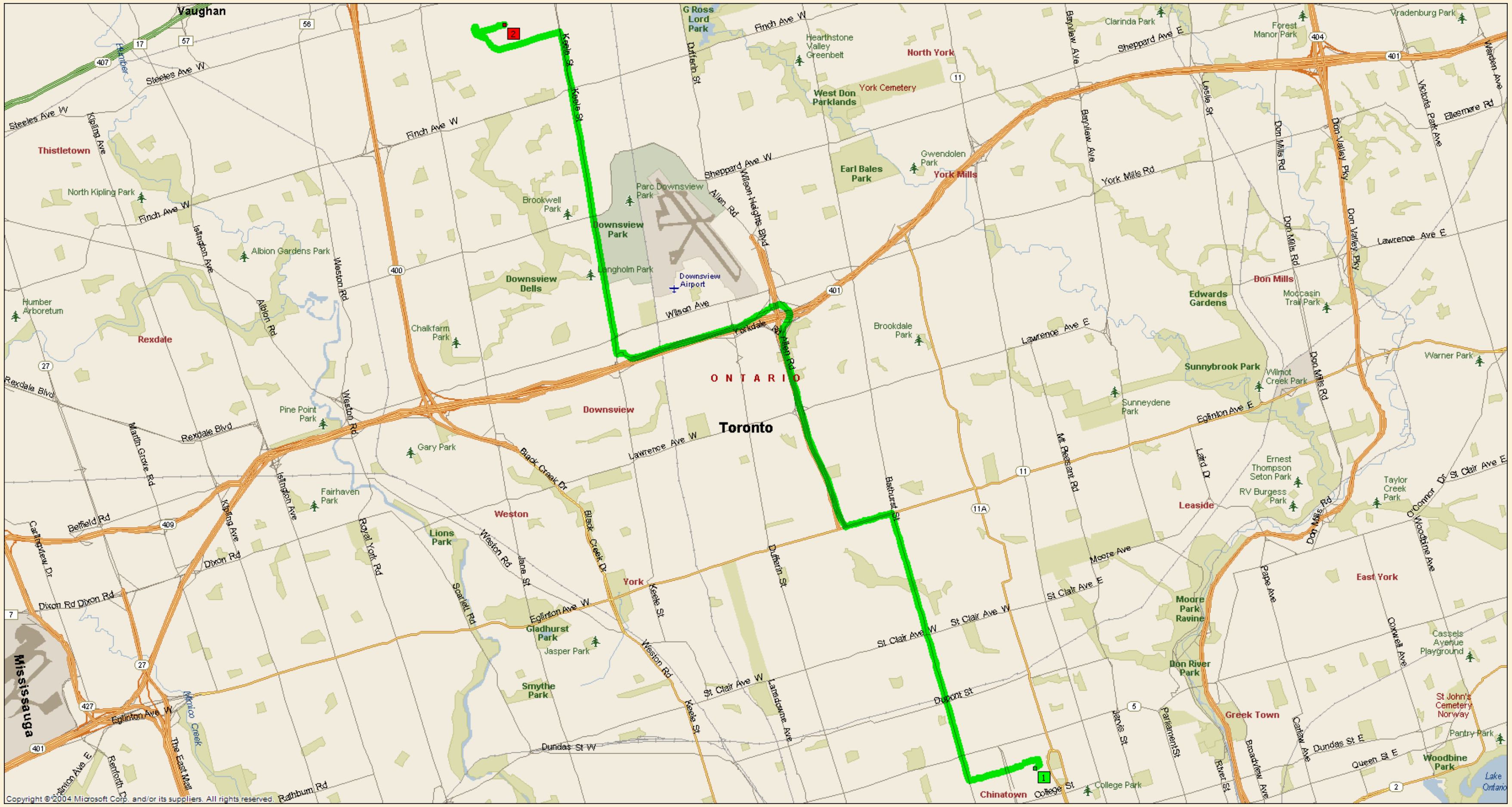
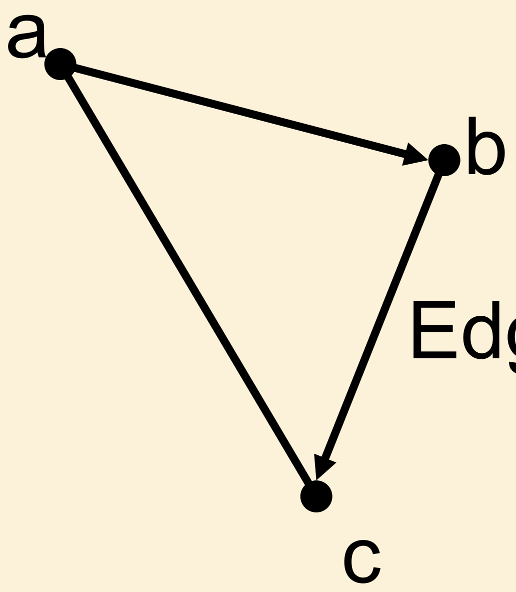
Graph Search Algorithms


Toronto
ONTARIO
Keele St
Allen Rd
Bathurst St
Copyright © 2004 Microsoft Corp. and/or its suppliers. All rights reserved.

Graph



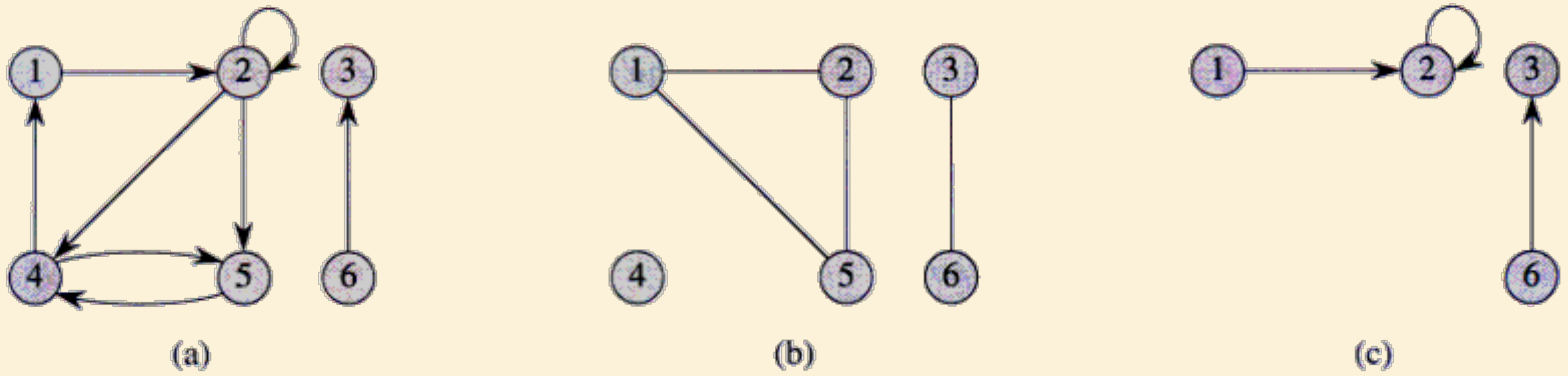
Node ~ city or computer

Edge ~ road or data cable

Undirected or Directed

A surprisingly large number of computational problems can be expressed as graph problems.

Directed and Undirected Graphs

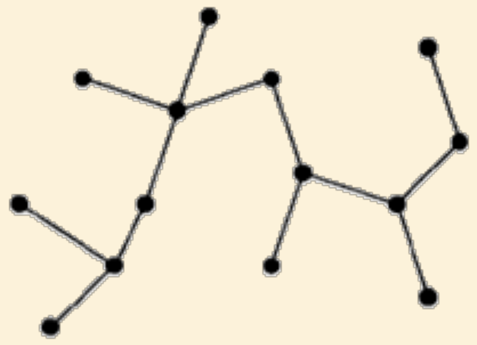


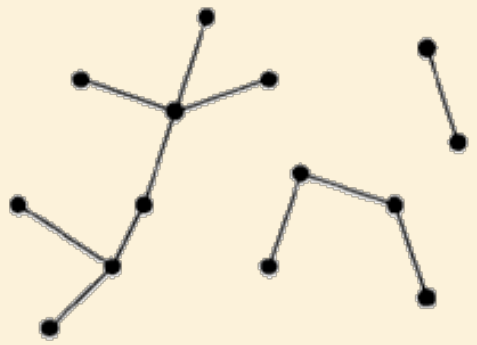
(a) A directed graph $G = (V, E)$, where $V = \{1,2,3,4,5,6\}$ and $E = \{(1,2), (2,2), (2,4), (2,5), (4,1), (4,5), (5,4), (6,3)\}$. The edge (2,2) is a self-loop.

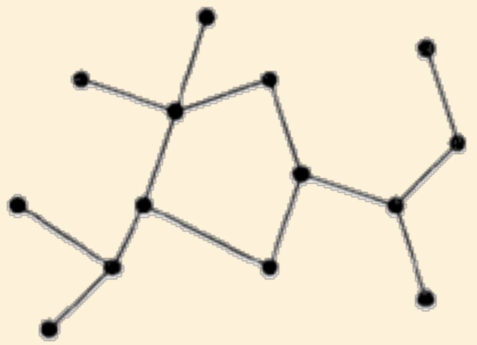
(b) An undirected graph $G = (V,E)$, where $V = \{1,2,3,4,5,6\}$ and $E = \{(1,2), (1,5), (2,5), (3,6)\}$. The vertex 4 is isolated.

(c) The subgraph of the graph in part (a) induced by the vertex set {1,2,3,6}.

Trees


Tree


Forest


Graph with Cycle

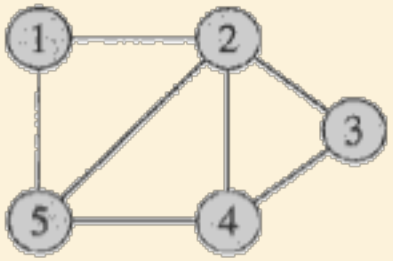
A tree is a connected, acyclic, undirected graph.

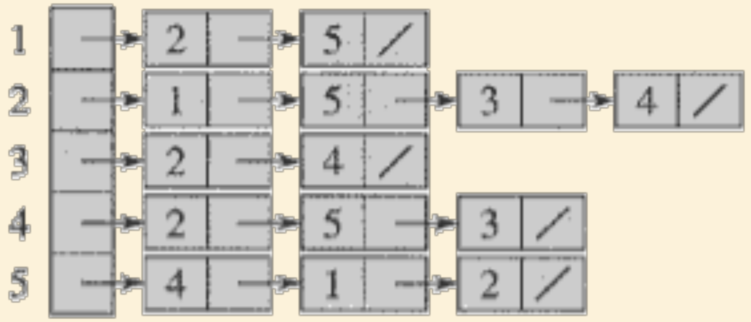
A forest is a set of trees (not necessarily connected)

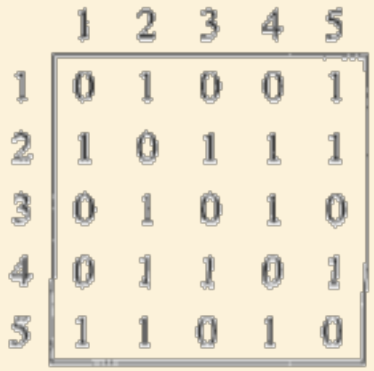
Running Time of Graph Algorithms

- Running time often a function of both |V| and |E|.
- For convenience, drop the | . | in asymptotic notation, e.g. $O(V+E)$.

Representations: Undirected Graphs



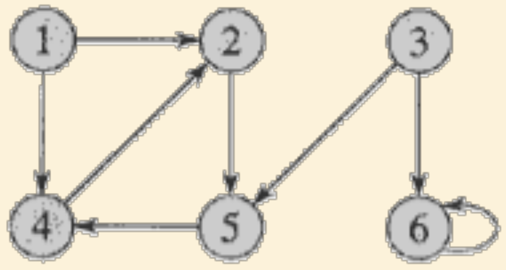


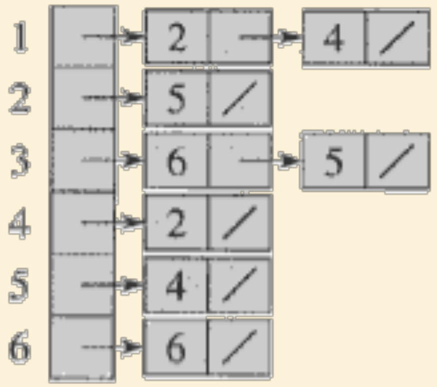


$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 1 & 1 & 0 & 1 & 0 \end{array}$$

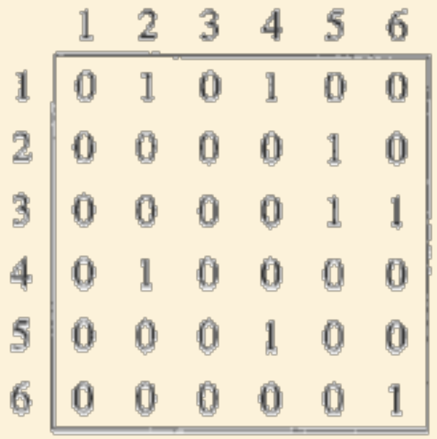
	Adjacency List	Adjacency Matrix
Space complexity:	$\theta(V+E)$	$\theta(V^2)$
Time to find all neighbours of vertex u:	$\theta(\text{degree}(u))$	$\theta(V)$
Time to determine if $(u,v)\in E$:	$\theta(\text{degree}(u))$	$\theta(1)$

Representations: Directed Graphs





Adjacency List



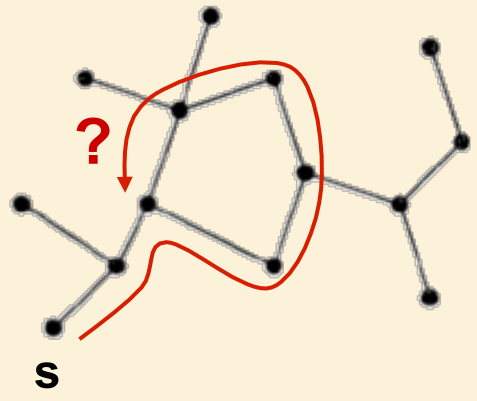
	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Adjacency Matrix

Space complexity:	$\theta(V+E)$	$\theta(V^2)$
Time to find all neighbours of vertex u:	$\theta(\text{degree}(u))$	$\theta(V)$
Time to determine if $(u,v)\in E$:	$\theta(\text{degree}(u))$	$\theta(1)$

Breadth-First Search

- Goal: To recover the shortest paths from a source node *s* to all other reachable nodes *v* in a graph.
  - The length of each path and the paths themselves are returned.
- Notes:
  - There are an exponential number of possible paths
  - This problem is harder for general graphs than trees because of cycles!



Breadth-First Search

Input: Graph $G = (V, E)$ (directed or undirected) and source vertex $s \in V$.

Output:

$d[v] =$ shortest path distance $\delta(s,v)$ from s to v, $\forall v \in V$.

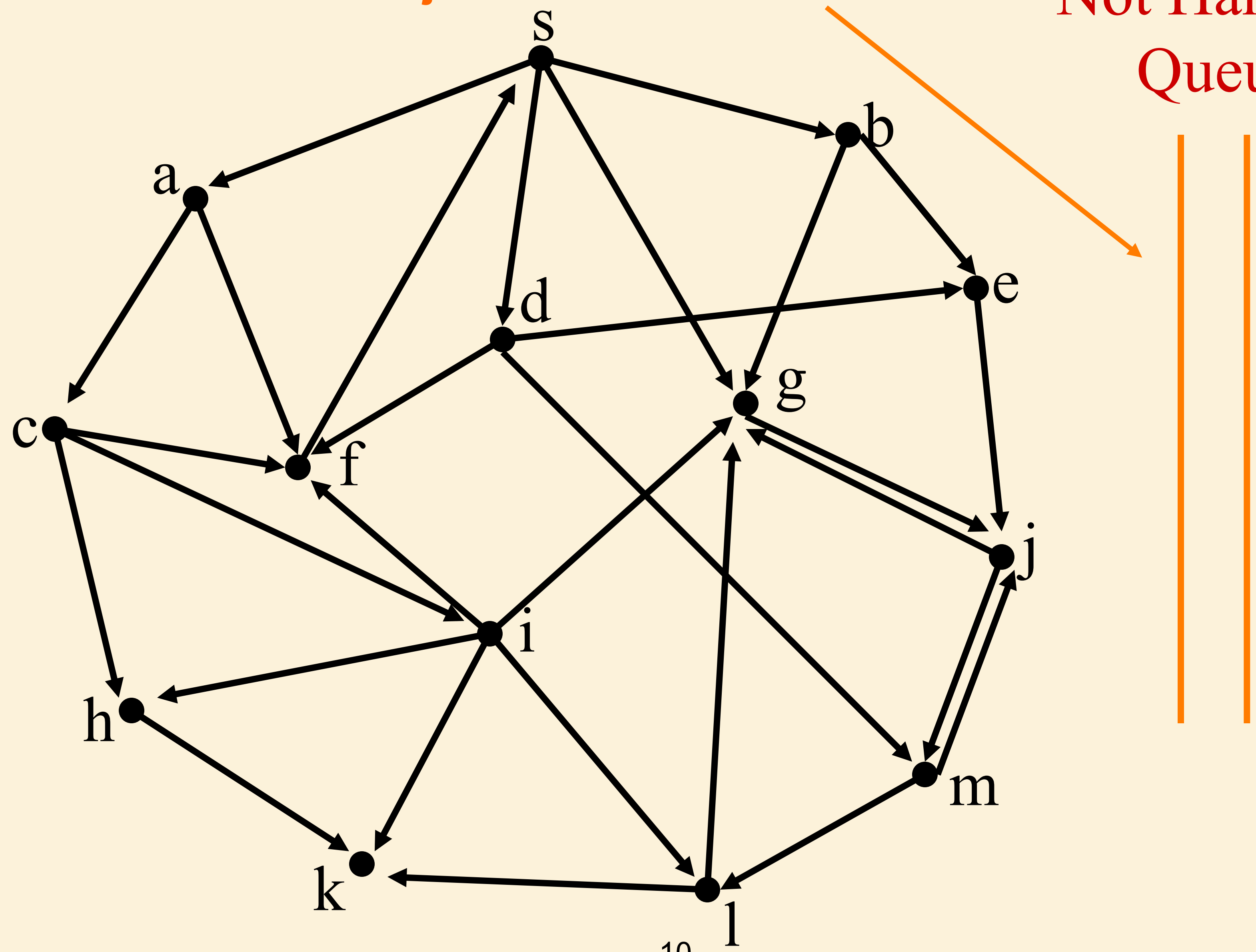
$\pi[v] = u$ such that (u,v) is last edge on a shortest path from s to v.

- Idea: send out search 'wave' from s.
- Keep track of progress by colouring vertices:
  - **Undiscovered** vertices are coloured **black**
  - **Just discovered** vertices (on the wavefront) are coloured **red.**
  - **Previously discovered** vertices (behind wavefront) are coloured **grey.**

