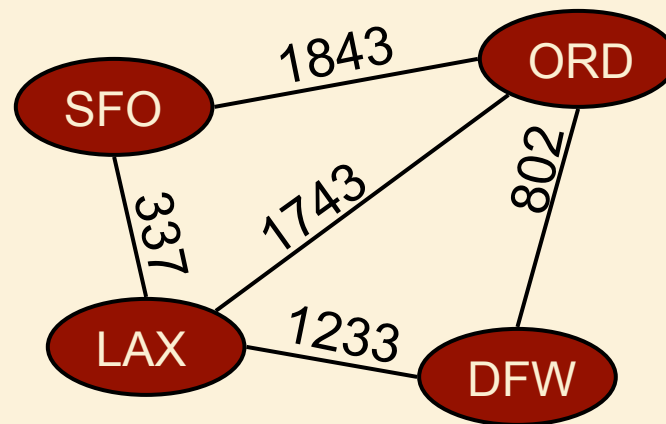
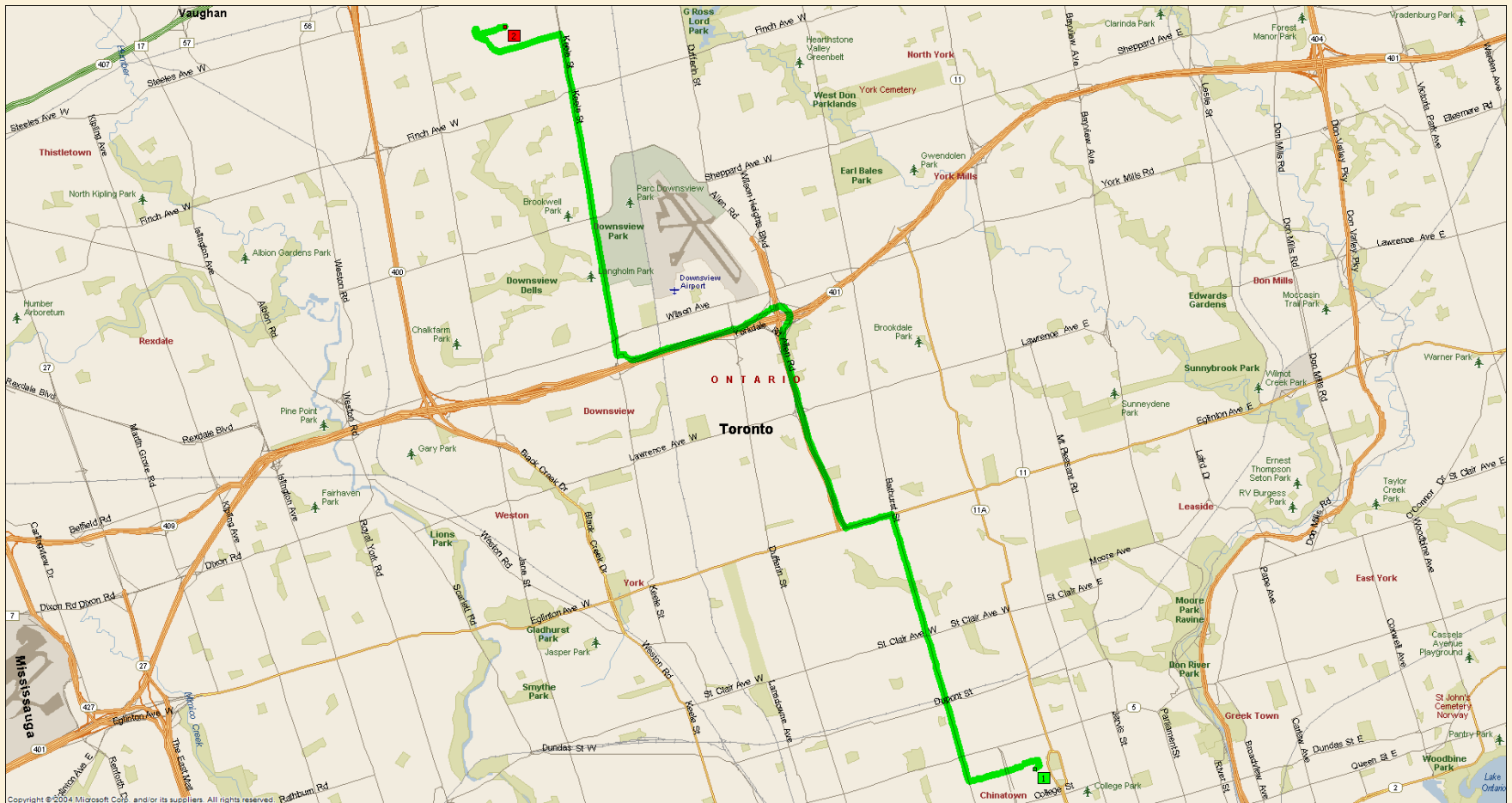


Graphs – Depth First Search



Graph Search Algorithms



Outline

- DFS Algorithm
- DFS Example
- DFS Applications

Outline

- **DFS Algorithm**
- DFS Example
- DFS Applications

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.
- ❑ Analogous to Euler tour for trees

➤ Used to help solve many graph problems, including

- ❑ Nodes that are reachable from a specific node v
- ❑ Detection of cycles
- ❑ Extraction of strongly connected components
- ❑ Topological sorts

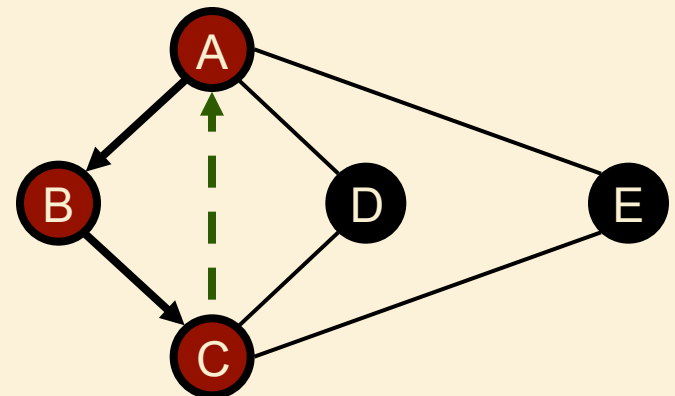
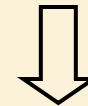
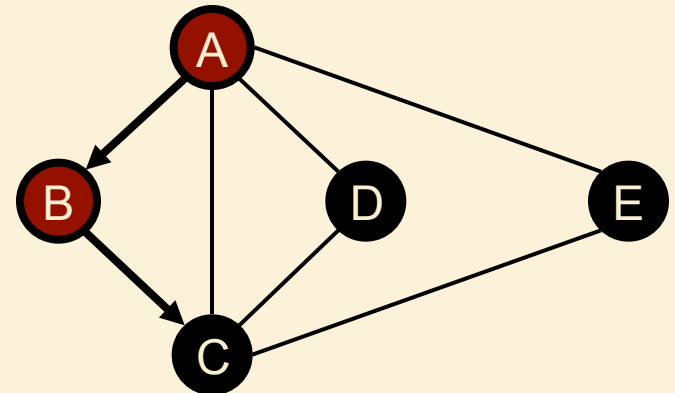
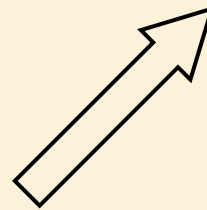
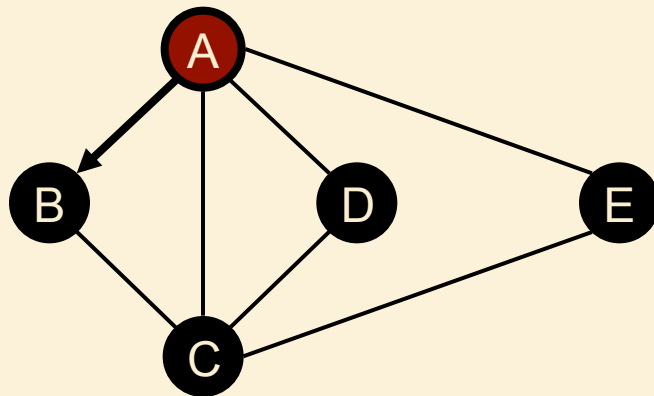
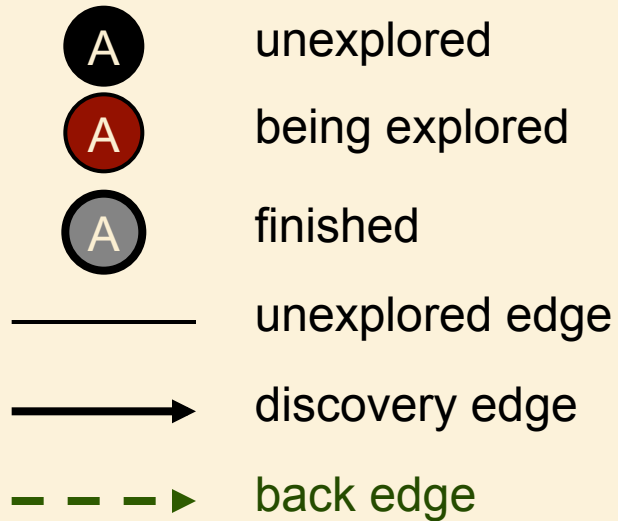
- [illegible]

Depth-First Search

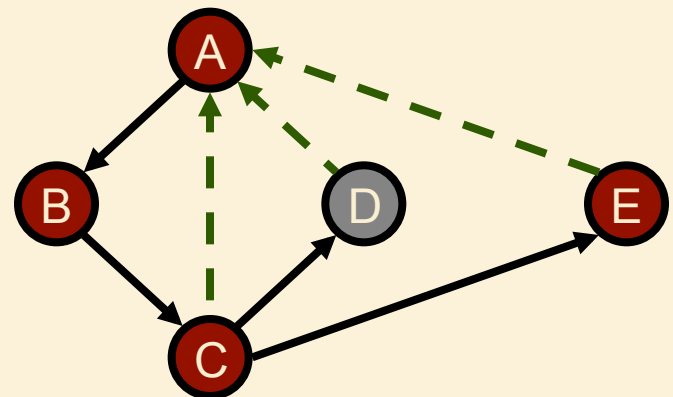
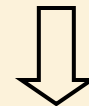
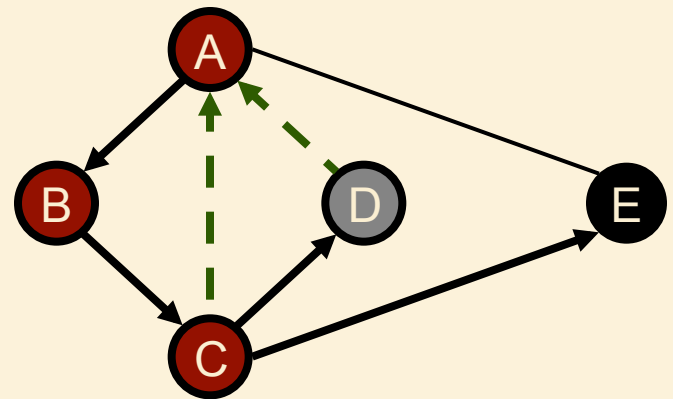
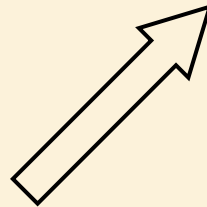
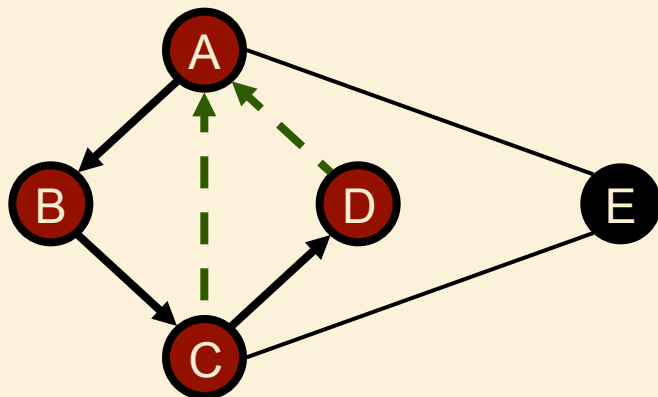
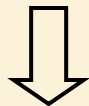
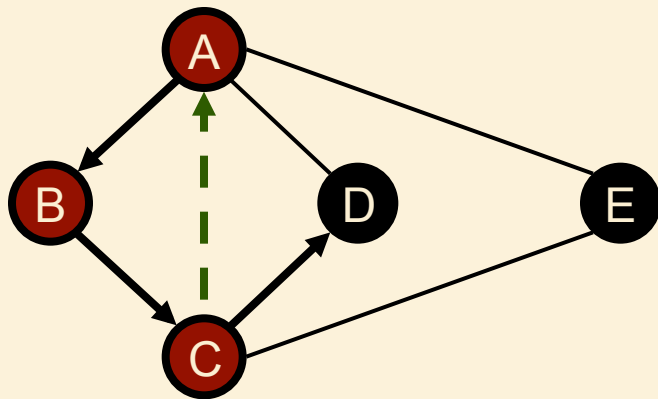
Input: Graph $G = (V, E)$ (directed or undirected)

- 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 Example on Undirected Graph



Example (cont.)



End of Lecture

APRIL 9, 2015

DFS Algorithm Pattern

DFS(G)

Precondition: G is a graph

Postcondition: all vertices in G have been visited

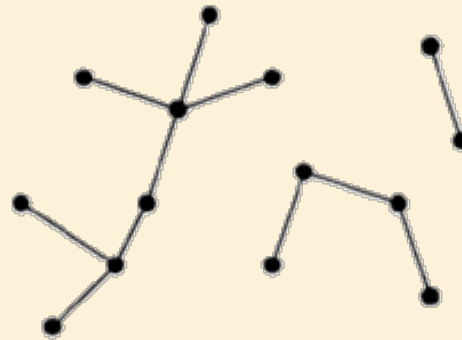
for each vertex $u \in V[G]$

$\text{color}[u] = \text{BLACK}$ //initialize vertex

for each vertex $u \in V[G]$

 if $\text{color}[u] = \text{BLACK}$ //as yet unexplored

 DFS-Visit(u)



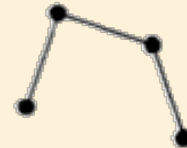
DFS Algorithm Pattern

DFS-Visit (u)

Precondition: vertex u is undiscovered

Postcondition: all vertices reachable from u have been processed

```
colour[u] ← RED
for each  $v \in \text{Adj}[u]$  //explore edge  $(u,v)$ 
    if color[v] = BLACK
        DFS-Visit( $v$ )
colour[u] ← GRAY
```



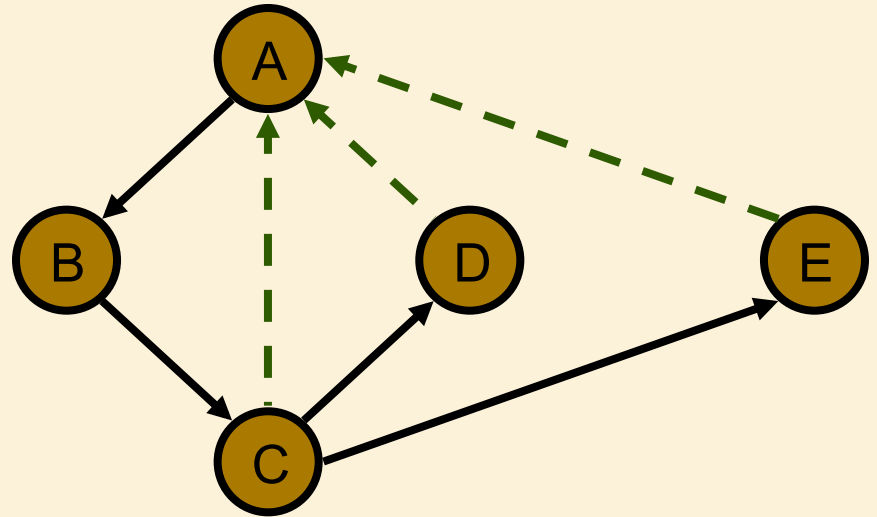
Properties of DFS

Property 1

DFS-Visit(u) visits all the vertices and edges in the connected component of u

Property 2

The discovery edges labeled by *DFS-Visit*(u) form a spanning tree of the connected component of u



DFS Algorithm Pattern

DFS(G)

Precondition: G is a graph

Postcondition: all vertices in G have been visited

for each vertex $u \in V[G]$

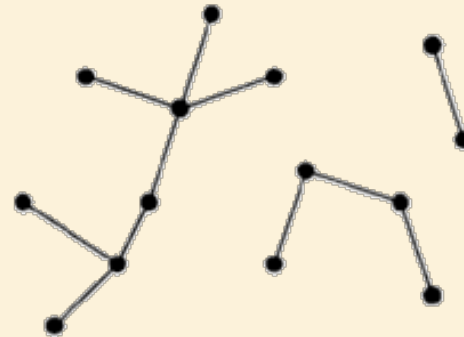
 color[u] = BLACK //initialize vertex

for each vertex $u \in V[G]$

 if color[u] = BLACK //as yet unexplored

 DFS-Visit(u)

} total work
= $\theta(V)$



DFS Algorithm Pattern

DFS-Visit (u)

Precondition: vertex u is undiscovered

Postcondition: all vertices reachable from u have been processed

$\text{colour}[u] \leftarrow \text{RED}$

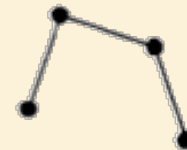
for each $v \in \text{Adj}[u]$ //explore edge (u,v)

if $\text{color}[v] = \text{BLACK}$

DFS-Visit(v)

$\text{colour}[u] \leftarrow \text{GRAY}$

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



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

(assuming adjacency list structure)

Variants of Depth-First Search

- In addition to, or instead of labeling vertices with colours, they can be labeled with **discovery** and **finishing** times.
- 'Time' is an integer that is incremented whenever a vertex changes state
 - ❑ from **unexplored** to **discovered**
 - ❑ from **discovered** to **finished**
- These **discovery** and **finishing** times can then be used to solve other graph problems (e.g., computing strongly-connected components)

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

Output: 2 timestamps on each vertex:

$d[v]$ = discovery time.

$f[v]$ = finishing time.

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

DFS Algorithm with Discovery and Finish Times

DFS(G)

Precondition: G is a graph

Postcondition: all vertices in G have been visited

for each vertex $u \in V[G]$

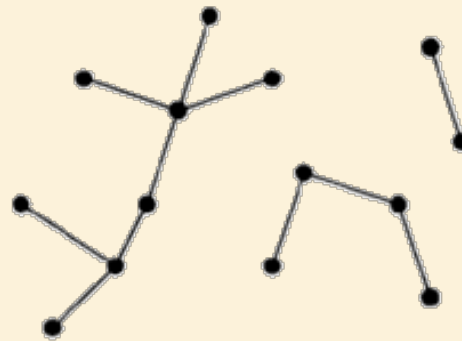
 color[u] = BLACK //initialize vertex

time $\leftarrow$ 0

for each vertex $u \in V[G]$

 if color[u] = BLACK //as yet unexplored

 DFS-Visit(u)



DFS Algorithm with Discovery and Finish Times

DFS-Visit (u)

Precondition: vertex u is undiscovered

Postcondition: all vertices reachable from u have been processed

$\text{colour}[u] \leftarrow \text{RED}$

$\text{time} \leftarrow \text{time} + 1$

$d[u] \leftarrow \text{time}$

for each $v \in \text{Adj}[u]$ //explore edge (u, v)

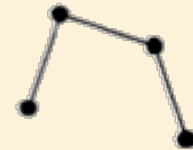
if $\text{color}[v] = \text{BLACK}$

DFS-Visit(v)

$\text{colour}[u] \leftarrow \text{GRAY}$

$\text{time} \leftarrow \text{time} + 1$

$f[u] \leftarrow \text{time}$



Other Variants of Depth-First Search

- The DFS Pattern can also be used to
 - ❑ Compute a forest of spanning trees (one for each call to DFS-visit) encoded in a predecessor list $\pi[u]$
 - ❑ Label edges in the graph according to their role in the search
 - ✧ **Tree edges**, traversed to an undiscovered vertex
 - ✧ **Forward edges**, traversed to a descendent vertex on the current spanning tree
 - ✧ **Back edges**, traversed to an ancestor vertex on the current spanning tree
 - ✧ **Cross edges**, traversed to a vertex that has already been discovered, but is not an ancestor or a descendent

