Single-source shortest path

Directed Graph

A directed graph G, also known as a digraph, is a graph in which every edge has a direction assigned to it. An edge of a directed graph is given as an ordered pair (u, v) of nodes in G. For an edge (u, v),

- The edge begins at u and terminates at v.
- u is known as the origin or initial point of edge e. Correspondingly, v is known as the destination or terminal point of edge e.
- u is the predecessor of v. Correspondingly, v is the successor of u.
- Nodes u and v are adjacent to each other.

Terminologies

- **Out-degree of a node:** The out-degree of a node u, written as outdeg(u), is the number of edges that originate at u.
- **In-degree of a node:** The in-degree of a node u, written as indeg(u), is the number of edges that terminate at u.
- **Degree of a node:** The degree of a node, written as deg(u), is equal to the sum of in-degree and out-degree of that node. Therefore, deg(u) = indeg(u) + outdeg(u).
- **Isolated vertex:** A vertex with degree zero.
- **Pendant vertex:** (also known as leaf vertex) A vertex with degree one.
- **Cut vertex:** A vertex which when deleted would disconnect the remaining graph.
- **Source:** A node u is known as a source if it has a positive out-degree but a zero in-degree.
- **Sink:** A node u is known as a sink if it has a positive in-degree but a zero out-degree.
- **Reachability:** A node v is said to be reachable from node u, if and only if there exists a (directed) path from node u to node v.
- **Strongly connected directed graph:** A digraph is said to be strongly connected if and only if there exists a path between every pair of nodes in G. That is, if there is a path from node u to v, then there must be a path from node v to u.

Shortest Path

Given a weighted directed graph, one common problem is finding the shortest path between two given vertices

◦ Recall that in a weighted graph, the *length* of a path is the sum of the weights of each of the edges in that path

Given the graph, suppose we wish to find the shortest path from vertex 1 to vertex 13
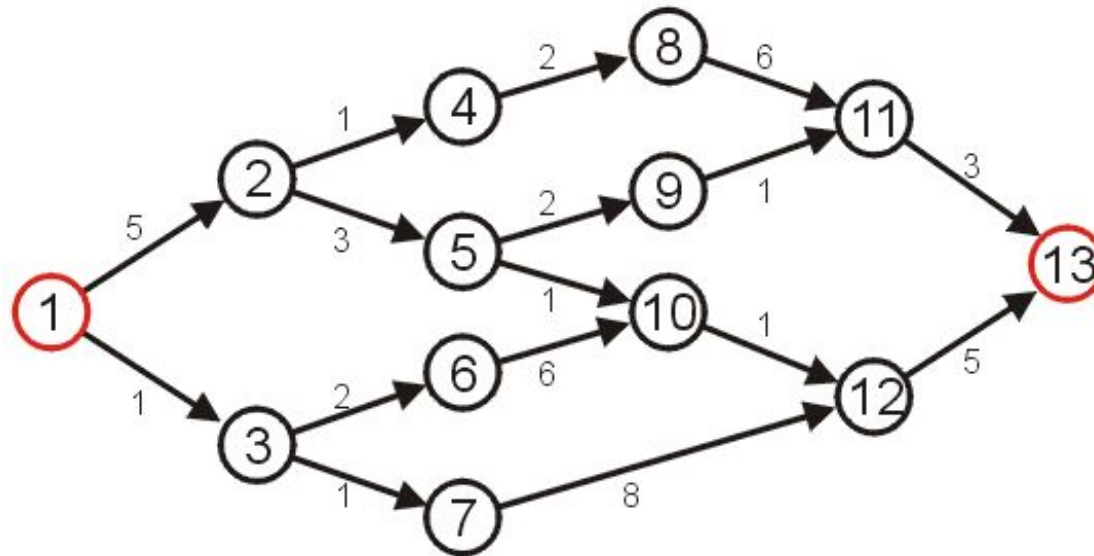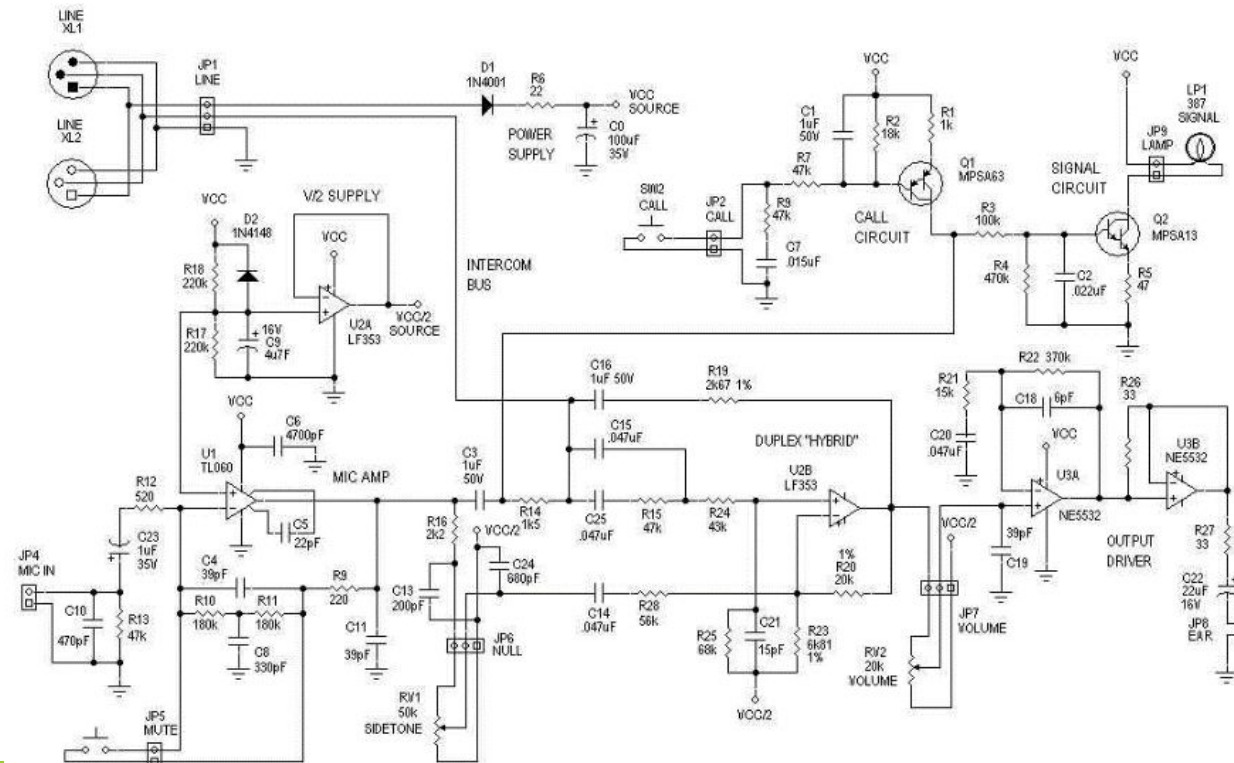
Shortest Path

After some consideration, we may determine that the shortest path is as follows, with length 14
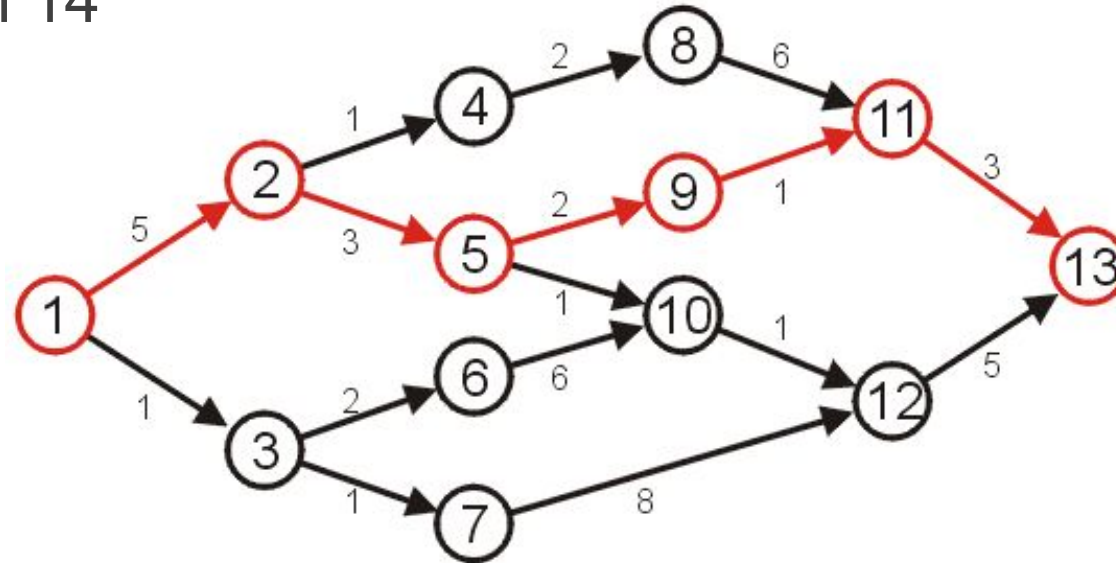
Other paths exists, but they are longer

Applications

one application is circuit design: the time it takes for a change in input to affect an output depends on the shortest path

Applications

The Internet is a collection of interconnected computer networks
◦ Information is passed through *packets*

Packets are passed from the source, through routers, to their destination

Routers are connected to either:
◦ individual computers, or
◦ other routers

These may be represented as graphs

Applications

A visualization of the graph of the routers and their various connections through a portion of the Internet
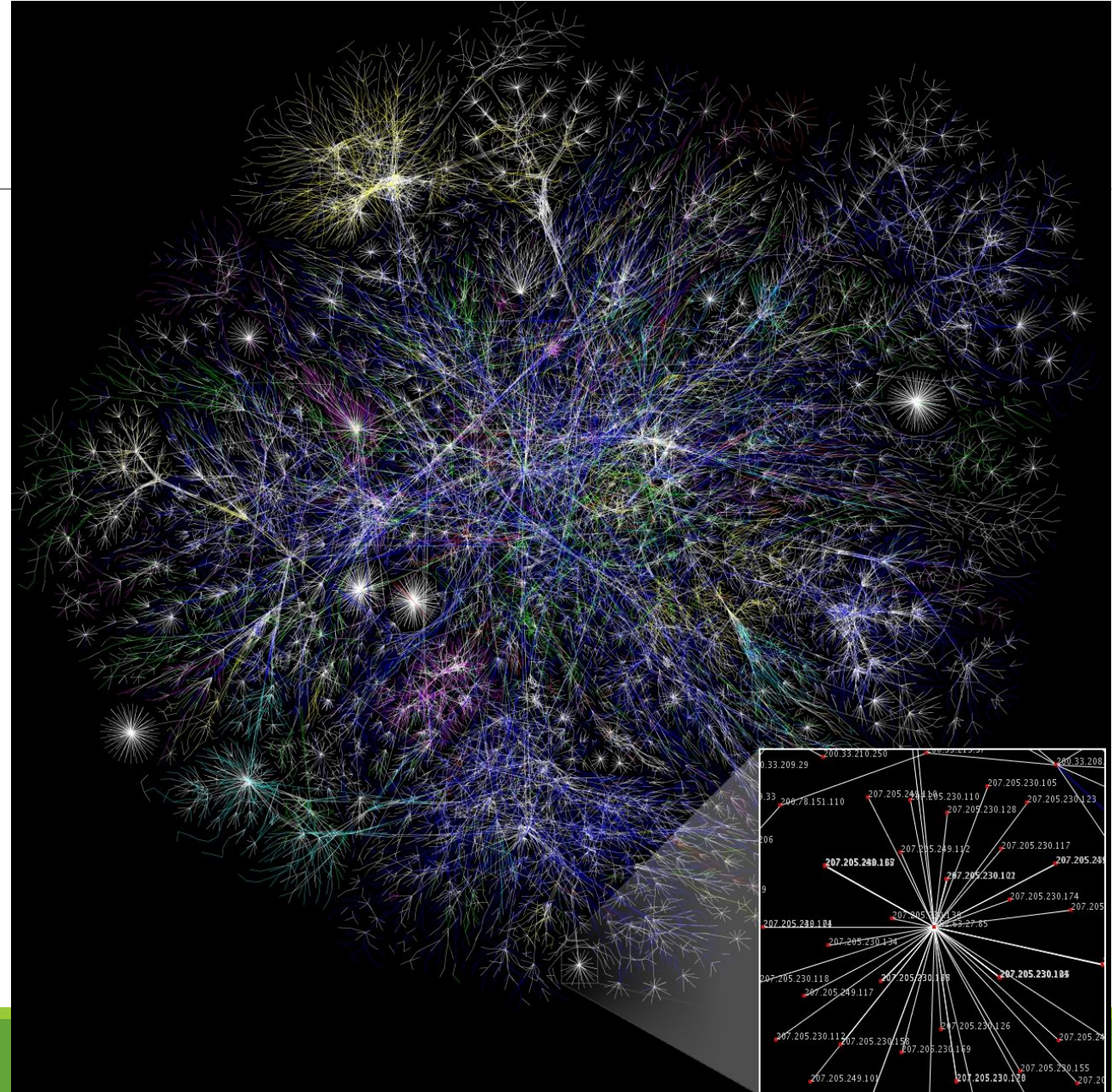
Applications

In software engineering, one obvious problem is finding the shortest route between to points on a map

◦ Shortest path, however, need not refer to distance...
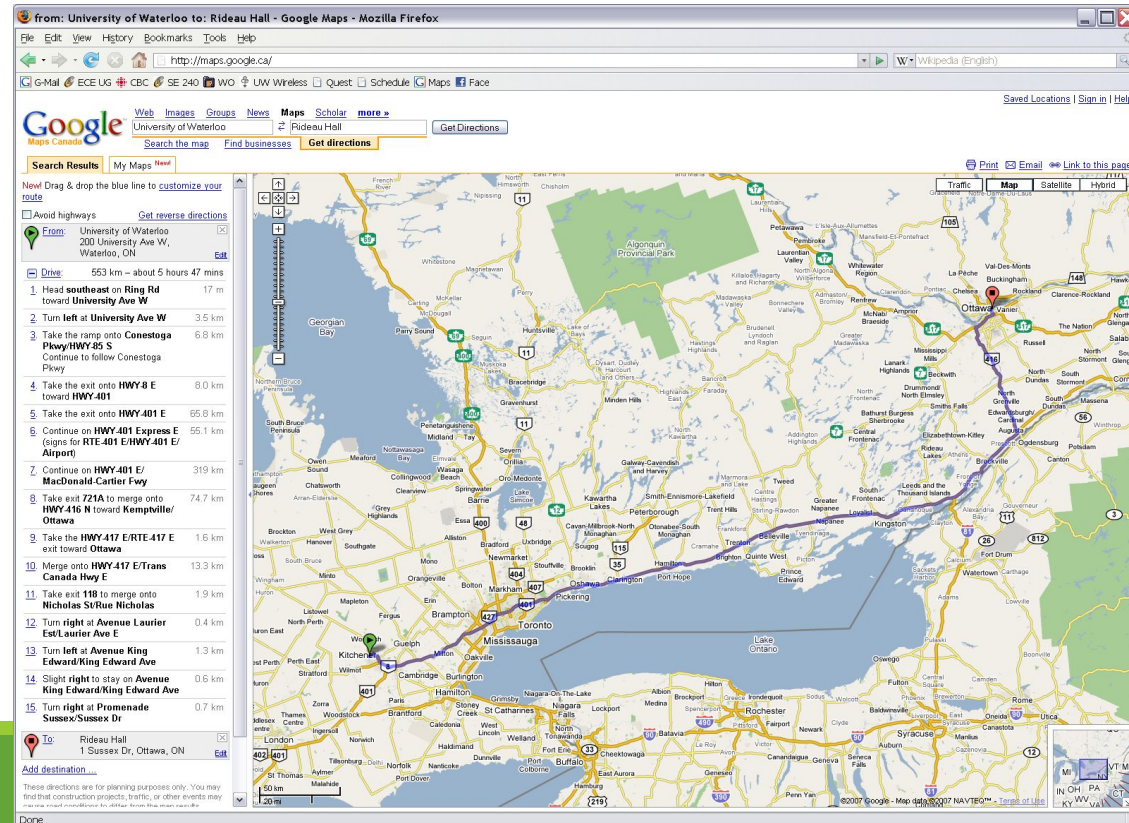
Applications

The *shortest path* using distance as a metric is obvious, however, a driver may be more interested in minimizing time

For example, using the 407 may be preferable to using the 401 during rush hour, however, there is an added cost

A company will be interested in minimizing the cost which includes the following factors:
- salary of the truck driver (overtime?)
- possible tolls and administrative costs
- bonuses for being early
- penalties for being late
- cost of fuel

Shortest Path

The goal of this algorithm will be to find the shortest path and its length

We will make the assumption that the weights on all edges is a positive number

- ◦ Clearly, if we have negative vertices, it may be possible to end up in a cycle whereby each pass through the cycle decreases the total *length*
- ◦ Thus, a shortest length would be undefined for such a graph
- ◦ Consider the shortest path from vertex 1 to 4...
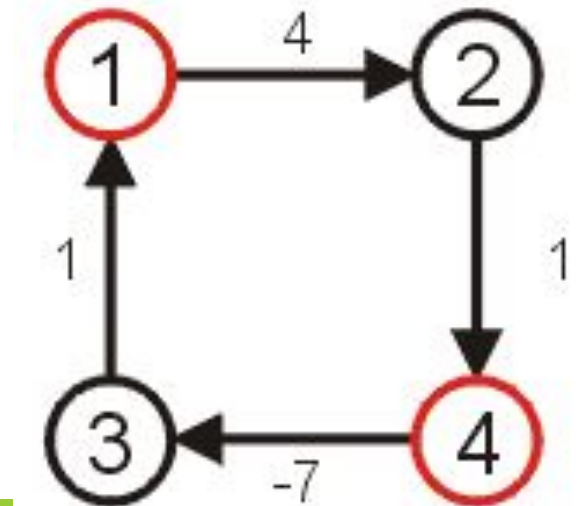
Shortest Path

Consider the following graph
  ◦ All edges have positive weight
  ◦ There exists cycles—it is not a DAG
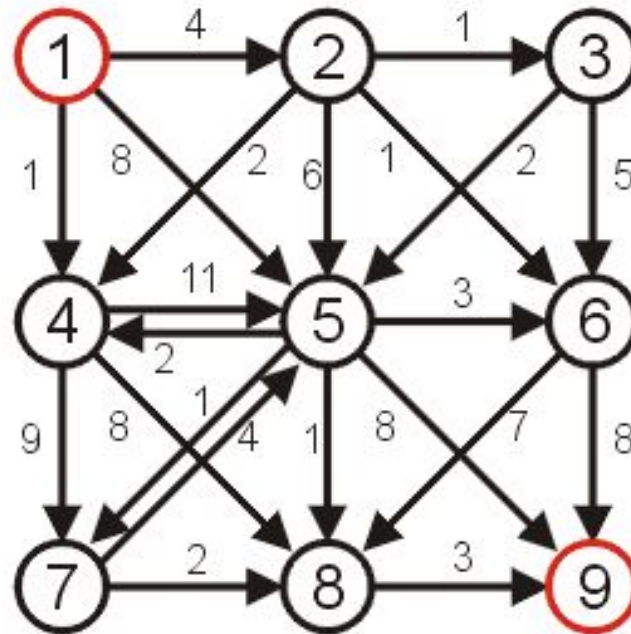
Algorithms

Algorithms for finding the shortest path include:
- Dijkstra's algorithm
- A* search algorithm
- Bellman-Ford algorithm
- BFS

Breadth-First Search

Breadth-first search (BFS) is a general technique for traversing a graph

A BFS traversal of a graph G

◦ Visits all the vertices and edges of G

◦ Determines whether G is connected

◦ Computes the connected components of G

BFS on a graph with n vertices and m edges takes $O(n + m)$ time

BFS can be further extended to solve other graph problems

◦ Find and report a path with the minimum number of edges between two given vertices

◦ Find a simple cycle, if there is one

BFS Algorithm

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

Algorithm *BFS*(*G, s*)

 L_0 ← new empty sequence

 $L_0.insertLast(s)$

 setLabel(*s, VISITED*)

 $i ← 0$

 while ¬$L_i.isEmpty$()

  L_{i+1} ← new empty sequence

  for all $v ∈ L_i.elements$()

   for all $e ∈ G.incidentEdges(v)$

    if *getLabel*(*e*) = *UNEXPLORED*

     $w ← opposite(v,e)$

     if *getLabel*(*w*) = *UNEXPLORED*

      setLabel(*e, DISCOVERY*)

      setLabel(*w, VISITED*)

      $L_{i+1}.insertLast(w)$

    else

     setLabel(*e, CROSS*)

