INTERVIEW
KICKSTART

Graph Algorithms

{ik} INTERVIEW KICKSTART

Bio - Dr. Li Wang

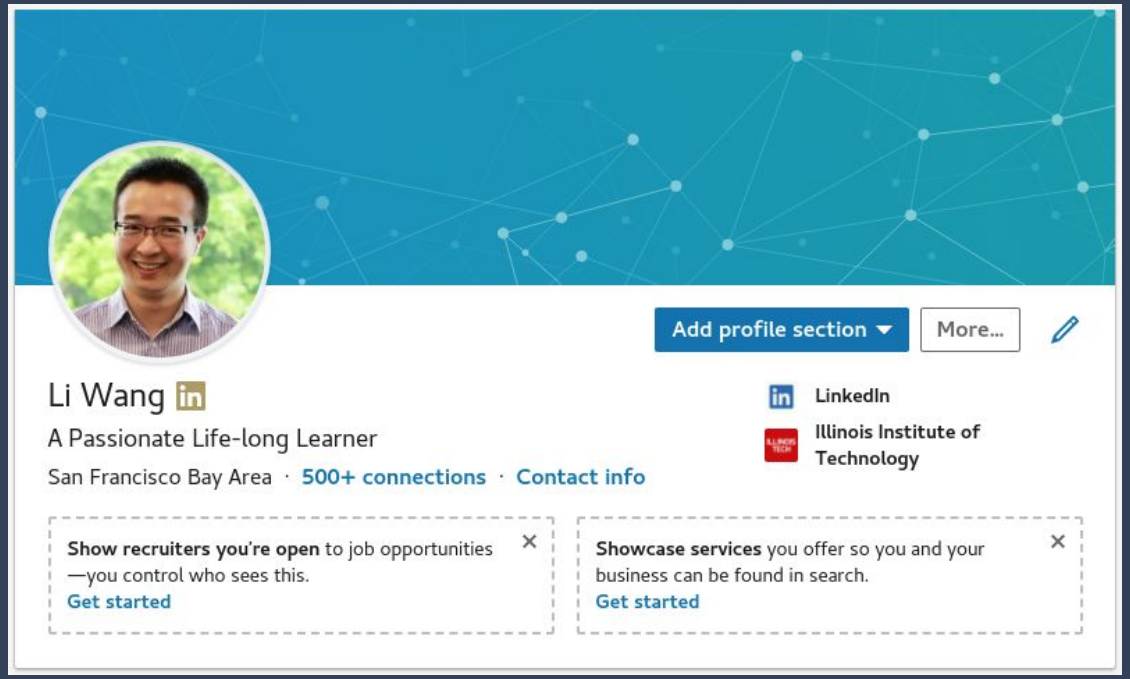
Staff Software Engineer @ LinkedIn

Professor @ DeVry University

Ph.D in CS @ Illinois Institute of Technology, US

B.S & M.S in CS @ Chongqing University, China

Let’s Connect on LinkedIn!



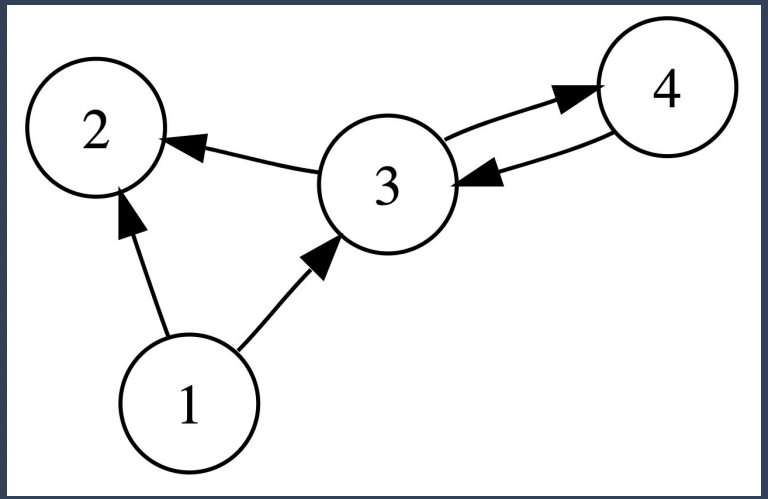
https://www.linkedin.com/in/li-wang-67049615/

{ik} INTERVIEW KICKSTART

How to Solve Graph Interview Questions

- First determine whether the problem can be modeled as a graph problem.
- Graph and its representation
- What are the most common graph problems
- How to solve graph problems
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)
  - Topological Sort
  - Dijkstra’s algorithm (optional)
  - Prim's algorithm (optional)
  - Kruskal's algorithm (optional)
  - ……

