Computer Science Department
College of Computer at Al-Lith

جامعة أم القرى
UMM AL-QURA UNIVERSITY

Data Structures

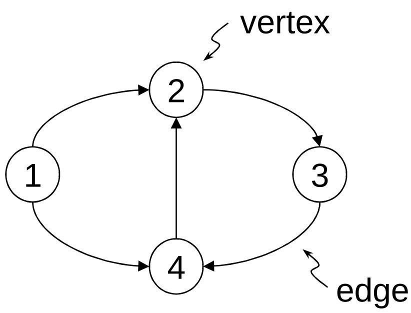
Chapter 7: Graph

Instructor Maher Hadiji
hdiji.maher@gmail.com

2015-2016

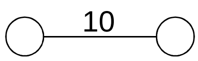
What's a Graph?

- A bunch of vertices connected by edges.



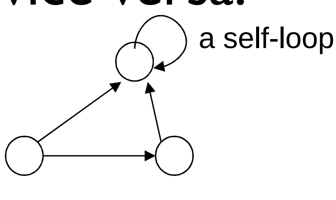
Basic Concepts

- A *graph* is an ordered pair (V, E).
- V is the set of vertices. (You can think of them as integers 1, 2, …, n.)
- E is the set of edges. An edge is a pair of vertices: (u, v).
- Note: since E is a set, there is at most one edge between two vertices. (*Hypergraphs* permit multiple edges.)
- Edges can be labeled with a *weight*:

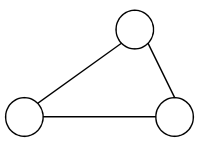


Concepts: Directedness

- In a *directed* graph, the edges are "one-way." So an edge (u, v) means you can go from u to v, but not vice versa.

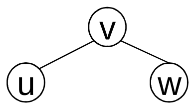


- In an *undirected* graph, there is no direction on the edges: you can go either way. (Also, no self-loops.)

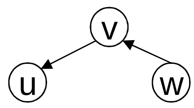


Concepts: Adjacency

- Two vertices are *adjacent* if there is an edge between them.
- For a directed graph, *u* is adjacent to *v* iff there is an edge (*v*, *u*).



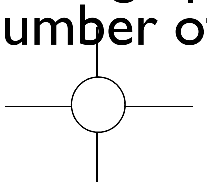
u is adjacent to v.
v is adjacent to u and w.
w is adjacent to v.



u is adjacent to v.
v is adjacent to w.

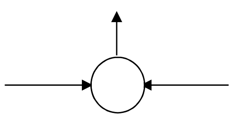
Concepts: Degree

- Undirected graph: The *degree* of a vertex is the number of edges touching it.



degree 4

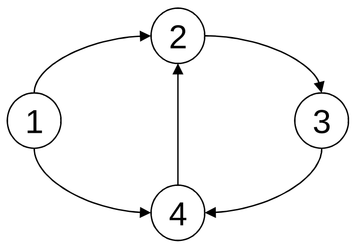
- For a directed graph, the *in-degree* is the number of edges entering the vertex, and the *out-degree* is the number leaving it. The *degree* is the *in-degree* + the *out-degree*.



in-degree 2, out-degree 1

Concepts: Path

- A *path* is a sequence of adjacent vertices. The *length* of a path is the number of edges it contains, i.e. one less than the number of vertices.



Is there a path from 1 to 4?

What is its length?

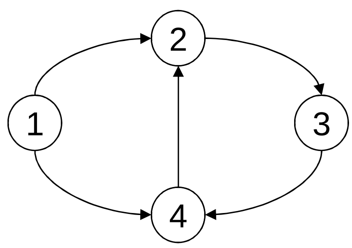
What about from 4 to 1?

How many paths are there from 2 to 3? From 2 to 2? From 1 to 1?

- We write u $\Rightarrow$ v if there is path from u to v. We say v is *reachable* from u.

Concepts: Cycle

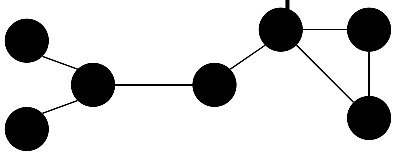
- A *cycle* is a path of length at least 1 from a vertex to itself.
- A graph with no cycles is *acyclic*.
- A path with no cycles is a *simple* path.



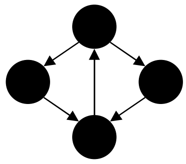
- The path <2, 3, 4, 2> is a cycle.

Connectivity

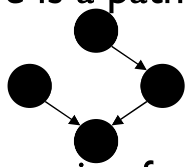
- Undirected graphs are *connected* if there is a path between any two vertices



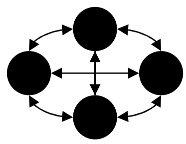
- Directed graphs are *strongly connected* if there is a path from any one vertex to any other



- Directed graphs are *weakly connected* if there is a path between any two vertices, *ignoring direction*

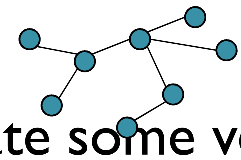


- A *complete* graph has an edge between every pair of vertices



Concepts: Trees

- A *free tree* is a connected, acyclic, undirected graph.



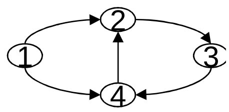
- To get a *rooted tree*, designate some vertex as the root.
- If the graph is disconnected, it's a *forest.*
- Facts about free trees:

- |E| = |V| -1
- Any two vertices are connected by exactly one path.
- Removing an edge disconnects the graph.
- Adding an edge results in a cycle.

Graph Size

- We describe the time and space complexity of graph algorithms in terms of the number of vertices, |V|, and the number of edges, |E|.
- |E| can range from 0 (a totally disconnected graph) to $|V|^2$ (a directed graph with every possible edge, including self-loops).
- we write $\Theta(V + E)$ instead of $\Theta(|V| + |E|)$.

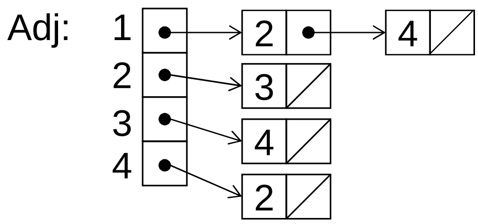
Representing Graphs



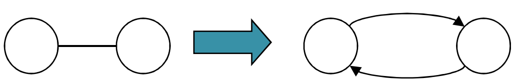
- Adjacency matrix: if there is an edge from vertex i to j, a_{ij} = 1; else, a_{ij} = 0.
- Space: $\Theta(V^2)$

$$
\begin{array}{c c}
 & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}
\end{array}
$$

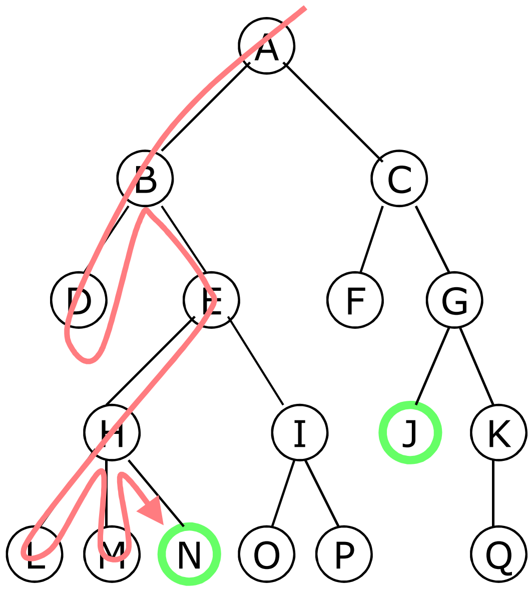
- Adjacency list: Adj[v] lists the vertices adjacent to v.
- Space: $\Theta(V+E)$



Represent an undirected graph by a directed one:



Depth-first searching



- 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

Fixing Bad-DFS

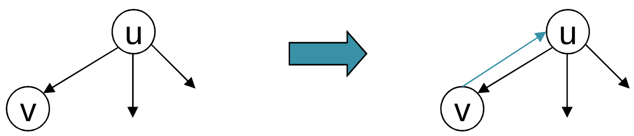
- We've got to indicate when a node has been visited.
- we'll use a color:
- WHITE never seen
- GRAY discovered but not finished (still exploring its descendants)
- BLACK finished

A Better DFS

- ▷ initially, all vertices are WHITE
- Better-DFS(u)
- color[u] ← GRAY
- number u with a "discovery time"
- for each v in Adj[u] do
- if color[v] = WHITE then ▷ avoid looping!
- Better-DFS(v)
- color[u] ← BLACK
- number u with a "finishing time"

Depth-First Spanning Tree

- As we'll see, DFS creates a tree as it explores the graph. Let's keep track of the tree as follows (actually it creates a forest not a tree):
- When v is explored directly from u, we will make u the parent of v, by setting the predecessor, aka, parent (π) field of v to u:



$\pi[v] \leftarrow u$

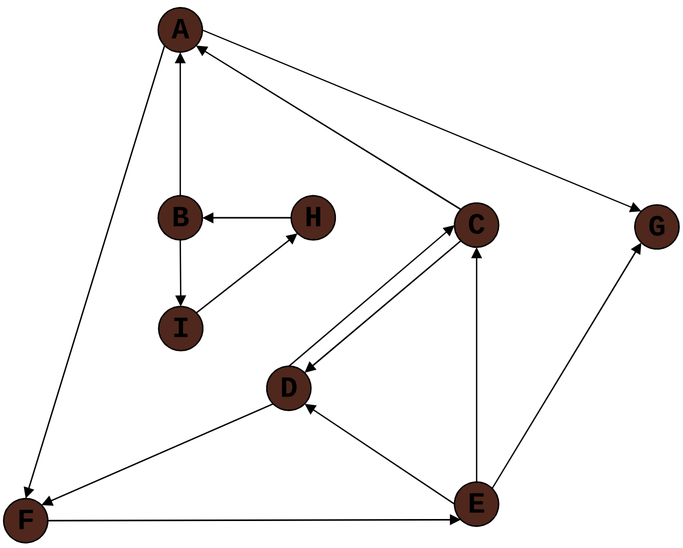
DFS(G)

```
for each vertex u ∈ V[G]
      do color[u] ← WHITE
         π[u] ← NIL
time ← 0
for each vertex u ∈ V[G]
      do if color[u] = WHITE
            then DFS-VISIT(u)
```

DFS-VISIT(u)

```
color[u] ← GRAY               ▷ White vertex u has just been discovered
time ← time + 1
d[u] ← time
for each v ∈ Adj[u]           ▷ Explore edge (u, v).
      do if color[v] = WHITE
            then π[v] ← u
                 DFS-VISIT(v)
color[u] ← BLACK              ▷ Blacken u; it is finished.
f[u] ← time ← time + 1
```

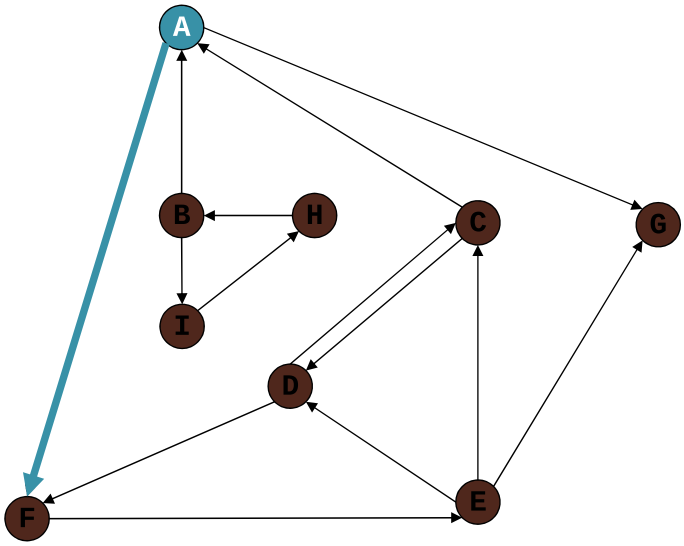
Directed Depth First Search



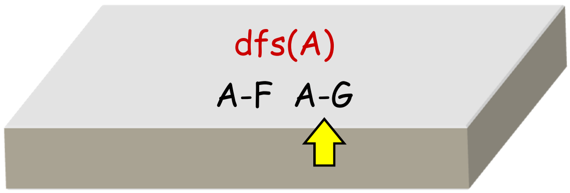
Adjacency Lists

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

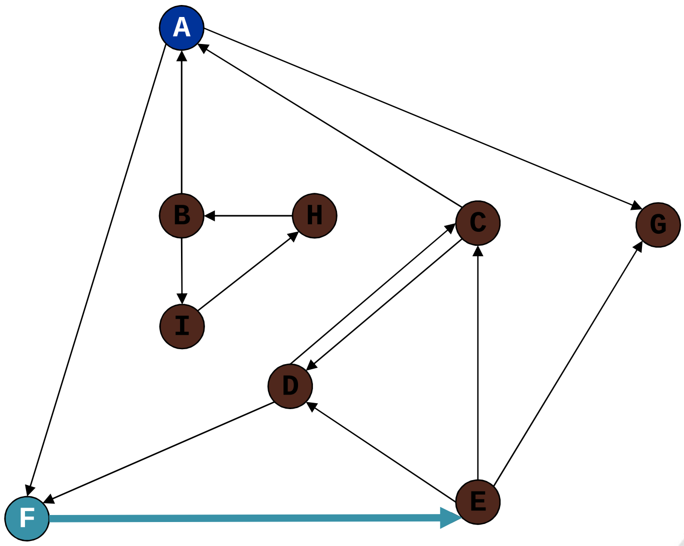
Directed Depth First Search



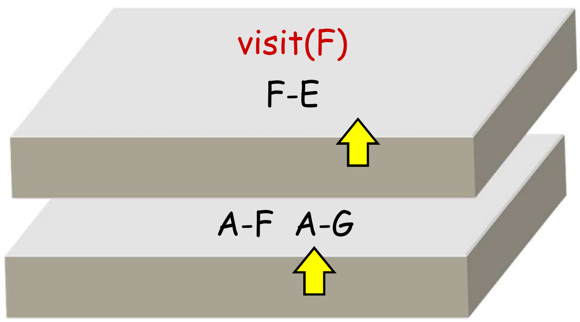
Function call stack:



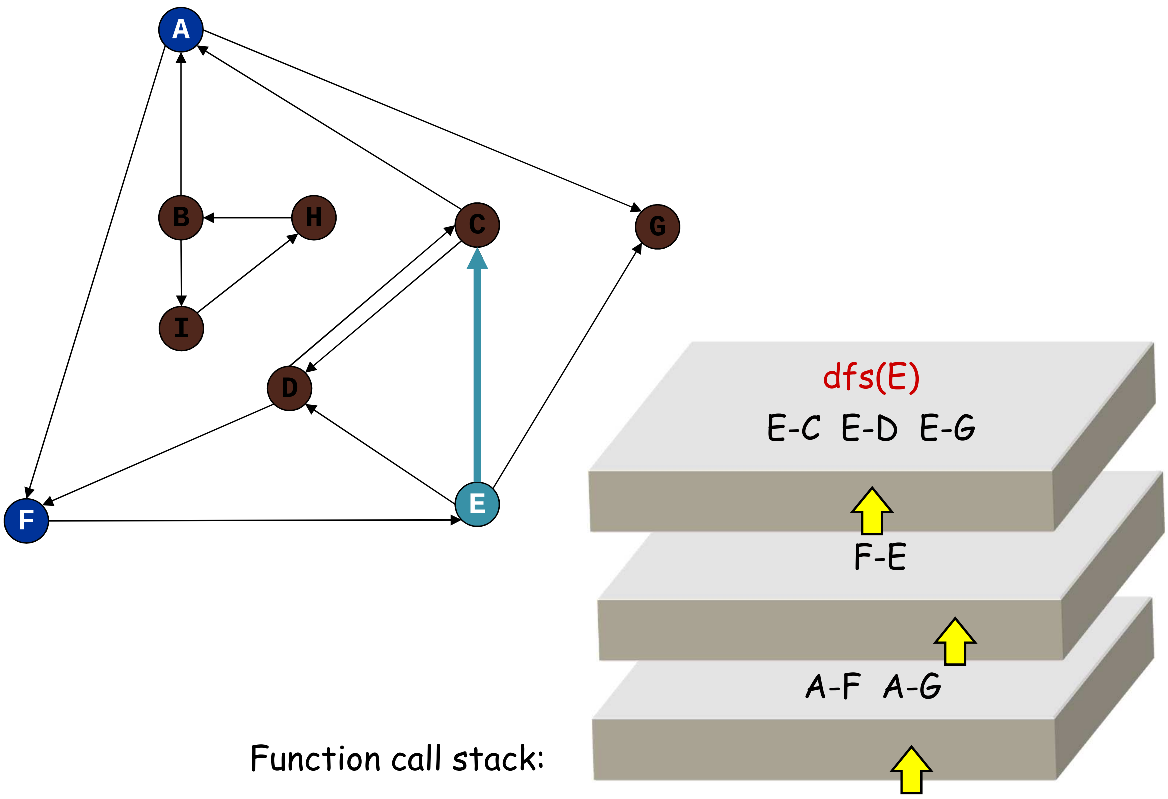
Directed Depth First Search



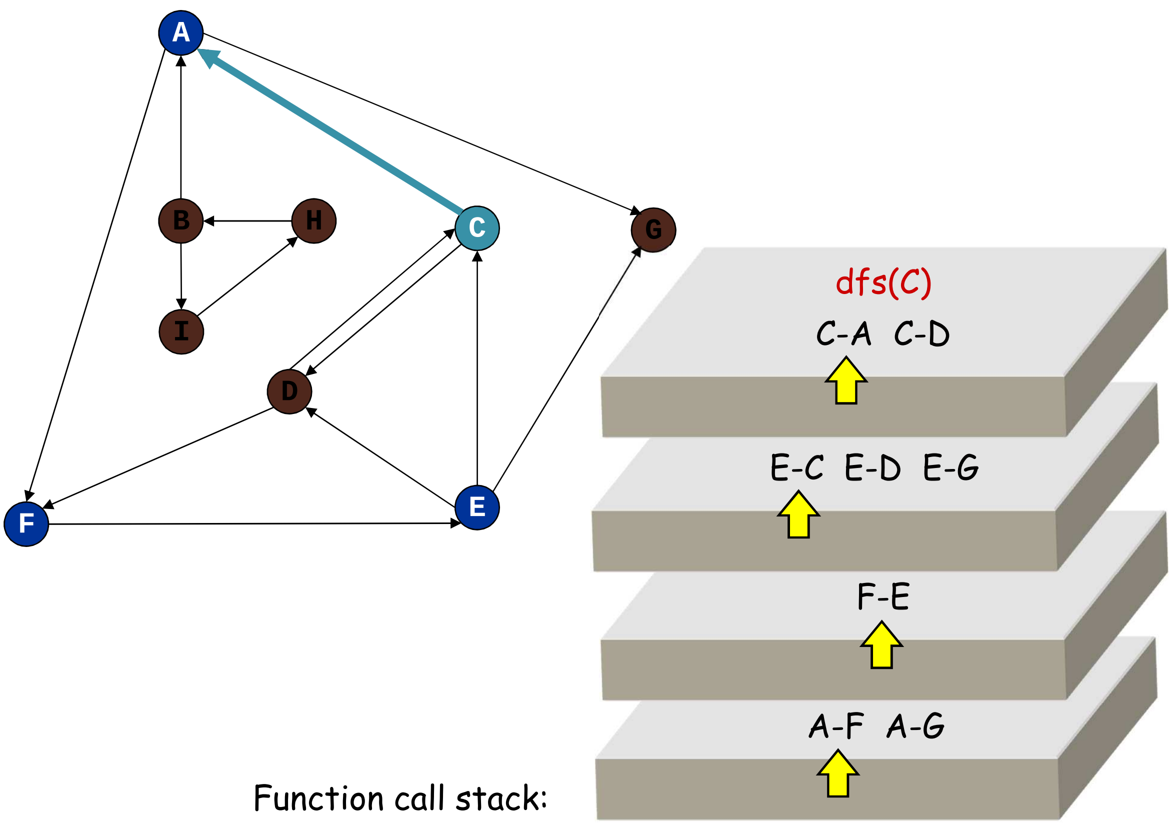
Function call stack:



