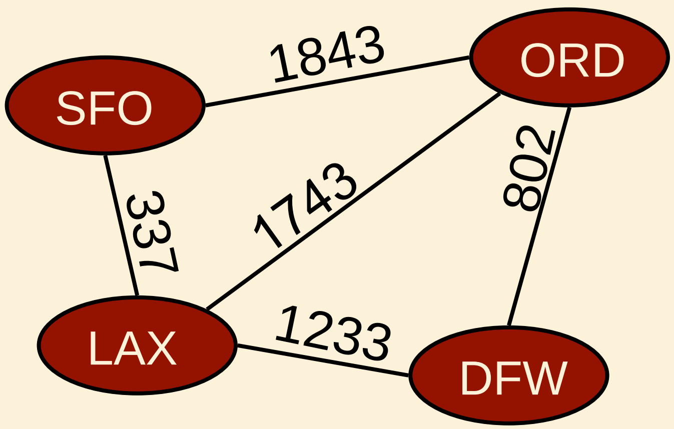
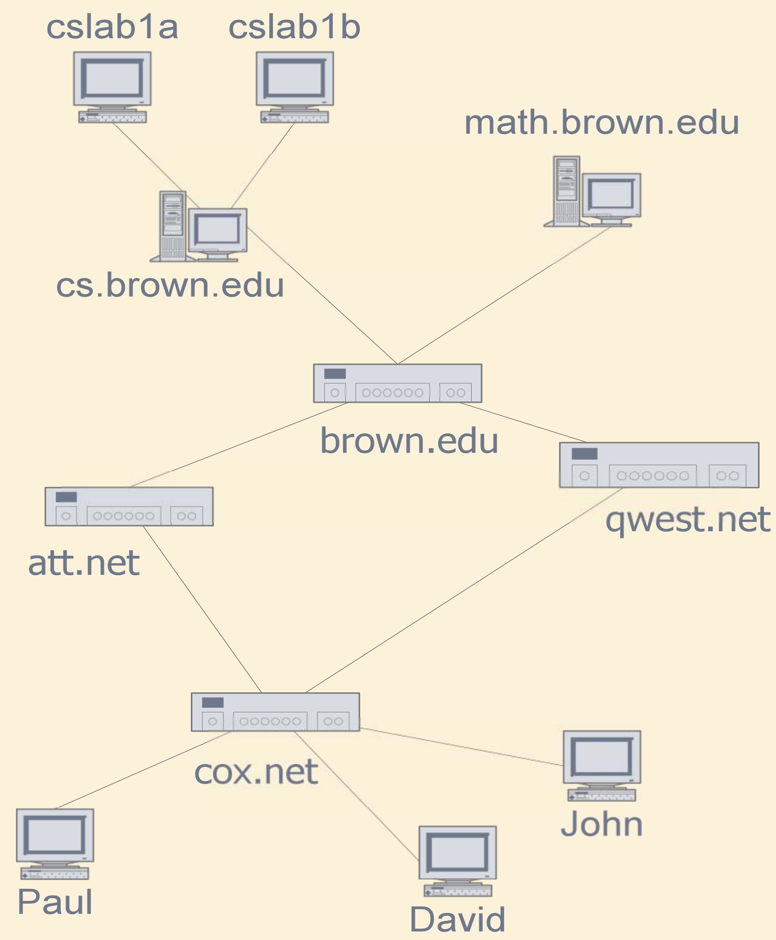
Graphs



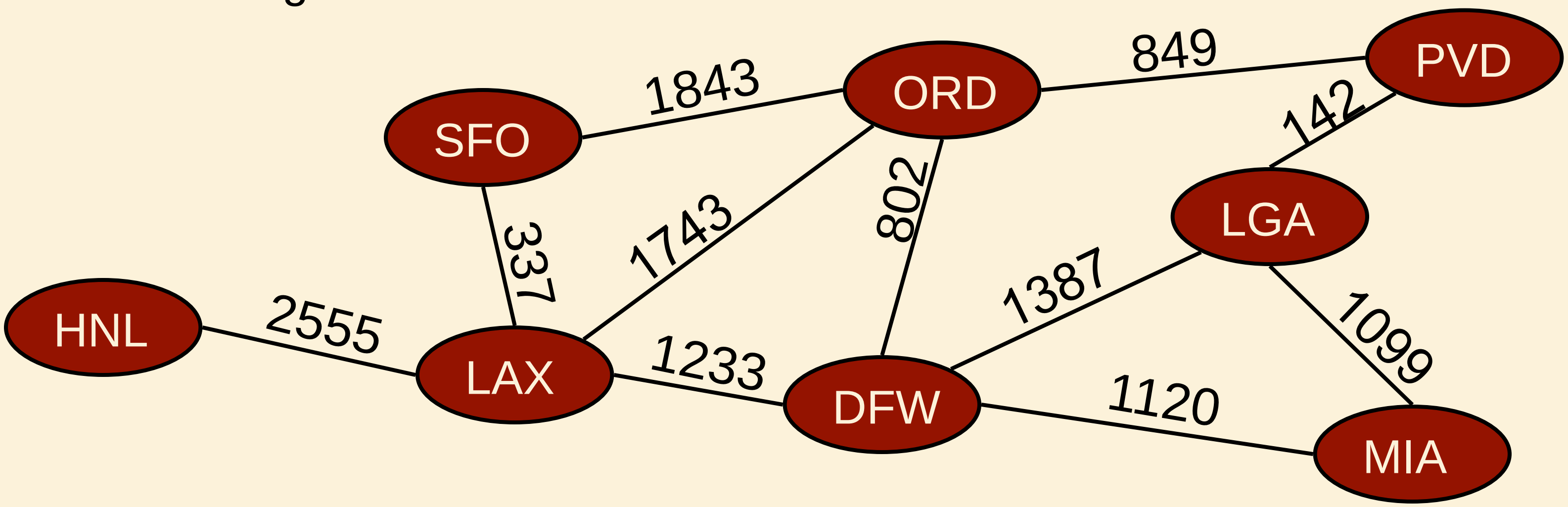
Applications

- Electronic circuits
  - Printed circuit board
  - Integrated circuit
- Transportation networks
  - Highway network
  - Flight network
- Computer networks
  - Local area network
  - Internet
  - Web
- Databases
  - Entity-relationship diagram



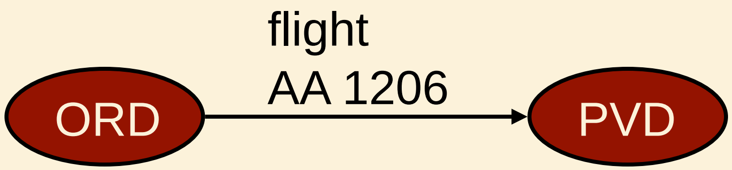
Graphs

- A graph is a pair (V, E), where
  - V is a set of nodes, called vertices
  - E is a collection of pairs of vertices, called edges
  - Vertices and edges are positions and store elements
- Example:
  - A vertex represents an airport and stores the three-letter airport code
  - An edge represents a flight route between two airports and stores the mileage of the route

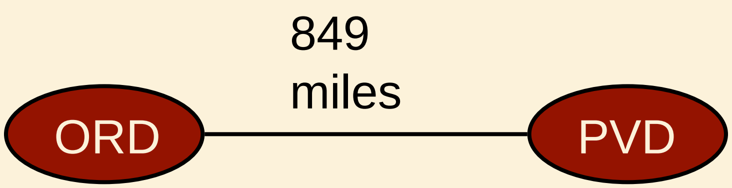


Edge Types

- Directed edge
  - ordered pair of vertices (u,v)
  - first vertex u is the origin
  - second vertex v is the destination
  - e.g., a flight



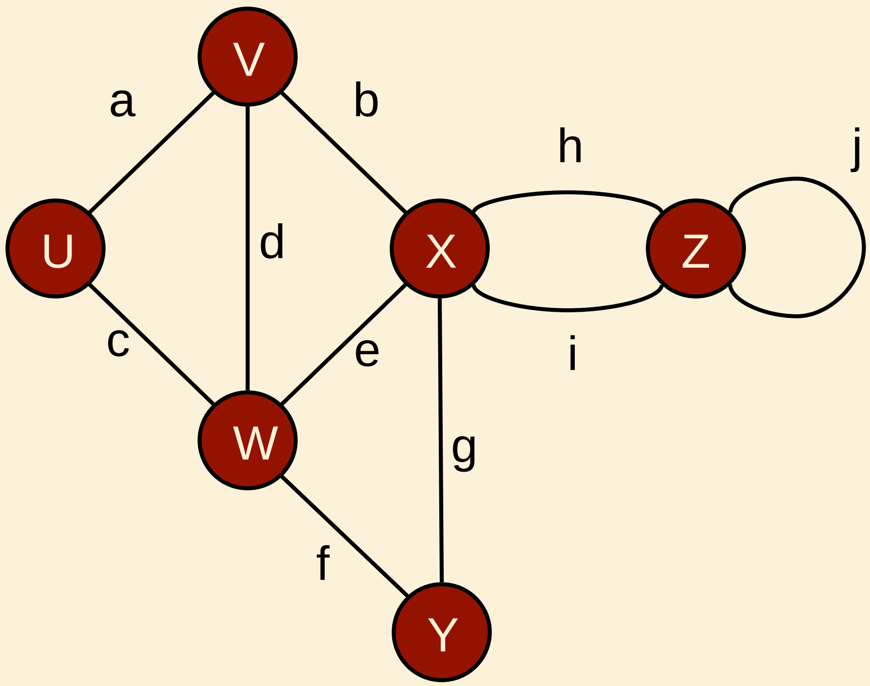
- Undirected edge
  - unordered pair of vertices (u,v)
  - e.g., a flight route



- Directed graph (Digraph)
  - all the edges are directed
  - e.g., route network
- Undirected graph
  - all the edges are undirected
  - e.g., flight network

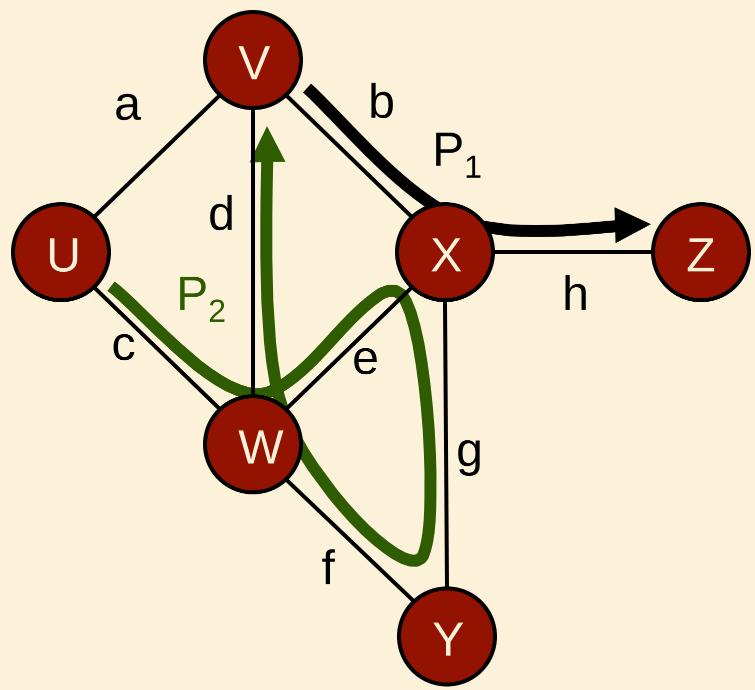
Vertices and Edges

- End vertices (or endpoints) of an edge
  - U and V are the endpoints of a
- Edges incident on a vertex
  - a, d, and b are incident on V
- Adjacent vertices
  - U and V are adjacent
- Degree of a vertex
  - X has degree 5
- Parallel edges
  - h and i are parallel edges
- Self-loop
  - j is a self-loop



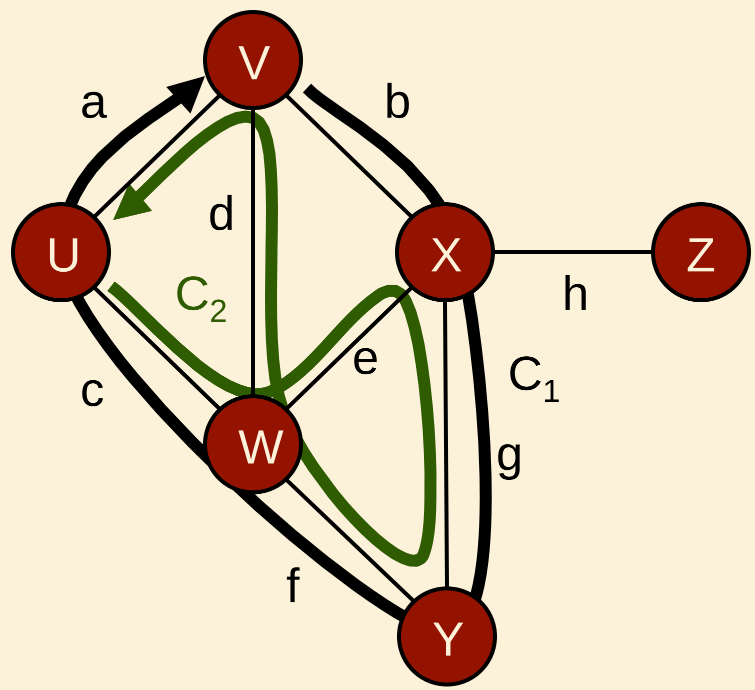
Paths

- Path
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is preceded and followed by its endpoints
- Simple path
  - path such that all its vertices and edges are distinct
- Examples
  - P_1=(V,b,X,h,Z) is a simple path
  - P_2=(U,c,W,e,X,g,Y,f,W,d,V) is a path that is not simple



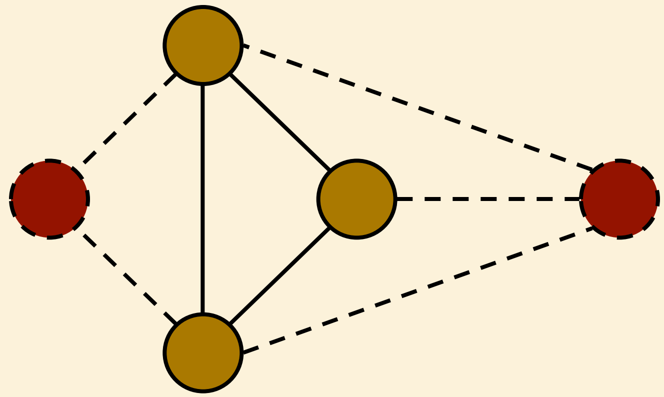
Cycles

- Cycle
  - circular sequence of alternating vertices and edges
  - each edge is preceded and followed by its endpoints
- Simple cycle
  - cycle such that all its vertices and edges are distinct
- Examples
  - C_1=(V,b,X,g,Y,f,W,c,U,a,↵) is a simple cycle
  - C_2=(U,c,W,e,X,g,Y,f,W,d,V,a,↵) is a cycle that is not simple

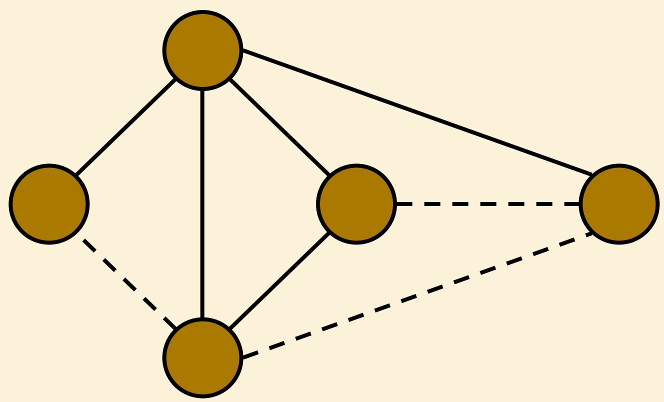


Subgraphs

- A subgraph S of a graph G is a graph such that
  - The vertices of S are a subset of the vertices of G
  - The edges of S are a subset of the edges of G
- A spanning subgraph of G is a subgraph that contains all the vertices of G



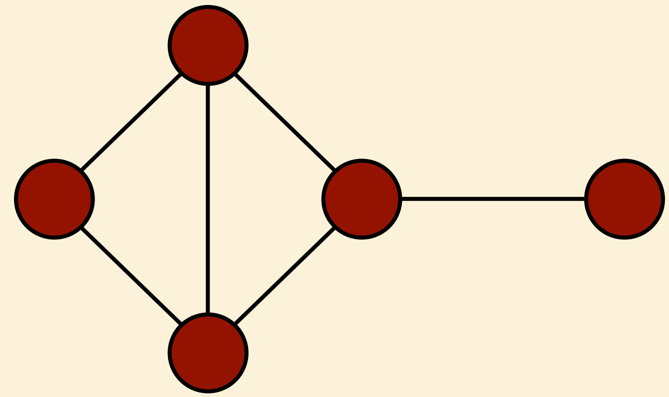
Subgraph



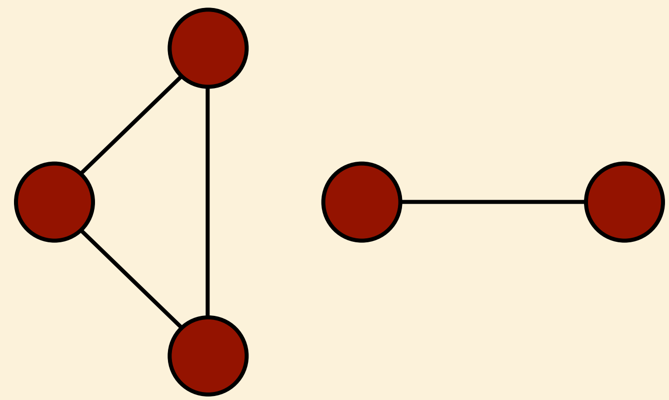
Spanning subgraph

Connectivity

- A graph is connected if there is a path between every pair of vertices
- A connected component of a graph G is a maximal connected subgraph of G

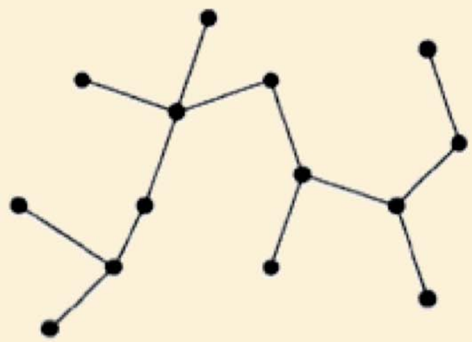


Connected graph

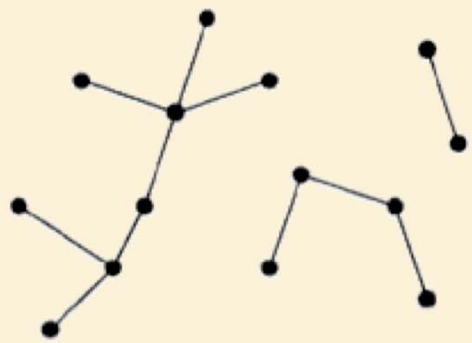


Non connected graph with two connected components

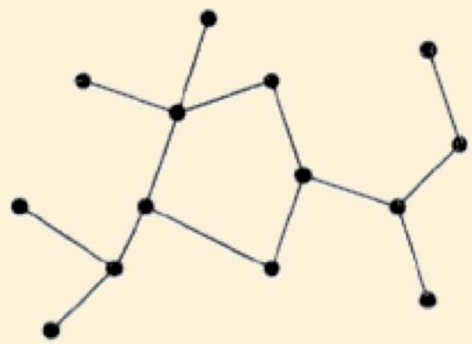
Trees



Tree



Forest



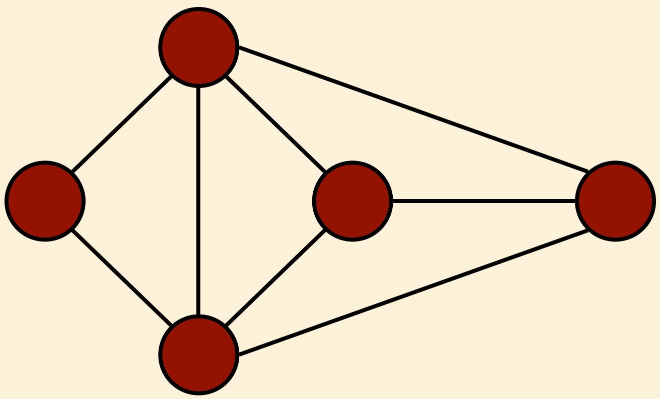
Graph with Cycle

A tree is a connected, acyclic, undirected graph.

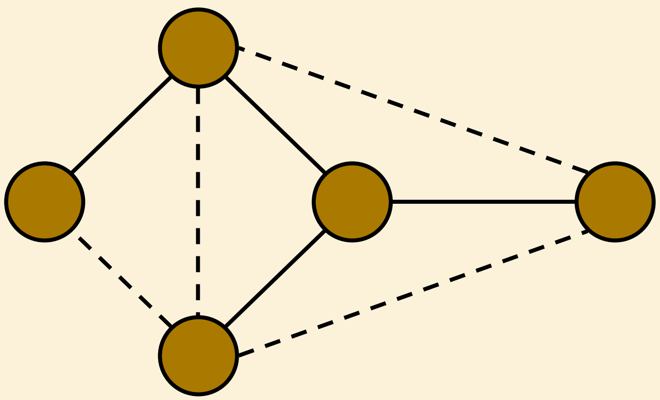
A forest is a set of trees (not necessarily connected)

Spanning Trees

- A spanning tree of a connected graph is a spanning subgraph that is a tree
- A spanning tree is not unique unless the graph is a tree
- Spanning trees have applications to the design of communication networks
- A spanning forest of a graph is a spanning subgraph that is a forest



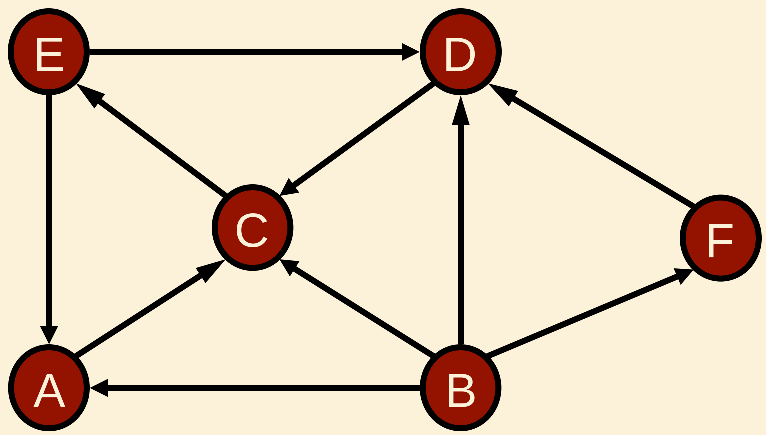
Graph



Spanning tree

Reachability in Directed Graphs

- A node w is ***reachable*** from v if there is a directed path originating at v and terminating at w.
  - E is reachable from B
  - B is not reachable from E



Properties

Property 1

$$\sum_v \deg(v) = 2|E|$$

Proof: each edge is counted twice

Property 2

In an undirected graph with no self-loops and no multiple edges

$$|E| \le |V|\,(|V| - 1)/2$$

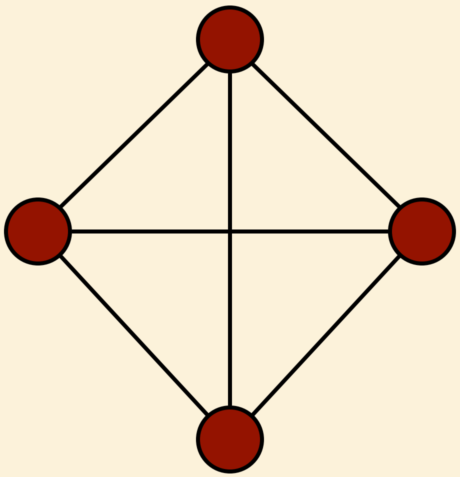
Proof: each vertex has degree at most $(|V| - 1)$

Q: What is the bound for a digraph?

$A:\ |E| \le |V|(|V|-1)$

Notation

$\|V\|$	number of vertices
$\|E\|$	number of edges
$\deg(v)$	degree of vertex v



Example

- $|V| = 4$
- $|E| = 6$
- $\deg(v) = 3$

Main Methods of the (Undirected) Graph ADT

- Vertices and edges
  - are positions
  - store elements
- Accessor methods
  - endVertices(e): an array of the two endvertices of e
  - opposite(v, e): the vertex opposite to v on e
  - areAdjacent(v, w): true iff v and w are adjacent
  - replace(v, x): replace element at vertex v with x
  - replace(e, x): replace element at edge e with x
- Update methods
  - insertVertex(o): insert a vertex storing element o
  - insertEdge(v, w, o): insert an edge (v,w) storing element o
  - removeVertex(v): remove vertex v (and its incident edges)
  - removeEdge(e): remove edge e
- Iterator methods
  - incidentEdges(v): edges incident to v
  - vertices(): all vertices in the graph
  - edges(): all edges in the graph

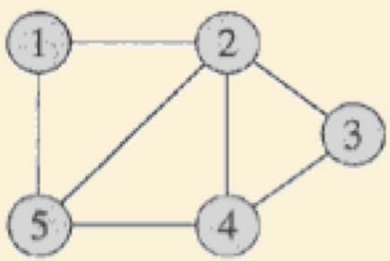
Directed Graph ADT

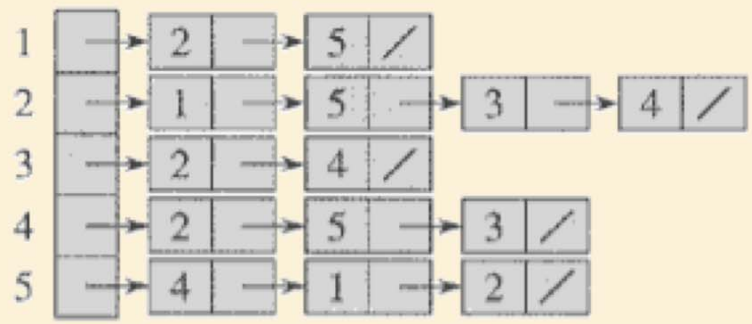
- Additional methods:
  - isDirected(e): return true if e is a directed edge
  - insertDirectedEdge(v, w, o): insert and return a new directed edge with origin *v* and destination *w*, storing element *o*

Running Time of Graph Algorithms

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

Implementing a Graph (Simplified)





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

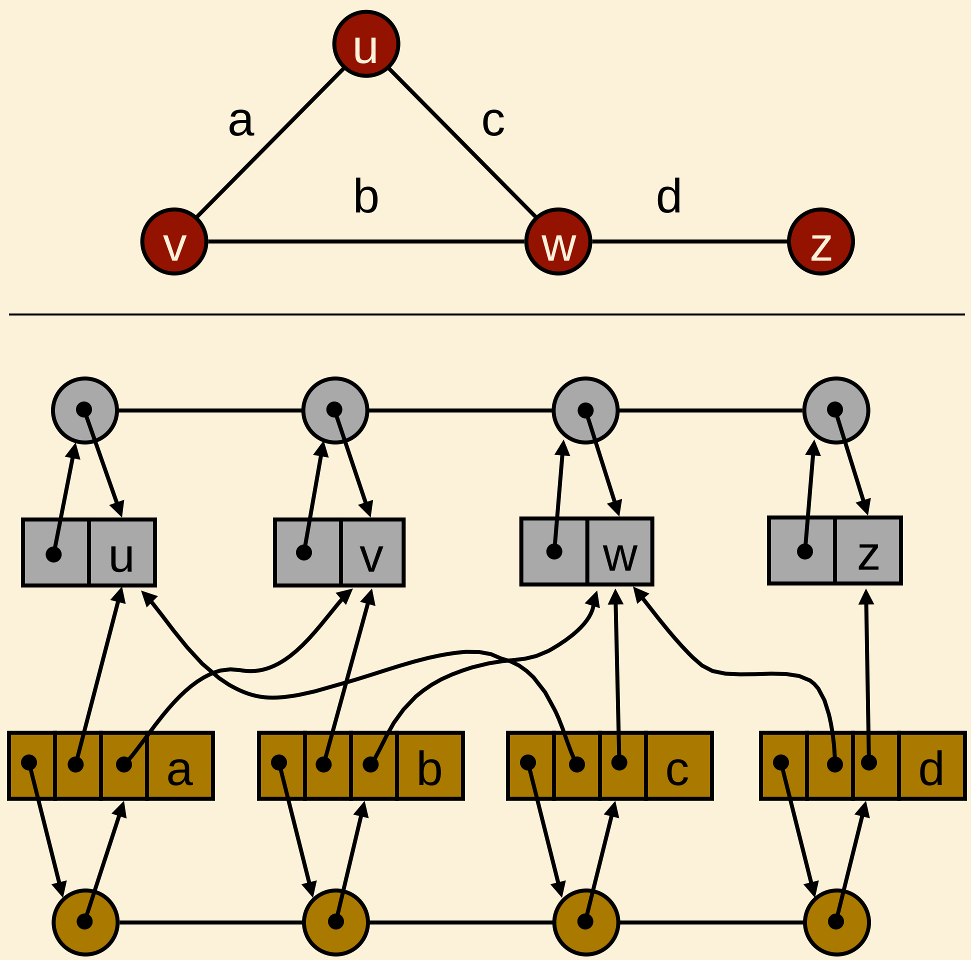
	Adjacency List	Adjacency Matrix
Space complexity:	$\theta(V+E)$	$\theta(V^2)$
Time to find all neighbours of vertex u :	$\theta(\text{degree}(u))$	$\theta(V)$
Time to determine if $(u,v) \in E$:	$\theta(\text{degree}(u))$	$\theta(1)$

Representing Graphs (Details)

- Three basic methods
  - Edge List
  - Adjacency List
  - Adjacency Matrix

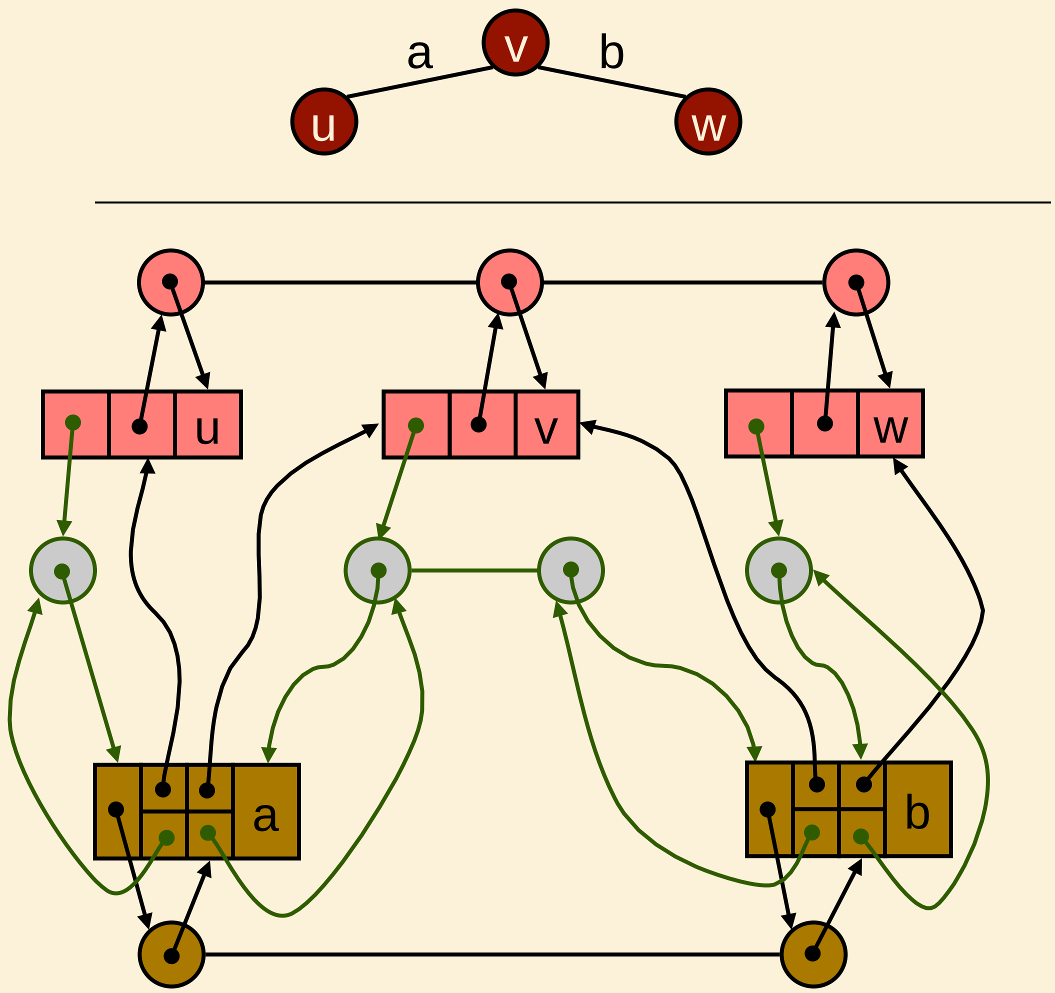
Edge List Structure

- Vertex object
  - element
  - reference to position in vertex sequence
- Edge object
  - element
  - origin vertex object
  - destination vertex object
  - reference to position in edge sequence
- Vertex sequence
  - sequence of vertex objects
- Edge sequence
  - sequence of edge objects



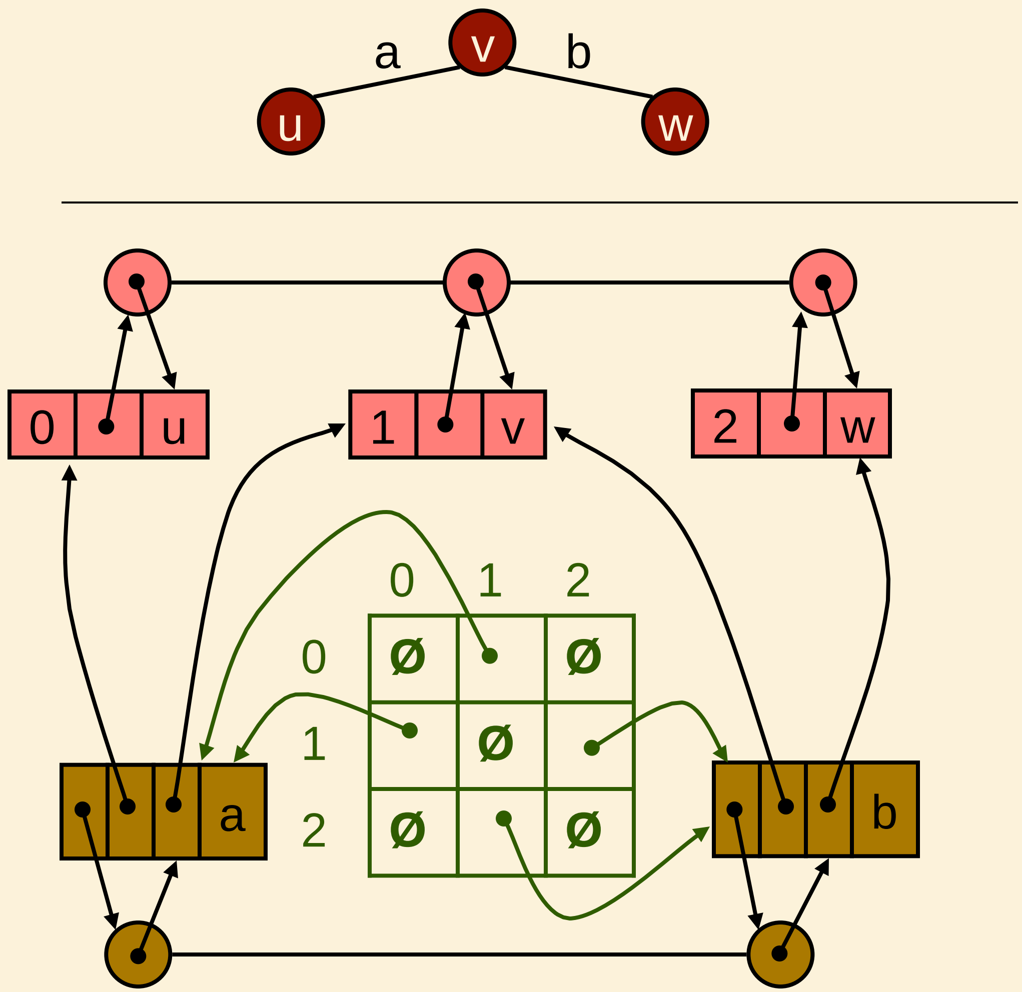
Adjacency List Structure

- Edge list structure
- Incidence sequence for each vertex
  - sequence of references to edge objects of incident edges
- Augmented edge objects
  - references to associated positions in incidence sequences of end vertices



Adjacency Matrix Structure

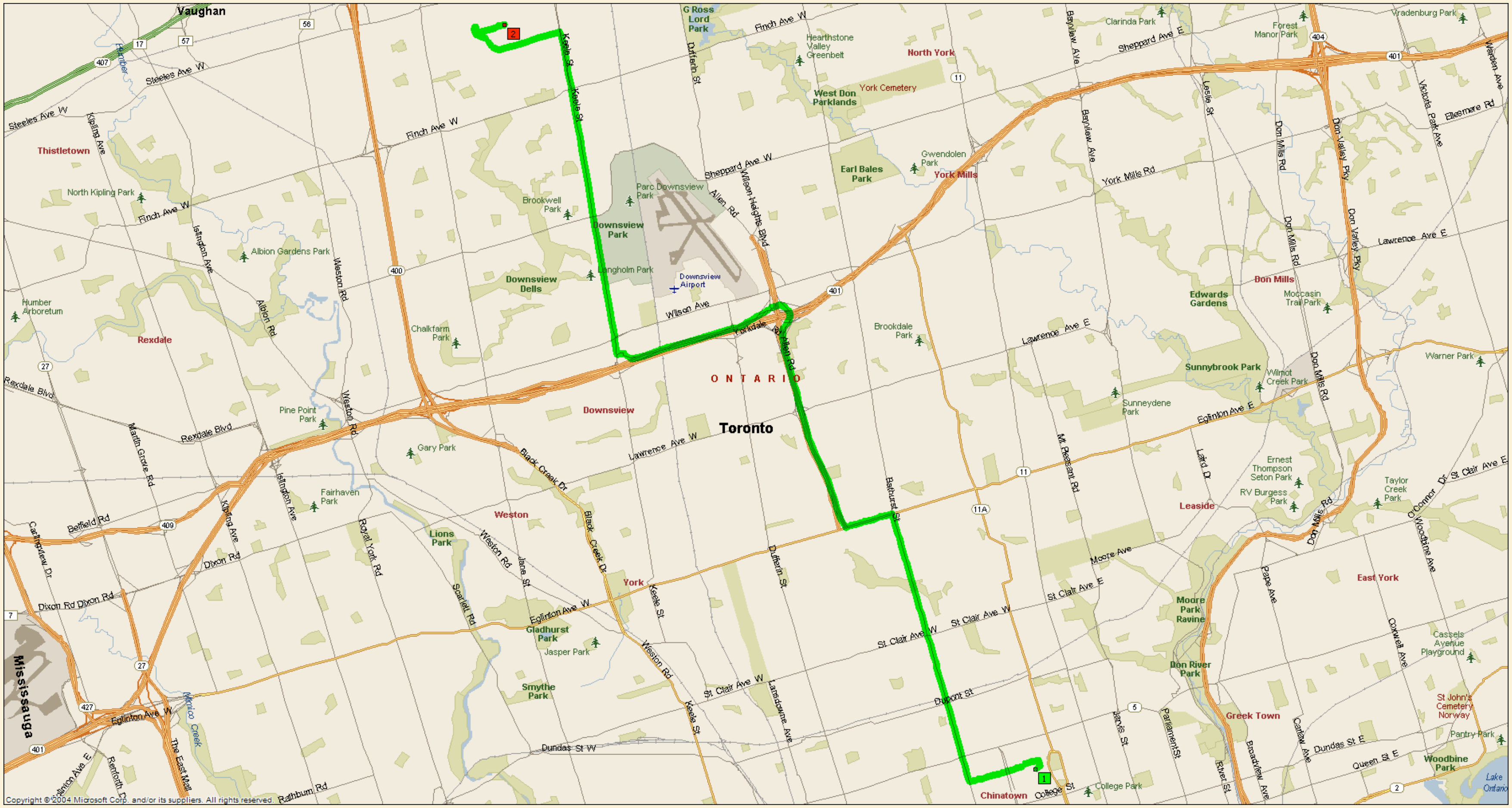
- Edge list structure
- Augmented vertex objects
  - Integer key (index) associated with vertex
- 2D-array adjacency array
  - Reference to edge object for adjacent vertices
  - Null for non-nonadjacent vertices



Asymptotic Performance (assuming collections V and E represented as doubly-linked lists)

◆ $\lvert V \rvert$ vertices, $\lvert E \rvert$ edges ◆ no parallel edges ◆ no self-loops ◆ Bounds are "big-Oh"	Edge List	Adjacency List	Adjacency Matrix
Space	$\lvert V\rvert+\lvert E\rvert$	$\lvert V\rvert+\lvert E\rvert$	$\lvert V\rvert^2$
incidentEdges(v)	$\lvert E\rvert$	$\deg(v)$	$\lvert V\rvert$
areAdjacent (v, w)	$\lvert E\rvert$	$\min(\deg(v), \deg(w))$	1
insertVertex(o)	1	1	$\lvert V\rvert^2$
insertEdge(v, w, o)	1	1	1
removeVertex(v)	$\lvert E\rvert$	$\deg(v)$	$\lvert V\rvert^2$
removeEdge(e)	1	1	1

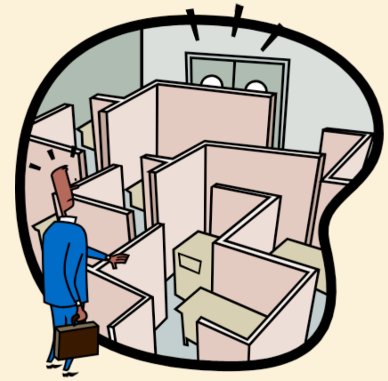
Graph Search Algorithms



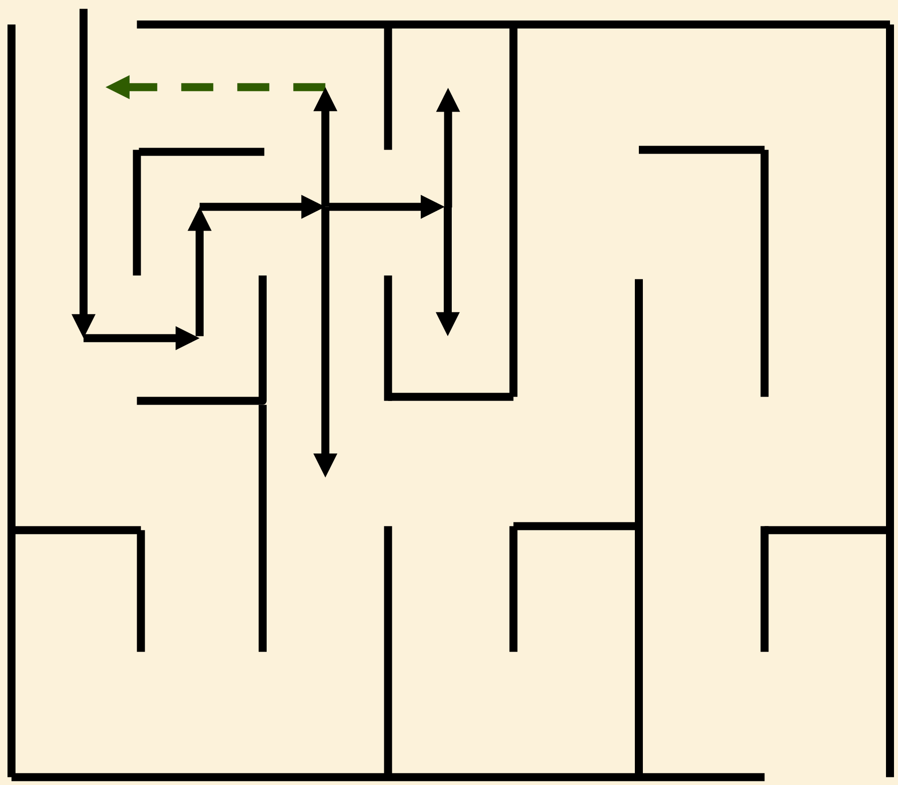
Depth First Search (DFS)

- 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*
  - Topological sorts
  - Detection of cycles
  - Extraction of strongly connected components

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)

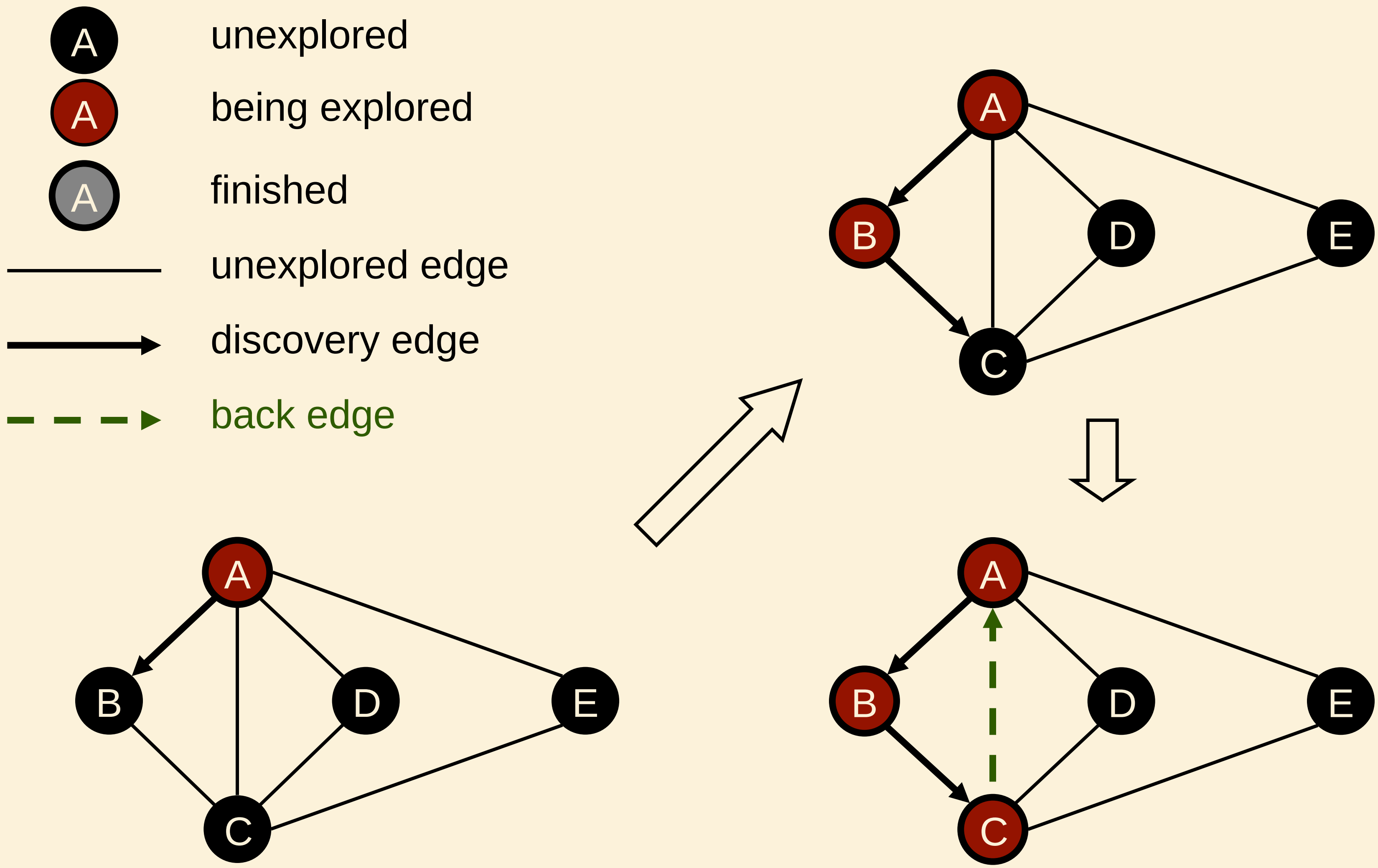


Depth-First Search

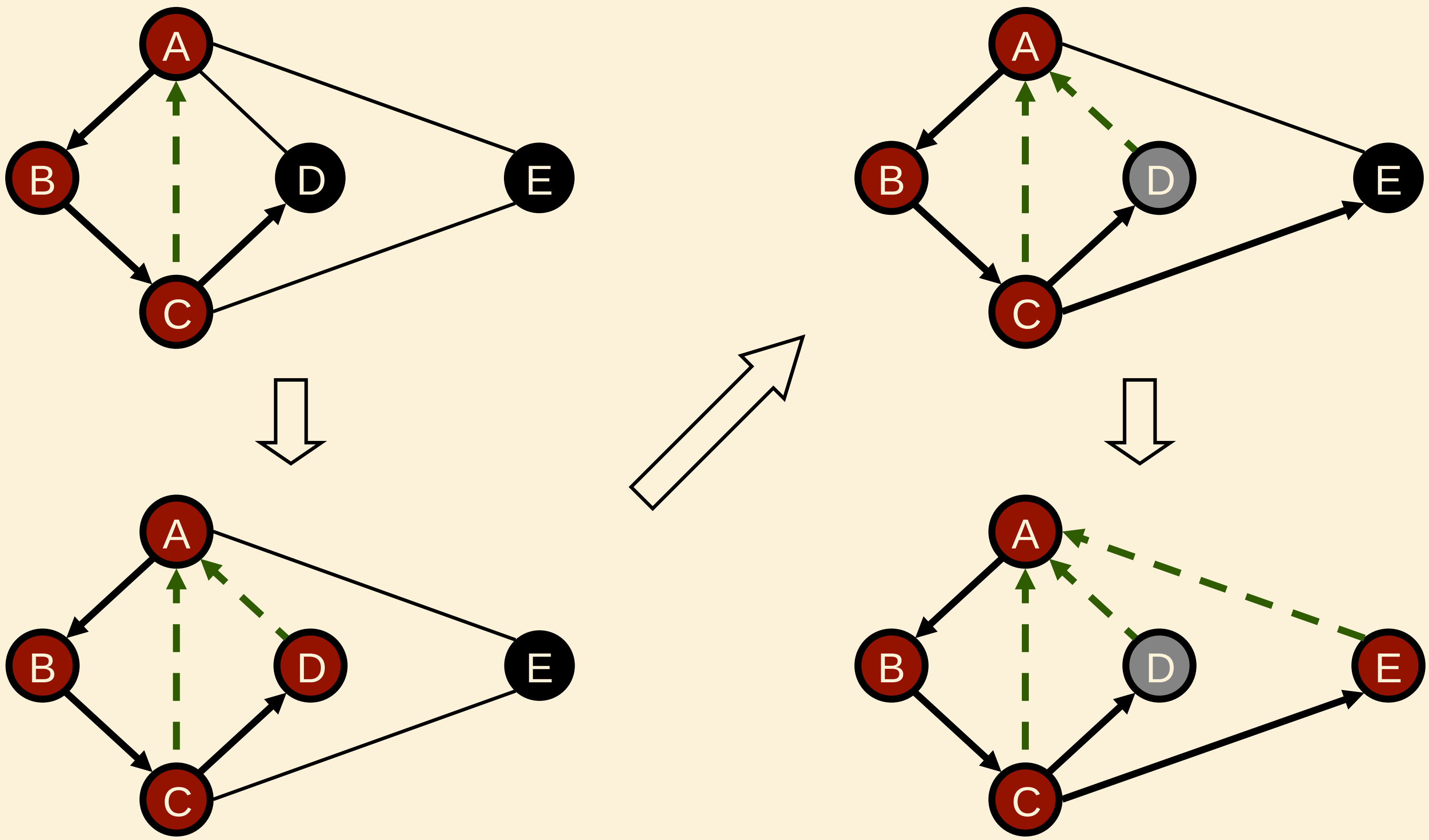
Input: Graph $G = (V, E)$ (directed or undirected)

- 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 (found everything reachable from it).

DFS Example on Undirected Graph



Example (cont.)



