Breadth First Search

CSE 2011

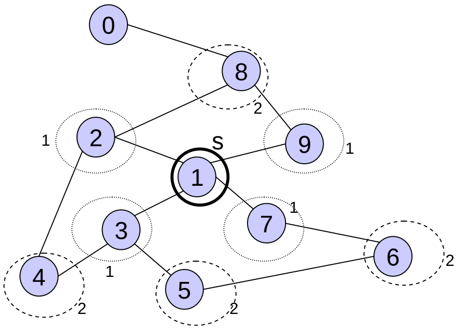
Winter 2011

Graph Traversal (13.3)

- Application example
  - Given a graph representation and a vertex **s** in the graph, find all paths from **s** to the other vertices.
- Two common graph traversal algorithms:
  - Breadth-First Search (BFS)
    - Idea is similar to level-order traversal for trees.
    - Implementation uses a queue.
    - Gives shortest path from a vertex to another.
  - Depth-First Search (DFS)
    - Idea is similar to preorder traversal for trees (visit a node then visit its children recursively).
    - Implementation uses a stack (implicitly via recursion).

BFS and Shortest Path Problem

- Given any source vertex ***s***, BFS visits the other vertices at increasing distances away from ***s***. In doing so, BFS discovers shortest paths from ***s*** to the other vertices.
- What do we mean by “distance”? The number of edges on a path from ***s*** (unweighted graph).



Example

Consider s=vertex 1

Nodes at distance 1?
2, 3, 7, 9

Nodes at distance 2?
8, 6, 5, 4

Nodes at distance 3?
0

How Does BSF Work?

- Similarly to level-order traversal for trees.
- Code: similar to code of topological sort.
  - *flag*[*v*] = false: we have not visited *v*
  - *flag*[*v*] = true: we already visited *v*
- The BFS code we will discuss works for both directed and undirected graphs.

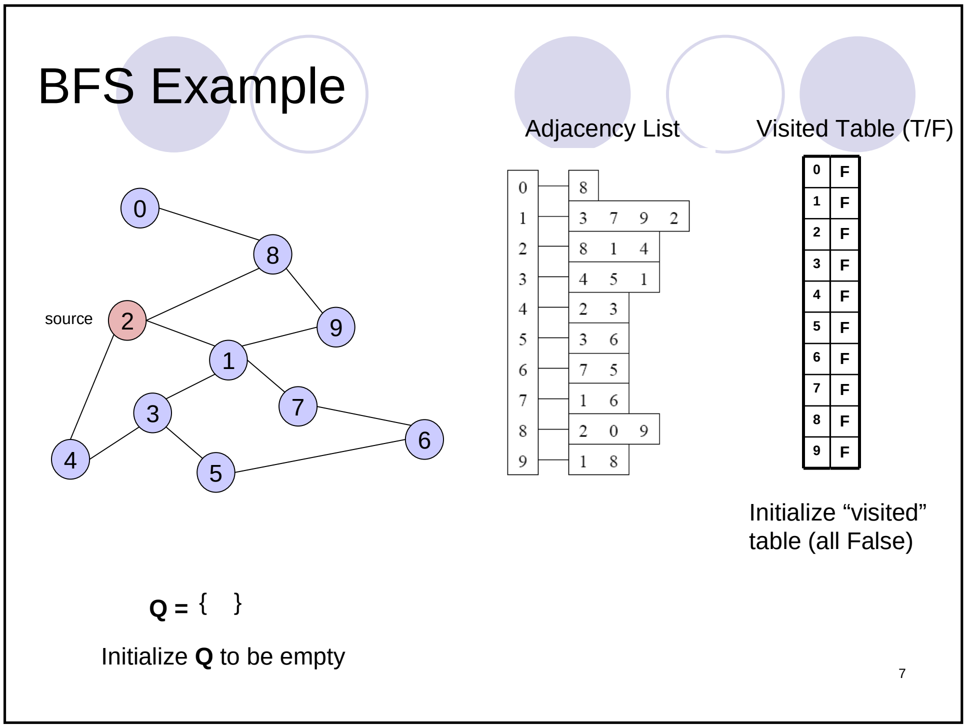
Skeleton of BFS Algorithm

```
Algorithm $BFS(s)$
Input: $s$ is the source vertex
Output: Mark all vertices that can be visited from $s$.
$Q$ = empty queue;
$enqueue(Q, s)$;
while $Q$ is not empty
    do $v := dequeue(Q)$;   output $v$;
        for each $w$ adjacent to $v$
            $enqueue(Q, w)$
```