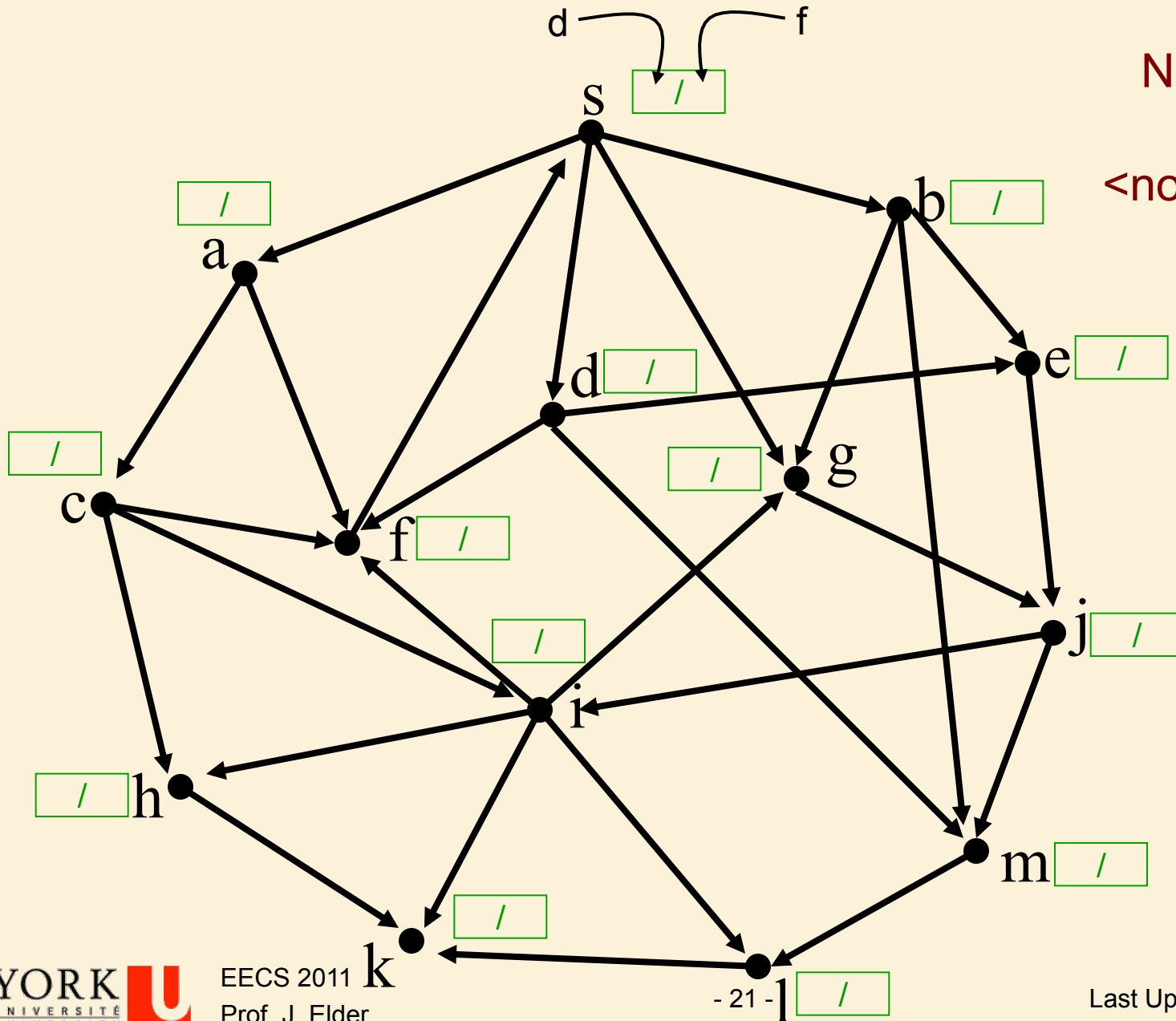
Outline

- DFS Algorithm
- **DFS Example**
- DFS Applications

DFS

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

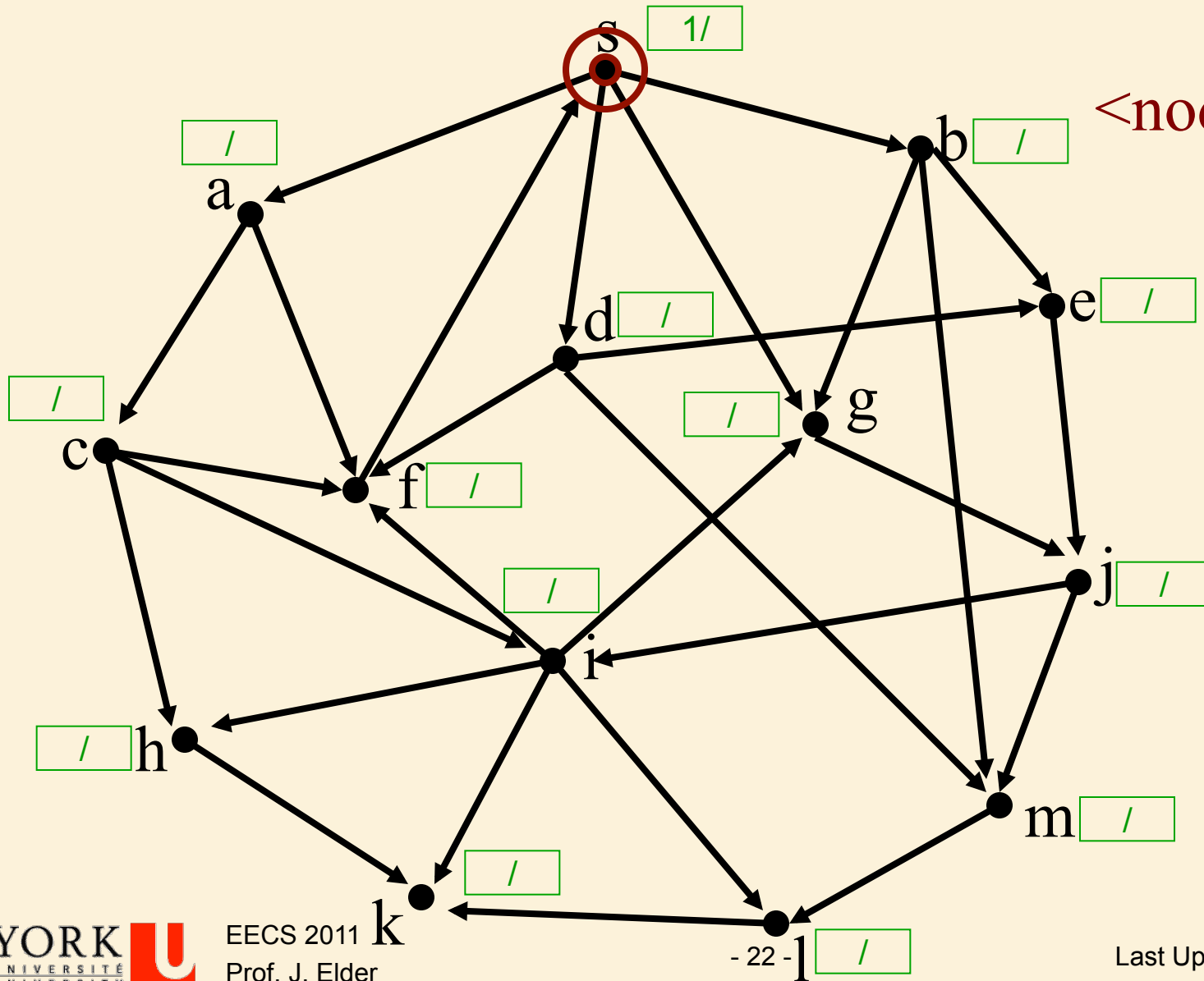
Found
Not Handled
Stack
<node,# edges>



DFS

Found
Not Handled
Stack

<node,# edges>

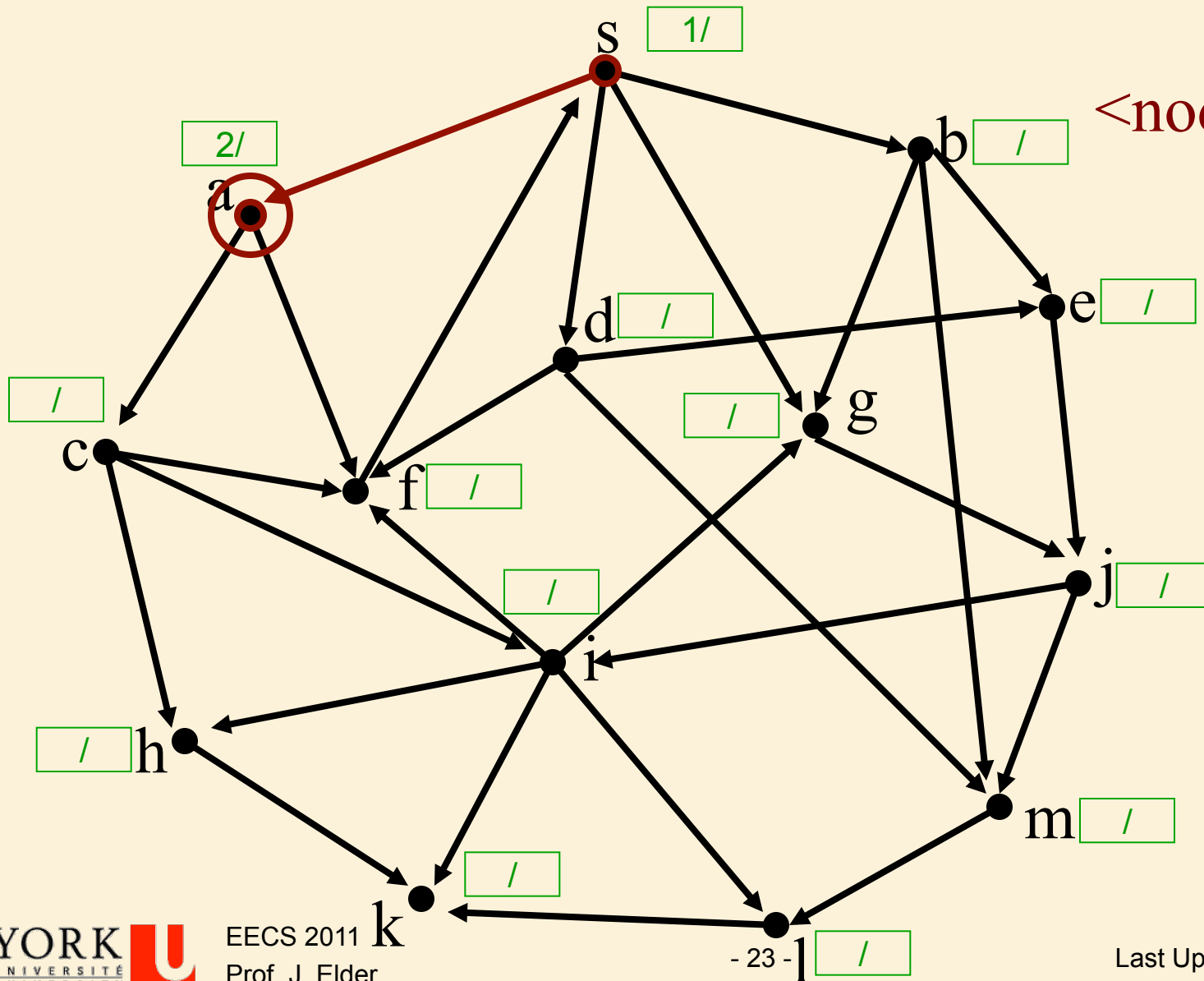


s,0

DFS

Found
Not Handled
Stack

<node,# edges>



a,0
s,1

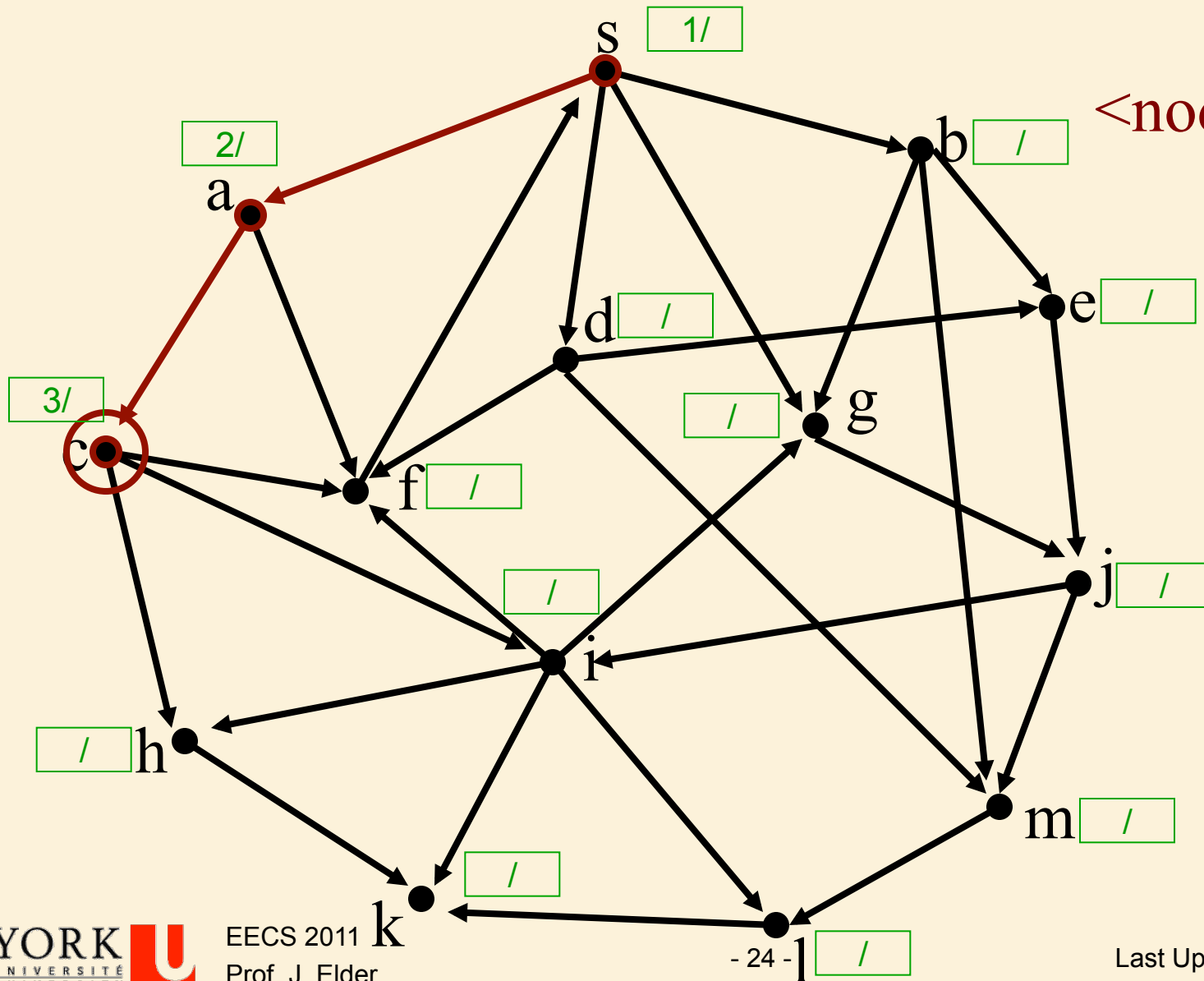
- 23 - 1

Last Updated April 9, 2015

DFS

Found
Not Handled
Stack

<node,# edges>

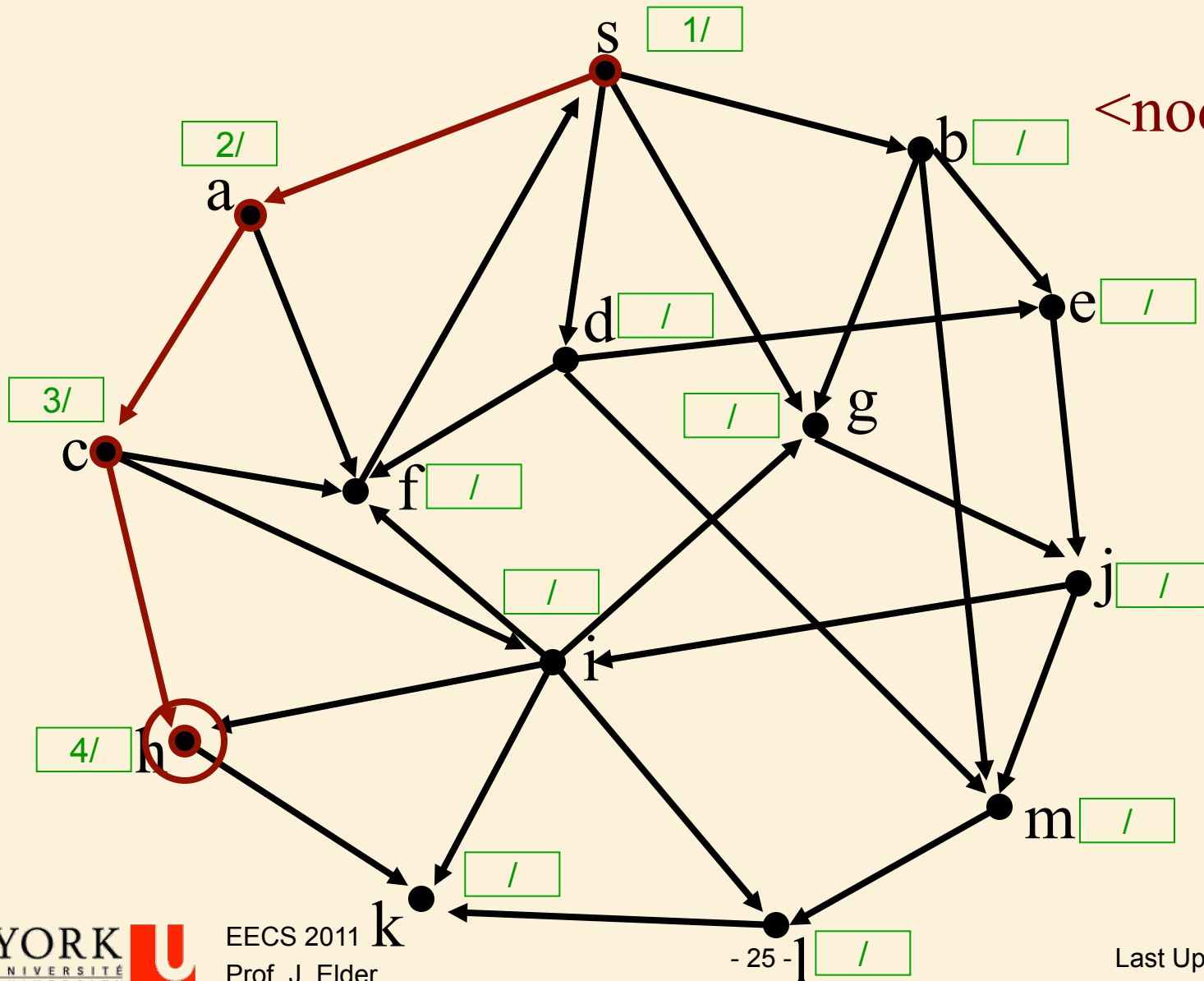


c,0
a,1
s,1

DFS

Found
Not Handled
Stack

<node,# edges>

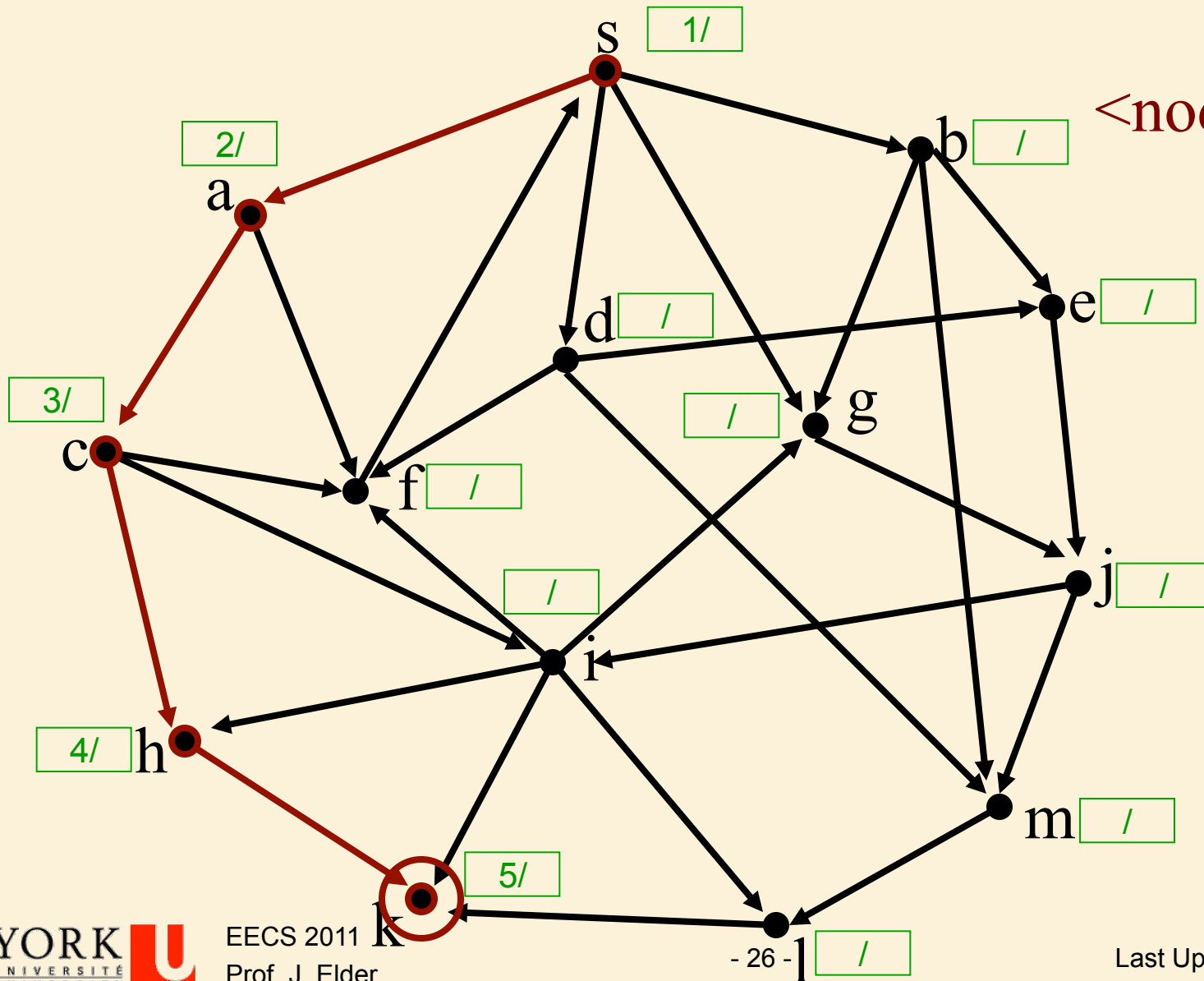


h,0
c,1
a,1
s,1

DFS

Found
Not Handled
Stack

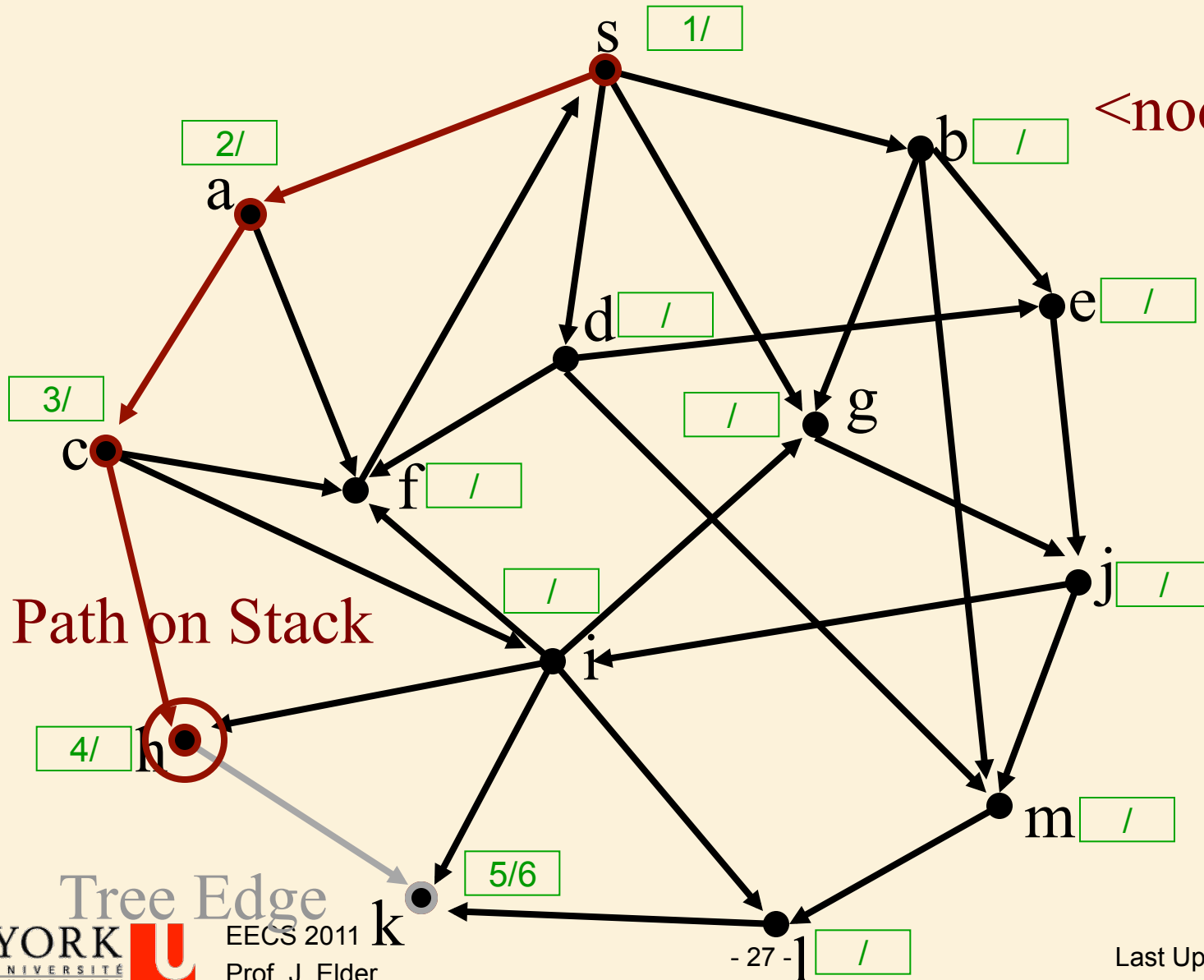
<node,# edges>



DFS

Found
Not Handled
Stack

<node,# edges>



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

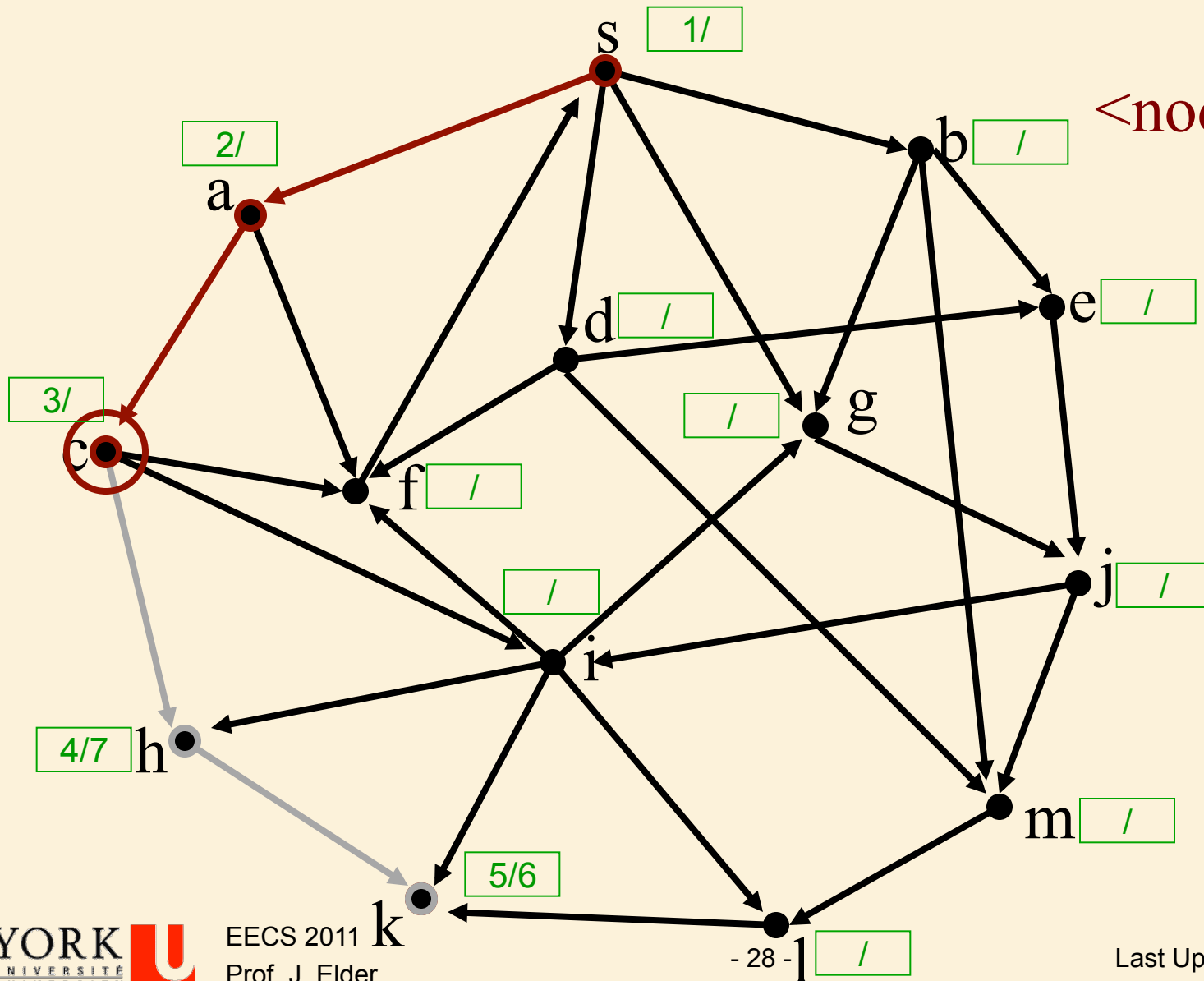
- 27 - 1

Last Updated April 9, 2015

DFS

Found
Not Handled
Stack

<node,# edges>

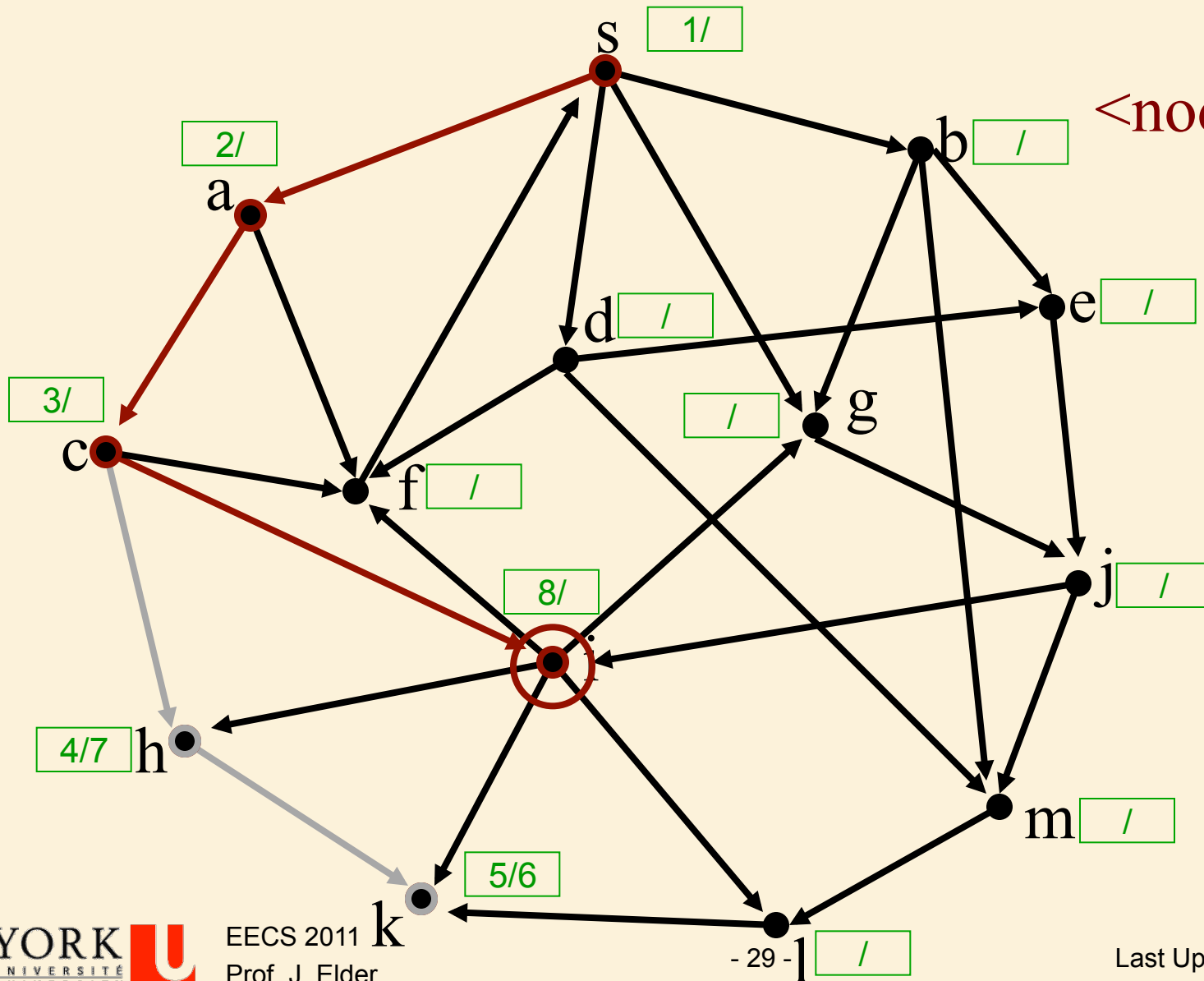


c,1
a,1
s,1

DFS

Found
Not Handled
Stack

<node,# edges>