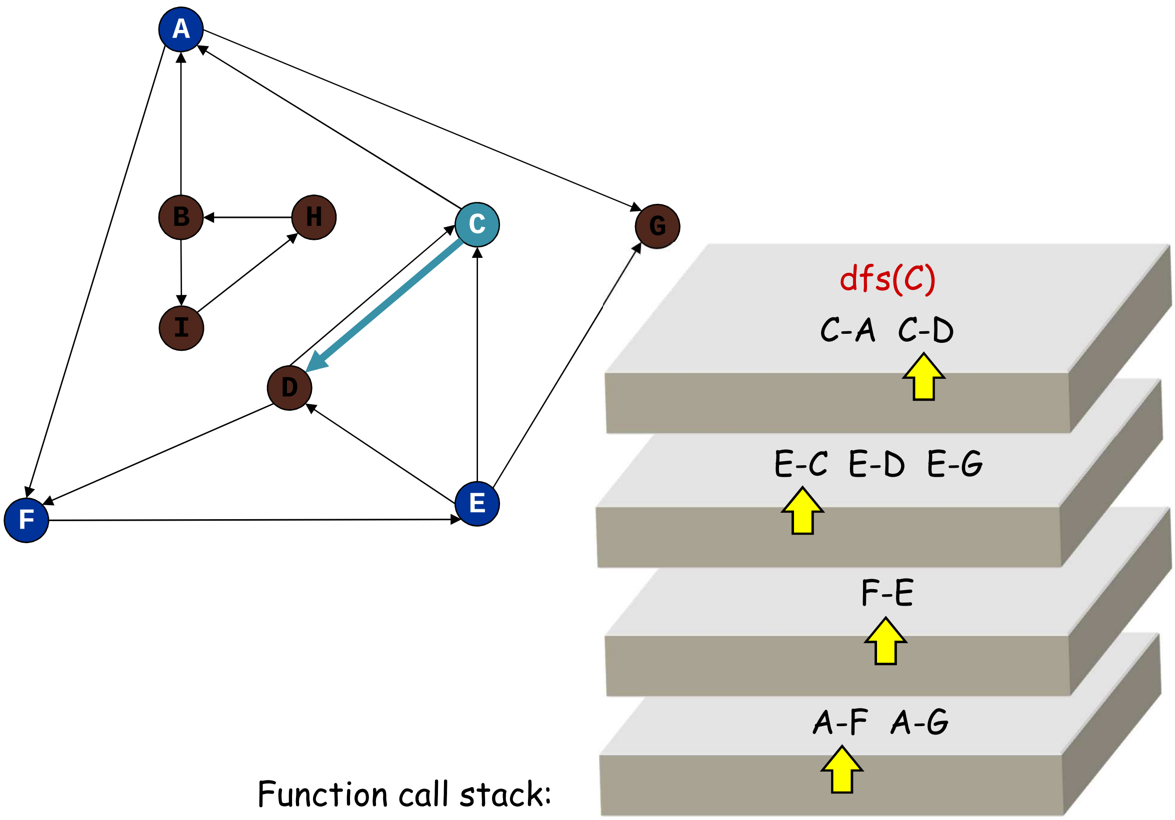
Directed Depth First Search



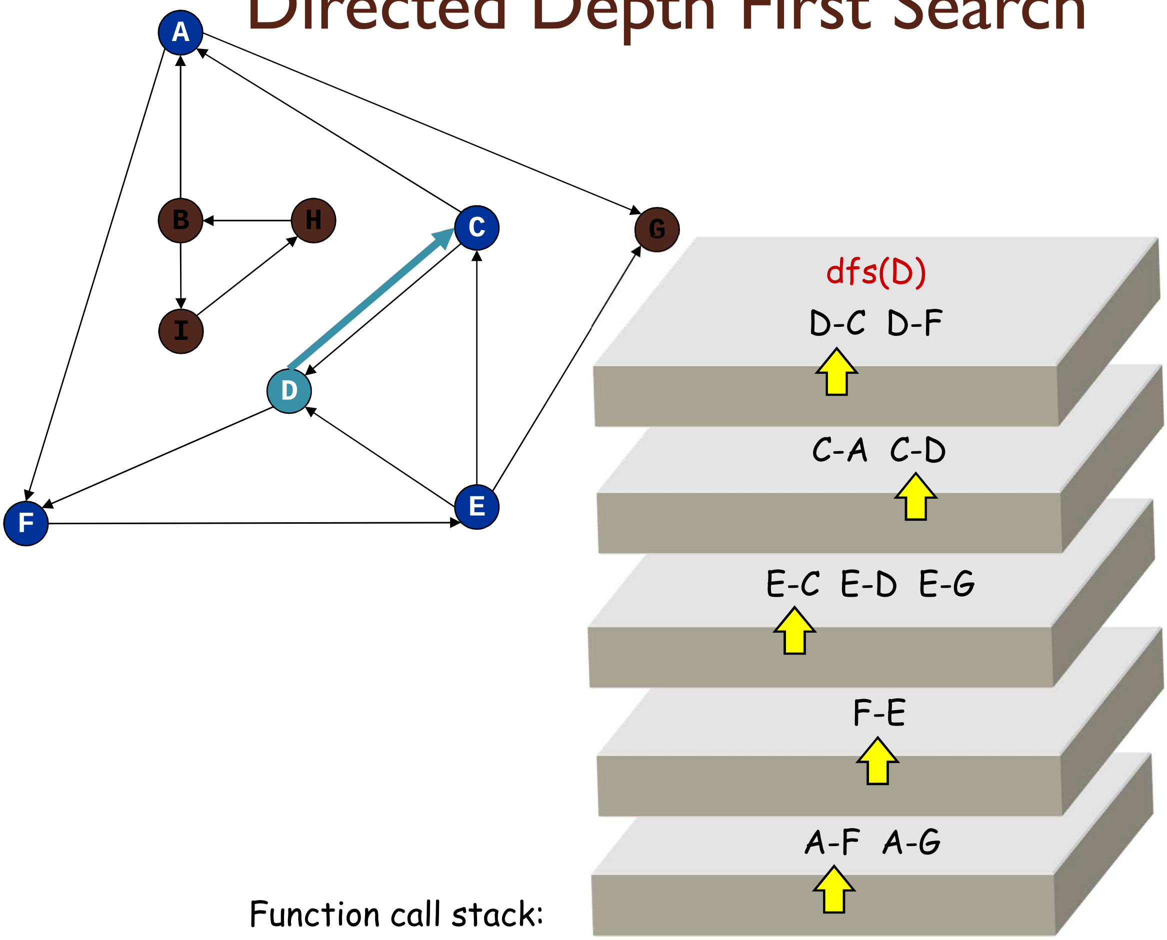
Directed Depth First Search



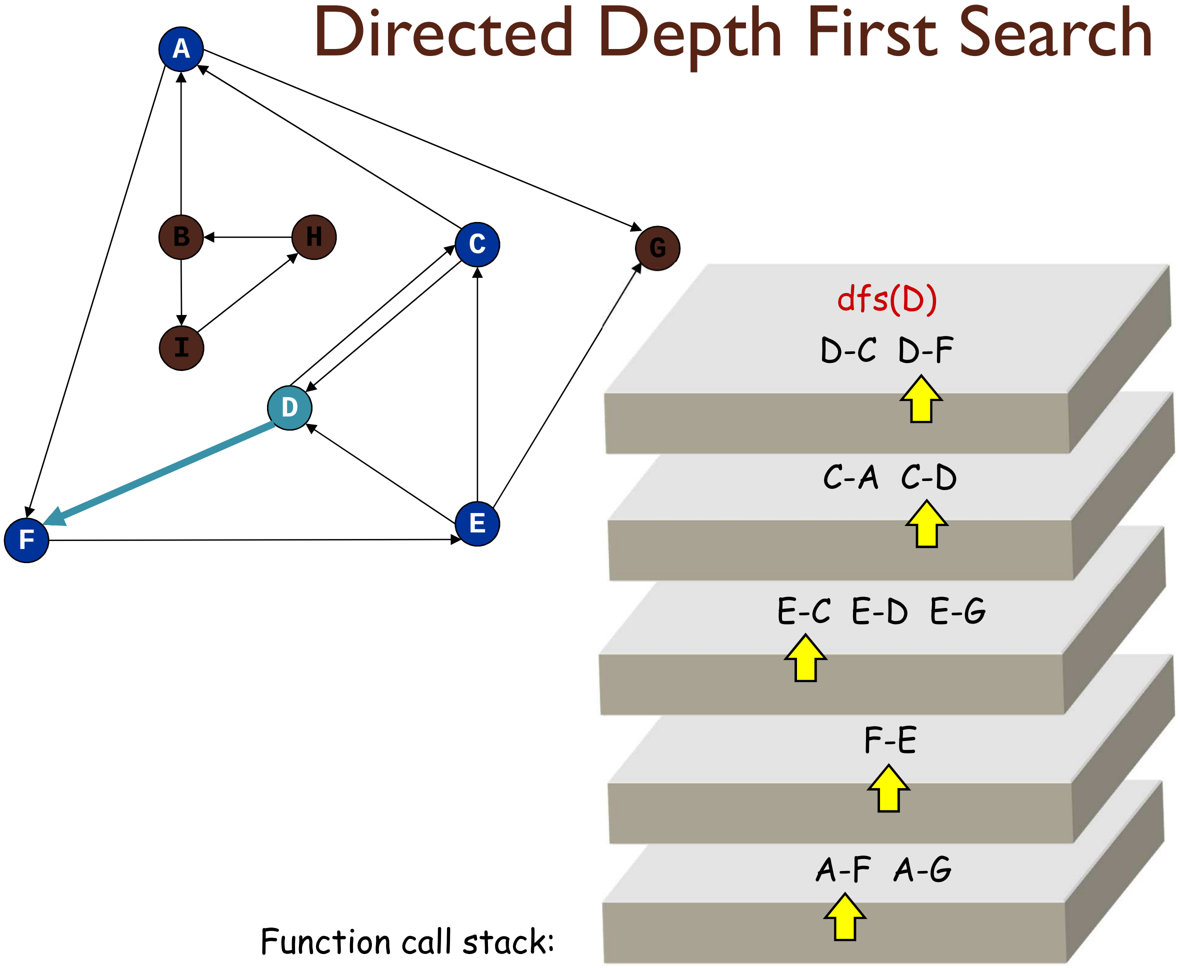
Directed Depth First Search



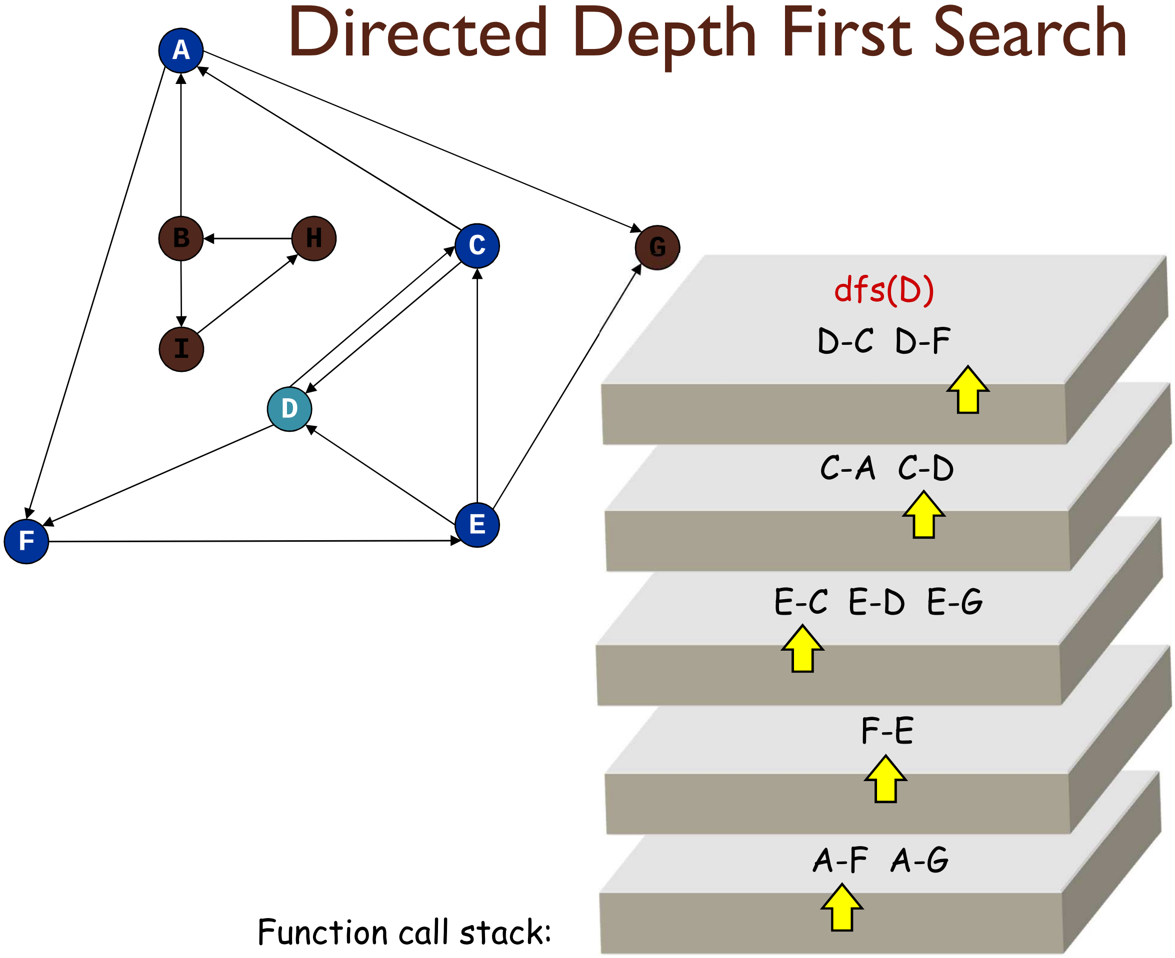
Directed Depth First Search



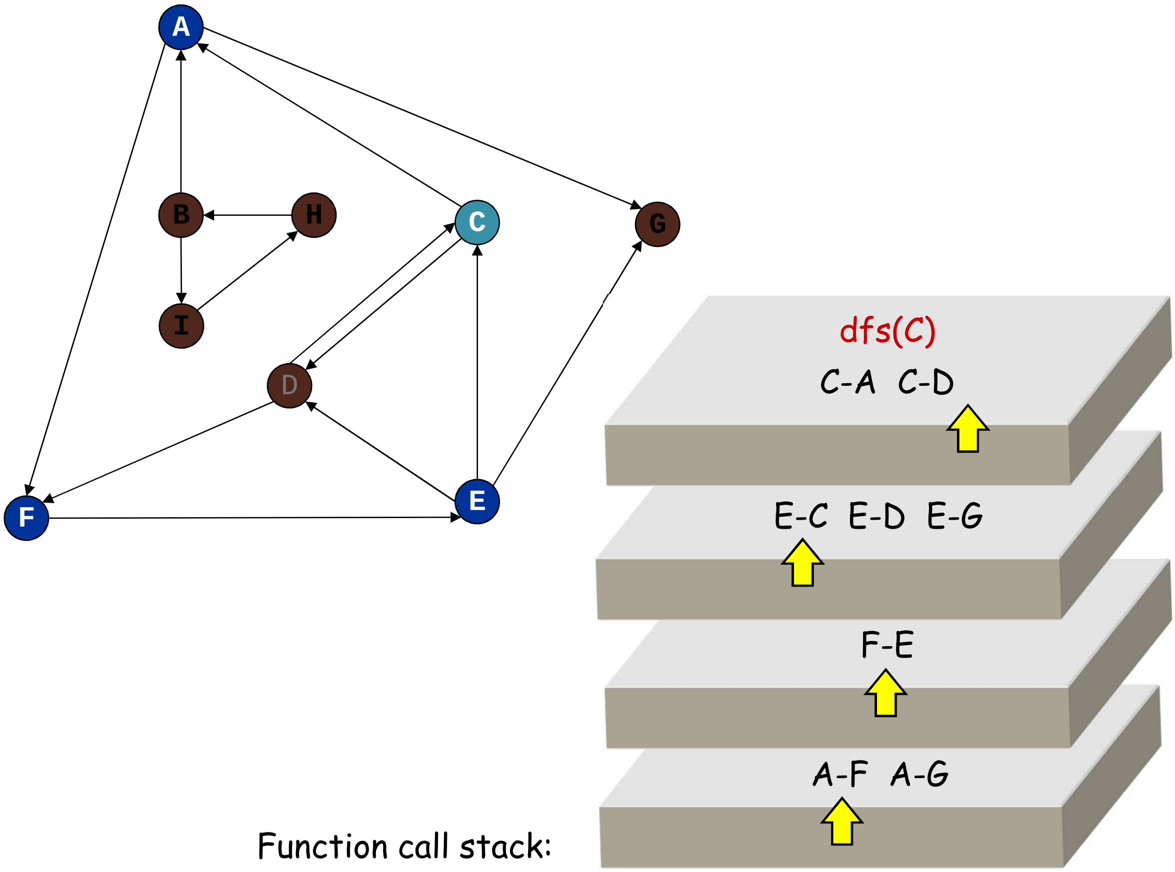
Directed Depth First Search



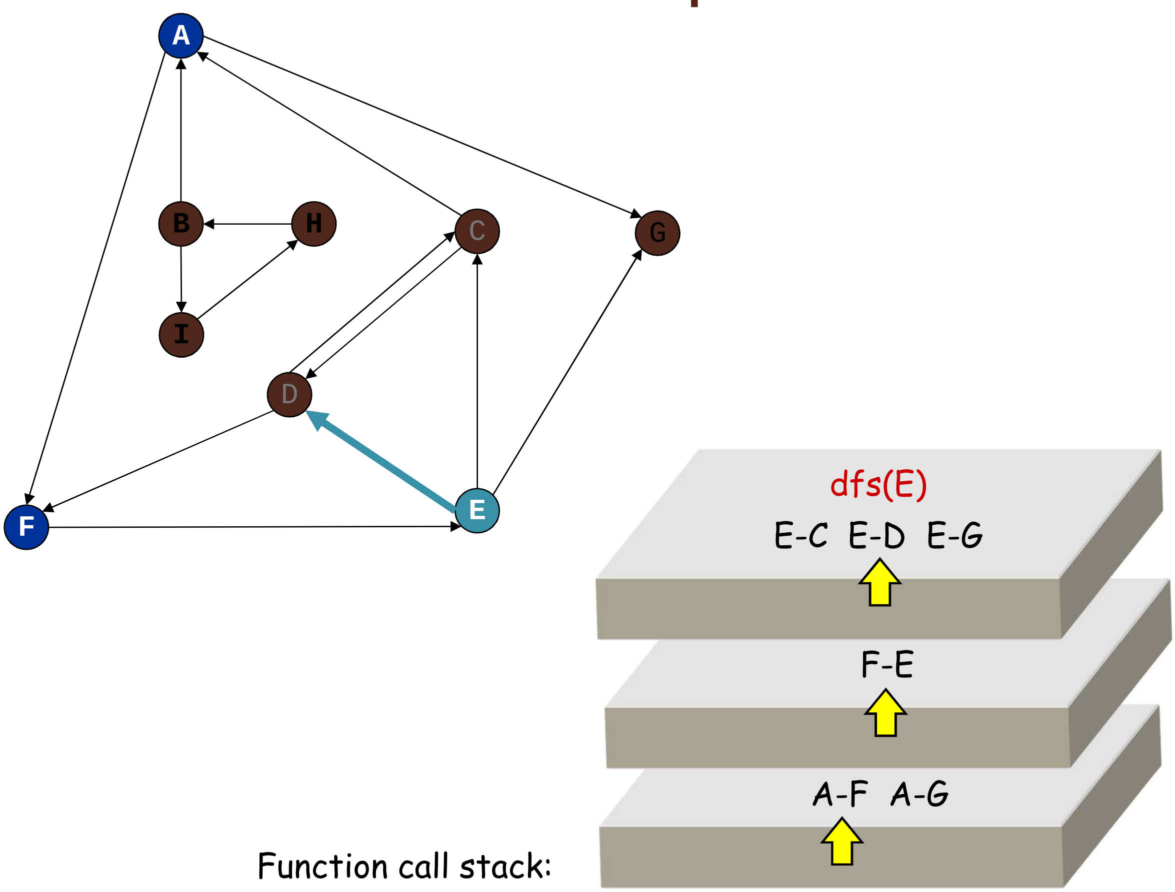
Directed Depth First Search



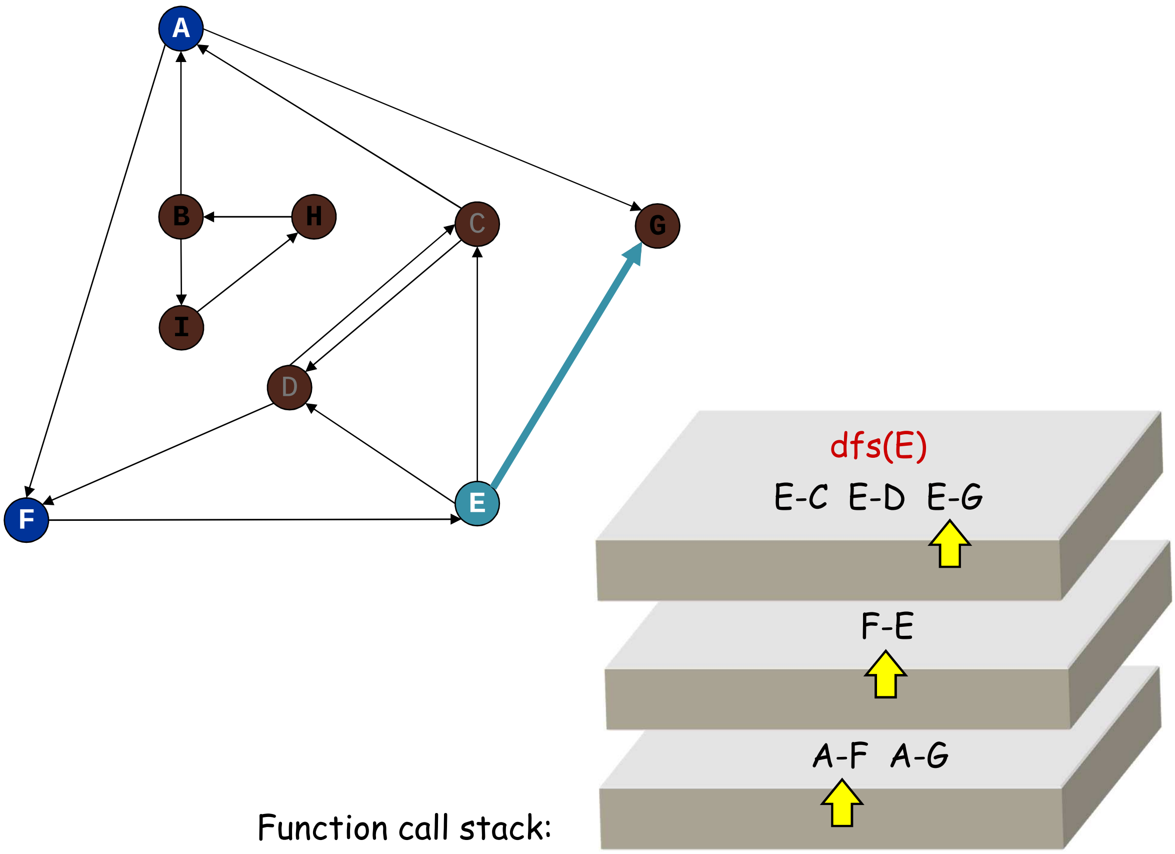
Directed Depth First Search



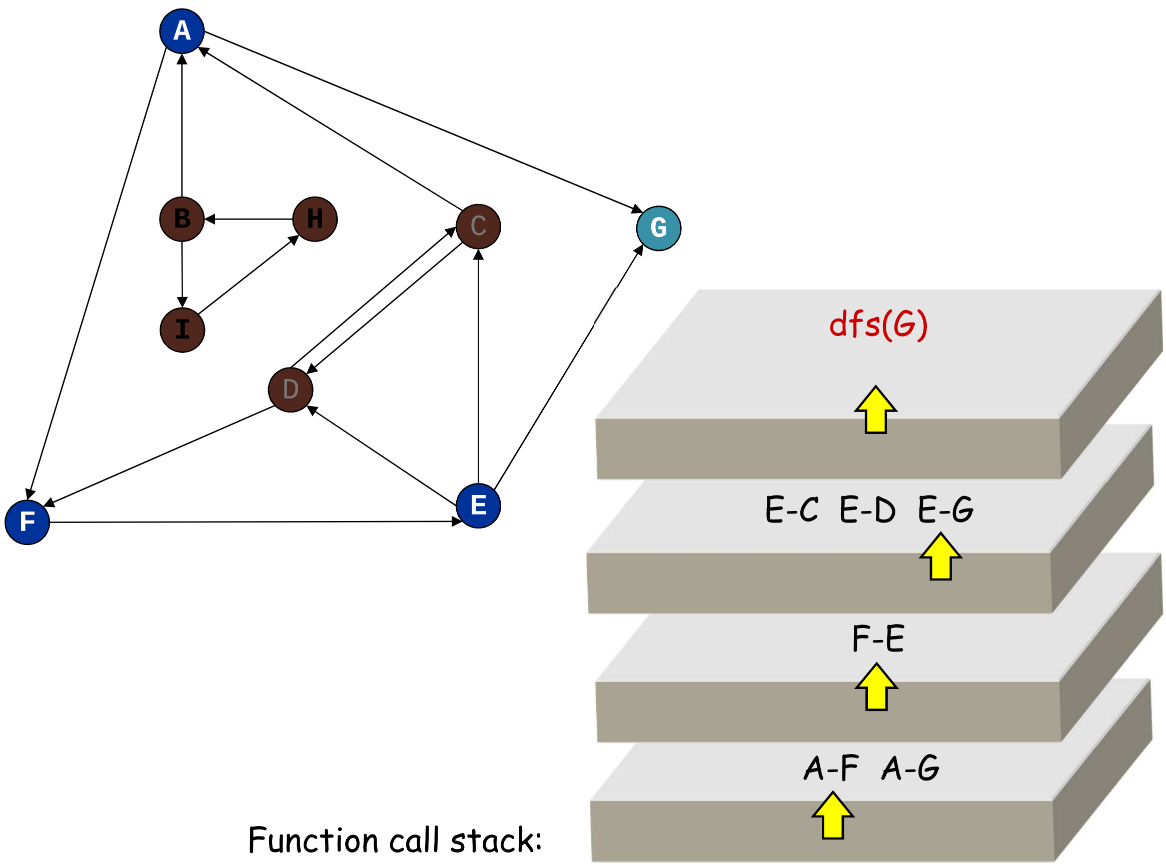
Directed Depth First Search



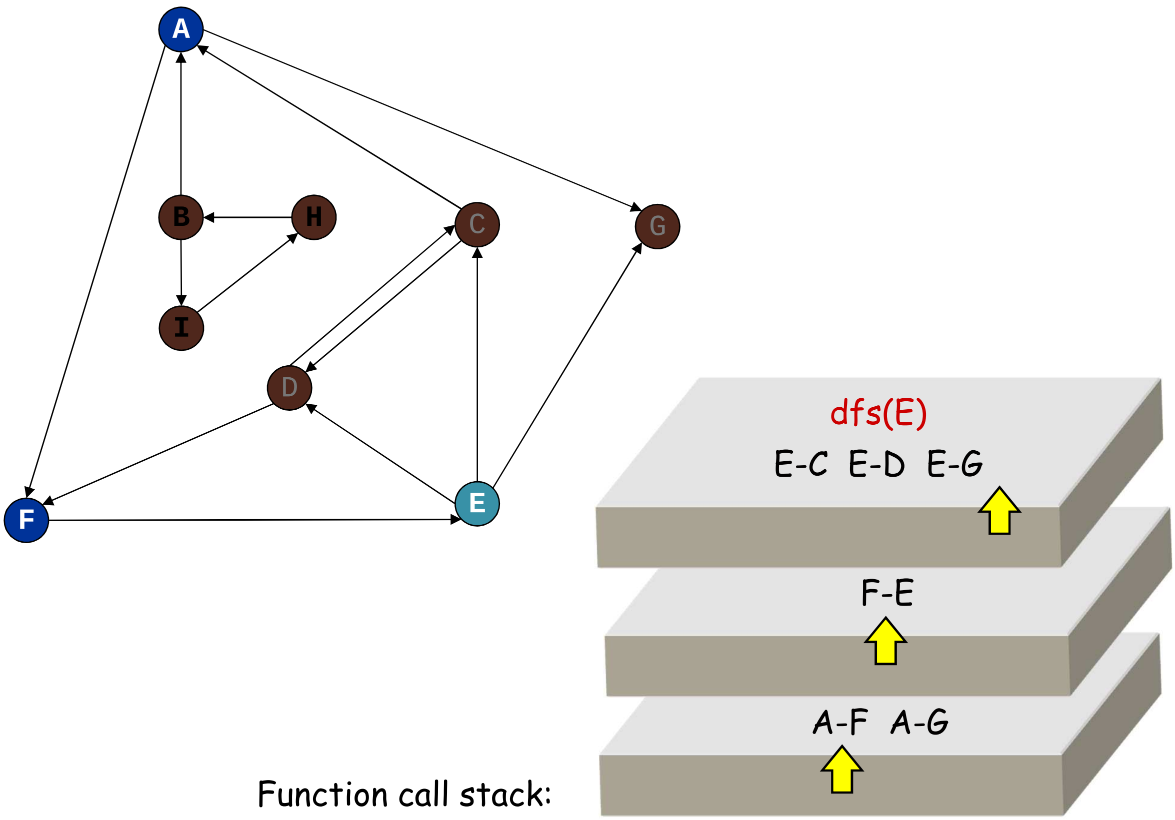
Directed Depth First Search



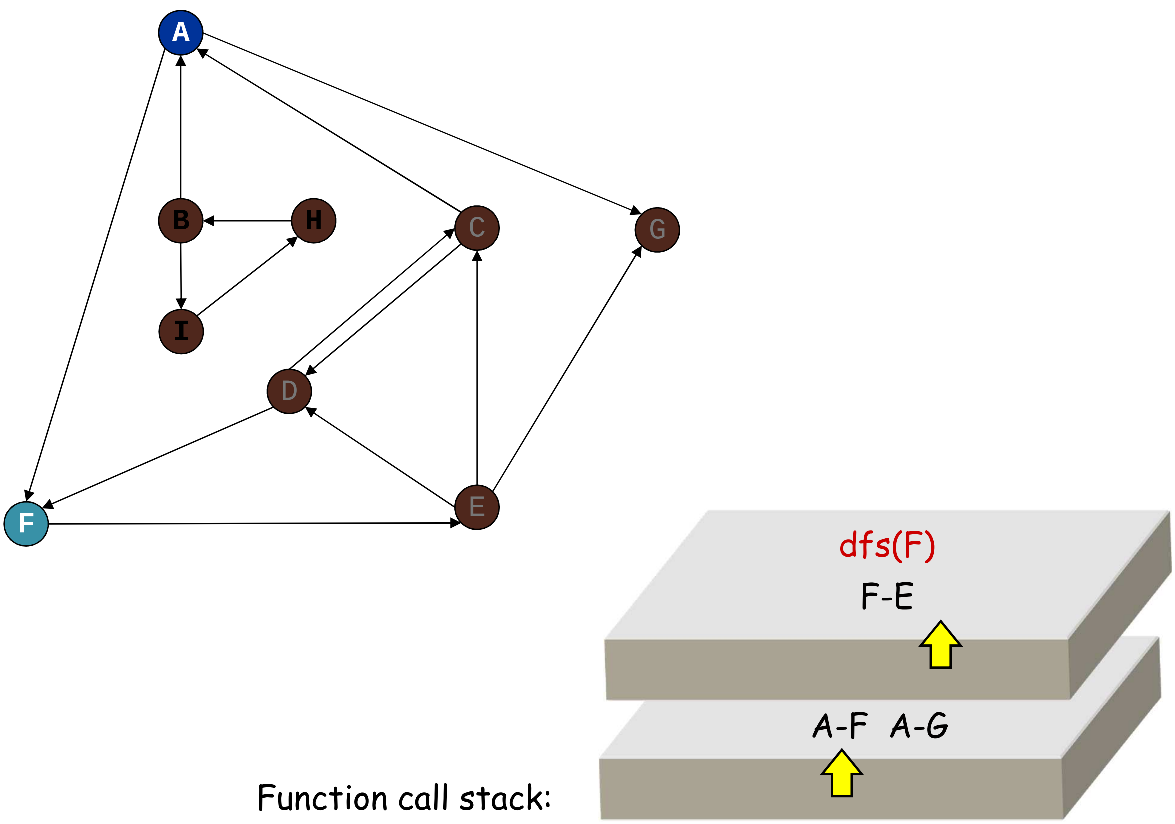
Directed Depth First Search



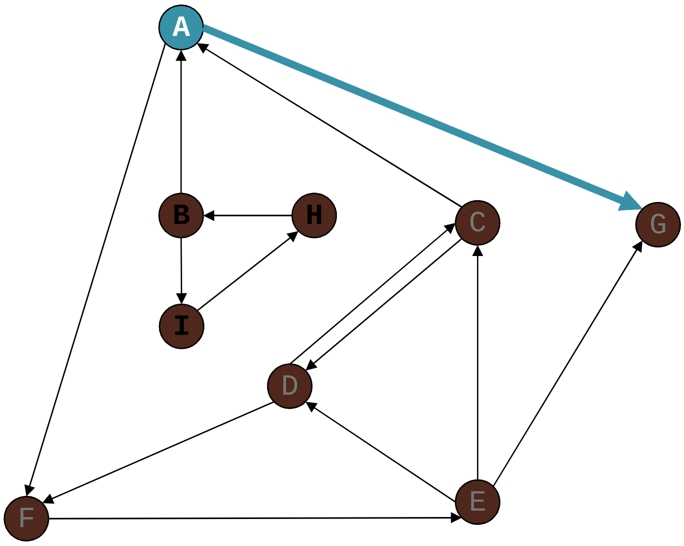
Directed Depth First Search



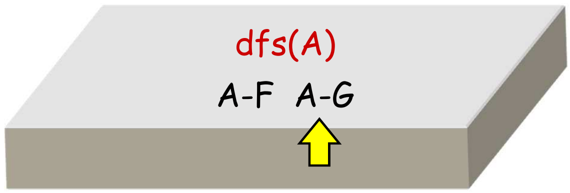
Directed Depth First Search



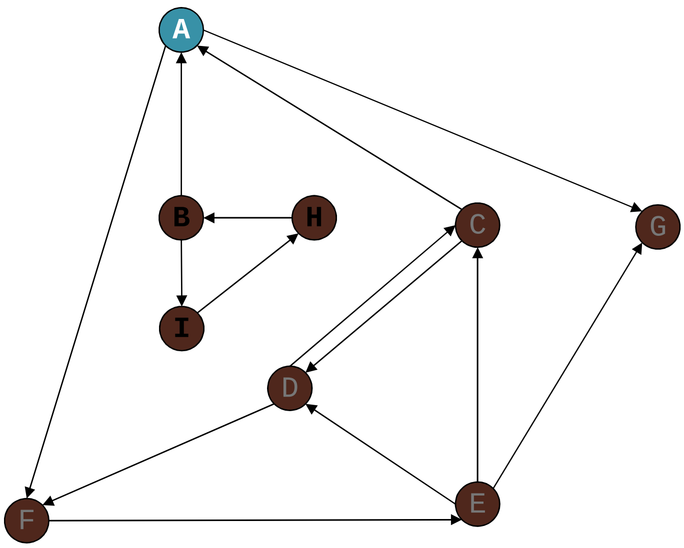
Directed Depth First Search



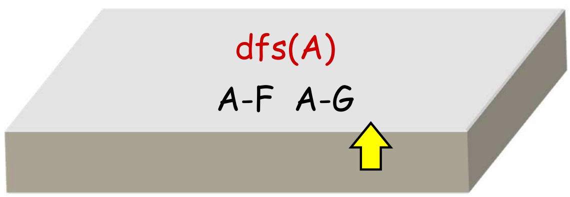
Function call stack:



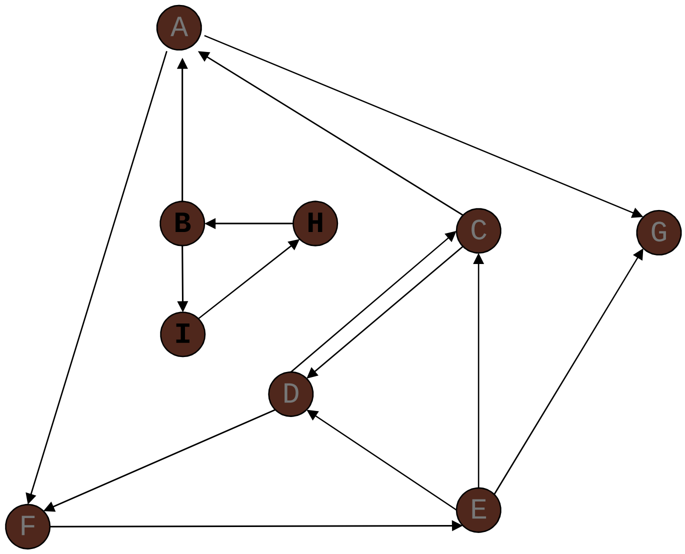
Directed Depth First Search



Function call stack:

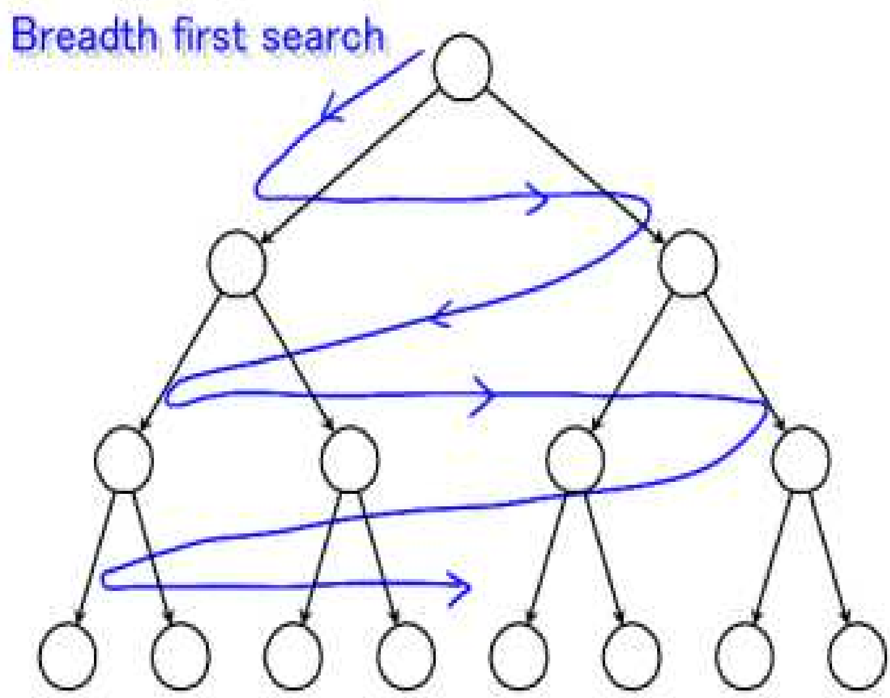


Directed Depth First Search



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

Breadth First Search



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
