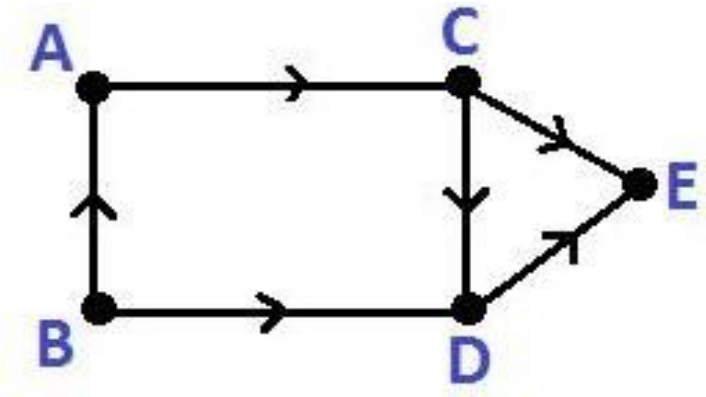
- **Graphs-**
  - A graph stores a collection of items in a **non-linear fashion.**
  - Graphs are made up of a finite set of nodes also known as vertices and lines that connect them also known as edges.
  - These are useful for representing real-life systems suchas computer networks.



Graph

Types of data structures

- ***The different types of Graphs are :***
  - Directed Graph
  - Non-directed Graph
  - Connected Graph
  - Non-connected Graph
  - Simple Graph
  - Multi-Graph

Graph Basics

- Graphs are collections of nodes connected by edges – G = (V,E) where V is a set of nodes and E a set of edges.
- Graphs are useful in a number of applications including
  - Shortest path problems
  - Maximum flow problems
- Graphs unlike trees are more general for they can have connected components.

Graph Types

- **Directed Graphs**: A directed graph edges allow travel in one direction.
- **Undirected Graphs**: An undirected graph edges allow travel in either direction.

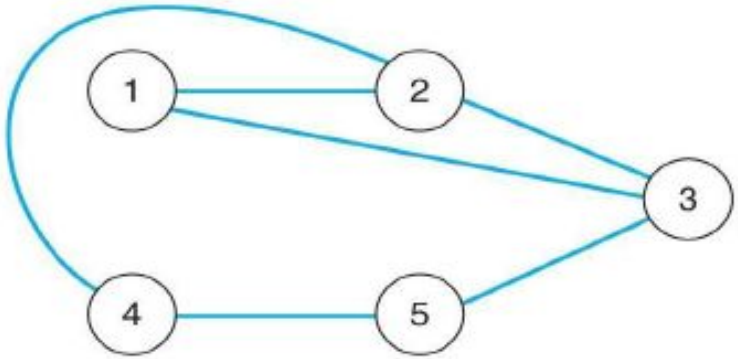


FIGURE 8.1A
The graph G = ({1, 2, 3, 4, 5}, {{1, 2}, {1, 3}, {2, 3}, {2, 4}, {3, 5}, {4, 5}})

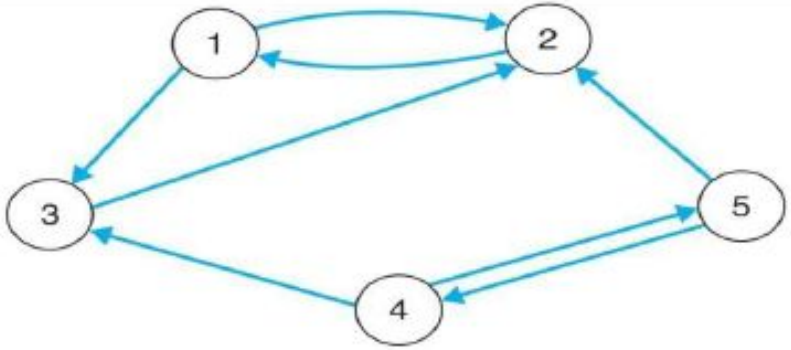


FIGURE 8.1B
The directed graph G = ({1, 2, 3, 4, 5}, {(1, 2), (1, 3), (2, 1), (3, 2), (4, 3), (4, 5), (5, 2), (5,4)})

Graph Terminology

- A graph is an ordered pair G=(V,E) with a set of vertices or nodes and the edges that connect them.
- A subgraph of a graph has a subset of the vertices and edges.
- The edges indicate how we can move through the graph.
- A path is a subset of E that is a series of edges between two nodes.
- A graph is connected if there is at least one path between every pair of nodes.

Graph Terminology

- The length of a path in a graph is the number of edges in the path.
- A complete graph is one that has an edge between every pair of nodes.
- A weighted graph is one where each edge has a cost for traveling between the nodes.
- A cycle is a path that begins and ends at the same node.
- An acyclic graph is one that has no cycles.
- An acyclic, connected graph is also called an unrooted tree

Data Structures for Graphs
An Adjacency Matrix

- For an undirected graph, the matrix will be symmetric along the diagonal.
- For a weighted graph, the adjacency matrix would have the weight for edges in the graph, zeros along the diagonal, and infinity (∞) every place else.

Adjacency Matrix Example 1

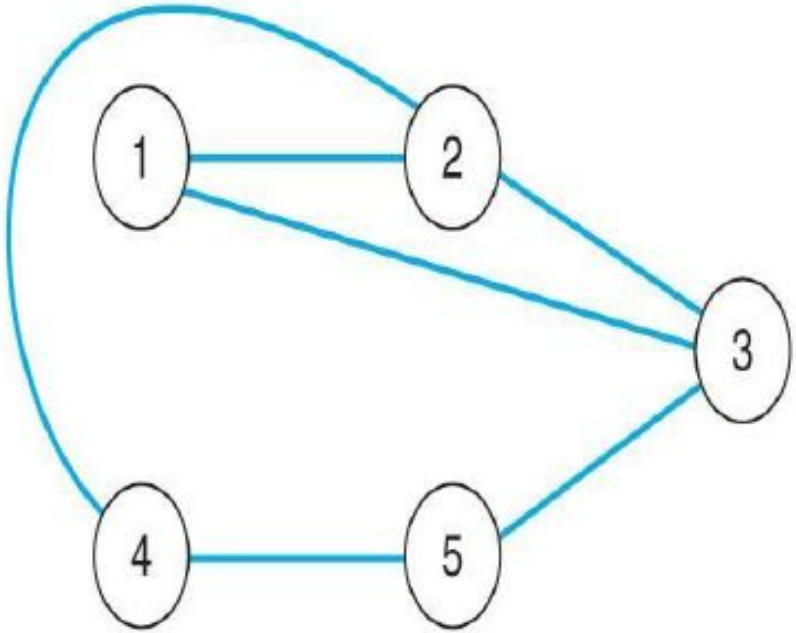


FIGURE 8.1A

The graph G = ({1, 2, 3, 4, 5}, {{1, 2}, {1, 3}, {2, 3}, {2, 4}, {3, 5}, {4, 5}})

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

FIGURE 8.2A

The adjacency matrix for the graph in Fig. 8.1(a)

Adjacency Matrix Example 2

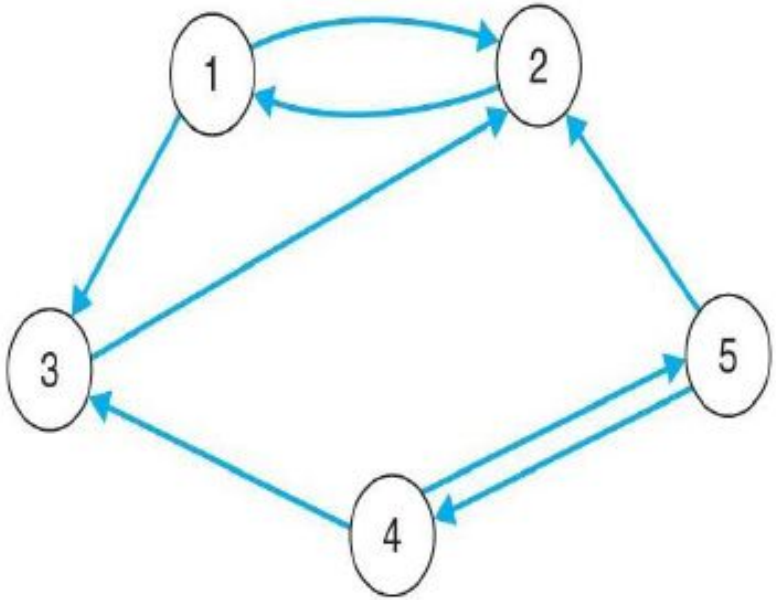


FIGURE 8.1B

The directed graph G = ({1, 2, 3, 4, 5}, {(1, 2), (1, 3), (2, 1), (3, 2), (4, 3), (4, 5), (5, 2), (5,4)})

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

FIGURE 8.2B

The adjacency matrix for the digraph in Fig. 8.1(b)

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.

Adjacency List Example 1

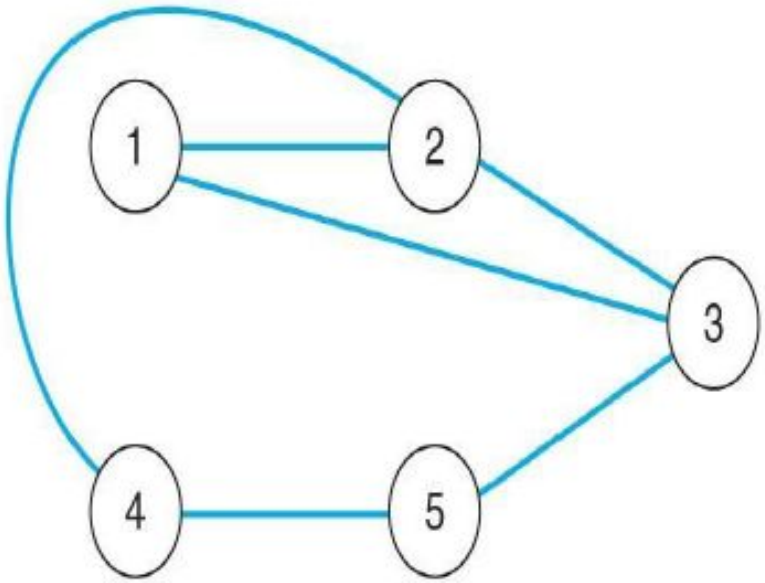


FIGURE 8.1A

The graph G = ({1, 2, 3, 4, 5}, {{1, 2}, {1, 3}, {2, 3}, {2, 4}, {3, 5}, {4, 5}})

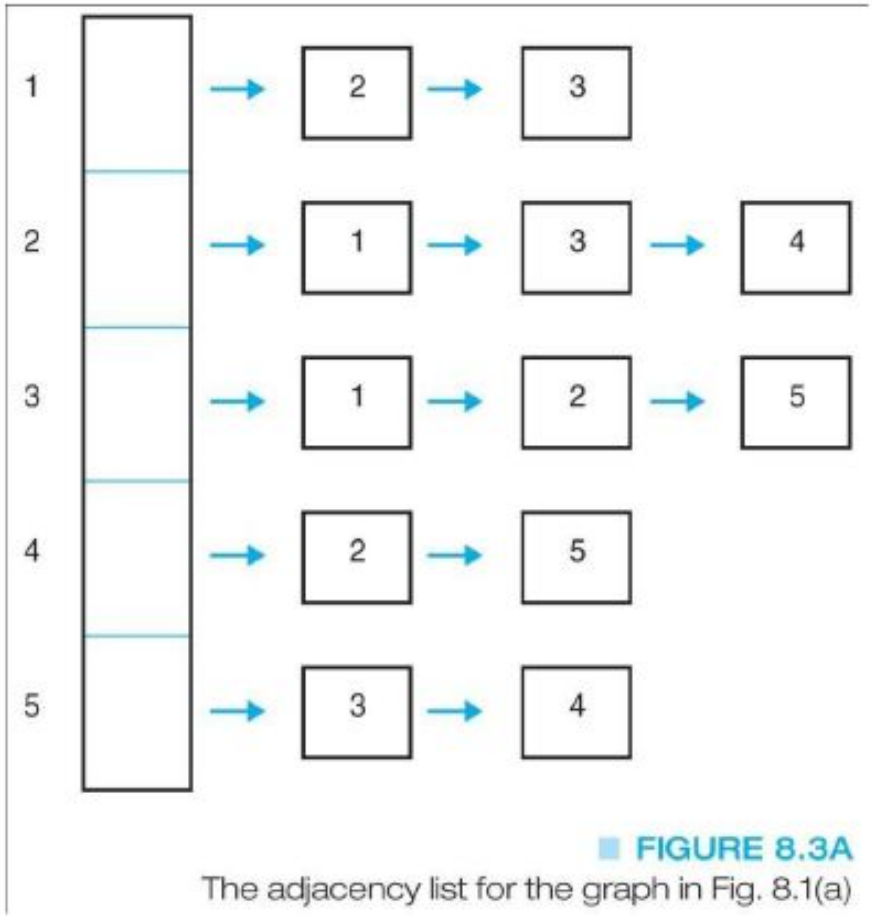


FIGURE 8.3A

The adjacency list for the graph in Fig. 8.1(a)

Adjacency List Example 2

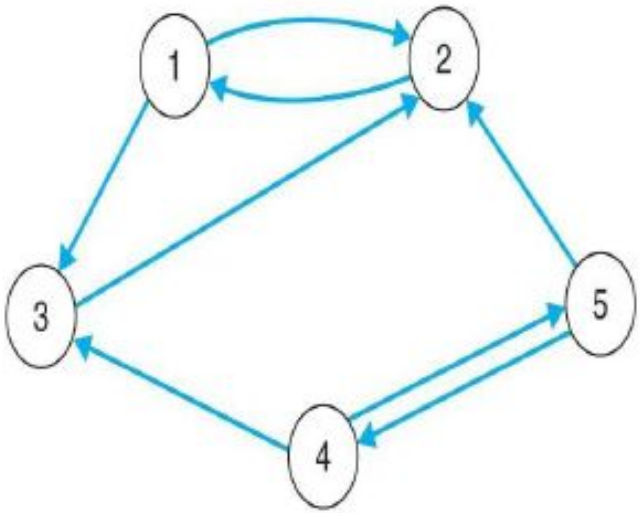


FIGURE 8.1B

The directed graph G = ([1, 2, 3, 4, 5], {(1, 2), (1, 3), (2, 1), (3, 2), (4, 3), (4, 5), (5, 2), (5,4)})

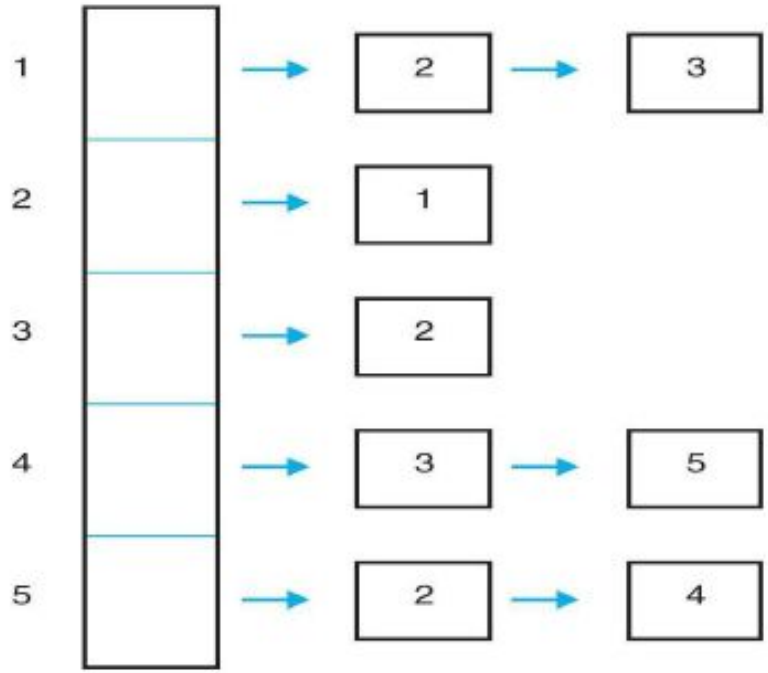


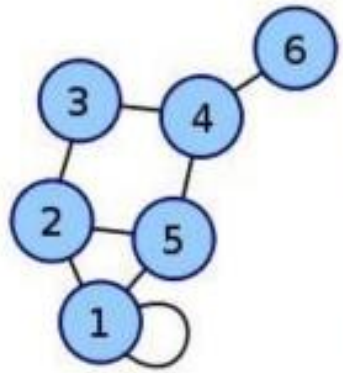
FIGURE 8.3B

The adjacency list for the graph in Fig. 8.1(b)

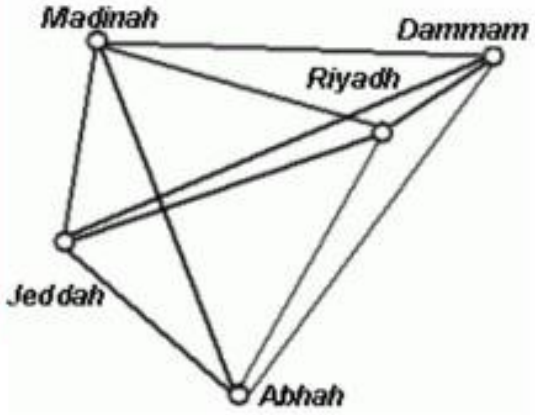
What is a Graph?

Graphs are Generalization of Trees.

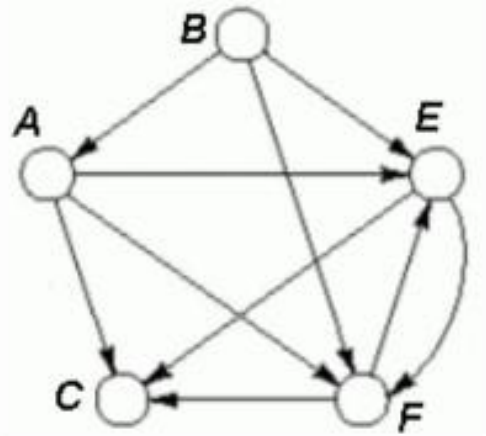
A simple graph G = (V, E) consists of a non-empty set V, whose members are called the vertices of G, and a set E of pairs of distinct vertices from V, called the edges of G.
A simple graph has *no loop* (An edge that connects a vertex to itself)
A simple graph has *no vertices joined by multiple edges*



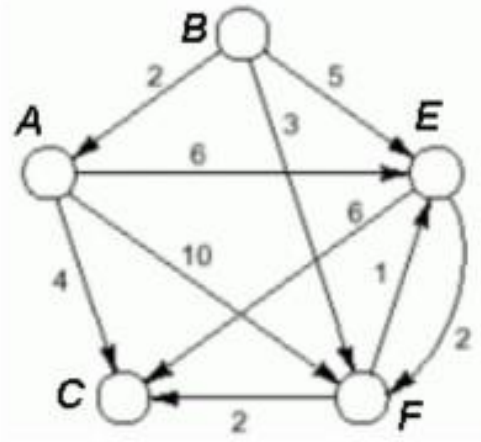
Undirected



Directed (Digraph)



Weighted



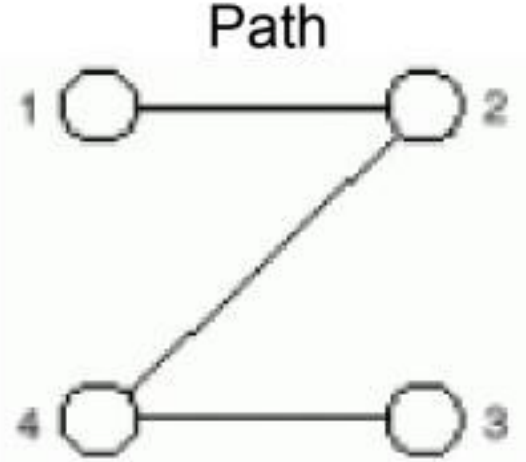
Some Applications of Graphs

- Finding the least congested route between two phones, given connections between switching stations.
- Determining if there is a way to get from one page to another, just by following links.
- Finding the shortest path from one city to another.
- As a traveling sales-person, finding the cheapest path that passes through all the cities that the sales person must visit.
- Determining an ordering of courses so that prerequisite courses are always taken first.
- Networks:Vertices represent computers and edges represent network
- connectionsbetween them.
- World Wide Web: Vertices represent webpages, and edges represent hyperlinks.

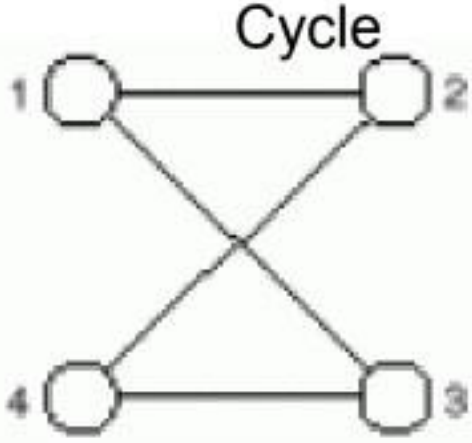
Graphs Terminologies

Adjacent Vertex: A vertex **W** is adjacent to a vertex **V** if there is an edge from **V** to **W**
Example: **B** is adjacent to **A**; but **A** is not adjacent to **B**

A path: A path is a sequence of consecutive edges in a graph. The length of the path is the number of edges traversed.



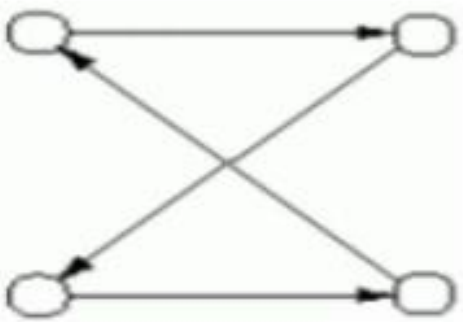
A cycle (or circuit): a subset of the edge set that forms a path such that the first vertex of the path is also the last.



Graphs Terminologies

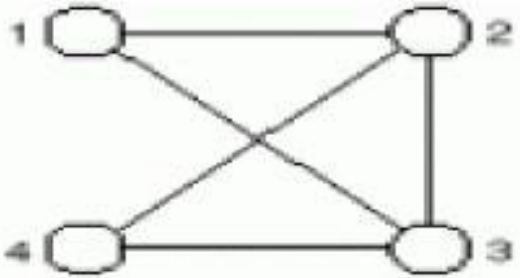
•A graph containing no cycles of any length is known as an acyclic graph, whereas a graph containing at least one cycle is called cyclic graph.

