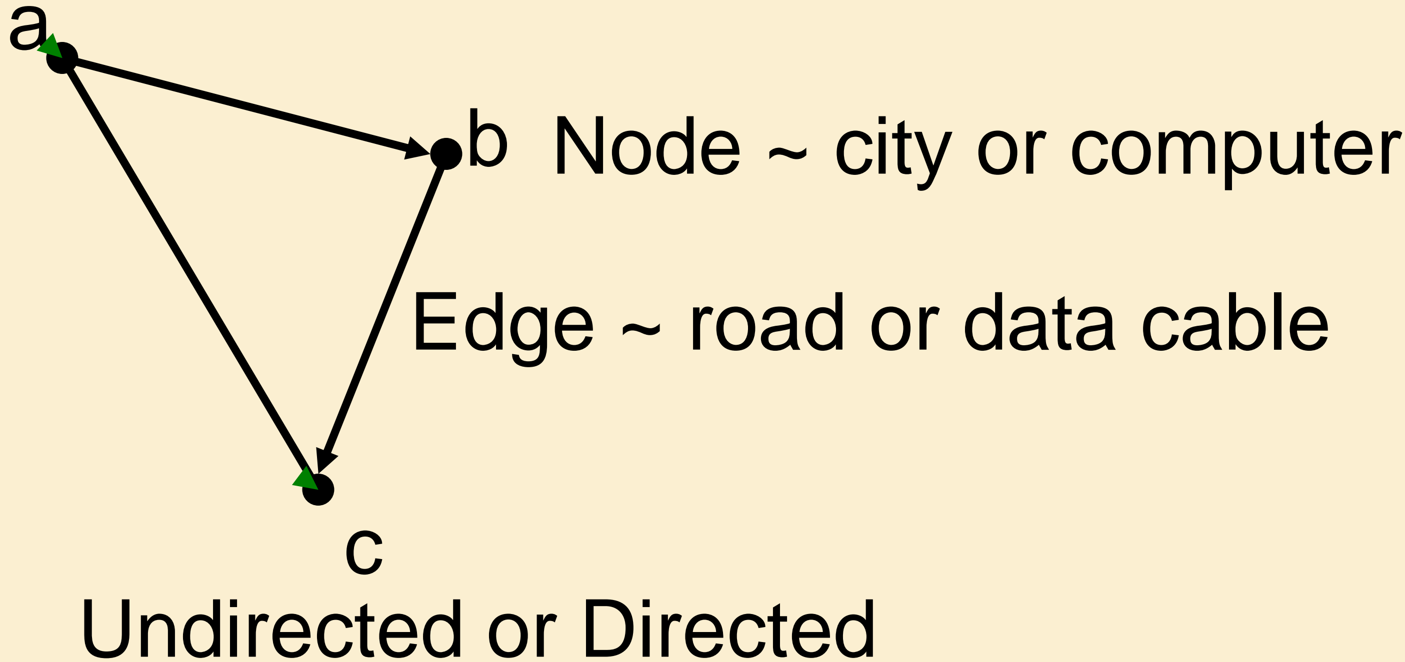
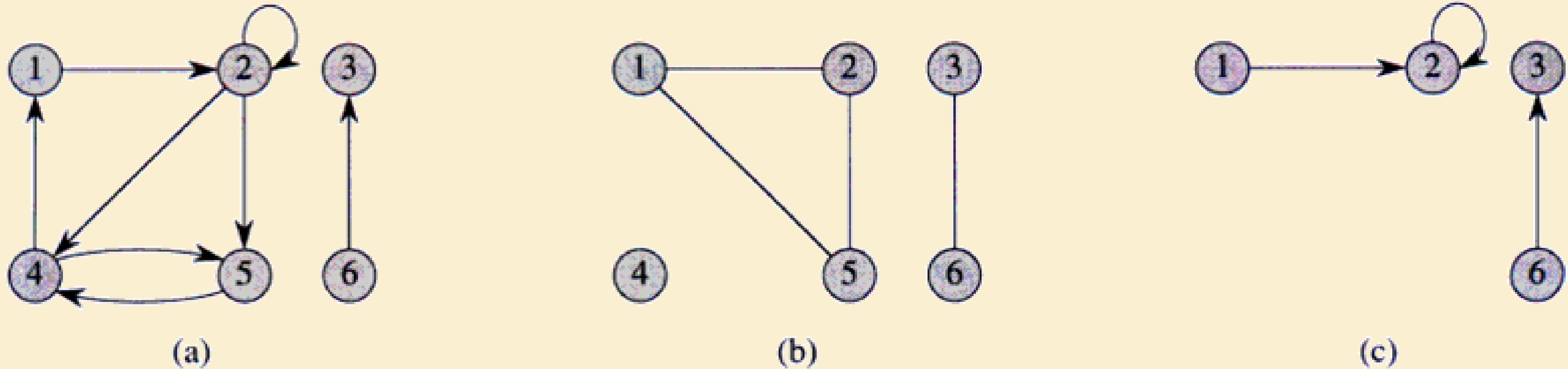
Graph

الدكتور
اثير العاني



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

Directed and Undirected Graphs

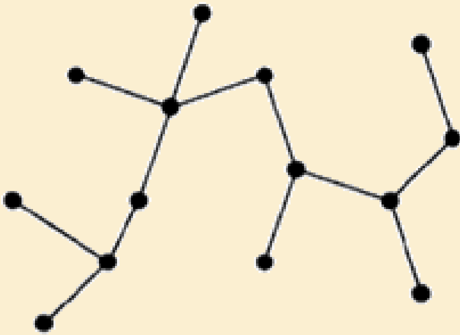


(a) A directed graph $G = (V, E)$, where $V = \{1,2,3,4,5,6\}$ and $E = \{(1,2), (2,2), (2,4), (2,5), (4,1), (4,5), (5,4), (6,3)\}$. The edge (2,2) is a self-loop.

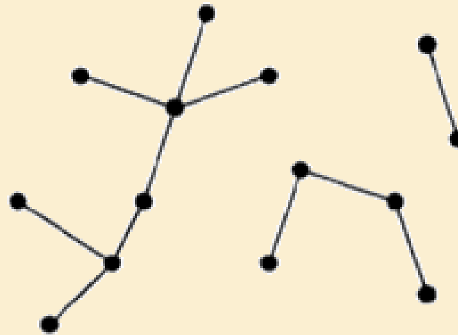
(b) An undirected graph $G = (V,E)$, where $V = \{1,2,3,4,5,6\}$ and $E = \{(1,2), (1,5), (2,5), (3,6)\}$. The vertex 4 is isolated.

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

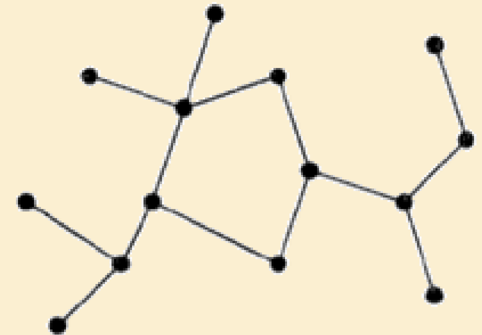
Trees



Tree



Forest



Graph with Cycle

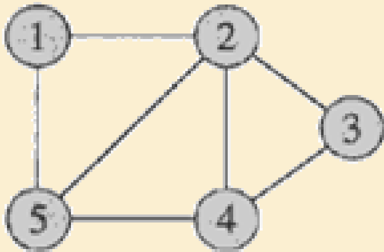
A tree is a connected, acyclic, undirected graph.

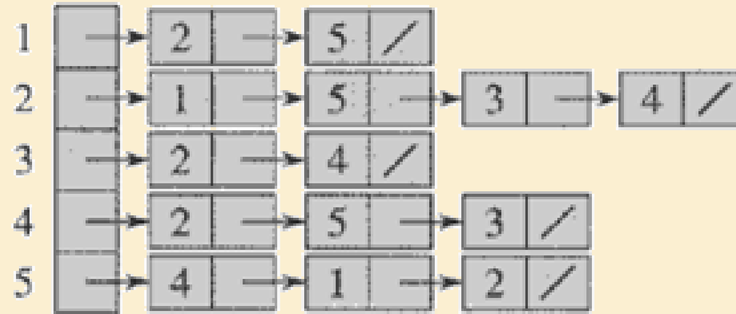
A forest is a set of trees (not necessarily connected)

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

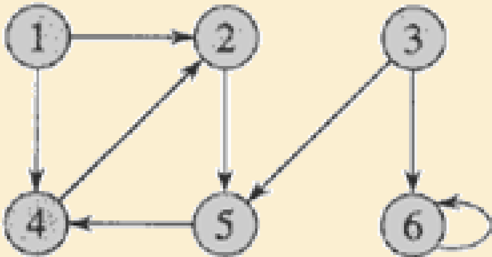


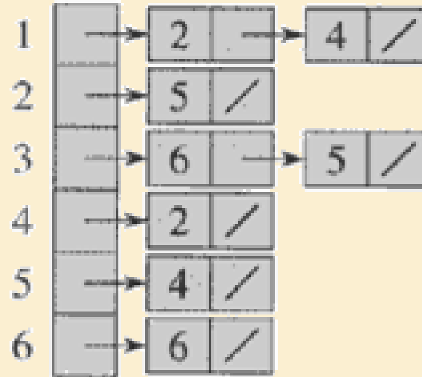


	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

	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

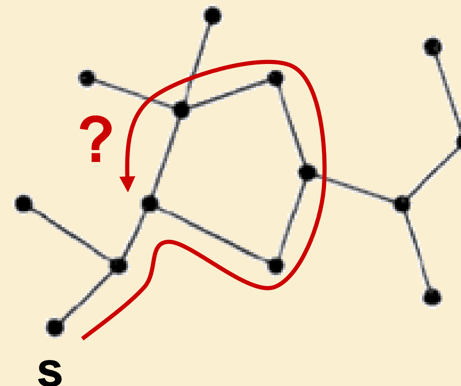
	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Adjacency Matrix

	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 $(u,v) \in E$:	$\theta(\text{degree}(u))$	$\theta(1)$

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:

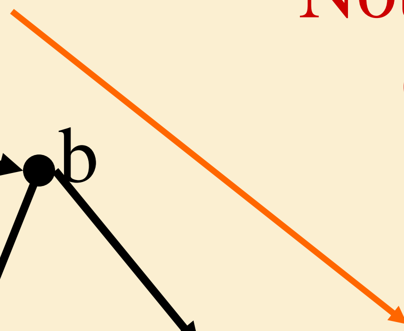
$d[v] =$ shortest path distance $\delta(s, v)$ from s to v, $\forall v \in V$.

