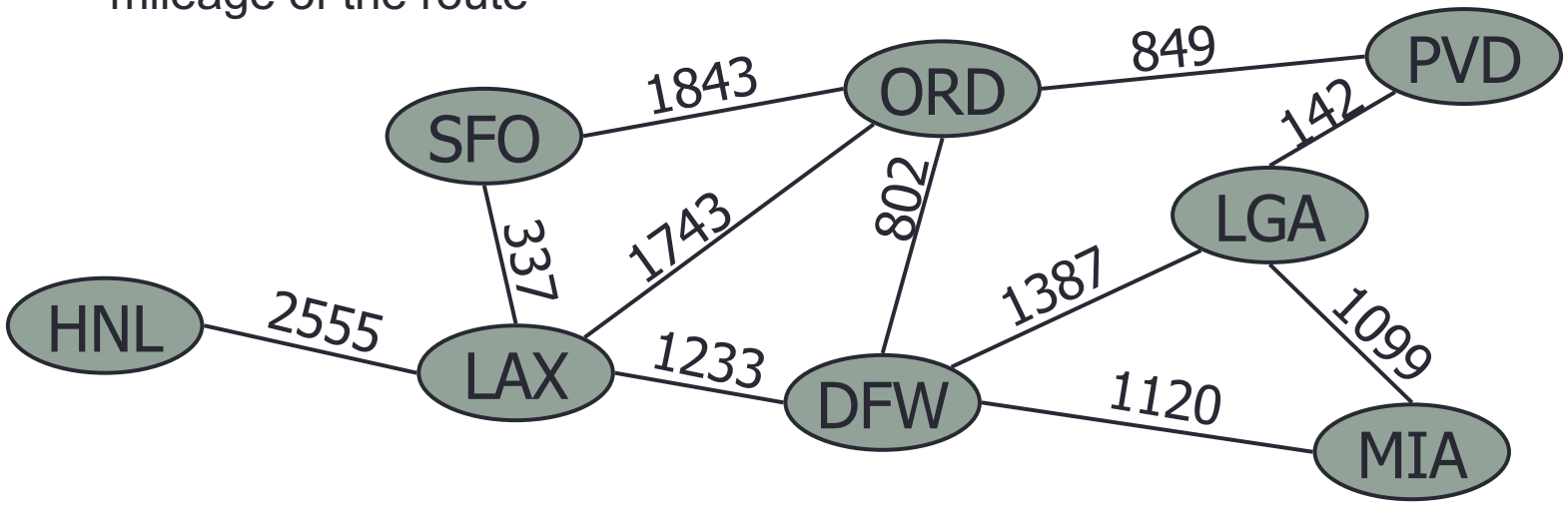
GRAPHS

EECS 2011

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 objects and store elements
- Example:
  - A vertex represents an airport and stores the three-letter airport code
  - An edge represents a flight route between two airports and stores the mileage of the route

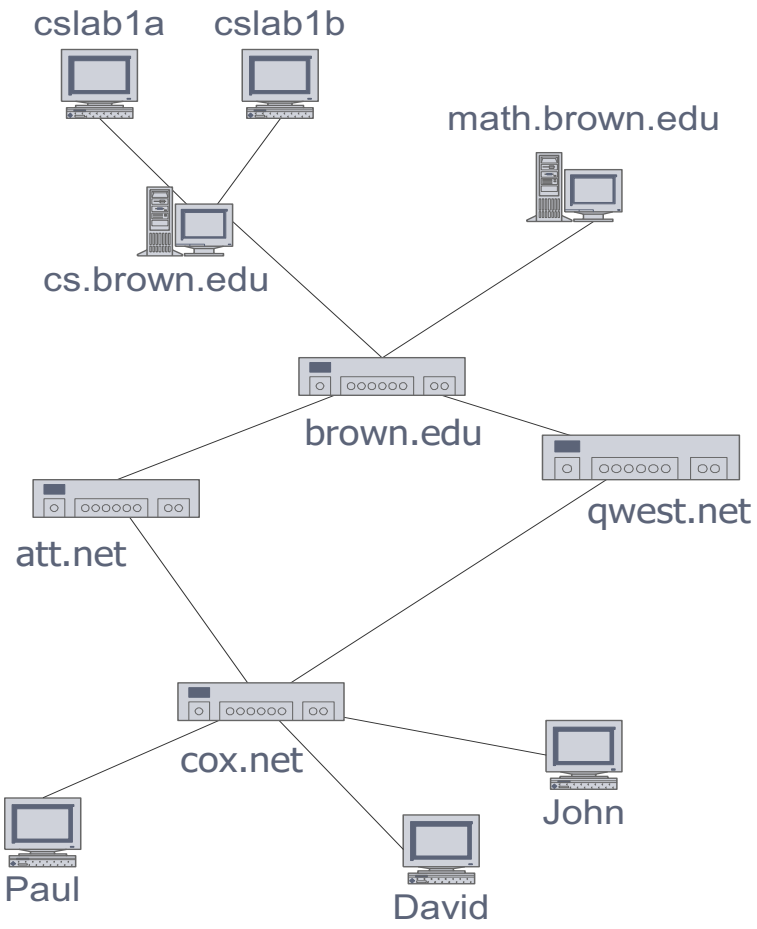


Edge Types

- Directed edge
  - ordered pair of vertices (u,v)
  - first vertex u is the origin
  - second vertex v is the destination
  - e.g., a flight
- Undirected edge
  - unordered pair of vertices (u,v)
  - e.g., a flight route
- Directed graph (digraph)
  - all the edges are directed
  - e.g., flight network
- Undirected graph
  - all the edges are undirected
  - e.g., route network

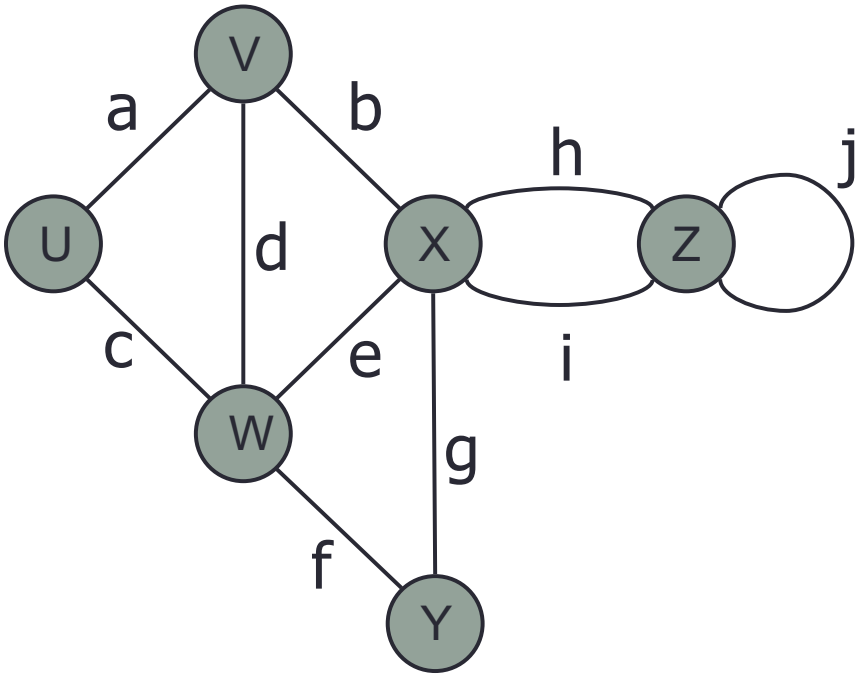
Applications

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



Terminology

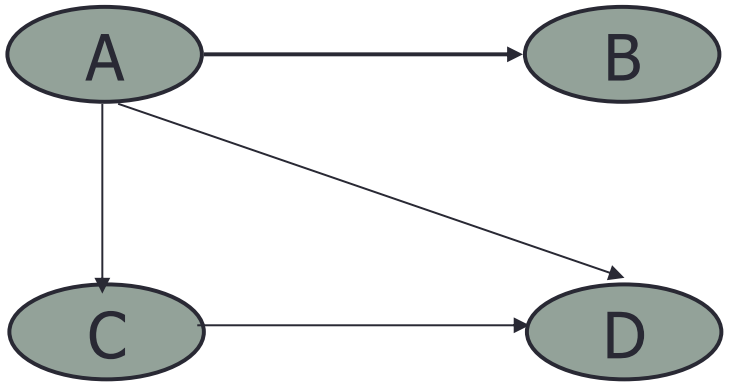
- End vertices (or endpoints) of an edge
  - U and V are the endpoints of a
- Edges incident on a vertex
  - a, d, and b are incident on V
- Adjacent vertices
  - U and V are adjacent
- Degree of a vertex
  - X has degree 5
- Parallel edges
  - h and i are parallel edges
- Self-loop
  - j is a self-loop



Terminology: Directed Graphs

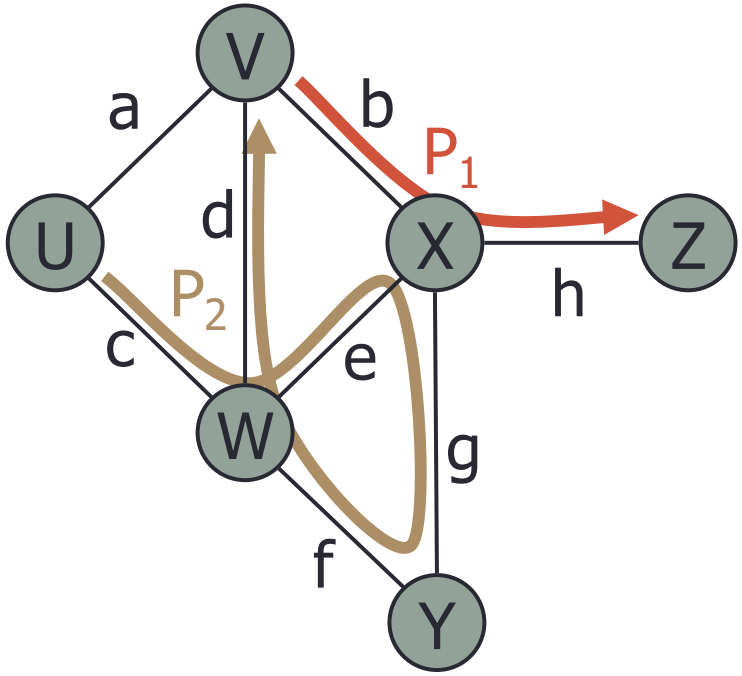
For directed graphs:

- Origin of an edge
- Destination of an edge
- Outgoing edge
- Incoming edge
- Out-degree of vertex v: number of outgoing edges of v
- In-degree of vertex v: number of incoming edges of v



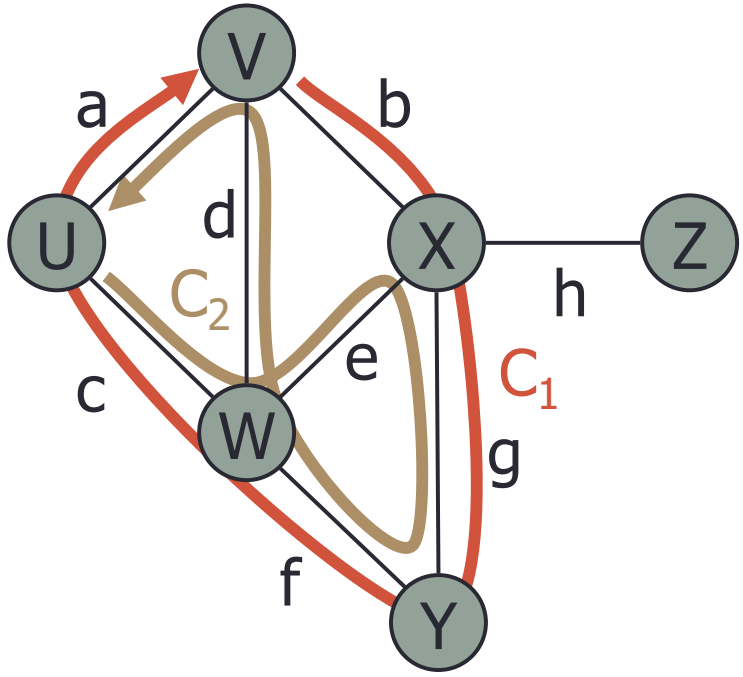
Paths

- Path
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is preceded and followed by its endpoints
- Path length
  - the total number of edges on the path
- Simple path
  - path such that all vertices are distinct (except that the first and last could be the same)
- 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 are distinct (except the first and the last)
- Examples
  - C_1=(V,b,X,g,Y,f,W,c,U,a,V) is a simple cycle
  - C_2=(U,c,W,e,X,g,Y,f,W,d,V,a,U) is a cycle that is not simple
- A directed graph is *acyclic* if it has no cycles $\Rightarrow$ called DAG (directed acyclic graph)



Properties of Undirected Graphs

Property 1

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

Proof: each edge is counted twice

Property 2

In an undirected graph with no loops

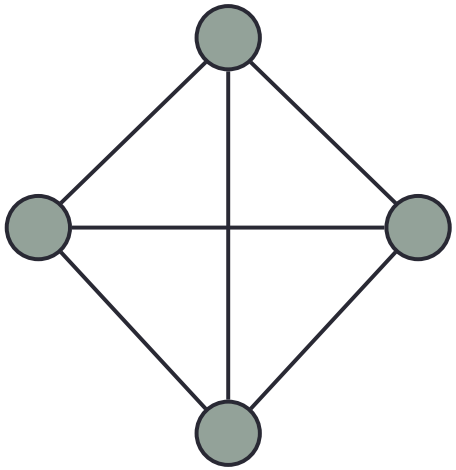
$E \leq V\,(V-1)/2$

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

What is the bound for a directed graph?

Notation

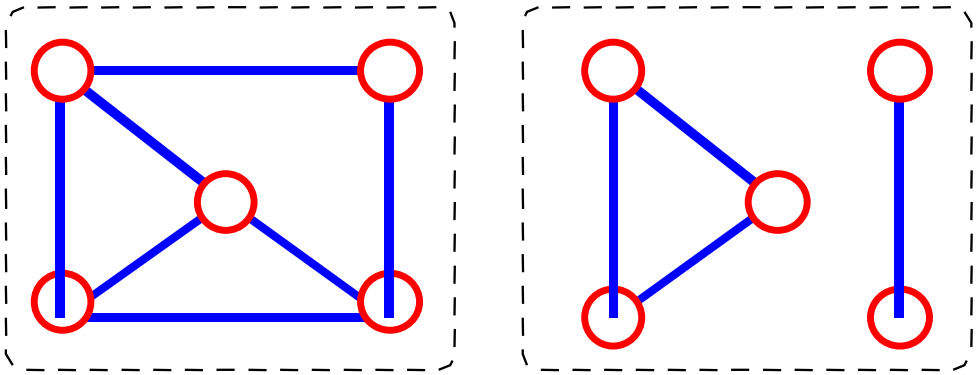
V	number of vertices
E	number of edges
$\deg(v)$	degree of vertex v



Example

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

Connectivity – Undirected Graphs

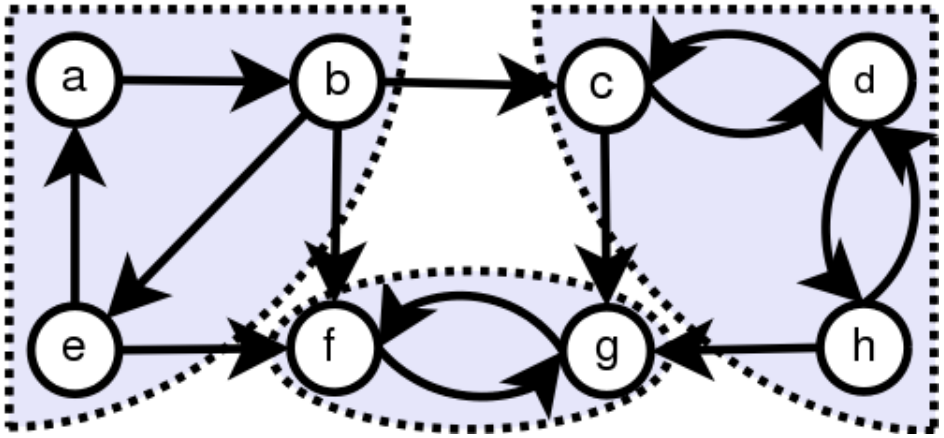


connected not connected

- An undirected graph is *connected* if there is a path from every vertex to every other vertex.

Connectivity – Directed Graphs

- A directed graph is called *strongly connected* if there is a path from every vertex to every other vertex.
- If a directed graph is not strongly connected, but the corresponding undirected graph is connected, then the directed graph is said to be *weakly connected*.



GRAPH ADT AND DATA STRUCTURES

EECS 2011

Vertices and Edges

- A **graph** is a collection of **vertices** and **edges**.
- We model the abstraction as a combination of three data types: Vertex, Edge, and Graph.
- A **vertex** is a lightweight object that stores an arbitrary element provided by the user (e.g., an airport code)
  - We assume it supports a method, element(), to retrieve the stored element.
- An **edge** stores an associated object (e.g., a flight number, travel distance, cost), retrieved with the element( ) method.

Graph ADT

numVertices( ):	Returns the number of vertices of the graph.
vertices( ):	Returns an iteration of all the vertices of the graph.
numEdges( ):	Returns the number of edges of the graph.
edges( ):	Returns an iteration of all the edges of the graph.
getEdge(u, v):	Returns the edge from vertex u to vertex v, if one exists; otherwise return null. For an undirected graph, there is no difference between getEdge(u, v) and getEdge(v, u).
endVertices(e):	Returns an array containing the two endpoint vertices of edge e. If the graph is directed, the first vertex is the origin and the second is the destination.
opposite(v, e):	For edge e incident to vertex v, returns the other vertex of the edge; an error occurs if e is not incident to v.
outDegree(v):	Returns the number of outgoing edges from vertex v.
inDegree(v):	Returns the number of incoming edges to vertex v. For an undirected graph, this returns the same value as does outDegree(v).
outgoingEdges(v):	Returns an iteration of all outgoing edges from vertex v.
incomingEdges(v):	Returns an iteration of all incoming edges to vertex v. For an undirected graph, this returns the same collection as does outgoingEdges(v).
insertVertex(x):	Creates and returns a new Vertex storing element x.
insertEdge(u, v, x):	Creates and returns a new Edge from vertex u to vertex v, storing element x; an error occurs if there already exists an edge from u to v.
removeVertex(v):	Removes vertex v and all its incident edges from the graph.
removeEdge(e):	Removes edge e from the graph.

Representation of Graphs

- Two popular computer representations of a graph: Both represent the vertex set and the edge set, but in different ways.

1. Adjacency Matrices

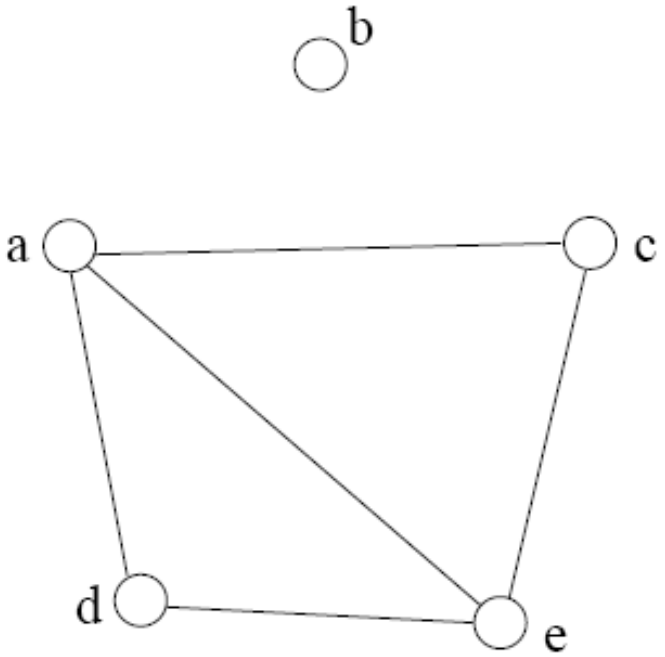
   Use a 2D matrix to represent the graph

2. Adjacency Lists

   Use a set of linked lists, one list per vertex

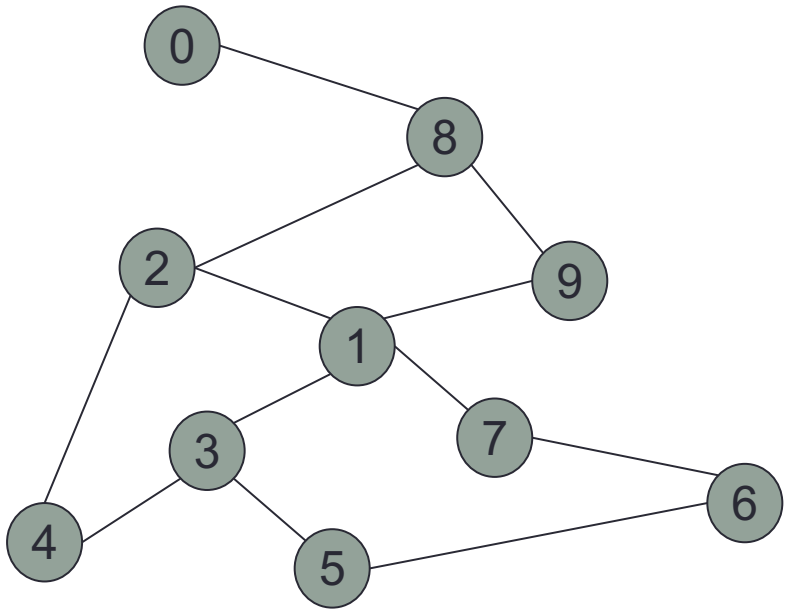
Adjacency Matrix Representation

- 2D array of size ***n*** x ***n*** where ***n*** is the number of vertices in the graph
- A[i][j]=1 if there is an edge connecting vertices i and j; otherwise, A[i][j]=0



	a	b	c	d	e
a	0	0	1	1	1
b	0	0	0	0	0
c	1	0	0	0	1
d	1	0	0	0	1
e	1	0	1	1	0

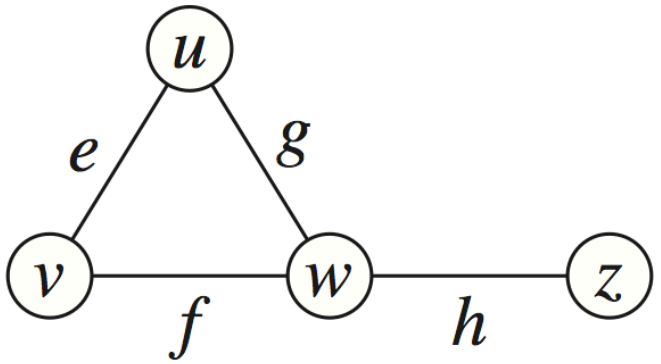
Adjacency Matrix Example



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

Adjacency Matrix 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
- The "old fashioned" version just has 0 for no edge and 1 for edge



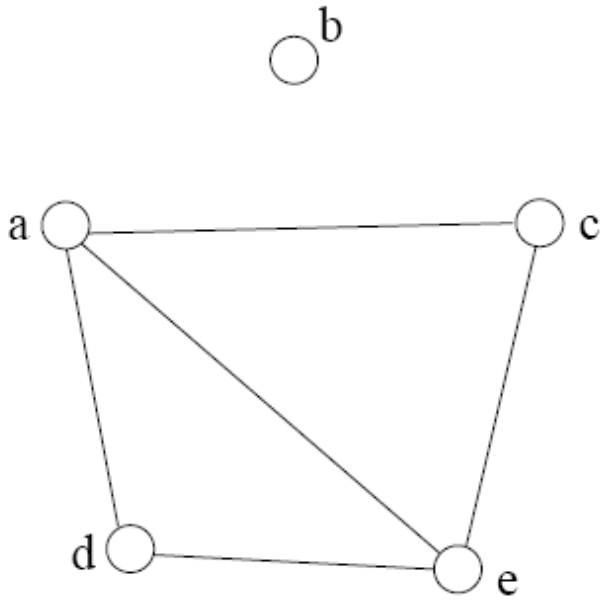
		0	1	2	3
u	→ 0		e	g	
v	→ 1	e		f	
w	→ 2	g	f		h
z	→ 3			h	

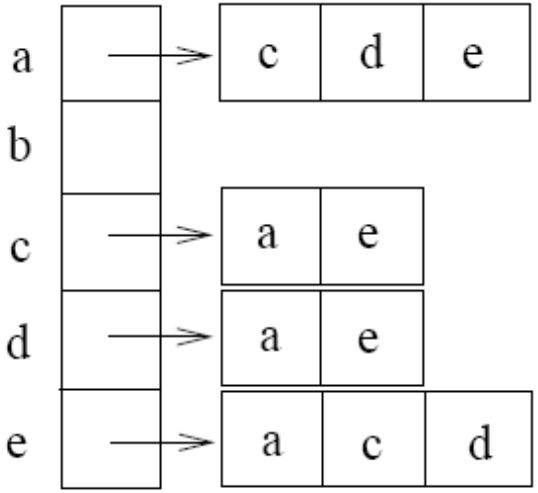
Adjacency Matrices: Analysis

- The storage requirement is $\Theta(V^2)$.
  - not efficient if the graph has few edges.
  - appropriate if the graph is dense; that is E= $\Theta(V^2)$
- If the graph is undirected, the matrix is symmetric. There exist methods to store a symmetric matrix using only half of the space.
  - Note: the space requirement is still $\Theta(V^2)$.
- We can detect in O(1) time whether two vertices are connected.
  - areAdjacent (v, w): returns true if v and w are adjacent, and false otherwise.