Directed Cyclic



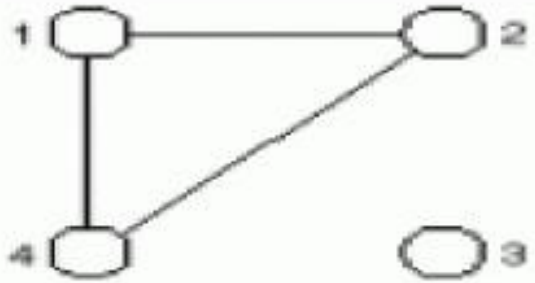
•*Connected graph:* A graph is connected if there is a path from any vertex to every other vertex. Alternatively a graph is connected if there is a path connecting every pair of vertices.

Connected



•An undirected graph that is not connected can be divided into **connected components** (maximal disjoint connected subgraphs). For example, the disconnected graph below is made of two connected components.

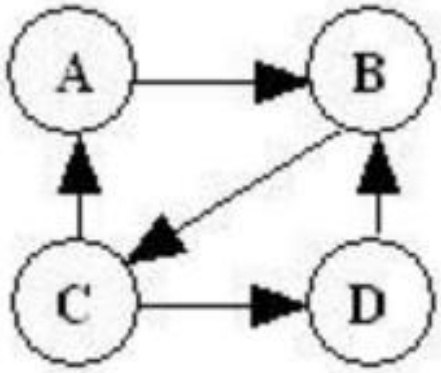
Disconnected



More Graph Terminologies

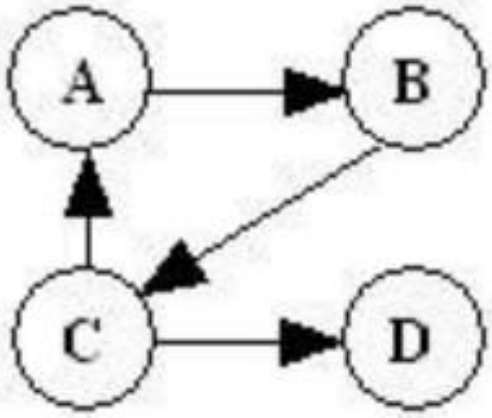
•Strongly Connected: If connected as a digraph [There is a path from each vertex to every other vertex]

Strongly Connected



•Weakly connected: If the underlying undirected graph is connected [There is at least one vertex that does not have a path to every other vertex]

Weakly Connected



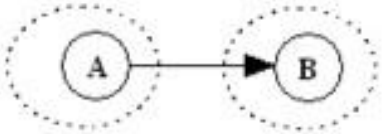
Strongly Connected Components (SCCs)

- A disconnected or weakly connected directed graph can be divided into two or more strongly connected components. Two vertices of a directed graph are in the same strongly connected component if and only if they are reachable from each other.

- Note: If a graph is strongly connected it has only one strongly connected component.

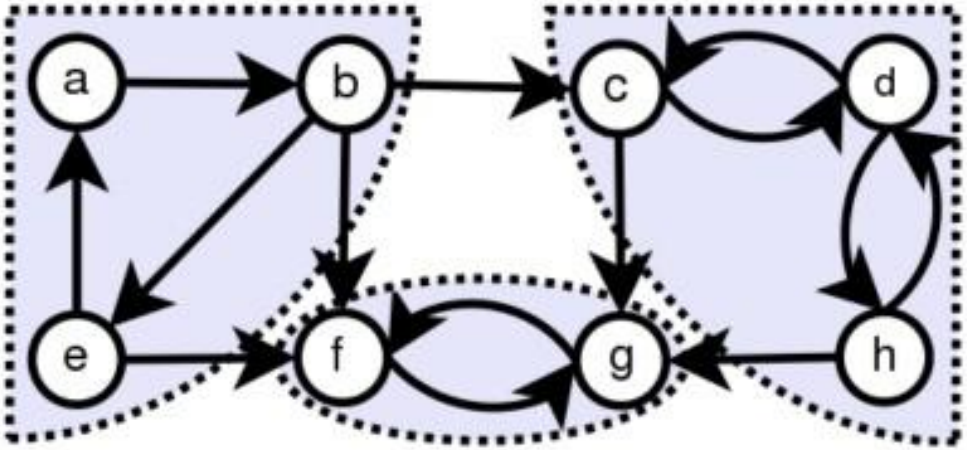
- Examples:

Graph1:



Graph1 has two strongly connected components

Graph2:



Graph2 has three strongly connected components

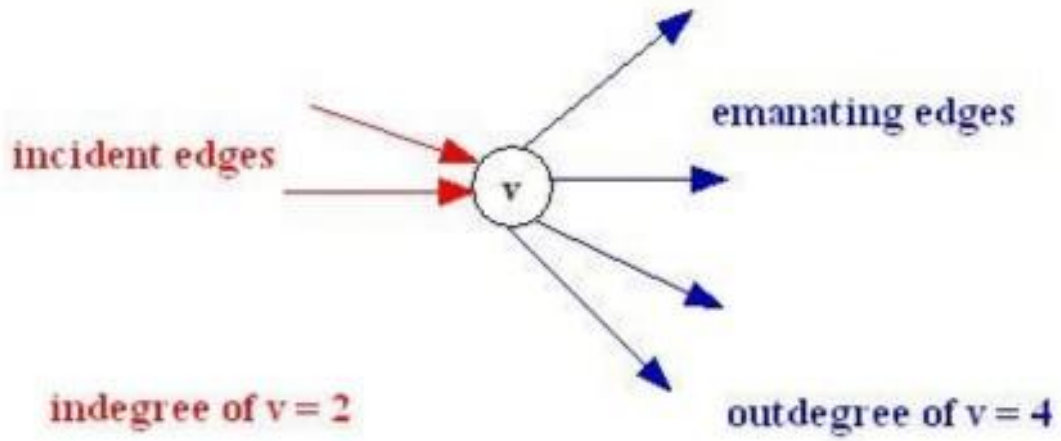
Further Graph Terminologies

Emanating edge: an edge $e = (v, w)$ is said to emanate from v.
$A(v)$ denotes the set of all edges emanating from v.

Incident edge: an edge $e = (v, w)$ is said to be incident to w.
$I(w)$ denote the set of all edges incident to w.

Out-degree: number of edges emanating from v: $|A(v)|$

In-degree: number of edges incident to w: $|I(w)|$



Further Graph Terminologies

Subgraph: A graph $G = (V_G , E_G)$ is a subgraph of $H = (V_H, E_H)$ if $V_G \subseteq V_H$ and $E_G \subseteq E_H$.

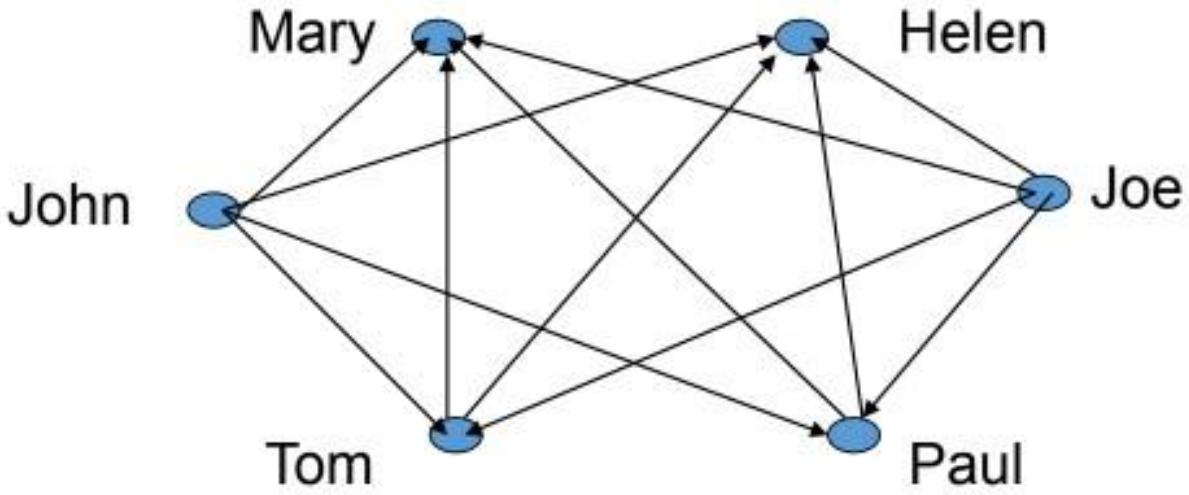
A spanning subgraph of G is a subgraph that contains all the vertices of G

A spanning tree of a connected graph is a spanning subgraph that is a tree

A "Real-life" Example of a Graph

V=set of 6 people: John, Mary, Joe, Helen, Tom, and Paul, of ages 12, 15, 12, 15, 13, and 13, respectively.

E ={(x,y) | if x is younger than y}



Intuition Behind Graphs

The nodes represent entities (such as people, cities, computers, words, etc.)

Edges (x,y) represent relationships between entities x and y, such as:

- "x loves y"
- "x hates y"
- "x is a friend of y" (note that this not necessarily reciprocal)
- "x considers y a friend"
- "x is a child of y"
- "x is a half-sibling of y"
- "x is a full-sibling of y"

In those examples, each relationship is a different graph

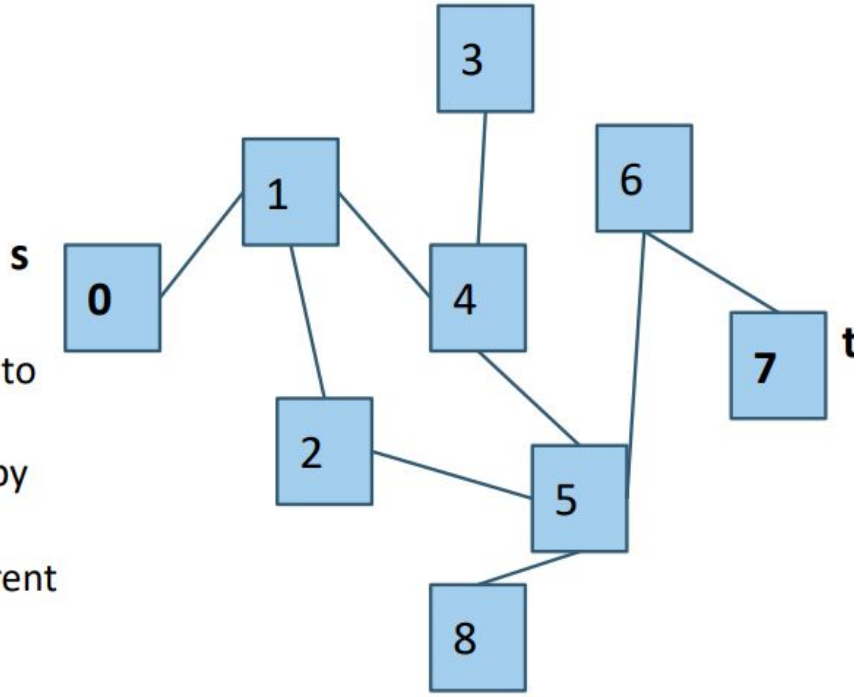
s-t path Problem

s-t path problem

- Given source vertex **s** and a target vertex **t**, does there exist a path between **s** and **t**?

Why does this problem matter? Some possible context:

- real life maps and trip planning – can we get from one location (vertex) to another location (vertex) given the current available roads (edges)
- family trees and checking ancestry – are two people (vertices) related by some common ancestor (edges for direct parent/child relationships)
- game states (Artificial Intelligence) can you win the game from the current vertex (think: current board position)? Are there moves (edges) you can take to get to the vertex that represents an already won game?



Graph Representations

For vertices:

an array or a linked list can be used

For edges:

Adjacency Matrix (Two-dimensional array)

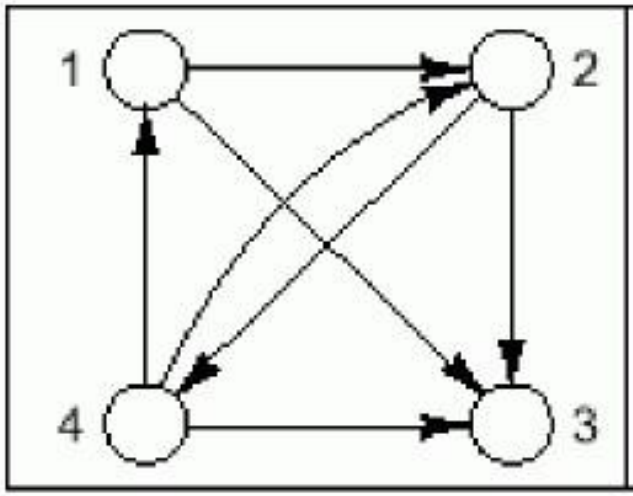
Adjacency List (One-dimensional array of linked lists)

Adjacency Matrix Representation

Adjacency Matrix uses a 2-D array of dimension |V|x|V| for edges. (For vertices, a 1-D array is used)

The presence or absence of an edge, (v, w) is indicated by the entry in row v, column w of the matrix.

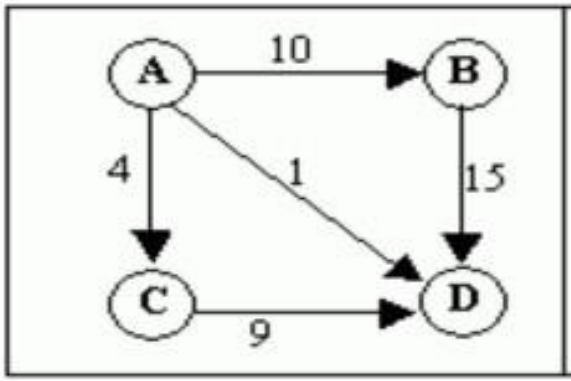
For an unweighted graph, boolean values could be used.



	1	2	3	4
1	F	T	T	F
2	F	F	T	T
3	F	F	F	F
4	T	T	T	F

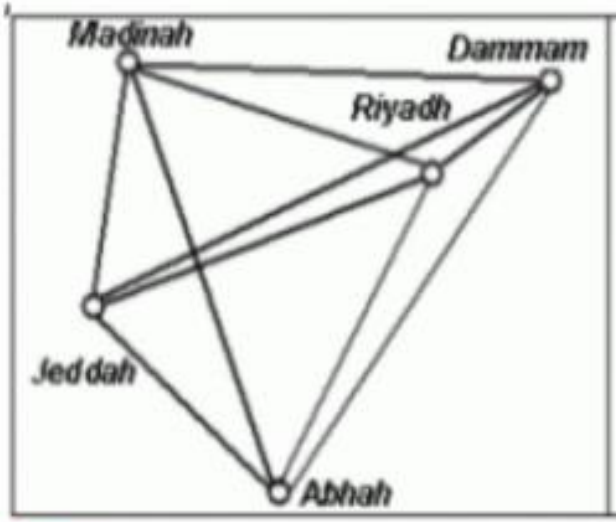
Adjacency Matrix Representation

For a weighted graph, the actual weights are used.



	A	B	C	D
A	∞	10	4	1
B	∞	∞	∞	15
C	∞	∞	∞	9
D	∞	∞	∞	∞

For undirected graph, the adjacency matrix is always symmetric.



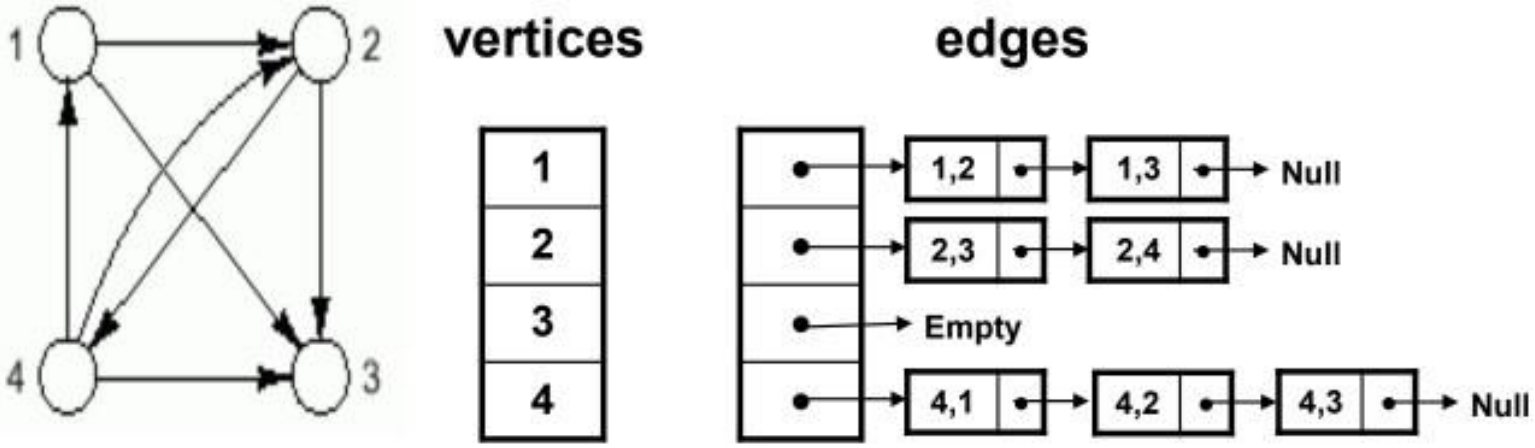
	Abhah	Dammam	Jeddah	Madinah	Riyadh
Abhah	F	T	T	T	T
Dammam	T	F	T	T	T
Jeddah	T	T	F	T	T
Madinak	T	T	T	F	T
Riyadh	T	T	T	T	F

Adjacency List Representation

This involves representing the set of vertices adjacent to each vertex as a list. Thus, generating a set of lists.

This can be implemented in different ways.

Vertices as a one dimensional array
Edges as an array of linked list (the emanating edges of vertex 1 will be in the list of the first element, and so on, …



Graph Traversals

- Introduction
- Breadth-First Traversal.
- Depth-First Traversals.
- Some traversal applications:
- Review Questions.

Graph Traversals

- Some algorithms require that every vertex of a graph be visited exactly once.
- The order in which the vertices are visited may be important, and may depend upon the particular algorithm.
- The two common traversals:

  - depth-first

  - breadth-first

Graph Traversals:
Depth First Search 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 Search Traversal Example

- Consider the following graph:

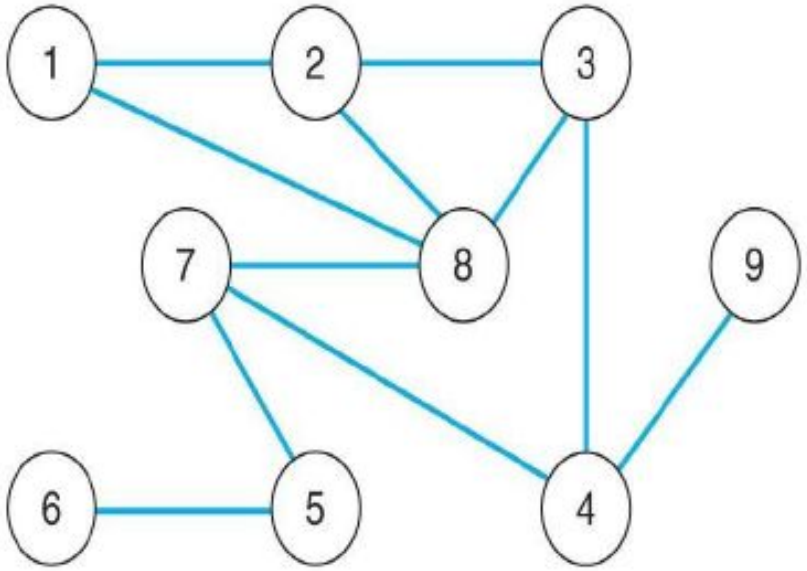


FIGURE 8.4
A graph

- The order of the depth-first traversal of this graph starting at node 1 would be:
  1, 2, 3, 4, 7, 5, 6, 8, 9

Breadth First Search 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 Traversal Example

- Consider the following graph:

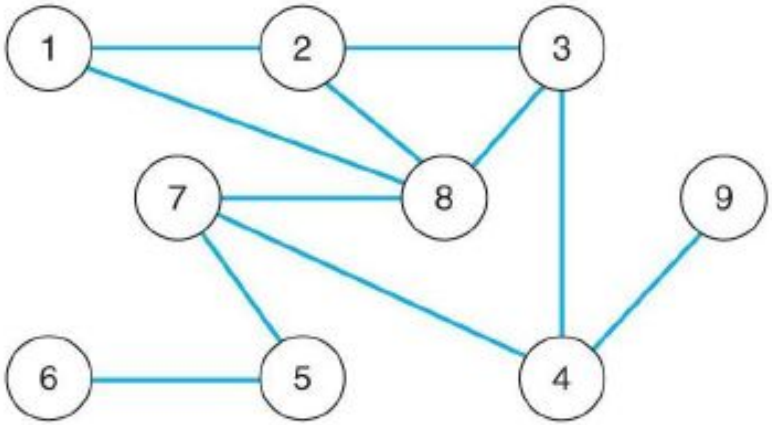


FIGURE 8.4
A graph

- The order of the breadth-first traversal of this graph starting at node 1 would be: 1, 2, 8, 3, 7, 4, 5, 9, 6

Introduction

- To traverse a graph is to systematically visit and process each node in the graph exactly once.

- There are two common graph traversal algorithms that are applicable to both directed and undirected graphs :

  - BreadthFirst Traversal (BFS)
  - DepthFirst Traversal (DFS)

Introduction (Cont'd)

The BFS and DFS traversal of a graph G is not unique. A traversal depends both on the starting vertex, and on the order of traversing the adjacent vertices of each node.

General traversal algorithms [dfsAllVertices and bfsAllVertices] that will visit each vertex of a graph G in those algorithms that require each vertex to be visited.

