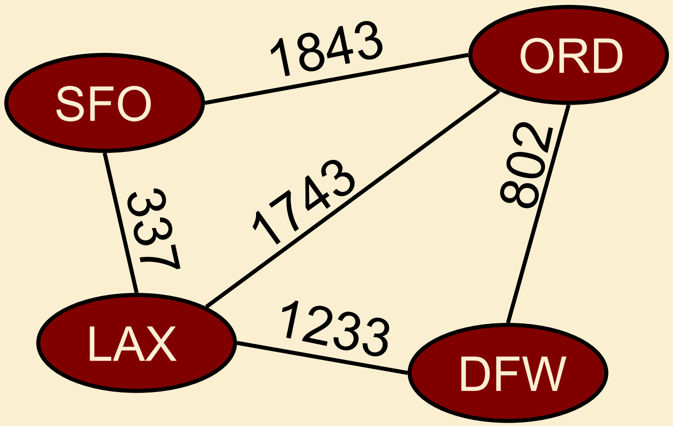
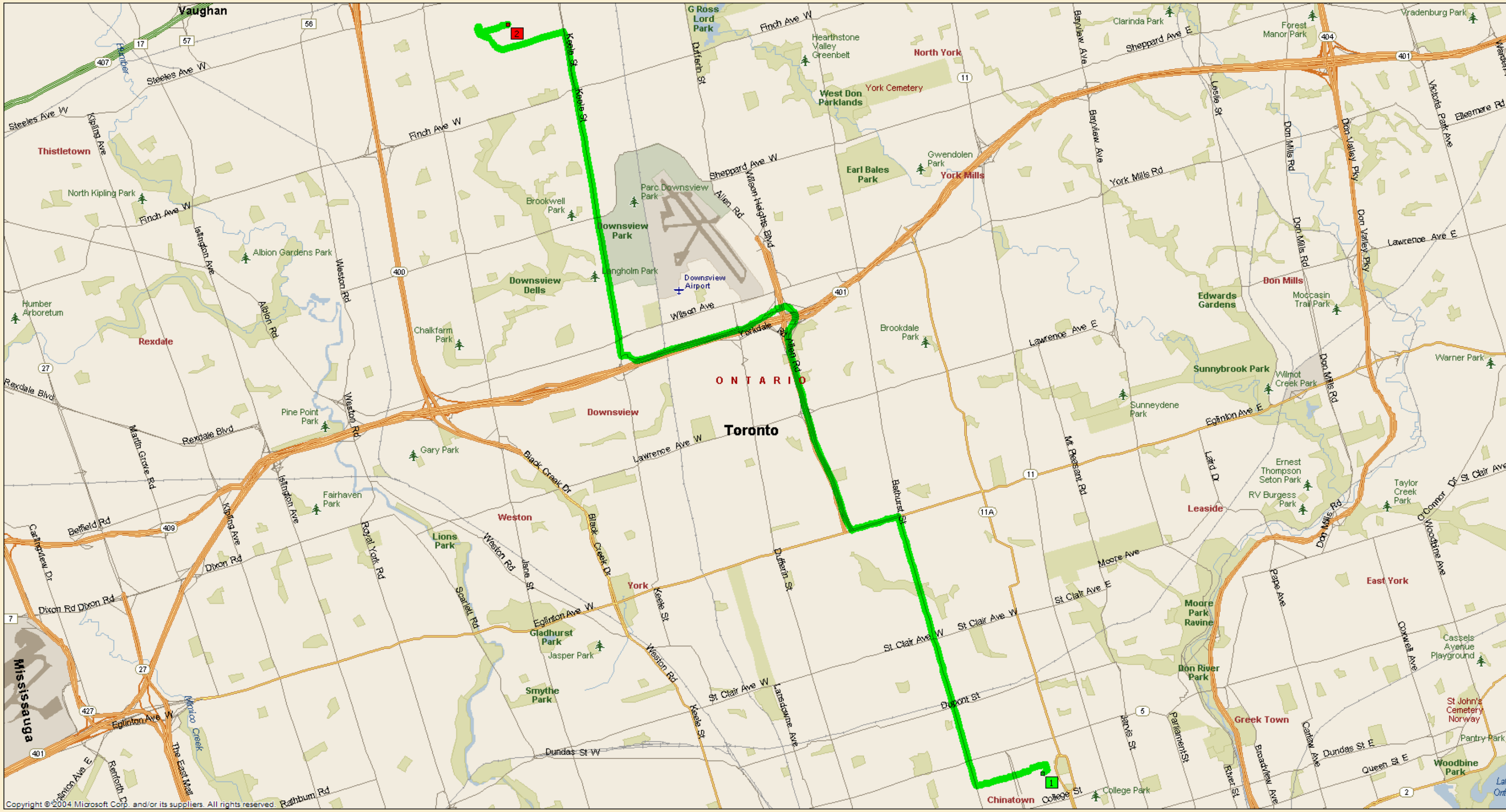
Graphs – Depth First Search



Graph Search Algorithms



Outcomes

- By understanding this lecture, you should be able to:
  - Label a graph according to the order in which vertices are discovered, explored from and finished in a depth-first search.
  - Classify edges of the depth-first search as tree edges, back edges, forward edges and cross edges
  - Implement depth-first search
  - Demonstrate simple applications of depth-first search

Outline

- DFS Algorithm
- DFS Example
- DFS Applications

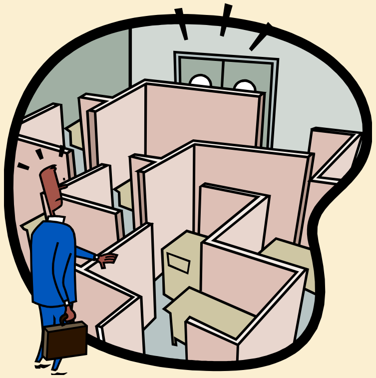
Outline

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

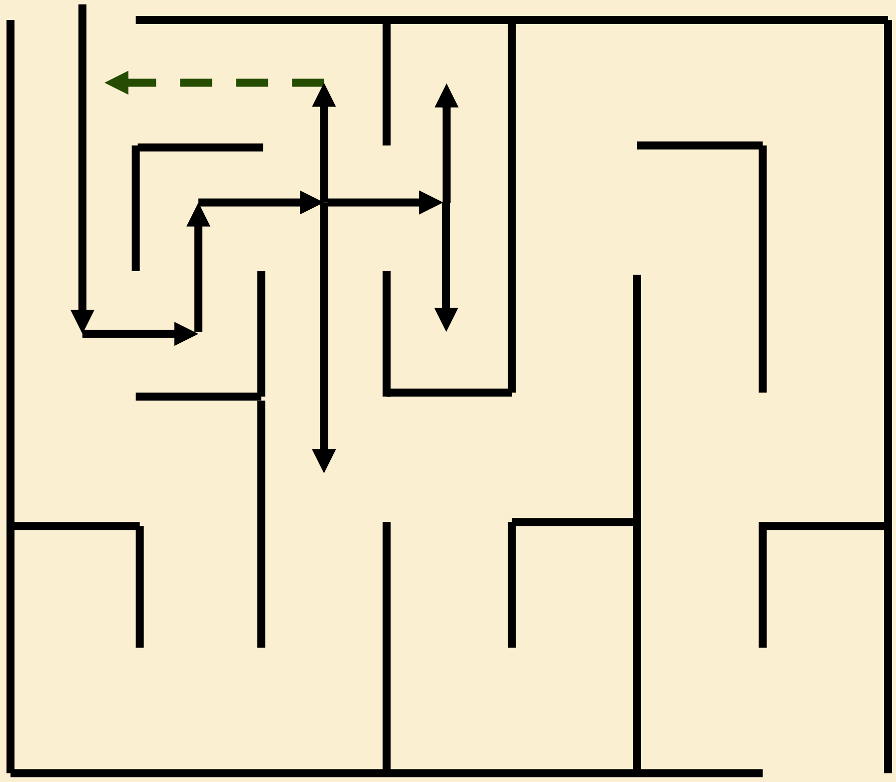
Depth First Search (DFS)

- 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
  - Identifying nodes that are reachable from a specific node *v*
  - Detecting cycles
  - Extracting strongly connected components
  - Topological sorts

Depth-First Search



- The DFS algorithm is similar to a classic strategy for exploring a maze
  - We mark each intersection, corner and dead end (vertex) visited
  - We mark each corridor (edge) traversed
  - We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)



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 (Discovered everything reachable from it).

DFS Example on Undirected Graph

unexplored

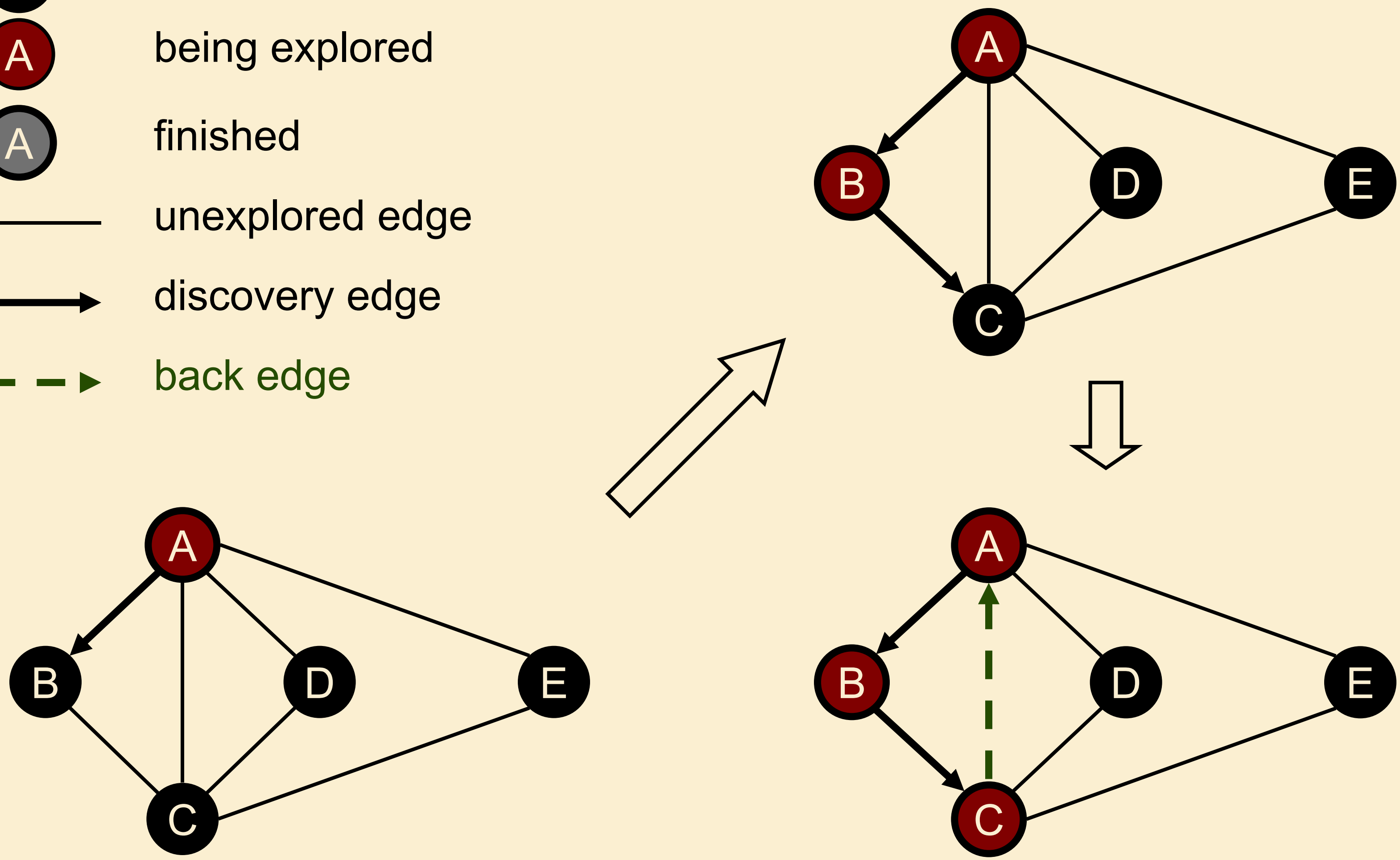
being explored

finished

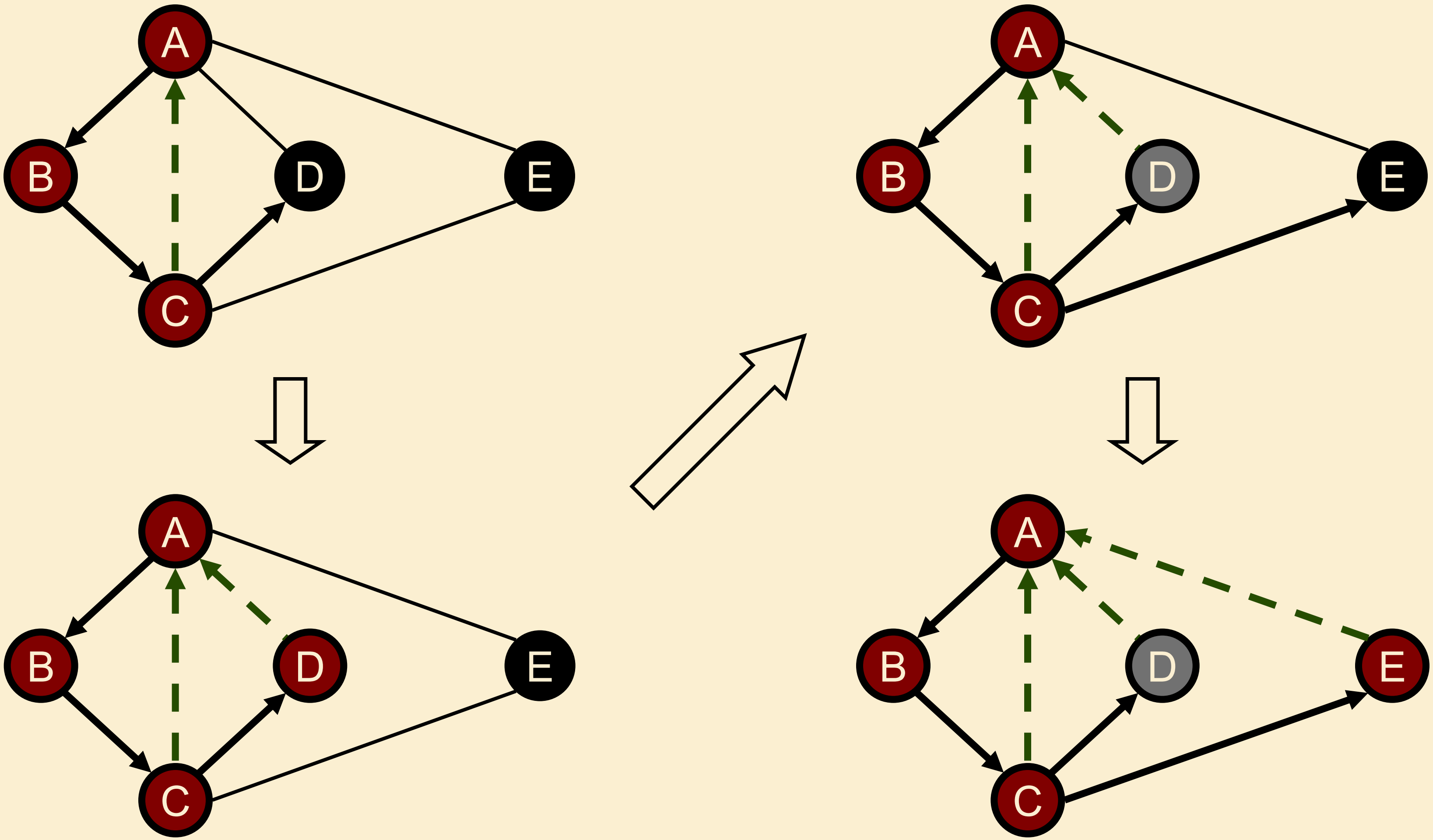
unexplored edge

discovery edge

back edge



Example (cont.)



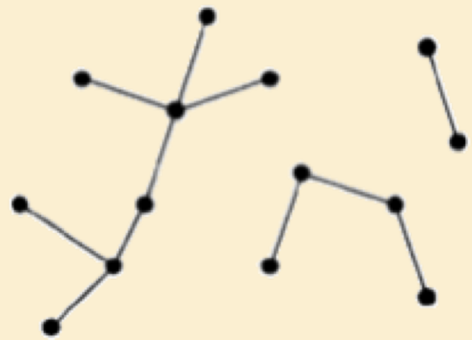
DFS Algorithm Pattern

DFS(G)

Precondition: G is a graph

Postcondition: all vertices in G have been visited

```
for each vertex u ∈ V[G]
        color[u] = BLACK //initialize vertex
for each vertex u ∈ V[G]
        if color[u] = 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] $\leftarrow$ RED

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

if color[v] = BLACK

DFS-Visit(v)

colour[u] $\leftarrow$ *GRAY*

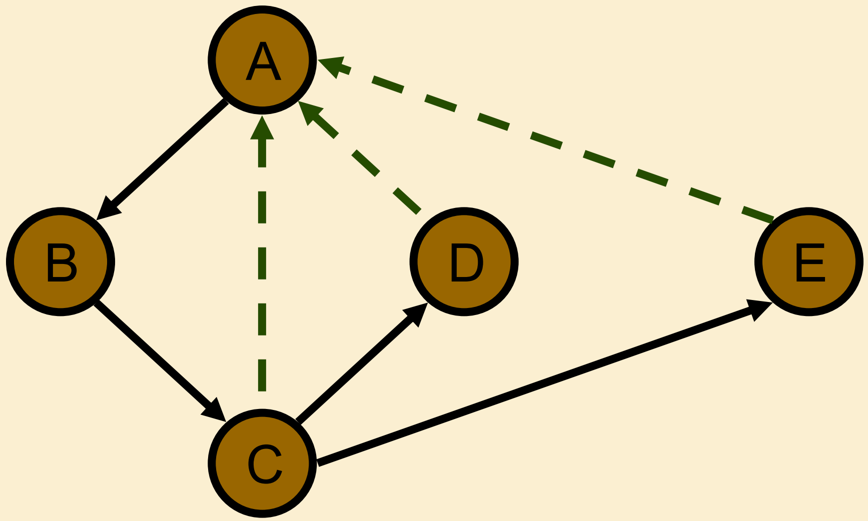
Properties of DFS

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

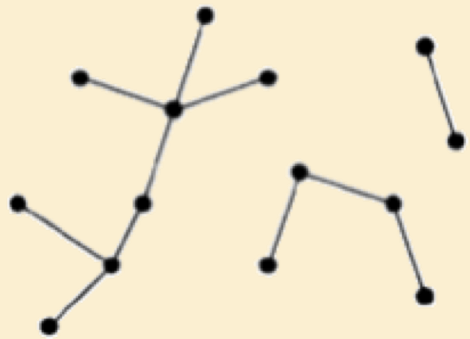
DFS(G)

Precondition: G is a graph

Postcondition: all vertices in G have been visited

```
for each vertex u ∈ V[G]
        color[u] = BLACK //initialize vertex
for each vertex u ∈ 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

colour[u] $\leftarrow$ RED

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

if color[v] = BLACK

DFS-Visit(v)

$colour[u] \leftarrow GRAY$

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

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.

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

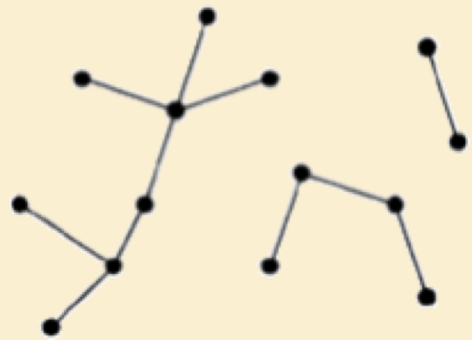
DFS Algorithm with Discovery and Finish Times

DFS(G)