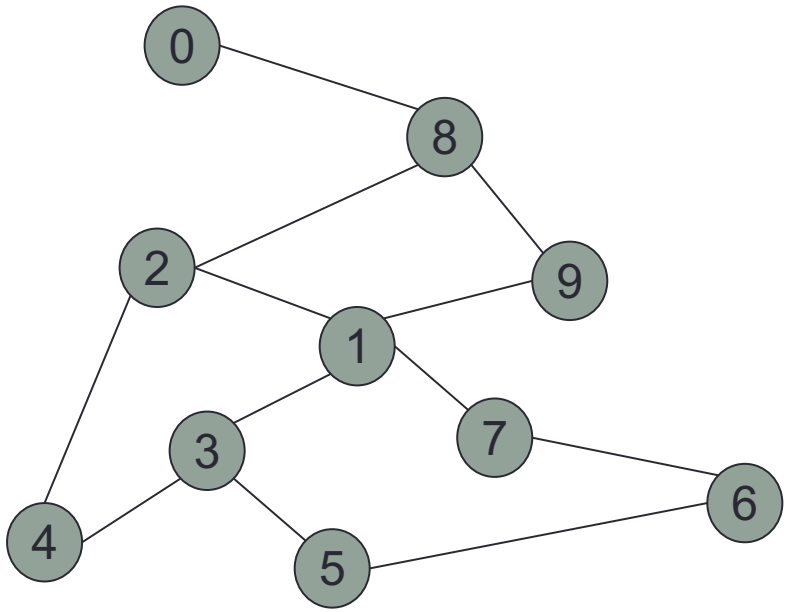
Adjacency Lists

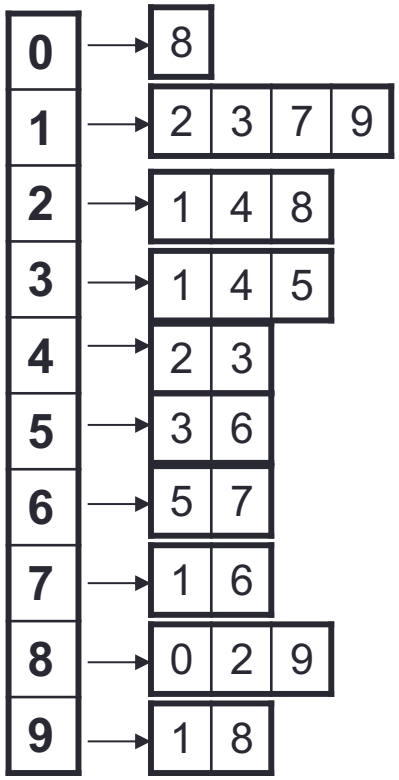
- If the graph is sparse, a better solution is an adjacency list representation.
- For each vertex ***v*** in the graph, we keep a list of vertices adjacent to ***v***.





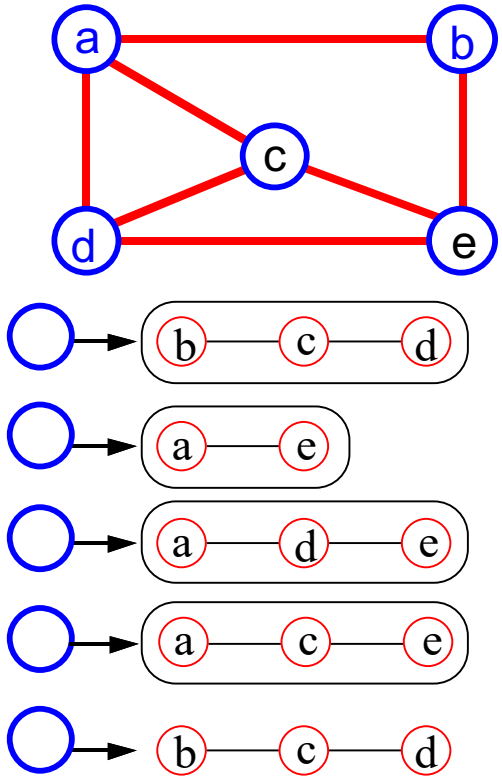
Adjacency List Example





Adjacency Lists: Analysis

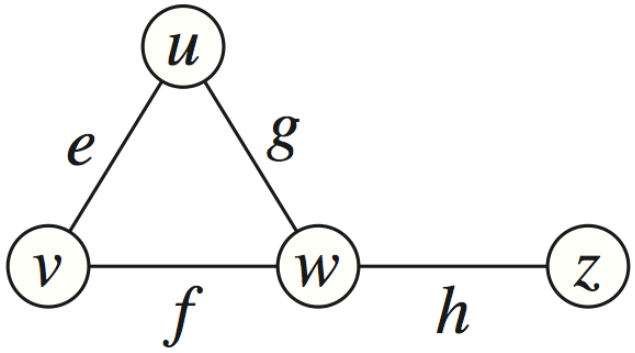
Space =
Θ (V + Σ_v deg(v)) = Θ (V + E)

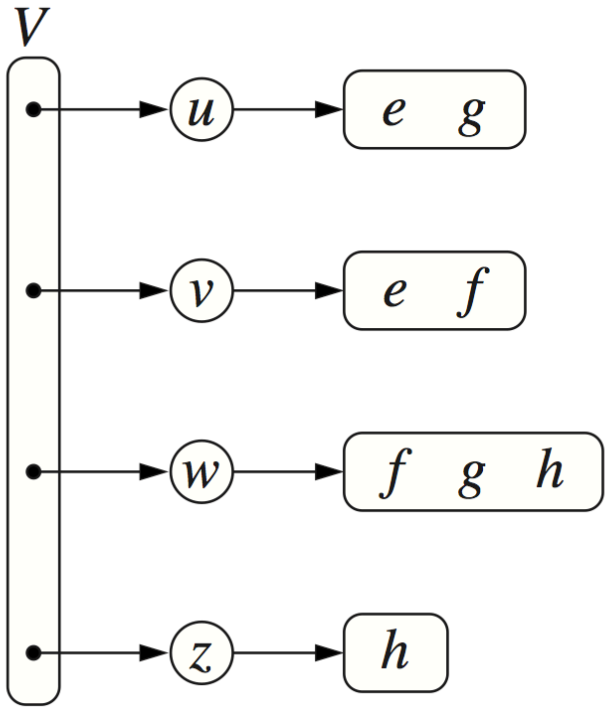


- Testing whether u is adjacency to v takes time O(deg(v)) or O(deg(u)).

Adjacency List Structure

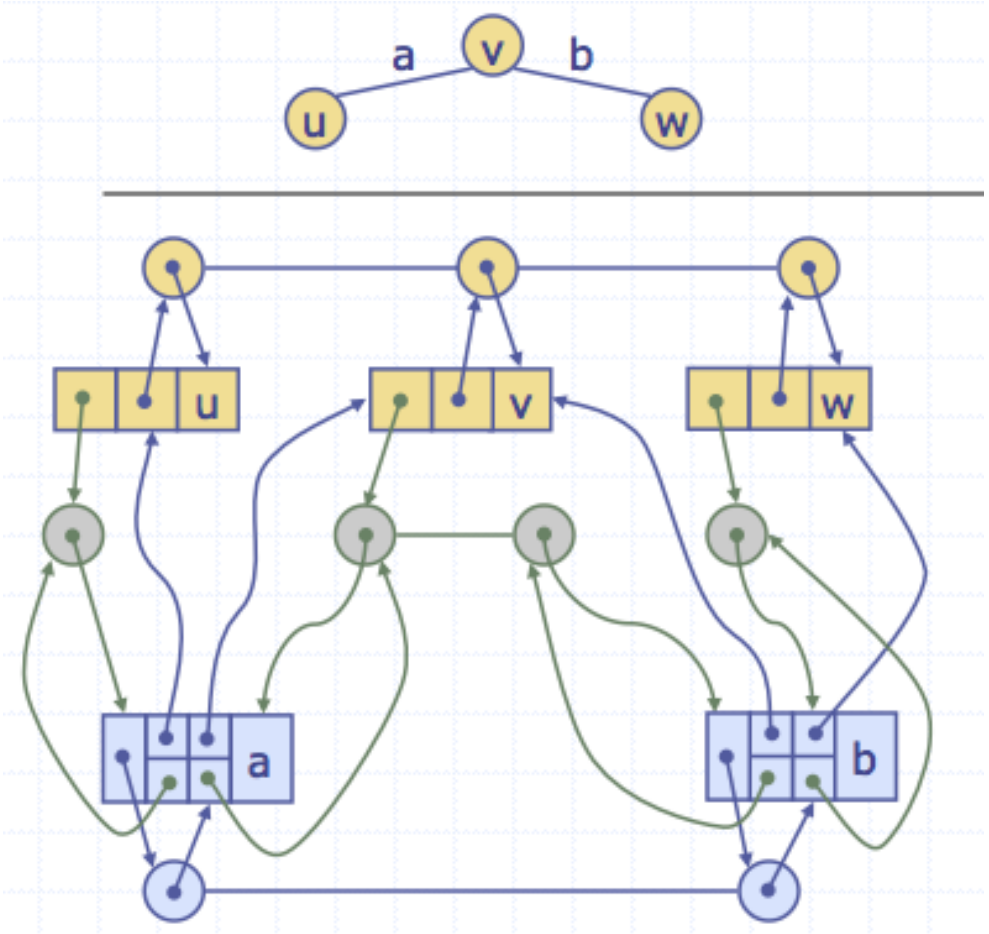
- Incidence sequence for each vertex
  - sequence of references to edge objects of incident edges
- Augmented edge objects
  - references to associated positions in incidence sequences of end vertices





Adjacency List Structure Implementation

- 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 Lists vs. Adjacency Matrices

- An adjacency list takes $\Theta(V + E)$.
  - If $E = O(V^2)$ (dense graph), both use $\Theta(V^2)$ space.
  - If $E = O(V)$ (sparse graph), adjacency lists are more space efficient.

- Adjacency lists
  - More compact than adjacency matrices if graph has few edges
  - Requires more time to find if an edge exists

- Adjacency matrices
  - Always require $\Theta(V^2)$ space
    - This can waste lots of space if the number of edges is small
  - Can quickly find if an edge exists

Homework

- Prove the big-Oh running time of the graph methods shown in the next slide.

- incidentEdges(v): returns the edges incident on v.
- areAdjacent (v, w): returns true if v and w are adjacent, and false otherwise.

Running Time of Graph Methods

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

BREADTH FIRST SEARCH

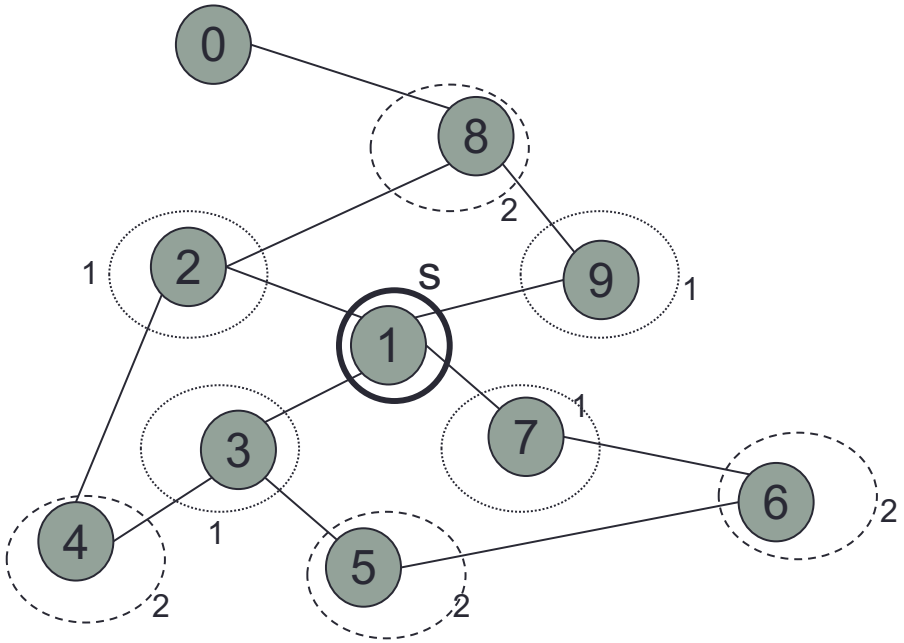
EECS 2011

Graph Traversal (14.3)

- 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.
- 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)
```

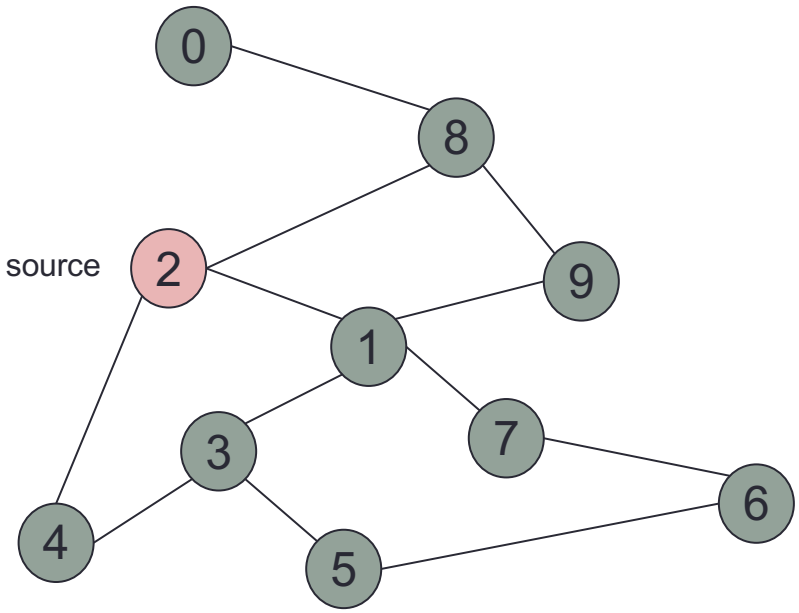
BFS Algorithm

Algorithm $BFS(s)$
Input: s is the source vertex
Output: Mark all vertices that can be visited from s.

```
1.   for each vertex v
2.       do flag[v] := false;
3.   Q = empty queue;
4.   flag[s] := true;
5.   enqueue(Q, s);
6.   while Q is not empty
7.       do v := dequeue(Q);   output v;
8.           for each w adjacent to v
9.               do if flag[w] = false
10.                  then flag[w] := true;
11.                       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

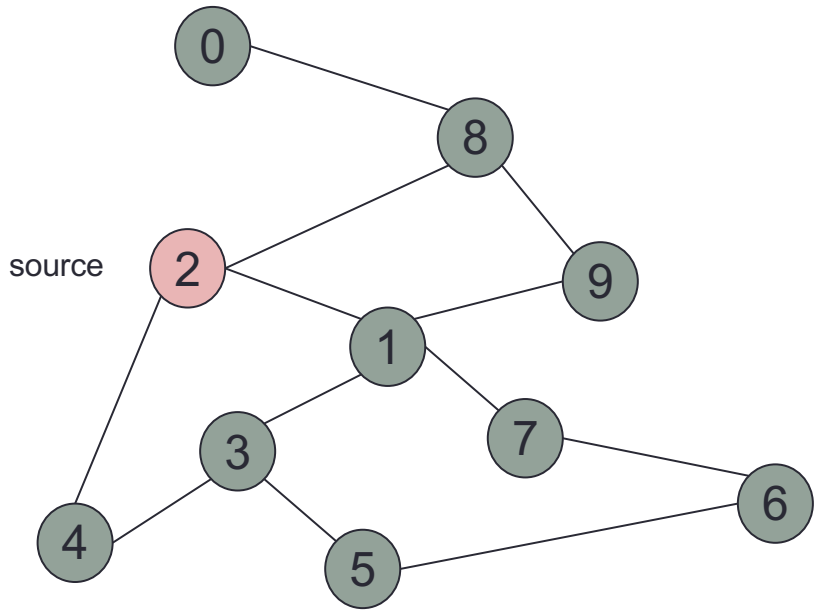
Visited Table (T/F)

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

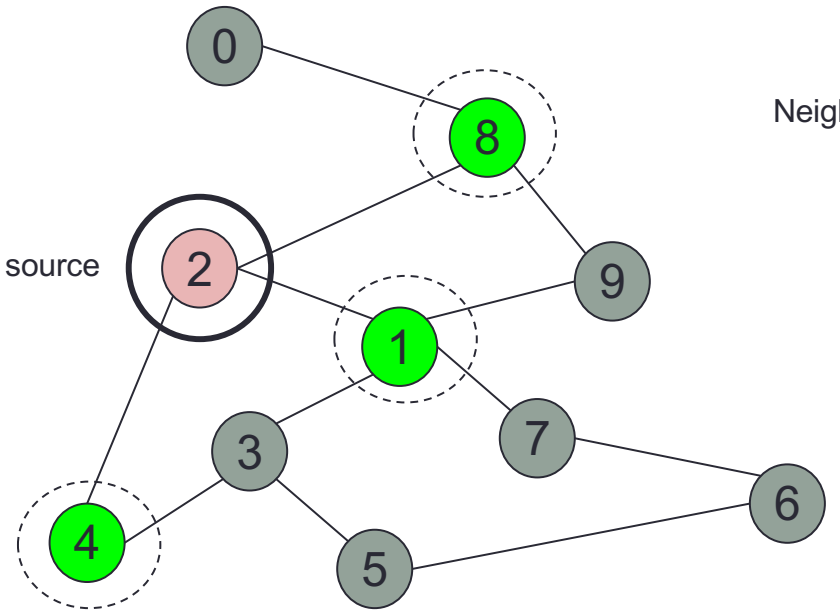
Visited Table (T/F)

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

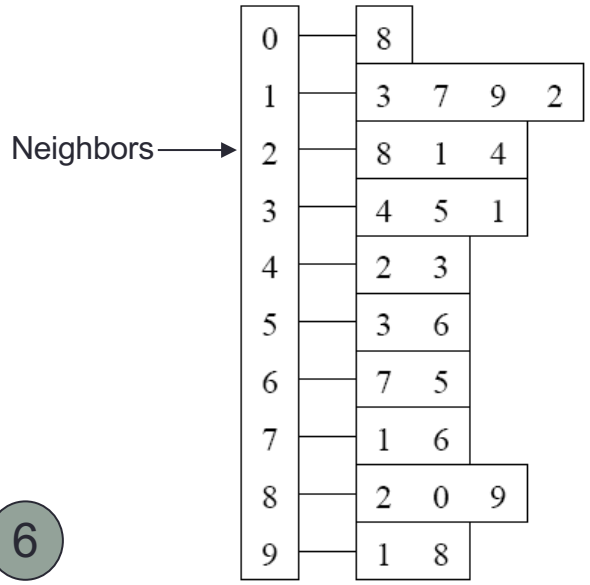
Flag that 2 has been visited

Q = { 2 }

Place source 2 on the queue



Adjacency List



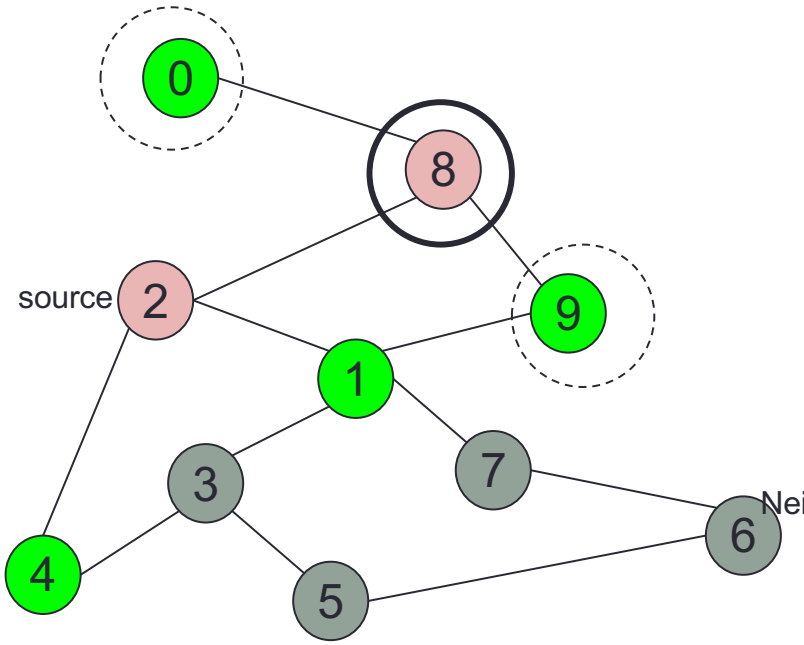
Visited Table (T/F)

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

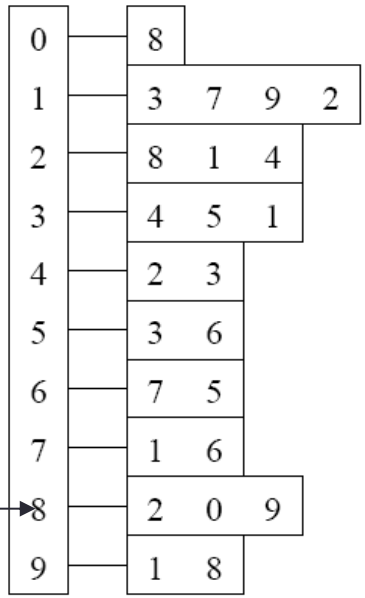
Mark neighbors as visited 1, 4, 8

Q = {2} → { 8, 1, 4 }

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



Adjacency List



Visited Table (T/F)

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

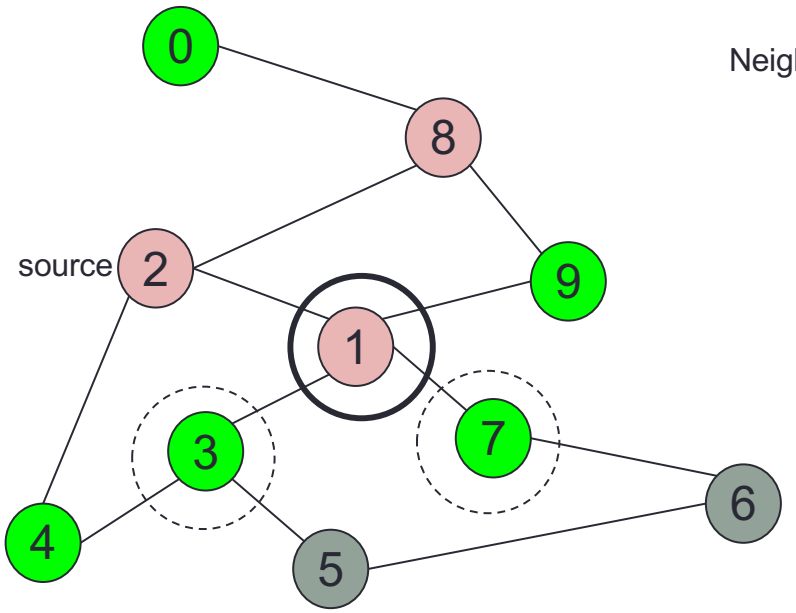
Mark newly visited neighbors 0, 9

Q = { 8, 1, 4 } → { 1, 4, 0, 9 }

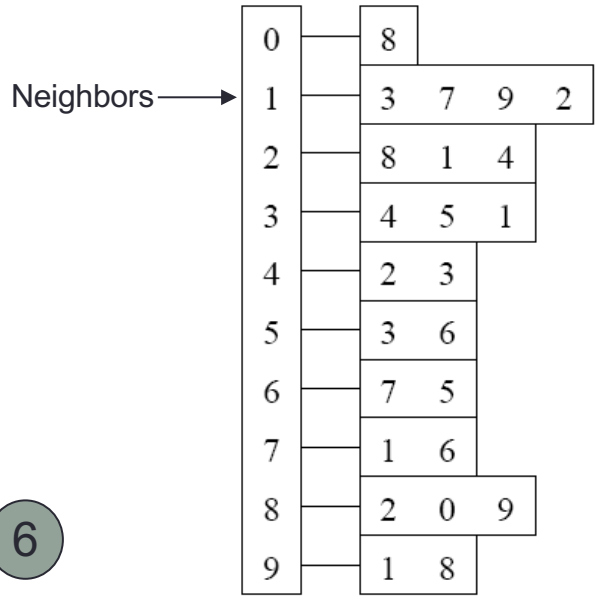
Dequeue 8.