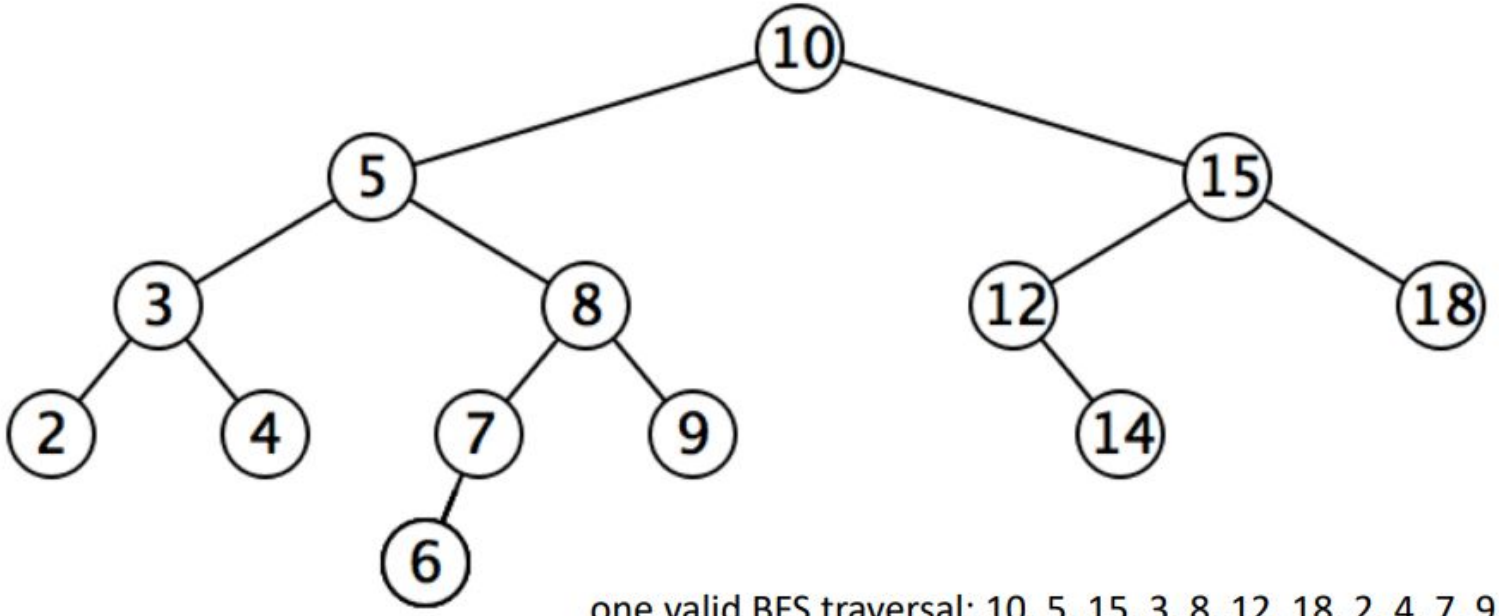
- **BFS:** Tries to explore all the neighbors it can reach from the current node. It will use a **queue** data structure.
- **DFS:** Tries to reach the farthest node from the current node and come back (backtrack) to the current node to explore its other neighbors. This will use a **stack** data structure.

Graph traversals: BFS

Breadth First Search - a traversal on graphs (or on trees since those are also graphs) where you traverse level by level. So in this one we'll get to all the shallow nodes before any "deep nodes".

Intuitive ways to think about BFS:

- opposite way of traversing compared to DFS
- a sound wave spreading from a starting point, going outwards in all directions possible.
- mold on a piece of food spreading outwards so that it eventually covers the whole surface



one valid BFS traversal: 10, 5, 15, 3, 8, 12, 18, 2, 4, 7, 9, 14, 6

Breadth-First Traversal Algorithm

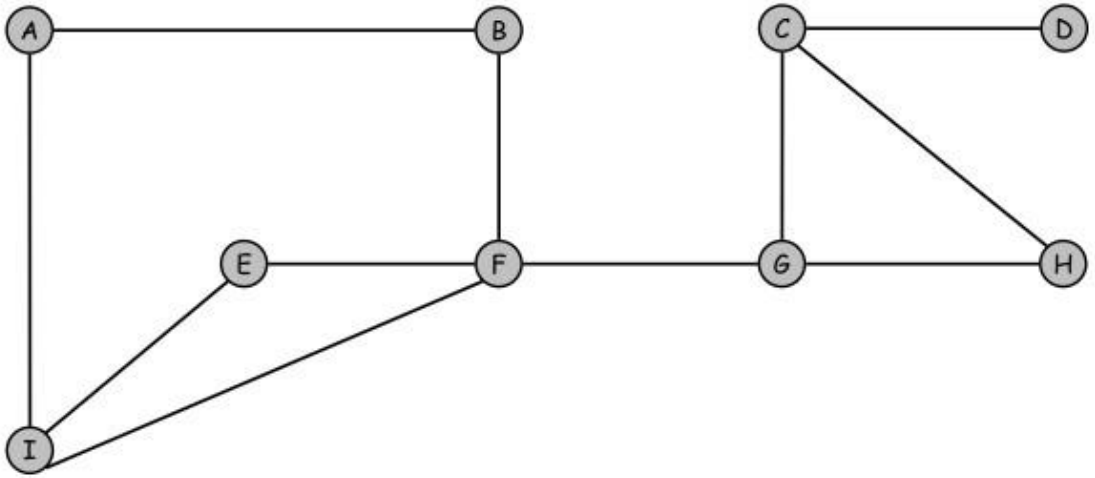
In this method, After visiting a vertex v, we must visit all its adjacent vertices w1, w2, w3, ..., before going down to the next level to visit vertices adjacent to w1 etc.

The method can be implemented using a queue.

```
enqueue the starting vertex
while(queue is not empty){
    dequeue a vertex v from the queue;
    visit v.
    enqueue vertices adjacent to v that were never enqueued;
}
```

• A BFS traversal of a graph results in a breadth-first tree or in a forest of such trees.

