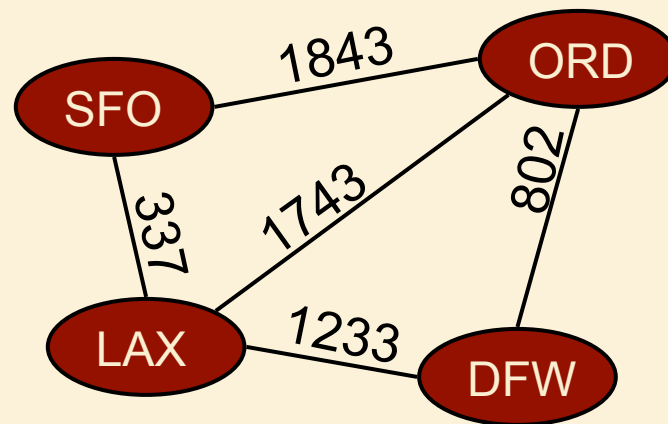
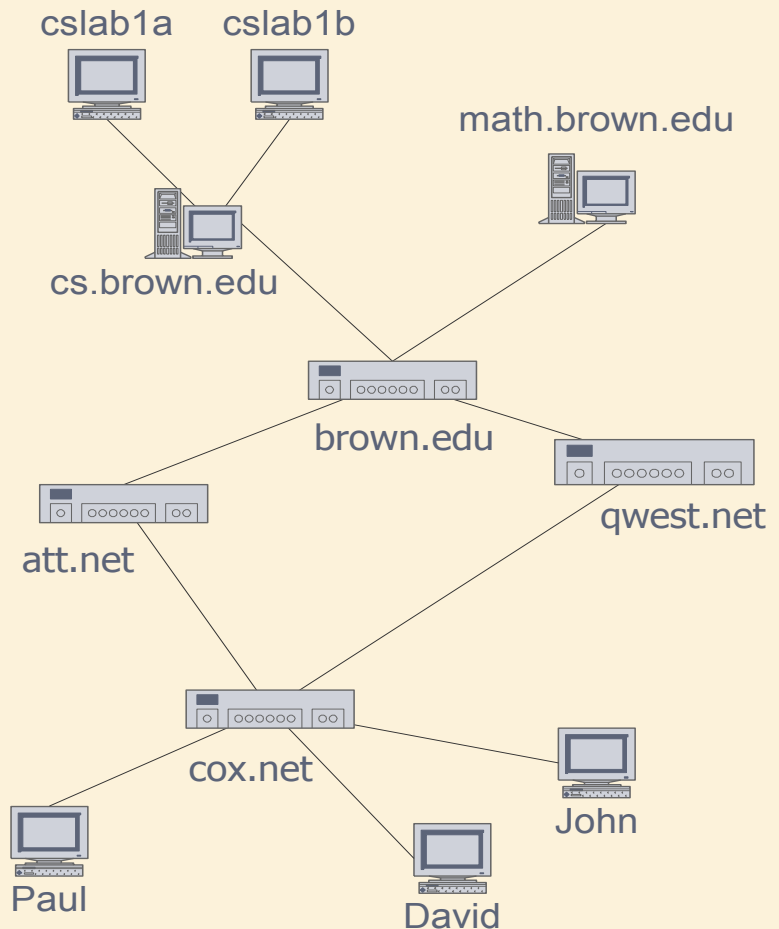


Graphs



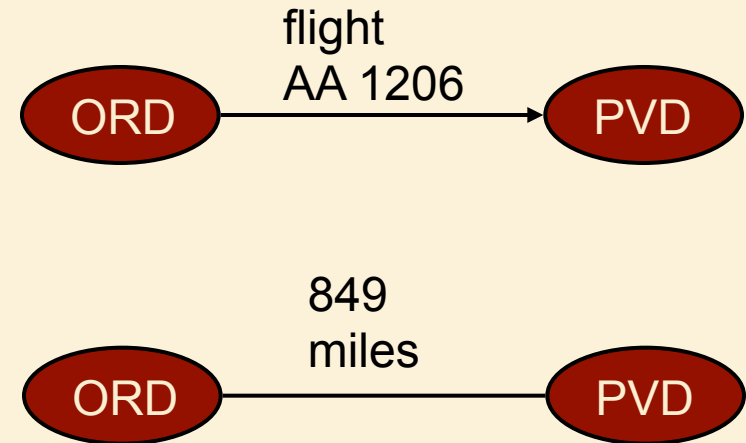
Applications

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



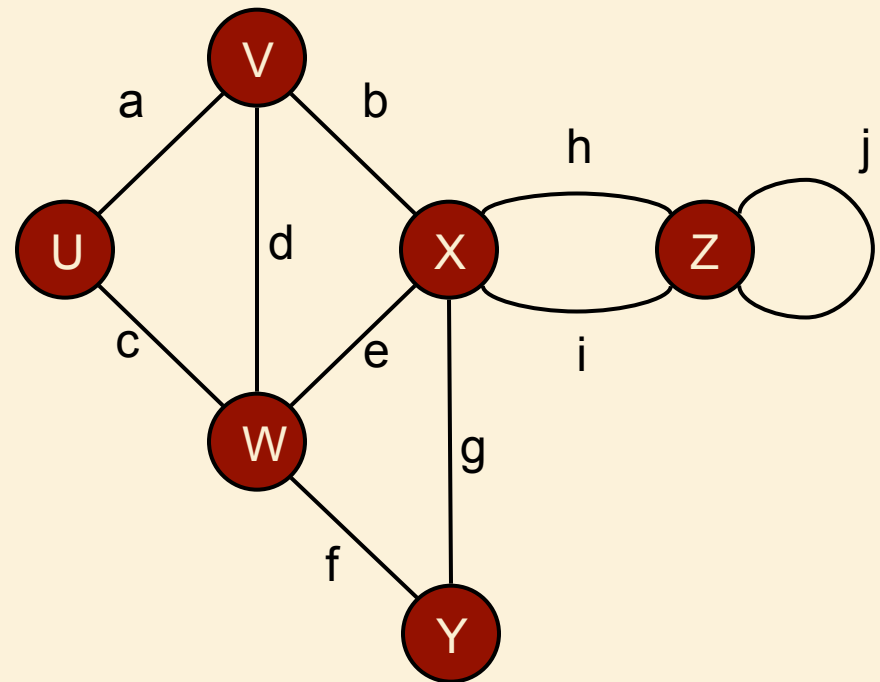
Edge Types

- 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



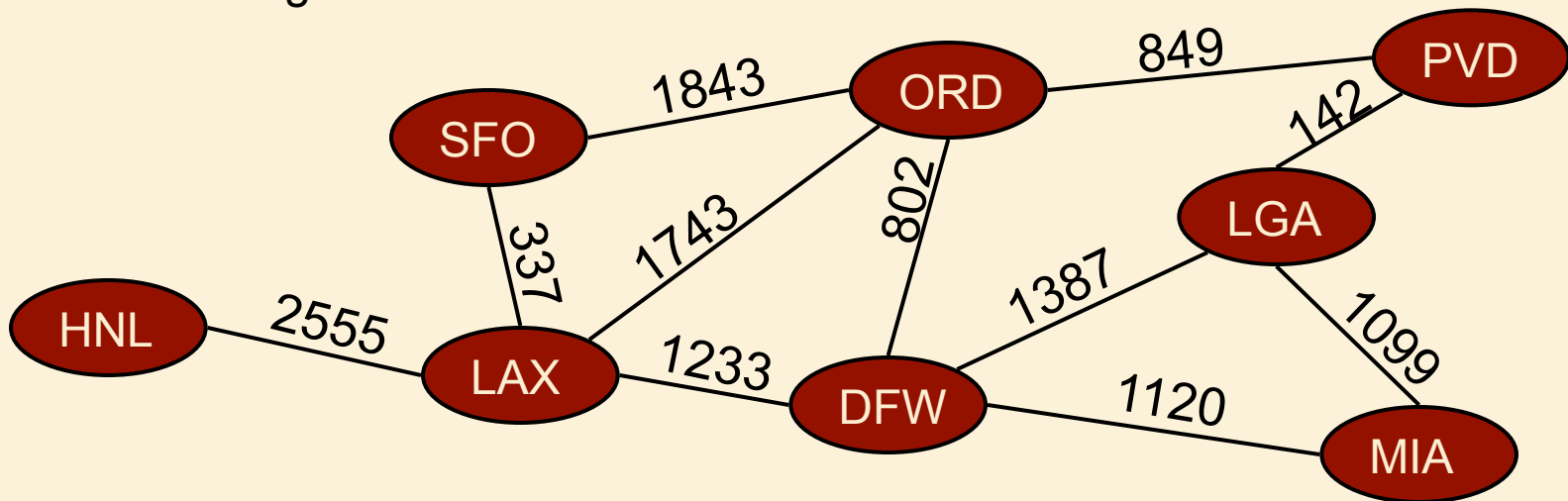
Vertices and Edges

- 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



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



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



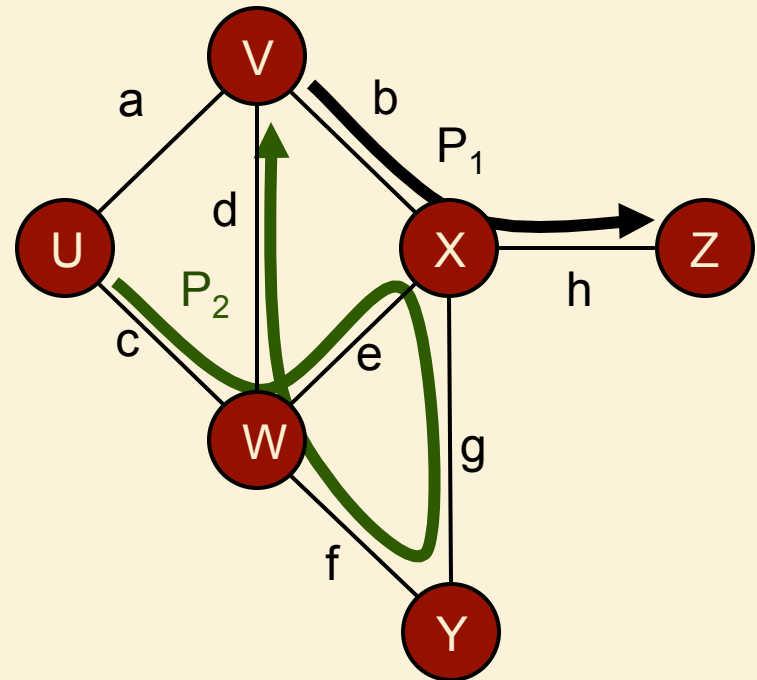
Simple path

- 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

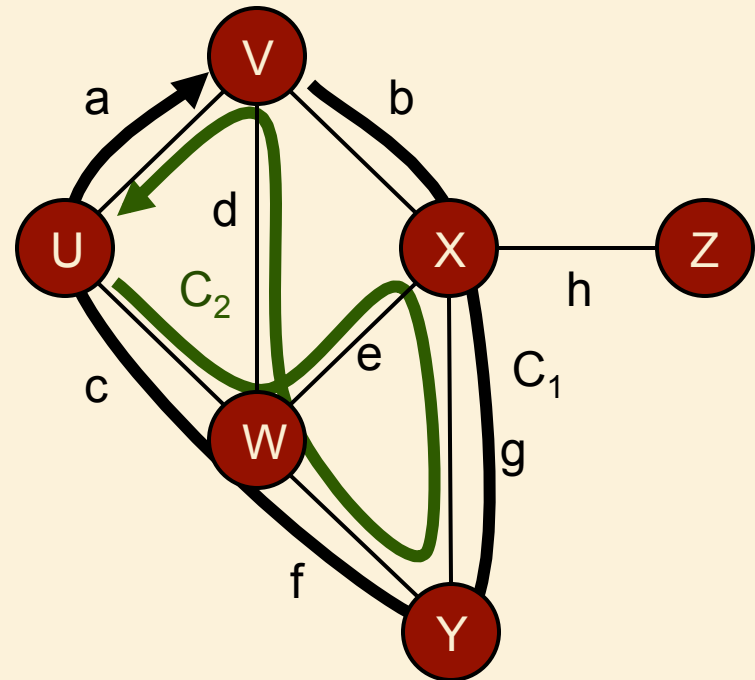
- ❑ circular sequence of alternating vertices and edges
- ❑ each edge is preceded and followed by its endpoints

➤ Simple cycle

- ❑ cycle such that all its vertices and edges are distinct

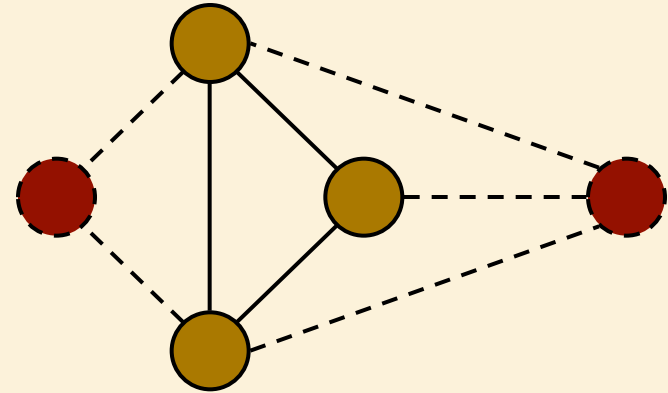
➤ Examples

- ❑ $C_1 = (V, b, X, g, Y, f, W, c, U, a, \leftarrow V)$ is a simple cycle
- ❑ $C_2 = (U, c, W, e, X, g, Y, f, W, d, V, a, \leftarrow U)$ is a cycle that is not simple

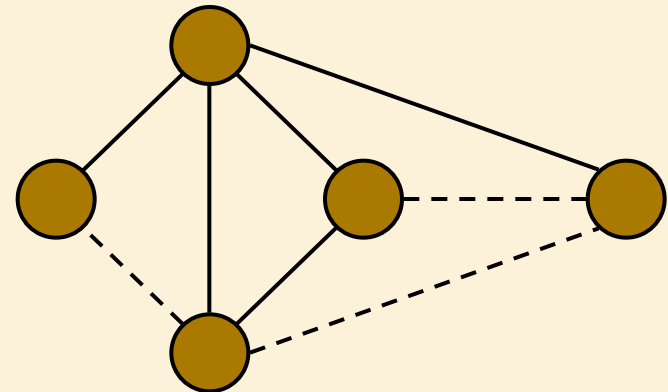


Subgraphs

- 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



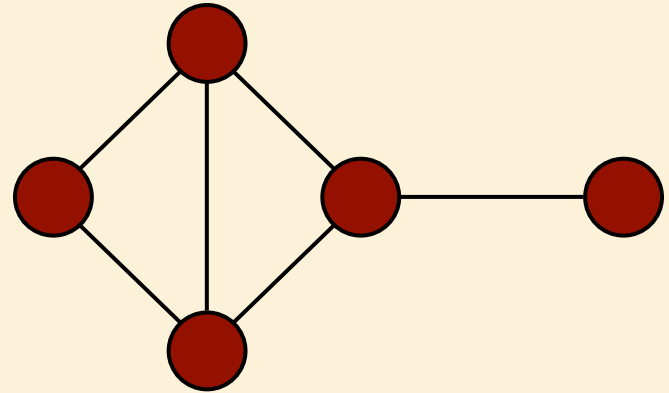
Subgraph



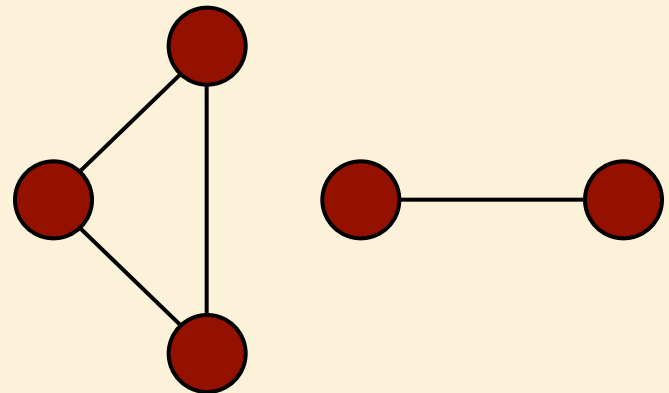
Spanning subgraph

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

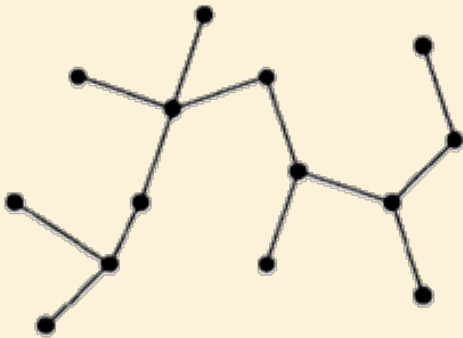


Connected graph

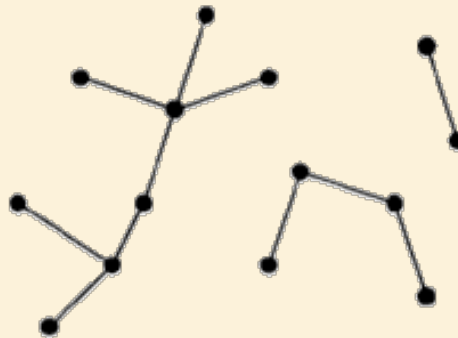


Non connected graph with two connected components

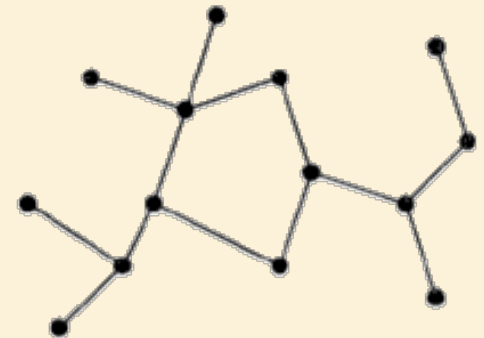
Trees



Tree



Forest



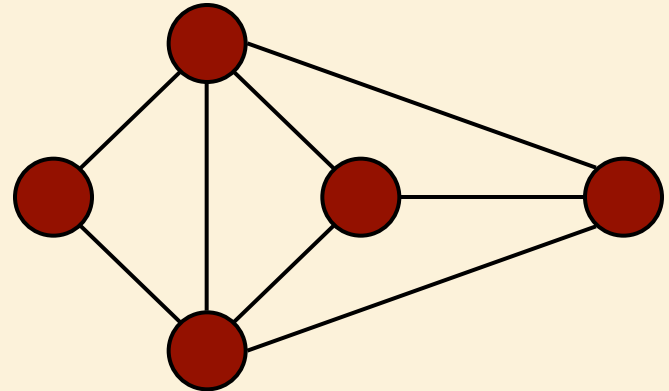
Graph with Cycle

A tree is a **connected**, **acyclic**, **undirected** graph.

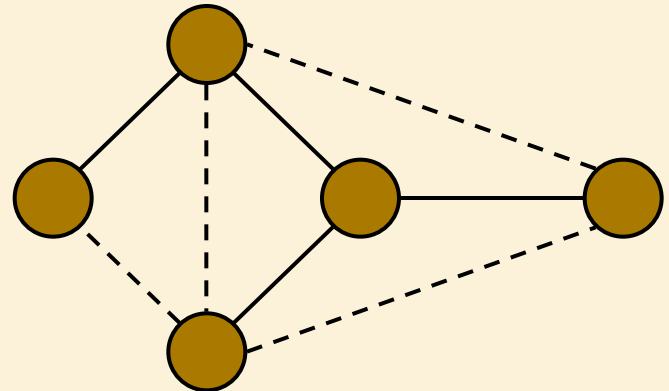
A forest is a **set** of trees (not necessarily connected)

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



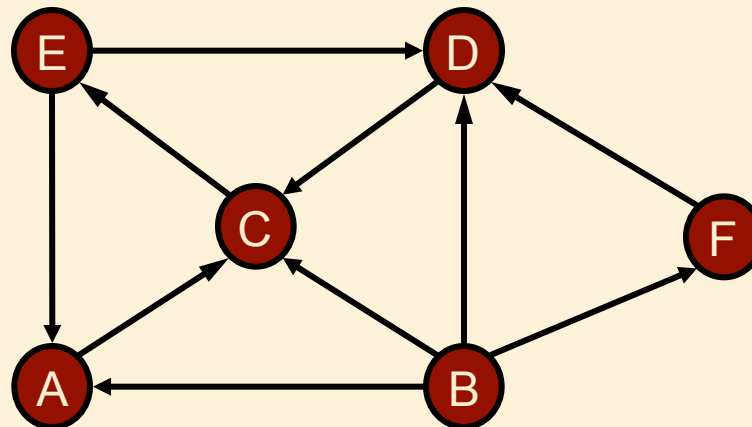
Spanning tree

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

Notation

$|V|$ number of vertices

$|E|$ number of edges

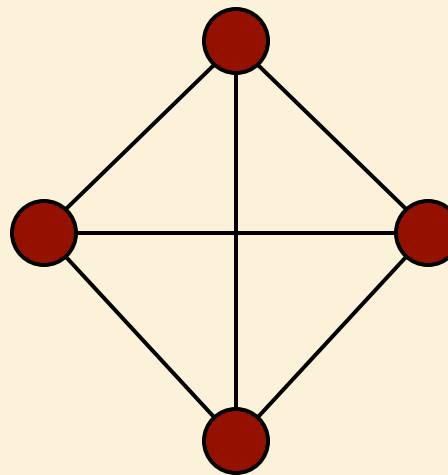
$\deg(v)$ degree of vertex v

Property 2

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

$$|E| \leq |V|(|V| - 1)/2$$

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



Example

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

Q: What is the bound for a digraph?

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

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

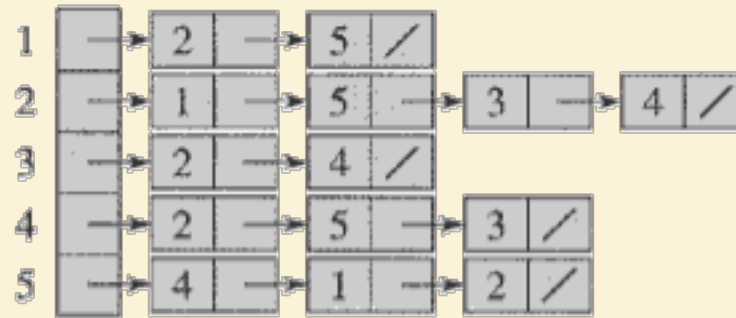
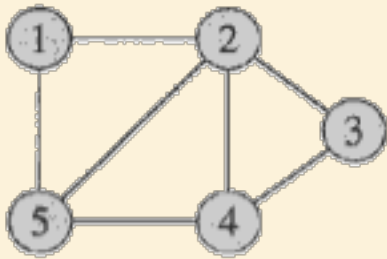
➤ Additional methods:

- ❑ `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 $| \cdot |$ in asymptotic notation, e.g. $O(V+E)$.

Implementing a Graph (Simplified)



Adjacency List

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

Adjacency 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

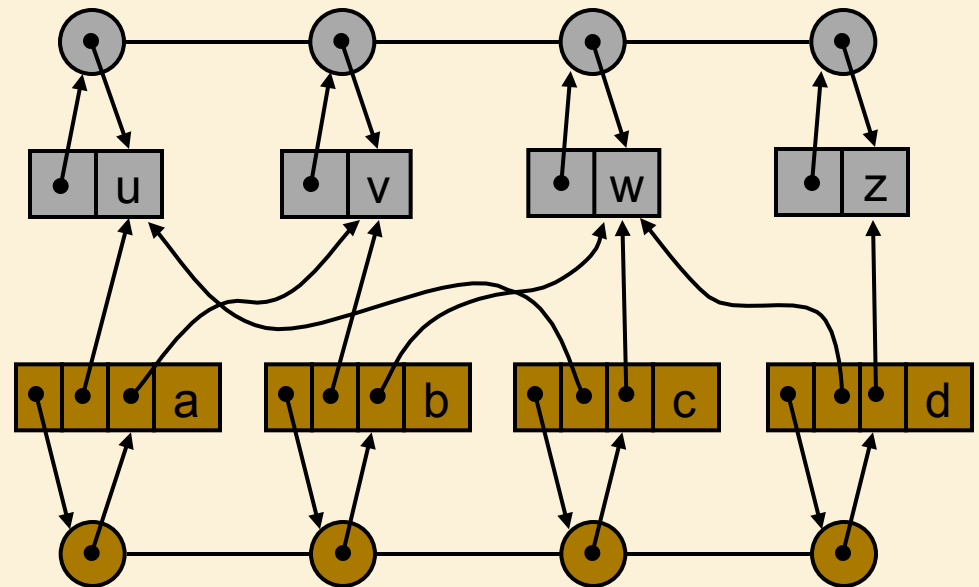
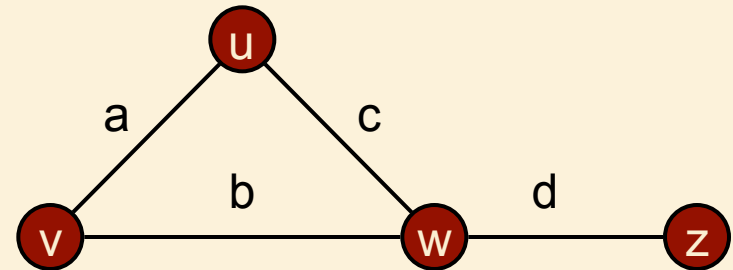
- element
- origin vertex object
- destination vertex object
- reference to position in edge sequence

➤ Vertex sequence

- sequence of vertex objects

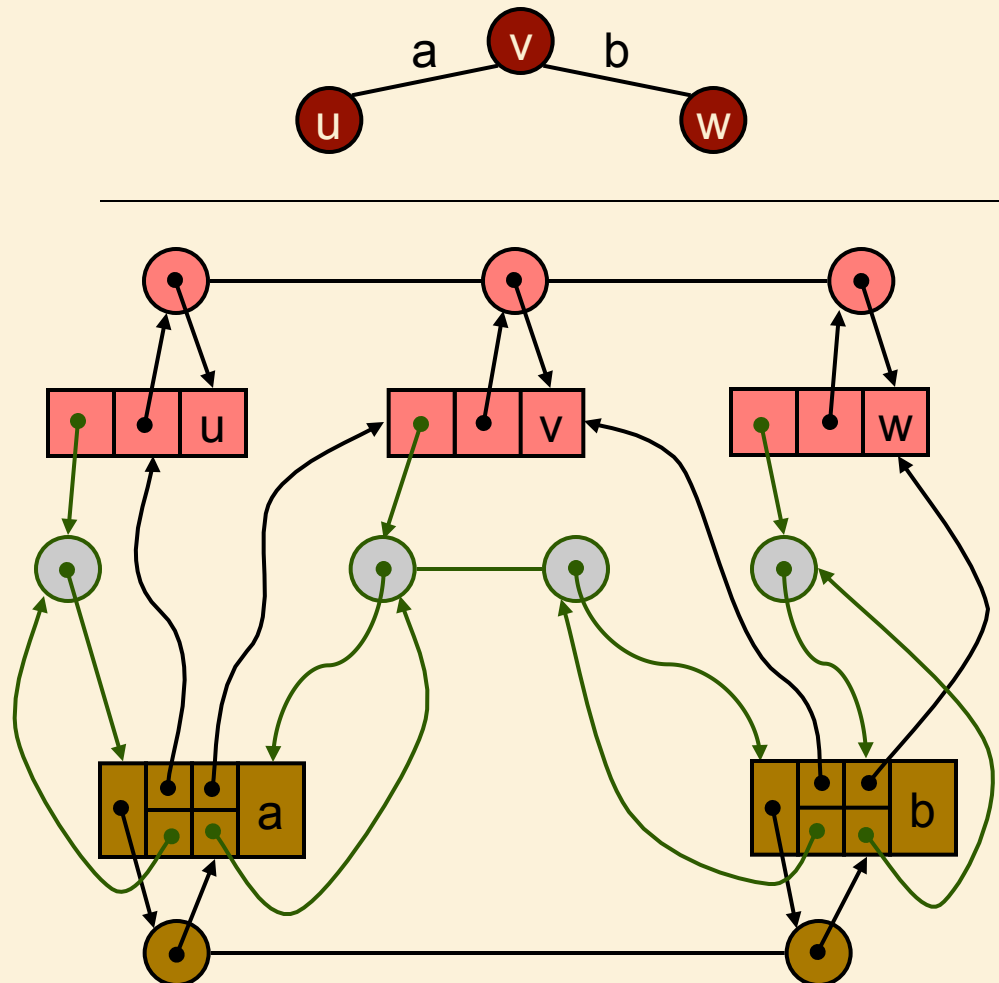
➤ Edge sequence

- sequence of edge objects



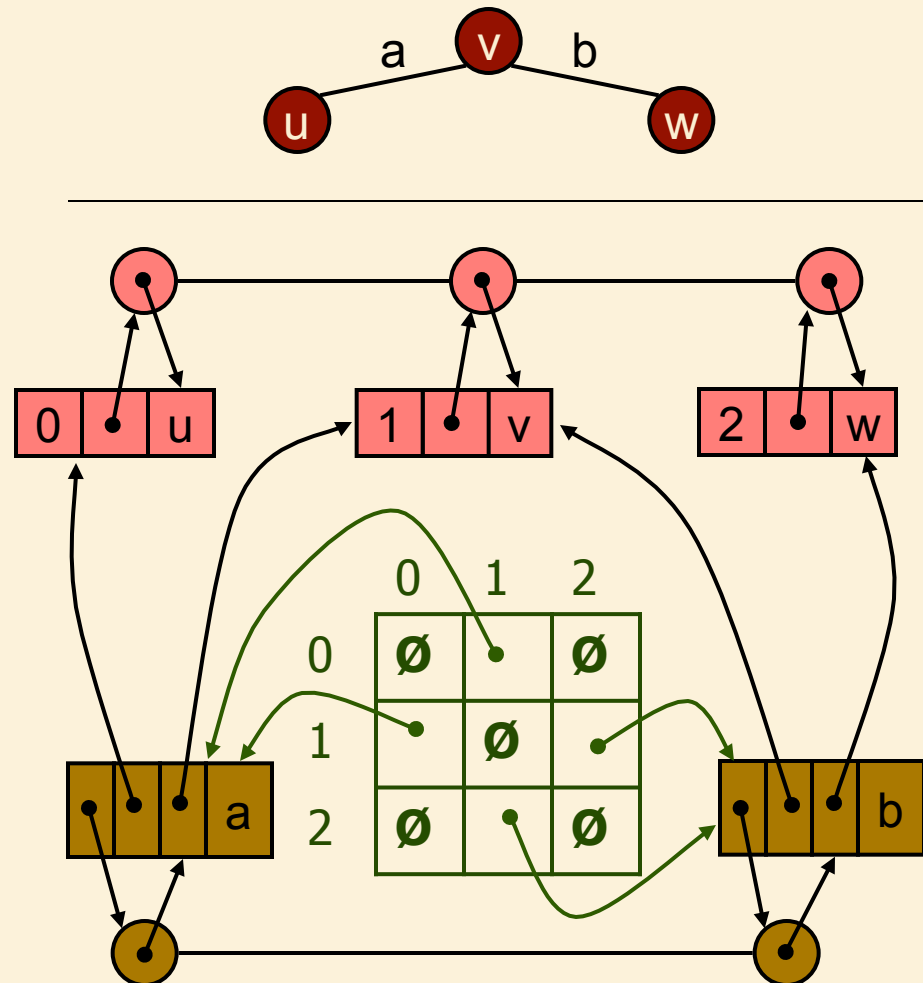
Adjacency List Structure

- 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-adjacent vertices

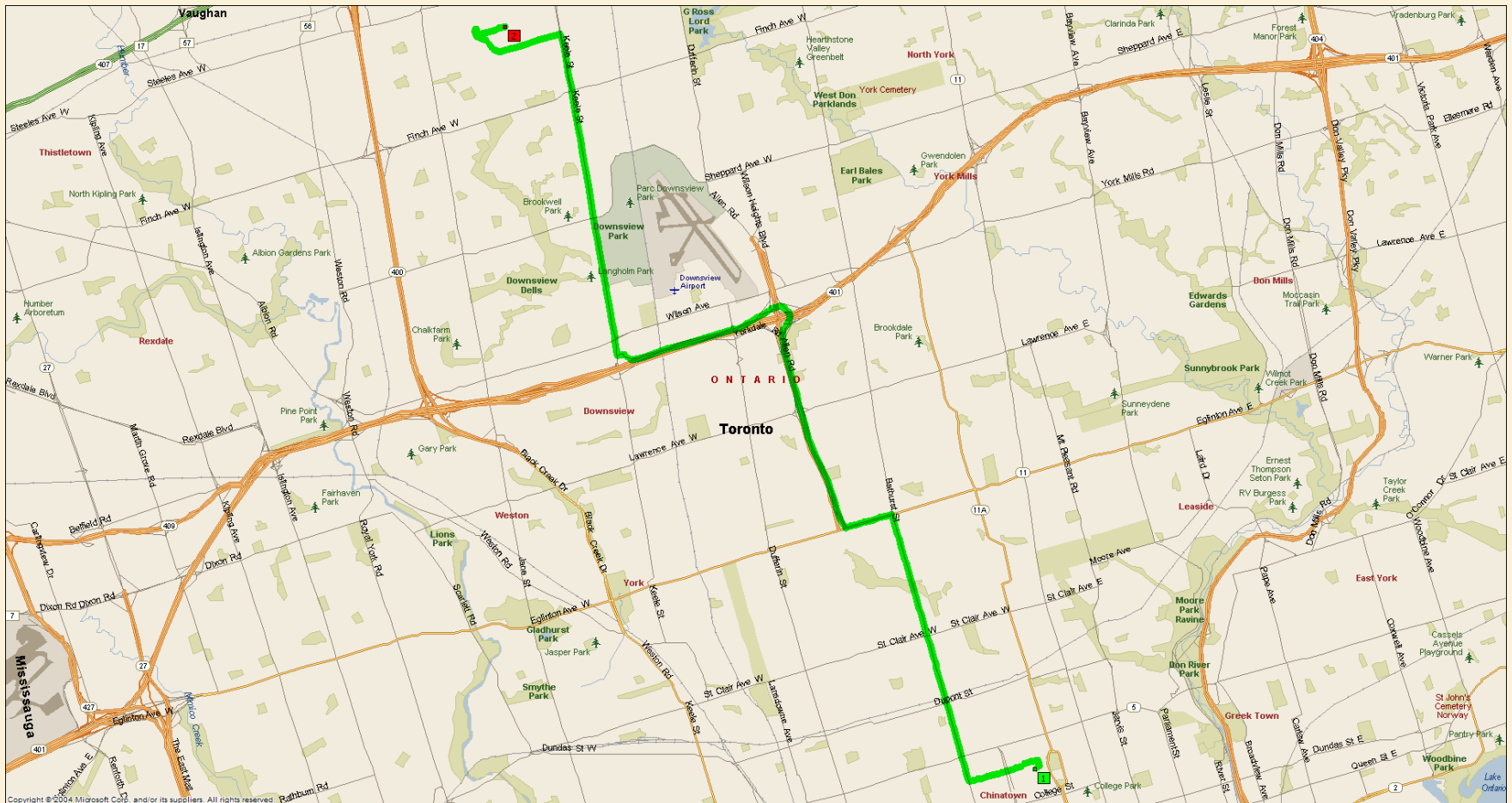


Asymptotic Performance

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

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

Graph Search Algorithms



Depth First Search (DFS)

➤ Idea:

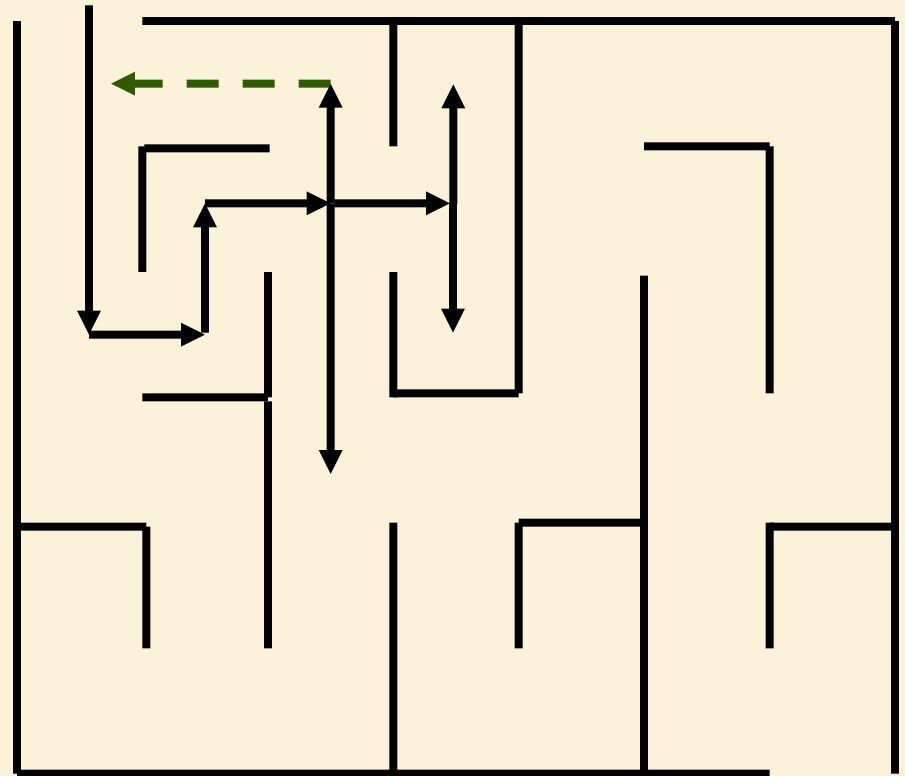
- ❑ Continue searching “deeper” into the graph, until we get stuck.
- ❑ If all the edges leaving v have been explored we “backtrack” to the vertex from which v was discovered.
- ❑ Analogous to Euler tour for trees

➤ Used to help solve many graph problems, including

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



- 

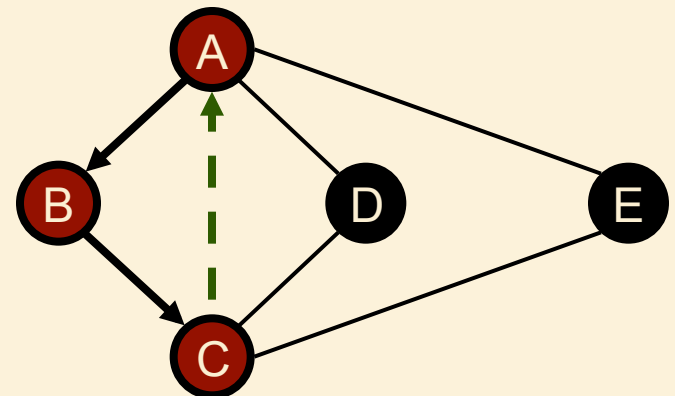
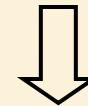
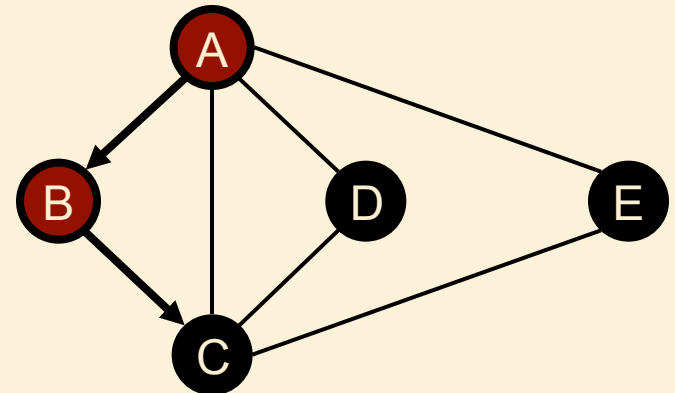
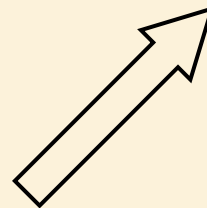
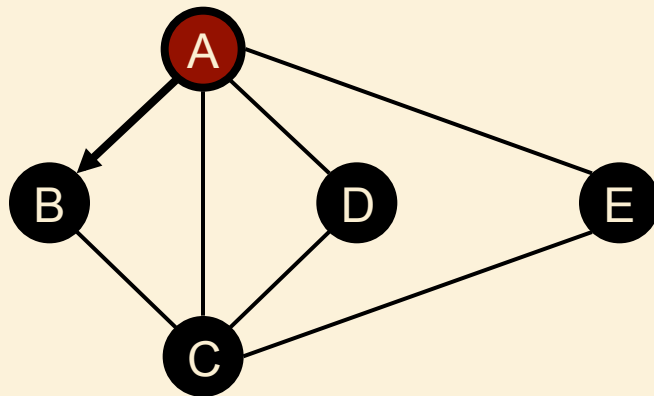
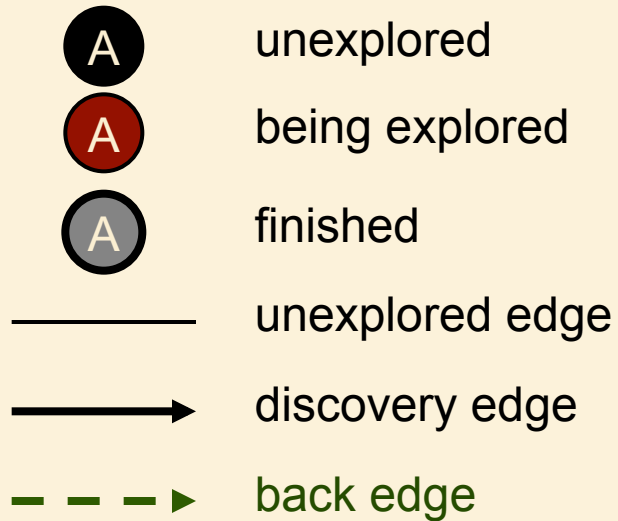


Depth-First Search

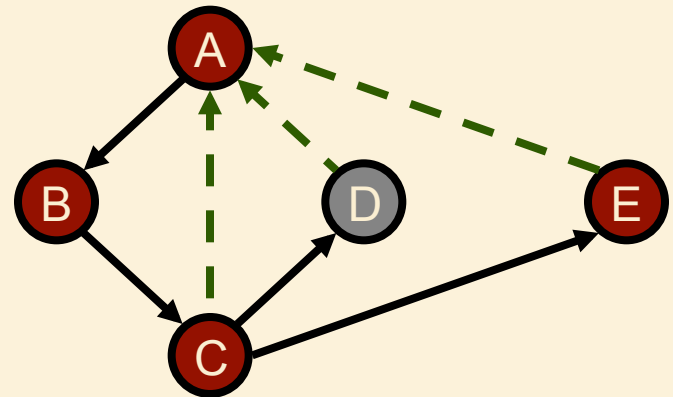
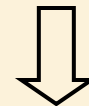
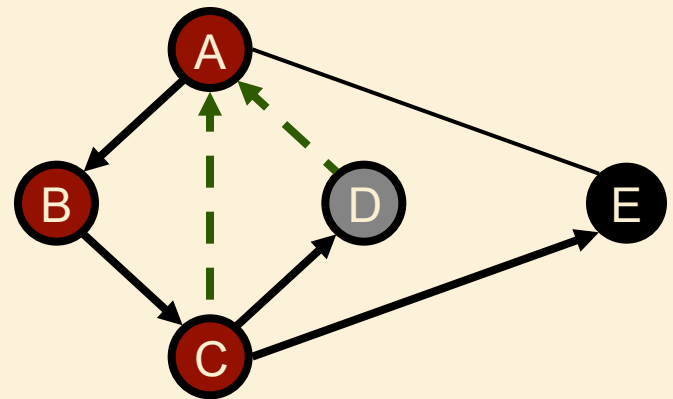
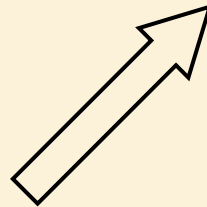
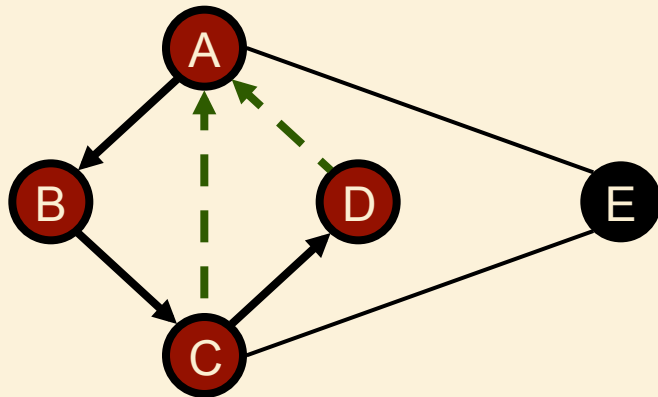
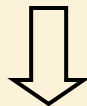
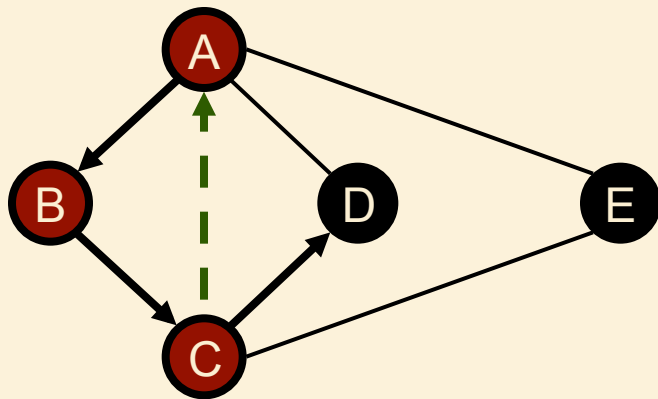
Input: Graph $G = (V, E)$ (directed or undirected)

- 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

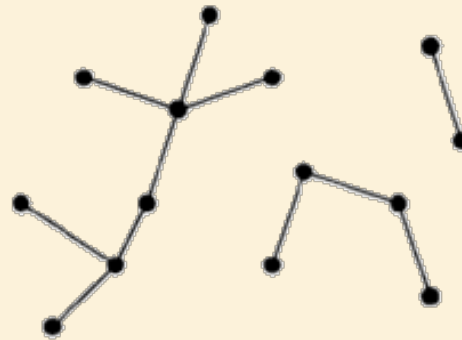
for each vertex $u \in V[G]$

color[u] = BLACK //initialize vertex

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

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

DFS-Visit(u)



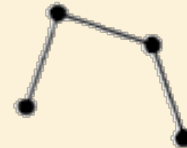
DFS Algorithm Pattern

DFS-Visit (u)

Precondition: vertex u is undiscovered

Postcondition: all vertices reachable from u have been processed

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



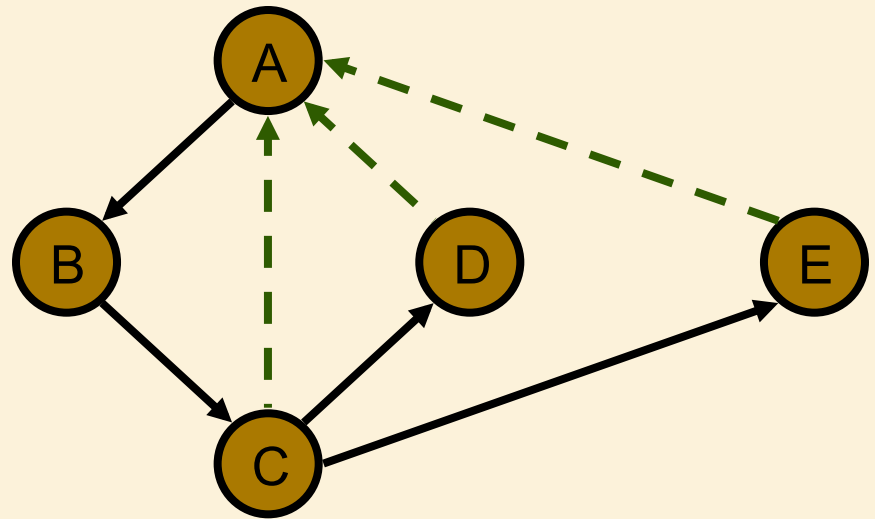
Properties of DFS

Property 1

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

Property 2

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



DFS Algorithm Pattern

DFS(G)

Precondition: G is a graph

Postcondition: all vertices in G have been visited

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

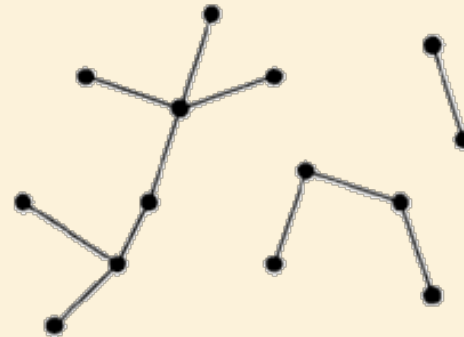
 color[u] = BLACK //initialize vertex

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

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

 DFS-Visit(u)

} total work
= $\theta(V)$



DFS Algorithm Pattern

DFS-Visit (u)

Precondition: vertex u is undiscovered

Postcondition: all vertices reachable from u have been processed

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

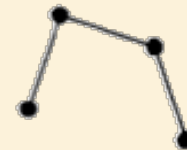
for each $v \in \text{Adj}[u]$ //explore edge (u,v)

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

DFS-Visit(v)

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

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



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

(assuming adjacency list structure)

Variants of Depth-First Search

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

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

Output: 2 timestamps on each vertex:

$d[v]$ = discovery time.

$f[v]$ = finishing time.