BFS Algorithm

```
Algorithm $BFS(s)$
Input: $s$ is the source vertex
Output: Mark all vertices that can be visited from $s$.
for each vertex $v$
    do $flag[v]$ := false;
$Q$ = empty queue;
$flag[s]$ := true;
$enqueue(Q, s)$;
while $Q$ is not empty
    do $v := dequeue(Q)$;   output $v$;
        for each $w$ adjacent to $v$
            do if $flag[w]$ = false
                then $flag[w]$ := true;
                    $enqueue(Q, w)$
```

flag[ ]: visited or not



BFS Example

Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

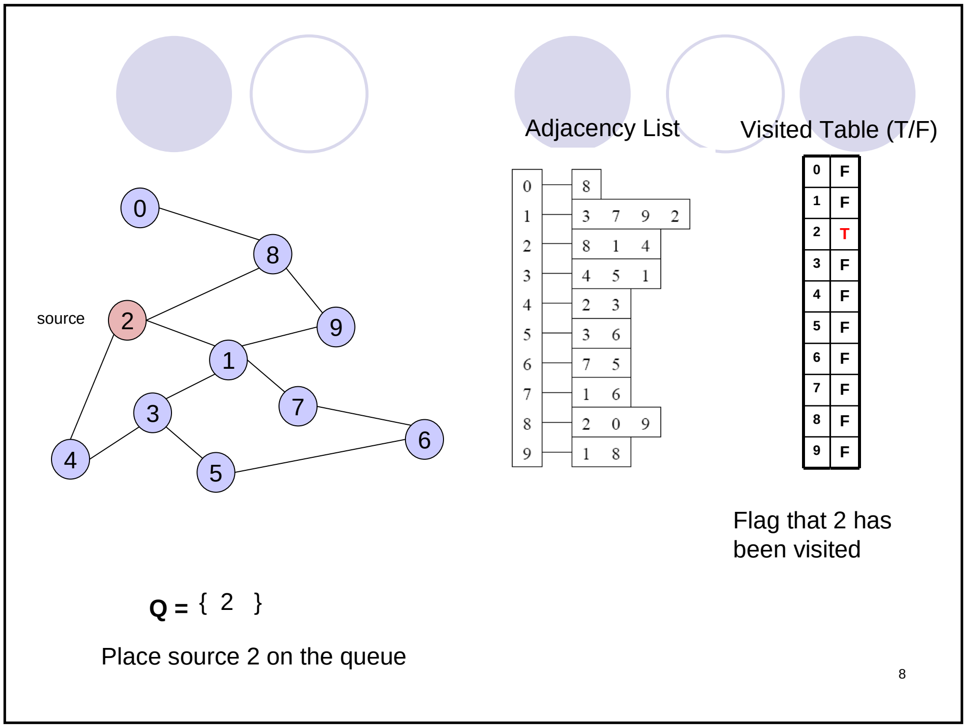
Visited Table (T/F)

0	F
1	F
2	F
3	F
4	F
5	F
6	F
7	F
8	F
9	F

Initialize “visited” table (all False)

Q = { }

Initialize **Q** to be empty



Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

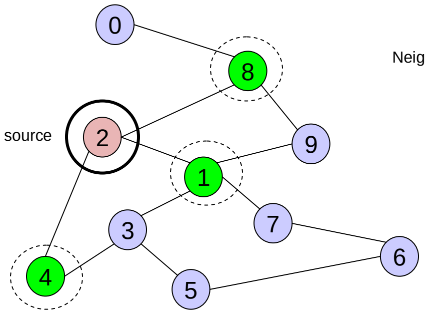
Visited Table (T/F)

