RECITATION 2
ANALYSIS OF ALGORITHMS II

2016 SPRING

Outline

- Problem – 1 BFS Example
- Problem – 2 BFS and Shortest Path Problem
- Problem – 3 DFS Example
- Problem – 4 Bipartite Graph Example
- Problem – 5 Greedy Alg, *Coffee Shop Problem*

Graph Traversal

- Application examples
  - 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)**
  - **Depth-First Search (DFS)**

Reference: http://www.eecs.yorku.ca/course_archive/2006-07/W/2011/Notes/

Data Structures for DFS and BFS Implementation

- What is the main data structures employed when implementing DFS and BFS?

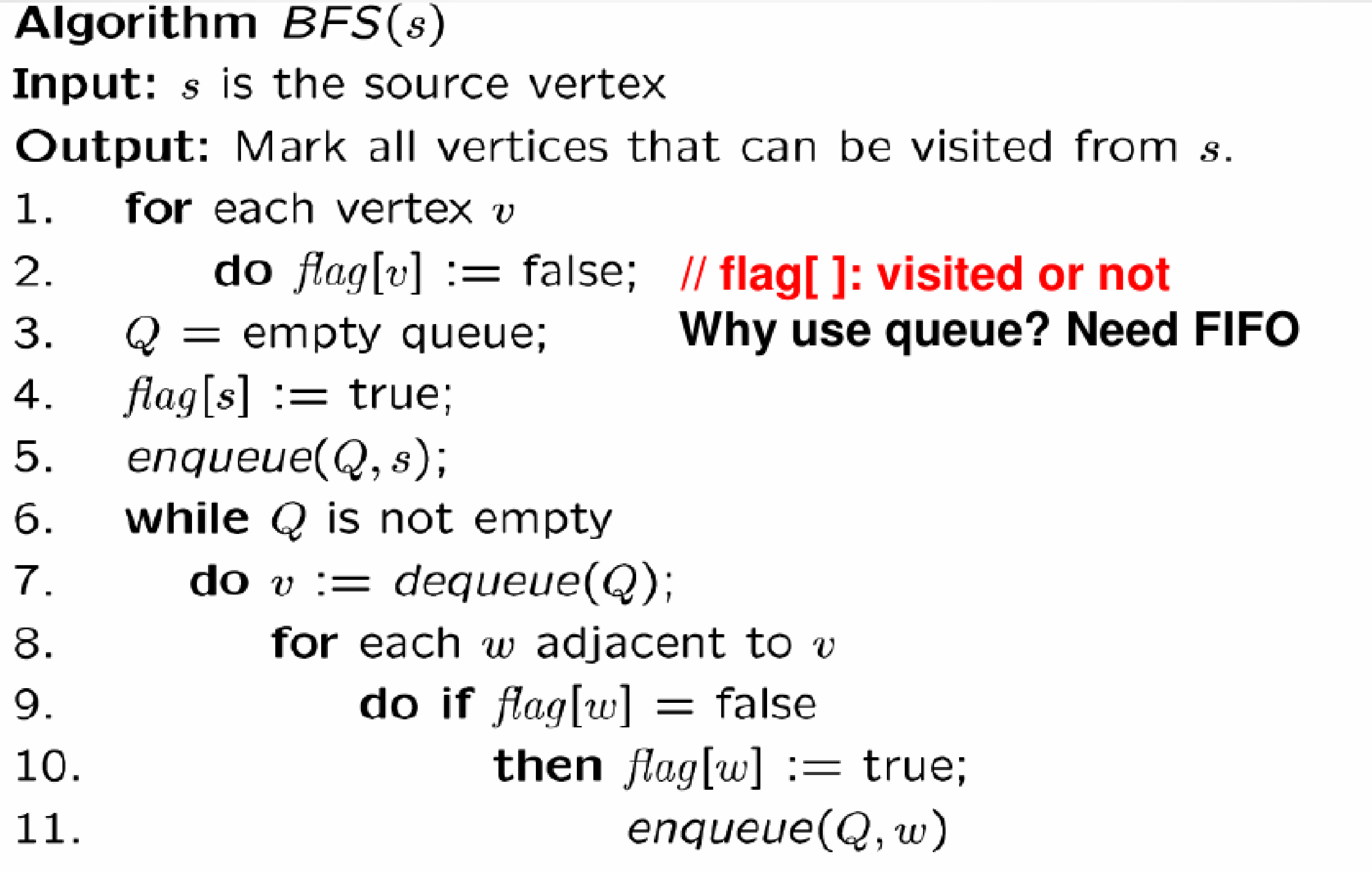
✓ BFS → QUEUE (FIFO Queue can be used)

- – We trace the graph layer by layer by considering all of the children of a node before starting to trace nodes further away

✓ DFS → STACK

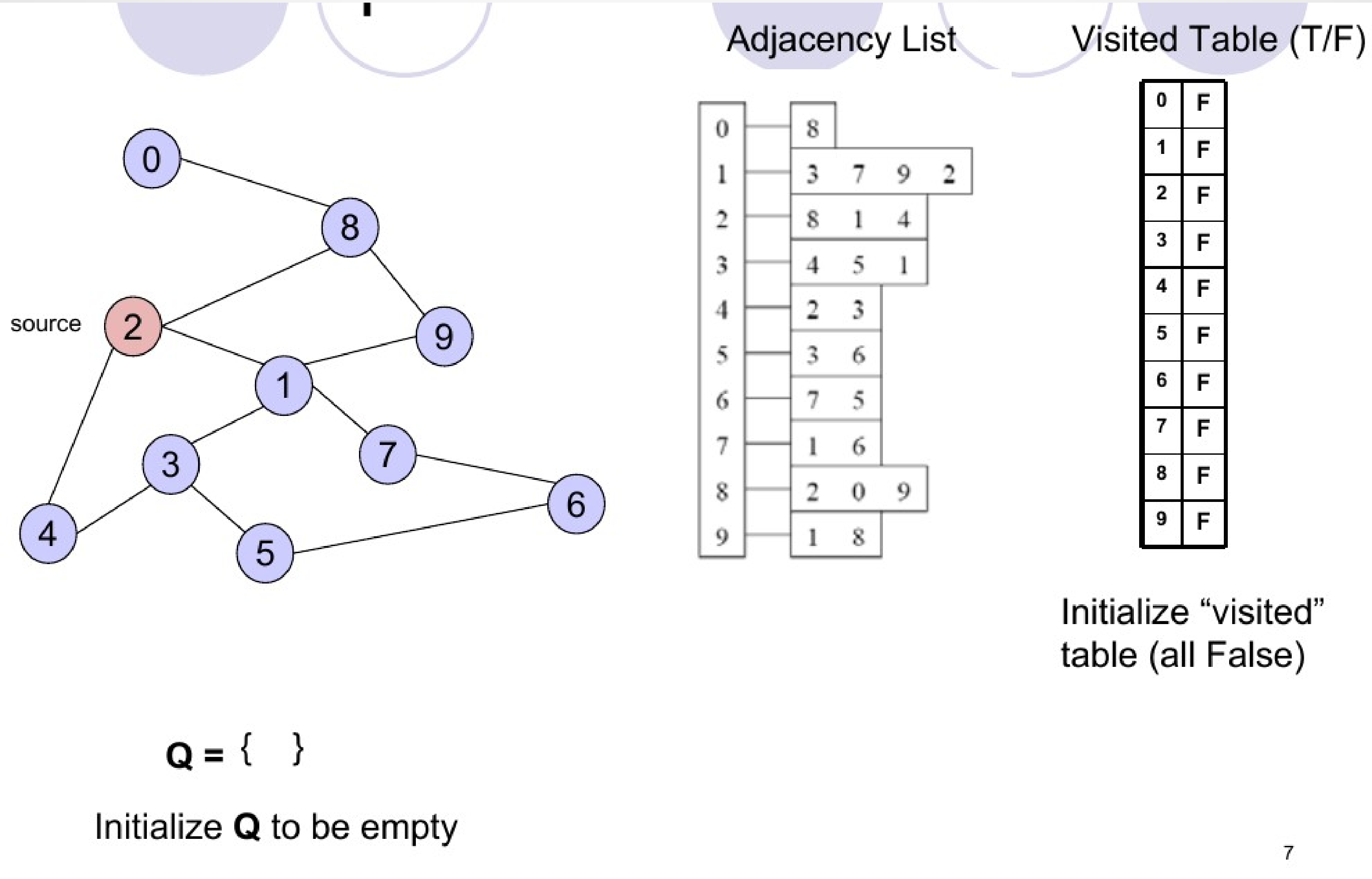
- – Algorithm considers the immediate unexplored children before considering the other children of a node while moving from parent node to child node.
- – It goes deeper through the branches of the graph, before tracing other branches.