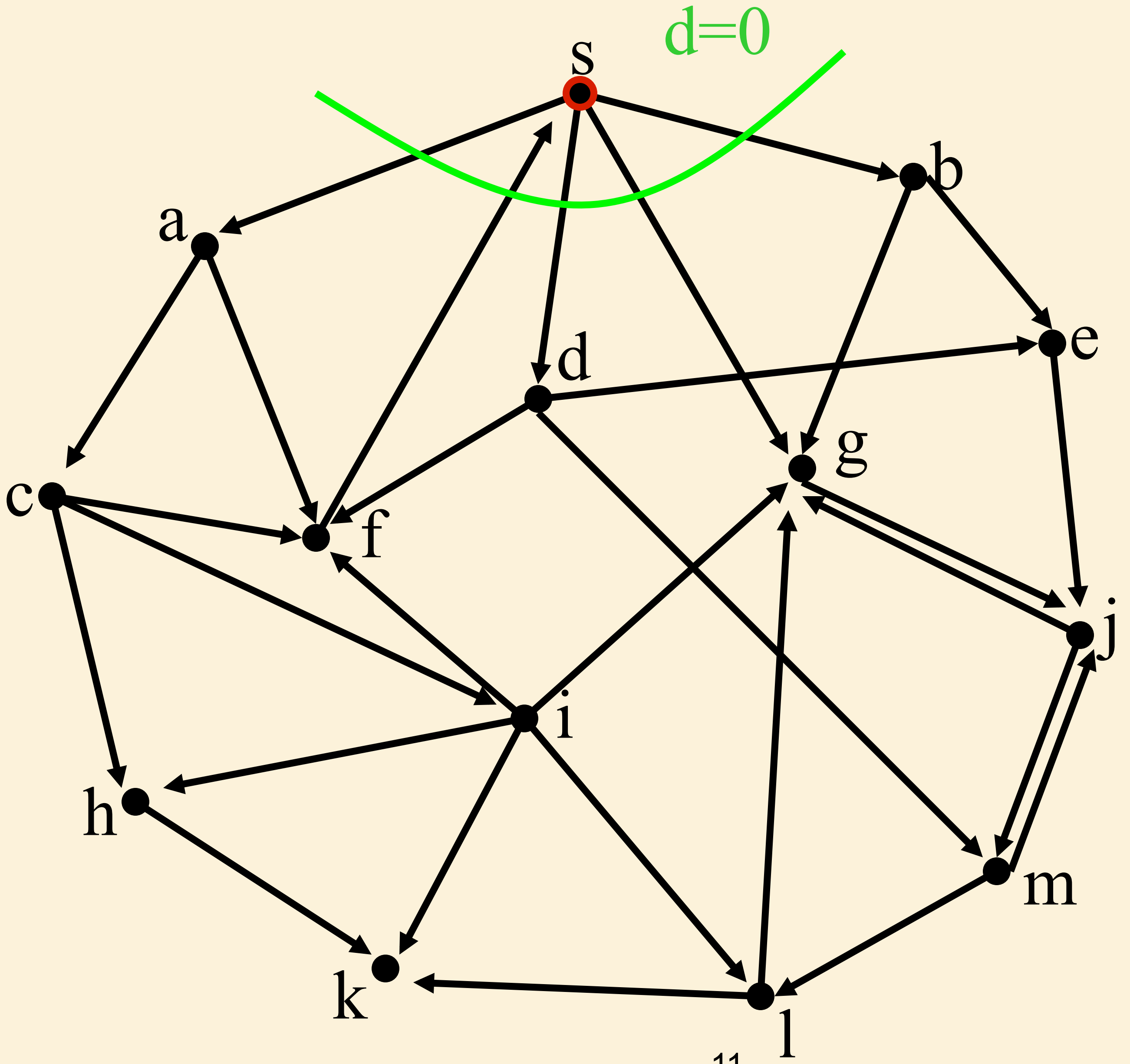
BFS

First-In First-Out (FIFO) queue stores 'just discovered' vertices

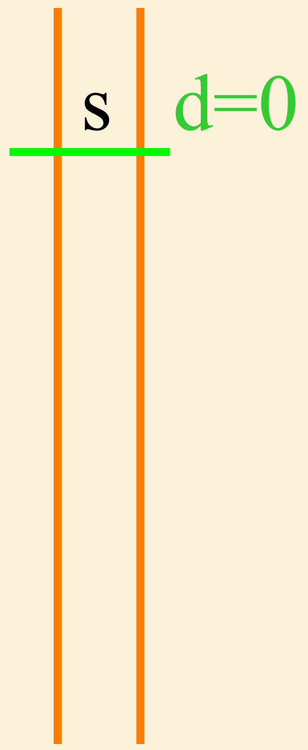
Found
Not Handled
Queue



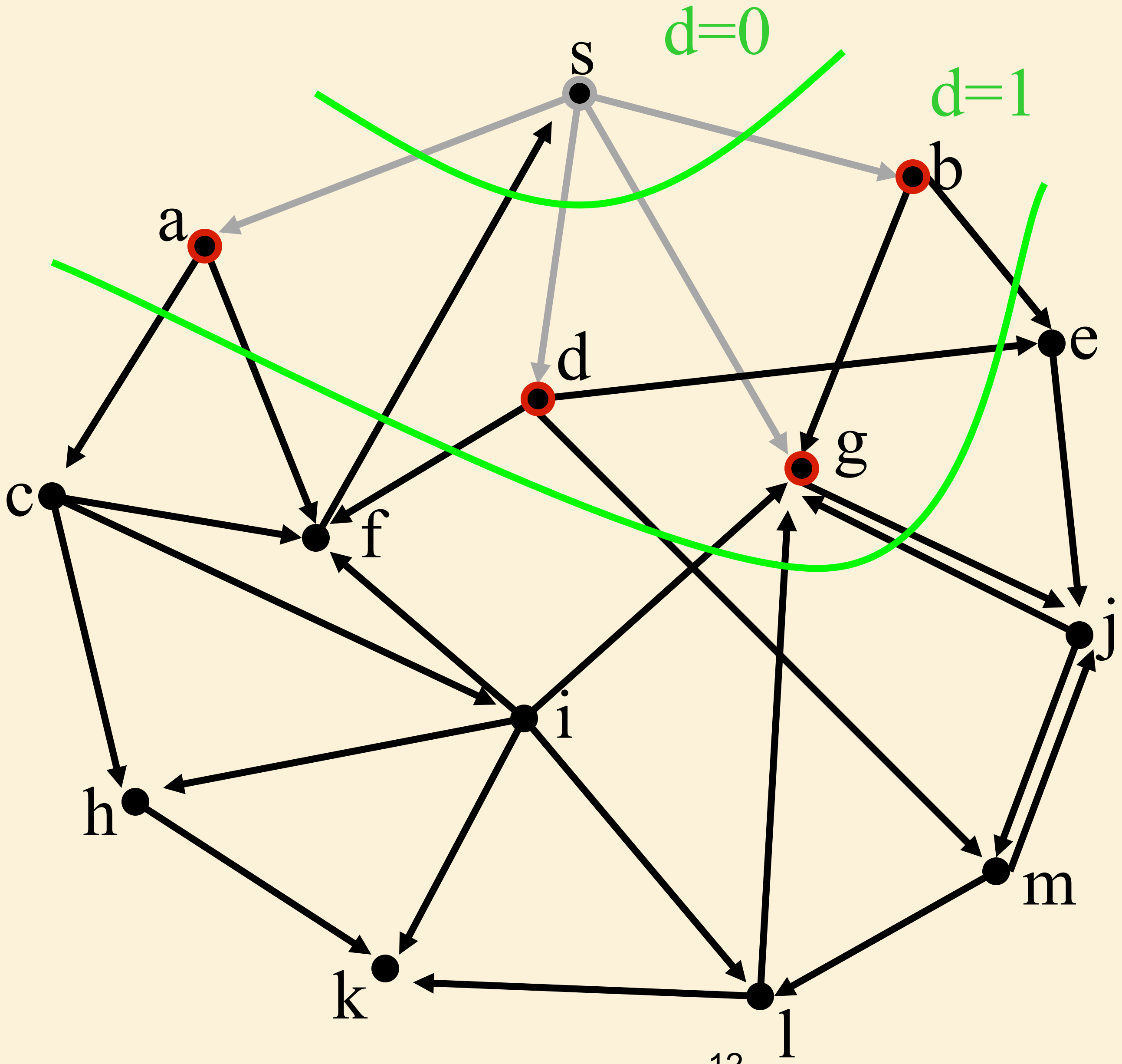
BFS



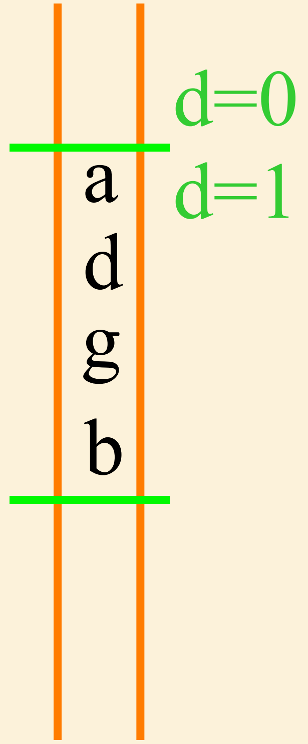
Found
Not Handled
Queue



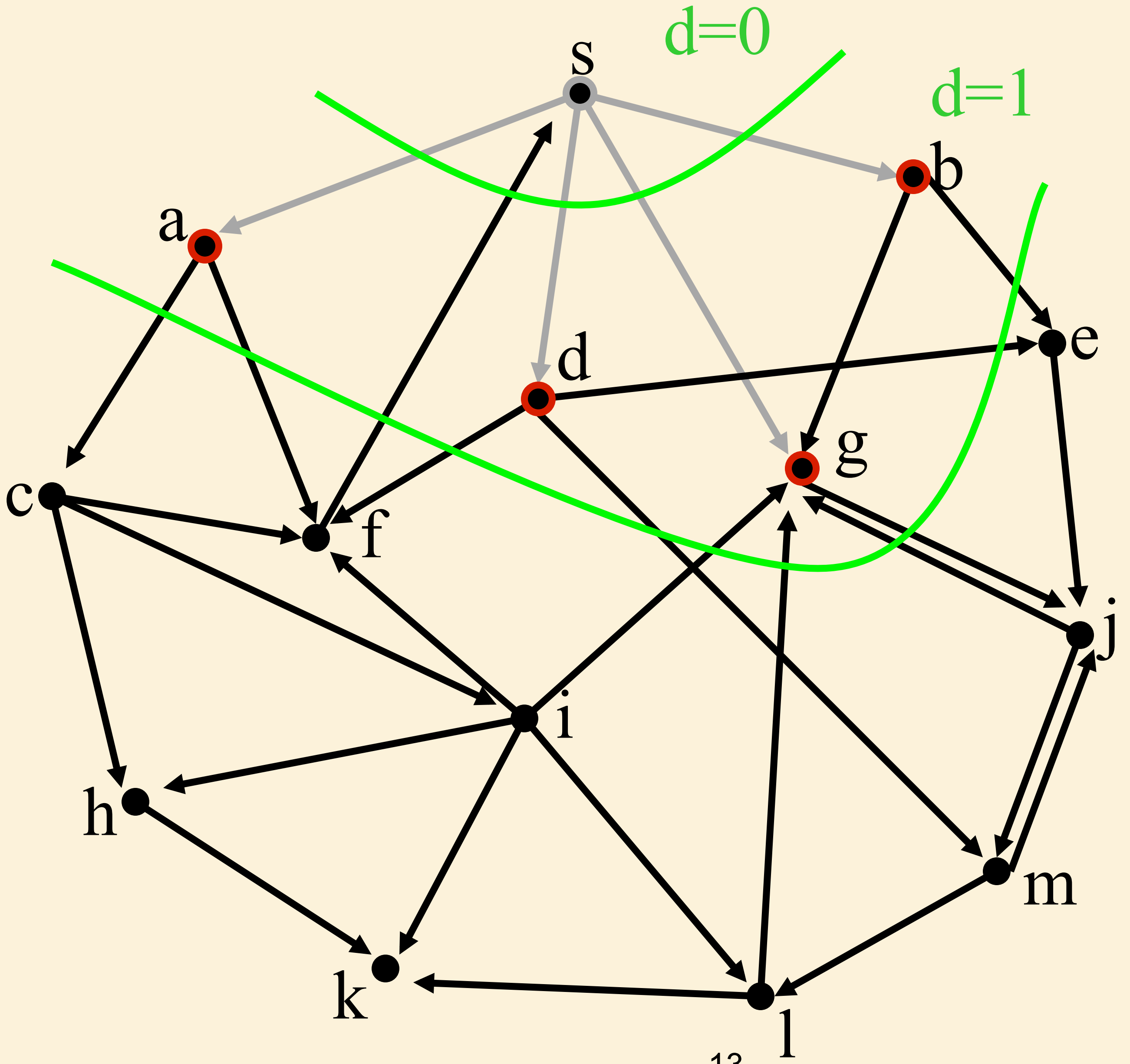
BFS



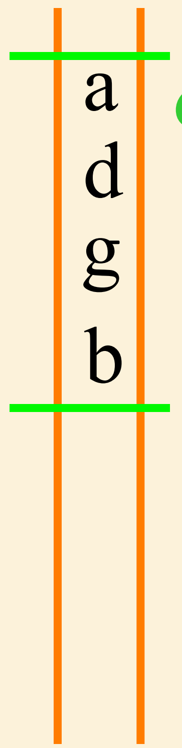
Found
Not Handled
Queue



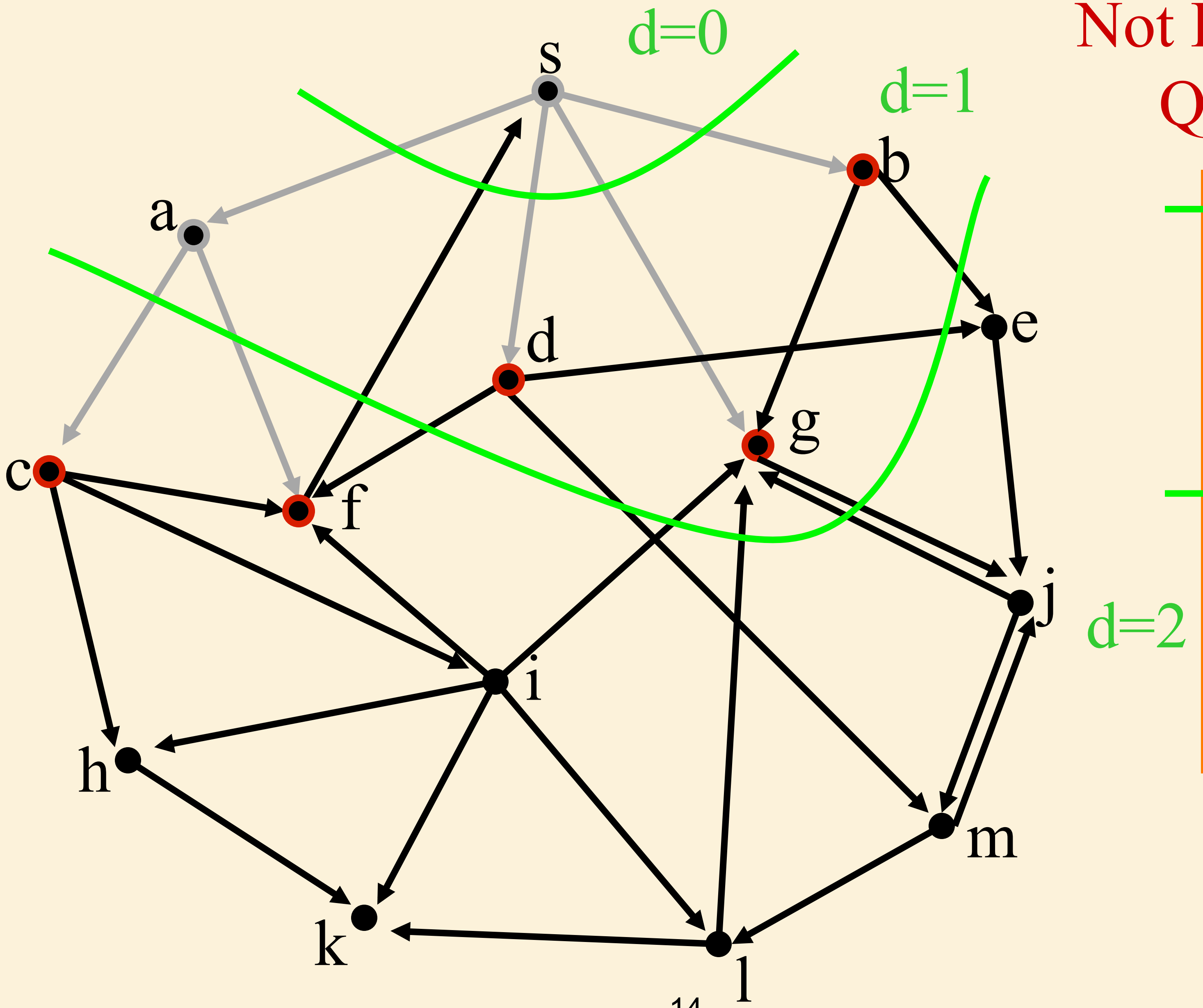
BFS



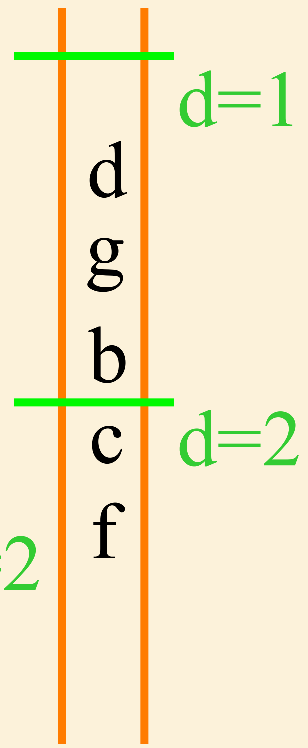
Found
Not Handled
Queue



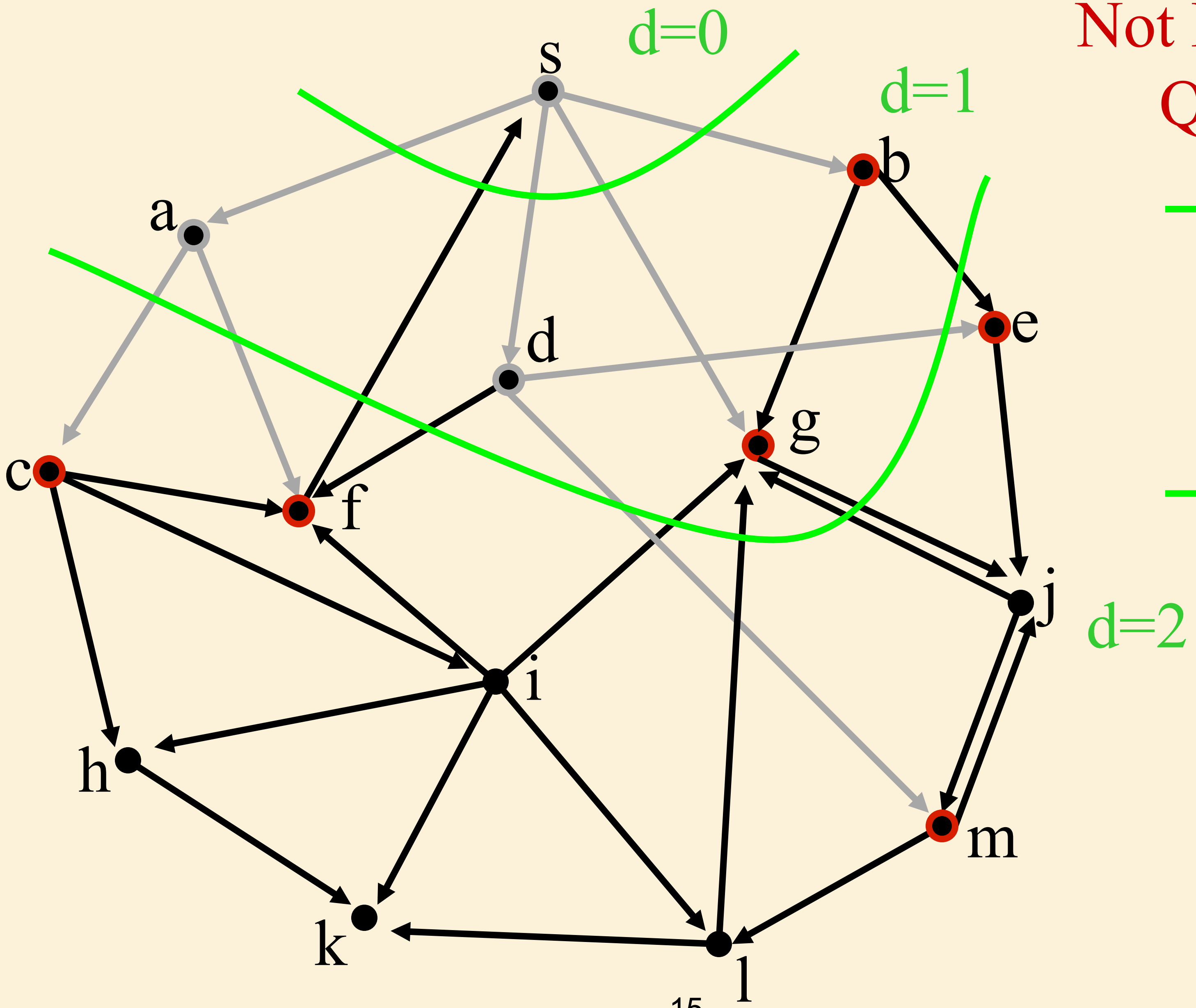
BFS



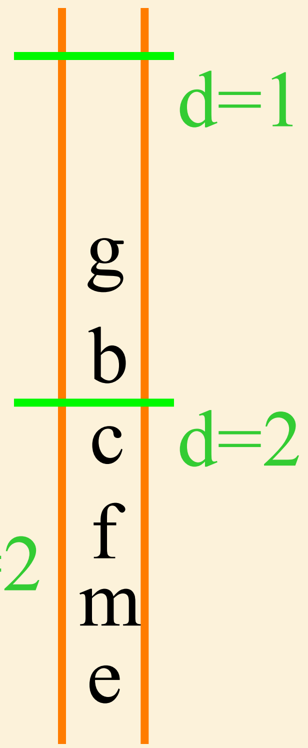
Found
Not Handled
Queue



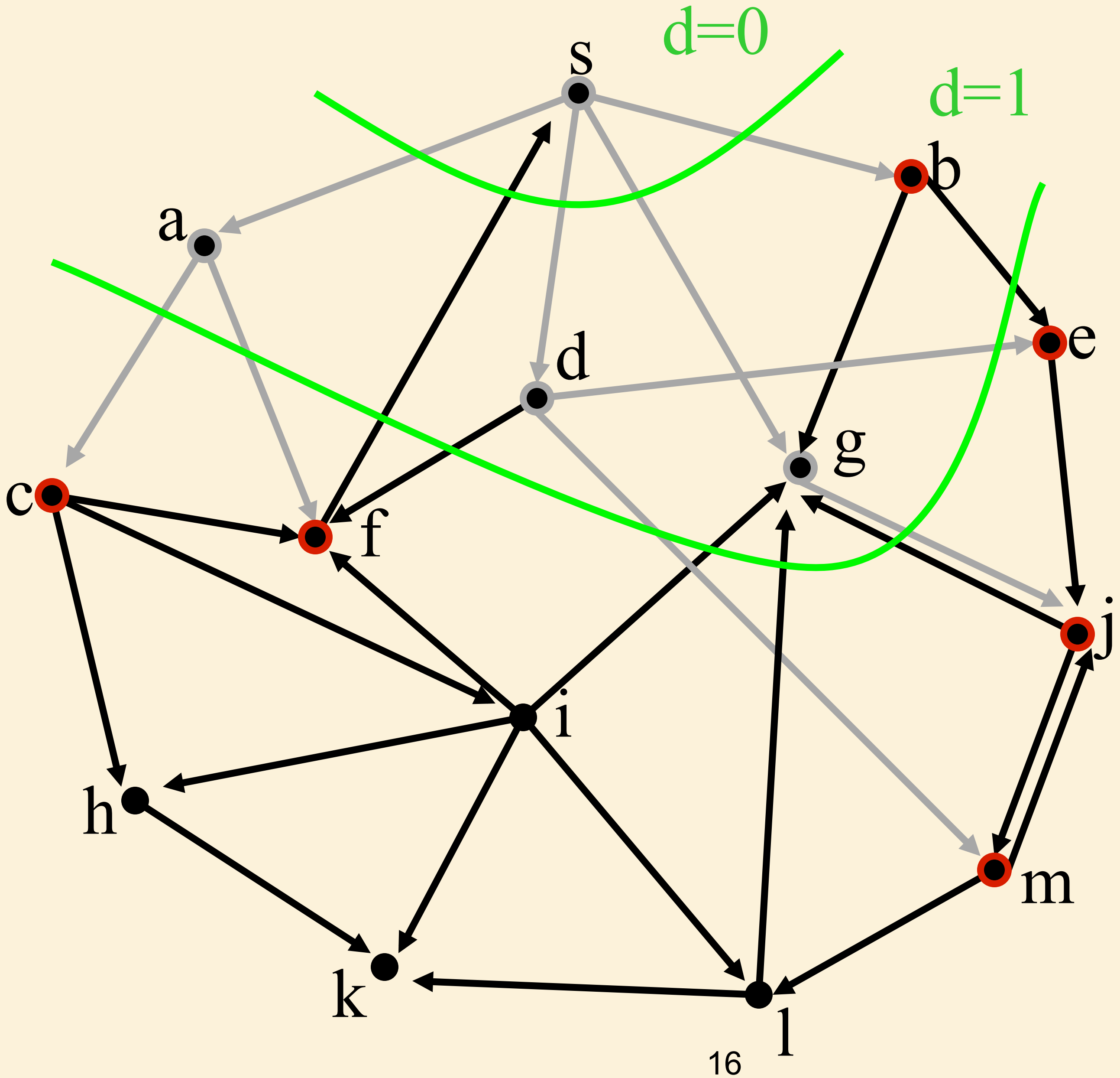
BFS



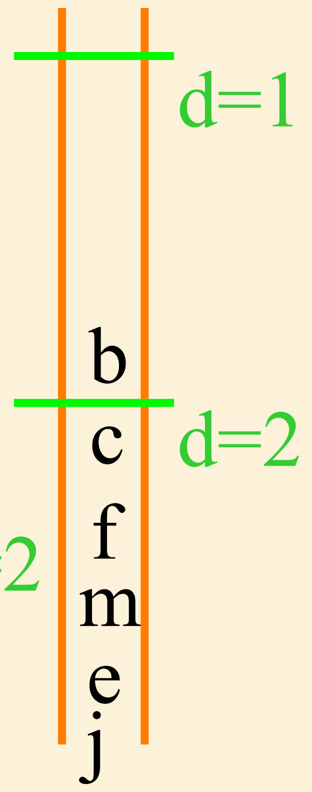
Found
Not Handled
Queue



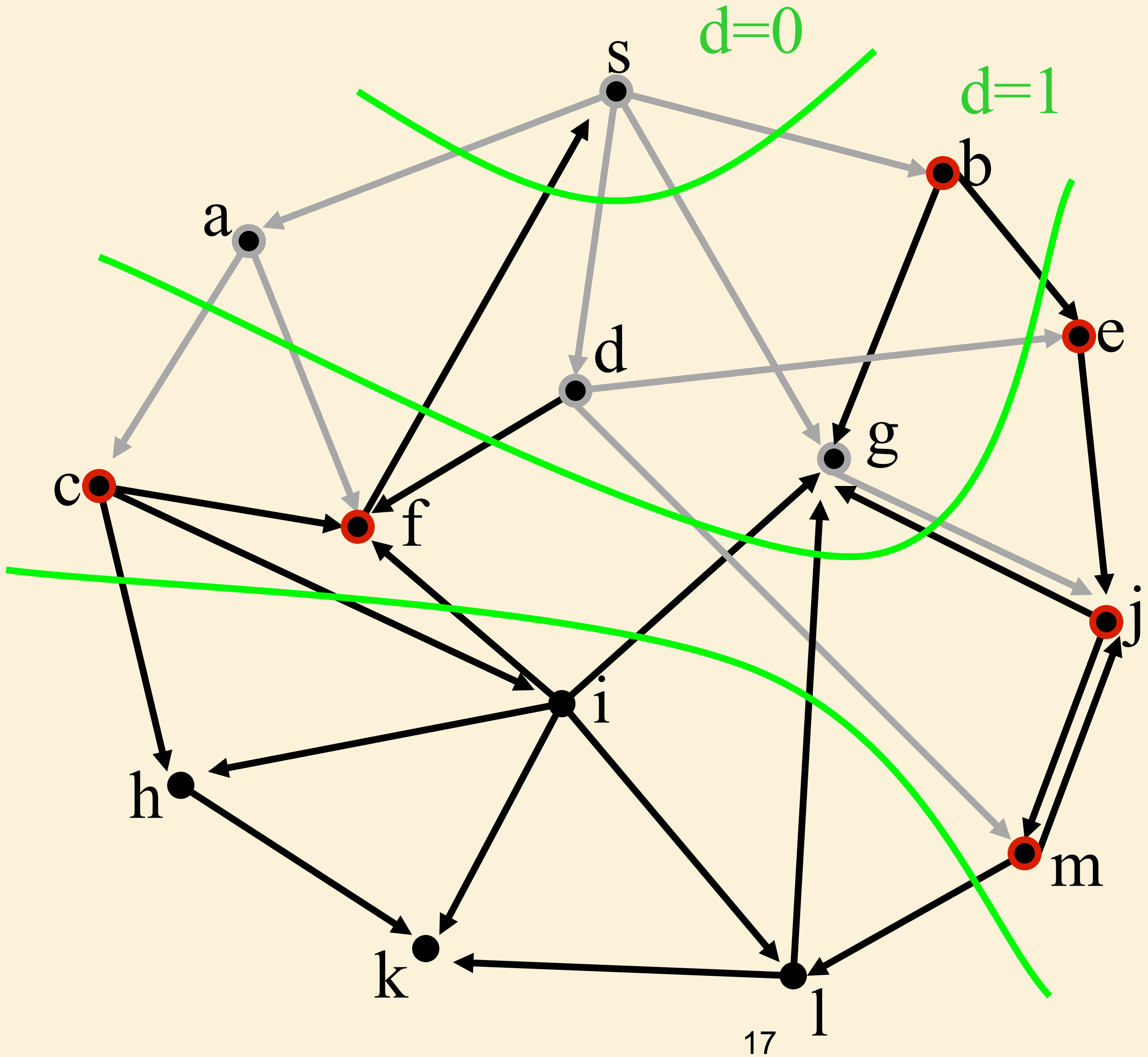
BFS



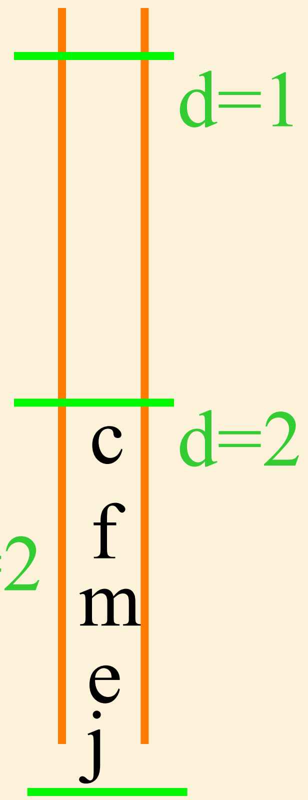
Found
Not Handled
Queue



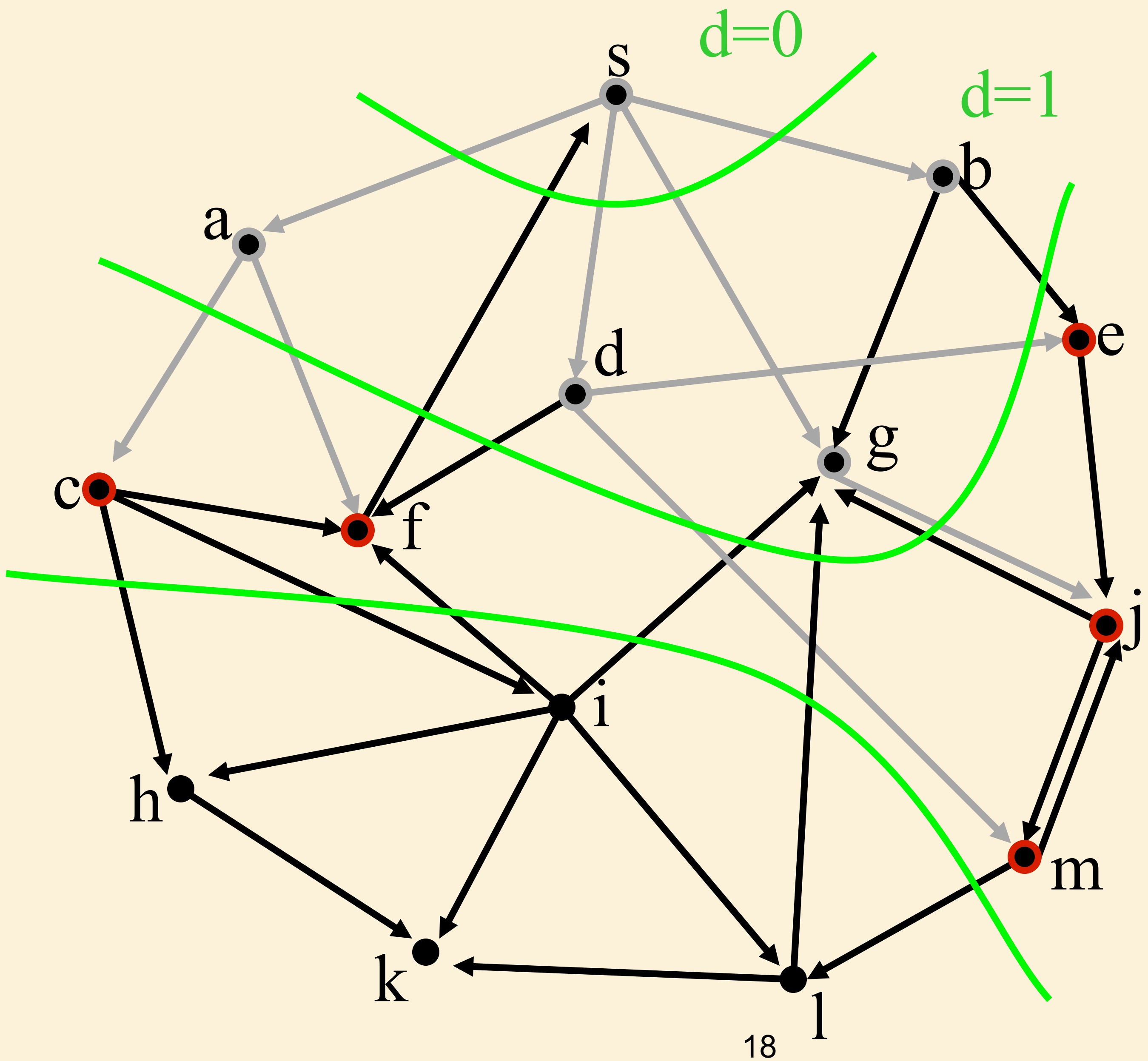
BFS

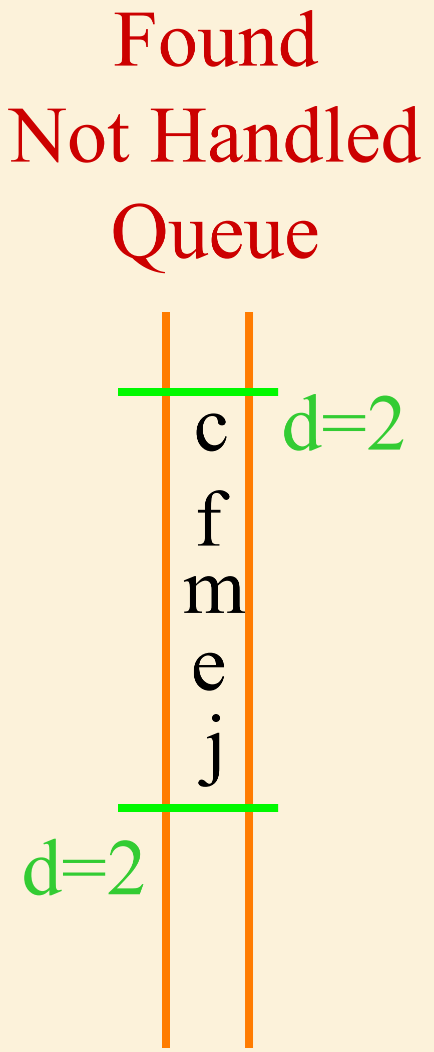


Found
Not Handled
Queue

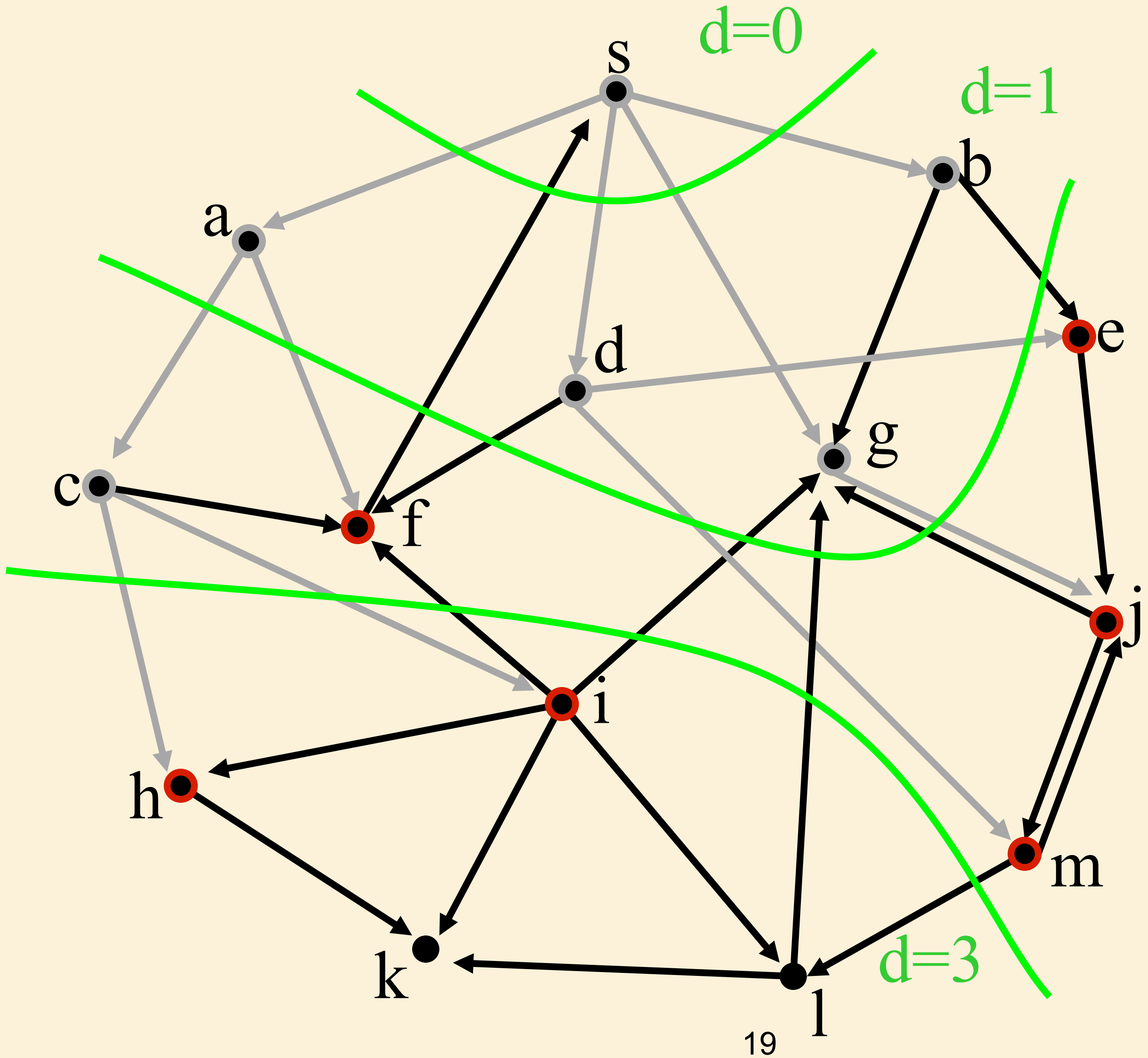


BFS

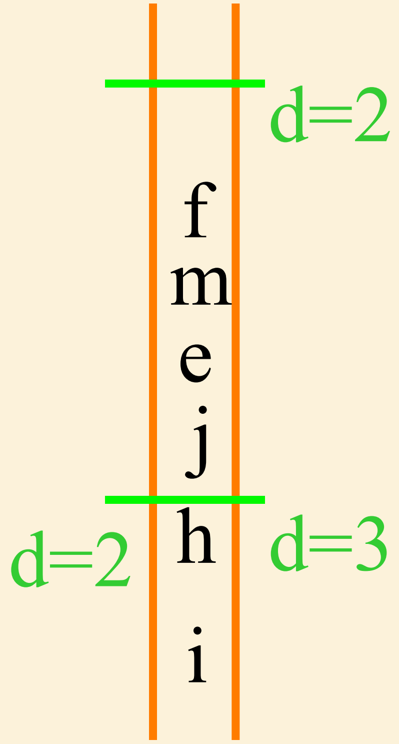




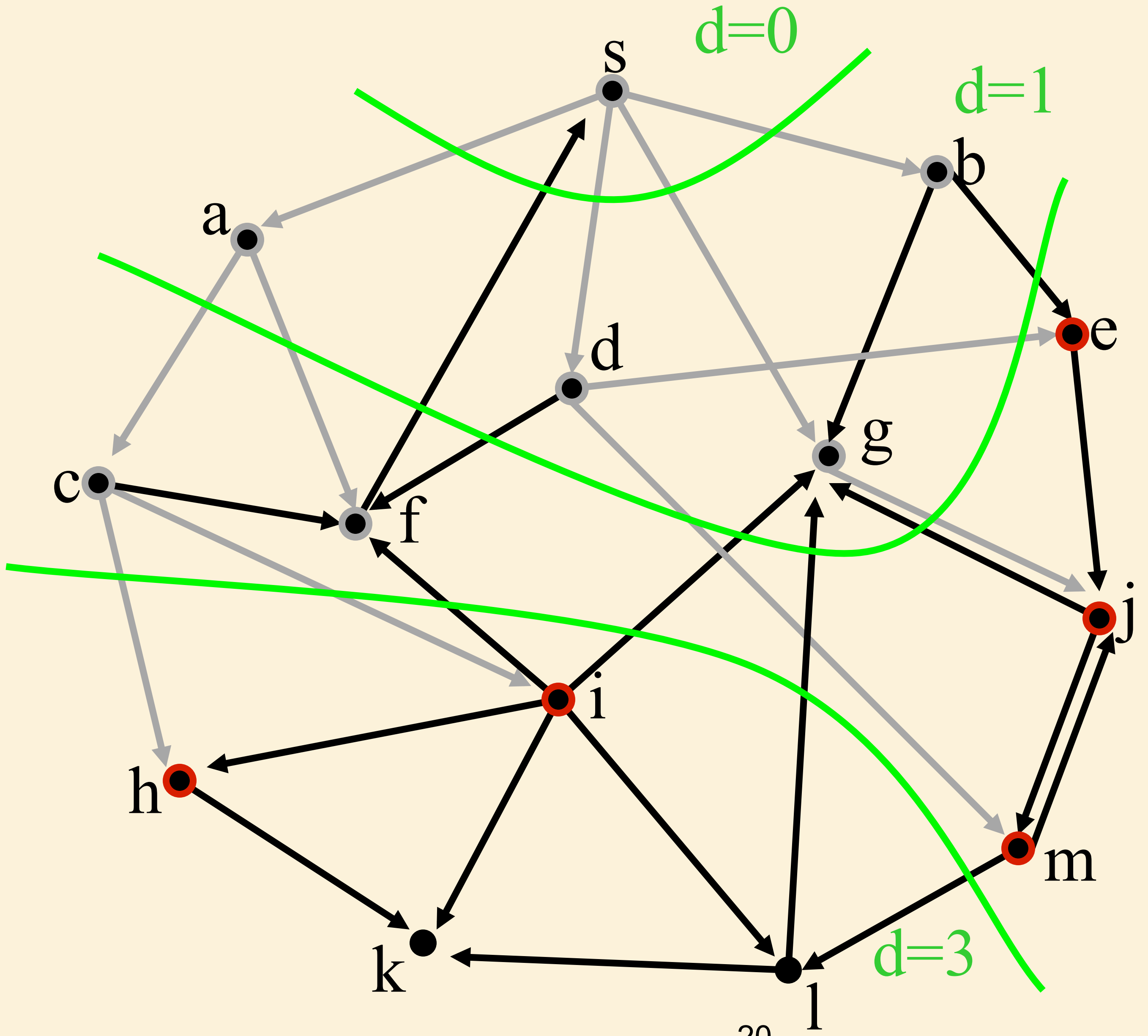
BFS



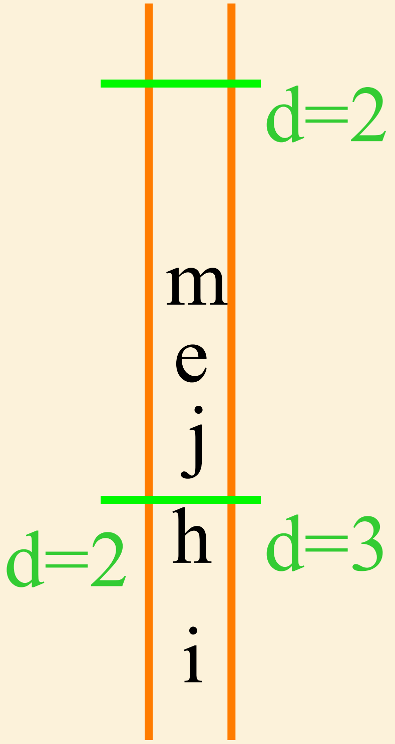
Found
Not Handled
Queue



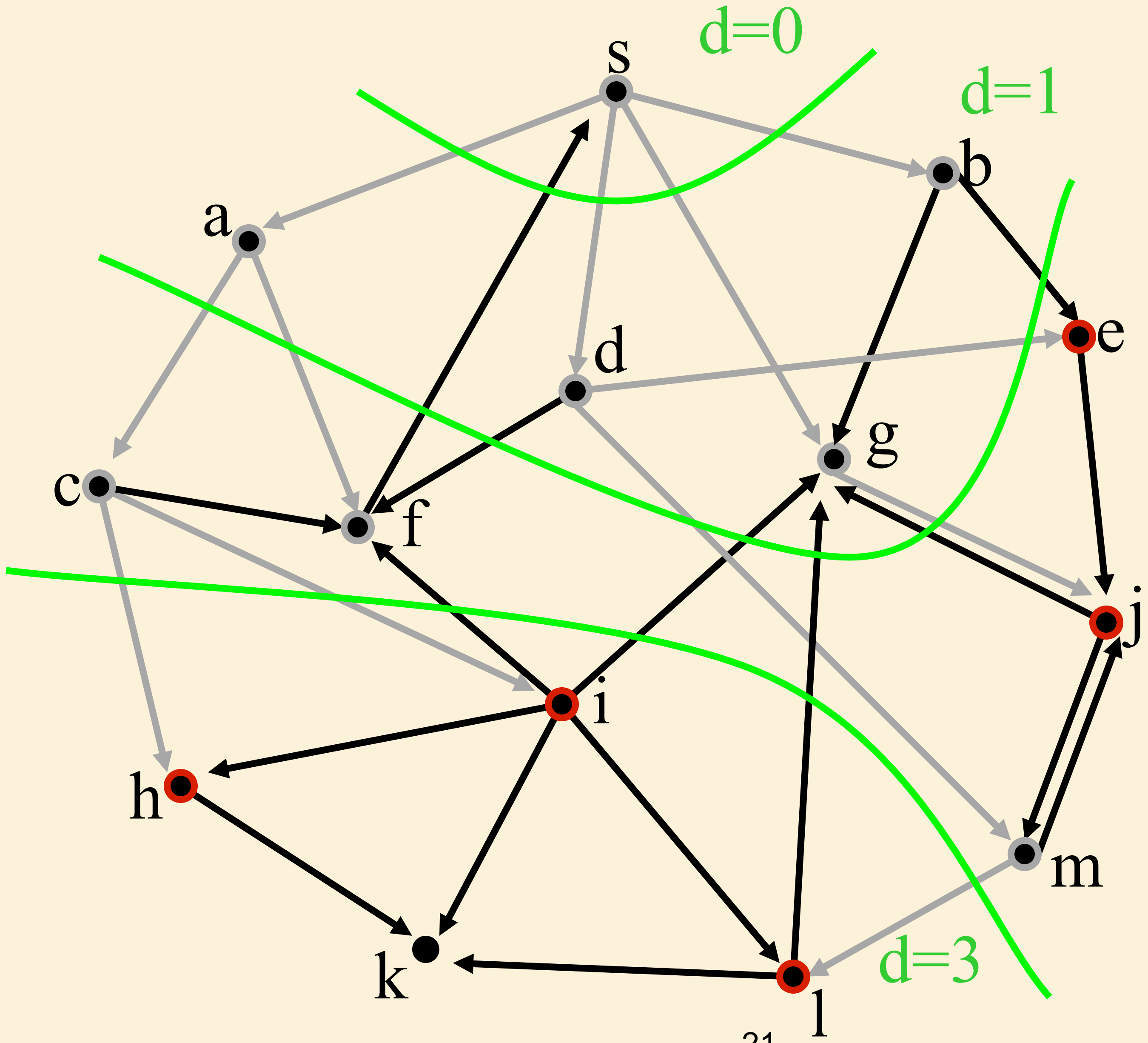
BFS

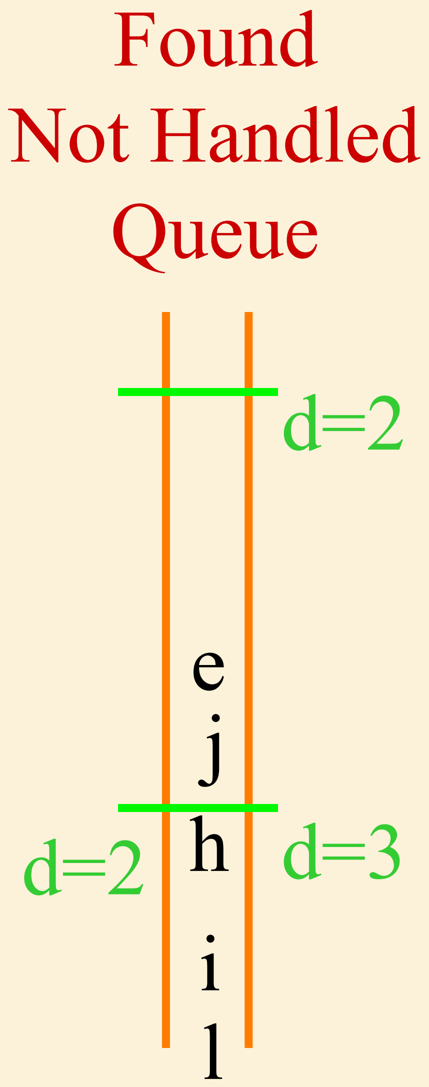


Found
Not Handled
Queue

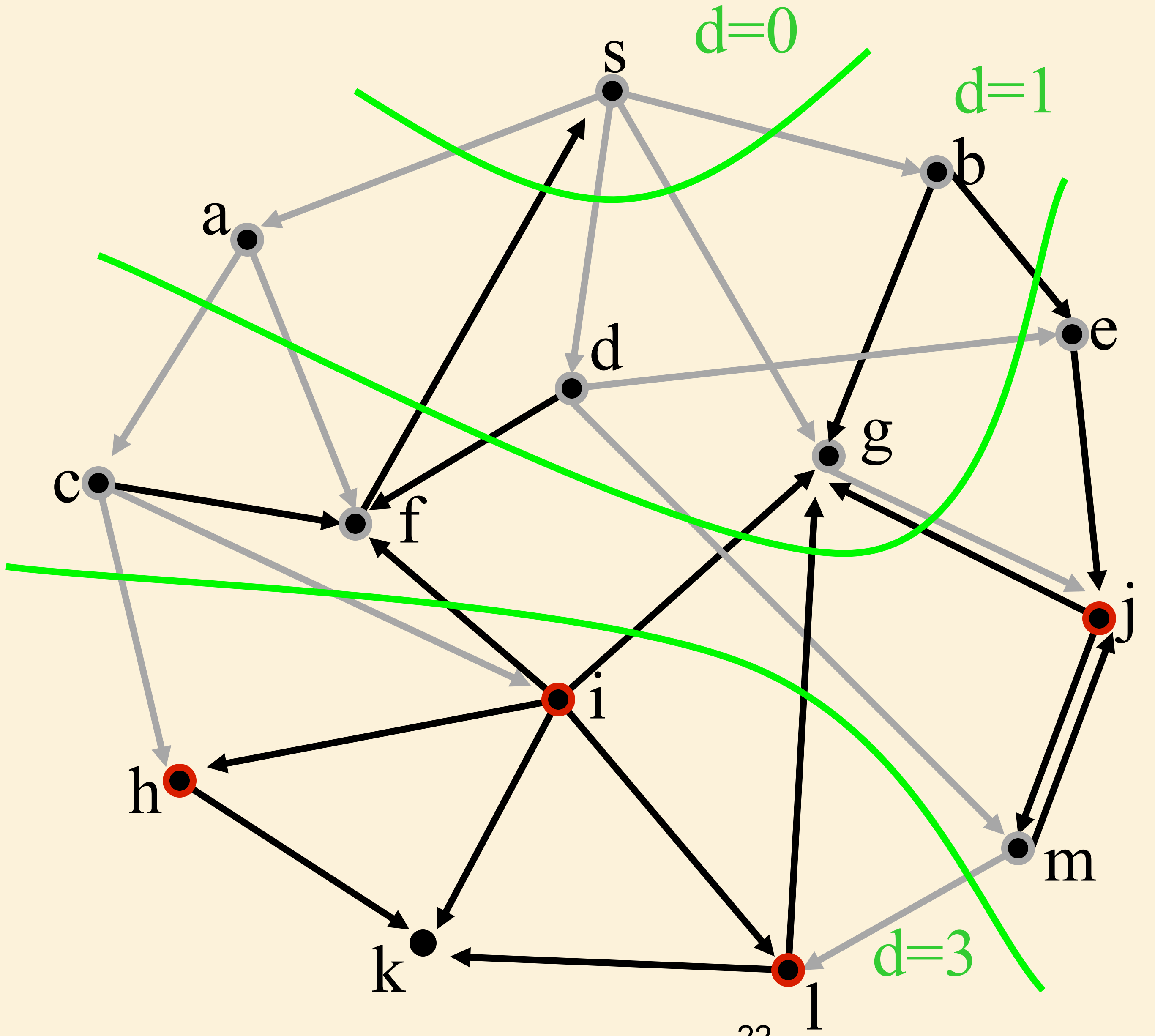


BFS

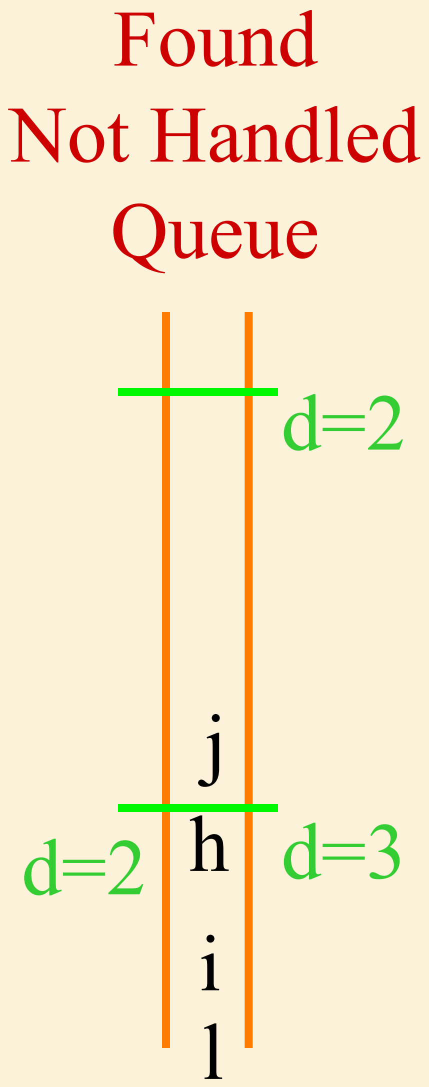

d=0
d=1
s
a
b
c
d
e
f
g
h
i
j
k
l
m
d=3


Found
Not Handled
Queue
d=2
e
j
d=2
h
d=3
i
l

BFS

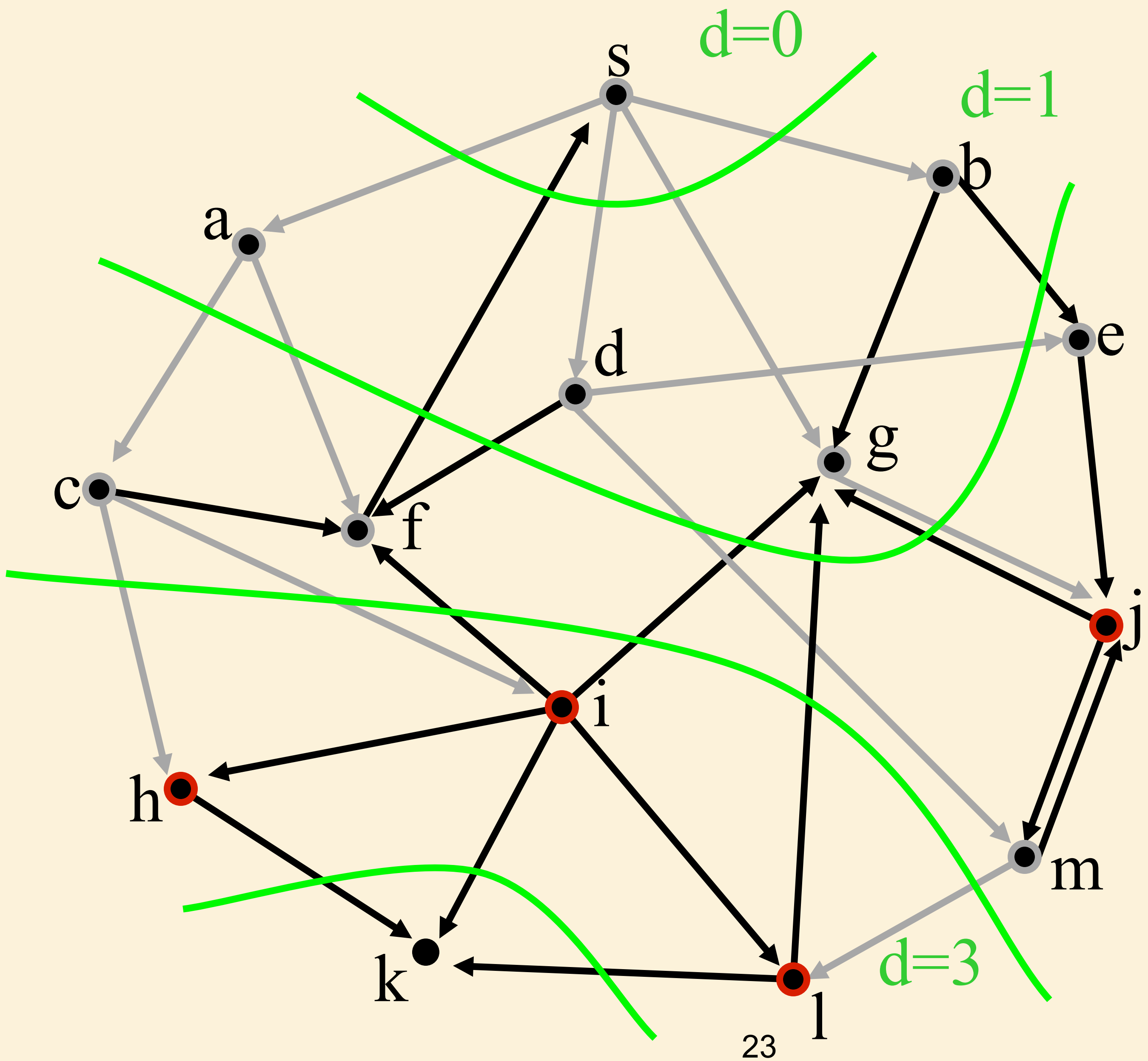


d=0
d=1
s
a
b
d
e
g
c
f
j
i
h
m
k
l
d=3

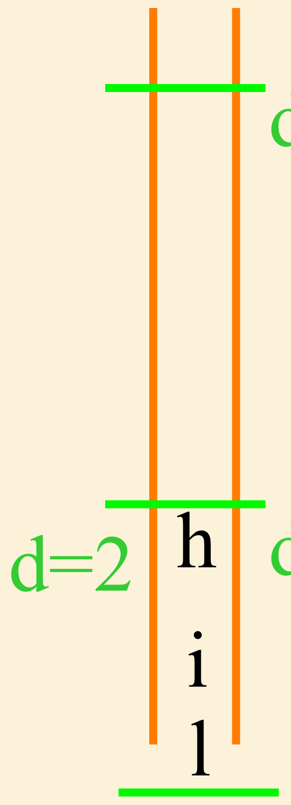


Found
Not Handled
Queue
d=2
j
h
i
l
d=2
d=3

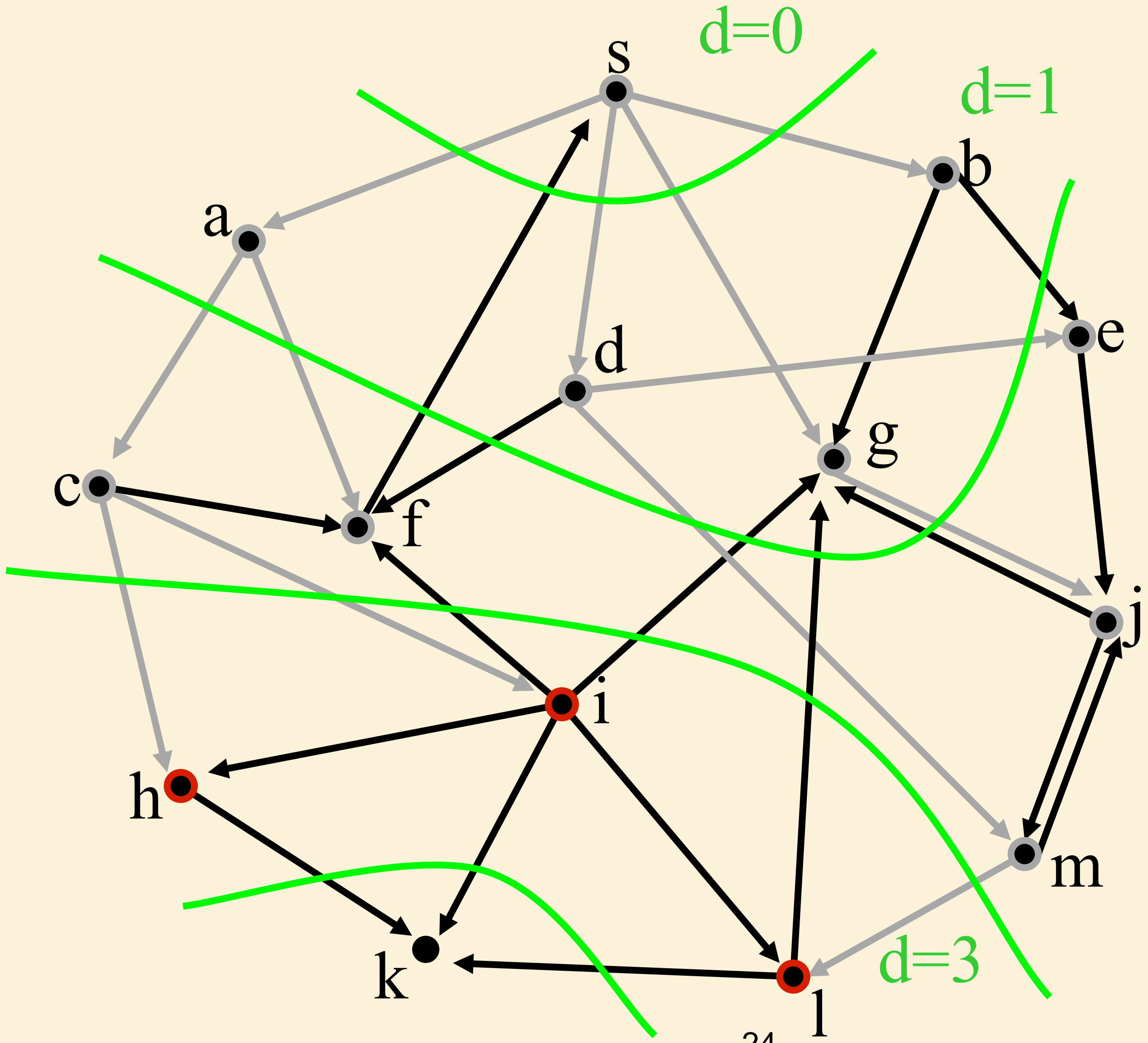
BFS



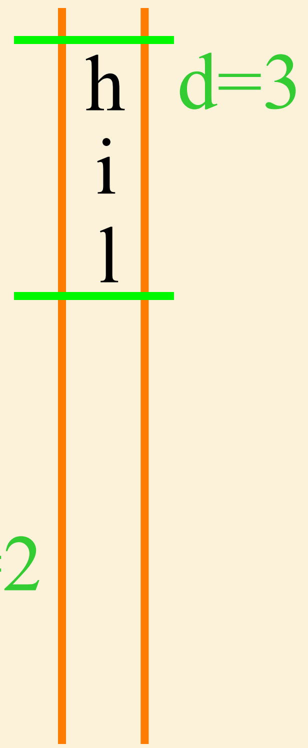
Found
Not Handled
Queue



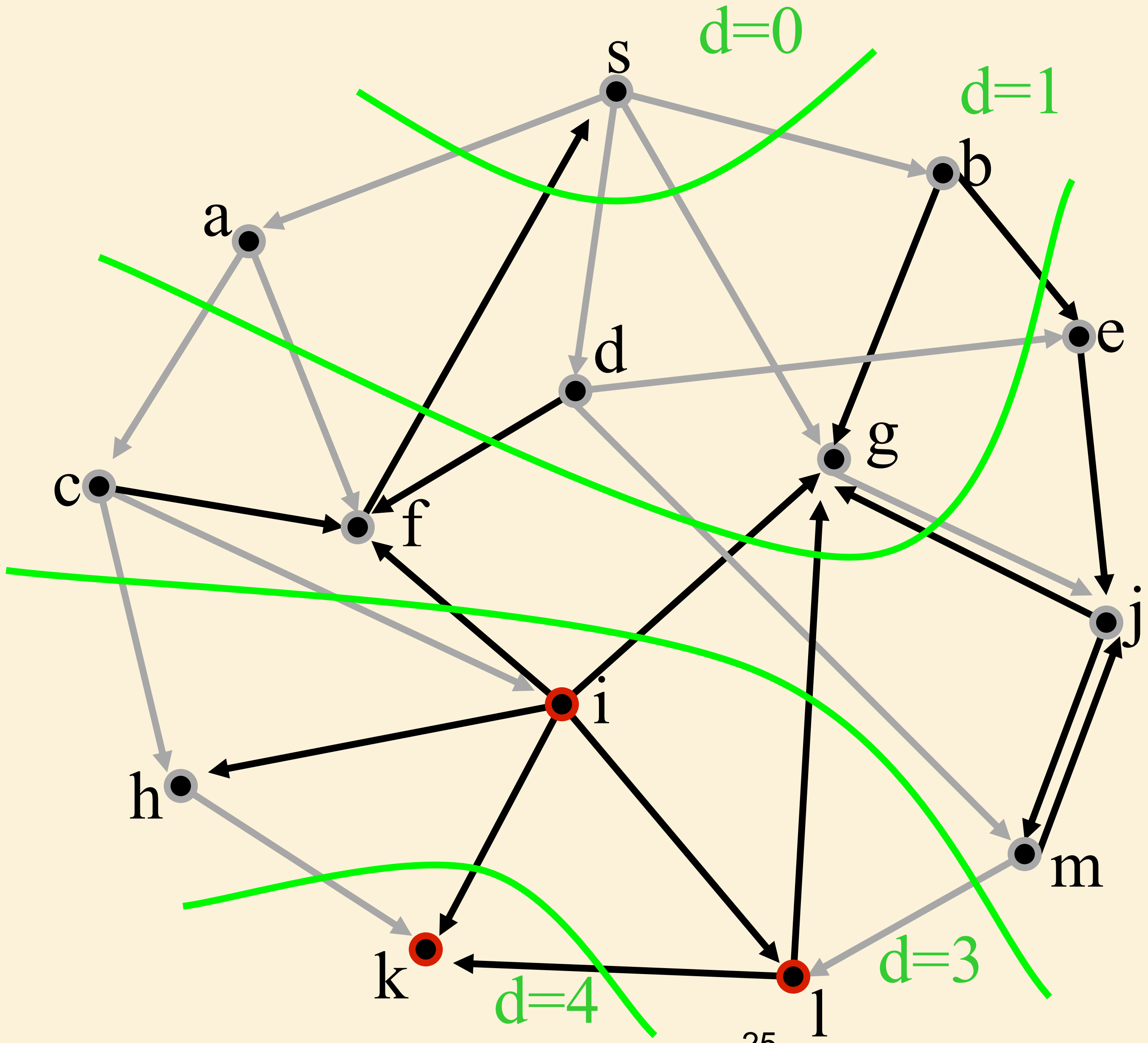
BFS



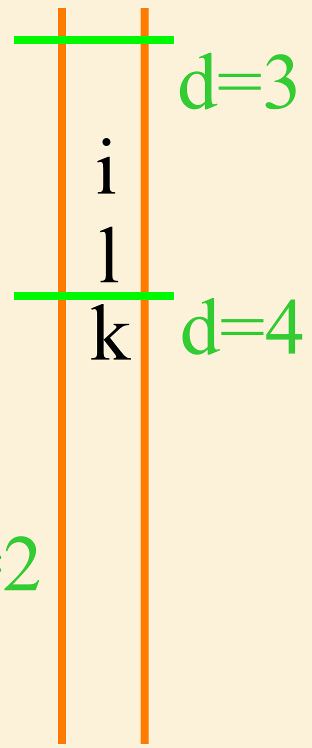
Found
Not Handled
Queue



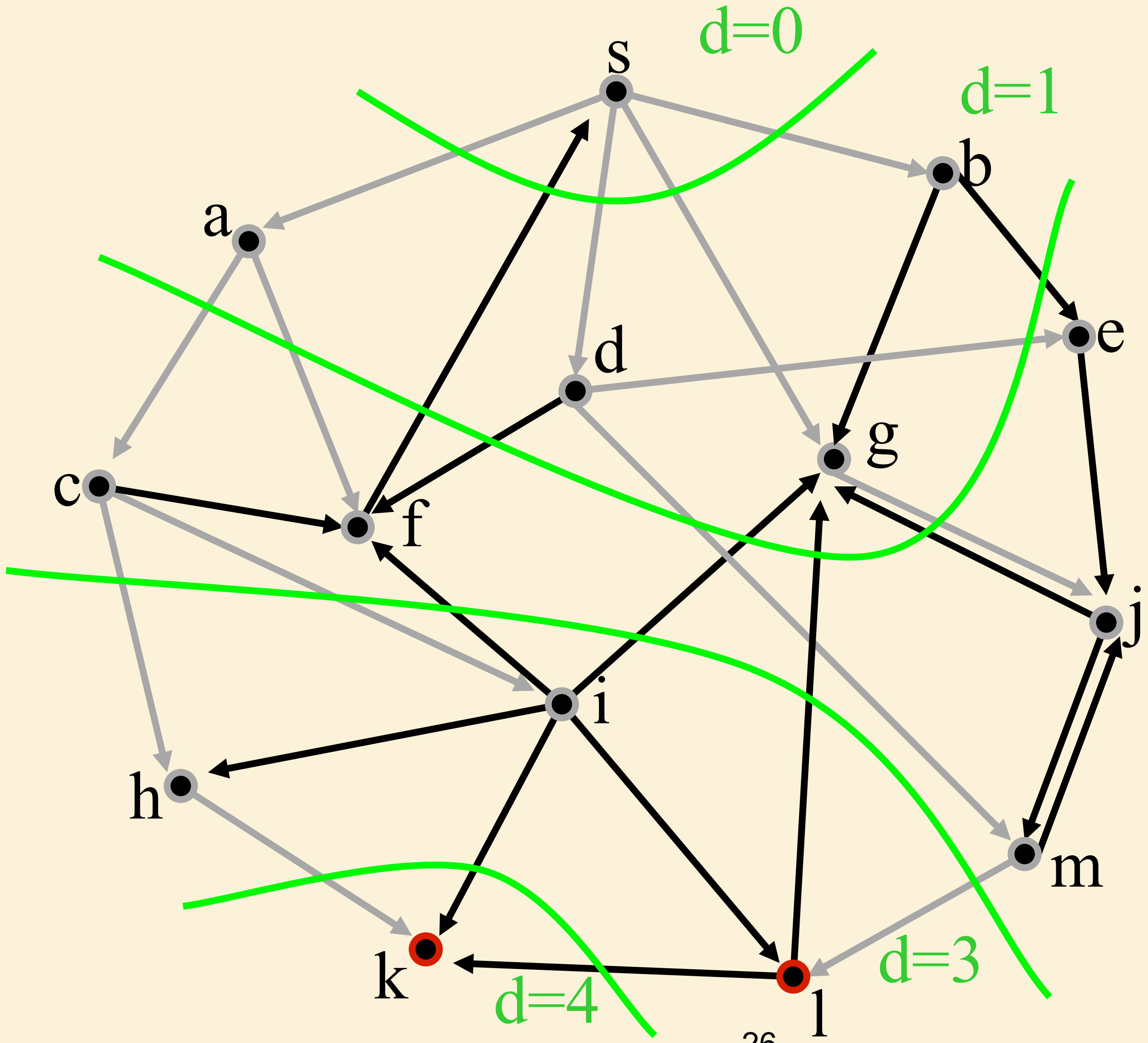
BFS



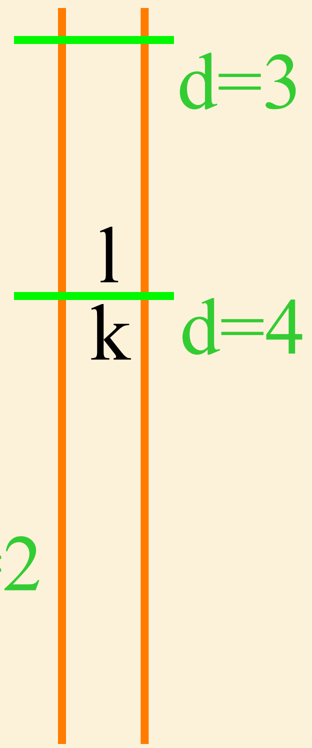
Found
Not Handled
Queue



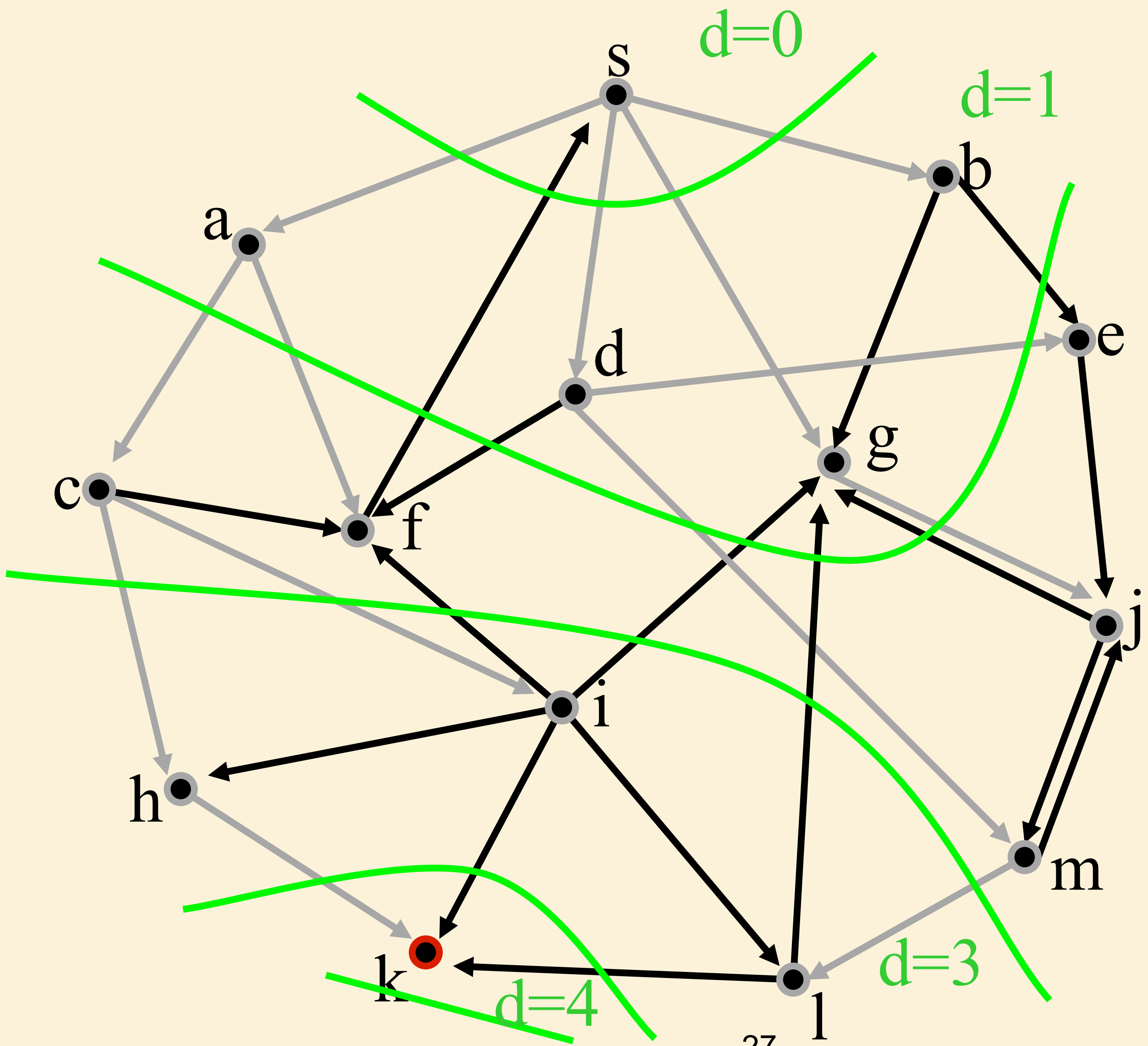
BFS



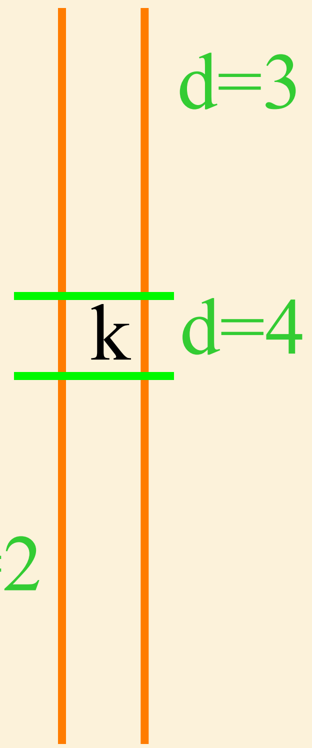
Found
Not Handled
Queue



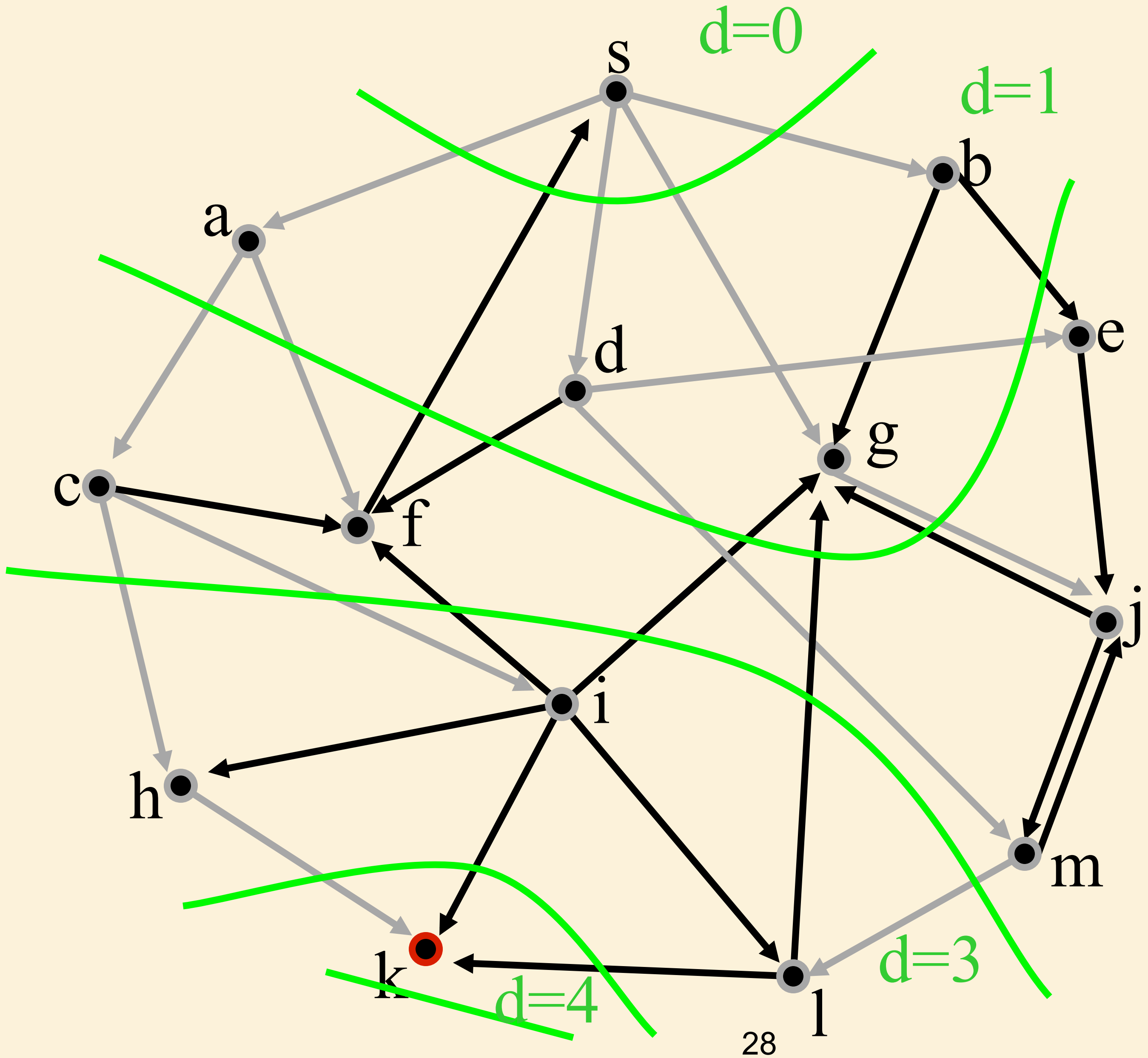
BFS



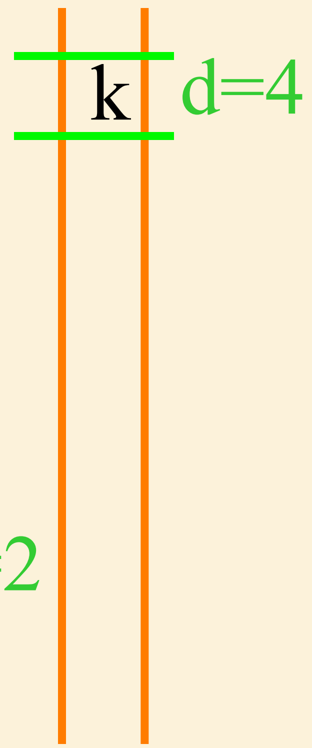
Found
Not Handled
Queue



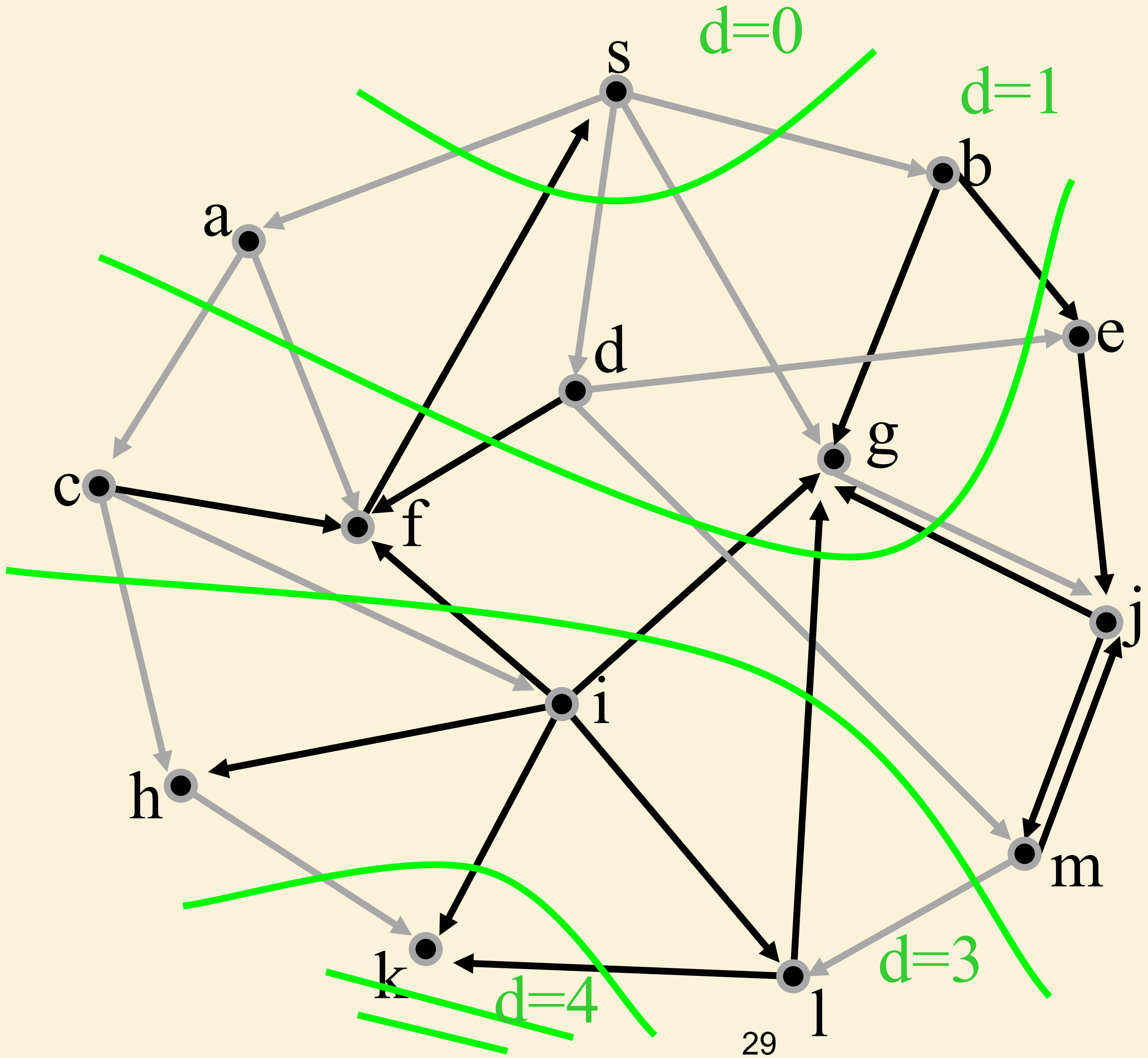
BFS



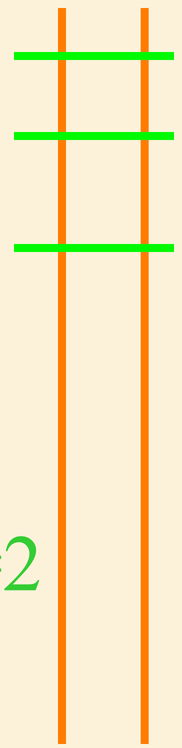
Found
Not Handled
Queue



BFS



Found
Not Handled
Queue



Breadth-First Search Algorithm

```
BFS($G, s$)
for each vertex $u \in V[G] - \{s\}$
    do $color[u] \leftarrow$ BLACK
       $d[u] \leftarrow \infty$
       $\pi[u] \leftarrow$ NIL
$color[s] \leftarrow$ RED
$d[s] \leftarrow 0$
$\pi[s] \leftarrow$ NIL
$Q \leftarrow \emptyset$
ENQUEUE($Q, s$)
while $Q \neq \emptyset$
    do $u \leftarrow$ DEQUEUE($Q$)
       for each $v \in Adj[u]$
           do if $color[v] =$ BLACK
                 then $color[v] \leftarrow$ RED
                      $d[v] \leftarrow d[u] + 1$
                      $\pi[v] \leftarrow u$
                      ENQUEUE($Q, v$)
       $color[u] \leftarrow$ GRAY
```

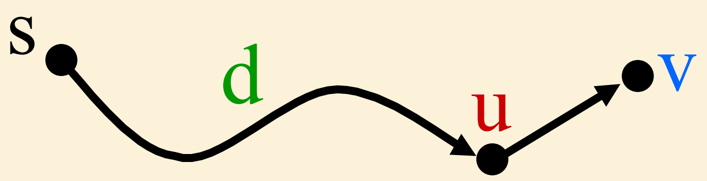
- Q is a FIFO queue.
- Each vertex assigned finite *d* value at most once.
- *Q* contains vertices with d values {*i*, …, *i*, *i*+1, …, *i*+1}
- *d* values assigned are monotonically increasing over time.