 $i ← i +1$

Breadth First Search
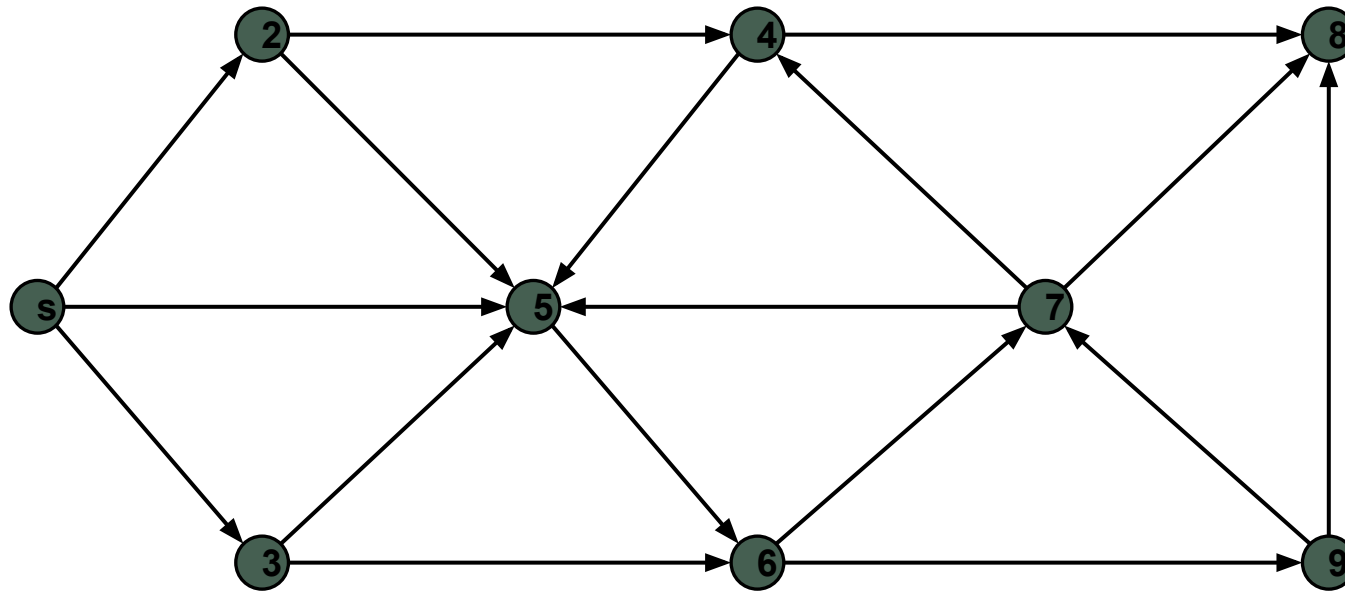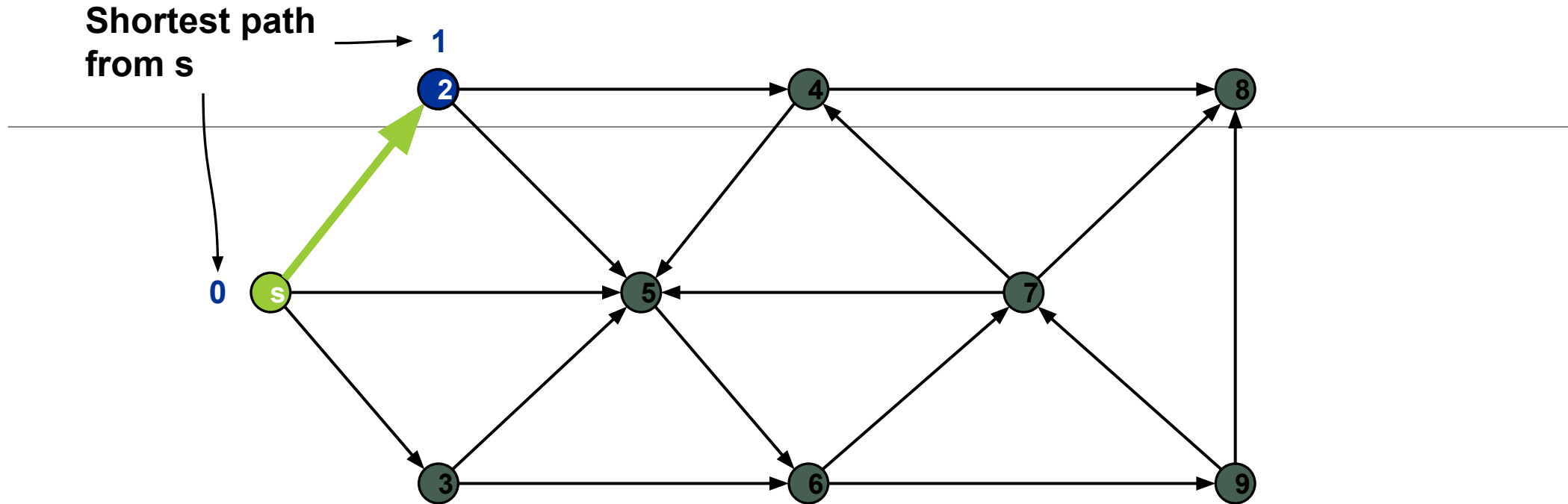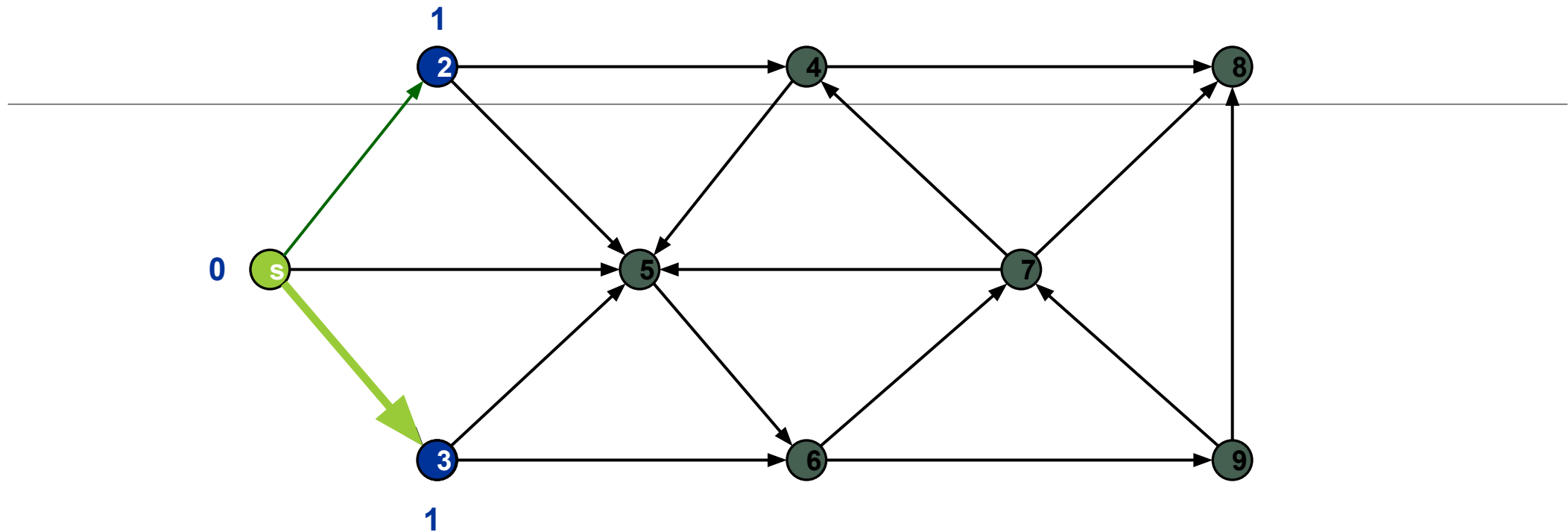
Breadth First Search

Breadth First Search

Breadth First Search
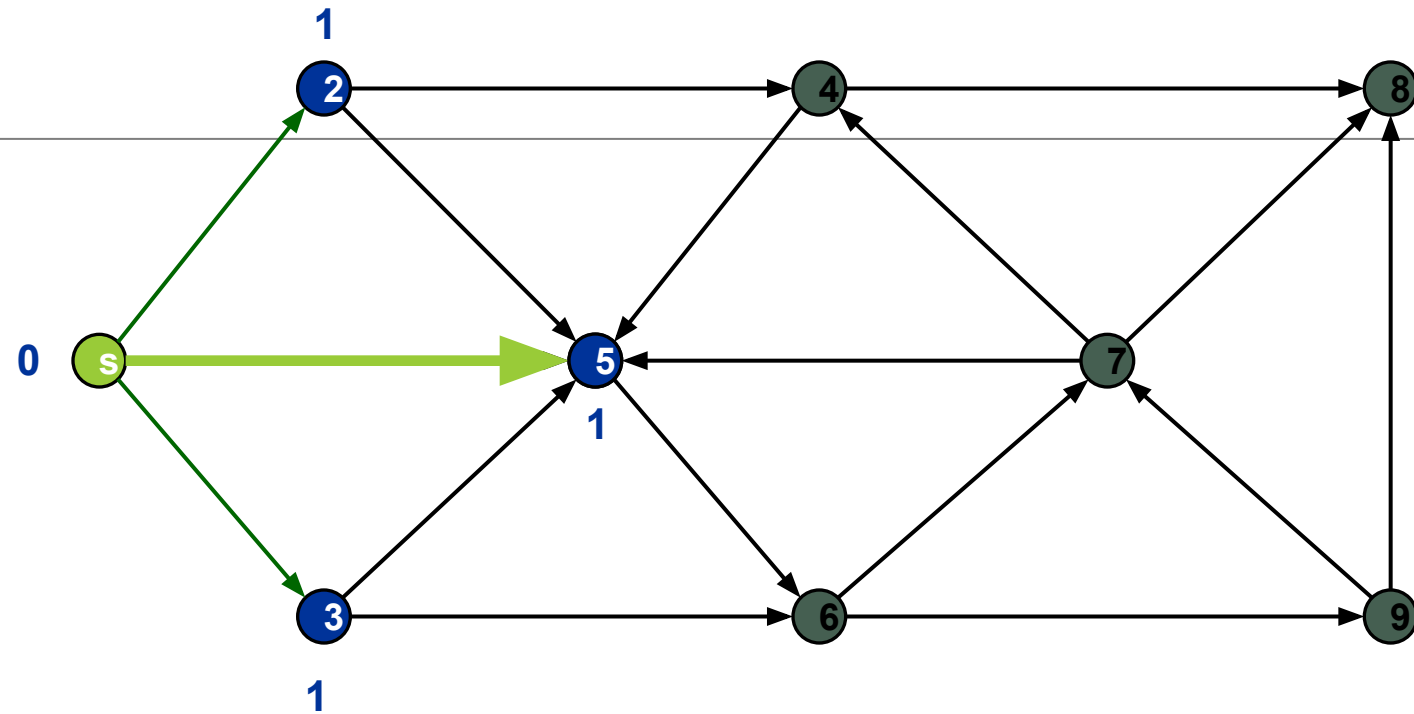
Undiscovered
Discovered
Top of queue
Finished

Queue: s 2 3

19

Breadth First Search

Breadth First Search
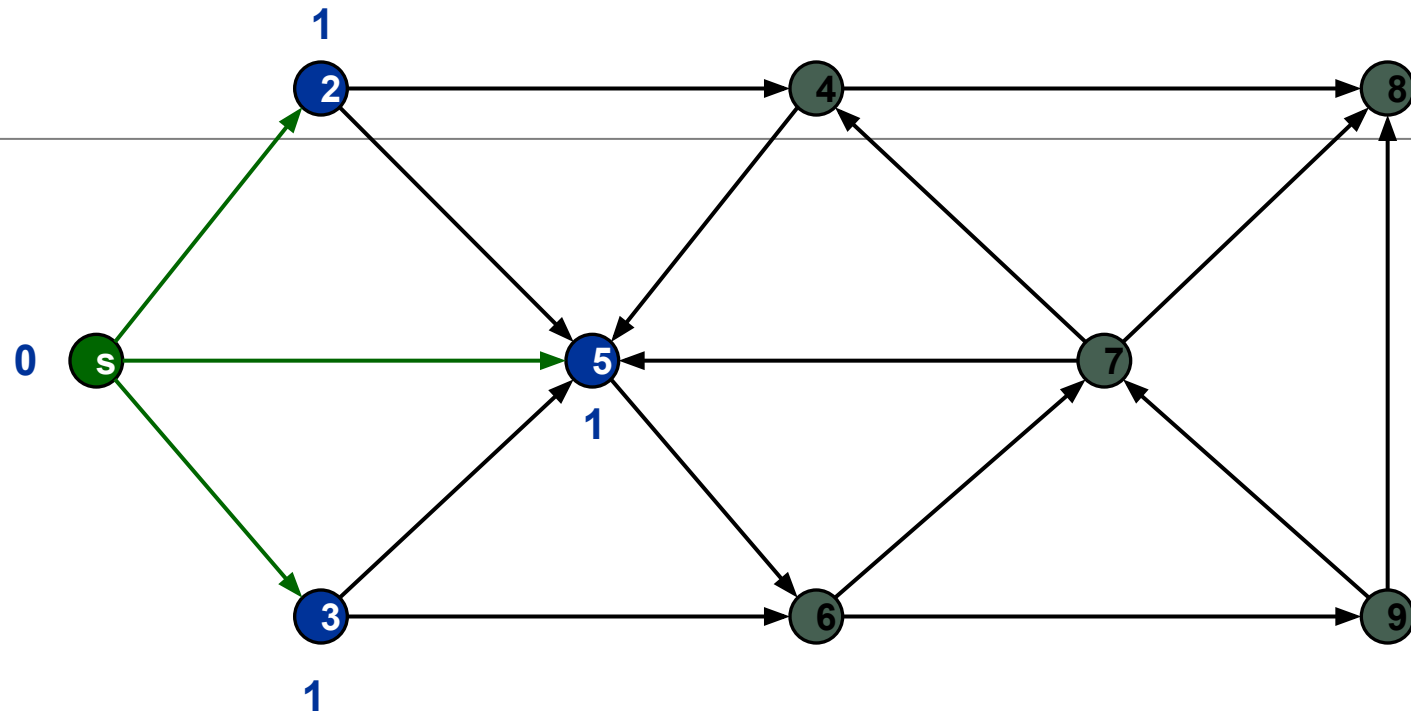
Undiscovered
Discovered
Top of queue
Finished

Queue: 2 3 5

Breadth First Search

Breadth First Search
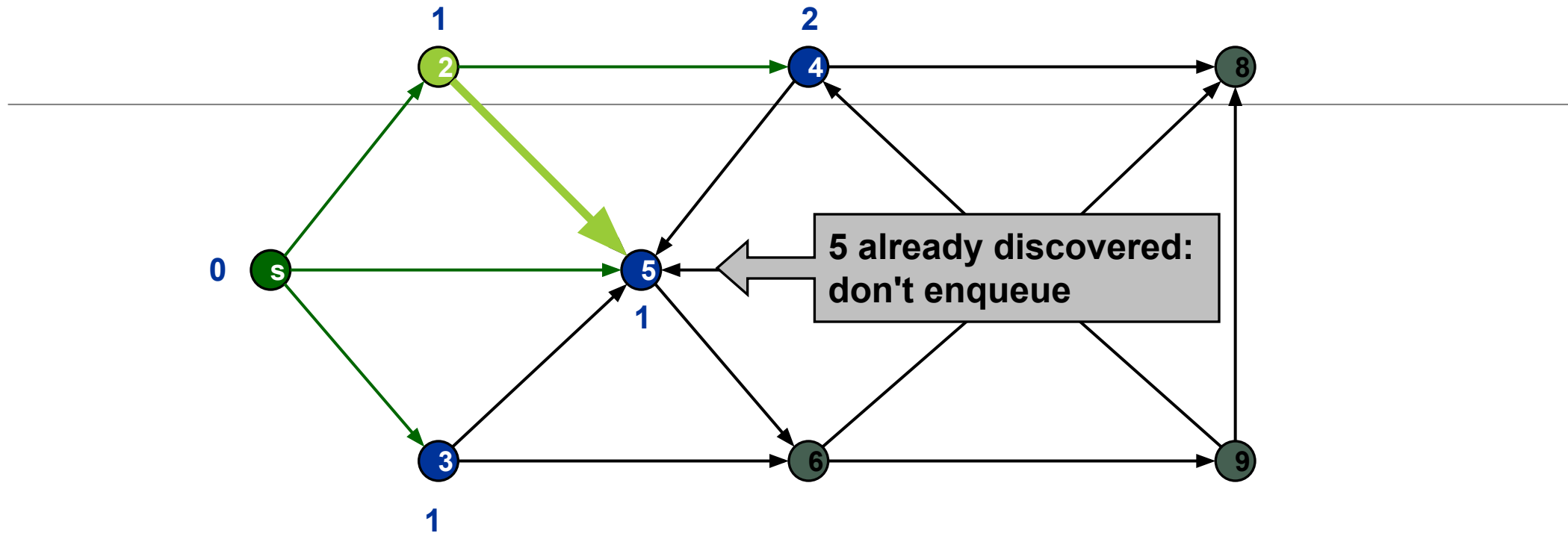
Undiscovered
Discovered
Top of queue
Finished

