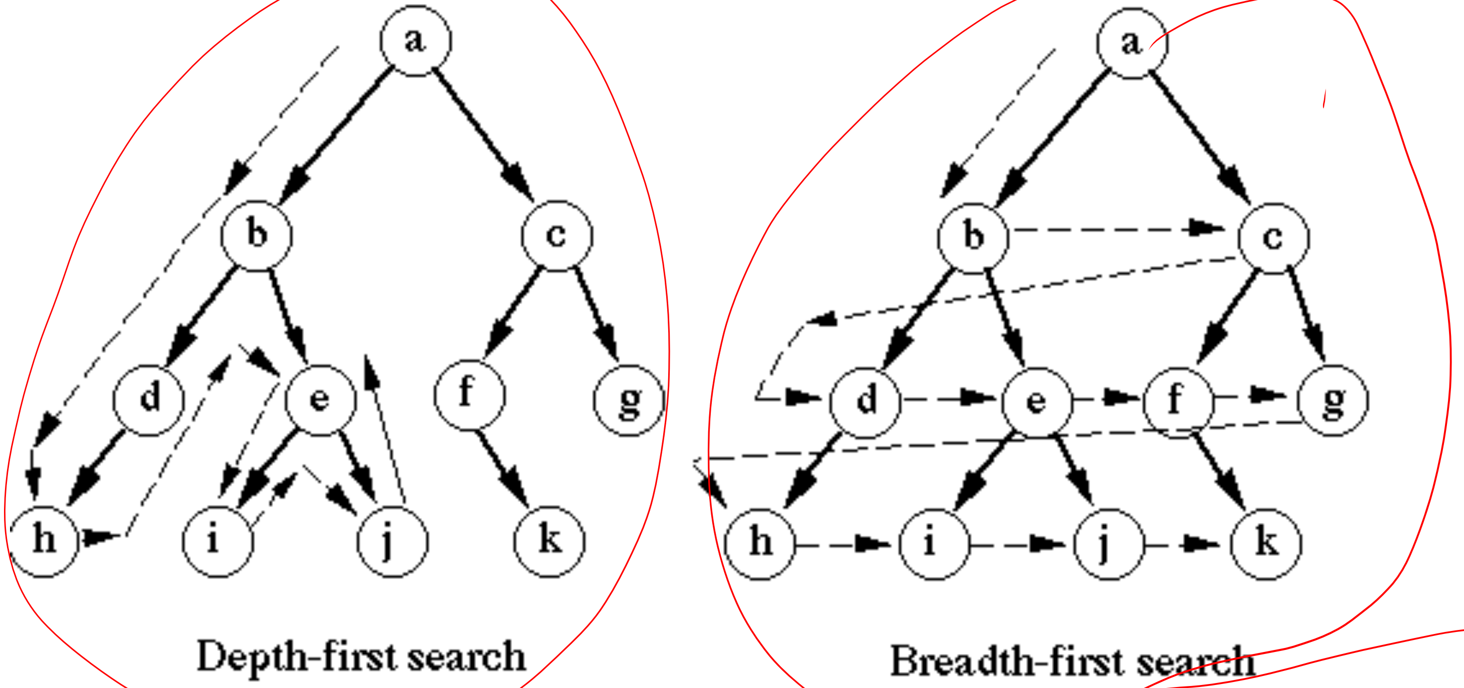
TREE (part 2)

Dr. Seung Chul Han
Dept. Computer Engineering
Myongji University

Tree Search

- Depth-First Search / Breadth-First-Search



Depth-first search

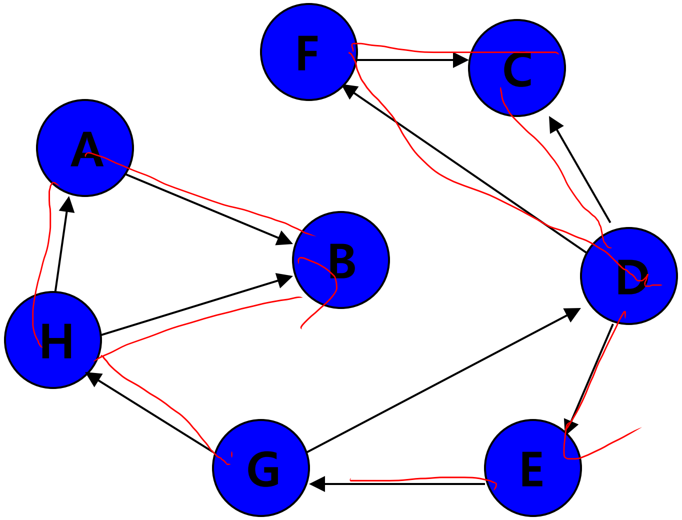
Breadth-first search

사이클 확인, stack을 사용

최단거리 확인, queue 사용

Example: DFS

사이클확인.

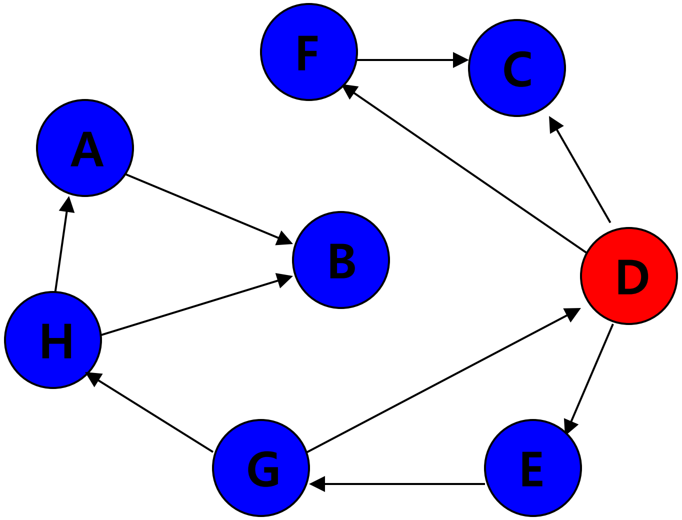


Visited Array

A	✓
B	✓
C	✓
D	✓
E	✓
F	✓
G	✓
H	✓

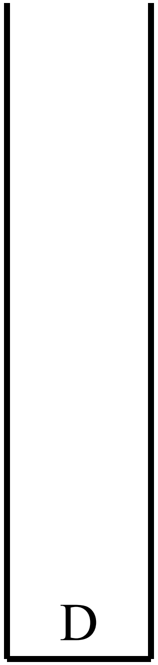
Task: Conduct a depth-first search of the graph starting with node D

Walk-Through



Visited Array

A	
B	
C	
D	√
E	
F	
G	
H	

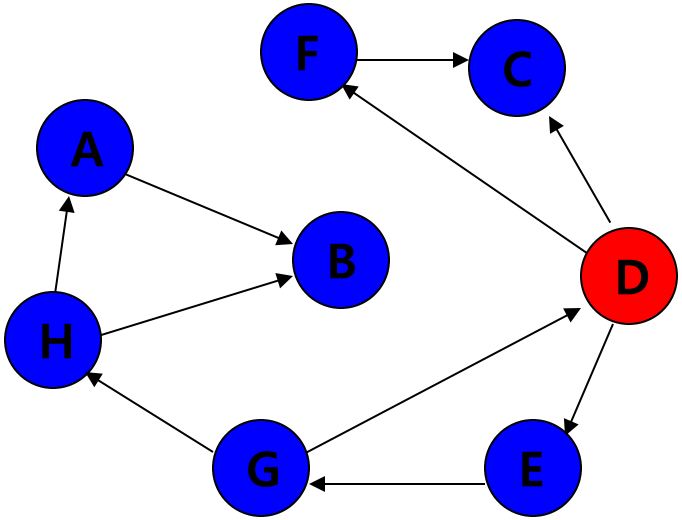

D

The order nodes are visited:

D

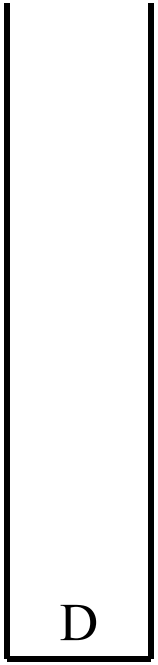
Visit D

Walk-Through



Visited Array

A	
B	
C	
D	√
E	
F	
G	
H	

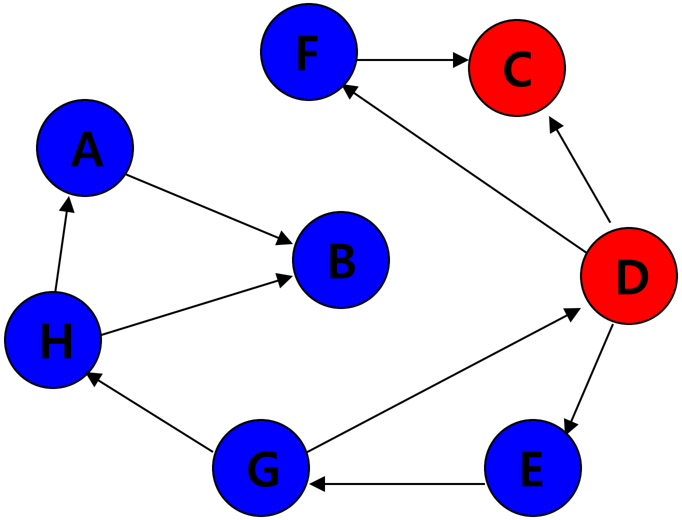


The order nodes are visited:

D

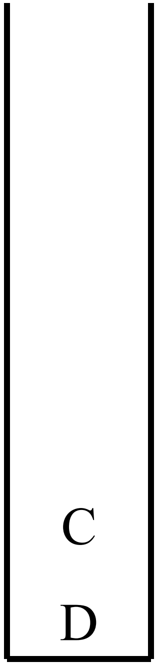
Consider nodes adjacent to D, decide to visit C first (Rule: visit adjacent nodes in alphabetical order)

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	
F	
G	
H	

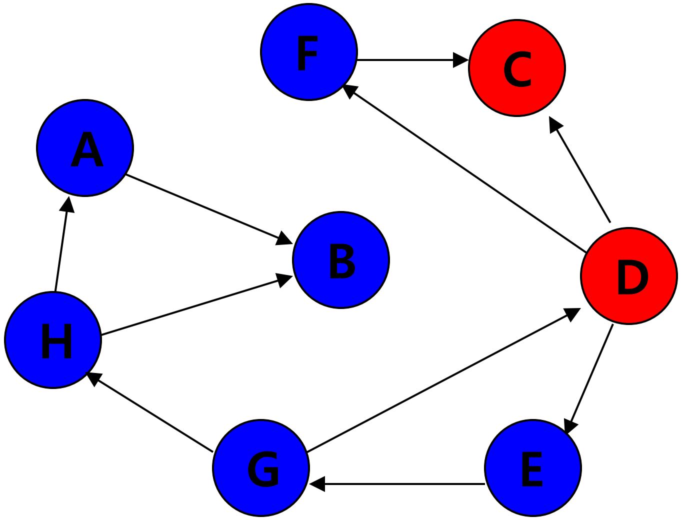


Visit C

The order nodes are visited:

D, C

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	
F	
G	
H	

C

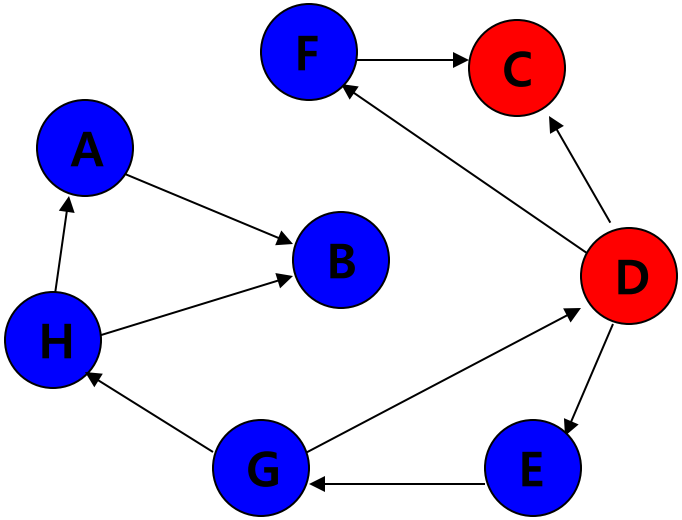
D

The order nodes are visited:

D, C

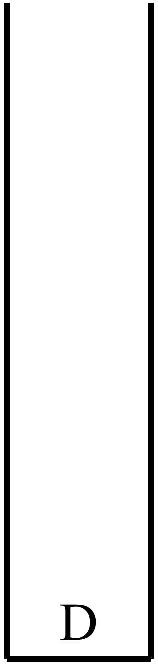
No nodes adjacent to C; cannot continue ➔ *backtrack*, i.e., pop stack and restore previous state

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	
F	
G	
H	

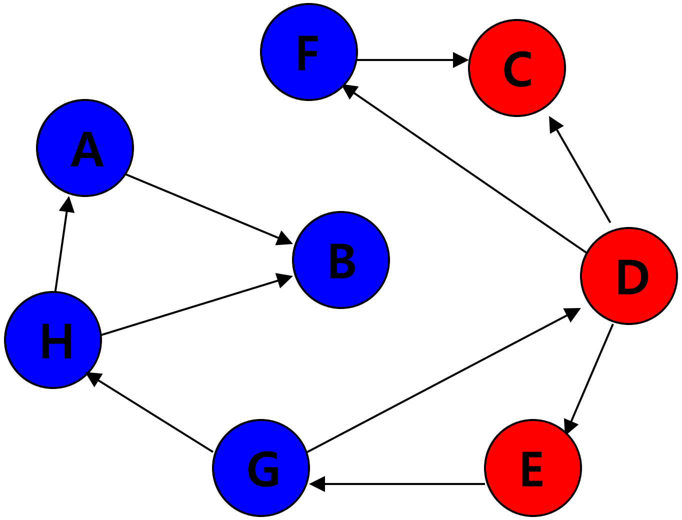


The order nodes are visited:

D, C

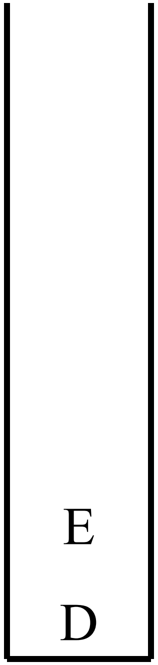
Back to D – C has been visited, decide to visit E next

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	
H	

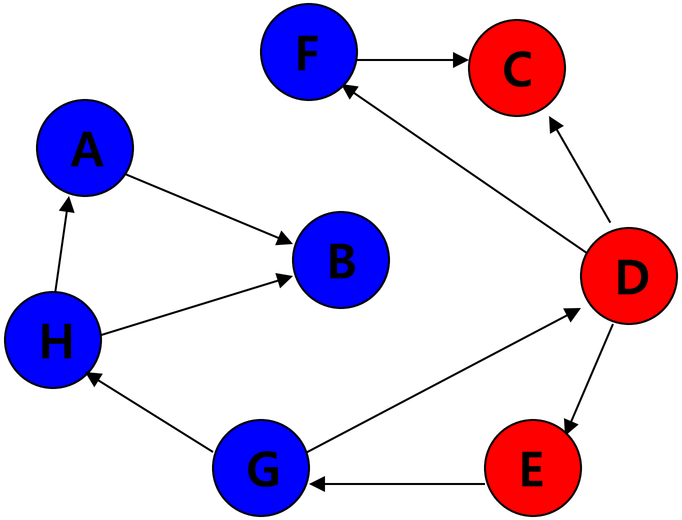


The order nodes are visited:

D, C, E

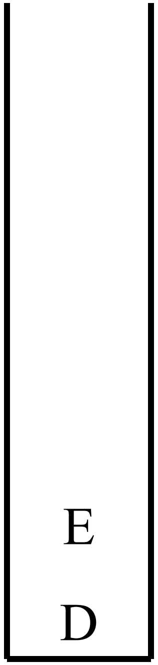
Back to D – C has been visited, decide to visit E next

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	
H	


E

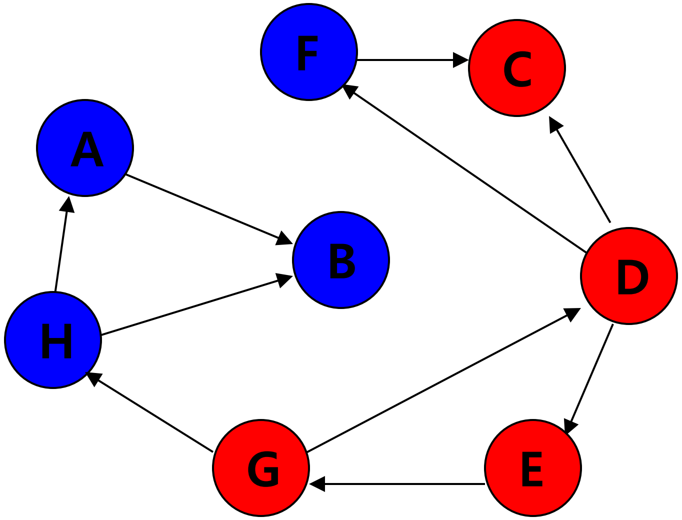
D

The order nodes are visited:

D, C, E

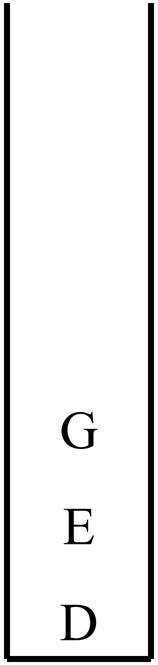
Only G is adjacent to E

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	√
H	



G

E

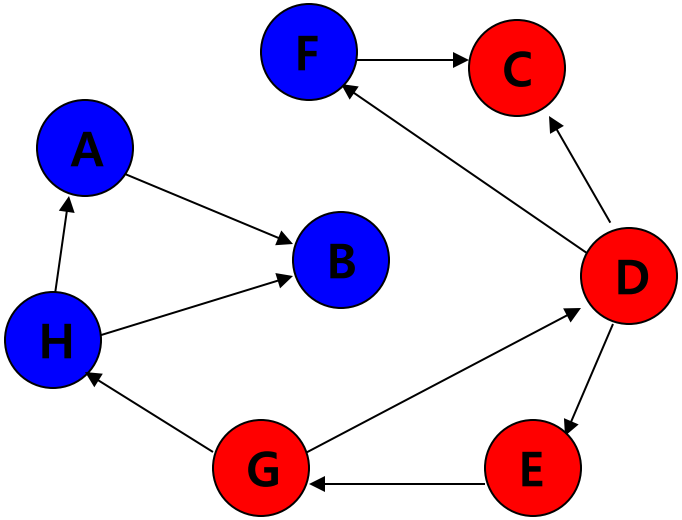
D

Visit G

The order nodes are visited:

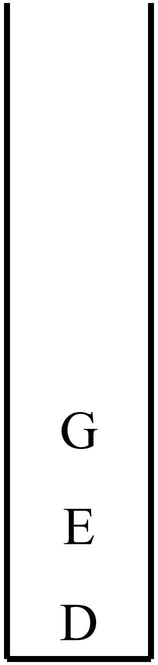
D, C, E, G

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	√
H	


G

E

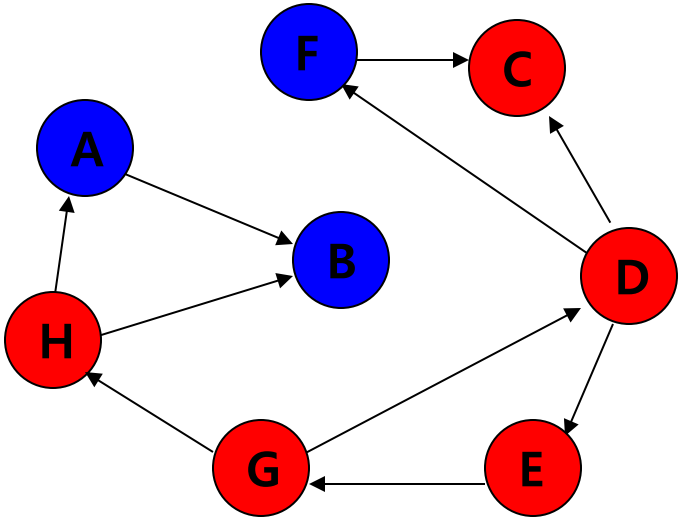
D

The order nodes are visited:

D, C, E, G

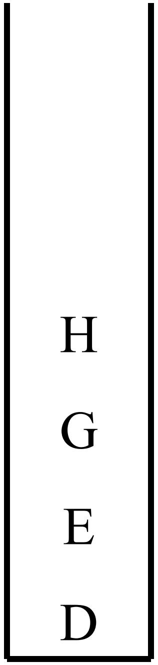
Nodes D and H are adjacent to G. D has already been visited. Decide to visit H.

