Graphs

dont be scared it only contains dfs,bfs

What is a Graph?

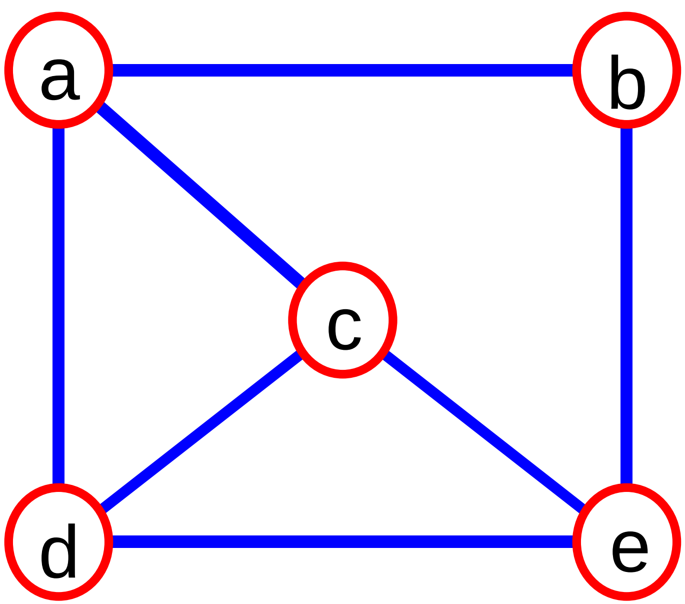
- A graph G = (V,E) is composed of:

  V: a finite, nonempty set of vertices

  E: set of edges connecting the vertices in V
- An edge e = (u,v) is a pair of vertices
- Example:

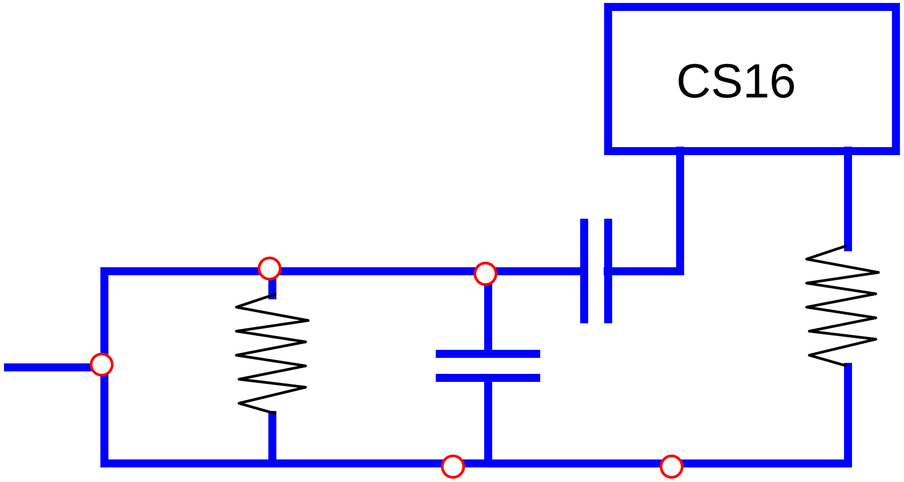
V= {a,b,c,d,e}

E= {(a,b), (a,c), (a,d), (b,e), (c,d), (c,e), (d,e)}

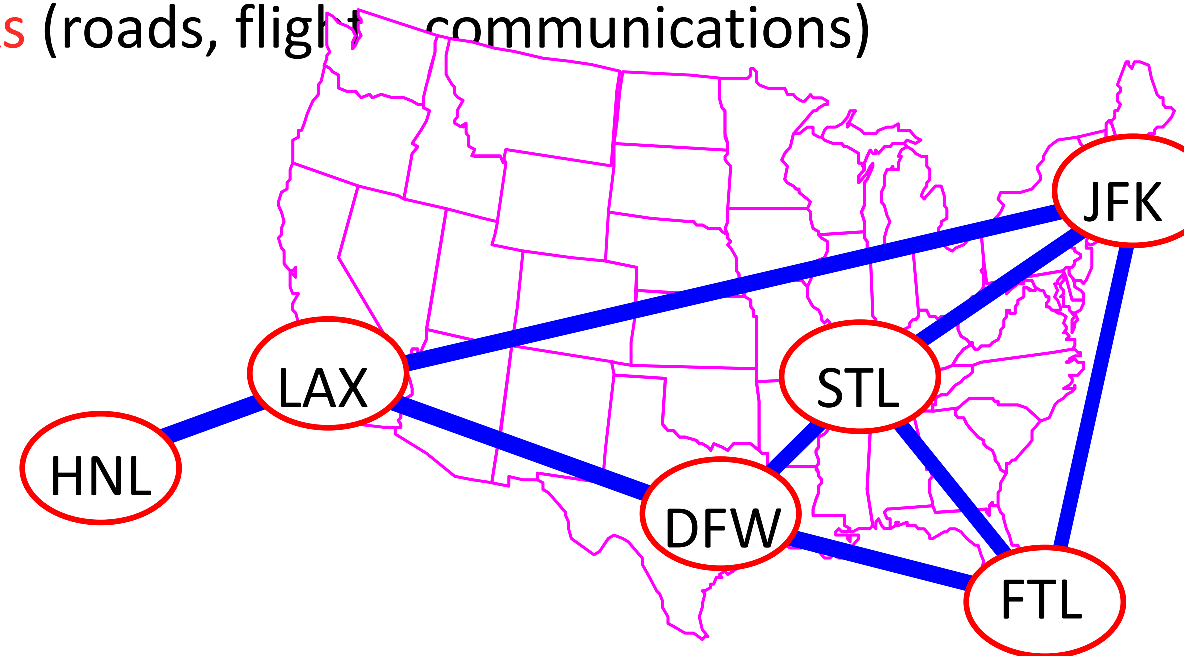


Applications

- Electronic circuits

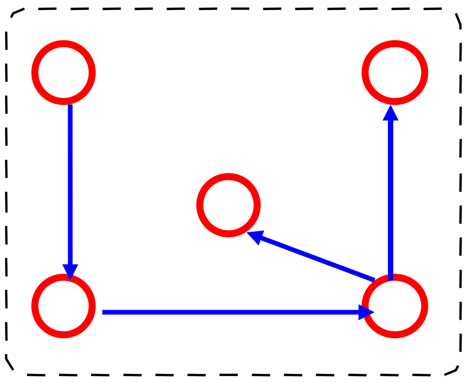


- Networks (roads, fligh t communications)



Directed Graph

- A graph where edges are directed



Directed vs. Undirected Graph

- An undirected graph is one in which the pair of vertices in an edge is unordered, $(v_0, v_1) = (v_1, v_0)$
- A directed graph is one in which each edge is a directed pair of vertices, $\langle v_0, v_1 \rangle \mathrel{!=} \langle v_1, v_0 \rangle$

$\text{tail} \longrightarrow \text{head}$

Terminology:
Adjacent and Incident

- If (v_0, v_1) is an edge in an undirected graph,
  - v_0 and v_1 are adjacent
  - The edge (v_0, v_1) is incident on vertices v_0 and v_1
- If (v_0, v_1) is an edge in a directed graph
  - v_0 is adjacent to v_1, and v_1 is adjacent from v_0
  - The edge (v_0, v_1) is incident on v_0 and v_1

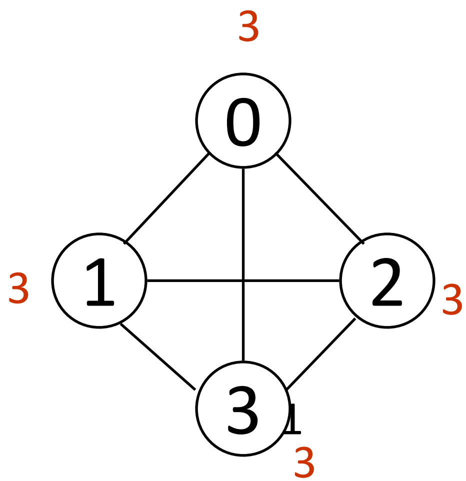
Terminology: Degree of a Vertex

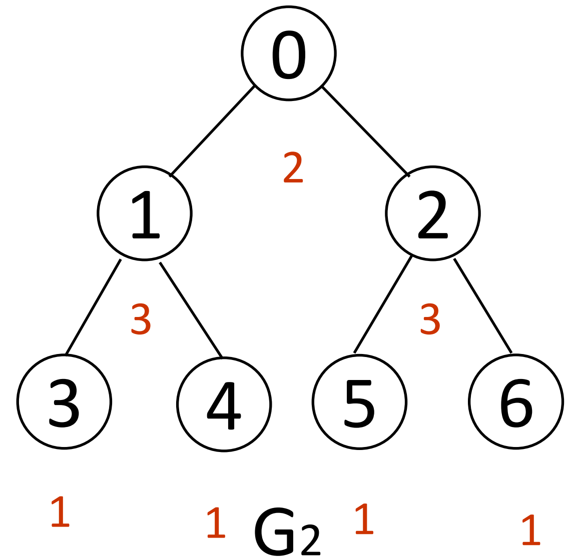
- The degree of a vertex is the number of edges incident to that vertex
- For directed graph,
  - the in-degree of a vertex *v* is the number of edges that have *v* as the head
  - the out-degree of a vertex *v* is the number of edges that have *v* as the tail
  - if d_i is the degree of a vertex *i* in a graph *G* with *n* vertices and *e* edges, the number of edges is

$$e = \left(\sum_{0}^{n-1} d_i\right) / 2$$

Why? Since adjacent vertices each count the adjoining edge, it will be counted twice

Examples

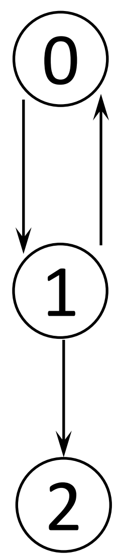




G2

directed graph

in-degree
out-degree



in:1, out: 1

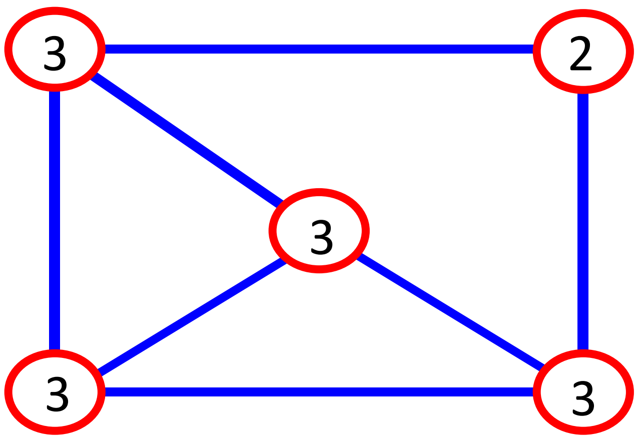
in: 1, out: 2

in: 1, out: 0

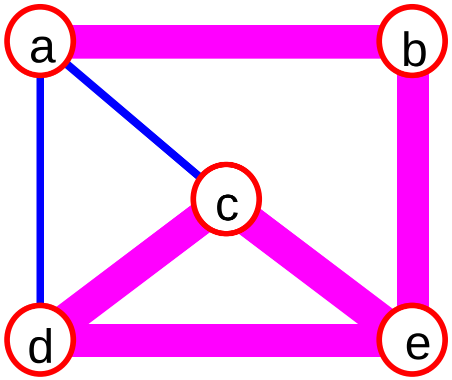
G3

Terminology: Path

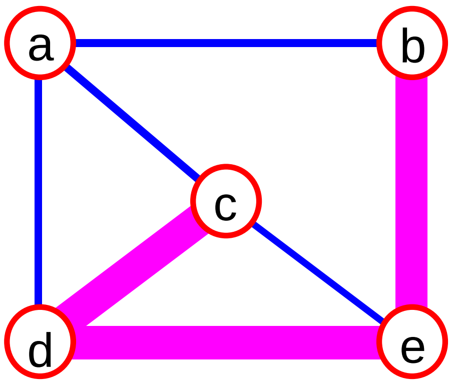
- path: sequence of vertices $v_1, v_2, \ldots v_k$ such that consecutive vertices v_i and v_{i+1} are adjacent.



Not a PATH a c b e



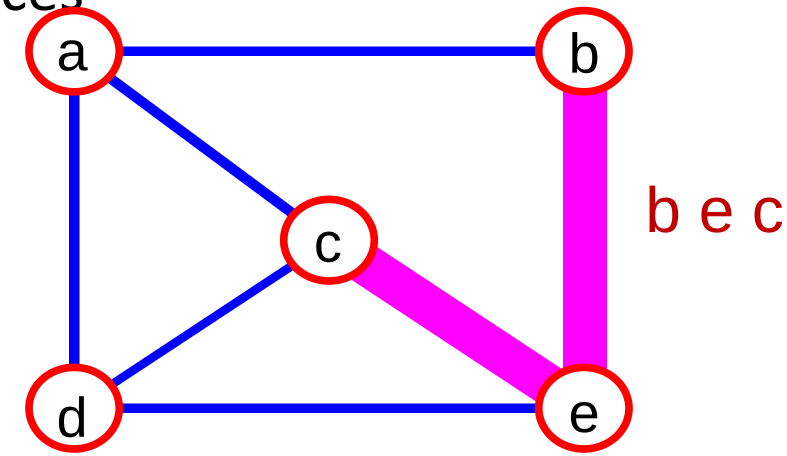
a b e d c e



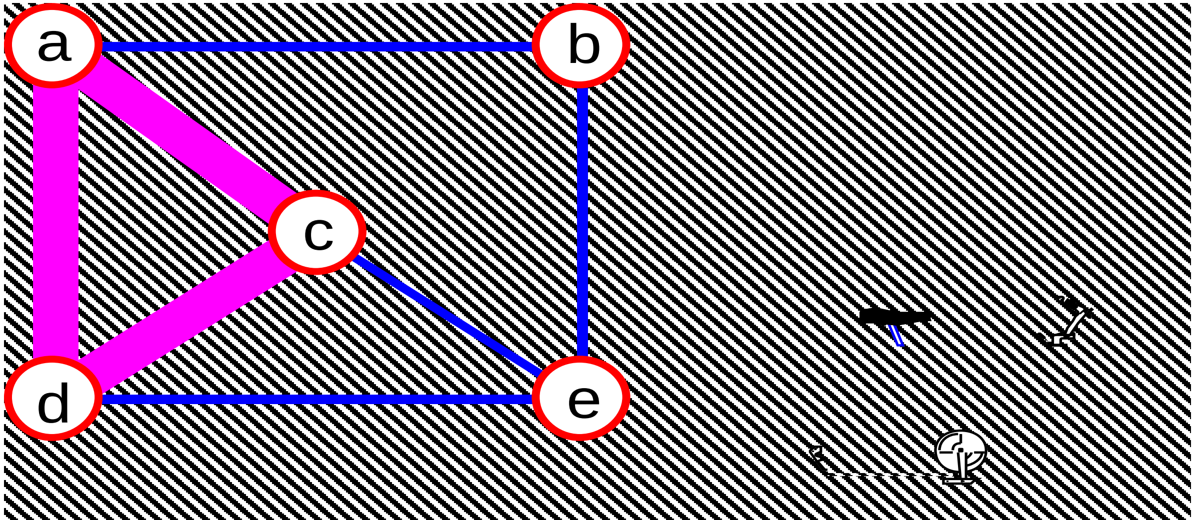
b e d c

More Terminology

- simple path: no repeated vertices

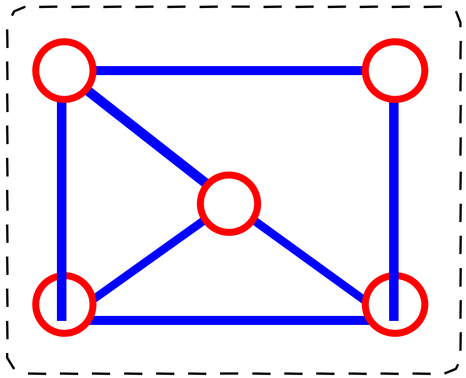


- cycle: simple path, except that the last vertex is the same as the first vertex

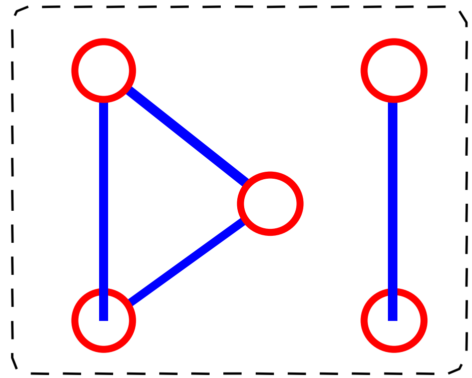


Even More Terminology

- Connected graph: any two vertices are connected by some path

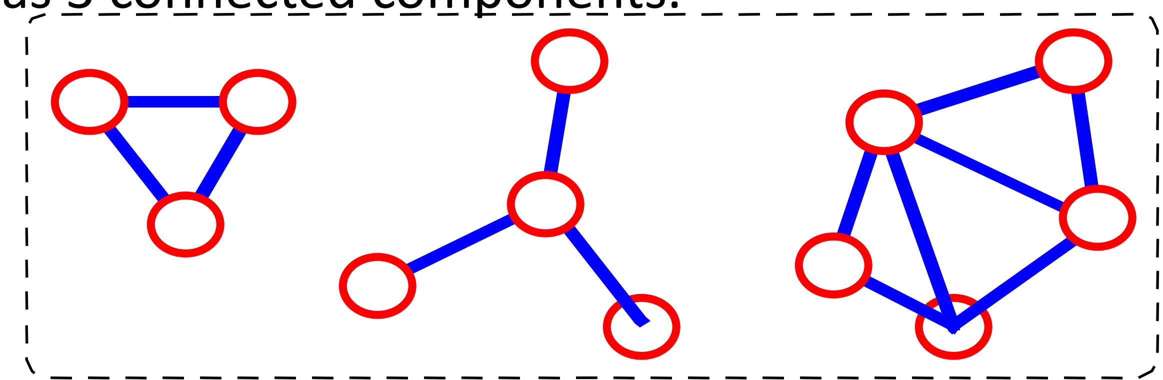


connected

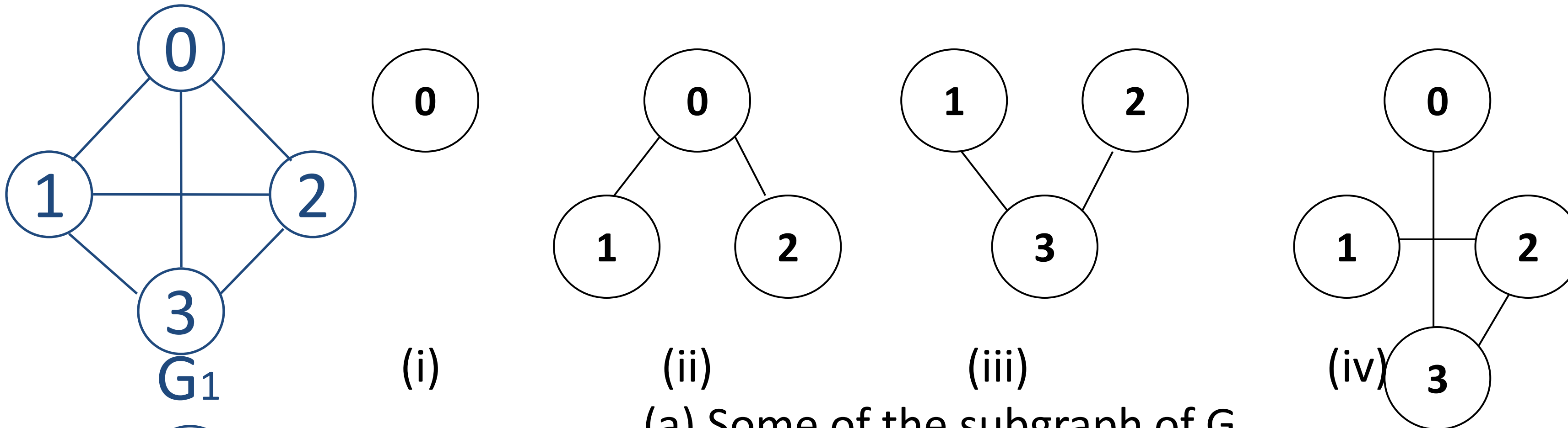


not connected

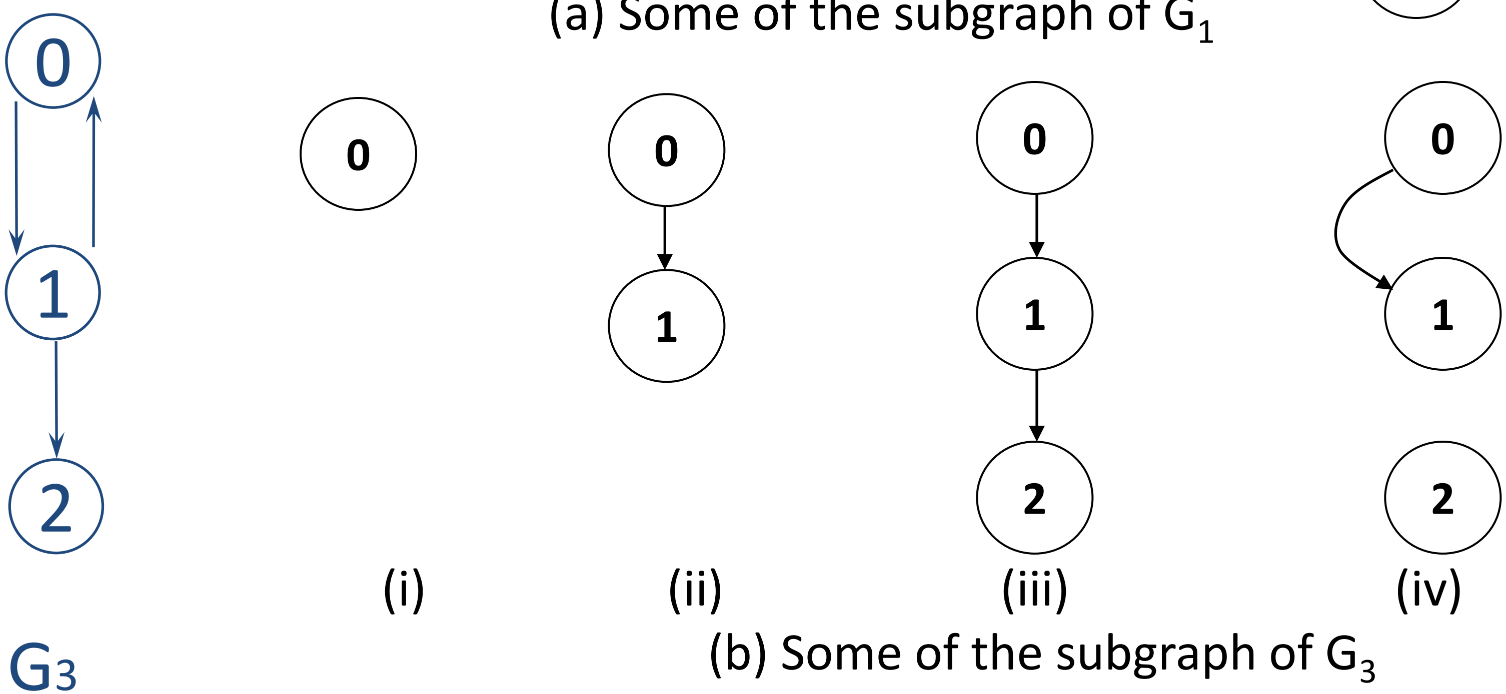
- Subgraph: subset of vertices and edges forming a graph
- Connected component: maximal connected subgraph. E.g., the graph below has 3 connected components.



Subgraphs Examples



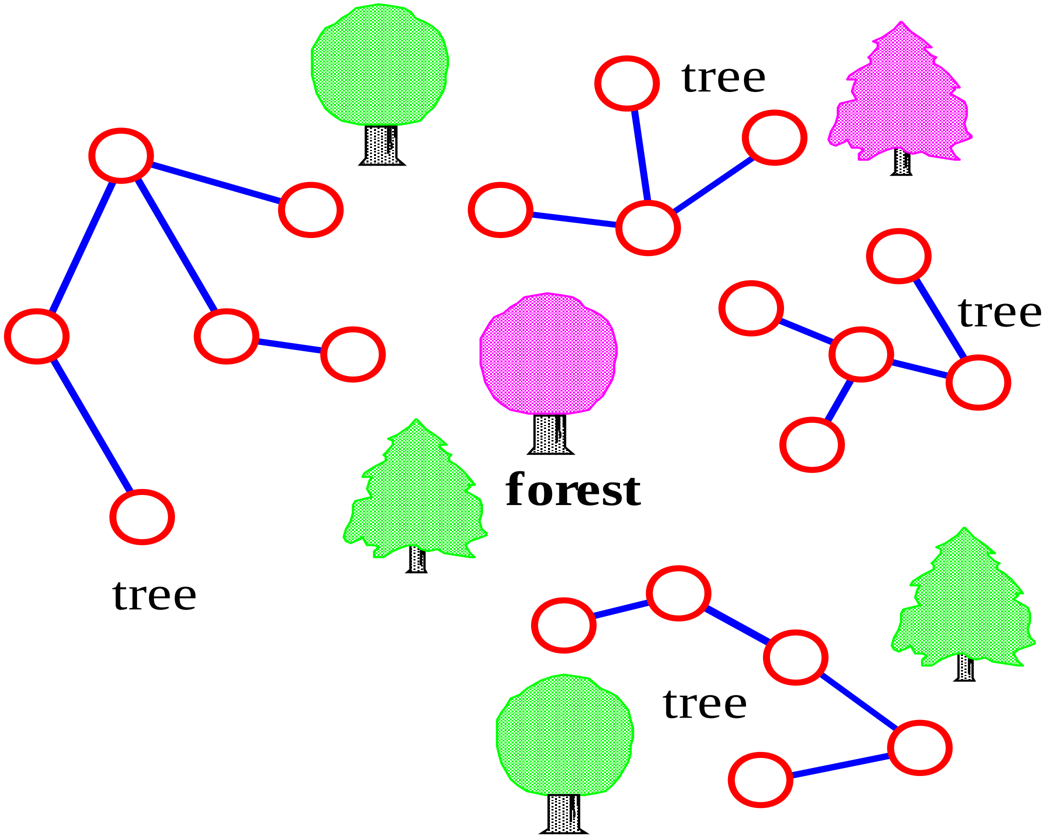
(a) Some of the subgraph of G_1



(b) Some of the subgraph of G_3

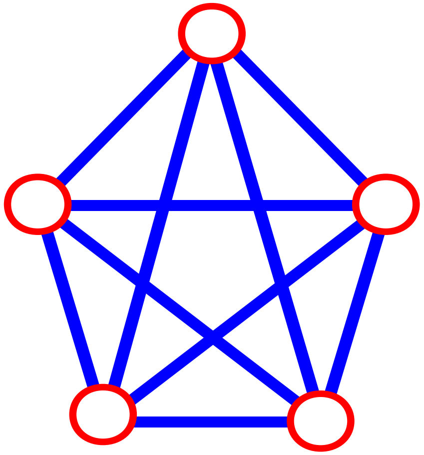
More…

- tree - connected graph without cycles
- forest - collection of trees



Connectivity

- Let **n** = #vertices, and **m** = #edges
- **A complete graph**: one in which all pairs of vertices are adjacent
- *How many total edges in a complete graph?*
  - Each of the n vertices is incident to **n**-1 edges, however, we would have counted each edge twice! Therefore, intuitively, m = **n**(**n** -1)/2.
- Therefore, if a graph is not complete, m < **n**(**n** -1)/2



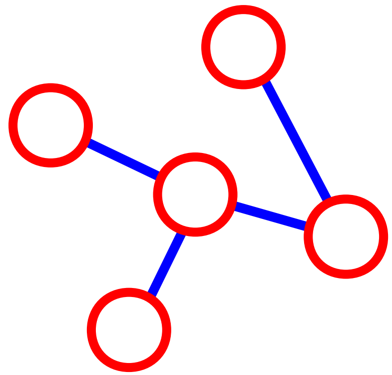
$\mathbf{n} = 5$
$\mathbf{m} = (5 * 4)/2 = 10$

More Connectivity

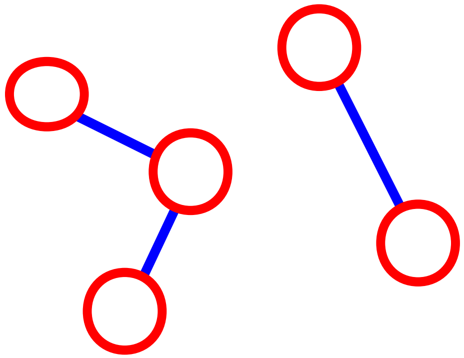
n = #vertices

m = #edges

- For a tree **m** = **n** - 1



If **m** < **n** - 1, G is not connected



ADT for Graph

objects: a nonempty set of vertices and a set of undirected edges, where each edge is a pair of vertices

functions: for all *graph* $\in$ *Graph*, *v*, v_1 and $v_2 \in$ *Vertices*

Graph Create()::=return an empty graph

Graph InsertVertex(*graph*, *v*)::= return a graph with *v* inserted. *v* has no incident edge.

Graph InsertEdge(*graph*, v_1,v_2)::= return a graph with new edge between v_1 and v_2

Graph DeleteVertex(*graph*, *v*)::= return a graph in which *v* and all edges incident to it are removed

Graph DeleteEdge(*graph*, v_1, v_2)::=return a graph in which the edge (v_1, v_2) is removed

Boolean IsEmpty(*graph*)::= if (*graph*==*empty graph*) return TRUE else return FALSE

List Adjacent(*graph*,*v*)::= return a list of all vertices that are adjacent to *v*

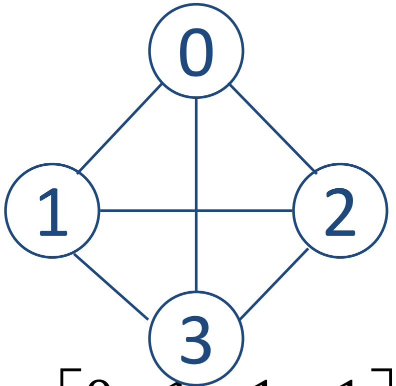
Graph Representations

- Adjacency Matrix
- Adjacency Lists

Data Structures for Graphs
An Adjacency Matrix

- Let G=(V,E) be a graph with n vertices.
- The adjacency matrix of G is a two-dimensional n by n array, say adj_mat
- If the edge (v_i, v_j) is in E(G), adj_mat[i][j]=1
- If there is no such edge in E(G), adj_mat[i][j]=0

Examples for Adjacency Matrix

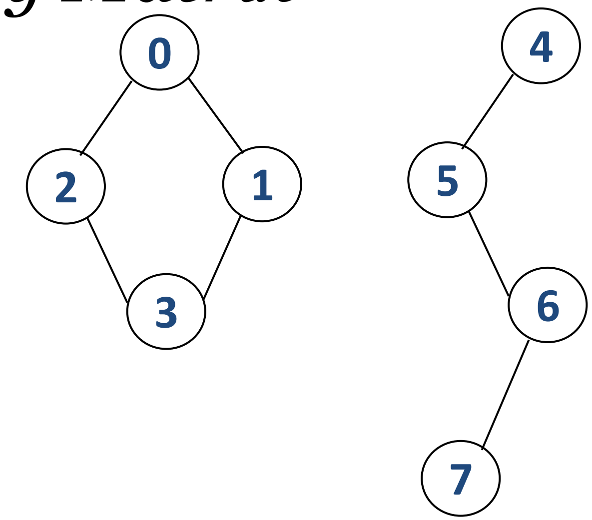


$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

G_1

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

G_2



$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

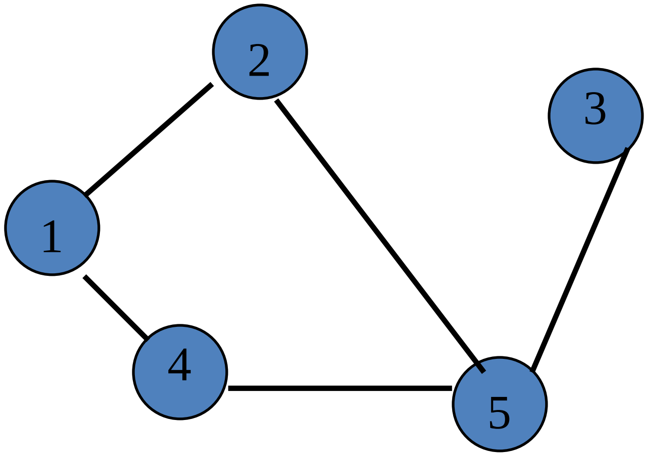
G_4

symmetric

undirected: $n^2/2$

directed: n^2

Adjacency Matrix Properties



	1	2	3	4	5
1	0	1	0	1	0
2	1	0	0	0	1
3	0	0	0	0	1
4	1	0	0	0	1
5	0	1	1	1	0

- Diagonal entries are zero
- The adjacency matrix of an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric

Adjacency Matrix

- The degree of a vertex i is $\sum_{j=0}^{n-1} A[i][j]$
- For a digraph (= directed graph), the row sum is the out_degree, while the column sum is the in_degree of a vertex i

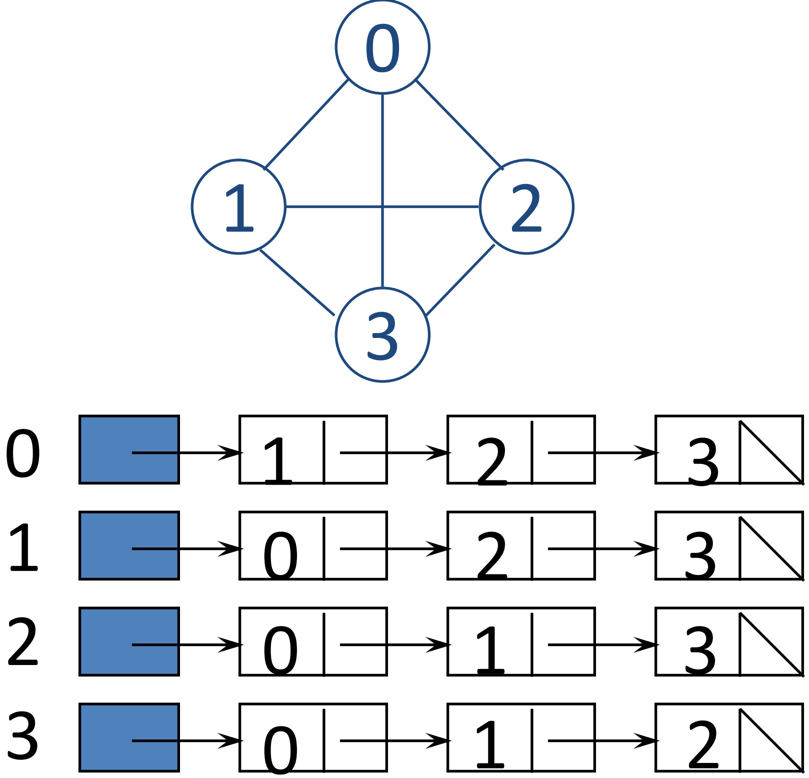
$$ind(v_i) = \sum_{j=0}^{n-1} A[j][i] \qquad outd(v_i) = \sum_{j=0}^{n-1} A[i][j]$$

Adjacency Matrix

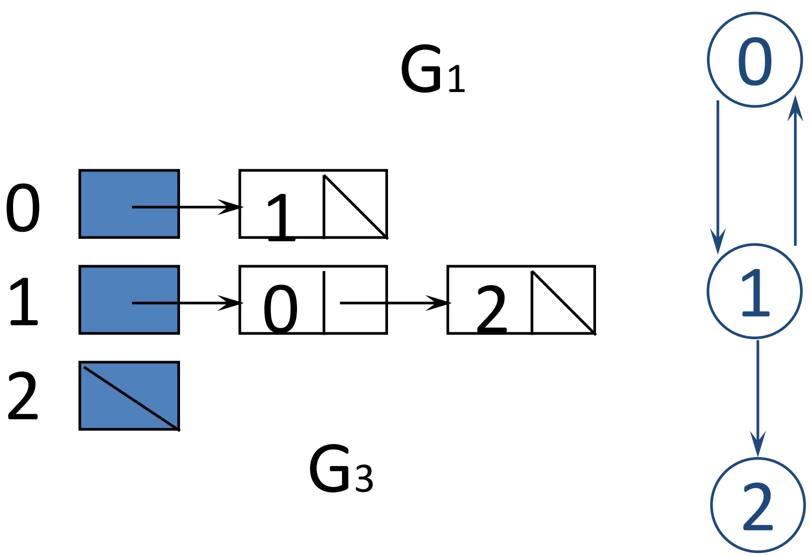
- n^2 bits of space
- All algos will require at least $O(n^2)$ time to find edges in G as n^2-n entries of the matrix have to be examined (diagonal entries are zero)
- For an undirected graph, may store only lower or upper triangle (exclude diagonal)
  - $(n-1)n/2$ bits
- $O(n)$ time to find vertex degree and/or vertices adjacent to a given vertex
- Sparse graphs: problem
  - Speed up is possible through the use of linked lists in which only the edges that are in G are represented

Data Structures for Graphs
An Adjacency List

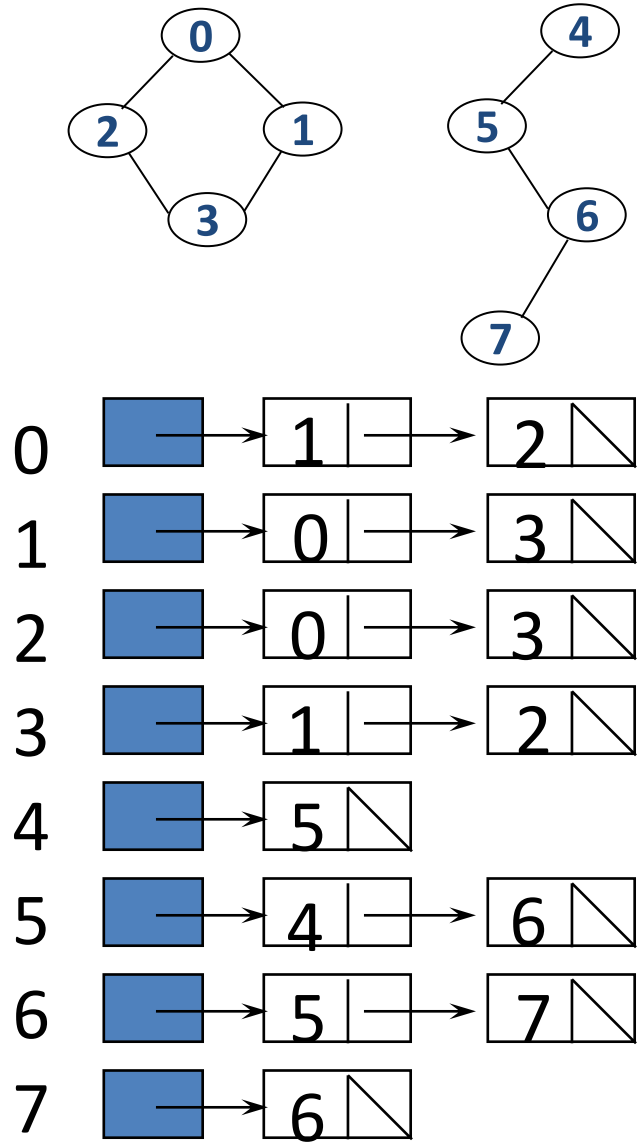
- A list of pointers, one for each node of the graph
- These pointers are the start of a linked list of nodes that can be reached by one edge of the graph
- For a weighted graph, this list would also include the weight for each edge



G1



G3



G4

An undirected graph with n vertices and e edges ==> n head nodes and 2e list nodes

Adjacency Lists (data structures)

Each row in adjacency matrix is represented as an adjacency list.

```
#define MAX_VERTICES 50
typedef struct node {
     int vertex_id;
     struct node *link;
};
typedef struct node *node_pointer;
node_pointer graph[MAX_VERTICES];
```

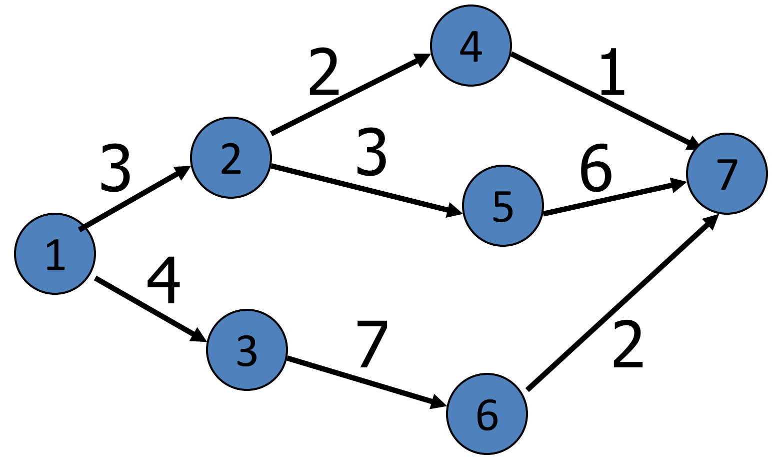
VERTEX_ID	LINK

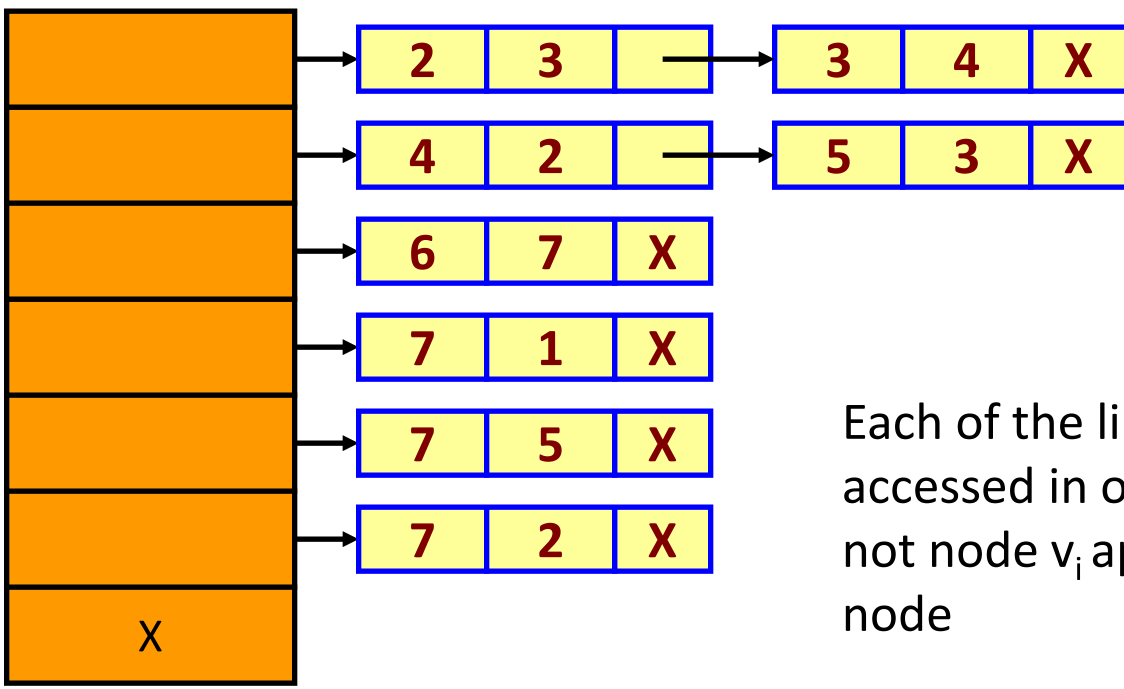
Node Structure

Adjacency Lists

- Consider a weighted graph

Node Structure





Each of the linked lists must be accessed in order to detect whether or not node v_i appears as a destination node

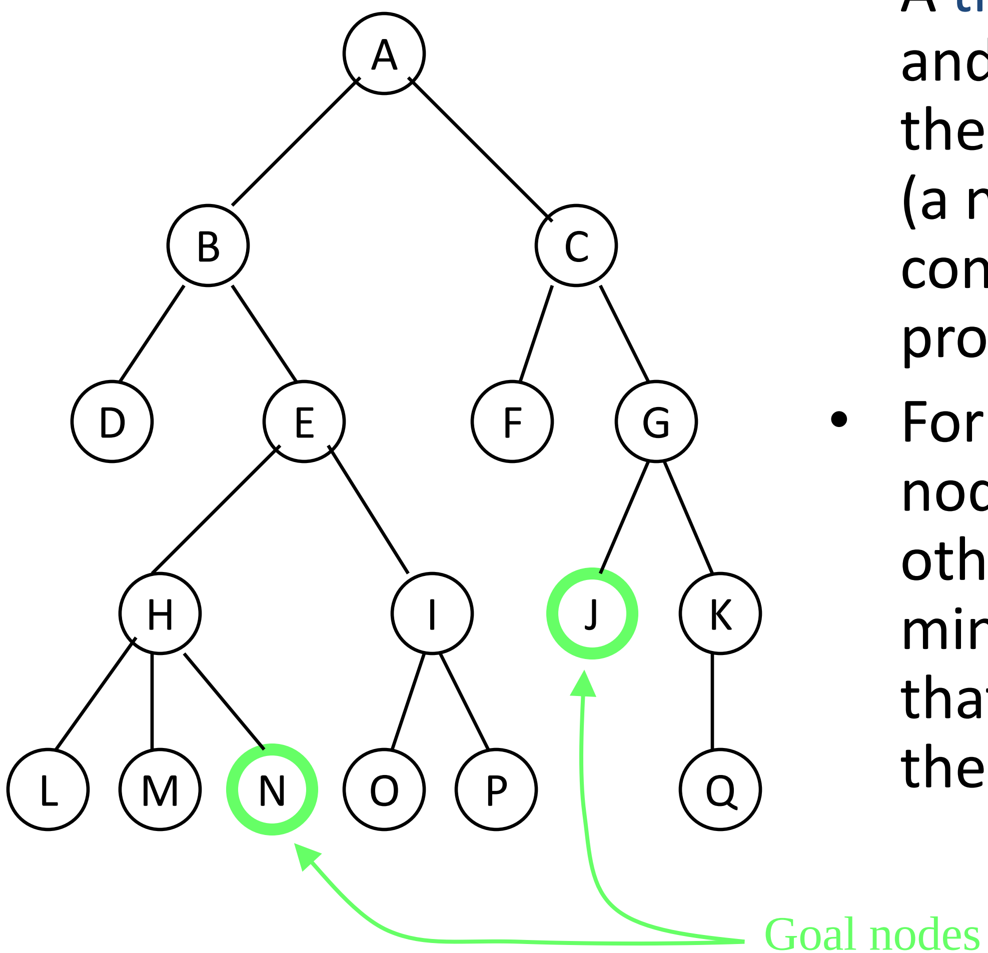
Some Operations

- degree of a vertex in an undirected graph
  - # of nodes in its adjacency list
- # of edges in a graph
  - determined in O(v+e)
- out-degree of a vertex in a directed graph
  - # of nodes in its adjacency list
- in-degree of a vertex in a directed graph
  - traverse the whole data structure

Graph Traversals

- We want to travel to every node in the graph.
- Traversals guarantee that we will get to each node exactly once.
- This can be used if we want to search for information held in the nodes or if we want to distribute information to each node.