Queue: 2 3 5 4

Breadth First Search

Breadth First Search

Breadth First Search

Undiscovered
Discovered
Top of queue
Finished

Queue: 3 5 4 6

26

Breadth First Search

27

Breadth First Search

Undiscovered
Discovered
Top of queue
Finished

Queue: 5 4 6
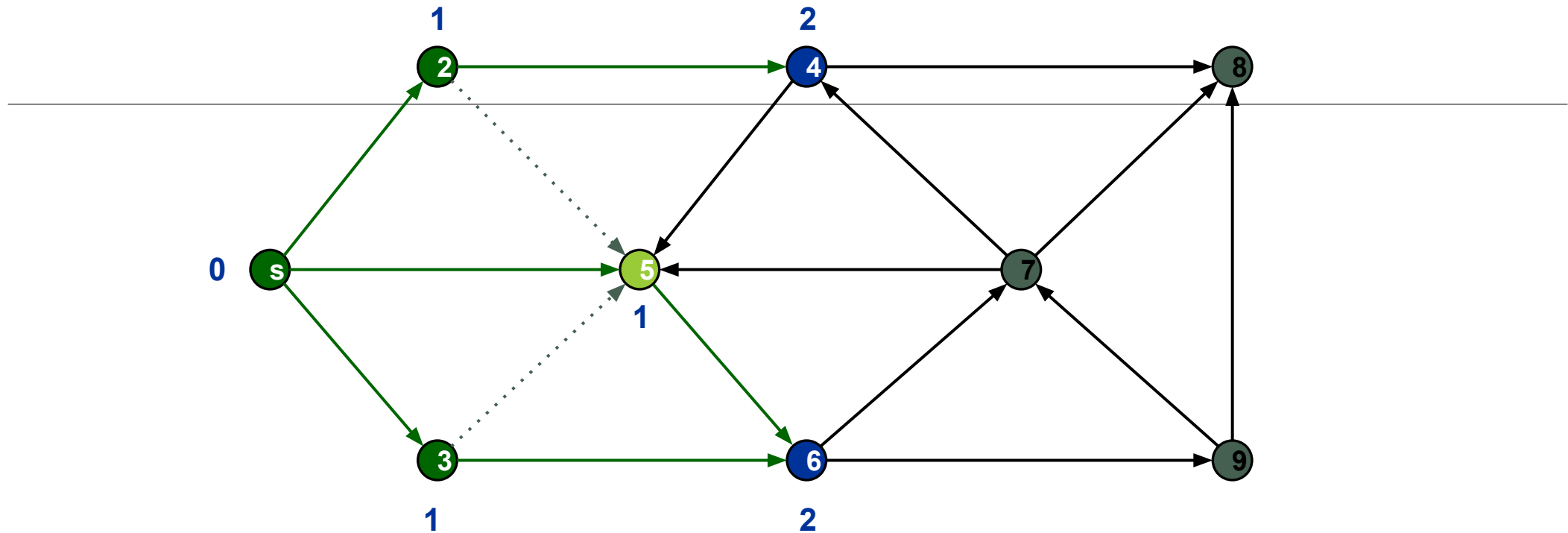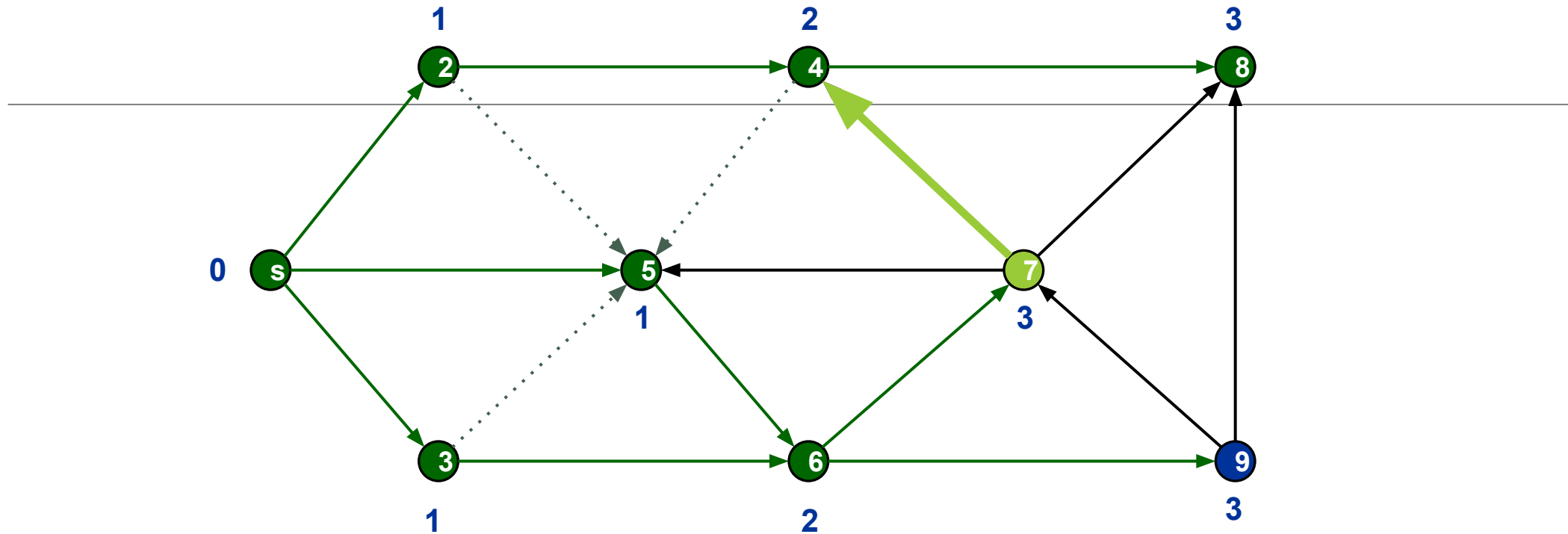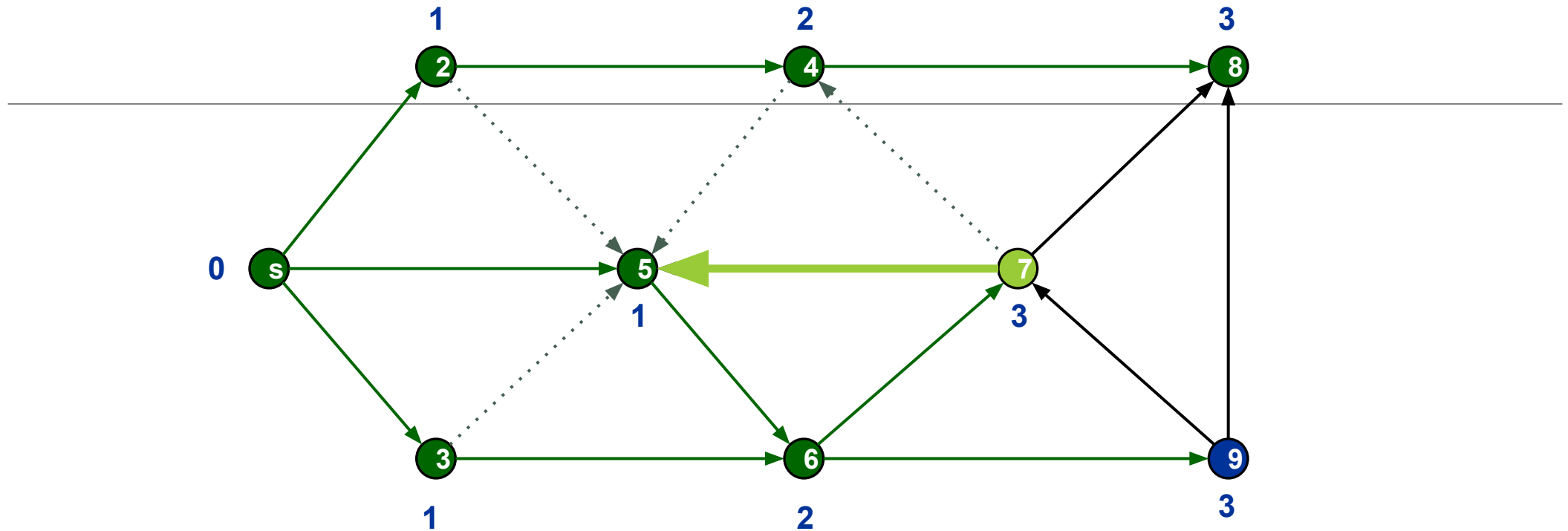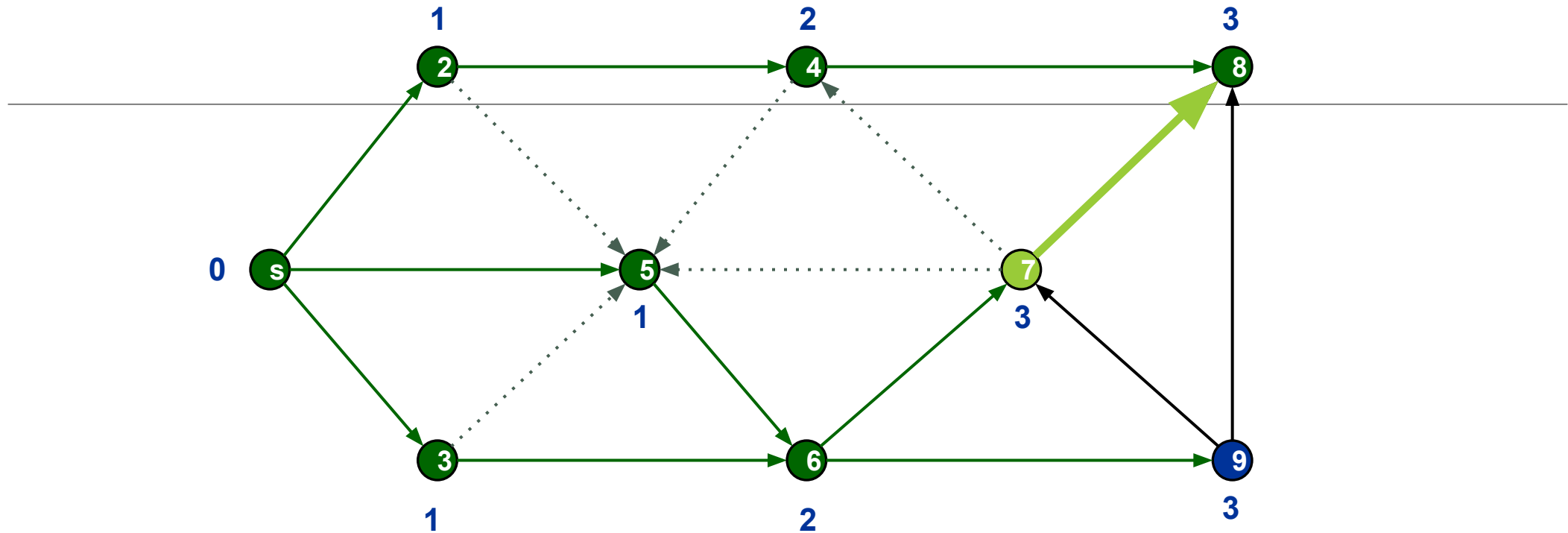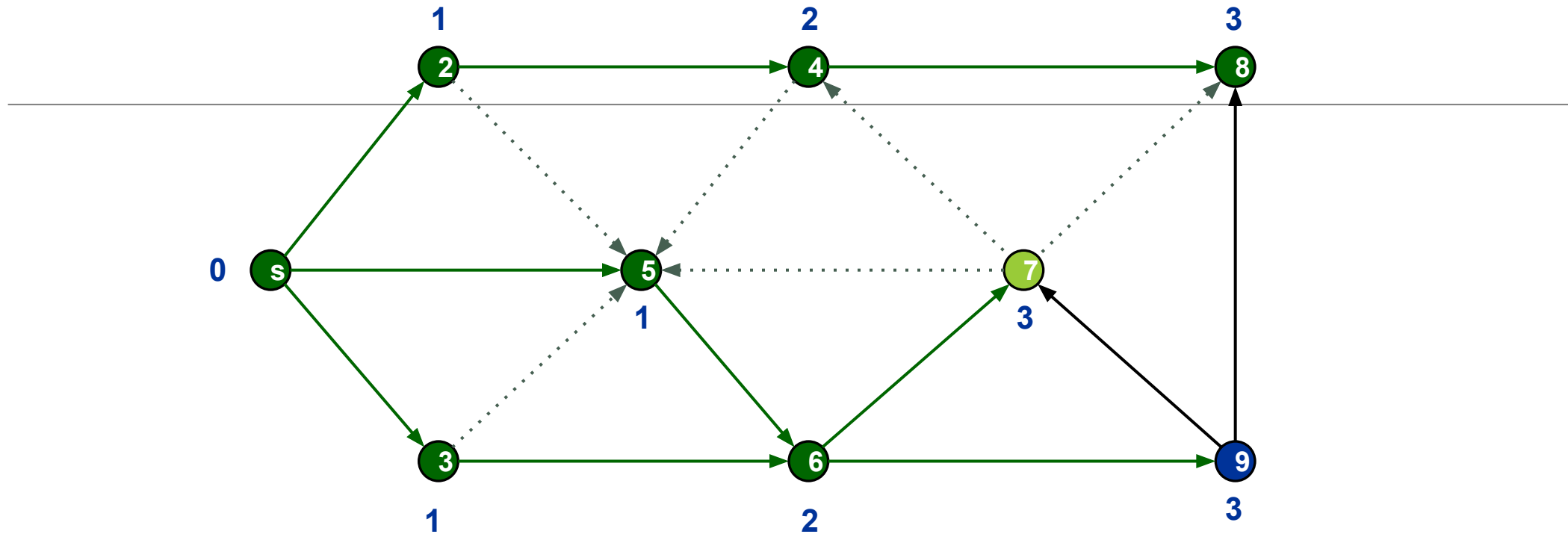
28

Breadth First Search

Queue: 4 6

Breadth First Search
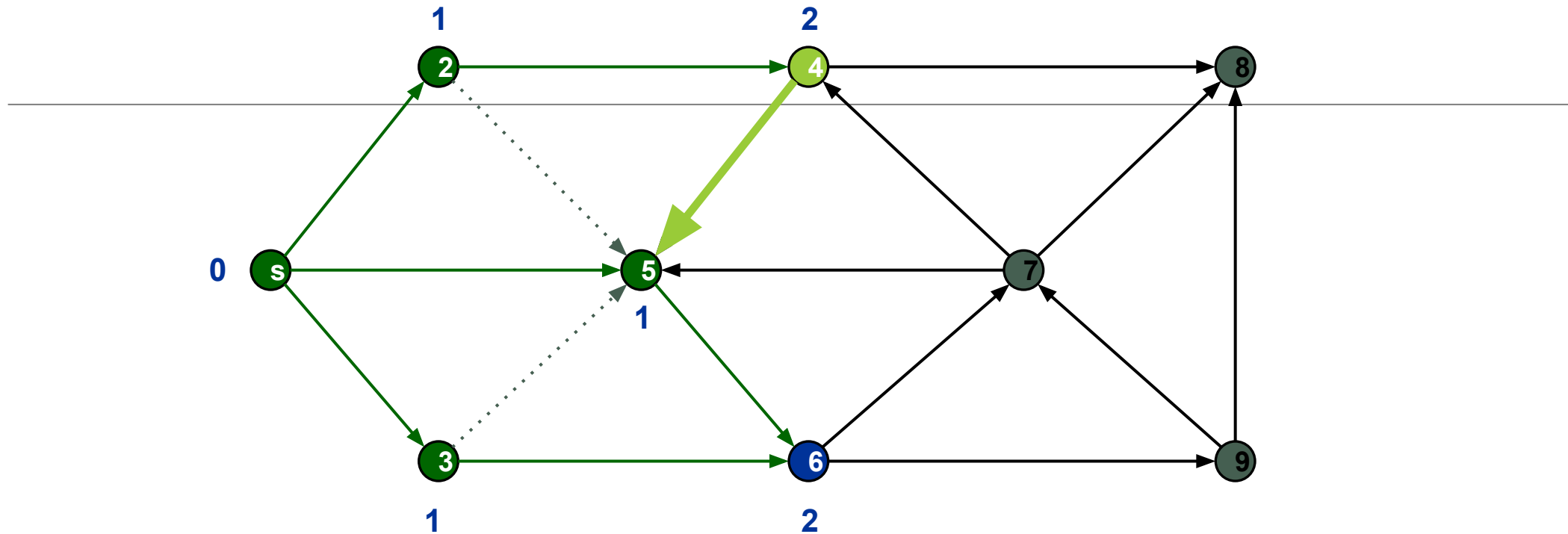
Undiscovered
Discovered
Top of queue
Finished

Queue: 4 6

Breadth First Search
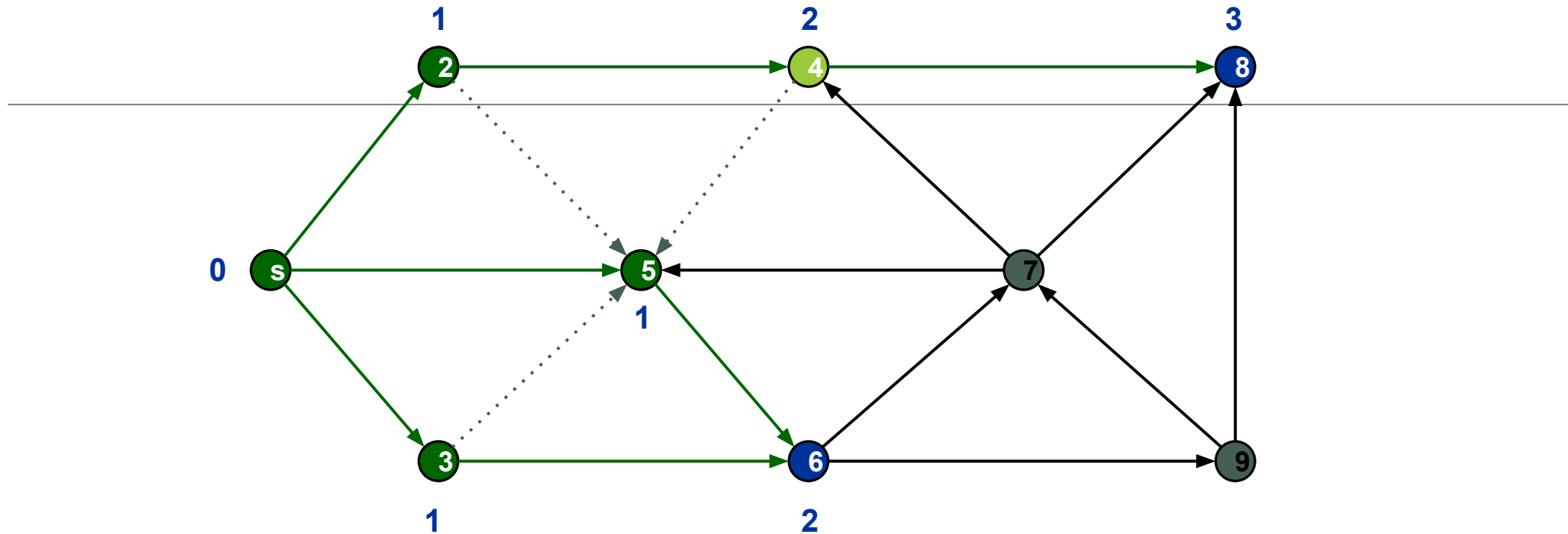
Undiscovered
Discovered
Top of queue
Finished

Queue: 4 6 8

Breadth First Search
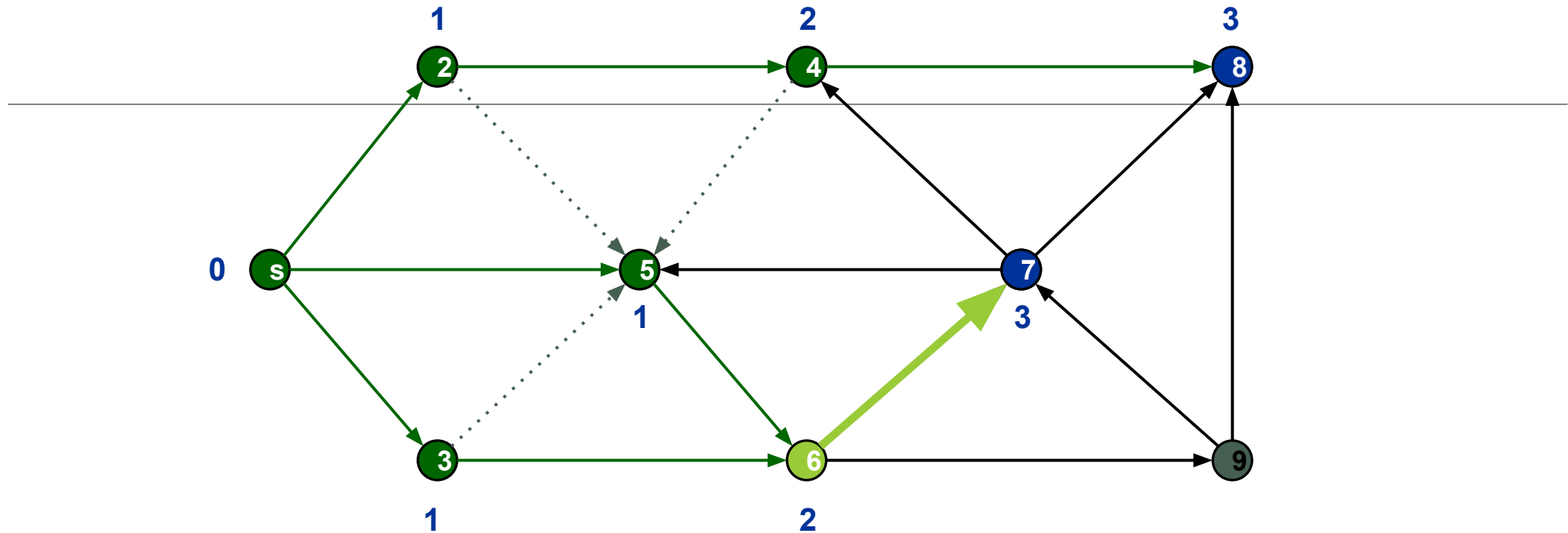
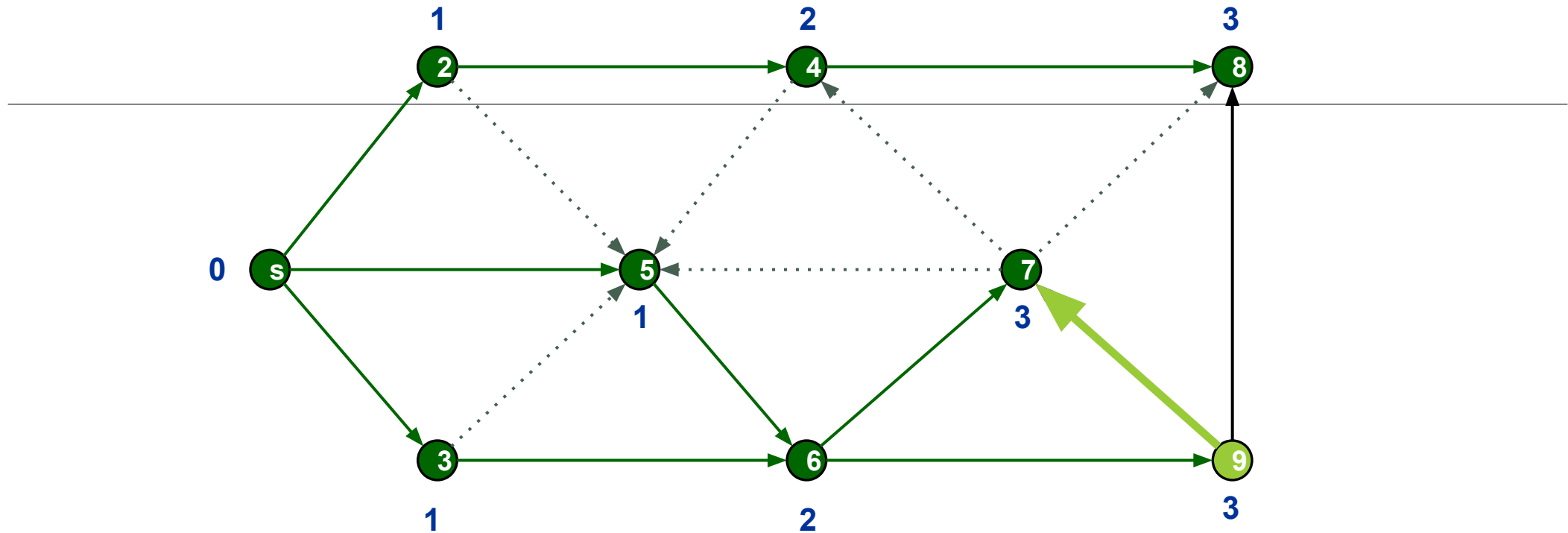