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	√
H	√



H

G

E

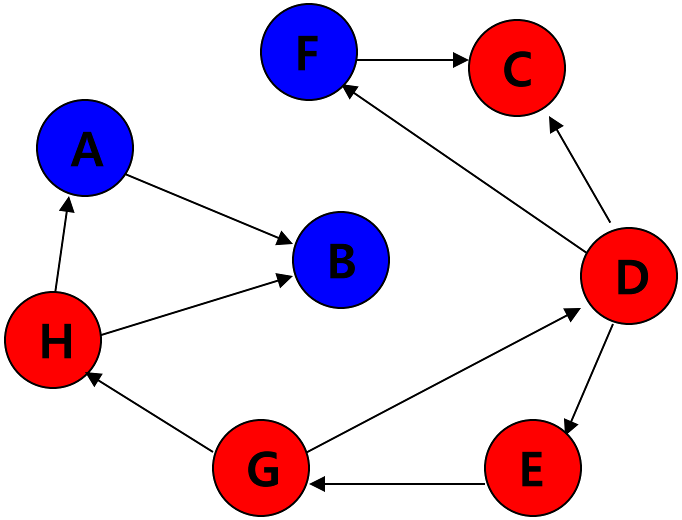
D

Visit H

The order nodes are visited:

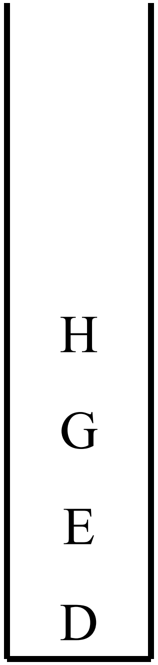
D, C, E, G, H

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	√
H	√



H

G

E

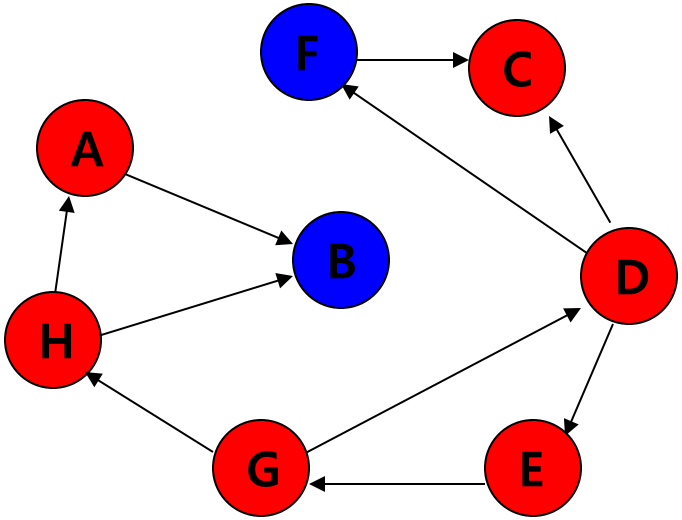
D

The order nodes are visited:

D, C, E, G, H

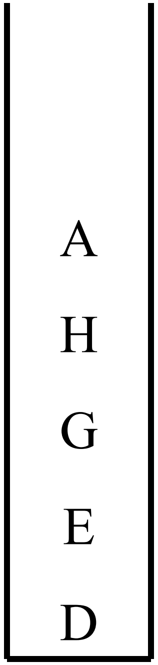
Nodes A and B are adjacent to F.
Decide to visit A next.

Walk-Through



Visited Array

A	√
B	
C	√
D	√
E	√
F	
G	√
H	√


A

H

G

E

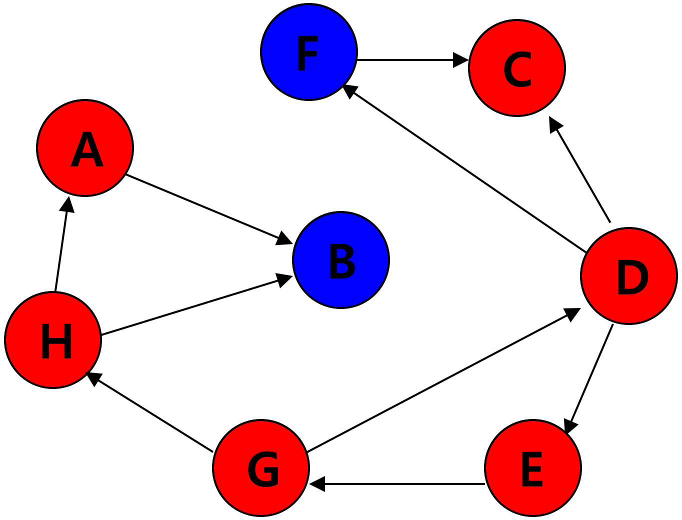
D

Visit A

The order nodes are visited:

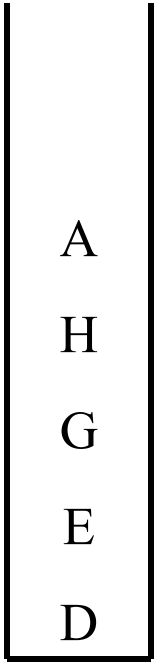
D, C, E, G, H, A

Walk-Through



Visited Array

A	√
B	
C	√
D	√
E	√
F	
G	√
H	√



A

H

G

E

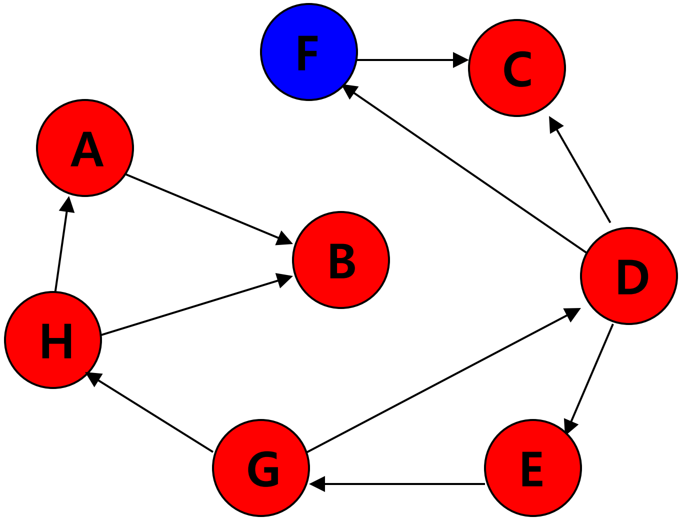
D

The order nodes are visited:

D, C, E, G, H, A

Only Node B is adjacent to A.
Decide to visit B next.

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

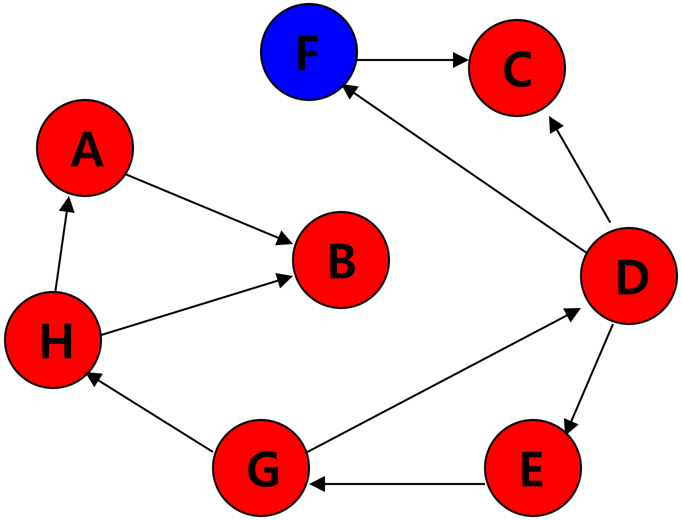
B
A
H
G
E
D

Visit B

The order nodes are visited:

D, C, E, G, H, A, B

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

A

H

G

E

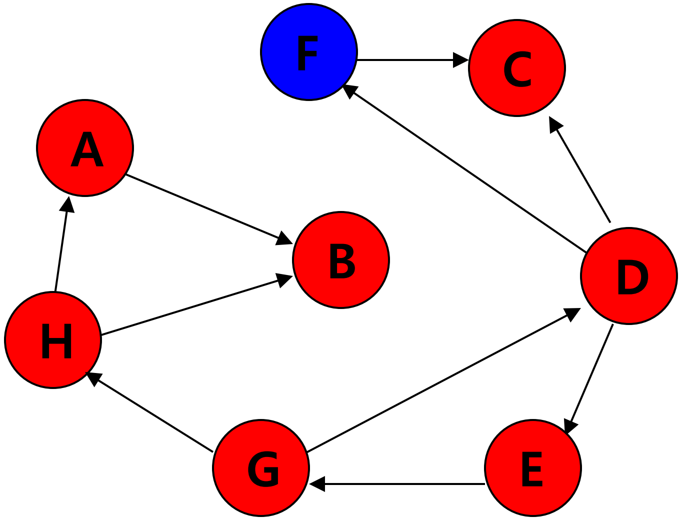
D

The order nodes are visited:

D, C, E, G, H, A, B

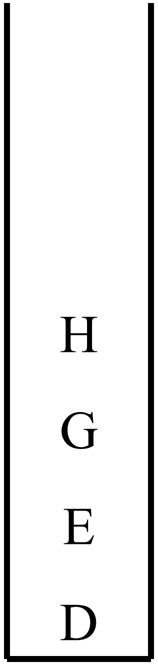
No unvisited nodes adjacent to B. Backtrack (pop the stack).

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√



H

G

E

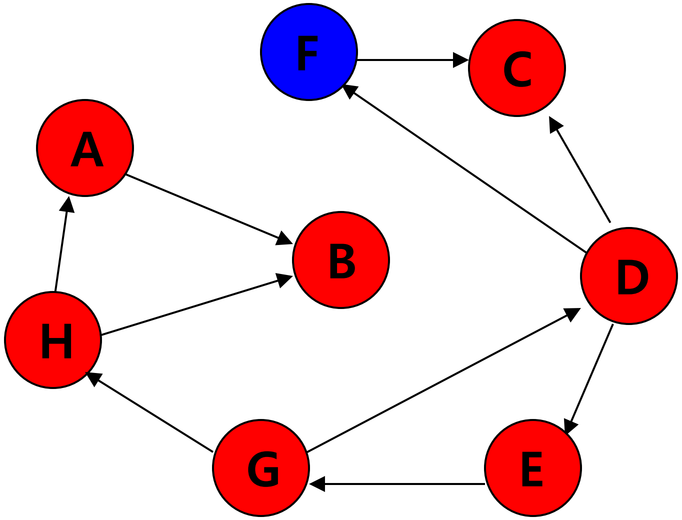
D

The order nodes are visited:

D, C, E, G, H, A, B

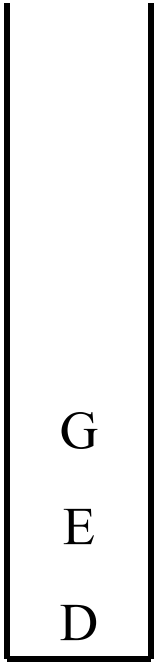
No unvisited nodes adjacent to A. Backtrack (pop the stack).

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√



G

E

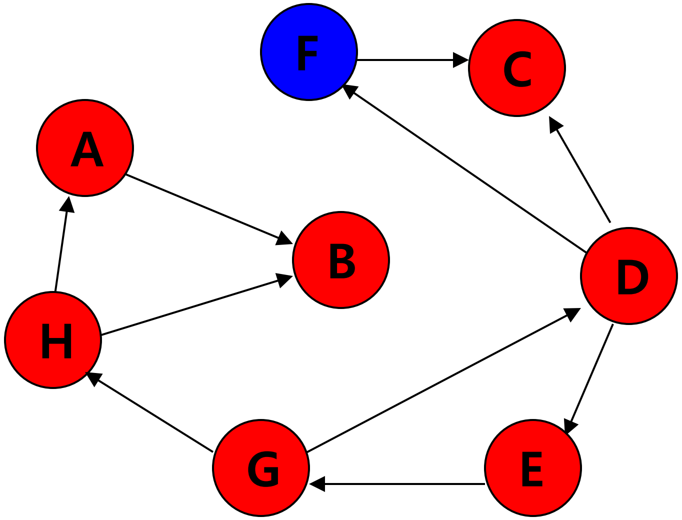
D

The order nodes are visited:

D, C, E, G, H, A, B

No unvisited nodes adjacent to H. Backtrack (pop the stack).

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

E

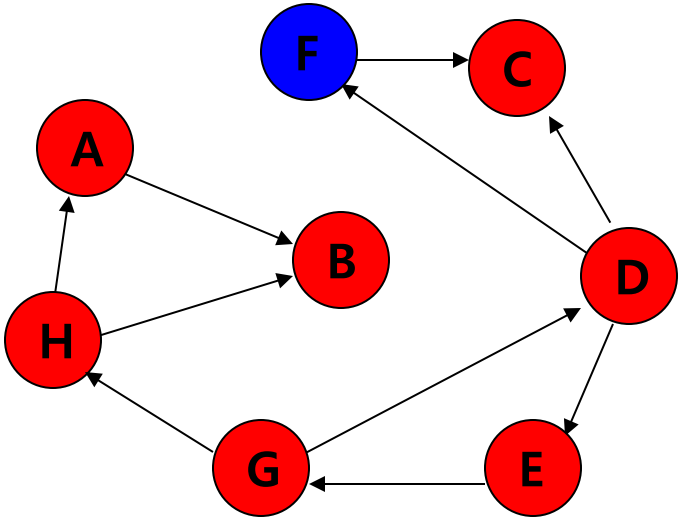
D

The order nodes are visited:

D, C, E, G, H, A, B

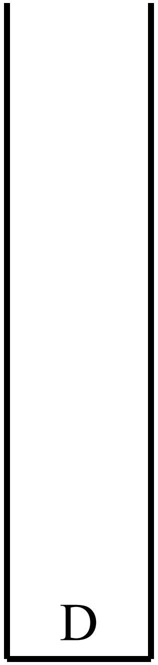
No unvisited nodes adjacent to G. Backtrack (pop the stack).

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

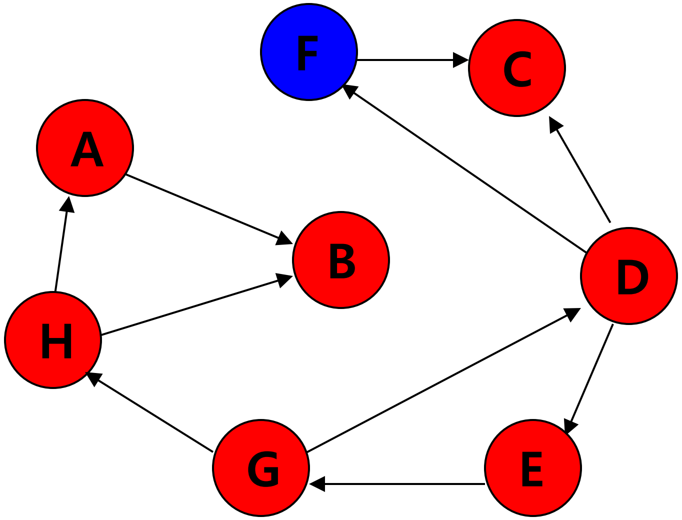


The order nodes are visited:

D, C, E, G, H, A, B

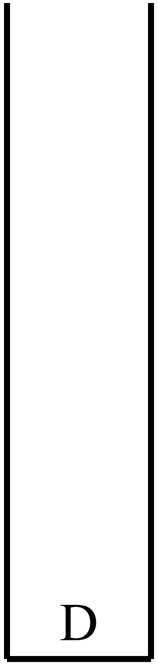
No unvisited nodes adjacent to E. Backtrack (pop the stack).

Walk-Through



Visited Array

A	✓
B	✓
C	✓
D	✓
E	✓
F	
G	✓
H	✓

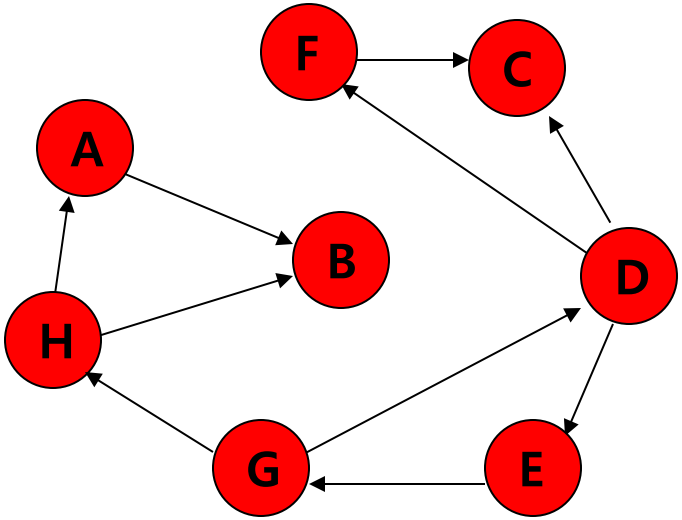


The order nodes are visited:

D, C, E, G, H, A, B

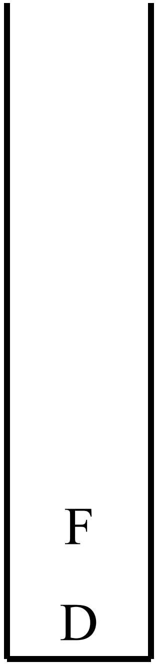
F is unvisited and is adjacent to D. Decide to visit F next.