  - Computes a spanning forest 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

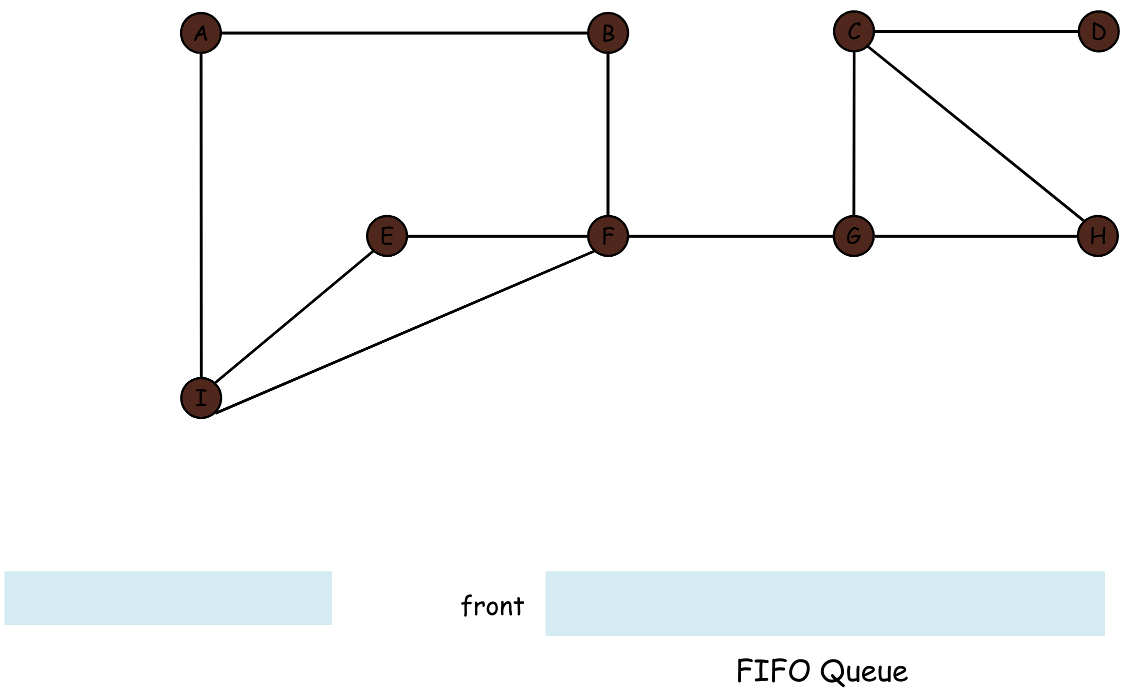
- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

```
Algorithm BFS(G)
    Input graph G
    Output labeling of the edges
        and partition of the
        vertices of G
  for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
        BFS(G, v)
```

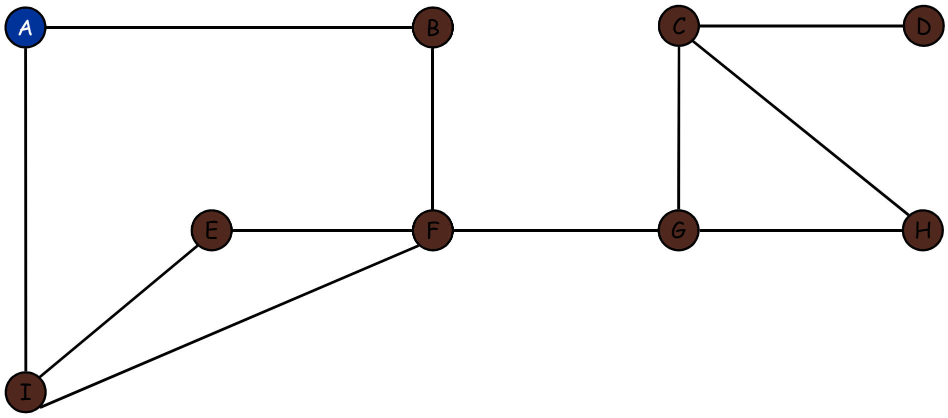
```
Algorithm BFS(G, s)
  L0 ← new empty sequence
  L0.insertLast(s)
  setLabel(s, VISITED)
  i ← 0
  while ¬Li.isEmpty()
    Li+1 ← new empty sequence
    for all v ∈ Li.elements()
      for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
          w ← opposite(v,e)
          if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            setLabel(w, VISITED)
            Li+1.insertLast(w)
          else
            setLabel(e, CROSS)
    i ← i +1
```

- Expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.
  - A vertex is "discovered" the first time it is encountered during the search.
  - A vertex is "finished" if all vertices adjacent to it have been discovered.
- Colors the vertices to keep track of progress.
  - White – Undiscovered.
  - Gray – Discovered but not finished.
  - Black – Finished.
    - Colors are required only to reason about the algorithm. Can be implemented without colors.