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

DFS Algorithm with Discovery and Finish Times

DFS(G)

Precondition: G is a graph

Postcondition: all vertices in G have been visited

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

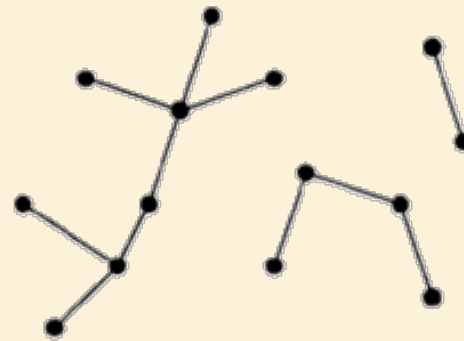
 color[u] = BLACK //initialize vertex

time $\leftarrow$ 0

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

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

 DFS-Visit(u)



DFS Algorithm with Discovery and Finish Times

DFS-Visit (u)

Precondition: vertex u is undiscovered

Postcondition: all vertices reachable from u have been processed

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

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

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

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

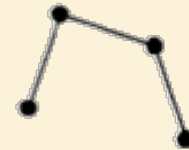
if $\text{color}[v] = \text{BLACK}$

DFS-Visit(v)

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

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

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



Other Variants of Depth-First Search

- The DFS Pattern can also be used to
 - ❑ Compute a forest of spanning trees (one for each call to DFS-visit) encoded in a predecessor list $\pi[u]$
 - ❑ Label edges in the graph according to their role in the search (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

End of Lecture

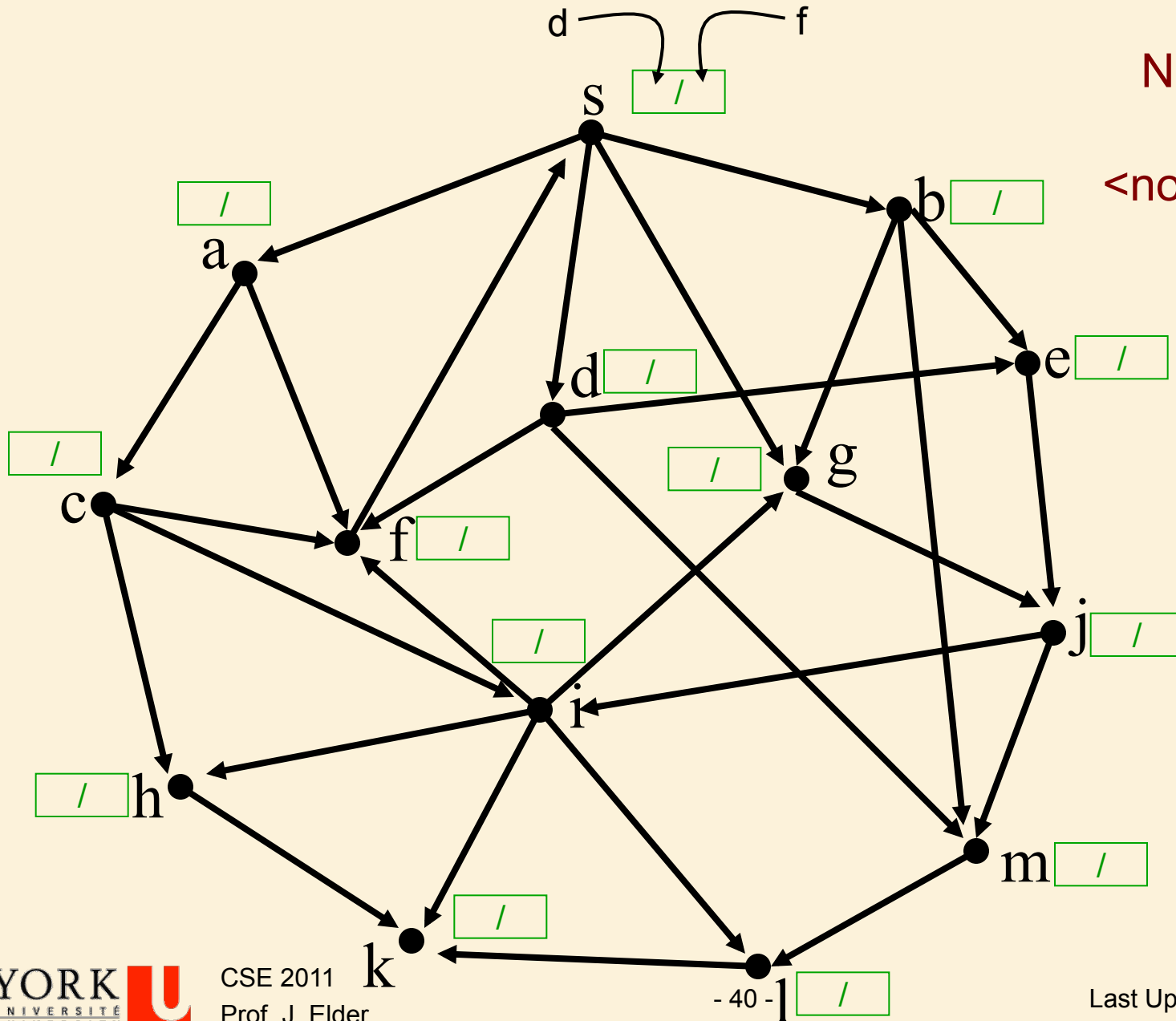
Tuesday, Mar 20, 2012

Example DFS on Directed Graph

DFS

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

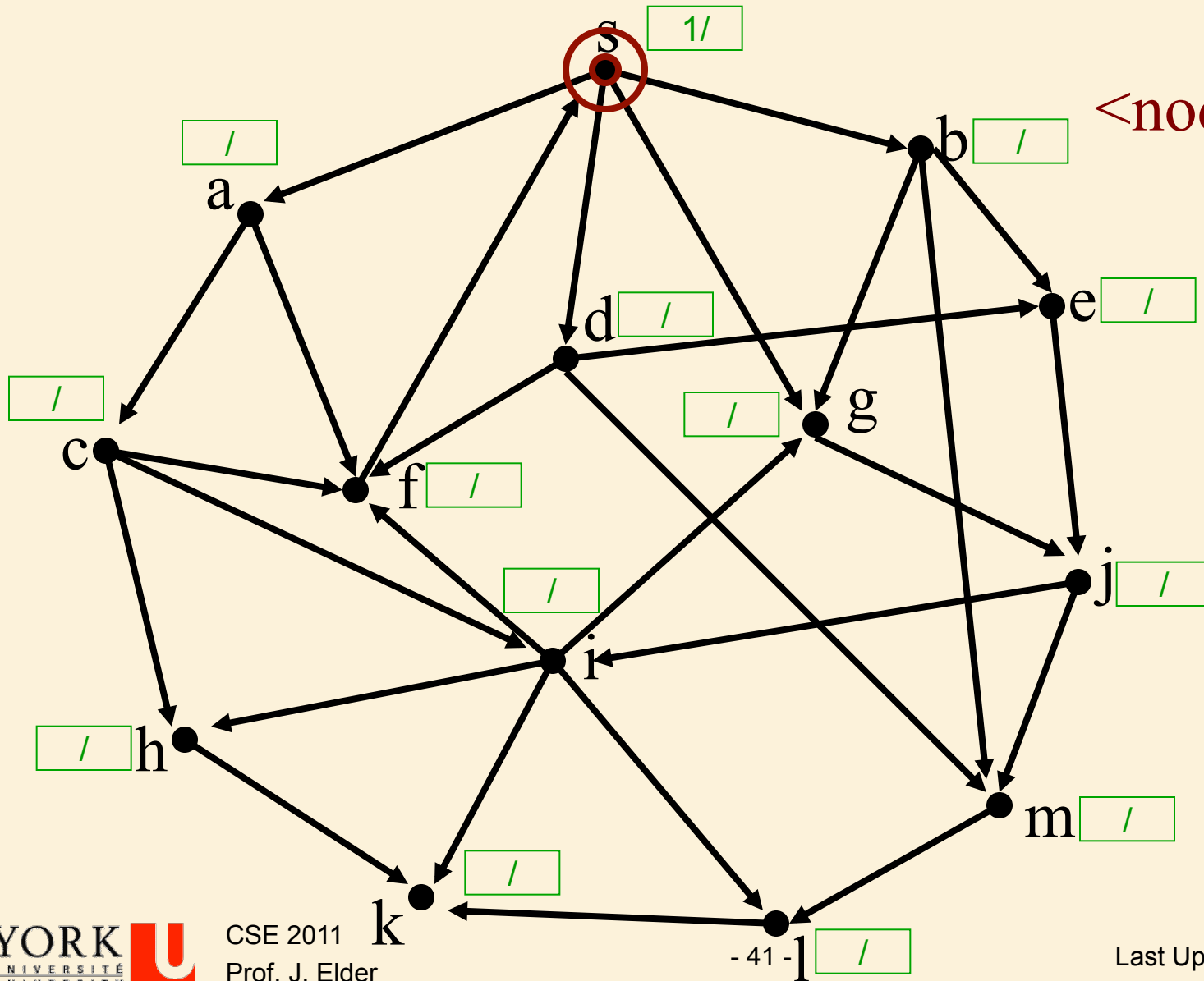
Found
Not Handled
Stack
<node,# edges>



DFS

Found
Not Handled
Stack

<node,# edges>

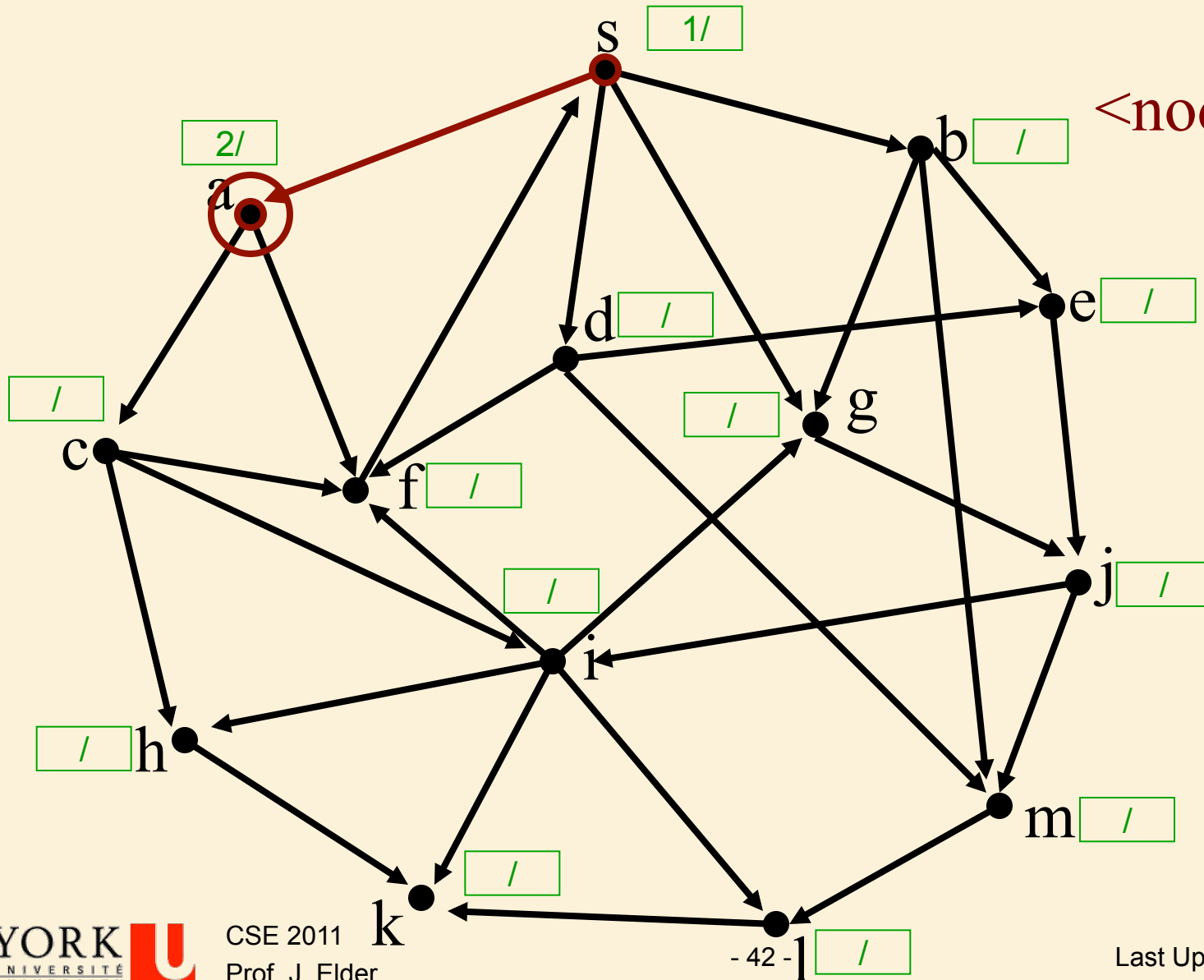


s,0

DFS

Found
Not Handled
Stack

<node,# edges>

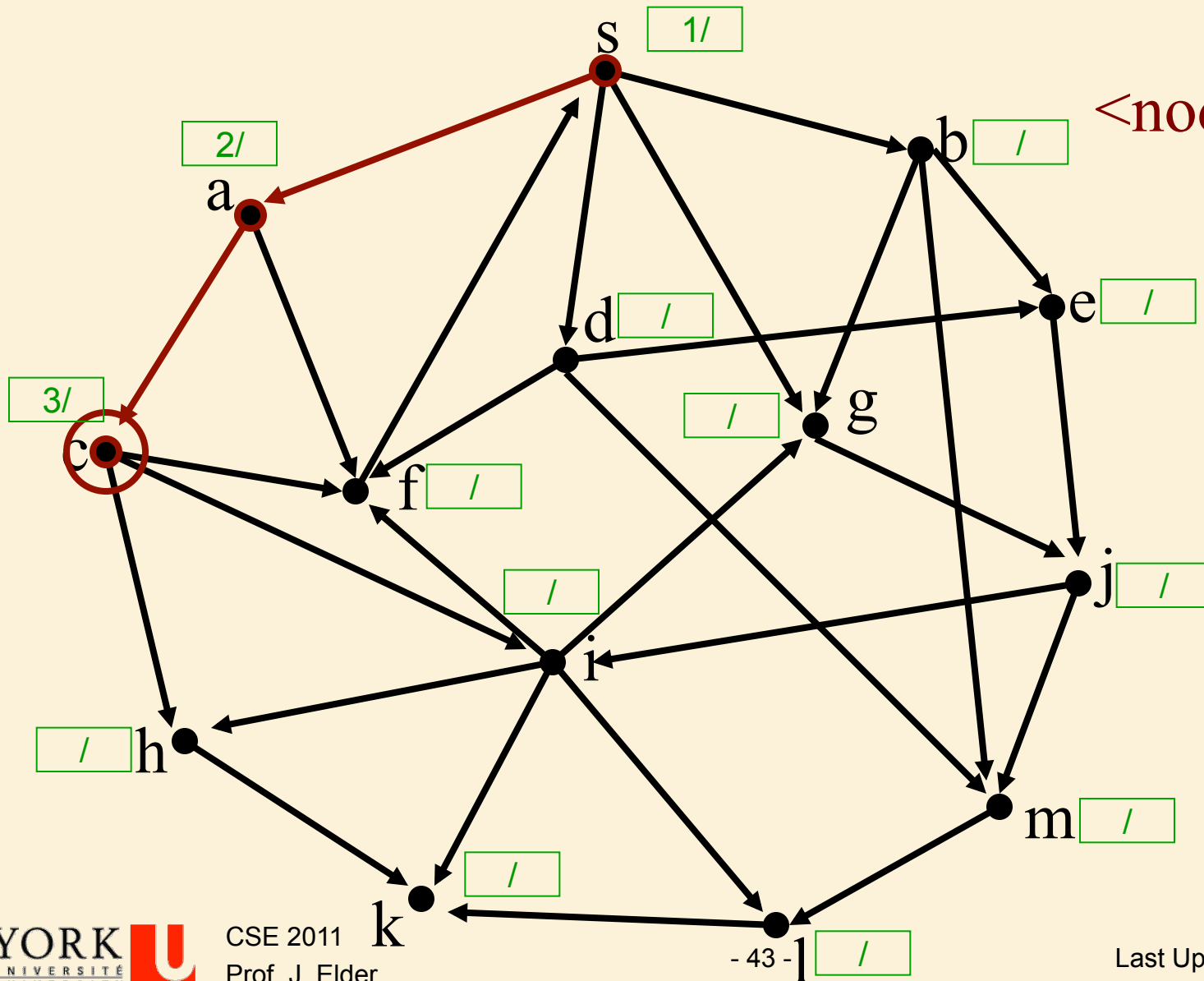


a,0
s,1

DFS

Found
Not Handled
Stack

<node,# edges>

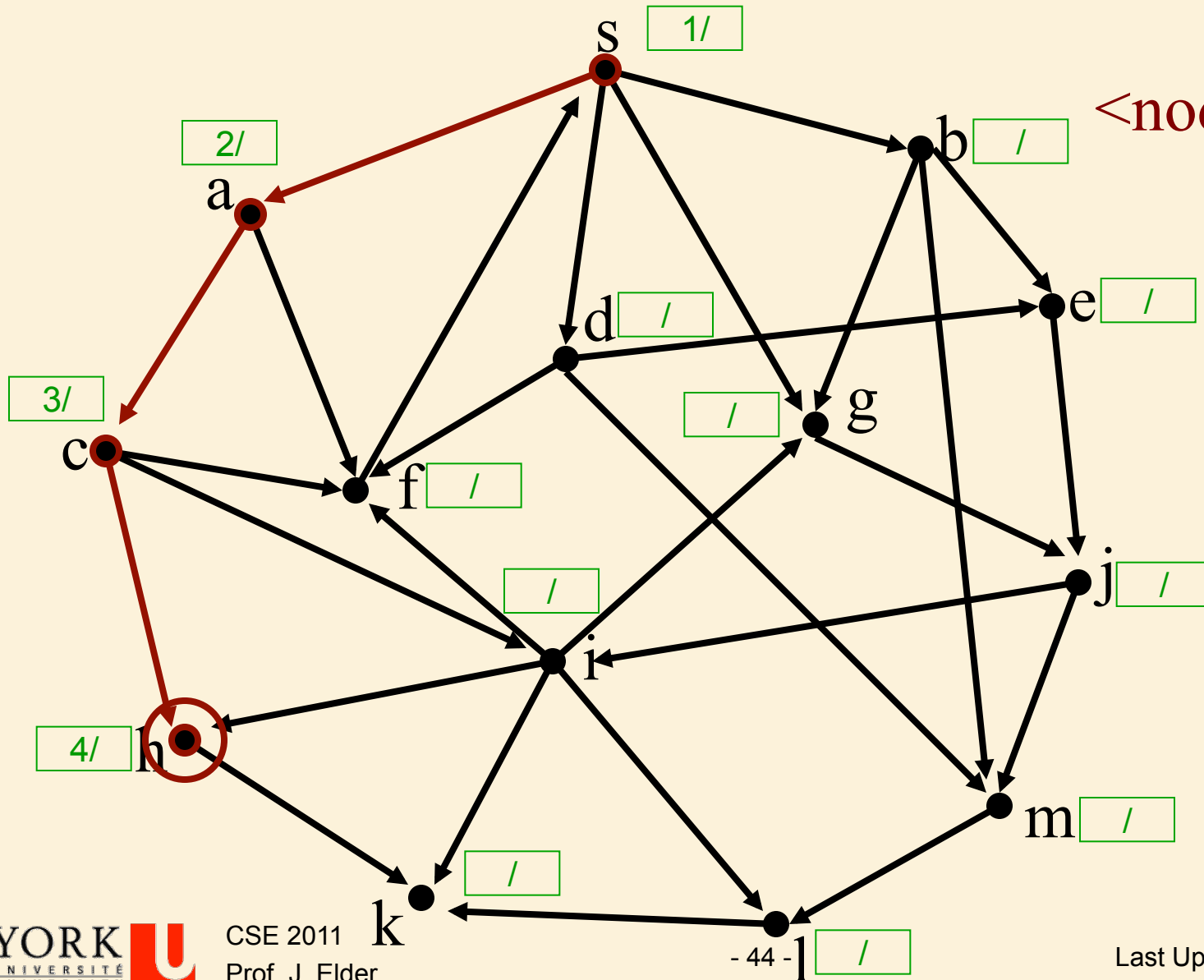


c,0
a,1
s,1

DFS

Found
Not Handled
Stack

<node,# edges>

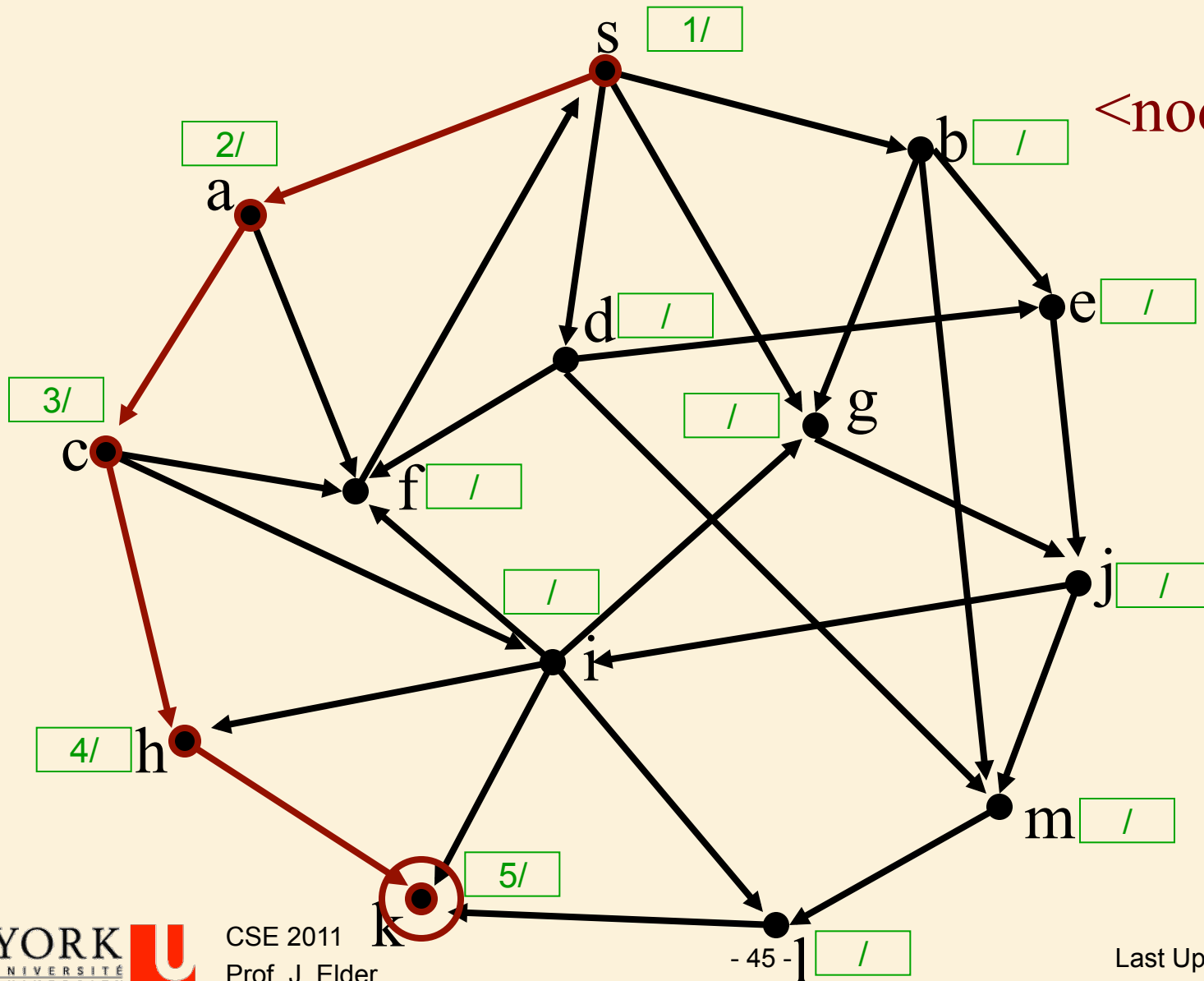


h,0
c,1
a,1
s,1

DFS

Found
Not Handled
Stack

<node,# edges>

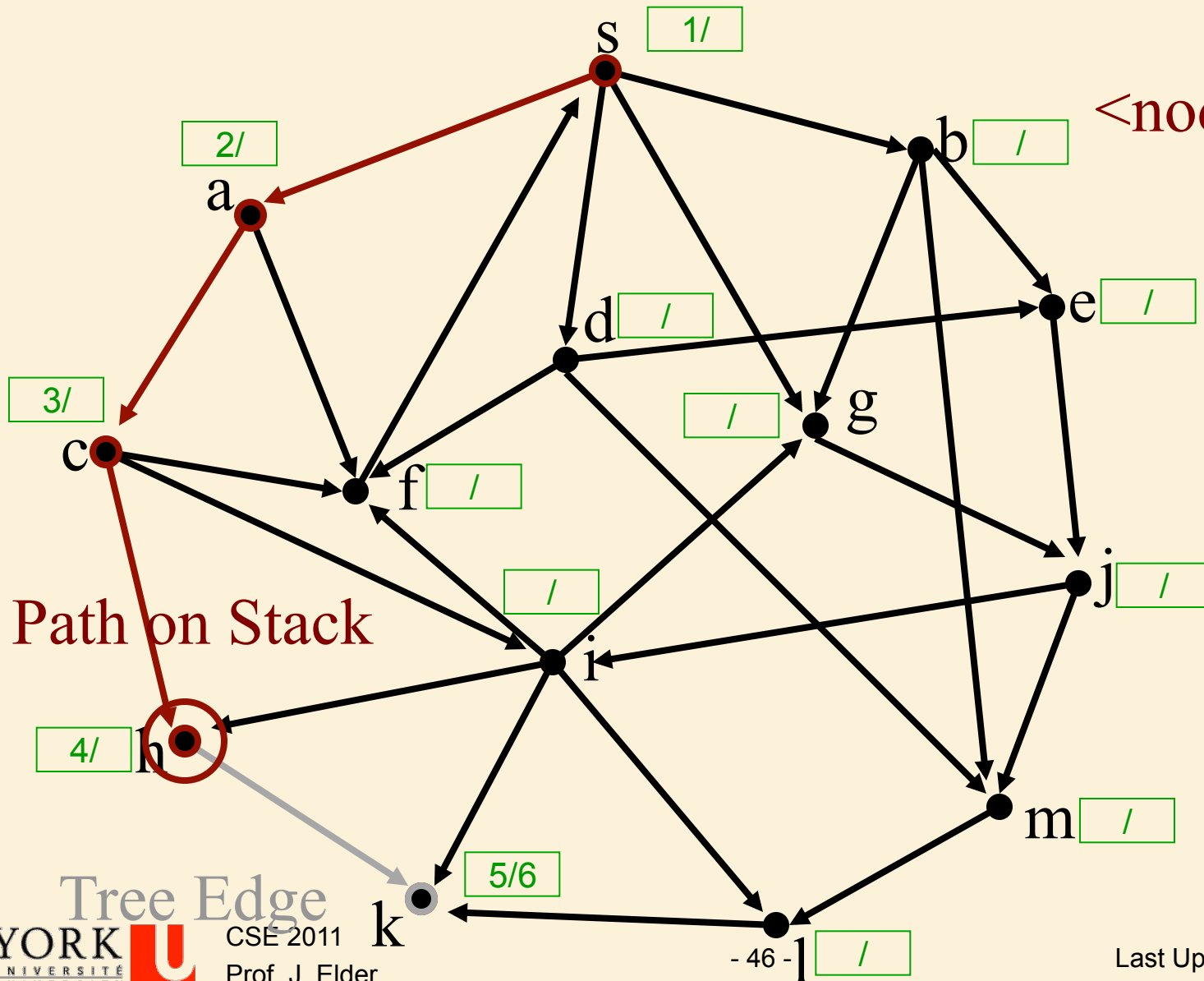


k,0
h,1
c,1
a,1
s,1

DFS

Found
Not Handled
Stack

<node,# edges>

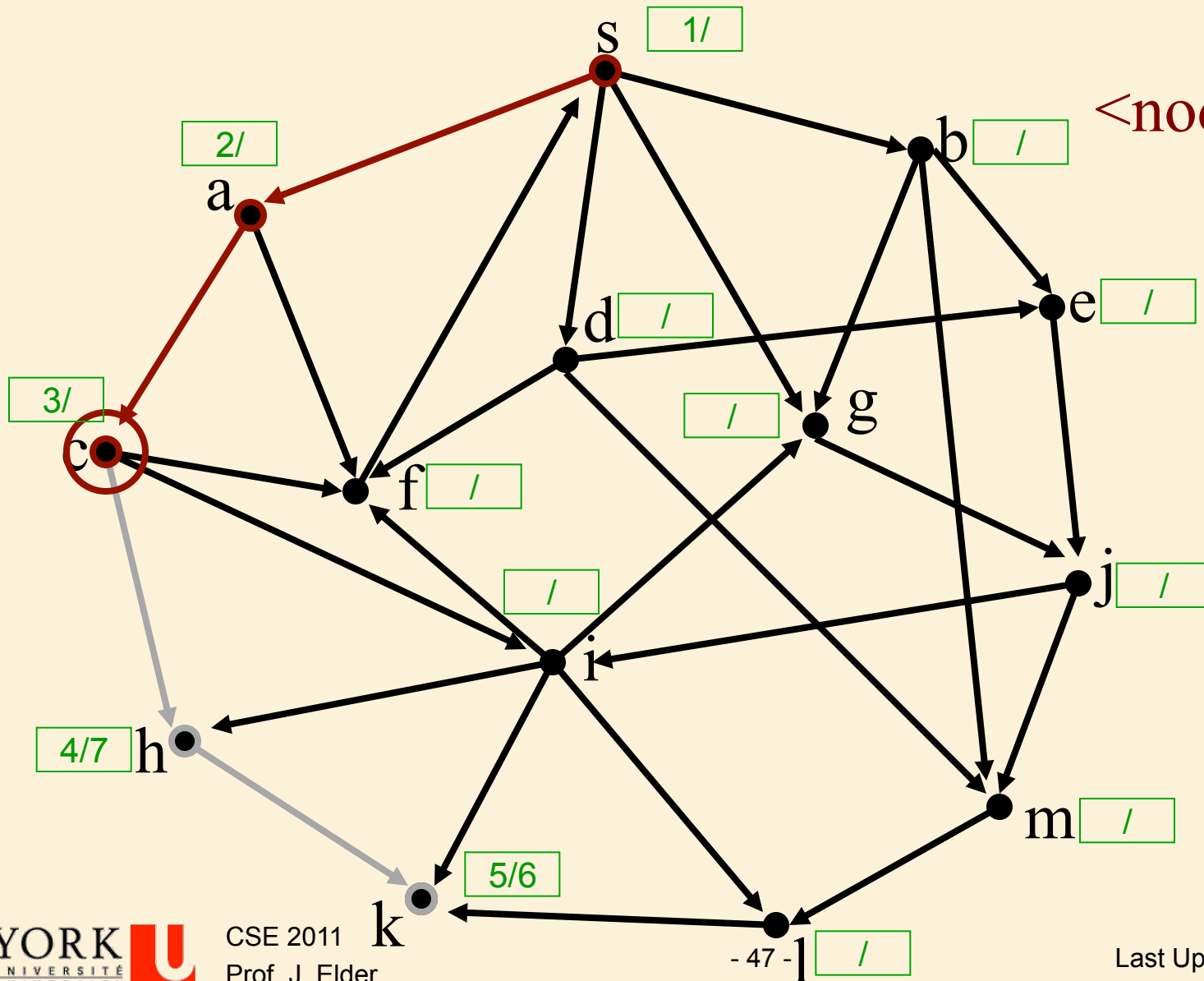


h,1
c,1
a,1
s,1

DFS

Found
Not Handled
Stack

<node,# edges>

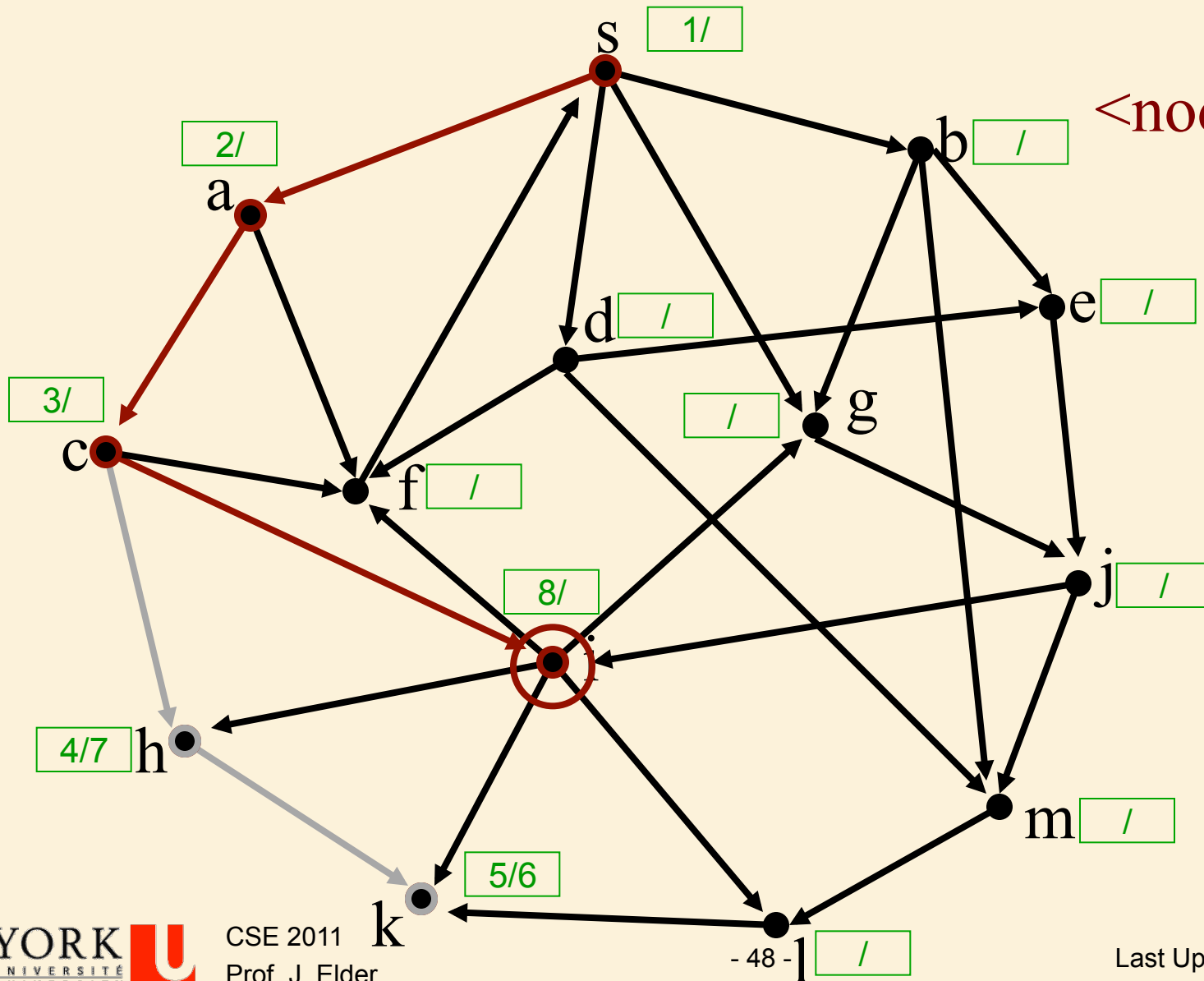


c,1
a,1
s,1

DFS

Found
Not Handled
Stack

<node,# edges>



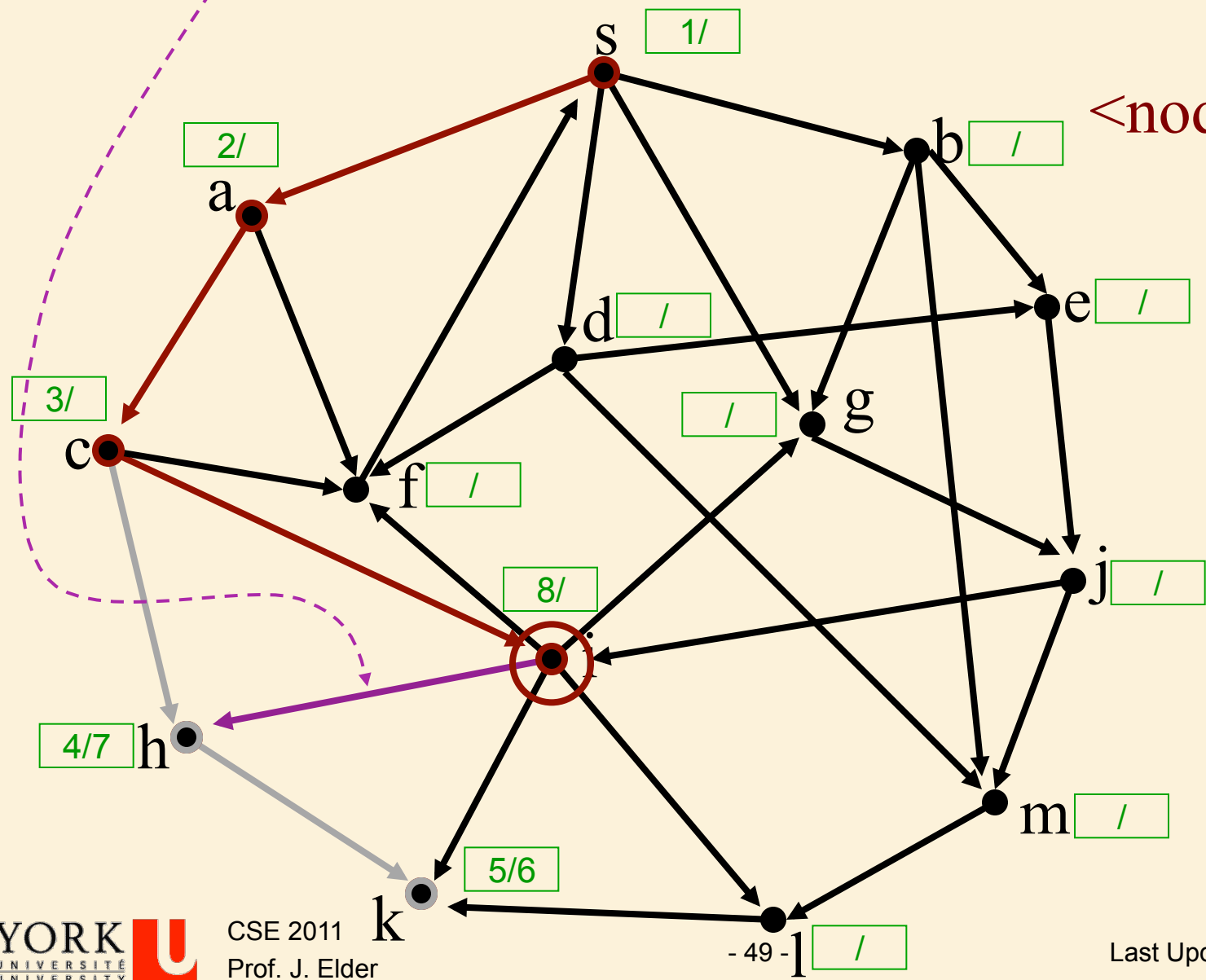
i,0
c,2
a,1
s,1

DFS

Found
Not Handled
Stack

<node,# edges>

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

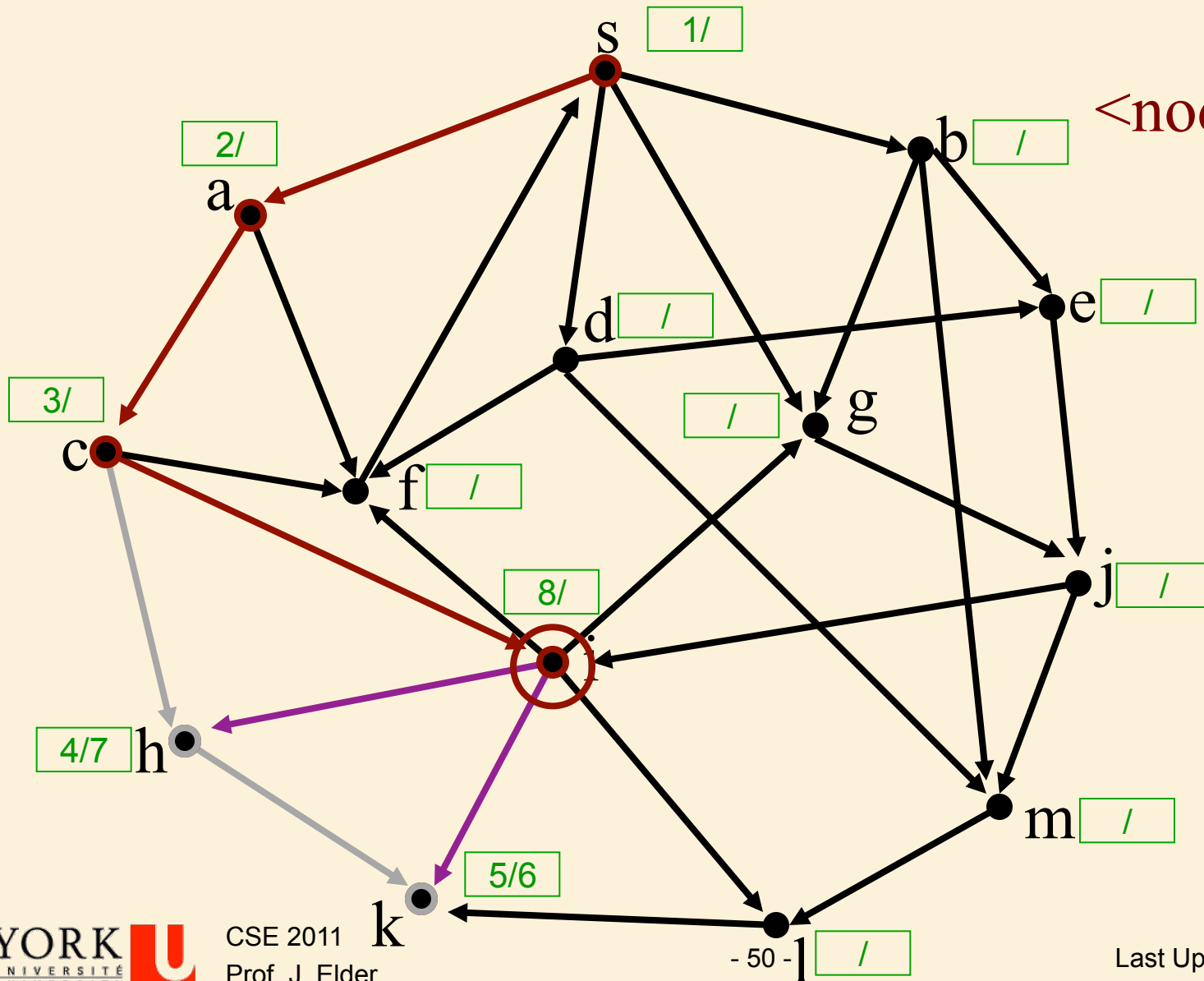


i,1
c,2
a,1
s,1

DFS

Found
Not Handled
Stack

<node,# edges>

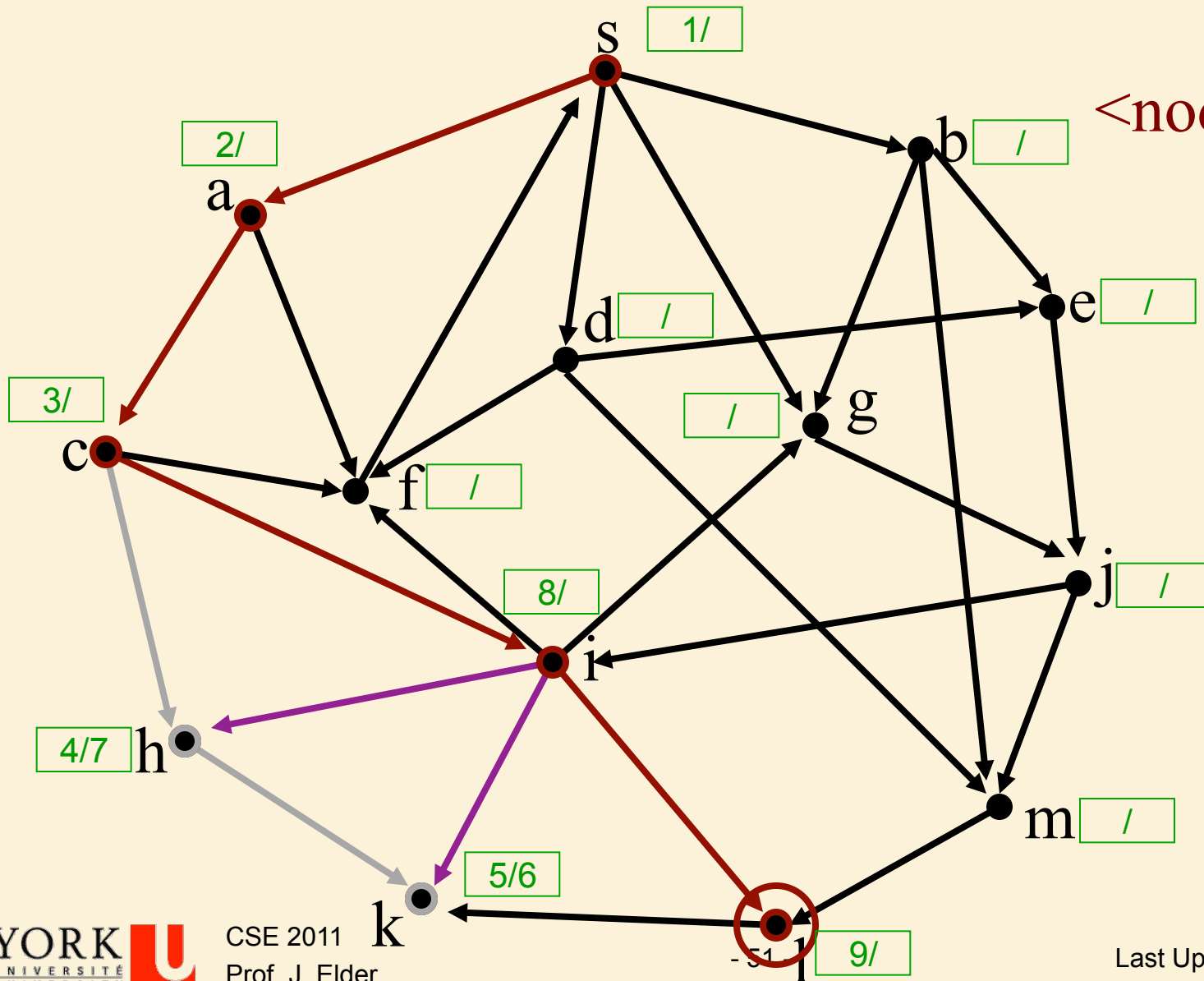


i,2
c,2
a,1
s,1

DFS

Found
Not Handled
Stack

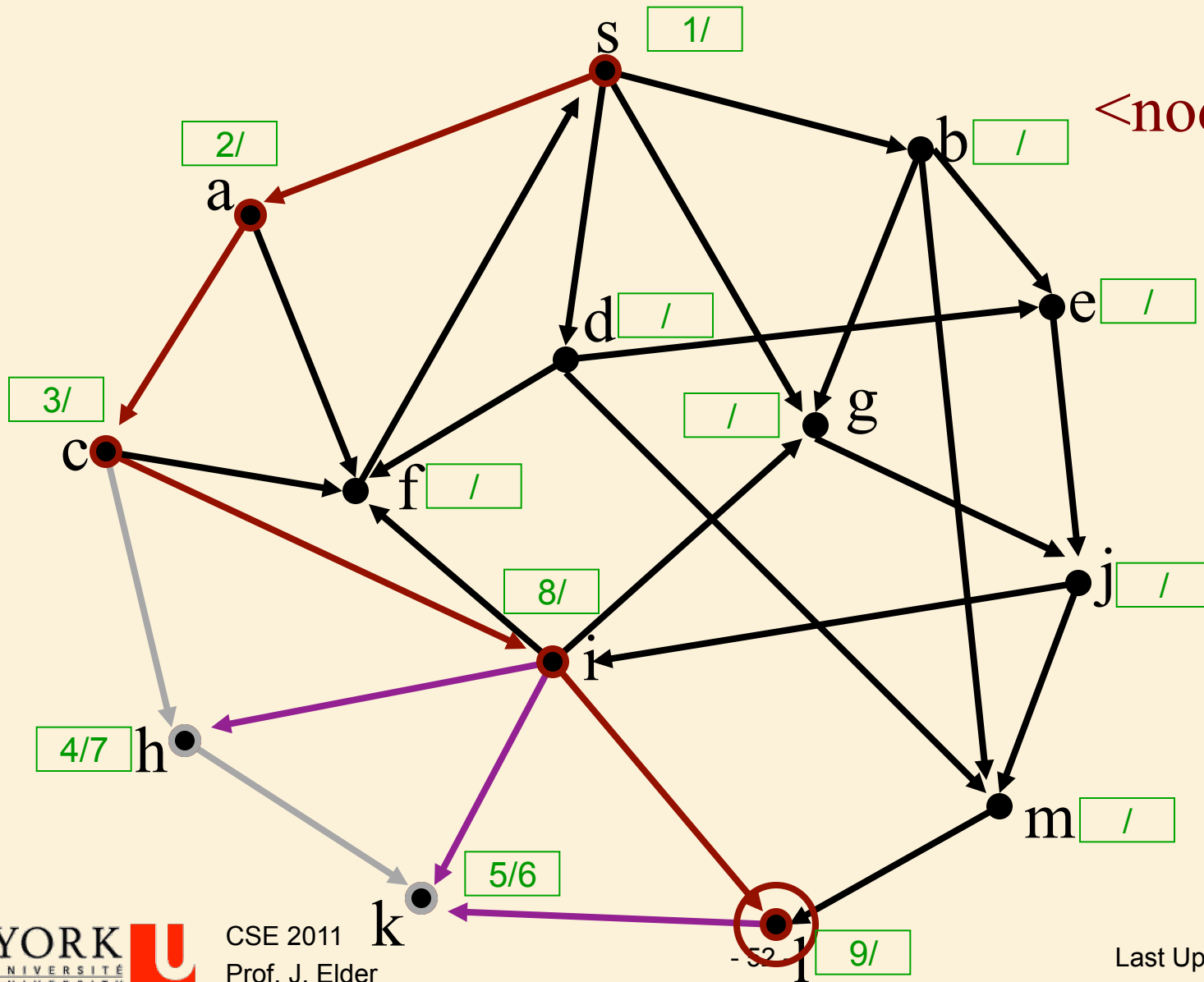
<node,# edges>



DFS

Found
Not Handled
Stack

<node,# edges>



Found Not Handled Stack

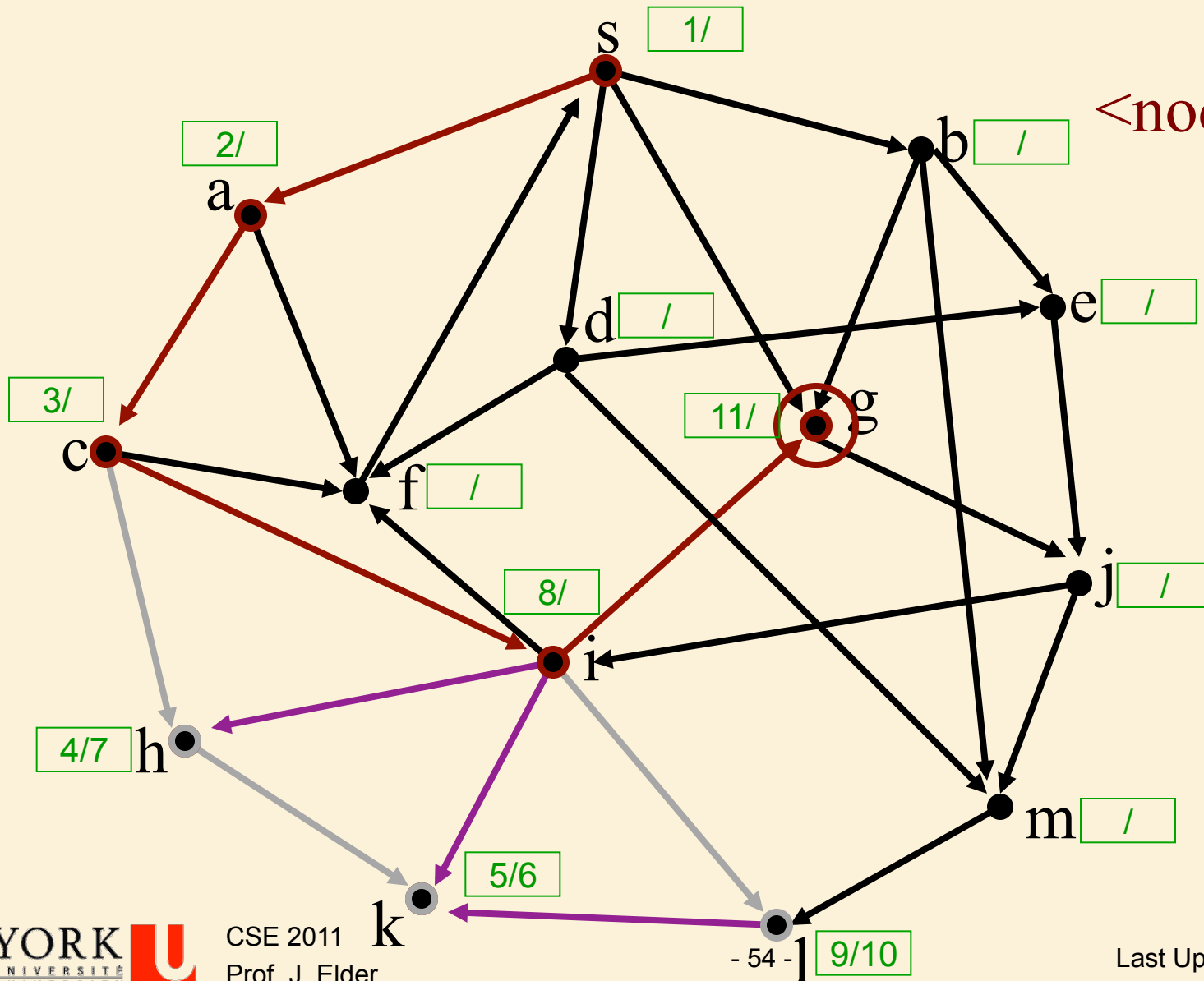
i,3
c,2
a,1
s,1



DFS

Found
Not Handled
Stack

<node,# edges>



Found Not Handled Stack

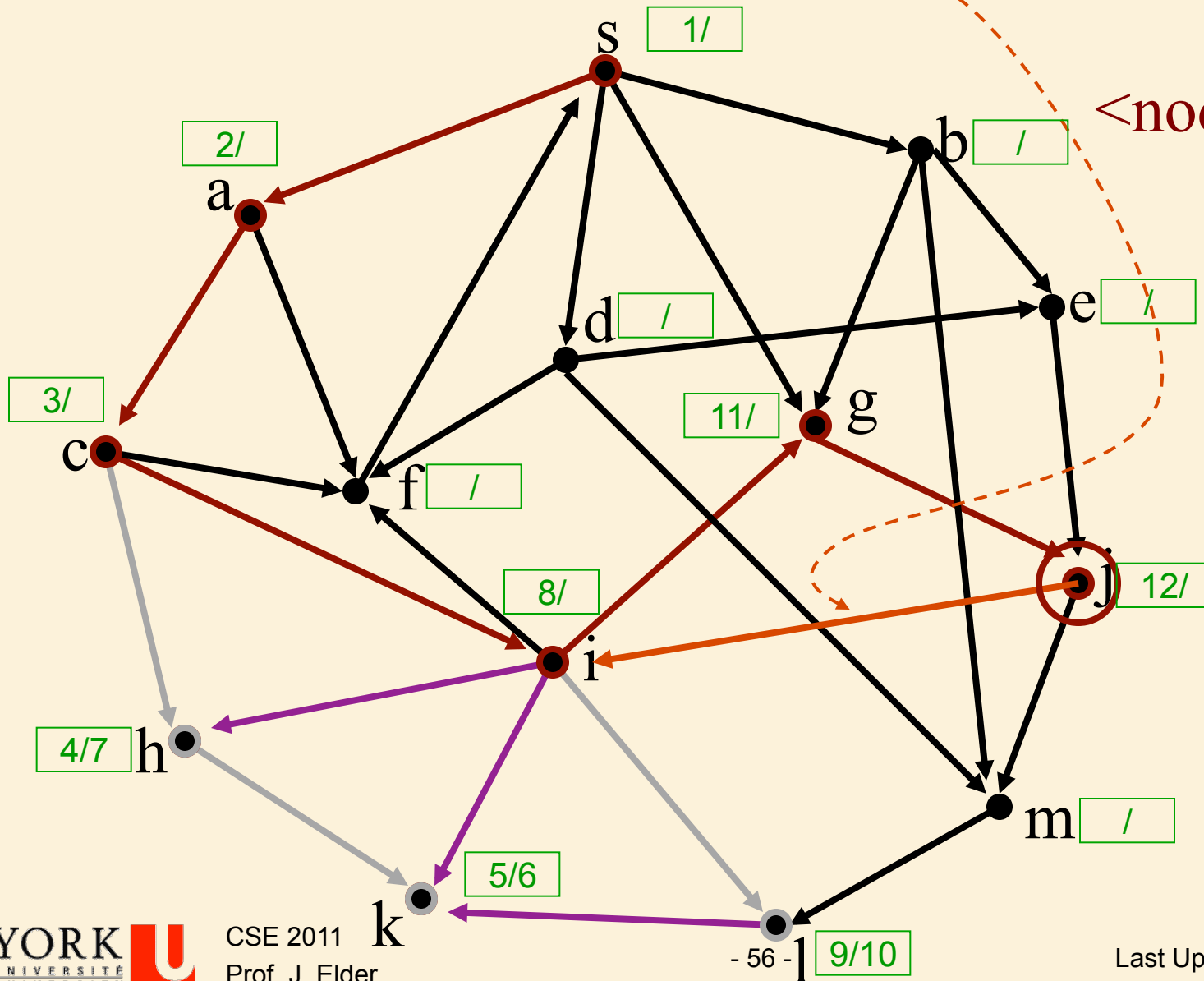
j,0
g,1
i,4
c,2
a,1
s,1



DFS

Found
Not Handled
Stack

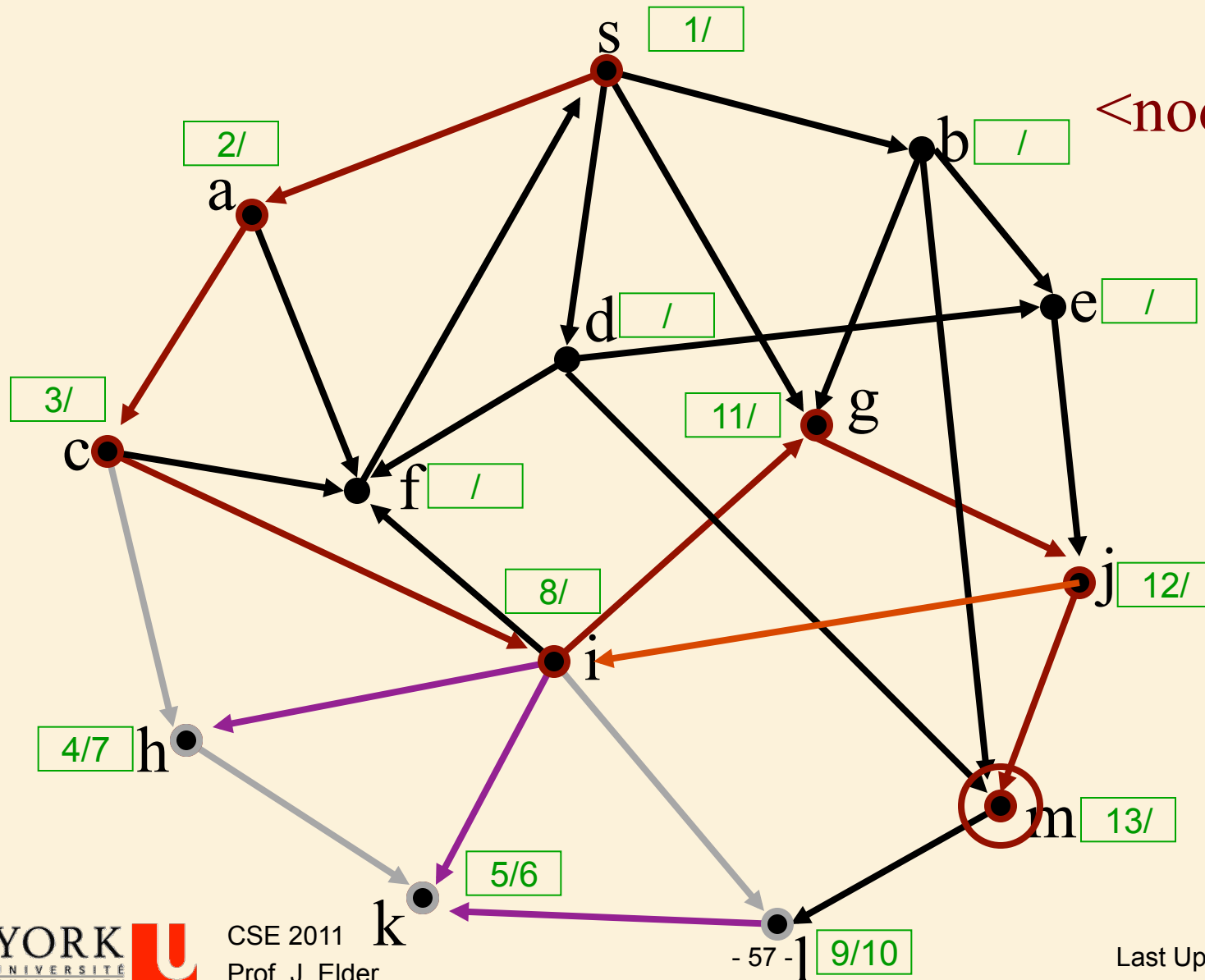
Back Edge to node on Stack:



