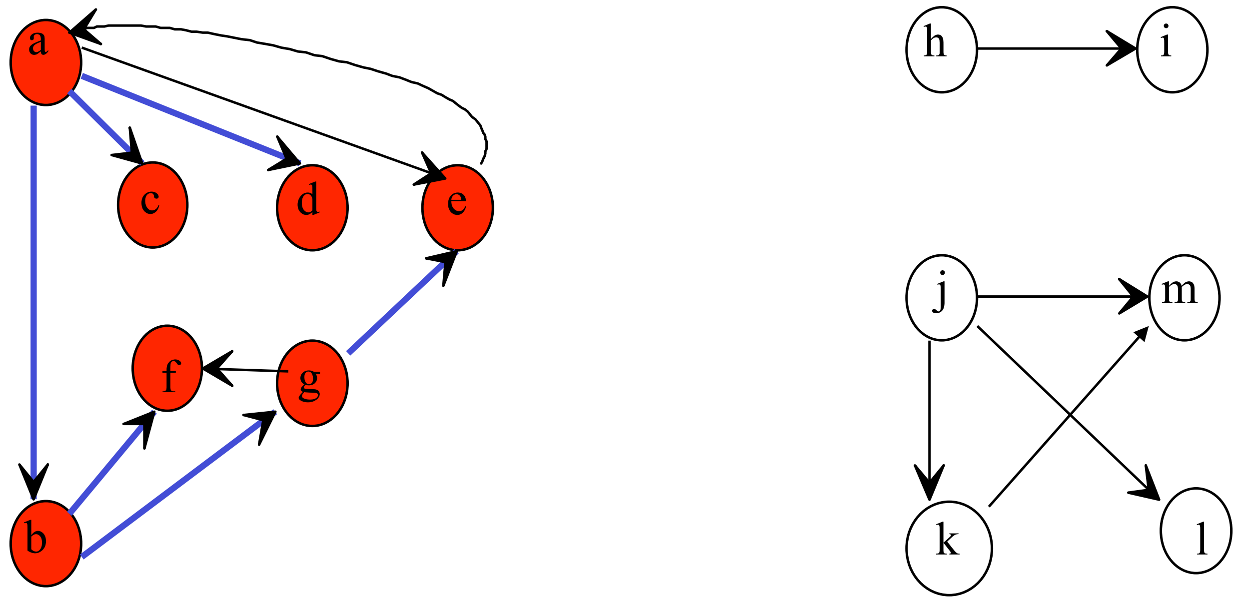


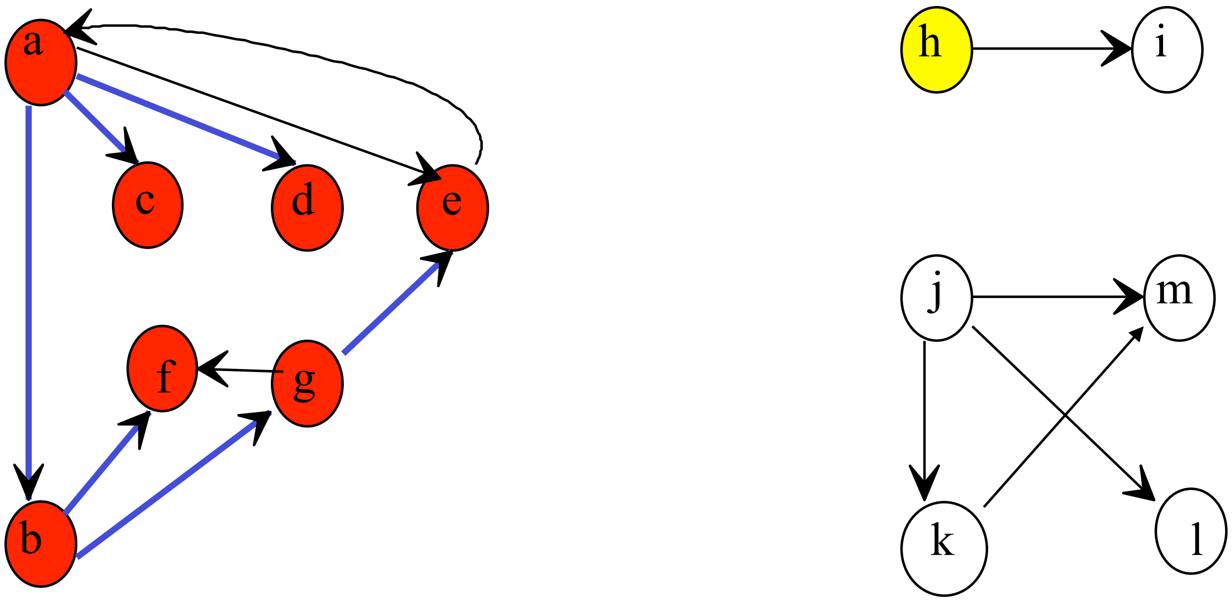

a
b
c
d
e
f
g
h
i
j
k
l
m

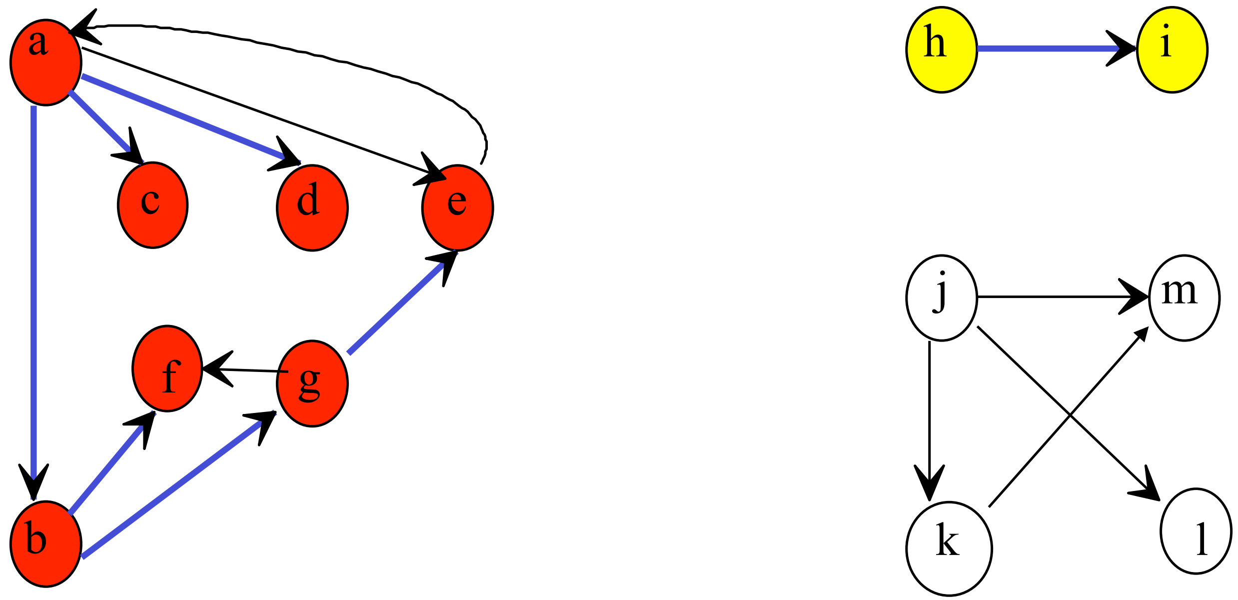

a
b
c
d
e
f
g
h
i
j
k
l
m


a
b
c
d
e
f
g
h
i
j
k
l
m

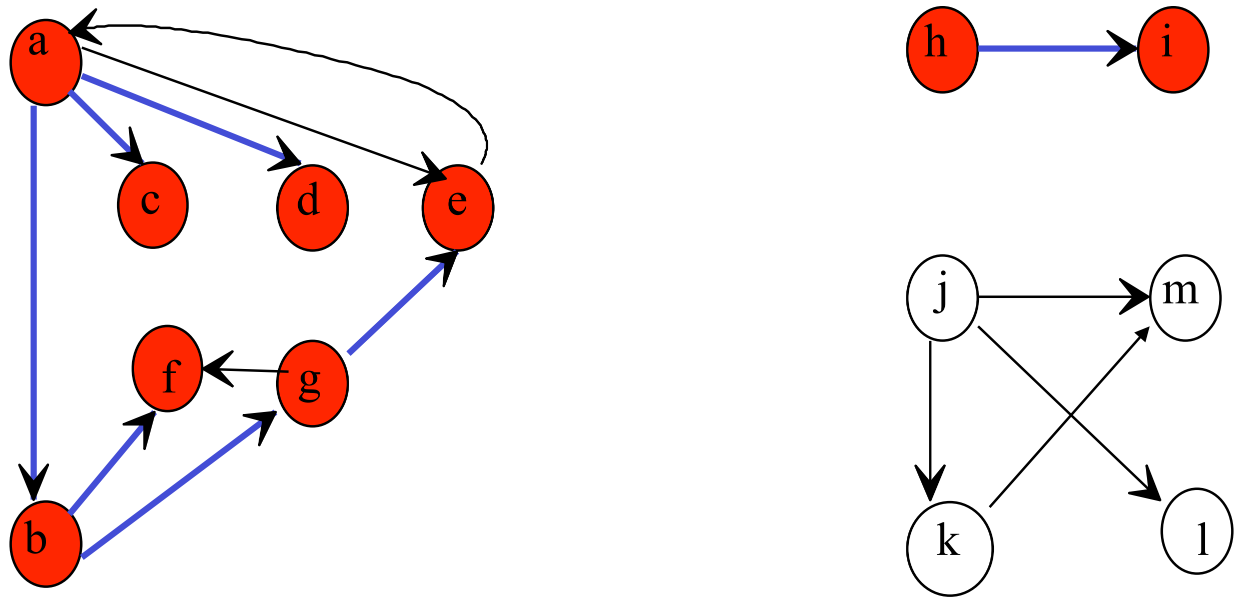
UMLV ©



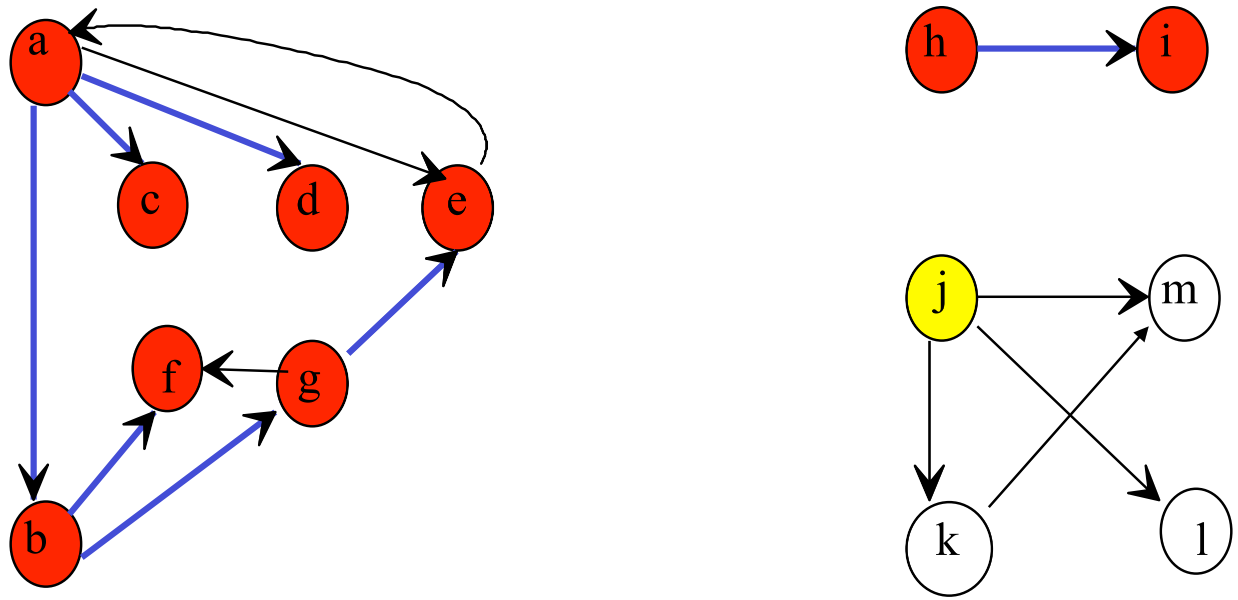

a
b
c
d
e
f
g
h
i
j
k
l
m

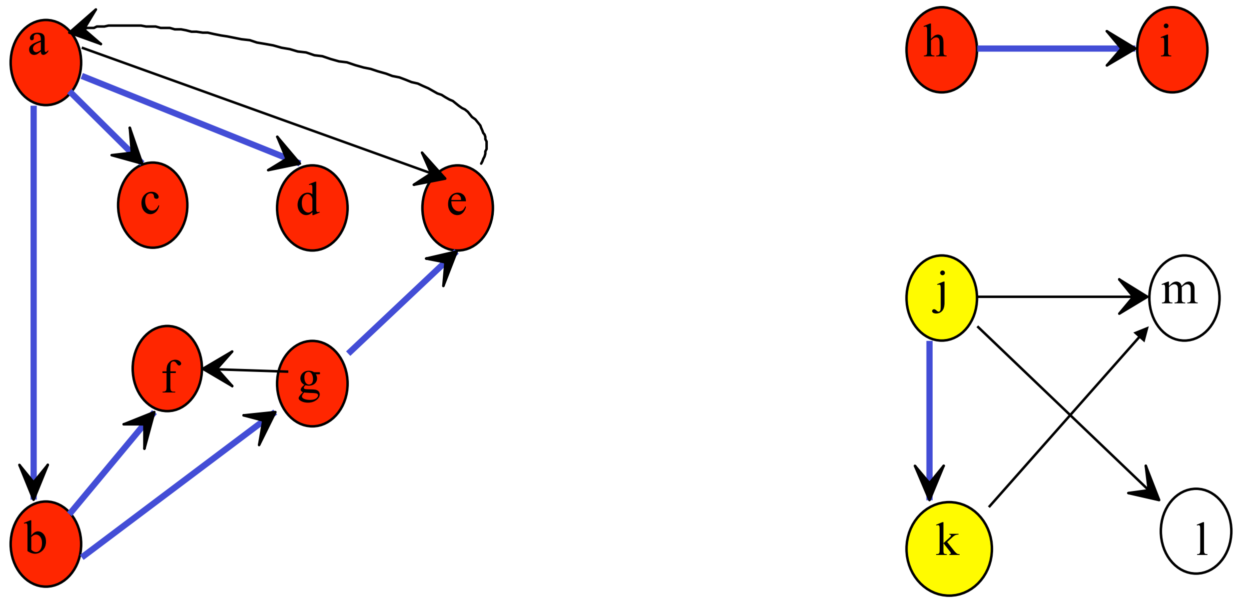

a
b
c
d
e
f
g
h
i
j
k
l
m

UMLV ©

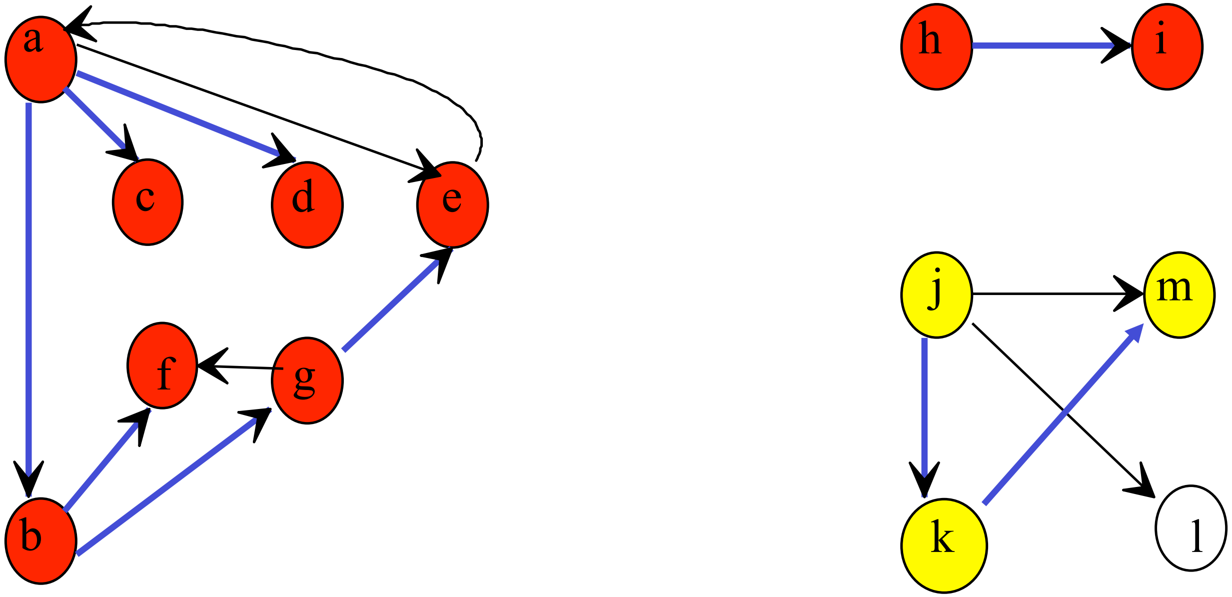


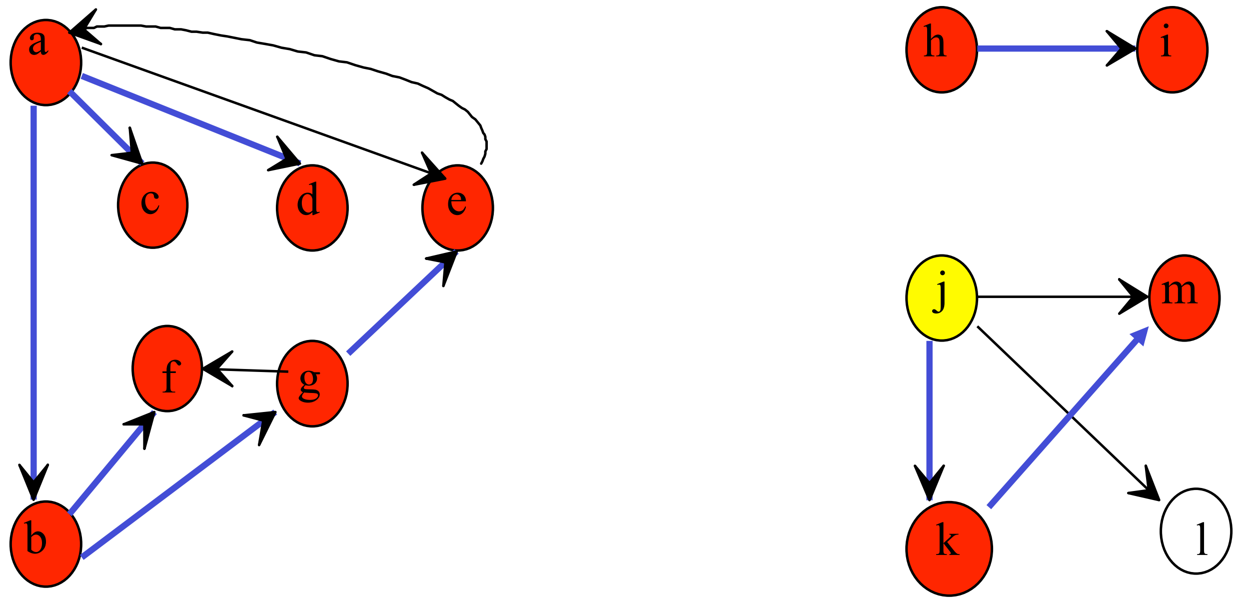
UMLV ©



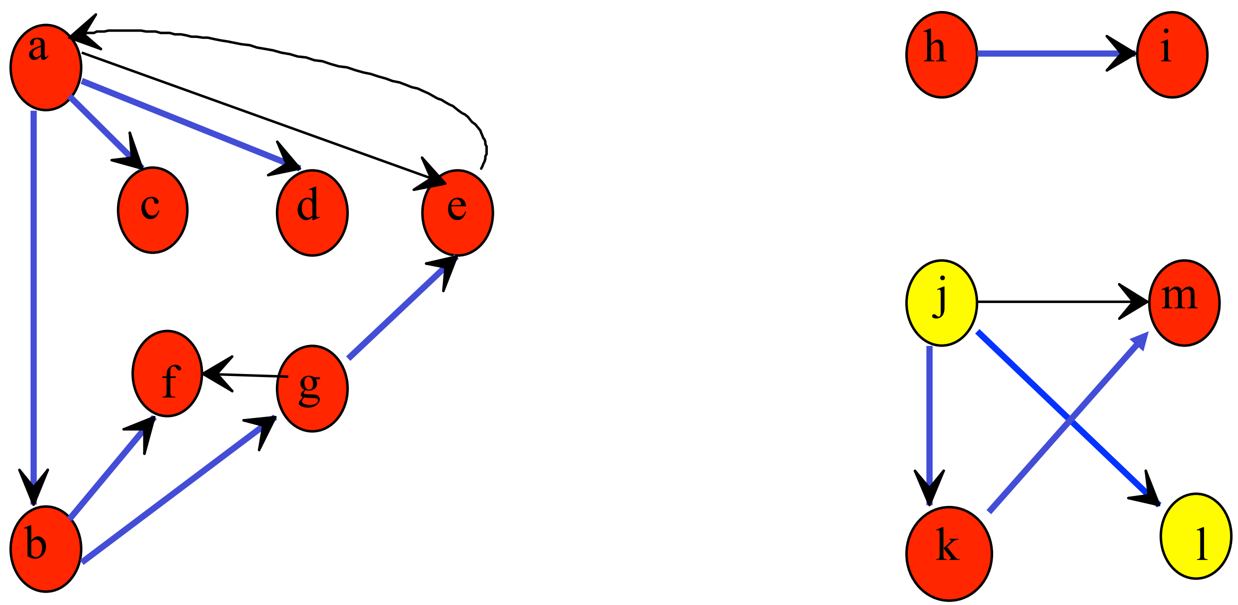

a
b
c
d
e
f
g
h
i
j
k
l
m

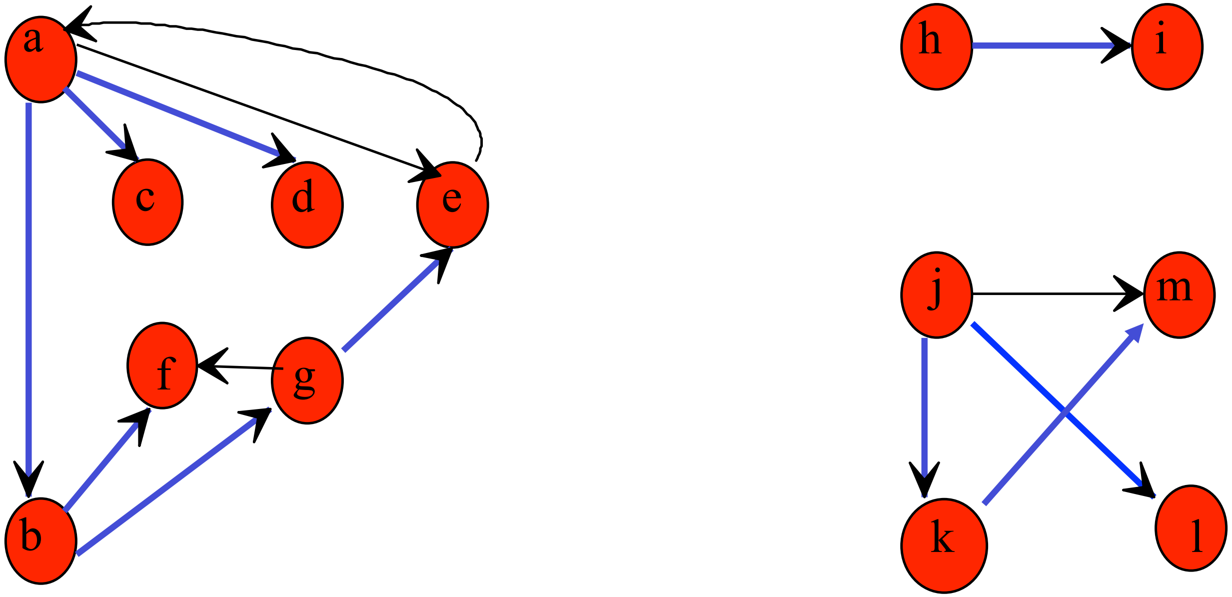
UMLV ©




a
b
c
d
e
f
g
h
i
j
k
l
m

UMLV ©




a
b
c
d
e
f
g
h
i
j
k
l
m

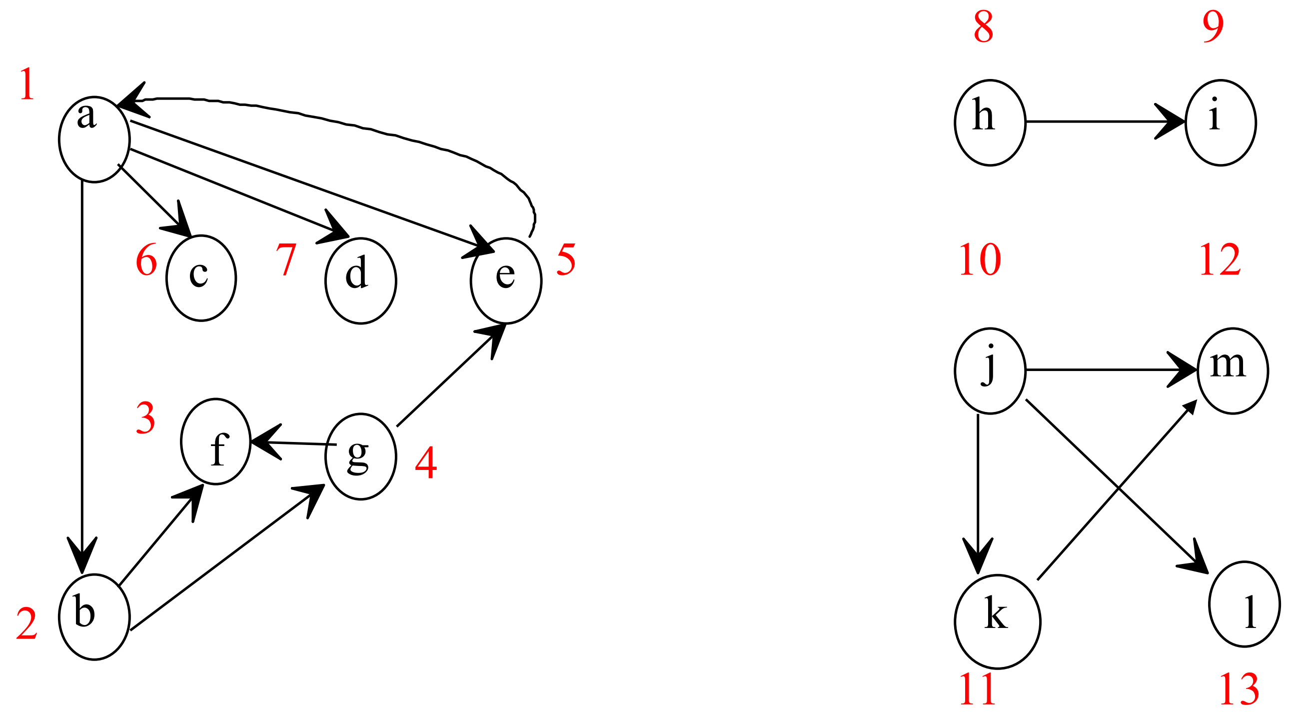
Enumeration

```
function Enumeration (G graph) : array of numbers
        for each node s de G  do
                num [s]  ←  0 ;
        count ← 0 ;
        for each node s de G  do
                if num [s] = 0  then  Number (s) ;
        return (no) ;
end