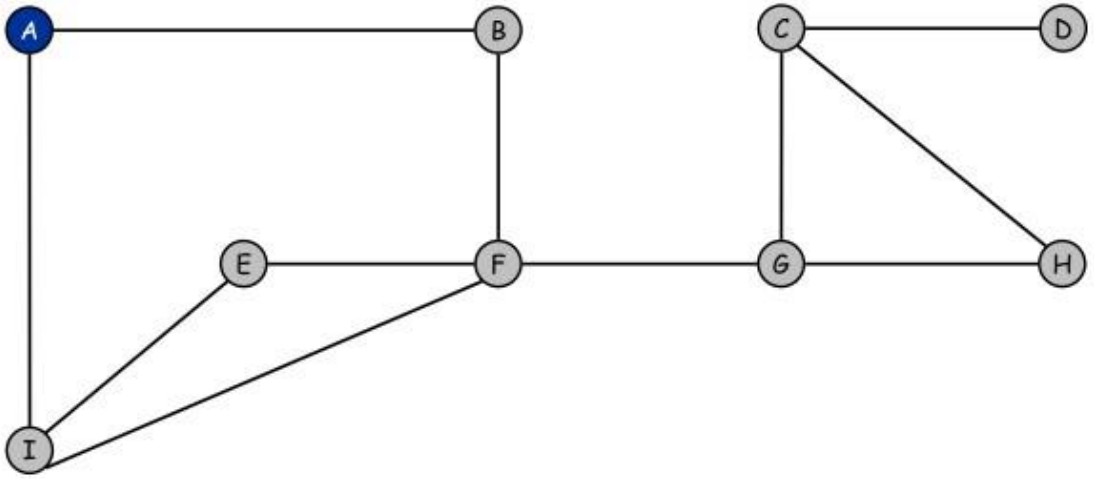
Breadth First Search



front

FIFO Queue

Breadth First Search

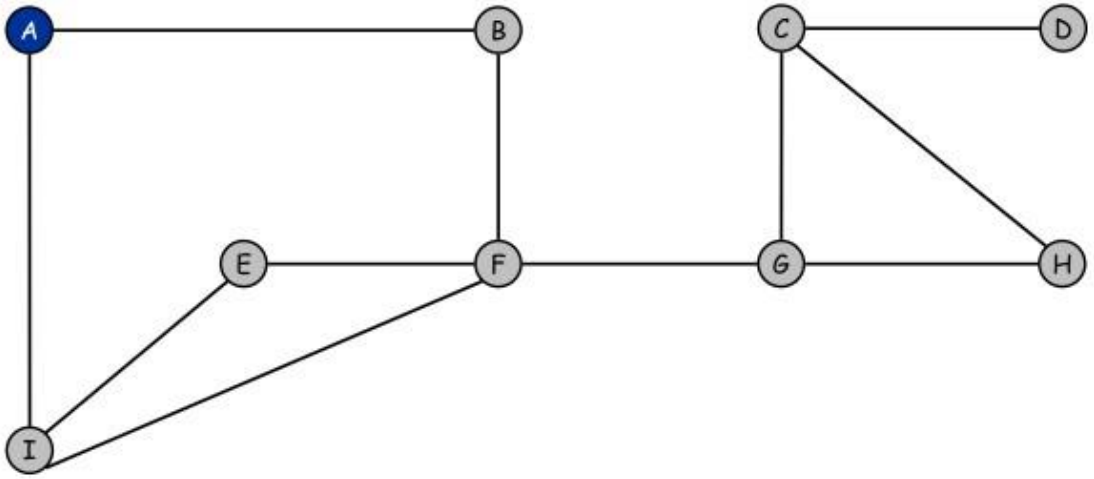


enqueue source node

front **A**

FIFO Queue

Breadth First Search

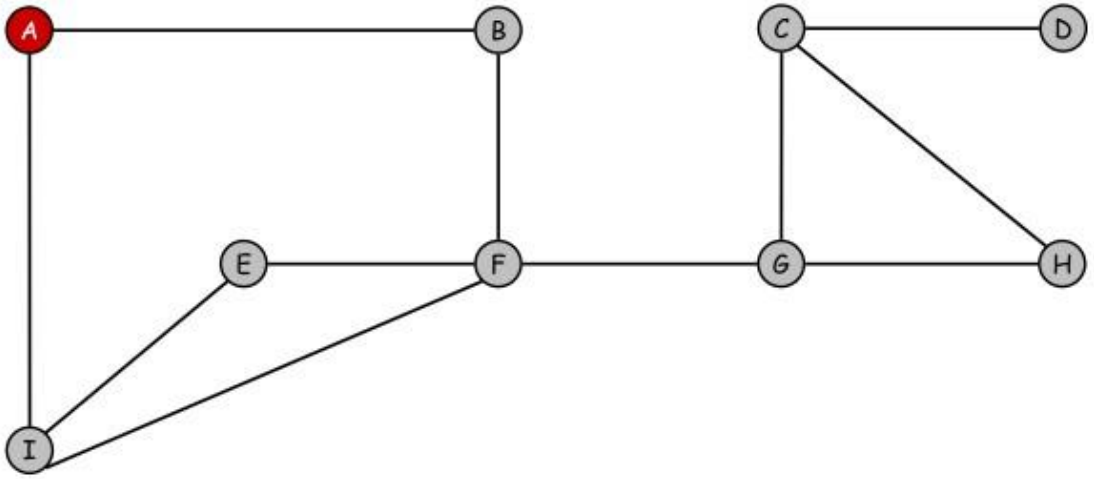


dequeue next vertex

front **A**

FIFO Queue

Breadth First Search

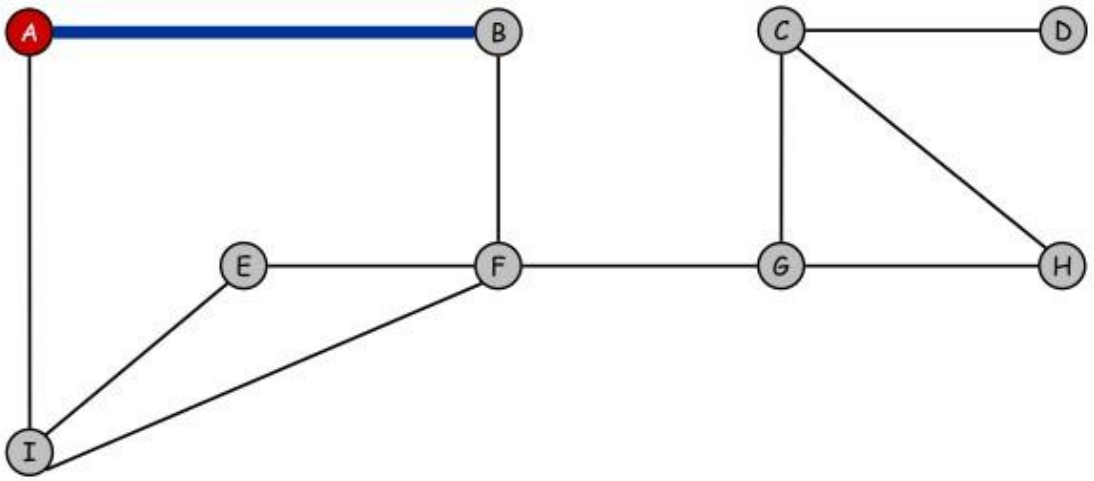


visit neighbors of A

front

FIFO Queue

Breadth First Search

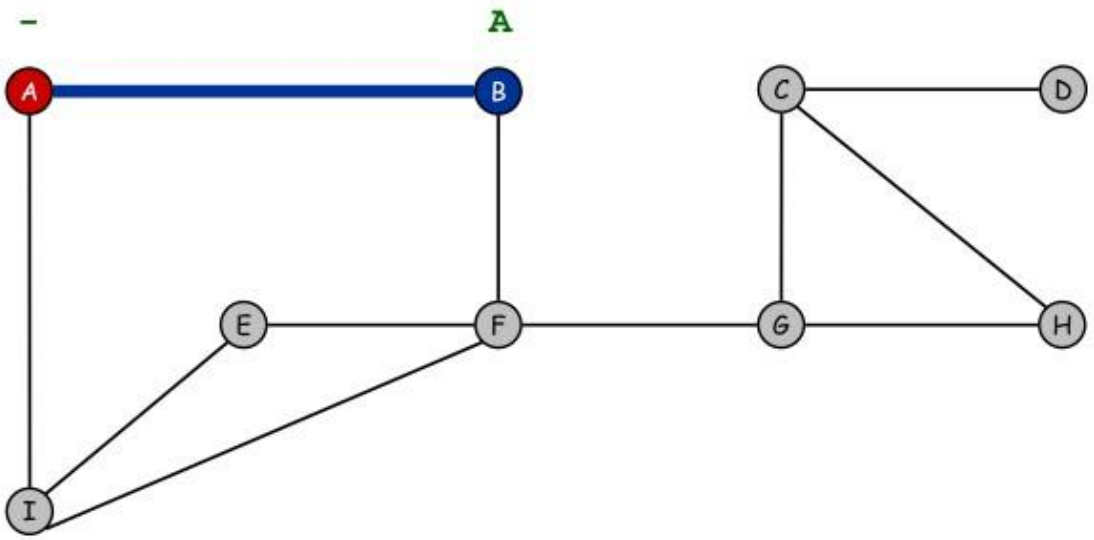


visit neighbors of A

front

FIFO Queue

Breadth First Search

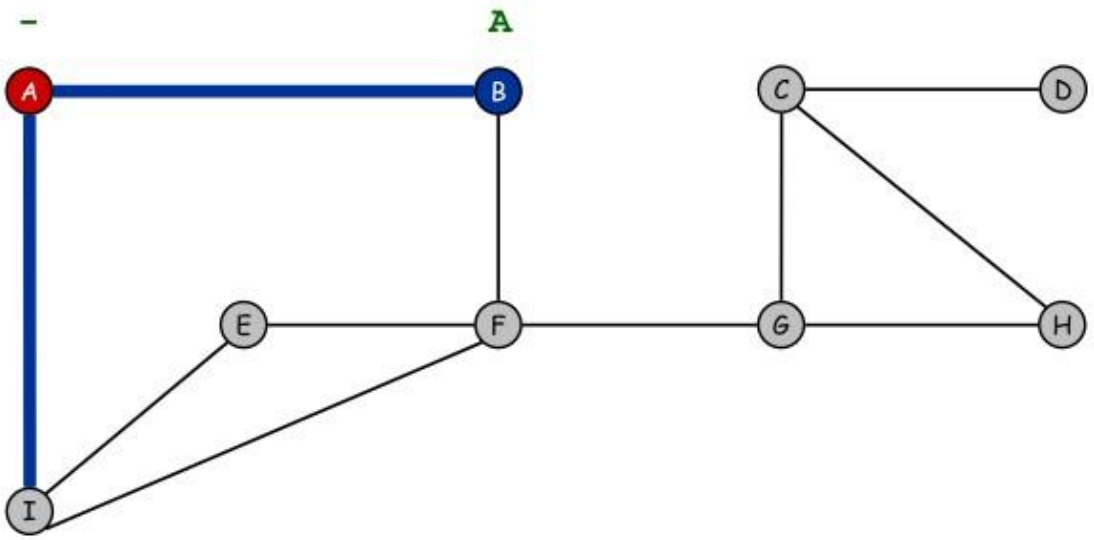


B discovered

front **B**

FIFO Queue

Breadth First Search

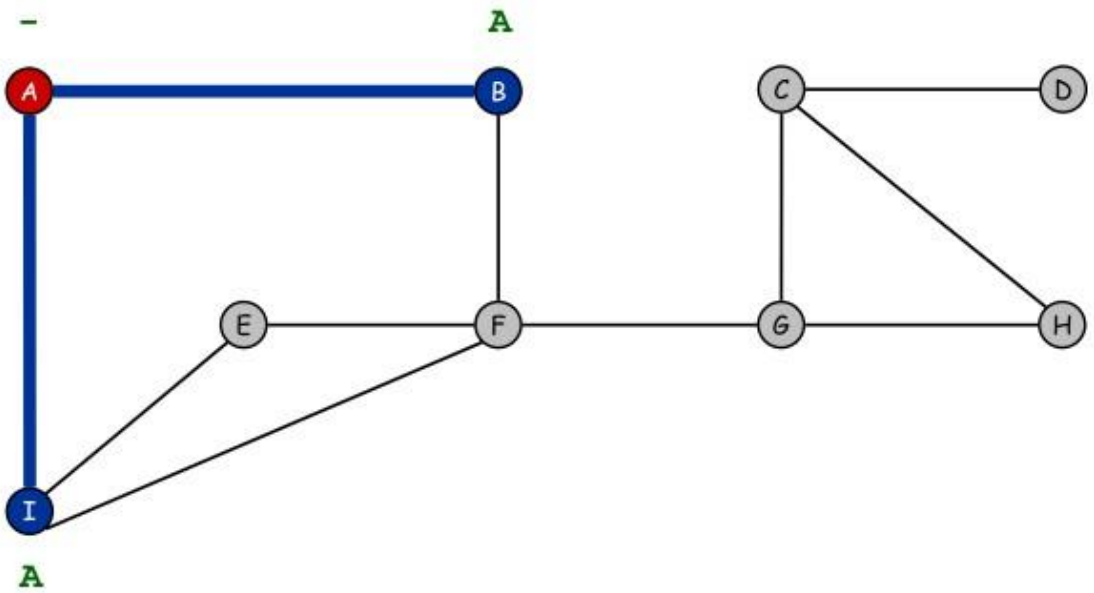


visit neighbors of *A*

front **B**

FIFO Queue

Breadth First Search

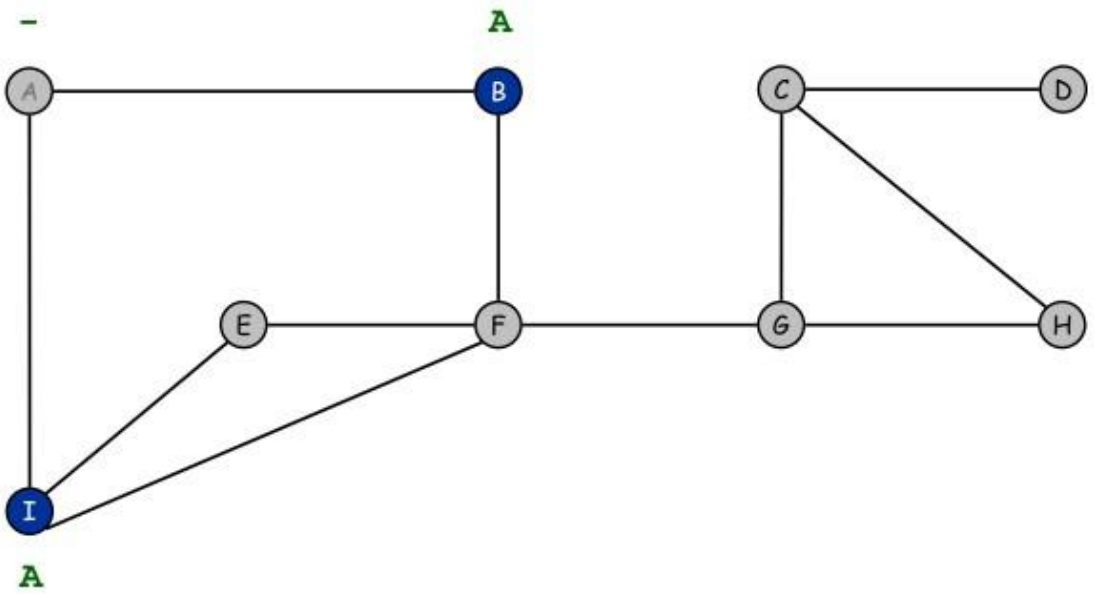


I discovered

front **B I**

FIFO Queue

Breadth First Search

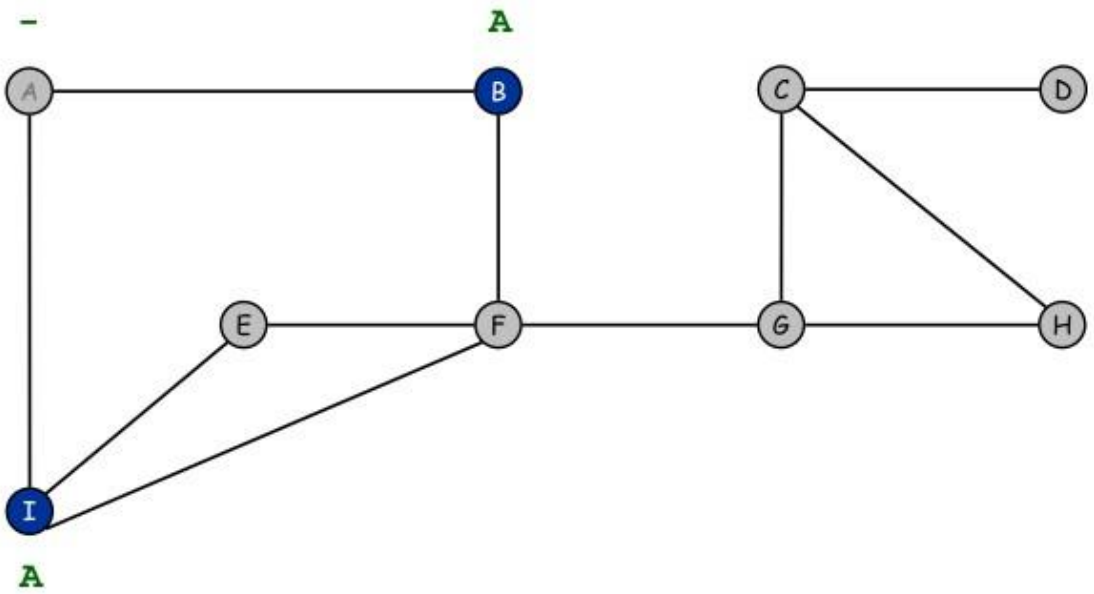


finished with *A*

front **B I**

FIFO Queue

Breadth First Search

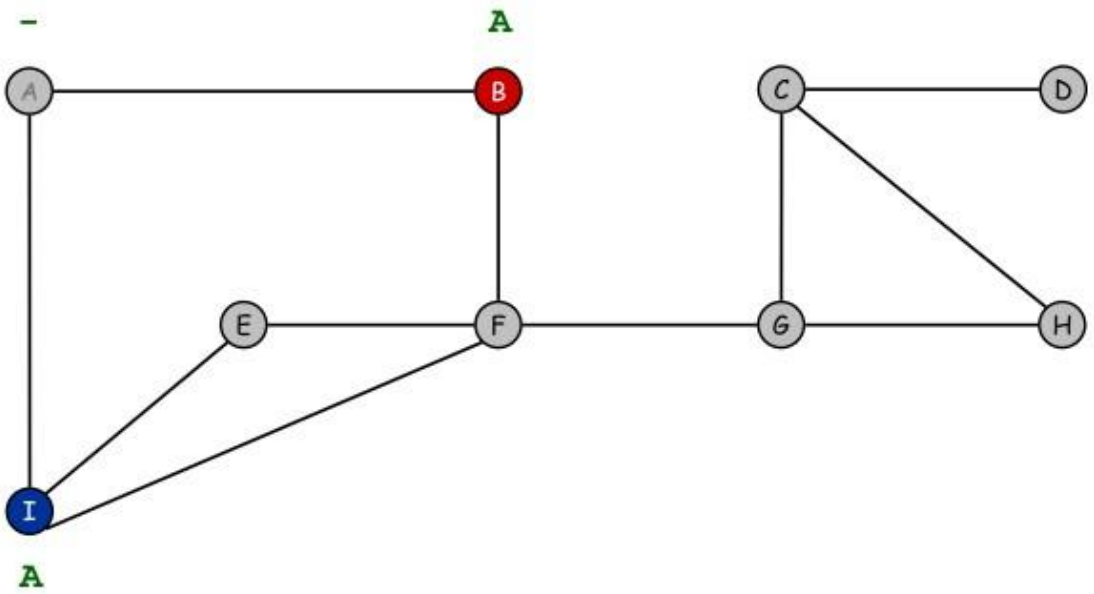


dequeue next vertex

front **B I**

FIFO Queue

Breadth First Search

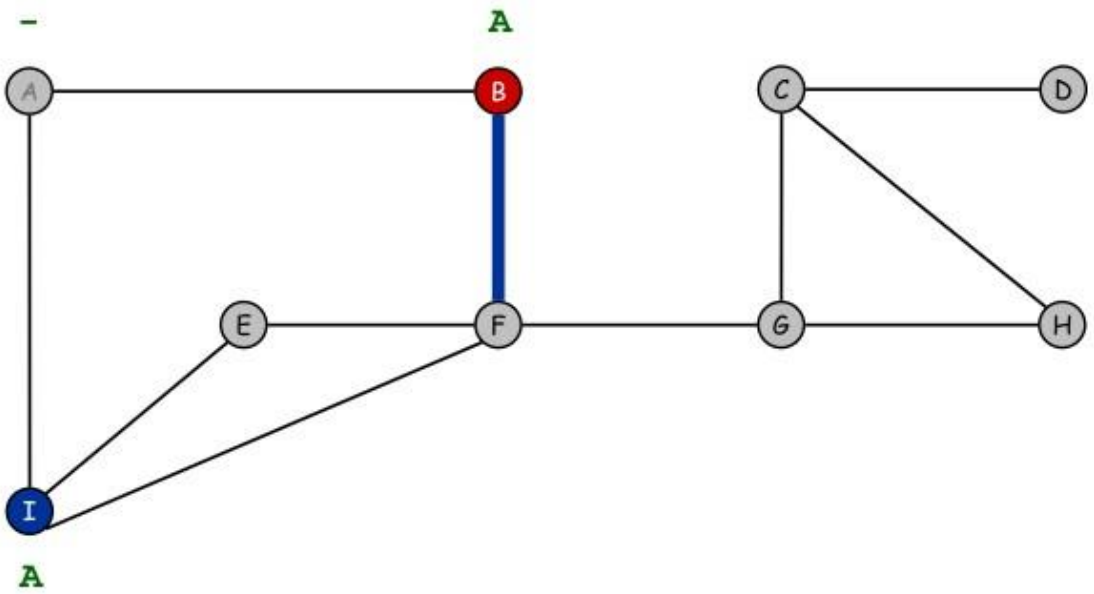


visit neighbors of B

front | I

FIFO Queue

Breadth First Search

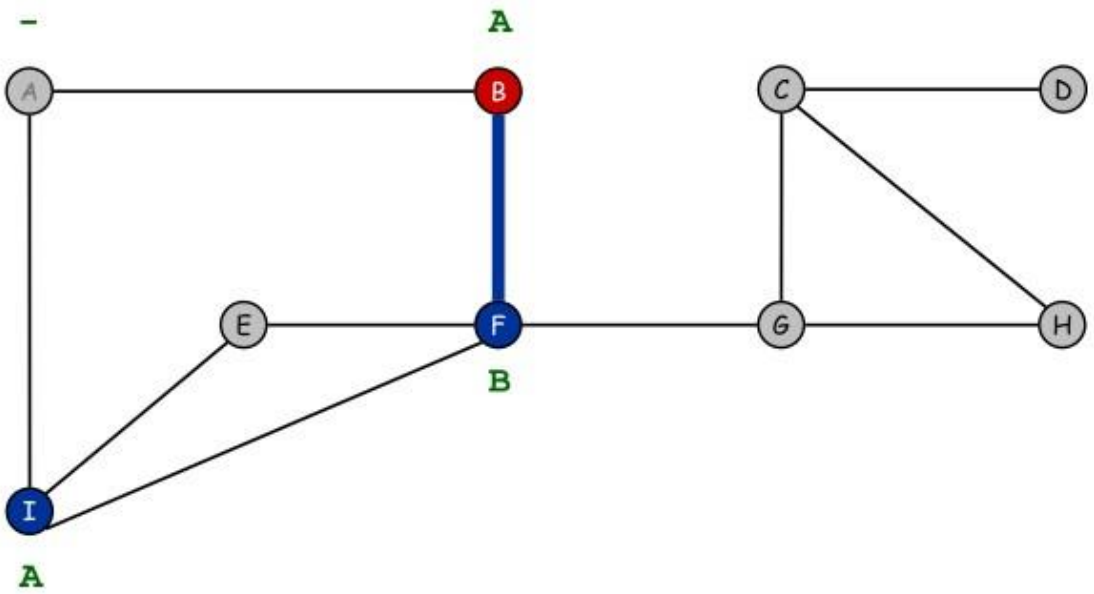


visit neighbors of B

front | I

FIFO Queue

Breadth First Search

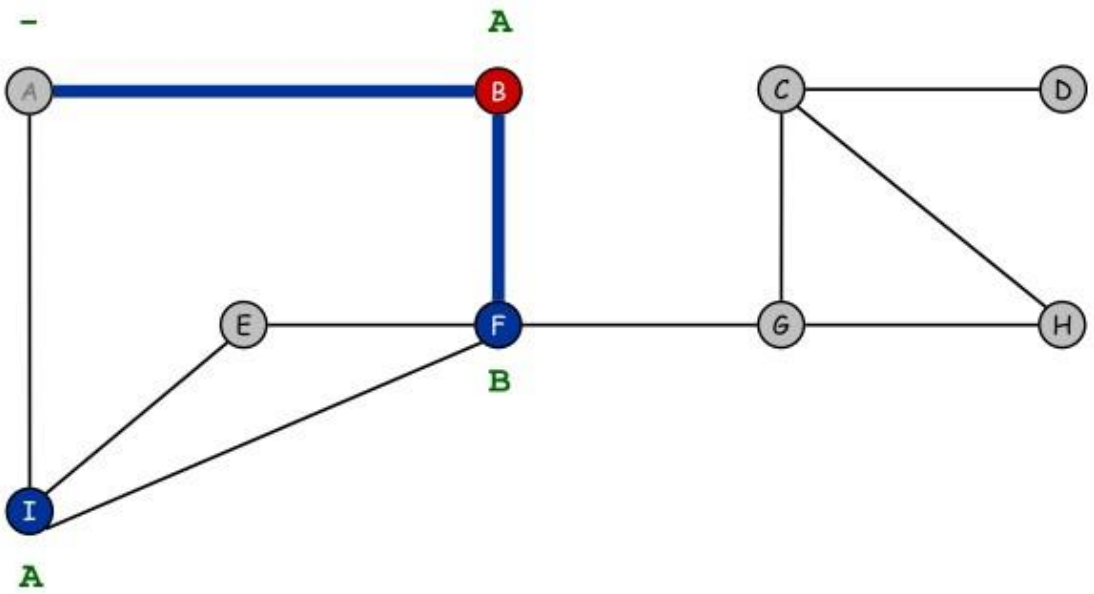


F discovered

front **I F**

FIFO Queue

Breadth First Search

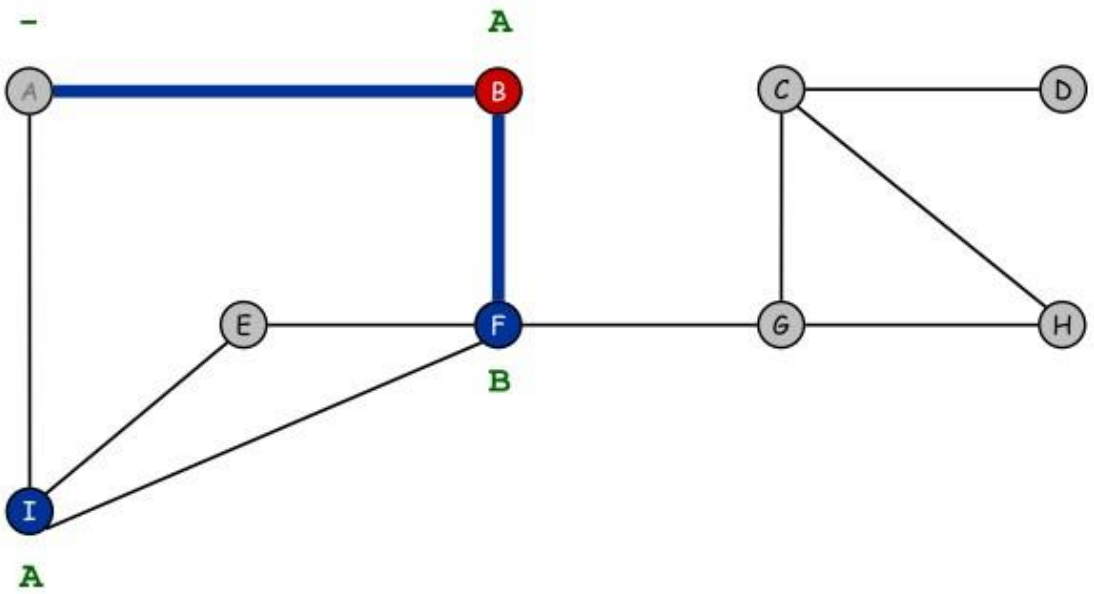


visit neighbors of B

front **I F**

FIFO Queue

Breadth First Search

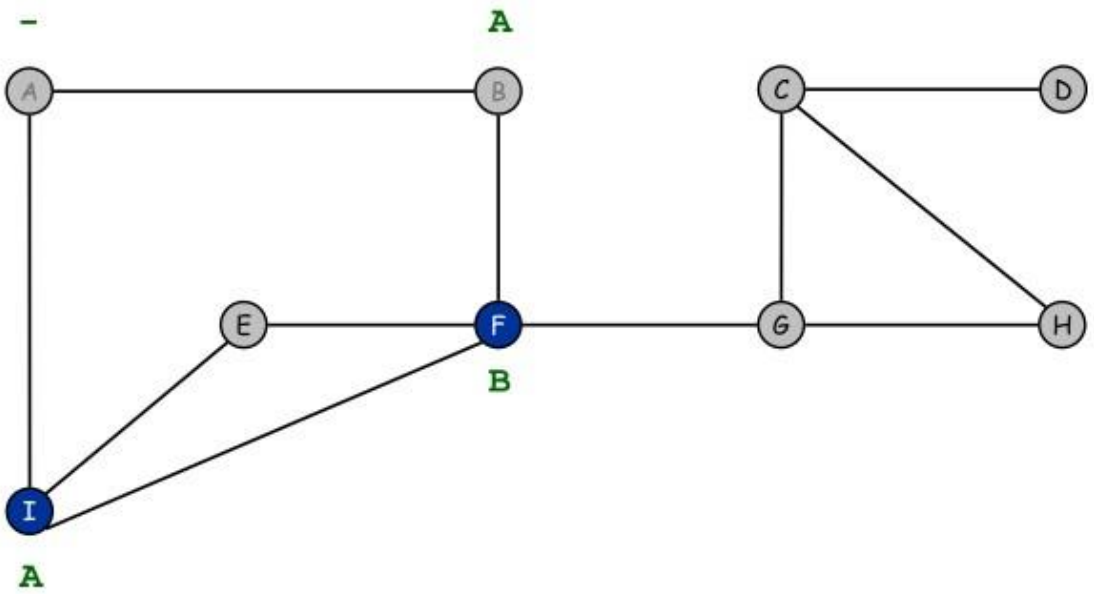


A already discovered

front **I F**

FIFO Queue

Breadth First Search

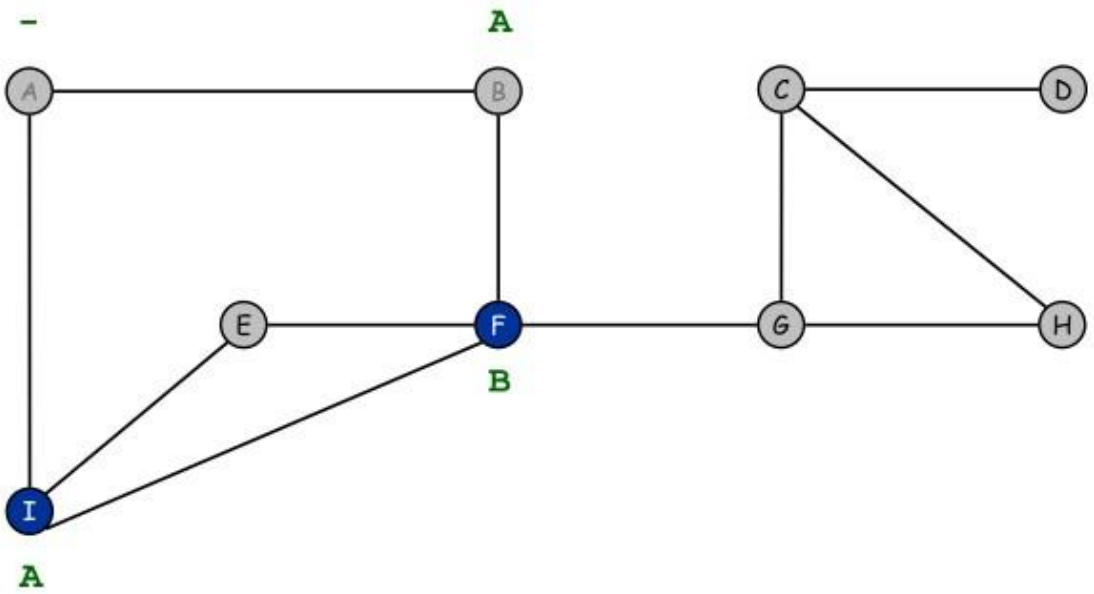


finished with B

front **I F**

FIFO Queue

Breadth First Search

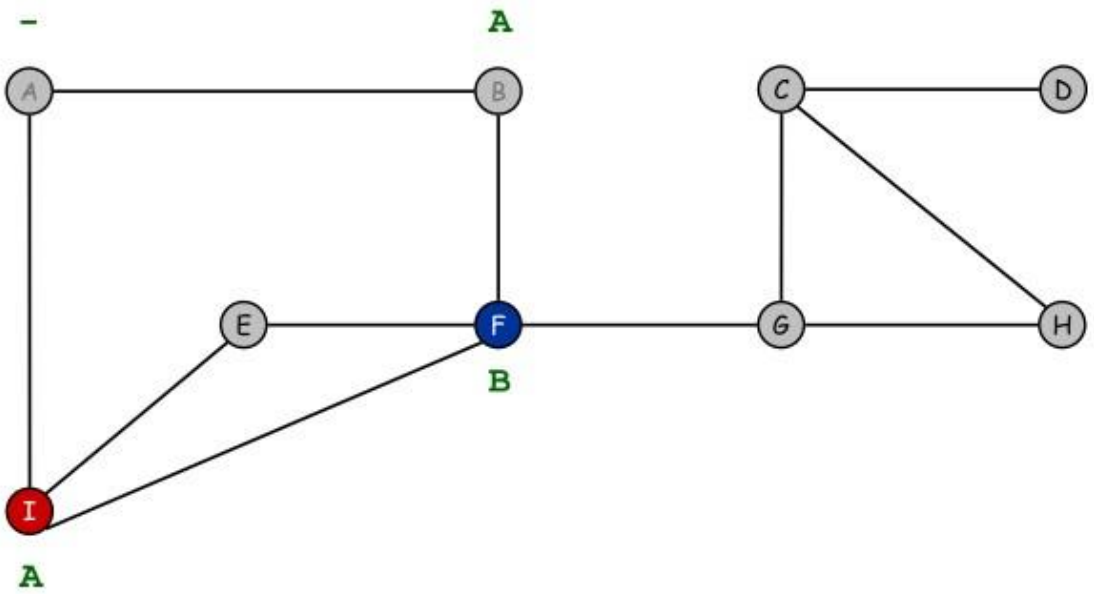


dequeue next vertex

front **I F**

FIFO Queue

Breadth First Search

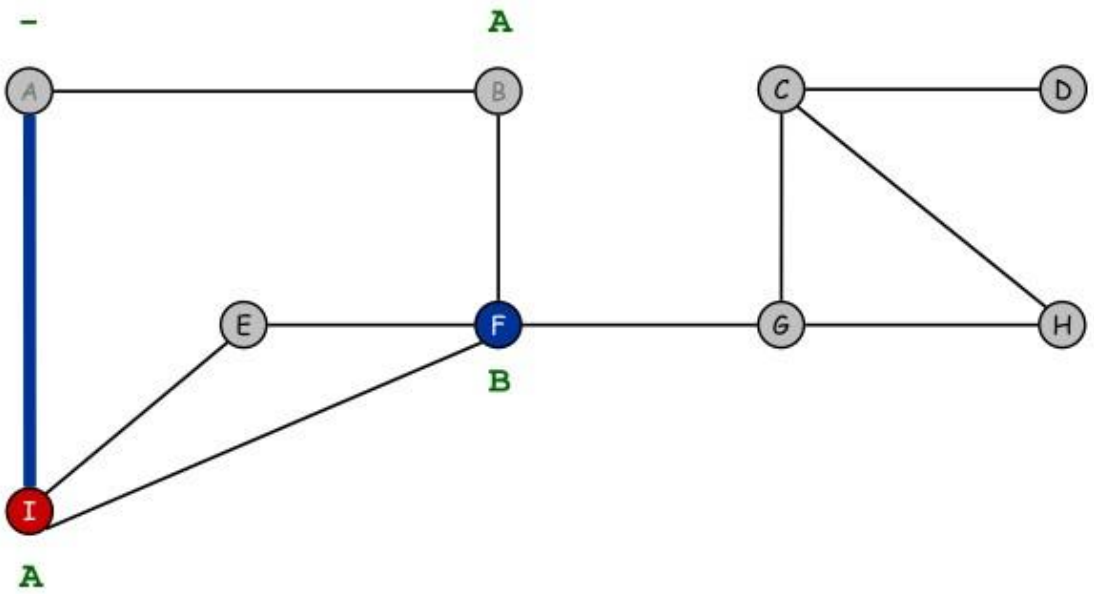


visit neighbors of I

front **F**

FIFO Queue

Breadth First Search

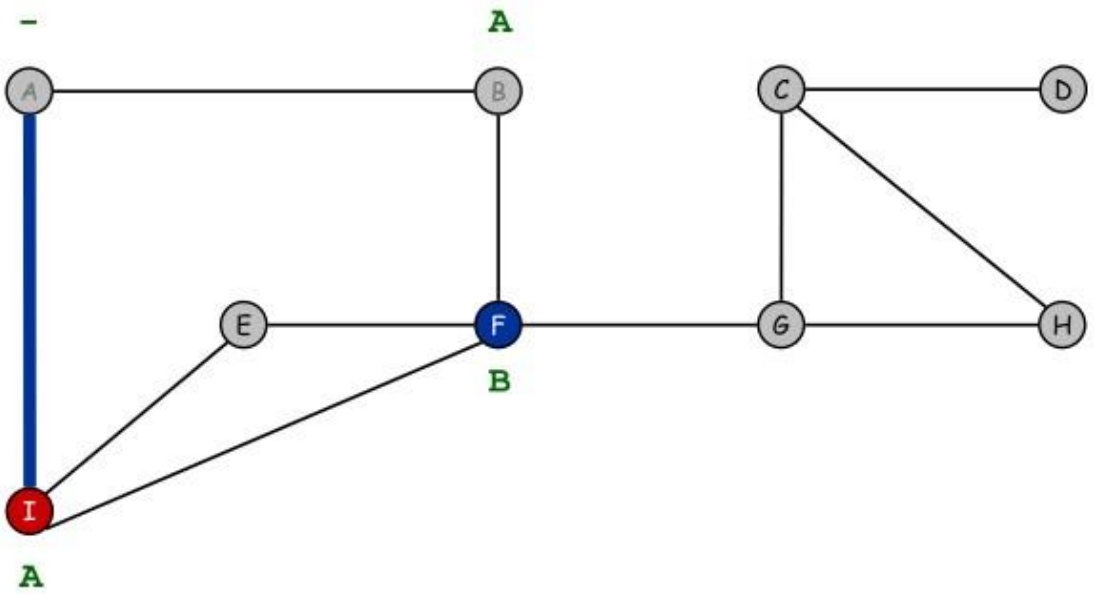


visit neighbors of I

front **F**

FIFO Queue

Breadth First Search

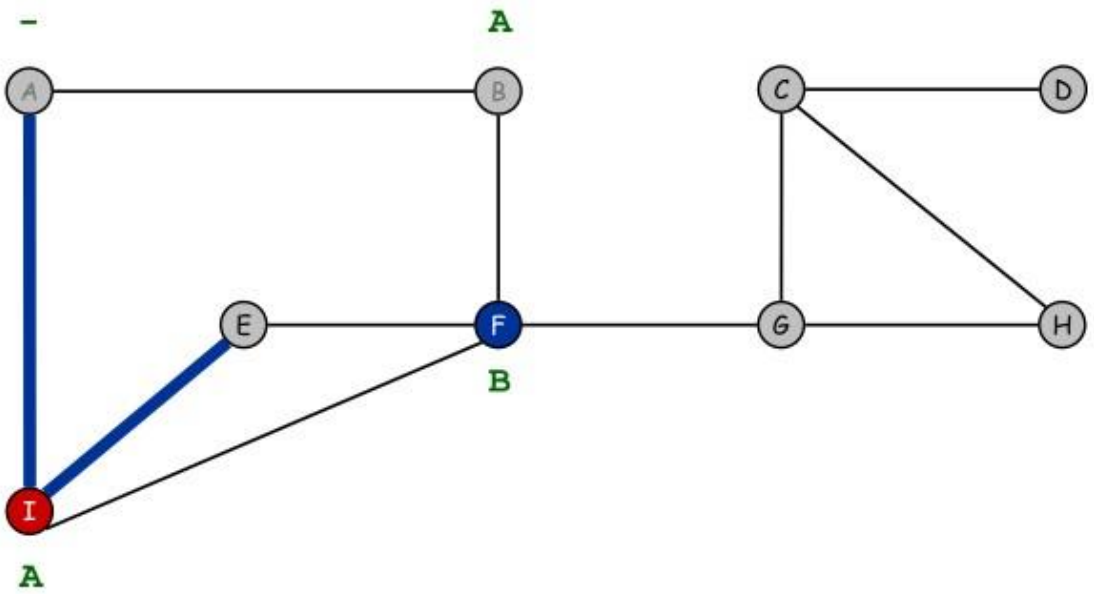


A already discovered

front **F**

FIFO Queue

Breadth First Search

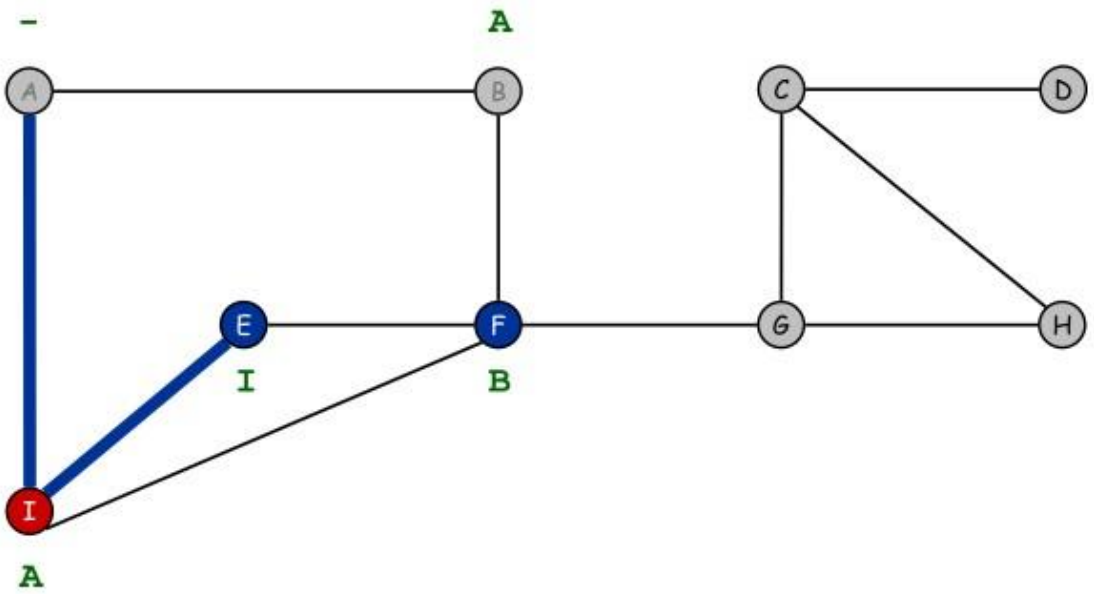


visit neighbors of I

front **F**

FIFO Queue

Breadth First Search

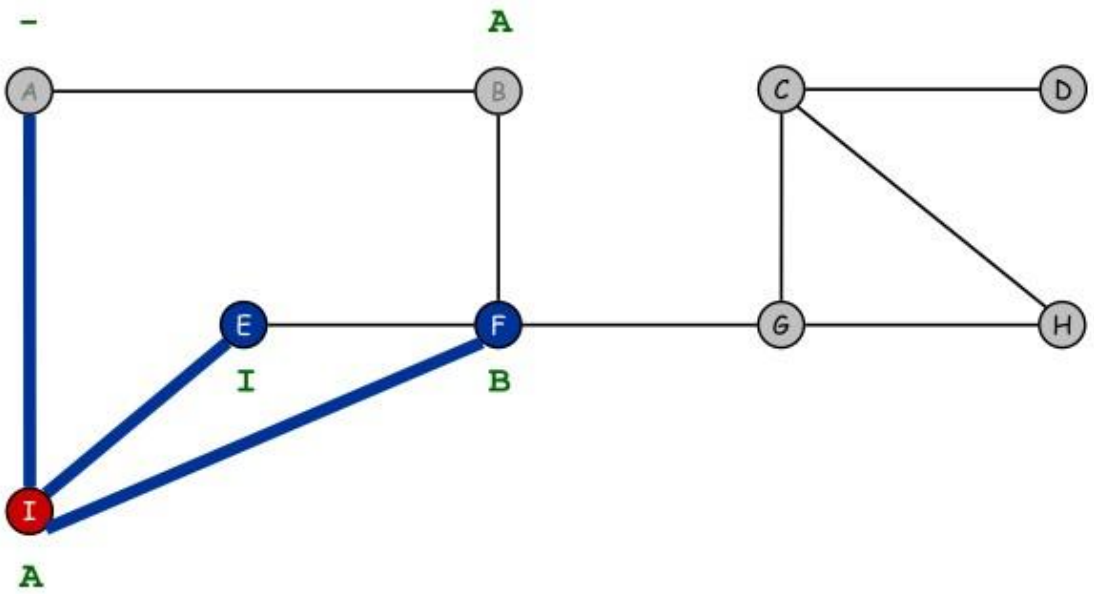


E discovered

front **F E**

FIFO Queue

Breadth First Search

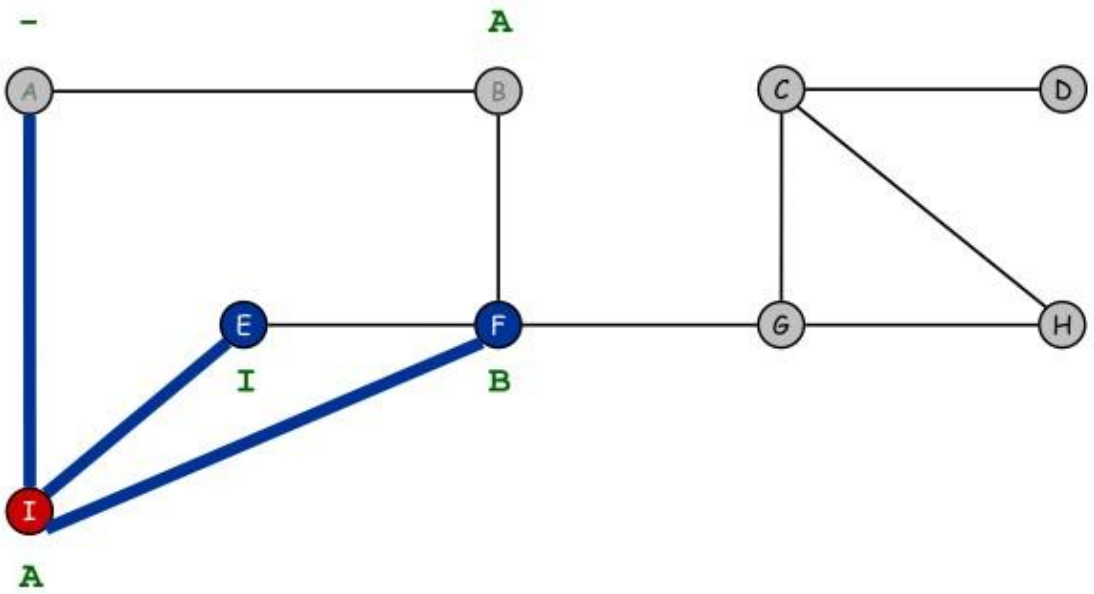


visit neighbors of I

front **F E**

FIFO Queue

Breadth First Search

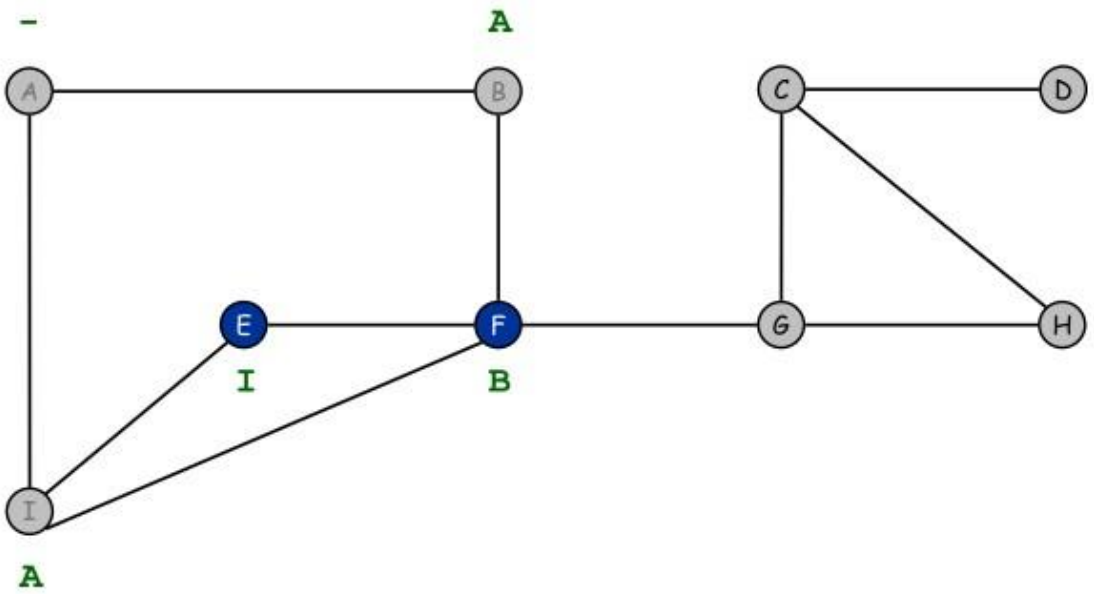


F already discovered

front **F E**

FIFO Queue

Breadth First Search

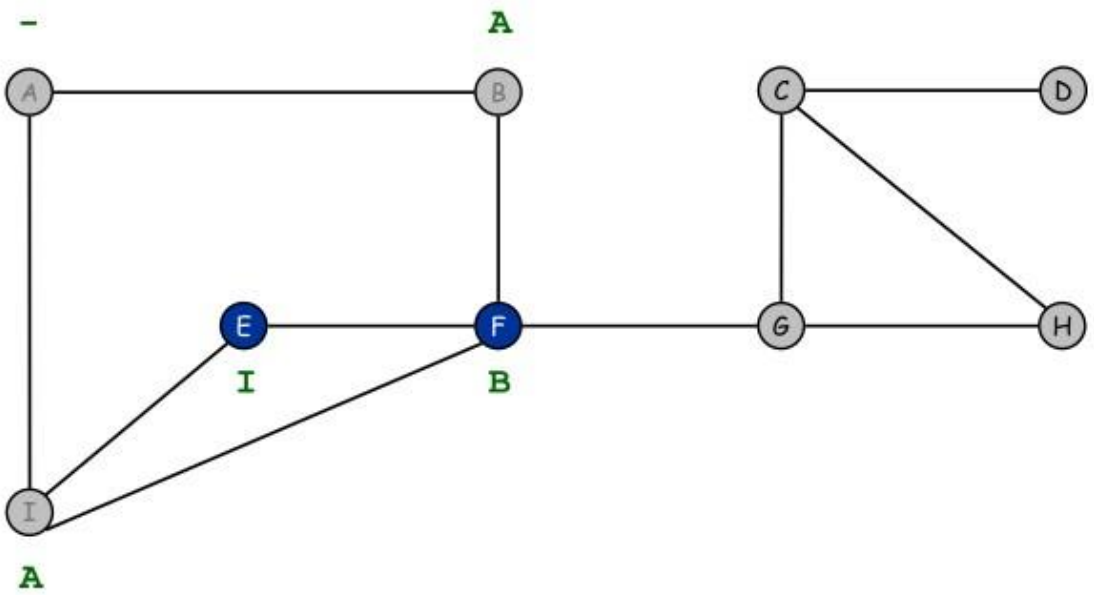


I finished

front **F E**

FIFO Queue

Breadth First Search

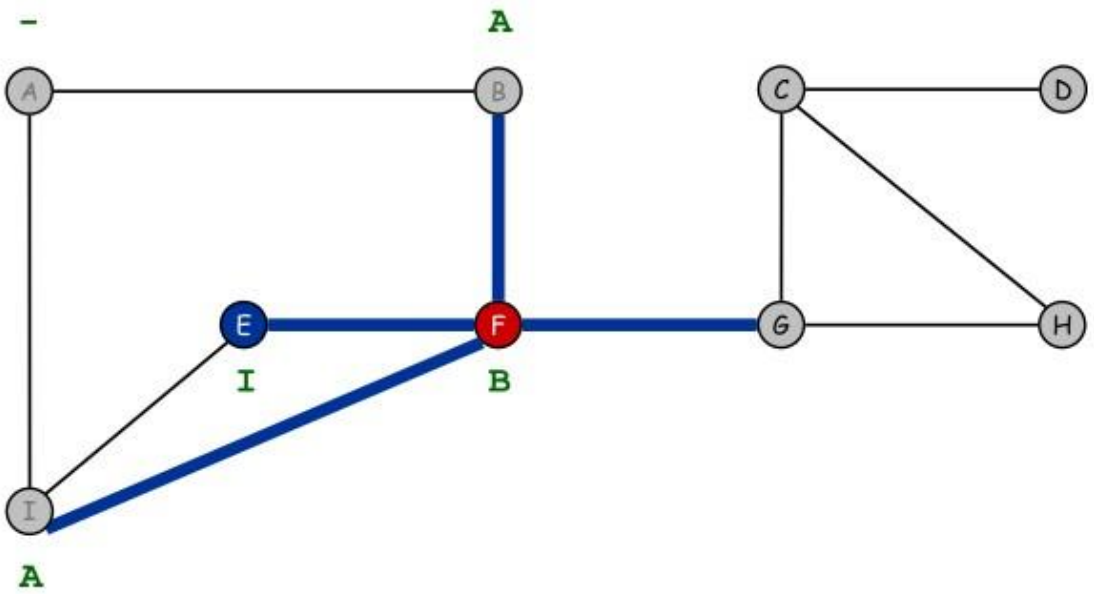


dequeue next vertex

front **F E**

FIFO Queue

Breadth First Search

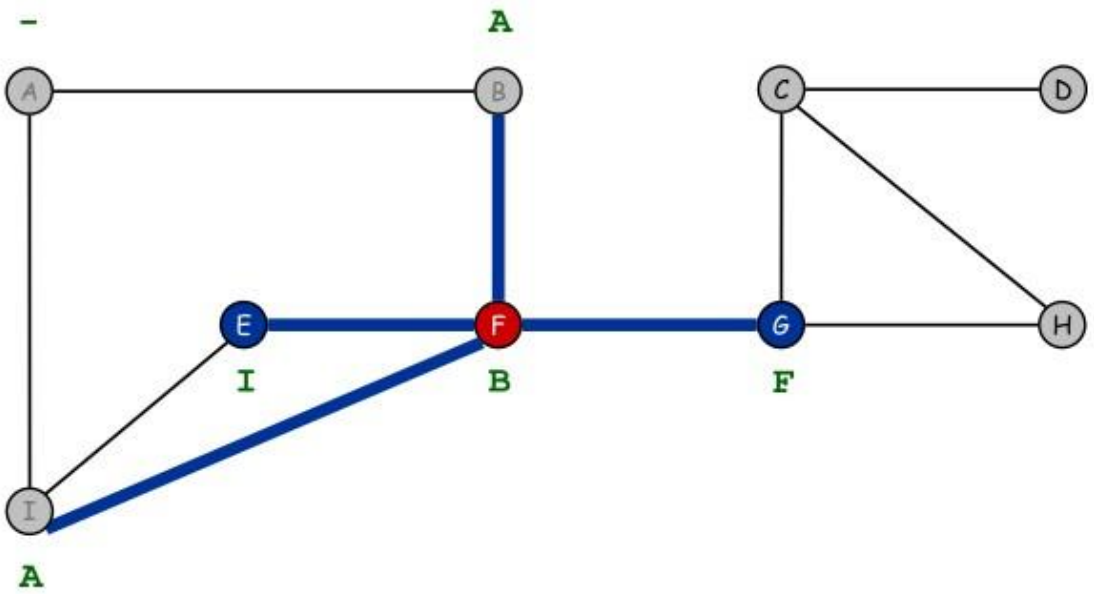


visit neighbors of F

front **E**

FIFO Queue

Breadth First Search