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√



F

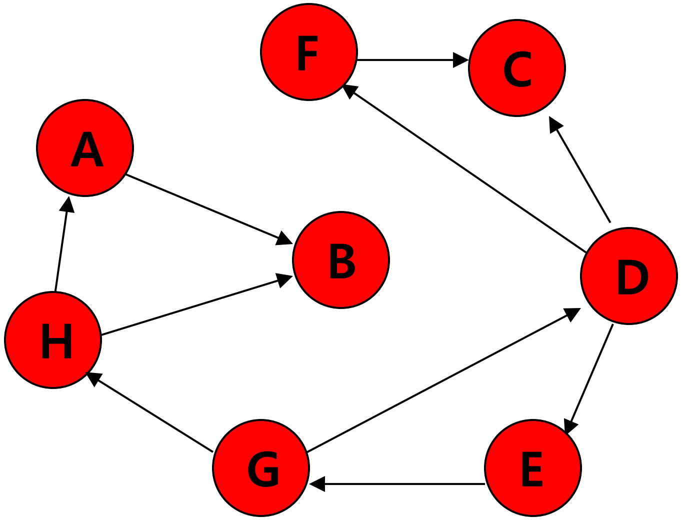
D

Visit F

The order nodes are visited:

D, C, E, G, H, A, B, F

Walk-Through

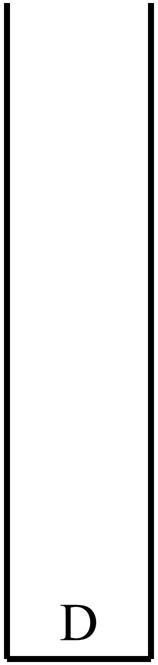


The order nodes are visited:

D, C, E, G, H, A, B, F

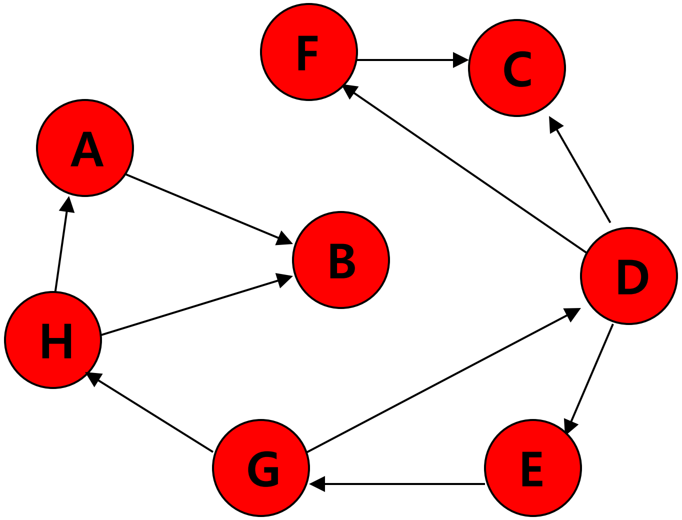
Visited Array

A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√



No unvisited nodes adjacent to F. Backtrack.

Walk-Through



Visited Array

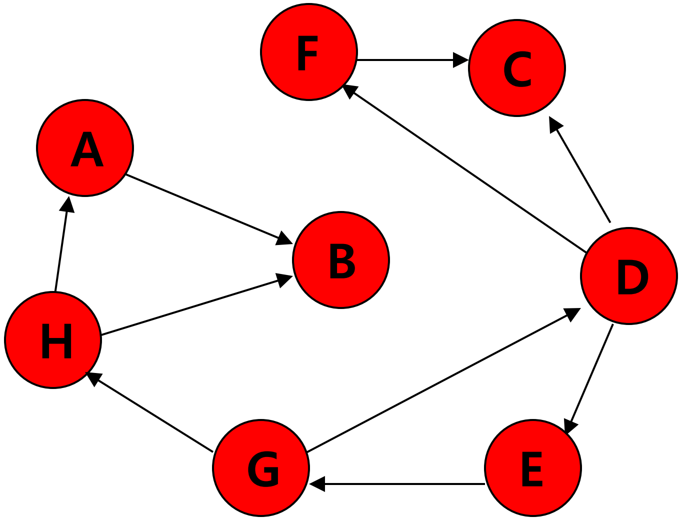
A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

The order nodes are visited:

D, C, E, G, H, A, B, F

No unvisited nodes adjacent to D. Backtrack.

Walk-Through



Visited Array

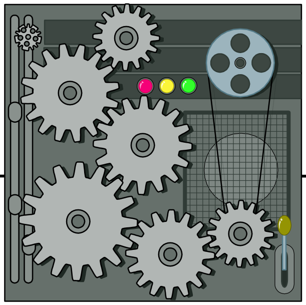
A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

The order nodes are visited:

D, C, E, G, H, A, B, F

Stack is empty. Depth-first traversal is done.

Analysis of DFS



- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or BACK
- Method incidentEdges is called once for each vertex
- DFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\Sigma_v \deg(v) = 2m$

Path Finding

- We can specialize the DFS algorithm to find a path between two given vertices ***u*** and ***z*** using the template method pattern
- We call ***DFS*(*G*, *u*)** with ***u*** as the start vertex
- We use a stack ***S*** to keep track of the path between the start vertex and the current vertex
- As soon as destination vertex ***z*** is encountered, we return the path as the contents of the stack

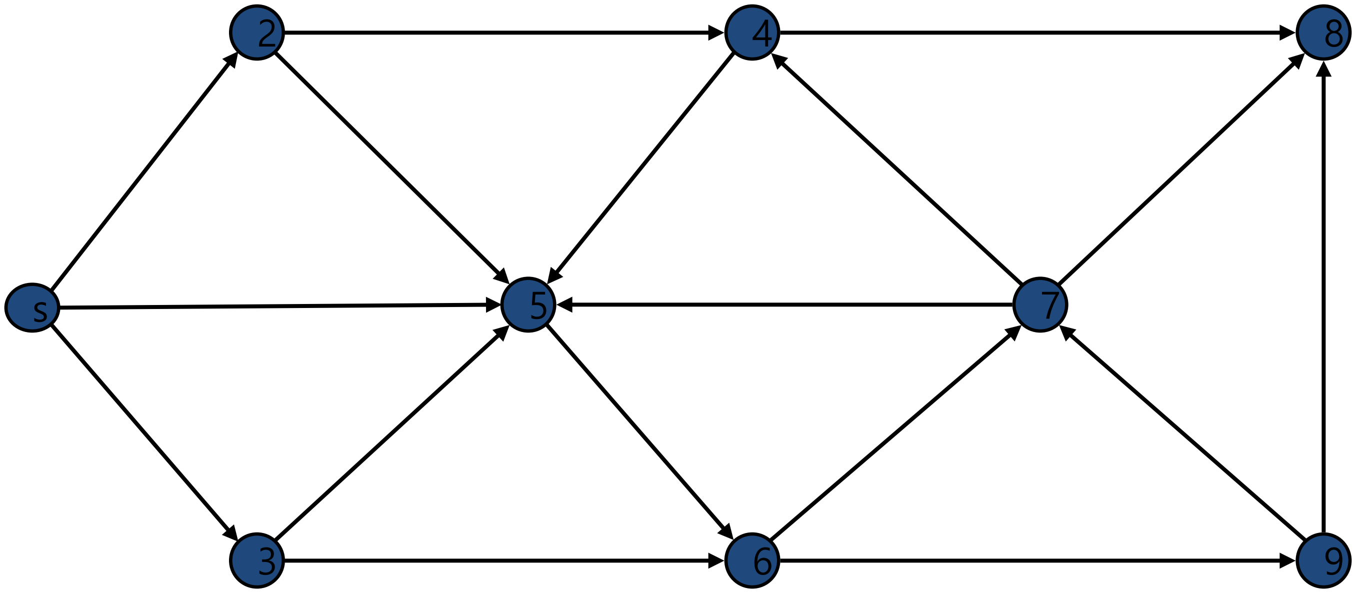
```
Algorithm pathDFS(G, v, z)
    setLabel(v, VISITED)
    S.push(v)
    if v = z
        return S.elements()
    for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
            w ← opposite(v,e)
            if getLabel(w) = UNEXPLORED
                setLabel(e, DISCOVERY)
                S.push(e)
                pathDFS(G, w, z)
                S.pop(e)
            else
                setLabel(e, BACK)
    S.pop(v)
```

Cycle Finding

- We can specialize the DFS algorithm to find a simple cycle using the template method pattern
- We use a stack ***S*** to keep track of the path between the start vertex and the current vertex
- As soon as a back edge (***v***, ***w***) is encountered, we return the cycle as the portion of the stack from the top to vertex ***w***

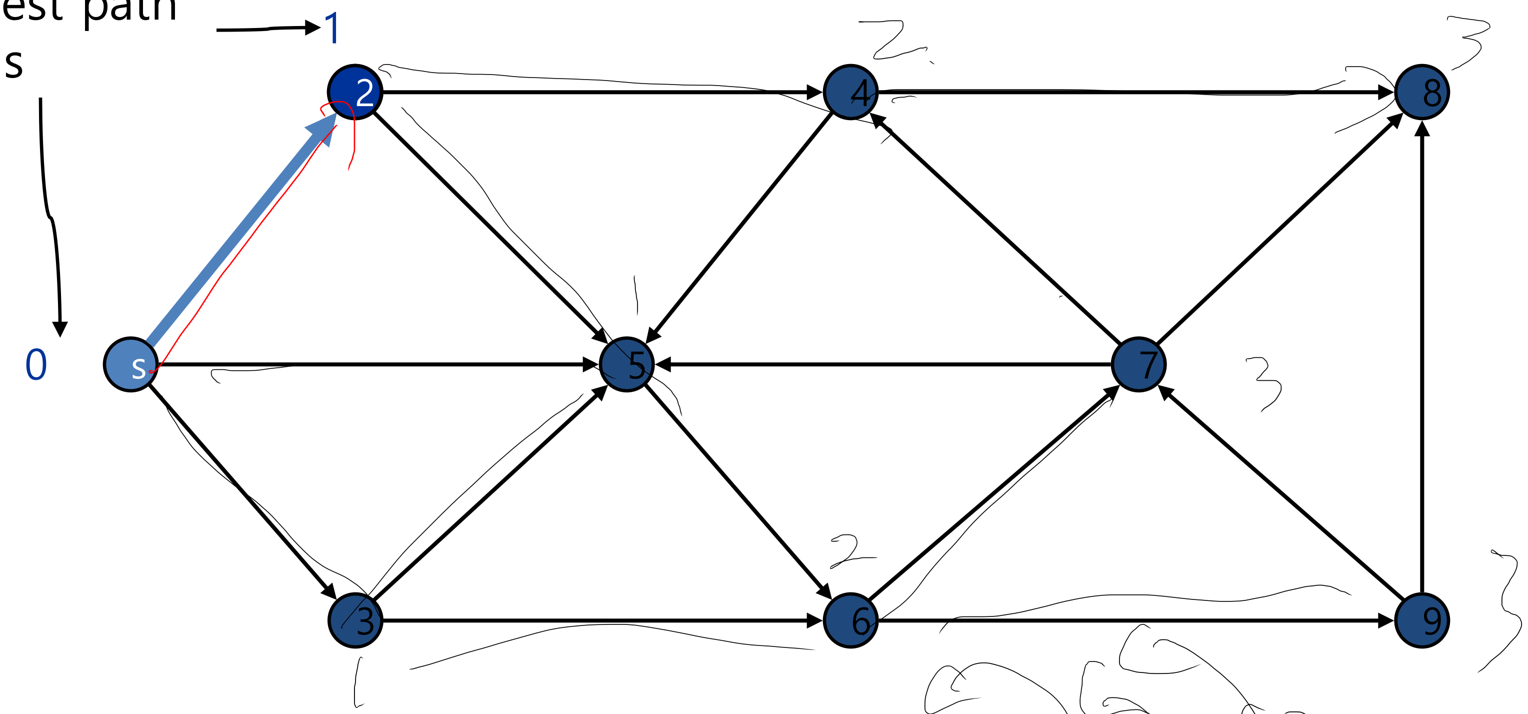
```
Algorithm cycleDFS(G, v, z)
    setLabel(v, VISITED)
    S.push(v)
    for all  $e \in$ G.incidentEdges(v)
        if  getLabel(e) = UNEXPLORED
            $w \leftarrow$ opposite(v,e)
            S.push(e)
            if getLabel(w) = UNEXPLORED
                setLabel(e, DISCOVERY)
                pathDFS(G, w, z)
                S.pop(e)
            else
                $T \leftarrow$ new empty stack
                repeat
                    $o \leftarrow$ S.pop()
                    T.push(o)
                until $o = w$
                return T.elements()
    S.pop(v)
```

Breadth First Search



Breadth First Search

Shortest path from s



Undiscovered

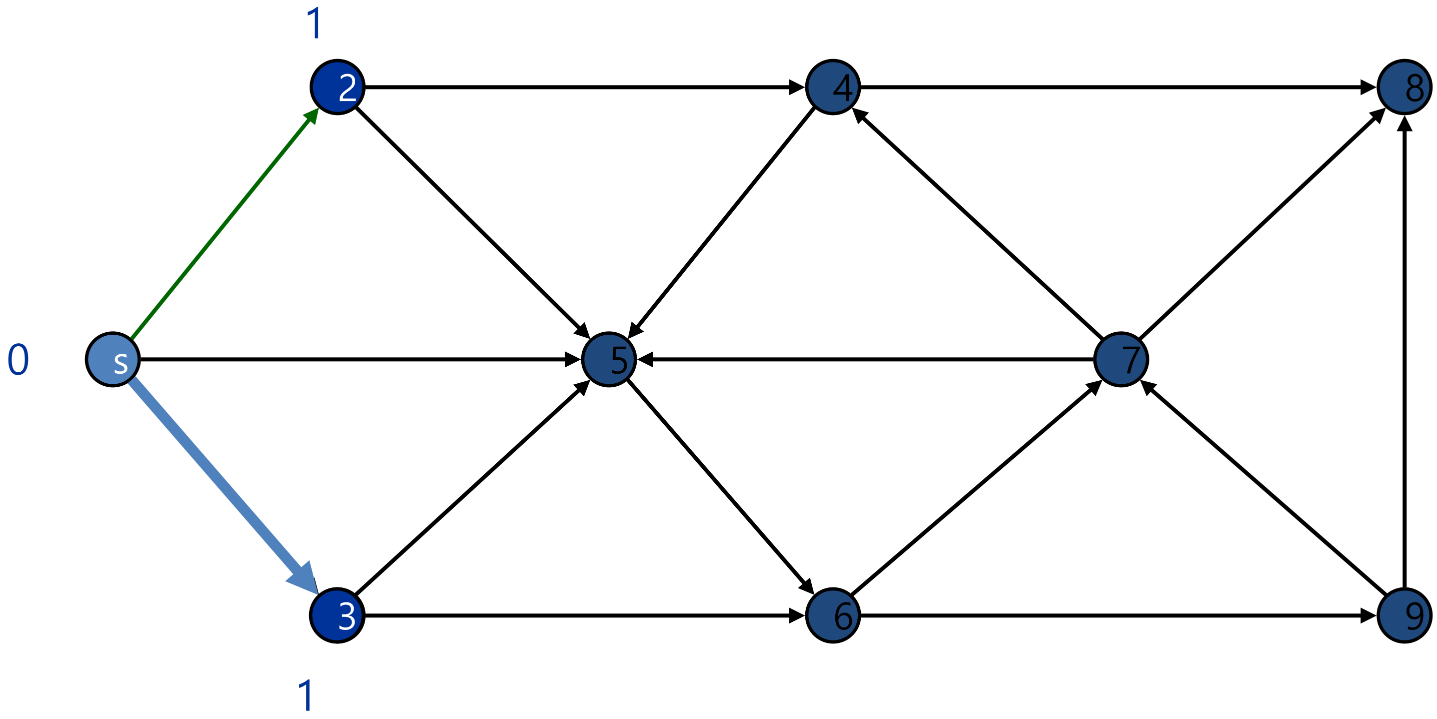
Discovered

Top of queue

Finished

Queue: s 2 3 5 4 6 8 7 9

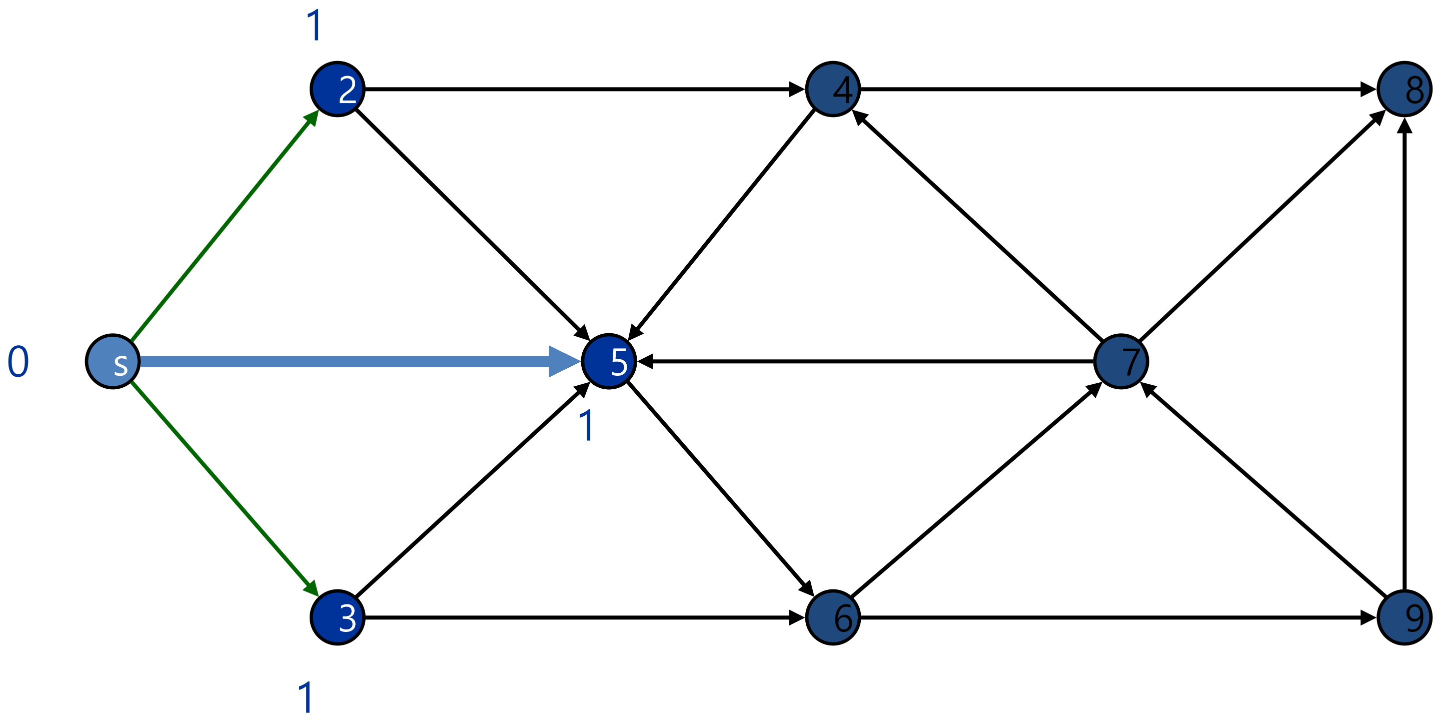
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: s 2