-- 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



Visited Table (T/F)

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

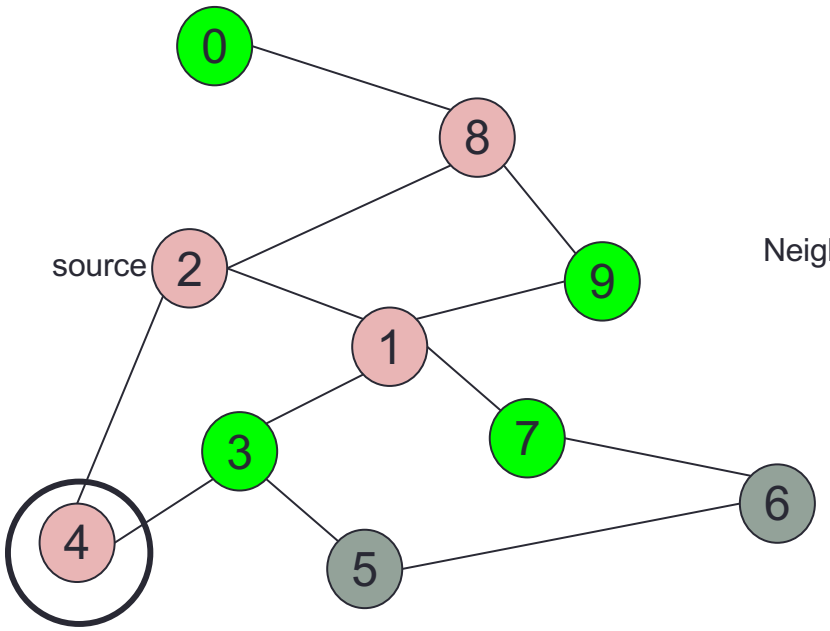
Mark newly visited neighbors 3, 7

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 →

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

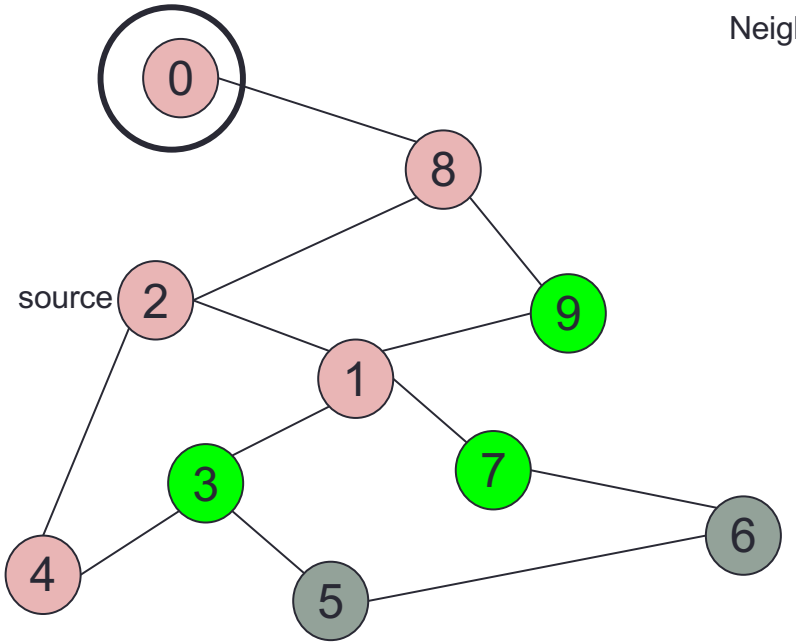
Visited Table (T/F)

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

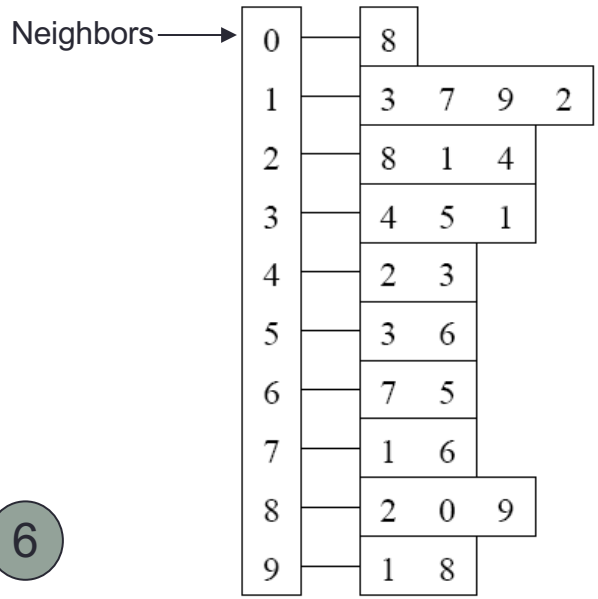
Q = { 4, 0, 9, 3, 7 } → { 0, 9, 3, 7 }

Dequeue 4.

-- 4 has no unvisited neighbors!



Adjacency List



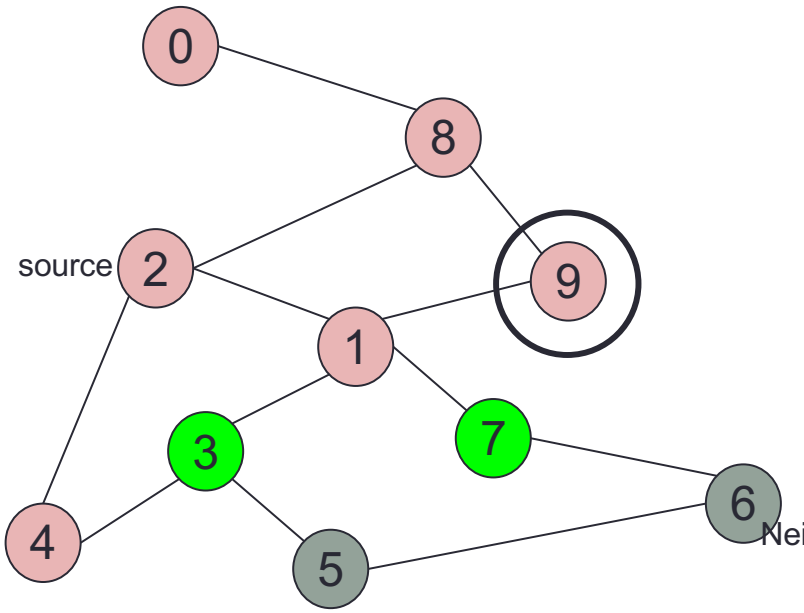
Visited Table (T/F)

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

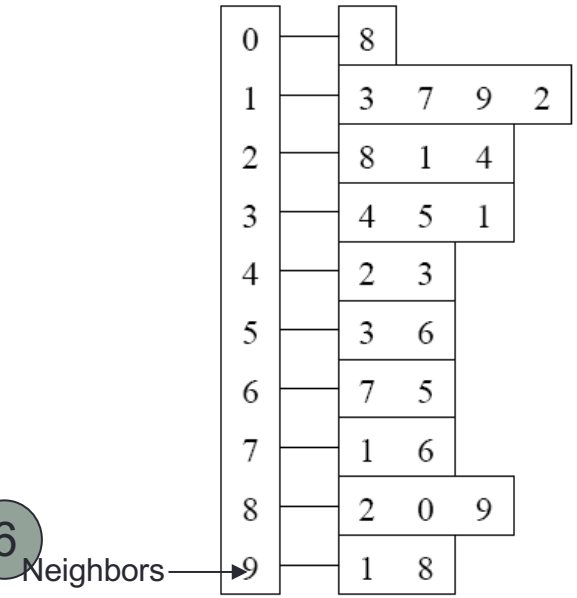
Q = { 0, 9, 3, 7 } → { 9, 3, 7 }

Dequeue 0.

-- 0 has no unvisited neighbors!



Adjacency List



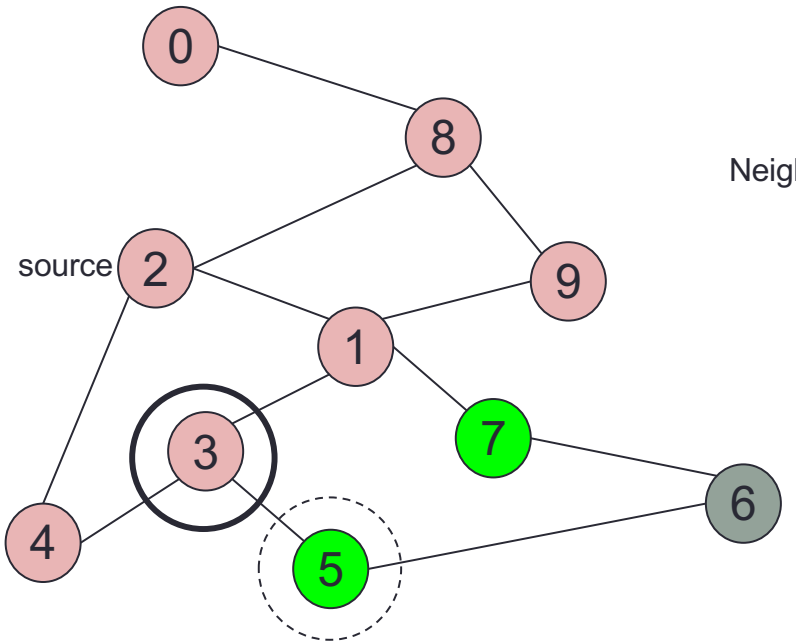
Visited Table (T/F)

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

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

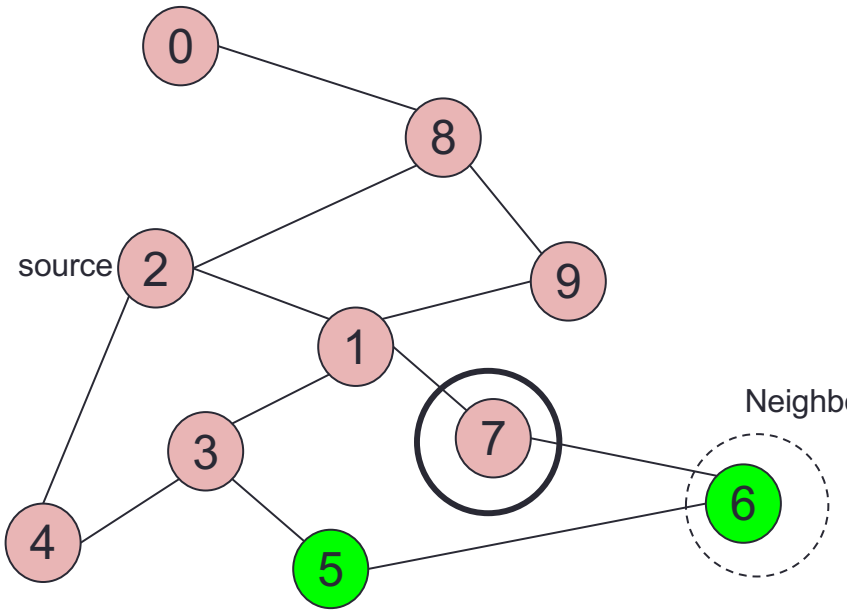
Visited Table (T/F)

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

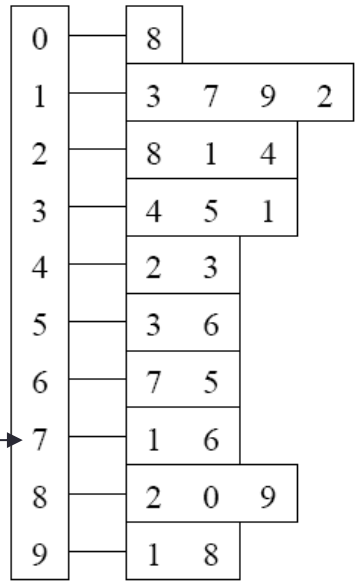
Mark new visited Vertex 5

Q = { 3, 7 } → { 7, 5 }

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



Adjacency List



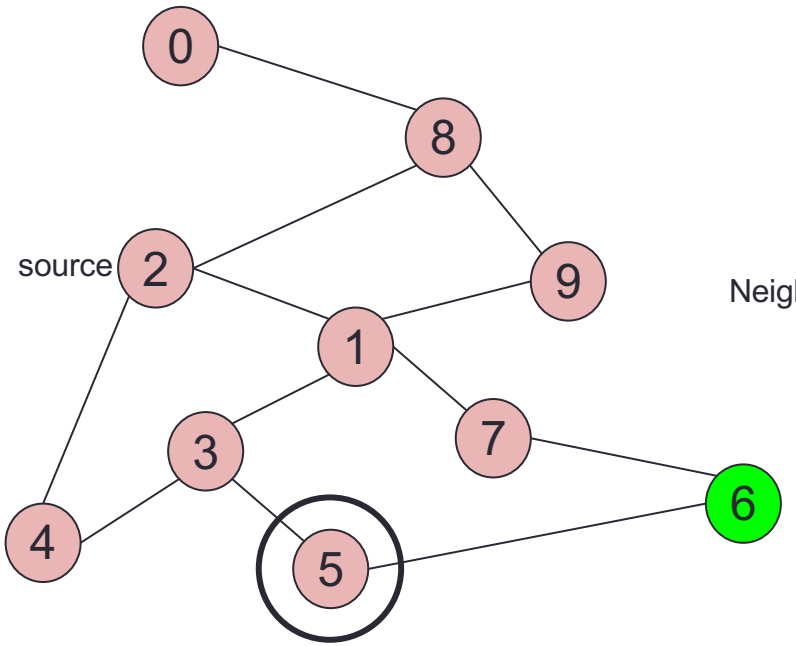
Visited Table (T/F)

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 →

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

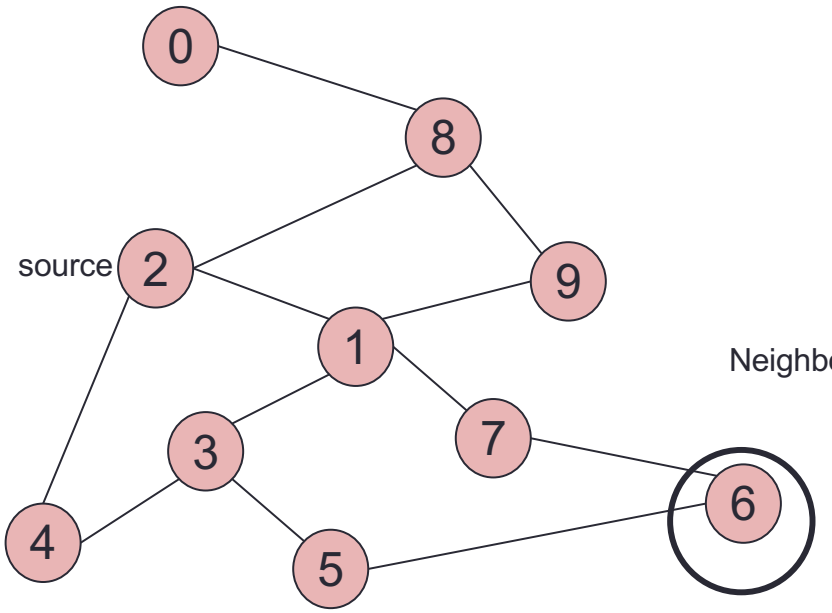
Visited Table (T/F)

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 →

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

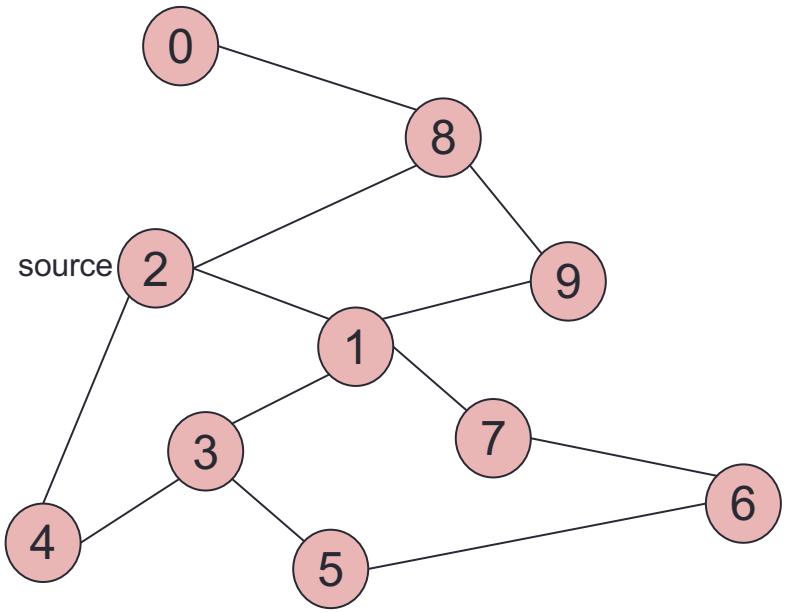
Visited Table (T/F)

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.

```
1.   for each vertex v
2.       do flag[v] := false;
3.   Q = empty queue;
4.   flag[s] := true;
5.   enqueue(Q, s);
6.   while Q is not empty
7.      do v := dequeue(Q);
8.         for each w adjacent to v
9.            do if flag[w] = false
10.                 then flag[w] := true;
11.                      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

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

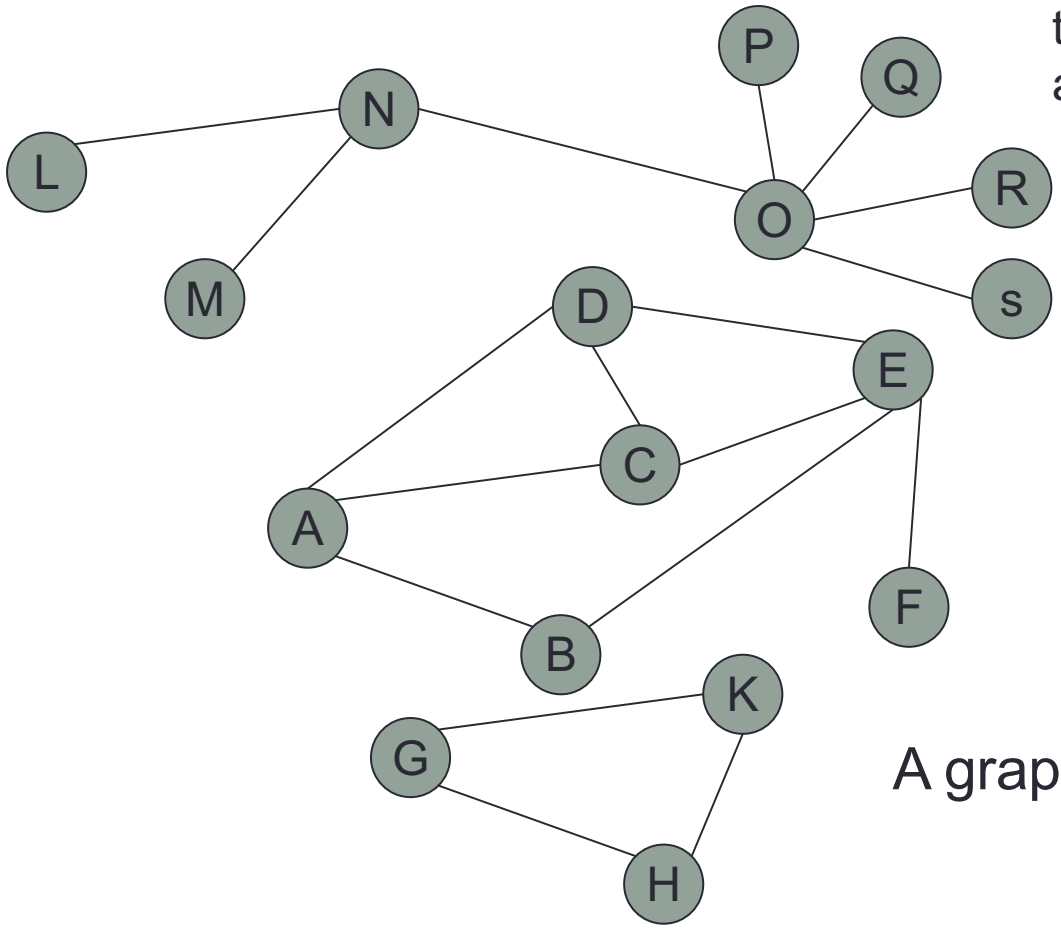
- The total running time of the while loop is:

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

- 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) $\Rightarrow$ enhance the above BFS code to accommodate this case.

A graph with 3 components

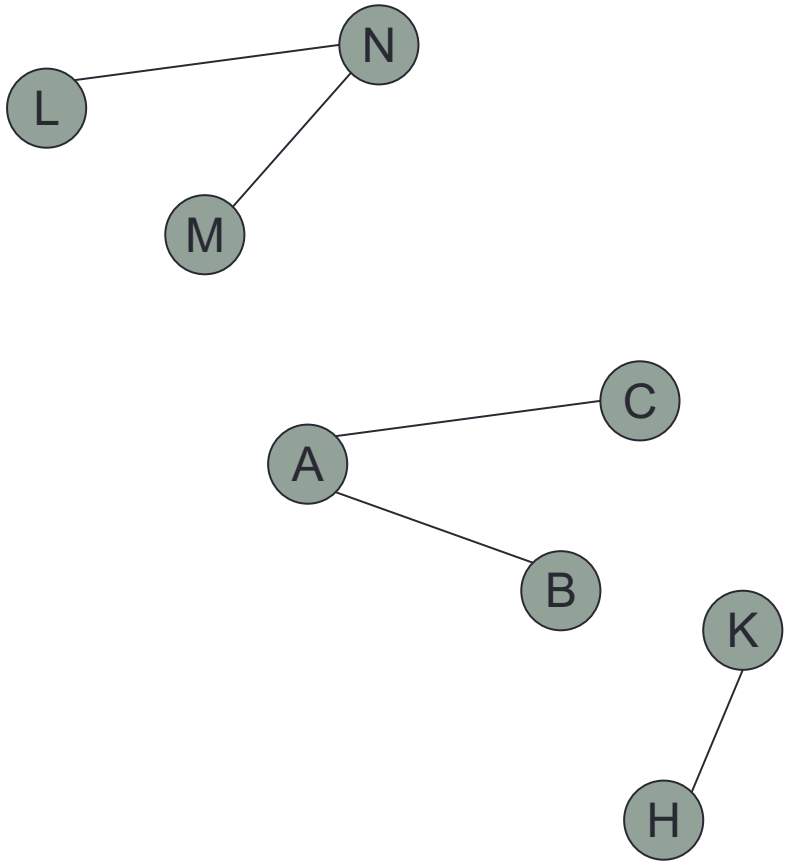
Recall the BFS Algorithm …

Algorithm $BFS(s)$
Input: s is the source vertex
Output: Mark all vertices that can be visited from s.