$\pi[v] = u$ 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

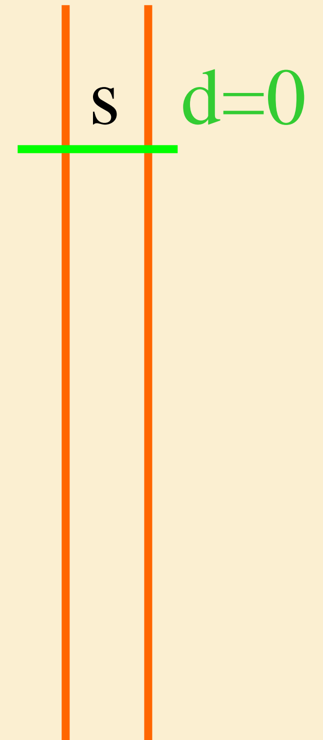
s
a
b
d
e
c
f
g
j
i
h
m
k
l

BFS

d=0

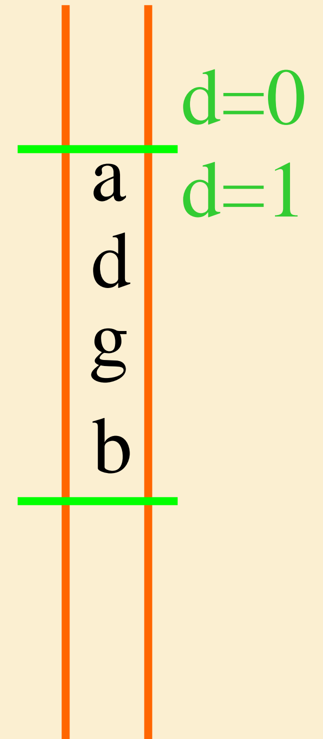
s a b c d e f g h i j k l m

Found
Not Handled
Queue



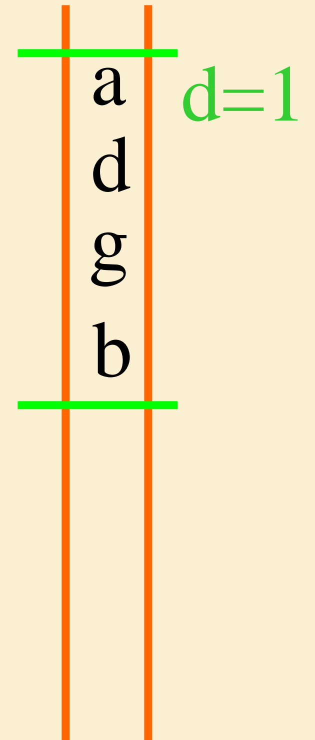
BFS

Found
Not Handled
Queue



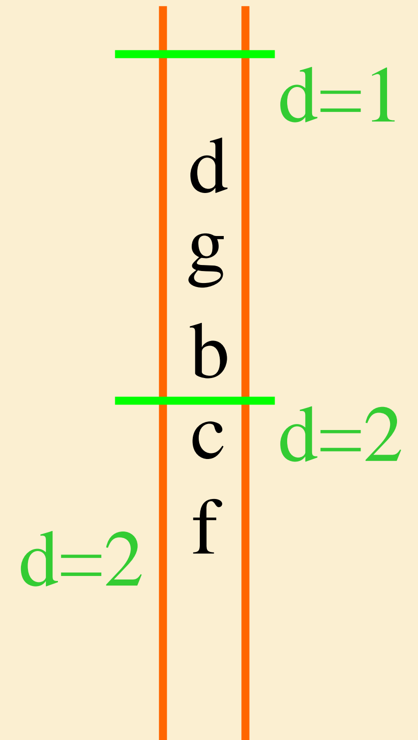
BFS

Found
Not Handled
Queue



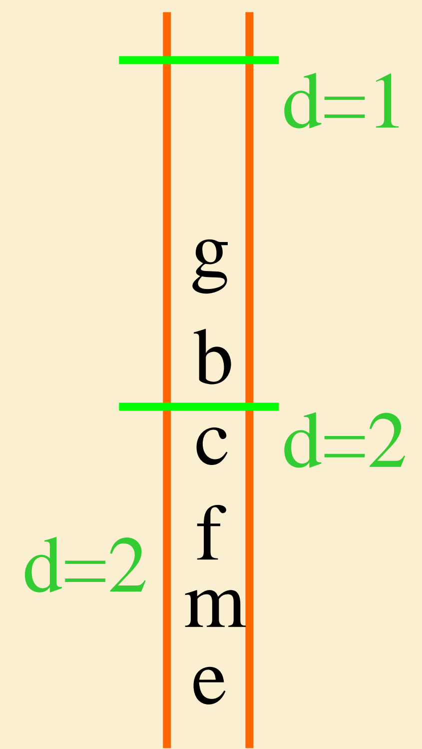
BFS

Found
Not Handled
Queue



BFS

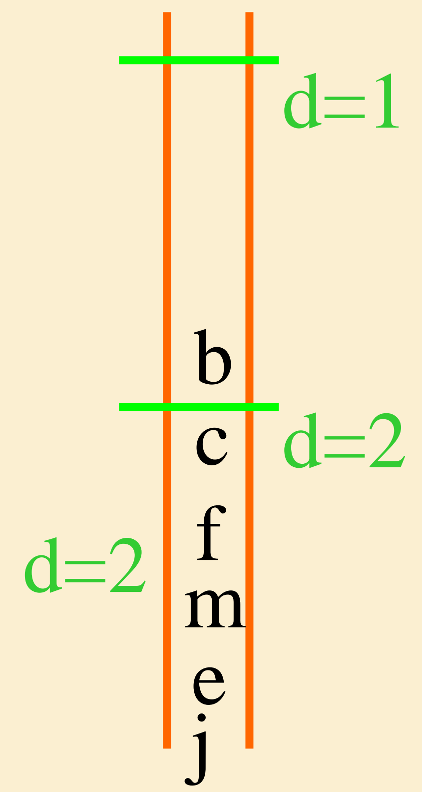
Found
Not Handled
Queue



BFS

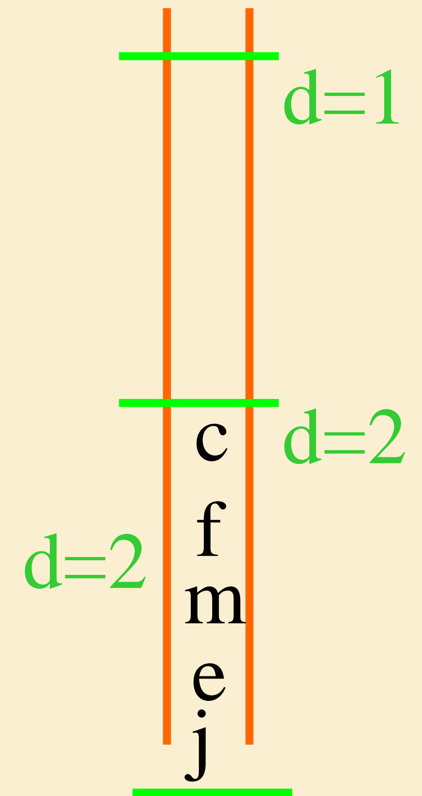
d=0
s
d=1
a
b
d
e
g
c
f
j
d=2
i
h
m
k
l

Found
Not Handled
Queue



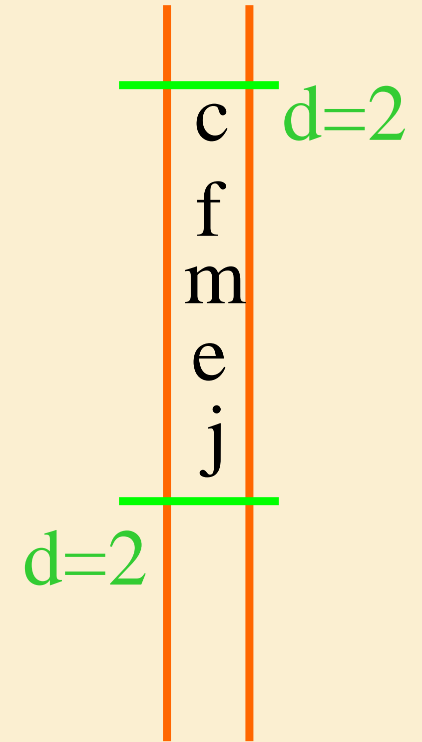
BFS

Found
Not Handled
Queue



BFS

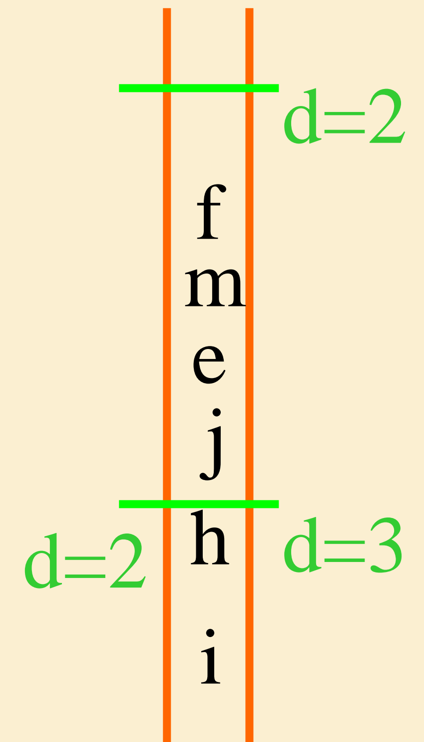
Found
Not Handled
Queue



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

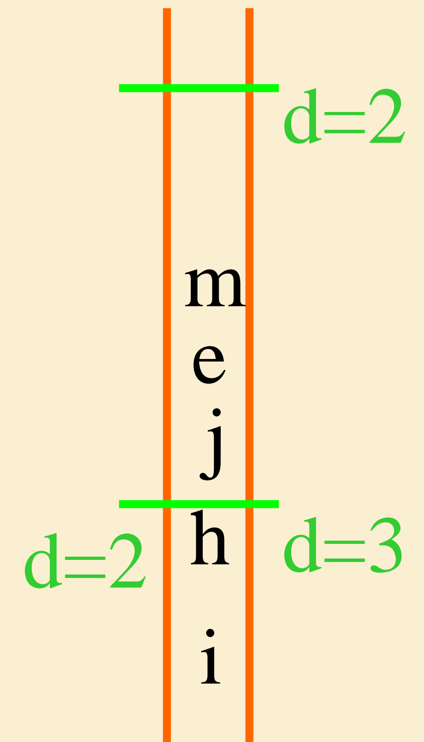
BFS

Found
Not Handled
Queue



BFS

Found
Not Handled
Queue

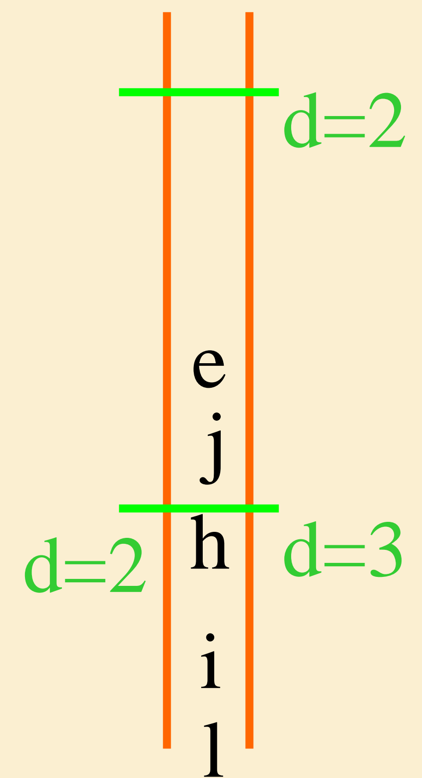


BFS

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

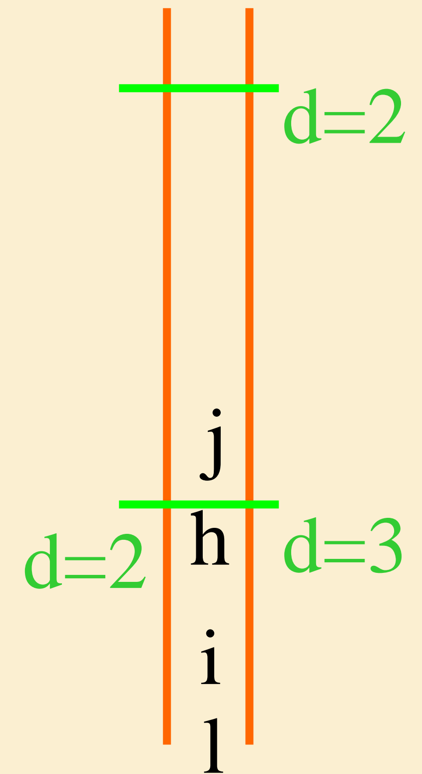
d=0 d=1 d=3

Found
Not Handled
Queue



BFS

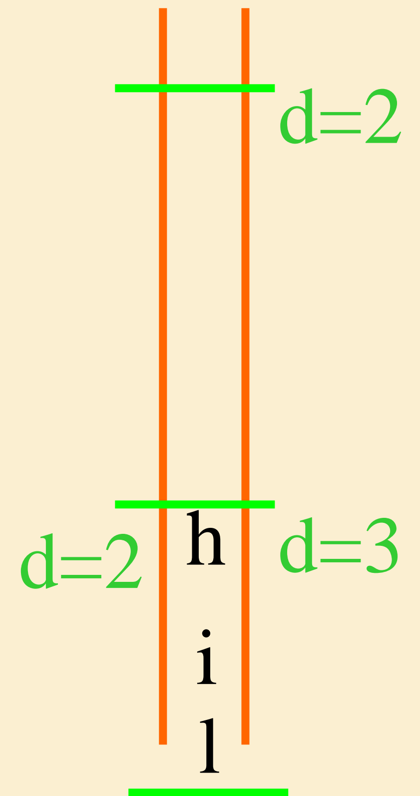
Found
Not Handled
Queue



BFS

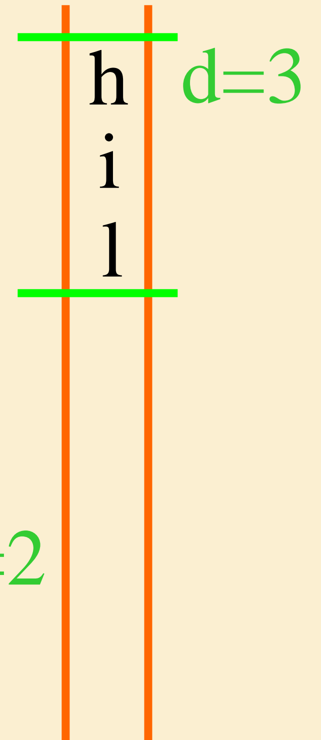
s
d=0
d=1
a
b
e
d
g
c
f
j
i
h
m
k
l
d=3

Found
Not Handled
Queue



BFS

Found
Not Handled
Queue



d=0
d=1
d=2
d=3

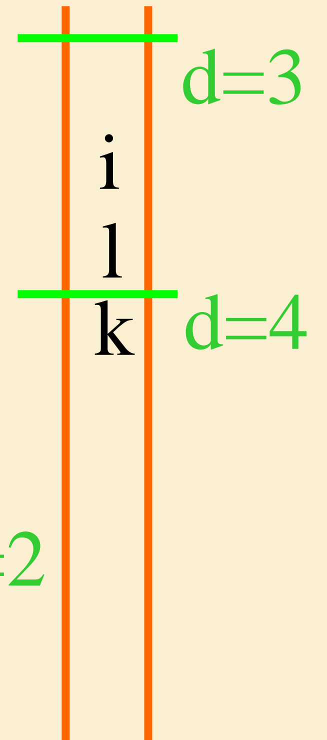
s a b c d e f g h i j k l m

BFS

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

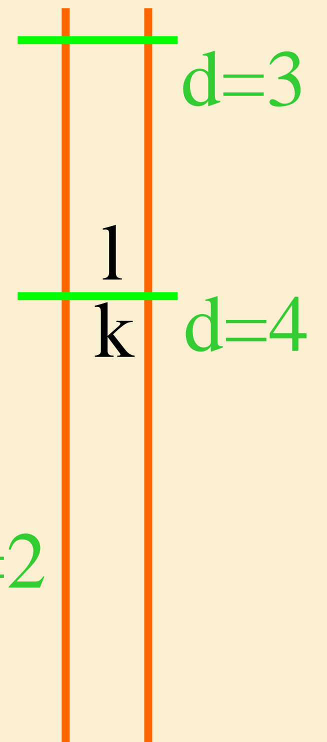
d=0 d=1 d=2 d=3 d=4

Found
Not Handled
Queue



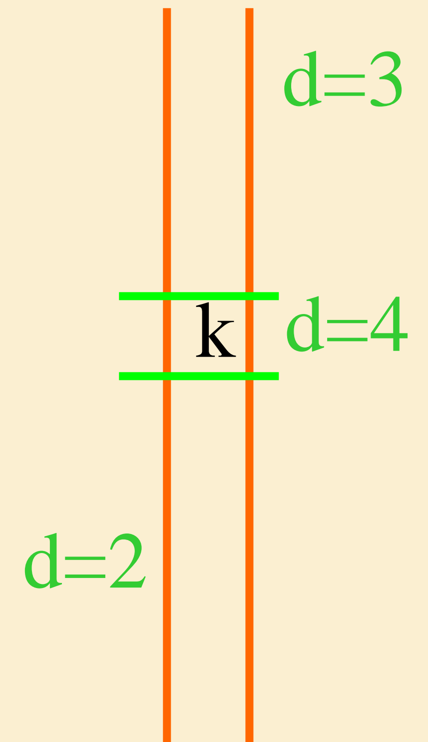
BFS

Found
Not Handled
Queue



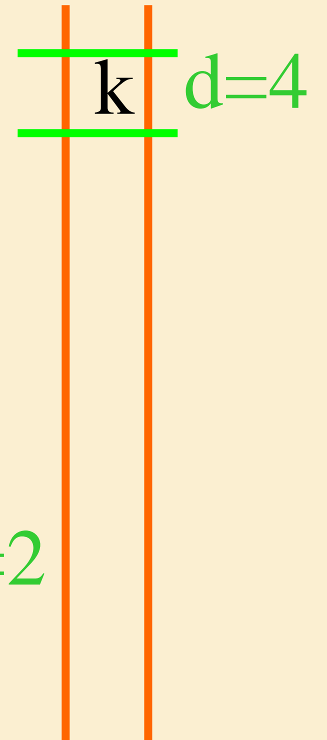
BFS

Found
Not Handled
Queue



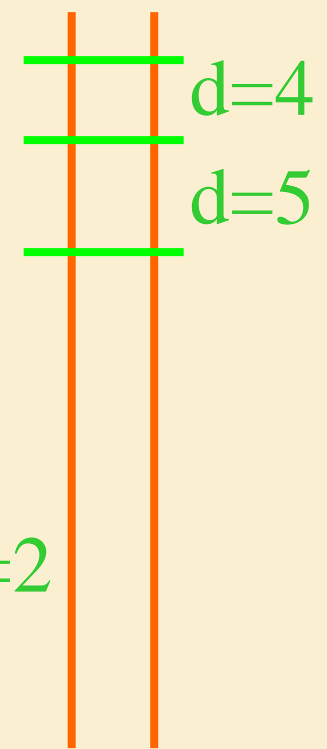
BFS

Found
Not Handled
Queue



BFS

Found
Not Handled
Queue



d=0
d=1
d=2
d=3
d=4

Breadth-First Search Algorithm

```
BFS(G, s)
 1  for each vertex u ∈ V[G] − {s}
 2       do color[u] ← BLACK
 3          d[u] ← ∞
 4          π[u] ← NIL
 5  color[s] ← RED
 6  d[s] ← 0
 7  π[s] ← NIL
 8  Q ← Ø
 9  ENQUEUE(Q, s)
10  while Q ≠ Ø
11       do u ← DEQUEUE(Q)
12          for each v ∈ Adj[u]
13              do if color[v] = BLACK
14                    then color[v] ← RED
15                         d[v] ← d[u] + 1
16                         π[v] ← u
17                         ENQUEUE(Q, v)
18          color[u] ← 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

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)
 1  for each vertex u ∈ V[G] − {s}
 2       do color[u] ← BLACK
 3          d[u] ← ∞
 4          π[u] ← NIL
 5  color[s] ← RED
 6  d[s] ← 0
 7  π[s] ← NIL
 8  Q ← Ø
 9  ENQUEUE(Q, s)
10  while Q ≠ Ø
11       do u ← DEQUEUE(Q)
12          for each v ∈ Adj[u]
13              do if color[v] = BLACK
14                    then color[v] ← RED
15                         d[v] ← d[u] + 1
16                         π[v] ← u
17                         ENQUEUE(Q, v)
18          color[u] ← GRAY
```

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.

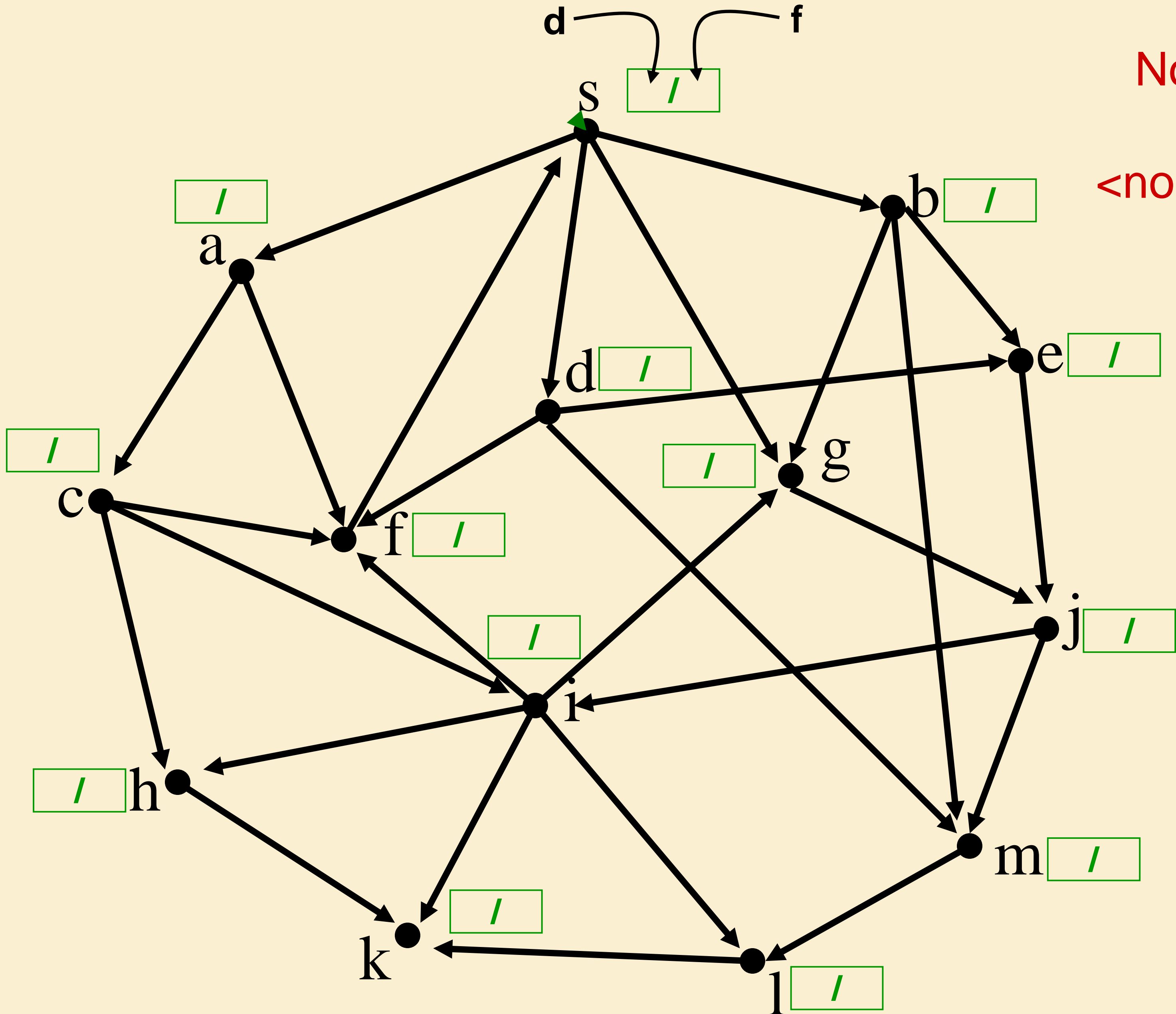
$f[v] =$ finishing time.

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

- 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).