DFS

Found
Not Handled
Stack

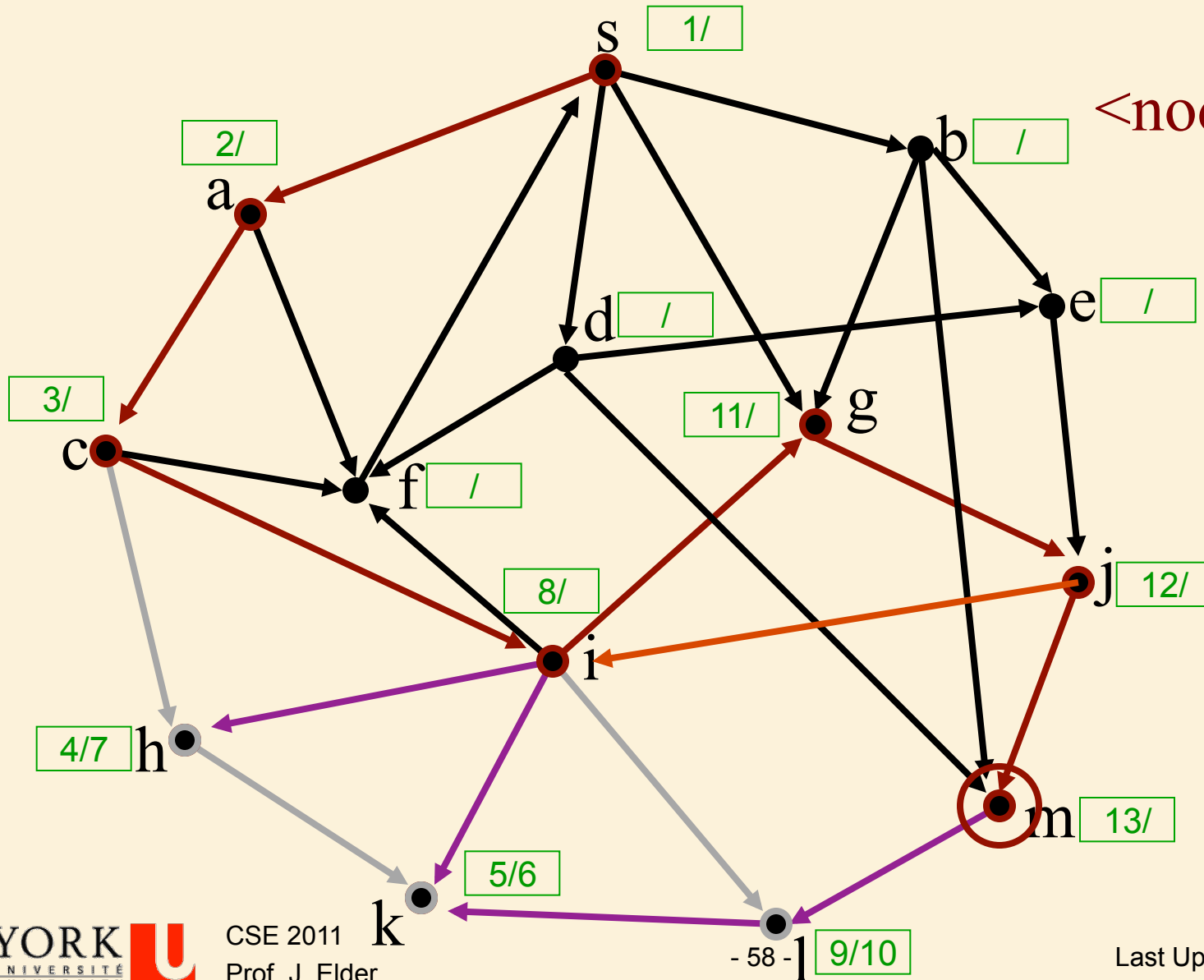
<node,# edges>



DFS

Found
Not Handled
Stack

<node,# edges>

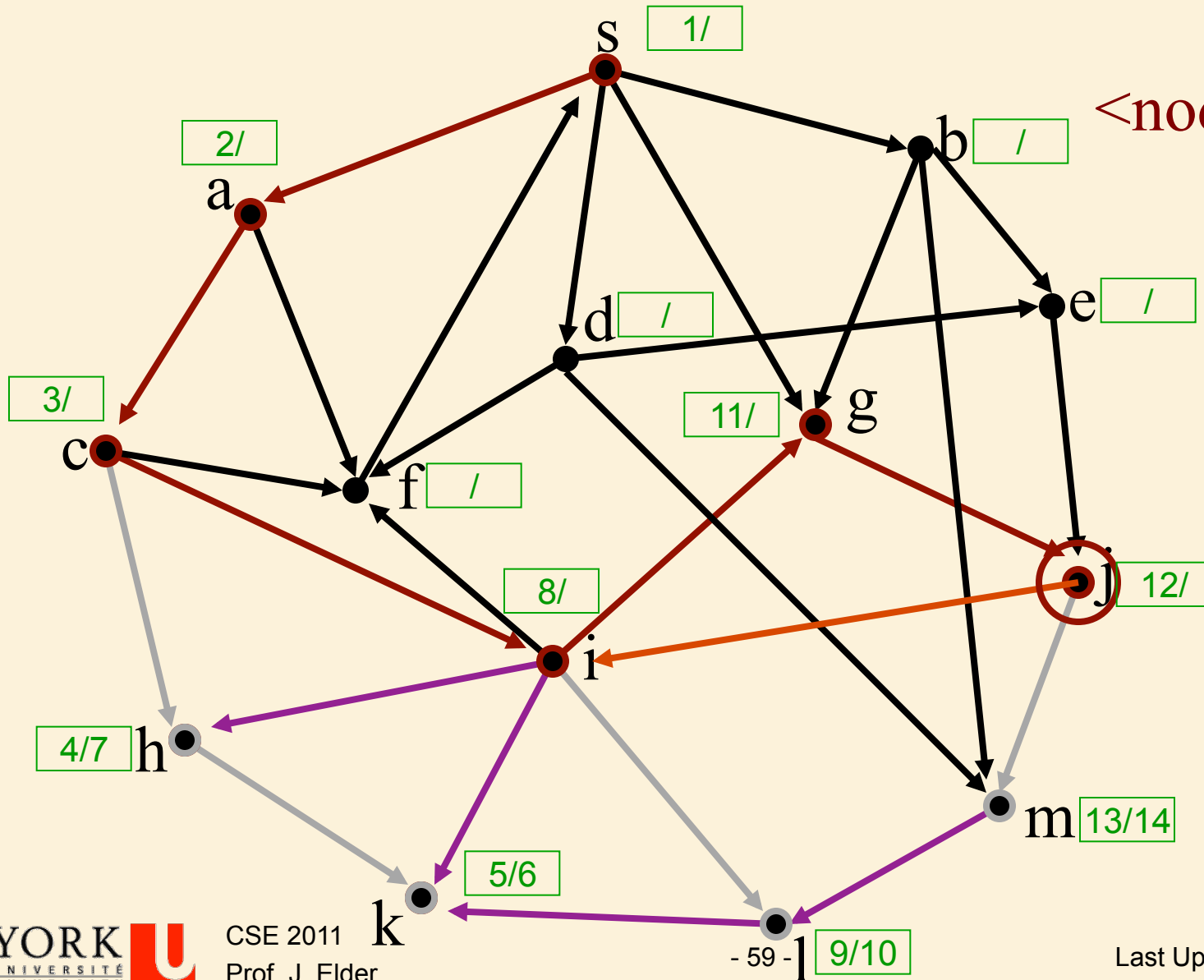


m,1
j,2
g,1
i,4
c,2
a,1
s,1

DFS

Found
Not Handled
Stack

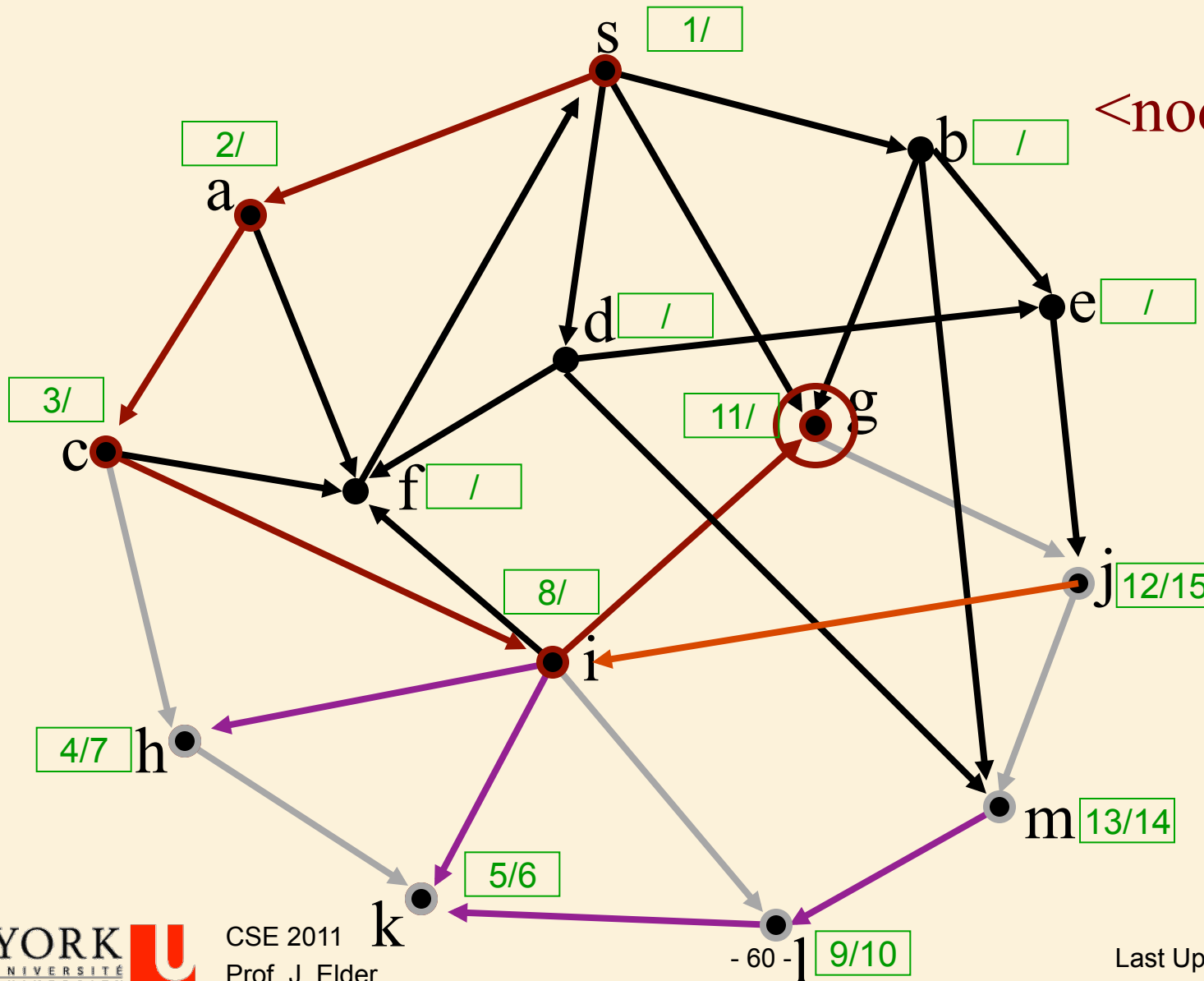
<node,# edges>



DFS

Found
Not Handled
Stack

<node,# edges>

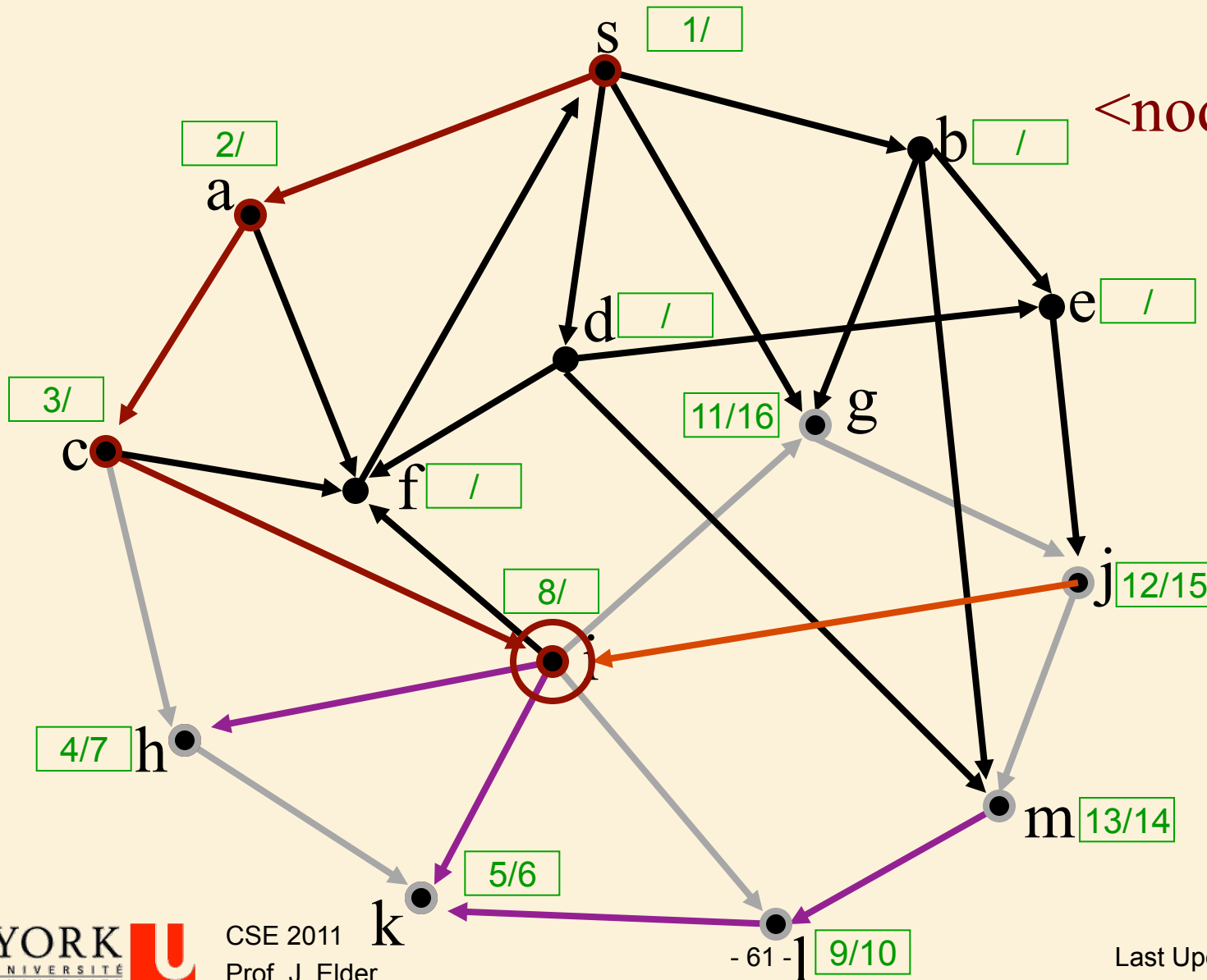


g,1
i,4
c,2
a,1
s,1

DFS

Found
Not Handled
Stack

<node,# edges>

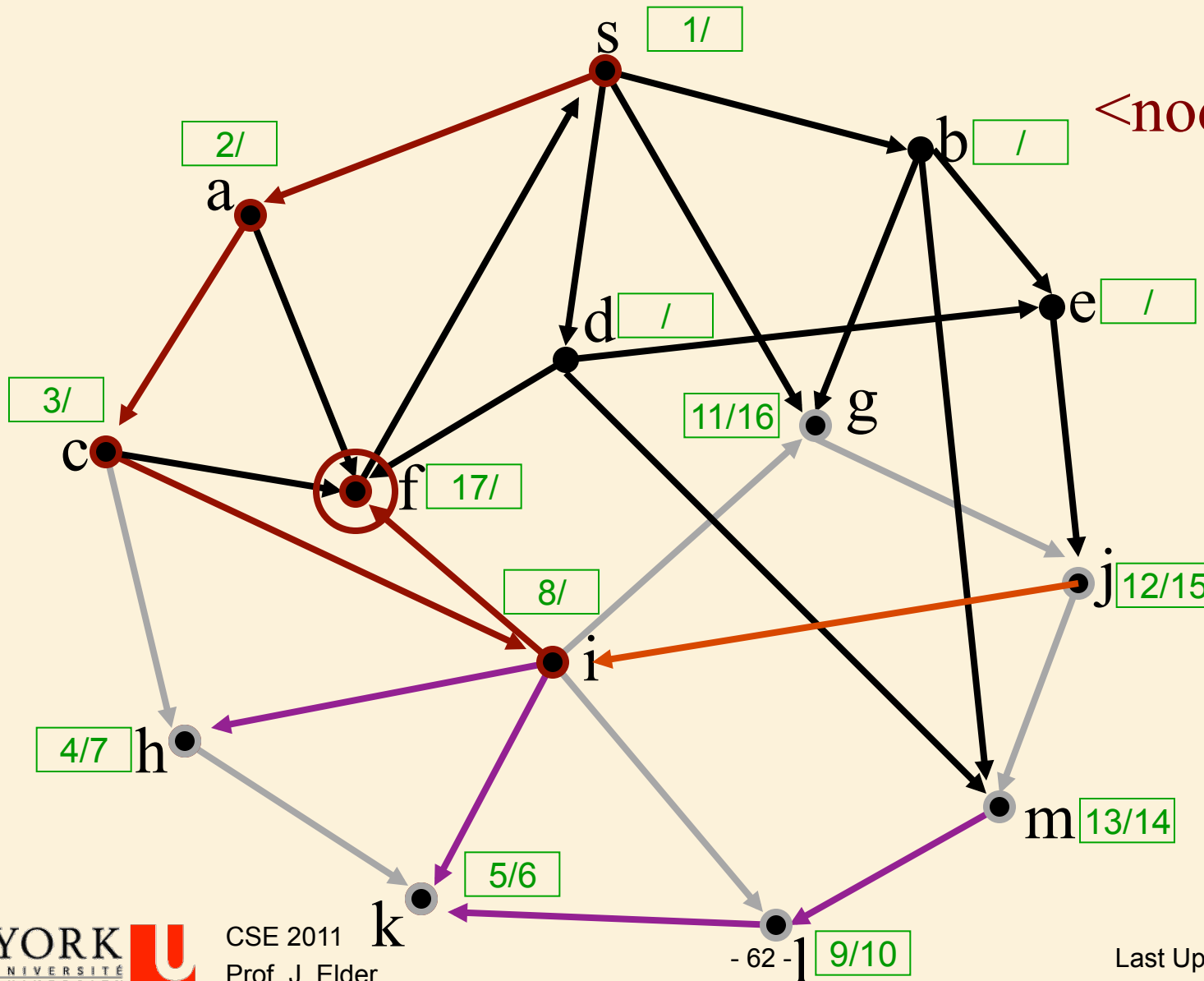


i,4
c,2
a,1
s,1

DFS

Found
Not Handled
Stack

<node,# edges>

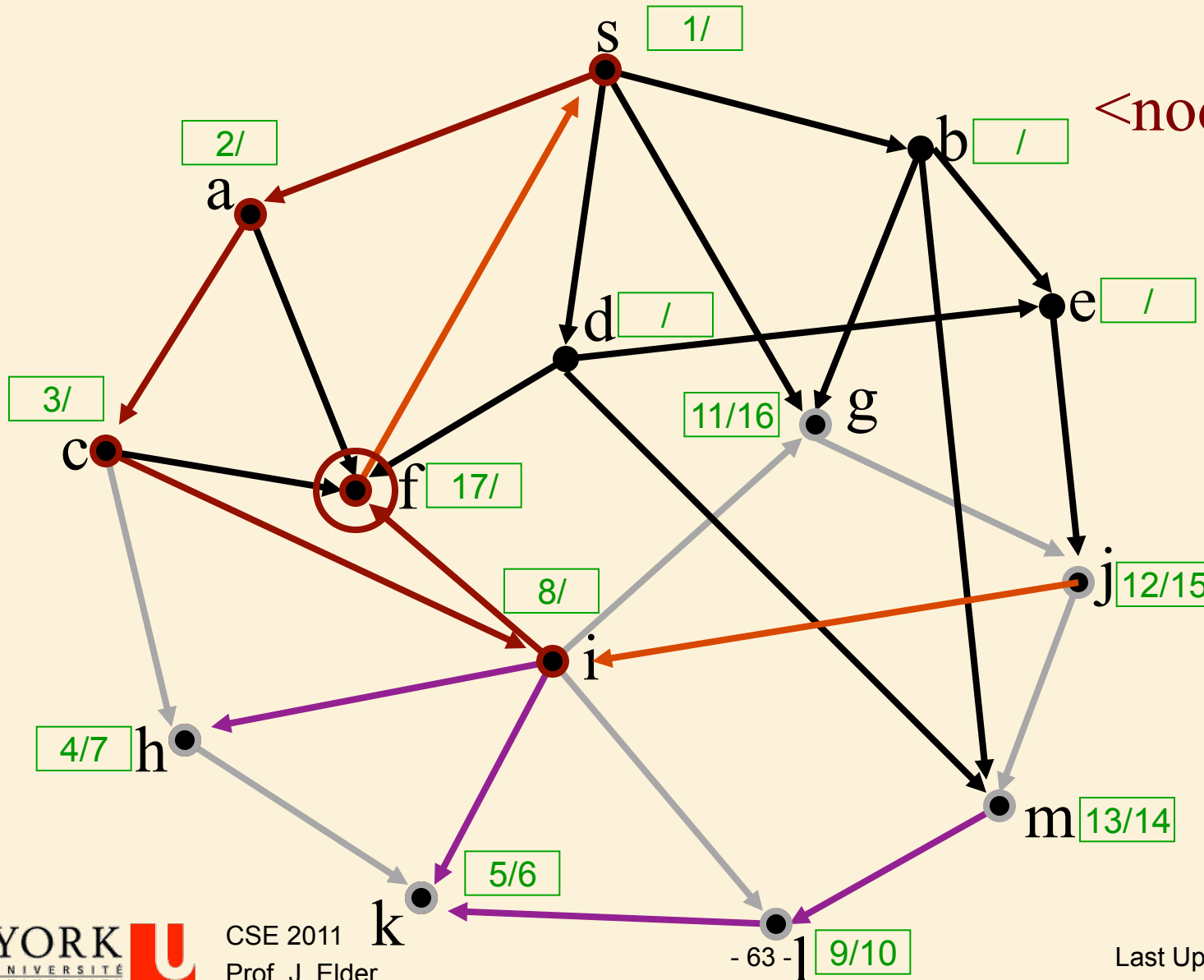


f,0
i,5
c,2
a,1
s,1

DFS

Found
Not Handled
Stack

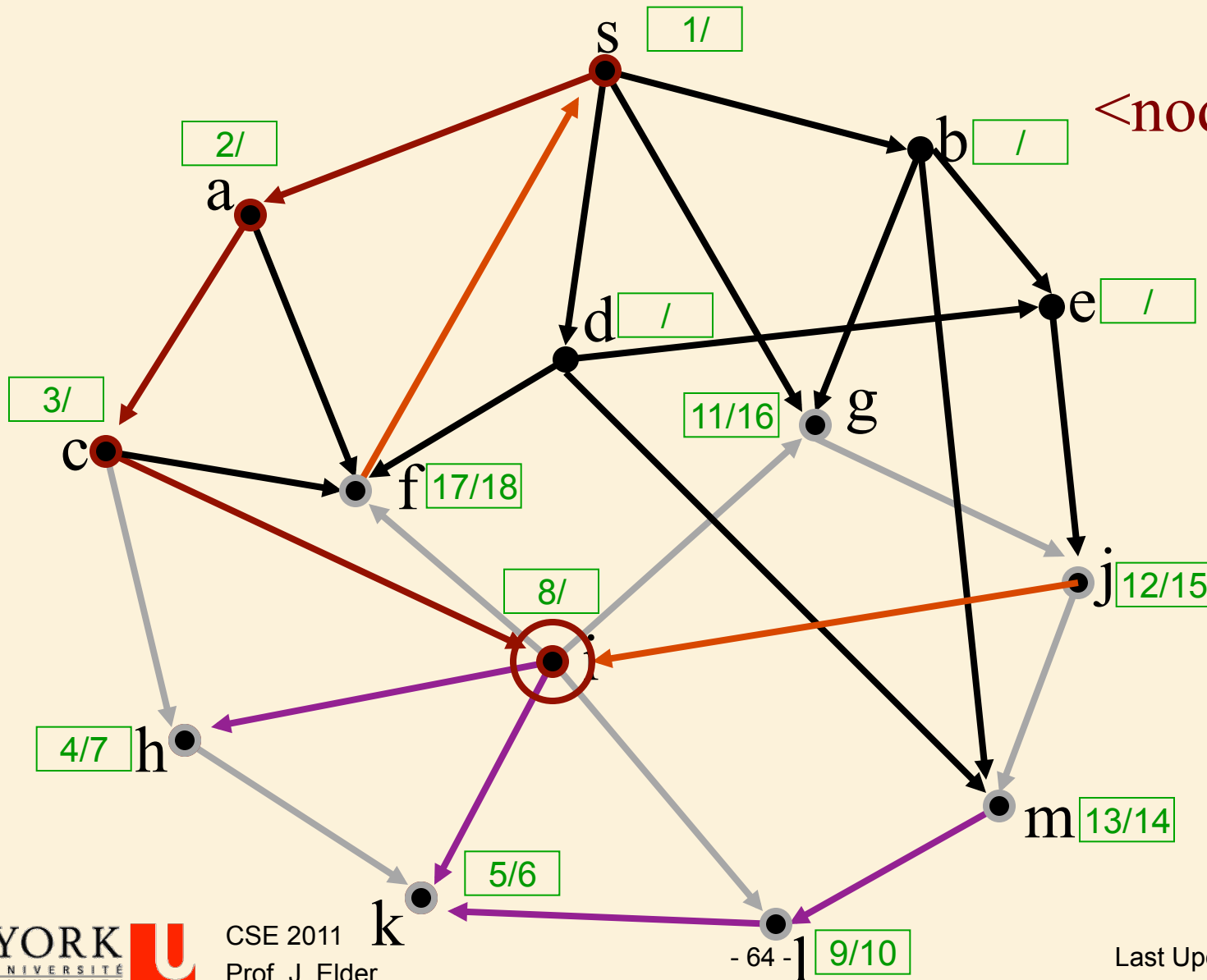
<node,# edges>



DFS

Found
Not Handled
Stack

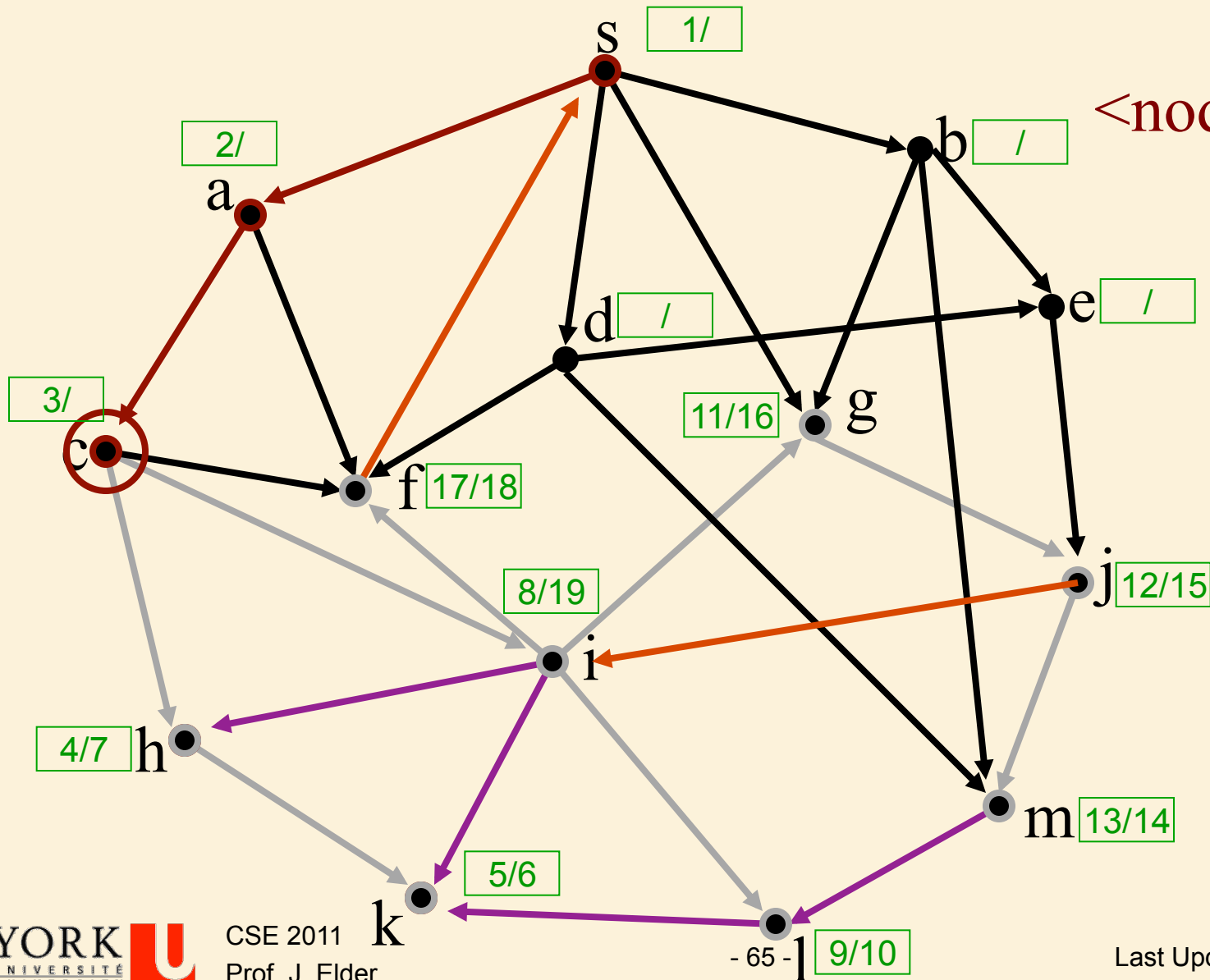
<node,# edges>



DFS

Found
Not Handled
Stack

<node,# edges>

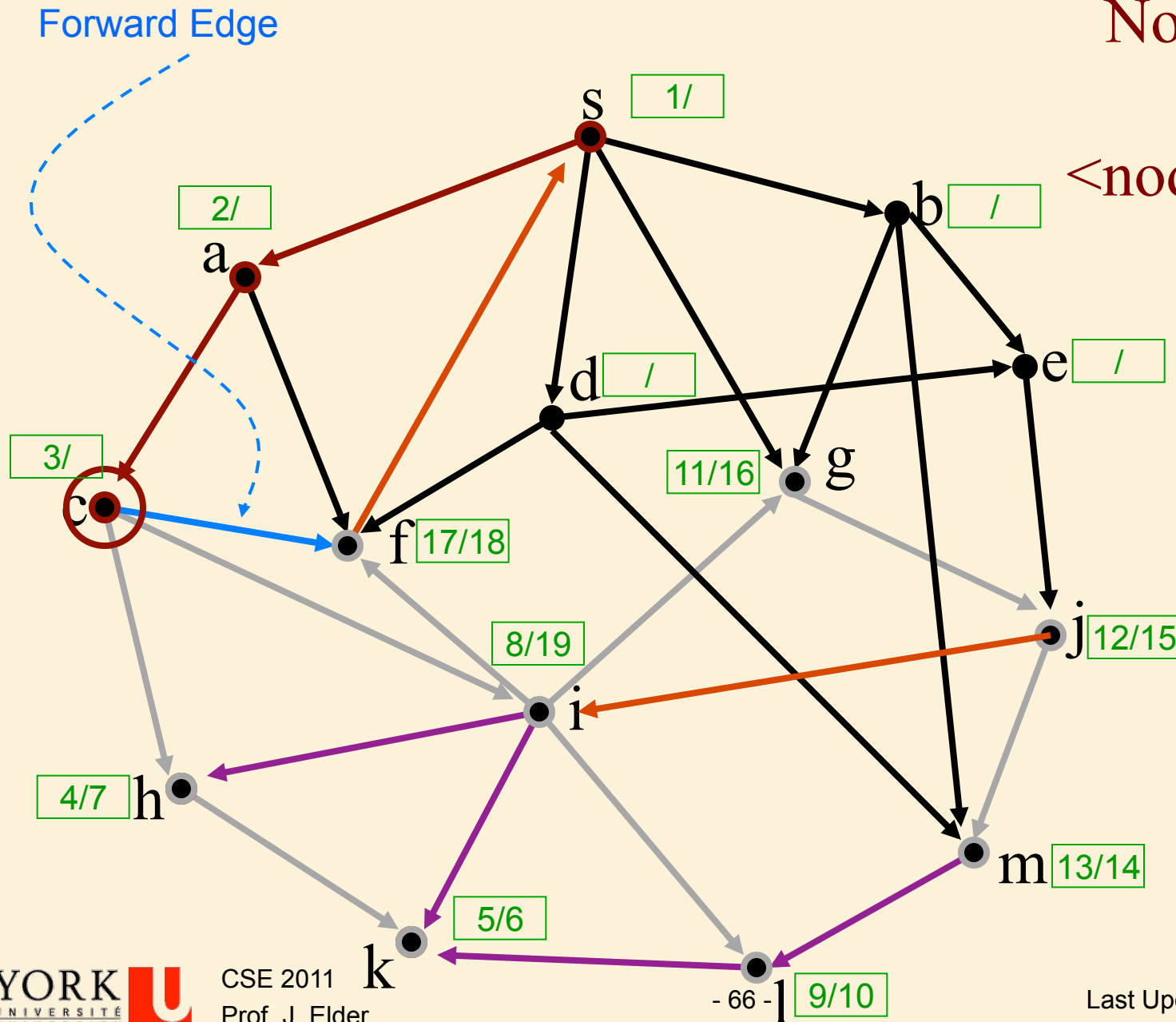


c,2
a,1
s,1

DFS

Found
Not Handled
Stack

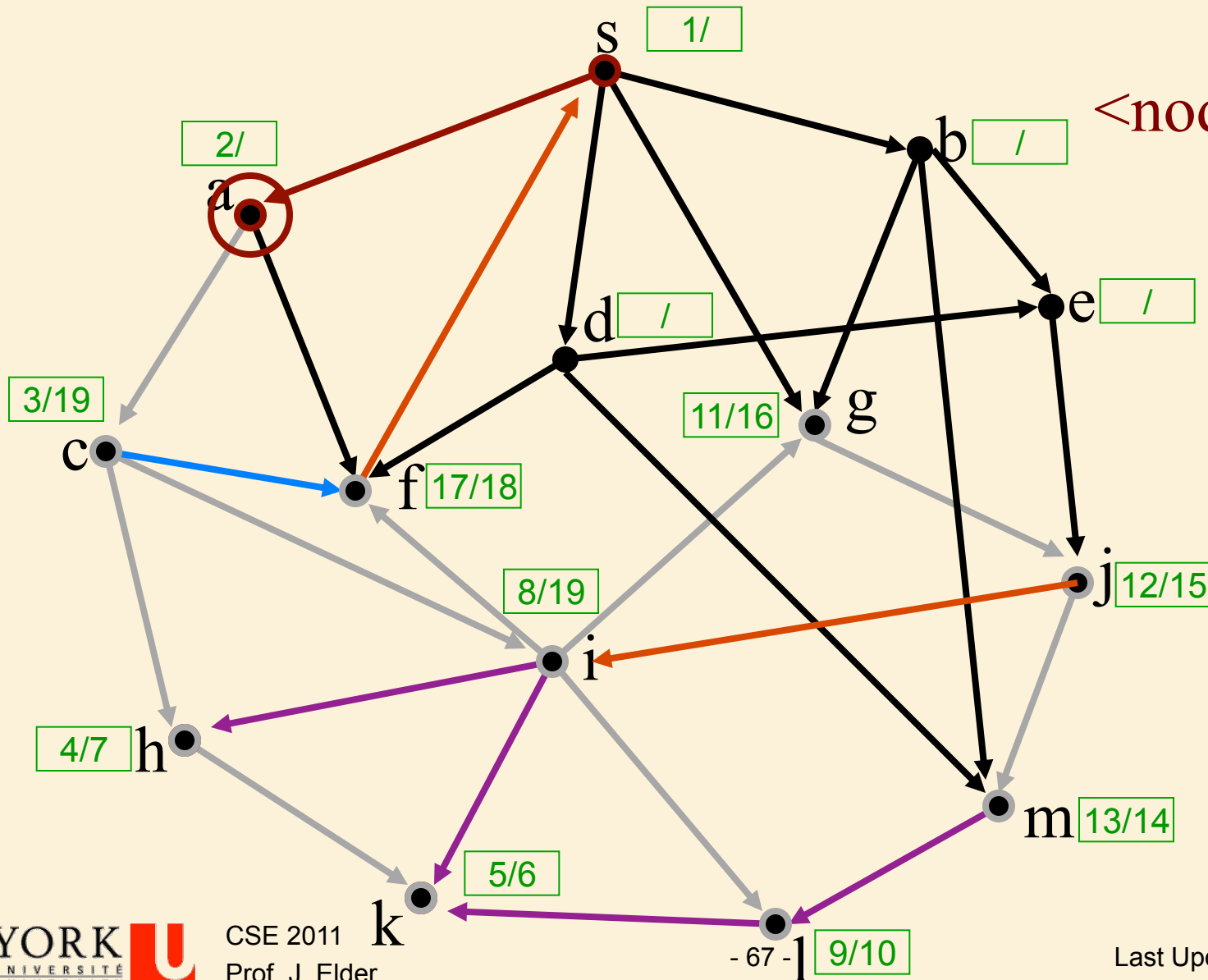
<node,# edges>



DFS

Found
Not Handled
Stack

<node,# edges>



a,1
s,1

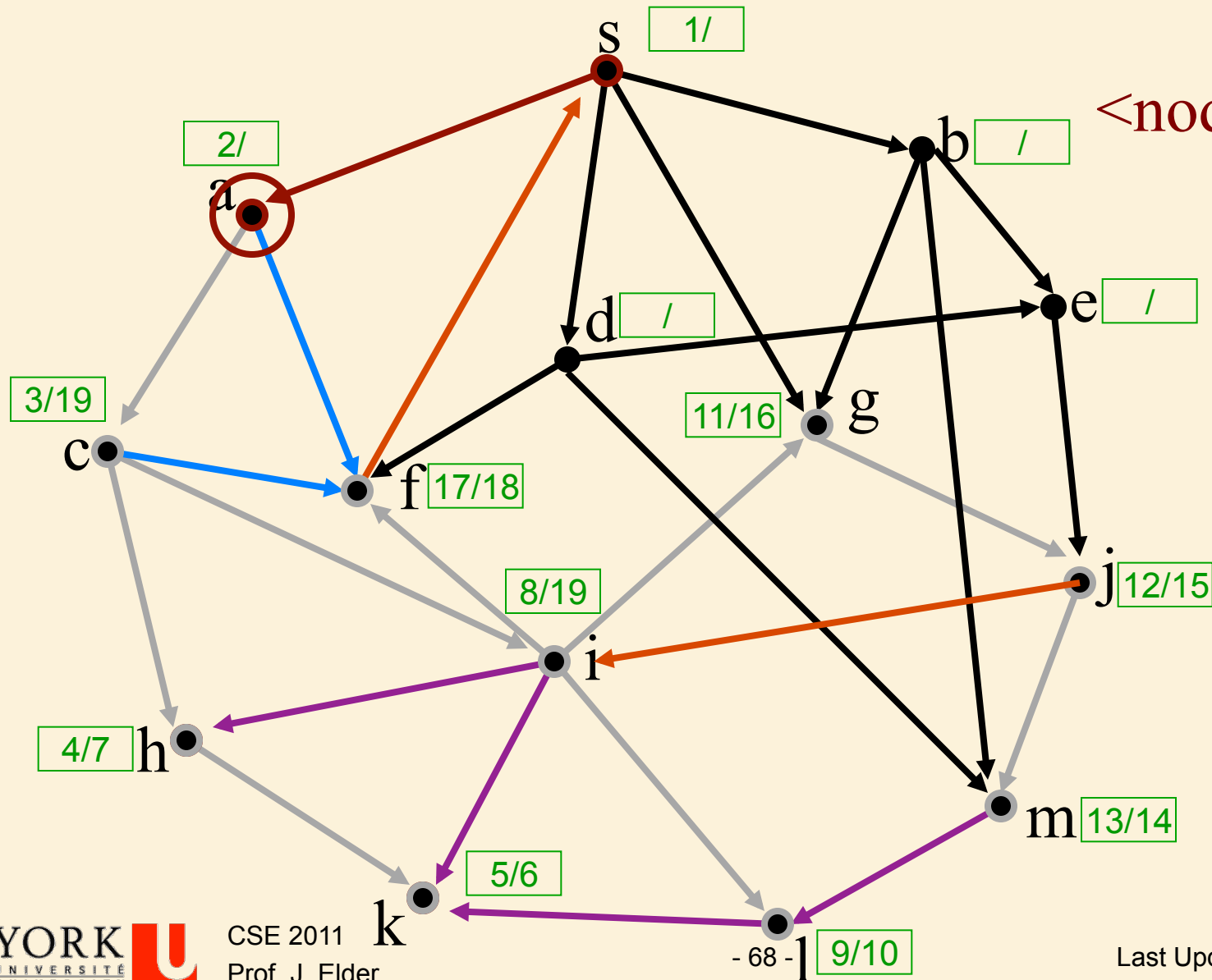
- 67 - 1 9/10

Last Updated: 12-03-22 10:12 AM

DFS

Found
Not Handled
Stack

<node,# edges>

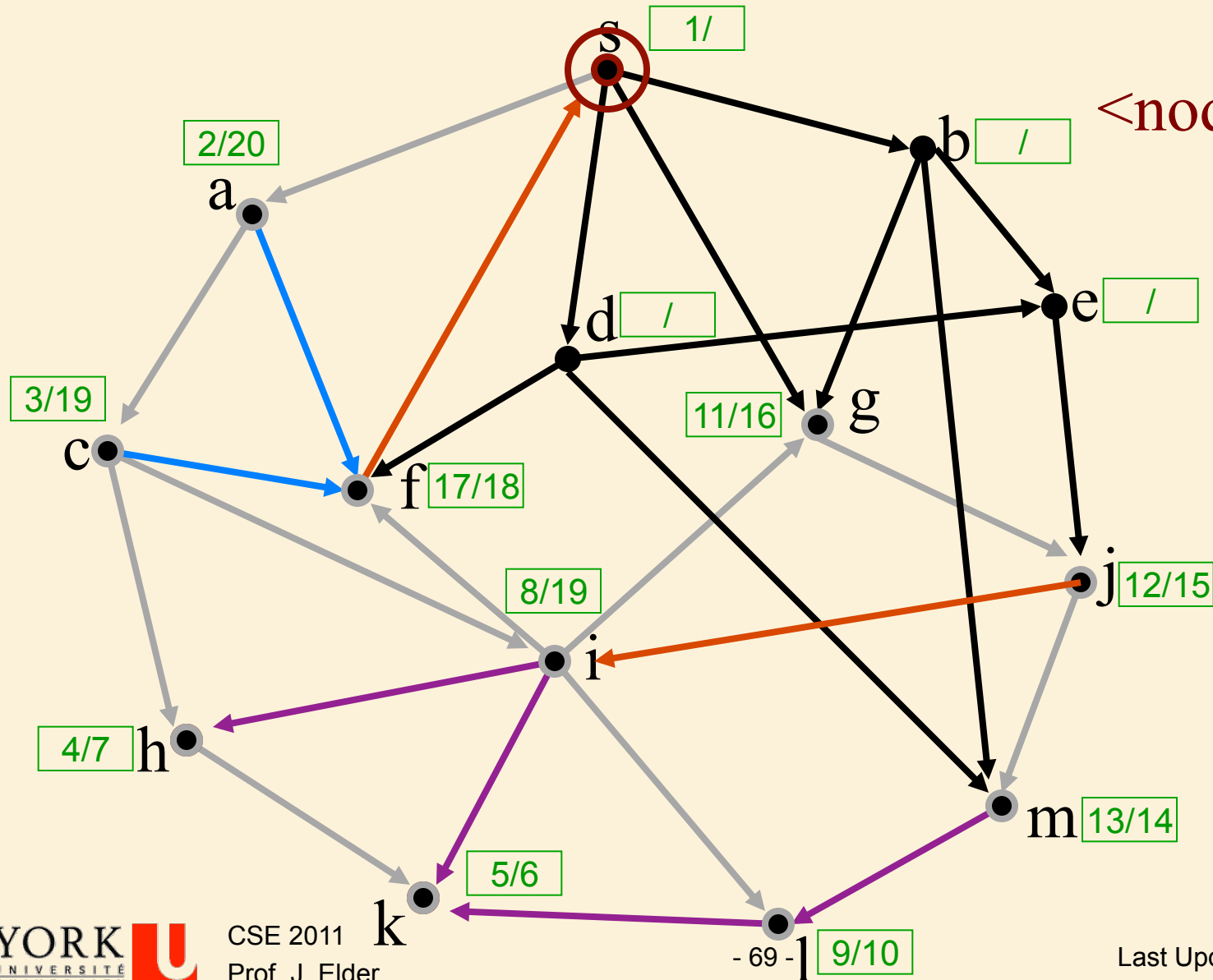


a,2
s,1

DFS

Found
Not Handled
Stack

<node,# edges>

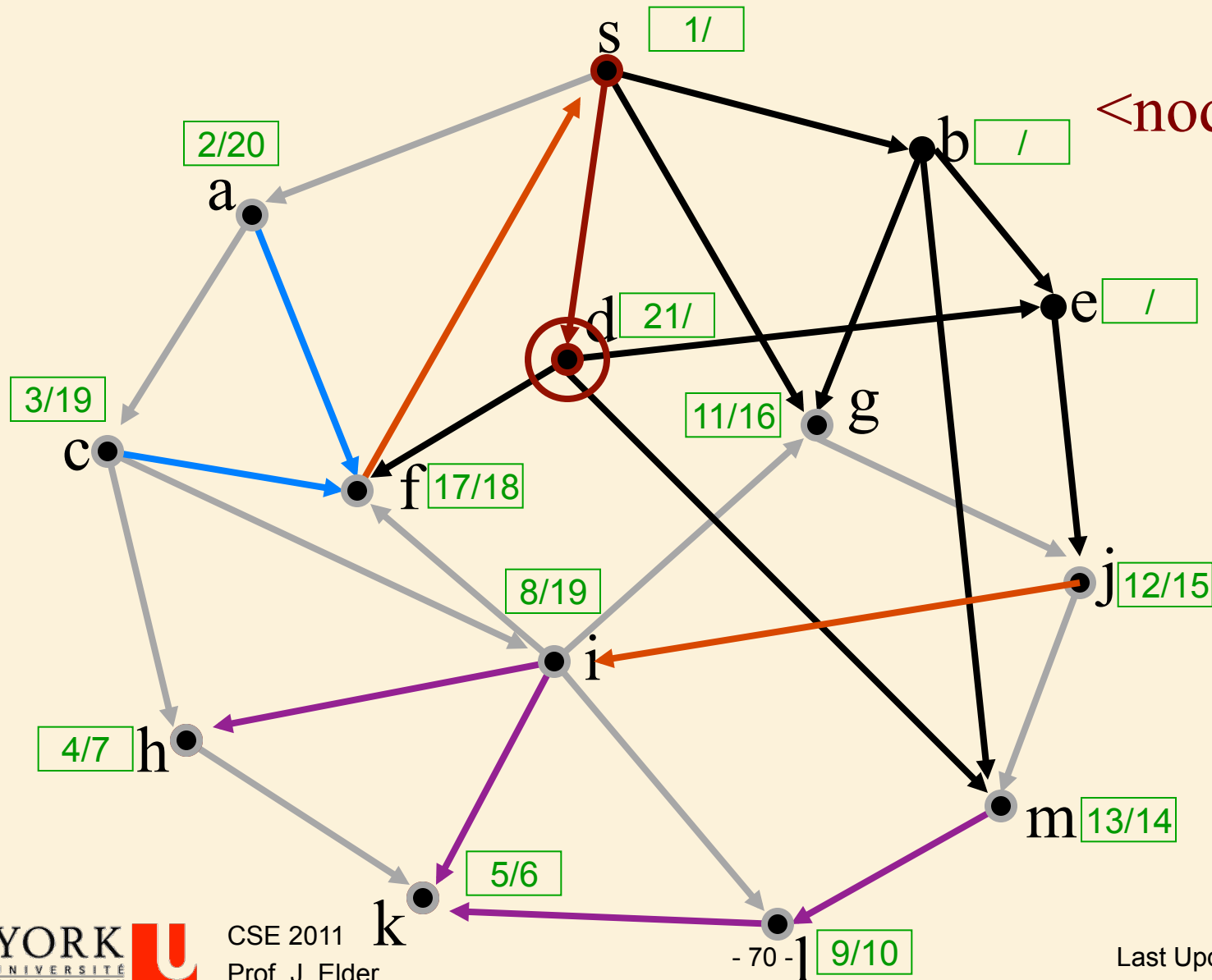


s, 1

DFS

Found
Not Handled
Stack

<node,# edges>

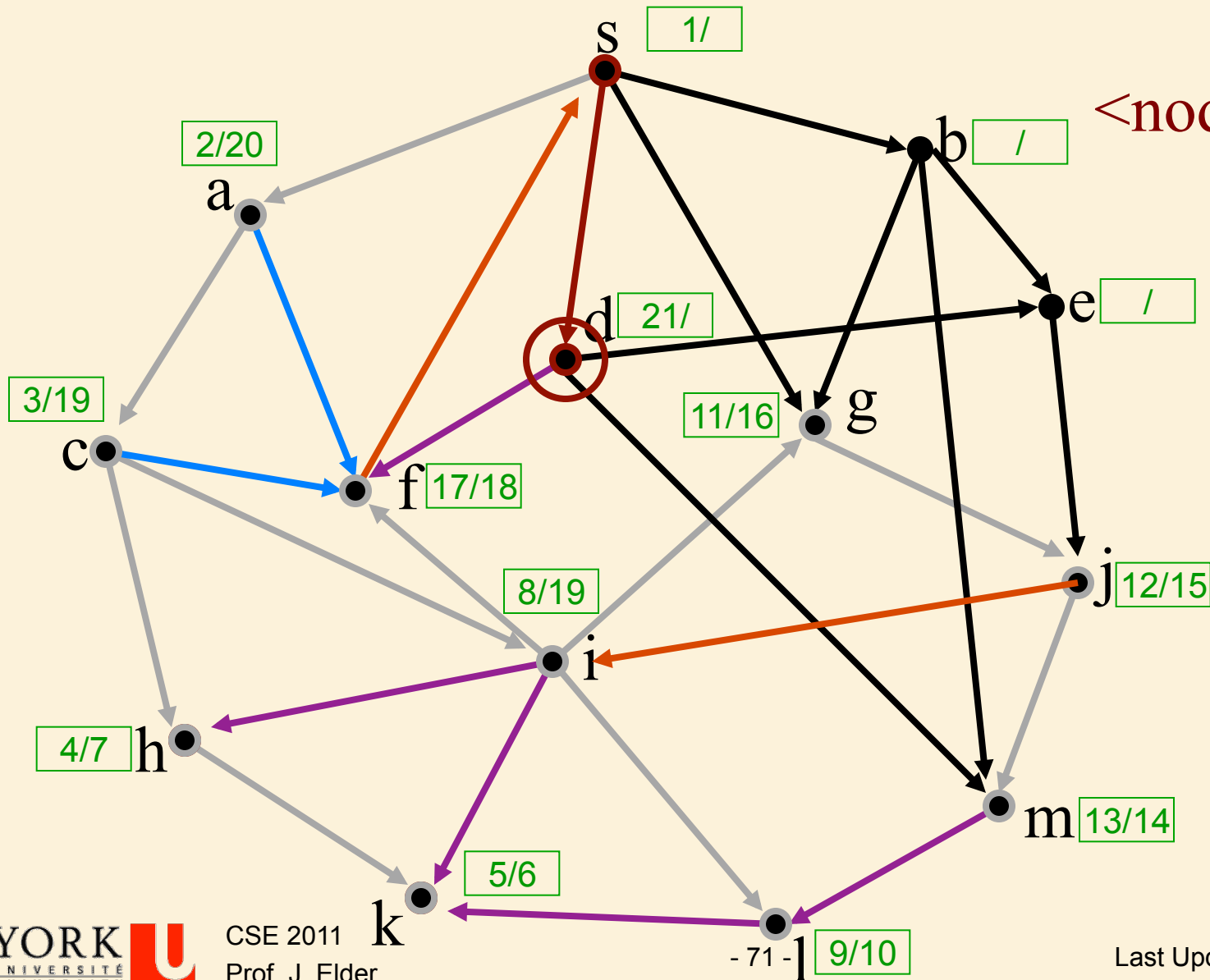


d,0
s,2

DFS

Found
Not Handled
Stack

<node,# edges>

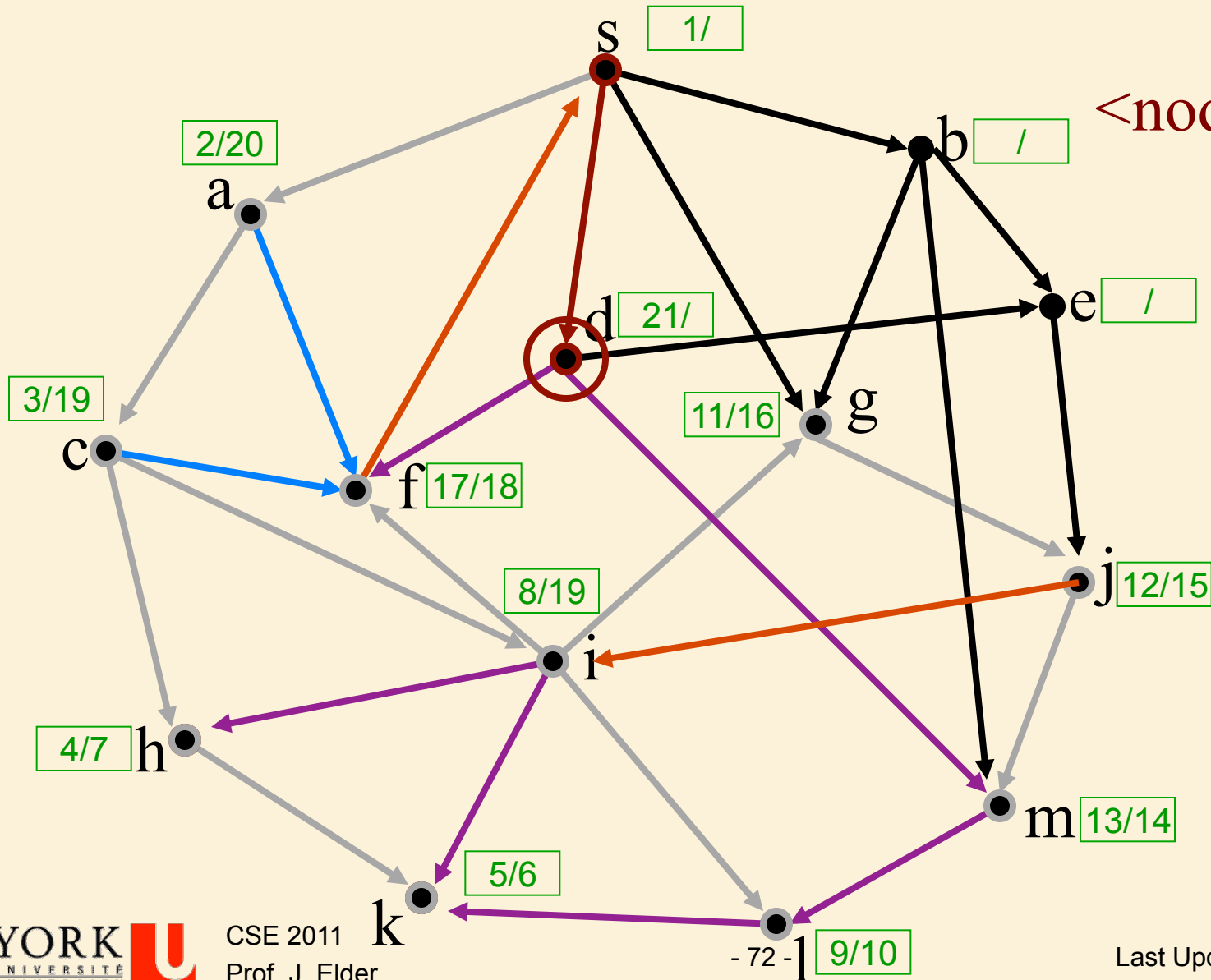


d,1
s,2

DFS

Found
Not Handled
Stack

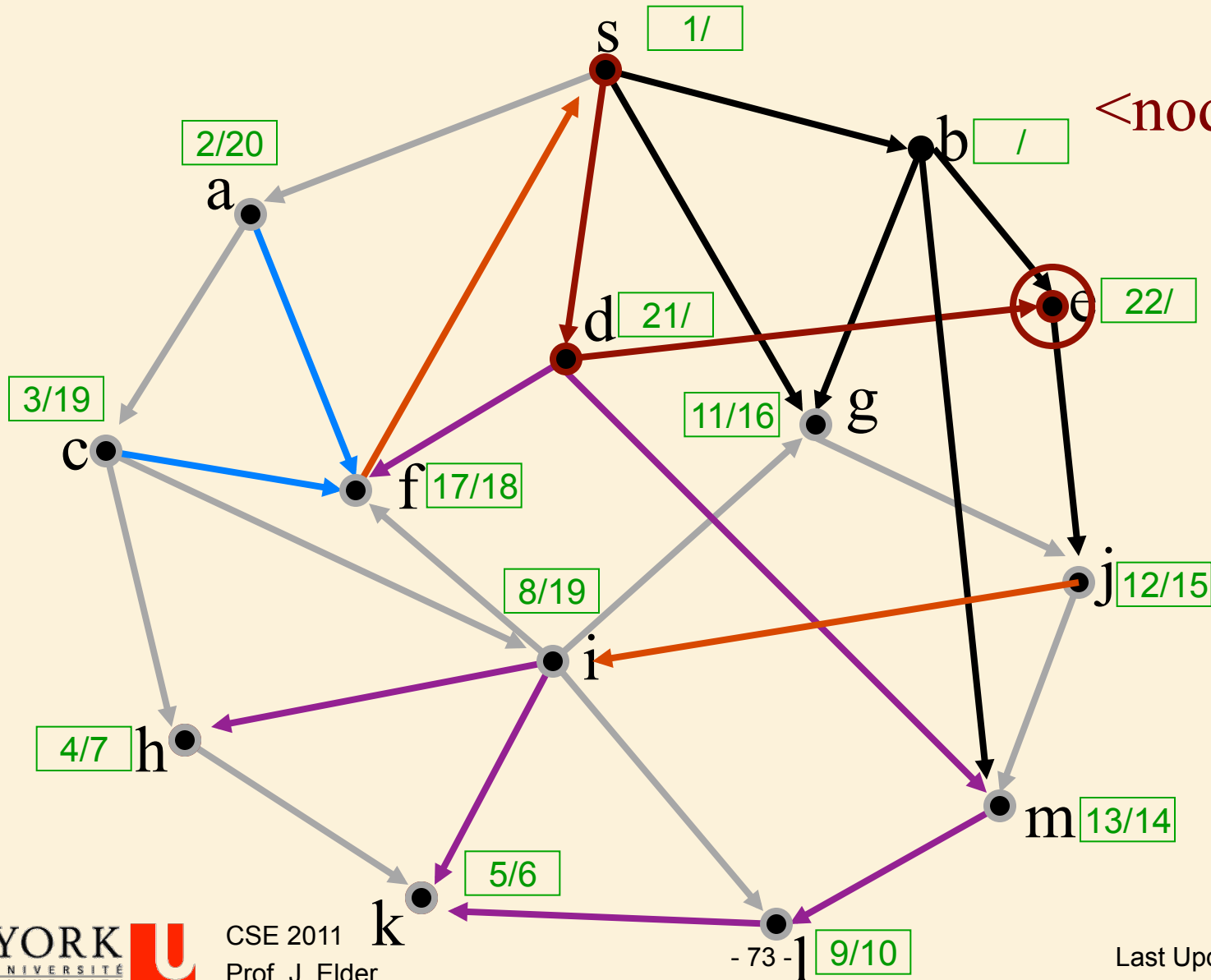
<node,# edges>



DFS

Found
Not Handled
Stack

<node,# edges>

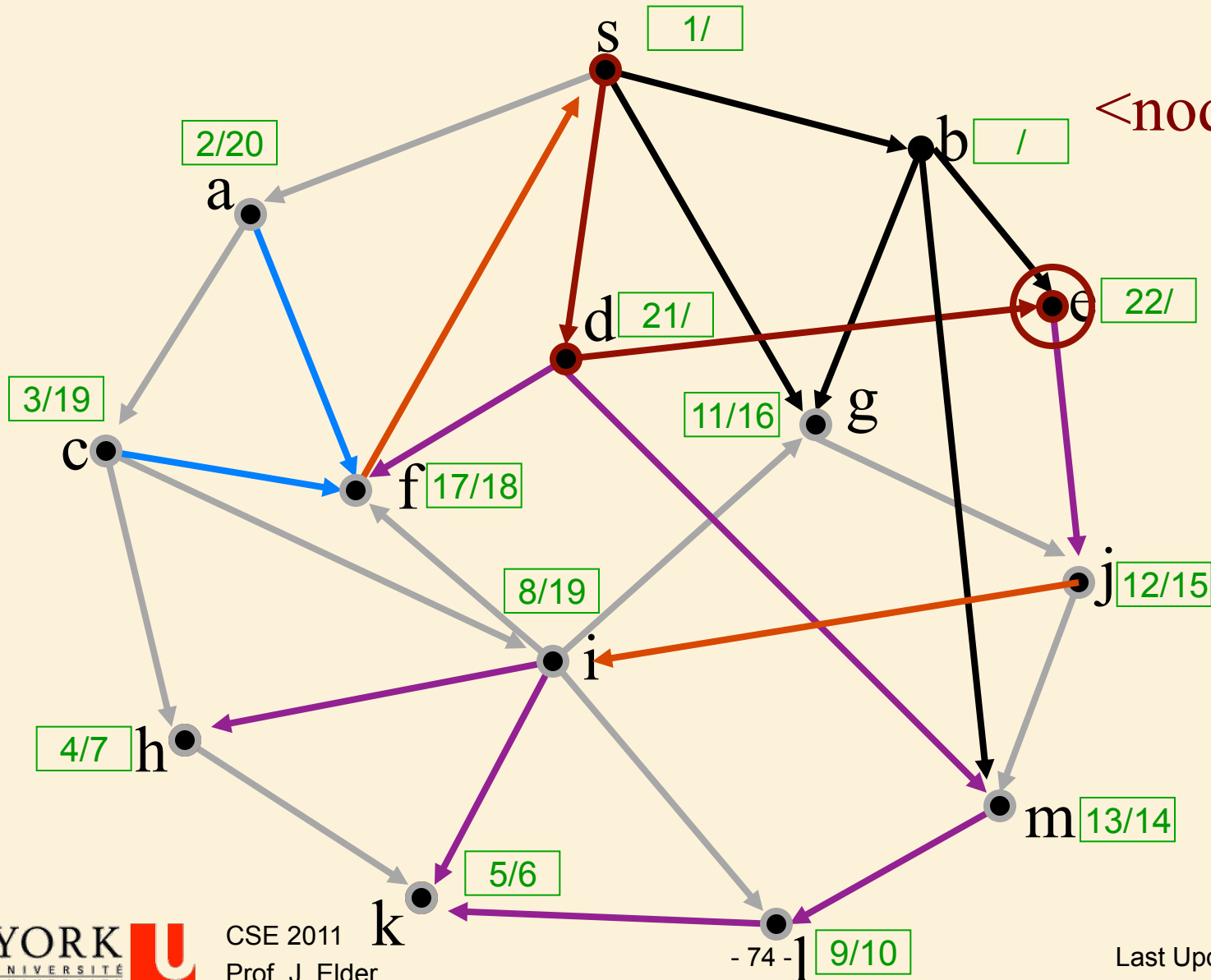


e,0
d,3
s,2

DFS

Found
Not Handled
Stack

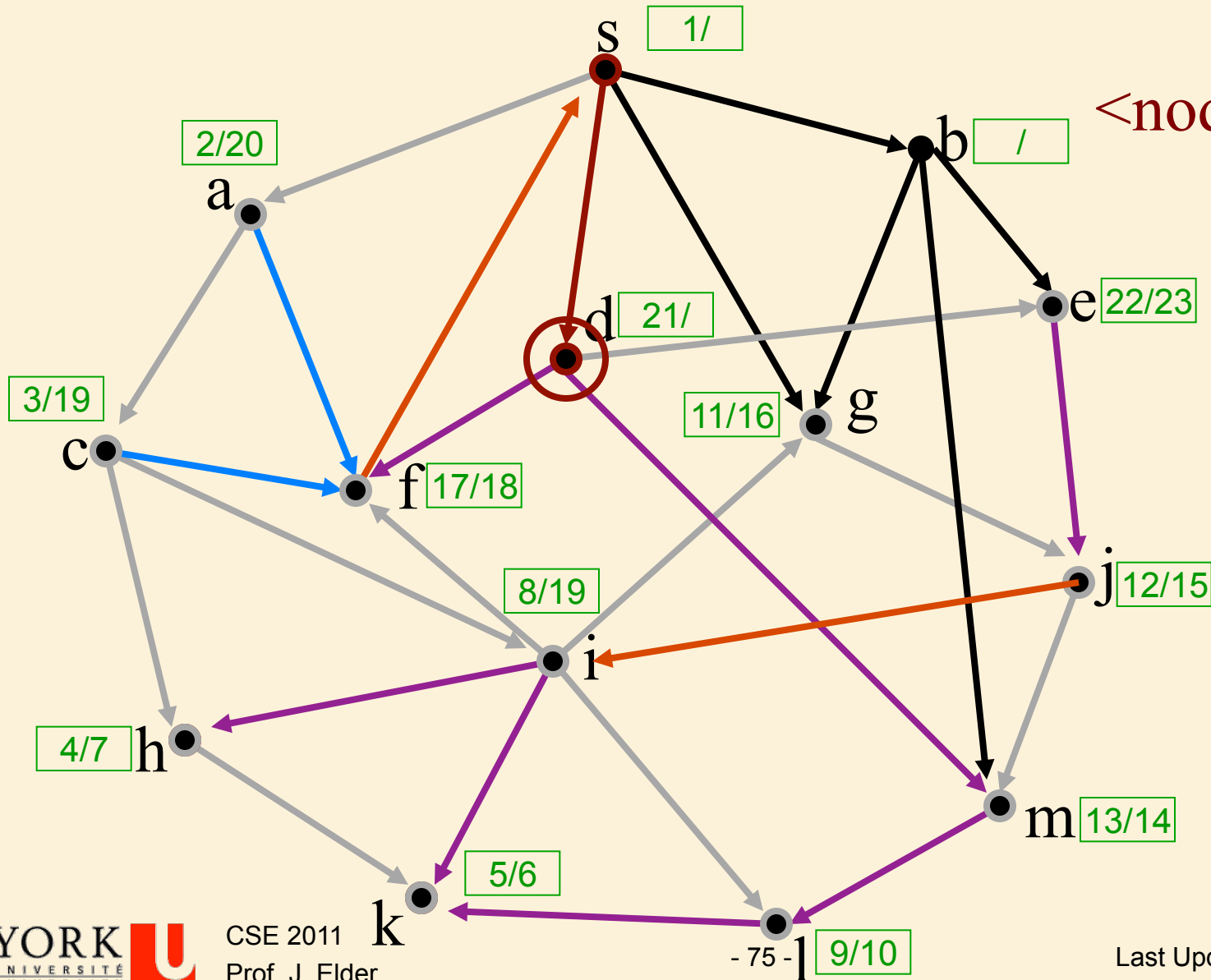
<node,# edges>



DFS

Found
Not Handled
Stack

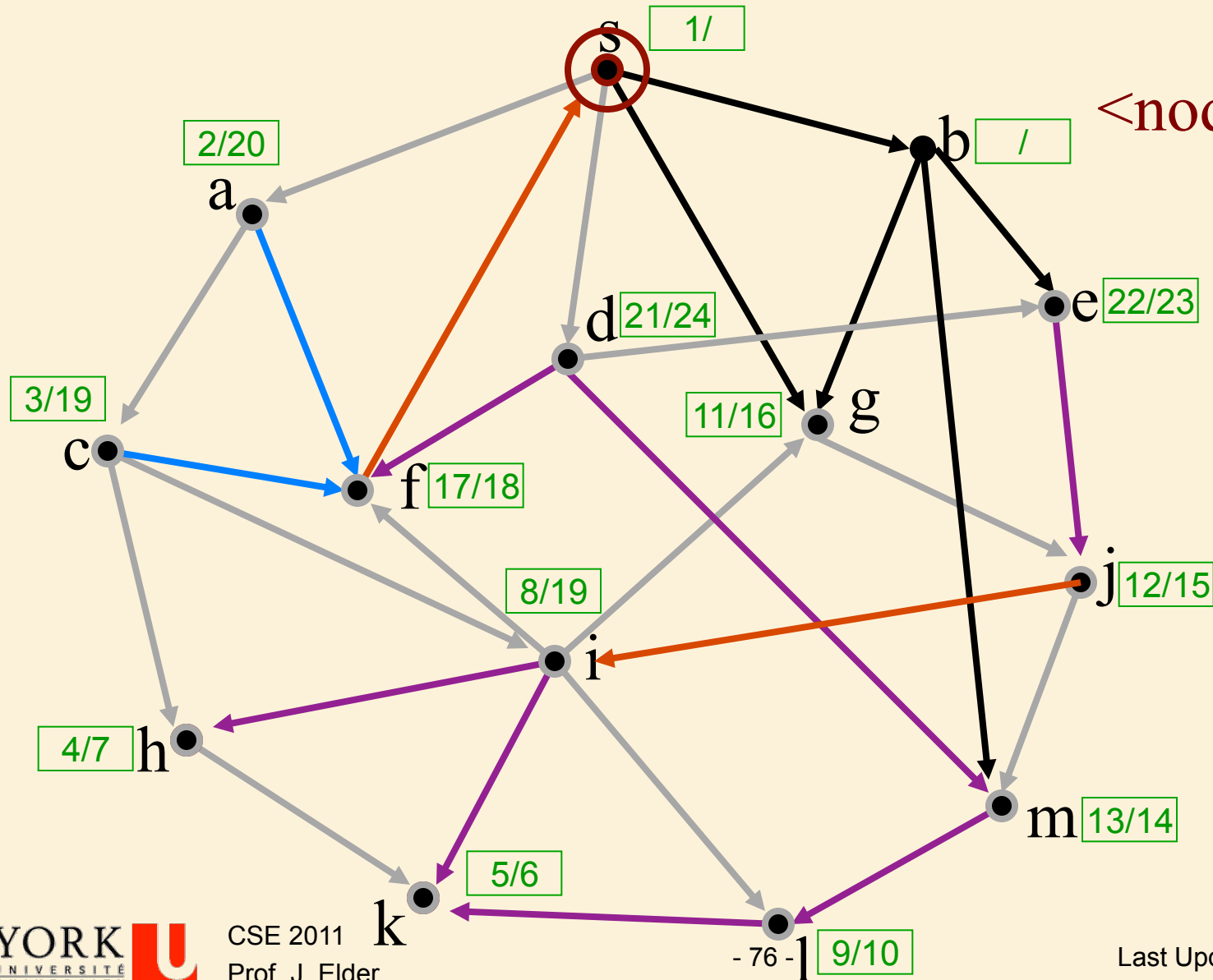
<node,# edges>



