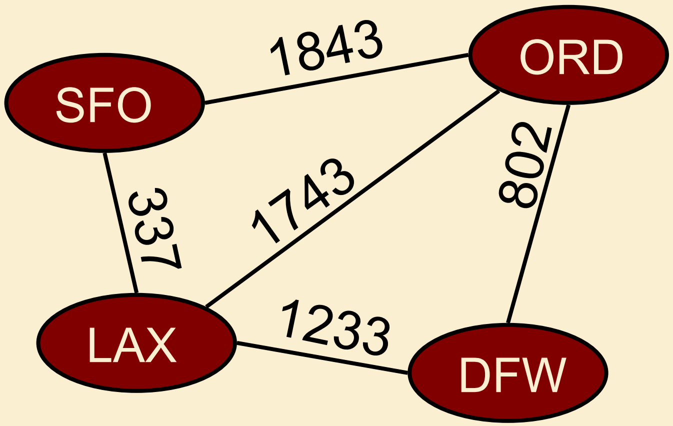
Graphs – Breadth First Search



Outcomes

- By understanding this lecture, you should be able to:
  - Label a graph according to the order in which vertices are discovered in a breadth-first search.
  - Identify the current state of a breadth-first search in terms of vertices that are previously discovered, just discovered or undiscovered.
  - Identify the contents of the breadth-first search queue at any state of the search.
  - Implement breadth-first search
  - Demonstrate simple applications of breadth-first search

Outline

- BFS Algorithm
- BFS Application: Shortest Path on an unweighted graph

Outline

- **BFS Algorithm**
- BFS Application: Shortest Path on an unweighted graph

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

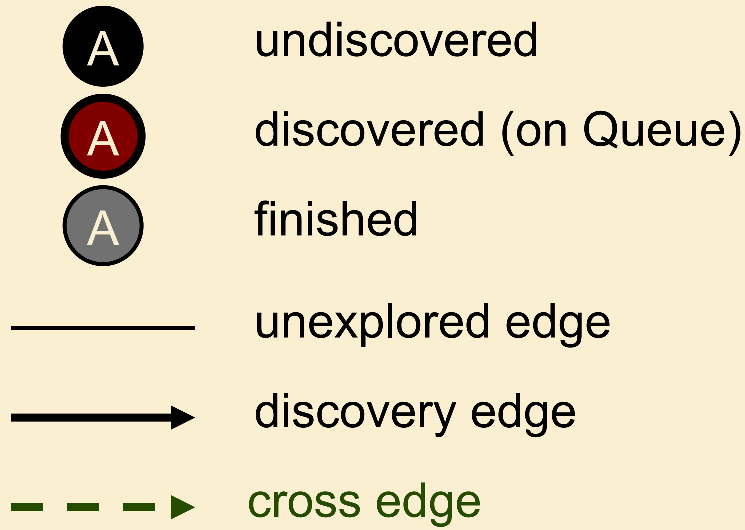
Postcondition: all vertices in G reachable from s have been visited

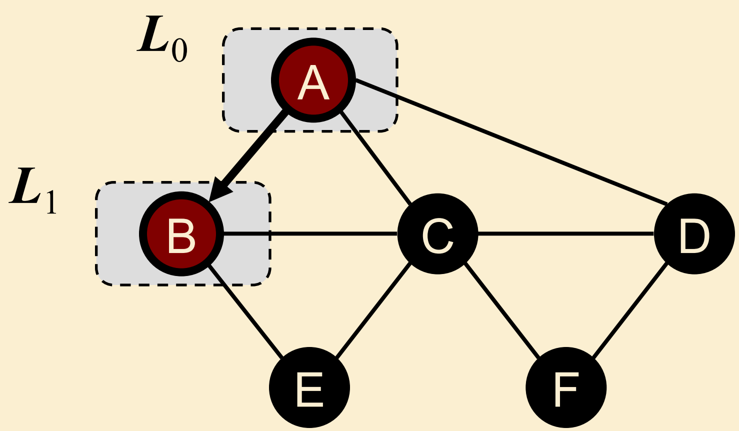
```
for each vertex u ∈ V[G]
        color[u] ← BLACK //initialize vertex
colour[s] ← RED
Q.enqueue(s)
while Q ≠ ∅
        u ← Q.dequeue()
        for each v ∈ Adj[u] //explore edge (u,v)
                if color[v] = BLACK
                        colour[v] ← RED
                        Q.enqueue(v)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

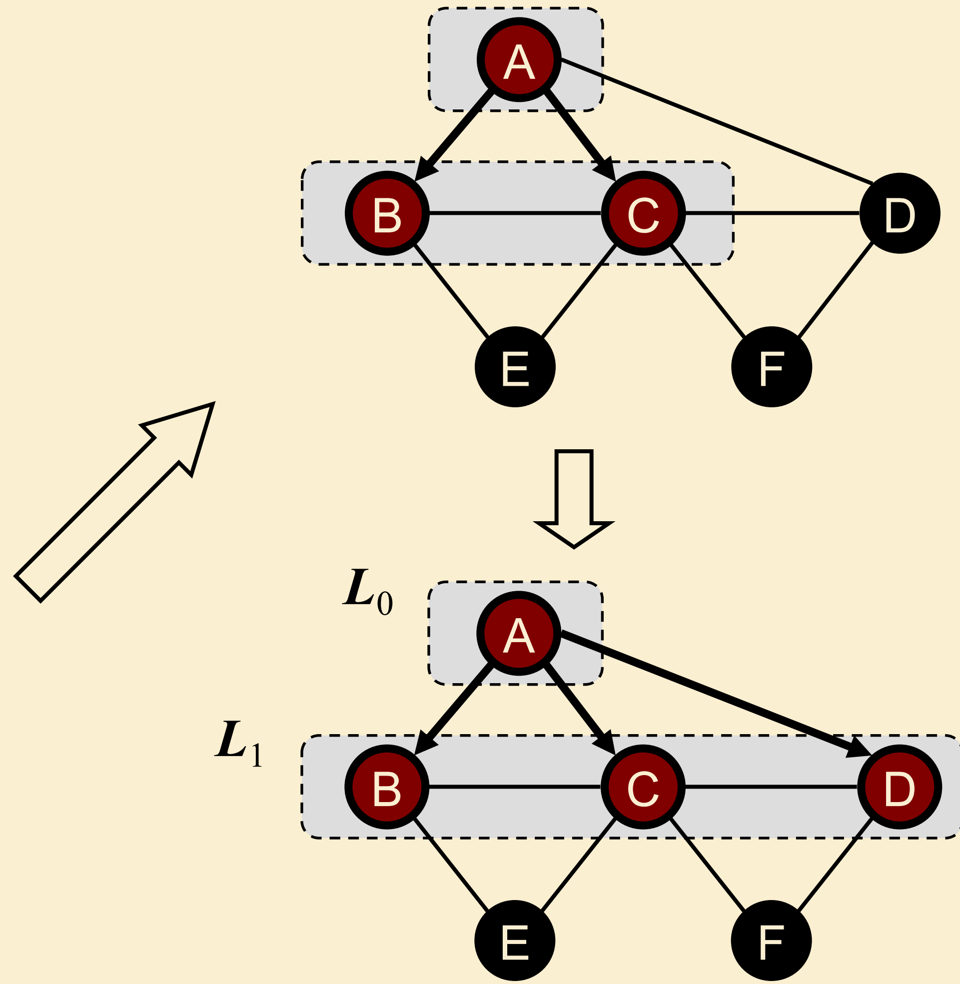
BFS is a Level-Order Traversal

- 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, …