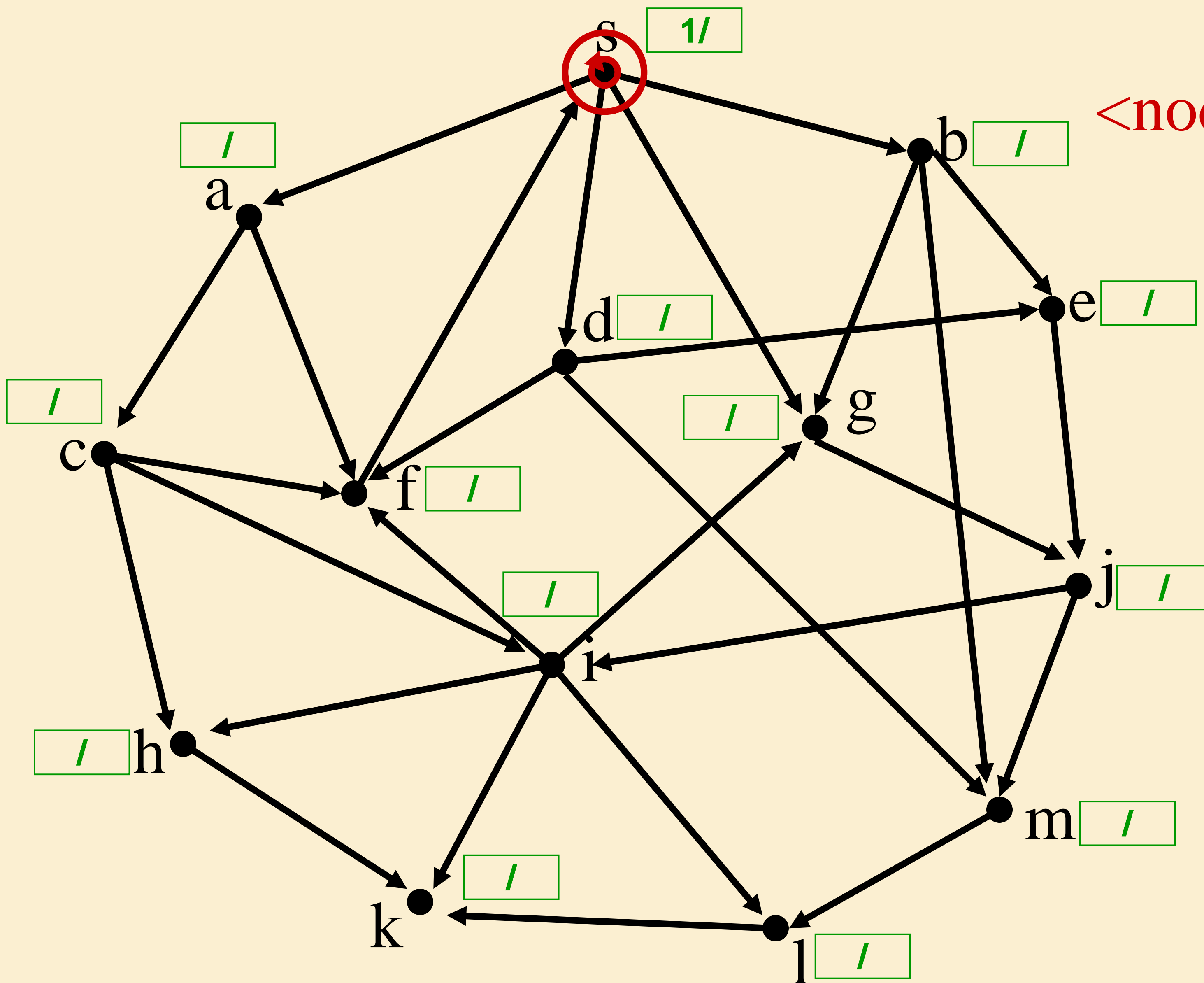
DFS

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



Found
Not Handled
Stack
<node,# edges>

DFS



Found
Not Handled
Stack
<node,# edges>

s,0

DFS

Found
Not Handled
Stack
<node,# edges>

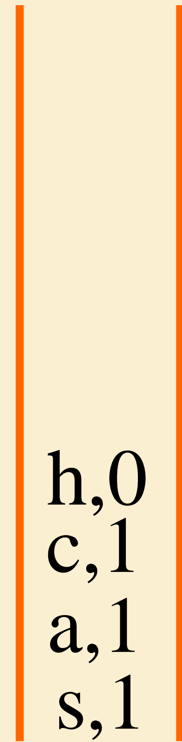
DFS

Found
Not Handled
Stack
<node,# edges>

s 1/
a 2/
c 3/
b /
d /
e /
f /
g /
h /
i /
j /
k /
l /
m /

DFS

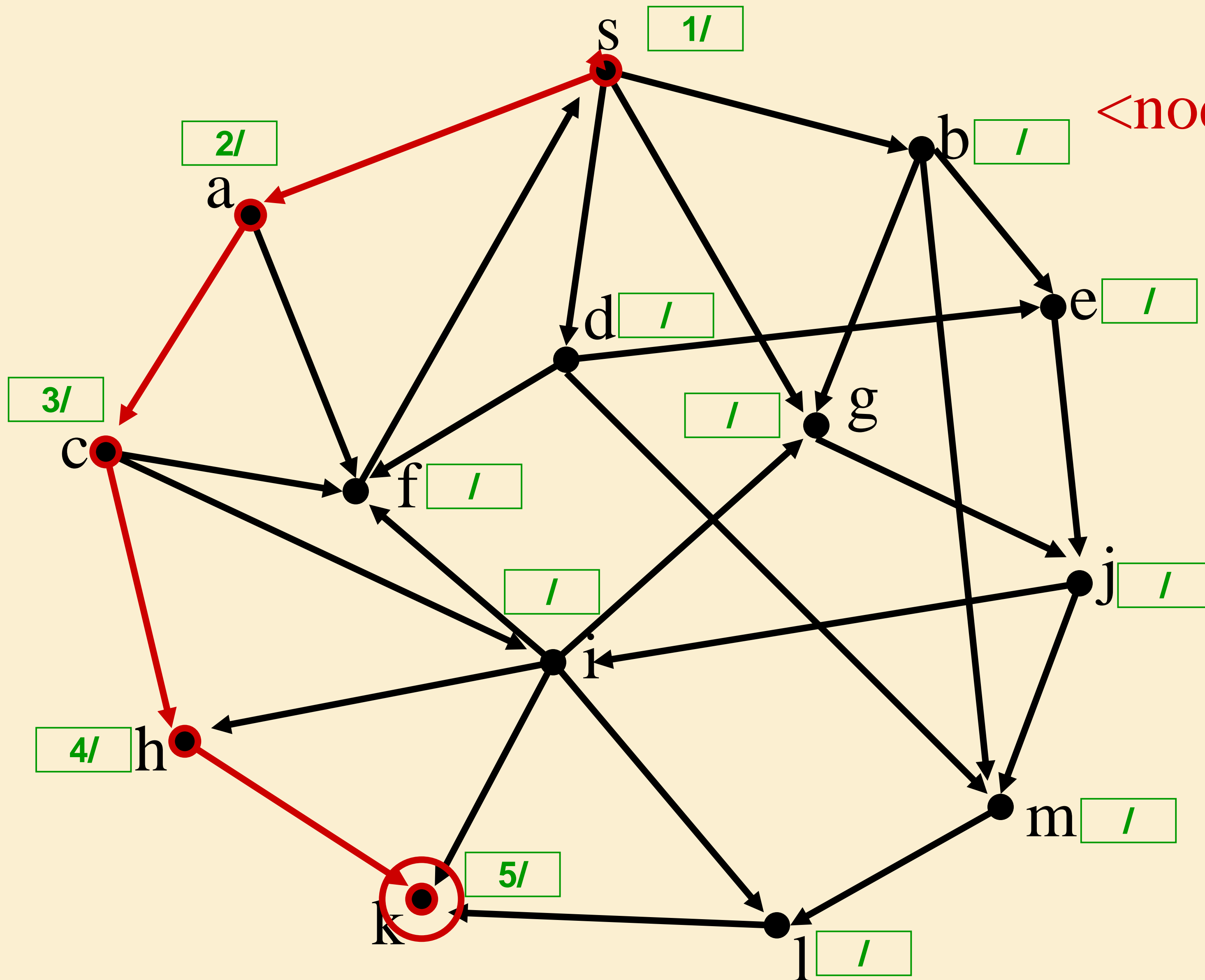
Found
Not Handled
Stack
<node,# edges>



s 1/
a 2/
b /
d /
e /
c 3/
g /
f /
i /
j /
h 4/
m /
k /
l /

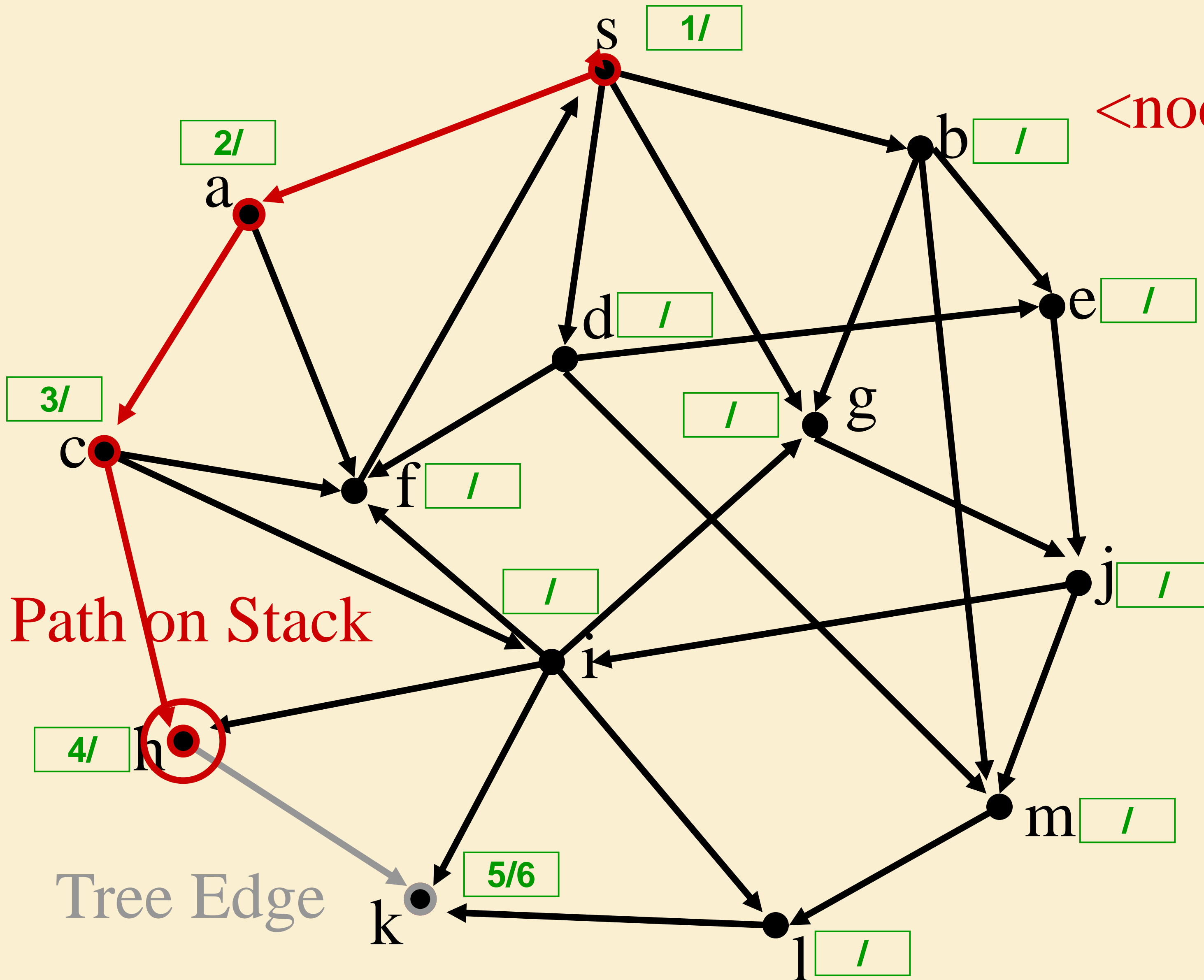
DFS

Found
Not Handled
Stack
<node,# edges>



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

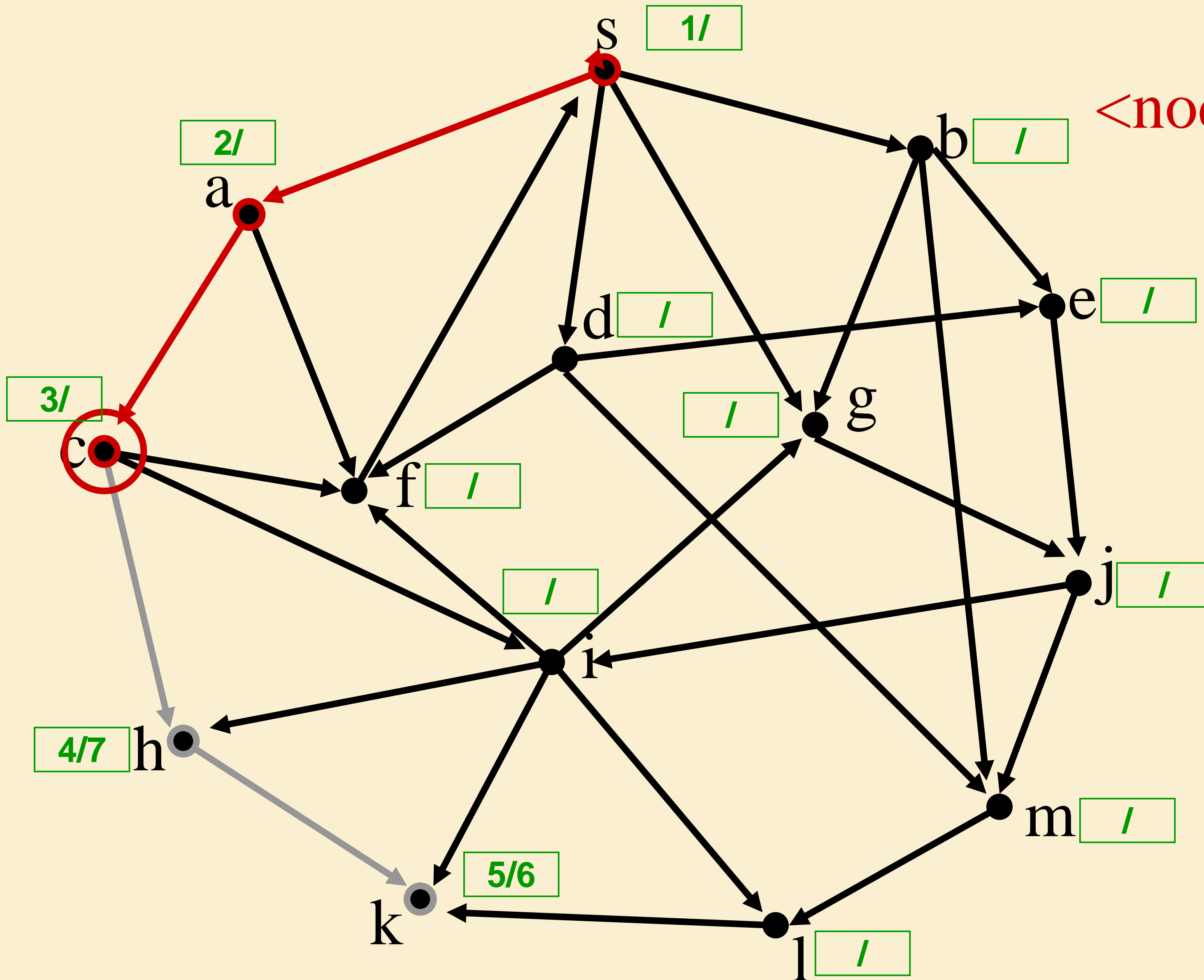
DFS



Found
Not Handled
Stack
<node,# edges>

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

DFS

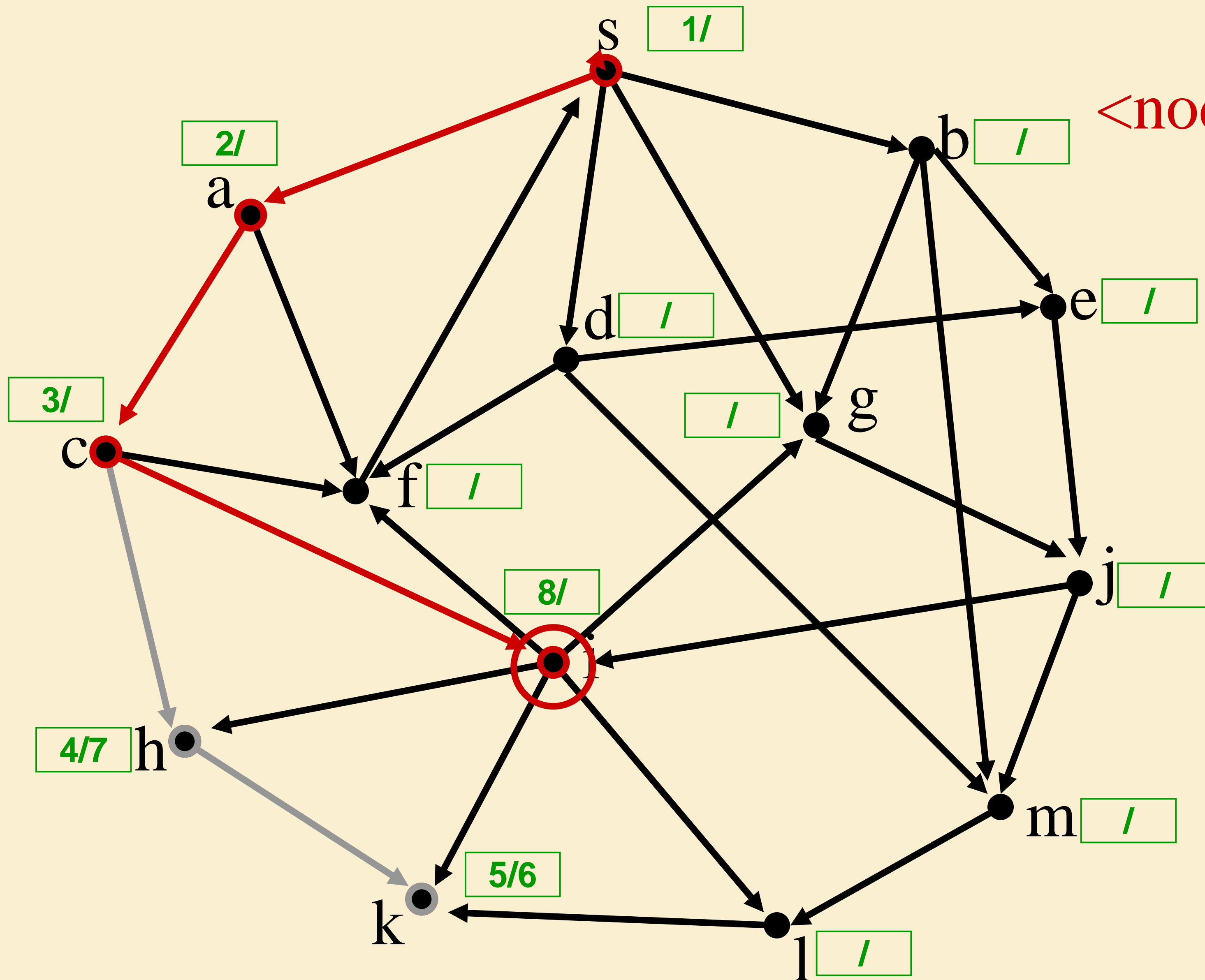


Found
Not Handled
Stack
<node,# edges>

c,1
a,1
s,1

DFS

Found
Not Handled
Stack
<node,# edges>



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

DFS

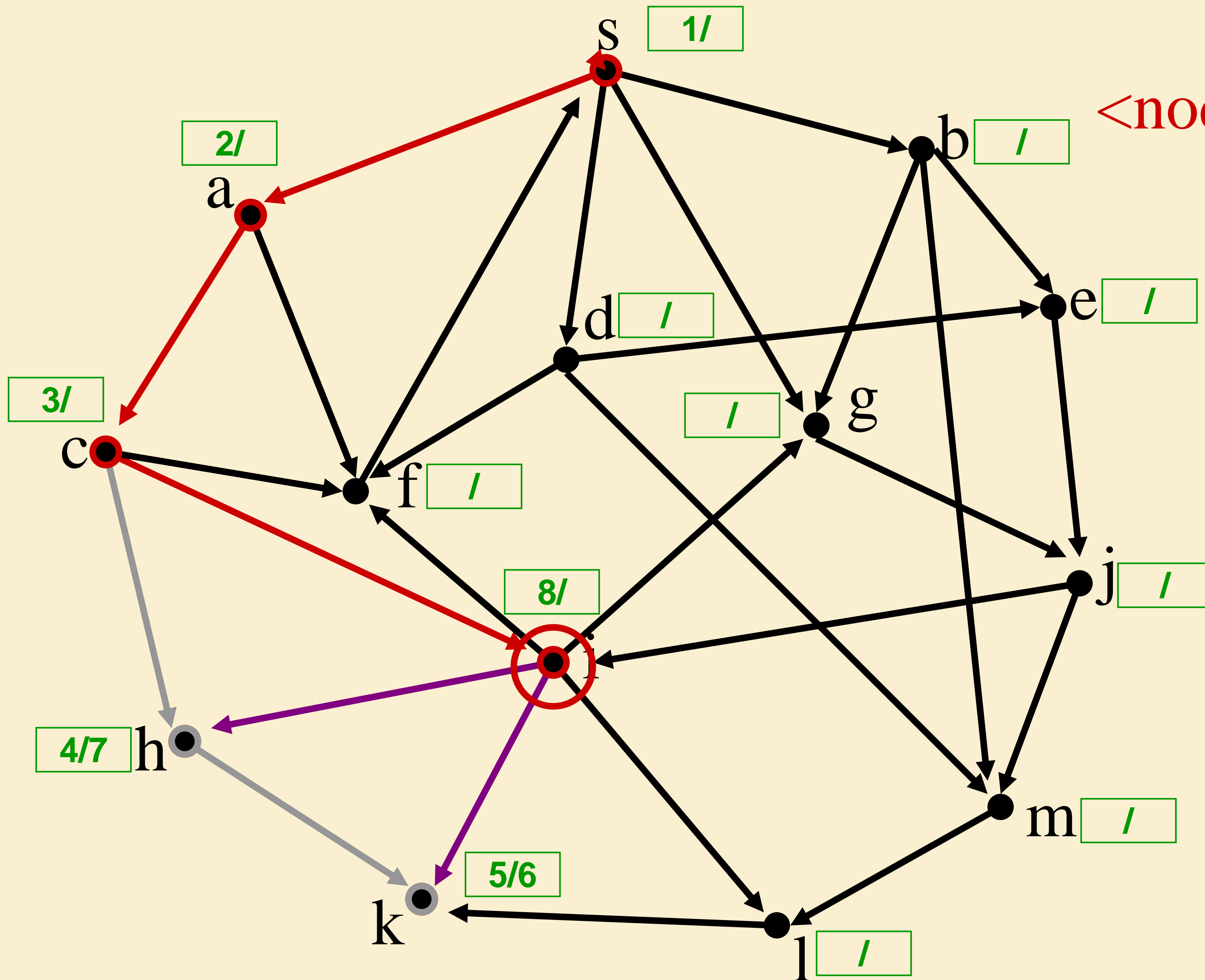
Cross Edge to handled node: d[h]<d[i]