```
1.    for each vertex v
2.        do flag[v] := false;
3.    Q = empty queue;
4.    flag[s] := true;
5.    enqueue(Q, s);
6.    while Q is not empty
7.        do v := dequeue(Q);      output ( v );
8.            for each w adjacent to v
9.                do if flag[w] = false
10.                   then flag[w] := true;
11.                        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 …
- To output the contents (e.g., the vertices) of a connected graph
  - What if the graph is not connected? Slide 24
- To output the contents of a strongly connected graph
  - What if the graph is not connected or 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
- To construct a BSF tree/forest from a graph

APPLICATIONS OF BFS

EECS 2011

Applications of BFS

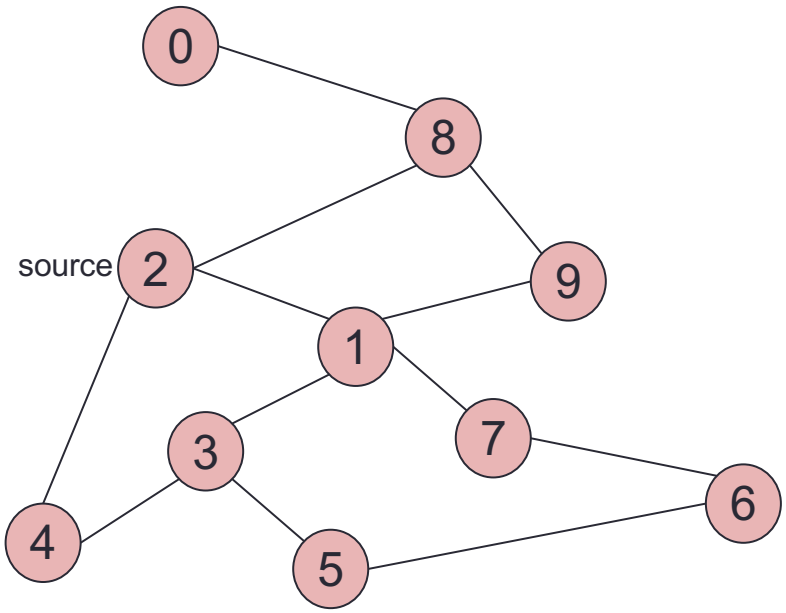
- To find the shortest path from a vertex *s* to a vertex *v* in an unweighted graph
- To find the length of such a path
- To find out if a strongly connected directed graph contains cycles
- To find out if an undirected graph contains cycles
- To construct a BSF tree/forest from a graph

FINDING SHORTEST PATHS USING BFS

Finding Shortest Paths

- The BFS code we have seen
  - find outs if there exists a path from a vertex s to a vertex v
  - prints the vertices of a graph (connected/strongly connected).
- What if we want to find
  - the shortest path from *s* to a vertex *v* (or to every other vertex)?
  - the length of the shortest path from *s* to a vertex *v*?
- In addition to array *flag*[ ], use an array named *prev*[ ], one element per vertex.
  - *prev*[*w*] = *v* means that vertex *w* was visited right after *v*

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

8
2
-
1
2
3
7
1
2
8

***prev*[ ]**

***prev*[ ] now can be traced backward to report the path!**

BFS and Finding Shortest Path

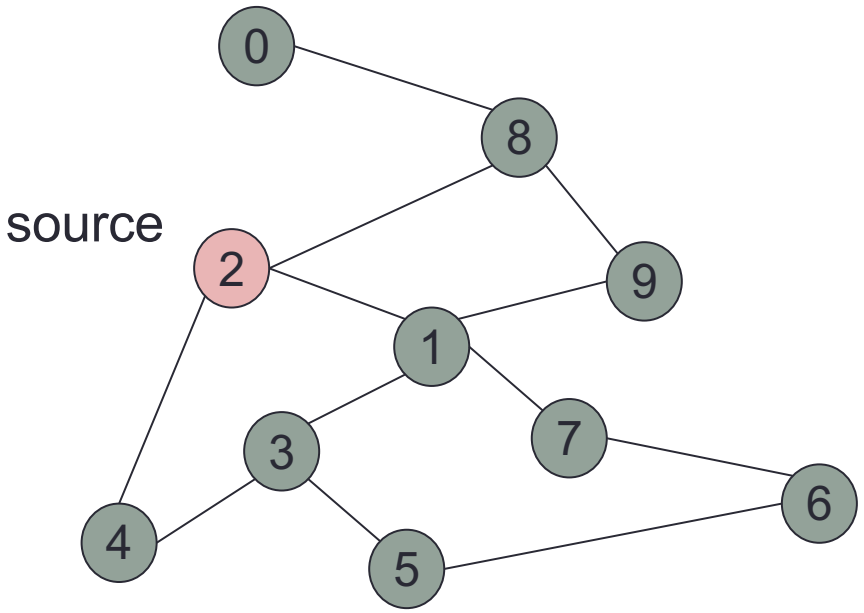
```
Algorithm BFS(s)
1.    for each vertex v
2.        do flag(v) := false;
3.           pred[v] := −1;
4.    Q = empty queue;
5.    flag[s] := true;
6.    enqueue(Q, s);
7.    while Q is not empty
8.       do v := dequeue(Q);
9.          for each w adjacent to v
10.            do if flag[w] = false
11.                  then flag[w] := true;
12.                       pred[w] := v;
13.                       enqueue(Q, w)
```

initialize
all pred[v] to -1

already got shortest path from s to v

record where you
came from

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

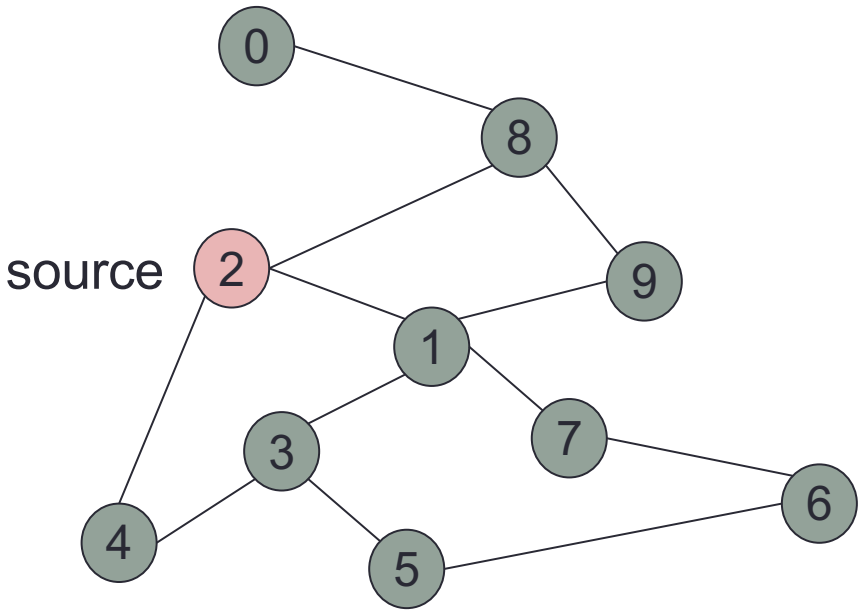
prev[ ]
-
-
-
-
-
-
-
-
-
-

Initialize visited table (all false)

Initialize prev[ ] to -1

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)

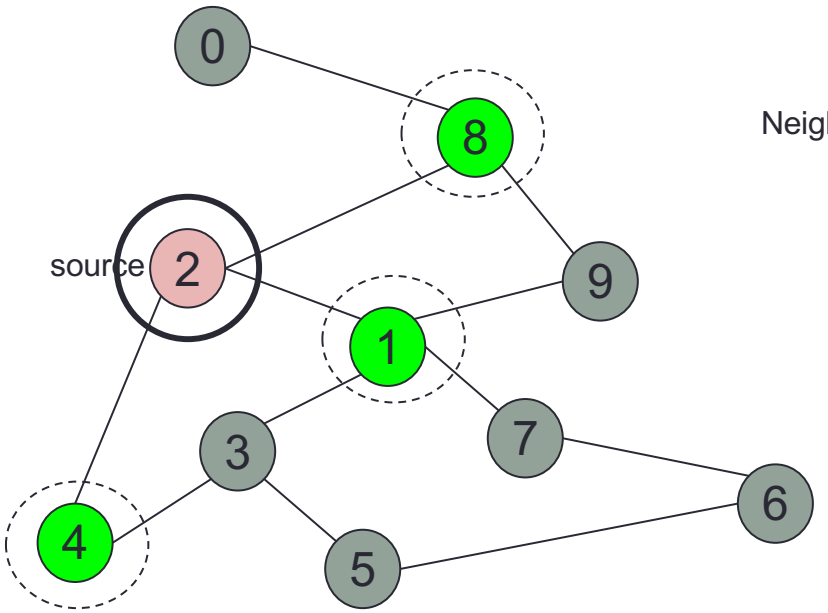
0	F	-
1	F	-
2	T	-
3	F	-
4	F	-
5	F	-
6	F	-
7	F	-
8	F	-
9	F	-

prev

Flag that 2 has been visited.

Q = { 2 }

Place source 2 on the queue.



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

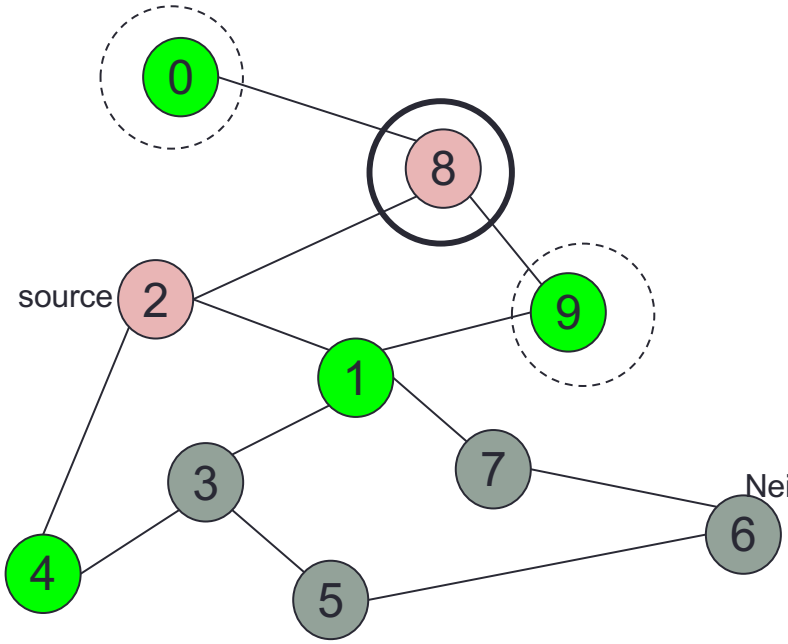
prev

Q = {2} → { 8, 1, 4 }

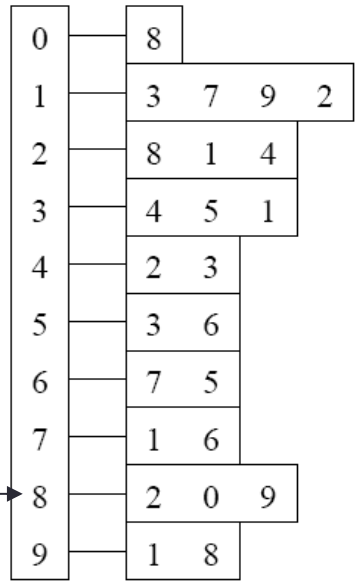
Dequeue 2.
Place all unvisited neighbors of 2 on the queue

Mark neighbors as visited.

Record in prev that we came from 2.



Adjacency List



Visited Table (T/F)

prev

Q = { 8, 1, 4 } → { 1, 4, 0, 9 }

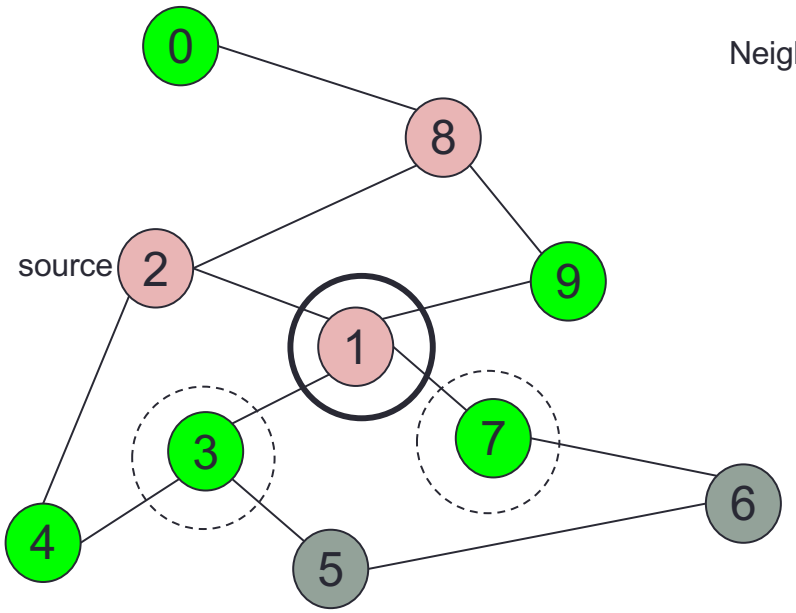
Mark new visited Neighbors.

Record in prev that we came from 8.

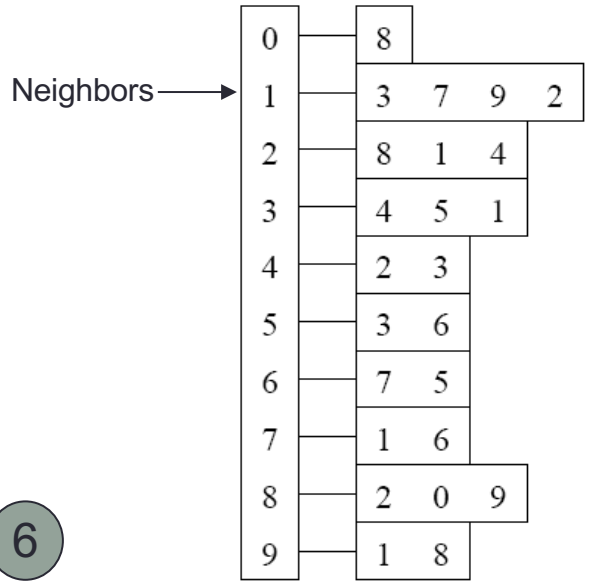
Dequeue 8.

-- Place all unvisited neighbors of 8 on the queue.

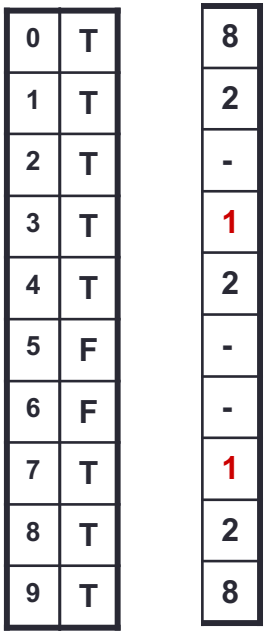
-- Notice that 2 is not placed on the queue again, it has been visited!



Adjacency List



Visited Table (T/F)



Mark new visited Neighbors.

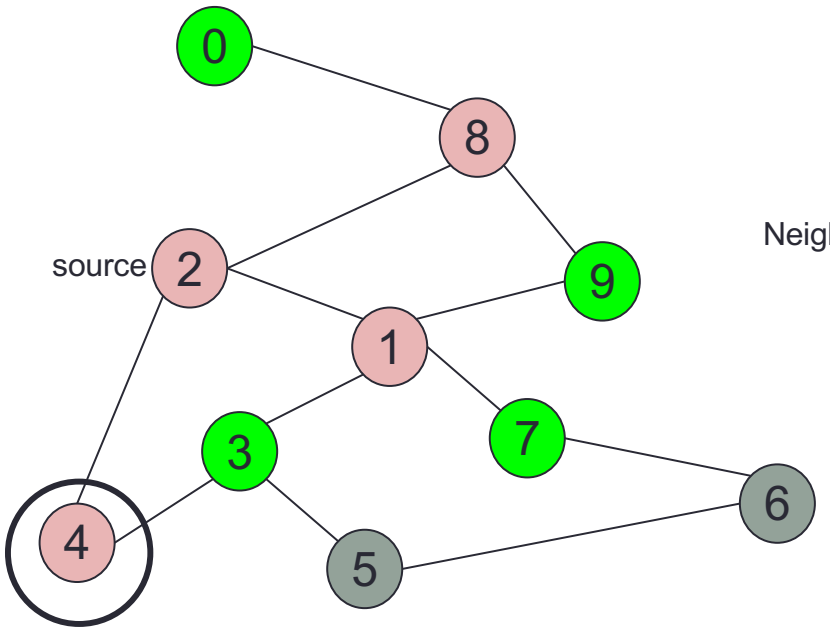
Record in prev that we came from 1.

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 →

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)

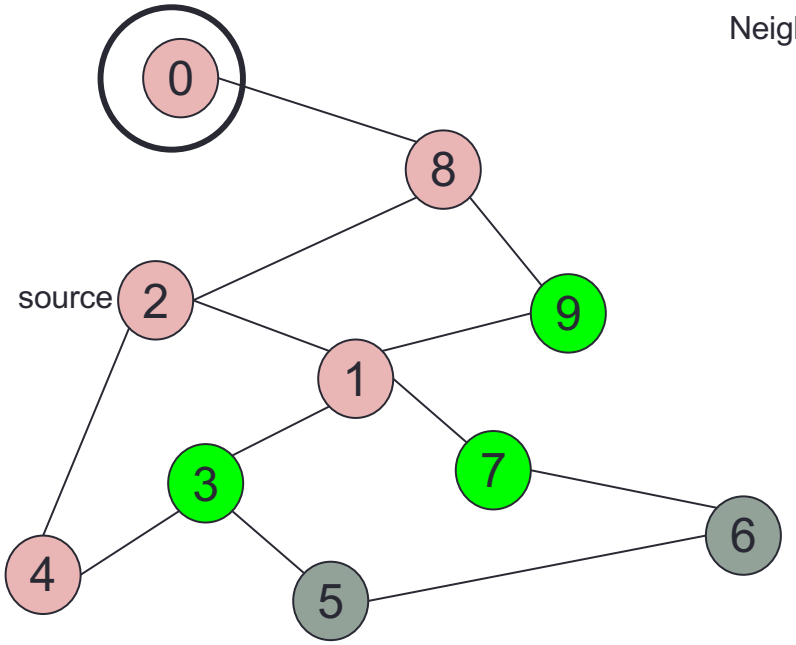
0	T	8
1	T	2
2	T	-
3	T	1
4	T	2
5	F	-
6	F	-
7	T	1
8	T	2
9	T	8

prev

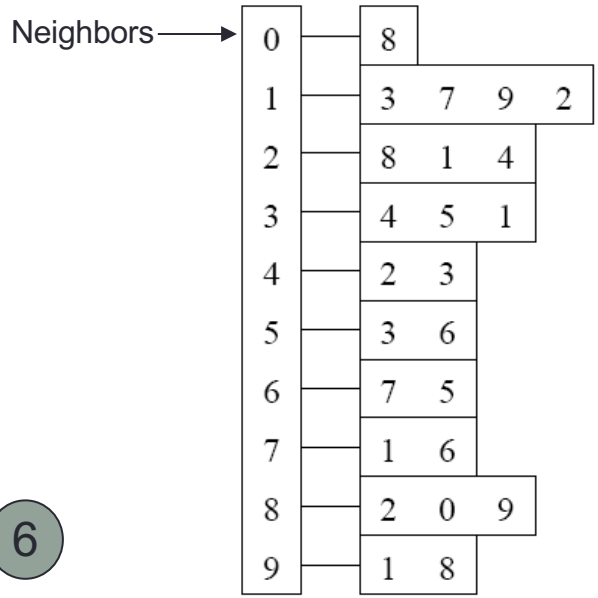
Q = { 4, 0, 9, 3, 7 } → { 0, 9, 3, 7 }

Dequeue 4.

-- 4 has no unvisited neighbors!



Adjacency List



Visited Table (T/F)

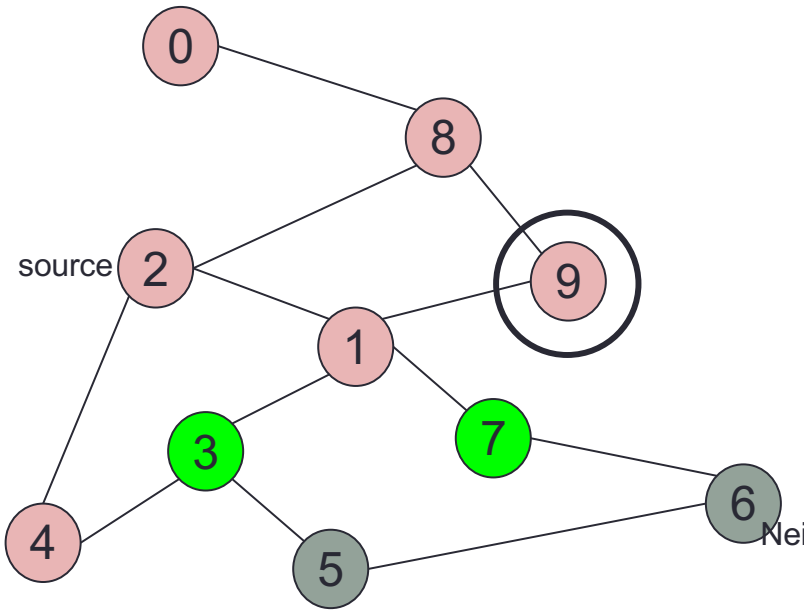
0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

prev
8
2
-
1
2
-
-
1
2
8

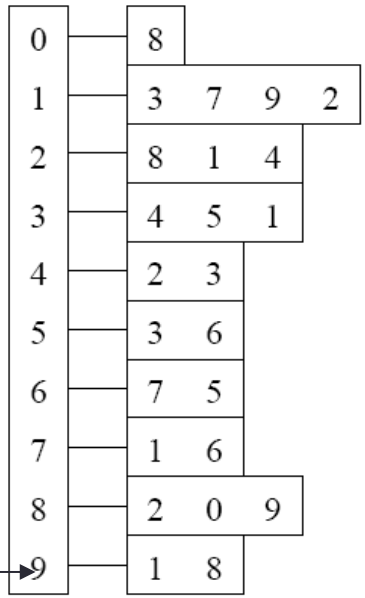
Q = { 0, 9, 3, 7 } → { 9, 3, 7 }

Dequeue 0.