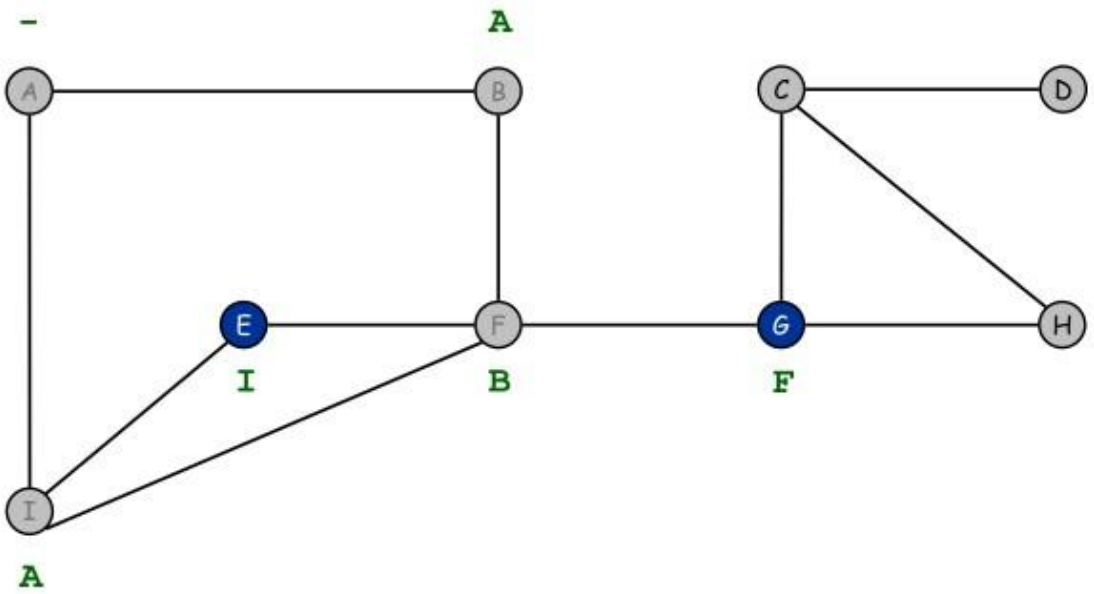


G discovered

front **E G**

FIFO Queue

Breadth First Search

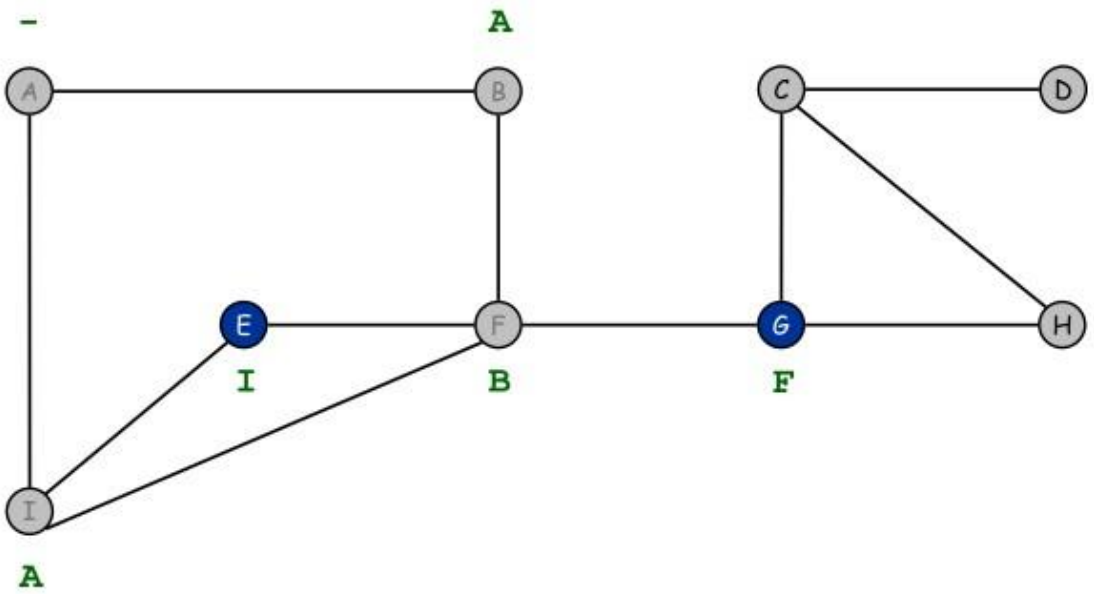


F finished

front **E G**

FIFO Queue

Breadth First Search

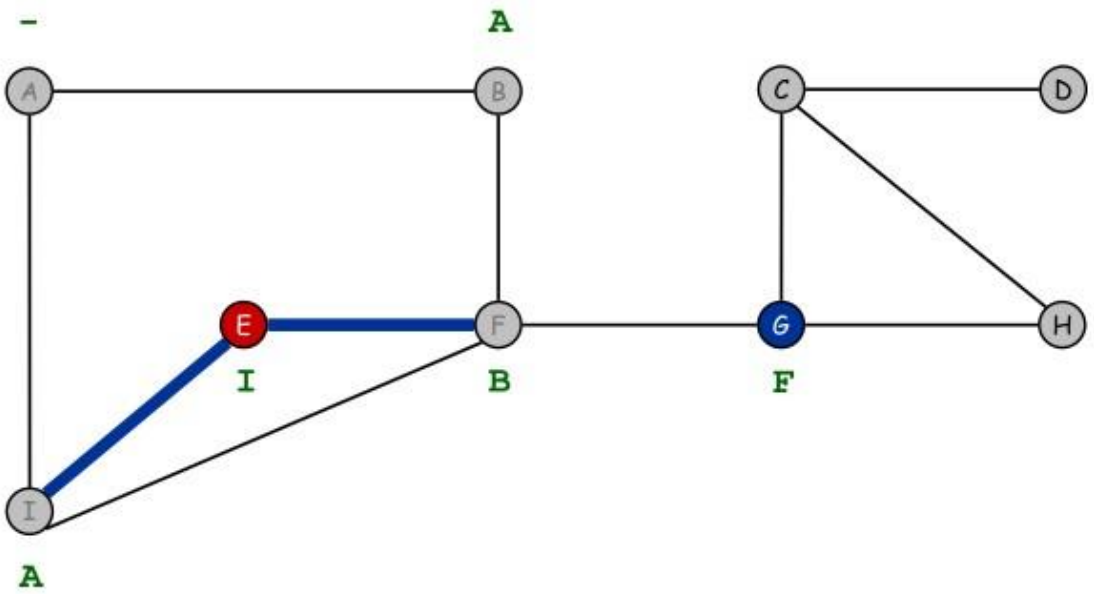


dequeue next vertex

front **E G**

FIFO Queue

Breadth First Search

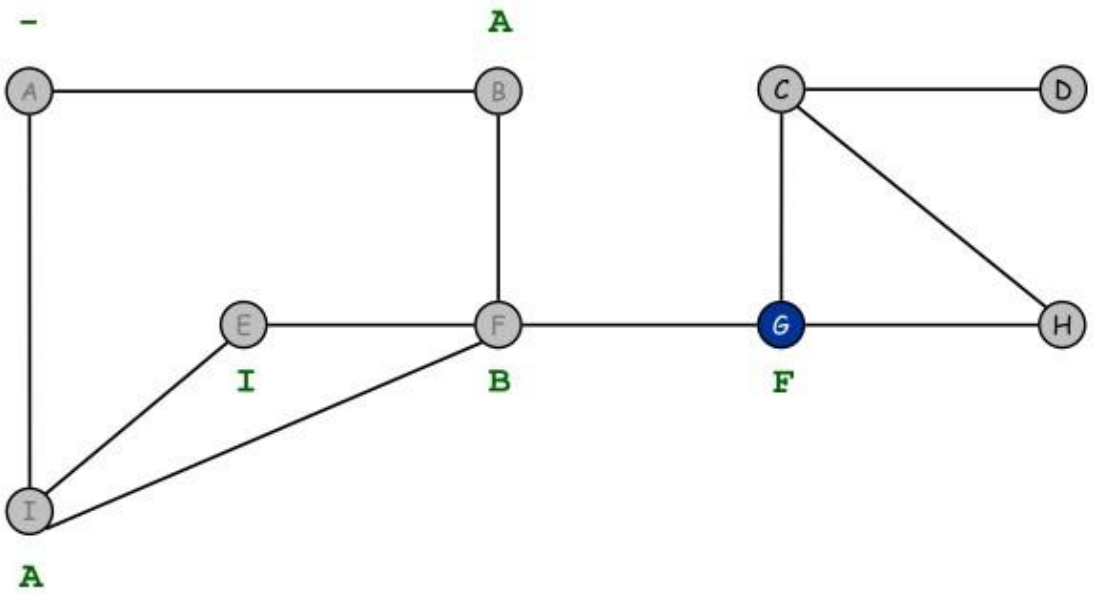


visit neighbors of E

front **G**

FIFO Queue

Breadth First Search

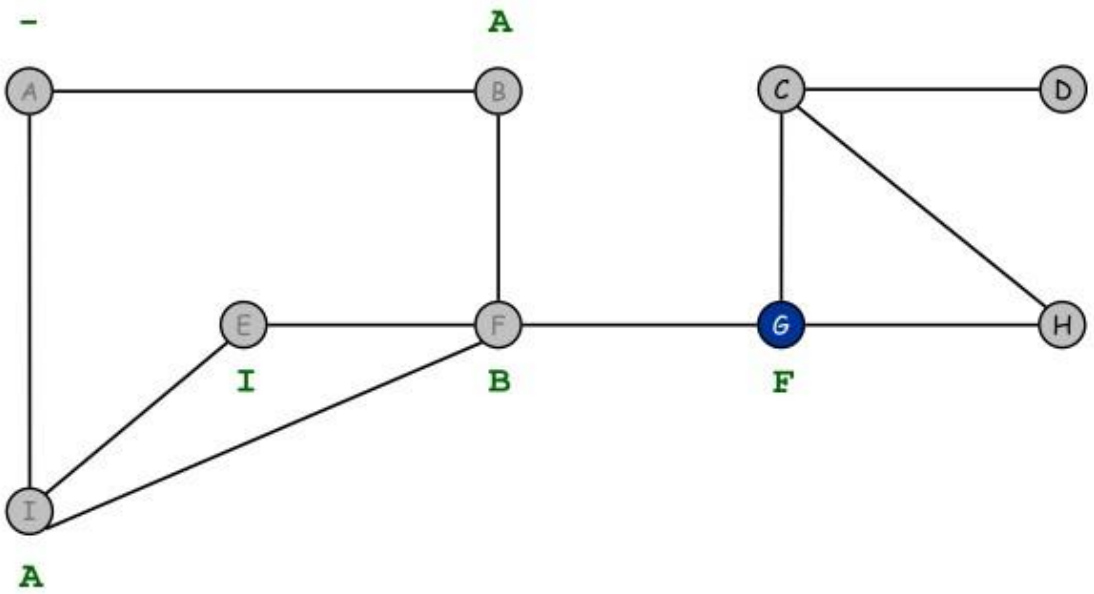


E finished

front G

FIFO Queue

Breadth First Search

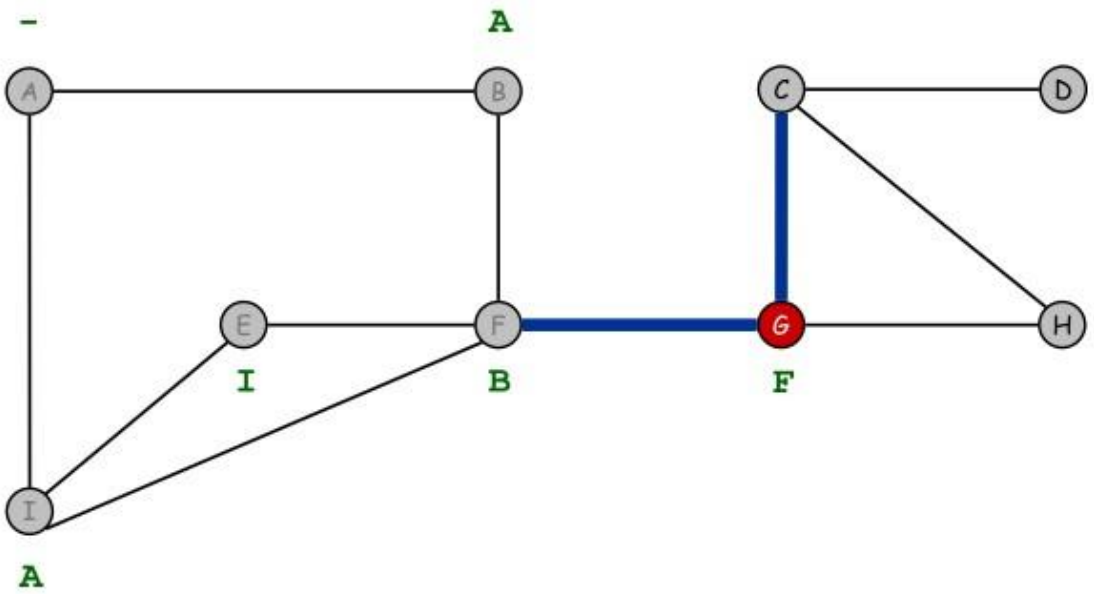


dequeue next vertex

front **G**

FIFO Queue

Breadth First Search

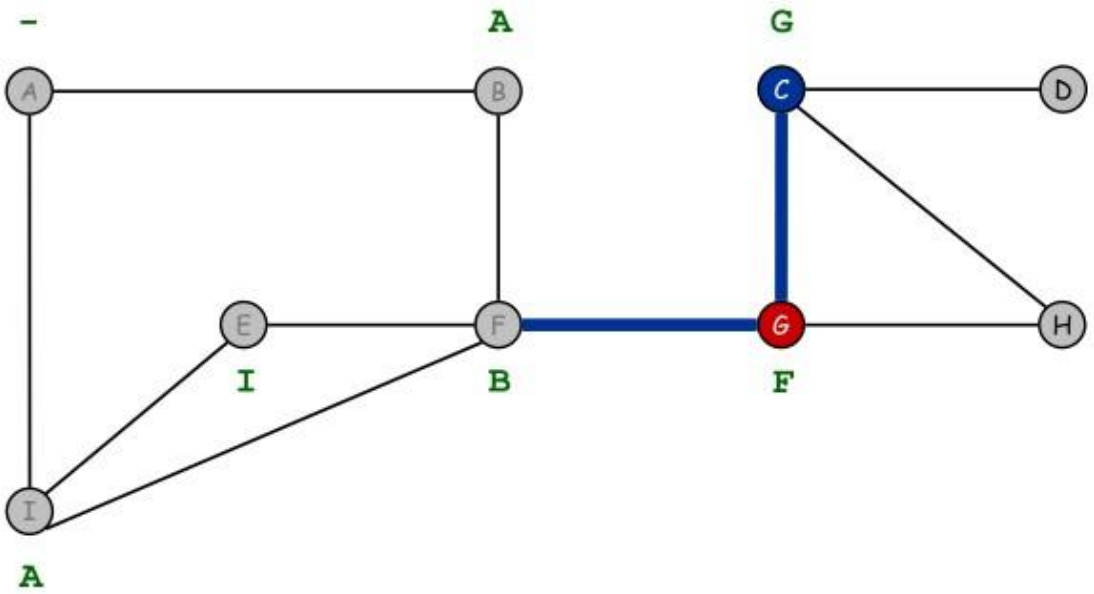


visit neighbors of G

front

FIFO Queue

Breadth First Search

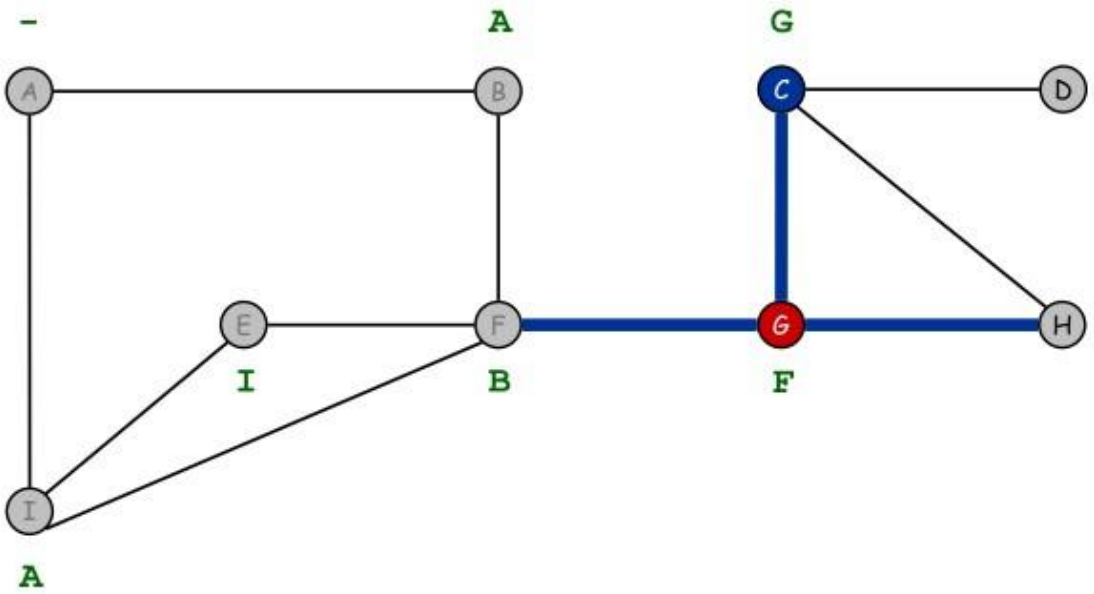


C discovered

front C

FIFO Queue

Breadth First Search

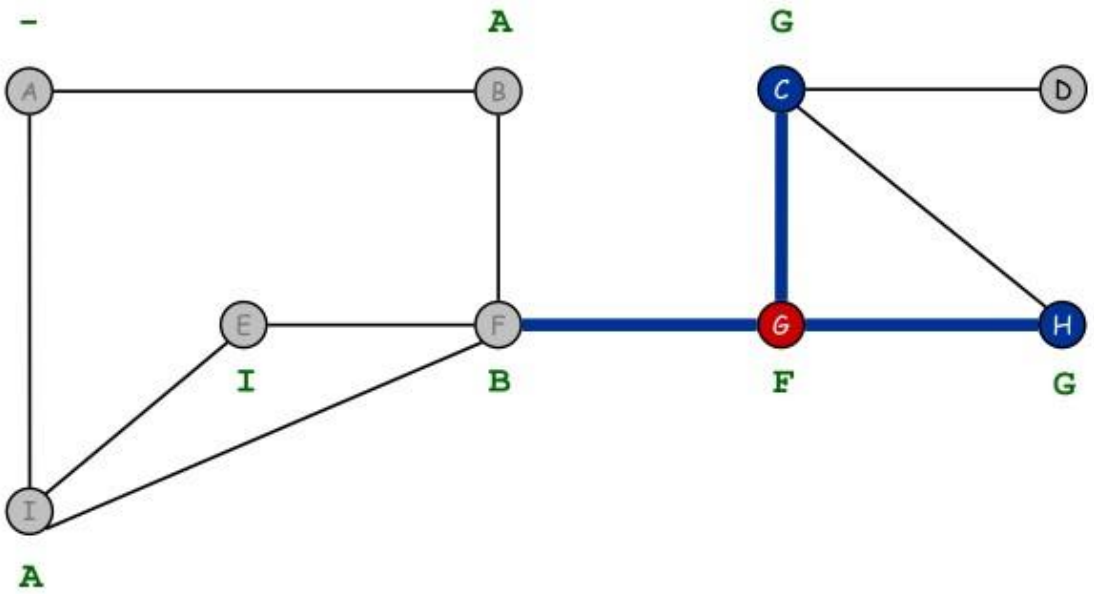


visit neighbors of *G*

front **C**

FIFO Queue

Breadth First Search

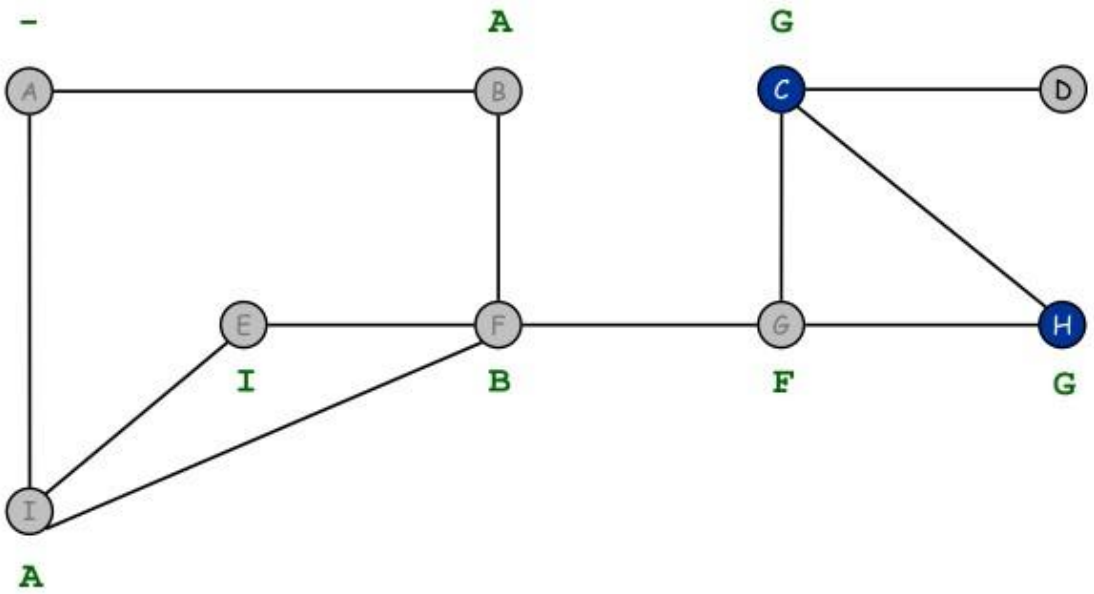


H discovered

front **C H**

FIFO Queue

Breadth First Search

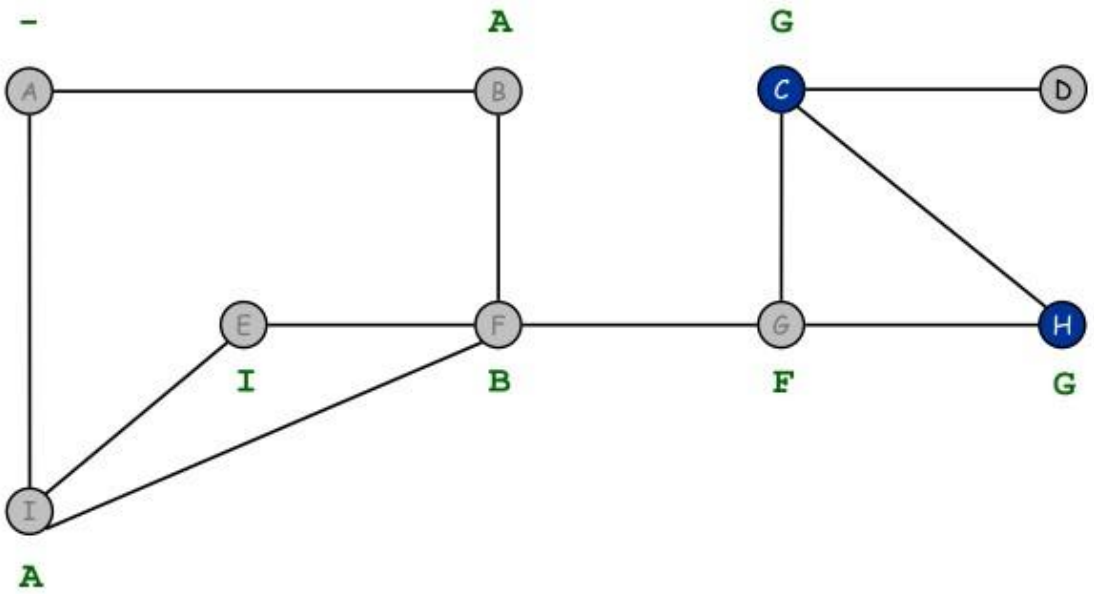


G finished

front **C H**

FIFO Queue

Breadth First Search

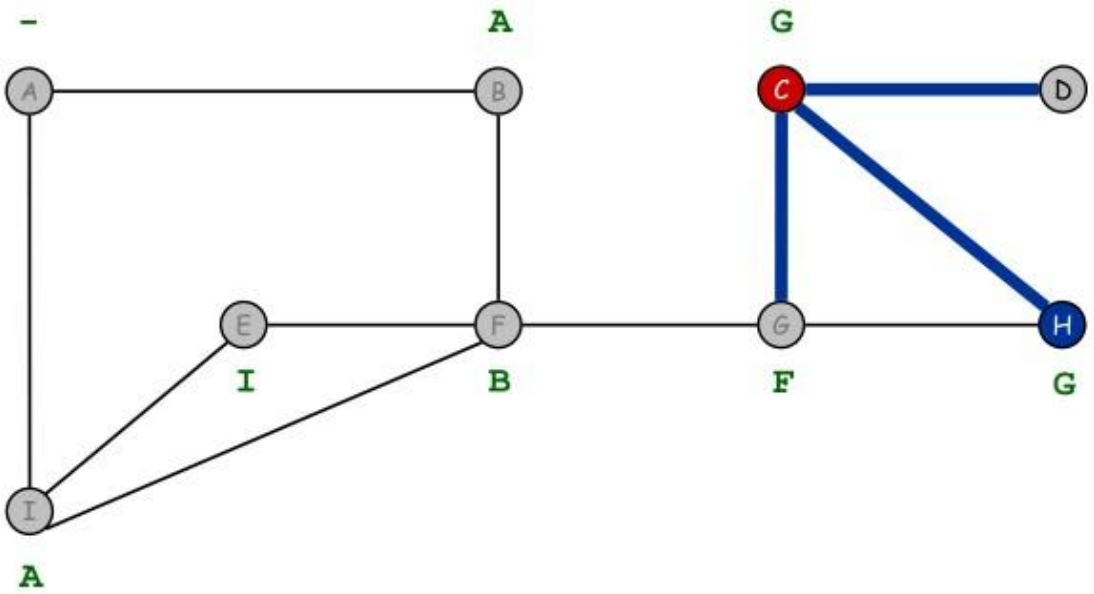


dequeue next vertex

front **C H**

FIFO Queue

Breadth First Search

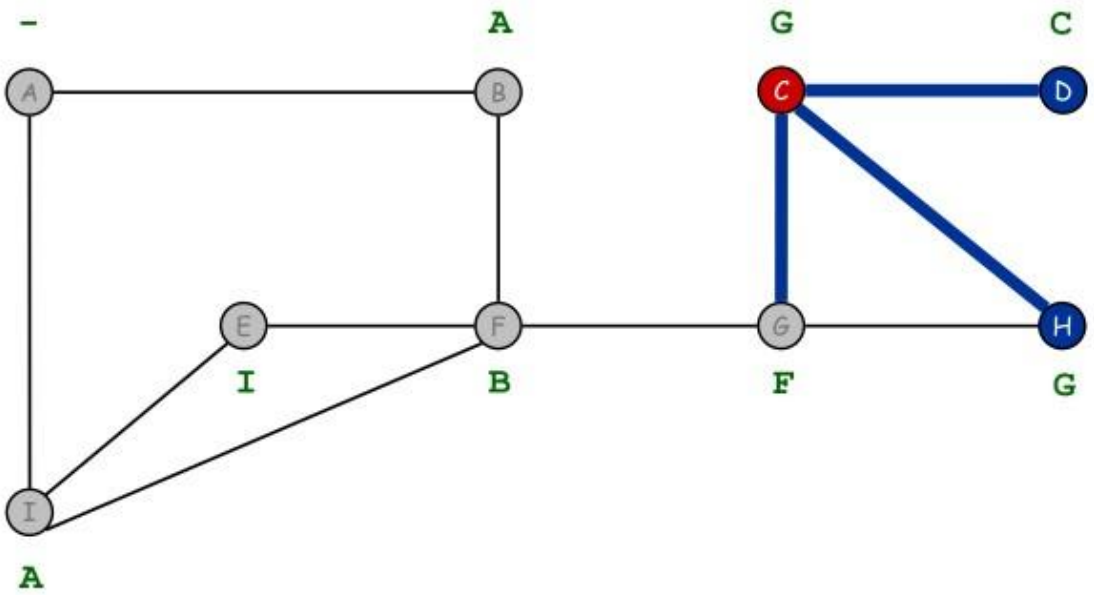


visit neighbors of *C*

front **H**

FIFO Queue

Breadth First Search

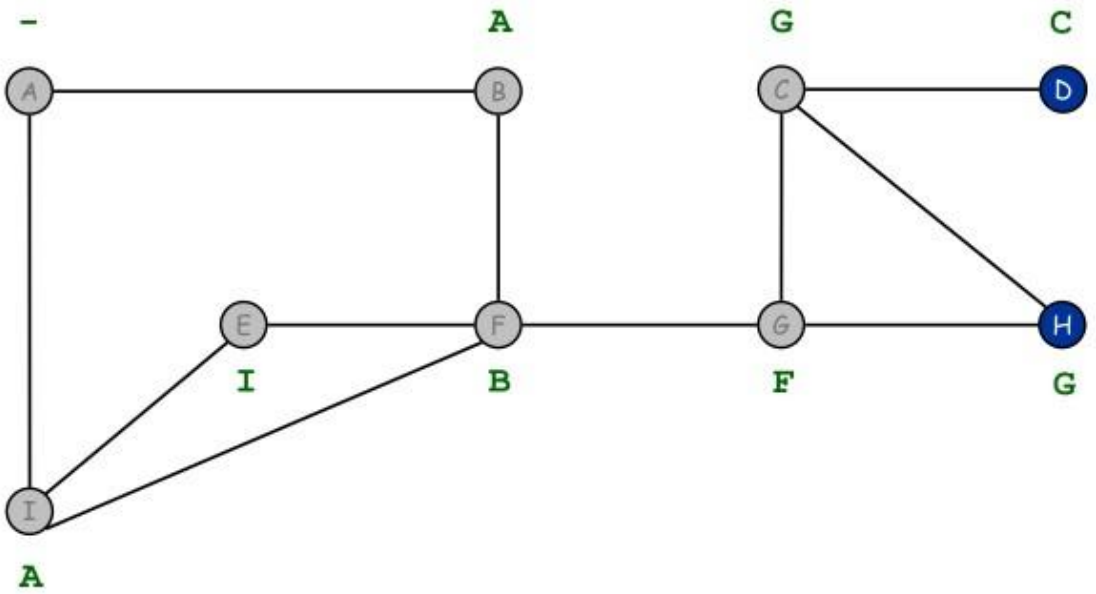


D discovered

front **H D**

FIFO Queue

Breadth First Search

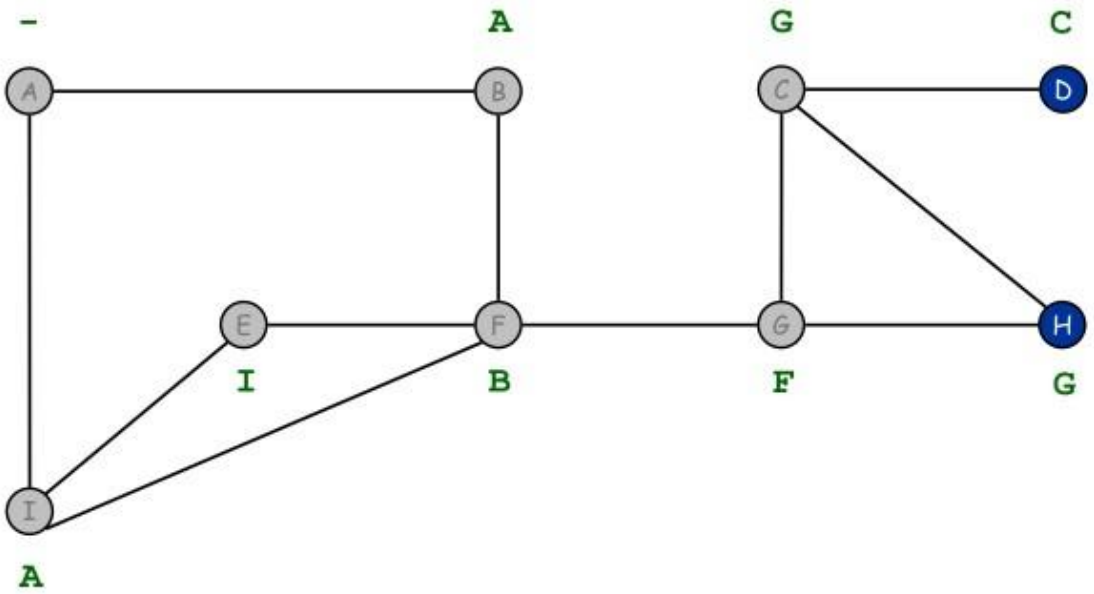


C finished

front **H D**

FIFO Queue

Breadth First Search

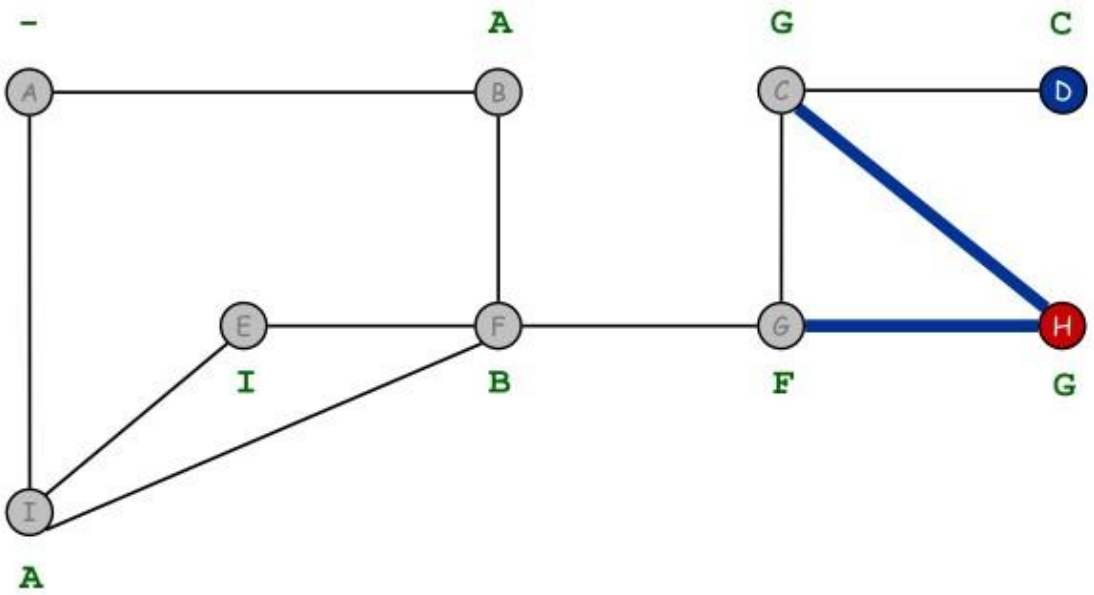


get next vertex

front **H D**

FIFO Queue

Breadth First Search

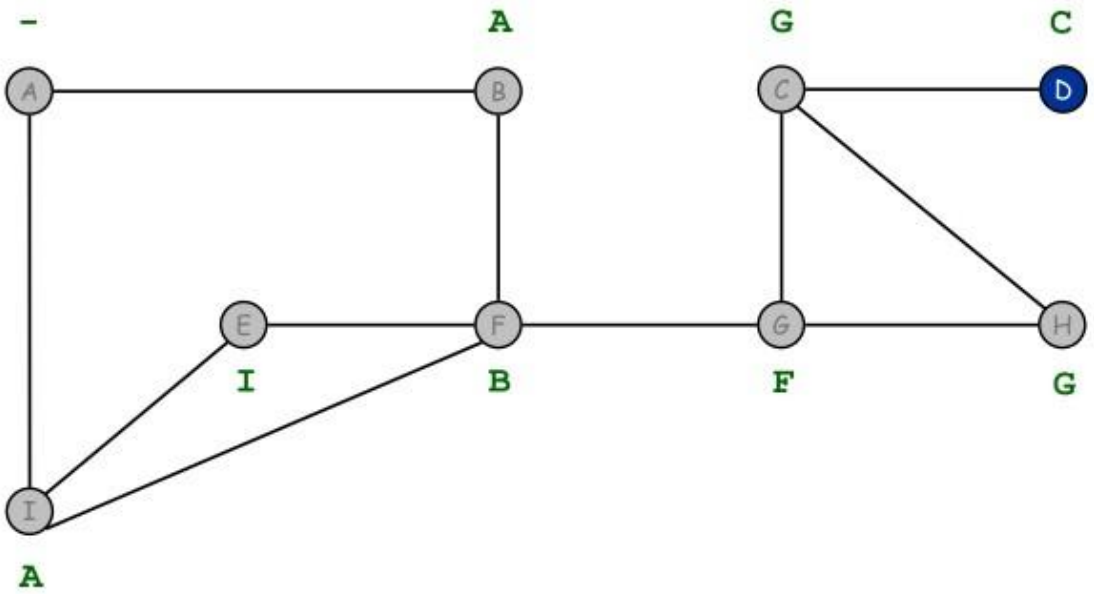


visit neighbors of H

front **D**

FIFO Queue

Breadth First Search

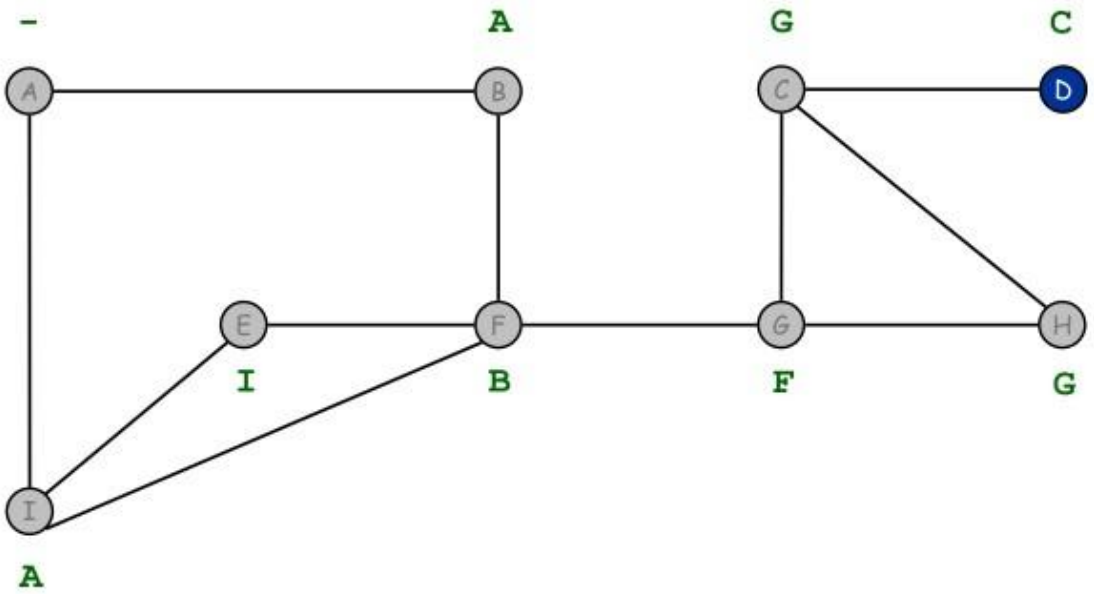


finished H

front **D**

FIFO Queue

Breadth First Search

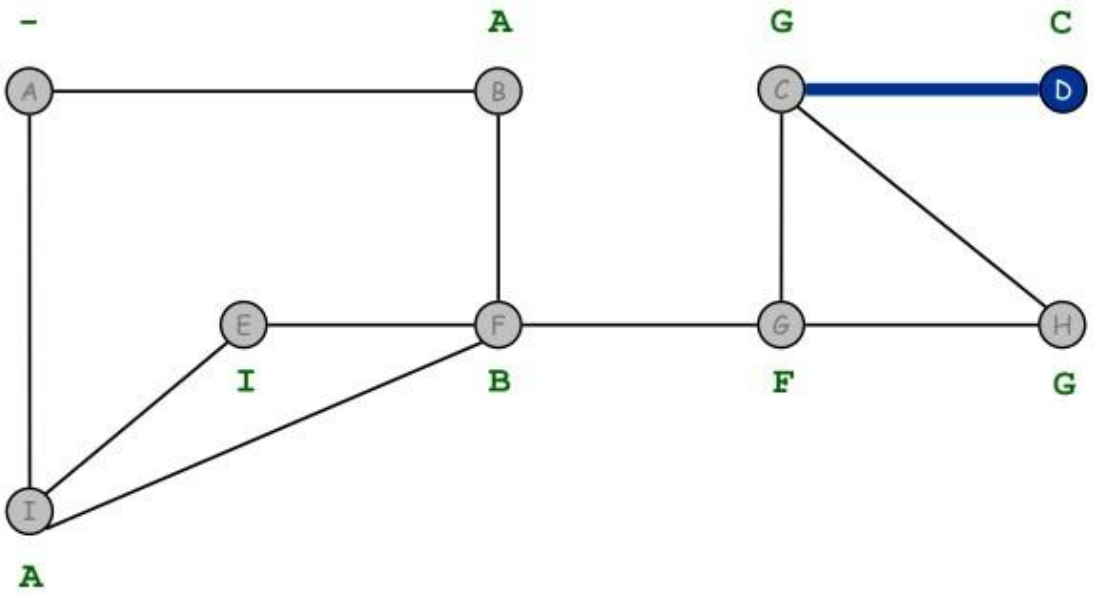


dequeue next vertex

front **D**

FIFO Queue

Breadth First Search

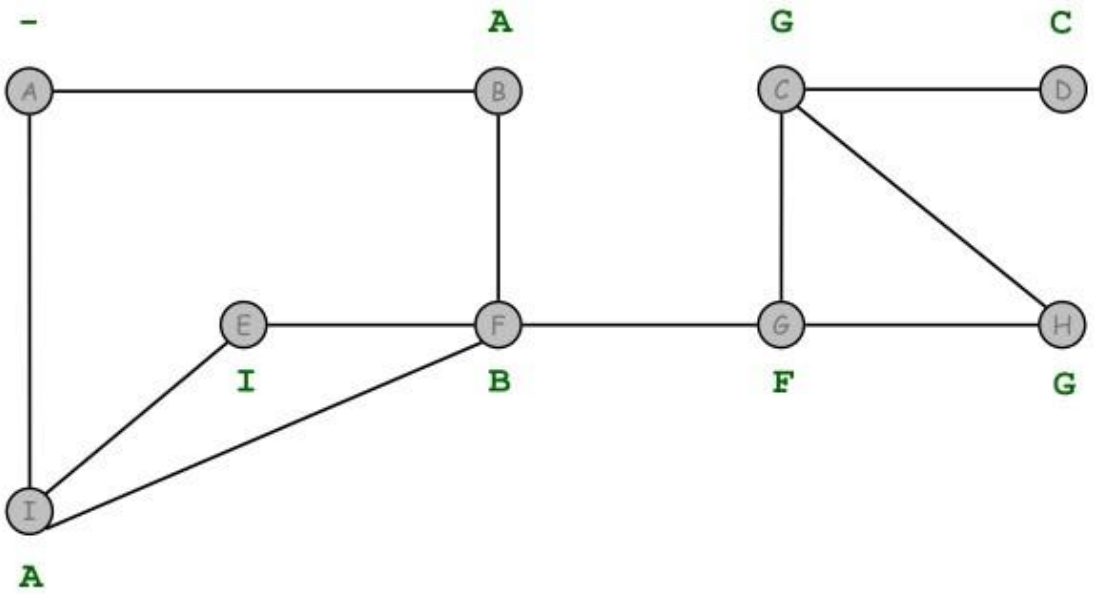


visit neighbors of D

front

FIFO Queue

Breadth First Search

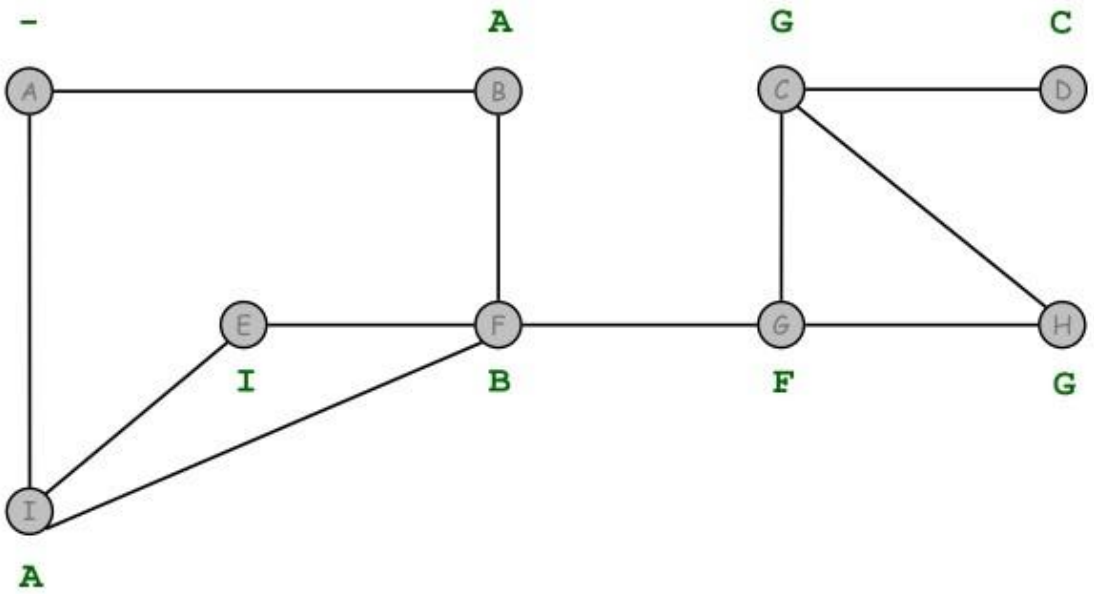


D finished

front

FIFO Queue

Breadth First Search

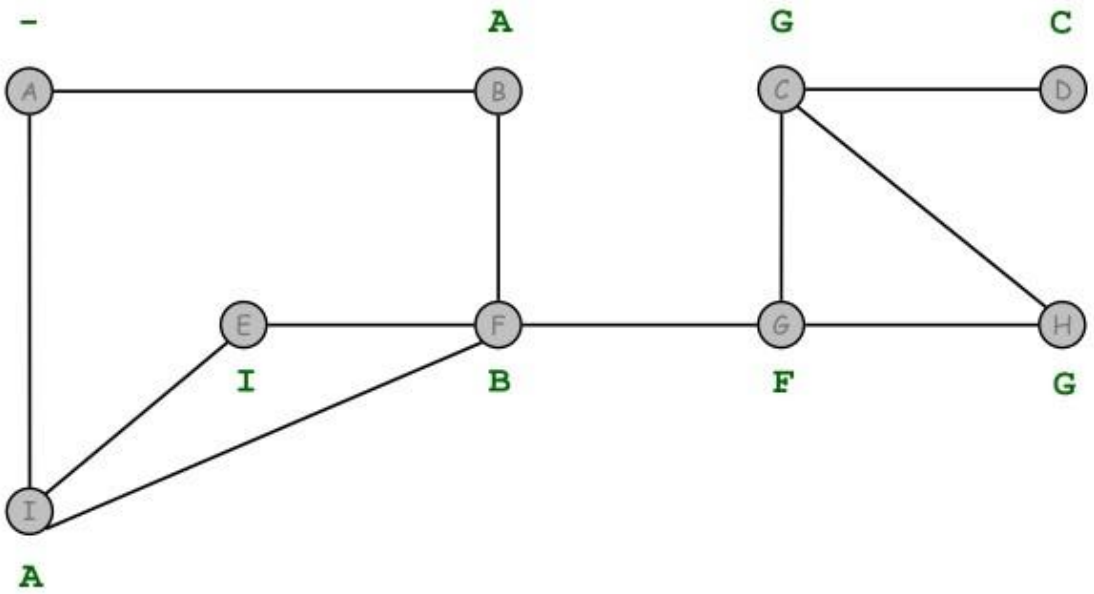


dequeue next vertex

front

FIFO Queue

Breadth First Search

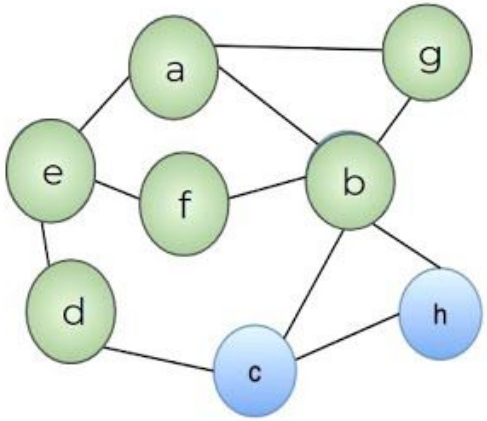


STOP

front

FIFO Queue

Breadth First Search



Graph

a	e	b	g		
b	a	g	f	c	h
c	b	h	d		
d	c	e			
e	a	d	f		
f	e	b			
g	a	b			
h	b	c			

Adjacency List

BFS order

a e b g d f c h

Visited

a e b g d f

Q ⇦ b g d f ⇦

v e

BFS

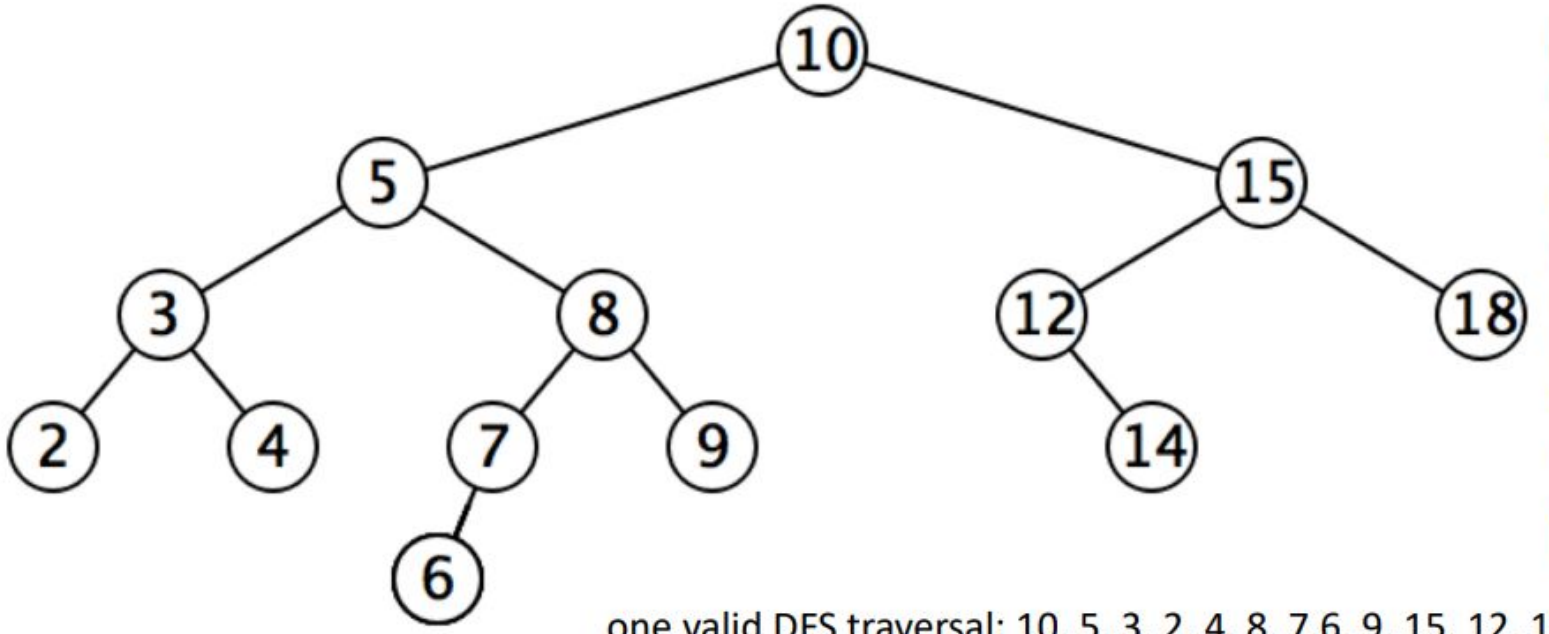
add start vertex to Q
mark start as visited

while Q not empty
    v ← dequeue from Q
    for each adj vertex av of v
        //.... process vertex av....
        if av is not visited
            add av to Q
            mark av as visited

Graph traversals:

Depth First Search - a traversal on graphs (or on trees since those are also graphs) where you traverse "deep nodes" before all the shallow ones