32
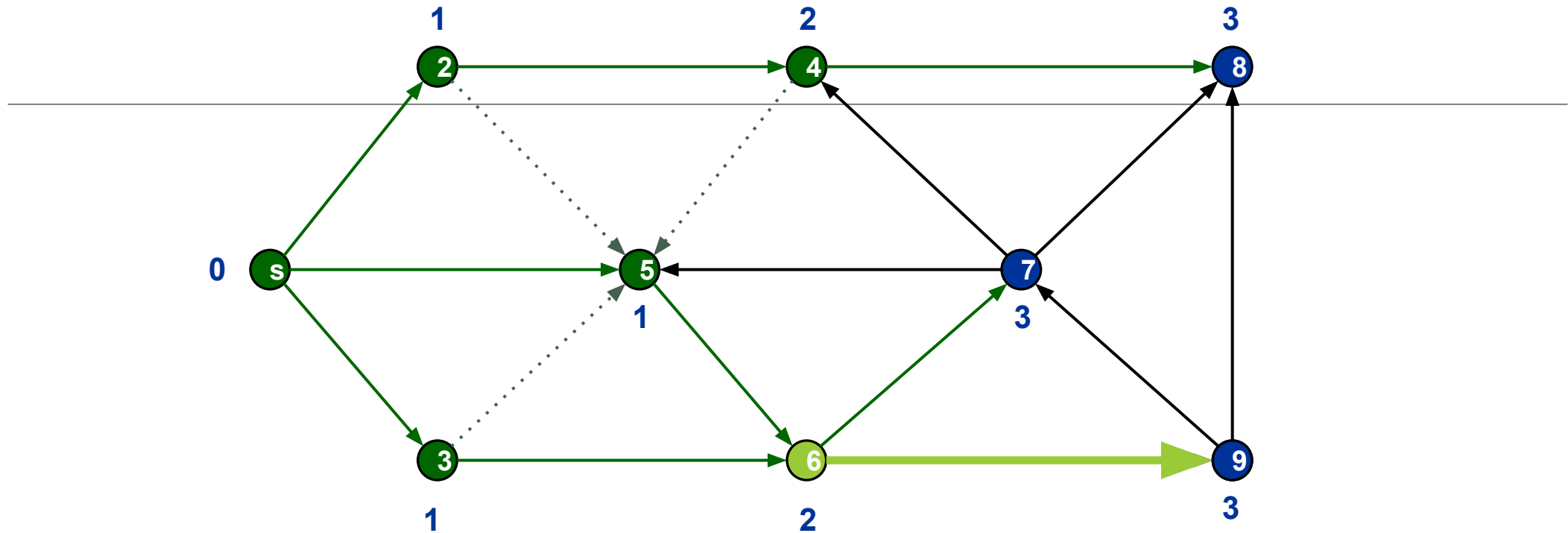
Breadth First Search

Breadth First Search
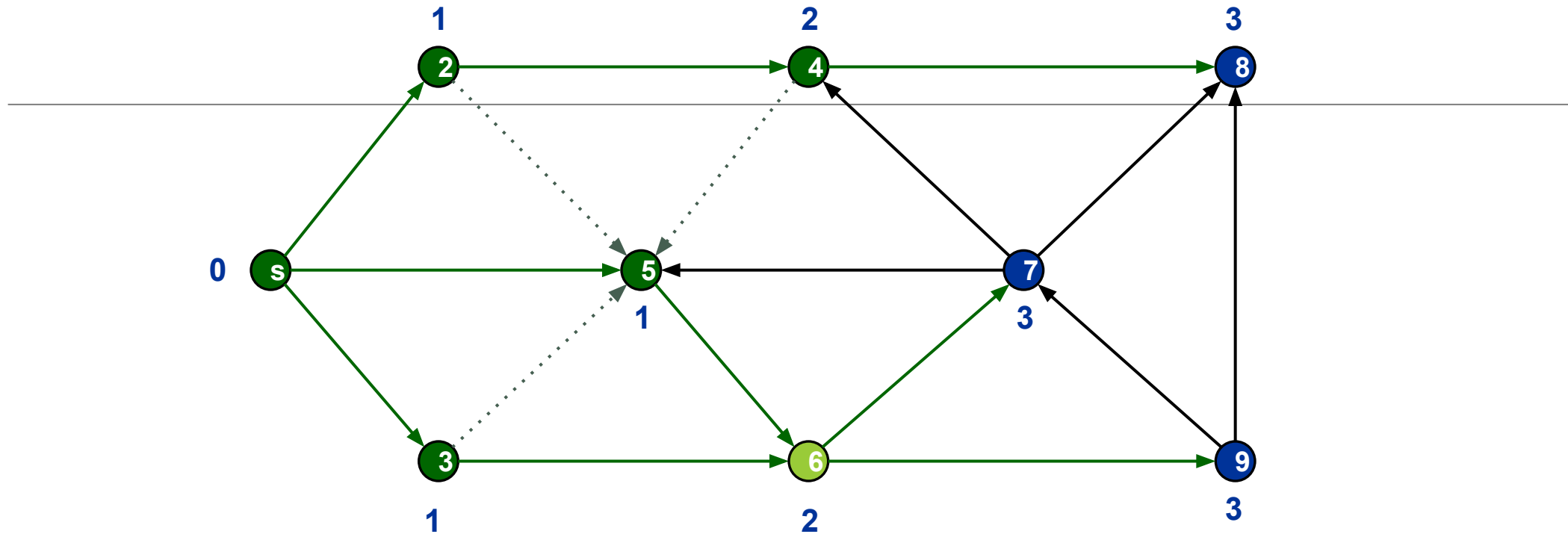
Undiscovered
Discovered
Top of queue
Finished

Queue: 6 8 7 9

Breadth First Search
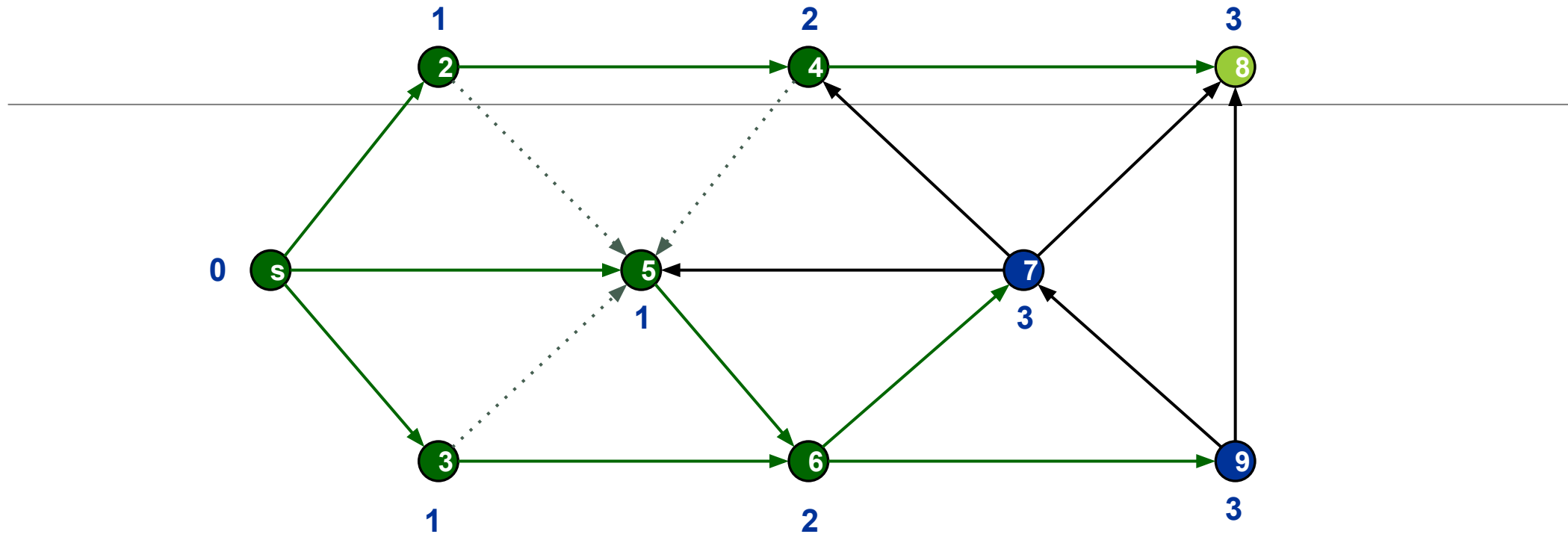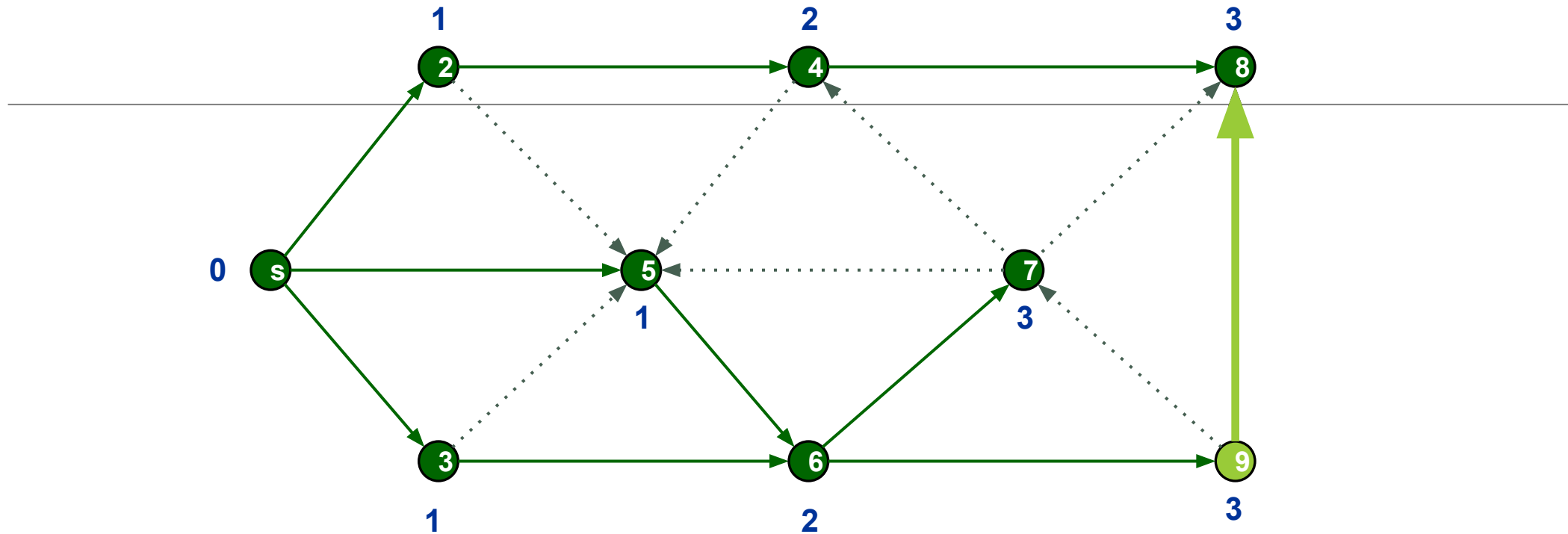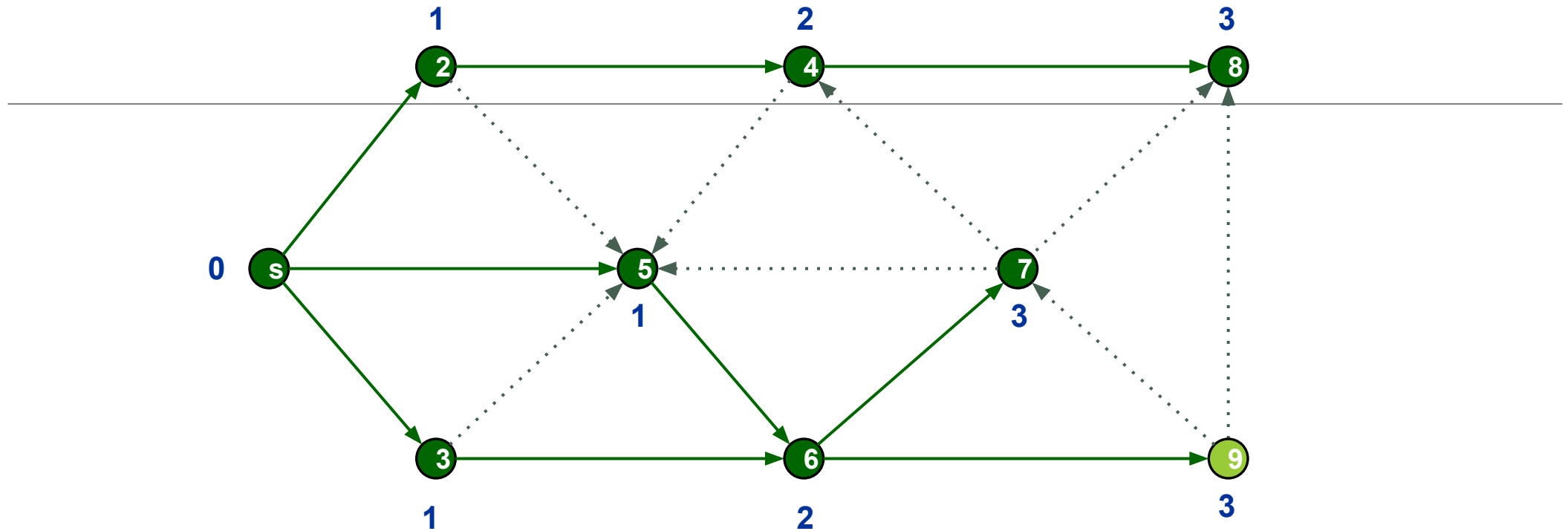

Undiscovered
Discovered
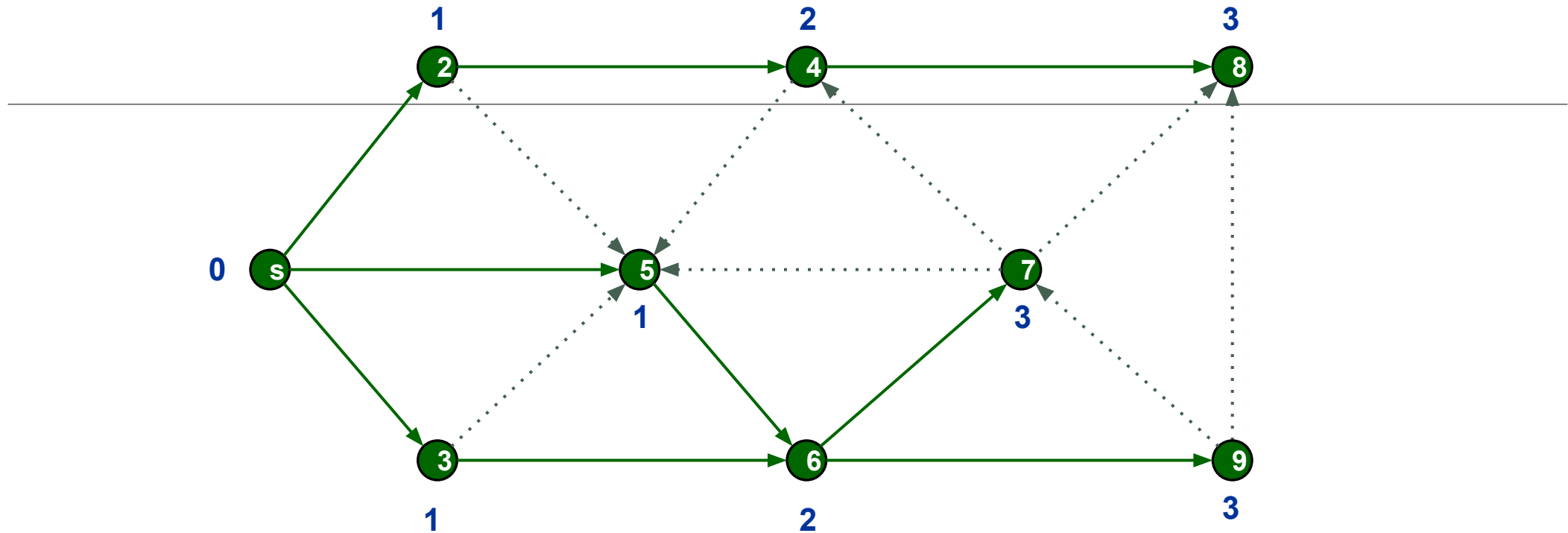Top of queue
Finished

Queue: 8 7 9

35

Breadth First Search

Undiscovered
Discovered
Top of queue
Finished

Queue: 7 9

36

Breadth First Search

Breadth First Search

Undiscovered
Discovered
Top of queue
Finished

Queue: 7 9

Breadth First Search

Undiscovered
Discovered
Top of queue
Finished

Queue: 7 9

39

Breadth First Search

Breadth First Search

Breadth First Search

42

Breadth First Search

43

Breadth First Search
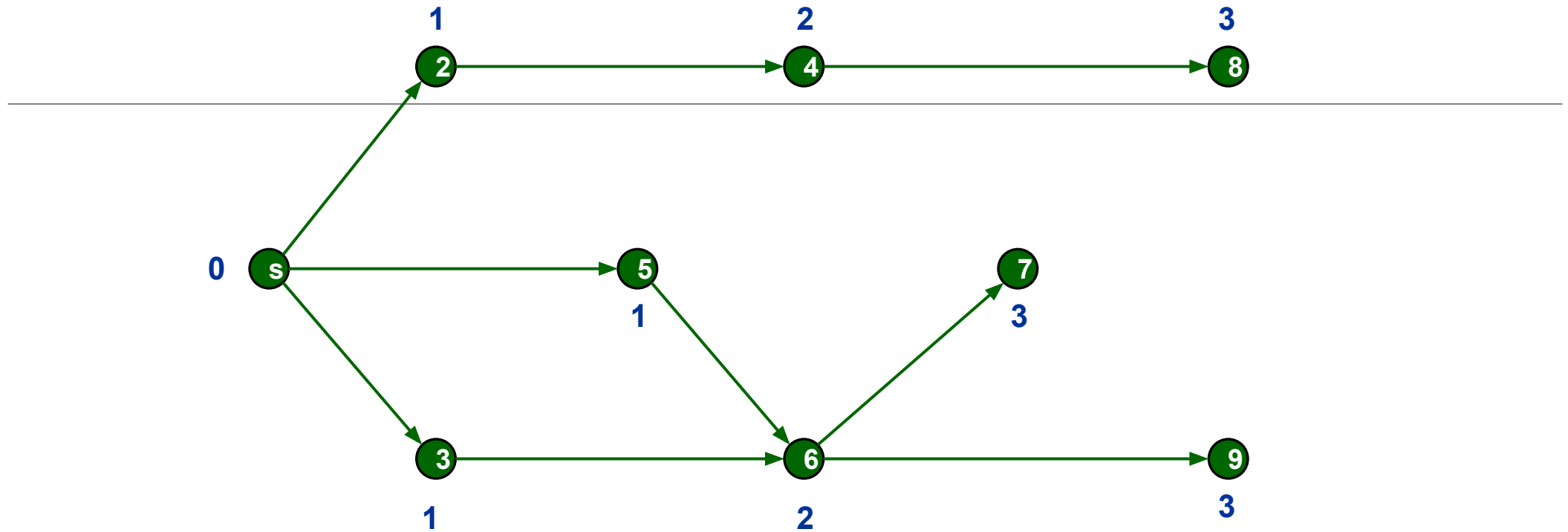
Level Graph

Depth-First Search

Can be used to attempt to visit all nodes of a graph in a systematic manner

Works with directed and undirected graphs

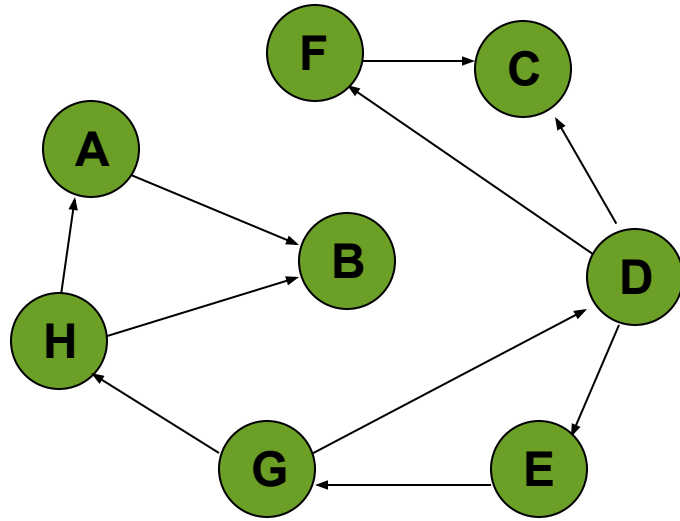Works with weighted and unweighted graphs

<u>depth-first-search</u>
mark vertex as visited
for each adjacent vertex
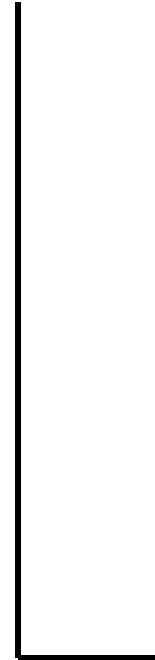if unvisited
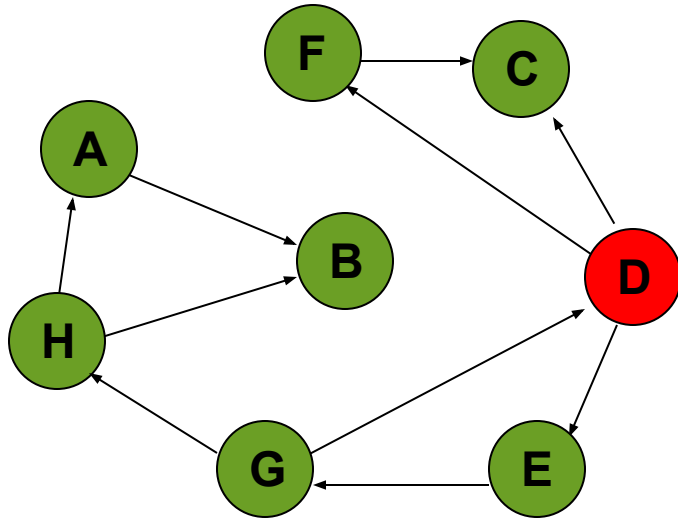do a depth-first search on adjacent vertex

Walk-Through

Visited Array

A	
B	
C	
D	
E	
F	
G	
H	

Stack

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

Visit D

The order nodes are visited:

D

Walk-Through

Visited Array

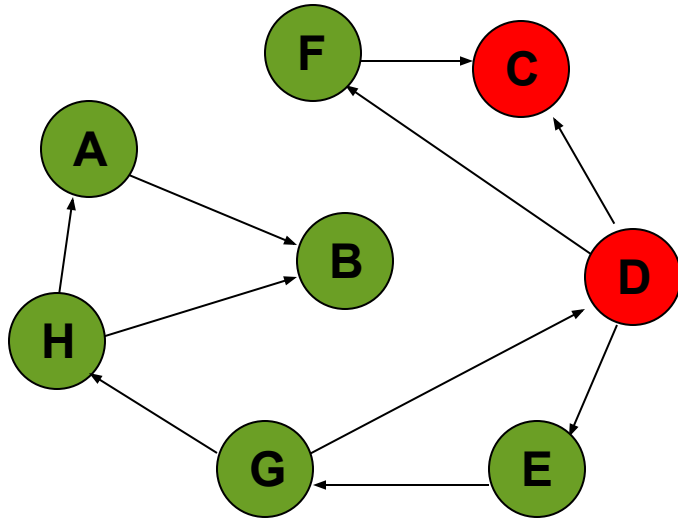|	
A	
B	
C	
D	√
E	
F	
G	
H	

The order nodes are visited:

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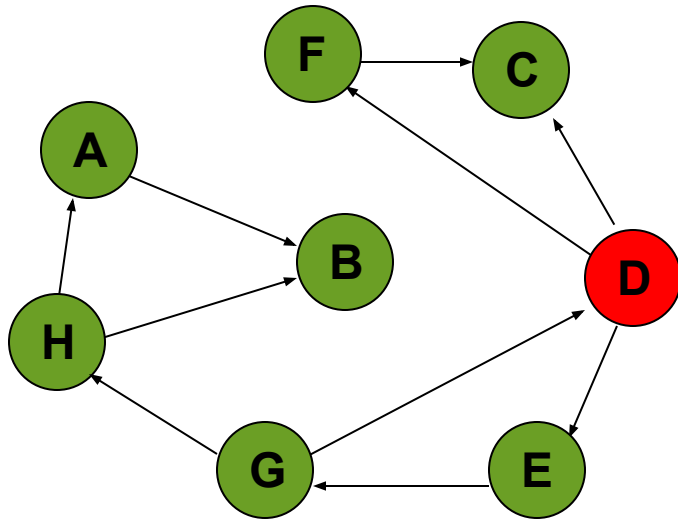
Consider nodes adjacent to D, decide to visit C first (Rule: visit adjacent nodes in alphabetical order)
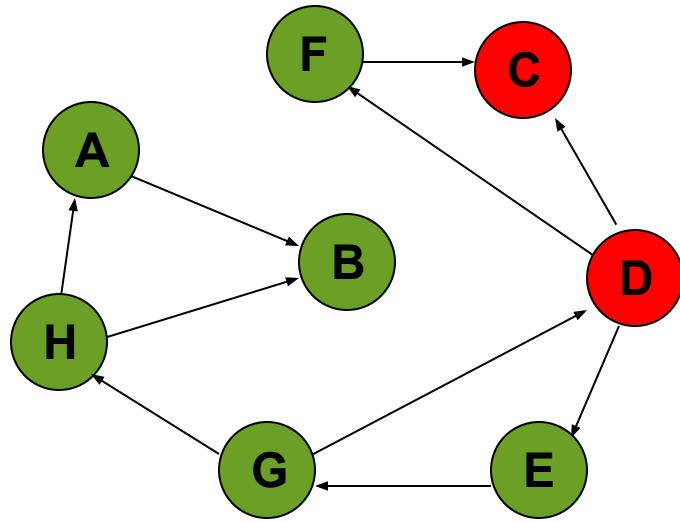
Walk-Through

Visited Array

A	
B	
C	✔
D	✔
E	
F	
G	
H	

C
D

Visit C

The order nodes are visited:

D, C

Walk-Through

Visited Array

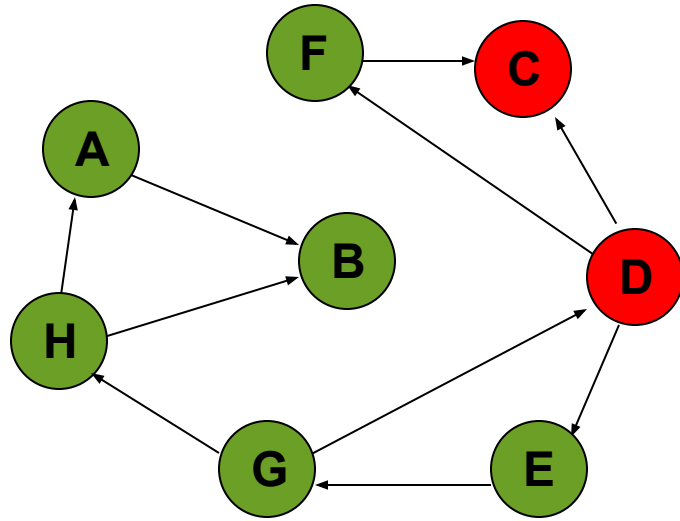|	
A	
B	
C	✔
D	✔
E	
F	
G	
H	

Stack: 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
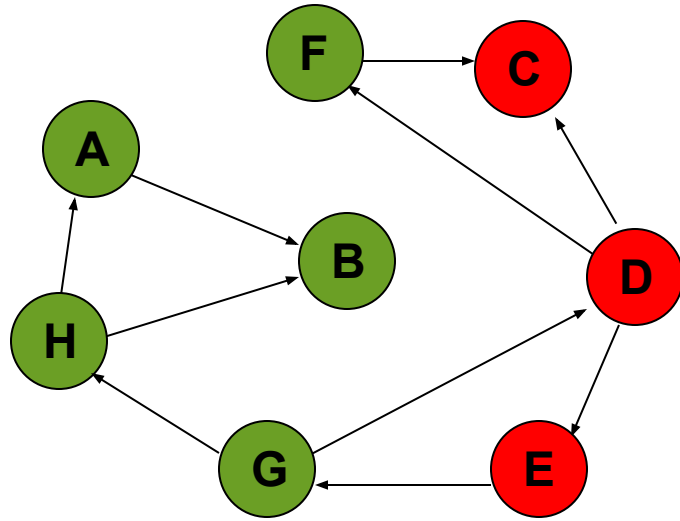
Walk-Through

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	

E
D

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

The order nodes are visited:

D, C, E

Walk-Through

Visited Array

A	
B	
C	✓
D	✓
E	✓
F	
G	
H	

The order nodes are visited:

D, C, E
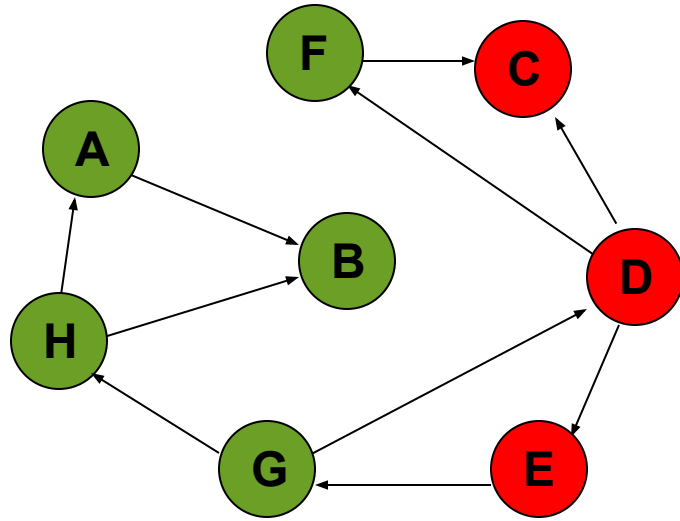
Only G is adjacent to E

Walk-Through
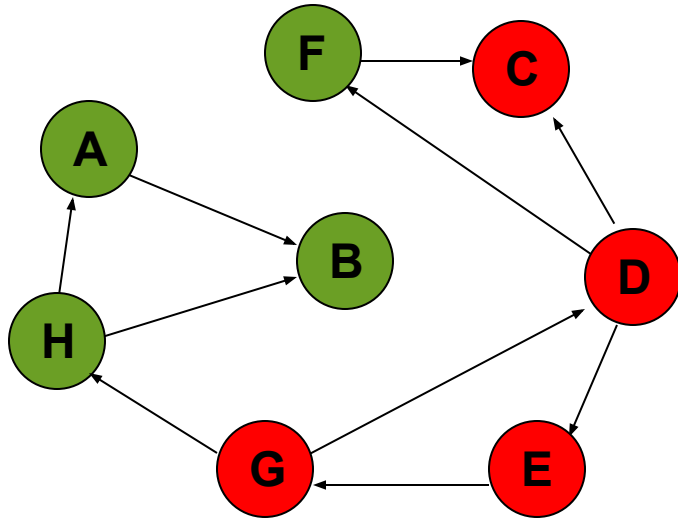
Visited Array

A	
B	
C	✓
D	✓
E	✓
F	
G	✓
H	

Visit G
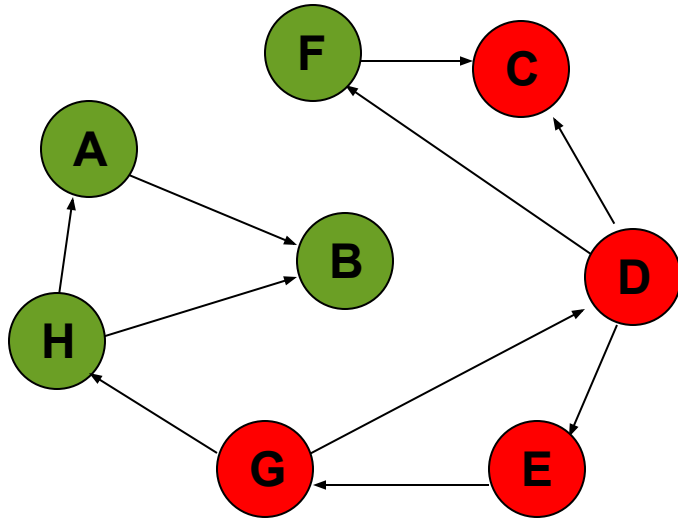
The order nodes are visited:

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
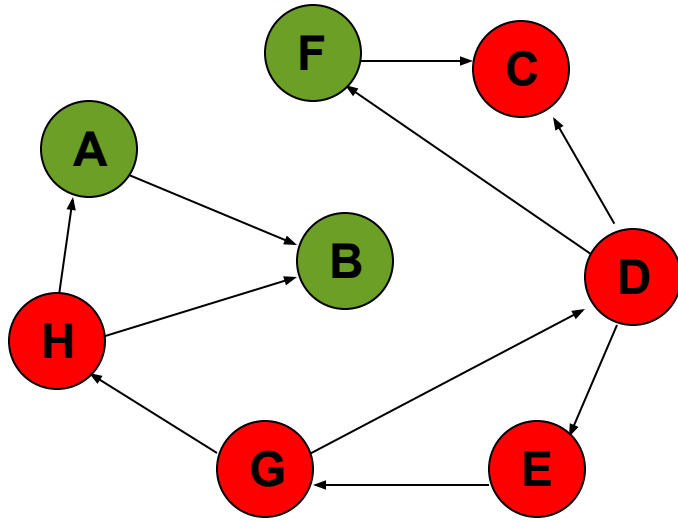
Visited Array

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
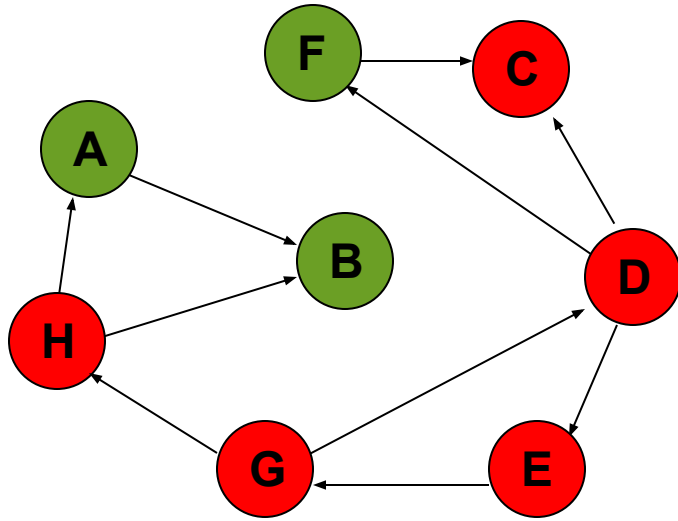
The order nodes are visited:

D, C, E, G, H

**Nodes A and B are adjacent to H.
Decide to visit A next.**

Walk-Through
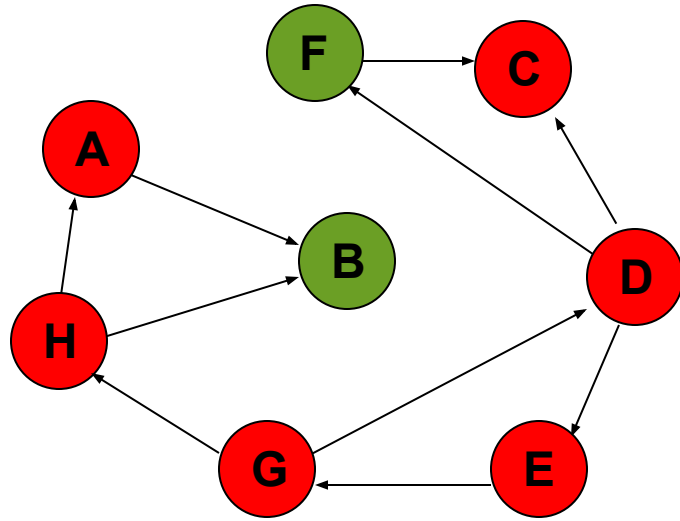
The order nodes are visited:

D, C, E, G, H, A

Walk-Through

Visited Array

A	✓
B	
C	✓
D	✓
E	✓
F	
G	✓
H	✓

A
H
G
E
D
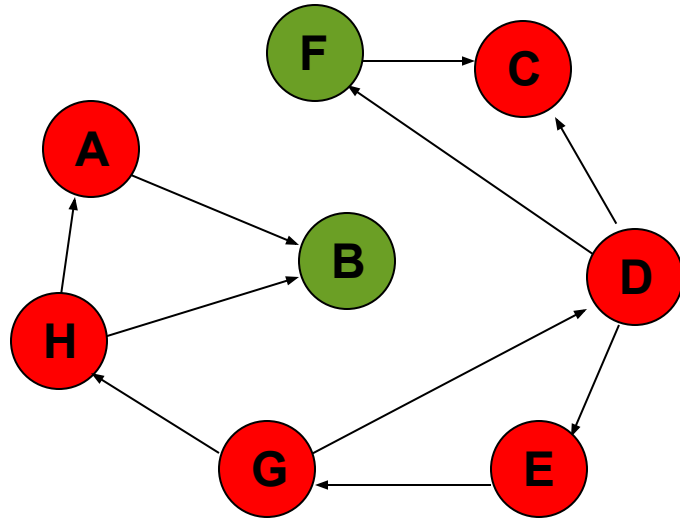
The order nodes are visited:

D, C, E, G, H, A

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

Walk-Through
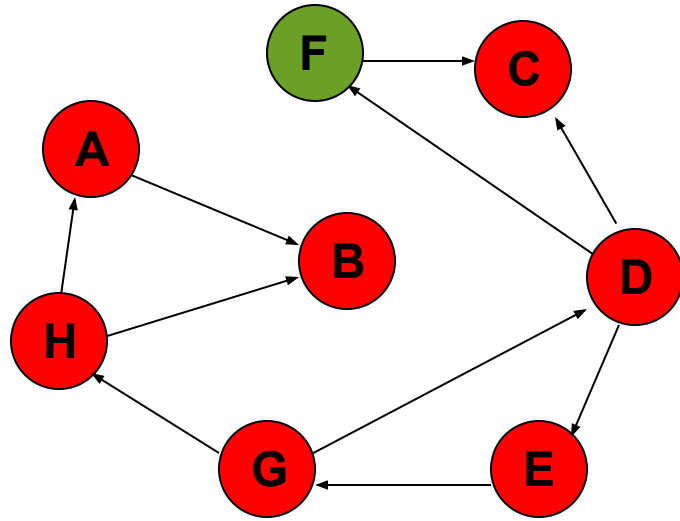
Visited Array

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
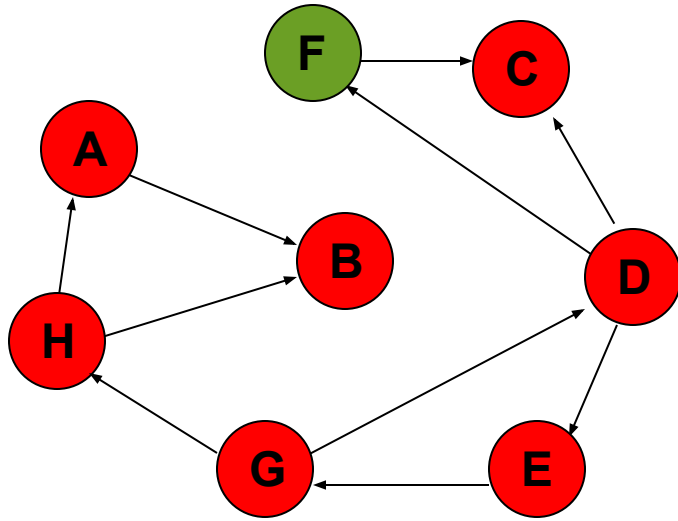
Walk-Through

The order nodes are visited:

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

Visited Array

A	✔
B	✔
C	✔
D	✔
E	✔
F	
G	✔
H	✔

A
H
G
E
D

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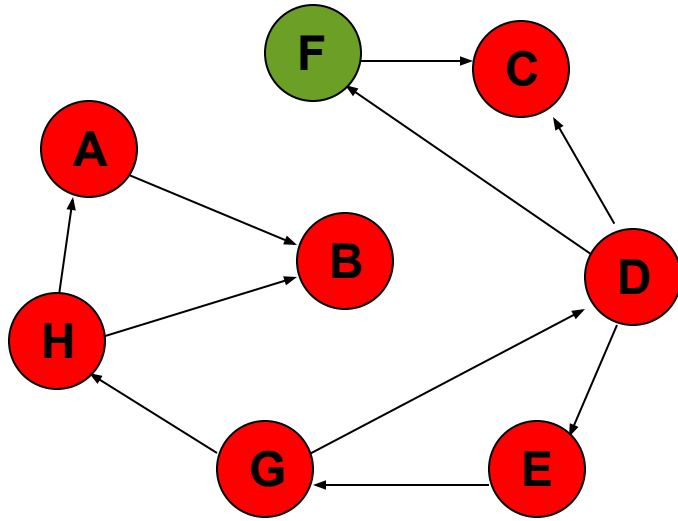
Walk-Through

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).
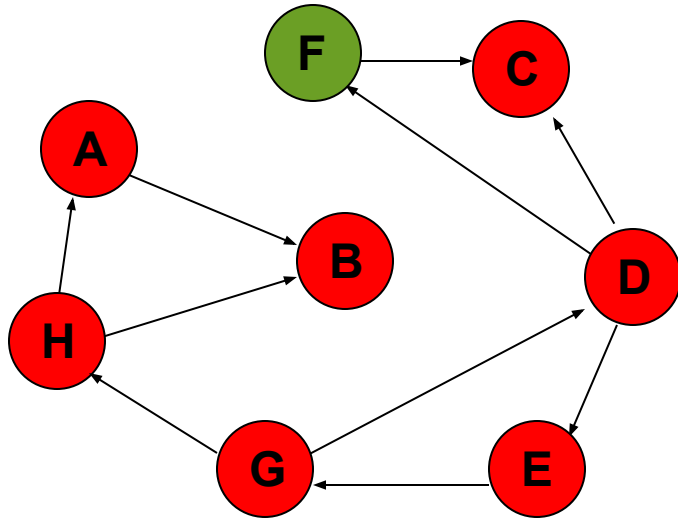
Walk-Through

Visited Array

The order nodes are visited:

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

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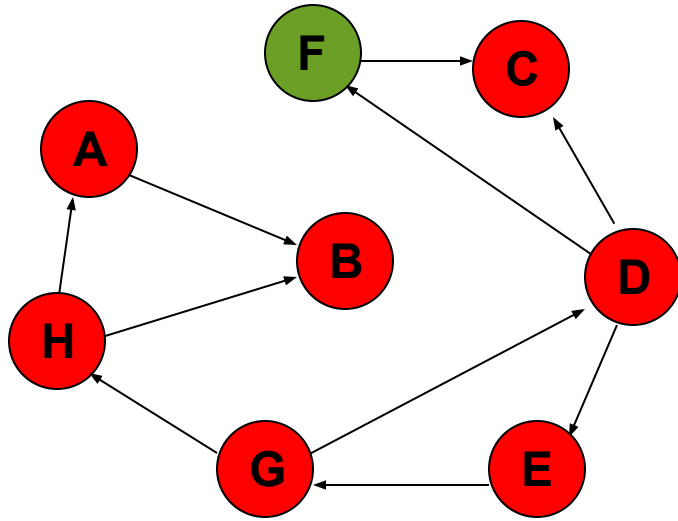
Walk-Through

Visited Array

A	✔
B	✔
C	✔
D	✔
E	✔
F	
G	✔
H	✔

Stack: E, D

The order nodes are visited:

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

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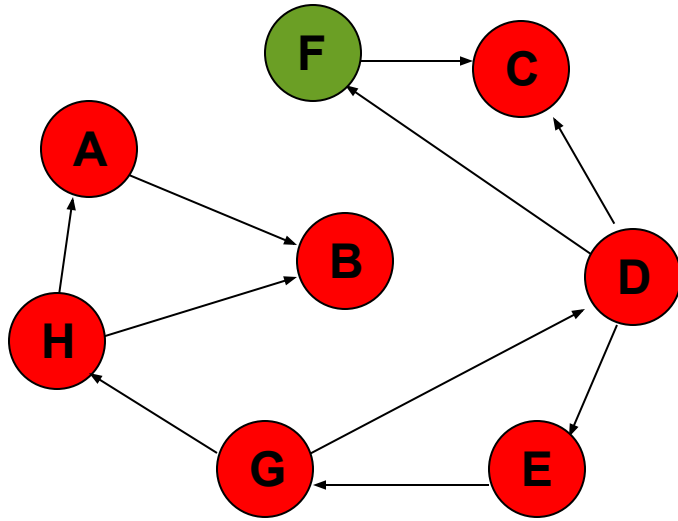
Walk-Through

The order nodes are visited:

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

Visited Array

A	✓
B	✓
C	✓
D	✓
E	✓
F	
G	✓
H	✓

D

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

Walk-Through

Visited Array

A	✔
B	✔
C	✔
D	✔
E	✔
F	
G	✔
H	✔

D
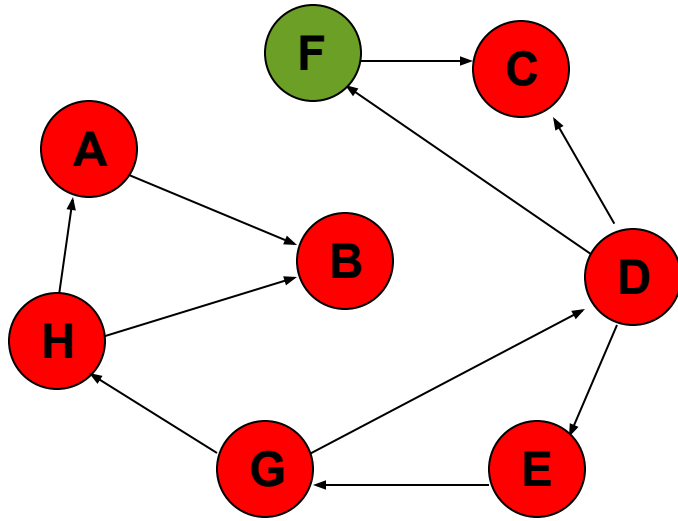
The order nodes are visited:
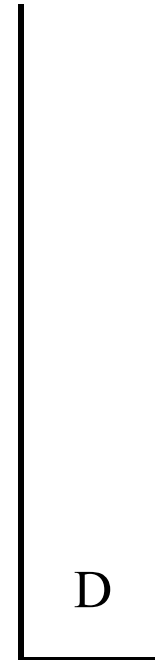
D, C, E, G, H, A, B

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

Walk-Through
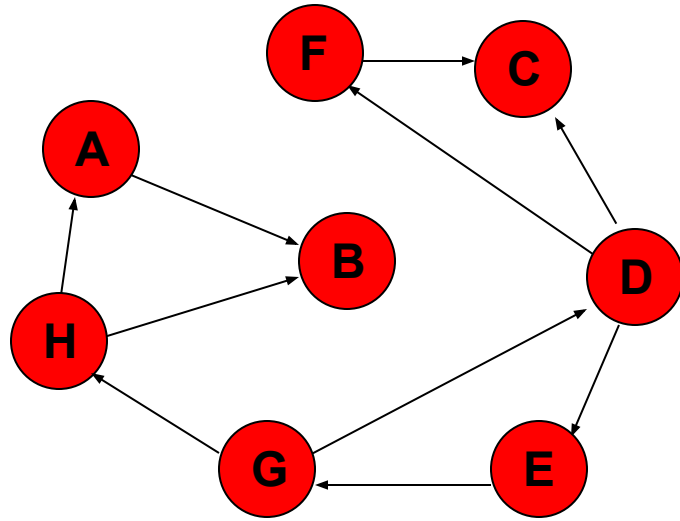
The order nodes are visited:

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

Visited Array

A	✔
B	✔
C	✔
D	✔
E	✔
F	✔
G	✔
H	✔

D

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

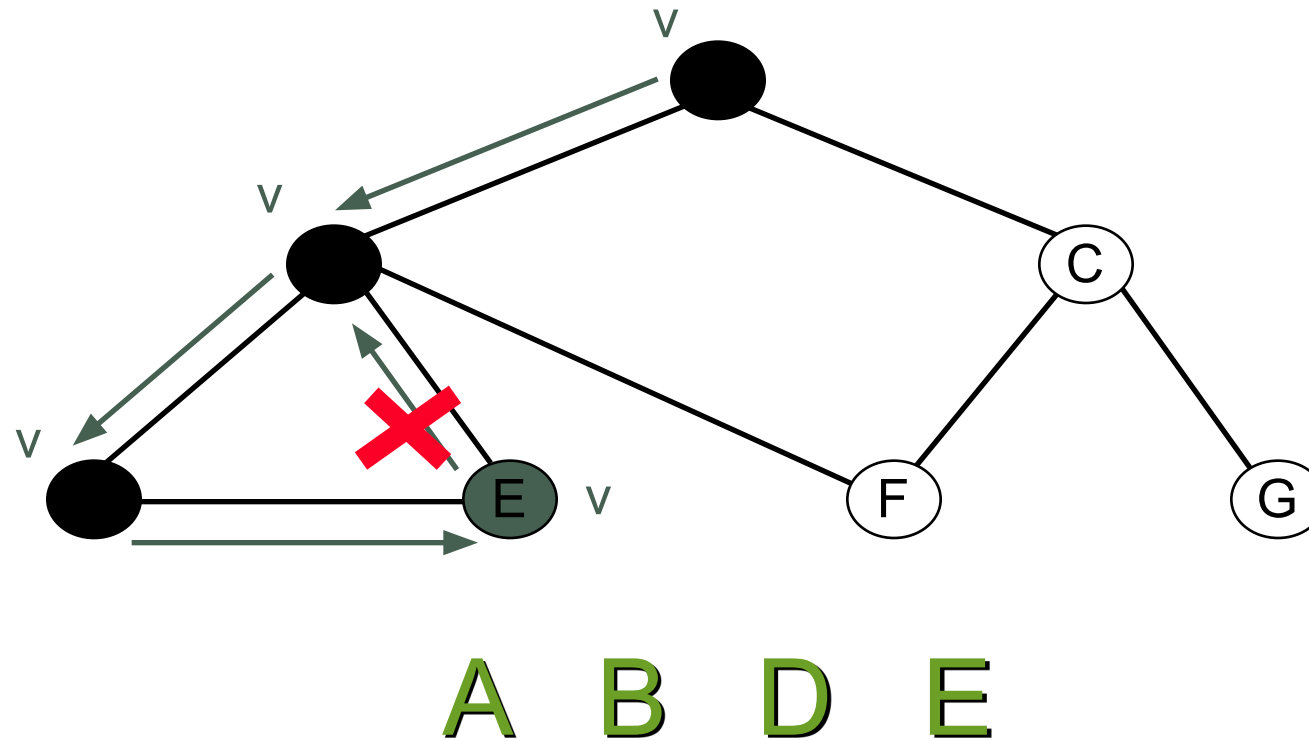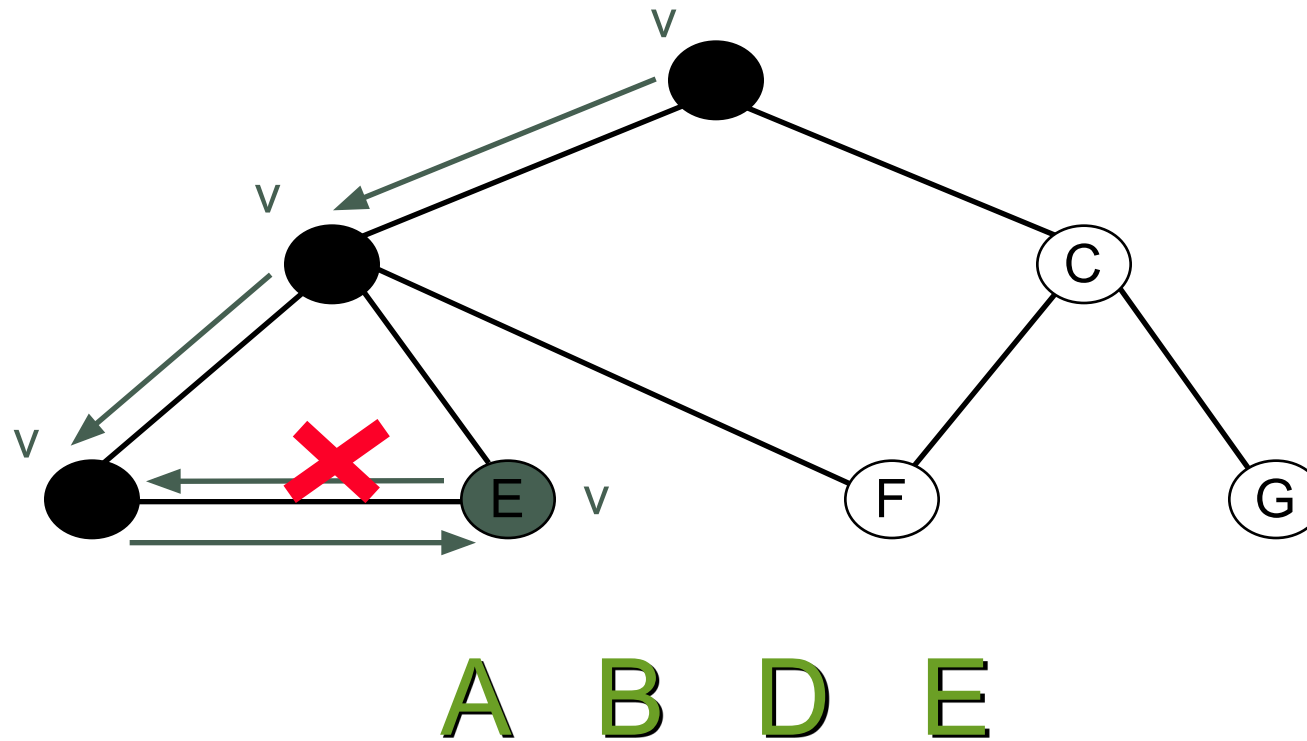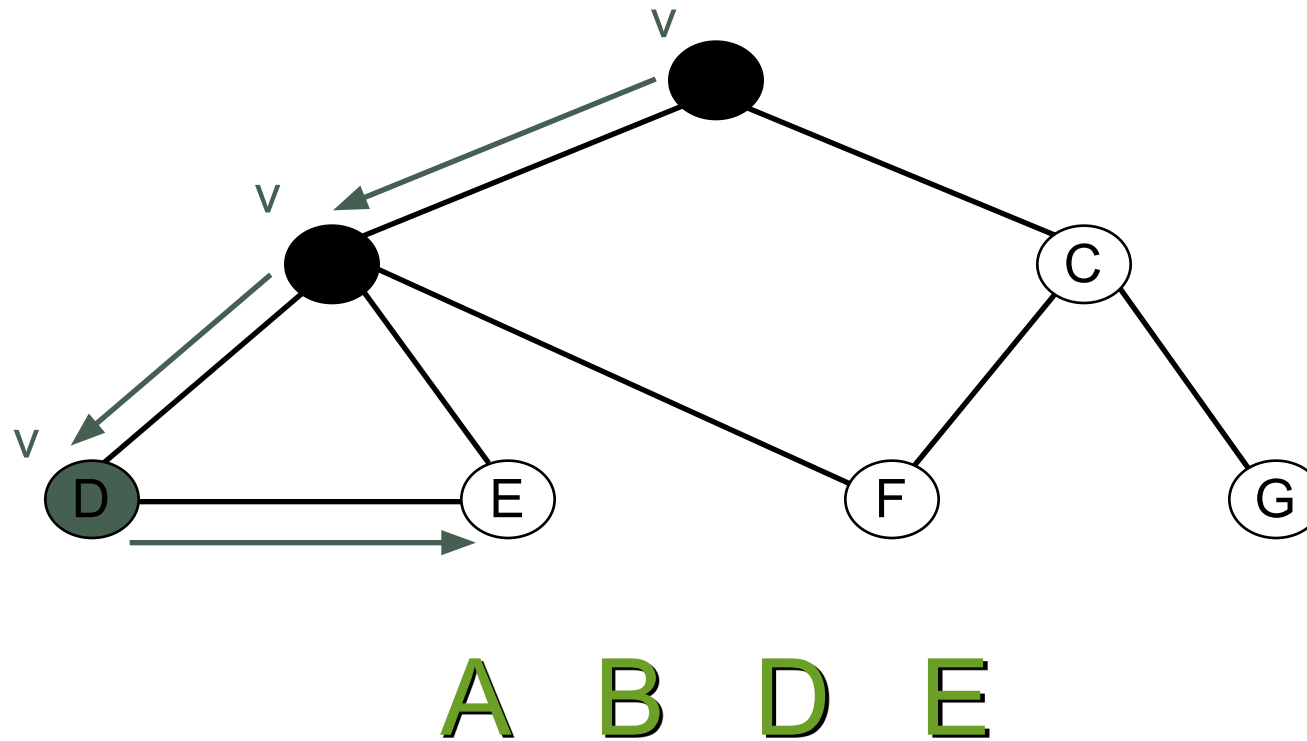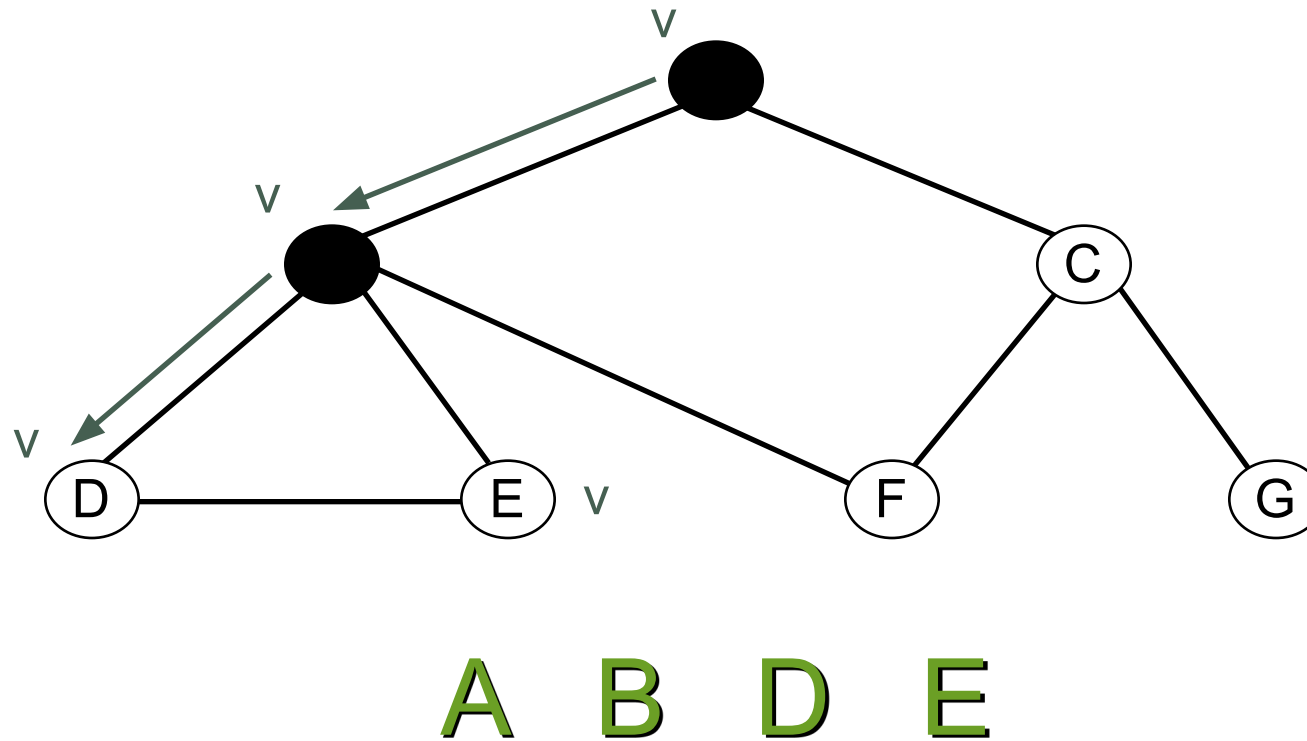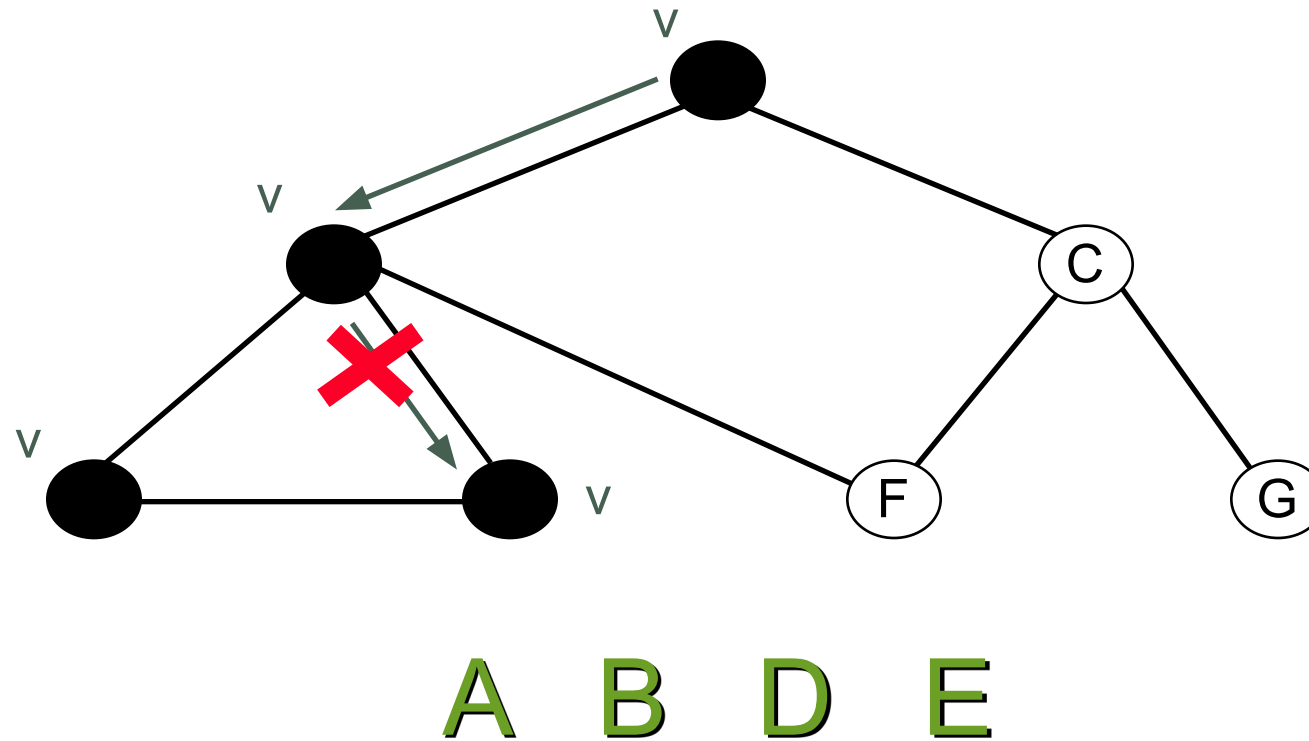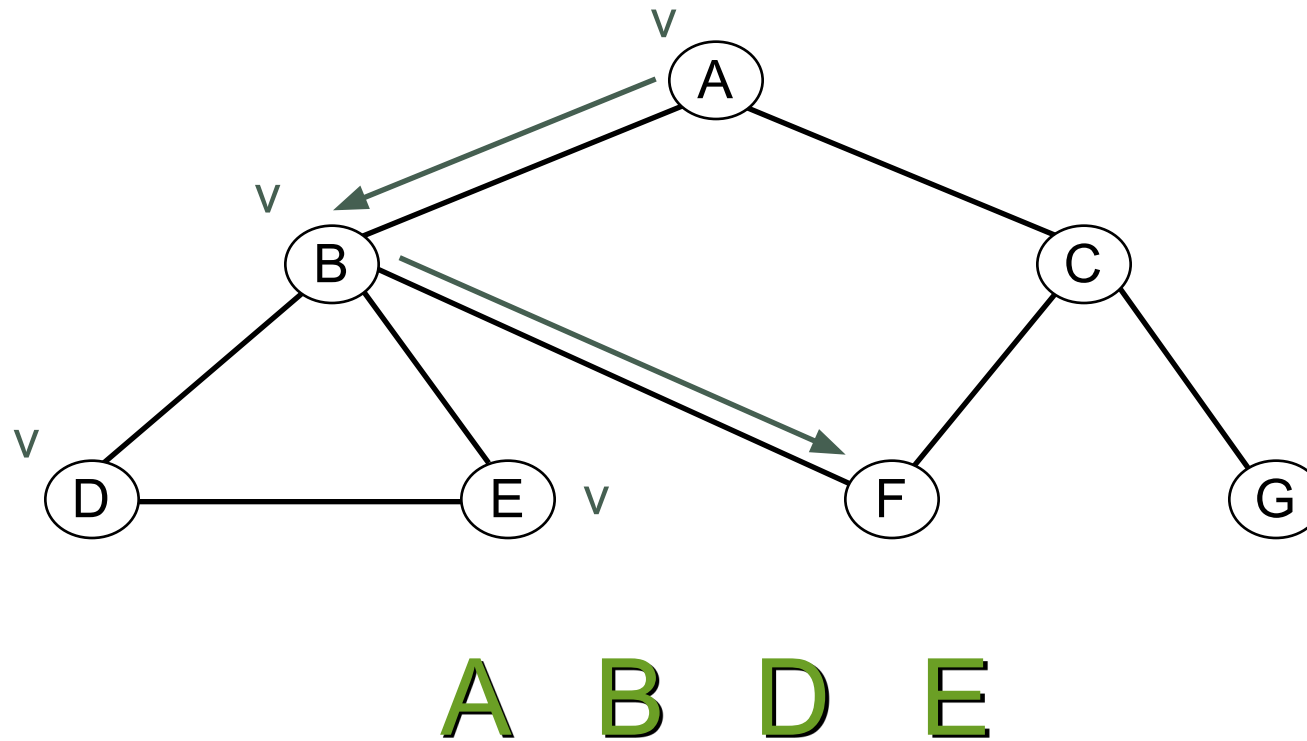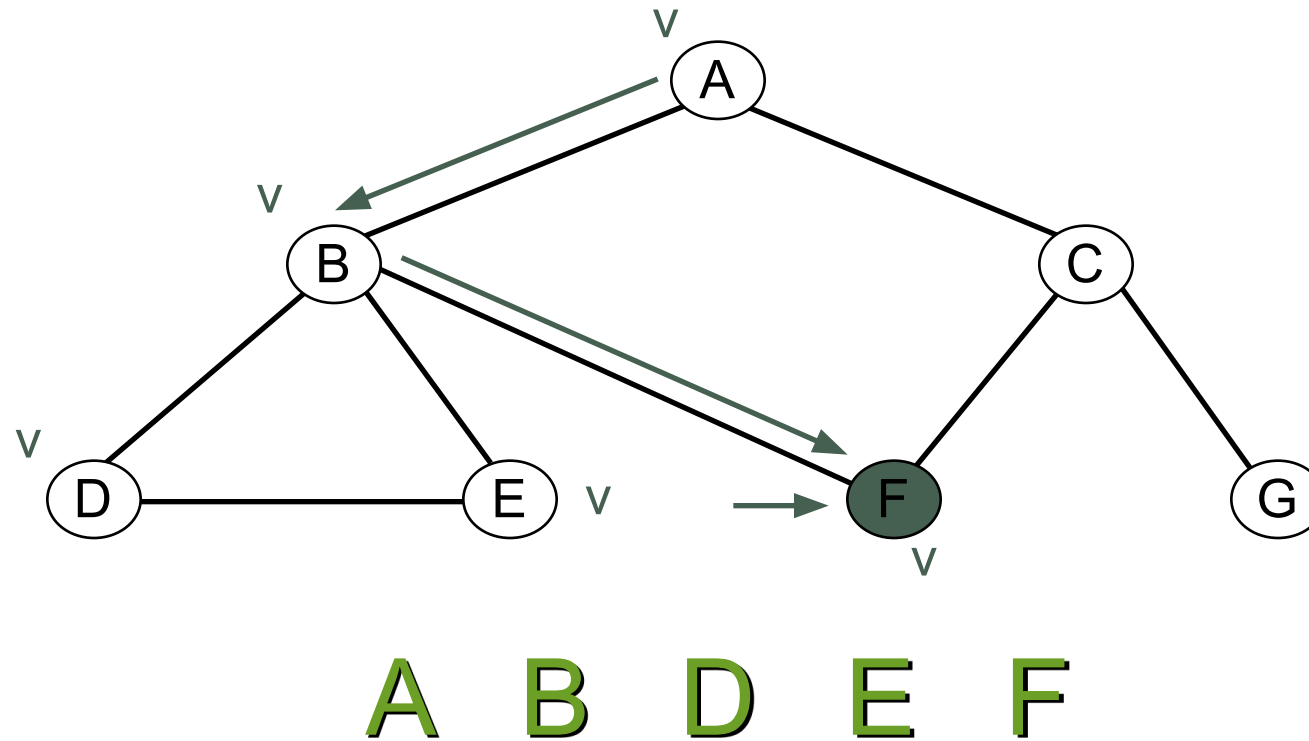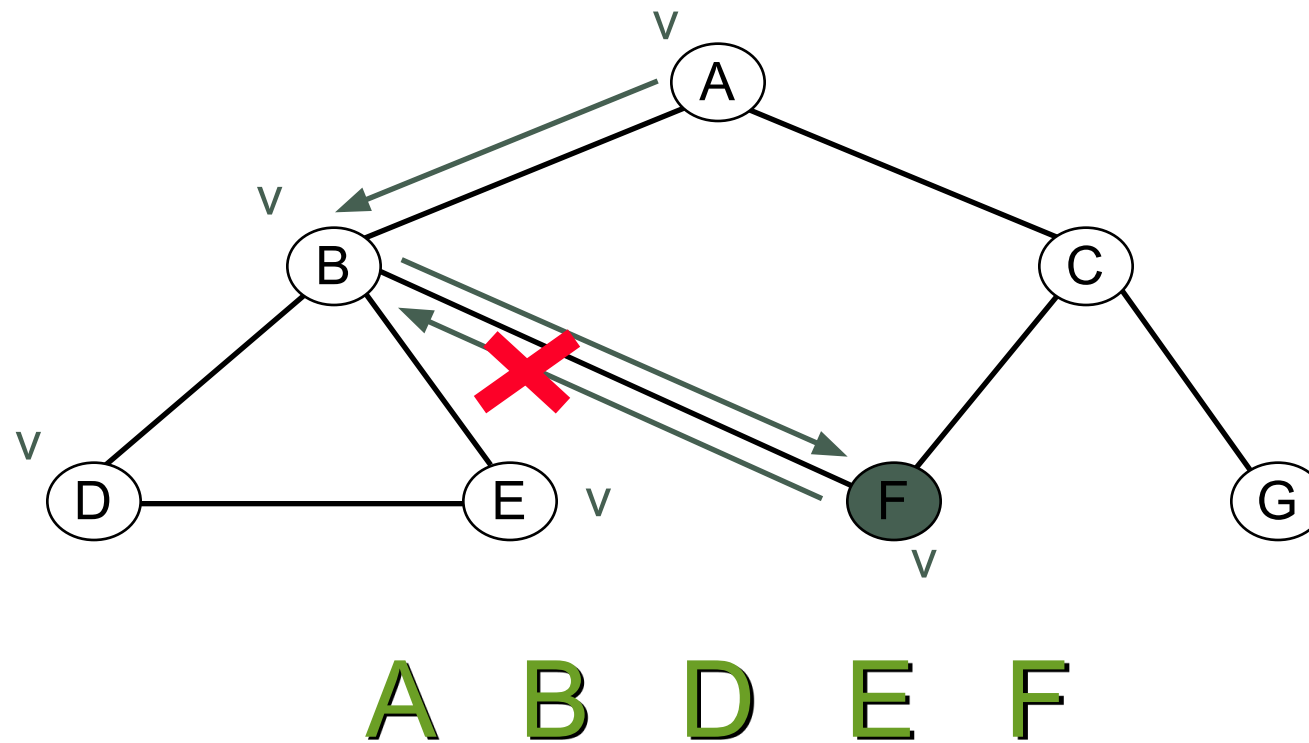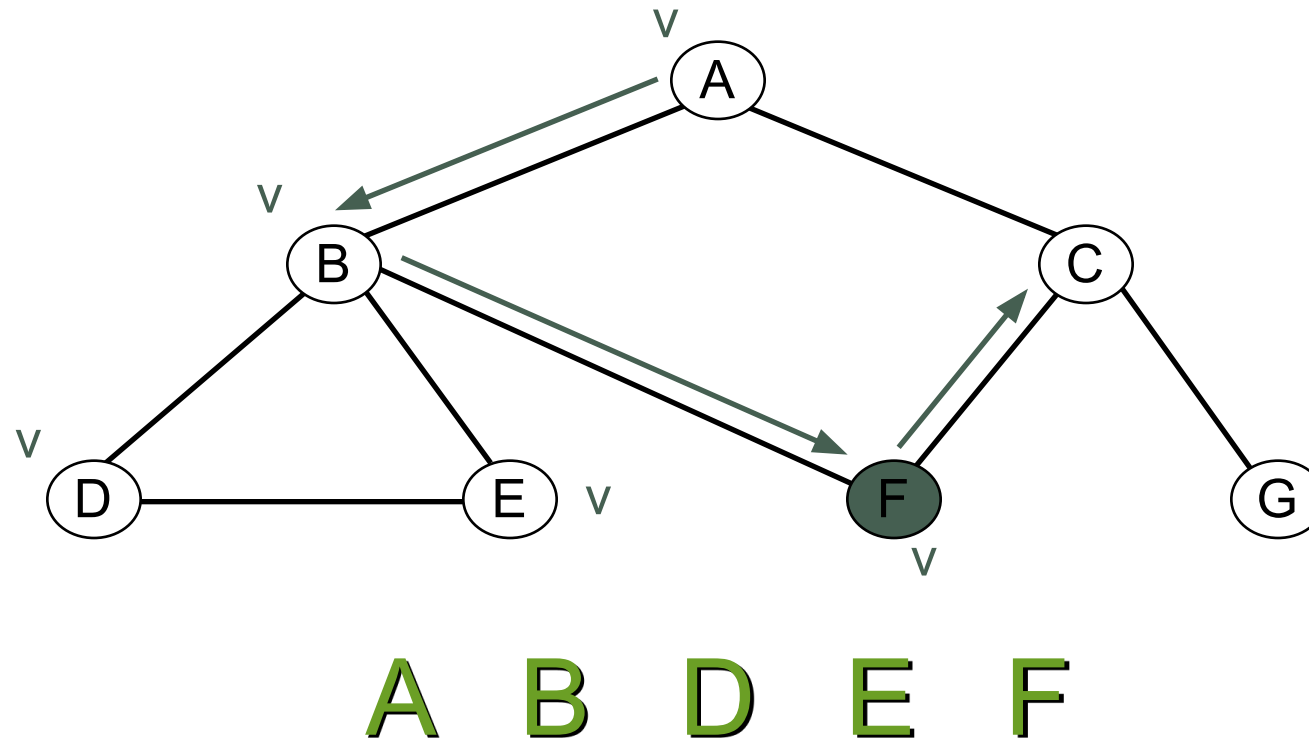**Stack is empty. Depth-first traversal is done.**