Breadth First Search



Undiscovered

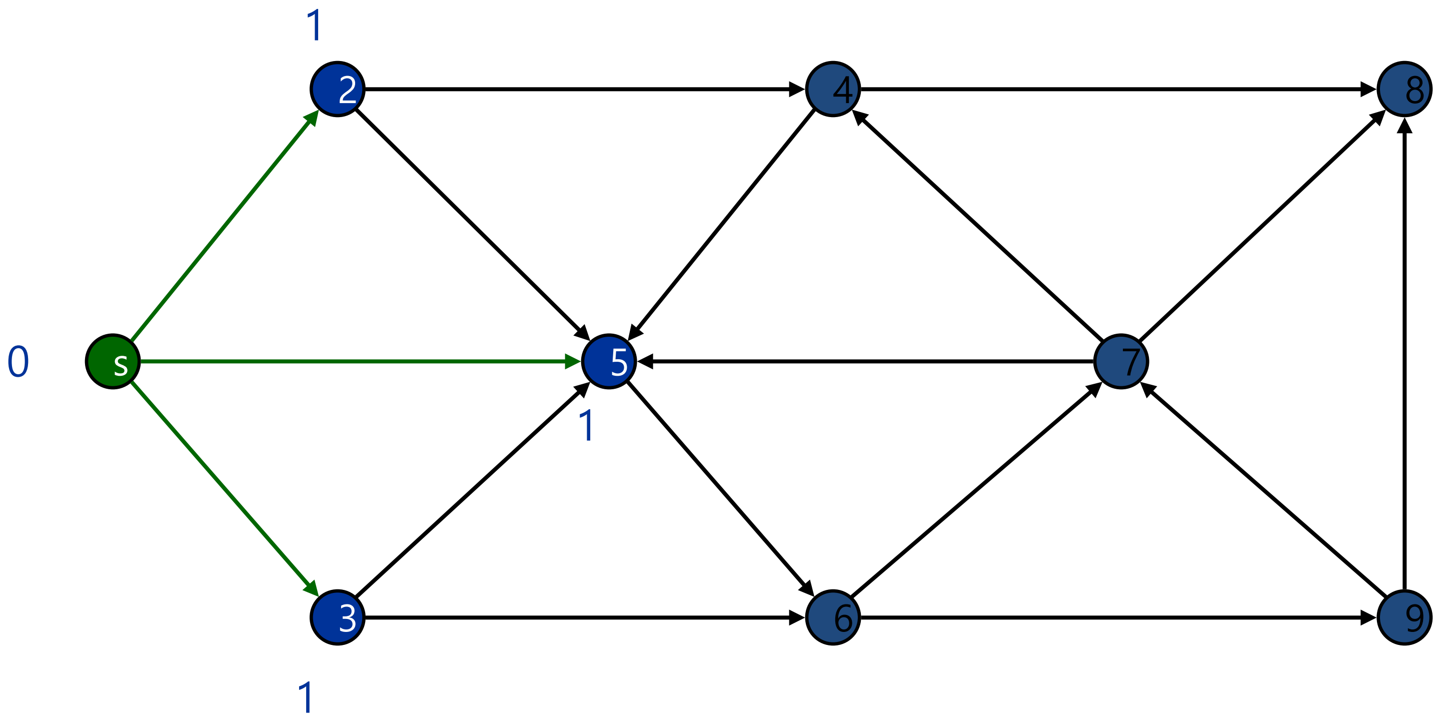
Discovered

Top of queue

Finished

Queue: s 2 3

Breadth First Search



Undiscovered

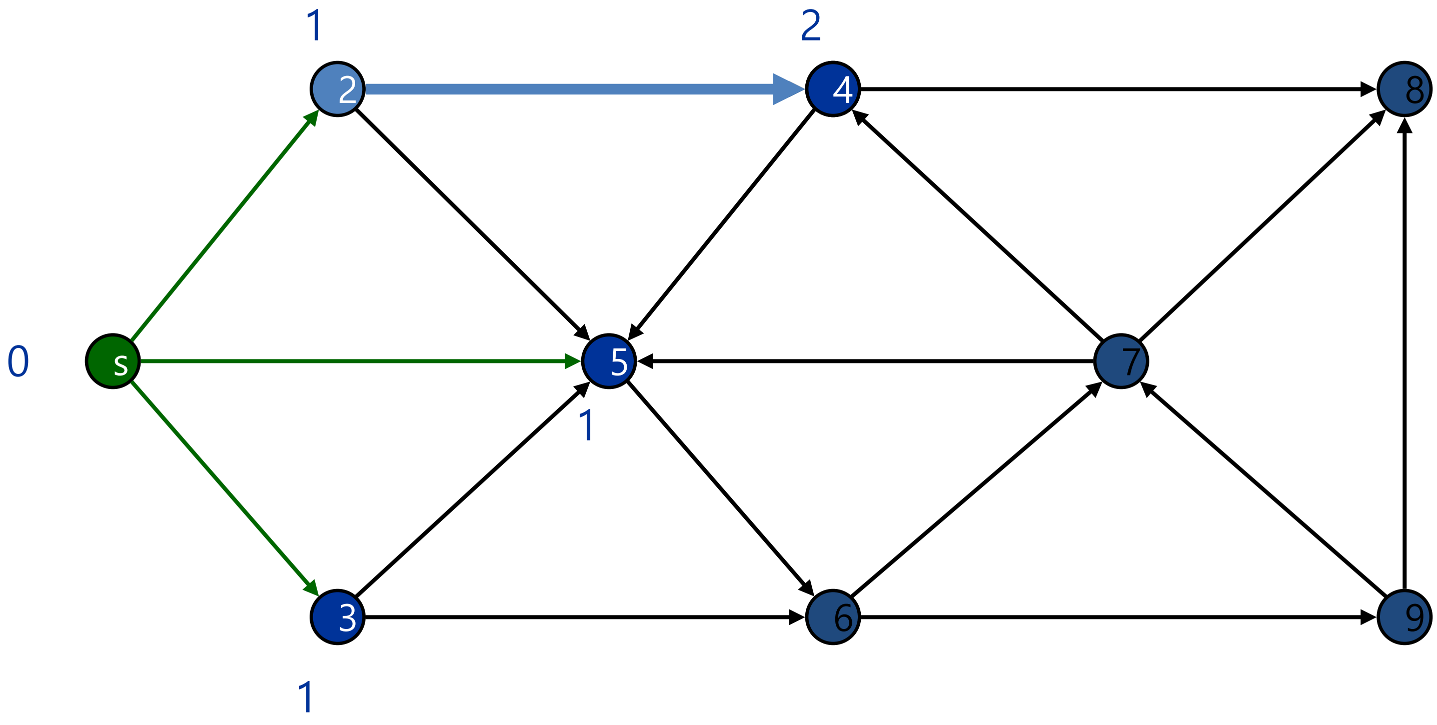
Discovered

Top of queue

Finished

Queue: 2 3 5

Breadth First Search



Undiscovered

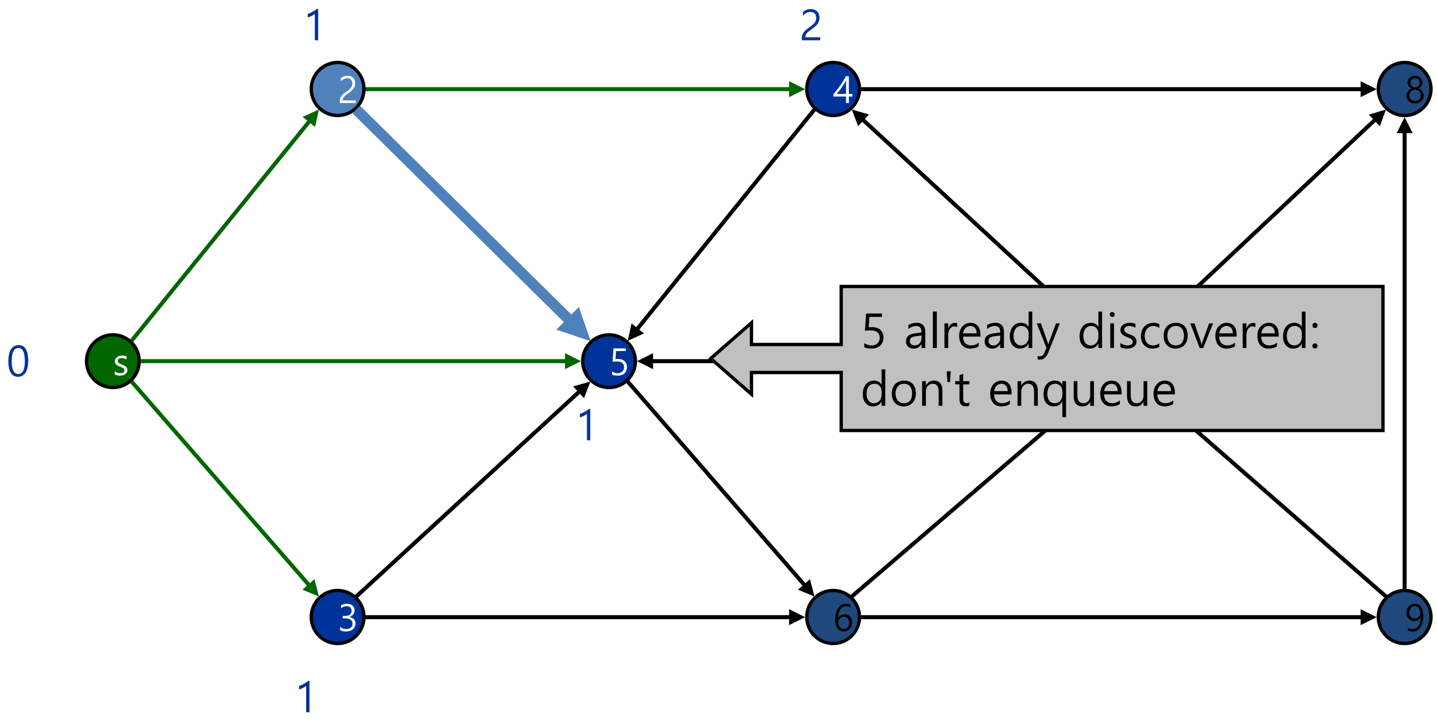
Discovered

Top of queue

Finished

Queue: 2 3 5

Breadth First Search

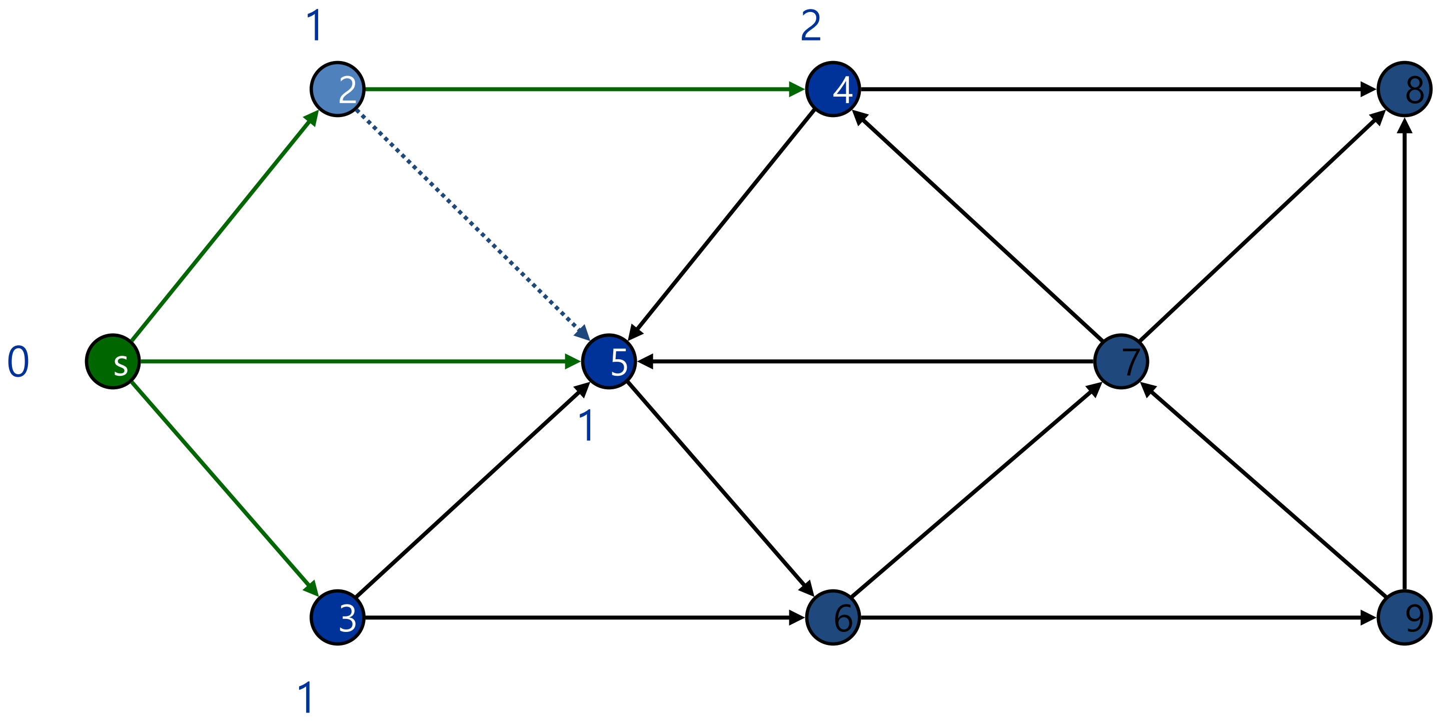


5 already discovered: don't enqueue

- Undiscovered
- Discovered
- Top of queue
- Finished

Queue: 2 3 5 4

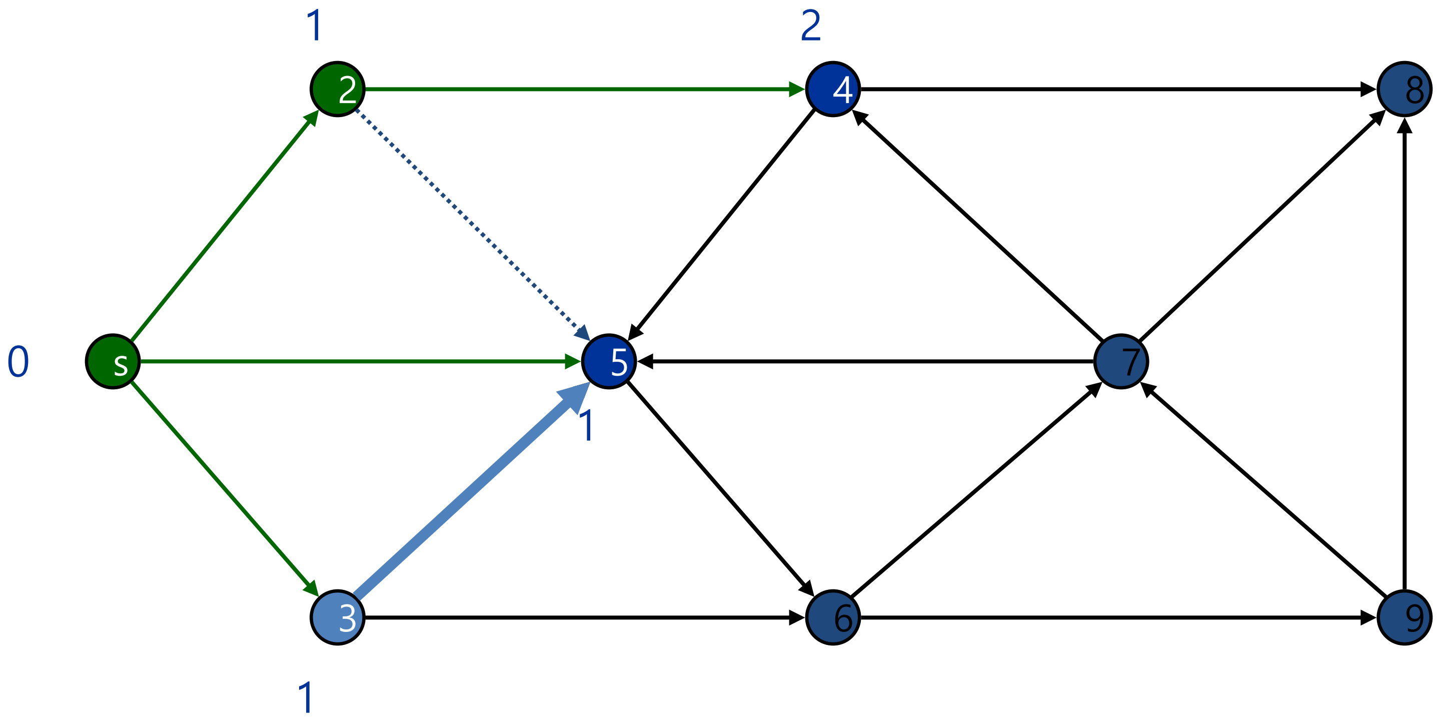
Breadth First Search



- Undiscovered
- Discovered
- Top of queue
- Finished

Queue: 2 3 5 4

Breadth First Search



Undiscovered

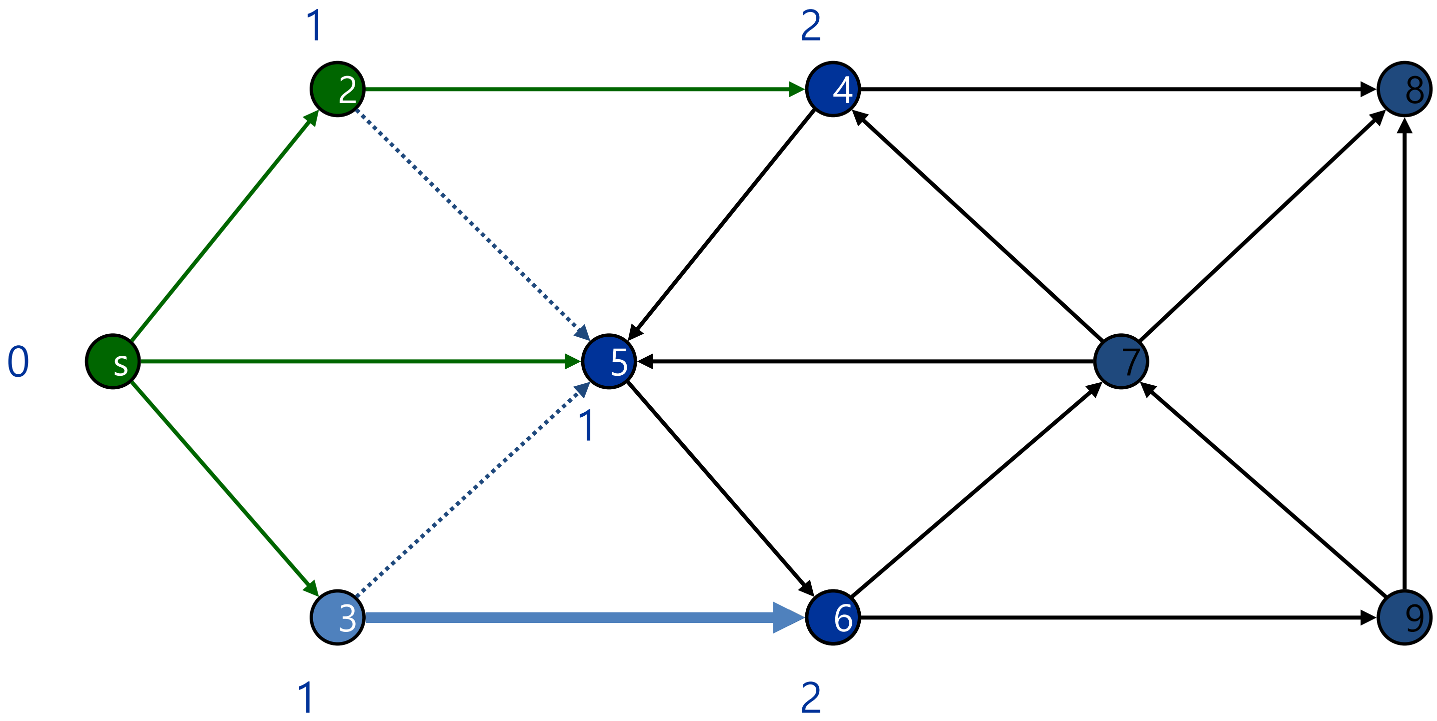
Discovered