Breadth First Search



Breadth First Search

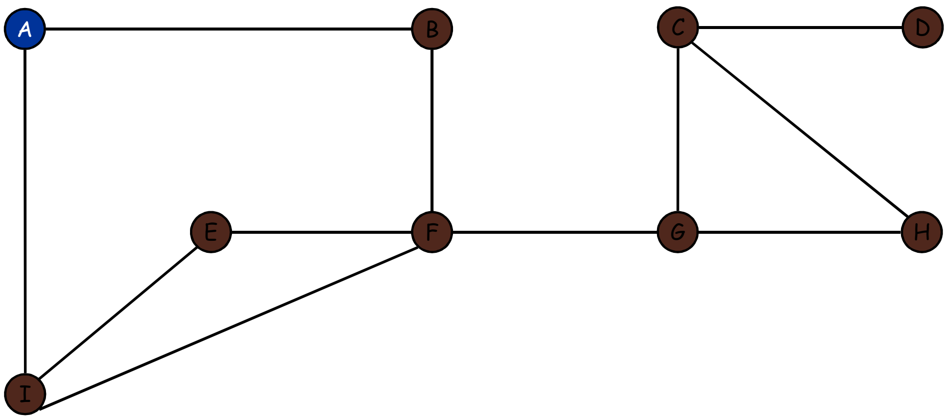


enqueue source node

front **A**

FIFO Queue

Breadth First Search

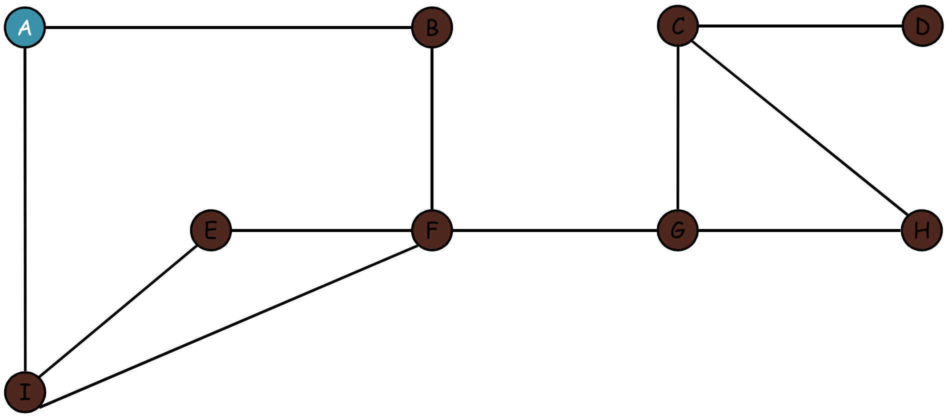


dequeue next vertex

front **A**

FIFO Queue

Breadth First Search

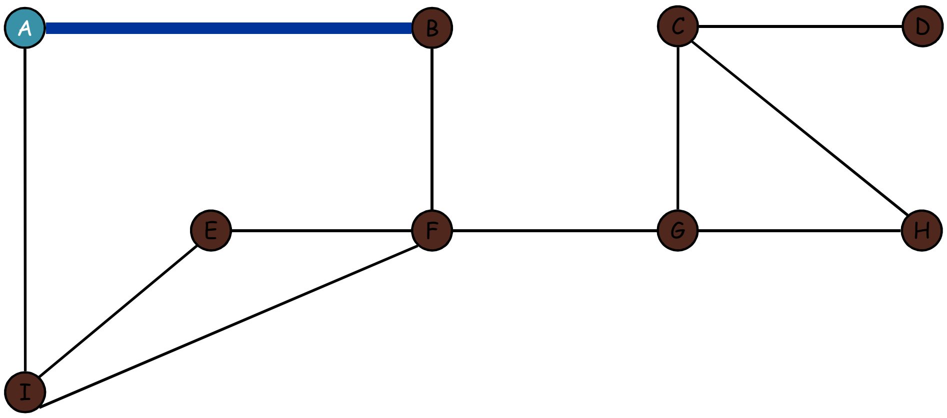


visit neighbors of A

front

FIFO Queue

Breadth First Search

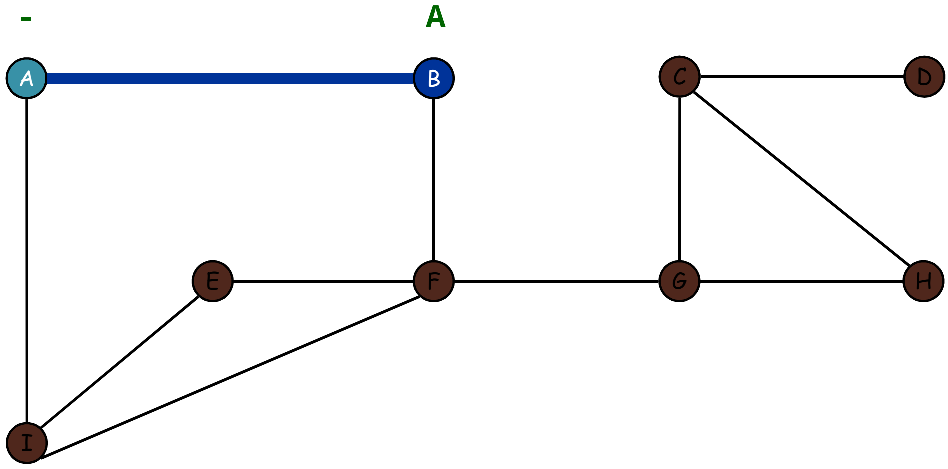


visit neighbors of A

front

FIFO Queue

Breadth First Search

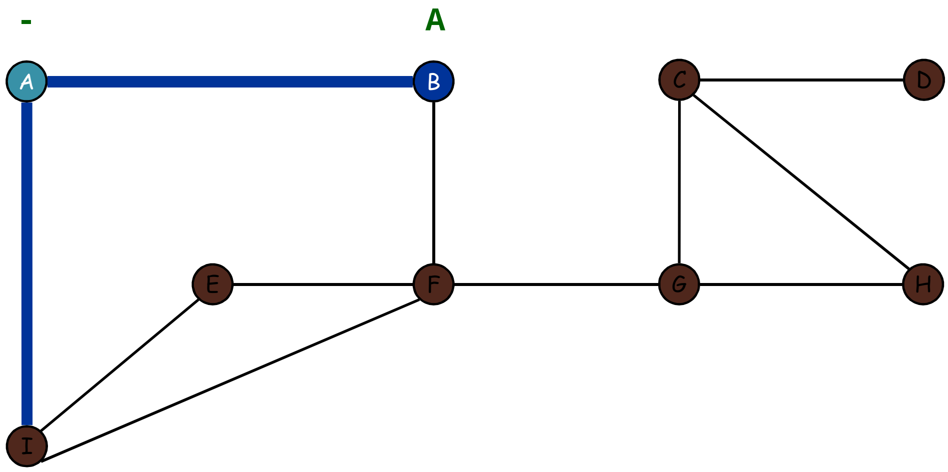


B discovered

front **B**

FIFO Queue

Breadth First Search

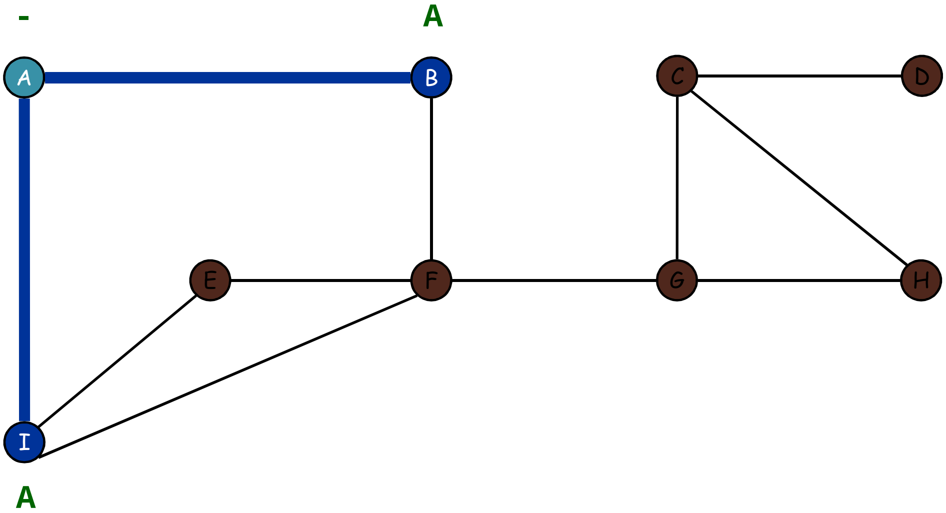


visit neighbors of A

front **B**

FIFO Queue

Breadth First Search

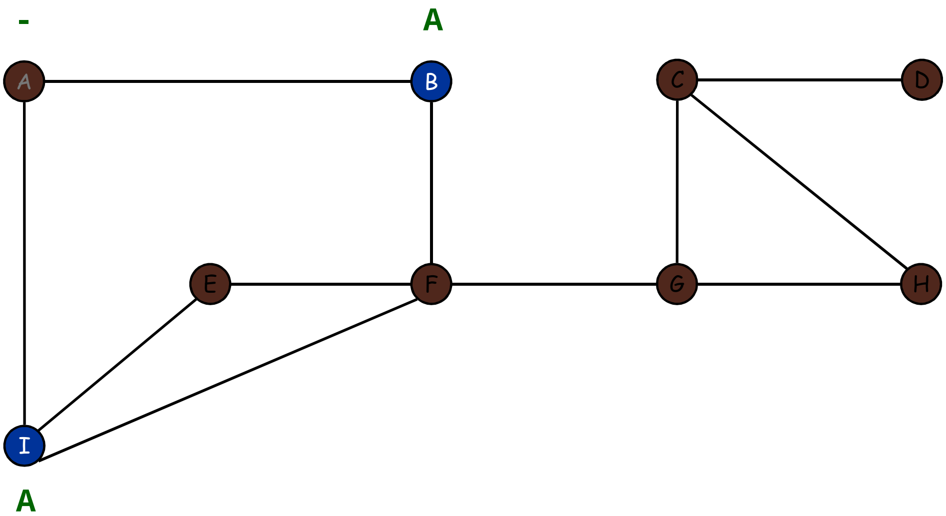


I discovered

front **B I**

FIFO Queue

Breadth First Search

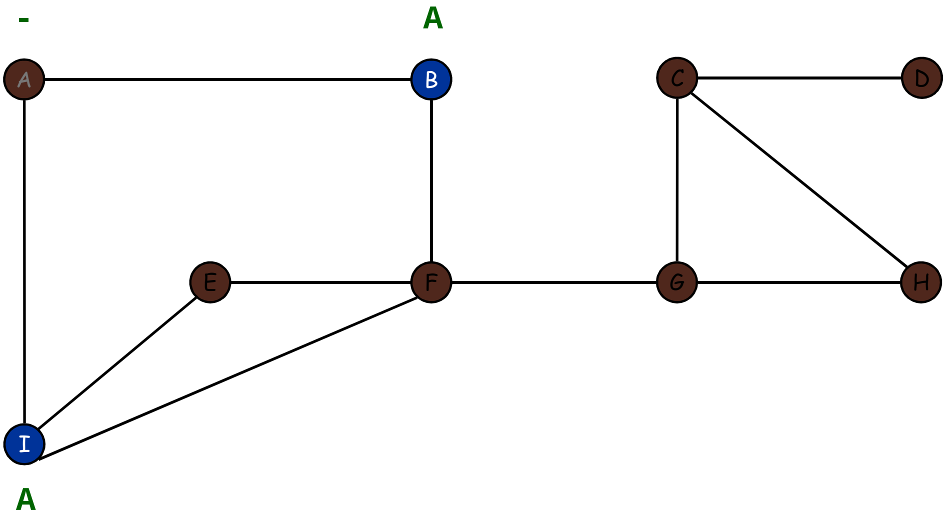


finished with A

front **B I**

FIFO Queue

Breadth First Search

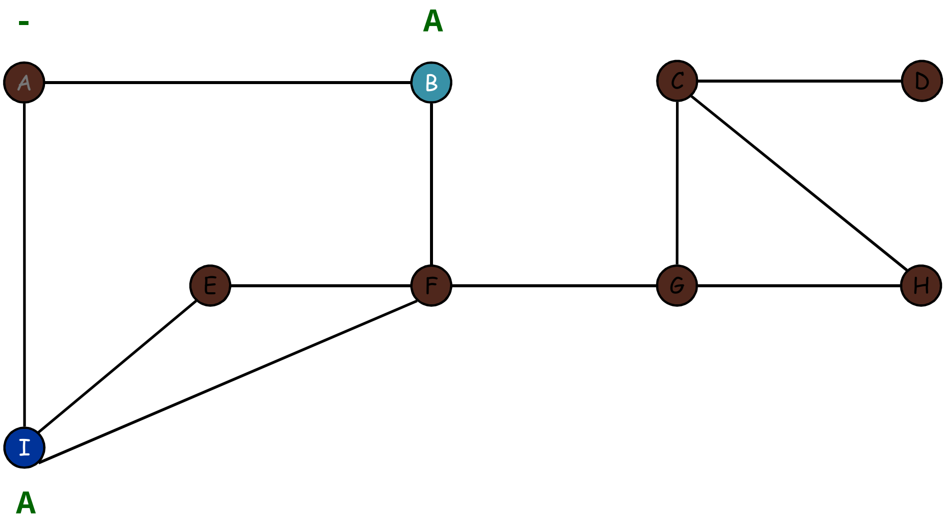


dequeue next vertex

front **B I**

FIFO Queue

Breadth First Search

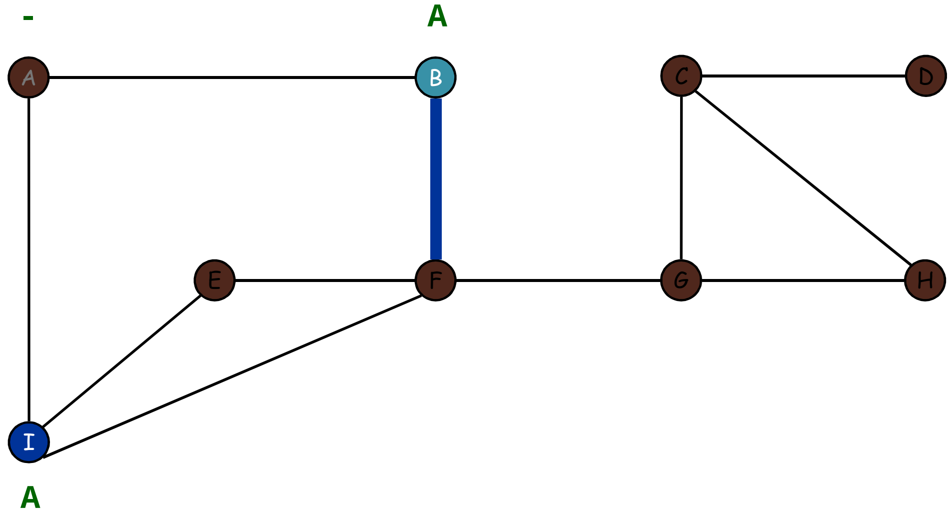


visit neighbors of B

front **I**

FIFO Queue

Breadth First Search

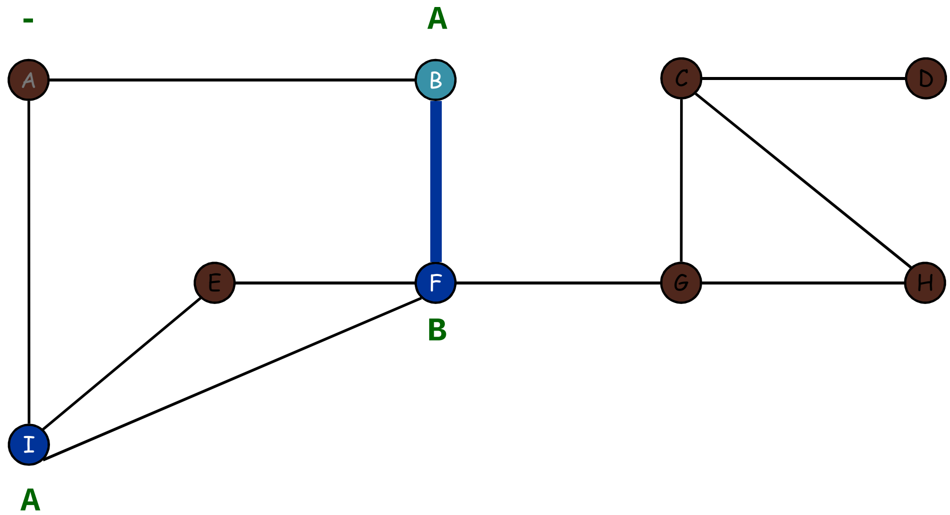


visit neighbors of B

front I

FIFO Queue

Breadth First Search

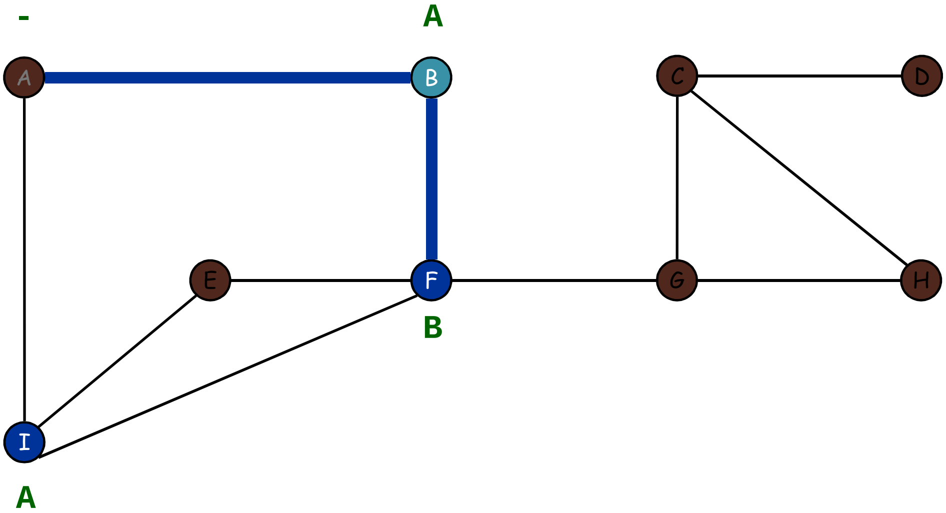