procedure  Number (s node of G) ;
begin
        count ← count + 1 ; num [s] ← count ;
        for each t successor of  s do
                if num [ t ] = 0  then  Number ( t ) ;
end
```

number of calls of Number = $|S|$

number of « *num* [*t*] = 0 » in Number = $|A|$

time = O ($|S| + |A|$)
on adjacency list


1
a
6
c
7
d
e
5
3
f
g
4
2
b
8
9
h
i
10
12
j
m
k
l
11
13

Depth-first traversal

Node marking

for each node *s* of *G* **do**

visited[*s*] $\longleftarrow$ false ; //*s* is *white*

for each node s of G **do**

if not visited [*s*] **then** DFT(*s*) ;

procedure DFT(*s* node of *G*) ;

begin

opening action on *s* ;

visited[*s*] $\longleftarrow$ true ; //*s* becomes *yellow*

for each *t* successor of *s* **do {**

processing edge (*s*,*t*) ;

if not visited[*t*] **then** DFT(*t*) ;

}

closing action on *s* ; //*s* becomes *red*

end

Time of traversal

T (« for each node ») = $O(|S|)$

Adjacency matrix

T (« **for** each t adjacent to s ») =

T $\left(\begin{array}{l}\text{« } \textbf{for} \text{ each node } t \\ \quad \text{such that } M[s,t] = 1 \text{ »}\end{array}\right)$ $= O(|S|)$

$\Rightarrow$ traversal in time $O((|S|)^2)$

Adjacency list

T (« **for** each t adjacent to s ») = $O(|A(s)|)$

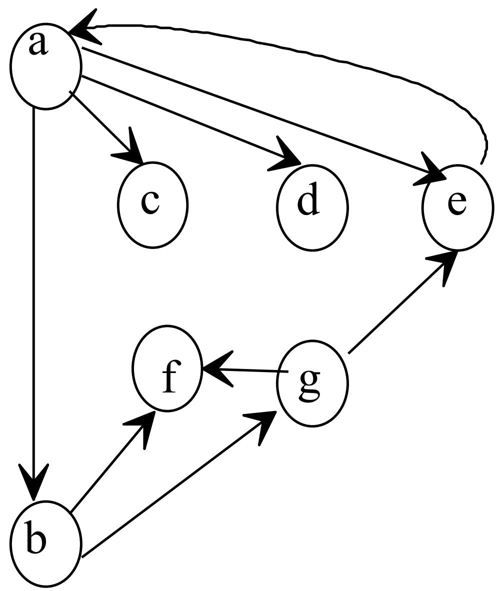
$\Rightarrow$ traversal in $O(|S| + |A|)$

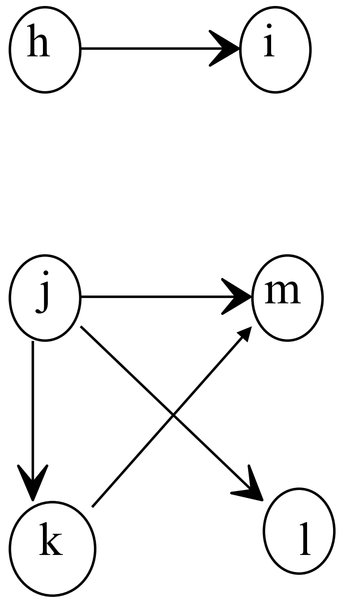
Depth-first traversal: iterative version

```
Procedure DFT-iter (s node of G) ;
begin
        S ← push (empty-stack, s) ;
        visited [s] ← true ;
        while not empty (S) do {
                s′ ← pop(S) ;
                {
                        for  t ← last to first successor of s′  do
                                if not  visited [ t ]  then
                                        {
                                        visited [t] ← true ;
                                        S ← push (S, t) ;
                                }
                }
        }
end
```

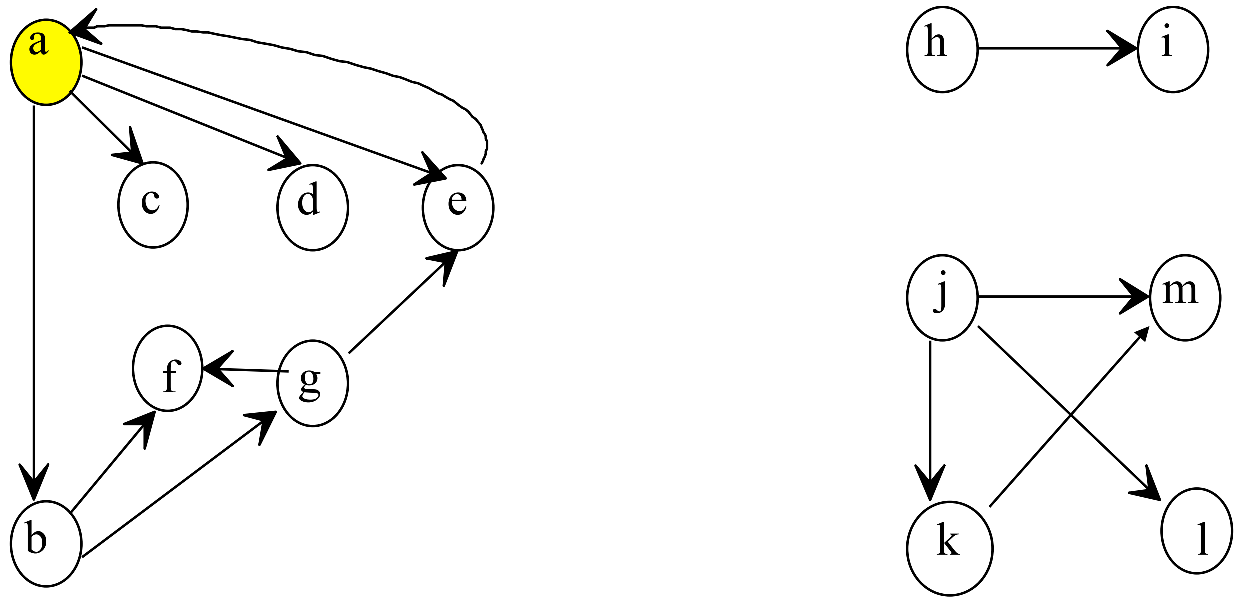
Remarks :