Precondition: G is a graph

Postcondition: all vertices in G have been visited

```
for each vertex u ∈ V[G]
        color[u] = BLACK //initialize vertex
time ← 0
for each vertex u ∈ 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

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

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

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

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

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

DFS-Visit(v)

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

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

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

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 π[u]
  - Label edges in the graph according to their role in the search
    - **Discovery 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

End of Lecture

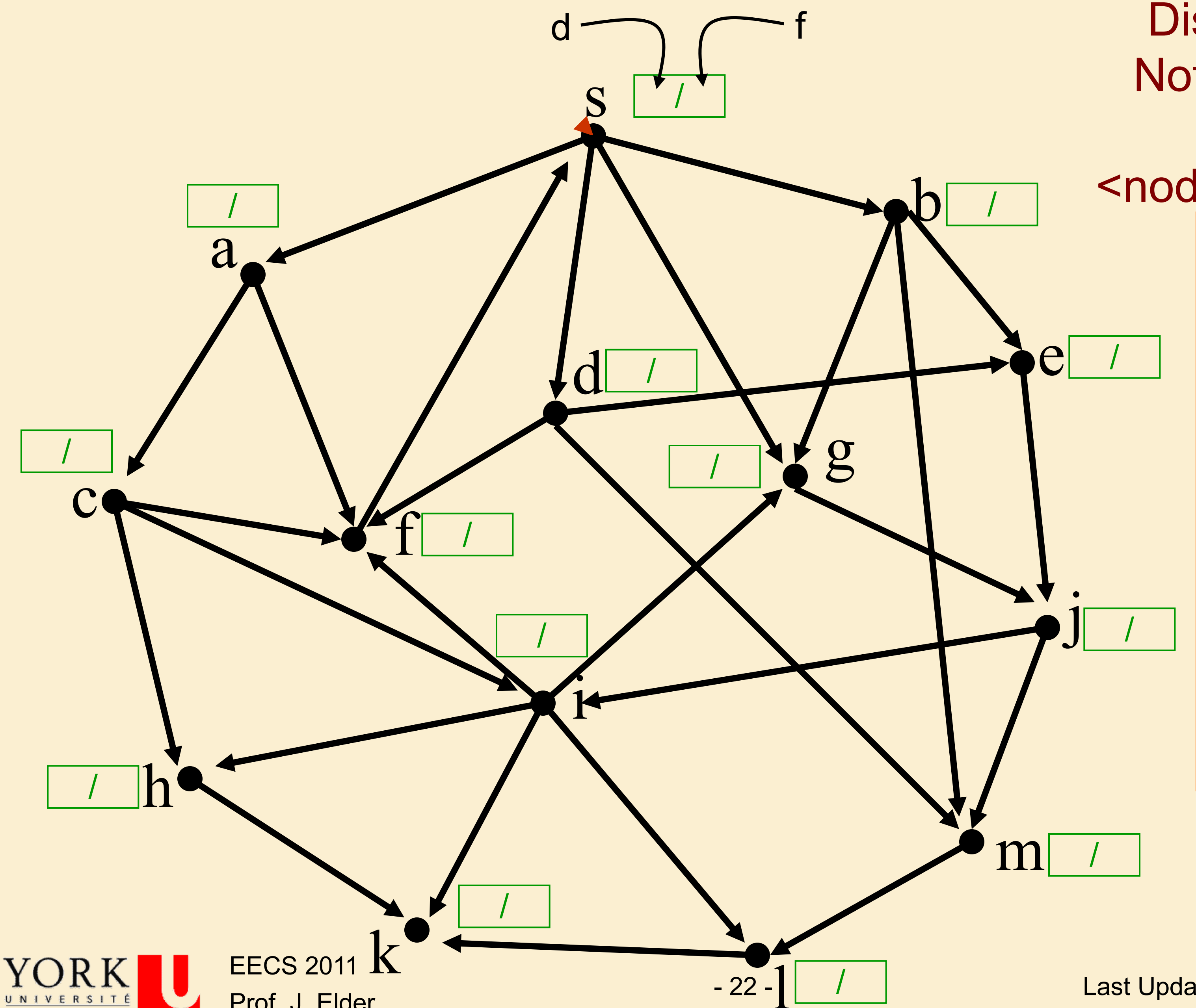
March 27, 2018

Outline

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

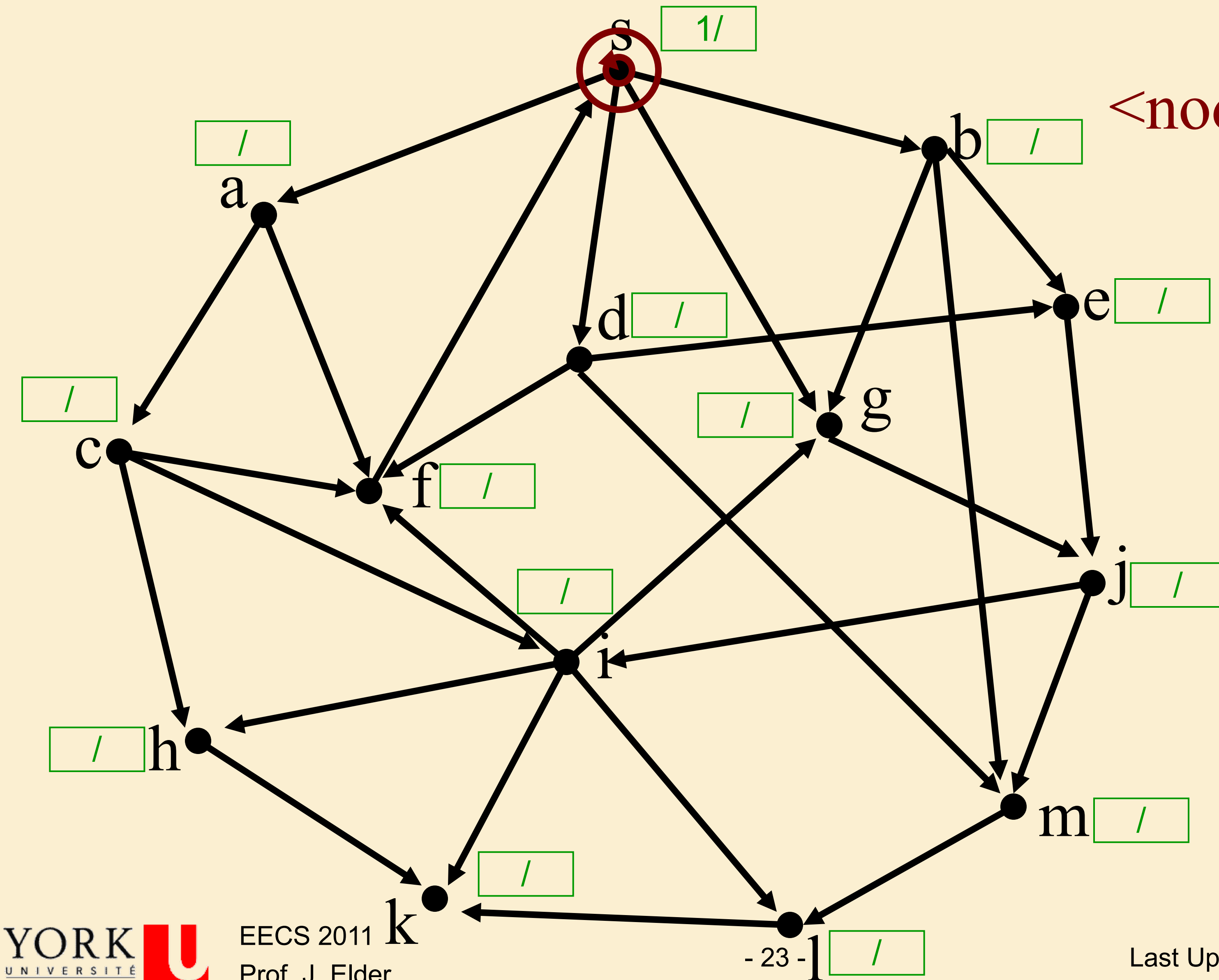
DFS

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



Discovered
Not Finished
Stack
<node,# edges>

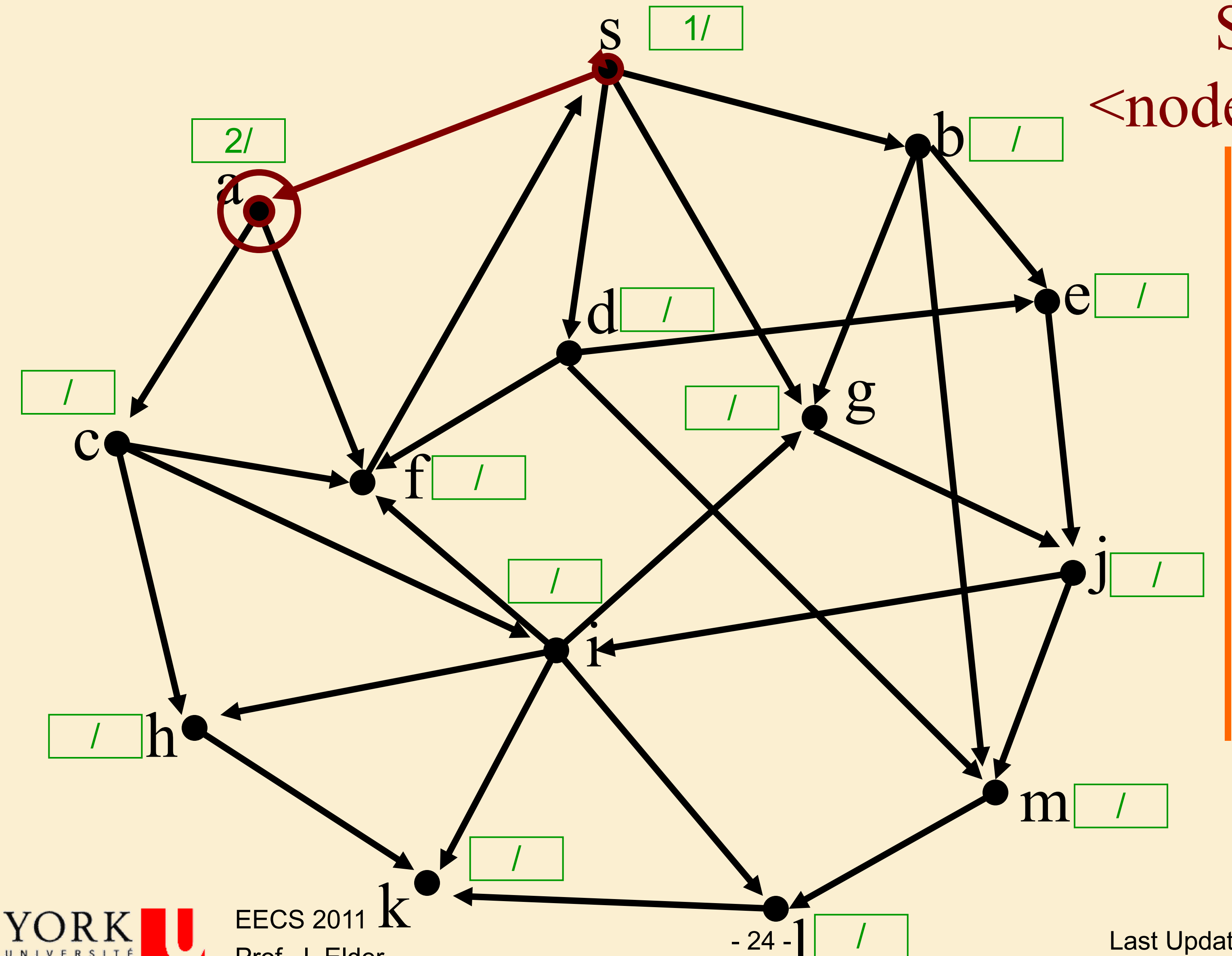
DFS



Discovered
Not Finished
Stack