BFS Algorithm using Queue



```
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;      // flag[ ]: visited or not
$Q$ = empty queue;              Why use queue? Need FIFO
$flag[s]$ := 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;
                     $enqueue(Q, w)$
```

Problem 1 - 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

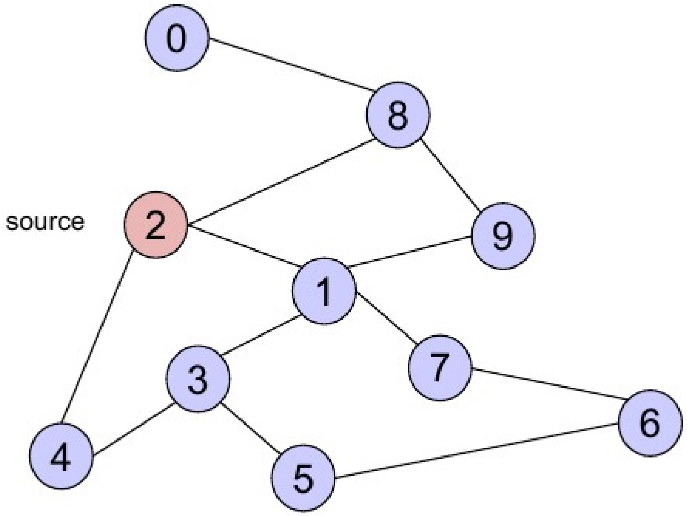
Initialize “visited” table (all False)

Q = { }

Initialize **Q** to be empty

BFS Example Reference: http://www.eecs.yorku.ca/course_archive/2006-07/W/2011/Notes/

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)

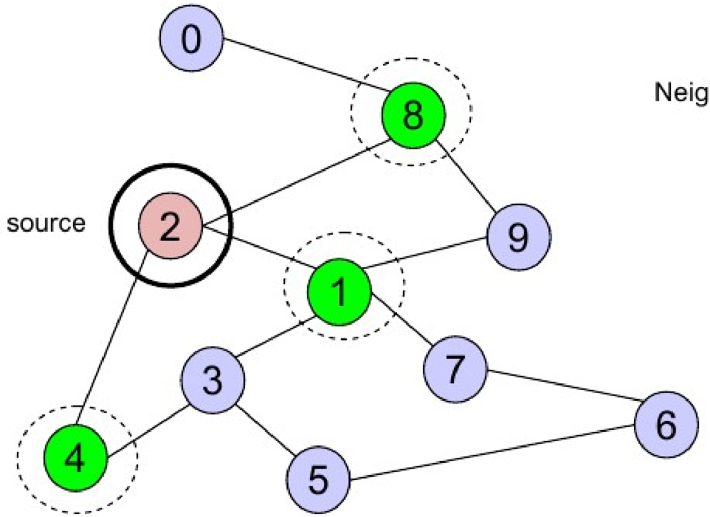
0	F
1	F
2	T
3	F
4	F
5	F
6	F
7	F
8	F
9	F

Flag that 2 has been visited

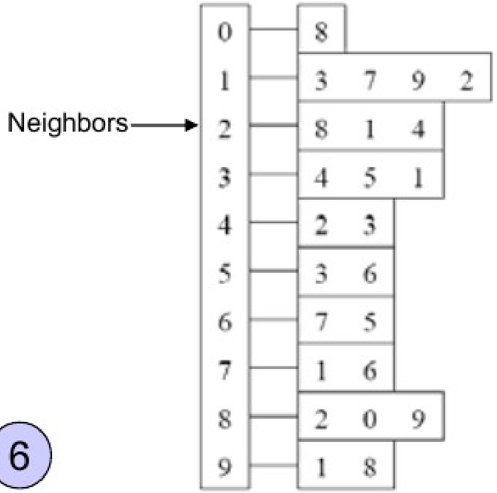
Q = { 2 }

Place source 2 on the queue

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)

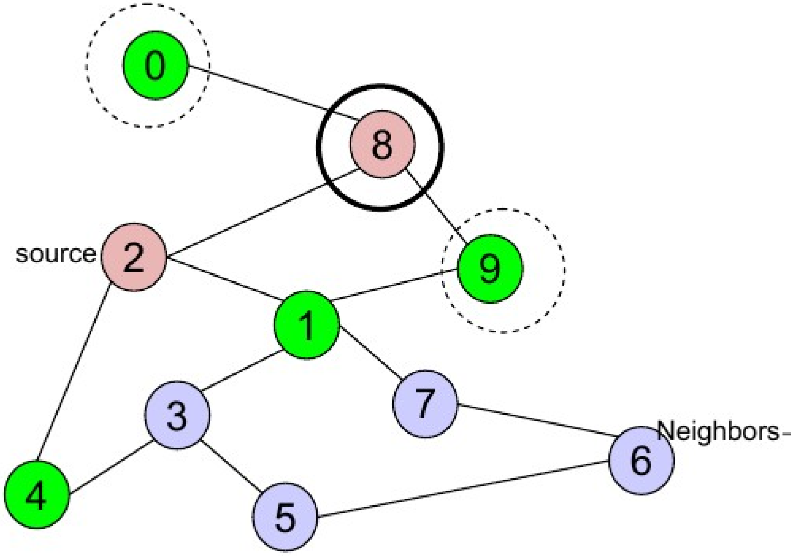
0	F
1	T
2	T
3	F
4	T
5	F
6	F
7	F
8	T
9	F

Mark neighbors as visited 1, 4, 8

Q = {2} → { 8, 1, 4 }

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

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)

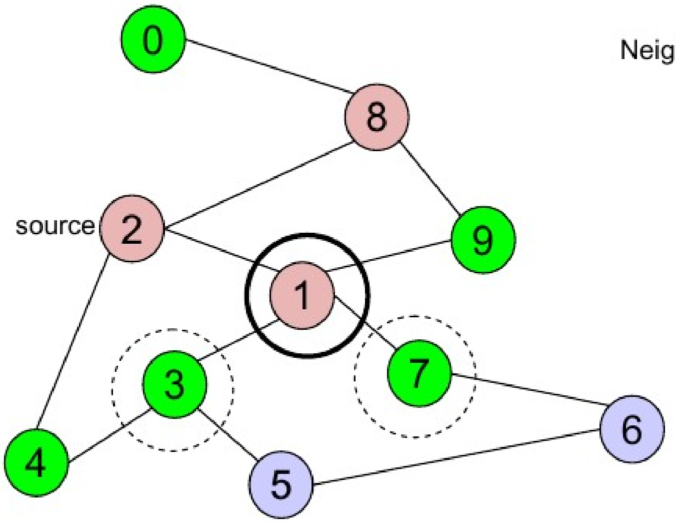
0	T
1	T
2	T
3	F
4	T
5	F
6	F
7	F
8	T
9	T

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!

BFS Example



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)

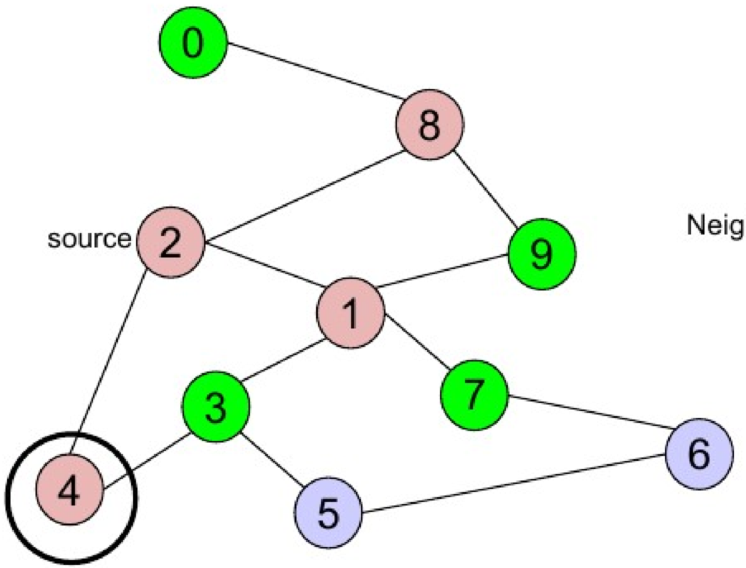
0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

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.

BFS Example



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		

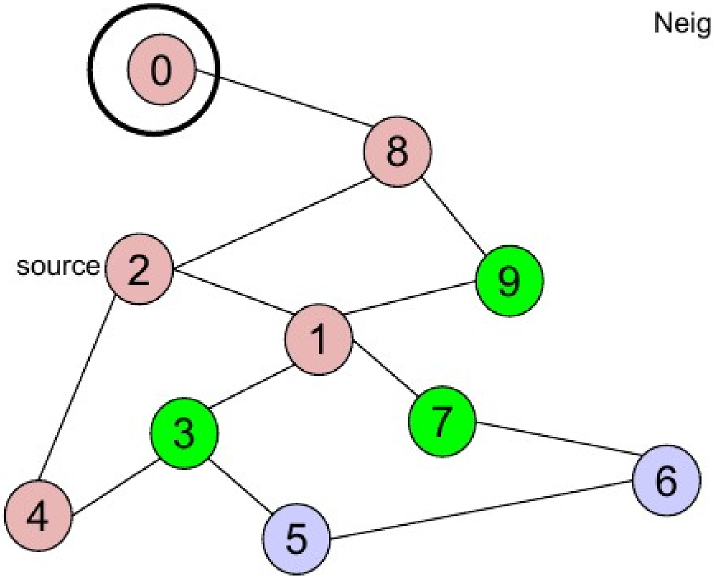
Visited Table (T/F)

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!