DFS

Found
Not Handled
Stack

<node,# edges>

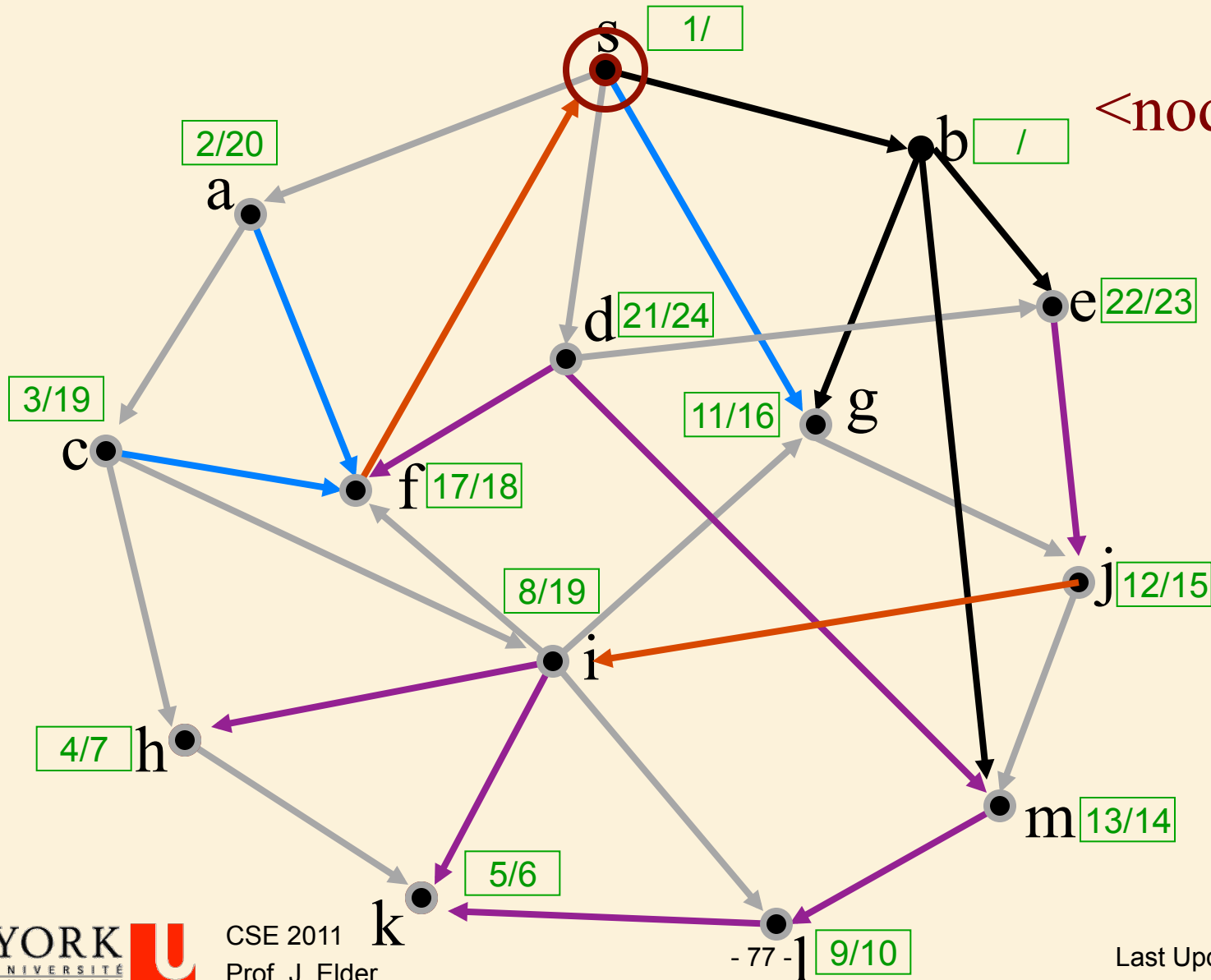


s,2

DFS

Found
Not Handled
Stack

<node,# edges>



s,3

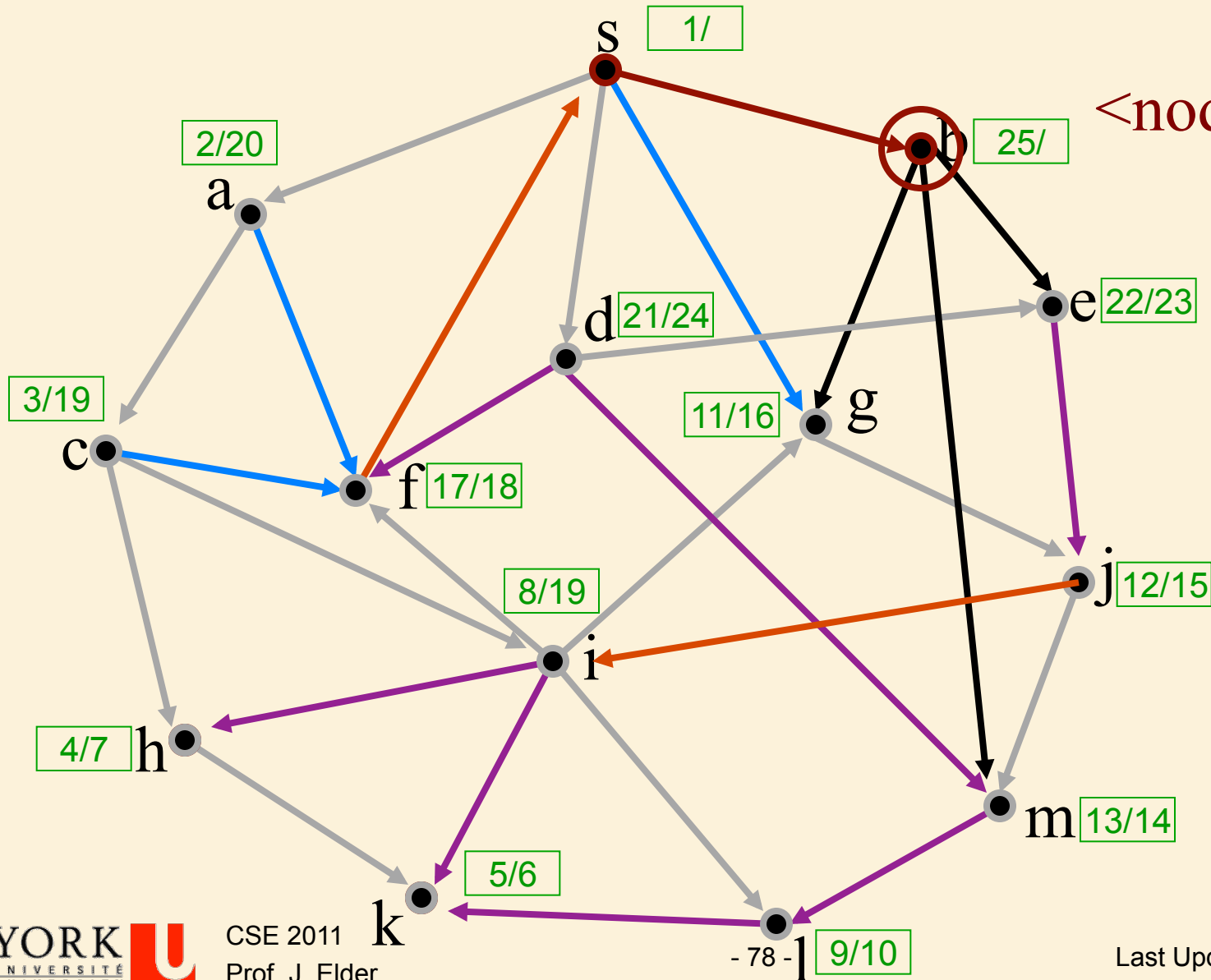
- 77 - 1 9/10

Last Updated: 12-03-22 10:12 AM

DFS

Found
Not Handled
Stack

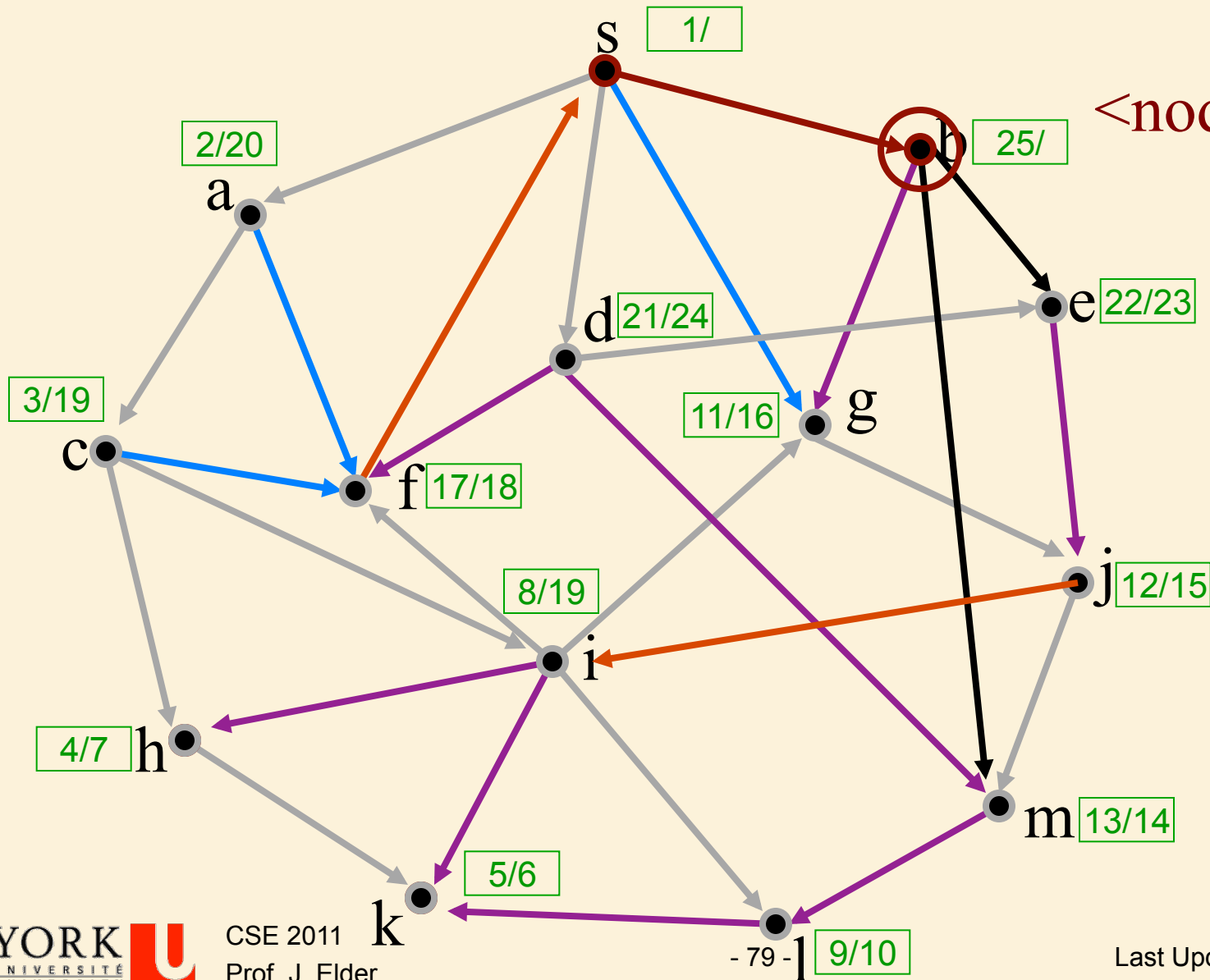
<node,# edges>



DFS

Found
Not Handled
Stack

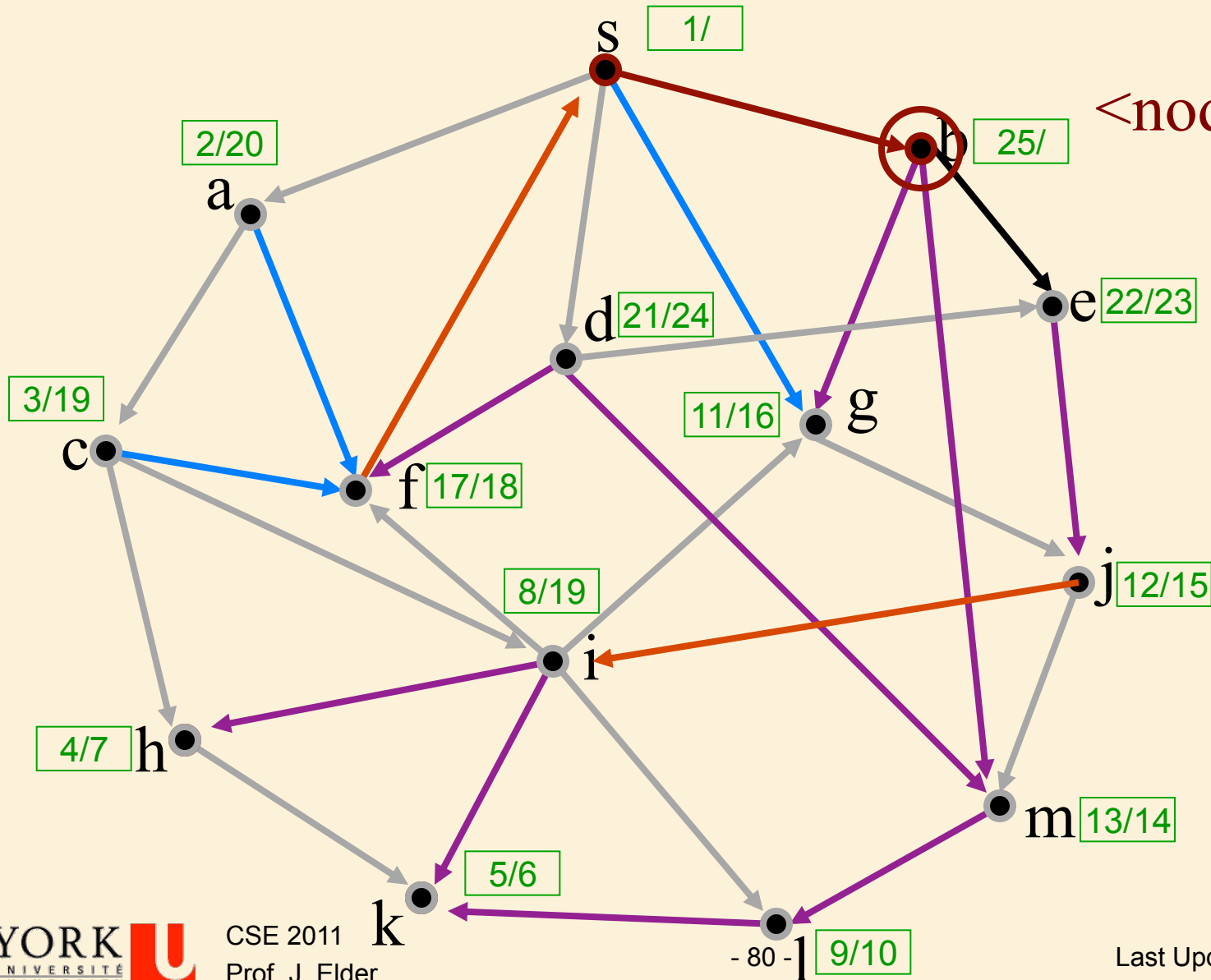
<node,# edges>



DFS

Found
Not Handled
Stack

<node,# edges>

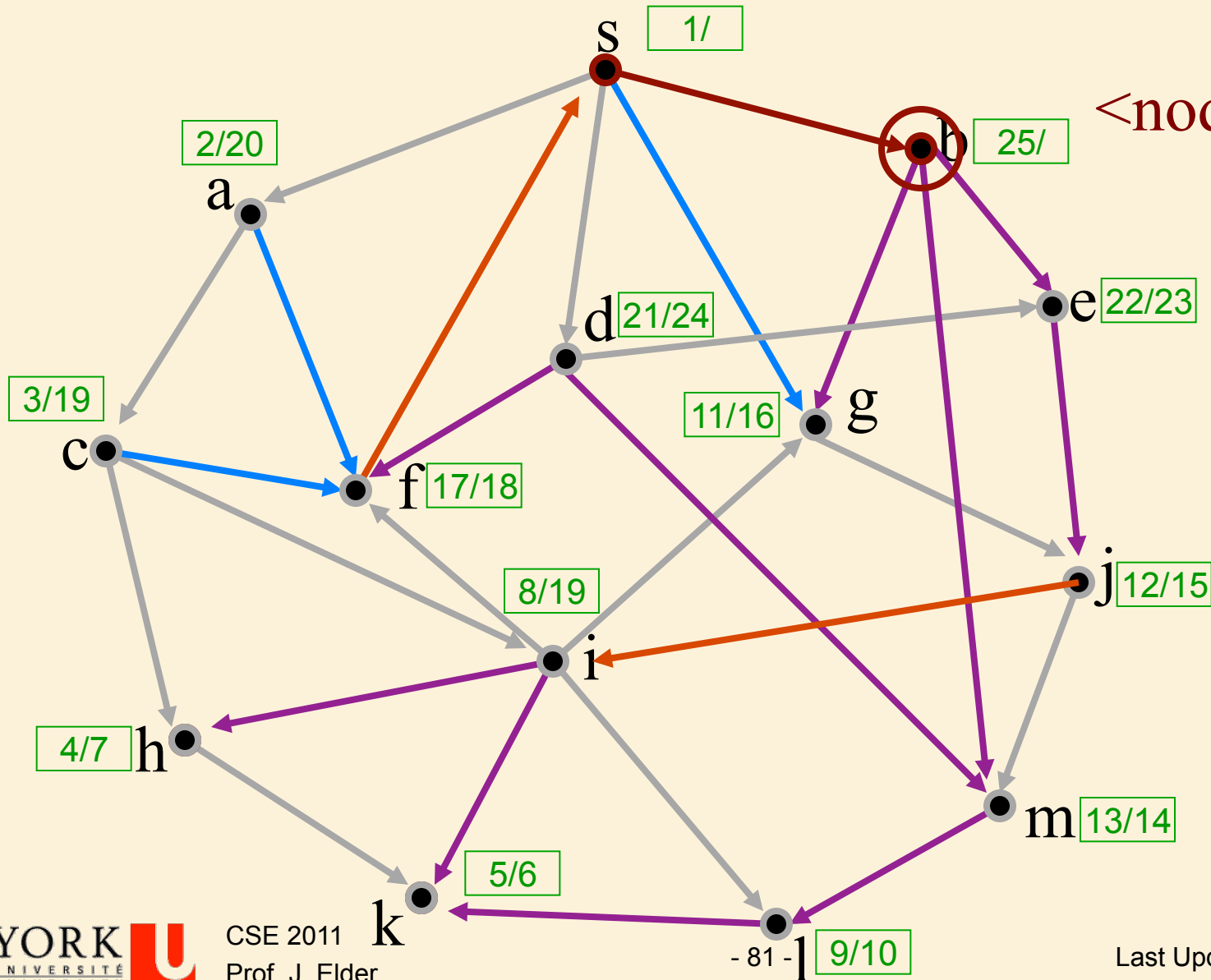


b,2
s,4

DFS

Found
Not Handled
Stack

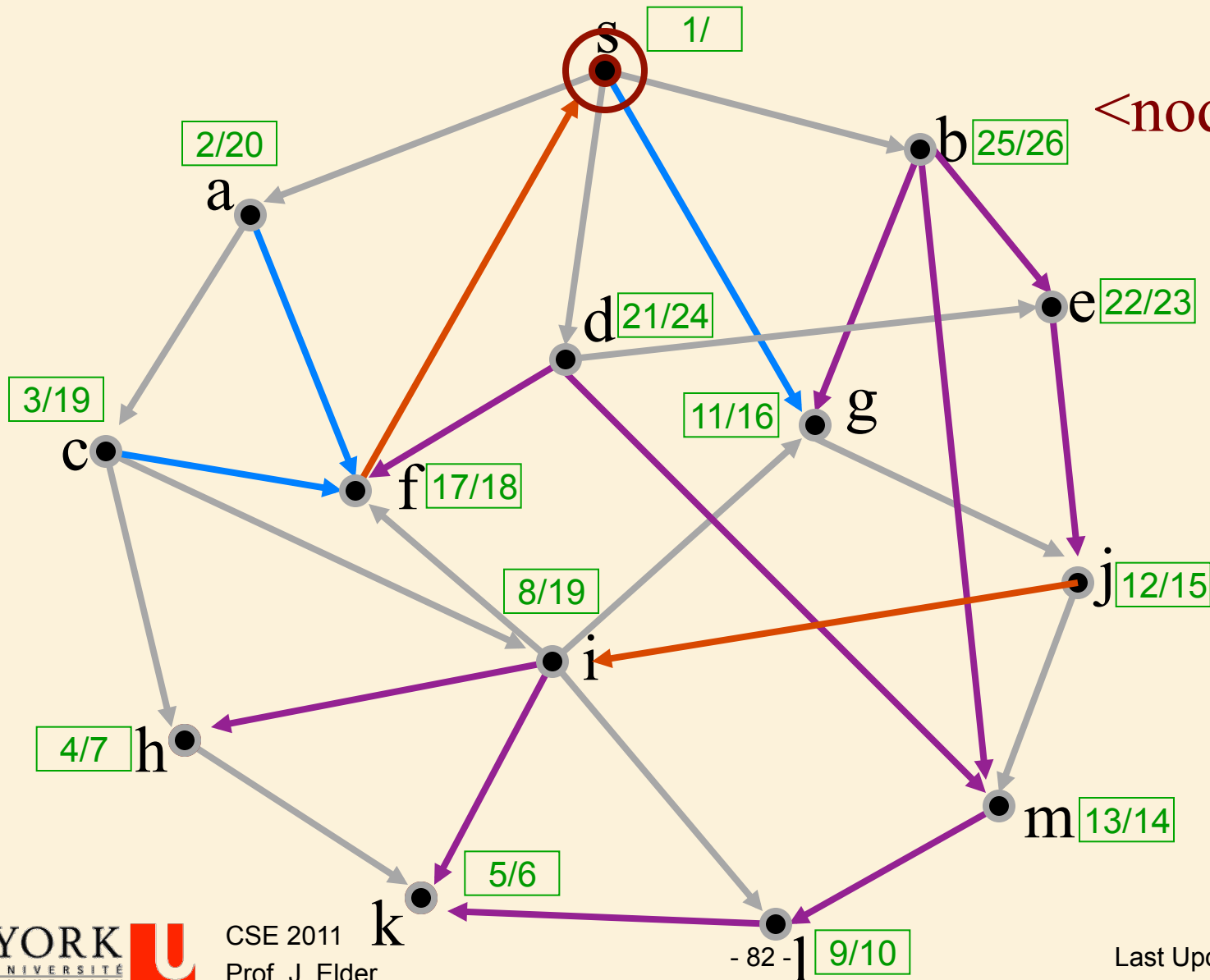
<node,# edges>



DFS

Found
Not Handled
Stack

<node,# edges>



s,4

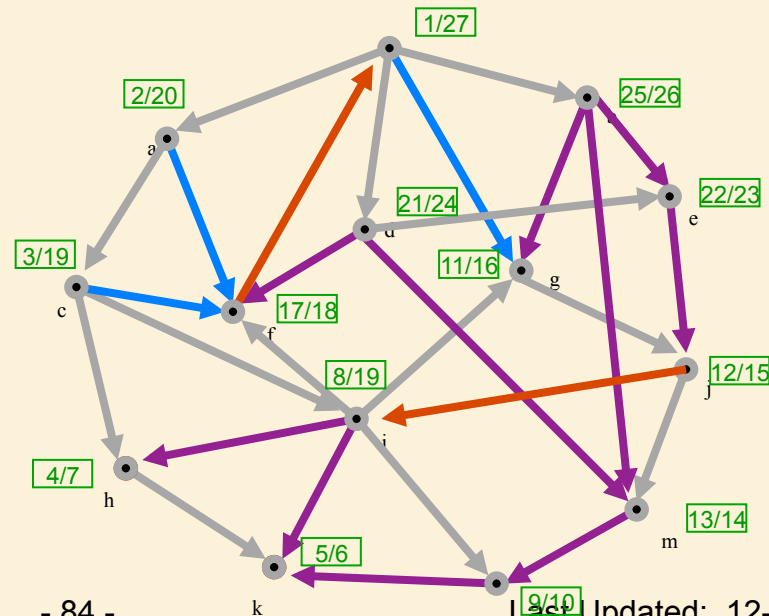
Found Not Handled Stack

Journal Pre-proof



Classification of Edges in DFS

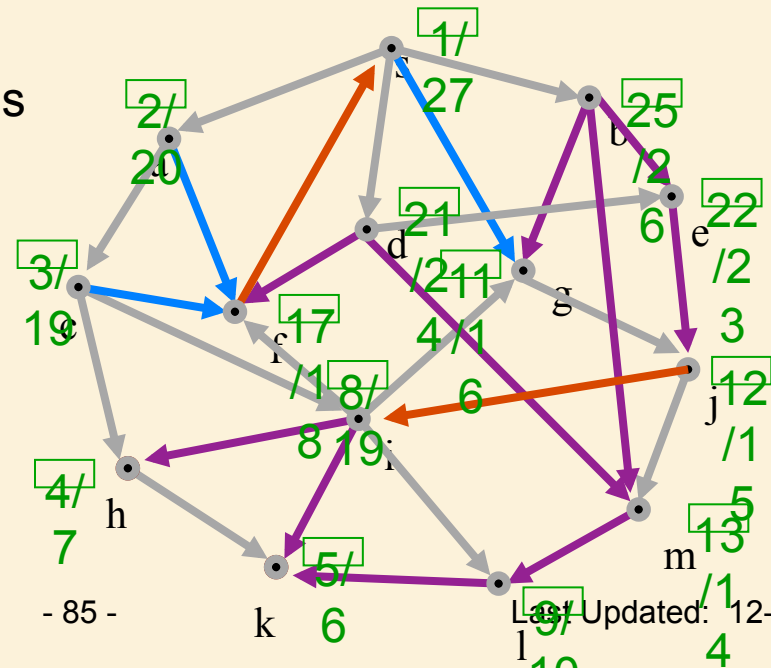
1. **Tree edges** are edges in the depth-first forest G_π . Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v) .
2. **Back edges** are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree.
3. **Forward edges** are non-tree edges (u, v) connecting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

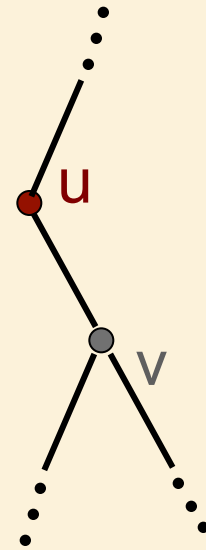


DFS on Undirected Graphs

- 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] $\leftarrow$ RED

push u onto stack

if $u = z$

 return list of elements on stack

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

 if color[v] = BLACK

 DFS-Path(v, z)

colour[u] $\leftarrow$ 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, if one exists

colour[u] $\leftarrow$ RED

push u onto stack

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

if color[v] = RED //back edge

return top of stack down to v

else if color[v] = BLACK

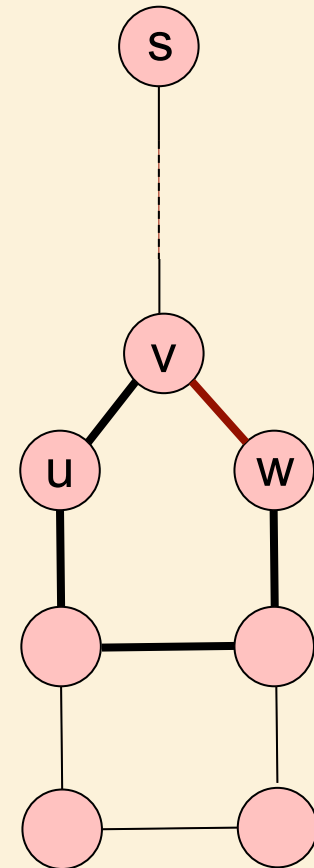
DFS-Cycle(v)

colour[u] $\leftarrow$ GRAY

pop u from stack

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, s) will be discovered.