Found
Not Handled
Stack
<node,# edges>

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

s 1/
a 2/
c 3/
h 4/7
k 5/6
i 8/
b /
d /
e /
g /
f /
j /
m /
l /

DFS



Found

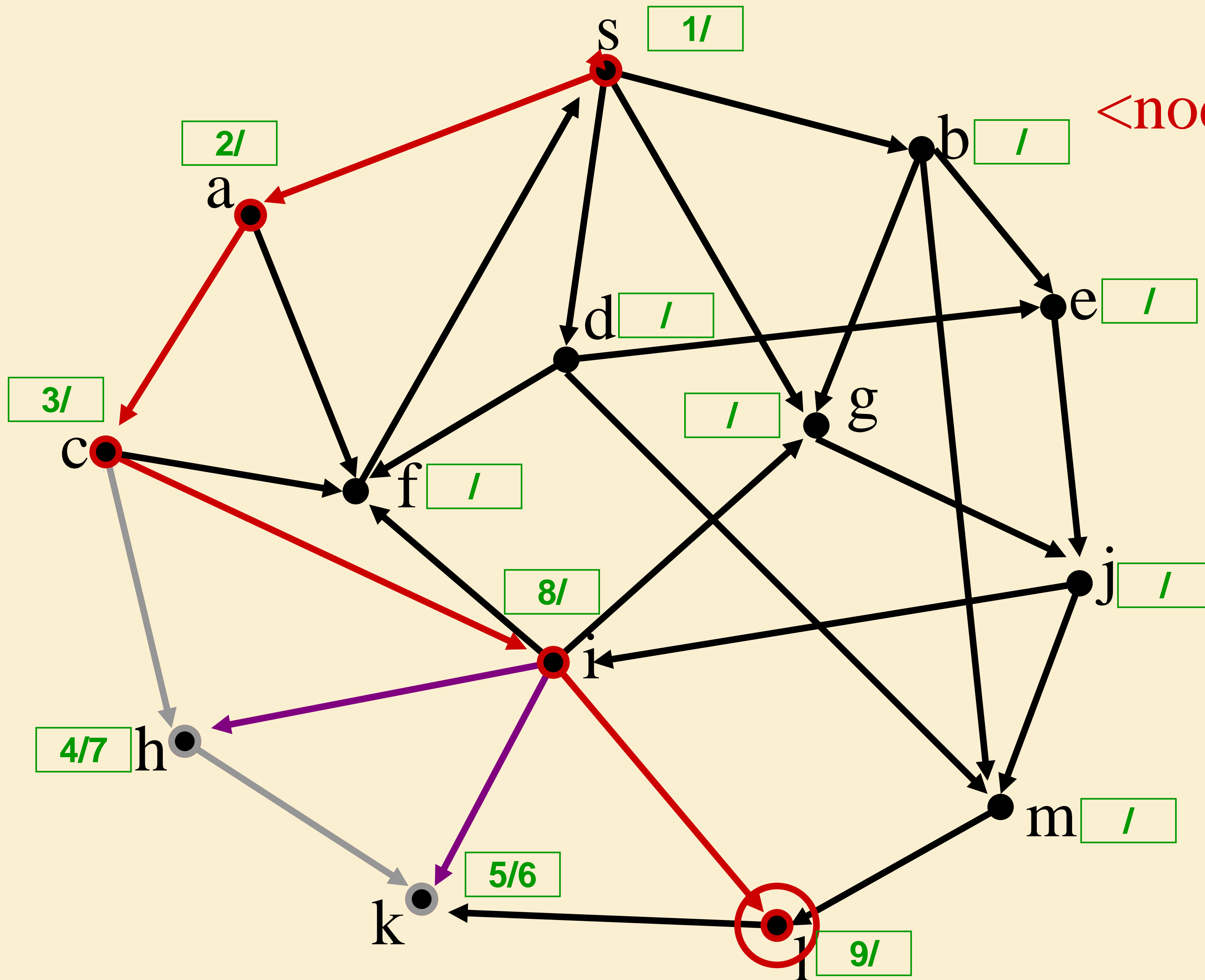
Not Handled

Stack

<node,# edges>

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

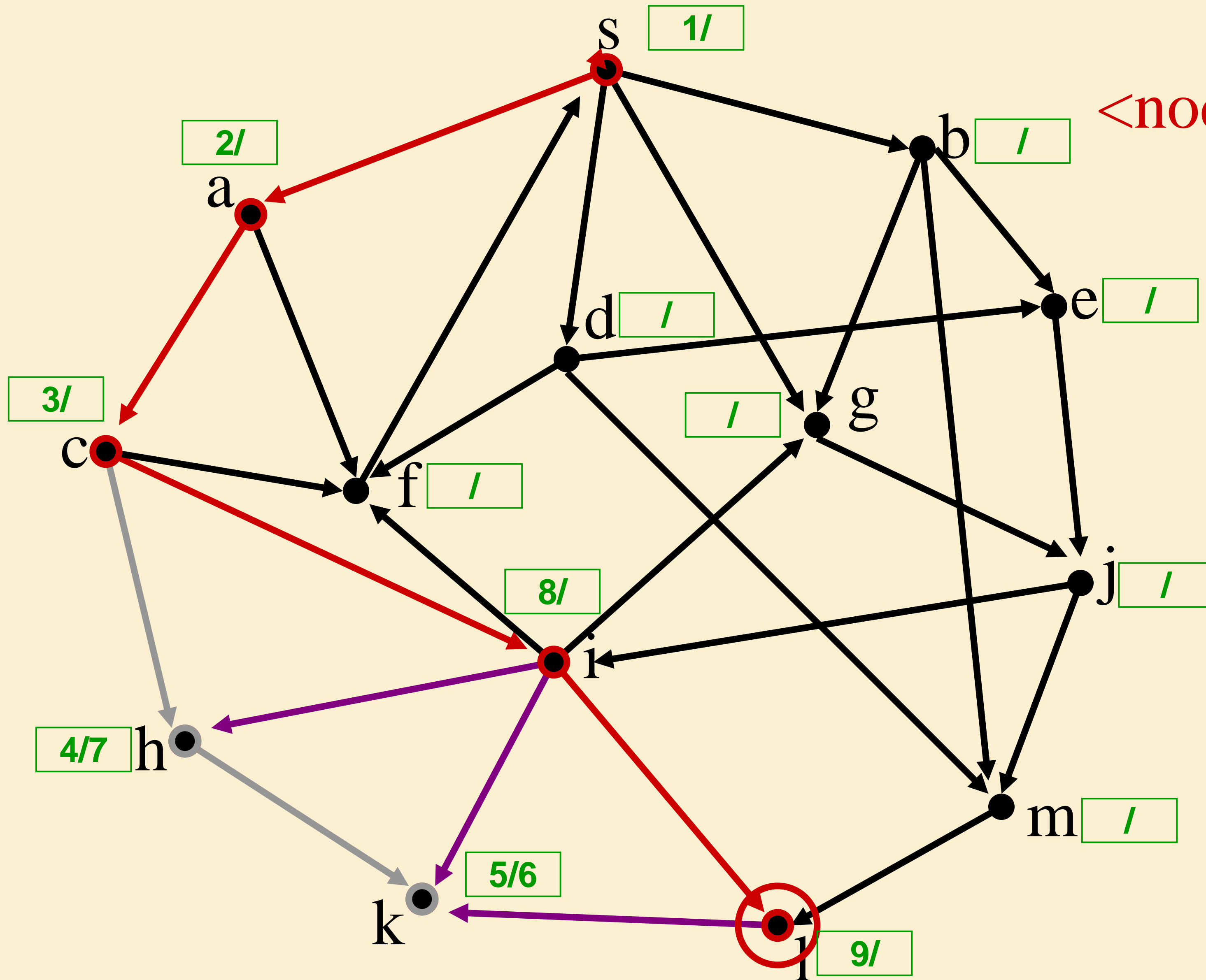
DFS



Found
Not Handled
Stack
<node,# edges>

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

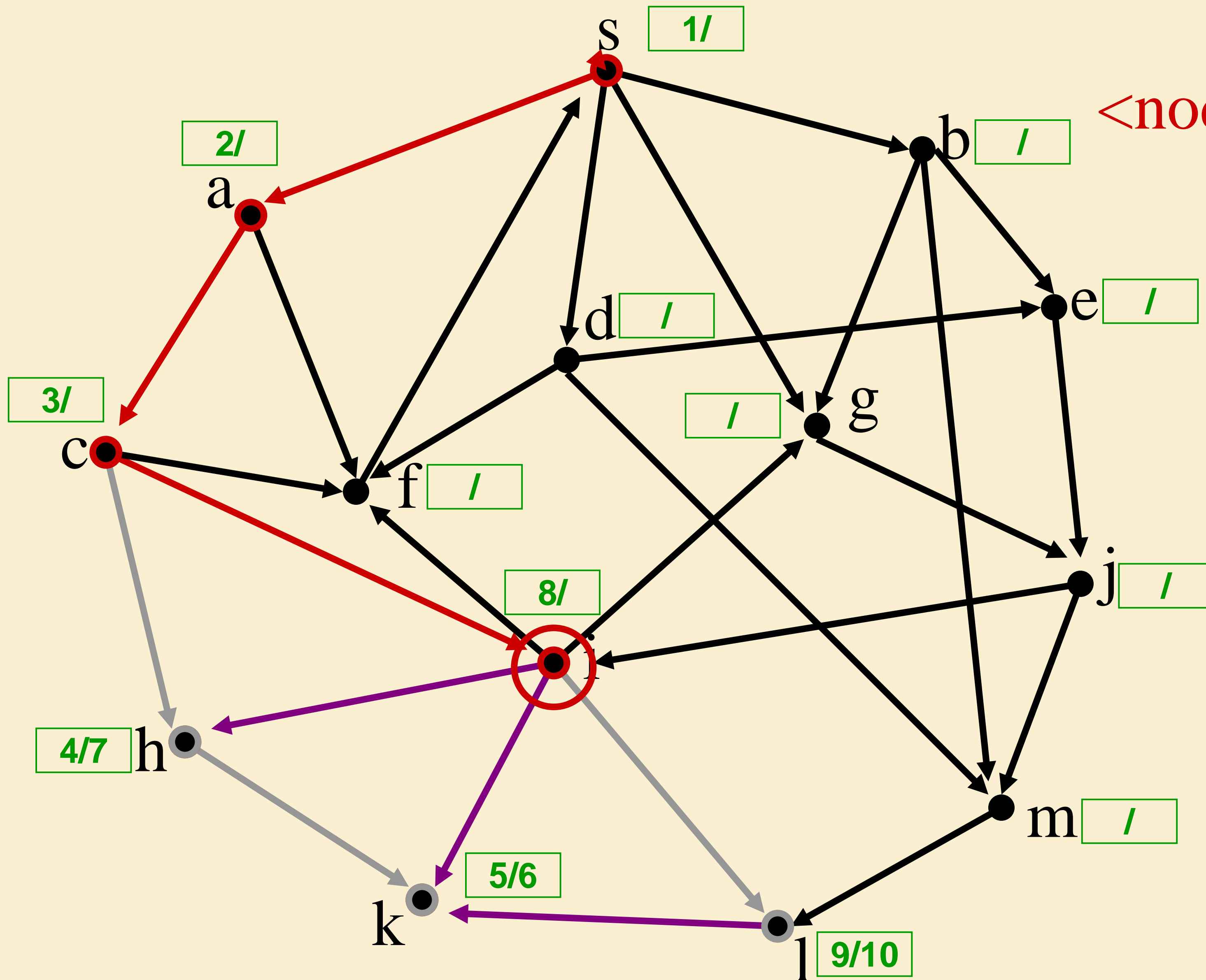
DFS



Found
Not Handled
Stack
<node,# edges>

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

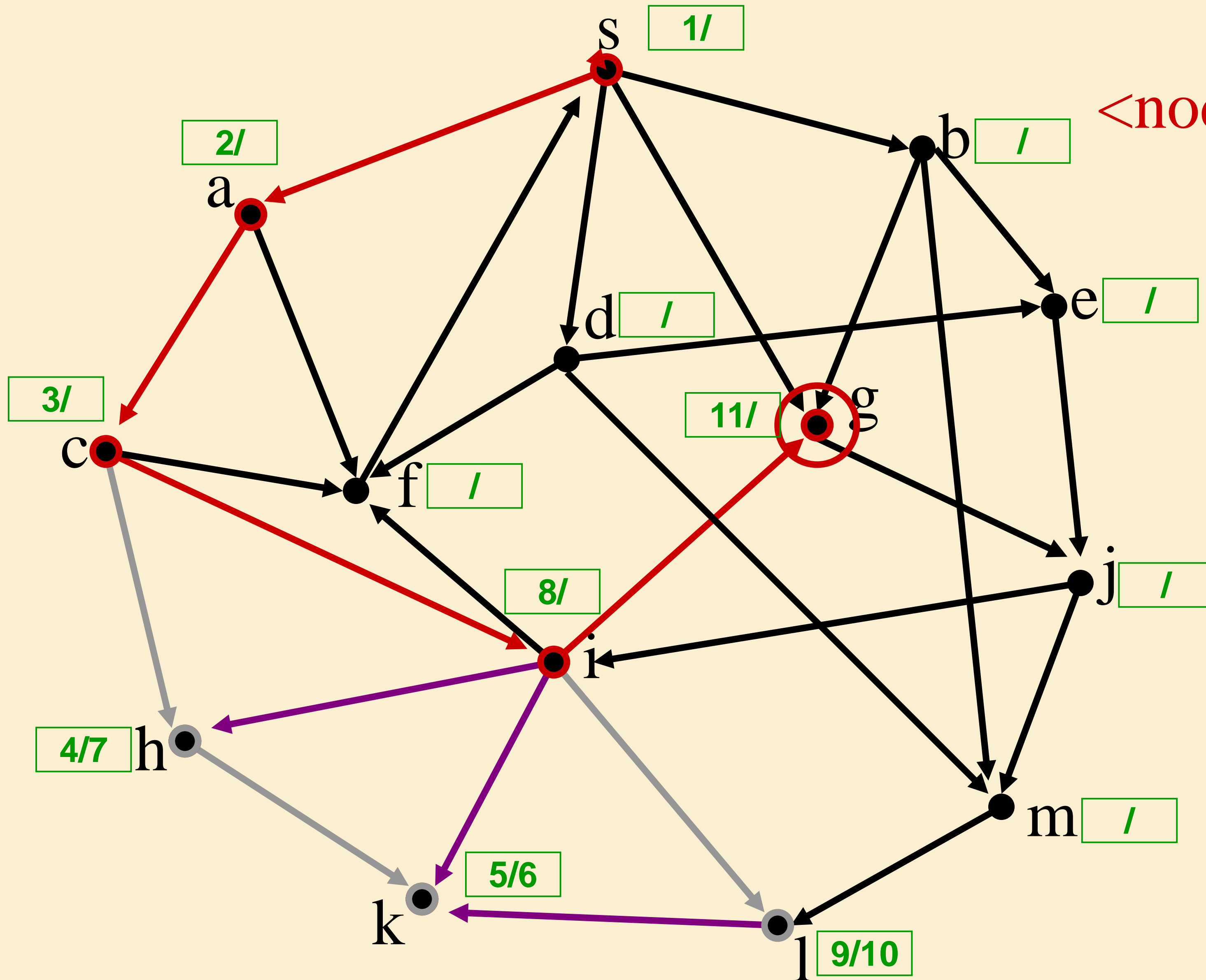
DFS



Found
Not Handled
Stack
<node,# edges>

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

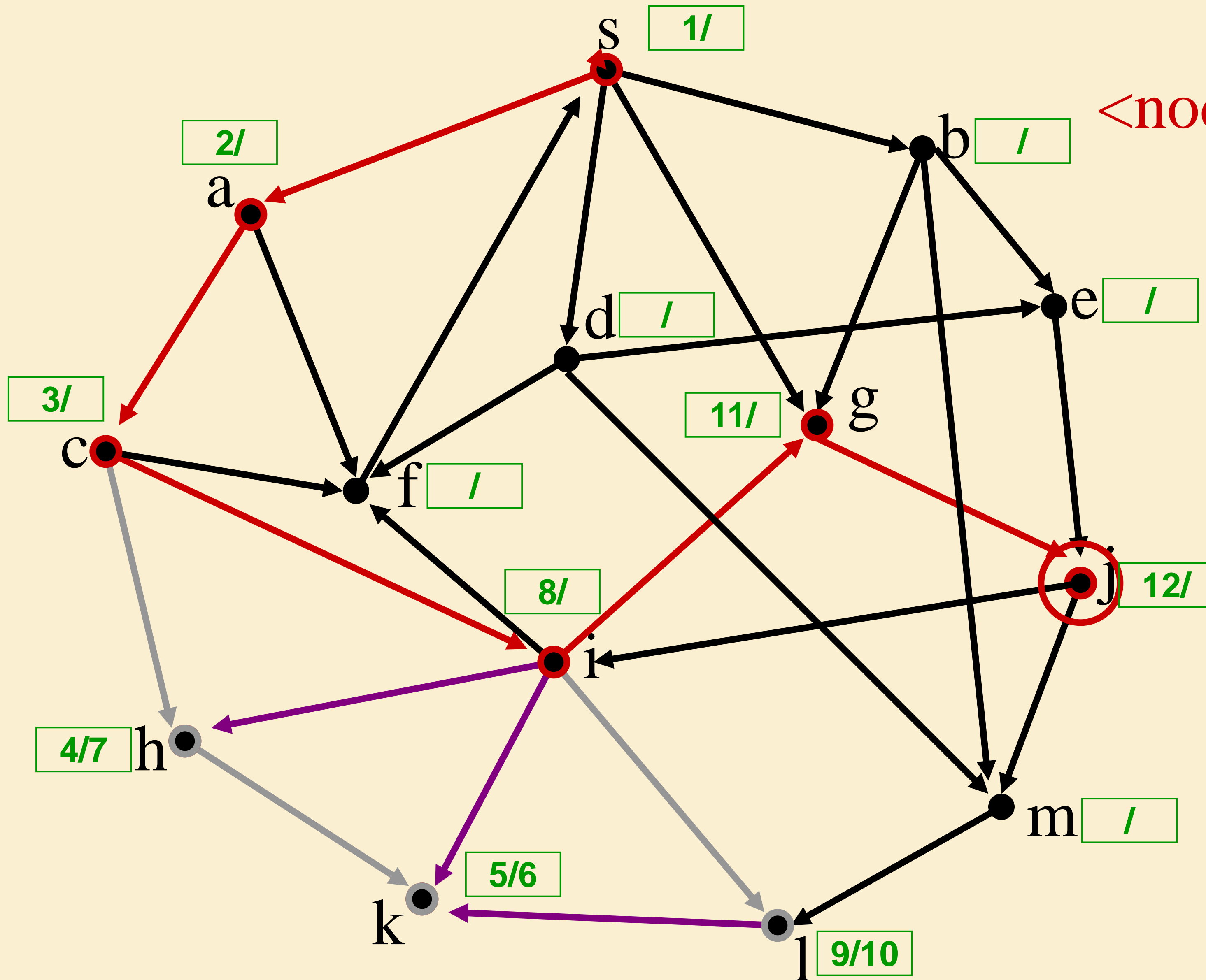
DFS



Found
Not Handled
Stack
<node,# edges>

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

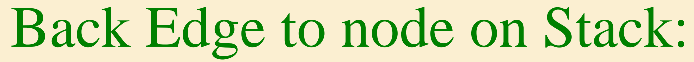
DFS



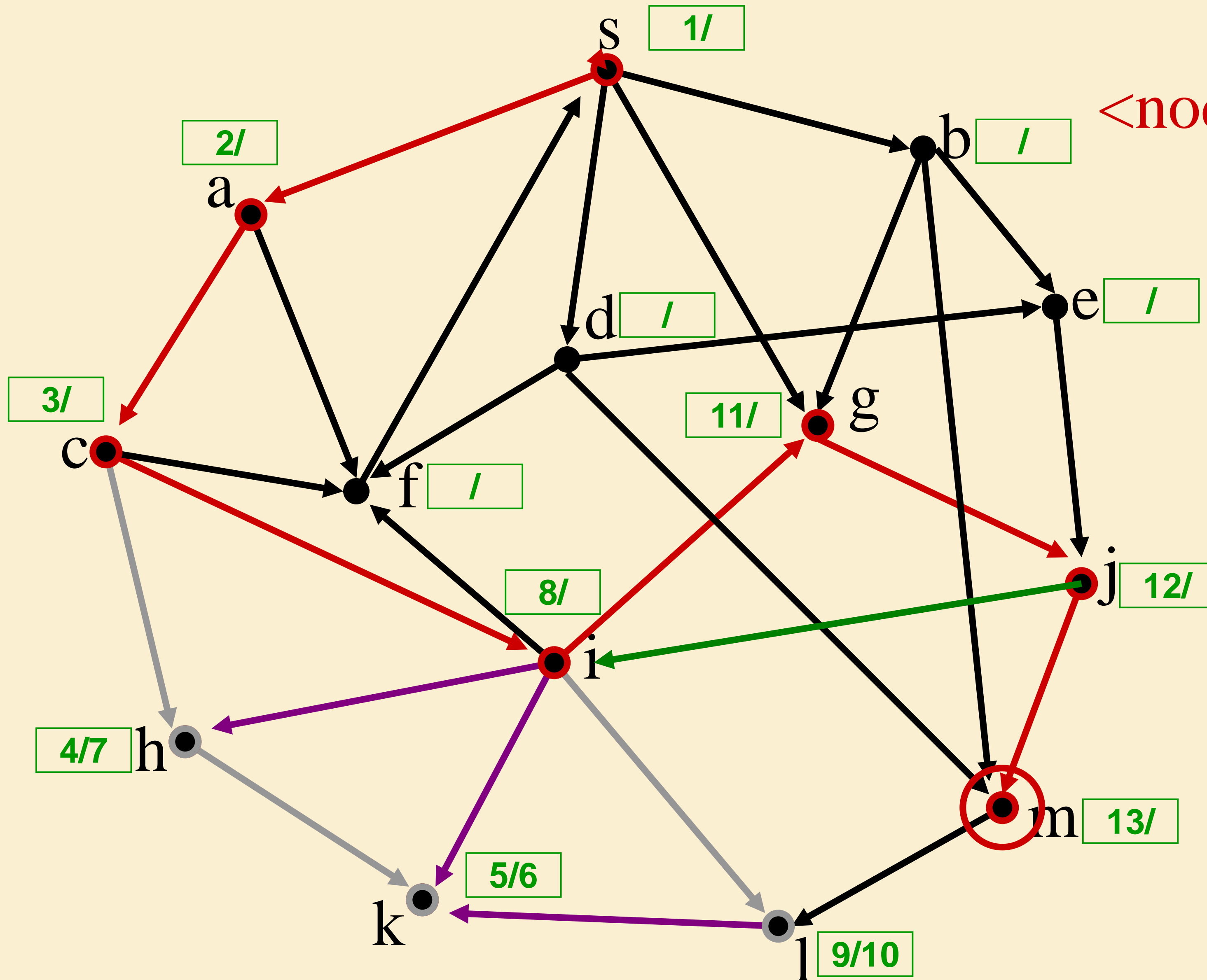
Found Not Handled
Stack
<node,# edges>

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

DFS



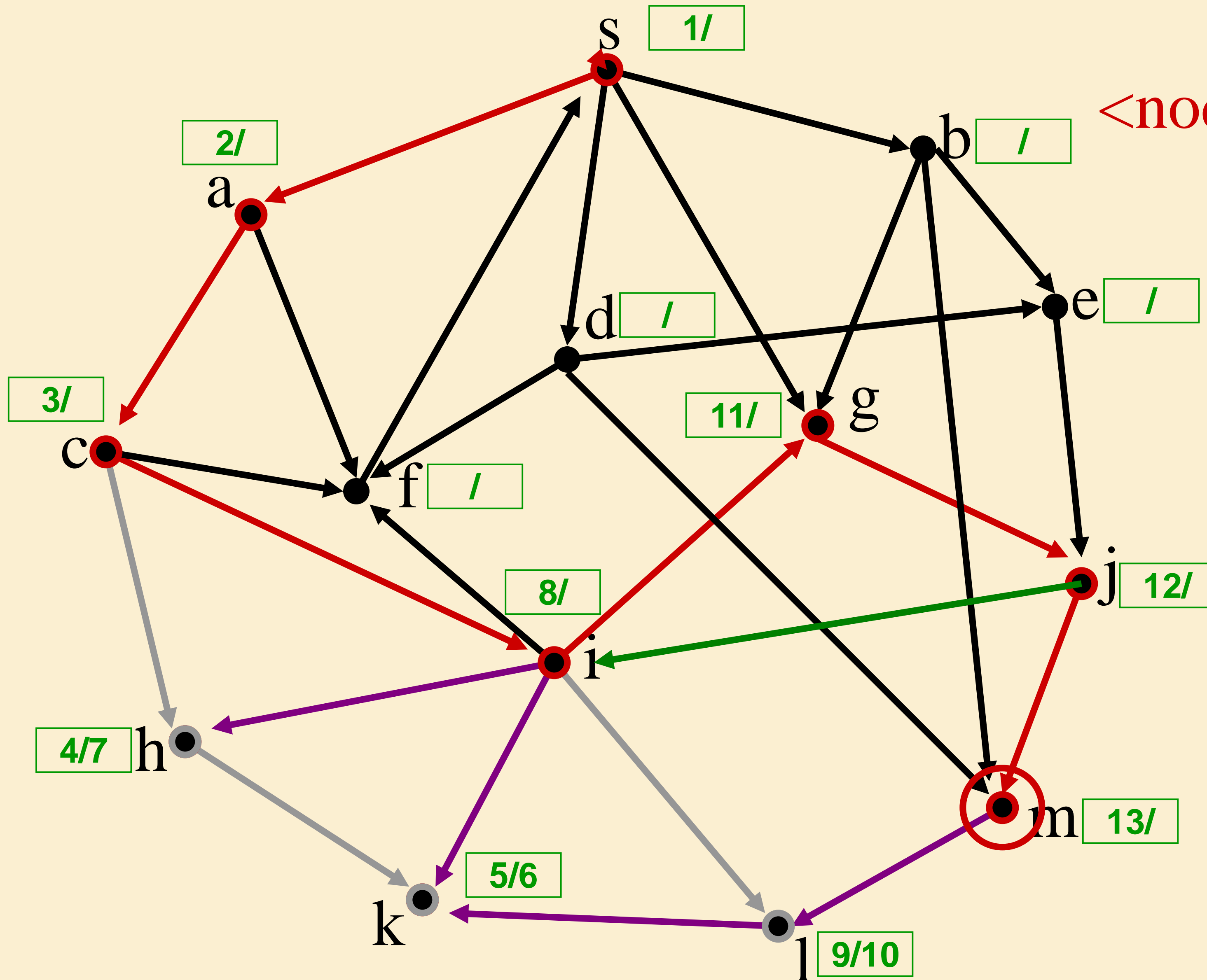
DFS



Found
Not Handled
Stack
<node,# edges>

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

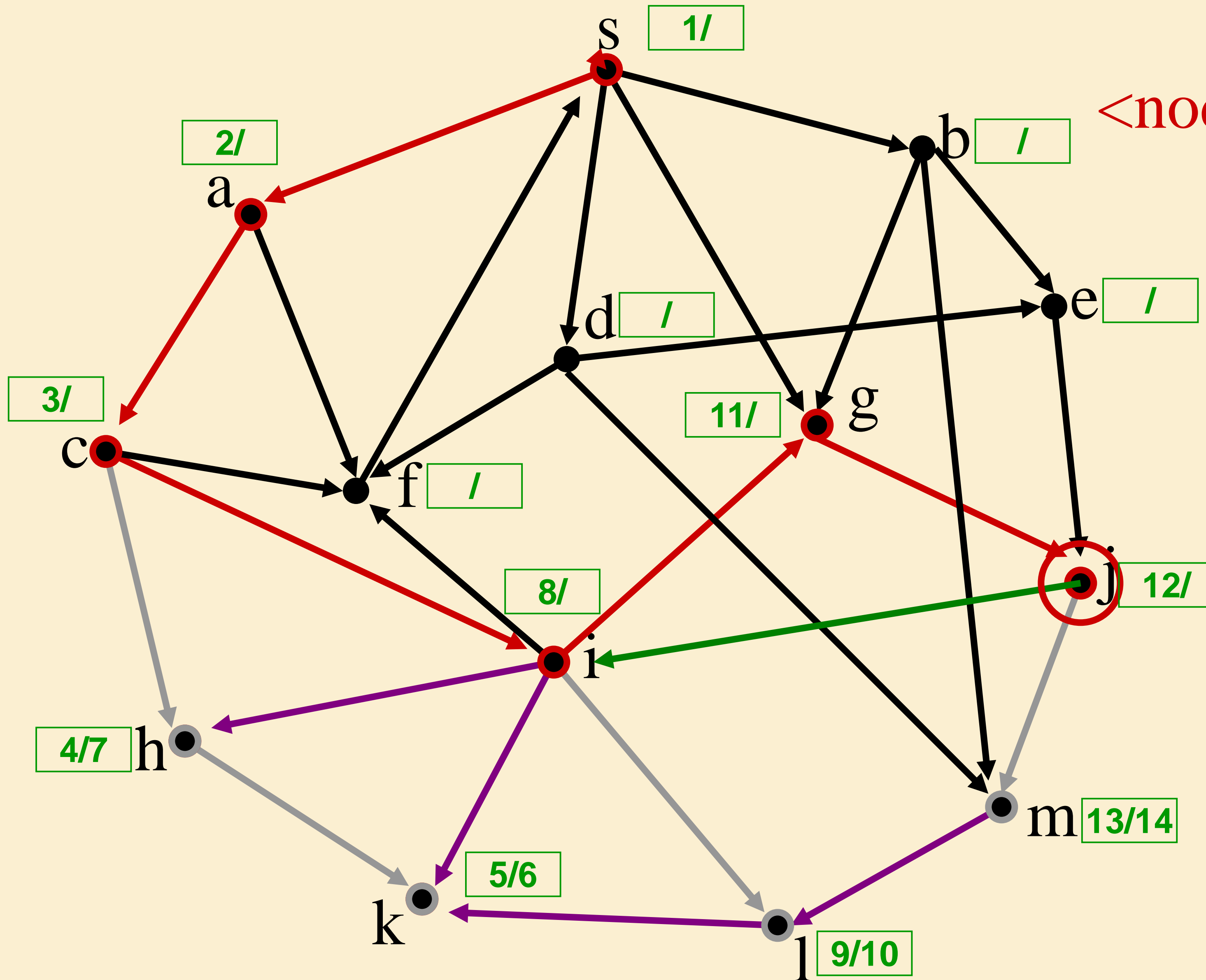
DFS



Found
Not Handled
Stack
<node,# edges>

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

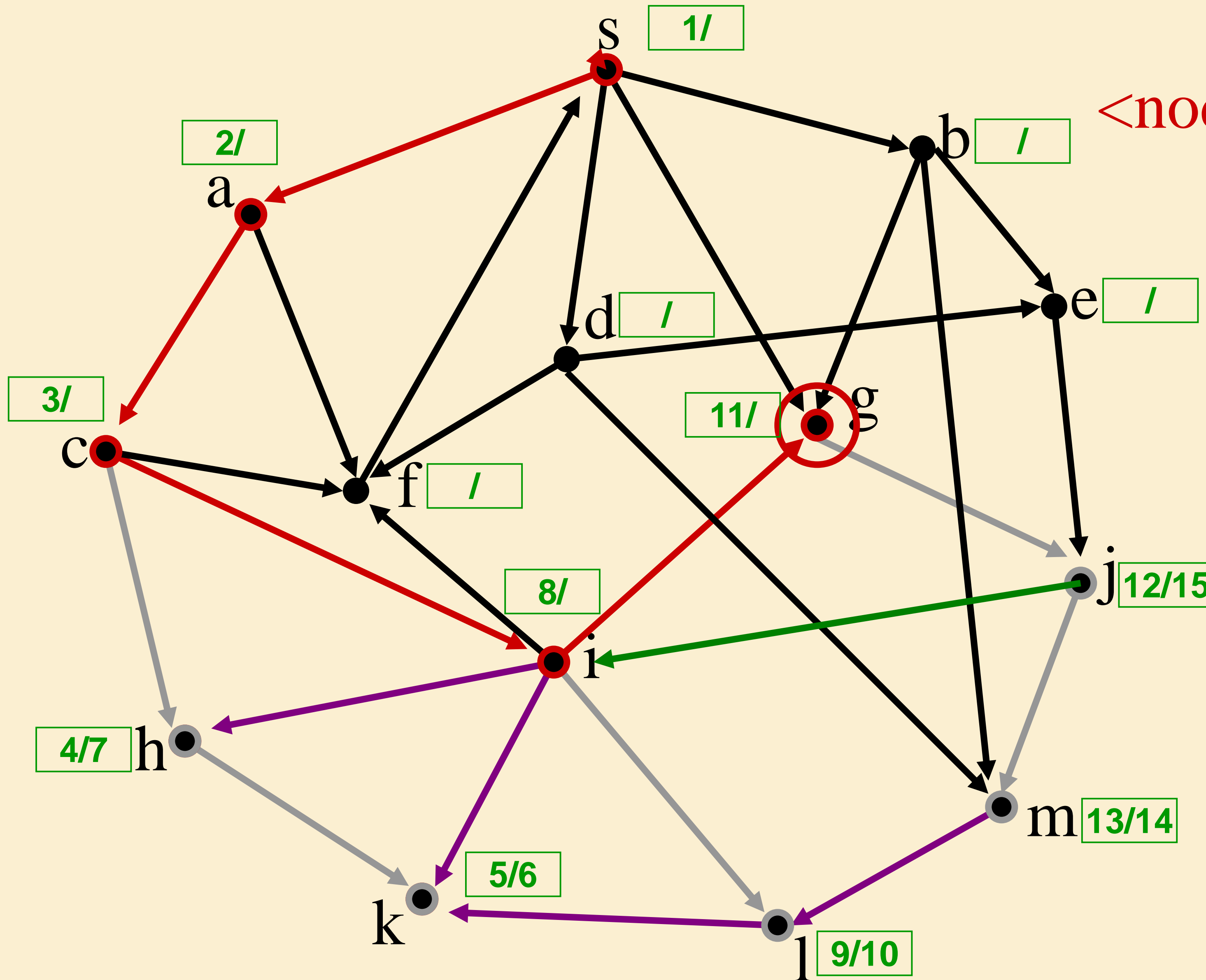