DFS Algorithm Pattern

DFS(G)

Precondition: G is a graph

Postcondition: all vertices in G have been visited

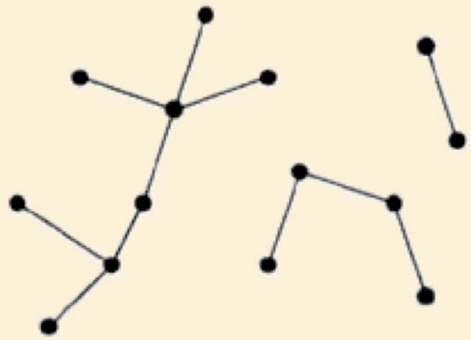
for each vertex u $\in V[G]$

color[u] = BLACK //initialize vertex

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

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

DFS-Visit(u)



DFS Algorithm Pattern

DFS-Visit (u)

Precondition: vertex u is undiscovered

Postcondition: all vertices reachable from u have been processed

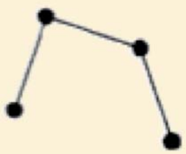
colour[u] $\leftarrow$ RED

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

if color[v] = BLACK

DFS-Visit(v)

$colour[u] \leftarrow GRAY$



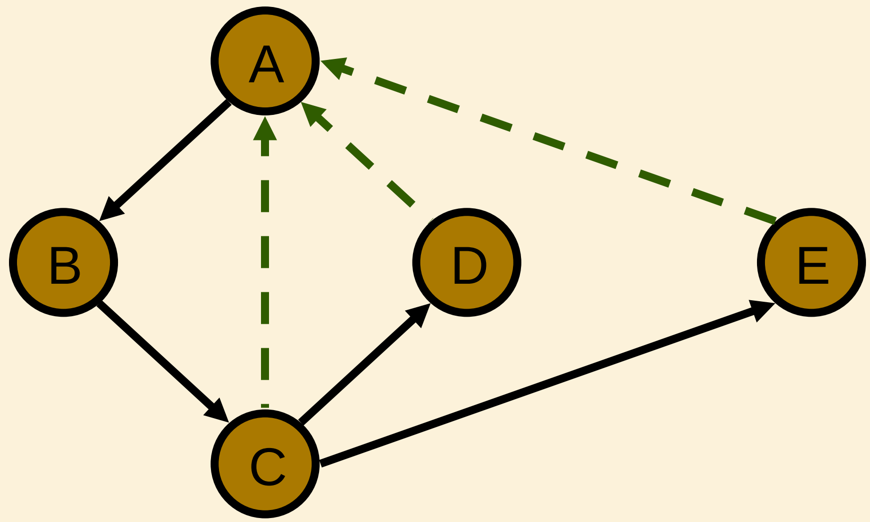
Properties of DFS

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 $u \in V[G]$

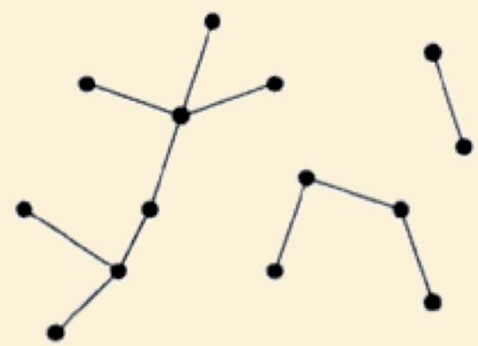
    color[u] = BLACK //initialize vertex

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

colour[u] $\leftarrow$ RED

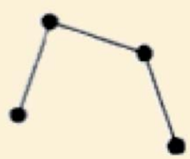
for each $v \in$ Adj[u] //explore edge (u,v)

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 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 $u \in V[G]$

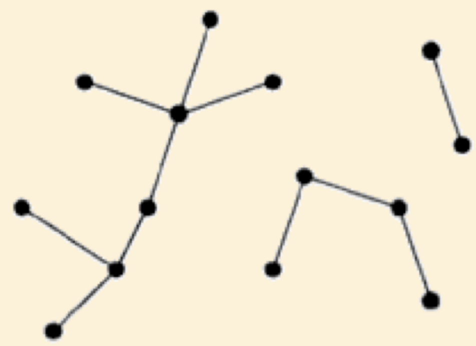
color[u] = BLACK //initialize vertex

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

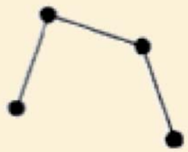
Postcondition: all vertices reachable from *u* have been processed

$$\text{colour}[u] \leftarrow \text{RED}$$
$$\text{time} \leftarrow \text{time} + 1$$
$$d[u] \leftarrow \text{time}$$

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

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

    DFS-Visit(v)



$$colour[u] \leftarrow GRAY$$
$$\text{time} \leftarrow \text{time} + 1$$
$$f[u] \leftarrow \text{time}$$

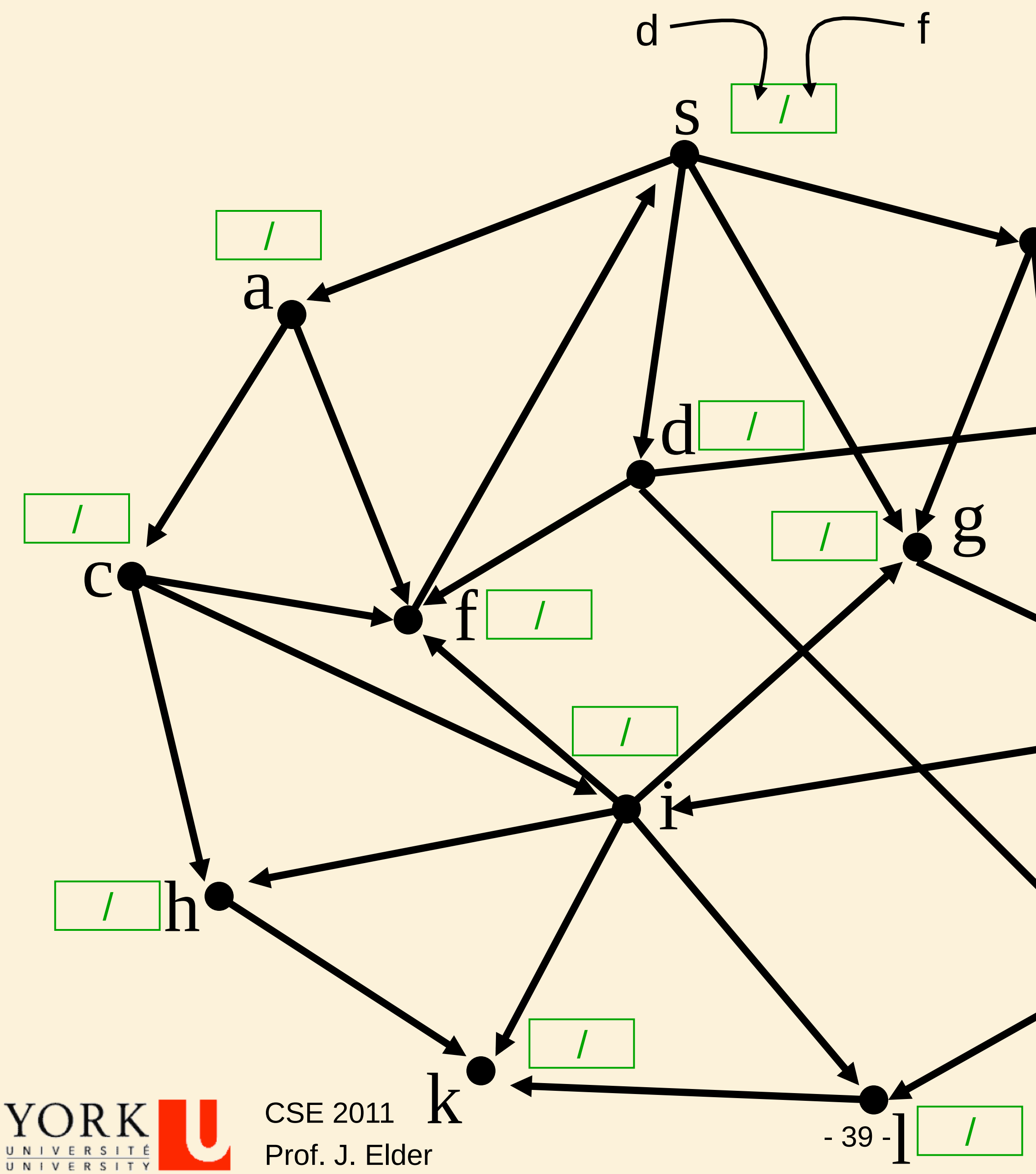
Other Variants of Depth-First Search

- 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 (see textbook)
    - **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

Example DFS on Directed Graph

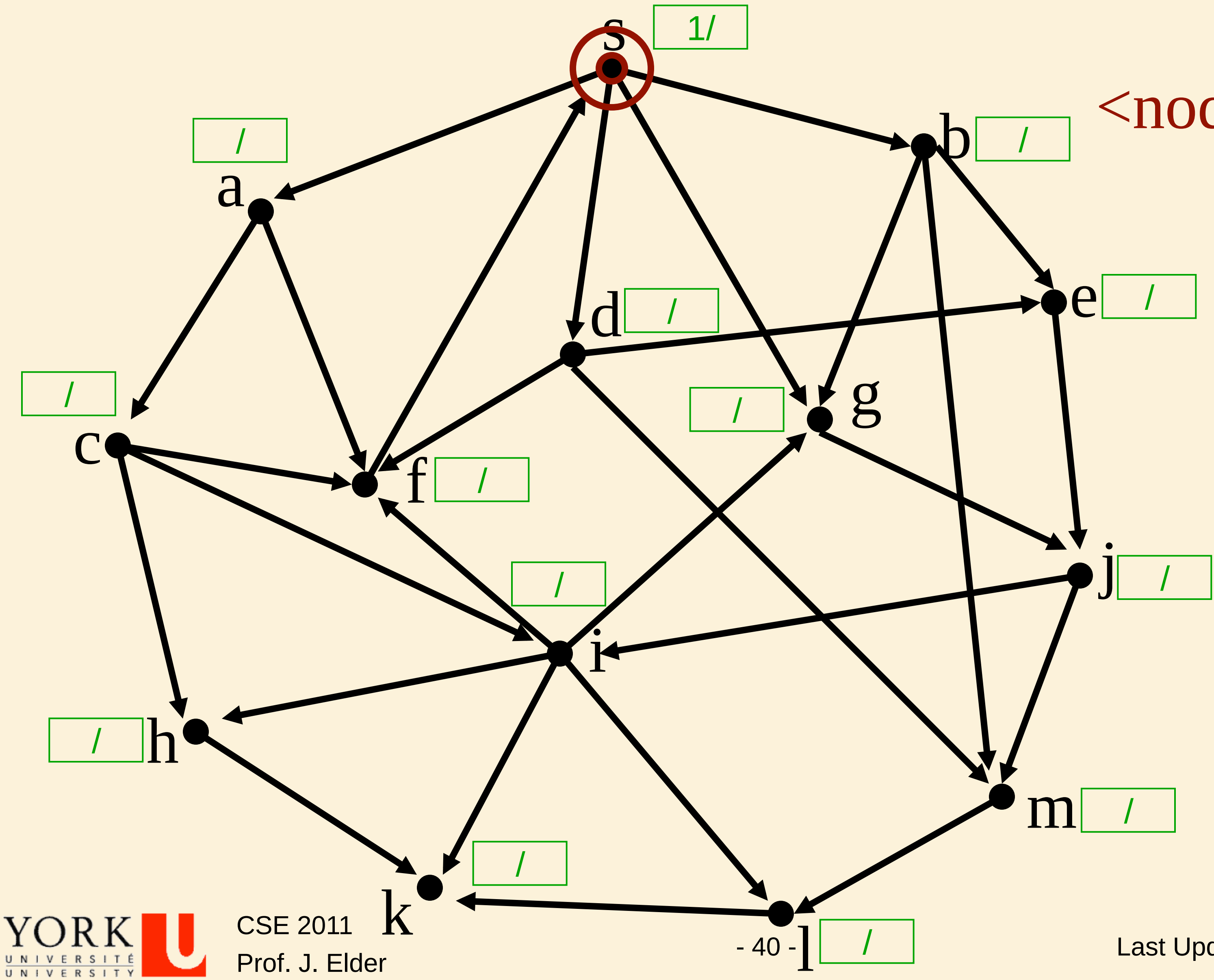
DFS

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



Found
Not Handled
Stack
<node,# edges>

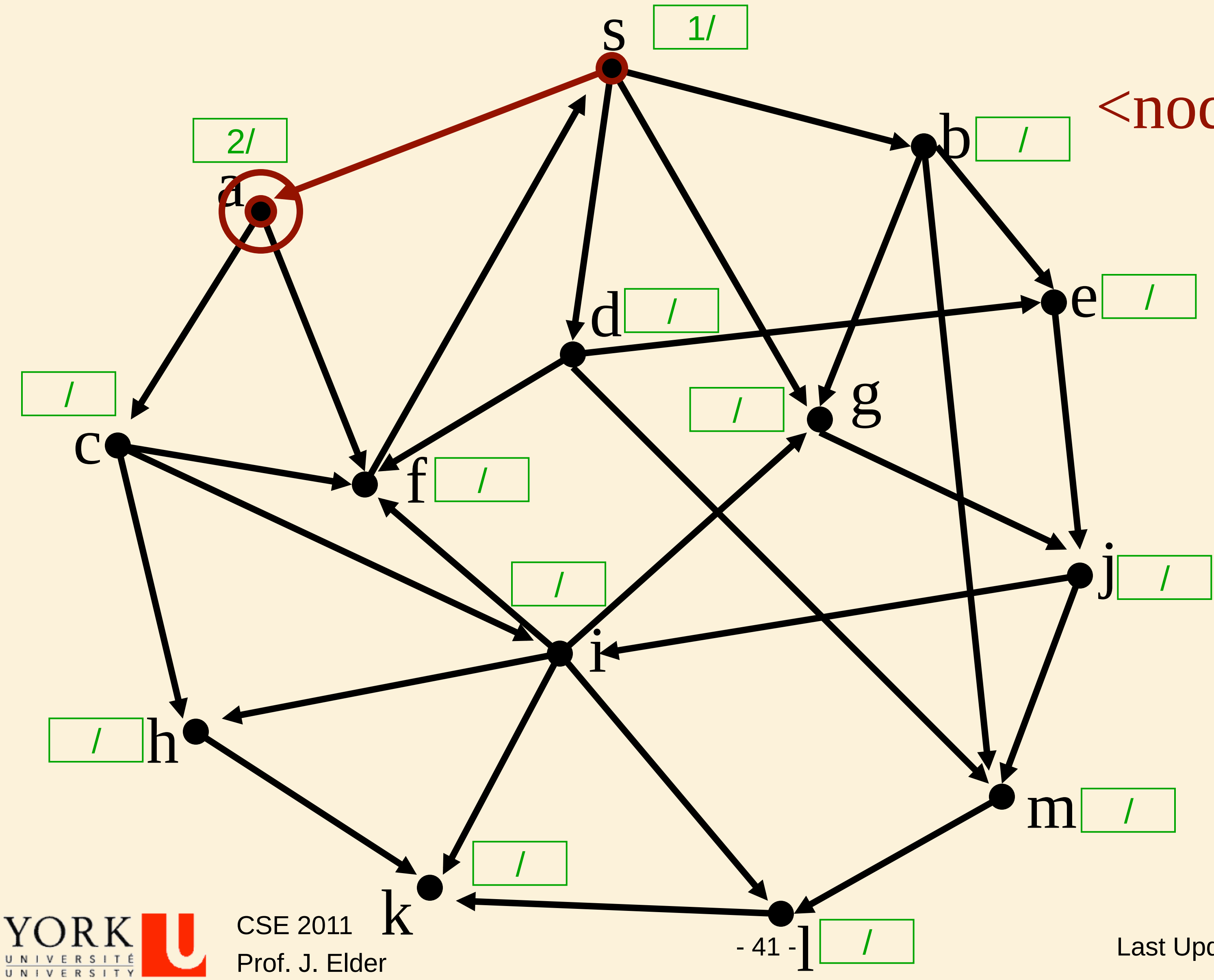
DFS



Found
Not Handled
Stack
<node,# edges>

s,0

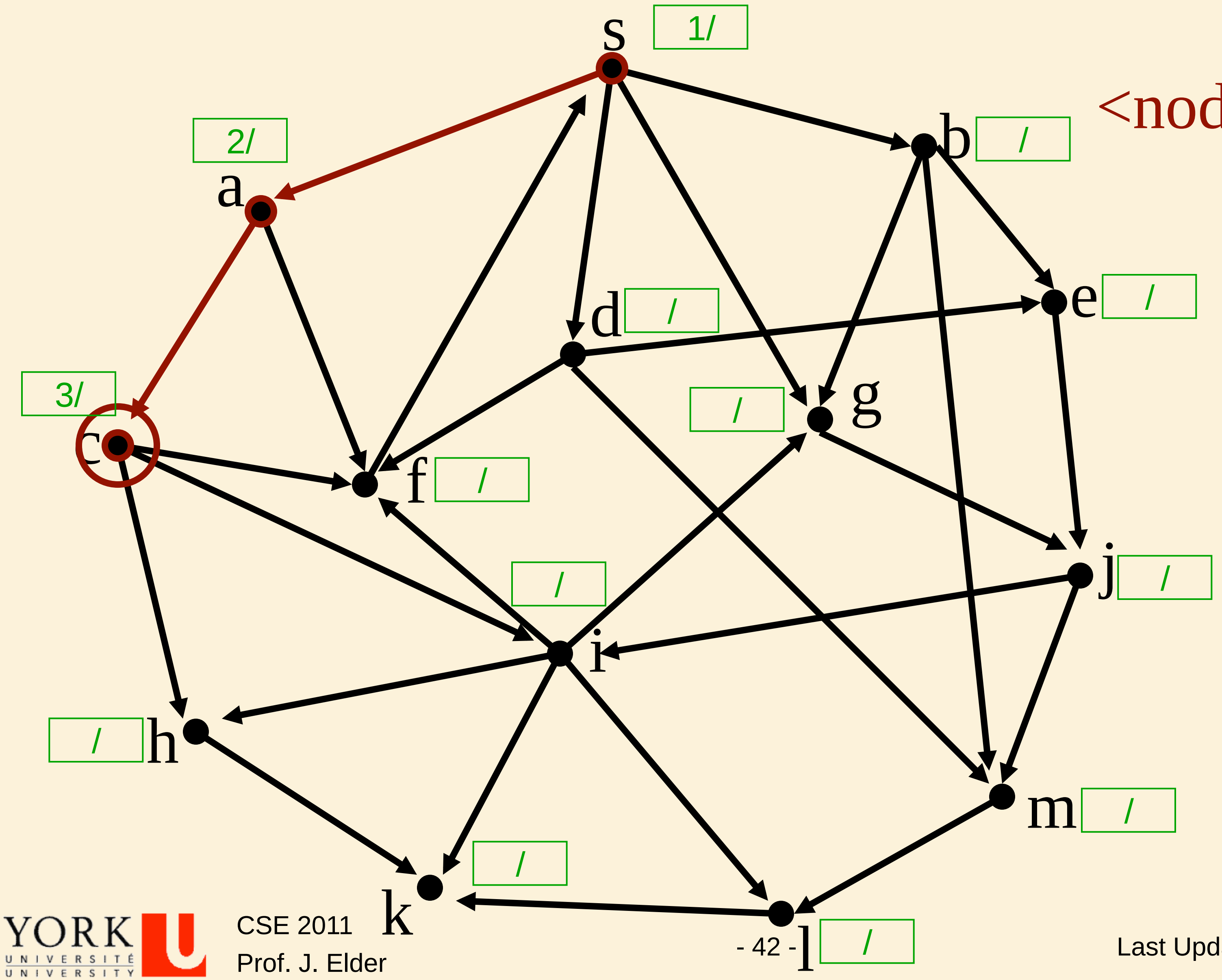
DFS



Found
Not Handled
Stack
<node,# edges>

a,0
s,1

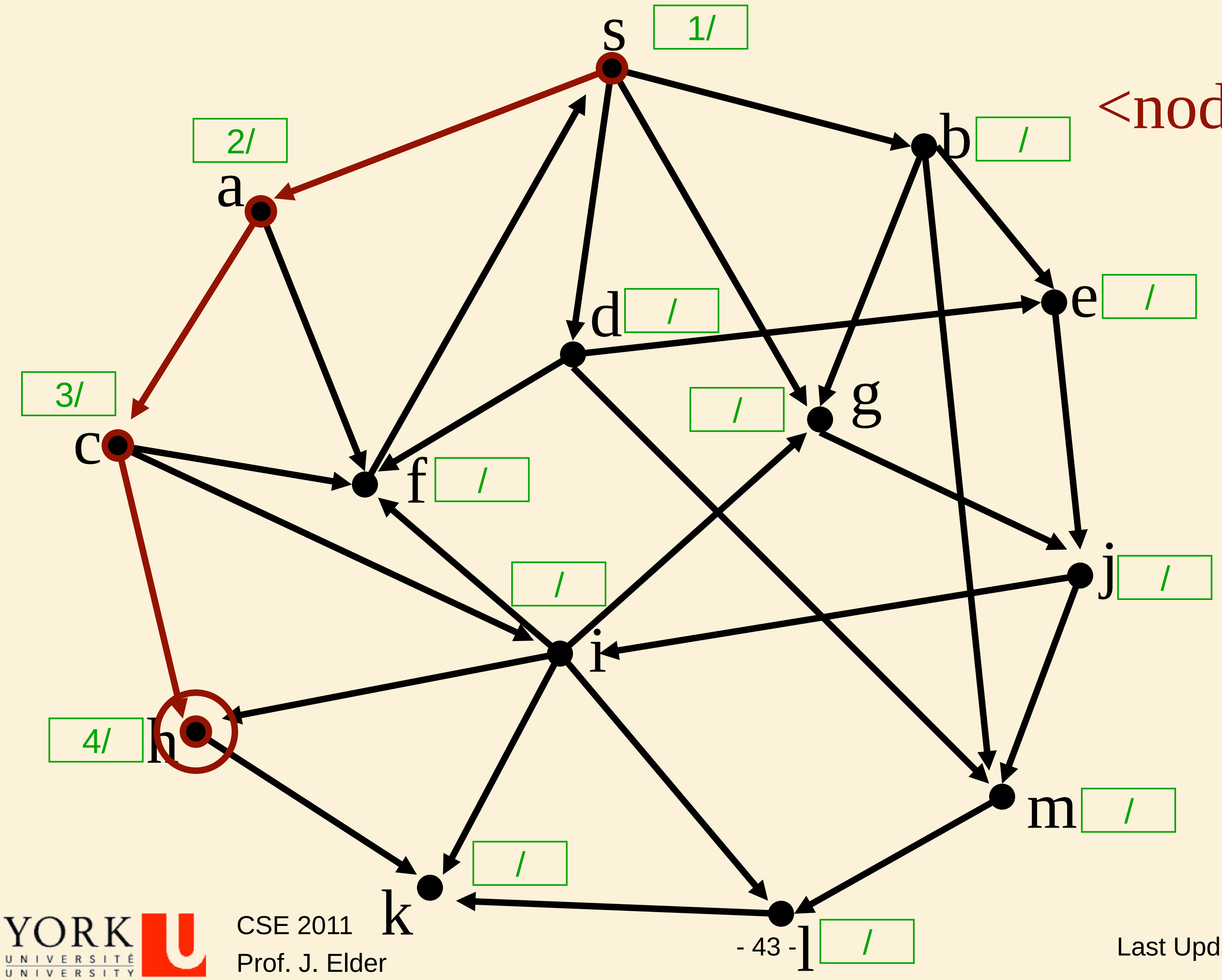
DFS



Found
Not Handled
Stack
<node,# edges>

c,0
a,1
s,1

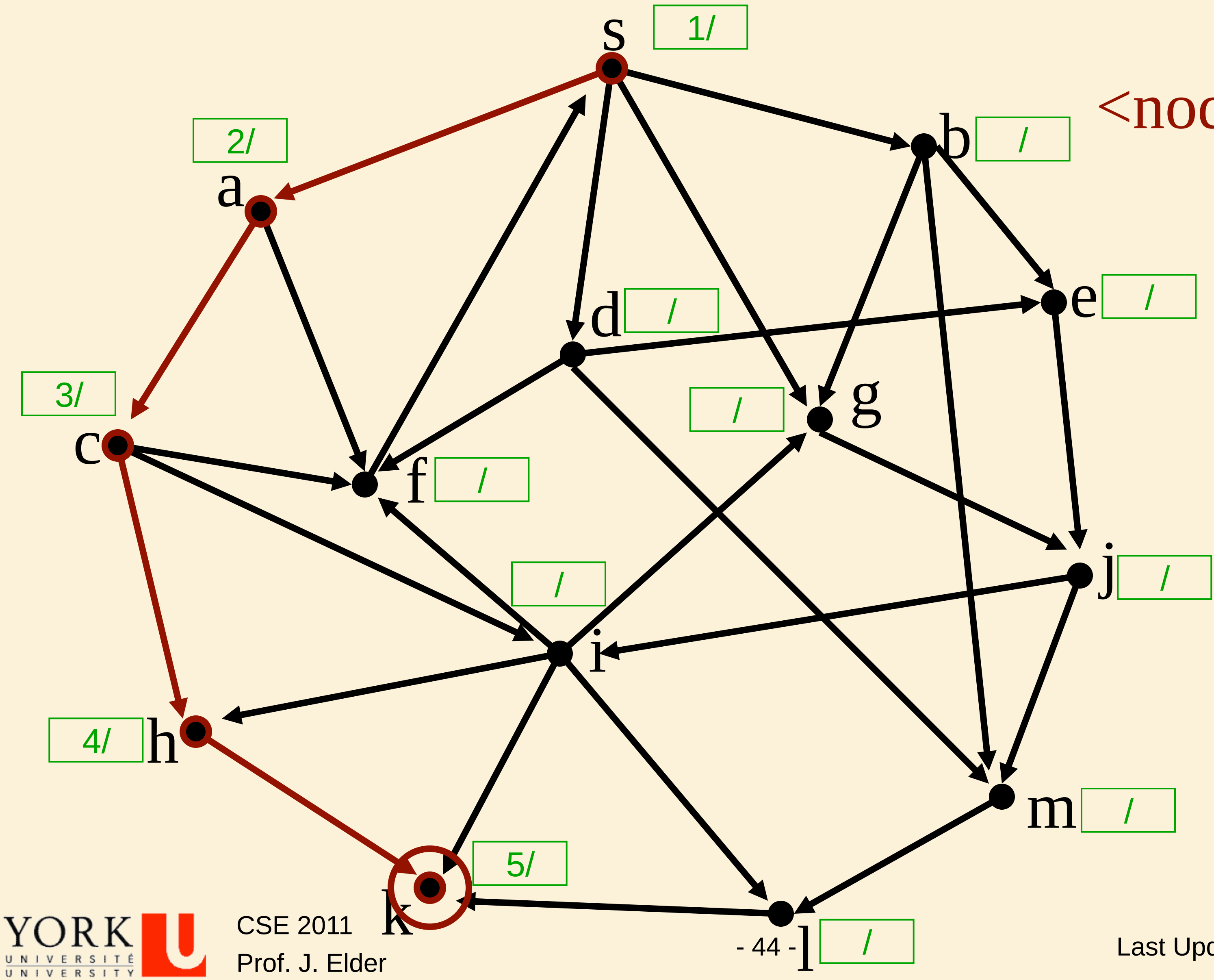
DFS



Found
Not Handled
Stack
<node,# edges>

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

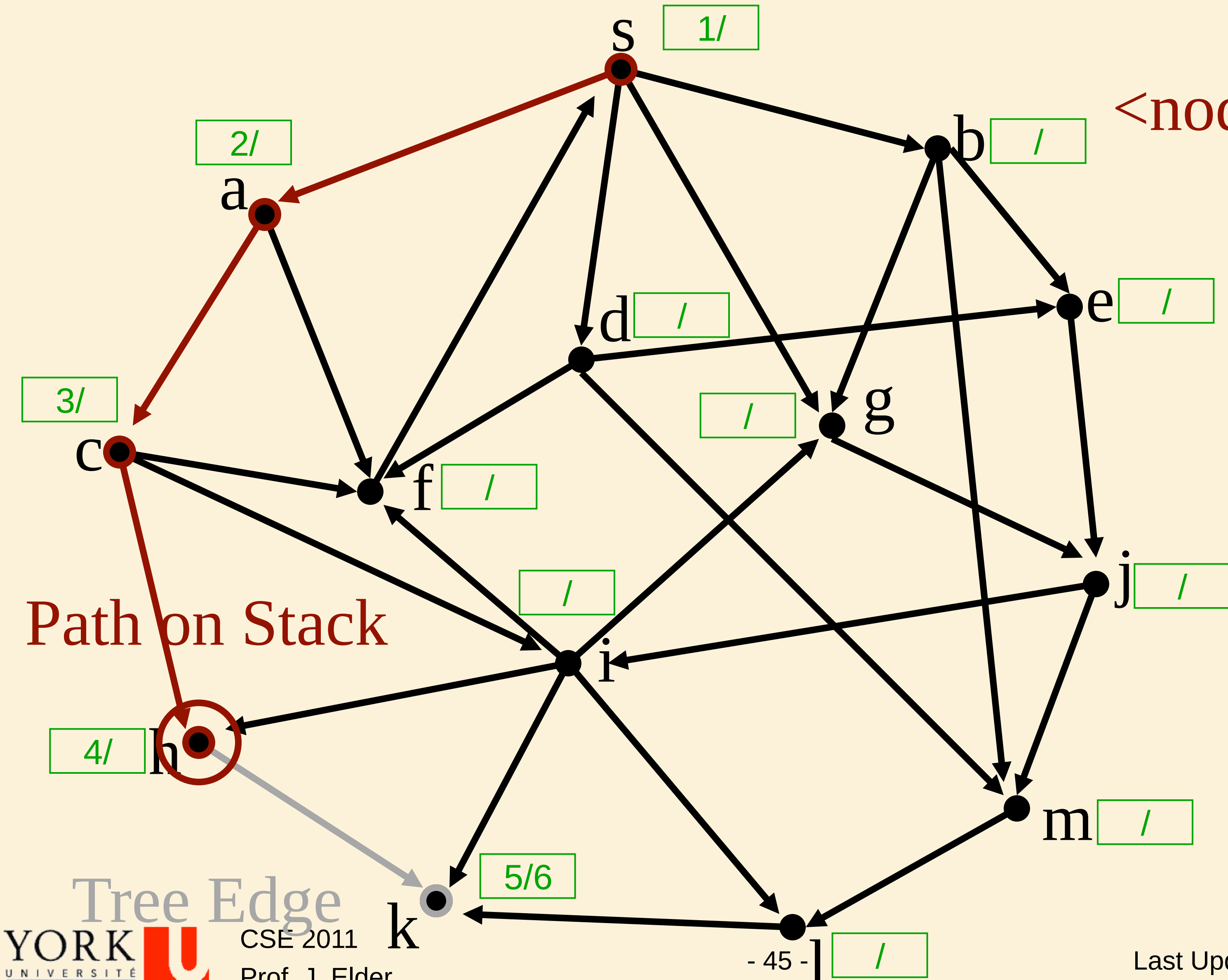
DFS



Found
Not Handled
Stack
<node,# edges>

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

DFS

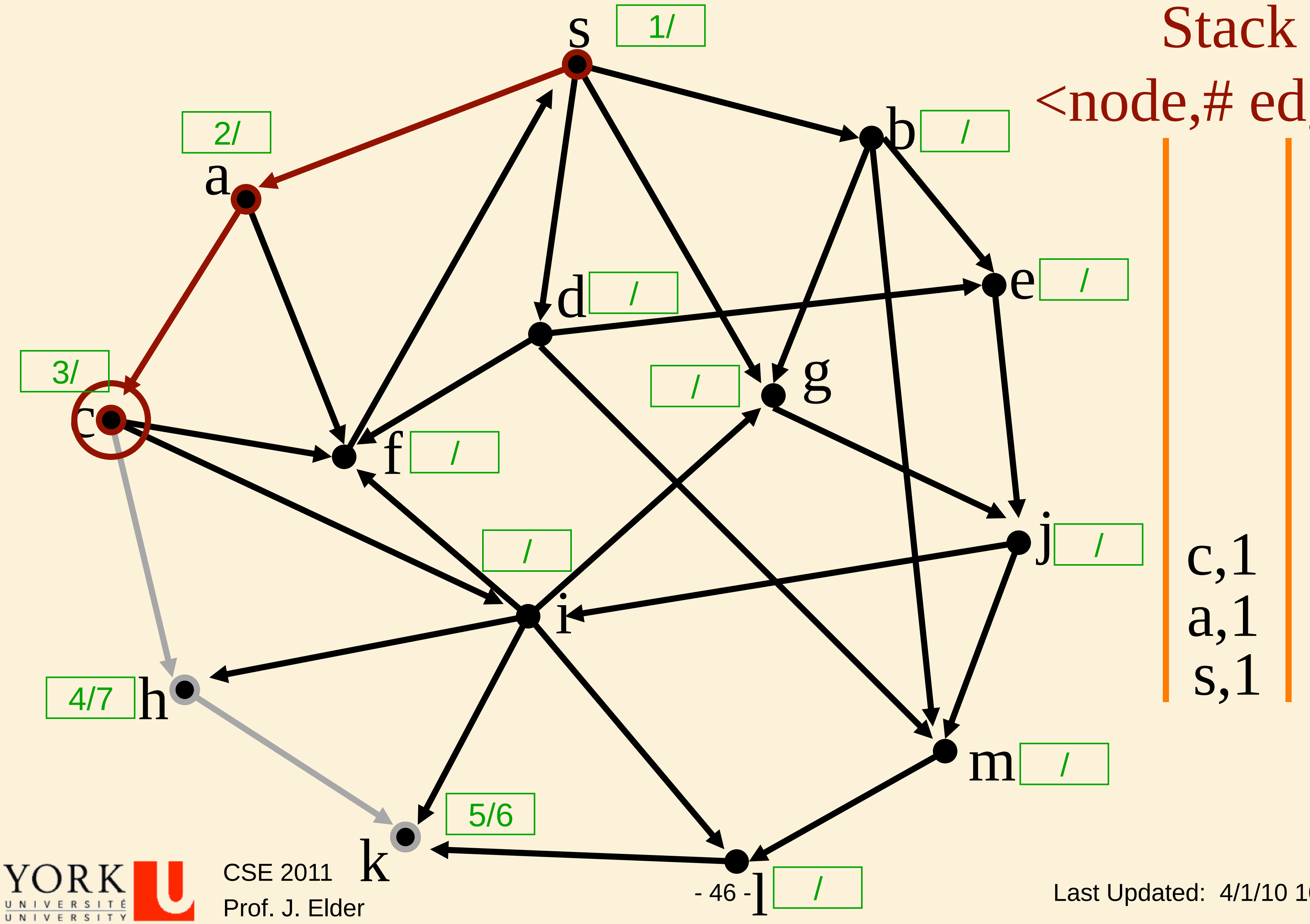


Found
Not Handled
Stack
<node,# edges>

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

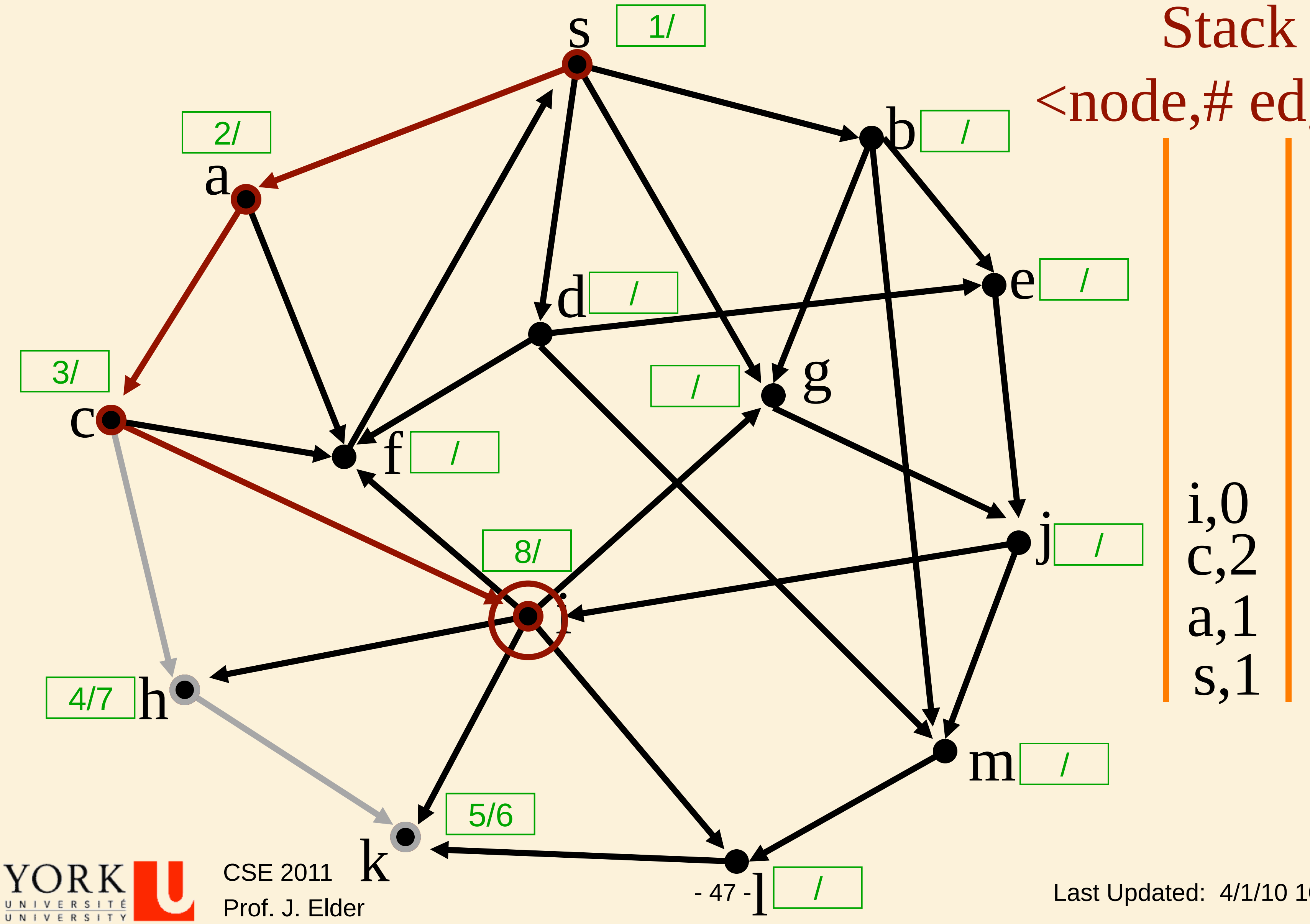
DFS

Found
Not Handled
Stack
<node,# edges>



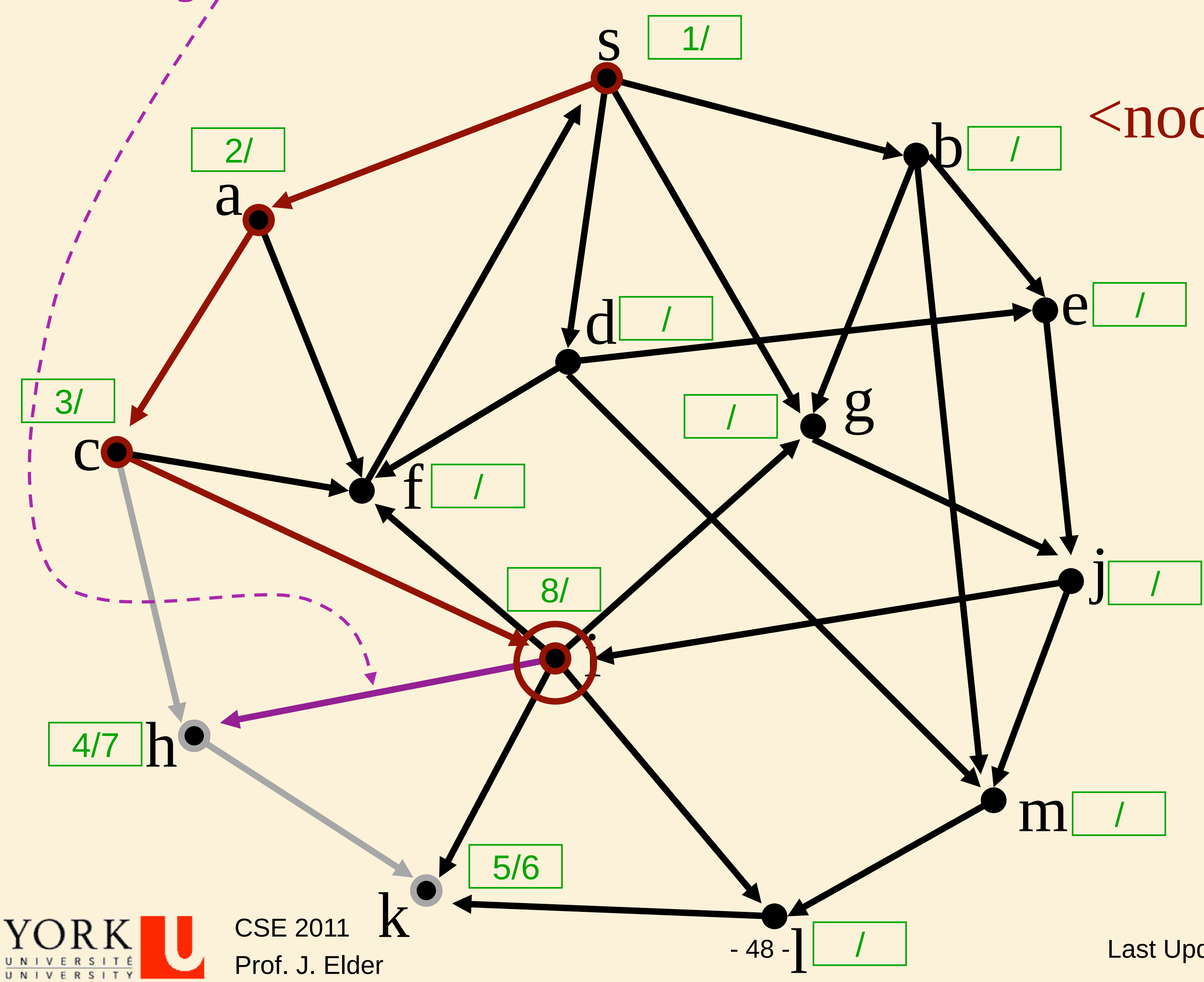
DFS

Found
Not Handled
Stack
<node,# edges>



DFS

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

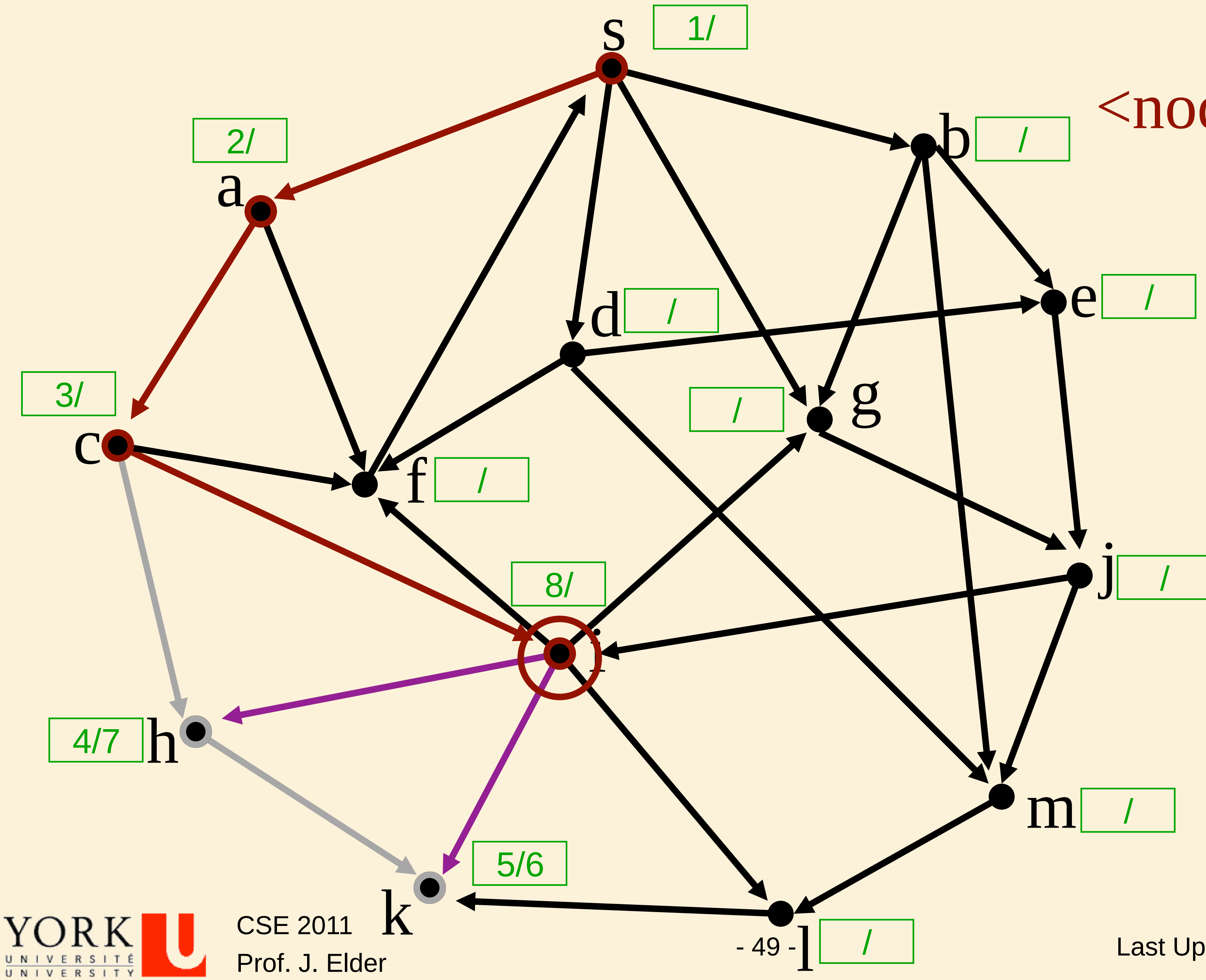


Found
Not Handled
Stack
<node,# edges>

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

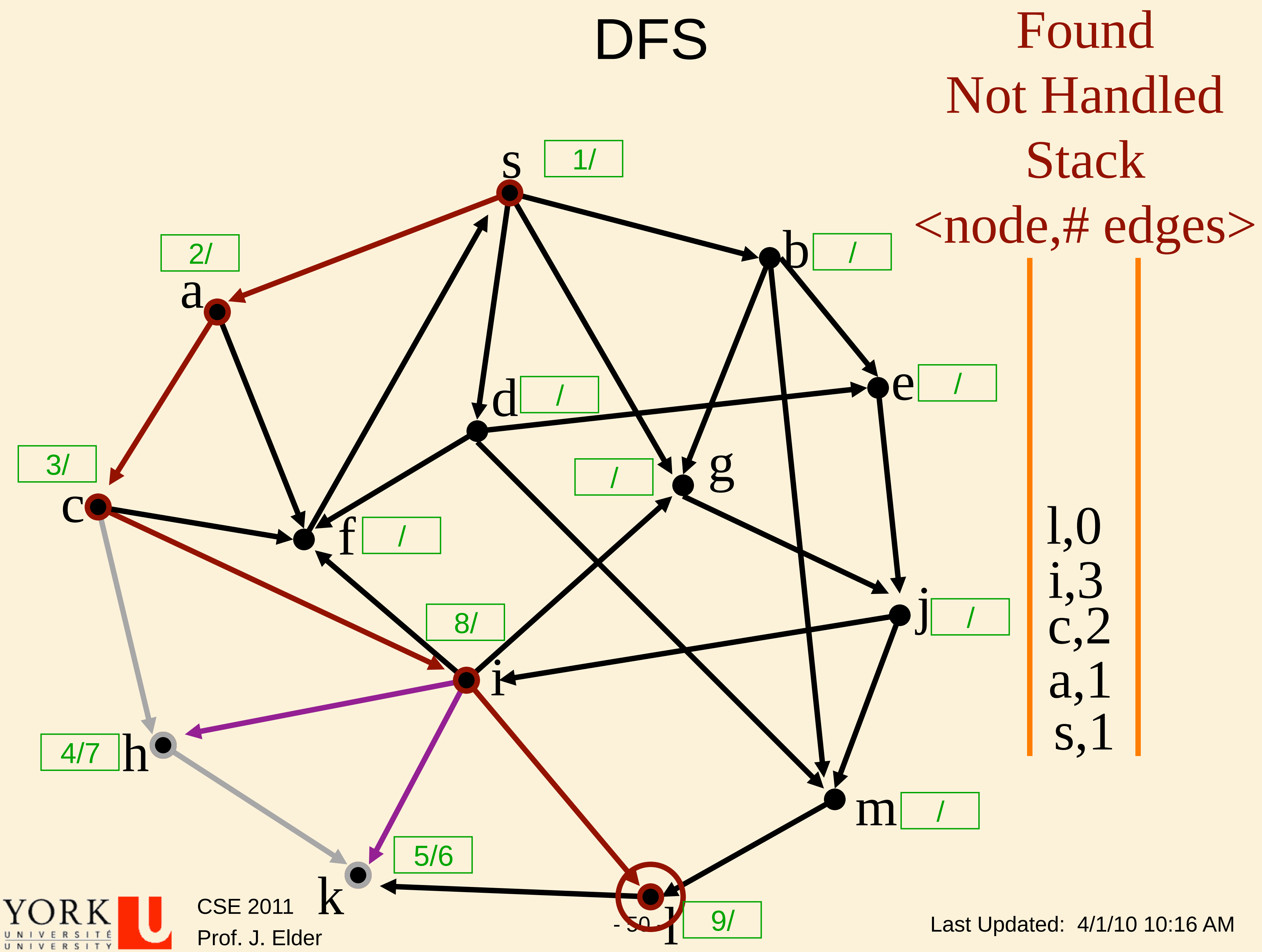
DFS

Found
Not Handled
Stack
<node,# edges>

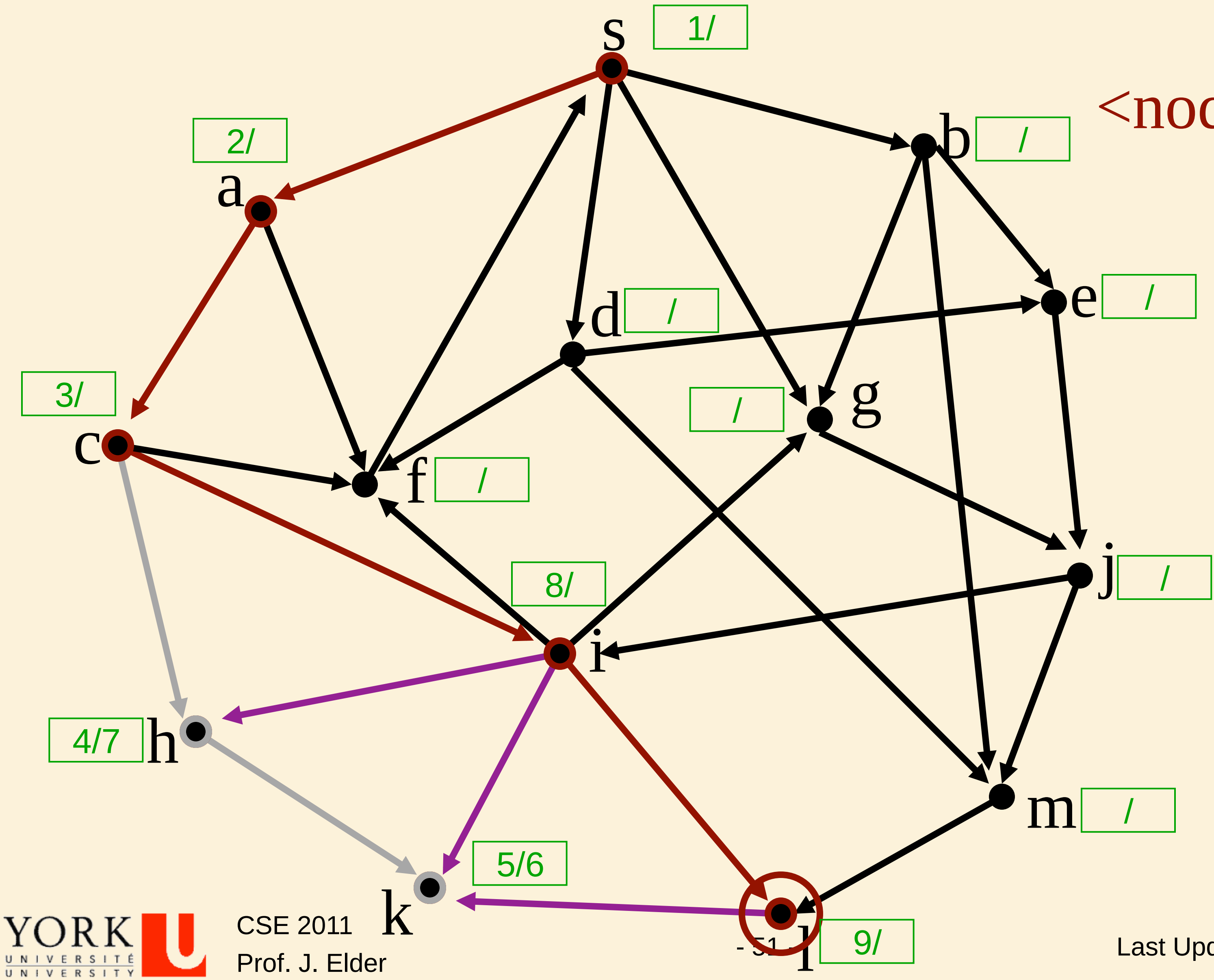


i,2
c,2
a,1
s,1

DFS


Found
Not Handled
Stack
<node,# edges>
s
1/
a
2/
b
/
c
3/
d
/
e
/
f
/
g
/
h
4/7
i
8/
j
/
k
5/6
l
9/
m
/
l,0
i,3
c,2
a,1
s,1