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

Note: The textbook also describes a breadth-first TopologicalSort algorithm (Section 13.4.3)

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

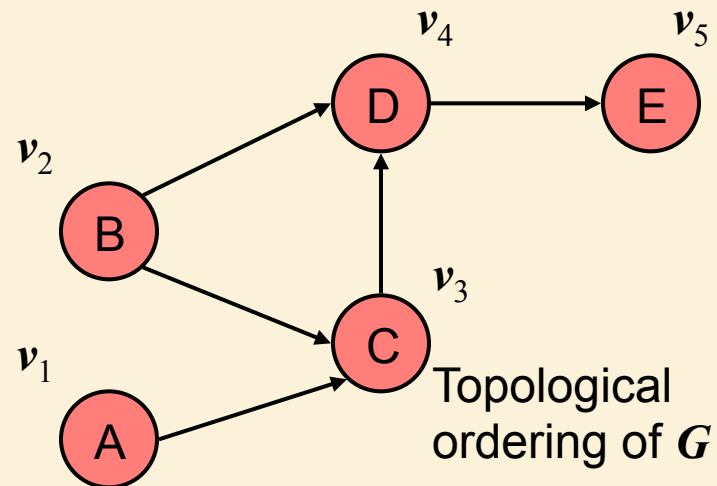
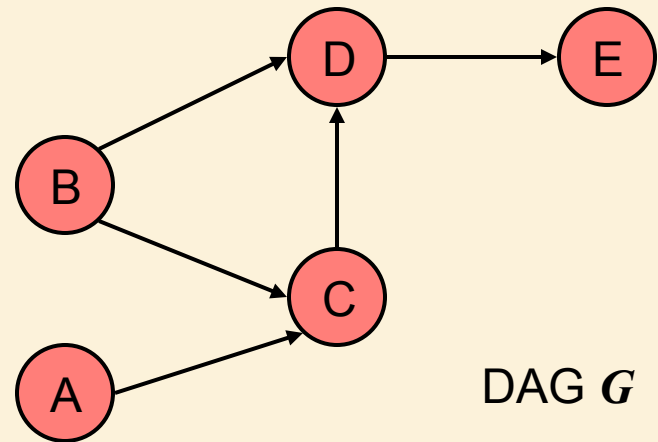
$$v_1, \dots, v_n$$

of the vertices such that for every edge (v_i, v_j) , we have $i < j$

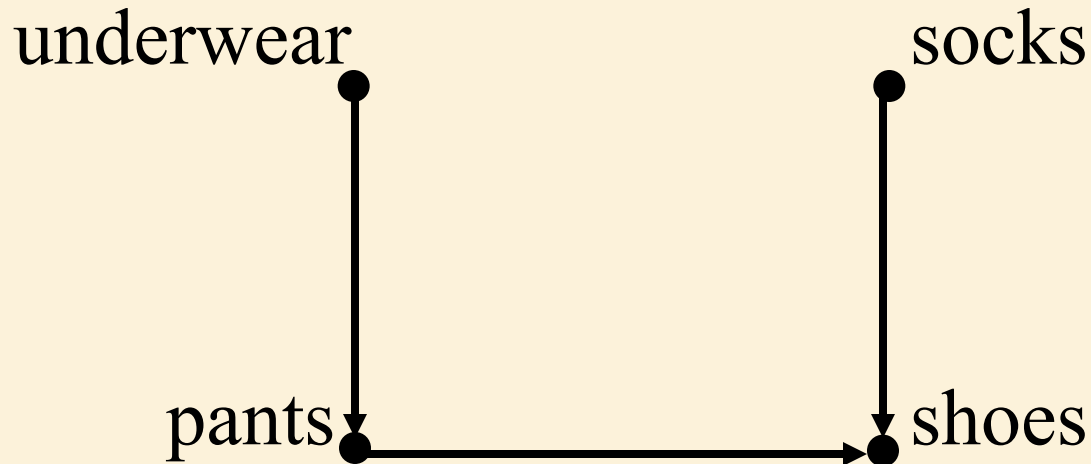
- Example: in a task scheduling digraph, a topological ordering is a task sequence that satisfies the precedence constraints

Theorem

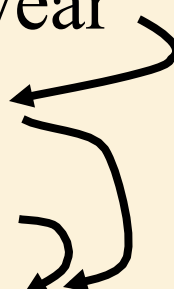
A digraph admits a topological ordering if and only if it is a DAG



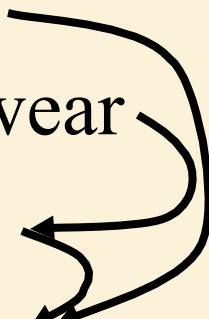
Topological (Linear) Order



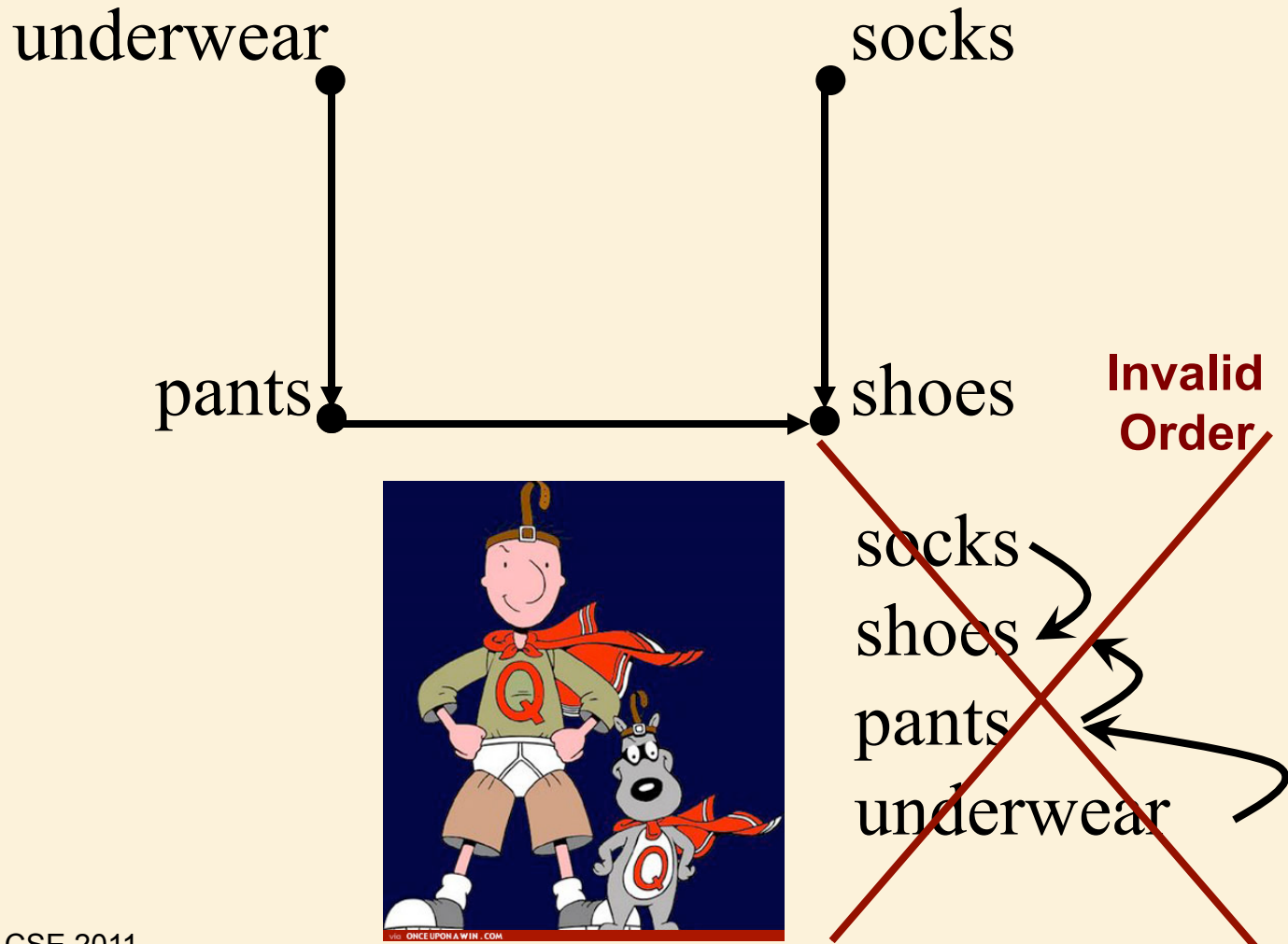
underwear
pants
socks
shoes



socks
underwear
pants
shoes



Topological (Linear) Order



Algorithm for Topological Sorting

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

Method TopologicalSort(**G**)

H $\leftarrow$ **G** // Temporary copy of **G**

n $\leftarrow$ **G.numVertices()**

while **H** is not empty **do**

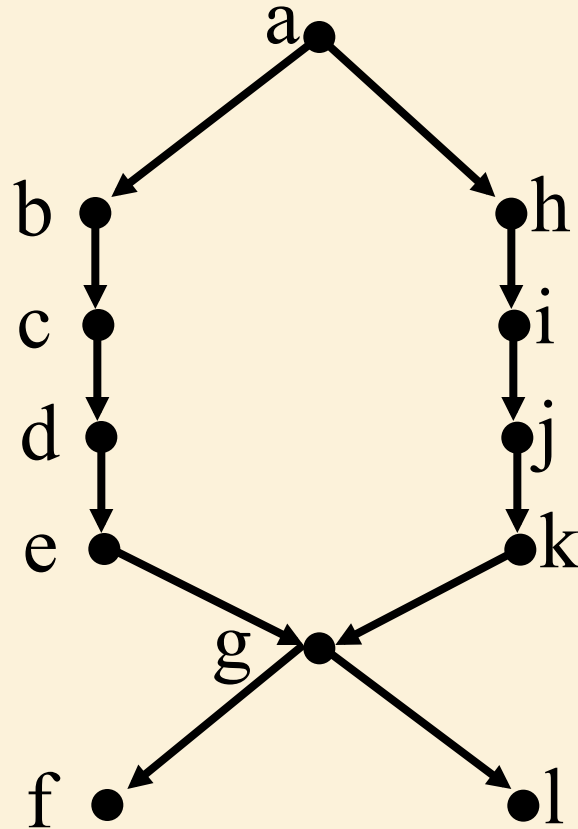
 Let **v** be a vertex with no outgoing edges

 Label **v** $\leftarrow$ **n**

n $\leftarrow$ **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:

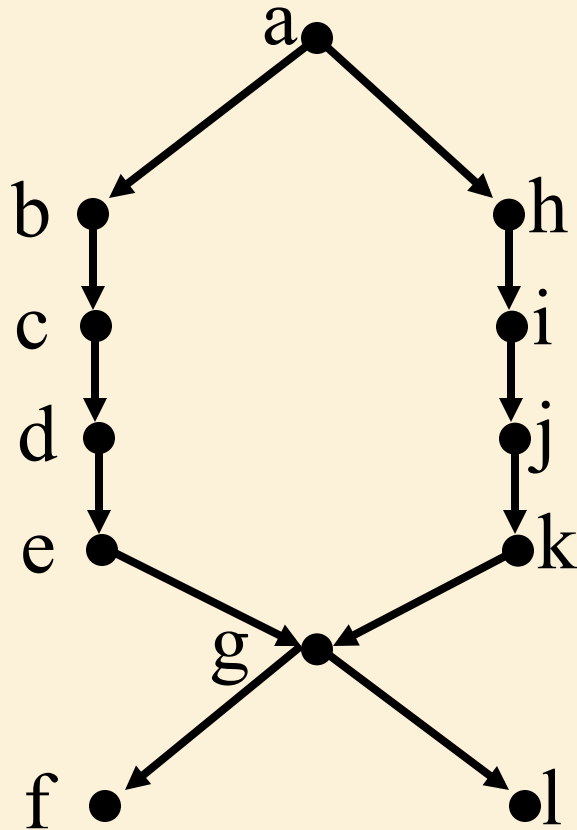
- Find a terminal node (sink).
 - Put it last in sequence.
 - Delete from graph & repeat
- } $O(|V|)$

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

..... 1 Can we do better?

Linear Order

Alg: DFS

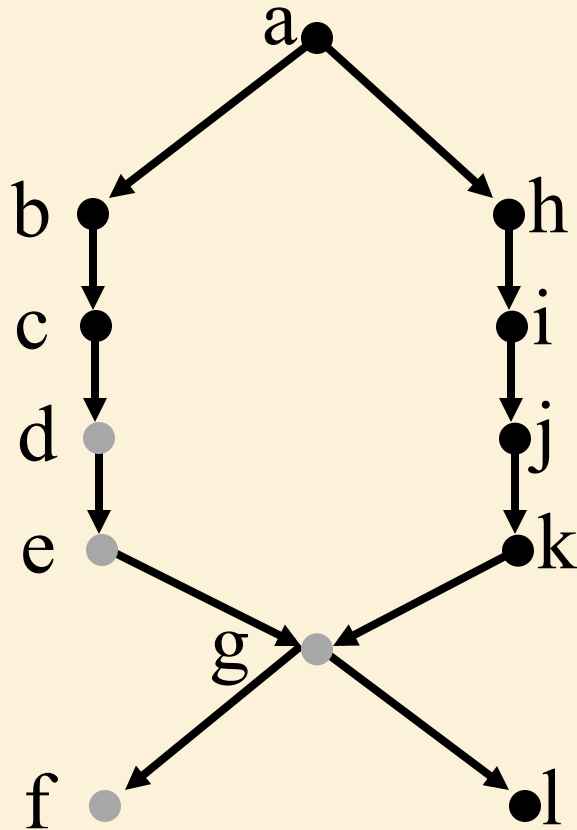


Found
Not Handled
Stack

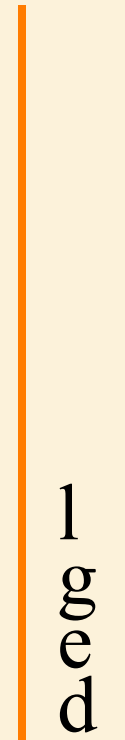


Linear Order

Alg: DFS



Found
Not Handled
Stack



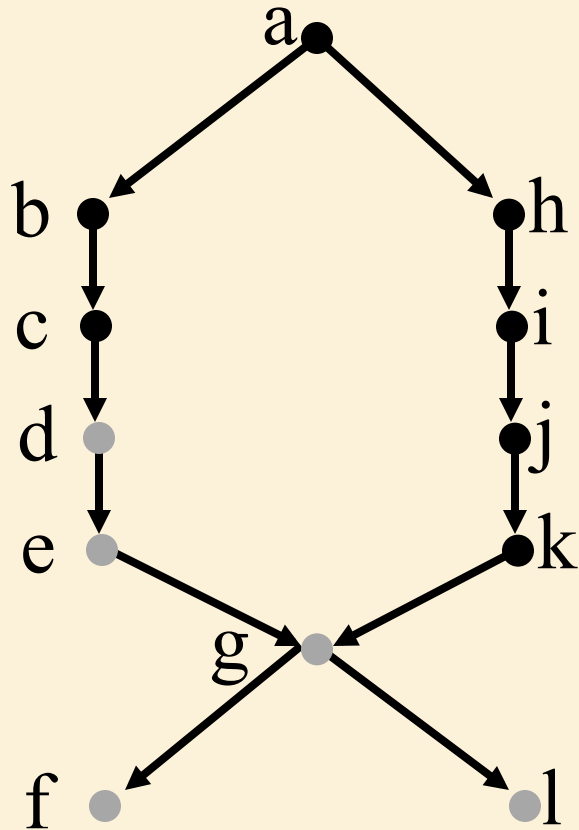
When node is popped off stack, insert at front of linearly-ordered “to do” list.

Linear Order:

..... f

Linear Order

Alg: DFS



Found
Not Handled
Stack

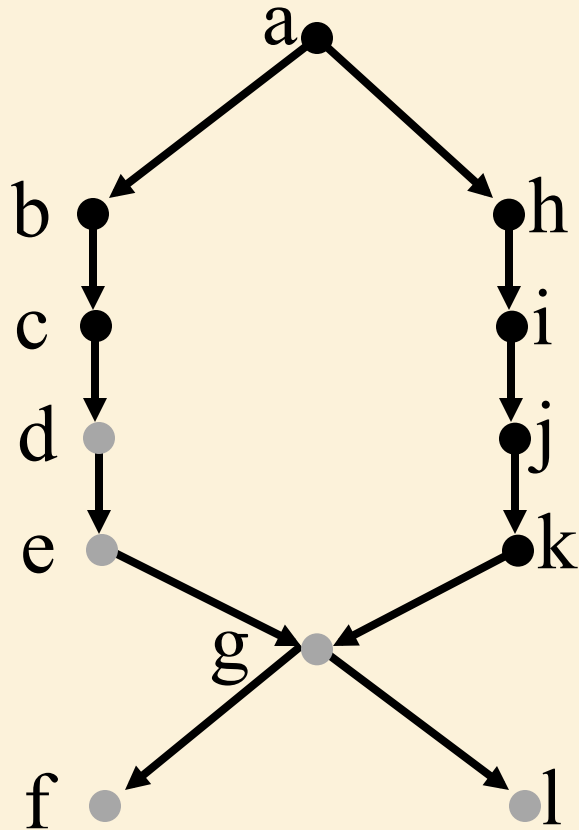


Linear Order:

l,f

Linear Order

Alg: DFS



Found
Not Handled
Stack

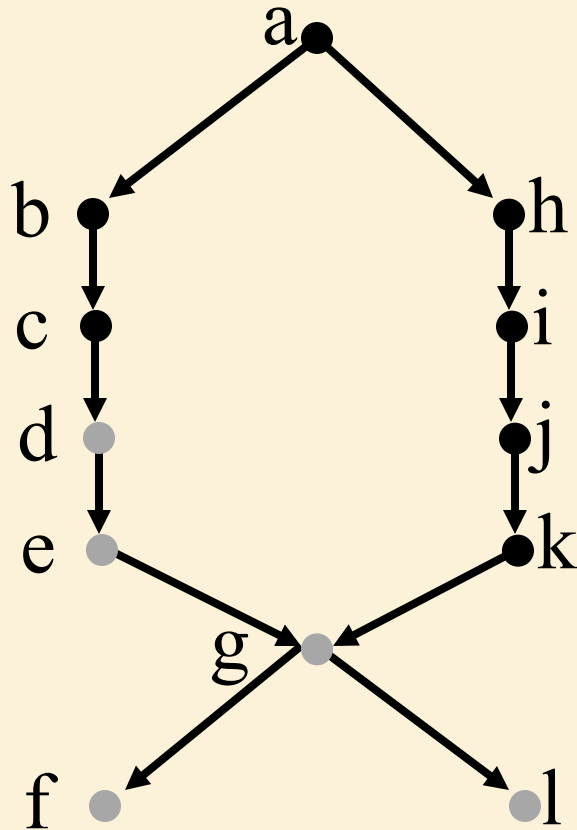


Linear Order:

g,l,f

Linear Order

Alg: DFS



Found
Not Handled
Stack

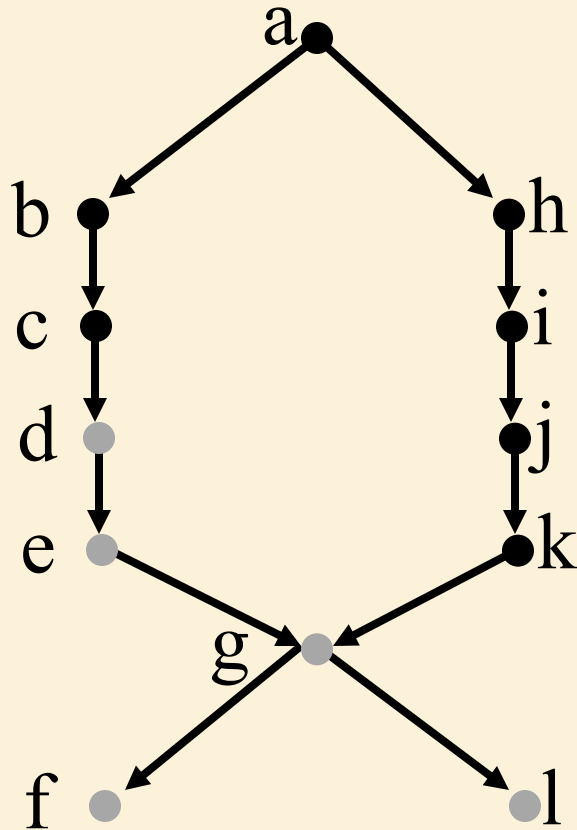


Linear Order:

e,g,l,f

Linear Order

Alg: DFS



Found
Not Handled
Stack

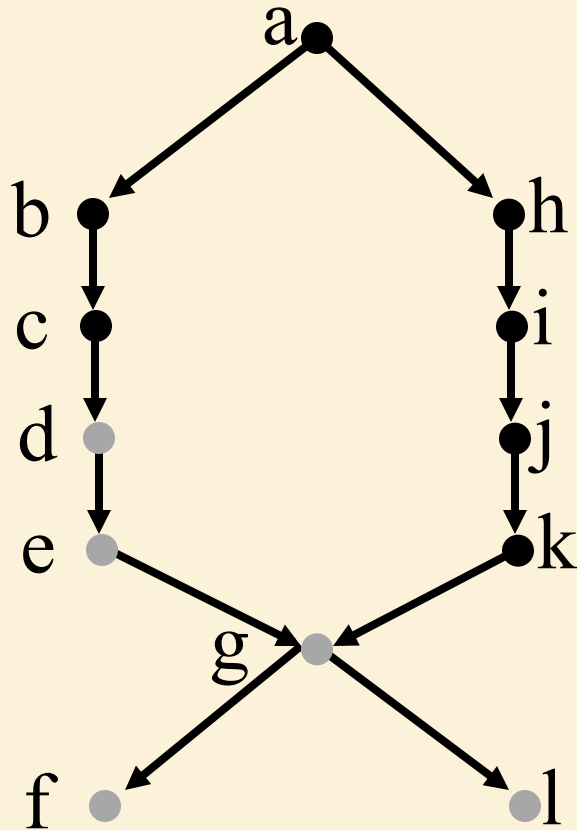


Linear Order:

d,e,g,l,f

Linear Order

Alg: DFS



Found
Not Handled
Stack

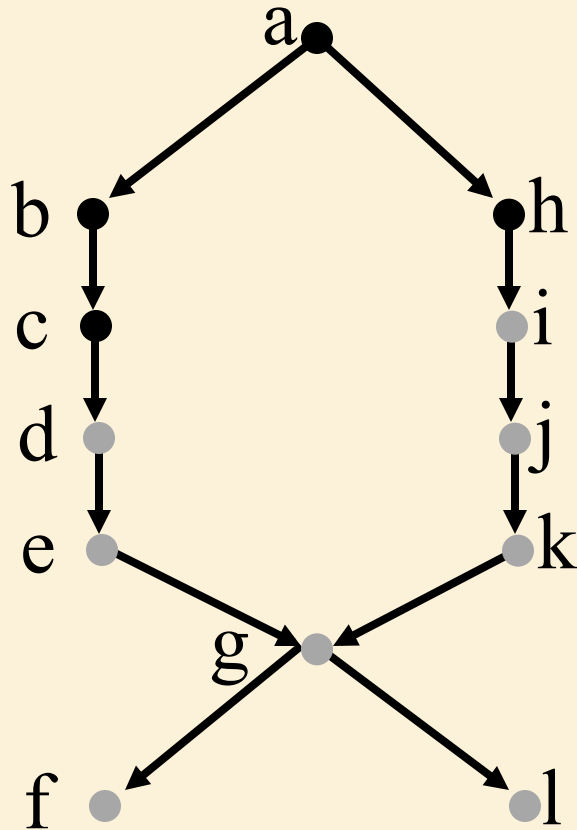


Linear Order:

d,e,g,l,f

Linear Order

Alg: DFS



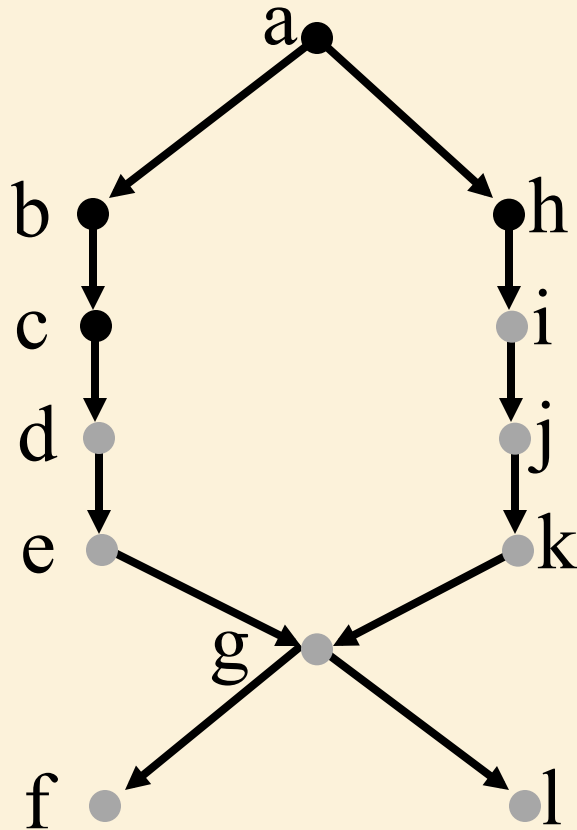
Found
Not Handled
Stack



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

Linear Order

Alg: DFS



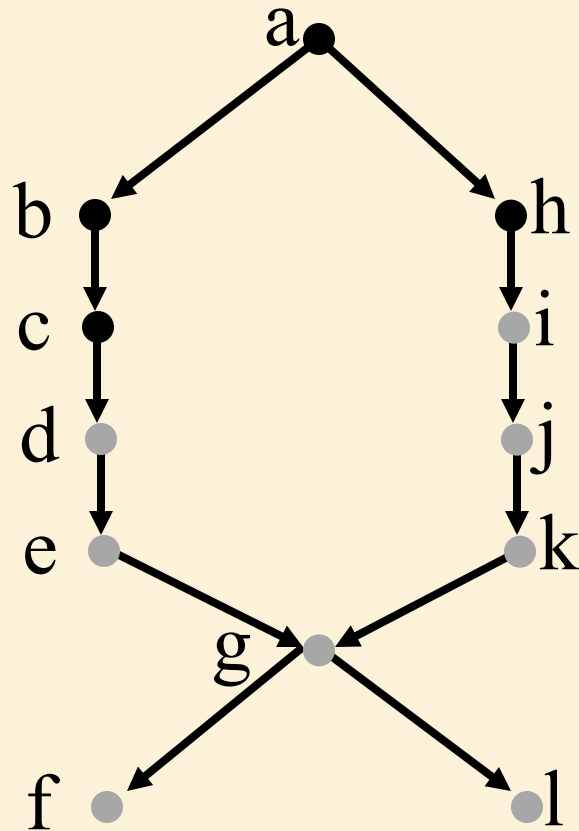
Found
Not Handled
Stack



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

Linear Order

Alg: DFS



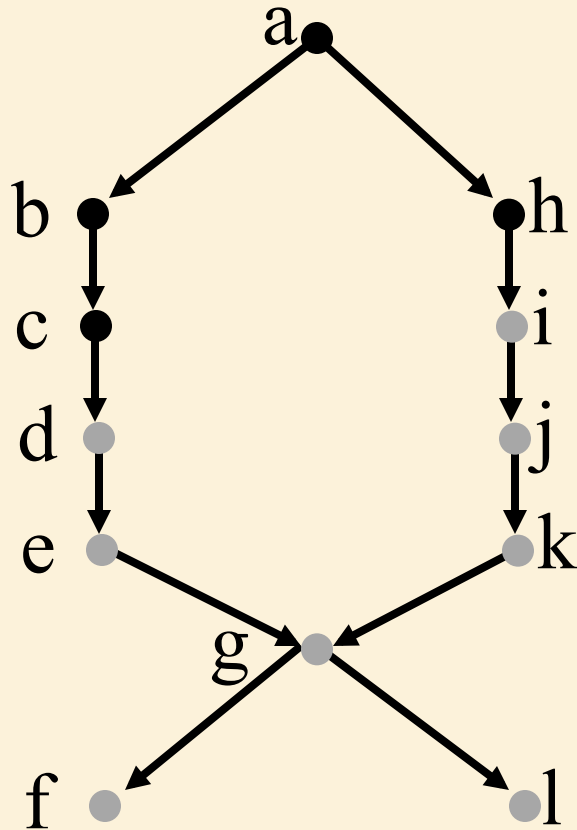
Found
Not Handled
Stack



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

Linear Order

Alg: DFS



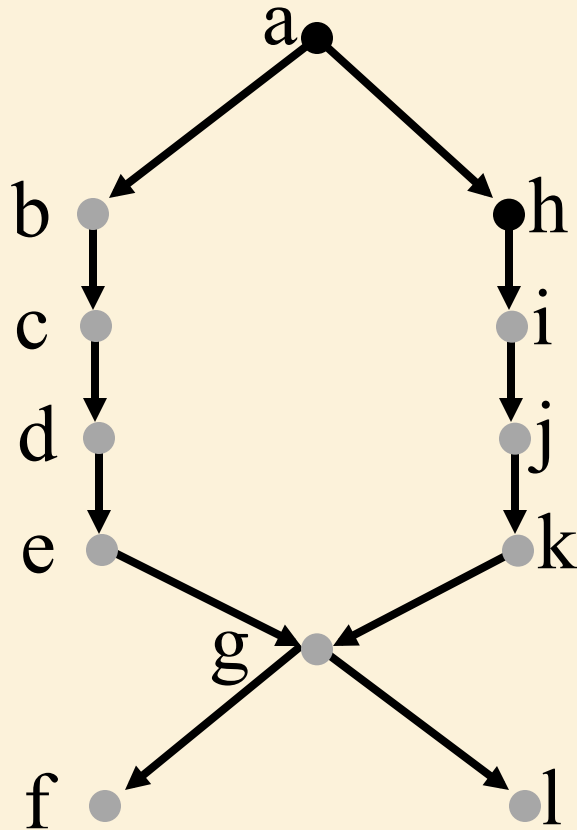
Found
Not Handled
Stack



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

Linear Order

Alg: DFS



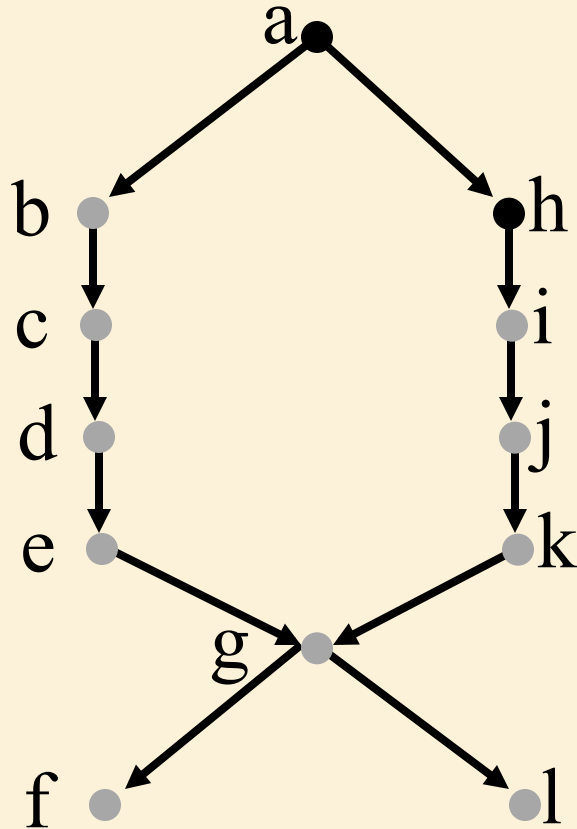
Found
Not Handled
Stack



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

Linear Order

Alg: DFS



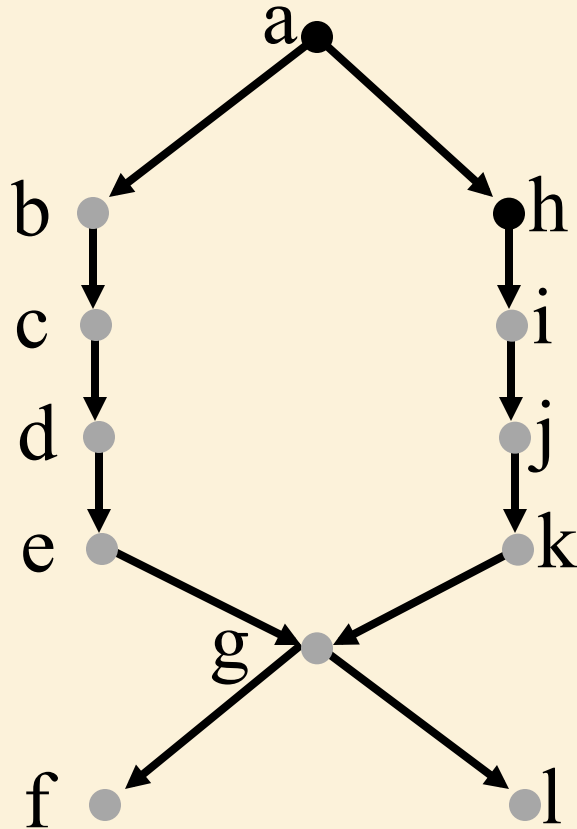
Found
Not Handled
Stack



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

Linear Order

Alg: DFS



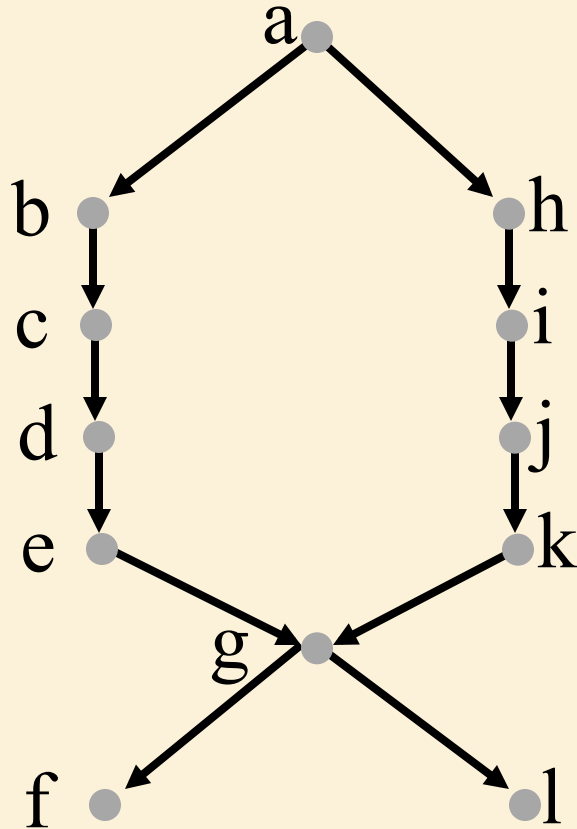
Found
Not Handled
Stack



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

Linear Order

Alg: DFS



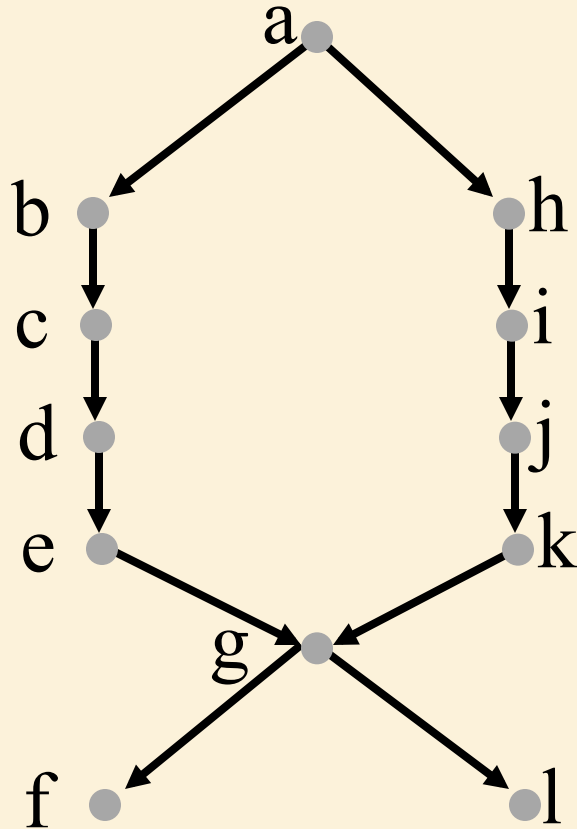
Found
Not Handled
Stack



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 Topological Sort

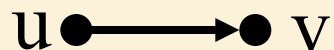
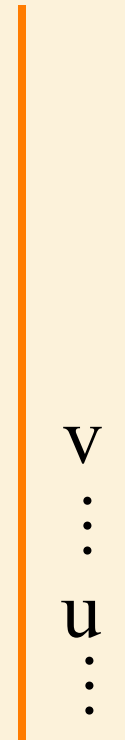
➤ Makes sense. But how do we prove that it works?

Linear Order

Proof: Consider each edge

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

Found
Not Handled
Stack



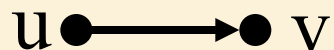
$u \dots v \dots$

Linear Order

Proof: Consider each edge

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

Found
Not Handled
Stack



$u \dots v \dots$

Linear Order

Proof: Consider each edge

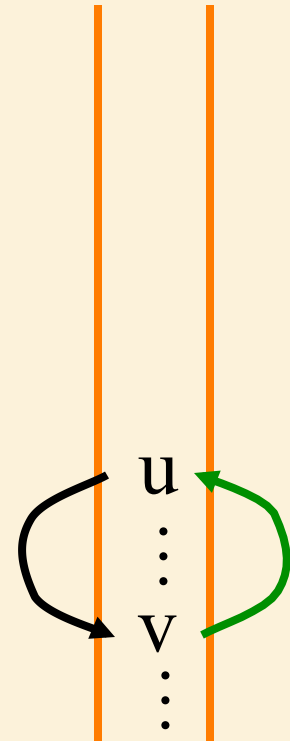
- Case 1: u goes on stack first before v.
- Case 2: v goes on stack first before u.
v comes off before u goes on.
- Case 3: v goes on stack first before u.
u goes on before v comes off.
- Panic: u goes after v in order. ☹
- Cycle means linear order
is impossible ☺



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

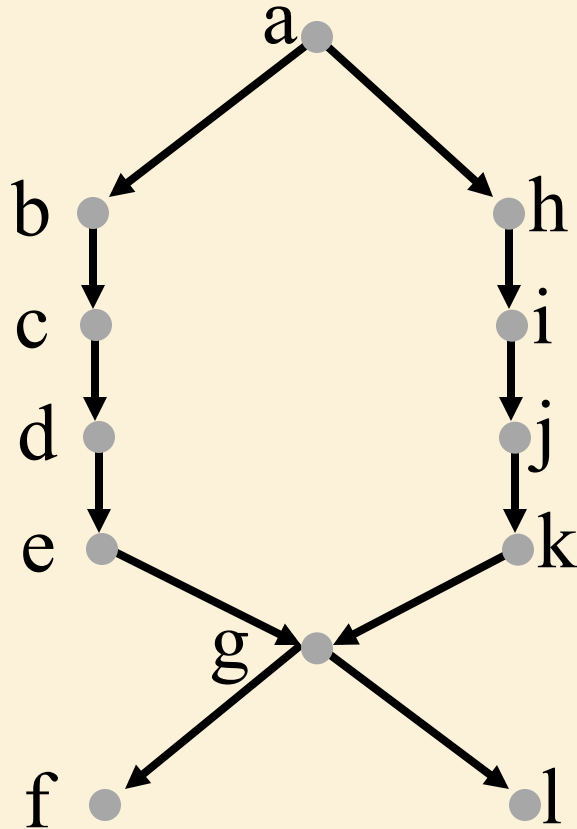
u ● → ● v

Found
Not Handled
Stack



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!

End of Lecture

March 22, 2012

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

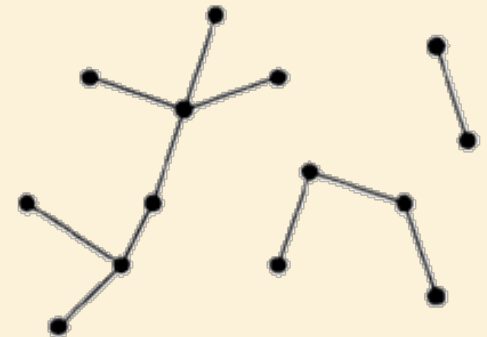
for each vertex $u \in V[G]$

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

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

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

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

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

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

Topological-Sort-Visit(v)

push u onto stack

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

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
 - ❑ Cycle detection
 - ❑ **Find and report a path with the minimum number of edges between two given vertices**