High-level DFS: you go as far as you can down one path till you hit a dead end (no neighbors are still undiscovered or you have no neighbors). Once you hit a dead end, you backtrack / undo until you find some options/edges that you haven't actually tried yet.



Kind of like wandering a maze – if you get stuck at a dead end (since you physically have to go and try it out to know it's a dead end), trace your steps backwards towards your last decision and when you get back there, choose a different option than you did before.

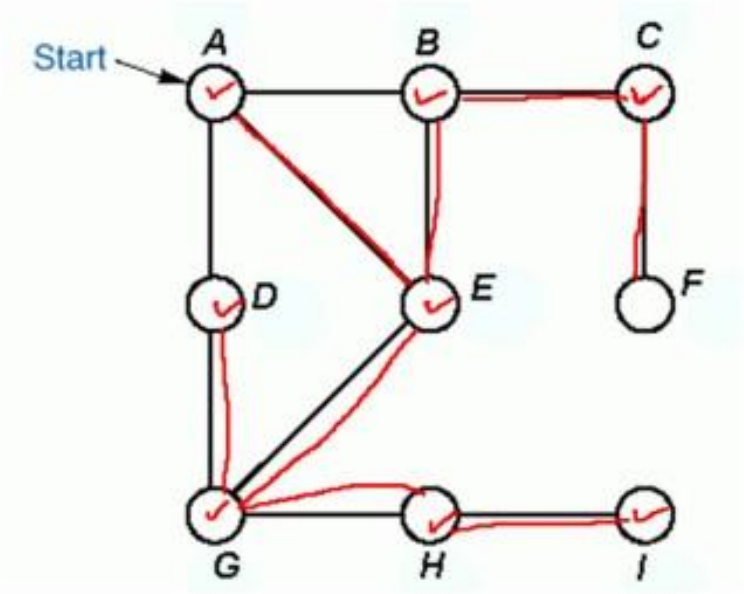
one valid DFS traversal: 10, 5, 3, 2, 4, 8, 7,6, 9, 15, 12, 14, 18

Depth-First Traversal

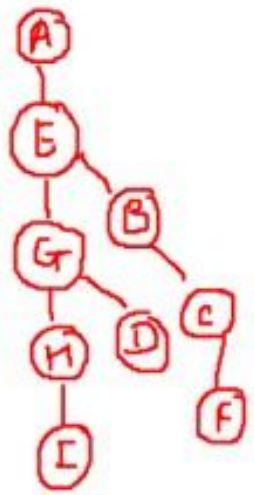
- A DFS starting at a vertex v first visits v, then some neighbour w of v, then some neighbour x of w that has not been visited before, etc.
- When it gets stuck, the DFS backtracks until it finds the first vertex that still has a neighbour that has not been visited before.
- It continues with this neighbour until it has to backtrack again.
- Eventually, it will visit all vertices reachable from v
- Must keep track of vertices already visited to avoid cycles.
- The method can be implemented using recursion or iteration.
- A DFS traversal of a graph results in a depth-first tree or in a forest of such trees.

Example

Depth-first traversal using an explicit stack.



Dfs Tree



Stack

stack is empty ∴ stop

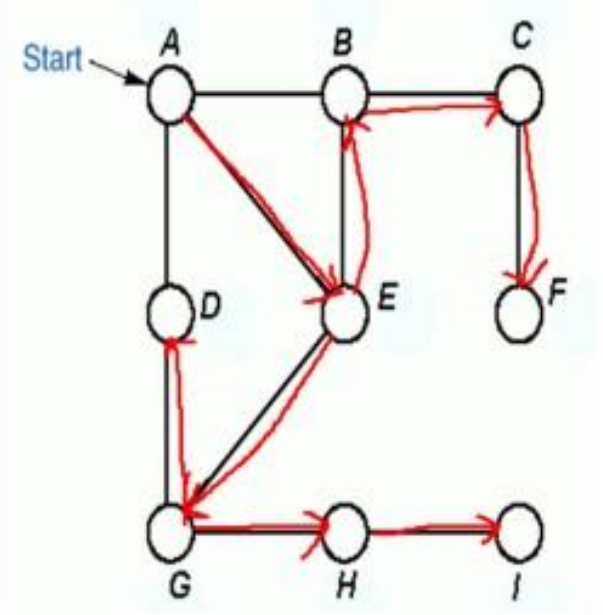
Order of Traversal	1	2	3	4	5	6	7	8	9
	A	G	H	I	D	B	C	F	

Visited array	A	B	C	D	E	F	G	H	I
	✓	✓	✓	✓	✓	✓	✓	✓	✓

Note: The DFS-tree for undirected graph is a free tree

DFS Pseudo code

```
dfs (v)
 initialize stack s
initialize visited of all vertices to false
visited [v] = true
push(s,v)
while not empty(s)
    v=pop(s)
    add v into dfs sequence
   for each vertex w adjacent to v
      if not visited [w] then
           push(s,w)
           visited [w]=true
```



Order of Traversal

1	2	3	4	5	6	7	8	9
A	E	G	H	I	D	B	C	F

Stack

Visited array

A	B	C	D	E	F	G	H	I
✓	✓	✓	✓	✓	✓	✓	✓	✓

Pseudo code (Iterative)

```
DFS(Node start)
  initialize stack s to hold start
  mark start as visited
  while(s is not empty)
     next = s.pop() // and "process"
     for each node u adjacent to
next
        if(u is not marked)
          mark u and push onto s
  endwhile
enddfs
```

```
BFS(Node start)
  initialize queue q to hold start
  mark start as visited
  while(q is not empty)
     next = q.dequeue() // and
"process"
     for each node u adjacent to next
       if(u is not marked)
         mark u and enqueue onto q
  endwhile
Endbfs
```

When to use DFS or BFS to solve a Graph problem?

Most of the problems can be solved using either BFS or DFS. It won't make much difference. For example, consider a very simple example where we need to **count the total number of cities connected to each other**. Where in a graph nodes represent cities and edges represent roads between cities.