Breadth-First-Search is Greedy

- Vertices are handled:
  - in order of their discovery (FIFO queue)
  - Smallest *d* values first

Correctness

Basic Steps:

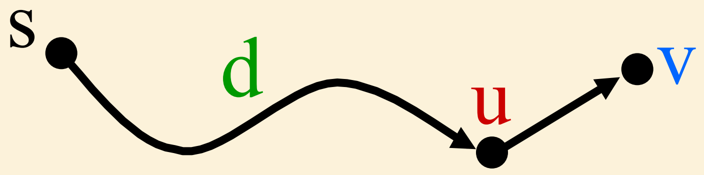


The shortest path to u has length d & there is an edge from u to v

There is a path to v with length d+1.

Correctness

- Vertices are discovered in order of their distance from the source vertex *s*.
- When we discover *v*, how do we know there is not a shorter path to *v*?
  - Because if there was, we would already have discovered it!



Correctness

Input: Graph $G = (V, E)$ (directed or undirected) and source vertex $s \in V$.

Output:

$d[v] =$ distance from s to v, $\forall v \in V$.

$\pi[v] = u$ such that (u,v) is last edge on shortest path from s to v.

Two-step proof:

On exit:

1. $d[v] \geq \delta(s,v) \forall v \in V$

2. $d[v] \not> \delta(s,v) \forall v \in V$

Claim 1. d is never too small: $d[v] \geq \delta(s,v) \forall v \in V$

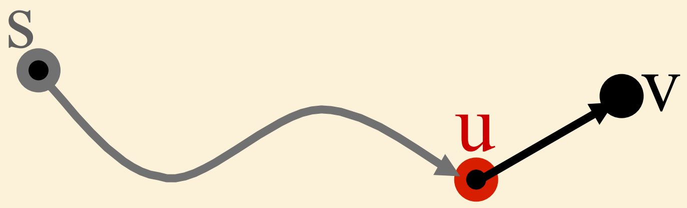
Proof: There exists a path from s to v of length $d[v]$.

By Induction:

Suppose it is true for all vertices thus far discovered (red and grey).

v is discovered from some adjacent vertex u being handled.

$$\begin{aligned} \rightarrow d[v] &= d[u]+1 \\ &\geq \delta(s,u)+1 \\ &\geq \delta(s,v) \end{aligned}$$



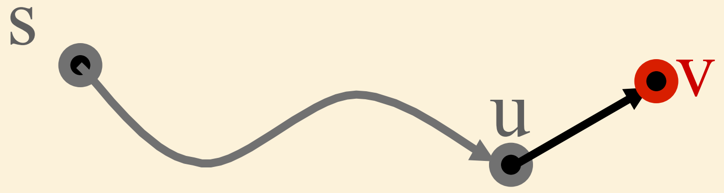
since each vertex v is assigned a d value exactly once,

it follows that on exit, $d[v] \geq \delta(s,v) \forall v \in V$.

Claim 1. *d* is never too small: $d[v] \geq \delta(s,v) \forall v \in V$

Proof: There exists a path from *s* to *v* of length *d*[*v*].

```
BFS(G, s)
 1  for each vertex $u \in V[G] - \{s\}$
 2        do $color[u] \leftarrow$ BLACK
 3           $d[u] \leftarrow \infty$
 4           $\pi[u] \leftarrow$ NIL
 5  $color[s] \leftarrow$ RED
 6  $d[s] \leftarrow 0$
 7  $\pi[s] \leftarrow$ NIL
 8  $Q \leftarrow \emptyset$
 9  ENQUEUE(Q, s)
10  while $Q \neq \emptyset$      $\leftarrow$ <LI>: $d[v] \geq \delta(s,v) \forall$ 'discovered' (red or grey) $v \in V$
11        do $u \leftarrow$ DEQUEUE(Q)
12           for each $v \in Adj[u]$
13               do if $color[v] =$ BLACK
14                     then $color[v] \leftarrow$ RED
15                          $d[v] \leftarrow d[u] + 1$      $\geq \delta(s,u) + 1 \geq \delta(s,v)$
16                          $\pi[v] \leftarrow u$
17                          ENQUEUE(Q, v)
18           $color[u] \leftarrow$ GRAY
```



Claim 2. d is never too big: $d[v] \le \delta(s,v) \forall v \in V$

Proof by contradiction:

Suppose one or more vertices receive a d value greater than δ.

Let v be the vertex with minimum $\delta(s,v)$ that receives such a d value.

Suppose that v is discovered and assigned this d value when vertex x is dequeued.

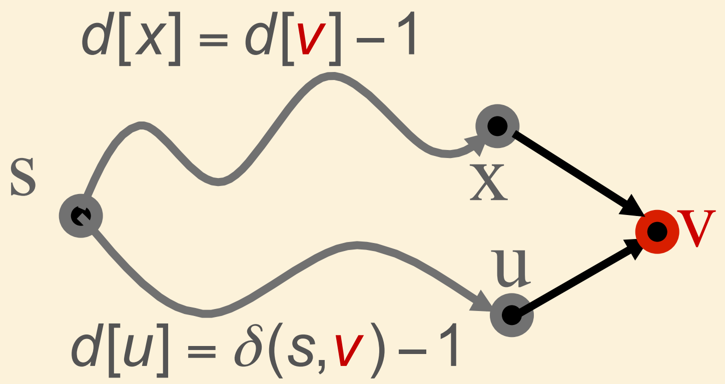
Let u be v's predecessor on a shortest path from s to v.

Then

$$\delta(s,v) < d[v]$$

$$\rightarrow \delta(s,v) - 1 < d[v] - 1$$

$$\rightarrow d[u] < d[x]$$



Recall: vertices are dequeued in increasing order of d value.

$\rightarrow$ u was dequeued before x.

$\rightarrow d[v] = d[u] + 1 = \delta(s,v)$ **Contradiction!**

Correctness

Claim 1. d is never too small: $d[v] \geq \delta(s,v) \forall v \in V$

Claim 2. d is never too big: $d[v] \leq \delta(s,v) \forall v \in V$

$\Rightarrow d$ is just right: $d[v] = \delta(s,v) \forall v \in V$

Progress?

- On every iteration one vertex is processed (turns gray).

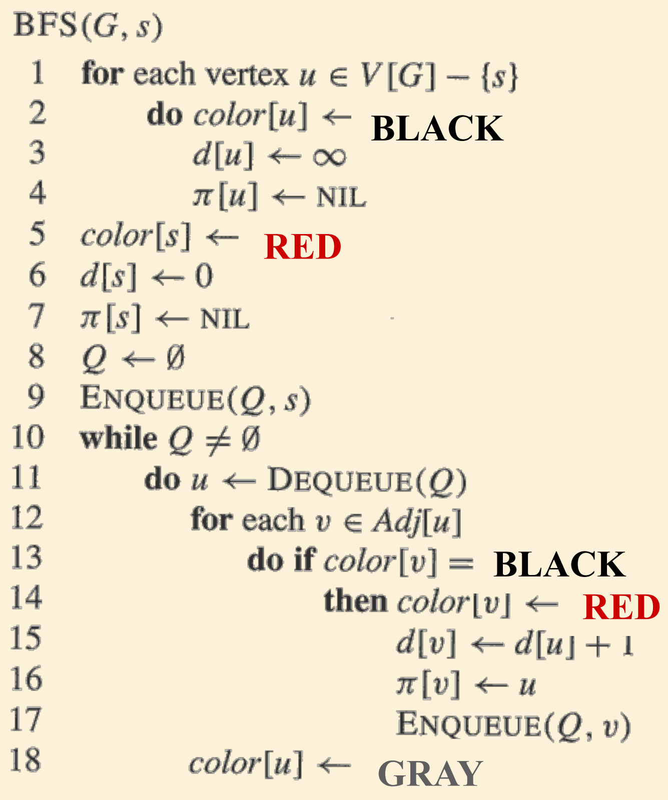
```
BFS($G, s$)
for each vertex $u \in V[G] - \{s\}$
    do $color[u] \leftarrow$ BLACK
       $d[u] \leftarrow \infty$
       $\pi[u] \leftarrow$ NIL
$color[s] \leftarrow$ RED
$d[s] \leftarrow 0$
$\pi[s] \leftarrow$ NIL
$Q \leftarrow \emptyset$
ENQUEUE($Q, s$)
while $Q \neq \emptyset$
    do $u \leftarrow$ DEQUEUE($Q$)
       for each $v \in Adj[u]$
           do if $color[v] =$ BLACK
                then $color[v] \leftarrow$ RED
                     $d[v] \leftarrow d[u] + 1$
                     $\pi[v] \leftarrow u$
                     ENQUEUE($Q, v$)
       $color[u] \leftarrow$ GRAY
```

Running Time

Each vertex is enqueued at most once $\rightarrow O(V)$

Each entry in the adjacency lists is scanned at most once $\rightarrow$ O(E)

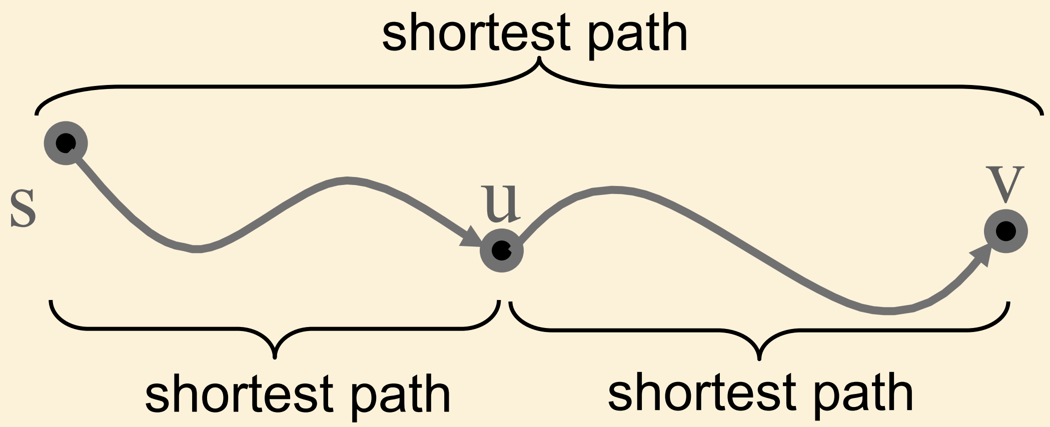
Thus run time is $O(V + E)$.



```
BFS($G, s$)
for each vertex $u \in V[G] - \{s\}$
    do $color[u] \leftarrow$ BLACK
       $d[u] \leftarrow \infty$
       $\pi[u] \leftarrow$ NIL
$color[s] \leftarrow$ RED
$d[s] \leftarrow 0$
$\pi[s] \leftarrow$ NIL
$Q \leftarrow \emptyset$
ENQUEUE($Q, s$)
while $Q \neq \emptyset$
    do $u \leftarrow$ DEQUEUE($Q$)
       for each $v \in Adj[u]$
           do if $color[v] =$ BLACK
                 then $color[v] \leftarrow$ RED
                      $d[v] \leftarrow d[u] + 1$
                      $\pi[v] \leftarrow u$
                      ENQUEUE($Q, v$)
       $color[u] \leftarrow$ GRAY
```

Optimal Substructure Property

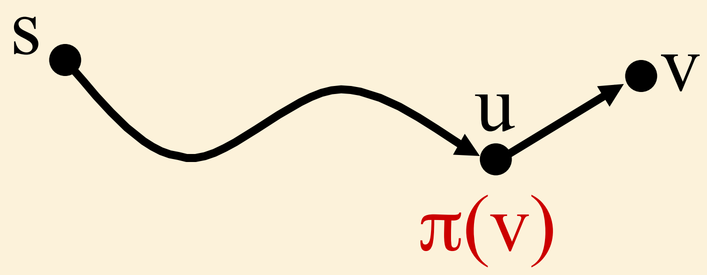
- The shortest path problem has the optimal substructure property:
  - Every subpath of a shortest path is a shortest path.

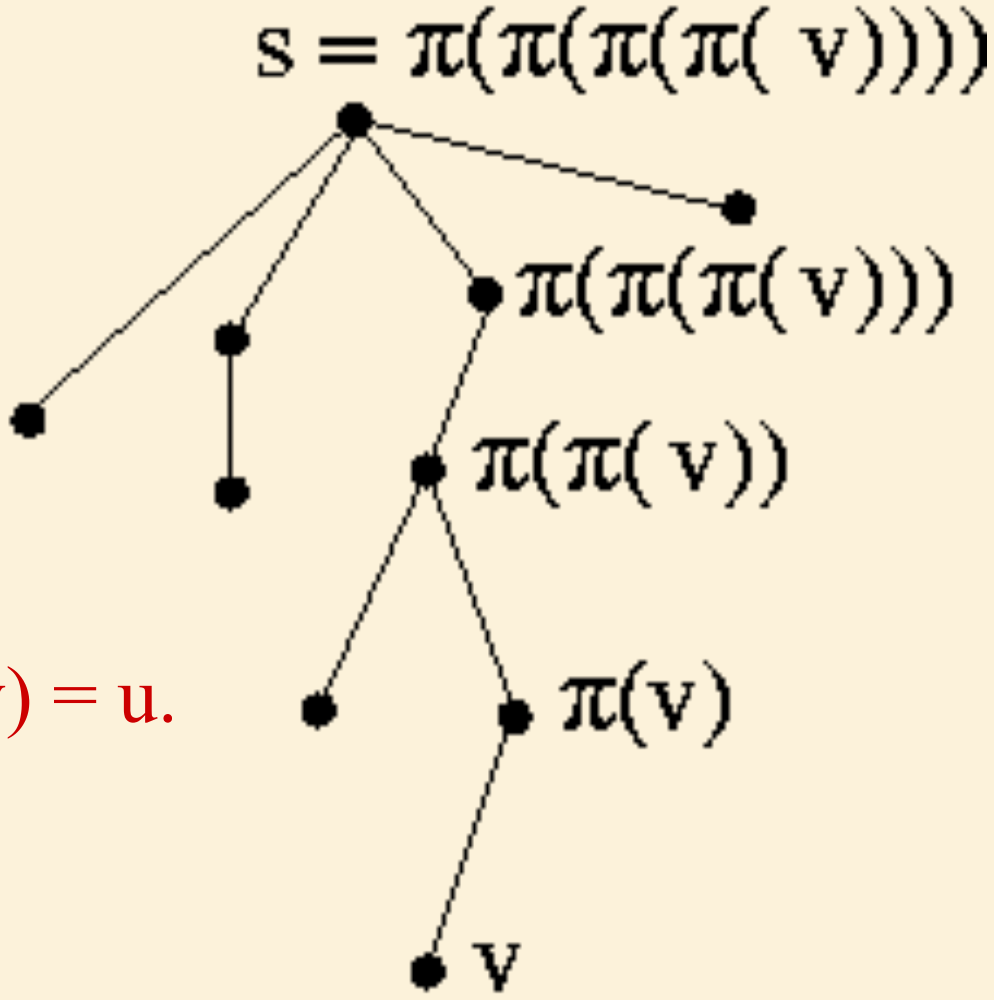


- The optimal substructure property
  - is a hallmark of both greedy and dynamic programming algorithms.
  - allows us to compute both shortest path distance and the shortest paths themselves by storing only one *d* value and one predecessor value per vertex.

Recovering the Shortest Path

For each node v, store predecessor of v in π(v).

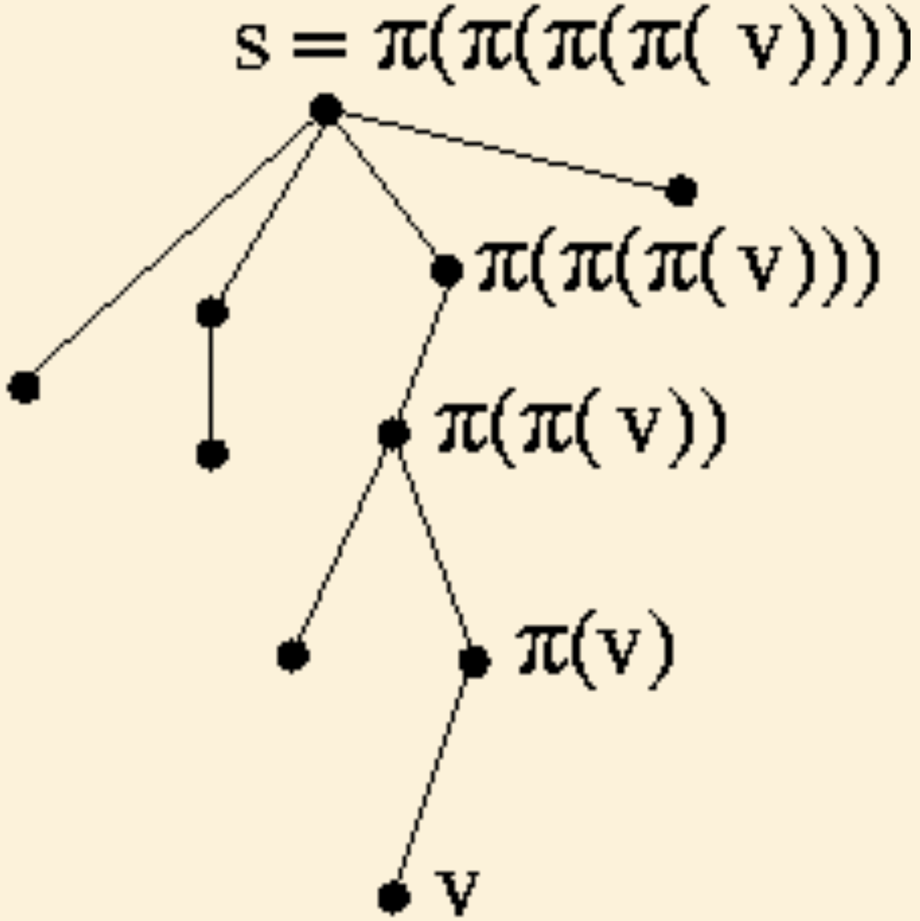




Predecessor of v is π(v) = u.

Recovering the Shortest Path

```
PRINT-PATH($G$, $s$, $v$)
Precondition:  $s$ and $v$ are vertices of graph $G$
Postcondition: the vertices on the shortest path from $s$ to $v$ have been printed in order
if $v = s$ then
    print $s$
else if $\pi[v]$ = NIL then
    print "no path from" $s$ "to" $v$ "exists"
else
    PRINT-PATH($G$, $s$, $\pi[v]$)
    print $v$
```



Colours are actually not required

```
BFS$(V, E, s)$
for each $u \in V - \{s\}$
    do $d[u] \leftarrow \infty$
$d[s] \leftarrow 0$
$Q \leftarrow \emptyset$
ENQUEUE$(Q, s)$
while $Q \neq \emptyset$
    do $u \leftarrow$ DEQUEUE$(Q)$
       for each $v \in Adj[u]$
           do if $d[v] = \infty$
                 then $d[v] \leftarrow d[u] + 1$
                      ENQUEUE$(Q, v)$
```

Depth First Search (DFS)

- Idea:
  - Continue searching “deeper” into the graph, until we get stuck.
  - If all the edges leaving *v* have been explored we “backtrack” to the vertex from which *v* was discovered.
- Does not recover shortest paths, but can be useful for extracting other properties of graph, e.g.,
  - Topological sorts
  - Detection of cycles
  - Extraction of strongly connected components

Depth-First Search

Input: Graph $G = (V, E)$ (directed or undirected)

Output: 2 timestamps on each vertex:

$d[v]$ = discovery time.

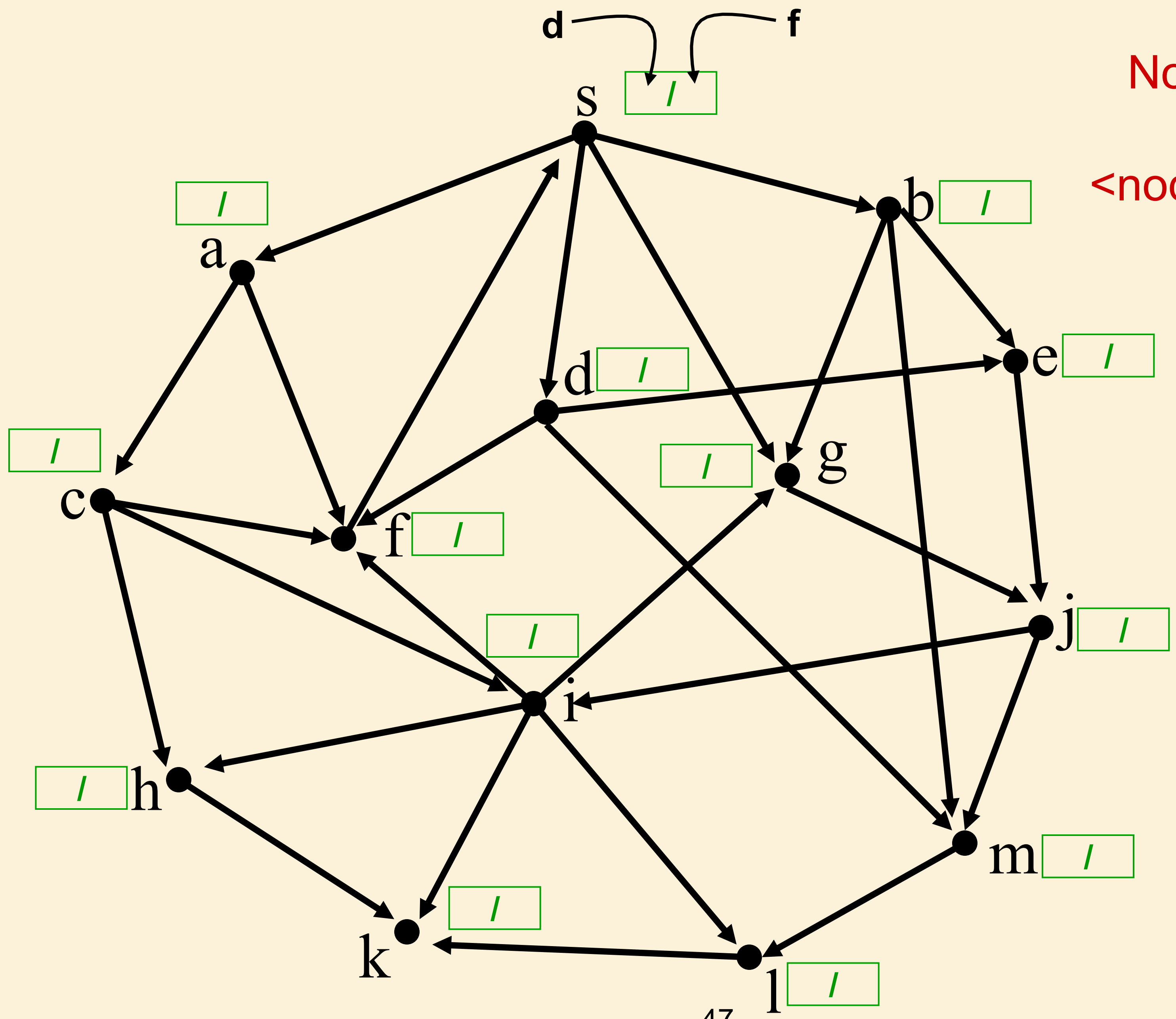
$f[v]$ = finishing time.

$$1 \le d[v] < f[v] \le 2|V|$$

- Explore *every* edge, starting from different vertices if necessary.
- As soon as vertex discovered, explore from it.
- Keep track of progress by colouring vertices:
  - Black: undiscovered vertices
  - Red: discovered, but not finished (still exploring from it)
  - Gray: finished (found everything reachable from it).

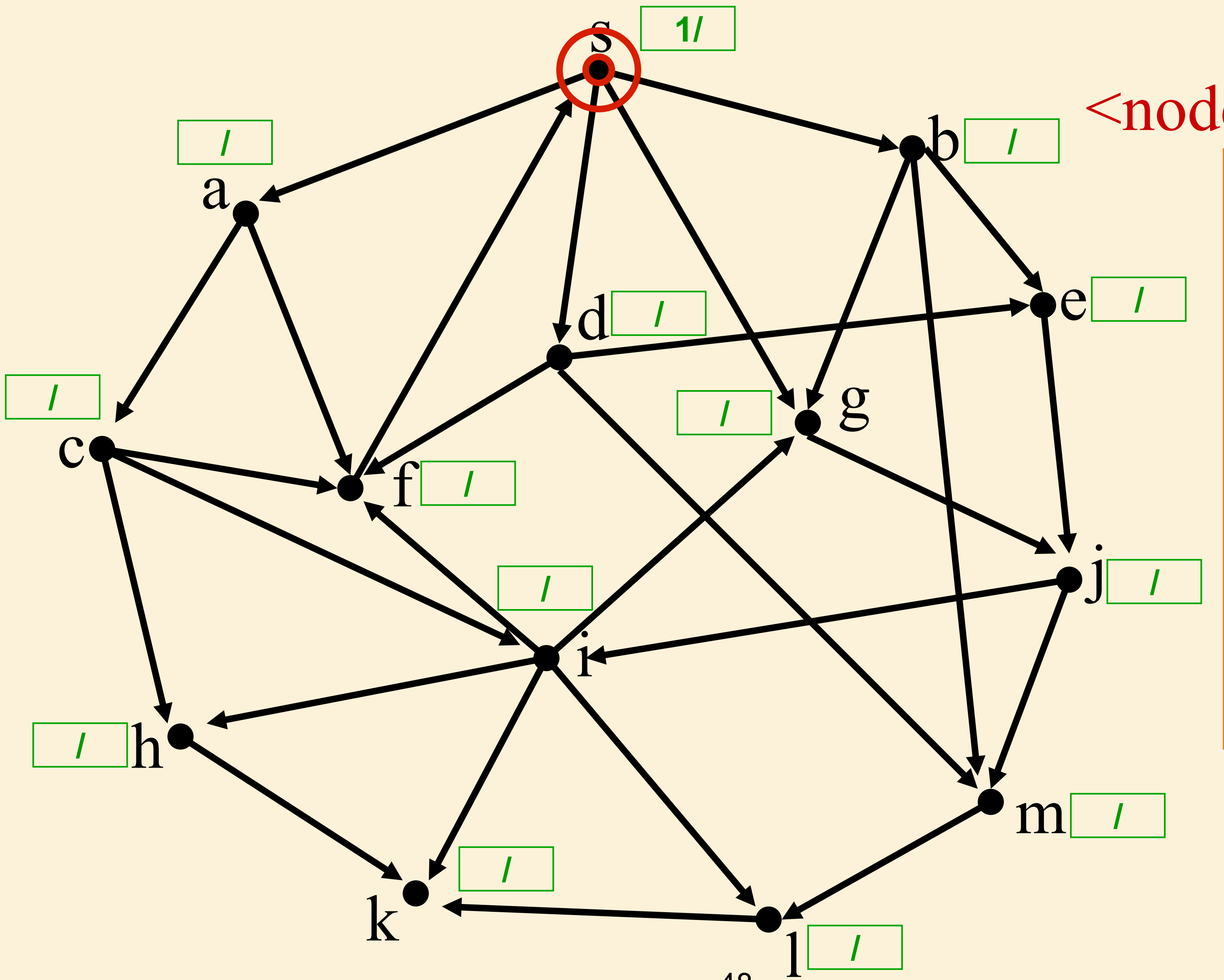
DFS

Note: Stack is Last-In First-Out (LIFO)