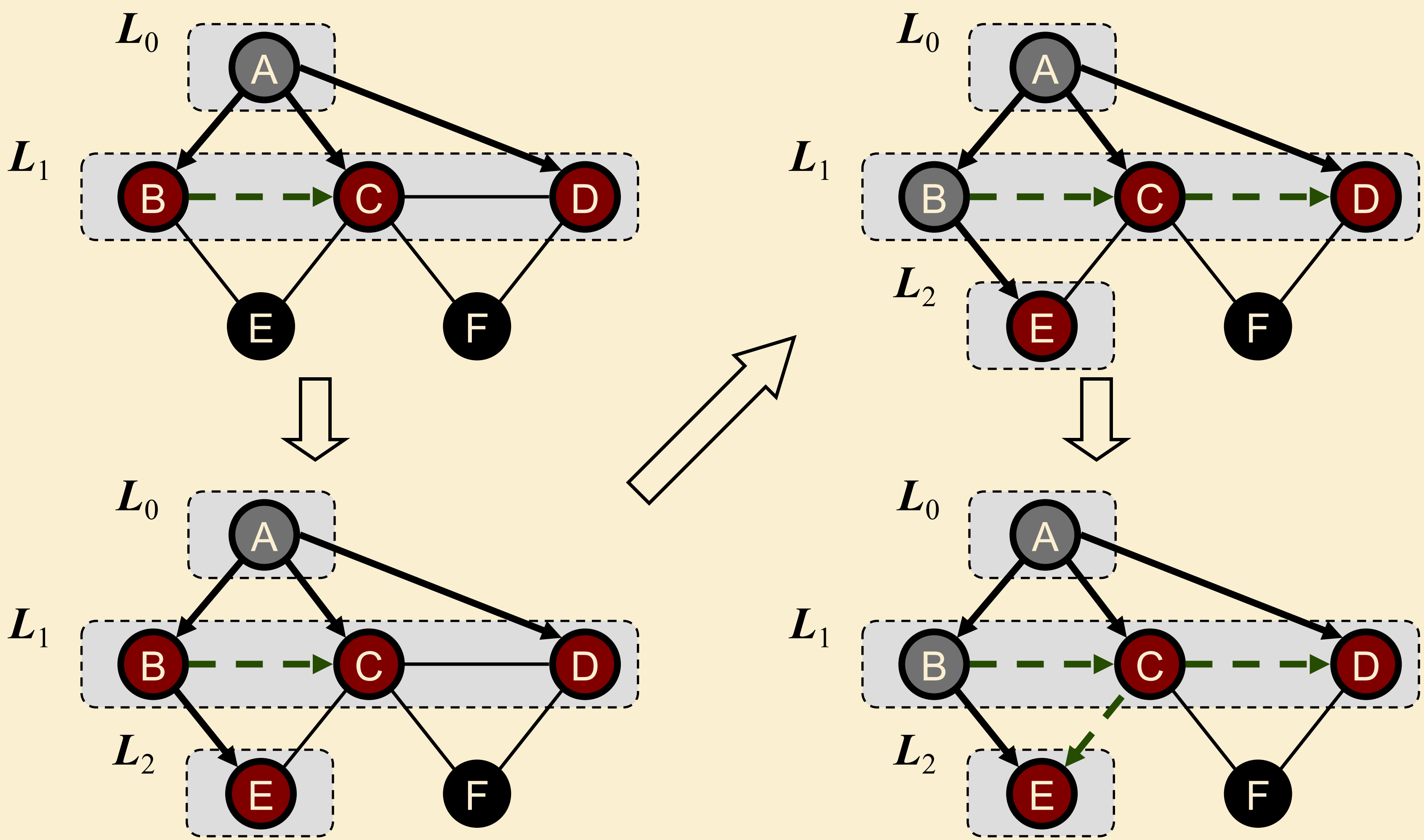
BFS Example


A
undiscovered
A
discovered (on Queue)
A
finished
unexplored edge
discovery edge
cross edge

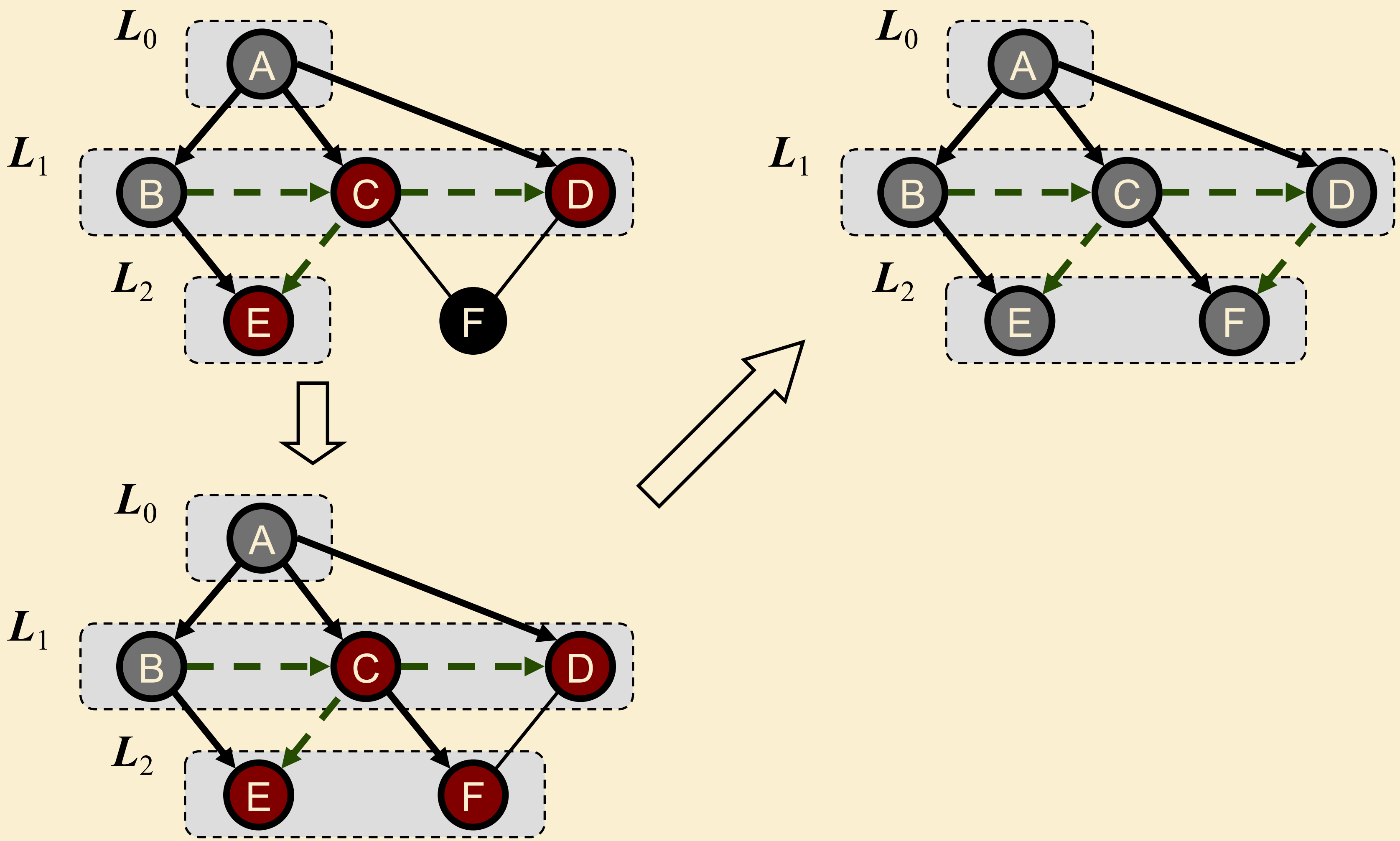

L_0
L_1
A
B
C
D
E
F


A
B
C
D
E
F
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

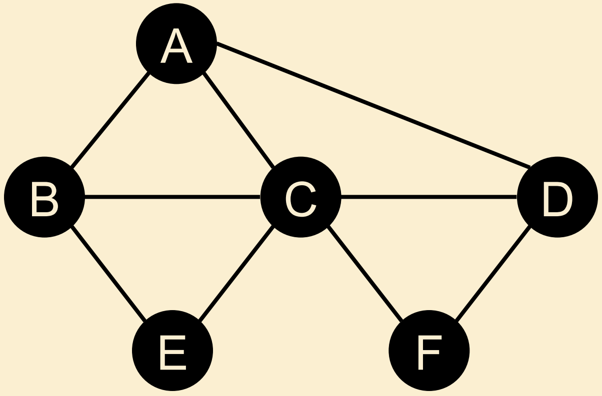
Property 2

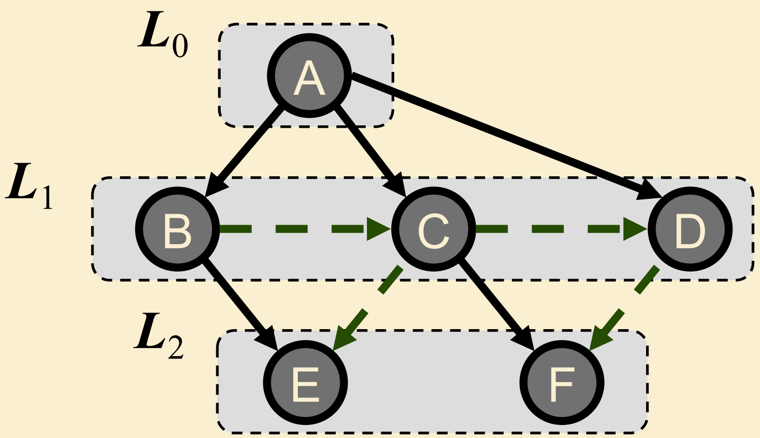
The discovery edges labeled by $BFS(G, s)$ form a spanning tree T_s of G_s

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 placed on the queue once
- Thus BFS runs in $O(|V|+|E|)$ time provided the graph is represented by an adjacency list structure

Applications

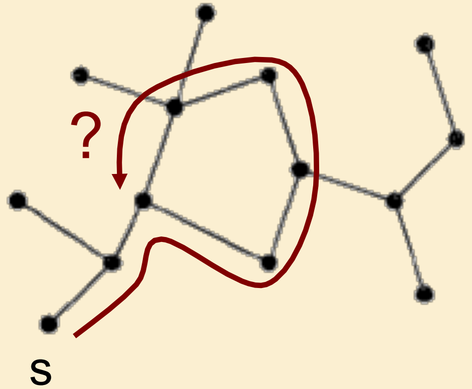
- BFS traversal can be specialized to solve the following problems in $O(|V|+|E|)$ time:
  - Compute the connected components of G
  - Compute a spanning forest of G
  - Find a simple cycle in G, or report that G is a forest
  - Given two vertices of G, find a path in G between them with the minimum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

Outline

- BFS Algorithm
- **BFS Application: Shortest Path on an unweighted graph**

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 trees
  - 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

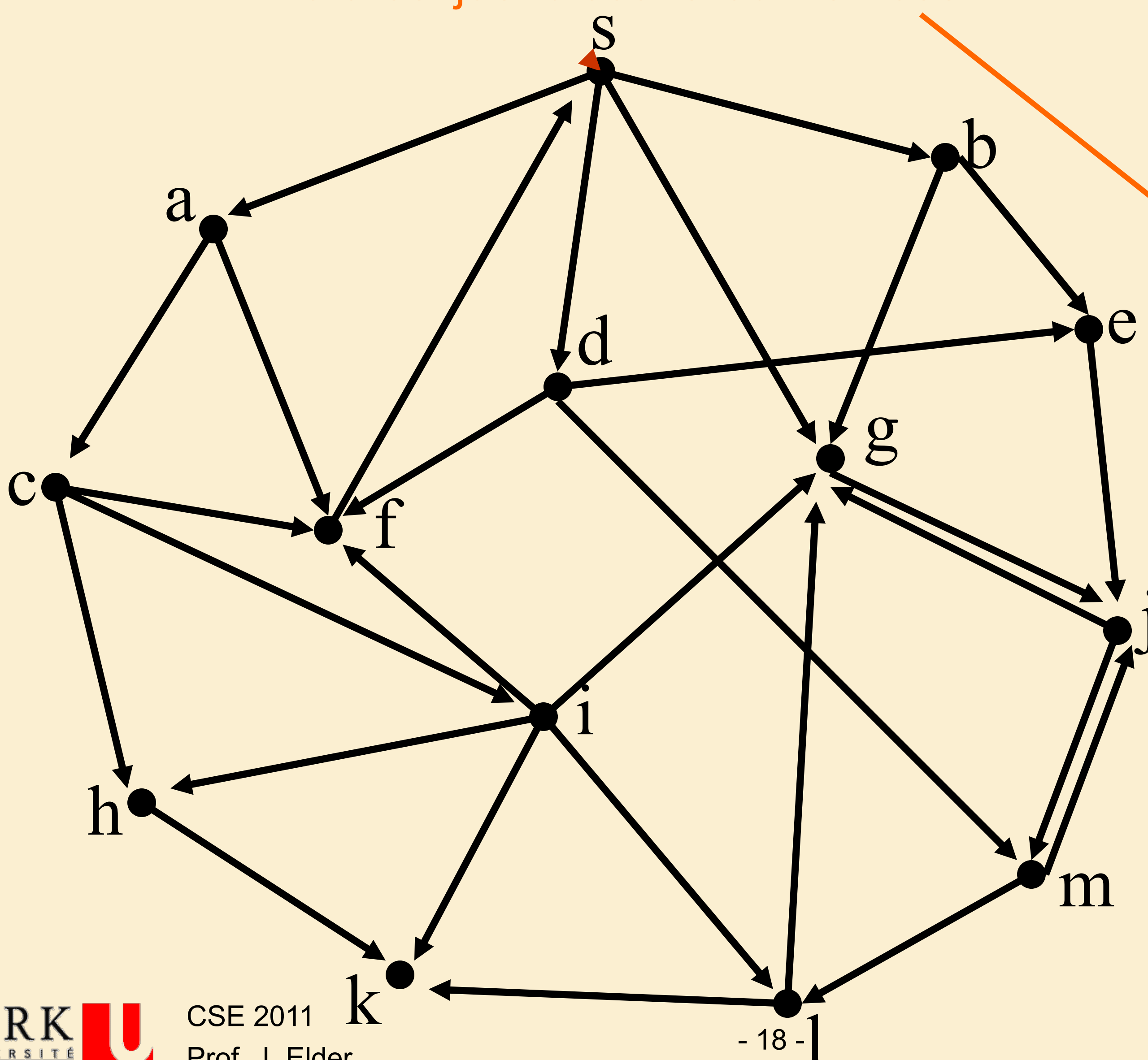
Postcondition: $d[u] =$ shortest distance $\delta[u]$ and

π[u] = predecessor of u on shortest path 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
```