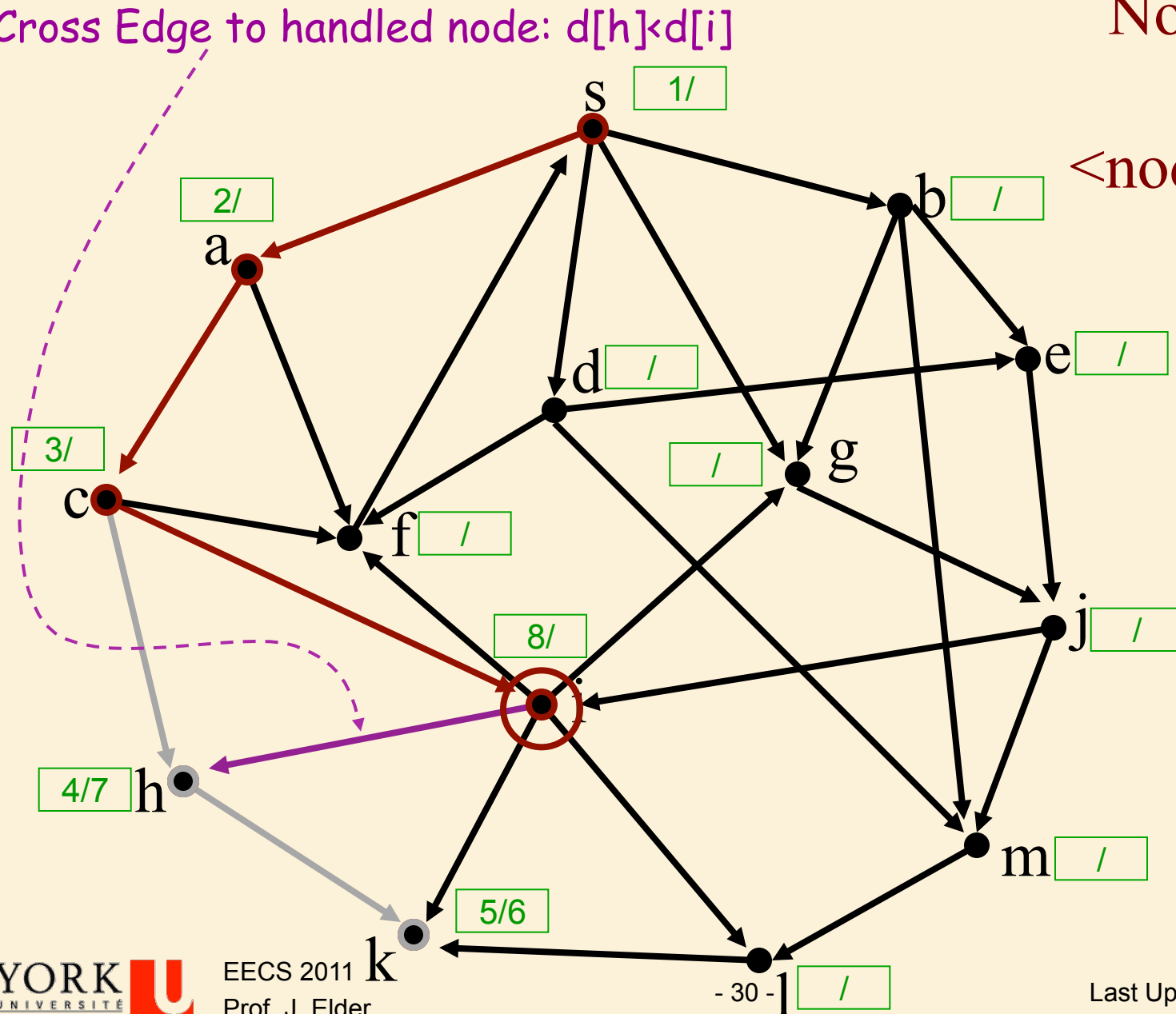


i,0
c,2
a,1
s,1

DFS

Found Not Handled Stack

<node,# edges>

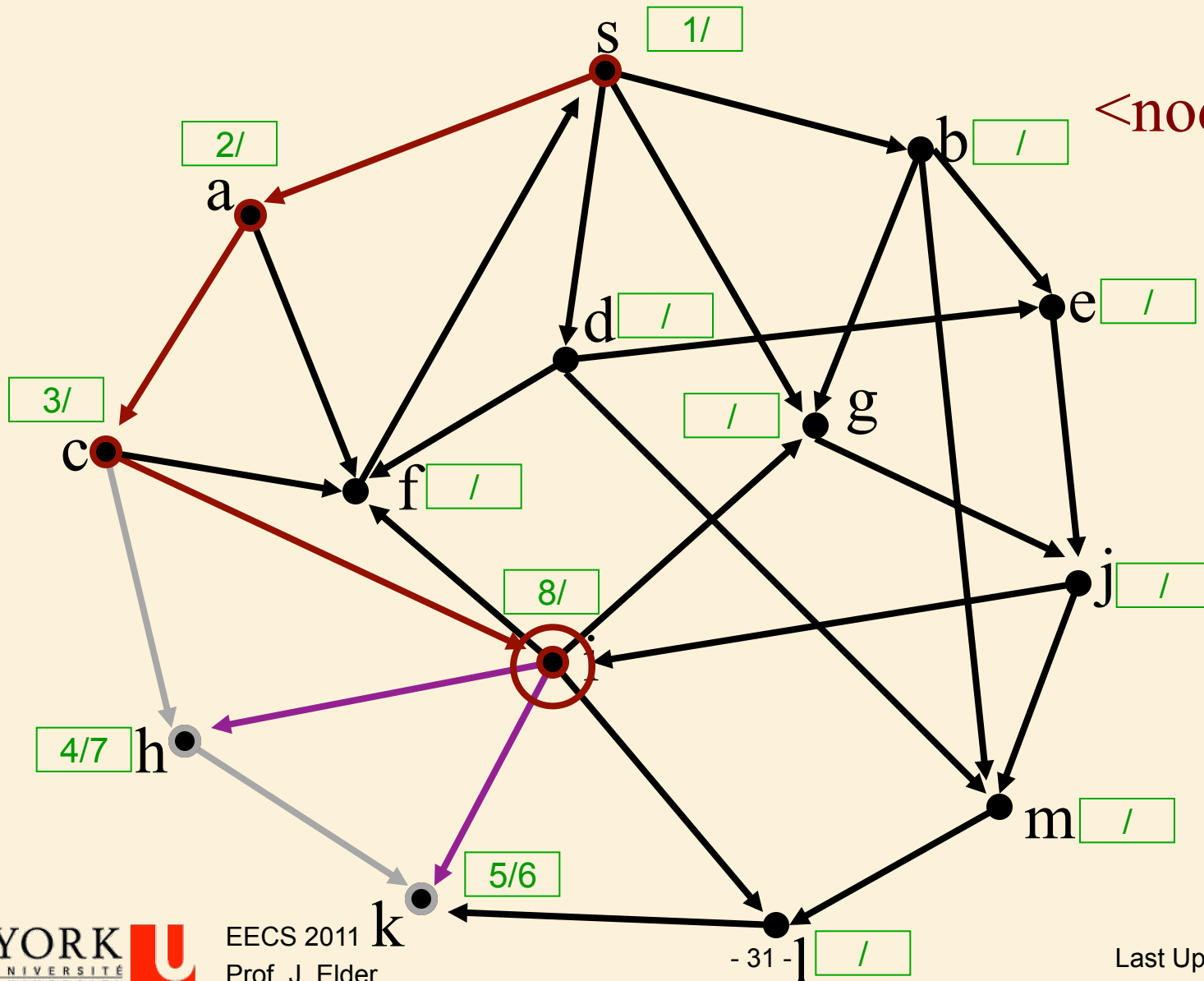


i,1
c,2
a,1
s,1

DFS

Found
Not Handled
Stack

<node,# edges>

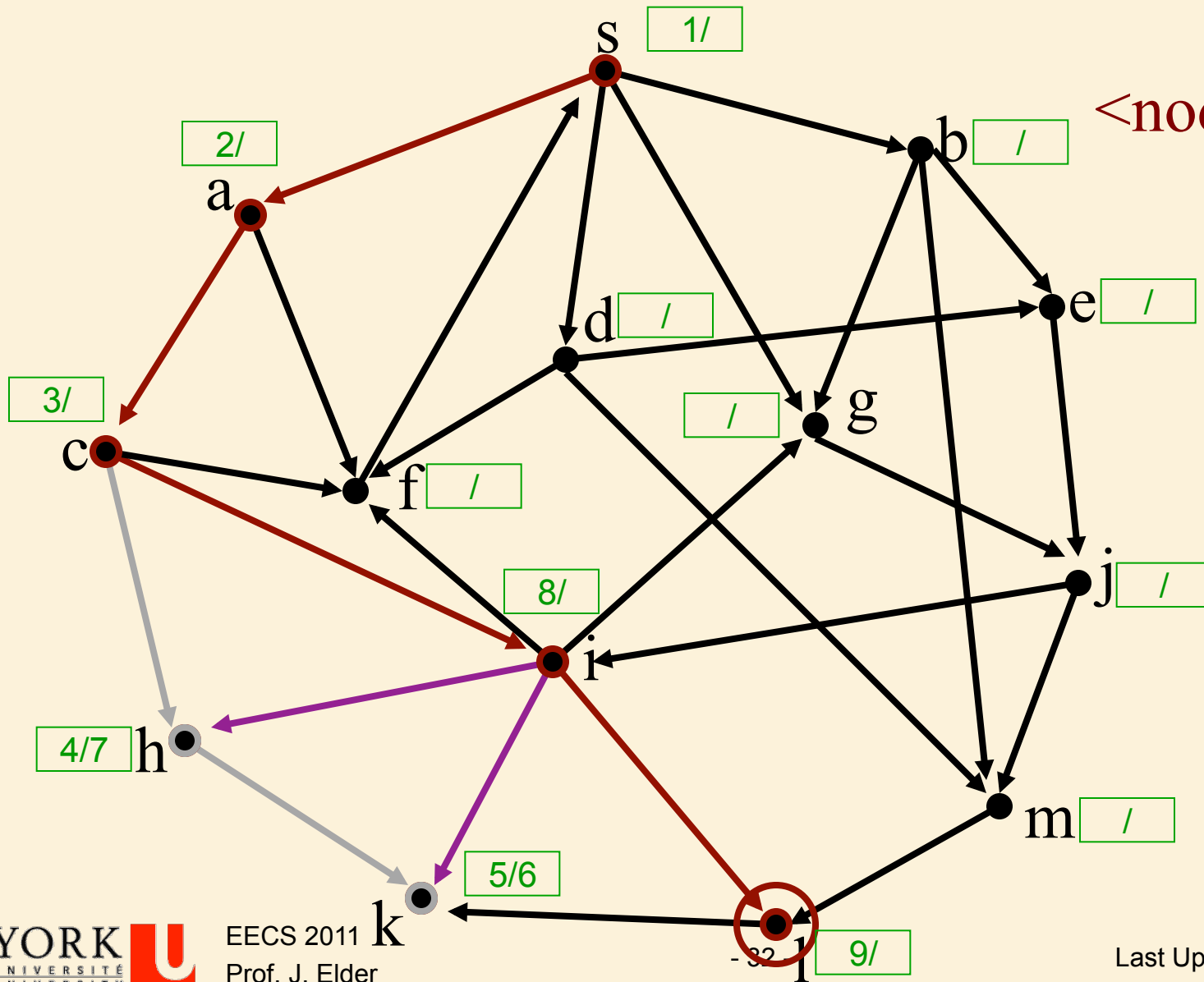


i,2
c,2
a,1
s,1

DFS

Found
Not Handled
Stack

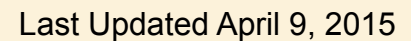
<node,# edges>



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

Found Not Handled Stack

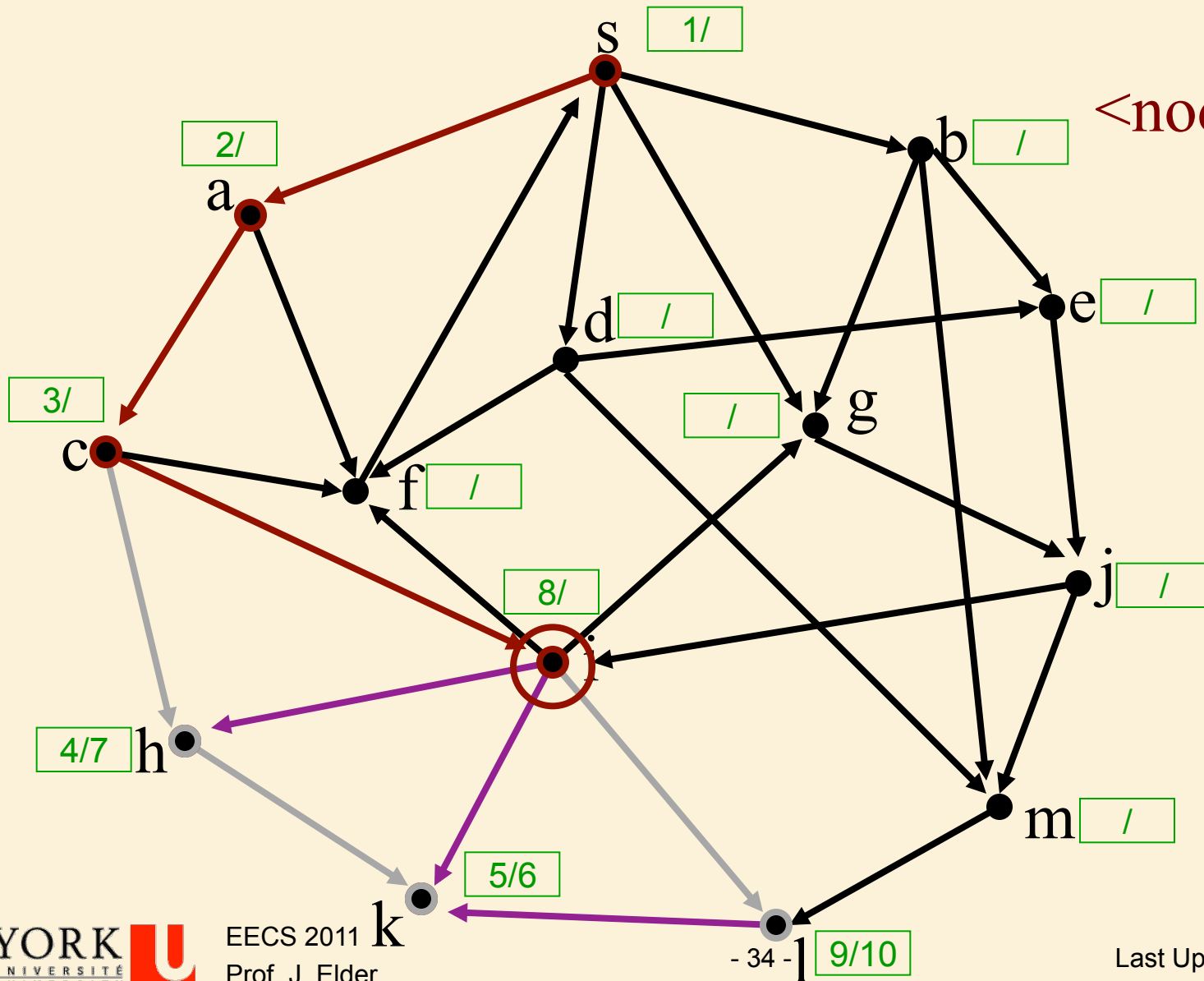
1,1
i,3
c,2
a,1
s,1



DFS

Found
Not Handled
Stack

<node,# edges>



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

Found Not Handled Stack

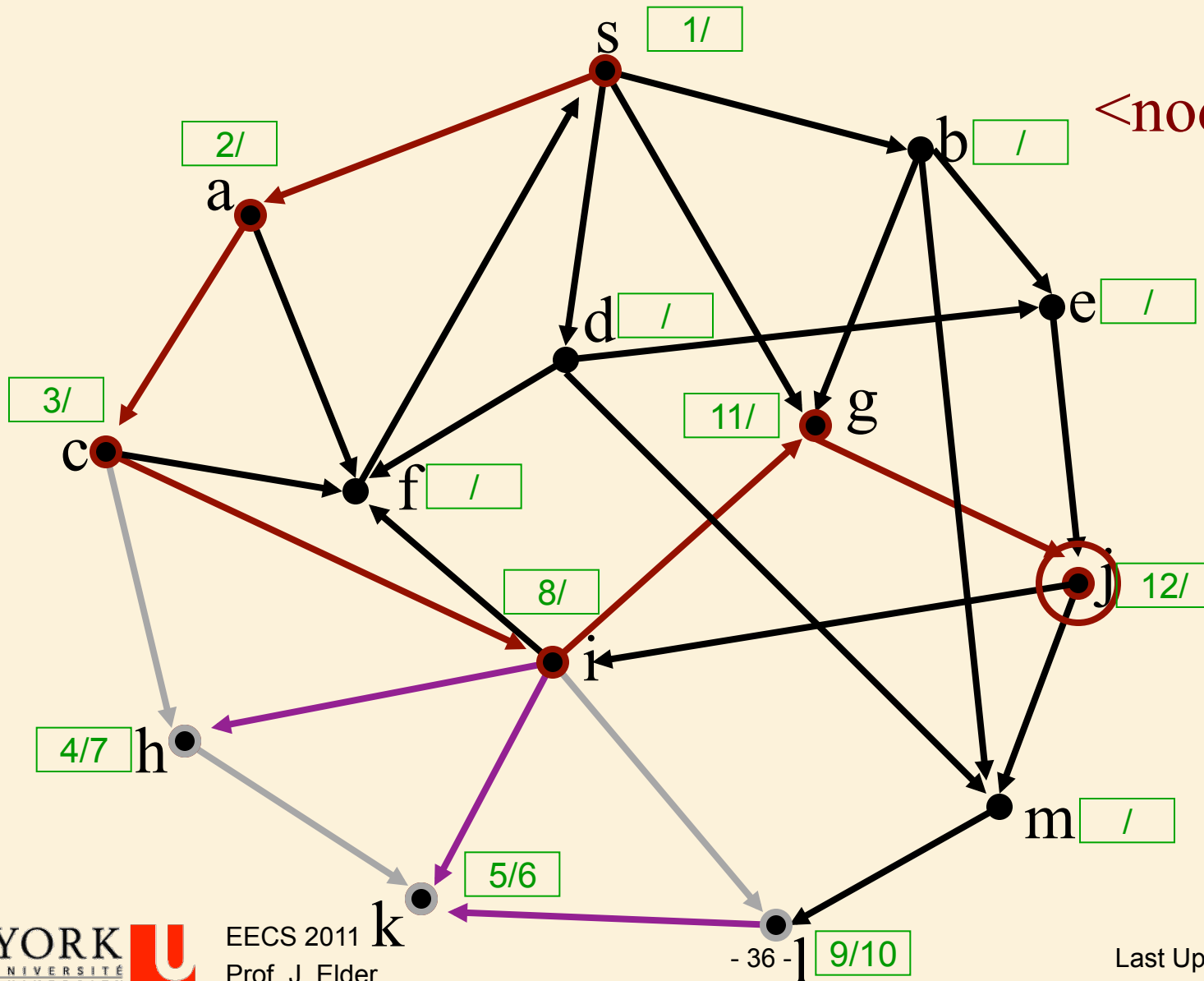
g,0
i,4
c,2
a,1
s,1



DFS

Found
Not Handled
Stack

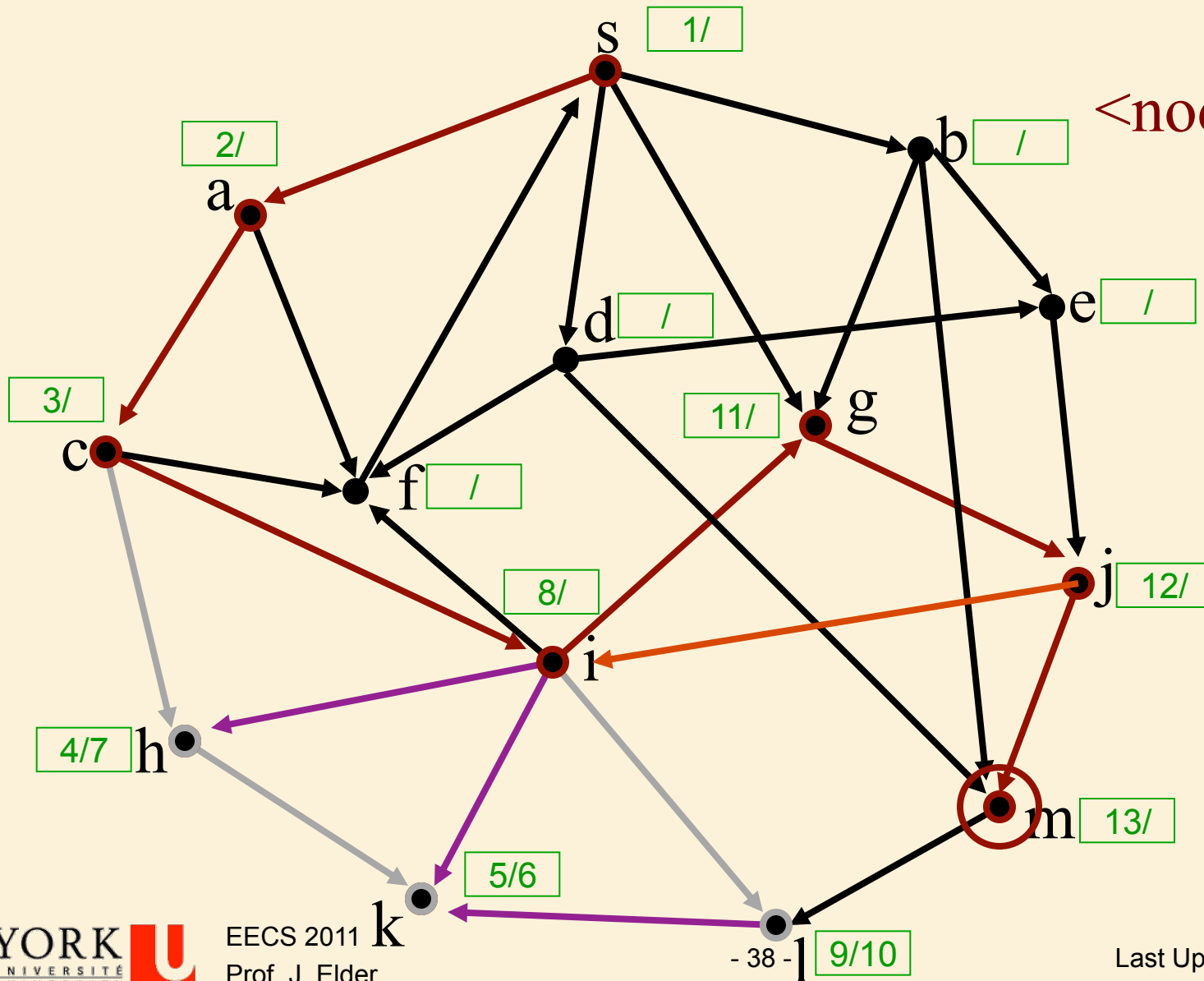
<node,# edges>



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

Found Not Handled Stack

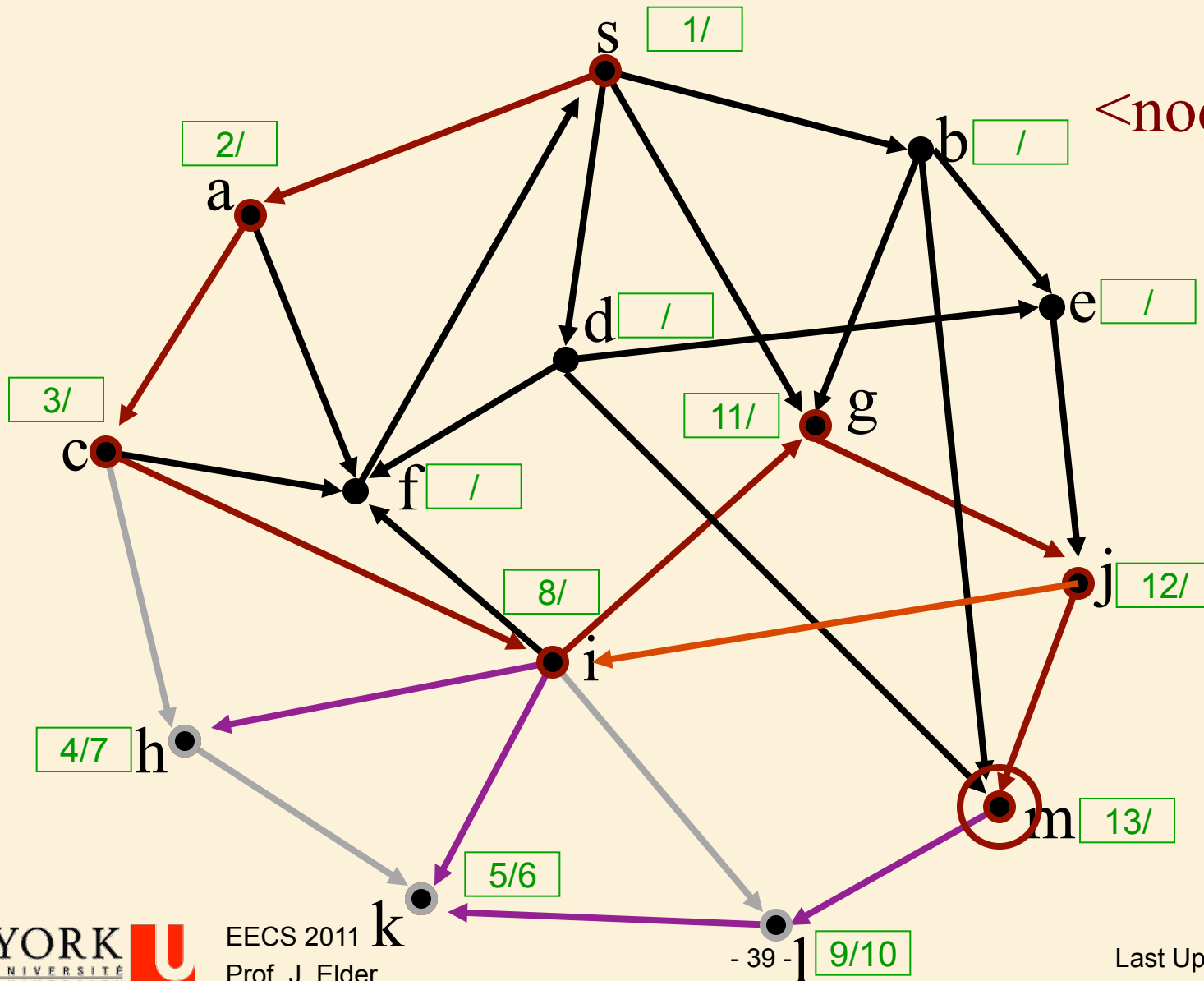
m,0
j,2
g,1
i,4
c,2
a,1
s,1



DFS

Found
Not Handled
Stack

<node,# edges>

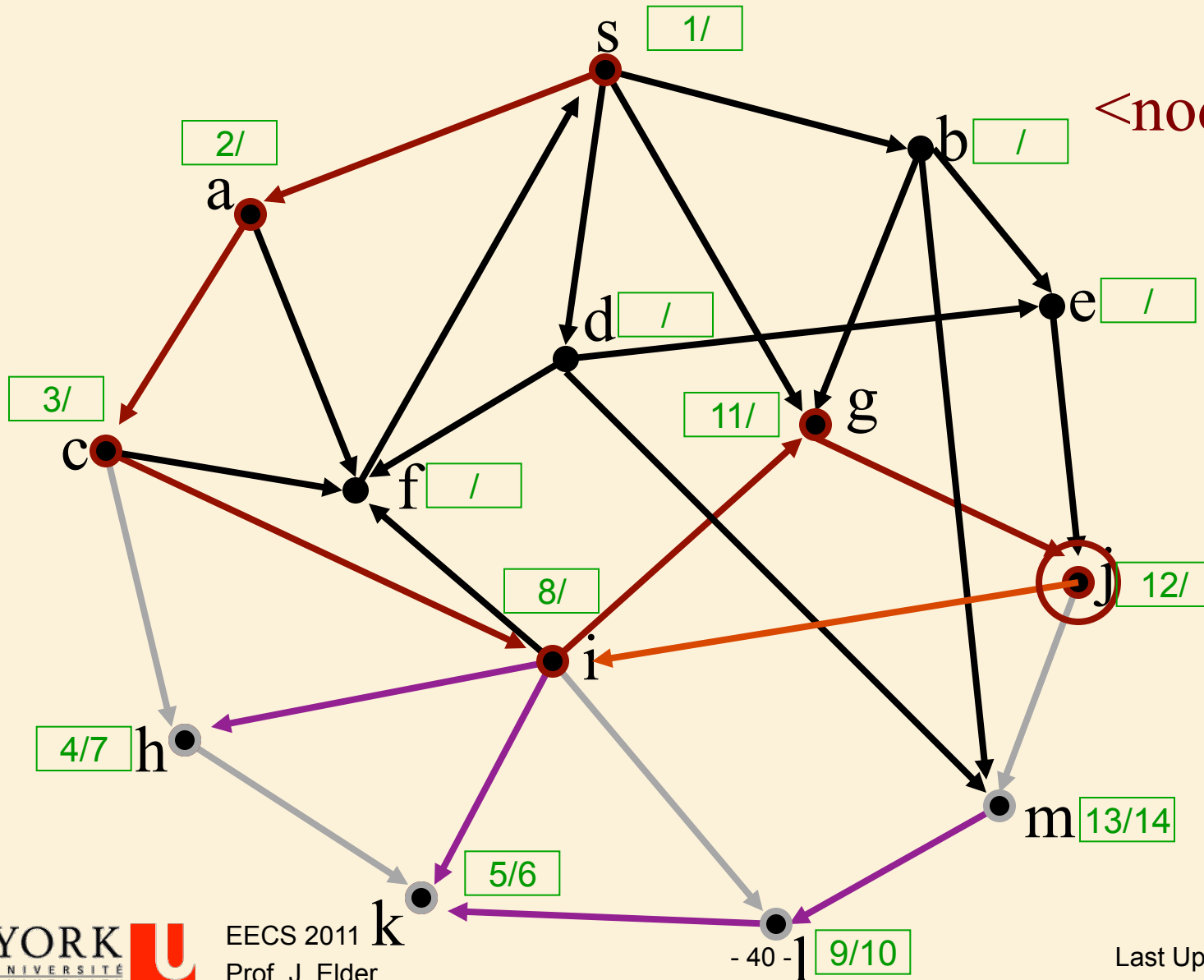


m,1
j,2
g,1
i,4
c,2
a,1
s,1

DFS

Found
Not Handled
Stack

<node,# edges>