Top of queue

Finished

Queue: 3 5 4

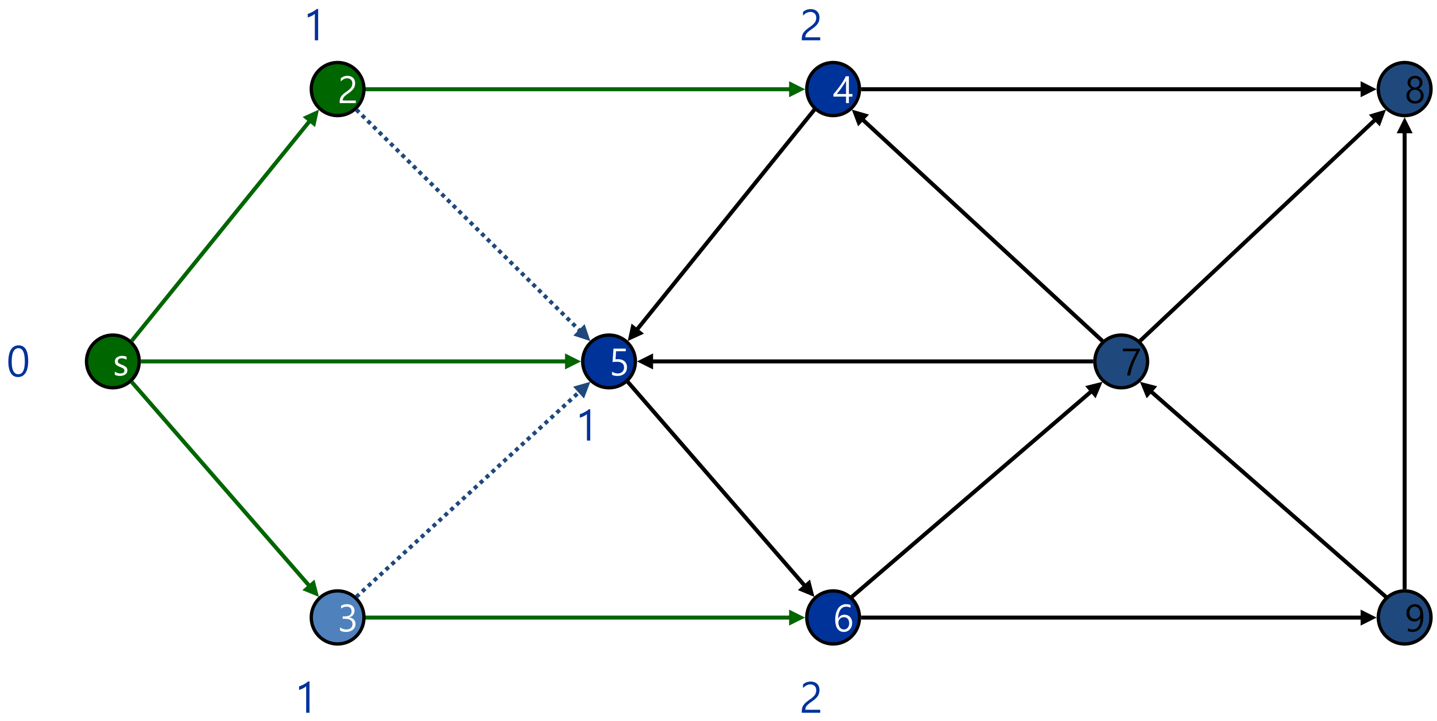
Breadth First Search



- Undiscovered
- Discovered
- Top of queue
- Finished

Queue: 3 5 4

Breadth First Search



Undiscovered

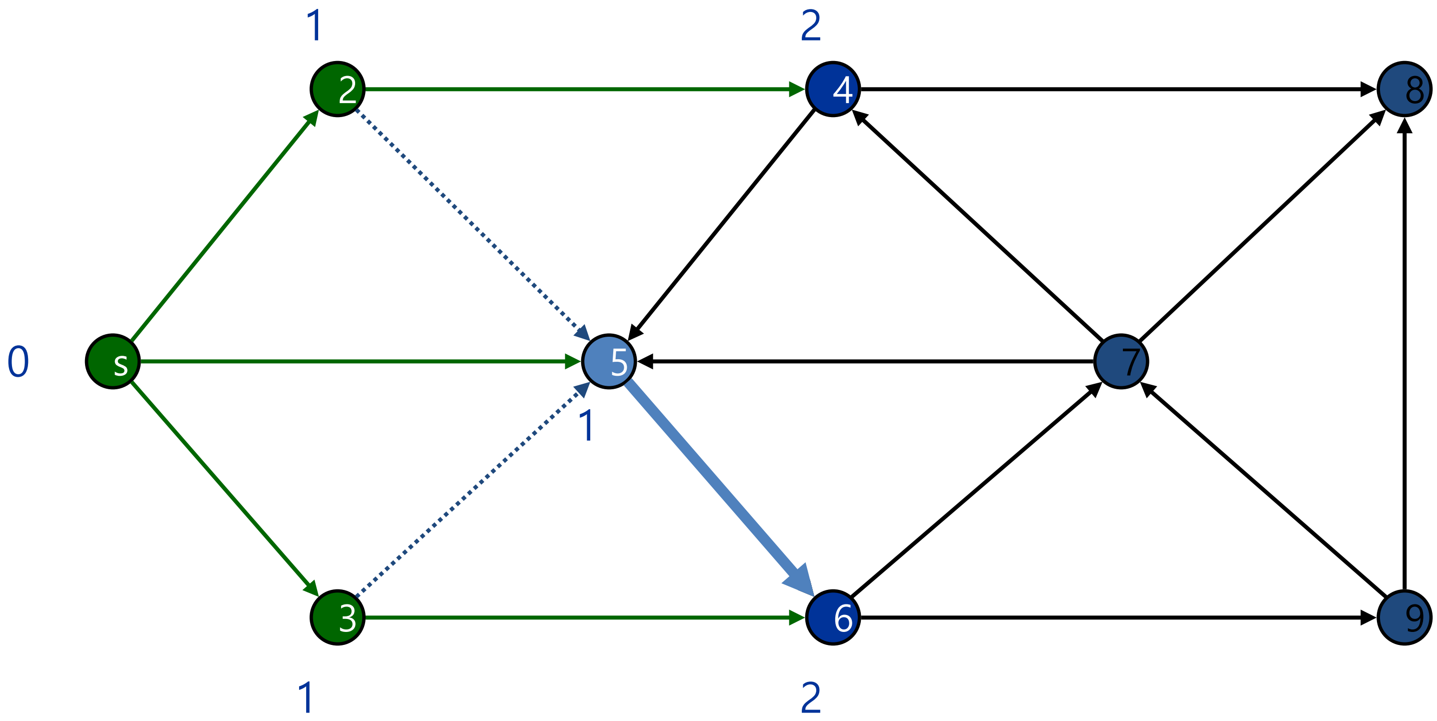
Discovered

Top of queue

Finished

Queue: 3 5 4 6

Breadth First Search



Undiscovered

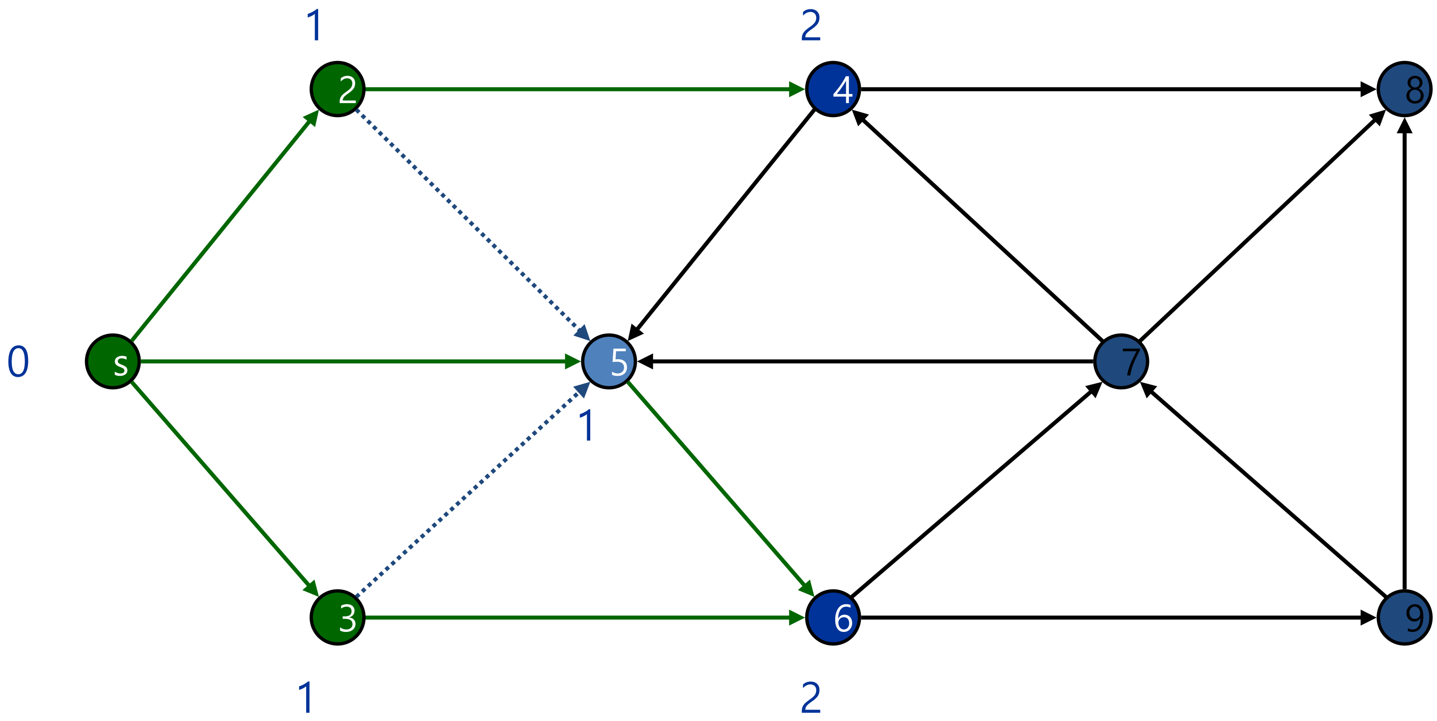
Discovered

Top of queue

Finished

Queue: 5 4 6

Breadth First Search



Undiscovered

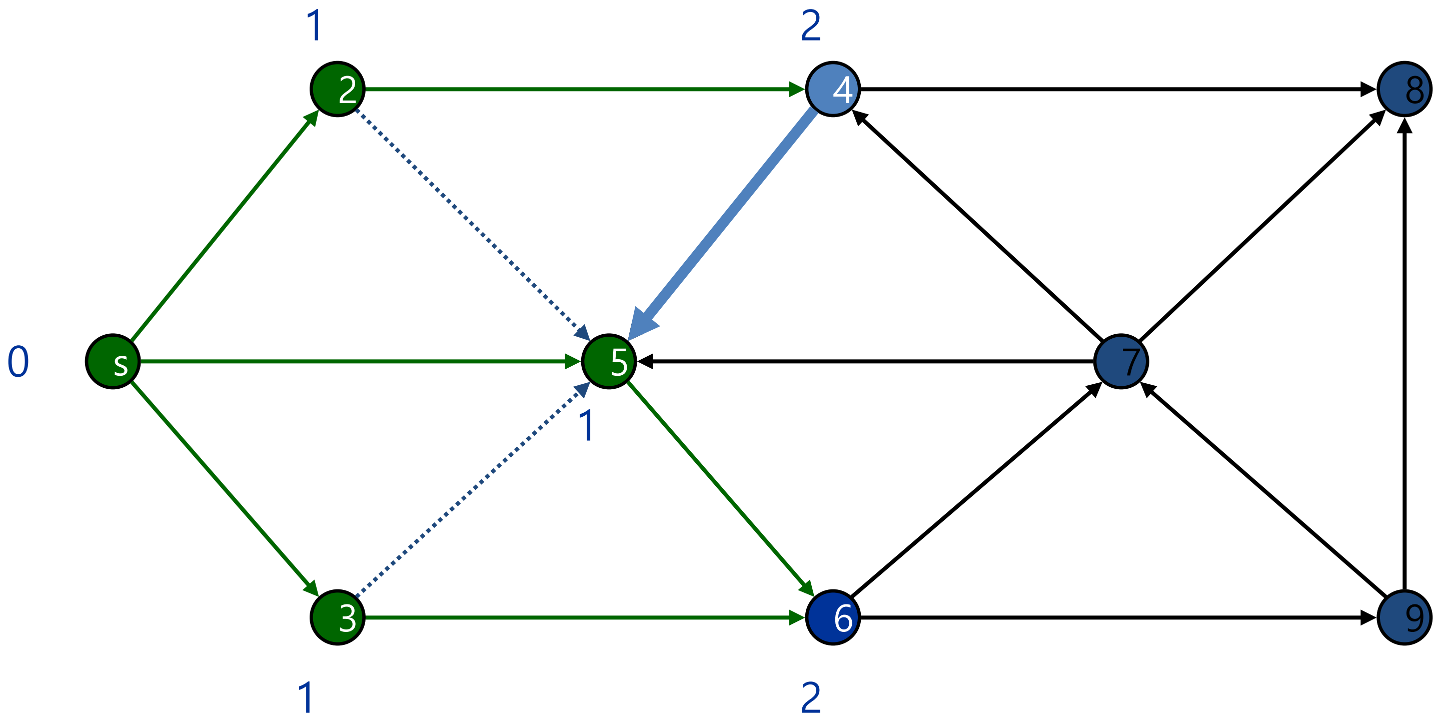
Discovered

Top of queue

Finished

Queue: 5 4 6

Breadth First Search



Undiscovered

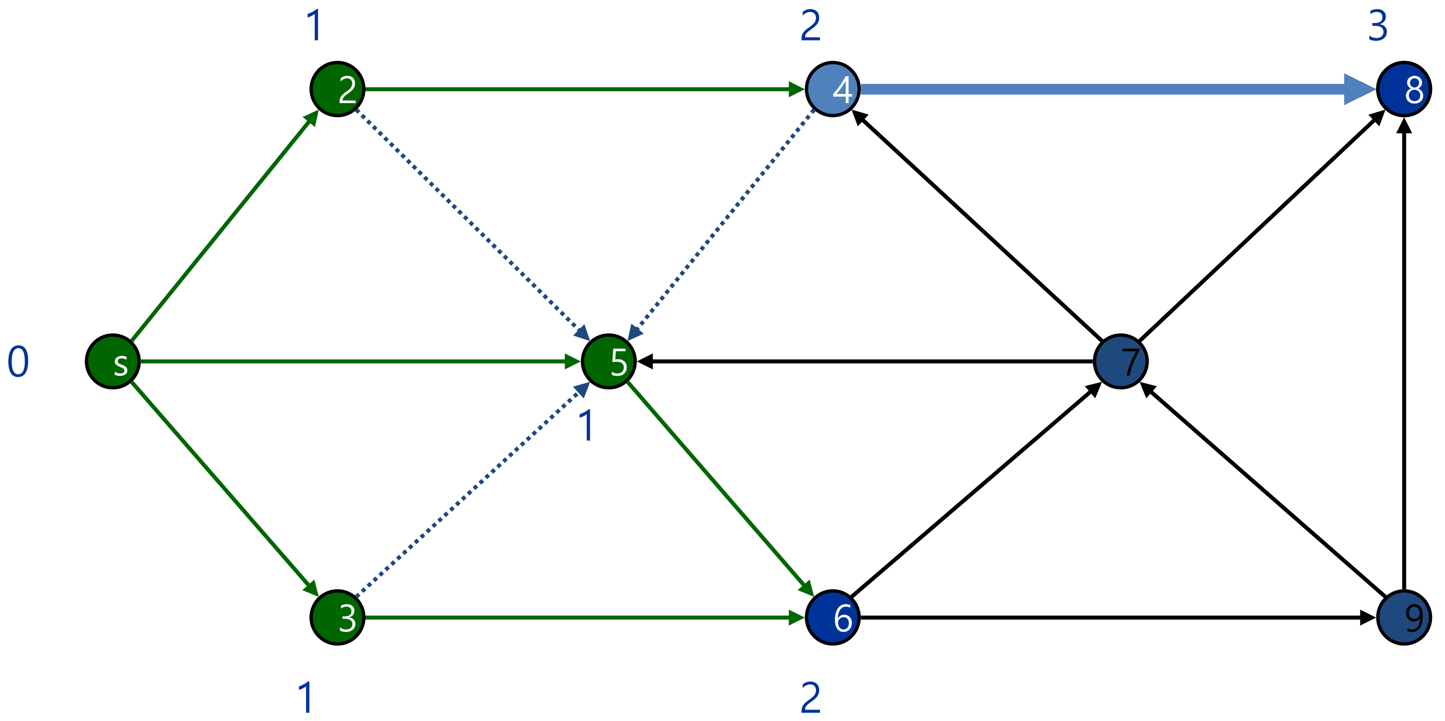
Discovered

Top of queue

Finished

Queue: 4 6

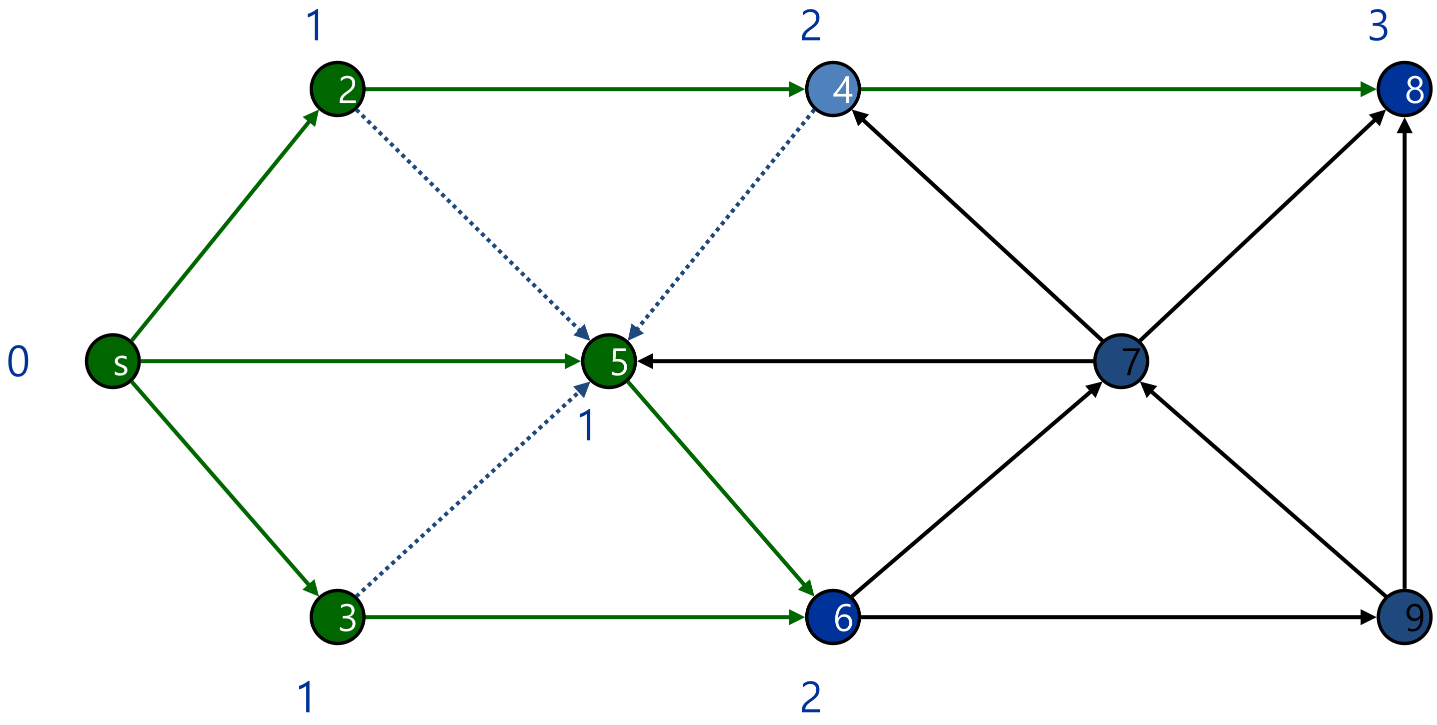
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 4 6