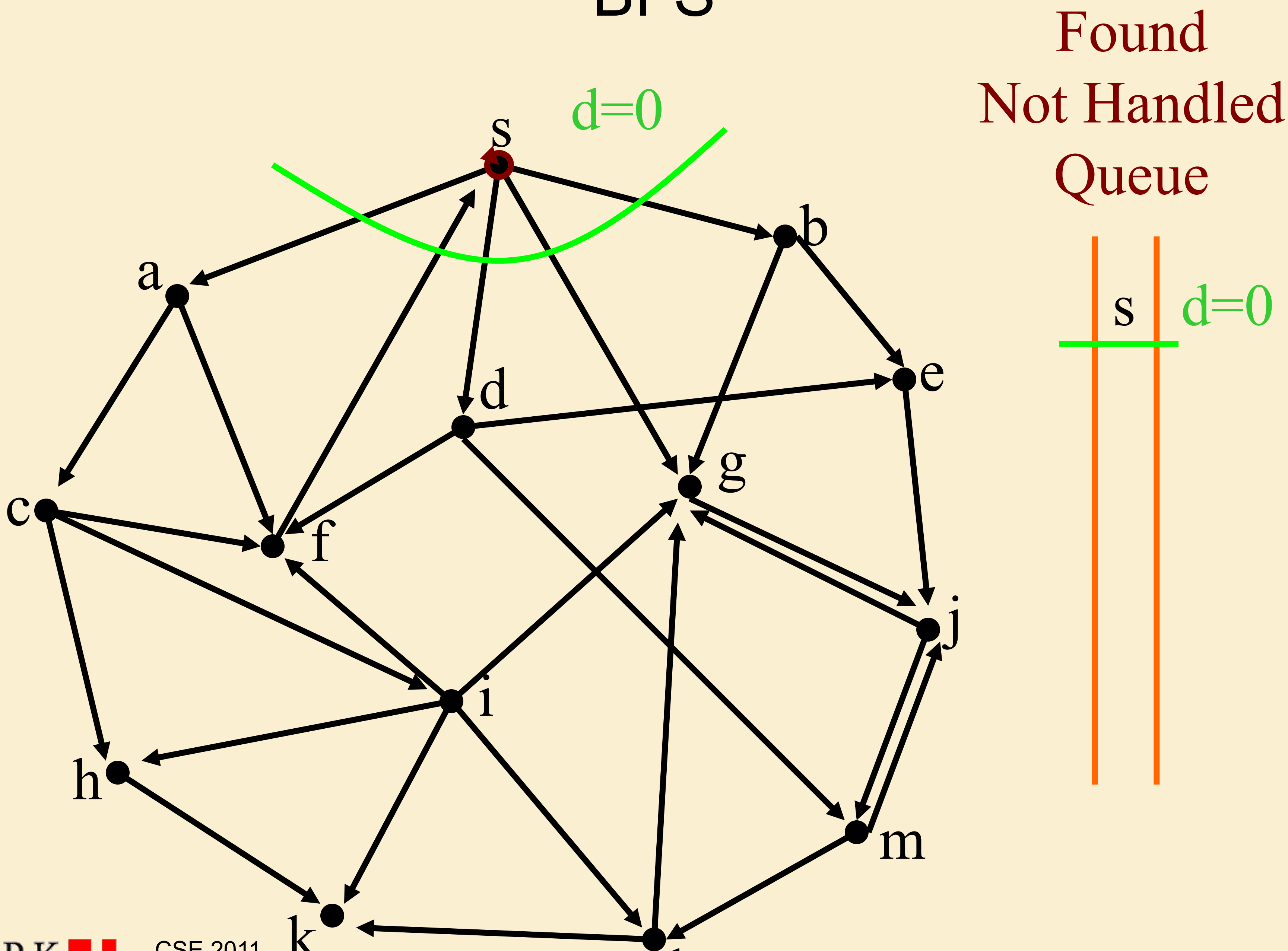
BFS

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

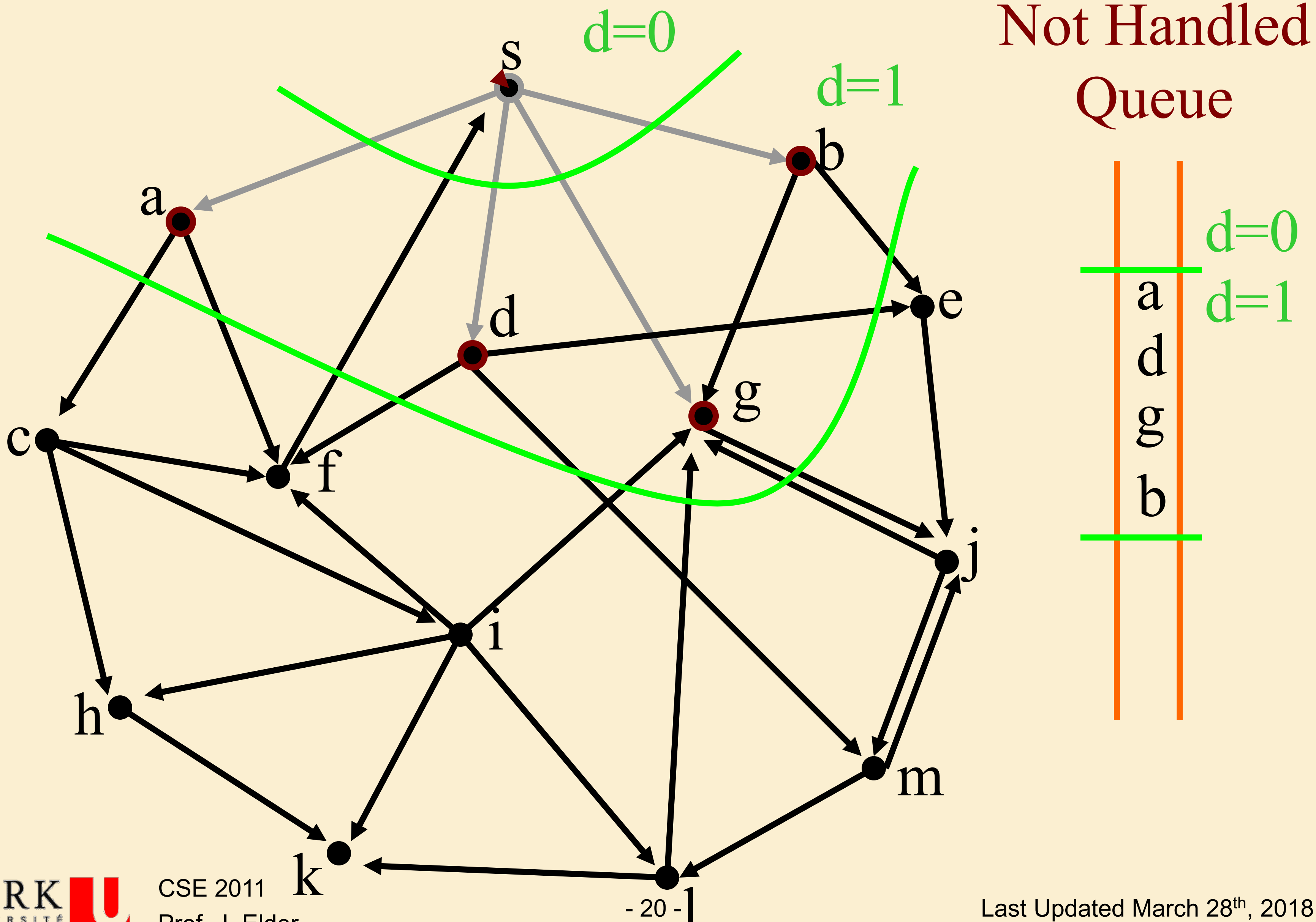


Found
Not Handled
Queue

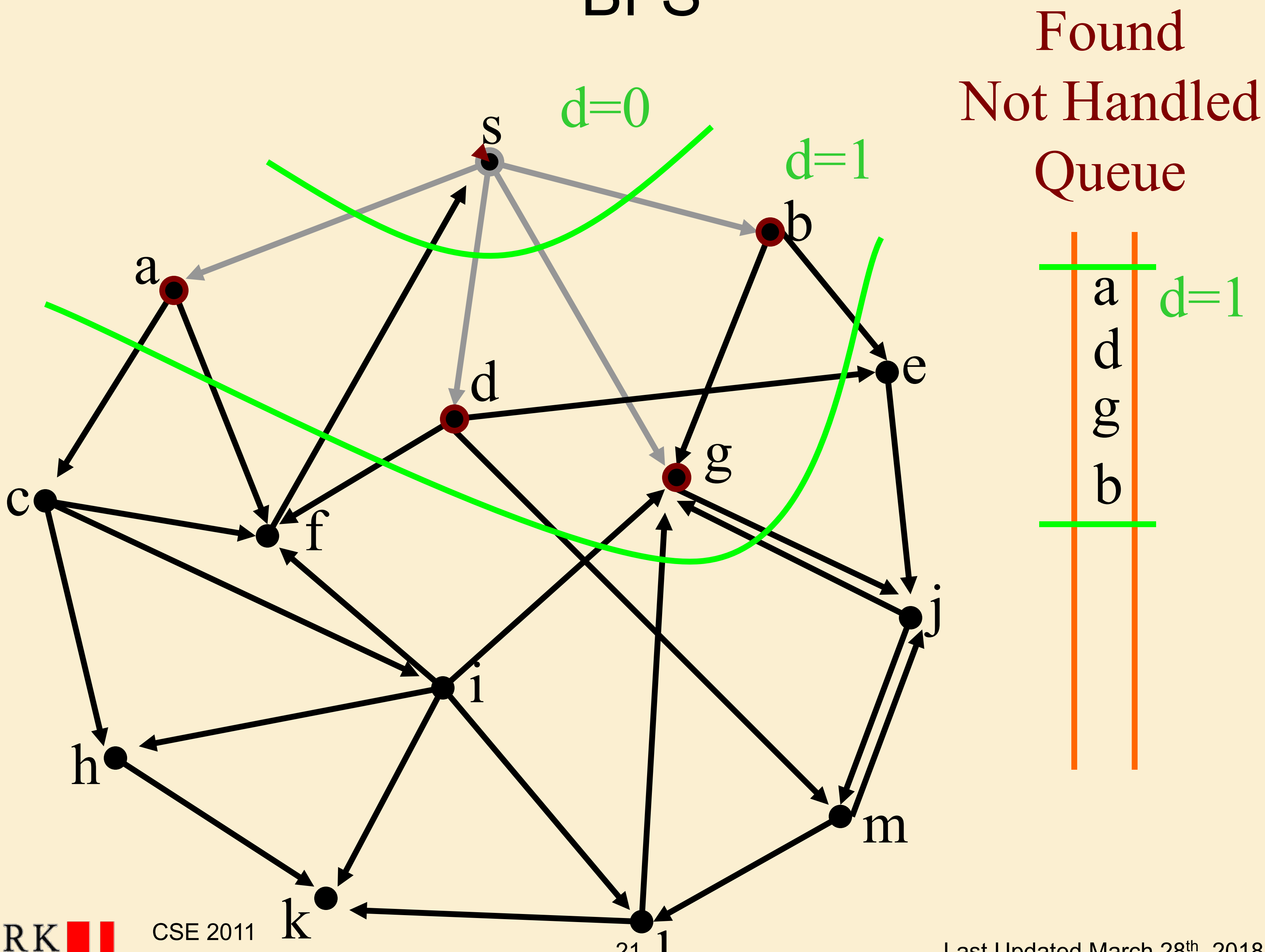
BFS



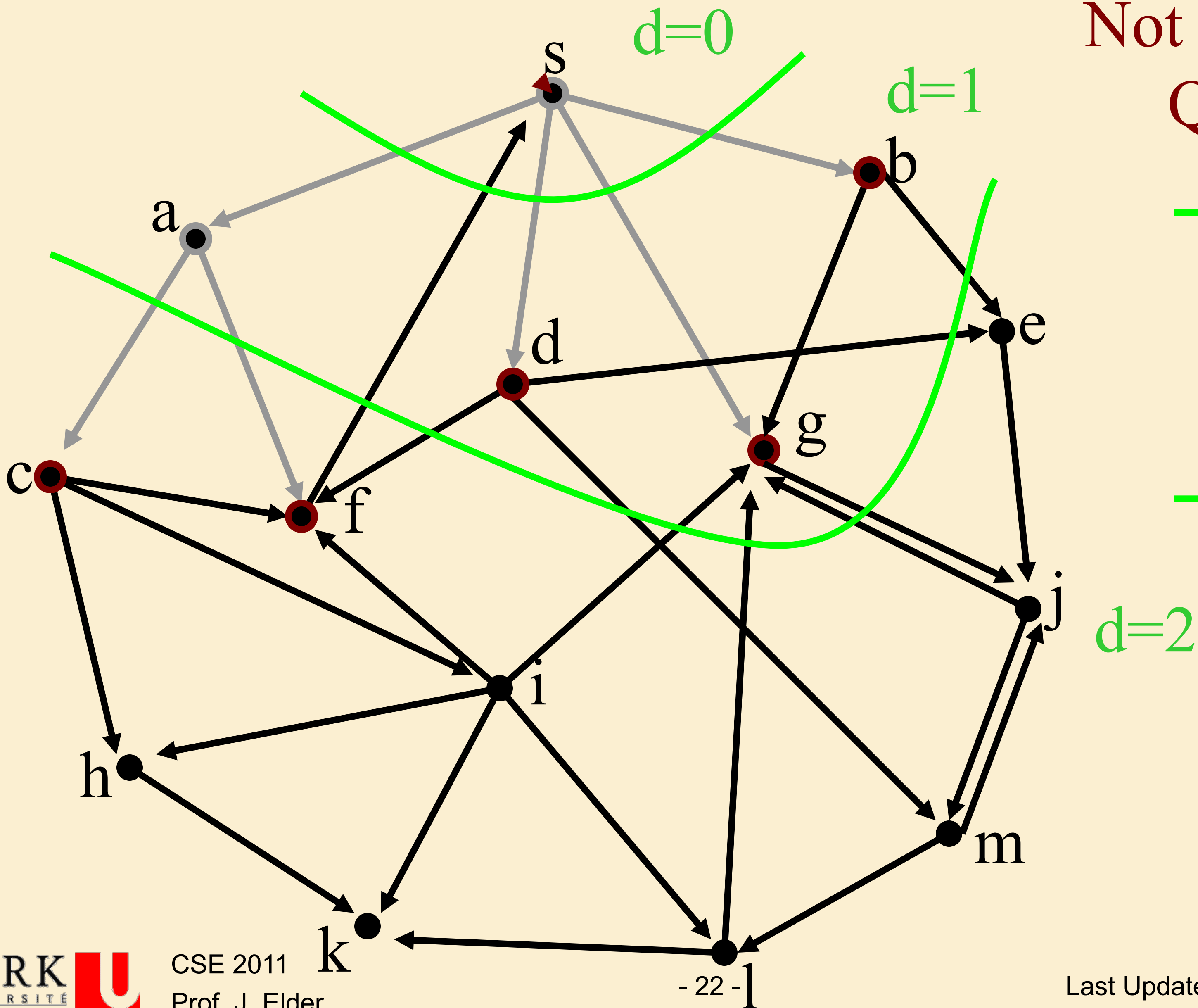
BFS



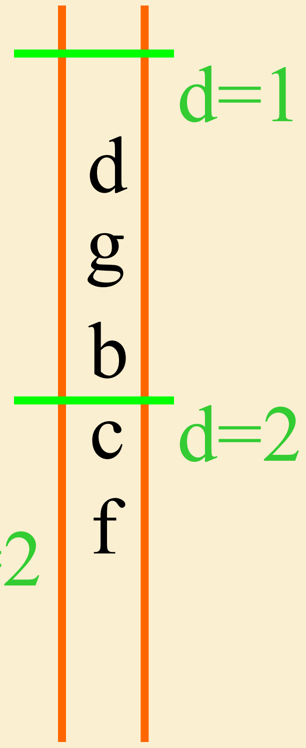
BFS