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

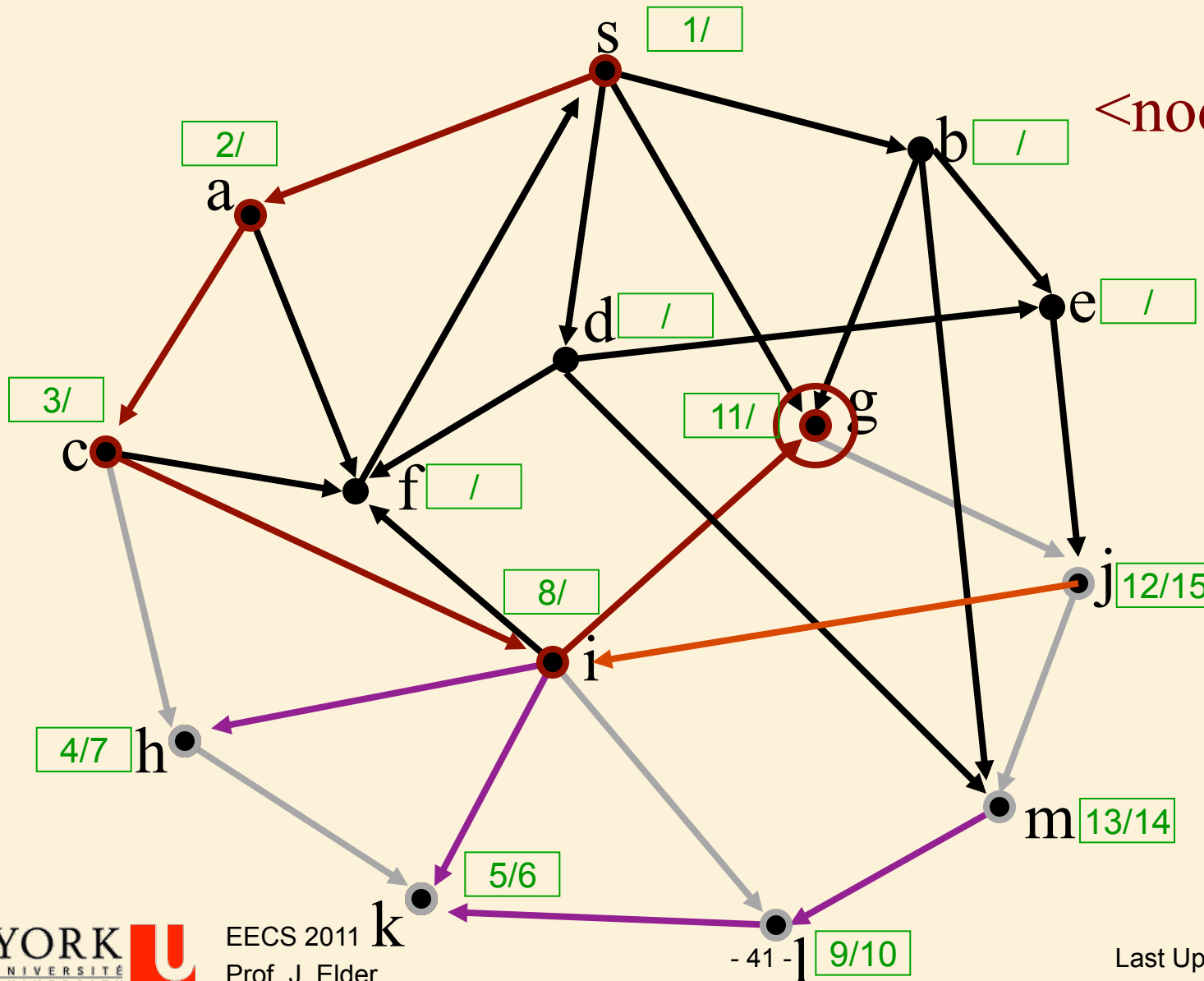
- 40 - 1 9/10

Last Updated April 9, 2015

DFS

Found
Not Handled
Stack

<node,# edges>

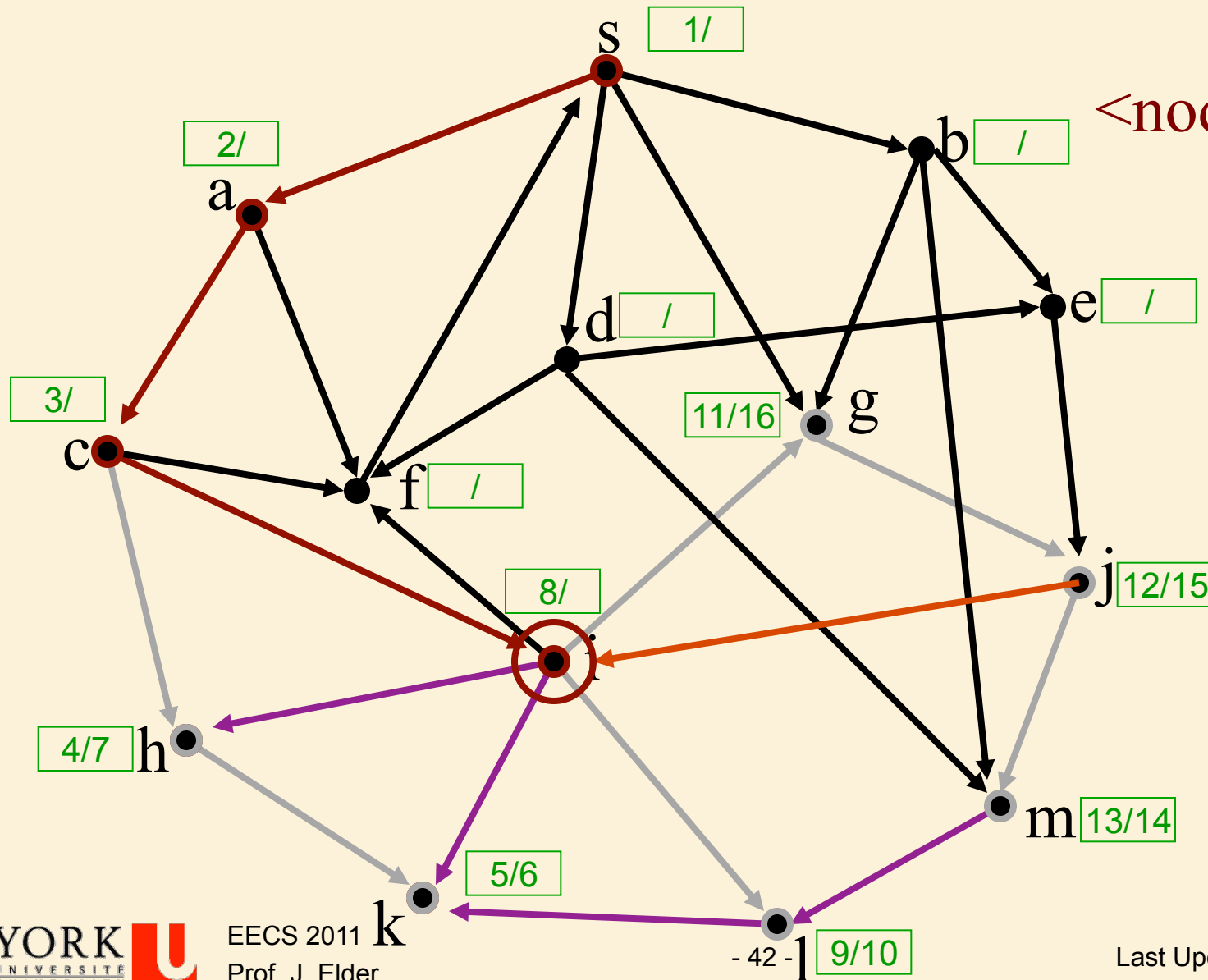


g,1
i,4
c,2
a,1
s,1

DFS

Found
Not Handled
Stack

<node,# edges>

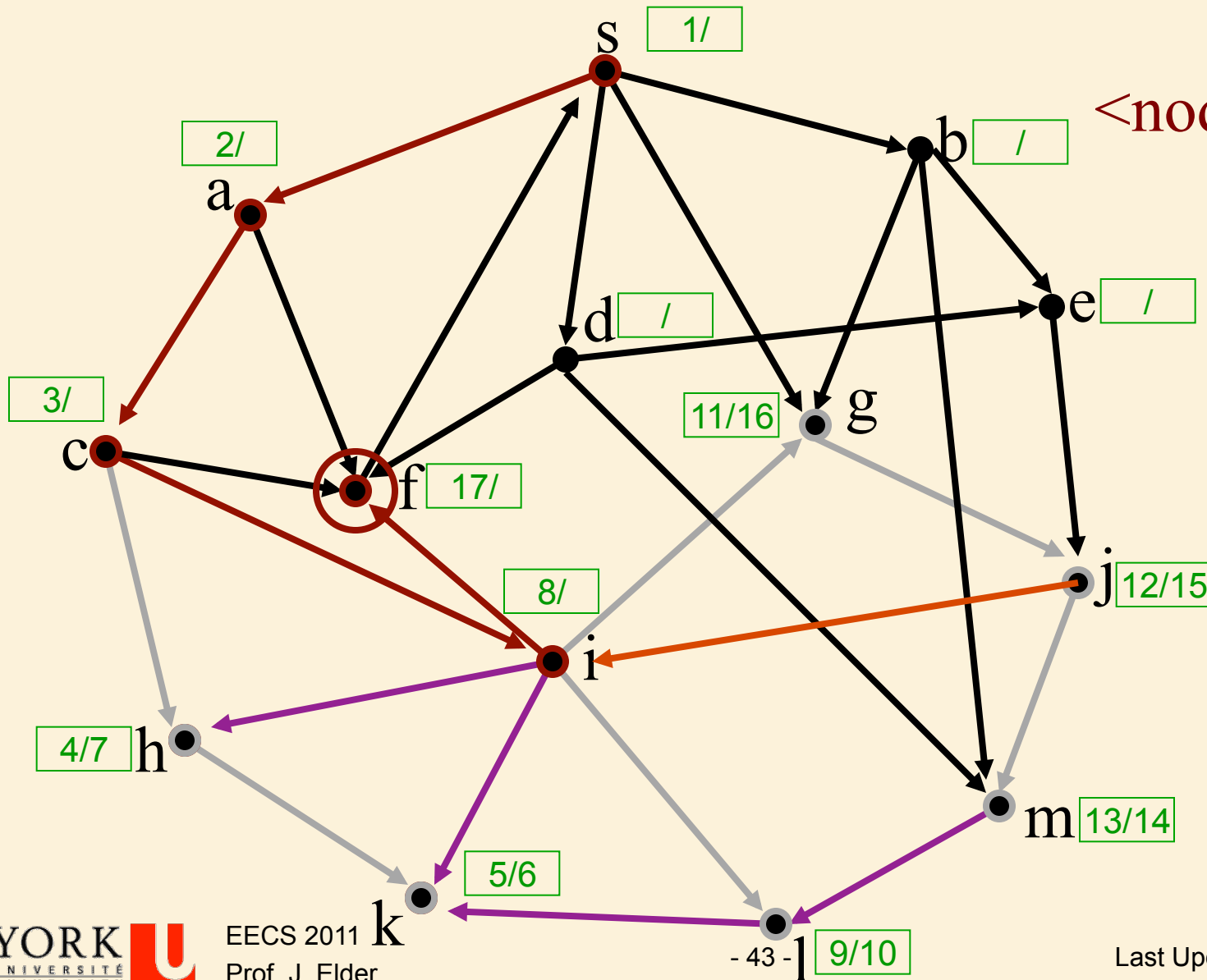


i,4
c,2
a,1
s,1

DFS

Found
Not Handled
Stack

<node,# edges>

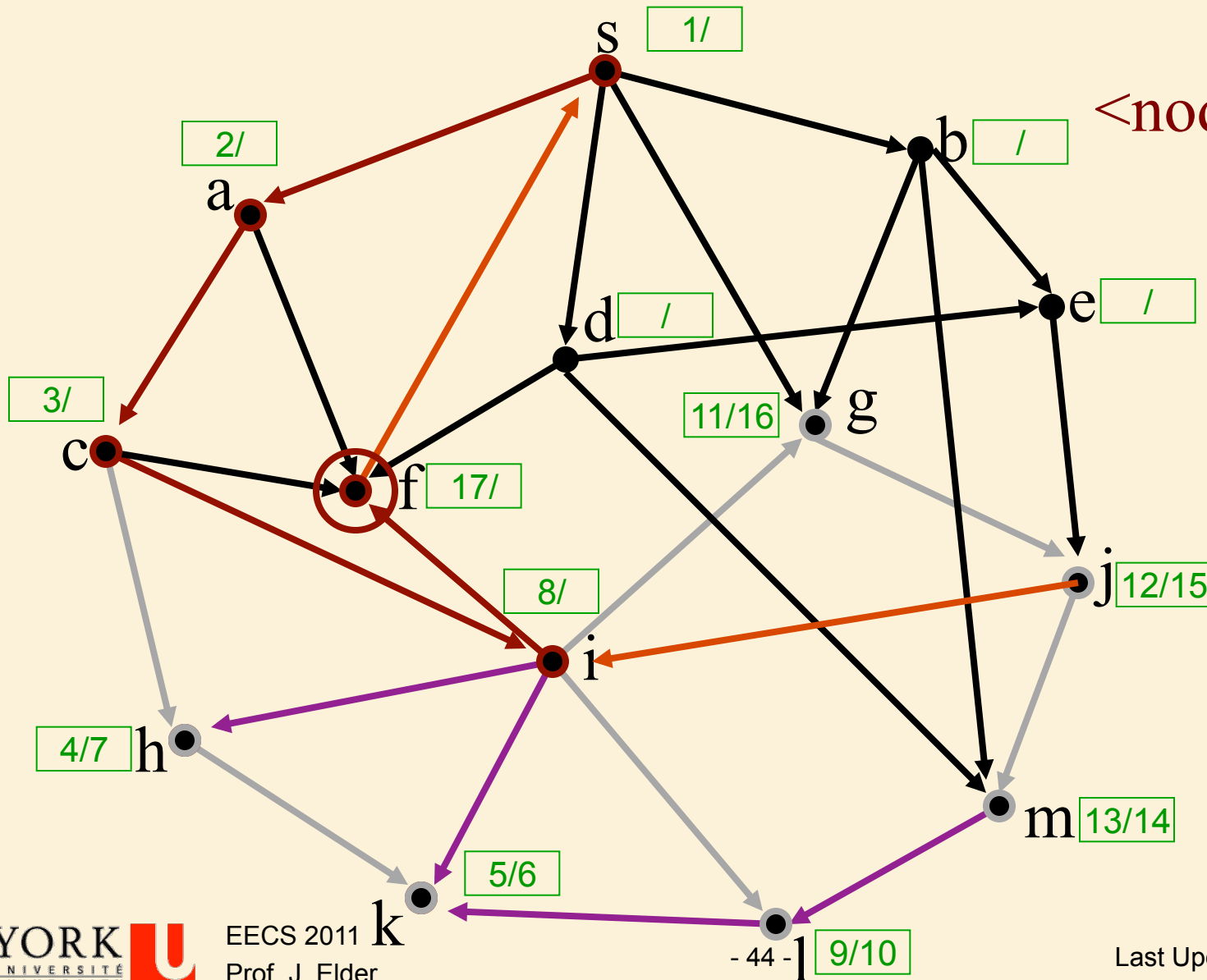


f,0
i,5
c,2
a,1
s,1

DFS

Found
Not Handled
Stack

<node,# edges>

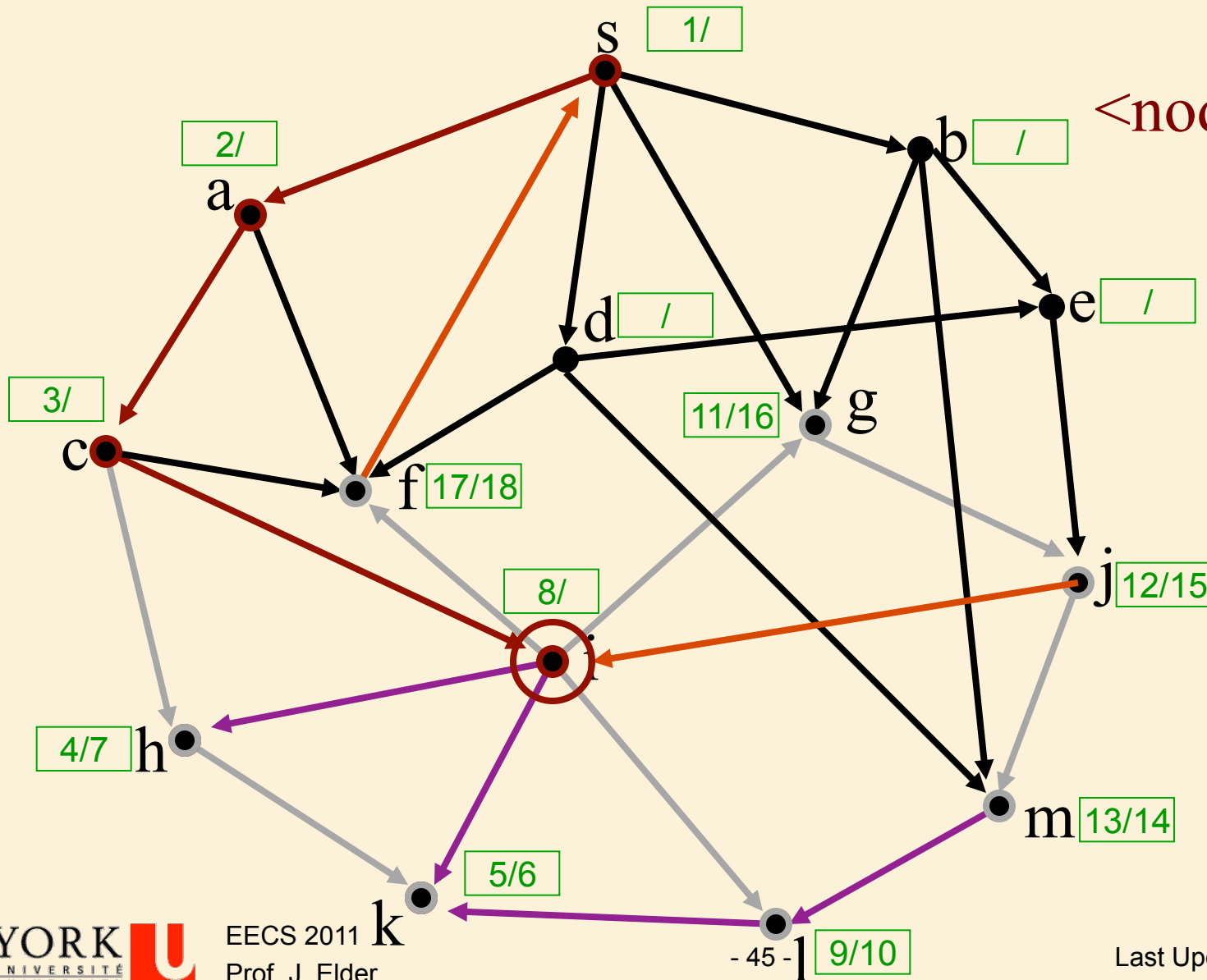


Last Updated April 9, 2015

DFS

Found
Not Handled
Stack

<node,# edges>

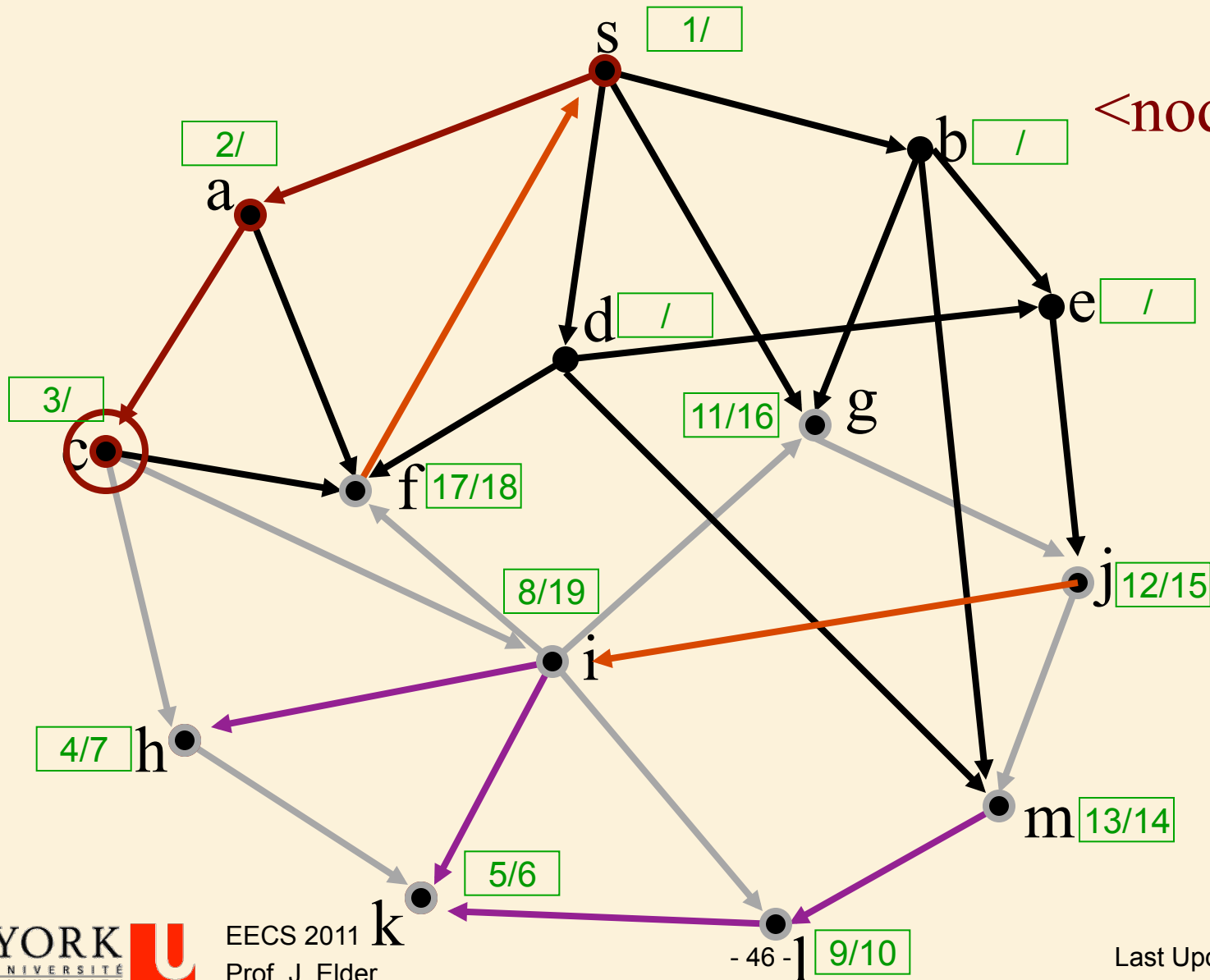


Last Updated April 9, 2015

DFS

Found
Not Handled
Stack

<node,# edges>

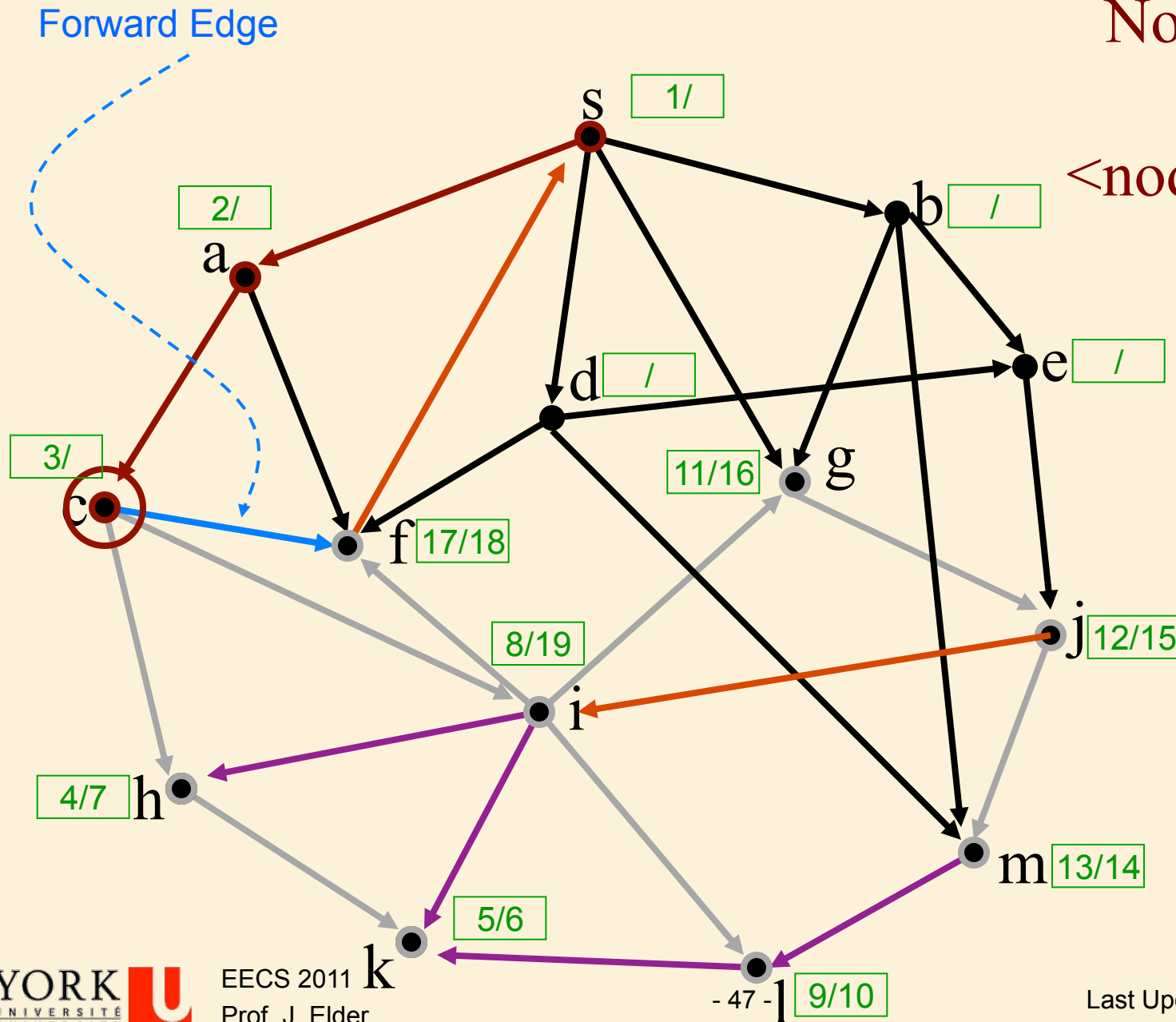


Last Updated April 9, 2015

DFS

Found
Not Handled
Stack

<node,# edges>

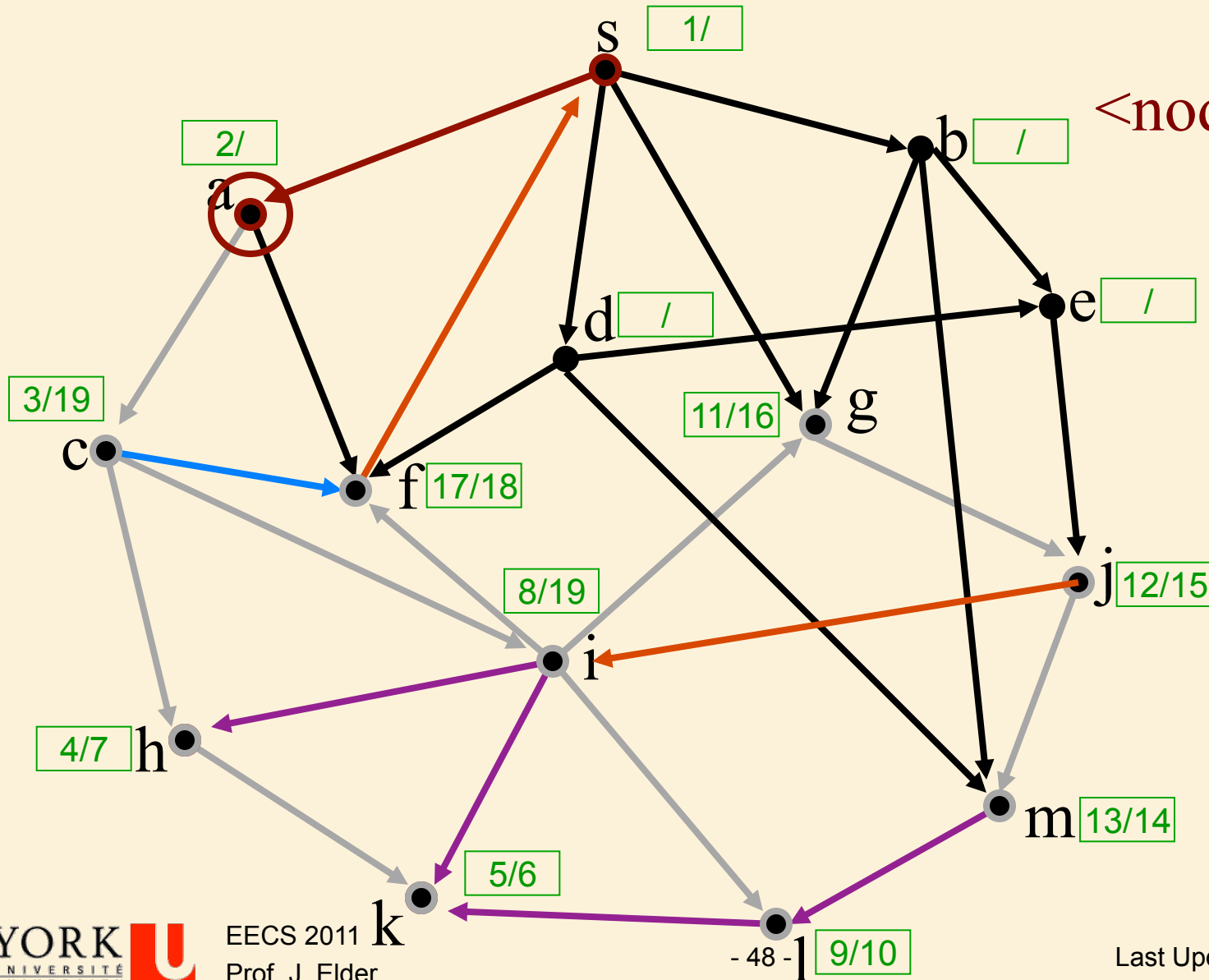


Last Updated April 9, 2015

DFS

Found
Not Handled
Stack

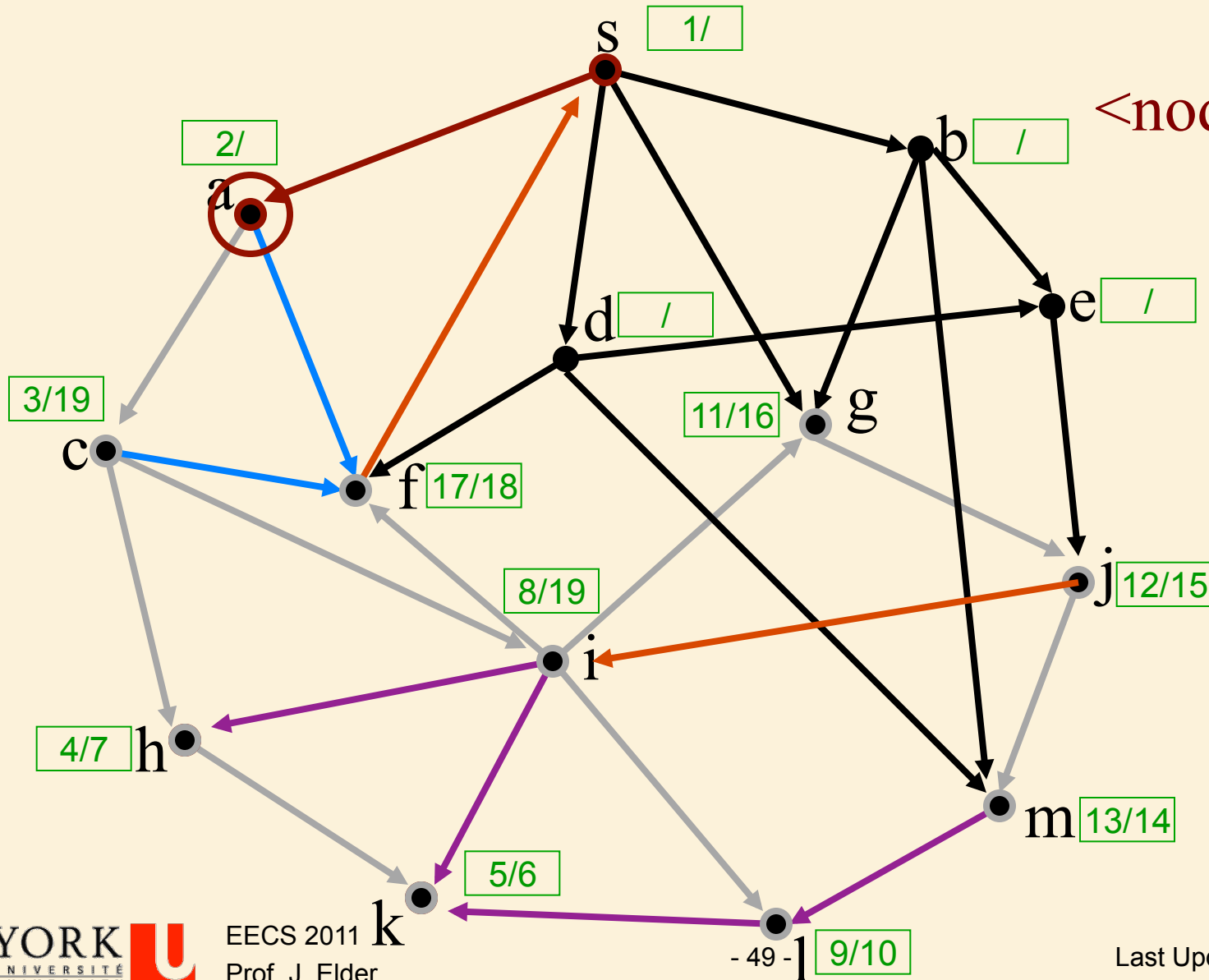
<node,# edges>



DFS

Found
Not Handled
Stack

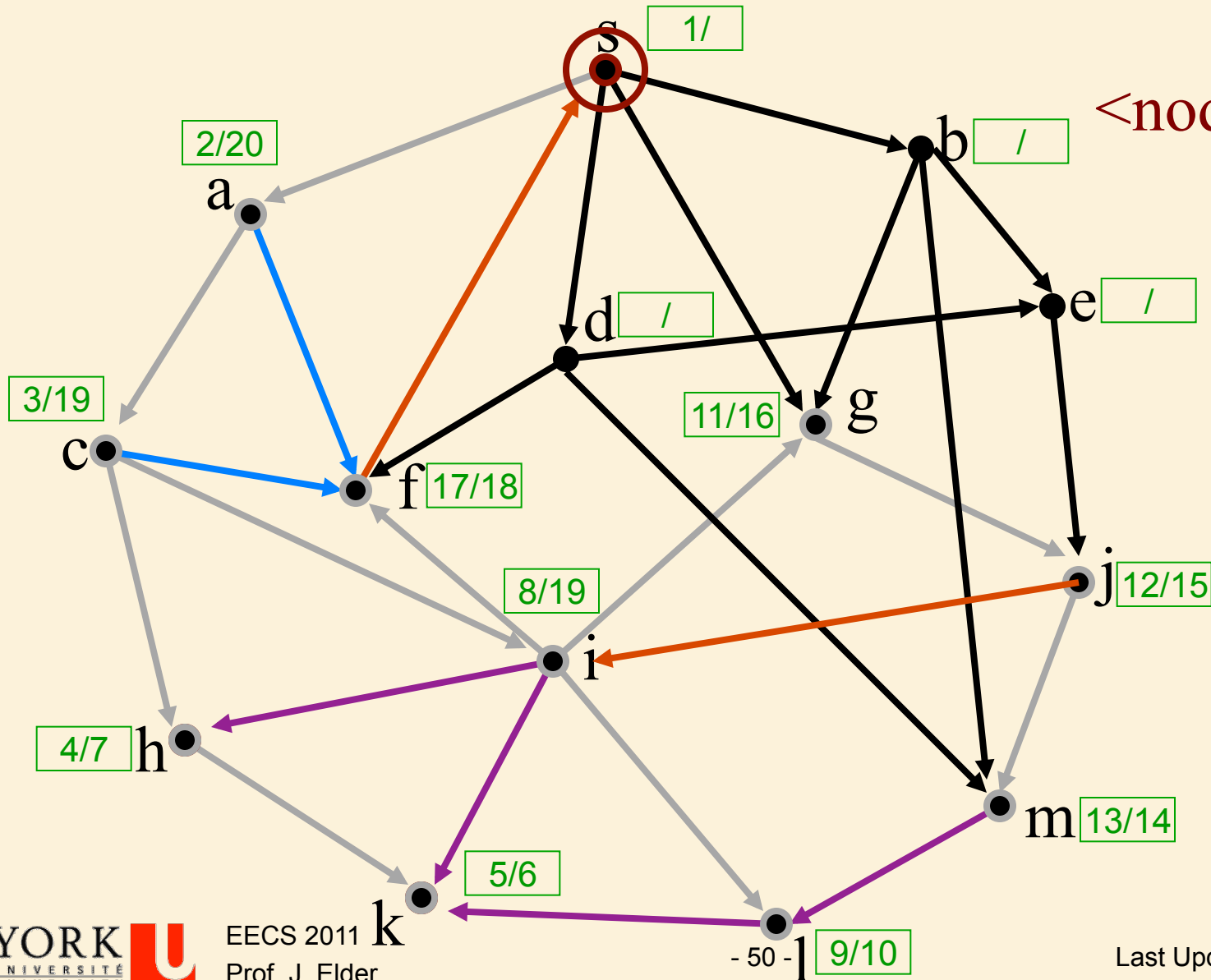
<node,# edges>



DFS

Found
Not Handled
Stack