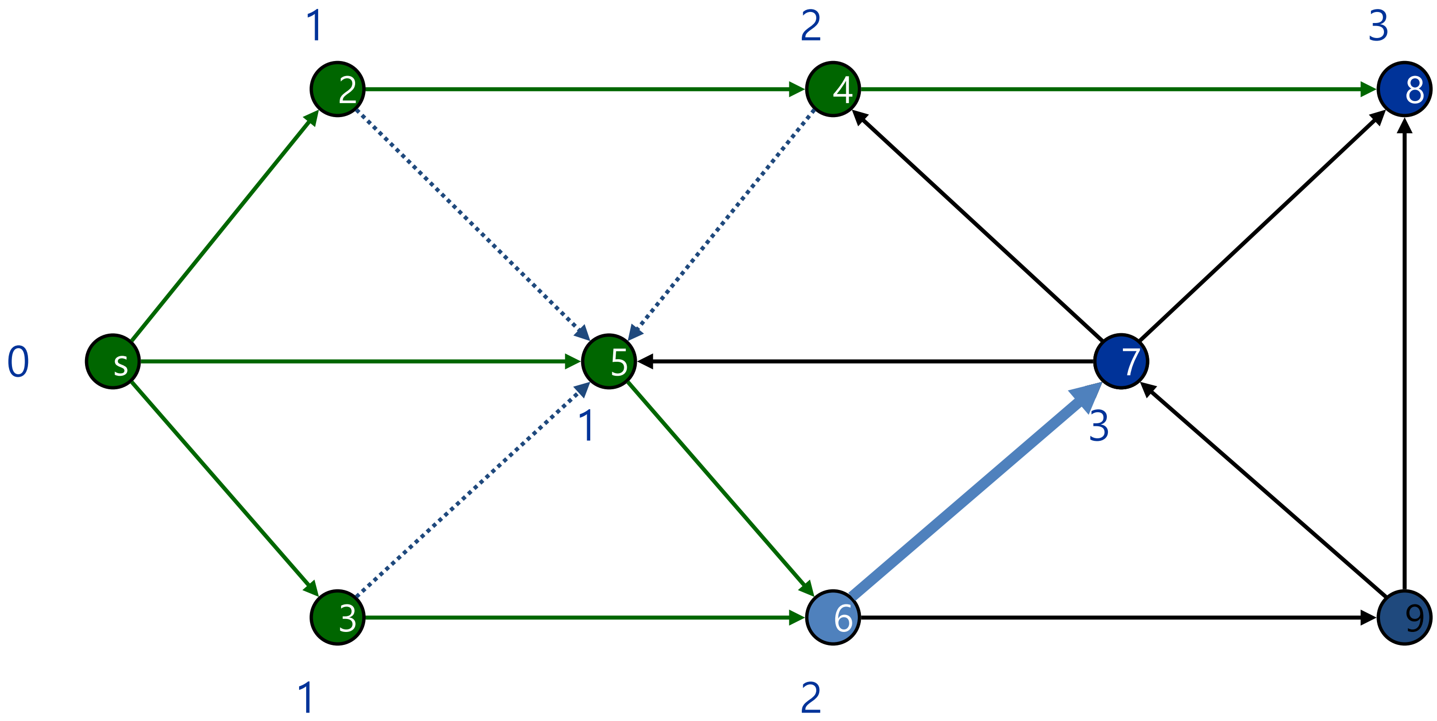
Breadth First Search



- Undiscovered
- Discovered
- Top of queue
- Finished

Queue: 4 6 8

Breadth First Search



Undiscovered

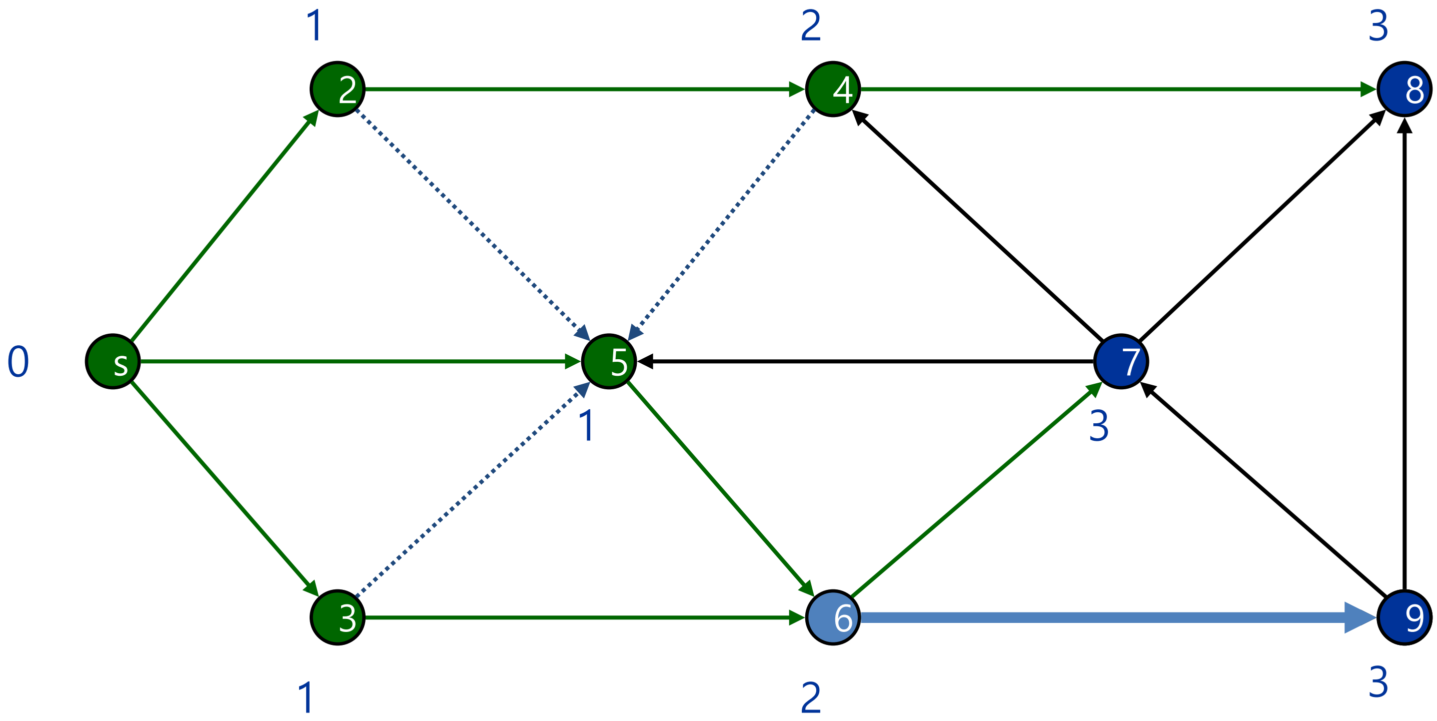
Discovered

Top of queue

Finished

Queue: 6 8

Breadth First Search



Undiscovered

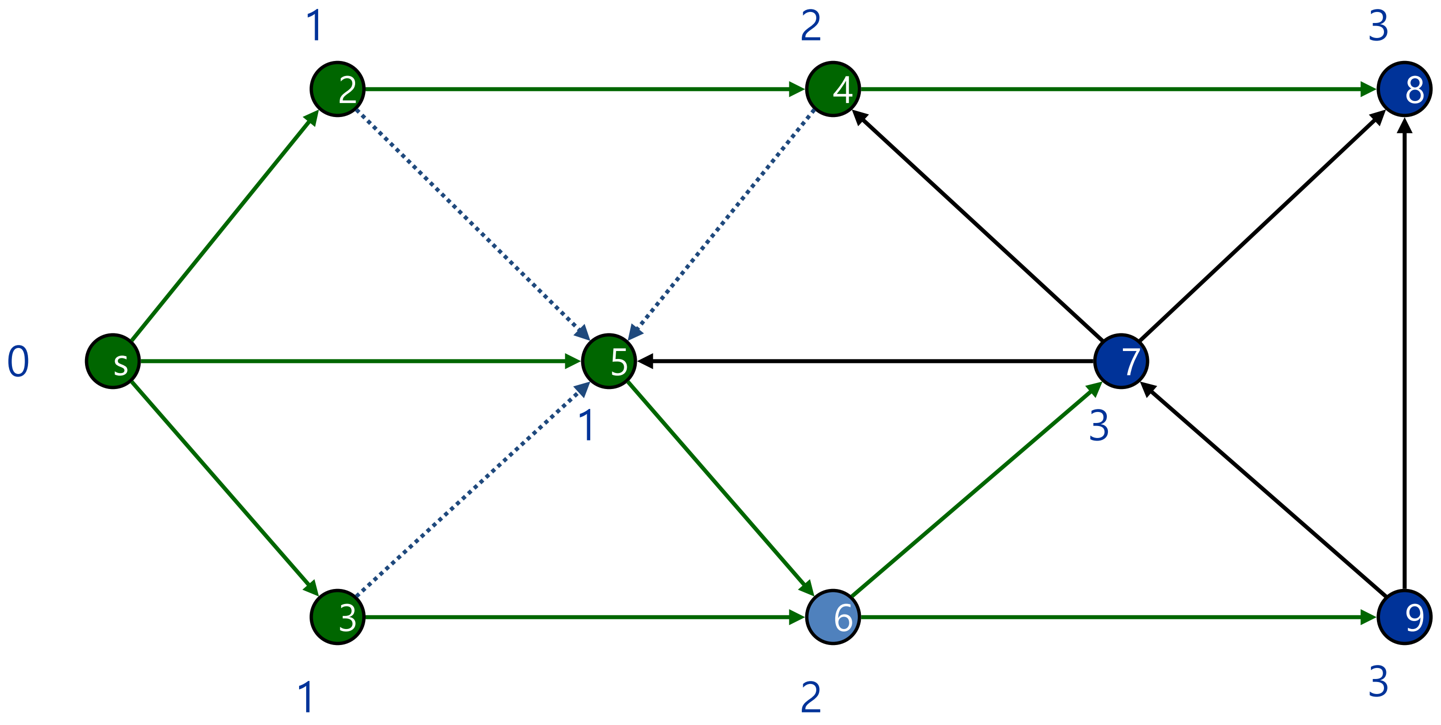
Discovered

Top of queue

Finished

Queue: 6 8 7

Breadth First Search



Undiscovered

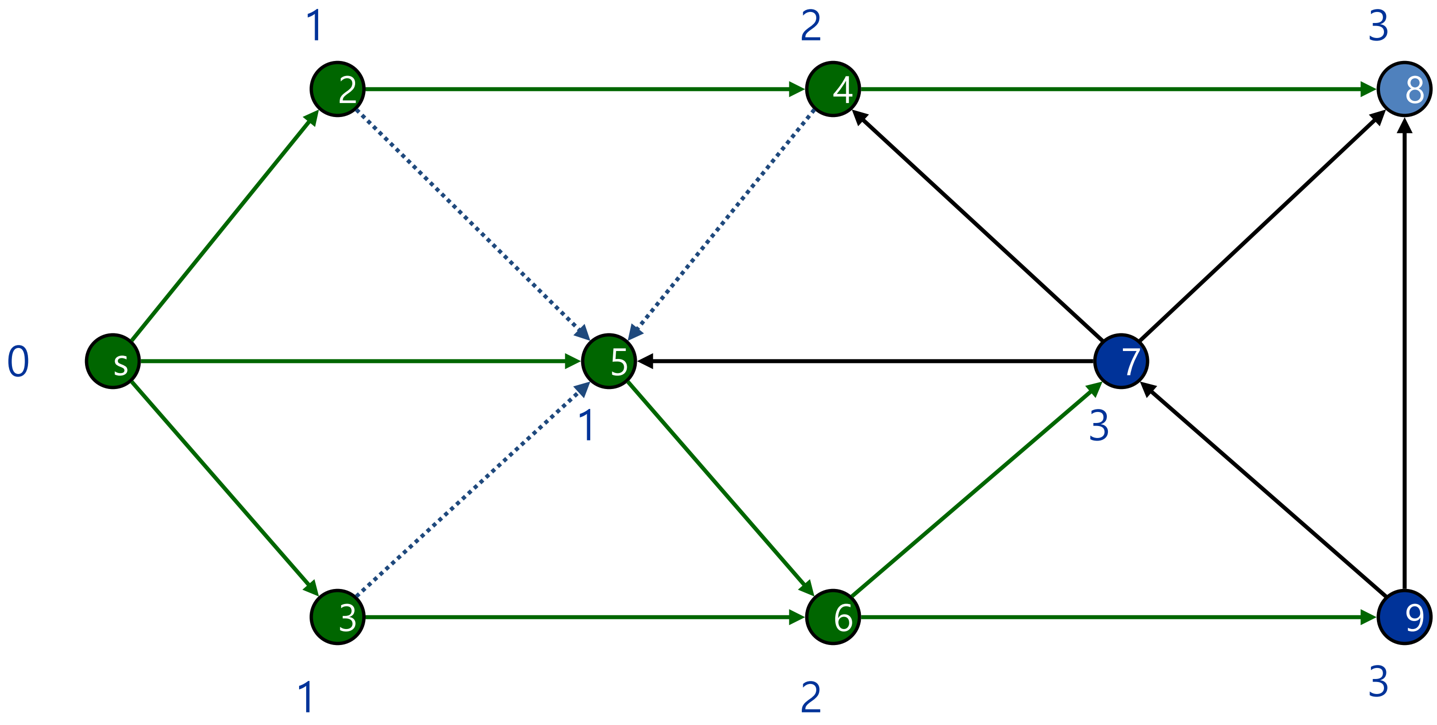
Discovered

Top of queue

Finished

Queue: 6 8 7 9

Breadth First Search



Undiscovered

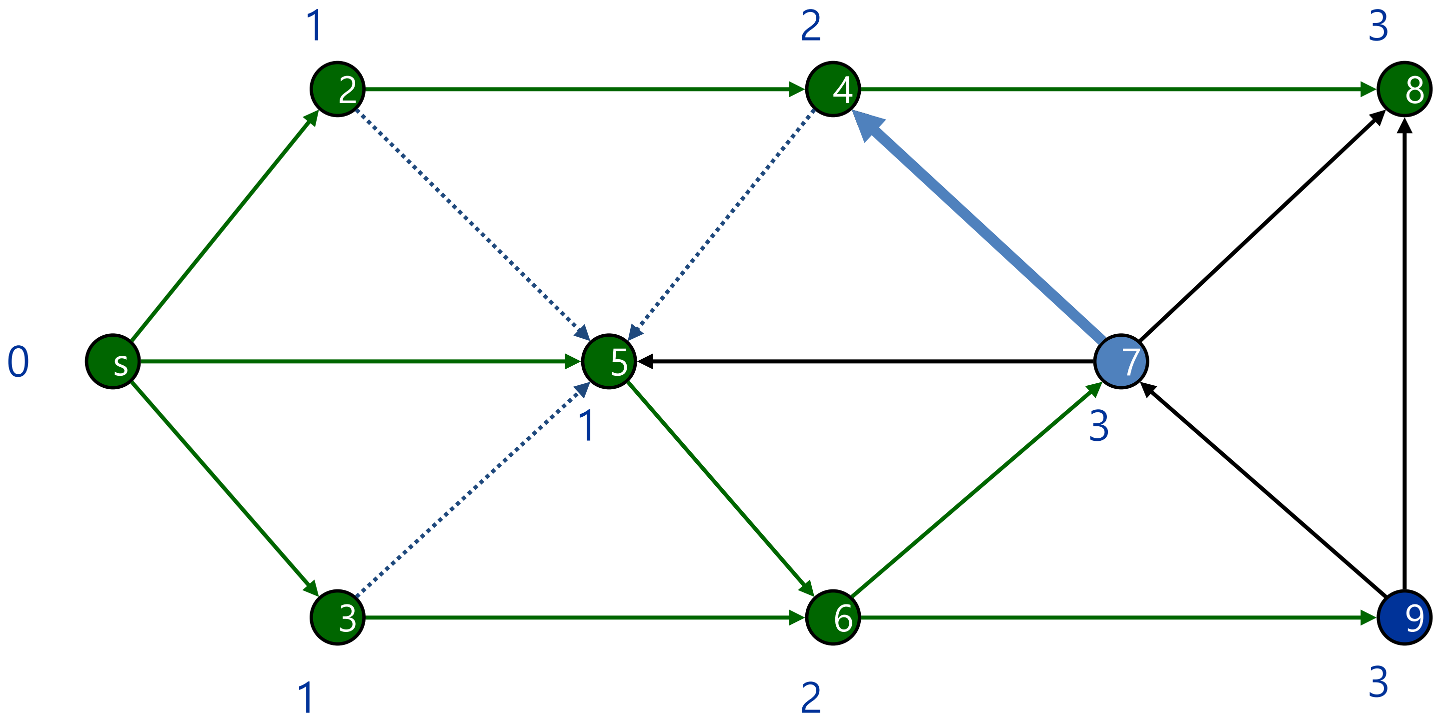
Discovered

Top of queue

Finished

Queue: 8 7 9

Breadth First Search



Undiscovered

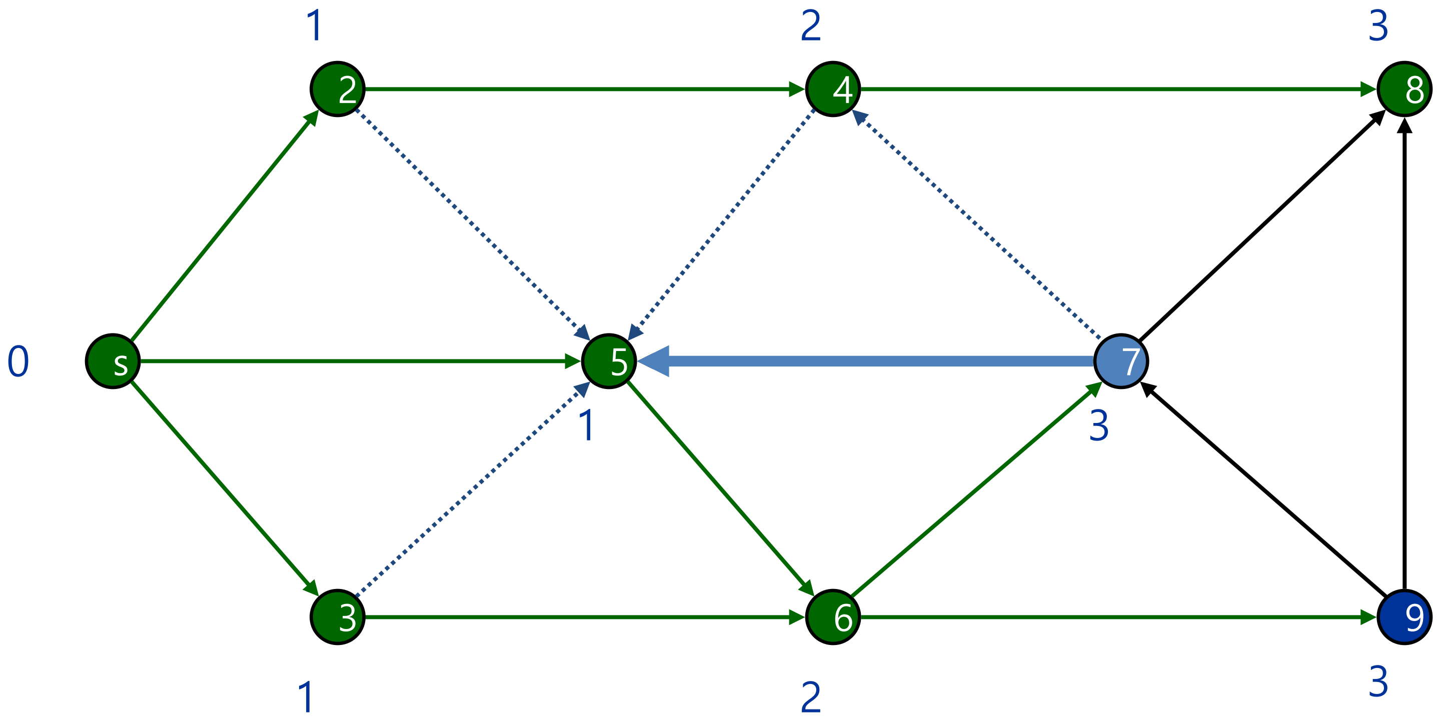
Discovered

Top of queue

Finished

Queue: 7 9

Breadth First Search



Undiscovered

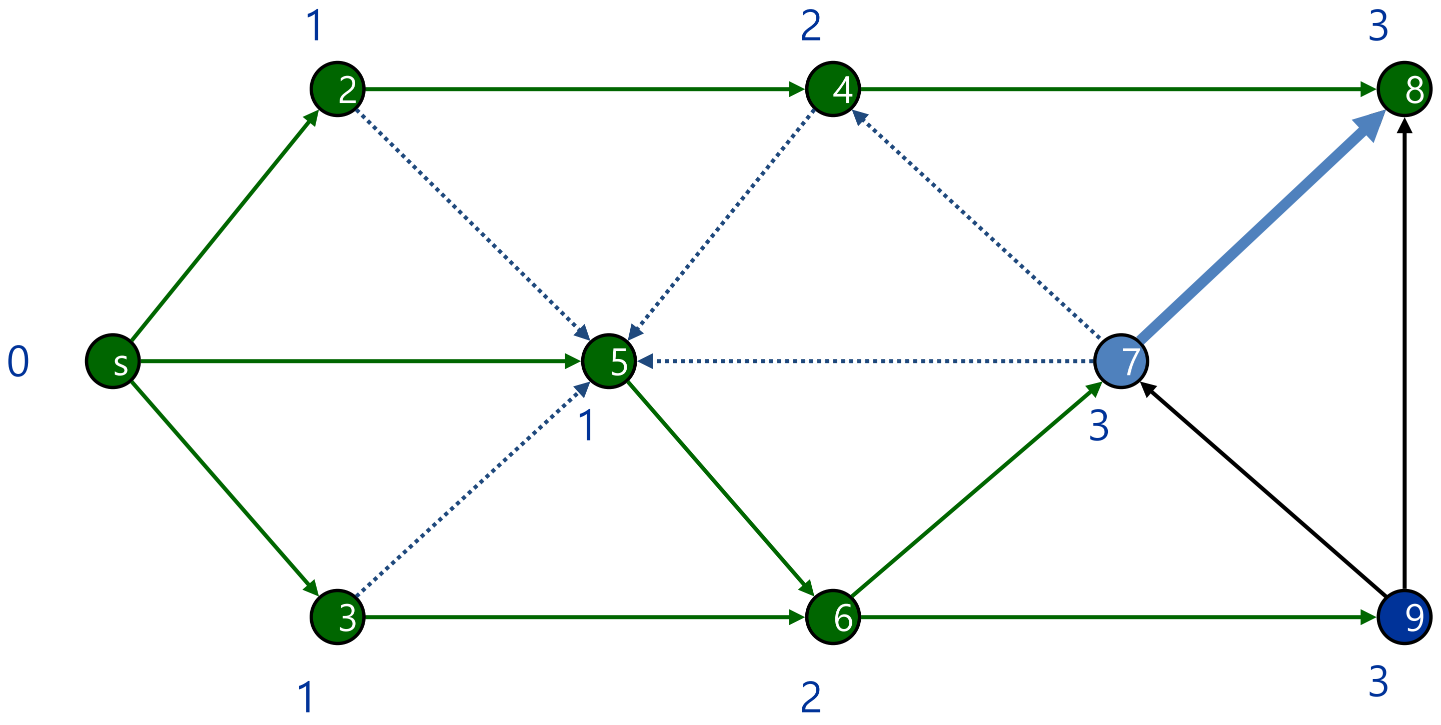
Discovered

Top of queue

Finished

Queue: 7 9

Breadth First Search



Undiscovered

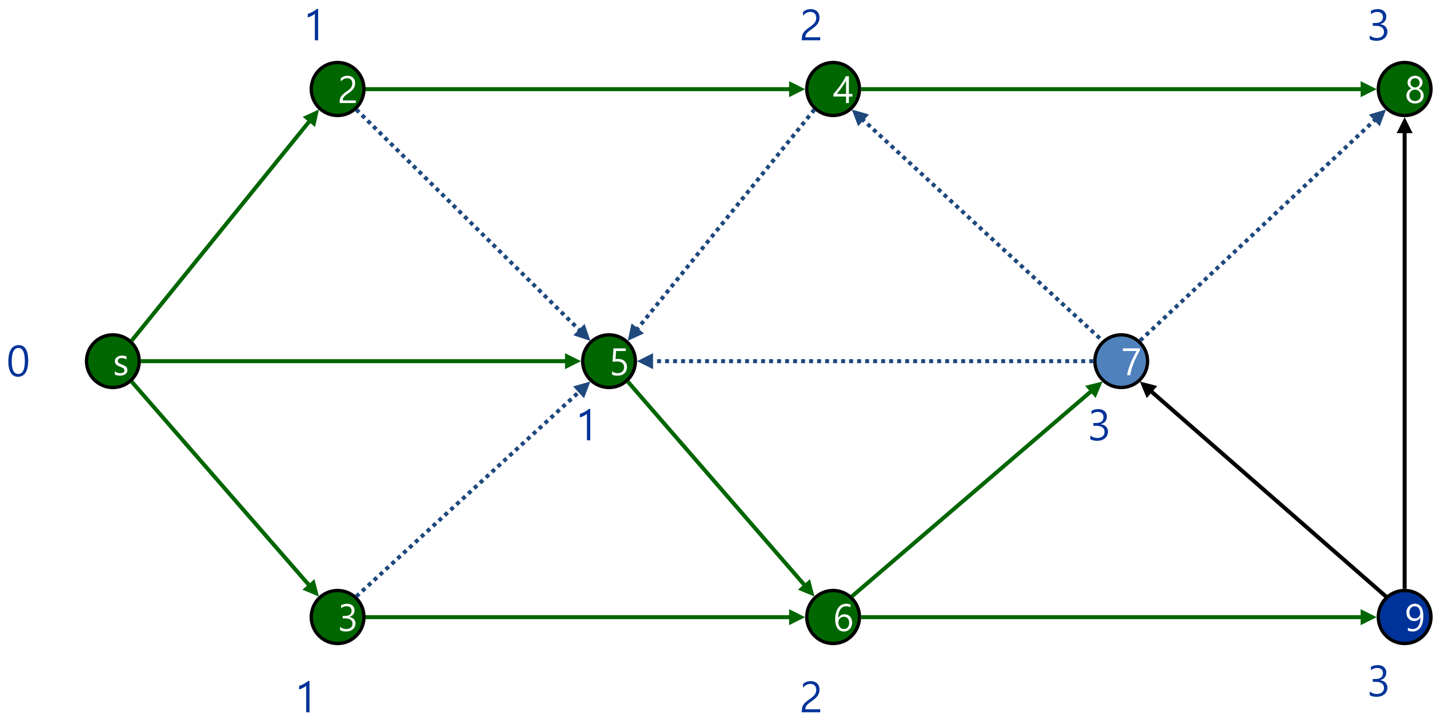
Discovered

Top of queue

Finished

Queue: 7 9

Breadth First Search



Undiscovered

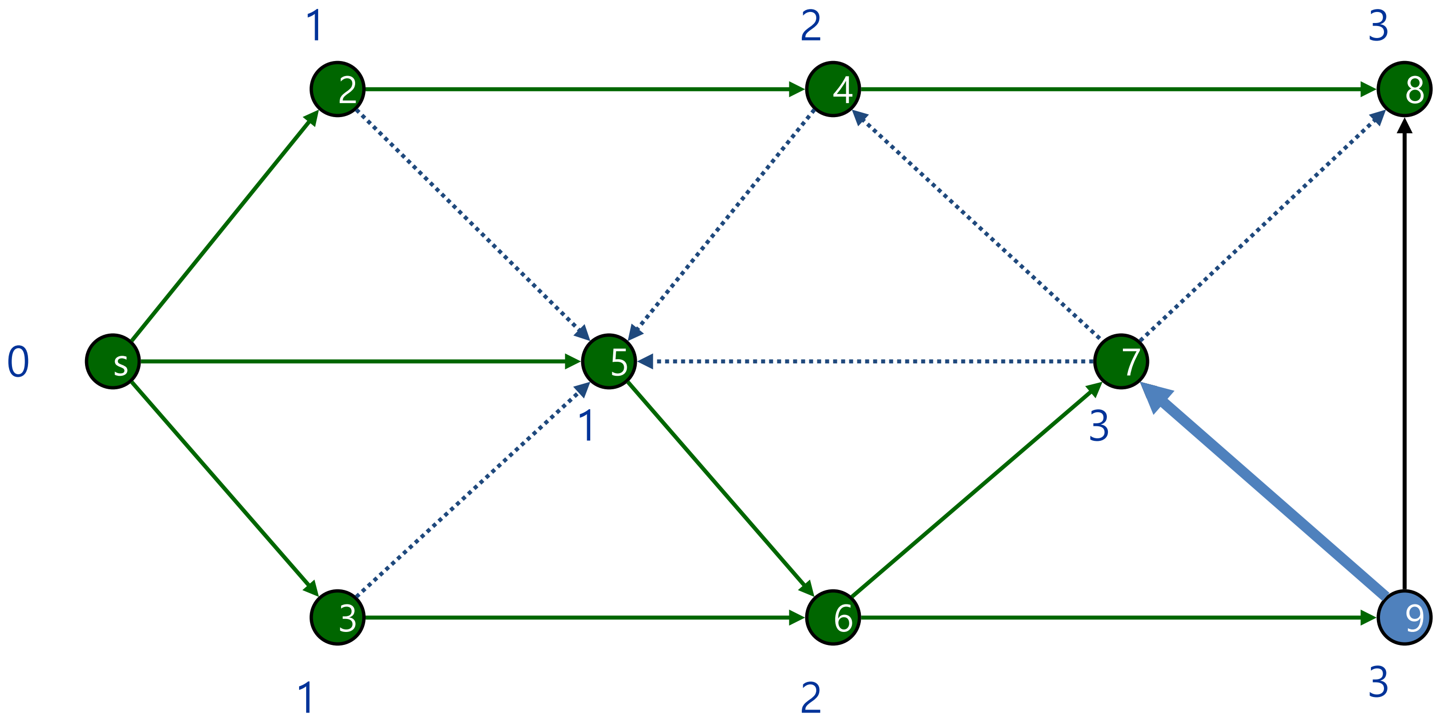
Discovered