Tree searches



- A tree search starts at the root and explores nodes from there, looking for a goal node (a node that satisfies certain conditions, depending on the problem)
- For some problems, any goal node is acceptable (N or `J`); for other problems, you want a minimum-depth goal node, that is, a goal node nearest the root (only `J`)

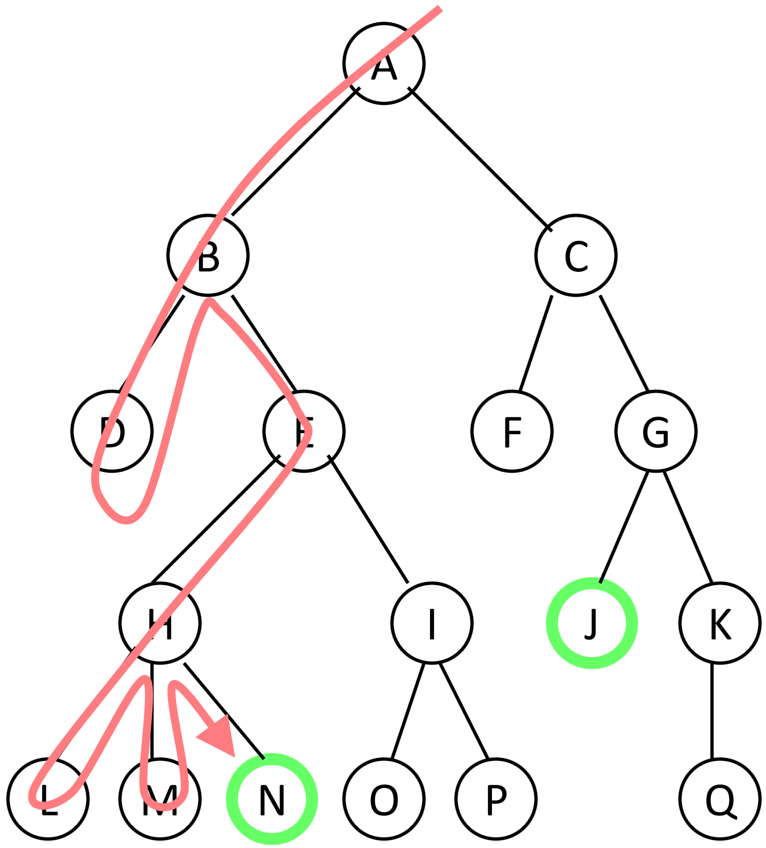
Graph Traversal

- Problem: Search for a certain node or traverse all nodes in the graph
- Depth First Search (DFS)
  - Once a possible path is found, continue the search until the end of the path
- Breadth First Search (BFS)
  - Start several paths at a time, and advance in each one step at a time

Depth-First Traversal

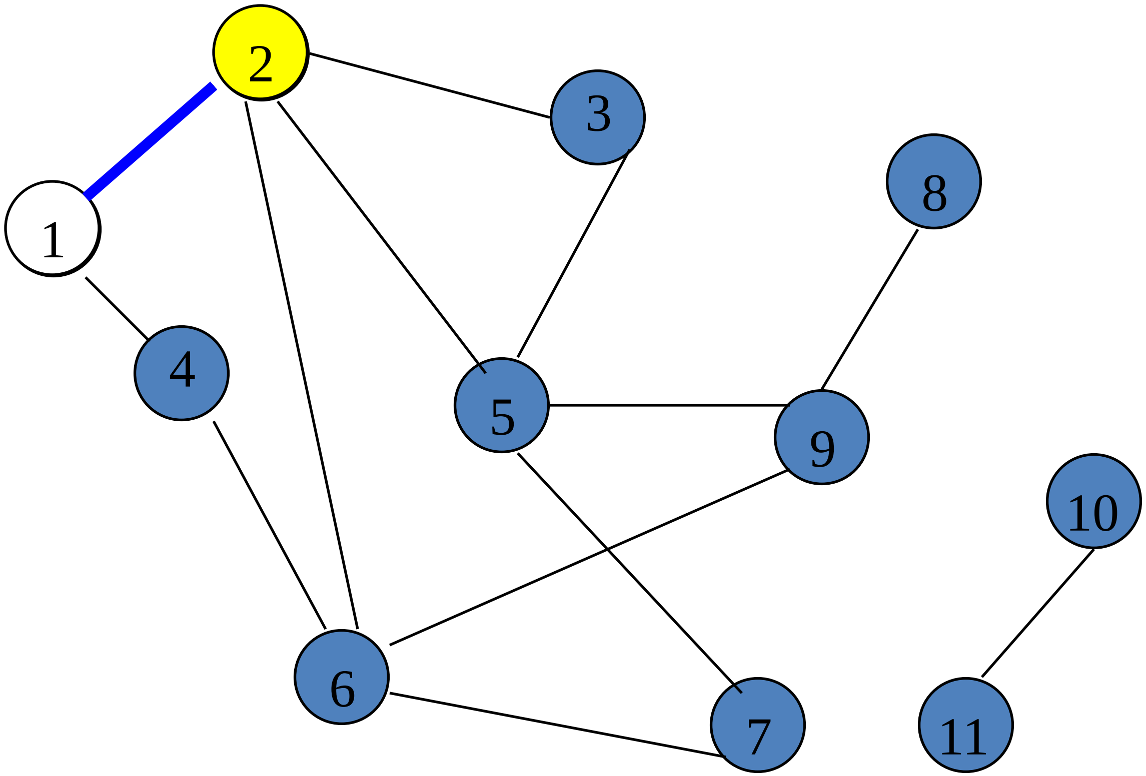
- We follow a path through the graph until we reach a dead end.
- We then back up until we reach a node with an edge to an unvisited node
- We take this edge and again follow it until we reach a dead end
- This process continues until we back up to the starting node and it has no edges to unvisited nodes

Depth-first searching in a Tree



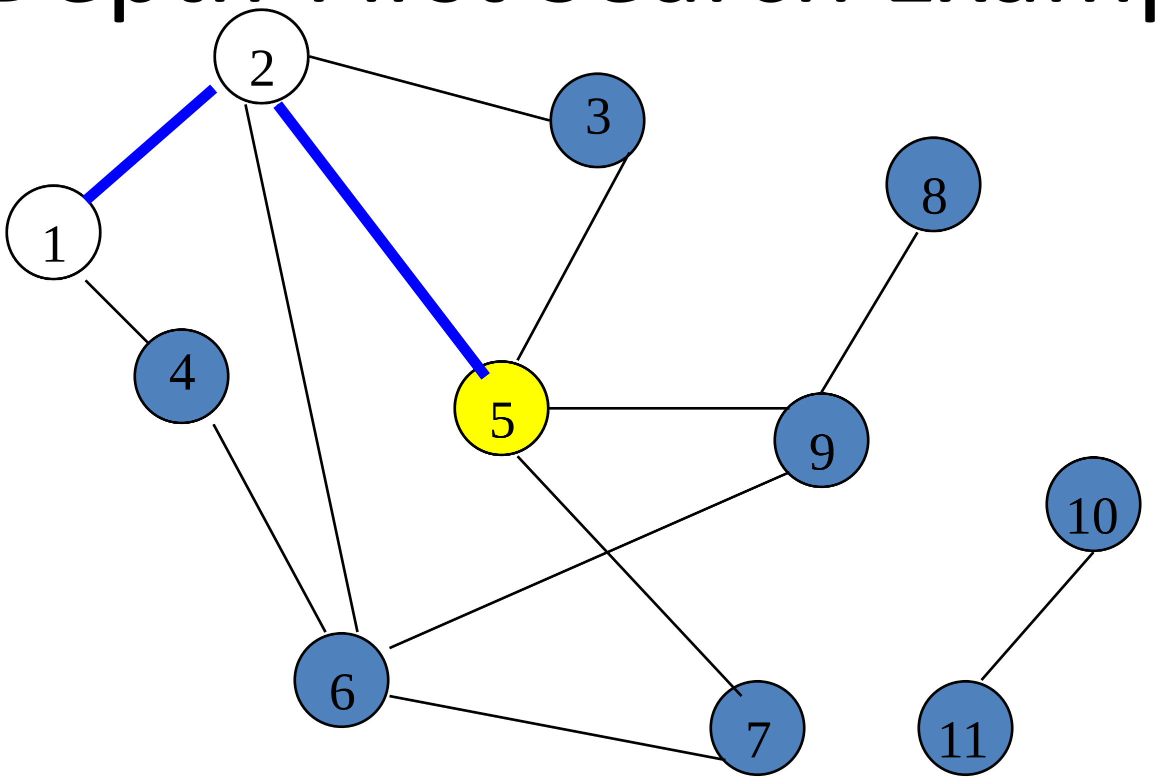
- A depth-first search (DFS) explores a path all the way to a leaf before backtracking and exploring another path
- For example, after searching A, then B, then D, the search backtracks and tries another path from B
- Node are explored in the order A B D E H L M N I O P C F G J K Q
- N will be found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

Depth-First Search Example



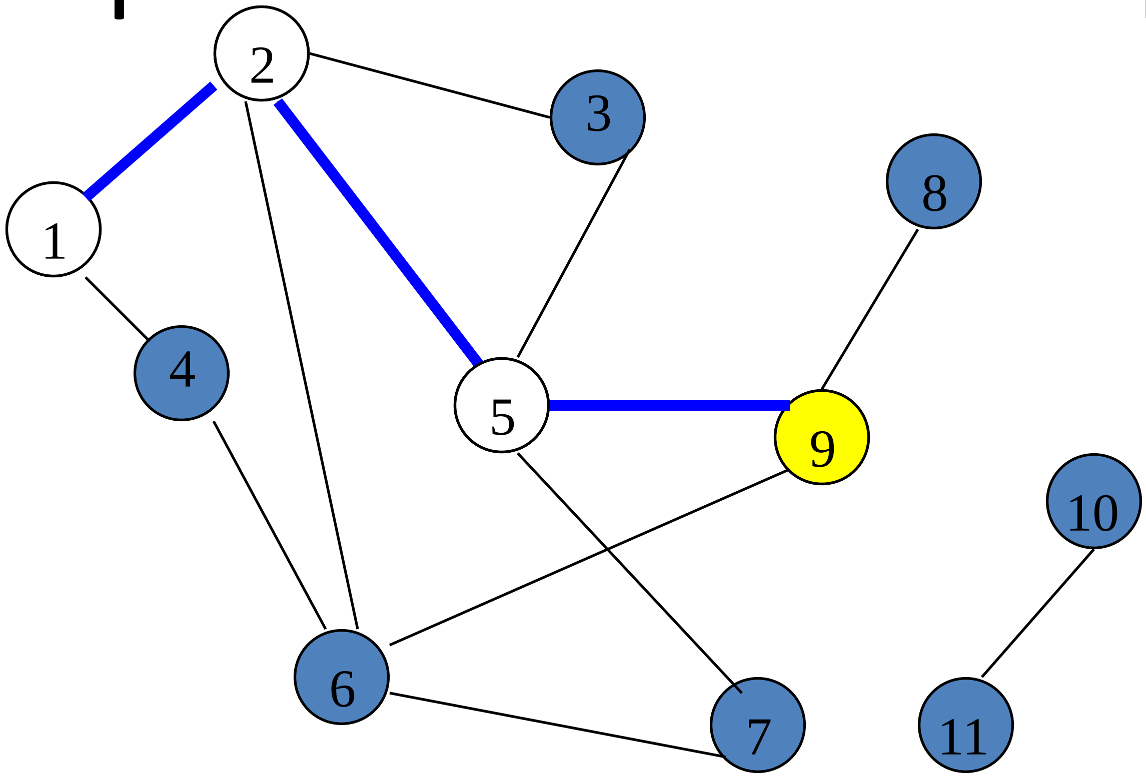
- Start search at vertex 1
- Label vertex 1 and do a depth first search from either 2 or 4
- Suppose that vertex 2 is selected

Depth-First Search Example



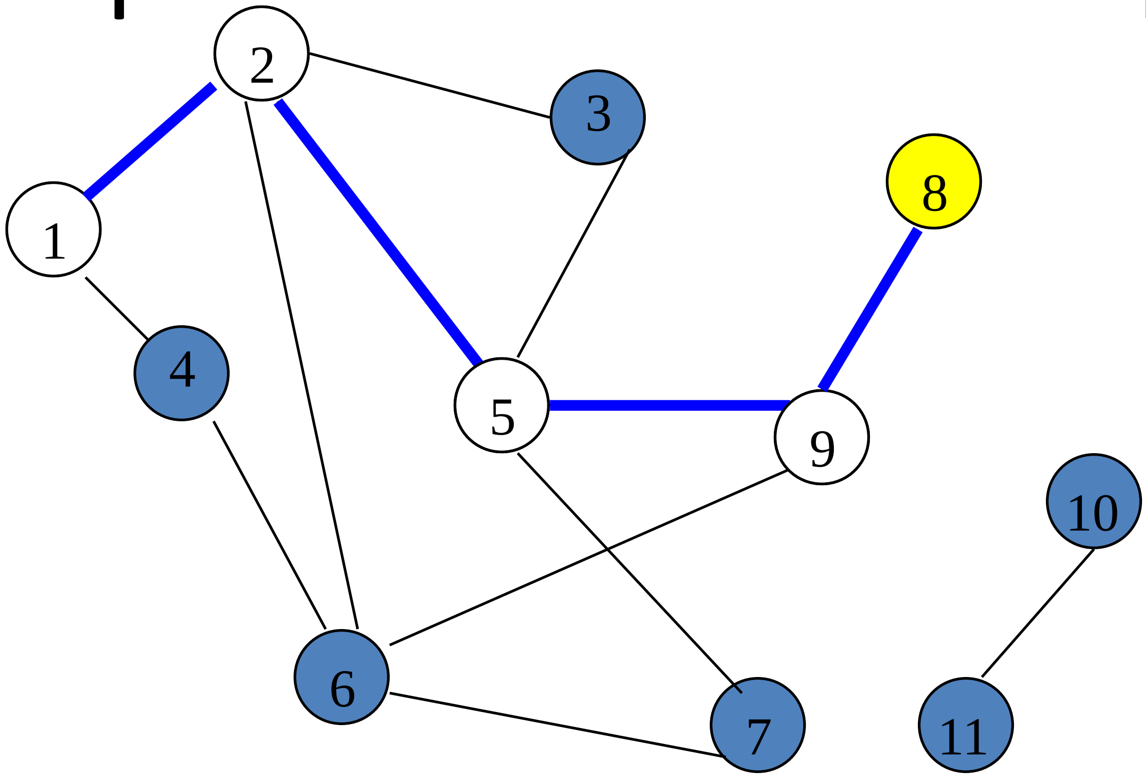
- Label vertex 2 and do a depth first search from either 3, 5, or 6
- Suppose that vertex 5 is selected

Depth-First Search Example



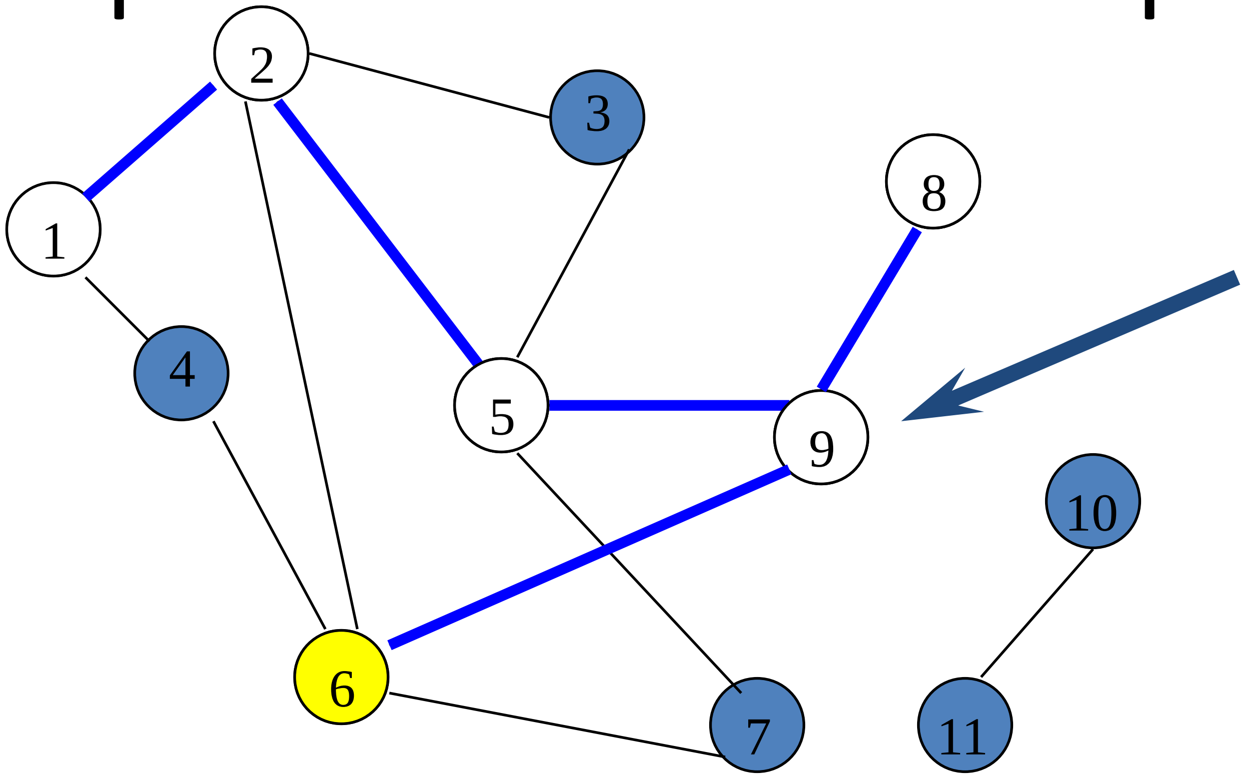
- Label vertex 5 and do a depth first search from either 3, 7, or 9
- Suppose that vertex 9 is selected

Depth-First Search Example



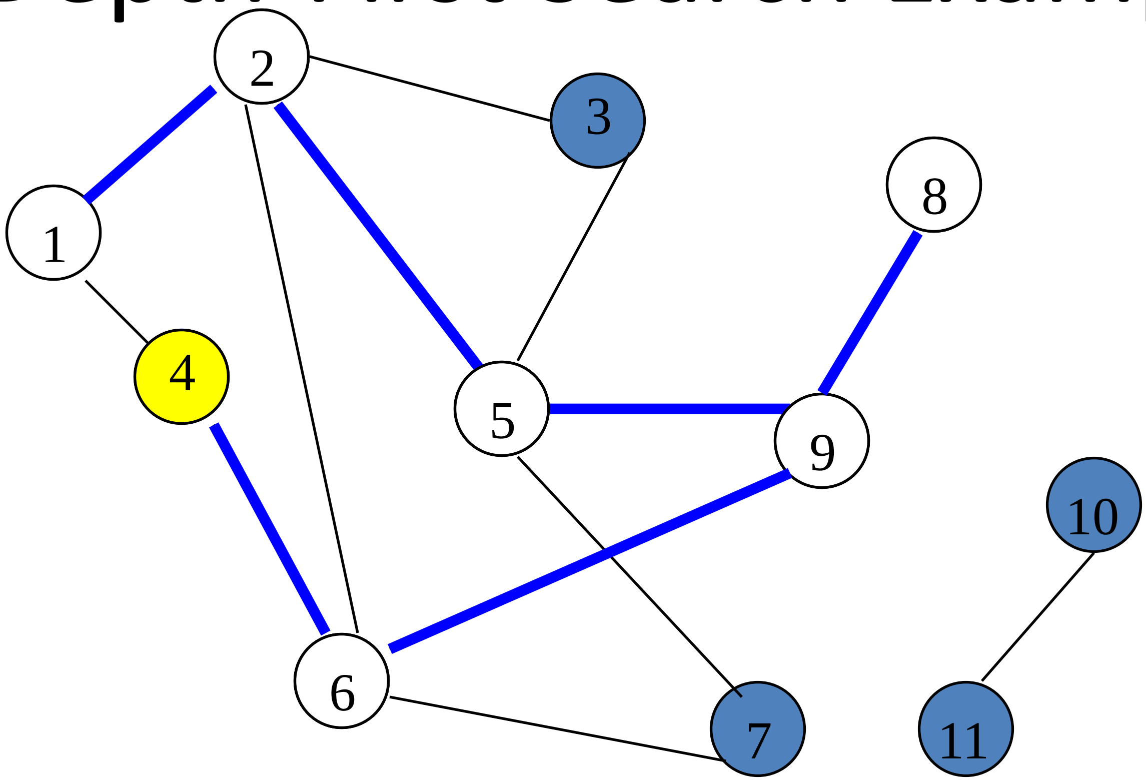
- Label vertex 9 and do a depth first search from either 6 or 8
- Suppose that vertex 8 is selected

Depth-First Search Example



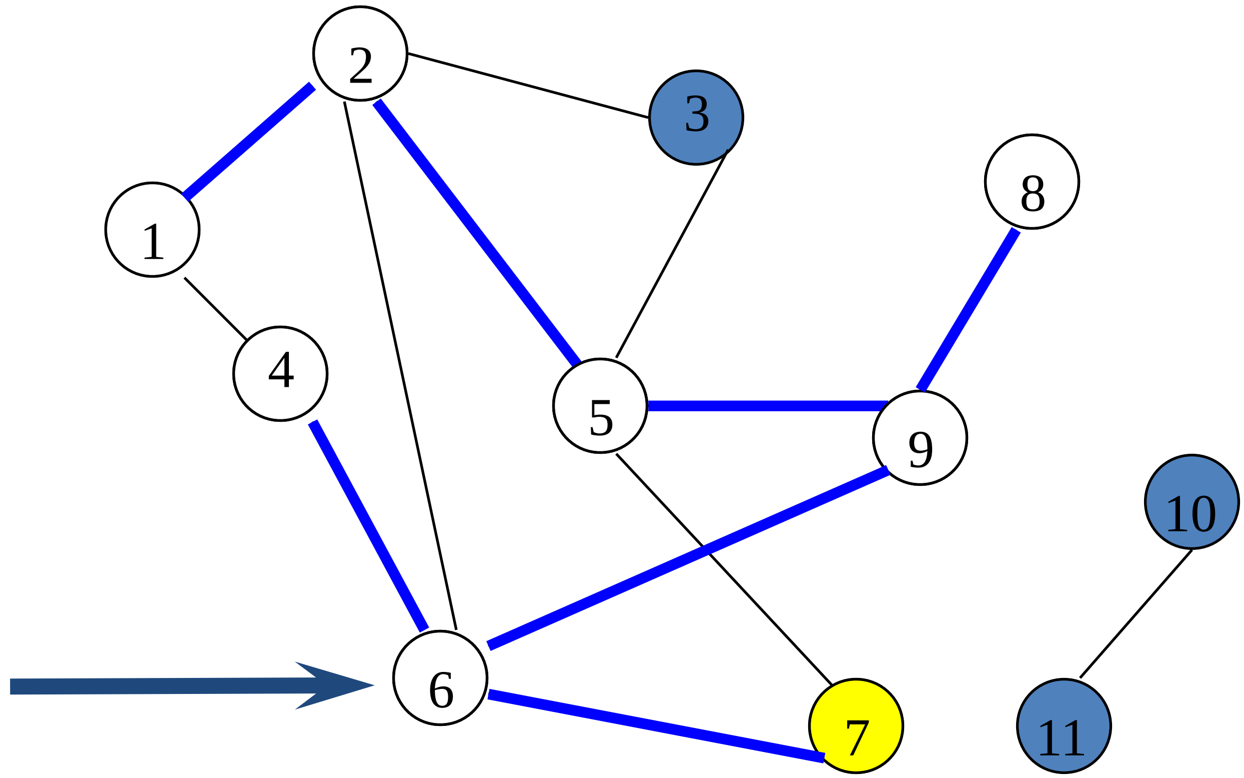
- **Label vertex 8 and return to vertex 9**
- From vertex 9 do a dfs(6)

Depth-First Search Example



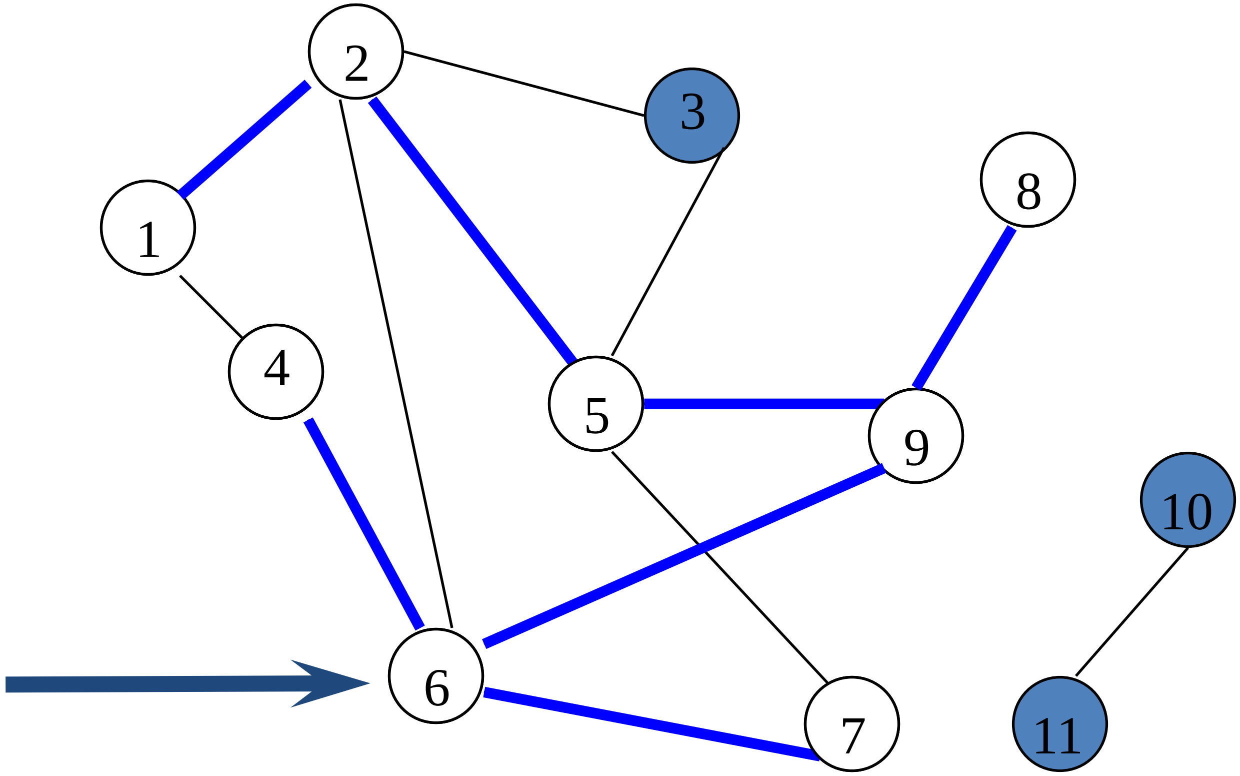
- Label vertex 6 and do a depth first search from either 4 or 7
- **Suppose that vertex 4 is selected**

Depth-First Search Example



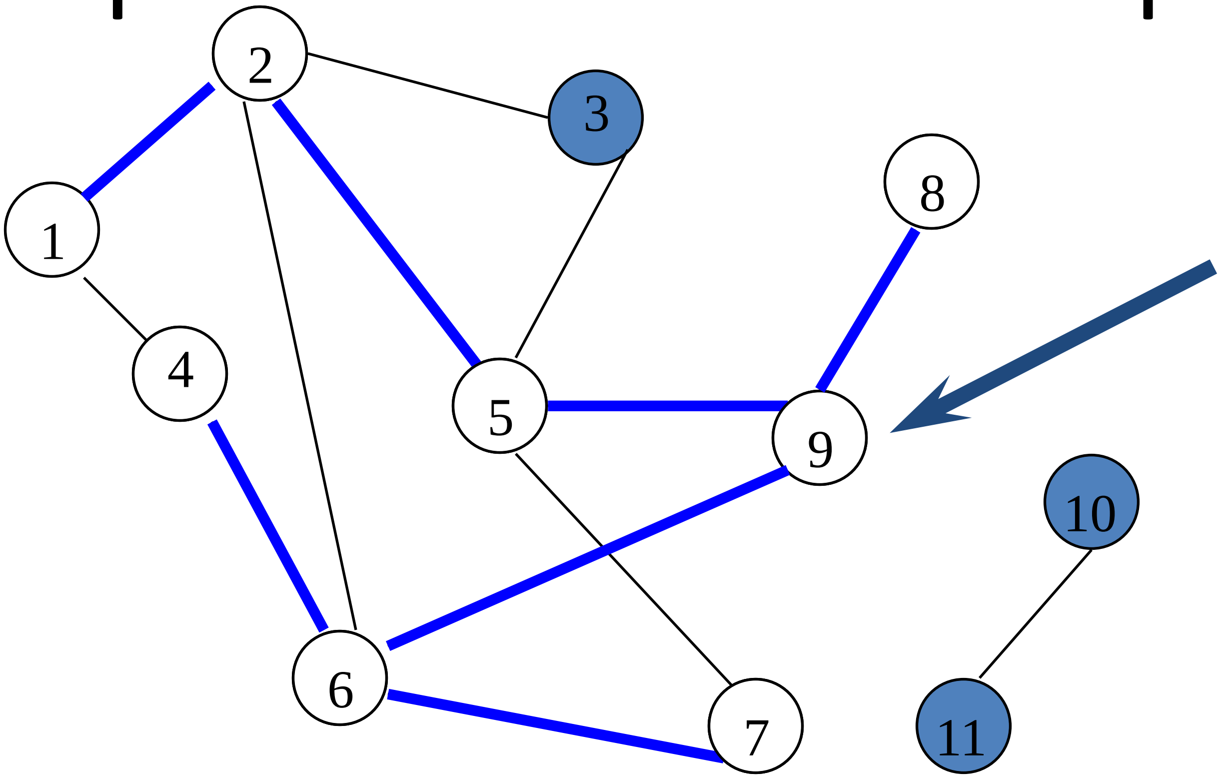
- Label vertex 4 and return to 6
- **From vertex 6 do a dfs(7)**

Depth-First Search Example



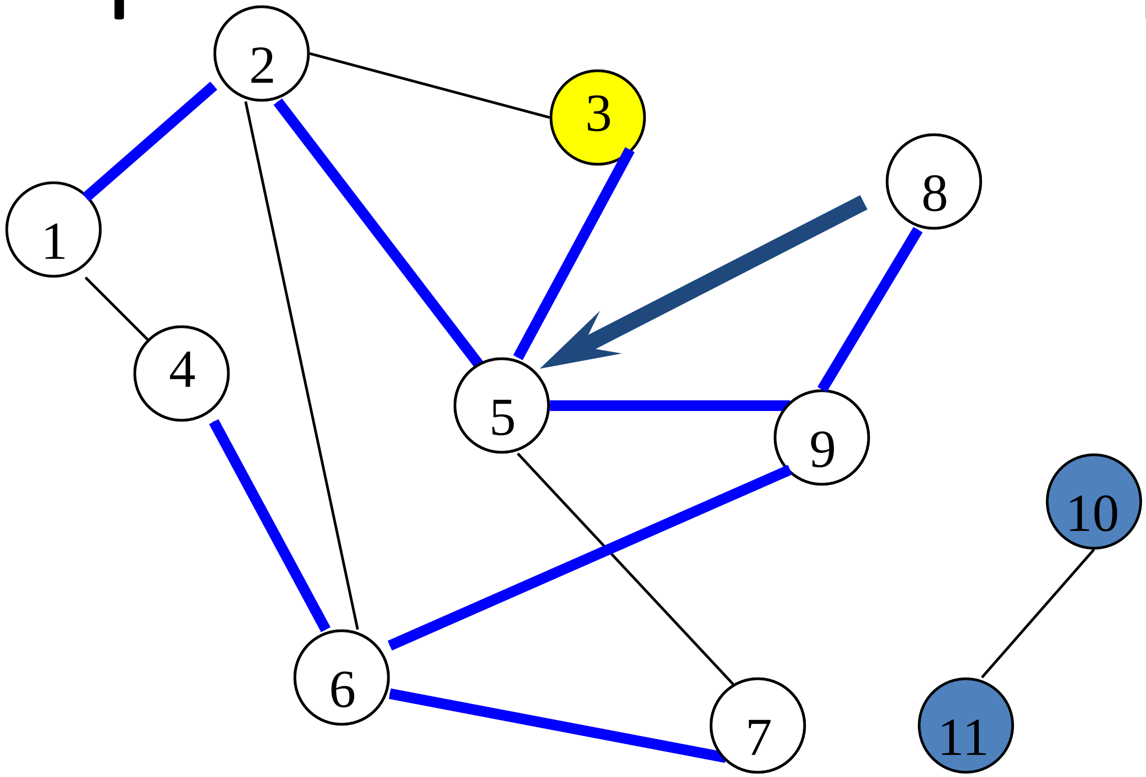
- Label vertex 7 and return to 6
- **Return to 9**

Depth-First Search Example



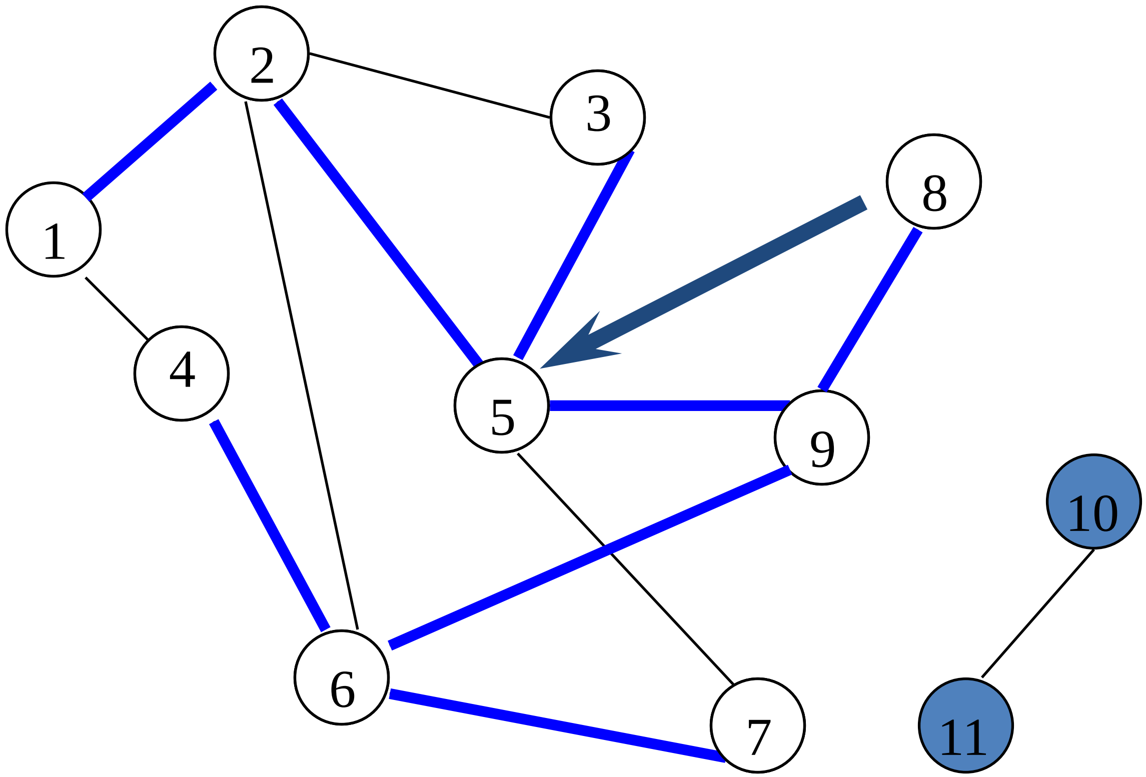
- Return to 5

Depth-First Search Example



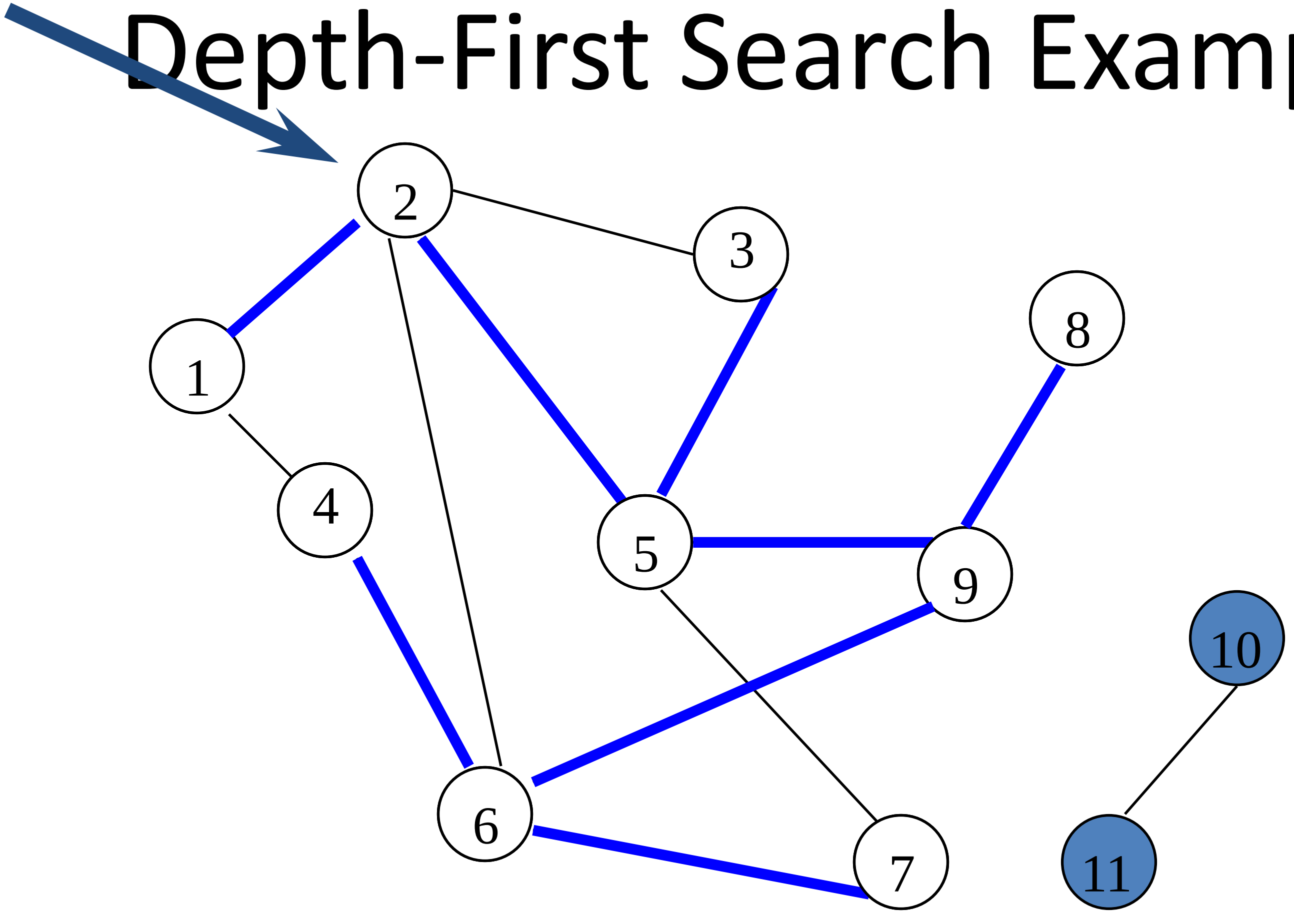
- Do a dfs(3)

Depth-First Search Example



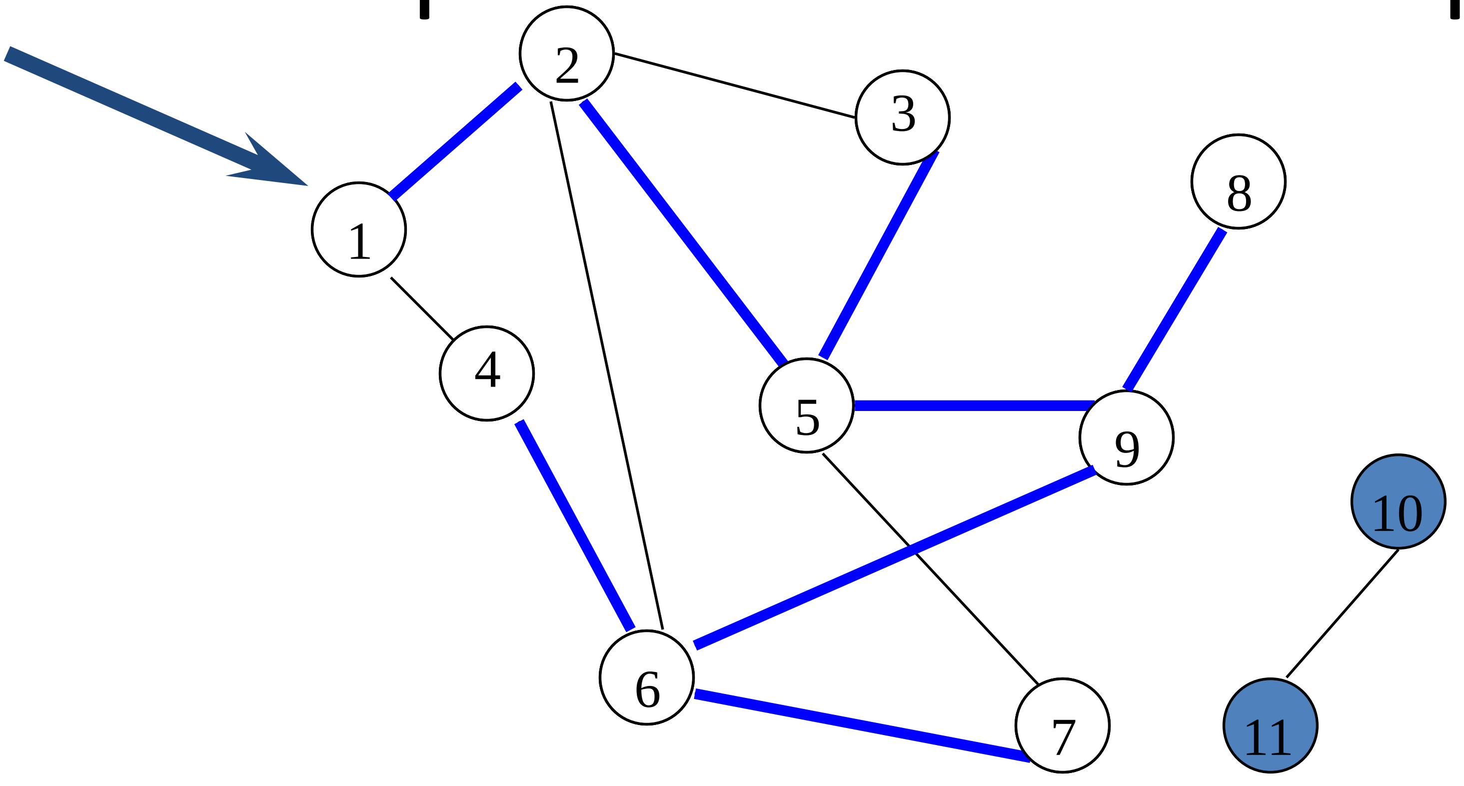
- Label 3 and return to 5
- Return to 2

Depth-First Search Example



- Return to 1

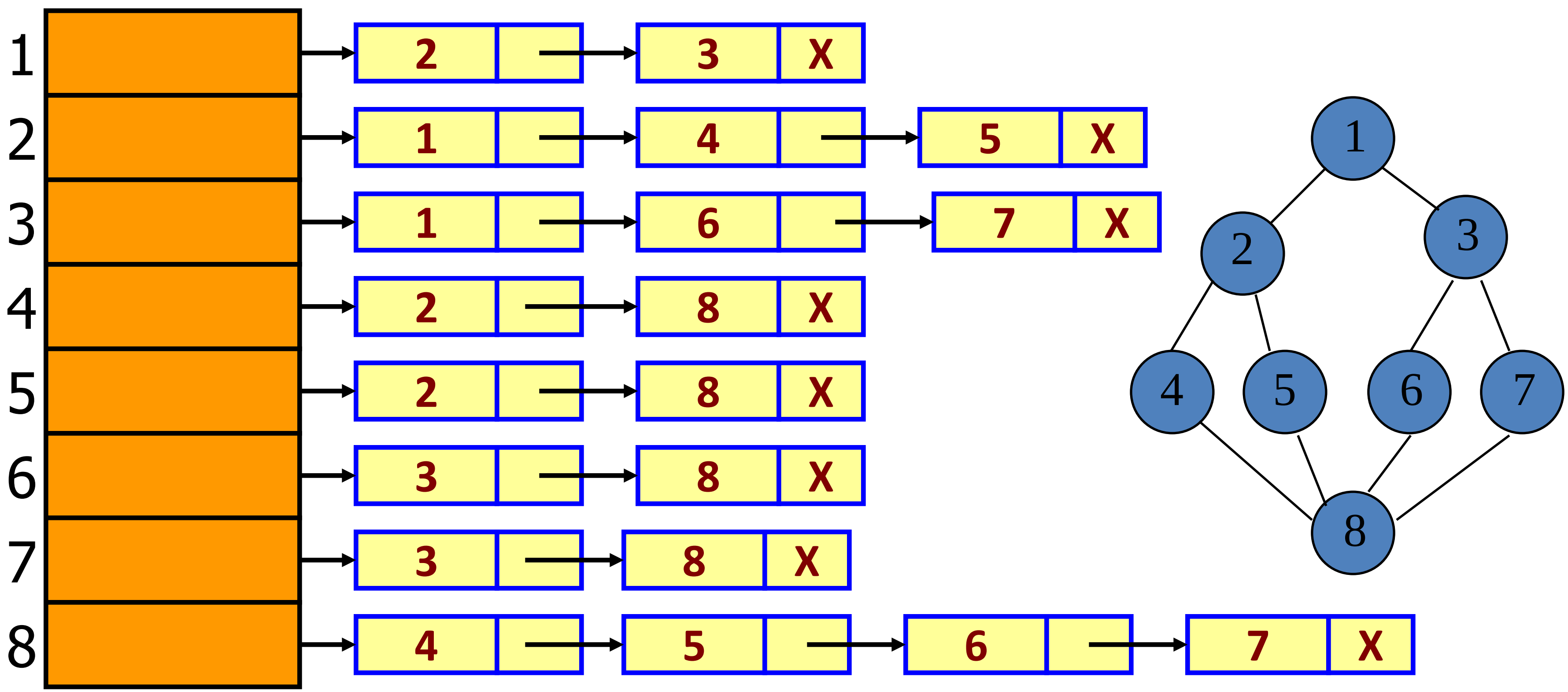
Depth-First Search Example



- Return to invoking method

Traversal: Another Example

- DFS (start vertex 1): 1, 2, 4, 8, 5, 6, 3, 7



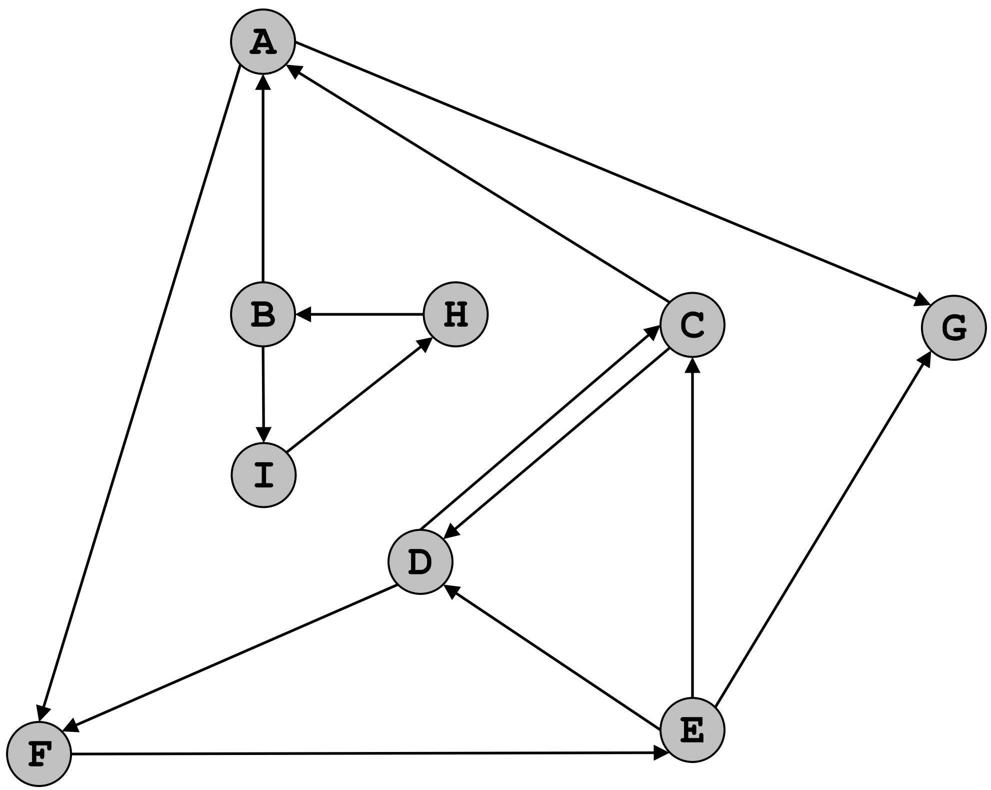
A Graph and its Adjacency List Representation

DFS (Pseudo Code)

```
DFS(input: Graph G) {
    Stack S; Integer x, t;
    while (G has an unvisited node x){
        visit(x); push(x,S);
        while (S is not empty){
            t := peek(S);
            if (t has an unvisited neighbor y){
                visit(y); push(y,S); }
            else
                pop(S);
        }
    }
}
```

```
//Recursive algorithm
DFS(v: vertex in G){
    Mark v as visited
    for (each unvisited vertex u adjacent to v)
        DFS(u)
}
```