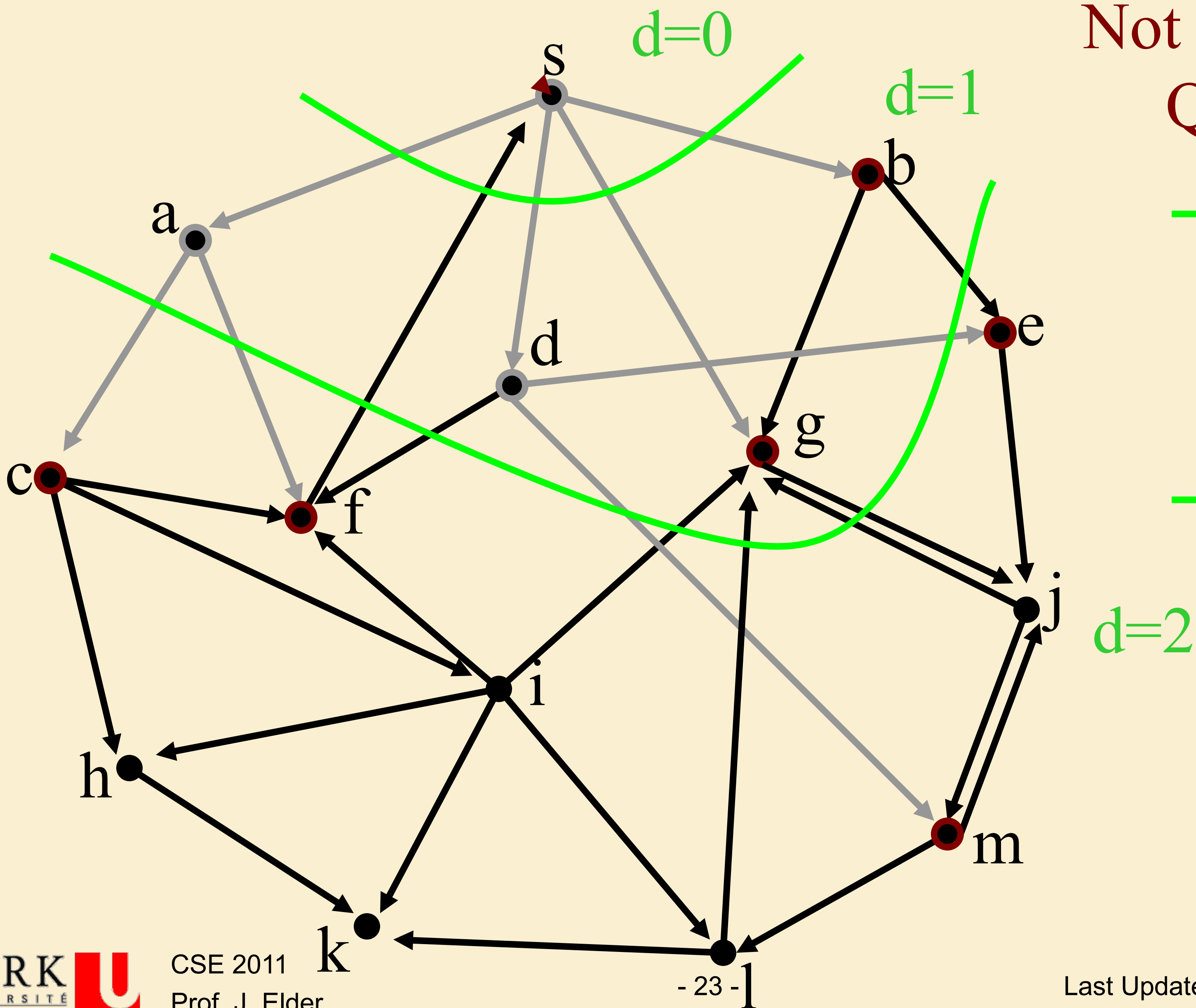
BFS



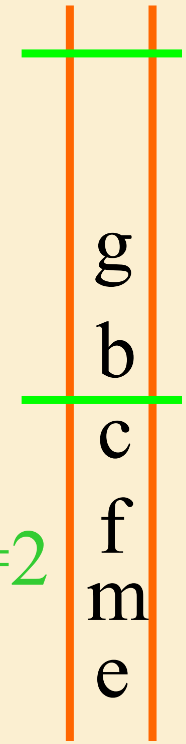
Found
Not Handled
Queue



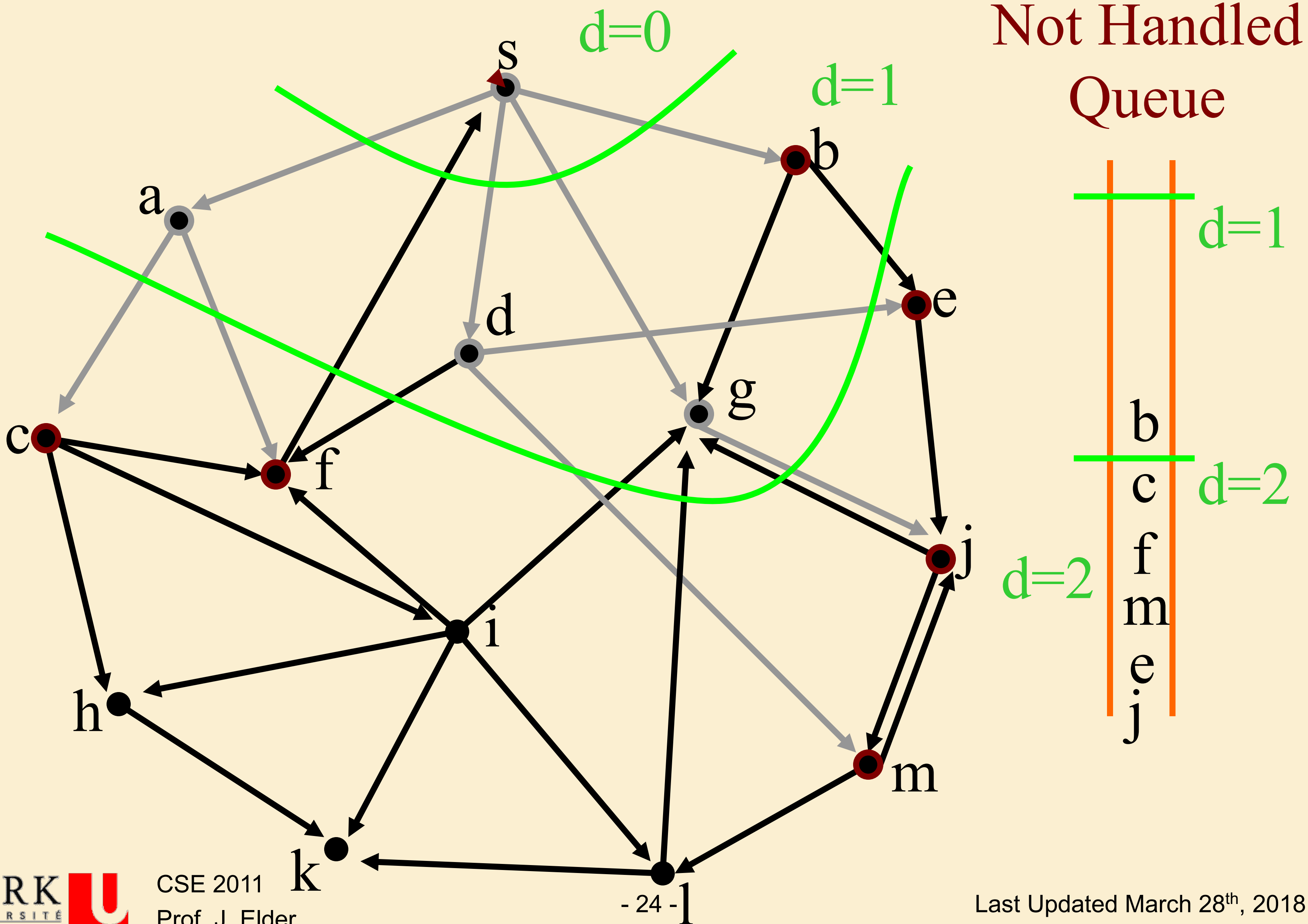
BFS



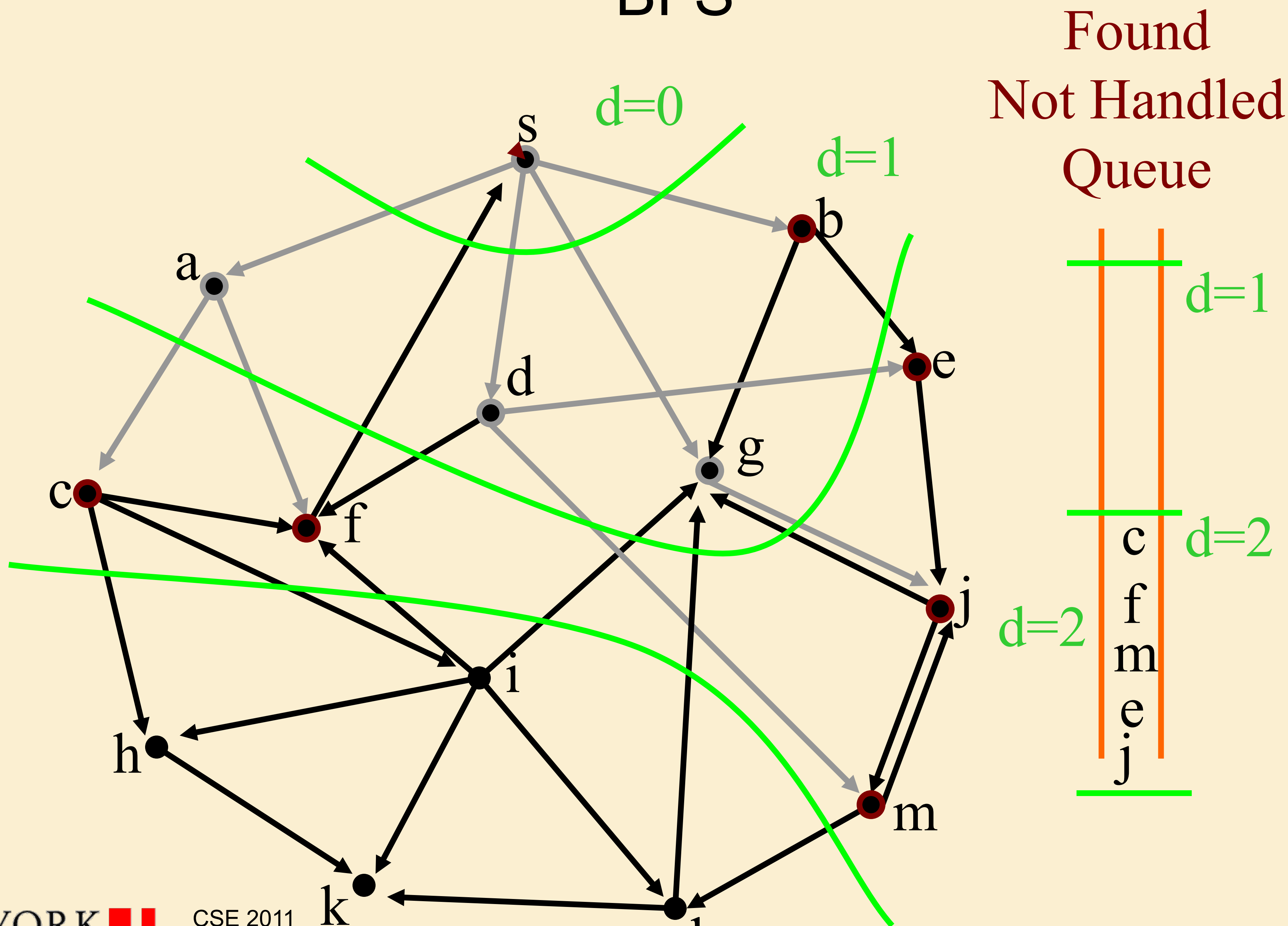
Found
Not Handled
Queue



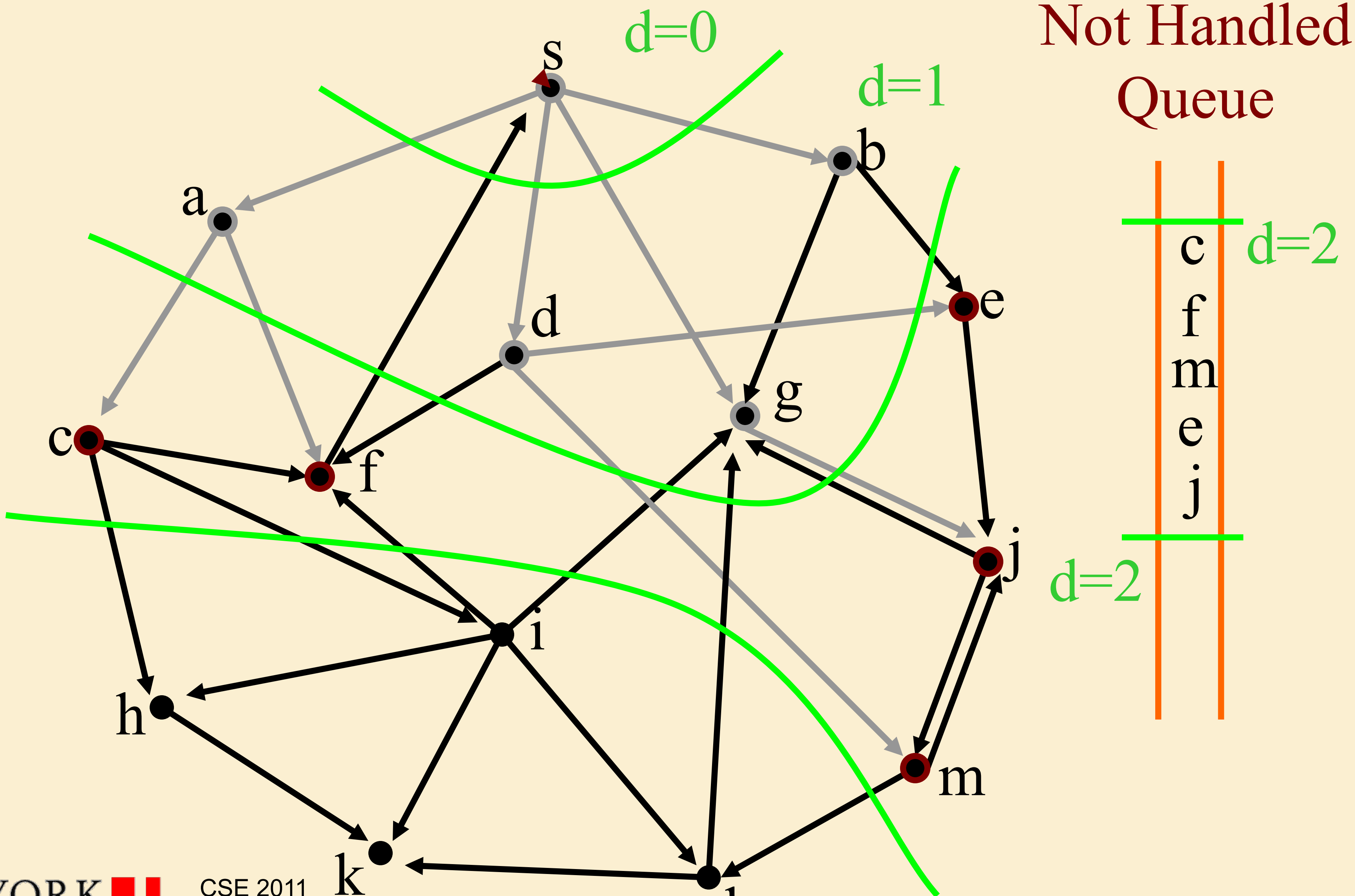
BFS


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