BFS Example



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

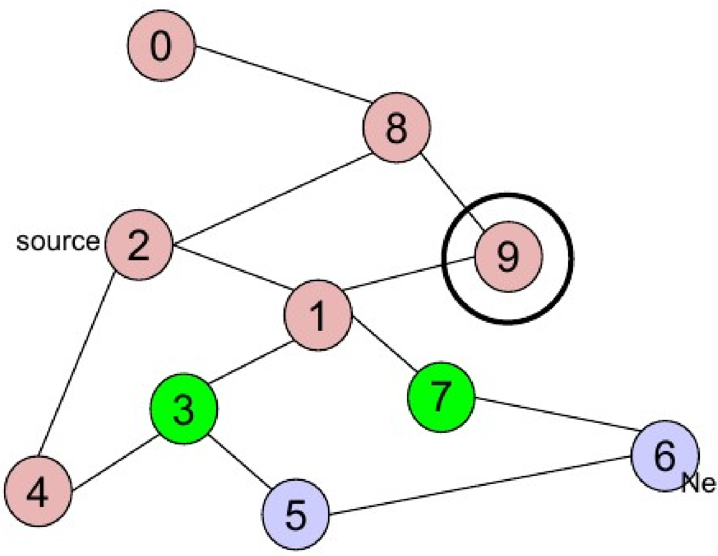
Visited Table (T/F)

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!

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		

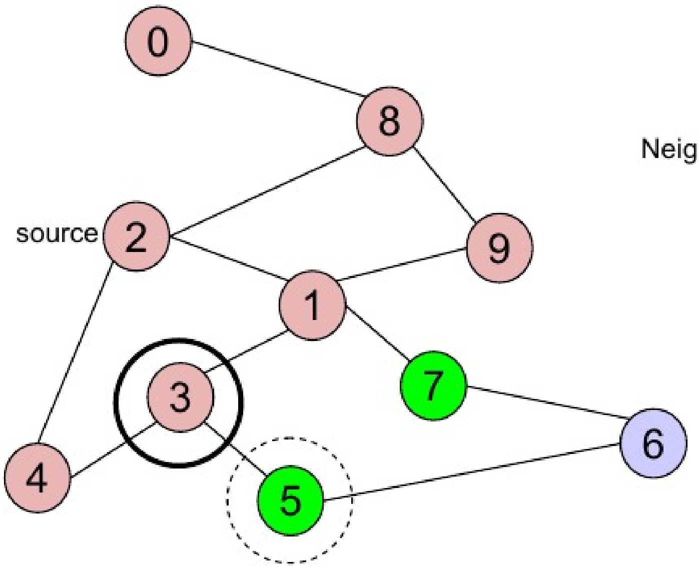
Visited Table (T/F)

0	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!

BFS Example



Q = { 3, 7 } → { 7, 5 }

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

Adjacency List

Neighbors →

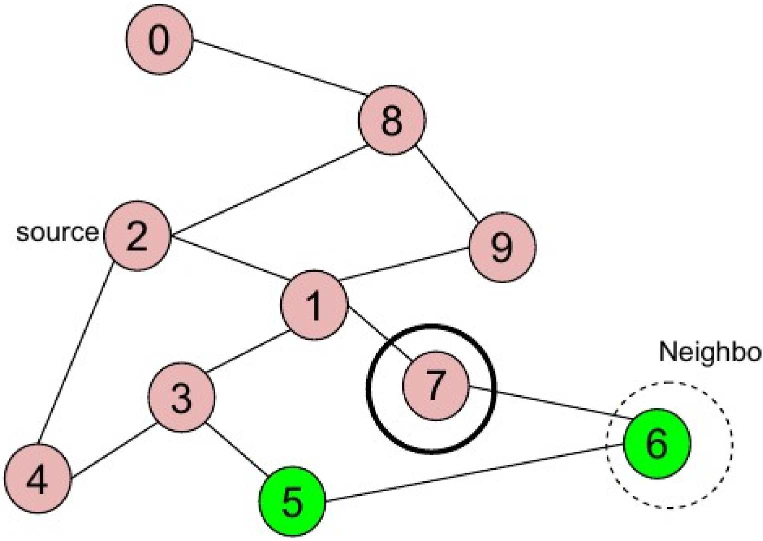
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

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

Mark new visited
Vertex 5

BFS Example



Neighbors → 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)

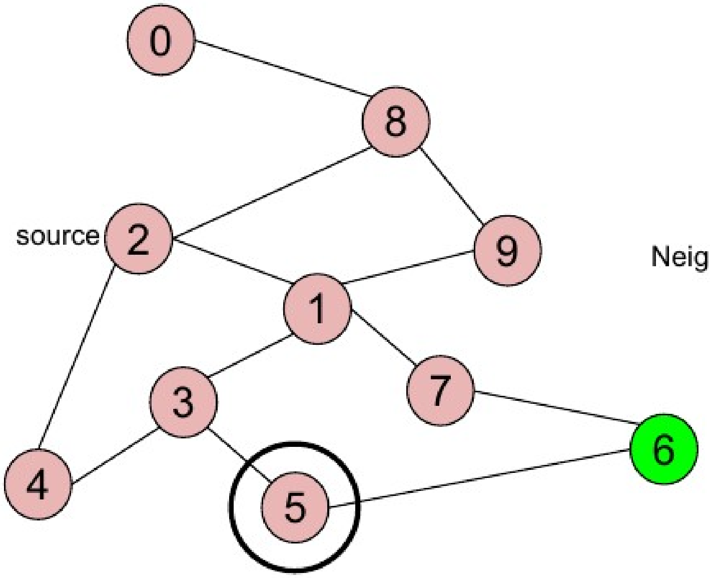
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Mark new visited
Vertex 6

Q = { 7, 5 } → { 5, 6 }

Dequeue 7.
-- place neighbor 6 on the queue

BFS Example



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		

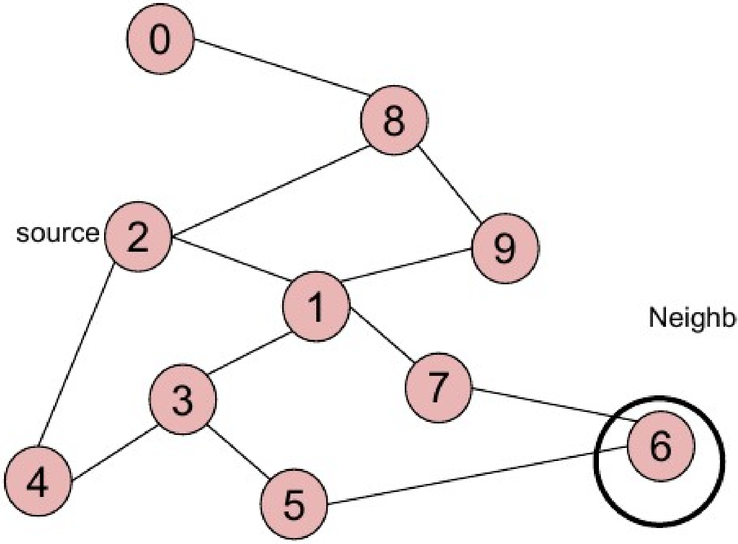
Visited Table (T/F)

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

BFS Example



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

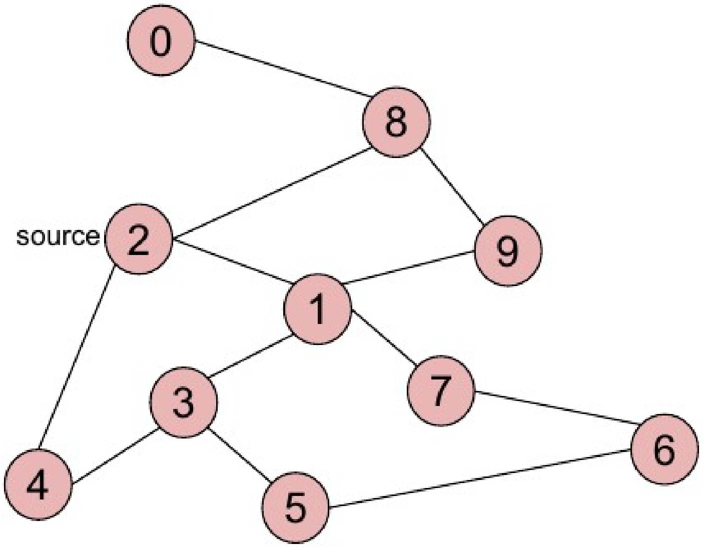
Visited Table (T/F)

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

BFS Example



Q = { } **STOP!!! Q is 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	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

What did we discover?

Look at “visited” tables.