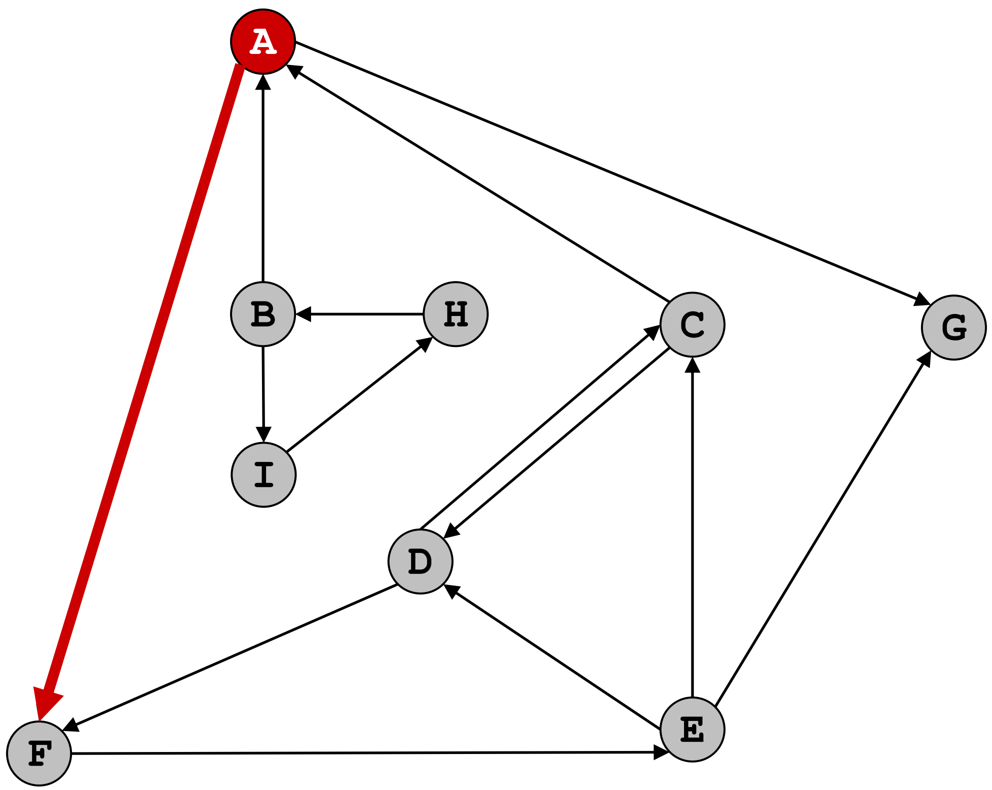
Directed Depth First Search

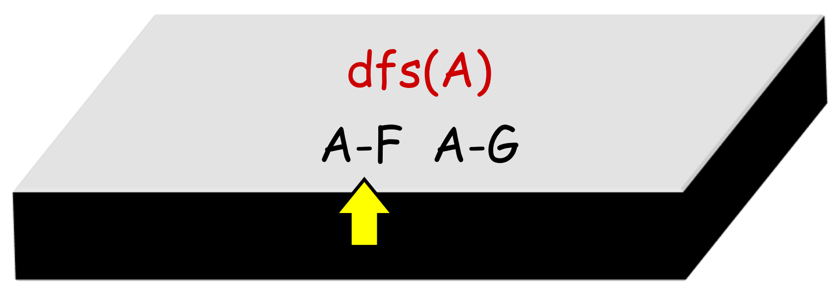


Adjacency Lists

A: F G
B: A I
C: A D
D: C F
E: C D G
F: E
G:
H: B
I: H

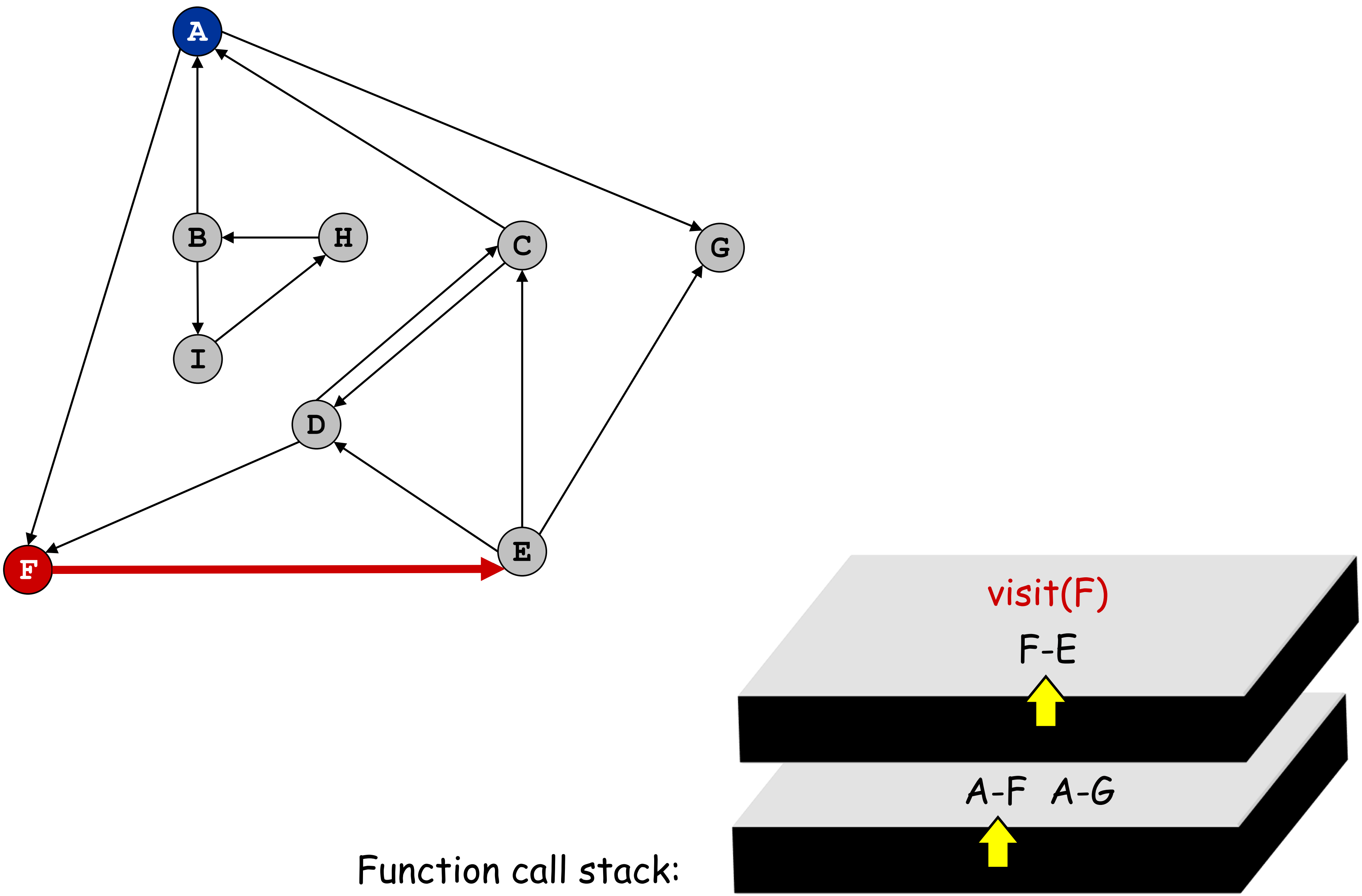
Directed Depth First Search



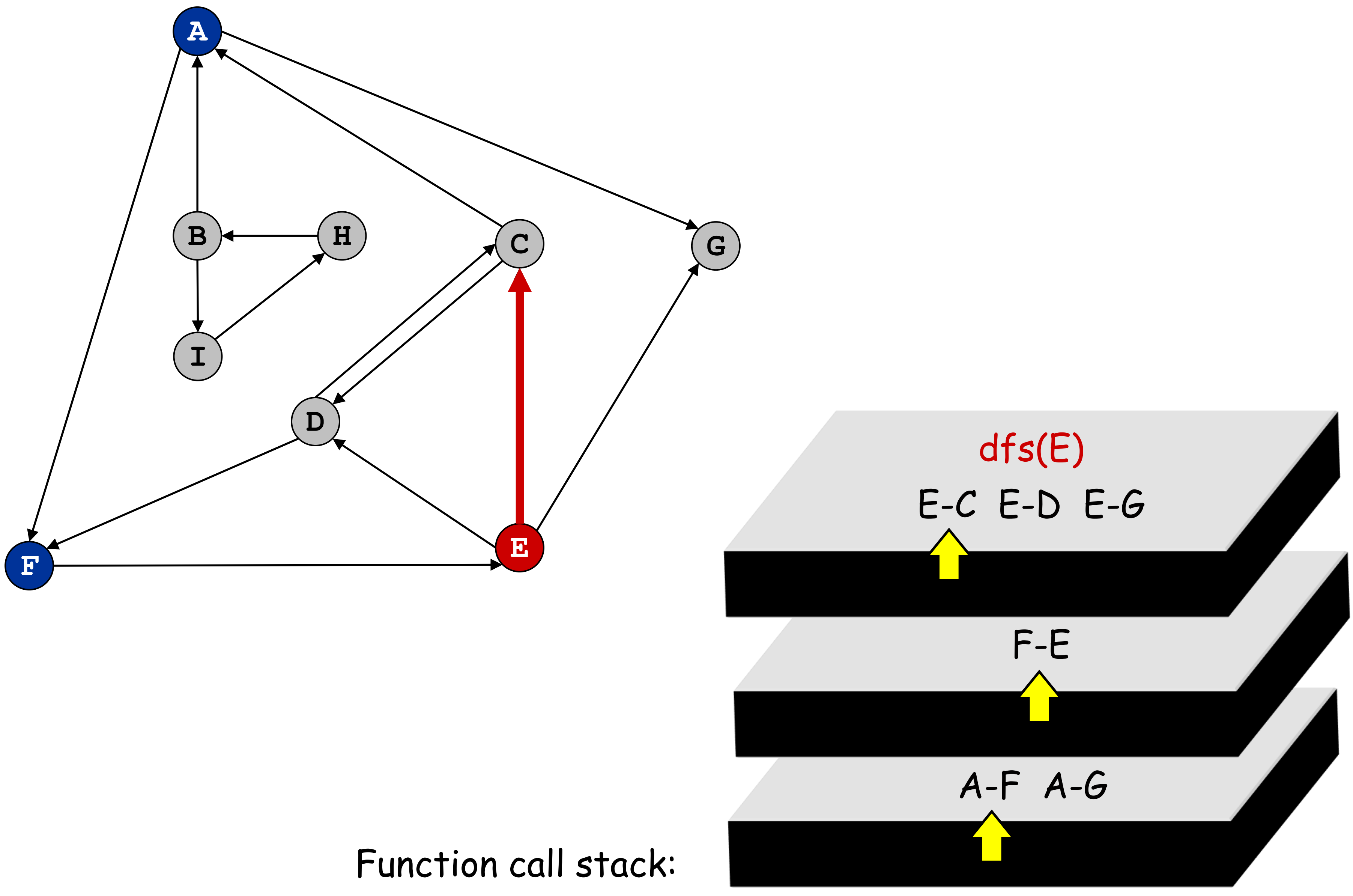


Function call stack:

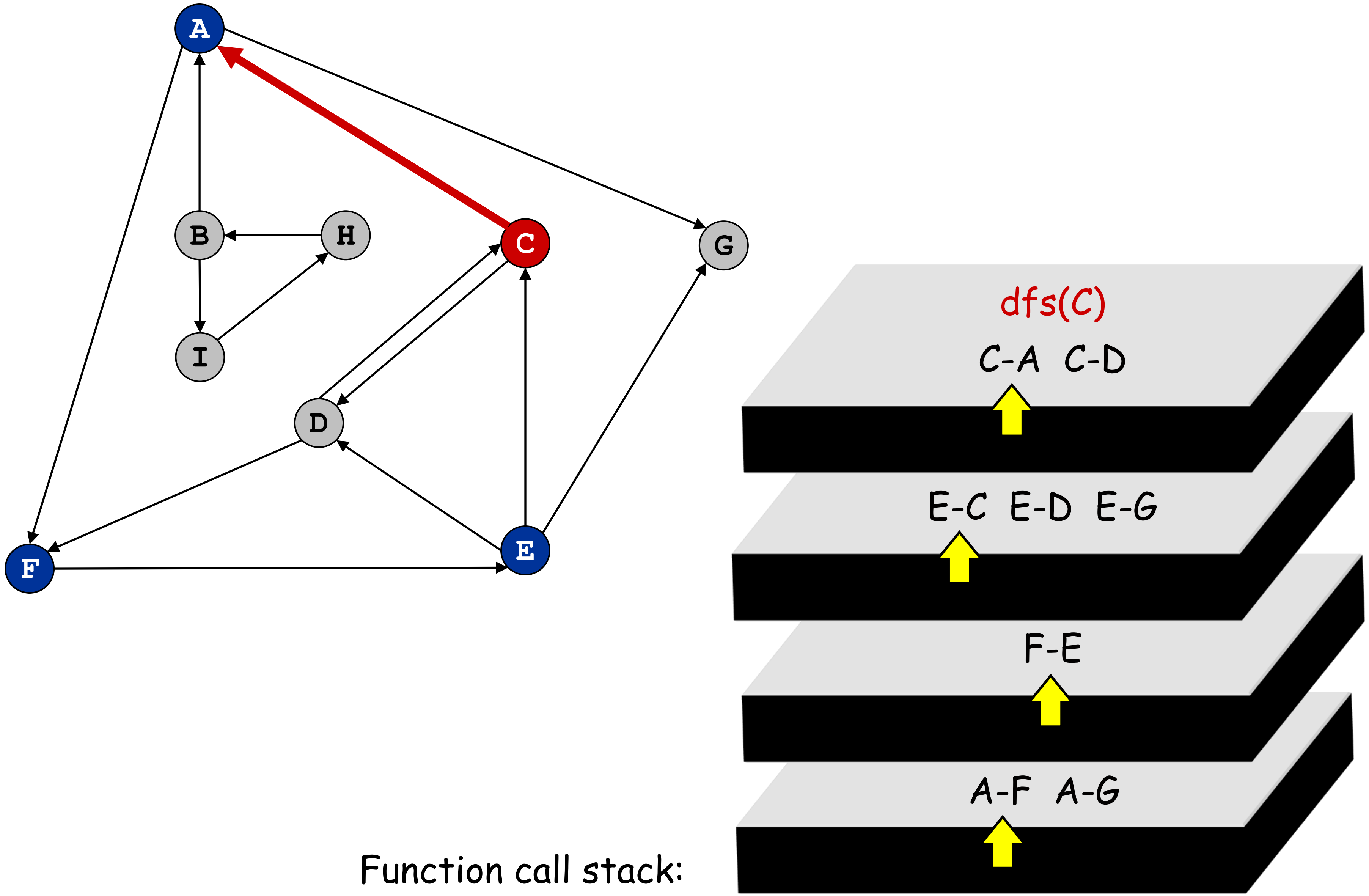
Directed Depth First Search



Directed Depth First Search

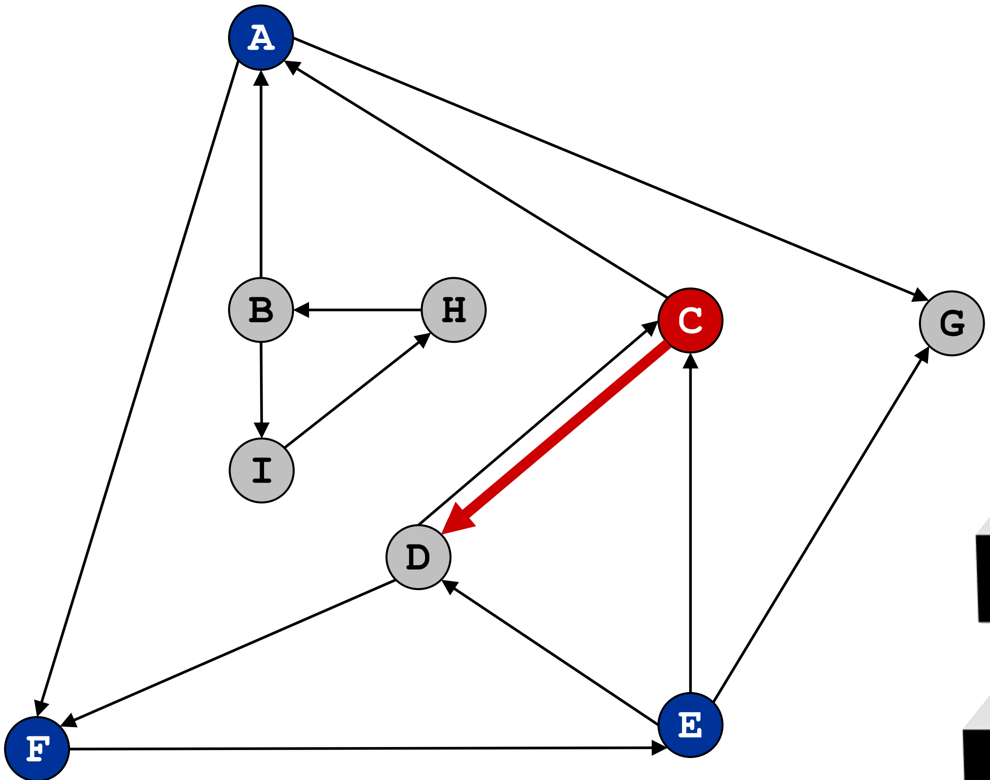


Directed Depth First Search

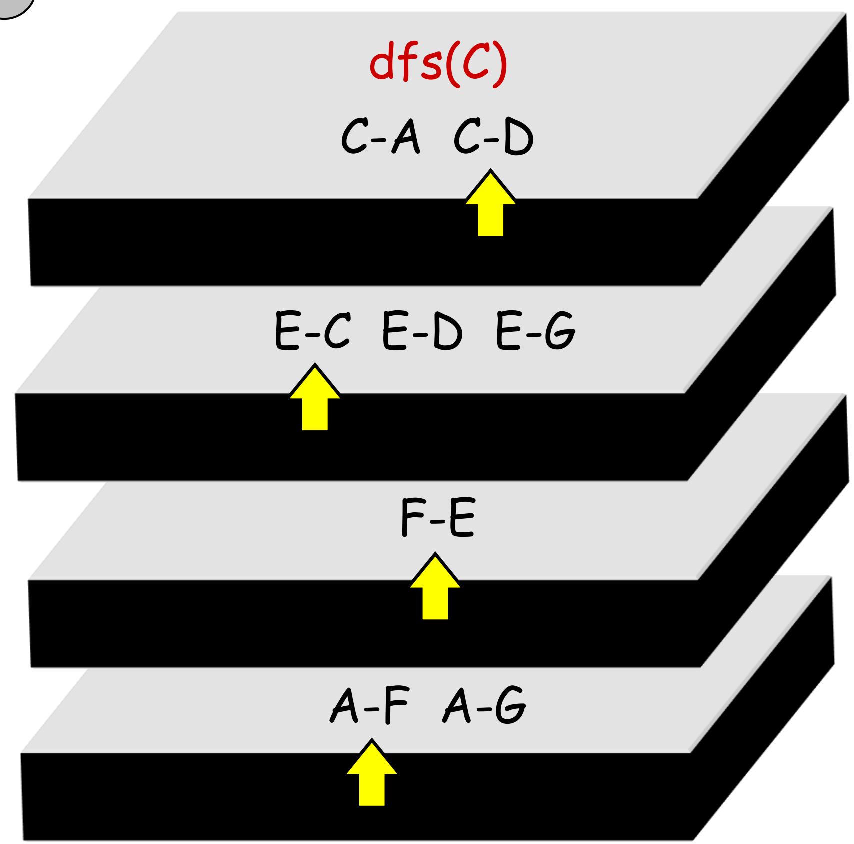


Function call stack:

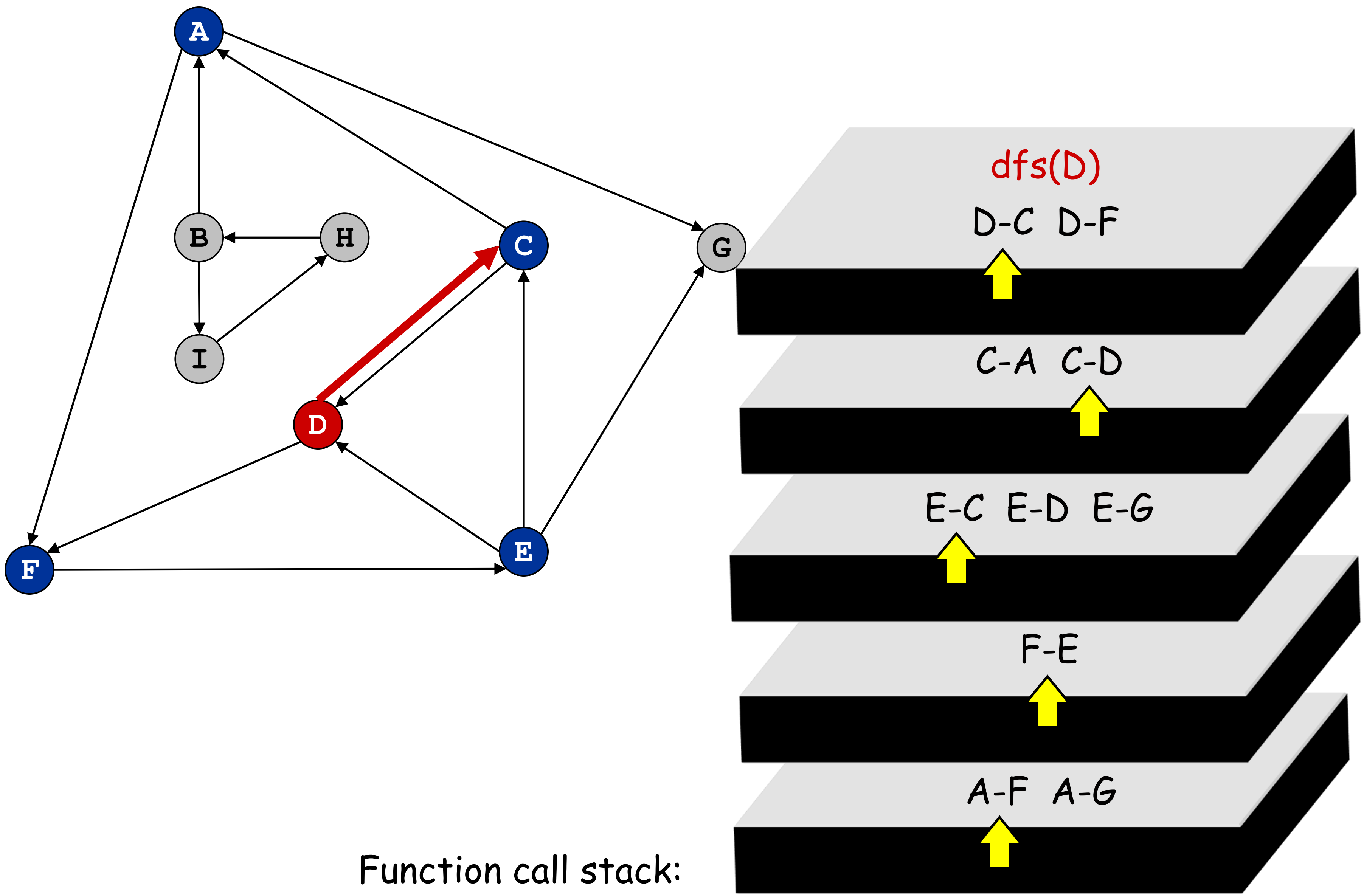
Directed Depth First Search



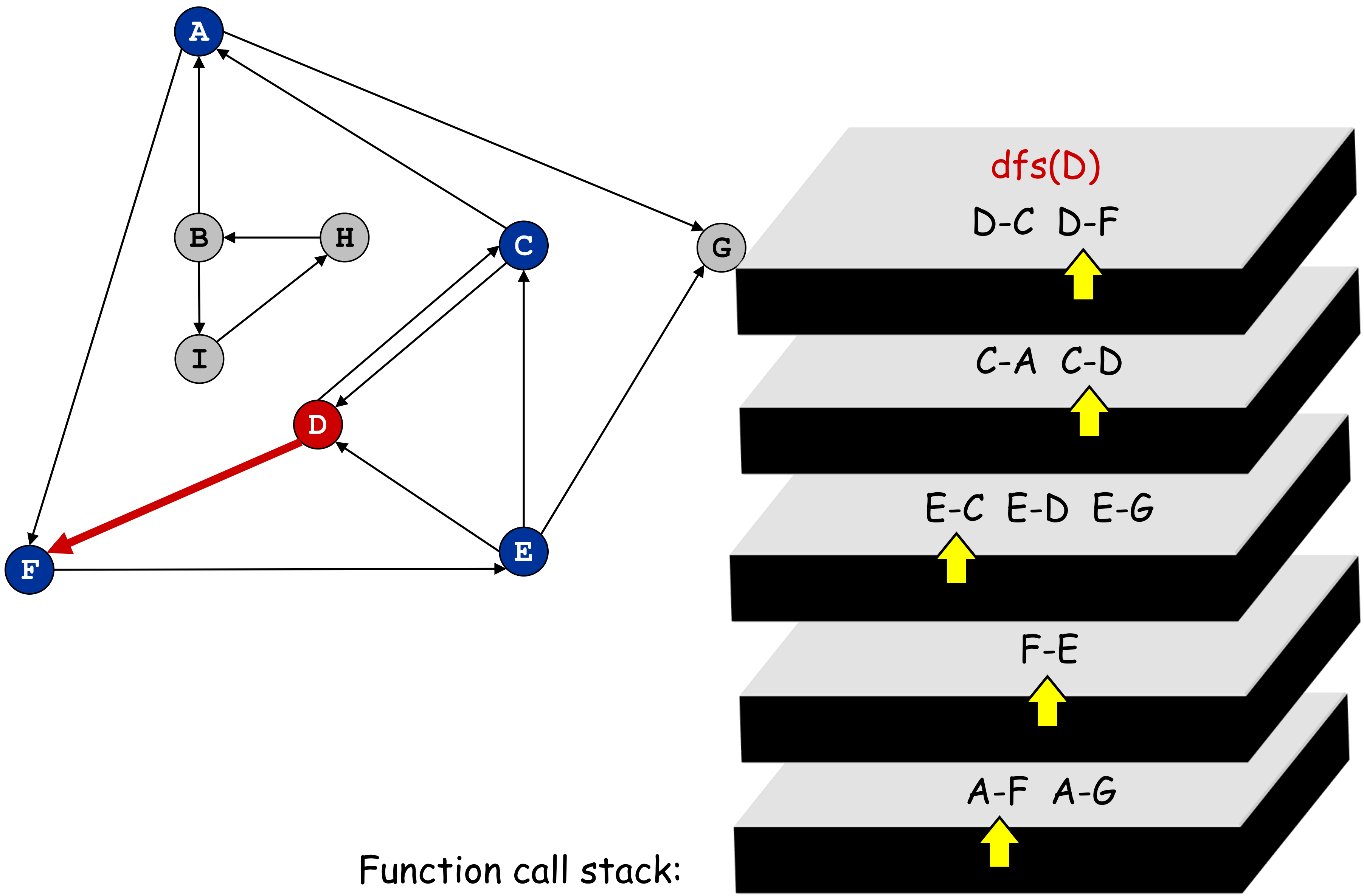
Function call stack:



Directed Depth First Search

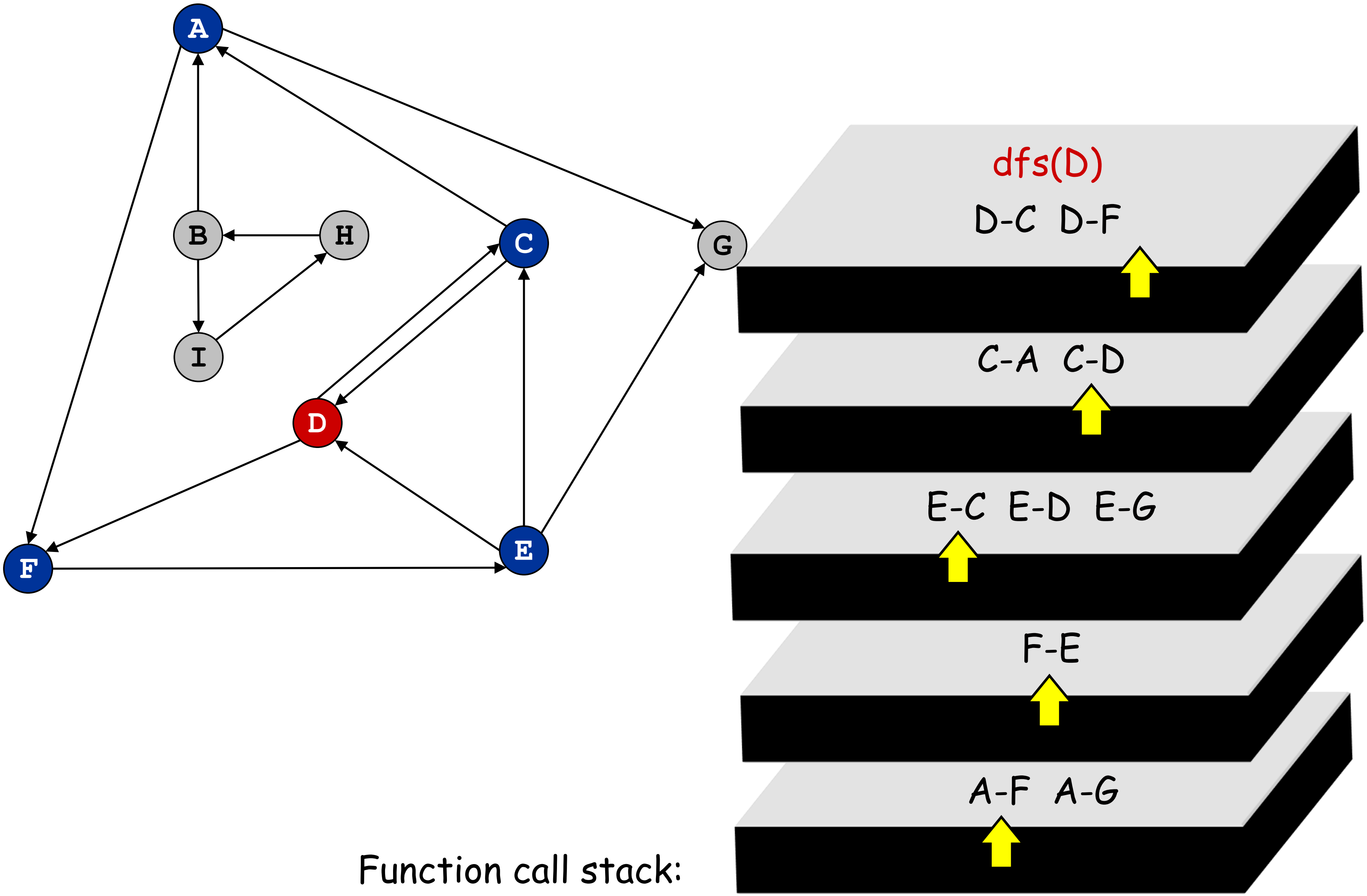


Directed Depth First Search

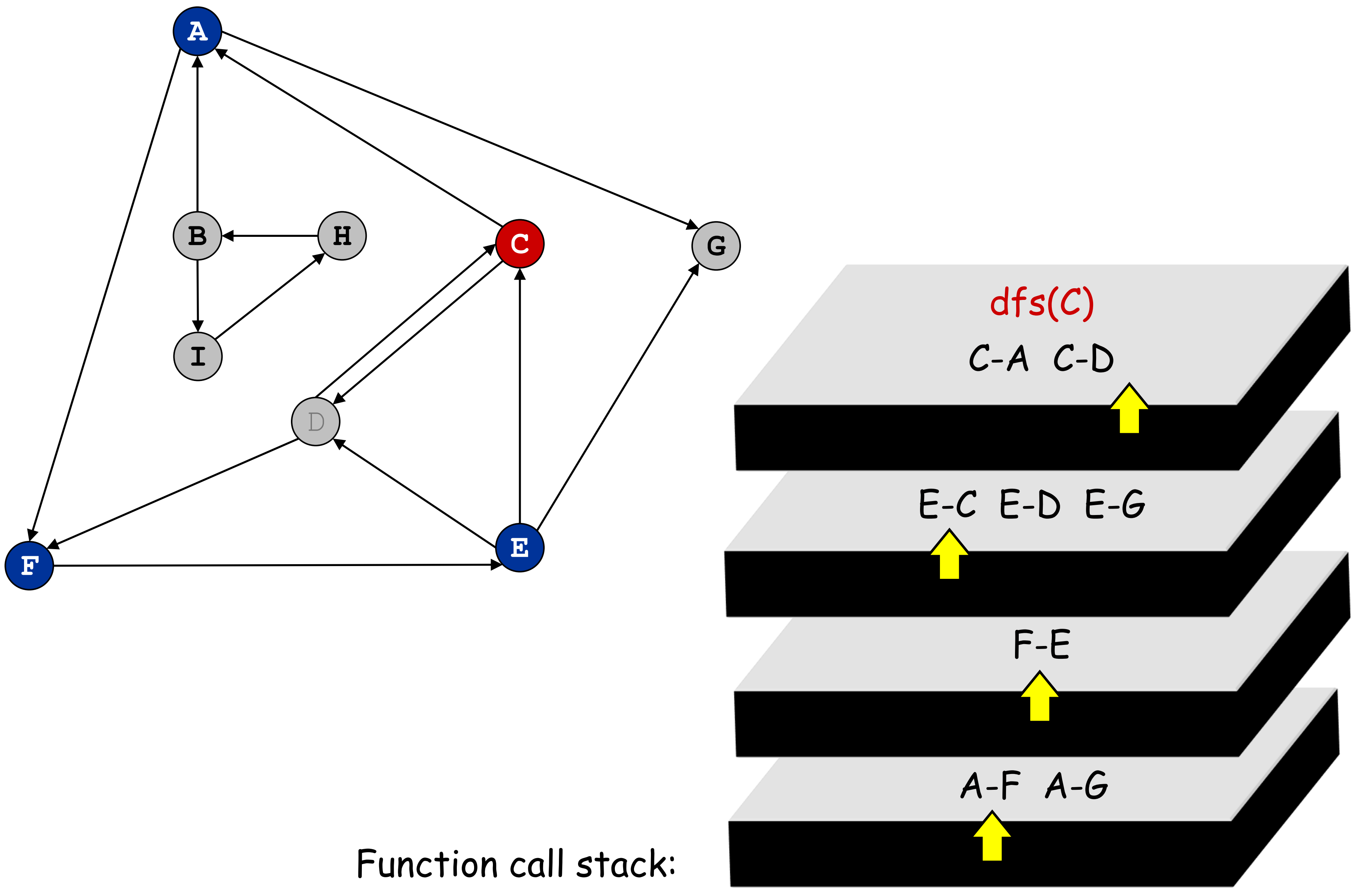


Function call stack:

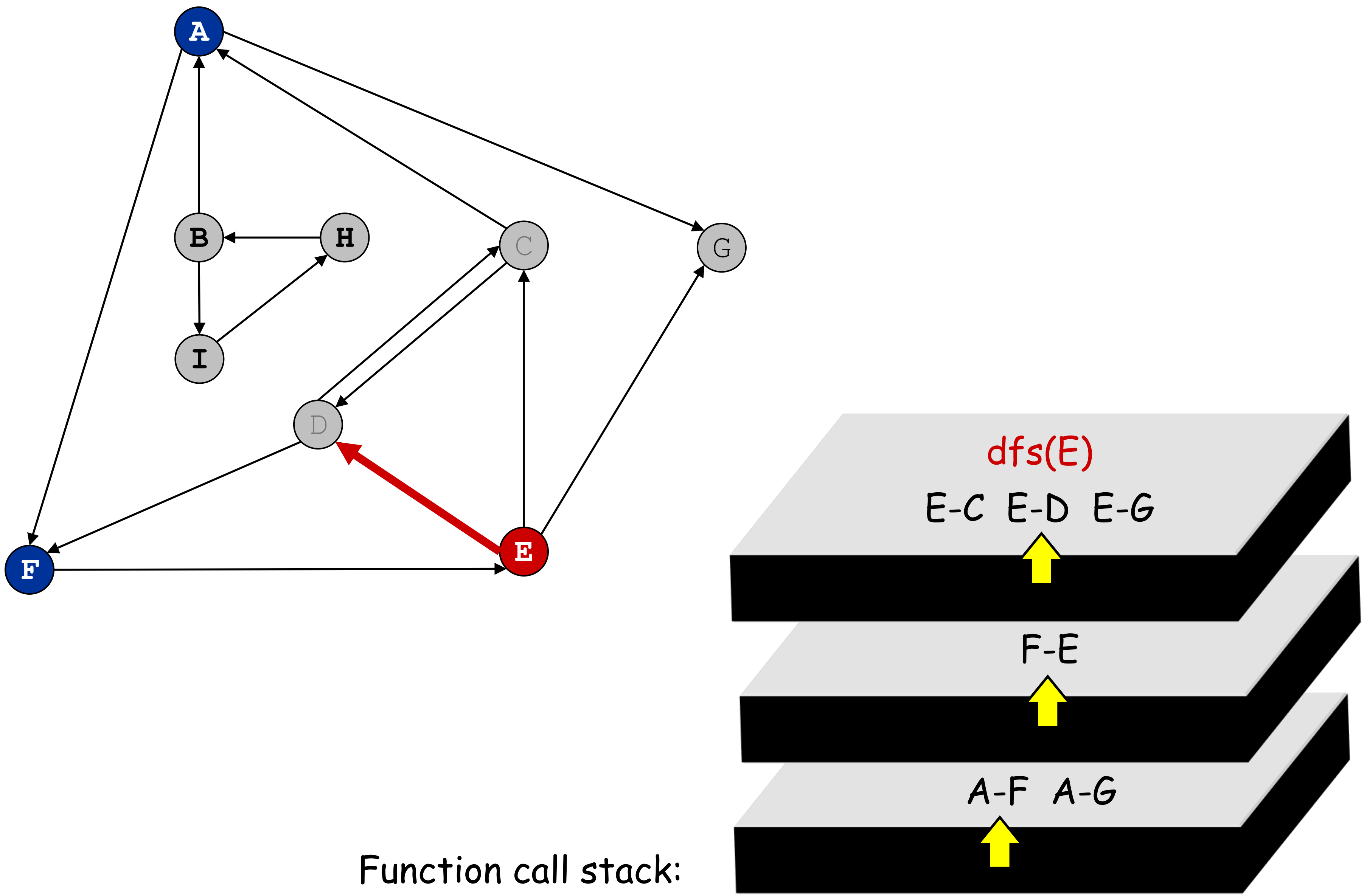
Directed Depth First Search



Directed Depth First Search

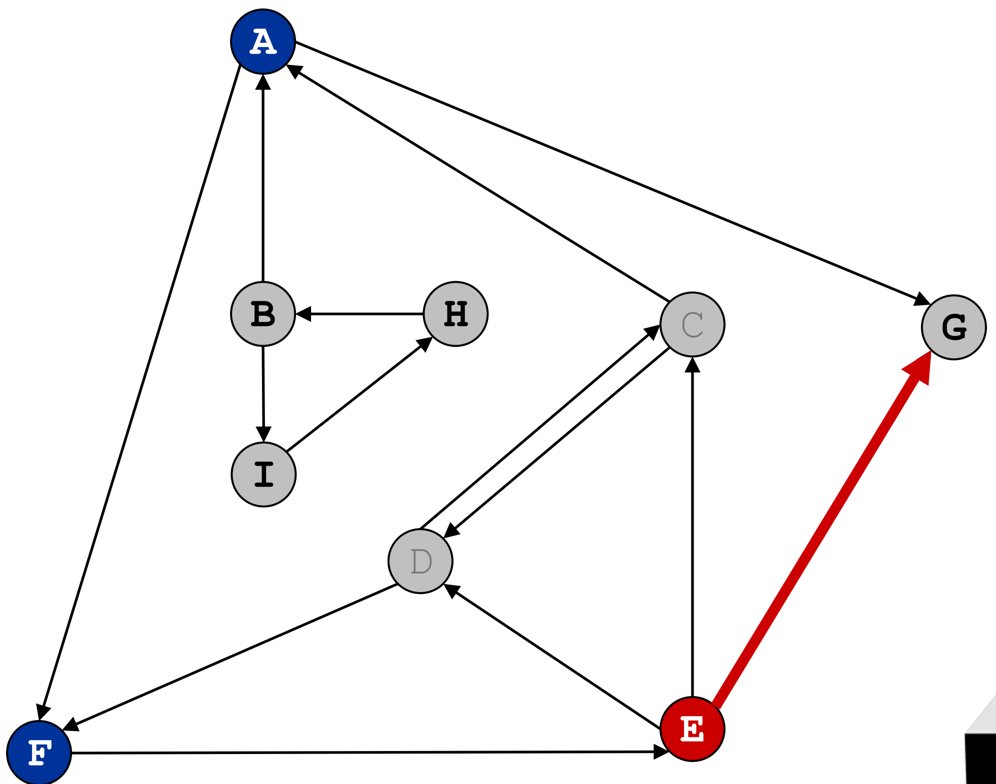


Directed Depth First Search

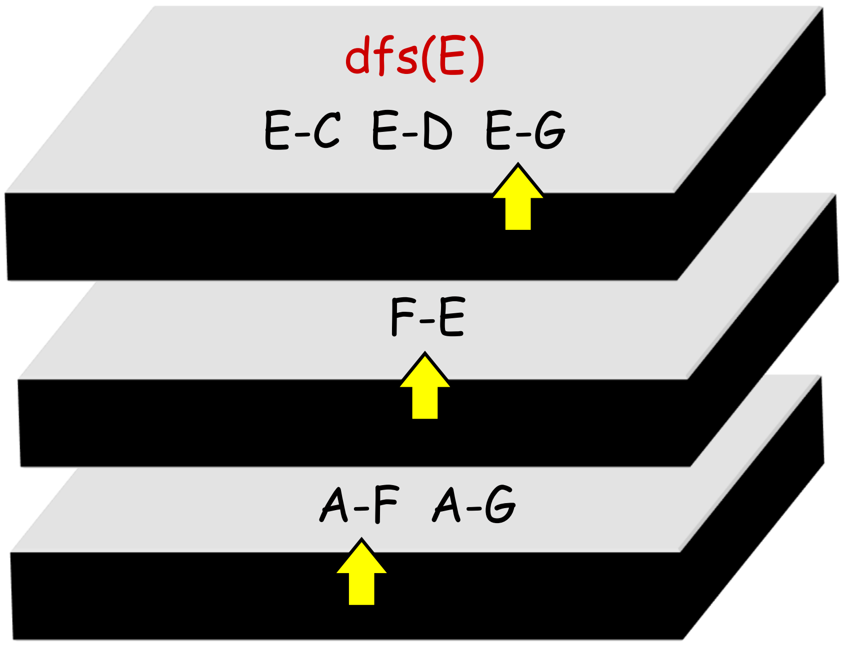


Function call stack:

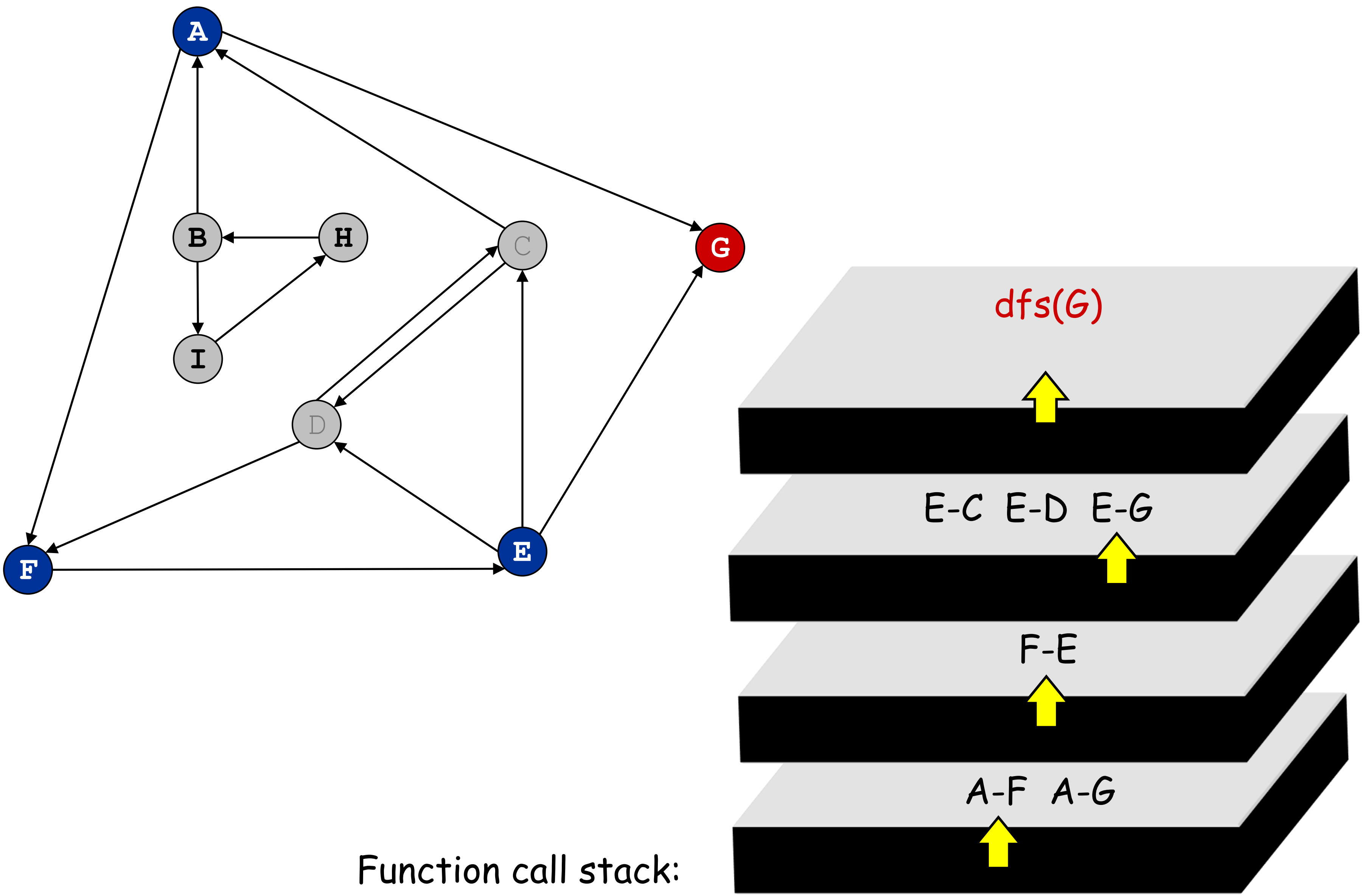
Directed Depth First Search



Function call stack:

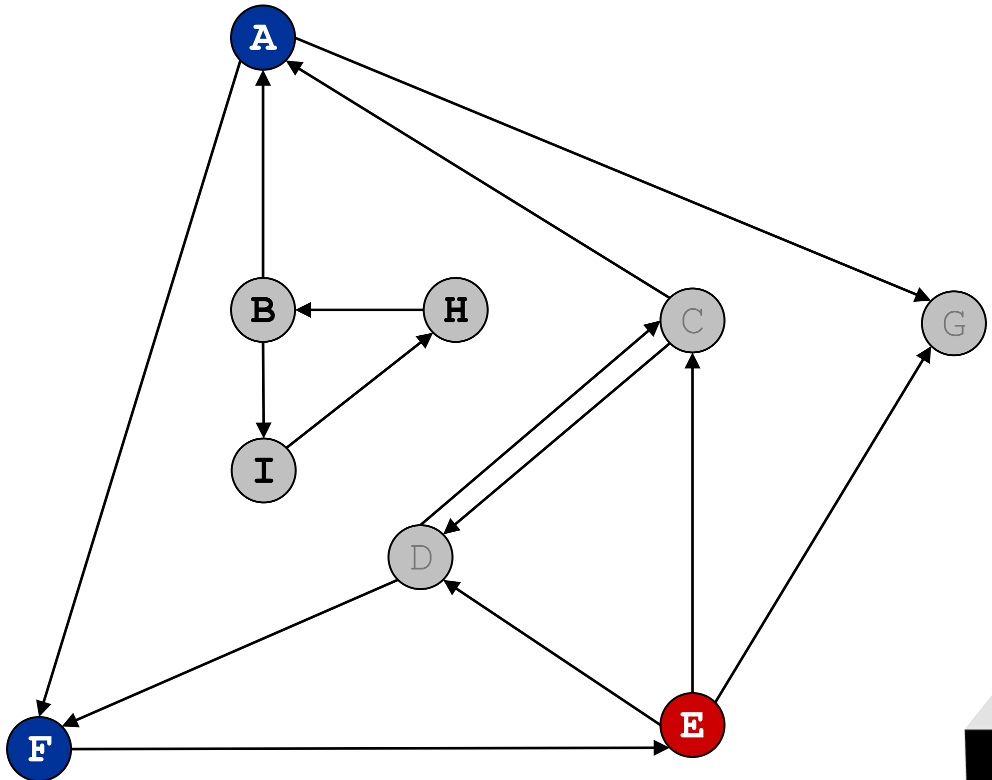


Directed Depth First Search

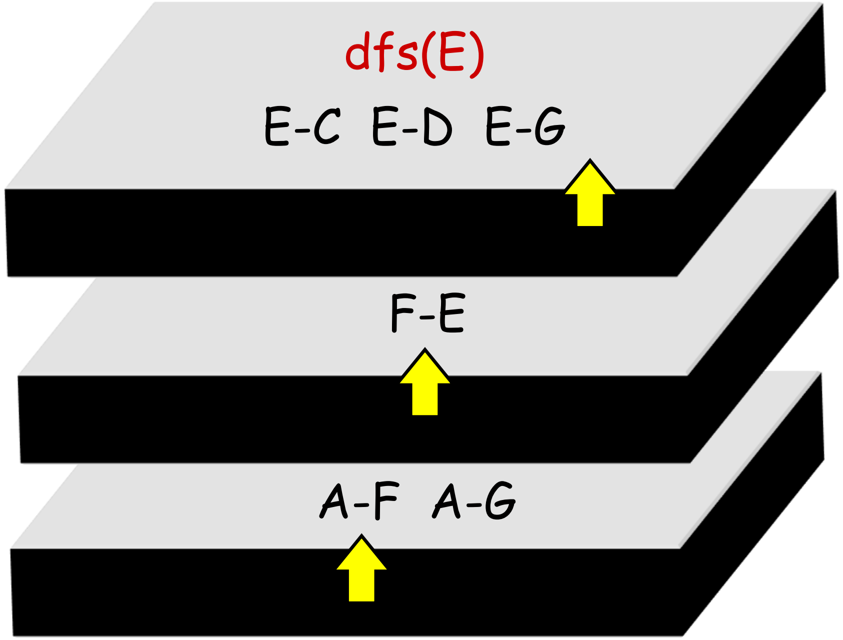


Function call stack:

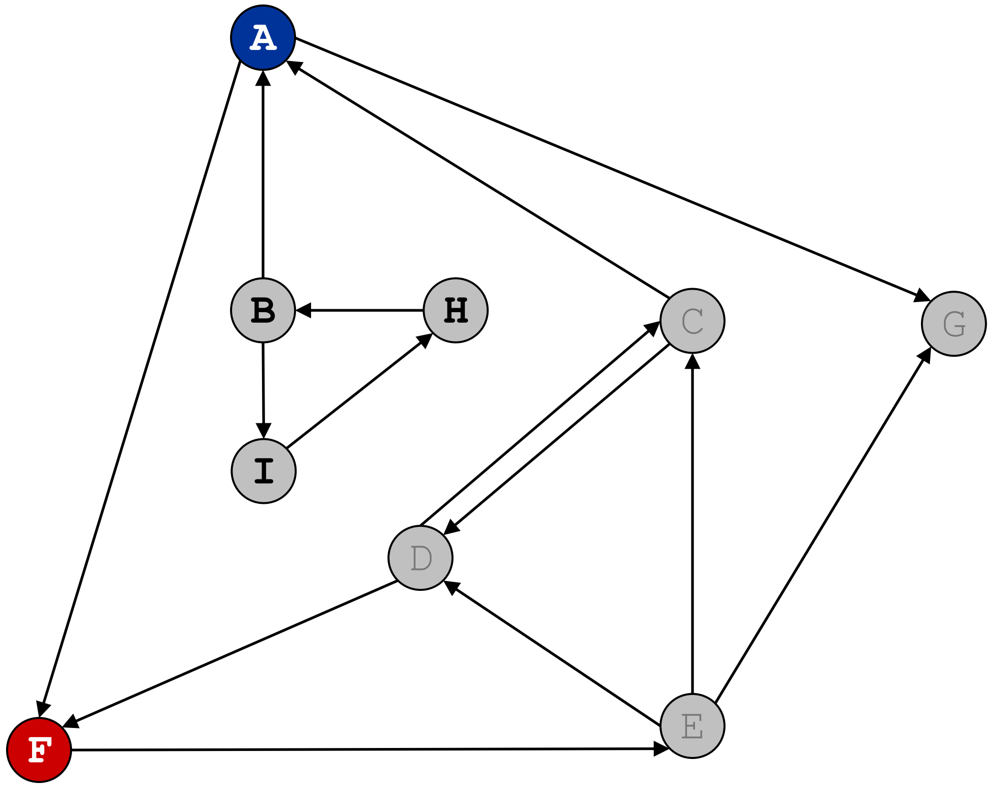
Directed Depth First Search



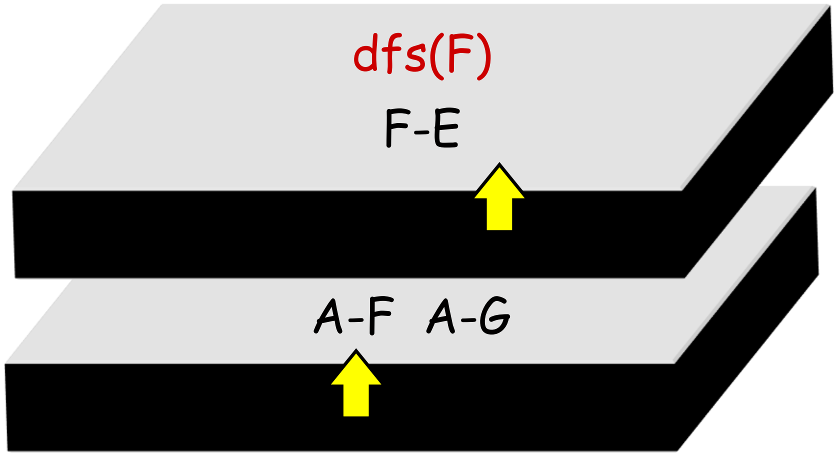
Function call stack:



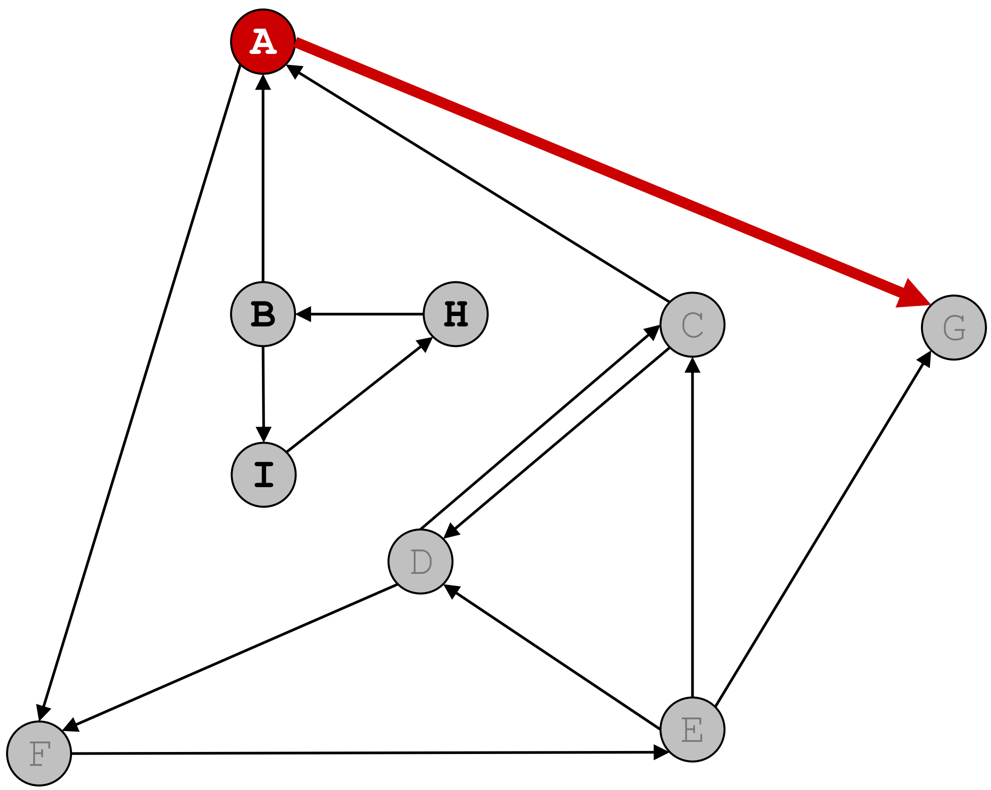
Directed Depth First Search



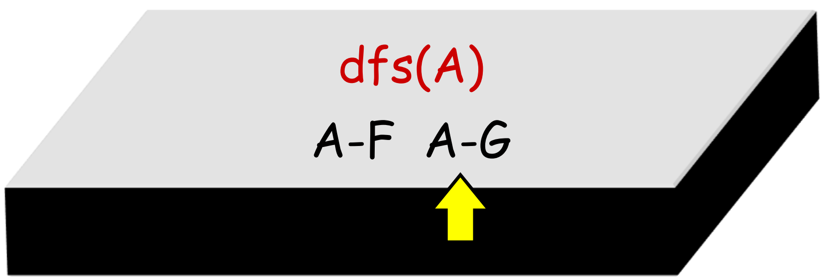
Function call stack:



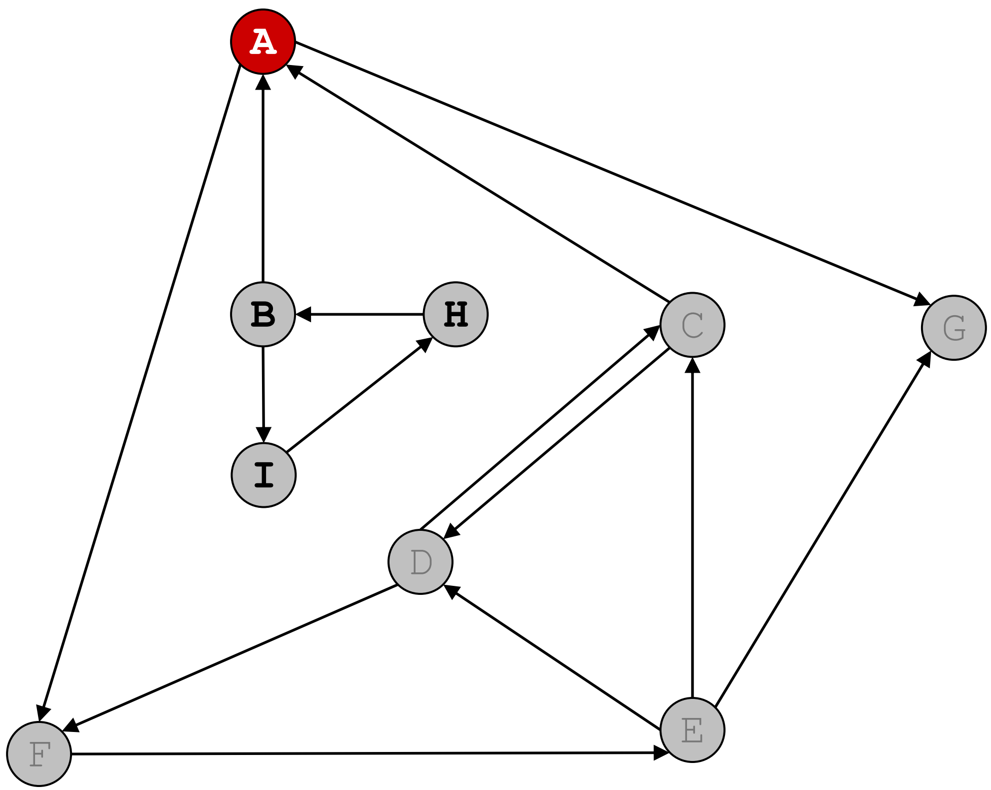
Directed Depth First Search

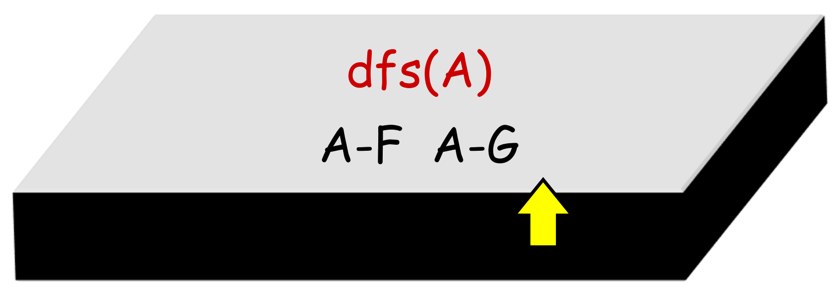


Function call stack:



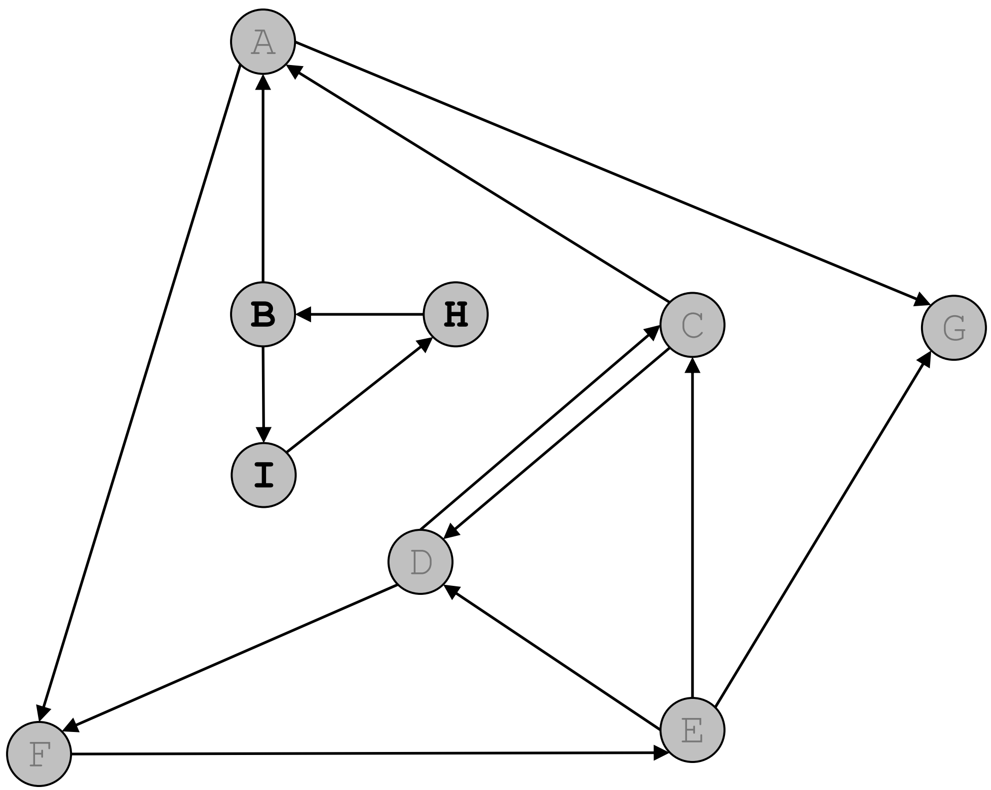
Directed Depth First Search





Function call stack:

Directed Depth First Search



Nodes reachable from A: A, C, D, E, F, G

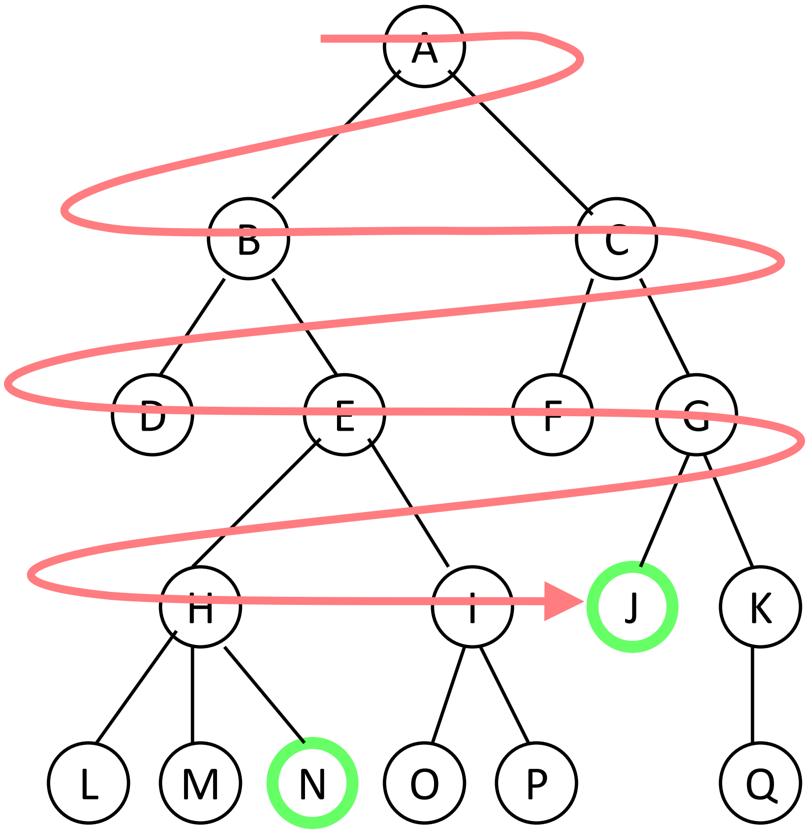
Breadth-First Traversal

- From the starting node, we follow all paths of length one
- Then we follow paths of length two that go to unvisited nodes
- We continue increasing the length of the paths until there are no unvisited nodes along any of the paths

Breadth-First Search

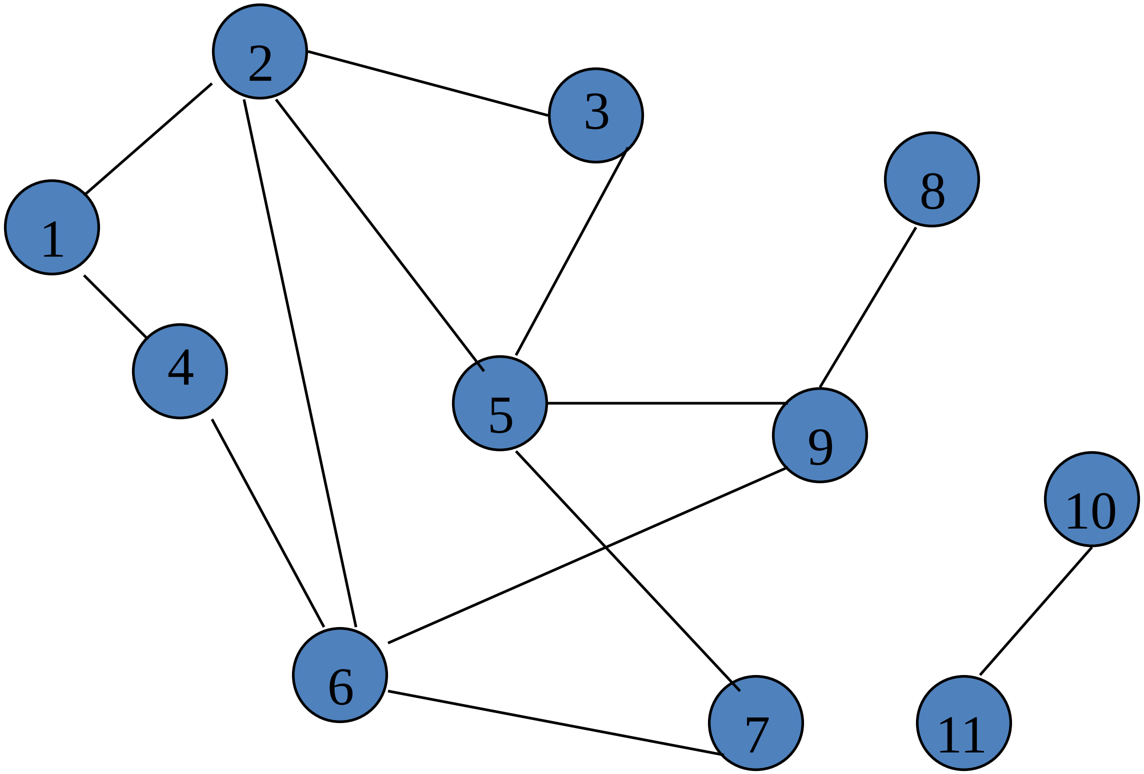
- Visit start vertex and put into a FIFO queue
- Repeatedly remove a vertex from the queue, visit its unvisited adjacent vertices, put newly visited vertices into the queue

Breadth-first searching in a Tree



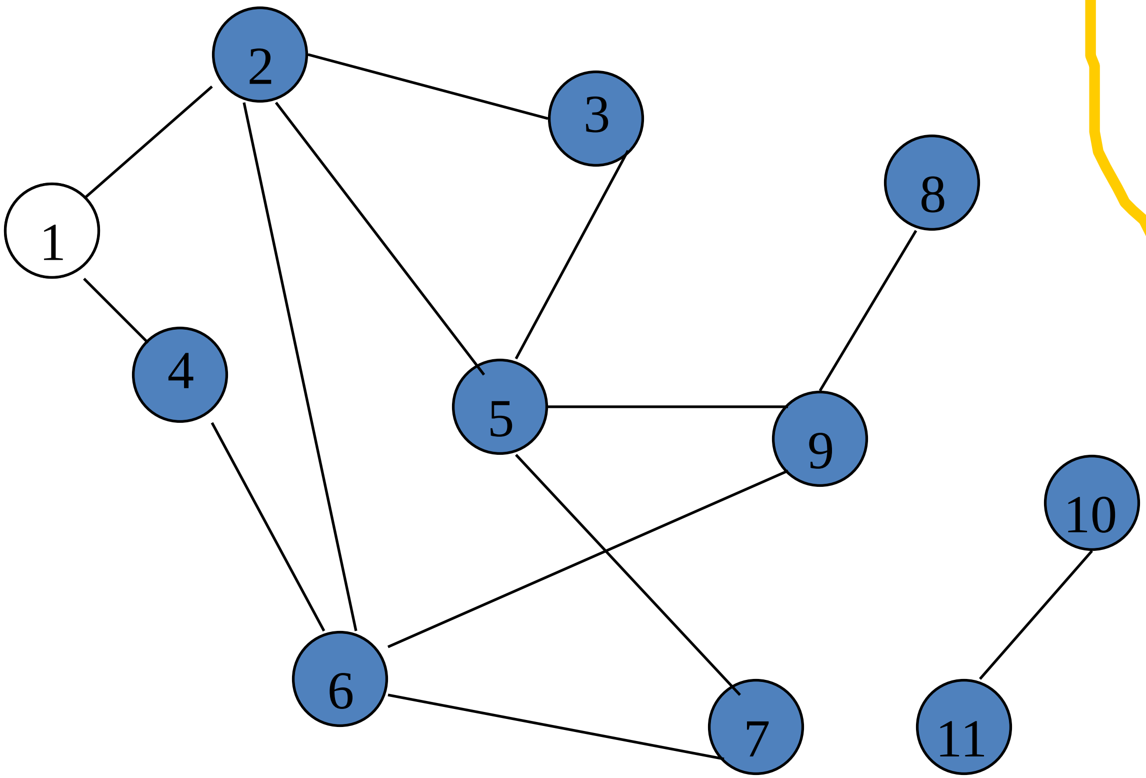
- A breadth-first search (BFS) explores nodes nearest the root before exploring nodes further away
- For example, after searching A, then B, then C, the search proceeds with D, E, F, G
- Node are explored in the order A B C D E F G H I J K L M N O P Q
- J will be found before N

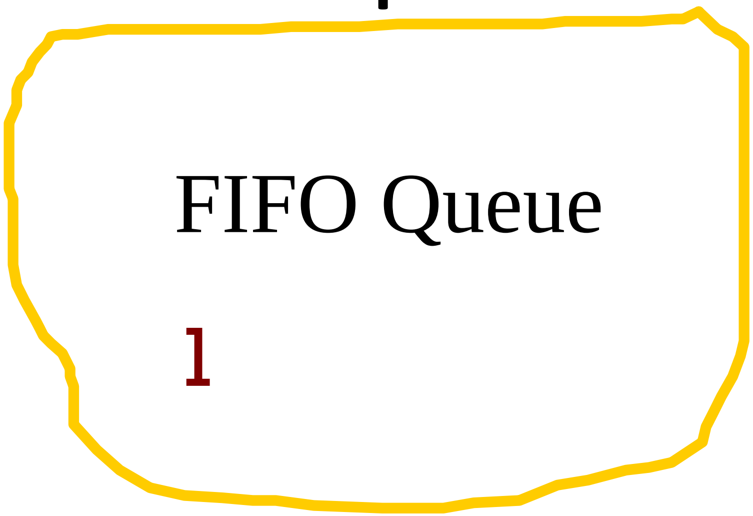
Breadth-First Search Example



- Start search at vertex 1

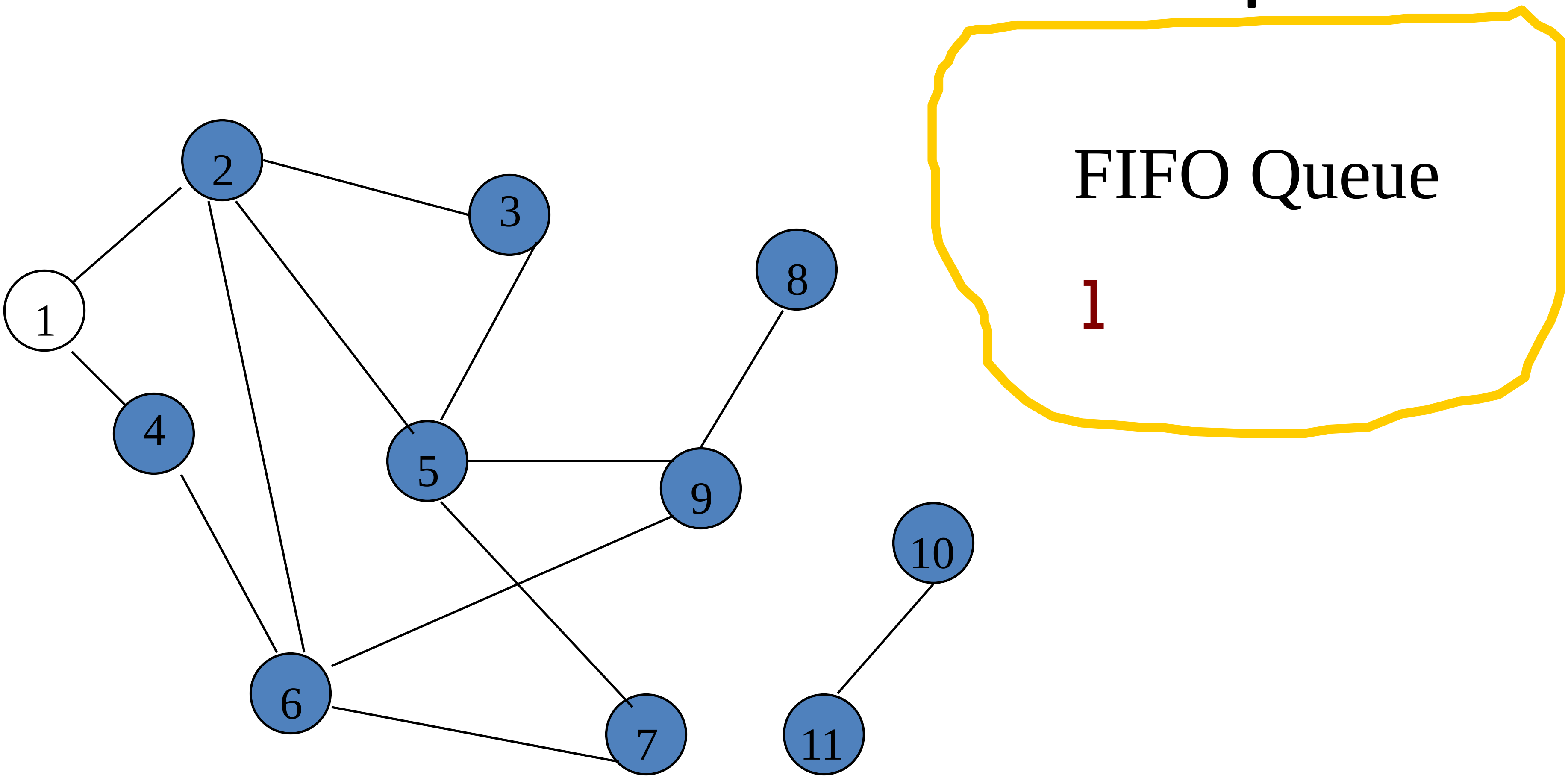
Breadth-First Search Example





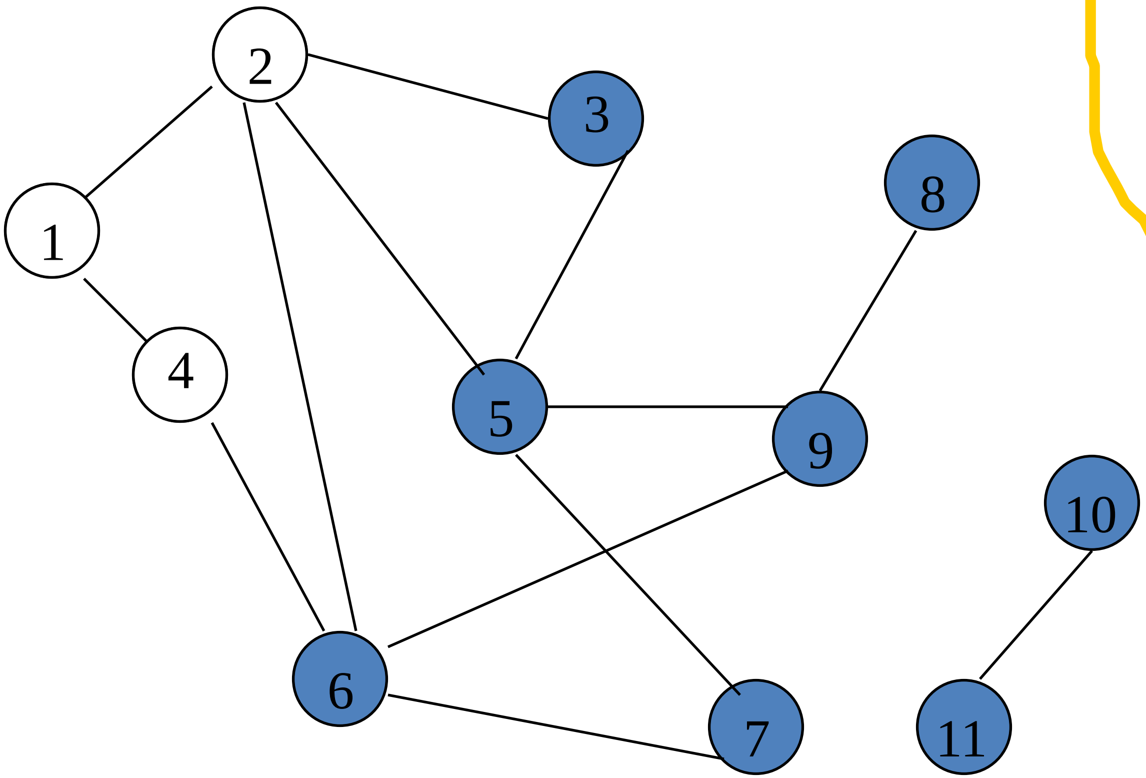
- Visit/mark/label start vertex and put in a FIFO queue

Breadth-First Search Example



- Remove 1 from Q
- Visit adjacent unvisited vertices & put them in Q

Breadth-First Search Example

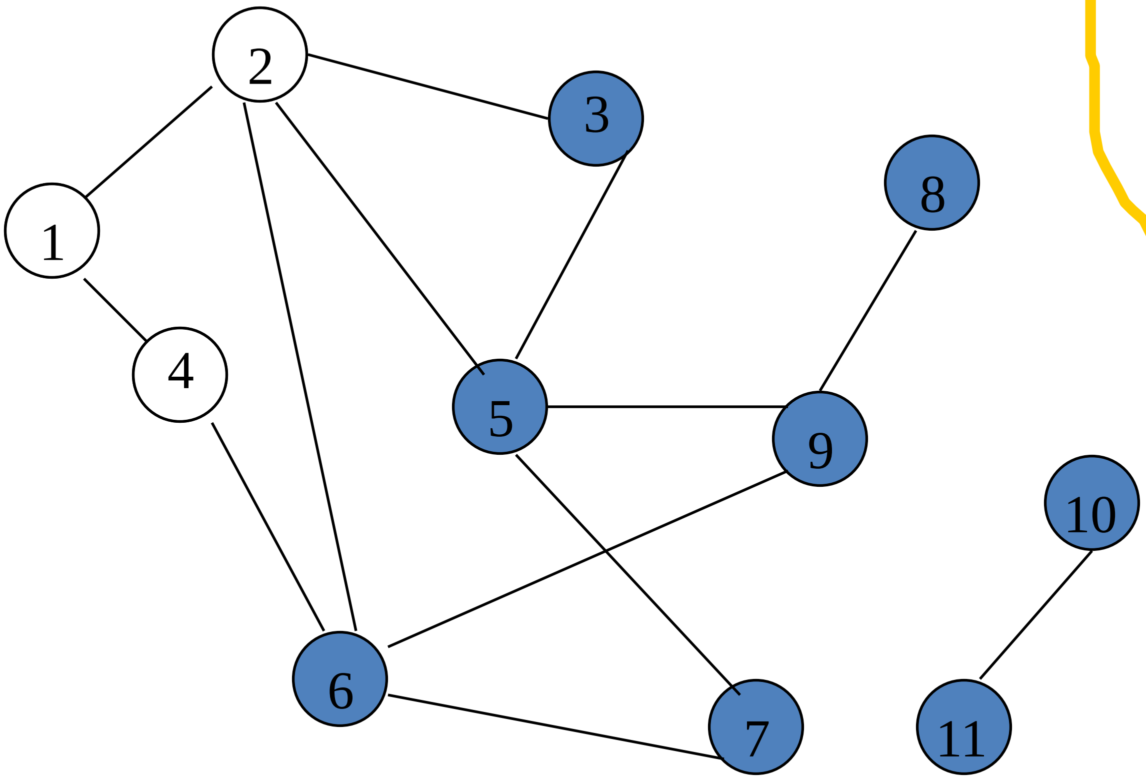


FIFO Queue

2 4

- Remove 1 from Q
- Visit adjacent unvisited vertices & put them in Q

Breadth-First Search Example

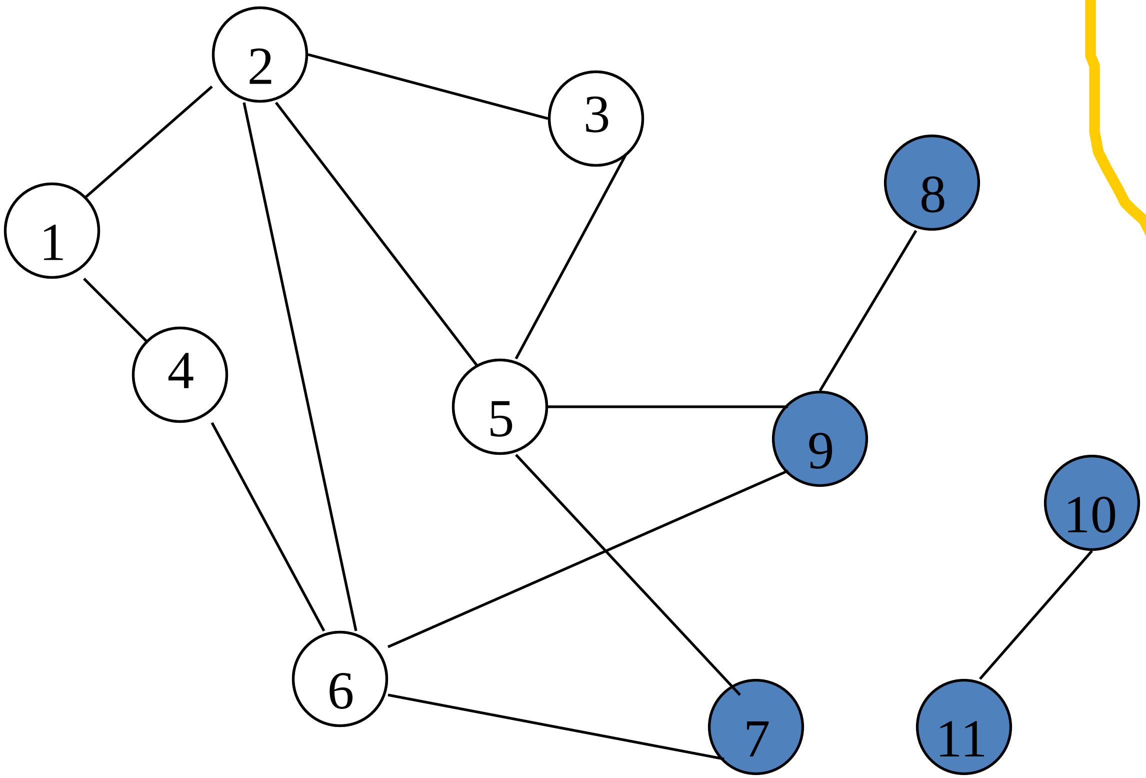


FIFO Queue

2 4

- Remove 2 from Q
- Visit adjacent unvisited vertices & put them in Q

Breadth-First Search Example

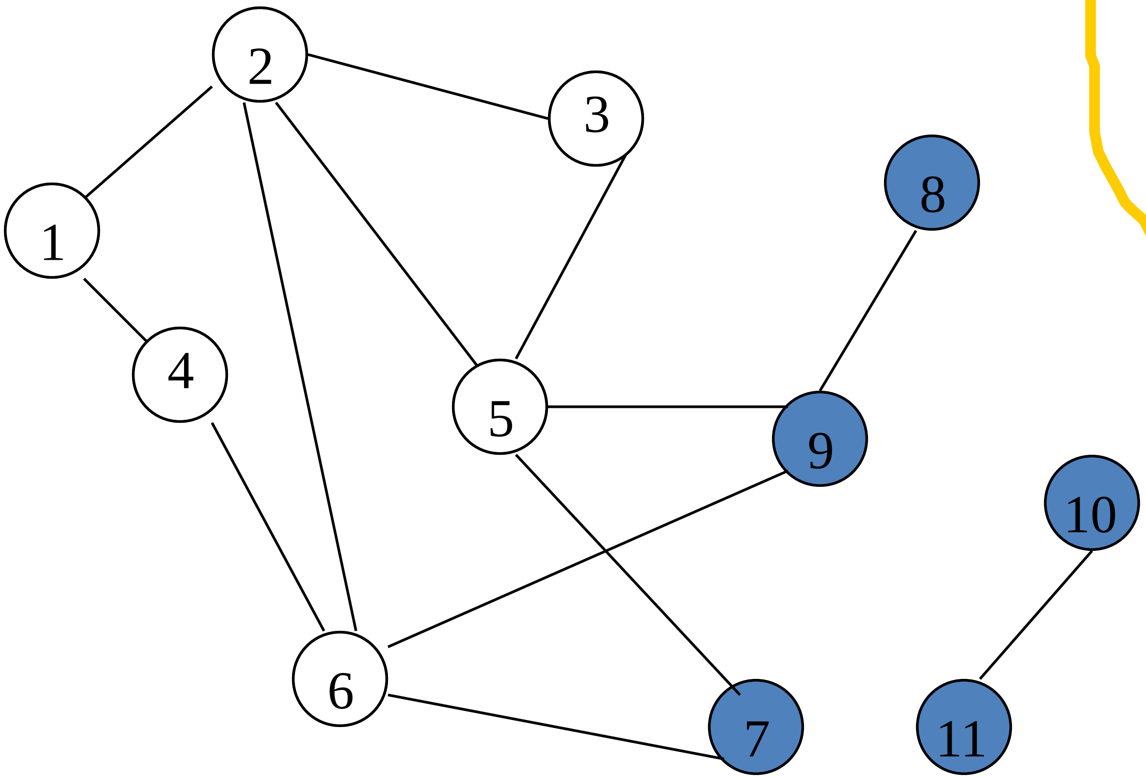


FIFO Queue

4 5 3 6

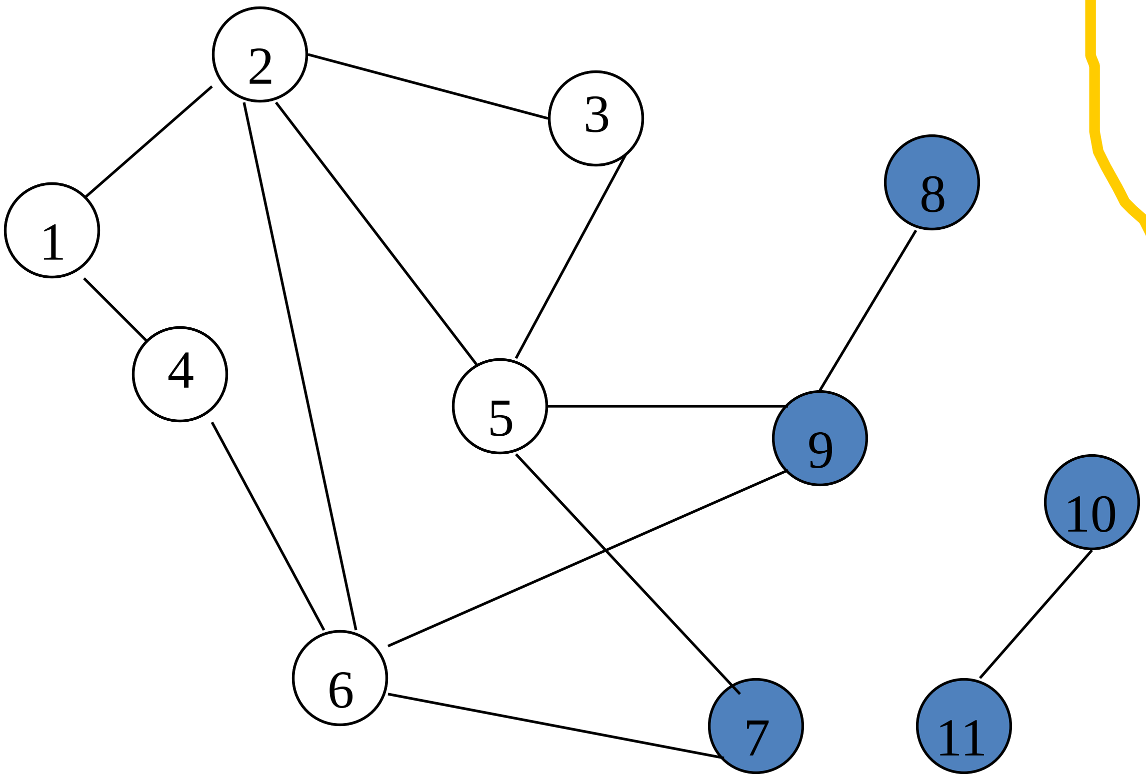
- Remove 2 from Q
- Visit adjacent unvisited vertices & put them in Q

Breadth-First Search Example



- Remove 4 from Q
- Visit adjacent unvisited vertices & put them in Q

Breadth-First Search Example

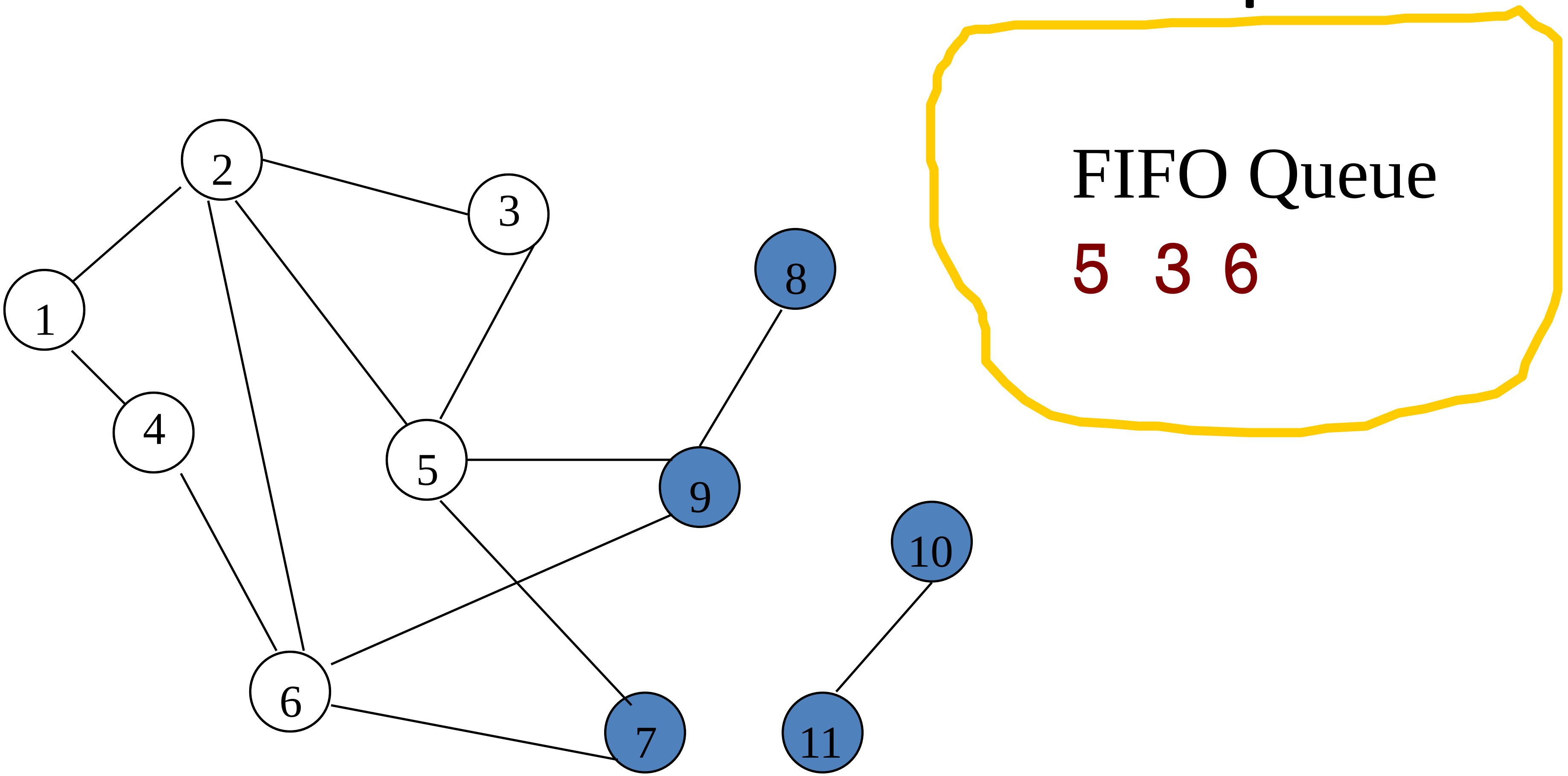


FIFO Queue

5 3 6

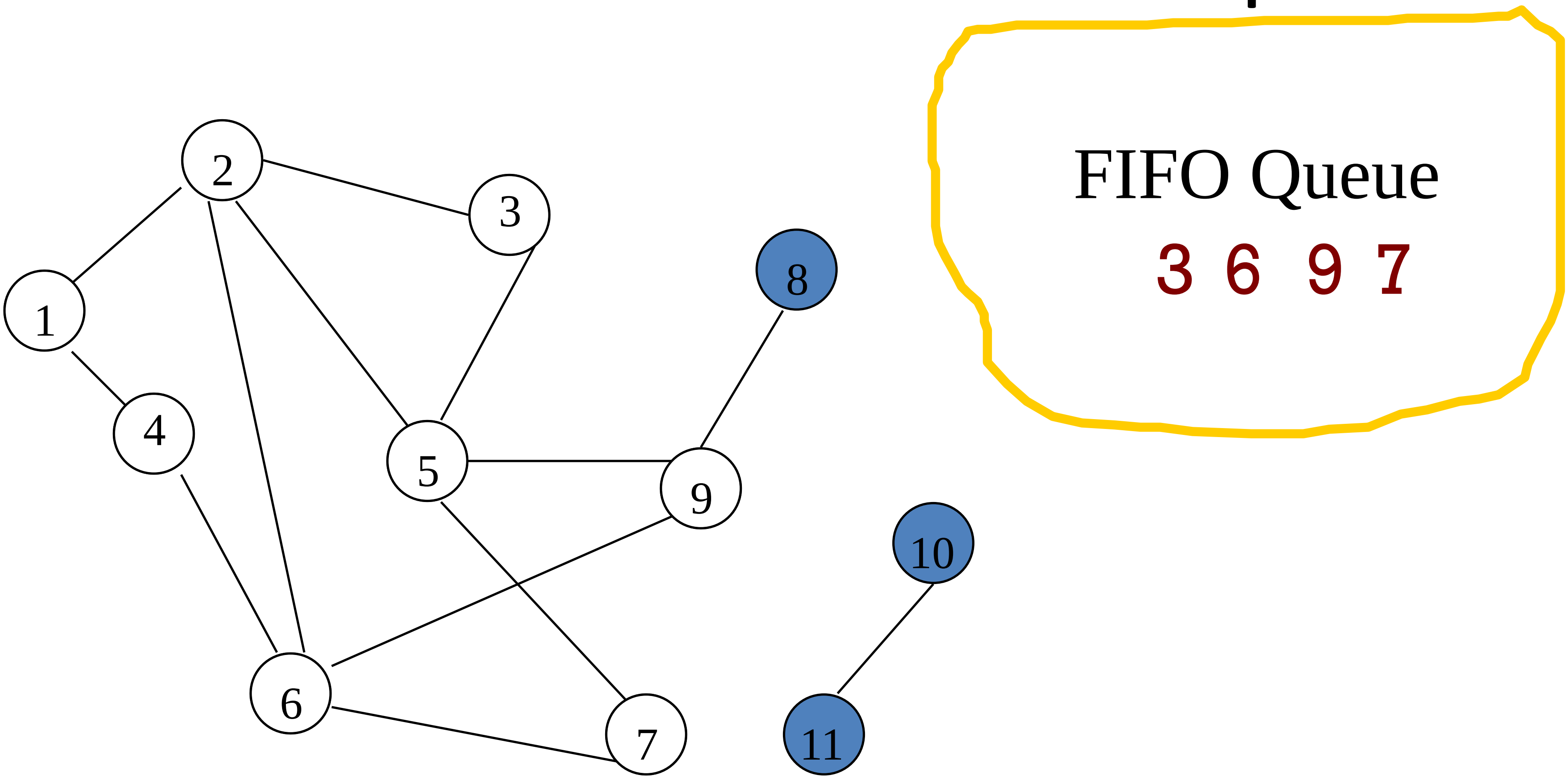
- Remove 4 from Q
- Visit adjacent unvisited vertices & put them in Q

Breadth-First Search Example



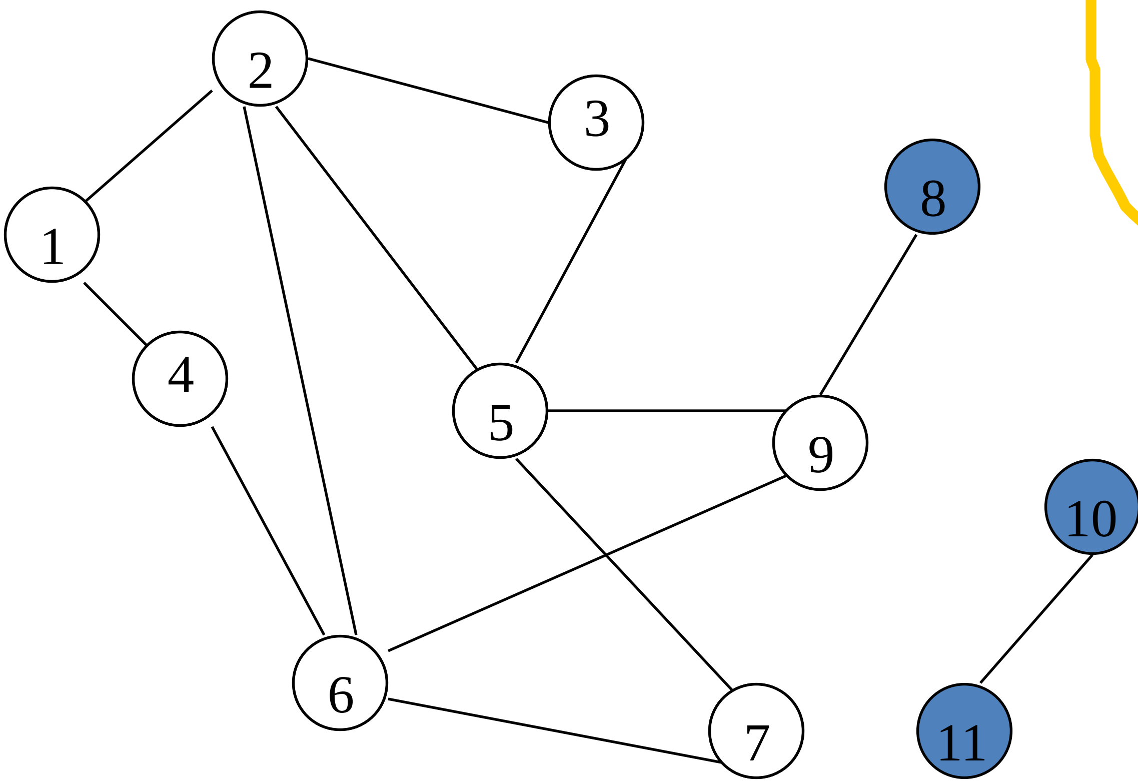
- Remove 5 from Q
- Visit adjacent unvisited vertices & put them in Q

Breadth-First Search Example



- Remove 5 from Q
- Visit adjacent unvisited vertices & put them in Q

Breadth-First Search Example

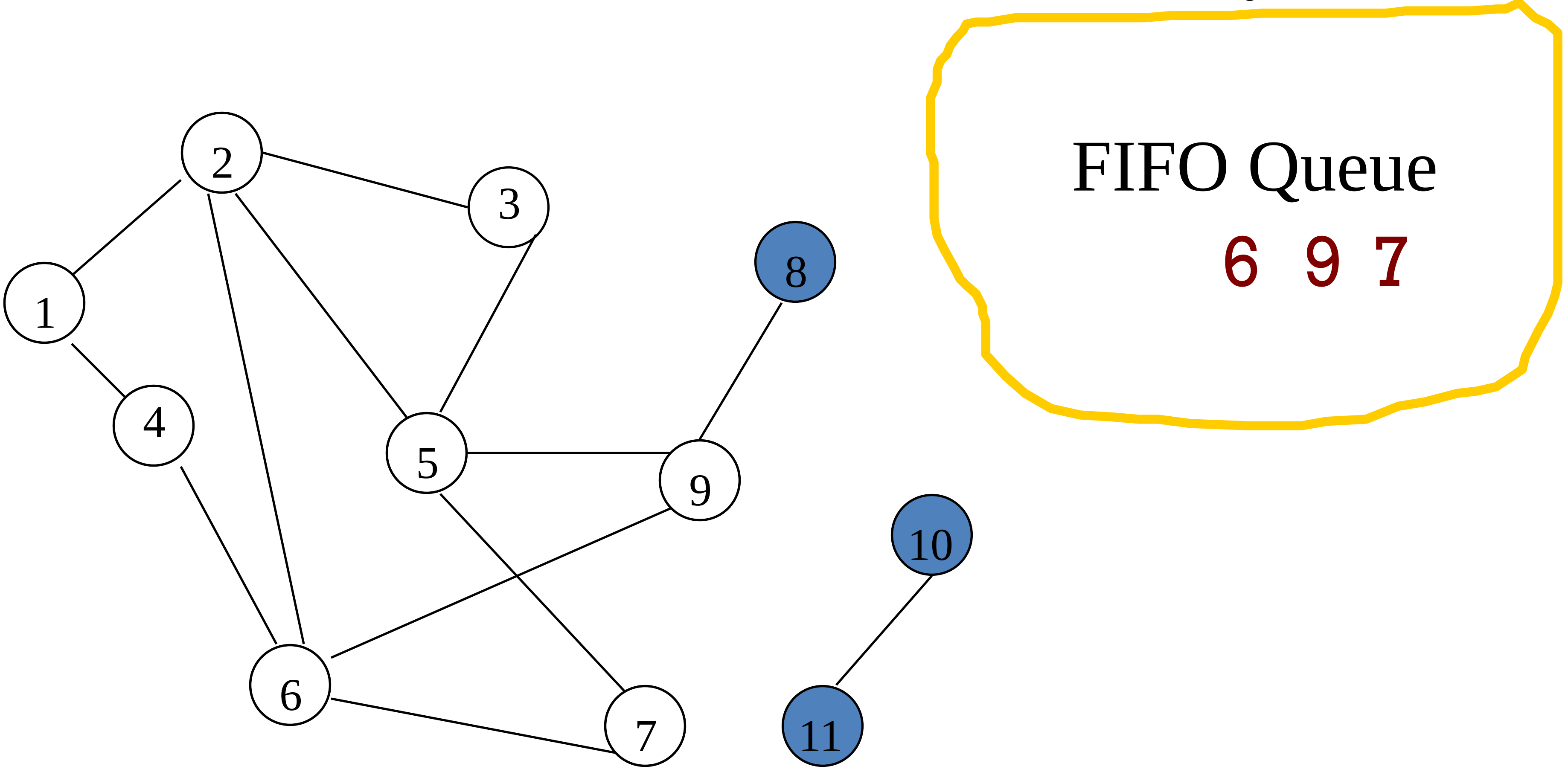


FIFO Queue

3 6 9 7

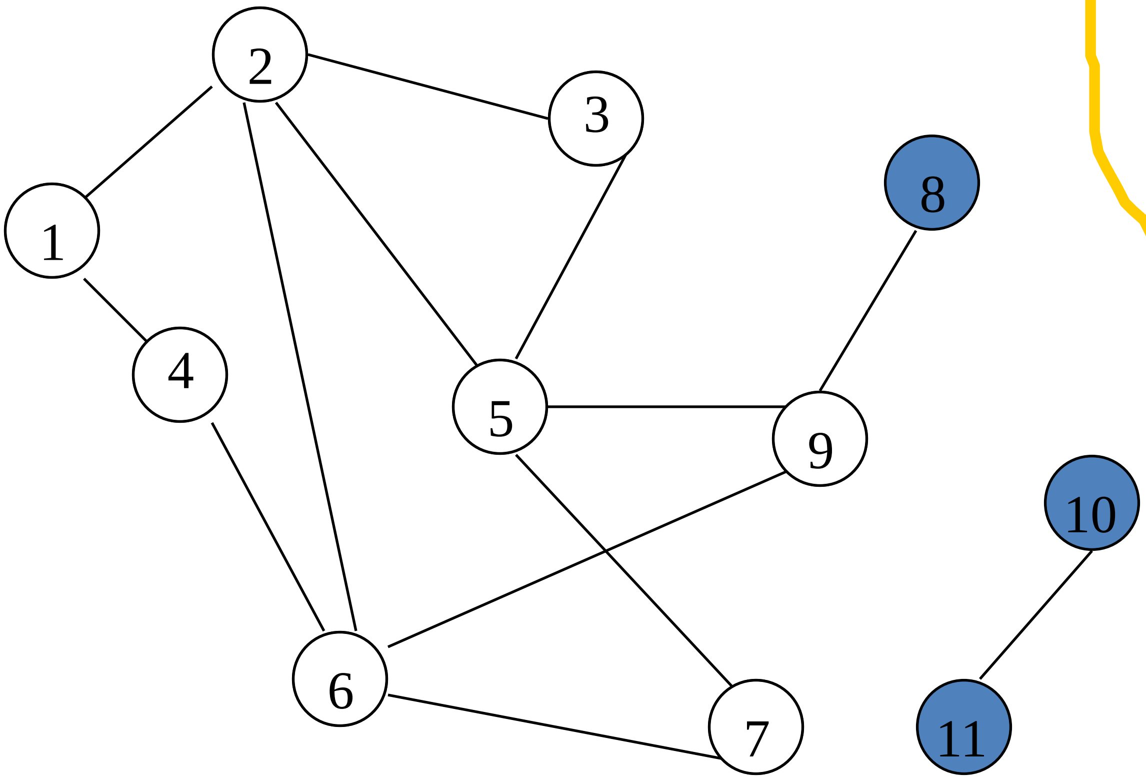
- Remove 3 from Q
- Visit adjacent unvisited vertices & put them in Q