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  $u \in V[G]$ 
    color[ $u$ ]  $\leftarrow$  BLACK //initialize vertex
colour[ $s$ ]  $\leftarrow$  RED
Q.enqueue( $s$ )
while  $Q \neq \emptyset$ 
     $u \leftarrow$  Q.dequeue()
    for each  $v \in \text{Adj}[u]$  //explore edge ( $u, v$ )
        if color[ $v$ ] = BLACK
            colour[ $v$ ]  $\leftarrow$  RED
            Q.enqueue( $v$ )
    colour[ $u$ ]  $\leftarrow$  GRAY
```

BFS is a Level-Order Traversal

- Notice that in BFS exploration takes place on a wavefront consisting of nodes that are all the same distance from the source s .
- We can label these successive wavefronts by their distance: $L_0, L_1, \dots$

BFS Example



undiscovered



discovered (on Queue)



finished



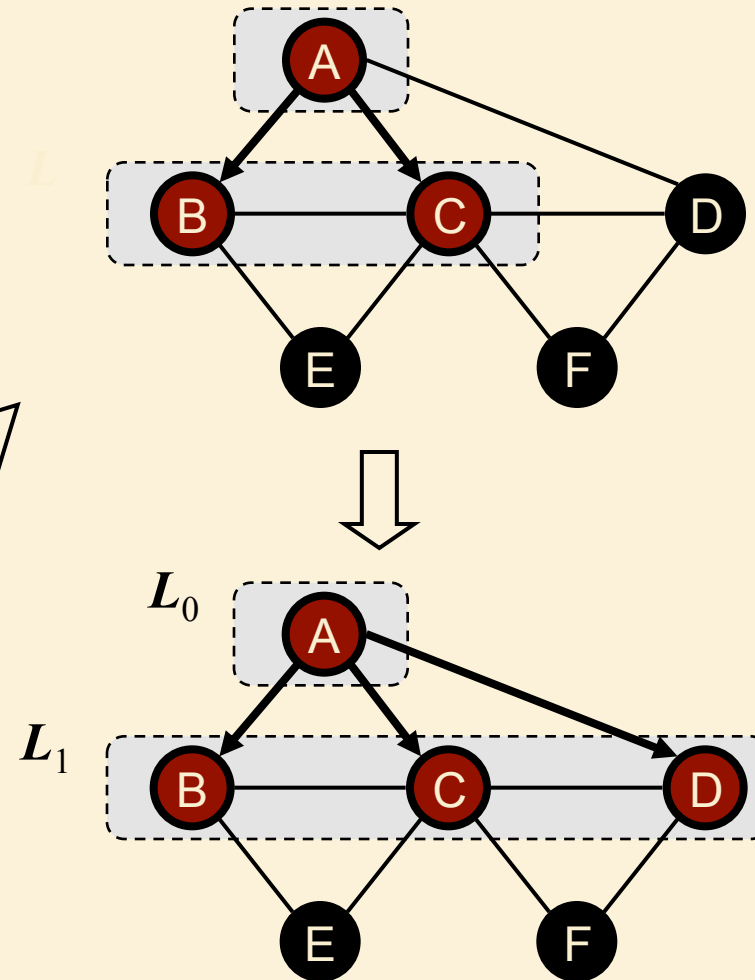
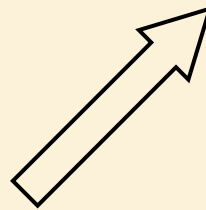
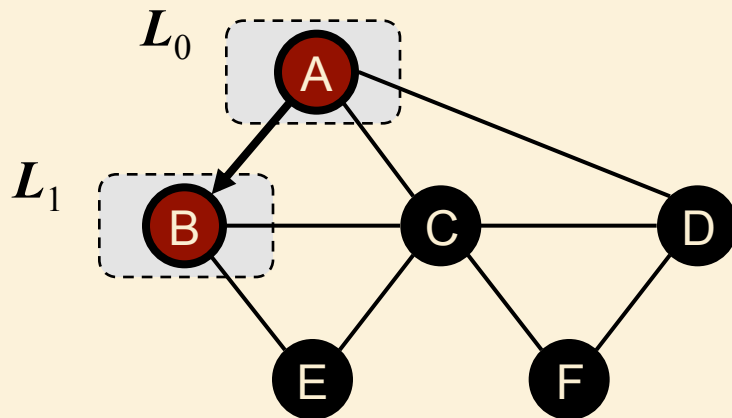
unexplored edge



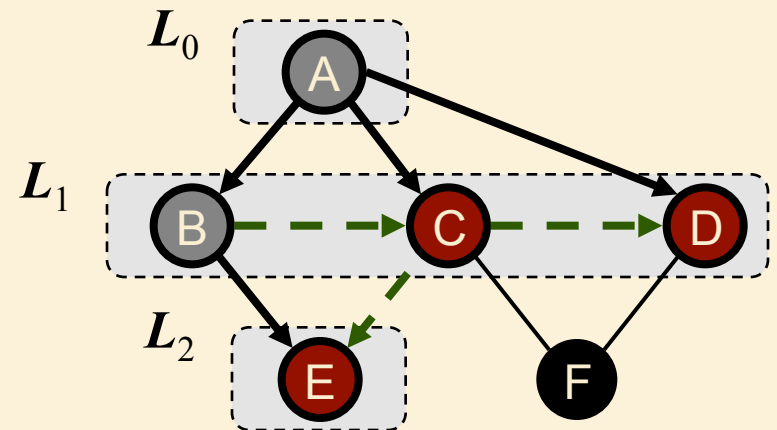
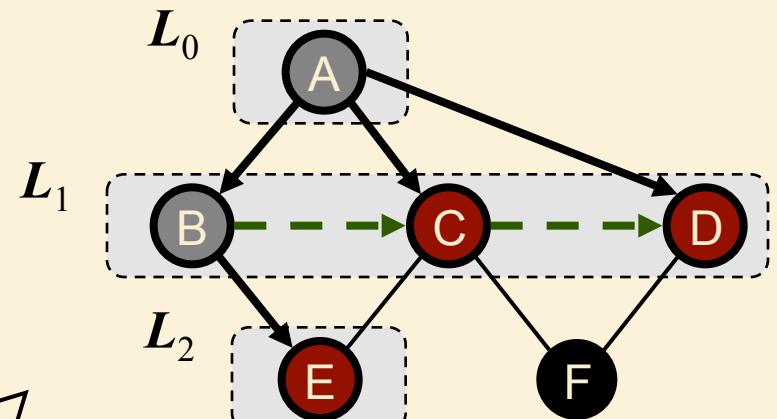
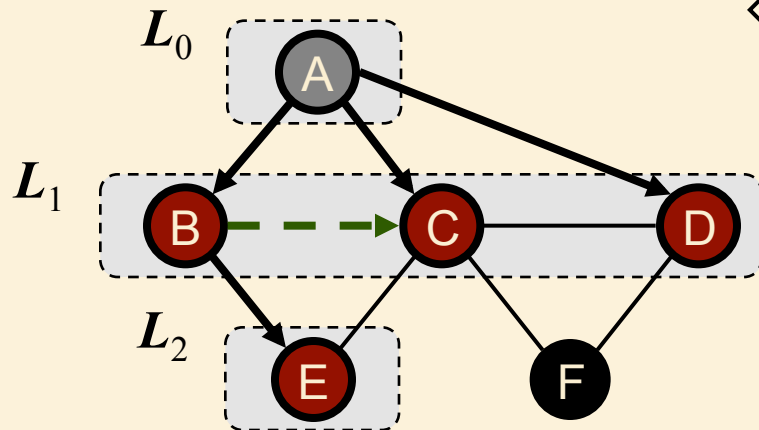
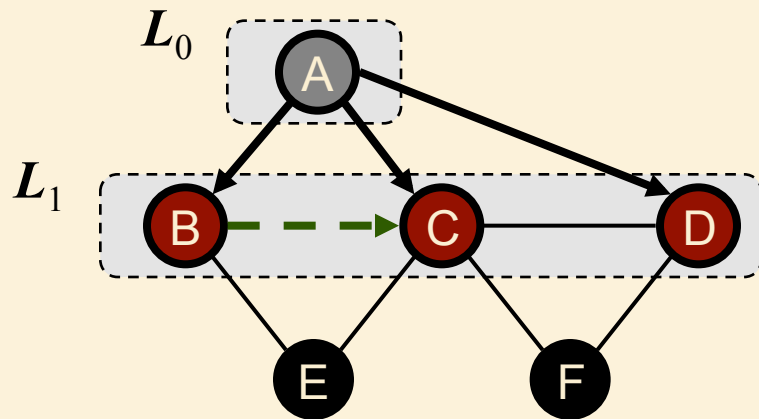
discovery edge



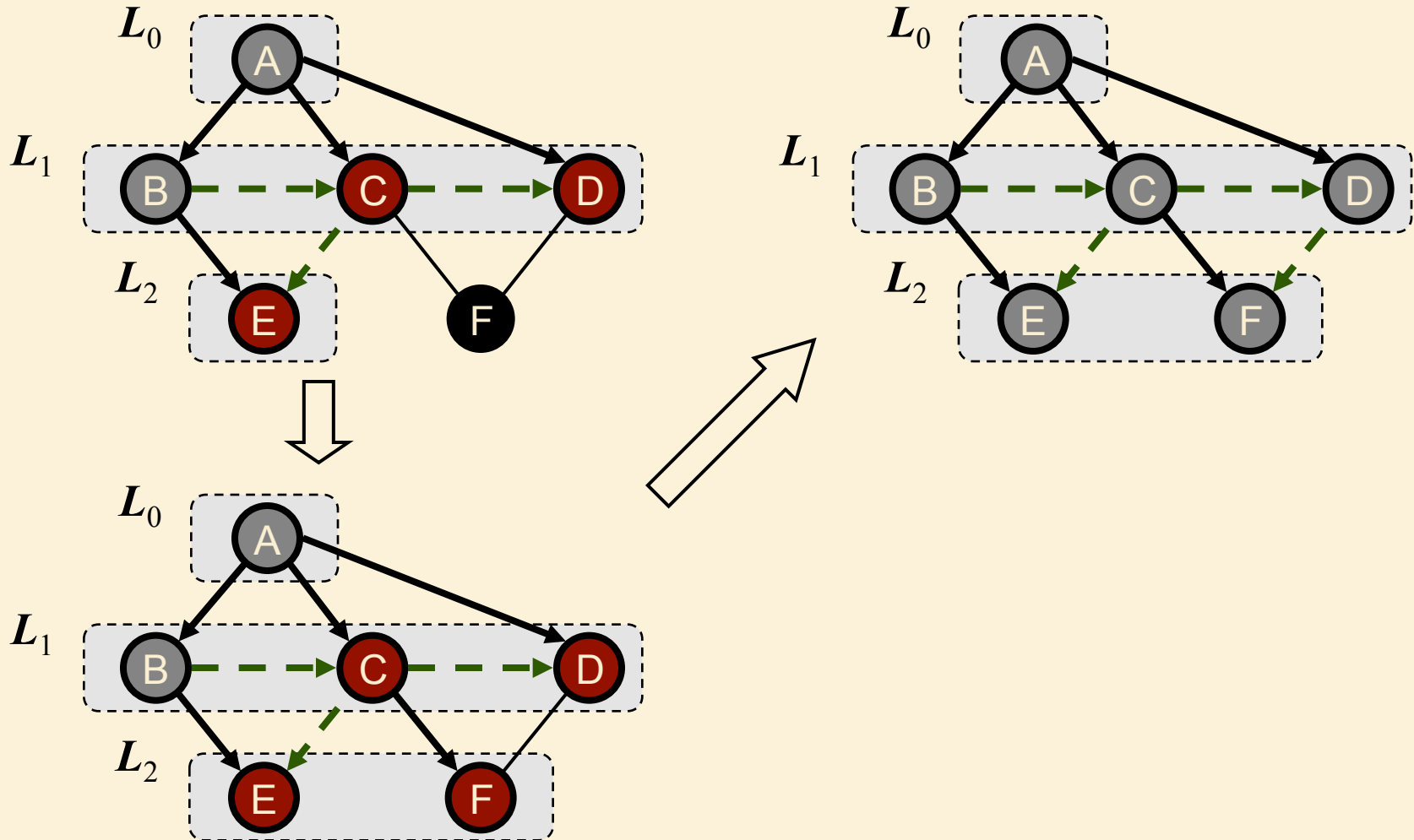
cross edge



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

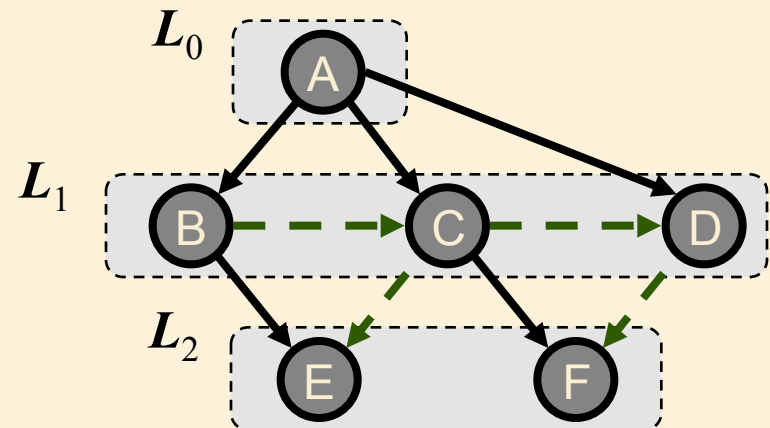
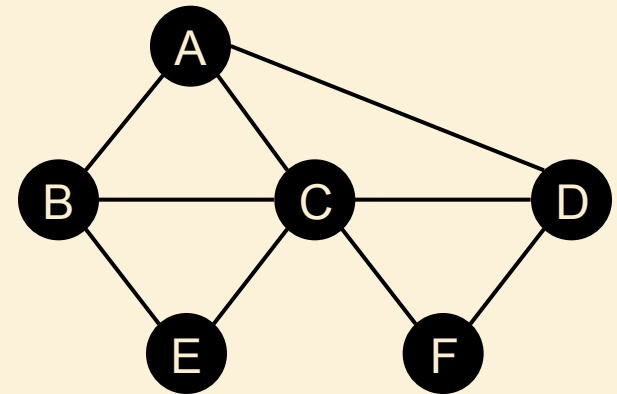
Property 2

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

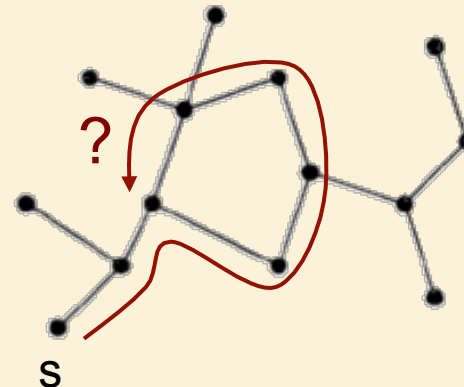
- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled three times
 - ❑ once as BLACK (undiscovered)
 - ❑ once as RED (discovered, on queue)
 - ❑ once as GRAY (finished)
- Each edge is considered twice (for an undirected graph)
- Each vertex is placed on the queue once
- Thus 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 $\delta[u]$ and

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

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

$d[u] \leftarrow \infty$

$\pi[u] \leftarrow \text{null}$

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

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

$d[s] \leftarrow 0$

$Q.\text{enqueue}(s)$

while $Q \neq \emptyset$

$u \leftarrow Q.\text{dequeue}()$

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

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

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

$d[v] \leftarrow d[u] + 1$

$\pi[v] \leftarrow u$

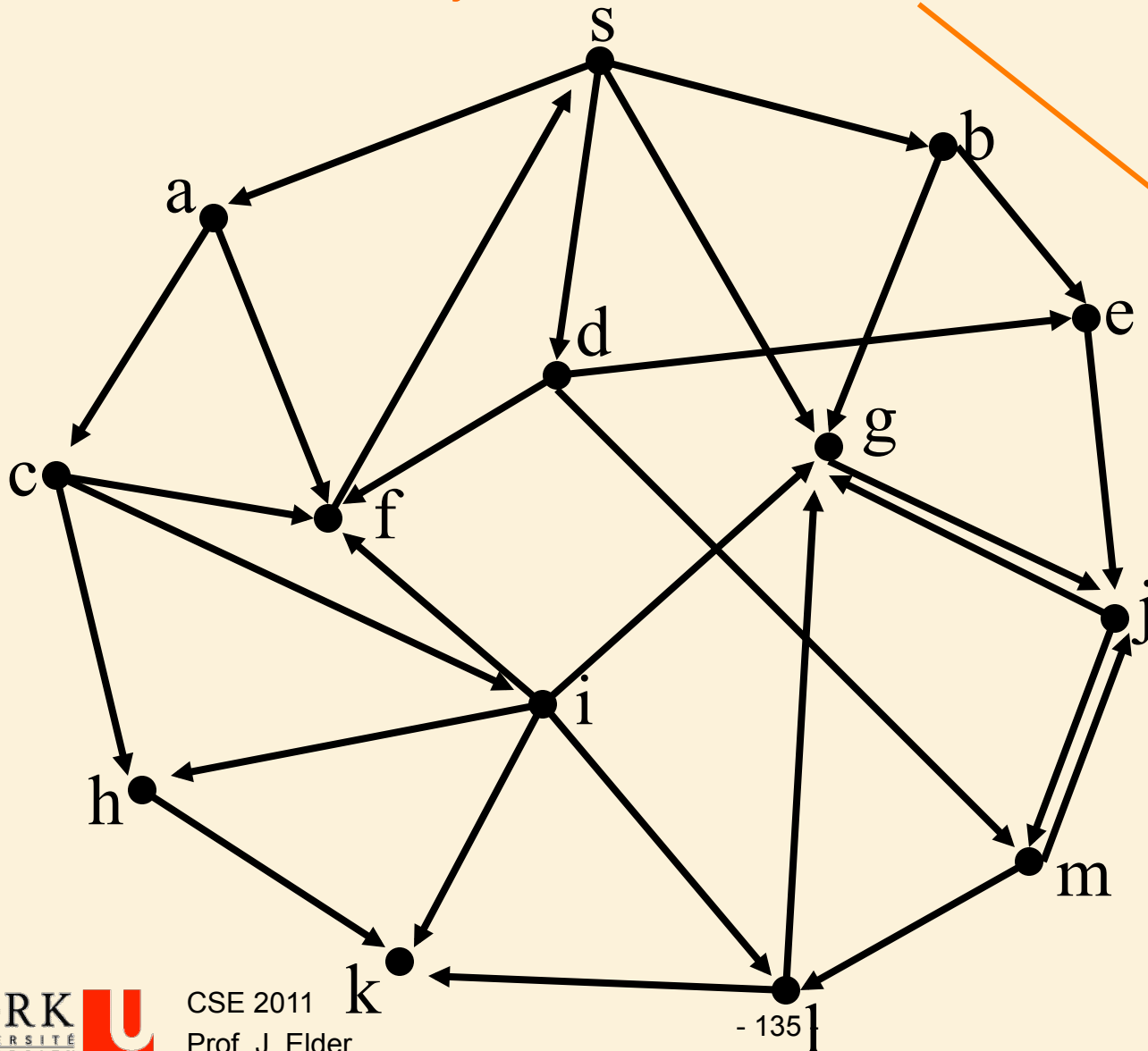
$Q.\text{enqueue}(v)$

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

BFS

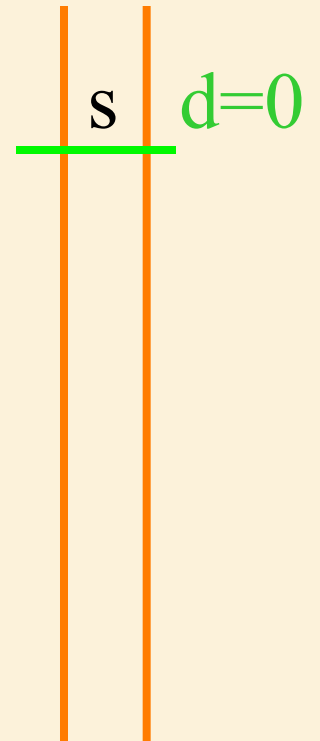
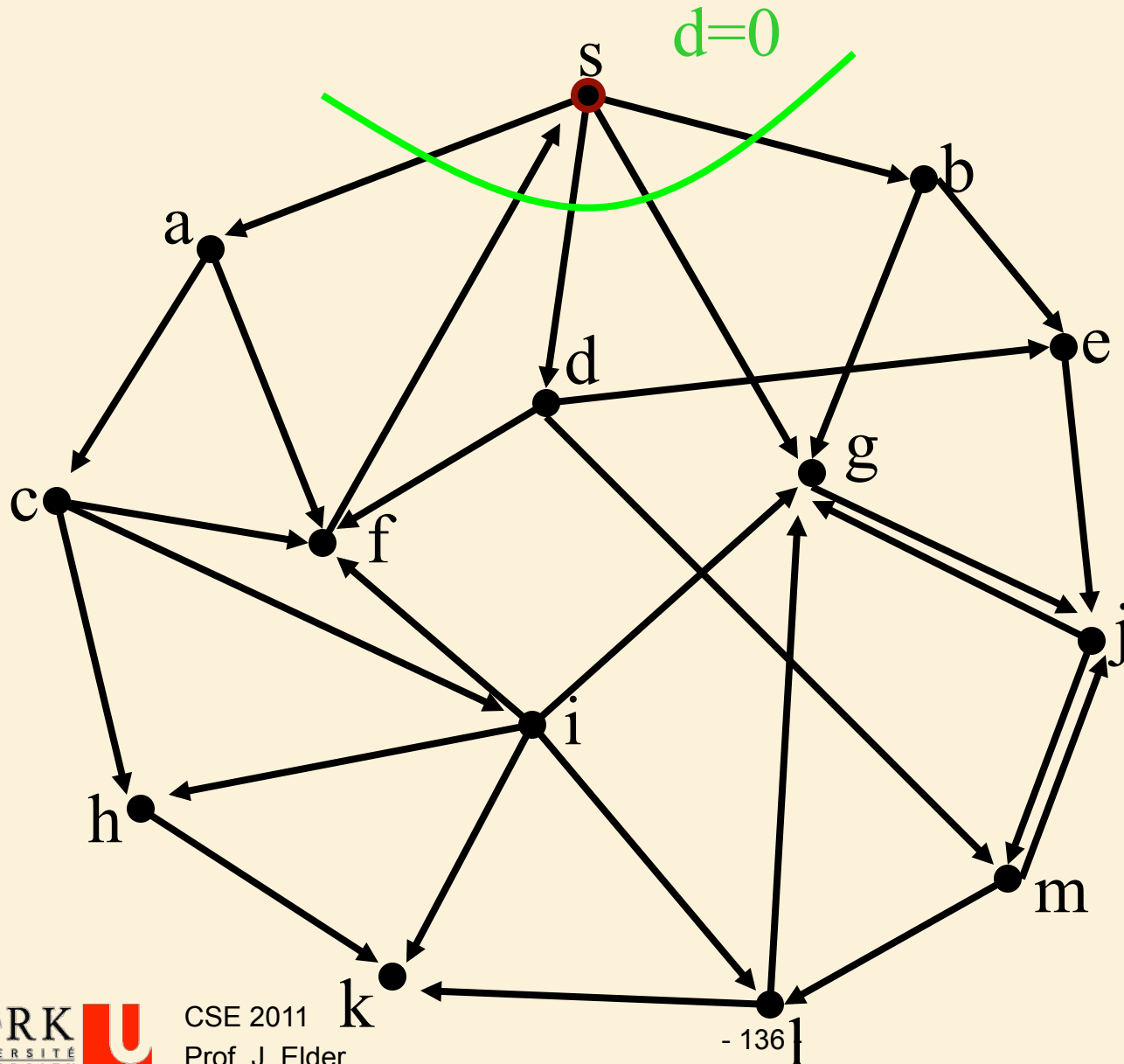
First-In First-Out (FIFO) queue
stores 'just discovered' vertices

Found
Not Handled
Queue



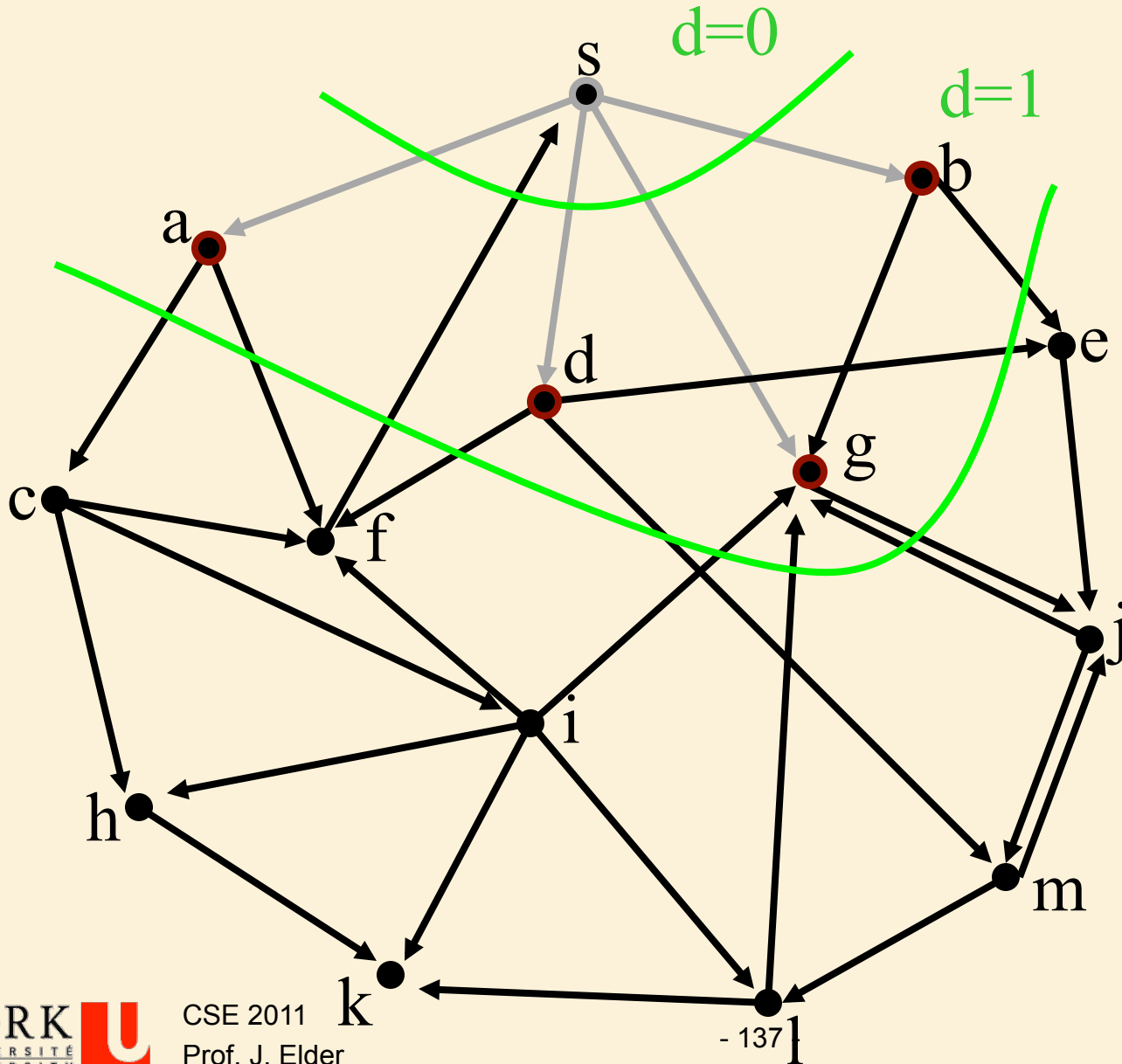
BFS

Found
Not Handled
Queue



BFS

Found
Not Handled
Queue

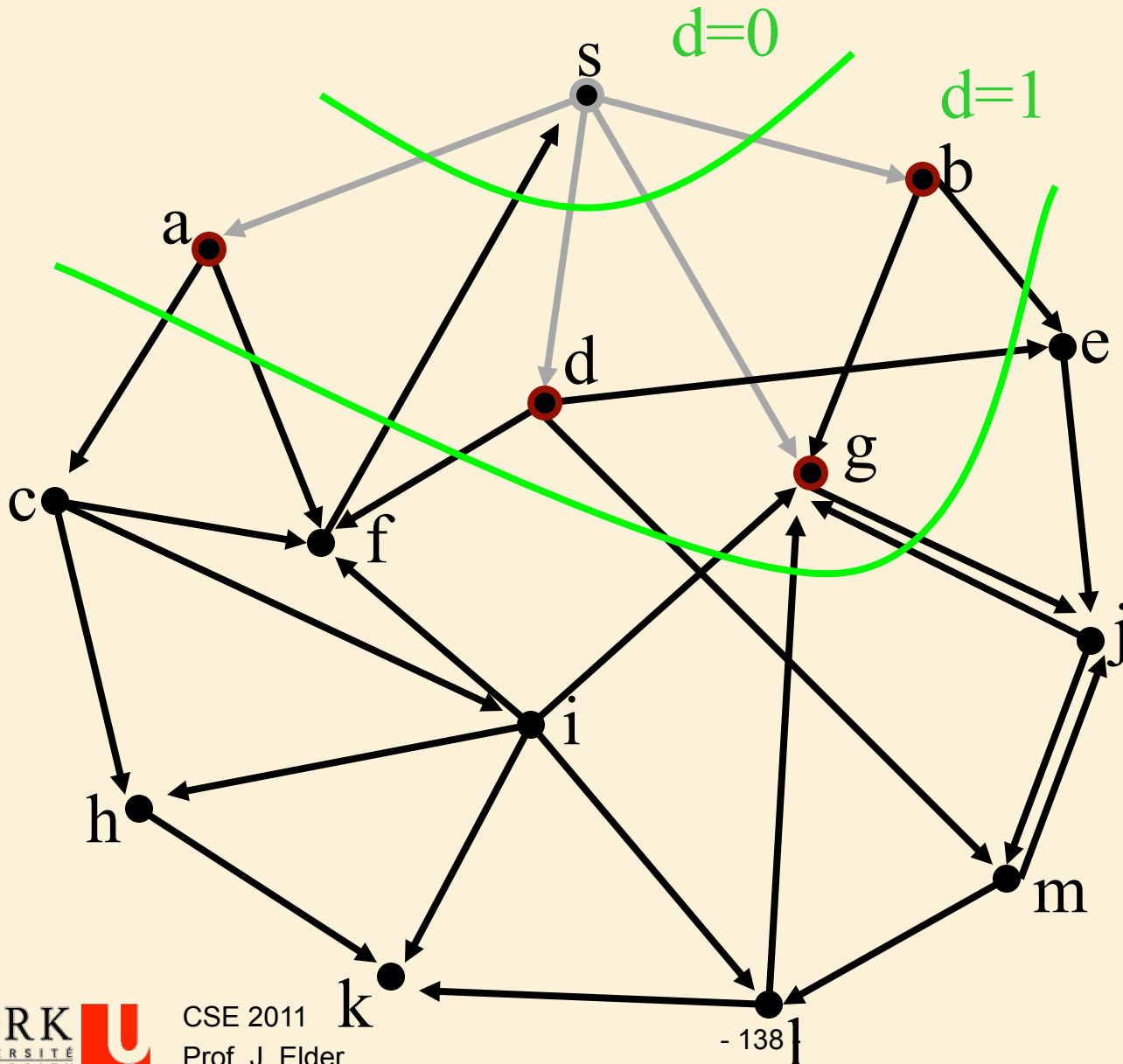


d=0
d=1

a
d
g
b

BFS

Found
Not Handled
Queue



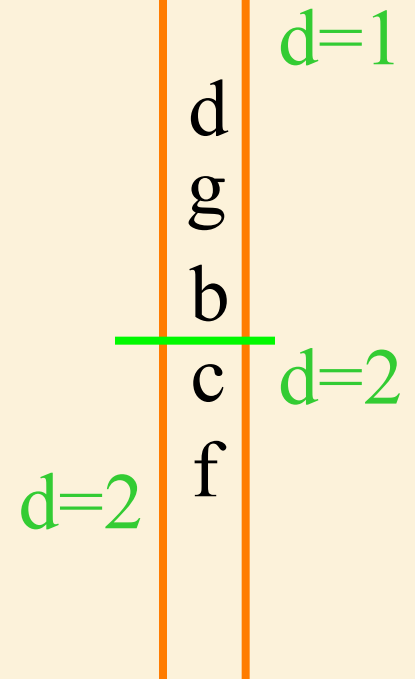
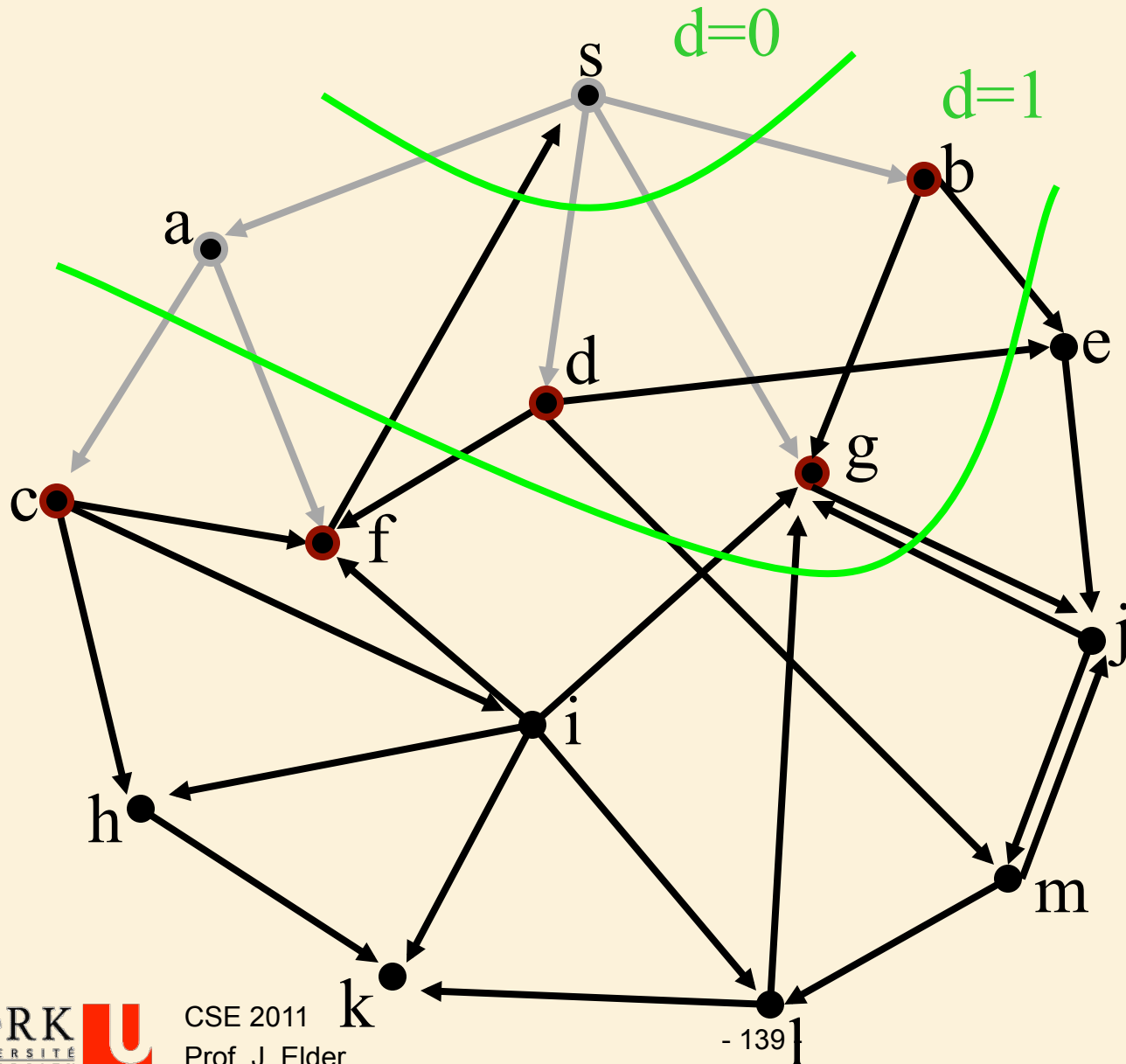
d=1

a
d
g
b

d=1

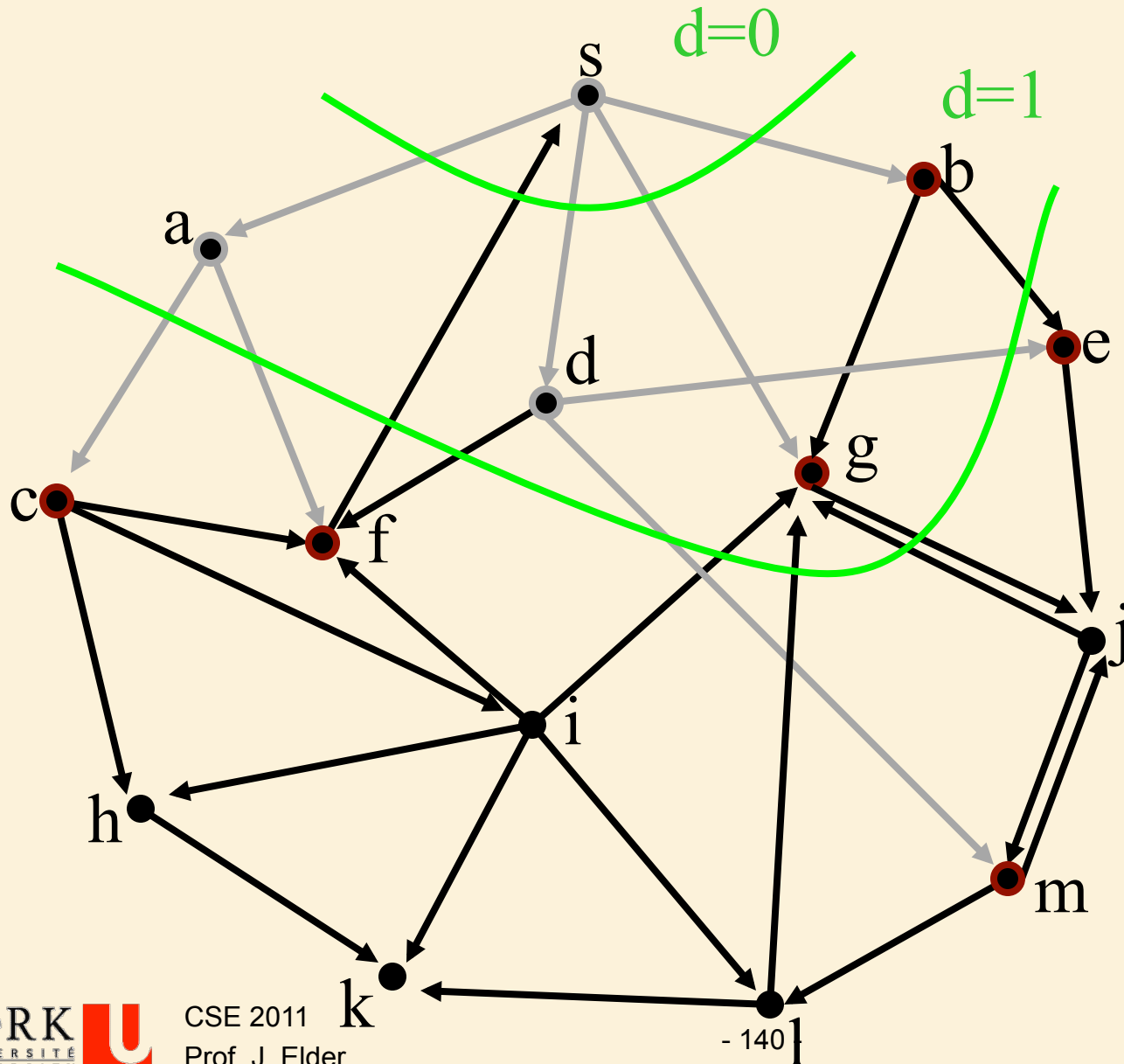
BFS

Found
Not Handled
Queue



BFS

Found
Not Handled
Queue



d=1

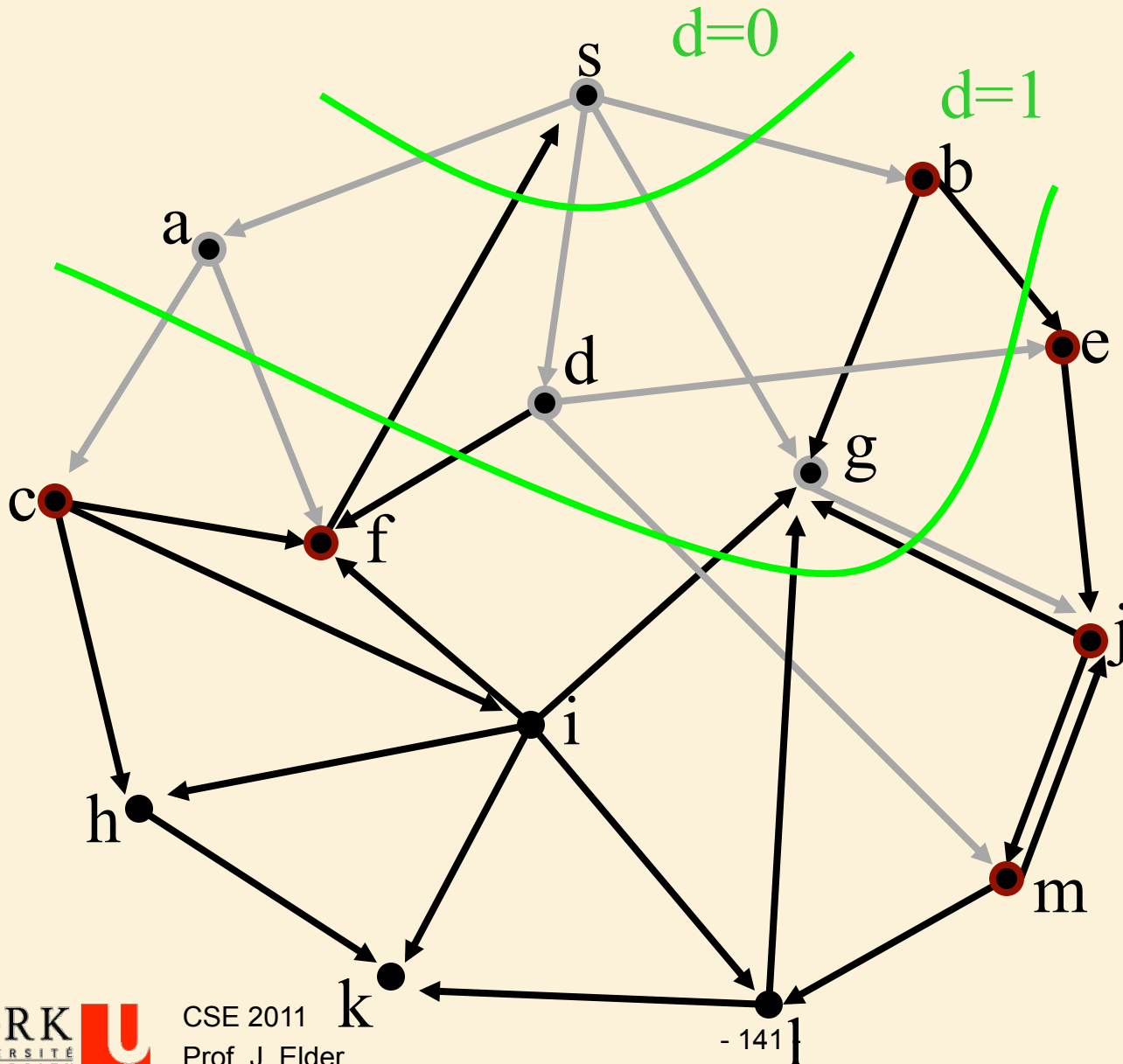
d=2

d=2

g
b
c
f
m
e

BFS

Found
Not Handled
Queue



d=1

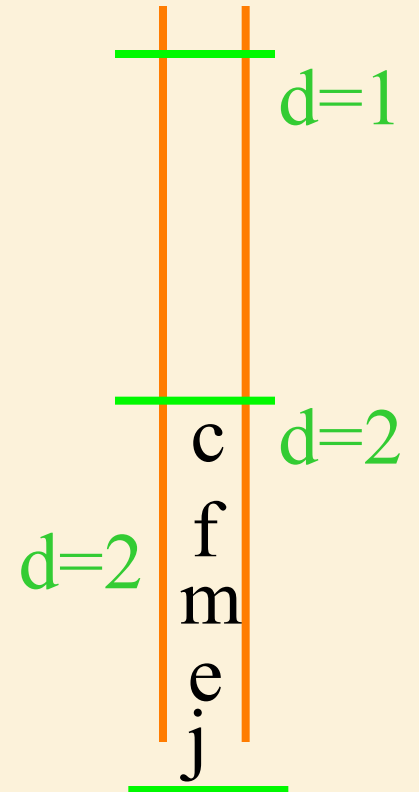
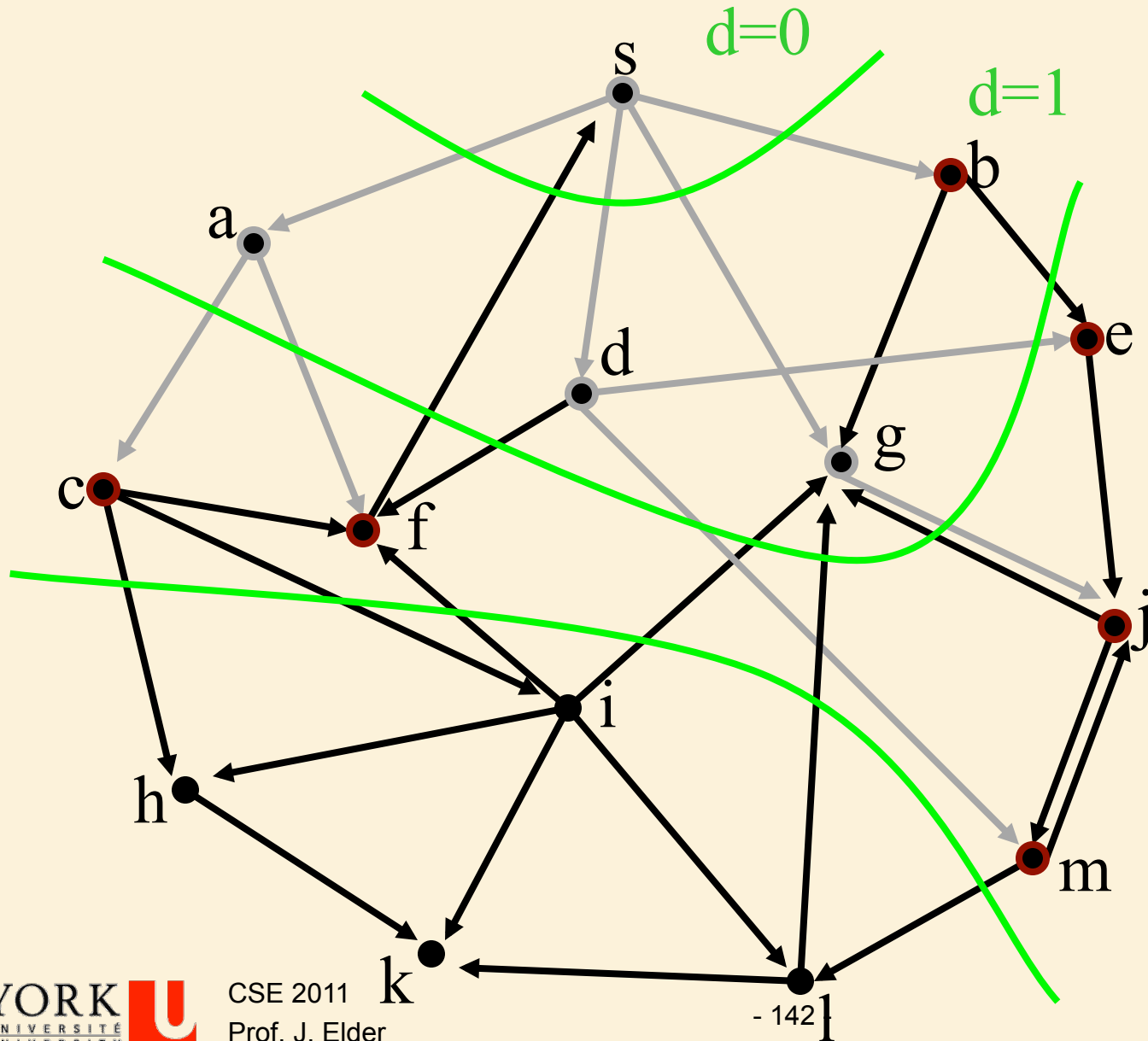
d=2

d=2

b
c
f
m
e
j

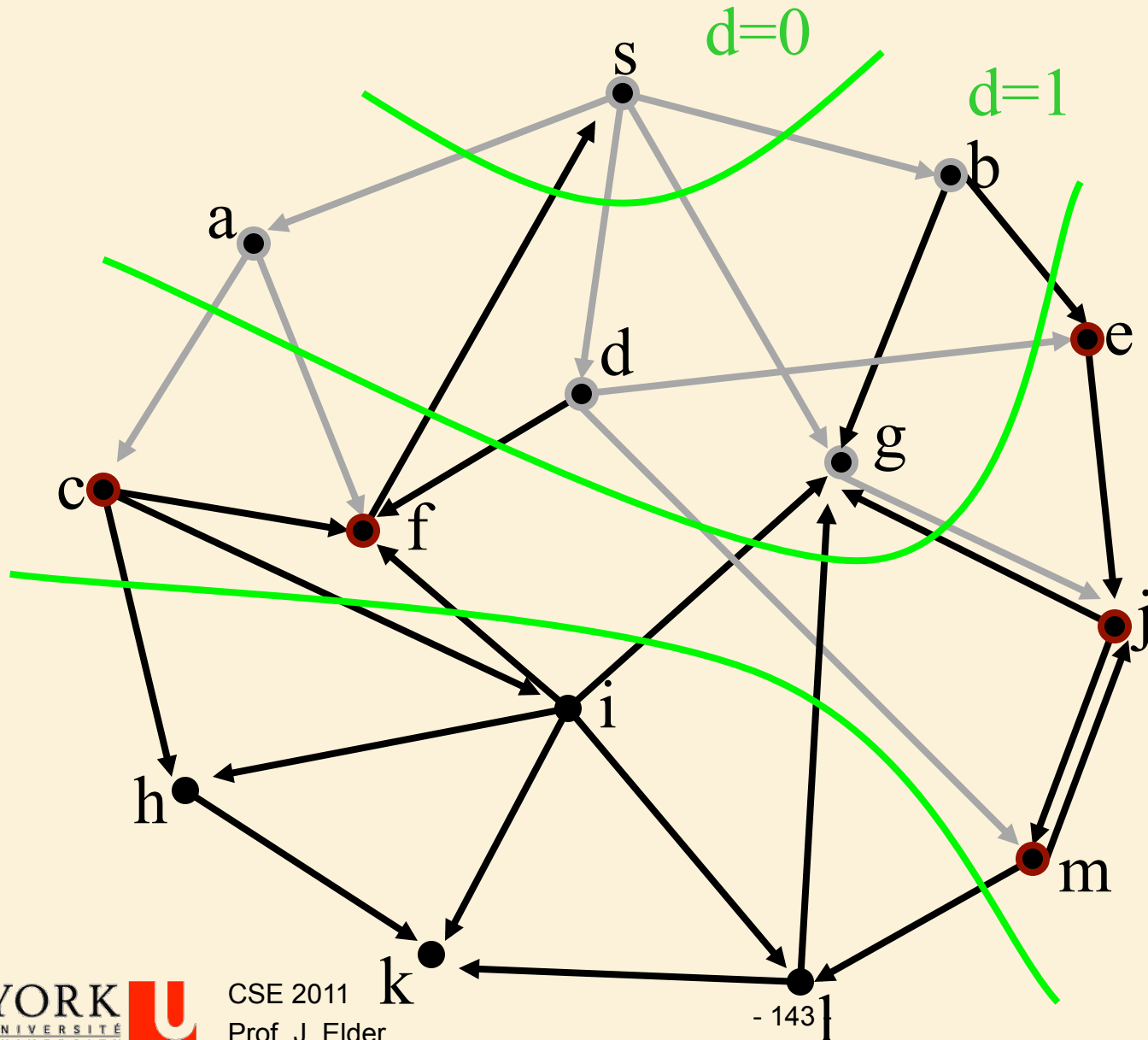
BFS

Found
Not Handled
Queue



BFS

Found
Not Handled
Queue

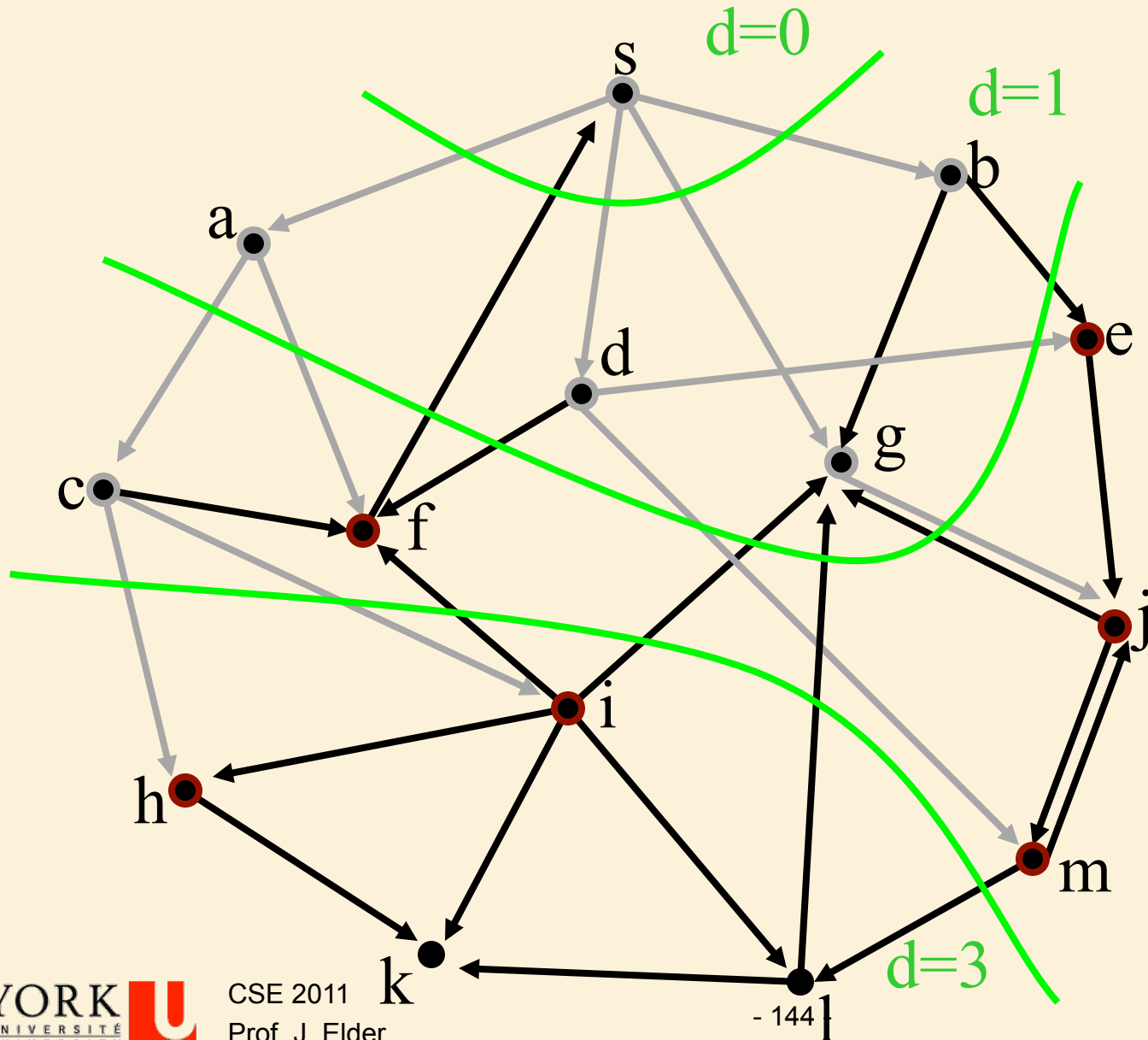


d=2

d=2

BFS

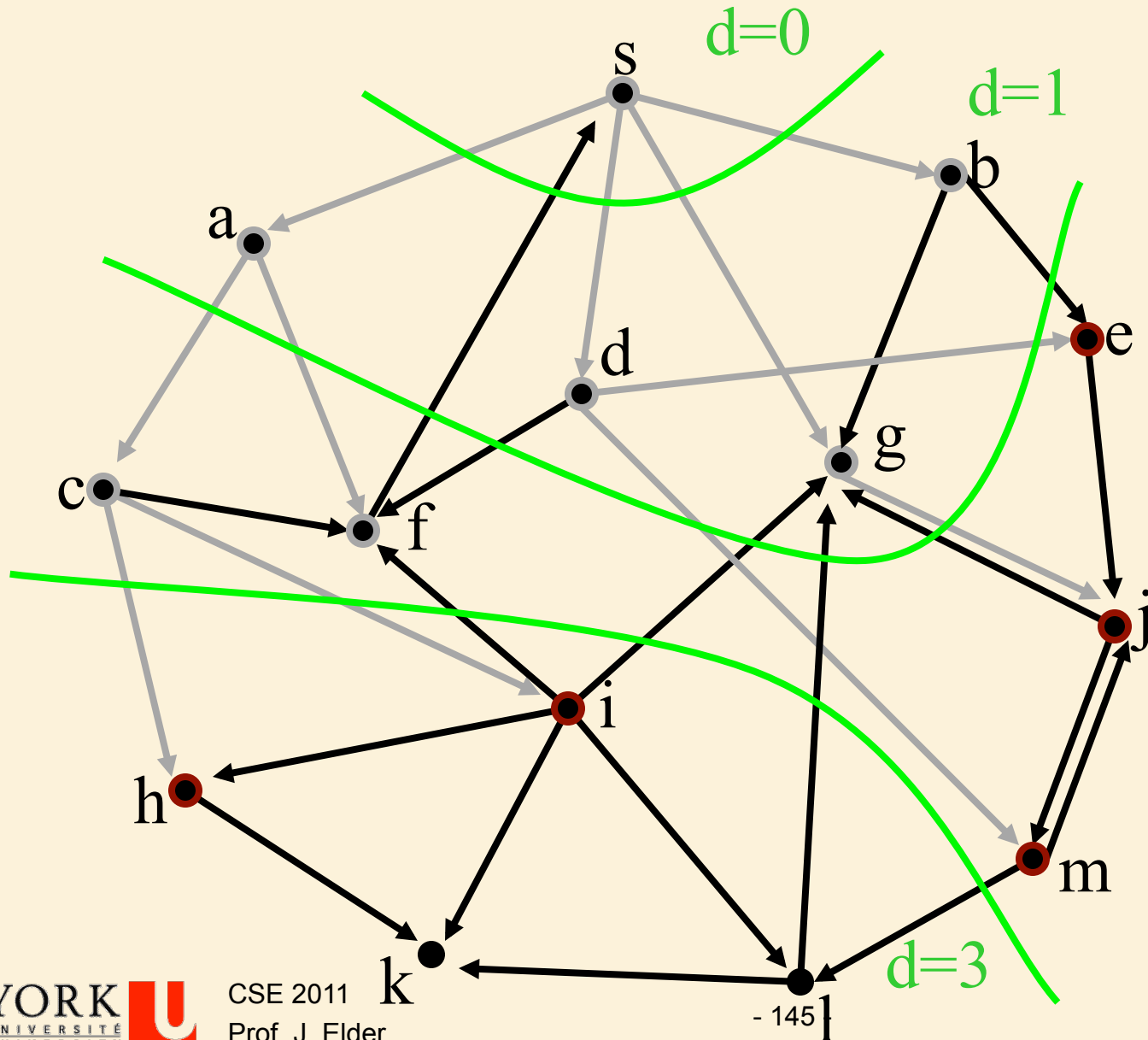
Found
Not Handled
Queue



		d=2
f	m	
e	j	
		d=3
h		
i		

BFS

Found
Not Handled
Queue



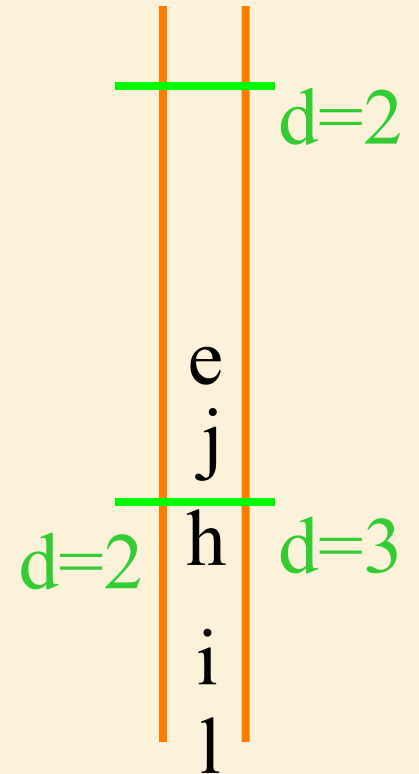
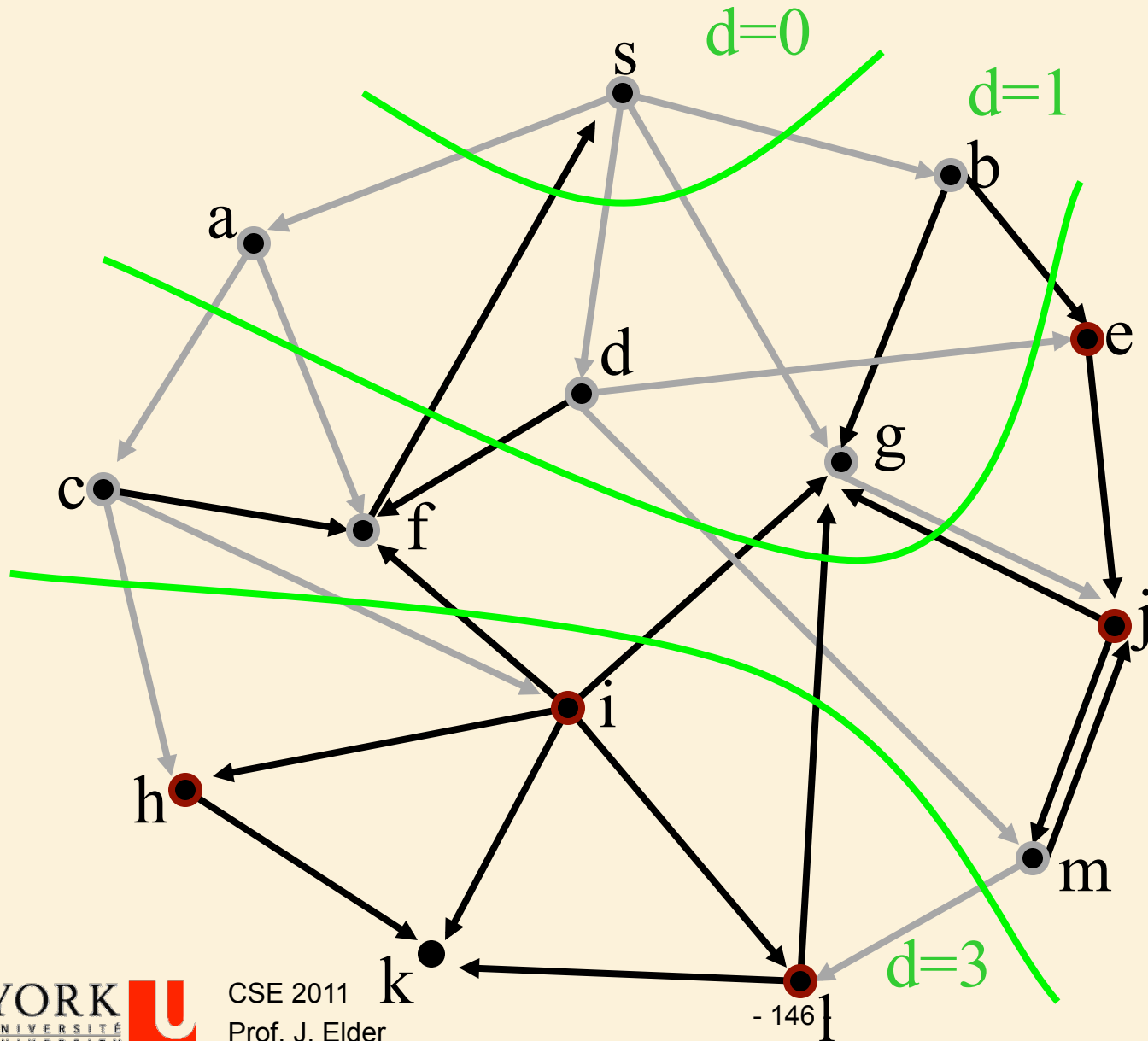
d=2

d=2 d=3

d=3

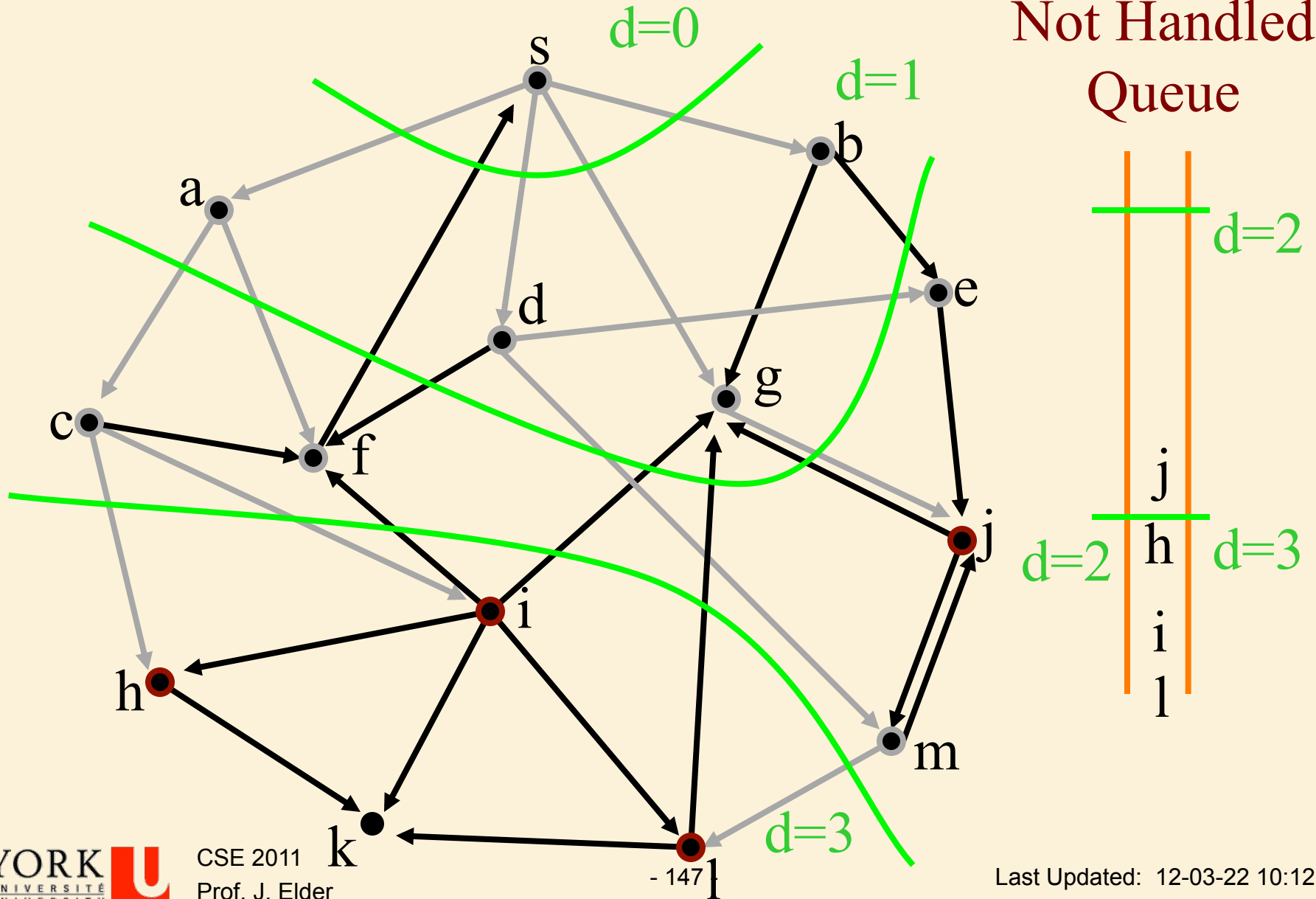
BFS

Found
Not Handled
Queue



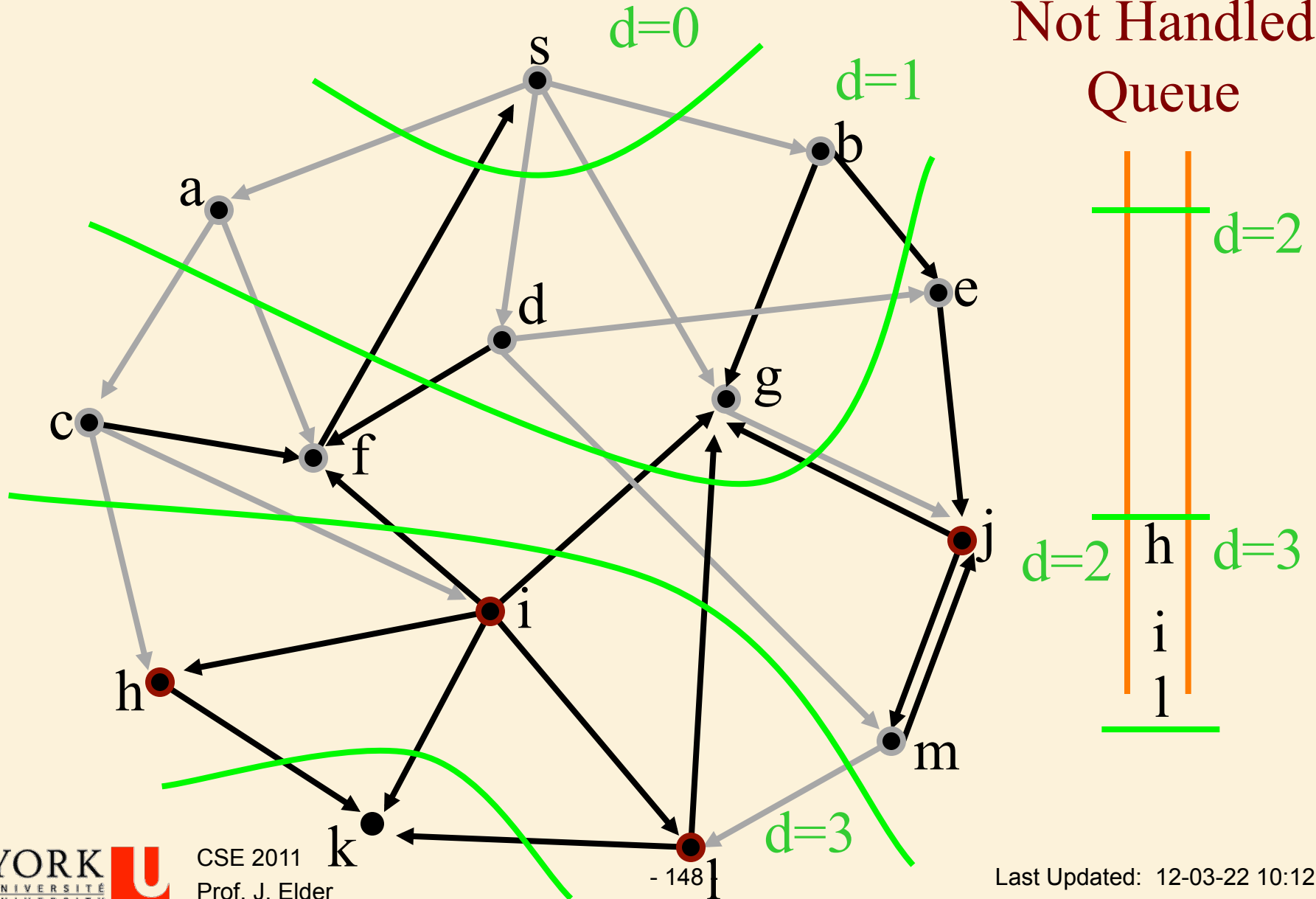
BFS

Found
Not Handled
Queue



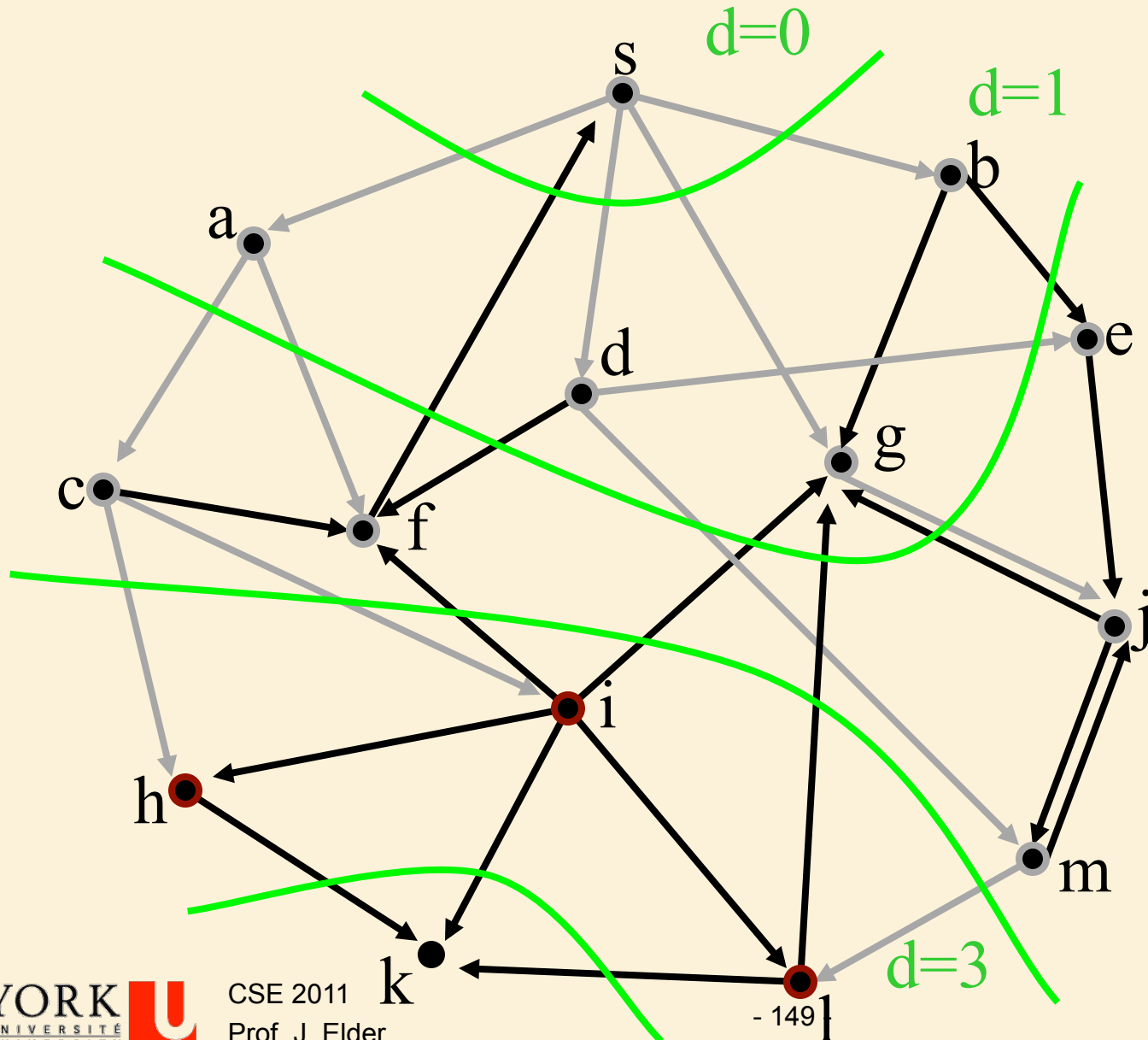
BFS

Found
Not Handled
Queue



BFS

Found
Not Handled
Queue



$d=2$

$d=3$

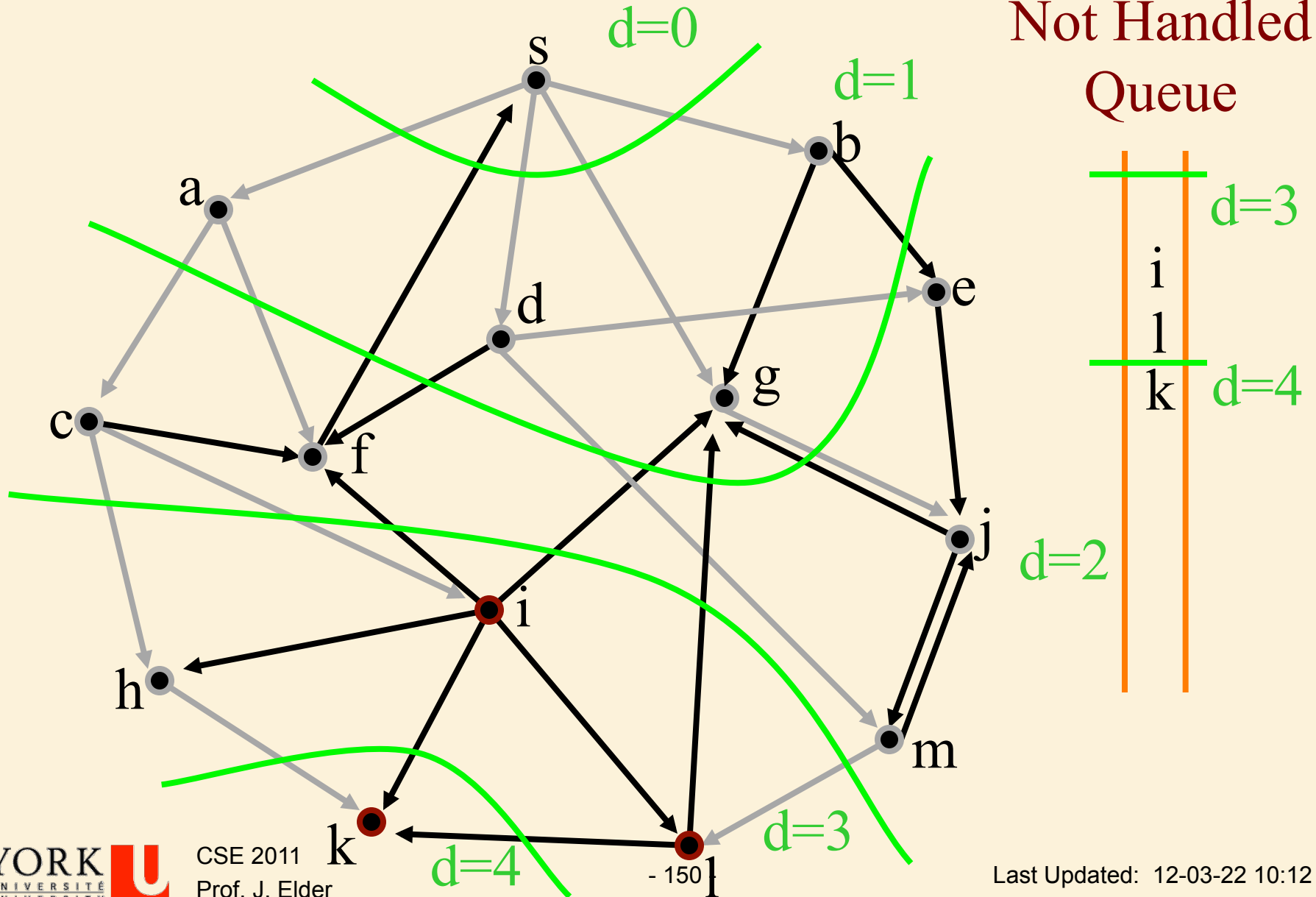
$d=3$

- 149

Last Updated: 12-03-22 10:12 AM

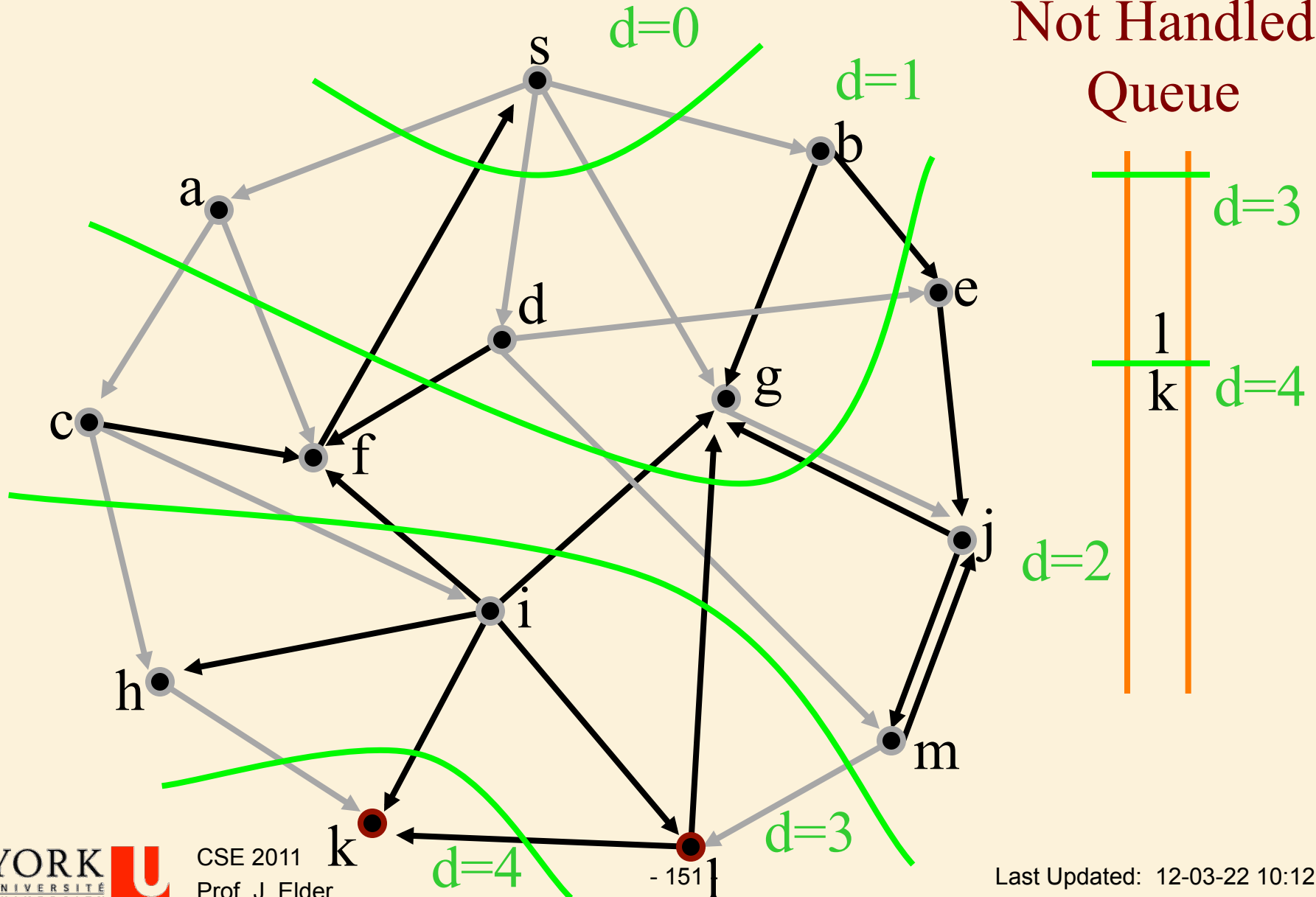
BFS

Found
Not Handled
Queue



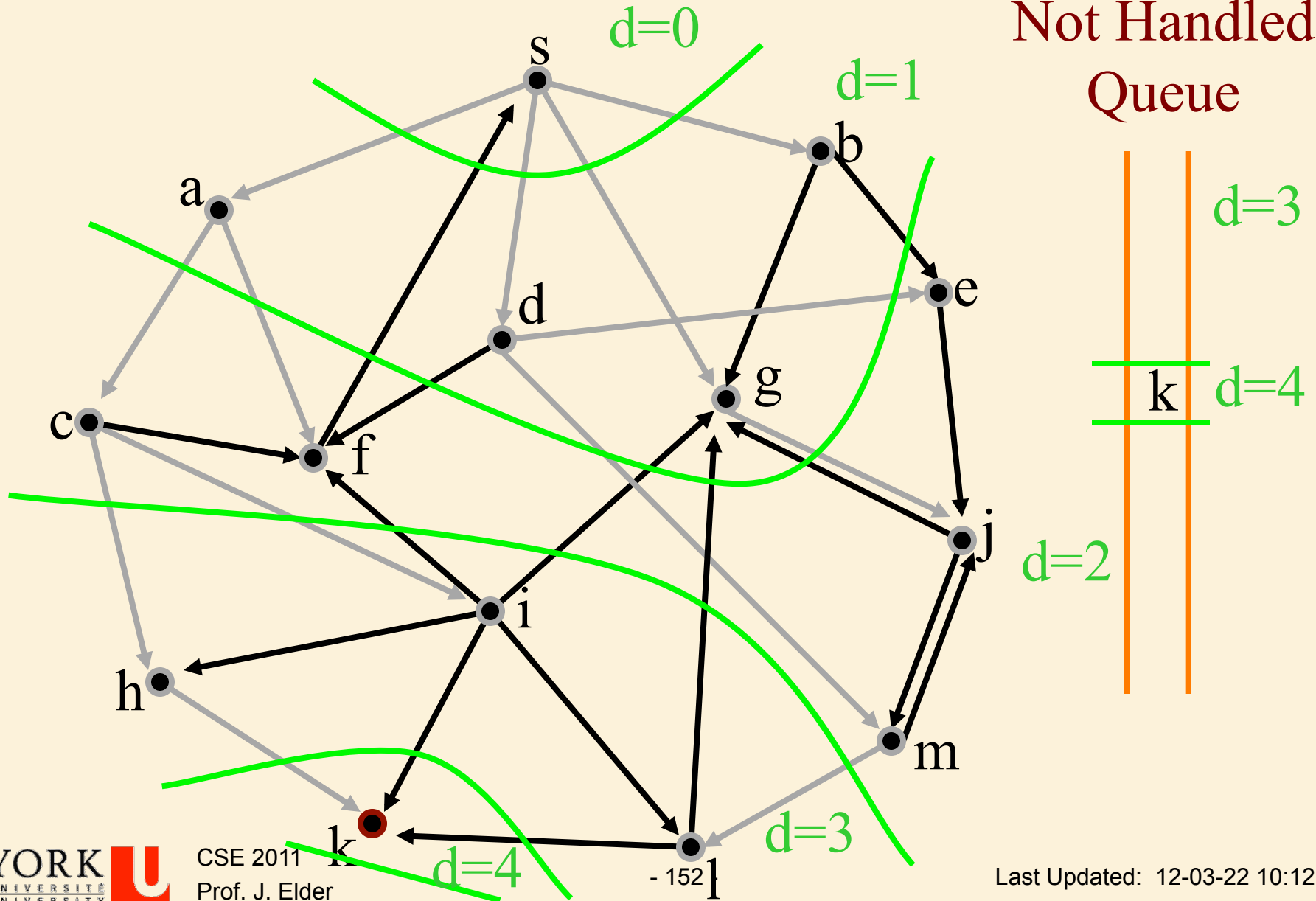
BFS

Found
Not Handled
Queue



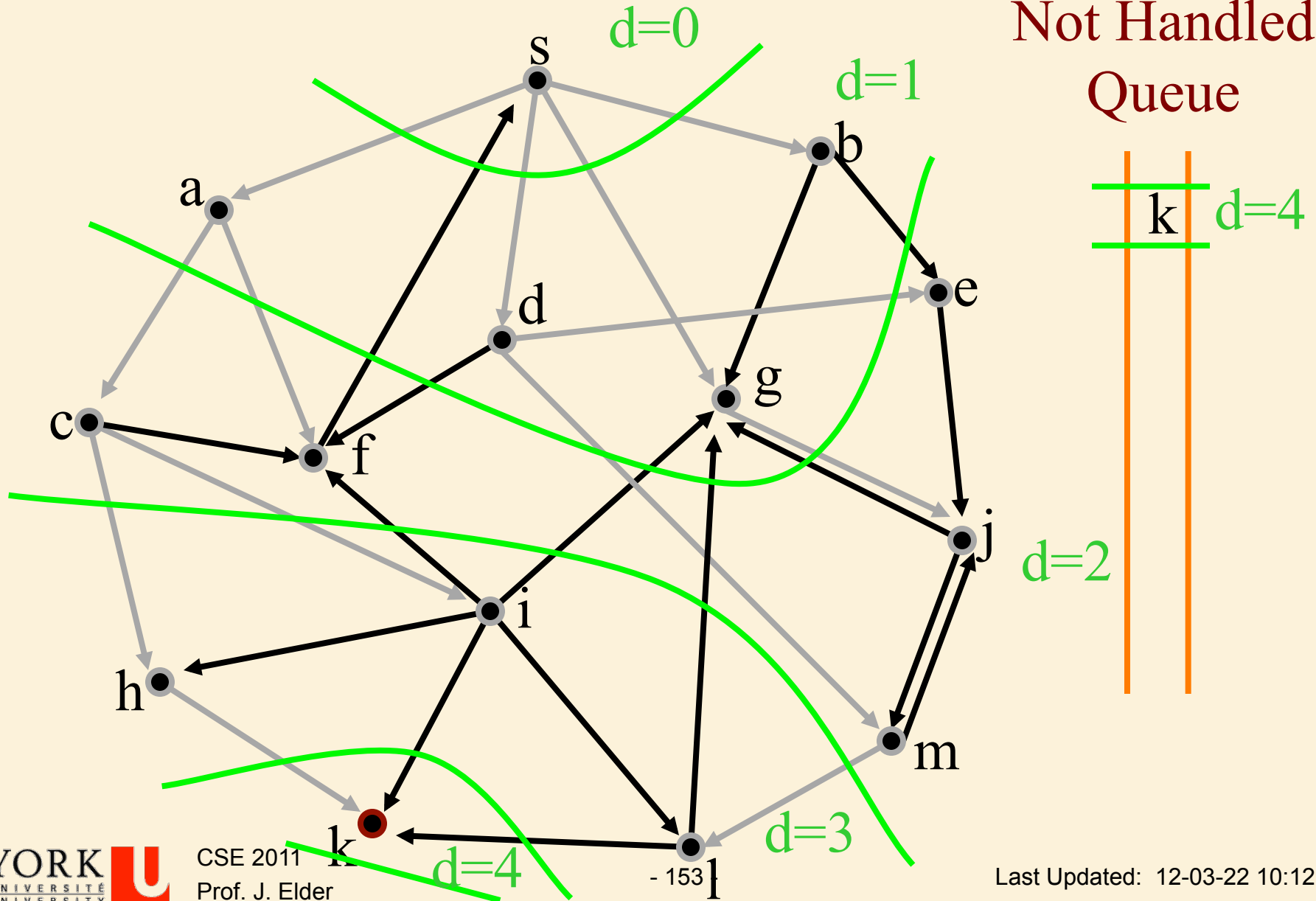
BFS

Found Not Handled Queue



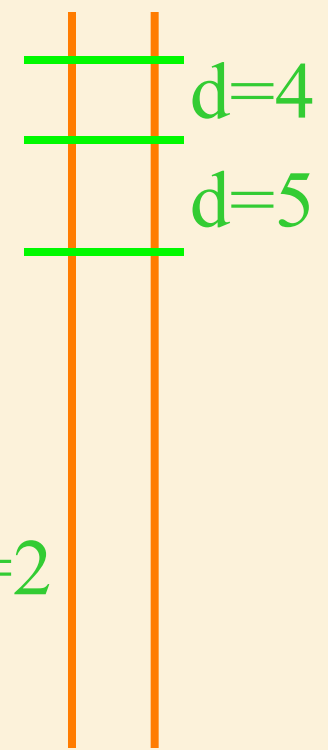
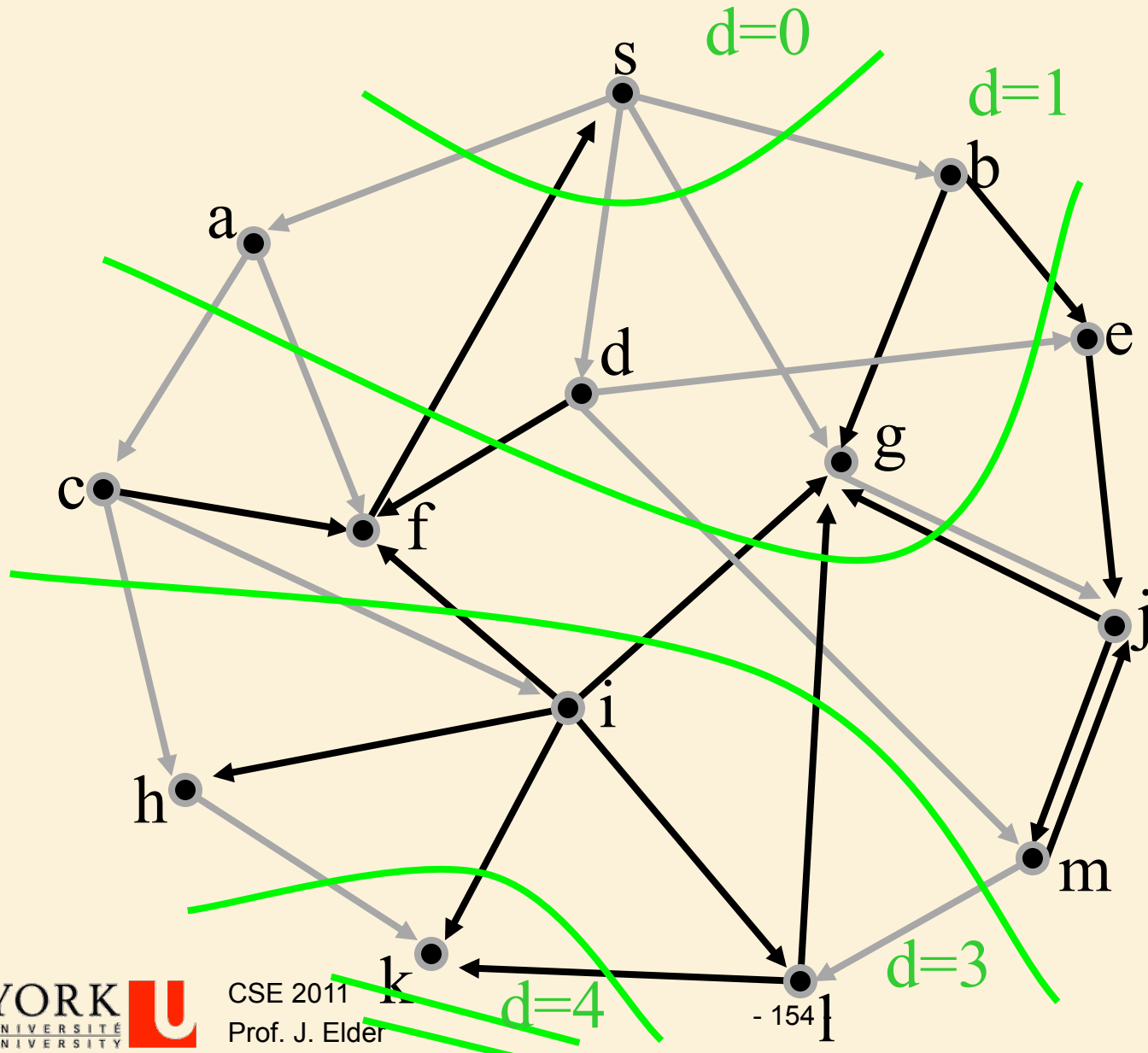
BFS

Found
Not Handled
Queue



BFS

Found
Not Handled
Queue



d=2

d=3

d=4

- 154

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 $u \in V[G]$

$d[u] \leftarrow \infty$

$\pi[u] \leftarrow \text{null}$

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

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

$d[s] \leftarrow 0$

$Q.\text{enqueue}(s)$

while $Q \neq \emptyset$

$u \leftarrow Q.\text{dequeue}()$

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

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

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

$d[v] \leftarrow d[u] + 1$

$\pi[v] \leftarrow u$

$Q.\text{enqueue}(v)$

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

➤ Q is a FIFO queue.

➤ Each vertex assigned finite d value at most once.

➤ Q contains vertices with d values $\{i, \dots, i, i+1, \dots, i+1\}$

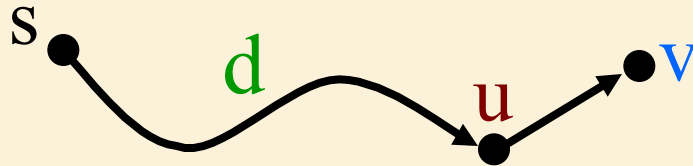
➤ d values assigned are monotonically increasing over time.

Breadth-First-Search is Greedy

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

Correctness

Basic Steps:

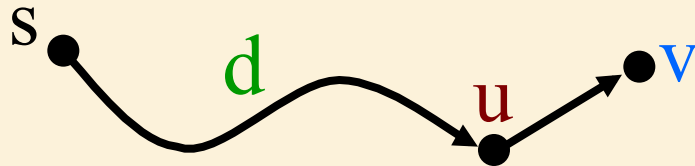


The shortest path to u has length d & there is an edge from u to v

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

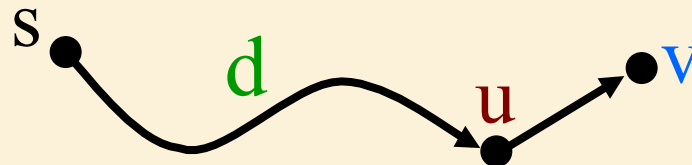
Correctness: Basic Intuition

- When we discover v , how do we know there is not a shorter path to v ?
 - Because if there was, we would already have discovered it!



Correctness: More Complete Explanation

- Vertices are discovered in order of their distance from the source vertex s .
- Suppose that at time t_1 we have discovered the set V_d of all vertices that are a distance of d from s .
- Each vertex in the set V_{d+1} of all vertices a distance of $d+1$ from s must be adjacent to a vertex in V_d .
- Thus we can correctly label these vertices by visiting all vertices in the adjacency lists of vertices in V_d .



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

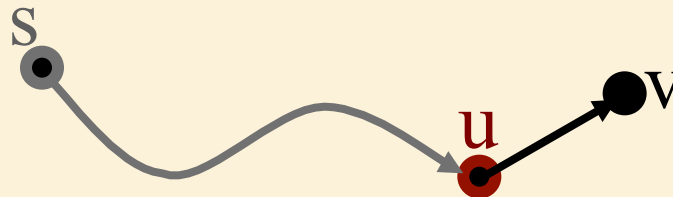
Claim 1. d is never too small: $d[v] \geq \delta(s, v) \forall v \in V$

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

By Induction:

Suppose it is true for all vertices thus far discovered (red and grey).
 v is discovered from some adjacent vertex u being handled.

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



since each vertex v is assigned a d value exactly once,
it follows that on exit, $d[v] \geq \delta(s, v) \forall v \in V$.

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

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

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 $u \in V[G]$

$d[u] \leftarrow \infty$

$\pi[u] \leftarrow \text{null}$

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