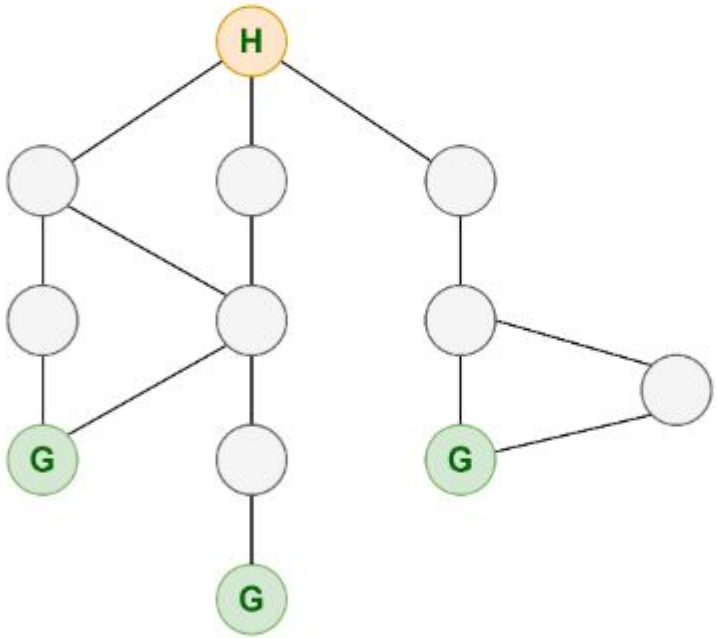
In such a problem, we know that we need to go to every node in order to count the nodes. **So it doesn't matter if we use BFS or DFS as using any of the ways we will be traversing all the edges and nodes of a graph.**

But there are problems when we need to decide to either use DFS or BFS for a **faster solution**. And there is no generalization. **It completely depends on the problem definition. It depends on what we are trying to find in the solution.** We need to understand clearly what our problem wants us to find. And the problem might not directly tell us to use BFS or DFS.

Examples of choosing DFS over BFS.

Example 1:

Consider a problem where you are standing at your house and you have multiple ways to go from your house to a grocery store. You are said that every path you choose has one store and is located at the end of every path. You just need to reach any of the stores.



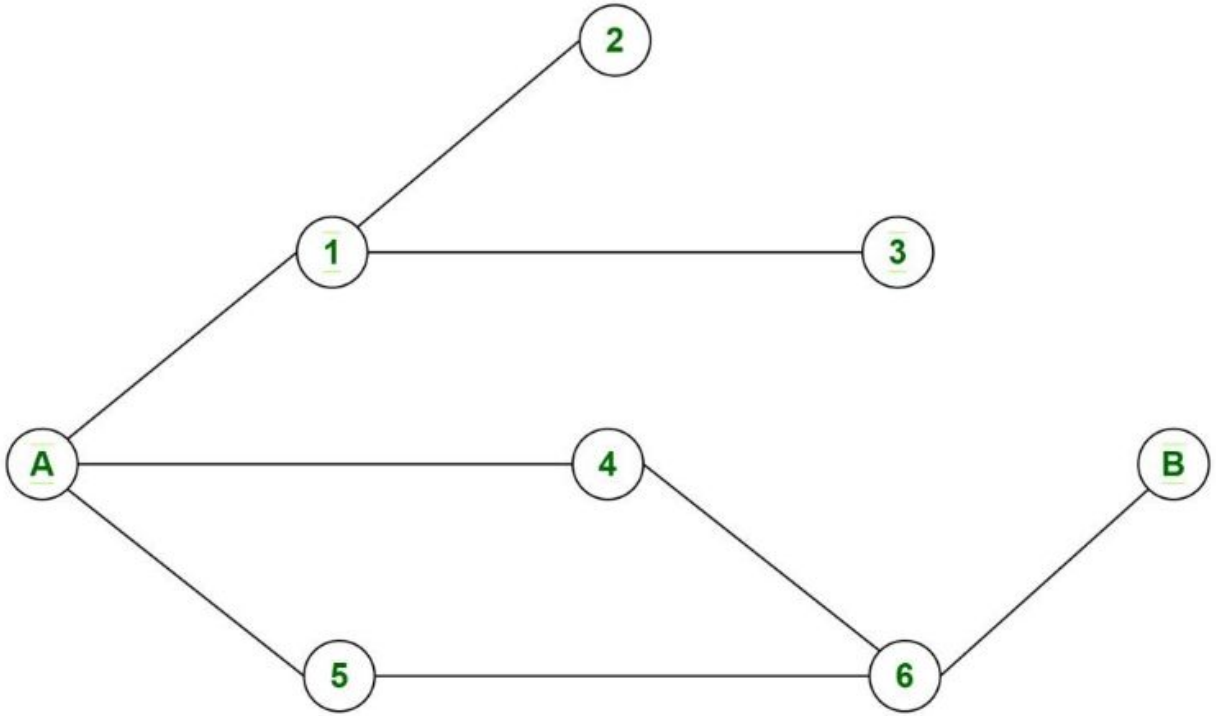
Representation showing path from room to grocery store

The obvious method here will be to choose **DFS**.

As we know we can find our solution (grocery store) in any of the paths, we can just go on traversing to any neighbor of the current node without exploring all the neighbors. There is no need of going through BFS as it will unnecessarily explore other paths but we can find our solution by traversing any of the paths. Also, we know that our solution is situated farthest from the starting point so if we choose BFS then we will have to almost visit all the nodes as we are visiting all nodes of a level and we will keep doing it till the end where we find a grocery store.

Example 2:

Consider a problem where you need to print all the nodes encountered in any one of the paths starting from node A to node B in the diagram.



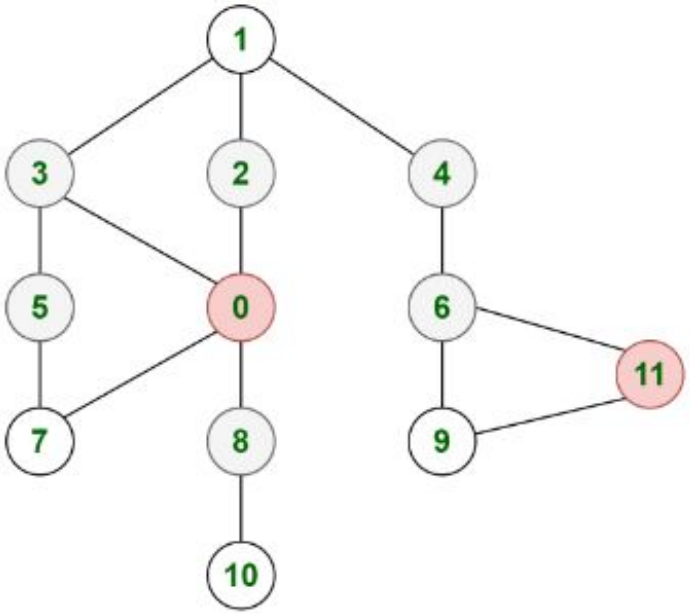
A graph

Here there are two possible paths “A -> 4 -> 6 -> B” and “A -> 5 -> 6 -> B”. Here we require to keep track of a single path so there is no need of exploring every other path using BFS. Also, not every path will lead us from A to B. So we need to backtrack to the current node and then explore another path and see if that leads us to B. Need for backtracking tells us that we can think in the DFS direction.

Examples of choosing BFS over DFS.

Example 1:

Consider an example of a graph representing connected cities through edges. There are a few nodes colored in red that indicates covid affected cities. White-colored nodes indicate healthy cities. You are asked to find out the time the covid virus will take to affect all the non-affected cities if it takes one unit of time to travel from one city to another.



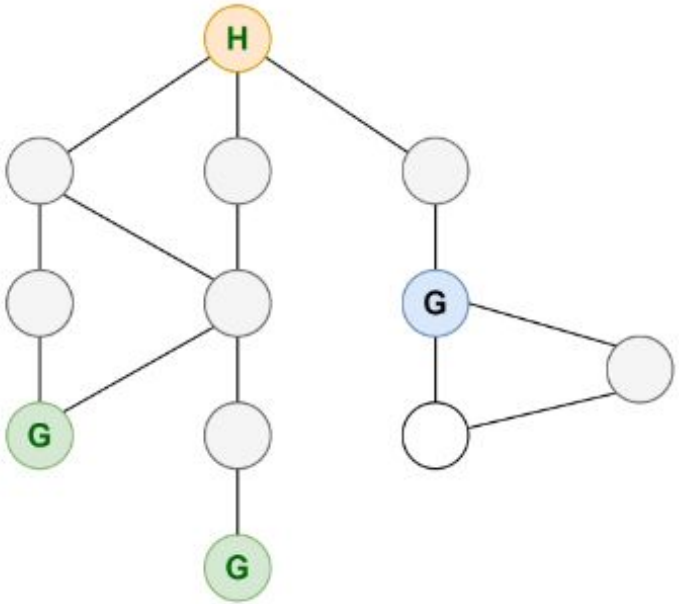
Graph representing above city arrangement