- « pointers » to nodes are stacked

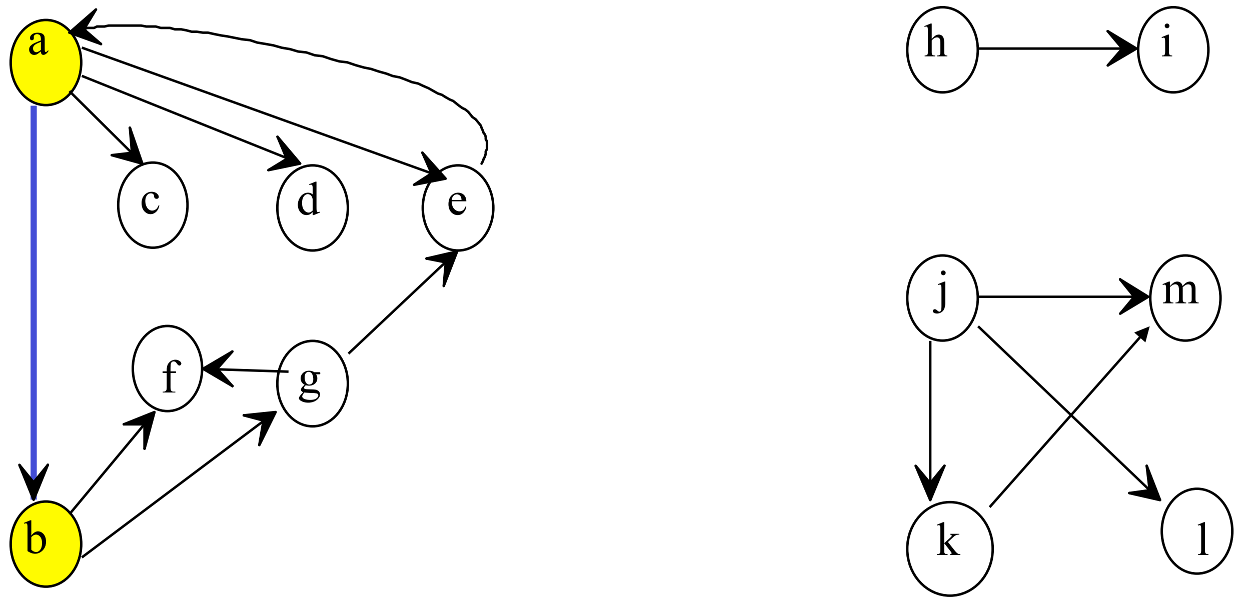




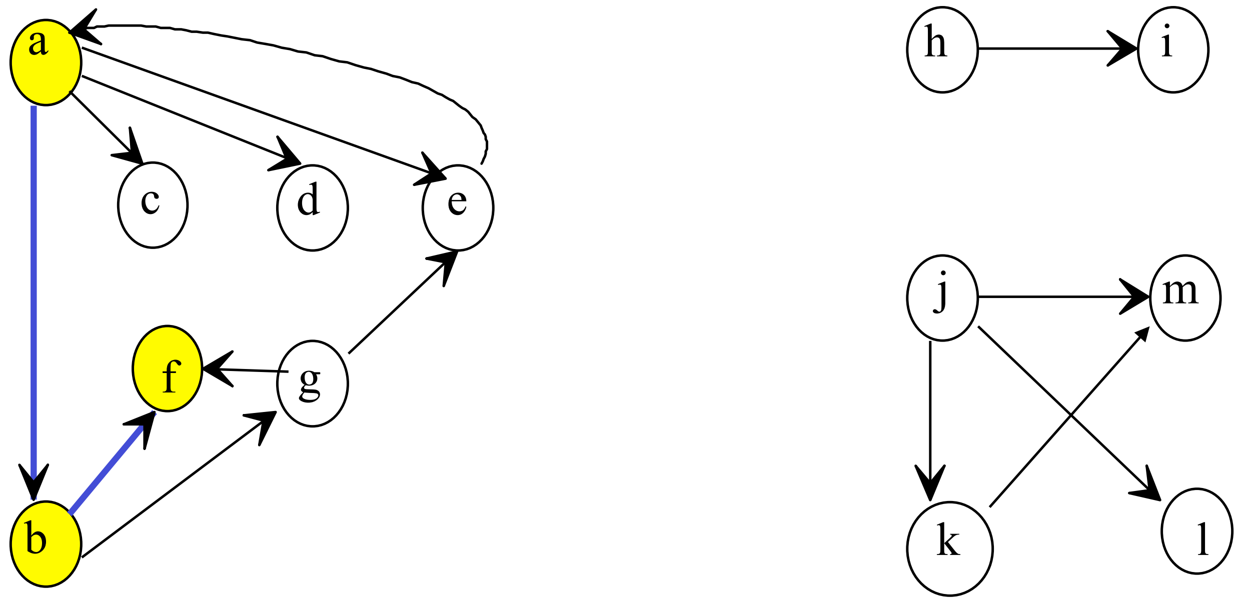
Stack : a



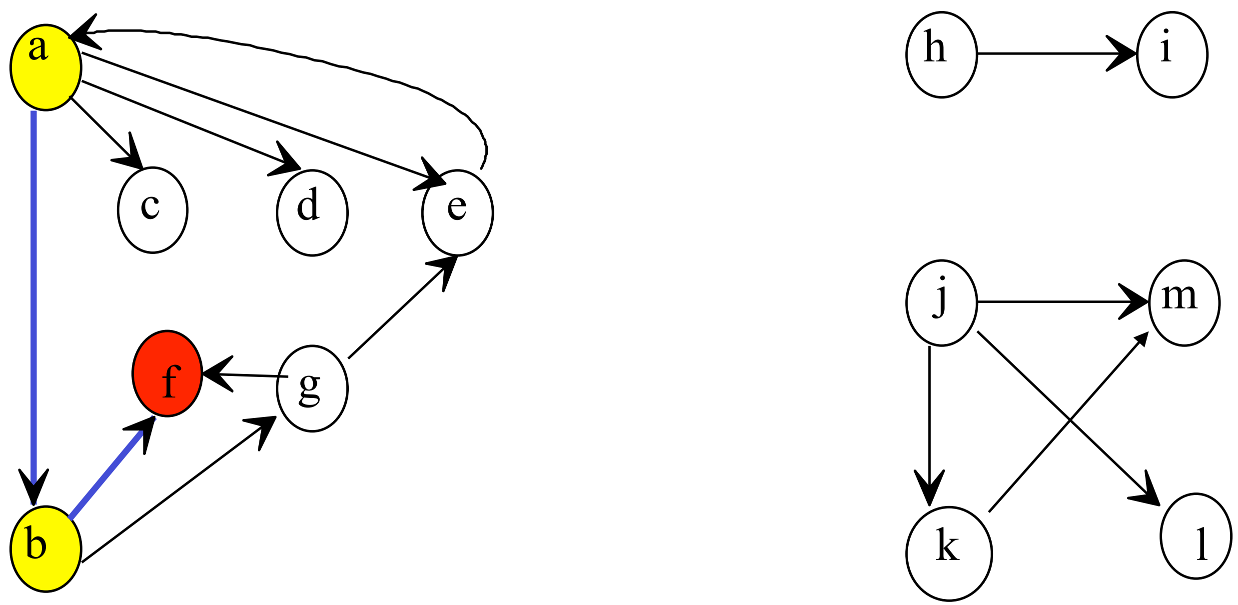
Stack : e d c b



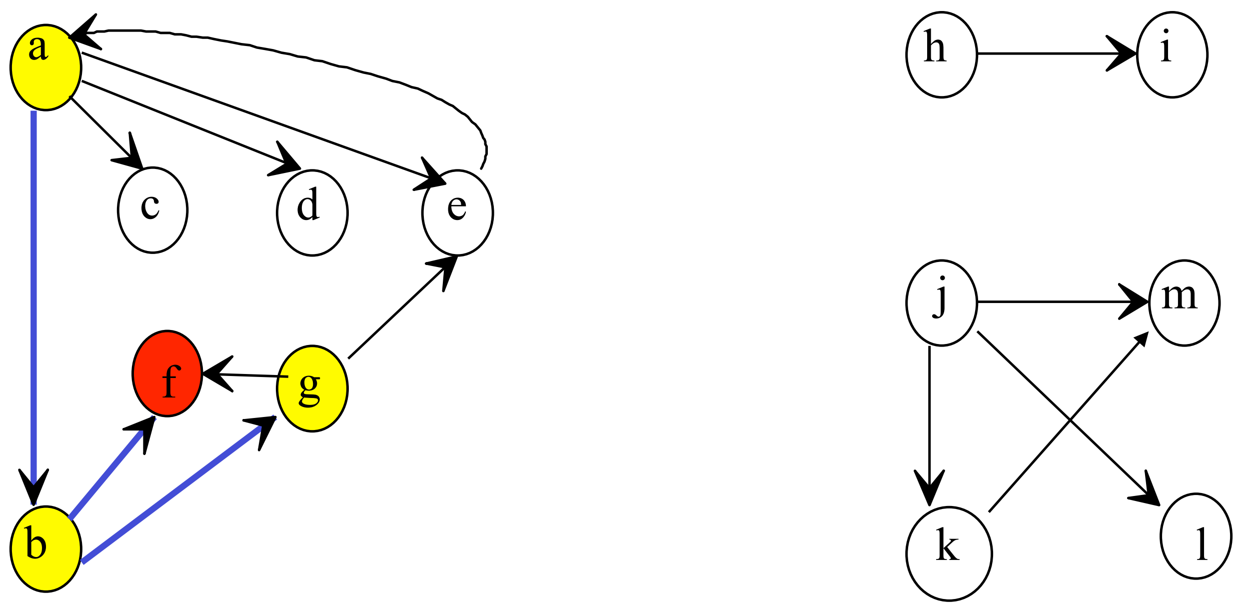
Stack : e d c g f



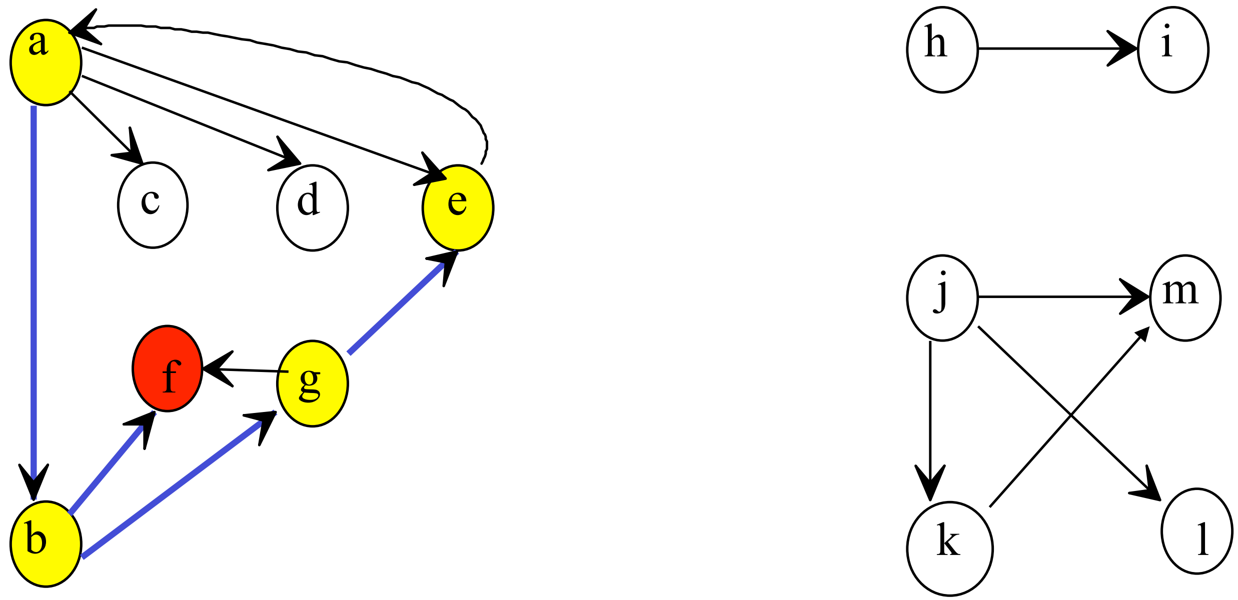
Stack : e d c g



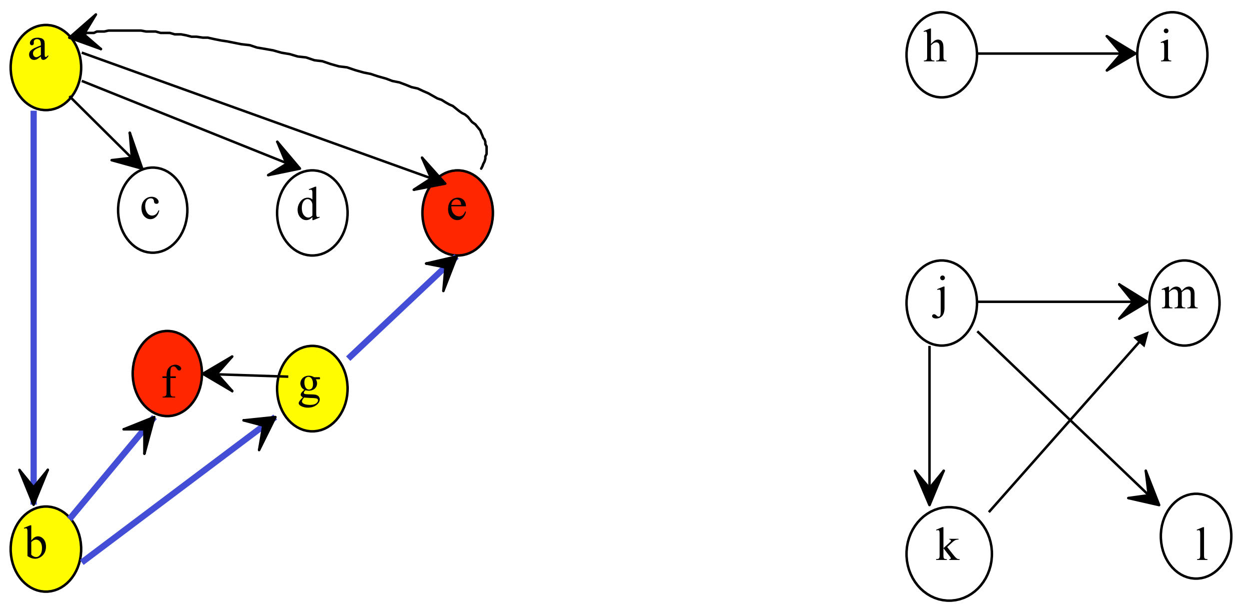
Stack : e d c g



Stack : e d c e



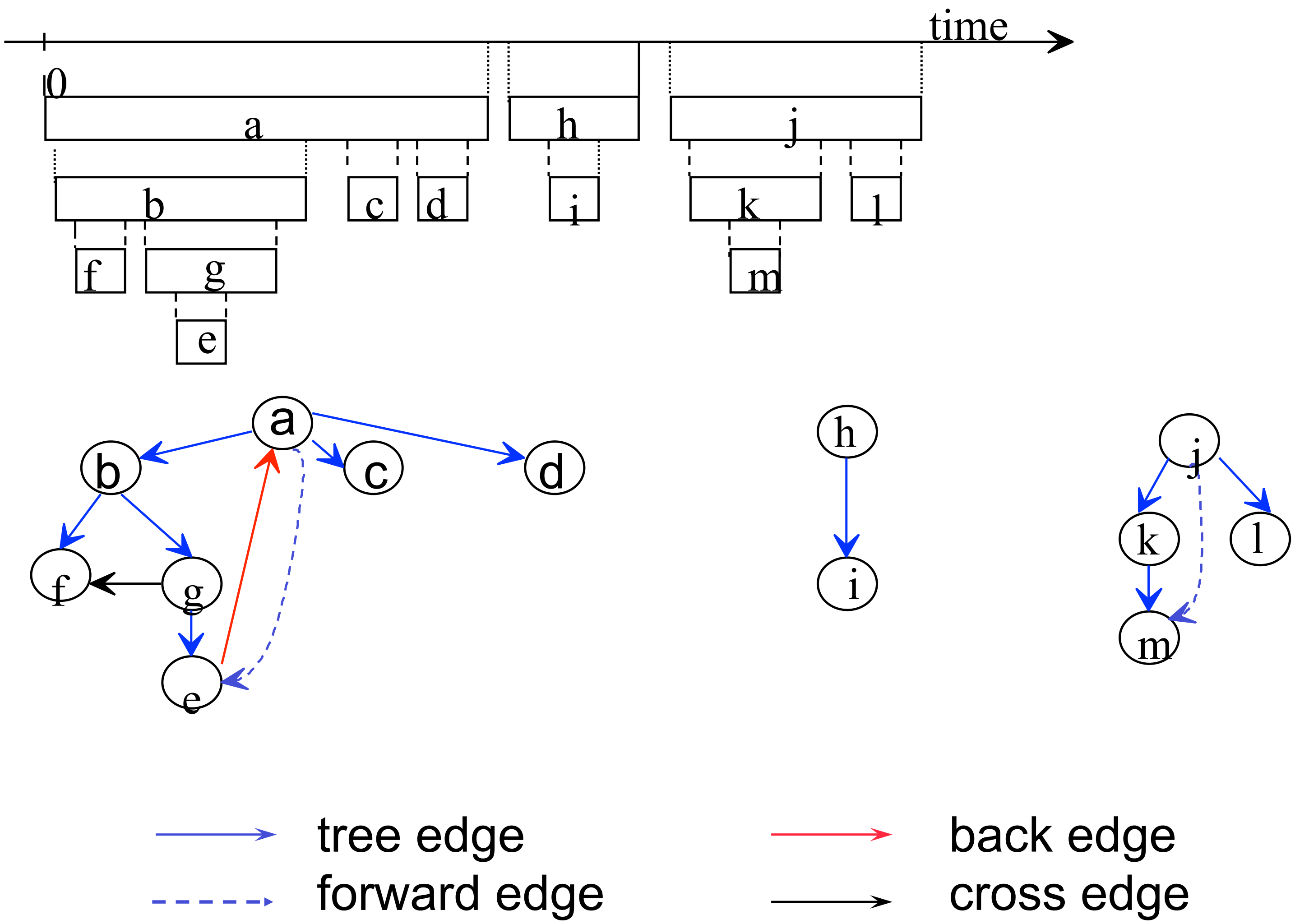
Stack : e d c



Stack : e d c

And so on …

Depth-first forest



Cycle detection

Proposition

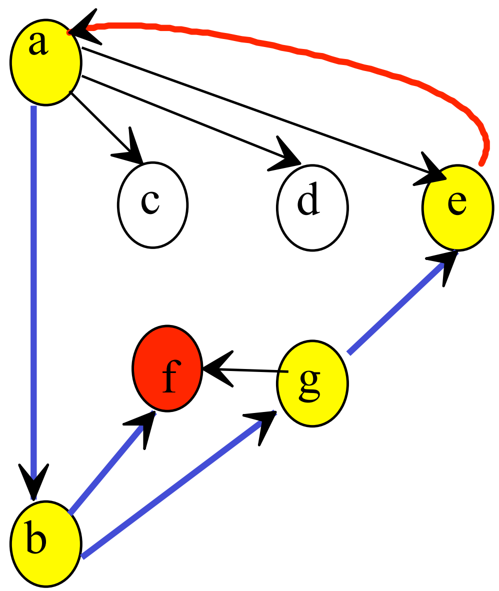
A directed graph G has a cycle iff there exists a back edge in the depth-first forest of G