<node,# edges>

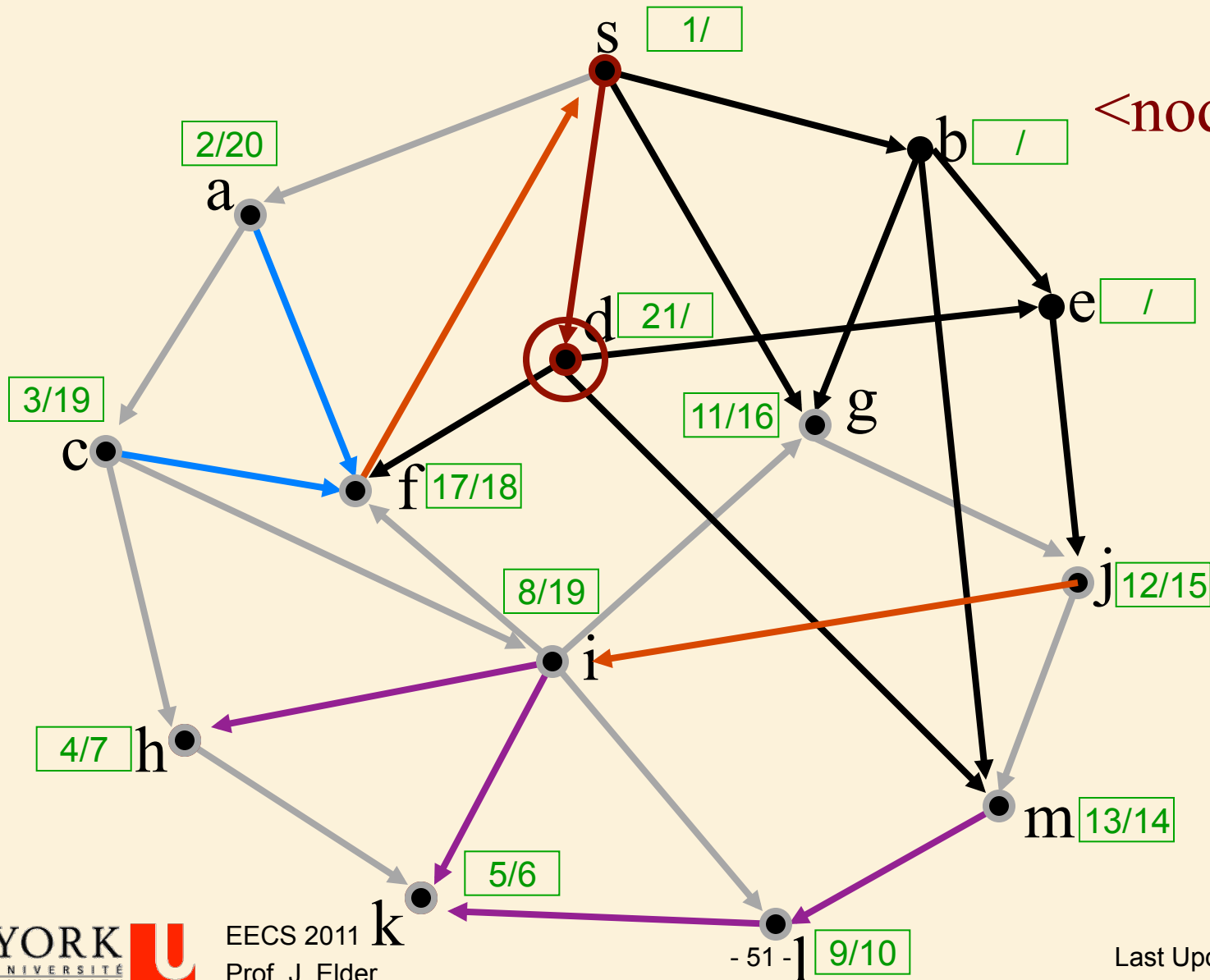


s,1

DFS

Found
Not Handled
Stack

<node,# edges>

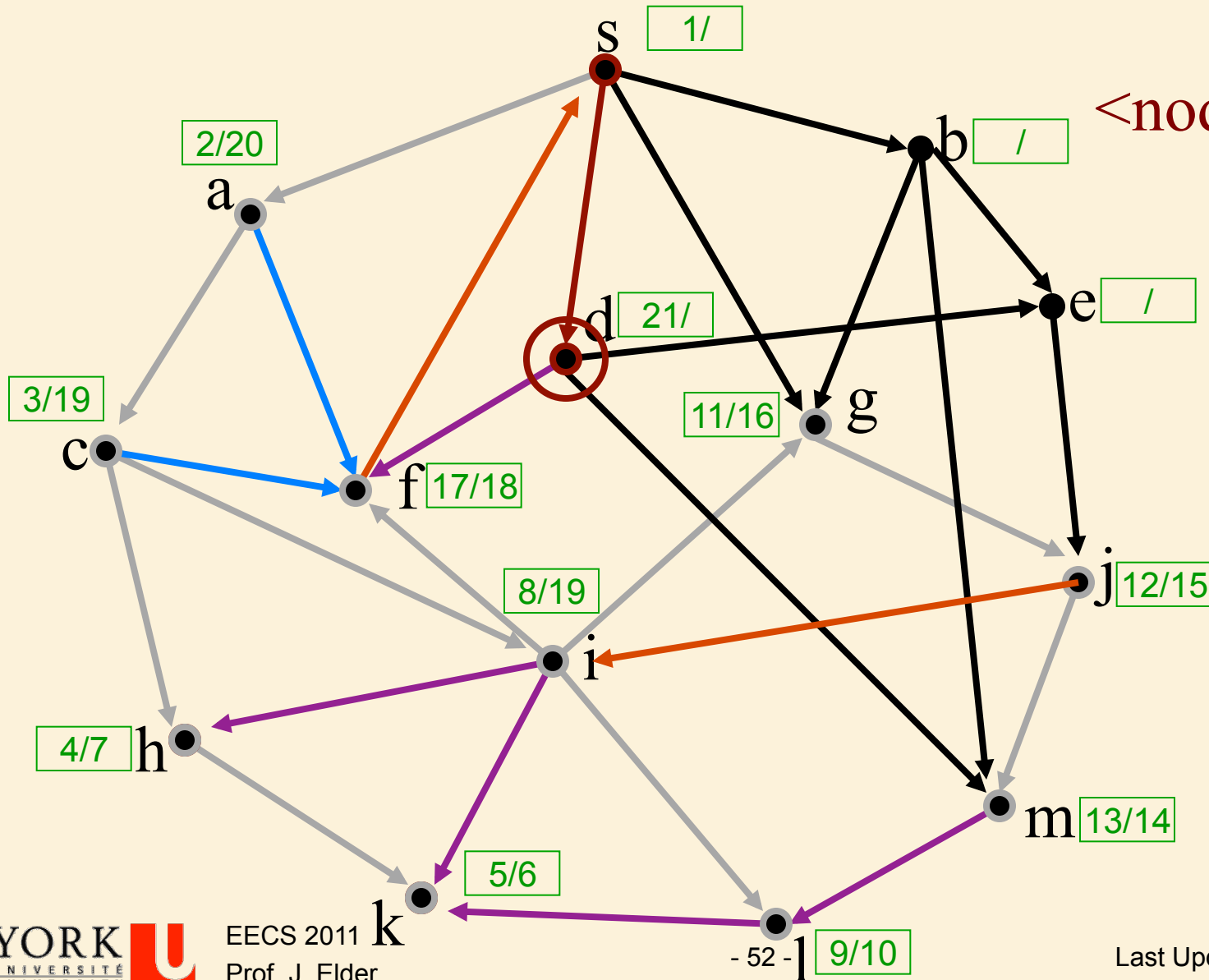


d,0
s,2

DFS

Found
Not Handled
Stack

<node,# edges>

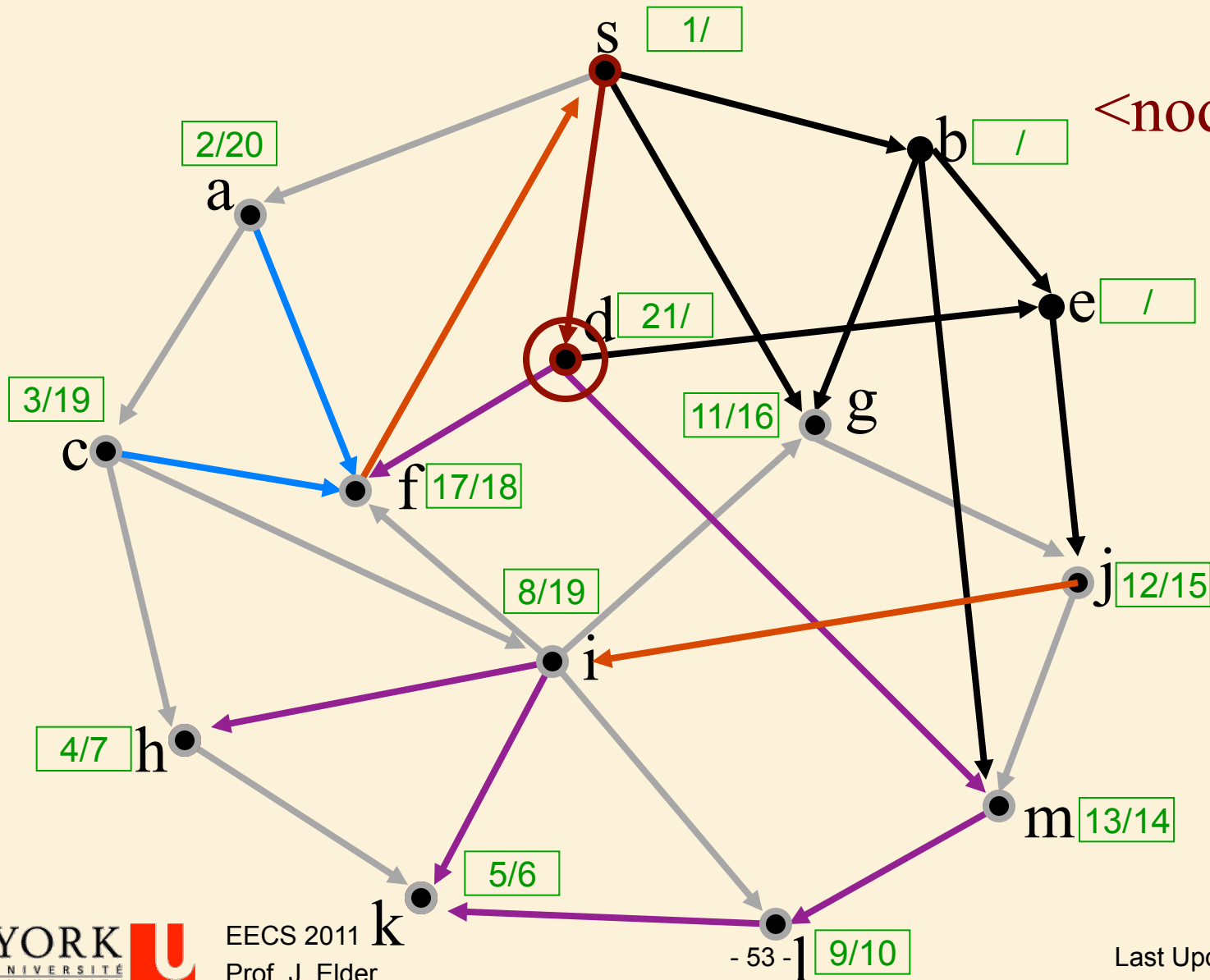


d,1
s,2

DFS

Found
Not Handled
Stack

<node,# edges>

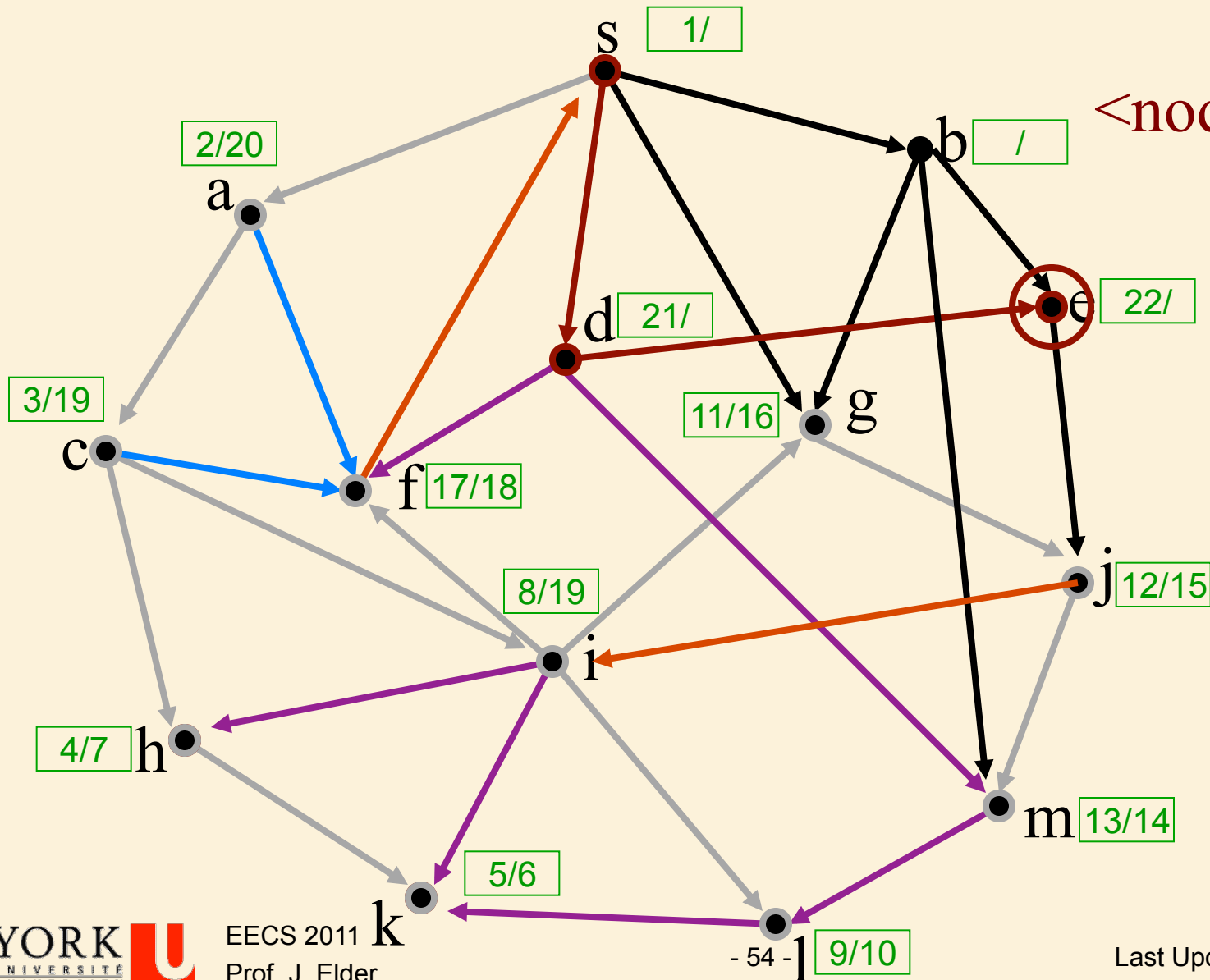


Last Updated April 9, 2015

DFS

Found
Not Handled
Stack

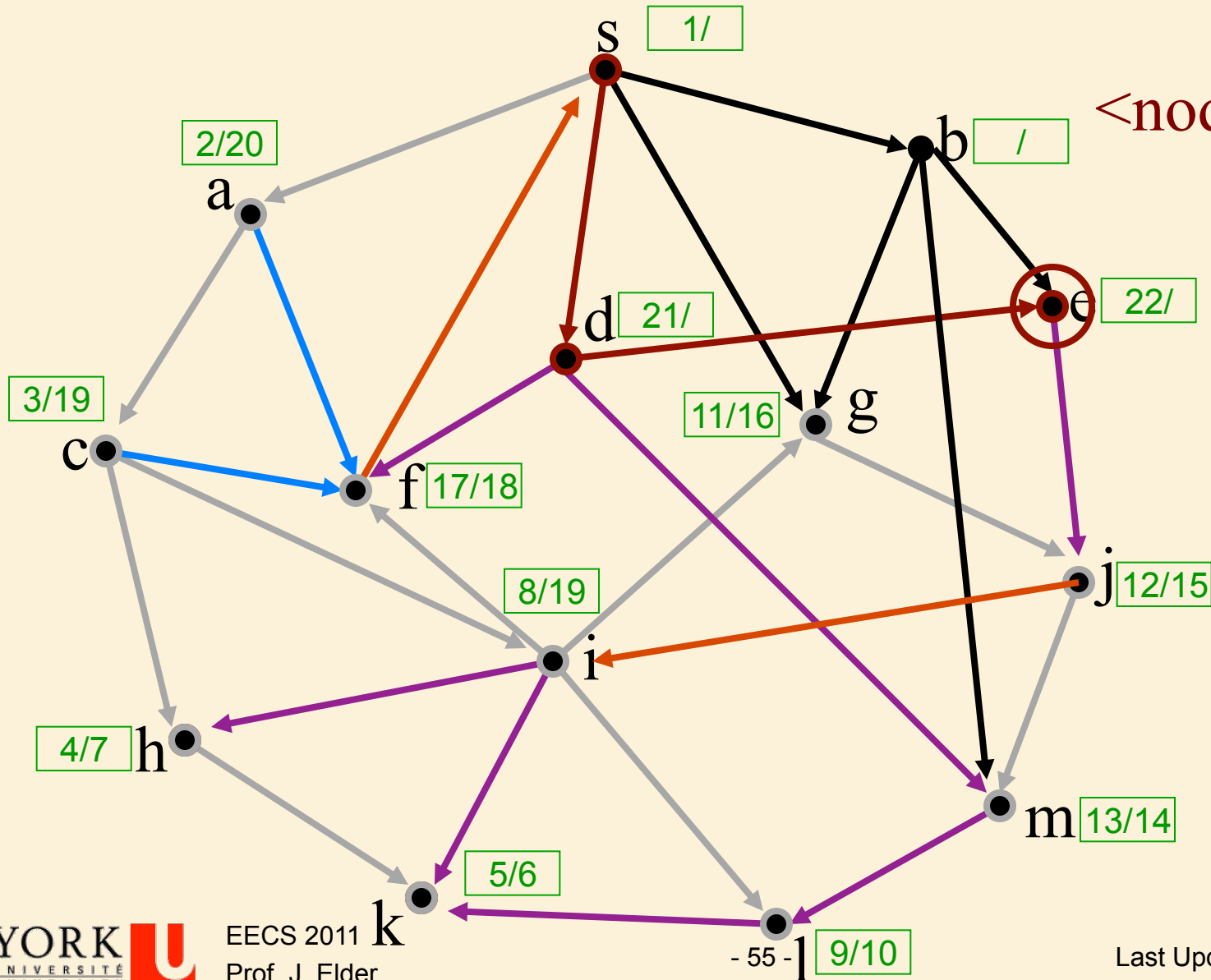
<node,# edges>



DFS

Found
Not Handled
Stack

<node,# edges>

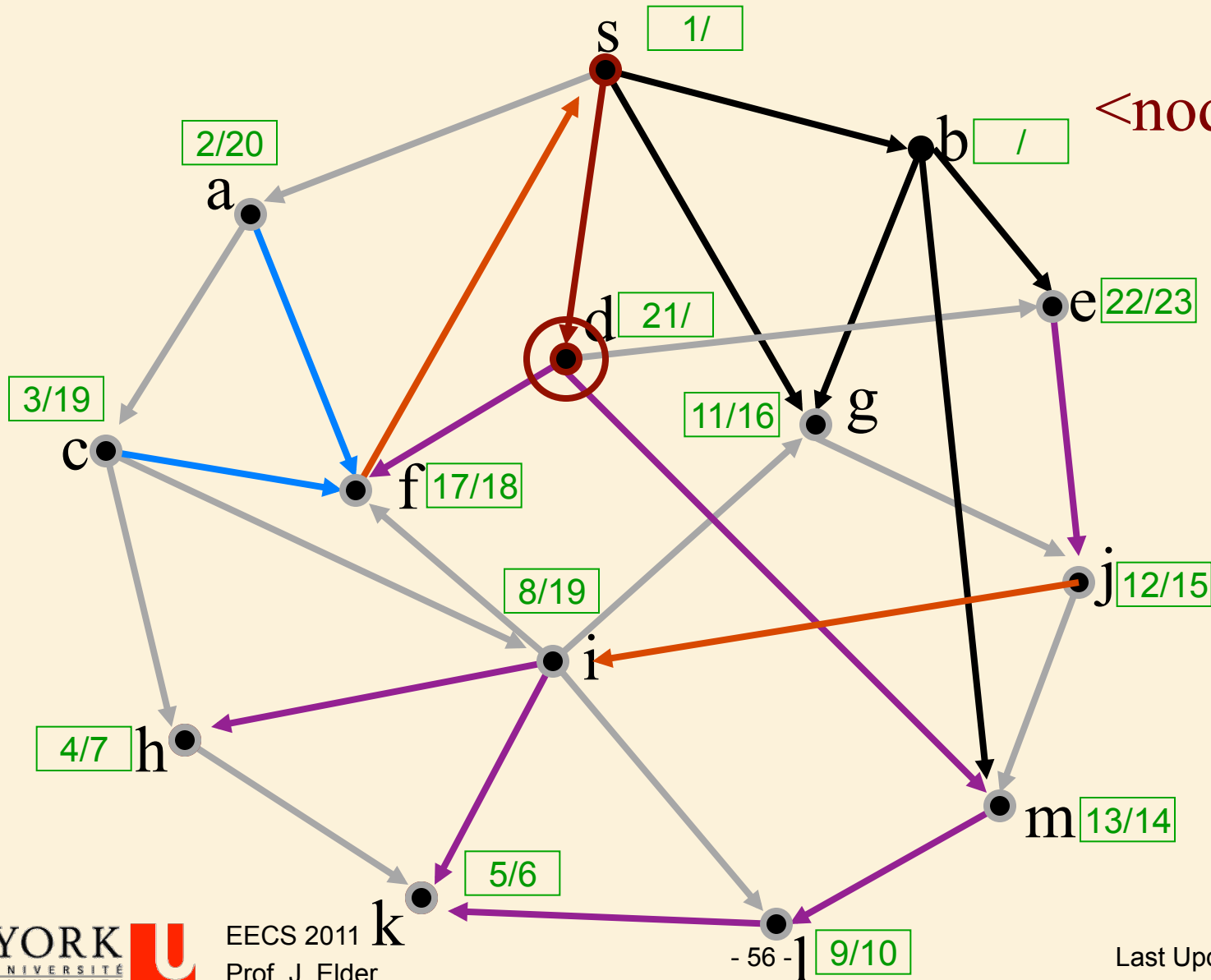


Last Updated April 9, 2015

DFS

Found
Not Handled
Stack

<node,# edges>



Found
Not Handled
Stack

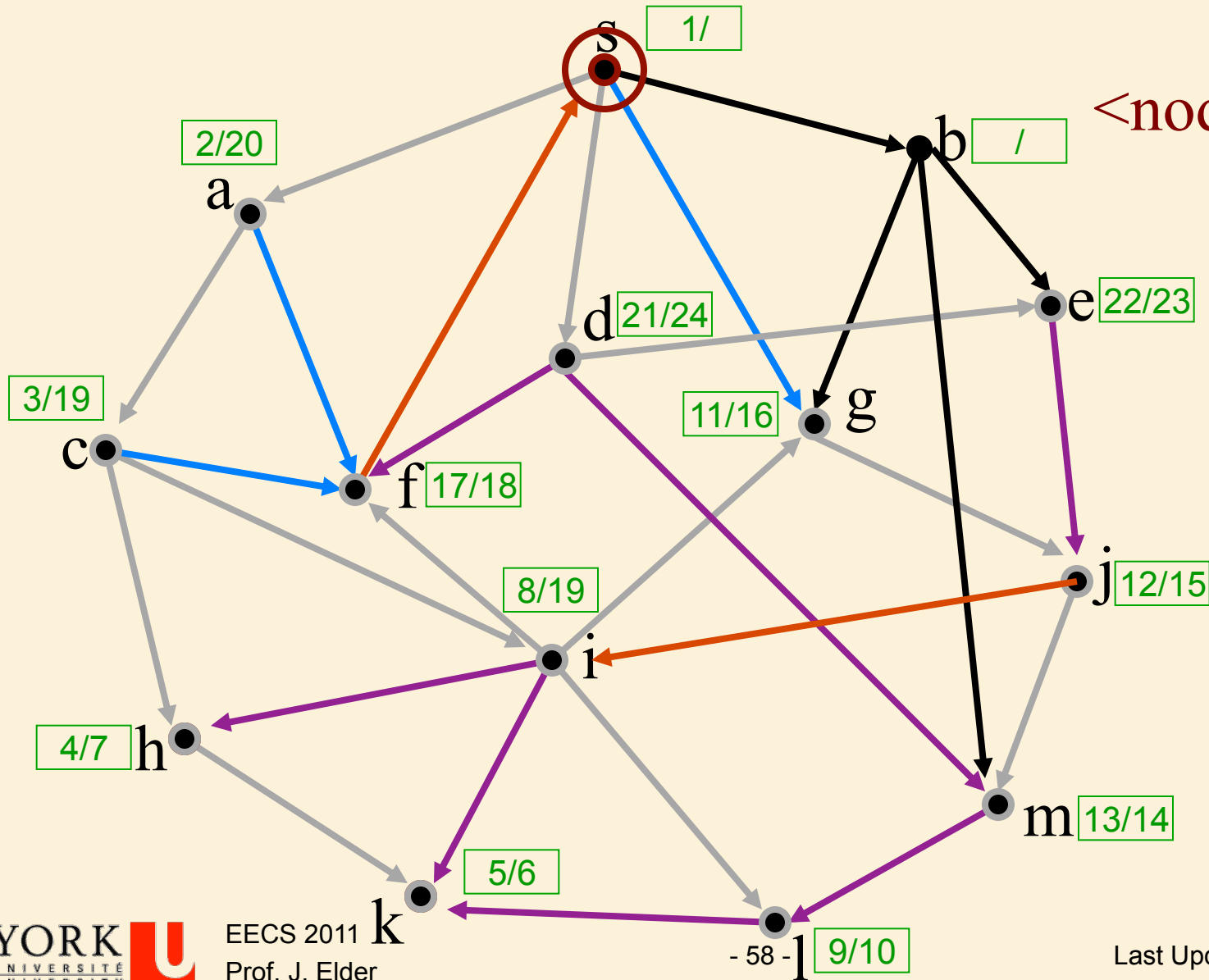
s,2



DFS

Found
Not Handled
Stack

<node,# edges>



s,3

Found Not Handled Stack

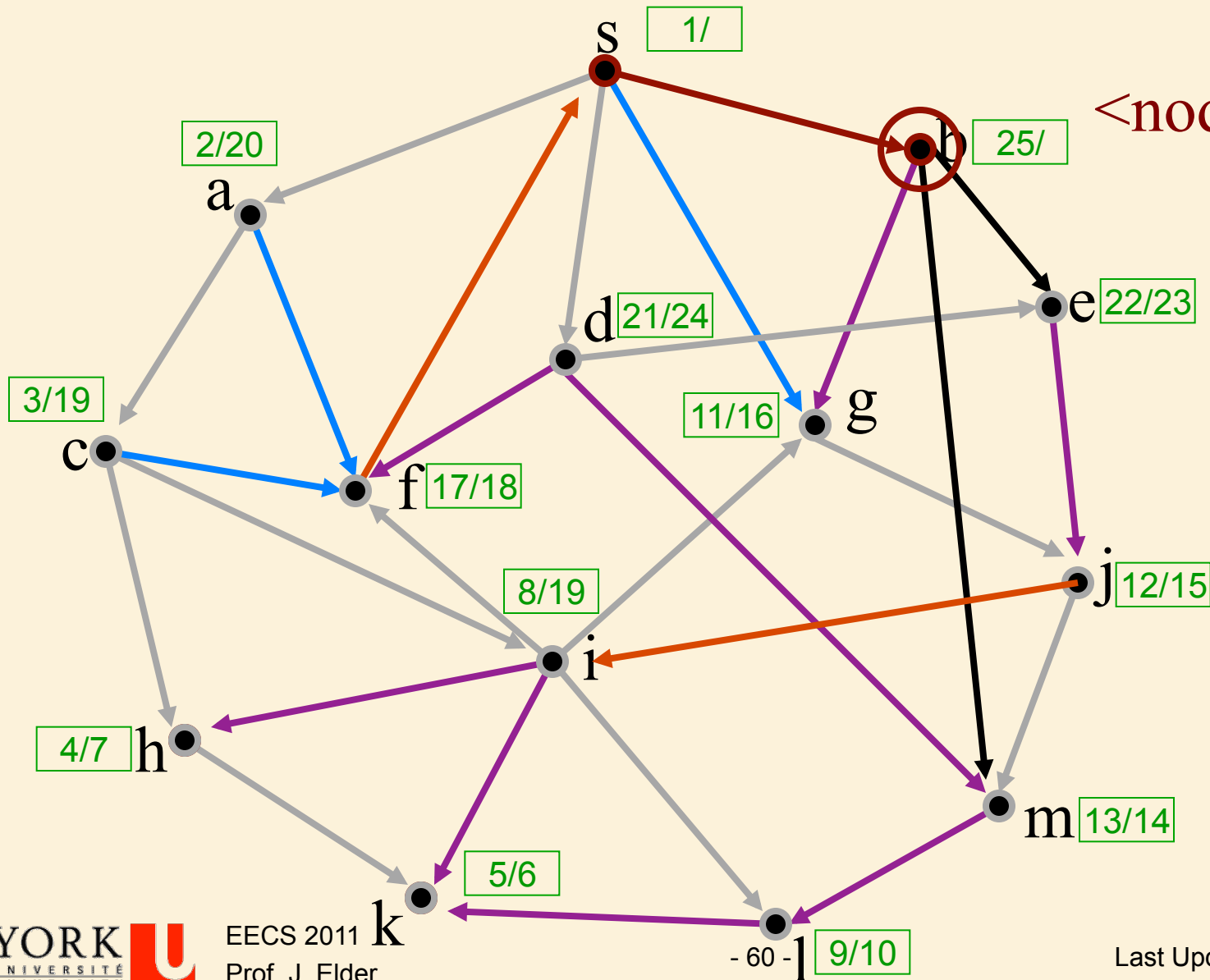
b,_{s,4}⁰



DFS

Found
Not Handled
Stack

<node,# edges>

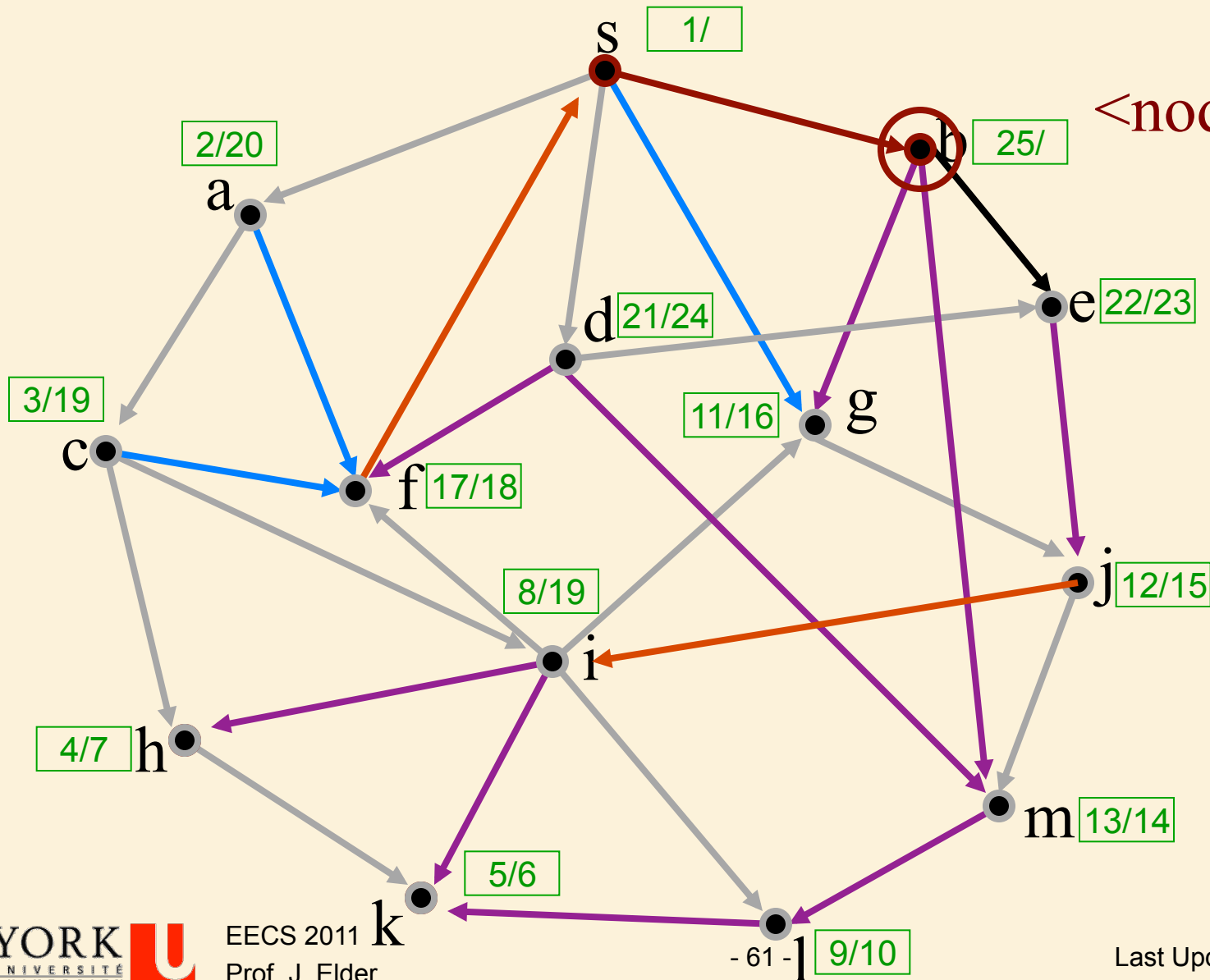


Last Updated April 9, 2015

DFS

Found
Not Handled
Stack

<node,# edges>

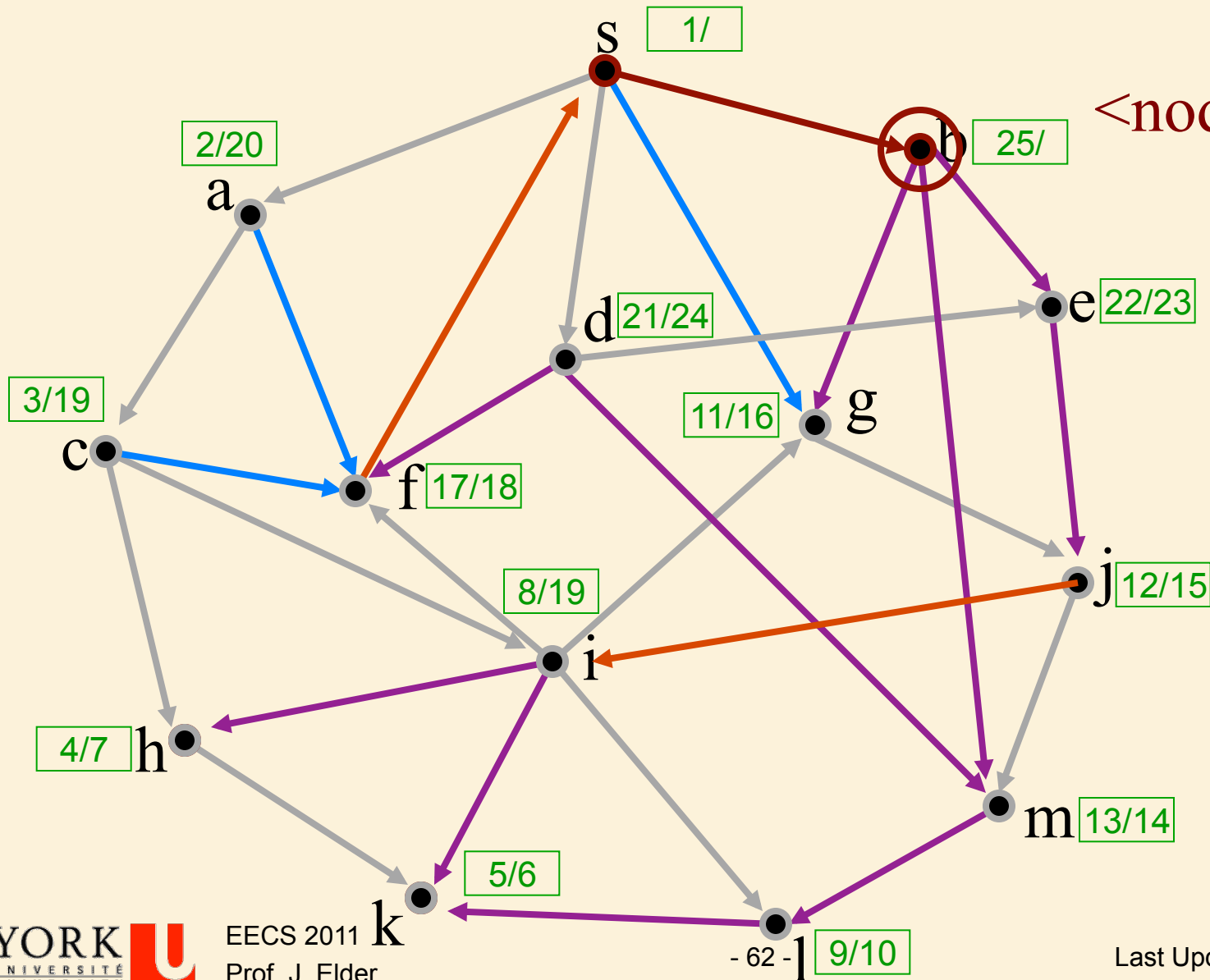


Last Updated April 9, 2015

DFS

Found
Not Handled
Stack

<node,# edges>