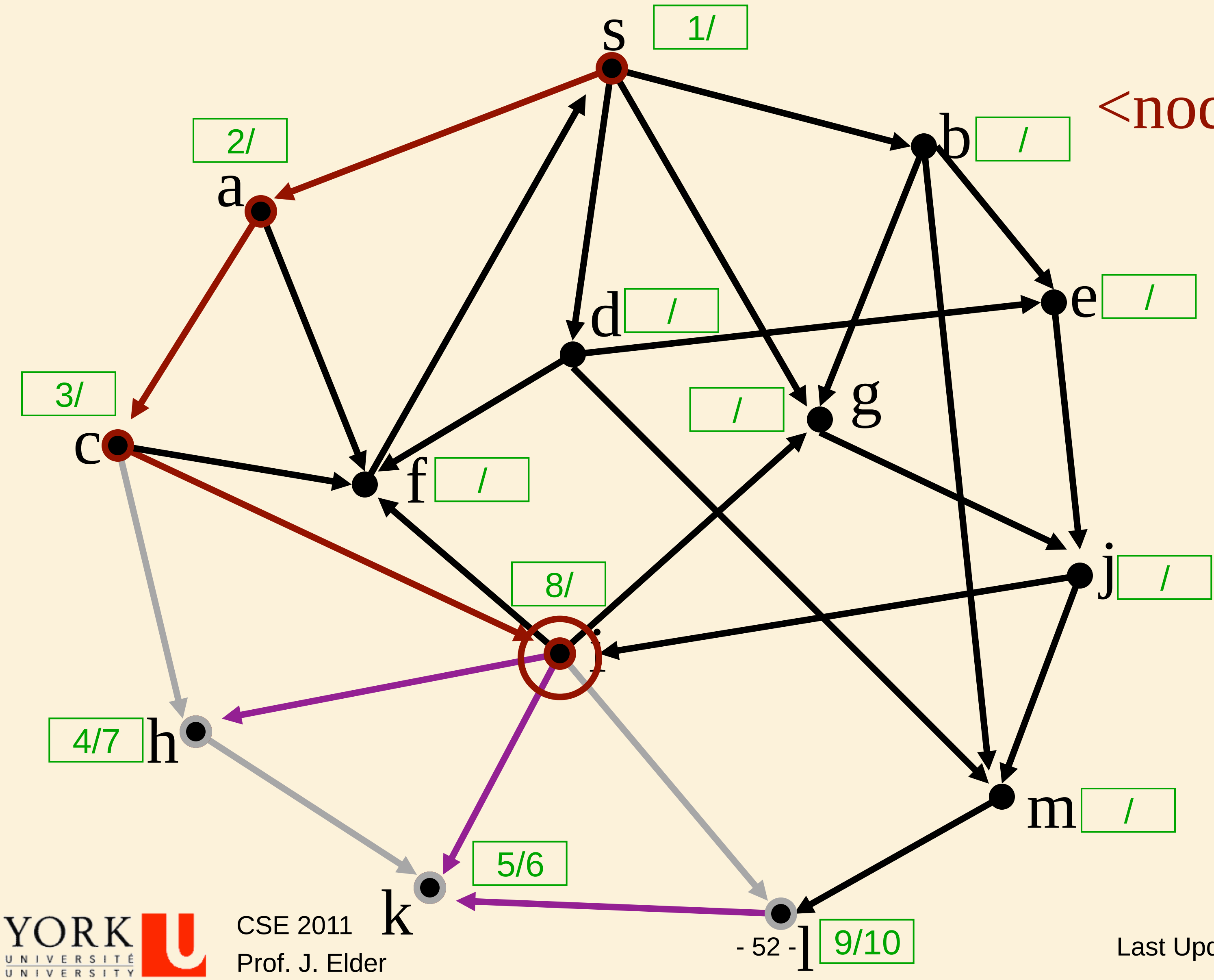
DFS



Found
Not Handled
Stack
<node,# edges>

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

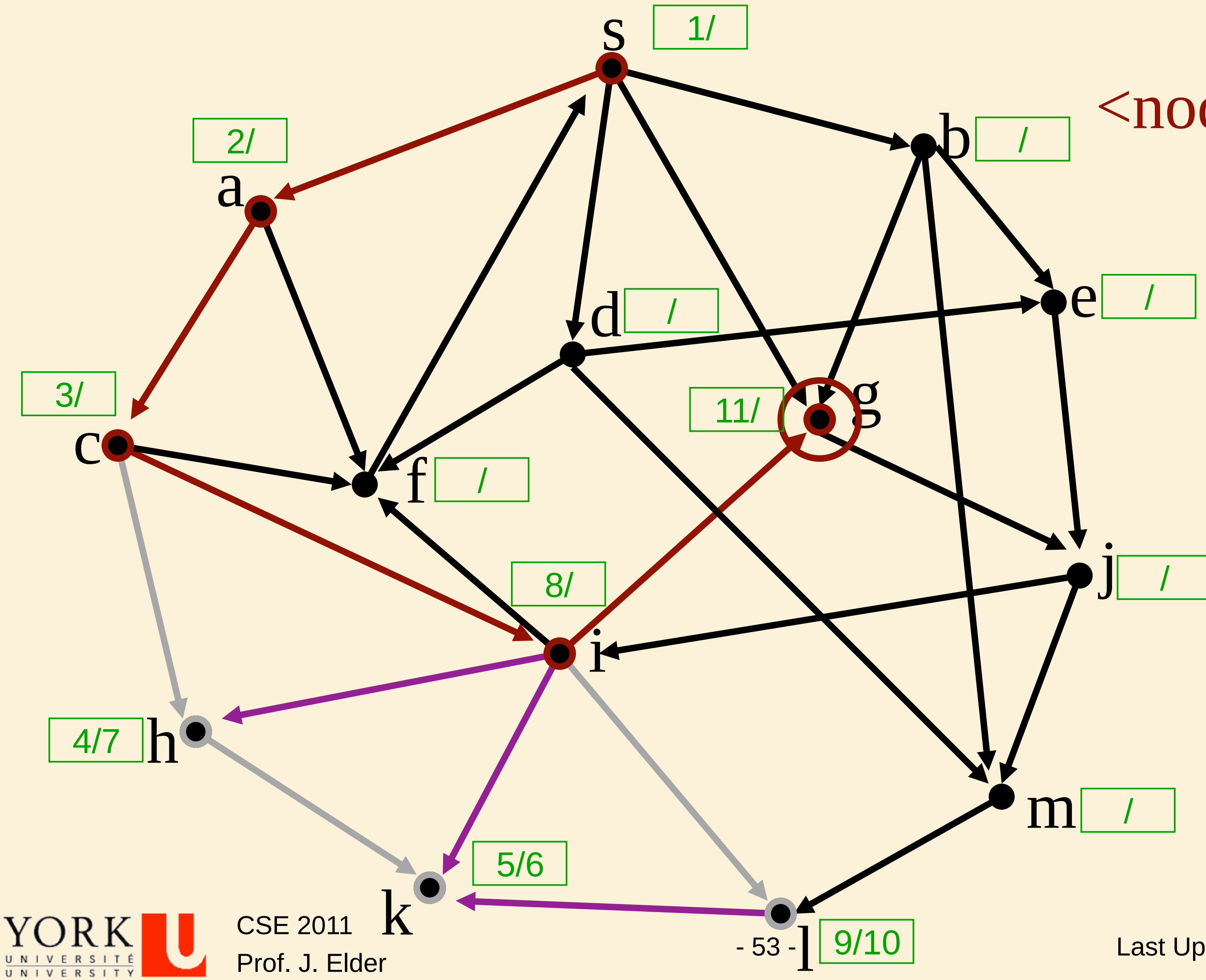
DFS



Found
Not Handled
Stack
<node,# edges>

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

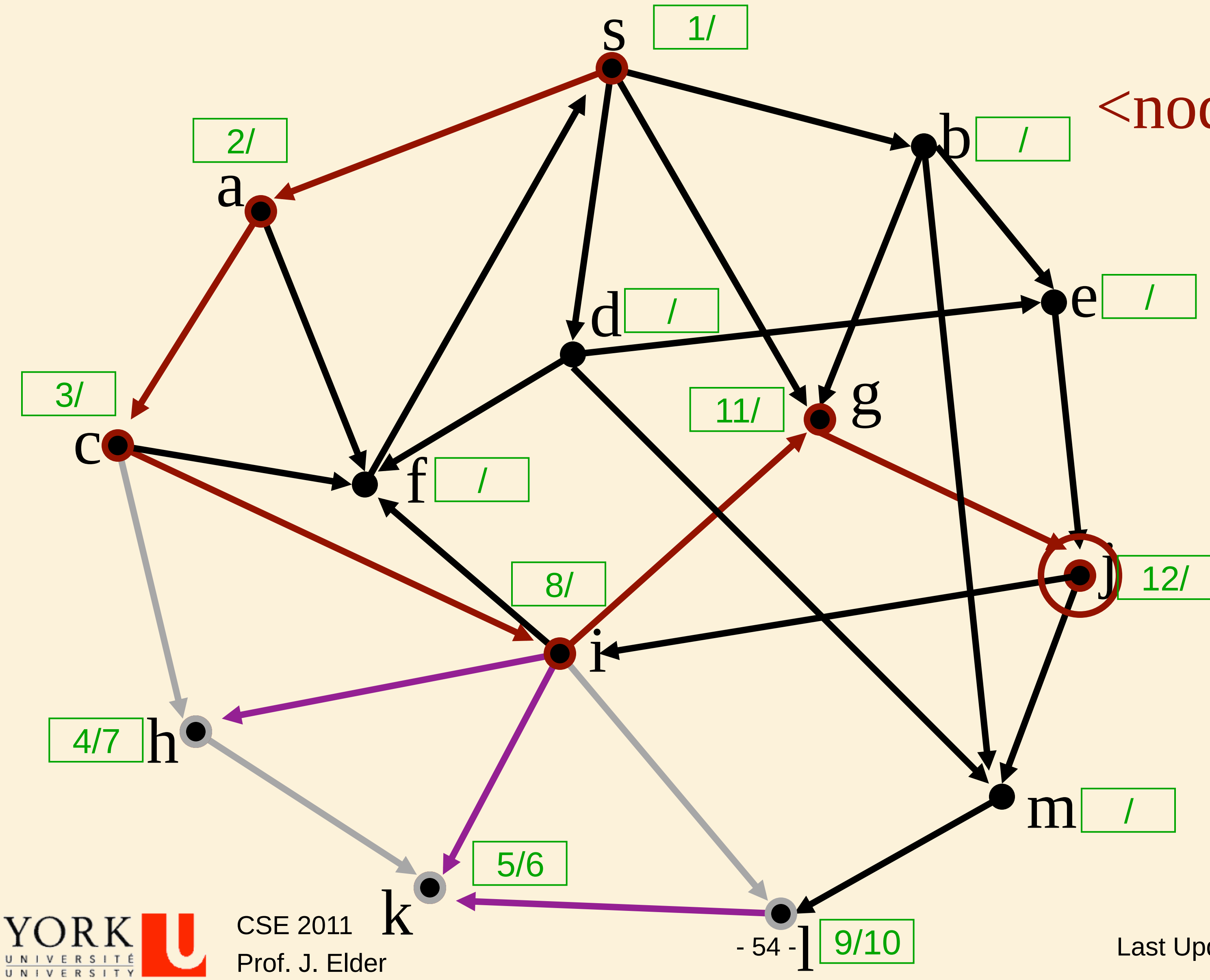
DFS



Found
Not Handled
Stack
<node,# edges>

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

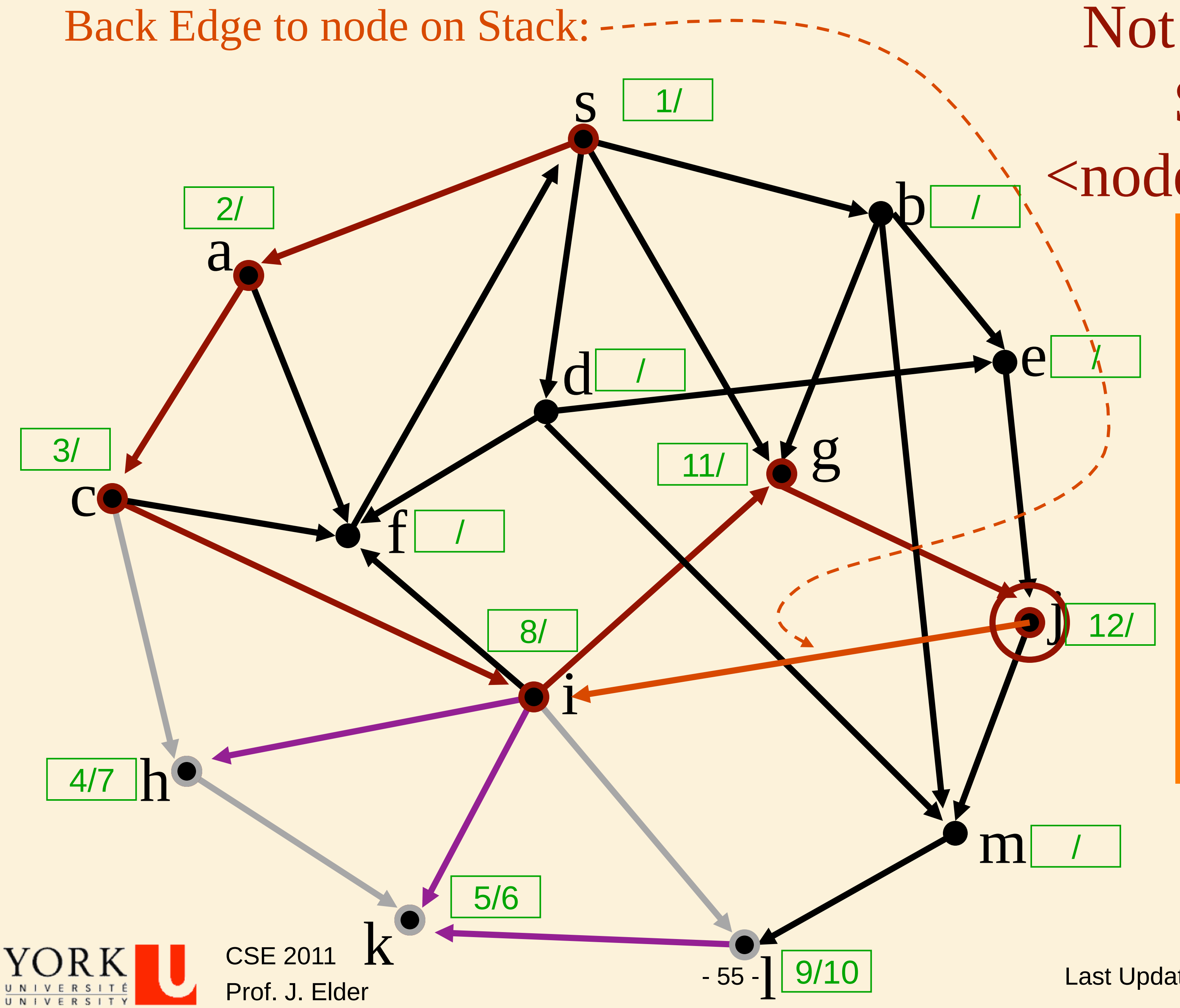
DFS



Found
Not Handled
Stack
<node,# edges>

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

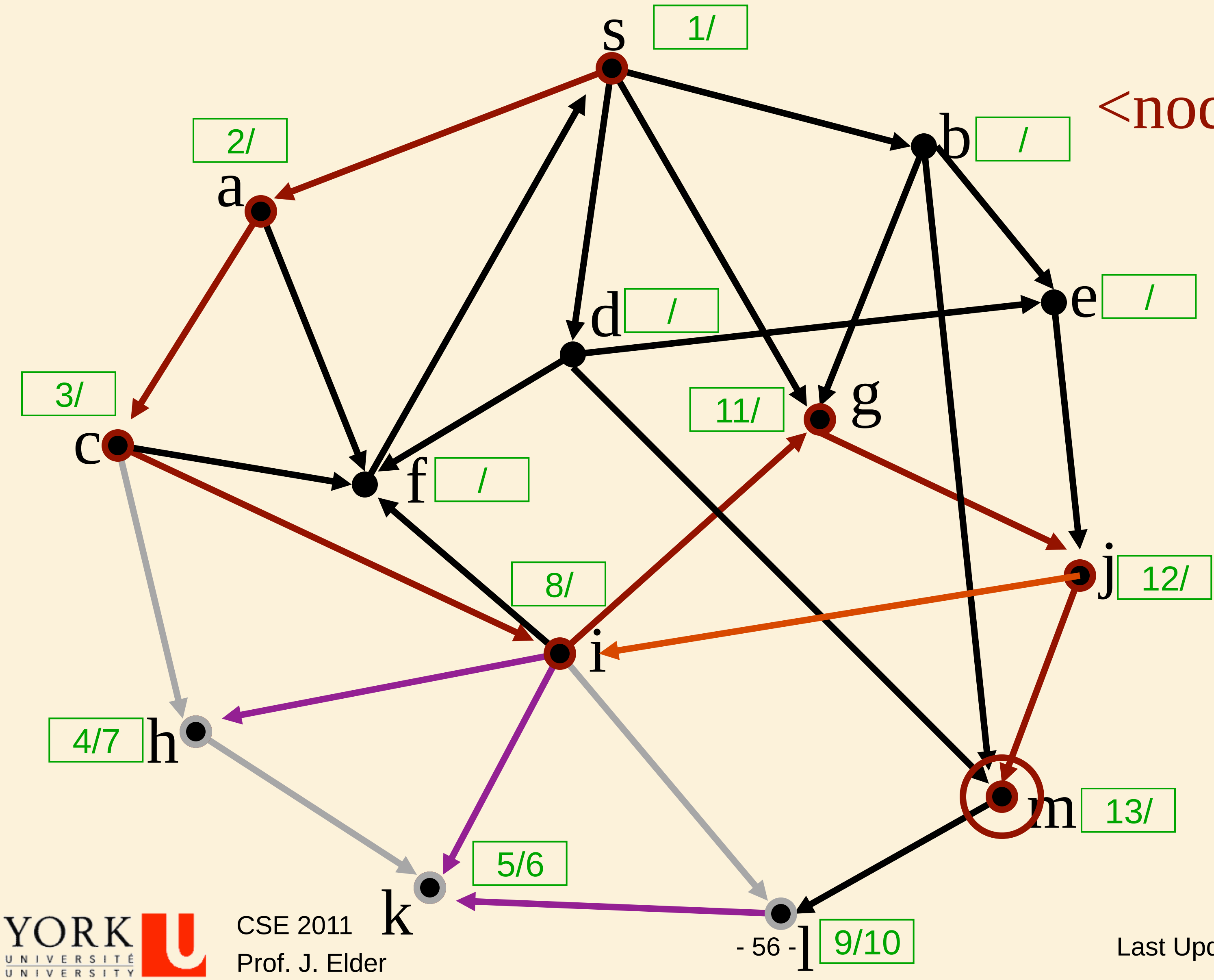
DFS



Found
Not Handled
Stack
<node,# edges>

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

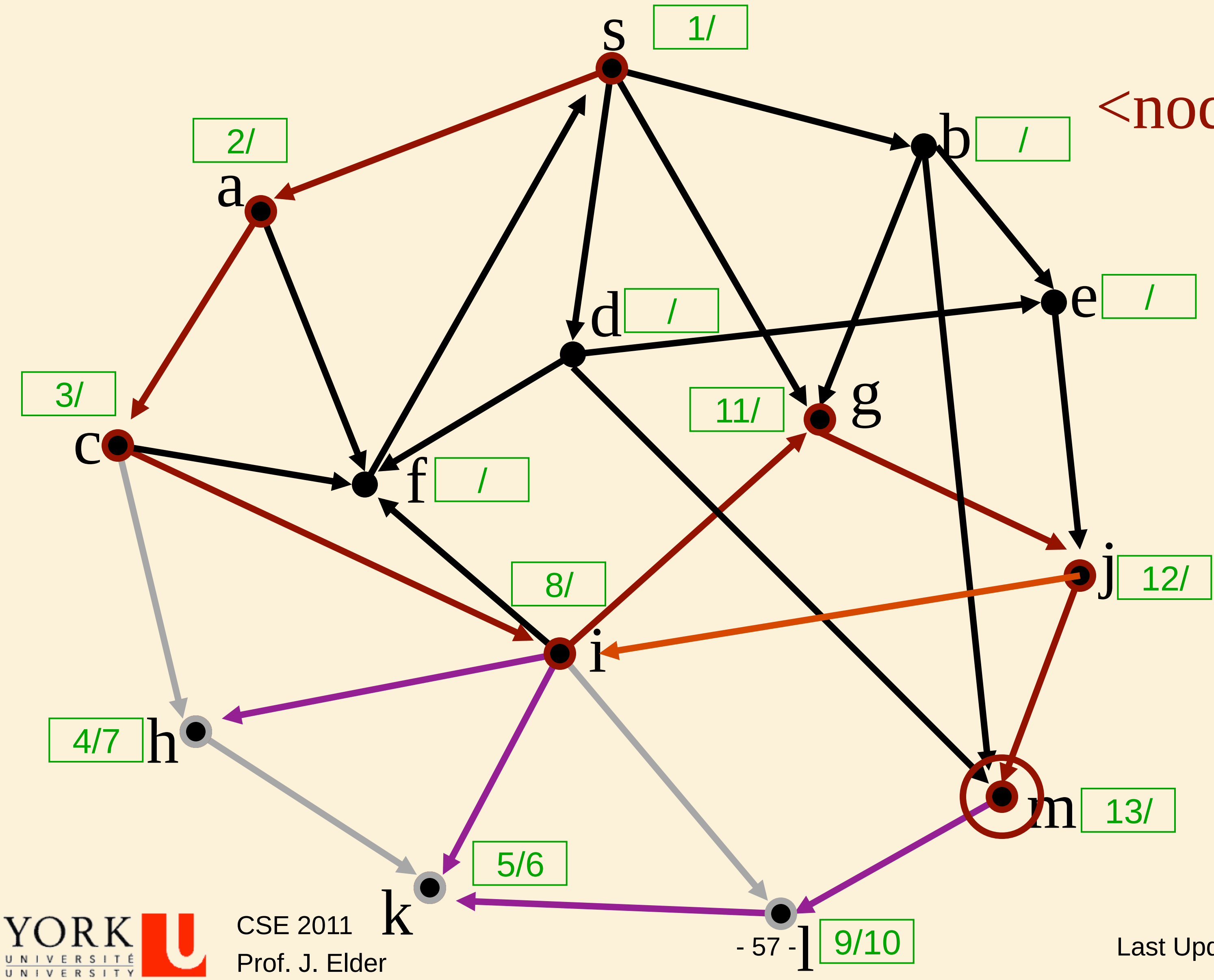
DFS



Found
Not Handled
Stack
<node,# edges>

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

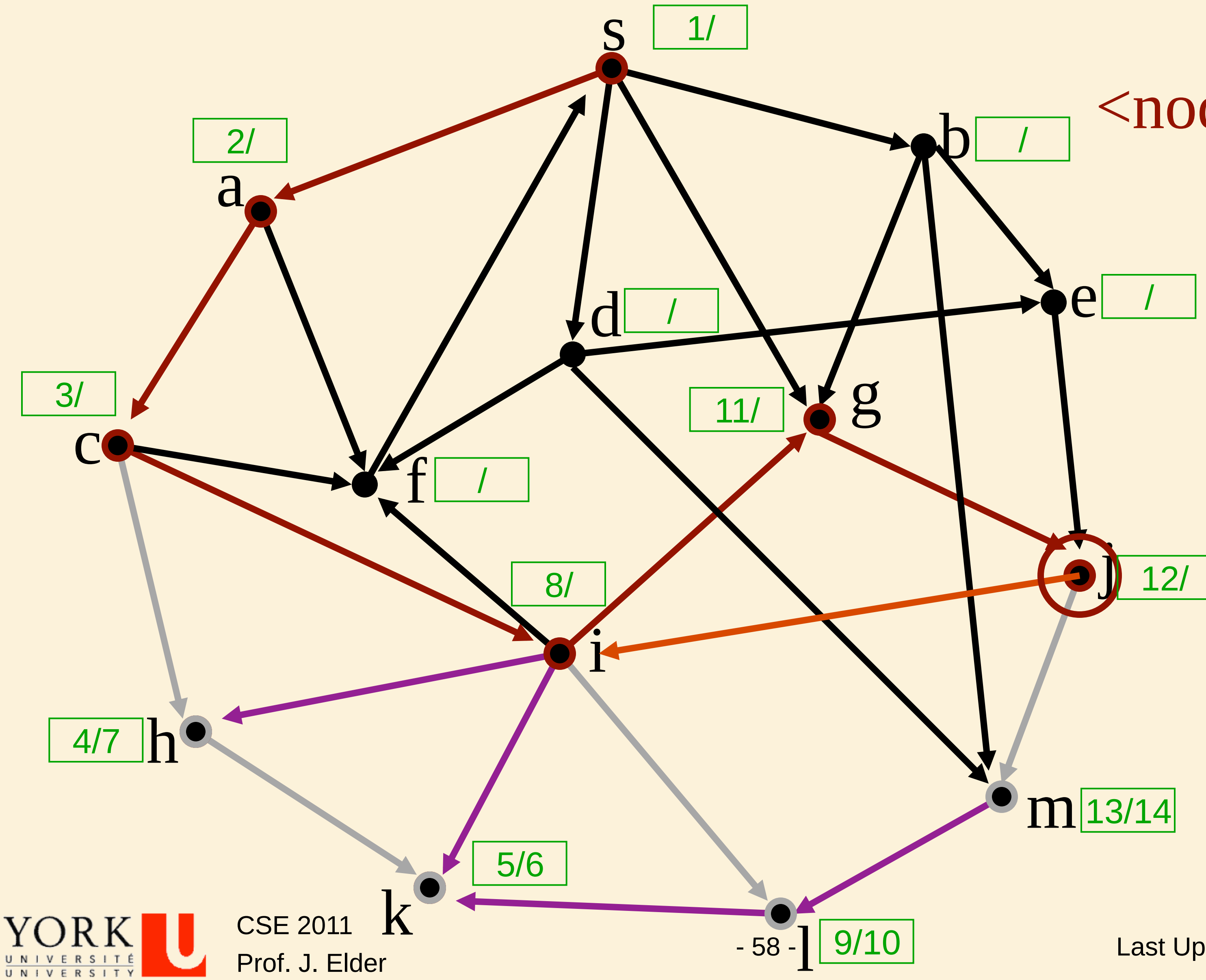
DFS



Found
Not Handled
Stack
<node,# edges>

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

DFS

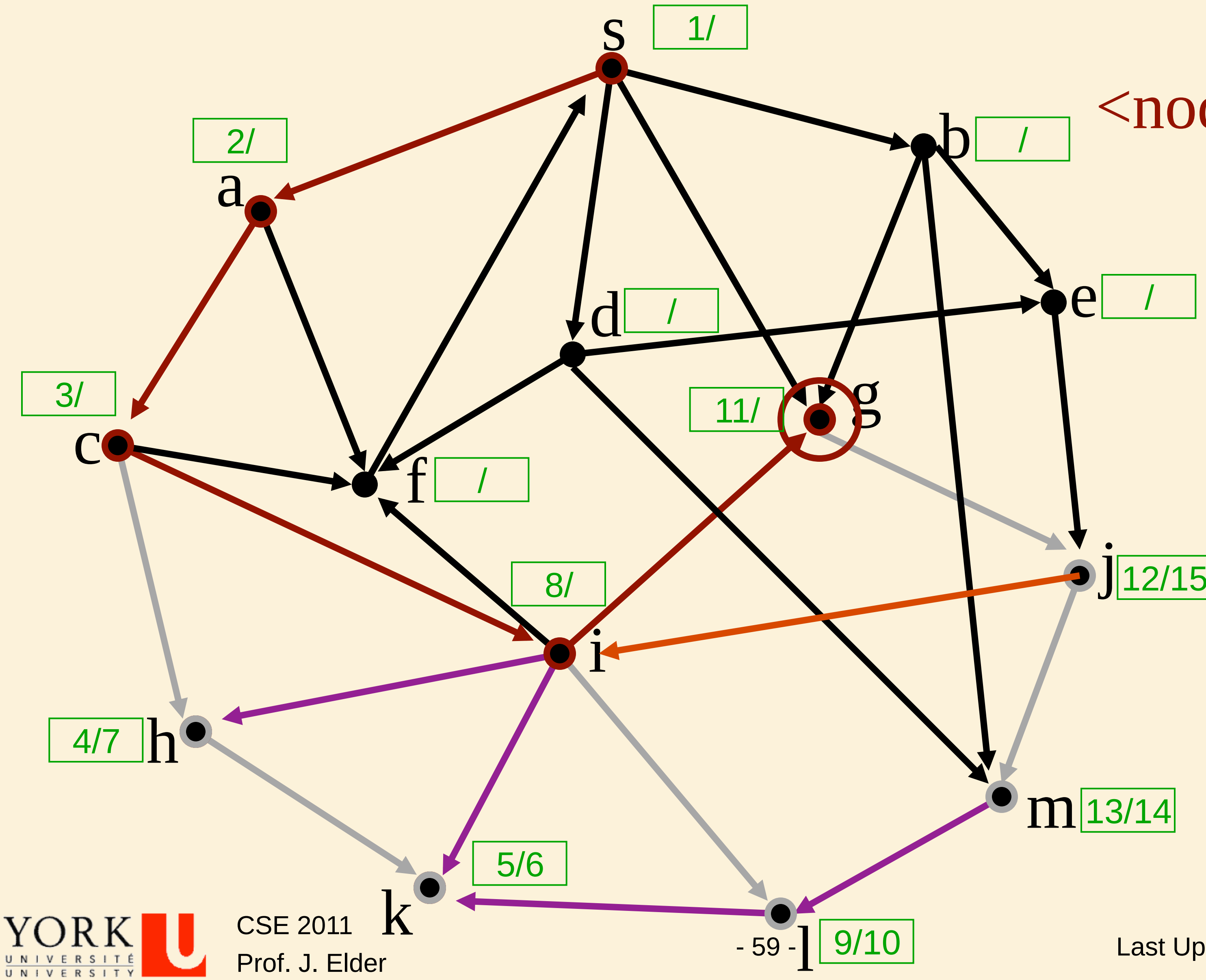


Found
Not Handled
Stack
<node,# edges>

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

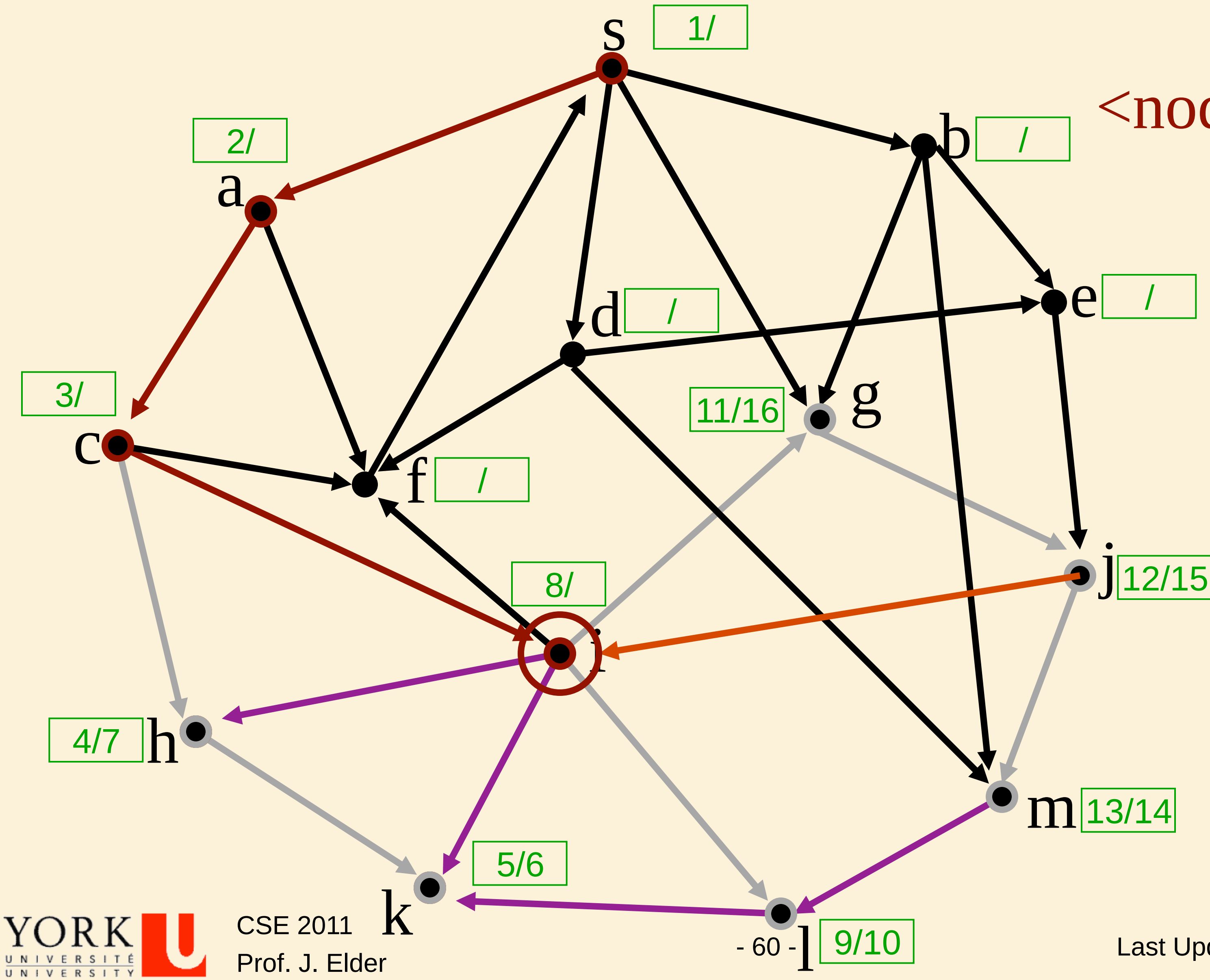
DFS

Found
Not Handled
Stack
<node,# edges>



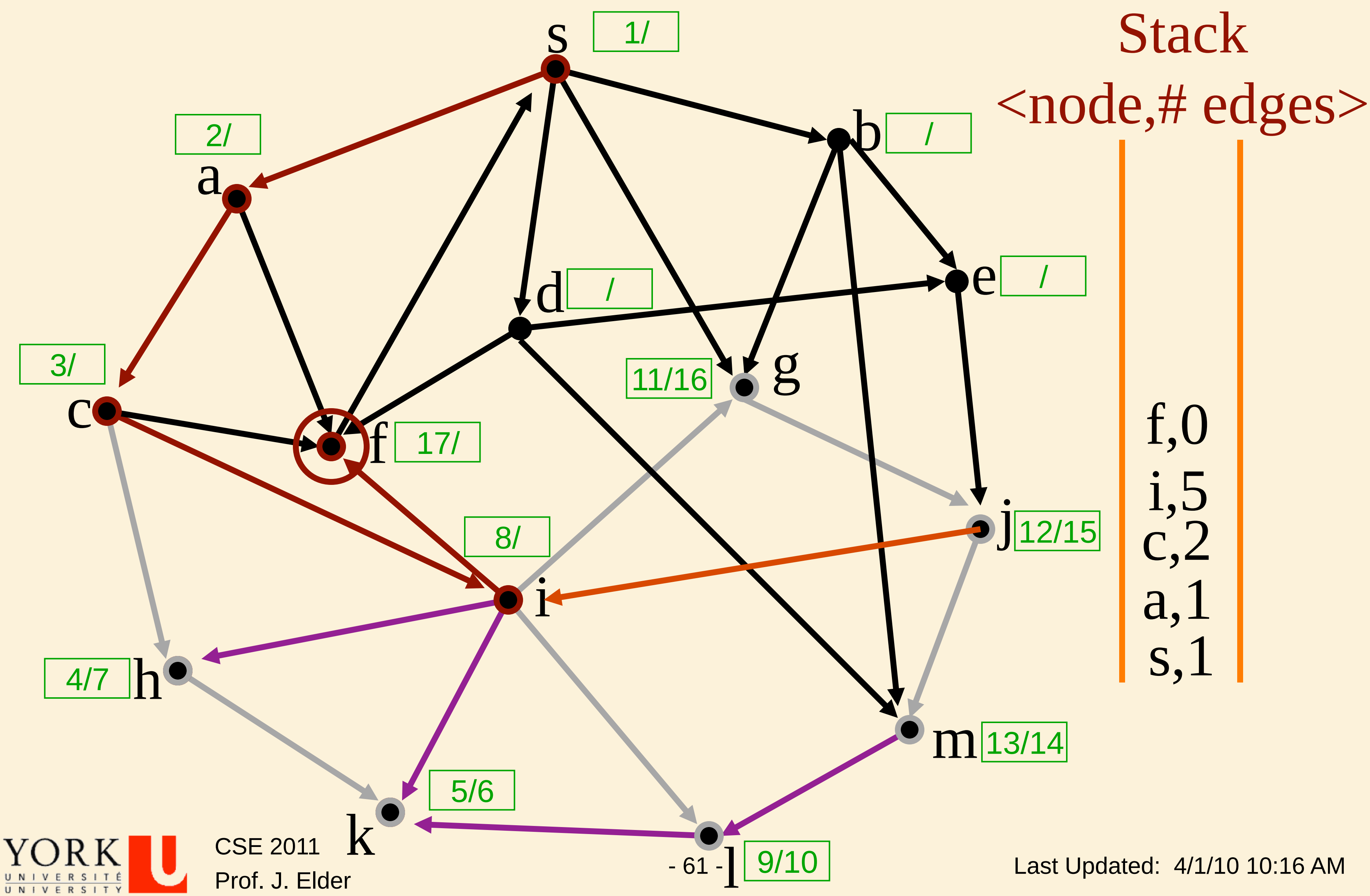
DFS

Found
Not Handled
Stack
<node,# edges>



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

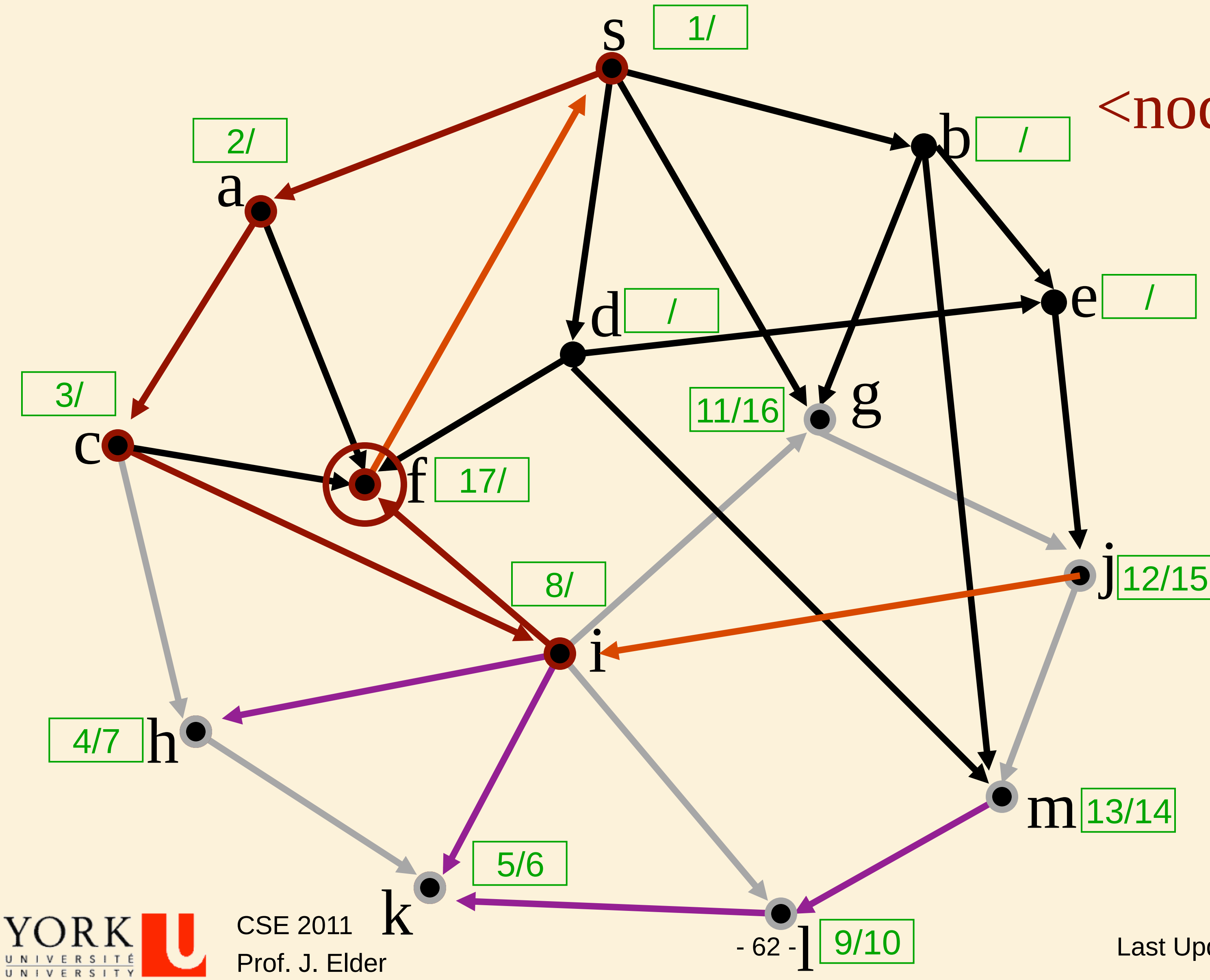
DFS



s
1/
a
2/
b
/
d
/
e
/
c
3/
f
17/
g
11/16
j
12/15
i
8/
h
4/7
m
13/14
k
5/6
l
9/10
Found
Not Handled
Stack
<node,# edges>
f,0
i,5
c,2
a,1
s,1

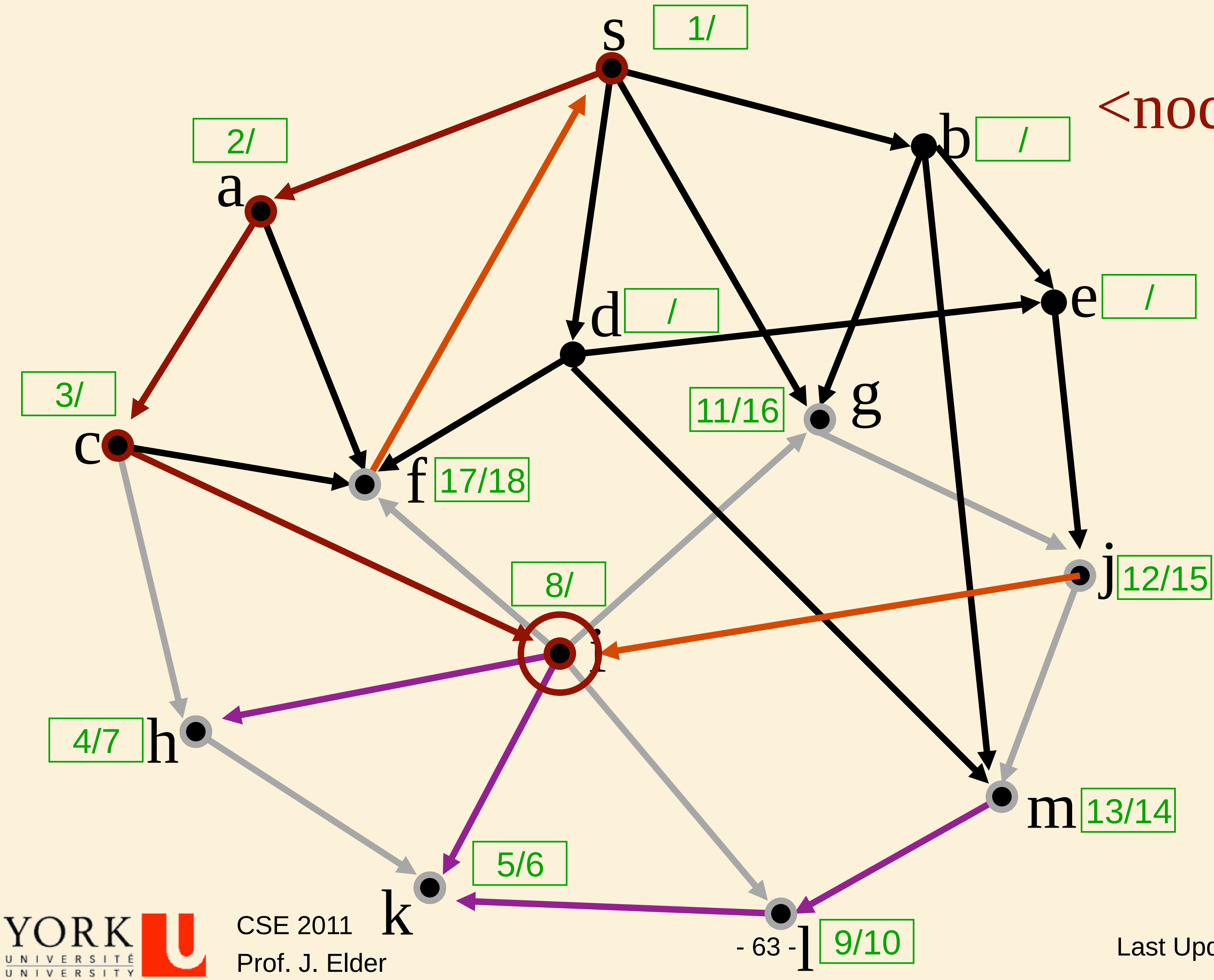
DFS

Found
Not Handled
Stack
<node,# edges>



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

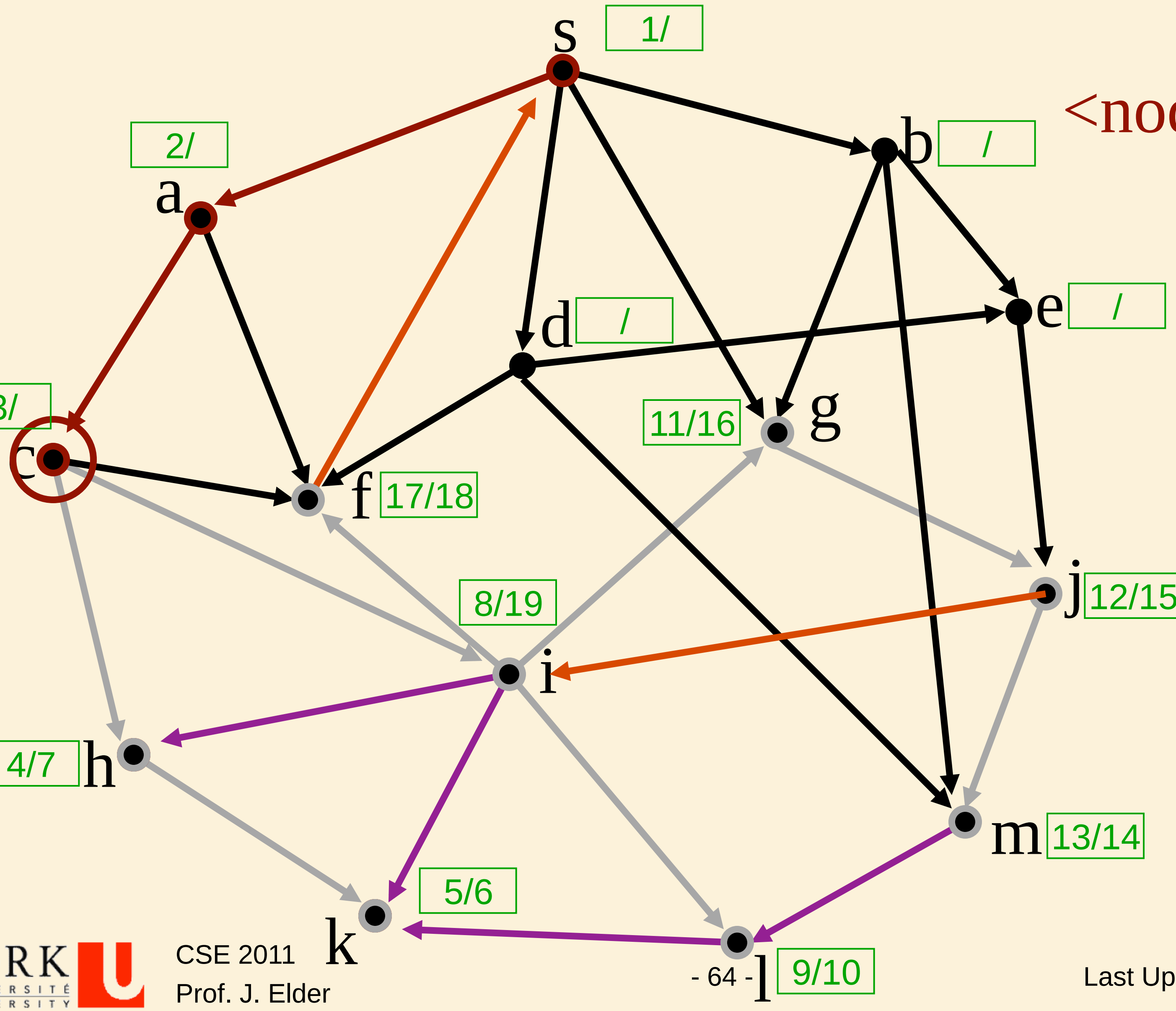
DFS



Found
Not Handled
Stack
<node,# edges>

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

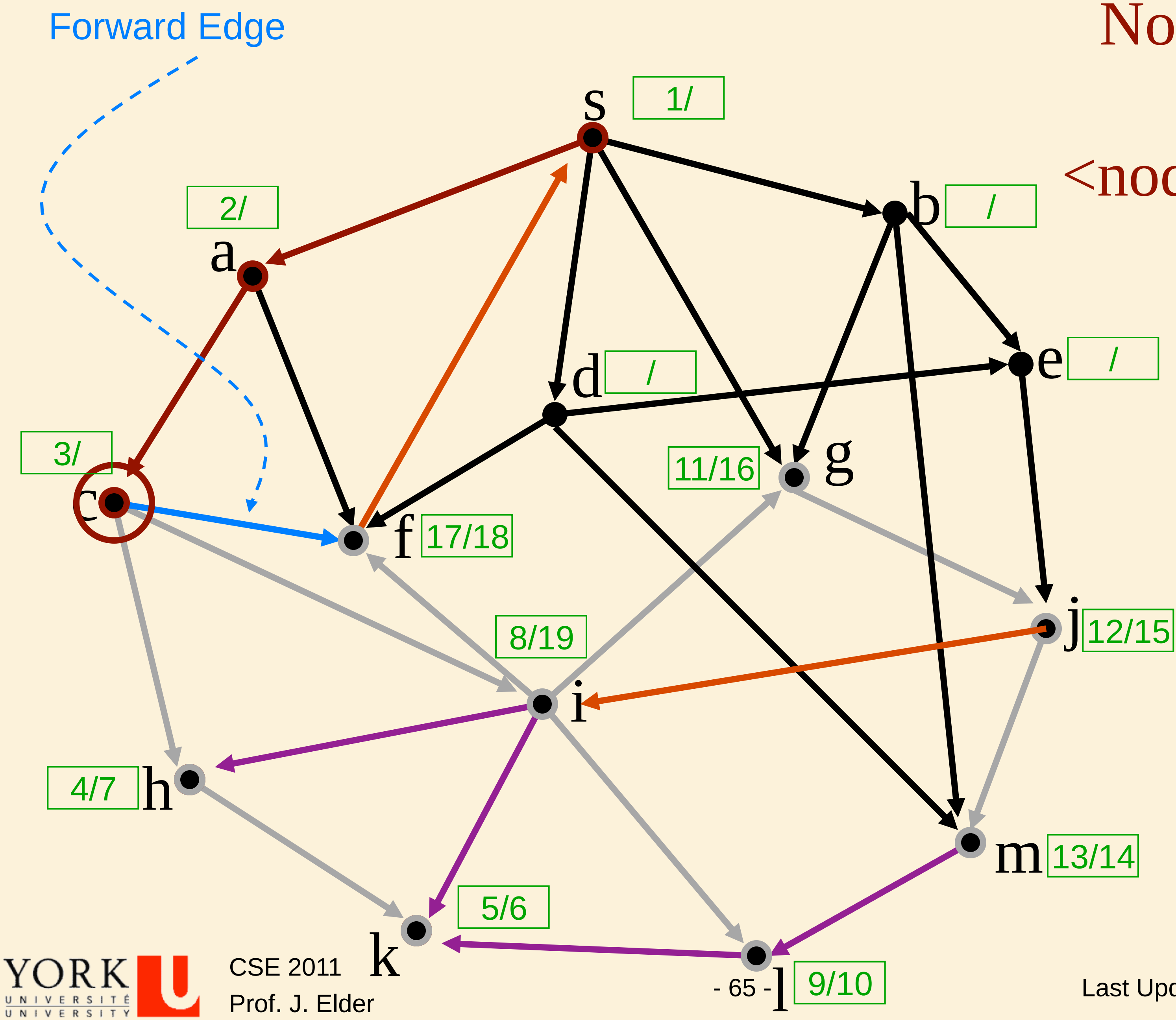
DFS



Found
Not Handled
Stack
<node,# edges>

c,2
a,1
s,1

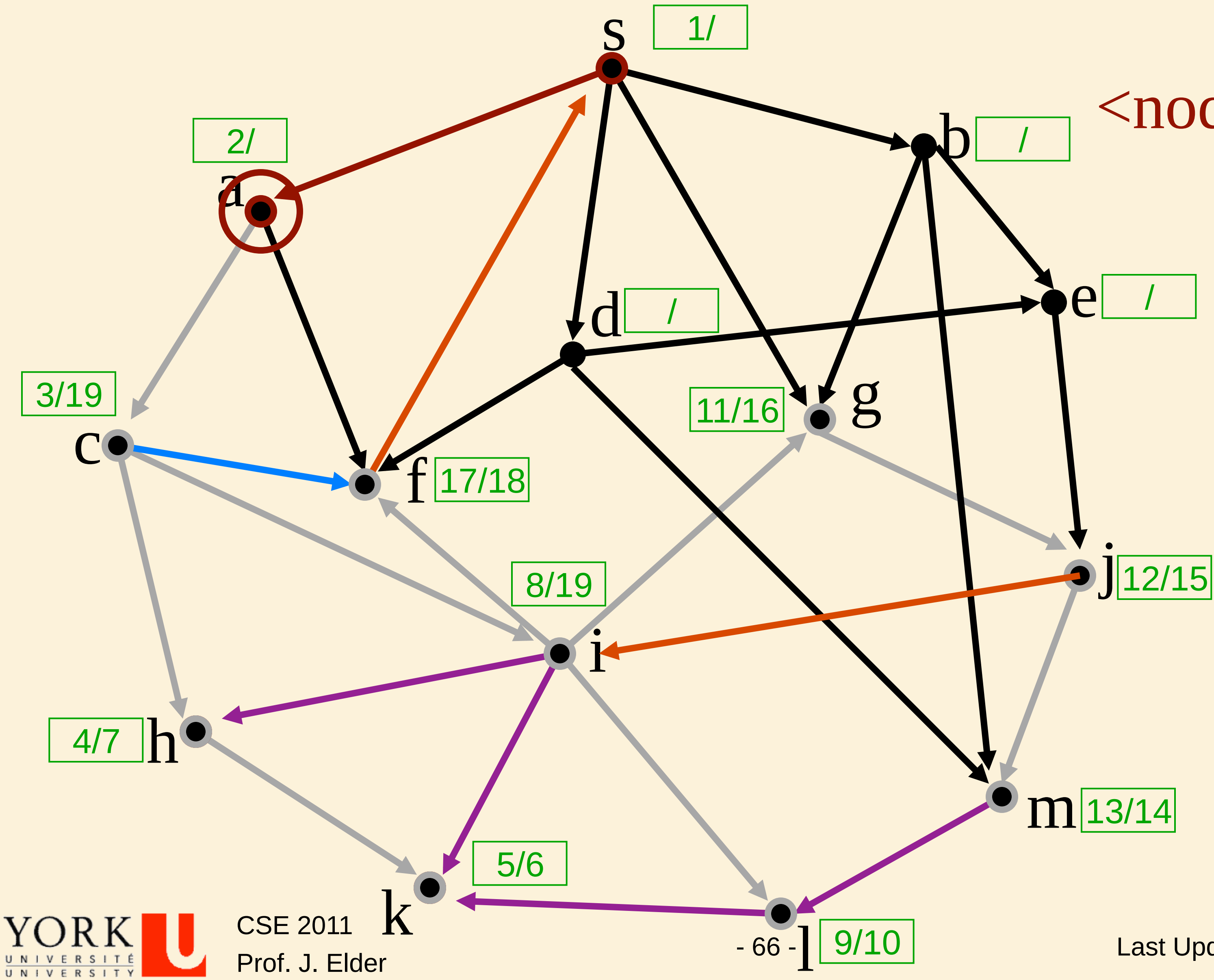
DFS



Found
Not Handled
Stack
<node,# edges>

c,3
a,1
s,1

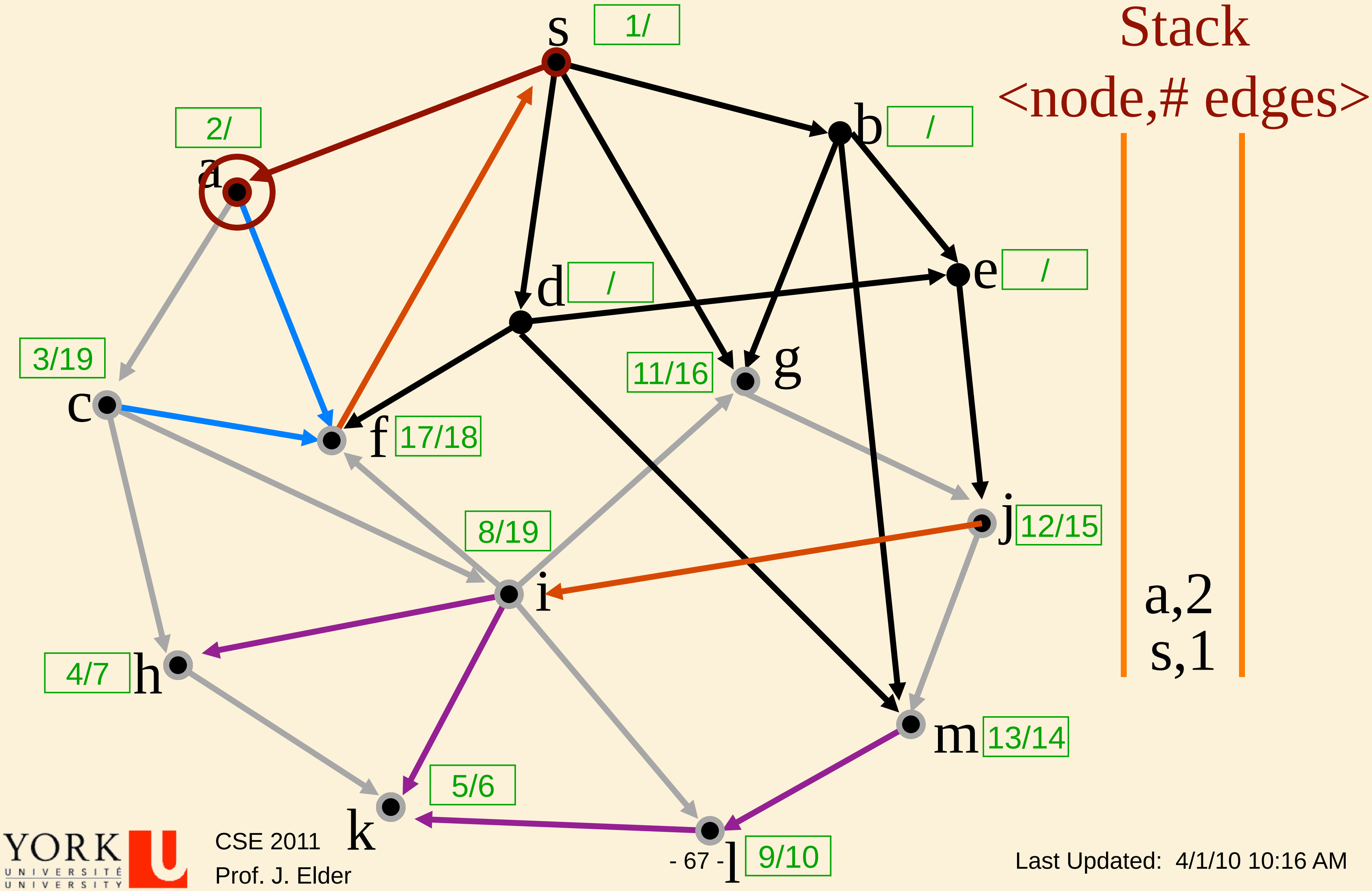
DFS



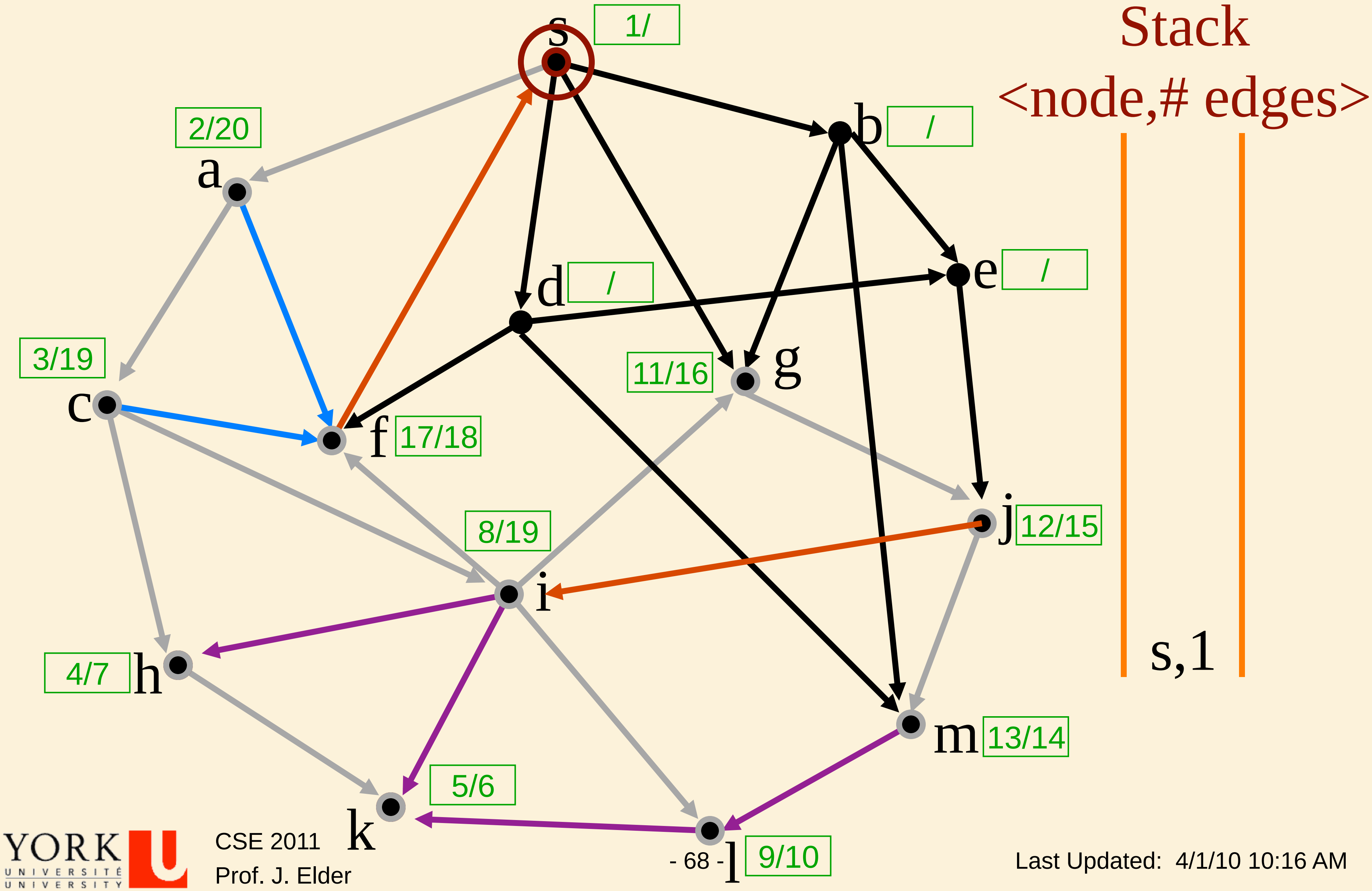
Found
Not Handled
Stack
<node,# edges>

a,1
s,1

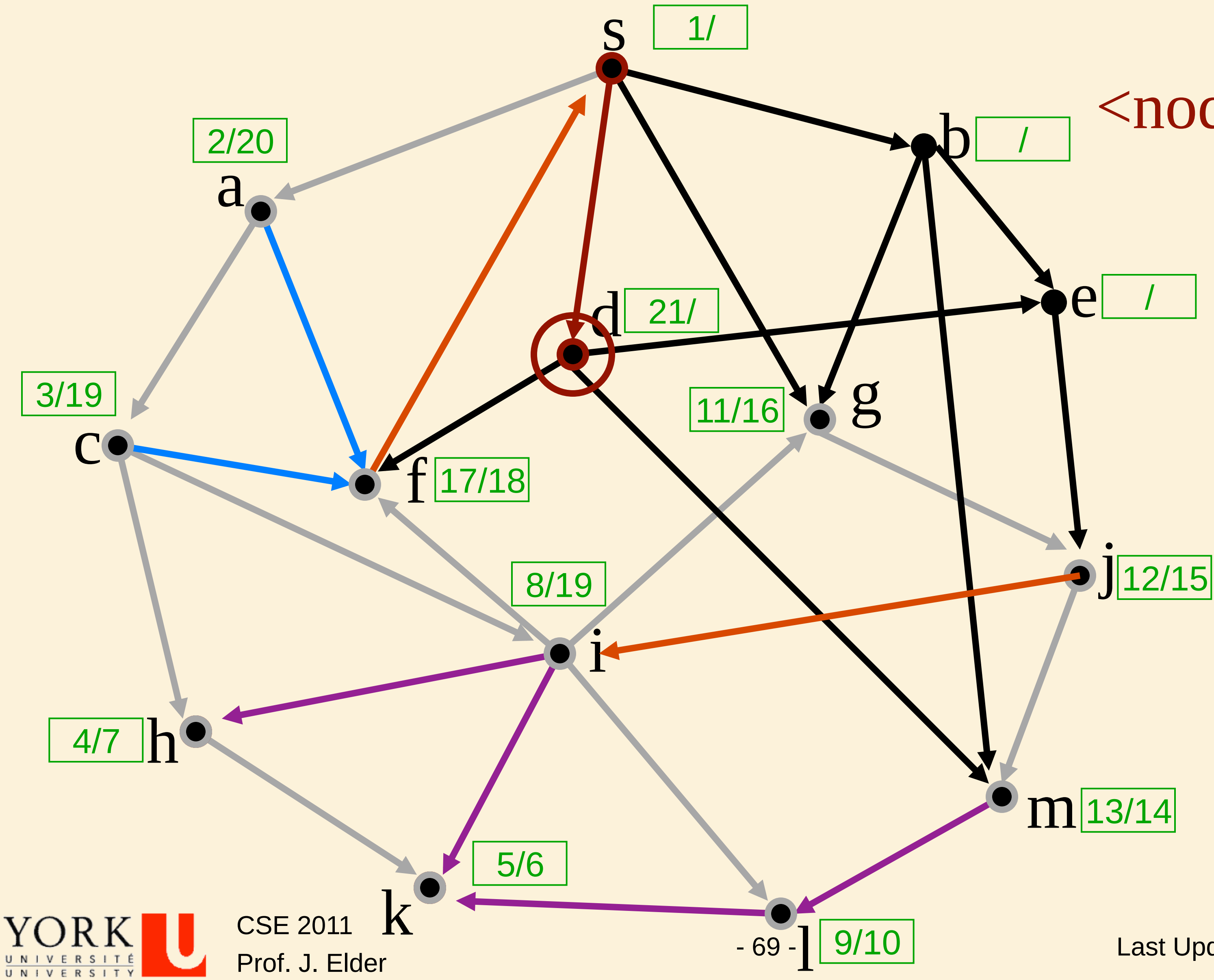
DFS


s
1/
a
2/
b
/
d
/
e
/
c
3/19
f
17/18
g
11/16
j
12/15
i
8/19
h
4/7
m
13/14
k
5/6
l
9/10
Found
Not Handled
Stack
<node,# edges>
a,2
s,1