<node,# edges>

s,0

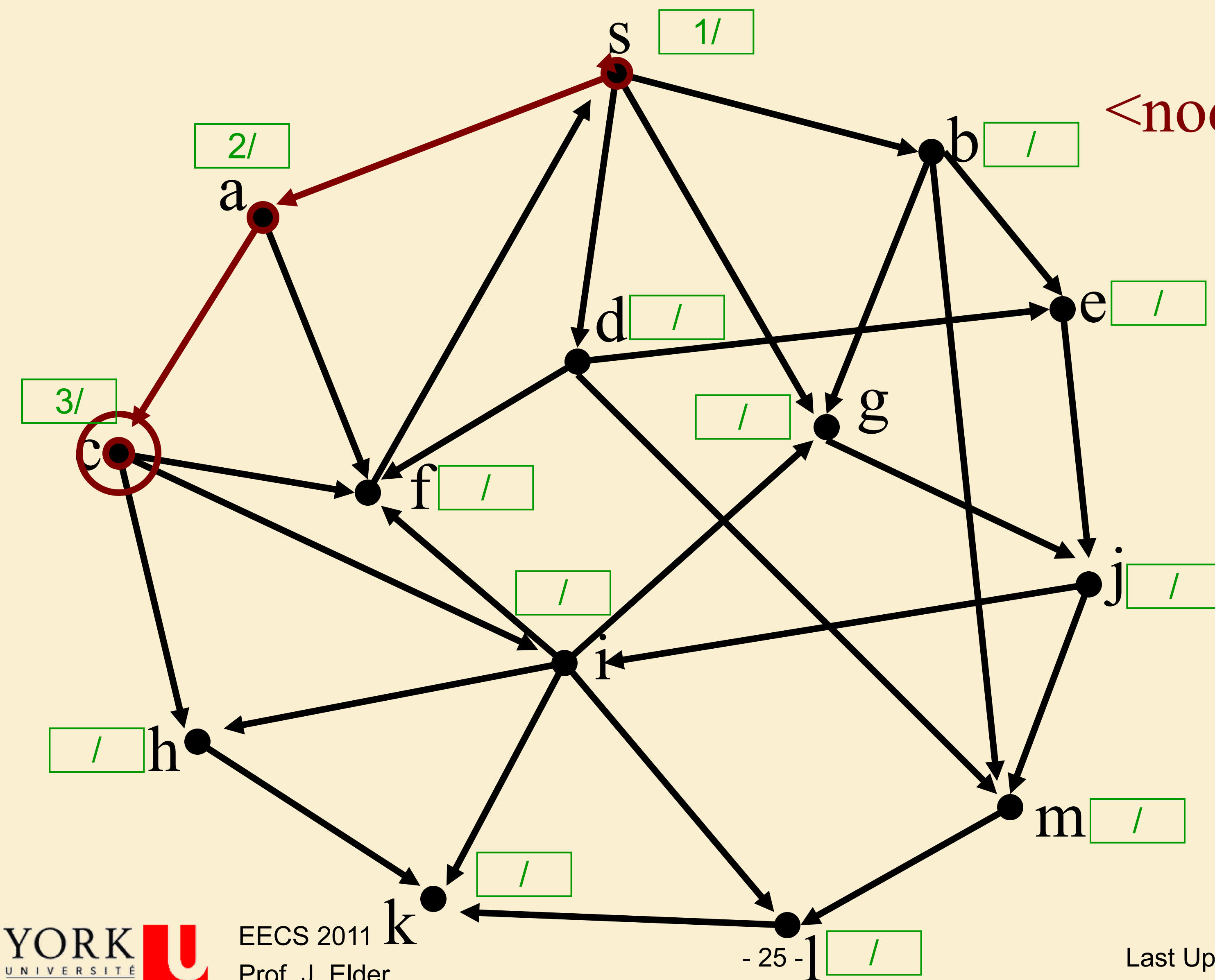
DFS



Discovered
Not Finished
Stack
<node,# edges>

a,0
s,1

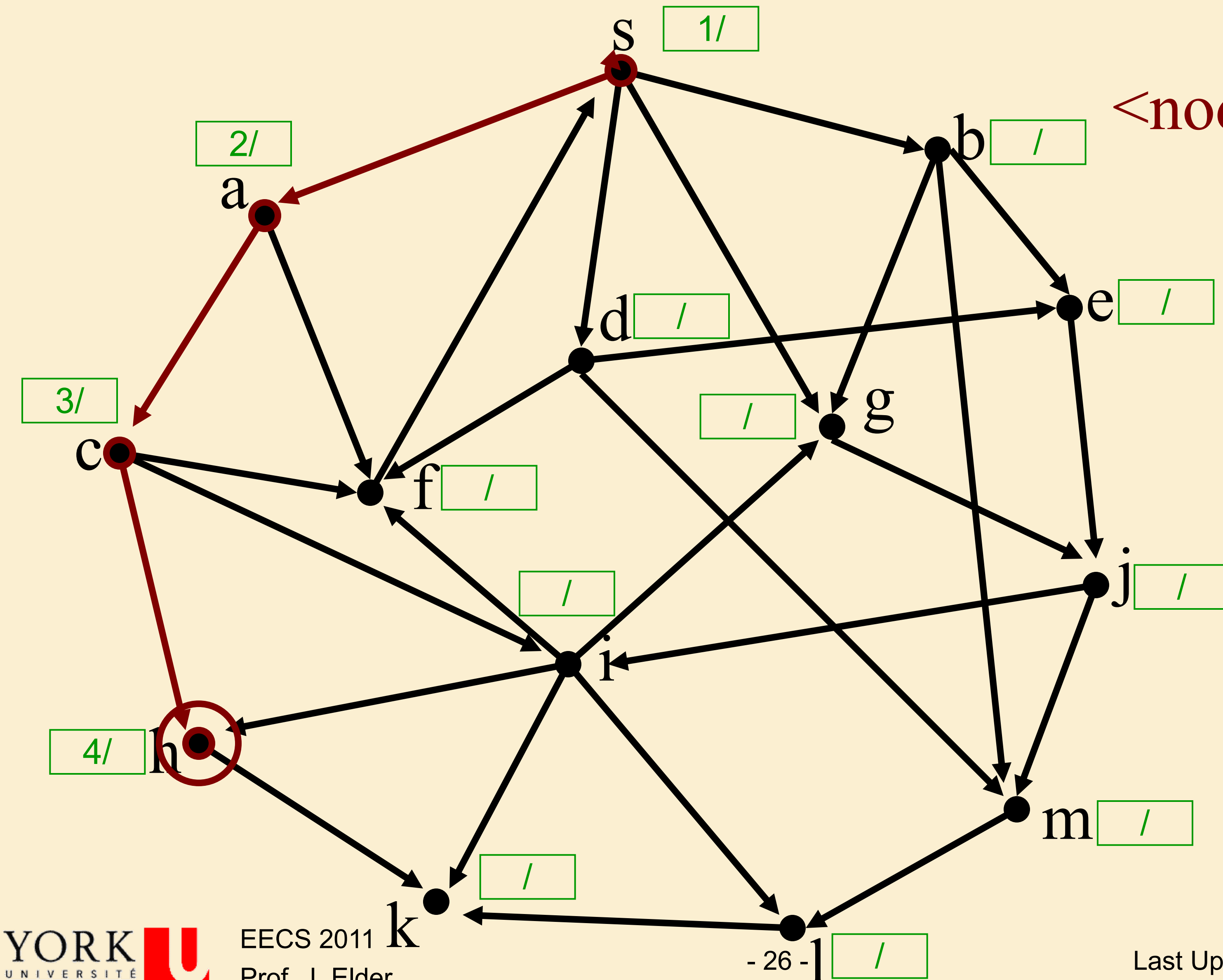
DFS



Discovered
Not Finished
Stack
<node,# edges>

c,0
a,1
s,1

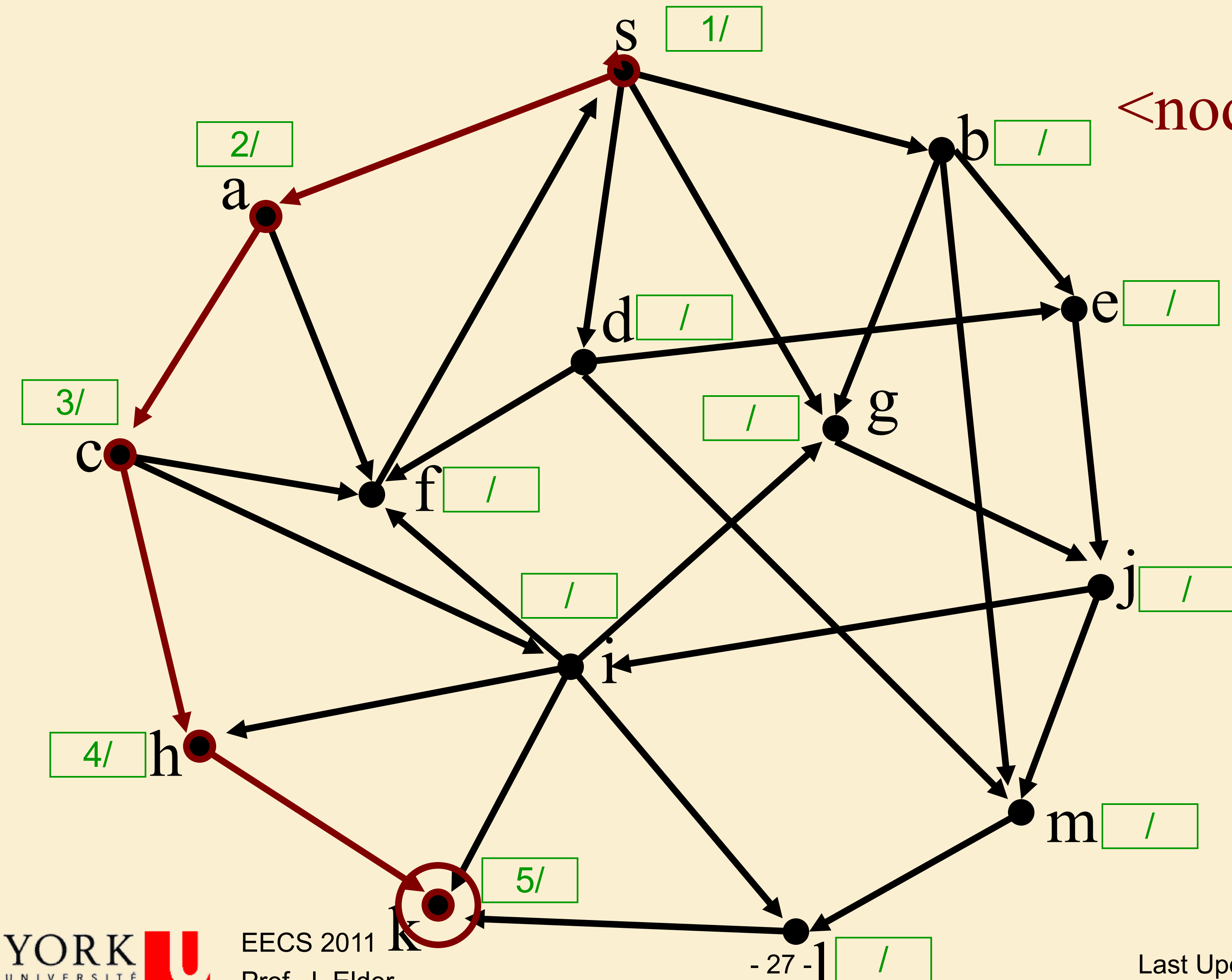
DFS



Discovered
Not Finished
Stack
<node,# edges>

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

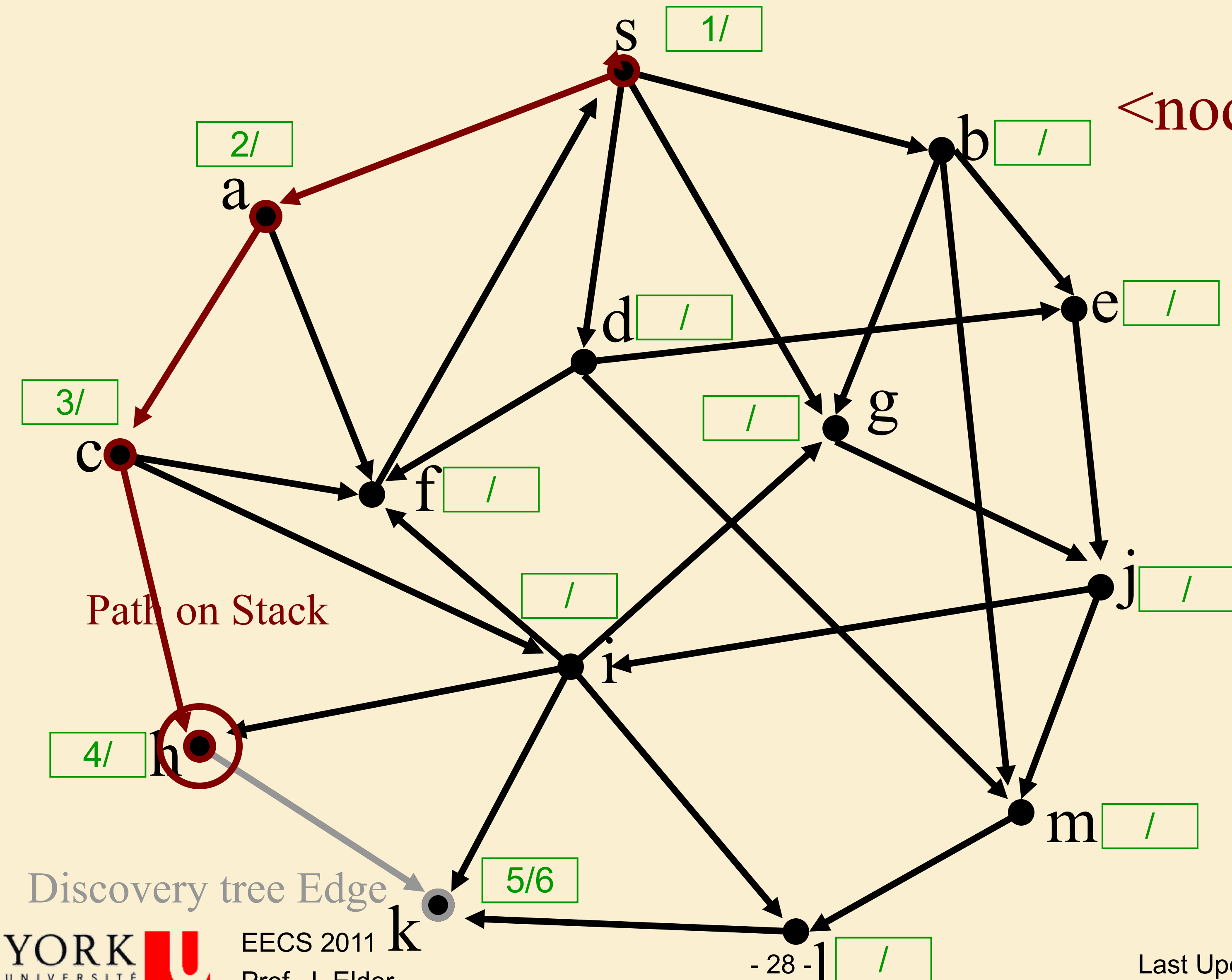
DFS



Discovered
Not Finished
Stack
<node,# edges>

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

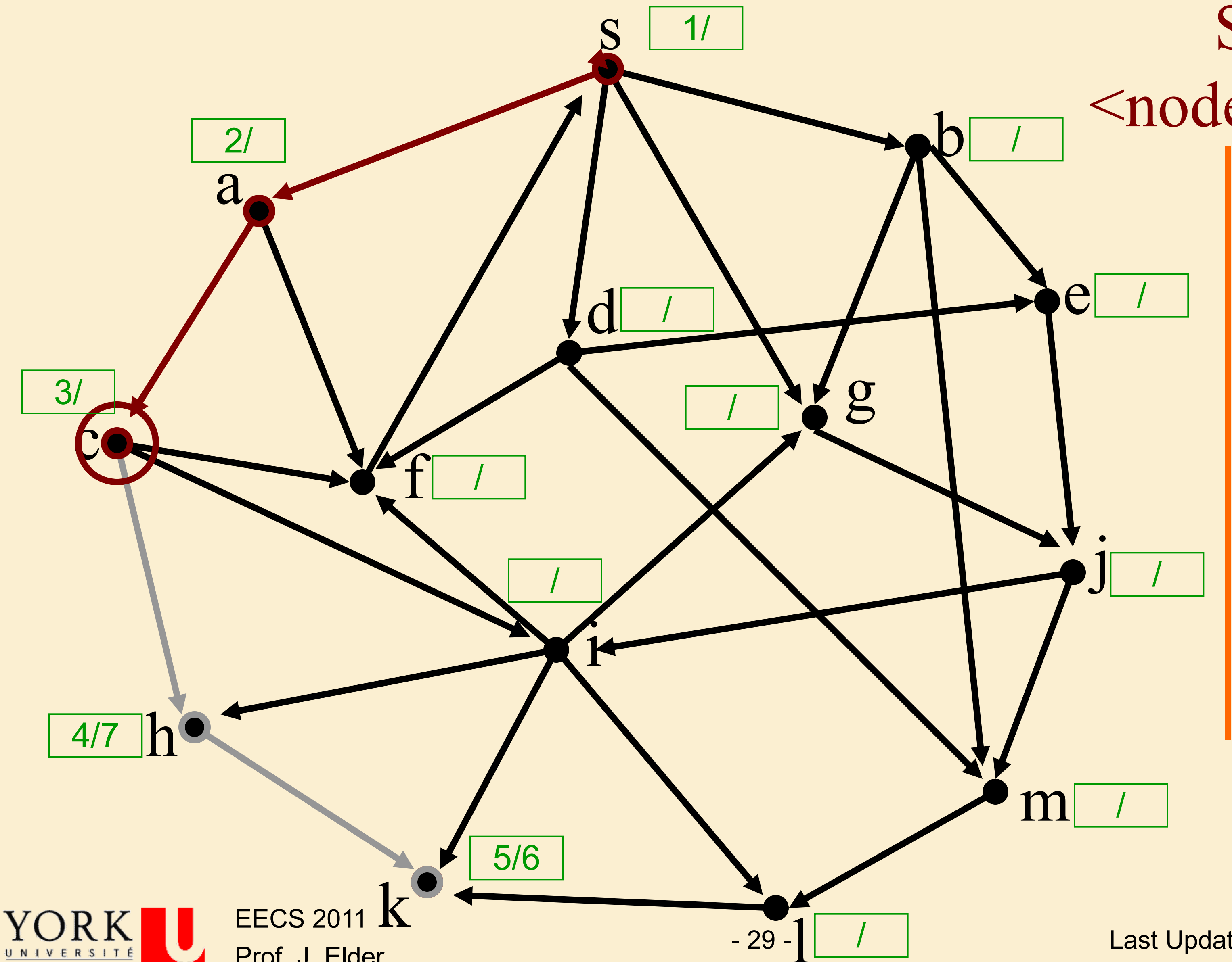
DFS



Discovered
Not Finished
Stack
<node,# edges>

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

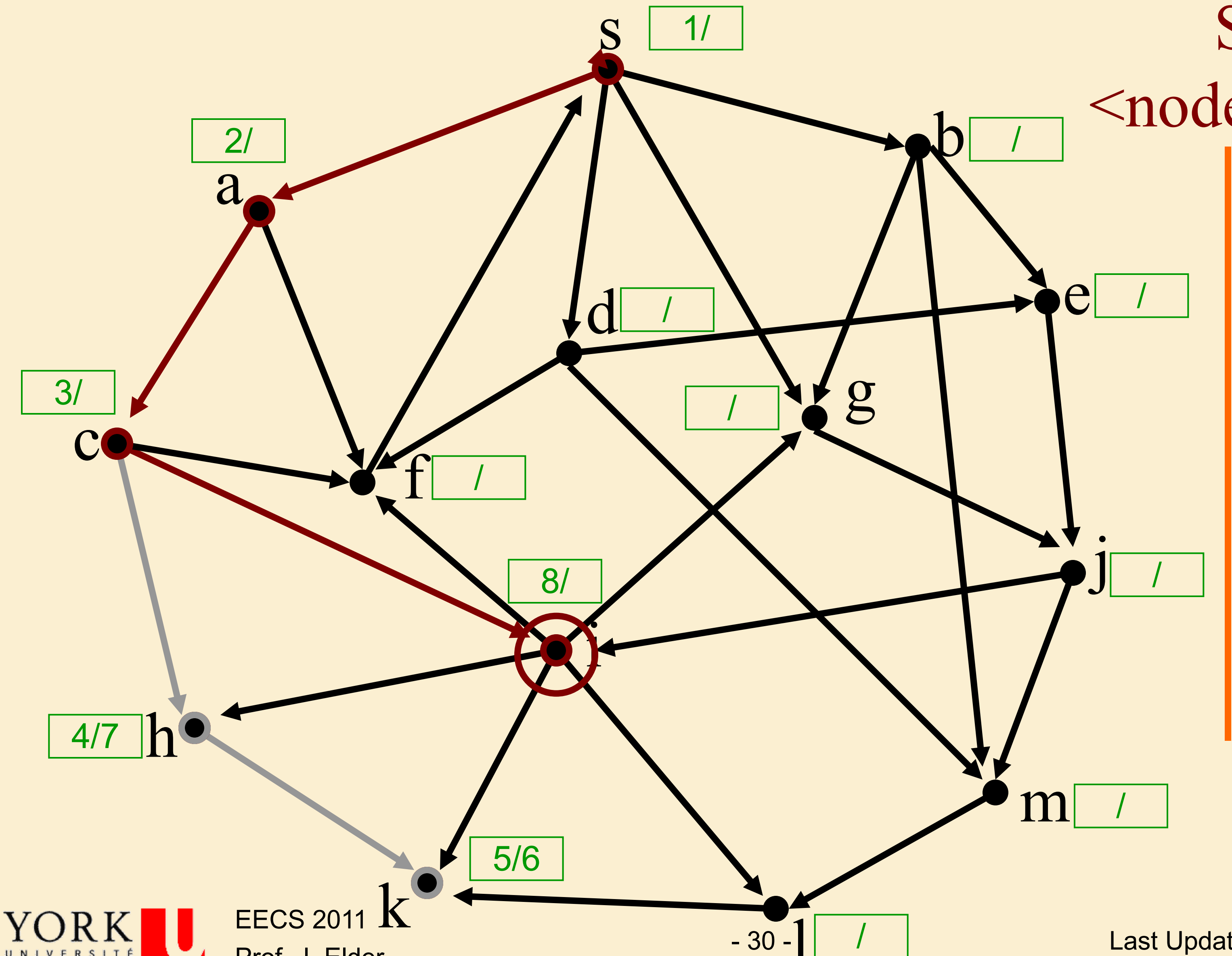
DFS



Discovered
Not Finished
Stack
<node,# edges>

c, 1
a, 1
s, 1

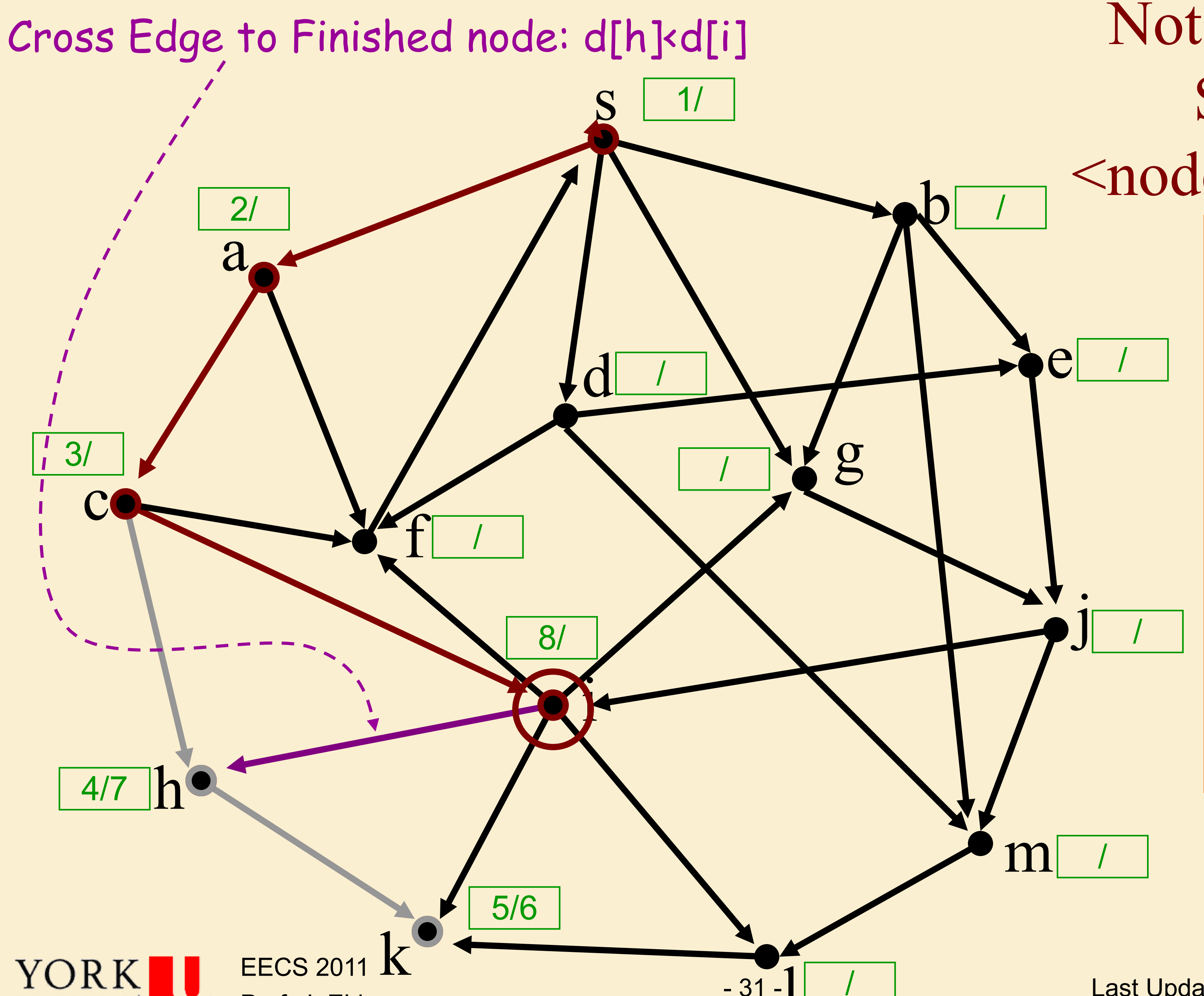
DFS



Discovered
Not Finished
Stack
<node,# edges>

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

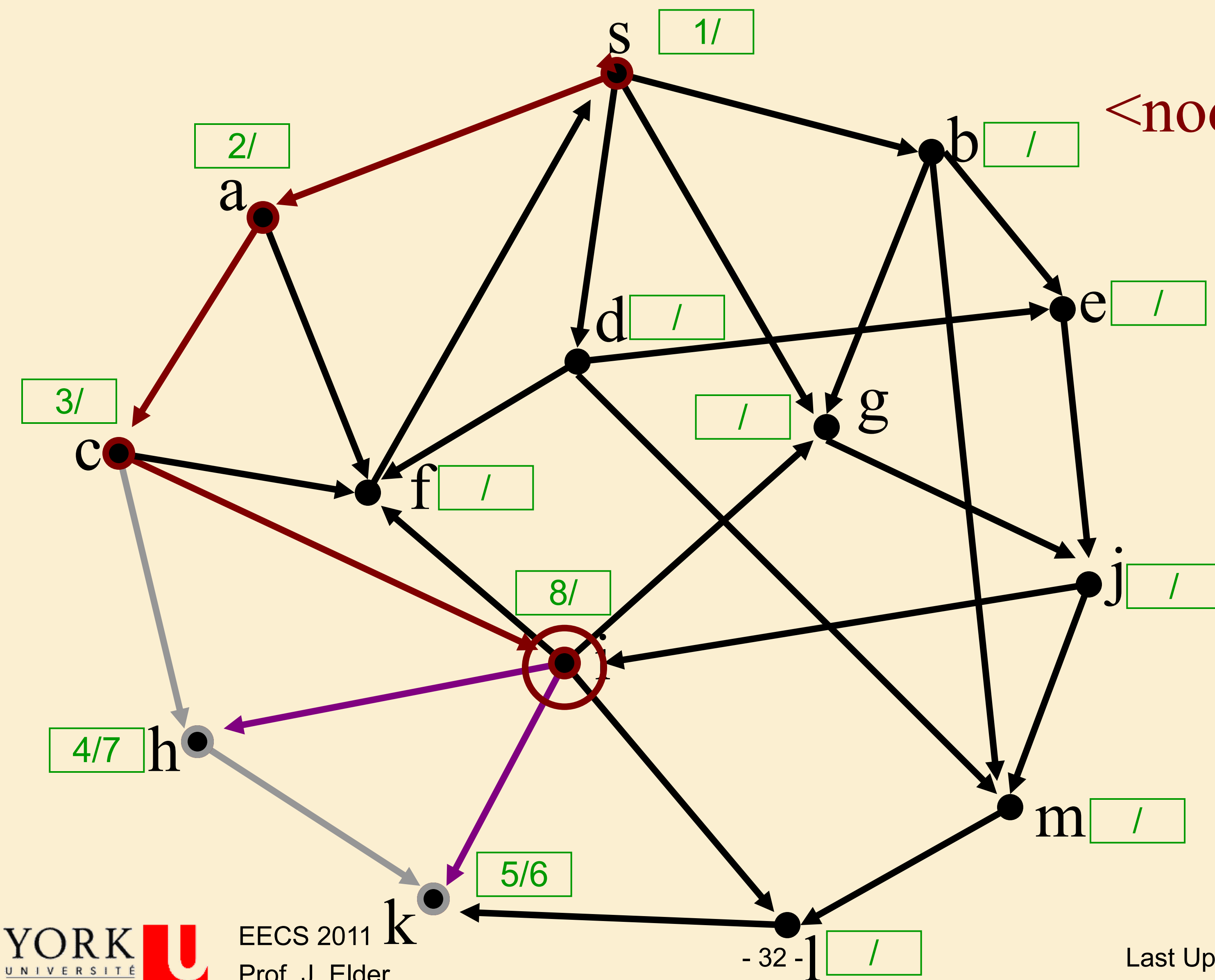
DFS



Discovered
Not Finished
Stack
<node,# edges>

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

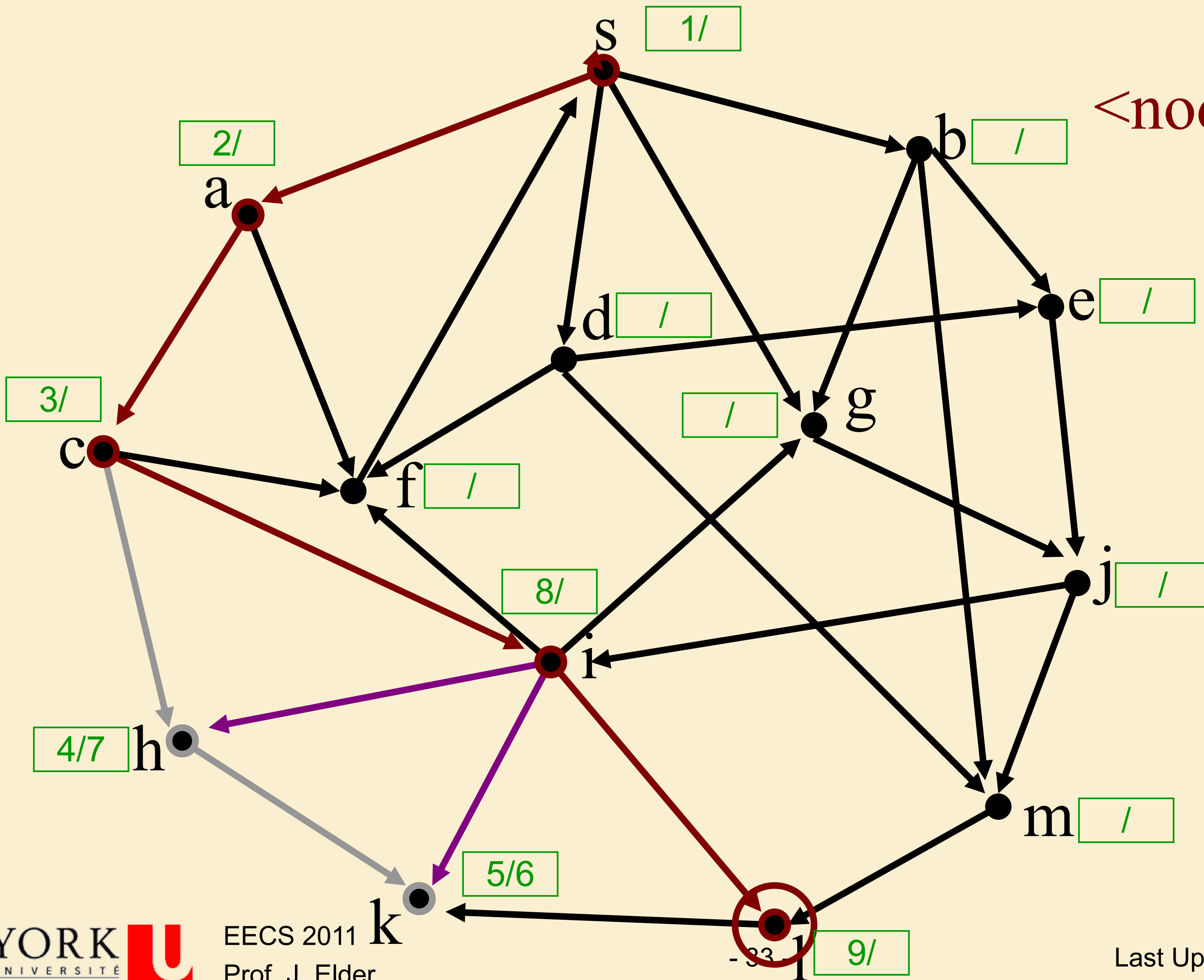
DFS



Discovered
Not Finished
Stack
<node,# edges>

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

DFS

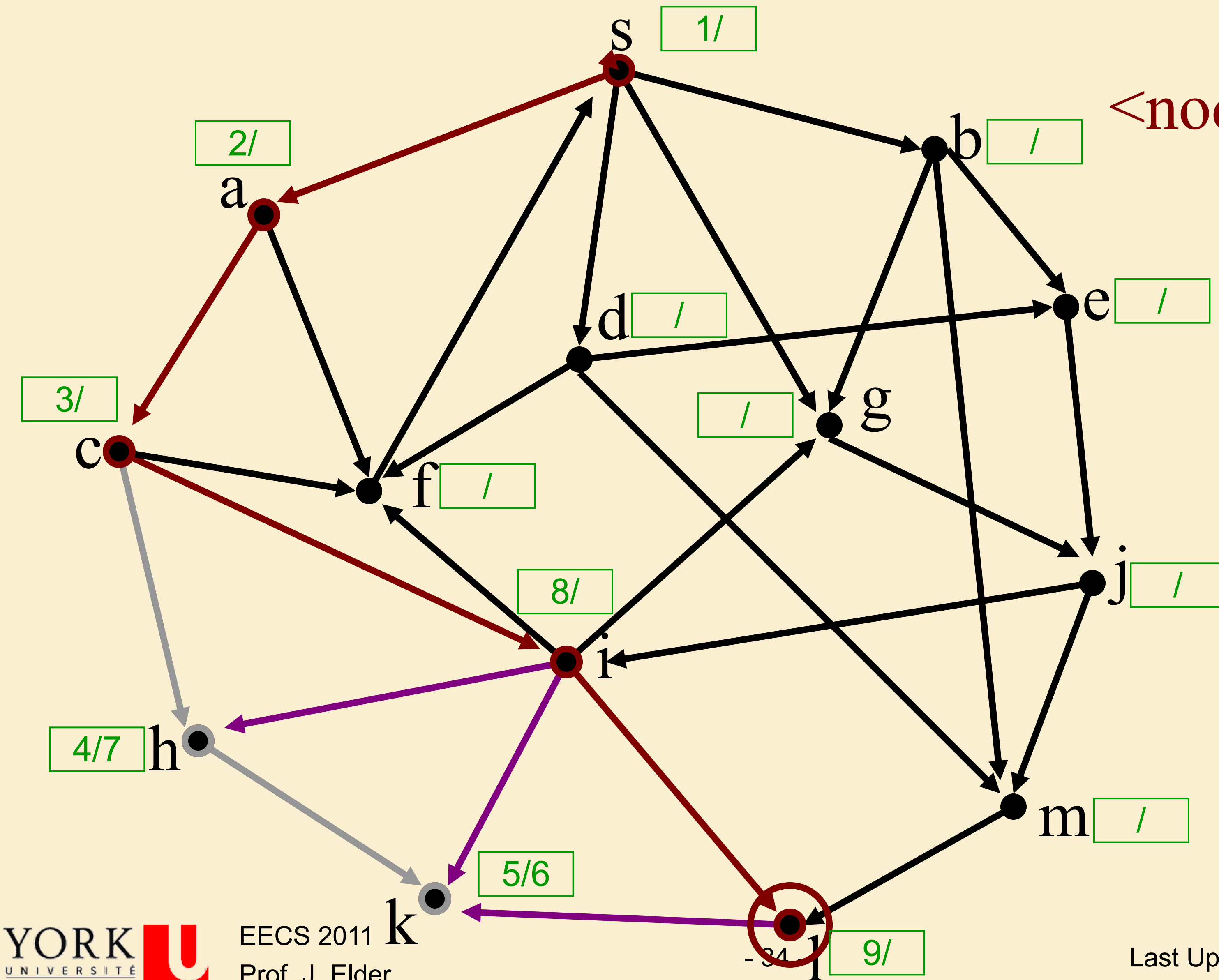