DFS



Found Not Handled Stack
<node,# edges>

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

DFS

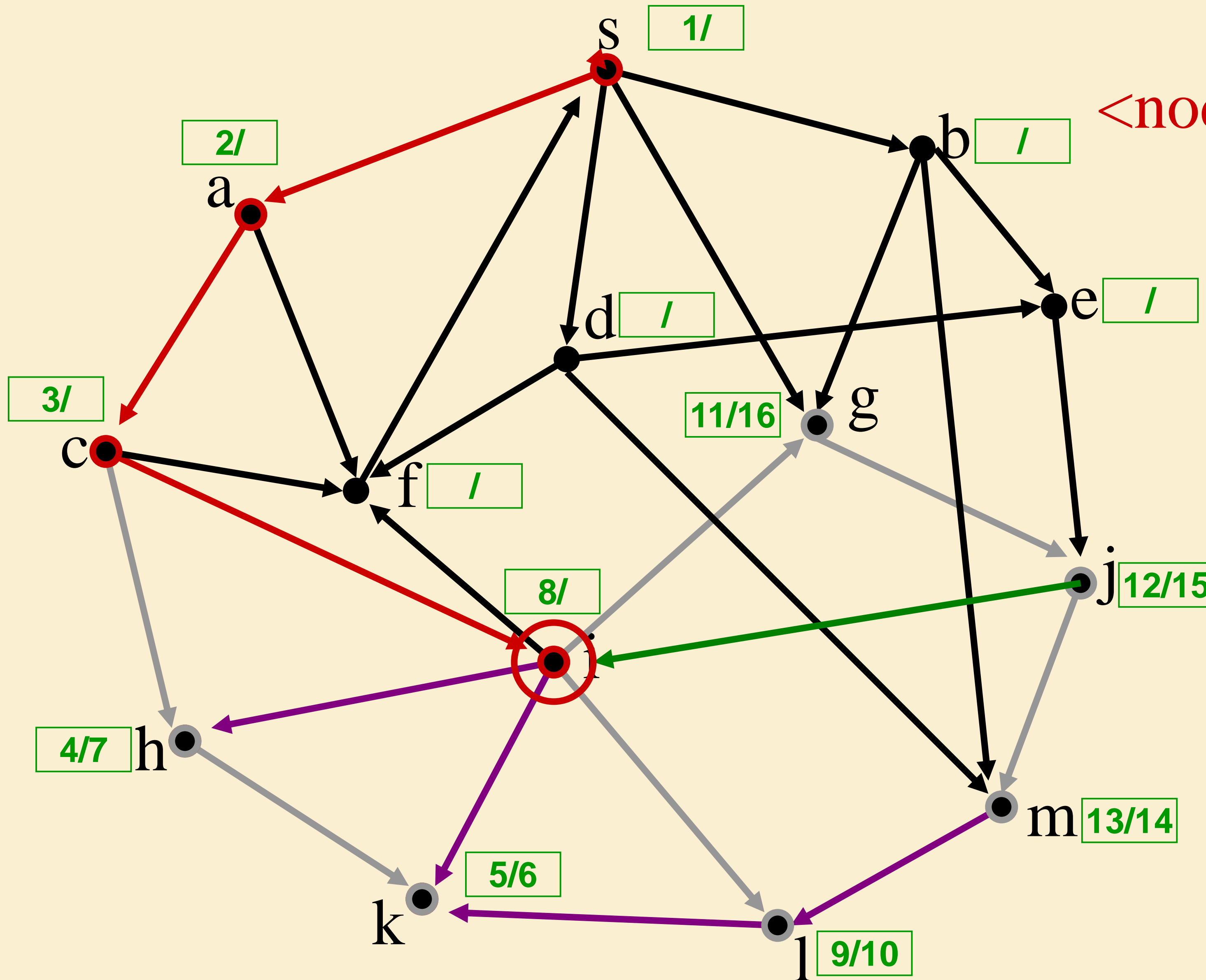


Found
Not Handled
Stack
<node,# edges>

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

DFS

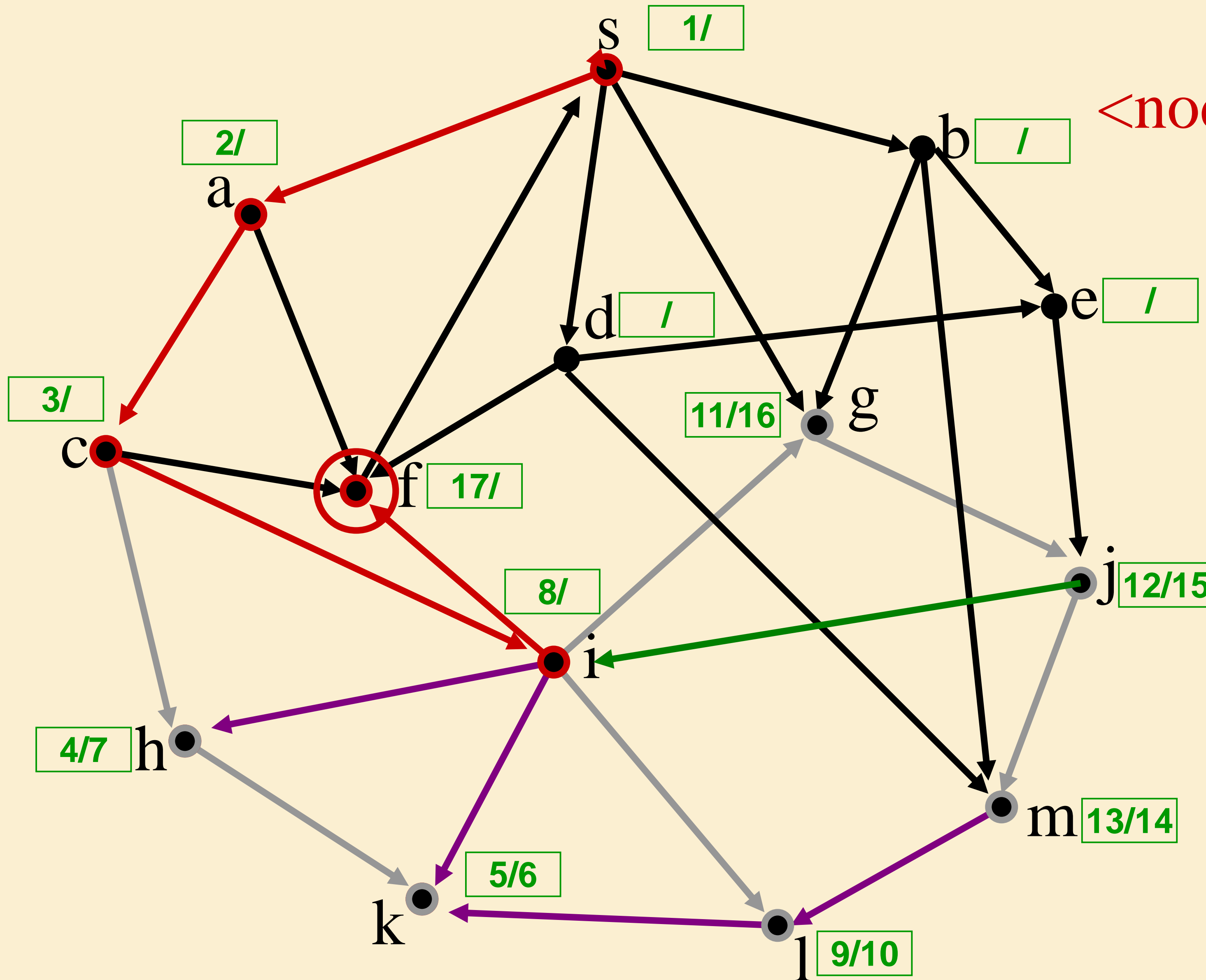
Found
Not Handled
Stack
<node,# edges>



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

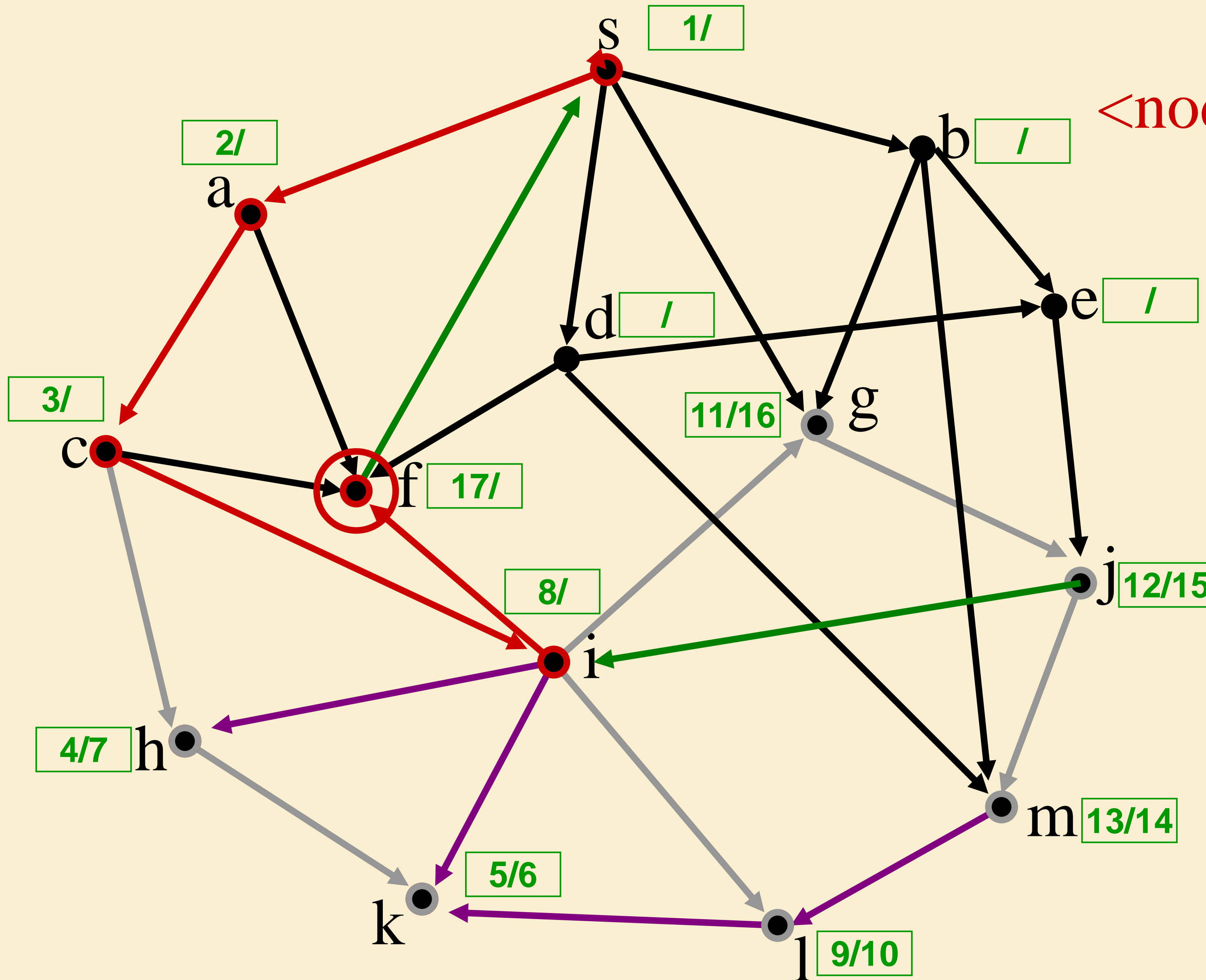
DFS

Found
Not Handled
Stack
<node,# edges>



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

DFS

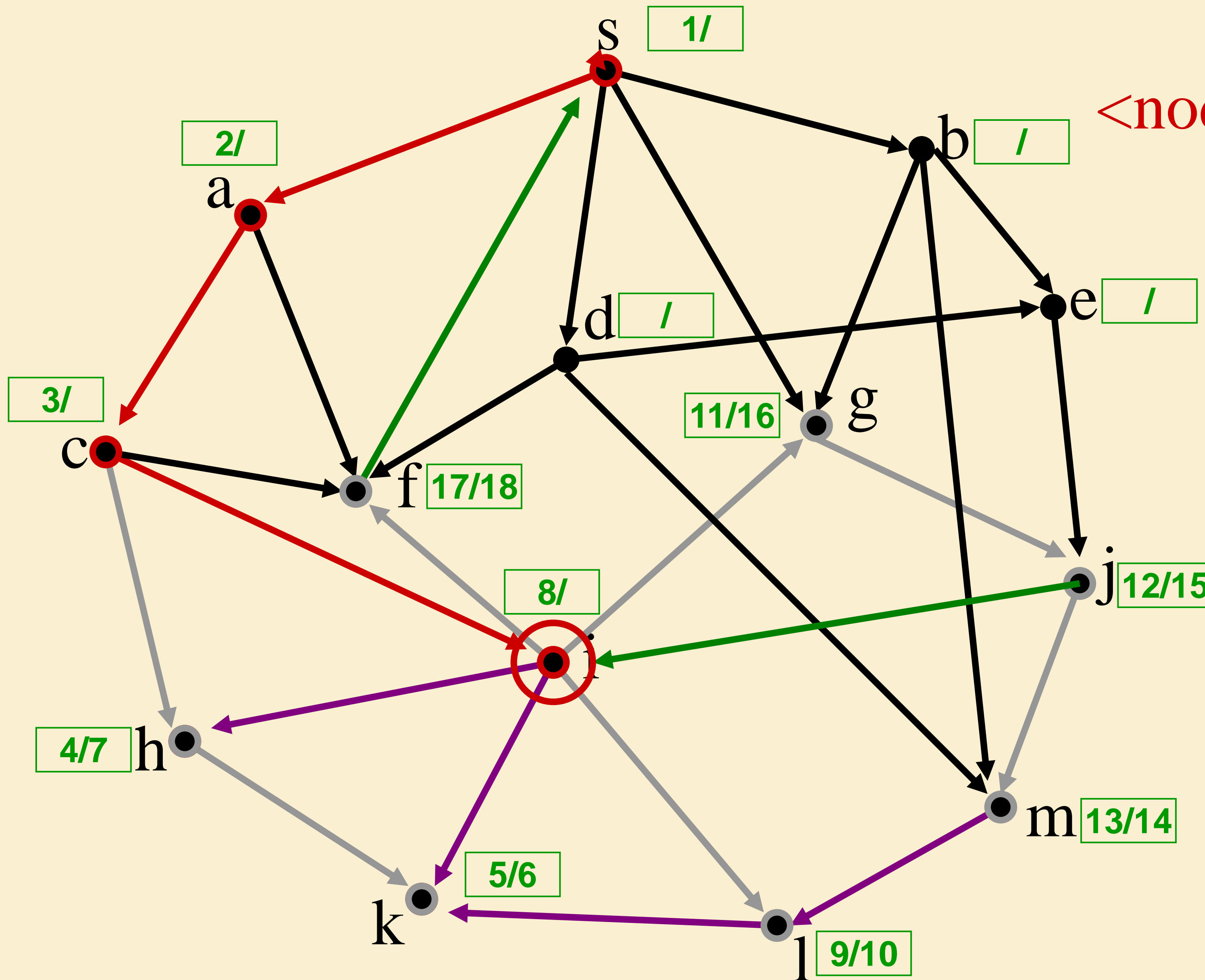


Found
Not Handled
Stack
<node,# edges>

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

DFS

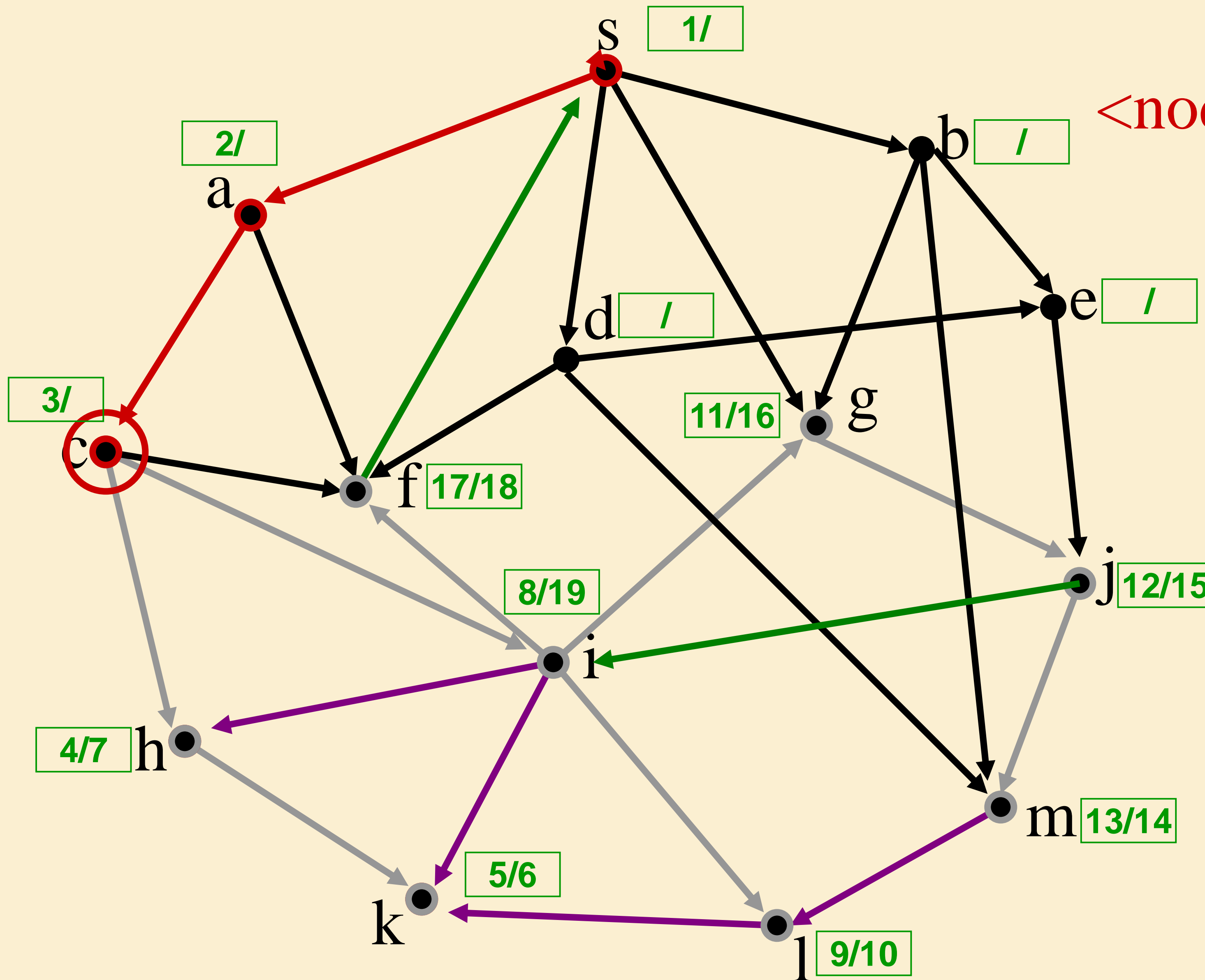
Found
Not Handled
Stack
<node,# edges>



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

DFS

Found
Not Handled
Stack
<node,# edges>

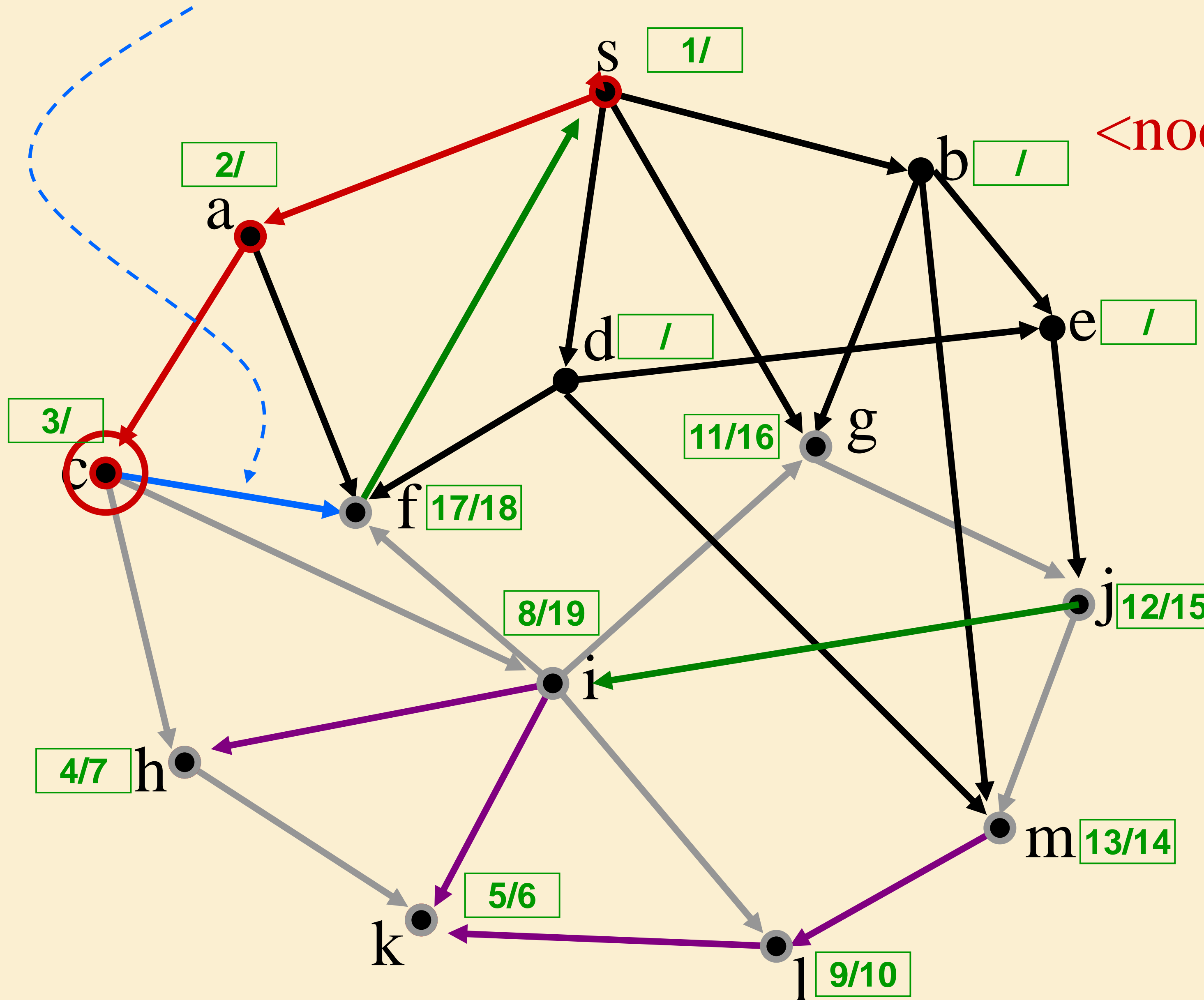


c,2
a,1
s,1

DFS

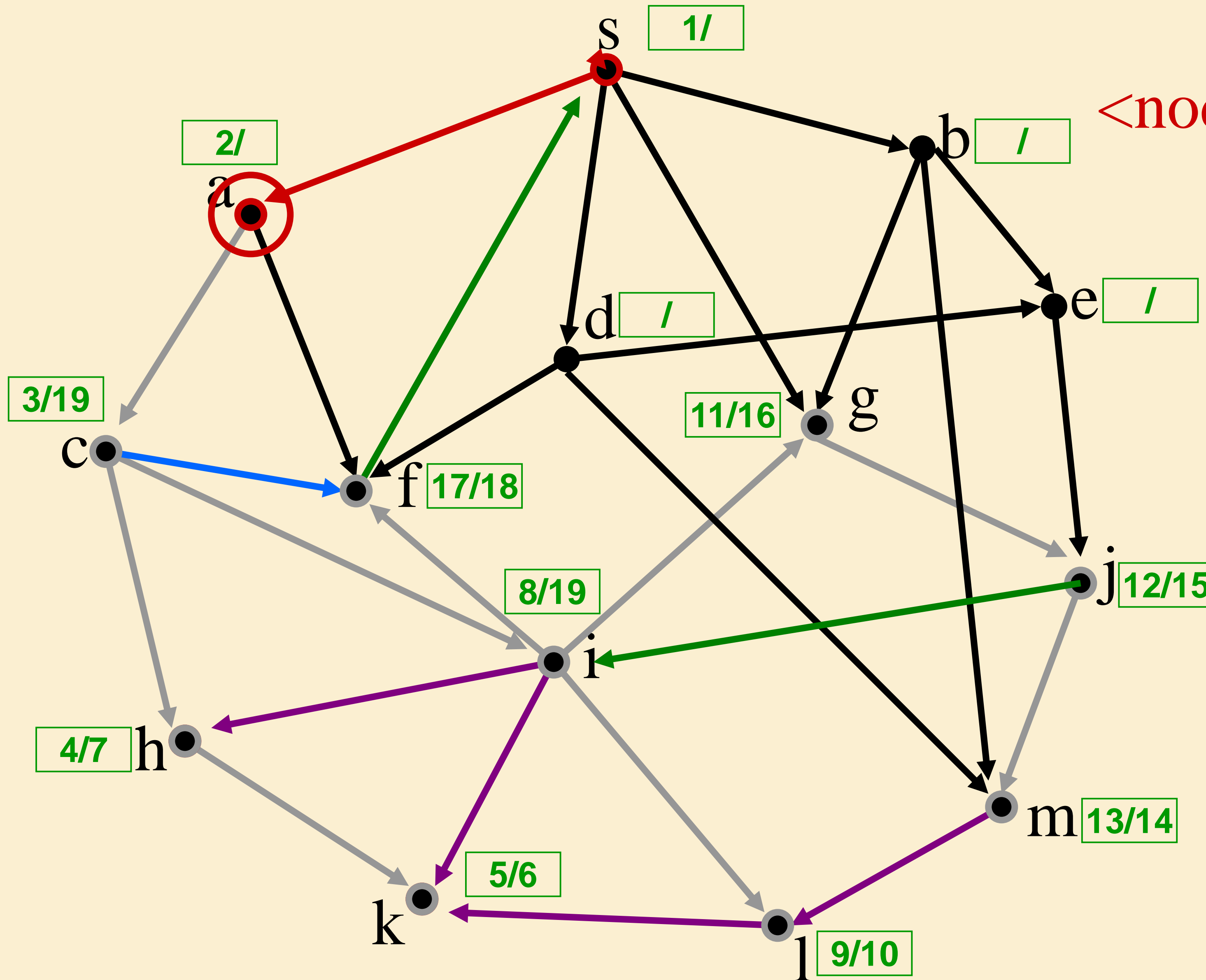
Forward Edge: d[f]>d[c]

Found
Not Handled
Stack
<node,# edges>



c,3
a,1
s,1

DFS

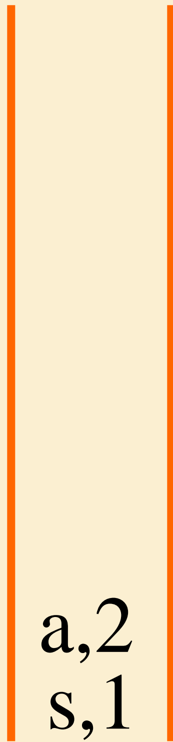


Found
Not Handled
Stack
<node,# edges>

a,1
s,1

DFS

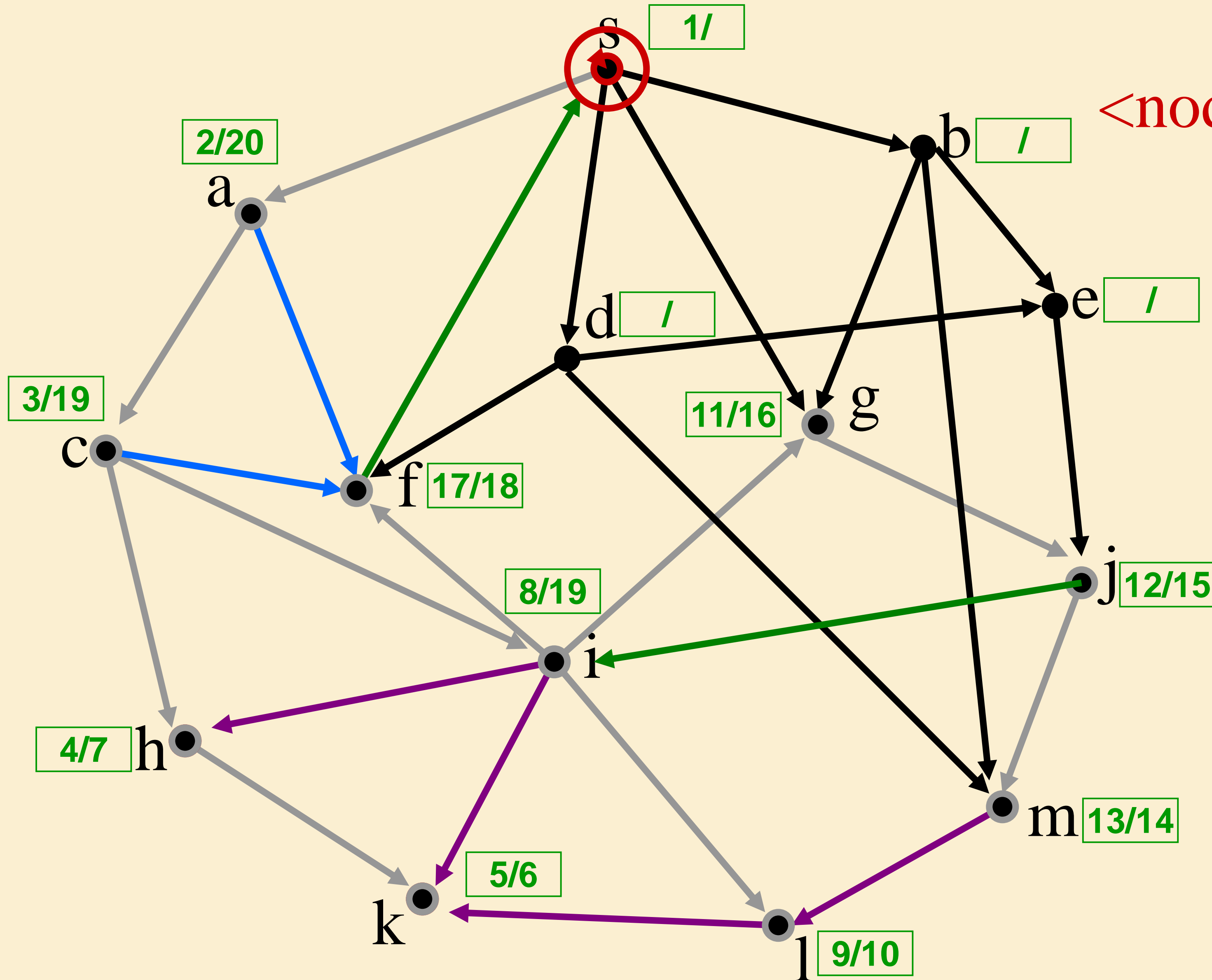
Found
Not Handled
Stack
<node,# edges>



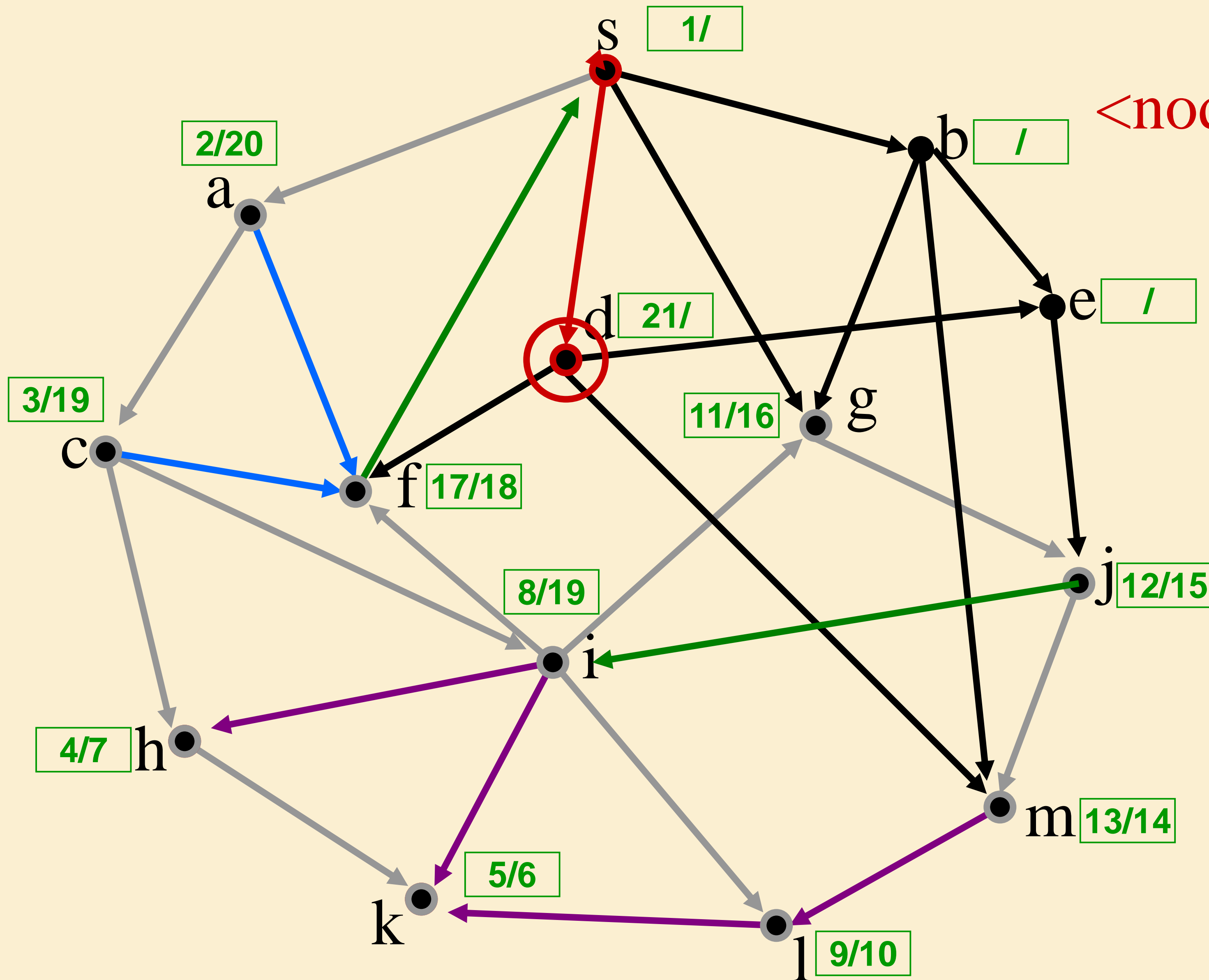
s 1/
a 2/
b /
c 3/19
d /
e /
f 17/18
g 11/16
h 4/7
i 8/19
j 12/15
k 5/6
l 9/10
m 13/14