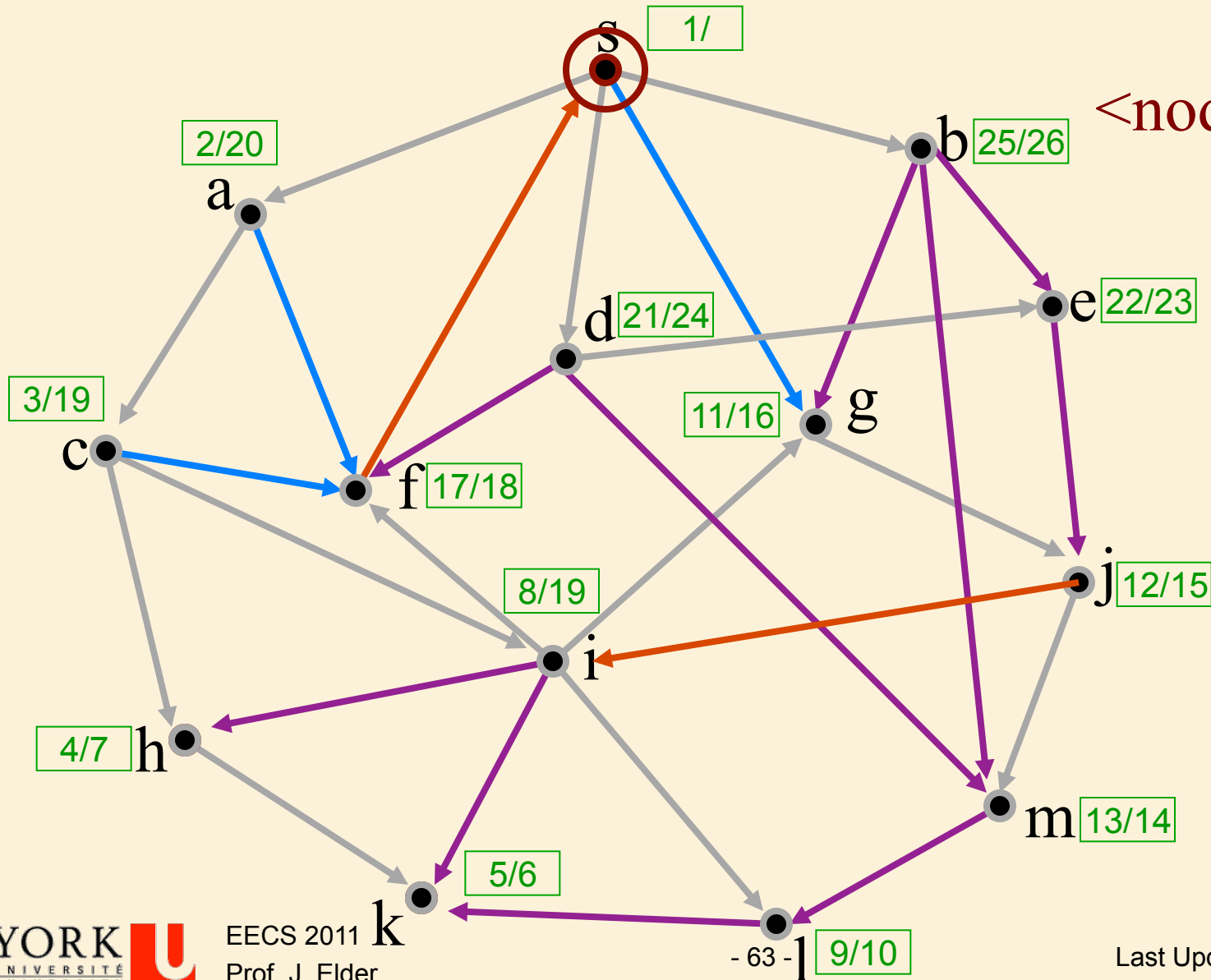


Last Updated April 9, 2015

DFS

Found
Not Handled
Stack

<node,# edges>



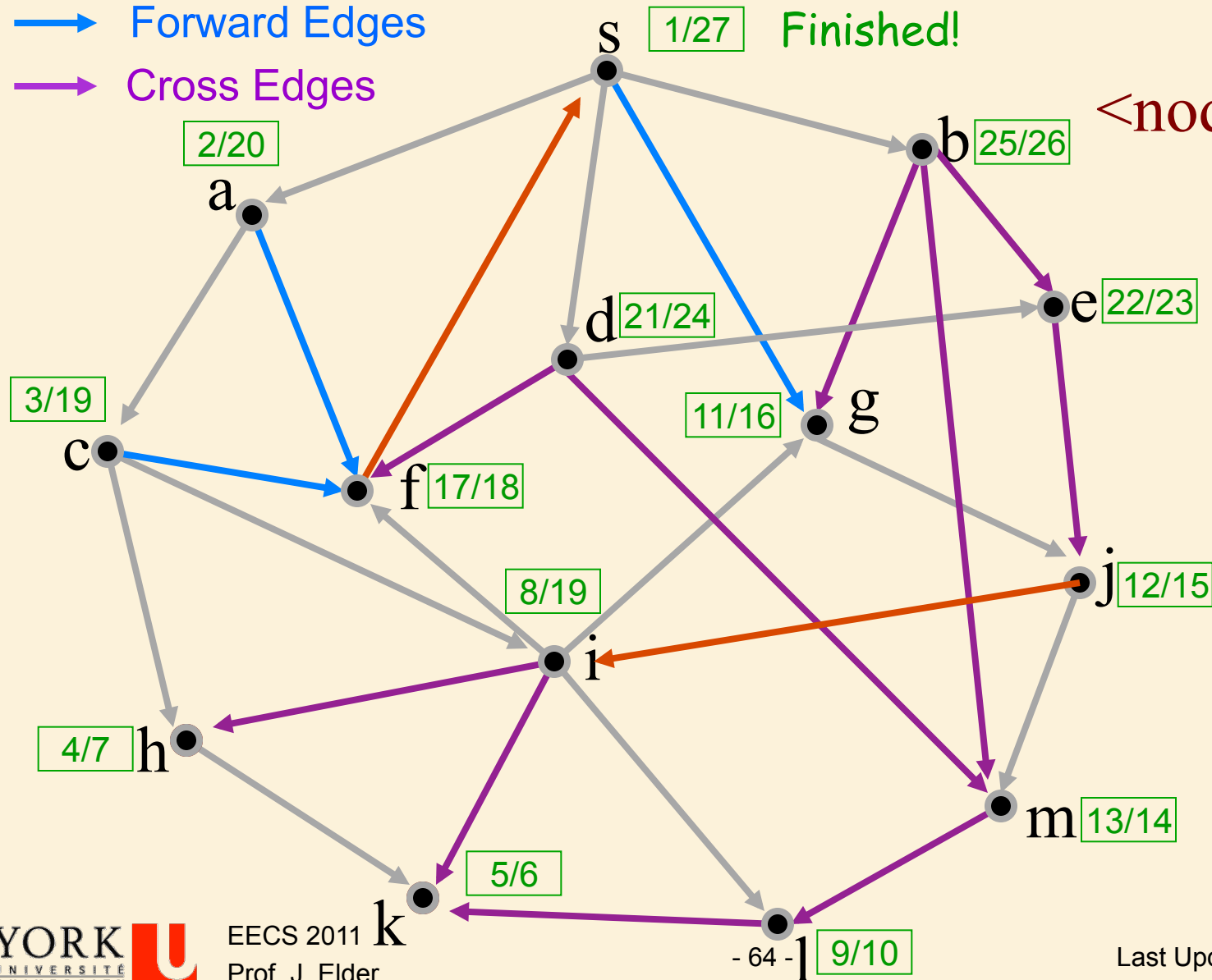
s,4

DFS

Found
Not Handled
Stack

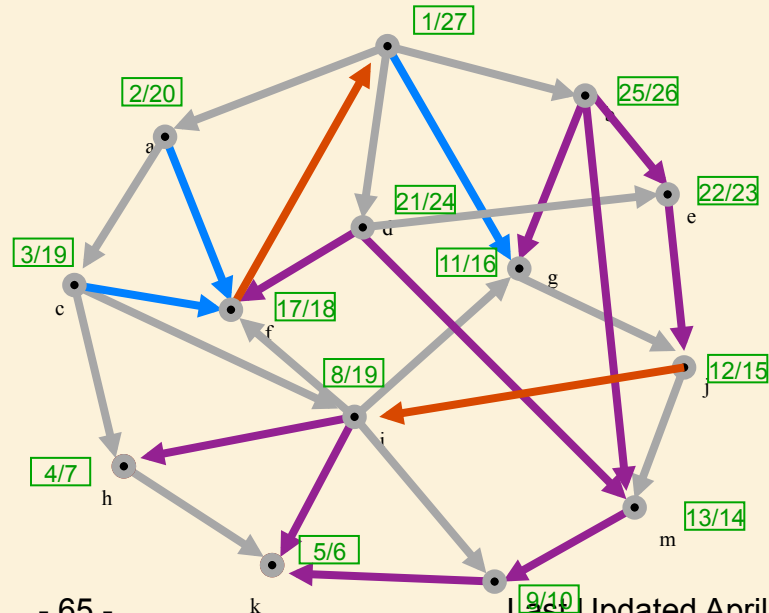
<node,# edges>

- Tree Edges
- Back Edges
- Forward Edges
- Cross Edges



Classification of Edges in DFS

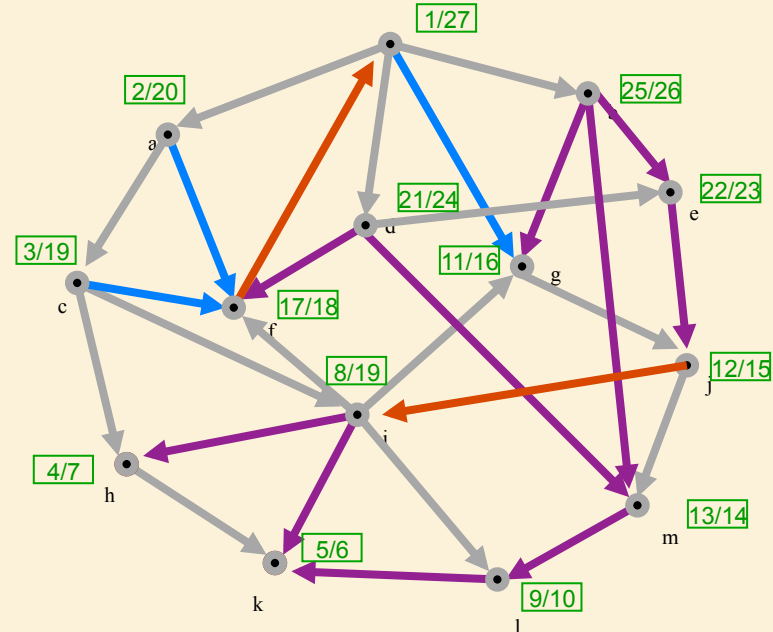
1. **Tree edges** are edges in the depth-first forest G_π . Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v) .
2. **Back edges** are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree.
3. **Forward edges** are non-tree edges (u, v) connecting 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**.

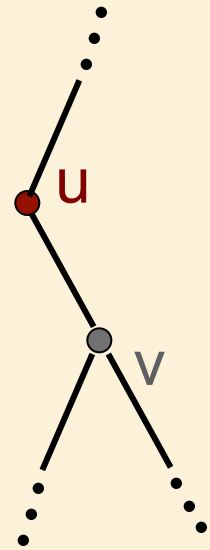


DFS on Undirected Graphs

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

DFS on 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!**



Outline

- DFS Algorithm
- DFS Example
- **DFS Applications**

DFS Algorithm Pattern

DFS-Visit (u)

Precondition: vertex u is undiscovered

Postcondition: all vertices reachable from u have been processed

$\text{colour}[u] \leftarrow \text{RED}$