$d(s)$: *discovery time* (turning yellow in DFT)
$f(s)$: *finishing time* (turning red in DFT)

(s,t) edge of G is

- tree edge or forward edge iff $d(s) < d(t) < f(t) < f(s)$
- back edge iff $d(t) < d(s) < f(s) < f(t)$
- cross edge iff $f(t) < d(s)$

Example

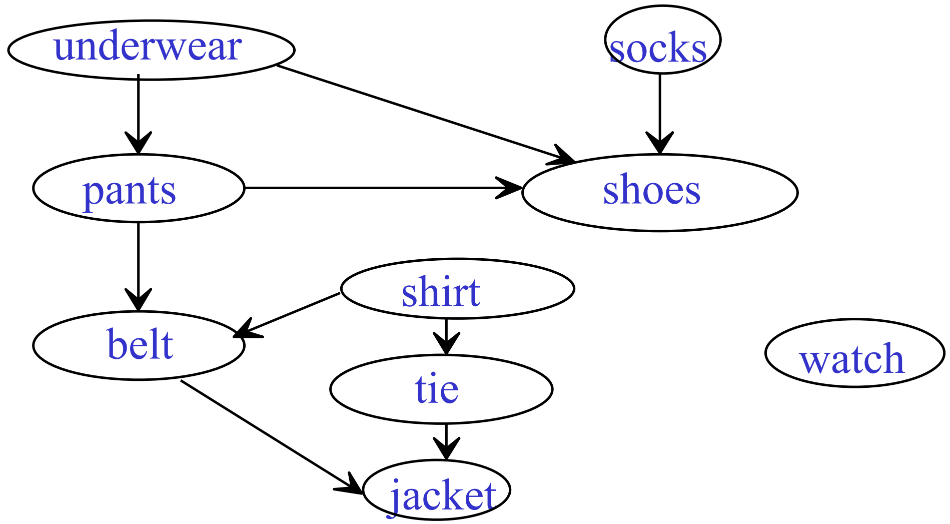


When visiting node e, we detect a cycle going through edge (e, a) as a is being processing as well

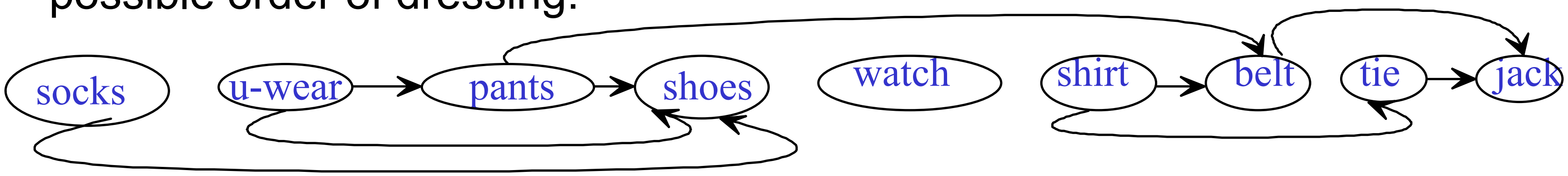
Topological sort

Embedding of a partial order into a linear order

DAG=Directed Acyclic Graph



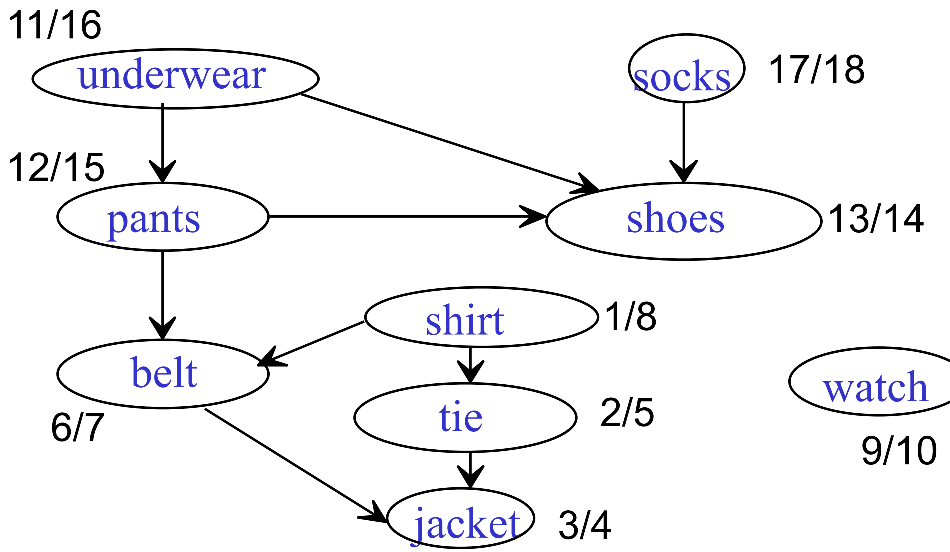
possible order of dressing:



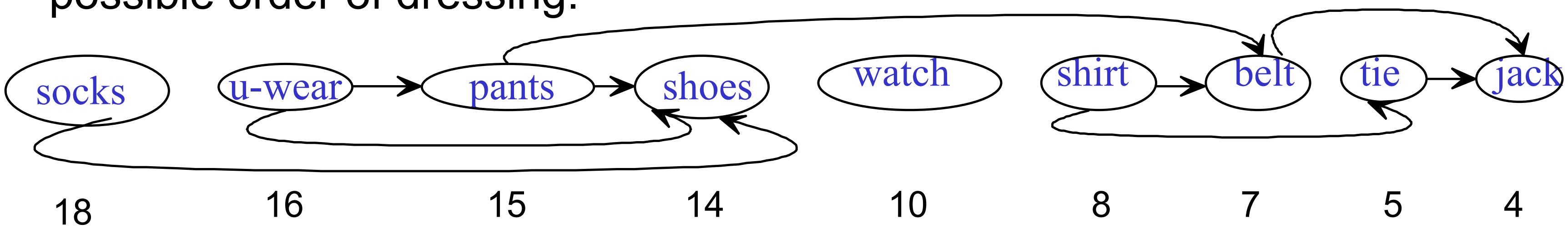
Topological sort

Embedding of a partial order into a linear order

DAG=Directed Acyclic Graph



possible order of dressing:



Resulting order: decreasing order of *f* (*s*) in depth-first traversal

Topological sort by depth-first traversal

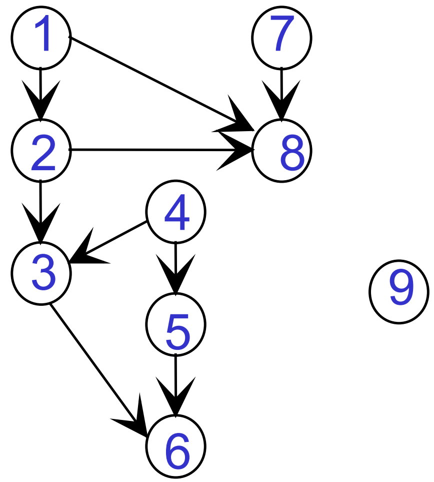
function Topological-sort (*G* acyclic graph) : list ;
begin
        for each node *s* of *G* **do**
                visited [*s*] ← false ;
        L ← empty-list ;
        for each node *s* of *G* **do**
                if not visited [*s*] **then** Topo (*s*) ;
        return (*L*) ;
end

procedure Topo (*s* node of *G*) ;
begin
        visited [*s*] ← true ;
        for each *t* successor of *s* **do**
                if not visited [*t*] **then** Topo (*t*) ;
        add s to head of *L* ;
end

Iterative method

	1	2	3	4	5	6	7	8	9
Nb-pred	0	1	2	0	1	2	0	3	0

Nodes to process : 1 4 7 9
(without predecessor)



After processing 1 :

	1	2	3	4	5	6	7	8	9
Nb-Pred	-	0	2	0	1	2	0	2	0

Nodes to process : 4 7 9 2

Iterative topological sort

function Topological-sort (*G* acyclic graph) : list ;
begin
    F ← empty-queue;
    while *G* not empty **do**
        if each node has a predecessor **then**
            « *G* contains a cycle » ;
        else {
            s ← a node without predecessor;
            G ← *G* without *s* and all edges outgoing from *s* ;
            F ← enqueue (*F*, s) ;
        }
    return (*F*) ;
end

Running time: $O(|S| + |A|)$
    using adjacency lists

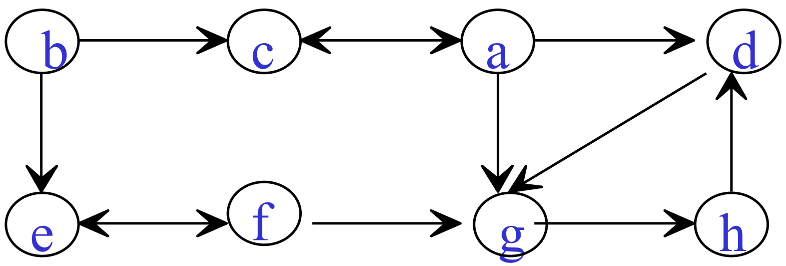
Strongly connected components

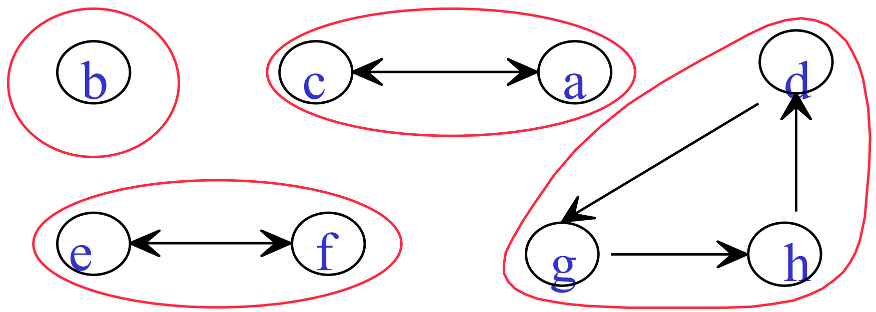
$G = (S, A)$ graph

$G' = (S', A')$ subgraph of G iff $S' \subseteq S$ et $A' \subseteq A \cap S' \times S'$

F **strongly connected component** of G :

F maximal subgraph of G such that
any two nodes of F are connected by a path





Algorithm

G^T = transpose of G

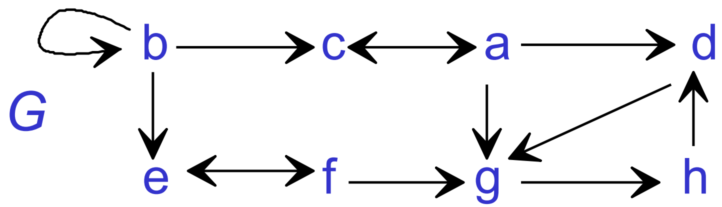
Algorithm [Kosaraju 78] [Sharir 81]

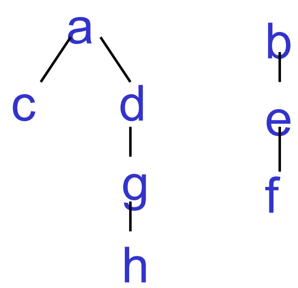
$L \leftarrow$ list of nodes of G obtained by

depth-first traversal *and ordered by increasing* f(s);

from L^R, apply depth-first traversal to G^T ;

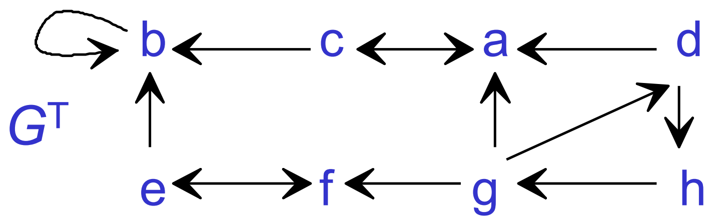
The trees of the depth-first forest of this traversal are strongly connected components of G