-- 0 has no unvisited neighbors!



Adjacency List



Visited Table (T/F)

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

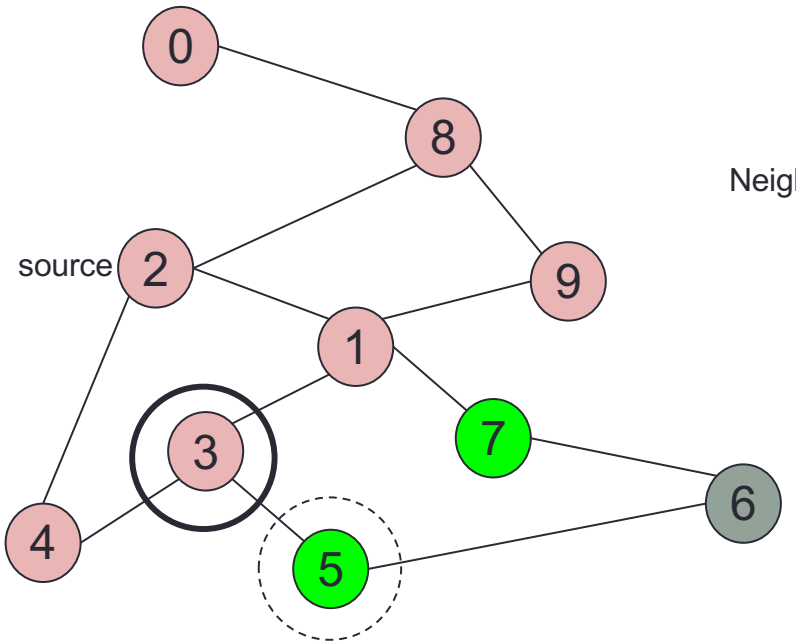
8
2
-
1
2
-
-
1
2
8

prev

Q = { 9, 3, 7 } → { 3, 7 }

Dequeue 9.

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

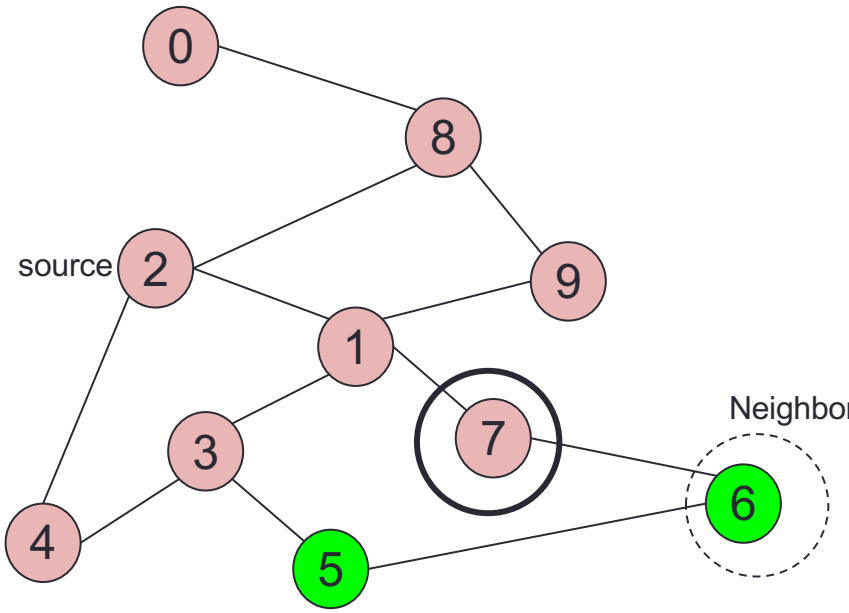
prev

Mark new visited Vertex 5.

Record in prev that we came from 3.

Q = { 3, 7 } → { 7, 5 }

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



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

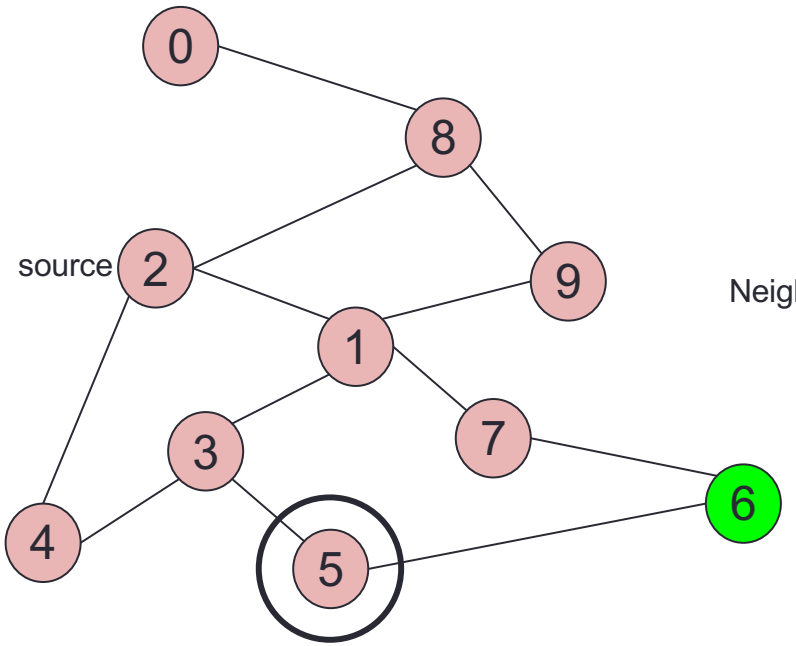
prev

Mark new visited Vertex 6.

Q = { 7, 5 } → { 5, 6 }

Dequeue 7.
-- place neighbor 6 on the queue.

Record in prev that we came from 7.



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)

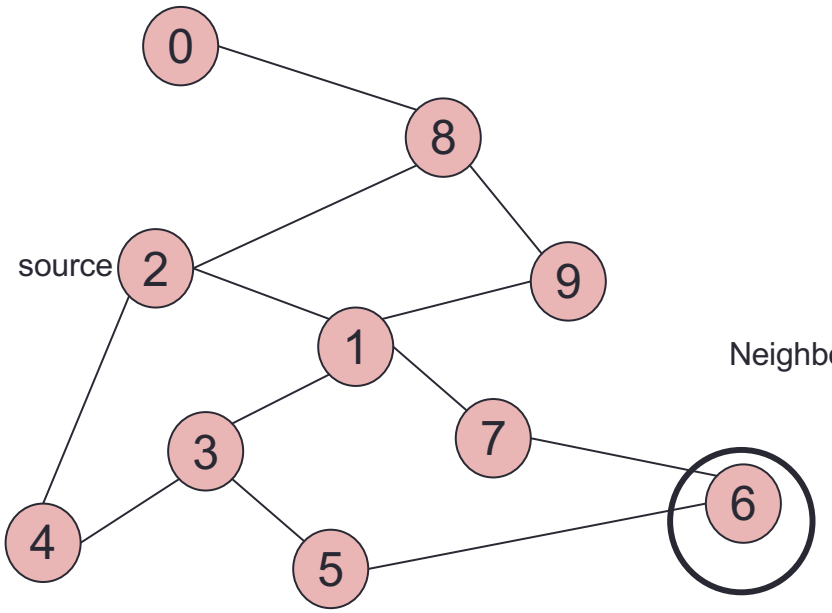
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

prev
8
2
-
1
2
3
7
1
2
8

Q = { 5, 6} → { 6 }

Dequeue 5.

-- no unvisited neighbors of 5.



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

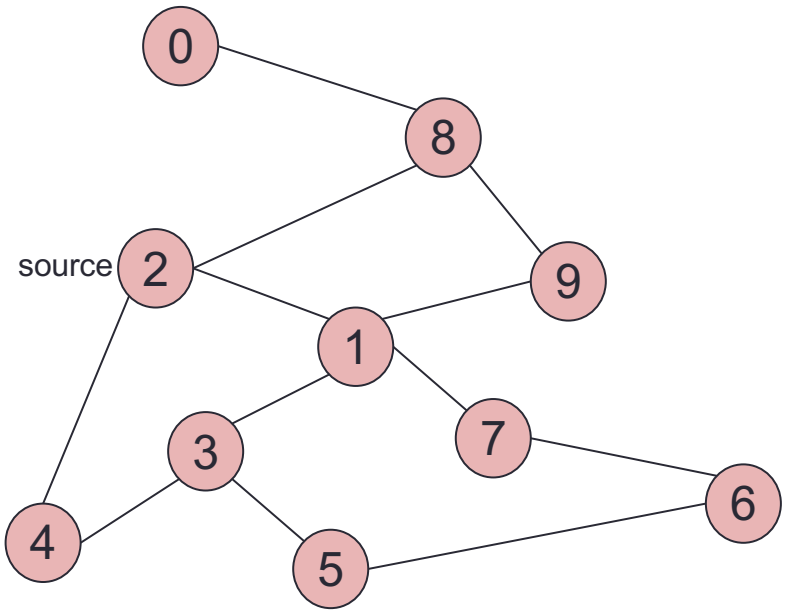
prev

Q = { 6 } → { }

Dequeue 6.

-- no unvisited neighbors of 6.

BFS Finished



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

8
2
-
1
2
3
7
1
2
8

***prev*[ ]**

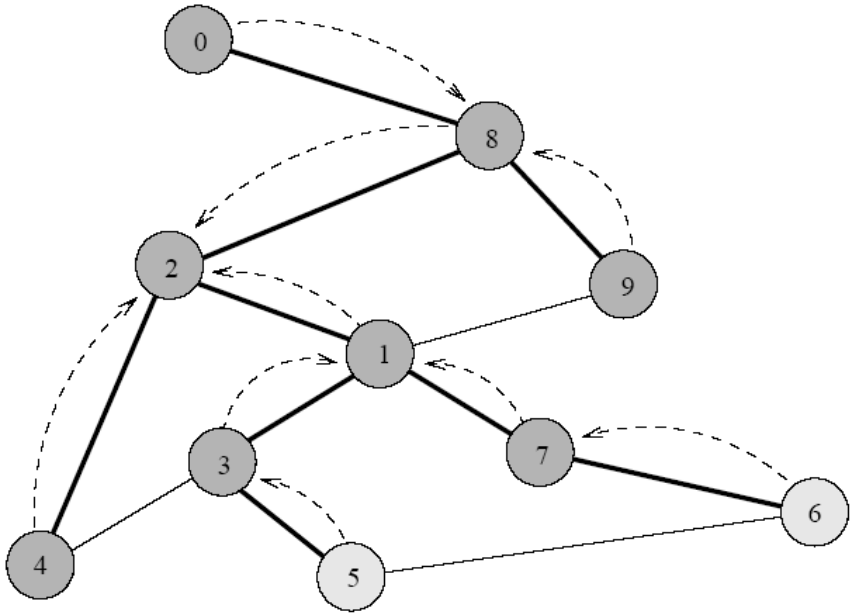
Q = { } STOP!!! Q is empty!!!

***prev*[ ] now can be traced backward to report the path!**

Finding the Shortest Path

- To print the shortest path from *s* to a vertex *u*, start with *prev*[*u*] and backtrack until reaching the source *s* (path reporting).
  - Running time of backtracking = ?

Example of Path Reporting



nodes	visited from
0	8
1	2
2	-
3	1
4	2
5	3
6	7
7	1
8	2
9	8

Try some examples; report path from s to v:

Path(2-0) ⇒

Path(2-6) ⇒

Path(2-1) ⇒

Path Reporting

- Given a vertex *w*, report the shortest path from *s* to *w*

```
currentV = w;
while (prev[currentV] ≠ –1) {
  output currentV;            // or add to a list
  currentV = prev[currentV];
}
output s;     // or add to a list
```

- The above code prints the path in *reverse* order.

Path Reporting (2)

- To output the path in the right order,
  - Print the above list in reverse order.
  - Use a stack instead of a list.

```
currentV = w;
while ( prev[currentV] ≠ –1 )
   S.push( currentV );
   currentV = prev[currentV];
}
while ( !S.isEmpty() )
    print( S.pop() );
print( s );
```

Path Reporting (3)

- To output the path in the right order,
  - Print the list in reverse order.
  - Use a stack instead of a list.
  - Use a recursive method (implicit use of a stack).

```
printPath (w) {
  if (prev[w] ≠ –1)
        printPath (prev[w]);
  output w;
}
```

Finding Shortest Path Length

- To find the length of the shortest path from *s* to *u*, start with *prev*[*u*], backtrack and increment a counter until reaching the source *s*.
  - Running time of backtracking = ?

- Following is a faster way to find the length of the shortest path from *s* to *u* (at the cost of using more space)
  - Allocate an array *d*[ ], one element per vertex.
  - When BSF algorithm ends, *d*[*u*] records the length of the shortest path from *s* to *u.*
  - Running time of finding path length = ?

Recording the Shortest Distance

```
Algorithm BFS(s)
1.    for each vertex v
2.        do flag(v) := false;
3.           pred[v] := −1;  d[v] = ∞;
4.    Q = empty queue;
5.    flag[s] := true;  d[s] = 0;
6.    enqueue(Q, s);
7.    while Q is not empty
8.       do v := dequeue(Q);
9.          for each w adjacent to v
10.            do if flag[w] = false
11.                  then flag[w] := true;
12.                       pred[w] := v;  d[w] = d[v] + 1;
13.                       enqueue(Q, w)
```

d[v] stores shortest distance from s to v

FINDING CYCLES

Finding Cycles in Undirected Graphs

- To detect/find cycles in an ***undirected*** graph, we need to classify the edges into 3 categories during program execution:
  - *unexplored* edge: never visited.
  - *discovery* edge: visited for the very **first** time.
  - *cross* edge: edge that forms a cycle.
- When the BFS algorithm terminates, the discovery edges form a spanning tree.
- If there exists a *cross* edge, the undirected graph contains a cycle.

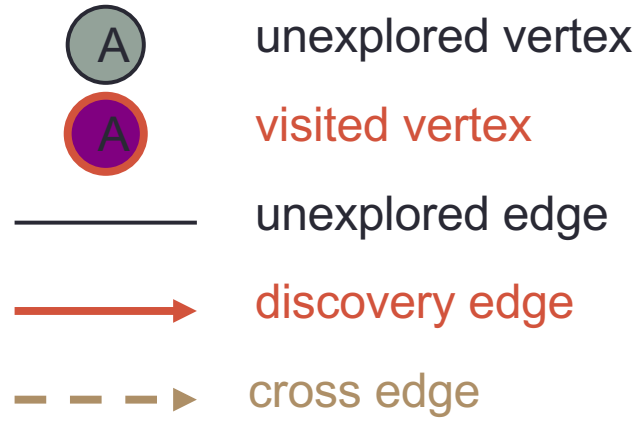
BFS Algorithm (in textbook)

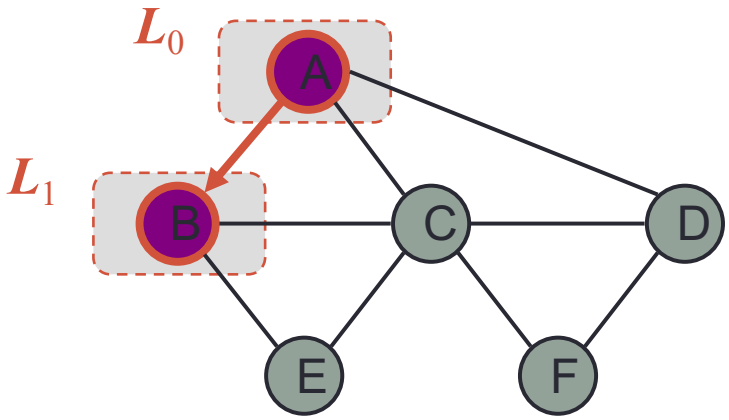
- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

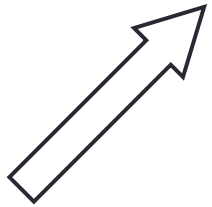
```
Algorithm BFS(G)
    Input graph G
    Output labeling of the edges
        and partition of the
        vertices of G
  for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
      BFS(G, v)
```

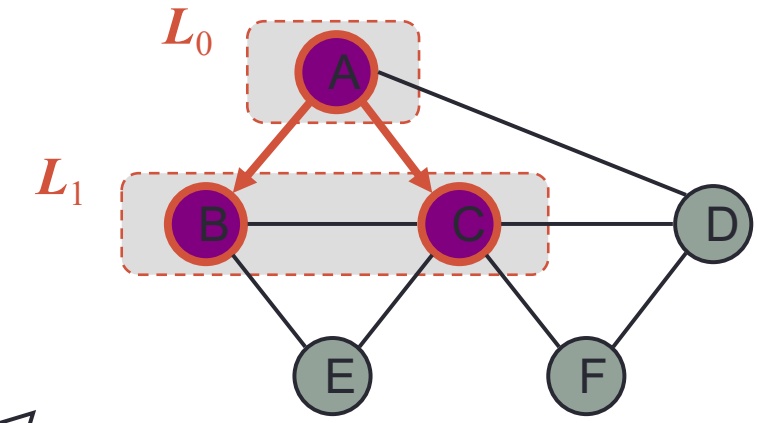
```
Algorithm BFS(G, s)
  L0 ← new empty sequence
  L0.insertLast(s)
  setLabel(s, VISITED)
  i ← 0
  while ¬Li.isEmpty()
    Li+1 ← new empty sequence
    for all v ∈ Li.elements()
      for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
          w ← opposite(v,e)
          if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            setLabel(w, VISITED)
            Li+1.insertLast(w)
          else
            setLabel(e, CROSS)
    i ← i +1
```