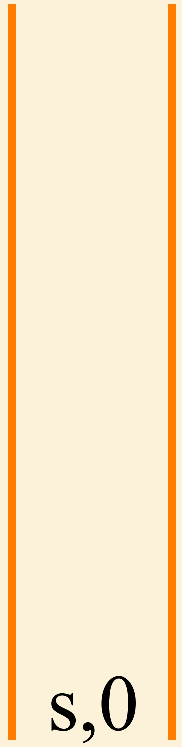


Found
Not Handled
Stack
<node,# edges>

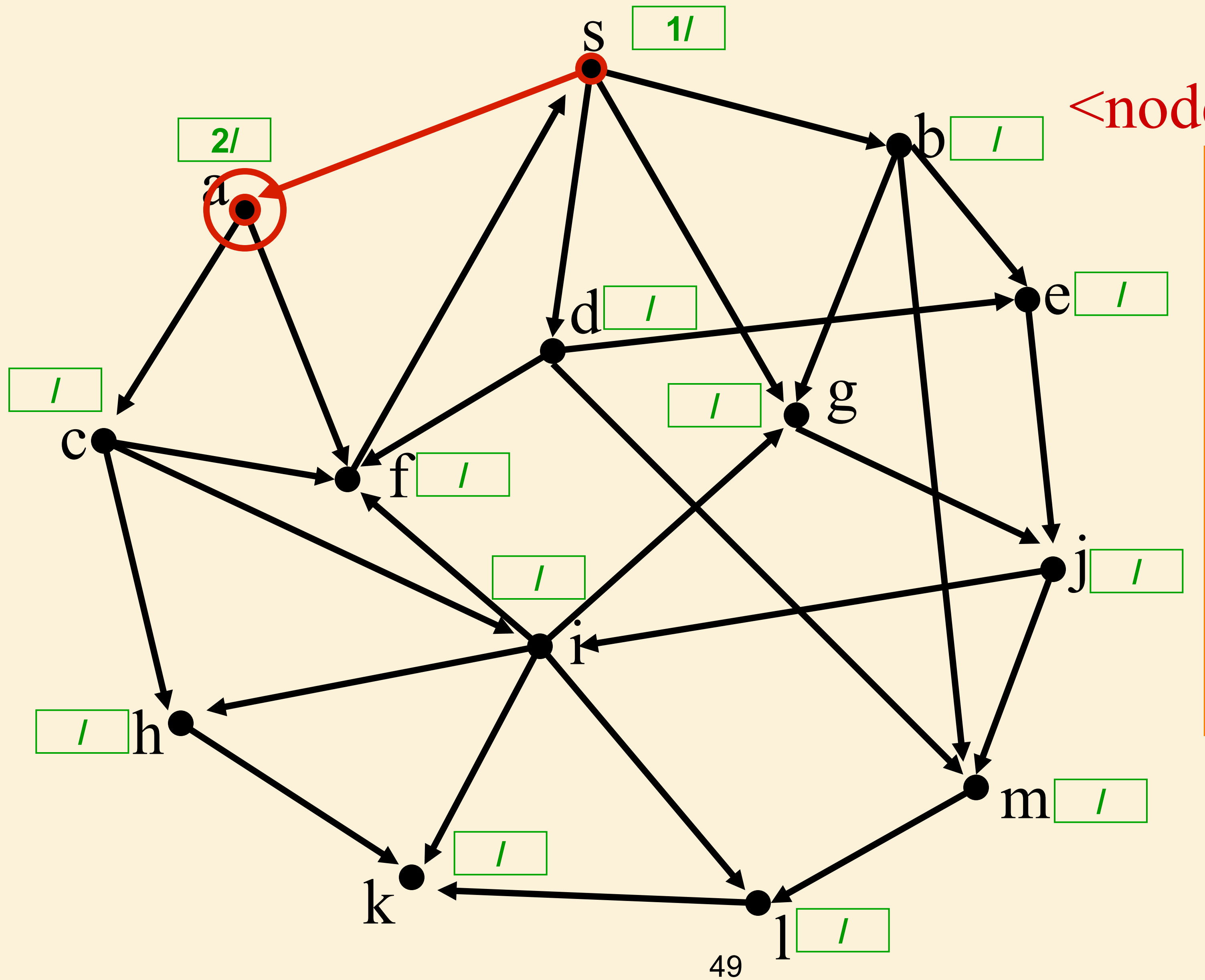
DFS



Found
Not Handled
Stack
<node,# edges>



DFS



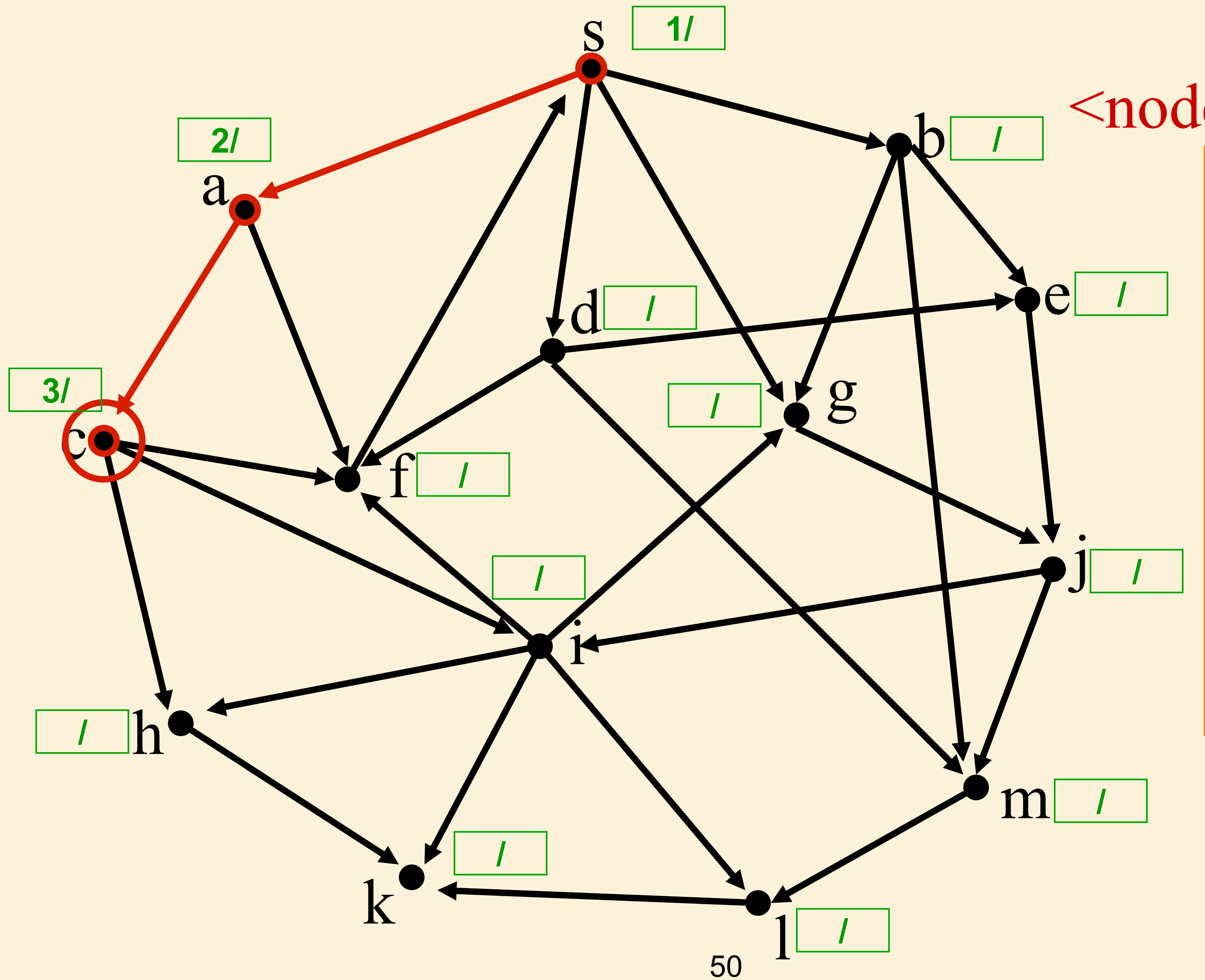
Found
Not Handled
Stack
<node,# edges>

a,0
s,1

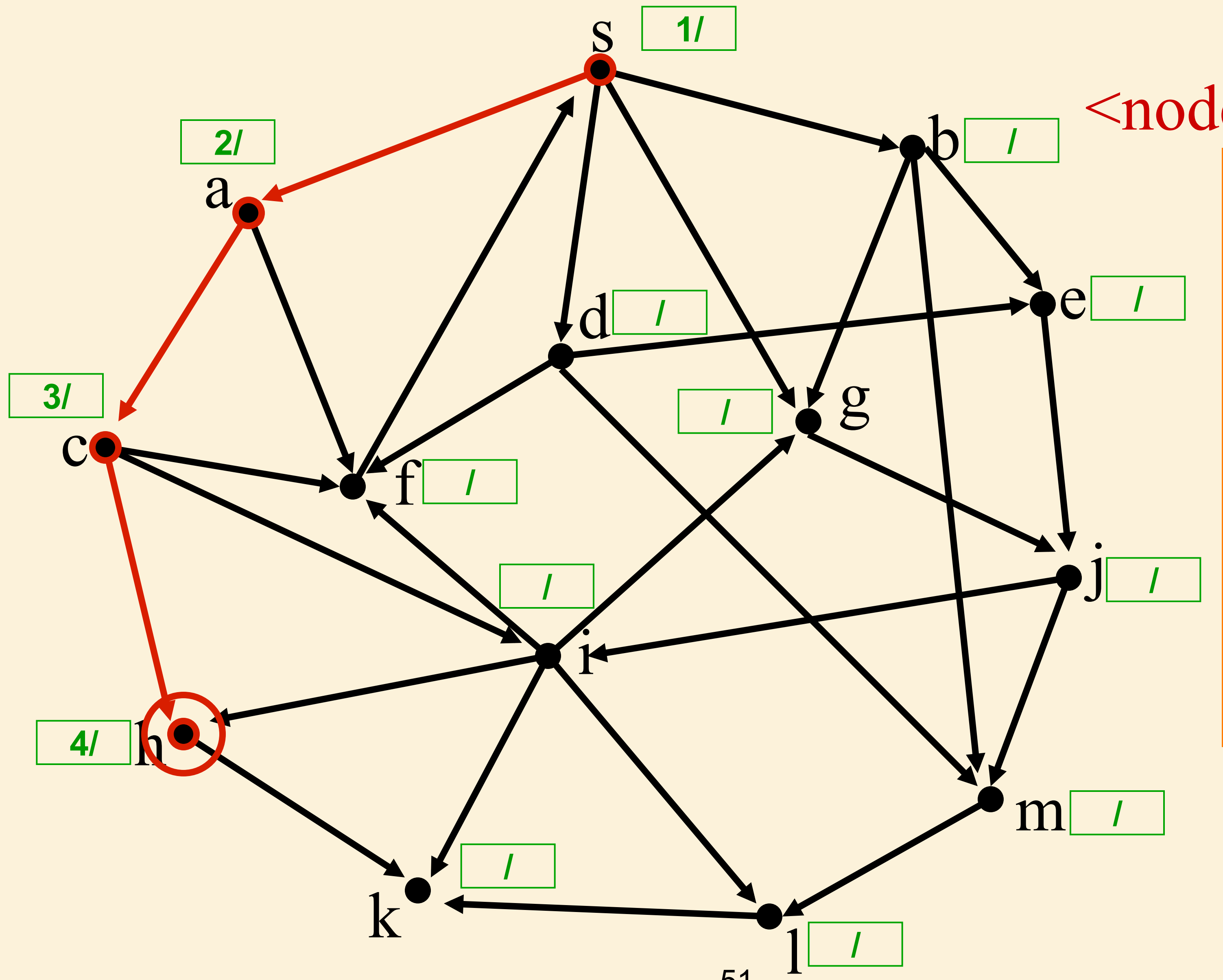
DFS

Found
Not Handled
Stack
<node,# edges>

c,0
a,1
s,1



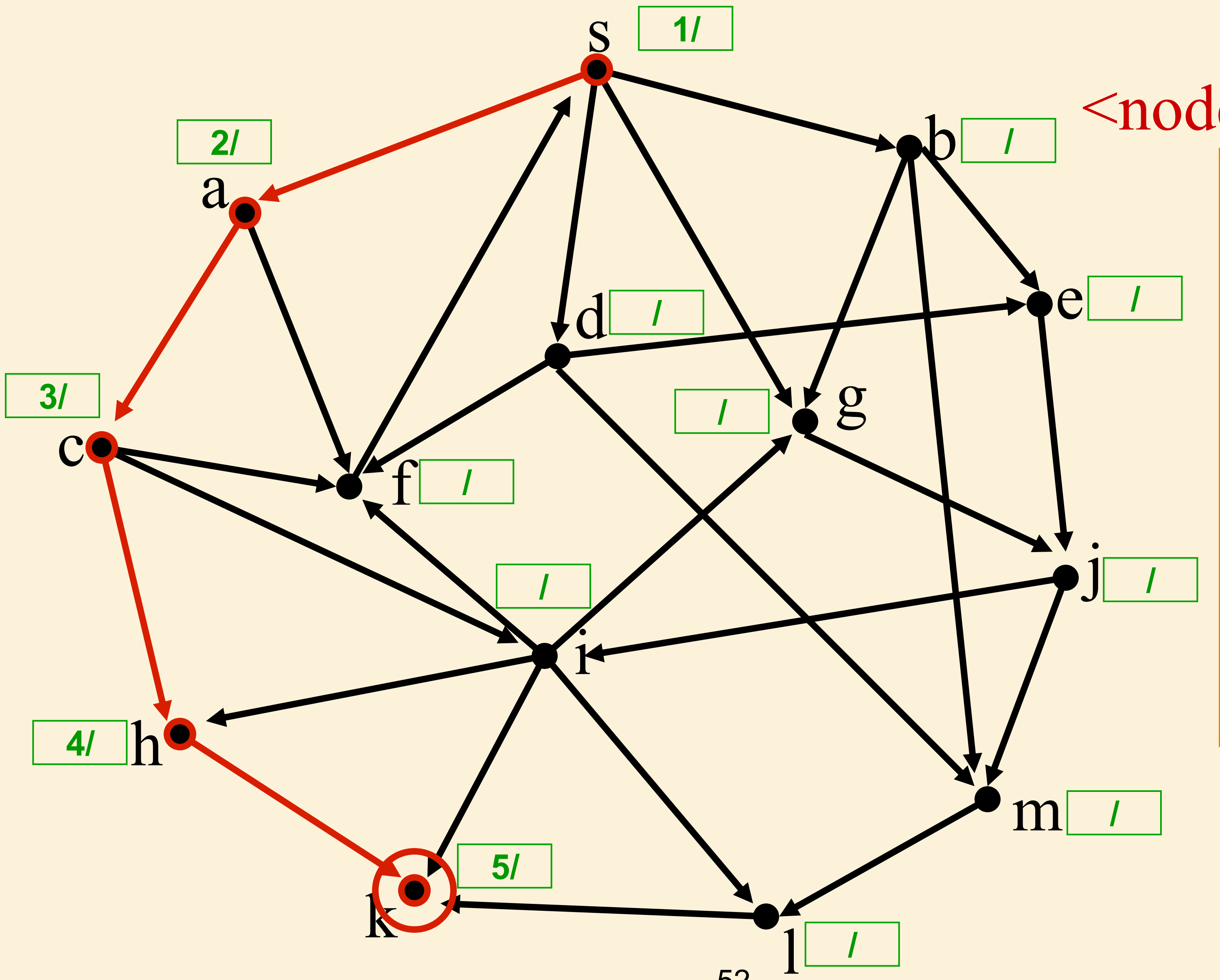
DFS



Found
Not Handled
Stack
<node,# edges>

h,0
c,1
a,1
s,1

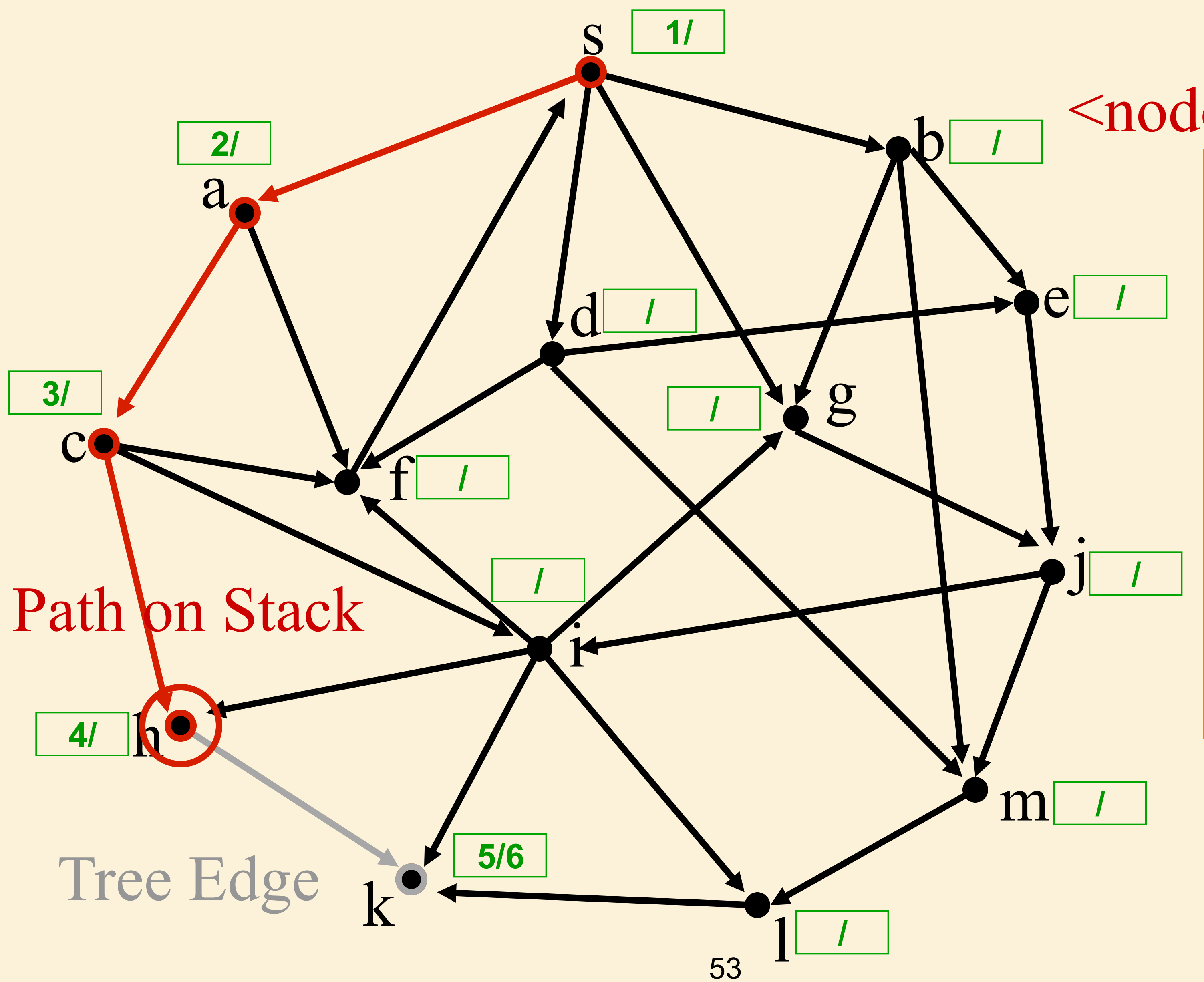
DFS



Found
Not Handled
Stack
<node,# edges>

k,0
h,1
c,1
a,1
s,1

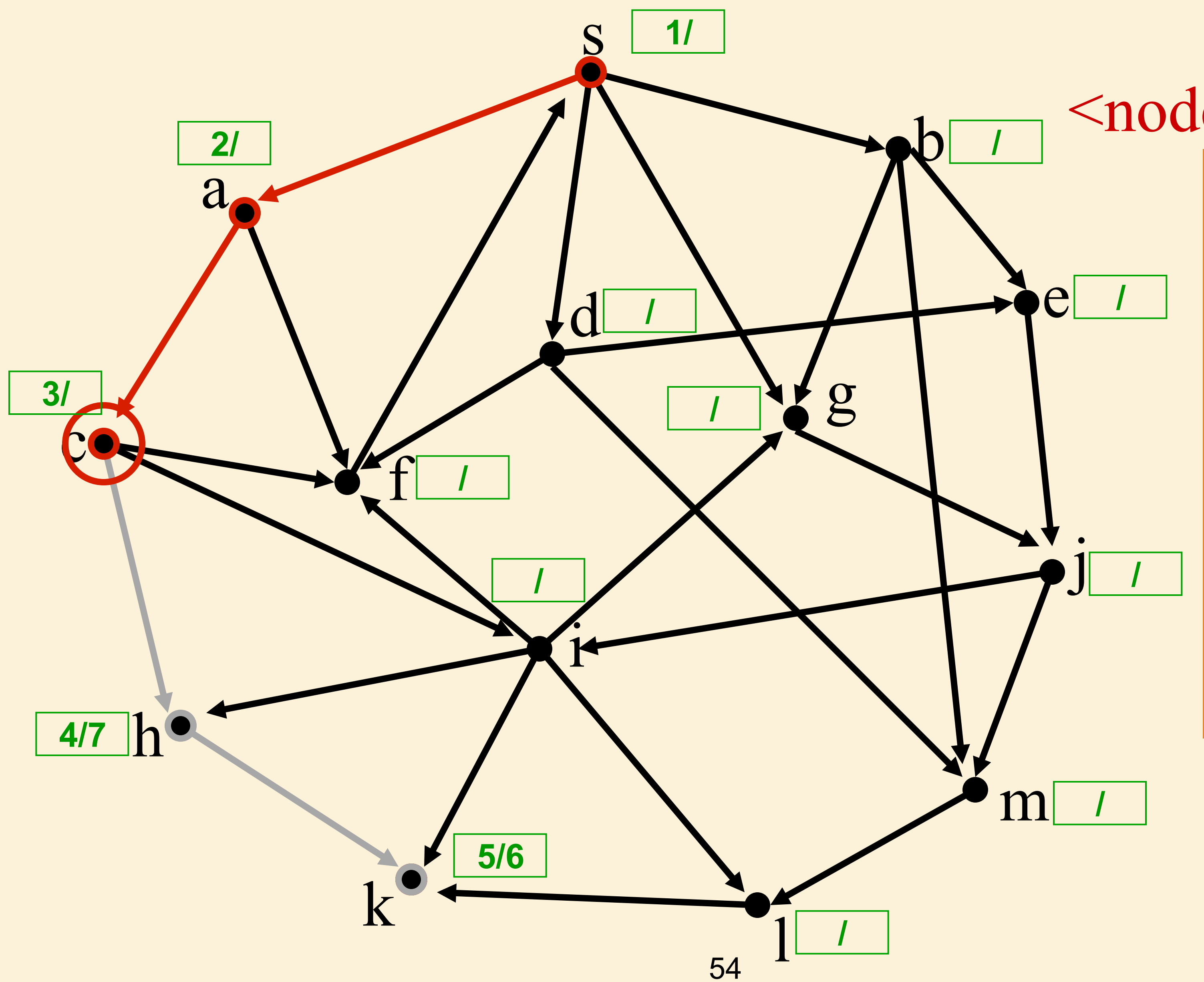
DFS



Found
Not Handled
Stack
<node,# edges>

h,1
c,1
a,1
s,1

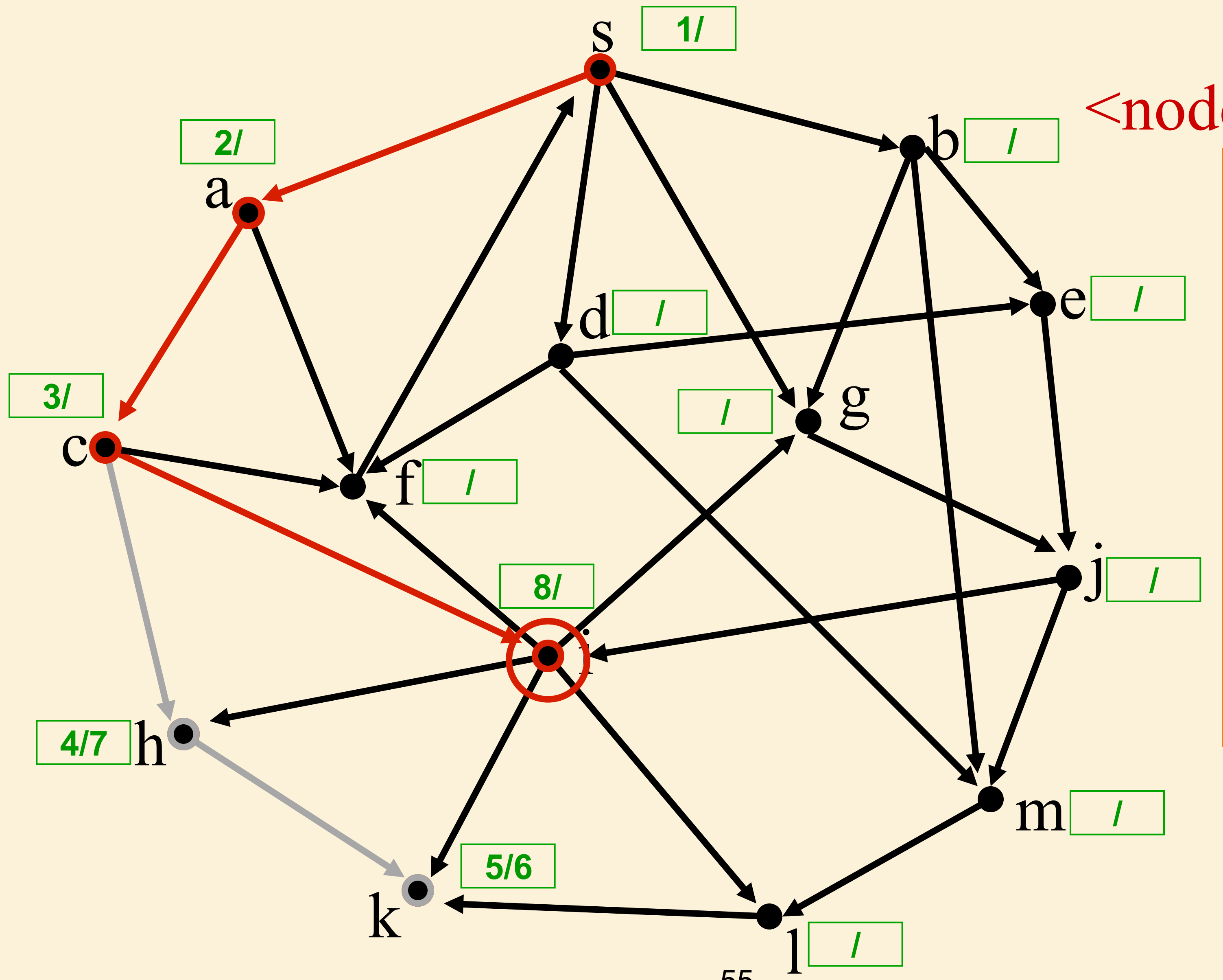
DFS



Found
Not Handled
Stack
<node,# edges>

c, 1
a, 1
s, 1

DFS

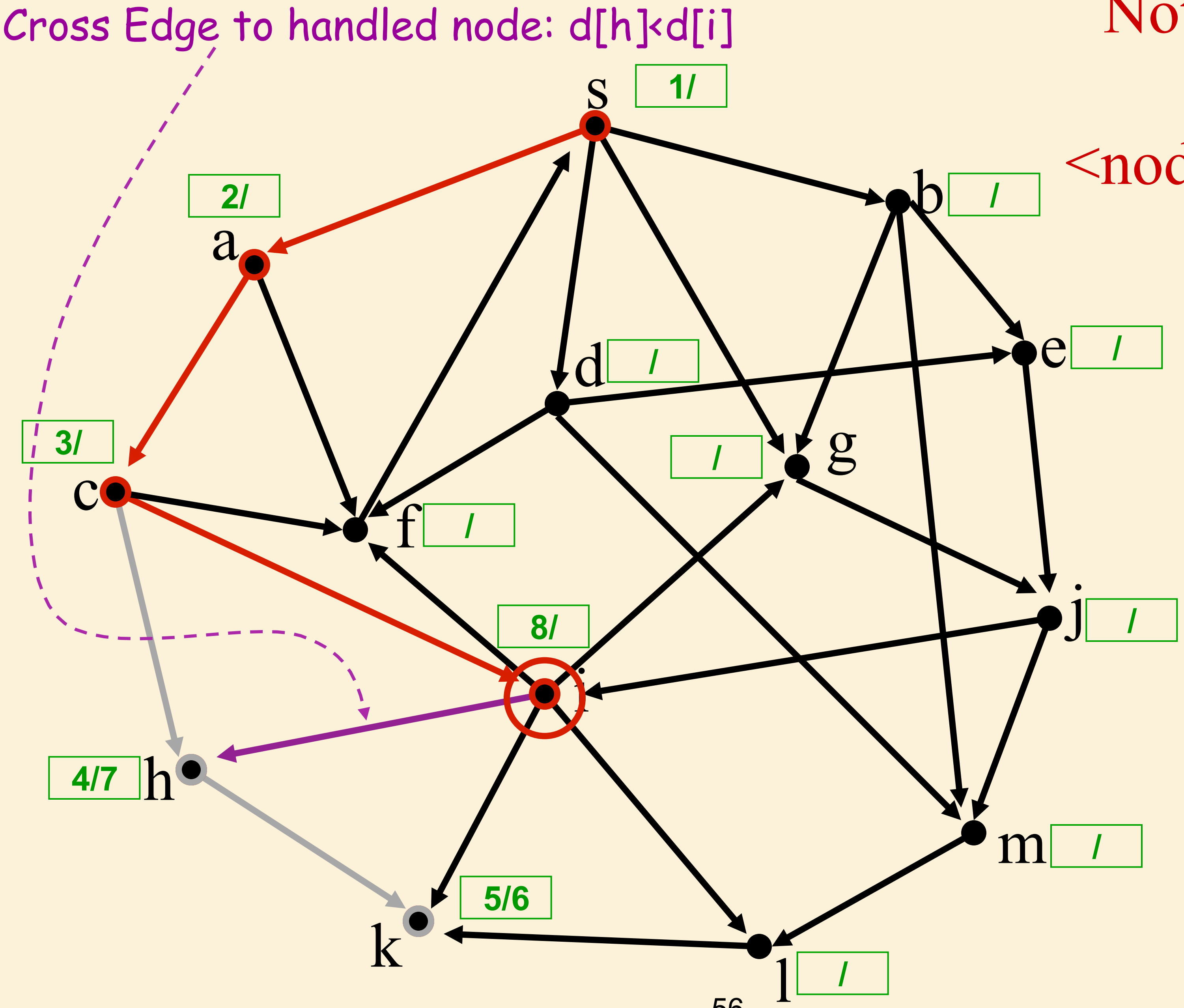


Found
Not Handled
Stack
<node,# edges>

i,0
c,2
a,1
s,1

DFS

Found
Not Handled
Stack
<node,# edges>

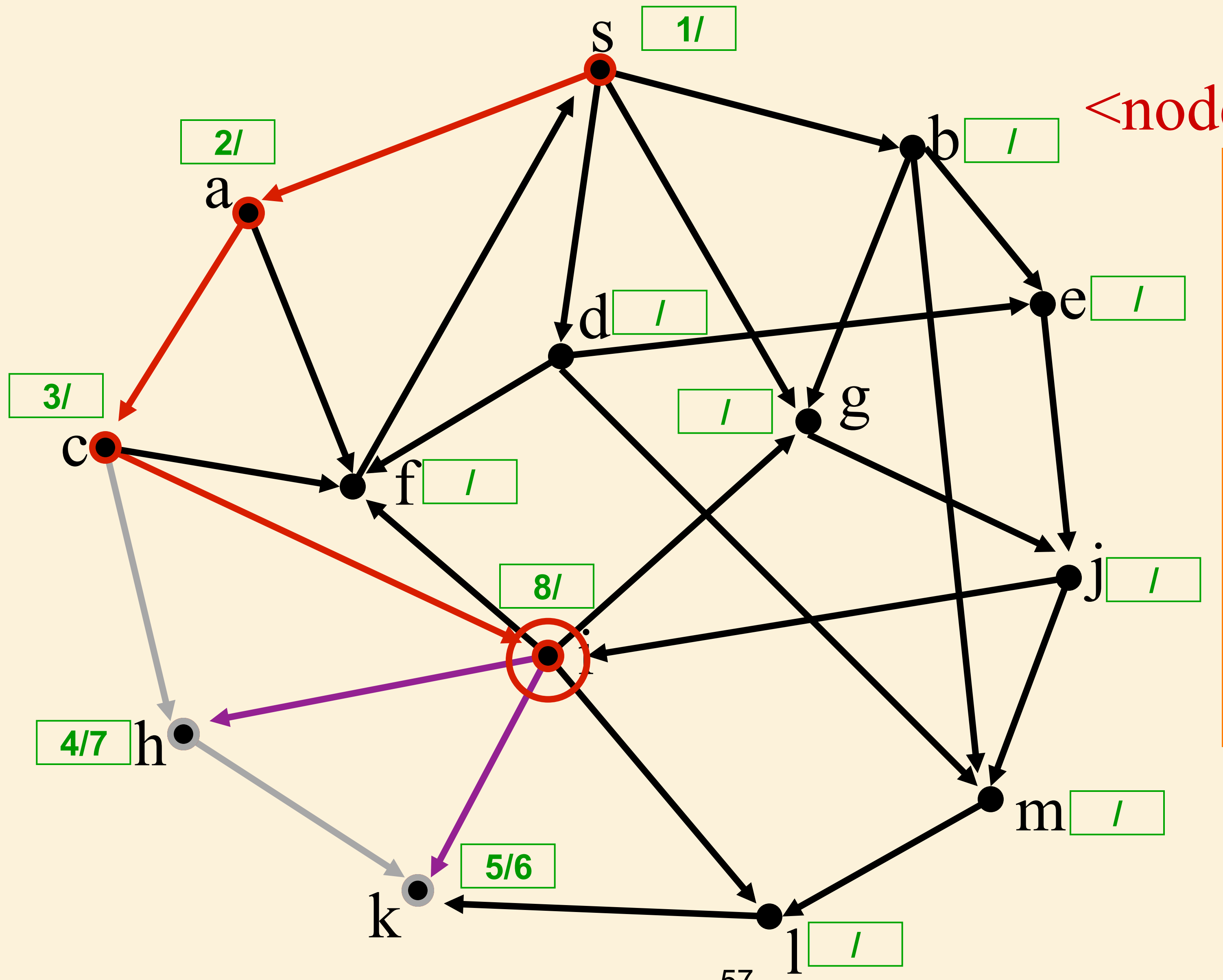


i,1
c,2
a,1
s,1

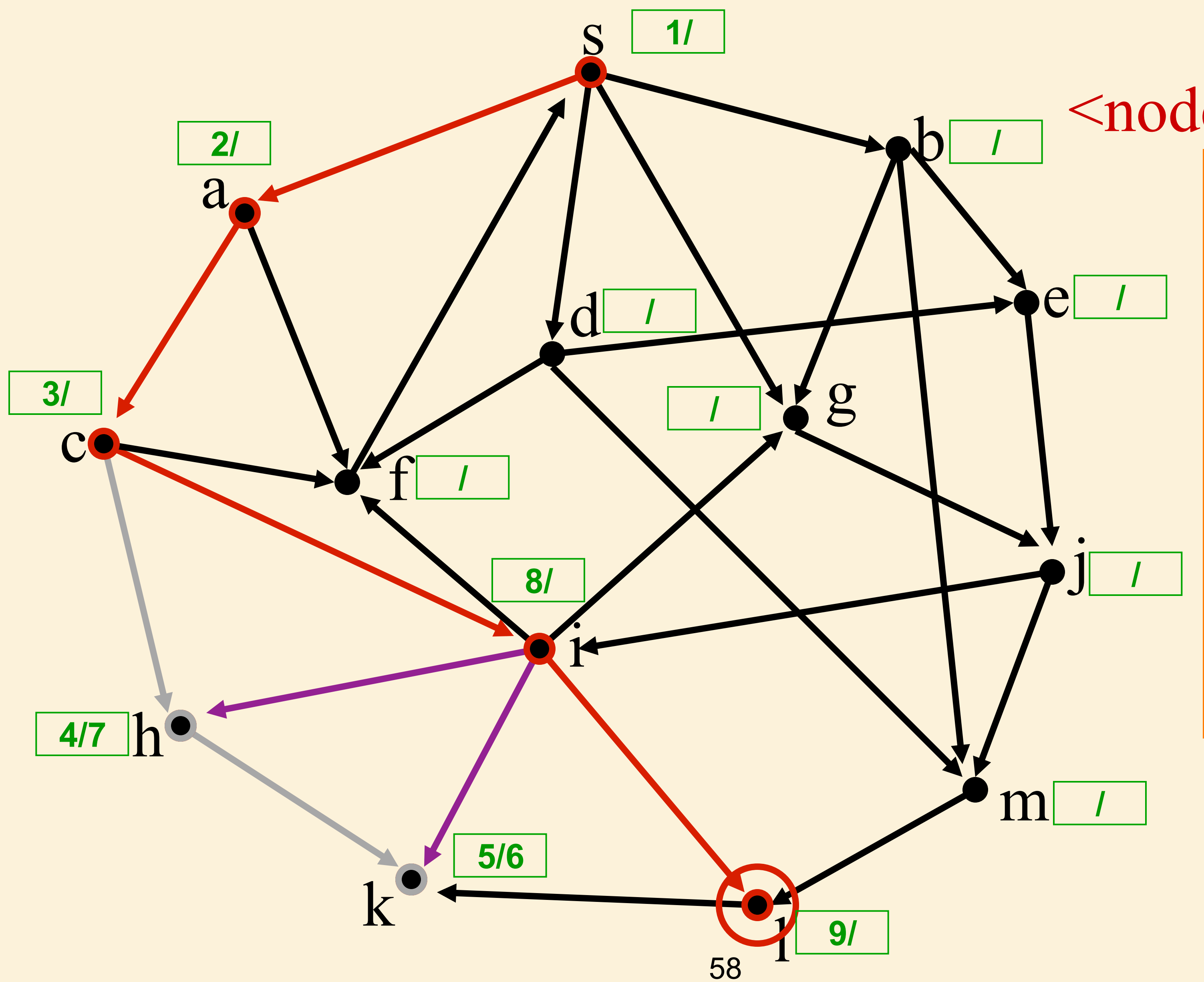
DFS

Found
Not Handled
Stack
<node,# edges>

i,2
c,2
a,1
s,1



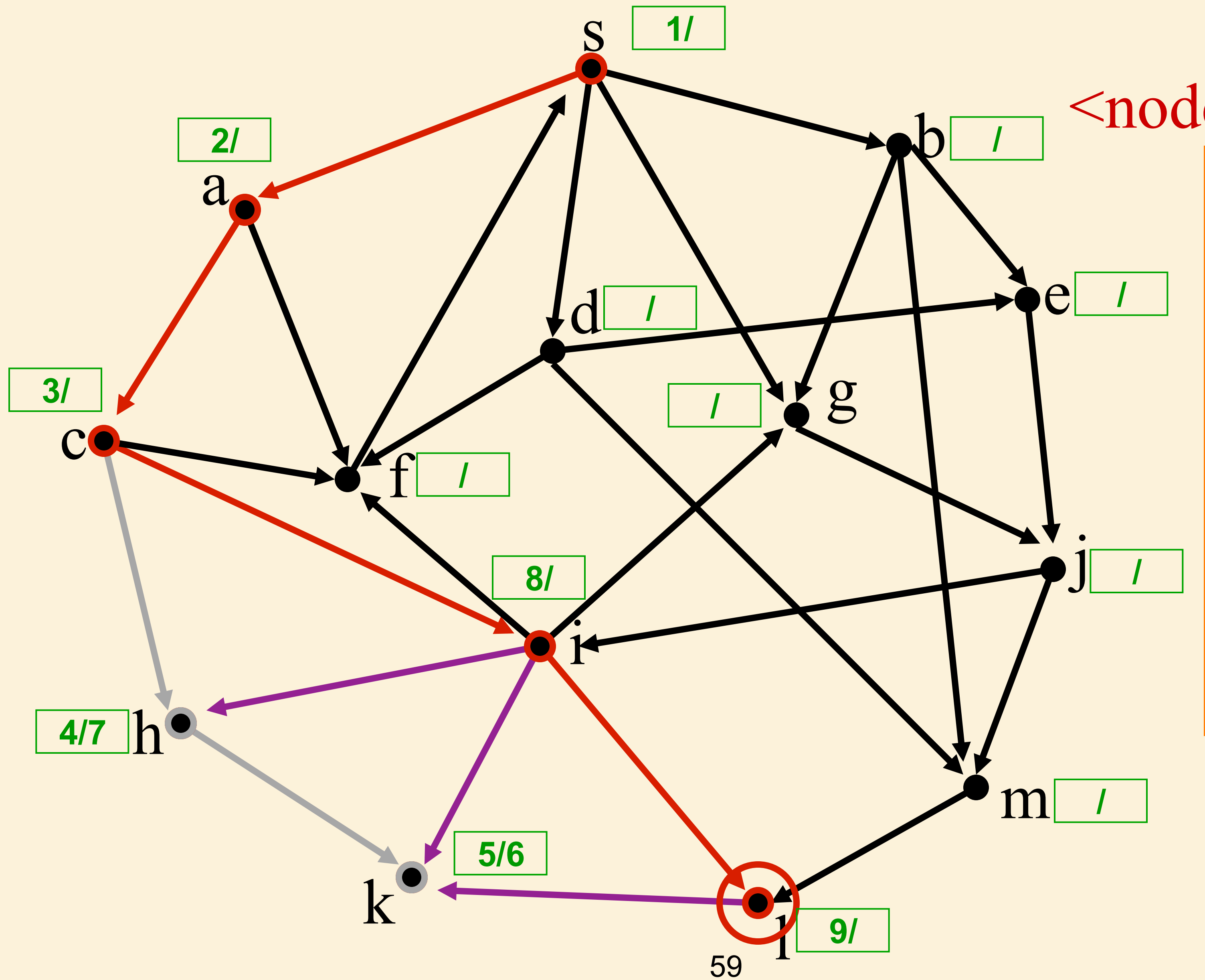
DFS



Found
Not Handled
Stack
<node,# edges>

l,0
i,3
c,2
a,1
s,1

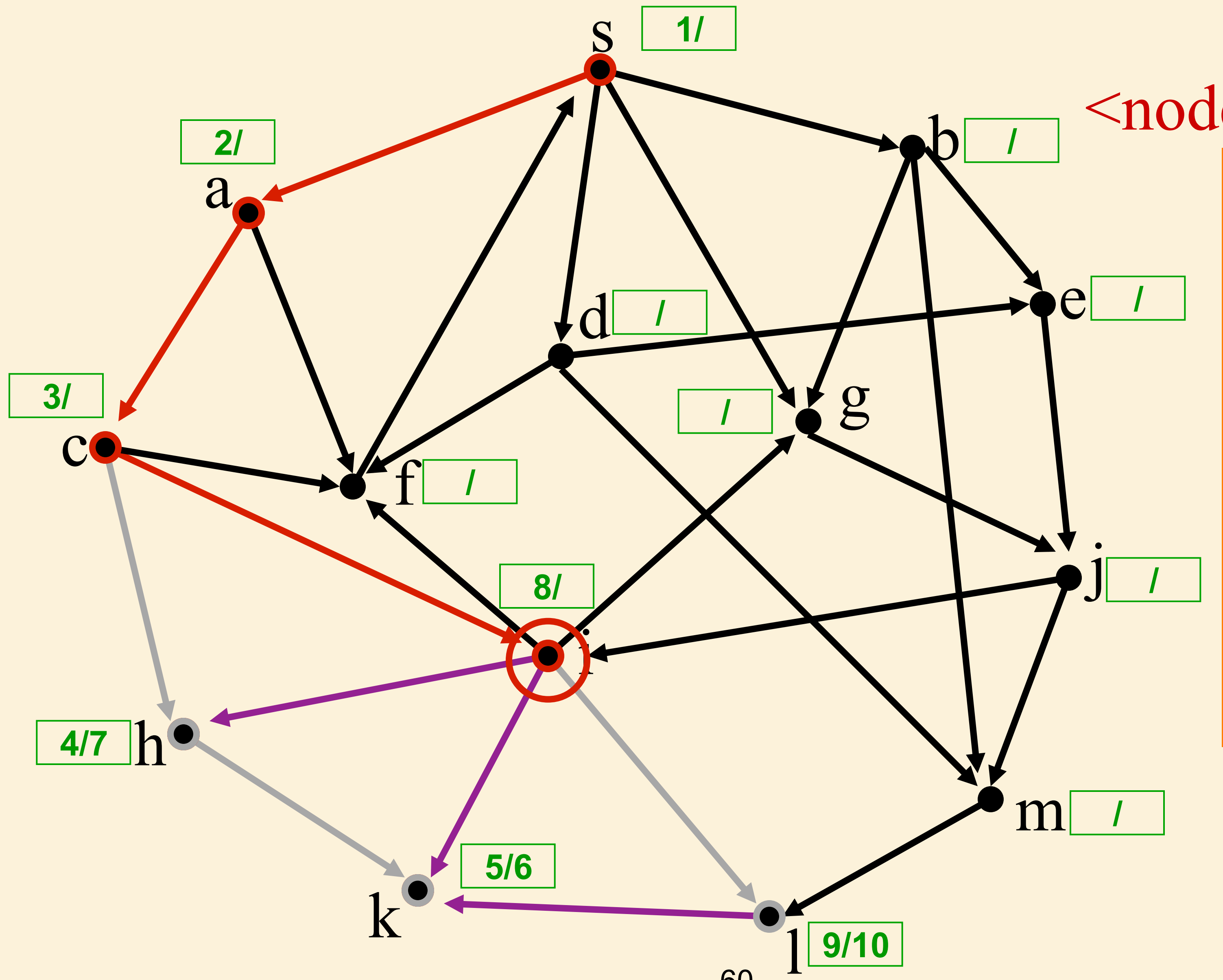
DFS



Found
Not Handled
Stack
<node,# edges>

l,1
i,3
c,2
a,1
s,1

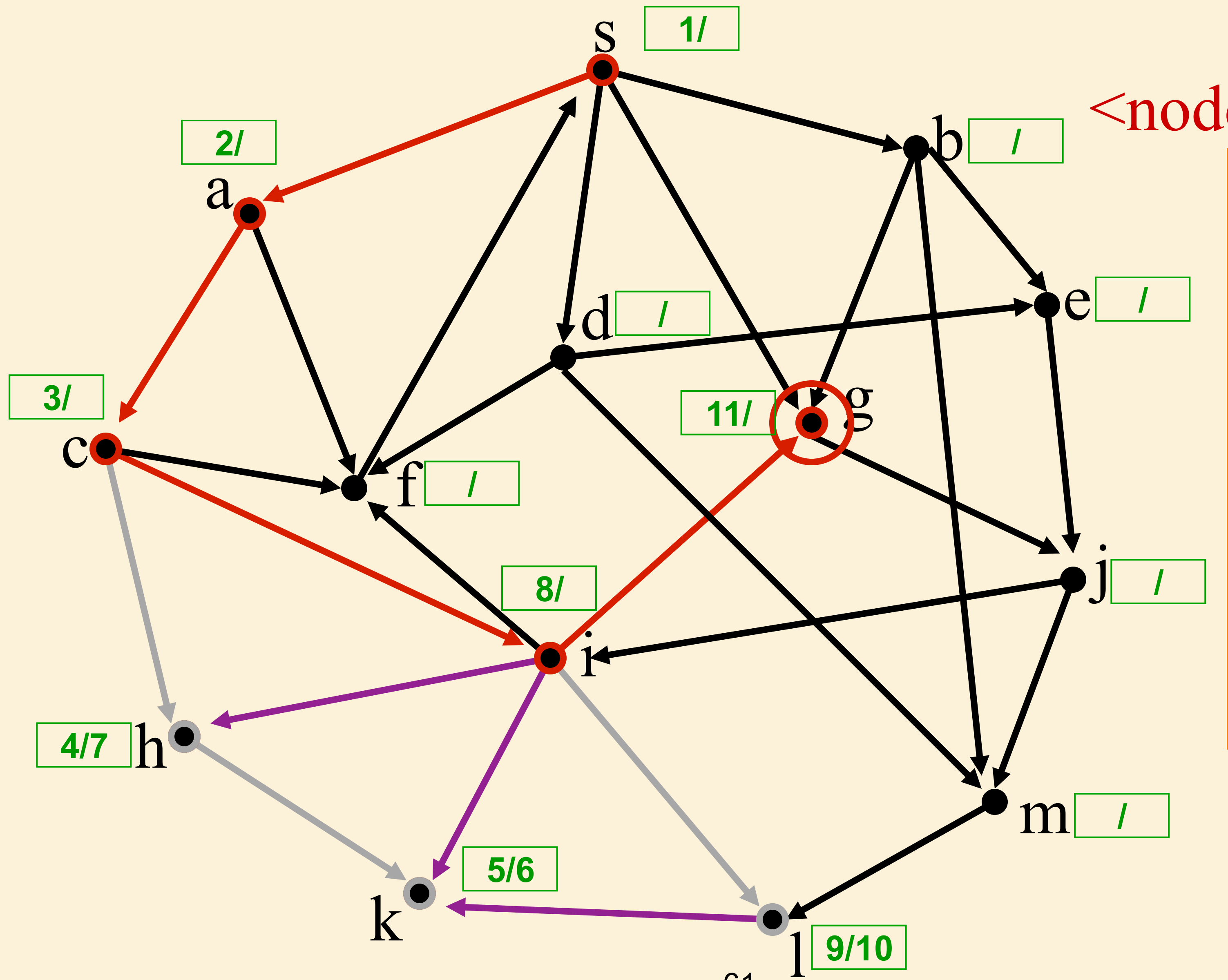
DFS



Found
Not Handled
Stack
<node,# edges>

i,3
c,2
a,1
s,1

DFS

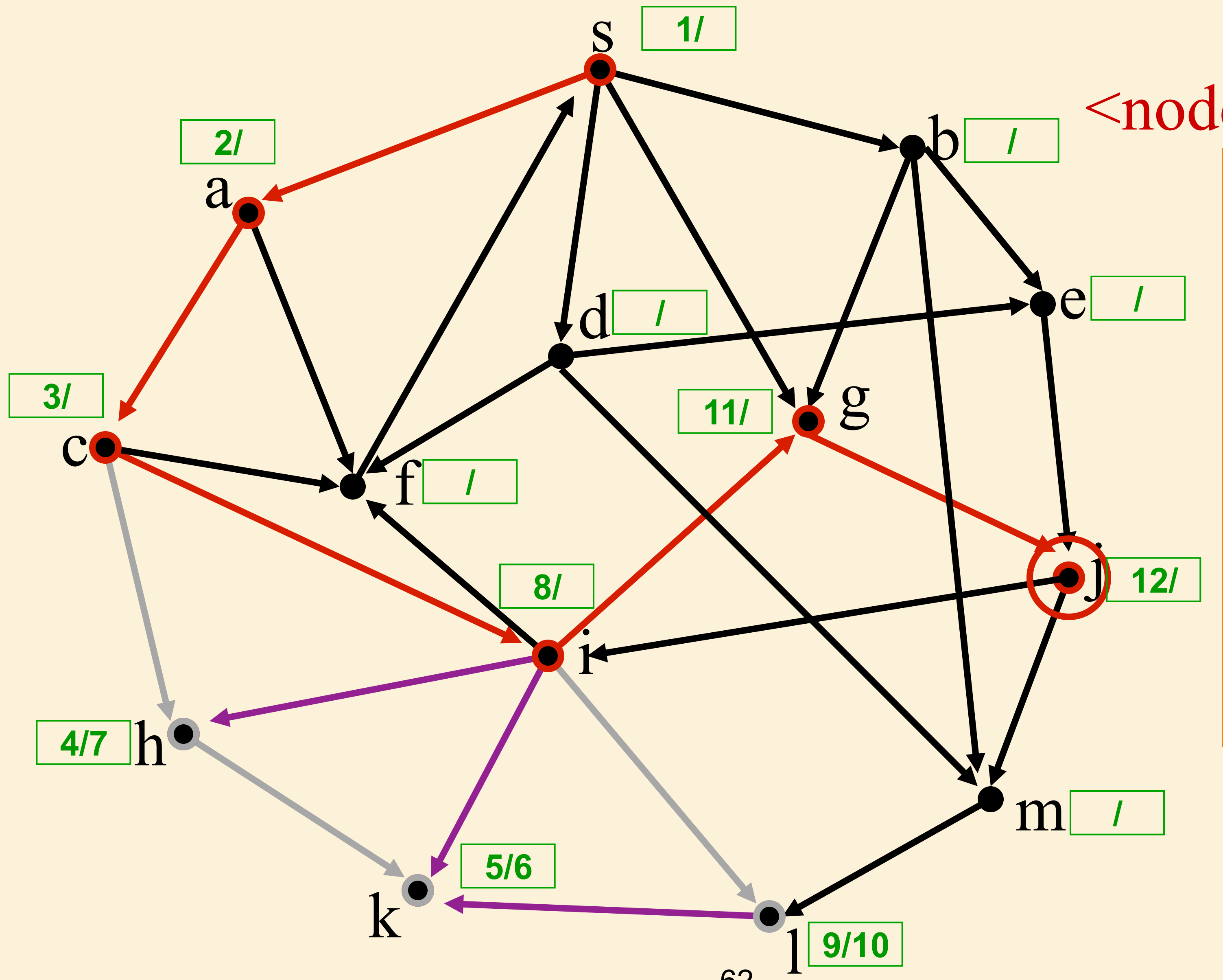


Found
Not Handled
Stack
<node,# edges>

g,0
i,4
c,2
a,1
s,1

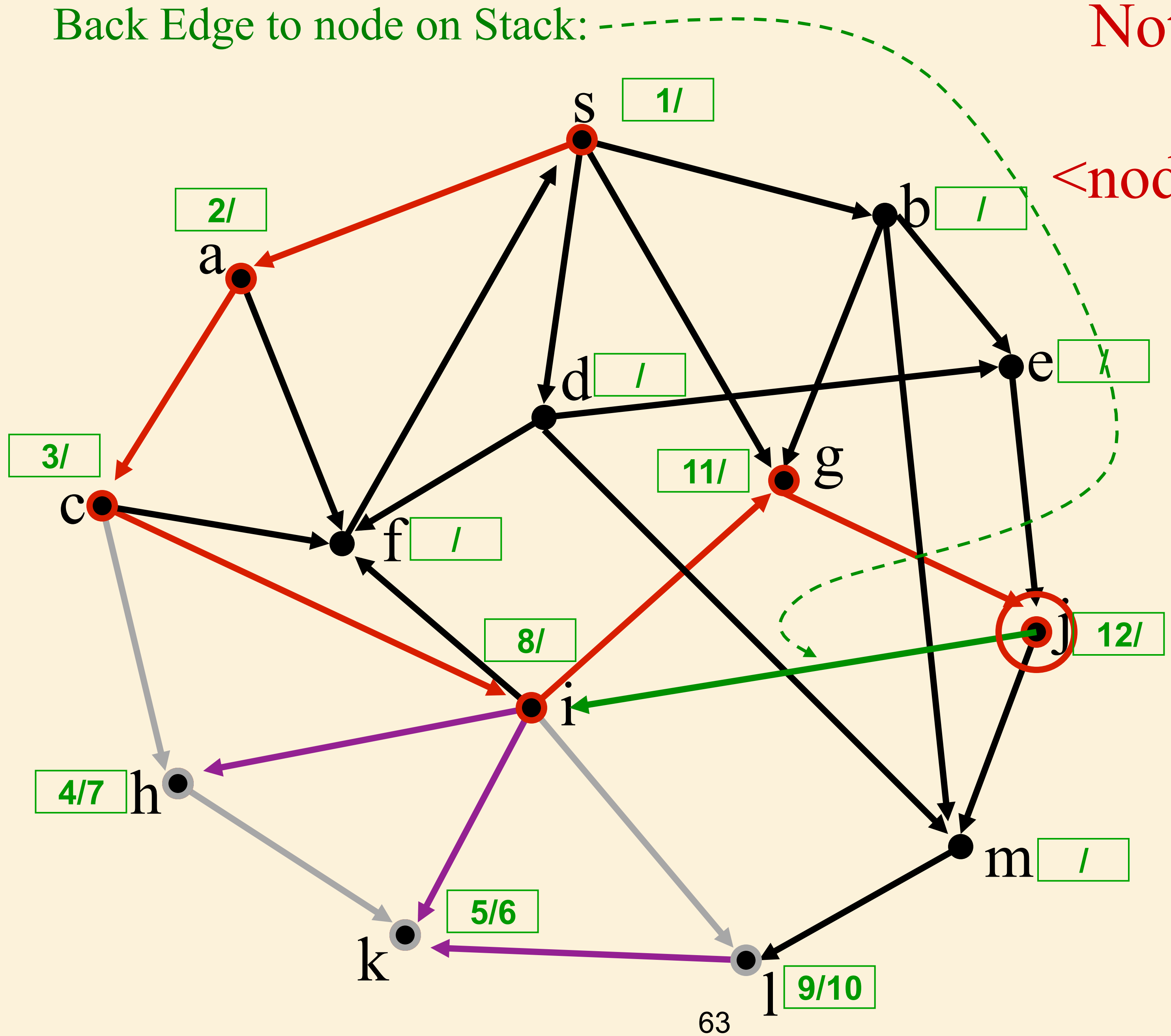
DFS

Found
Not Handled
Stack
<node,# edges>



j,0
g,1
i,4
c,2
a,1
s,1

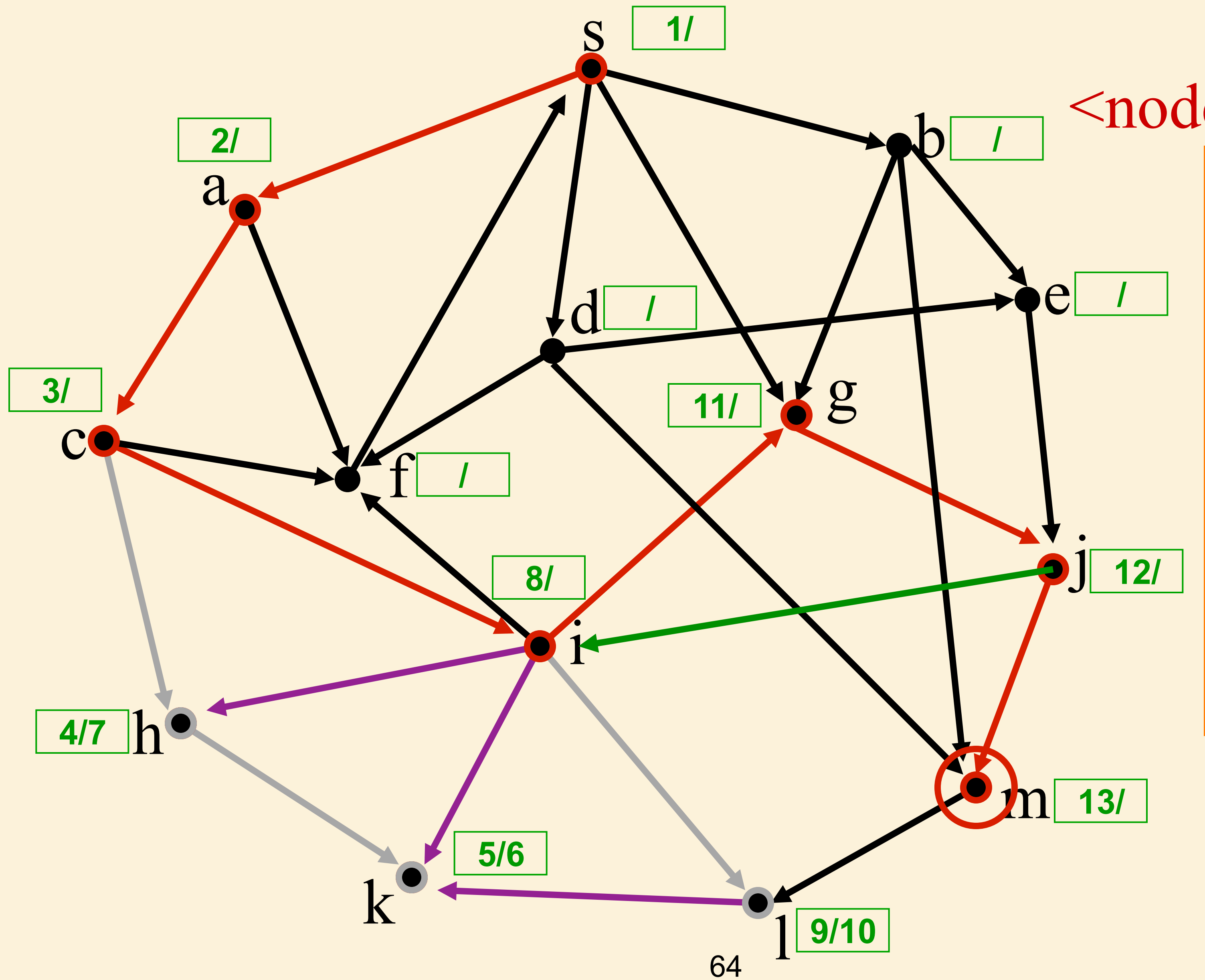
DFS



Found
Not Handled
Stack
<node,# edges>

j,1
g,1
i,4
c,2
a,1
s,1

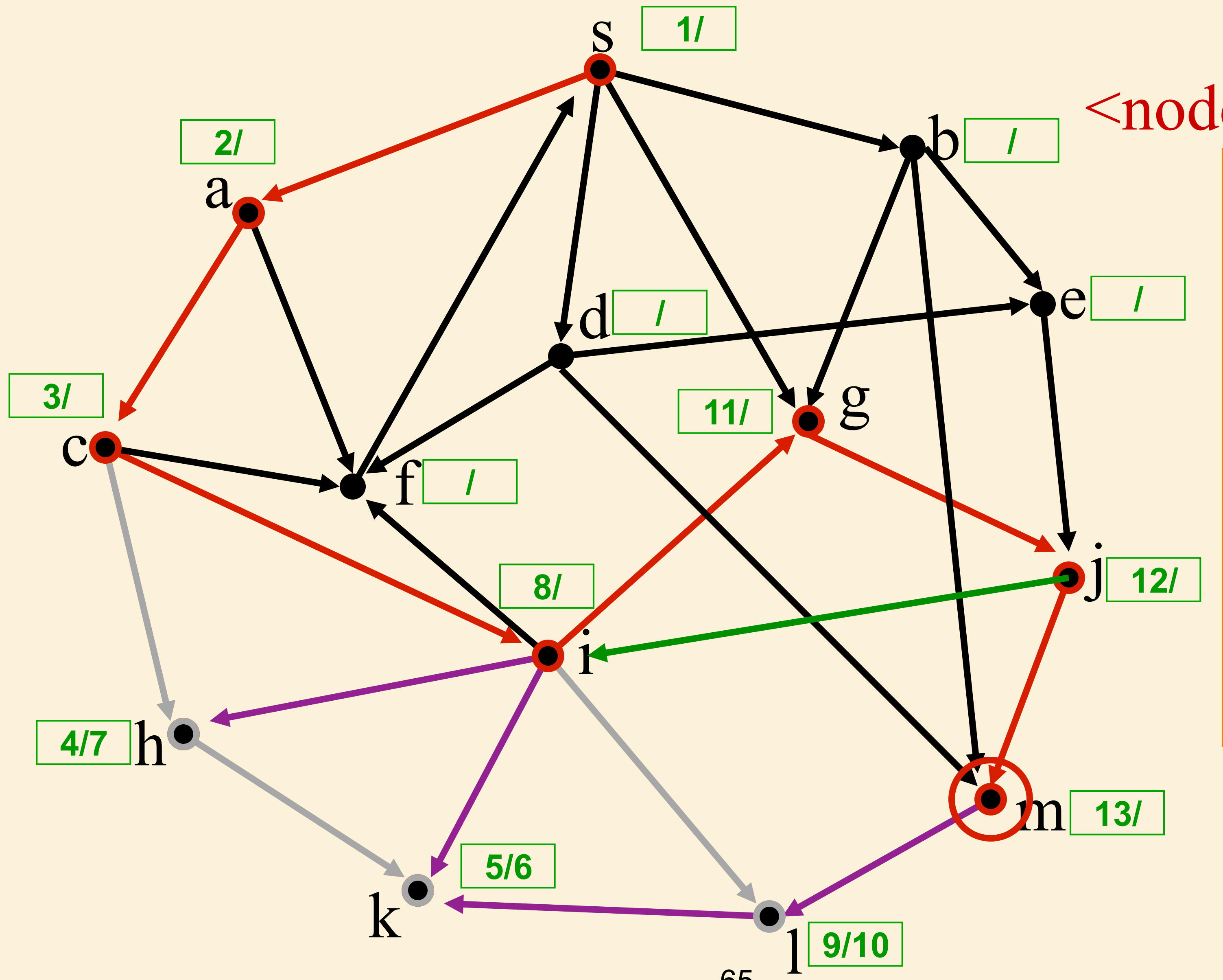
DFS



Found
Not Handled
Stack
<node,# edges>

m,0
j,2
g,1
i,4
c,2
a,1
s,1

DFS



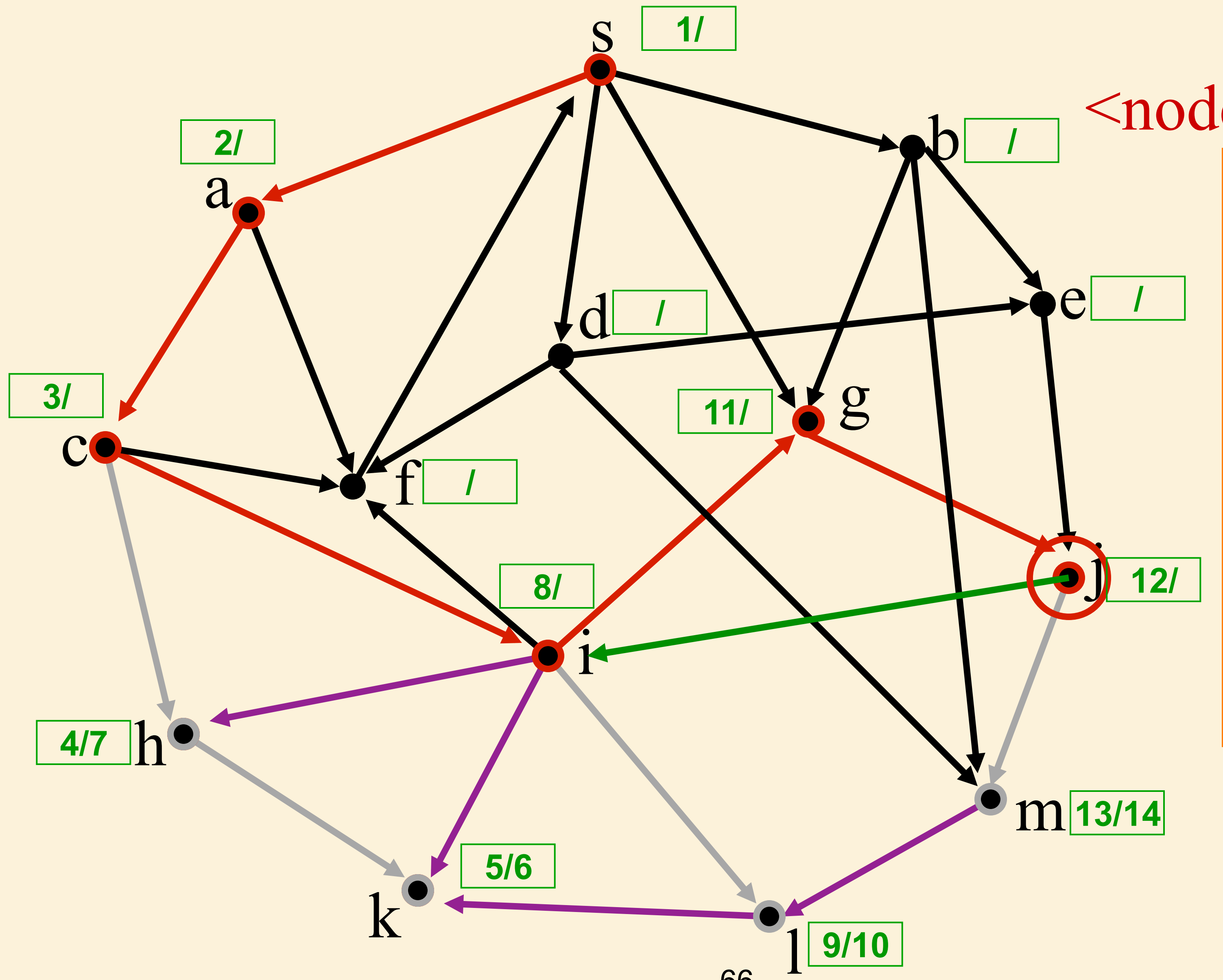
Found
Not Handled
Stack
<node,# edges>

m,1
j,2
g,1
i,4
c,2
a,1
s,1

DFS

Found
Not Handled
Stack
<node,# edges>

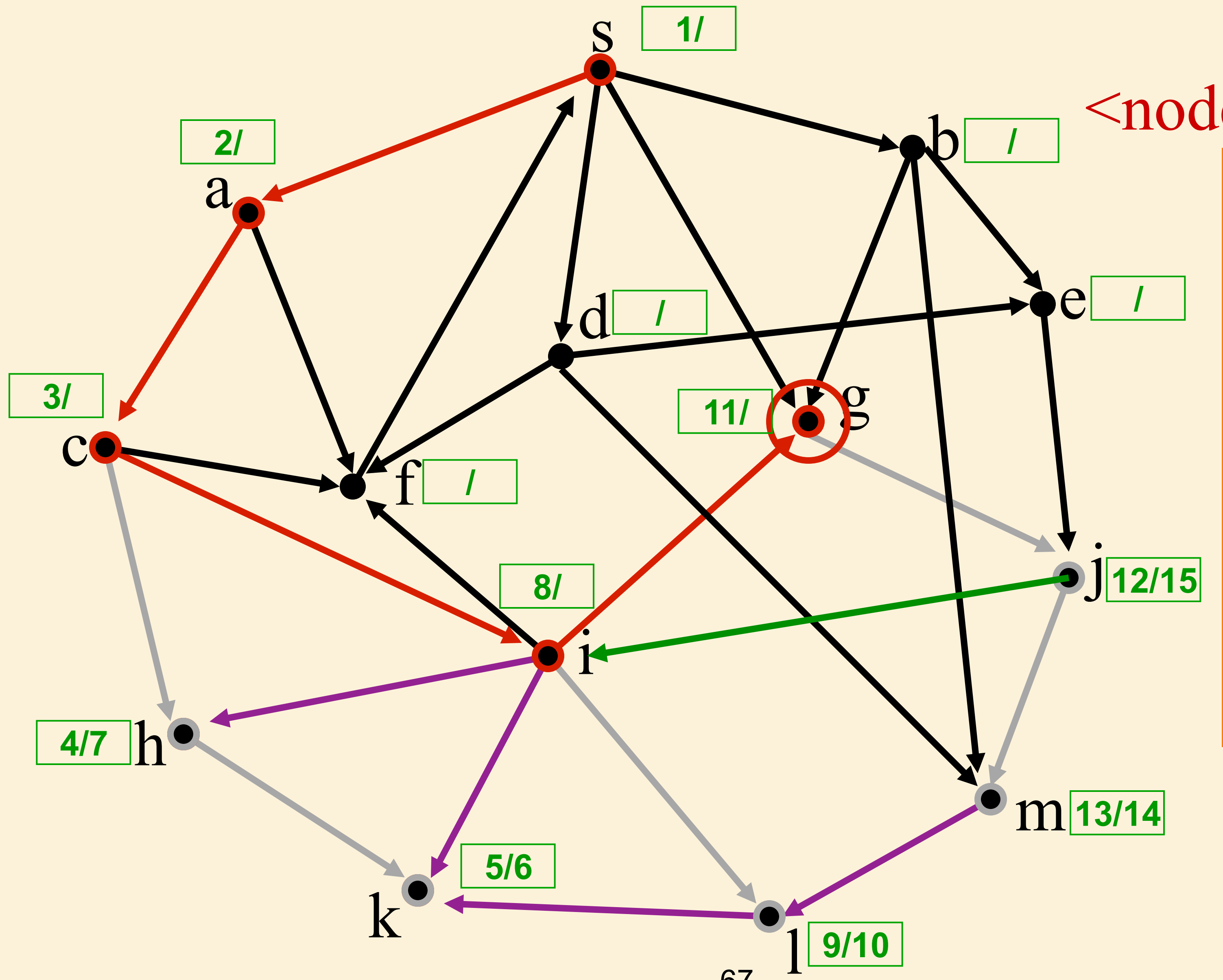
j,2
g,1
i,4
c,2
a,1
s,1



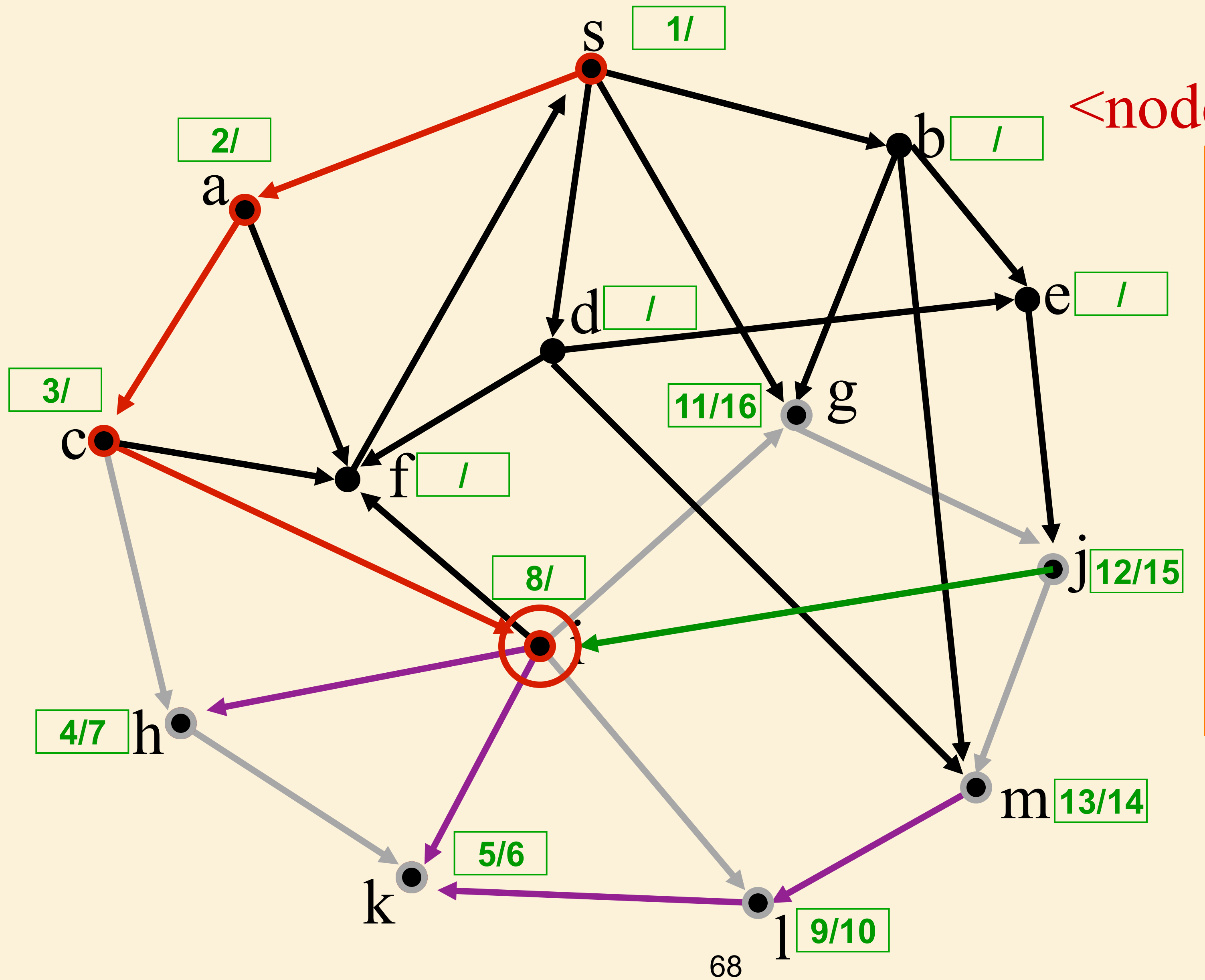
DFS

Found
Not Handled
Stack
<node,# edges>

g,1
i,4
c,2
a,1
s,1



DFS

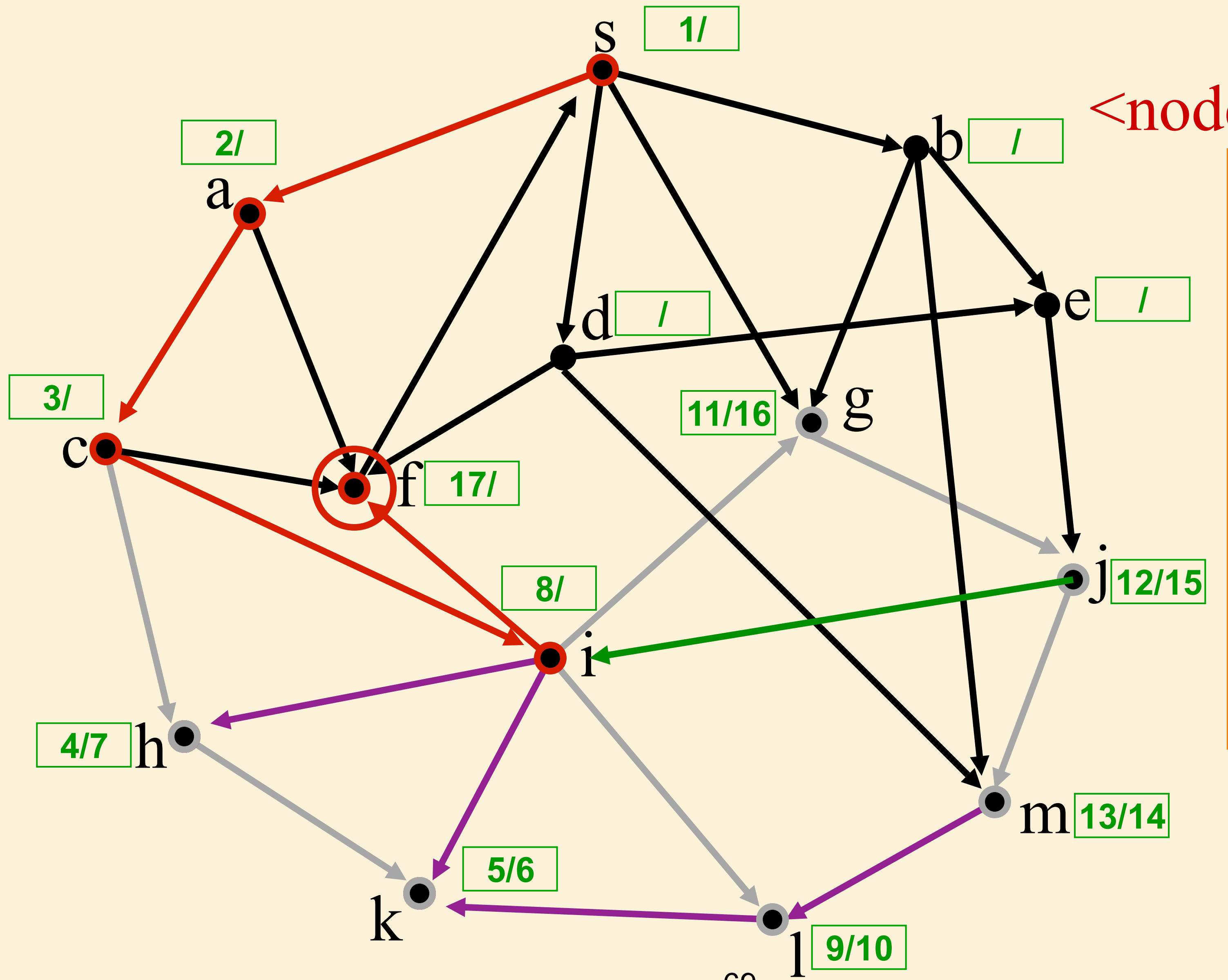


Found
Not Handled
Stack
<node,# edges>

i,4
c,2
a,1
s,1

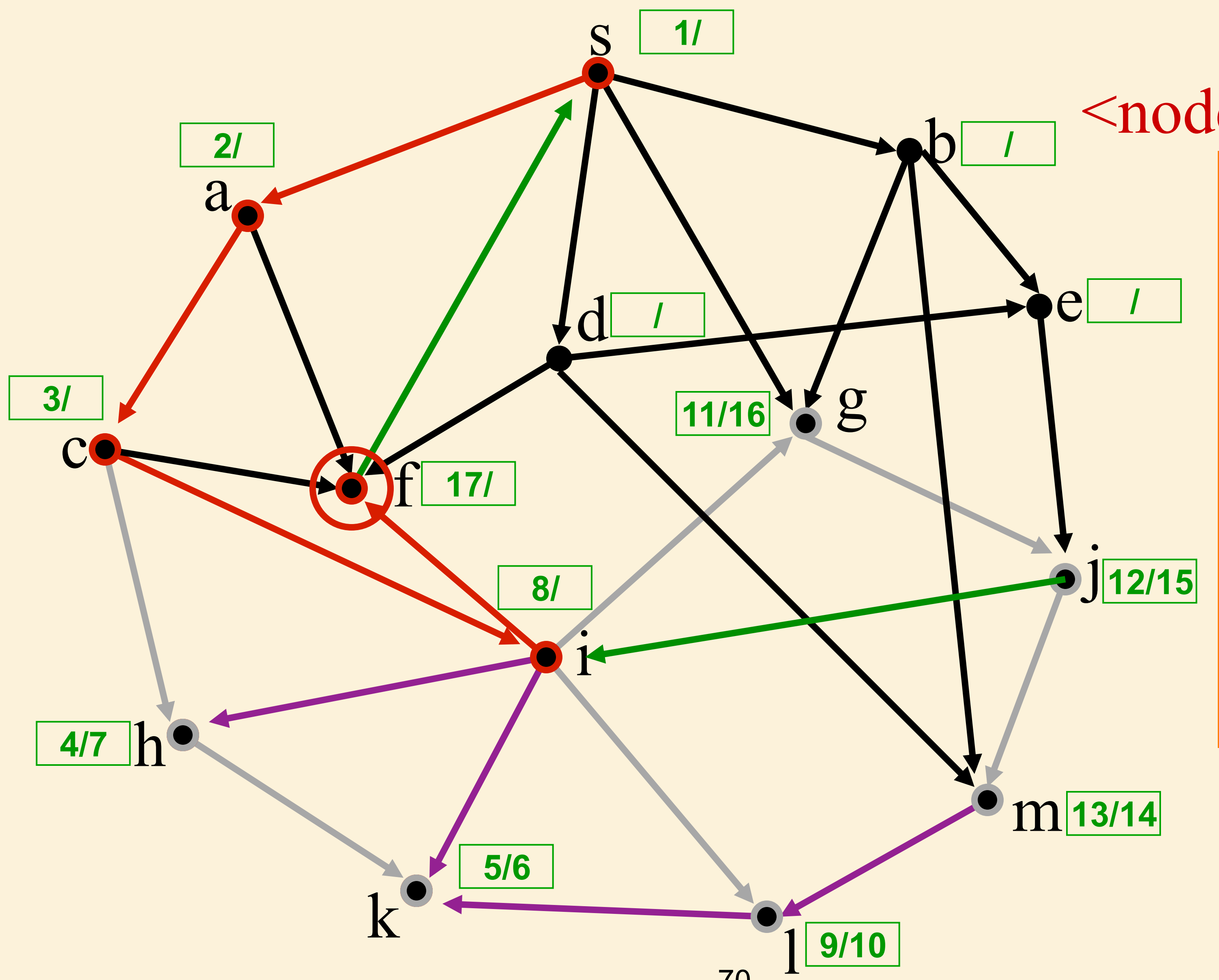
DFS

Found
Not Handled
Stack
<node,# edges>



f,0
i,5
c,2
a,1
s,1

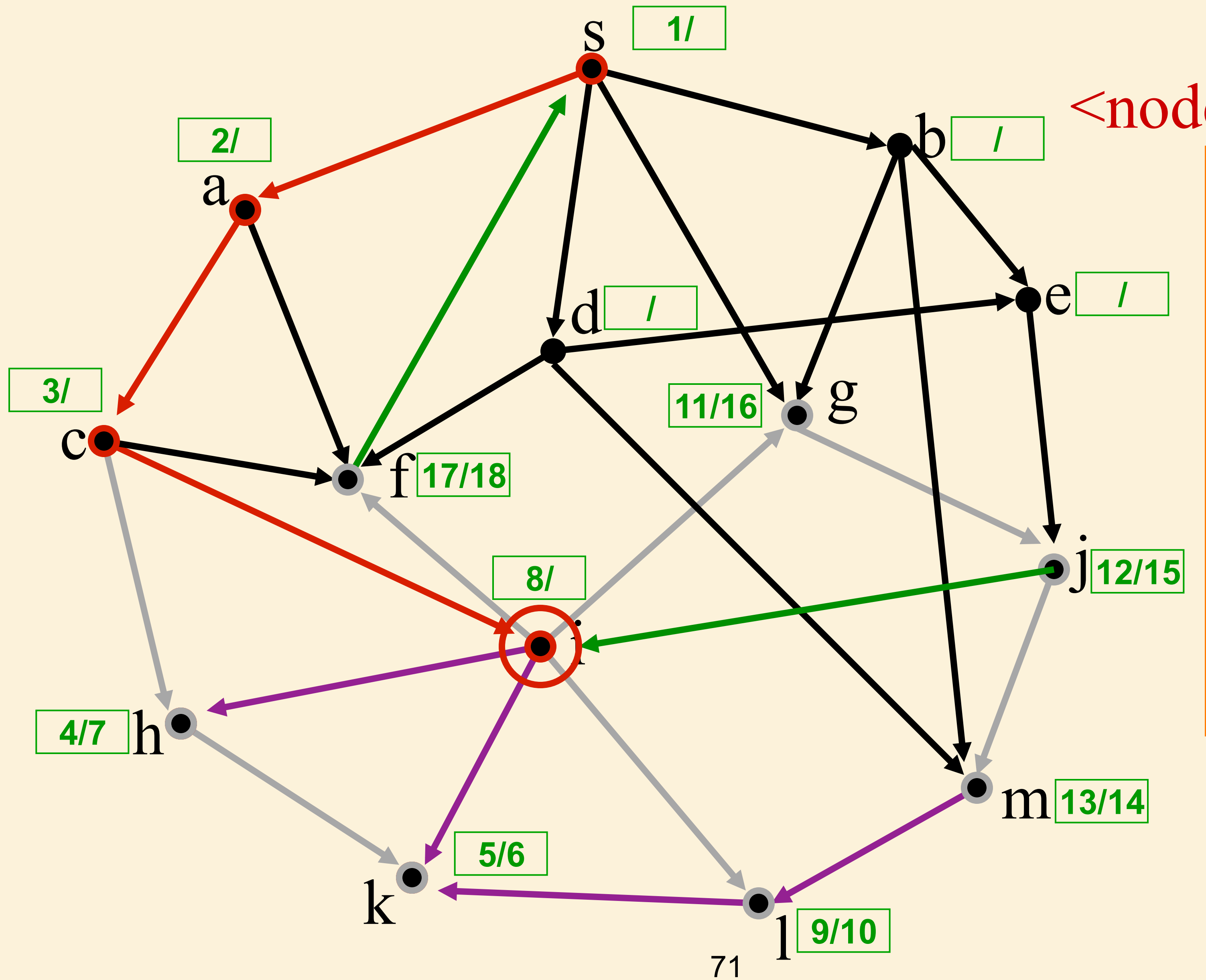
DFS



Found
Not Handled
Stack
<node,# edges>

f,1
i,5
c,2
a,1
s,1

DFS

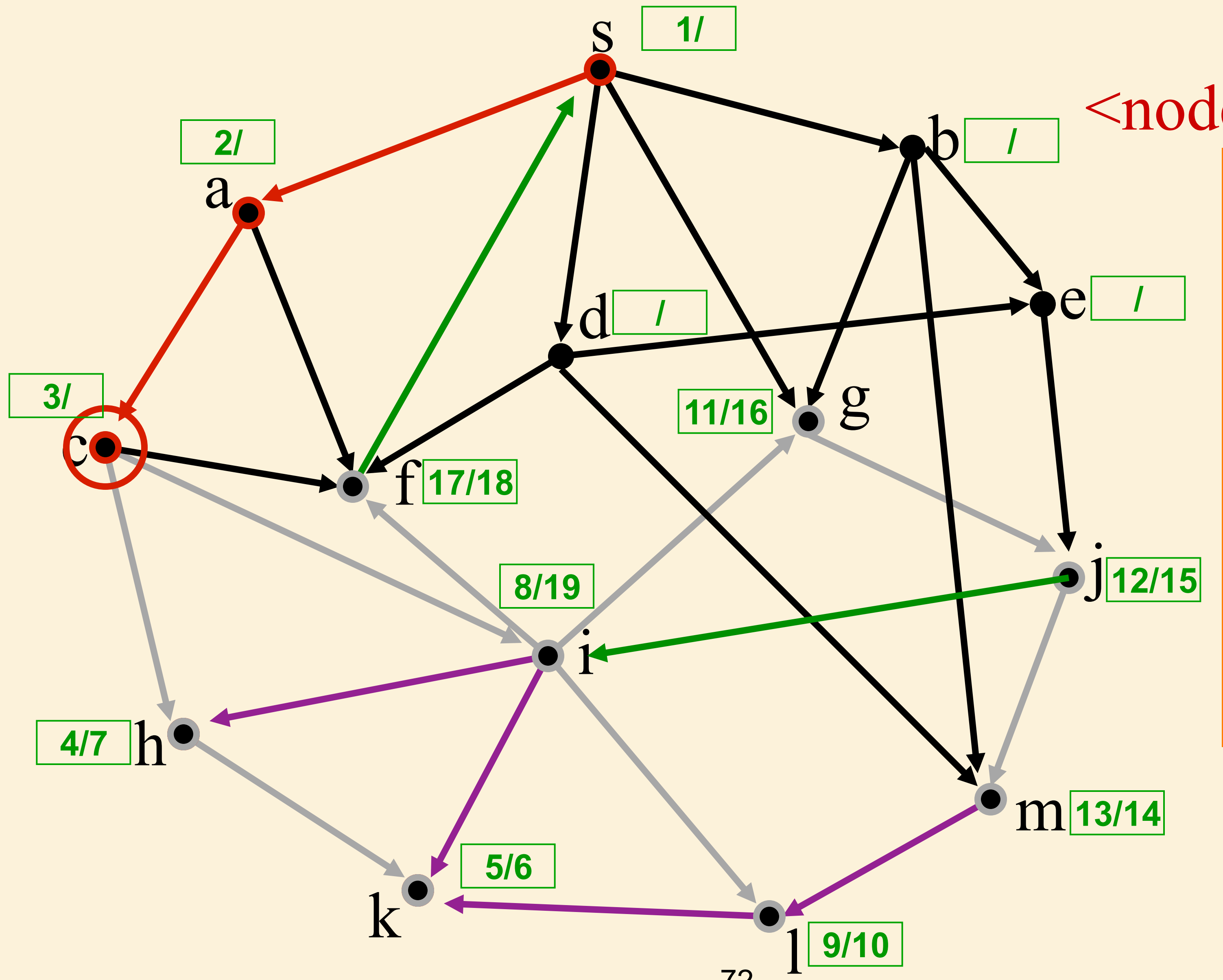


Found
Not Handled
Stack
<node,# edges>

i,5
c,2
a,1
s,1

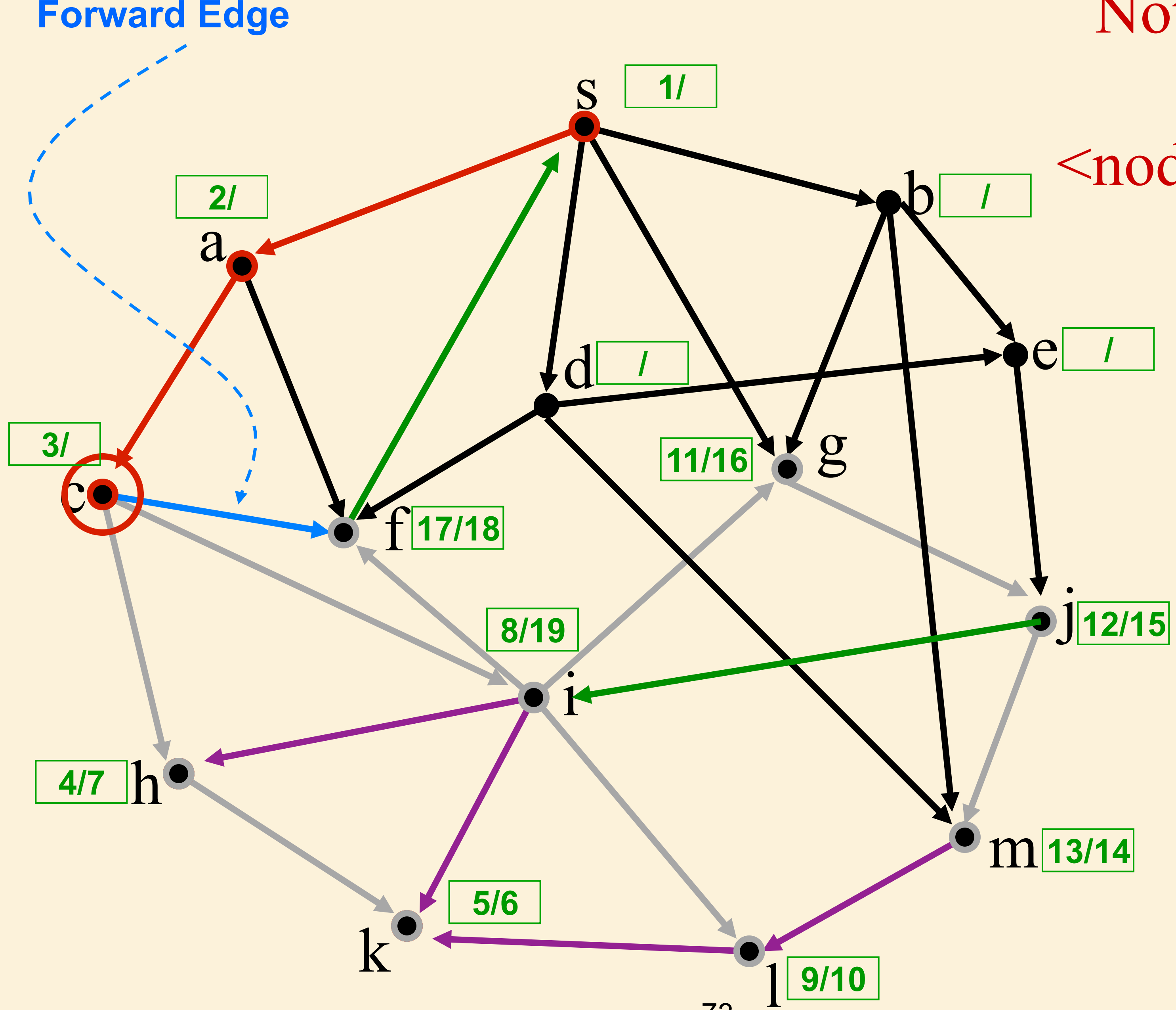
DFS

Found
Not Handled
Stack
<node,# edges>



c,2
a,1
s,1

DFS

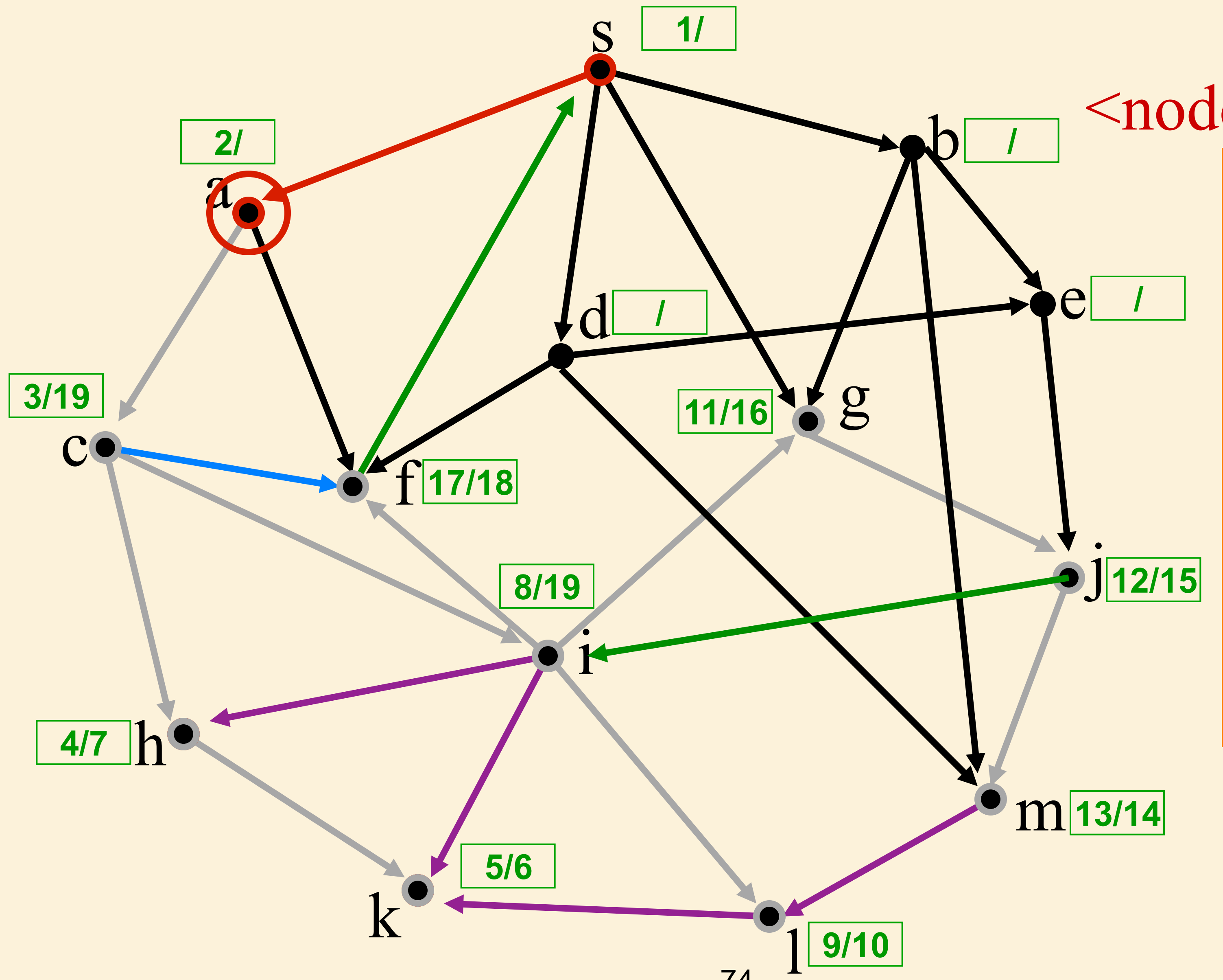


Found
Not Handled
Stack
<node,# edges>

c,3
a,1
s,1

DFS

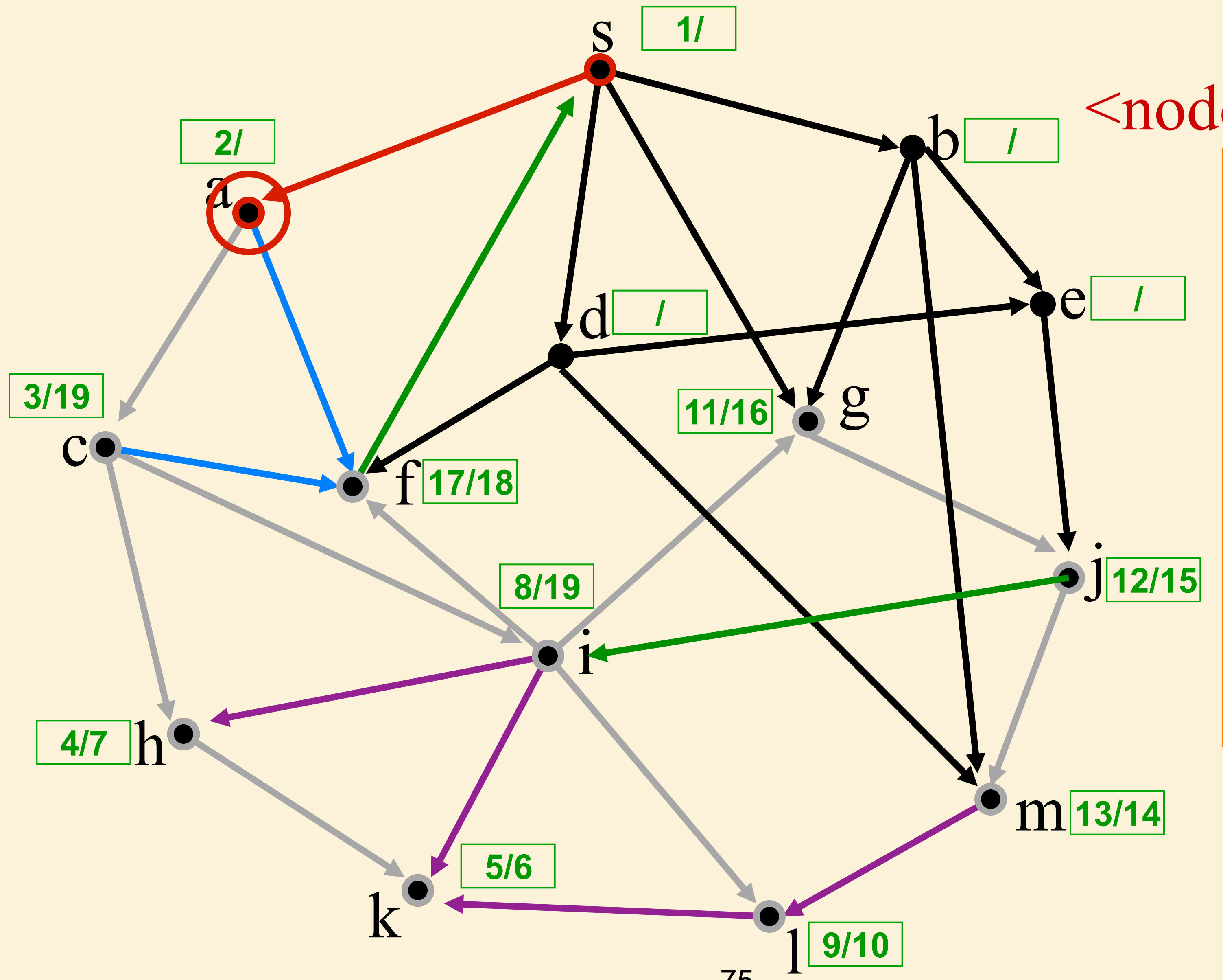
Found
Not Handled
Stack
<node,# edges>



a,1
s,1

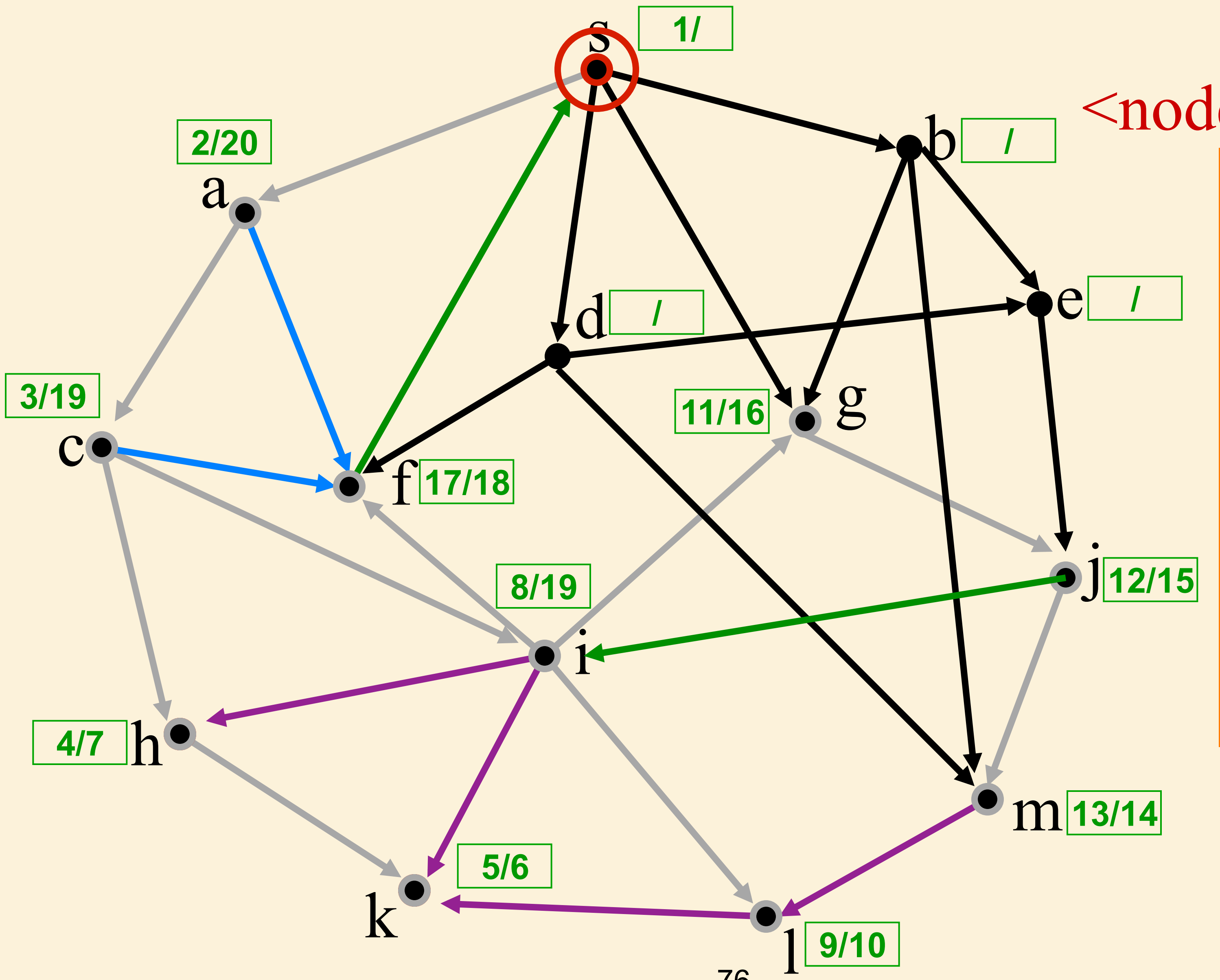
DFS