Breadth-First Search Example



- Remove 3 from Q
- Visit adjacent unvisited vertices & put them in Q

Breadth-First Search Example

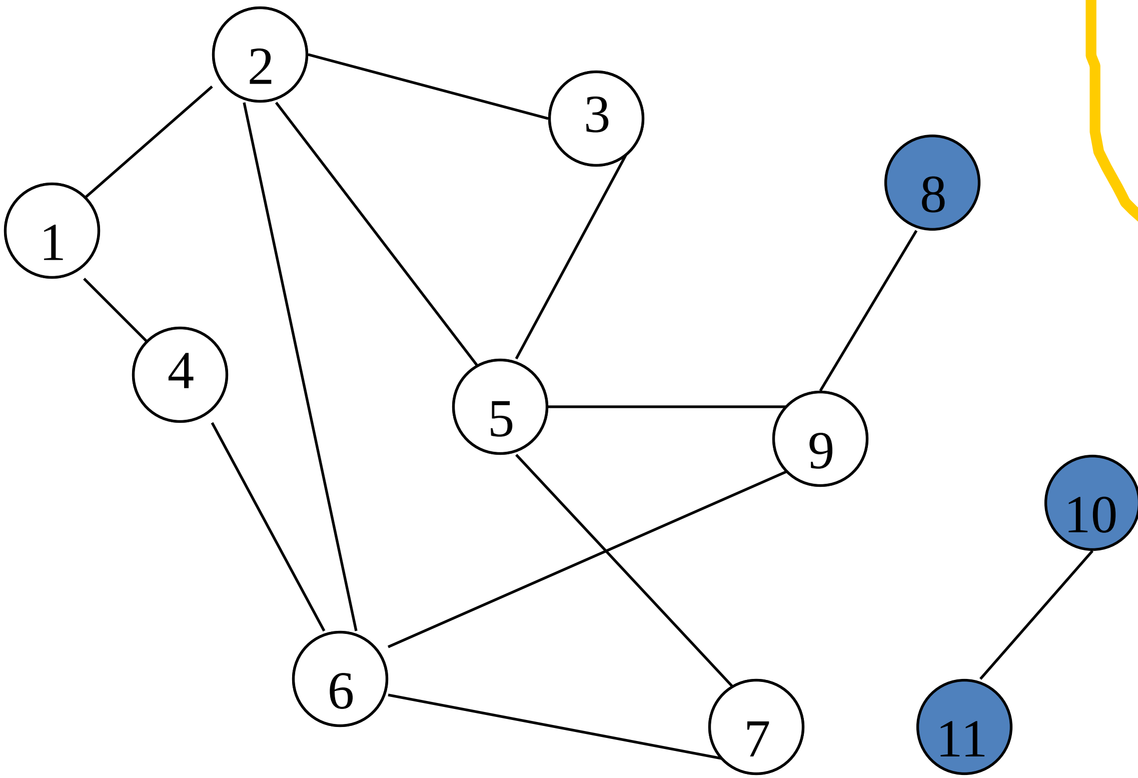


FIFO Queue

6 9 7

- Remove 6 from Q
- Visit adjacent unvisited vertices & put them in Q

Breadth-First Search Example

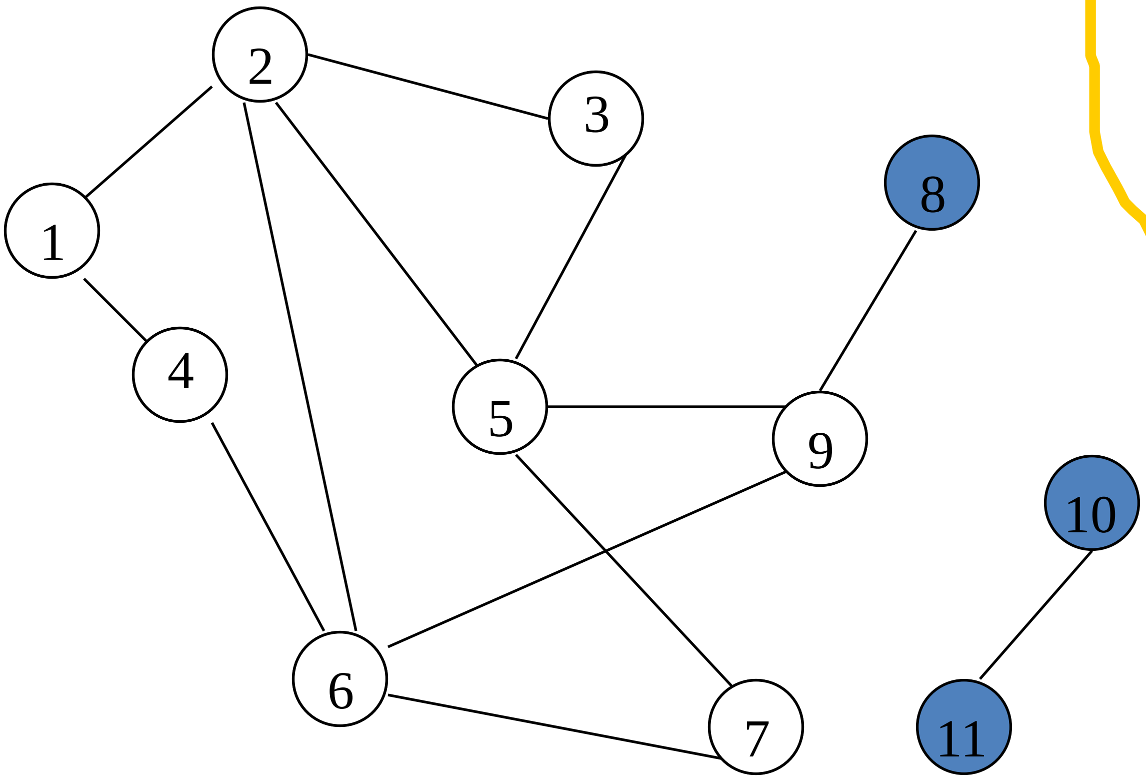


FIFO Queue

9 7

- Remove 6 from Q
- Visit adjacent unvisited vertices & put them in Q

Breadth-First Search Example

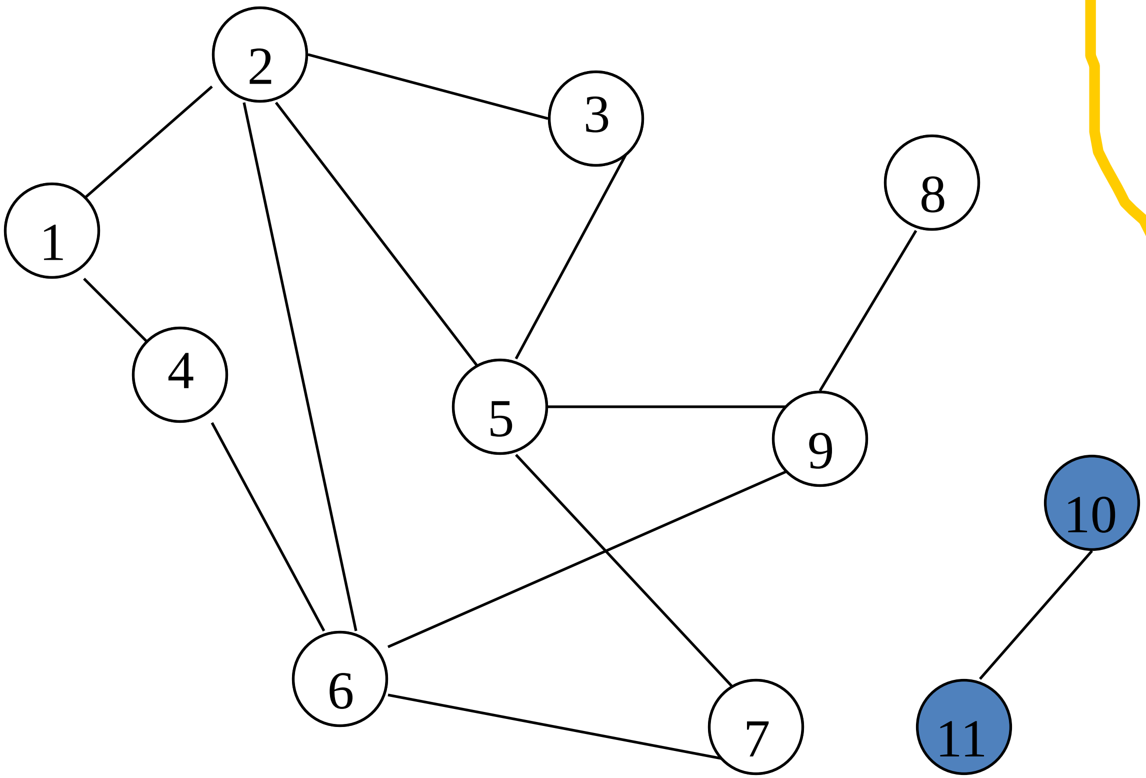


FIFO Queue

9 7

- Remove 9 from Q
- Visit adjacent unvisited vertices & put them in Q

Breadth-First Search Example

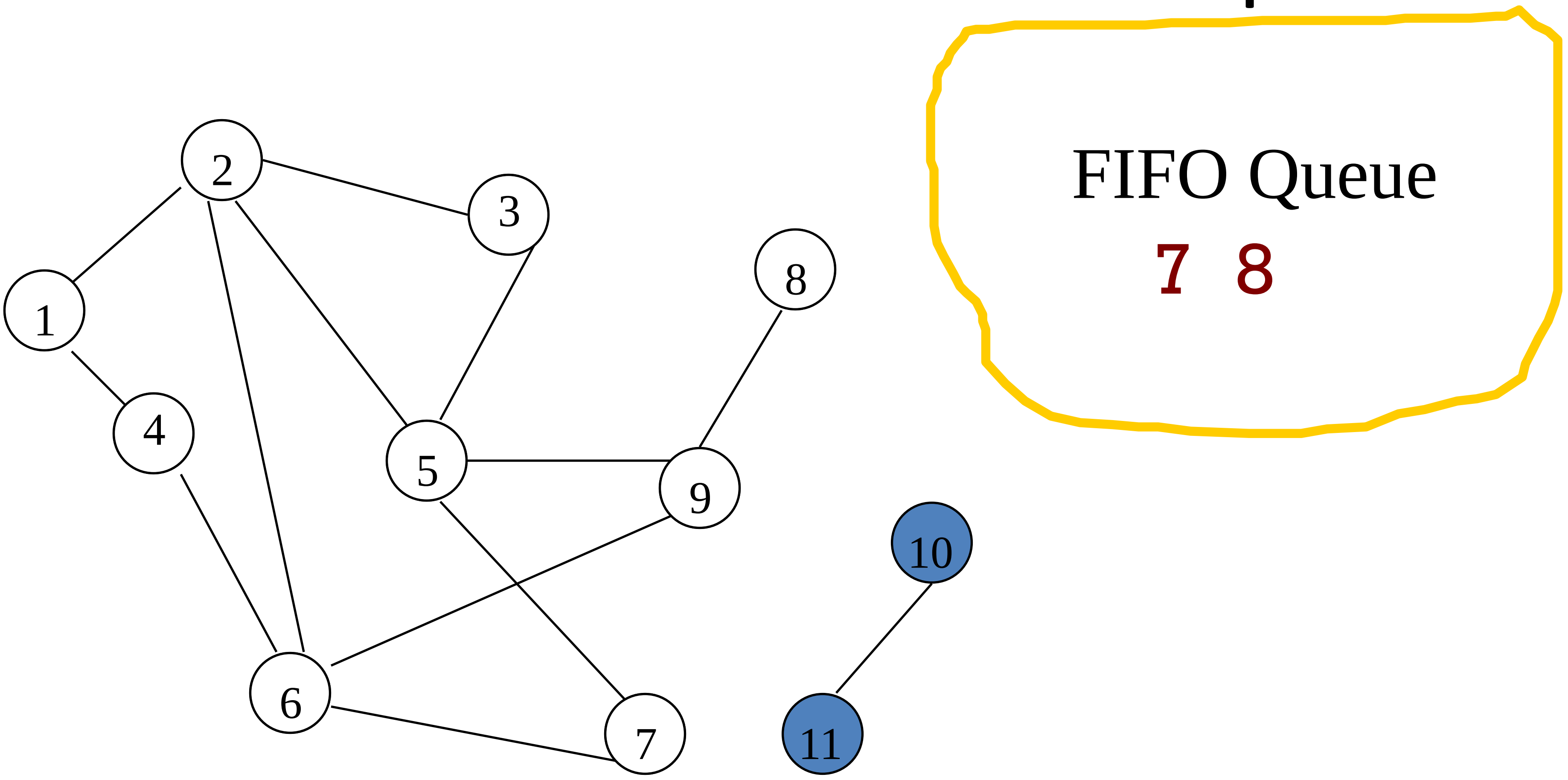


FIFO Queue

7 8

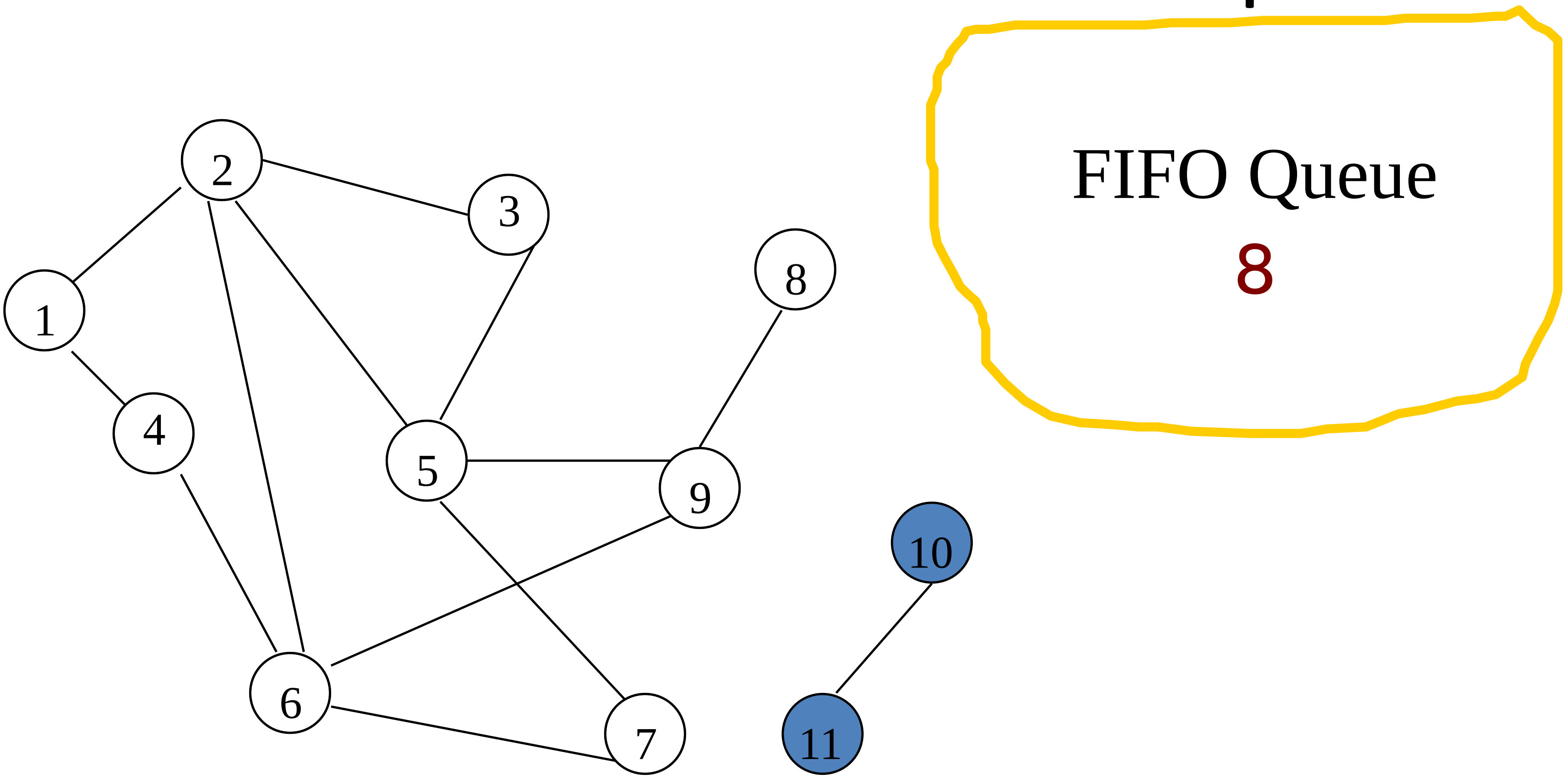
- Remove 9 from Q
- Visit adjacent unvisited vertices & put them in Q

Breadth-First Search Example



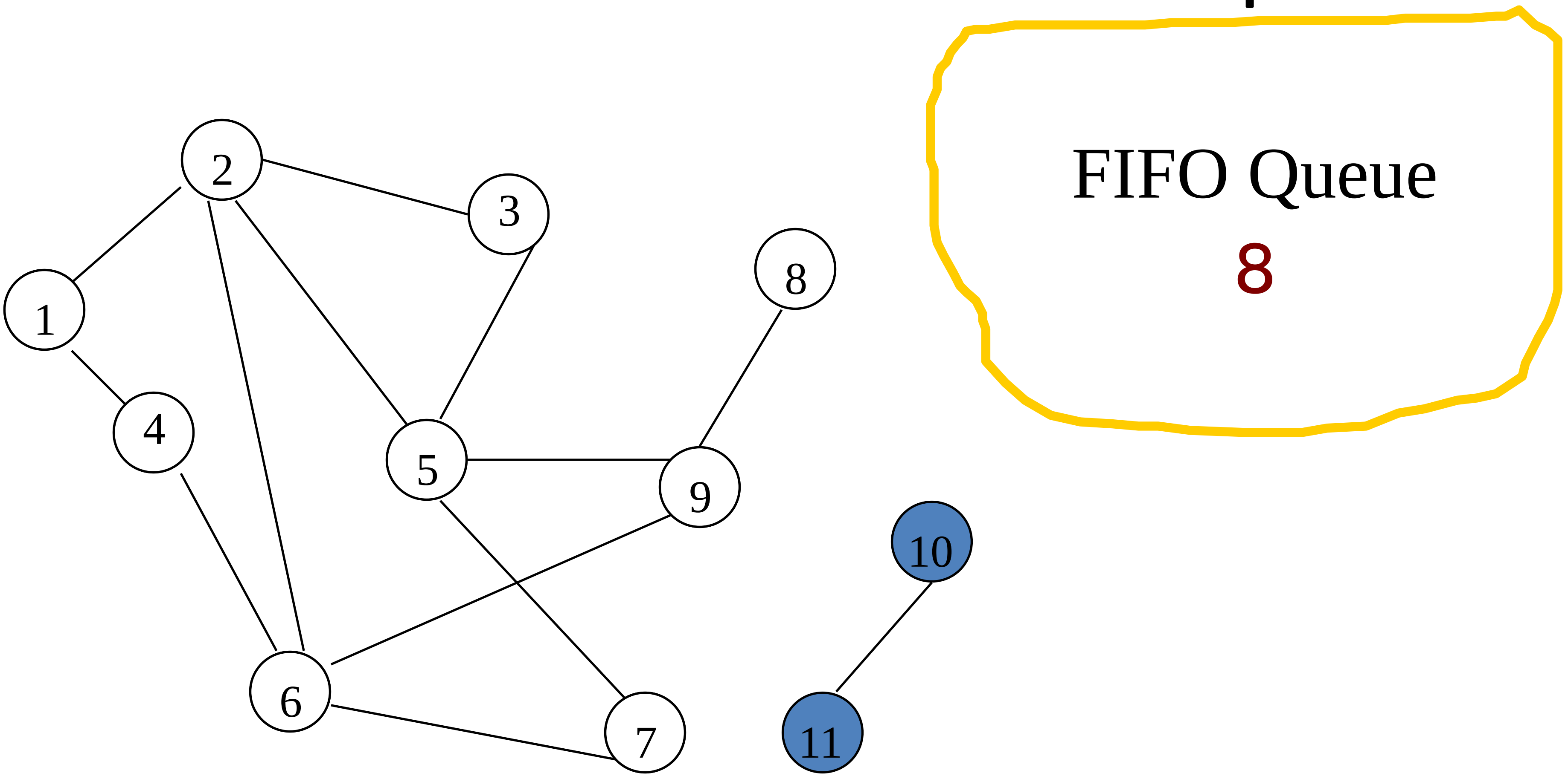
- Remove 7 from Q
- Visit adjacent unvisited vertices & put them in Q

Breadth-First Search Example



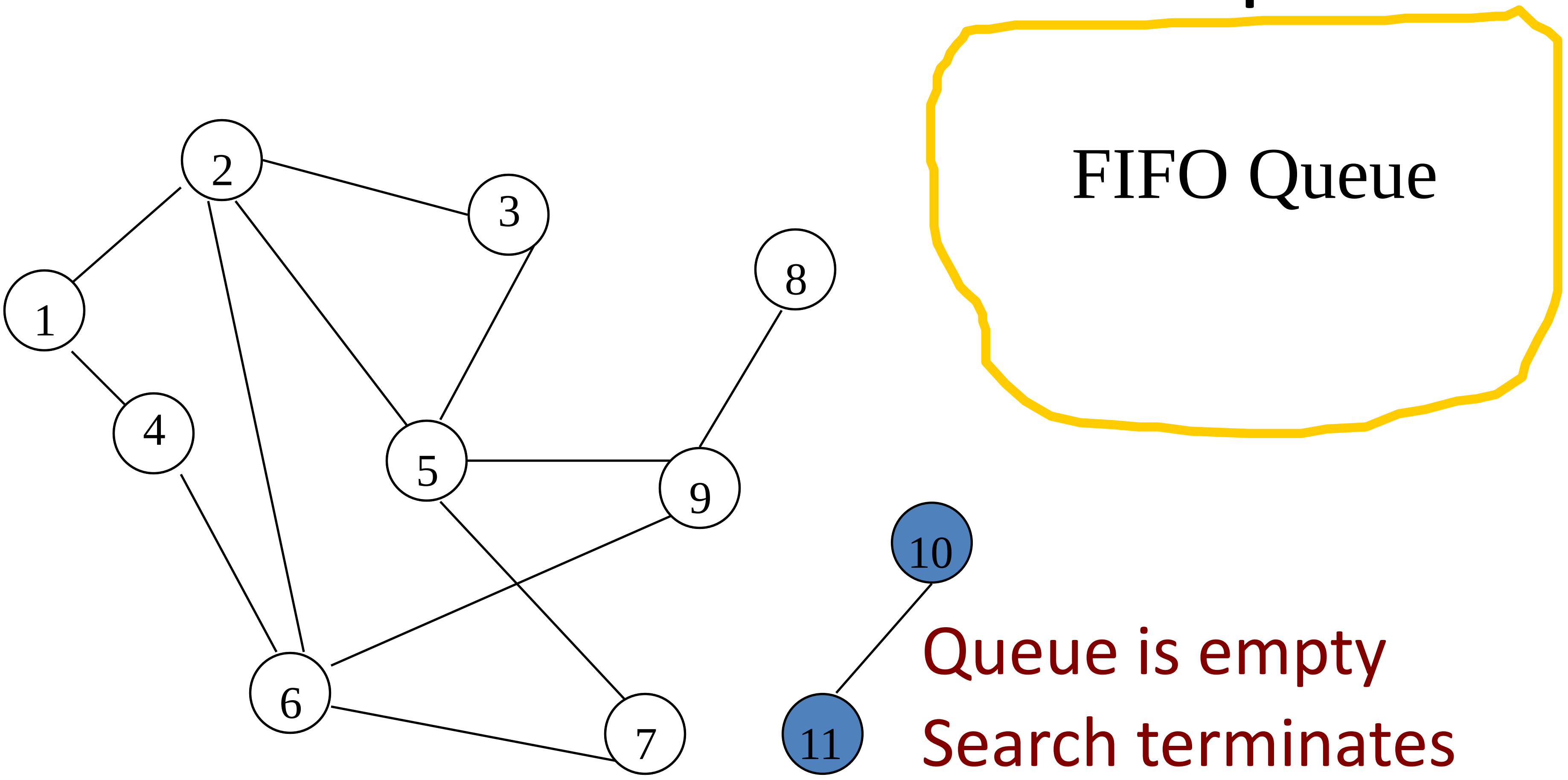
- Remove 7 from Q
- Visit adjacent unvisited vertices & put them in Q

Breadth-First Search Example



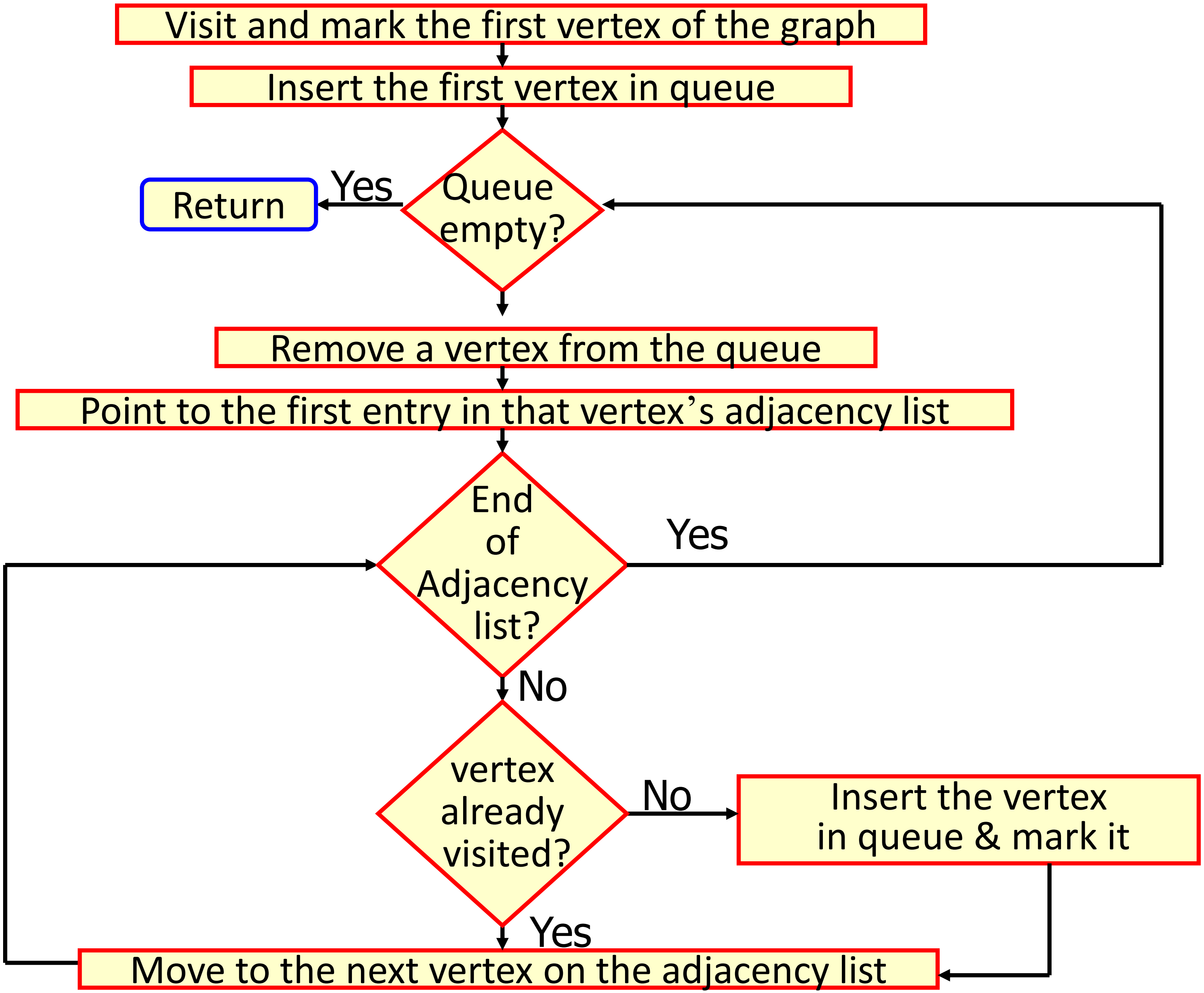
- Remove 8 from Q
- Visit adjacent unvisited vertices & put them in Q

Breadth-First Search Example



- All vertices reachable from the start vertex (including the start vertex) are visited

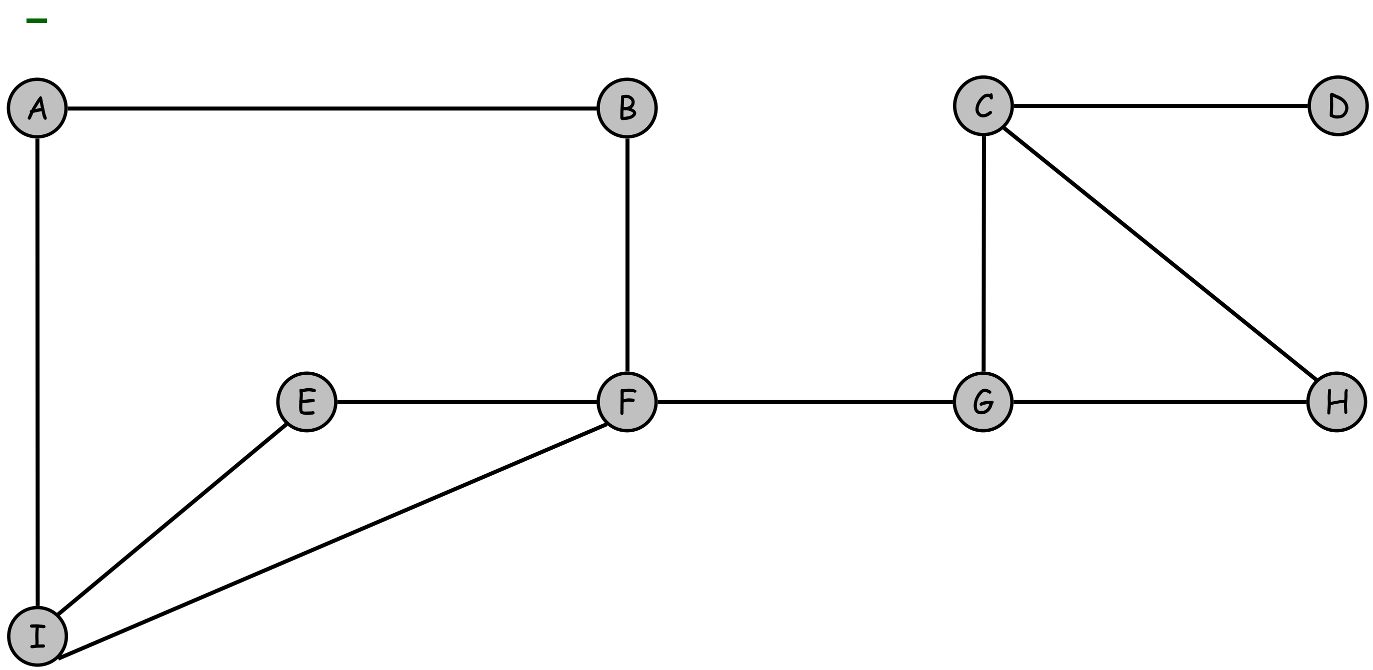
BFS- Flowchart



BFS (Pseudo Code)

```
BFS(input: graph G) {
    Queue Q;    Integer x, z, y;
    while (G has an unvisited node x) {
        visit(x); Enqueue(x,Q);
        while (Q is not empty){
            z := Dequeue(Q);
            for all (unvisited neighbor y of z){
                visit(y); Enqueue(y,Q);
            }
        }
    }
}
```