Discovered Not Finished Stack

<node,# edges>

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

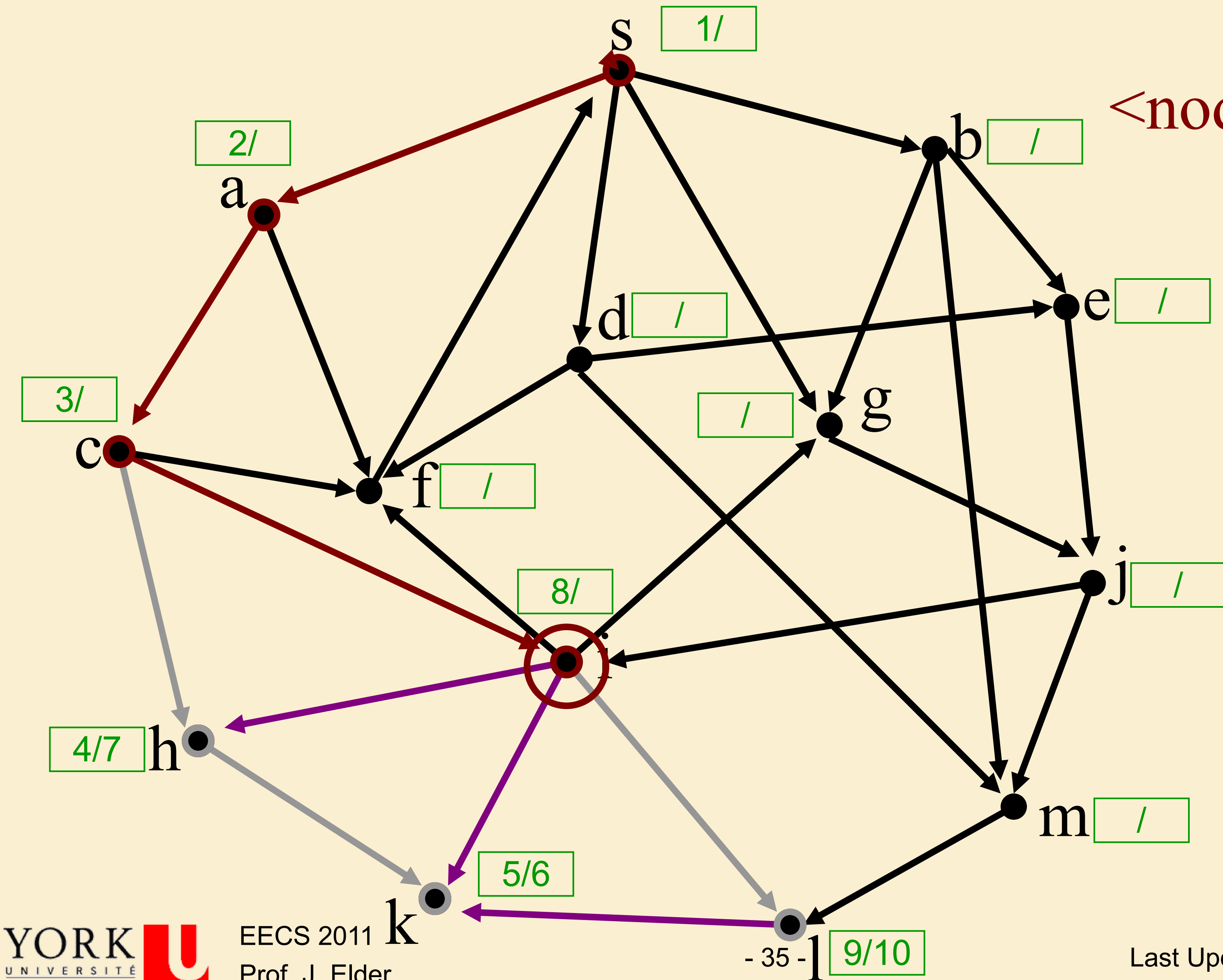
DFS



Discovered
Not Finished
Stack
<node,# edges>

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

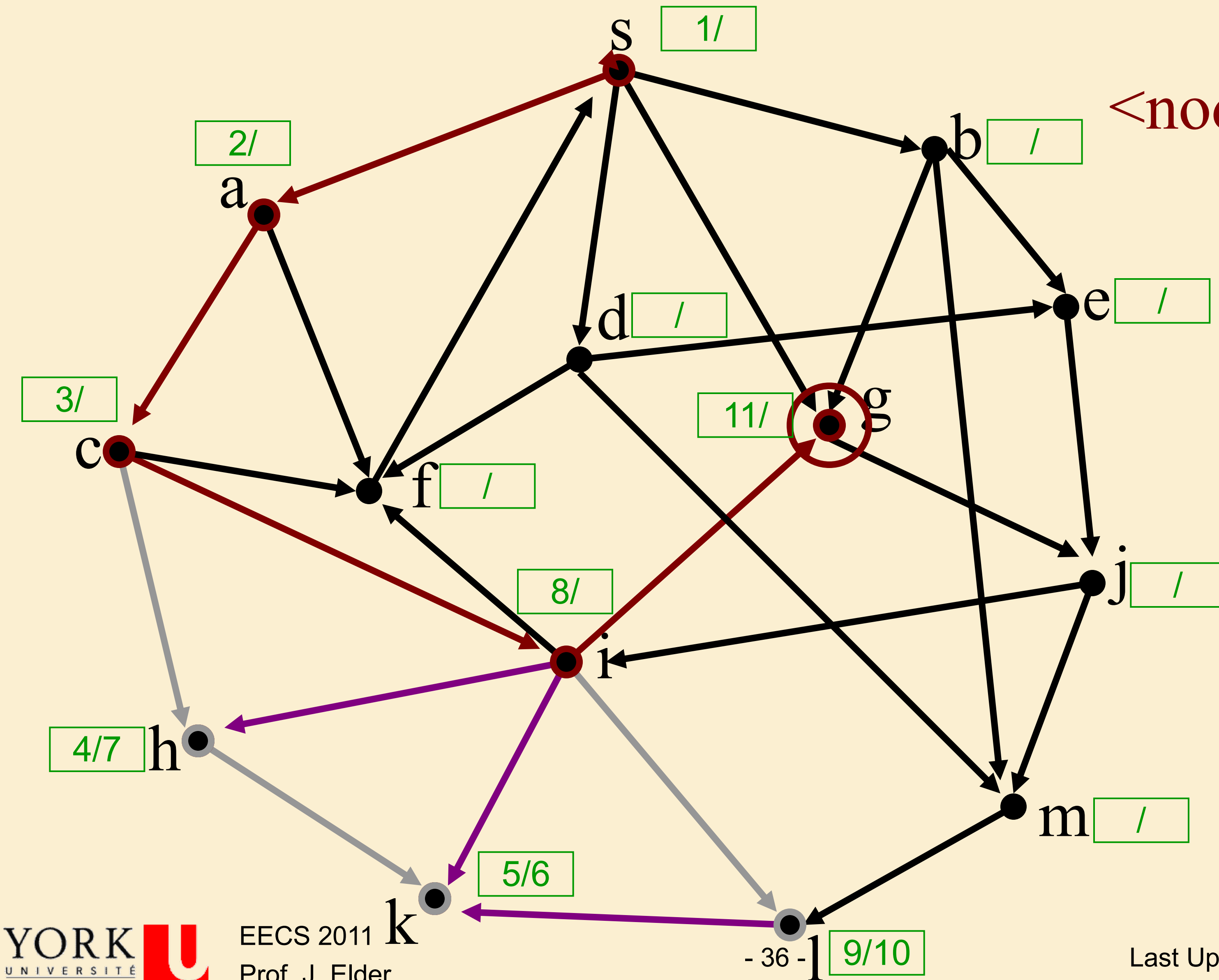
DFS



Discovered Not Finished Stack <node,# edges>

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

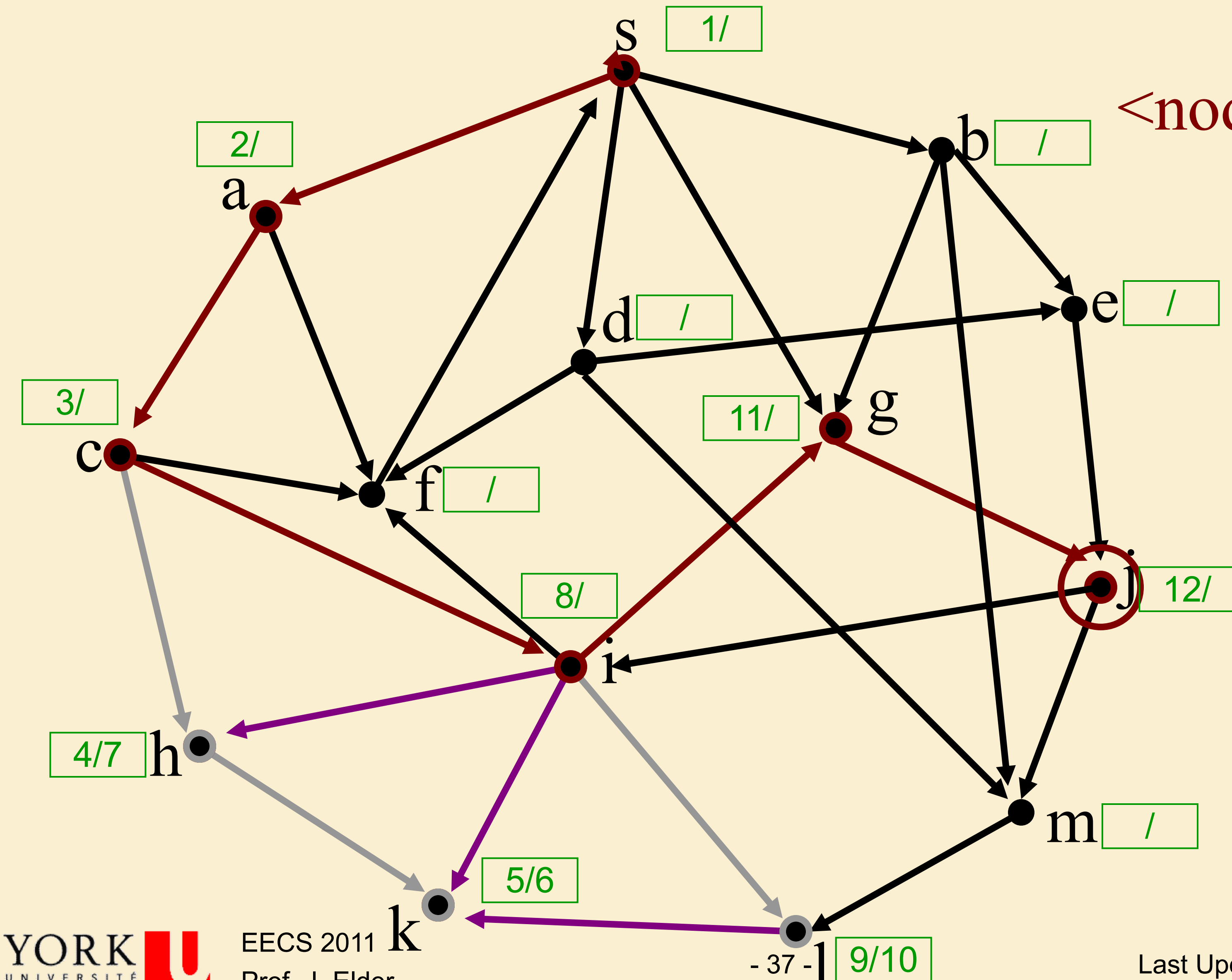
DFS



Discovered
Not Finished
Stack
<node,# edges>

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

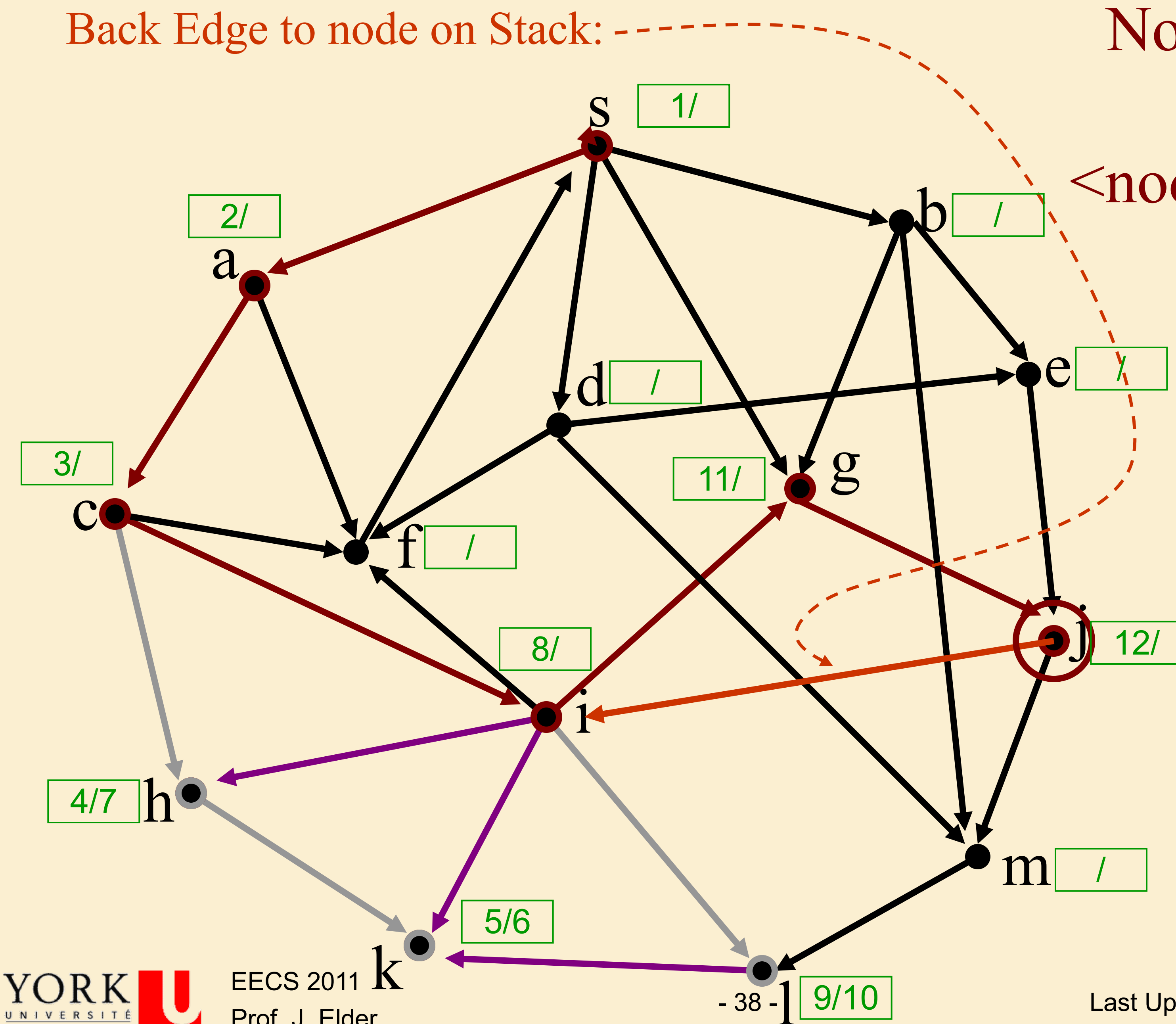
DFS



Discovered
Not Finished
Stack
<node,# edges>

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

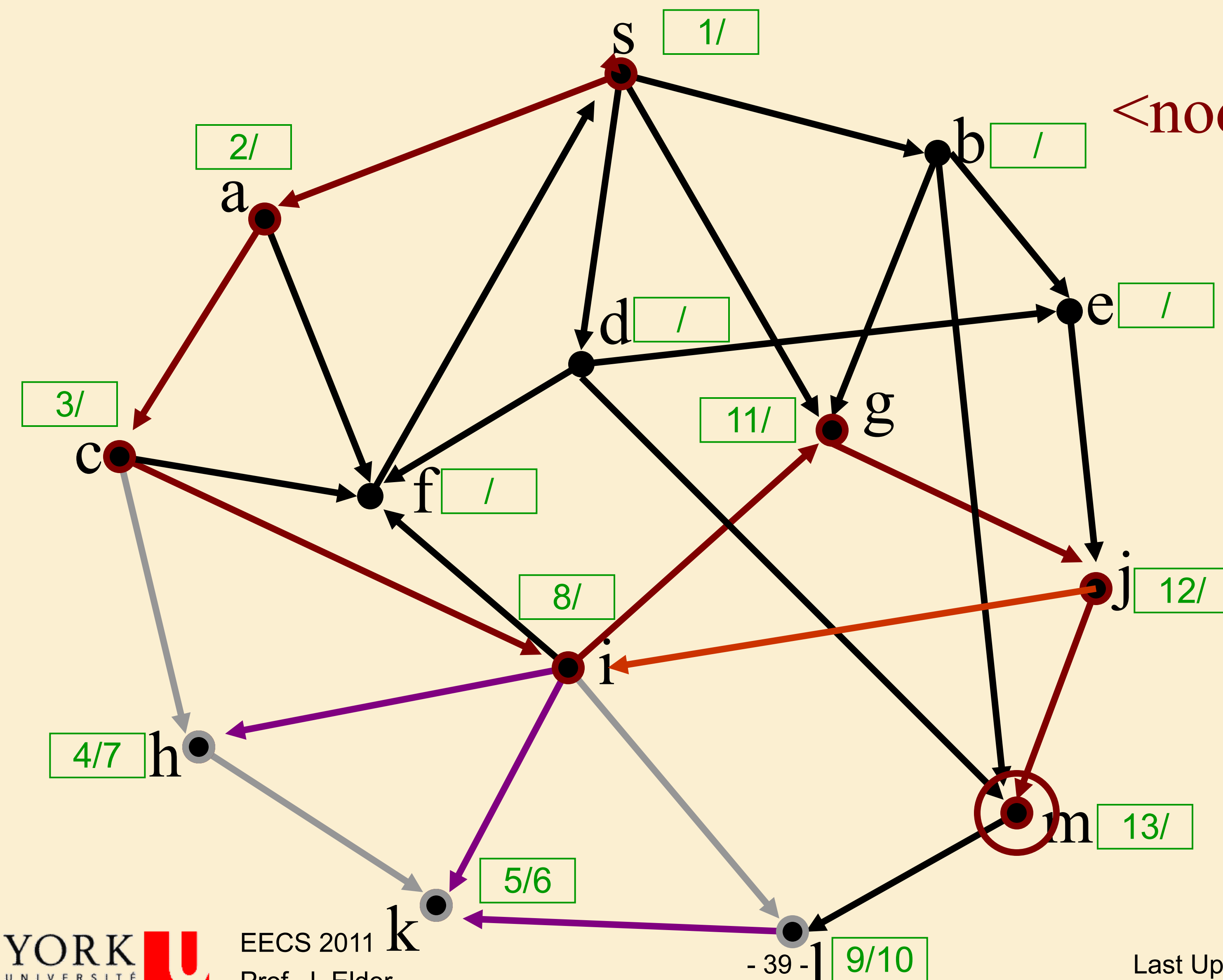
DFS



Discovered
Not Finished
Stack
<node,# edges>

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

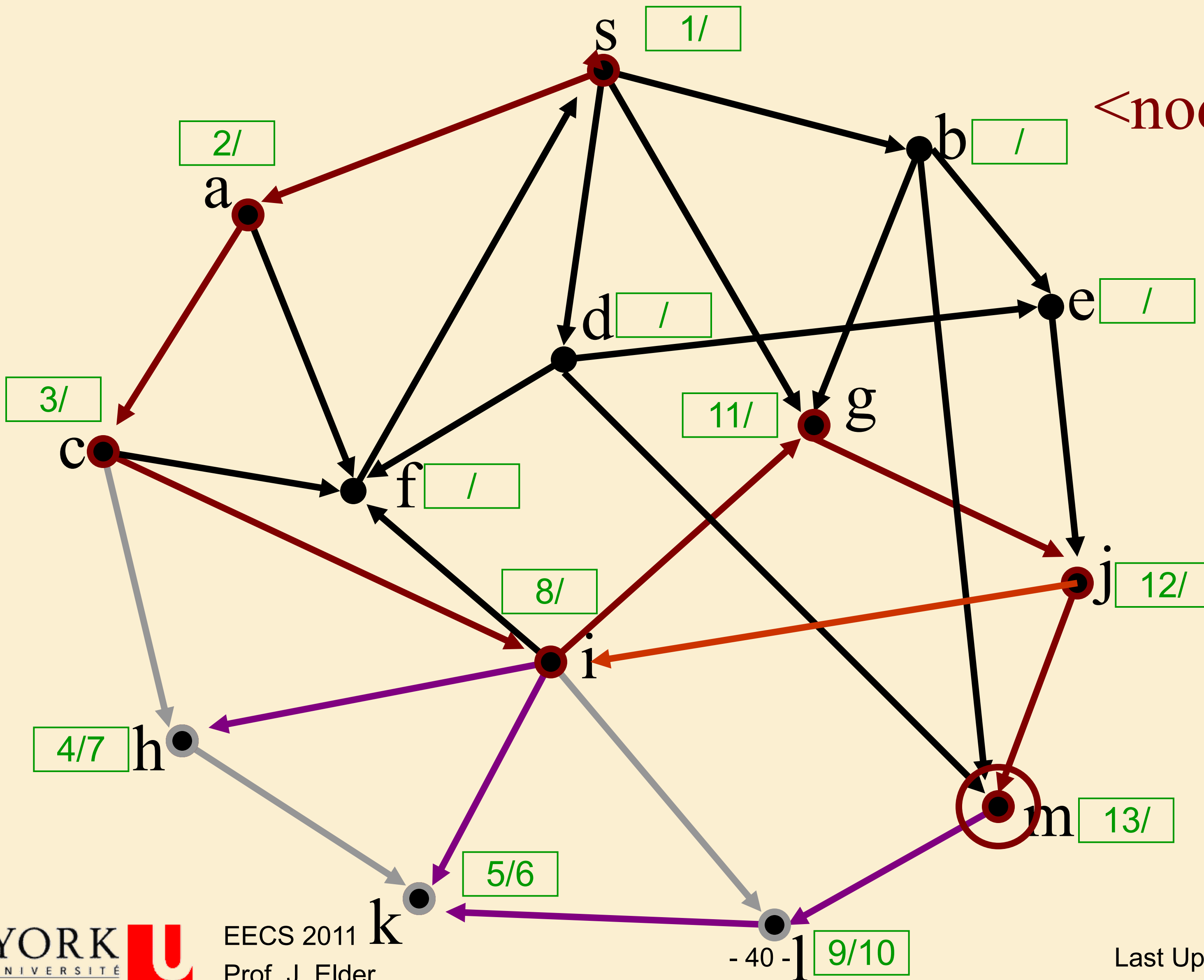
DFS



Discovered
Not Finished
Stack
<node,# edges>

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

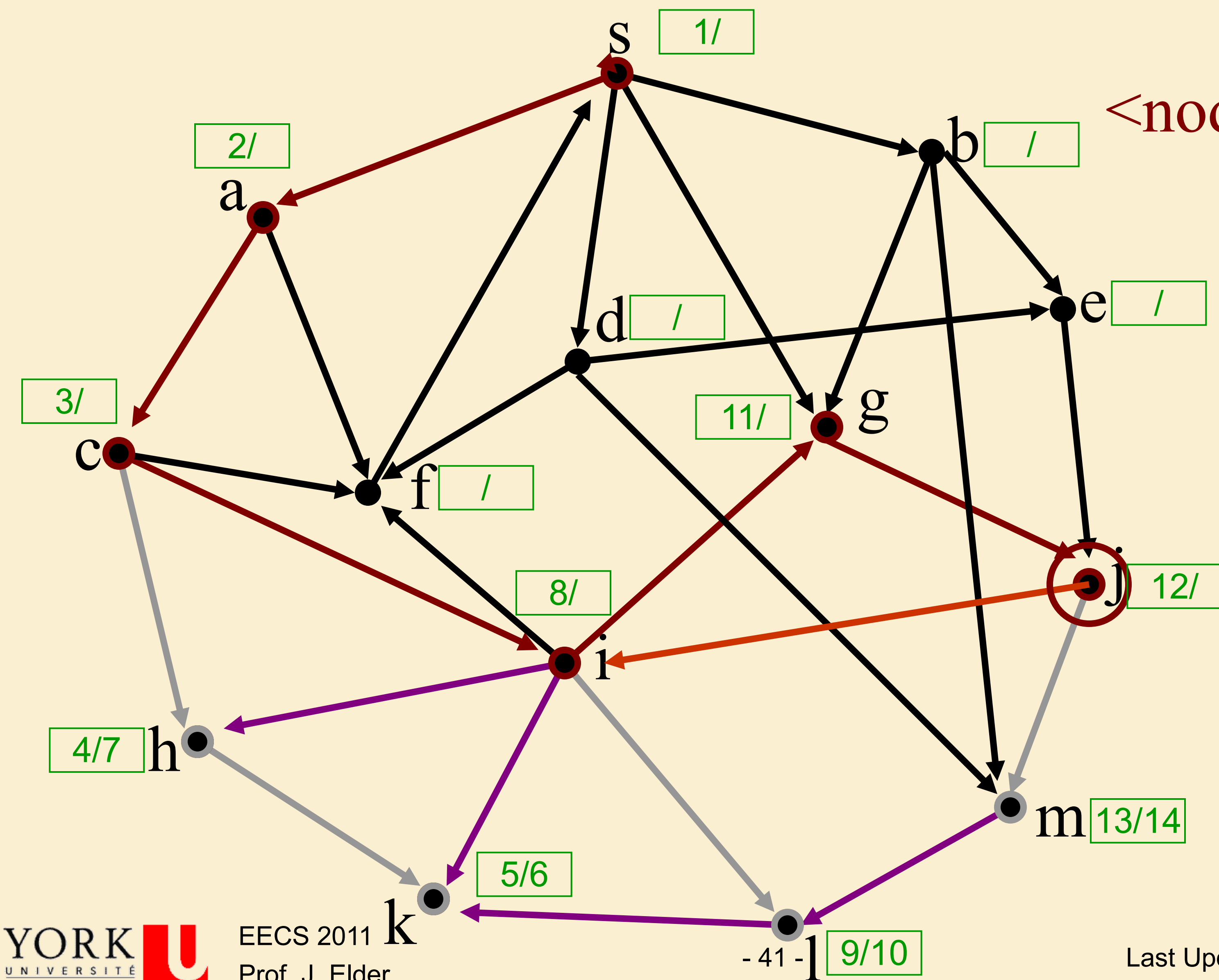
DFS



Discovered
Not Finished
Stack
<node,# edges>

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

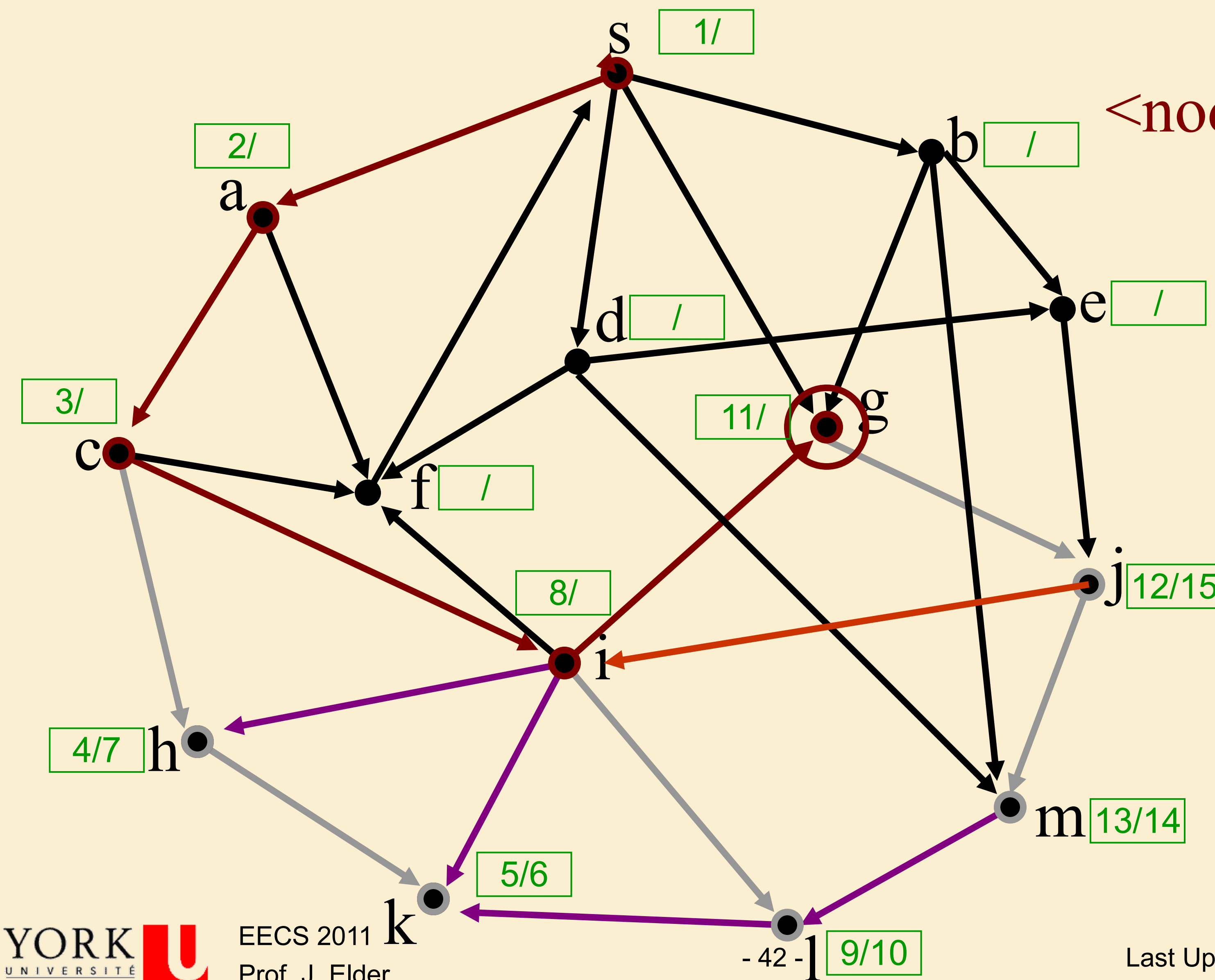
DFS



Discovered
Not Finished
Stack
<node,# edges>