Found
Not Handled
Stack
<node,# edges>



a,2
s,1

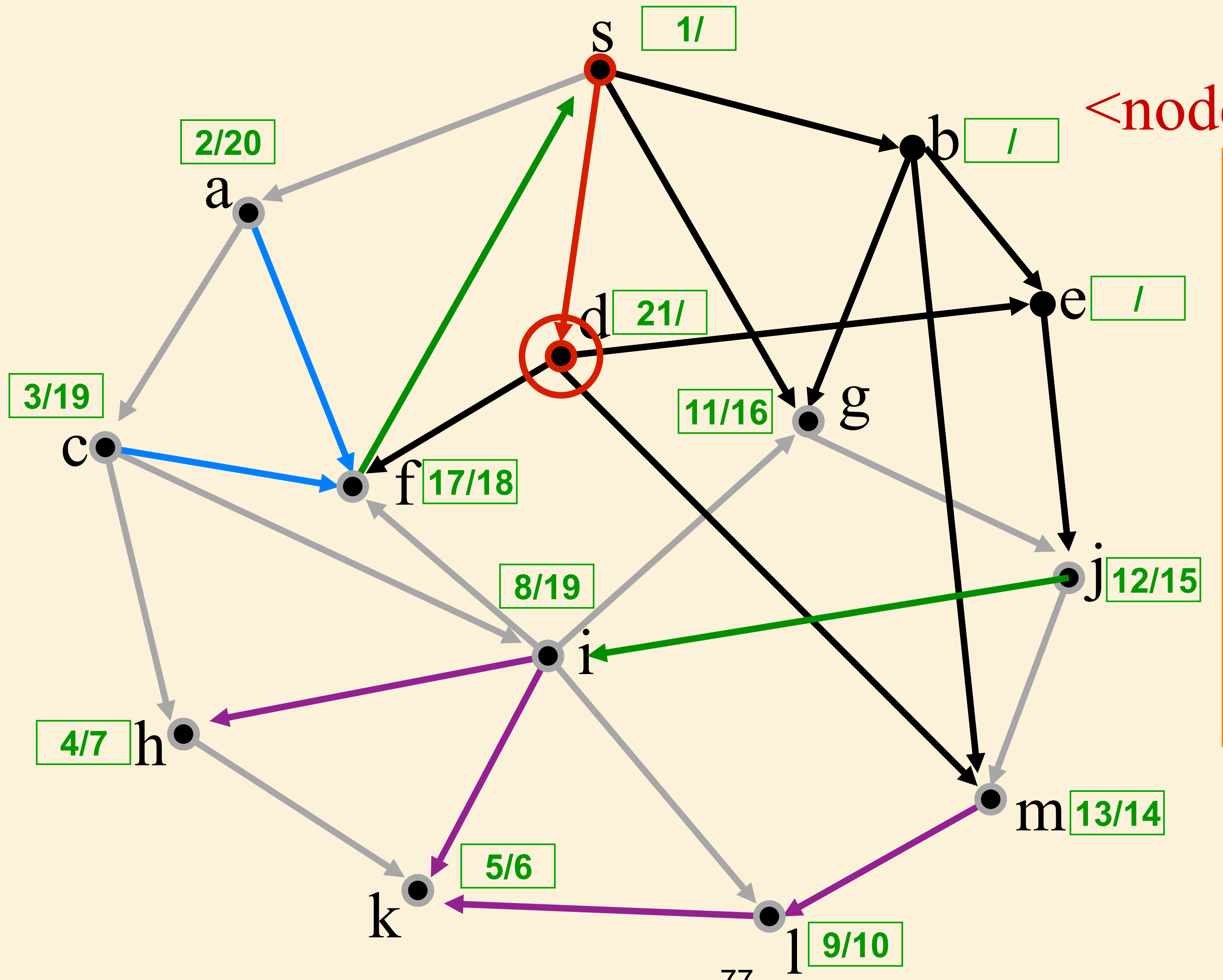
DFS



Found
Not Handled
Stack
<node,# edges>

s,1

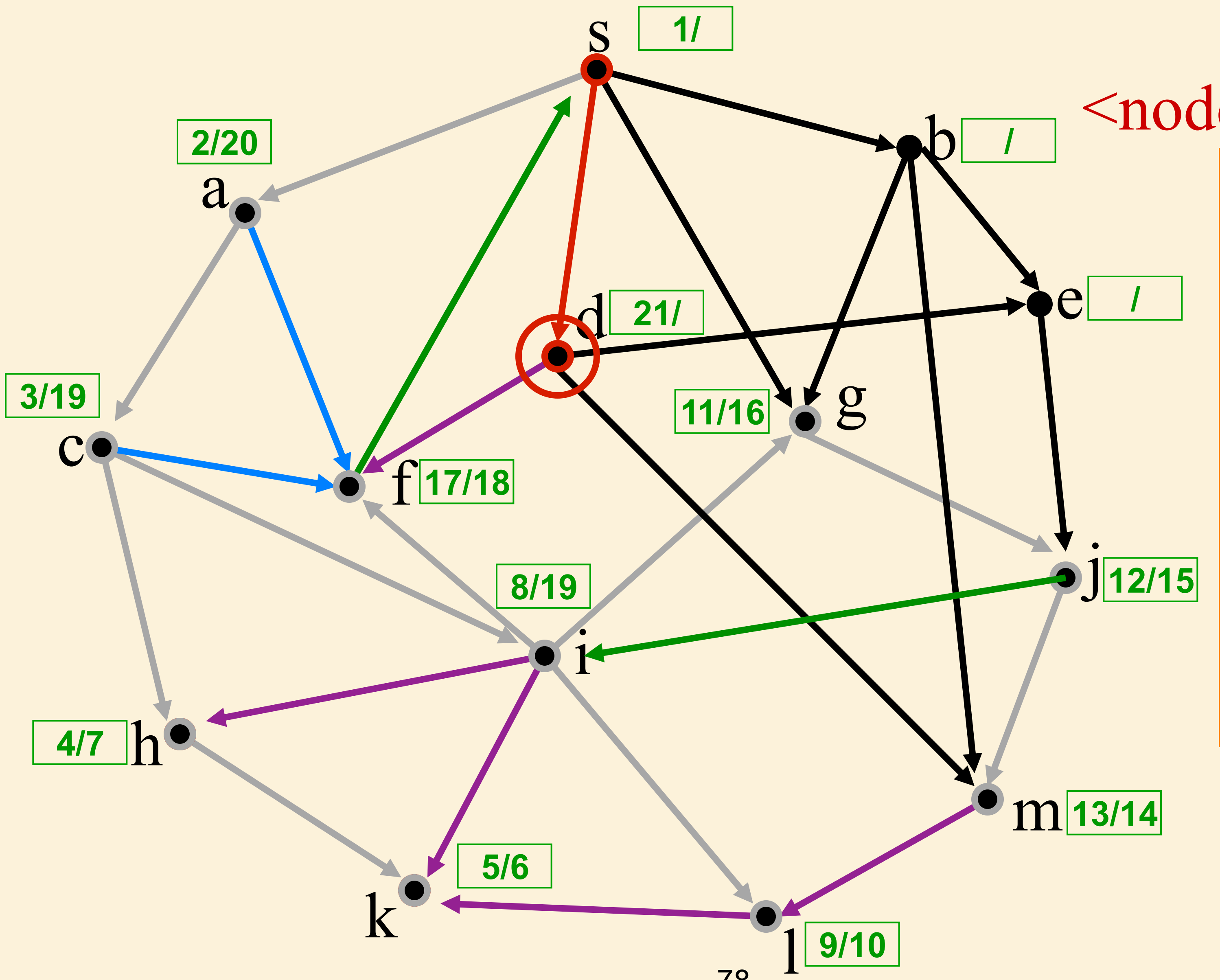
DFS



Found
Not Handled
Stack
<node,# edges>

d,0
s,2

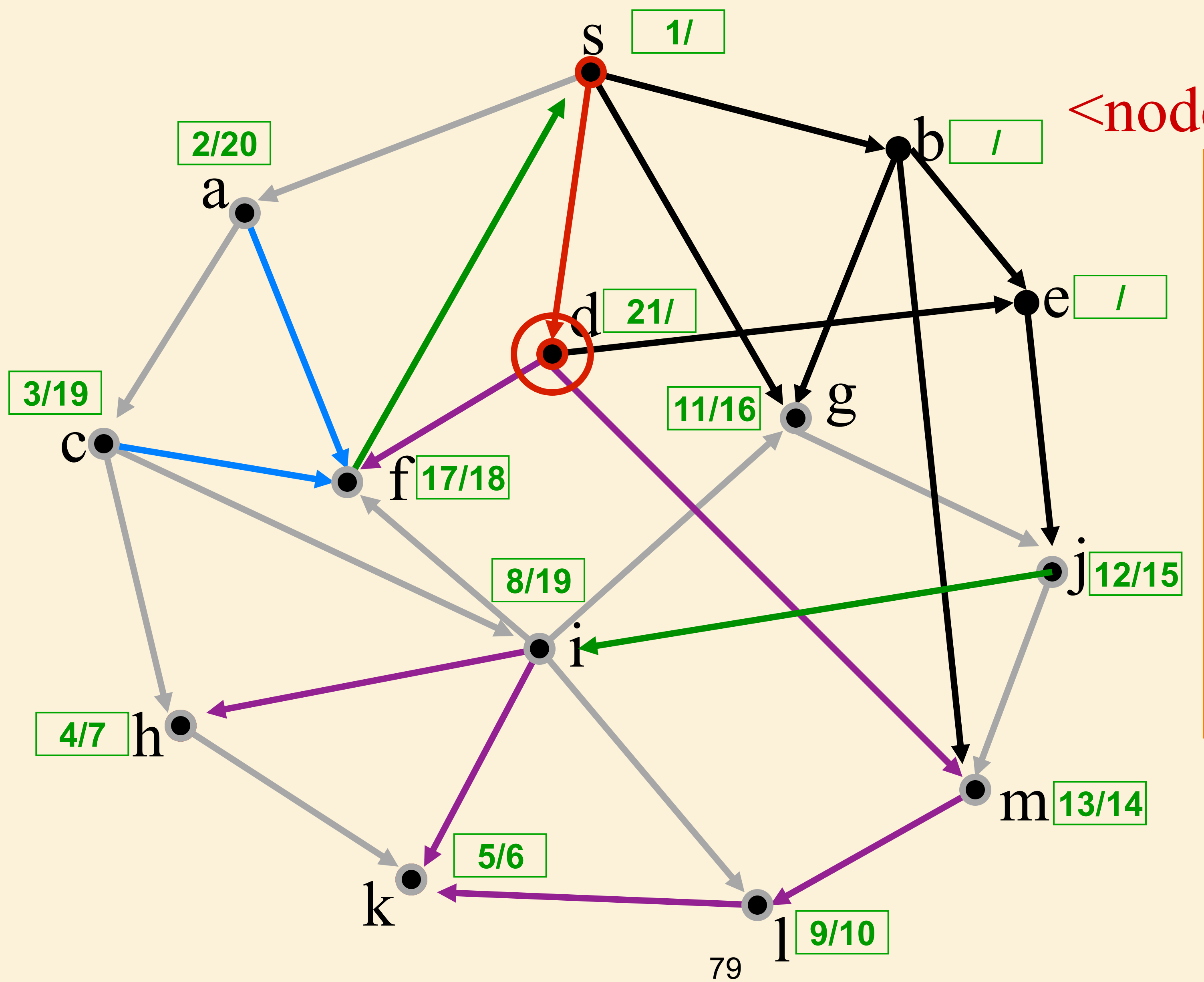
DFS



Found
Not Handled
Stack
<node,# edges>

d,1
s,2

DFS

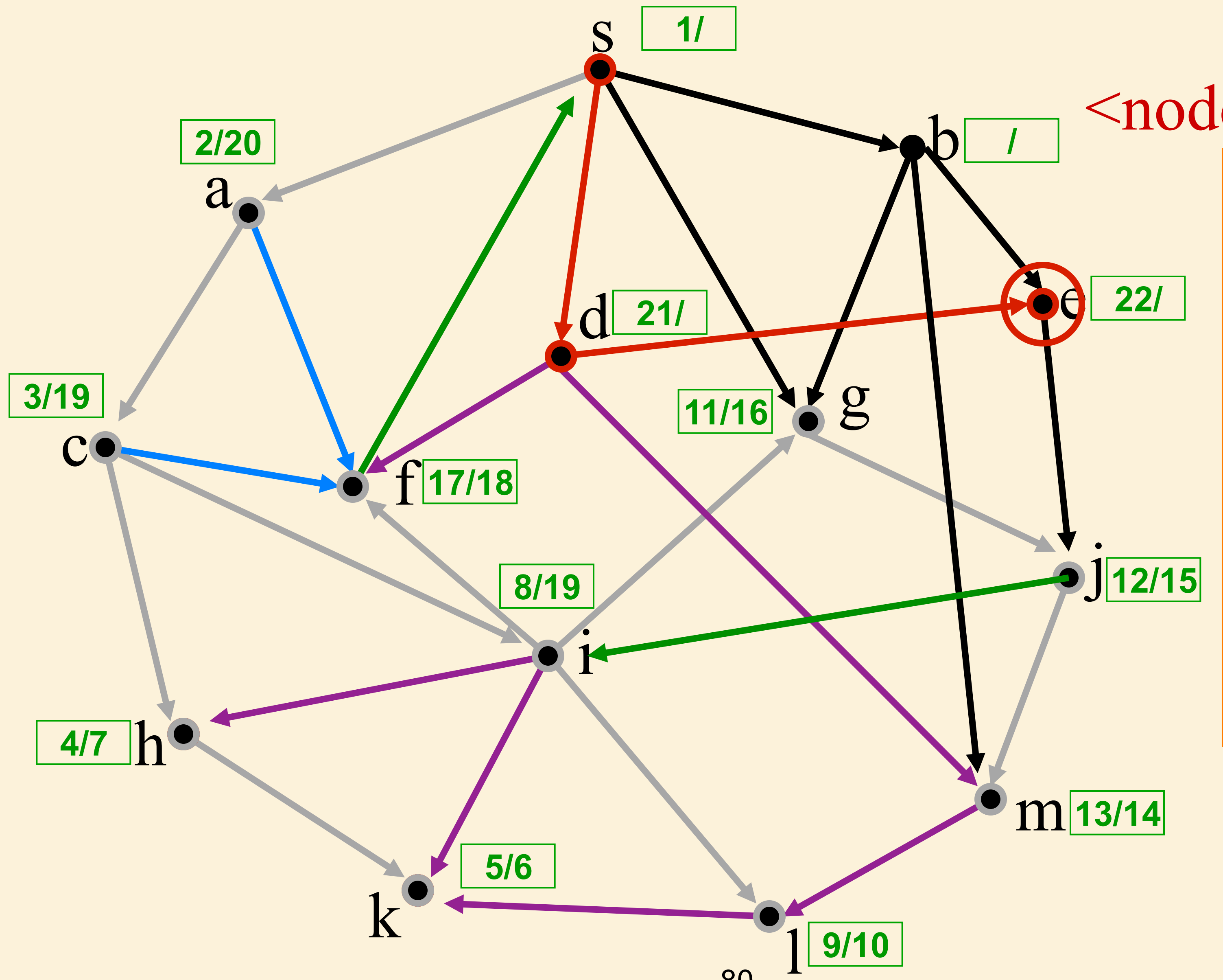


Found
Not Handled
Stack
<node,# edges>

d,2
s,2

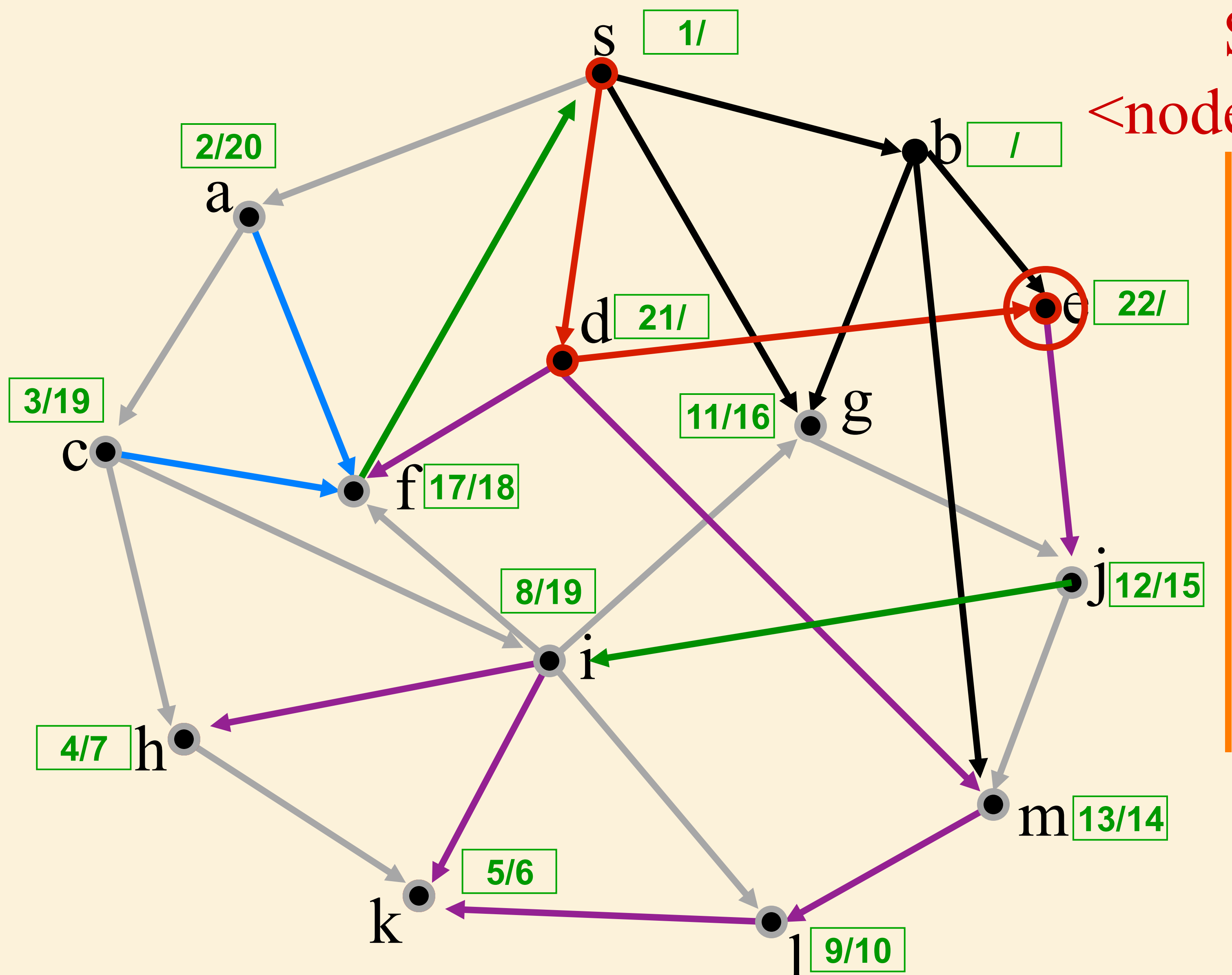
DFS

Found
Not Handled
Stack
<node,# edges>



e,0
d,3
s,2

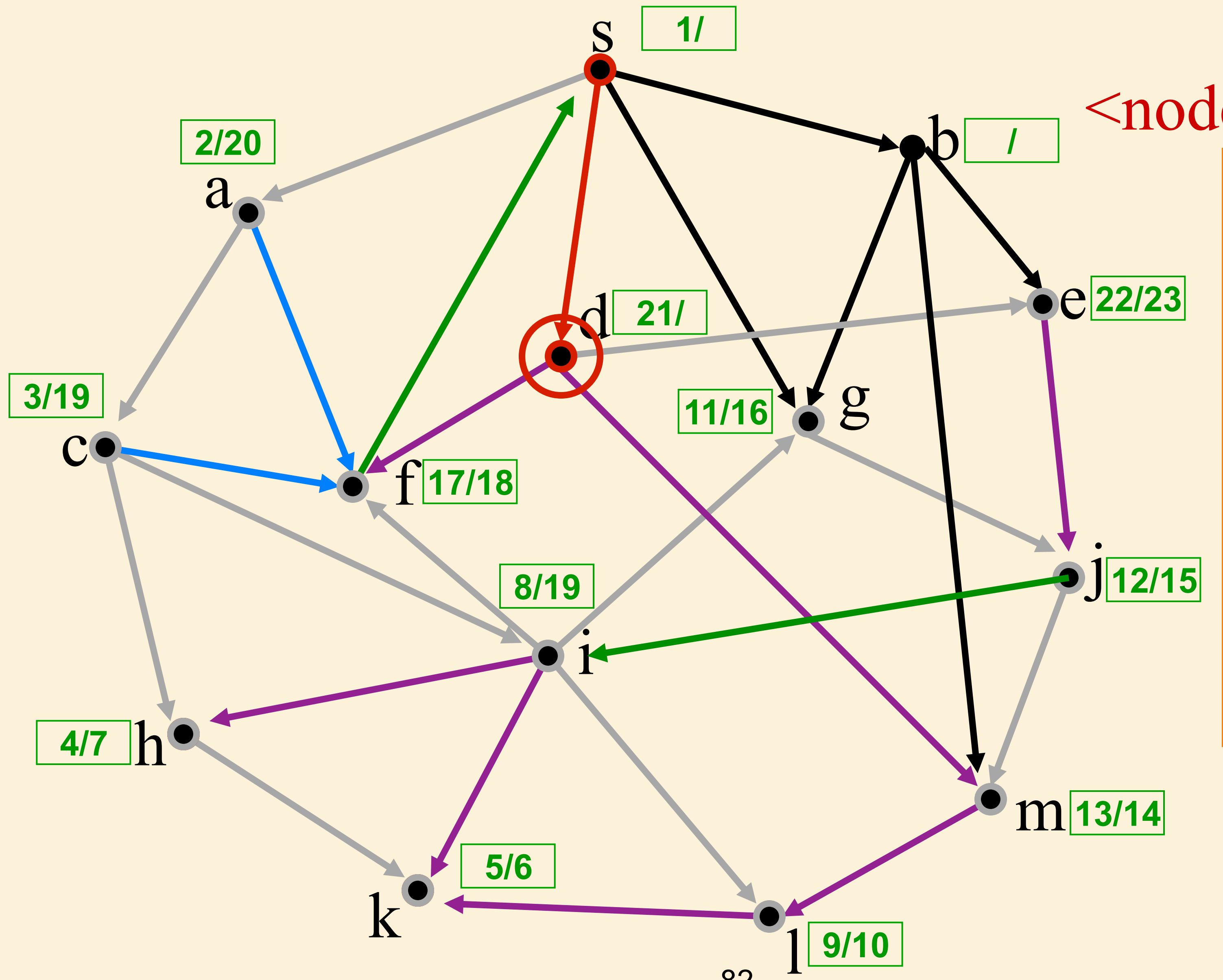
DFS



Found
Not Handled
Stack
<node,# edges>

e,1
d,3
s,2

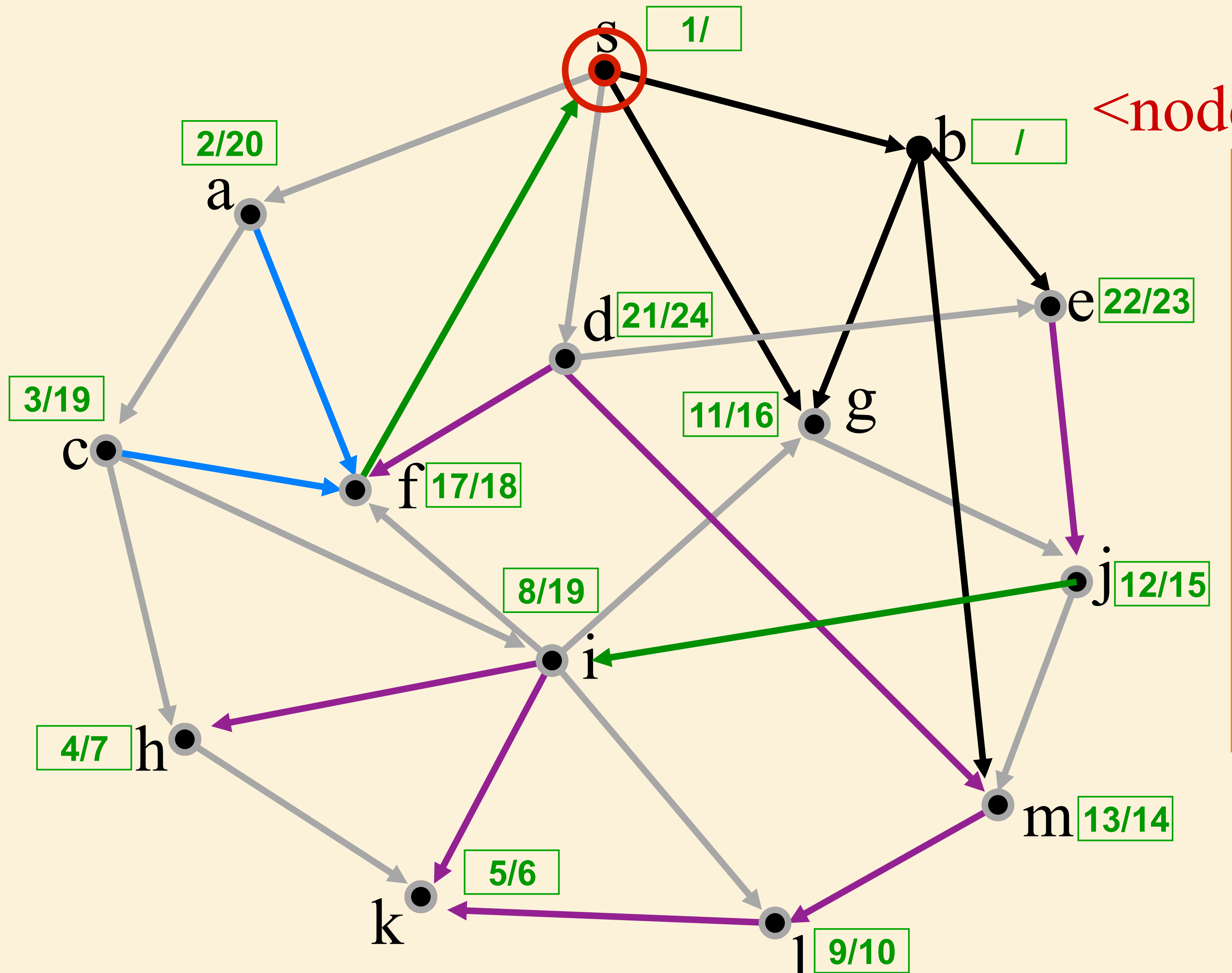
DFS



Found
Not Handled
Stack
<node,# edges>

d,3
s,2

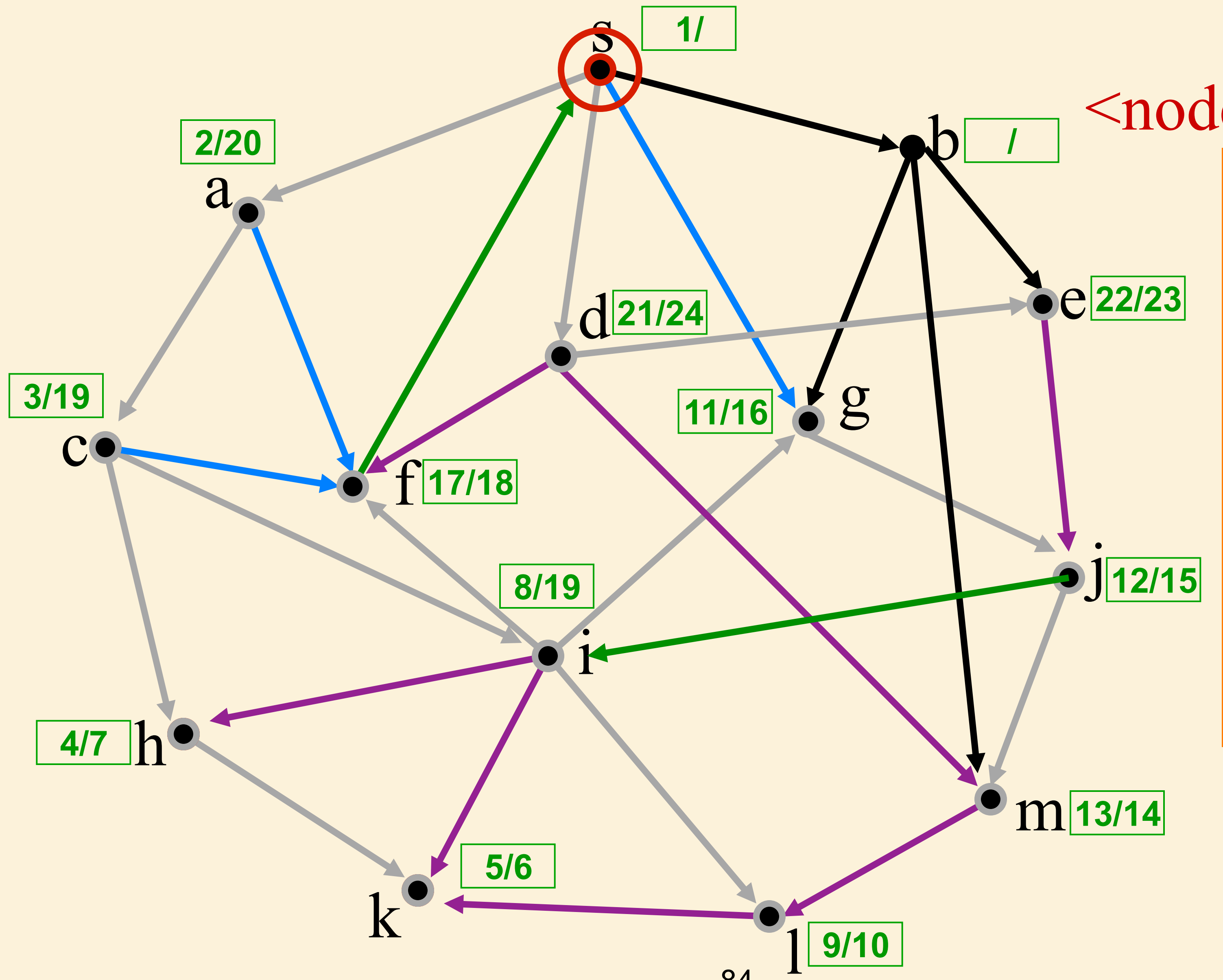
DFS



Found
Not Handled
Stack
<node,# edges>

s,2

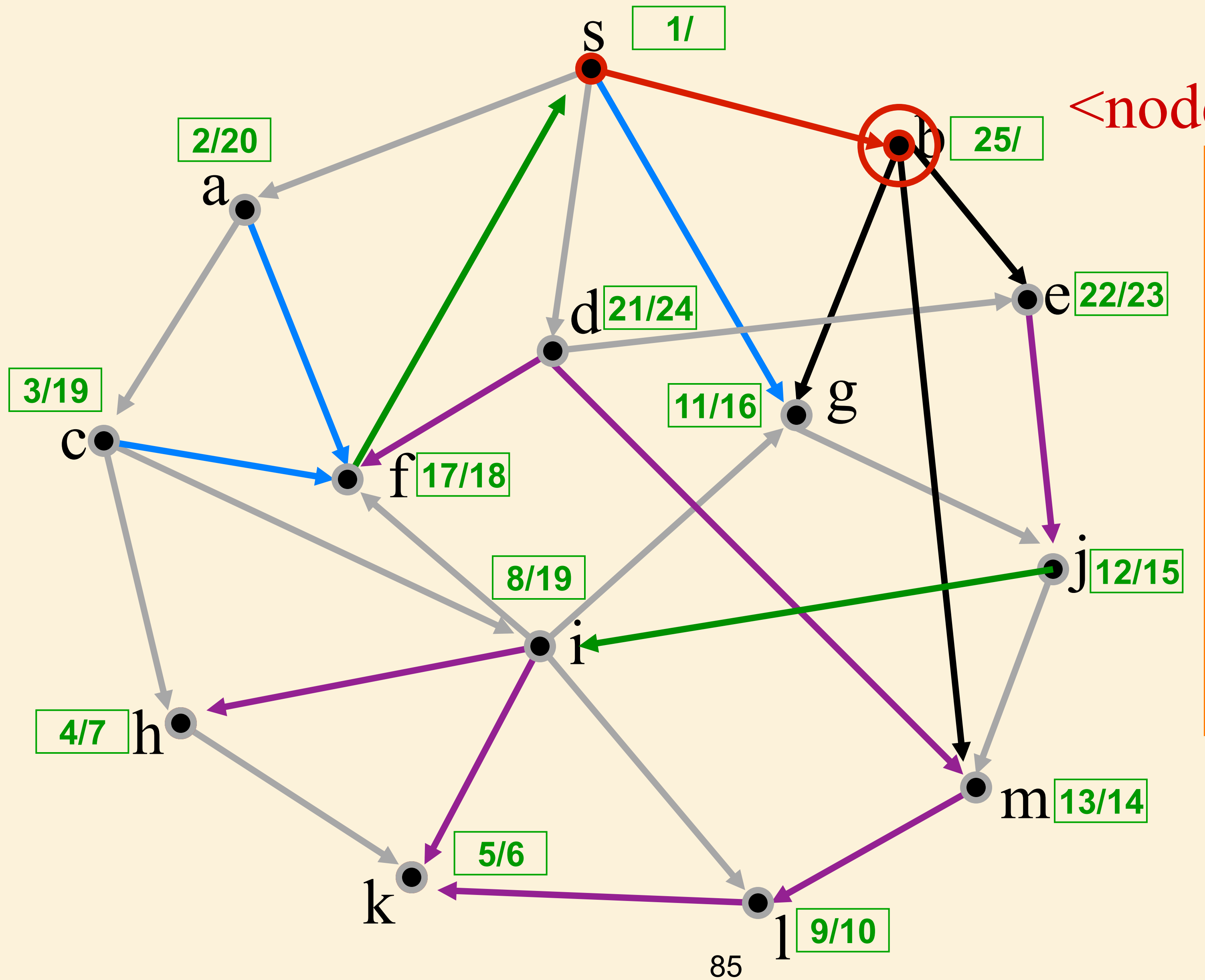
DFS



Found
Not Handled
Stack
<node,# edges>

s,3

DFS



Found

Not Handled

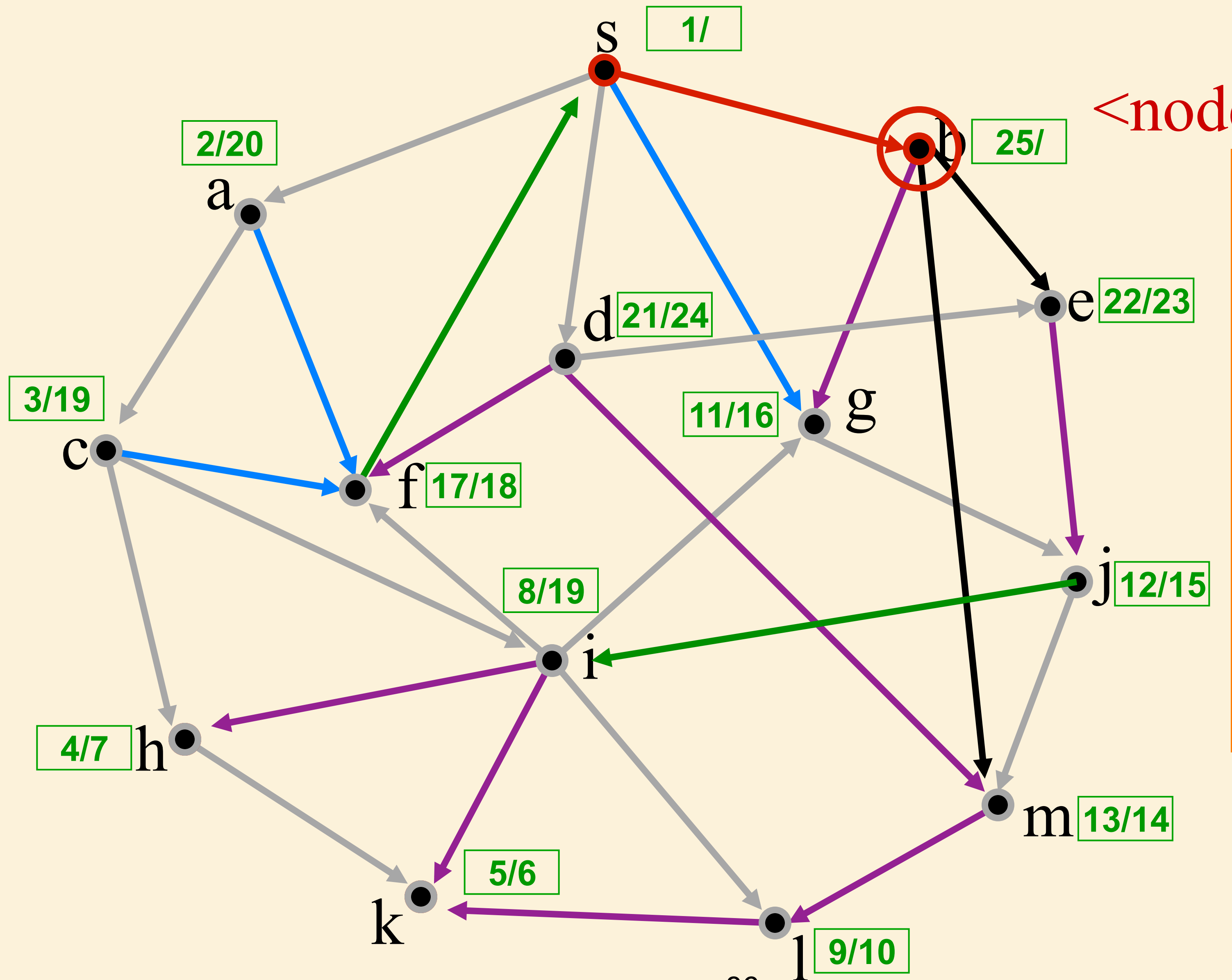
Stack

<node,# edges>

b,0

s,4

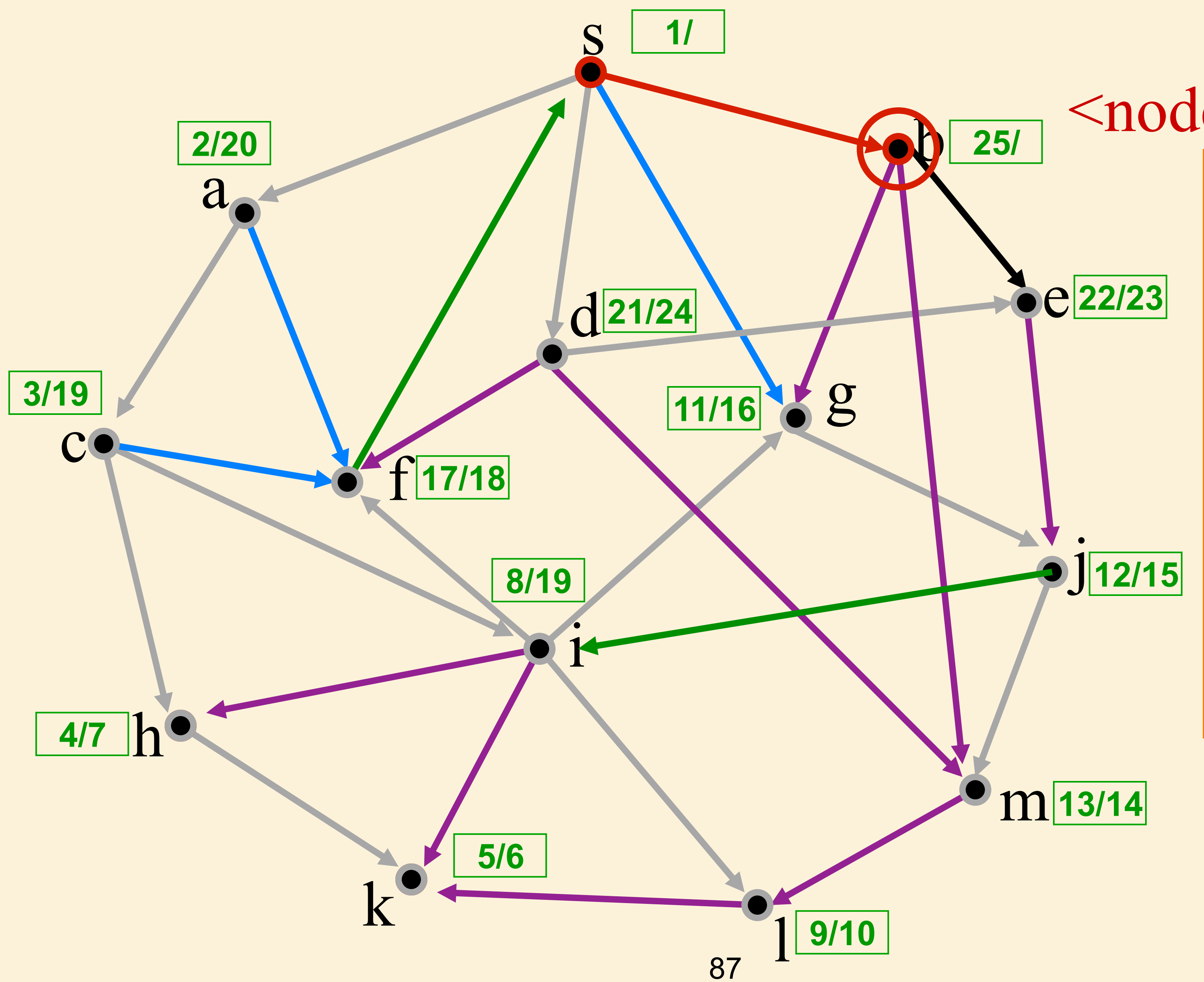
DFS



Found
Not Handled
Stack
<node,# edges>

b,1
s,4

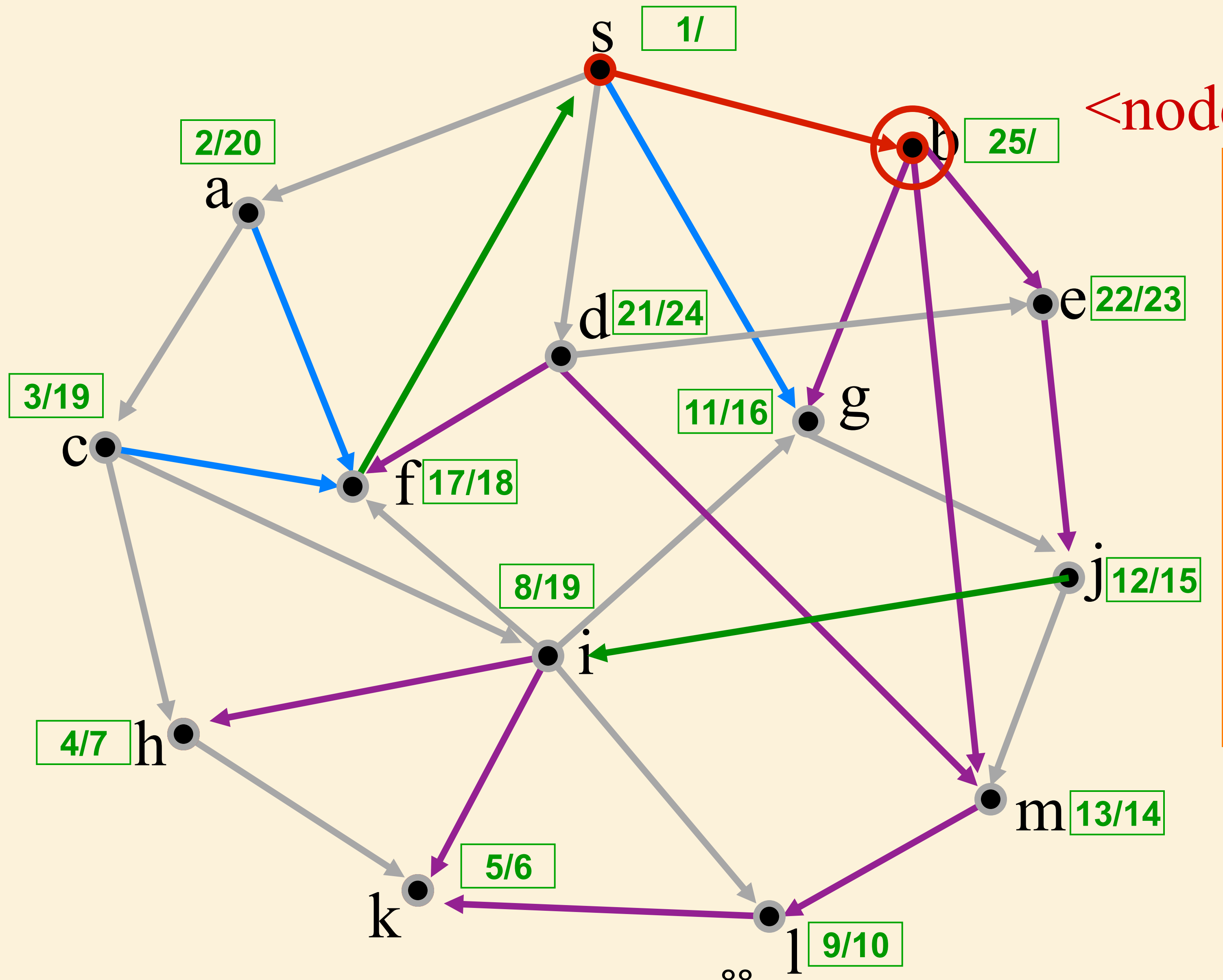
DFS



Found
Not Handled
Stack
<node,# edges>

b,2
s,4

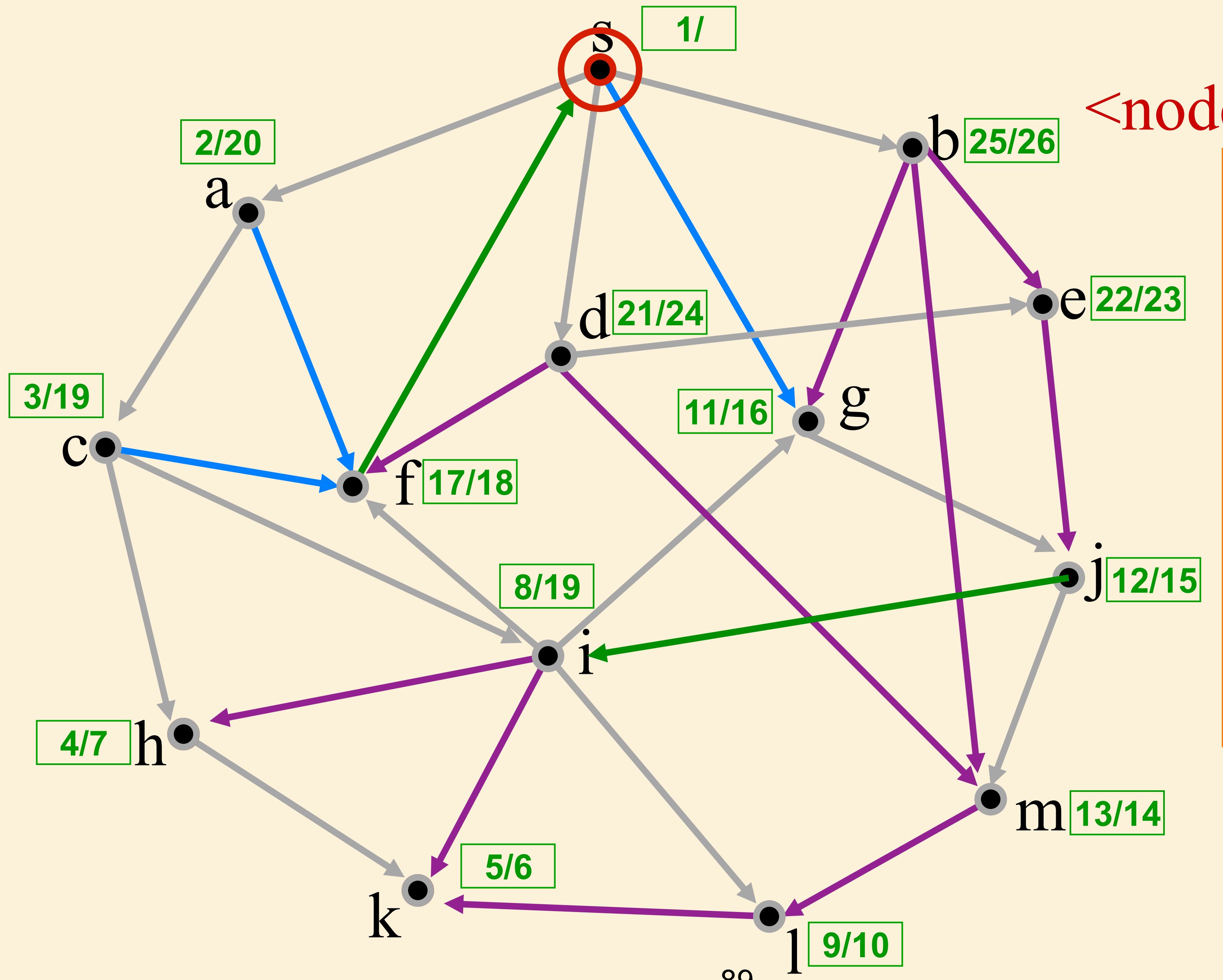
DFS



Found
Not Handled
Stack
<node,# edges>

b,3
s,4

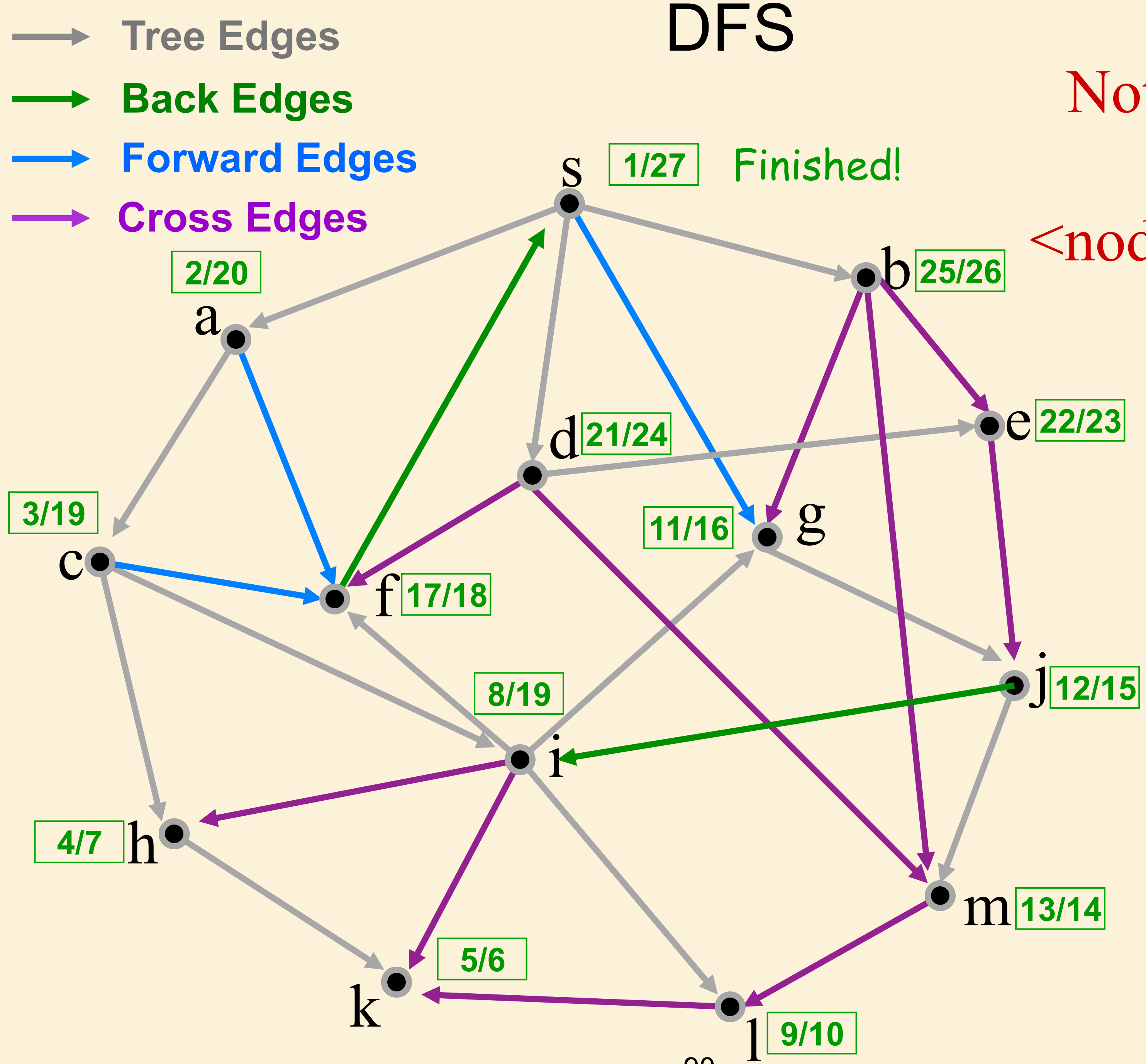
DFS



Found
Not Handled
Stack
<node,# edges>

s,4

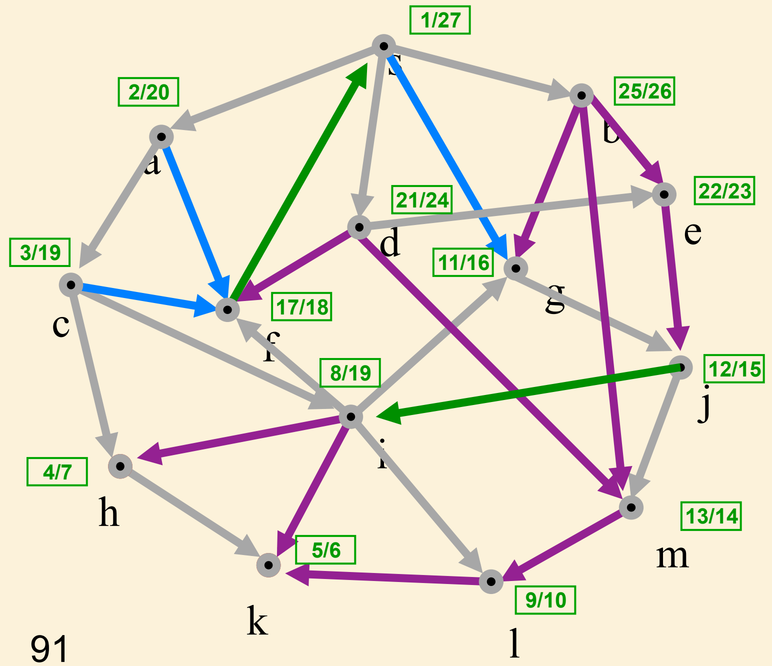
DFS



Found
Not Handled
Stack
<node,# edges>

Classification of Edges in DFS

1. ***Tree edges*** are edges in the depth-first forest G_π. Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v).
2. ***Back edges*** are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree.
3. ***Forward edges*** are non-tree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree.
4. ***Cross edges*** are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other.



Classification of Edges in DFS

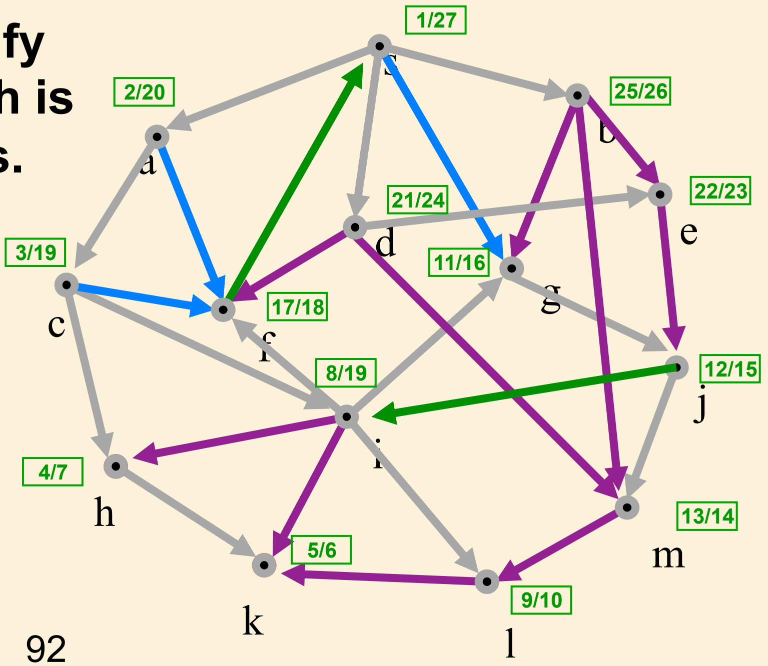
1. ***Tree edges***: Edge (*u*, *v*) is a **tree edge** if *v* was **black** when (*u*, *v*) traversed.

2. ***Back edges:*** (*u*, *v*) is a **back edge** if v was **red** when *(u, v)* traversed.

3. ***Forward edges***: *(u, v)* is a **forward edge** if v was **gray** when *(u, v)* traversed and *d[v] > d[u].*

4. ***Cross edges*** *(u,v)* is a **cross edge** if v was **gray** when *(u, v)* traversed and *d [v] < d[u].*

Classifying edges can help to identify properties of the graph, e.g., a graph is acyclic iff DFS yields no back edges.

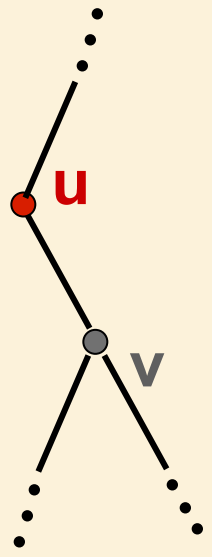


Undirected Graphs

- In a depth-first search of an ***undirected*** graph, every edge is either a **tree edge** or a **back edge**.
- **Why?**

Undirected Graphs

- **Suppose that (u,v)** is a **forward edge** or a **cross edge** in a DFS of an undirected graph.
- **(u,v)** is a **forward edge** or a **cross edge** when **v** is already **handled (grey)** when accessed from **u**.
- This means that all vertices reachable from **v** have been explored.