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

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 …

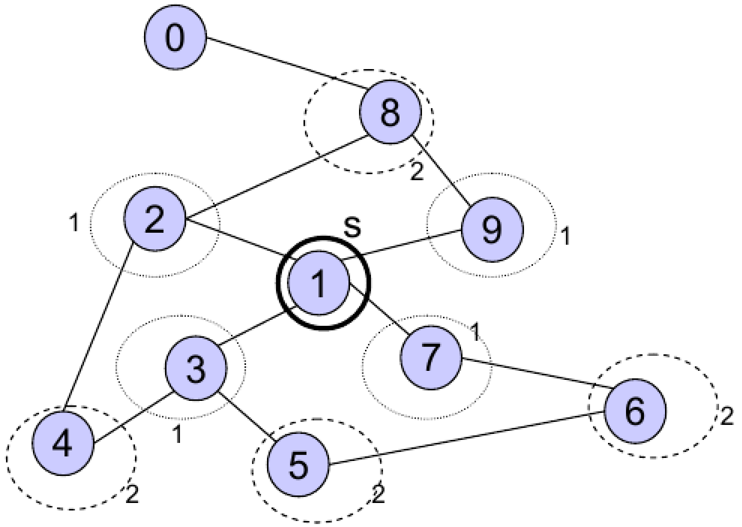
- Is a directed graph strongly connected?
  - Scan array *flag[ ]*.
  - If there exists *flag[u]* = *false* then …

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

Problem 2 - 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

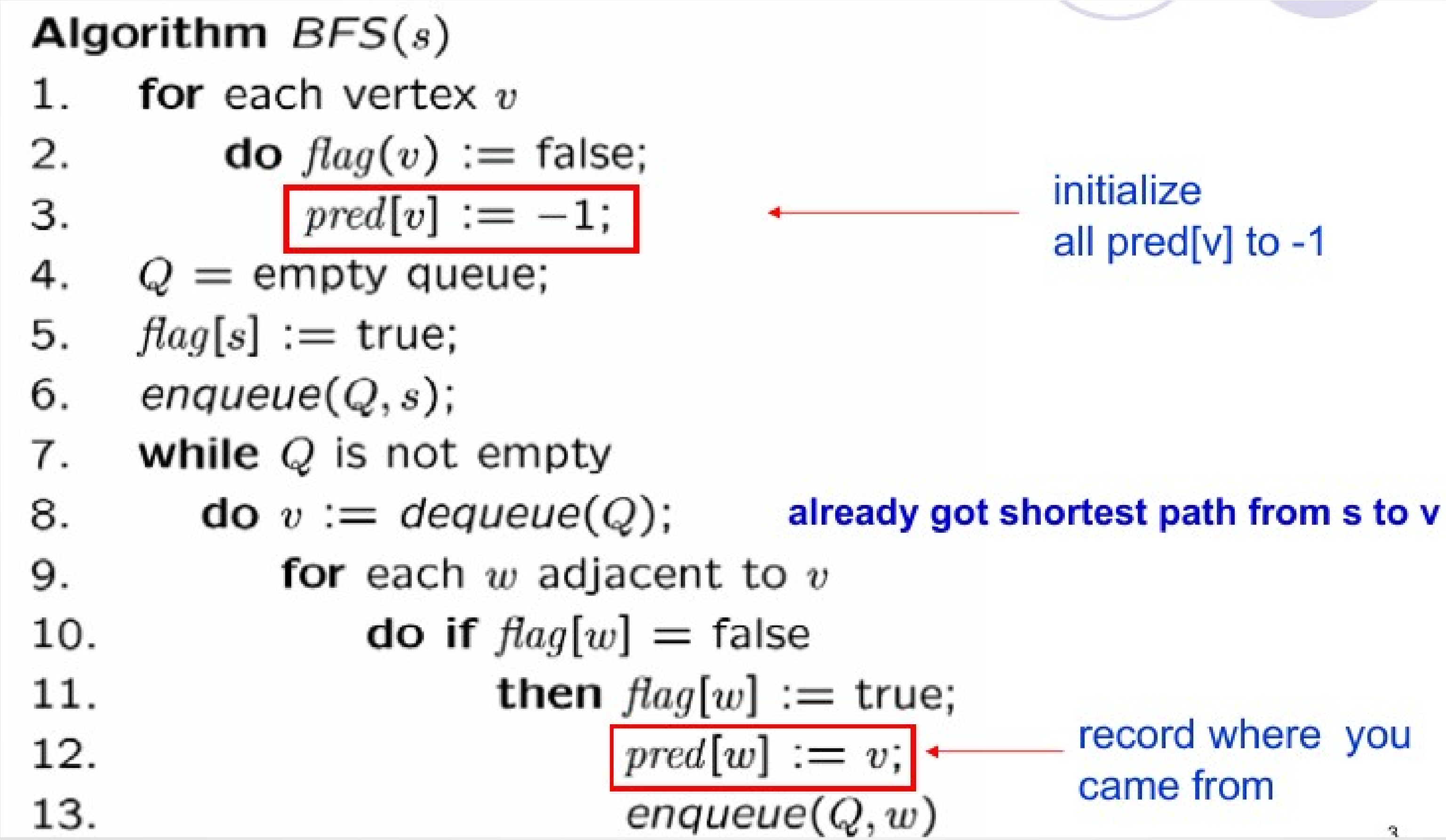
Nodes at distance 2?
8, 6, 5, 4

Nodes at distance 3?
0

Finding Shortest Paths Using BFS

- The BFS code we have seen
  - find outs if there exist 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*

Finding Shortest Paths Using BFS



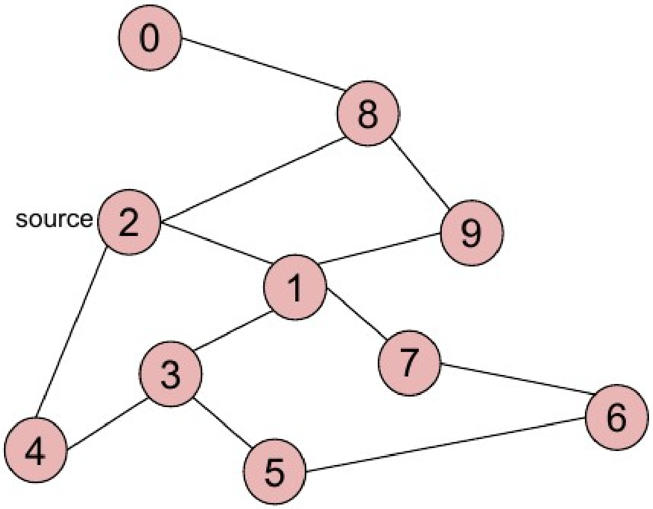
```
Algorithm $BFS(s)$
for each vertex $v$
    do $flag(v)$ := false;
        $pred[v] := -1;$
$Q$ = empty queue;
$flag[s]$ := true;
$enqueue(Q, s)$;
while $Q$ is not empty
    do $v$ := $dequeue(Q)$;
        for each $w$ adjacent to $v$
            do if $flag[w]$ = false
                then $flag[w]$ := true;
                    $pred[w] := v;$
                    $enqueue(Q, w)$
```

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)

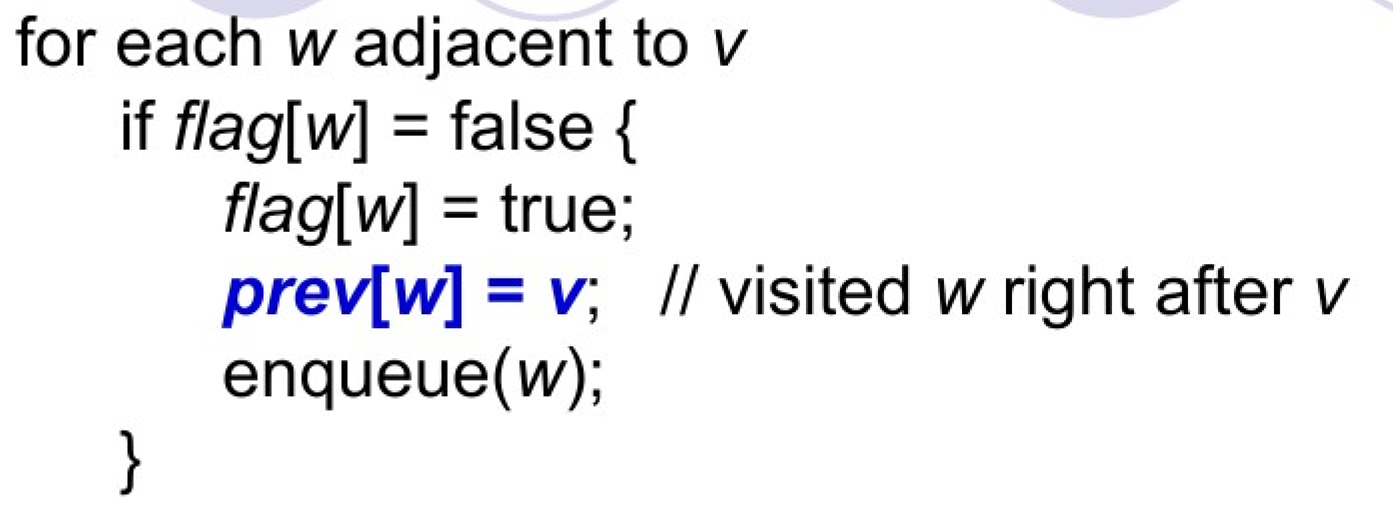
0	T	8
1	T	2
2	T	-
3	T	1
4	T	2
5	T	3
6	T	7
7	T	1
8	T	2
9	T	8

