Advanced Data Structures and Algorithms

Associate Professor Dr. Raed Ibraheem Hamed

University of Human Development, College of Science and Technology
Computer Science Department

2015 – 2016

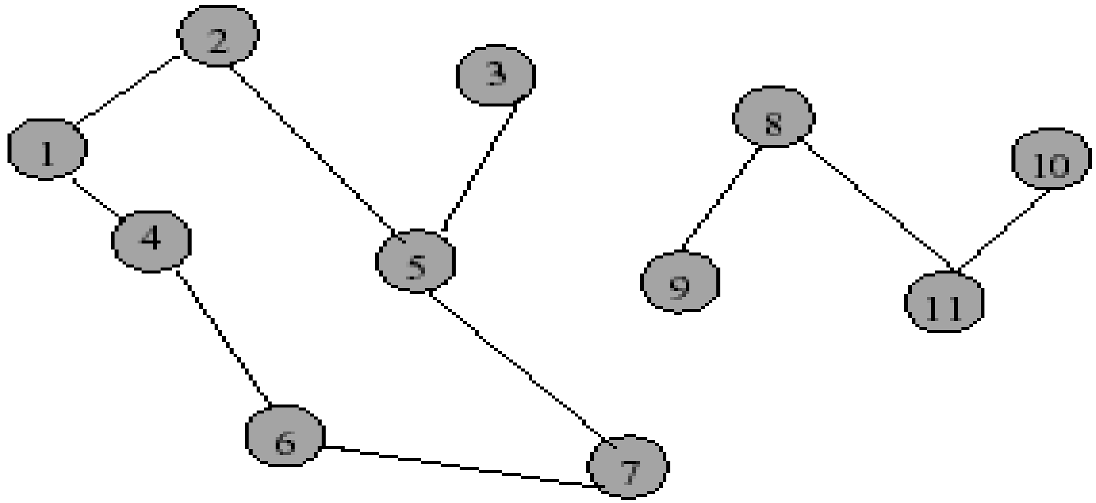
What this Lecture is about:

- Graph Related Concepts
- Vertex Degree
- In-Degree of a Vertex
- Out-Degree of a Vertex
- Sum of In- and Out-DegreesG
- Complete Undirected Graphs
- Graph terminology
- Shortest Paths
- Depth-First Search (DFS)
- Depth-First Search algorithm

Definition of Some Graph Related Concepts

- Let G be a directed graph
  - The *indegree* of a node x in G is the number of edges coming to x
  - The *outdegree* of x is the number of edges leaving x.
- Let G be an undirected graph
  - The *degree* of a node x is the number of edges that have x as one of their end nodes
  - The *neighbors* of x are the nodes adjacent to x

Vertex Degree

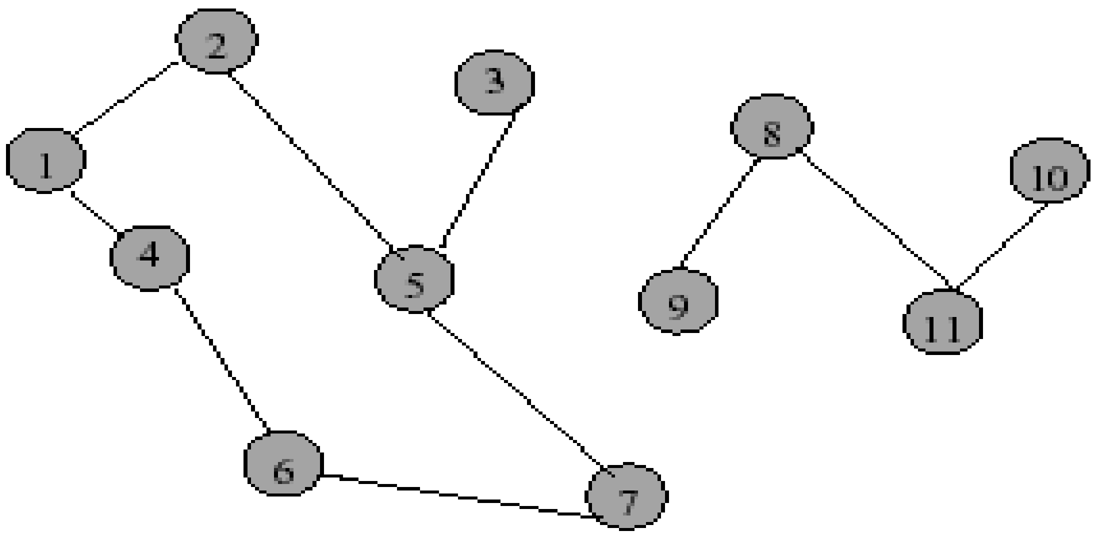


- The **degree** of vertex *i* is the **no. of edges incident** on vertex *i*.

e.g., degree(2) = 2, degree(5) = 3, degree(3) = 1

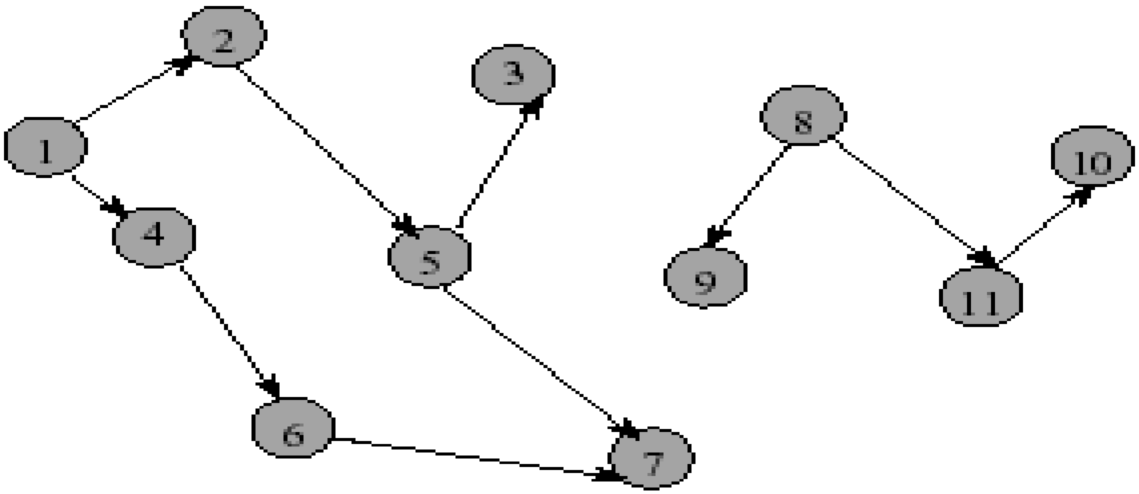
Unlike **trees**, a **graph** can have **cycles**:

Sum of Vertex Degrees



Sum of degrees = *2e* (where *e* is the number of edges)

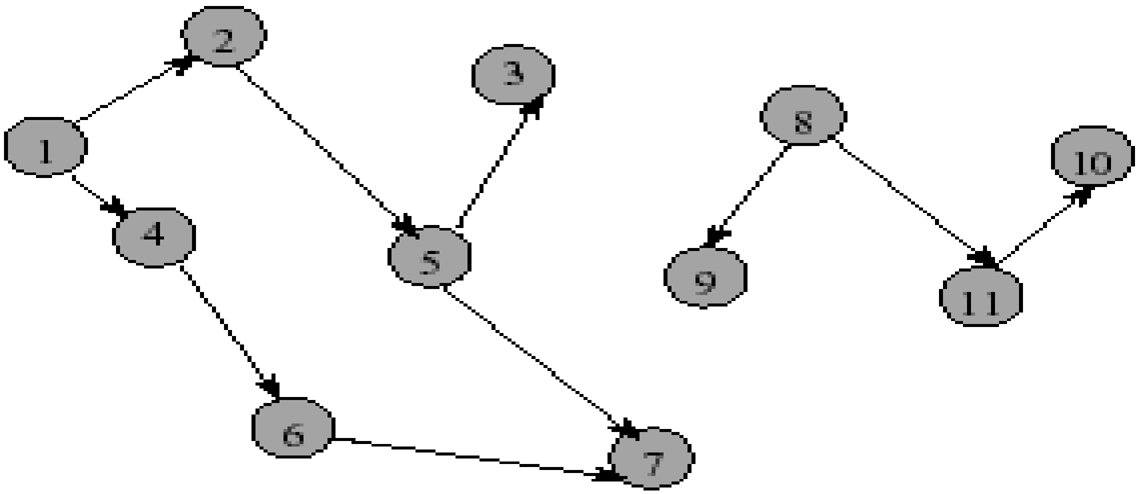
In-Degree of a Vertex



- **In-degree** of vertex *i* is the number of edges incident to *i* (i.e., the number of incoming edges).

e.g., indegree(2) = 1, indegree(8) = 0

Out-Degree of a Vertex



- **Out-degree** of vertex *i* is the number of edges incident from *i* (i.e., the number of outgoing edges).