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

$d[s] \leftarrow 0$

$Q.\text{enqueue}(s)$

while $Q \neq \emptyset$

$u \leftarrow Q.\text{dequeue}()$

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

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

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

$d[v] \leftarrow d[u] + 1$

$\pi[v] \leftarrow u$

$Q.\text{enqueue}(v)$

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



$\leftarrow \langle LI \rangle$: $d[v] \geq \delta(s, v) \forall$ 'discovered' (red or grey) $v \in V$

$d[v] \leftarrow d[u] + 1 \geq \delta(s, u) + 1 \geq \delta(s, v)$

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

Proof by contradiction:

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

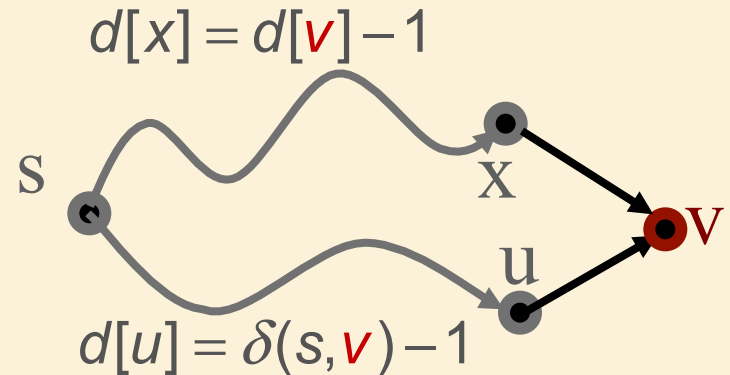
Let v be the vertex with minimum $\delta(s, v)$ that receives such a d value.

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

$$\begin{aligned}\delta(s, v) &< d[v] \\ \rightarrow \delta(s, v) - 1 &< d[v] - 1 \\ \rightarrow d[u] &< d[x]\end{aligned}$$



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

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

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

$d[u] \leftarrow \infty$

$\pi[u] \leftarrow \text{null}$

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

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

$d[s] \leftarrow 0$

$Q.\text{enqueue}(s)$

while $Q \neq \emptyset$

$u \leftarrow Q.\text{dequeue}()$

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

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

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

$d[v] \leftarrow d[u] + 1$

$\pi[v] \leftarrow u$

$Q.\text{enqueue}(v)$

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



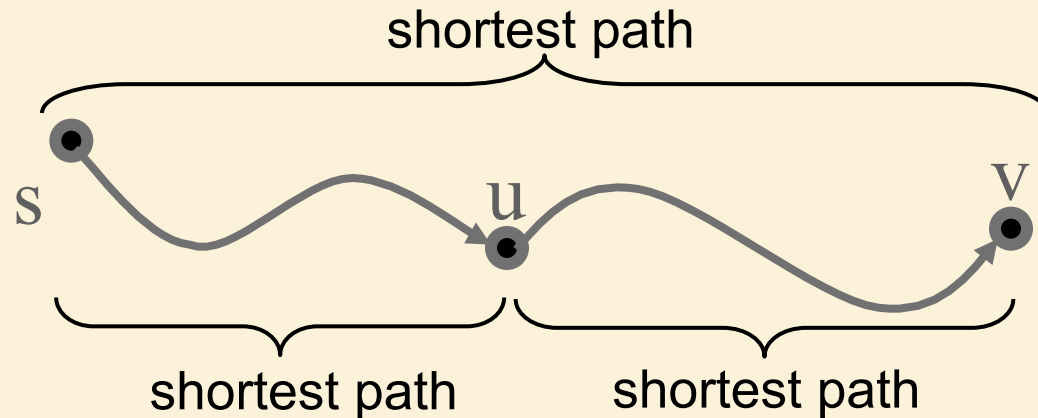
End of Lecture

March 27, 2012

Optimal Substructure Property

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

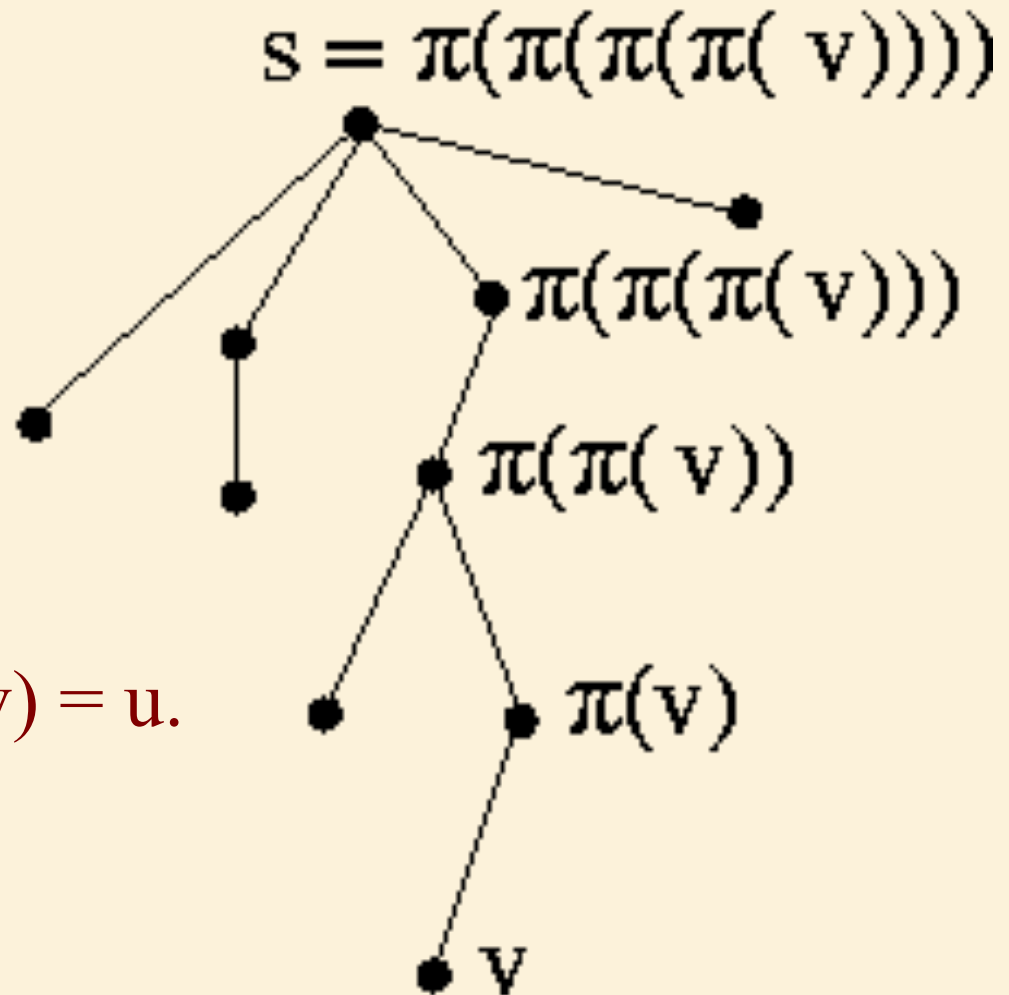
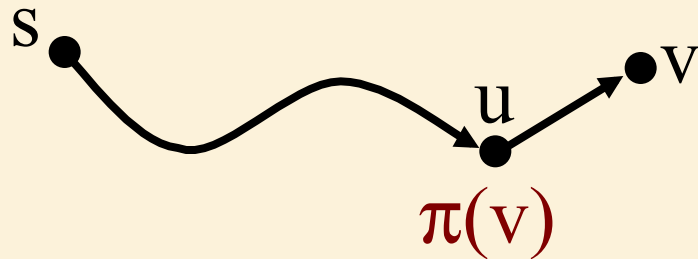
How would we prove this?



- 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 $\pi(v)$.



Predecessor of v is $\pi(v) = u$.

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

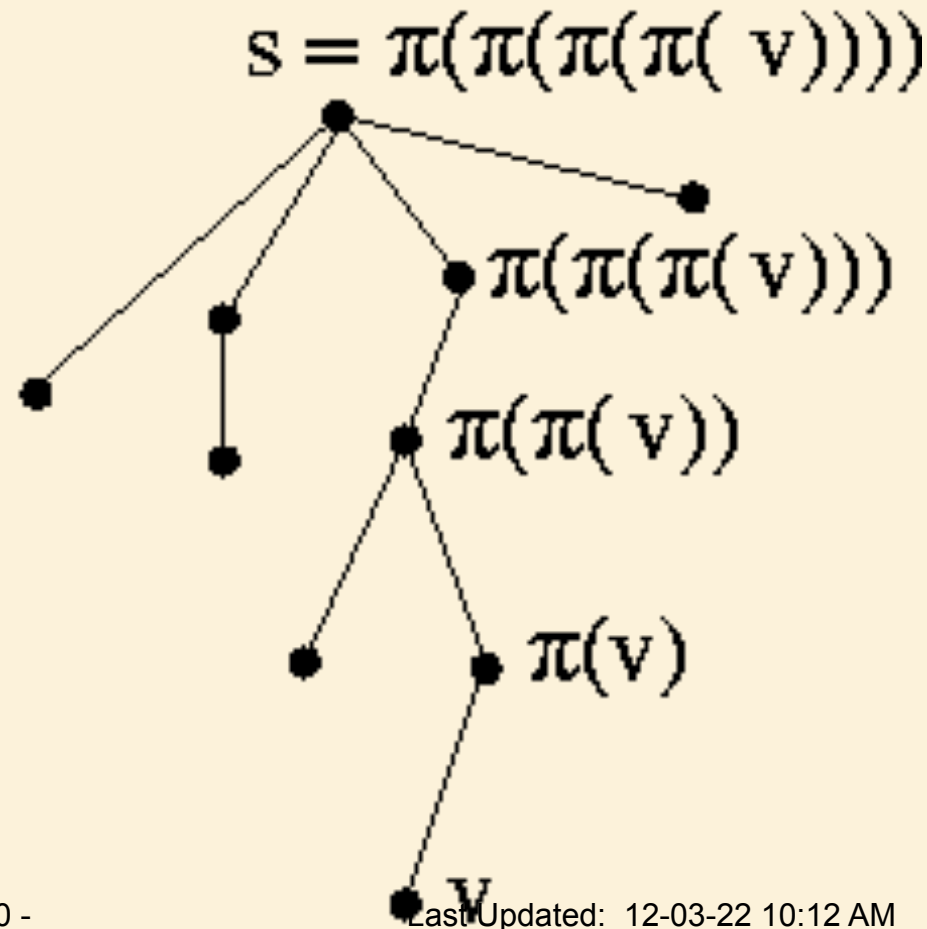
else if $\pi[v] = \text{NIL}$ then

 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 $u \in V[G]$

$d[u] \leftarrow \infty$

$\pi[u] \leftarrow \text{null}$

$d[s] \leftarrow 0$

Q.enqueue(s)

while $Q \neq \emptyset$

$u \leftarrow \text{Q.dequeue}()$

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

if $d[v] = \infty$

$d[v] \leftarrow d[u] + 1$

$\pi[v] \leftarrow u$

Q.enqueue(v)

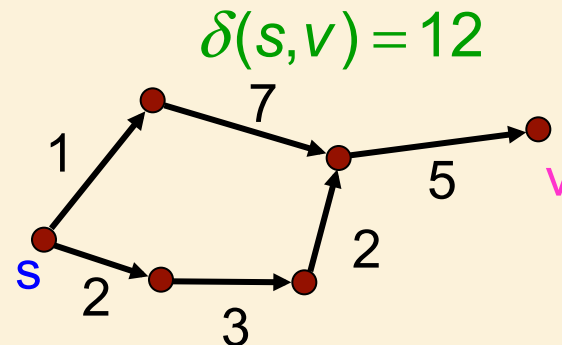
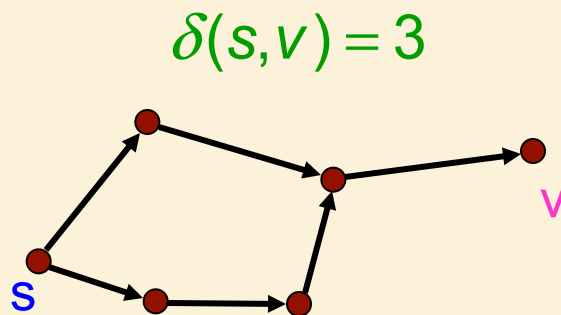
End of Lecture & End of Course

March 29, 2012

Single-Source (Weighted) Shortest Paths

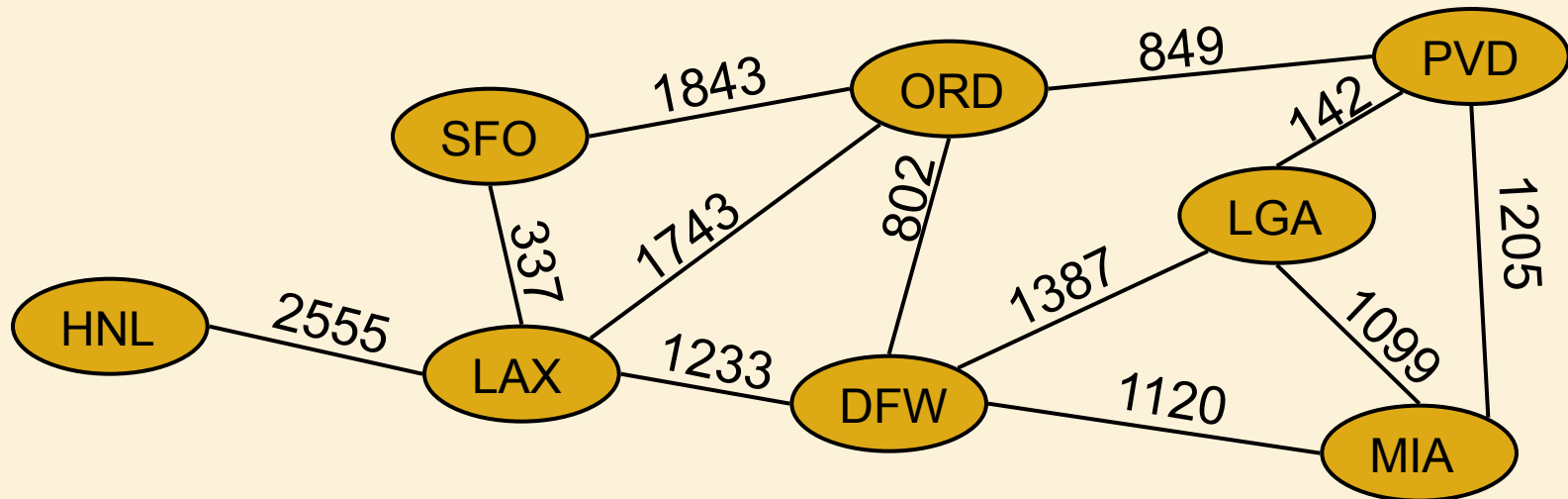
Shortest Path on Weighted Graphs

- 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'?



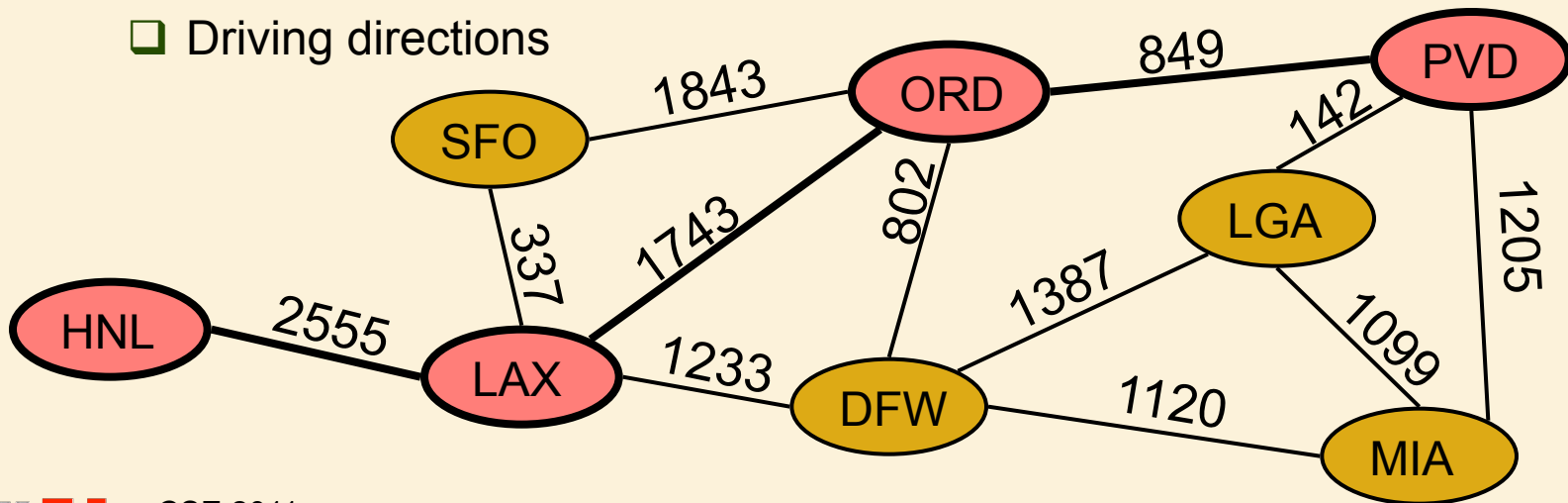
Weighted Graphs

- 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 $p = \langle v_0, v_1, \dots, v_k \rangle = \sum_{i=1}^k w(v_{i-1}, v_i)$

Shortest-path weight from u to v :

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

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

Shortest Path Properties

Property 1 (Optimal Substructure):

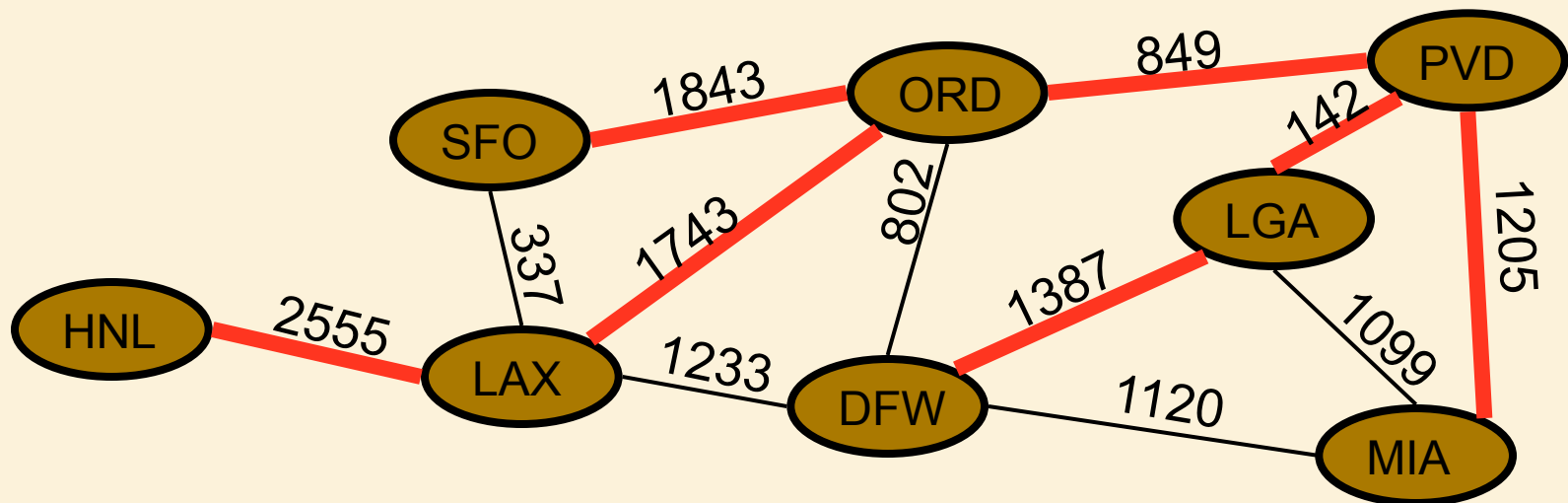
A subpath of a shortest path is itself a shortest path

Property 2 (Shortest Path Tree):

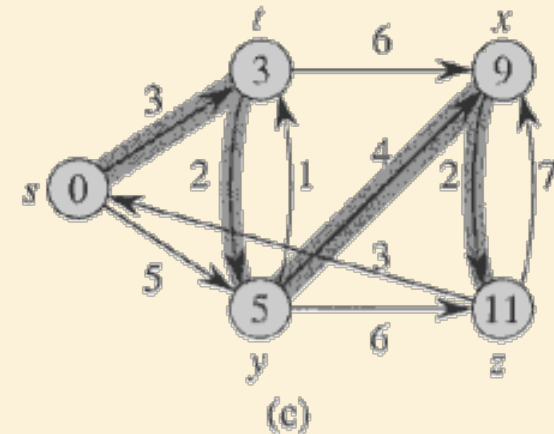
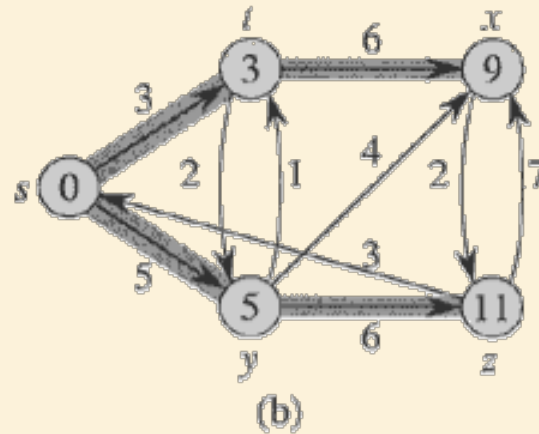
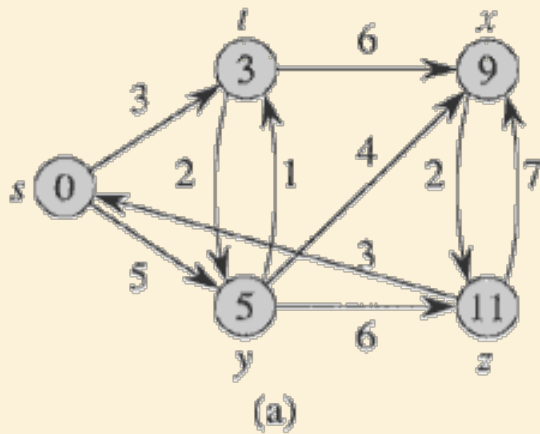
There is a tree of shortest paths from a start vertex to all the other vertices

Example:

Tree of shortest paths from Providence



Shortest path trees are not necessarily unique



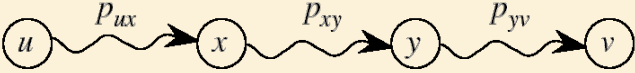
Single-source shortest path search induces a search tree rooted at s .

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 .

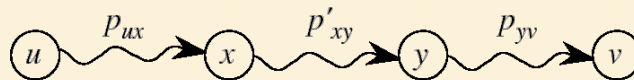


Then $\delta(u, v) = w(p) = w(p_{ux}) + w(p_{xy}) + w(p_{yv})$.

Now suppose there exists a shorter path $x \xrightarrow{p'_{xy}} \dots \rightarrow y$.

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

Construct p' :



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

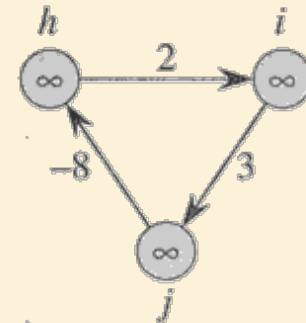
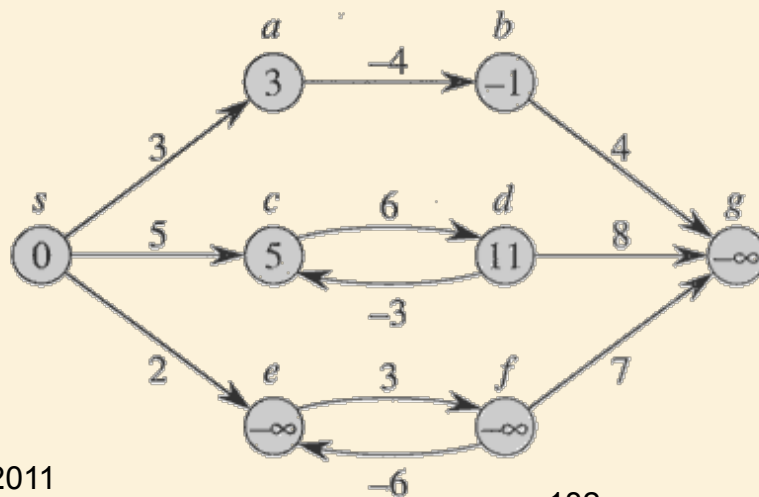
So p wasn't a shortest path after all!

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 $w(s, v) = -\infty$ 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, $\pi[v] = \text{NIL}$.

✧ π induces a tree — **shortest-path tree**.

Initialization

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

INIT-SINGLE-SOURCE(V, s)

for each v in V

do $d[v] \leftarrow \infty$

$\pi[v] \leftarrow \text{NIL}$

$d[s] \leftarrow 0$

Relaxing an edge

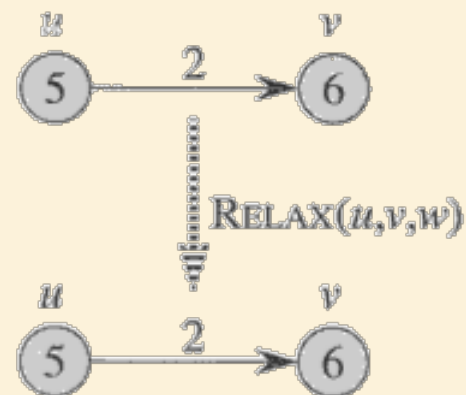
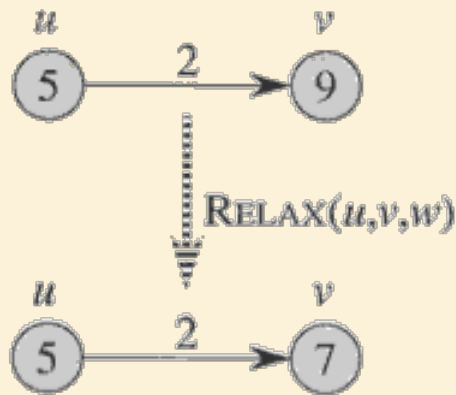
- Can we improve shortest-path estimate for v by first going to u and then following edge (u, v) ?

RELAX(u, v, w)

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

$d[v] \leftarrow d[u] + w(u, v)$

$\pi[v] \leftarrow u$



General single-source shortest-path strategy

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

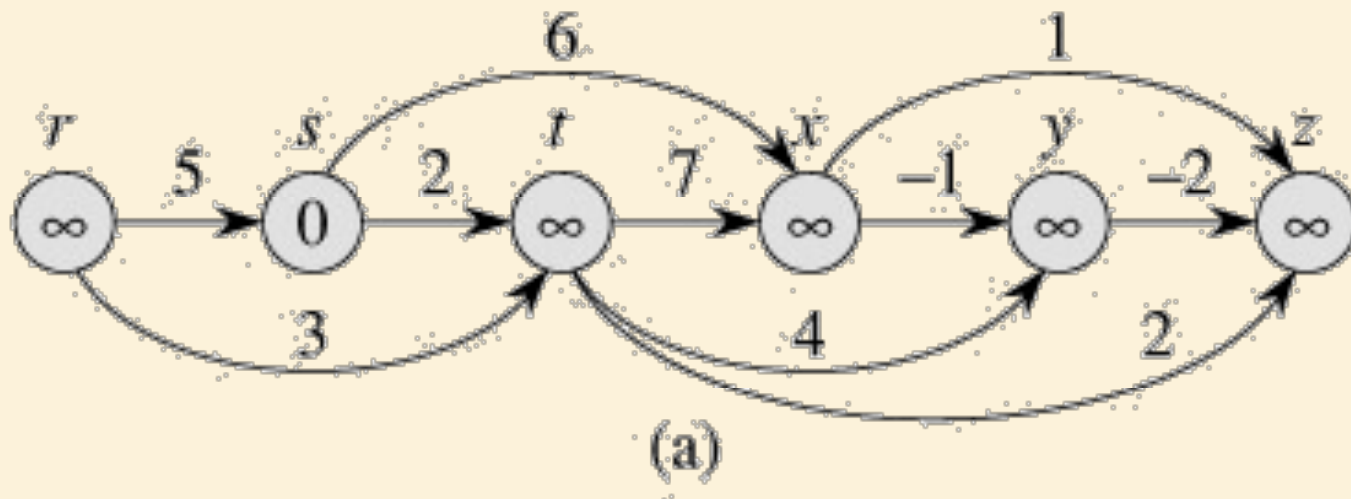
Algorithms differ in the order in which edges are taken and how many times each edge is relaxed.

Example 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



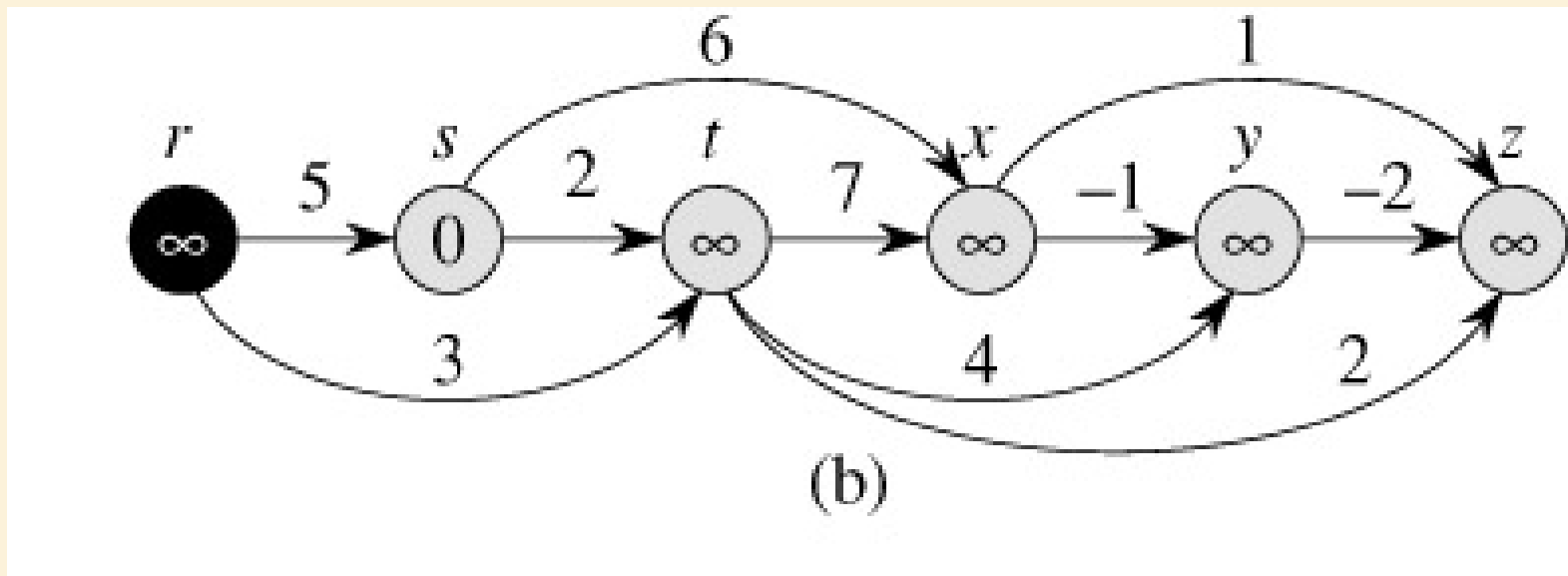
Algorithm

DAG-SHORTEST-PATHS(G, w, s)

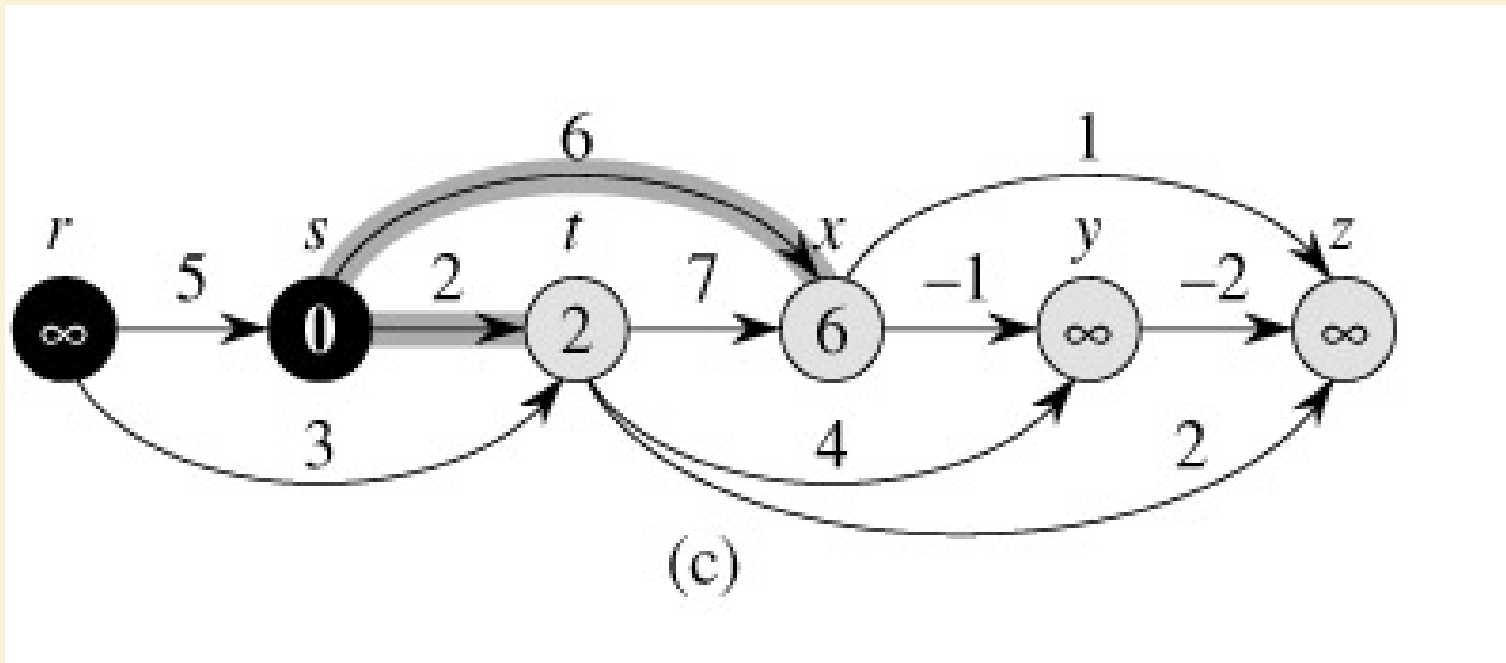
- 1 topologically sort the vertices of G
- 2 INITIALIZE-SINGLE-SOURCE(G, s)
- 3 **for** each vertex u , taken in topologically sorted order
- 4 **do for** each vertex $v \in Adj[u]$
- 5 **do** RELAX(u, v, w)

Time: $\Theta(V + E)$

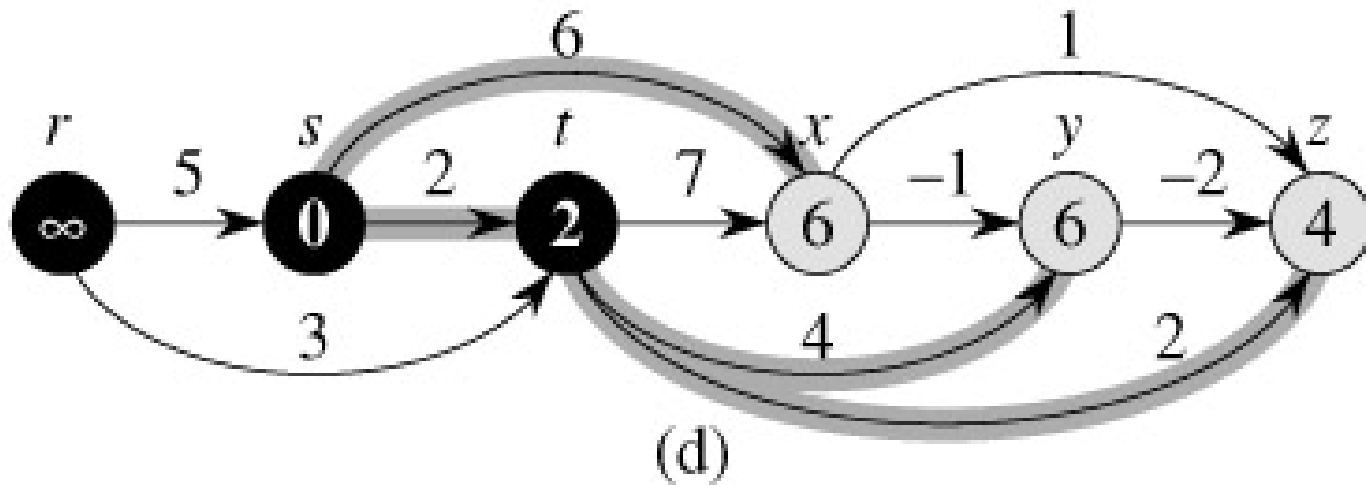
Example



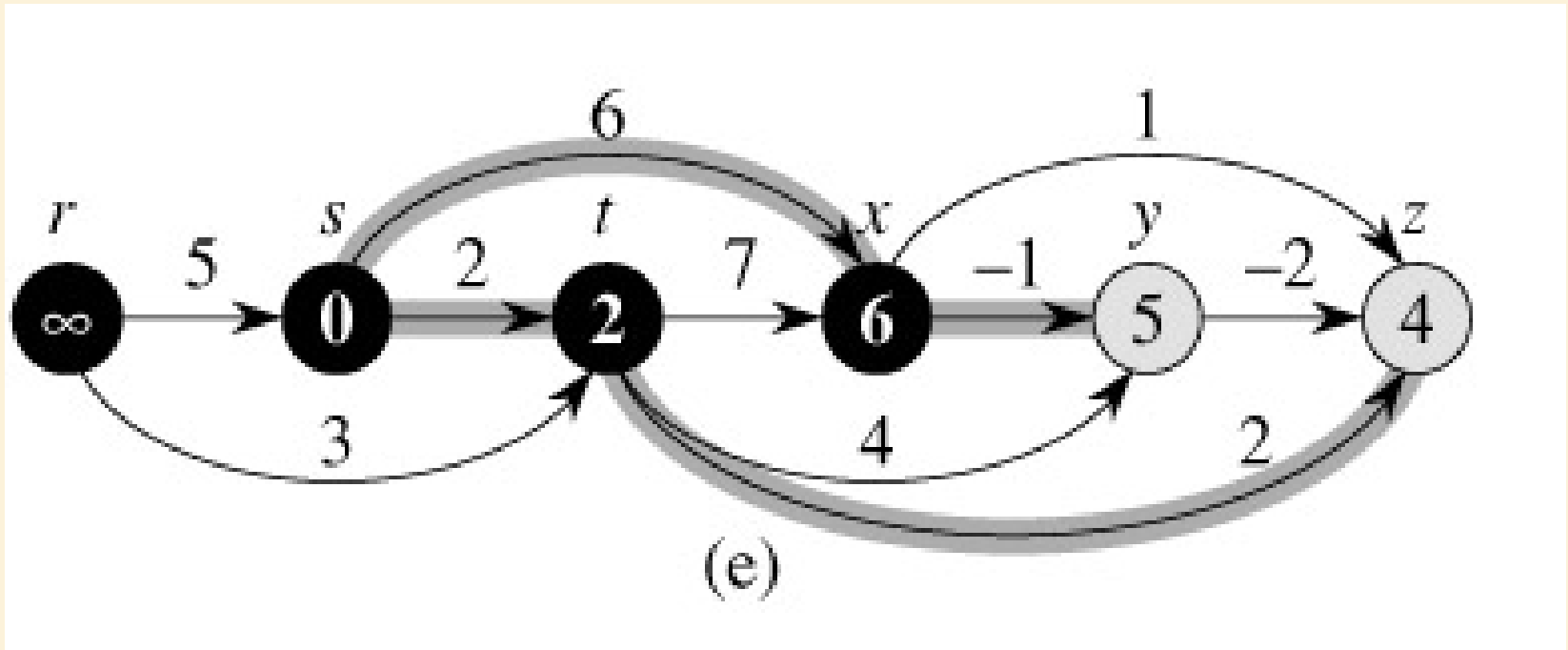
Example



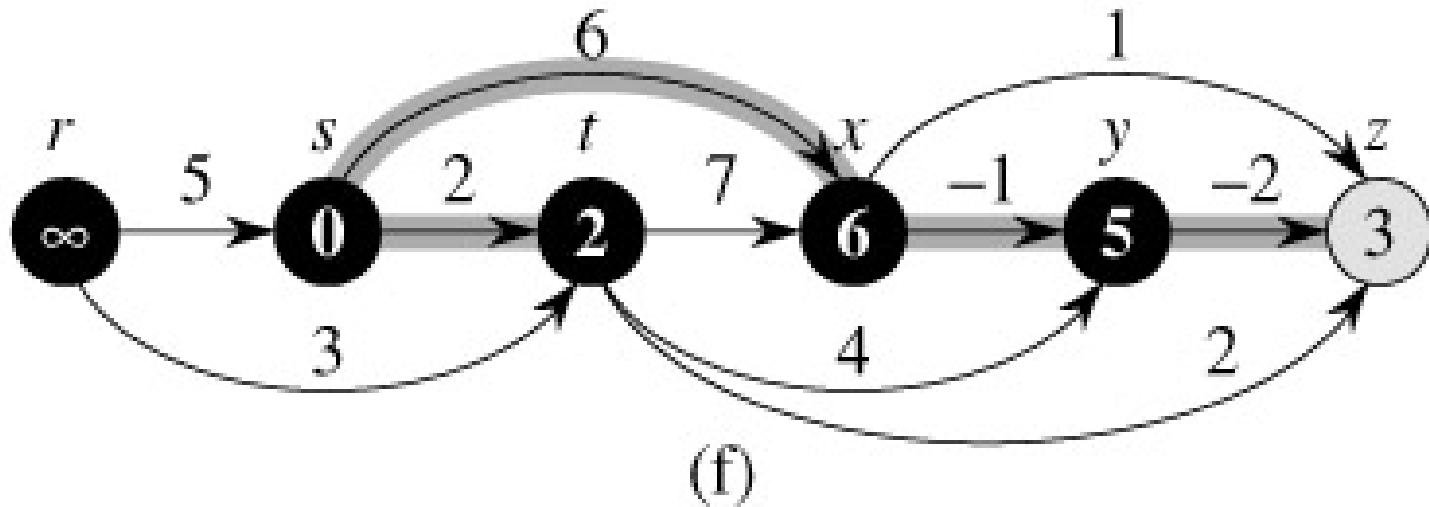
Example



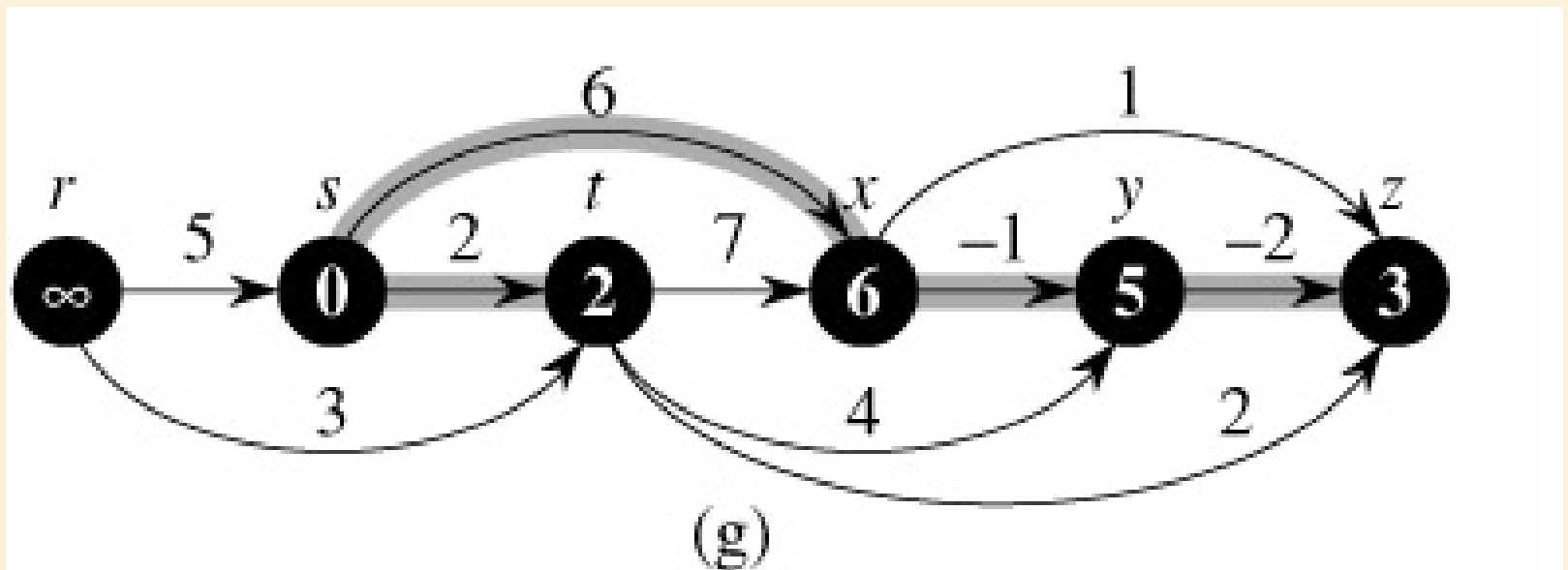
Example



Example



Example



Correctness: Path relaxation property

Let $p = \langle v_0, v_1, \dots, v_k \rangle$ be a shortest path from $s = v_0$ to v_k .

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

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)

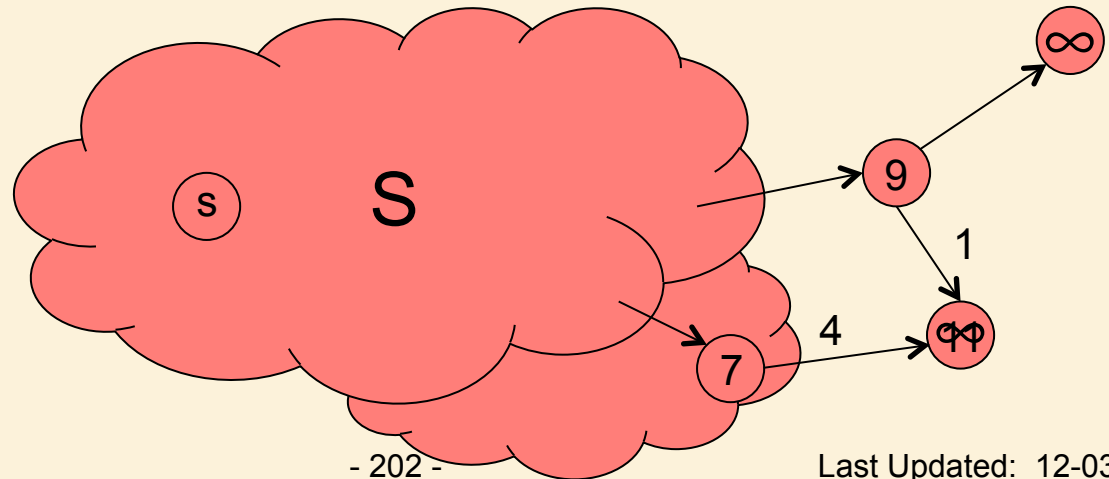
- Applies to general, weighted, directed or undirected graph (may contain cycles).
- But weights must be non-negative. (But they can be 0!)
- Essentially a weighted version of BFS.
 - ❑ Instead of a FIFO queue, uses a priority queue.
 - ❑ Keys are shortest-path weights ($d[v]$).
- Maintain 2 sets of 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( $G, w, s$ )
1  INITIALIZE-SINGLE-SOURCE( $G, s$ )
2   $S \leftarrow \emptyset$ 
3   $Q \leftarrow V[G]$ 
4  while  $Q \neq \emptyset$ 
5      do  $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
6           $S \leftarrow S \cup \{u\}$ 
7          for each vertex  $v \in \text{Adj}[u]$ 
8              do RELAX( $u, v, w$ )
```