***prev*[ ]**

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

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

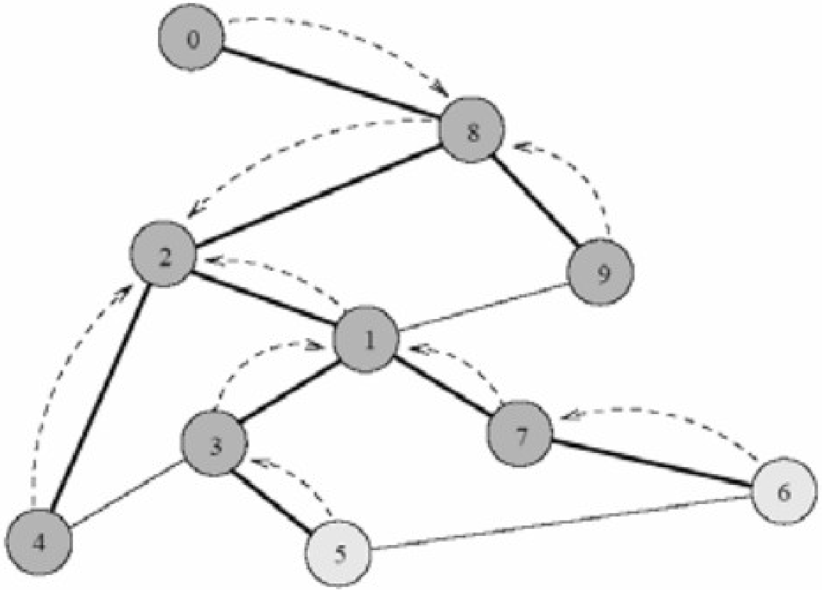
Example



```
for each w adjacent to v
    if flag[w] = false {
        flag[w] = true;
        prev[w] = v;   // visited w right after v
        enqueue(w);
    }
```

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

Example of Path Reporting

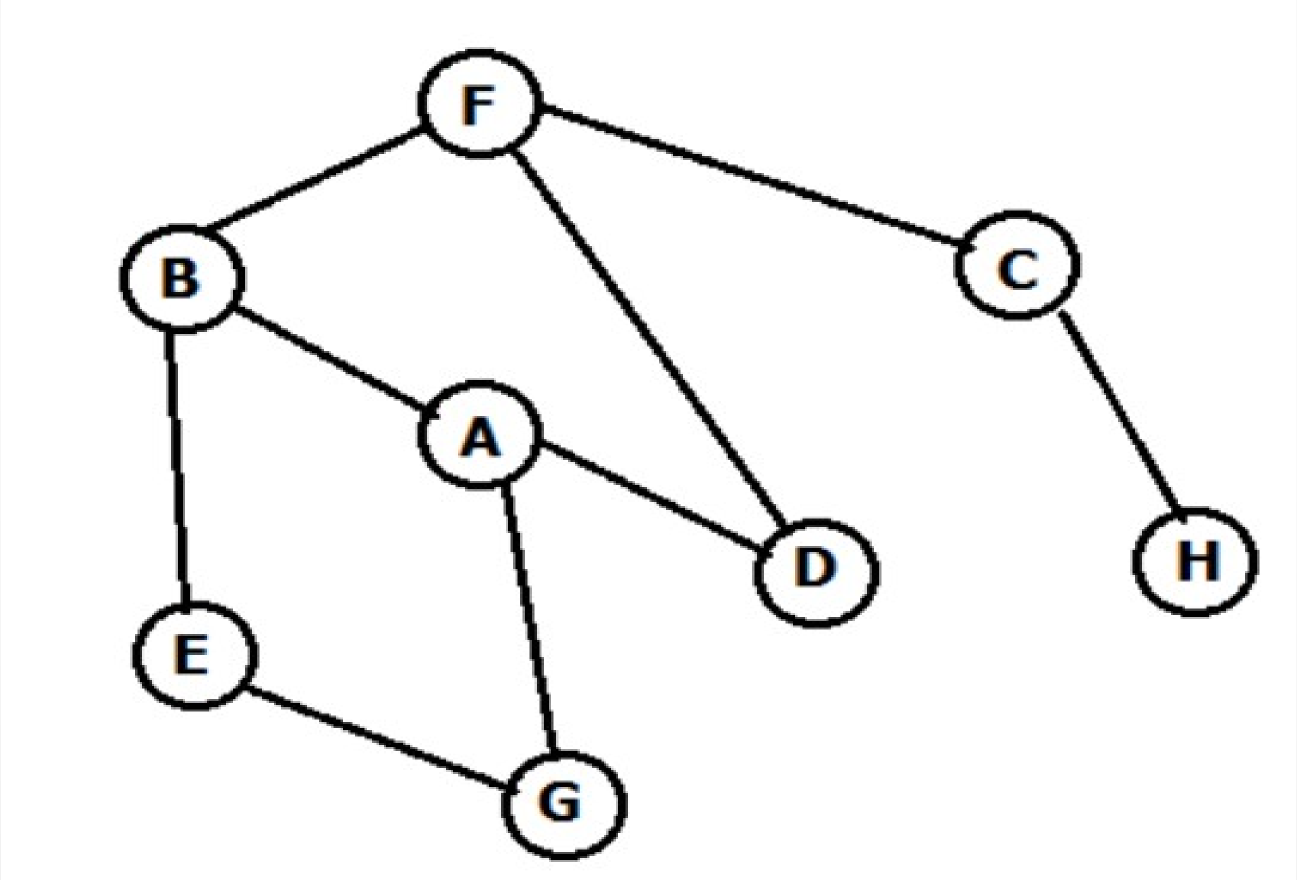


nodes	visited from
0	8
1	2
2	-
3	1
4	2
5	3
6	7
7	1
8	2
9	8

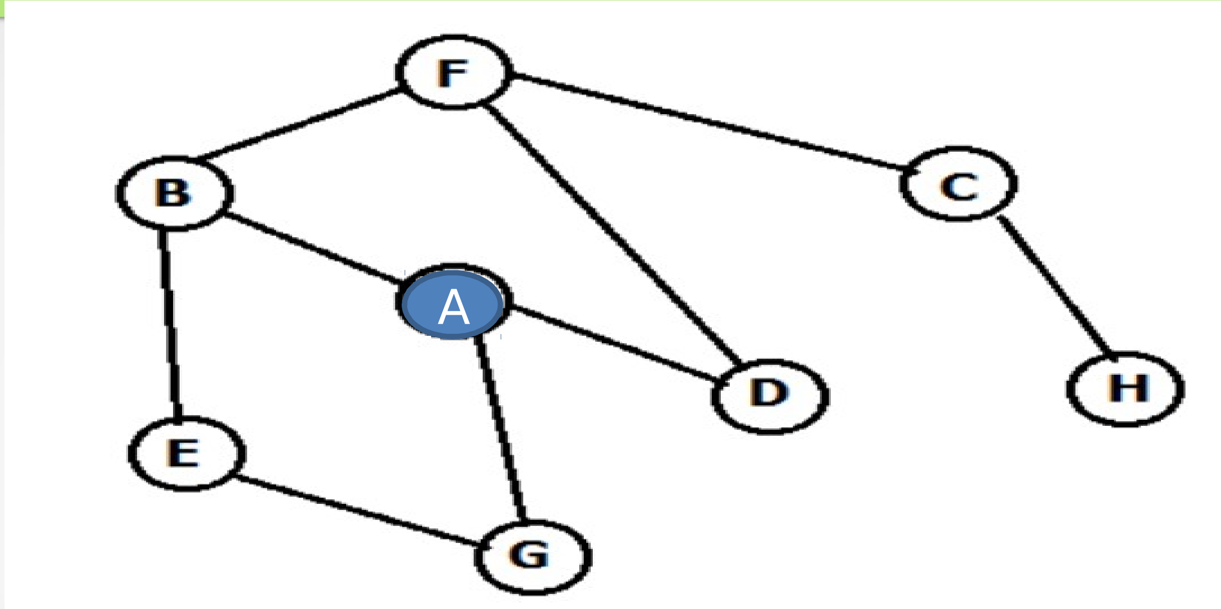
Try some examples; report path from s to v:
Path(2-0) ⇒
Path(2-6) ⇒
Path(2-1) ⇒