- e.g., outdegree(2) = 1, outdegree(8) = 2

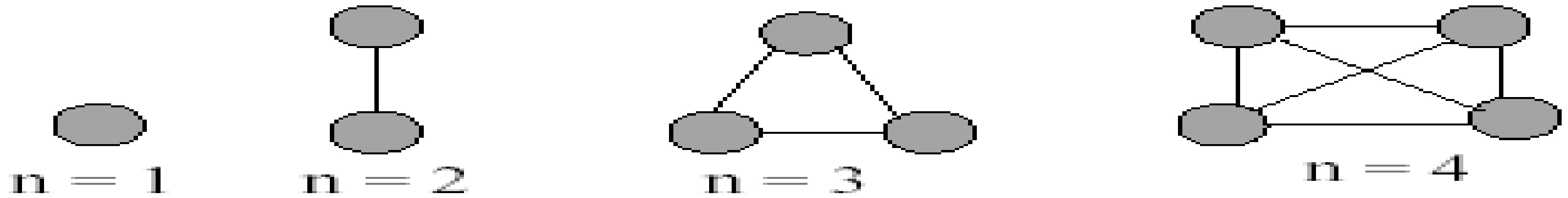
Sum of In- and Out-Degrees

- Each edge contributes
  1 to the in-degree of some vertex and
  1 to the out-degree of some other vertex.

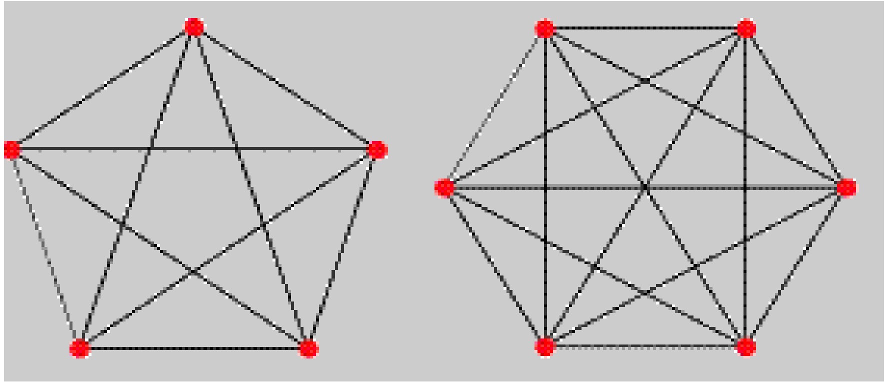
- Sum of in-degrees = sum of out-degrees = e, where e is the number of edges in the digraph.

Complete Undirected Graphs

- A **complete undirected graph** has n(n-1)/2 edges (i.e., all possible edges) and is denoted by K_n

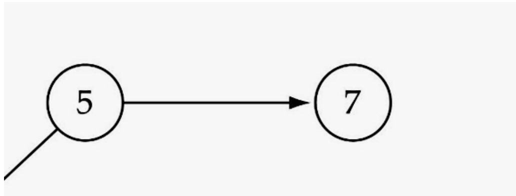


- What would a complete undirected graph look like when n=5? When n=6?



Graph terminology

- Adjacent nodes: two nodes are adjacent if they are connected by an edge



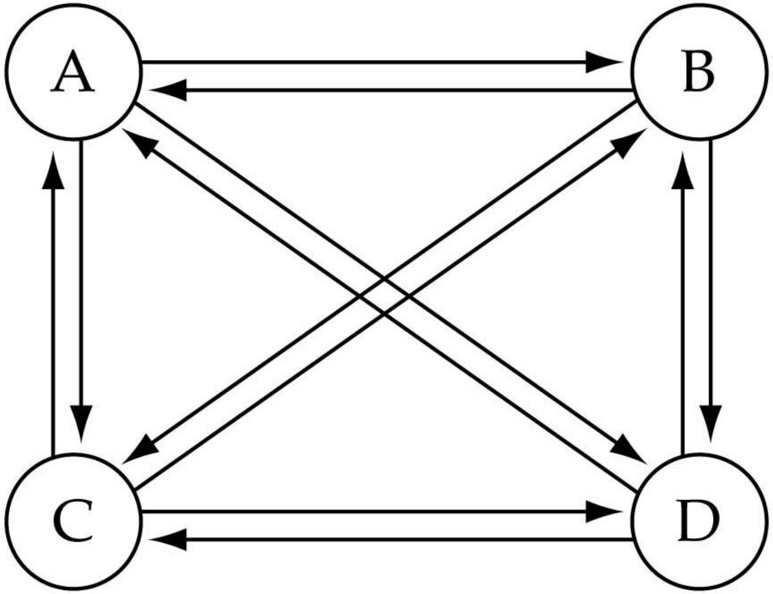
5 is adjacent to 7
7 is adjacent from 5

- Path: a sequence of vertices that connect two nodes in a graph
- Complete graph: a graph in which every vertex is directly connected to every other vertex

Graph terminology (cont.)

- What is the number of edges in a complete directed graph with N vertices?

    *N * (N-1)*

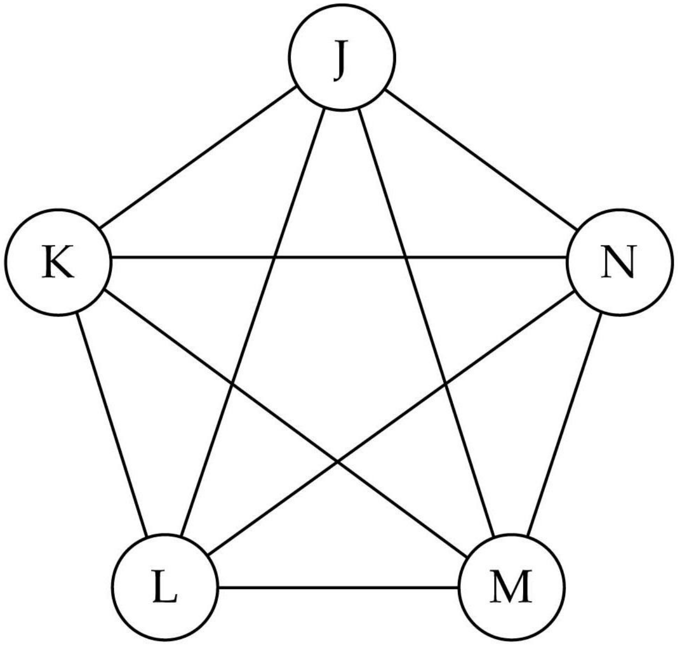


(a) Complete directed graph.

Graph terminology (cont.)

- What is the number of edges in a complete undirected graph with N vertices?

  N * (N-1) / 2



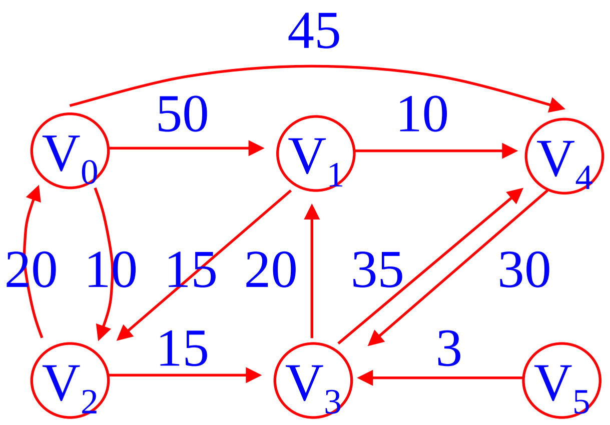
(b) Complete undirected graph.