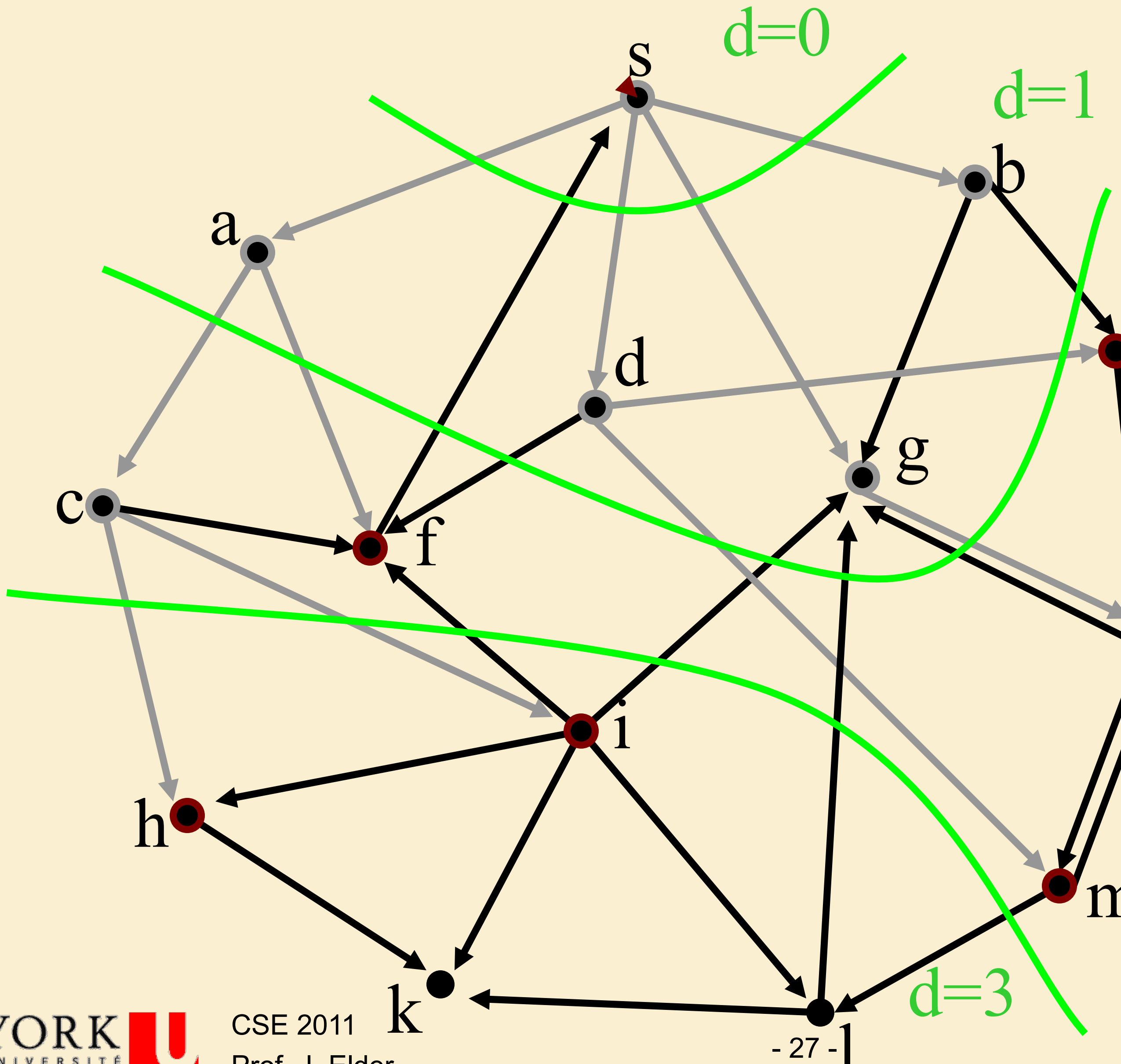
BFS



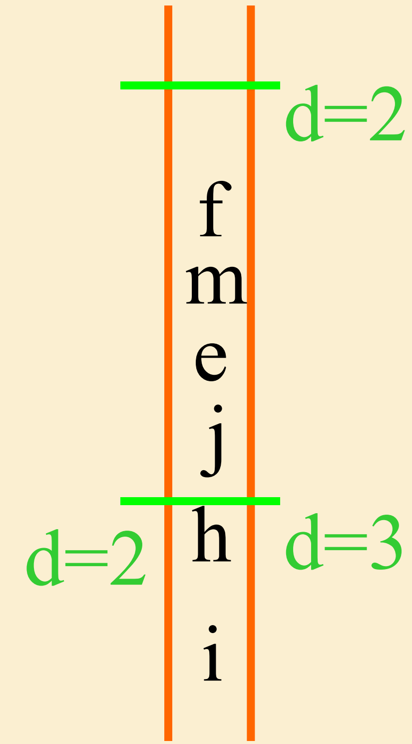
BFS



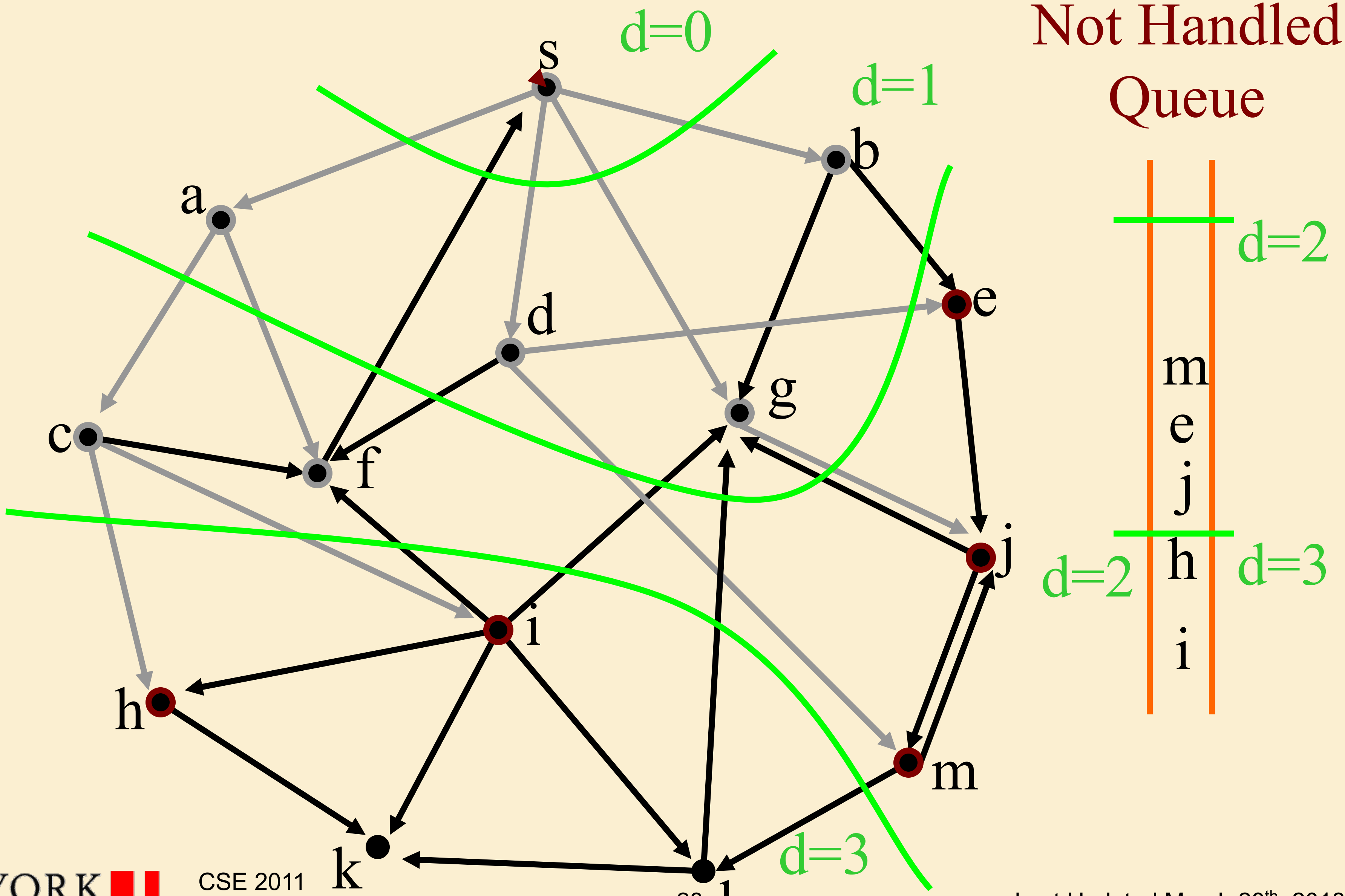
BFS



Found
Not Handled
Queue

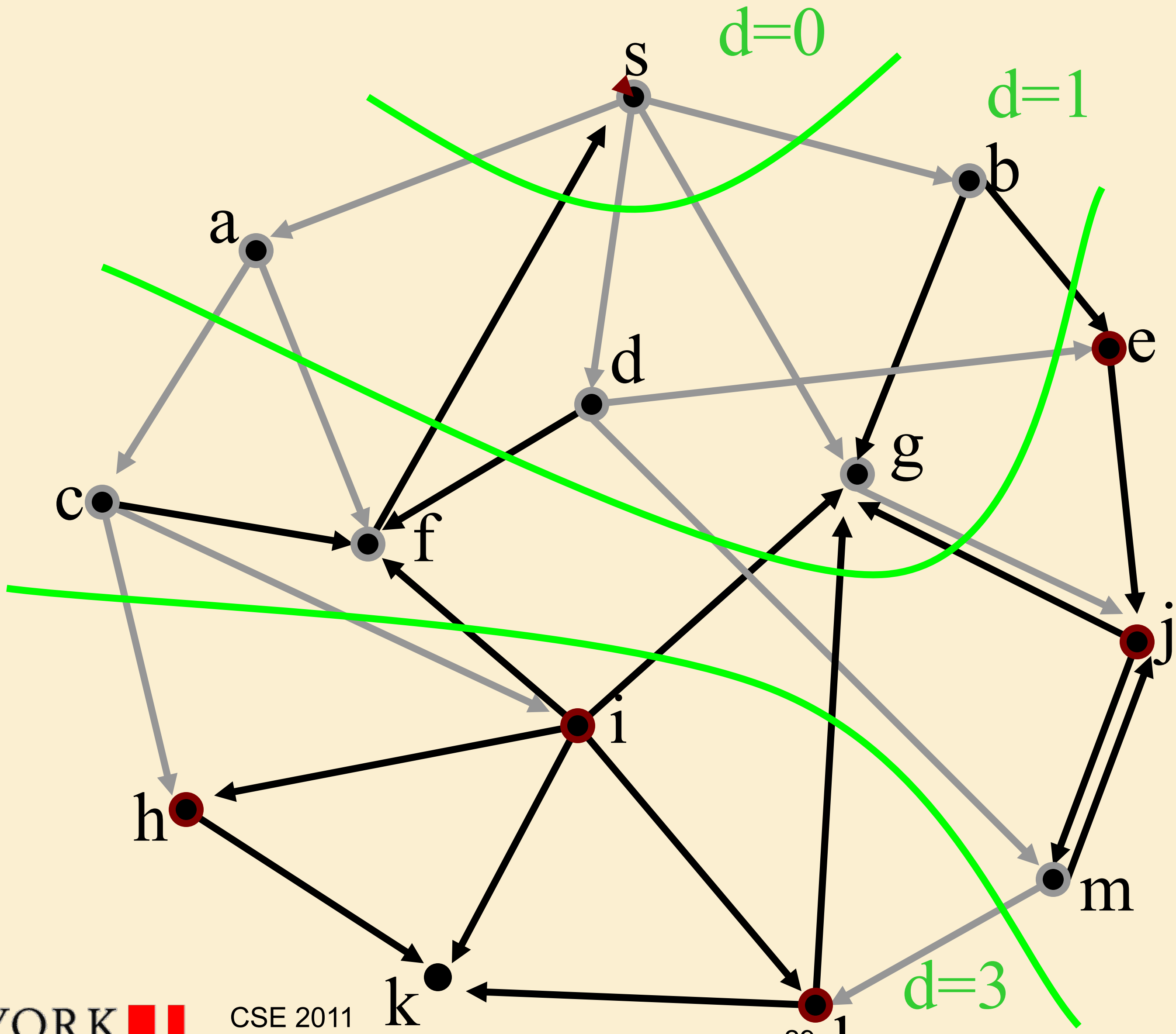


BFS

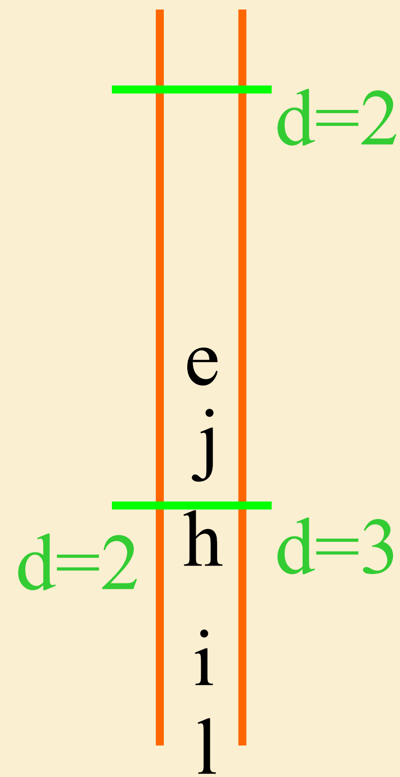


s
a
b
c
d
e
f
g
h
i
j
k
l
m
d=0
d=1
d=2
d=3
Found
Not Handled
Queue
m
e
j
h
i
d=2
d=3