F discovered

front I F

FIFO Queue

Breadth First Search

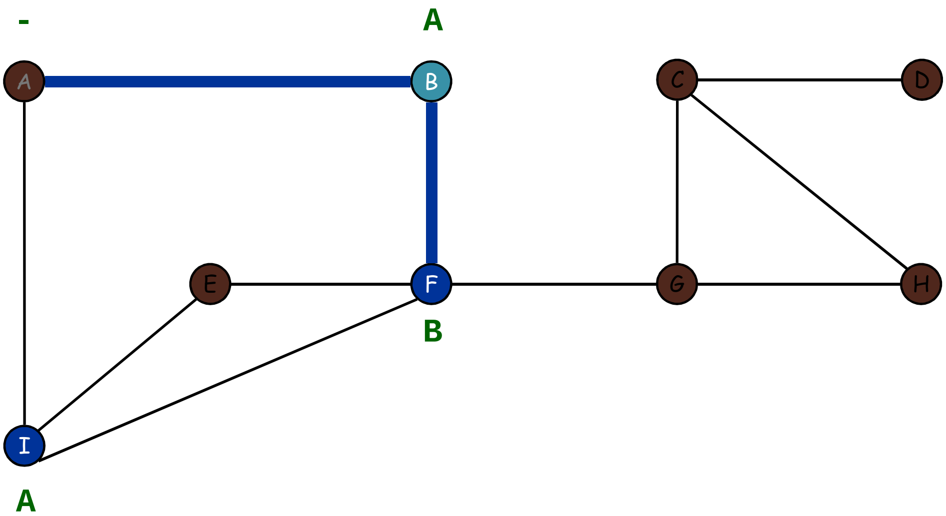


visit neighbors of B

front | **I F**

FIFO Queue

Breadth First Search

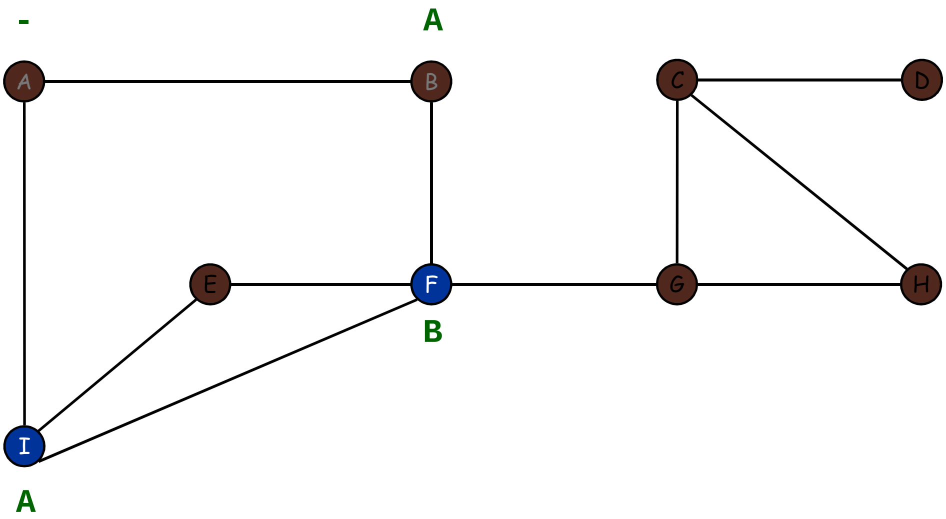


A already discovered

front | **I F**

FIFO Queue

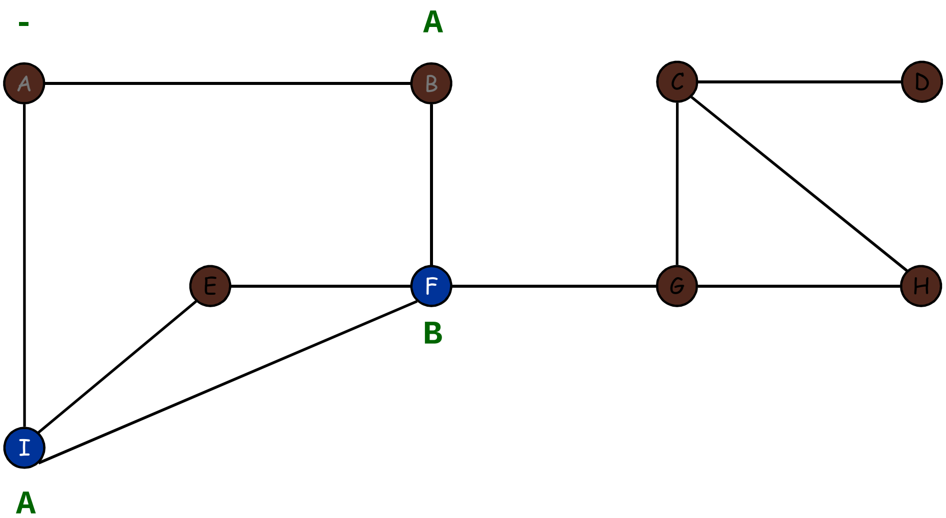
Breadth First Search



finished with B | front | I F

FIFO Queue

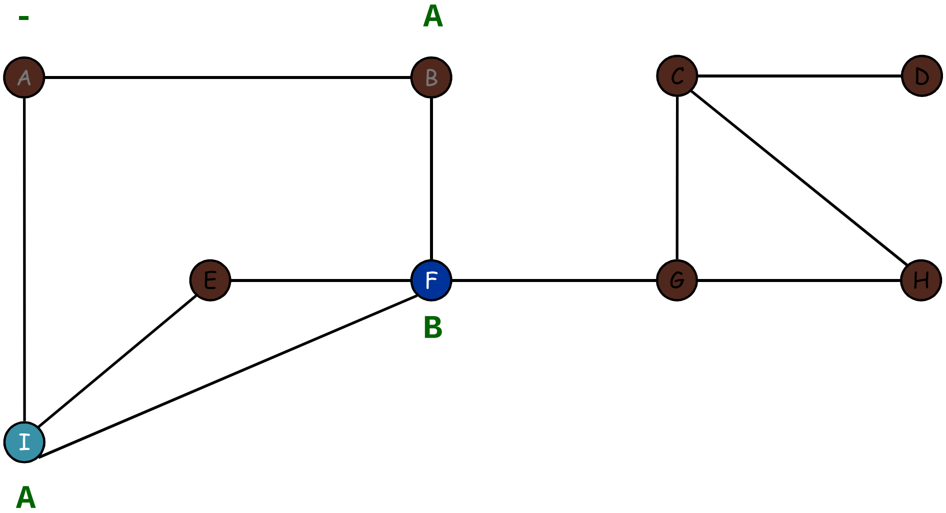
Breadth First Search



dequeue next vertex | front | I F

FIFO Queue

Breadth First Search

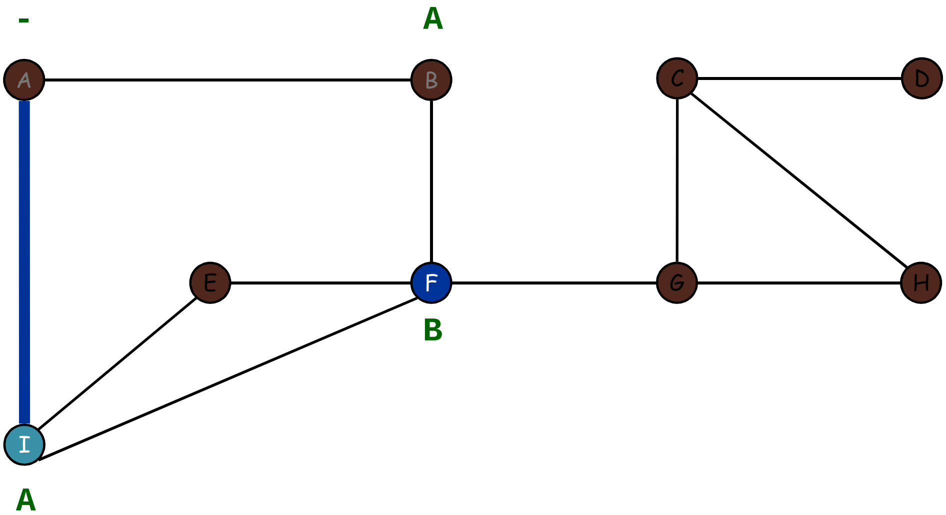


visit neighbors of I

front **F**

FIFO Queue

Breadth First Search

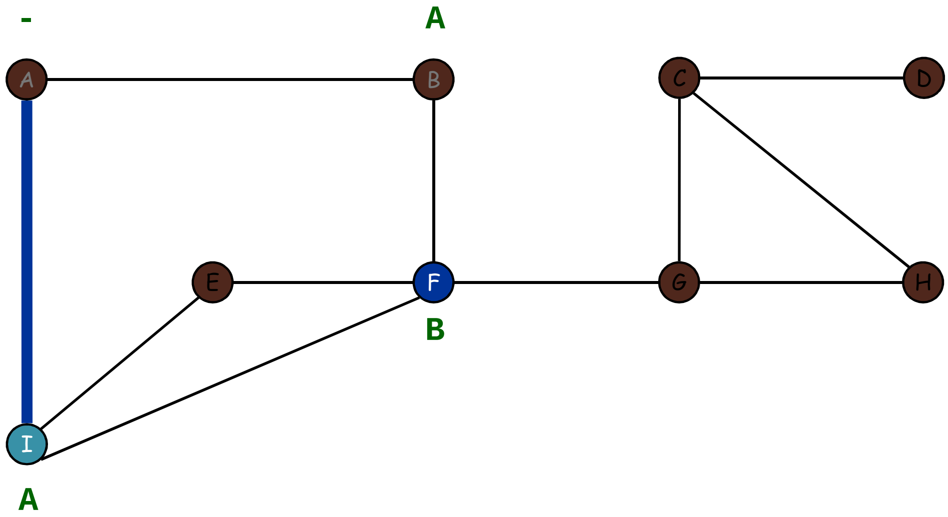


visit neighbors of I

front **F**

FIFO Queue

Breadth First Search

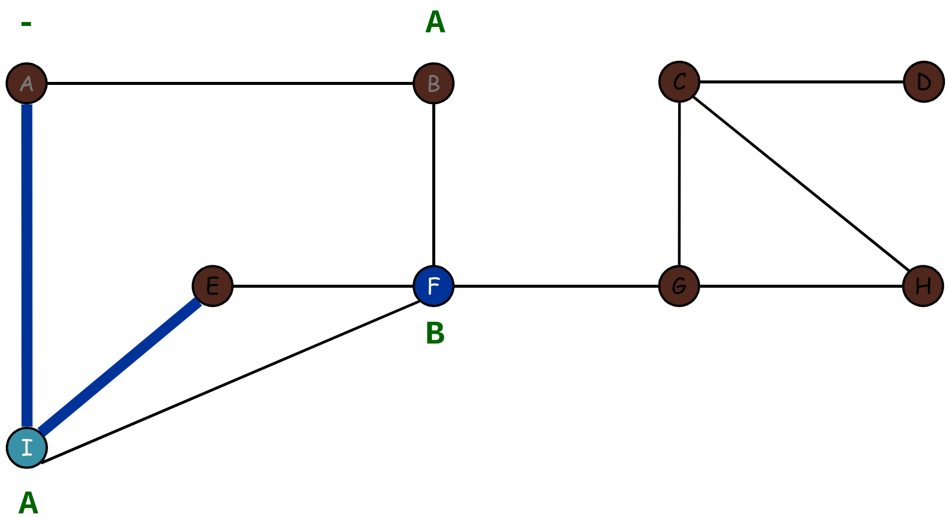


A already discovered

front F

FIFO Queue

Breadth First Search

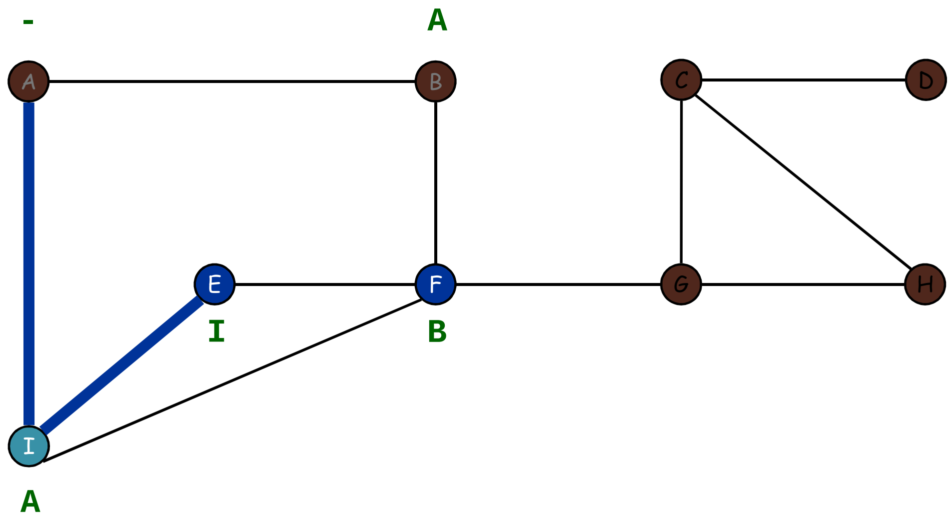


visit neighbors of I

front F

FIFO Queue

Breadth First Search

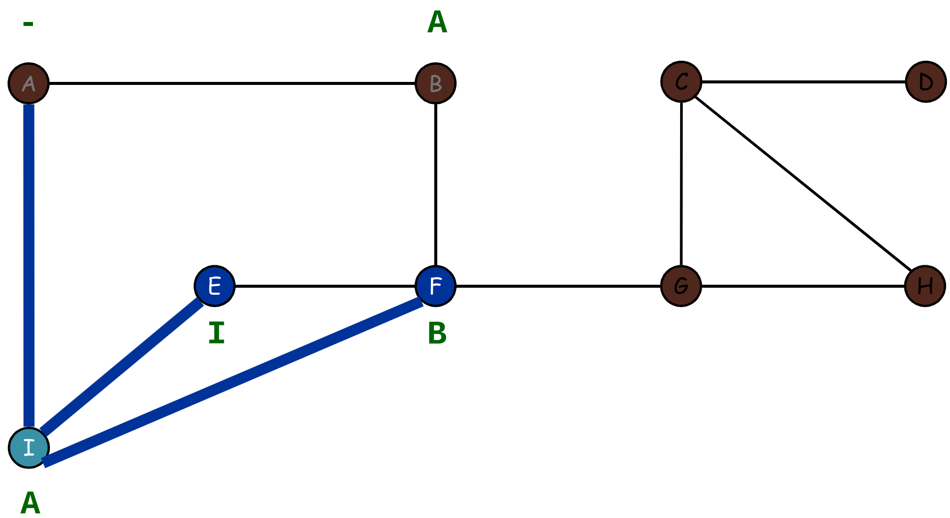


E discovered

front **F E**

FIFO Queue

Breadth First Search

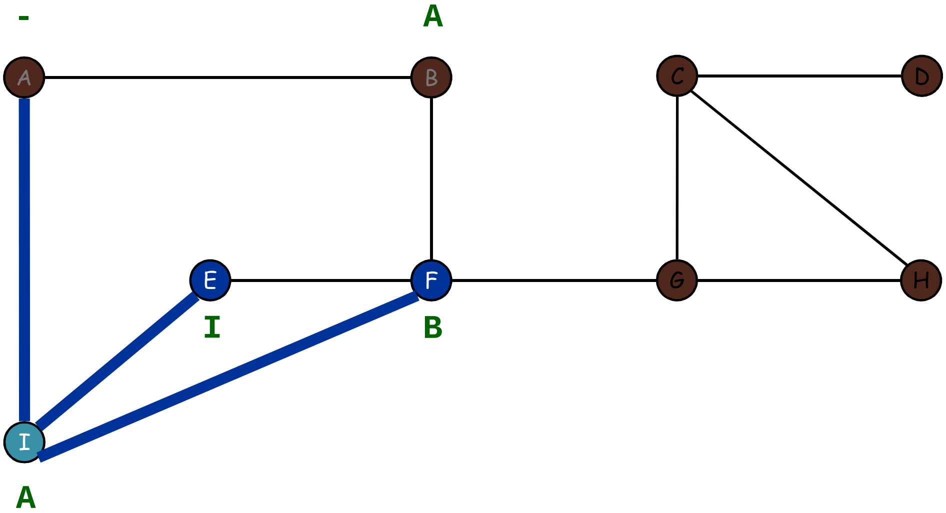


visit neighbors of I

front **F E**

FIFO Queue

Breadth First Search



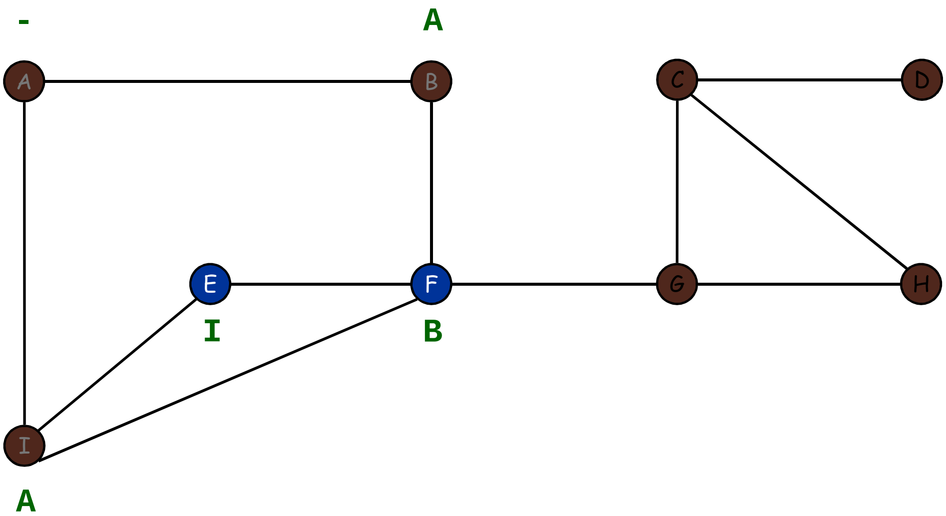
F already discovered

front

F E

FIFO Queue

Breadth First Search