Depth-First Search

Depth-First Search

Depth-First Search

Depth-First Search

A B

Depth-First Search

V

V

B   C

D   E   F   G

A B

Depth-First Search

A B

Depth-First Search

A B D

Depth-First Search

A B D

Depth-First Search

Depth-First Search

A B D E

Depth-First Search

A B D E

Depth-First Search

A B D E

Depth-First Search

A B D E

Depth-First Search

A B D E

Depth-First Search

v

v

v

C

v

v

F

G

A B D E

Depth-First Search

A B D E

Depth-First Search

A B D E F

Depth-First Search

A B D E F

Depth-First Search

A B D E F

Depth-First Search

A B D E F C

Depth-First Search

A B D E F C

Depth-First Search

A B D E F C

Depth-First Search

A B D E F C

Depth-First Search

A B D E F C G

Depth-First Search

A B D E F C G

Depth-First Search

A B D E F C G

Depth-First Search

A B D E F C G

Depth-First Search

A B D E F C G

Depth-First Search

A B D E F C G

Depth-First Search
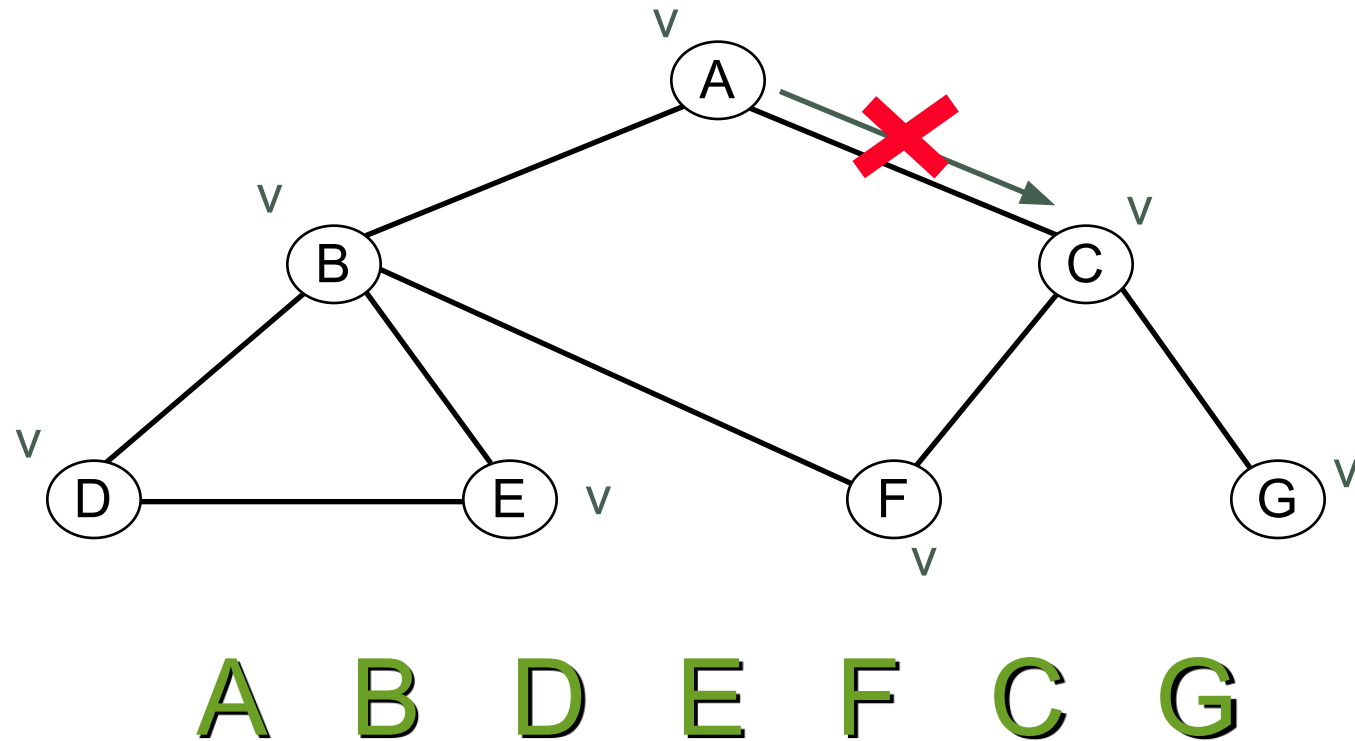

A B D E F C G

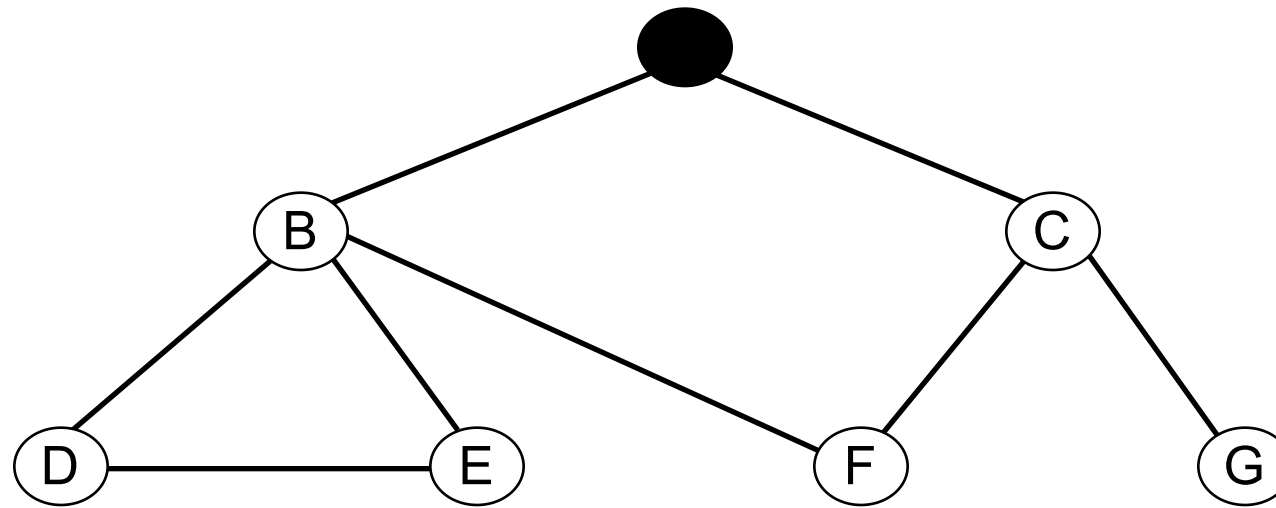# Depth-First Search

Depth-First Search

Time and Space Complexity for Depth-First Search

Time Complexity

◦ Adjacency Lists

◦ Each node is marked visited once

◦ Each node is checked for each incoming edge

◦ O (v + e)

◦ Adjacency Matrix

◦ Have to check all entries in matrix: $O(n^2)$
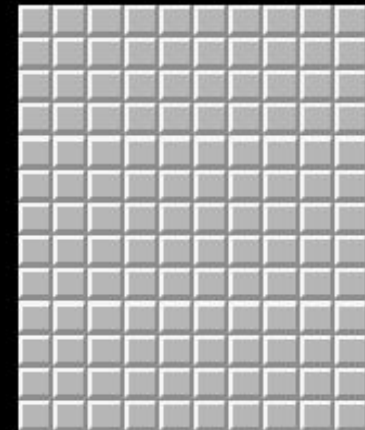
Space Complexity

◦ Stack to handle nodes as they are explored

◦ Worst case: all nodes put on stack (if graph is linear)

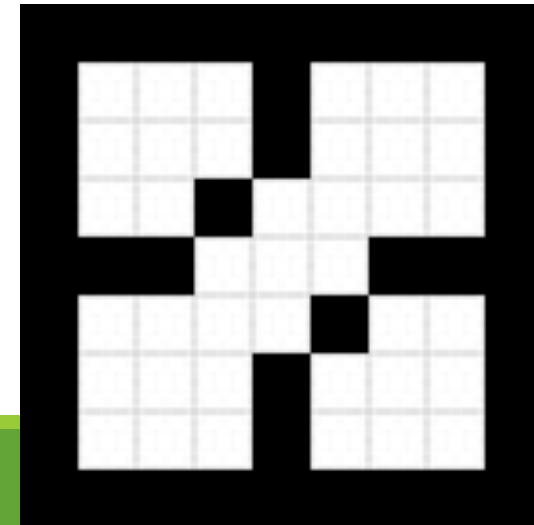◦ O(n)

The "Flood Fill" Algorithm
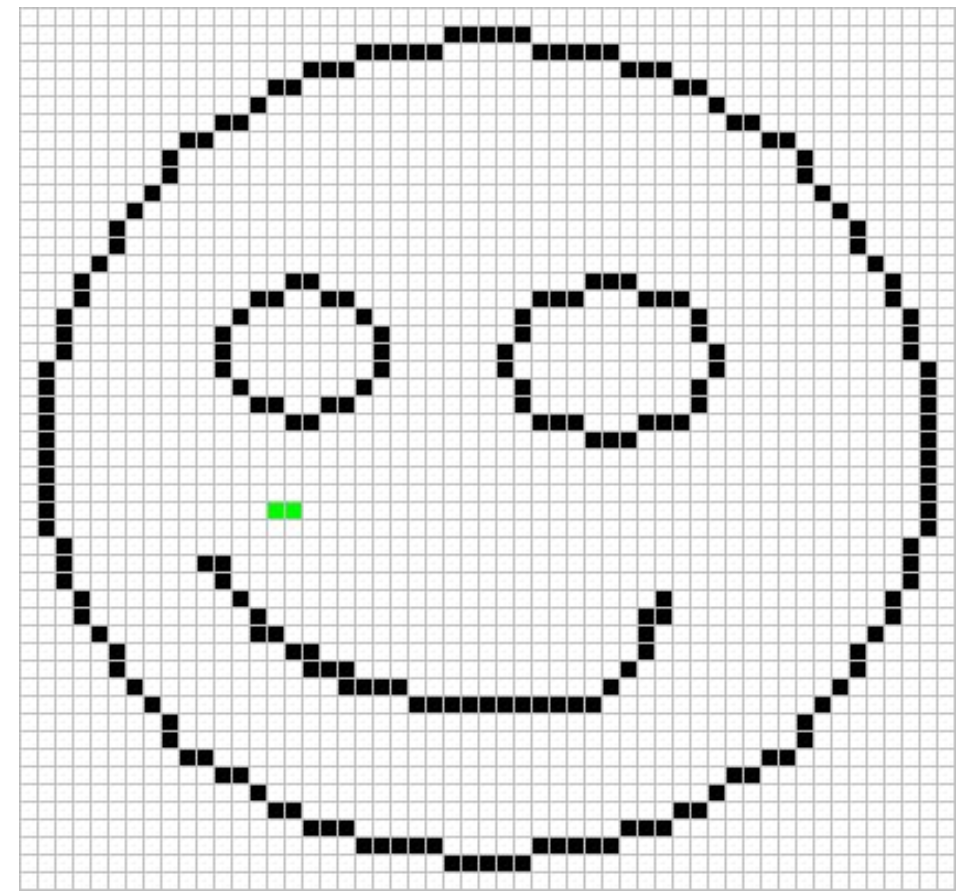
A RECURSIVE GRAPHICS ALGORITHM USED TO FILL IN IRREGULAR-SHAPED REGIONS WITH A SOLID COLOR

A flood-fill algorithm

void fill(int x, int y, int interiorcolor, int newcolor)

{   //Check boundary crossing

    if (get_pixel(x, y) == interiorcolor)

        {

        put_pixel(x, y, newcolor);

        fill(x -1, y, interiorcolor, newcolor);

        fill(x+1, y, interiorcolor, newcolor);

        fill(x, y -1, interiorcolor, newcolor);

        fill(x, y+1, interiorcolor, newcolor);