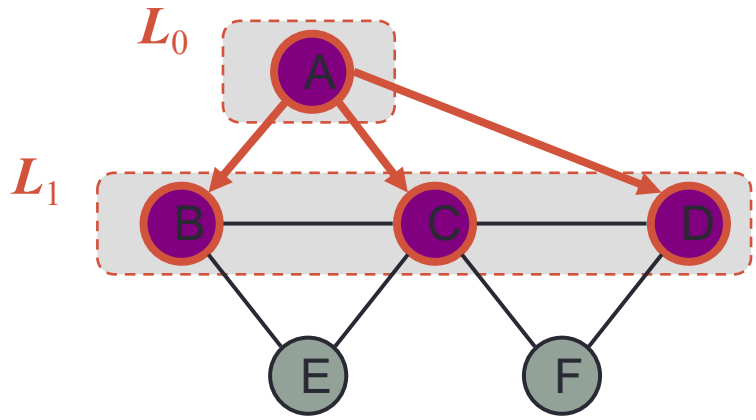
Example


A
unexplored vertex
A
visited vertex
unexplored edge
discovery edge
cross edge

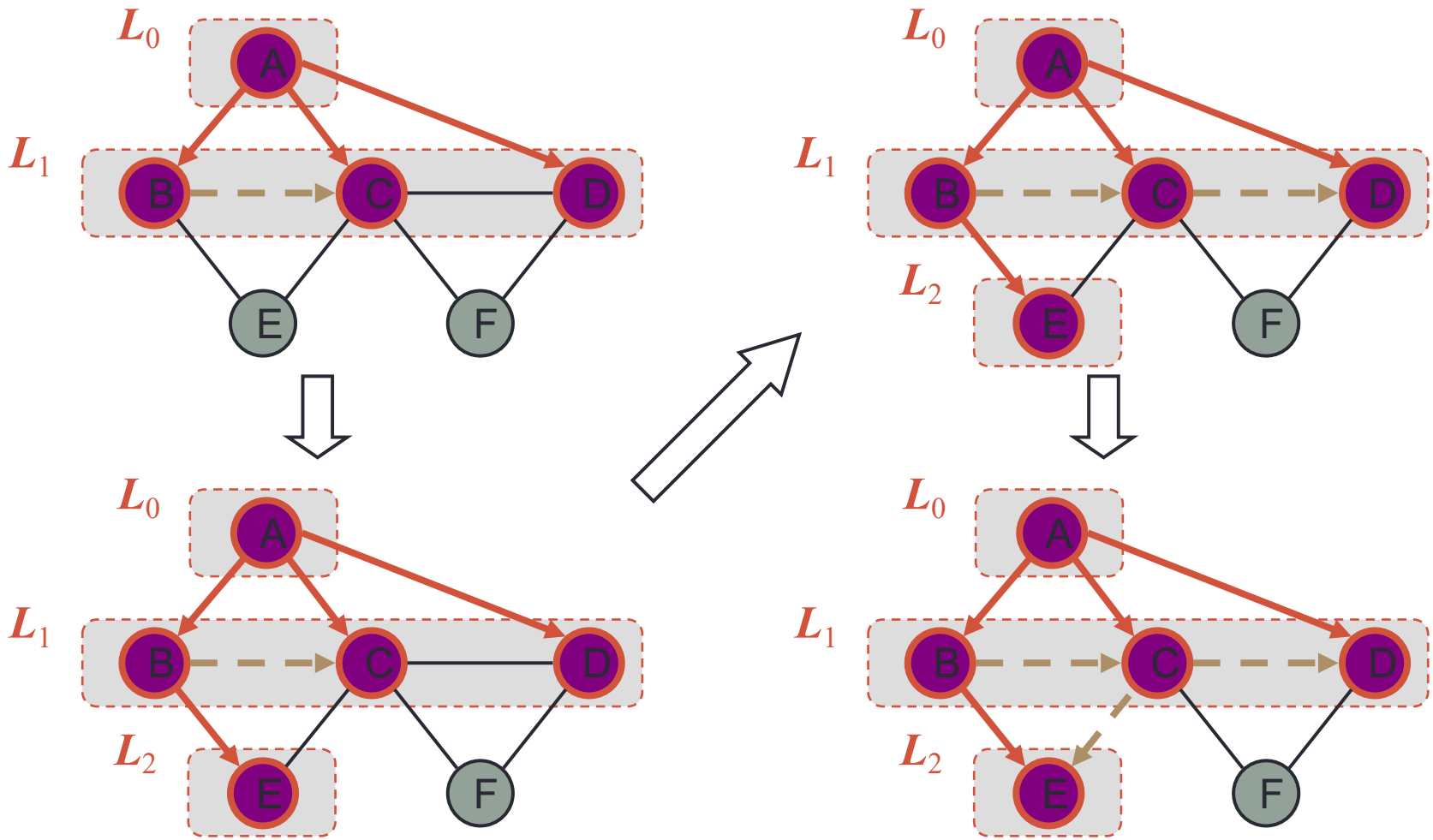

L_0
L_1
A
B
C
D
E
F



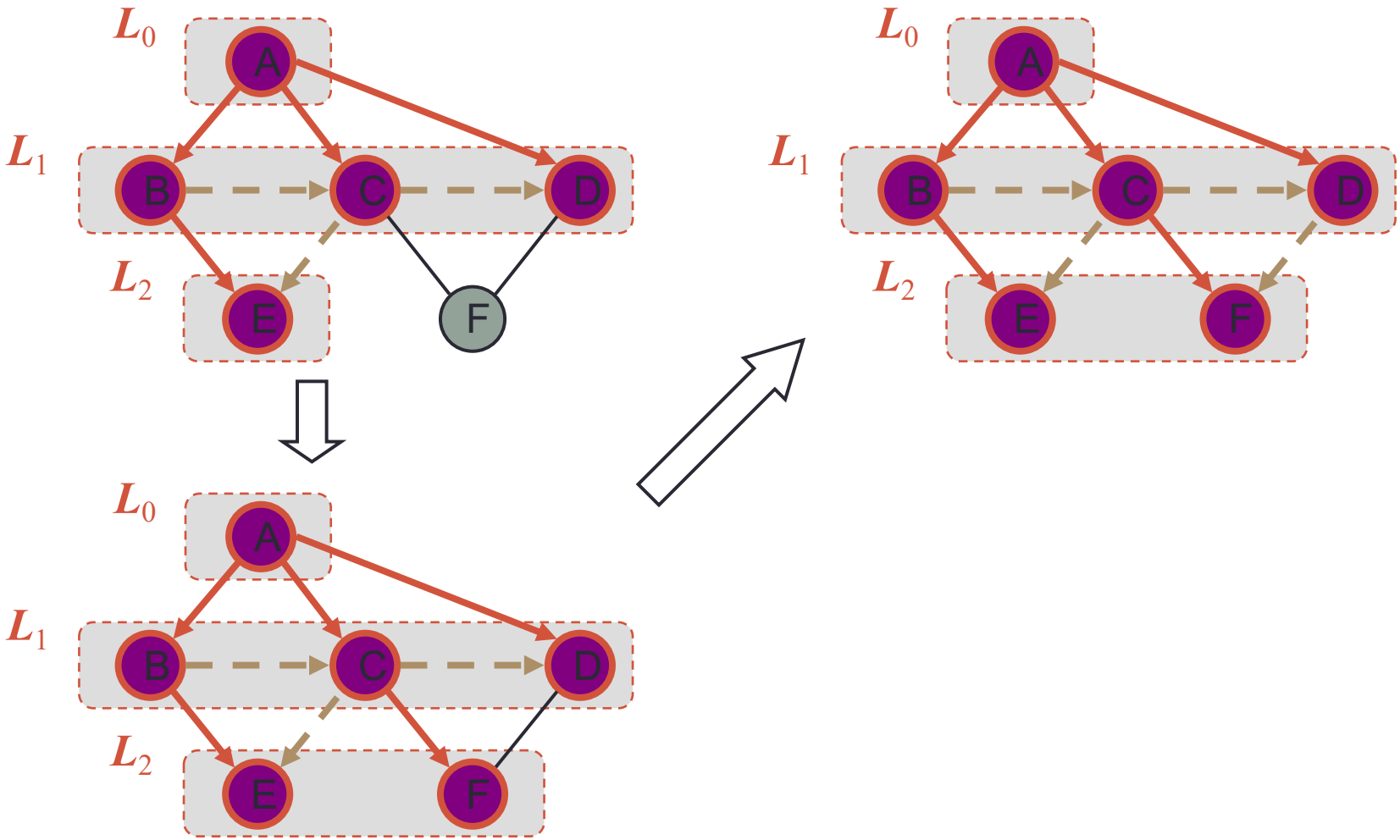

L_0
L_1
A
B
C
D
E
F


L_0
L_1
A
B
C
D
E
F

Example (2)



Example (3)



DEPTH FIRST SEARCH

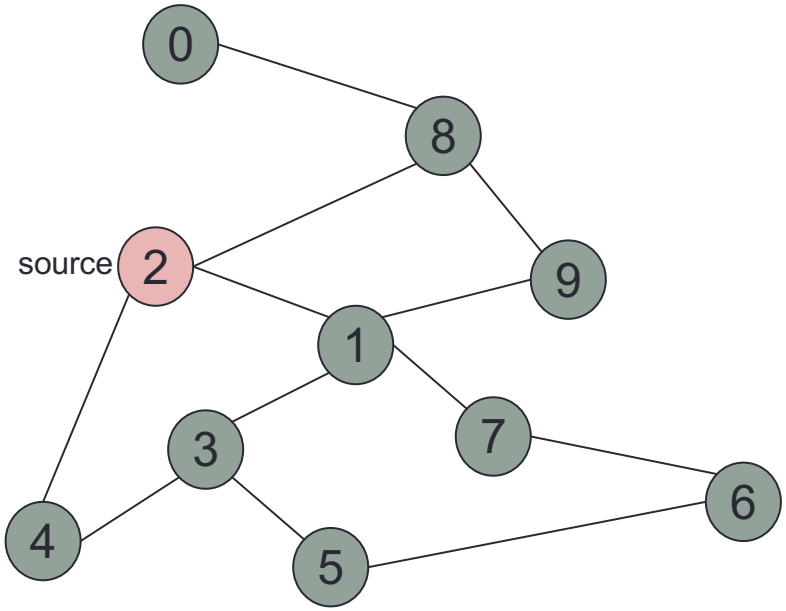
EECS 2011

Depth First Search (DFS)

- DFS is another popular graph search strategy
  - Idea is similar to pre-order traversal (visit node, then visit children recursively)

- DFS will continue to visit neighbors in a recursive pattern
  - Whenever we visit v from u, we recursively visit all unvisited neighbors of v. Then we backtrack (return) to u.

DFS Traversal 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	-

Pred

Initialize visited table (all False)

Initialize Pred to -1

DFS Algorithm Idea

- Similar to BFS algorithm, except that we use a stack instead of a queue for backtracking.

- In practice, we use recursion instead of an explicit stack.

BFS Algorithm

Algorithm $BFS(s)$
Input: s is the source vertex
Output: Mark all vertices that can be visited from s.

```
1.   for each vertex v
2.       do flag[v] := false;
3.   Q = empty queue;          T = empty stack;
4.   flag[s] := true;
5.   enqueue(Q, s);            T.push( s );
6.   while Q is not empty
7.       do v := dequeue(Q);   v = T.pop();
8.           for each w adjacent to v
9.               do if flag[w] = false
10.                  then flag[w] := true;
11.                       enqueue(Q, w)   T.push( w );
```

DFS Algorithm

```
Algorithm DFS(s)
1.    for each vertex v
2.        do flag[v] := false;
3.    RDFS(s);
```

Flag all vertices as not visited

```
Algorithm RDFS(v)
1.    flag[v] := true;    print v;
2.    for each neighbor w of v
3.        do if flag[w] = false
4.              then RDFS(w);
```

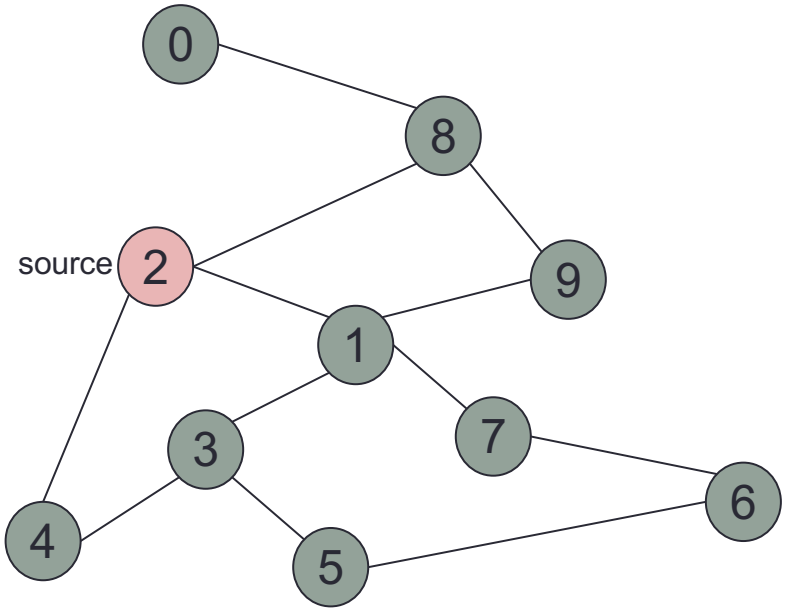
Flag v as visited; *print v*;

For unvisited neighbors, call *RDFS*(*w*) recursively

We can also record the paths using *prev*[ ].

Where do we insert the code for *prev*[ ]?

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

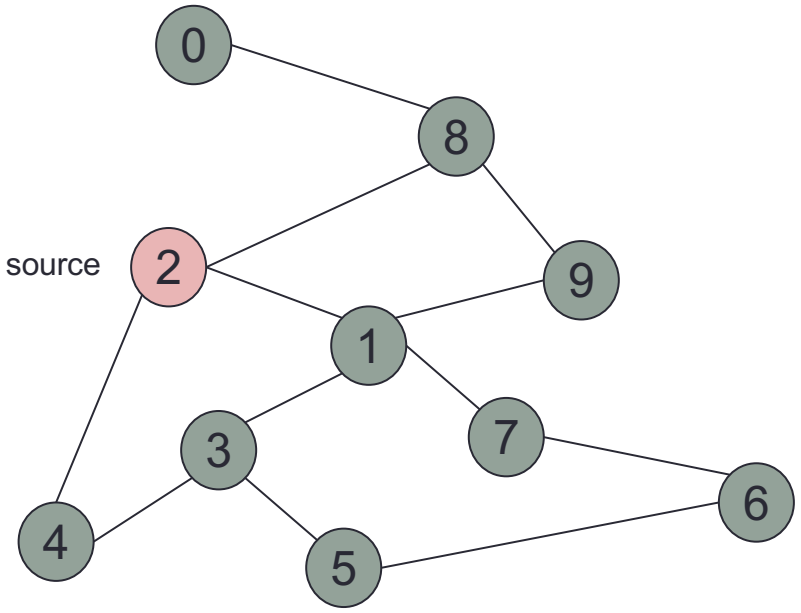
Visited Table (T/F)

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

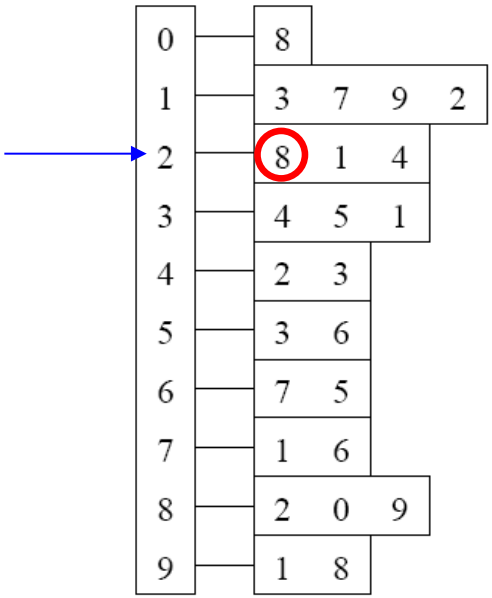
Pred

Initialize visited table (all False)

Initialize Pred to -1



Adjacency List



Visited Table (T/F)

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

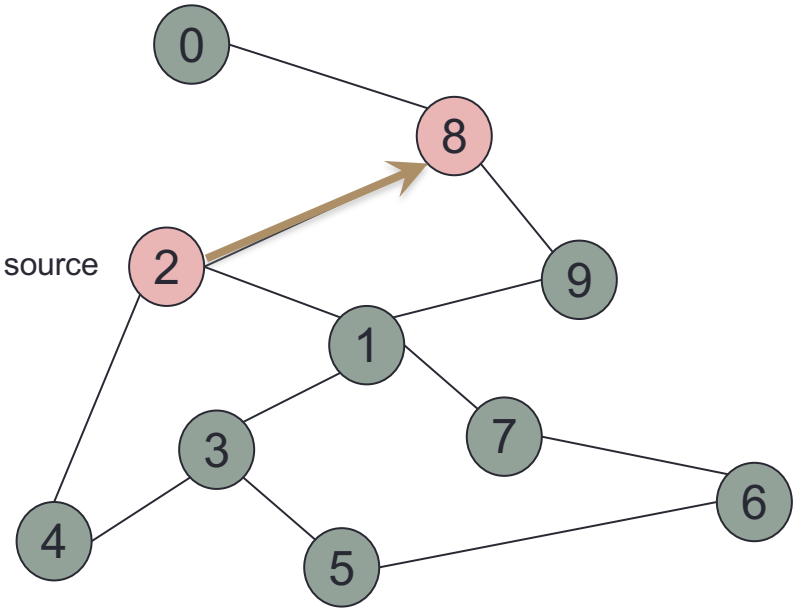
-
-
-
-
-
-
-
-
-
-

Pred

Mark 2 as visited

RDFS(2)

Now visit RDFS(8)



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	T	2
9	F	-

Pred

Mark 8 as visited

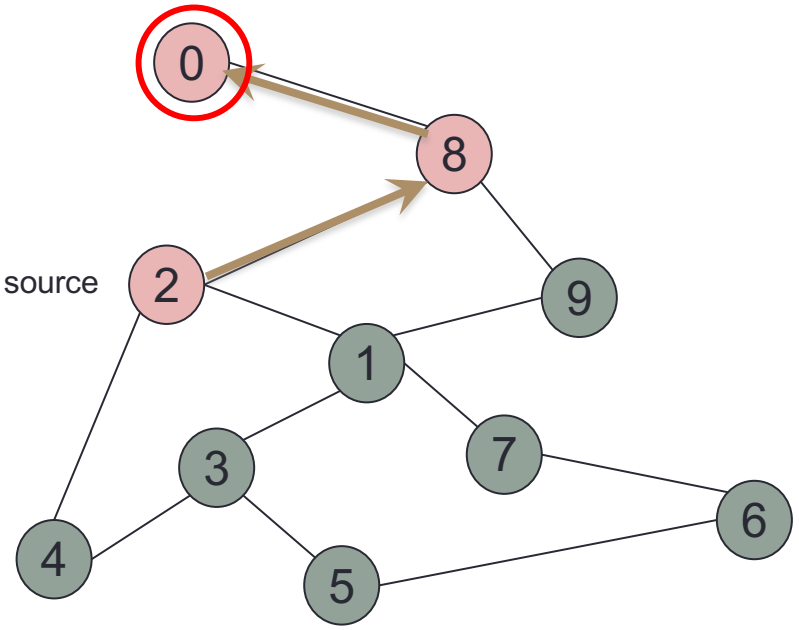
mark Pred[8]

Recursive calls

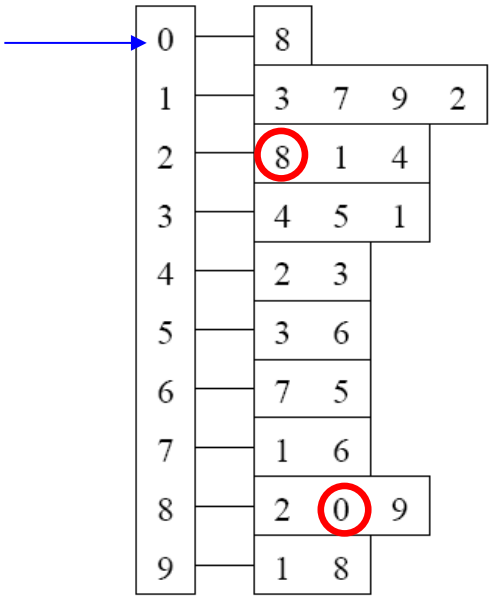
RDFS(2)

RDFS(8)

2 is already visited, so visit RDFS(0)



Adjacency List



Visited Table (T/F)

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

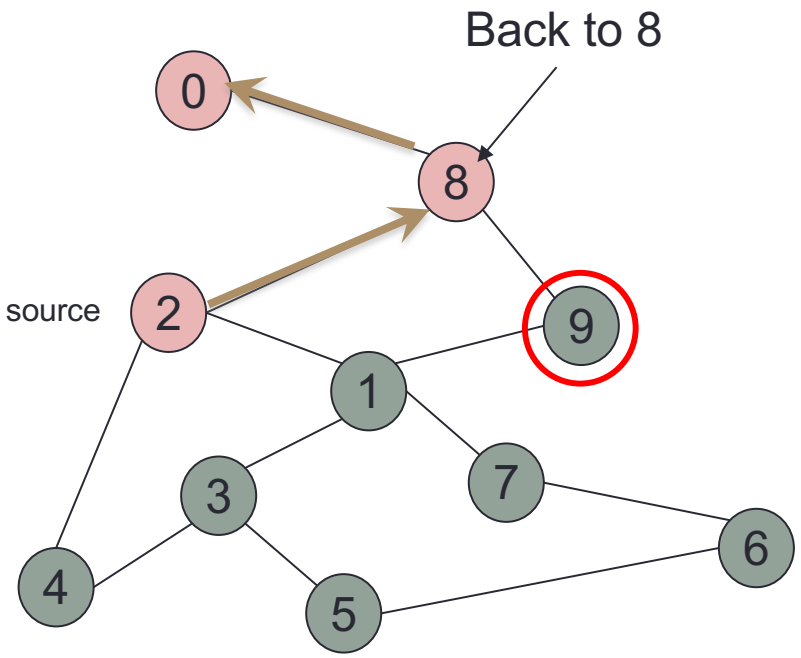
Pred
8
-
-
-
-
-
-
-
2
-

Mark 0 as visited

Mark Pred[0]

Recursive calls

RDFS(2)
RDFS(8)
RDFS(0) -> no unvisited neighbors, return to call RDFS(8)



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

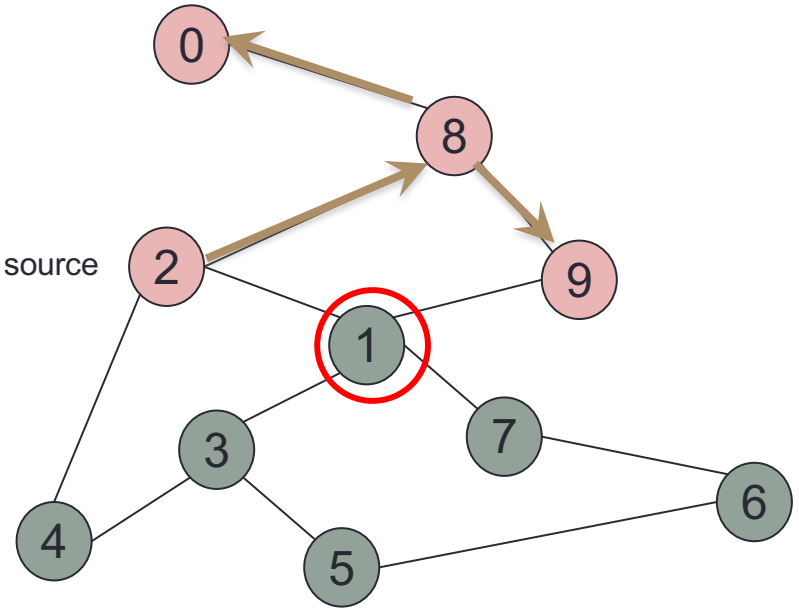
Visited Table (T/F)

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

Pred

Recursive calls

RDFS(2)
RDFS(8)
Now visit 9 -> RDFS(9)



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

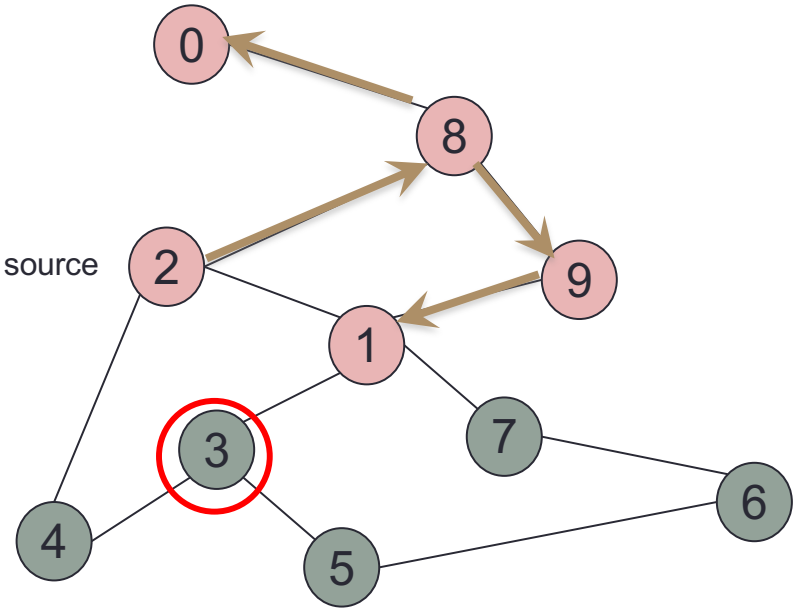
Pred

Mark 9 as visited

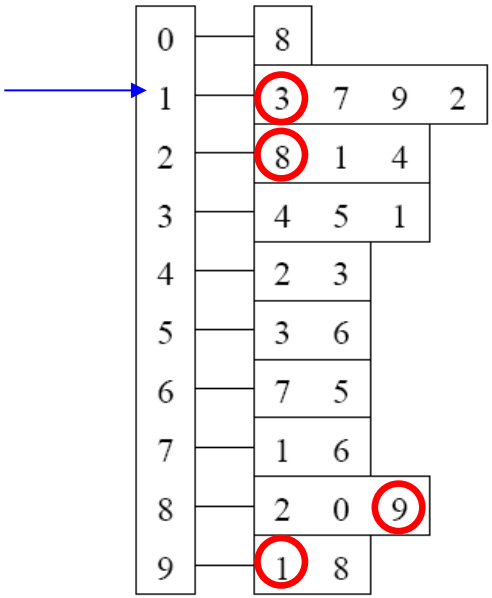
Mark Pred[9]

Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
-> visit 1, RDFS(1)



Adjacency List



Visited Table (T/F)

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

8
9
-
-
-
-
-
-
2
8

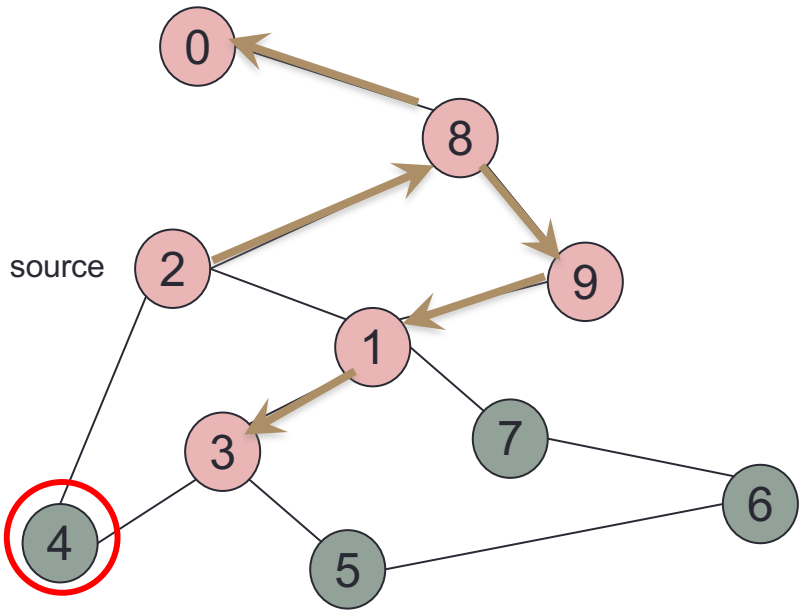
Pred

Mark 1 as visited

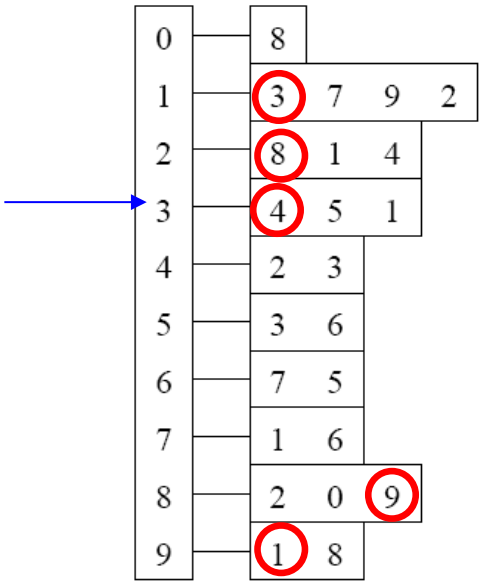
Mark Pred[1]