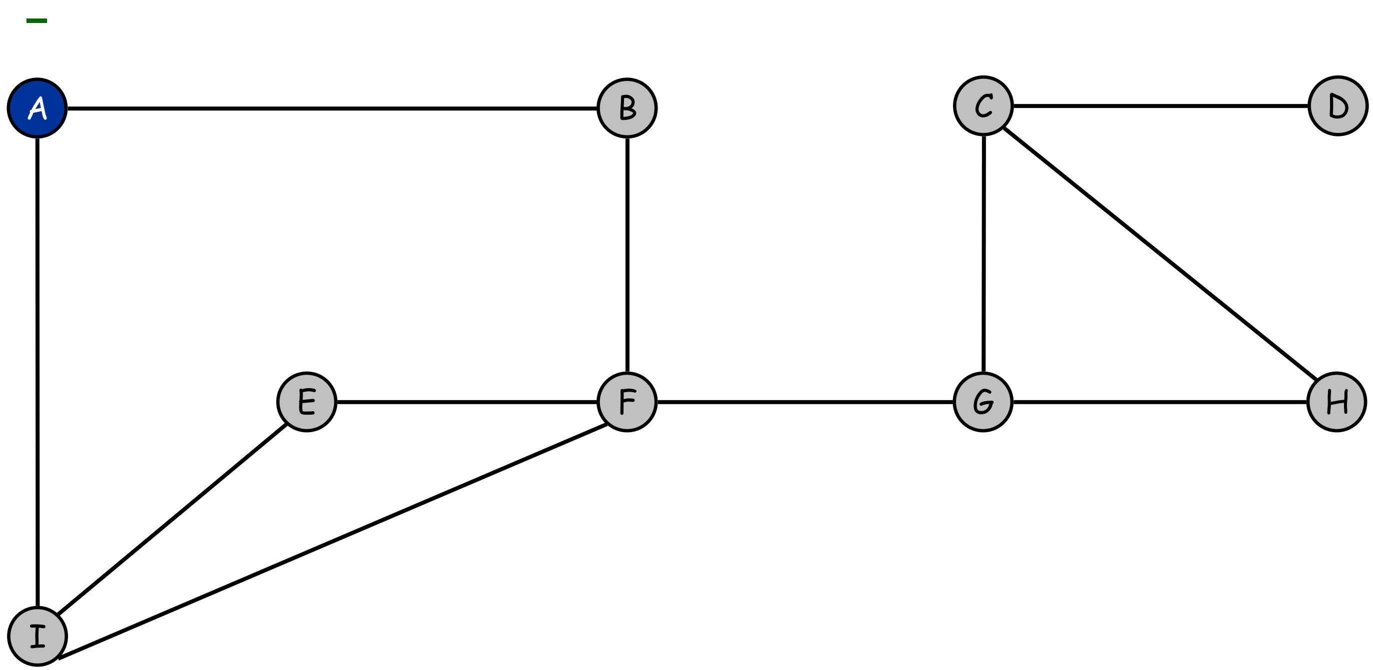
Breadth First Search



front

FIFO Queue

Breadth First Search

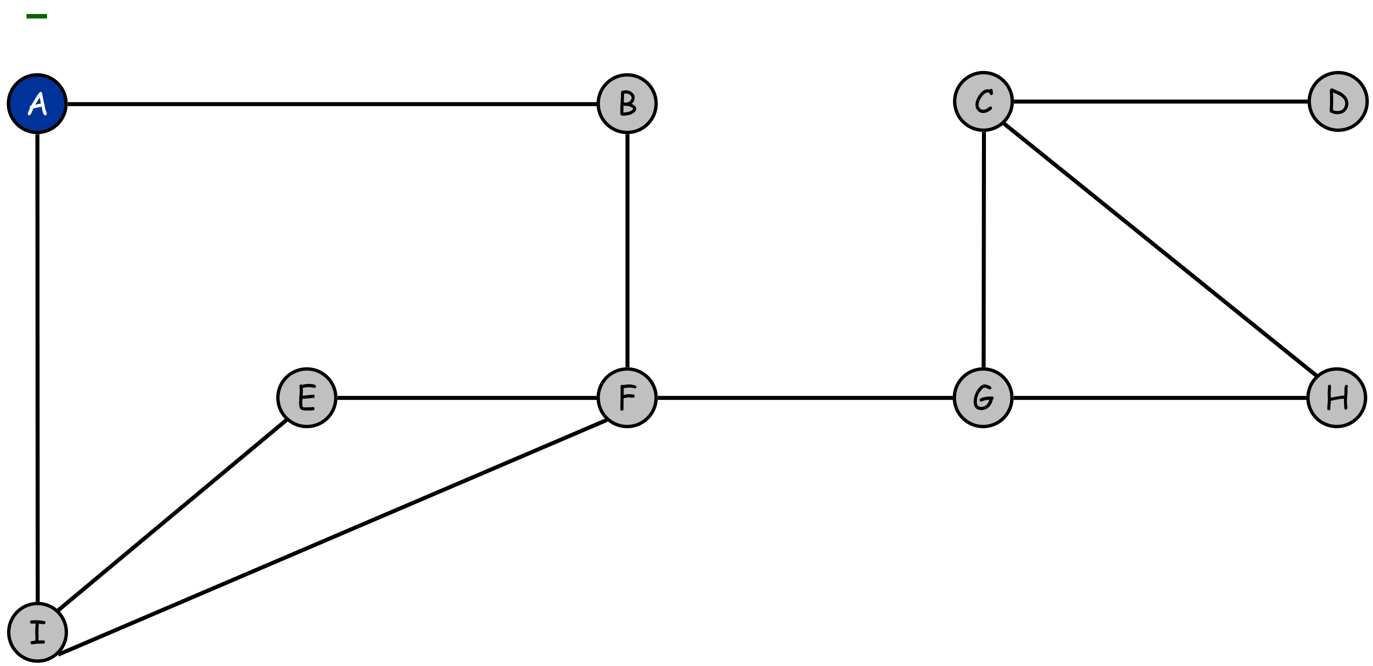


enqueue source node

front **A**

FIFO Queue

Breadth First Search

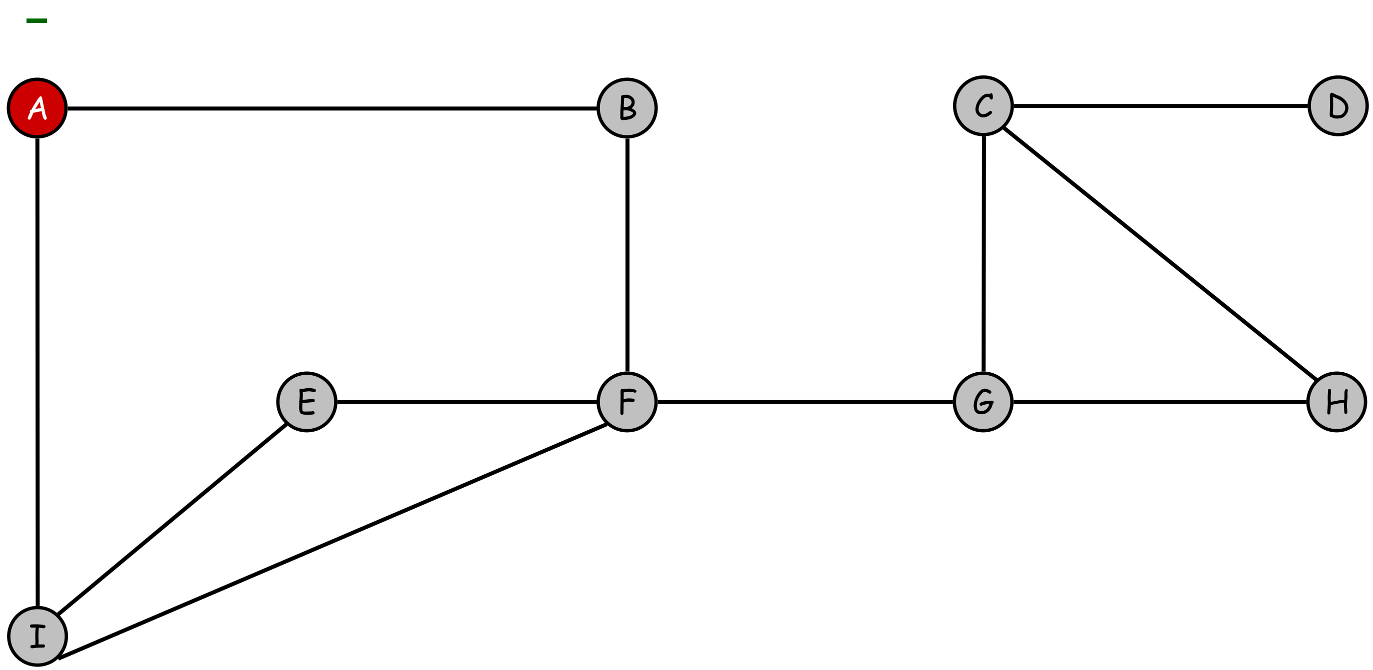


dequeue next vertex

front **A**

FIFO Queue

Breadth First Search

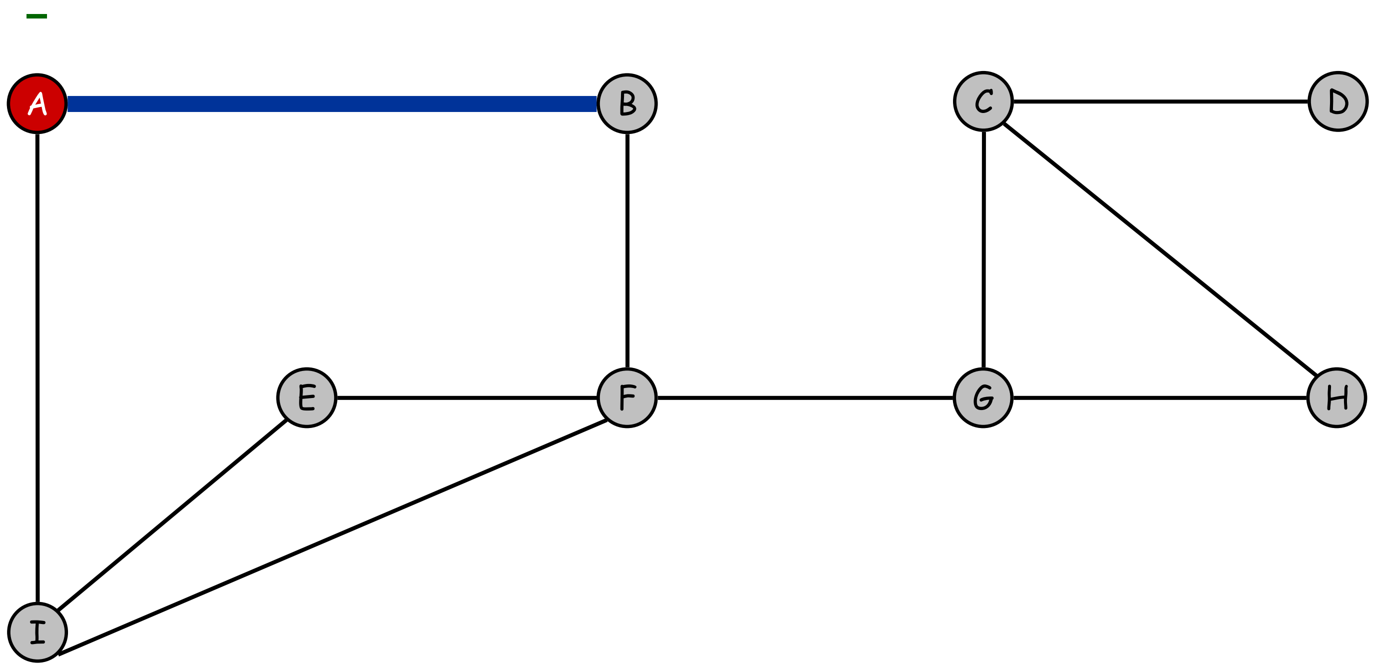


visit neighbors of A

front

FIFO Queue

Breadth First Search

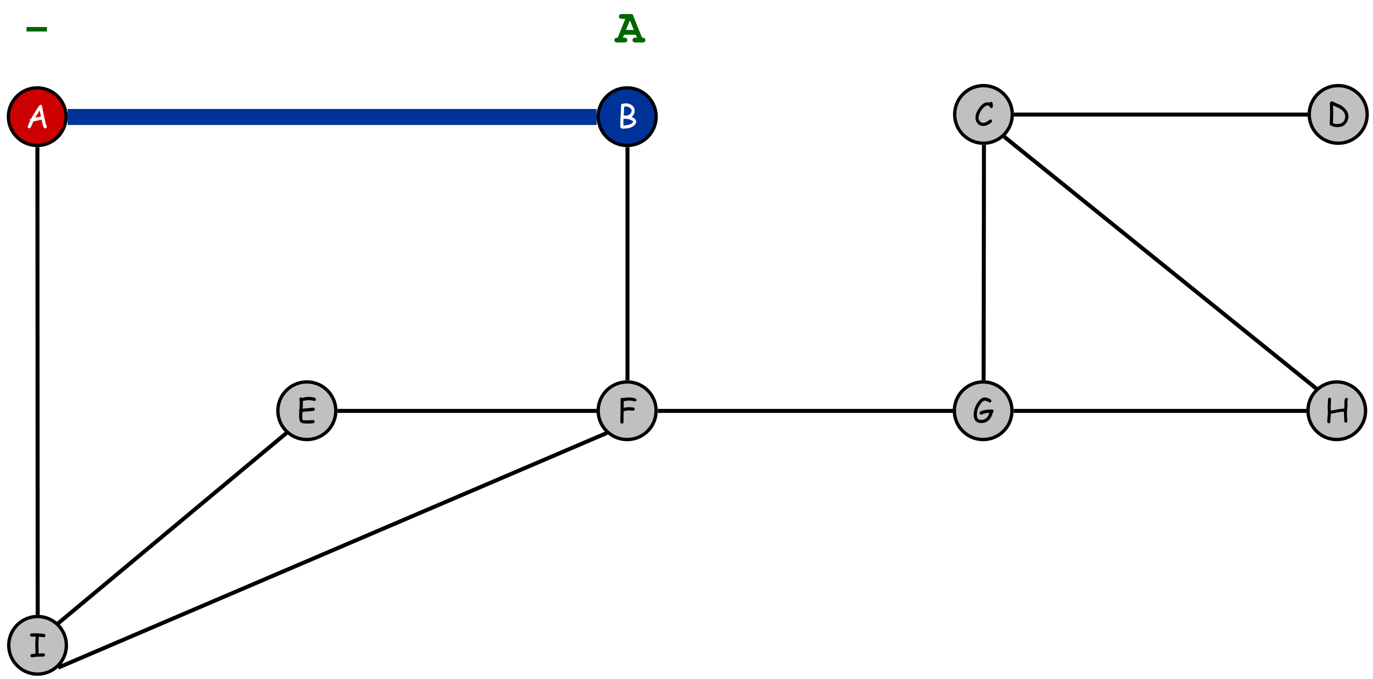


visit neighbors of A

front

FIFO Queue

Breadth First Search

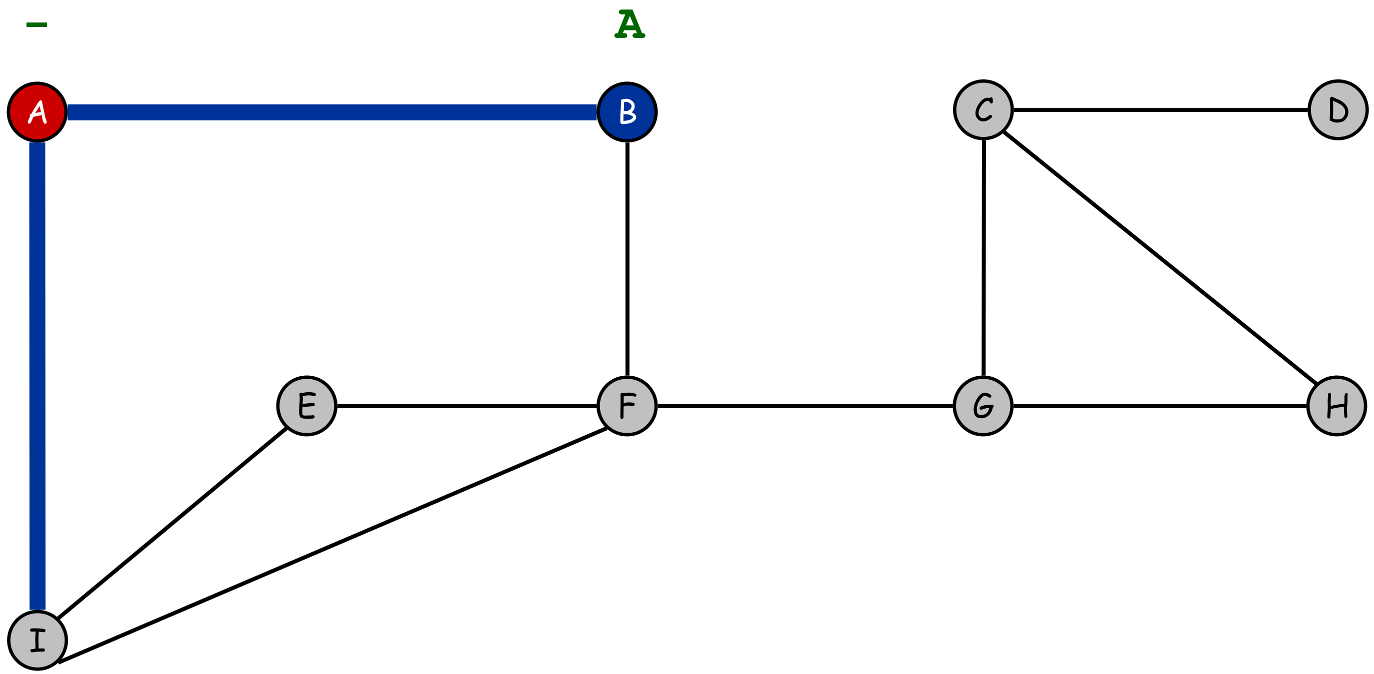


B discovered

front **B**

FIFO Queue

Breadth First Search



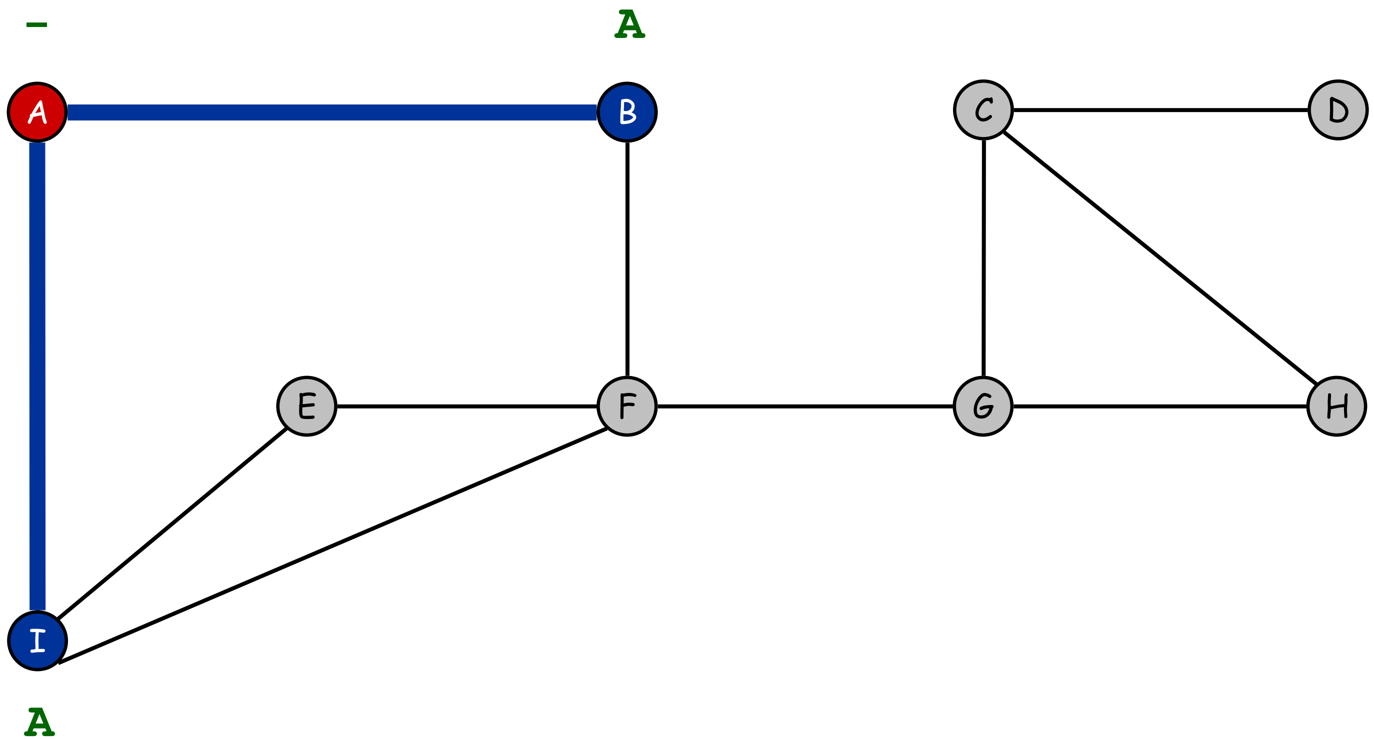
visit neighbors of A

front

B

FIFO Queue

Breadth First Search

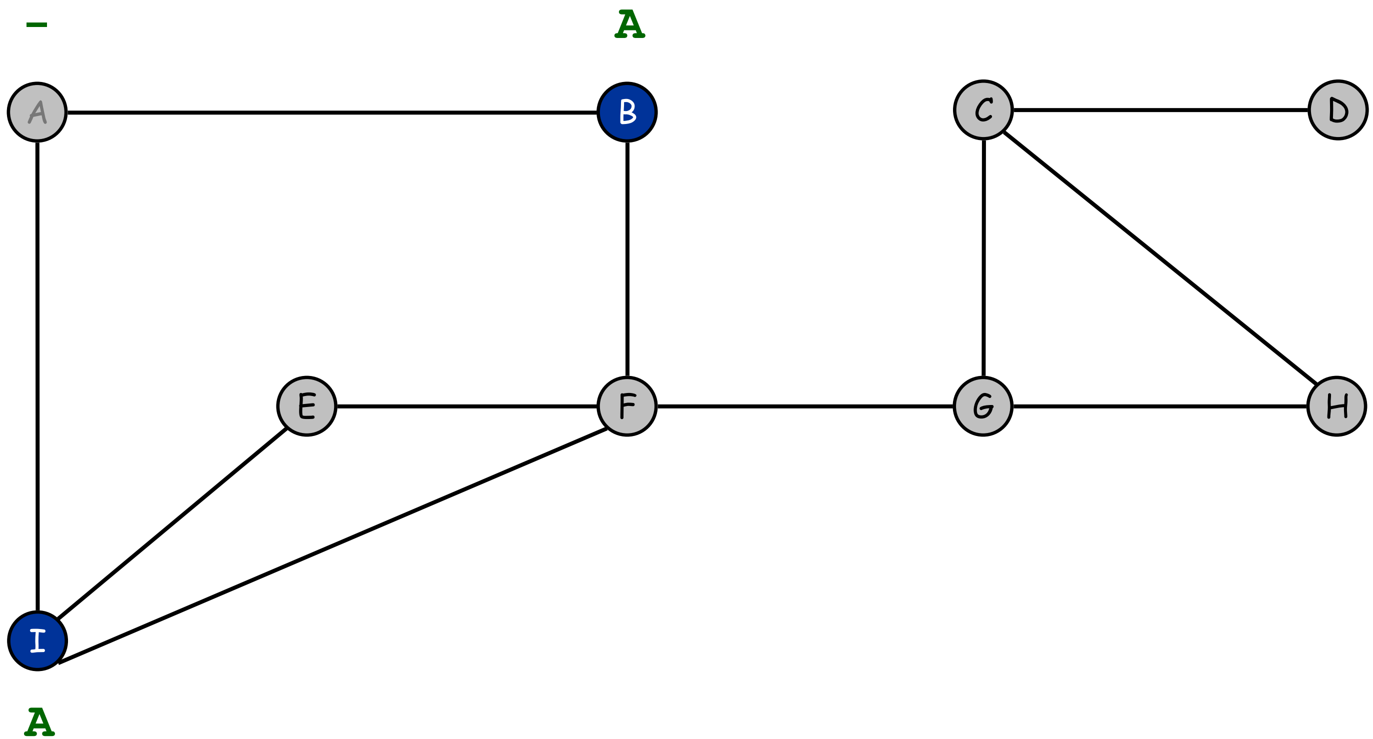


I discovered

front **B I**

FIFO Queue

Breadth First Search

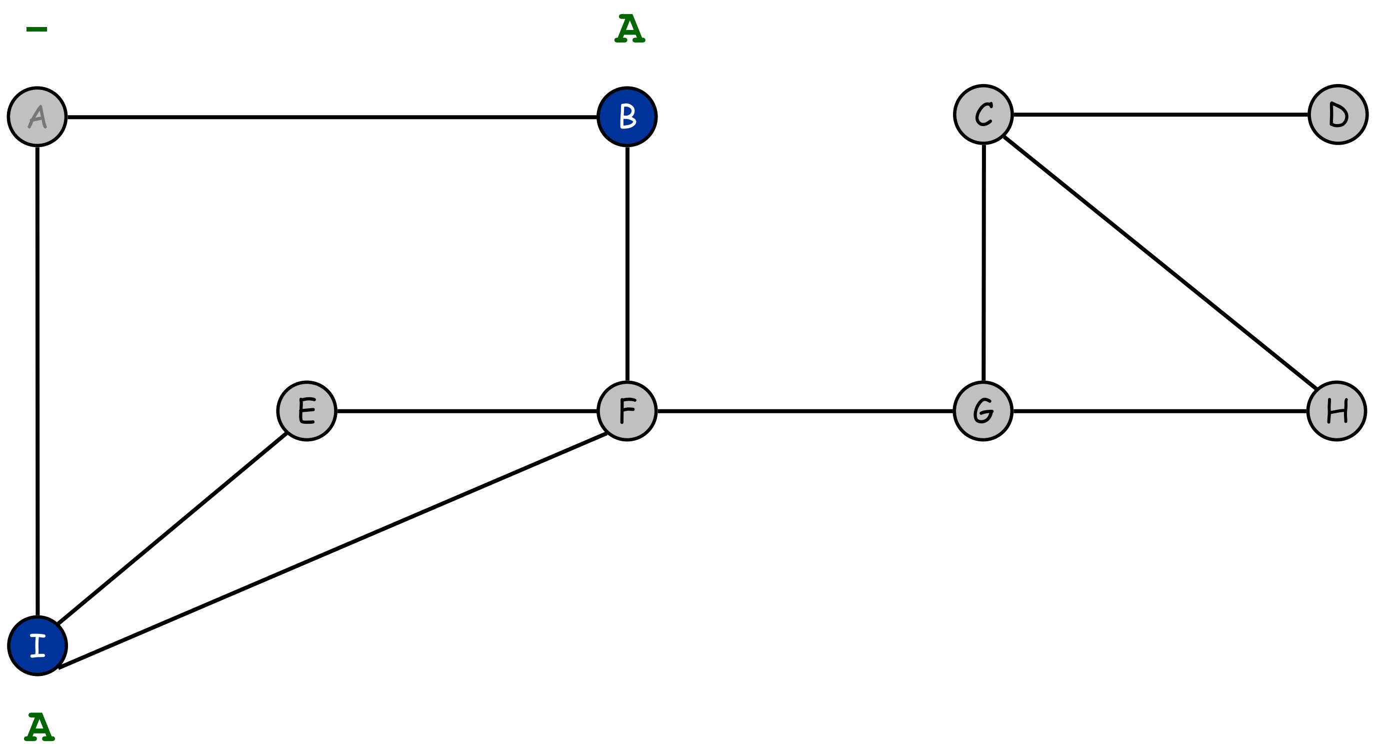


finished with A

front **B I**

FIFO Queue

Breadth First Search

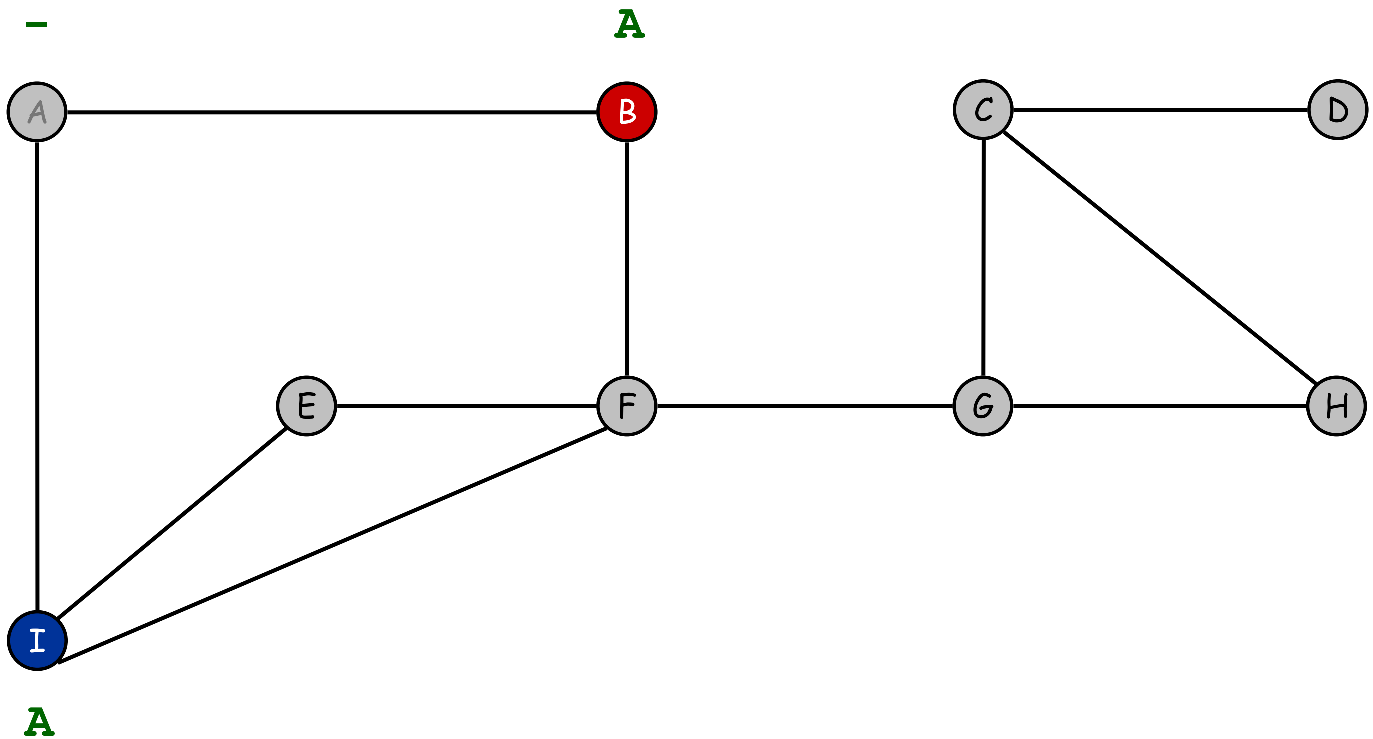


dequeue next vertex

front **B I**

FIFO Queue

Breadth First Search

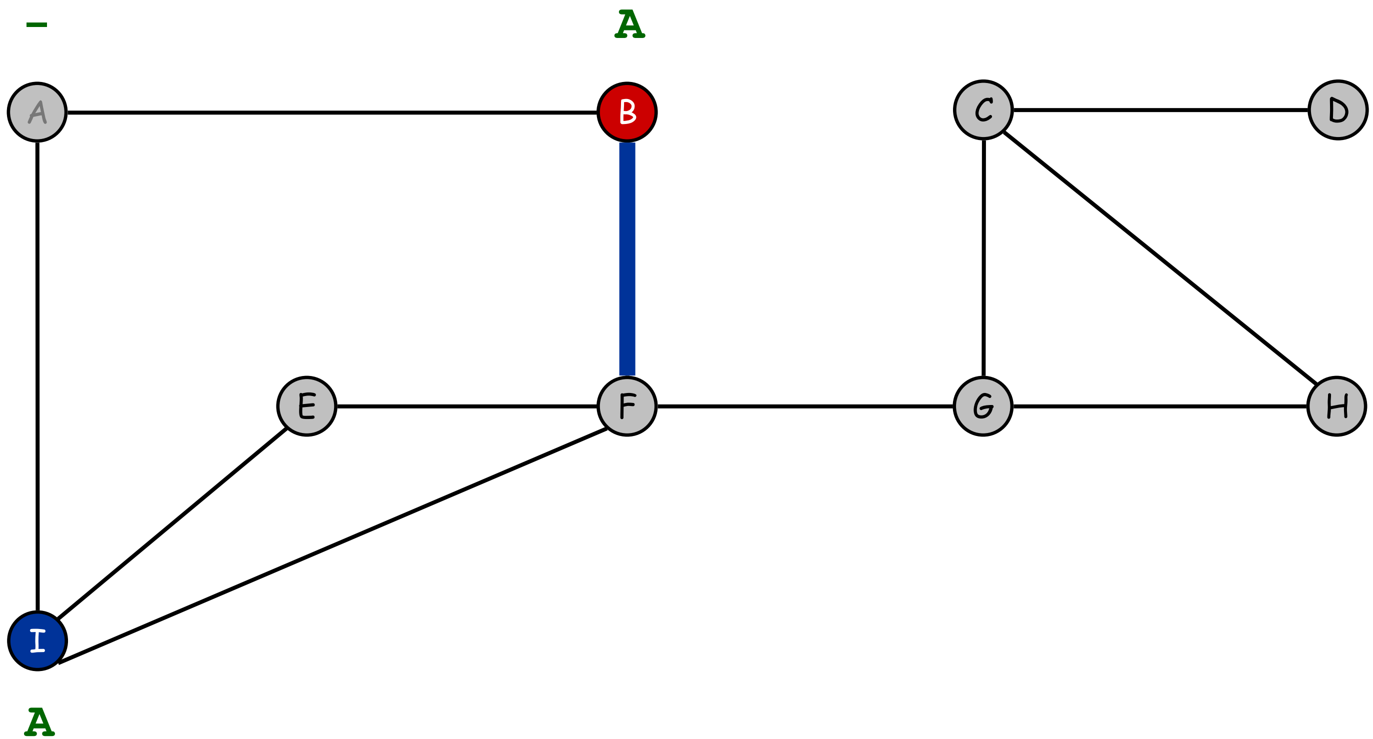


visit neighbors of B

front I

FIFO Queue

Breadth First Search

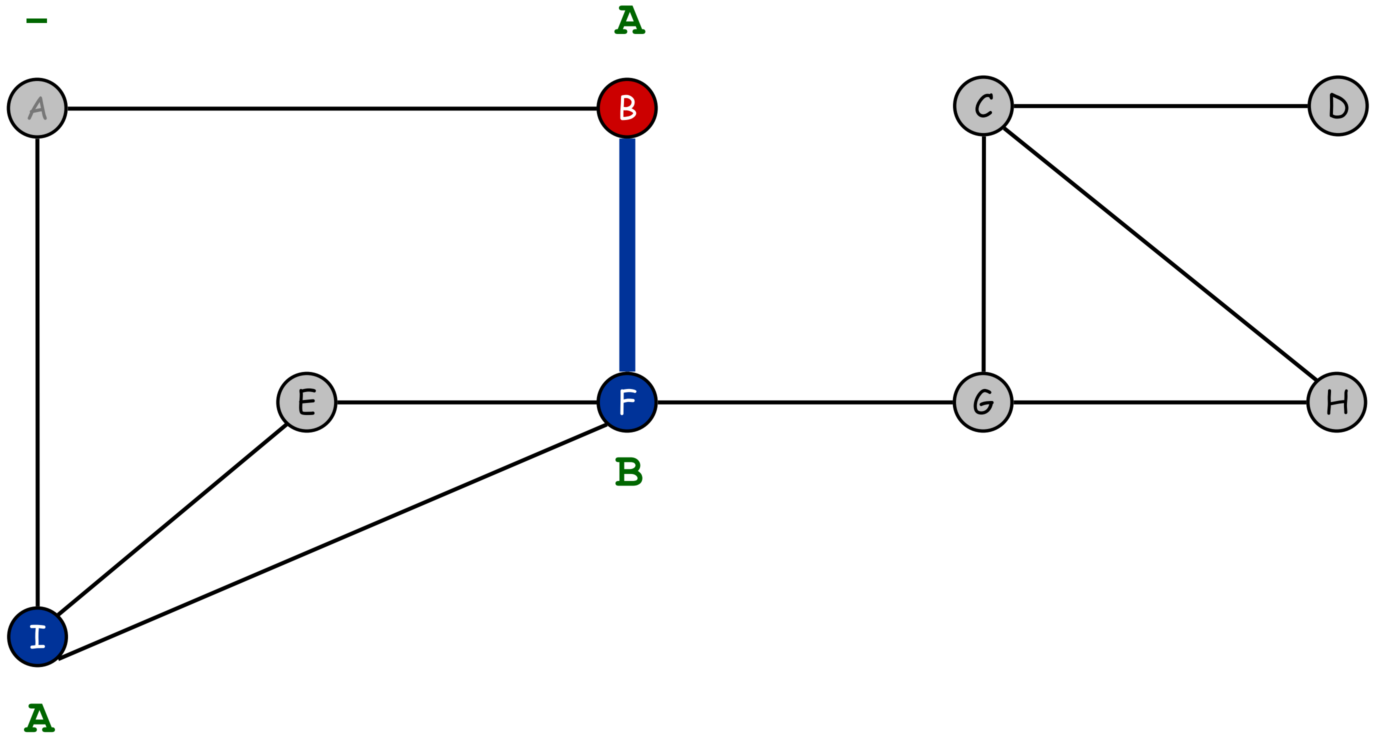


visit neighbors of B

front I

FIFO Queue

Breadth First Search

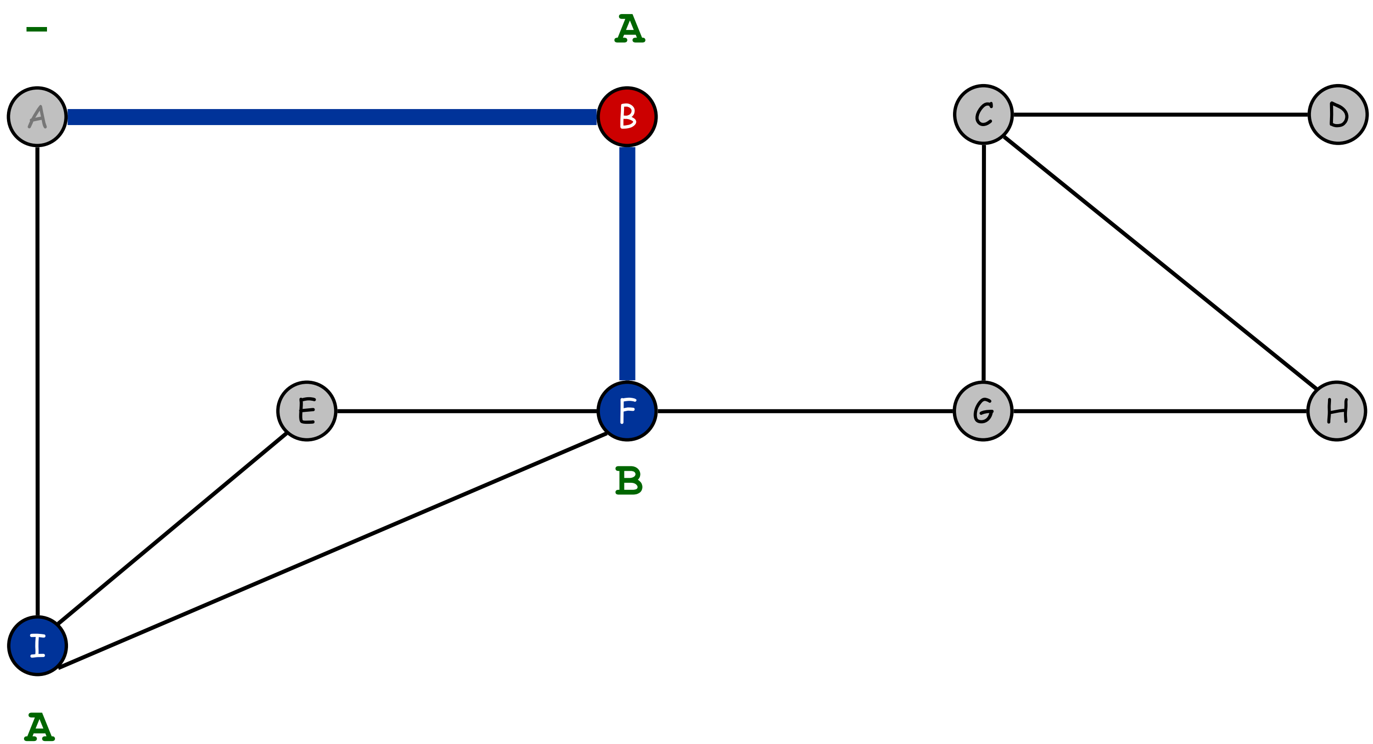


F discovered

front I F

FIFO Queue

Breadth First Search

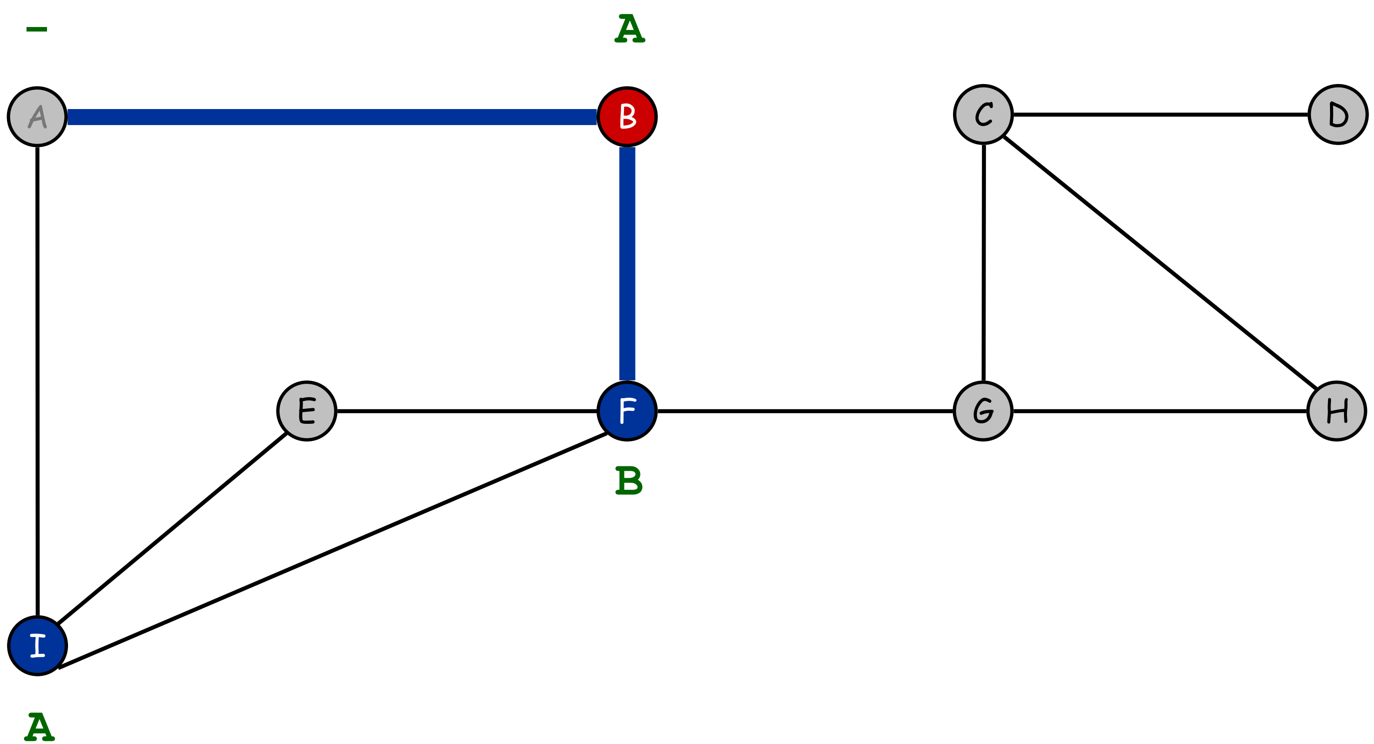


visit neighbors of B

front **I F**

FIFO Queue

Breadth First Search

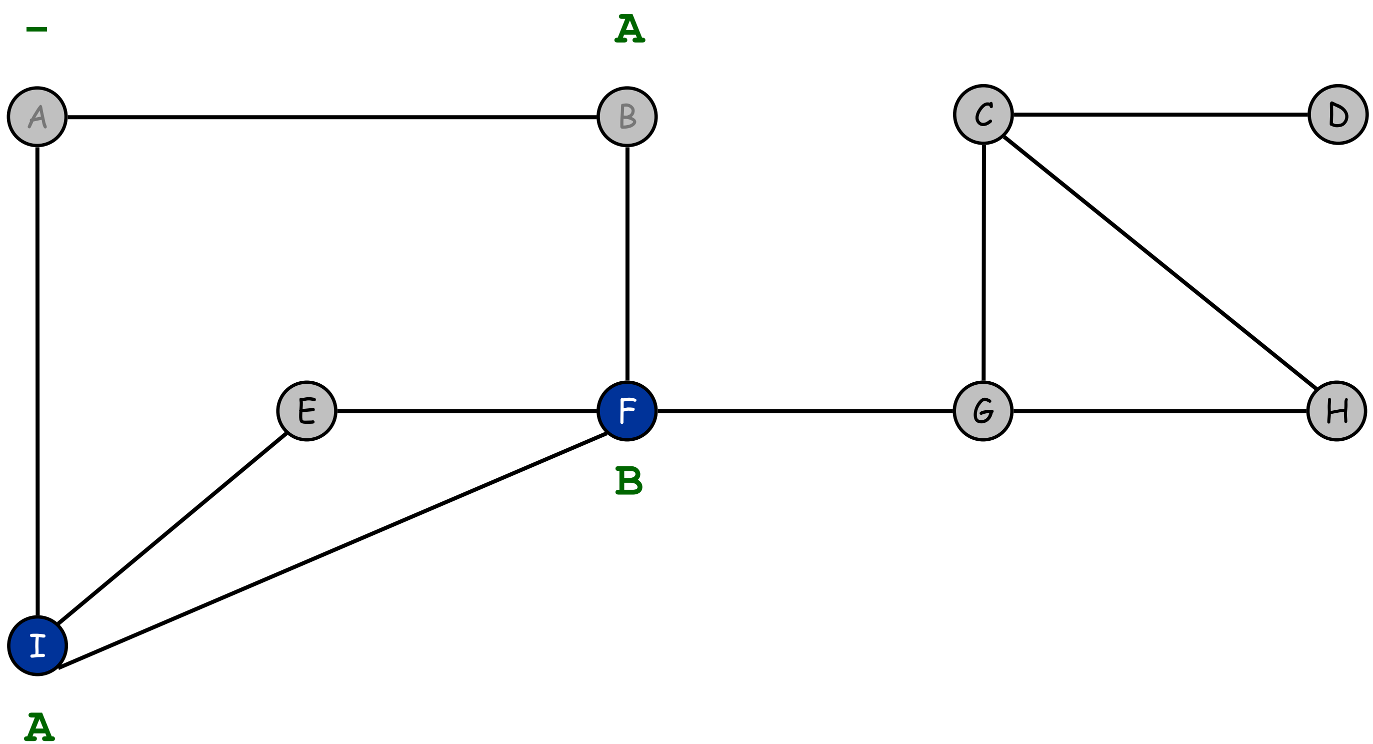


A already discovered

front **I F**

FIFO Queue

Breadth First Search

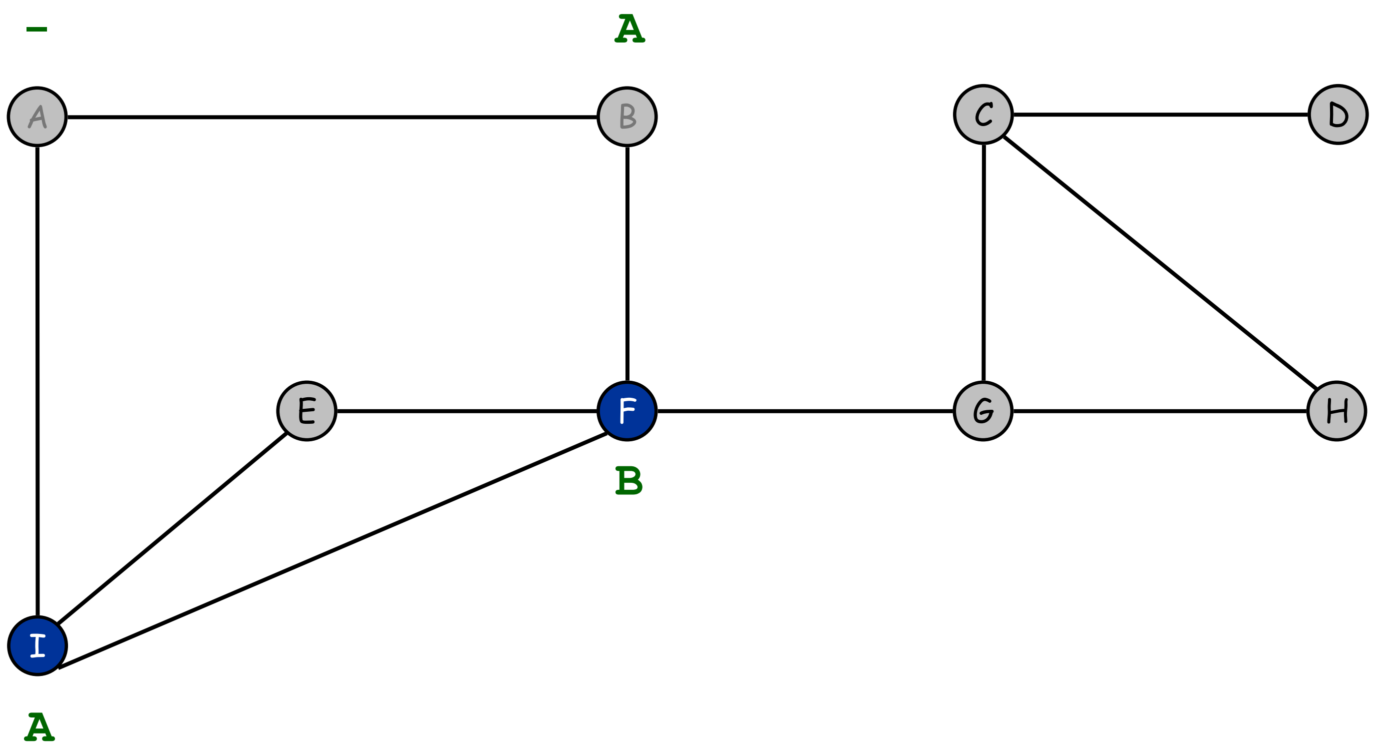


finished with B

front **I F**

FIFO Queue

Breadth First Search

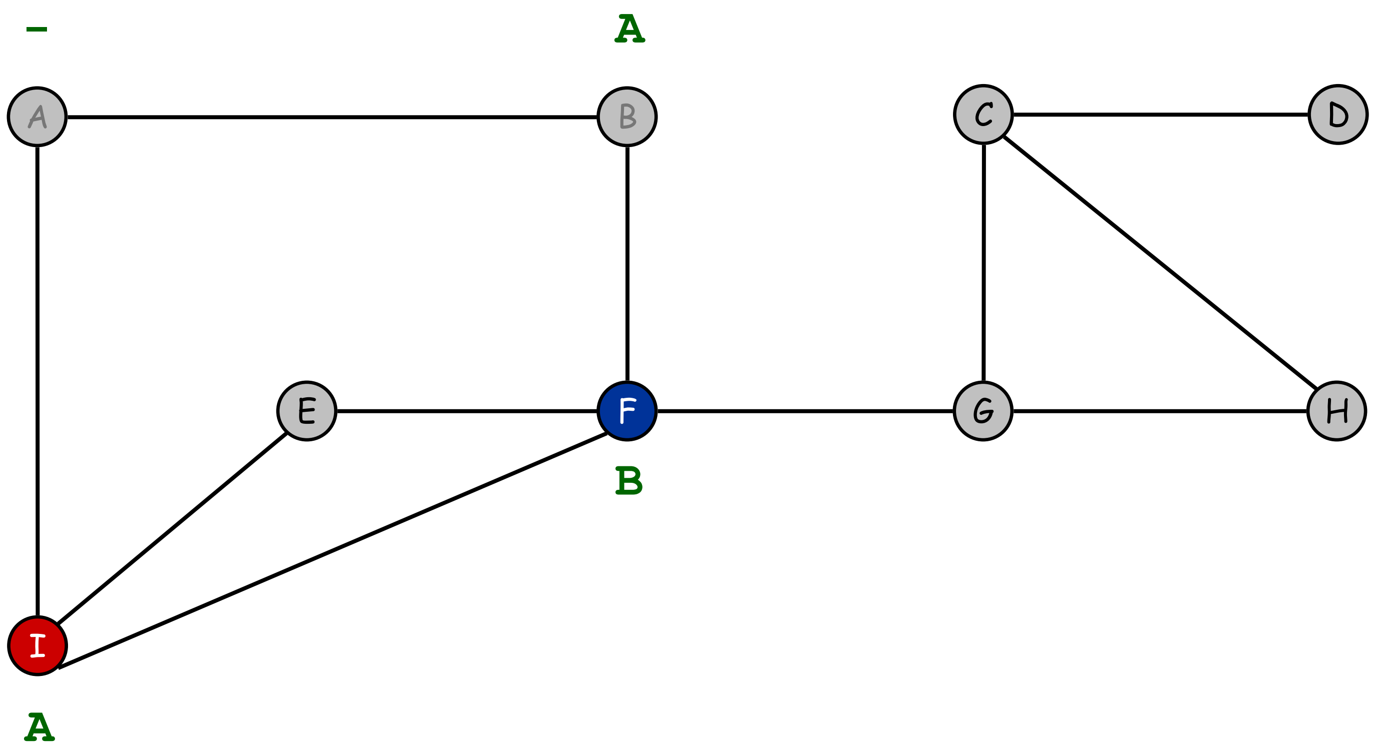


dequeue next vertex

front **I F**

FIFO Queue

Breadth First Search

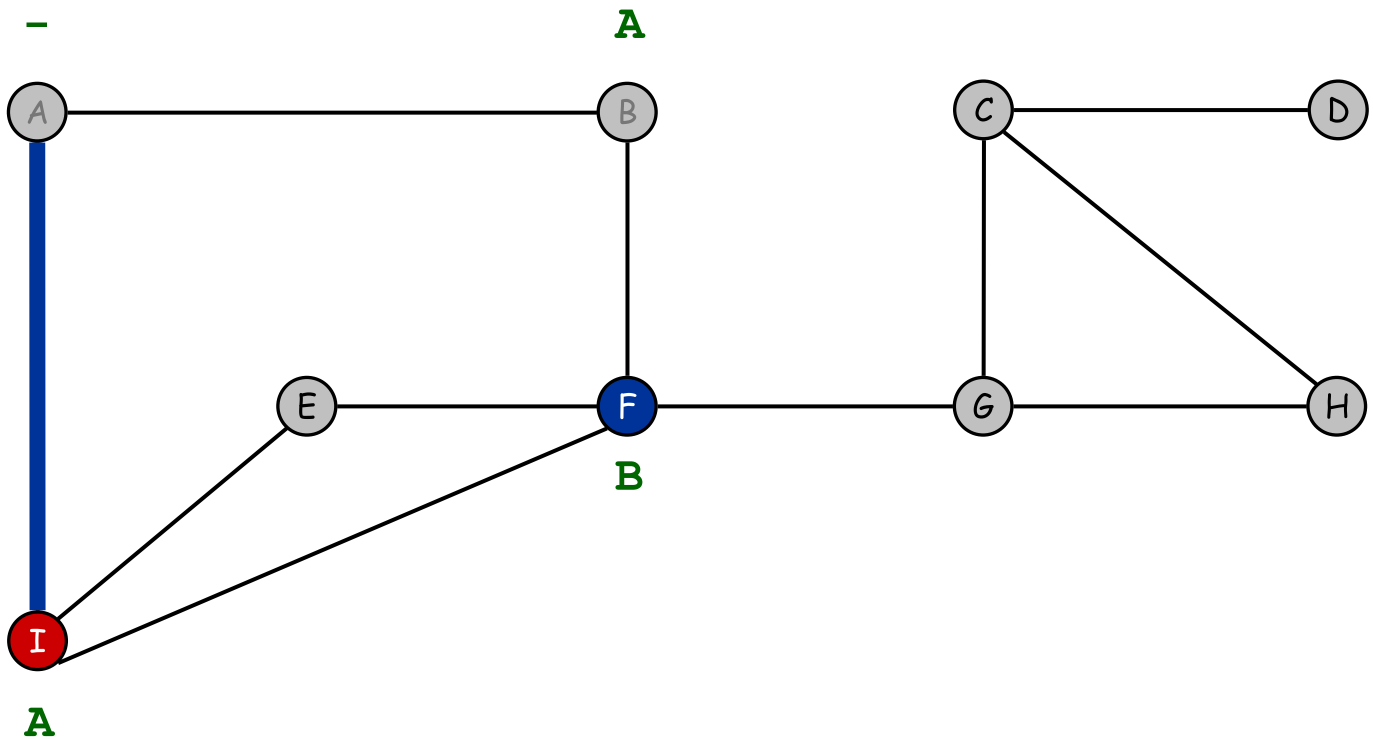


visit neighbors of I

front F

FIFO Queue

Breadth First Search

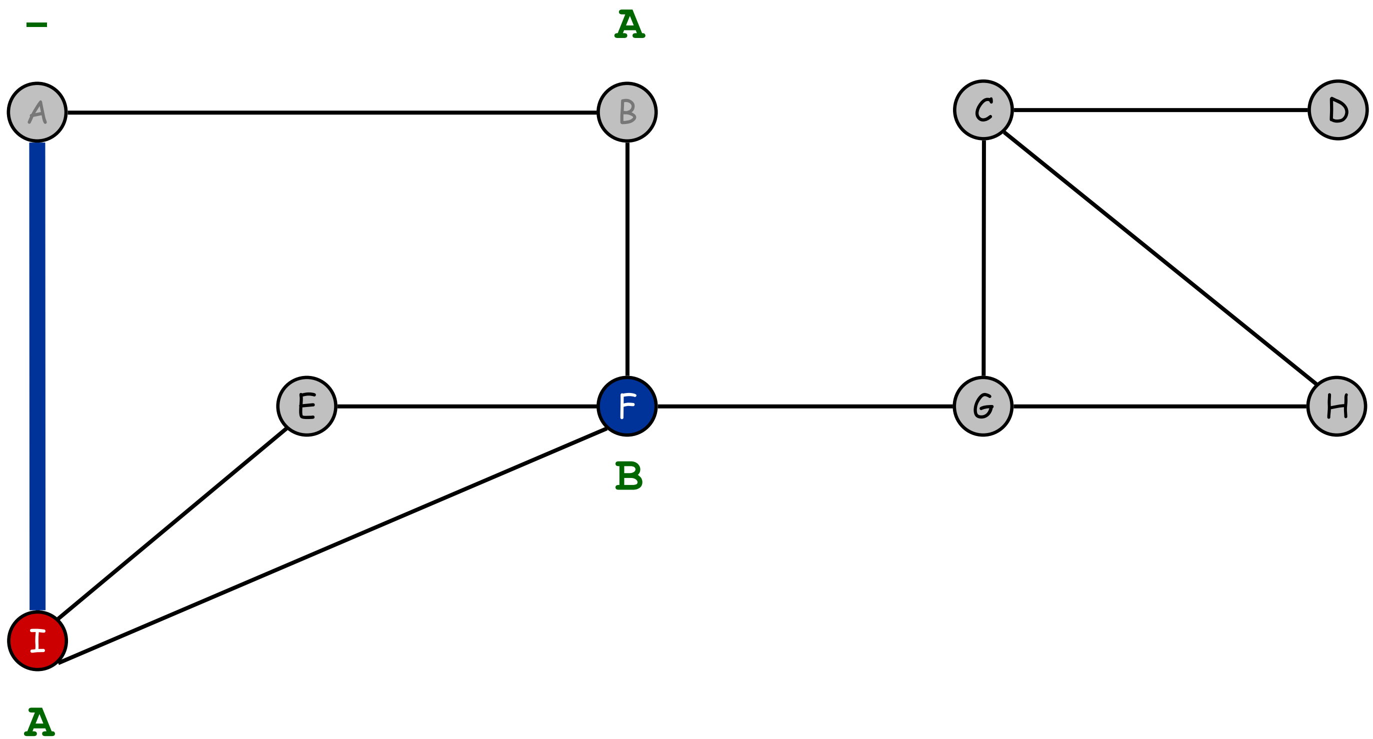


visit neighbors of I

front **F**

FIFO Queue

Breadth First Search

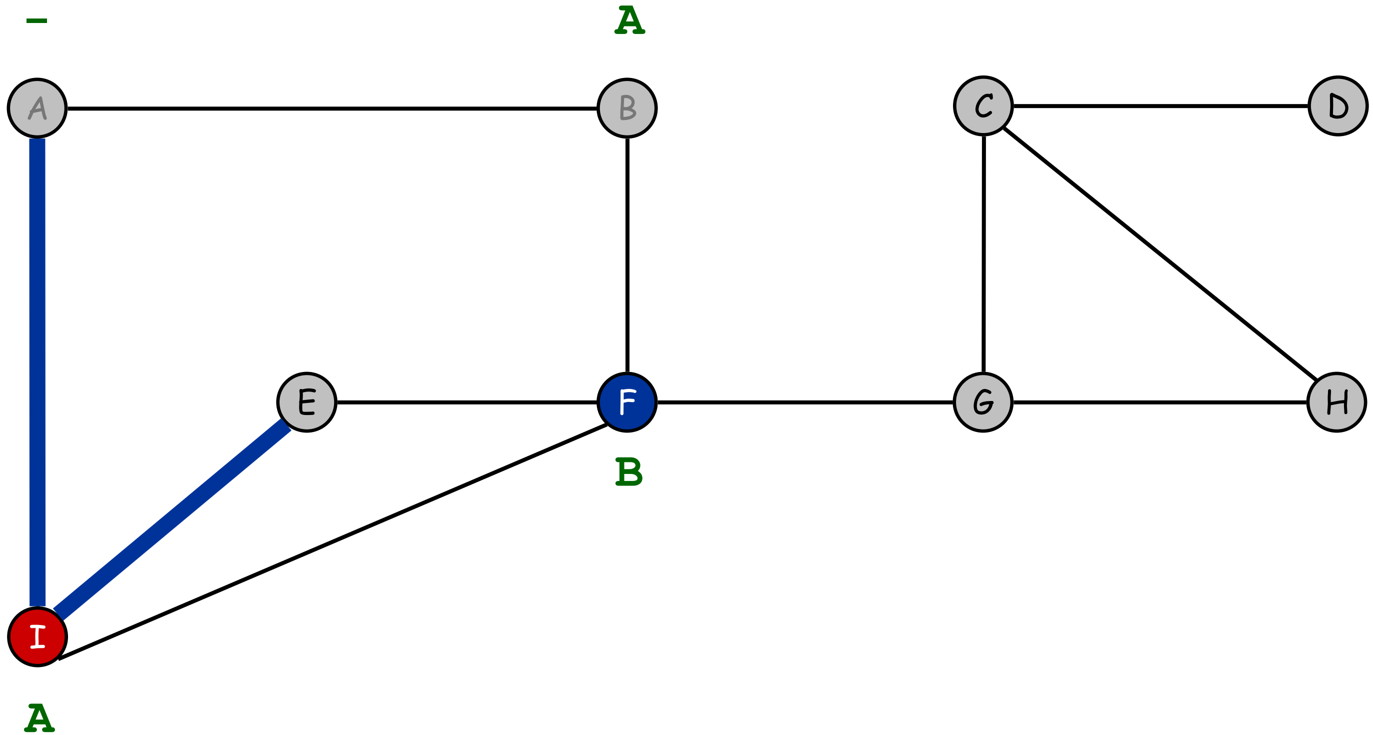


A already discovered

front **F**

FIFO Queue

Breadth First Search

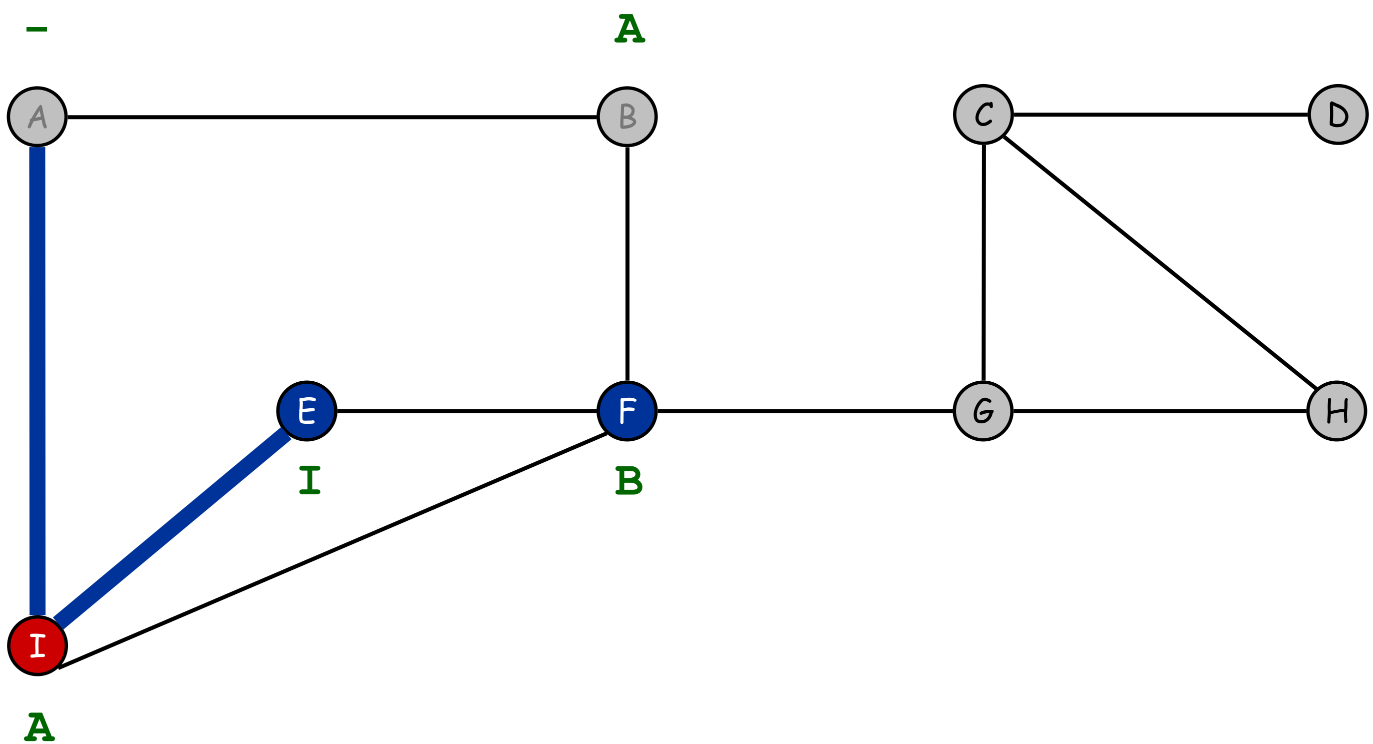


visit neighbors of I

front F

FIFO Queue

Breadth First Search

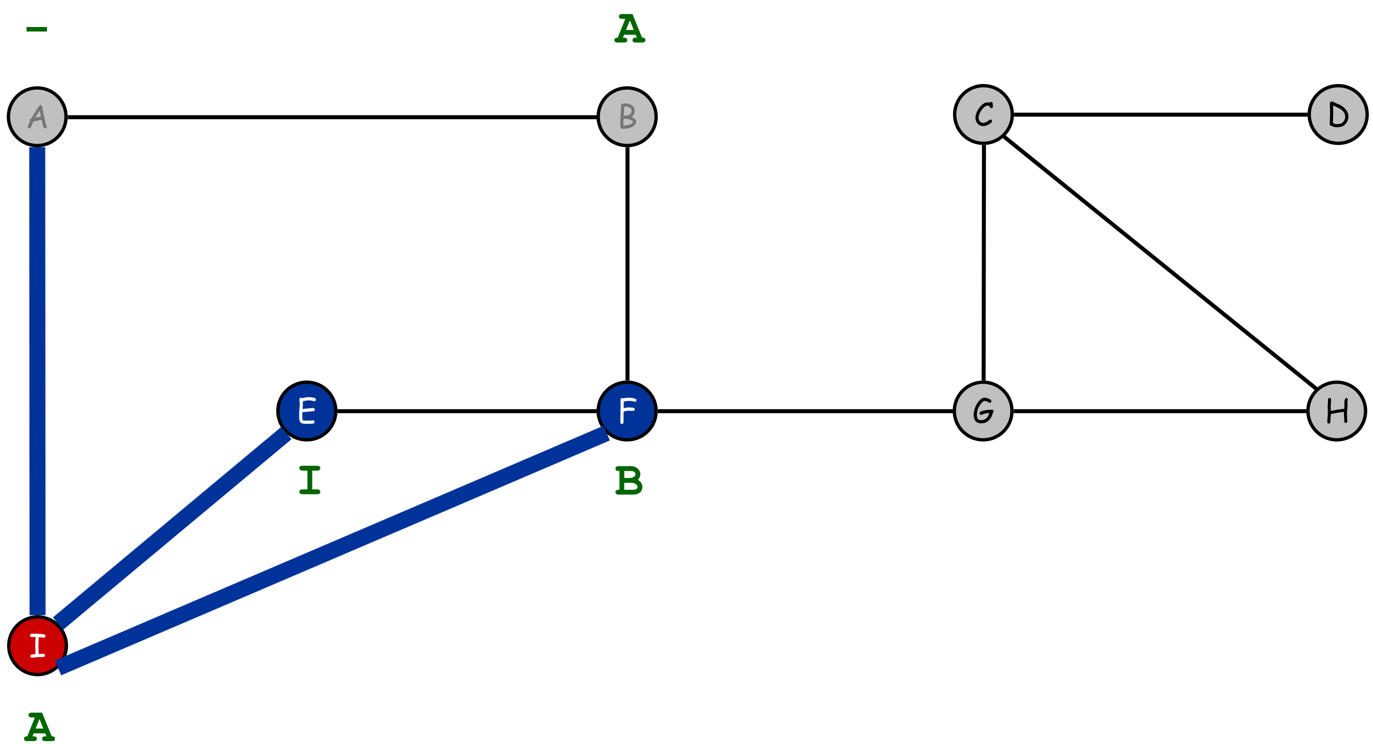


E discovered

front **F E**

FIFO Queue

Breadth First Search



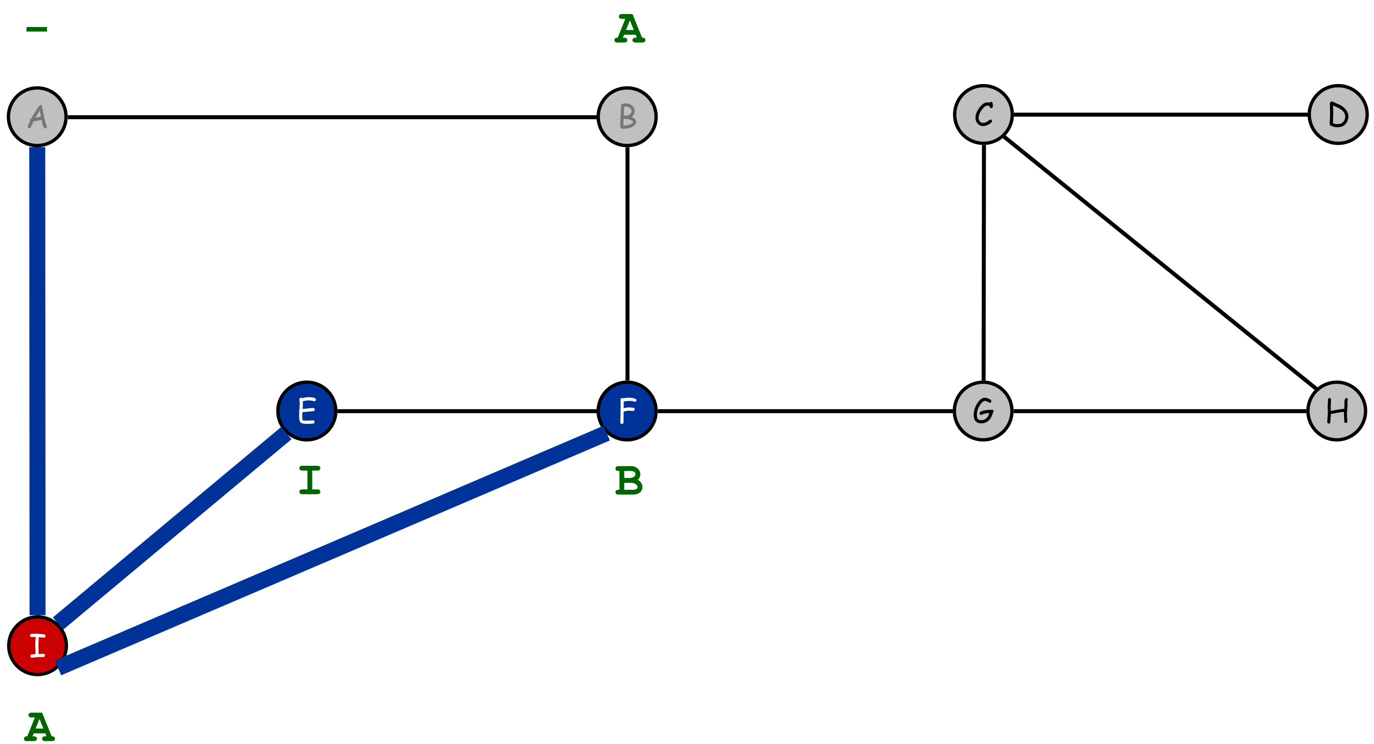
visit neighbors of I

front

F E

FIFO Queue

Breadth First Search

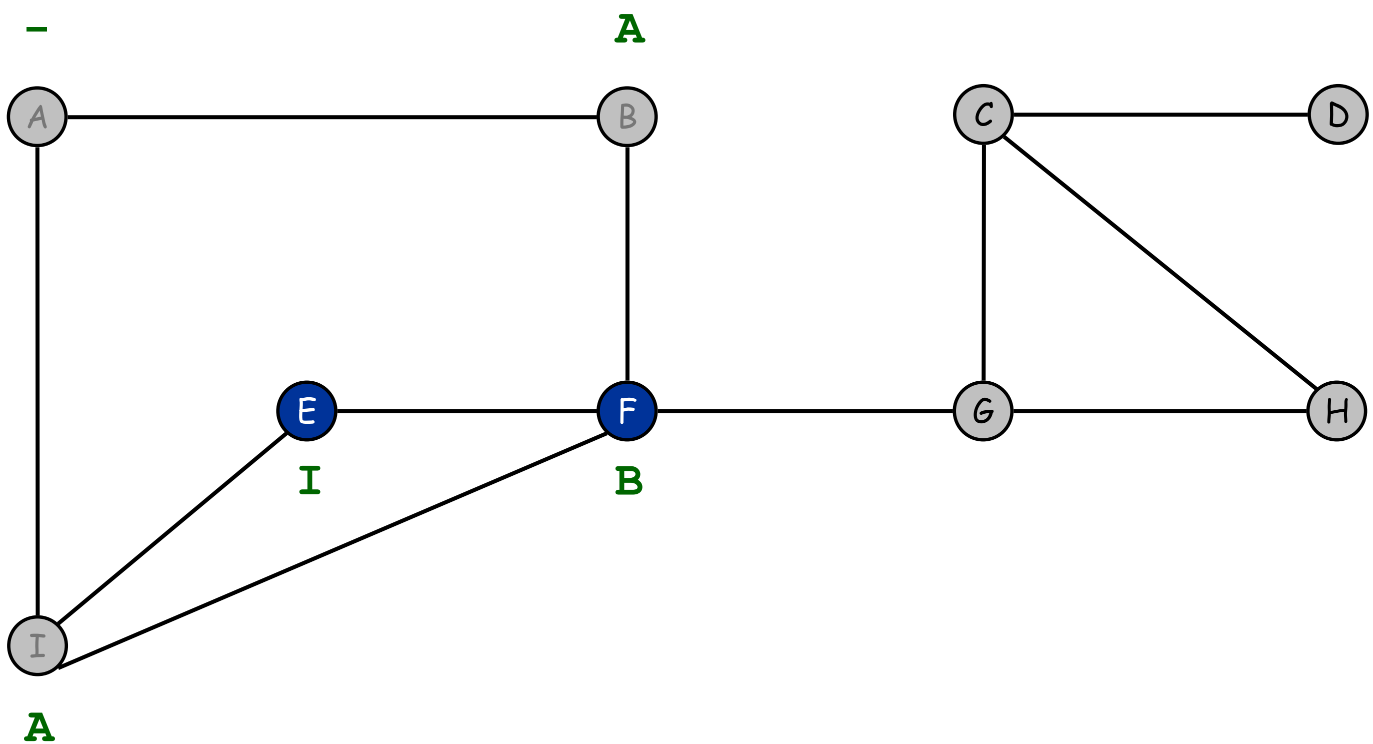


F already discovered

front **F E**

FIFO Queue