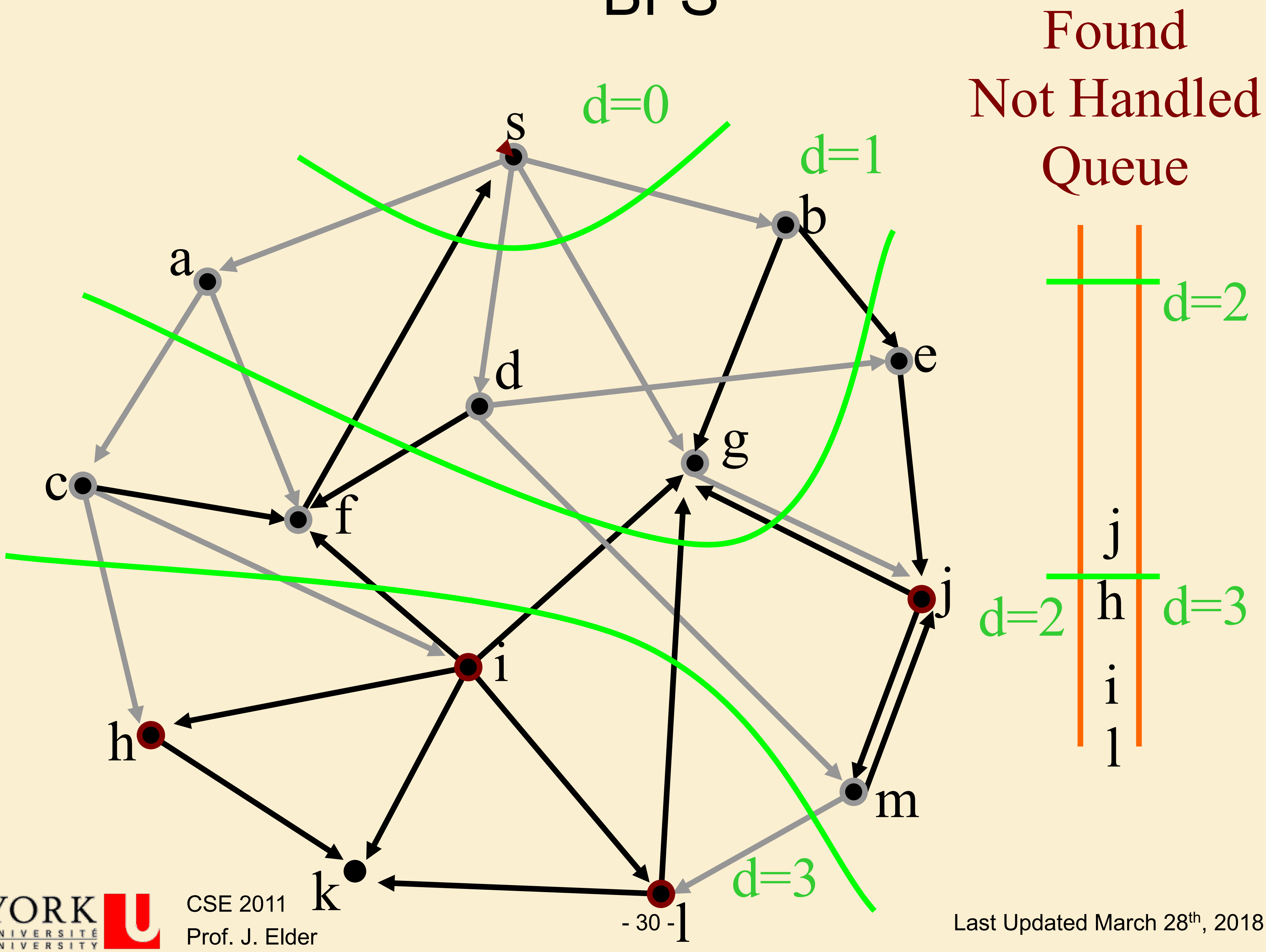
BFS



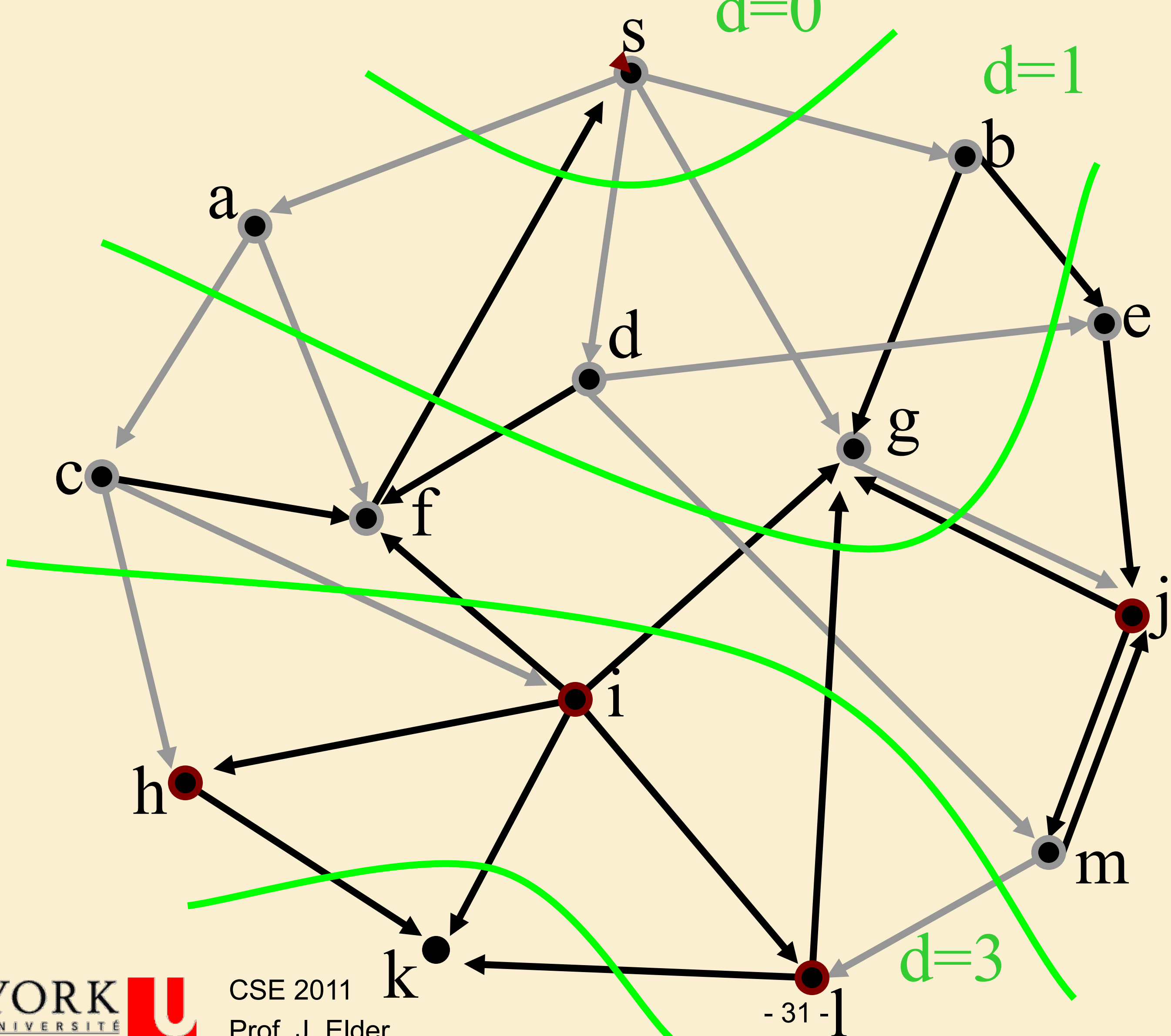
Found
Not Handled
Queue

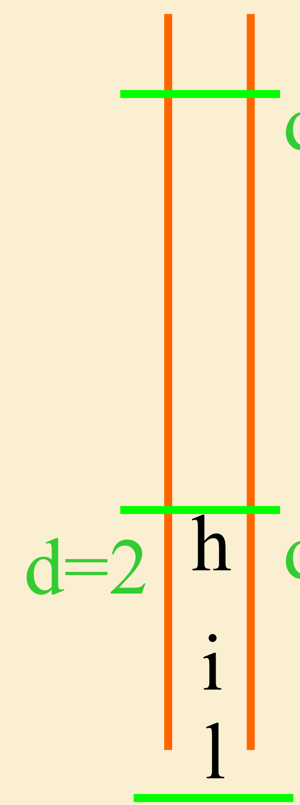


BFS

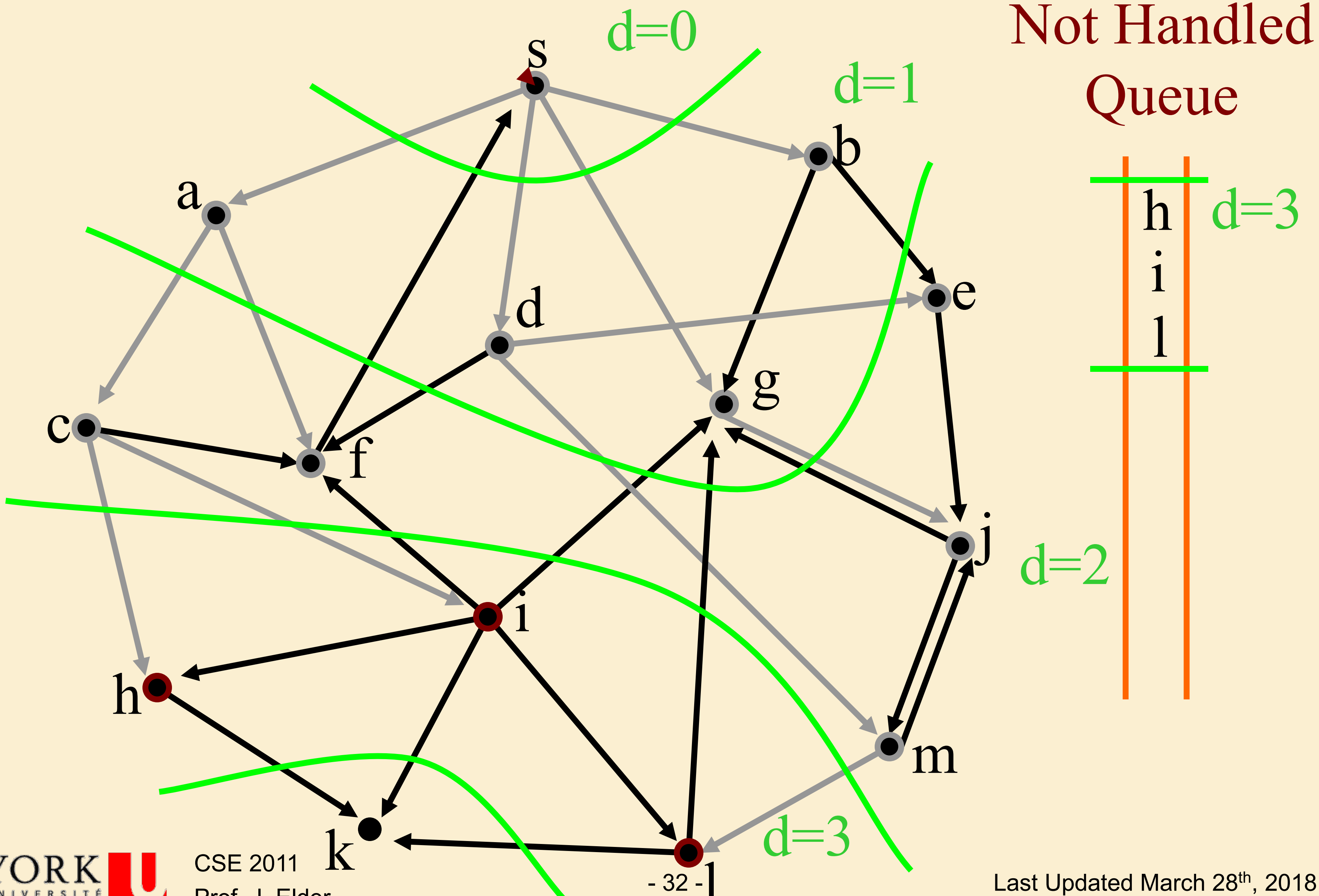

Found
Not Handled
Queue
d=0
d=1
d=2
d=3
s
a
b
c
d
e
f
g
h
i