0	F
1	F
2	T
3	F
4	F
5	F
6	F
7	F
8	F
9	F

Flag that 2 has been visited

Q = { 2 }

Place source 2 on the queue



Adjacency List

Neighbors → 2

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

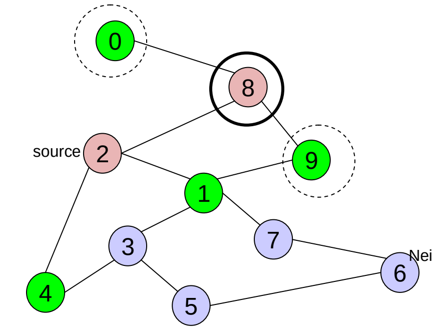
Visited Table (T/F)

0	F
1	T
2	T
3	F
4	T
5	F
6	F
7	F
8	T
9	F

Mark neighbors
as visited 1, 4, 8

Q = {2} → { 8, 1, 4 }

Dequeue 2.
Place all unvisited neighbors of 2 on the queue



Adjacency List

Neighbors → 8

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

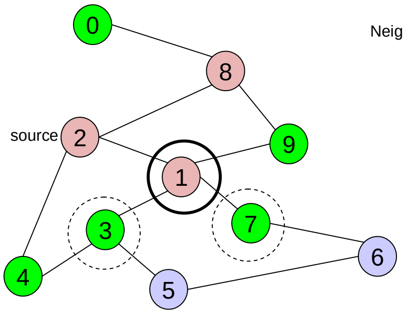
Visited Table (T/F)

0	T
1	T
2	T
3	F
4	T
5	F
6	F
7	F
8	T
9	T

Mark newly visited
neighbors 0, 9

Q = { 8, 1, 4 } → { 1, 4, 0, 9 }

Dequeue 8.
-- Place all unvisited neighbors of 8 on the queue.
-- Notice that 2 is not placed on the queue again, it has been visited!



Adjacency List

Neighbors → 1

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

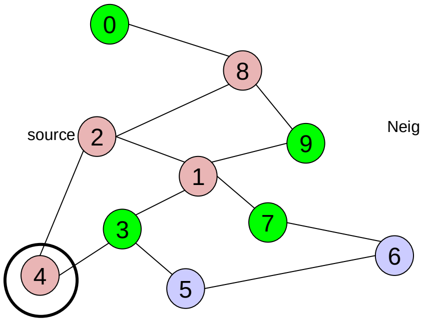
Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

Mark newly visited neighbors 3, 7

Q = { 1, 4, 0, 9 } → { 4, 0, 9, 3, 7 }

Dequeue 1.
-- Place all unvisited neighbors of 1 on the queue.
-- Only nodes 3 and 7 haven't been visited yet.



Adjacency List

Neighbors → 4

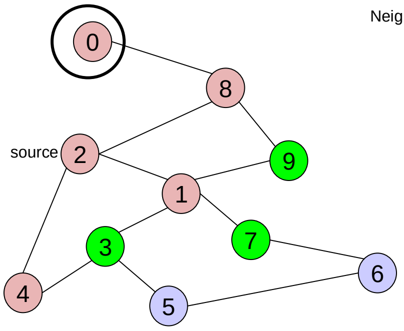
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

Q = { 4, 0, 9, 3, 7 } → { 0, 9, 3, 7 }

Dequeue 4.
-- 4 has no unvisited neighbors!



Adjacency List

Neighbors→

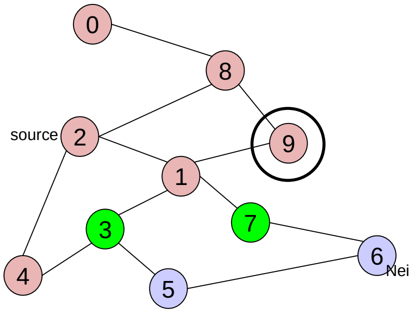
0	8			
1	3	7	9	2
2	8	1	4	
3	4	5	1	
4	2	3		
5	3	6		
6	7	5		
7	1	6		
8	2	0	9	
9	1	8		

Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

Q = { 0, 9, 3, 7 } → { 9, 3, 7 }

Dequeue 0.
-- 0 has no unvisited neighbors!