DFS

Found
Not Handled
Stack
<node,# edges>



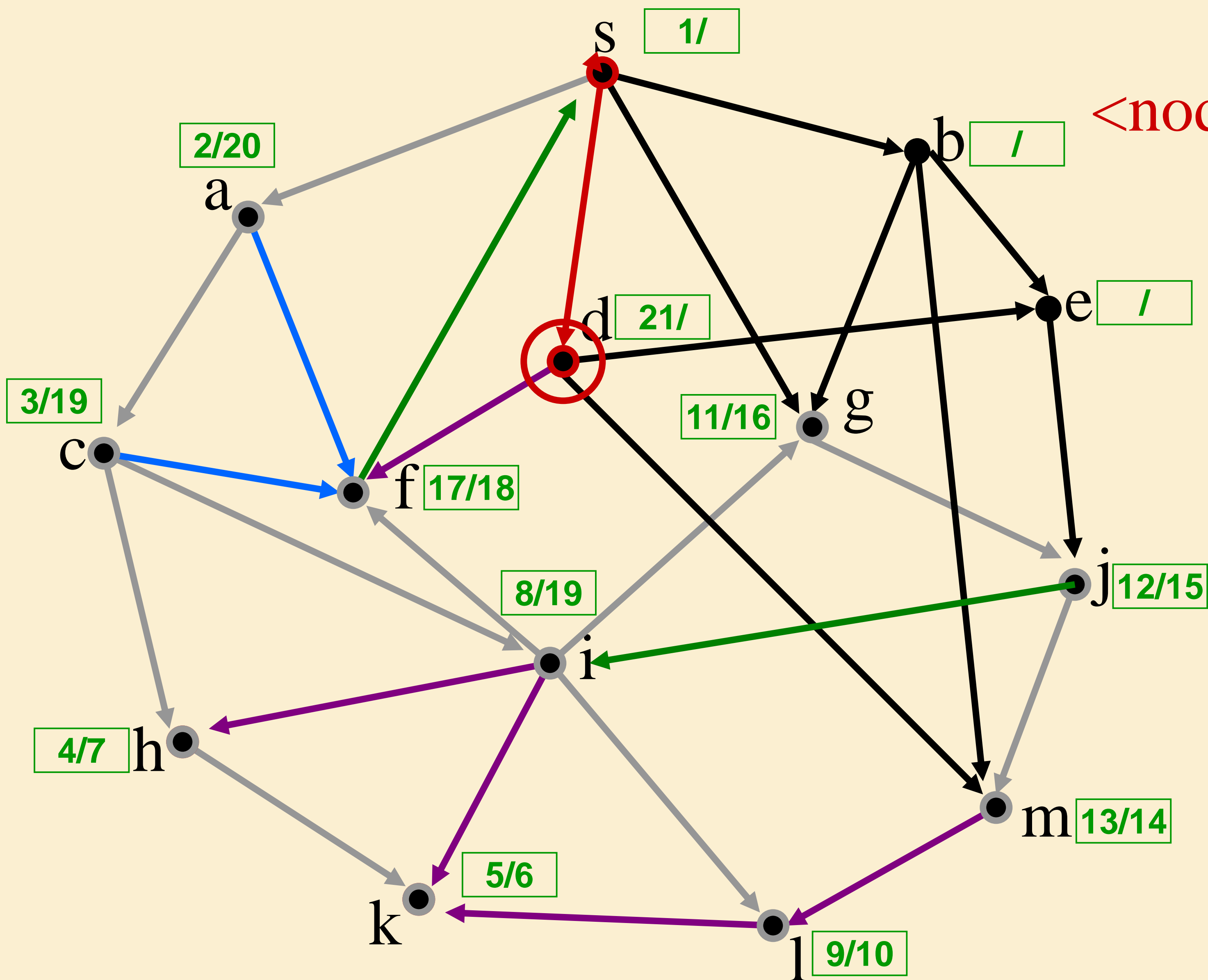
DFS



Found
Not Handled
Stack
<node,# edges>

d,0
s,2

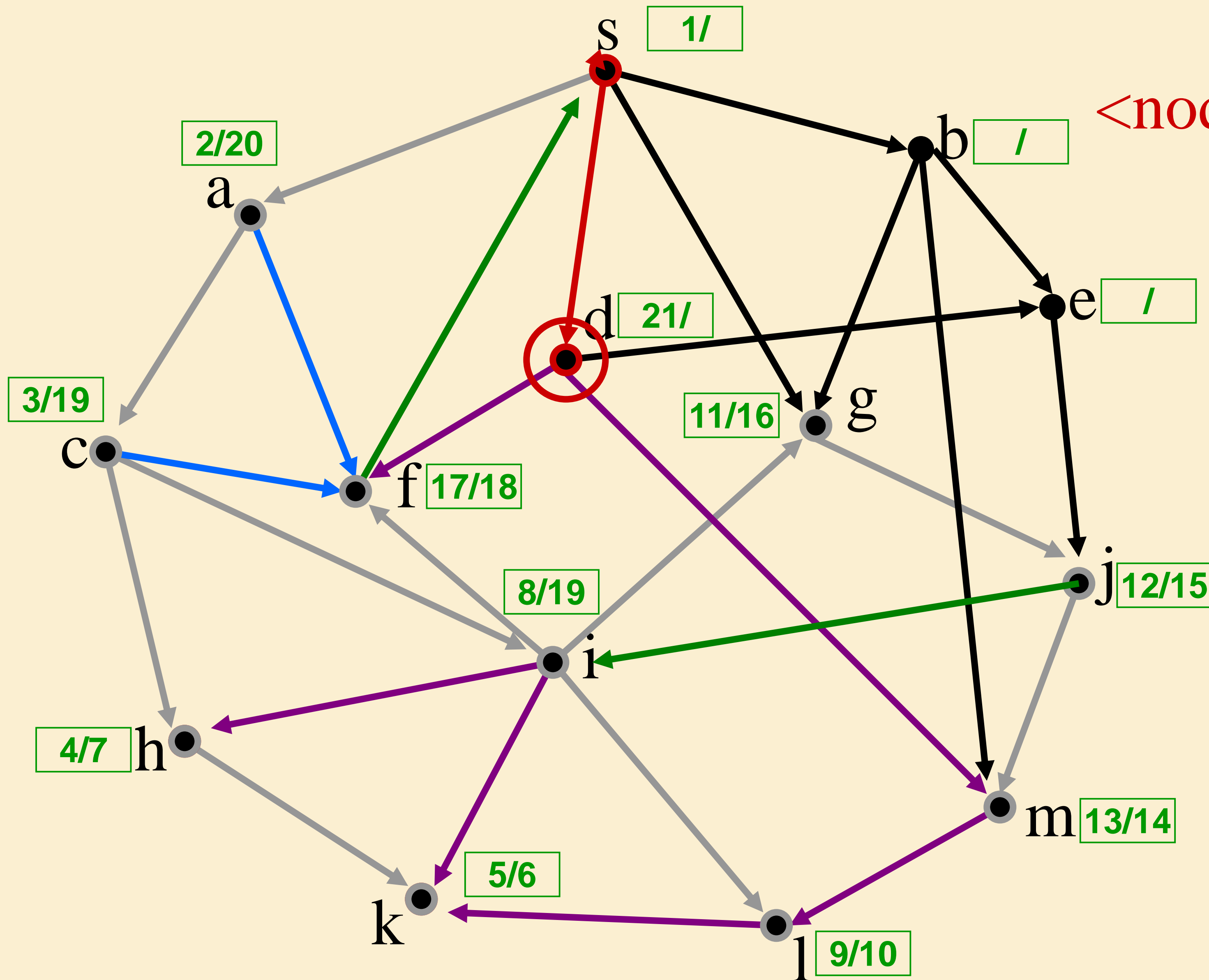
DFS



Found
Not Handled
Stack
<node,# edges>

d,1
s,2

DFS



Found
Not Handled
Stack
<node,# edges>

d,2
s,2

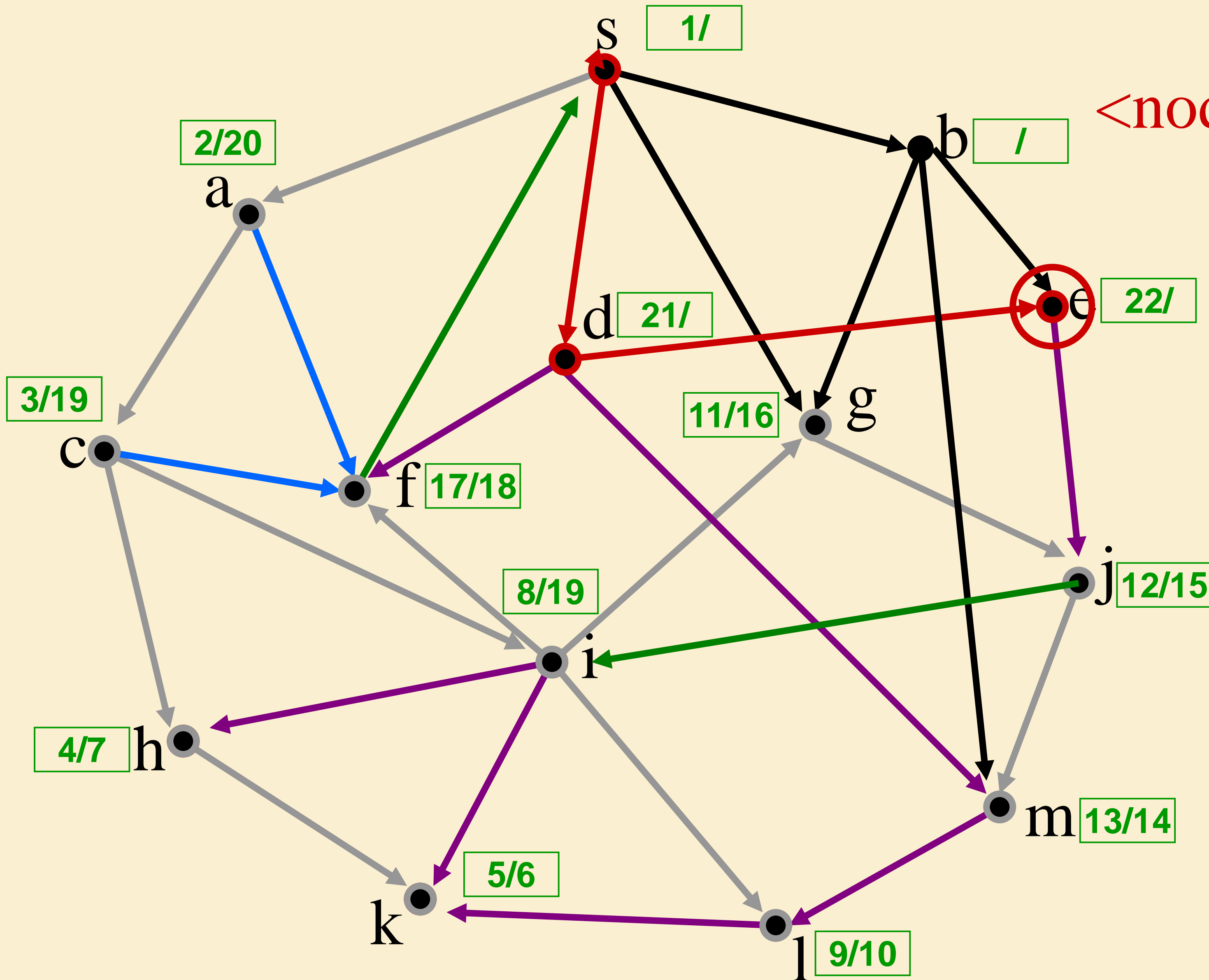
DFS

Found
Not Handled
Stack
<node,# edges>

e,0
d,3
s,2

s 1/
a 2/20
b /
c 3/19
d 21/
e 22/
f 17/18
g 11/16
h 4/7
i 8/19
j 12/15
k 5/6
l 9/10
m 13/14