j
k
l
m
j
h
i
l
d=2
d=3
d=2
d=3

BFS

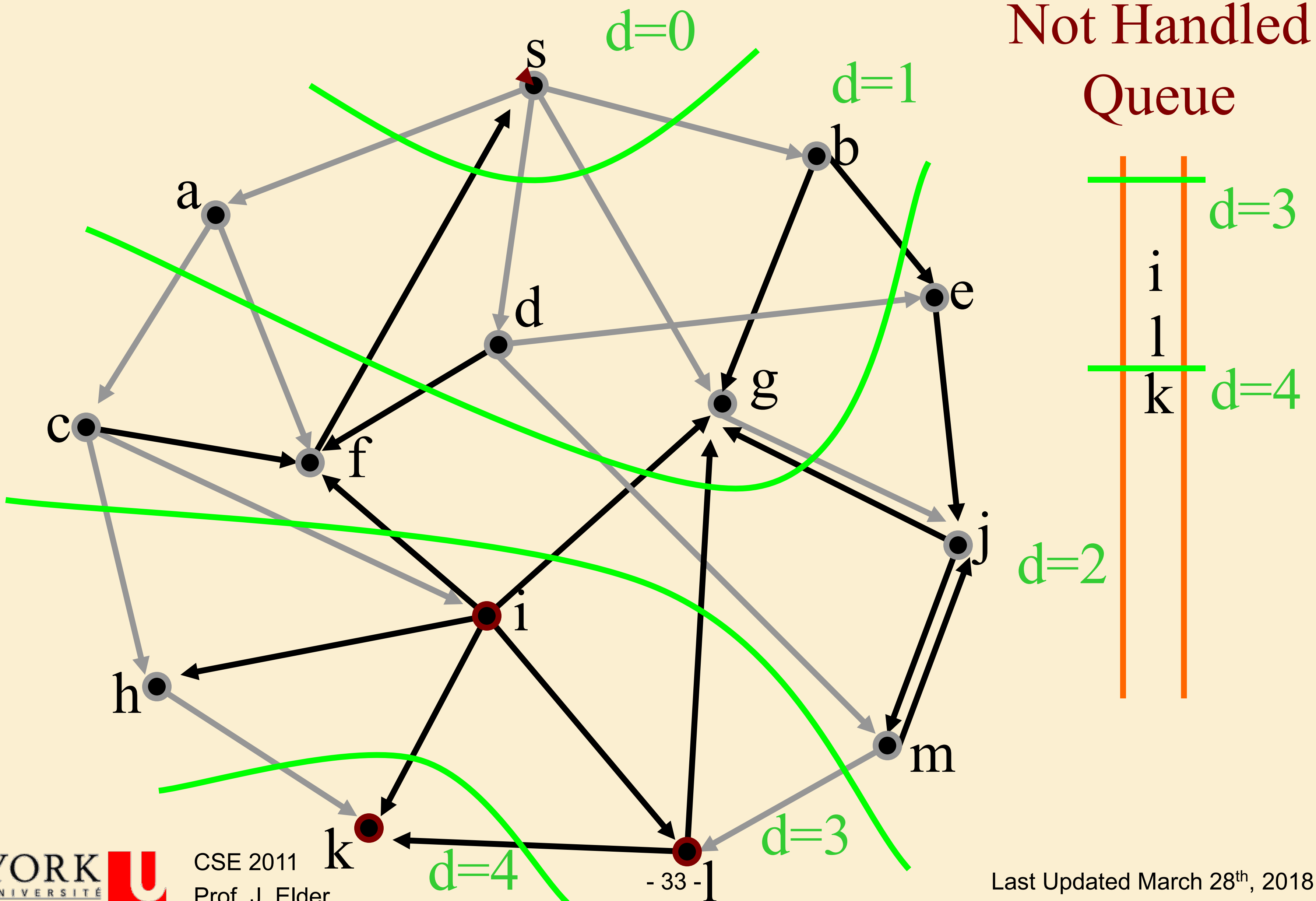




BFS

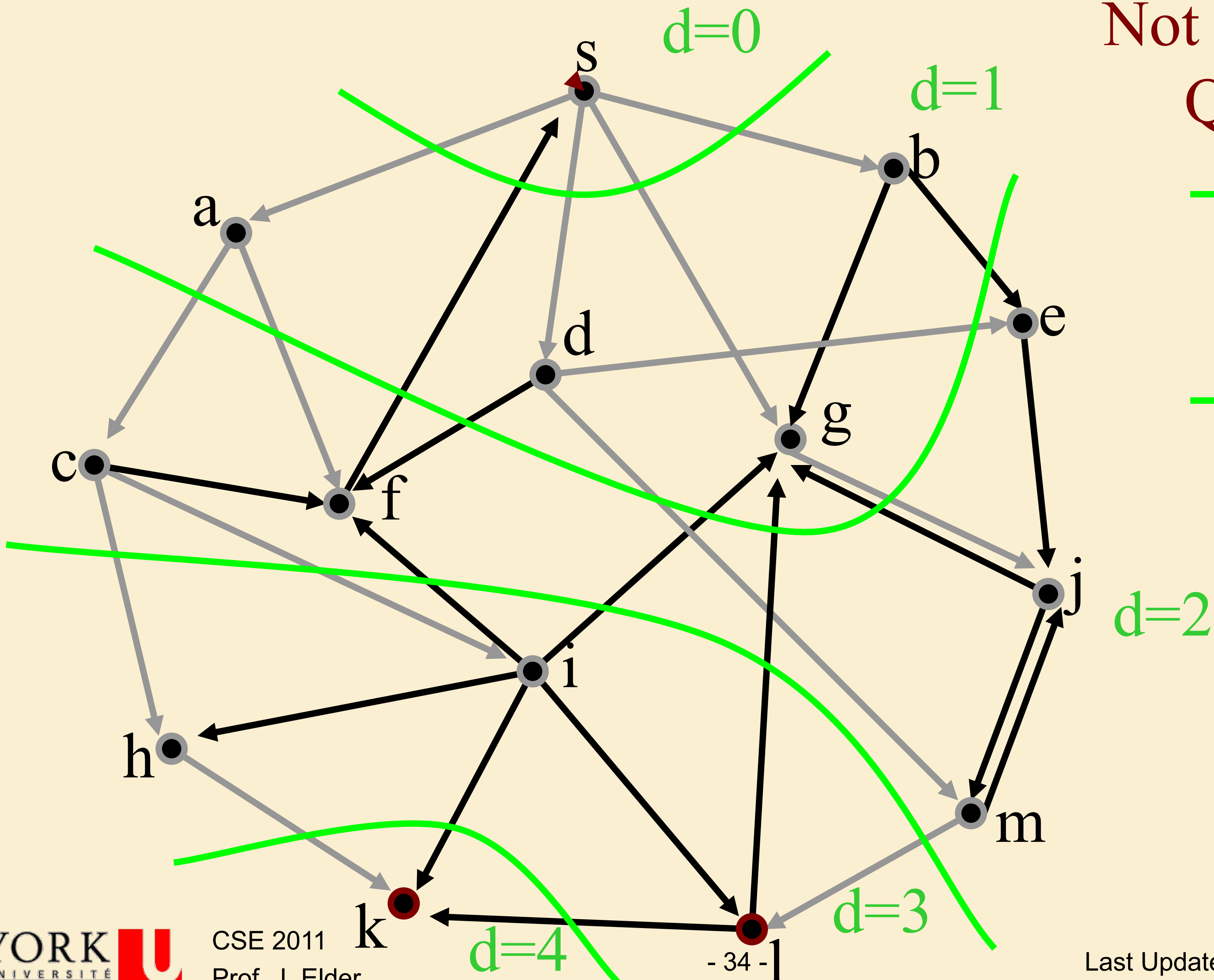


BFS

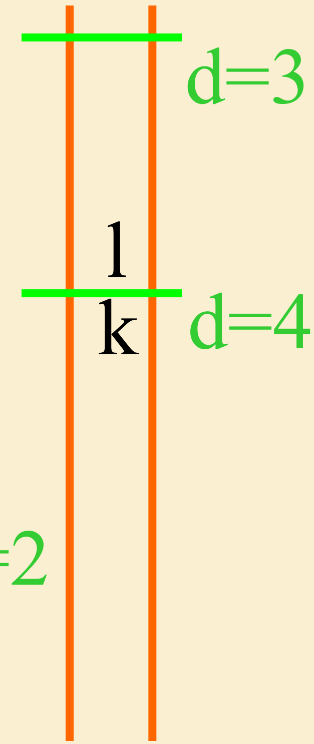


d=0
d=1
s
a
b
d
e
c
f
g
j
i
h
m
k
l
d=2
d=3
d=4
Found
Not Handled
Queue
d=3
i
l
k
d=4