Adjacency List

0	8			
1	3	7	9	2
2	8	1	4	
3	4	5	1	
4	2	3		
5	3	6		
6	7	5		
7	1	6		
8	2	0	9	
9	1	8		

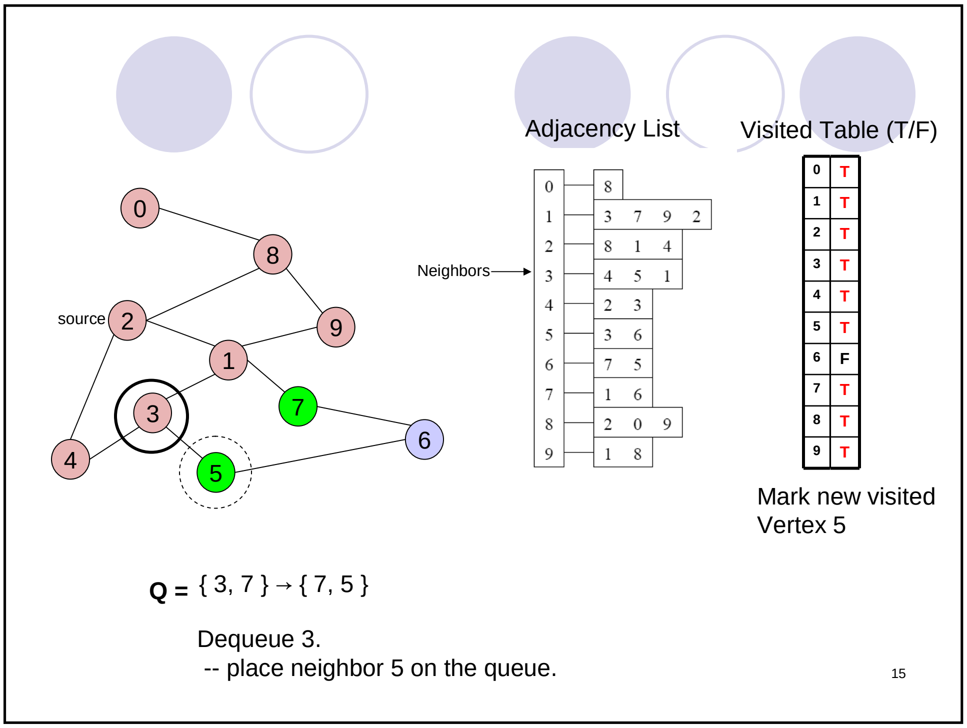
Neighbors→

Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

Q = { 9, 3, 7 } → { 3, 7 }

Dequeue 9.
-- 9 has no unvisited neighbors!



Adjacency List

0	8			
1	3	7	9	2
2	8	1	4	
3	4	5	1	
4	2	3		
5	3	6		
6	7	5		
7	1	6		
8	2	0	9	
9	1	8		