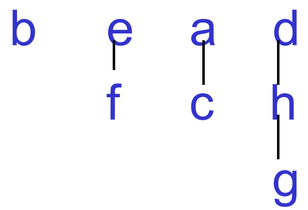




L = c h g d a f e b

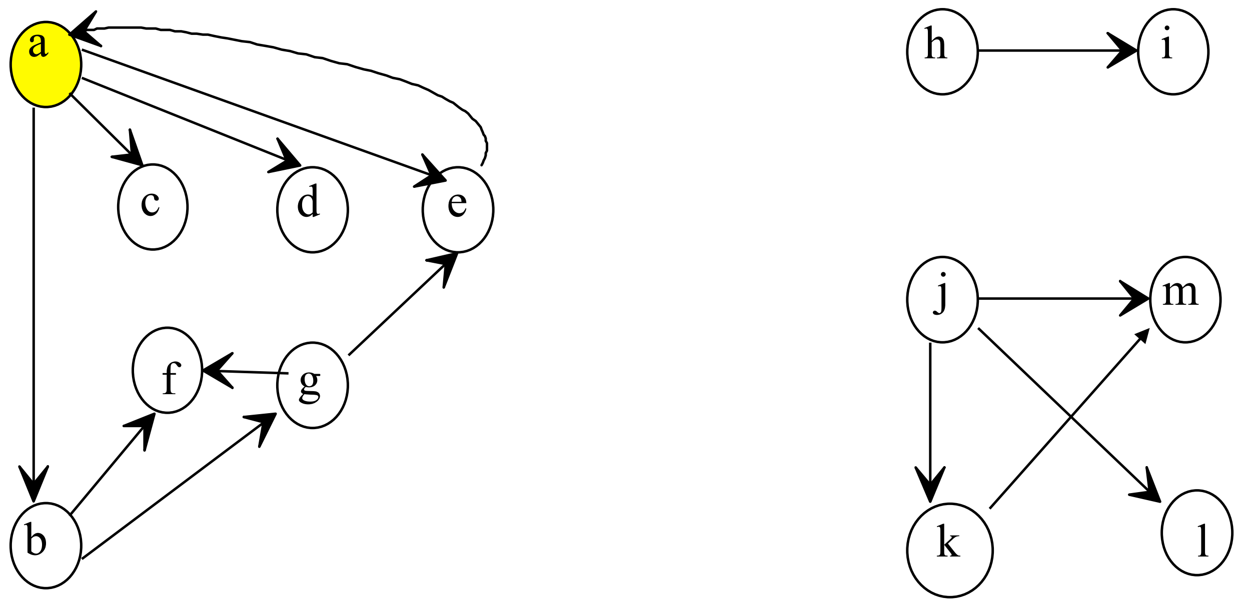
L^R = b e f a d g h c





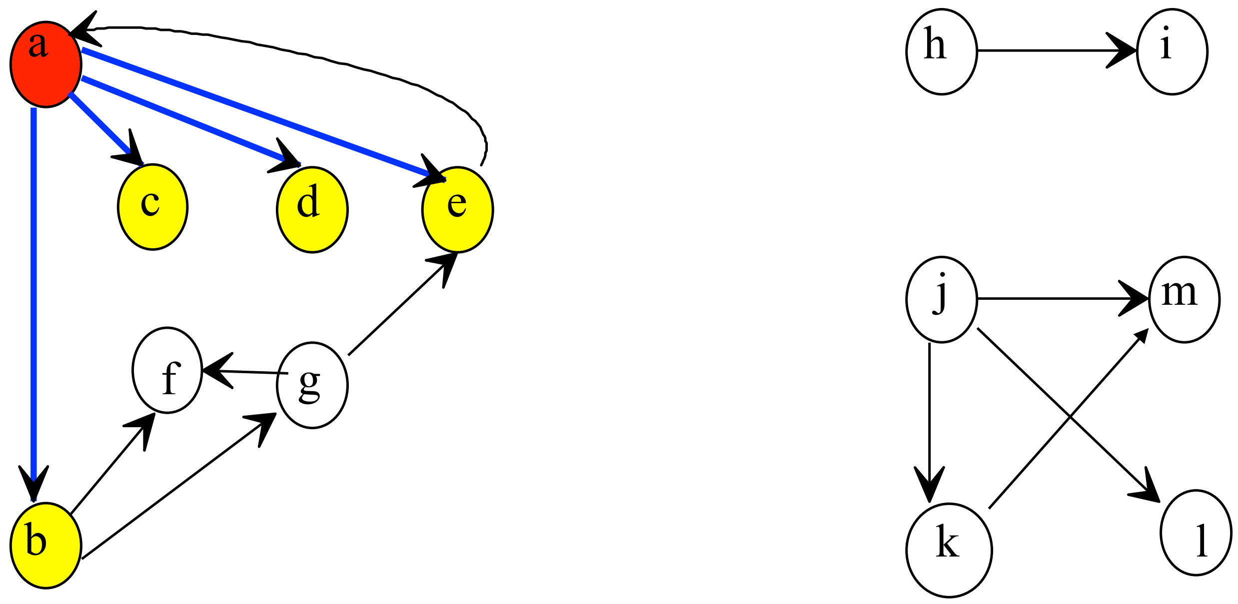
Breadth-first traversal

```
procedure BFT (s node of G) ;
begin
for each node v of G  do {
          visited[v] ← false ; //s is white
          d[v]=∞ ;
          }
Queue ← enqueue (empty-queue, s) ;
visited[s]=true ; //s becomes yellow
d[s]=0 ;
while not empty (Queue)  do {
          s′ ← dequeue (Queue) ;
          for  t ← first to last successor of s′ do
                    if not visited [ t ] then
                              d[ t ] ← d[ s’ ]+1 ;
                              Queue ← enqueue (Queue, t) ;
          //s’ becomes red
          }
end
```



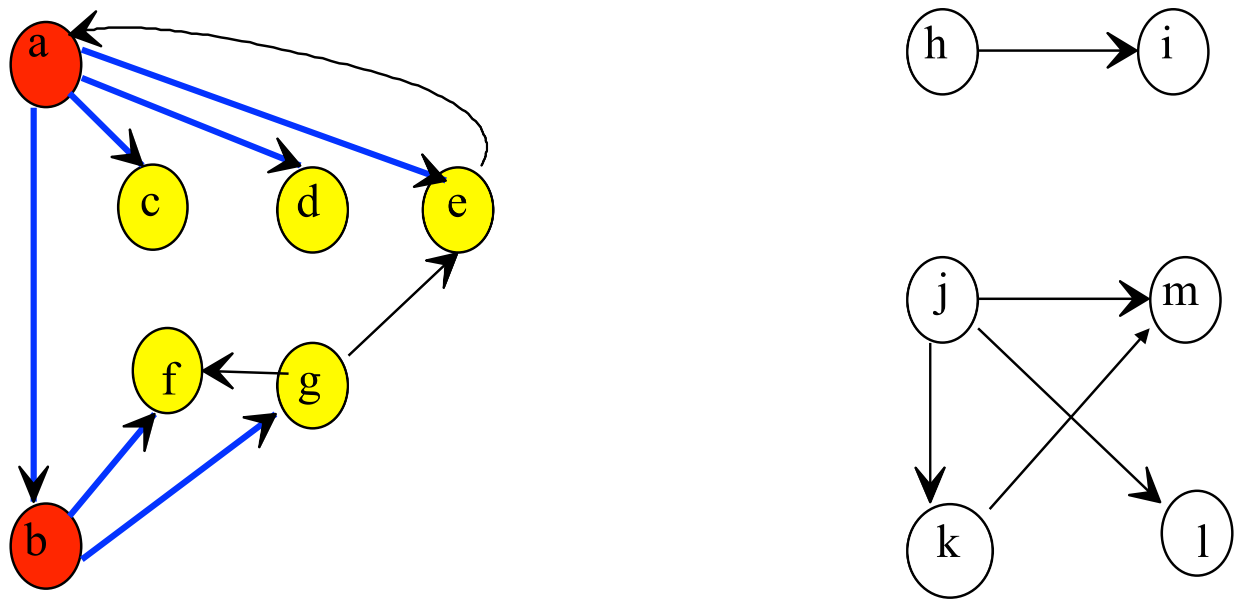
Queue : a

Order of traversal:



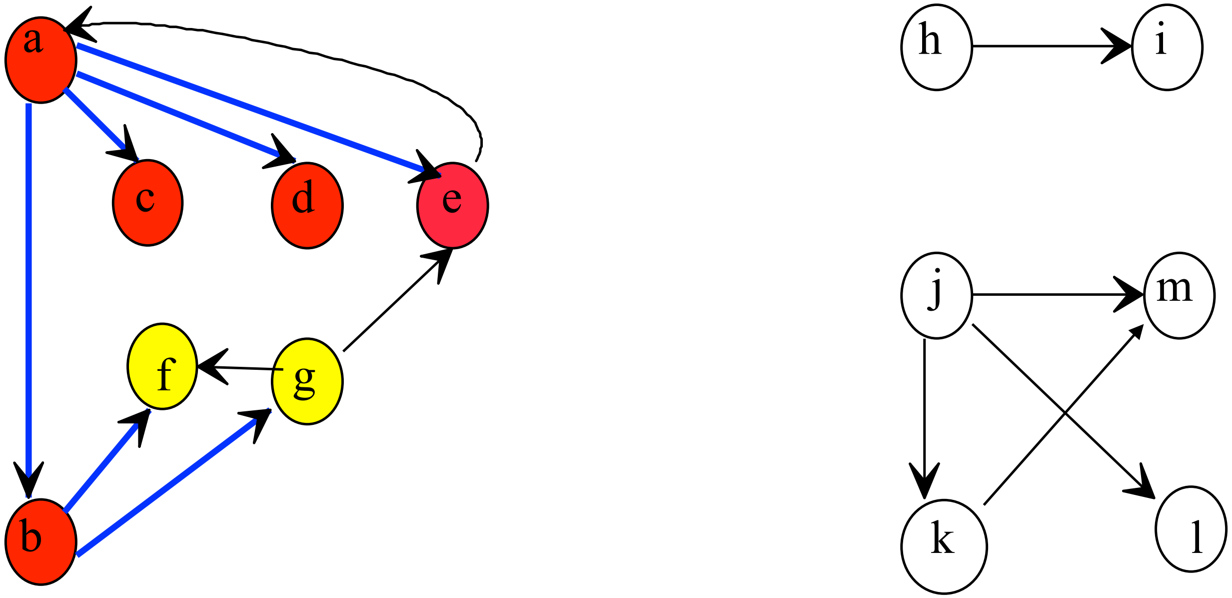
Queue : a b c d e

Order of traversal: a



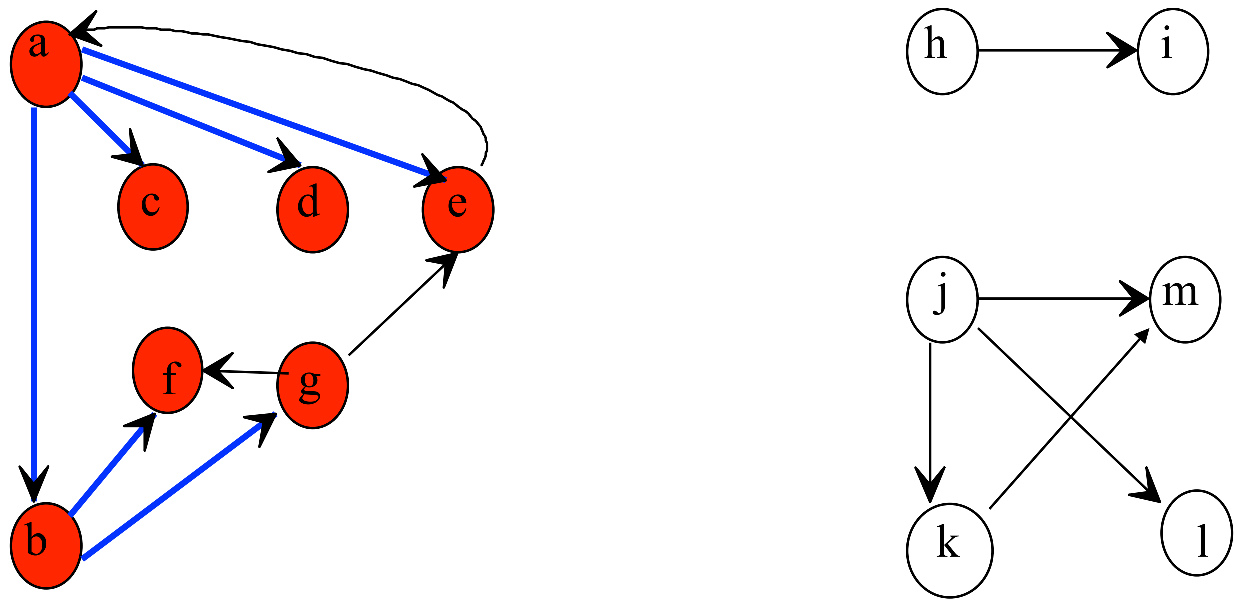
Queue : a b c d e f g

Order of traversal: a b



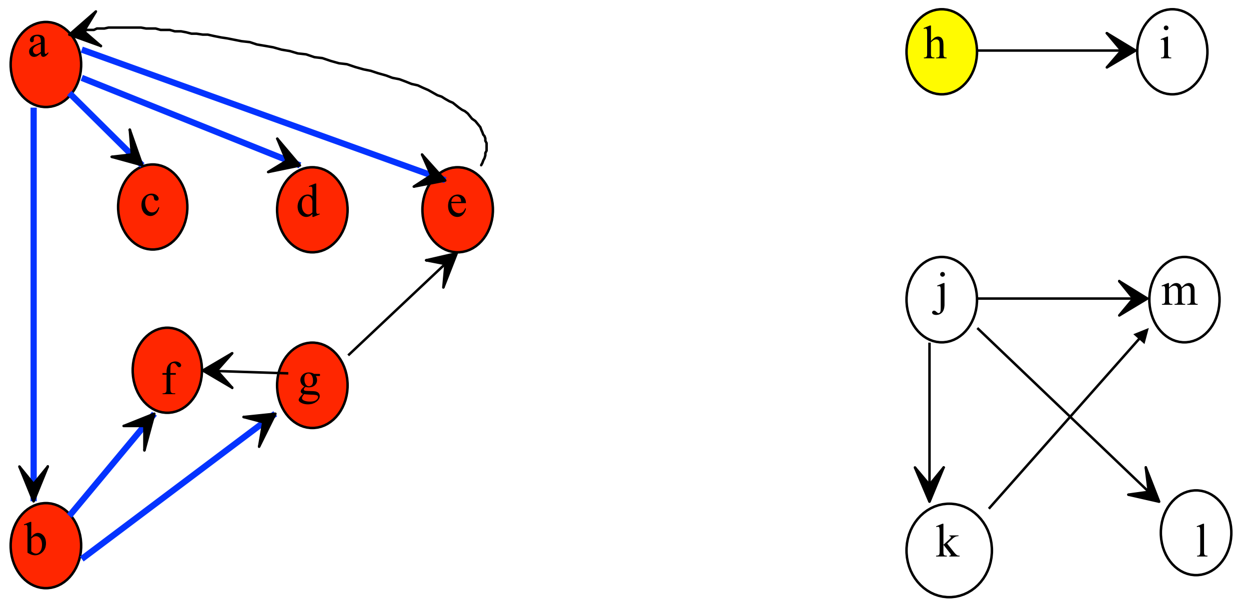
Queue : a b c d e f g

Order of traversal: a b c d e



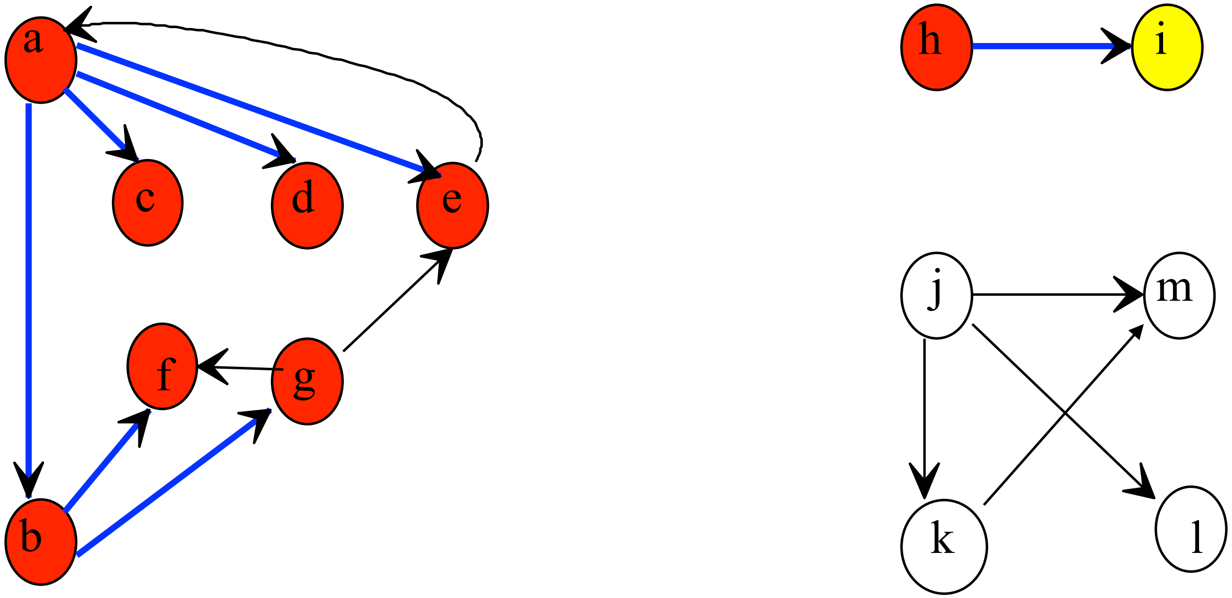
Queue : a b c d e f g

Order of traversal: a b c d f g



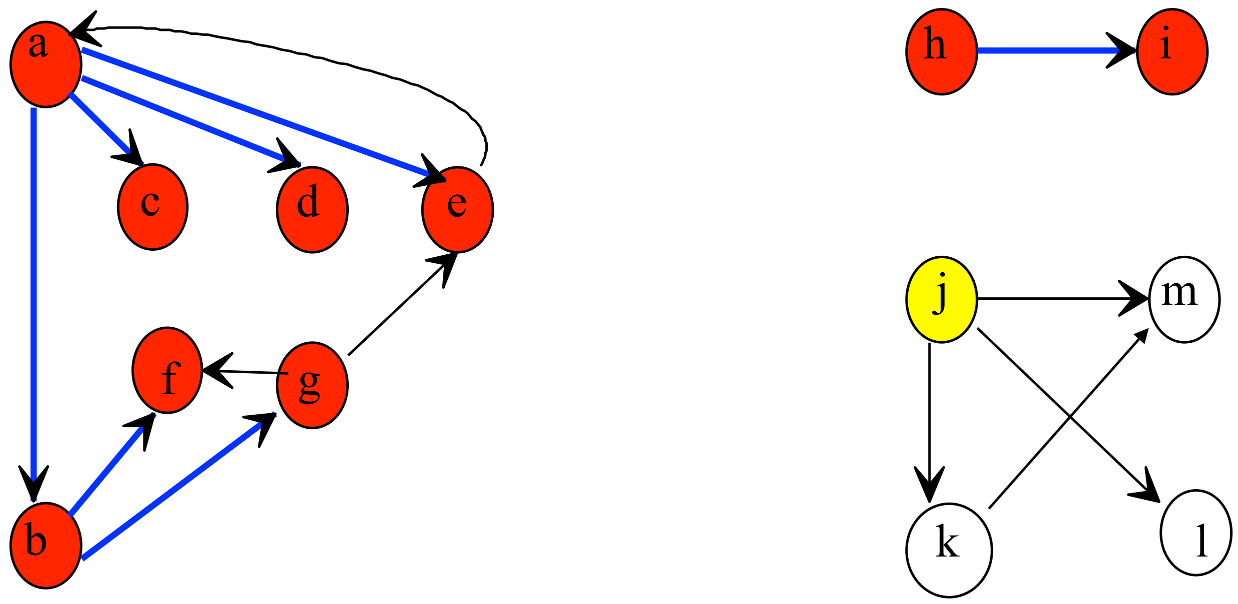
Queue : a b c d e f g h

Order of traversal: a b c d f g h



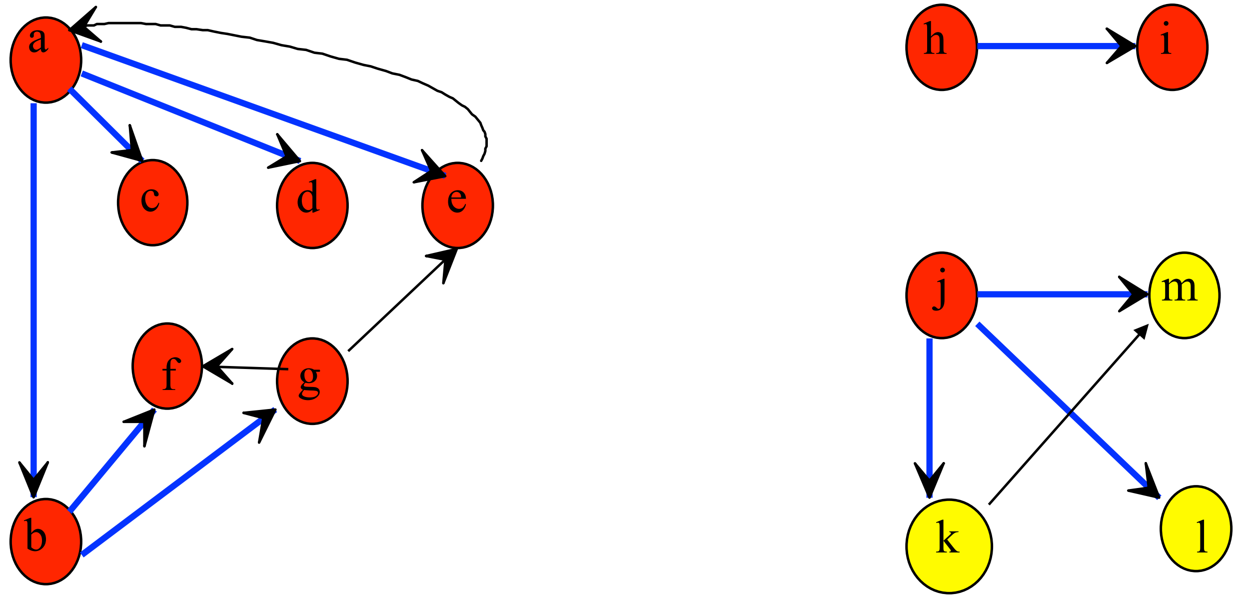
Queue : a b c d e f g h i

Order of traversal: a b c d f g h i



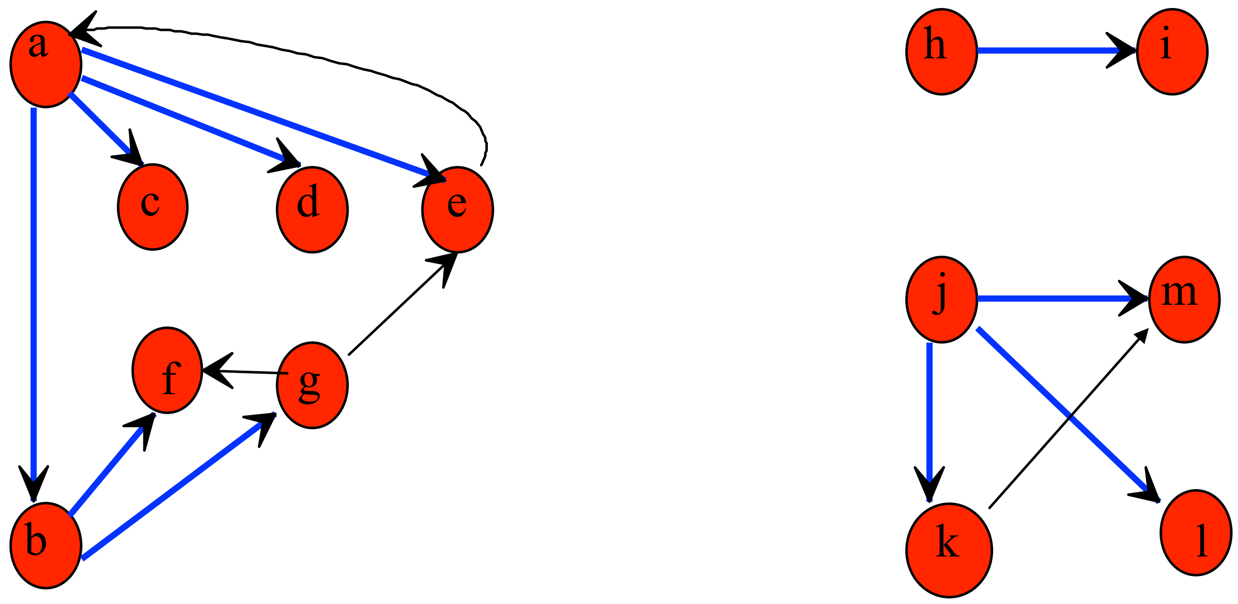
Queue : a b c d e f g h i j

Order of traversal: a b c d f g h i j



Queue : a b c d e f g h i j k l m

Order of traversal: a b c d f g h i j k



Queue : a b c d e f g h i j k l m

Order of traversal: a b c d f g h i j k l m

Shortest path

Assume there is a path from *s* to *v*

d(v) : the (rank of the) iteration at which *v* is first visited (becomes yellow)

Then *d(v)* is the length of the shortest path from *s* to *v*