PROBLEM 3

- Apply DFS to the following graph



PROBLEM 3- DFS

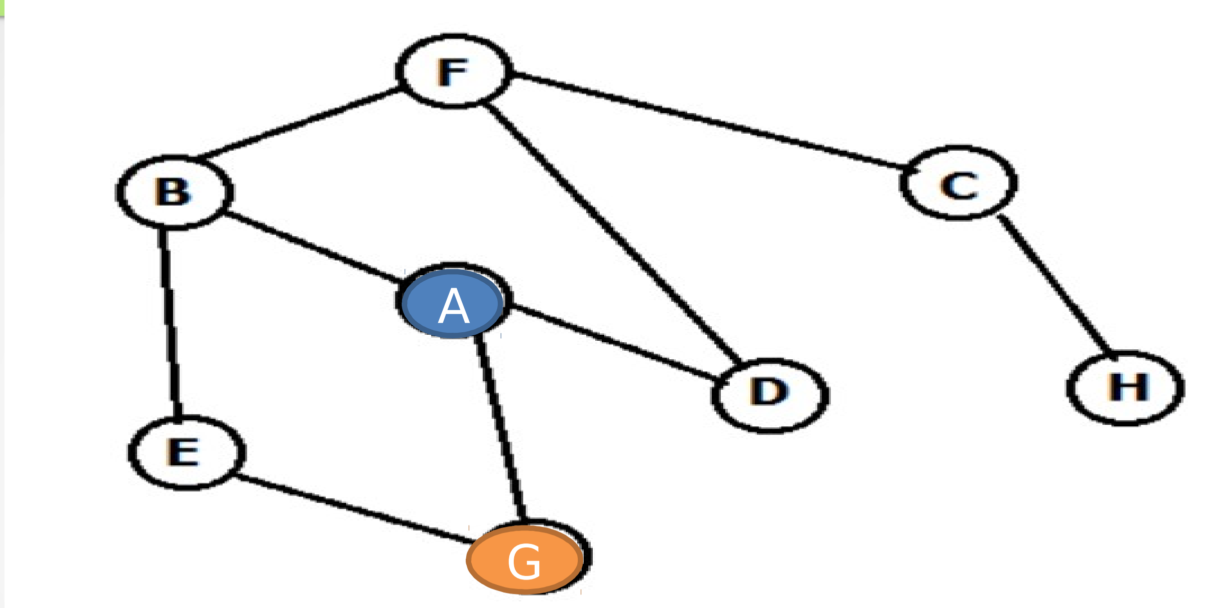


DFS Discovered Order= {A}

Stack={B,D,G}

head

PROBLEM 3- DFS

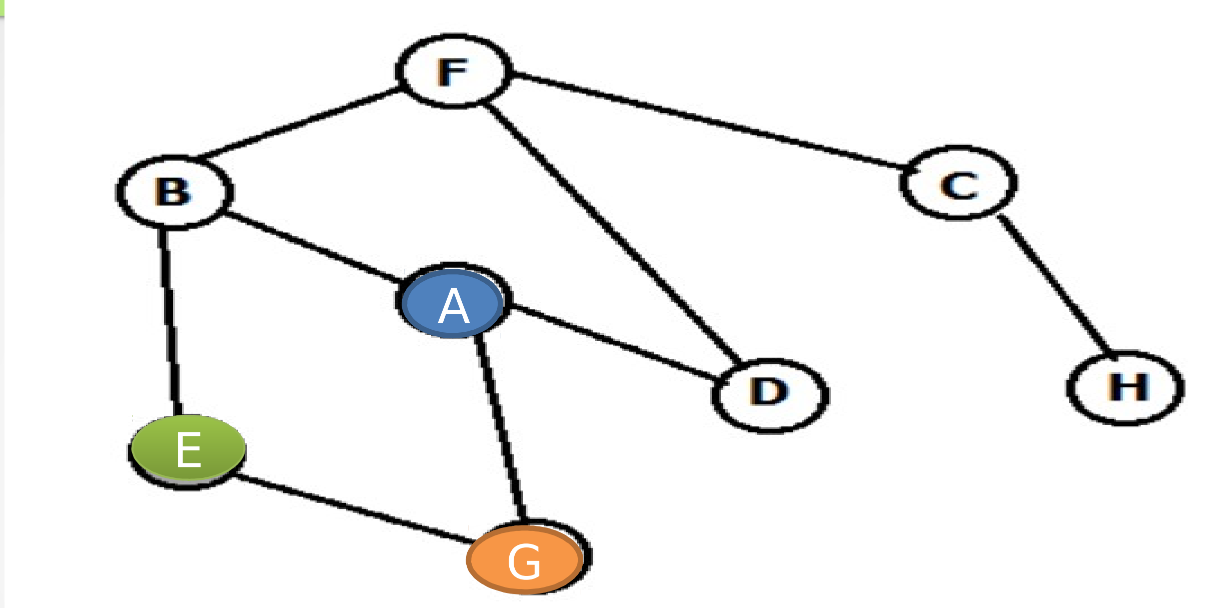


DFS Discovered Order= {A,G}

Stack={B,D,E}

head

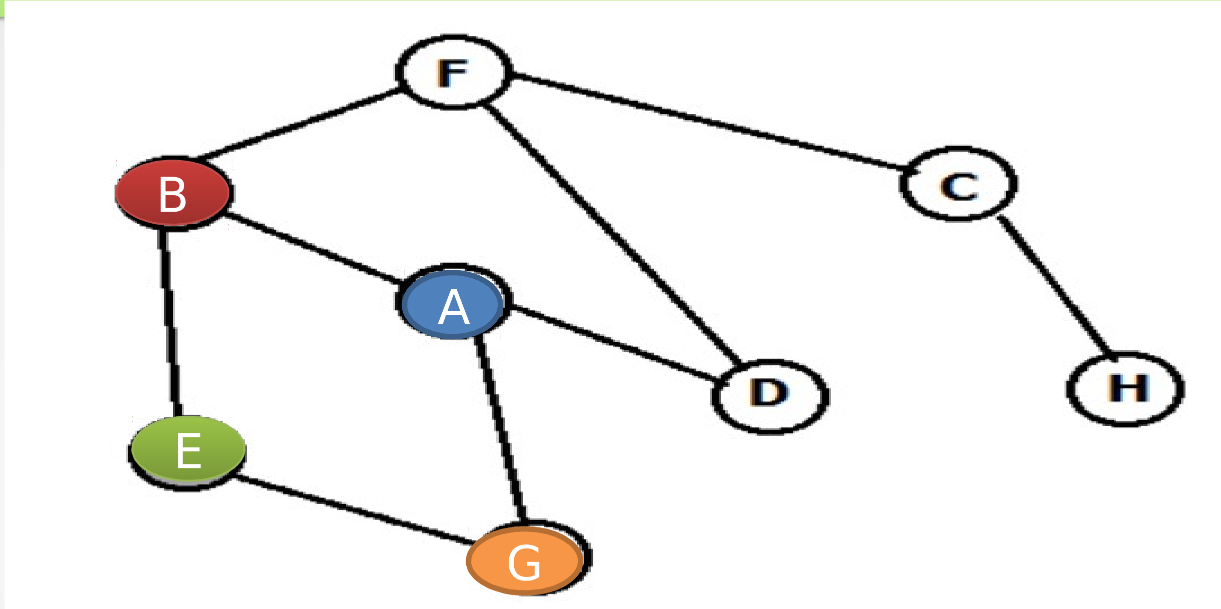
PROBLEM 3- DFS



DFS Discovered Order= {A,G,E}

Stack={B,D,B}

PROBLEM 3- DFS

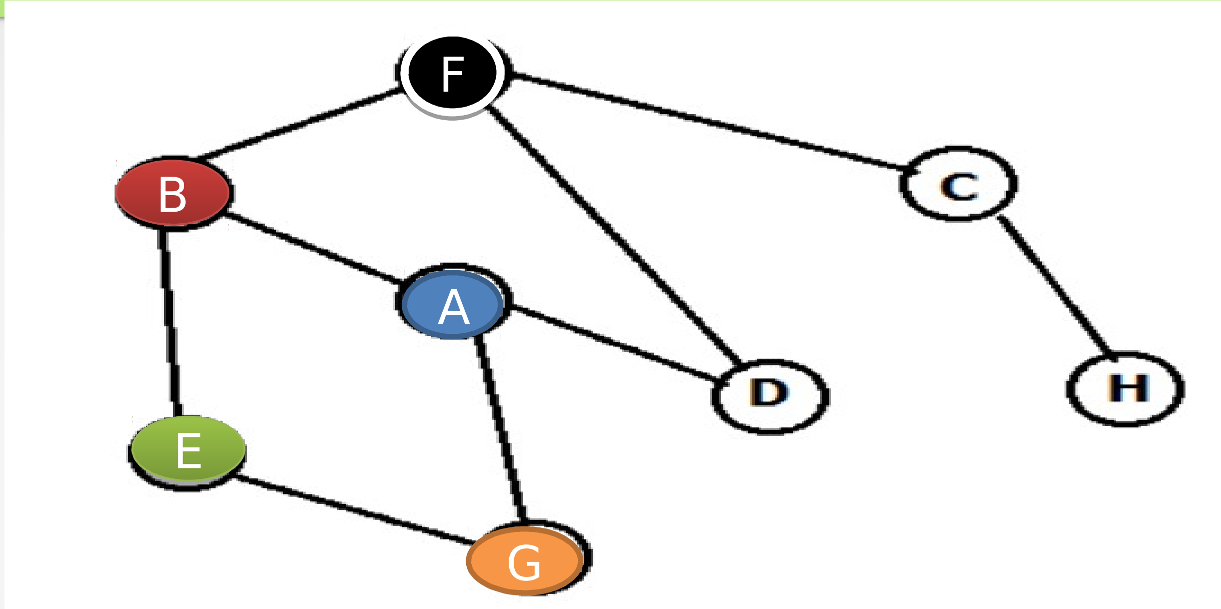


DFS Discovered Order= {A,G,E,B}

Stack={B,D,F}

head

PROBLEM 3- DFS

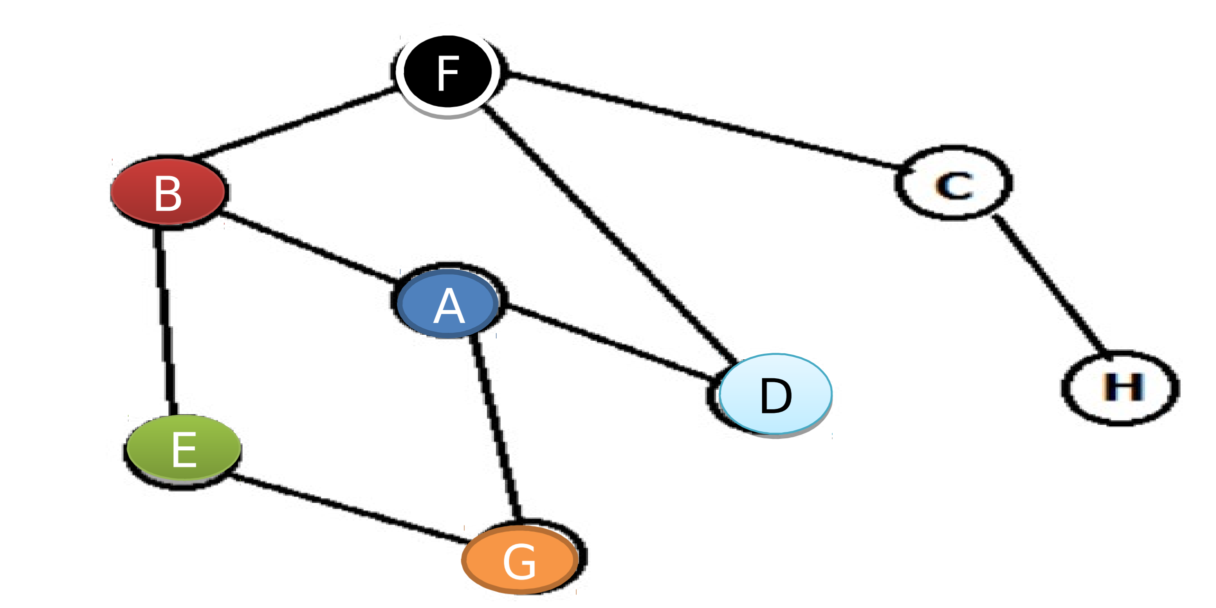


DFS Discovered Order= {A,G,E,B,F}

Stack={B,D,C,D}

head

PROBLEM 3- DFS