Top of queue

Finished

Queue: 7 9

Breadth First Search



Undiscovered

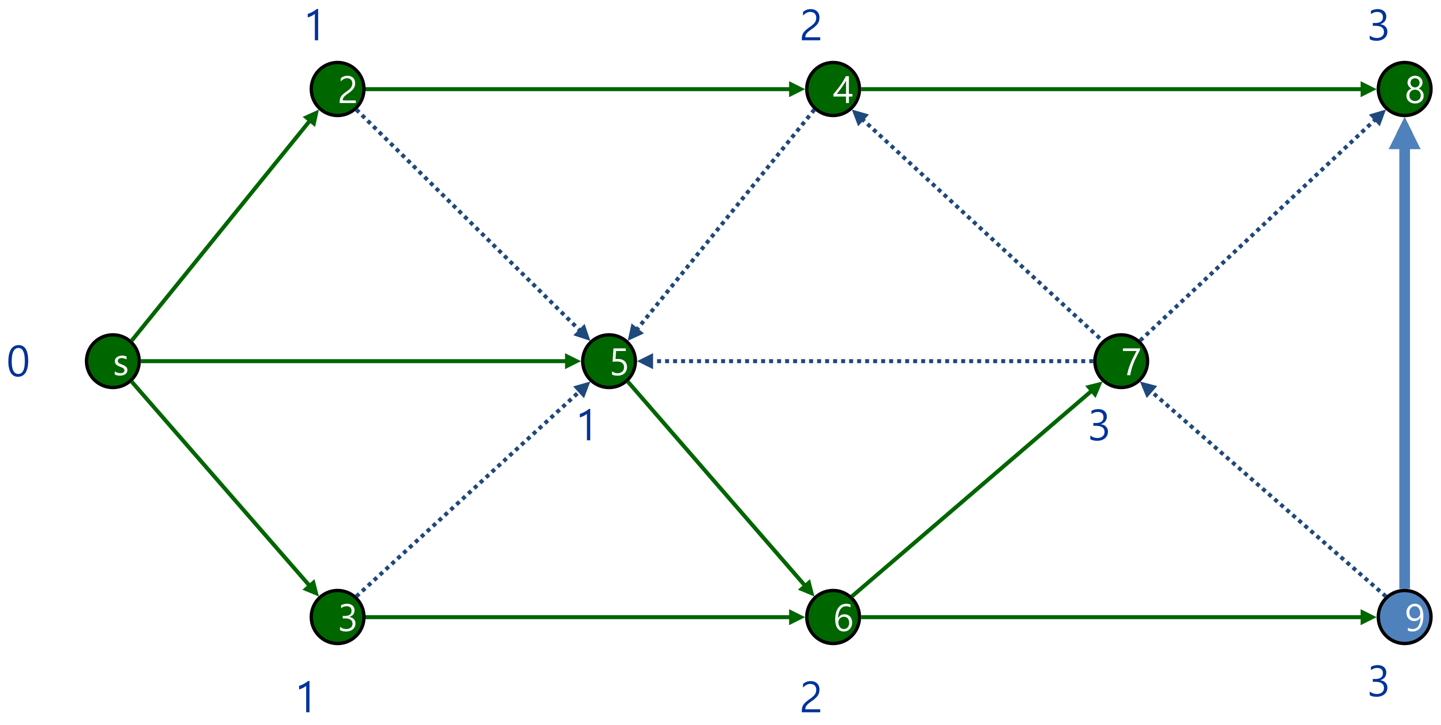
Discovered

Top of queue

Finished

Queue: 9

Breadth First Search



Undiscovered

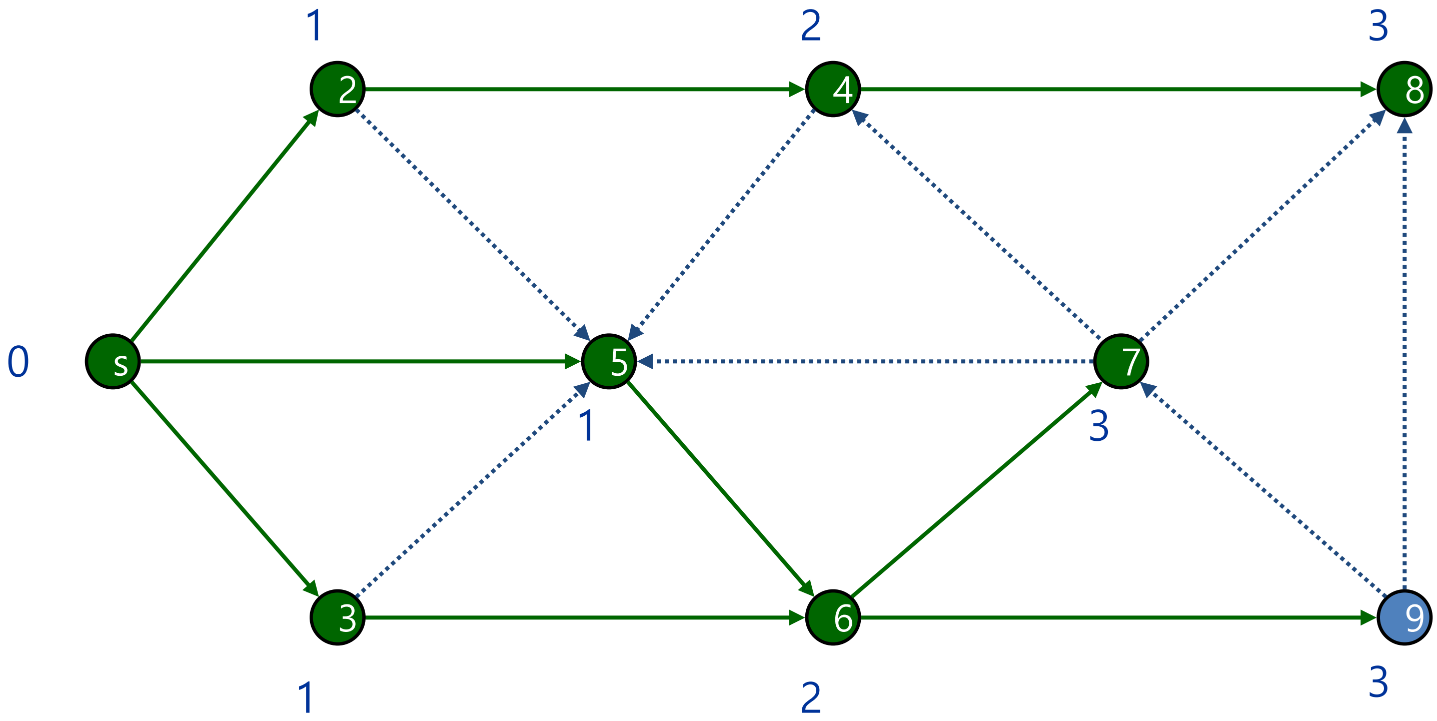
Discovered

Top of queue

Finished

Queue: 9

Breadth First Search



Undiscovered

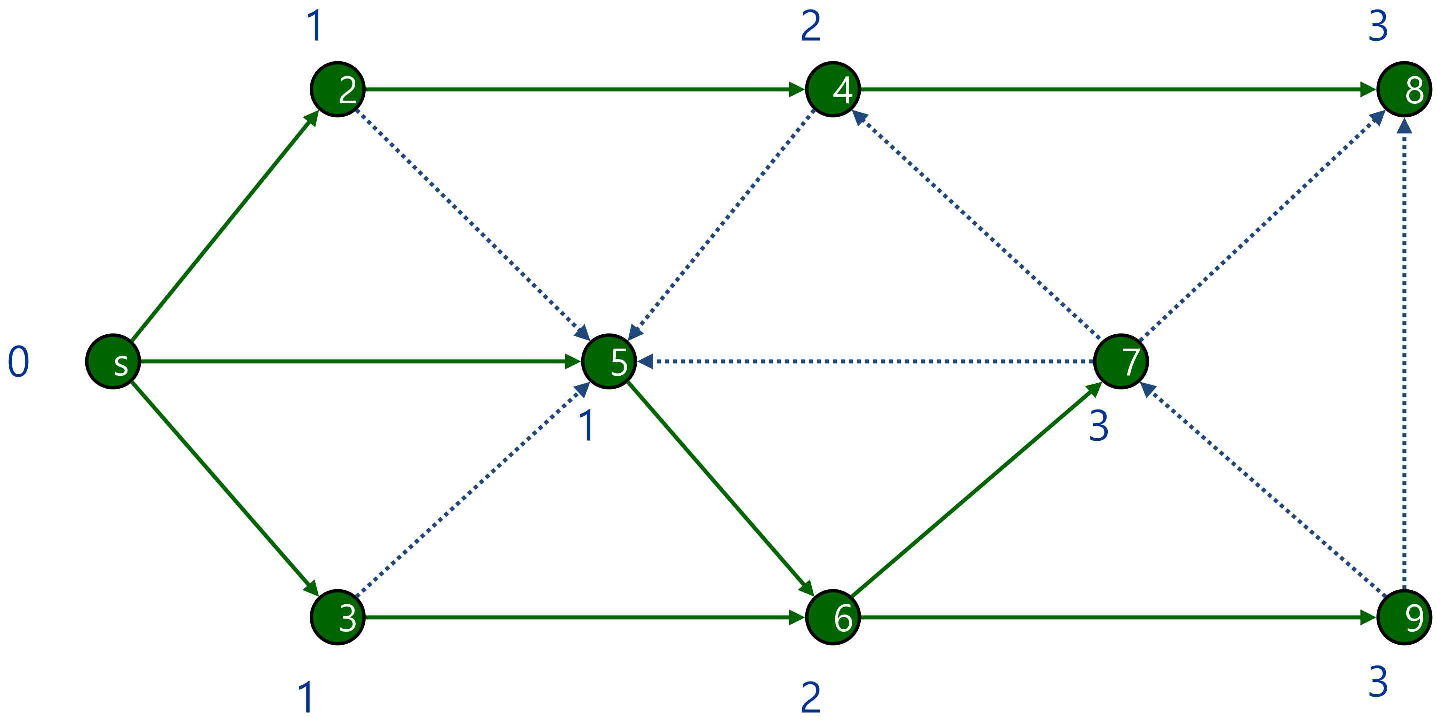
Discovered

Top of queue

Finished

Queue: 9

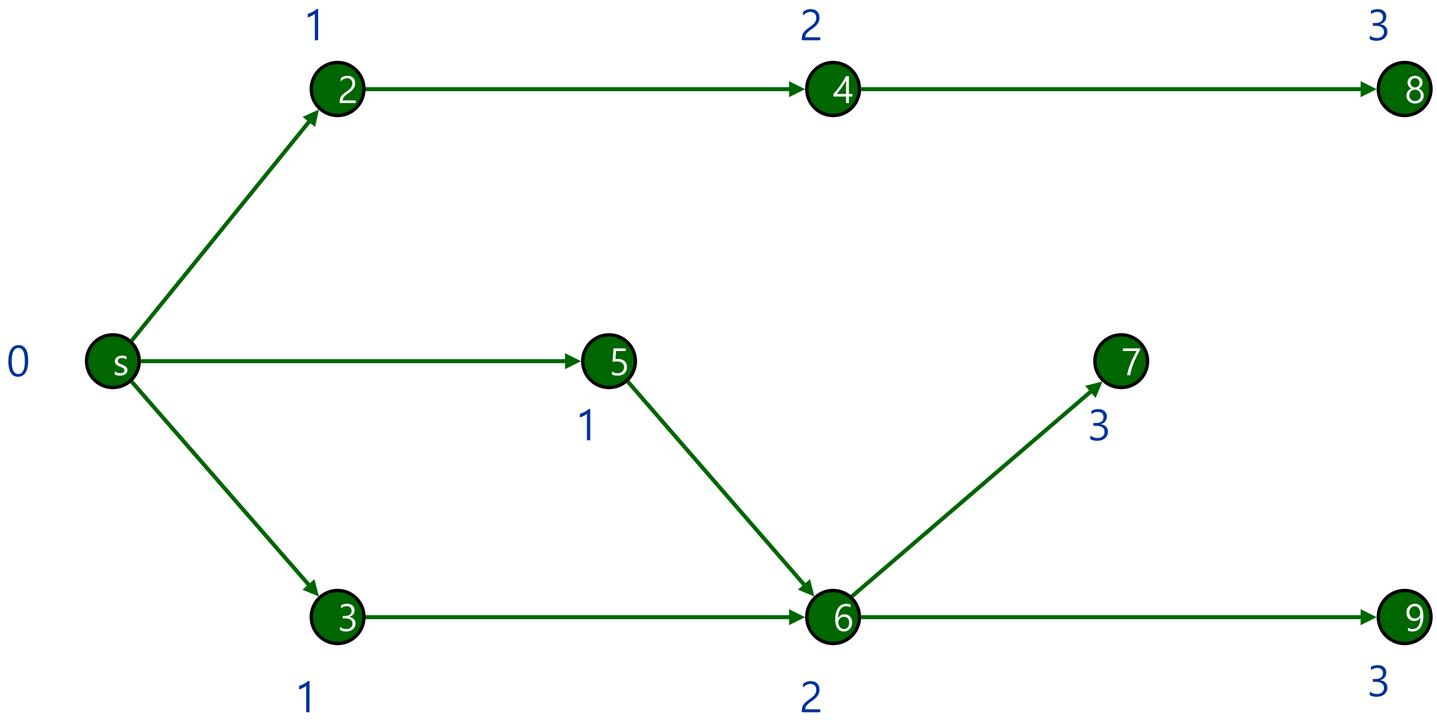
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue:

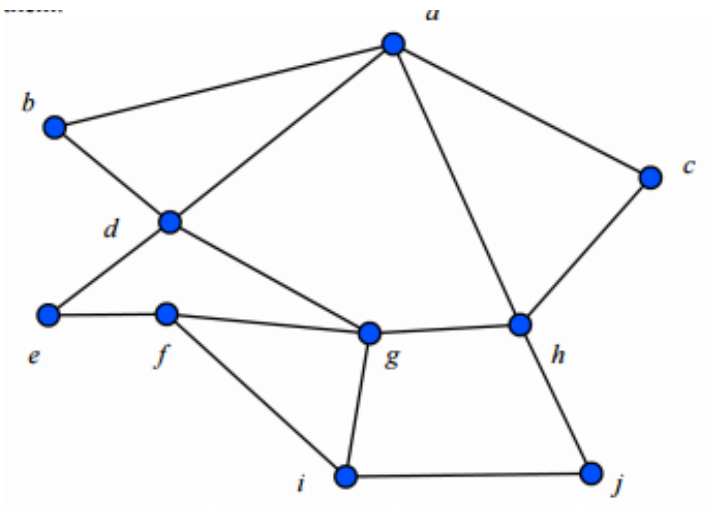
Breadth First Search



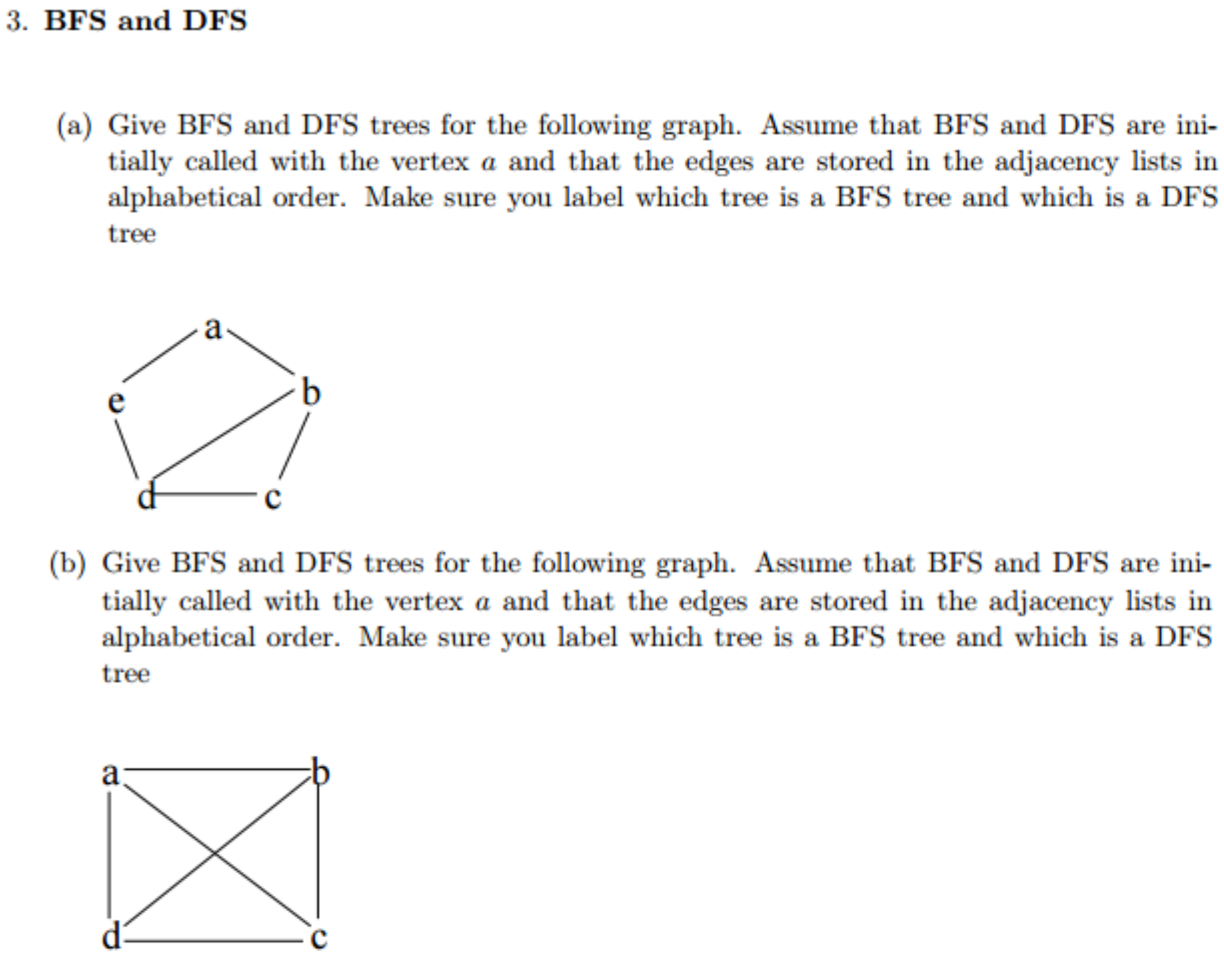
Level Graph

Psu.edu

- Do DFS and BFS starting with ‘a’



www.cs.unm.edu



3. BFS and DFS

(a) Give BFS and DFS trees for the following graph. Assume that BFS and DFS are initially called with the vertex a and that the edges are stored in the adjacency lists in alphabetical order. Make sure you label which tree is a BFS tree and which is a DFS tree

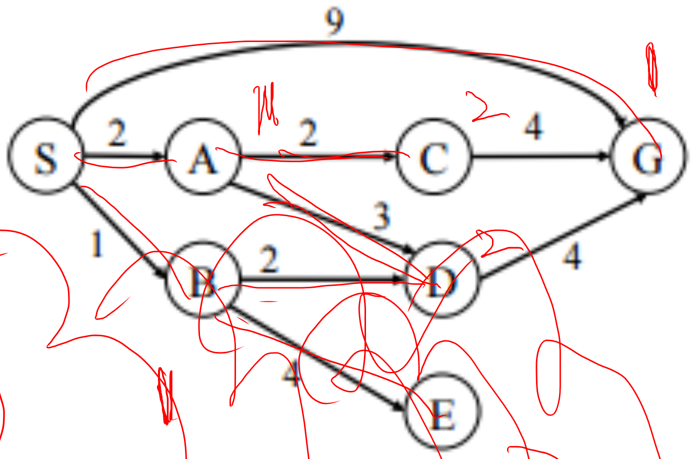