Shortest Paths

Single source/All destinations:

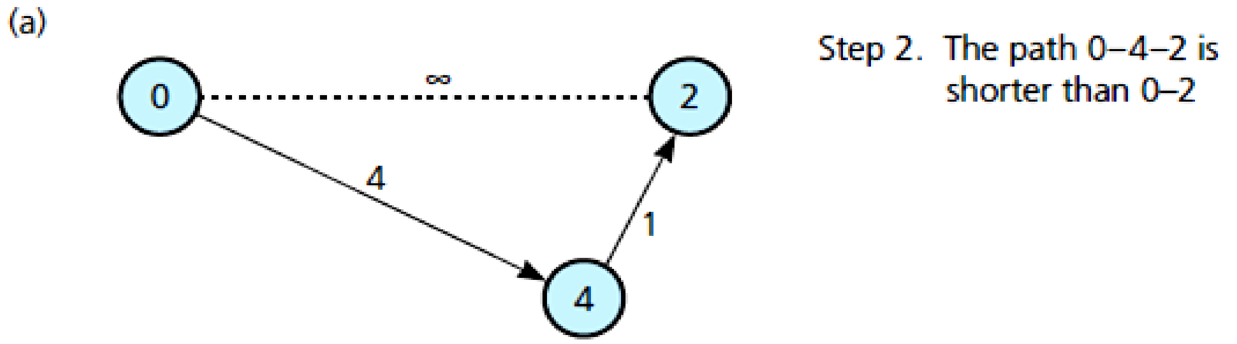
- **Problem:** given a directed graph $G = (V, E)$, a length function $length(i, j)$, $length(i, j) \geq 0$, for the edges of G, and a source vertex v.
- **Need to solve:** determine a shortest path from v to each of the remaining vertices of G.

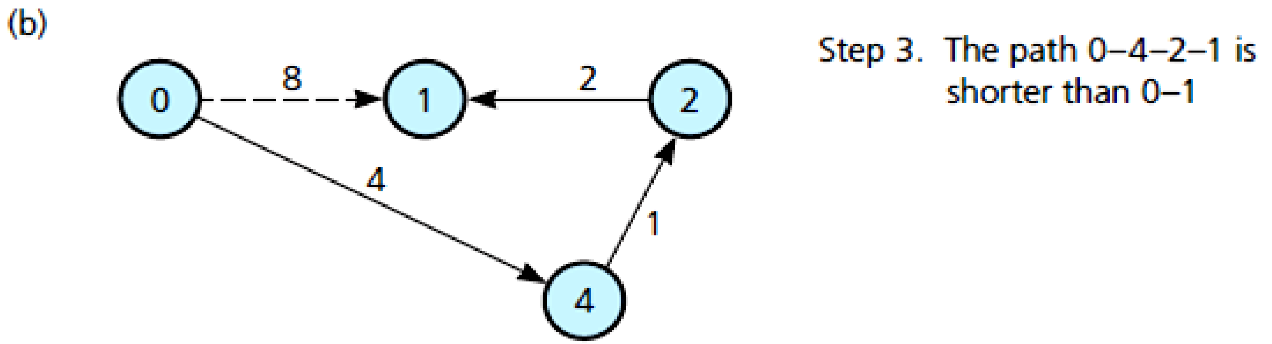
path	length
1) v0 v2	10
2) v0 v2 v3	25
3) v0 v2 v3 v1	45
4) v0 v4	45



Shortest Paths

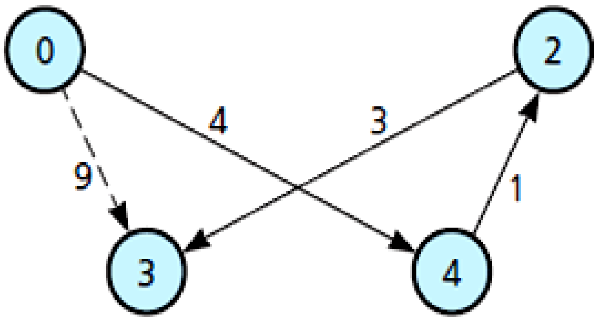
- **Weighted graph**: a graph in which each edge carries a value.





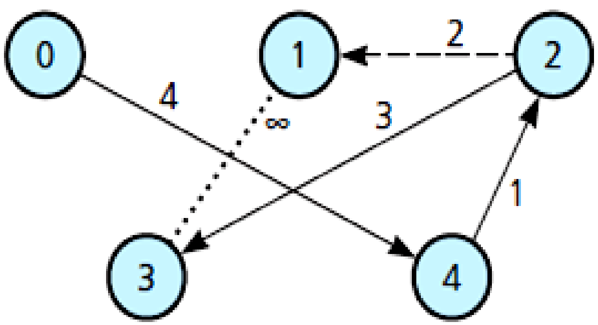
Shortest Paths

(c)



Step 3 continued. The path 0–4–2–3 is shorter than 0–3

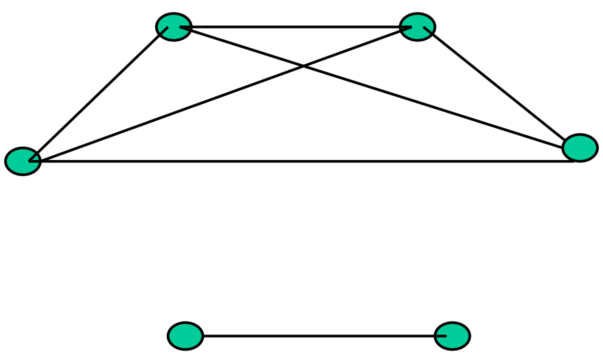
(d)



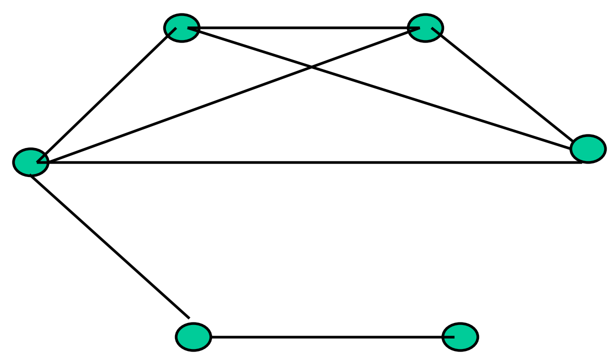
Step 4. The path 0–4–2–3 is shorter than 0–4–2–1–3

Graph Connectivity

- An undirected graph is said to be *connected* if there is a path between every pair of nodes. Otherwise, the graph is *disconnected*
- Informally, an undirected graph is connected if it hangs in one piece



Disconnected



Connected

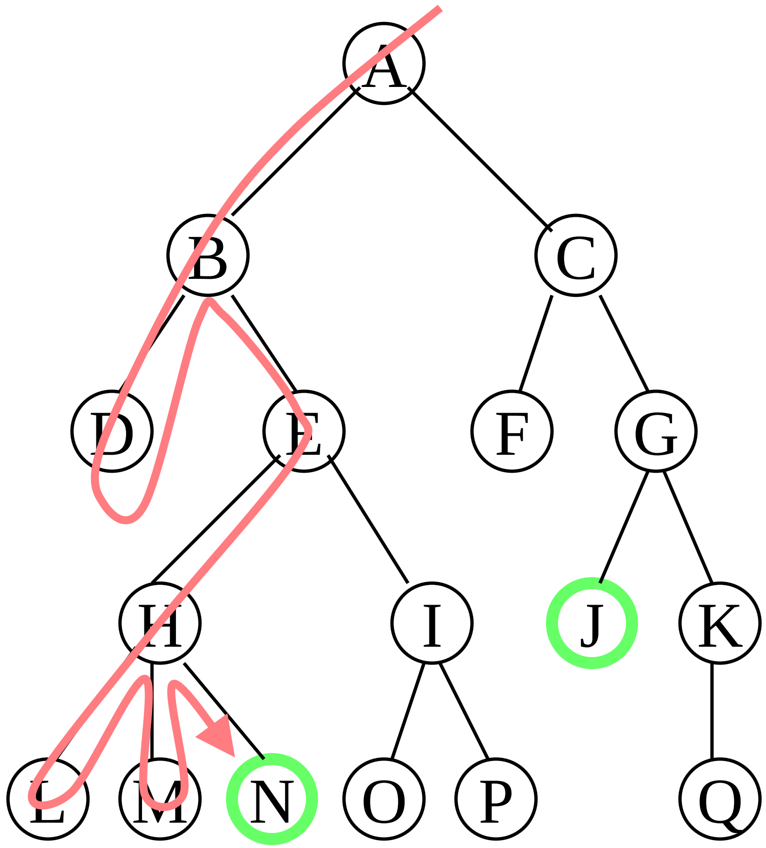
Graph Traversal Techniques

- The previous connectivity problem, as well as many other graph problems, can be solved using graph traversal techniques
- There are two standard graph traversal techniques:
  - *Depth-First Search* (DFS)
  - *Breadth-First Search* (BFS)

Graph Traversal (Contd.)