- Since we are currently handling **u**, **u** must be **red**.
- Clearly **v** is reachable from **u**.
- Since the graph is undirected, **u** must also be reachable from **v**.
- Thus **u** must already have been handled: **u** must be **grey**.
- **Contradiction!**



Depth-First Search Algorithm

DFS(G)

```
1  for each vertex $u \in V[G]$
2       do $color[u] \leftarrow$ BLACK
3          $\pi[u] \leftarrow$ NIL
4  $time \leftarrow 0$
5  for each vertex $u \in V[G]$
6       do if $color[u] =$ BLACK
7             then DFS-VISIT($u$)
```

DFS-Visit (u)

Precondition: vertex u is undiscovered

Postcondition: all vertices reachable from u have been processed

```
1  $color[u] \leftarrow$ RED              ▷BLACK vertex $u$ has just been discovered.
2  $time \leftarrow time + 1$
3  $d[u] \leftarrow time$
4  for each $v \in Adj[u]$            ▷ Explore edge $(u, v)$.
5       do if $color[v] =$ BLACK
6             then $\pi[v] \leftarrow u$
7                  DFS-VISIT($v$)
8  $color[u] \leftarrow$ GRAY            ▷   GRAY $u$; it is finished.
9  $f[u] \leftarrow time \leftarrow time + 1$
```

Depth-First Search Algorithm

DFS(G)

```
for each vertex $u \in V[G]$
    do $color[u] \leftarrow$ BLACK
        $\pi[u] \leftarrow$ NIL
$time \leftarrow 0$
for each vertex $u \in V[G]$
    do if $color[u] =$ BLACK
        then DFS-VISIT($u$)
```

total work = $\theta(V)$

Thus running time = $\theta(V+E)$

DFS-Visit (u)

Precondition: vertex u is undiscovered

Postcondition: all vertices reachable from u have been processed

```
$color[u] \leftarrow$ RED          ▷BLACK vertex $u$ has just been discovered.
$time \leftarrow time + 1$
$d[u] \leftarrow time$
for each $v \in Adj[u]$          ▷ Explore edge $(u, v)$.
    do if $color[v] =$ BLACK
        then $\pi[v] \leftarrow u$
            DFS-VISIT($v$)
$color[u] \leftarrow$ GRAY          ▷ GRAY $u$; it is finished.
$f[u] \leftarrow time \leftarrow time + 1$
```

total work = $\sum_{v \in V} |Adj[v]| = \theta(E)$

Topological Sorting
(e.g., putting tasks in linear order)

An application of Depth-First Search

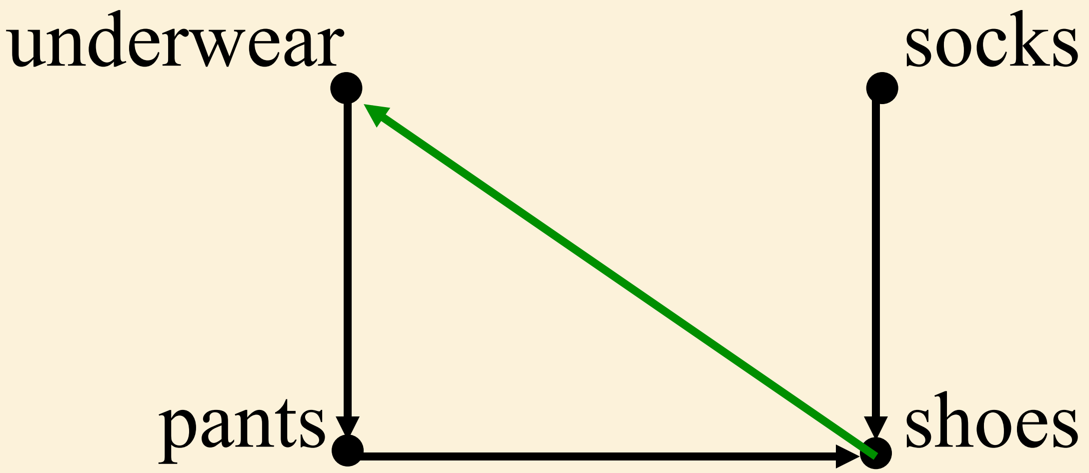
Linear Order

$$\begin{tikzcd}
\text{underwear} \arrow[d] & & \text{socks} \arrow[d] \\
\text{pants} \arrow[rr] & & \text{shoes}
\end{tikzcd}$$

$$\begin{tikzcd}
\text{underwear} \arrow[d, bend left] \\
\text{pants} \arrow[dd, bend left] \\
\text{socks} \arrow[d, bend left] \\
\text{shoes}
\end{tikzcd}$$

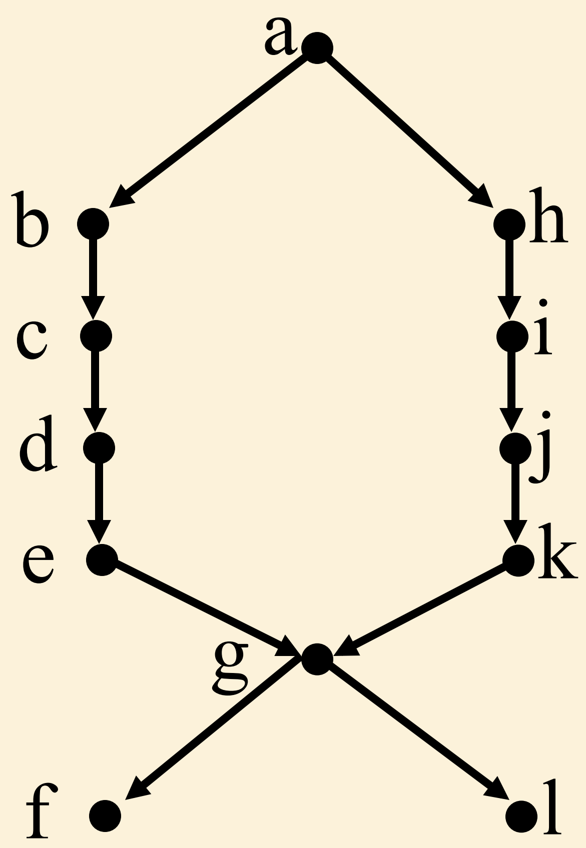
$$\begin{tikzcd}
\text{socks} \arrow[ddd, bend left] \\
\text{underwear} \arrow[d, bend left] \\
\text{pants} \arrow[d, bend left] \\
\text{shoes}
\end{tikzcd}$$

Linear Order



Too many video games?

Linear Order



Precondition:

A Directed Acyclic Graph (DAG)

Post Condition:

Find one valid linear order

Algorithm:

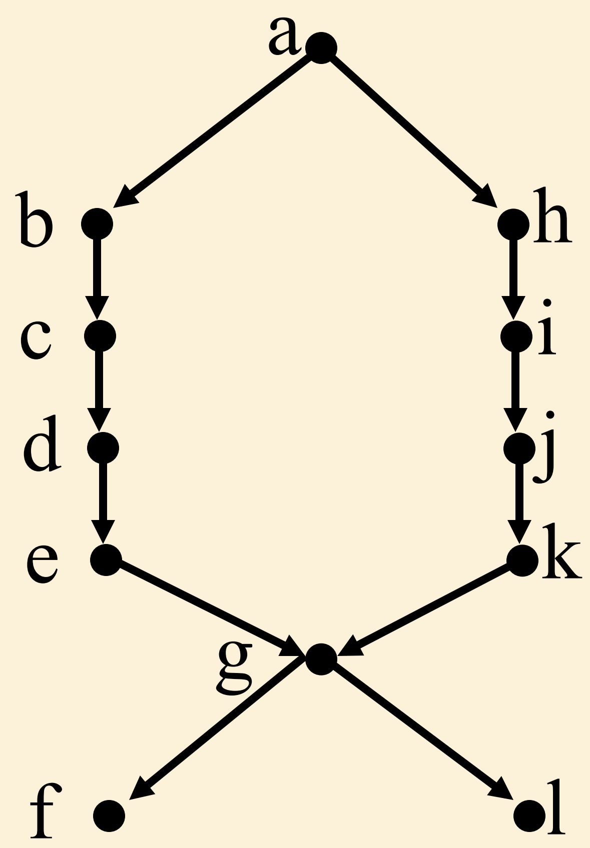
- Find a terminal node (sink). ~~$\Theta(V)$~~
- Put it last in sequence.
- Delete from graph & repeat ~~$\Theta(V^2)$~~

We can do better!