Recursive calls

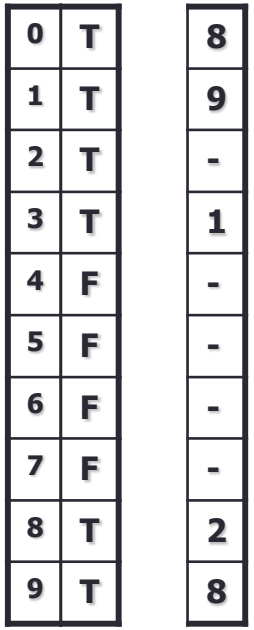
RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
visit RDFS(3)



Adjacency List



Visited Table (T/F)

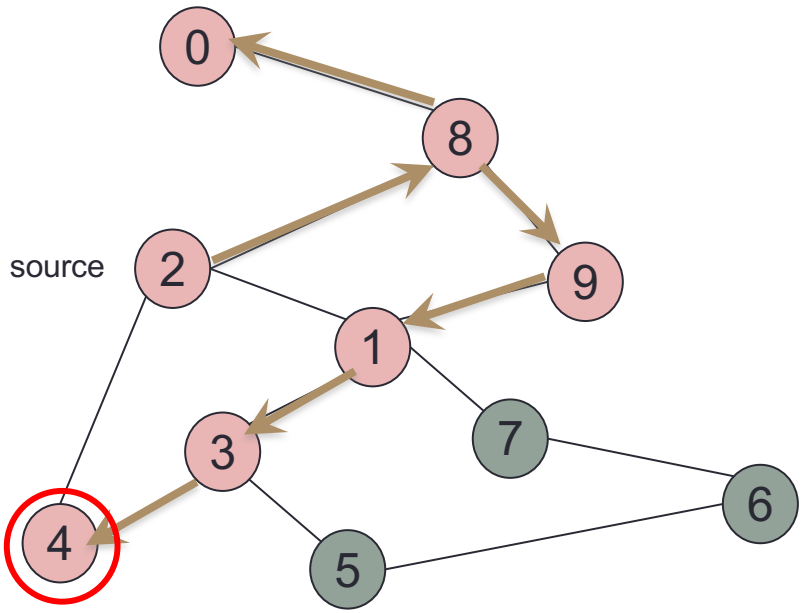


Mark 3 as visited

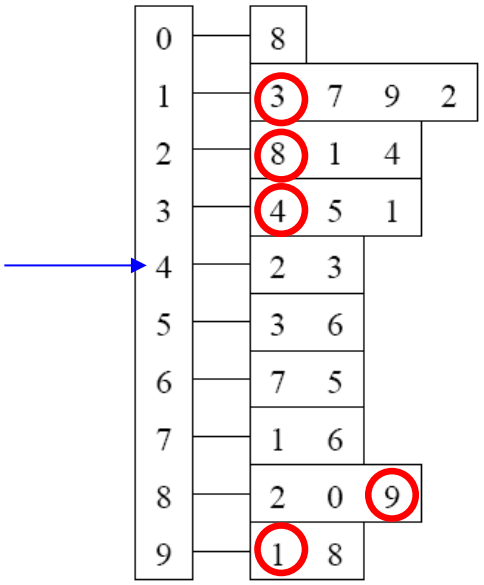
Mark Pred[3]

Recursive calls

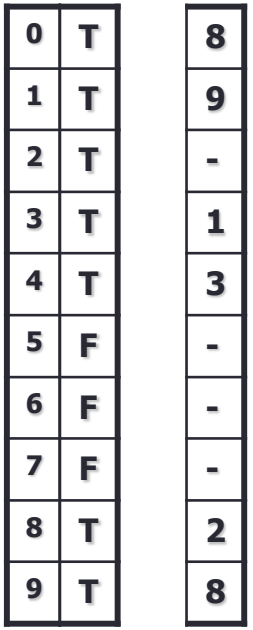
RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
visit RDFS(4)



Adjacency List



Visited Table (T/F)

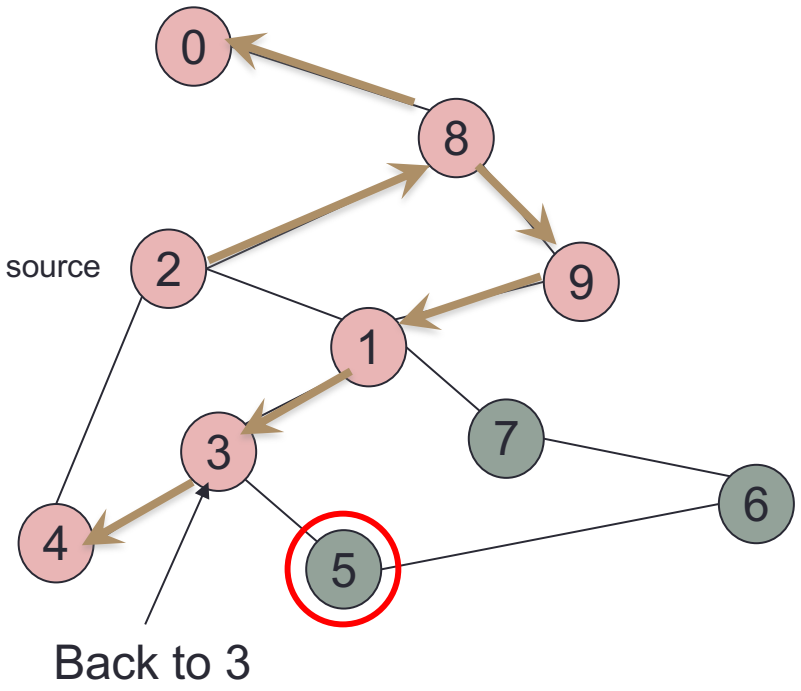


Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(4) → STOP all of 4's neighbors have been visited
return back to call RDFS(3)

Mark 4 as visited

Mark Pred[4]



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

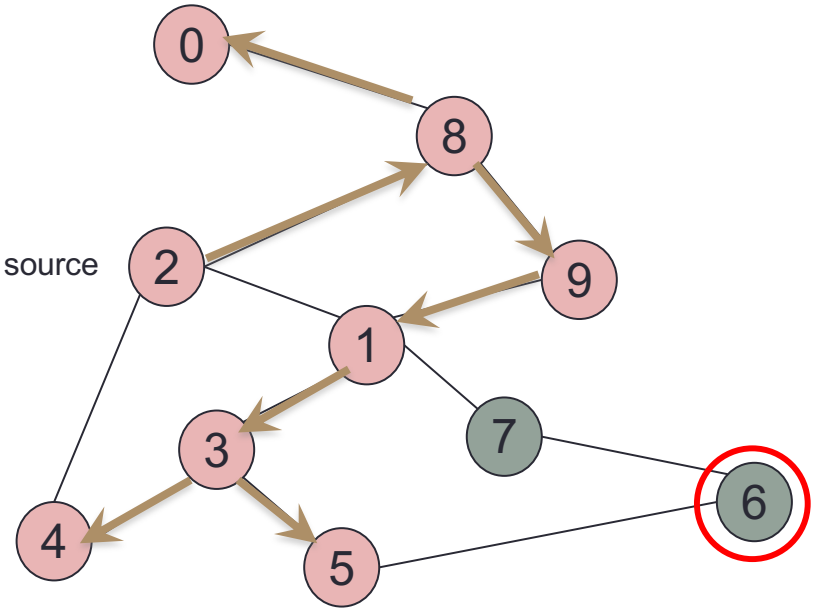
Visited Table (T/F)

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

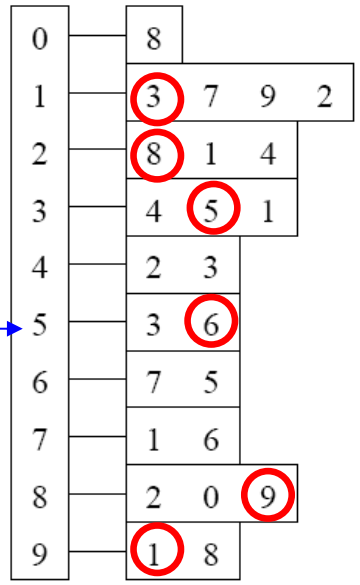
Pred

Recursive calls

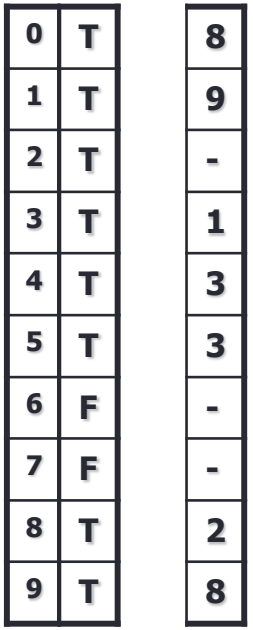
RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
visit 5 -> RDFS(5)



Adjacency List



Visited Table (T/F)



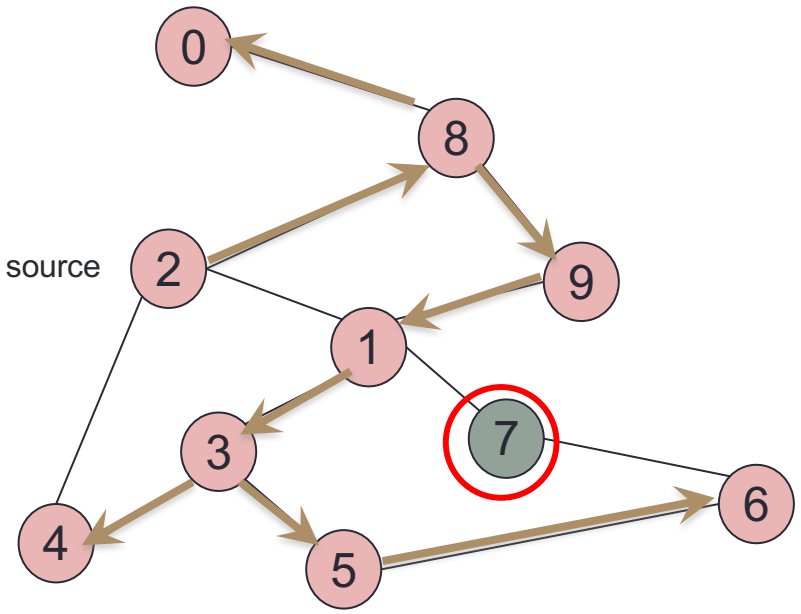
Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5)

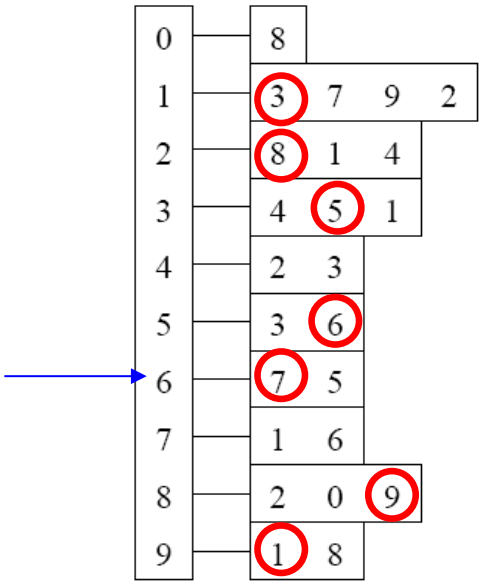
Mark 5 as visited

Mark Pred[5]

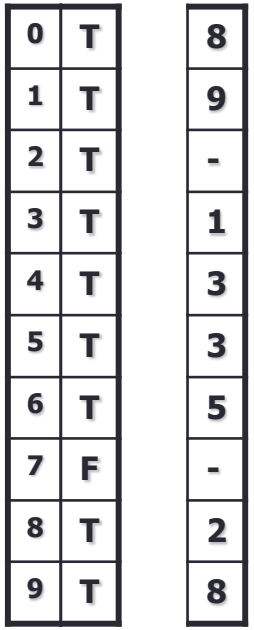
3 is already visited, so visit 6 -> RDFS(6)



Adjacency List



Visited Table (T/F)

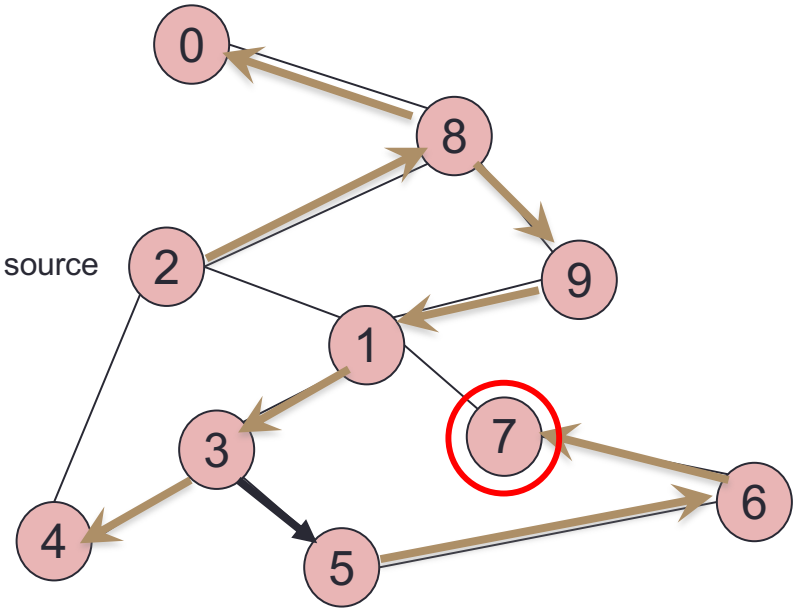


Recursive calls

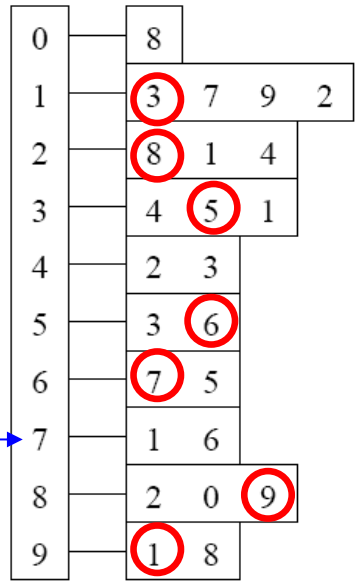
RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5)
RDFS(6)
visit 7 -> RDFS(7)

Mark 6 as visited

Mark Pred[6]



Adjacency List



Visited Table (T/F)

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

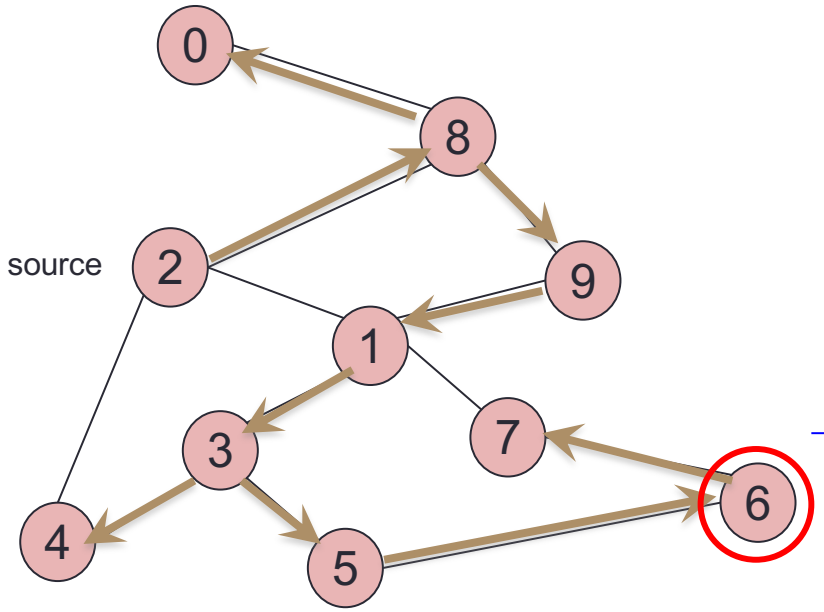
Pred
8
9
-
1
3
3
5
6
2
8

Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5)
RDFS(6)
RDFS(7) -> Stop no more unvisited neighbors

Mark 7 as visited

Mark Pred[7]



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

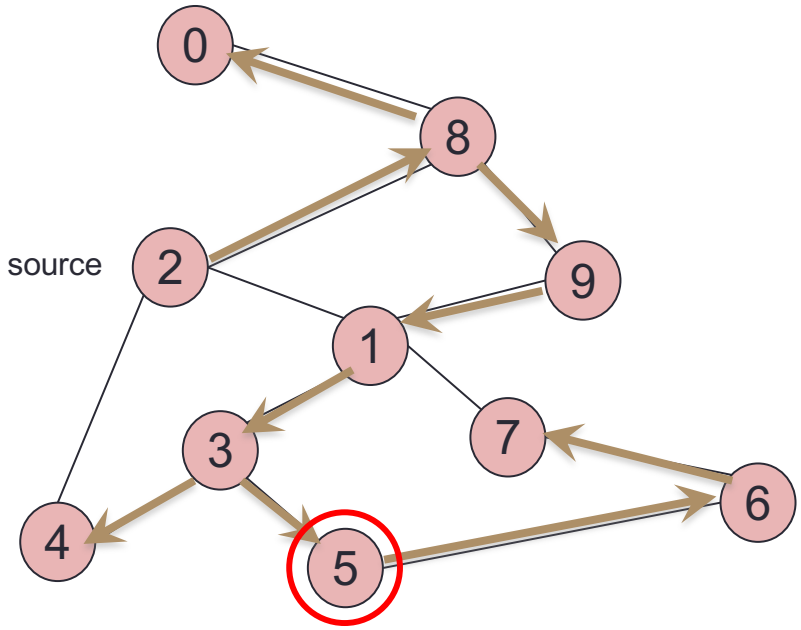
Visited Table (T/F)

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

Pred
8
9
-
1
3
3
5
6
2
8

Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5)
RDFS(6) -> Stop



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

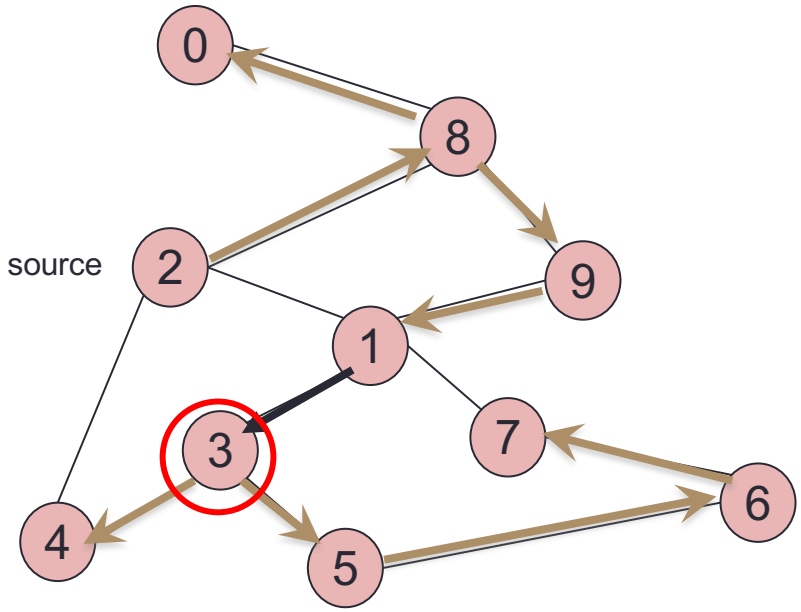
Visited Table (T/F)

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

Pred

Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5) -> Stop



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

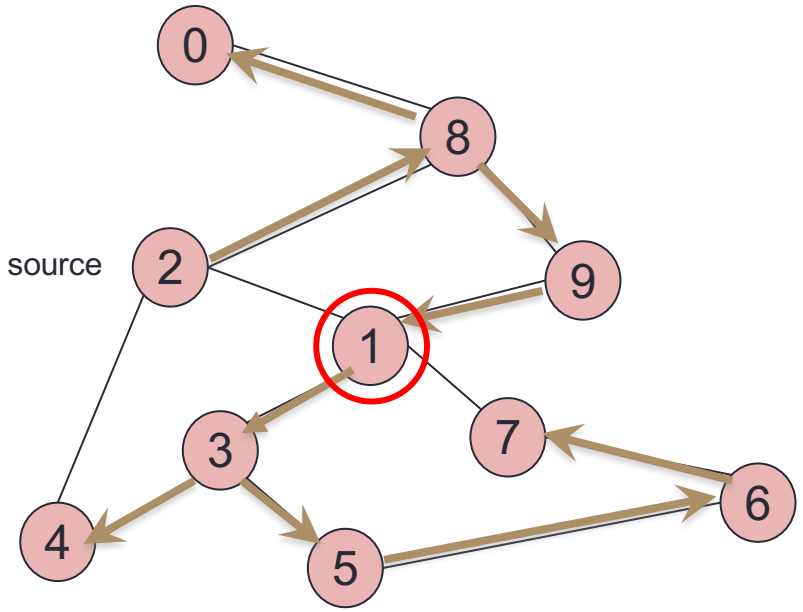
Visited Table (T/F)

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

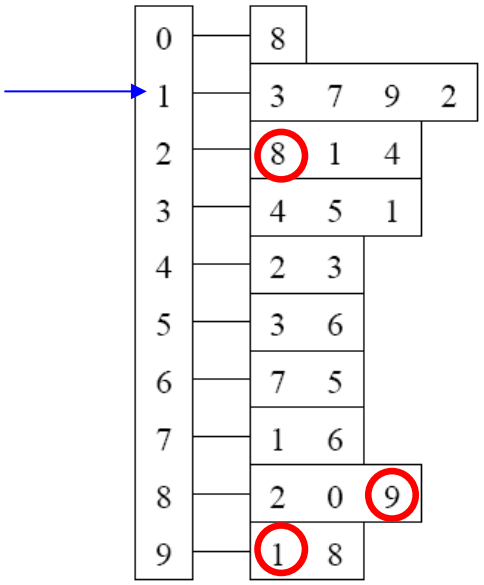
Pred

Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3) -> Stop



Adjacency List



Visited Table (T/F)