DFS


s
1/
a
2/20
b
/
d
/
e
/
c
3/19
f
17/18
g
11/16
j
12/15
i
8/19
h
4/7
m
13/14
k
5/6
l
9/10
Found
Not Handled
Stack
<node,# edges>
s,1

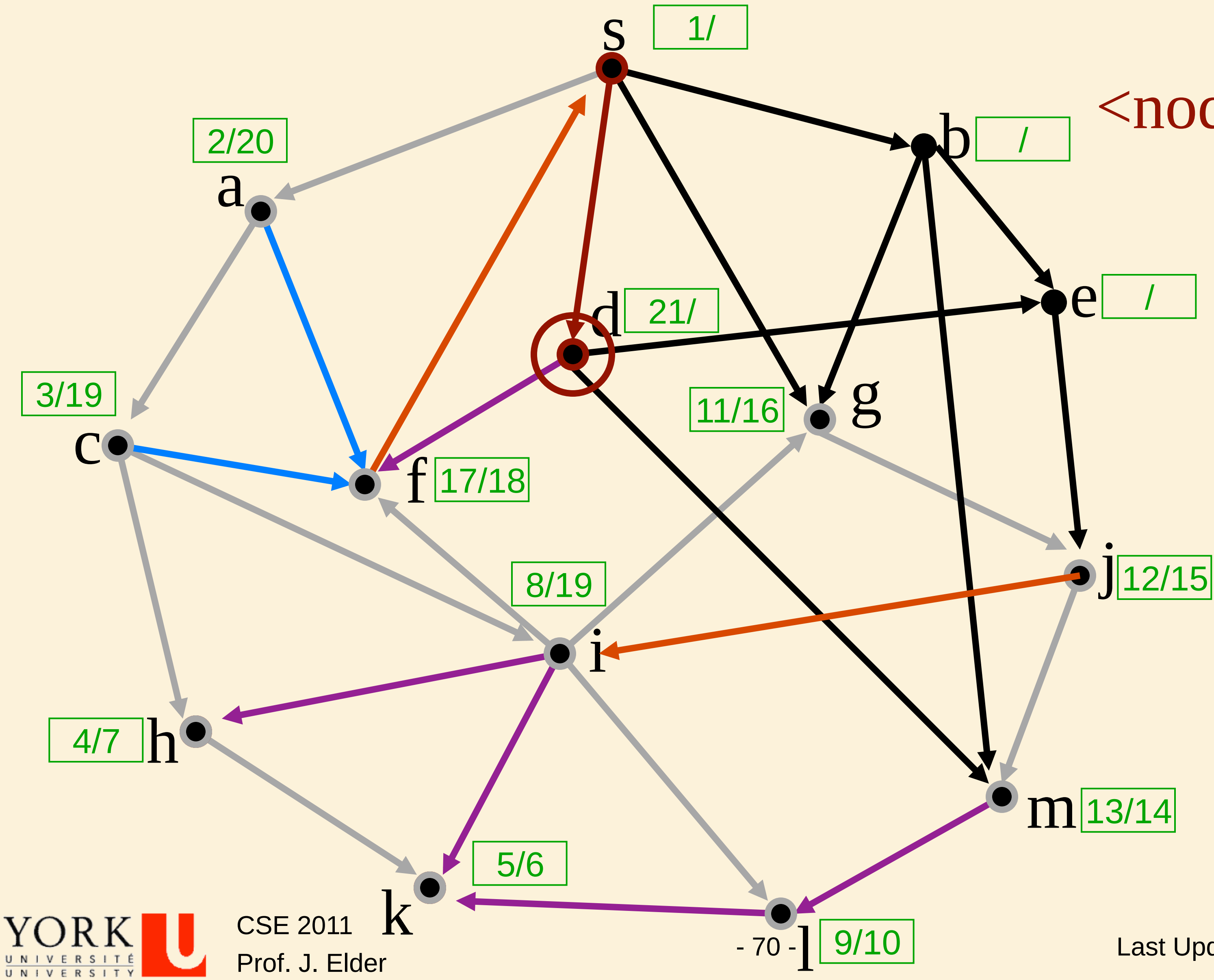
DFS



Found
Not Handled
Stack
<node,# edges>

d,0
s,2

DFS

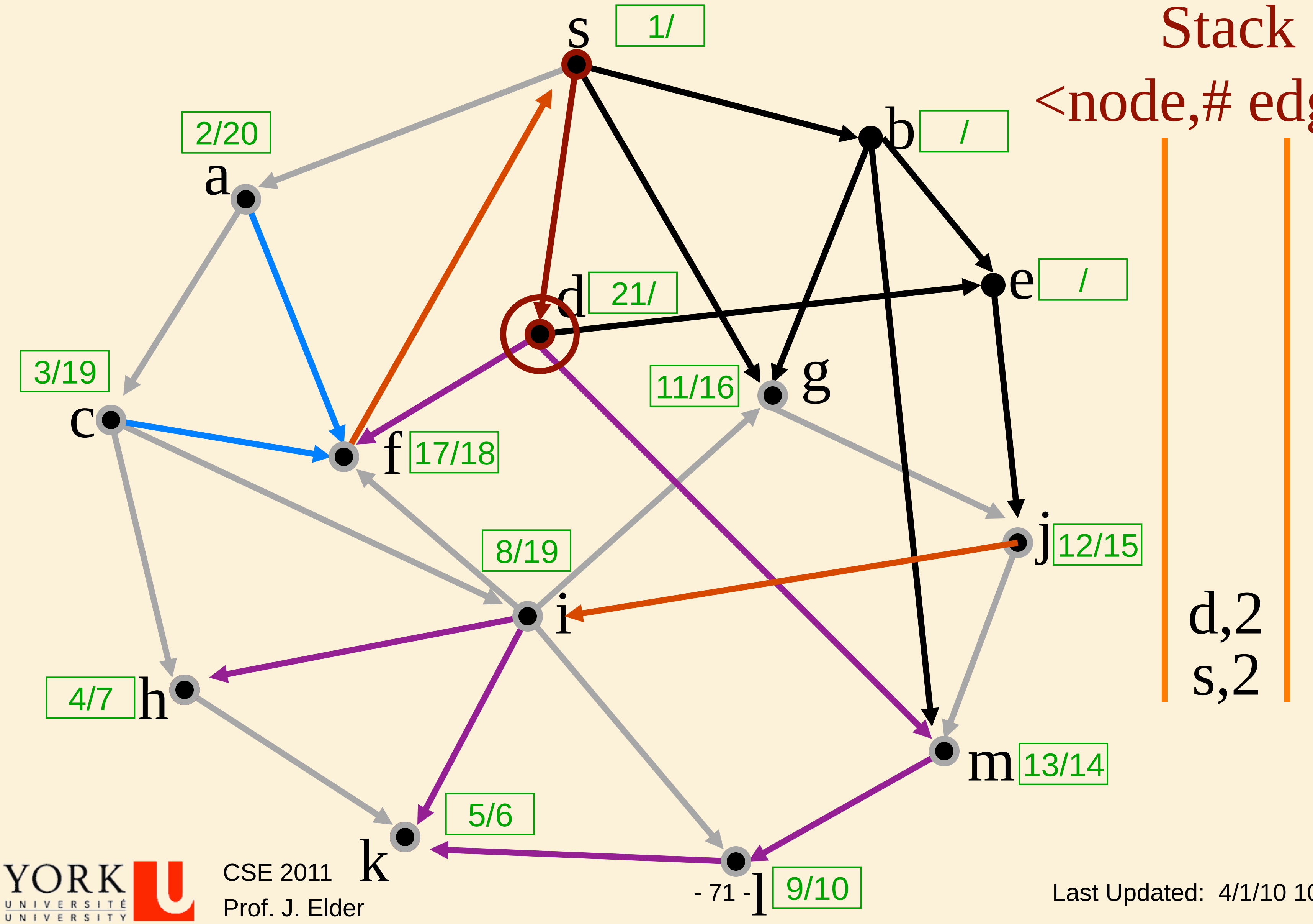


Found
Not Handled
Stack
<node,# edges>

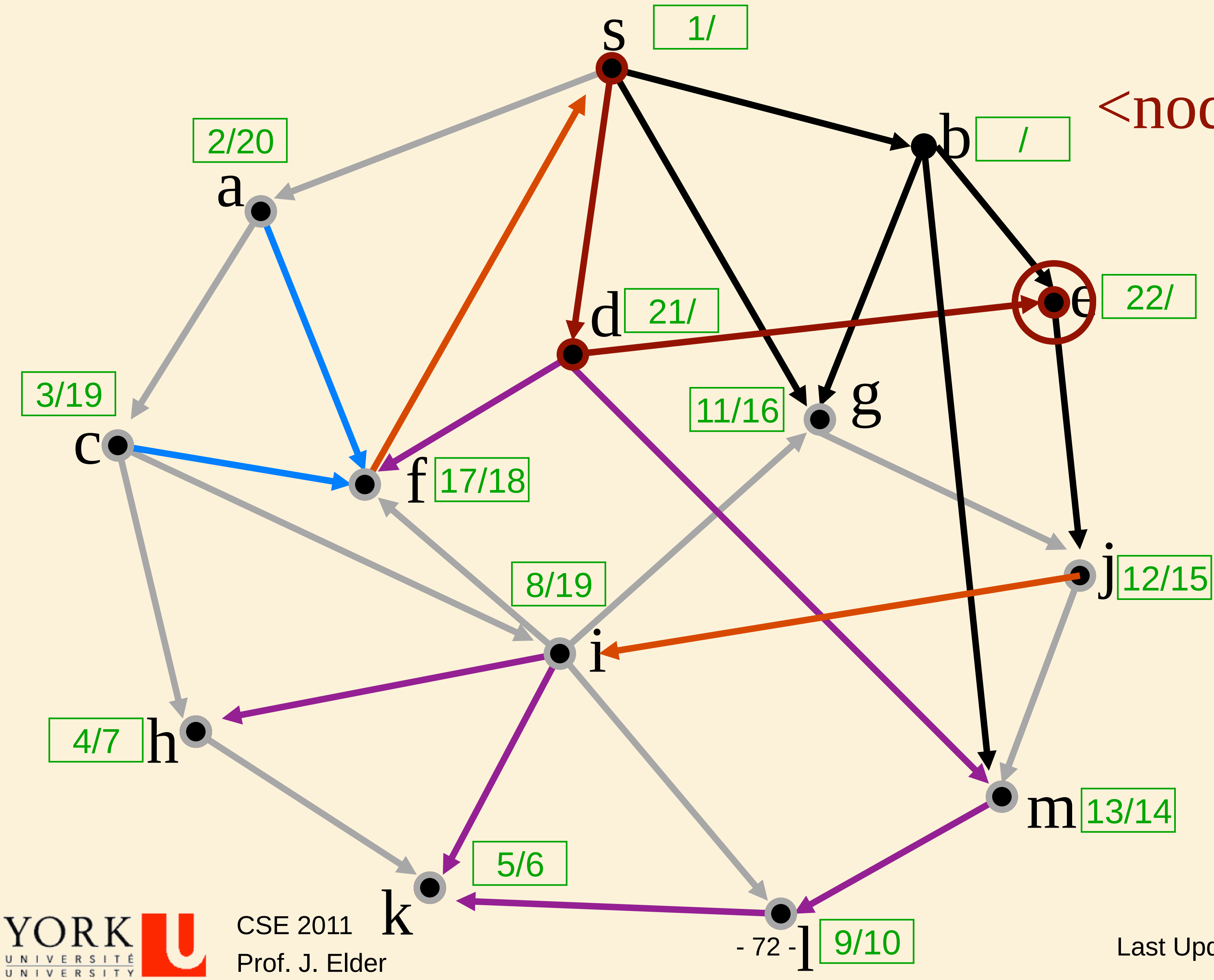
d,1
s,2

DFS

Found
Not Handled
Stack
<node,# edges>



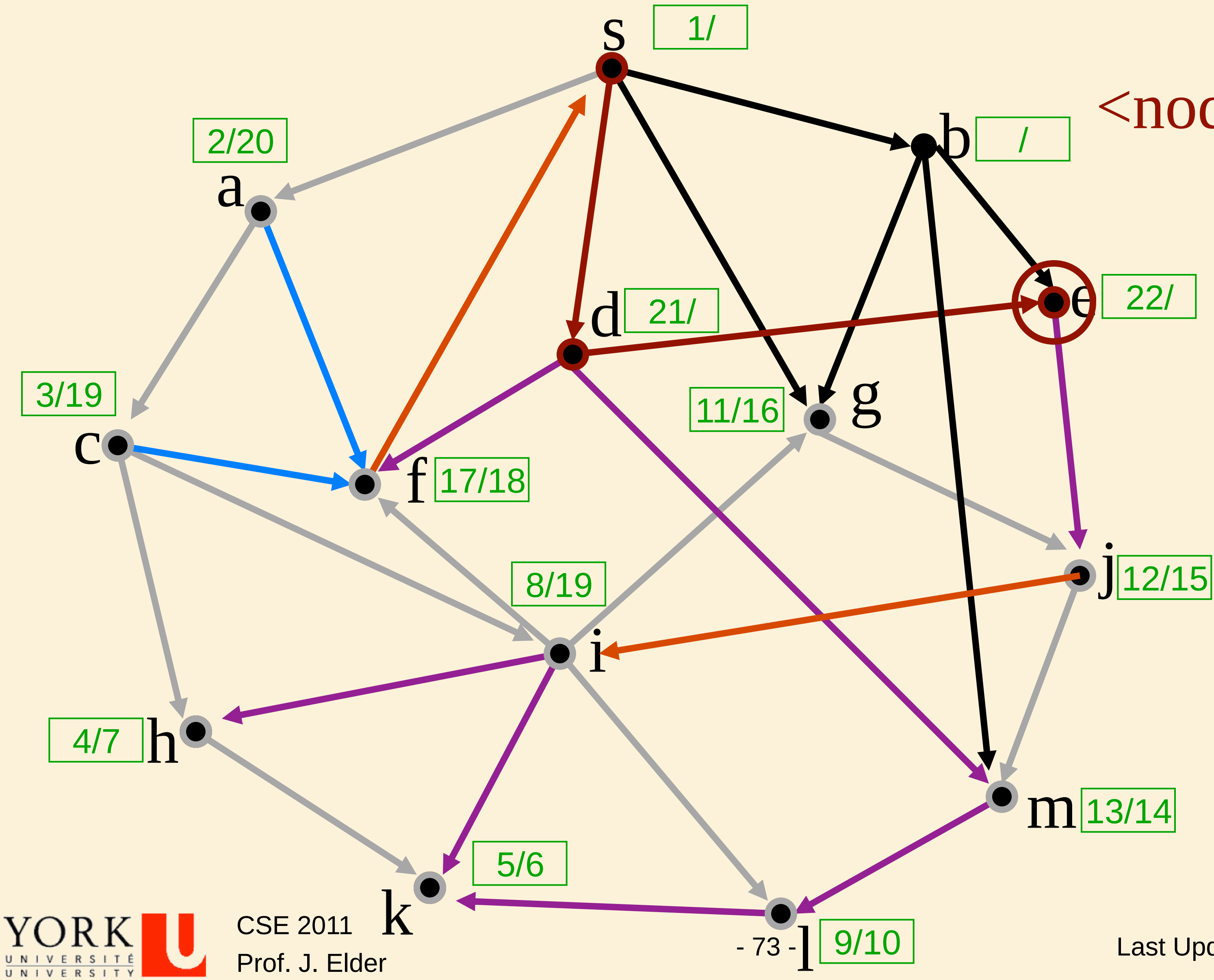
DFS



Found
Not Handled
Stack
<node,# edges>

e,0
d,3
s,2

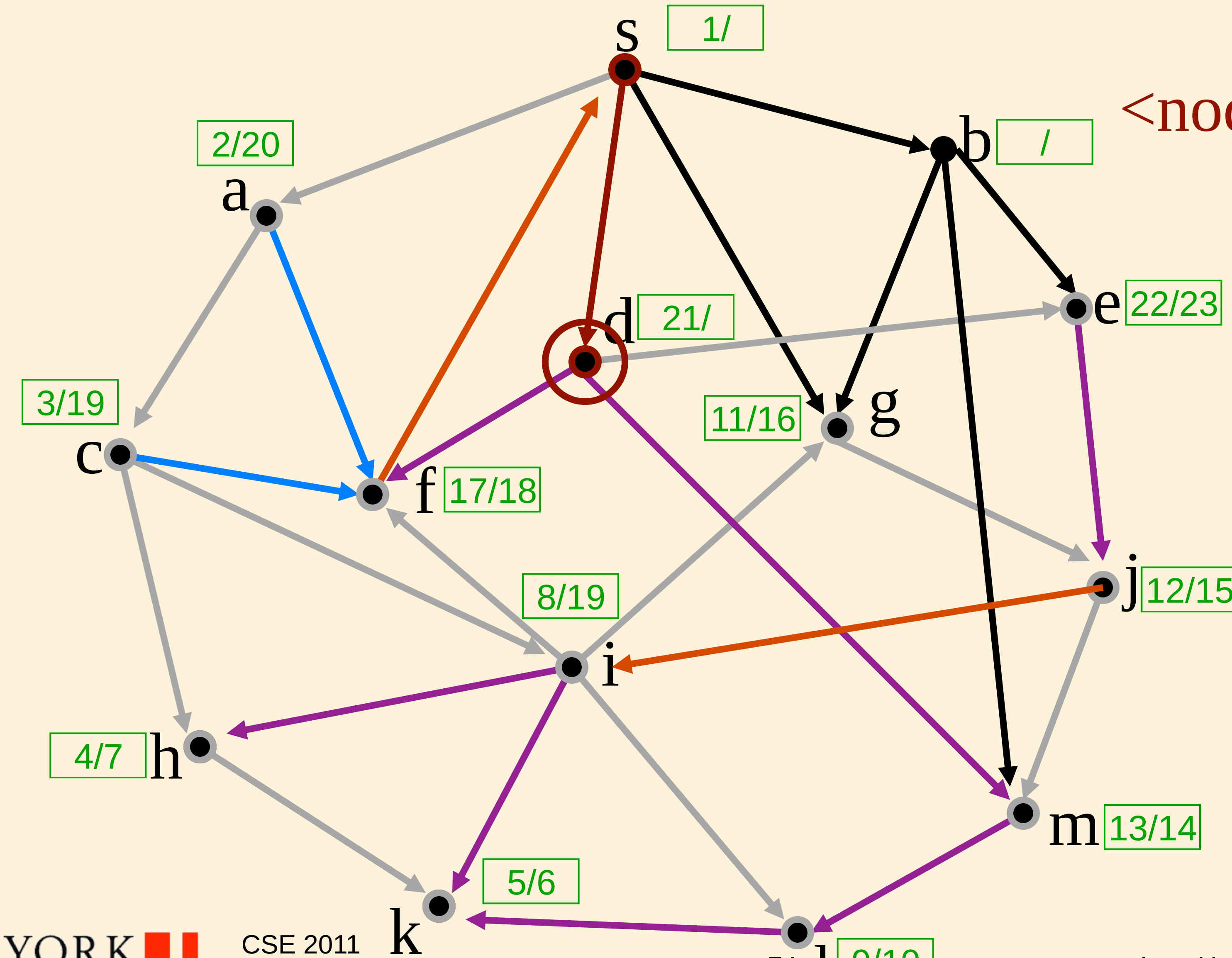
DFS



Found
Not Handled
Stack
<node,# edges>

e,1
d,3
s,2

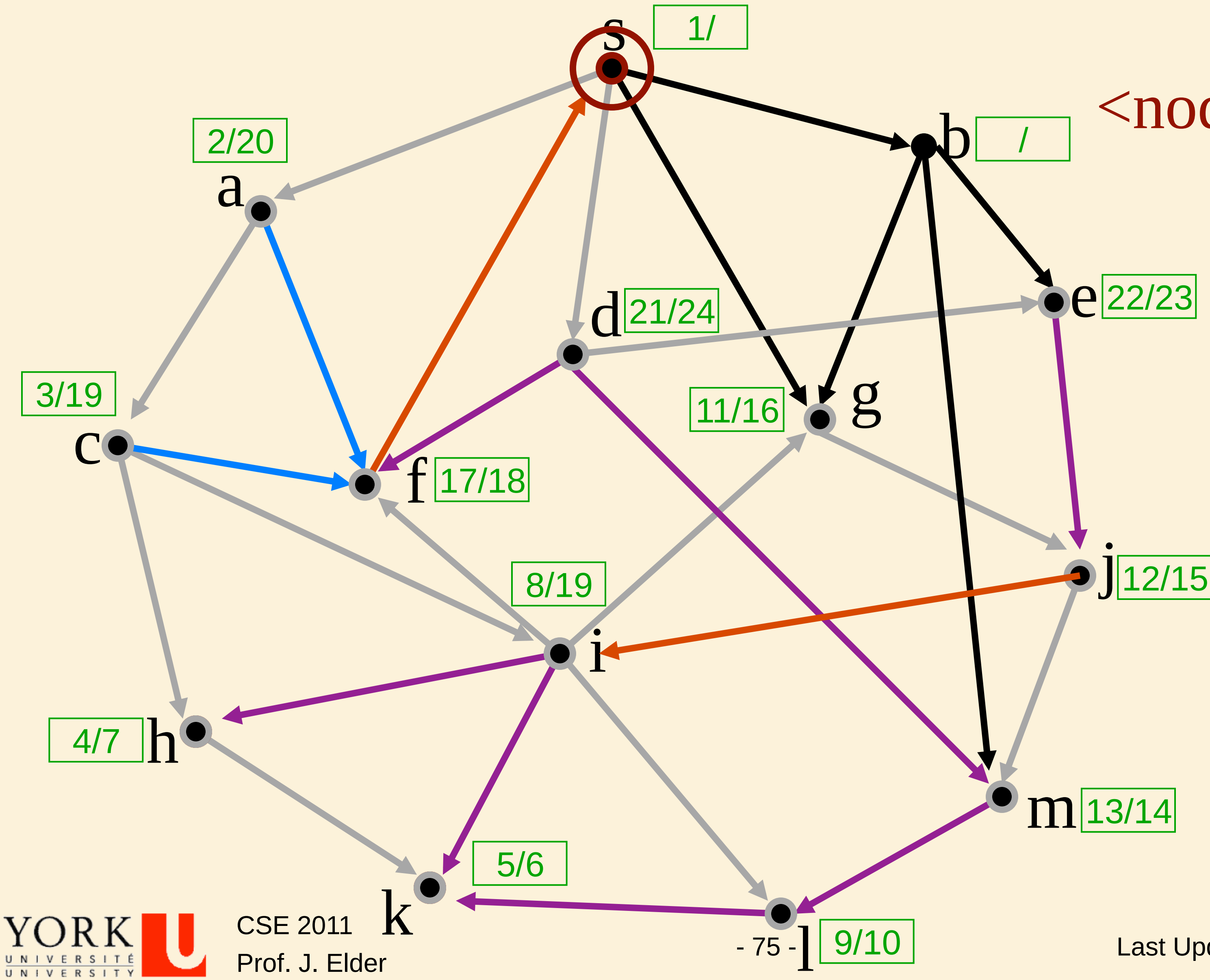
DFS



Found
Not Handled
Stack
<node,# edges>

d,3
s,2

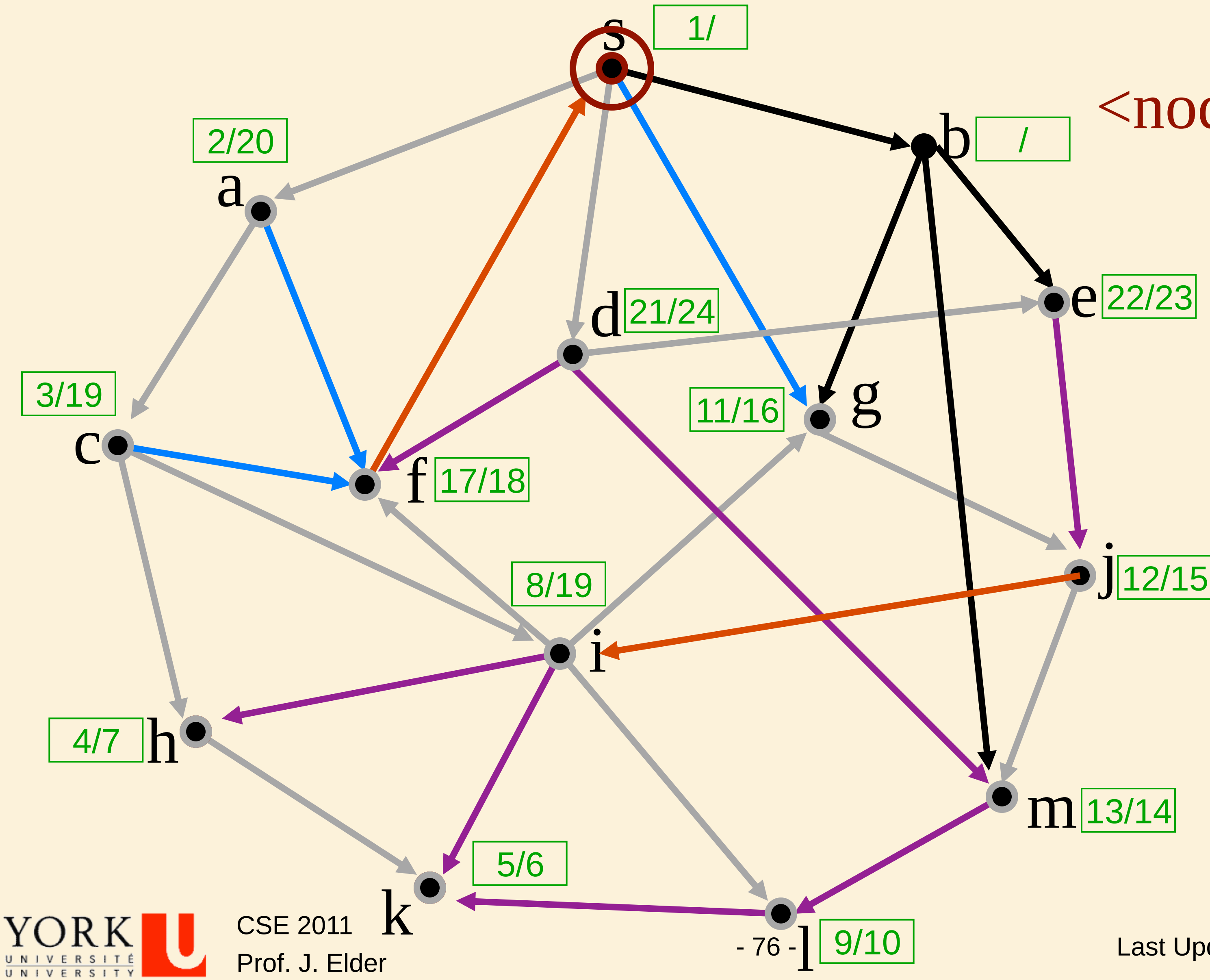
DFS



Found
Not Handled
Stack
<node,# edges>

s,2

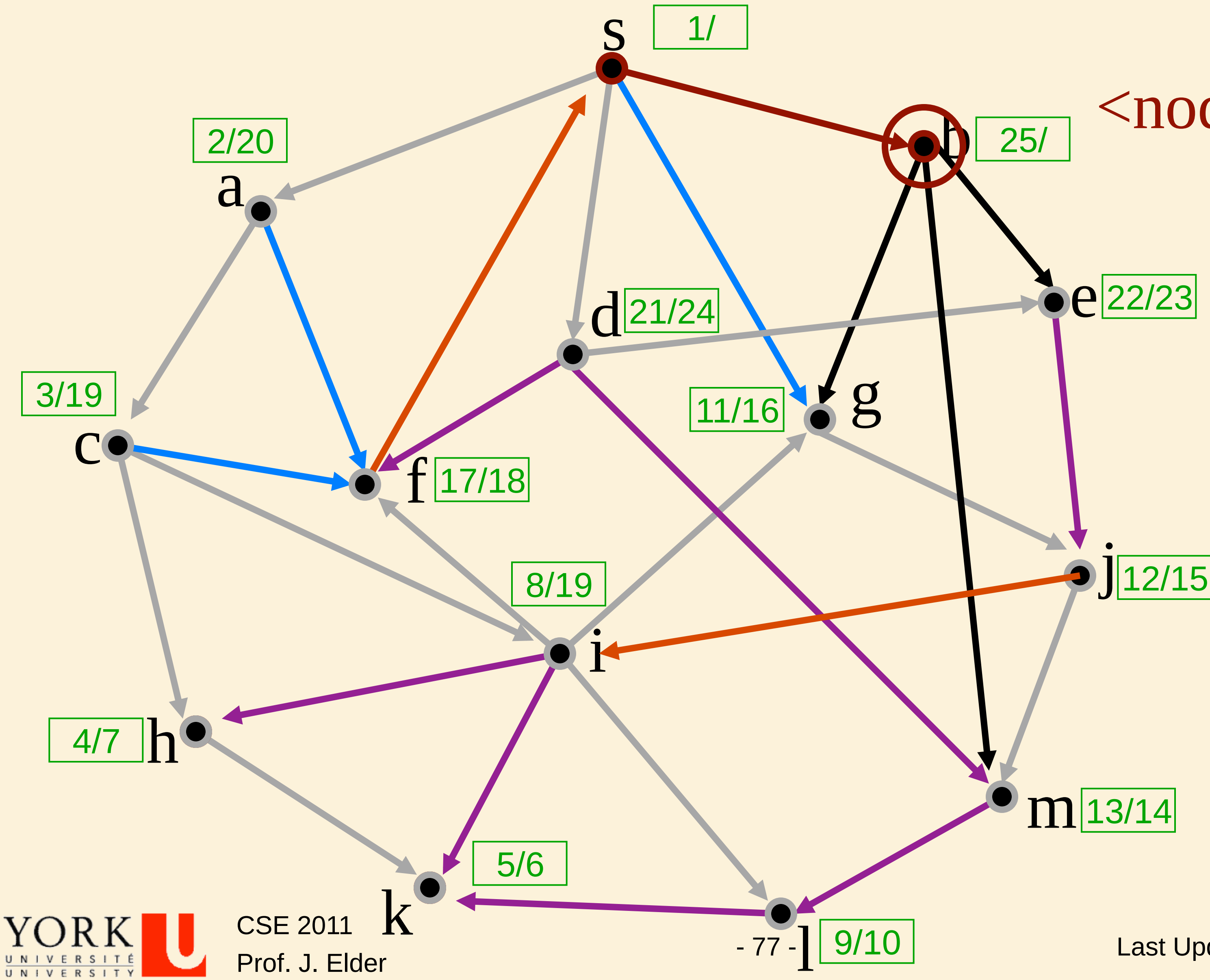
DFS



Found
Not Handled
Stack
<node,# edges>

s,3

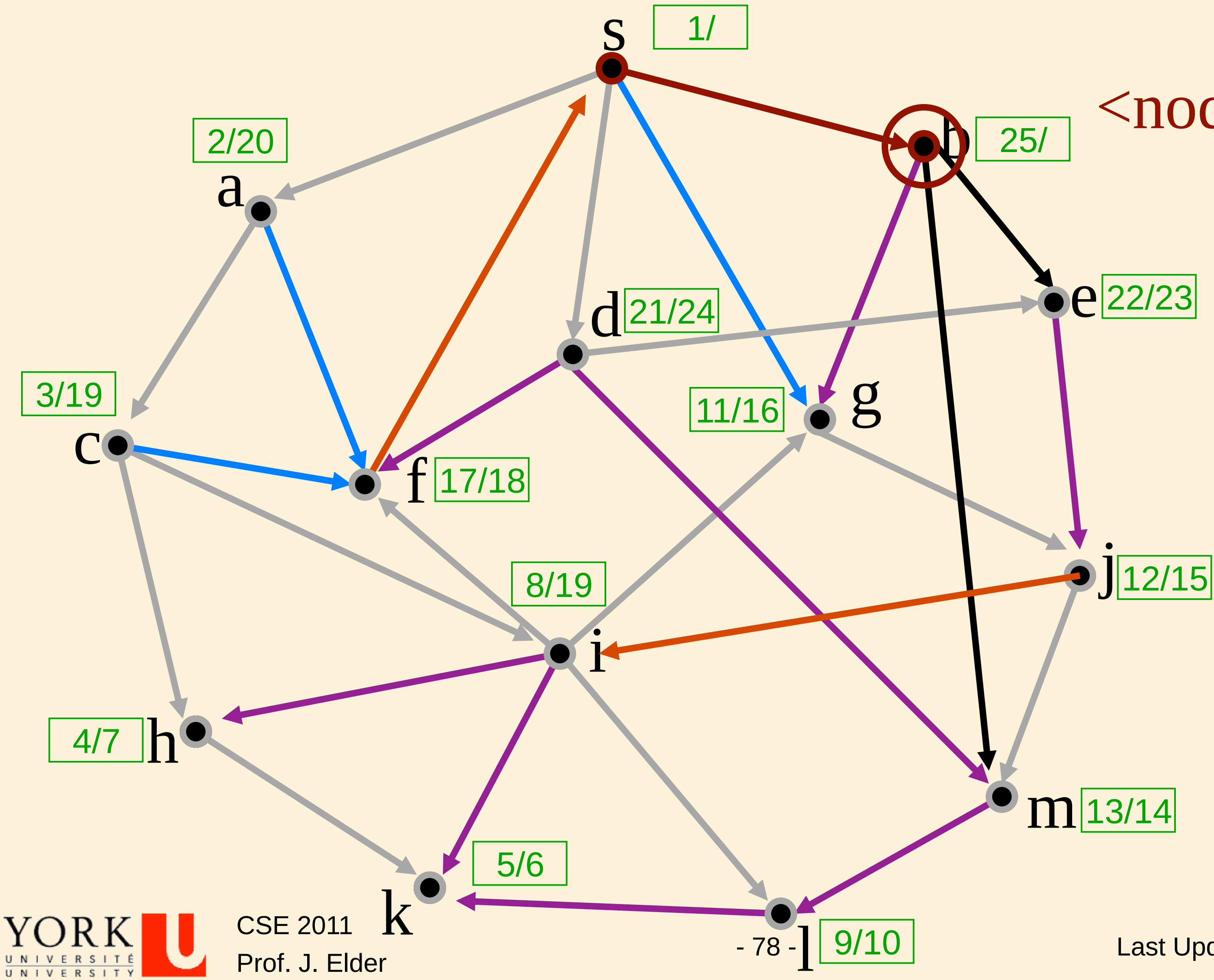
DFS



Found
Not Handled
Stack
<node,# edges>

b,0
s,4

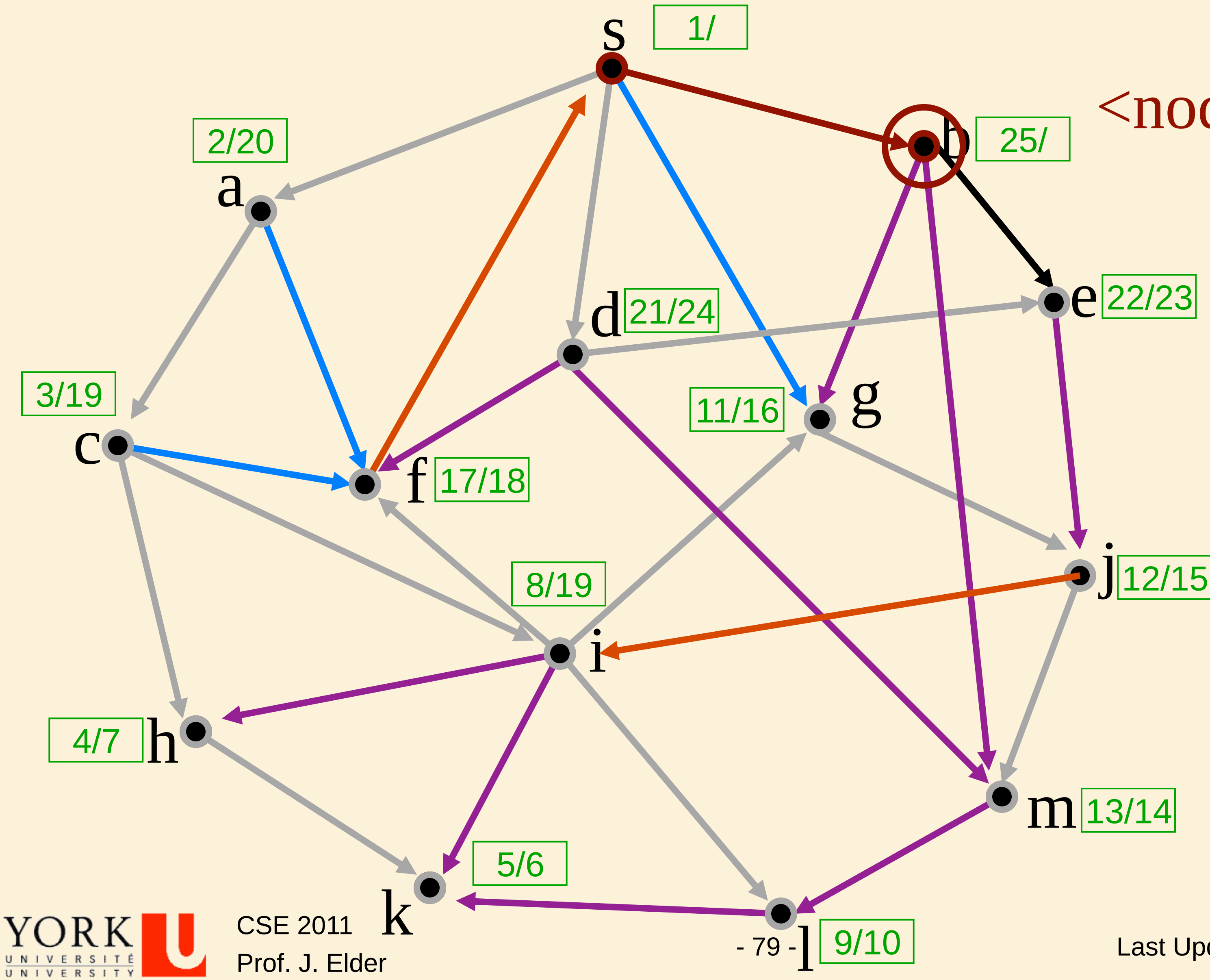
DFS



Found
Not Handled
Stack
<node,# edges>

b,1
s,4

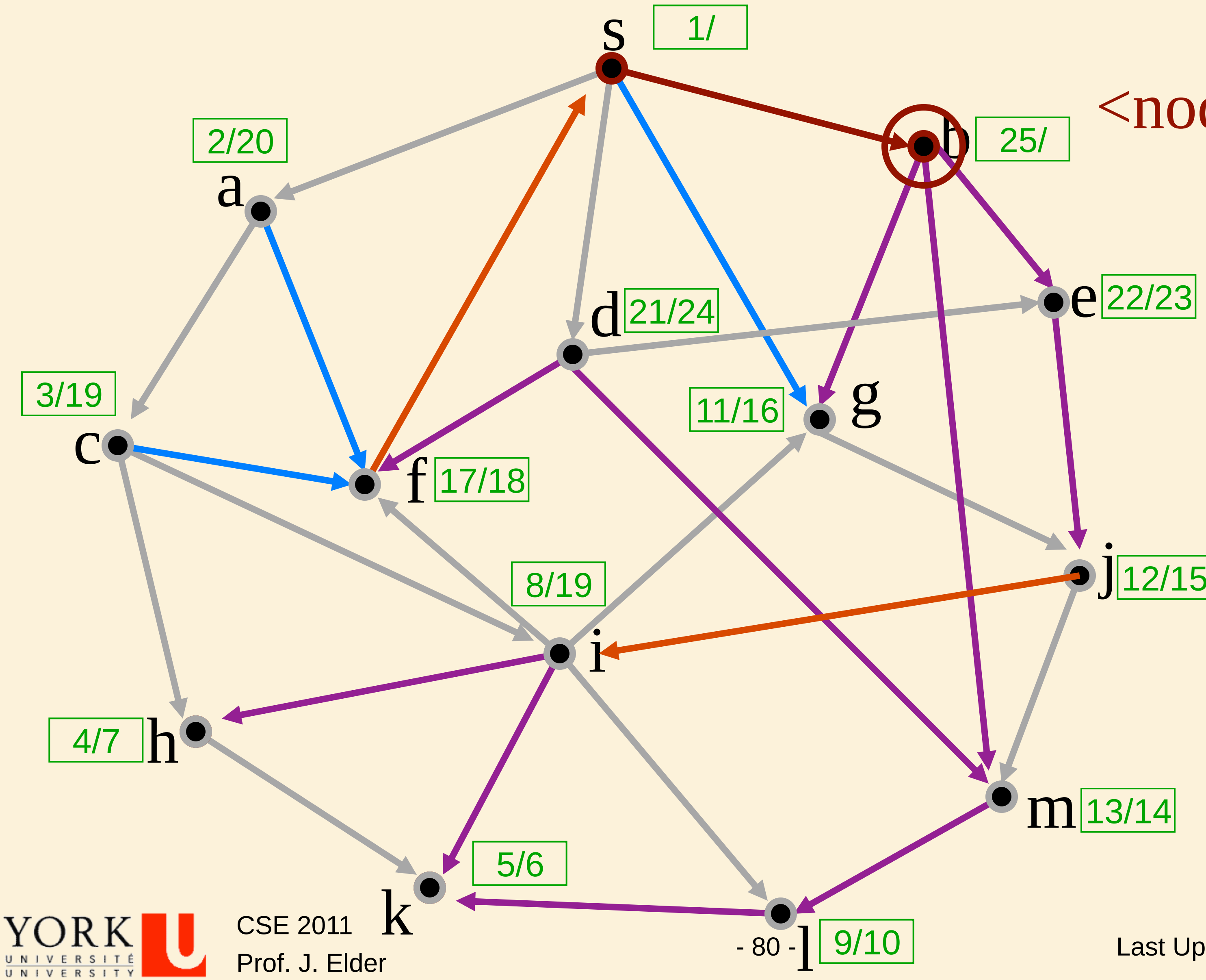
DFS



Found
Not Handled
Stack
<node,# edges>

b,2
s,4

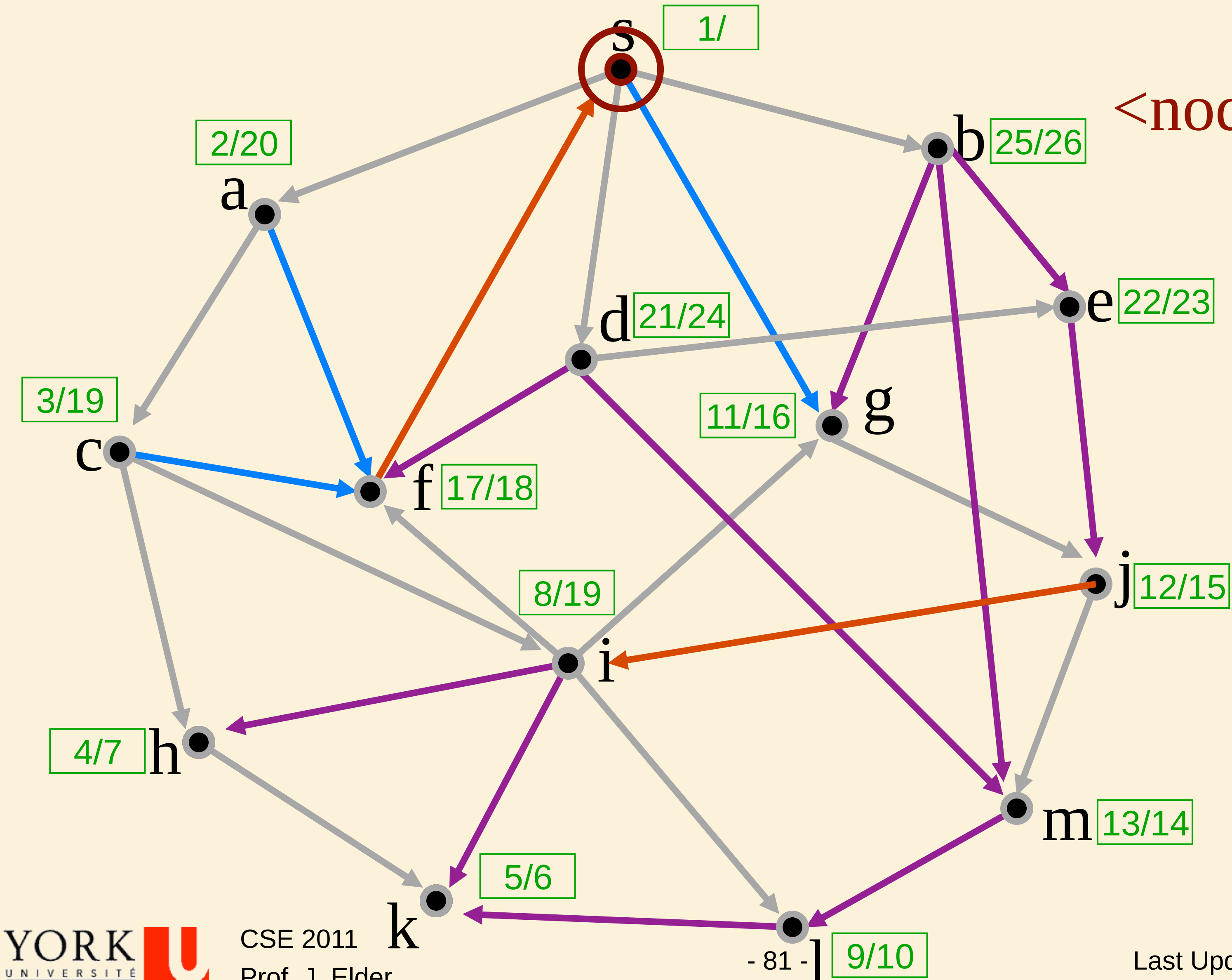
DFS



Found
Not Handled
Stack
<node,# edges>

b,3
s,4

DFS



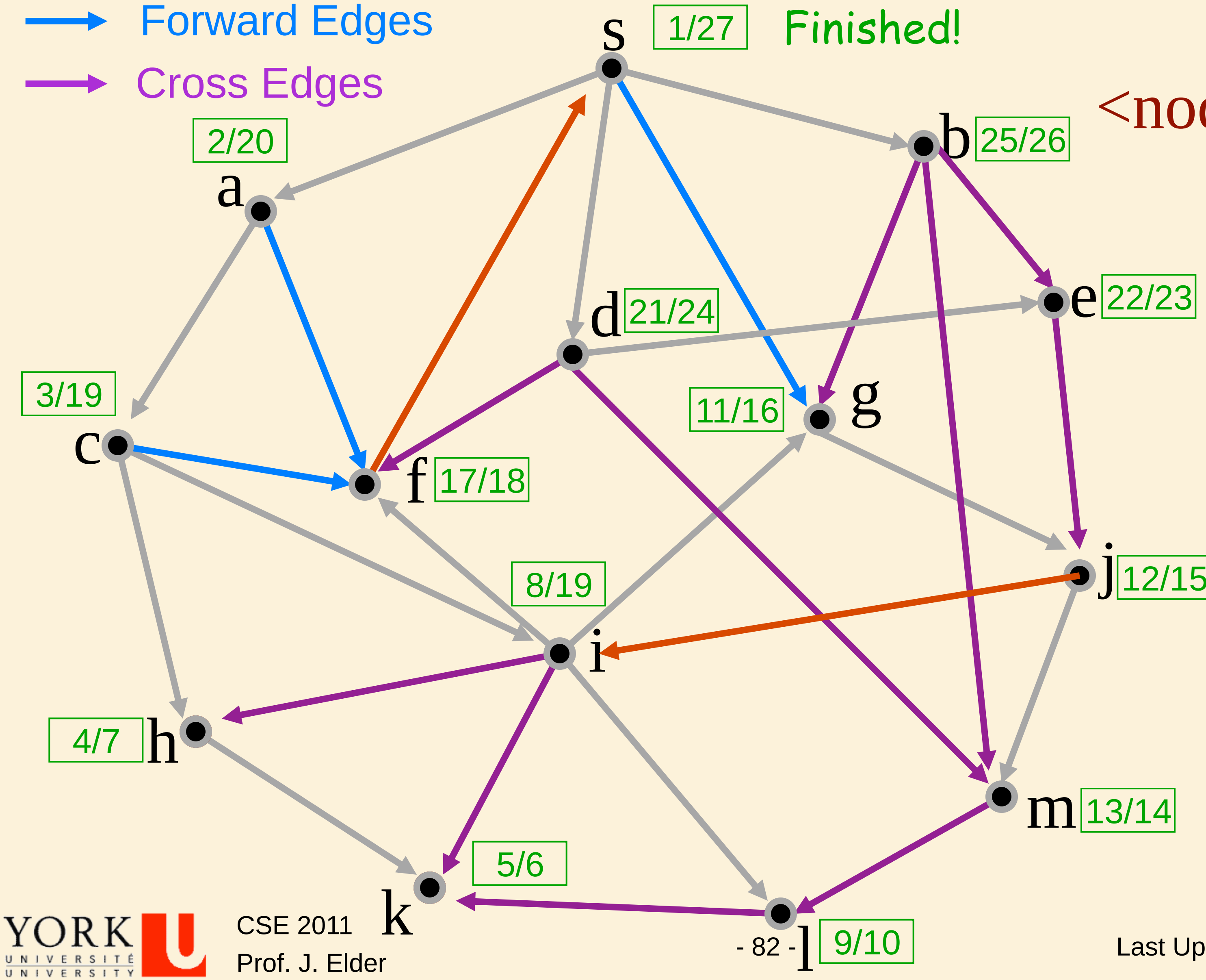
Found
Not Handled
Stack
<node,# edges>

s,4

DFS

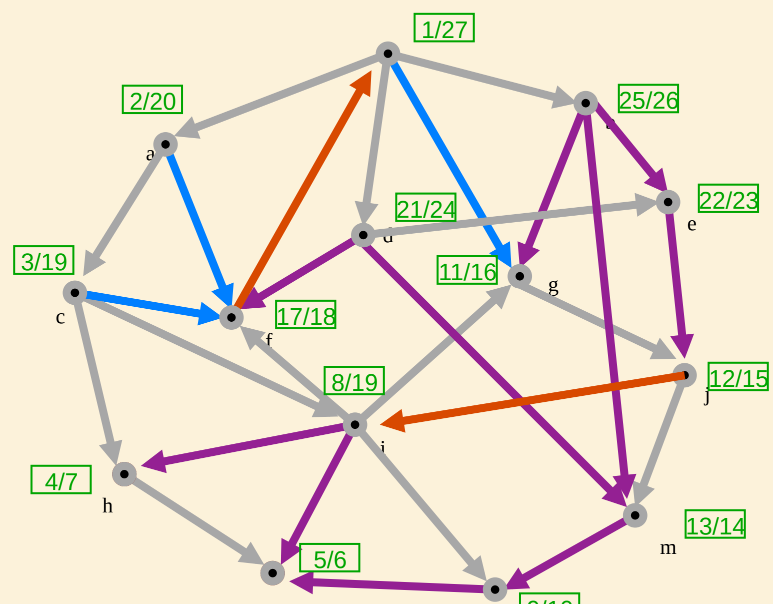
Tree Edges
Back Edges
Forward Edges
Cross Edges

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

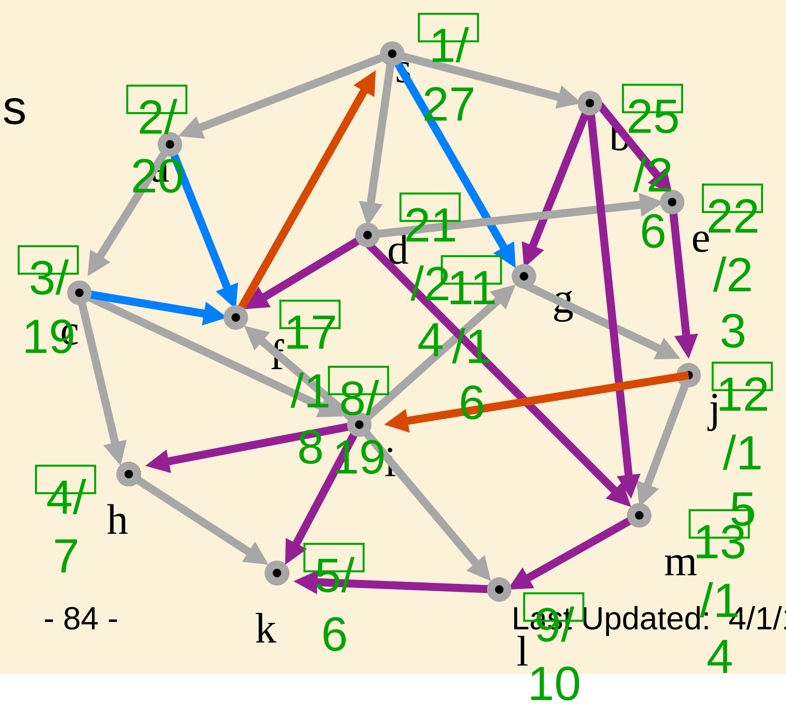
1. ***Tree edges***: Edge (*u*, *v*) is a **tree edge** if *v* was **black** when (*u*, *v*) traversed.