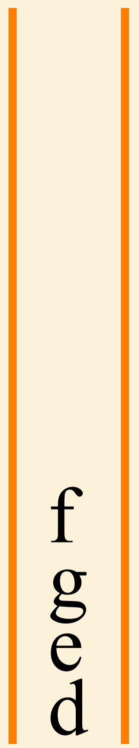
….. l

Linear Order

Alg: DFS



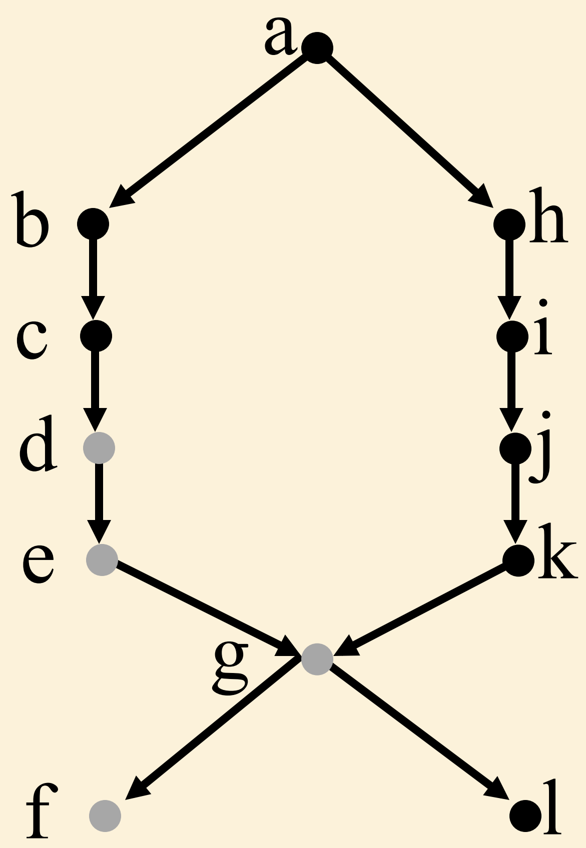
Found
Not Handled
Stack



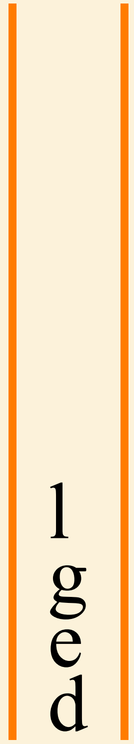
….. f

Linear Order

Alg: DFS



Found
Not Handled
Stack



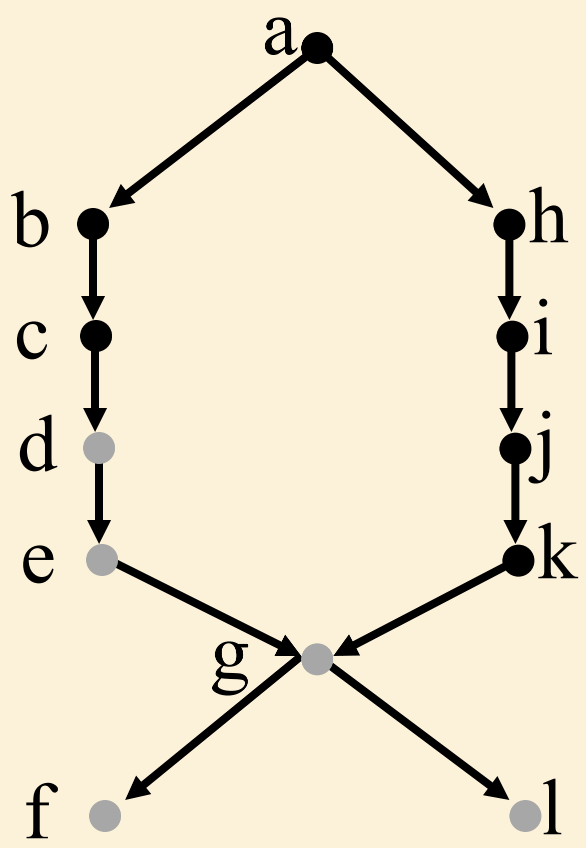
When node is popped off stack, insert at front of linearly-ordered “to do” list.

Linear Order:

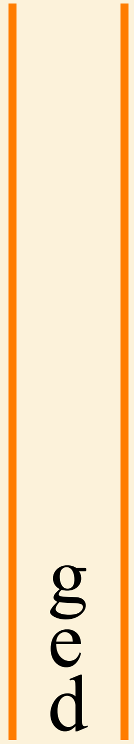
….. f

Linear Order

Alg: DFS



Found
Not Handled
Stack

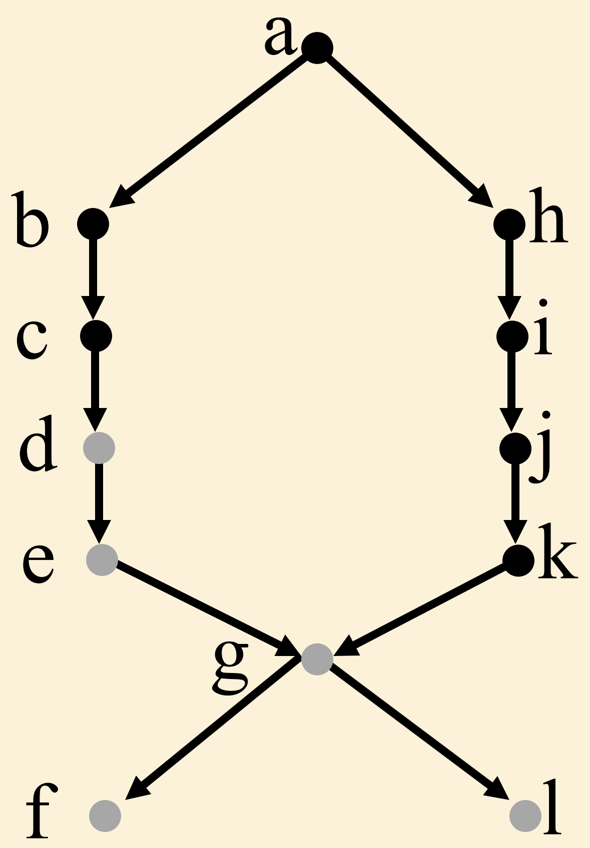


Linear Order:

l,f

Linear Order

Alg: DFS



Found
Not Handled
Stack

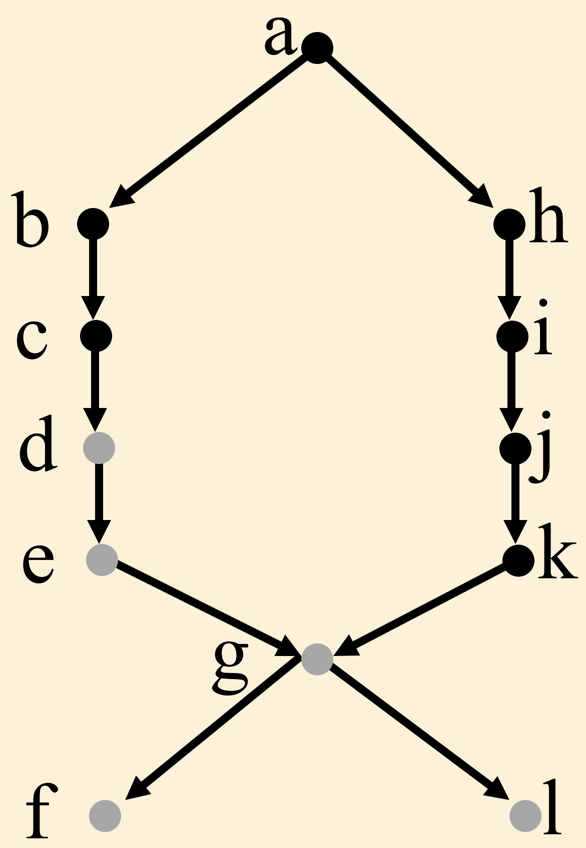
e
d

Linear Order:

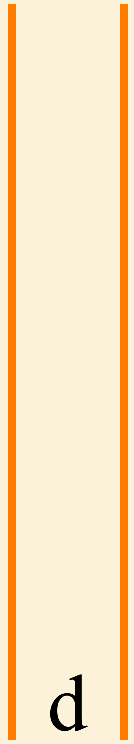
g,l,f

Linear Order

Alg: DFS



Found
Not Handled
Stack

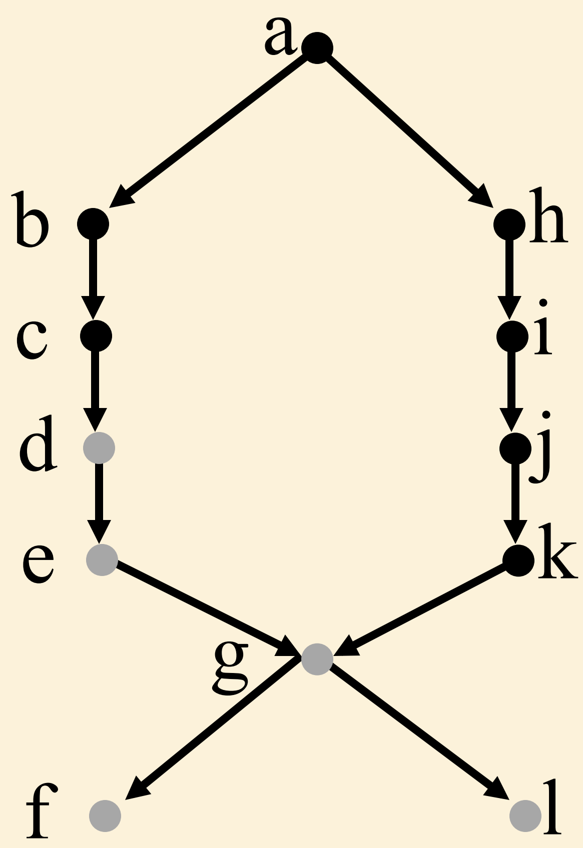


Linear Order:

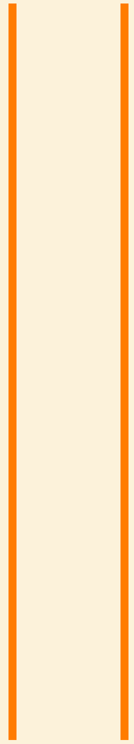
e,g,l,f

Linear Order

Alg: DFS



Found
Not Handled
Stack

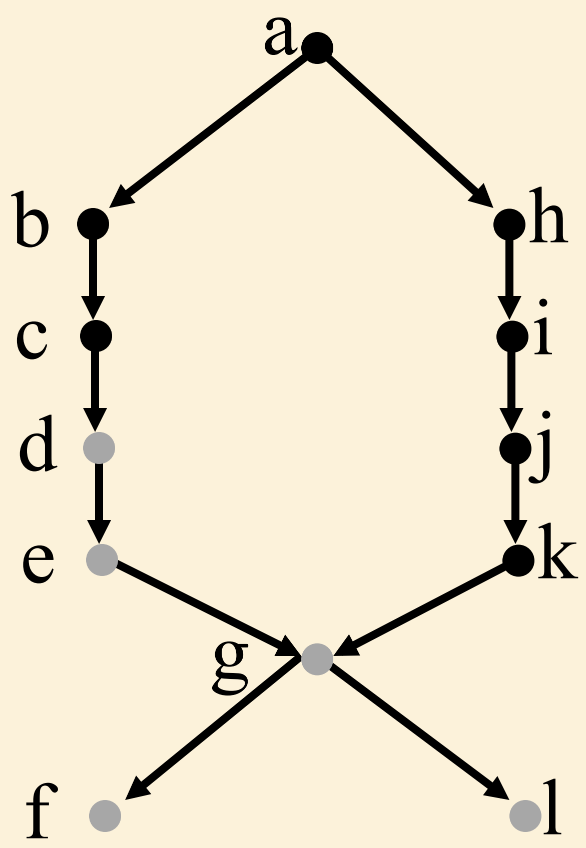


Linear Order:

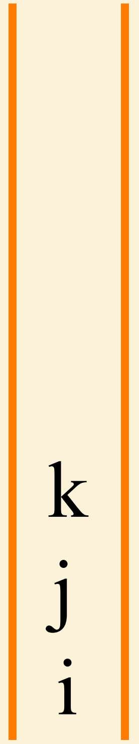
d,e,g,l,f

Linear Order

Alg: DFS



Found
Not Handled
Stack

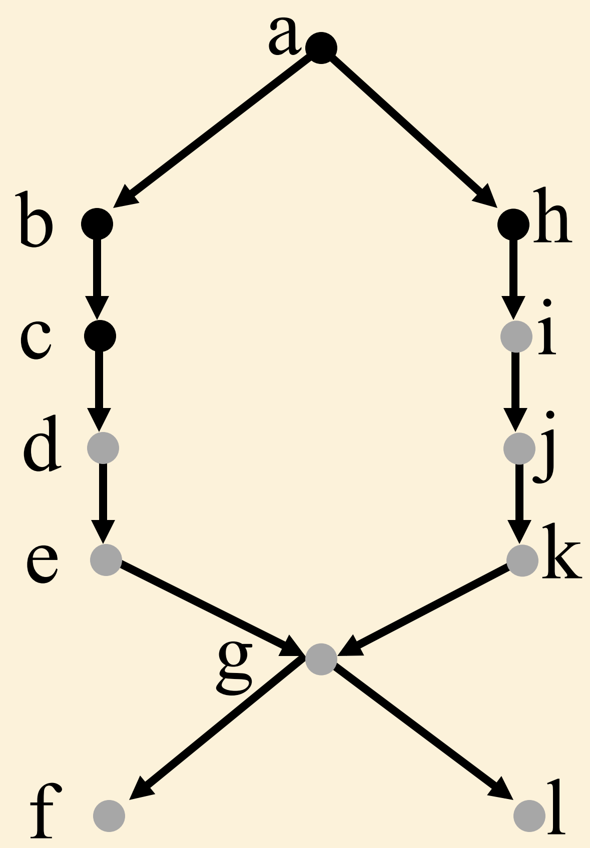


Linear Order:

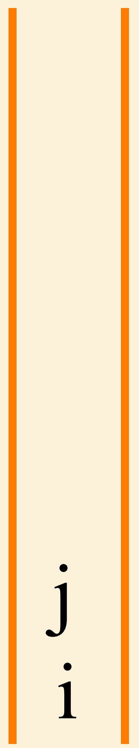
d,e,g,l,f

Linear Order

Alg: DFS



Found
Not Handled
Stack

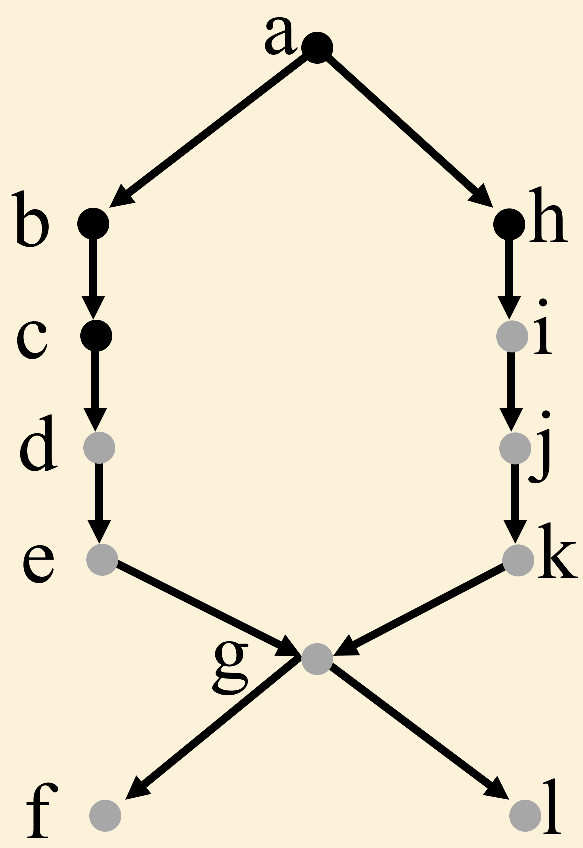


Linear Order:

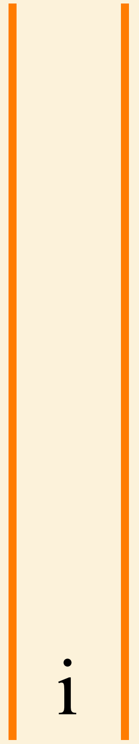
k,d,e,g,l,f

Linear Order

Alg: DFS



Found
Not Handled
Stack

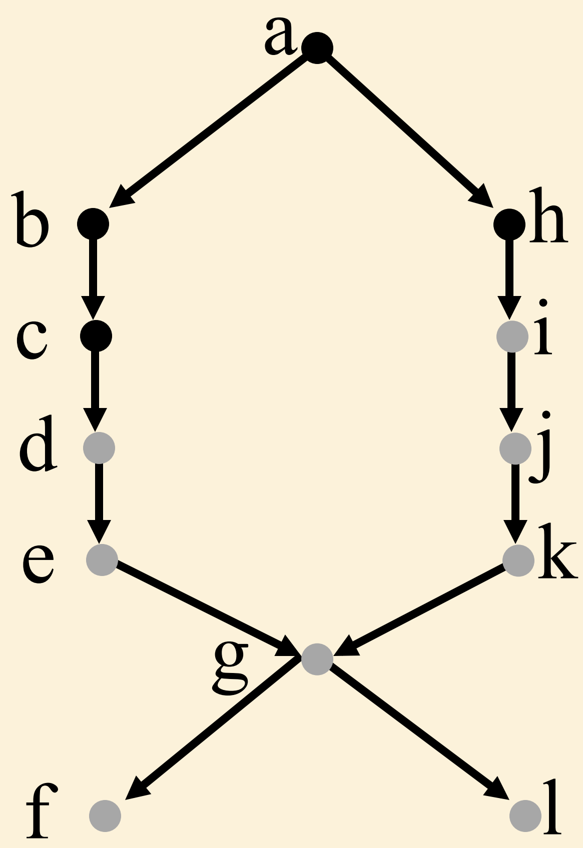


Linear Order:

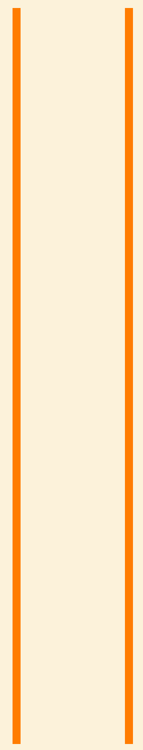
j,k,d,e,g,l,f

Linear Order

Alg: DFS



Found
Not Handled
Stack

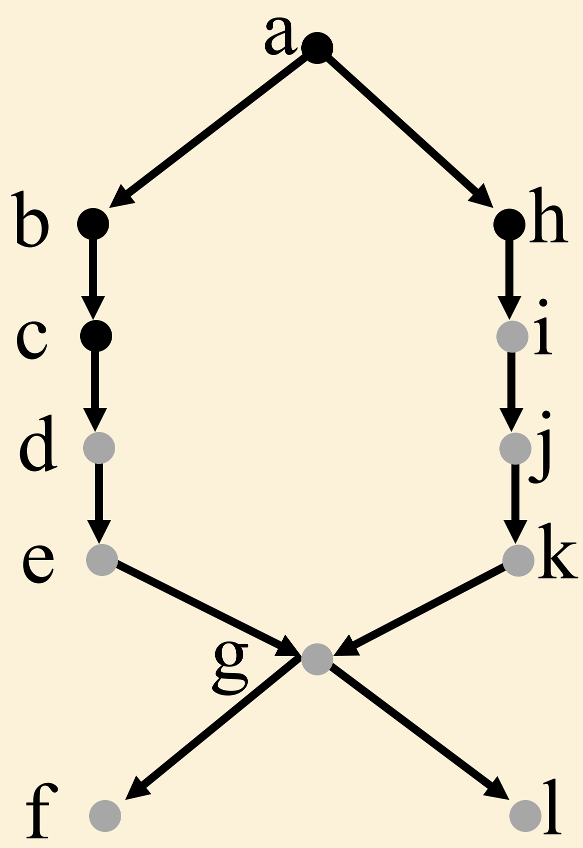


Linear Order:

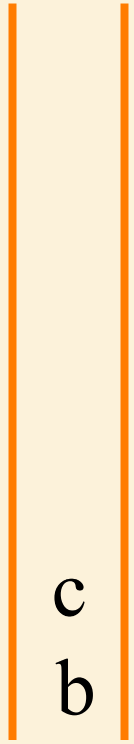
i,j,k,d,e,g,l,f

Linear Order

Alg: DFS



Found
Not Handled
Stack

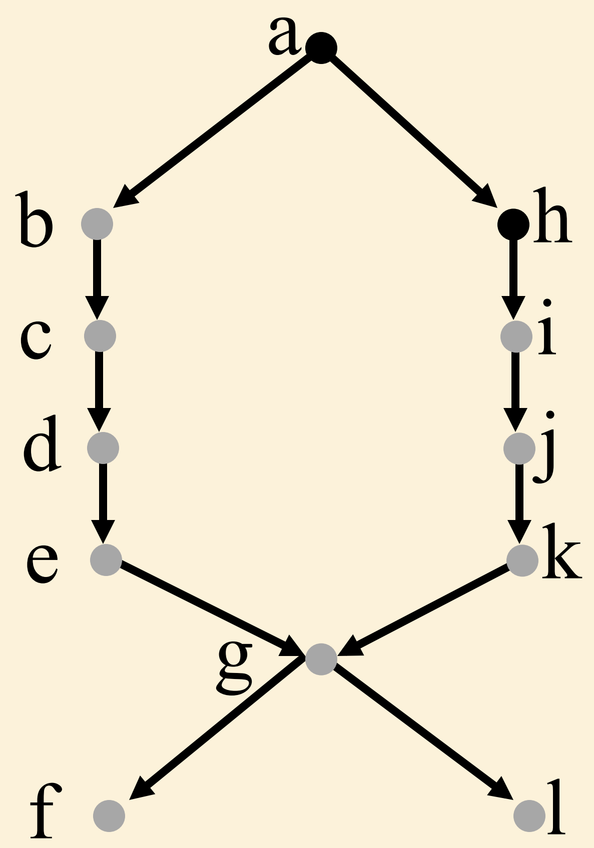


Linear Order:

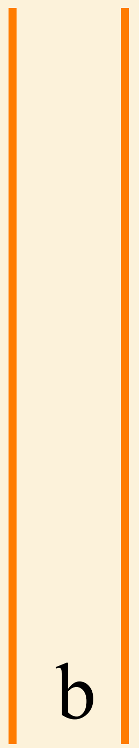
i,j,k,d,e,g,l,f

Linear Order

Alg: DFS



Found
Not Handled
Stack

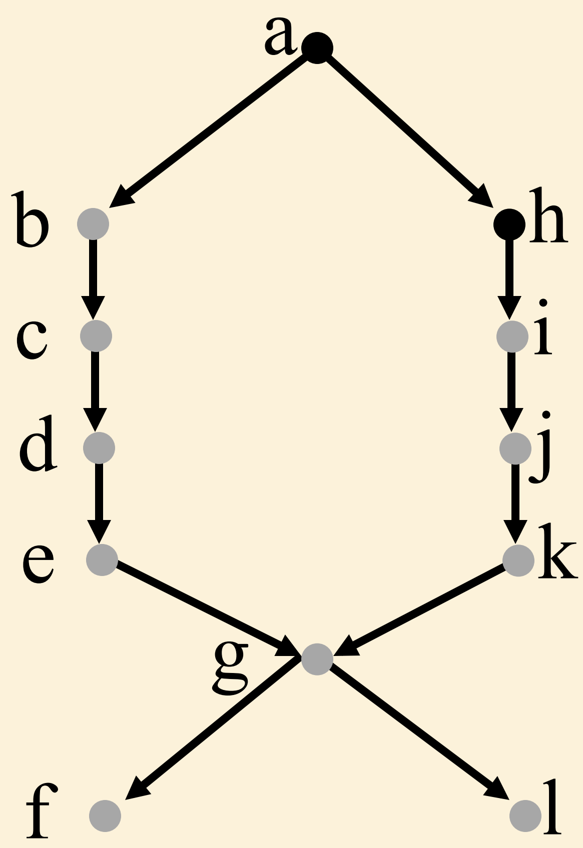


Linear Order:

c,i,j,k,d,e,g,l,f

Linear Order

Alg: DFS



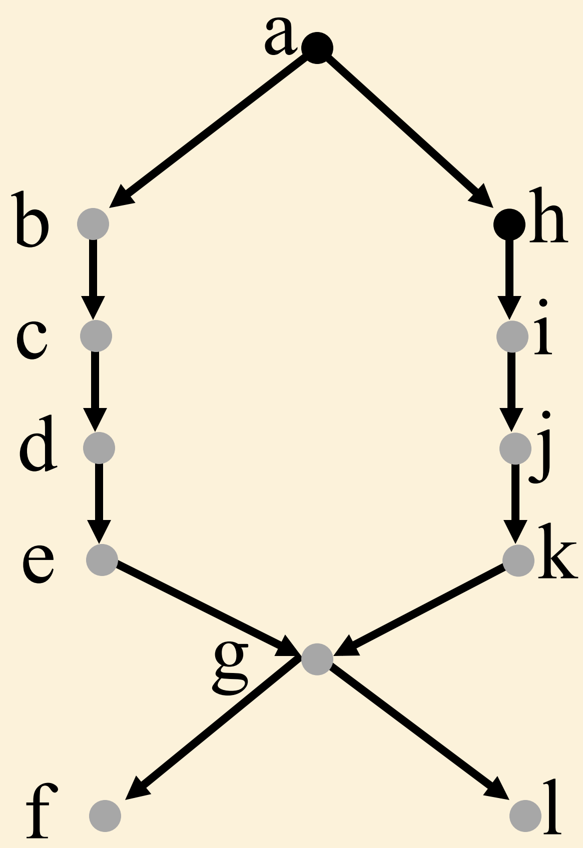
Found
Not Handled
Stack

Linear Order:

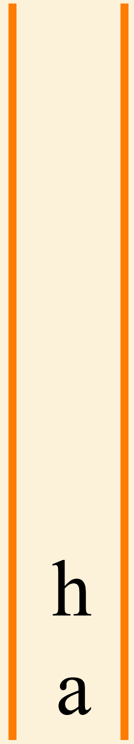
b,c,i,j,k,d,e,g,l,f

Linear Order

Alg: DFS



Found
Not Handled
Stack

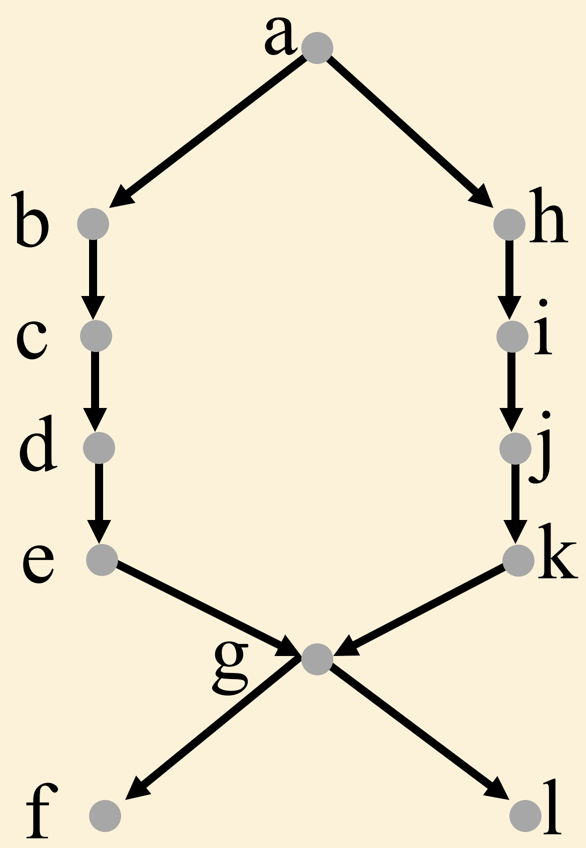


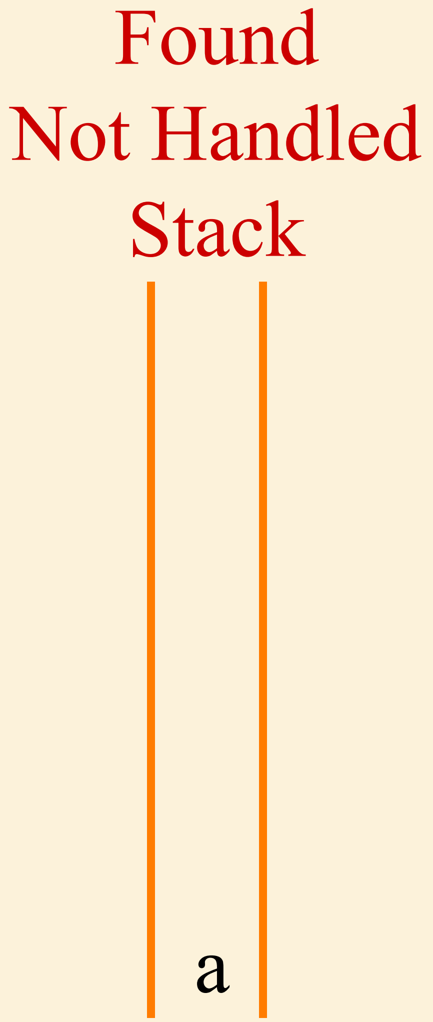
Linear Order:

b,c,i,j,k,d,e,g,l,f

Linear Order

Alg: DFS



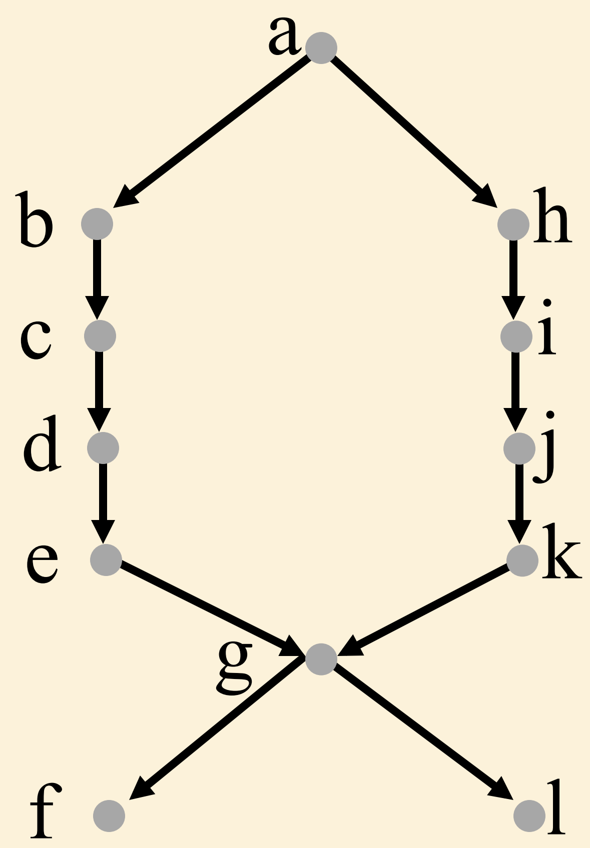


Linear Order:

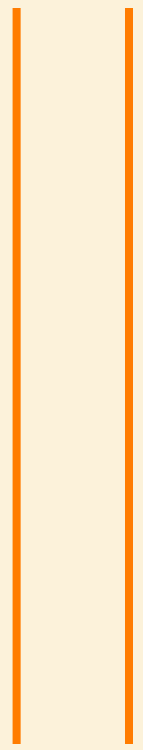
h,b,c,i,j,k,d,e,g,l,f

Linear Order

Alg: DFS



Found
Not Handled
Stack



Linear Order:

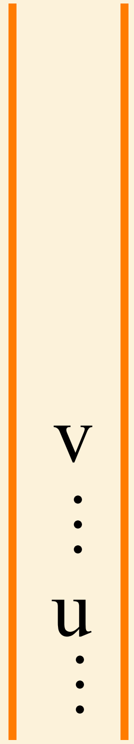
a,h,b,c,i,j,k,d,e,g,l,f **Done!**

Linear Order

Proof: Consider each edge

- Case 1: u goes on stack first before v.
  - Because of edge,
    v goes on before u comes off
  - v comes off before u comes off
  - v goes after u in order. ☺

Found
Not Handled
Stack



$$\begin{tikzcd} u \arrow[r] & v \end{tikzcd}$$

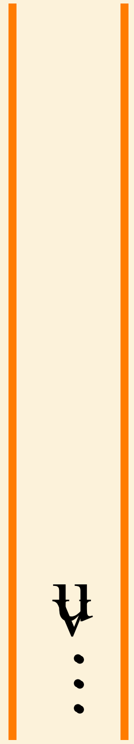
u…v…

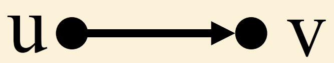
Linear Order

Proof: Consider each edge

- Case 1: u goes on stack first before v.
- Case 2: v goes on stack first before u.
  v comes off before u goes on.
  - v goes after u in order. ☺

Found
Not Handled
Stack





u…v…

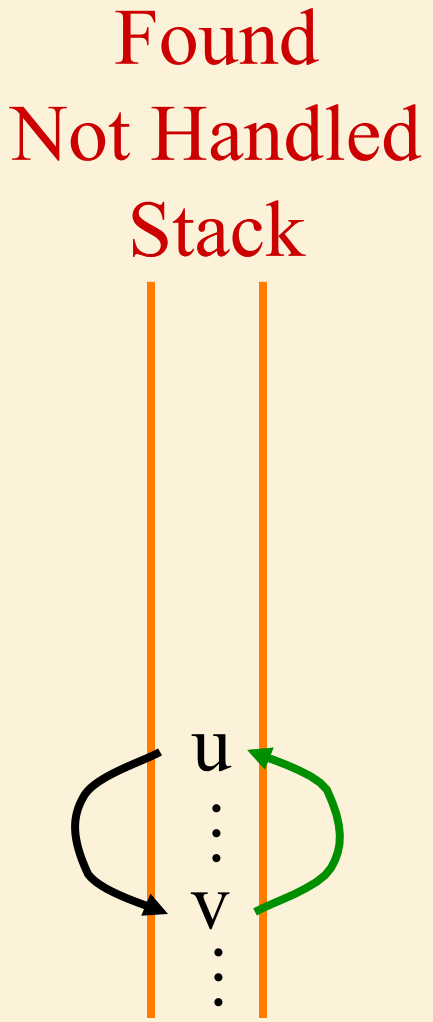
Linear Order

Proof: Consider each edge

- Case 1: u goes on stack first before v.
- Case 2: v goes on stack first before u.
  v comes off before u goes on.

Case 3: v goes on stack first before u.
  u goes on before v comes off.

- Panic: u goes after v in order. ☹
- Cycle means linear order
  is impossible ☺



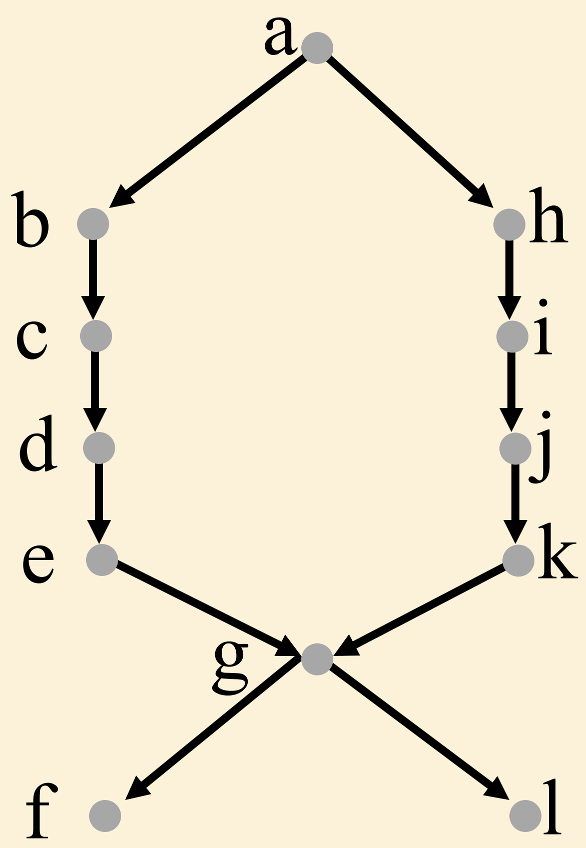
The nodes in the stack form a path starting at s.

$$\begin{tikzcd} u \arrow[r] & v \end{tikzcd}$$

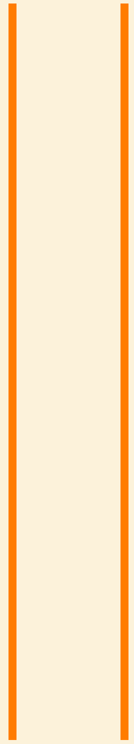
v…u…

Linear Order

Alg: DFS



Found
Not Handled
Stack



Analysis: $\Theta(V+E)$

Linear Order:

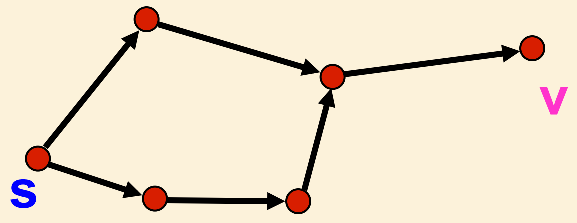
a,h,b,c,i,j,k,d,e,g,l,f **Done!**

Shortest Paths Revisited

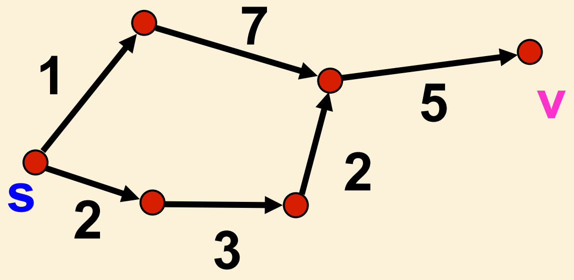
Back to Shortest Path

- BFS finds the **shortest paths** from a source node **s** to every vertex **v** in the graph.
- Here, the **length** of a path is simply the number of edges on the path.
- But what if edges have different ‘costs’?

$\delta(s,v) = 3$



$\delta(s,v) = 12$



Single-Source (Weighted) Shortest Paths

The Problem

- What is the shortest driving route from Toronto to Ottawa? (e.g. MAPQuest, Google Maps)
- Input:

Directed Graph $G = (V, E)$

Edge weights $w : E \to {}^{\circ}$

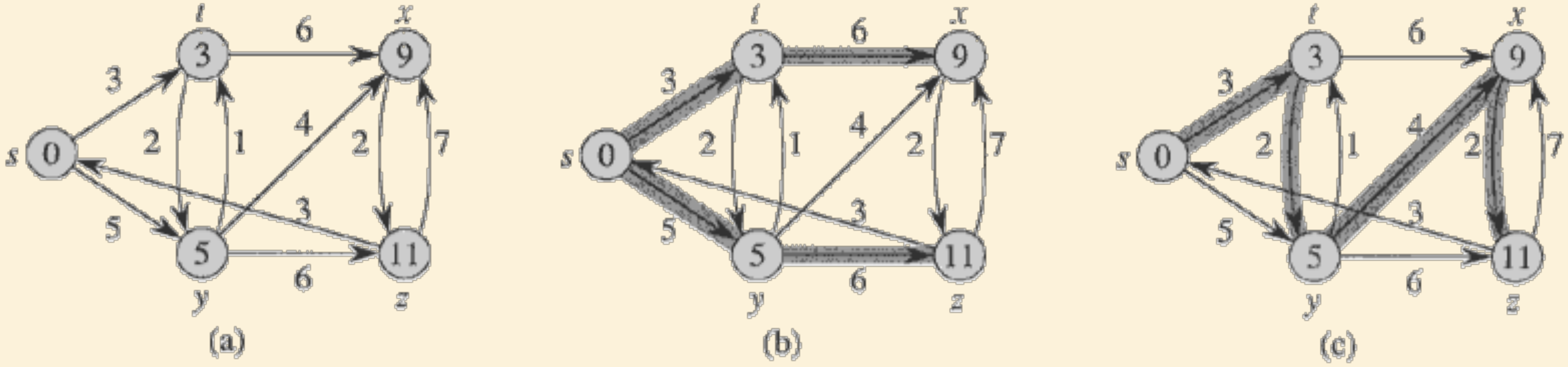
Weight of path $p = < v_0, v_1, \ldots, v_k > \; = \sum_{i=1}^{k} w(v_{i-1}, v_i)$

Shortest-path weight from u to v:

$$\delta(u,v) = \begin{cases} \min\{w(p) : \; u \overset{p}{\to \mathrm{L}} \to v\} & \text{if } \exists \text{ a path } u \to \mathrm{L} \to v, \\ \infty & \text{otherwise.} \end{cases}$$

Shortest path from u to v is any path p such that $w(p) = \delta(u,v)$.

Example



Single-source shortest path search induces a search tree rooted at *s*.

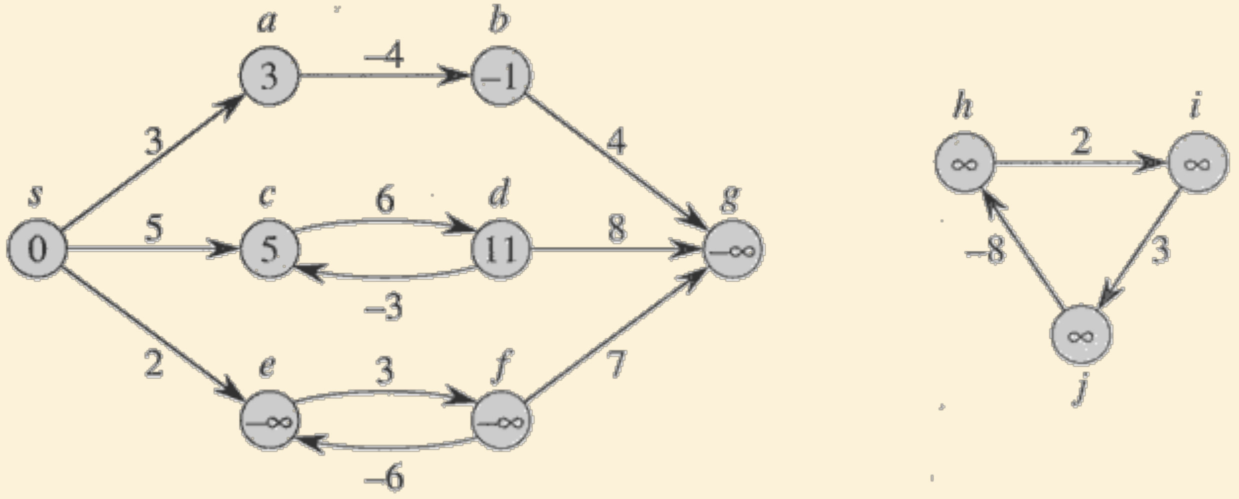
This tree, and hence the paths themselves, are not necessarily unique.

Shortest path variants

- **Single-source shortest-paths problem:** – the shortest path from *s* to each vertex *v*. (e.g. BFS)
- **Single-destination shortest-paths problem:** Find a shortest path to a given ***destination*** vertex *t* from each vertex *v*.
- **Single-pair shortest-path problem:** Find a shortest path from *u* to *v* for given vertices *u* and *v*.
- **All-pairs shortest-paths problem:** Find a shortest path from *u* to *v* for every pair of vertices *u* and *v*.

Negative-weight edges

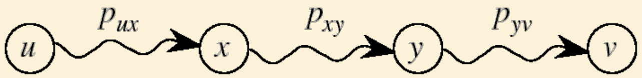
- OK, as long as no negative-weight cycles are reachable from the source.
  - If we have a negative-weight cycle, we can just keep going around it, and get w(s, v) = −∞ for all v on the cycle.
  - But OK if the negative-weight cycle is not reachable from the source.
  - Some algorithms work only if there are no negative-weight edges in the graph.



Optimal substructure

- Lemma: Any subpath of a shortest path is a shortest path
- Proof: Cut and paste.

Suppose this path p is a shortest path from u to v.

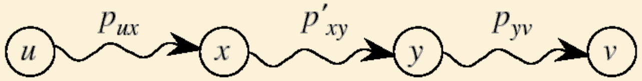


Then $\delta(u,v) = w(p) = w(p_{ux}) + w(p_{xy}) + w(p_{yv})$.

Now suppose there exists a shorter path $x \rightarrow \overset{p'_{xy}}{\mathrm{L}} \rightarrow y$.

Then $w(p'_{xy}) < w(p_{xy})$.

Construct p′:



Then $w(p') = w(p_{ux}) + w(p'_{xy}) + w(p_{yv}) < w(p_{ux}) + w(p_{xy}) + w(p_{yv}) = w(p)$.

So p wasn't a shortest path after all!

Cycles

- Shortest paths can’t contain cycles:
  - Already ruled out negative-weight cycles.
  - Positive-weight: we can get a shorter path by omitting the cycle.
  - Zero-weight: no reason to use them → assume that our solutions won’t use them.

Output of a single-source shortest-path algorithm

- For each vertex v in V:
  - d[v] = δ(s, v).
    - Initially, d[v]=∞.
    - Reduce as algorithm progresses.
      But always maintain d[v] ≥ δ(s, v).
    - Call d[v] a shortest-path estimate.
  - π[v] = predecessor of v on a shortest path from s.
    - If no predecessor, π[v] = NIL.
    - π induces a tree — **shortest-path tree**.

Initialization

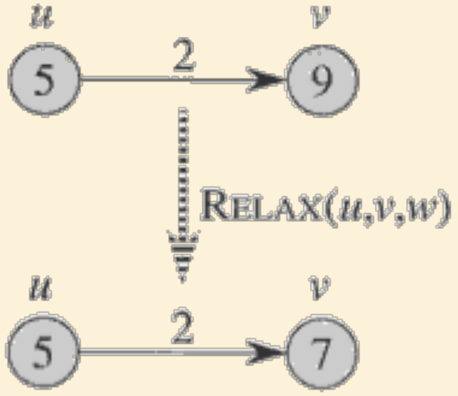
- All shortest-paths algorithms start with the same initialization:

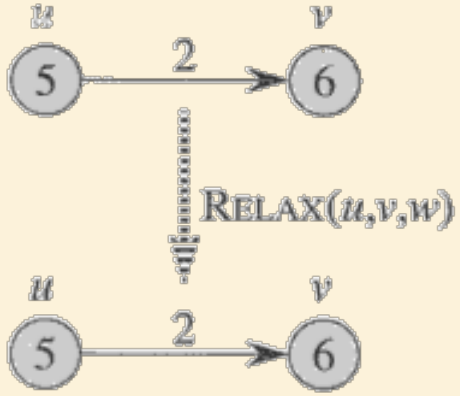
```
INIT-SINGLE-SOURCE(V, s)
for each v in V
    do d[v] ← ∞
       π[v] ← NIL
d[s] ← 0
```

Relaxing an edge

- Can we improve shortest-path estimate for v by going through u and taking (u,v)?

```
RELAX(u, v,w)
    if $d[v] > d[u] + w(u, v)$ then
        $d[v] \leftarrow d[u] + w(u, v)$
        $\pi[v] \leftarrow u$
```





General single-source shortest-path strategy

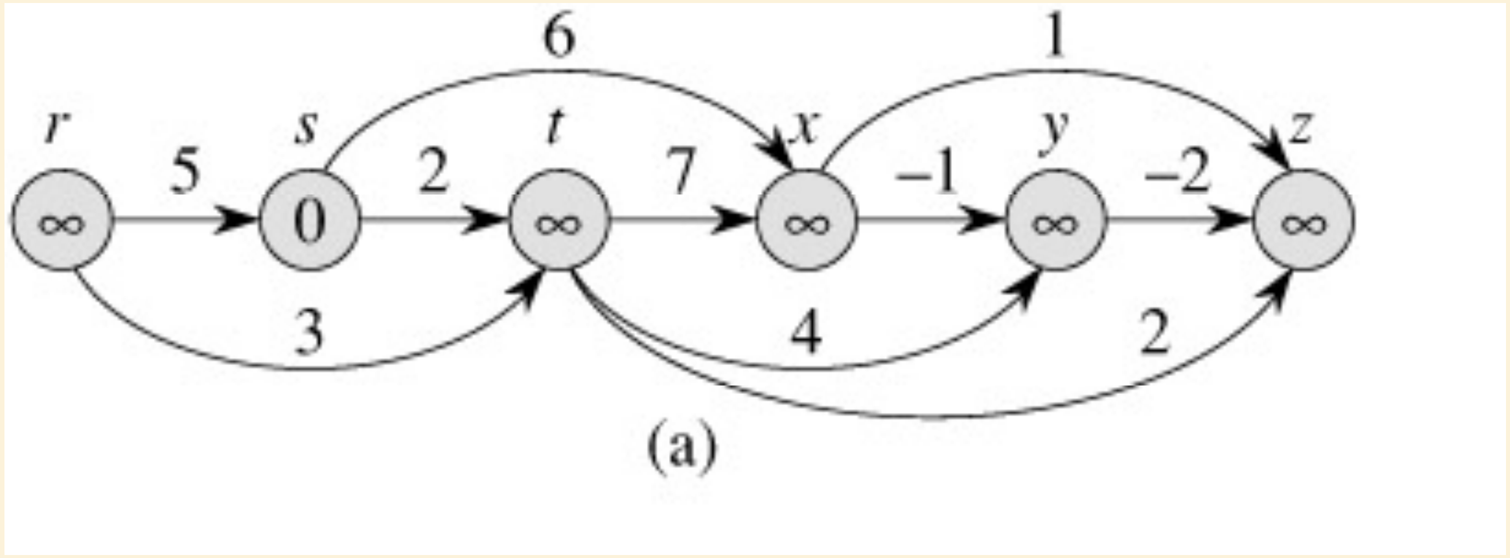
1. Start by calling INIT-SINGLE-SOURCE
2. Relax Edges

Algorithms differ in the order in which edges are taken and

how many times each edge is relaxed.

Example: Single-source shortest paths in a directed acyclic graph (DAG)

- Since graph is a DAG, we are guaranteed no negative-weight cycles.