- In both DFS and BFS, the nodes of the undirected graph are visited in a systematic manner so that every node is visited exactly one.
- Both BFS and DFS give rise to a tree:
    - When a node x is visited, it is labeled as visited, and it is added to the tree
    - If the traversal got to node x from node y, y is viewed as the parent of x, and x a child of y

Depth-First Search



- A depth-first search (DFS) explores a path all the way to a leaf before backtracking and exploring another path
- For example, after searching A, then B, then D, the search backtracks and tries another path from B
- Node are explored in the order **A B D E H L M N I O P C F G J K Q**
- N will be found before J

Iterative DFS Algorithm

Iterative DFS Algorithm

The iterative algorithm uses a stack to replace the recursive calls

```
iterative DFS(Vertex v)
    mark v visited
    make an empty Stack S
    push all vertices adjacent to v onto S
    while S is not empty do
        Vertex w is pop off S
        for all Vertex u adjacent to w do
            if u is not visited then
                mark u visited
                push u onto S
```

Recursive DFS Algorithm

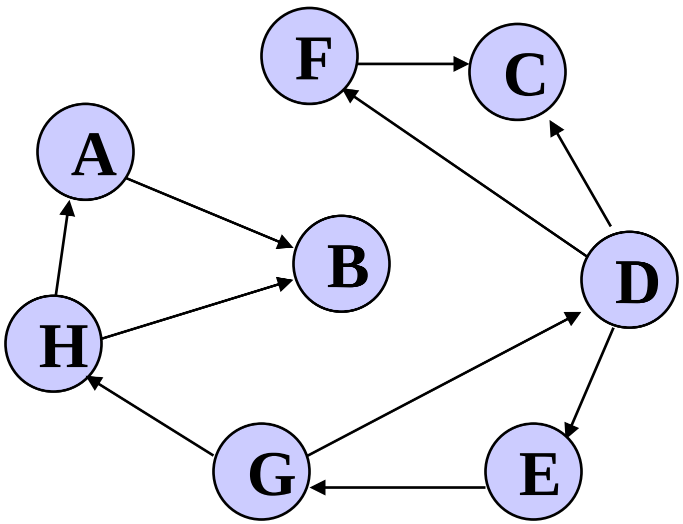
```
Algorithm DFS(graph G, Vertex v)
// Recursive algorithm

for all edges e in G.incidentEdges(v) do

    if edge e is unexplored then

        w = G.opposite(v, e)
        if vertex w is unexplored then
          label e as discovery edge
          recursively call DFS(G, w)
        else
          label e a back edge
```

Walk-Through

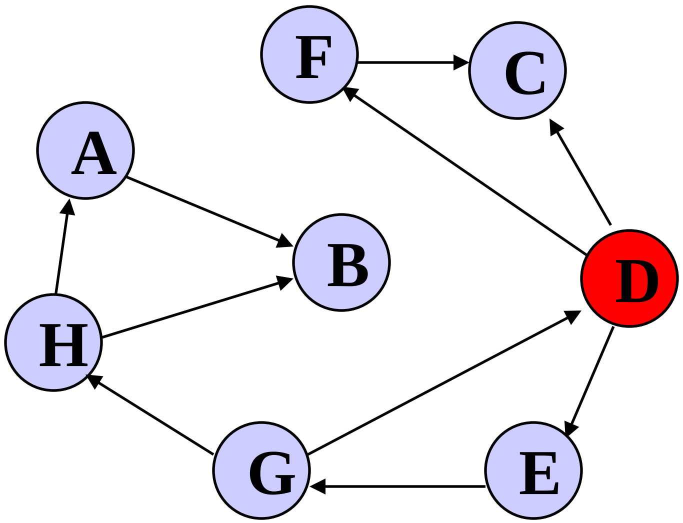


Visited Array

A	
B	
C	
D	
E	
F	
G	
H	

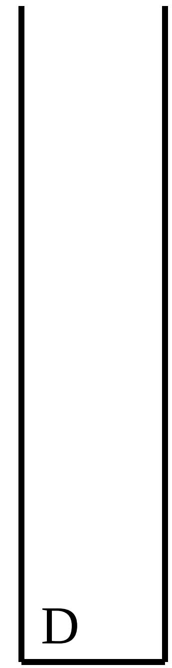
Task: Conduct a depth-first search of the graph starting with node D

Walk-Through



Visited Array

A	
B	
C	
D	√
E	
F	
G	
H	

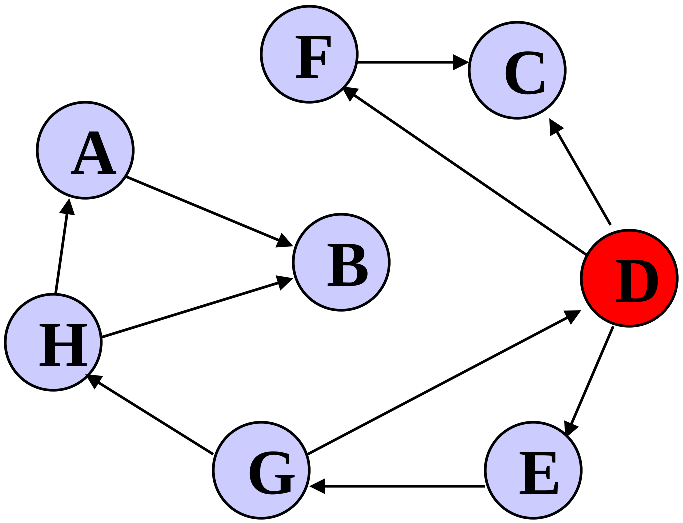


The order nodes are visited:

D

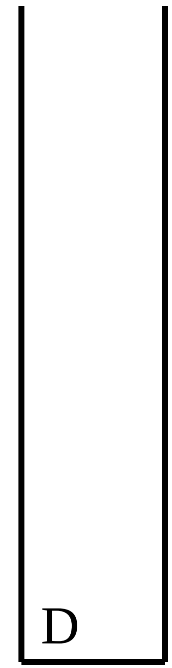
Visit D

Walk-Through



Visited Array

A	
B	
C	
D	√
E	
F	
G	
H	

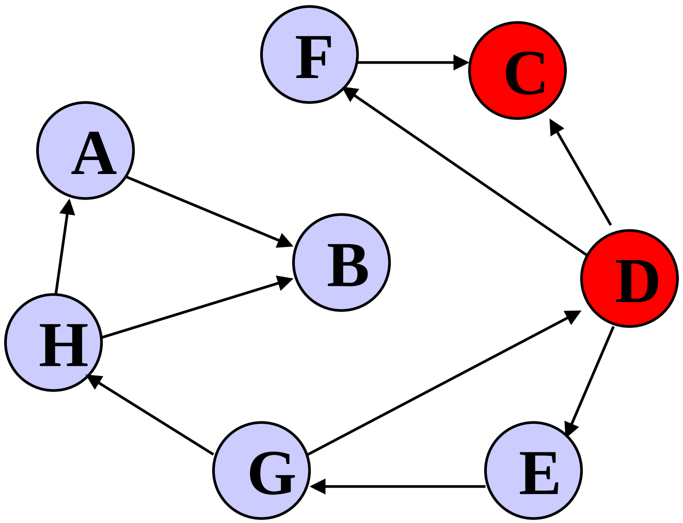


The order nodes are visited:

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	

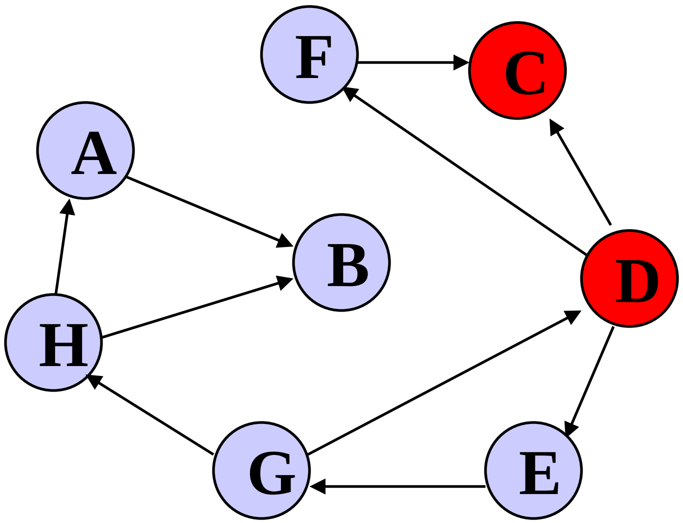
C
D

Visit C

The order nodes are visited:

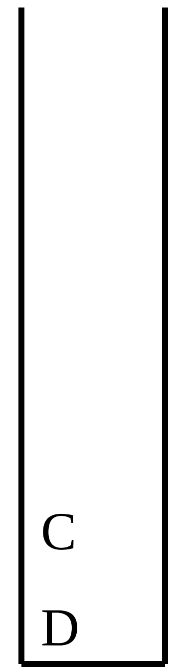
D, C

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	
F	
G	
H	

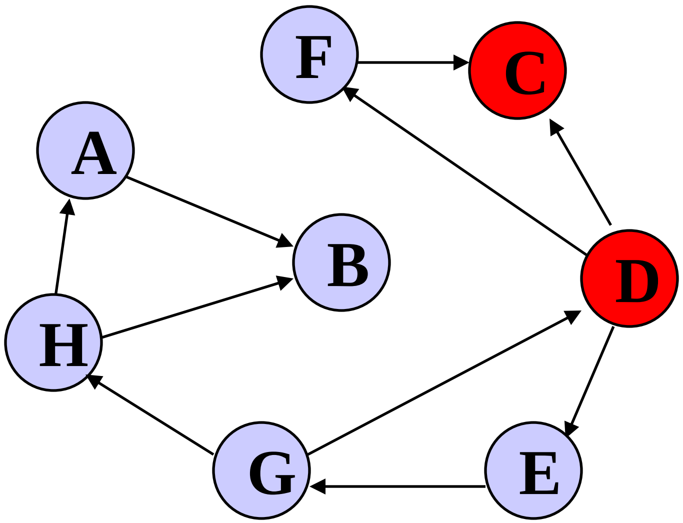


The order nodes are visited:

D, C

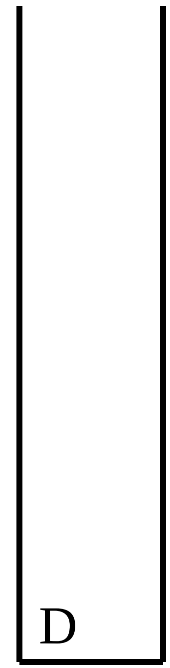
No nodes adjacent to C; cannot continue ➔ *backtrack*, i.e., pop stack and restore previous state

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	
F	
G	
H	

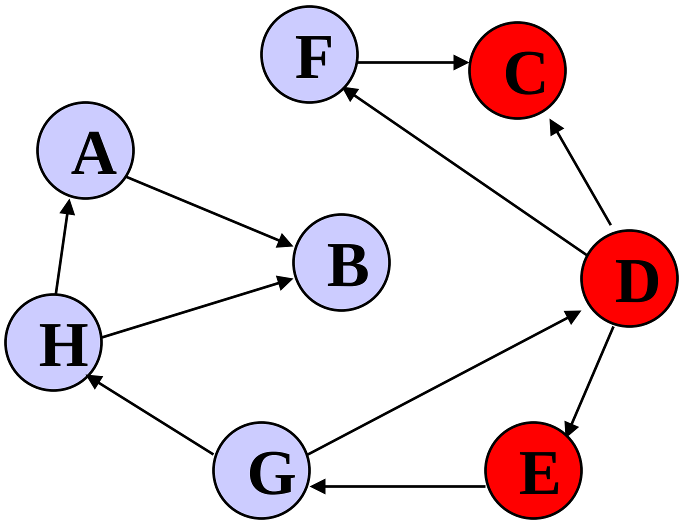


The order nodes are visited:

D, C

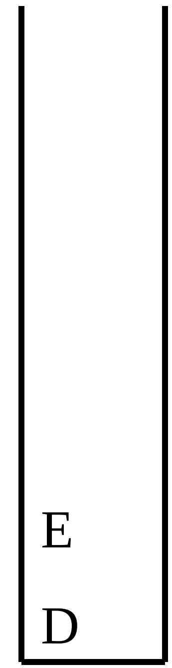
Back to D – C has been visited, decide to visit E next

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	
H	

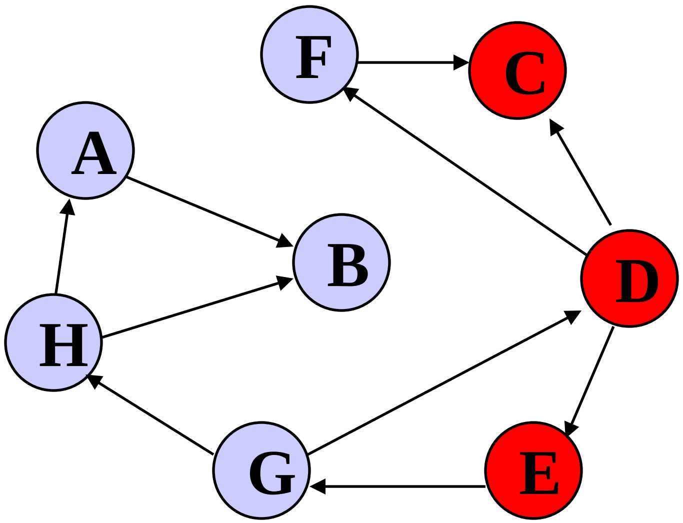


The order nodes are visited:

D, C, E

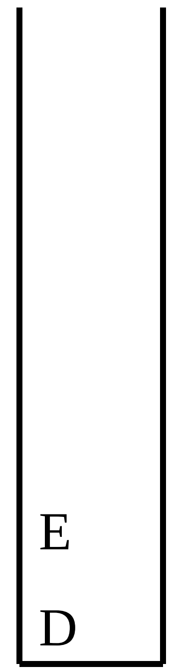
Back to D – C has been visited, decide to visit E next

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	
H	

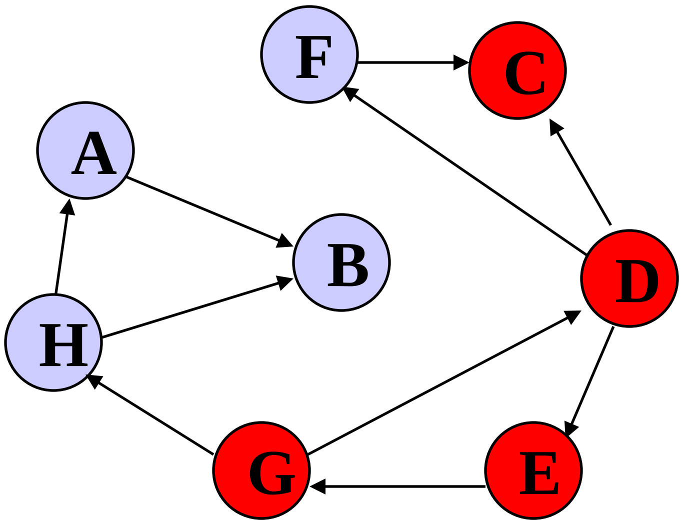


The order nodes are visited:

D, C, E

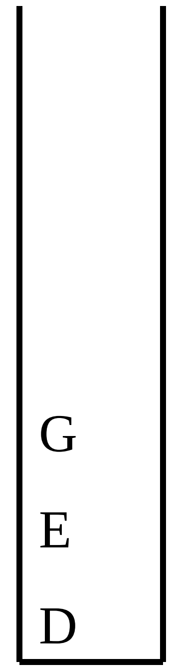
Only G is adjacent to E

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	√
H	

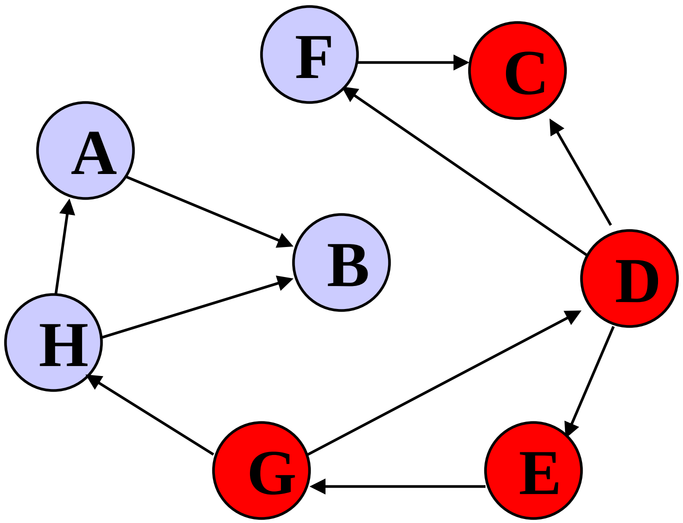


Visit G

The order nodes are visited:

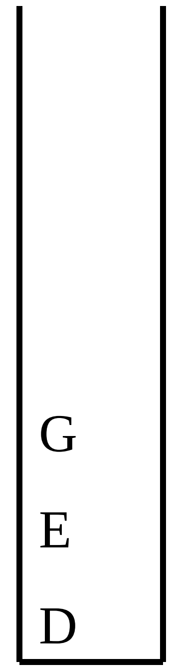
D, C, E, G

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	√
H	

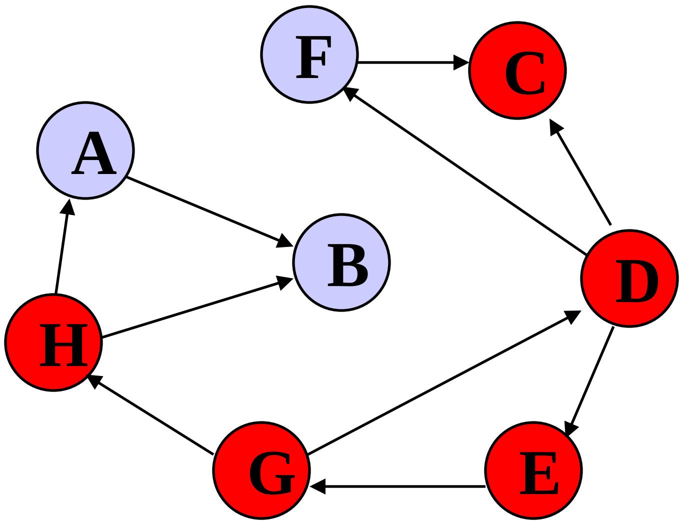


The order nodes are visited:

D, C, E, G

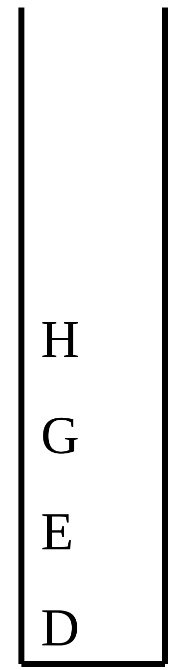
Nodes D and H are adjacent to G. D has already been visited. Decide to visit H.

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	√
H	√

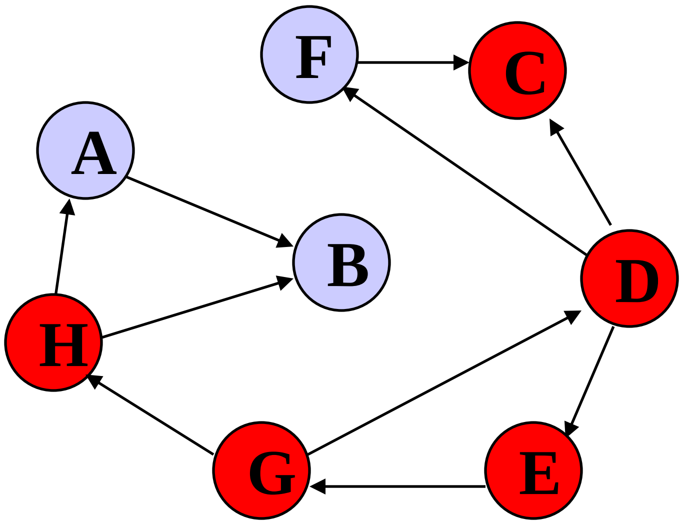


Visit H

The order nodes are visited:

D, C, E, G, H

Walk-Through

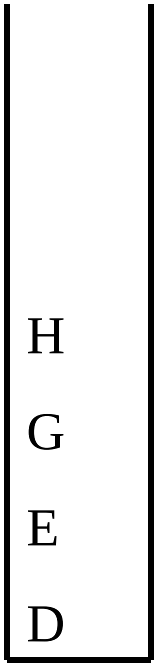


The order nodes are visited:

D, C, E, G, H

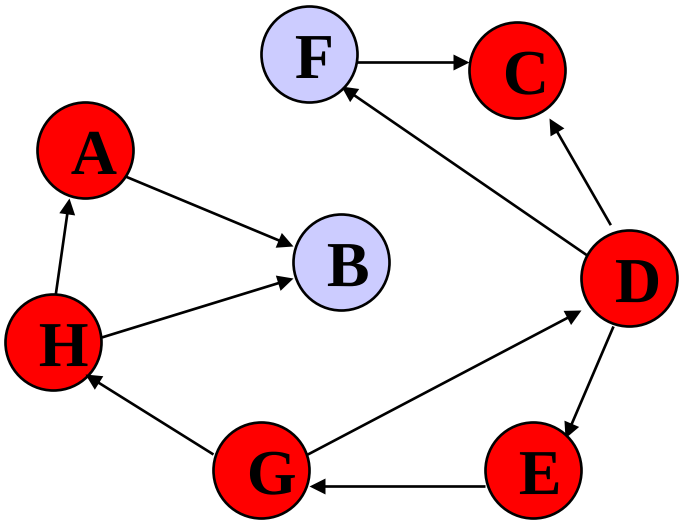
Visited Array

A	
B	
C	√
D	√
E	√
F	
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	√

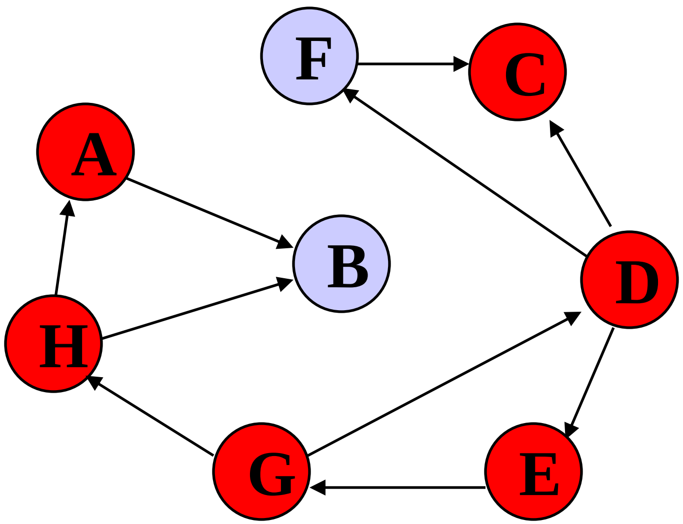
A
H
G
E
D

The order nodes are visited:

D, C, E, G, H, A

Visit A

Walk-Through



Visited Array

A	√
B	
C	√
D	√
E	√
F	
G	√
H	√

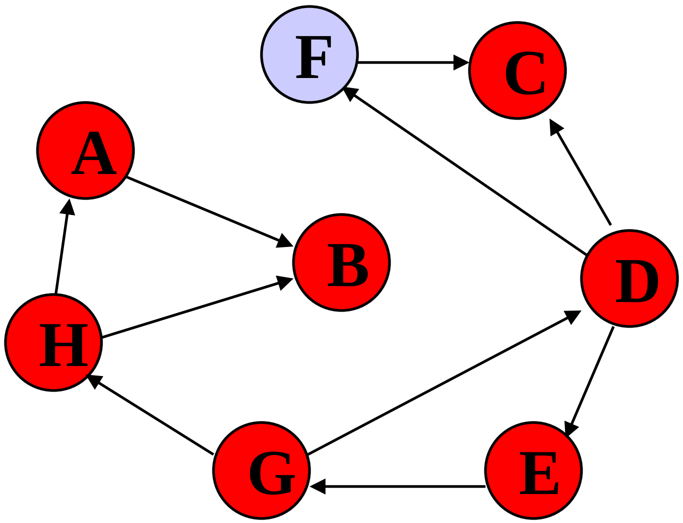
A
H
G
E
D

The order nodes are visited:

D, C, E, G, H, A

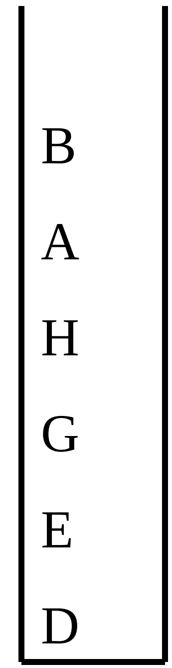
Only Node B is adjacent to A. Decide to visit B next.