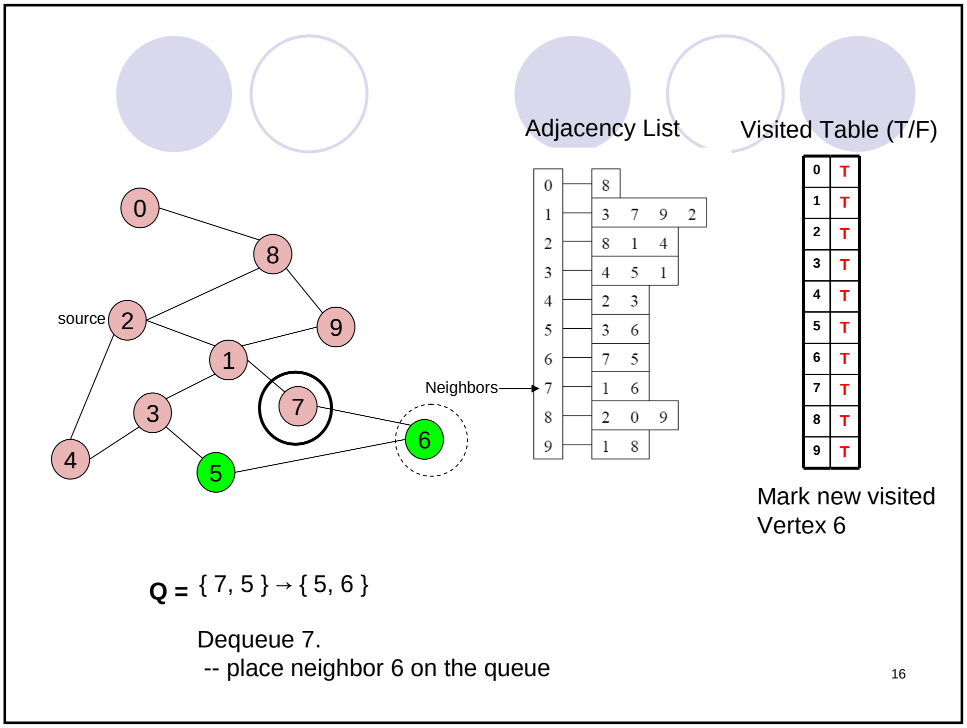
Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	F
7	T
8	T
9	T

Mark new visited
Vertex 5

Q = { 3, 7 } → { 7, 5 }

Dequeue 3.
-- place neighbor 5 on the queue.



Adjacency List

0	8			
1	3	7	9	2
2	8	1	4	
3	4	5	1	
4	2	3		
5	3	6		
6	7	5		
7	1	6		
8	2	0	9	
9	1	8		

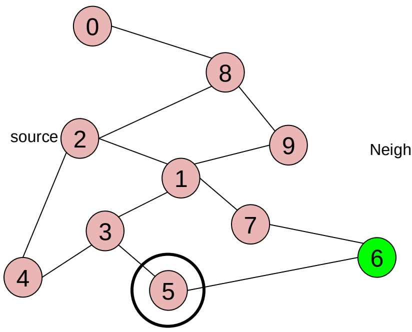
Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Mark new visited
Vertex 6

Q = { 7, 5 } → { 5, 6 }

Dequeue 7.
-- place neighbor 6 on the queue



Adjacency List

Neighbors → 5

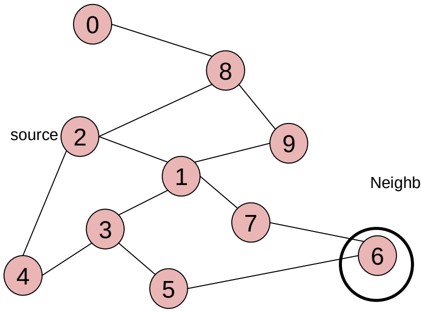
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Q = { 5, 6} → { 6 }

Dequeue 5.
-- no unvisited neighbors of 5



Adjacency List

Neighbors → 6

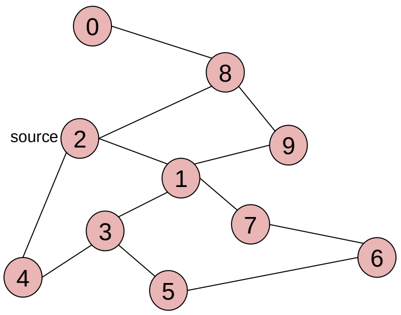
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Q = { 6 } → { }

Dequeue 6.
-- no unvisited neighbors of 6



Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Q = { } **STOP!!! Q is empty!!!**

What did we discover?

Look at "visited" tables.

There exists a path from source vertex 2 to all vertices in the graph

Running Time of BFS

- Assume adjacency list
  - V = number of vertices; E = number of edges

```
Algorithm BFS($s$)
Input: $s$ is the source vertex
Output: Mark all vertices that can be visited from $s$.
for each vertex $v$
    do $flag[v]$ := false;
$Q$ = empty queue;
$flag[s]$ := true;
enqueue($Q, s$);
while $Q$ is not empty
    do $v$ := dequeue($Q$);
        for each $w$ adjacent to $v$
            do if $flag[w]$ = false
                then $flag[w]$ := true;
                    enqueue($Q, w$)
```

Each vertex will enter *Q* at most once. dequeue is O(1).