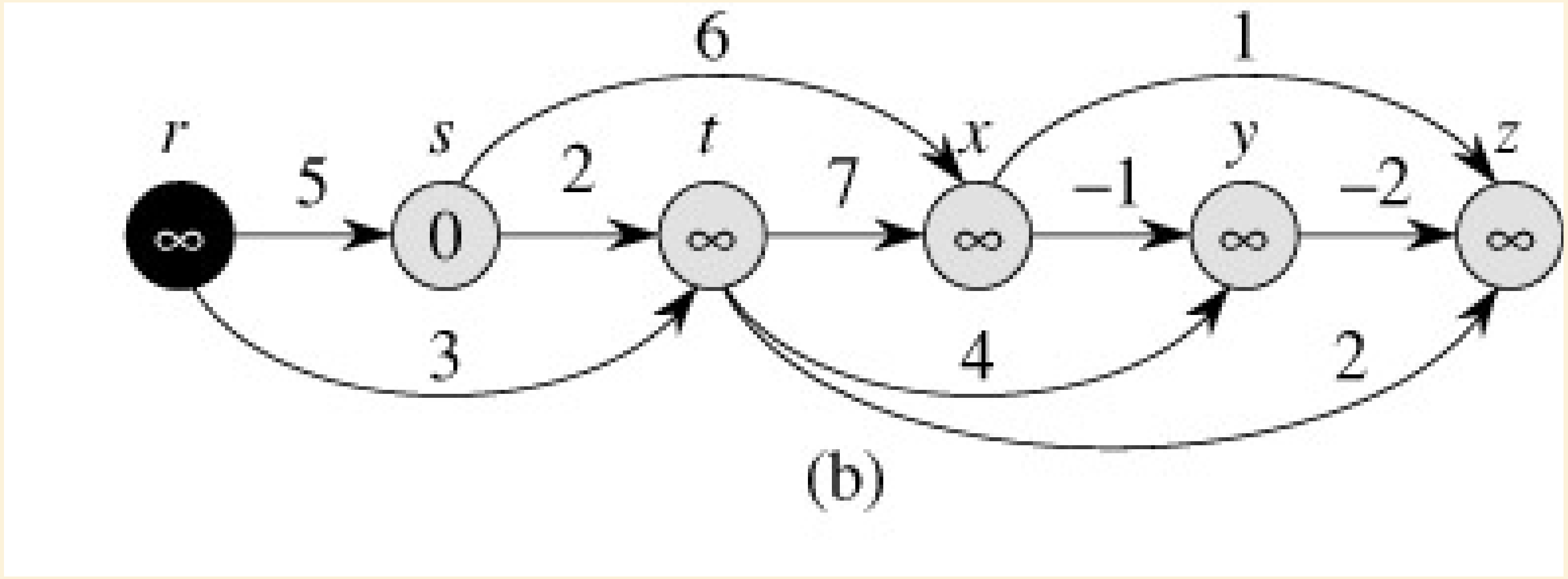


Algorithm

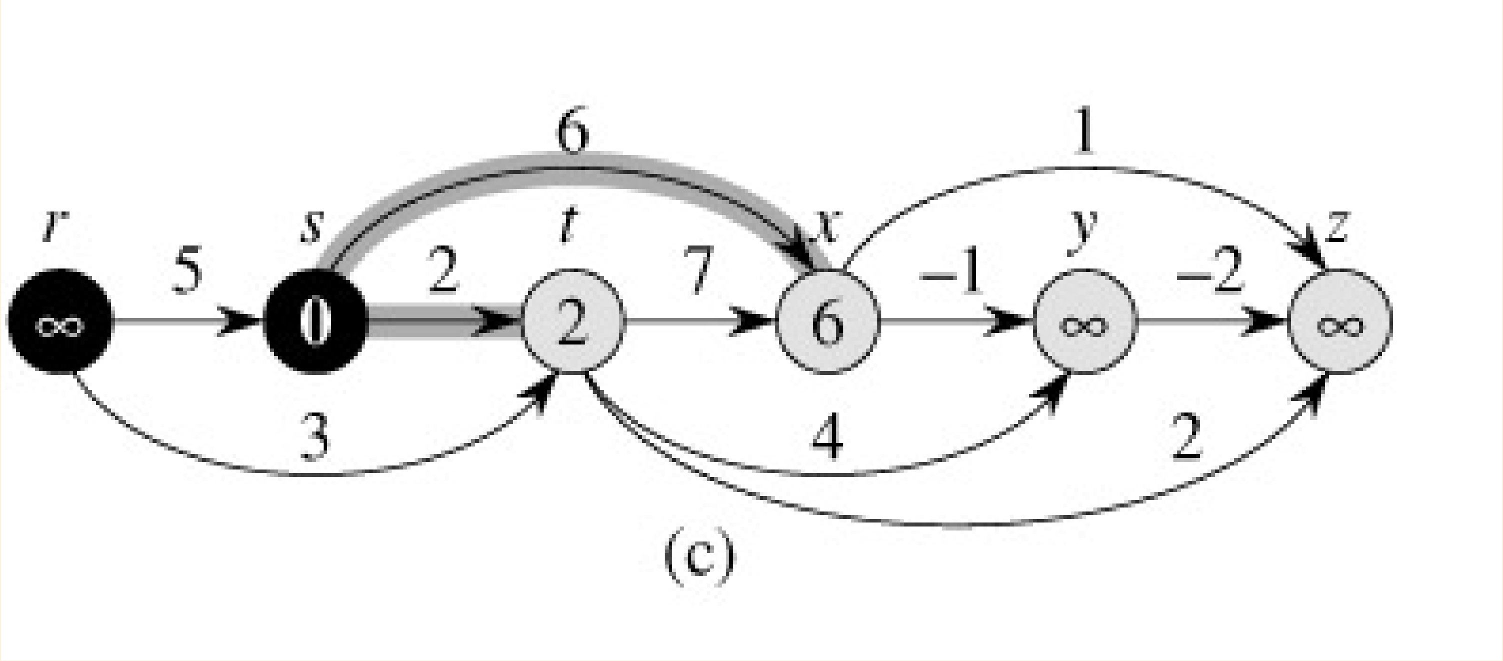
```
DAG-SHORTEST-PATHS$(G, w, s)$
1  topologically sort the vertices of $G$
2  INITIALIZE-SINGLE-SOURCE$(G, s)$
3  for each vertex $u$, taken in topologically sorted order
4       do for each vertex $v \in Adj[u]$
5              do RELAX$(u, v, w)$
```

Time: $\Theta(V + E)$

Example

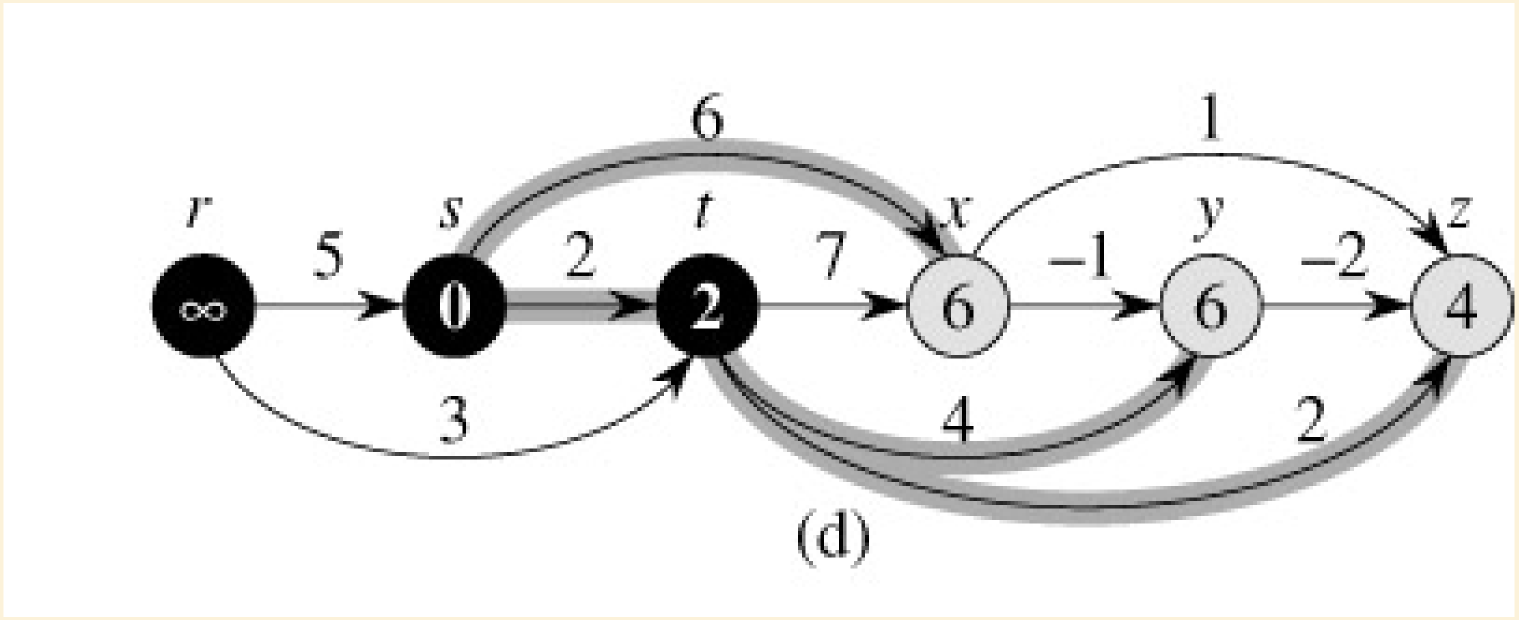


Example



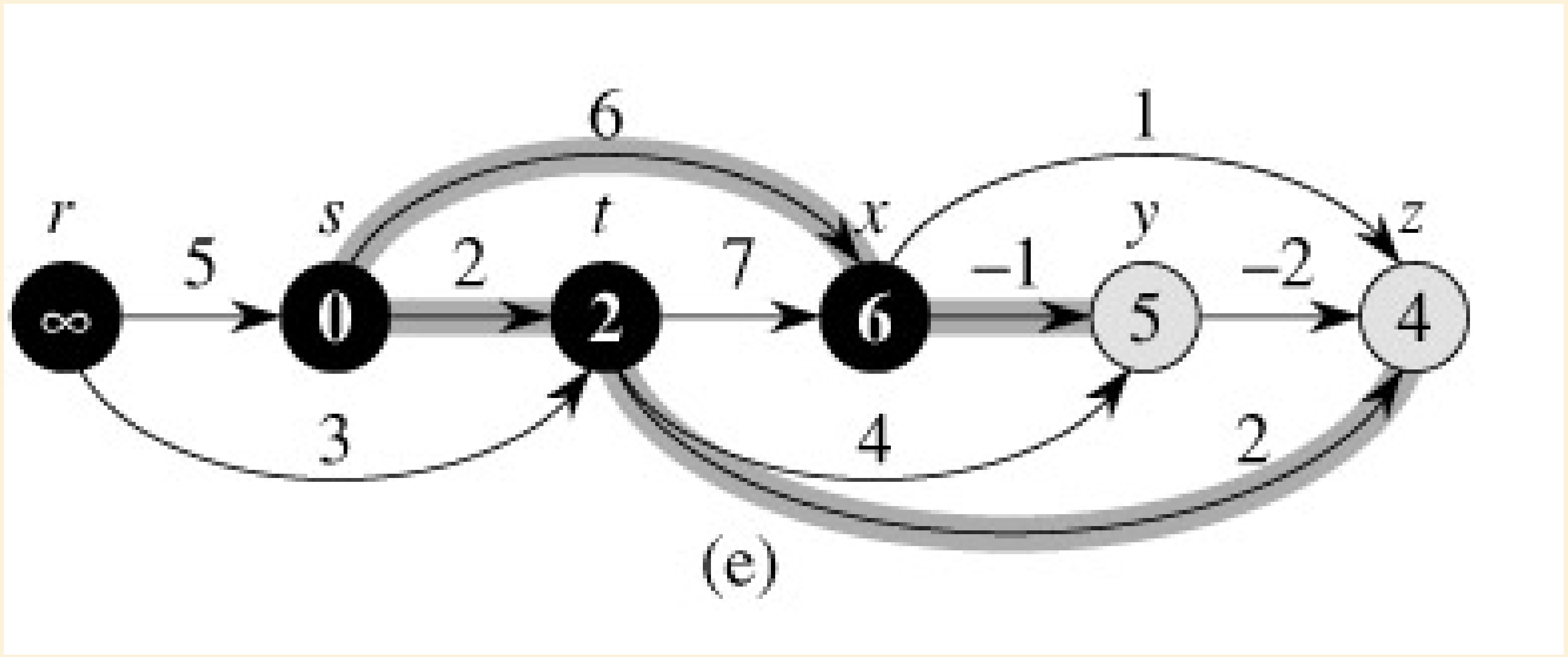
(c)

Example



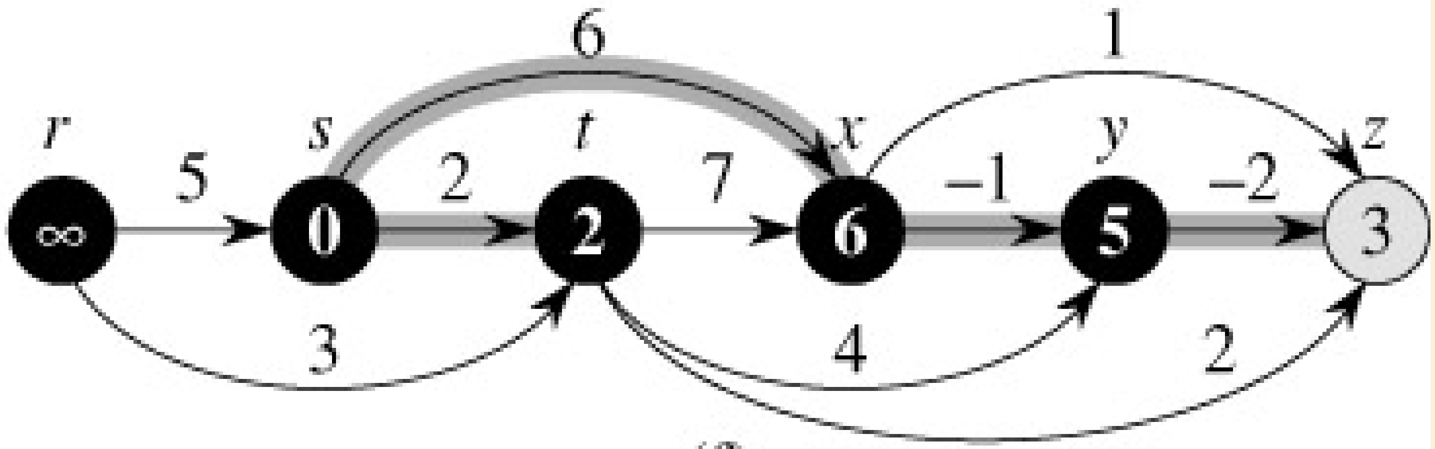
(d)

Example



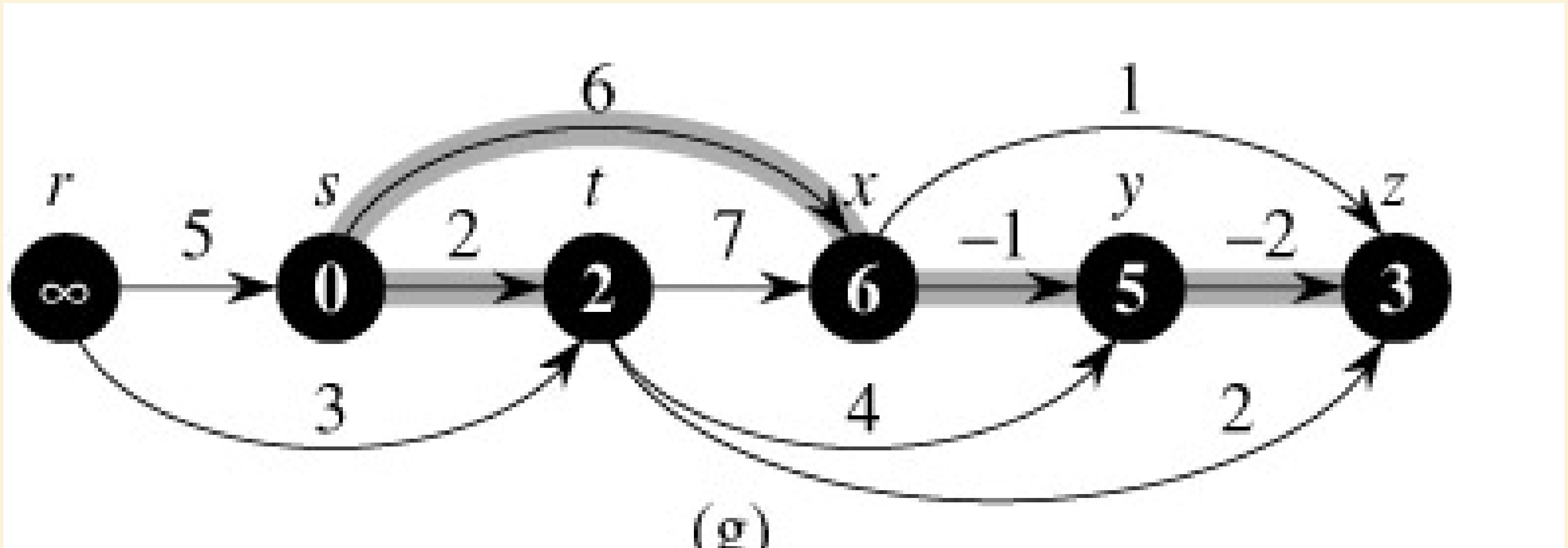
(e)

Example



(f)

Example



(g)

Correctness: Path relaxation property (Lemma 24.15)

Let $p = < v_0,\ v_1,\ .\ .\ .\ .\ ,\ v_k >$ be a shortest path from $s = v_0$ *to* v_k.

If we relax, in order, (v_0,v_1), (v_1,v_2), ... , (v_{k-1},v_k),

even intermixed with other relaxations,

then $d[v_k] = \delta(s,\ v_k)$.

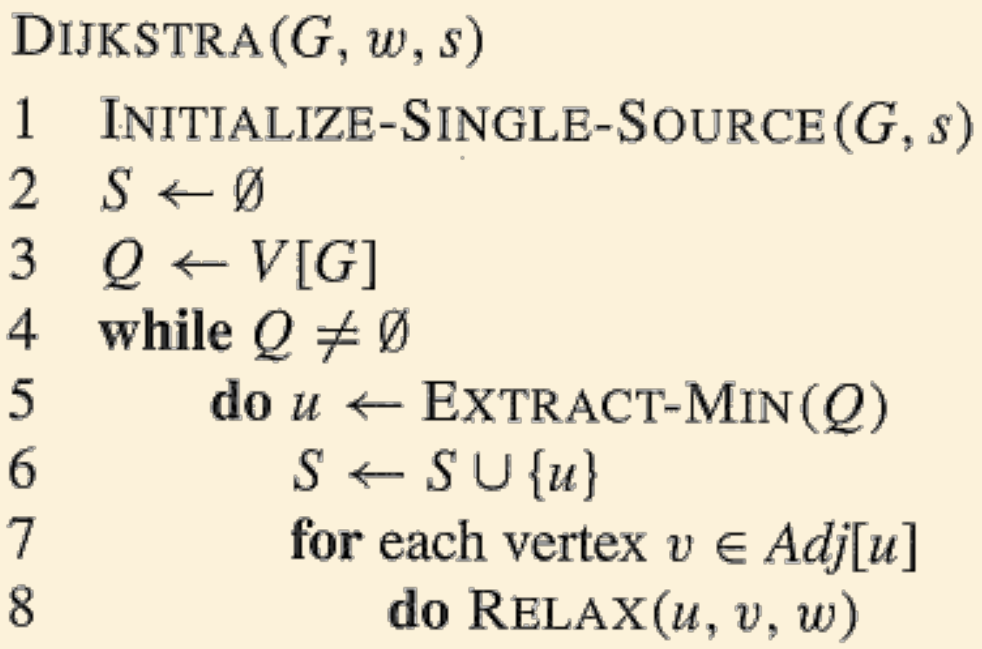
Correctness of DAG Shortest Path Algorithm

- Because we process vertices in topologically sorted order, edges of *any* path are relaxed in order of appearance in the path.
  - →Edges on any shortest path are relaxed in order.
  - →By path-relaxation property, correct.

Example: Dijkstra's algorithm

- Applies to general weighted directed graph (may contain cycles).
- But weights must be non-negative.
- Essentially a weighted version of BFS.
  - Instead of a FIFO queue, uses a priority queue.
  - Keys are shortest-path weights (d[v]).
- Maintain 2 sets of vertices:
  - S = vertices whose final shortest-path weights are determined.
  - Q = priority queue = V-S.

Dijkstra's algorithm

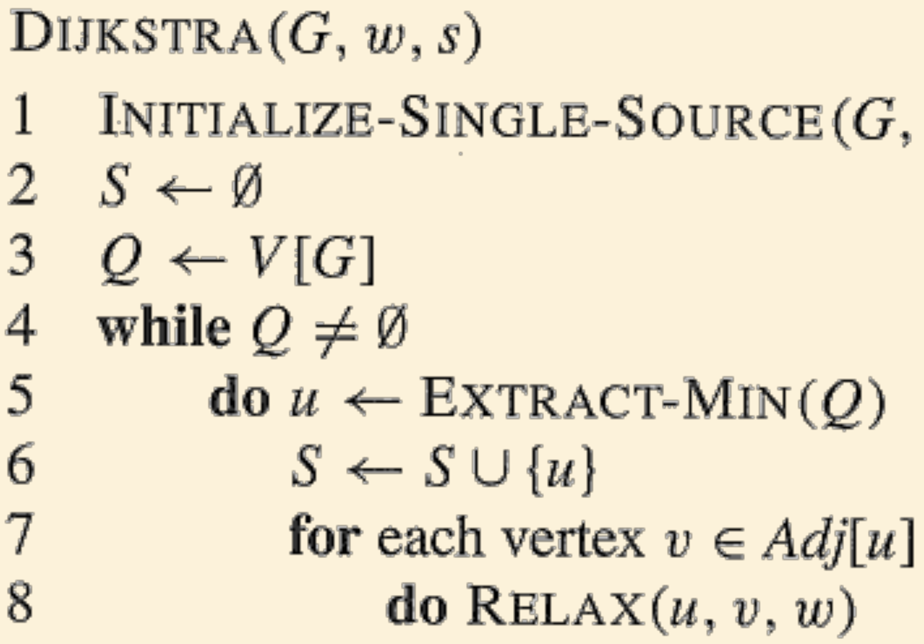


```
DIJKSTRA($G, w, s$)
INITIALIZE-SINGLE-SOURCE($G, s$)
$S \leftarrow \emptyset$
$Q \leftarrow V[G]$
while $Q \neq \emptyset$
    do $u \leftarrow$ EXTRACT-MIN($Q$)
       $S \leftarrow S \cup \{u\}$
       for each vertex $v \in Adj[u]$
           do RELAX($u, v, w$)
```

- Dijkstra's algorithm can be viewed as greedy, since it always chooses the "lightest" vertex in V – S to add to S.

Dijkstra's algorithm: *Analysis*

- *Analysis:*
  - Using minheap, queue operations takes *O(logV)* time



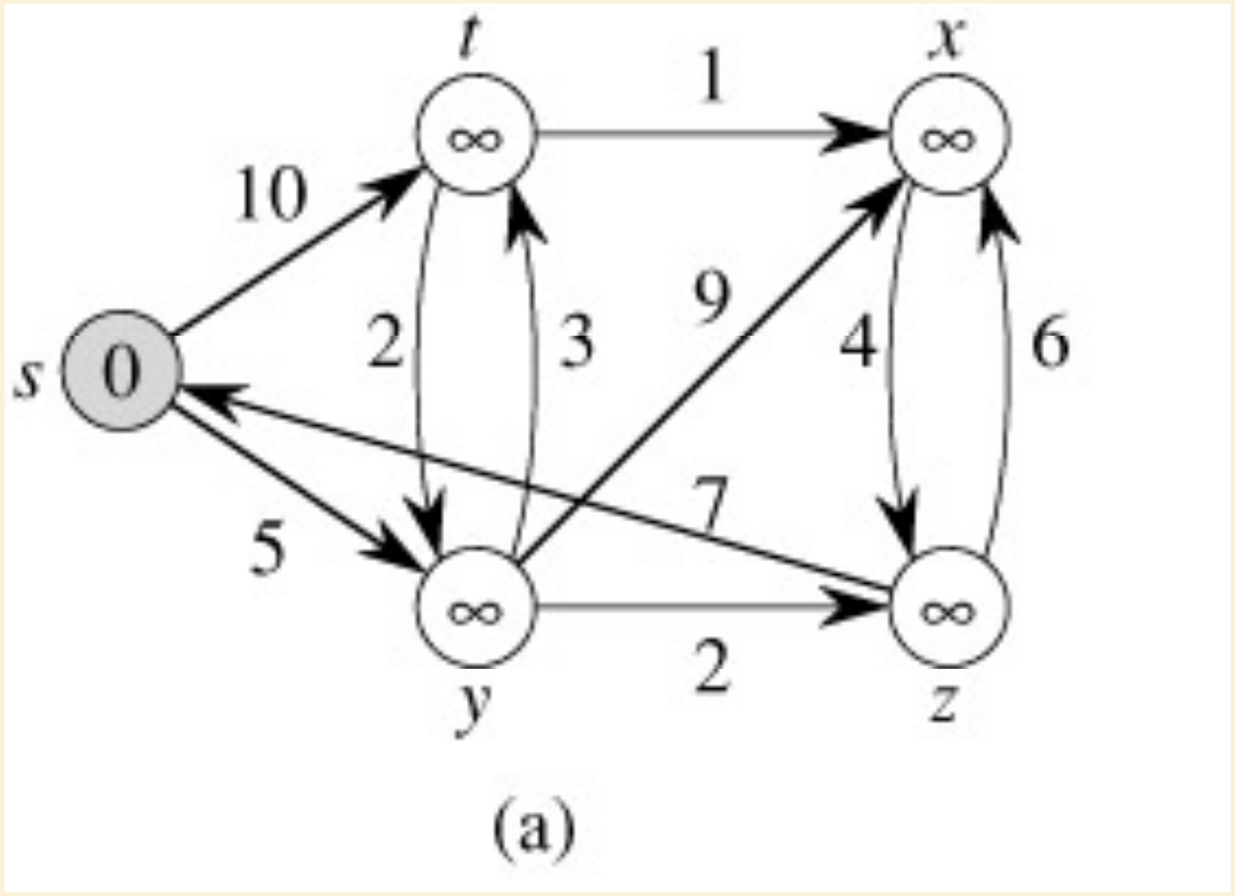
```
DIJKSTRA($G$, $w$, $s$)
1  INITIALIZE-SINGLE-SOURCE($G$, $s$)      $O(V)$
2  $S \leftarrow \emptyset$
3  $Q \leftarrow V[G]$
4  while $Q \neq \emptyset$
5      do $u \leftarrow$ EXTRACT-MIN($Q$)       $O(\log V) \times O(V)$ iterations
6         $S \leftarrow S \cup \{u\}$
7         for each vertex $v \in Adj[u]$
8             do RELAX($u$, $v$, $w$)       $O(\log V) \times O(E)$ iterations
```

$\rightarrow$ Running Time is $O(E \log V)$

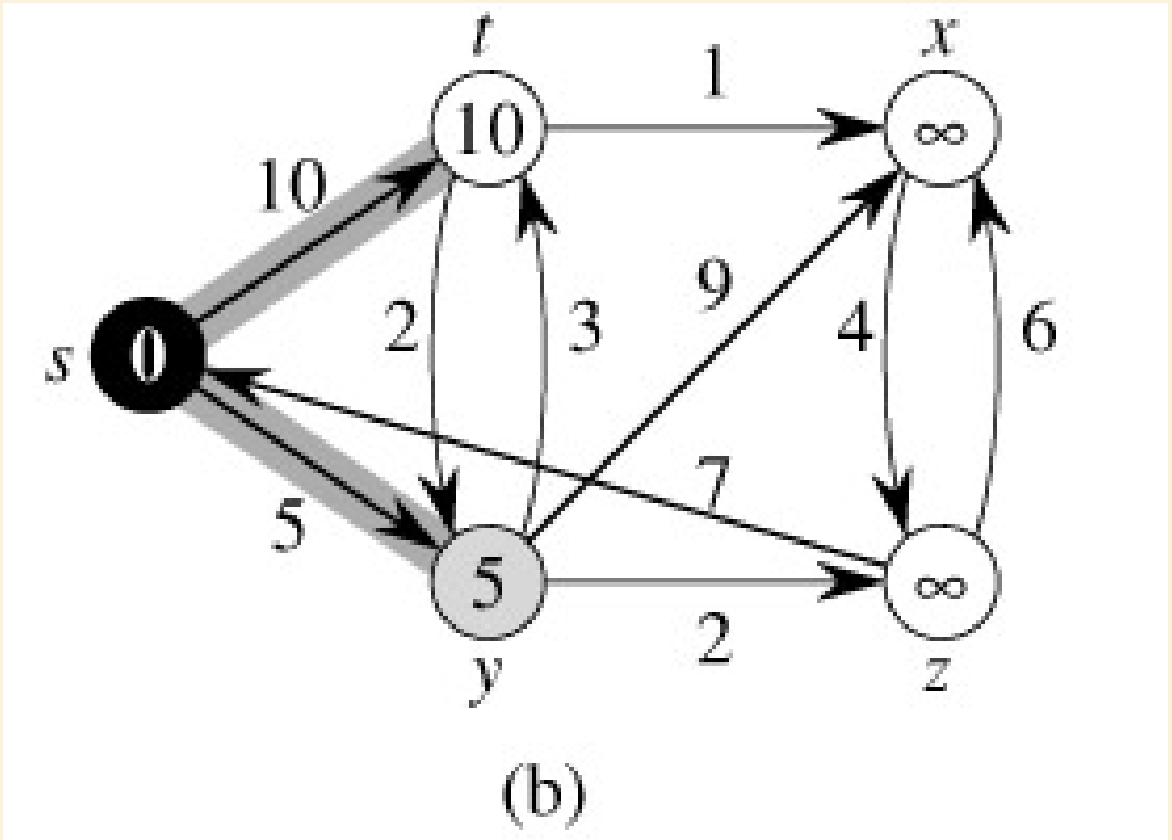
Example

Key:

White	⇔	Not Found
Grey	⇔	Handling
Black	⇔	Handled

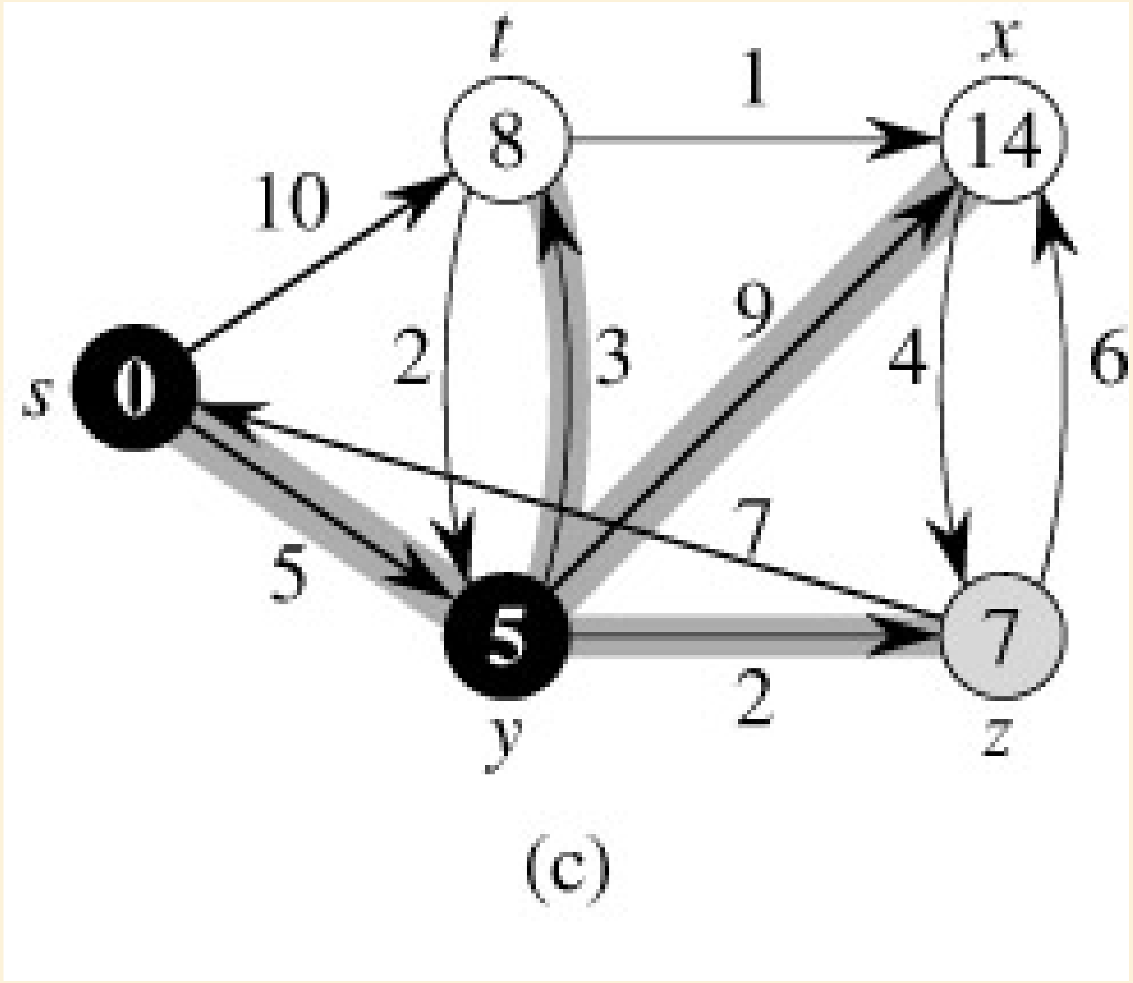


Example



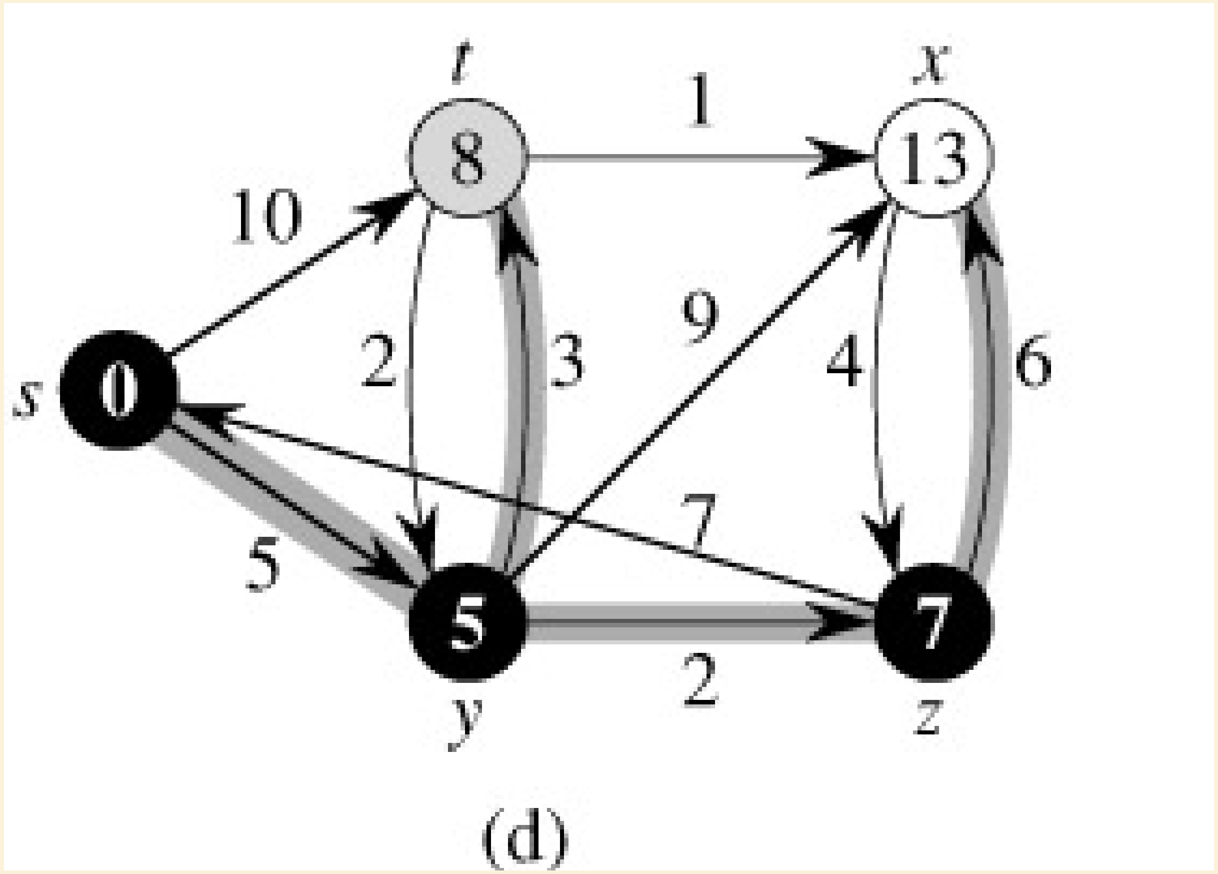
(b)

Example

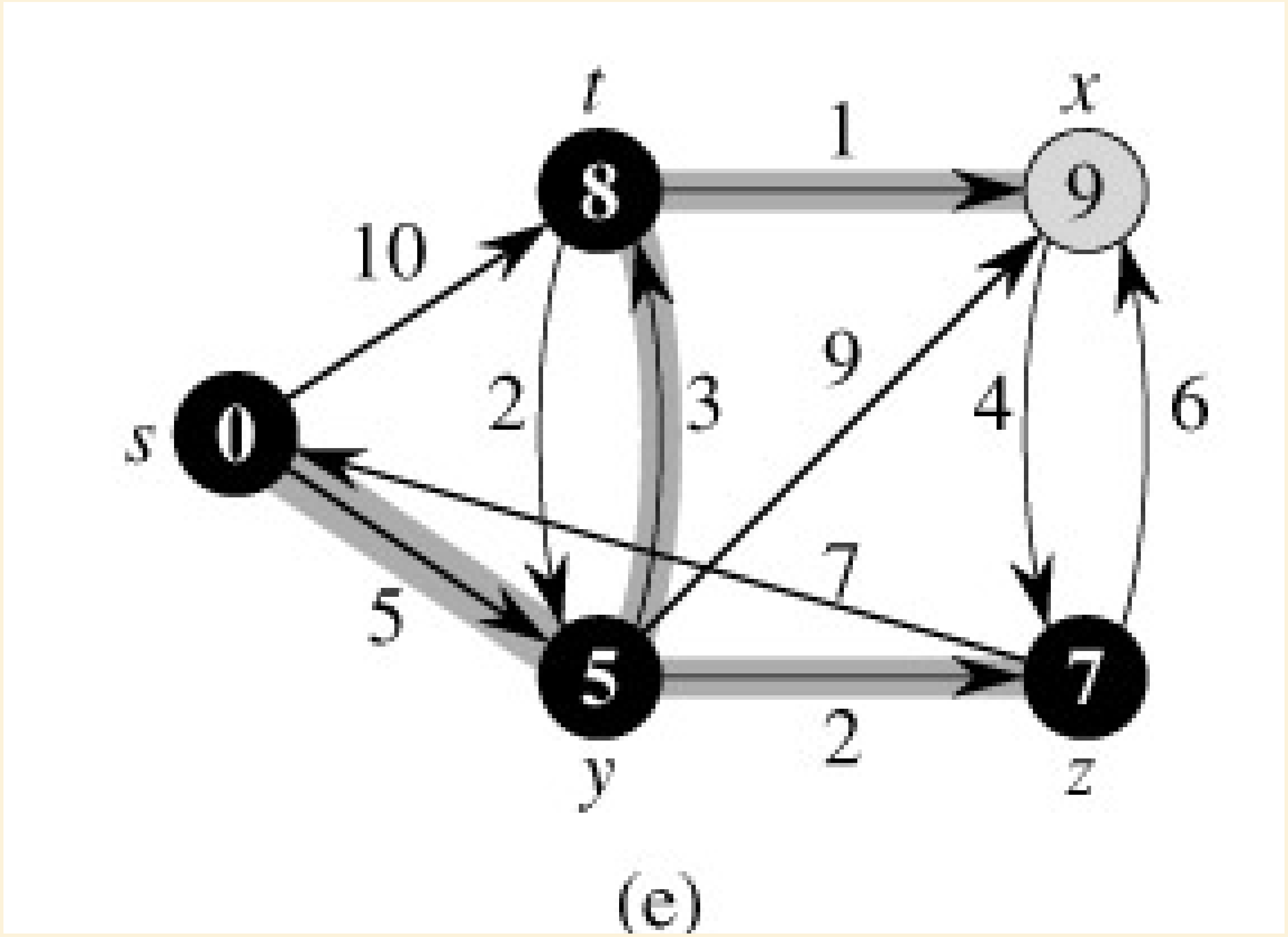


(c)

Example

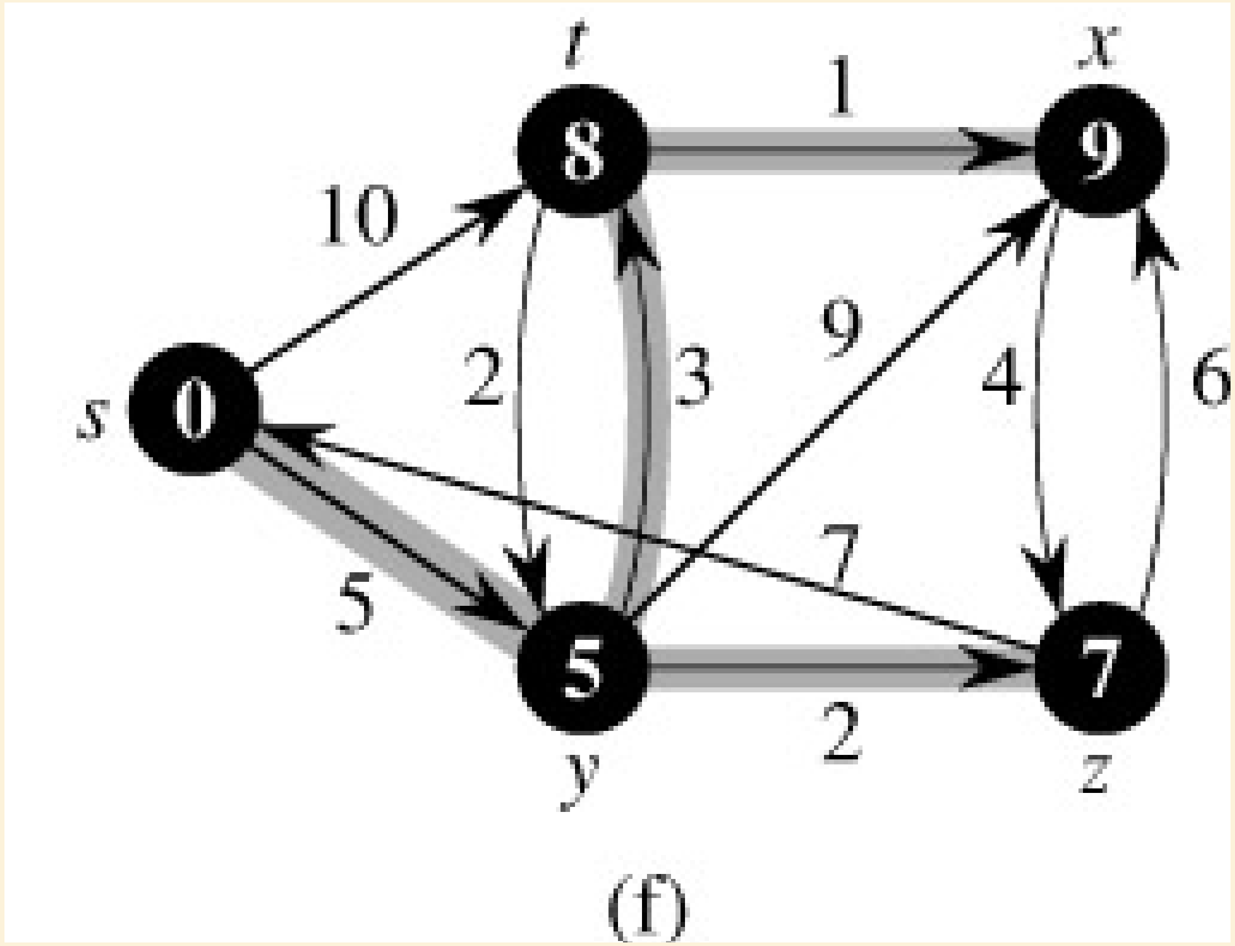


Example



(e)

Example



(f)

Correctness of Dijkstra's algorithm

```
DIJKSTRA($G, w, s$)
INITIALIZE-SINGLE-SOURCE($G, s$)
$S \leftarrow \emptyset$
$Q \leftarrow V[G]$
while $Q \neq \emptyset$
    do $u \leftarrow$ EXTRACT-MIN($Q$)
        $S \leftarrow S \cup \{u\}$
        for each vertex $v \in Adj[u]$
            do RELAX($u, v, w$)
```

- **Loop invariant:** d[v] = δ(s, v) for all v in S.
  - Initialization: Initially, S is empty, so trivially true.
  - Termination: At end, Q is empty → S = V → d[v] = δ(s, v) for all v in V.
  - Maintenance:
    - Need to show that
      - d[u] = δ(s, u) when u is added to S in each iteration.
      - d[u] does not change once u is added to S.

Correctness of Dijkstra's Algorithm: Upper Bound Property

- Upper Bound Property:

  1. $d[v] \geq \delta(s,v) \forall v \in V$
  2. Once $d[v] = \delta(s,v)$, it doesn't change

- Proof:

  By induction.

  Base Case: $d[v] \geq \delta(s,v) \forall v \in V$ immediately after initialization, since

  $d[s] = 0 = \delta(s,s)$

  $d[v] = \infty \forall v \neq s$

  Inductive Step:

  Suppose $d[x] \geq \delta(s,x) \forall x \in V$

  Suppose we relax edge (u,v).

  If $d[v]$ changes, then $d[v] = d[u] + w(u,v)$

  $$\geq \delta(s,u) + w(u,v)$$

  $$\geq \delta(s,v)$$

Correctness of Dijkstra's Algorithm

Claim: When u is added to S, $d[u] = \delta(s,u)$

Proof by Contradiction: Let u be the first vertex added to S such that $d[u] \neq \delta(s,u)$ when u is added.

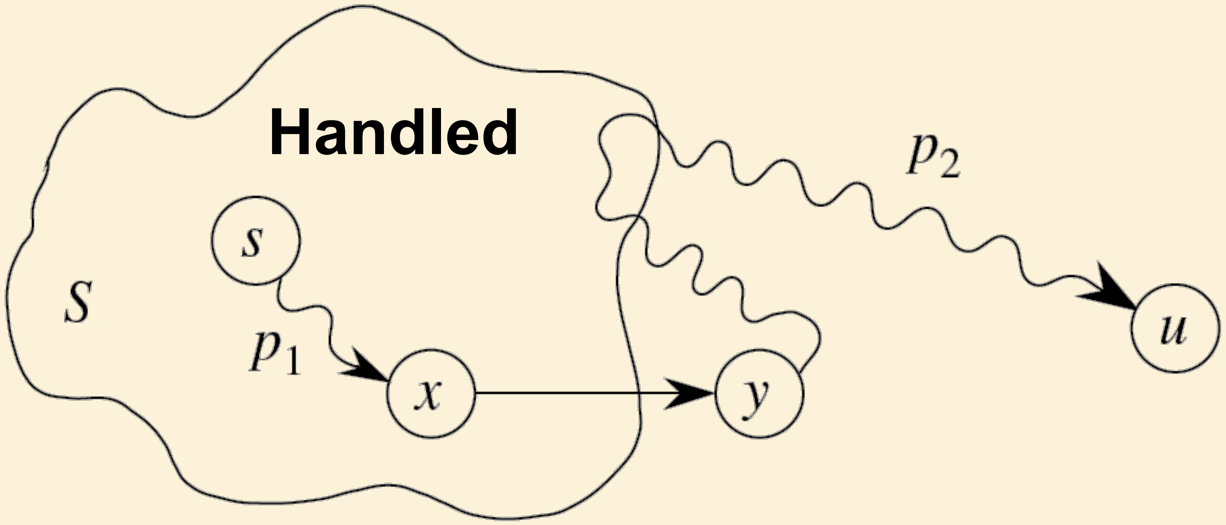
Let y be first vertex in $V - S$ on shortest path to u
Let x be the predecessor of y on the shortest path to u

Claim: $d[y] = \delta(s,y)$ when u is added to S.

Proof:

$d[x] = \delta(s,x)$, since $x \in S$.

(x,y) was relaxed when x was added to S $\rightarrow d[y] = \delta(s,x) + w(x,y) = \delta(s,y)$



Correctness of Dijkstra's Algorithm

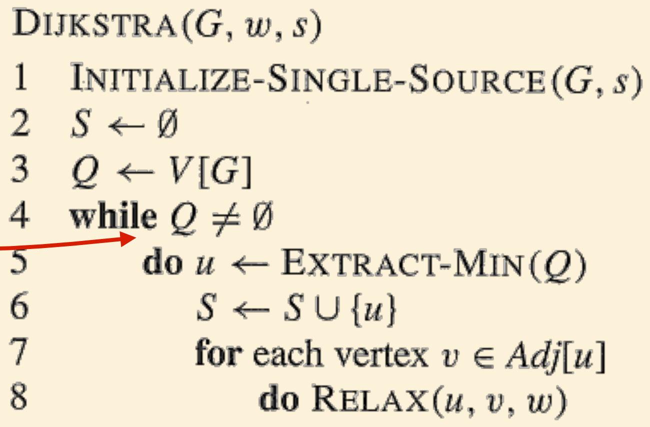
Thus $d[y] = \delta(s,y)$ when u is added to S.

$\rightarrow d[y] = \delta(s,y) \le \delta(s,u) \le d[u]$ (upper bound property)

But $d[u] \le d[y]$ when u added to S

Thus $d[y] = \delta(s,y) = \delta(s,u) = d[u]$!

Thus when u is added to S, $d[u] = \delta(s,u)$

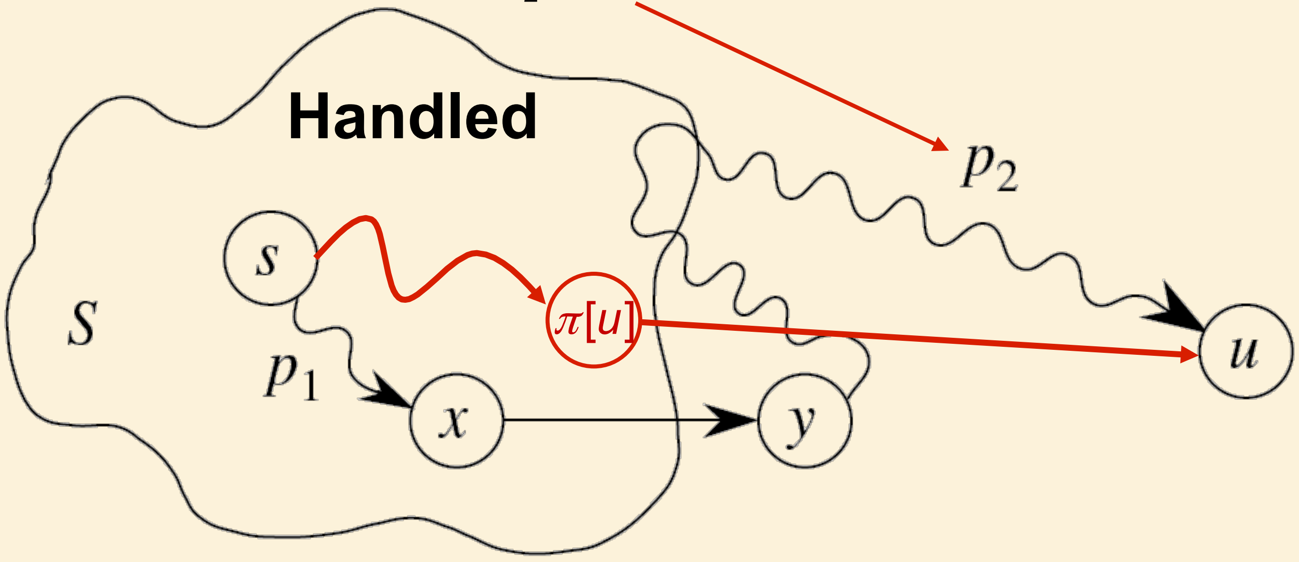


```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S ← ∅
3  Q ← V[G]
4  while Q ≠ ∅
5      do u ← EXTRACT-MIN(Q)
6         S ← S ∪ {u}
7         for each vertex v ∈ Adj[u]
8             do RELAX(u, v, w)
```

Consequences:

There is a shortest path to u such that the predecessor of u $\pi[u] \in S$ when u is added to S.

The path through y can only be a shortest path if $w[p_2] = 0$.



Correctness of Dijkstra's algorithm

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S ← ∅
3  Q ← V[G]
4  while Q ≠ ∅
5        do u ← EXTRACT-MIN(Q)
6           S ← S ∪ {u}
7           for each vertex v ∈ Adj[u]
8               do RELAX(u, v, w)
```

Relax(u,v,w) can only decrease $d[v]$.

By the upper bound property, $d[v] \geq \delta(s,v)$.

Thus once $d[v] = \delta(s,v)$, it will not be changed.

- **Loop invariant:** d[v] = δ(s, v) for all v in S.
  - Maintenance:
    - Need to show that
      - d[u] = δ(s, u) when u is added to S in each iteration. ✓
      - d[u] does not change once u is added to S. ?