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.

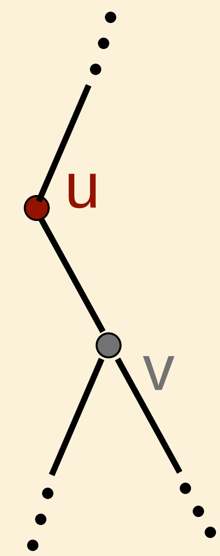


DFS on Undirected Graphs

- 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!**



Applications of Depth-First Search

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)
Precondition: u and z are vertices in a graph
Postcondition: a path from u to z is returned, if one exists
    colour[u] ← RED
    push u onto stack
    if u = z
        return list of elements on stack
    for each v ∈ Adj[u] //explore edge (u,v)
        if color[v] = BLACK
            DFS-Path(v,z)
    colour[u] ← GRAY
    pop u from stack
```

DFS Application 2: Cycle Finding

- The DFS pattern can be used to find a cycle in a graph, if one exists
- We use a stack to keep track of the current path
- If a back edge is encountered, we return the cycle as the contents of the stack

```
DFS-Cycle (u)
Precondition: u is a vertex in a graph G
Postcondition: a cycle reachable from u is returned, of one exists
        colour[u] ← RED
        push u onto stack
        for each v ∈ Adj[u] //explore edge (u,v)
                if color[v] = RED //back edge
                        return list of elements on stack
                else if color[v] = BLACK
                        DFS-Cycle(v)
        colour[u] ← GRAY
        pop u from stack
```

DFS Application 3. Topological Sorting (e.g., putting tasks in linear order)

Note: This topological sorting algorithm is different from the TopologicalSort algorithm given on p.617 of the textbook

DAGs and Topological Ordering

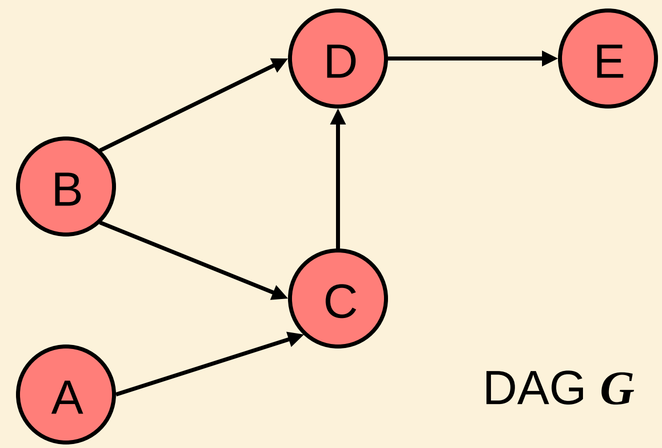
- A directed acyclic graph (DAG) is a digraph that has no directed cycles
- A topological ordering of a digraph is a numbering

$$v_1, \ldots, v_n$$

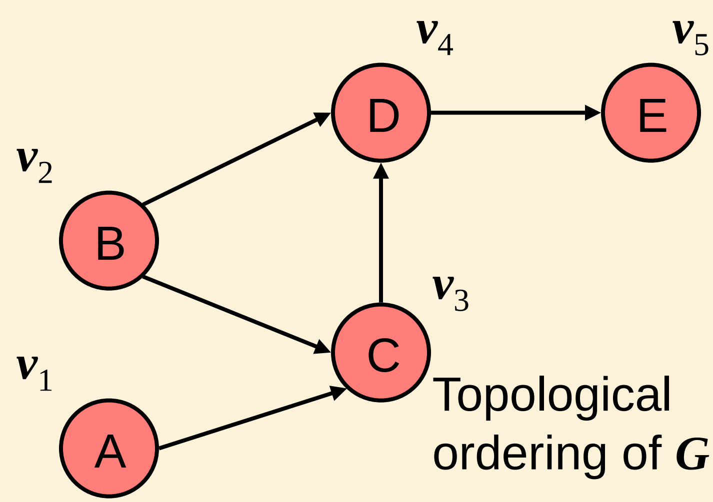
  of the vertices such that for every edge (v_i, v_j), we have $i < j$
- Example: in a task scheduling digraph, a topological ordering is a task sequence that satisfies the precedence constraints

Theorem

A digraph admits a topological ordering if and only if it is a DAG

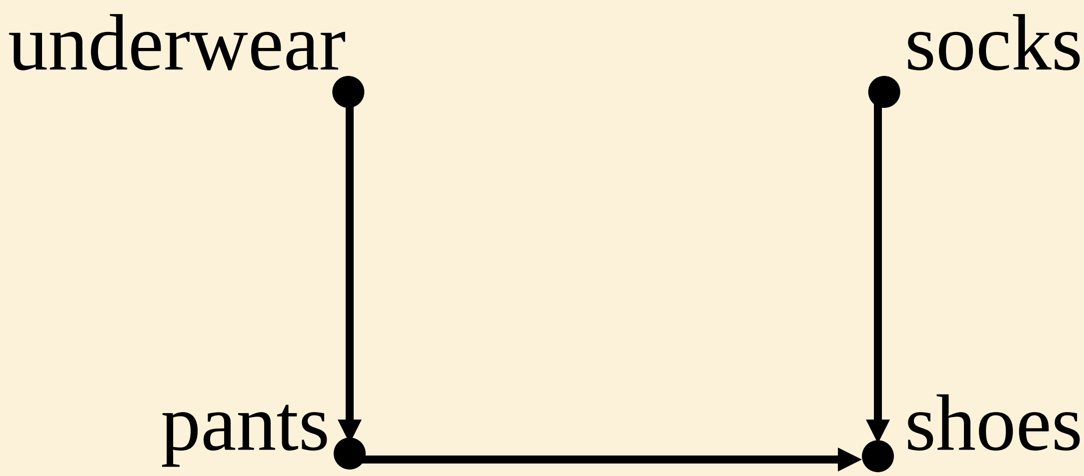


DAG G



Topological ordering of G

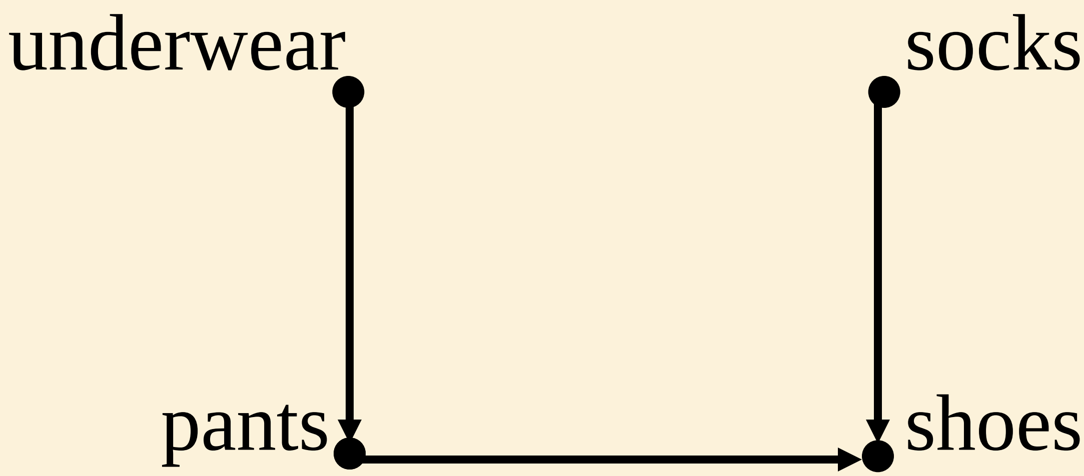
Topological (Linear) Order



underwear
pants
socks
shoes

socks
underwear
pants
shoes

Topological (Linear) Order



Invalid Order

~~socks~~
~~shoes~~
~~pants~~
~~underwear~~

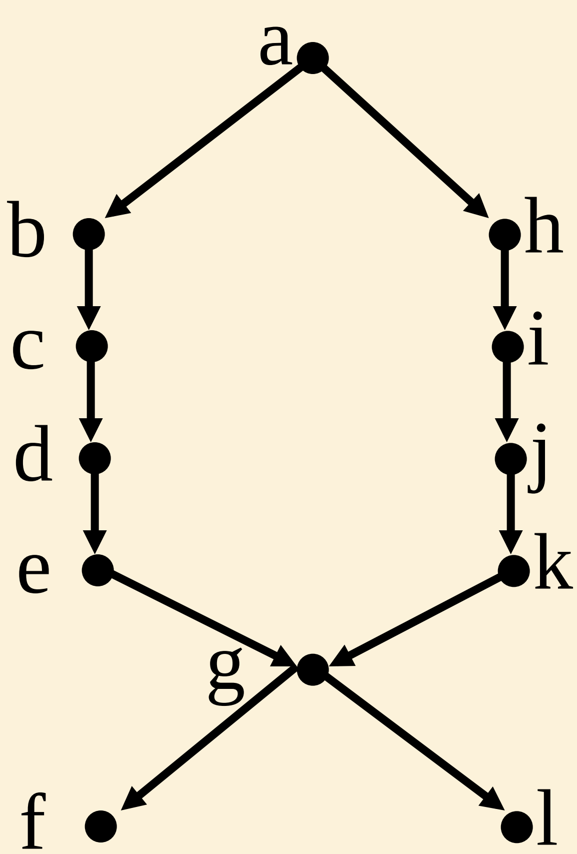
Algorithm for Topological Sorting

- Note: This algorithm is different than the one in Goodrich-Tamassia

Method TopologicalSort(***G***)

    H* 🡐 *G // Temporary copy of ***G***
    n* 🡐 *G.numVertices()
    while ***H*** is not empty **do**
        Let ***v*** be a vertex with no outgoing edges
        Label ***v* 🡐 *n***
        ***n* 🡐 *n* - 1**
        Remove *v* from ***H*** ***//as well as edges involving v***

Linear Order



Pre-Condition:

A Directed Acyclic Graph (DAG)

Post-Condition:

Find one valid linear order

Algorithm:

- Find a terminal node (sink).
- Put it last in sequence.
- Delete from graph & repeat

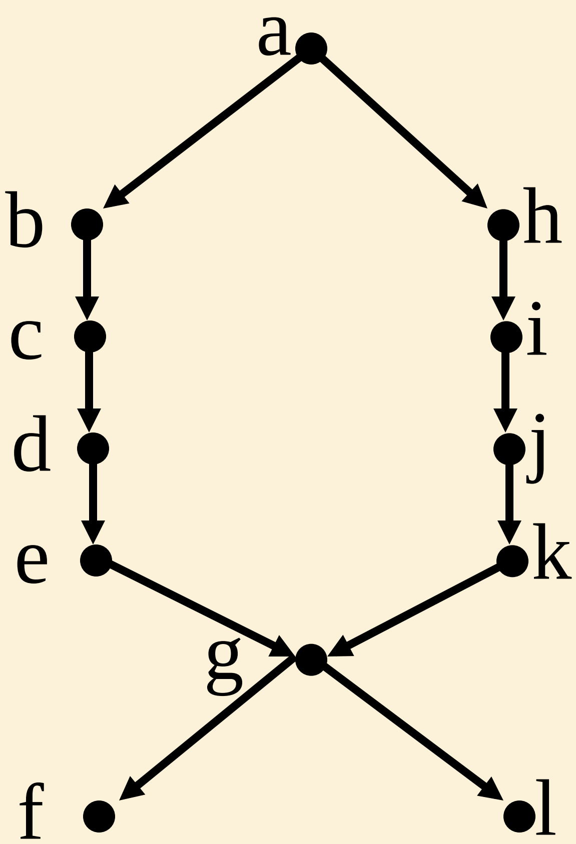
} O(|V|)

Running time: $\sum_{i=1}^{|V|} i = O\left(|V|^2\right)$

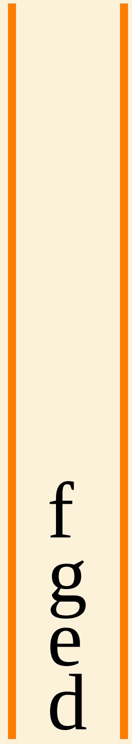
…..] Can we do better?

Linear Order

Alg: DFS



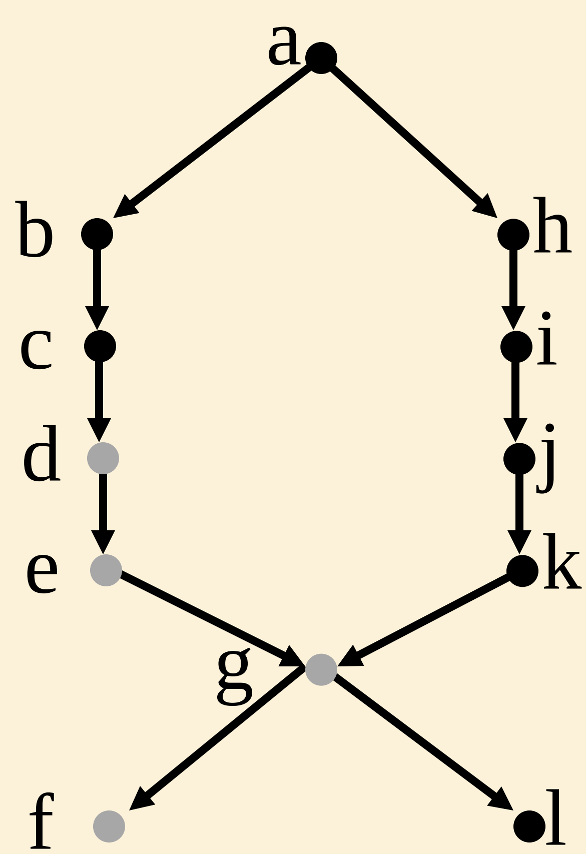
Found
Not Handled
Stack



….. f

Linear Order

Alg: DFS



Found
Not Handled
Stack

l
g
e
d

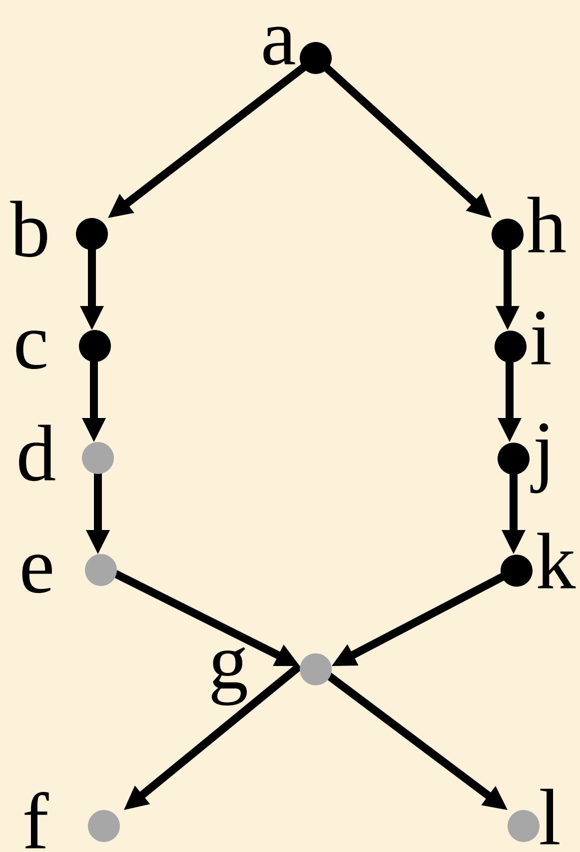
When node is popped off stack, insert at front of linearly-ordered “to do” list.

Linear Order:

….. f

Linear Order

Alg: DFS



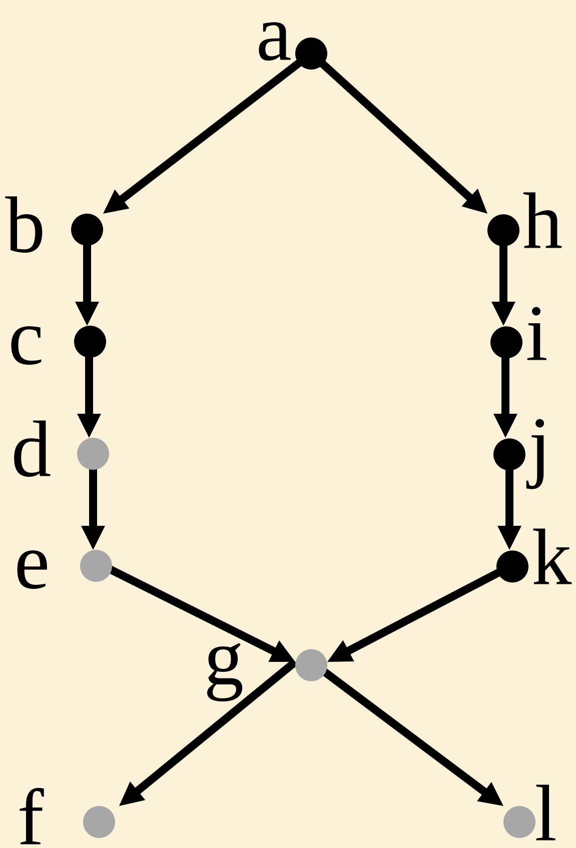
Found
Not Handled
Stack

g
e
d

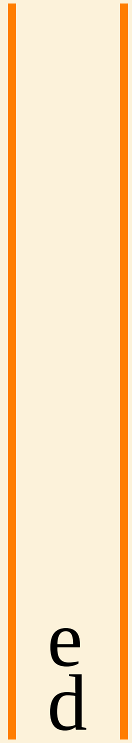
Linear Order:

l,f

Linear Order
Alg: DFS



Found
Not Handled
Stack

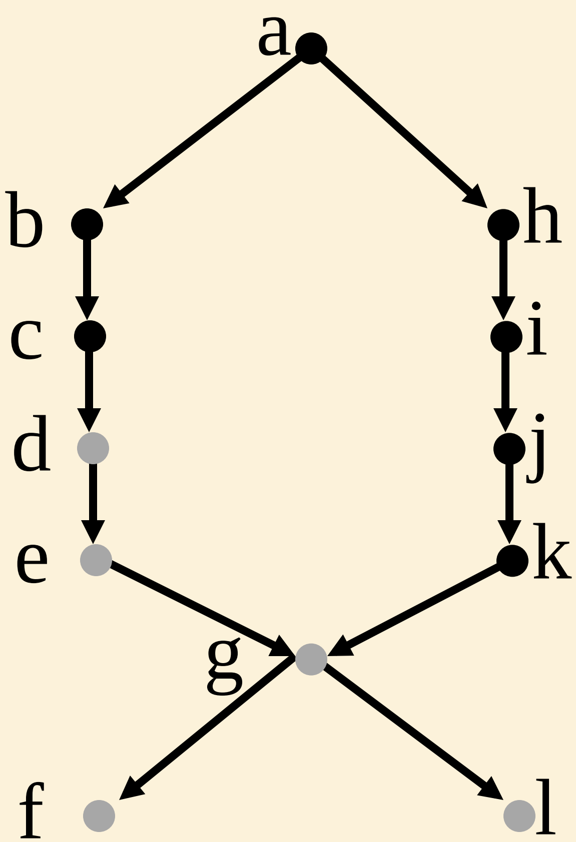


Linear Order:

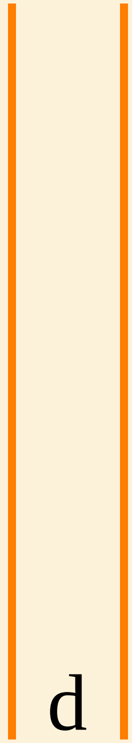
g,l,f

Linear Order

Alg: DFS



Found
Not Handled
Stack

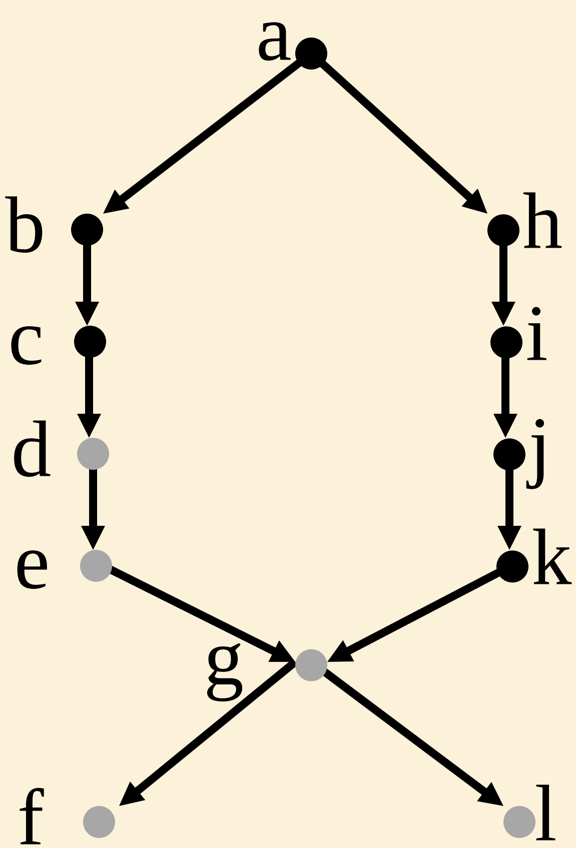


Linear Order:

e,g,l,f

Linear Order

Alg: DFS



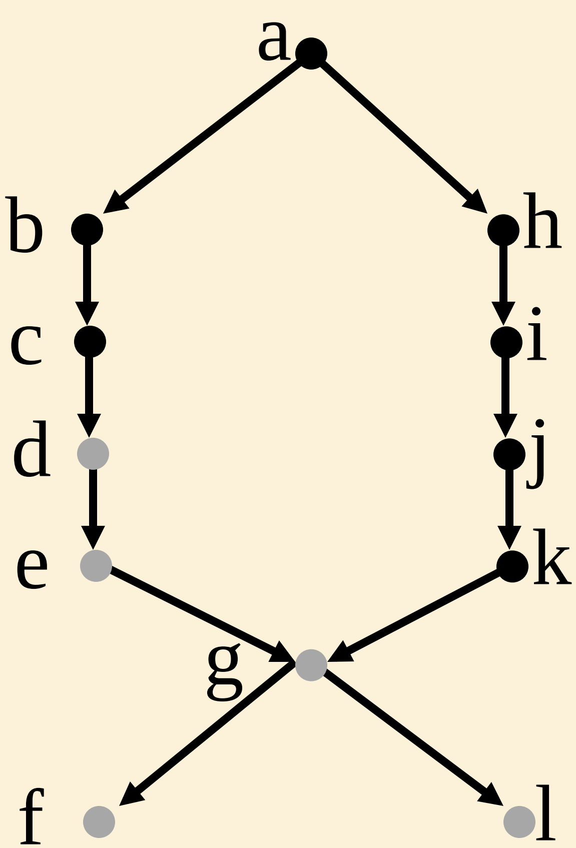
Found
Not Handled
Stack

Linear Order:

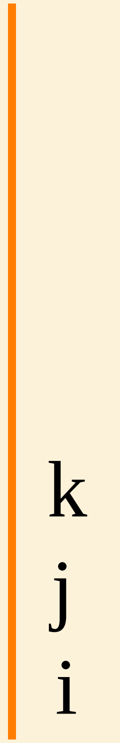
d,e,g,l,f

Linear Order

Alg: DFS



Found
Not Handled
Stack

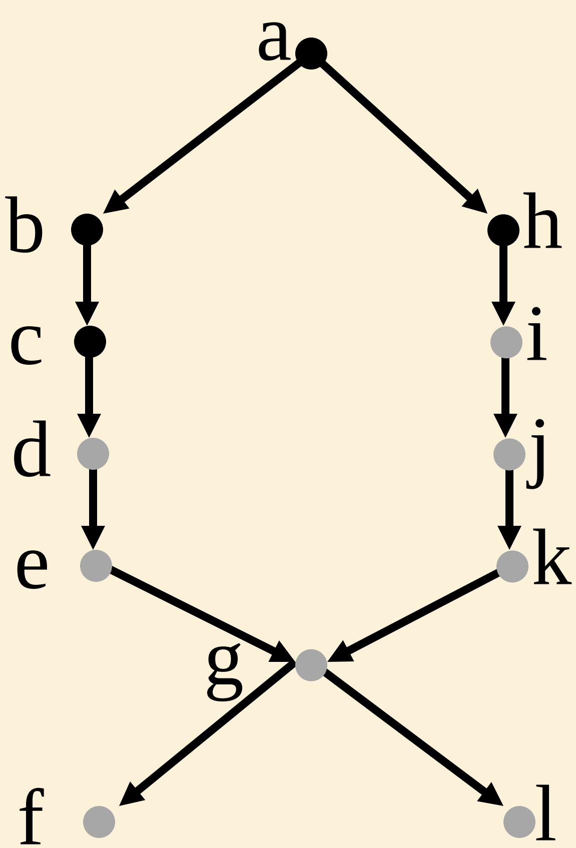


Linear Order:

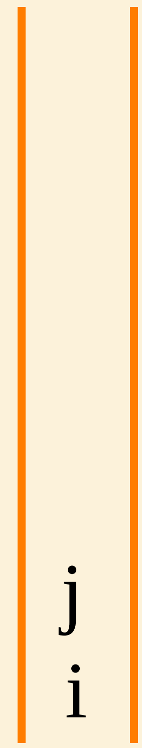
d,e,g,l,f

Linear Order

Alg: DFS



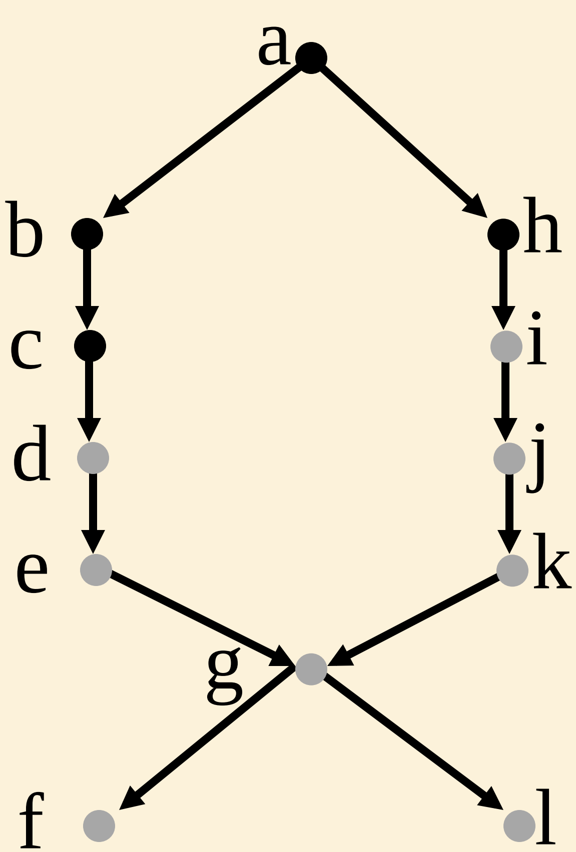
Found
Not Handled
Stack



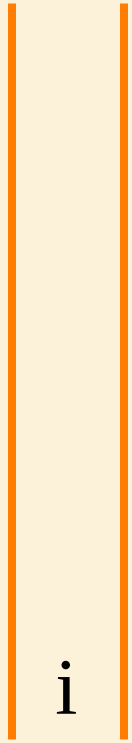
Linear Order: k,d,e,g,l,f

Linear Order

Alg: DFS



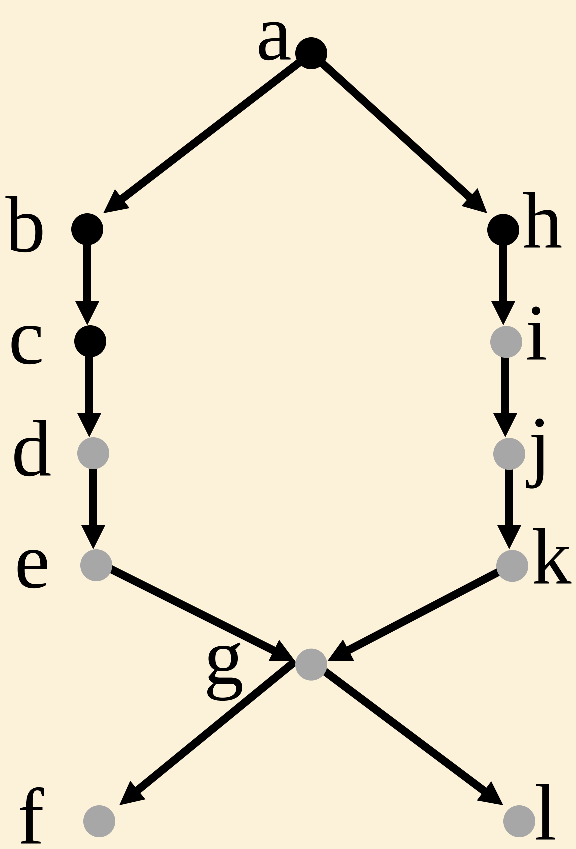
Found
Not Handled
Stack

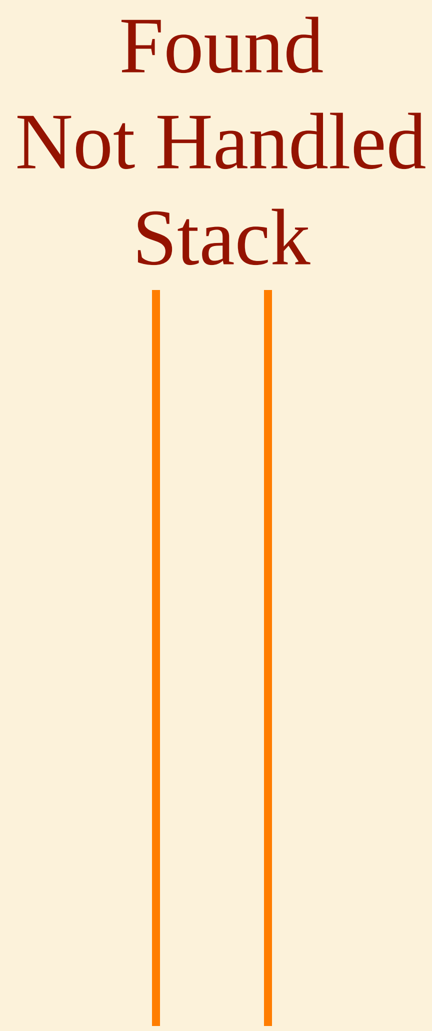


Linear Order: j,k,d,e,g,l,f

Linear Order

Alg: DFS

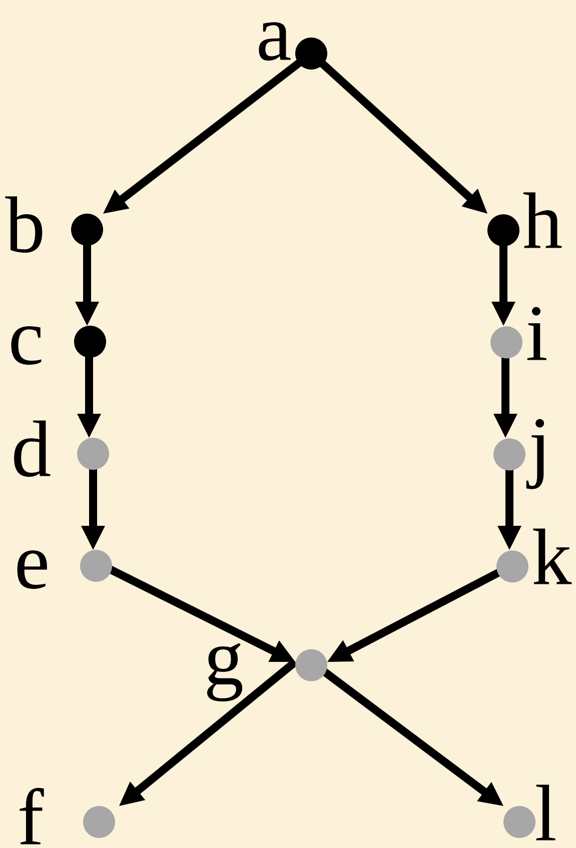




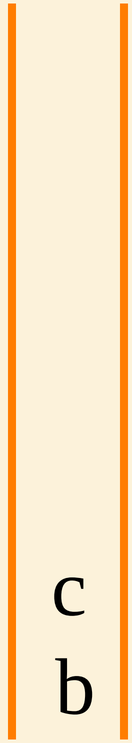
Linear Order: i,j,k,d,e,g,l,f

Linear Order

Alg: DFS



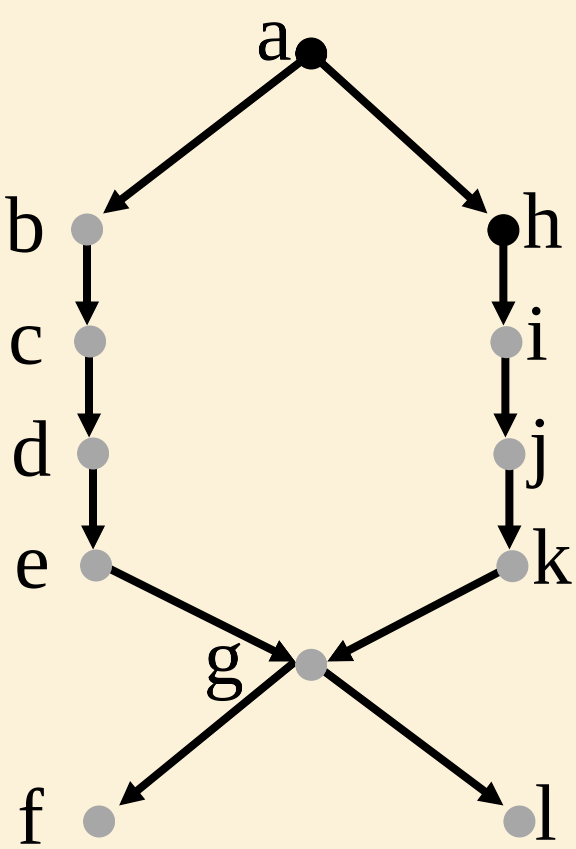
Found
Not Handled
Stack



Linear Order: i,j,k,d,e,g,l,f

Linear Order

Alg: DFS



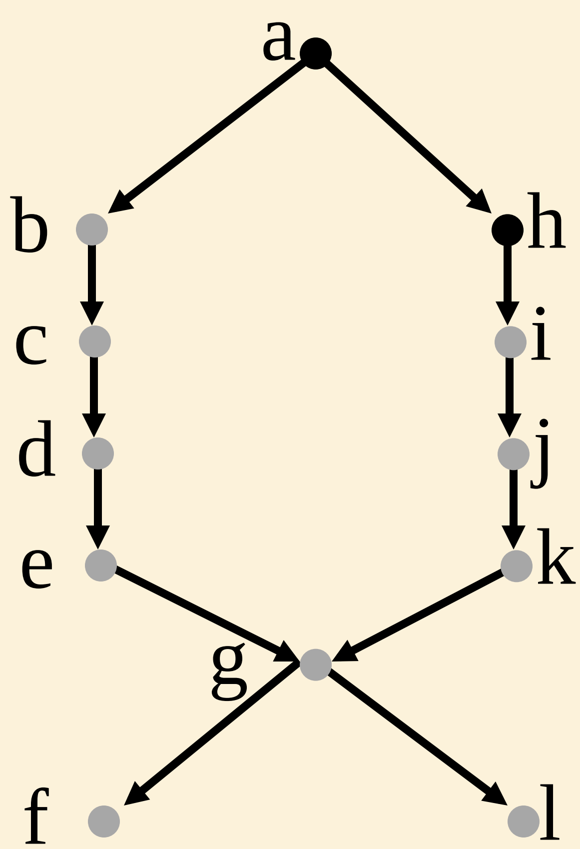
Found
Not Handled
Stack

b

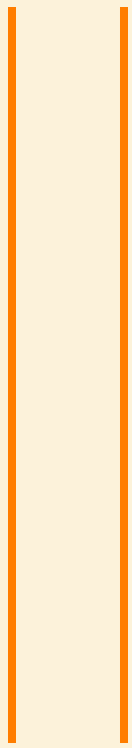
Linear Order: c,i,j,k,d,e,g,l,f

Linear Order

Alg: DFS



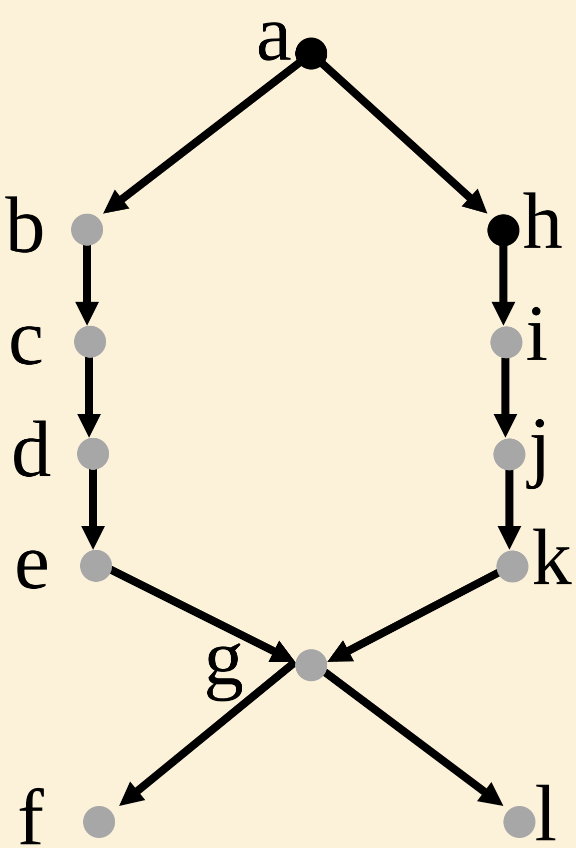
Found
Not Handled
Stack



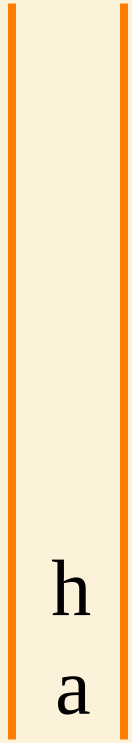
Linear Order: b,c,i,j,k,d,e,g,l,f

Linear Order

Alg: DFS



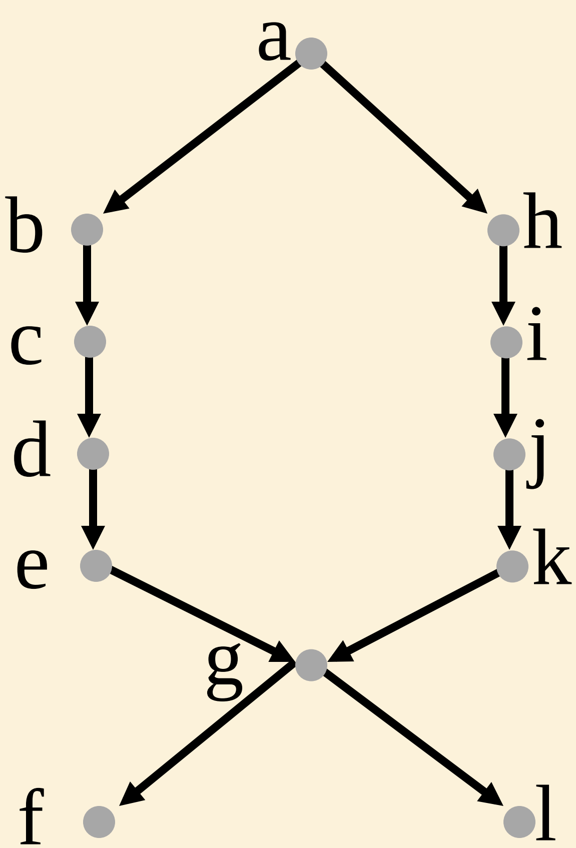
Found
Not Handled
Stack



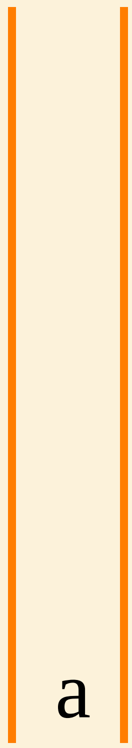
Linear Order: b,c,i,j,k,d,e,g,l,f

Linear Order

Alg: DFS



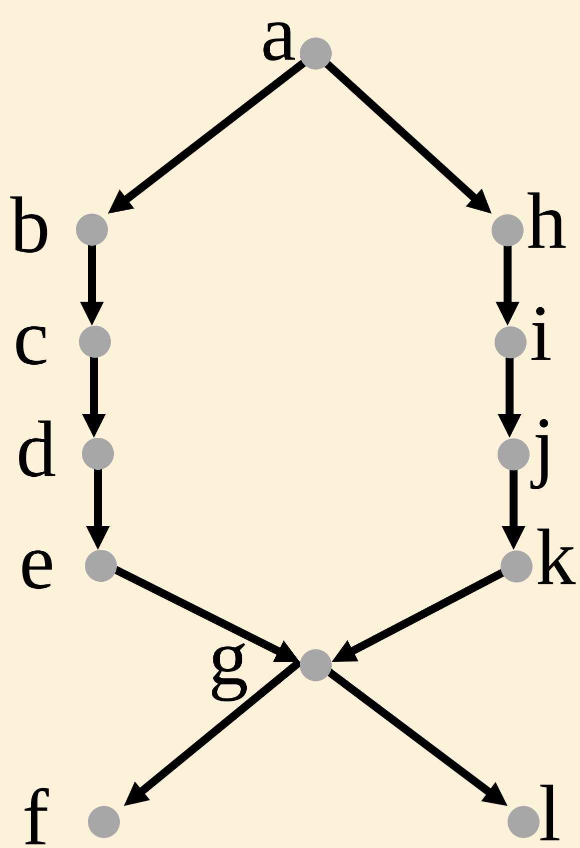
Found
Not Handled
Stack



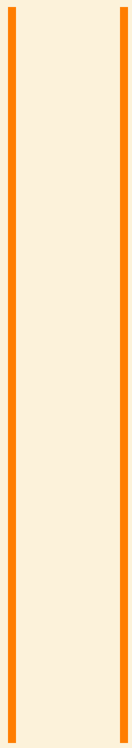
Linear Order: h,b,c,i,j,k,d,e,g,l,f

Linear Order

Alg: DFS



Found
Not Handled
Stack



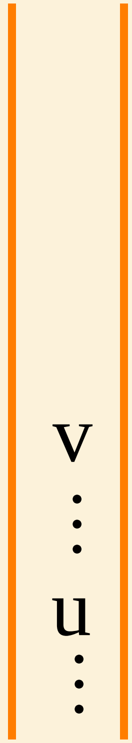
Linear Order: a,h,b,c,i,j,k,d,e,g,l,f Done!

DFS Algorithm for Topologial Sort

- Makes sense. But how do we prove that it works?

Linear Order

Found
Not Handled
Stack



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. ☺

u $\longrightarrow$ v

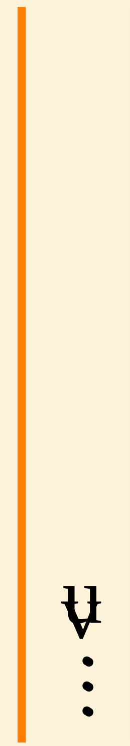
u…v…

Linear Order

Found
Not Handled
Stack

Proof: Consider each edge

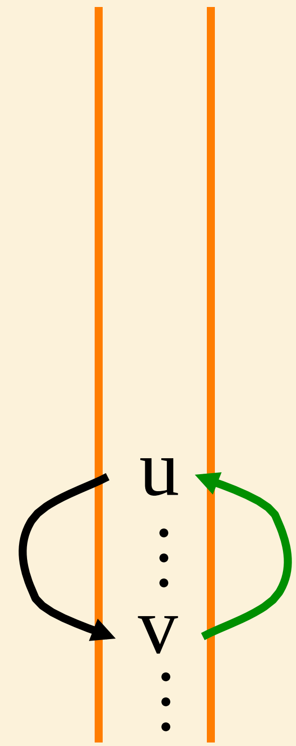
- Case 1: u goes on stack first before v.
- Case 2: v goes on stack first before u.
  v comes off before u goes on.
    - v goes after u in order. ☺



u…v…

Linear Order

Found
Not Handled
Stack



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 ☺

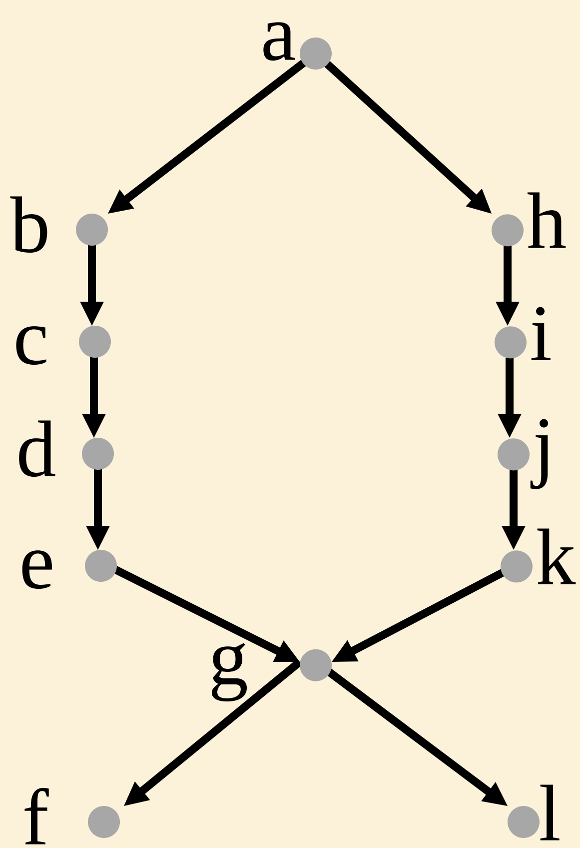
The nodes in the stack form a path starting at s.

$u \bullet \longrightarrow \bullet v$

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!

DFS Application 3. Topological Sort

Topological-Sort(G)

Precondition: G is a graph

Postcondition: all vertices in G have been pushed onto stack in reverse linear order