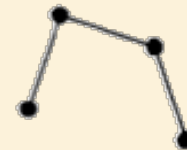
for each $v \in \text{Adj}[u]$ //explore edge (u,v)

 if $\text{color}[v] = \text{BLACK}$

 DFS-Visit(v)

$\text{colour}[u] \leftarrow \text{GRAY}$

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



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

(assuming adjacency list structure)

DFS Application 1: Path Finding

- The DFS pattern can be used to find a path between two given vertices u and z , if one exists
- We use a stack to keep track of the current path
- If the destination vertex z is encountered, we return the path as the contents of the stack

DFS-Path ($u, z, stack$)

Precondition: u and z are vertices in a graph, $stack$ contains current path

Postcondition: returns true if path from u to z exists, $stack$ contains path

colour[u] $\leftarrow$ RED

push u onto $stack$

if $u = z$

 return TRUE

for each $v \in \text{Adj}[u]$ //explore edge (u, v)

 if color[v] = BLACK

 if DFS-Path($v, z, stack$)

 return TRUE

colour[u] $\leftarrow$ GRAY

pop u from $stack$

return FALSE

DFS Application 2: Cycle Finding

- The DFS pattern can be used to determine whether a graph is acyclic.
- If a back edge is encountered, we return true.

DFS-Cycle (u)

Precondition: u is a vertex in a graph G

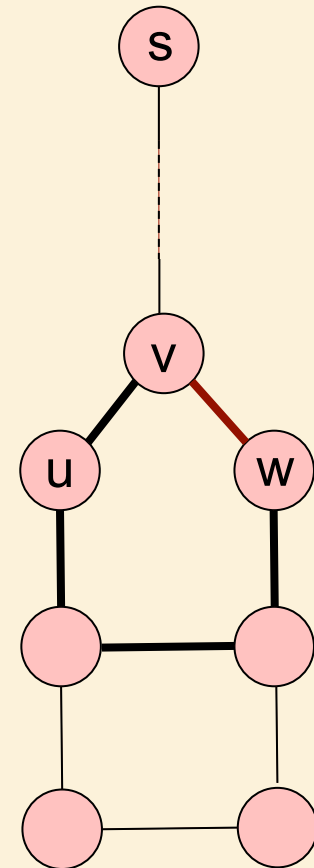
Postcondition: returns true if there is a cycle reachable from u .

```
colour[u] ← RED
for each  $v \in \text{Adj}[u]$  //explore edge ( $u,v$ )
    if color[v] = RED //back edge
        return true
    else if color[v] = BLACK
        if DFS-Cycle( $v$ )
            return true

colour[u] ← GRAY
return false
```


Why must DFS on a graph with a cycle generate a back edge?

- Suppose that vertex s is in a connected component S that contains a cycle C .
- Since all vertices in S are reachable from s , they will all be visited by a DFS from s .
- Let v be the first vertex in C reached by a DFS from s .
- There are two vertices u and w adjacent to v on the cycle C .
- wlog, suppose u is explored first.
- Since w is reachable from u , w will eventually be discovered.
- When exploring w 's adjacency list, the back-edge (w, v) will be discovered.



Outline

- DFS Algorithm
- DFS Example
- DFS Applications