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

8
9
-
1
3
3
5
6
2
8

Pred

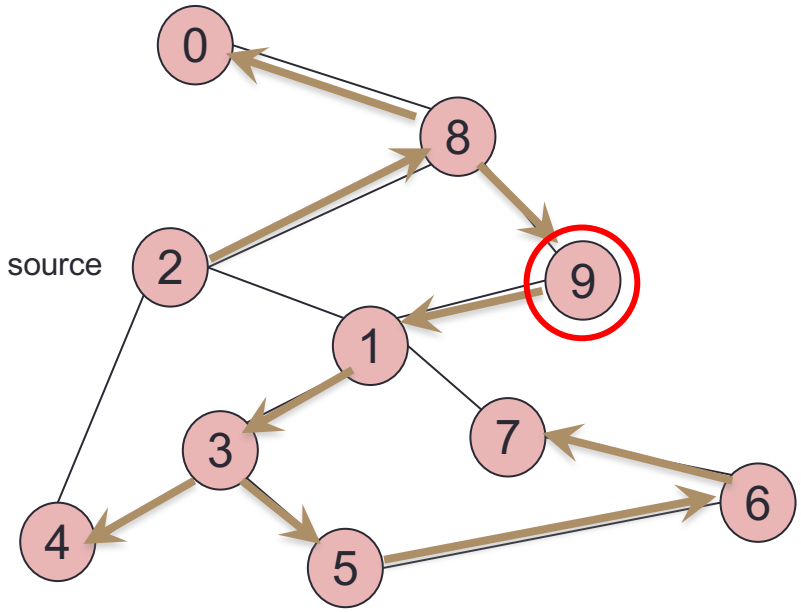
RDFS(2)

RDFS(8)

RDFS(9)

RDFS(1) -> Stop

Recursive calls



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

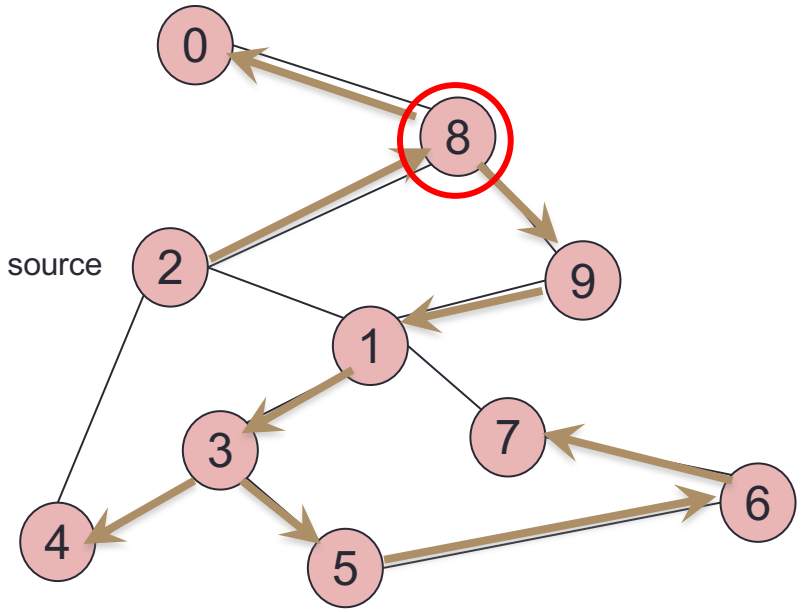
Pred
8
9
-
1
3
3
5
6
2
8

Recursive calls

RDFS(2)

RDFS(8)

RDFS(9) -> Stop



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

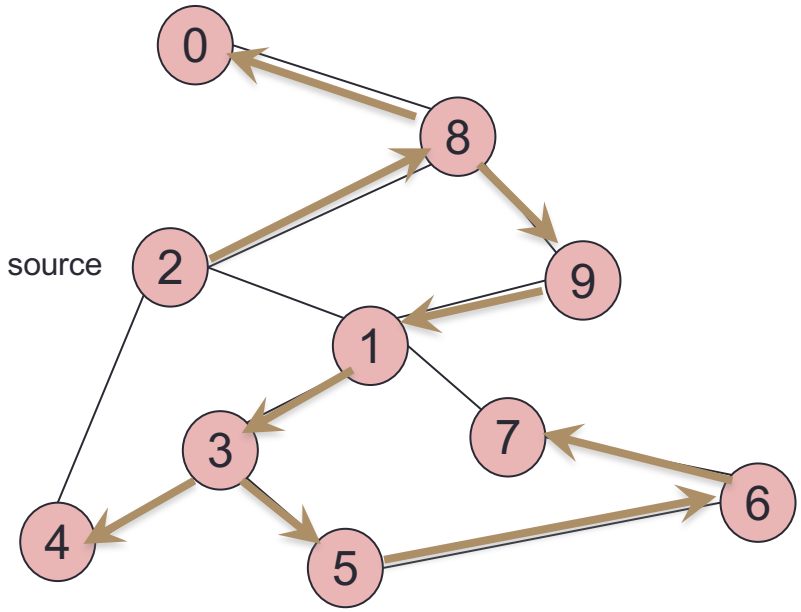
Pred
8
9
-
1
3
3
5
6
2
8

Recursive calls

RDFS(2)

RDFS(8) -> Stop

Example Finished



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

8
9
-
1
3
3
5
6
2
8

Pred

RDFS(2) -> Stop

Recursive calls finished.

Time Complexity of DFS

- We never visited a vertex more than once.
- We had to examine the adjacency lists of all vertices.
  - $\Sigma_{\text{vertex } v} \text{degree}(v) = 2E$
- So, the running time of DFS is proportional to the number of edges and number of vertices (same as BFS)
  - $O(V + E)$
- What is the running time of DFS if we use an adjacency matrix?

Enhanced DFS Algorithm

- What if a graph is not connected (strongly connected)?
  - Use an enhanced version of DFS, which is similar to the enhanced BFS algorithm.

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

Applications of DFS

- Is there a path from source *s* to a vertex *v*?
- Is an undirected graph connected?
- To output the contents of a graph
- To find the connected components of a graph
- To find out if a graph contains cycles and report cycles.
- To construct a DSF tree/forest from a graph

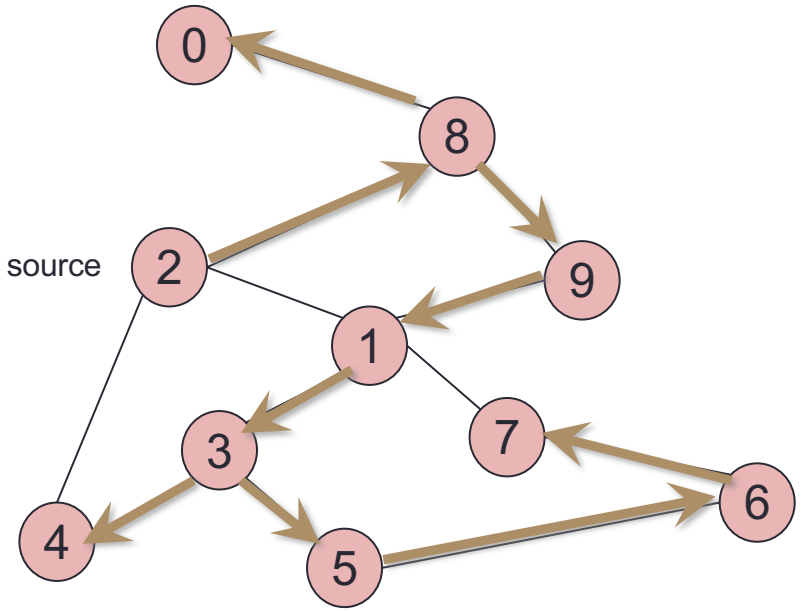
APPLICATIONS OF DFS

EECS 2011

Applications of DFS

- Is there a path from source *s* to a vertex *v*?
  - Check array *flag*[ ]
- Is an undirected graph connected?
  - Check array *flag*[ ]
- To output the contents (e.g., the vertices) of a graph
  - Call *RDFS*( ) if graph is connected
  - Call *DFSearch*( ) if the "connected" property is not known
- To find the connected components of a graph
  - Call *DFSearch*( )
- To find out if a graph contains cycles and report cycles.
- To construct a DSF tree/forest from a graph

DFS Path Tracking



DFS finds out path too.

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

8
9
-
1
3
3
5
6
2
8

Pred

Try some examples.
Path(0) ->
Path(6) ->
Path(7) ->

Finding Cycles Using DFS

- Similar to using BFS.

- For undirected graphs, classify the edges into 3 categories during program execution: *unvisited* edge, *discovery* edge, and *back* edge (equivalent to cross edge in BFS).
  - If there exists a *back* edge, the undirected graph contains a cycle.

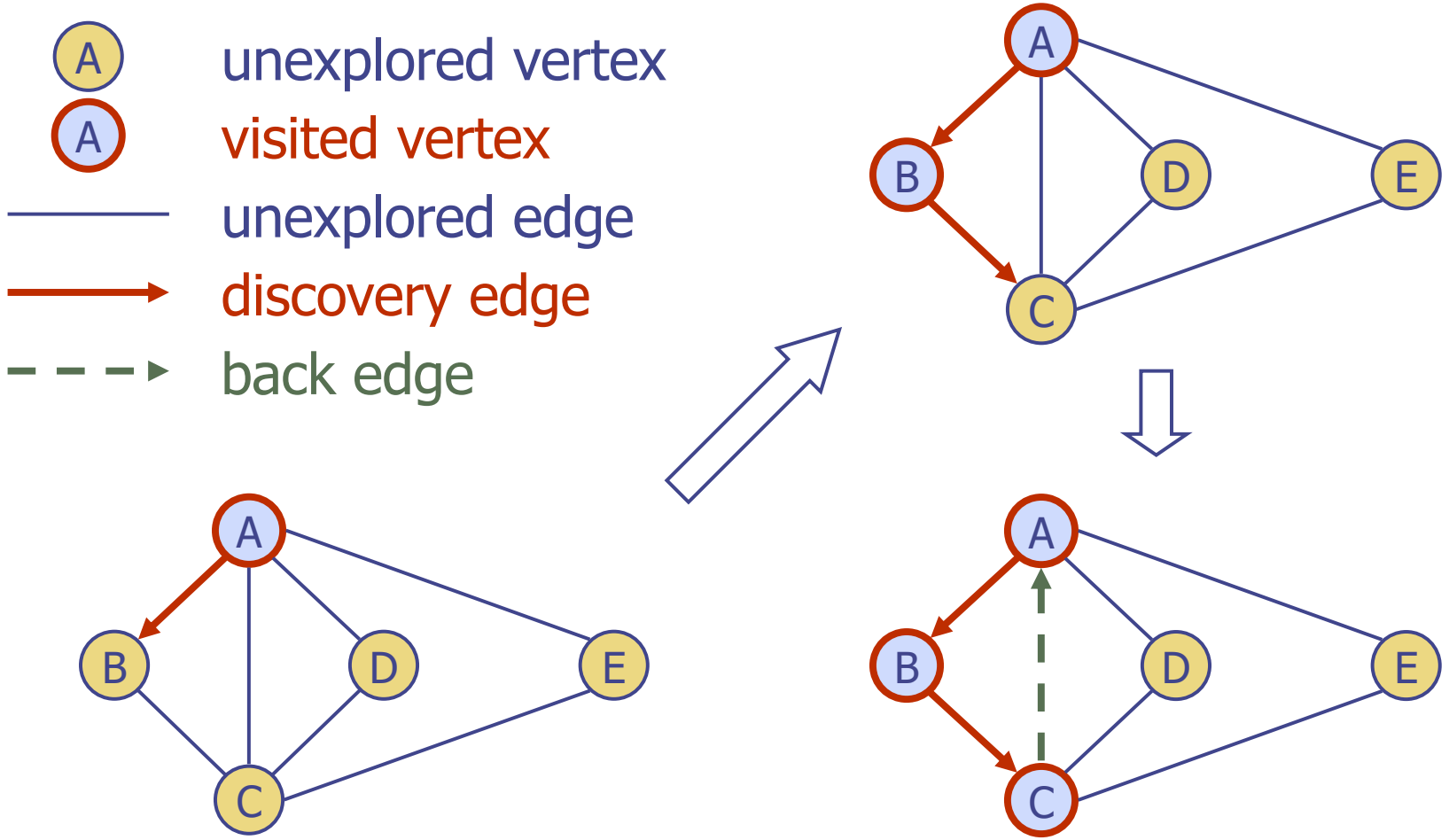
DFS Algorithm

- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

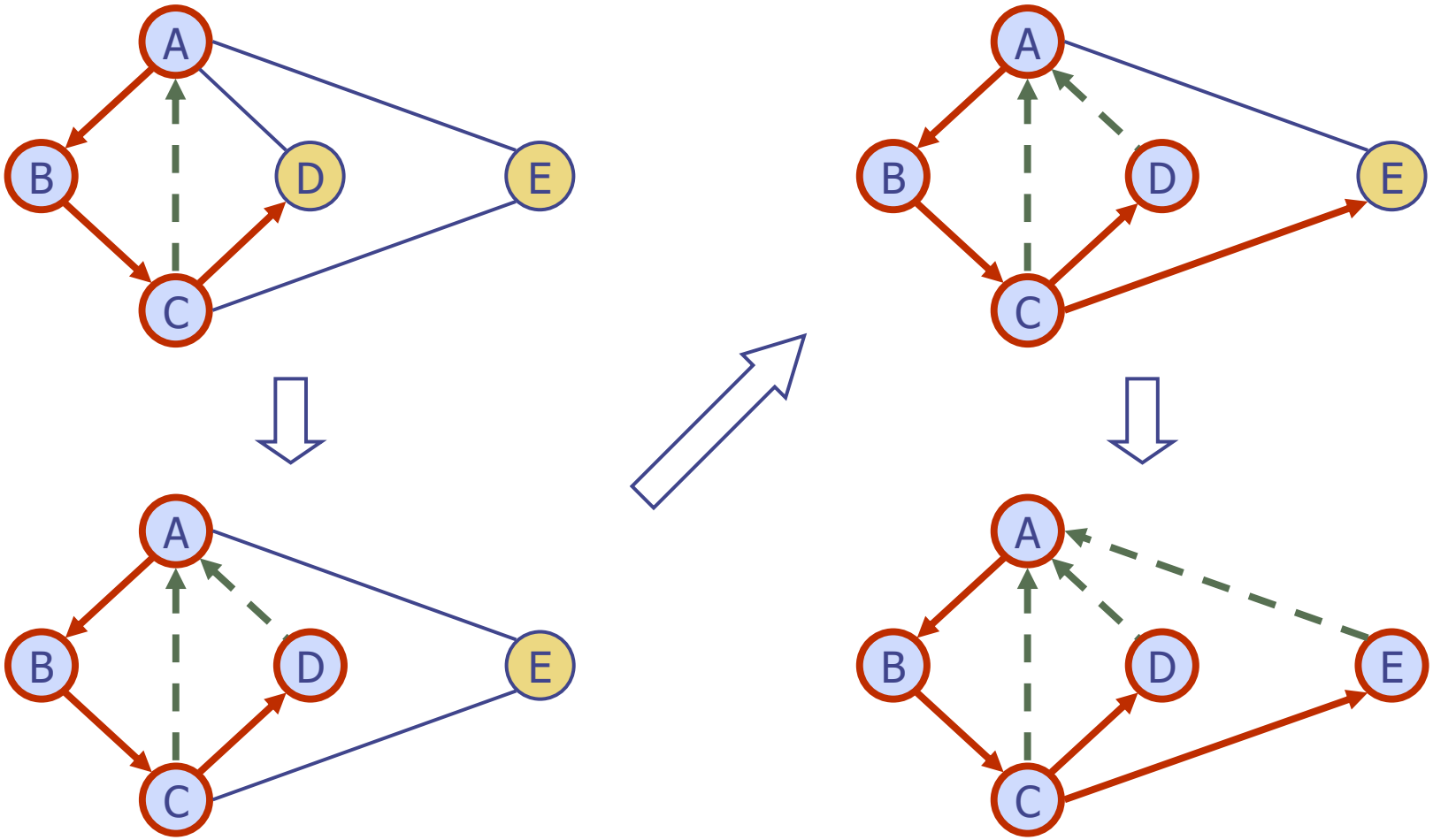
Algorithm *DFS*(*G*)
  Input graph *G*
  Output labeling of the edges of *G* as discovery edges and back edges
  for all *u* ∈ *G.vertices*()
    setLabel(*u*, *UNEXPLORED*)
  for all *e* ∈ *G.edges*()
    setLabel(*e*, *UNEXPLORED*)
  for all *v* ∈ *G.vertices*()
    if *getLabel*(*v*) = *UNEXPLORED*
      DFS(*G*, *v*)

Algorithm *DFS*(*G*, *v*)
  Input graph *G* and a start vertex *v* of *G*
  Output labeling of the edges of *G* in the connected component of *v* as discovery edges and back edges
  setLabel(*v*, *VISITED*)
  for all *e* ∈ *G.incidentEdges*(*v*)
    if *getLabel*(*e*) = *UNEXPLORED*
      w ← *opposite*(*v*,*e*)
      if *getLabel*(*w*) = *UNEXPLORED*
        setLabel(*e*, *DISCOVERY*)
        DFS(*G*, *w*)
      else
        setLabel(*e*, *BACK*)

Example


A
unexplored vertex
A
visited vertex
unexplored edge
discovery edge
back edge
A
B
D
E
C
A
B
D
E
C
A
B
D
E
C

Example (cont.)

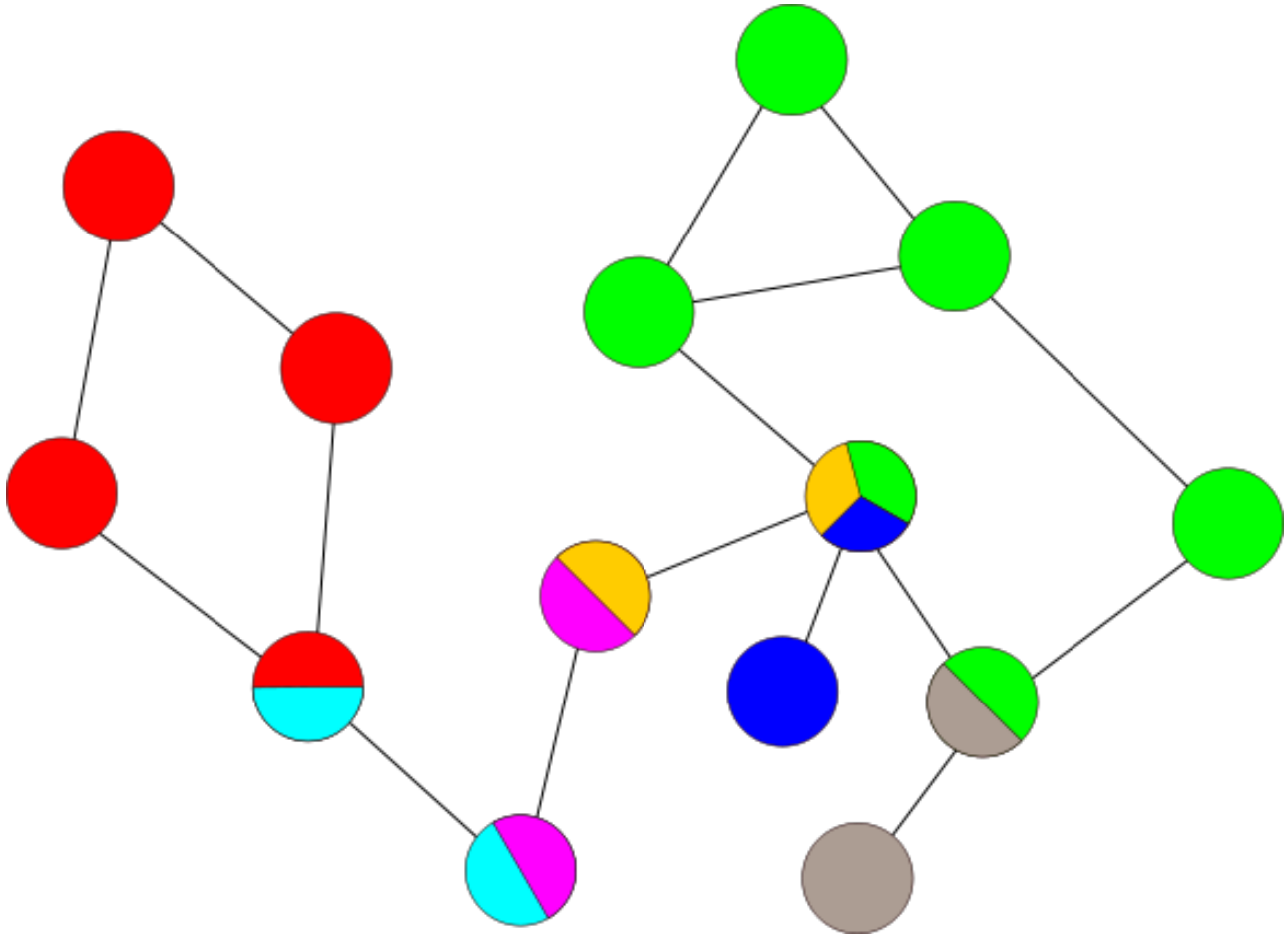


Applications – DFS vs. BFS

- What can BFS do and DFS can't?
  - Finding shortest paths (in unweighted graphs)

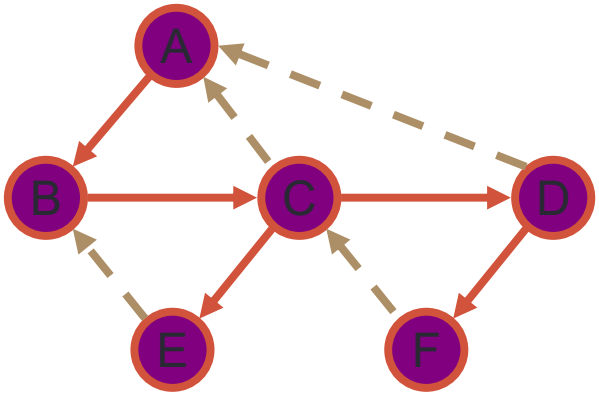
- What can DFS do and BFS can't?
  - Finding out if a connected undirected graph is *biconnected*
    - A connected undirected graph is biconnected if there are no vertices whose removal disconnects the rest of the graph.
    - Application in computer networks: ensuring that a network is still connected when a router/link fails.

A graph that is not biconnected

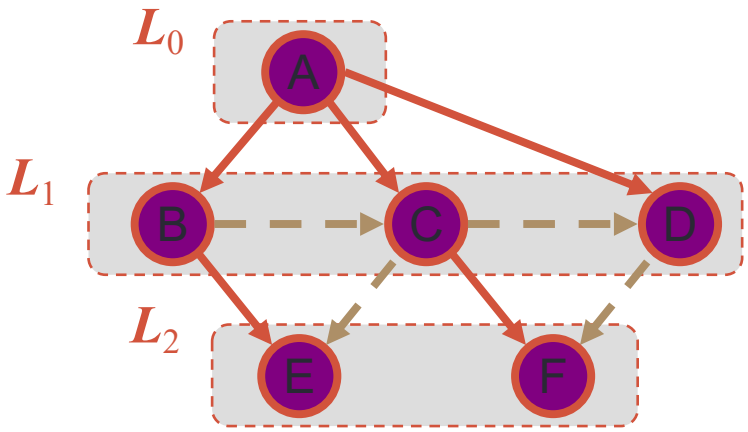


DFS vs. BFS

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	√	√
Shortest paths		√
Biconnected components	√	



DFS



BFS