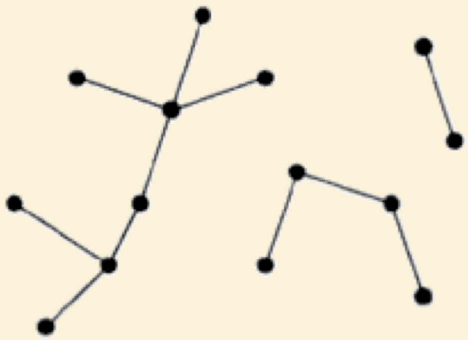
for each vertex u $\in V[G]$

  color[u] = BLACK //initialize vertex

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

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

    Topological-Sort-Visit(u)



DFS Application 3. Topological Sort

Topological-Sort-Visit (*u*)

Precondition: vertex *u* is undiscovered

Postcondition: u and all vertices reachable from *u* have been pushed onto stack in reverse linear order

colour[*u*] $\leftarrow$ RED

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

if color[*v*] = BLACK

Topological-Sort-Visit(*v*)

push *u* onto stack

$colour[u] \leftarrow GRAY$

Breadth-First Search

Breadth-First Search

- Breadth-first search (BFS) is a general technique for traversing a graph
- A BFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
- BFS on a graph with $|V|$ vertices and $|E|$ edges takes $O(|V|+|E|)$ time
- BFS can be further extended to solve other graph problems
  - Find and report a path with the minimum number of edges between two given vertices
  - Find a simple cycle, if there is one

BFS Algorithm Pattern

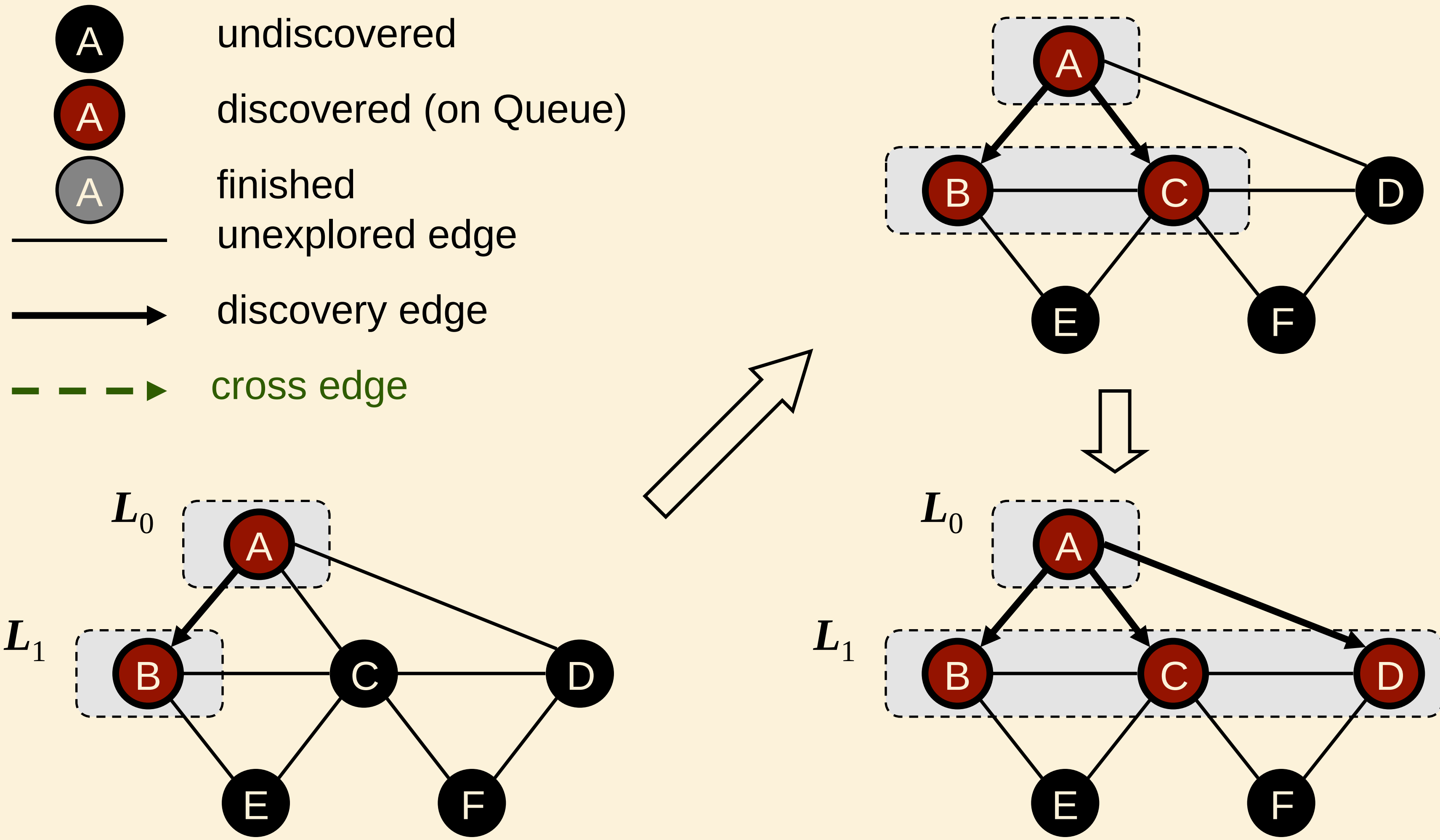
BFS(G,s)

Precondition: G is a graph, s is a vertex in G

Postcondition: all vertices in G reachable from s have been visited

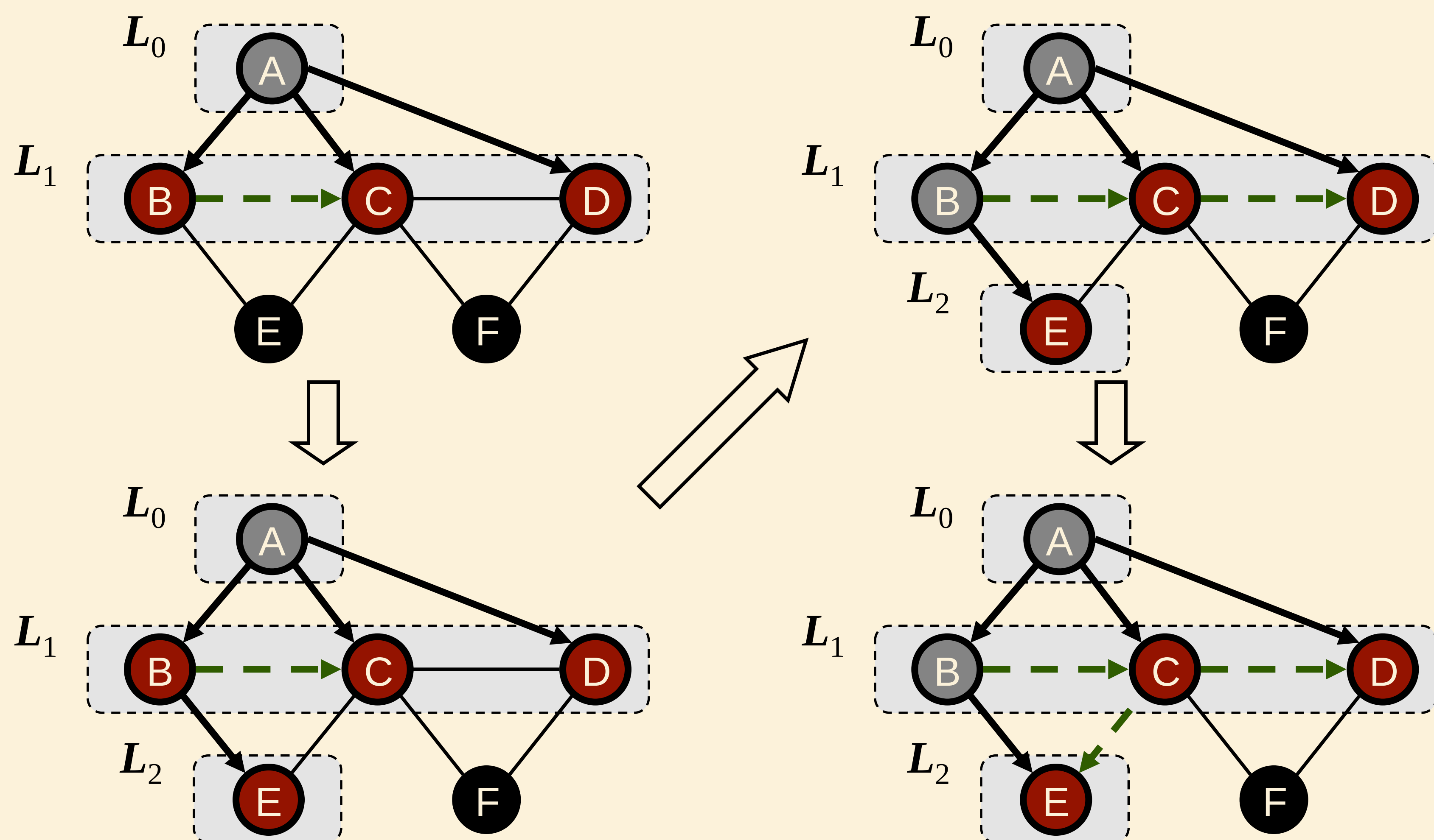
```
for each vertex u ∈ V[G]
        color[u] ← BLACK //initialize vertex
colour[s] ← RED
Q.enqueue(s)
while Q ≠ ∅
        u ← Q.dequeue()
        for each v ∈ Adj[u] //explore edge (u,v)
                if color[v] = BLACK
                        colour[v] ← RED
                        Q.enqueue(v)
        colour[u] ← GRAY
```

BFS Example

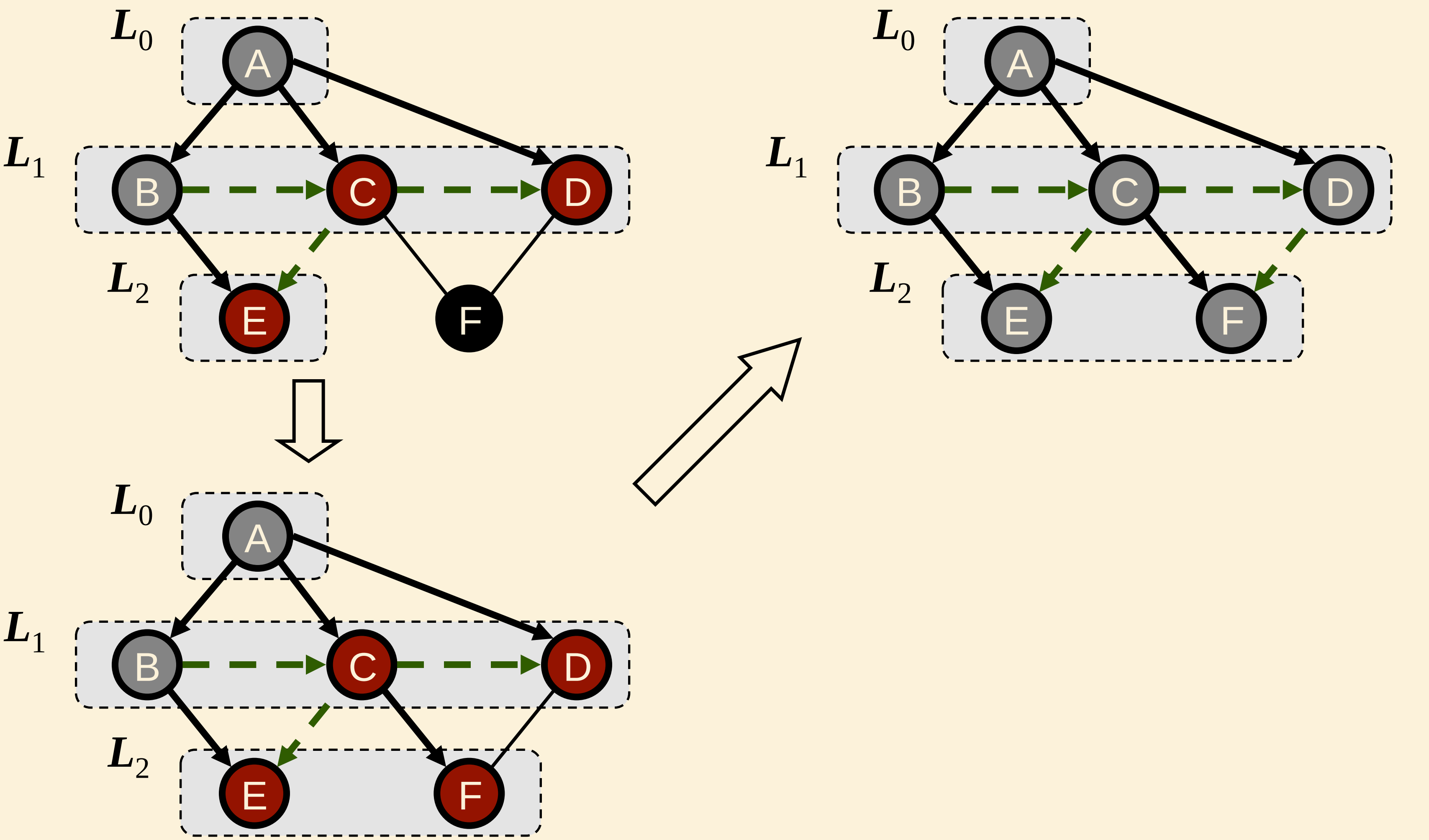


A
undiscovered
A
discovered (on Queue)
A
finished
unexplored edge
discovery edge
cross edge
L_0
L_1
A
B
C
D
E
F

BFS Example (cont.)



BFS Example (cont.)



Properties

Notation

G_s: connected component of s

Property 1

$BFS(G, s)$ visits all the vertices and edges of G_s

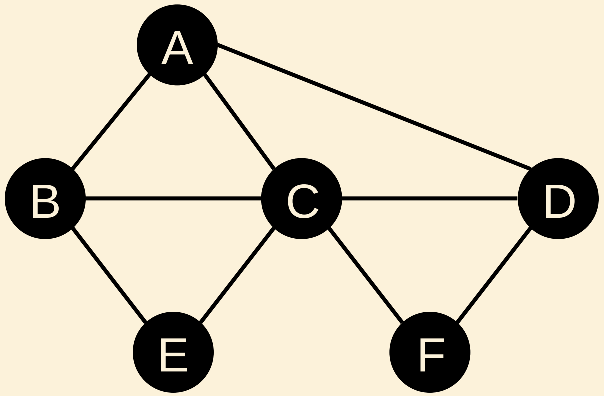
Property 2

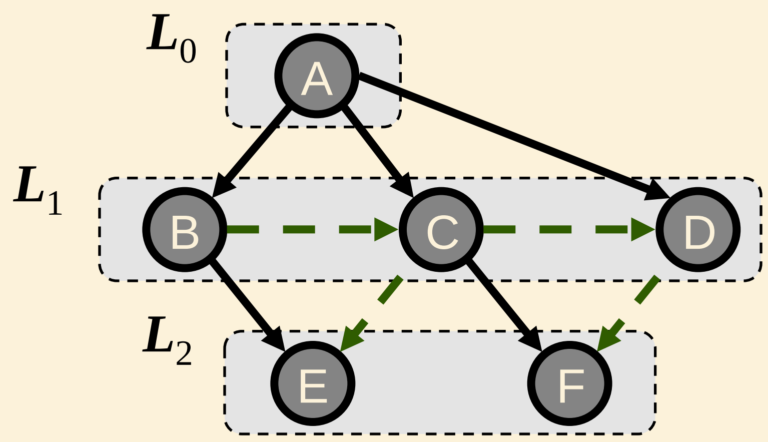
The discovery edges labeled by $BFS(G, s)$ form a spanning tree T_s of G_s

Property 3

For each vertex v in L_i

- The path of T_s from s to v has i edges
- Every path from s to v in G_s has at least i edges





Analysis

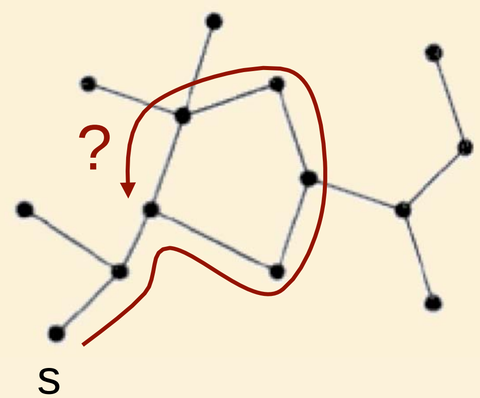
- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled three times
  - once as BLACK (undiscovered)
  - once as RED (discovered, on queue)
  - once as GRAY (finished)
- Each edge is considered twice (for an undirected graph)
- Each vertex is inserted once into a sequence L_i
- Thus BFS runs in $O(|V|+|E|)$ time provided the graph is represented by an adjacency list structure

Applications

- BFS traversal can be specialized to solve the following problems in $O(|V|+|E|)$ time:
  - Compute the connected components of G
  - Compute a spanning forest of G
  - Find a simple cycle in G, or report that G is a forest
  - Given two vertices of G, find a path in G between them with the minimum number of edges, or report that no such path exists

Application: Shortest Paths on an Unweighted Graph

- 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
  - Analogous to level order traversal for graphs
  - 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.

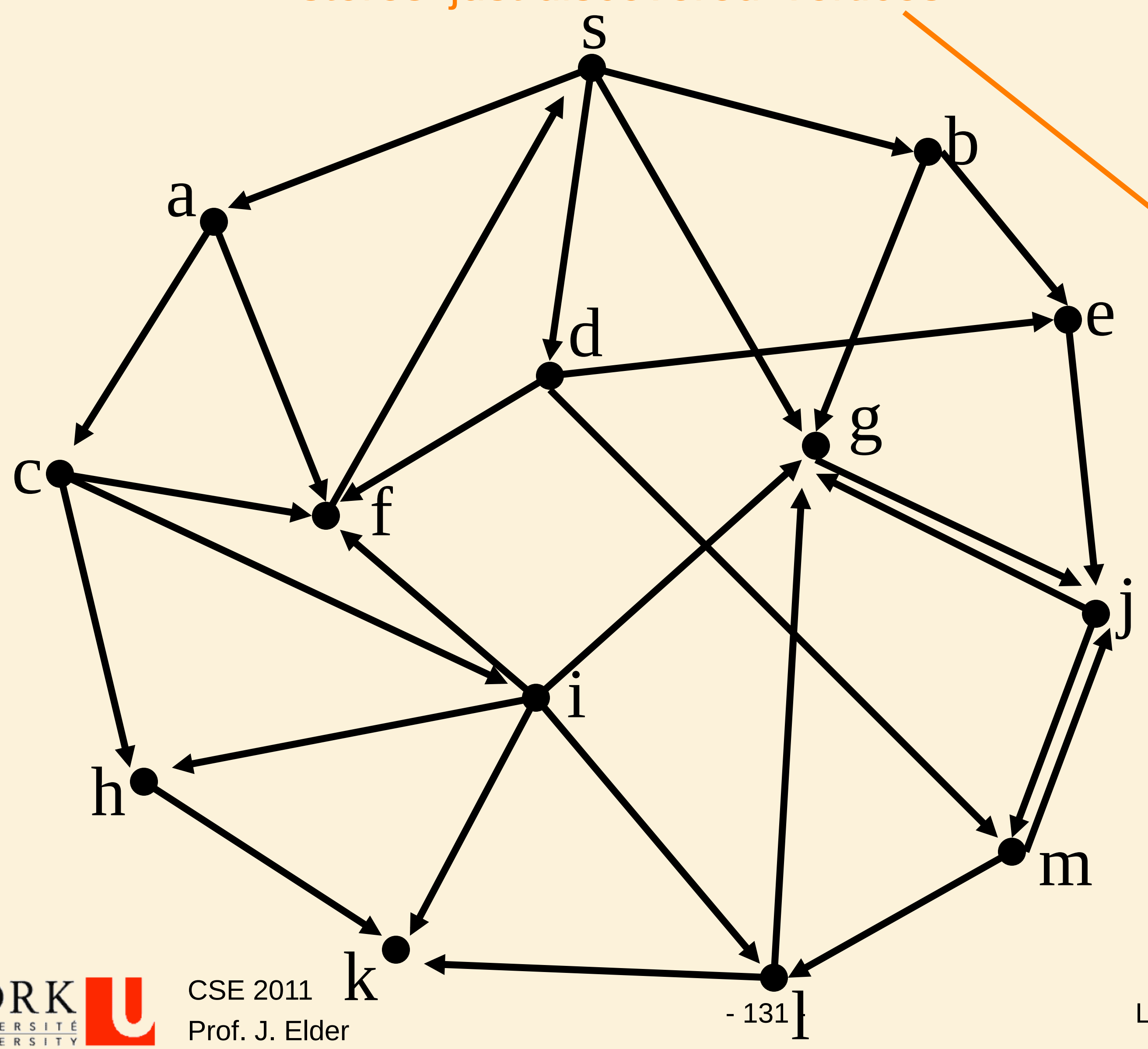
- Idea: send out search 'wave' from s.
- Keep track of progress by colouring vertices:
  - **Undiscovered** vertices are coloured **black**
  - **Just discovered** vertices (on the wavefront) are coloured **red.**
  - **Previously discovered** vertices (behind wavefront) are coloured **grey.**

BFS Algorithm with Distances and Predecessors

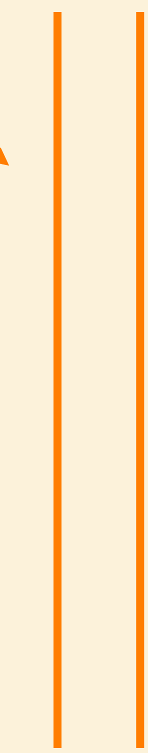
```
BFS(G,s)
Precondition: G is a graph, s is a vertex in G
Postcondition: d[u] = shortest distance δ[u] and
π[u] = predecessor of u on shortest paths from s to each vertex u in G
      for each vertex u ∈ V[G]
            d[u] ← ∞
            π[u] ← null
            color[u] = BLACK //initialize vertex
      colour[s] ← RED
      d[s] ← 0
      Q.enqueue(s)
      while Q ≠ ∅
            u ← Q.dequeue()
            for each v ∈ Adj[u] //explore edge (u,v)
                  if color[v] = BLACK
                        colour[v] ← RED
                        d[v] ← d[u] + 1
                        π[v] ← u
                        Q.enqueue(v)
            colour[u] ← GRAY
```

BFS

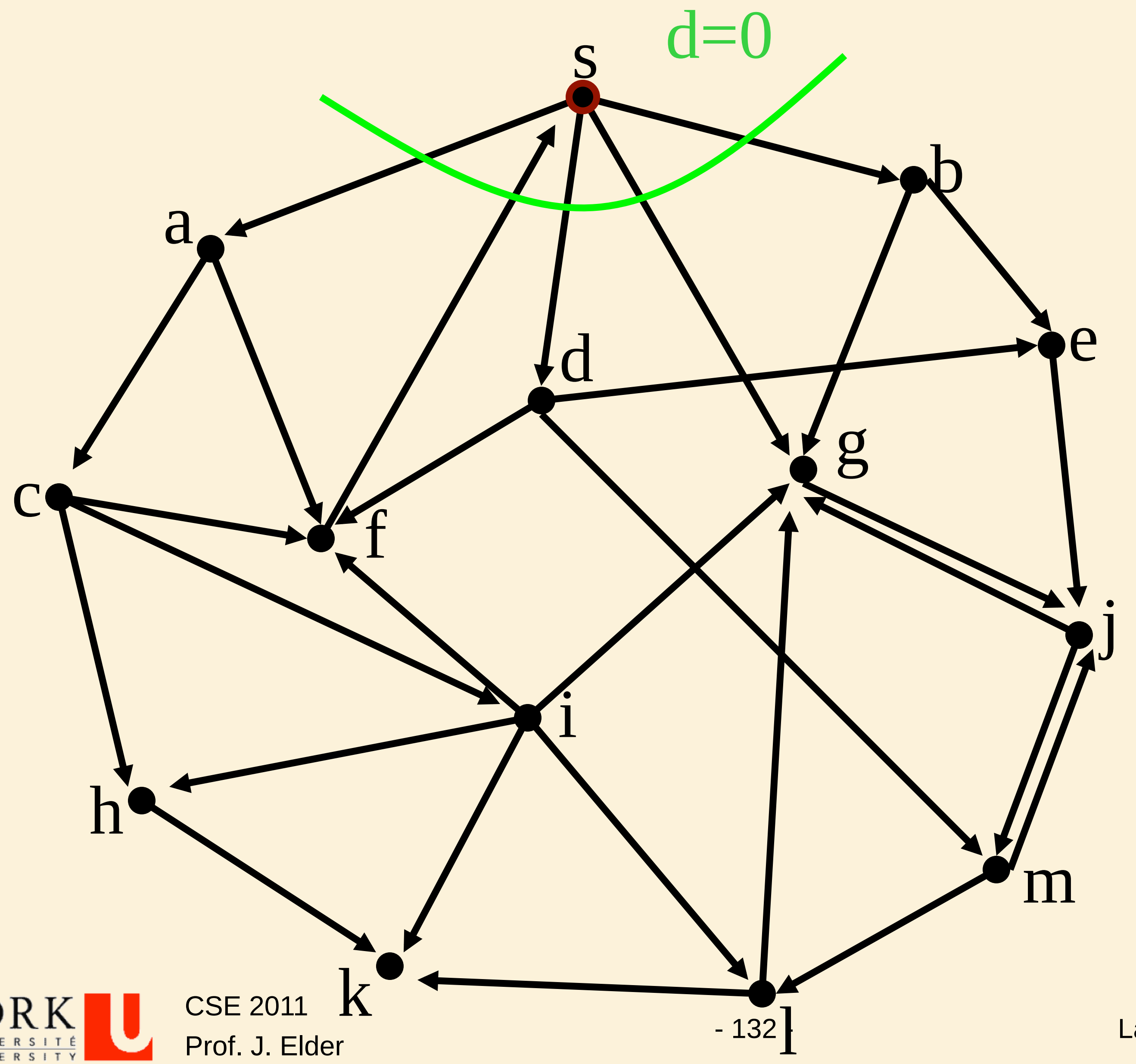
First-In First-Out (FIFO) queue
stores ‘just discovered’ vertices



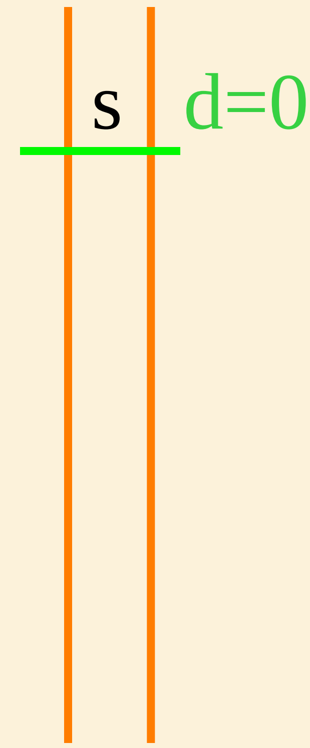
Found
Not Handled
Queue



BFS

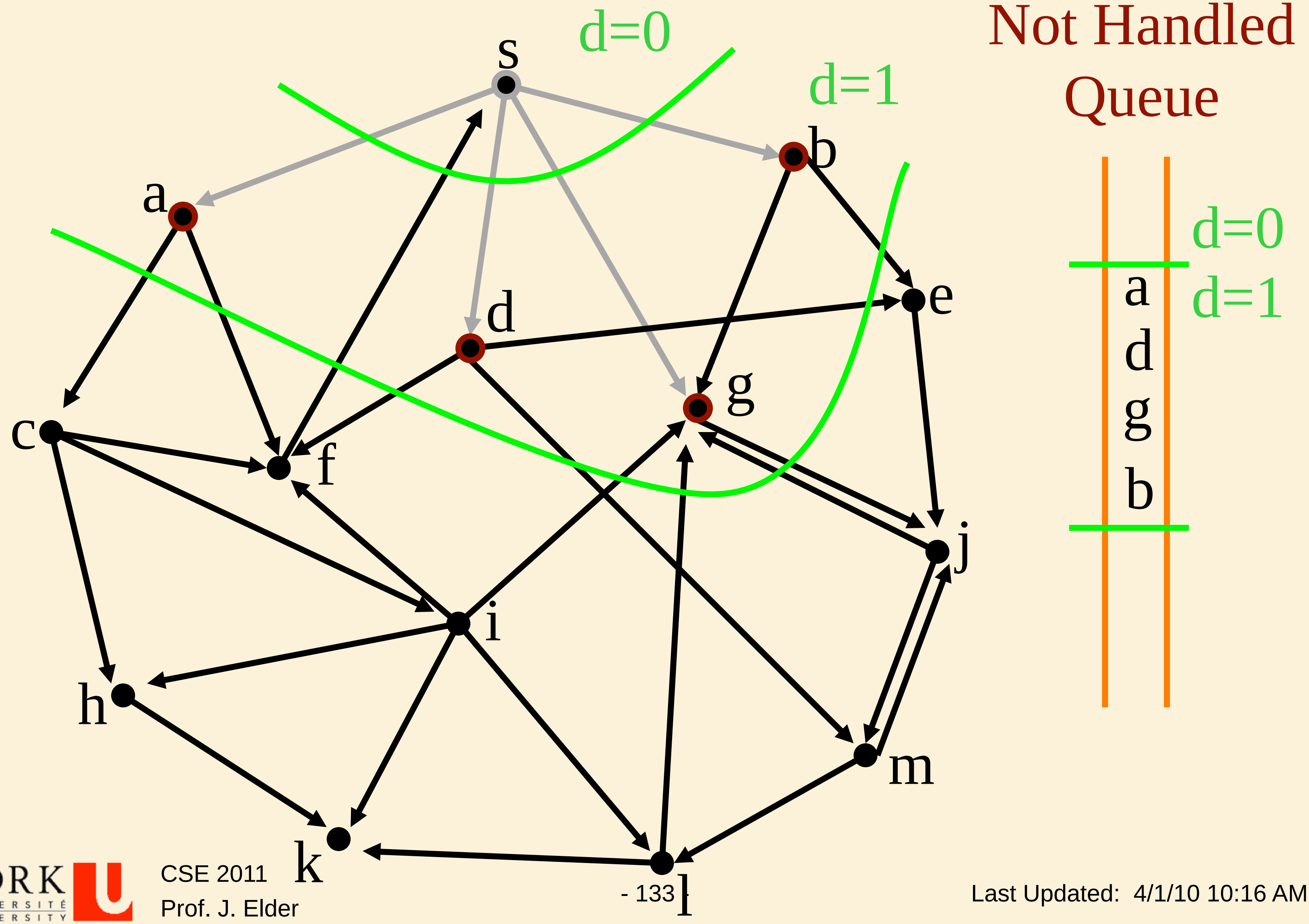


Found
Not Handled
Queue

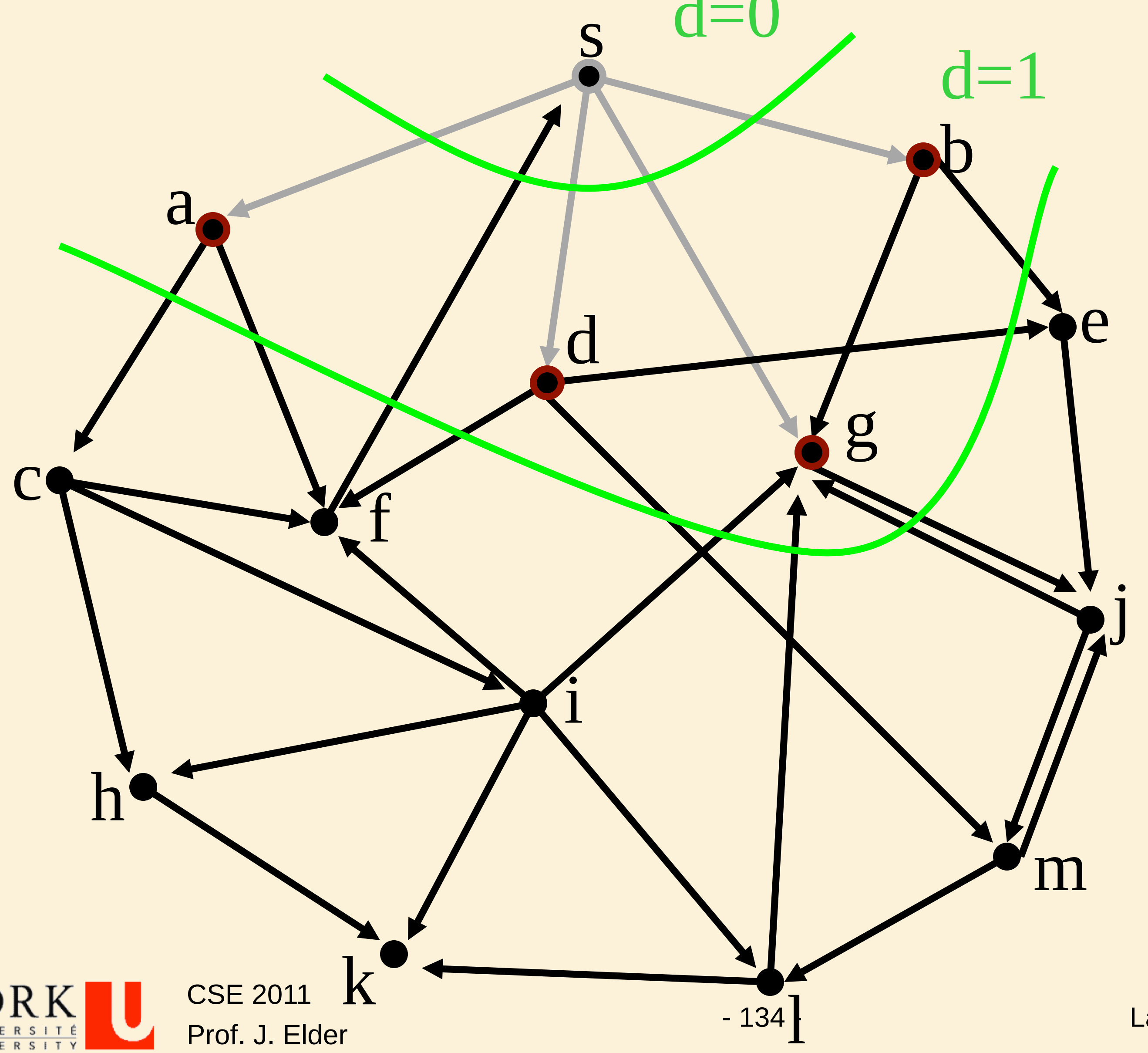


BFS

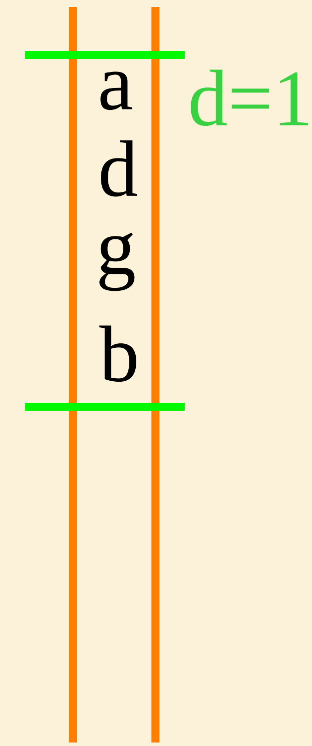
Found
Not Handled
Queue



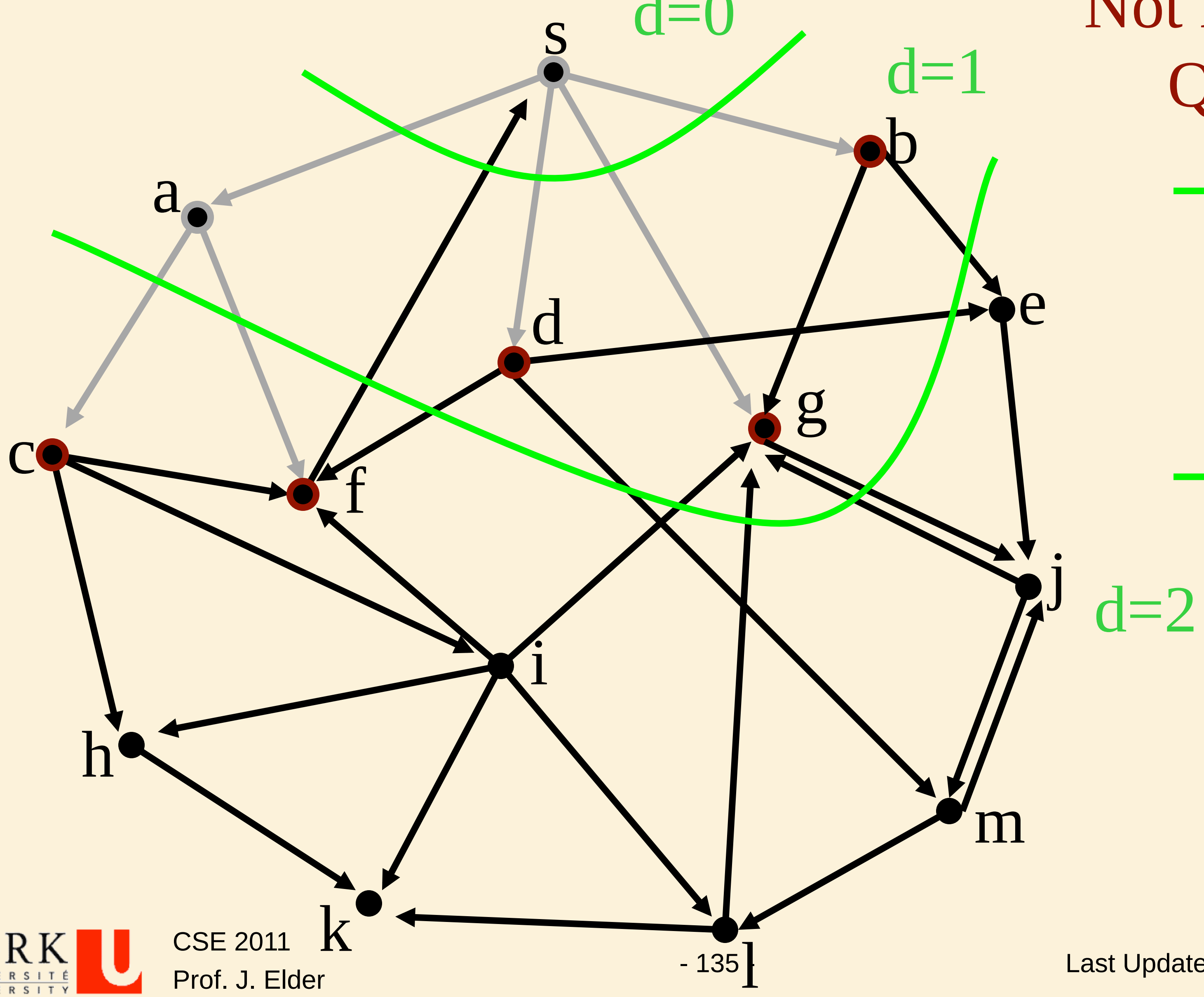
BFS



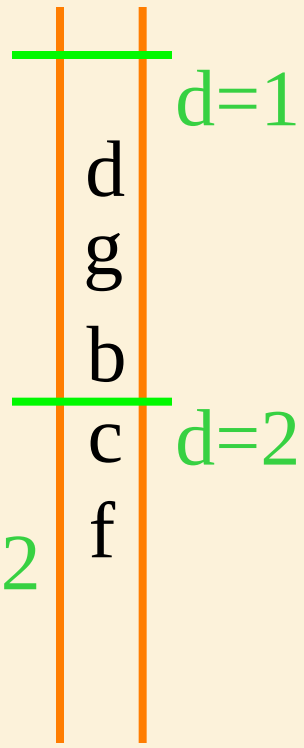
Found
Not Handled
Queue



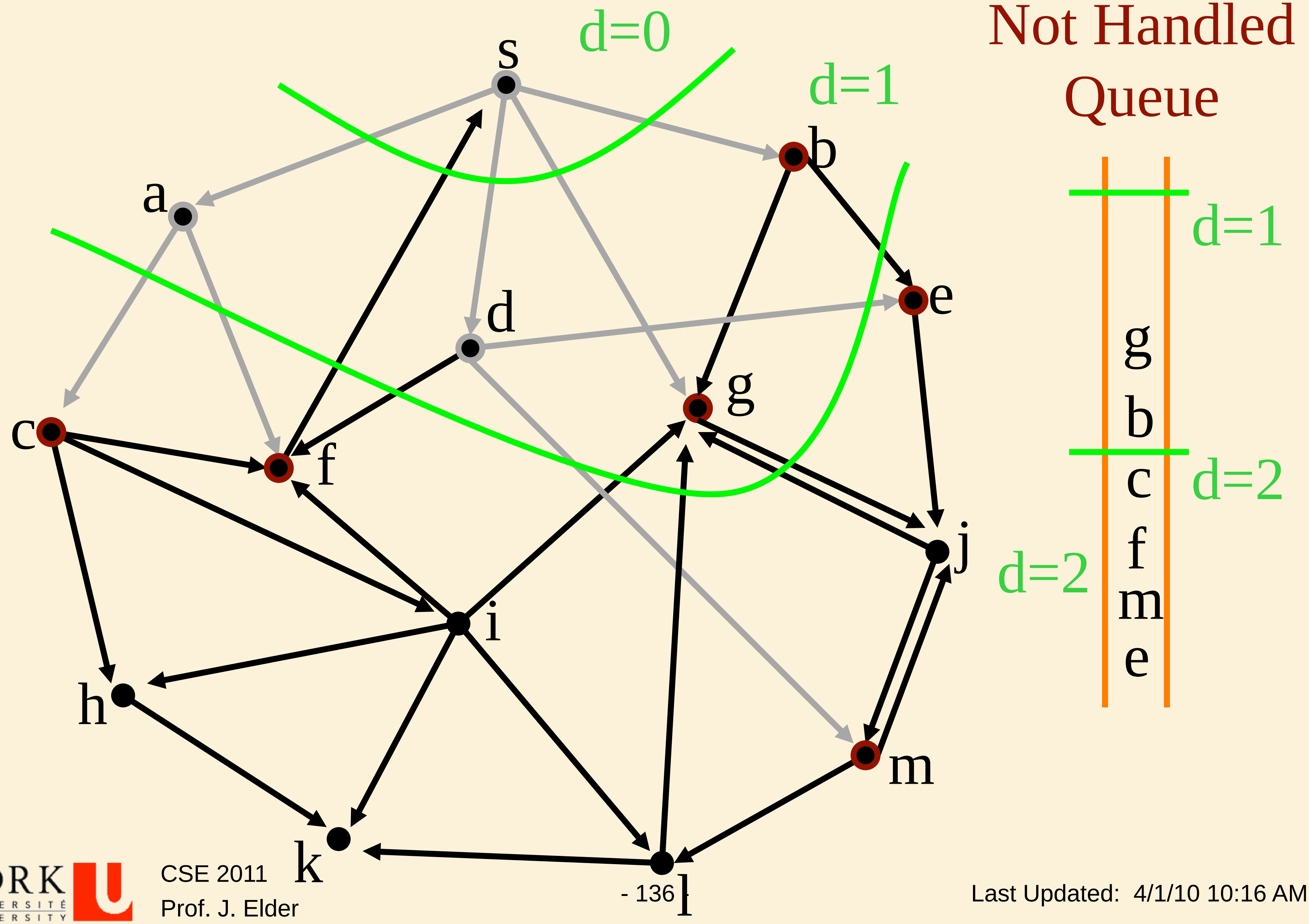
BFS



Found
Not Handled
Queue

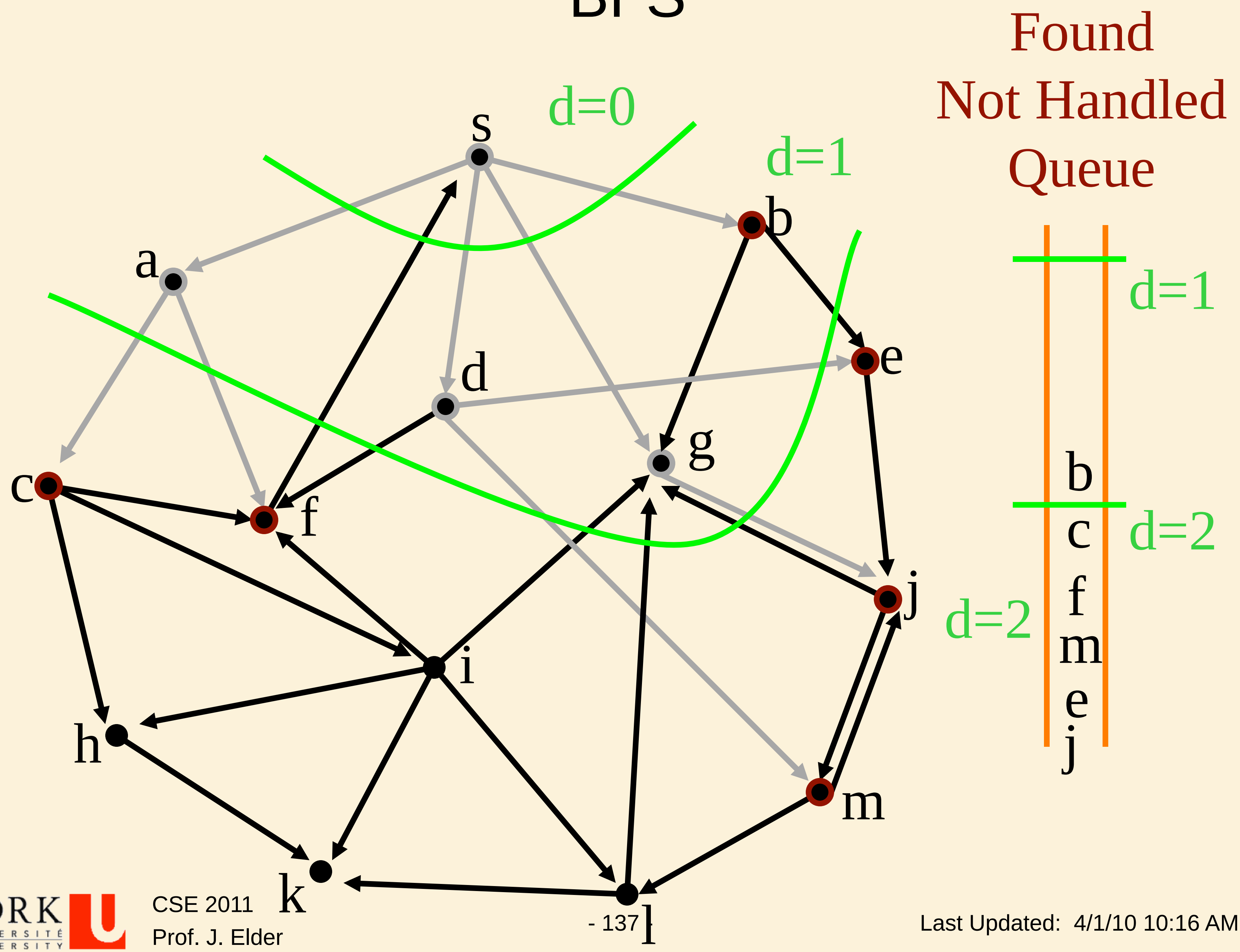


BFS



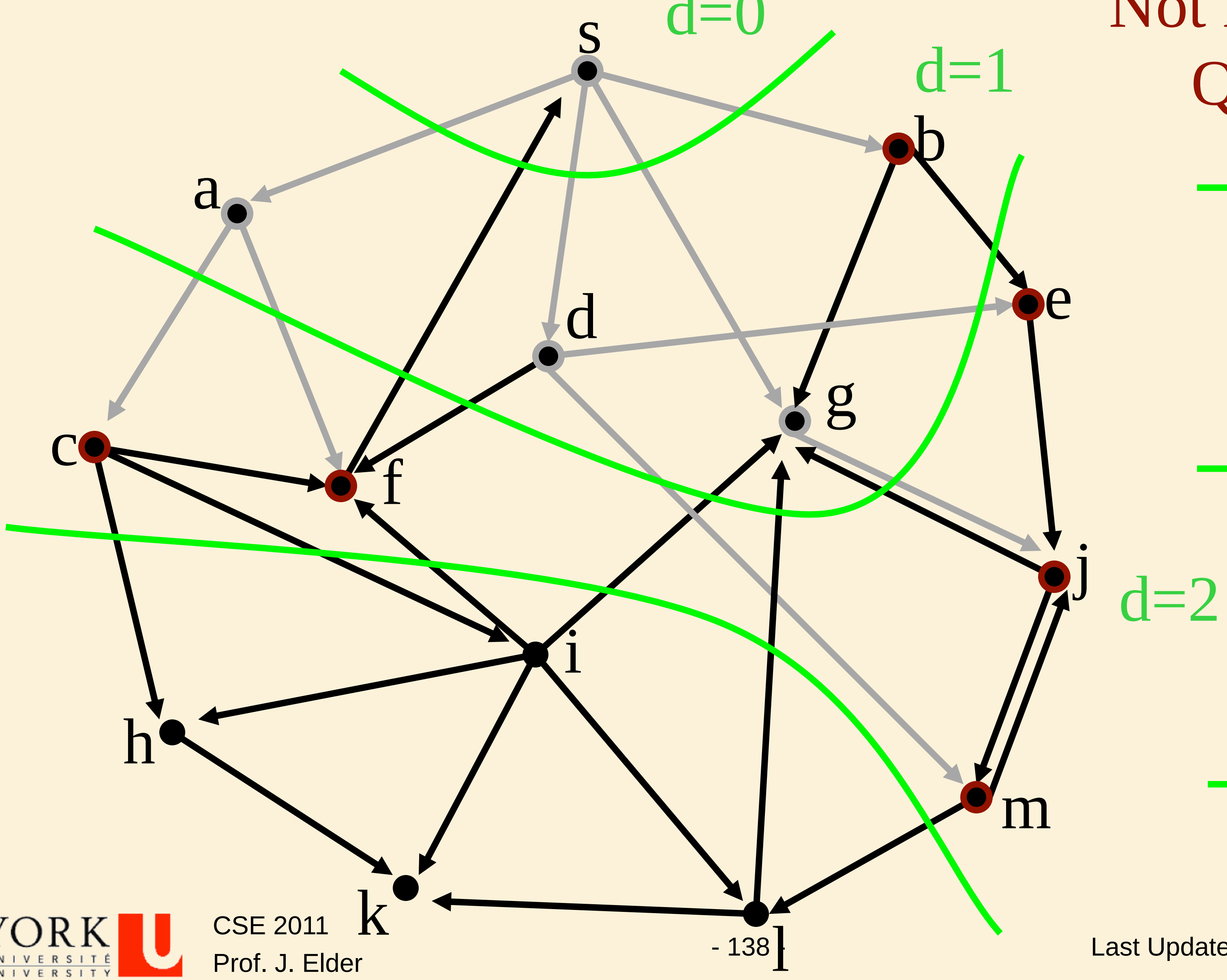
d=0
d=1
d=2
s
a
b
c
d
e
f
g
h
i
j
k
l
m
Found
Not Handled
Queue
d=1
g
b
d=2
c
f
m
e

BFS

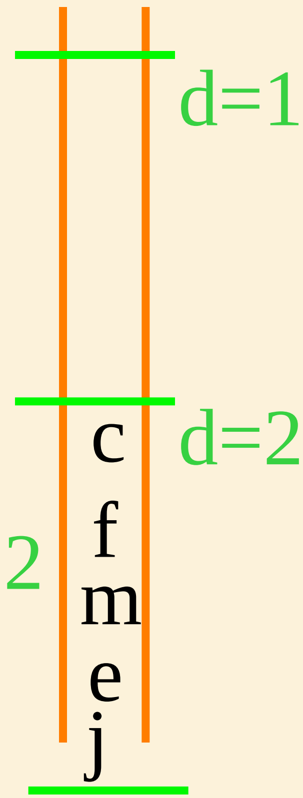


Found
Not Handled
Queue
d=0
d=1
d=1
d=2
d=2
s
a
b
c
d
e
f
g
h
i
j
k
l
m
b
c
f
m
e
j

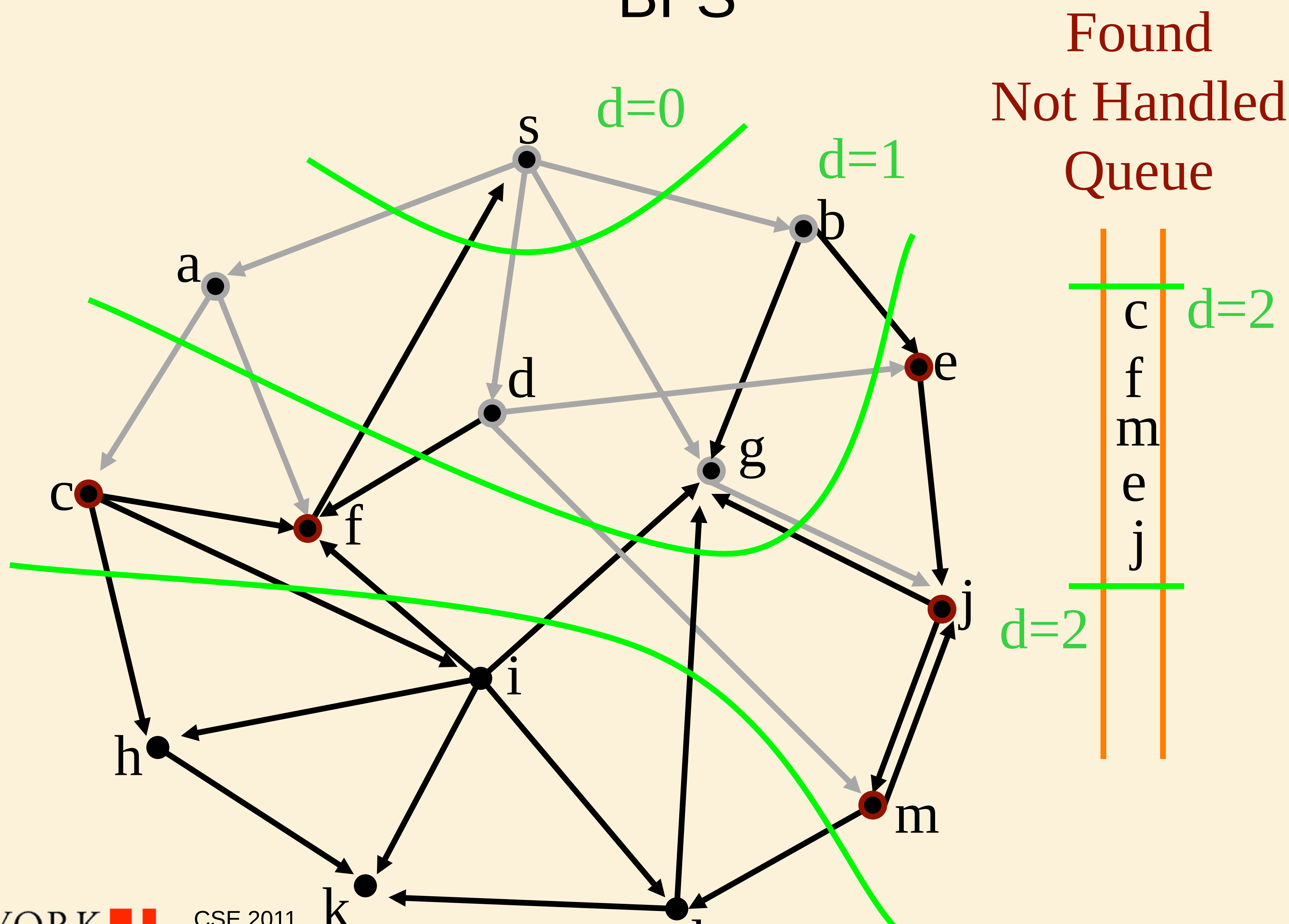
BFS



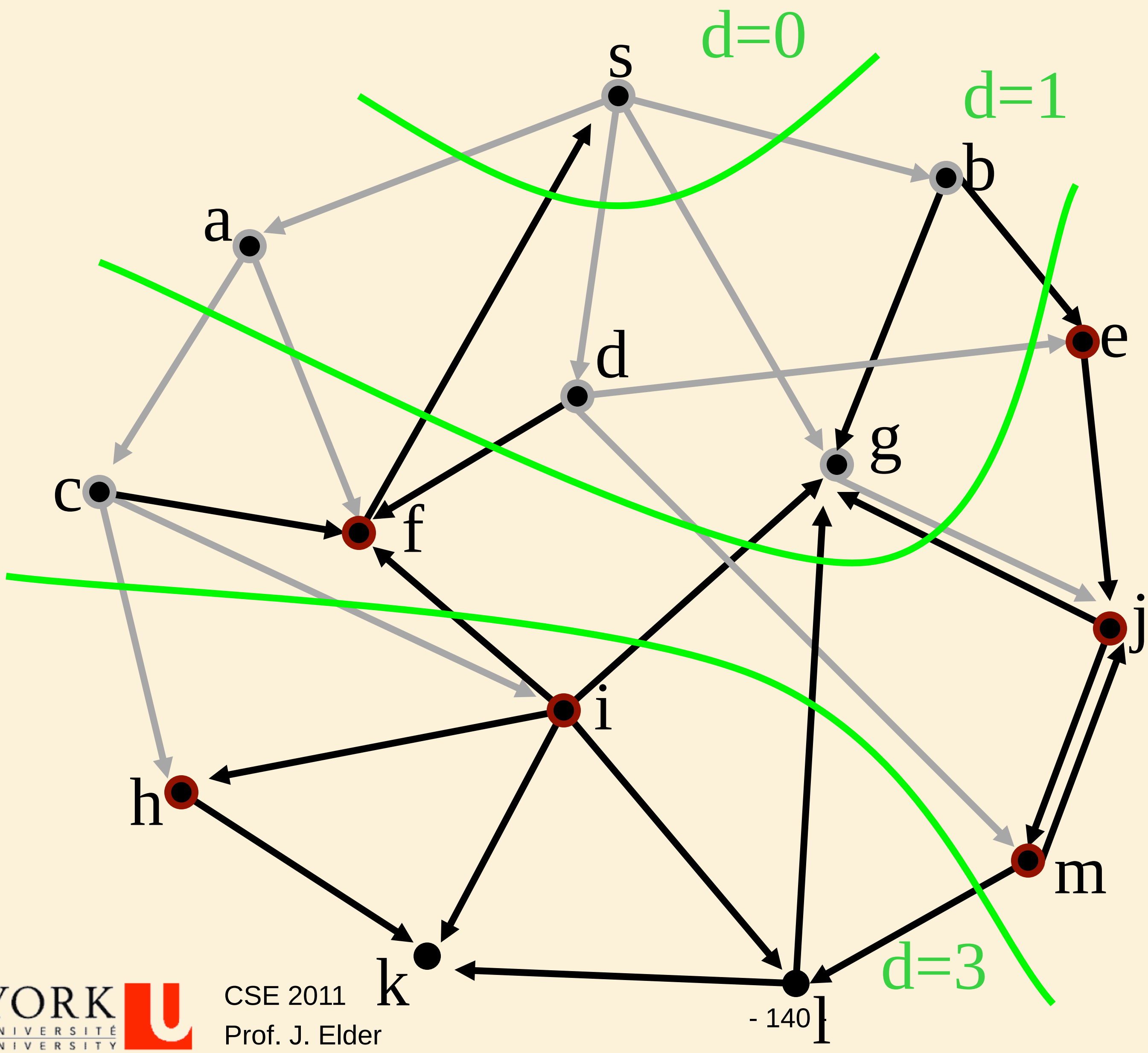
Found
Not Handled
Queue



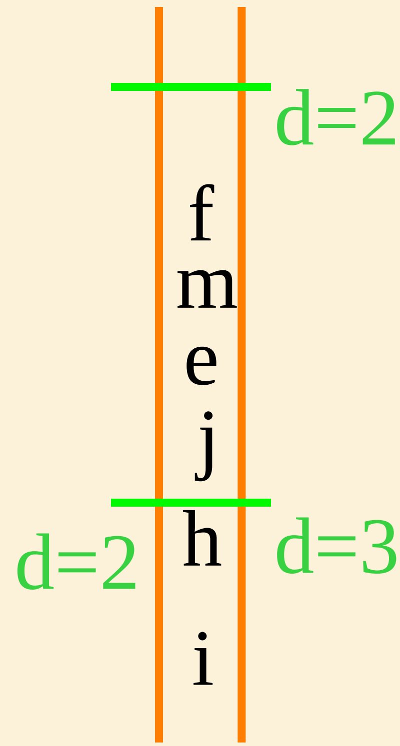
BFS



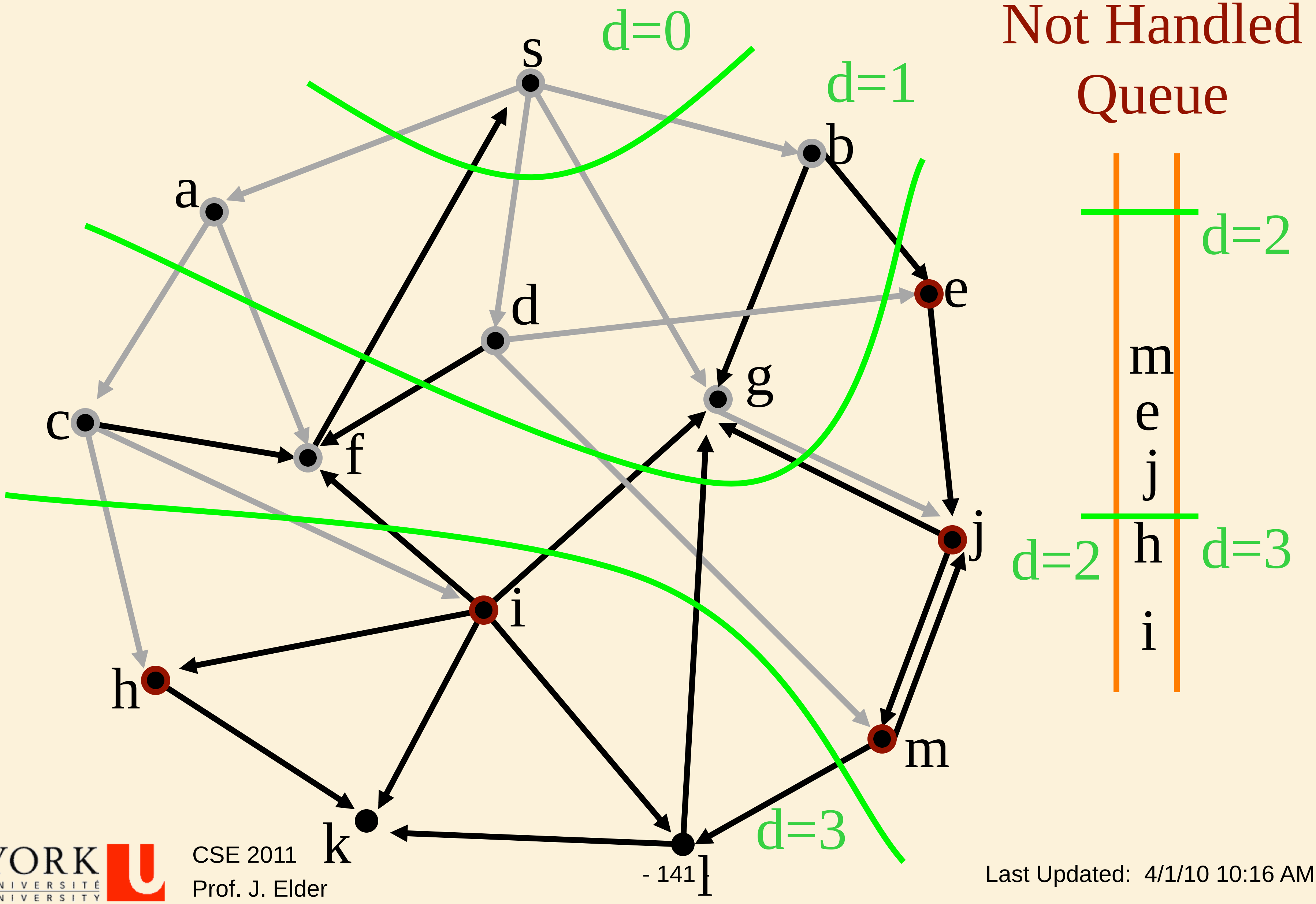
BFS



Found
Not Handled
Queue

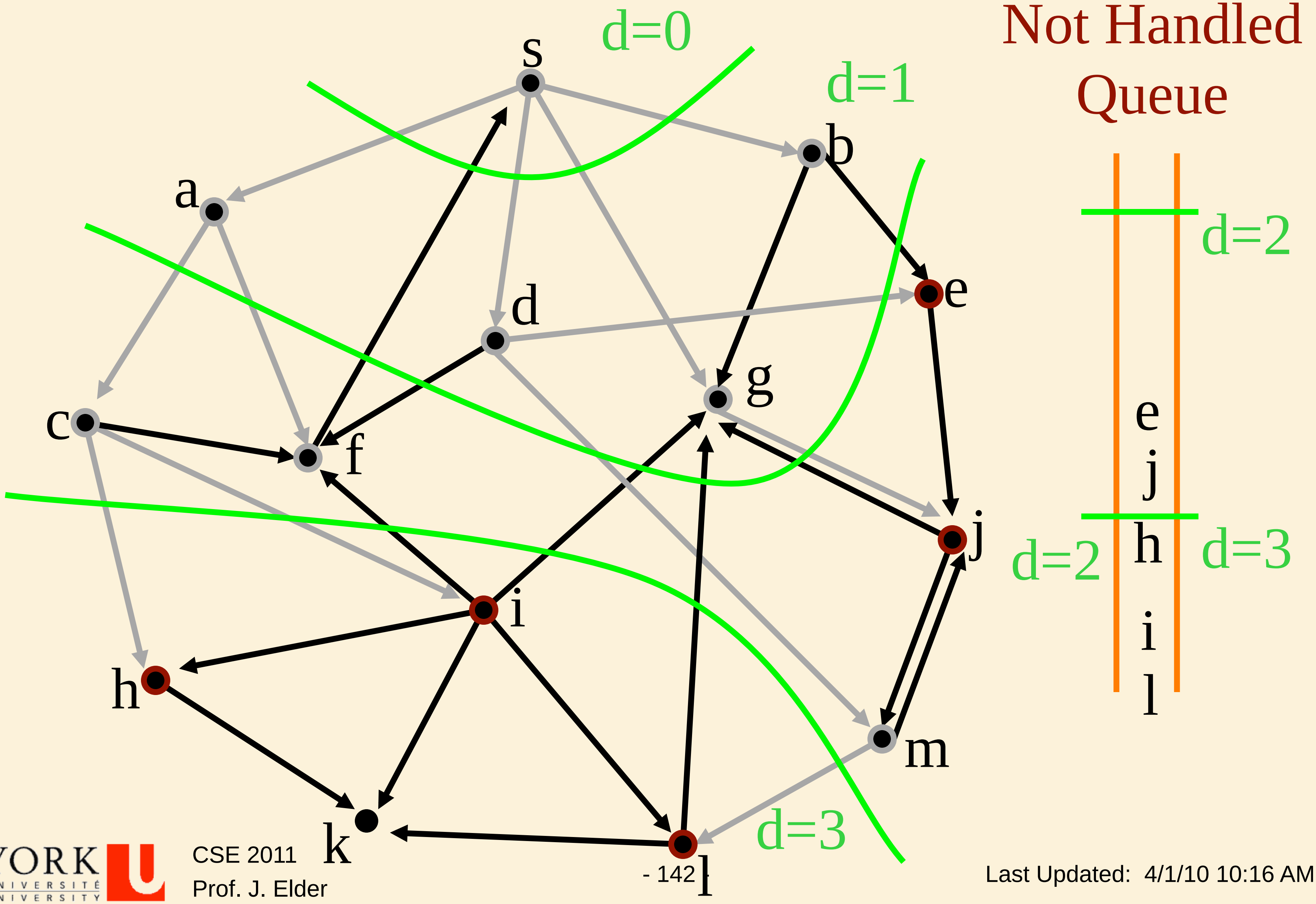


BFS

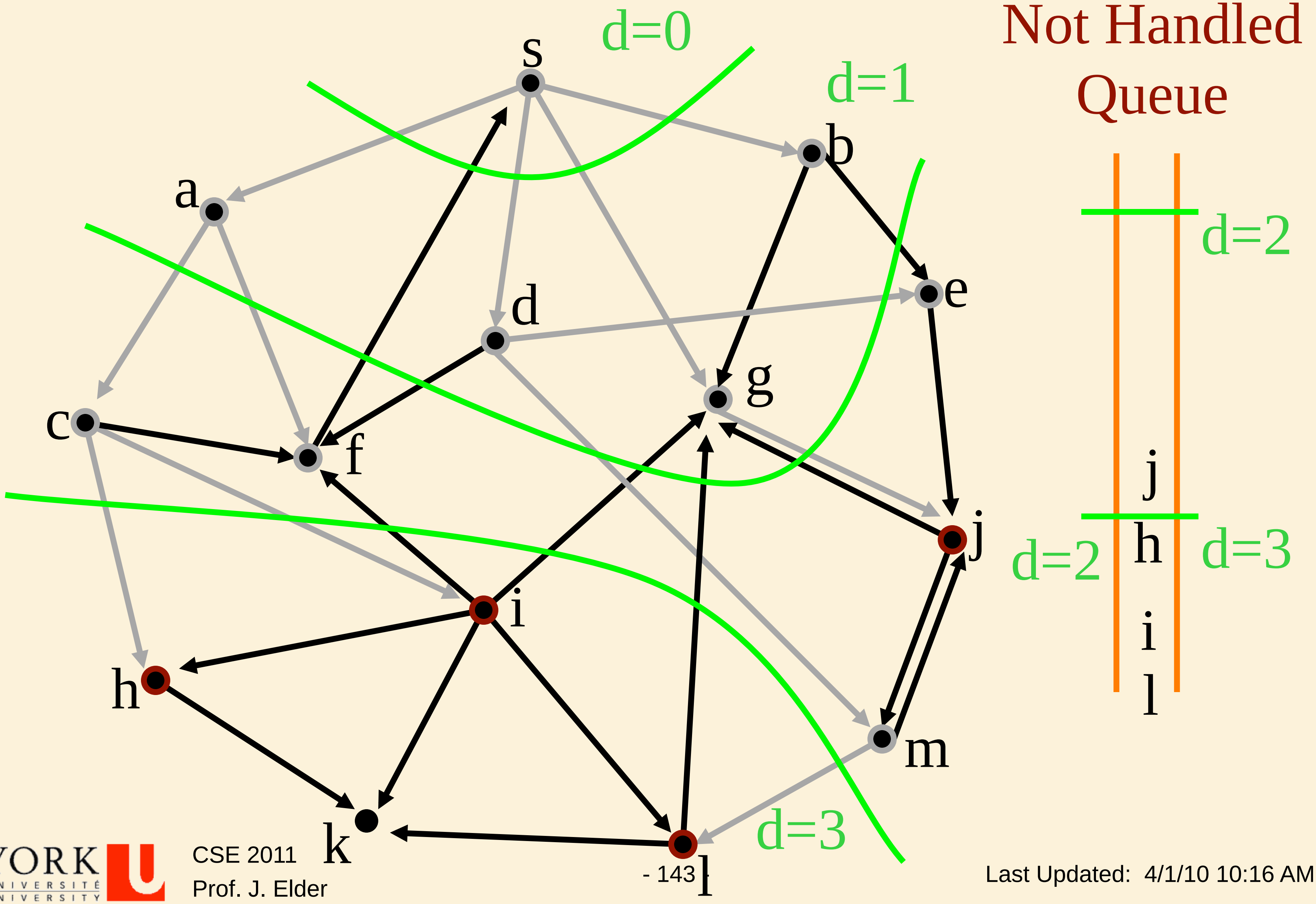


s
a
b
c
d
e
f
g
h
i
j
k
l
m
d=0
d=1
d=2
d=3
Found
Not Handled
Queue
m
e
j
h
i
d=2
d=3

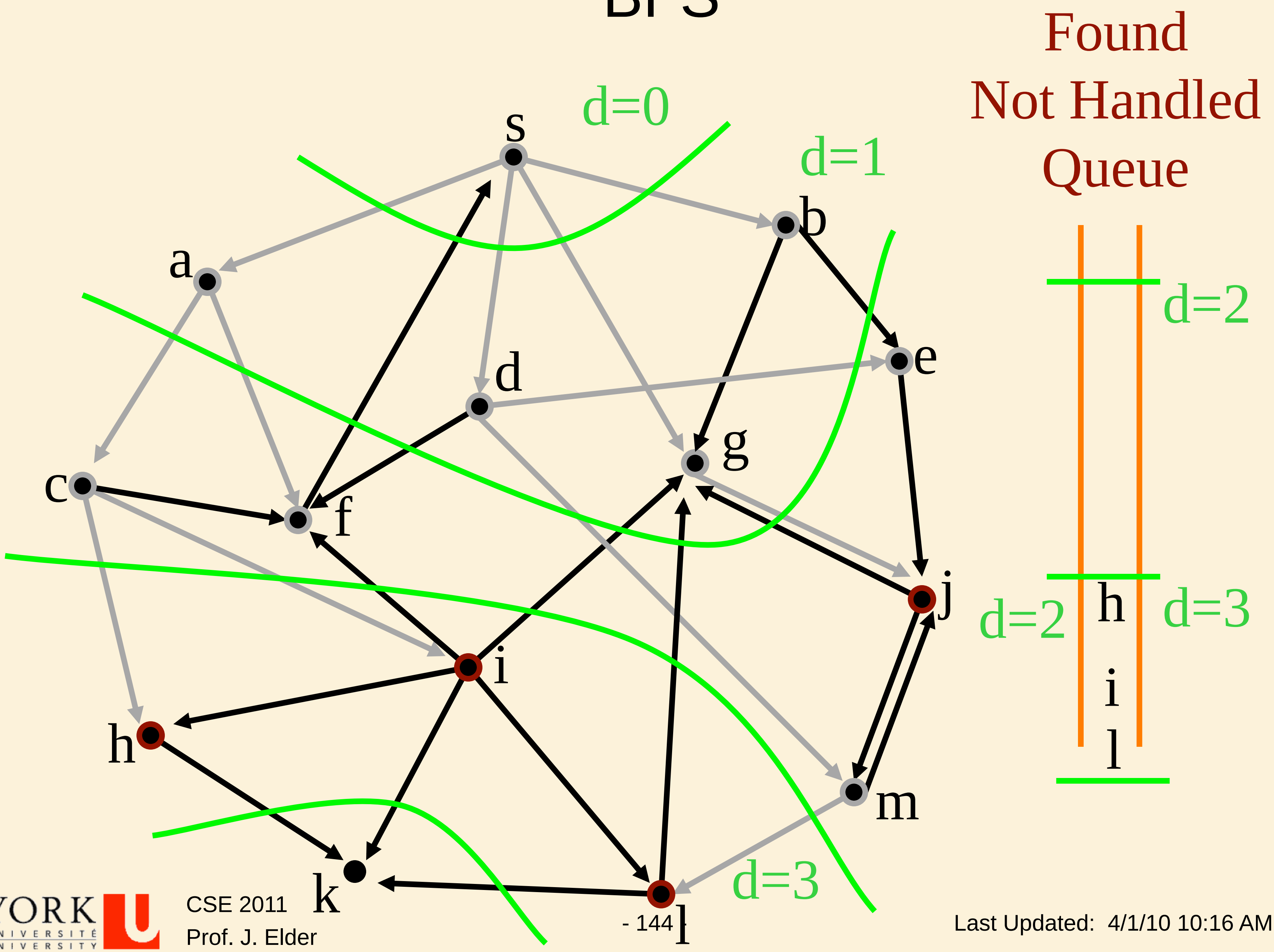
BFS


d=0
d=1
d=2
d=3
s
a
b
c
d
e
f
g
h
i
j
k
l
m
Found
Not Handled
Queue
d=2
e
j
h
i
l
d=2
d=3

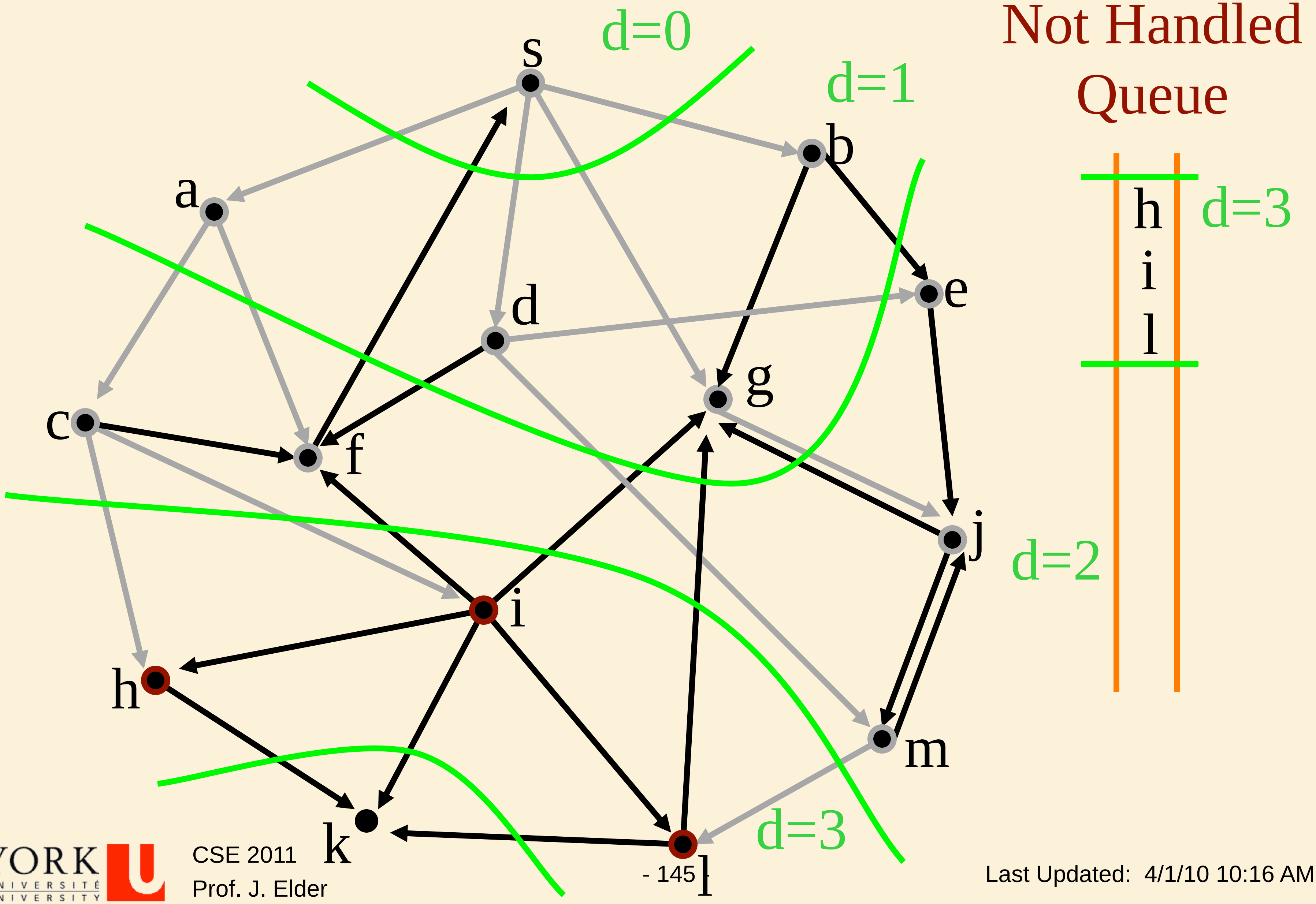
BFS


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
j
h
i
l
d=2
d=3

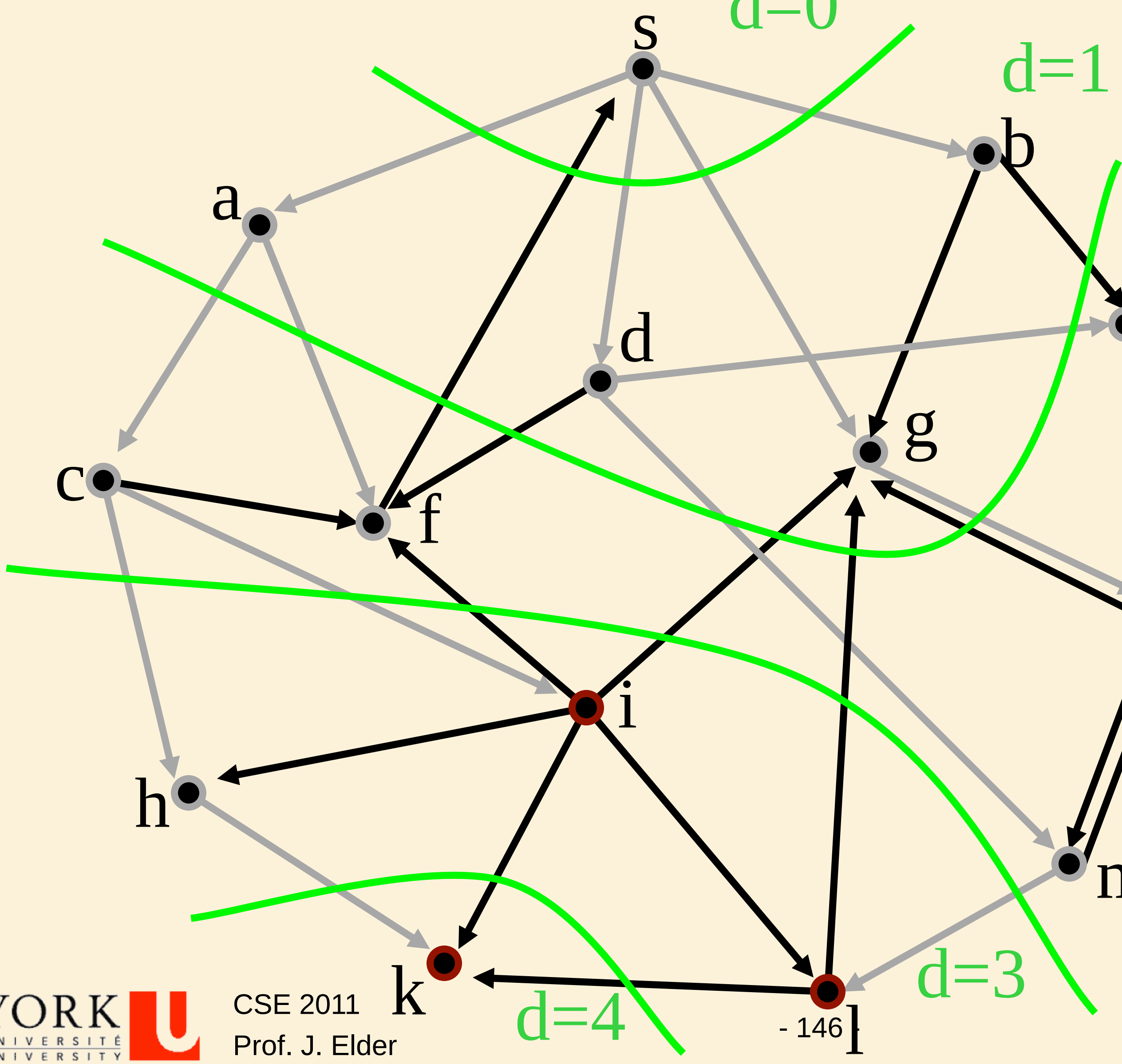
BFS



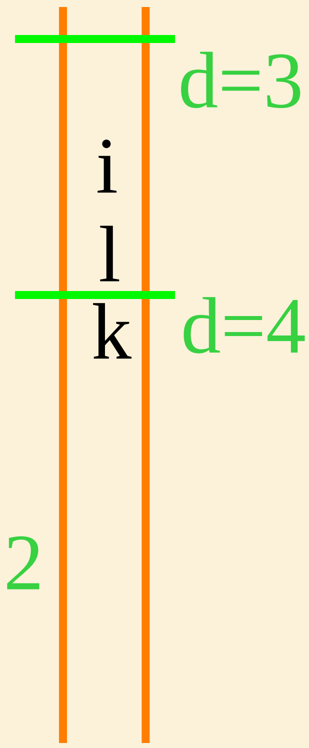
BFS



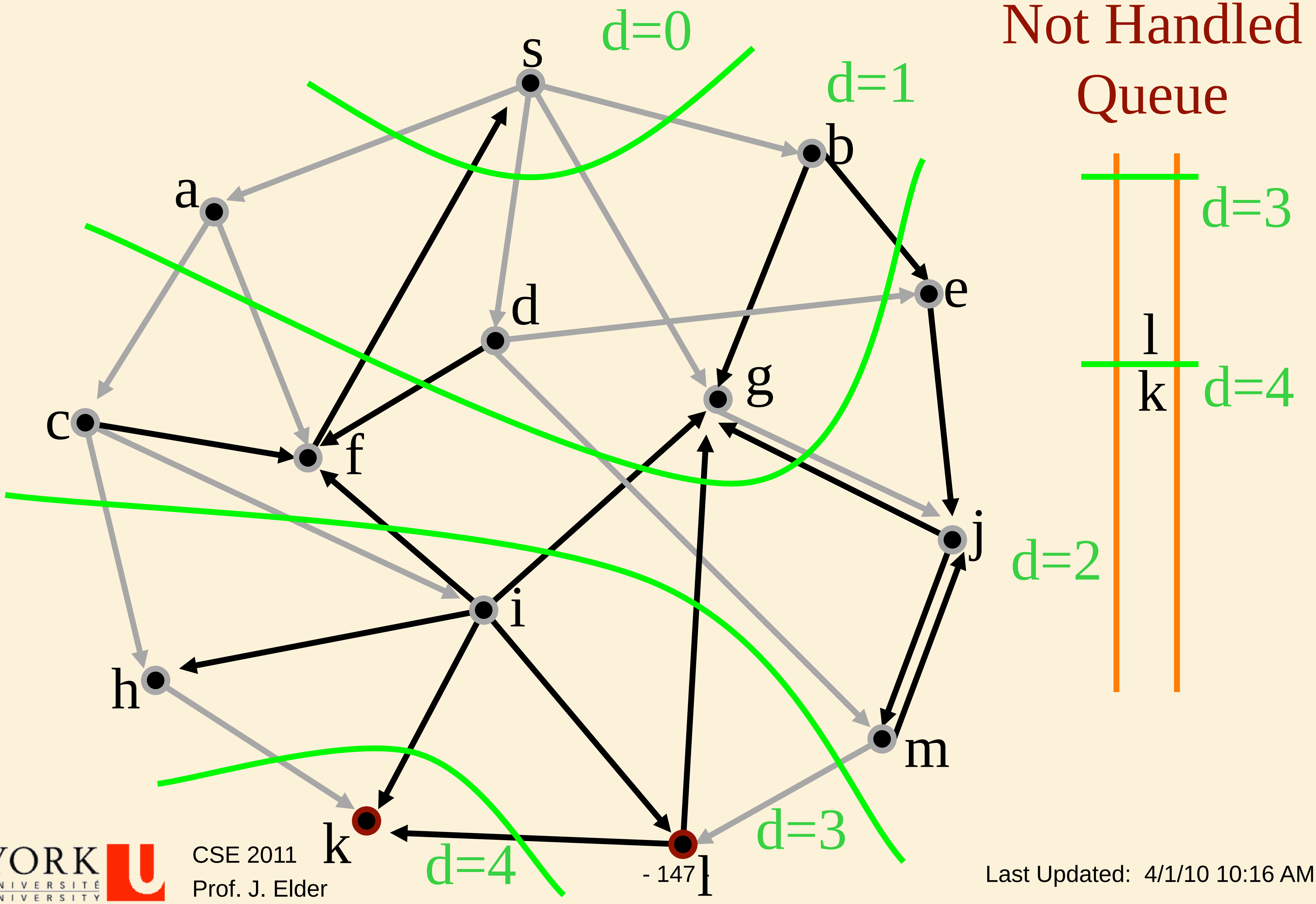
BFS



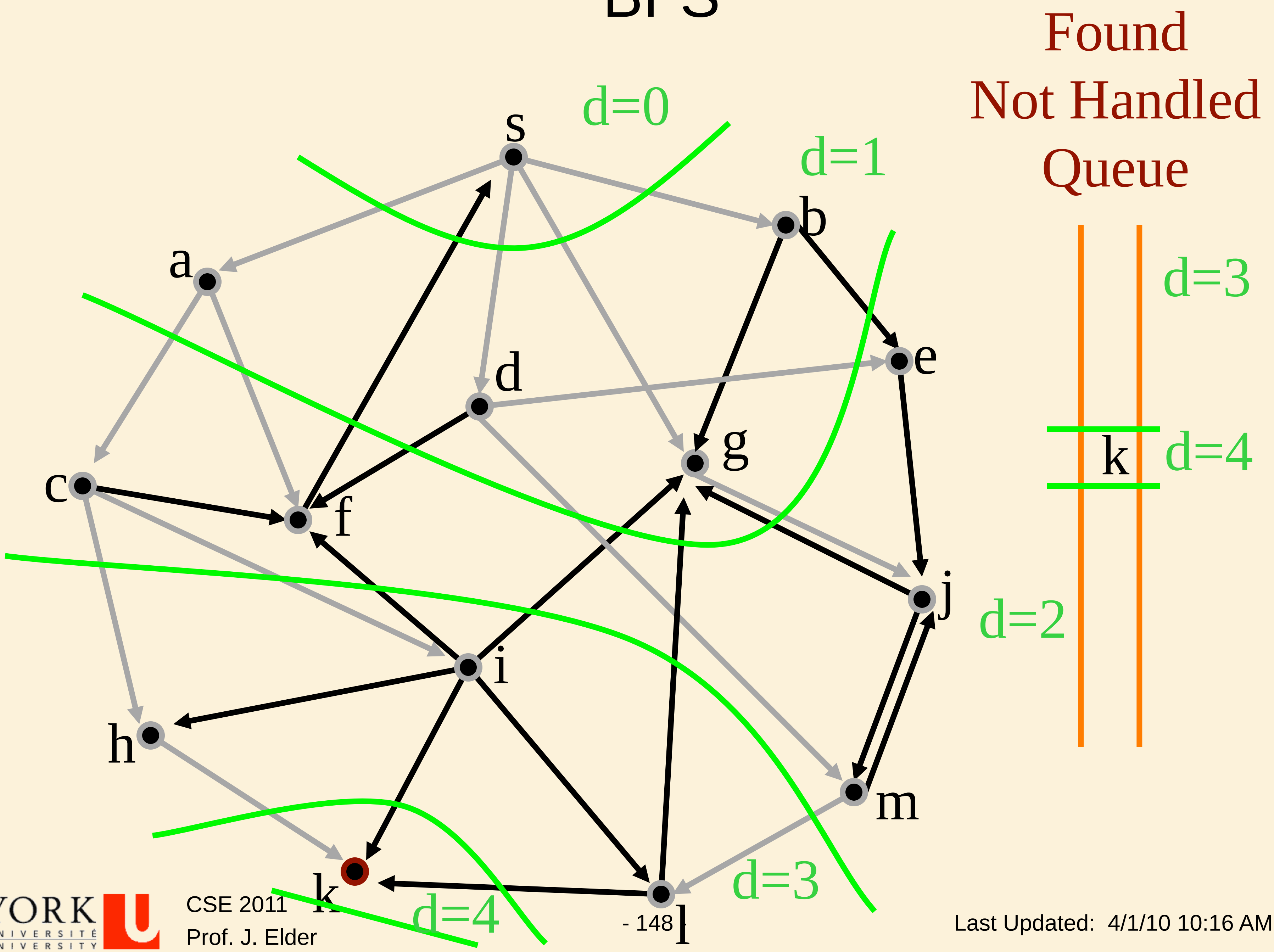
Found
Not Handled
Queue



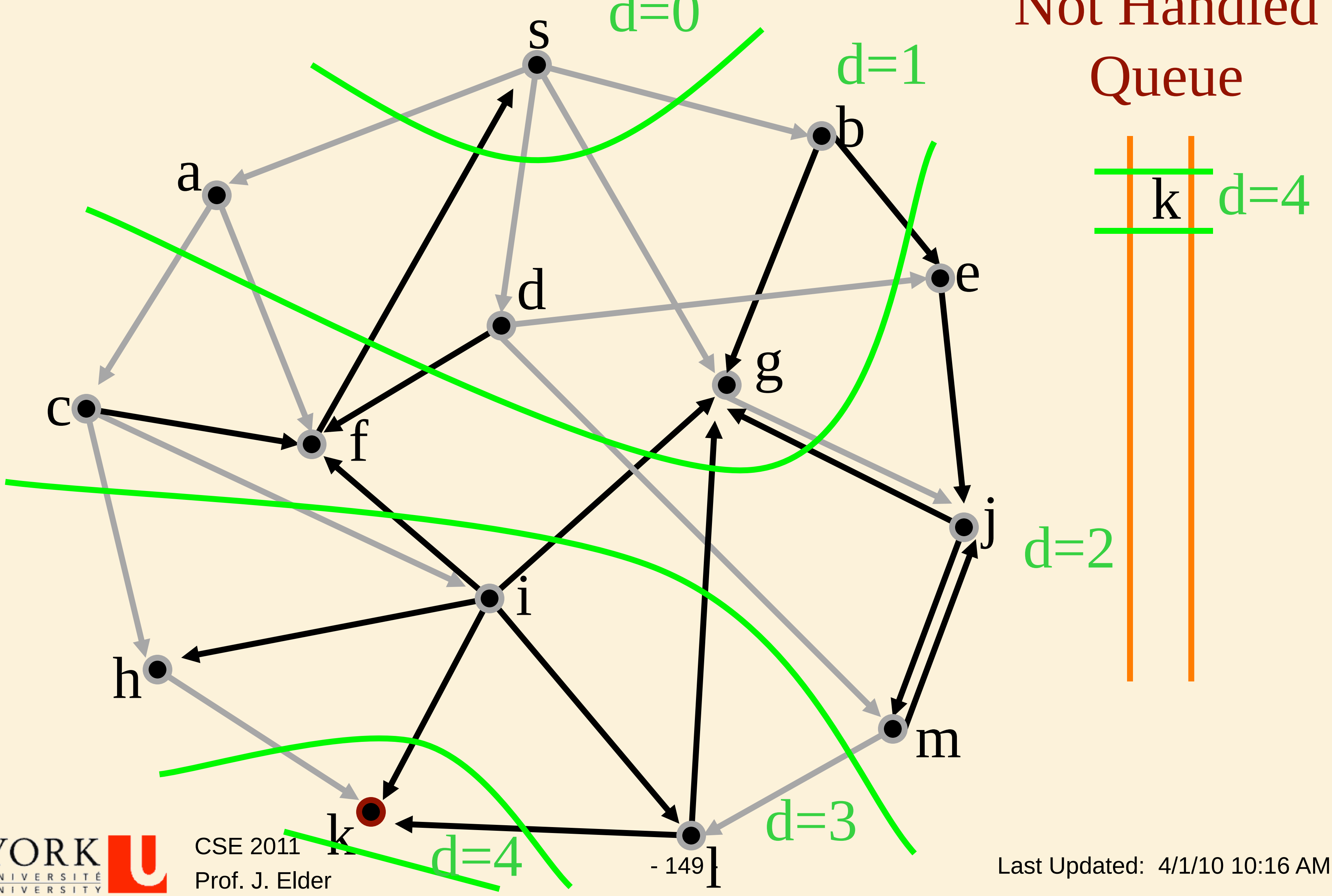
BFS


s
a
b
d
e
c
f
g
j
i
h
m
k
l
d=0
d=1
d=2
d=3
d=4
Found
Not Handled
Queue
l
k
d=3
d=4

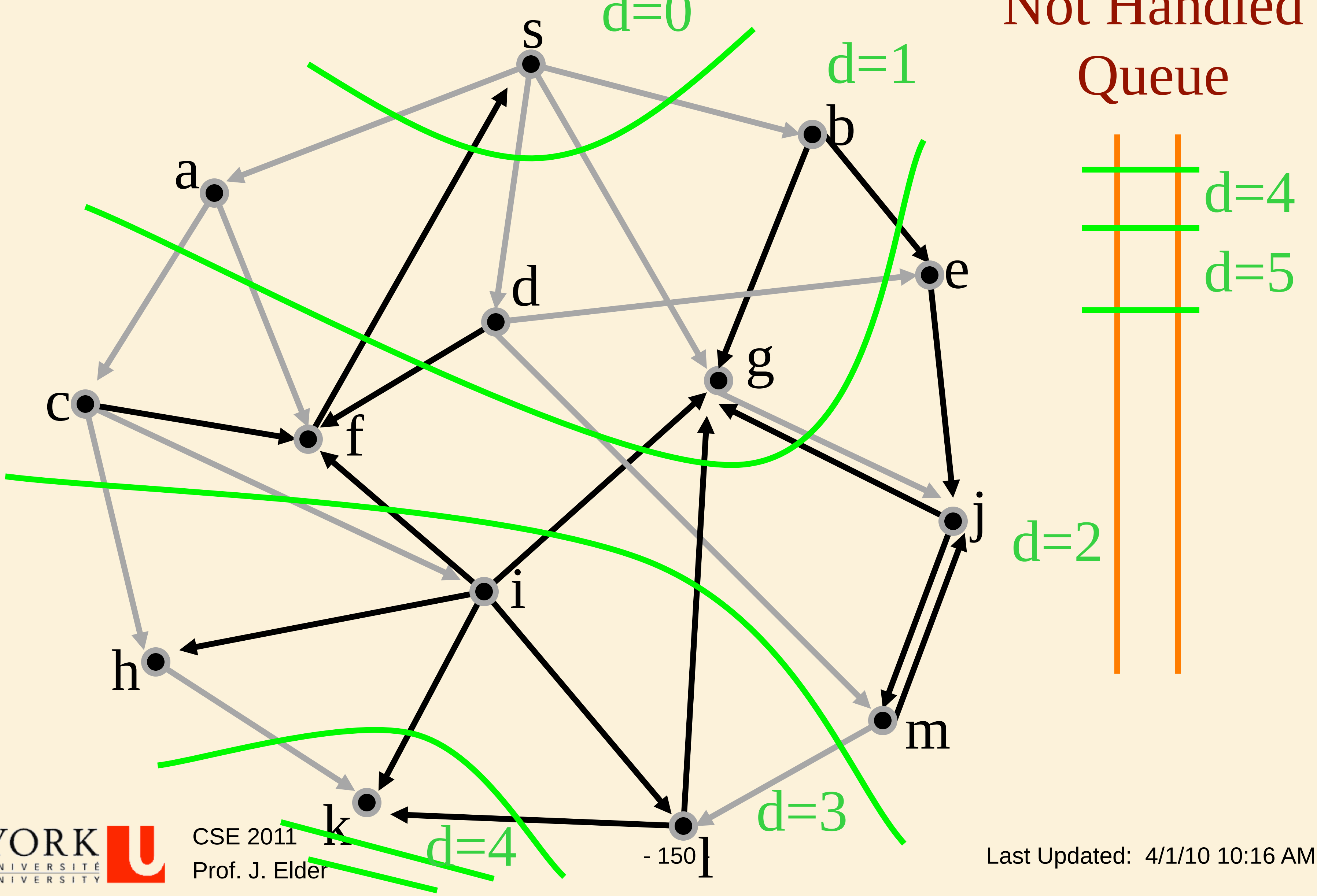
BFS



BFS



BFS



Breadth-First Search Algorithm: Properties

BFS(G,s)

Precondition: G is a graph, s is a vertex in G

Postcondition: $d[u] =$ shortest distance $\delta[u]$ and

$\pi[u]$ = predecessor of u on shortest paths from s to each vertex u in G