Breadth First Search

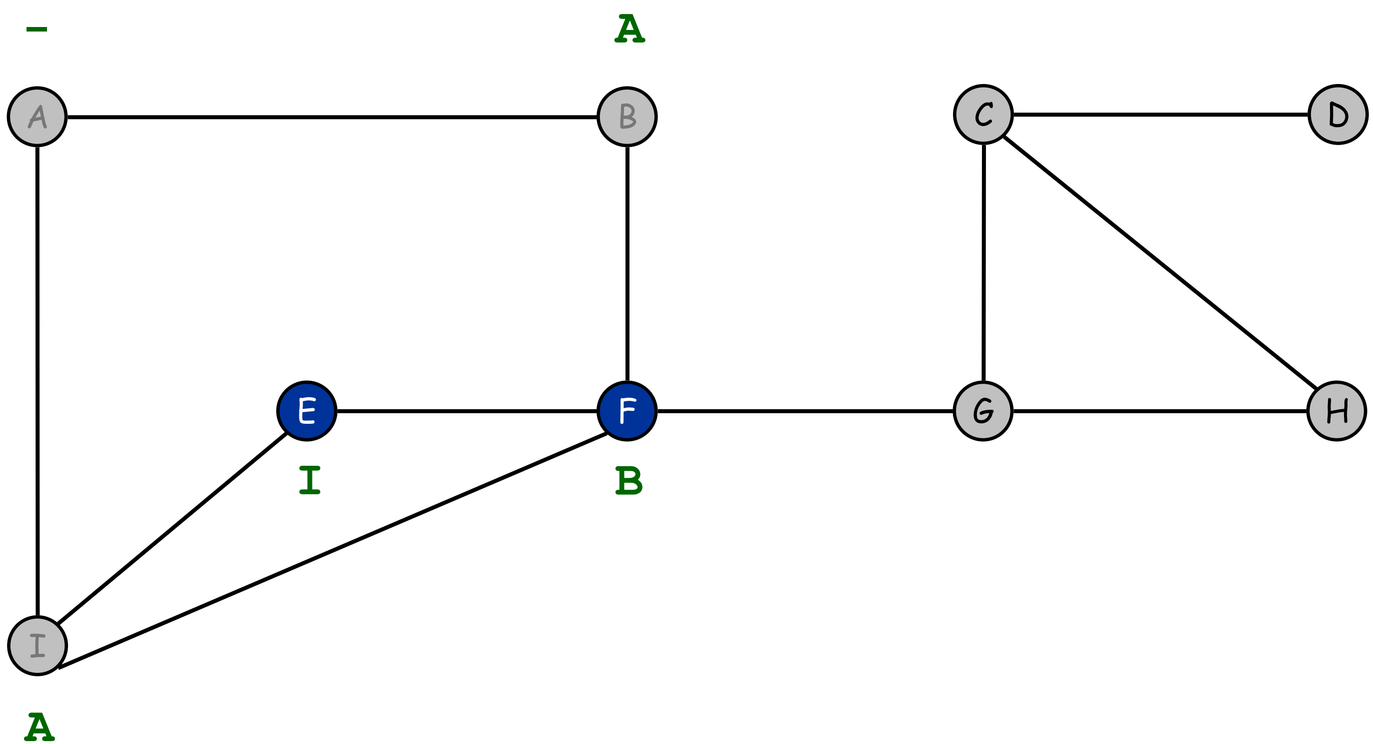


I finished

front **F E**

FIFO Queue

Breadth First Search

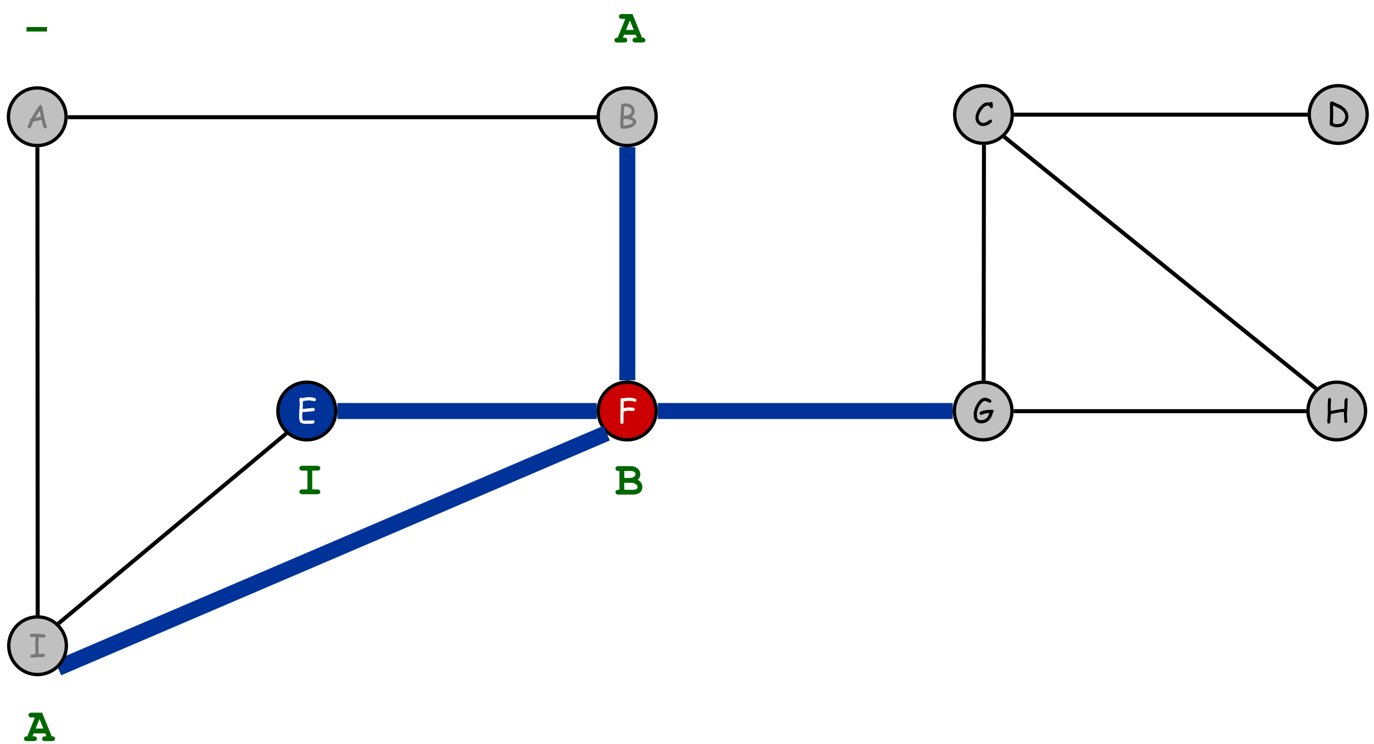


dequeue next vertex

front **F E**

FIFO Queue

Breadth First Search

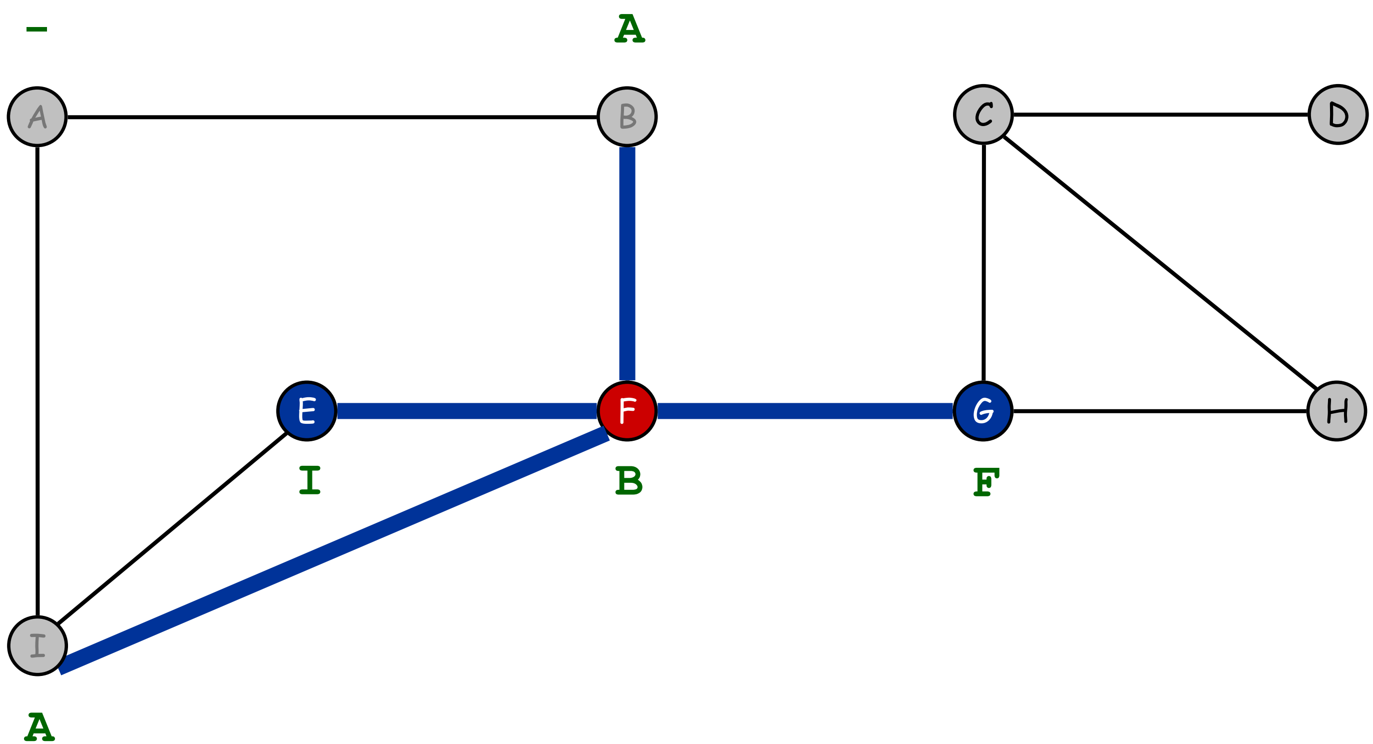


visit neighbors of F

front E

FIFO Queue

Breadth First Search

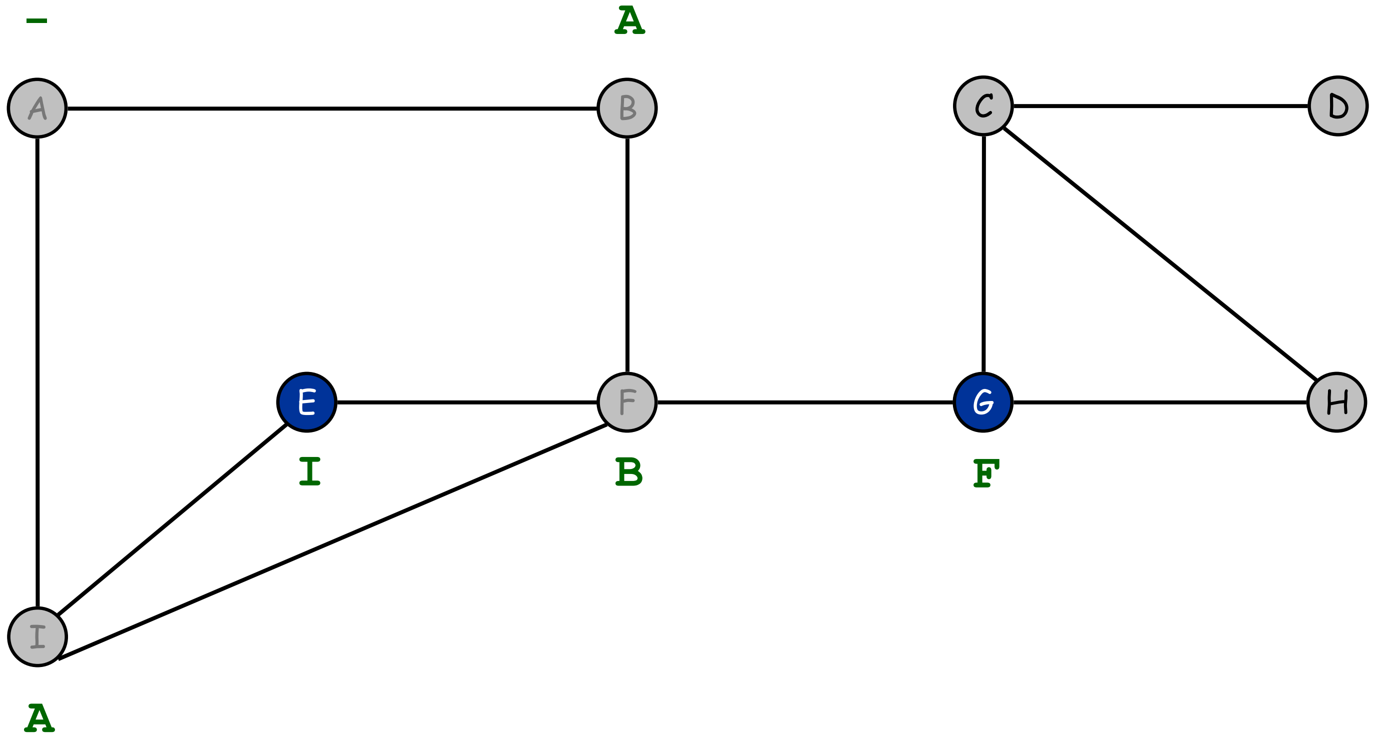


G discovered

front E G

FIFO Queue

Breadth First Search

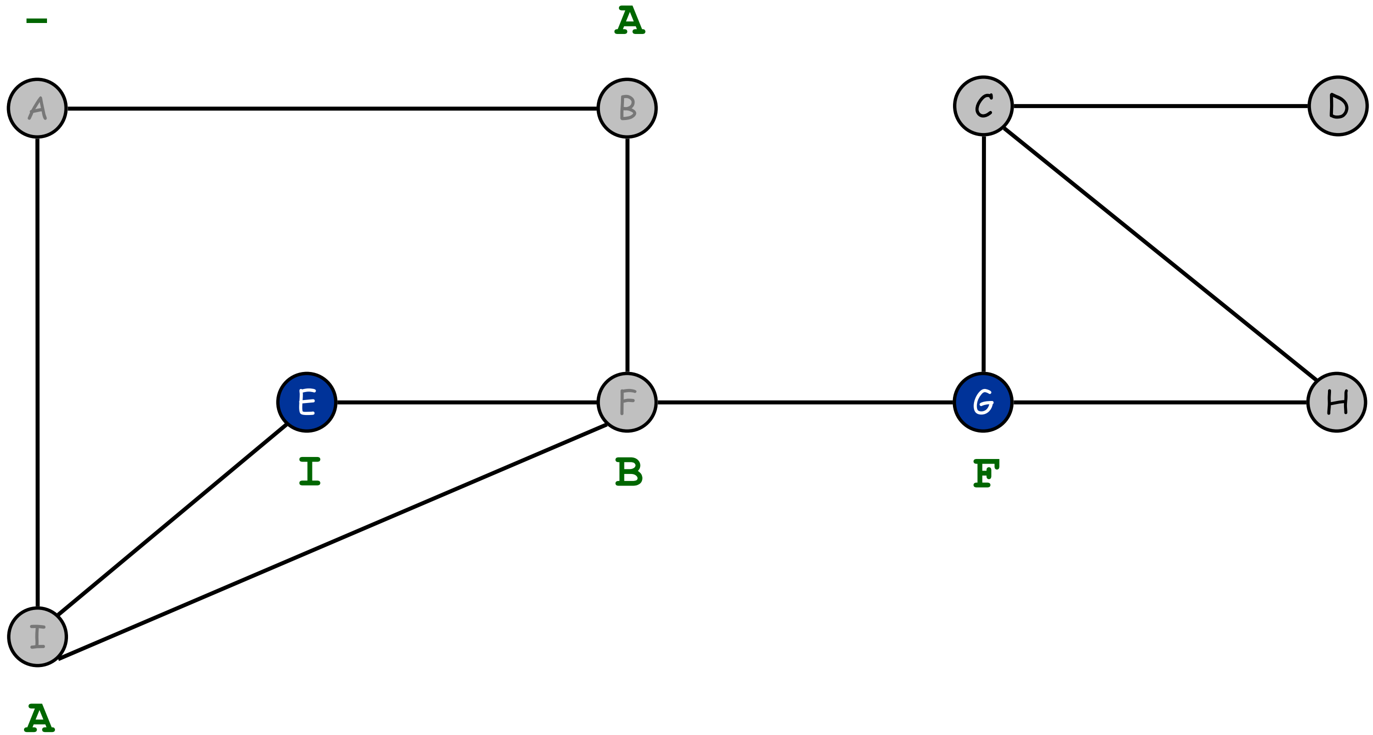


F finished

front **E G**

FIFO Queue

Breadth First Search

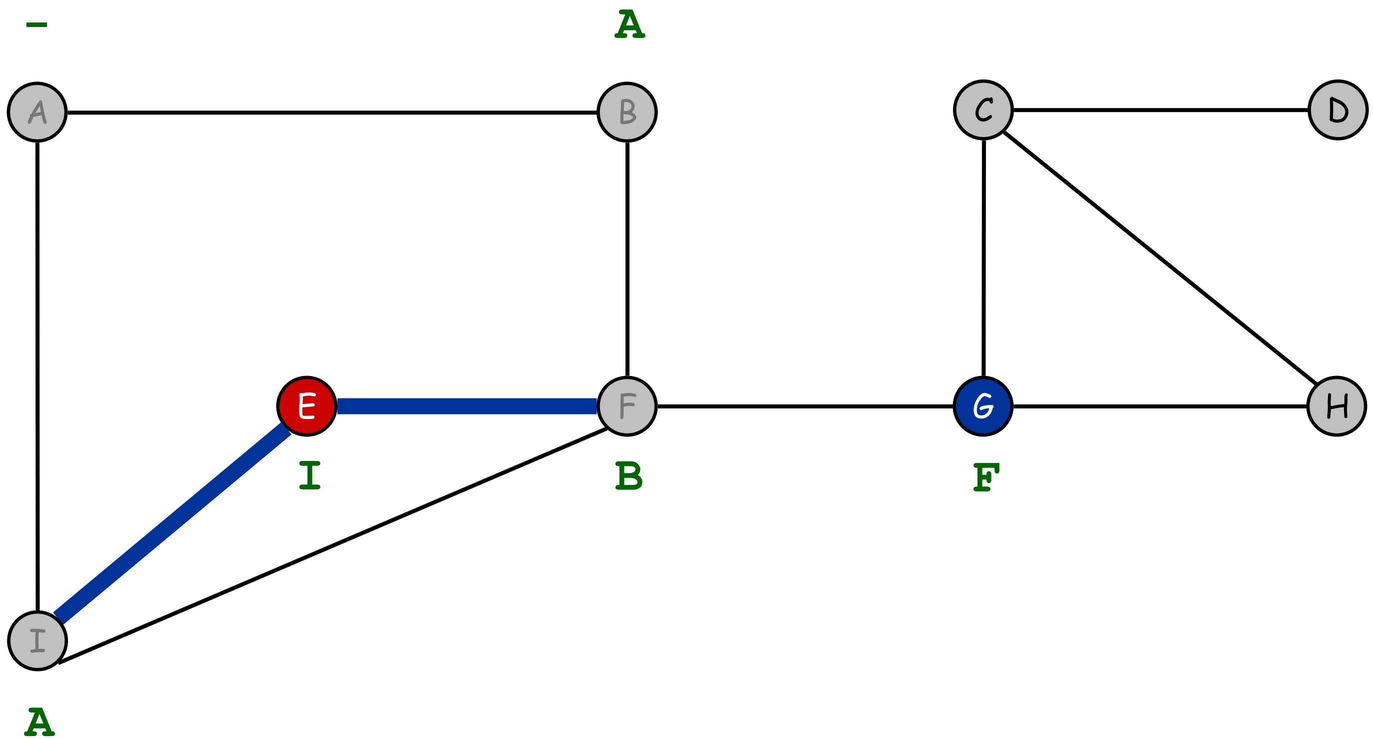


dequeue next vertex

front **E G**

FIFO Queue

Breadth First Search

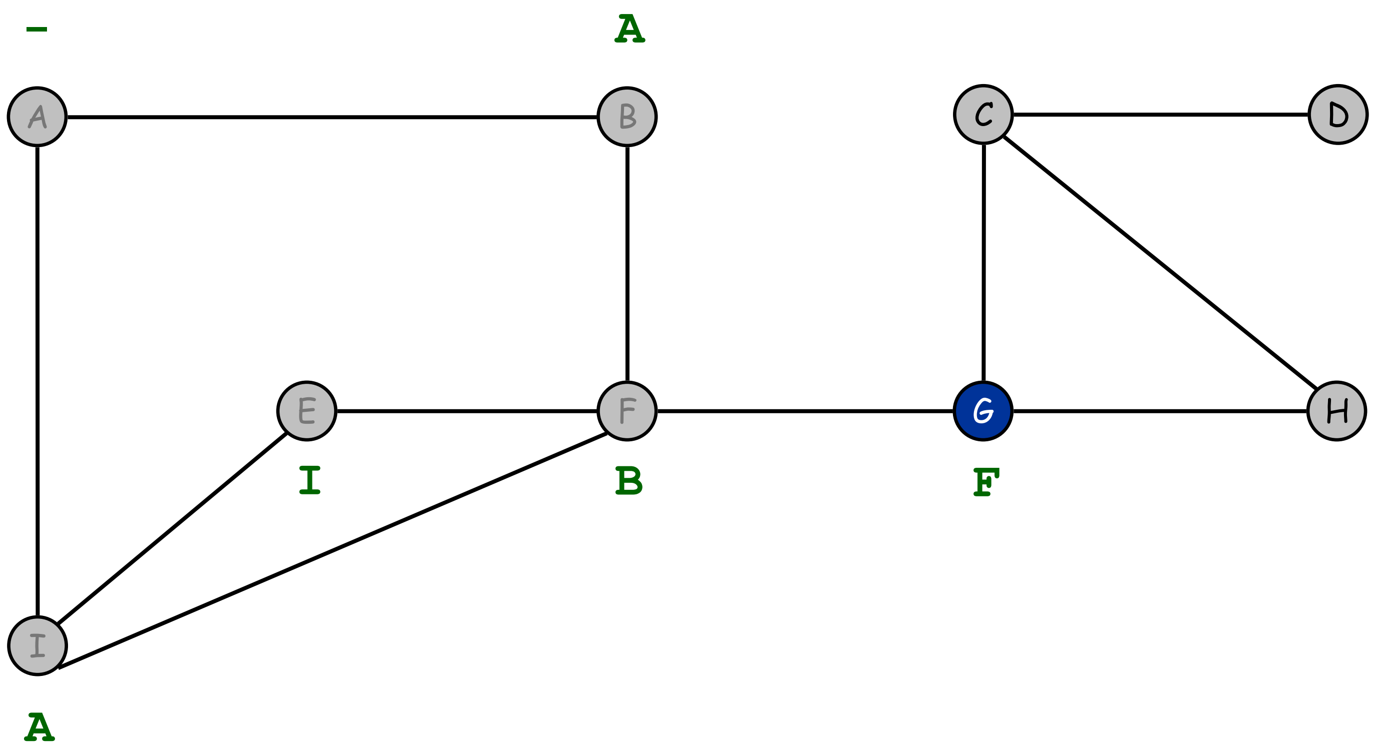


visit neighbors of E

front **G**

FIFO Queue

Breadth First Search

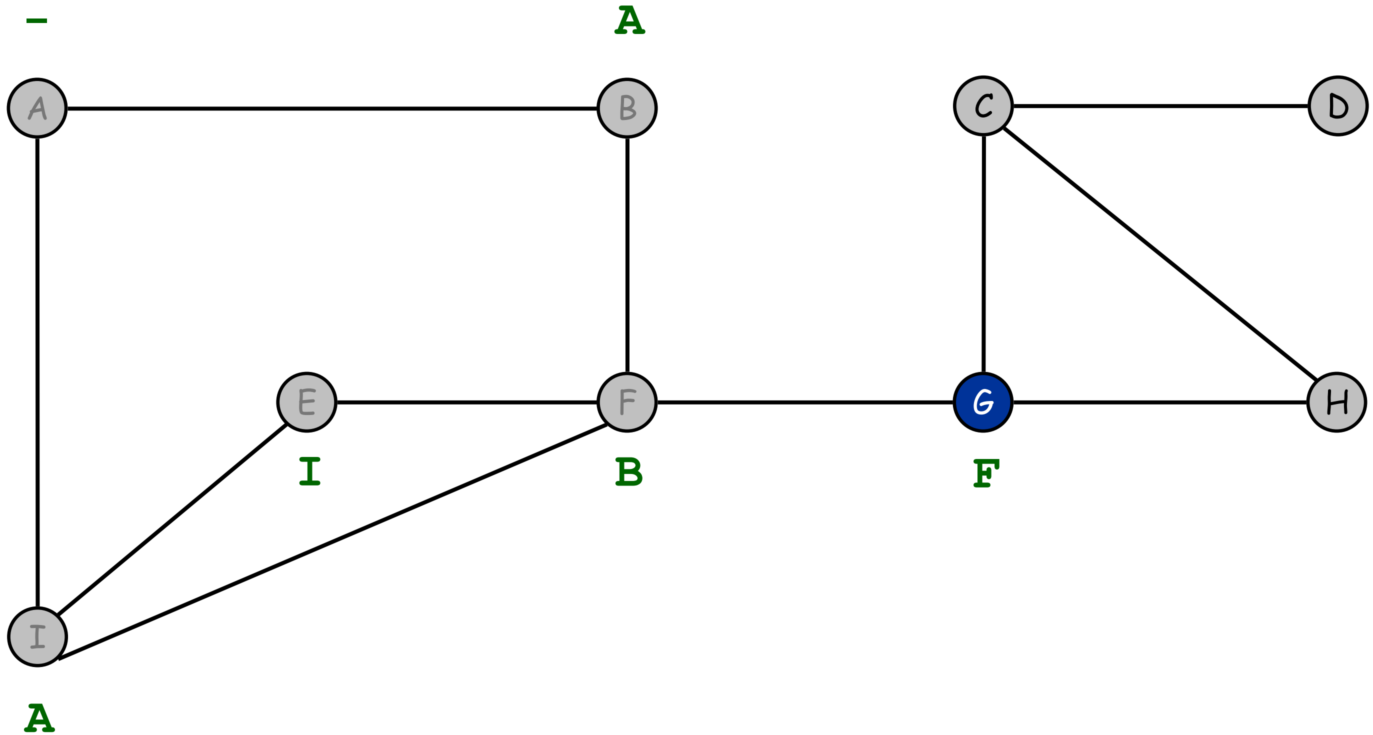


E finished

front G

FIFO Queue

Breadth First Search

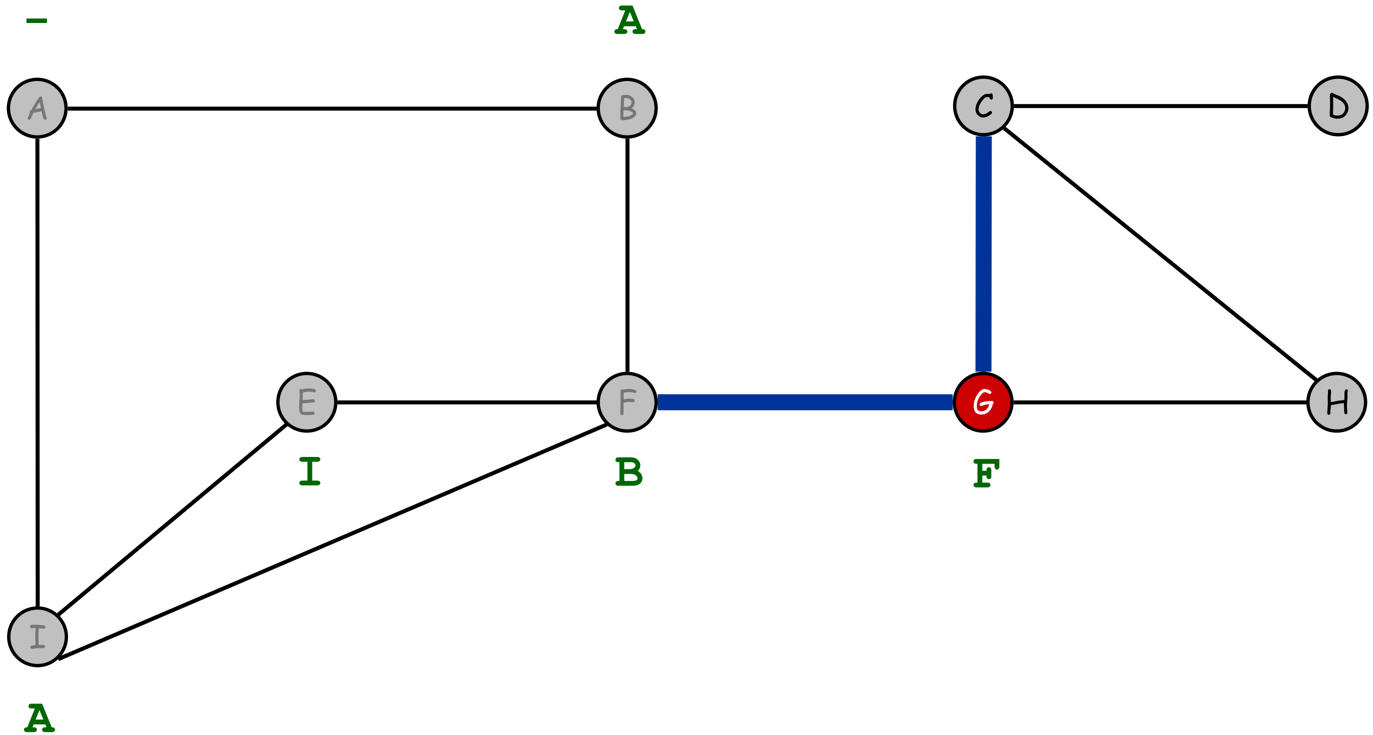


dequeue next vertex

front G

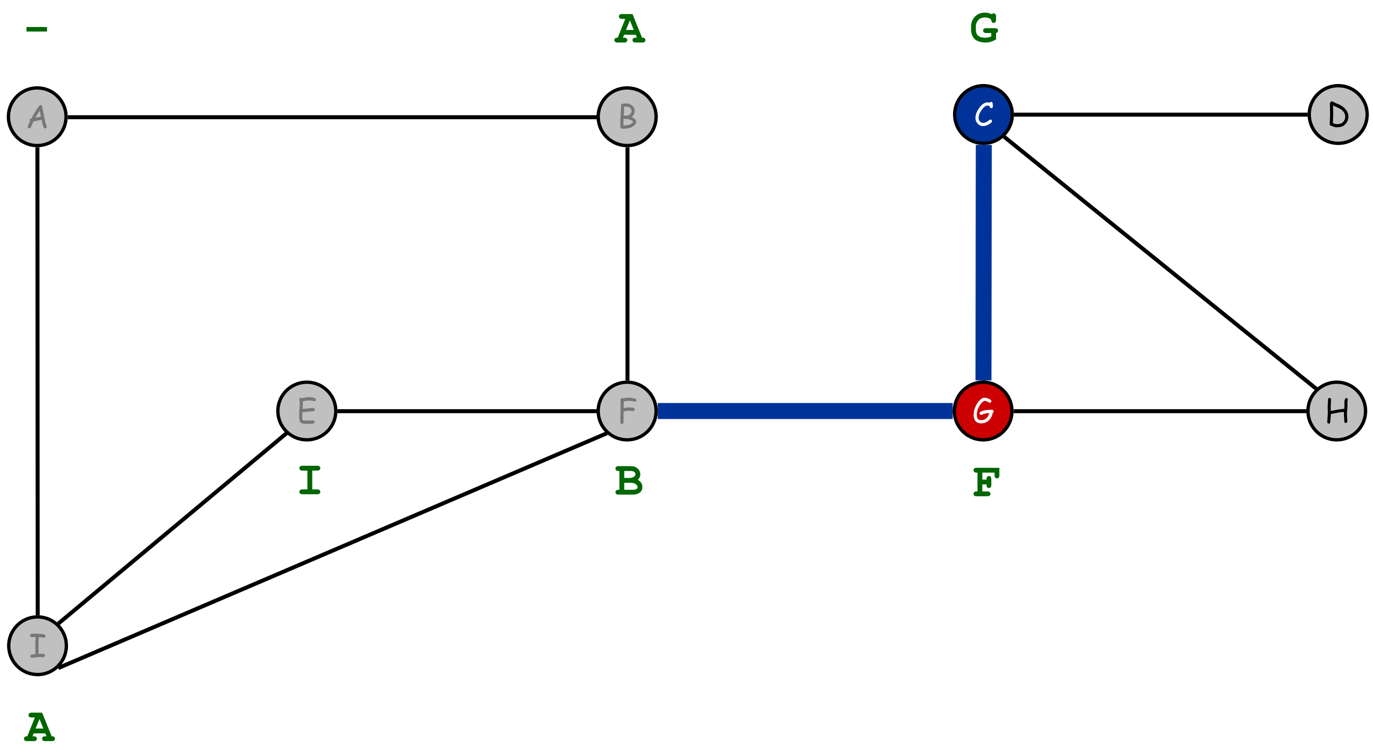
FIFO Queue

Breadth First Search



visit neighbors of G

Breadth First Search

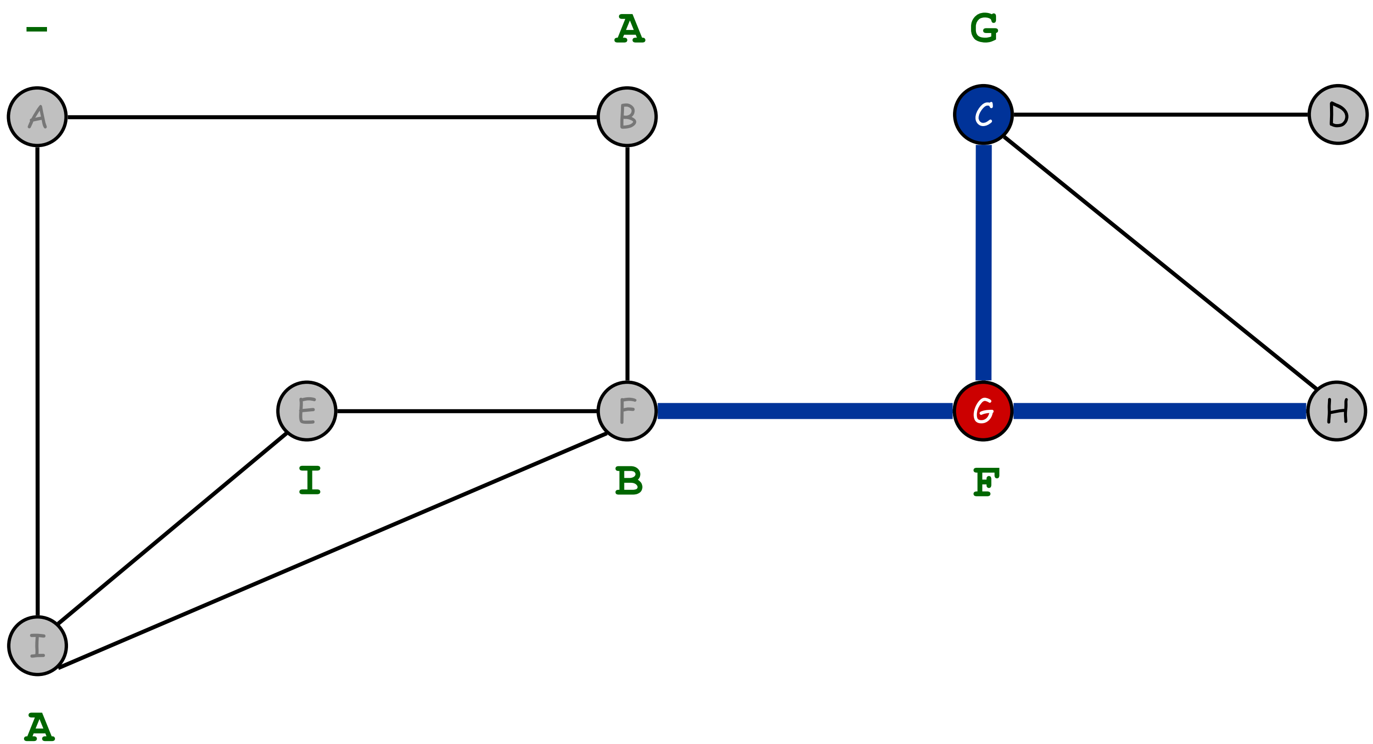


C discovered

front C

FIFO Queue

Breadth First Search



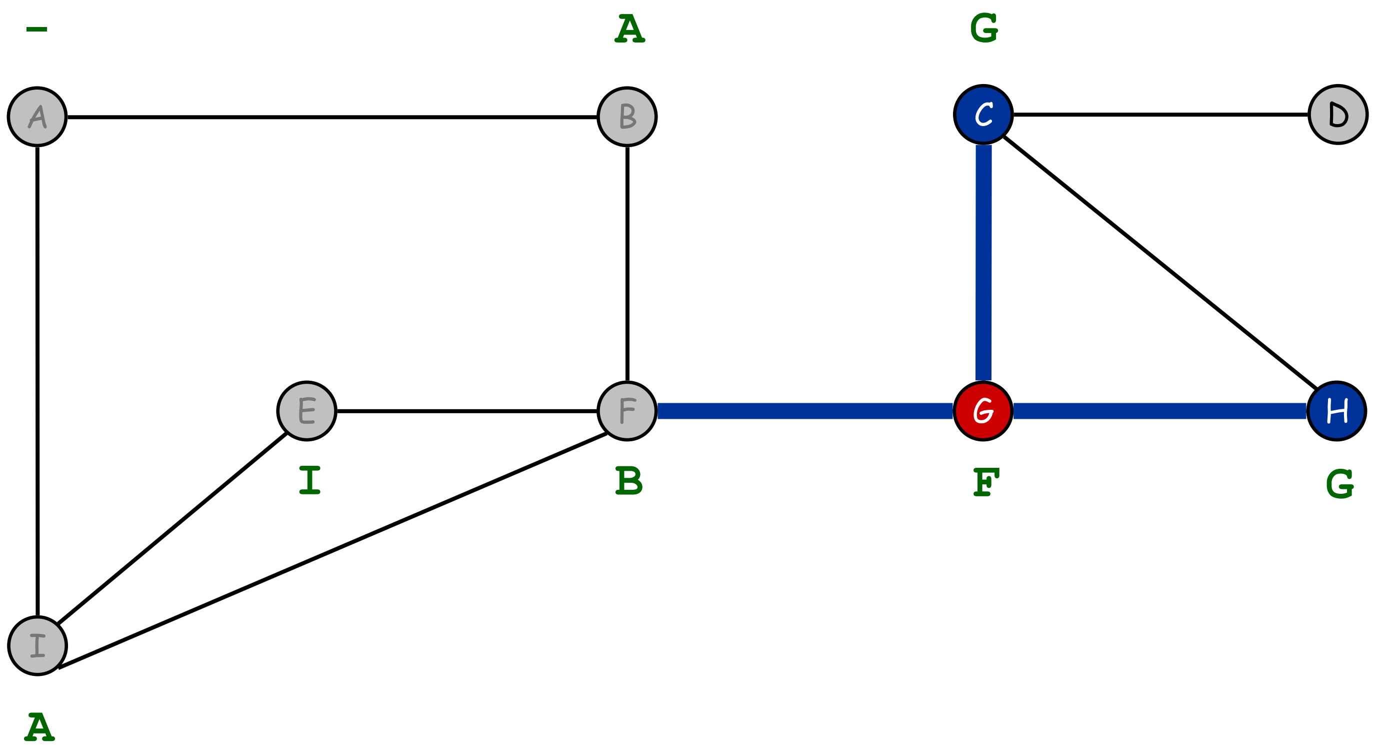
visit neighbors of G

front

C

FIFO Queue

Breadth First Search

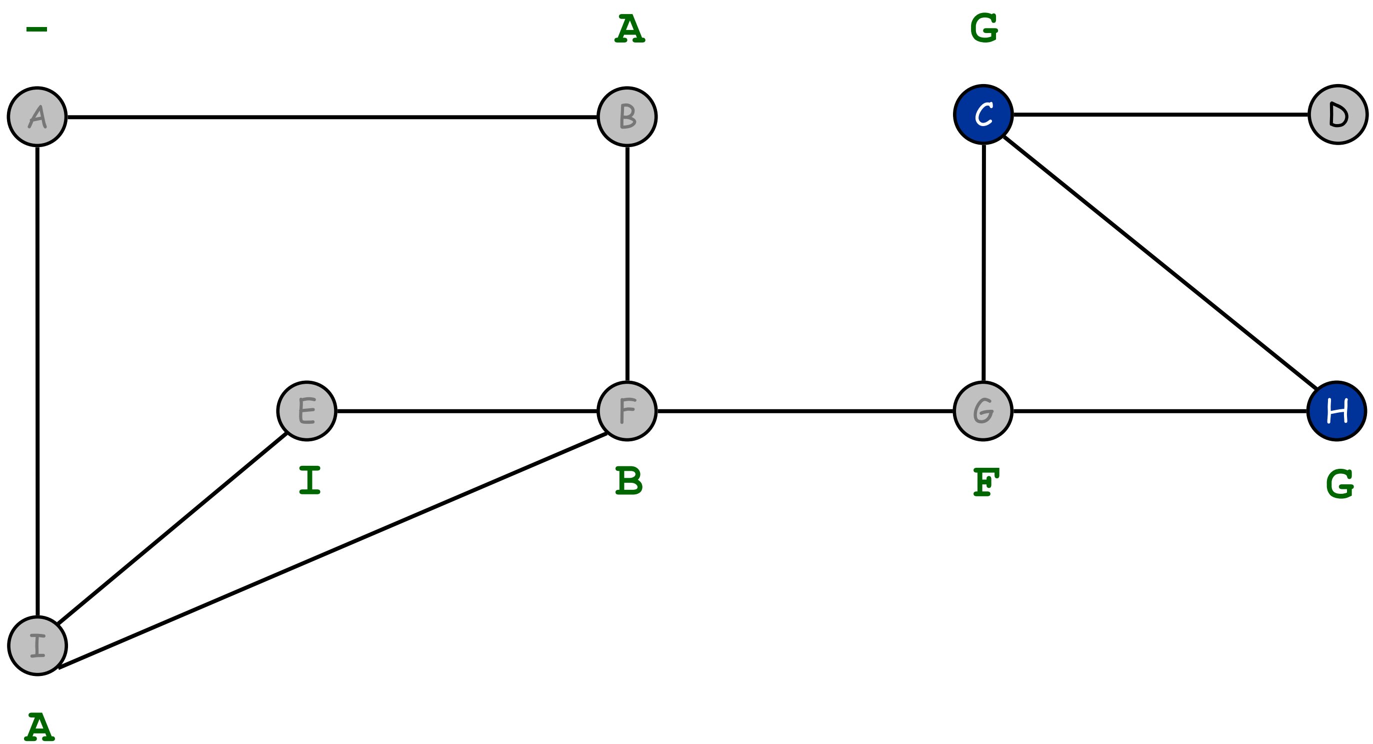


H discovered

front | C H

FIFO Queue

Breadth First Search



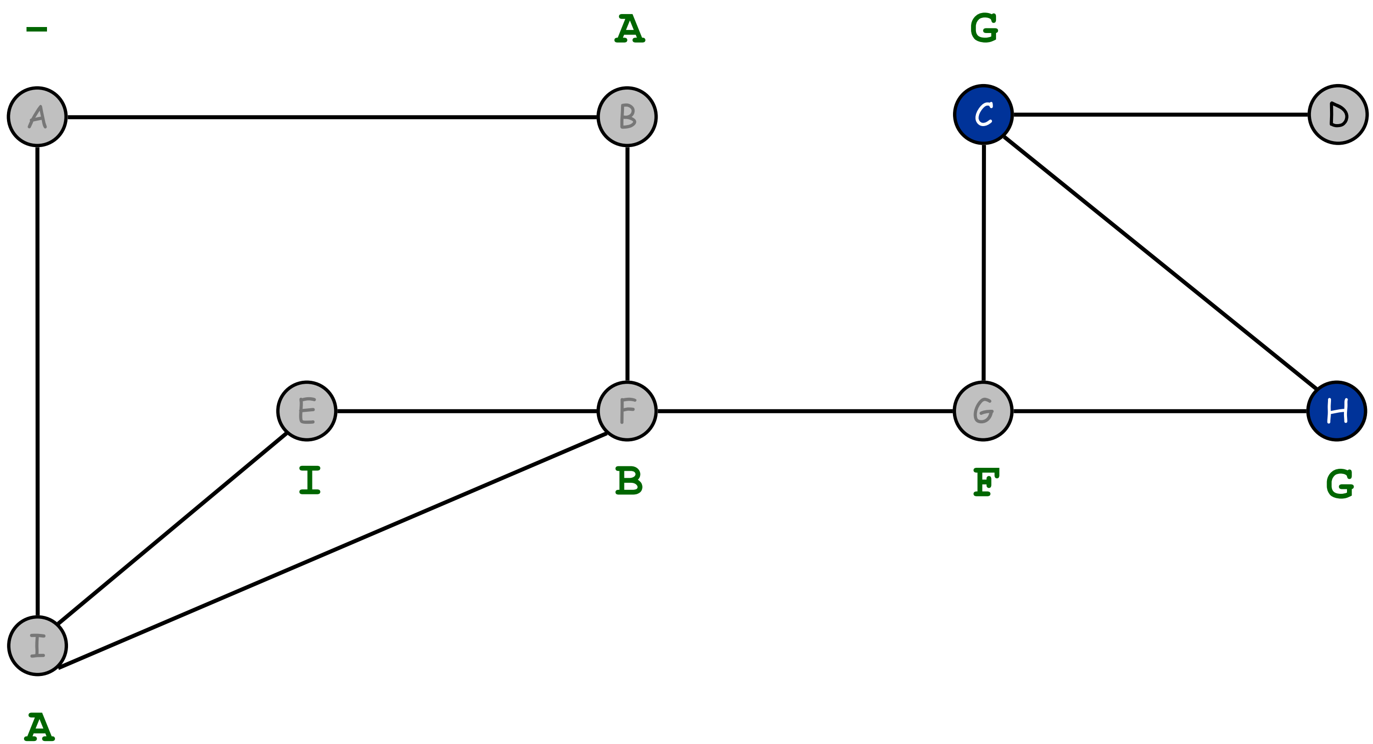
G finished

front

C H

FIFO Queue

Breadth First Search

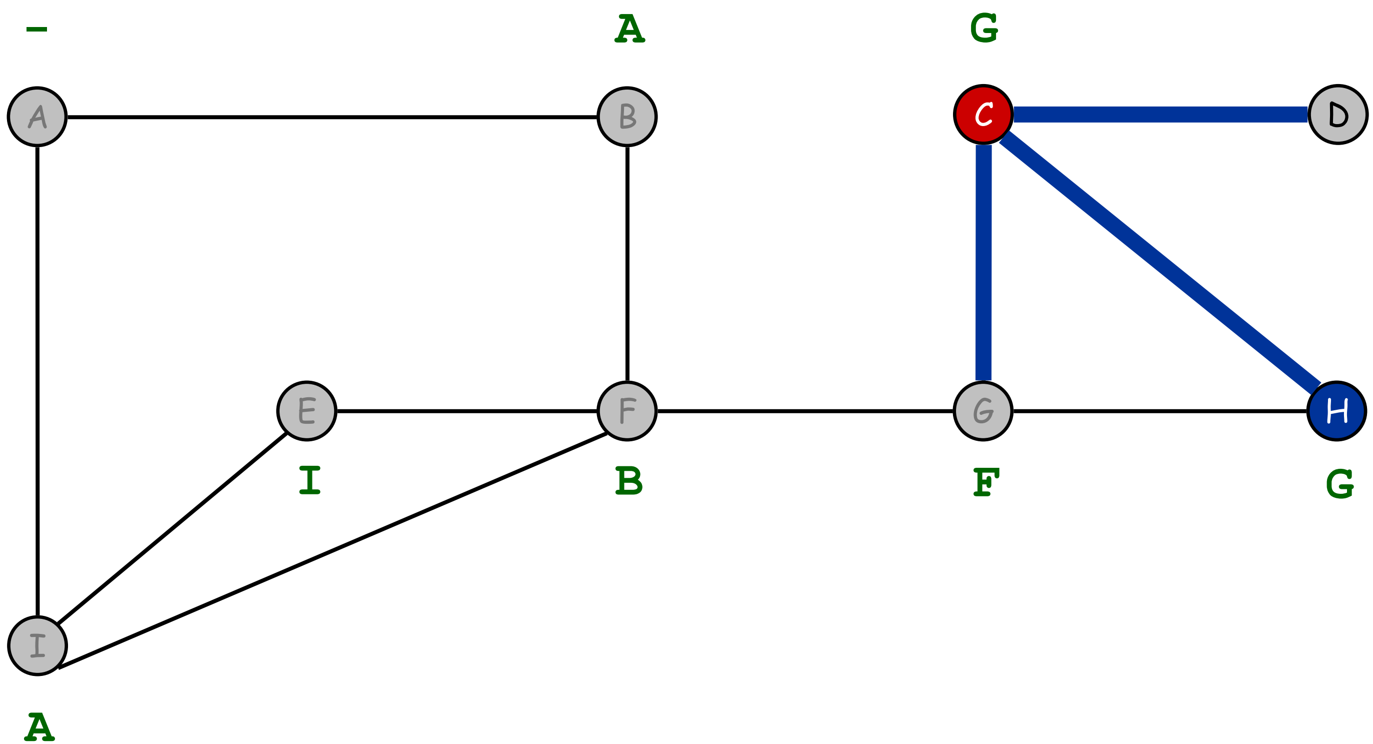


dequeue next vertex

front **C H**

FIFO Queue

Breadth First Search

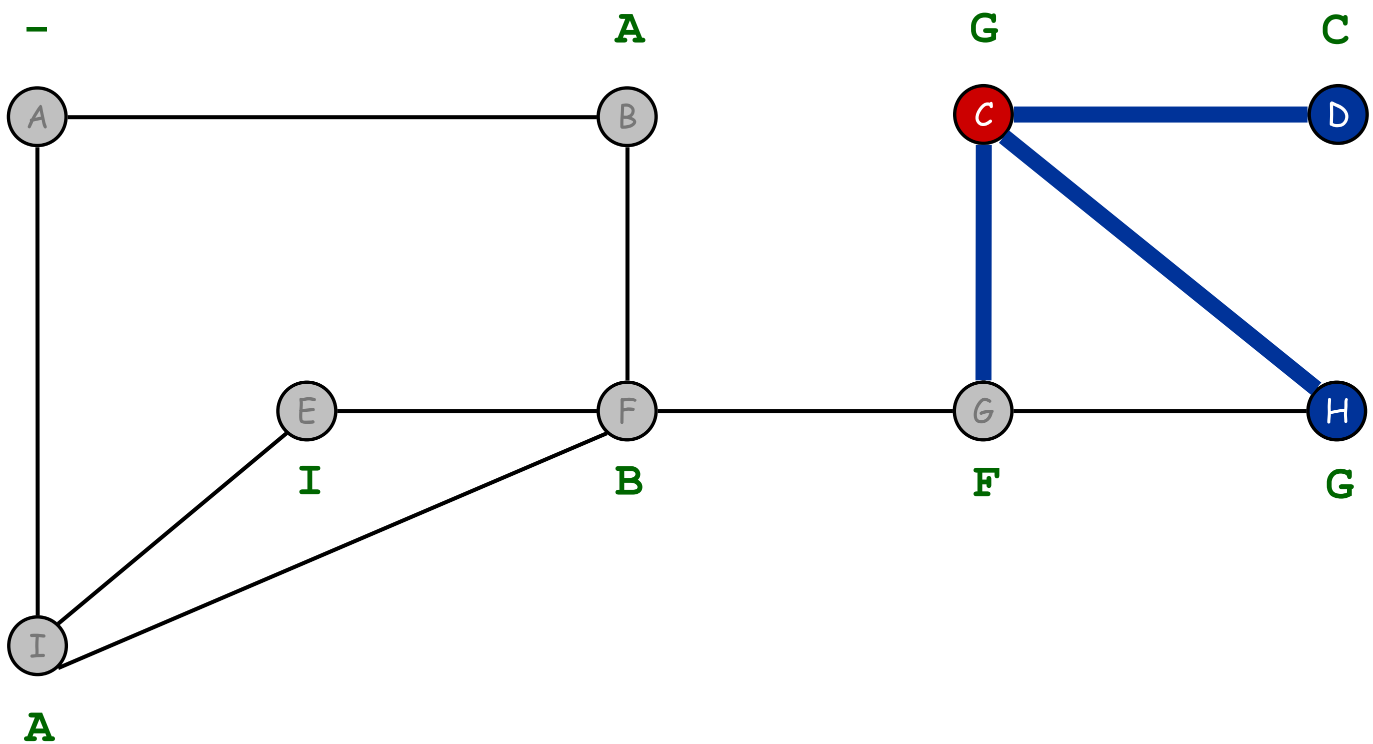


visit neighbors of C

front H

FIFO Queue

Breadth First Search

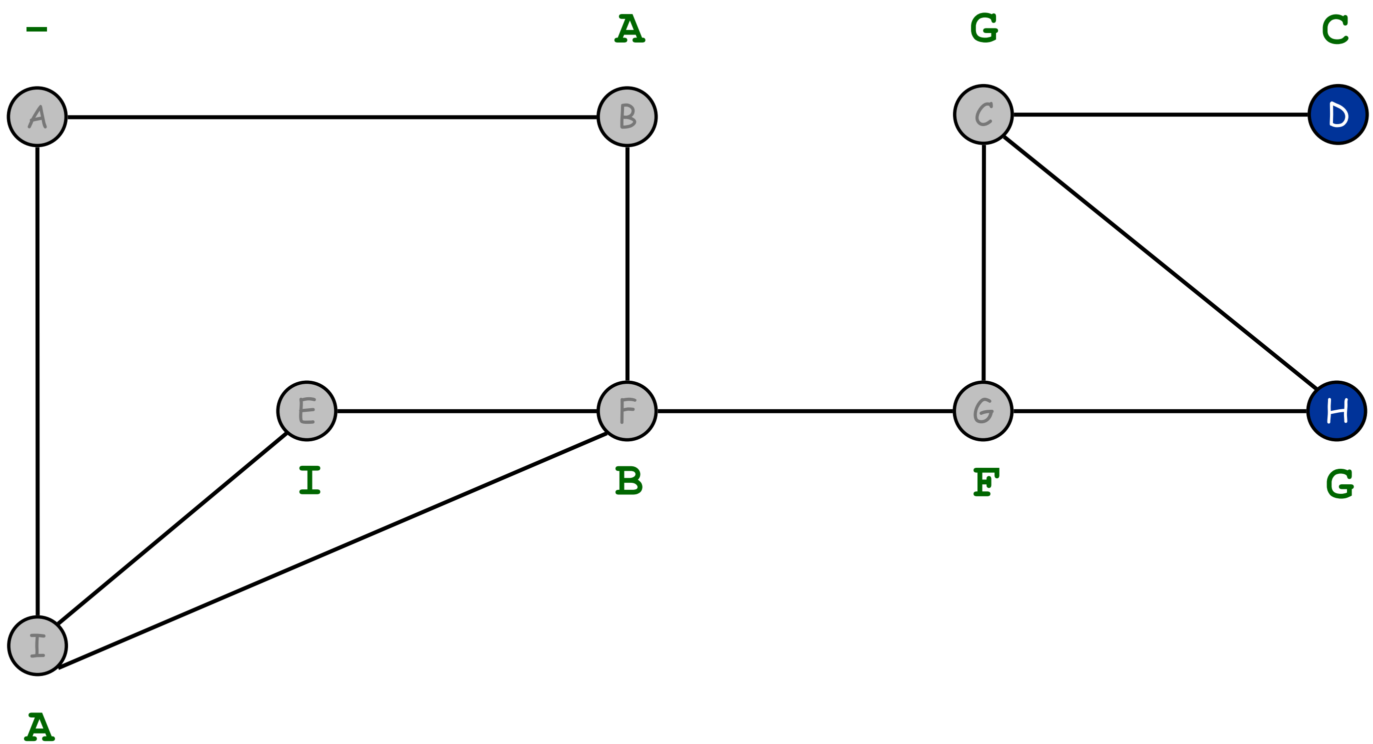


D discovered

front **H D**

FIFO Queue

Breadth First Search

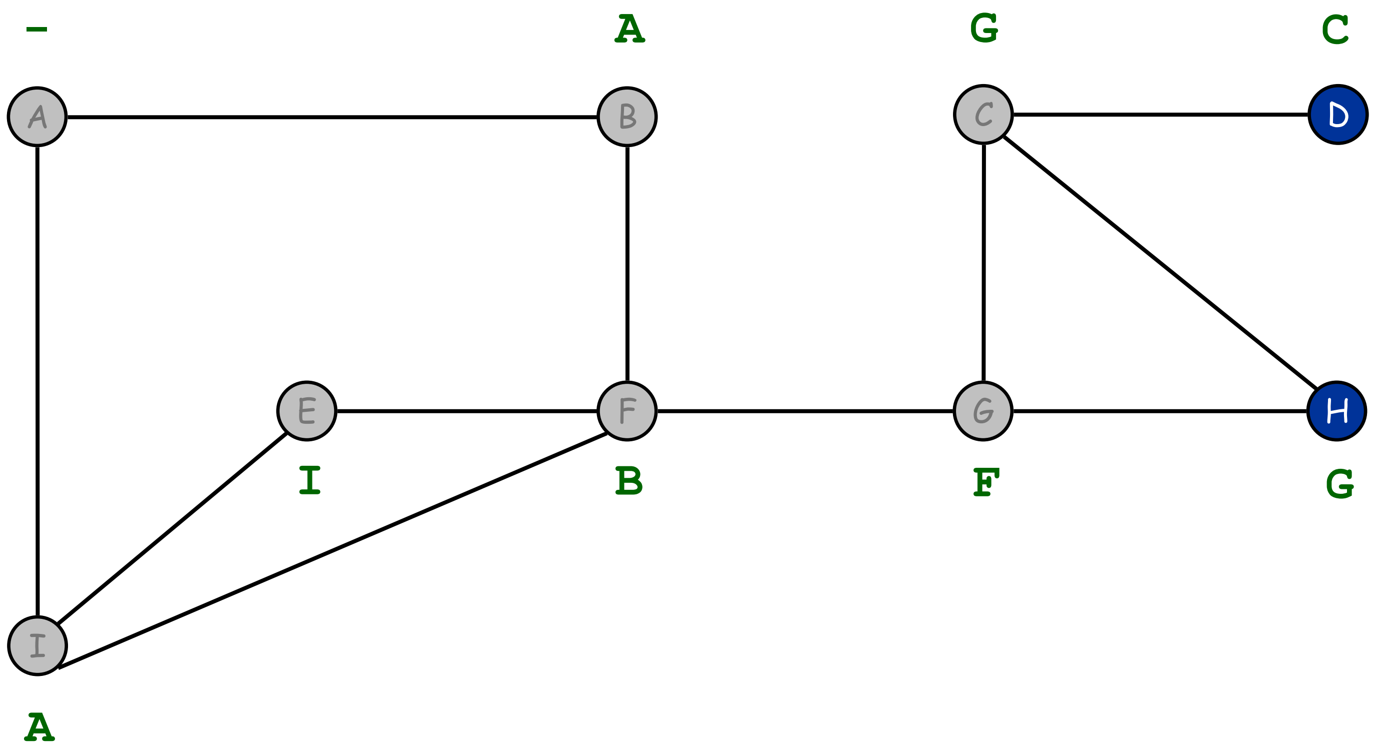


C finished

front H D

FIFO Queue

Breadth First Search

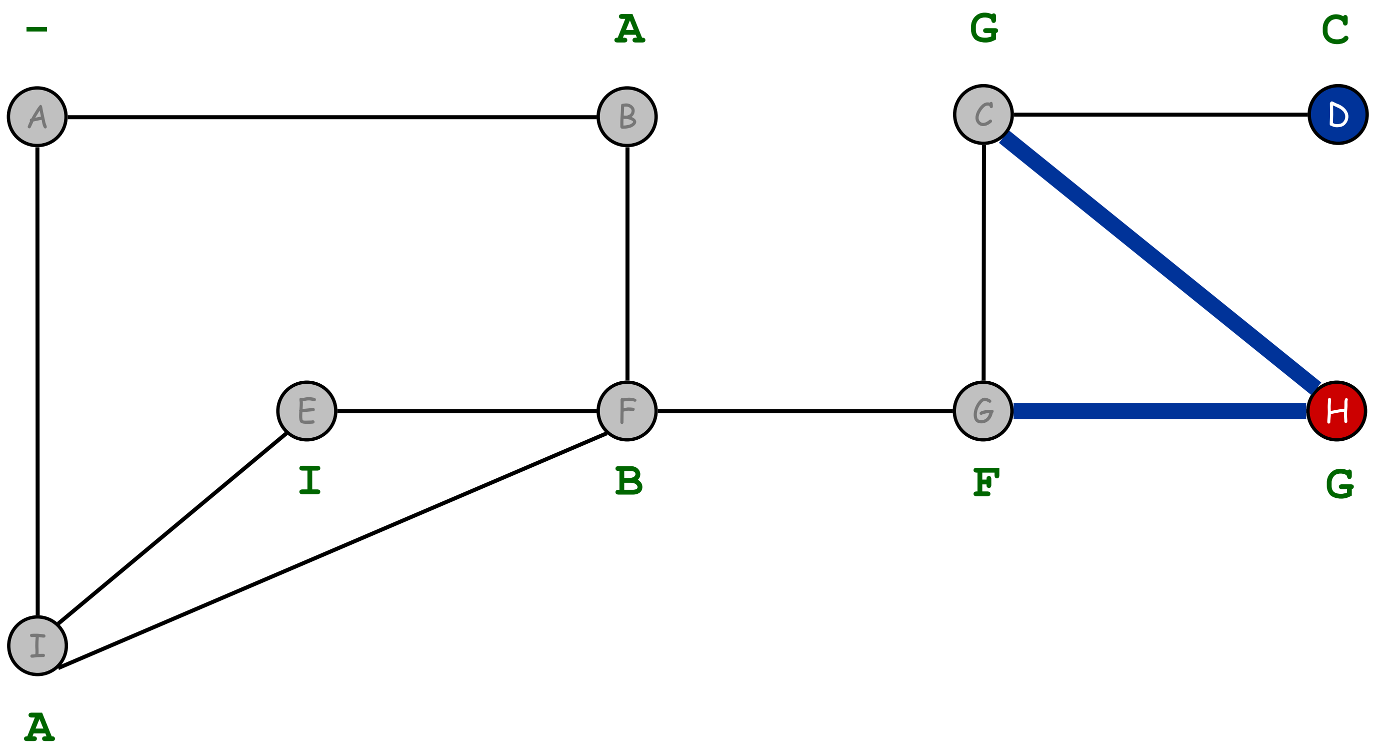


get next vertex

front H D

FIFO Queue

Breadth First Search

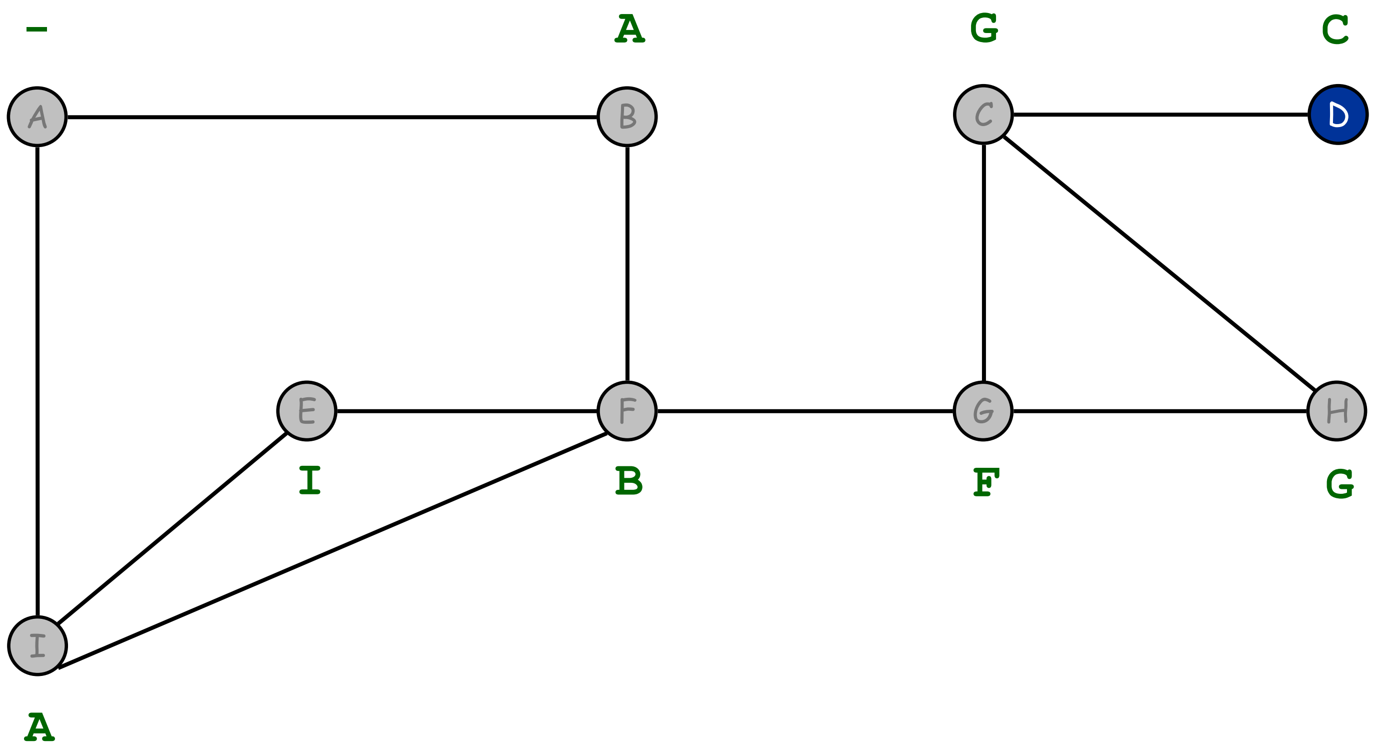


visit neighbors of H

front D

FIFO Queue

Breadth First Search

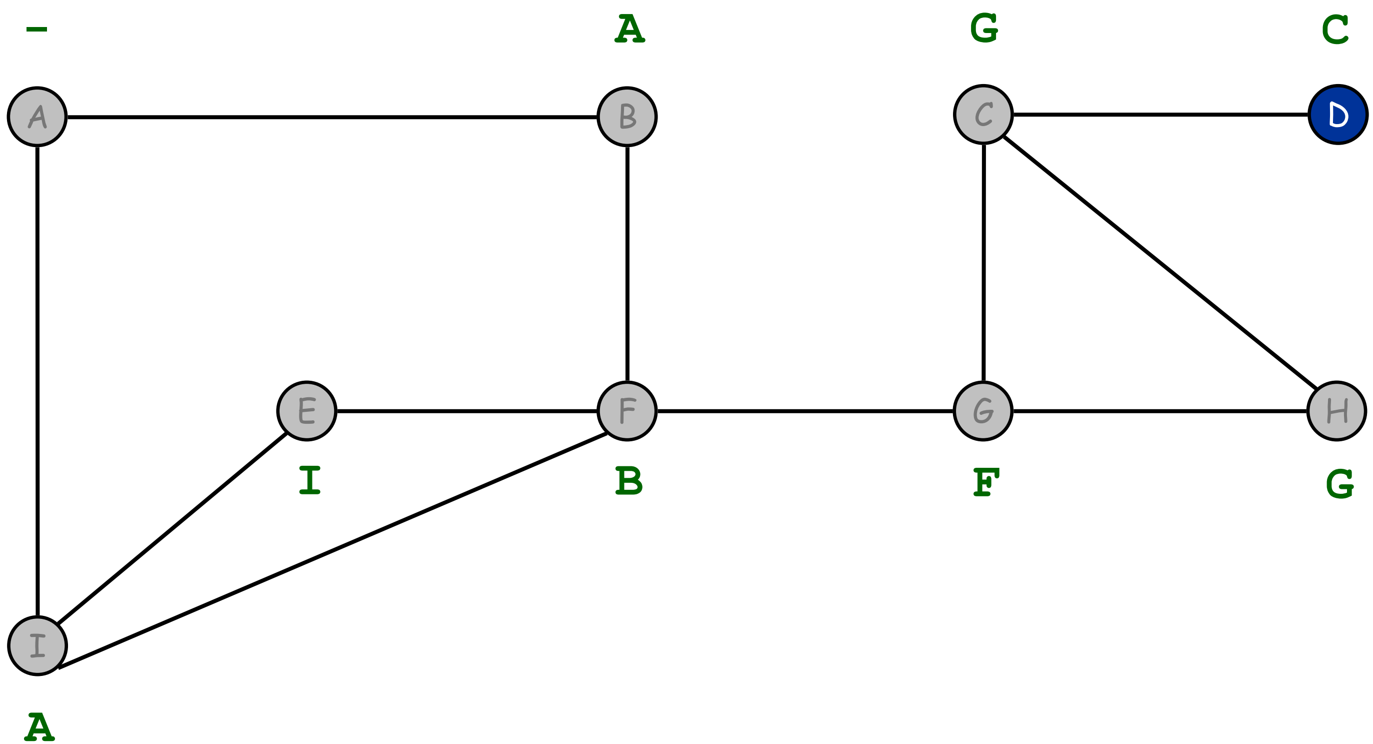


finished H

front D

FIFO Queue

Breadth First Search



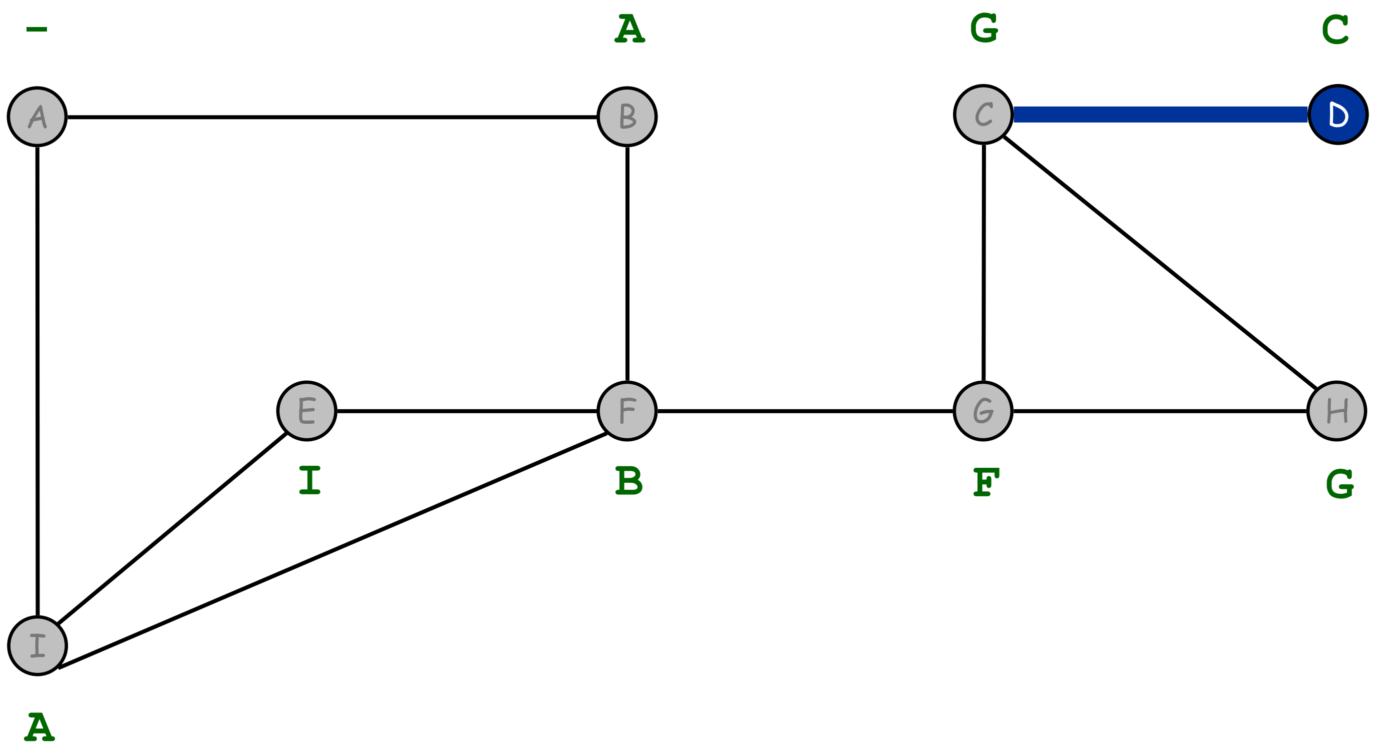
dequeue next vertex

front

D

FIFO Queue

Breadth First Search

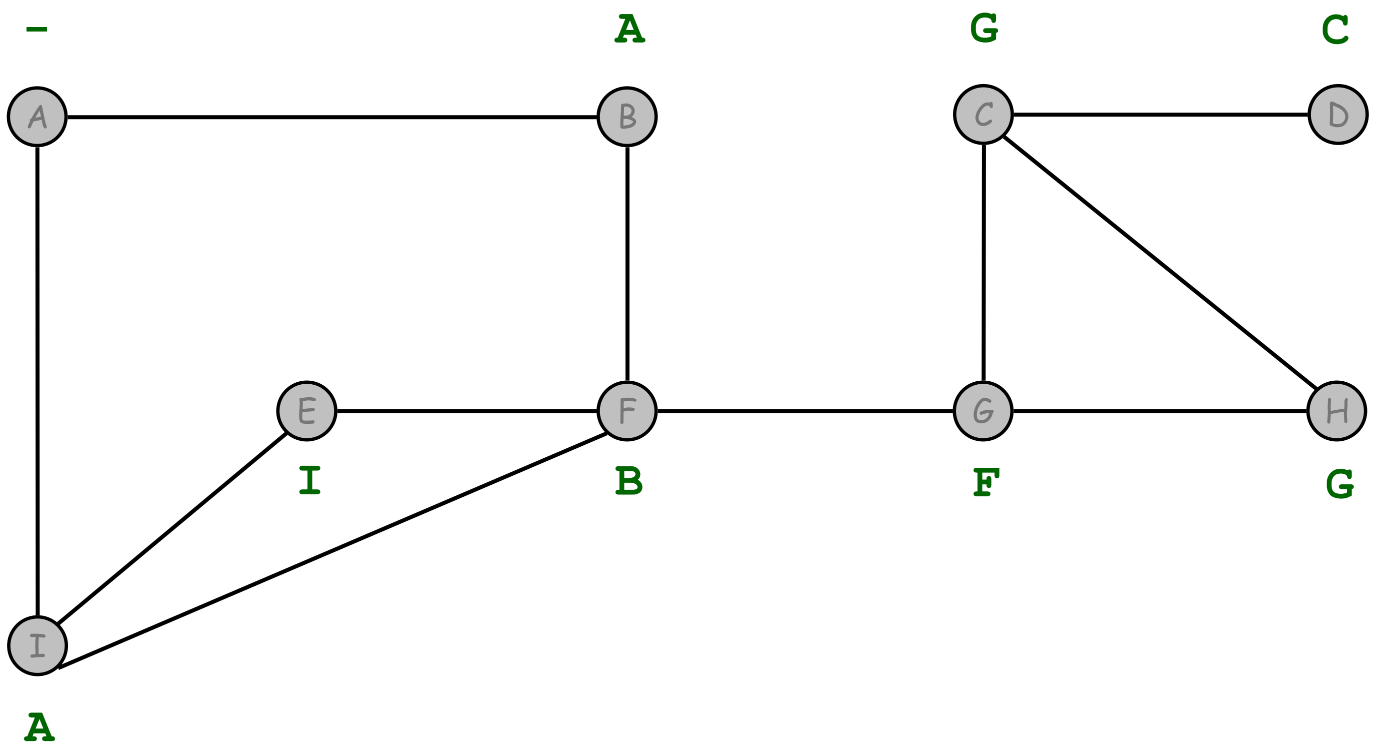


visit neighbors of D

front

FIFO Queue

Breadth First Search

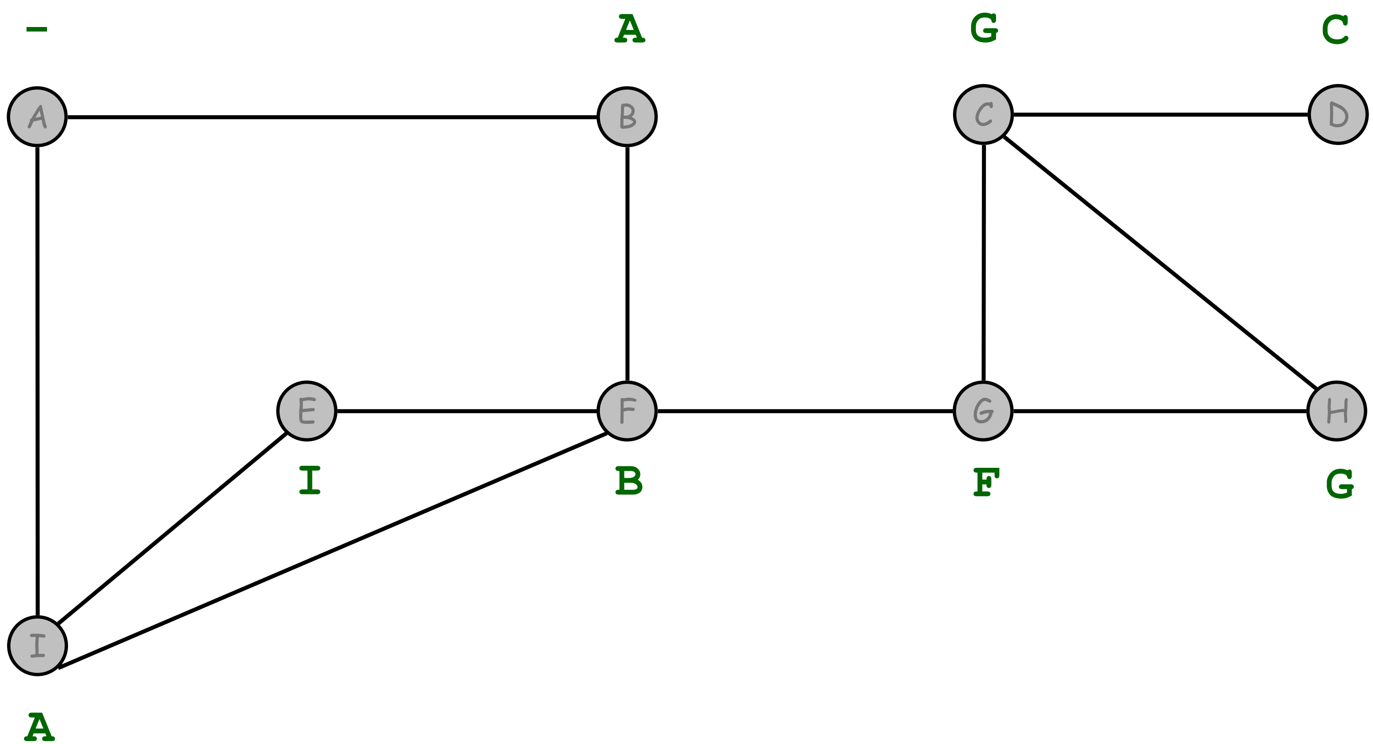


D finished

front

FIFO Queue

Breadth First Search

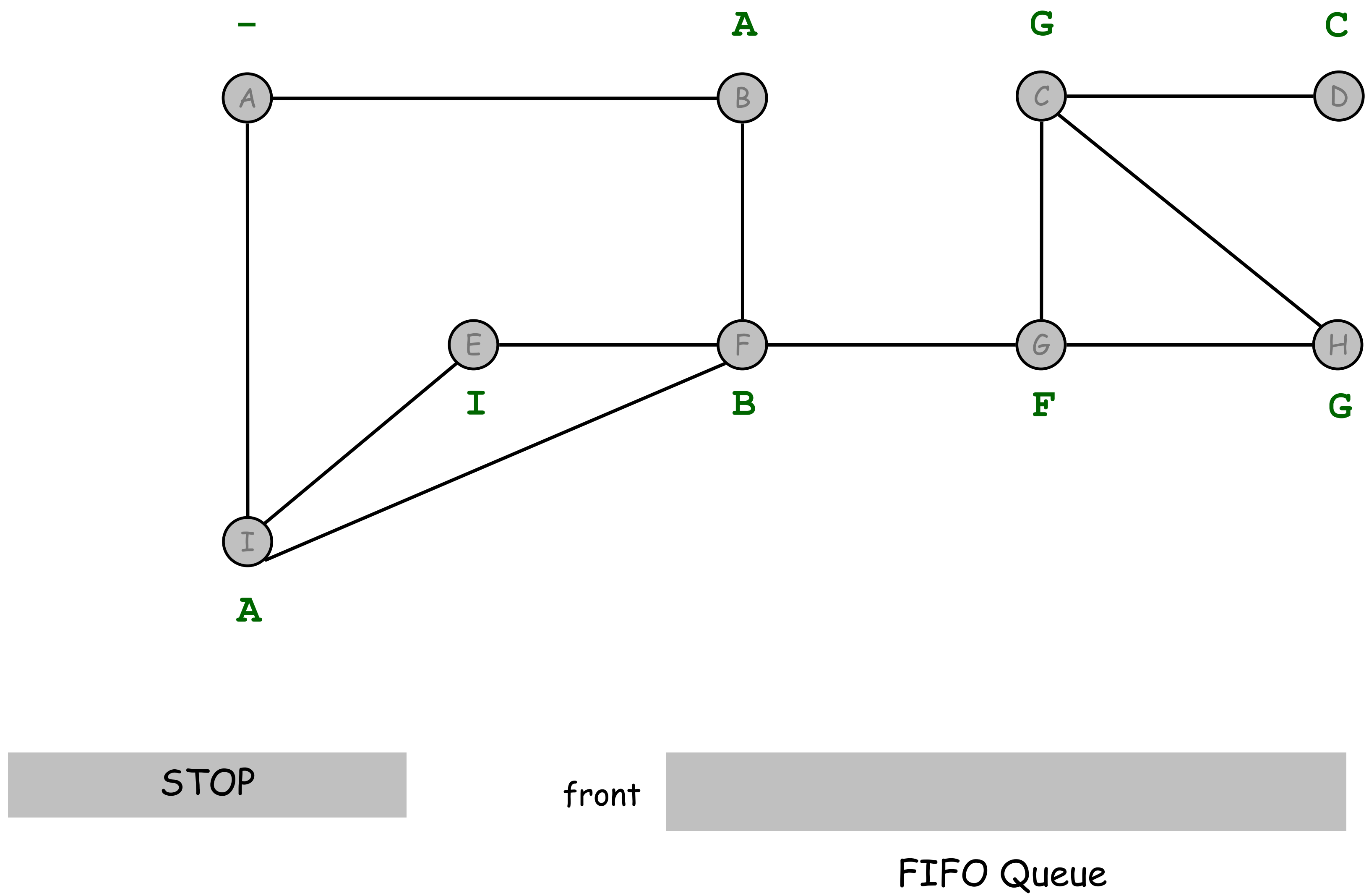


dequeue next vertex