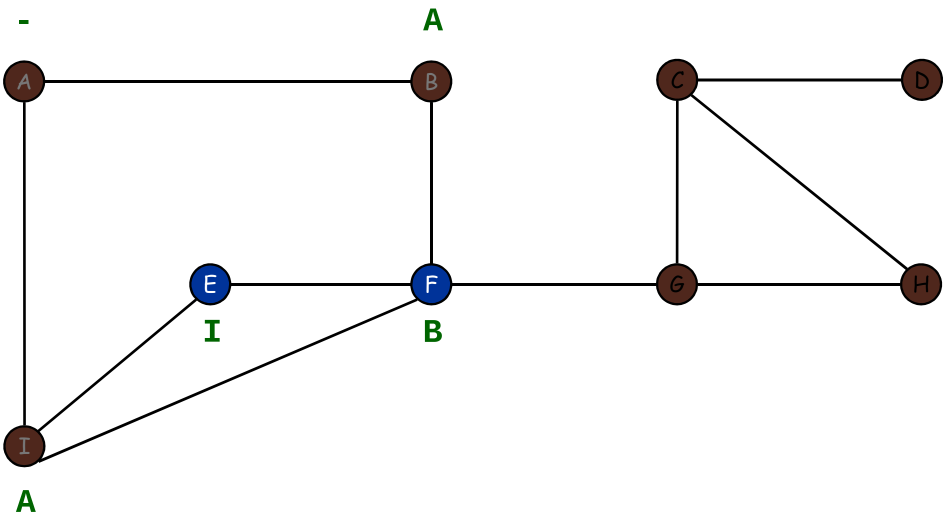
I finished

front

F E

FIFO Queue

Breadth First Search

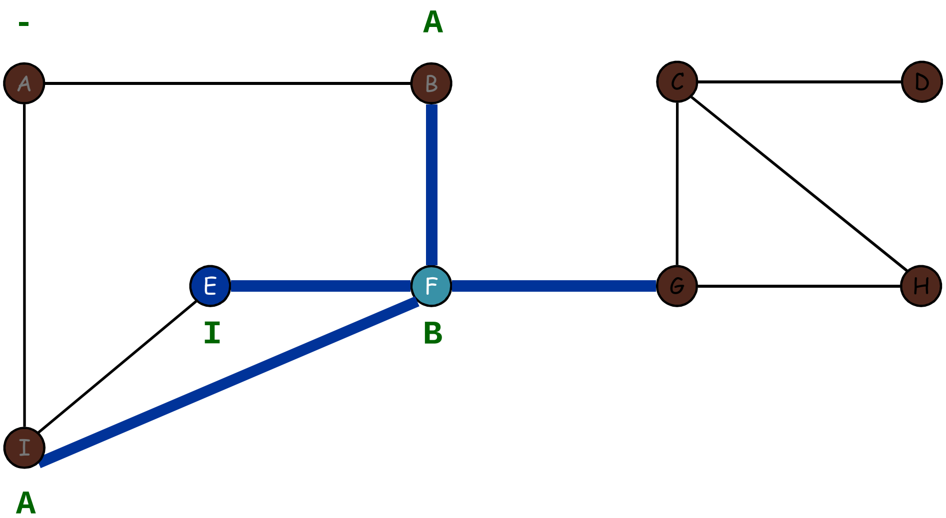


dequeue next vertex

front F E

FIFO Queue

Breadth First Search

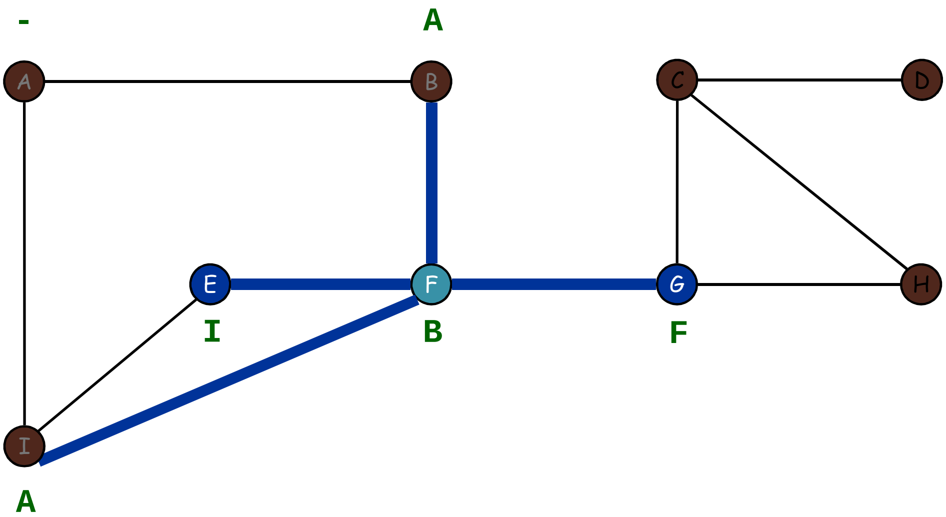


visit neighbors of F

front E

FIFO Queue

Breadth First Search



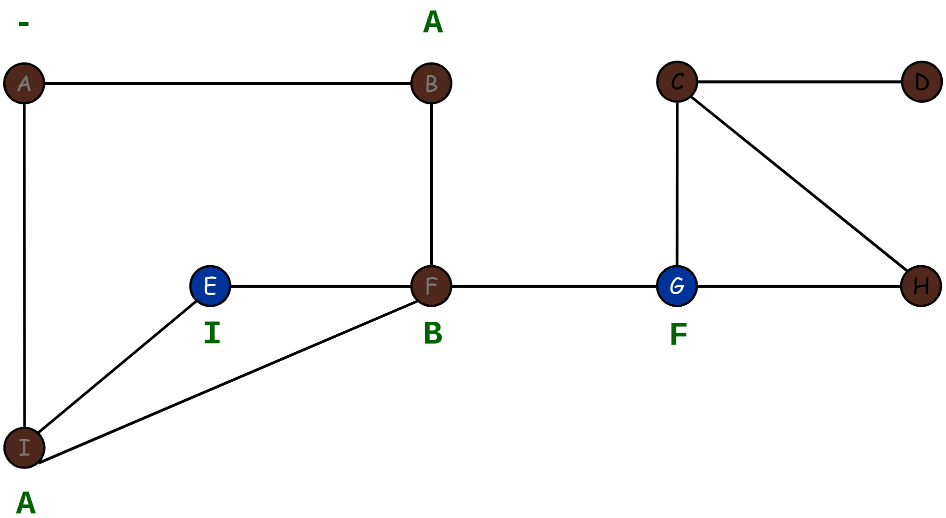
G discovered

front

E G

FIFO Queue

Breadth First Search



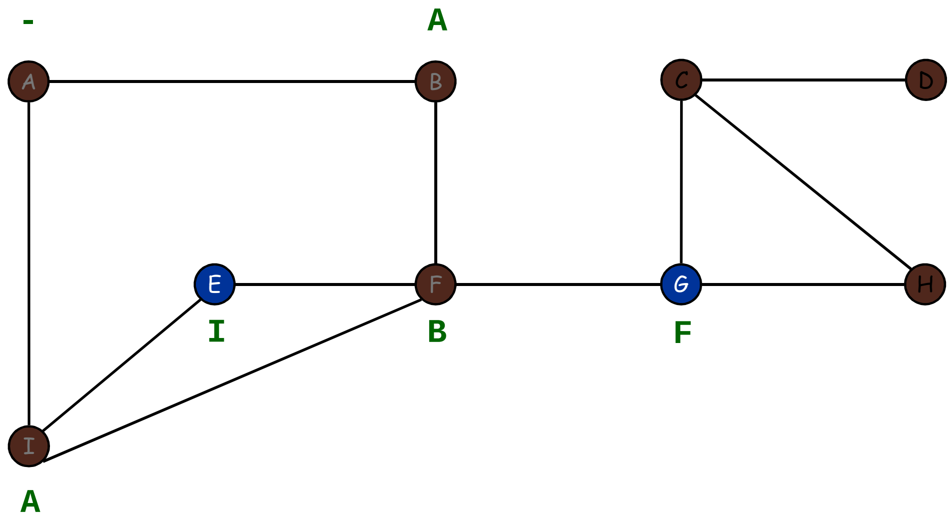
F finished

front

E G

FIFO Queue

Breadth First Search

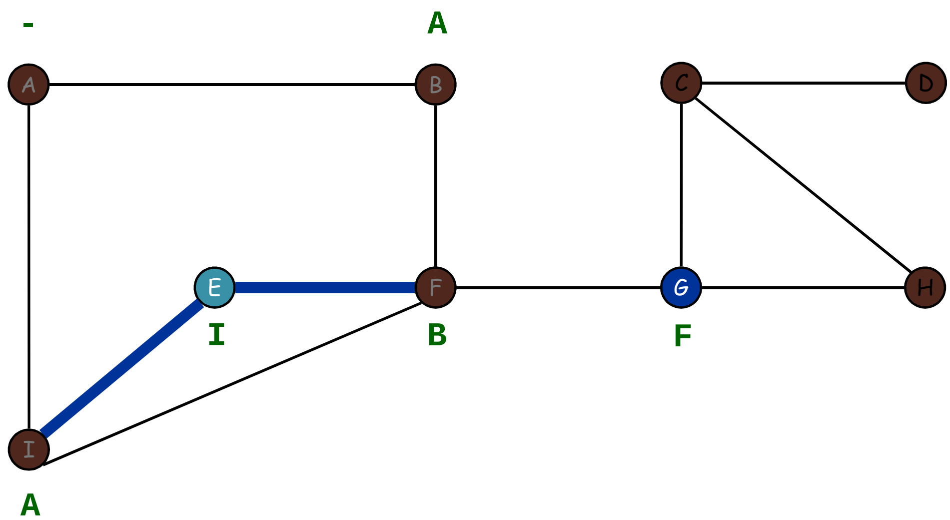


dequeue next vertex

front E G

FIFO Queue

Breadth First Search

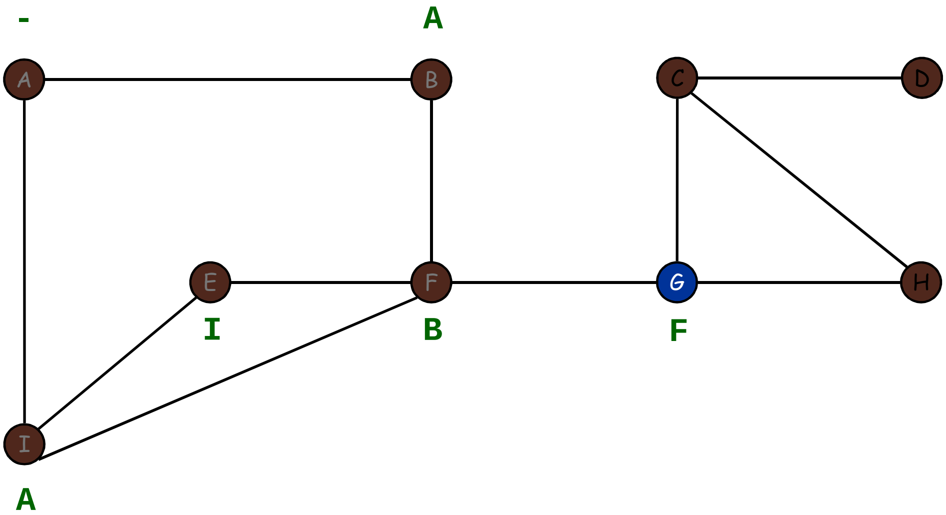


visit neighbors of E

front G

FIFO Queue

Breadth First Search



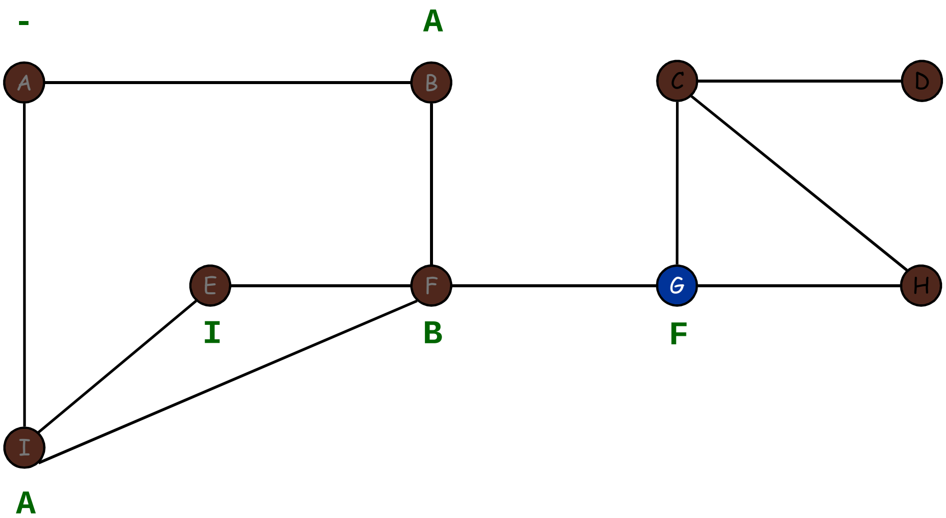
E finished

front

G

FIFO Queue

Breadth First Search



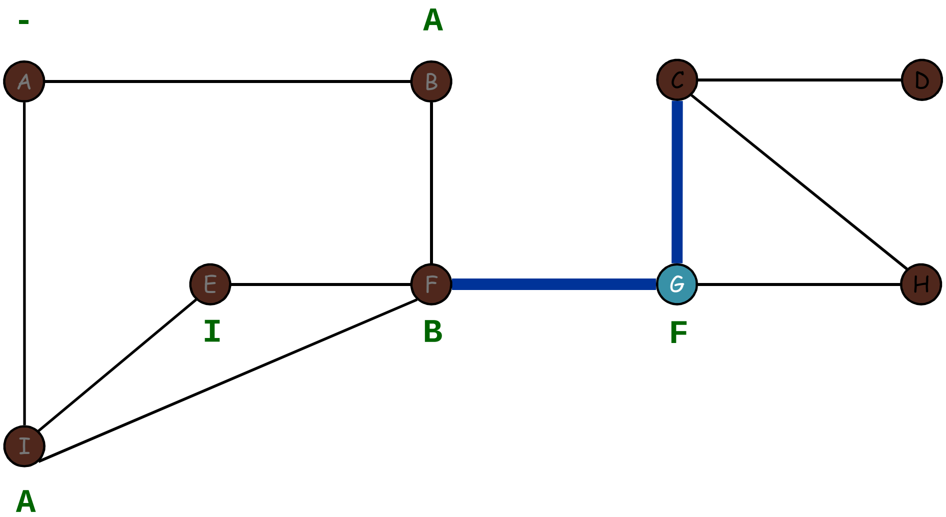
dequeue next vertex

front

G

FIFO Queue

Breadth First Search

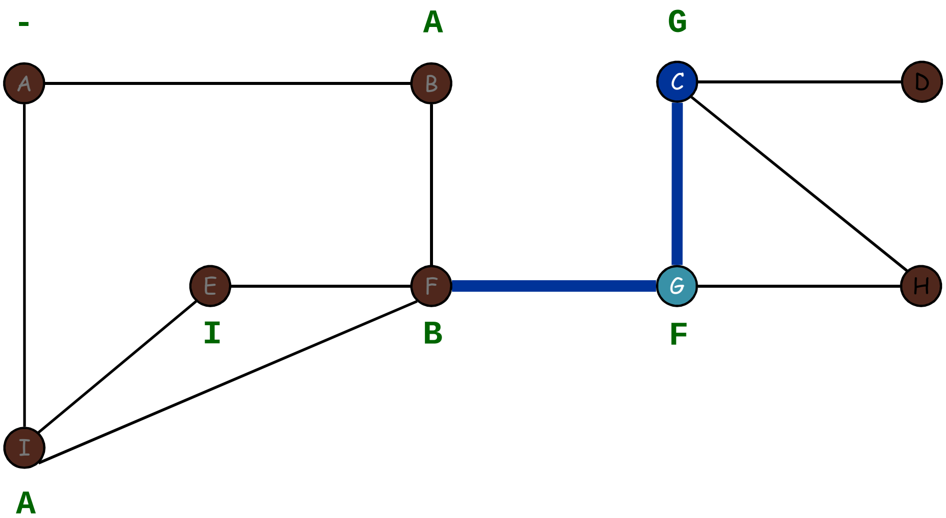


visit neighbors of G

front

FIFO Queue

Breadth First Search

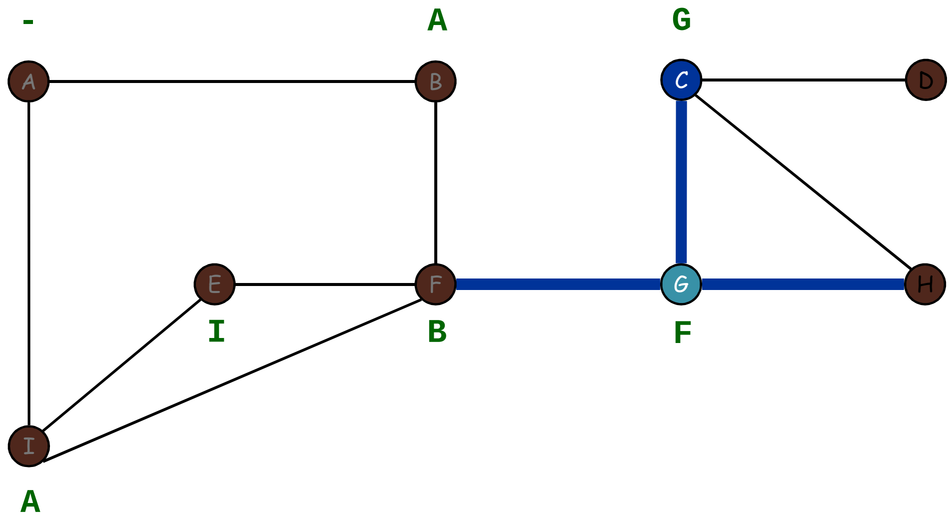


C discovered

front **C**

FIFO Queue

Breadth First Search



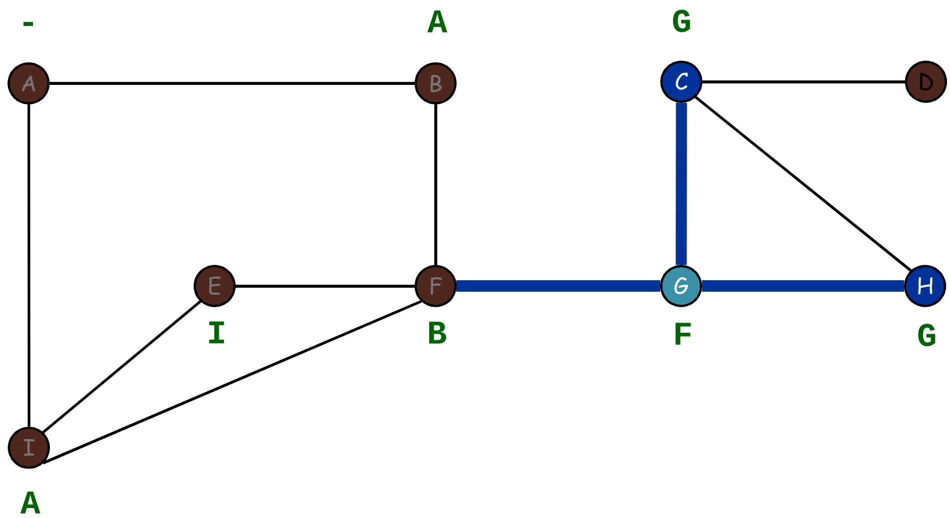
visit neighbors of G

front

C

FIFO Queue

Breadth First Search



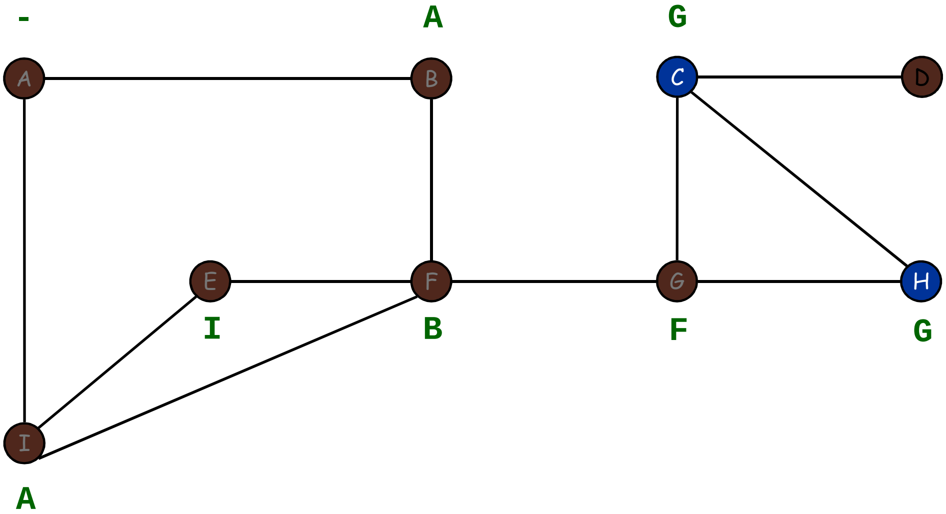