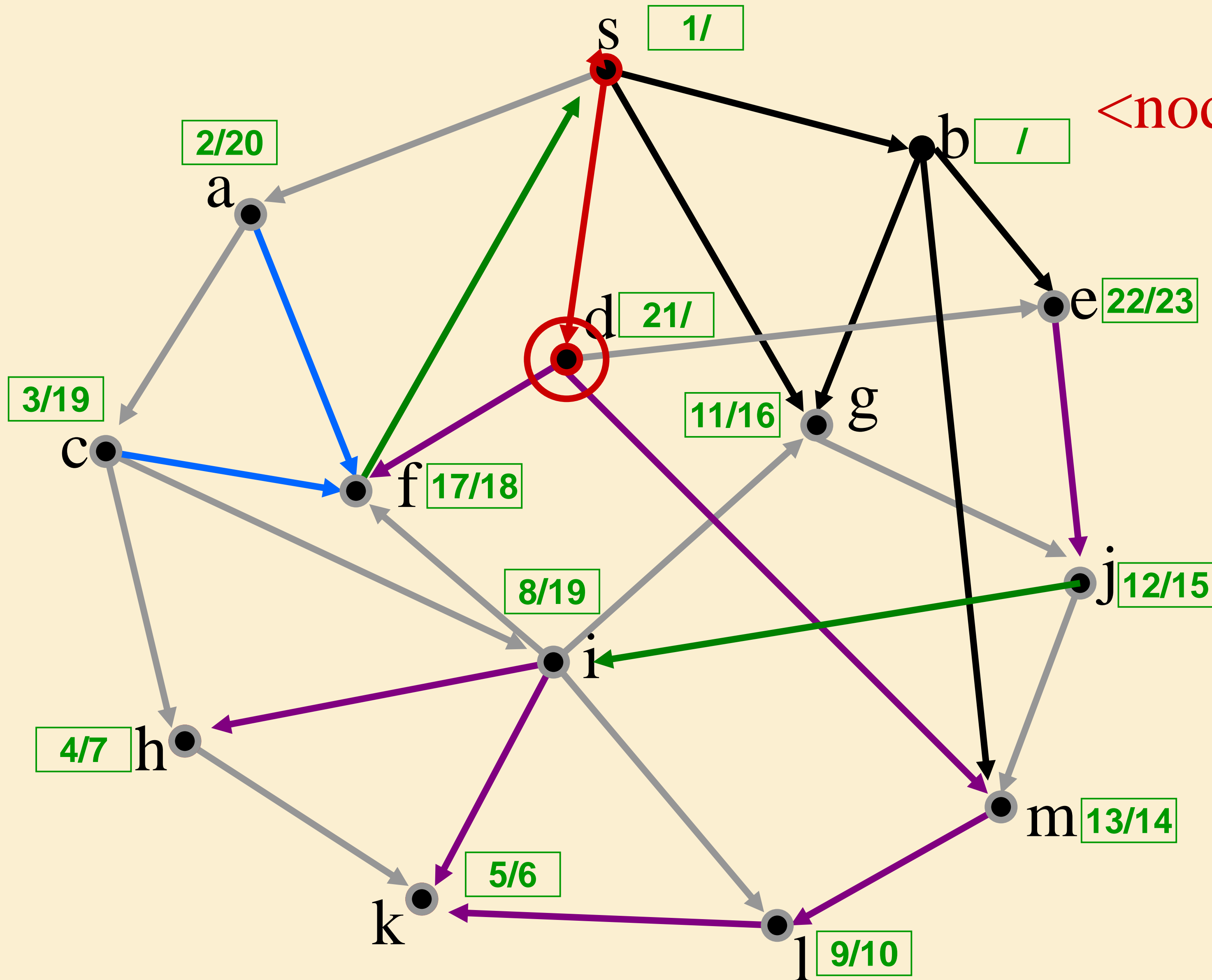
DFS



Found
Not Handled
Stack
<node,# edges>

e,1
d,3
s,2

DFS

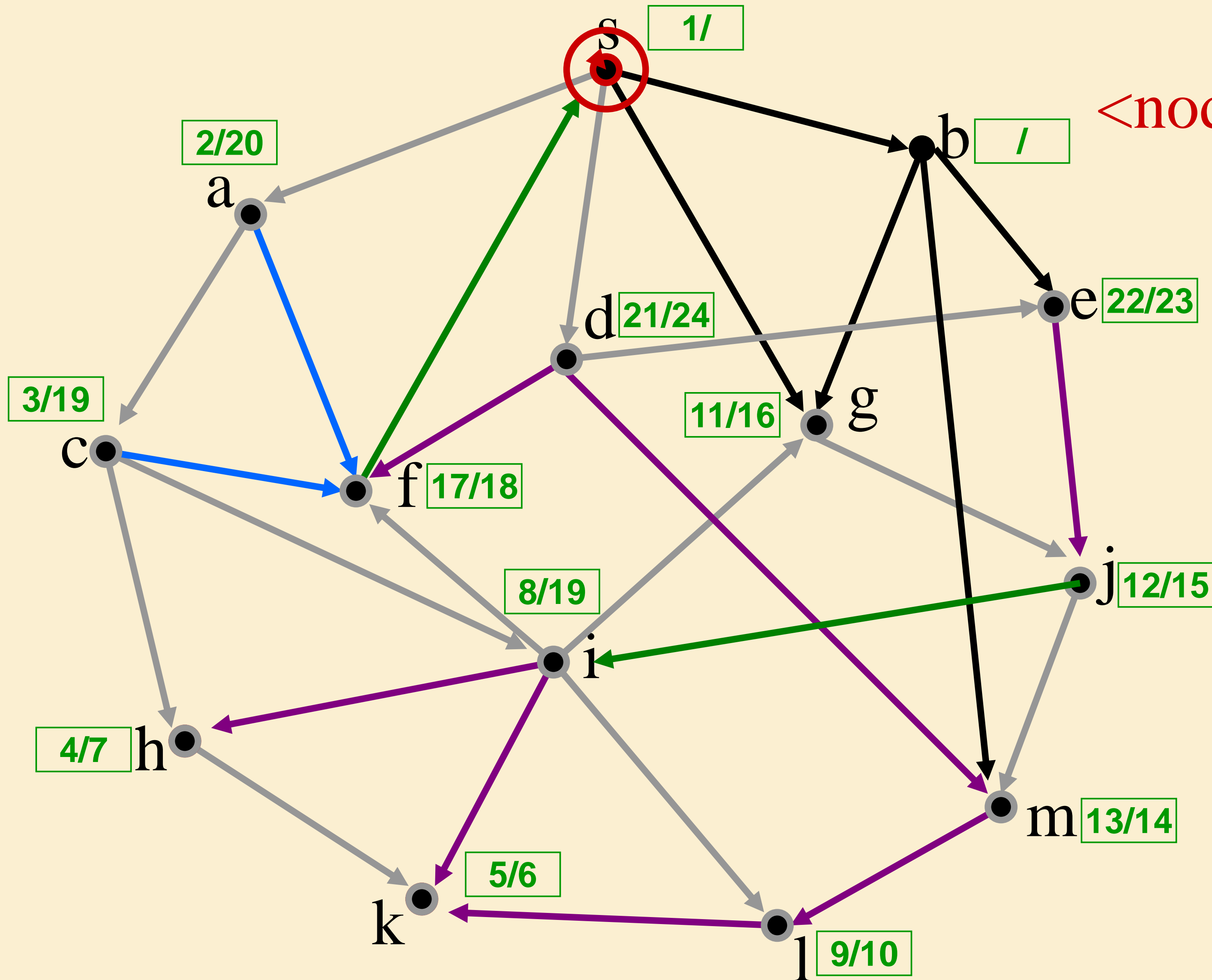


Found
Not Handled
Stack
<node,# edges>

d,3
s,2

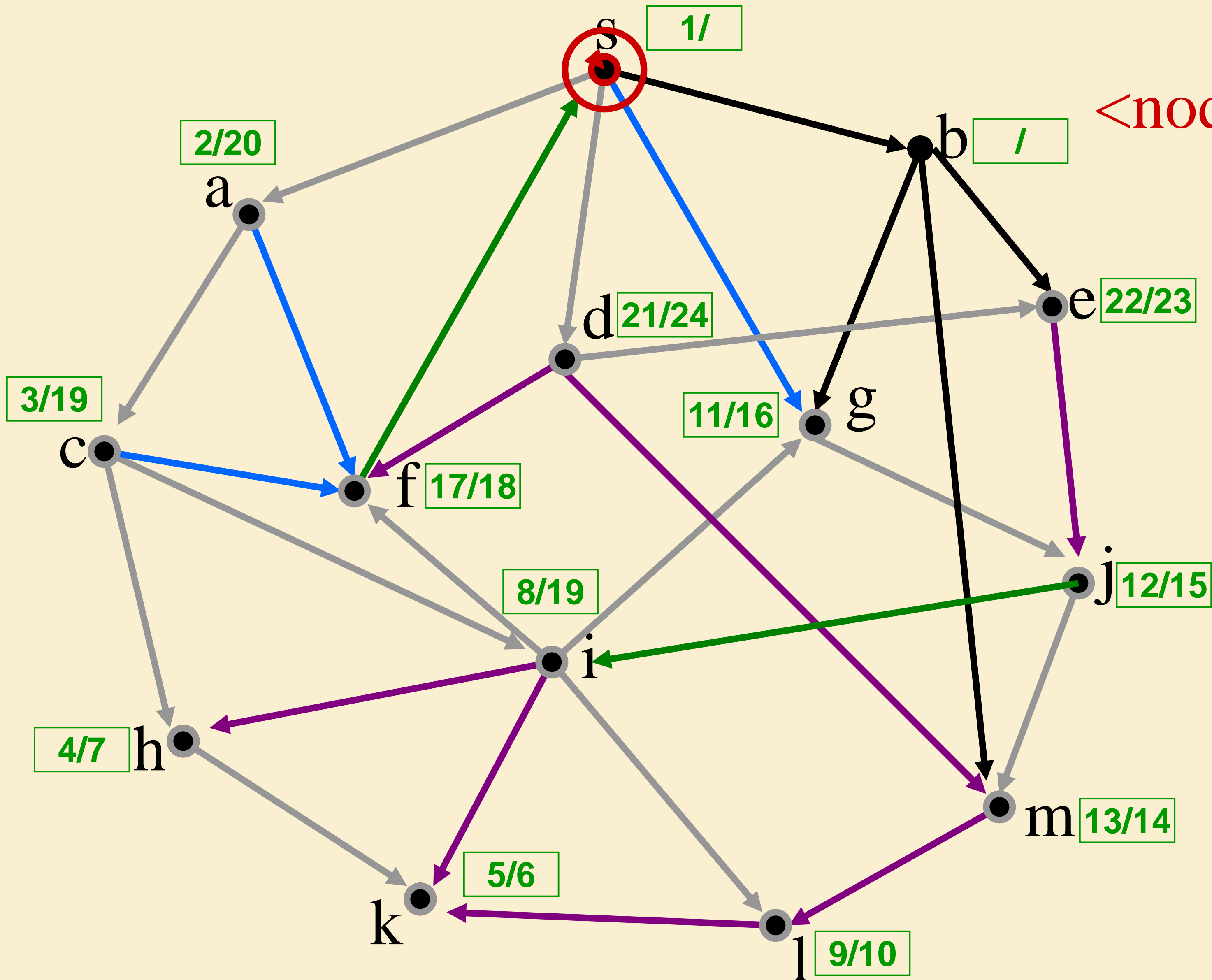
DFS

Found
Not Handled
Stack
<node,# edges>



s,2

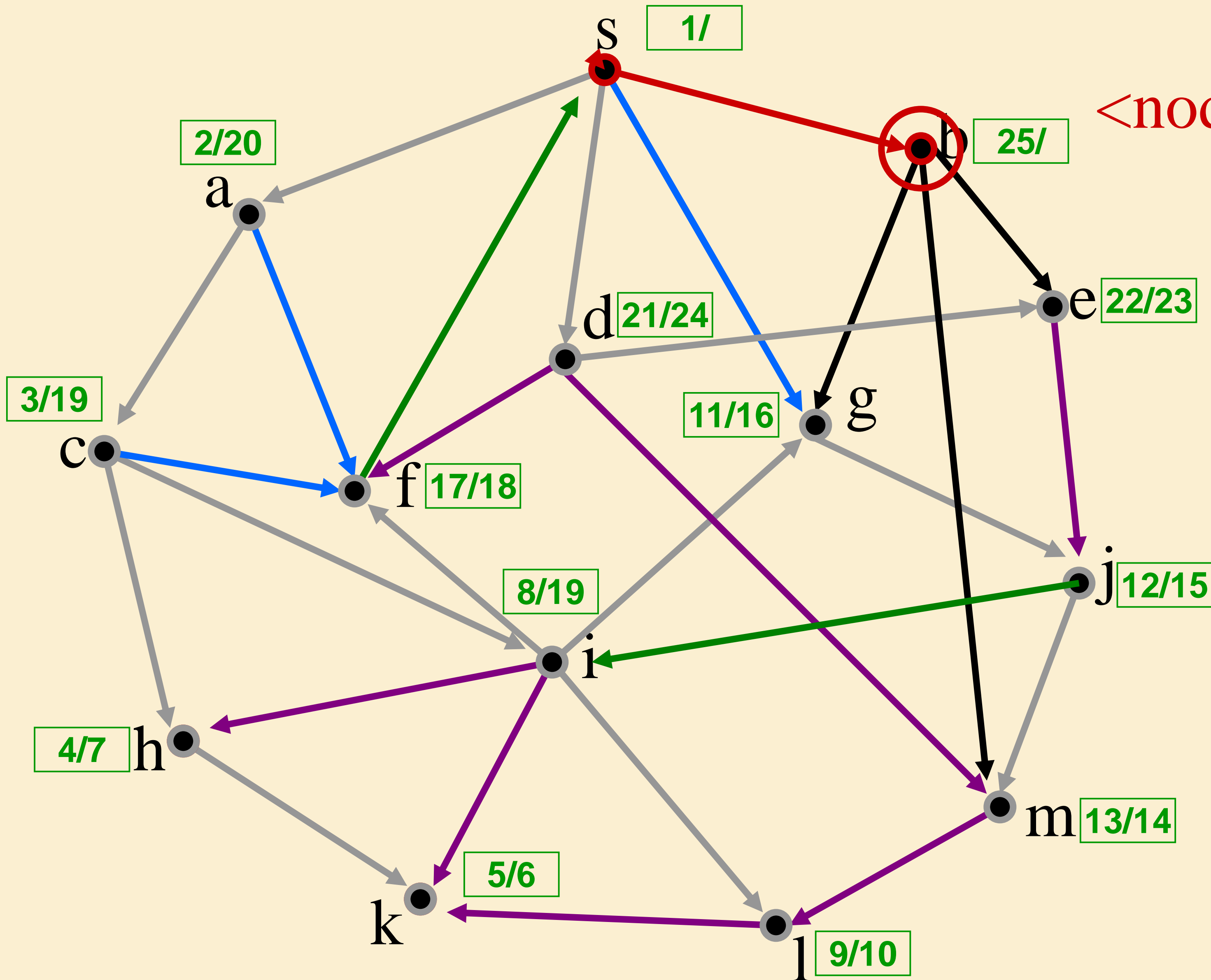
DFS



Found
Not Handled
Stack
<node,# edges>

s,3

DFS

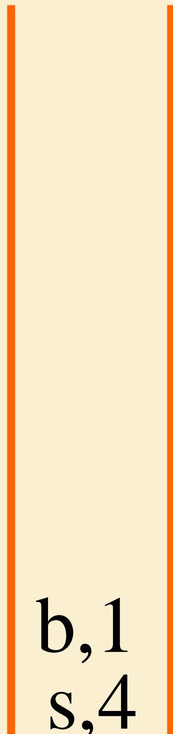


Found
Not Handled
Stack
<node,# edges>

b,0
s,4

DFS

Found
Not Handled
Stack
<node,# edges>



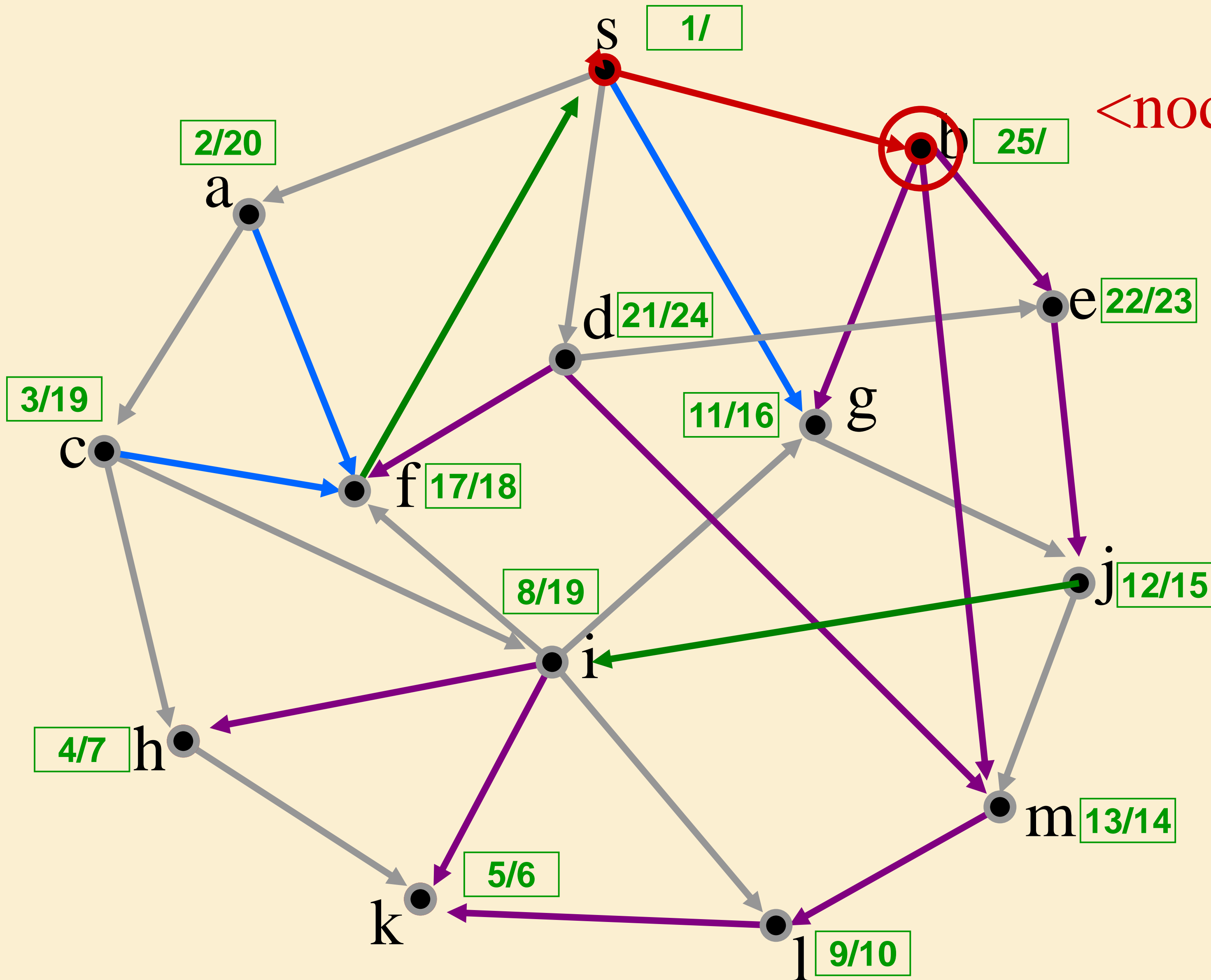
s 1/
a 2/20
b 25/
c 3/19
d 21/24
e 22/23
f 17/18
g 11/16
h 4/7
i 8/19
j 12/15
k 5/6
l 9/10
m 13/14

DFS

Found
Not Handled
Stack
<node,# edges>

s 1/
a 2/20
b 25/
c 3/19
d 21/24
e 22/23
f 17/18
g 11/16
h 4/7
i 8/19
j 12/15
k 5/6
l 9/10
m 13/14

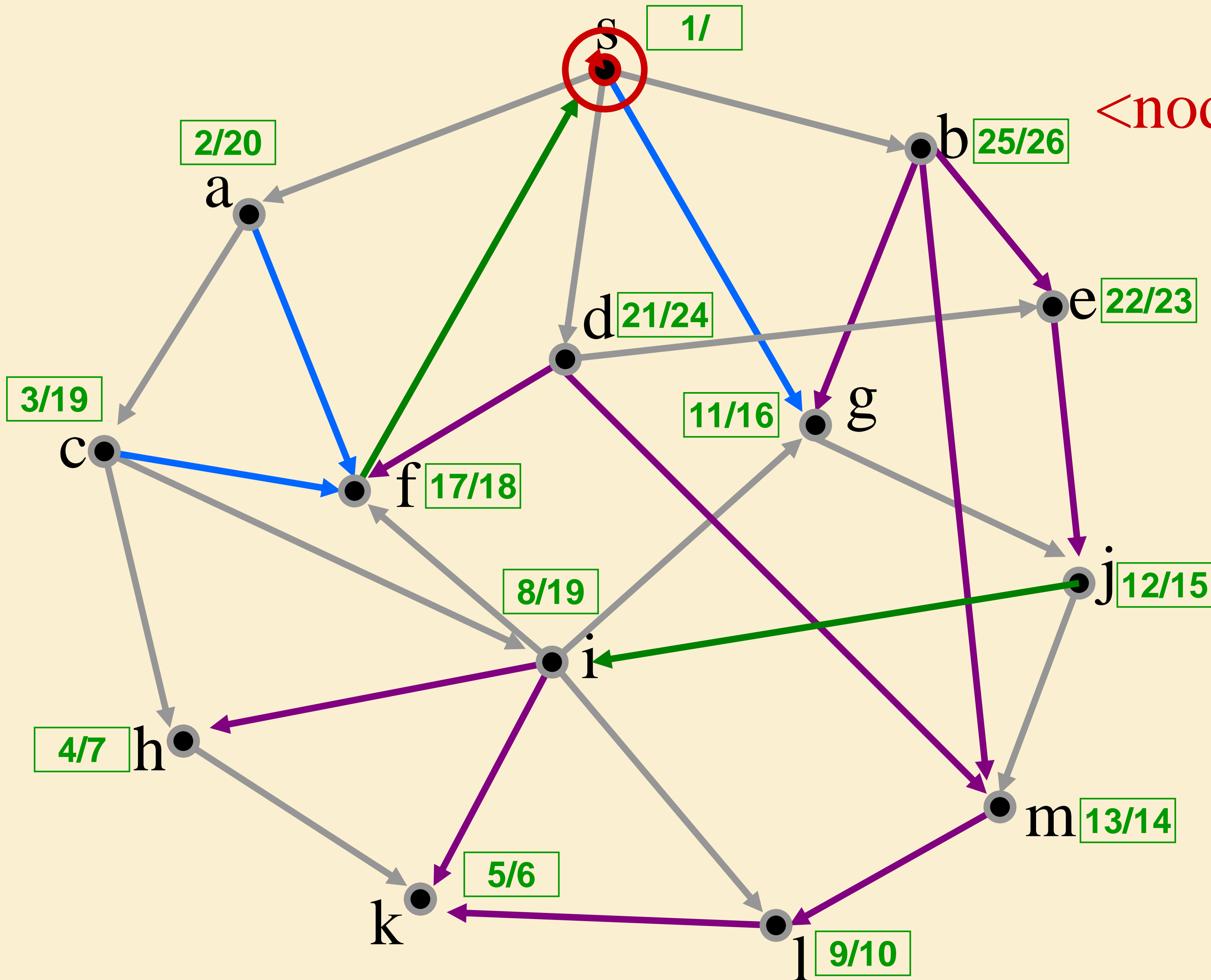
DFS



Found
Not Handled
Stack
<node,# edges>

b,3
s,4

DFS

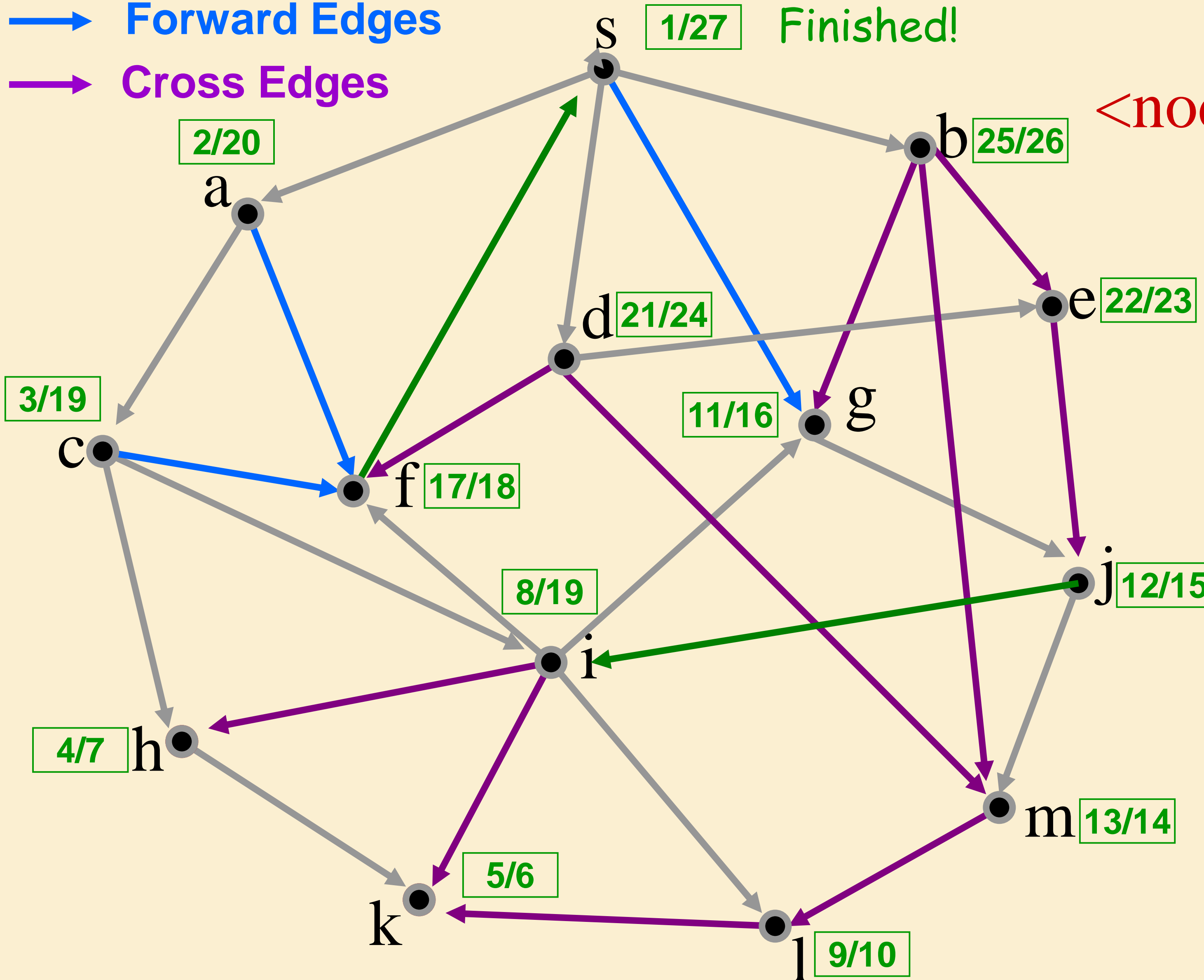


Found
Not Handled
Stack
<node,# edges>

s,4

DFS

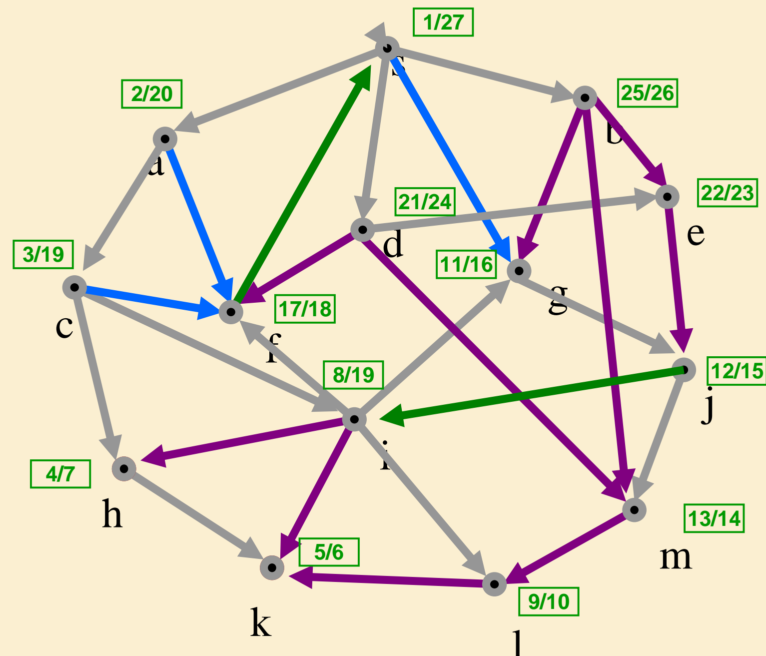
Tree Edges
Back Edges
Forward Edges
Cross Edges



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

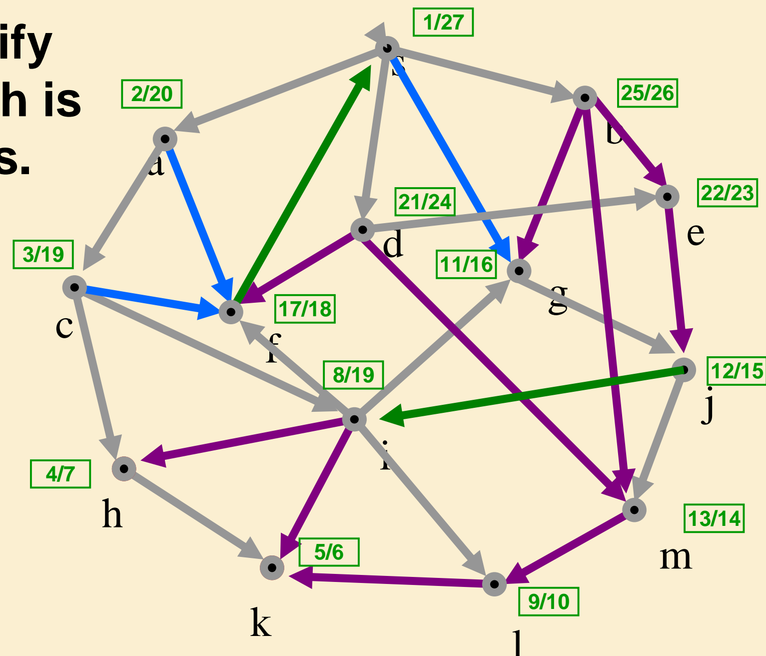
1. ***Tree edges***: Edge (*u*, *v*) is a **tree edge** if *v* was **black** when (*u*, *v*) traversed.

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.



Depth-First Search Algorithm

DFS(G)

```
1  for each vertex u ∈ V[G]
2        do color[u] ← BLACK
3           π[u] ← NIL
4  time ← 0
5  for each vertex u ∈ 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] ← RED              ▷BLACK vertex u has just been discovered.
2  time ← time + 1
3  d[u] ← time
4  for each v ∈ Adj[u]         ▷ Explore edge (u, v).
5       do if color[v] = BLACK
6             then π[v] ← u
7                  DFS-VISIT(v)
8  color[u] ← GRAY             ▷    GRAY u; it is finished.
9  f[u] ← time ← time + 1
```

Depth-First Search Algorithm

DFS(G)

```
1  for each vertex u ∈ V[G]
2       do color[u] ← BLACK
3          π[u] ← NIL
4  time ← 0
5  for each vertex u ∈ V[G]
6       do if color[u] = BLACK
7             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

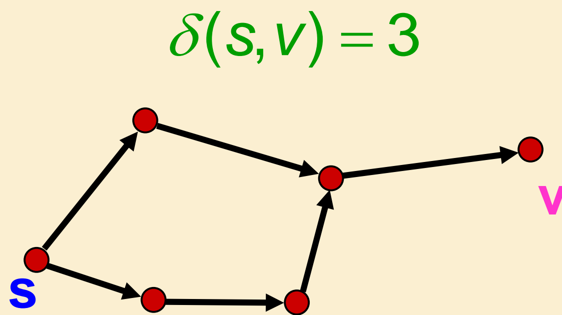
```
1  color[u] ← RED              ▷BLACK vertex u has just been discovered.
2  time ← time + 1
3  d[u] ← time
4  for each v ∈ Adj[u]         ▷ Explore edge (u, v).
5       do if color[v] = BLACK
6             then π[v] ← u
7                  DFS-VISIT(v)
8  color[u] ← GRAY             ▷   GRAY u; it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

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

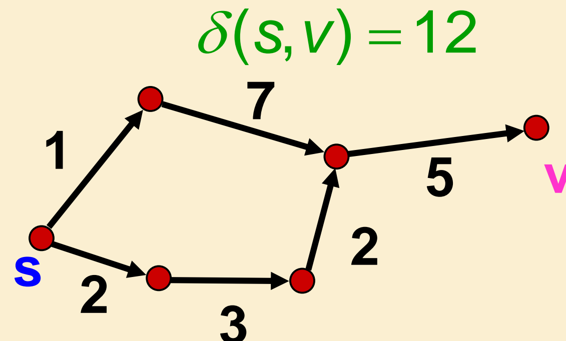
Back to Shortest Path

- 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?
- Input:

Directed Graph $G = (V, E)$

Edge weights $w : E \to \mathbb{R}$

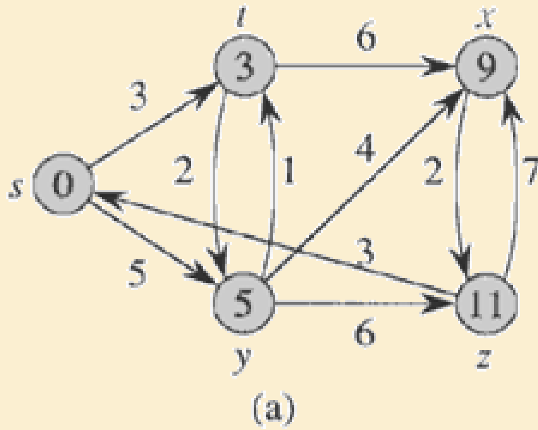
Weight of path $p = < v_0, v_1, \ldots, v_k > \; = \sum_{i=1}^{k} w(v_{i-1}, v_i)$

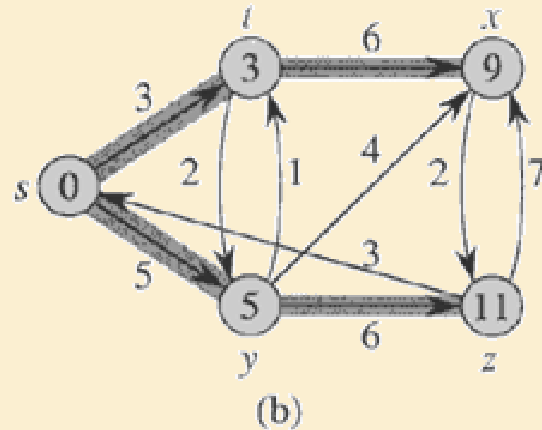
Shortest-path weight from u to v :

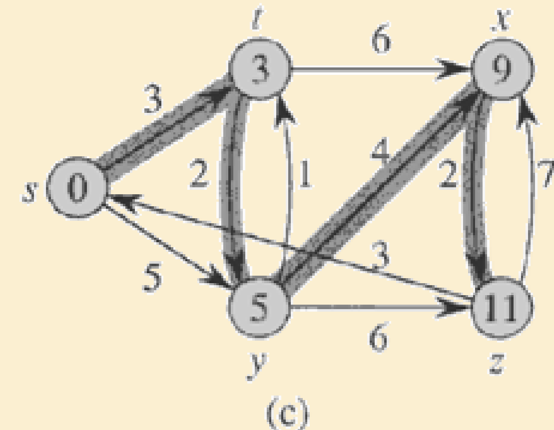
$$\delta(u,v) = \begin{cases} \min\{w(p) : u \overset{p}{\to} \cdots \to v\} & \text{if } \exists \text{ a path } u \to \cdots \to v, \\ \infty & \text{otherwise.} \end{cases}$$

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)

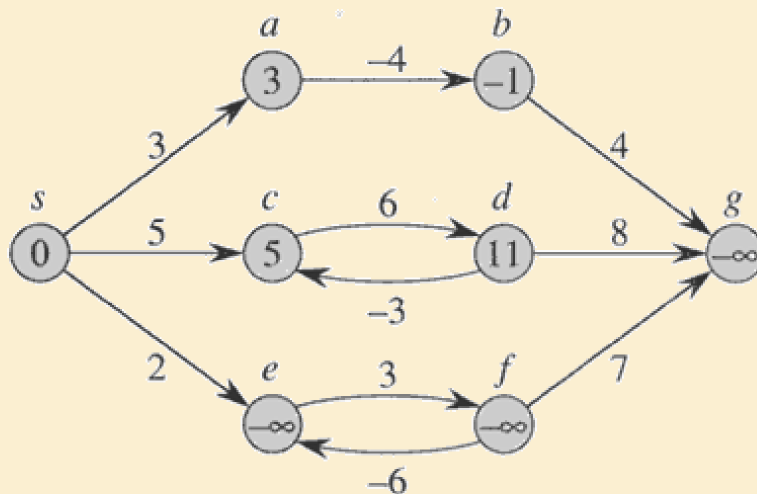
- **Single-destination shortest-paths problem:** Find a shortest path to a given ***destination*** vertex *t* from each vertex *v*.