The for loop takes time proportional to deg(*v*).

Running Time of BFS (2)

- Recall: Given a graph with E edges

$$\Sigma_{\text{vertex } v} \ \deg(v) = 2E$$

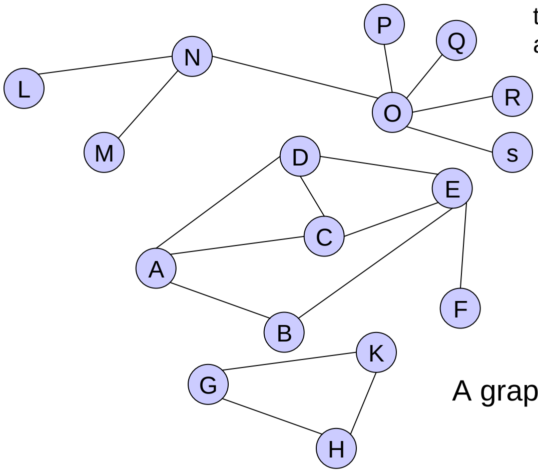
- The total running time of the while loop is:

$$O(\Sigma_{\text{vertex } v} (1 + \deg(v))) = O(V+E)$$

- This is the sum over all the iterations of the while loop!
- Homework: What is the running time of BFS if we use an adjacency matrix?

BFS and Unconnected Graphs

A graph may not be connected (strongly connected) ⇒ enhance the above BFS code to accommodate this case.



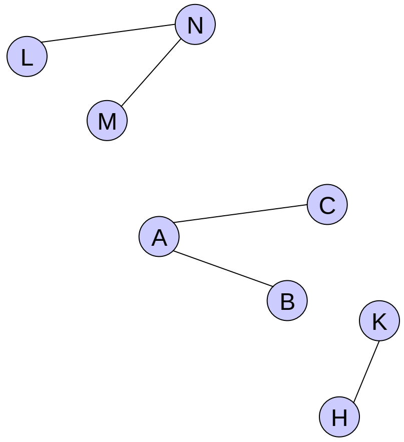
A graph with 3 components

Recall the BFS Algorithm …

```
Algorithm BFS($s$)
Input: $s$ is the source vertex
Output: Mark all vertices that can be visited from $s$.
for each vertex $v$
    do $flag[v]$ := false;
$Q$ = empty queue;
$flag[s]$ := true;
enqueue($Q, s$);
while $Q$ is not empty
    do $v$ := dequeue($Q$);     output ( v );
        for each $w$ adjacent to $v$
            do if $flag[w]$ = false
                then $flag[w]$ := true;
                     enqueue($Q, w$)
```

Enhanced BFS Algorithm

A graph with 3 components



- We can re-use the previous *BFS*(*s*) method to compute the connected components of a graph *G.*

```
BFSearch( G ) {
i = 1;     // component number
for every vertex v
    flag[v] = false;
for every vertex v
    if ( flag[v] == false ) {
        print ( "Component " +  i++ );
        BFS( v );
    }
}
```

Applications of BFS

What can we do with the BFS code we just discussed?

- Is there a path from source *s* to a vertex *v*?
  - Check *flag*[*v*].
- Is an undirected graph connected?
  - Scan array *flag*[ ].
  - If there exists *flag*[*u*] = false then …
- Is a directed graph strongly connected?
  - Scan array *flag*[ ].
  - If there exists *flag*[*u*] = false then …
- To output the contents (e.g., the vertices) of a connected (strongly connected) graph
  - What if the graph is not connected (weakly connected)? Slide 24

Other Applications of BFS

- To find the shortest path from a vertex *s* to a vertex *v* in an unweighted graph
- To find the length of such a path
- To find out if a graph contains cycles
- To find the connected components of a graph that is not connected (slide 24)
- To construct a BSF tree/forest from a graph

Next time …

- Depth First Search (DFS)

- Review
- Final exam