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

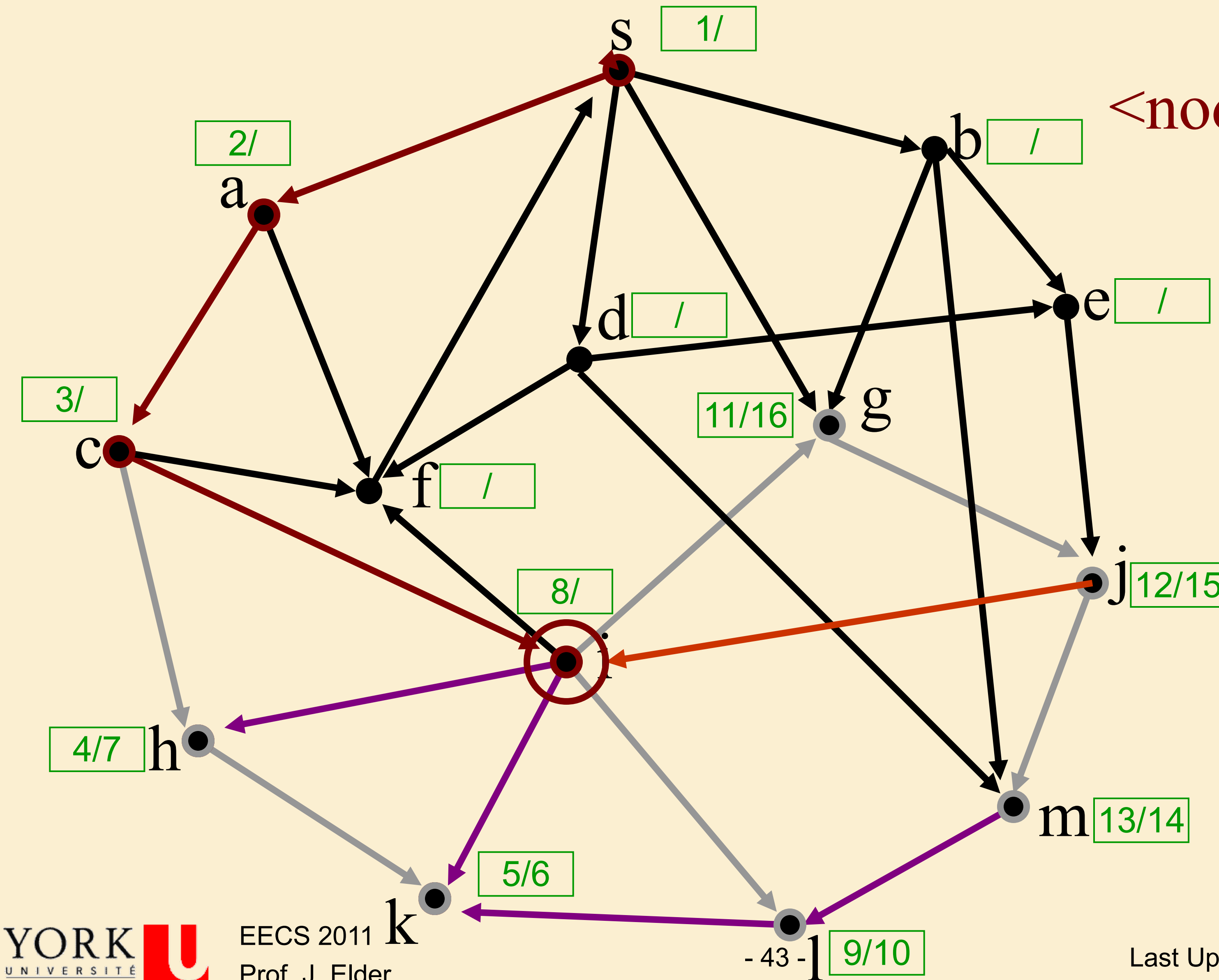
DFS



Discovered Not Finished Stack
<node,# edges>

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

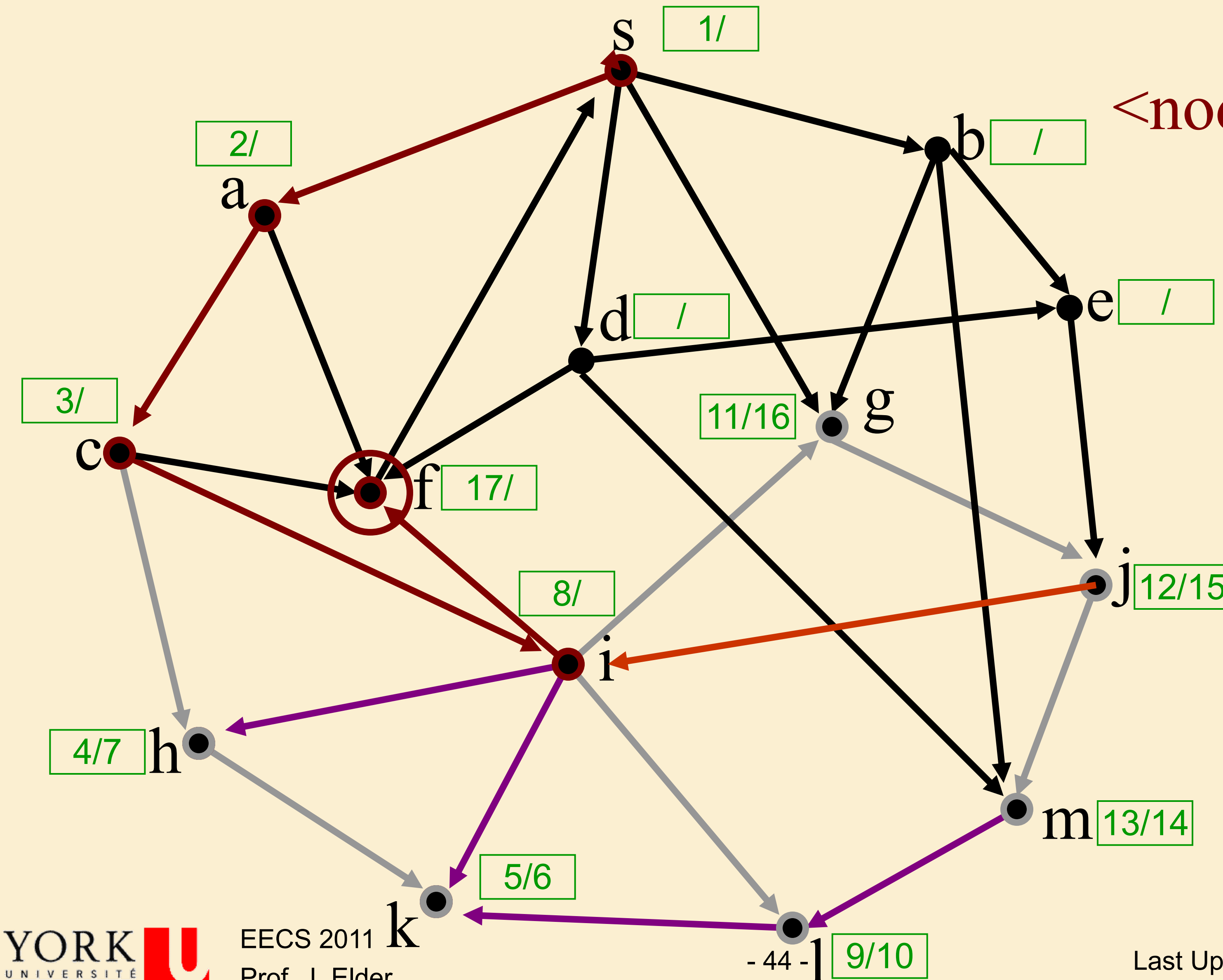
DFS



Discovered
Not Finished
Stack
<node,# edges>

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

DFS

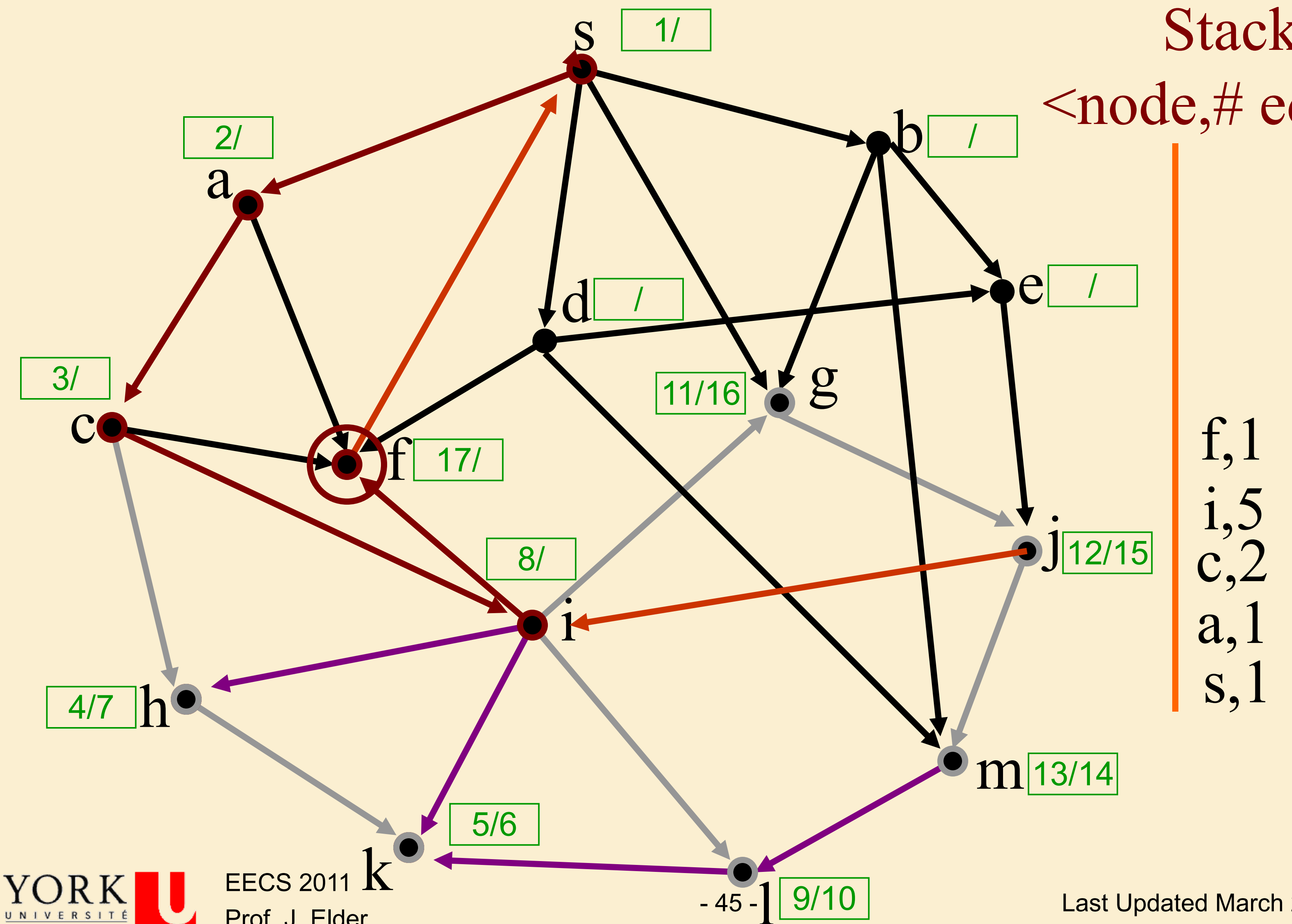


Discovered
Not Finished
Stack
<node,# edges>

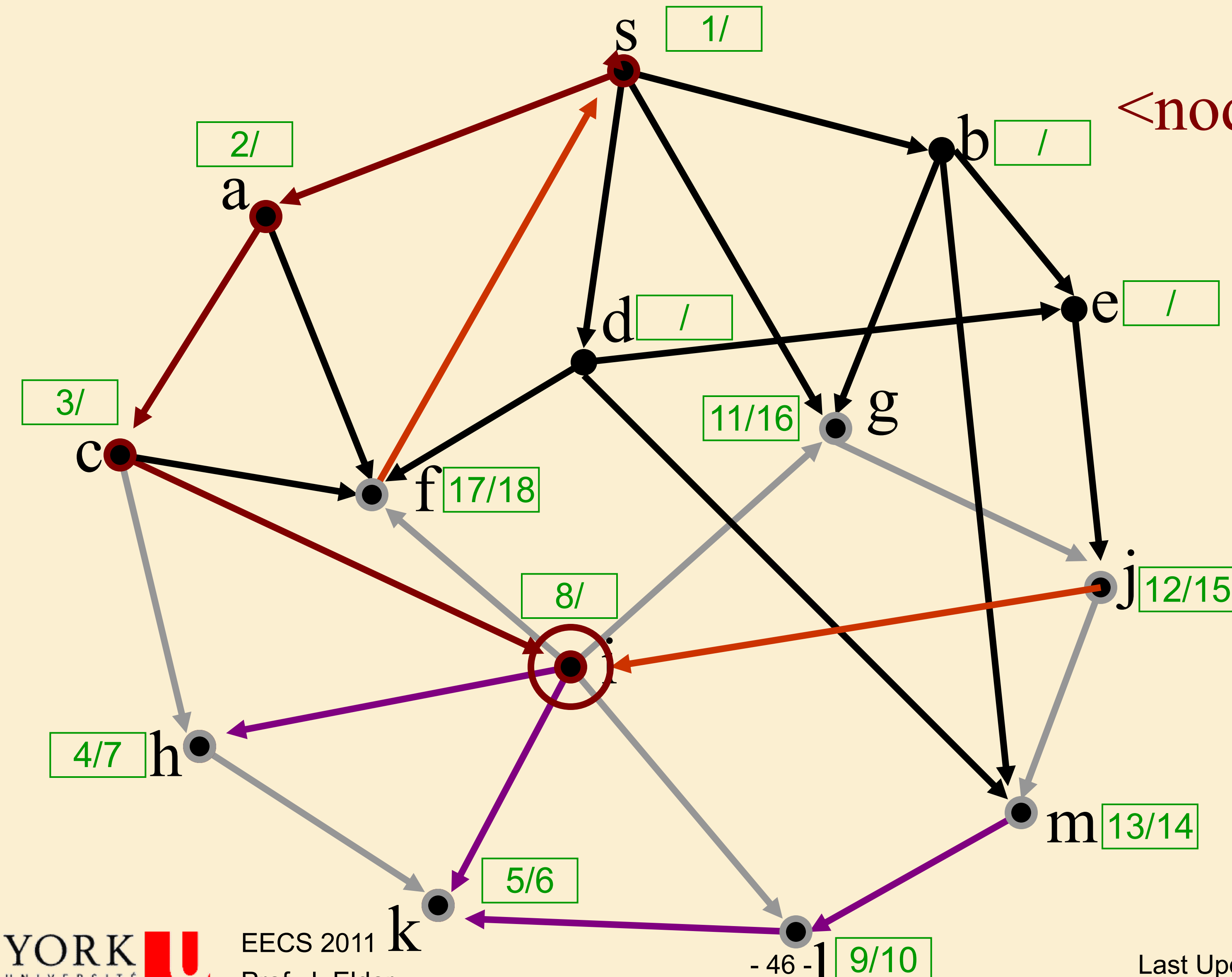
f,0
i,5
c,2
a,1
s,1

DFS

Discovered
Not Finished
Stack
<node,# edges>



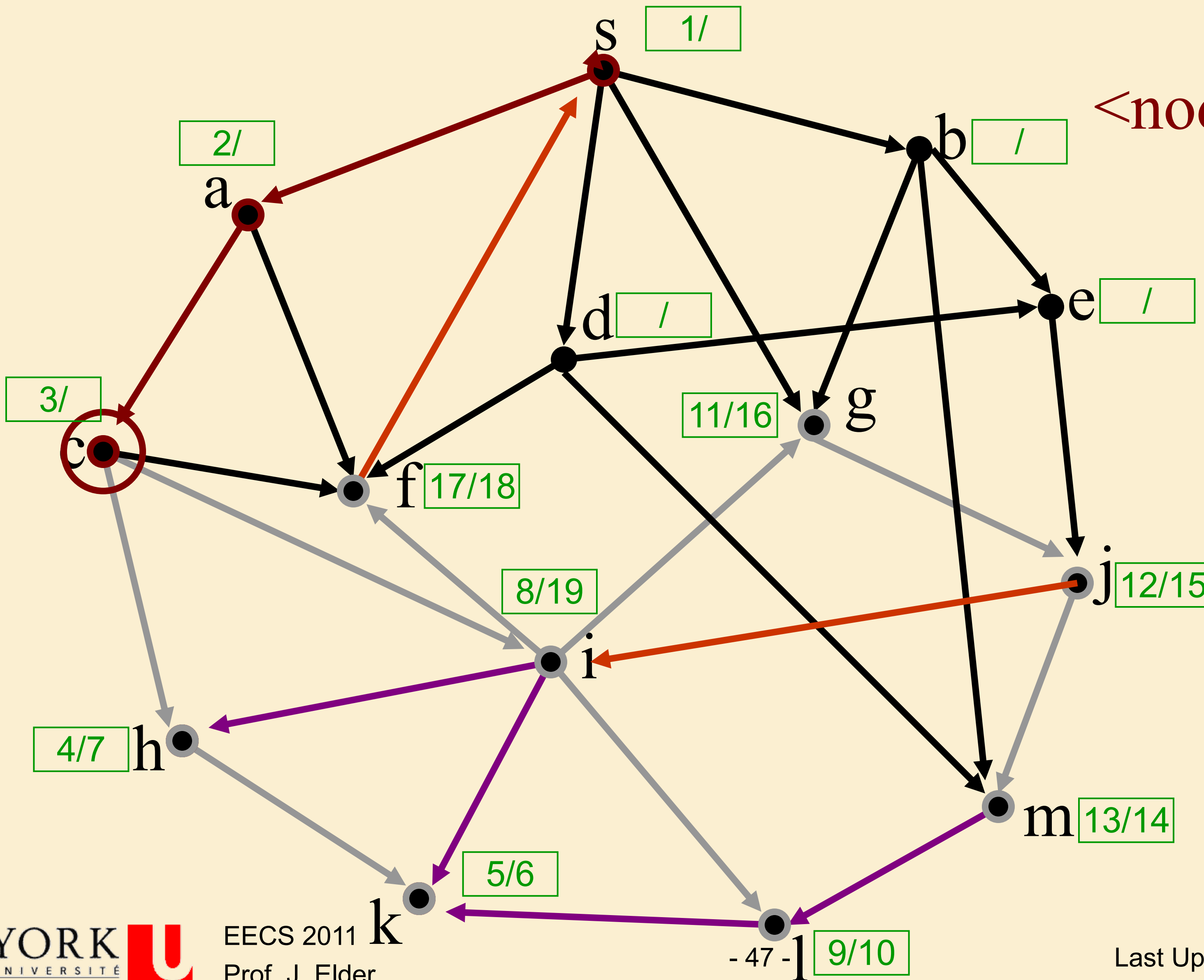
DFS



Discovered
Not Finished
Stack
<node,# edges>

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

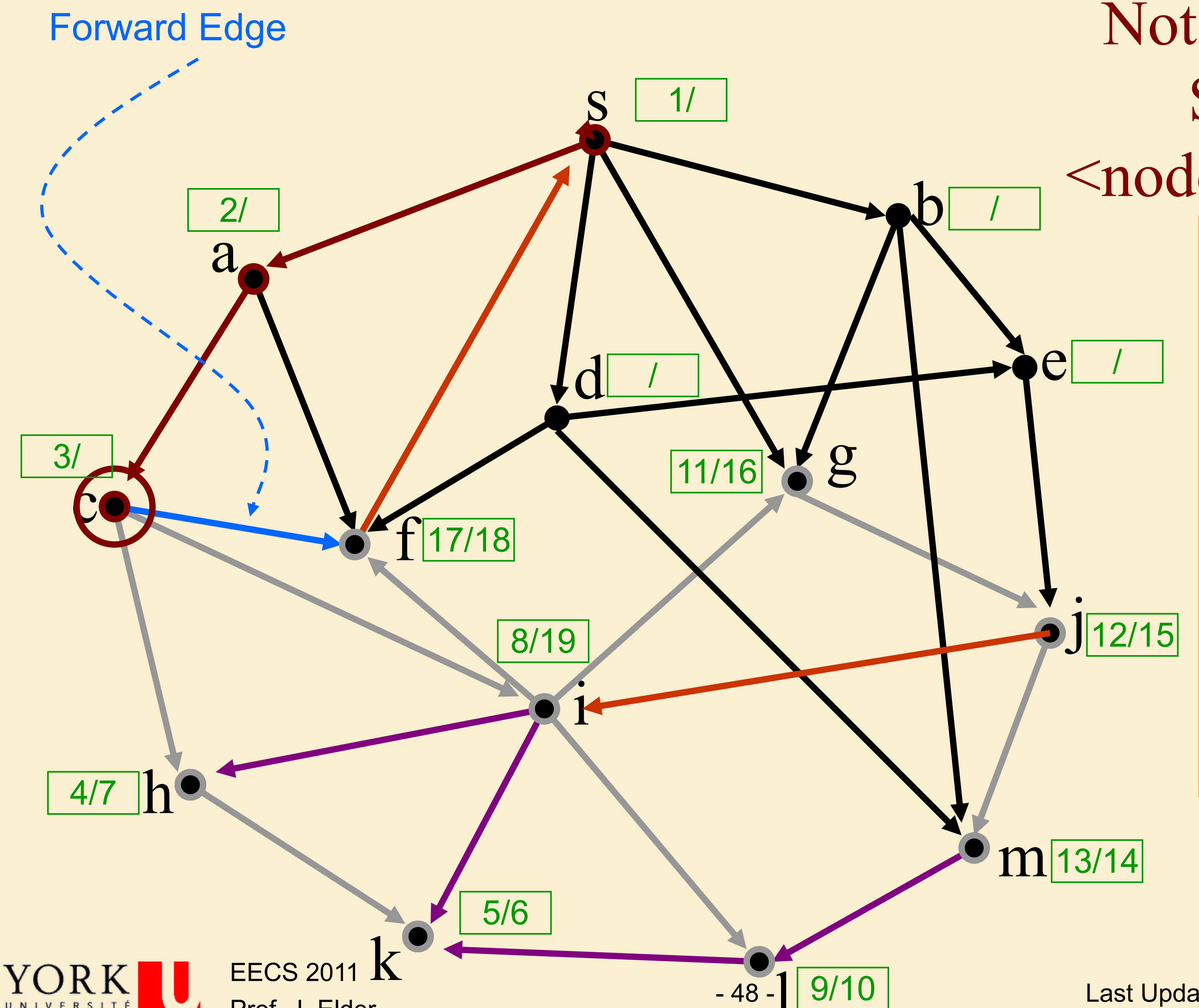
DFS



Discovered
Not Finished
Stack
<node,# edges>

c,2
a,1
s,1

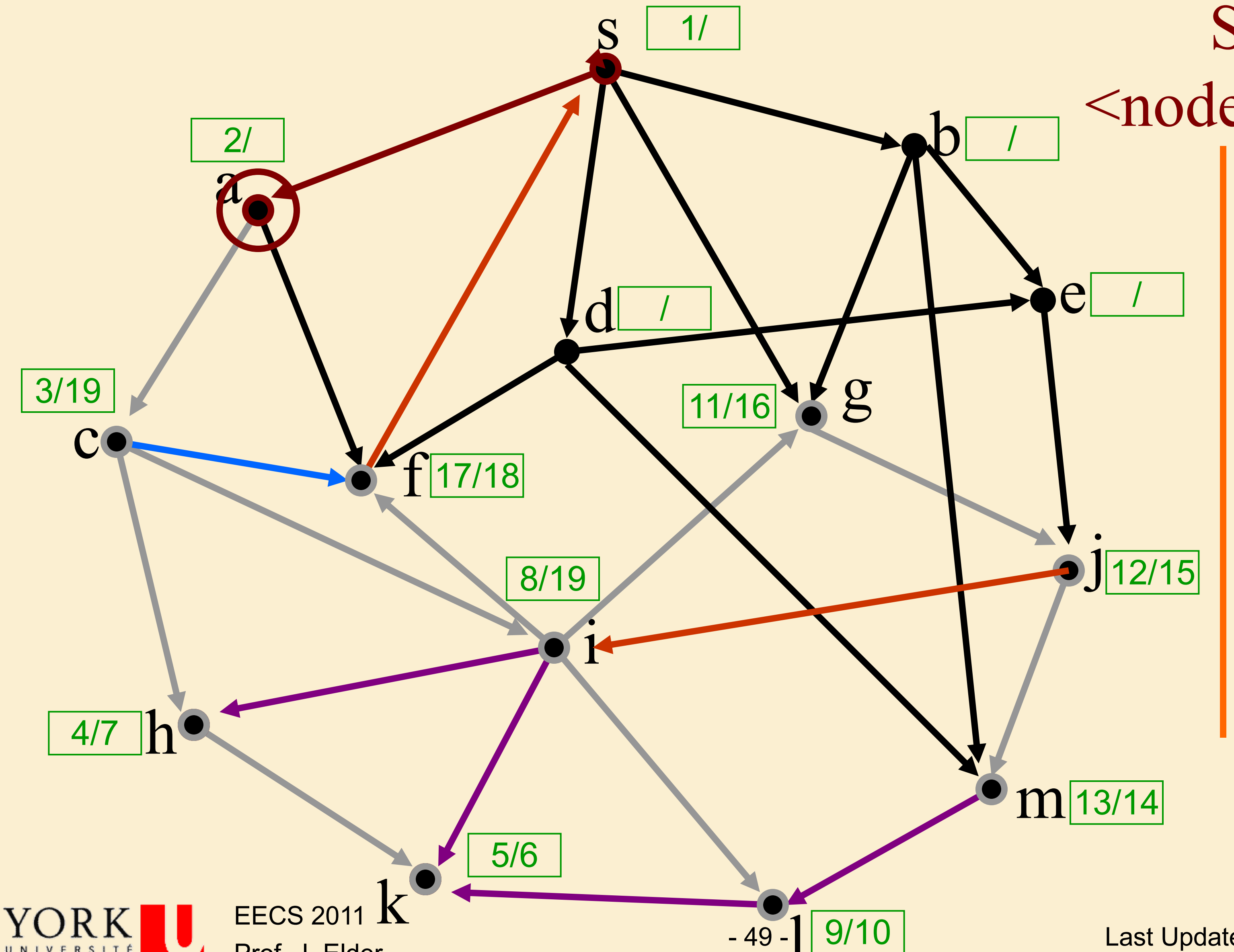
DFS



Discovered
Not Finished
Stack
<node,# edges>

c,3
a,1
s,1

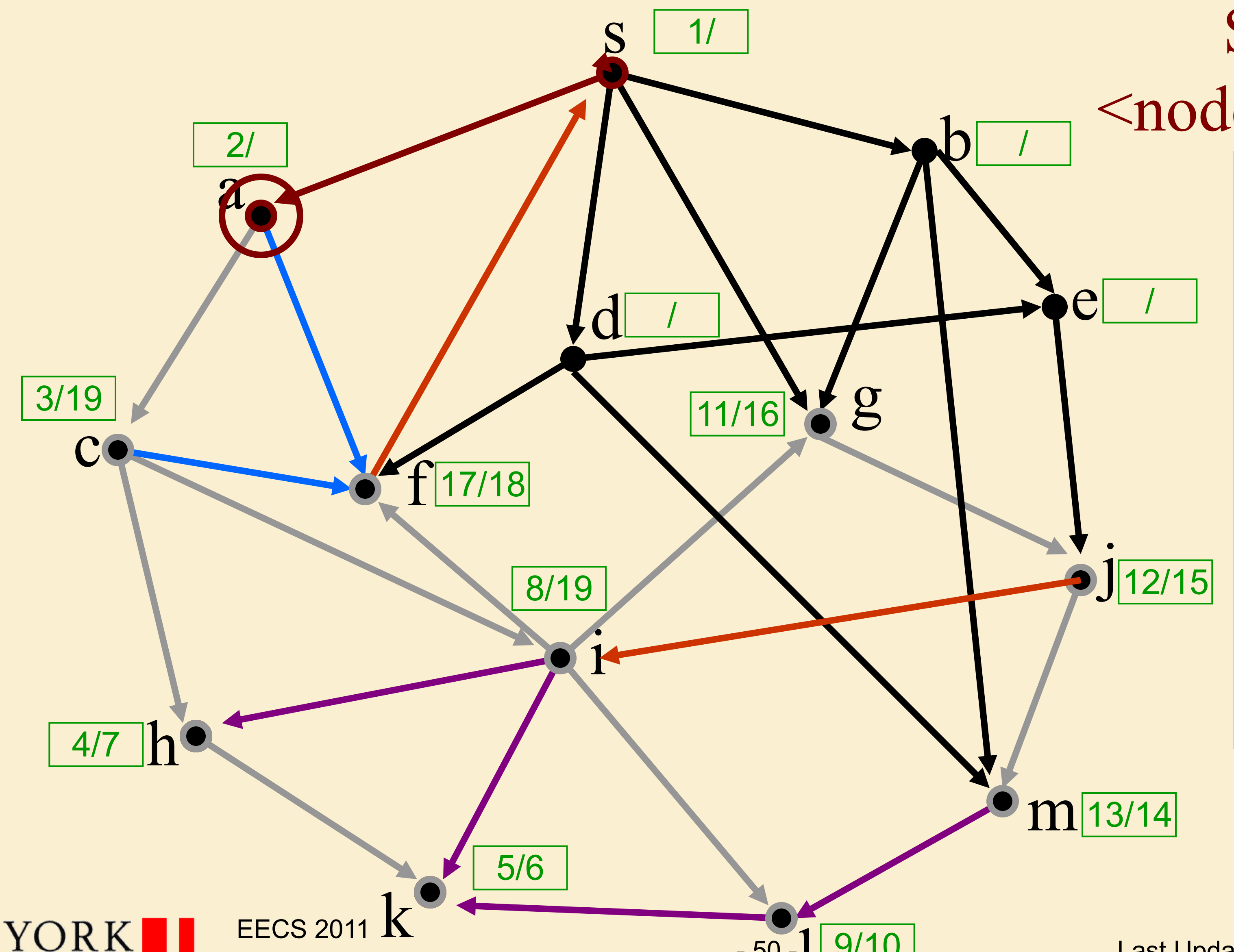
DFS



Discovered
Not Finished
Stack
<node,# edges>

a,1
s,1

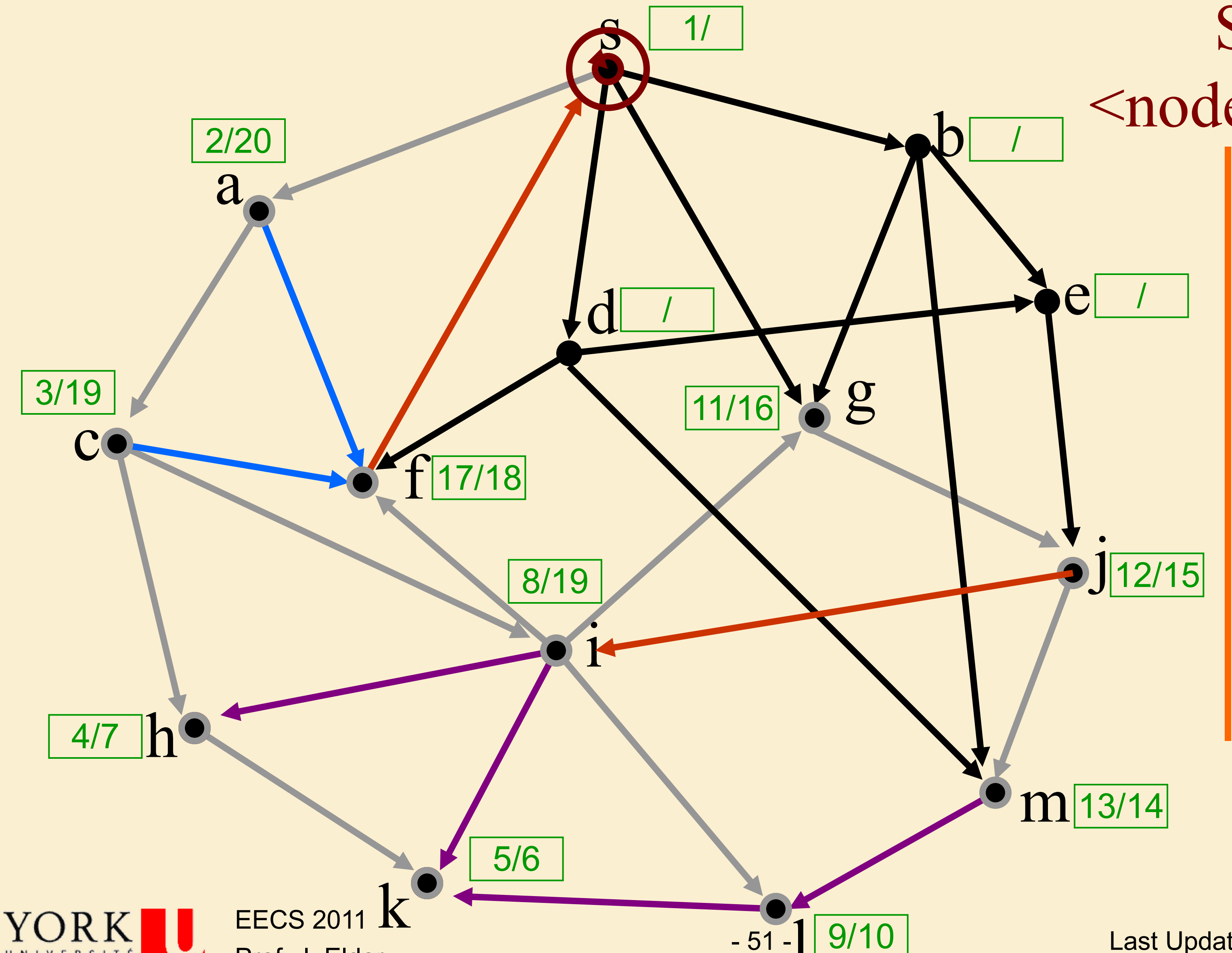
DFS



Discovered
Not Finished
Stack
<node,# edges>

a,2
s,1

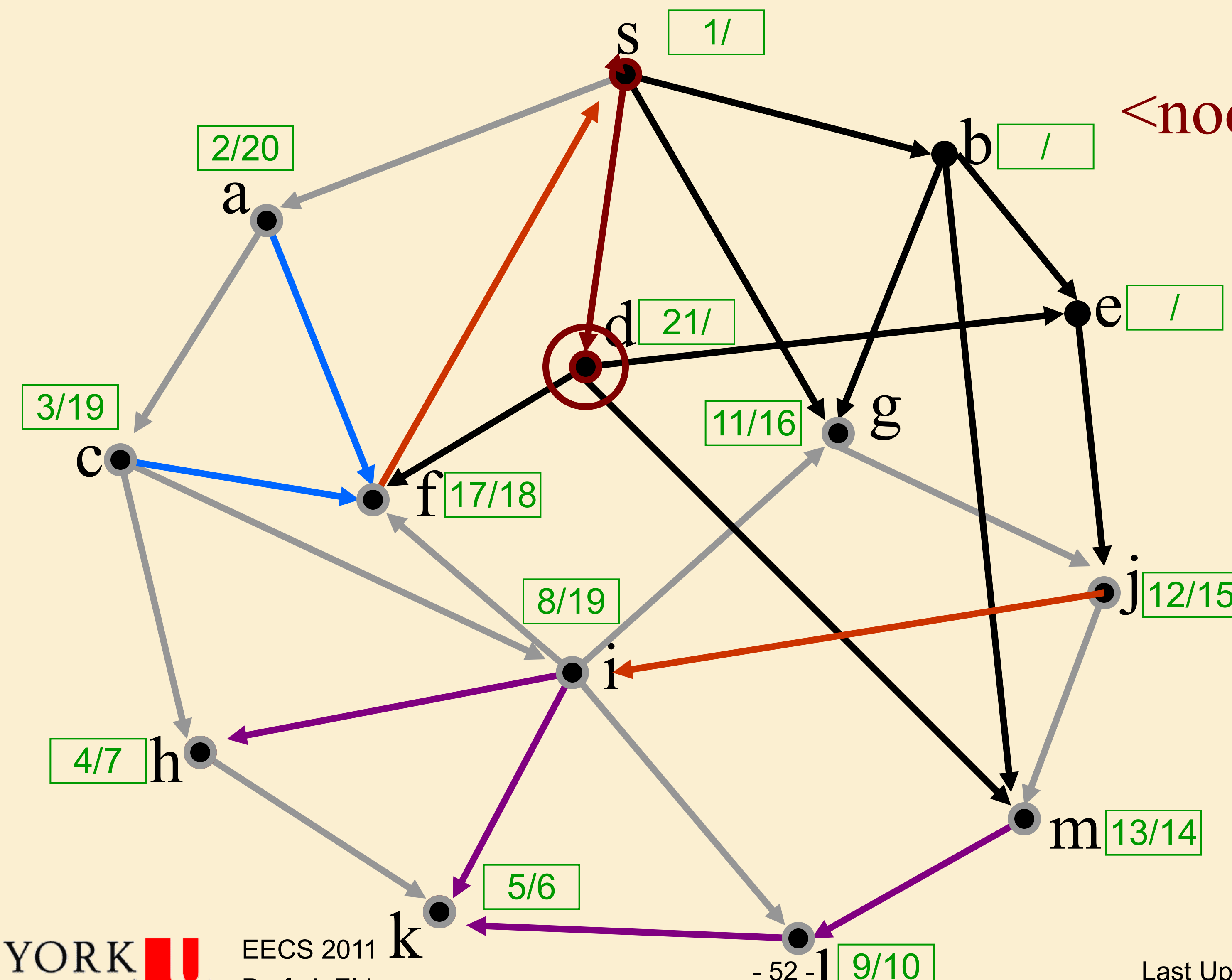
DFS



Discovered
Not Finished
Stack
<node,# edges>

s, 1

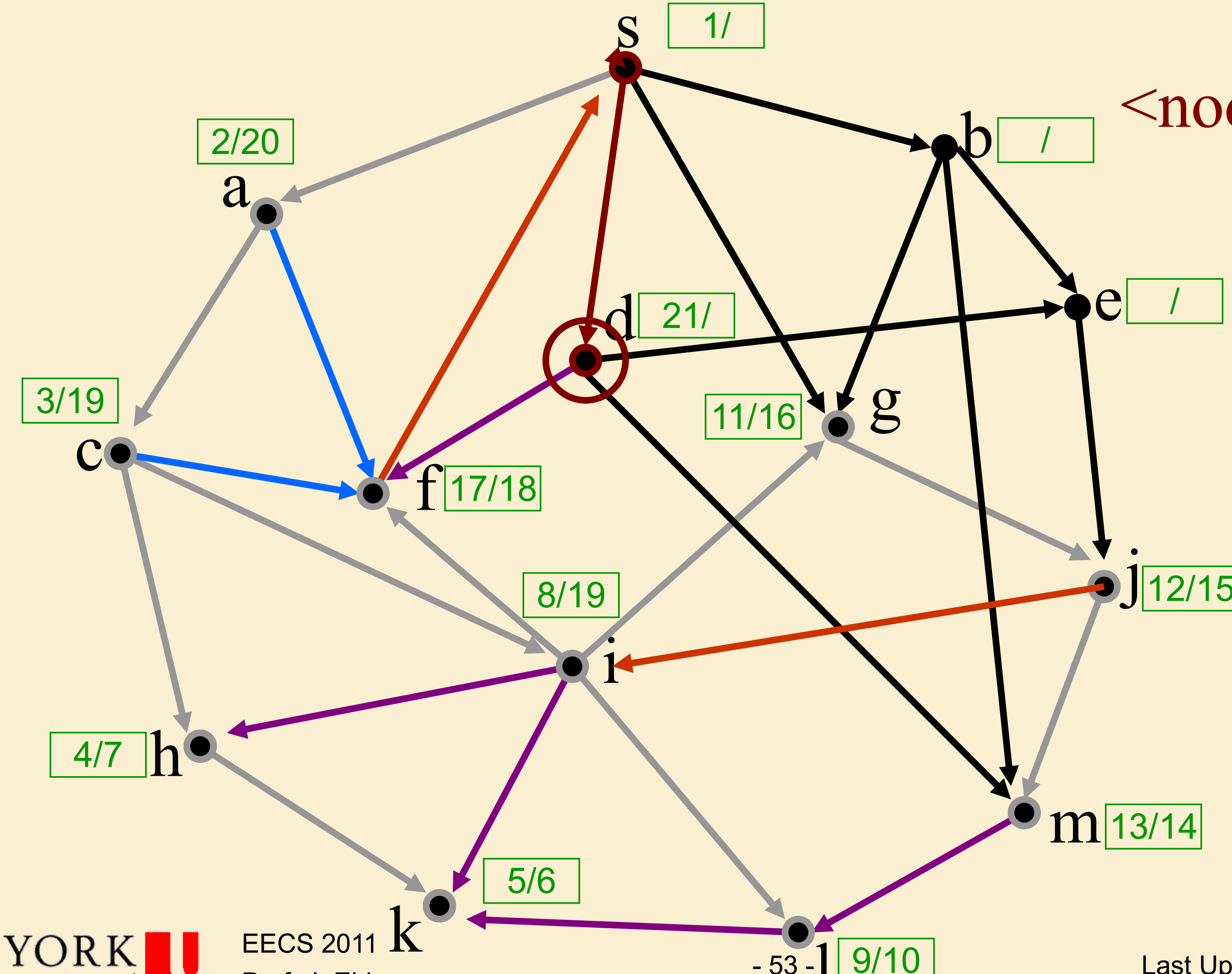