(b) Give BFS and DFS trees for the following graph. Assume that BFS and DFS are initially called with the vertex a and that the edges are stored in the adjacency lists in alphabetical order. Make sure you label which tree is a BFS tree and which is a DFS tree

Solution: The edge set $\{(a,b),(b,c),(c,d)(d,e)\}$ *is the DFS tree. The edge set* $\{(a,b),(b,c),(b,d)(d,e)\}$ *is the BFS tree*

Solution: The edge set $\{(a,b),(a,c),(a,d)\}$ *is a BFS tree. The edge set* $\{(a,b),(b,c),(c,d)\}$ *is a DFS tree.*

Eecs.Berkeley.edu

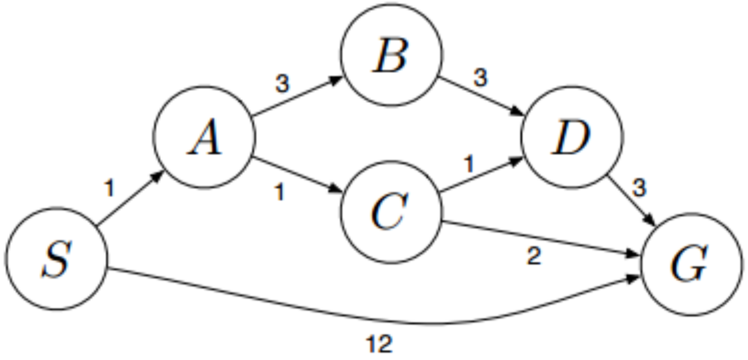
- Do DFS and BFS starting with s



A	✓
B	✓
C	✓
D	✓
E	✓
G	✓
S	✓

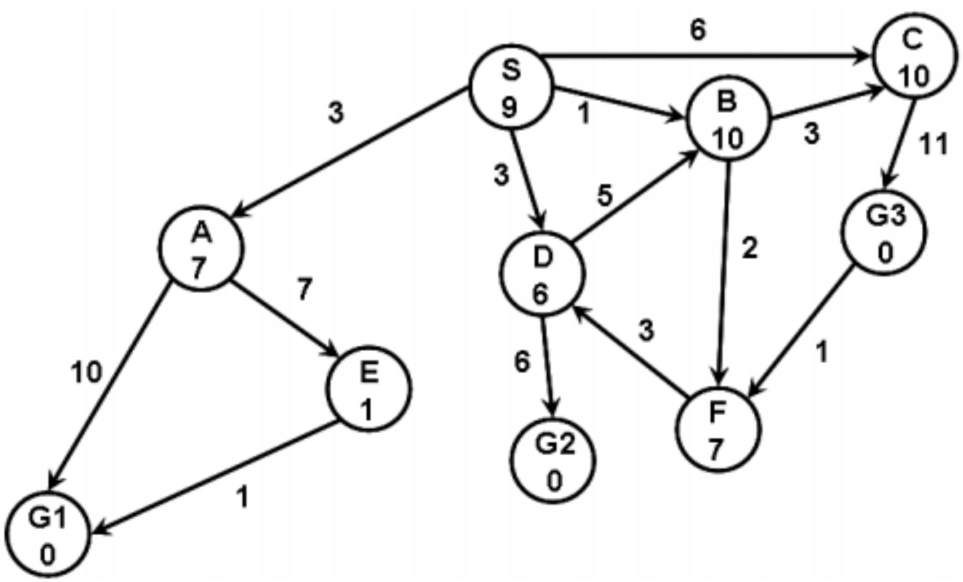
S A B G, C D E

- Do DFS and BFS starting with s



www.cs.cmu.edu

- Do DFS and BFS starting with s



4. For the graph below, do:

- **(a)** Do a DFS starting at vertex *a* (and using the "alphabetical rule"). Shade the edges of the resulting DFS spanning tree and show all vertex labels. Give the order vertices are added to T and P.
- **(b)** Do a BFS starting at vertex *a* (and using the "alphabetical rule"). Shade the edges of the resulting BFS spanning tree and show all vertex labels. Give the order vertices are added to the queue T.