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

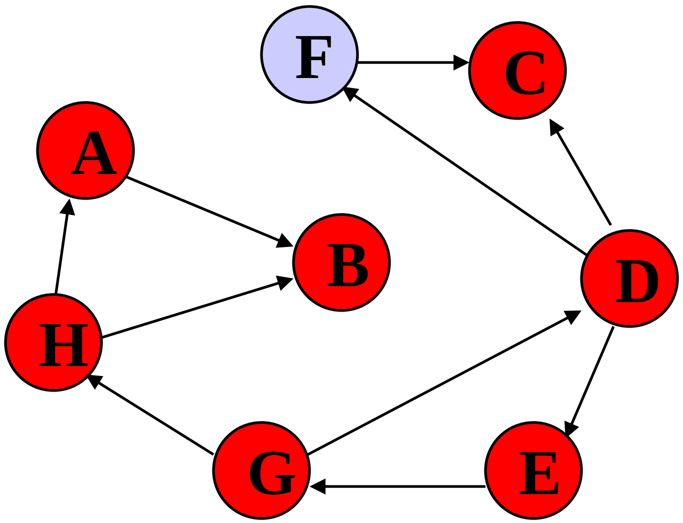


Visit B

The order nodes are visited:

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

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

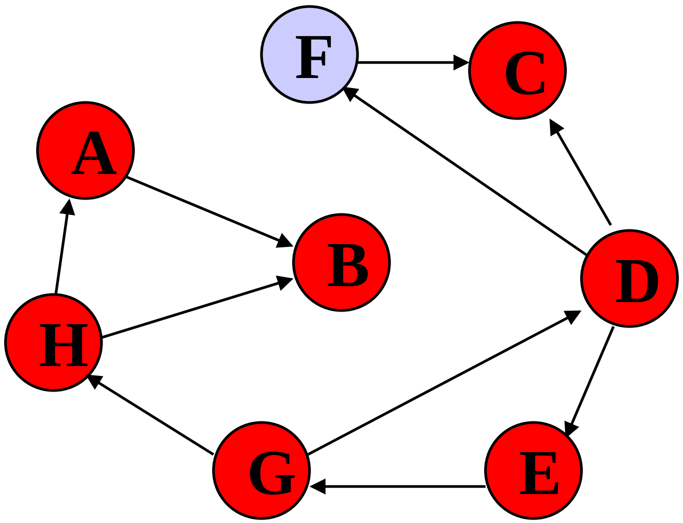
A
H
G
E
D

The order nodes are visited:

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

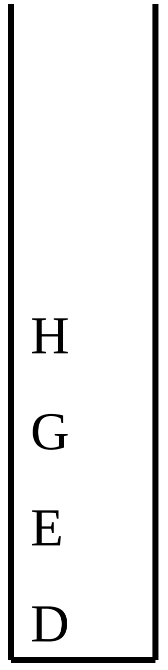
No unvisited nodes adjacent to B. Backtrack (pop the stack).

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

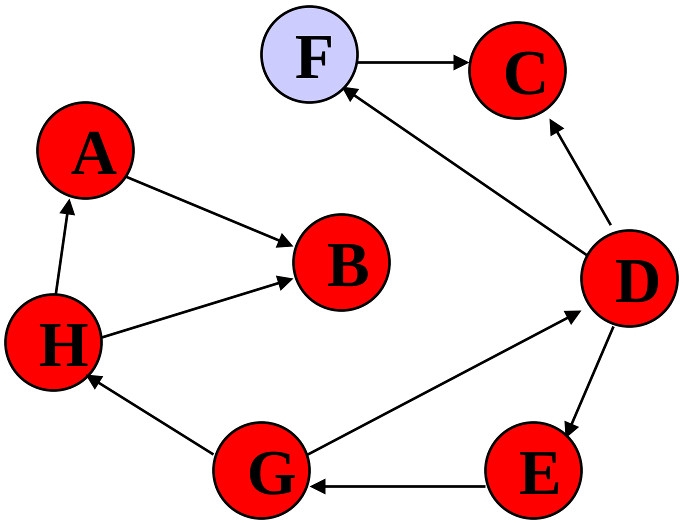


The order nodes are visited:

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

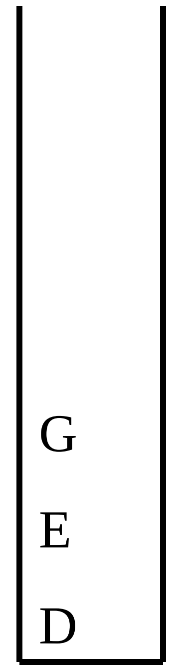
No unvisited nodes adjacent to A. Backtrack (pop the stack).

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

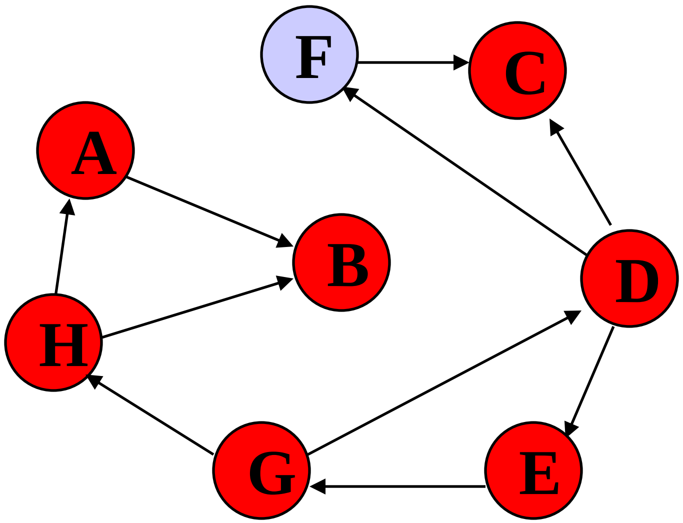


The order nodes are visited:

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

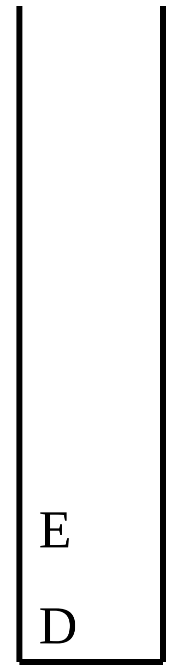
No unvisited nodes adjacent to H. Backtrack (pop the stack).

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

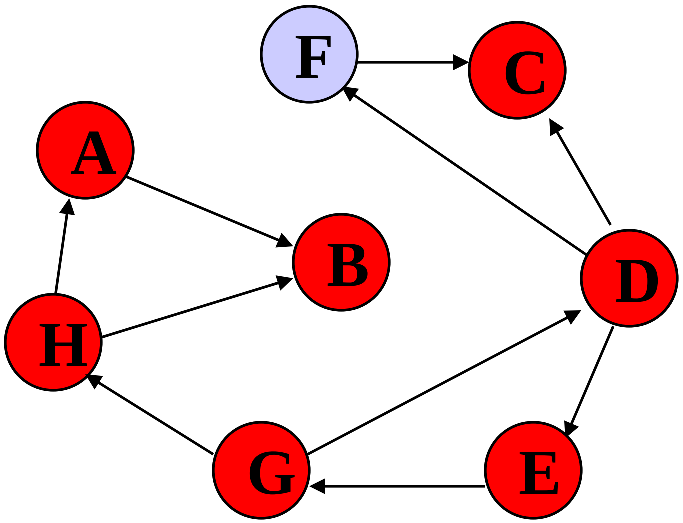


The order nodes are visited:

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

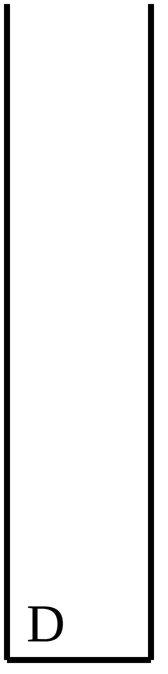
No unvisited nodes adjacent to G. Backtrack (pop the stack).