DFS



Discovered
Not Finished
Stack
<node,# edges>

d,0
s,2

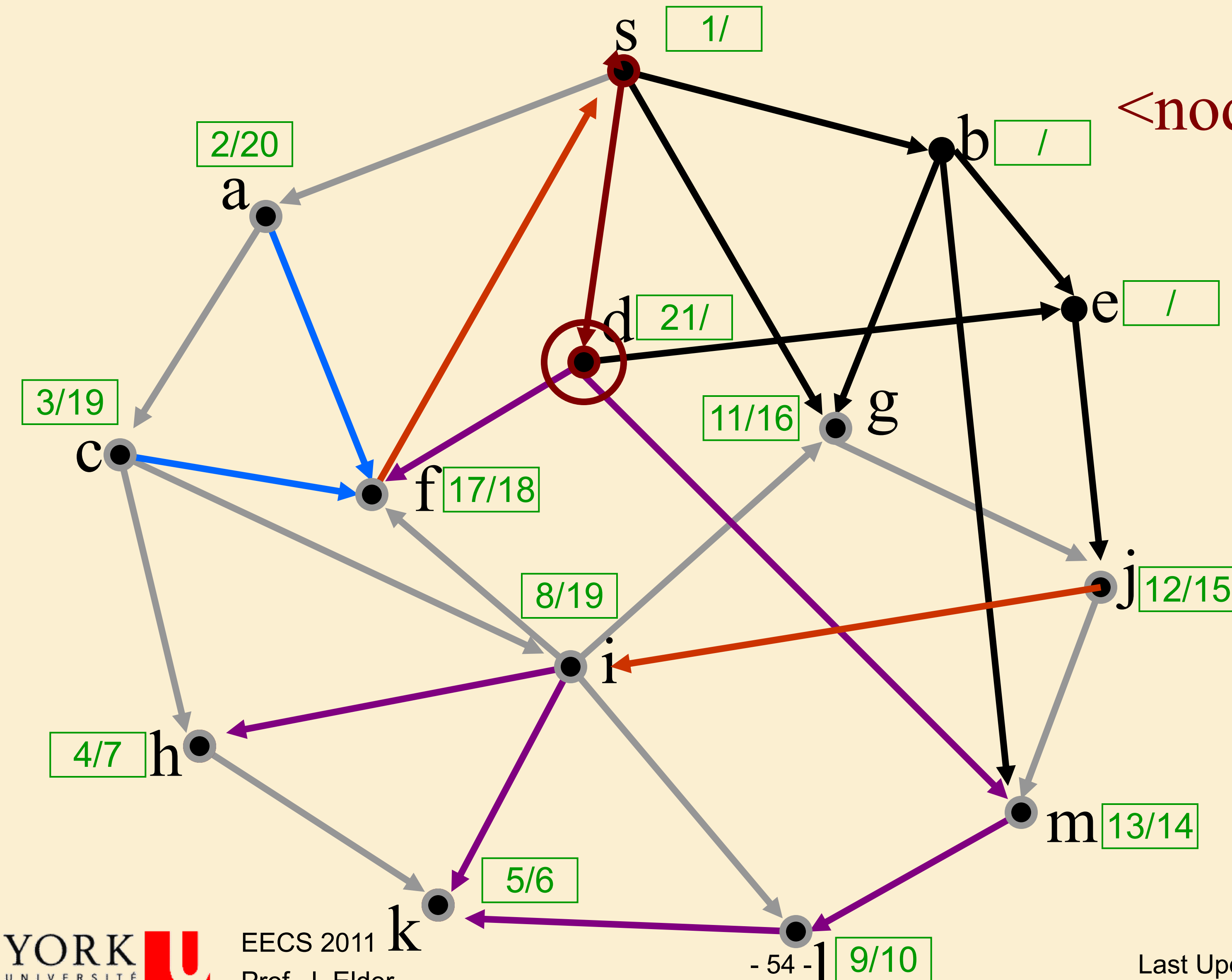
DFS



Discovered Not Finished Stack <node,# edges>

d,1
s,2

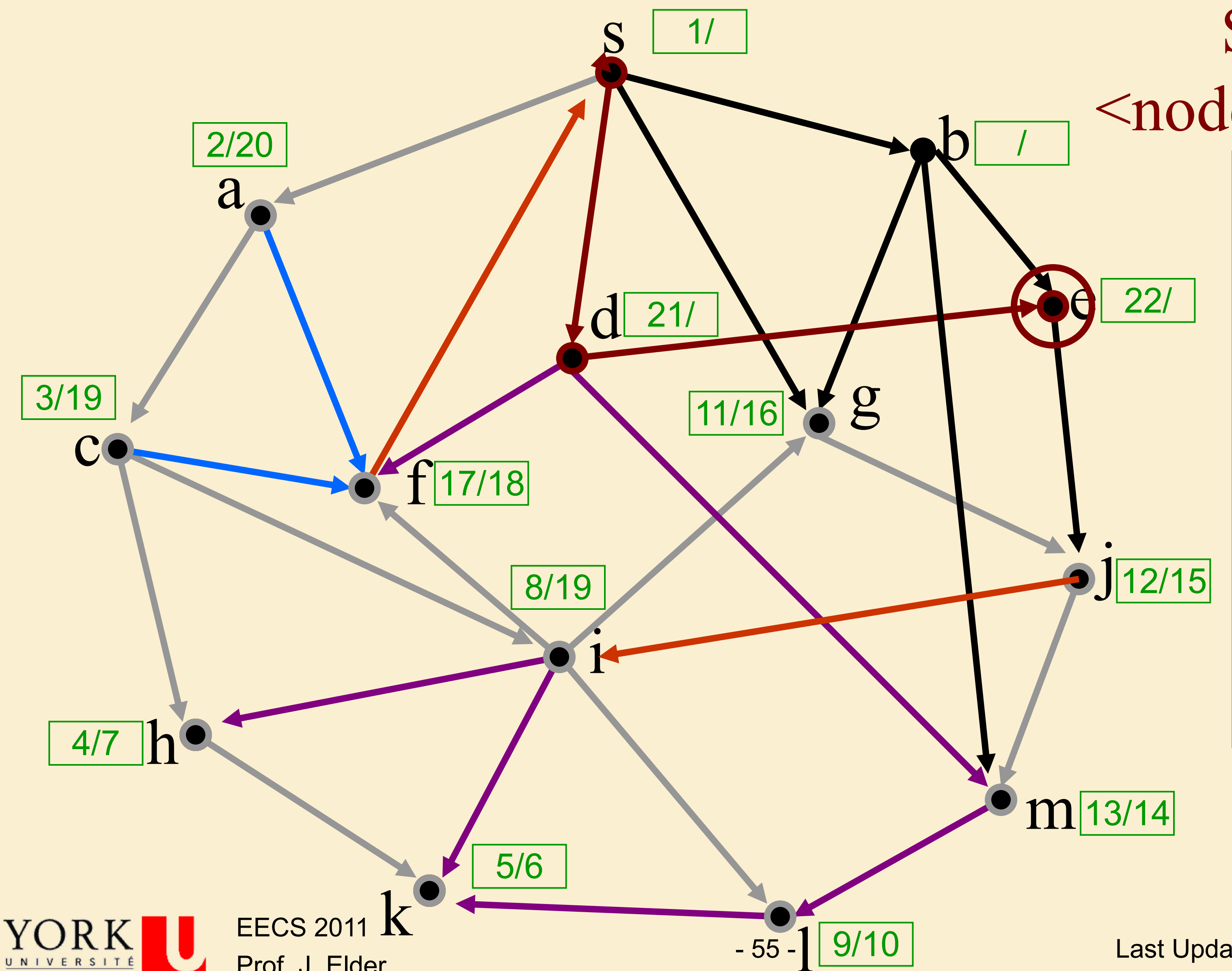
DFS



Discovered
Not Finished
Stack
<node,# edges>

d,2
s,2

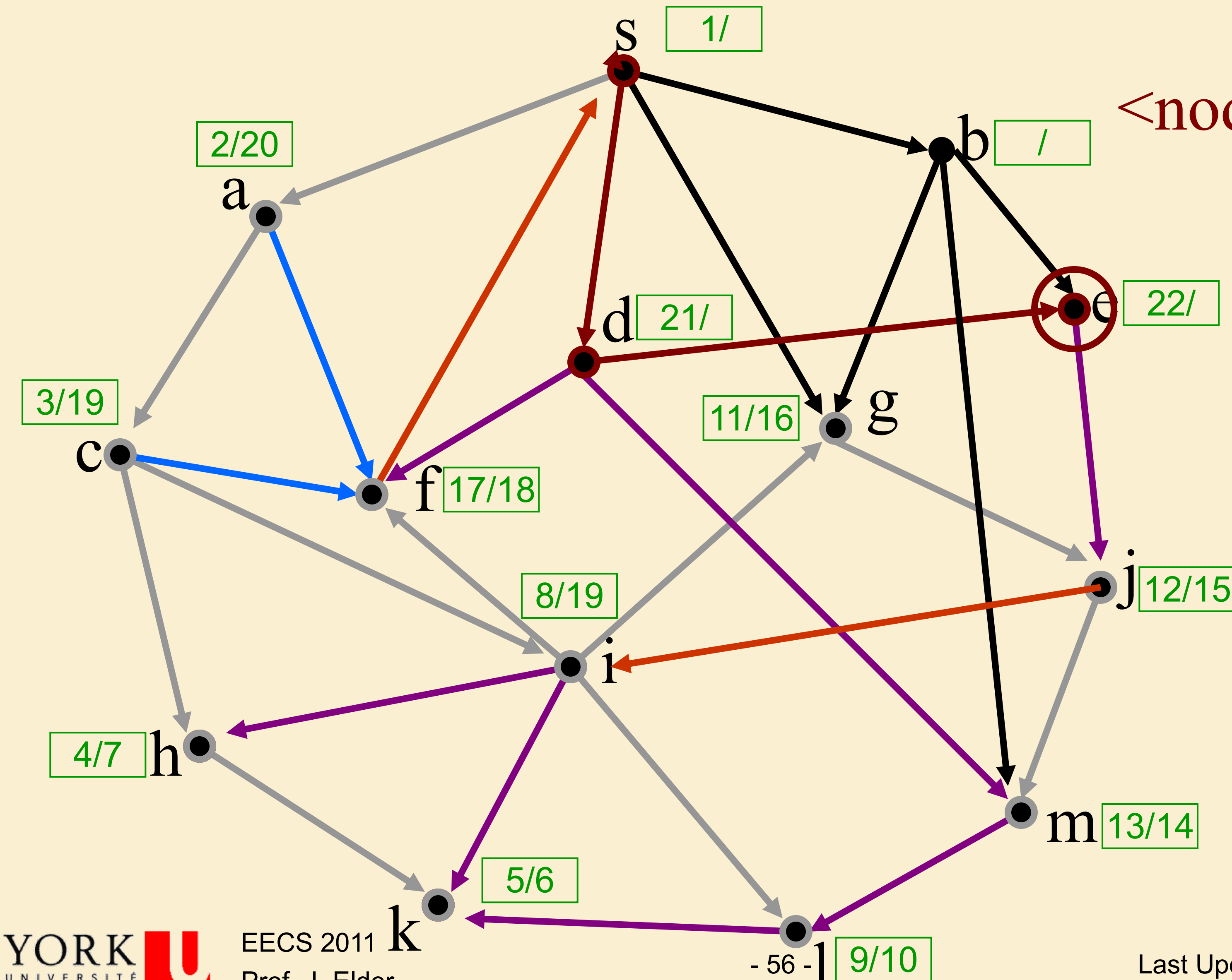
DFS



Discovered
Not Finished
Stack
<node,# edges>

e,0
d,3
s,2

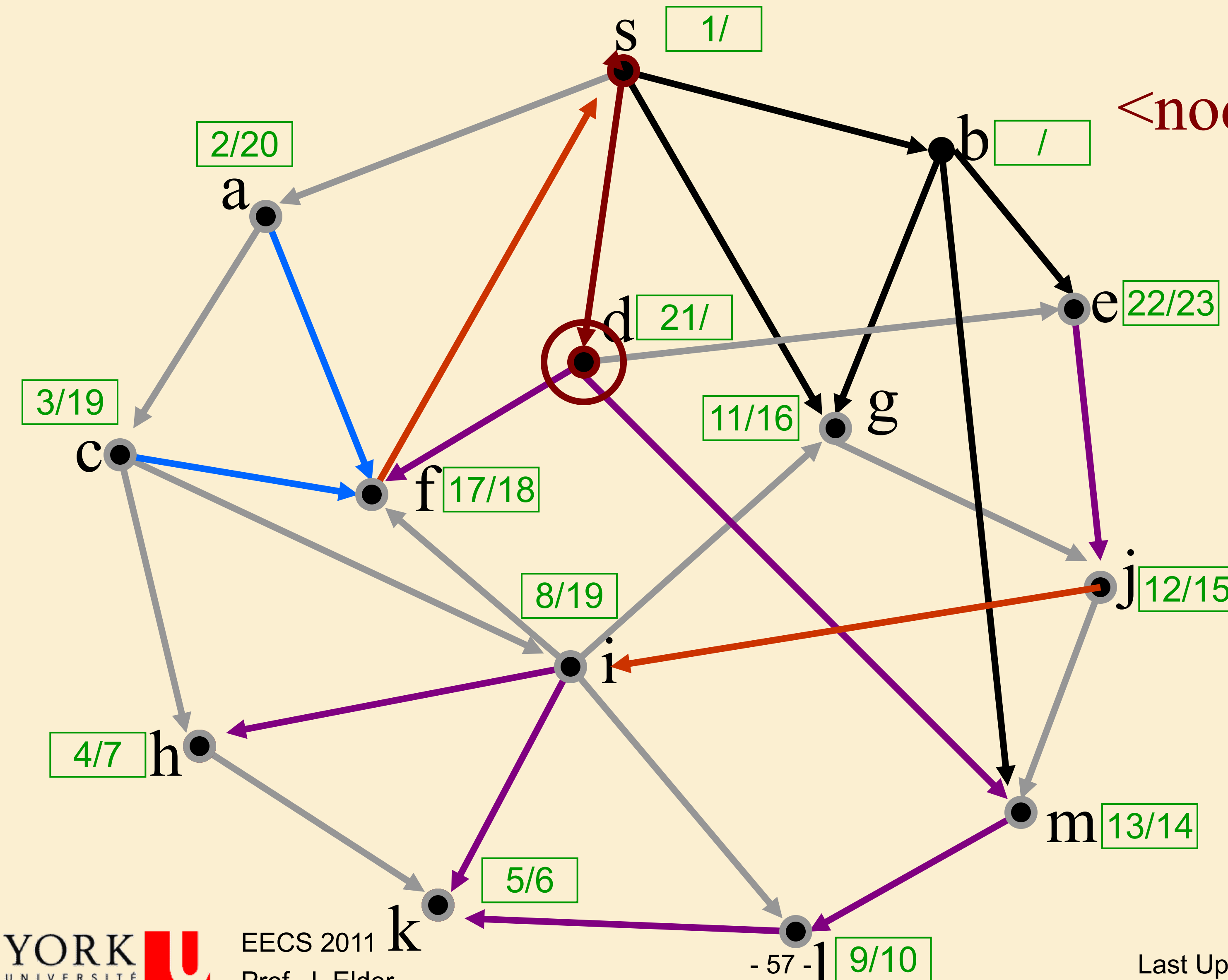
DFS



Discovered
Not Finished
Stack
<node,# edges>

e,1
d,3
s,2

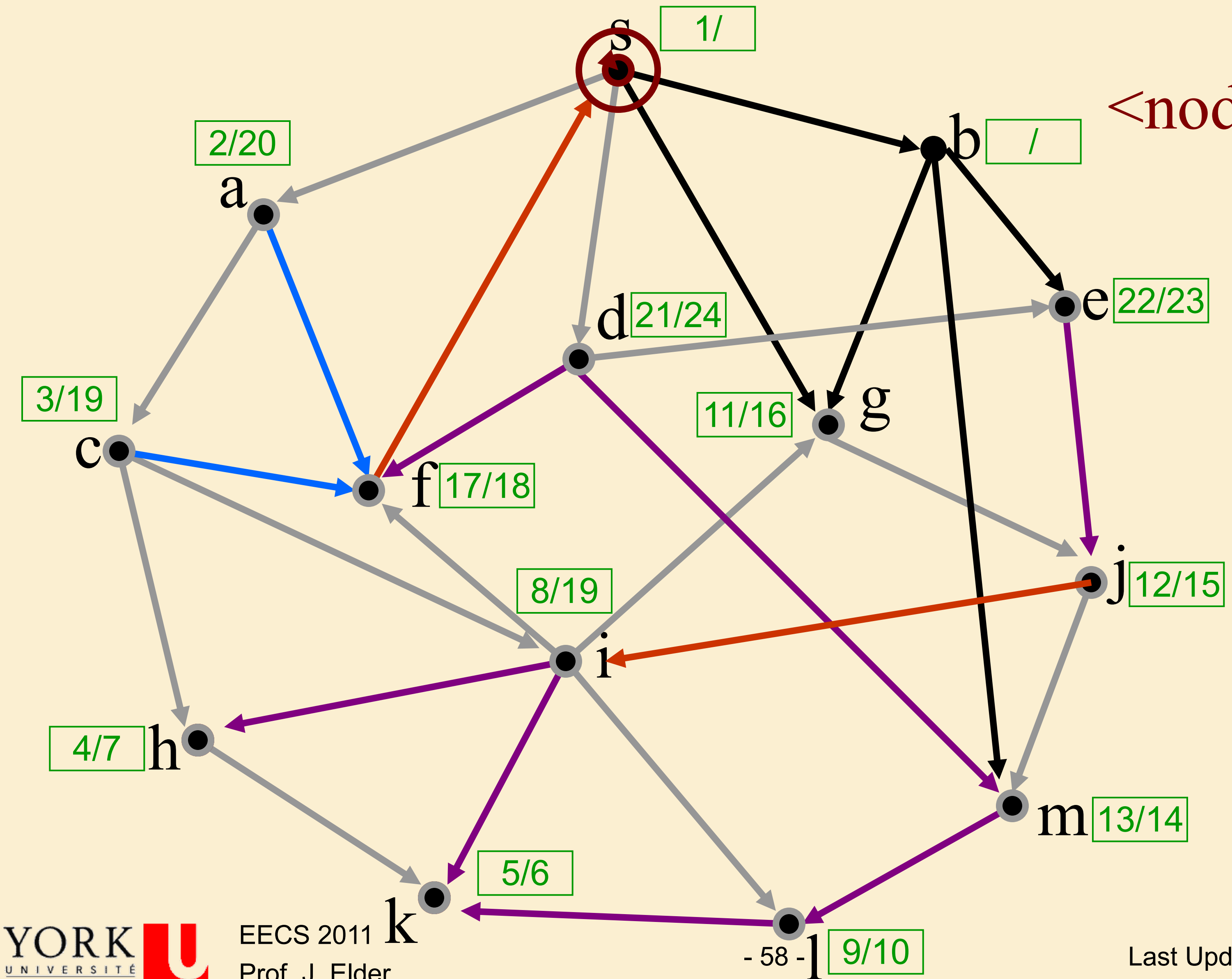
DFS



Discovered
Not Finished
Stack
<node,# edges>

d,3
s,2

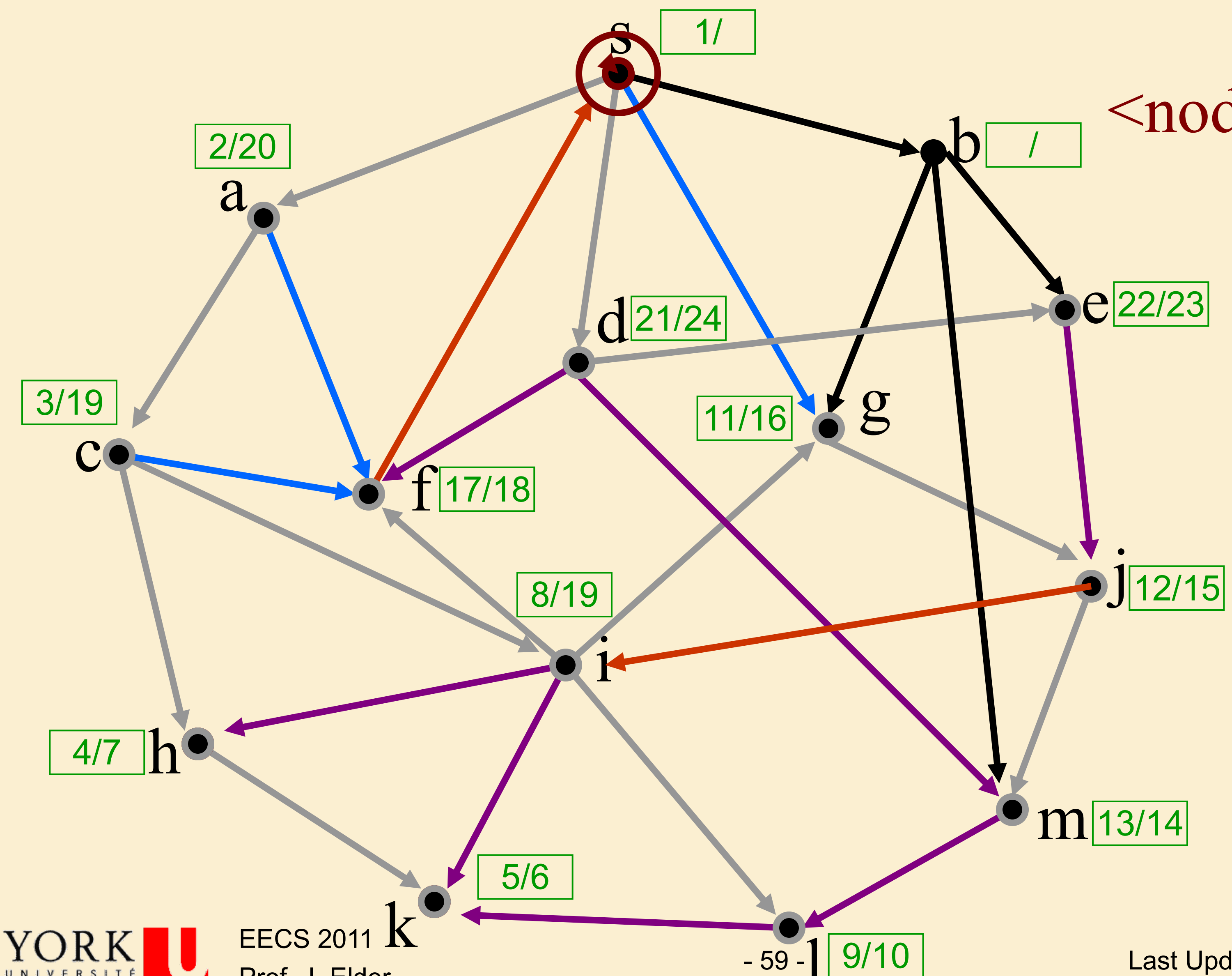
DFS



Discovered
Not Finished
Stack
<node,# edges>

s,2

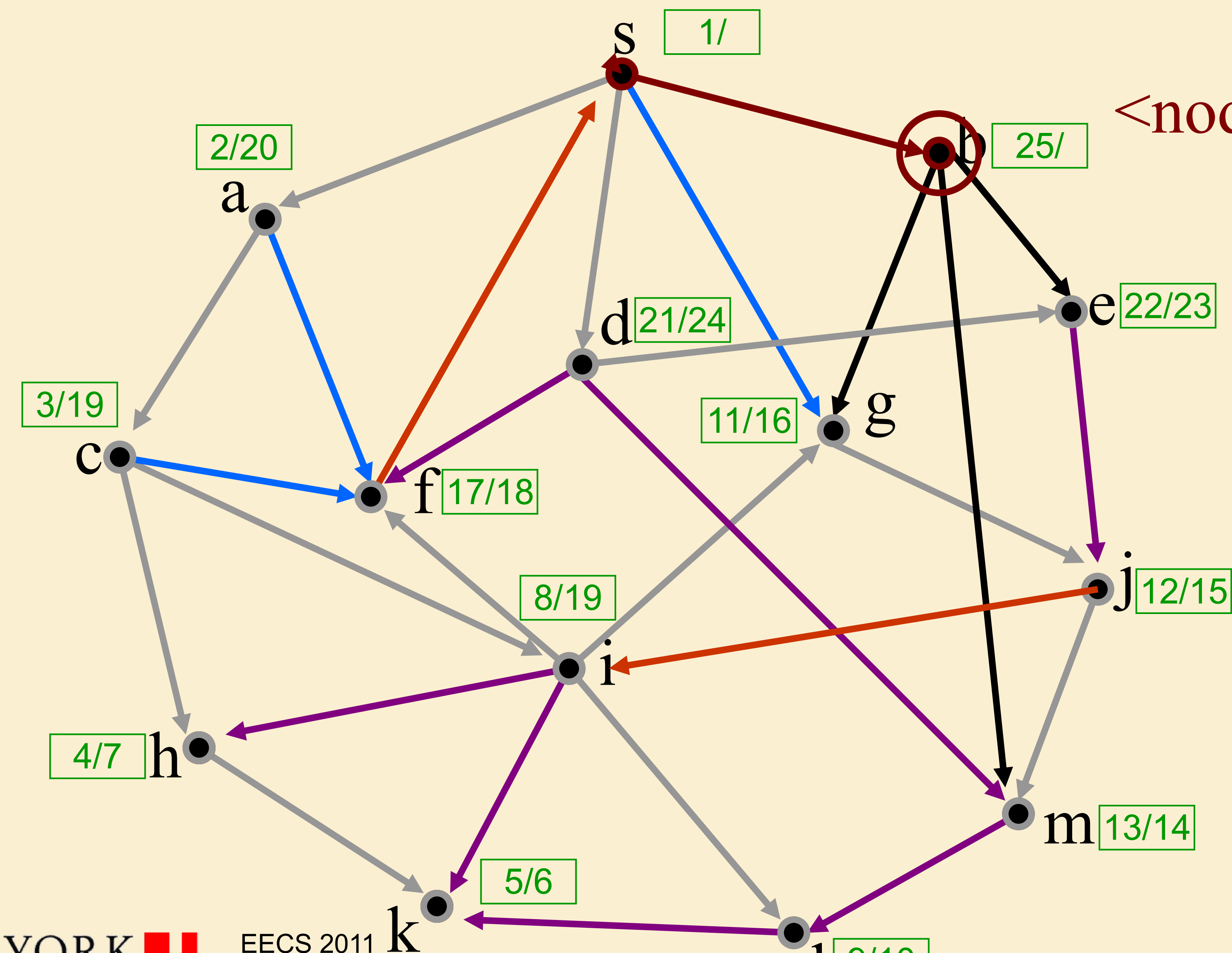
DFS



Discovered
Not Finished
Stack
<node,# edges>

s,3

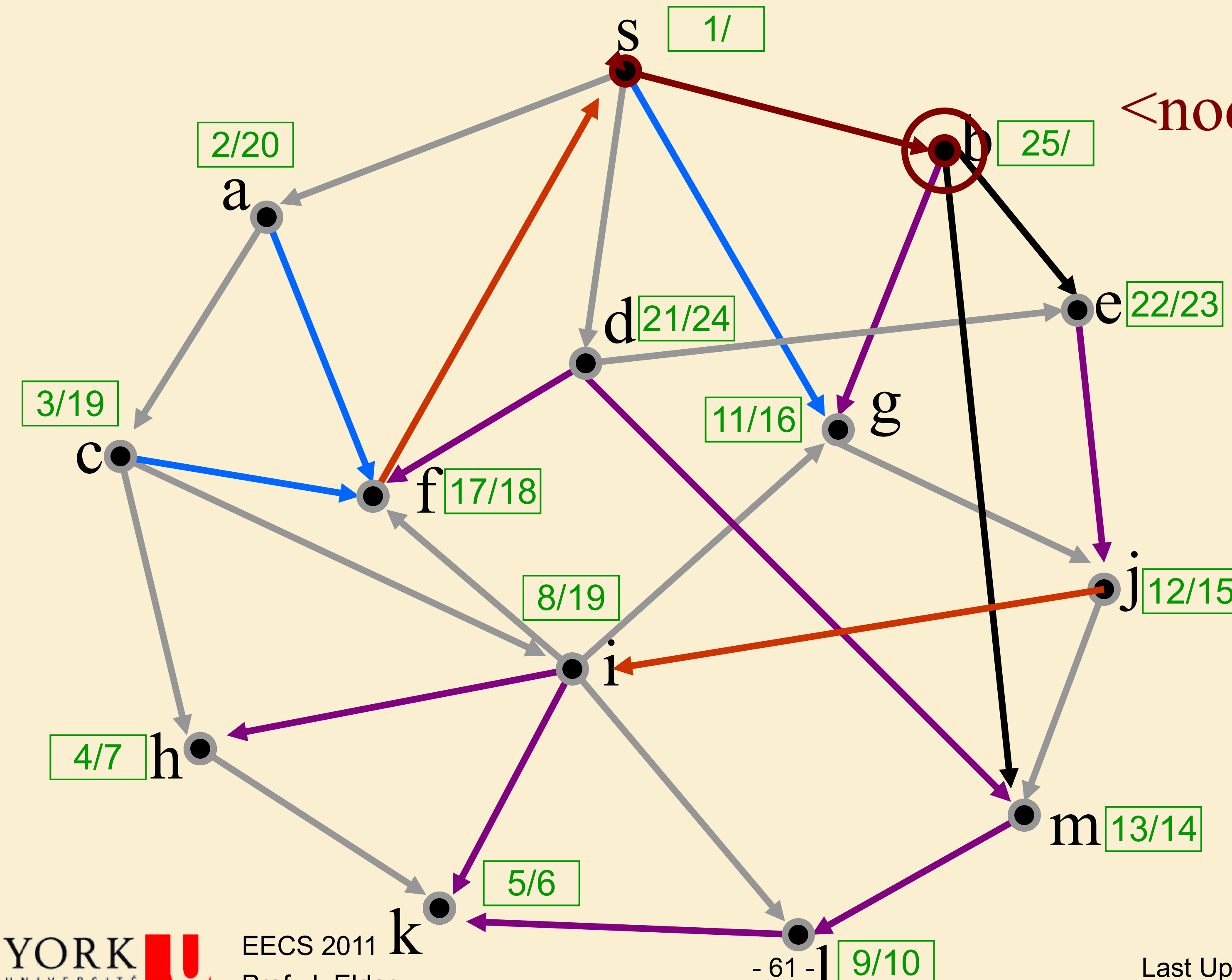
DFS



Discovered
Not Finished
Stack
<node,# edges>

b,0
s,4

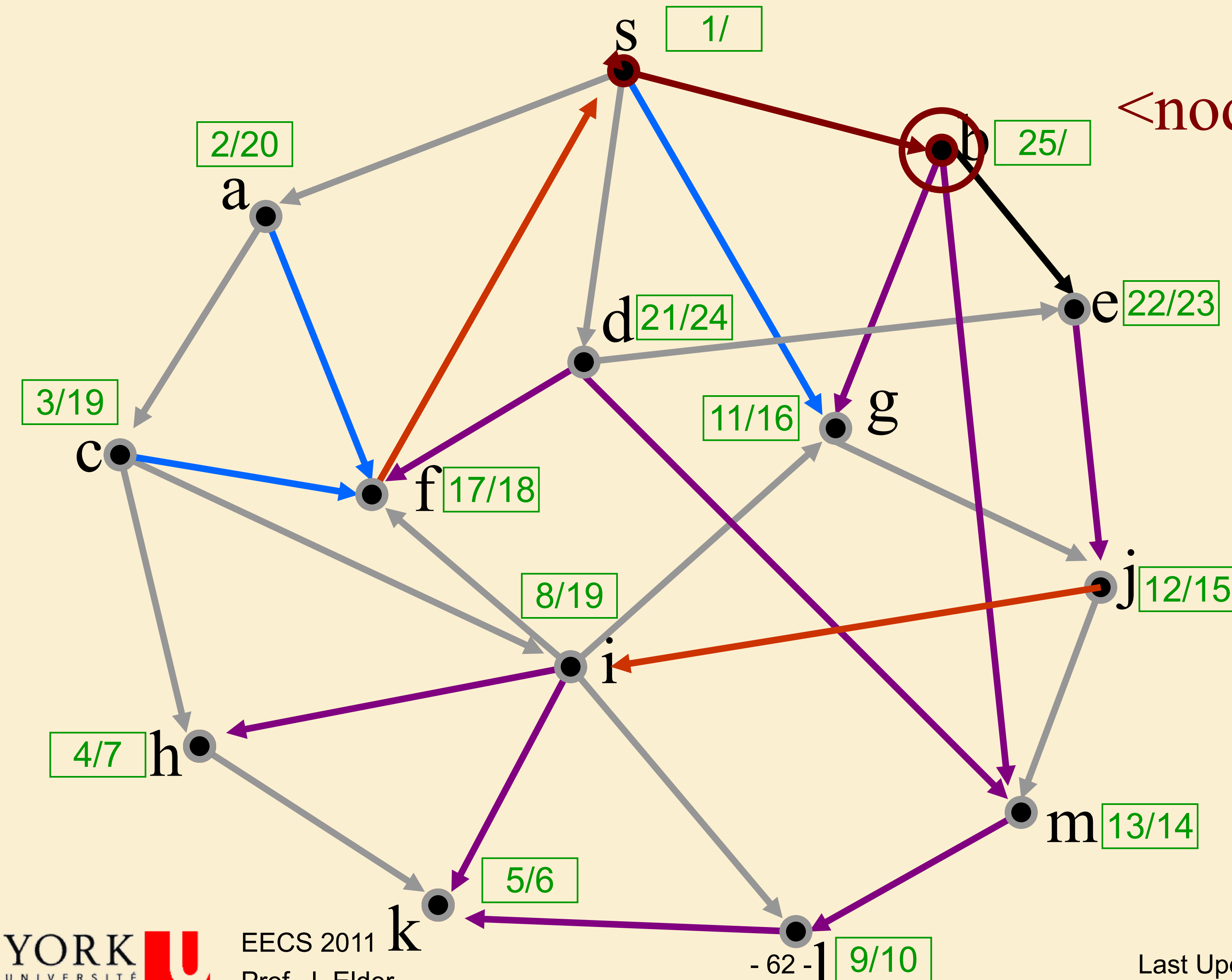
DFS



Discovered
Not Finished
Stack
<node,# edges>

b,1
s,4

DFS

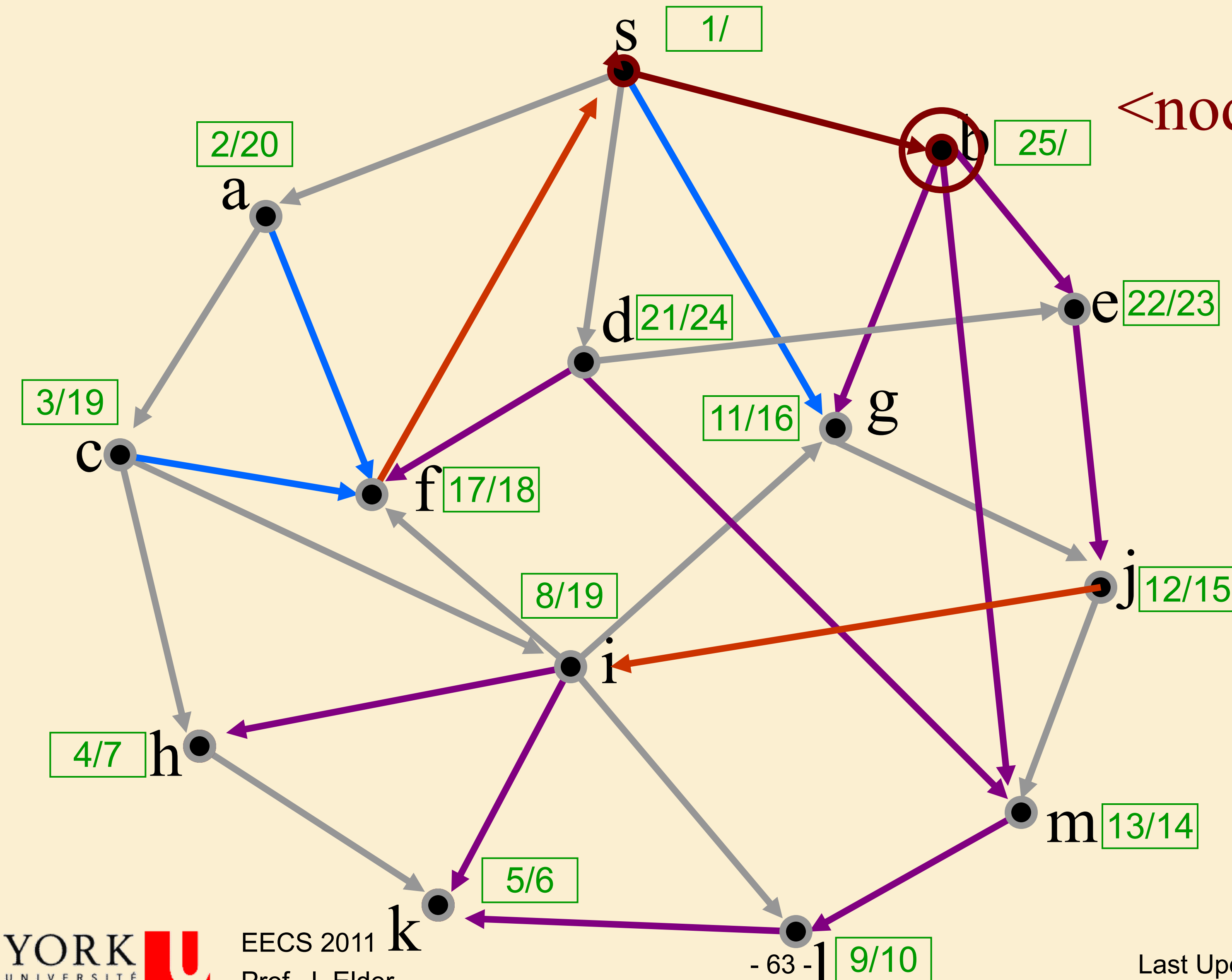


Discovered
Not Finished
Stack
<node,# edges>

b,2
s,4

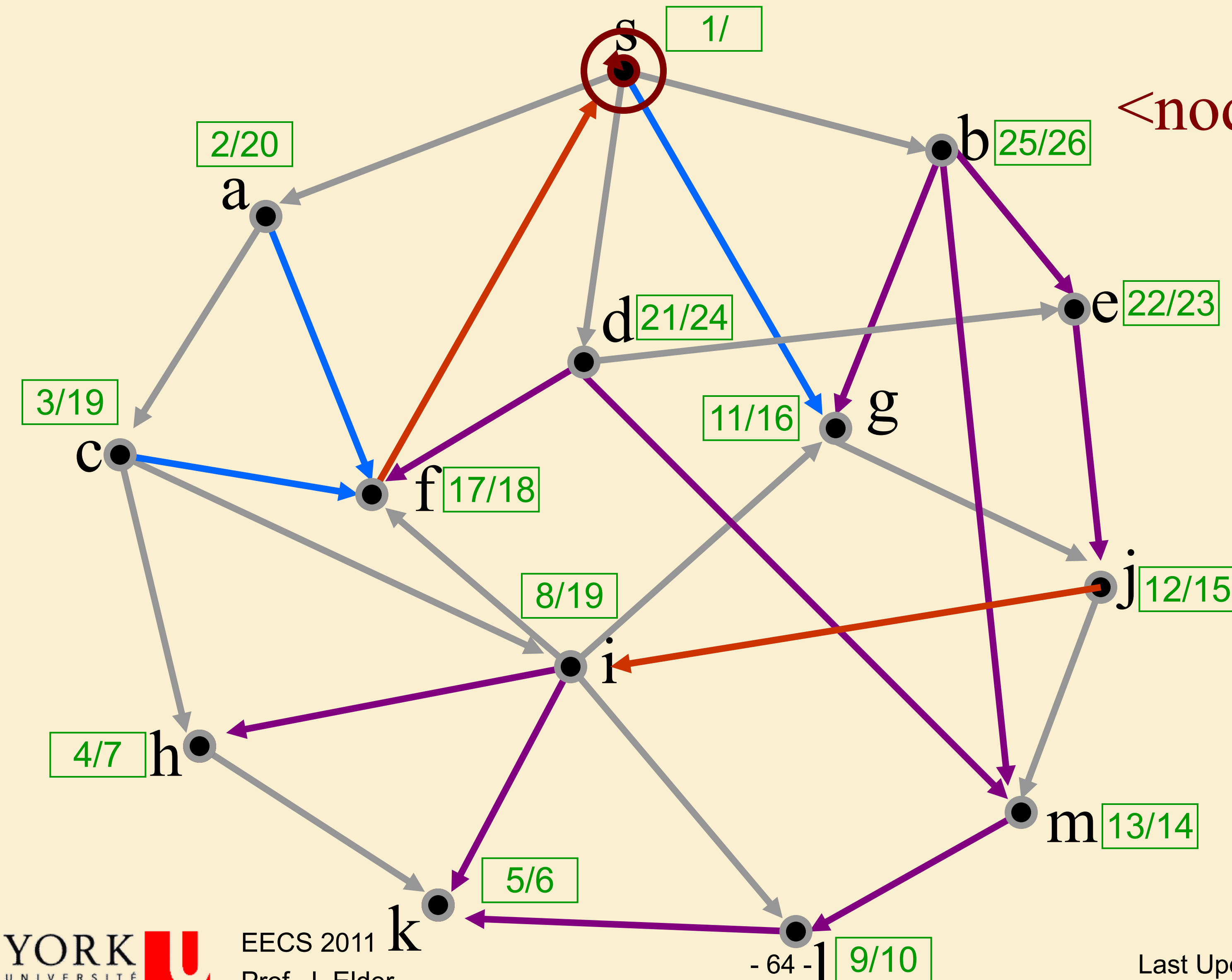
DFS

Discovered
Not Finished
Stack
<node,# edges>



b,3
s,4

DFS



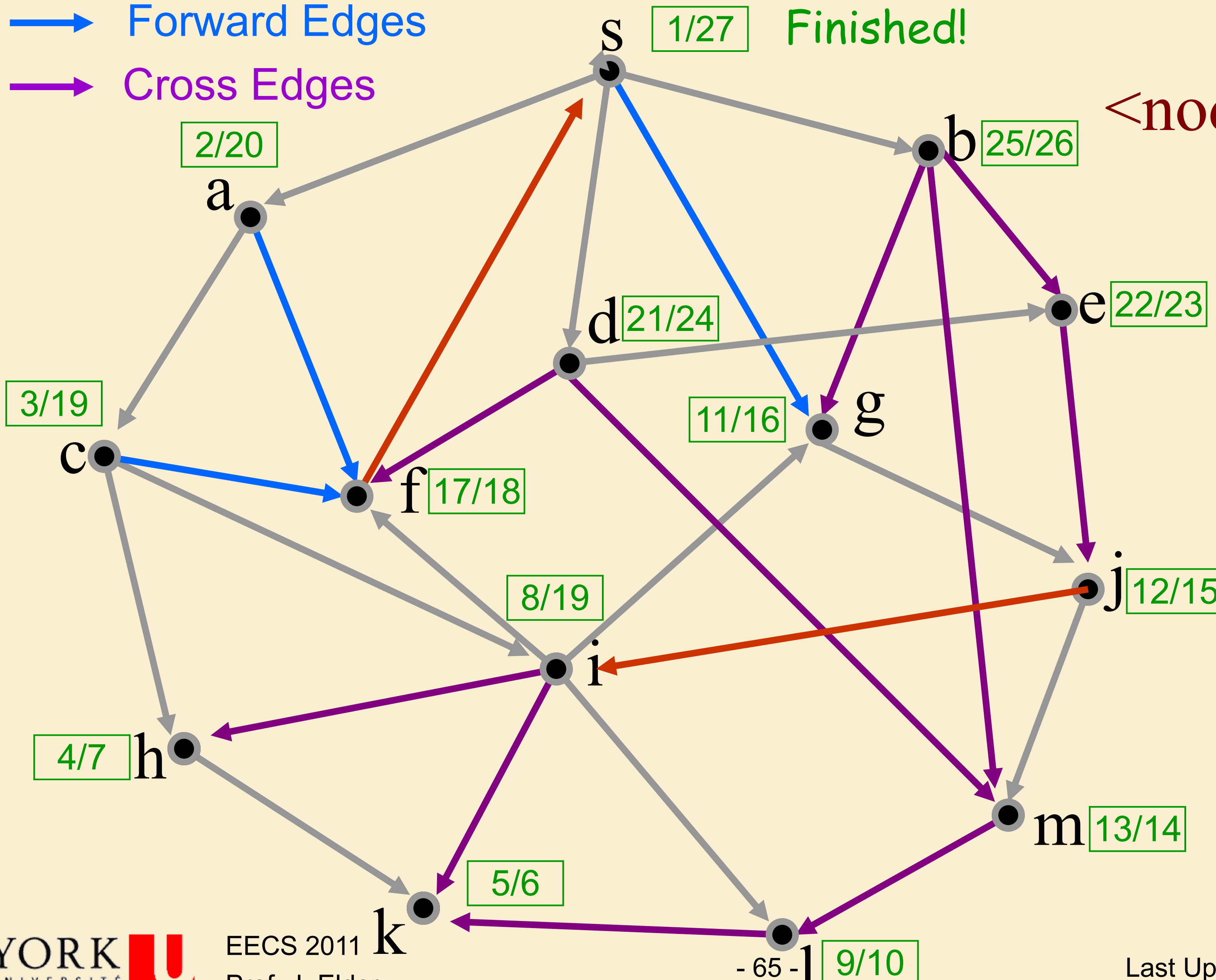
Discovered