H discovered

front

C H

FIFO Queue

Breadth First Search



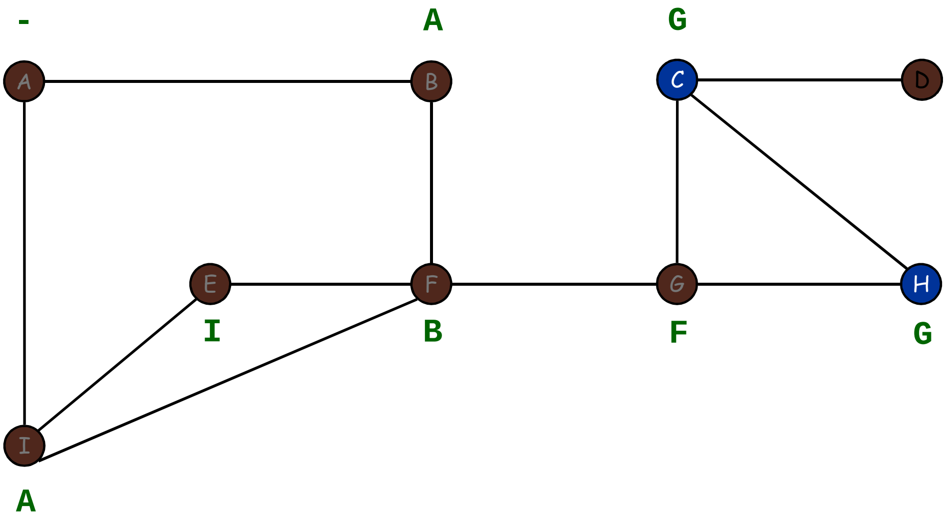
G finished

front

C H

FIFO Queue

Breadth First Search



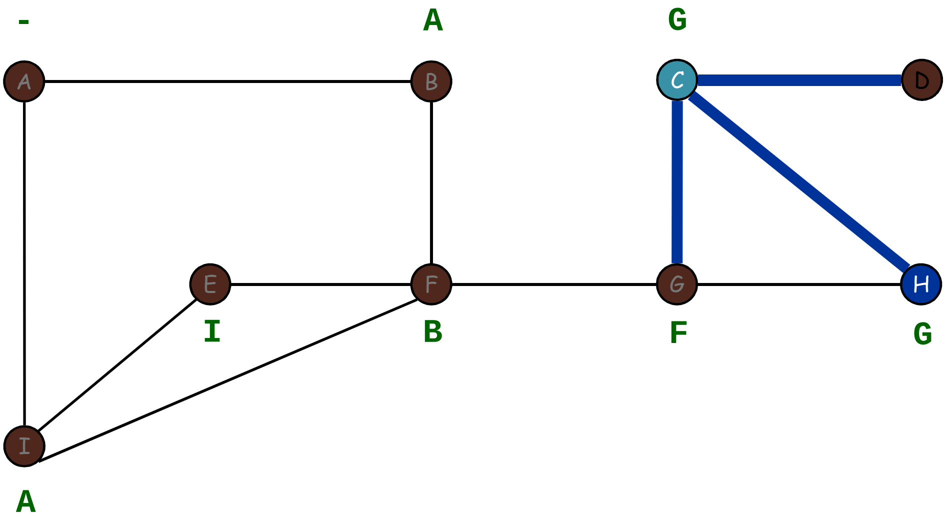
dequeue next vertex

front

C H

FIFO Queue

Breadth First Search

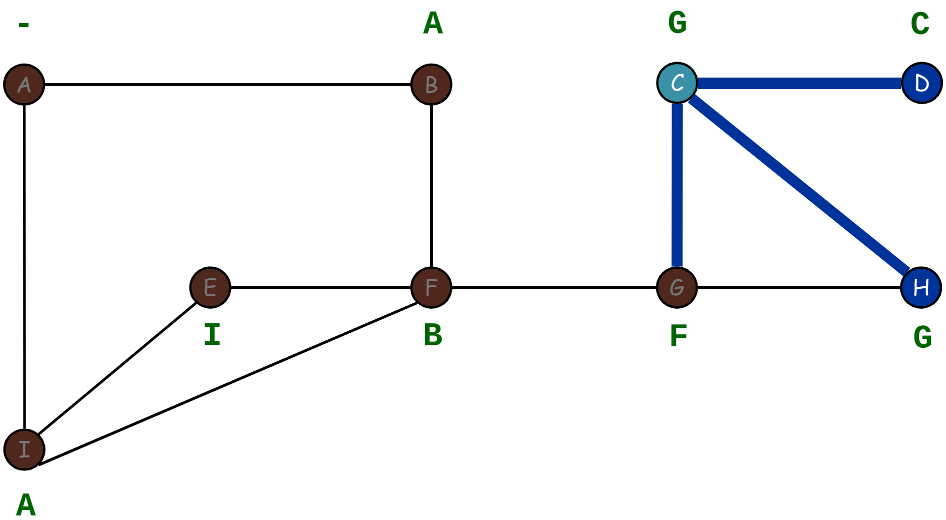


visit neighbors of C

front **H**

FIFO Queue

Breadth First Search

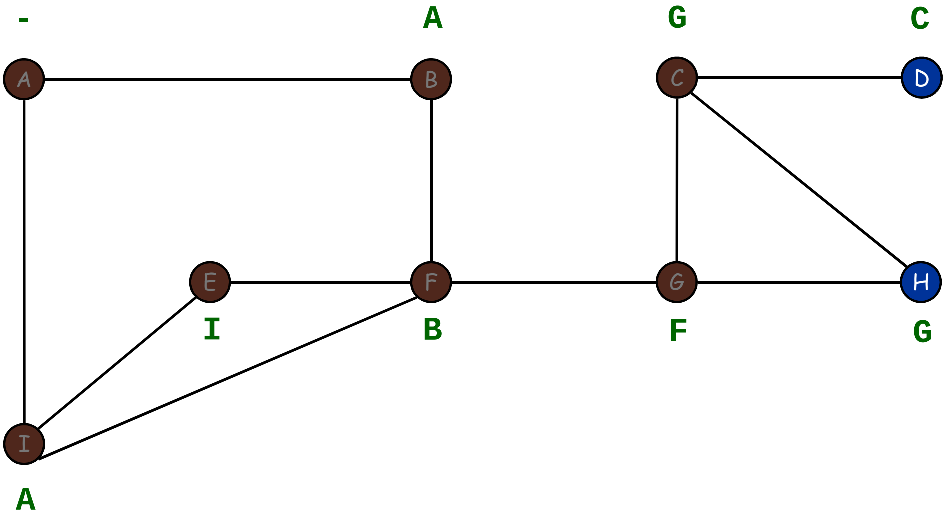


D discovered

front **H D**

FIFO Queue

Breadth First Search



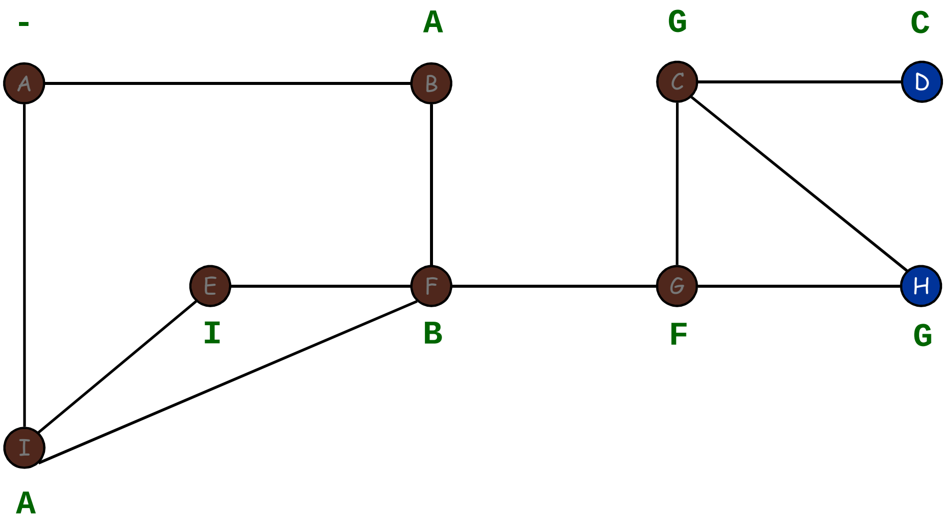
C finished

front

H D

FIFO Queue

Breadth First Search



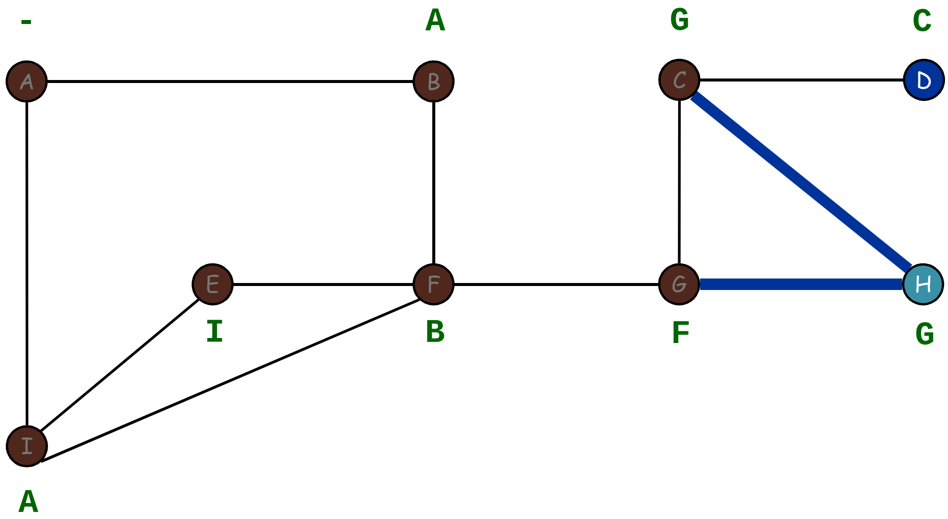
get next vertex

front

H D

FIFO Queue

Breadth First Search



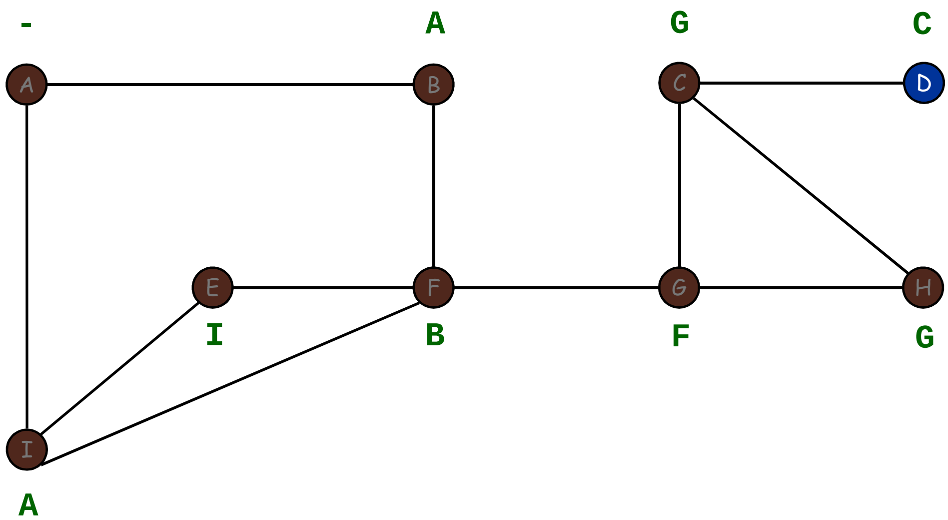
visit neighbors of H

front

D

FIFO Queue

Breadth First Search



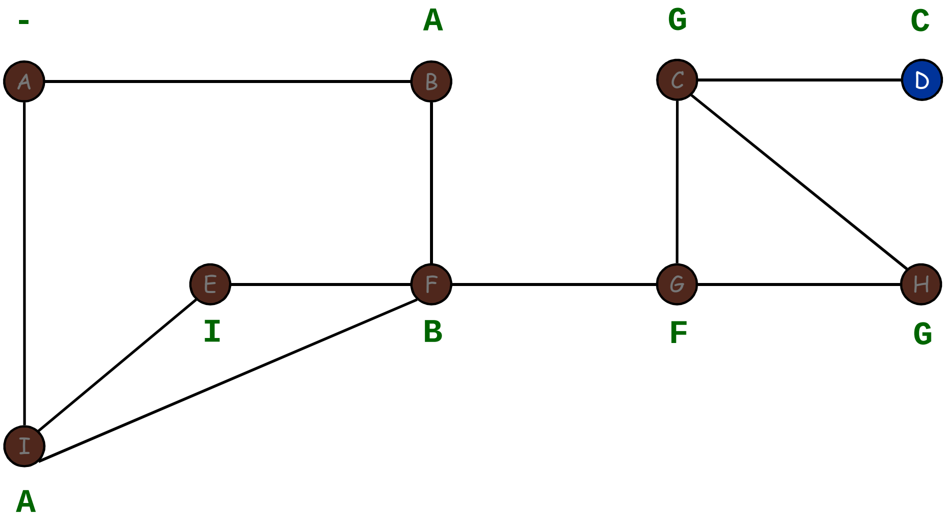
finished H

front

D

FIFO Queue

Breadth First Search

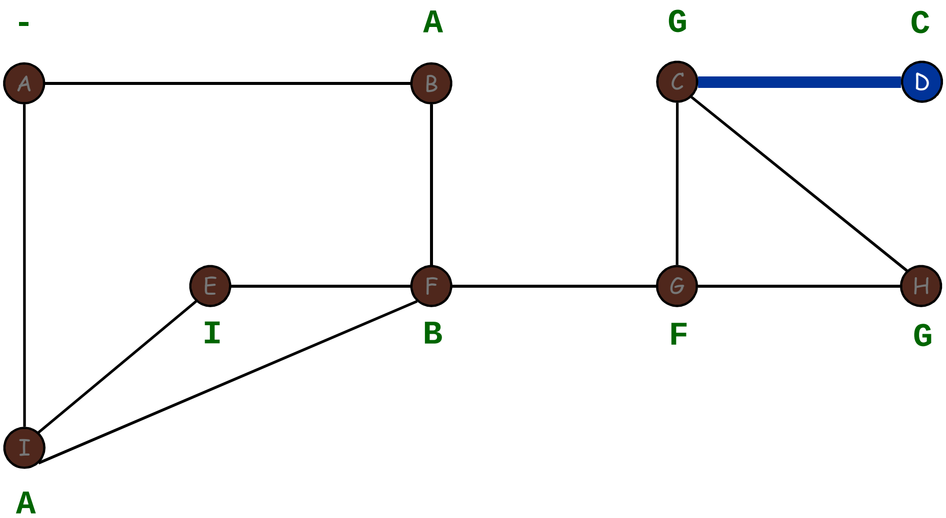


dequeue next vertex

front D

FIFO Queue

Breadth First Search

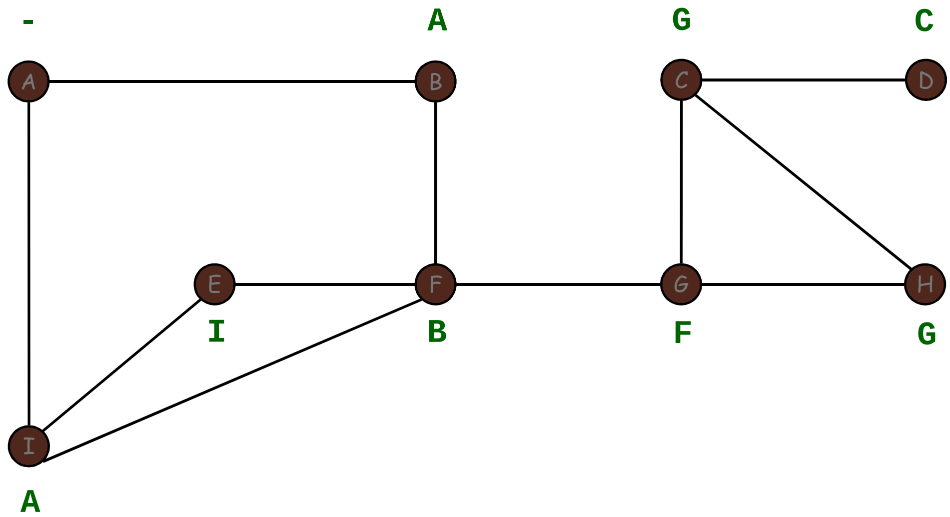


visit neighbors of D

front

FIFO Queue

Breadth First Search

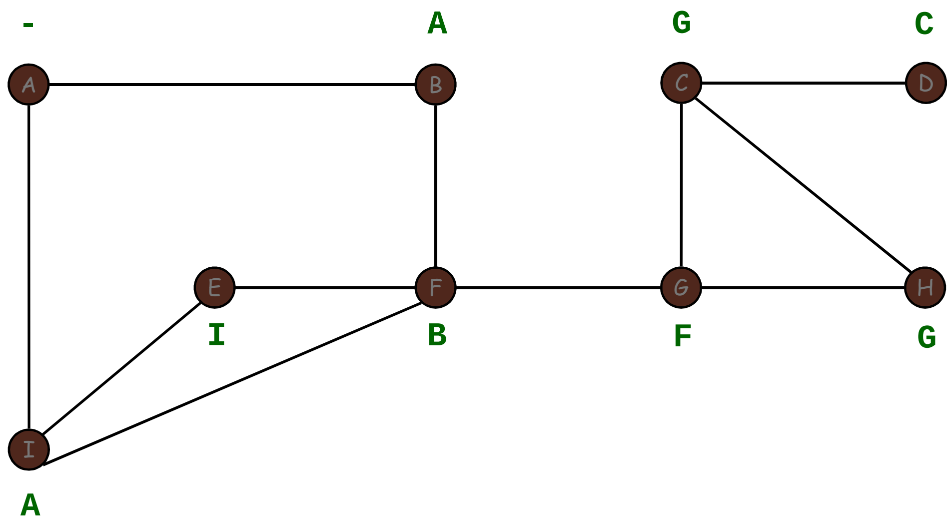


D finished

front

FIFO Queue

Breadth First Search



dequeue next vertex

front

FIFO Queue

Breadth First Search