Here thinking of DFS is not even feasible. Here one affected city will affect all of its neighbors in one unit of time. This is how we know that we need to apply BFS as we need to explore all neighbors of the current node first. Another strong reason for BFS here is that both nodes 0 and 11 will start affecting neighbor cities simultaneously. **This shows we require a parallel operation on both nodes 0 and 11**. So we need to start traversing all the neighbor nodes of both nodes simultaneously. So we can push nodes 0 and 11 in the queue and start traversal parallelly. It will require 2 units of time for all the cities to get affected.

1. At time = 0 units, Affected nodes = {0, 11}
2. At time = 1 units, Affected nodes = {0, 11, 3, 2, 8, 7, 6, 9}
3. At time = 2 units, Affected nodes = {0, 11, 3, 2, 8, 7, 6, 9, 5, 1, 4, 10}

Example 2:

Consider the same example of house and grocery stores mentioned in the above section. Suppose now you need to find the nearest grocery store from the house instead of any grocery store. Consider that each edge is of 1 unit distance. Consider the diagram below:



Graph representing grocery store

Here using DFS like previous will not be feasible. If we use DFS then we will travel down a path till we don't find a grocery store. But once we have found it we are not sure if it is the grocery store at the shortest distance. So we need to backtrack to find a grocery store on other paths and see if any other grocery store has a distance less than the current found grocery store. This will lead us to visit every node in the graph which is not probably the best way to do it.

We can use **BFS** here as BFS traverses nodes level by level. We first check all the nodes at a 1-unit distance from the house. If any of the nodes is a grocery store then we can stop else we will see the next level i.e all the nodes at a distance 2-unit from the house and so on. This will take less time in most situations as we will not be traversing all the nodes. For the given graph we will only explore nodes up to two levels as at the second level we will find the grocery store and we will return the shortest distance to be 2.

Conclusion:

We can't have fixed rules for using BFS or DFS. It totally depends on the problem we are trying to solve. But we can make some general intuition.

- We will prefer to use BFS when we know that our solution might lie closer to the starting point or if the graph has greater depths.
- We will prefer to use DFS when we know our solution might lie farthest from the starting point or when the graph has a greater width.
- If we have multiple starting points and the problem requires us to start traversing all those starting points parallelly then we can think of BFS as we can push all those starting points in the queue and start exploring them first.
- It's generally a good idea to use BFS if we need to find the shortest distance from a node in the unweighted graph.
- We will be using DFS mostly in path-finding algorithms to find paths between nodes.

VLAB Link for experiment

DFS:

https://ds1-iiith.vlabs.ac.in/exp/depth-first-search/index.html

BFS:

https://ds1-iiith.vlabs.ac.in/exp/breadth-first-search/index.html