front

FIFO Queue

Breadth First Search



Time Complexity

- Each visited vertex is put on (and so removed from) the queue exactly once
- When a vertex is removed from the queue, we examine its adjacent vertices
  - $O(|V|)$ if adjacency matrix used
  - O(vertex degree) if adjacency lists used
- Total time
  - $O(|E|+|V|)$, where E is number of vertices in the component that is searched (adjacency matrix)$=O(|V|^2)$
  - $O(|V|$ + sum of component vertex degrees$)$ (adj. lists)

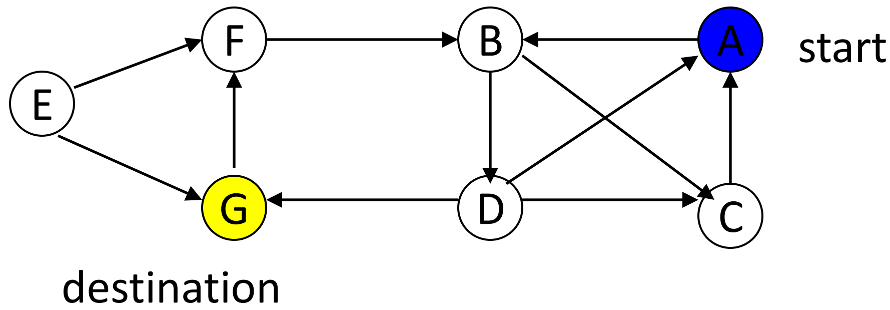
  $= O(|V|$ + number of edges in component$)=O(|V|+|E|)$

Applications: Finding a Path

- Find path from source vertex s to destination vertex d
- Use graph search starting at s and terminating as soon as we reach d
  - Need to remember edges traversed
- Use depth – first search ?
- Use breath – first search?

DFS vs. BFS

DFS Process



A

DFS on A

B
A

DFS on B

C
B
A

DFS on C

D
B
A

Call DFS on D

Return to call on B

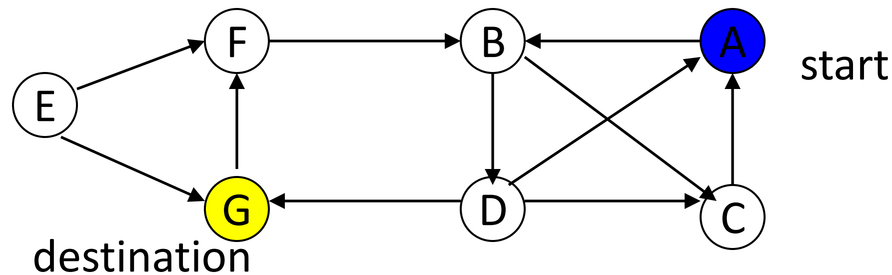
G
D
B
A

Call DFS on G

found destination - done!
Path is implicitly stored in DFS recursion
Path is: A, B, D, G

DFS vs. BFS

BFS Process



rear front

A

Initial call to BFS on A
Add A to queue

rear front

B

Dequeue A
Add B

rear front

D C

Dequeue B
Add C, D

rear front

D

Dequeue C
Nothing to add

rear front

G

Dequeue D
Add G

found destination - done!
Path must be stored separately

Path From Vertex s To Vertex d

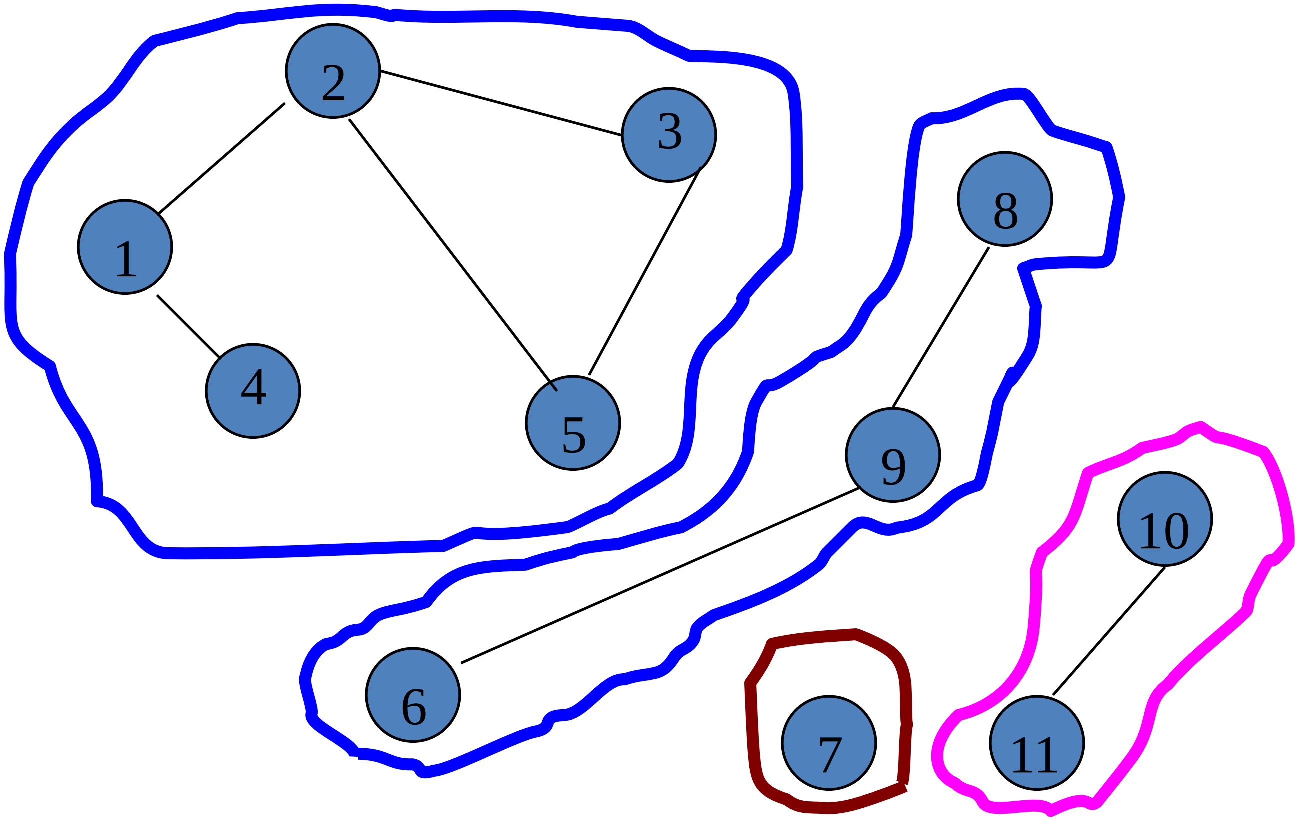
- Time
  - $O(|V|^2)$ when adjacency matrix used
  - $O(|V|+|E|)$ when adjacency lists used ($|E|$ is number of edges)

Is The Graph Connected?

- Start a breadth-first search at any vertex of the graph
- Graph is connected iff all n vertices get visited
- Time
  - $O(|V|^2)$ when adjacency matrix used
  - $O(|V|+|E|)$ when adjacency lists used ($|E|$ is number of edges)

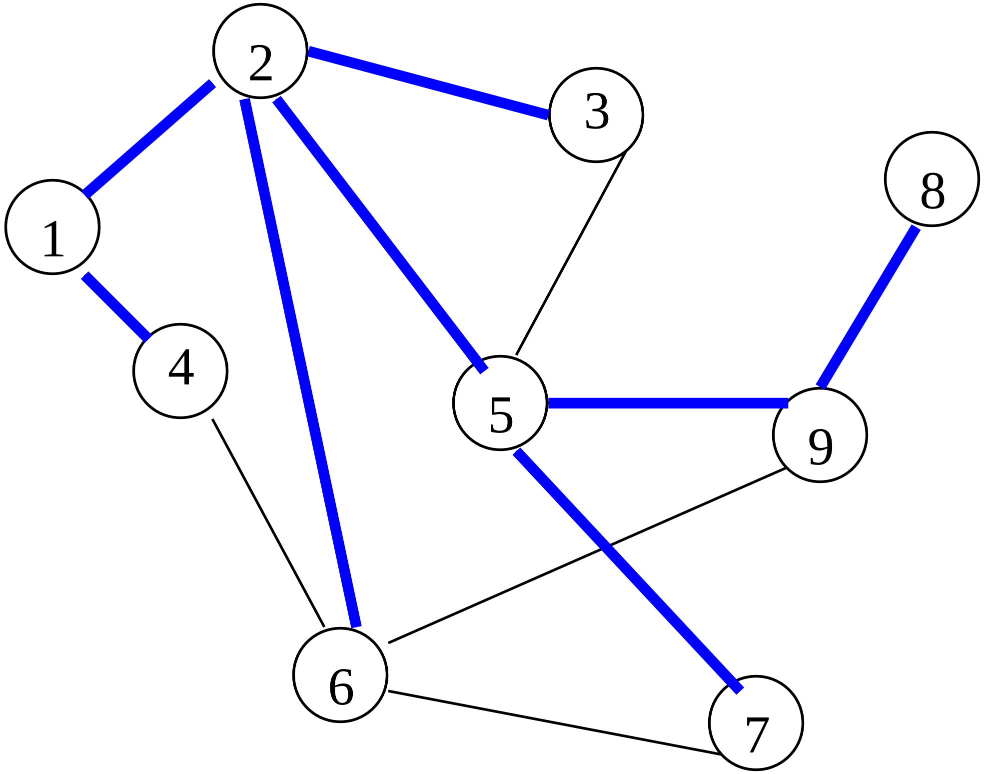
Connected Components

- Start a BFS at any as yet unvisited vertex of the graph
- Newly visited vertices (plus edges between them) define a component
- Repeat until all vertices are visited



Breadth First Spanning Tree

Breadth-first search from vertex 1



One possible breadth first spanning tree

- Keep track of edges used to reach new vertices
- These edges form a spanning tree if the graph is connected

Spanning Tree

- Start a breadth-first search at any vertex of the graph
- If graph is connected, the n-1 edges used to get to unvisited vertices define a spanning tree (breadth-first spanning tree)
- Time
  - $O(n^2)$ when adjacency matrix used
  - $O(n+e)$ when adjacency lists used (e is number of edges)