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

Dijkstra's algorithm: Analysis

- Analysis:
 - Using minheap, queue operations takes $O(\log V)$ time

DIJKSTRA(G, w, s)

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

→ Running Time is $O(E \log V)$

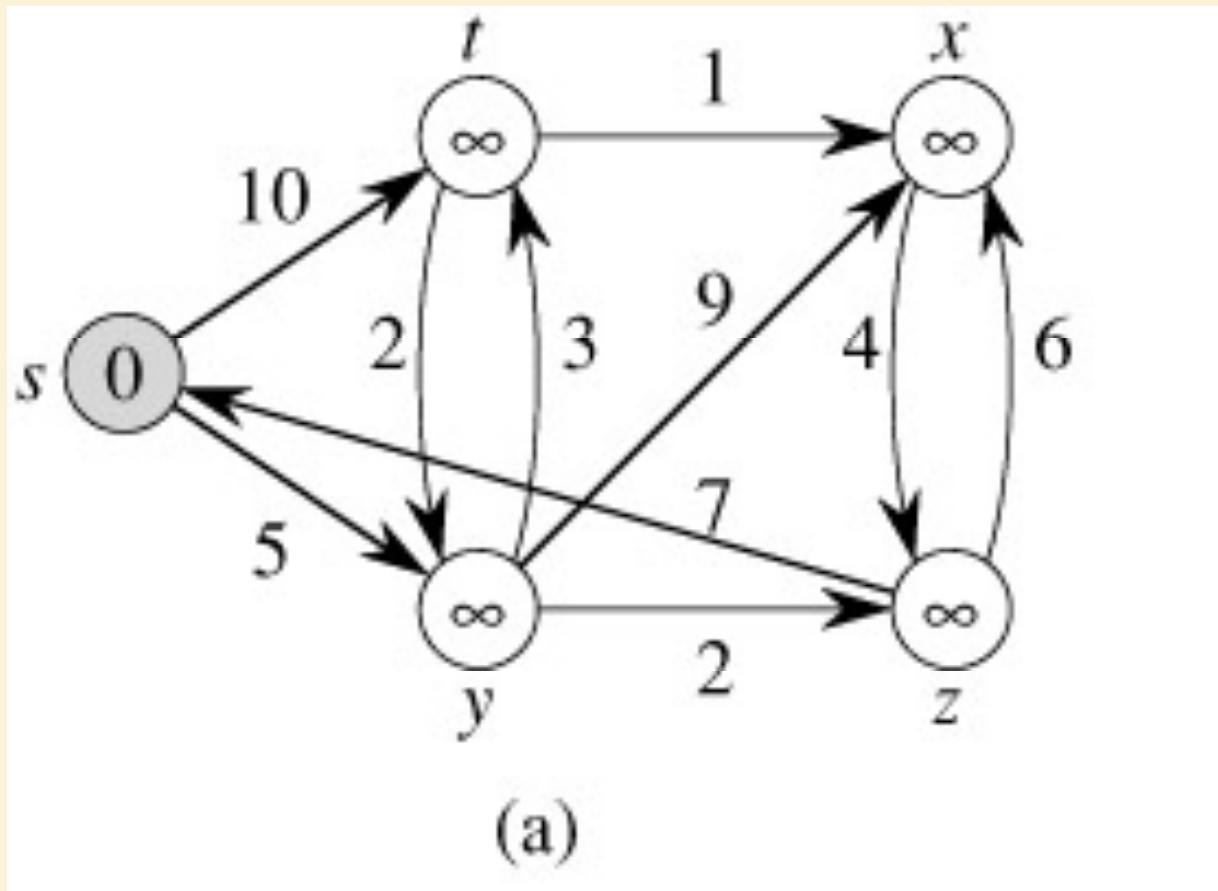
Example

Key:

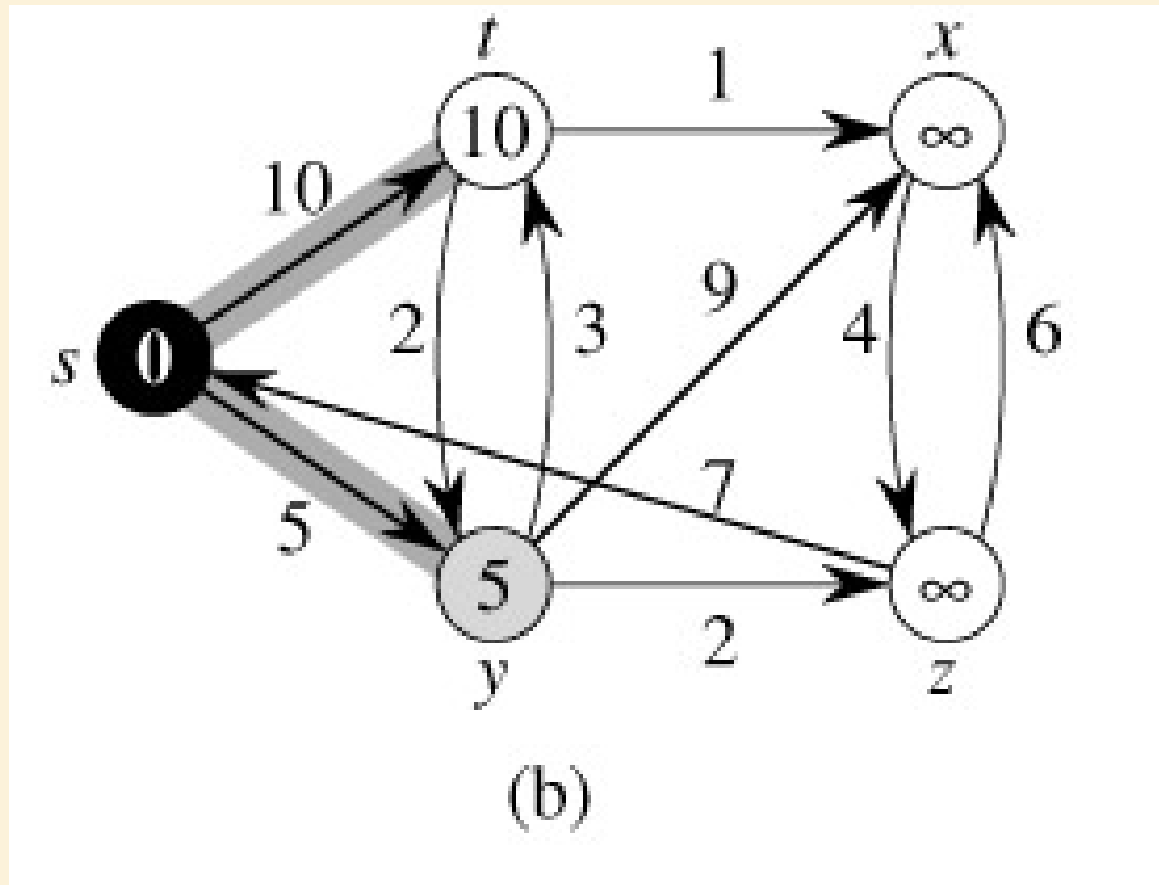
White $\Leftrightarrow$ Vertex $\in Q = V - S$

Grey $\Leftrightarrow$ Vertex = min(Q)

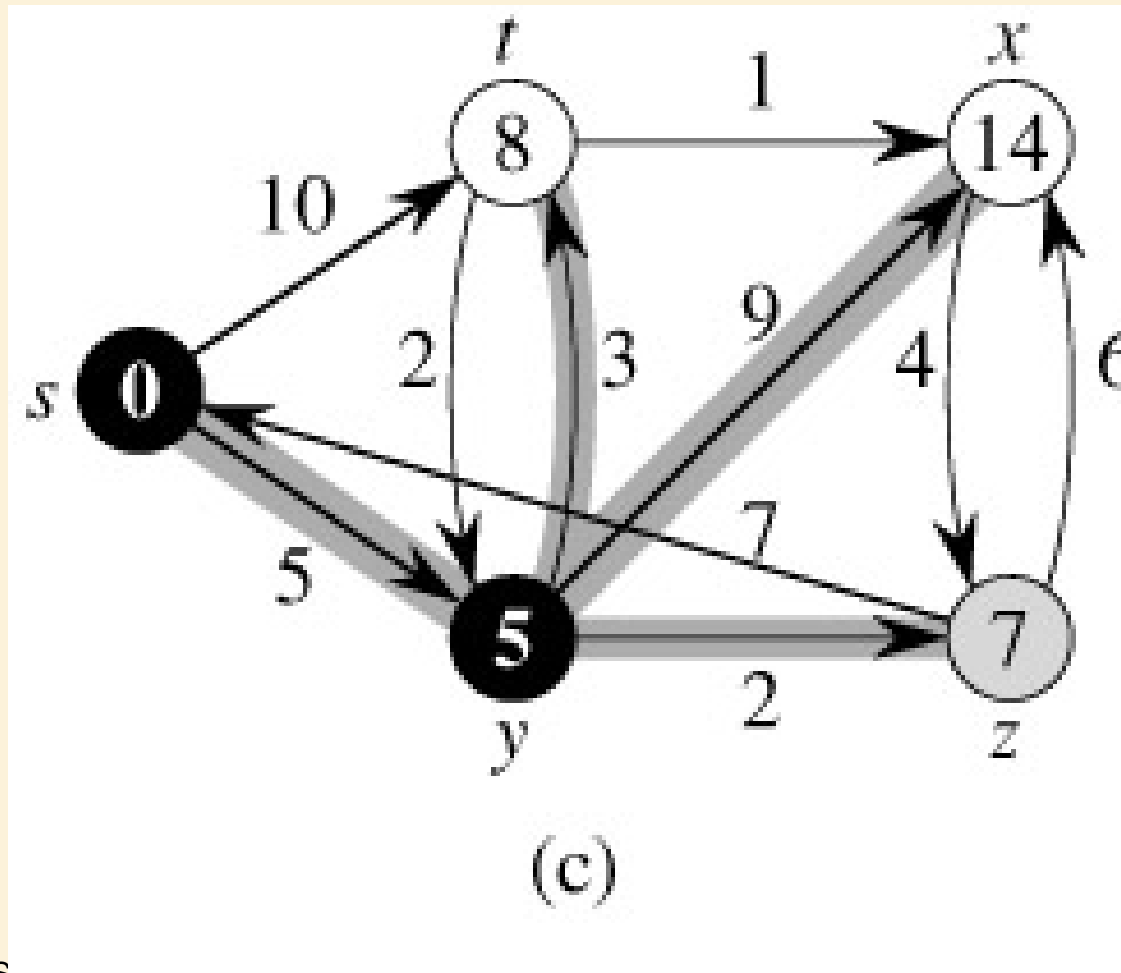
Black $\Leftrightarrow$ Vertex $\in S$, Off Queue



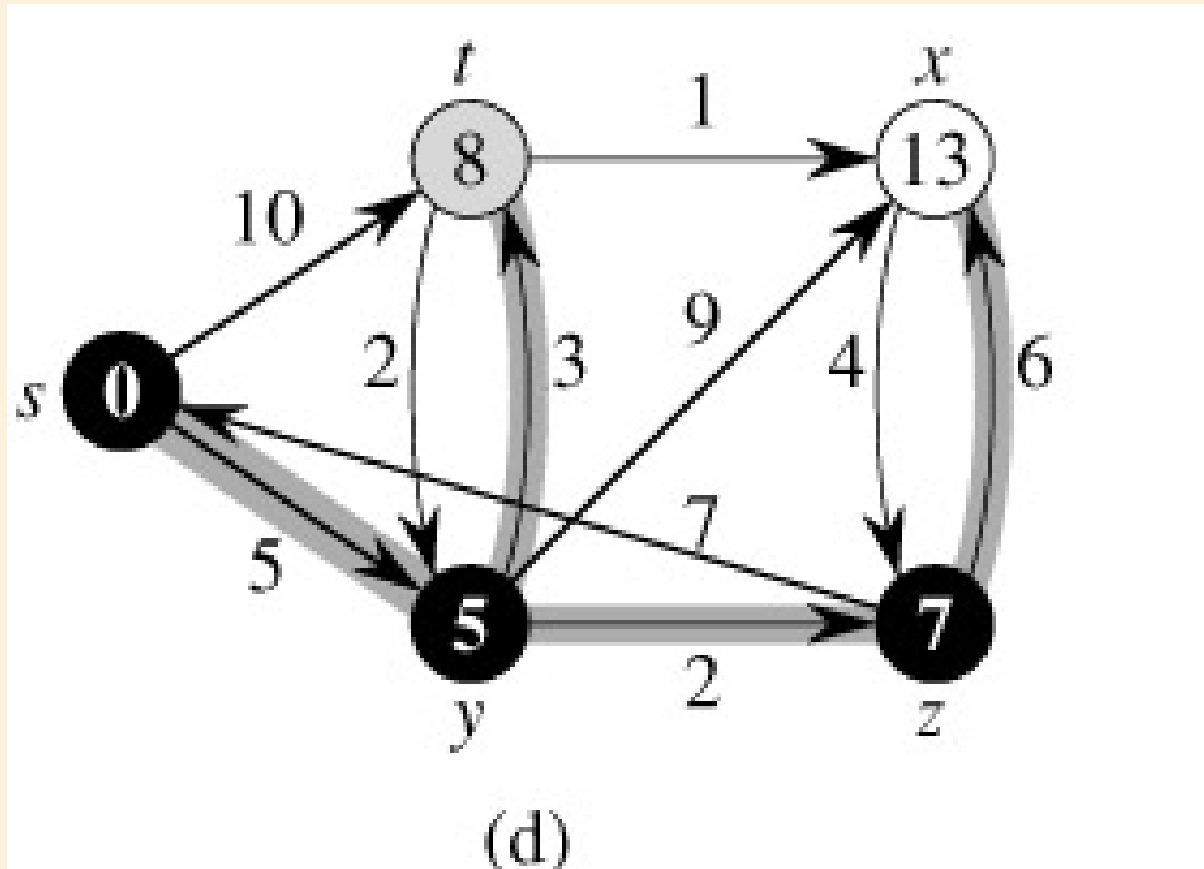
Example



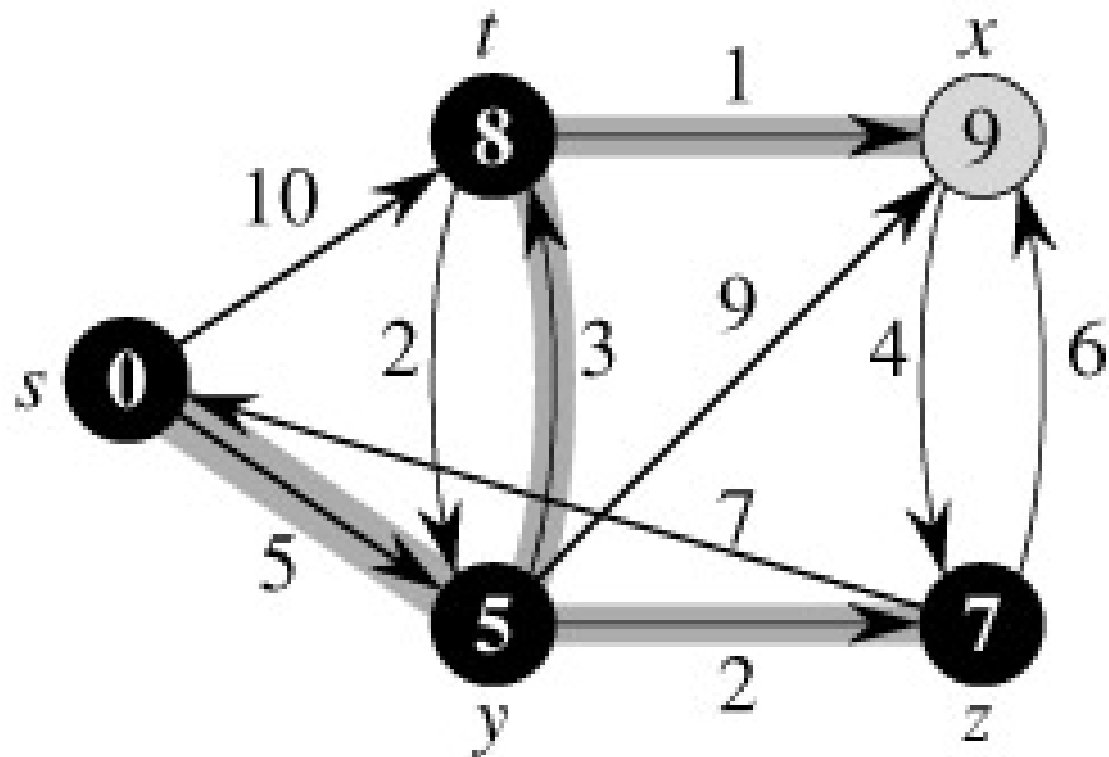
Example



Example

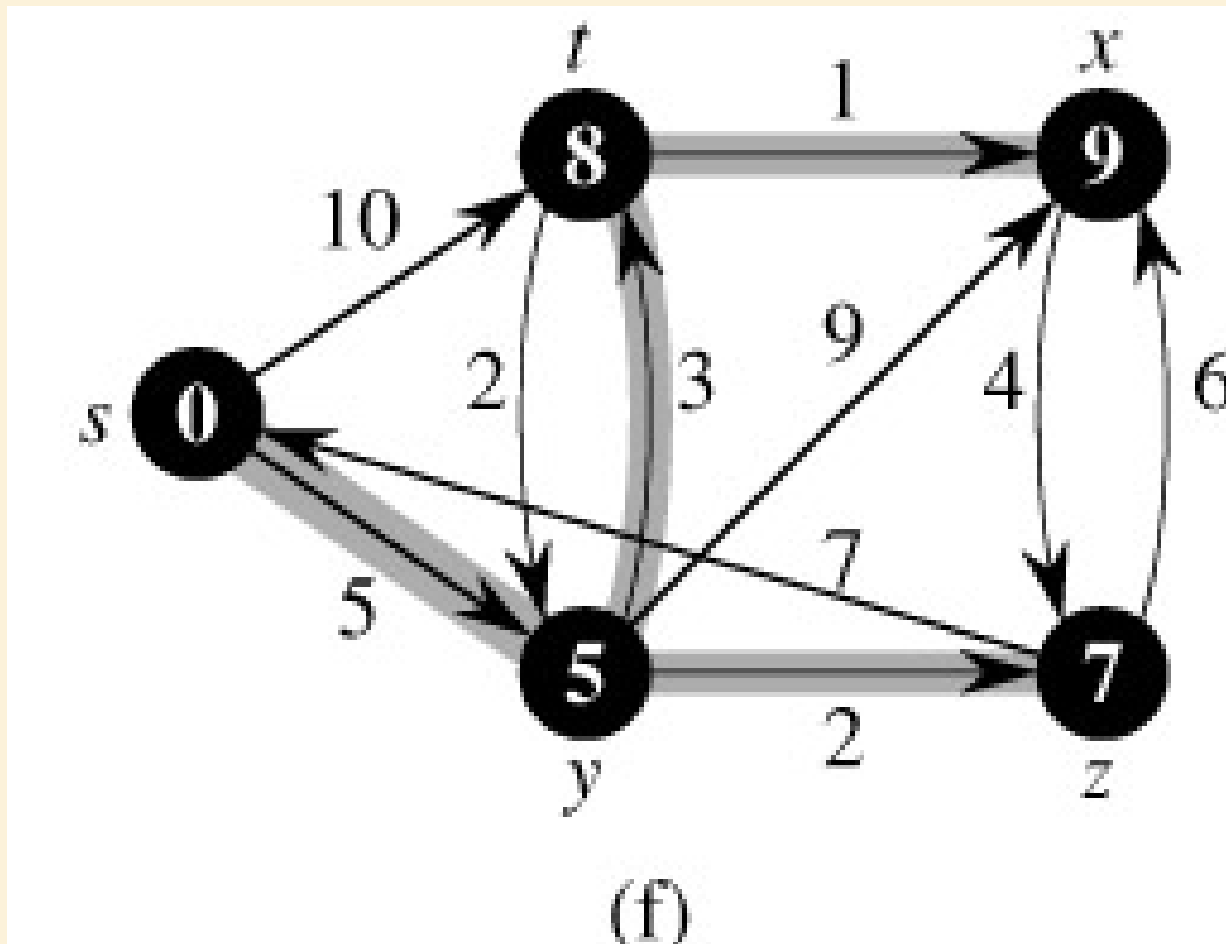


Example

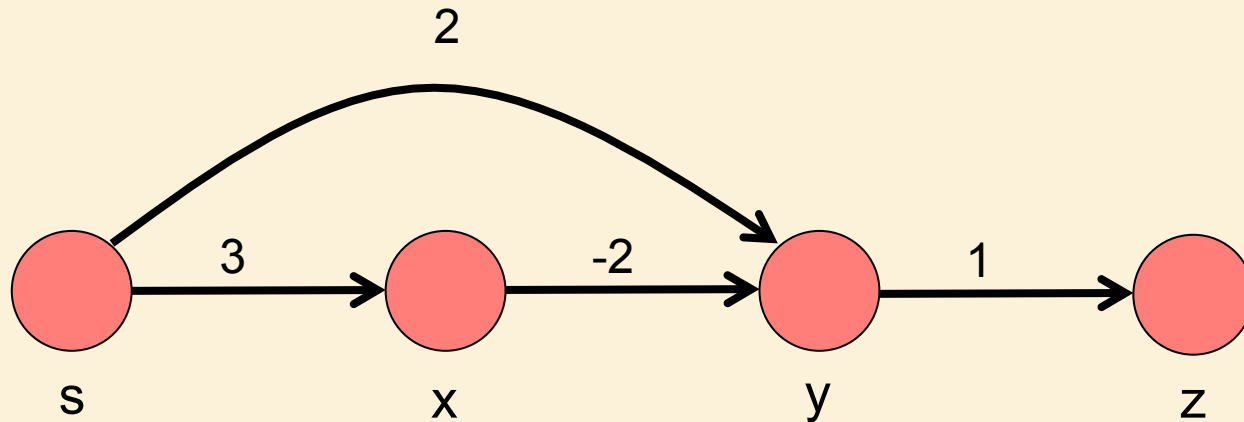


(e)

Example



Dijkstra's Algorithm Cannot Handle Negative Edges



Correctness of Dijkstra's algorithm

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

➤ **Loop invariant:** $d[v] = \delta(s, v)$ for all v in S .

- ❑ **Initialization:** Initially, S is empty, so trivially true.
- ❑ **Termination:** At end, Q is empty $\rightarrow S = V \rightarrow d[v] = \delta(s, v)$ for all v in V .
- ❑ **Maintenance:**

✧ Need to show that

- ✧ $d[u] = \delta(s, u)$ when u is added to S in each iteration.
- ✧ $d[u]$ does not change once u is added to S .

Correctness of Dijkstra's Algorithm: Upper Bound Property

➤ Upper Bound Property:

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

- Proof:

By induction.

Base Case: $d[v] \geq \delta(s, v) \forall v \in V$ immediately after initialization, since
 $d[s] = 0 = \delta(s, s)$
 $d[v] = \infty \forall v \neq s$

Inductive Step:

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

Suppose we relax edge (u, v) .

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

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

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

A valid path from s to v !

Correctness of Dijkstra's Algorithm

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

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

Let y be first vertex in $V - S$ on shortest path to u

Let x be the predecessor of y on the shortest path to u

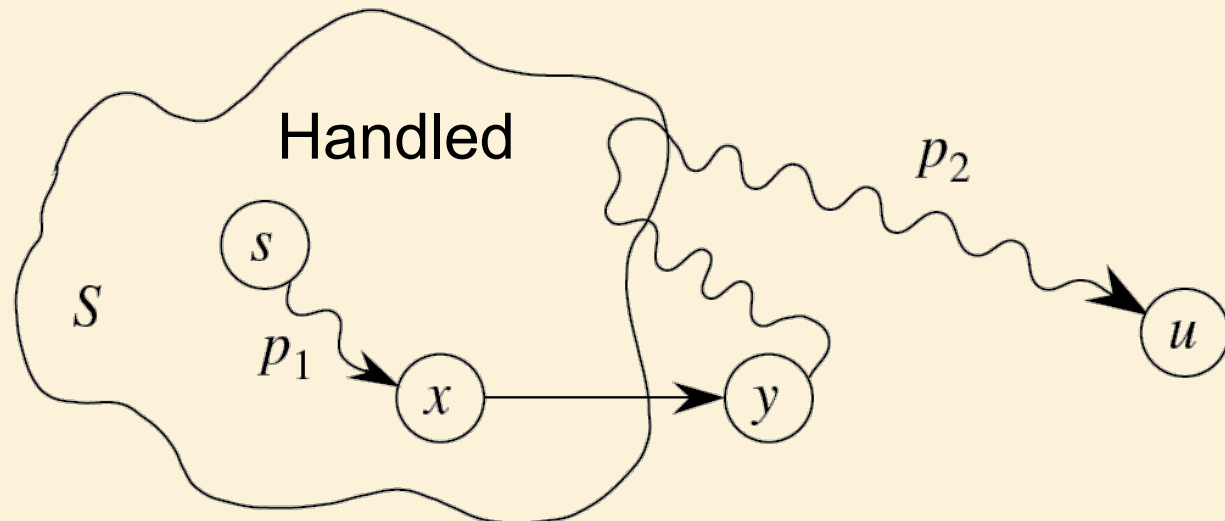
Claim: $d[y] = \delta(s, y)$ when u is added to S .

Proof:

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

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

Optimal substructure property!



Correctness of Dijkstra's Algorithm

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

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

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

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

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

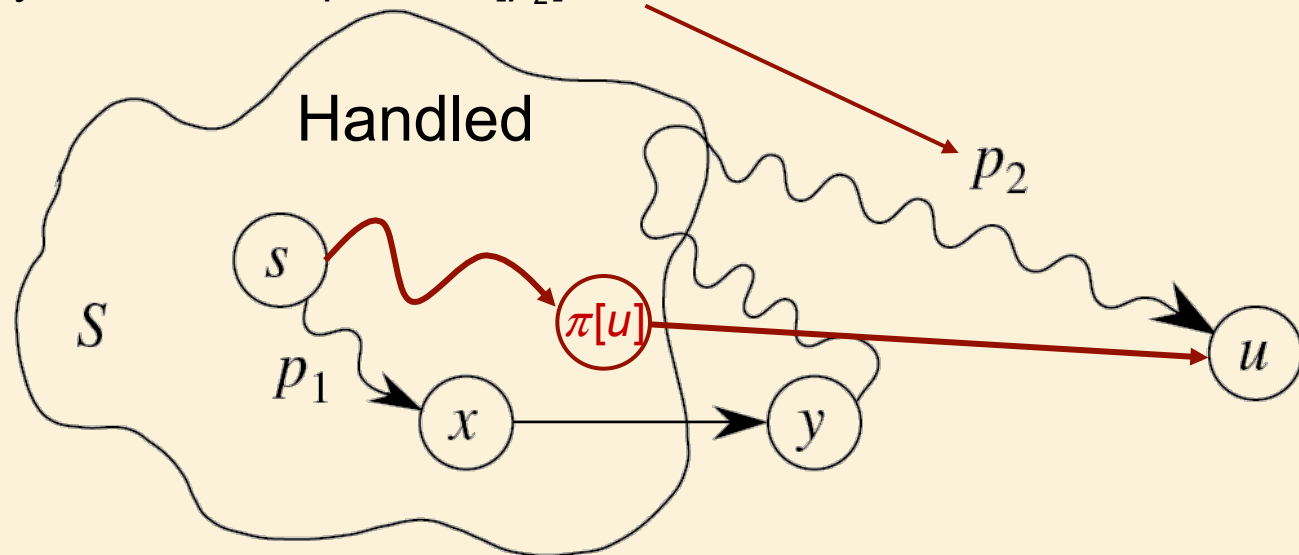
DIJKSTRA(G, w, s)

```

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

Consequences:

There is a shortest path to u such that the predecessor of u $\pi[u] \in S$ when u is added to S .
 The path through y can only be a shortest path if $w[p_2] = 0$.



Correctness of Dijkstra's algorithm

DIJKSTRA(G, w, s)

1 INITIALIZE-SINGLE-SOURCE(G, s)

2 $S \leftarrow \emptyset$

3 $Q \leftarrow V[G]$

4 **while** $Q \neq \emptyset$

5 **do** $u \leftarrow \text{EXTRACT-MIN}(Q)$

6 $S \leftarrow S \cup \{u\}$

7 **for each vertex** $v \in \text{Adj}[u]$

8 **do** RELAX(u, v, w)

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

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

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

➤ **Loop invariant:** $d[v] = \delta(s, v)$ for all v in S .

□ **Maintenance:**

✧ Need to show that

✧ $d[u] = \delta(s, u)$ when u is added to S in each iteration. ✓

✧ $d[u]$ does not change once u is added to S . ?