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

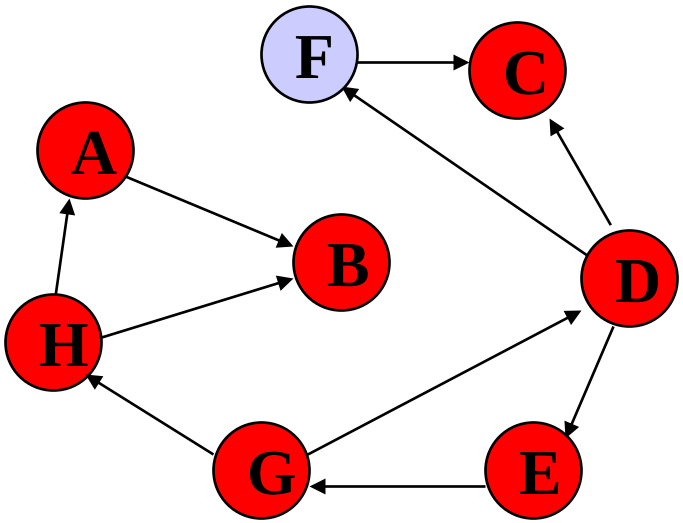


The order nodes are visited:

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

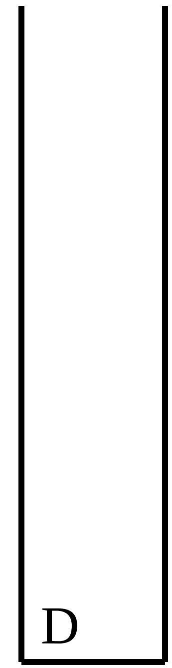
No unvisited nodes adjacent to E. Backtrack (pop the stack).

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

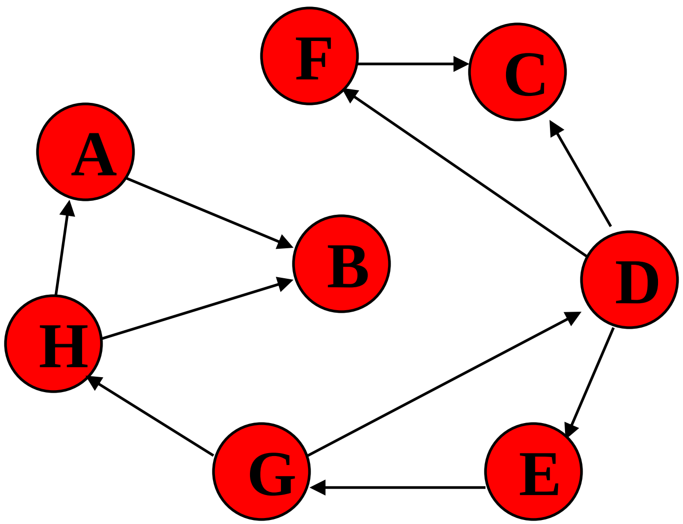


The order nodes are visited:

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

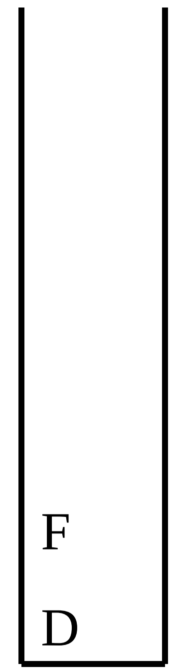
F is unvisited and is adjacent to D. Decide to visit F next.

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

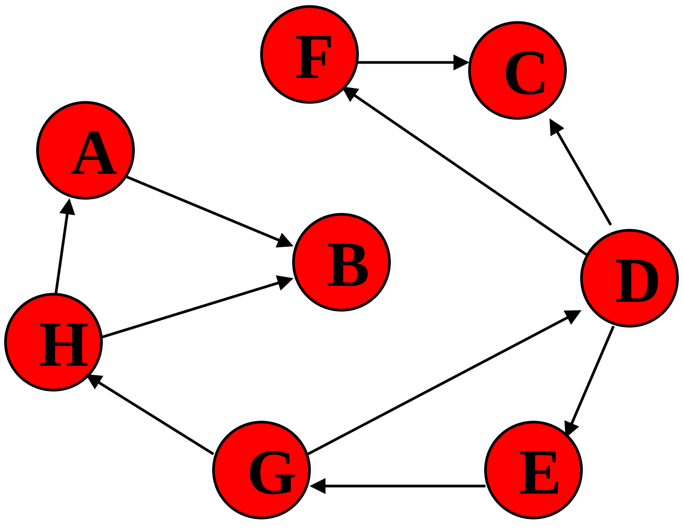


The order nodes are visited:

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

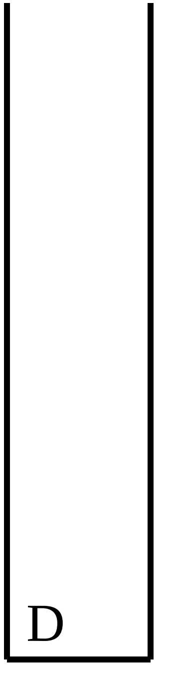
Visit F

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

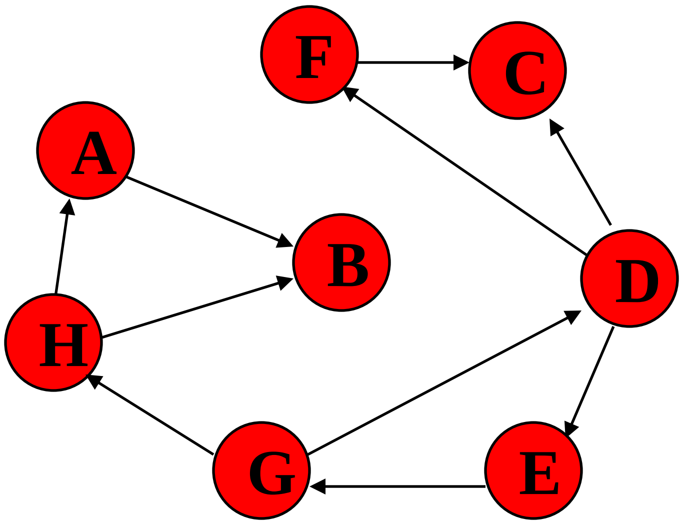


The order nodes are visited:

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

No unvisited nodes adjacent to F. Backtrack.

Walk-Through



Visited Array

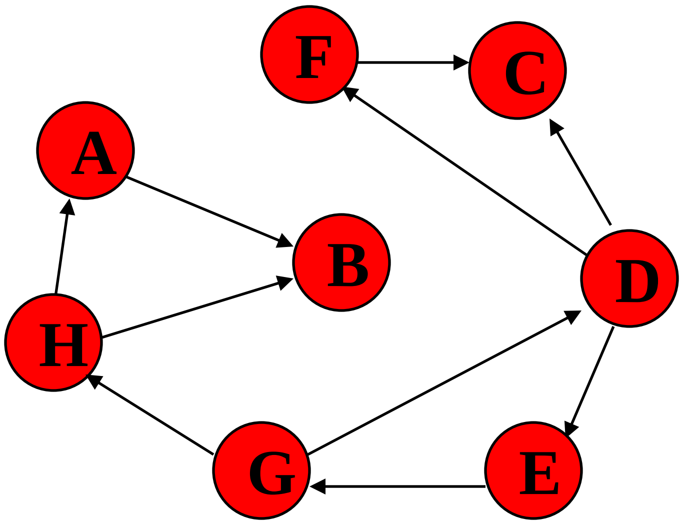
A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

The order nodes are visited:

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

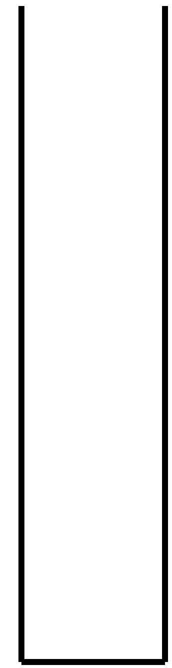
No unvisited nodes adjacent to D. Backtrack.

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.

Thank you

???