```
for each vertex u ∈ V[G]
    d[u] ← ∞
    π[u] ← null
    color[u] = BLACK //initialize vertex
colour[s] ← RED
d[s] ← 0
Q.enqueue(s)
while Q ≠ ∅
    u ← Q.dequeue()
    for each v ∈ Adj[u] //explore edge (u,v)
        if color[v] = BLACK
            colour[v] ← RED
            d[v] ← d[u] + 1
            π[v] ← u
            Q.enqueue(v)
    colour[u] ← GRAY
```

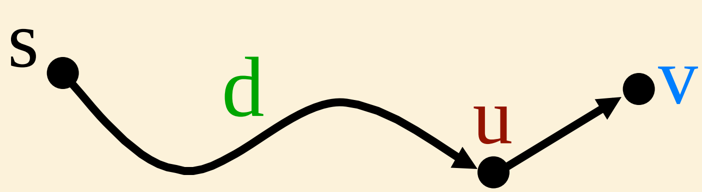
- Q is a FIFO queue.
- Each vertex assigned finite d value at most once.
- Q contains vertices with d values $\{i, \ldots, i, i+1, \ldots, i+1\}$
- d values assigned are monotonically increasing over time.

Breadth-First-Search is Greedy

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

Correctness

Basic Steps:



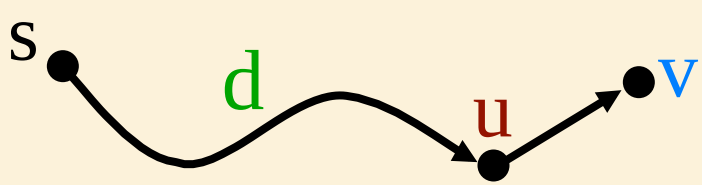
The shortest path to u has length d

& there is an edge from u to v

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

Correctness: Intuition

- 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!



Inductive Proof of BFS

Suppose at step i that the set of nodes S_i with distance $\delta(v) \leq d_i$ have been discovered and their distance values $d[v]$ have been correctly assigned.

Further suppose that the queue contains only nodes in S_i with d values of d_i.

Any node v with $\delta(v) = d_i + 1$ must be adjacent to S_i.

Any node v adjacent to S_i but not in S_i must have $\delta(v) = d_i + 1$.

At step $i+1$, all nodes on the queue with d values of d_i are dequeued and processed.

In so doing, all nodes adjacent to S_i are discovered and assigned d values of $d_i + 1$.

Thus after step $i+1$, all nodes v with distance $\delta(v) \leq d_i + 1$ have been discovered and their distance values $d[v]$ have been correctly assigned.

Furthermore, the queue contains only nodes in S_i with d values of $d_i + 1$.

Correctness: Formal Proof

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

Output:

$d[v] =$ distance δ(v) 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. $d[v] \ngtr \delta(s,v) \forall v \in V$

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

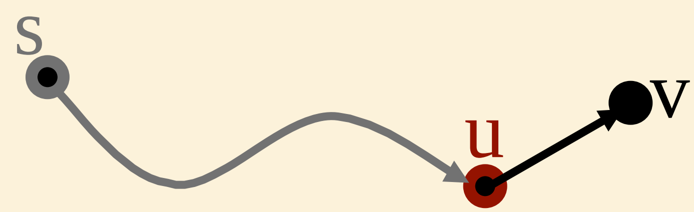
Proof: There exists a path from s to v of length $d[v]$.

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: $d[v] \ge \delta(s,v) \forall v \in V$

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

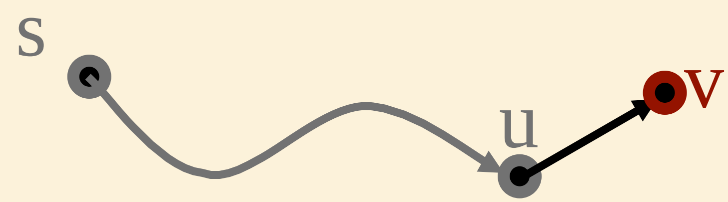
BFS(G,s)

Precondition: G is a graph, s is a vertex in G

Postcondition: $d[u]$ = shortest distance $\delta[u]$ and

$\pi[u]$ = predecessor of u on shortest paths from s to each vertex u in G

for each vertex $u \in V[G]$
    $d[u] \leftarrow \infty$
    $\pi[u] \leftarrow$ null
    color[u] = BLACK //initialize vertex
colour[s] $\leftarrow$ RED
$d[s] \leftarrow 0$
Q.enqueue(s)
while Q $\neq \emptyset$ $\leftarrow$ <LI>: $d[v] \ge \delta(s,v) \forall$ 'discovered' (red or grey) $v \in V$
    u $\leftarrow$ Q.dequeue()
    for each $v \in$ Adj[u] //explore edge (u,v)
        if color[v] = BLACK
            colour[v] $\leftarrow$ RED
            $d[v] \leftarrow d[u]+1$ $\ge \delta(s,u)+1 \ge \delta(s,v)$
            $\pi[v] \leftarrow u$
            Q.enqueue(v)
    $colour[u] \leftarrow GRAY$



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.

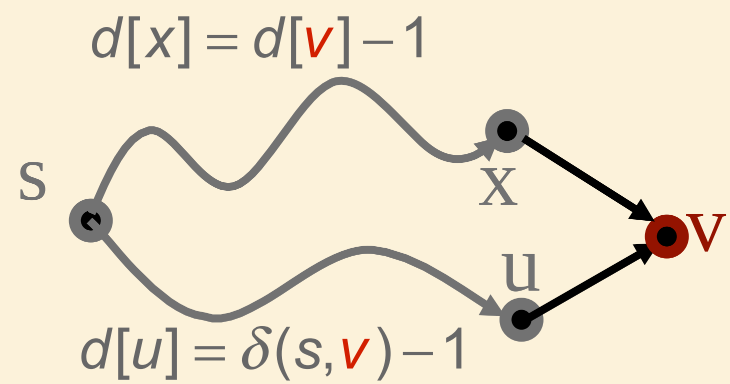
Let v be the vertex with minimum $\delta(s,v)$ that receives such a d value.

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)

Precondition: G is a graph, s is a vertex in G

Postcondition: $d[u] =$ shortest distance $\delta[u]$ and

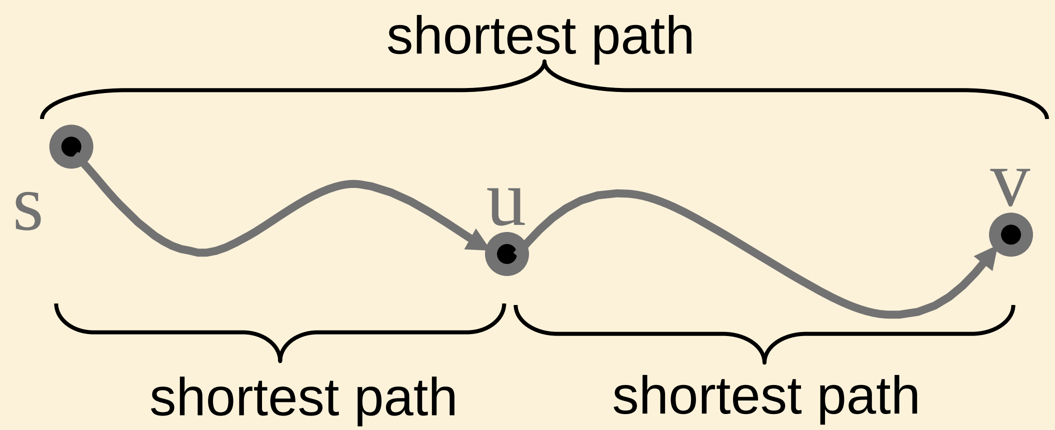
π[u] = predecessor of u on shortest paths from s to each vertex u in G