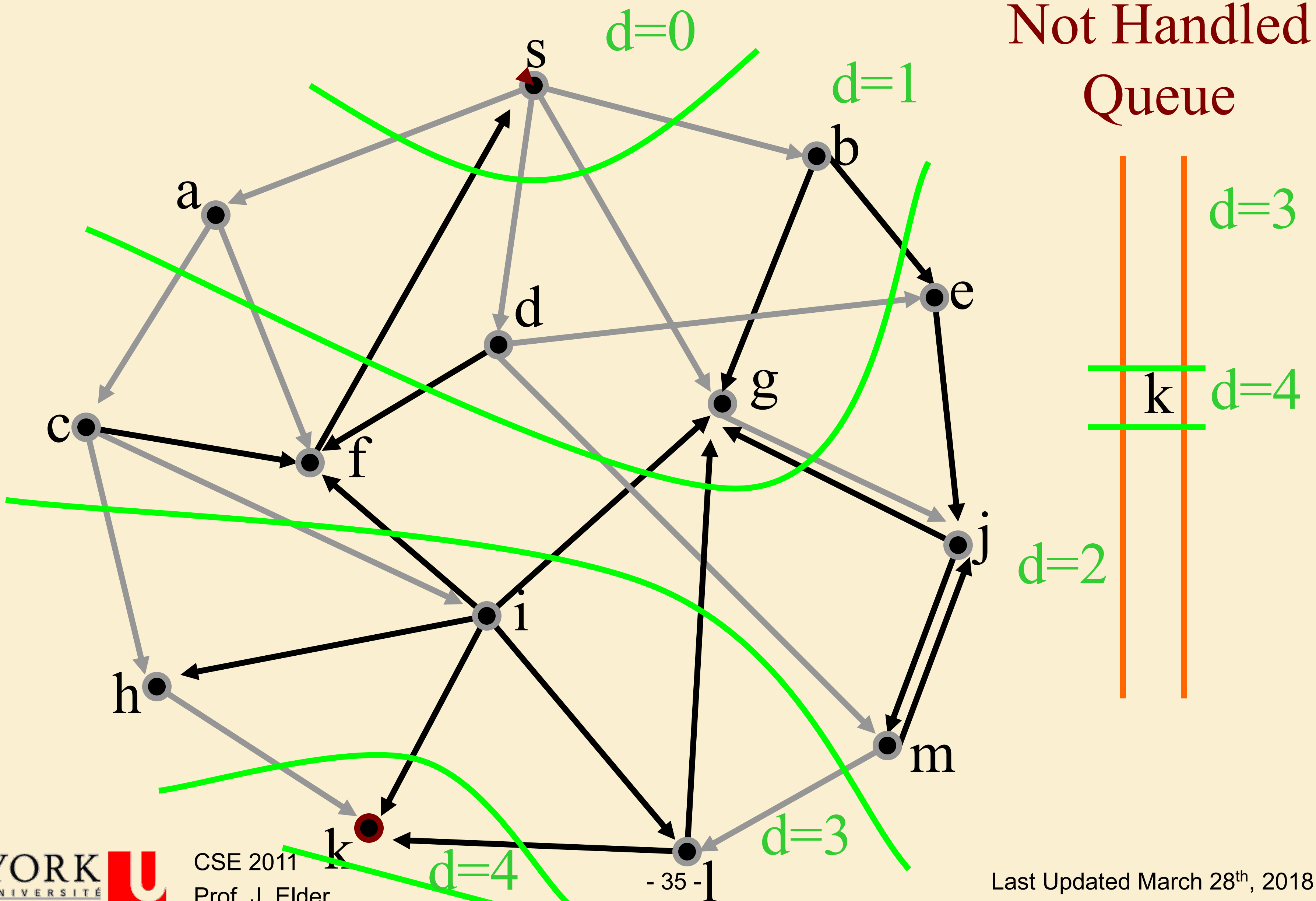
BFS



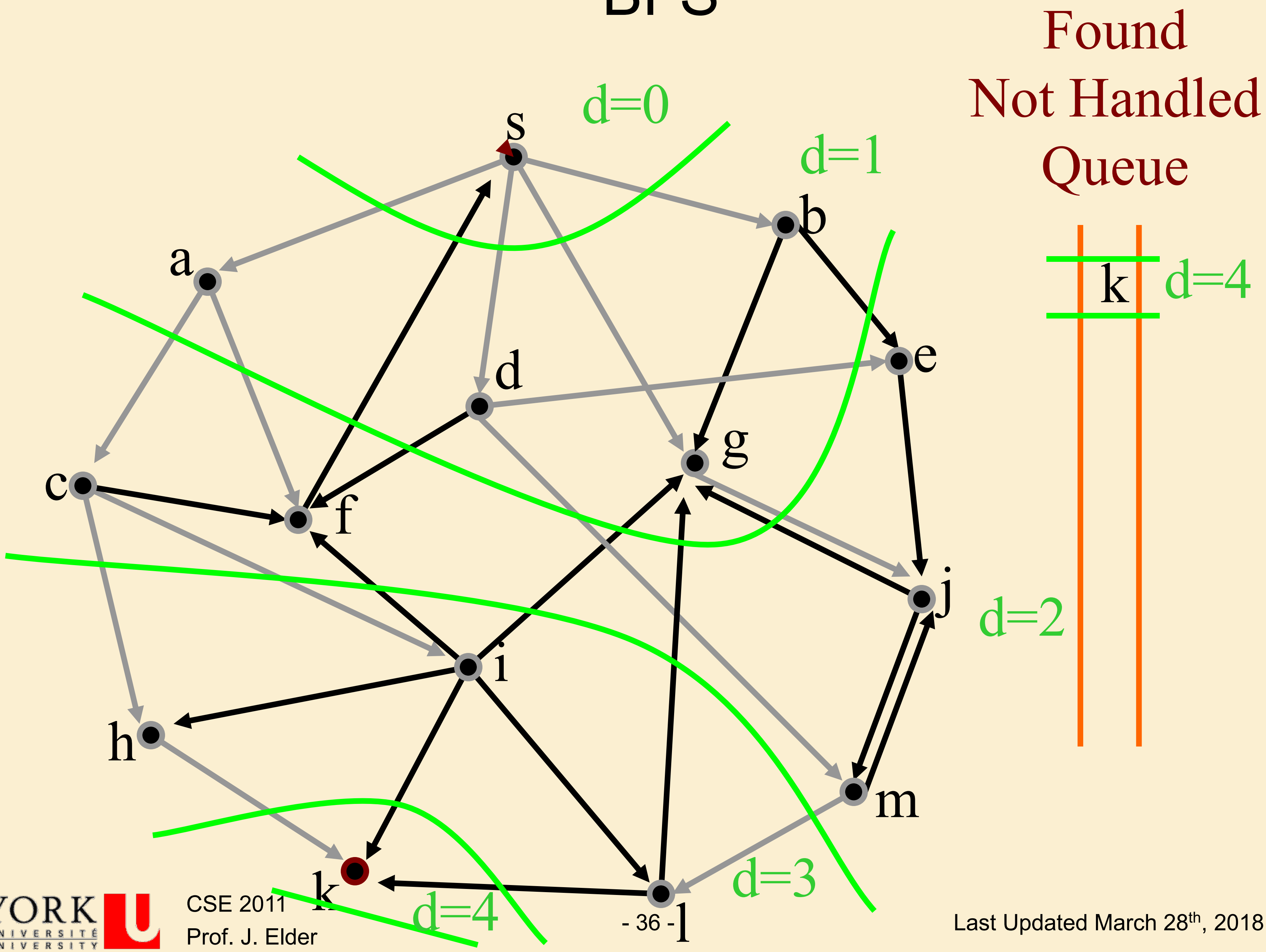
Found
Not Handled
Queue



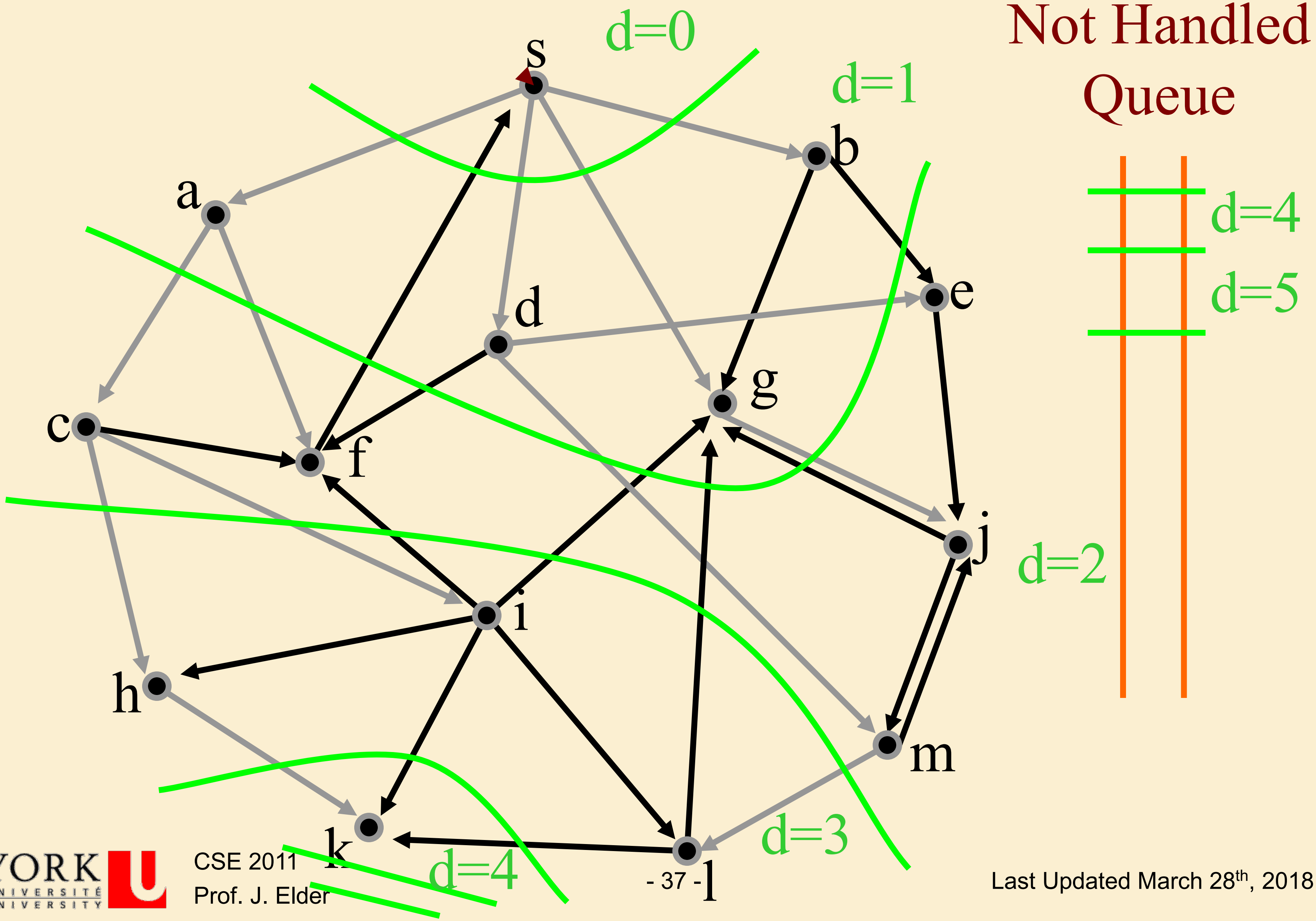
BFS



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*, …, *i*, *i*+1, …, *i*+1}
- *d* values assigned are monotonically increasing over time.

Breadth-First-Search is Greedy

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

Outline

- BFS Algorithm
- BFS Application: Shortest Path on an unweighted graph

Outcomes

- By understanding this lecture, you should be able to:
  - Label a graph according to the order in which vertices are discovered in a breadth-first search.
  - Identify the current state of a breadth-first search in terms of vertices that are previously discovered, just discovered or undiscovered.
  - Identify the contents of the breadth-first search queue at any state of the search.
  - Implement breadth-first search
  - Demonstrate simple applications of breadth-first search