Not Finished
Stack
<node,# edges>

s,4

DFS

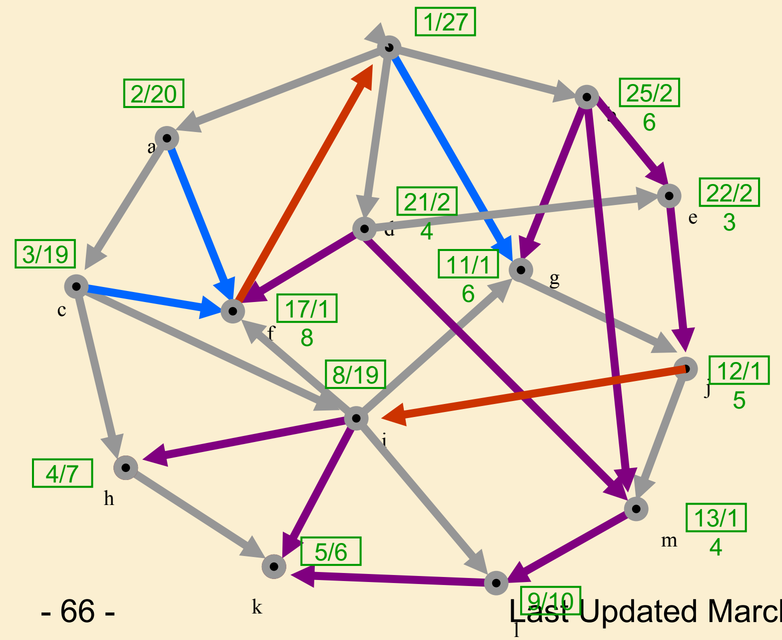
Tree Edges
Back Edges
Forward Edges
Cross Edges

Discovered
Not Finished
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

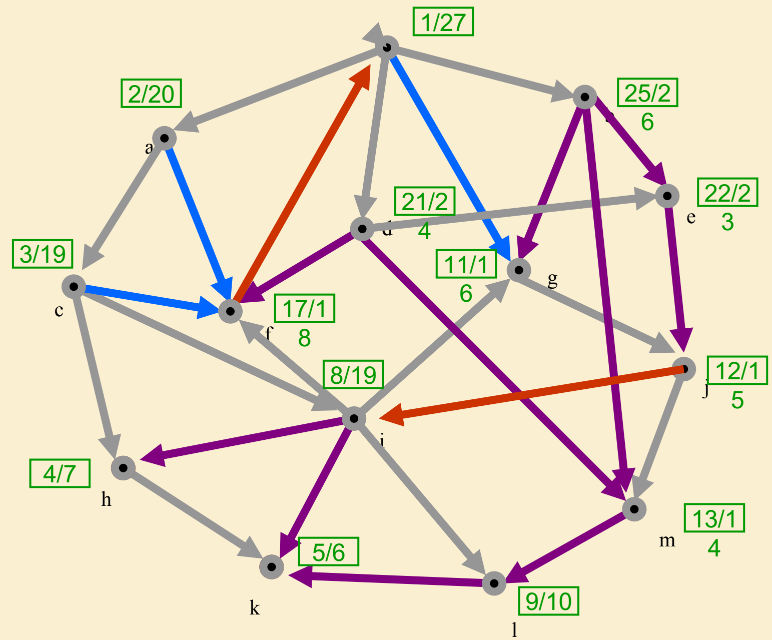
1. ***Tree edges***: Edge (*u*, *v*) is a **tree edge** if *v* was **black** when (*u*, *v*) traversed. Note that *d[v] > d[u].*

2. ***Back edges:*** (*u*, *v*) is a **back edge** if v was **red** when *(u, v)* traversed. Note that *d[v] < d[u].*

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.

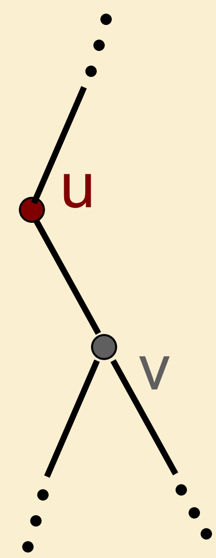


DFS on Undirected Graphs

- 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 **Finished (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 Finished: **u** must be **grey**.
- **Contradiction!**



Outline

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

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] ← RED
        push u onto stack
        if u = z
                return TRUE
        for each v ∈ Adj[u] //explore edge (u,v)
                if color[v] = BLACK
                        if DFS-Path(v,z,stack)
                                return TRUE
        colour[u] ← GRAY
        pop u from stack
        return FALSE
```