```
for each vertex u ∈ V[G]
        d[u] ← ∞
        π[u] ← null
        color[u] = BLACK //initialize vertex
colour[s] ← RED
d[s] ← 0
Q.enqueue(s)
while Q ≠ ∅
        u ← Q.dequeue()
        for each v ∈ Adj[u] //explore edge (u,v)
                if color[v] = BLACK
                        colour[v] ← RED
                        d[v] ← d[u] + 1
                        π[v] ← u
                        Q.enqueue(v)
        colour[u] ← GRAY
```

Optimal Substructure Property

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

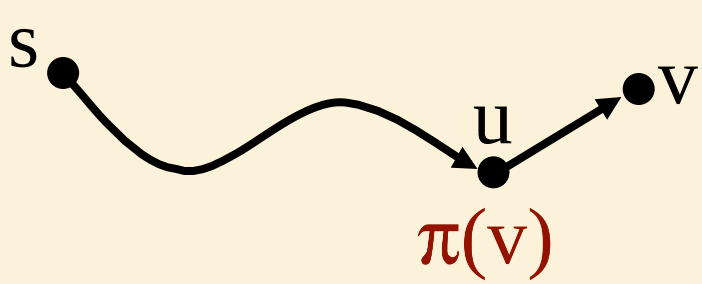
How would we prove this?

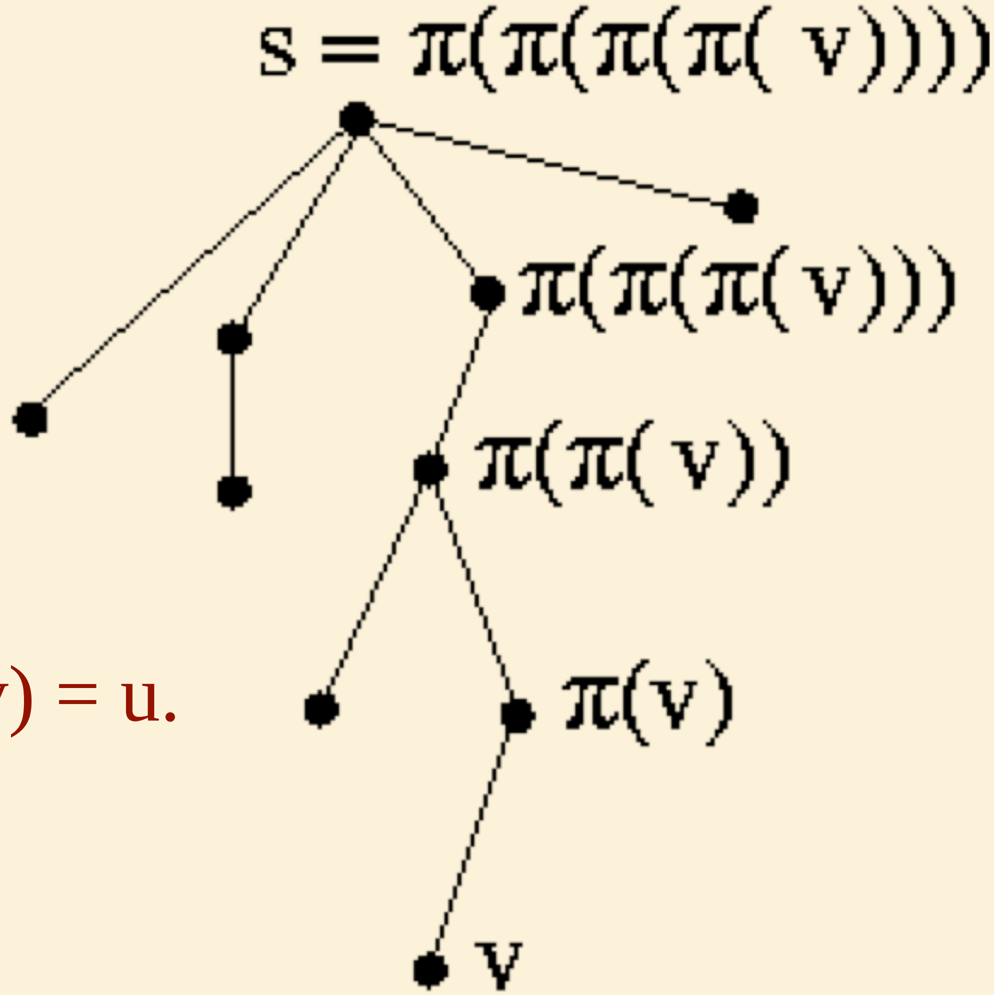


- The optimal substructure property
  - is a hallmark of both greedy and dynamic programming algorithms.
  - allows us to compute both shortest path distance and the shortest paths themselves by storing only one *d* value and one predecessor value per vertex.

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

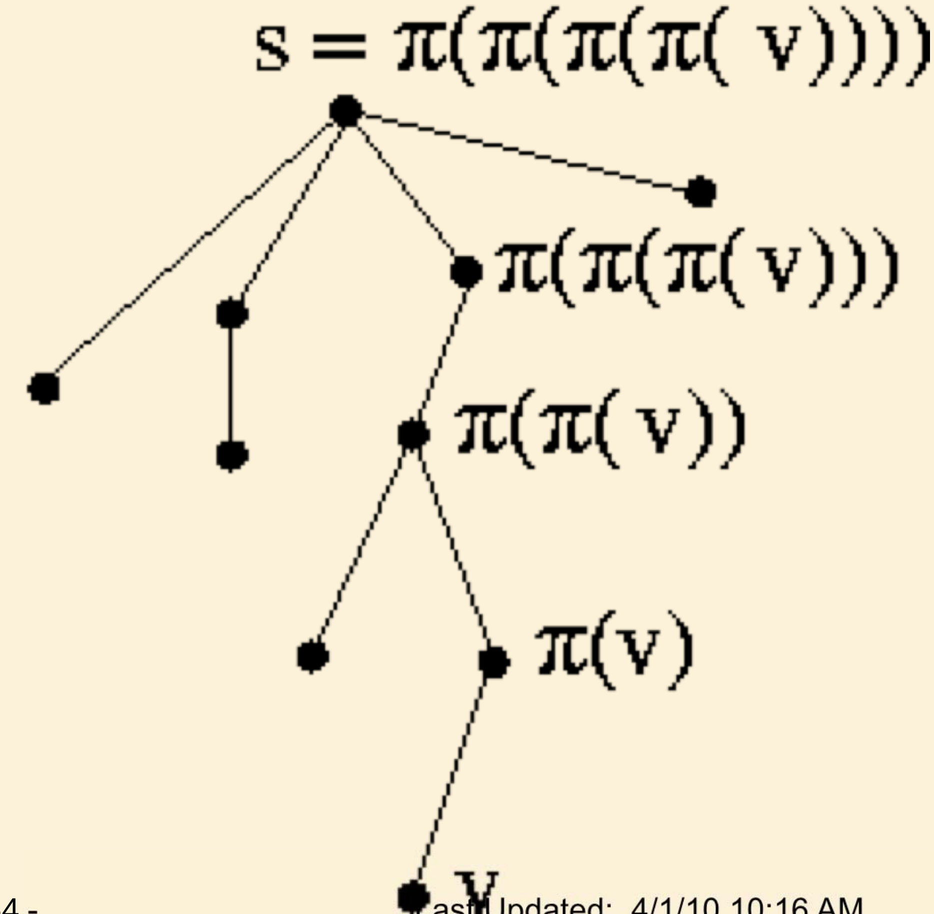
else if $\pi[v] = \text{NIL}$ then

  print "no path from" s "to" v "exists"

else

  PRINT-PATH(G, s, $\pi[v]$)

  print v



BFS Algorithm without Colours

BFS(G,s)

Precondition: G is a graph, s is a vertex in G

Postcondition: predecessors $\pi[u]$ and shortest distance $d[u]$ from s to each vertex u in G has been computed

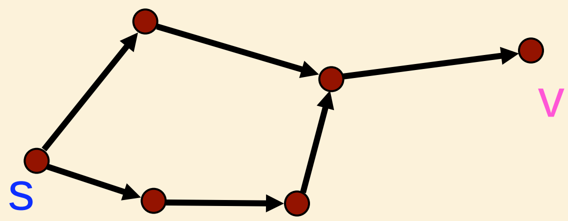
```
for each vertex u ∈ V[G]
        d[u] ← ∞
        π[u] ← null
d[s] ← 0
Q.enqueue(s)
while Q ≠ ∅
        u ← Q.dequeue()
        for each v ∈ Adj[u] //explore edge (u,v)
                if d[v] = ∞
                        d[v] ← d[u] + 1
                        π[v] ← u
                        Q.enqueue(v)
```

Single-Source (Weighted) Shortest Paths

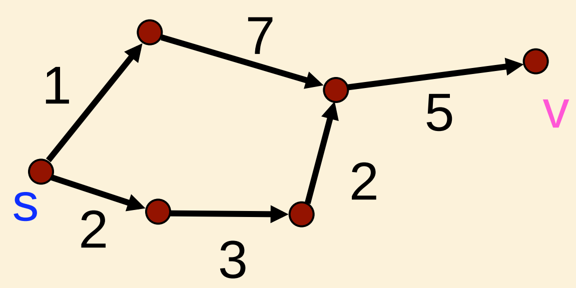
Shortest Path on Weighted Graphs

- 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'?

$\delta(s,v) = 3$

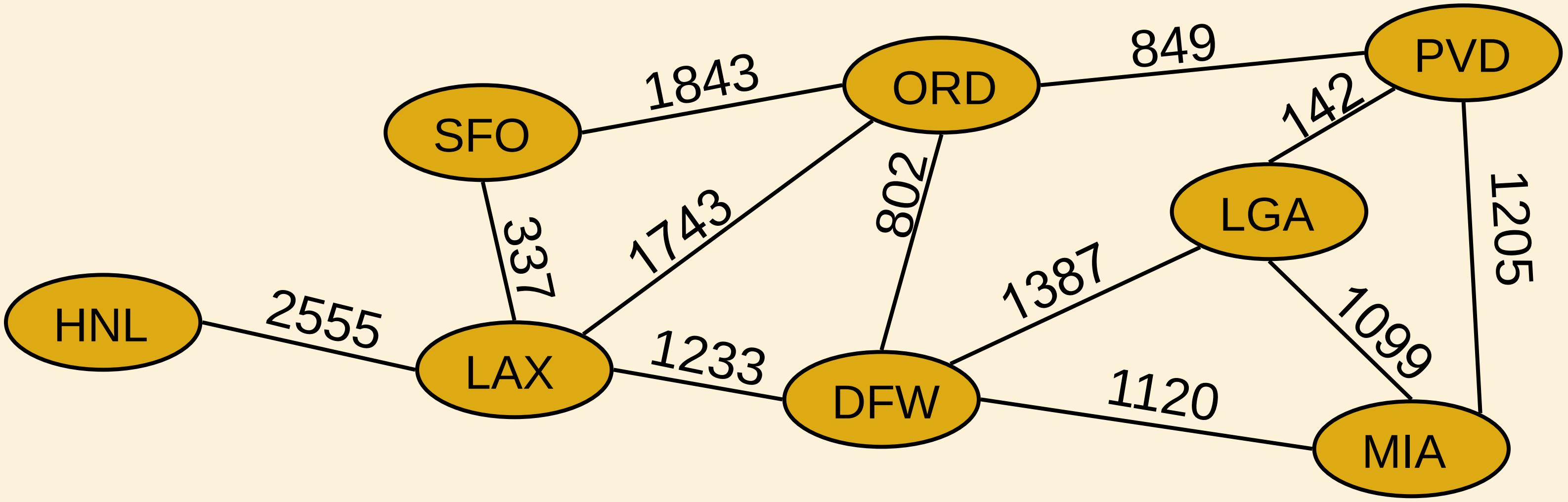


$\delta(s,v) = 12$



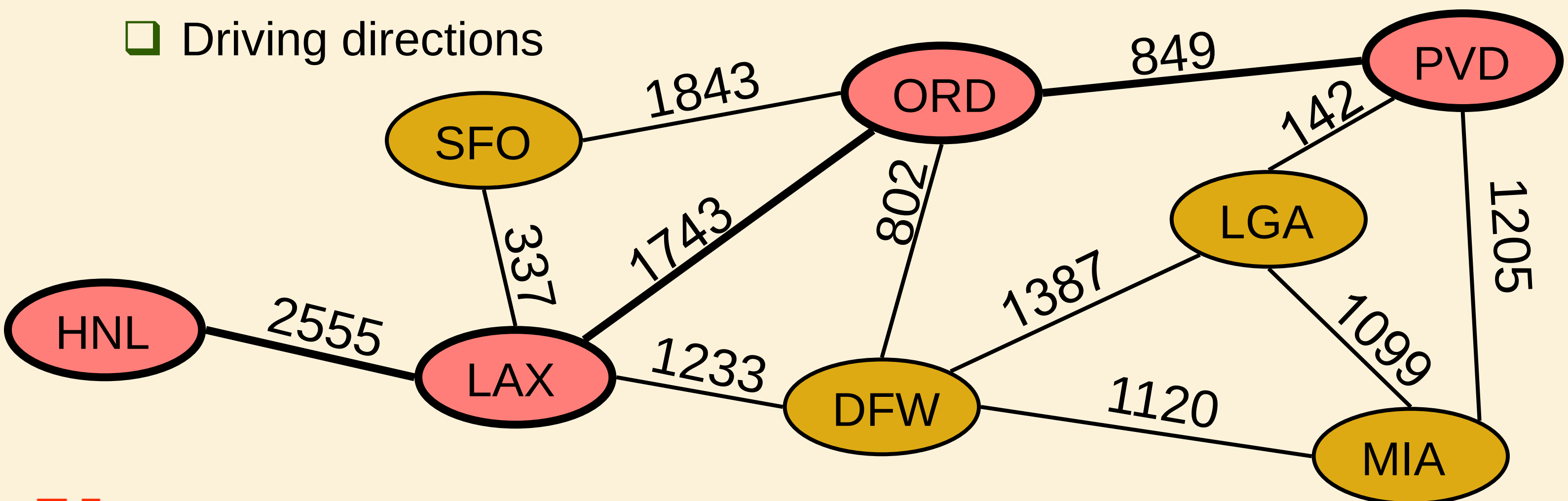
Weighted Graphs

- In a weighted graph, each edge has an associated numerical value, called the weight of the edge
- Edge weights may represent, distances, costs, etc.
- Example:
  - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports



Shortest Paths

- Given a weighted graph and two vertices ***u*** and ***v***, we want to find a path of minimum total weight between ***u*** and ***v*.**
  - Length of a path is the sum of the weights of its edges.
- Example:
  - Shortest path between Providence and Honolulu
- Applications
  - Internet packet routing
  - Flight reservations
  - Driving directions



Shortest Path: Notation

- Input:

Directed Graph $G=(V,E)$

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

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

Shortest-path weight from u to v:

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

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

Shortest Path Properties

Property 1 (Optimal Substructure):

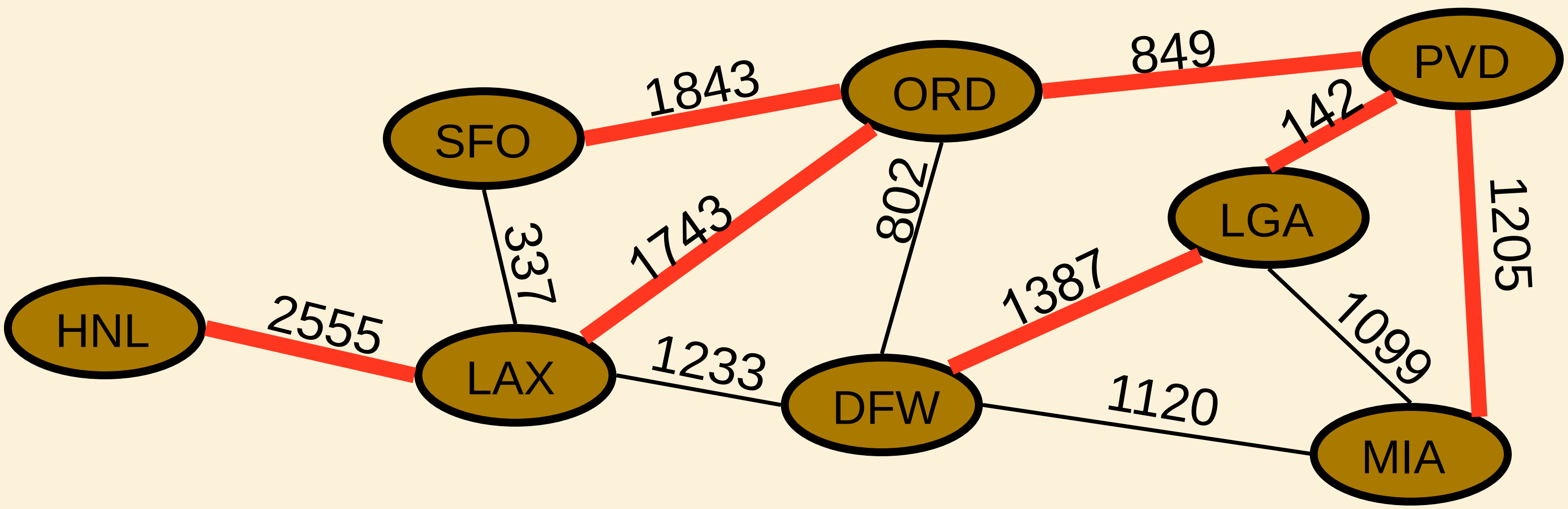
A subpath of a shortest path is itself a shortest path

Property 2 (Shortest Path Tree):

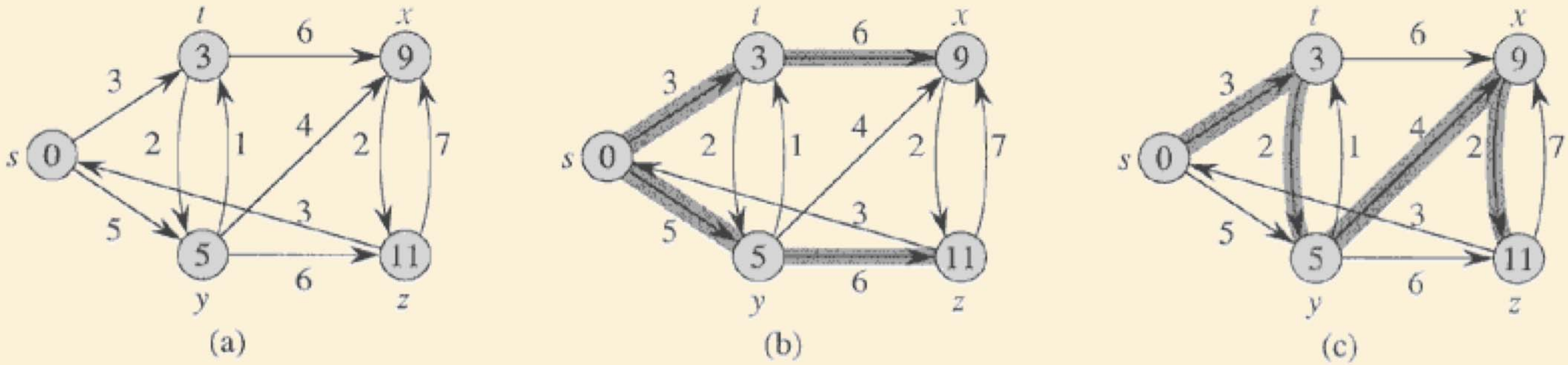
There is a tree of shortest paths from a start vertex to all the other vertices

Example:

Tree of shortest paths from Providence



Shortest path trees are not necessarily unique

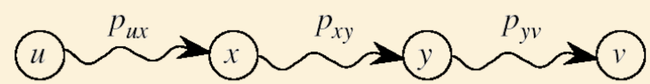


Single-source shortest path search induces a search tree rooted at *s*.

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

Optimal substructure: Proof

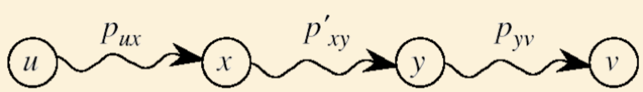
- Lemma: Any subpath of a shortest path is a shortest path
- Proof: Cut and paste.

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 $x \rightarrow \overset{p'_{xy}}{\cdots} \rightarrow y$.

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

Construct p′: 

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

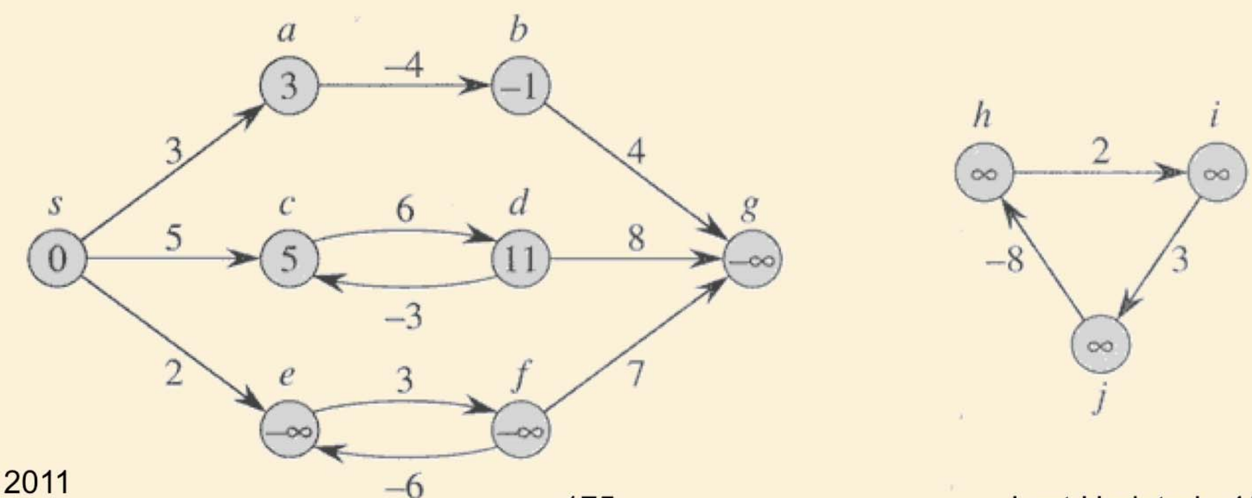
So p wasn't a shortest path after all!

Shortest path variants

- **Single-source shortest-paths problem:** – the shortest path from *s* to each vertex *v*.
- **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.

Shortest-Path Example: Single-Source

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

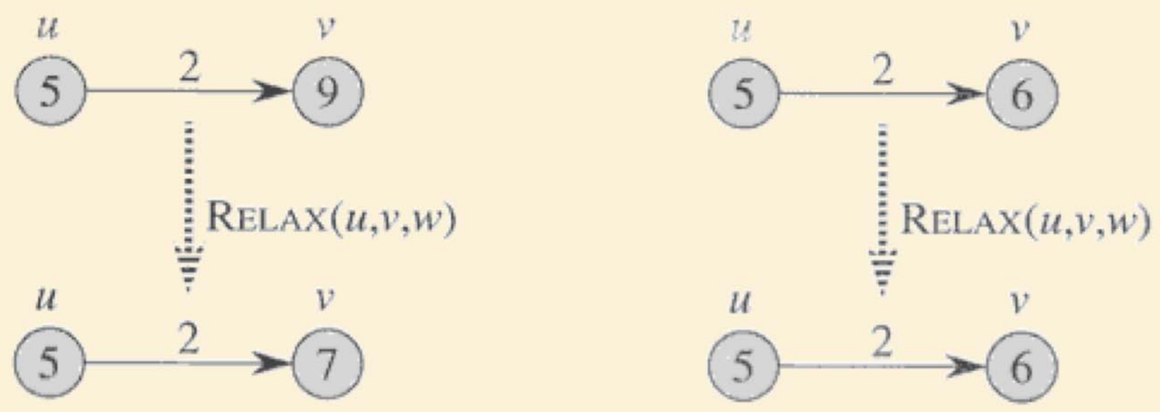
- Can we improve shortest-path estimate for v by first going to ***u*** and then following edge ***(u,v)***?

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

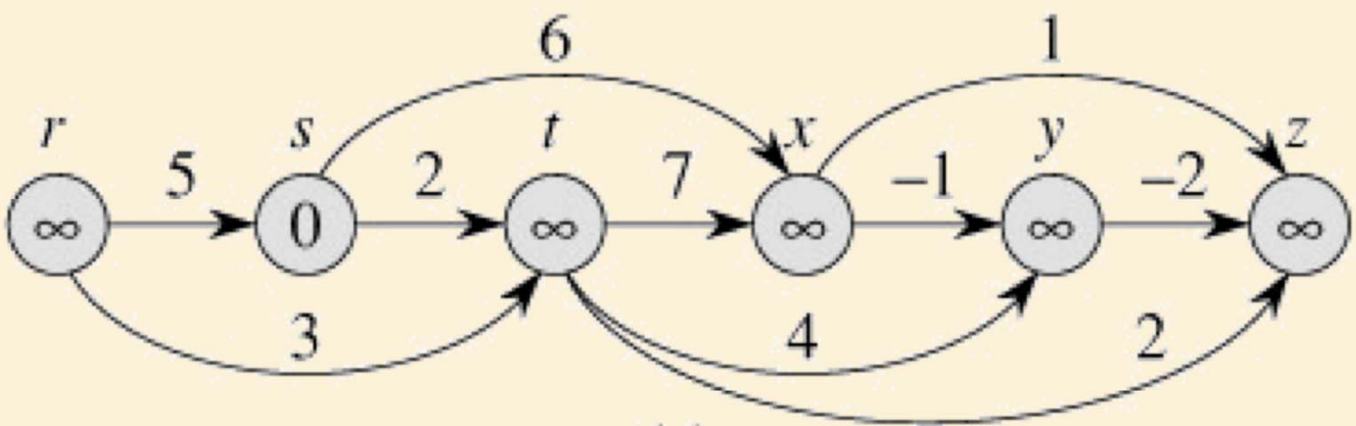
Algorithms differ in the order in which edges are taken and how many times each edge is relaxed.

Example 1. Single-Source Shortest Path on a Directed Acyclic Graph

- Basic Idea: topologically sort nodes and relax in linear order.
- Efficient, since $\delta[u]$ (shortest distance to *u*) has already been computed when edge (u,v) is relaxed.
- Thus we only relax each edge once, and never have to backtrack.

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

- Since graph is a DAG, we are guaranteed no negative-weight cycles.
- Thus algorithm can handle negative edges



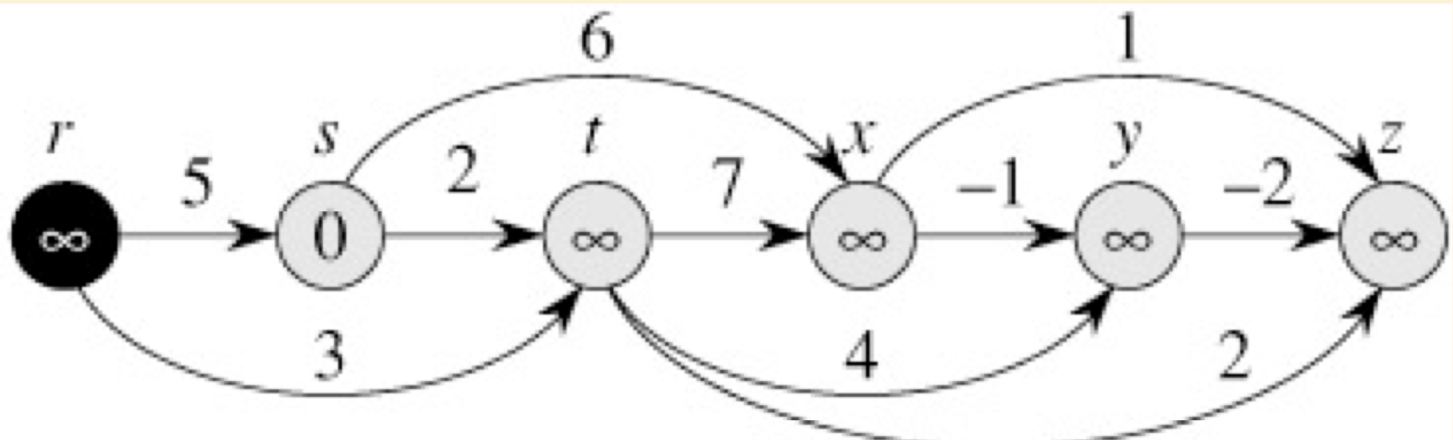
(a)

Algorithm

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

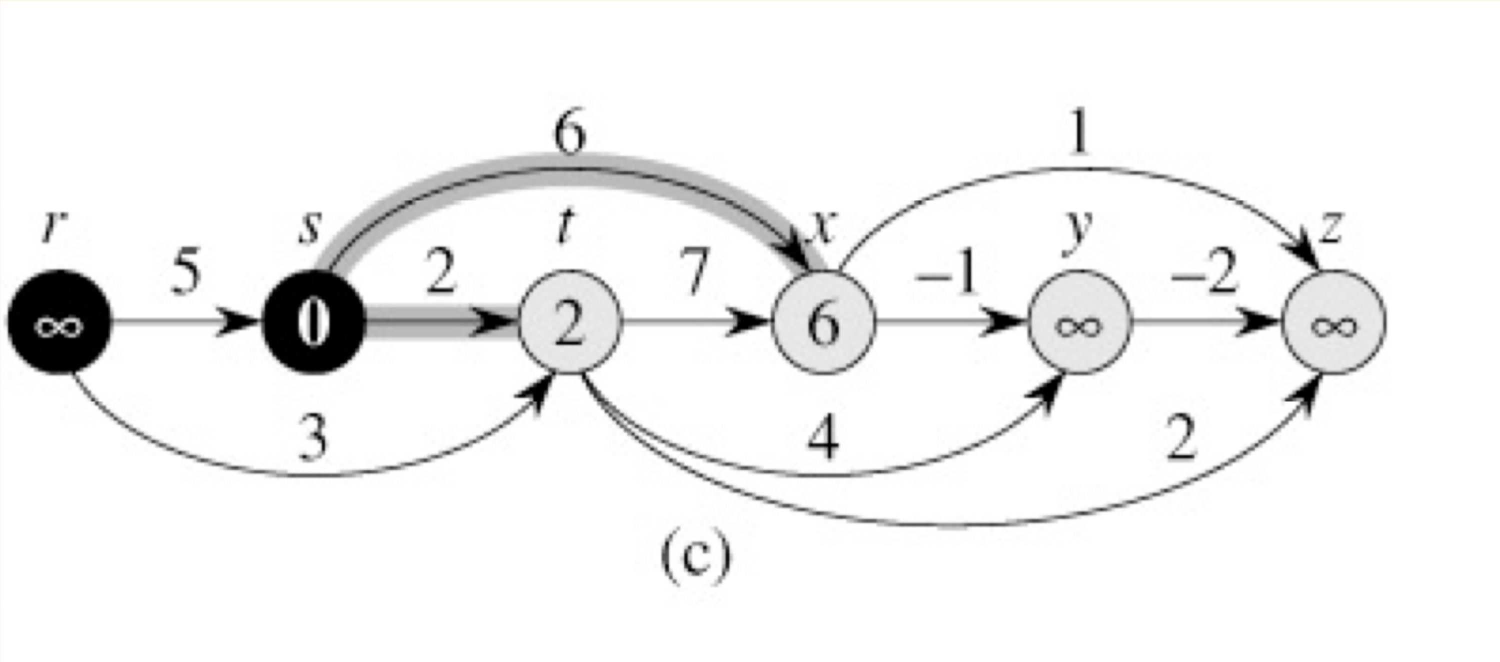
Time: $\Theta(V+E)$

Example

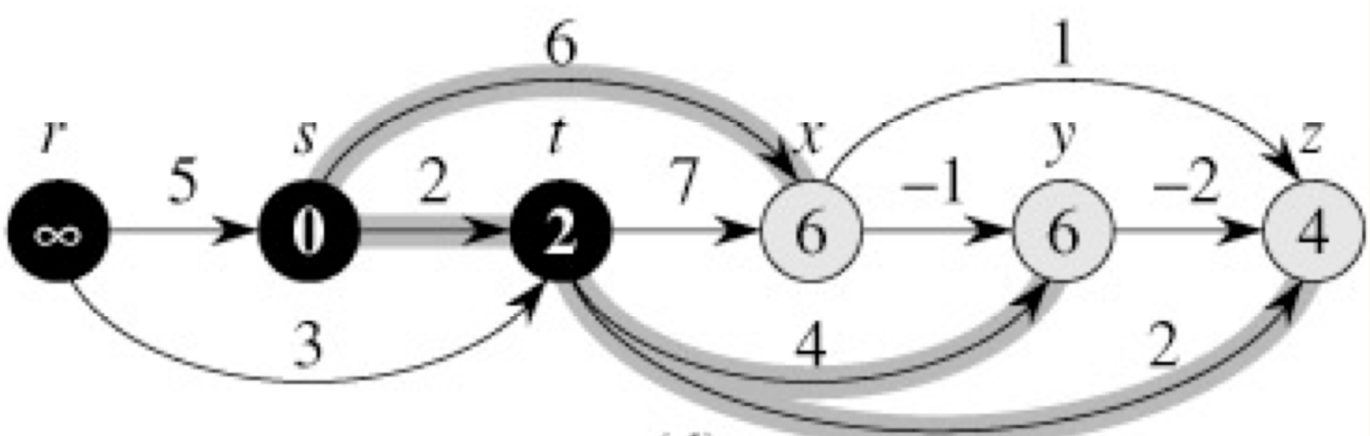


(b)

Example

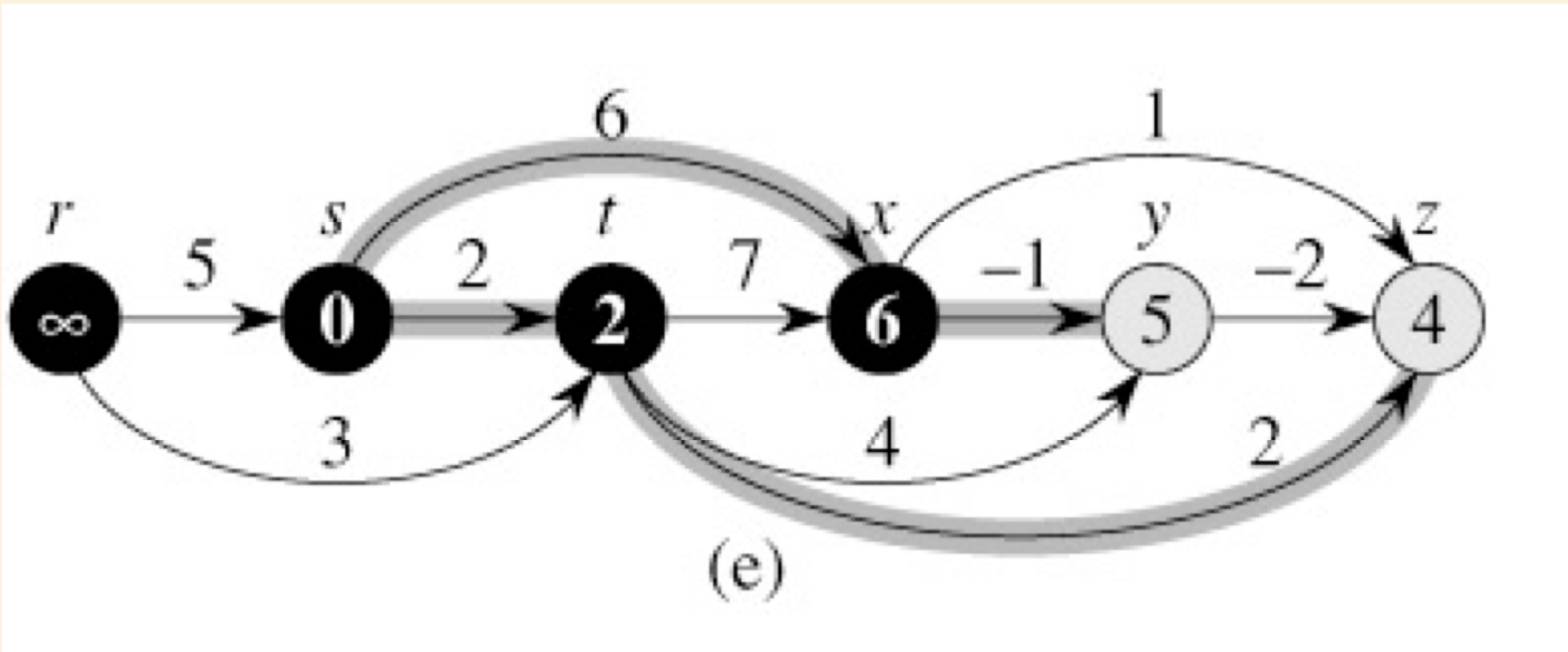


Example

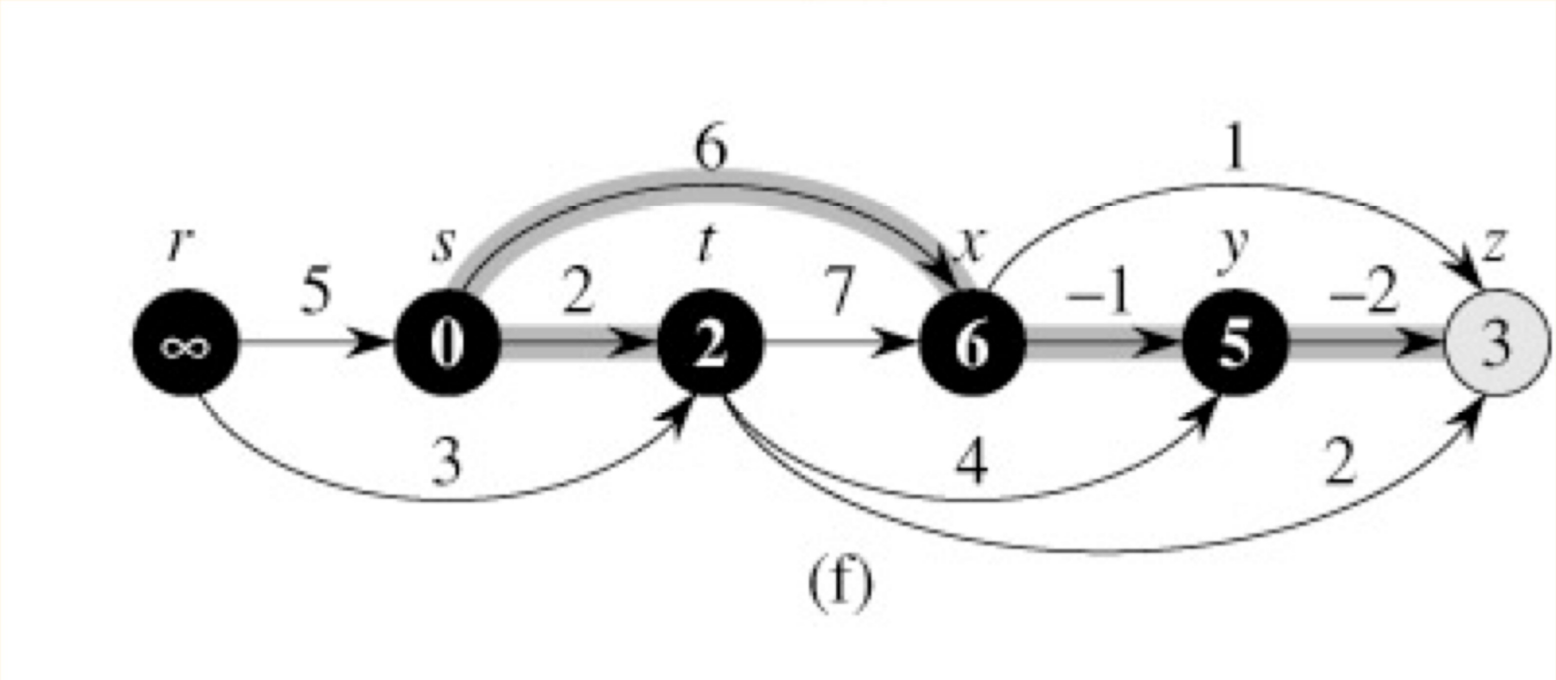


(d)

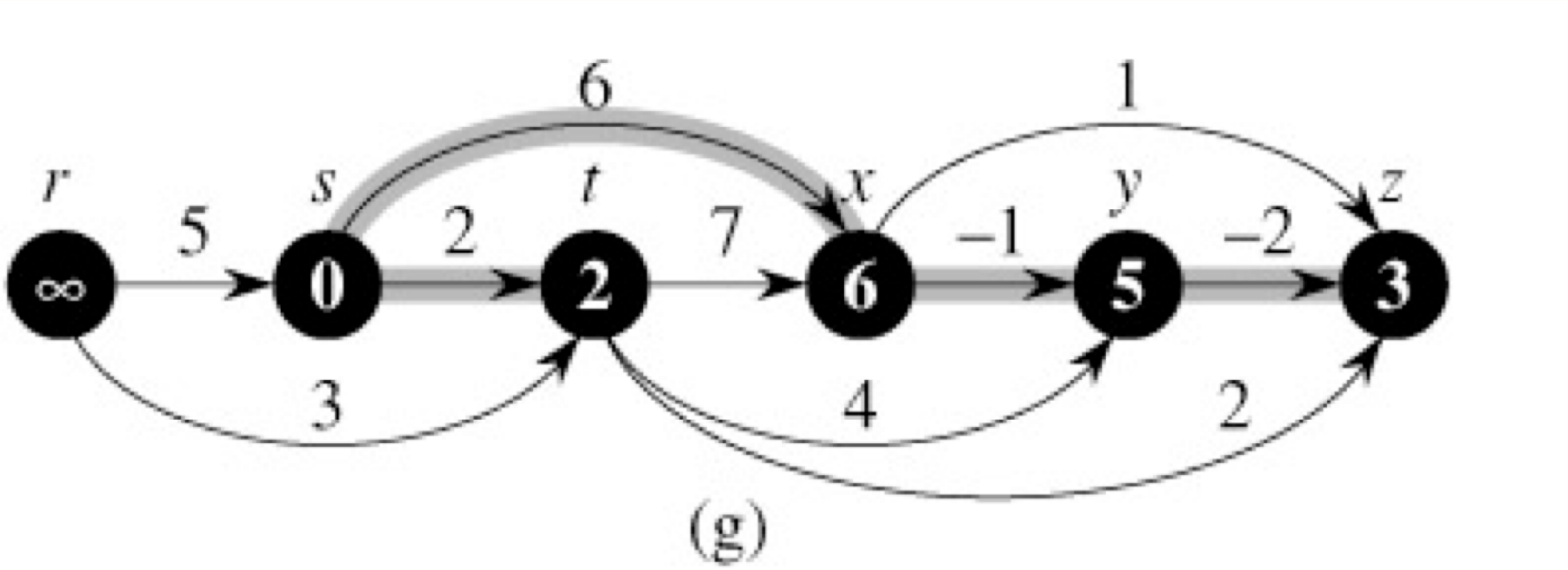
Example



Example



Example



Correctness: Path relaxation property

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 2. Single-Source Shortest Path on a General Graph (May Contain Cycles)

- This is fundamentally harder, because the first paths we discover may not be the shortest (not monotonic).

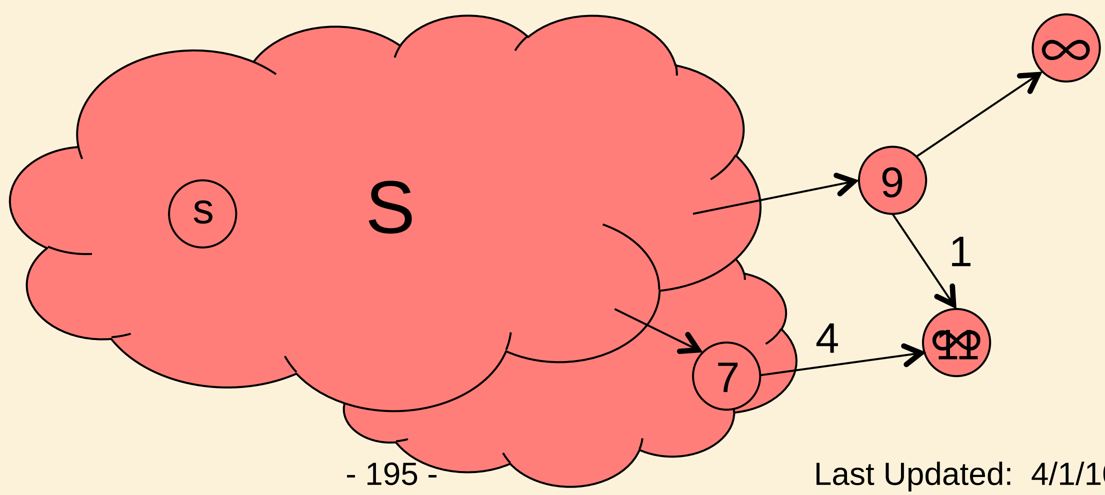
Dijkstra's algorithm (E. Dijkstra,1959)

- Applies to general, weighted, directed or undirected graph (may contain cycles).
- But weights must be non-negative. (But they can be 0!)
- 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.

Edsger Dijkstra

Dijkstra's Algorithm: Operation

- We grow a "**cloud**" *S* of vertices, beginning with s and eventually covering all the vertices
- We store with each vertex v a label $d(v)$ representing the distance of v from s in the subgraph consisting of the cloud *S* and its adjacent vertices
- At each step
  - We add to the cloud *S* the vertex u outside the cloud with the smallest distance label, $d(u)$
  - We update the labels of the vertices adjacent to u



Dijkstra's algorithm

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S ← Ø
Q ← V[G]
while Q ≠ Ø
      do u ← EXTRACT-MIN(Q)
         S ← S ∪ {u}
         for each vertex v ∈ 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

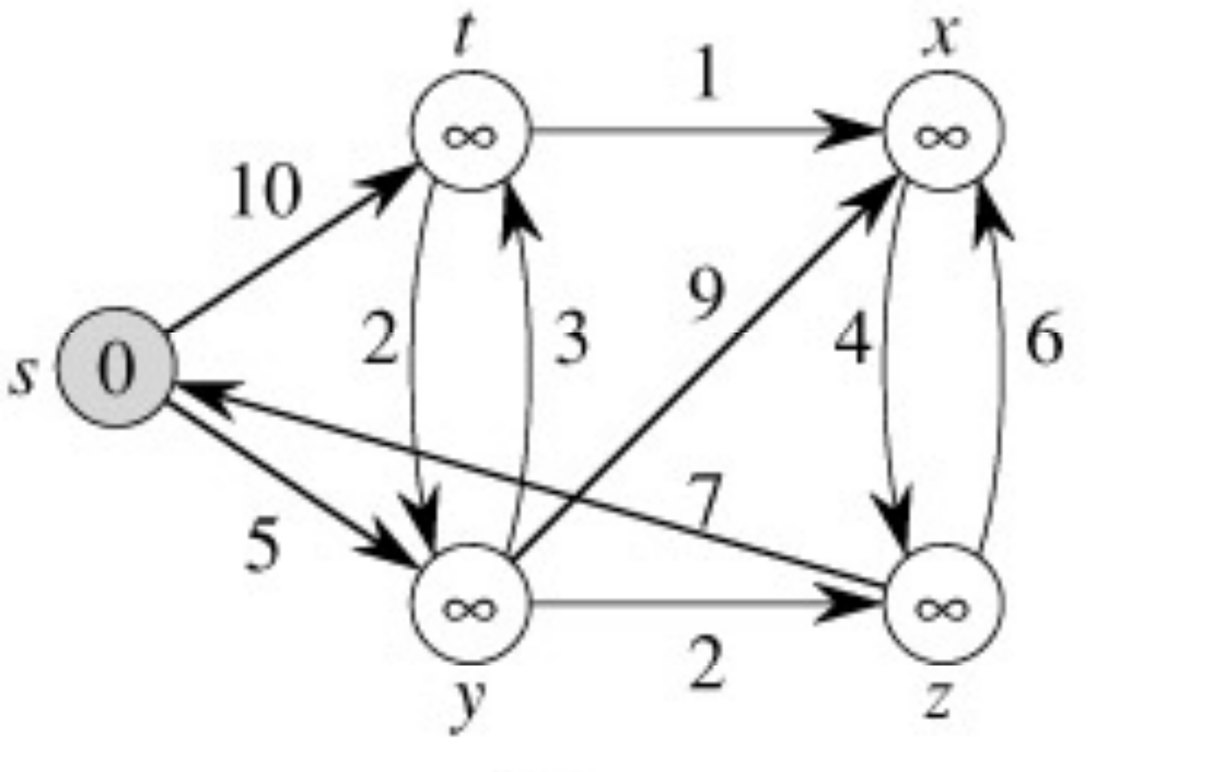
```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)        O(V)
S ← Ø
Q ← V[G]
while Q ≠ Ø
    do u ← EXTRACT-MIN(Q)             O(logV)×O(V) iterations
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)          O(logV)×O(E) iterations
```

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

Example

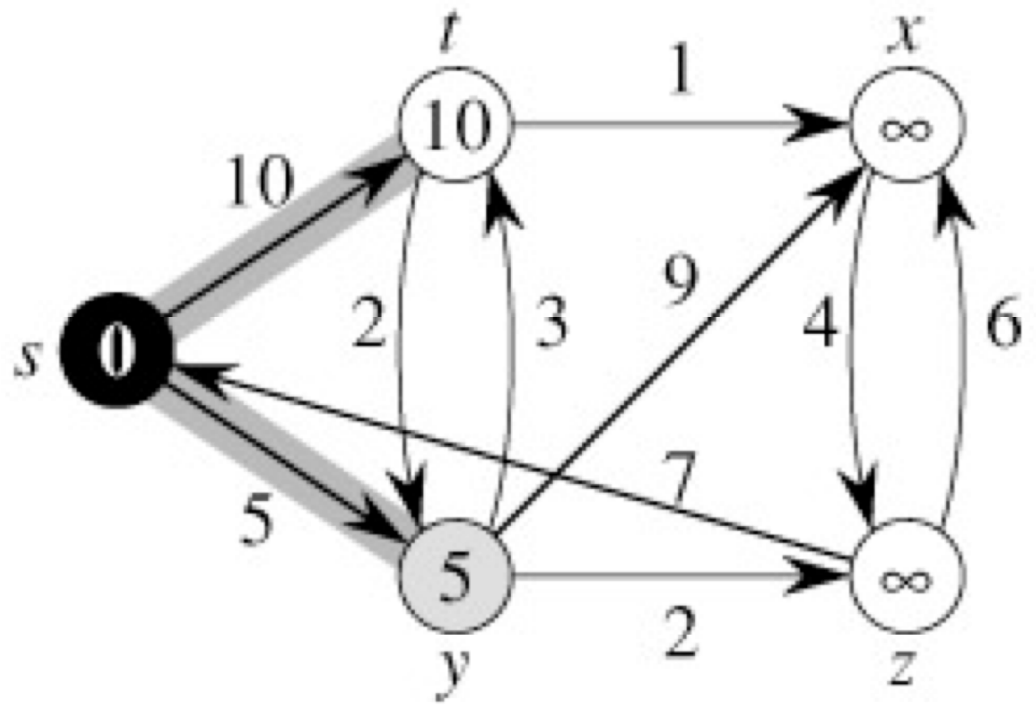
Key:

White $\Leftrightarrow$ Vertex $\in Q = V - S$
Grey $\Leftrightarrow$ Vertex = min(Q)
Black $\Leftrightarrow$ Vertex $\in S$, Off Queue



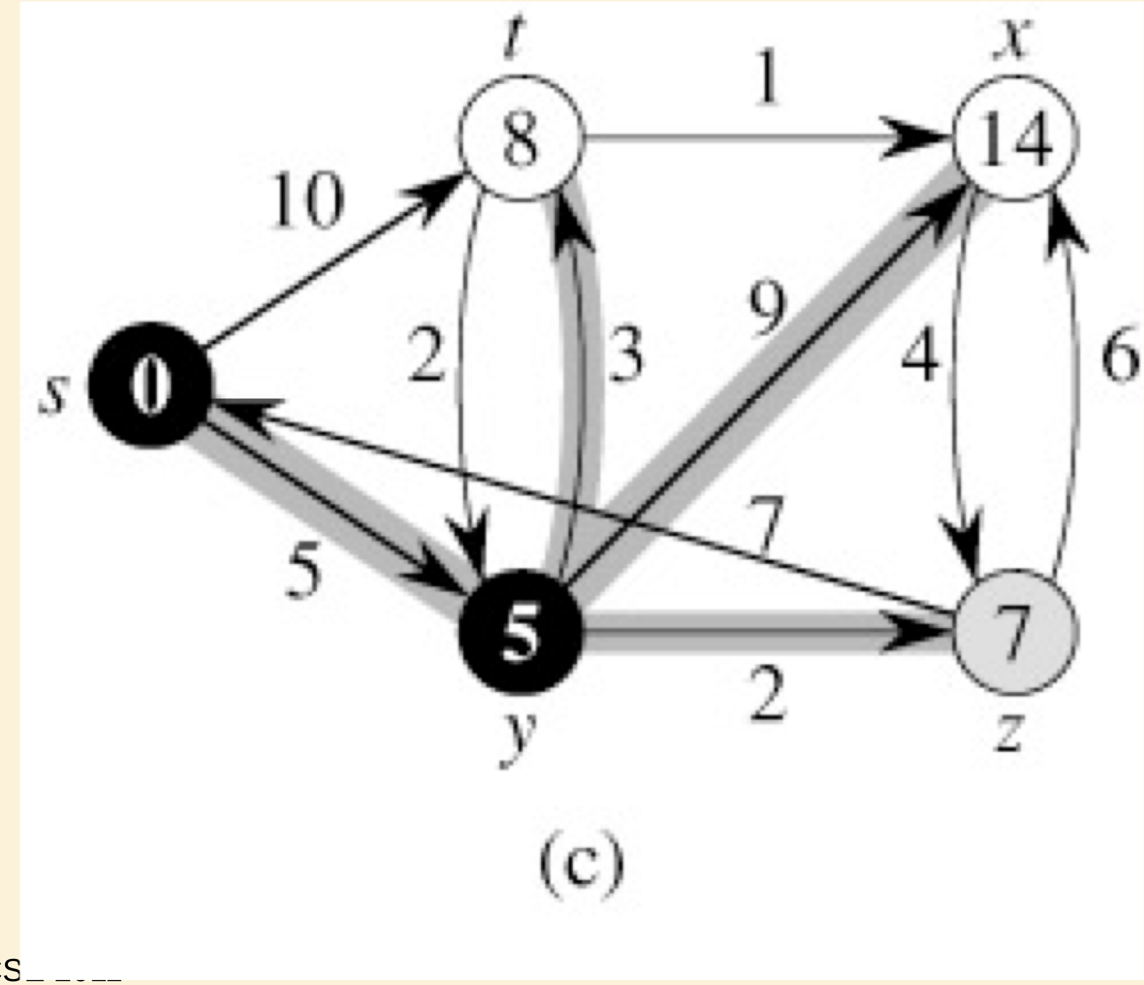
(a)

Example

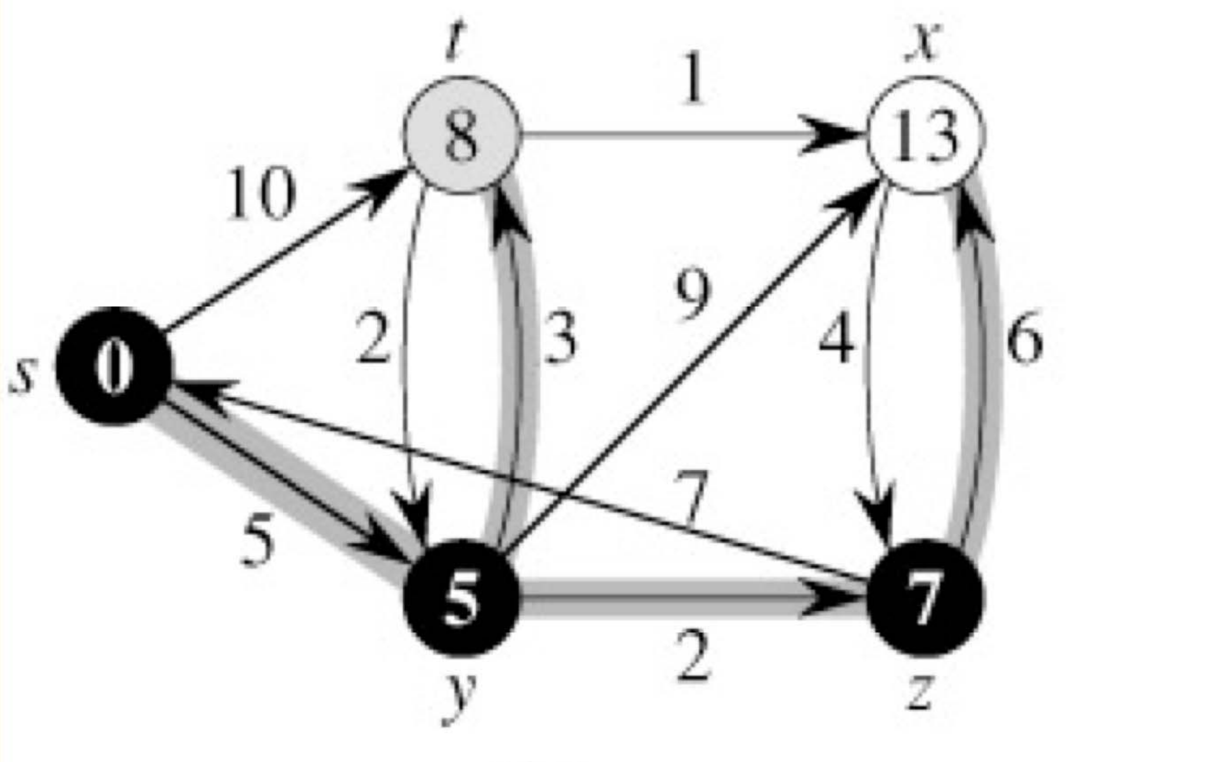


(b)

Example

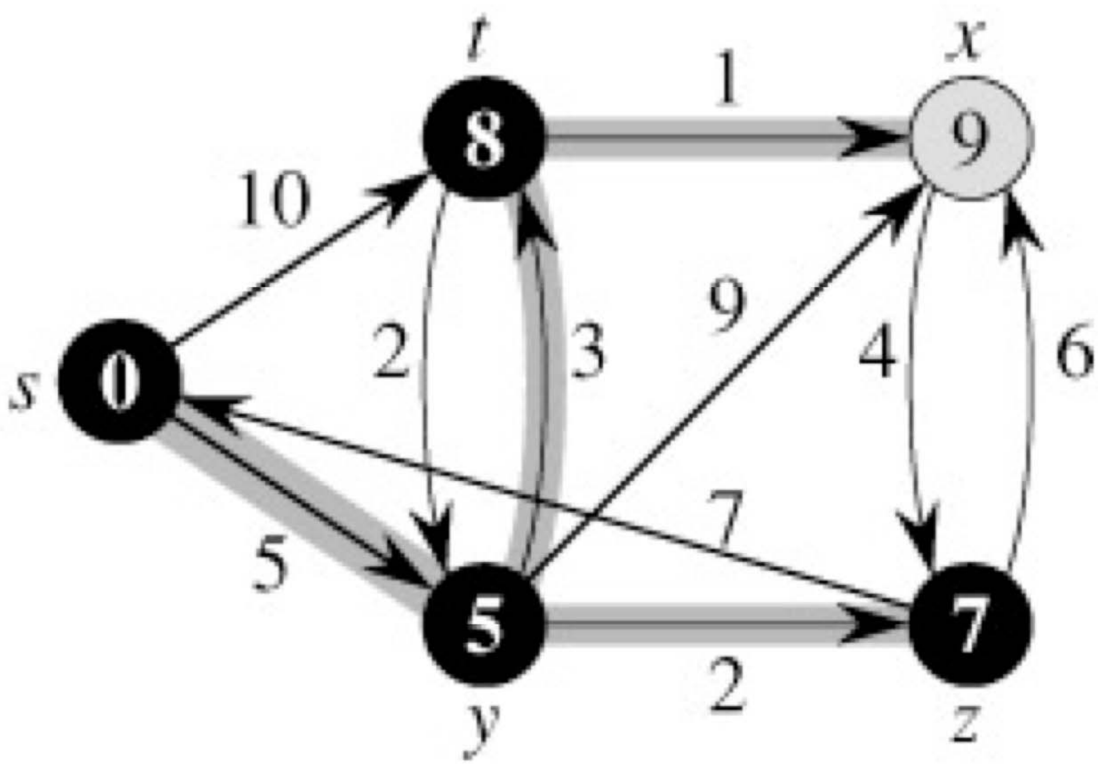


Example



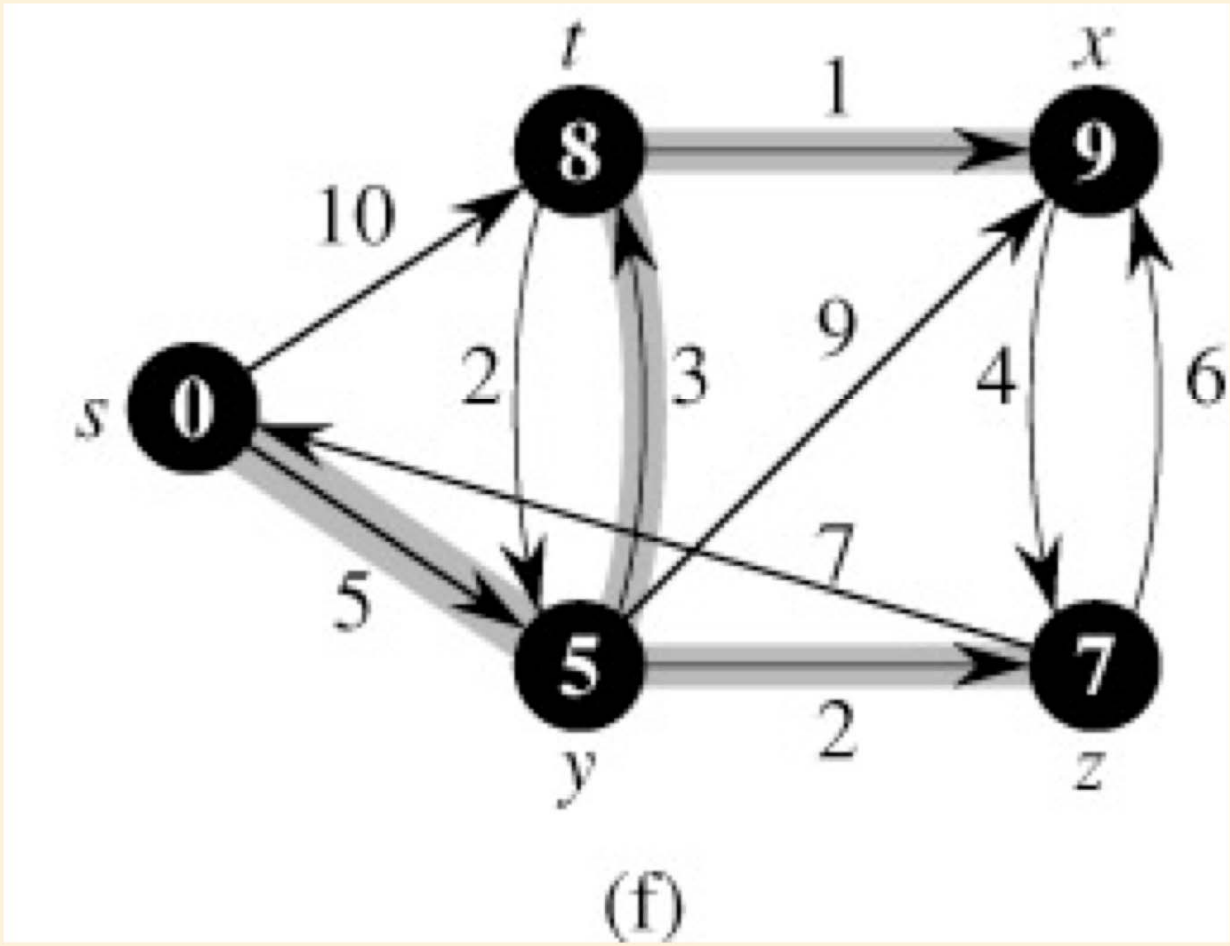
(d)

Example

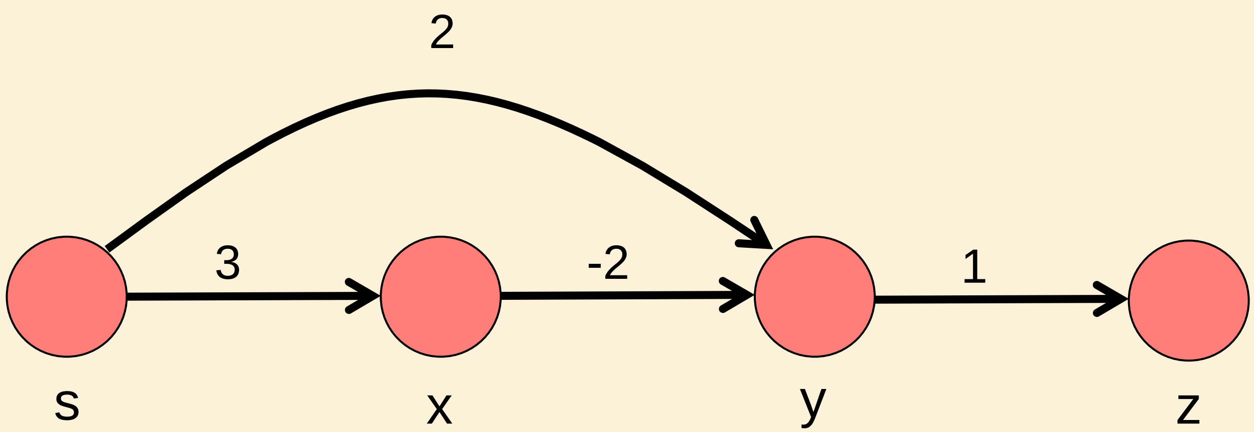


(e)

Example



Djikstra's Algorithm Cannot Handle Negative Edges



Correctness of Dijkstra's algorithm

```
DIJKSTRA(G, w, s)
  INITIALIZE-SINGLE-SOURCE(G, s)
  S ← Ø
  Q ← V[G]
  while Q ≠ Ø
      do u ← EXTRACT-MIN(Q)
         S ← S ∪ {u}
         for each vertex v ∈ 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 $d[v] = d[u] + w(u,v)$

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

  A valid path from s to 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

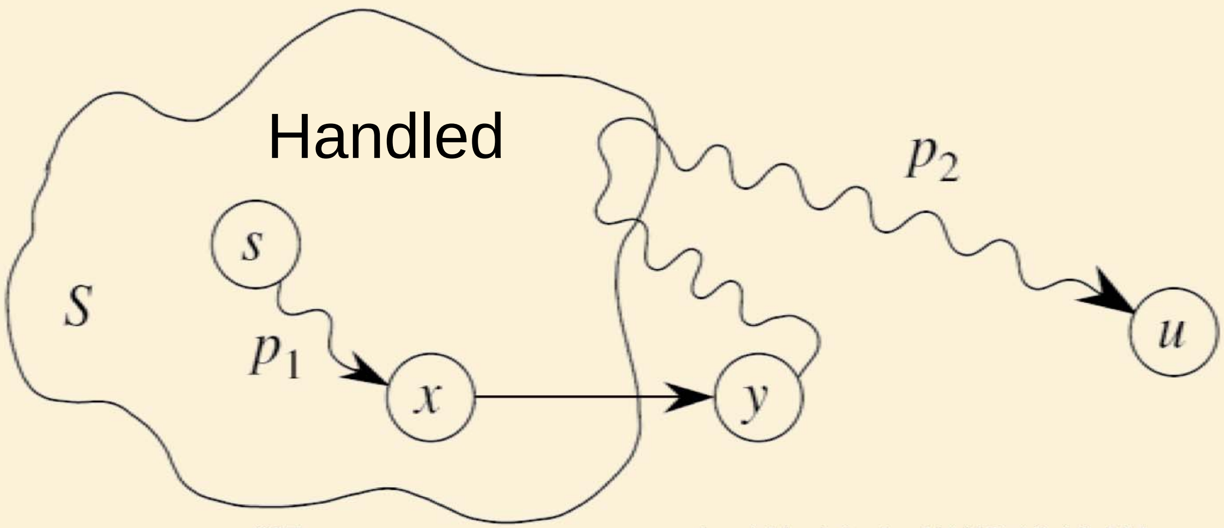
Claim: $d[y] = \delta(s,y)$ when u is added to S.

Optimal substructure property!

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

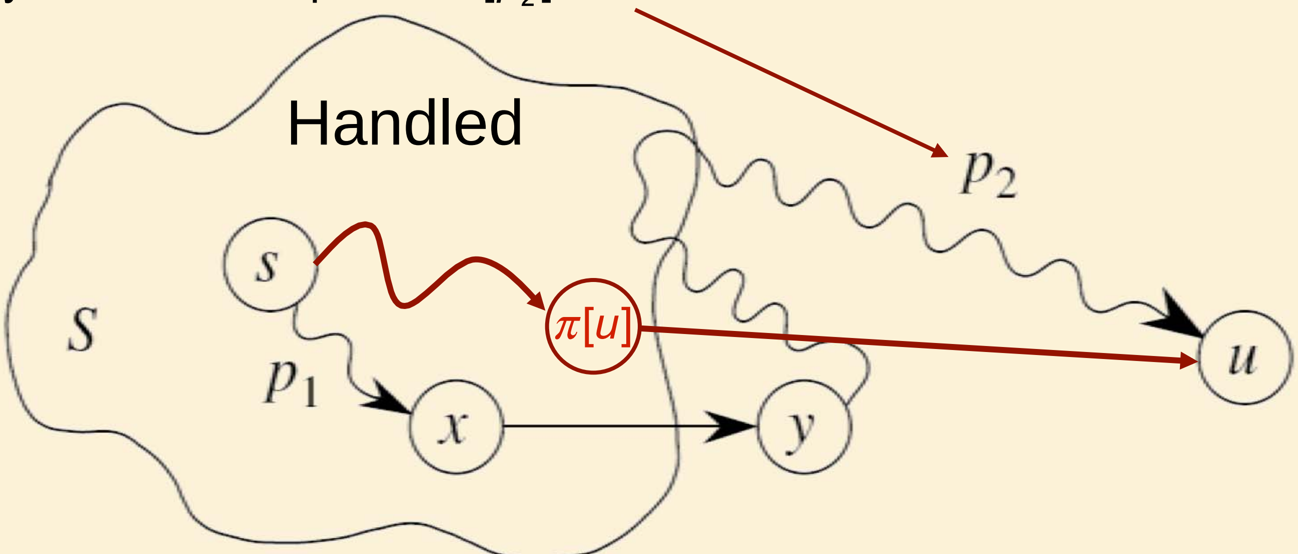
Thus when u is added to S, $d[u] = \delta(s,u)$

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S ← Ø
Q ← V[G]
while Q ≠ Ø
      do u ← EXTRACT-MIN(Q)
         S ← S ∪ {u}
         for each vertex v ∈ Adj[u]
             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)

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)

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. ?