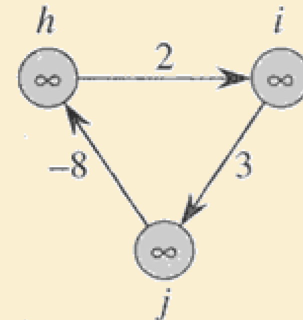
- **Single-pair shortest-path problem:** Find a shortest path from *u* to *v* for given vertices *u* and *v*.

- **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.





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:
  - $d[v] = \delta(s, v)$.
    - Initially, $d[v]=\infty$.
    - Reduce as algorithm progresses.
      But always maintain $d[v] \geq \delta(s, v)$.
    - Call d[v] a shortest-path estimate.
  - $\pi[v]$ = predecessor of v on a shortest path from s.
    - If no predecessor, $\pi[v]$ = 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]$\leftarrow\infty$

    π[v] $\leftarrow$ NIL

d[s] $\leftarrow$ 0

Relaxing an edge

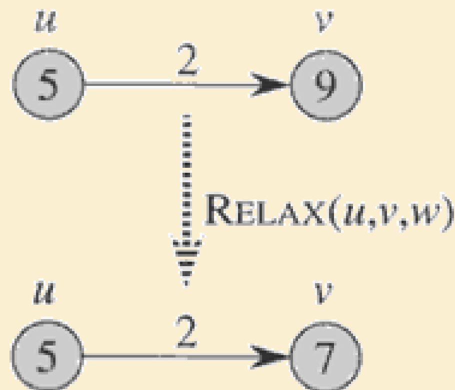
- Can we improve shortest-path estimate for v by going through u and taking (u,v)?

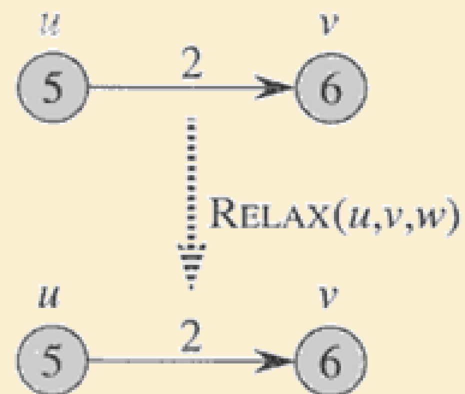
RELAX(u, v,w)

if d[v] > d[u] + w(u, v) then

d[v] ← d[u] + w(u, v)

π[v]← 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: 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)
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)
```

- 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

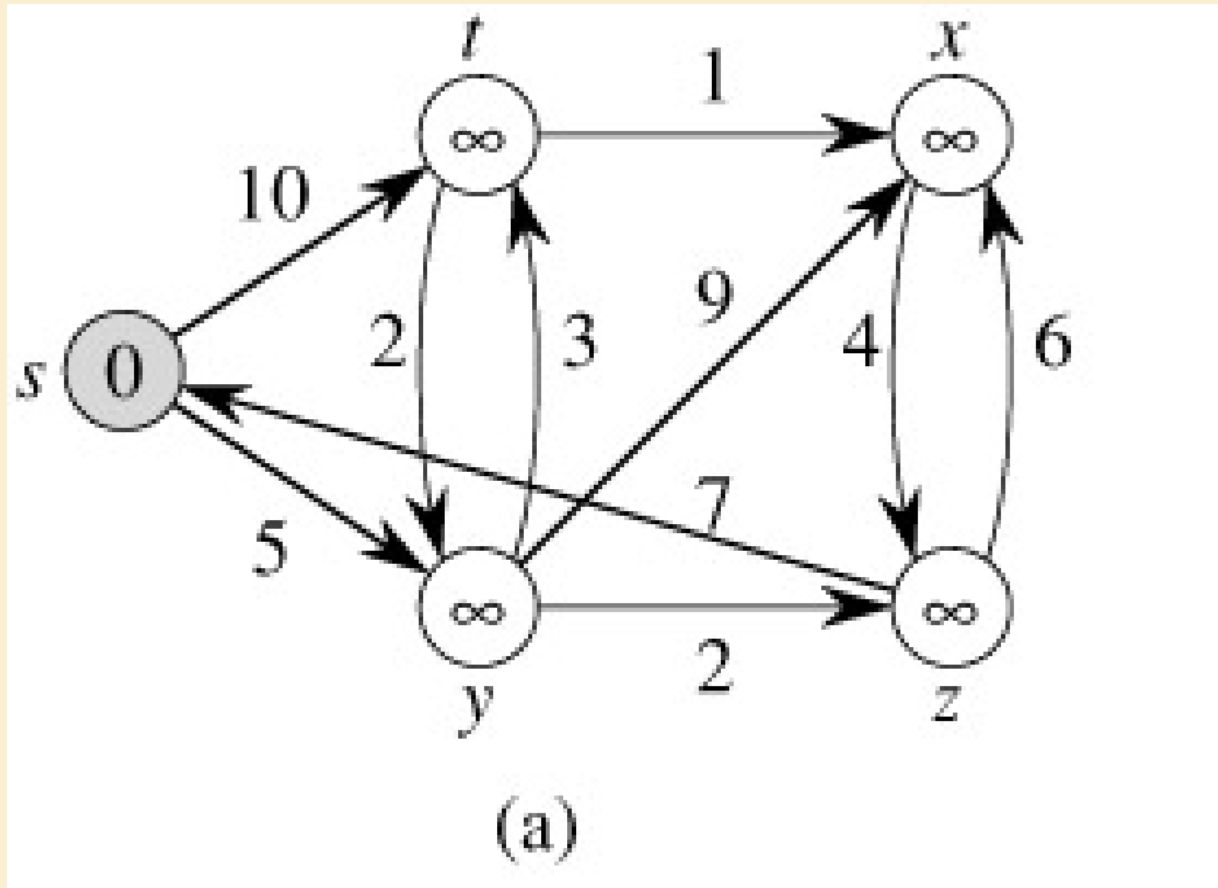
DIJKSTRA(G, w, s)
1 INITIALIZE-SINGLE-SOURCE(G, s) $O(V)$
2 $S \leftarrow \emptyset$
3 $Q \leftarrow V[G]$
4 **while** $Q \neq \emptyset$
5 **do** $u \leftarrow$ EXTRACT-MIN(Q) $O(\log V) \times O(V)$ iterations
6 $S \leftarrow S \cup \{u\}$
7 **for** each vertex $v \in Adj[u]$
8 **do** RELAX(u, v, w)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

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

Example

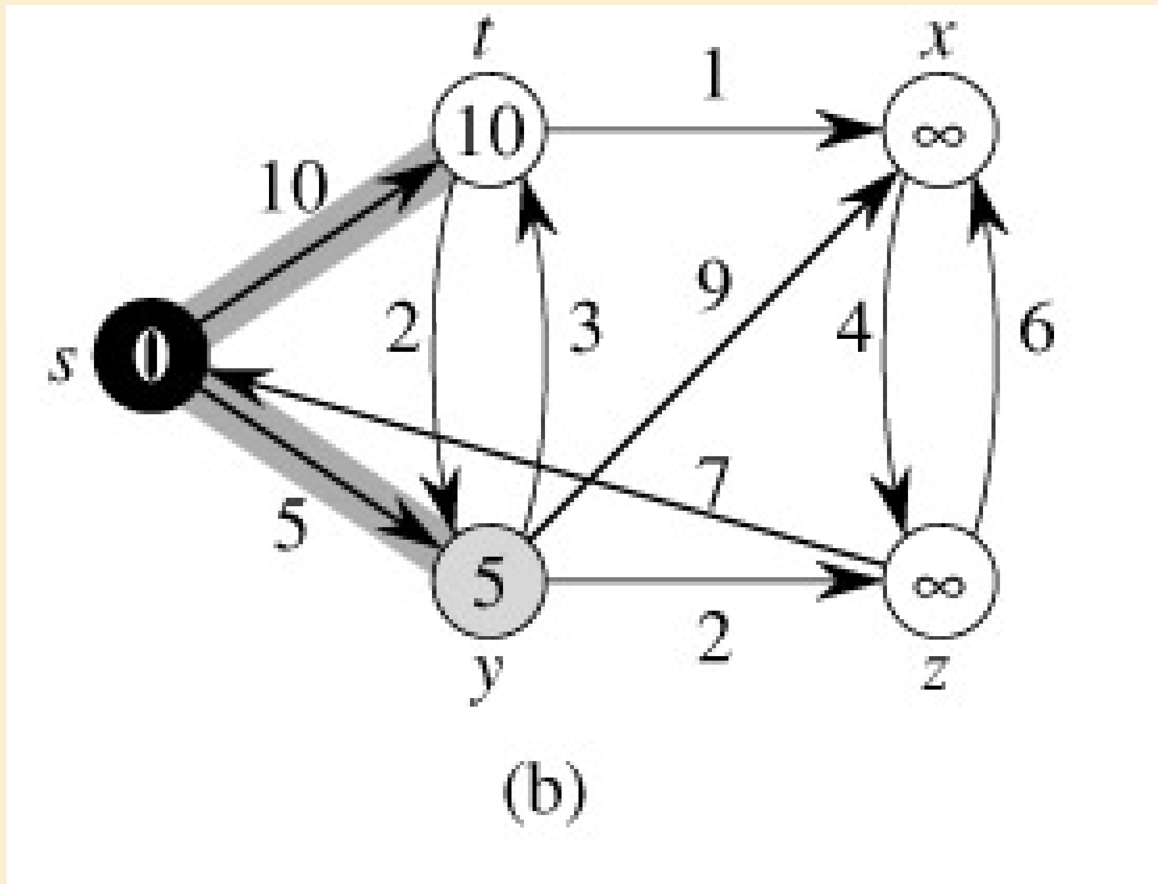
Key:

White	⇔	Not Found
Grey	⇔	Handling
Black	⇔	Handled

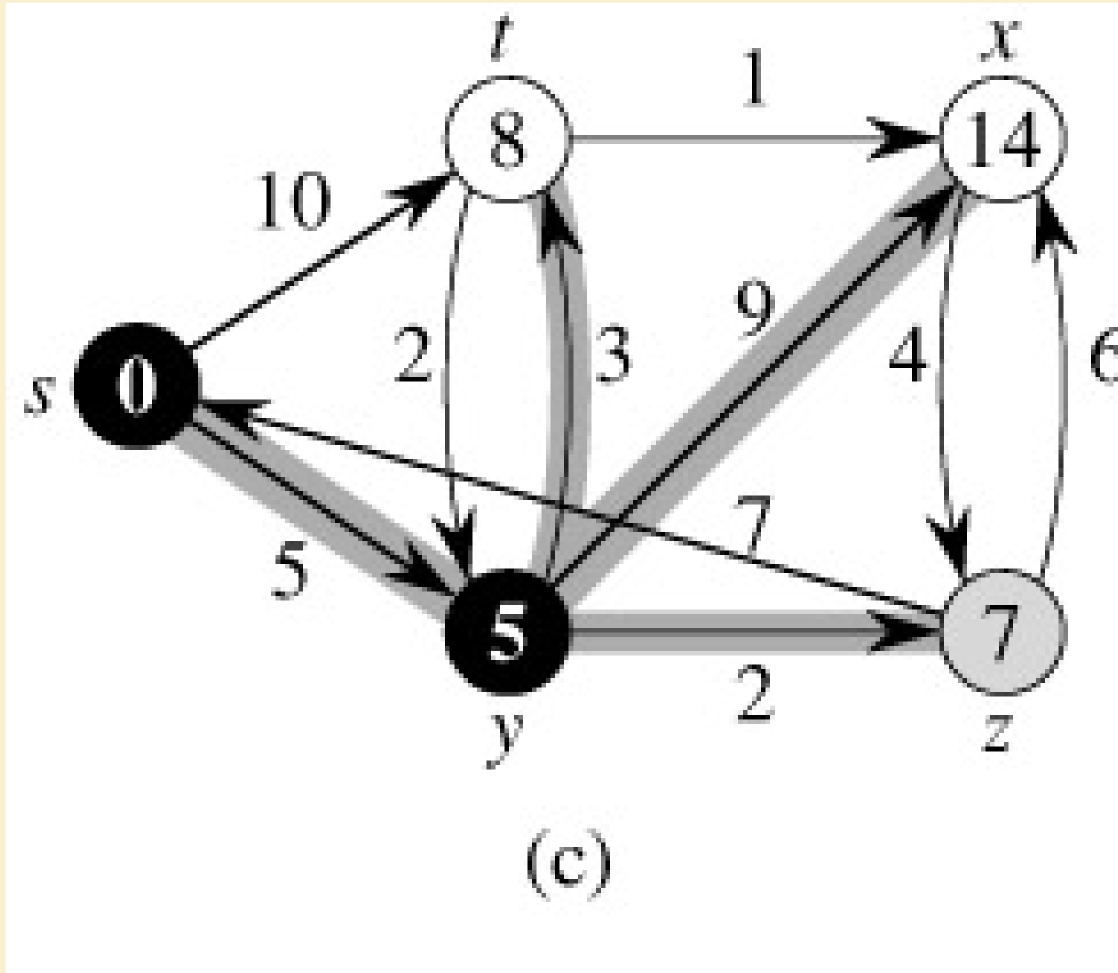


(a)

Example

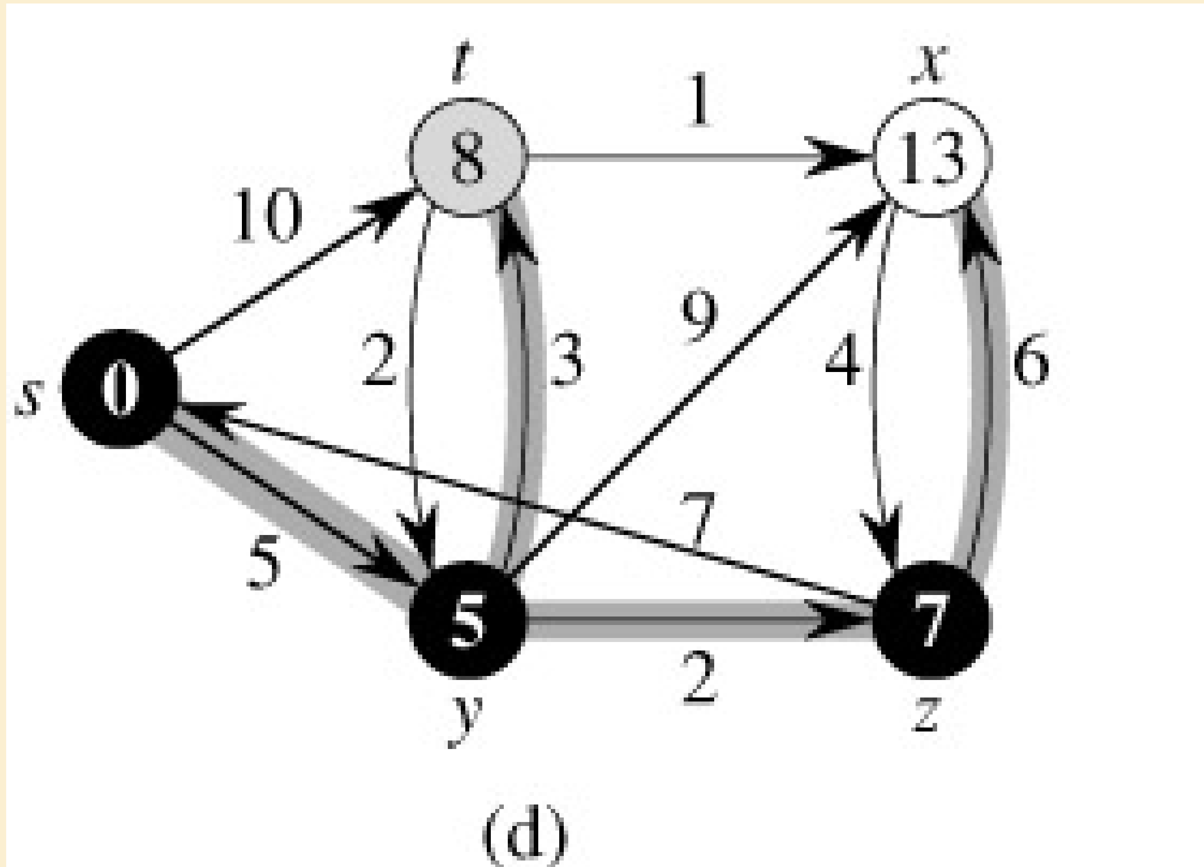


Example

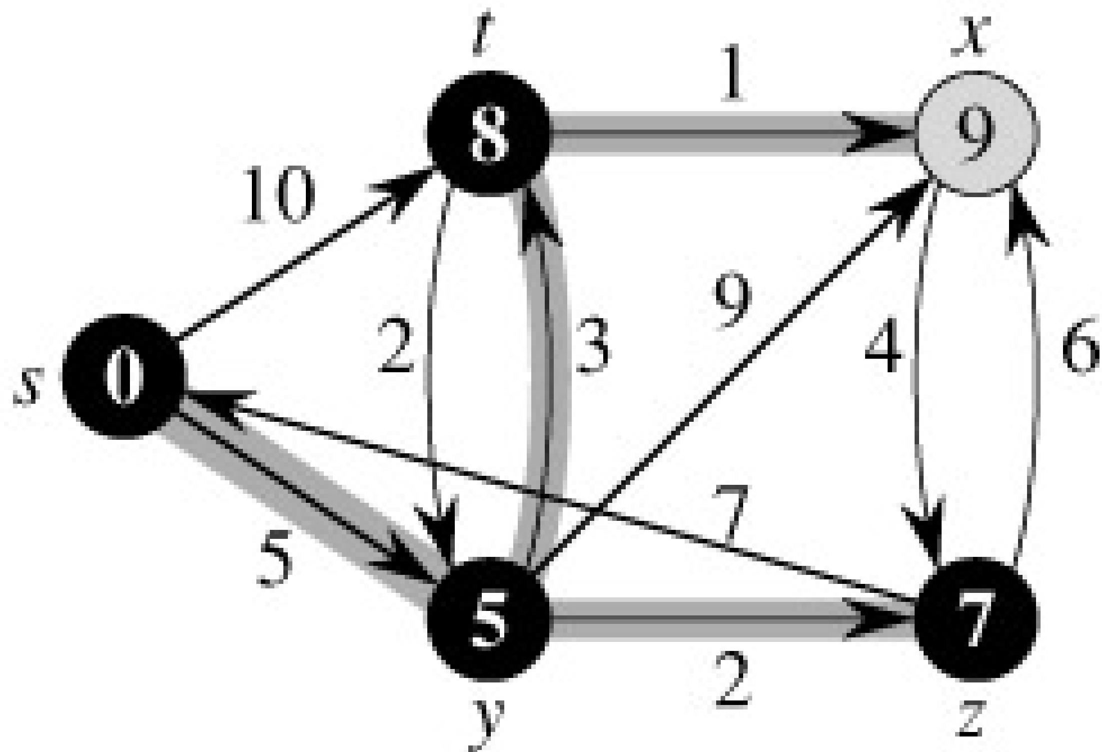


(c)

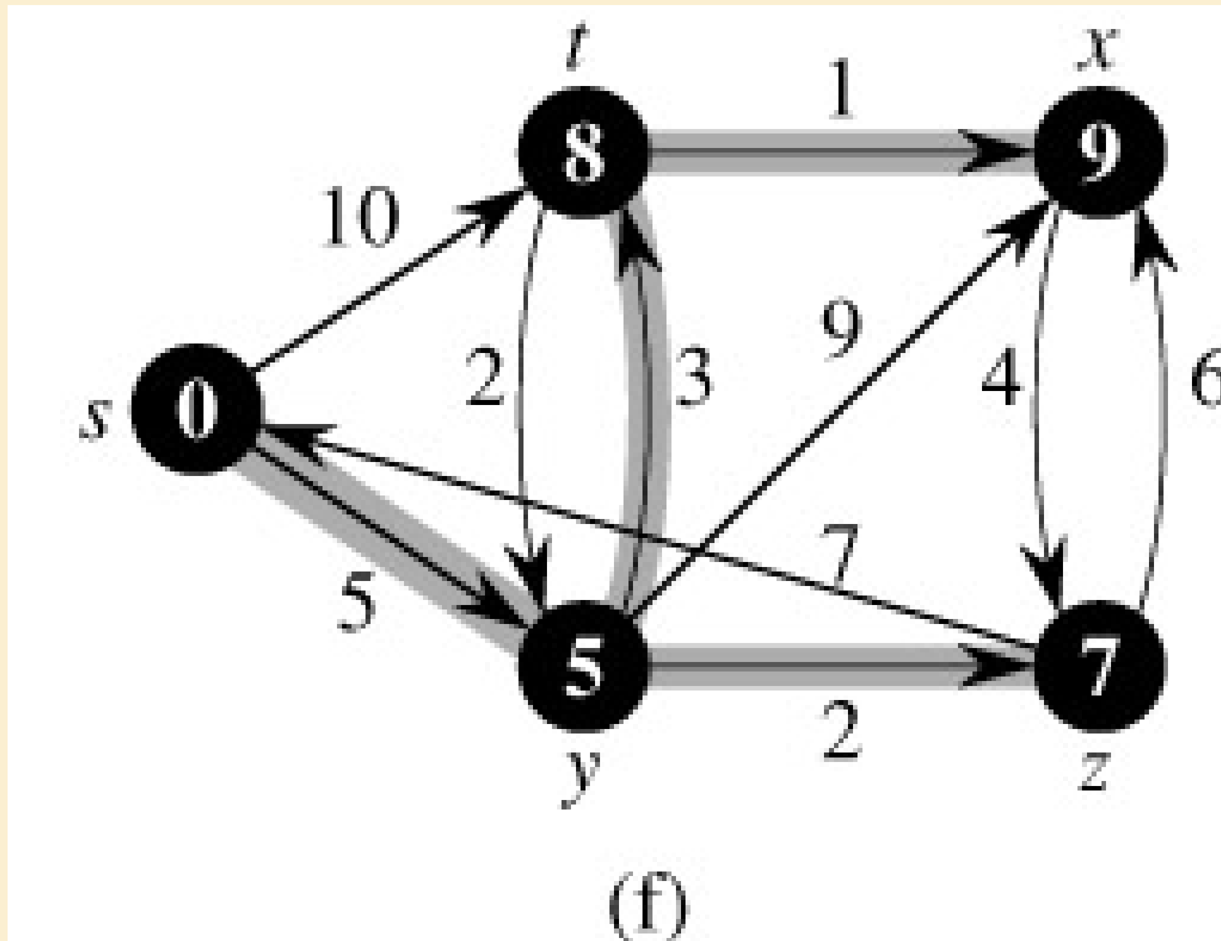
Example



Example



Example



(f)