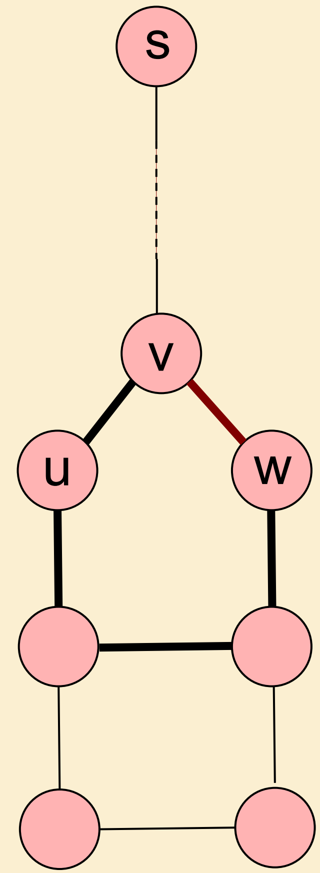
DFS Application 2: Cycle Finding

- 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 ∈ 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
```

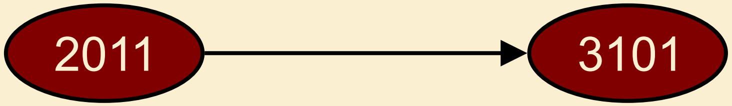
Why must DFS on a graph with a cycle generate a back edge?

- 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.



A4Q2: Course Prerequisites

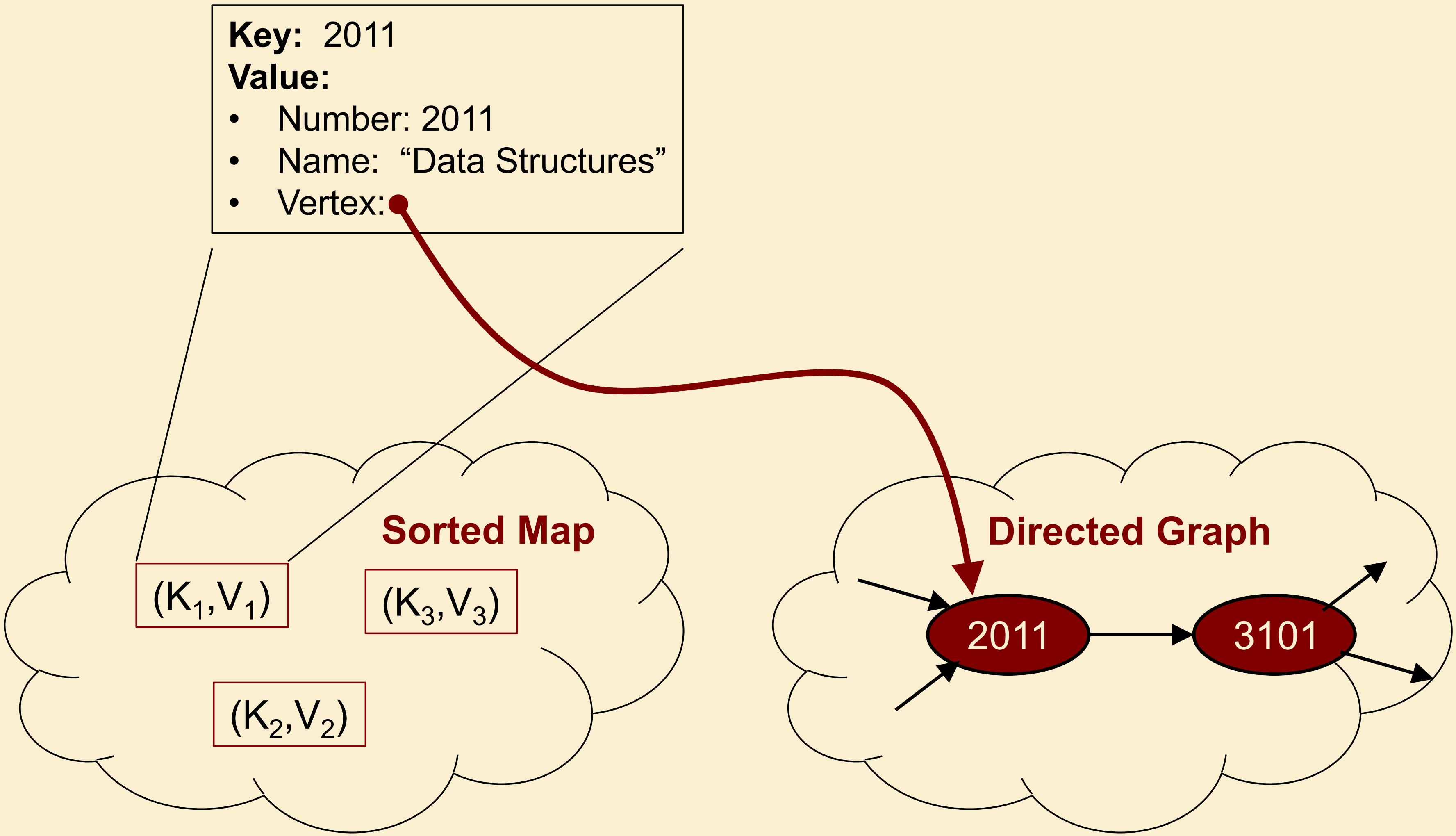
- In most post-secondary programs, courses have prerequisites.
- For example, you cannot take EECS 3101 until you have passed EECS 2011.
- How can we represent such a system of dependencies?
- A natural choice is a **directed graph**.
  - Each vertex represents a course
  - Each directed edge represents a prerequisite
    - A directed edge from Course U to Course V means that Course U must be taken before Course V.



A4Q2: Course Prerequisites

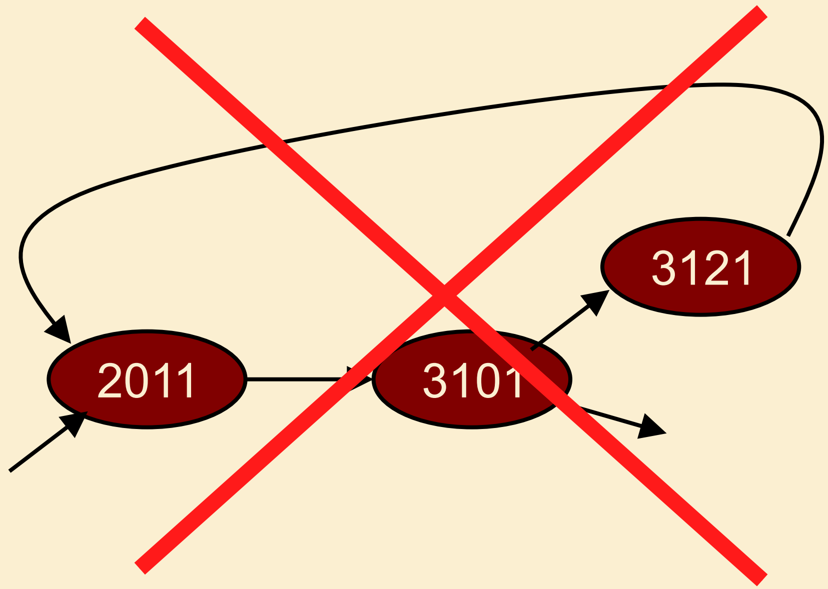
- We also want to be able to find the information for a particular course quickly.
- The course number provides a convenient key that can be used to organize course records in a sorted map, implemented as a binary search tree (cf. A3Q1).
- Thus it makes sense to represent courses using both a sorted map (for efficient access) and a directed graph (to represent dependencies).
- By storing a reference to the directed graph vertex for a course in the sorted map, we can efficiently access course dependencies.

A4Q2: Course Prerequisites



A4Q2: Course Prerequisites

- It is important that the course prerequisite graph be a directed acyclic graph (DAG). Why?

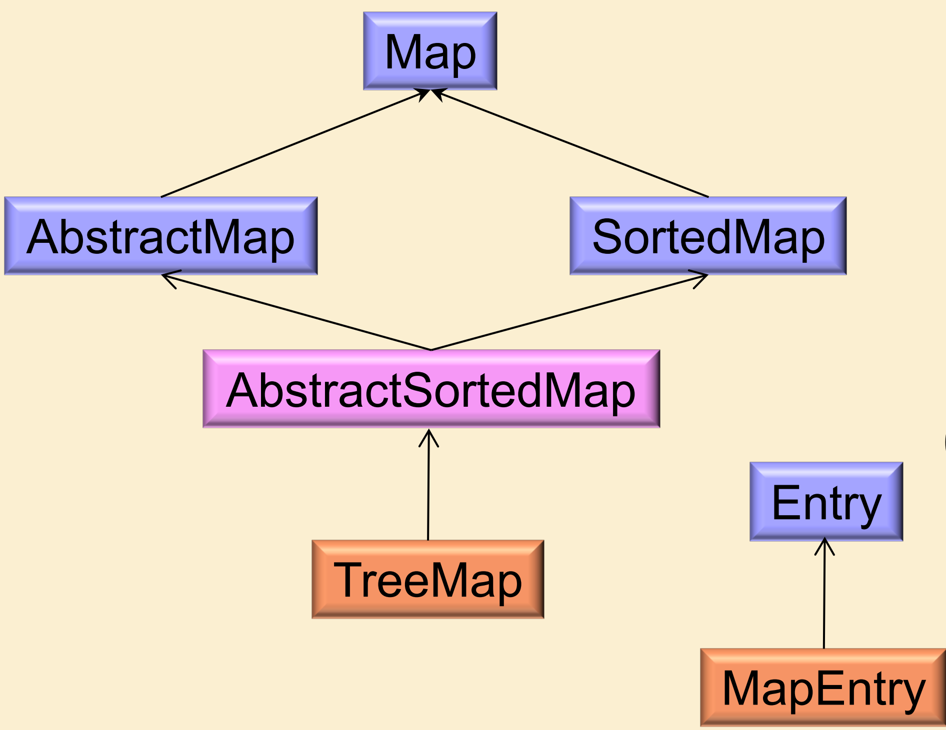


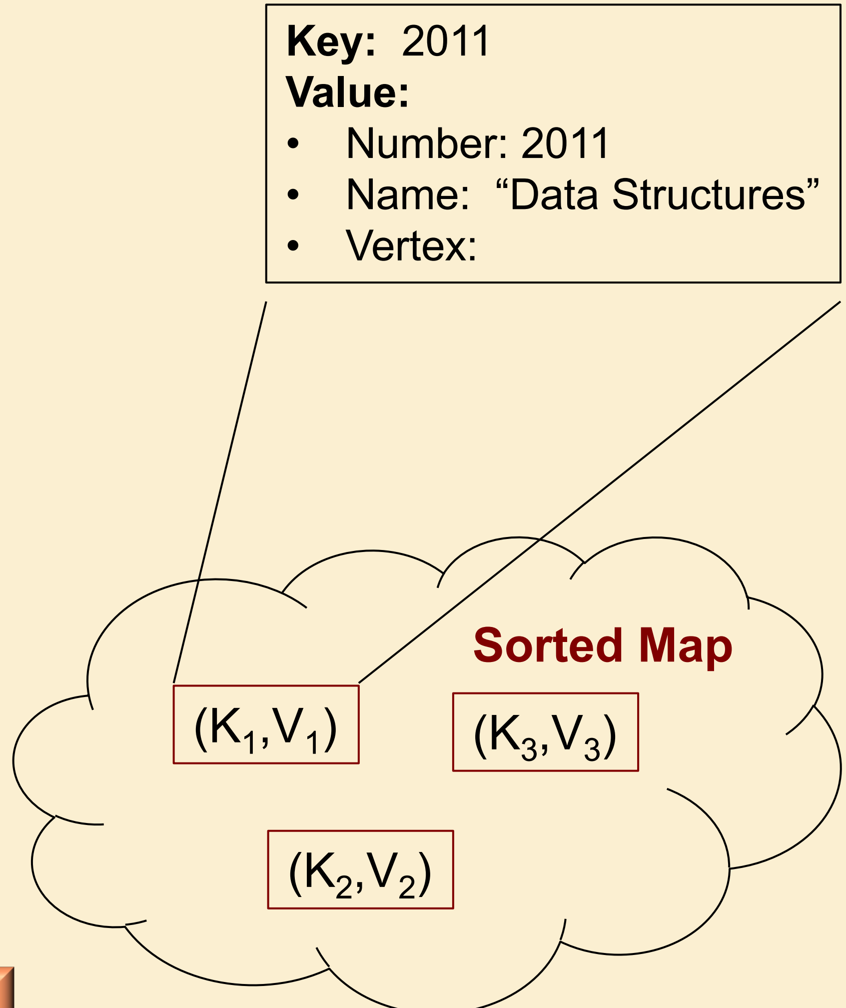
A4Q2: Course Prerequisites

- In this question, you are provided with a basic implementation of a system to represent courses and dependencies.
- Methods for adding courses and getting prerequisites are provided.
- You need only write the method for adding a prerequisite.
- This method will use a depth-first-search algorithm (also provided) that can be used to prevent the addition of prerequisites that introduce cycles.

A4Q2: Implementation using net.datastructures

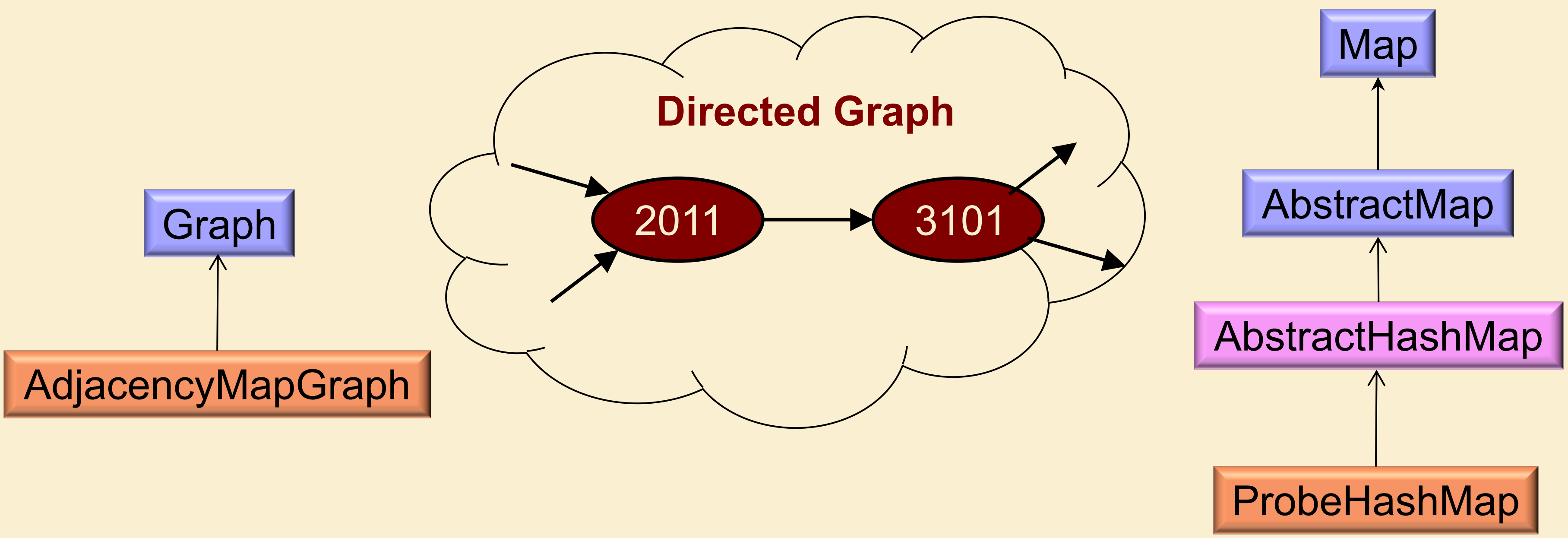
- We use the TreeMap class to represent the sorted map (cf. A3Q1).





A4Q2: Implementation using net.datastructures

- We use the **AdjacencyMapGraph** class to represent the directed graph.
- This implementation uses **ProbeHashMap**, a linear probe hash table, to represent the incoming and outgoing edges for each vertex.



Outline

- DFS Algorithm
- DFS Example
- DFS Applications

Outcomes

- By understanding this lecture, you should be able to:
  - Label a graph according to the order in which vertices are discovered, explored from and finished in a depth-first search.
  - Classify edges of the depth-first search as tree edges, back edges, forward edges and cross edges
  - Implement depth-first search
  - Demonstrate simple applications of depth-first search