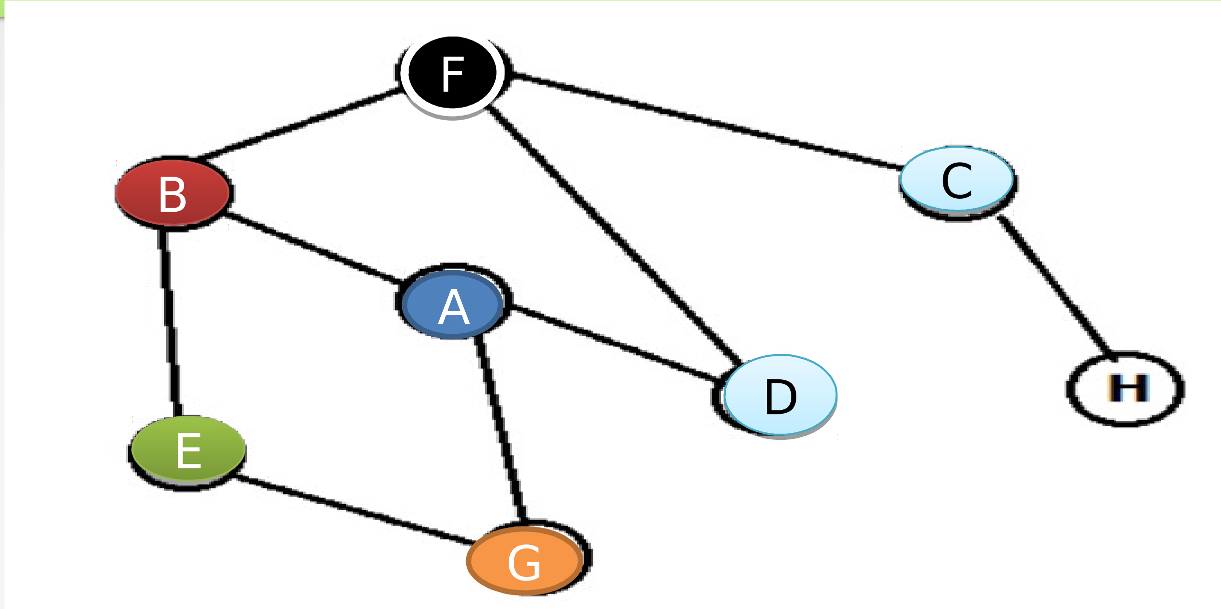


DFS Discovered Order= {A,G,E,B,F, D}

Stack={B,D,C}

head

PROBLEM 3- DFS

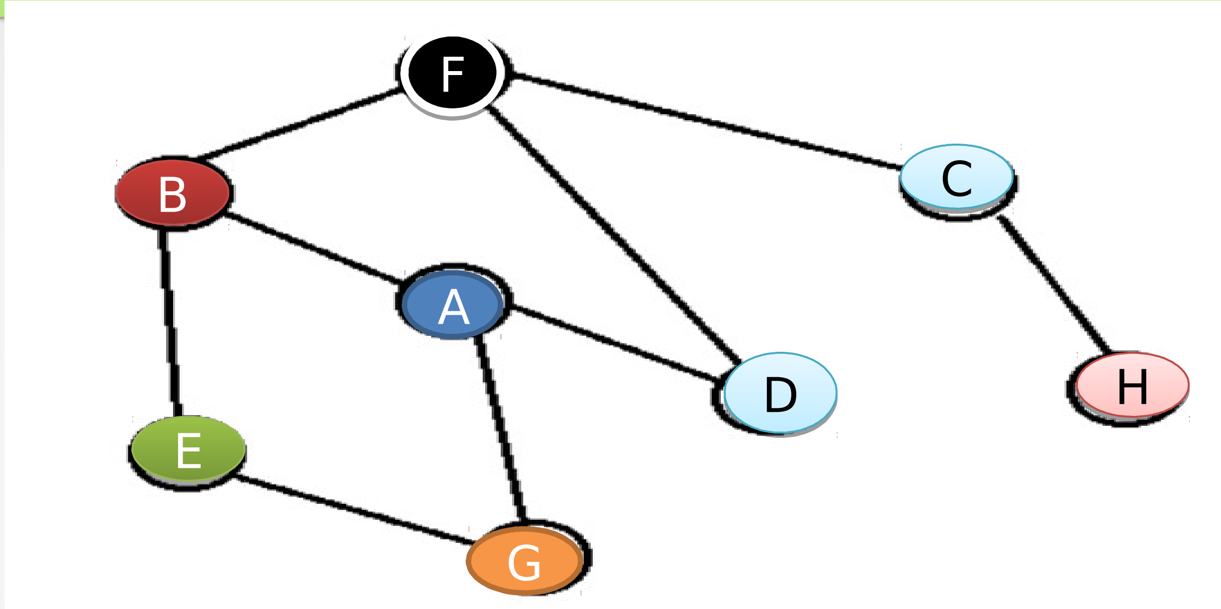


DFS Discovered Order= {A,G,E,B,F, D, C}

Stack={B,D,H}

head

PROBLEM 3- DFS



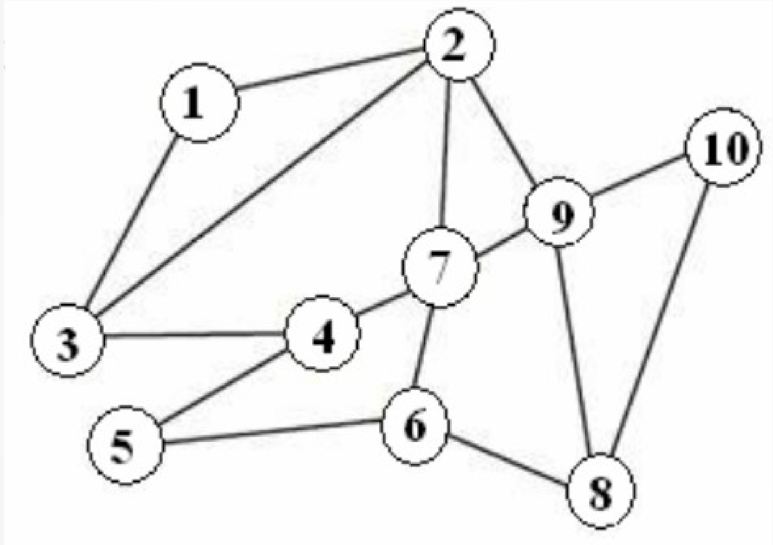
DFS Discovered Order= {A,G,E,B,F, D, C, H}

Stack={}

head

PROBLEM 4

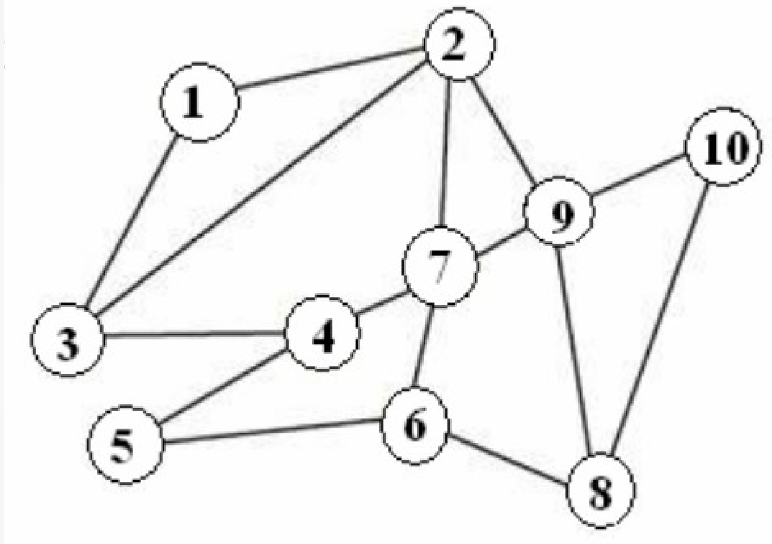
- What is a bipartite graph? Is the following graph is bipartite?



- A bipartite graph is a graph whose vertices can be grouped into two groups such that all the edges are between these two vertex group and there is no edge within a group.