        }

}

4-Connected — Three objects

8-Connected — One diagonal object

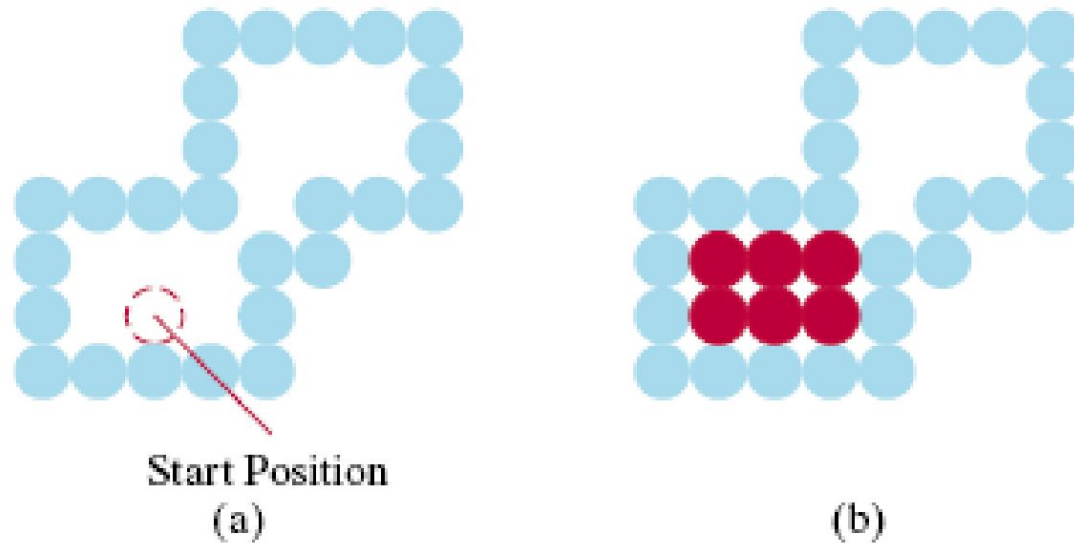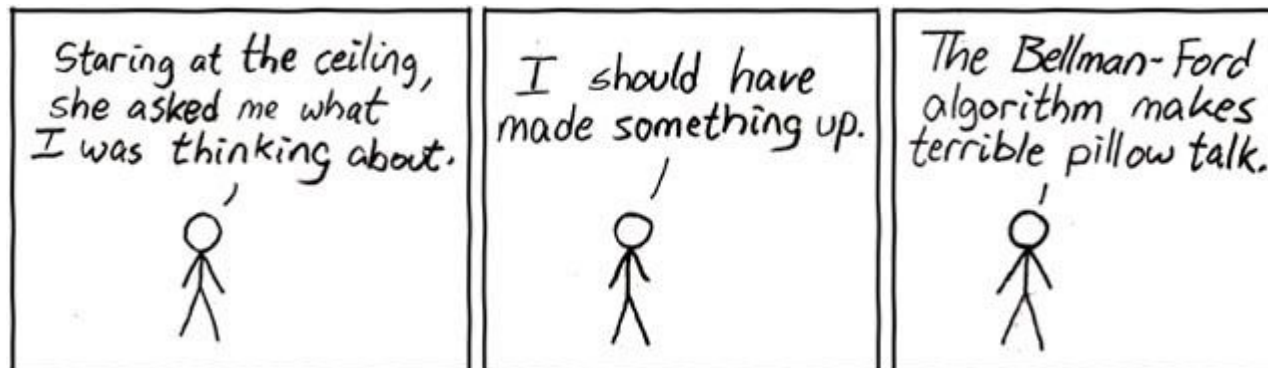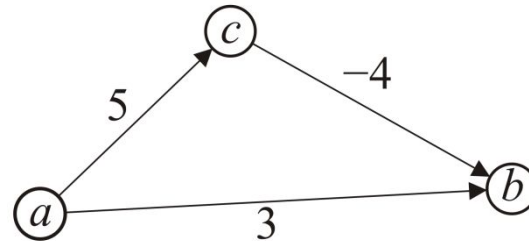# A flood-fill algorithm

Start Position
(a)

(b)

FIGURE 4–28   The area defined within the color
boundary (a) is only partially filled in (b) using a
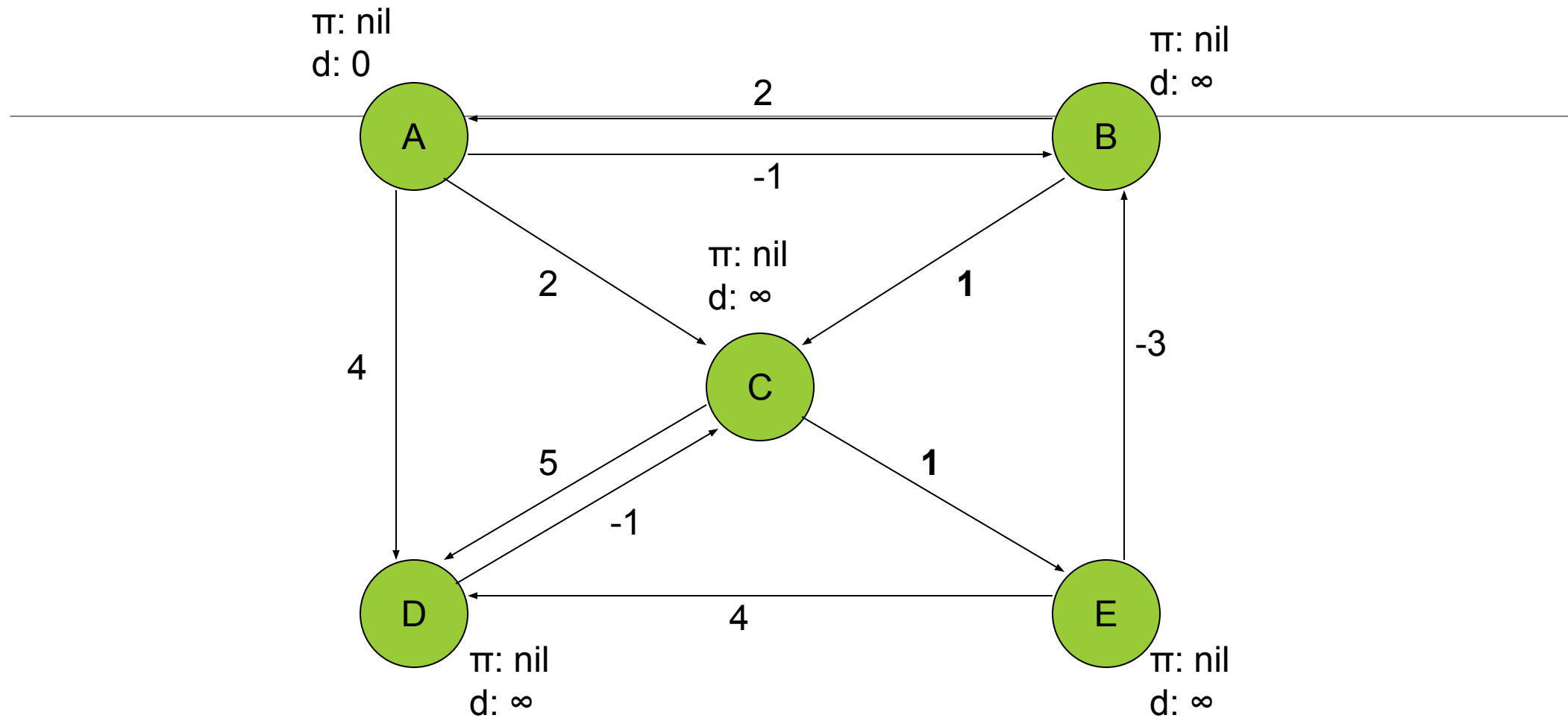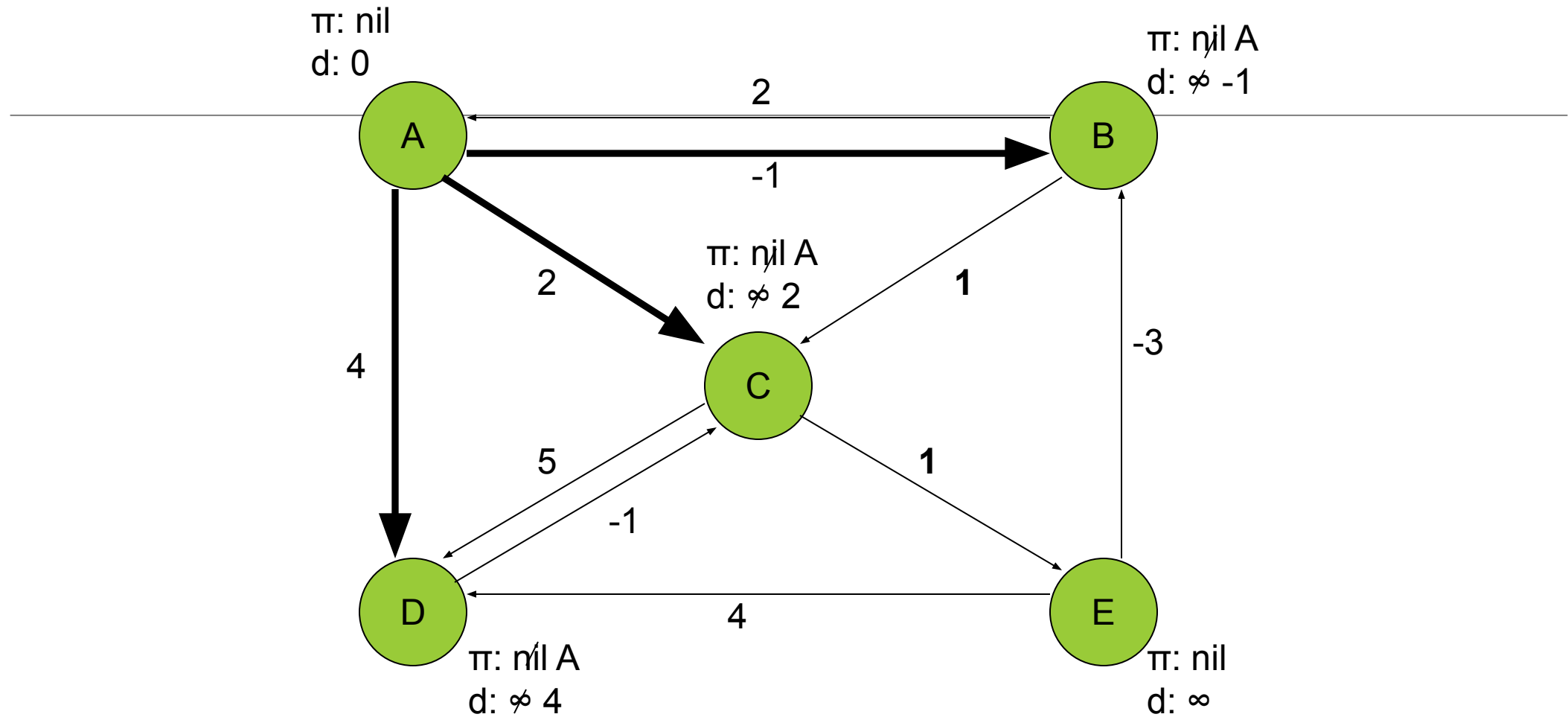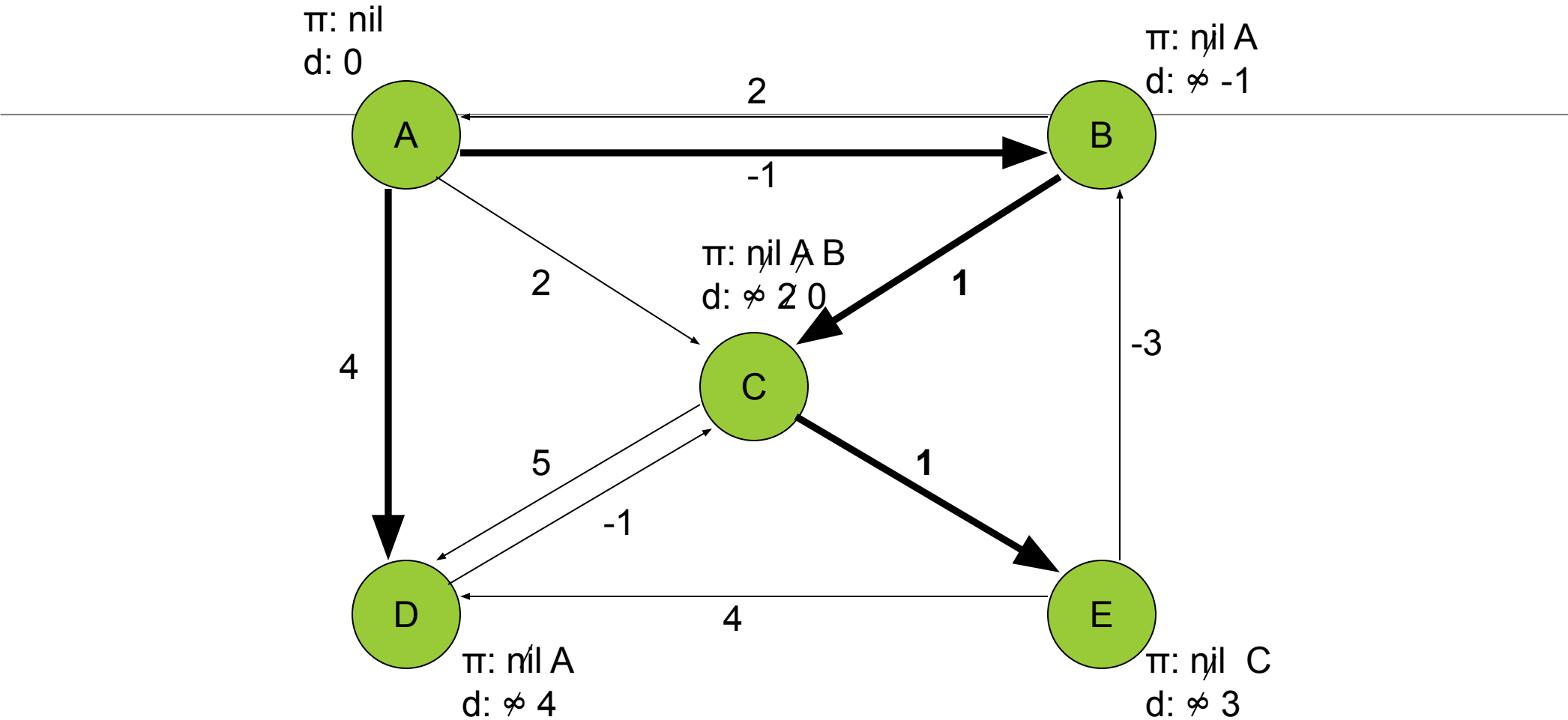4-connected boundary-fill algorithm.

Negative Weights

If some of the edges have negative weight, so long as there are no cycles with negative weight, the Bellman-Ford algorithm will find the minimum distance
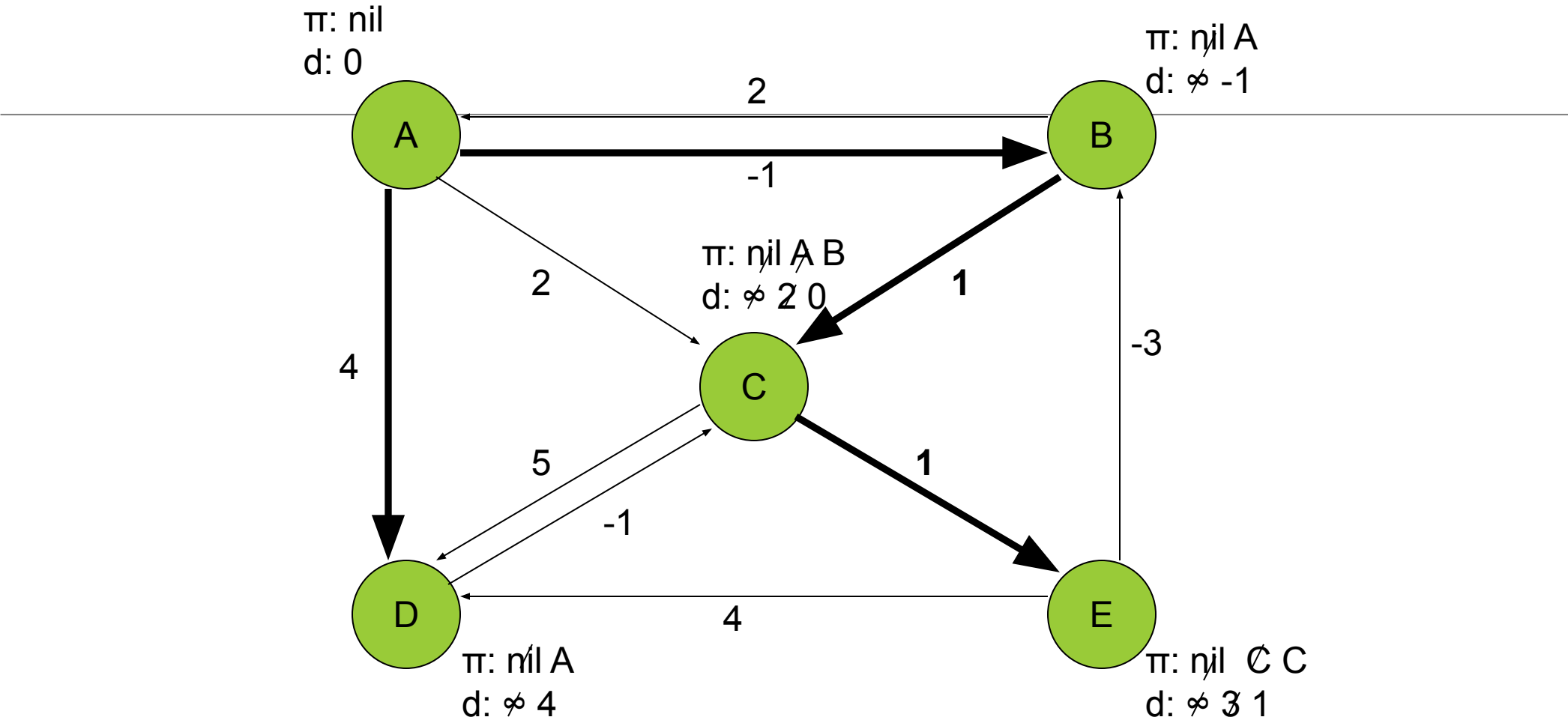
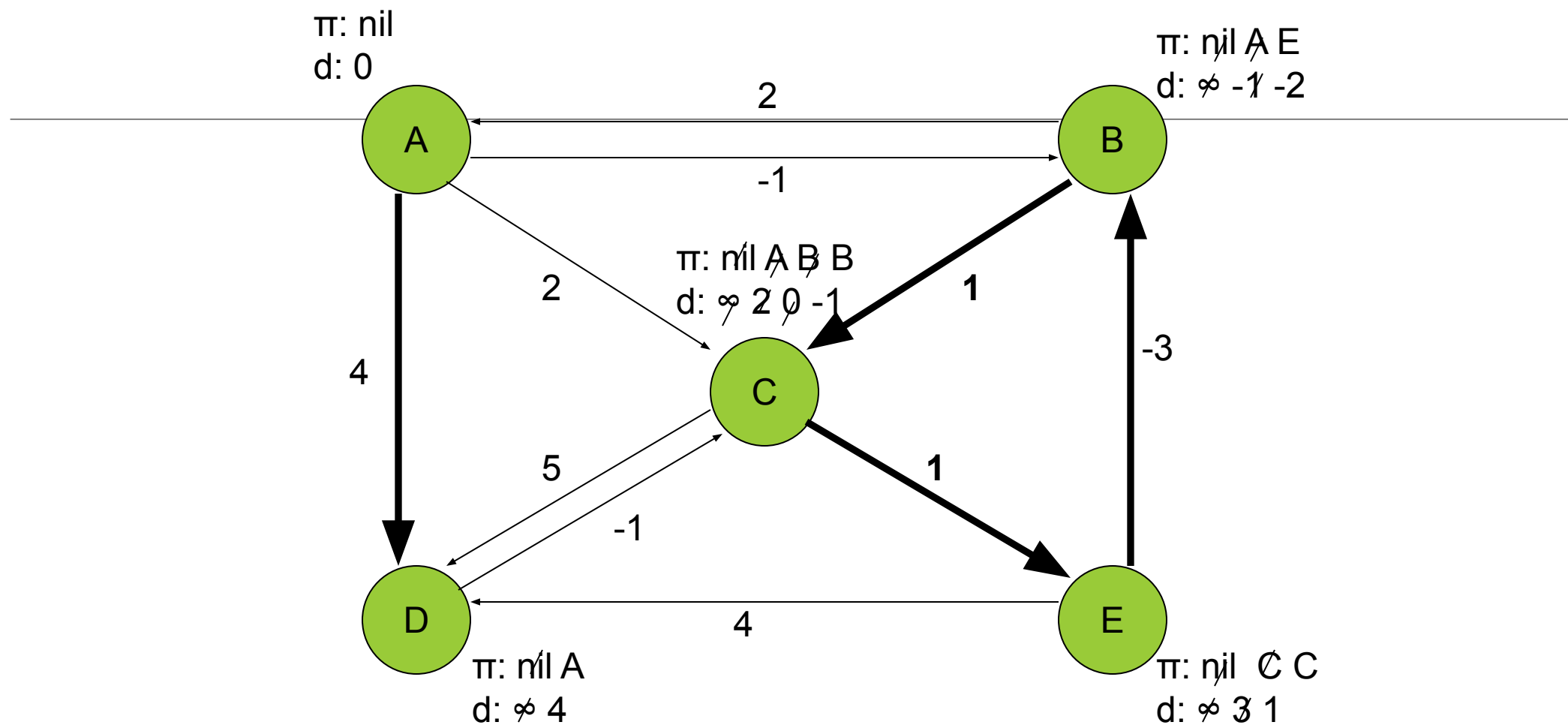◦ It is slower than Dijkstra's algorithm

π: nil
d: 0

π: nil A E
d: ∞ -1 -2

A

2

B

-1

π: nil A B B
d: ∞ 2 0 -1

2

1

1

4

C

-3

5

1

-1

D

4

E

π: nil A
d: ∞ 4

π: nil C C
d: ∞ 3 1

The Bellman-Ford Algorithm

Bellman-Ford(G, w, s)

1. Initialize-Single-Source(G, s)
2. **for** i := 1 to |V| - 1 **do**
3.   **for** each edge (u, v) ∈ E **do**
4.   Relax(u, v, w)
5. **for** each vertex v ∈ u.adj **do**
6.   if d[v] > d[u] + w(u, v)
7.   **then return** False   // there is a negative cycle
8. **return** True

Relax(u, v, w)
  if d[v] > d[u] + w(u, v)
    then d[v] := d[u] + w(u, v)
        parent[v] := u

Time Complexity

Bellman-Ford(G, w, s)

1. Initialize-Single-Source(G, s)
2. **for** i := 1 to |V| - 1 **do** $\longrightarrow$ O(|V|)
3.   **for** each edge (u, v) ∈ E **do**
4.   Relax(u, v, w) $\longrightarrow$ O(|V||E|)
5. **for** each vertex v ∈ u.adj **do** $\longrightarrow$ O(|E|)
6.   if d[v] > d[u] + w(u, v)
7.   **then return** False   // there is a negative cycle
8. **return** True

Time complexity: O(|V||E|)