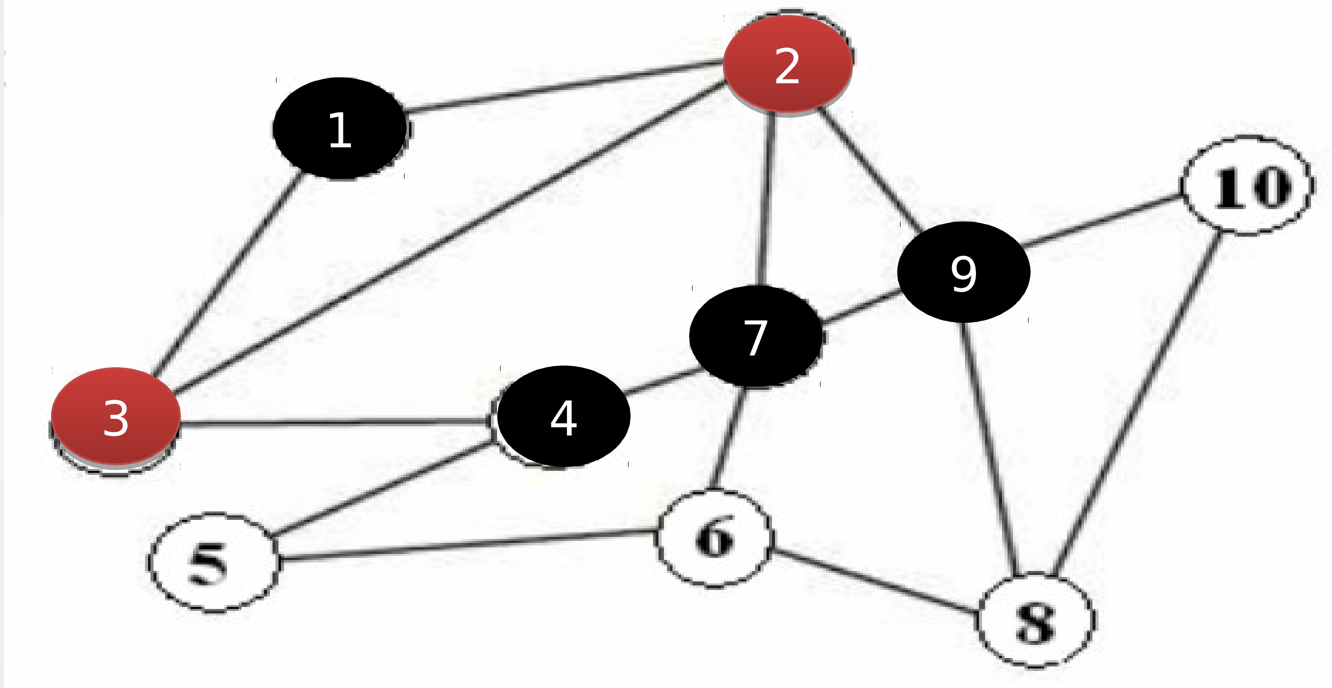
PROBLEM 4

- What is a bipartite graph? Is the following graph is bipartite?



- If a graph can be two colored and there is no odd length cycles then the graph is bipartite.

PROBLEM 4



- 4 and 7 has same color. So this graph is not bipartite.
- Also there many odd length cycles. One of them is (1,2,3)

Greedy Algorithm:

“*An algorithm that builds a solution in small steps, choosing a decision at each step myopically [=locally, not considering what may happen ahead] to optimize some underlying criterion.*”

- Produces optimum solution for some problems.
  - Minimum spanning tree
  - Single-source shortest paths
  - Huffman trees
- Produces good aproximate solutions for some other problems.
  - NP-Complete problems such as graph coloring

Problem 5 - Coffee Shop

- You own a coffee shop that has n customers.
- It takes t_i minutes to prepare coffee for the i^{th} customer.
- i^{th}customer's value for you (i.e. how frequent s/he comes to your shop) is v_i
- If you start preparing coffee for the i^{th} customer at time s_i you finish at $f_i = s_i + t_i$
- All customers arrive at the same time.
- You can prepare one coffee at a time.
- There is no gap after you finish one coffee and start another.

Coffee Shop

- You are asked to design an algorithm.

Input: n, t_i , v_i

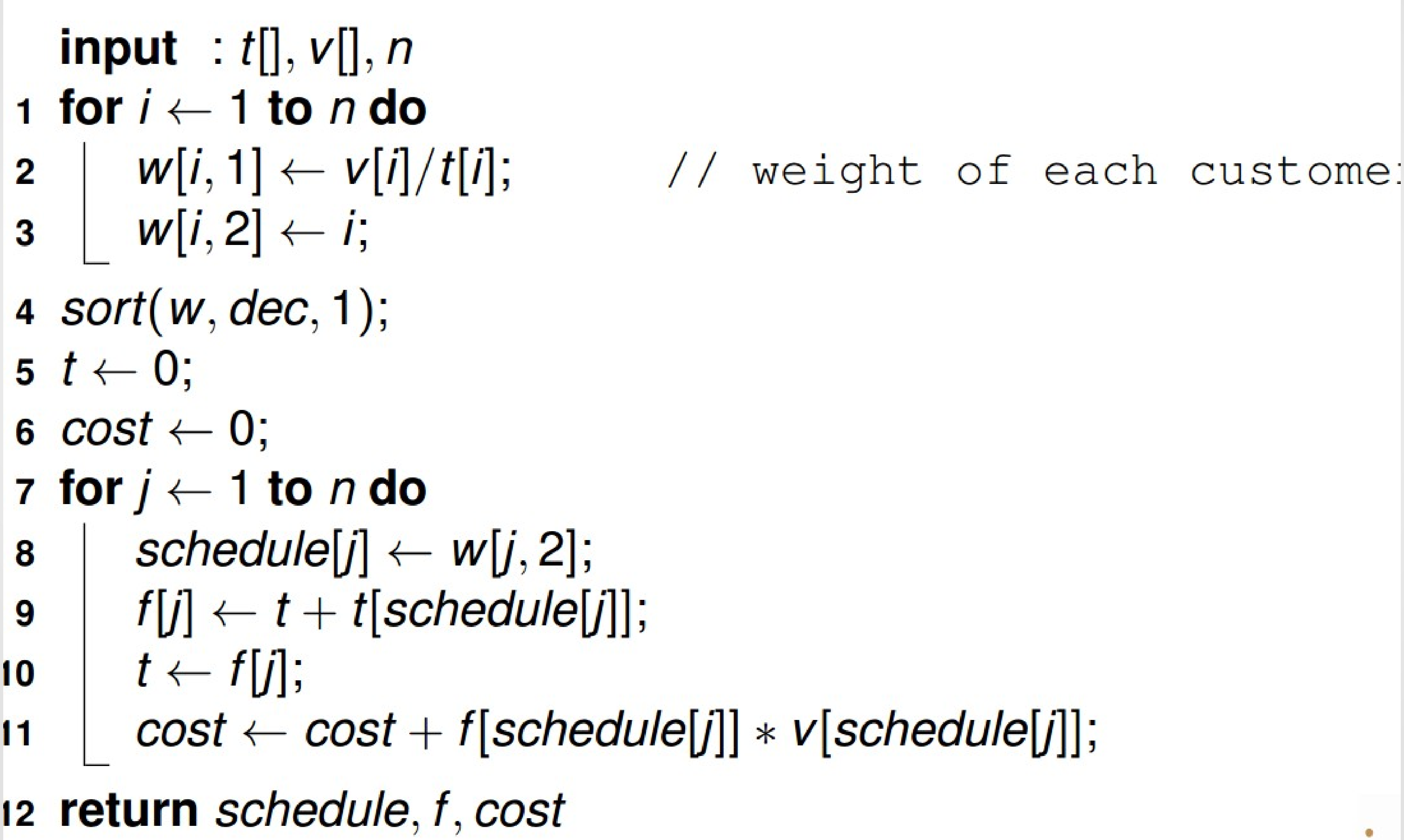
Output: A schedule (i.e. ordering of customer requests)

Aim: Minimize wait time especially for valued customers

$$Minimize : \sum_{i=1}^{n} f_i * v_i$$

- **What is the time complexity of your algorithm?**
- **Run your algorithm for a sample input.**

Algorithm



```
input : $t[], v[], n$
for $i \leftarrow 1$ to $n$ do
    $w[i,1] \leftarrow v[i]/t[i]$;        // weight of each custome
    $w[i,2] \leftarrow i$;
$sort(w, dec, 1)$;
$t \leftarrow 0$;
$cost \leftarrow 0$;
for $j \leftarrow 1$ to $n$ do
    $schedule[j] \leftarrow w[j,2]$;
    $f[j] \leftarrow t + t[schedule[j]]$;
    $t \leftarrow f[j]$;
    $cost \leftarrow cost + f[schedule[j]] * v[schedule[j]]$;
return $schedule, f, cost$
```

Complexity of the algorithm

- Both for loops take O(n) time.
- Complexity of the algorithm depends on *sort method*.
- Typically O(nlogn)

A sample input

Input:

- t1= 2, t2 = 3, t3 = 1
- V1 = 10, v2 = 2, v3 = 1

Output:

- Weights = 5, 0.67, 1
- Schedule: 1, 3, 2
- Finish times: 2, 3, 6
- Cost: 2*10 + 3*1 + 6*2 = 35