Graph Representation

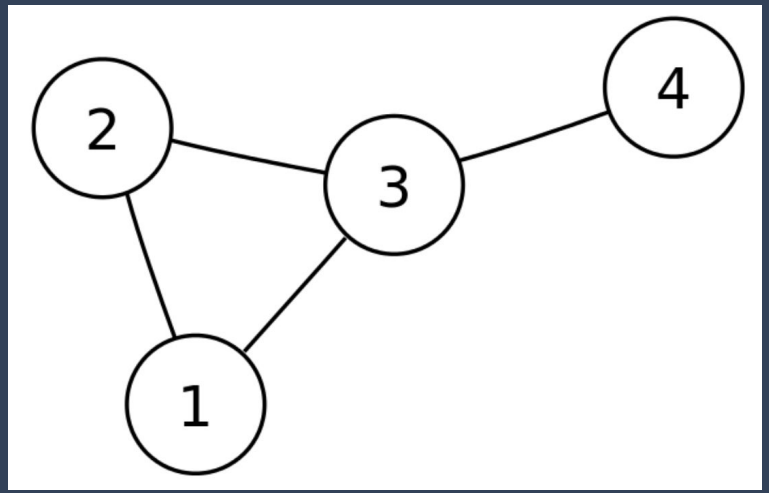


Directed Graph

***G* = (*V*, *E*)**

V = {1, 2, 3, 4}

E = {(1, 2), (1, 3), (3, 2), (3, 4), (4, 3)}



Undirected Graph

***G* = (*V*, *E*)**

V = {1, 2, 3, 4}

E = {(1, 2), (2, 1), (1, 3), (3, 1), (3, 2), (2, 3), (3, 4), (4, 3)}

Question

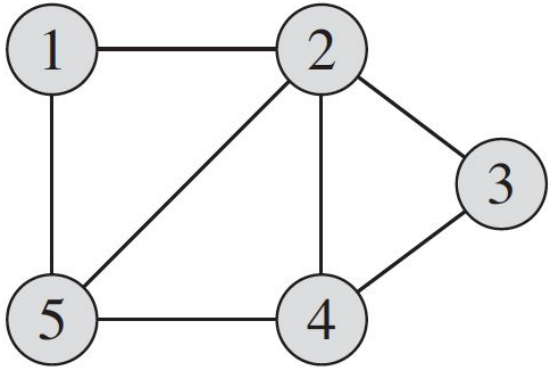
- Given a graph below

  $G = (V, E)$
  $V = \{1, 2, 3, 4\}$
  $E = \{(1, 2), (1, 3), (3, 2), (3, 4), (4, 3)\}$

- Questions:
  - How do we know whether vertex A can reach vertex B **directly** or not?
  - What are the neighbors of a vertex?

- Time complexity of both operations: O(|E|)

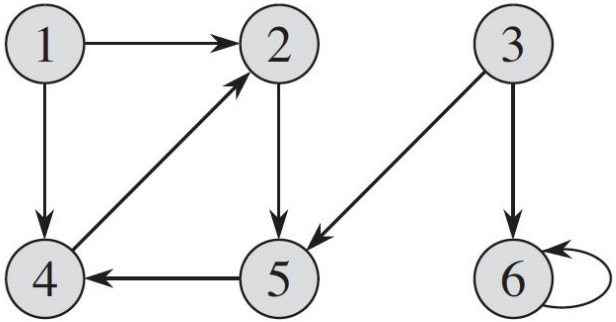
Adjacency Matrix (Undirected Graph)



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

Introduction to Algorithms, 3rd Edition by Thomas H. Cormen, Charles E. Leiserso, Ronald L. Rivest, Clifford Stein

Adjacency Matrix (Directed Graph)

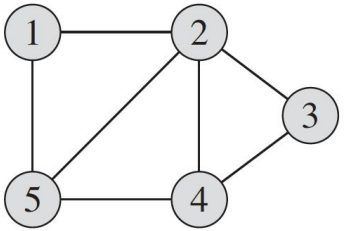


	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Introduction to Algorithms, 3rd Edition by Thomas H. Cormen, Charles E. Leiserso, Ronald L. Rivest, Clifford Stein

Adjacency Matrix Implementation

```
class Graph {
        int[][] matrix;
        int n;

        public Graph(int n) {
                this.n = n;
                this.matrix = new int[n][n];
        }
        void addEdge(int start, int end) {
                matrix[start][end] = 1;
        }
}
```



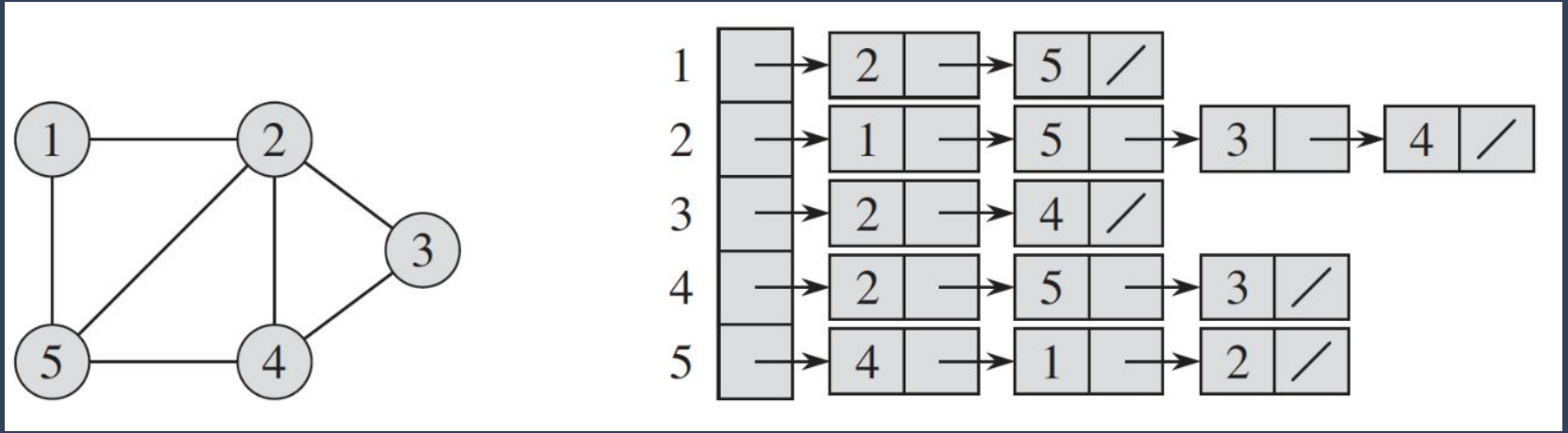
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Adjacency Matrix

- Time Complexity
  - O(1) time to check whether 2 vertices are **directly** connected or not
  - O(|V|) time to get all neighbours of a vertex

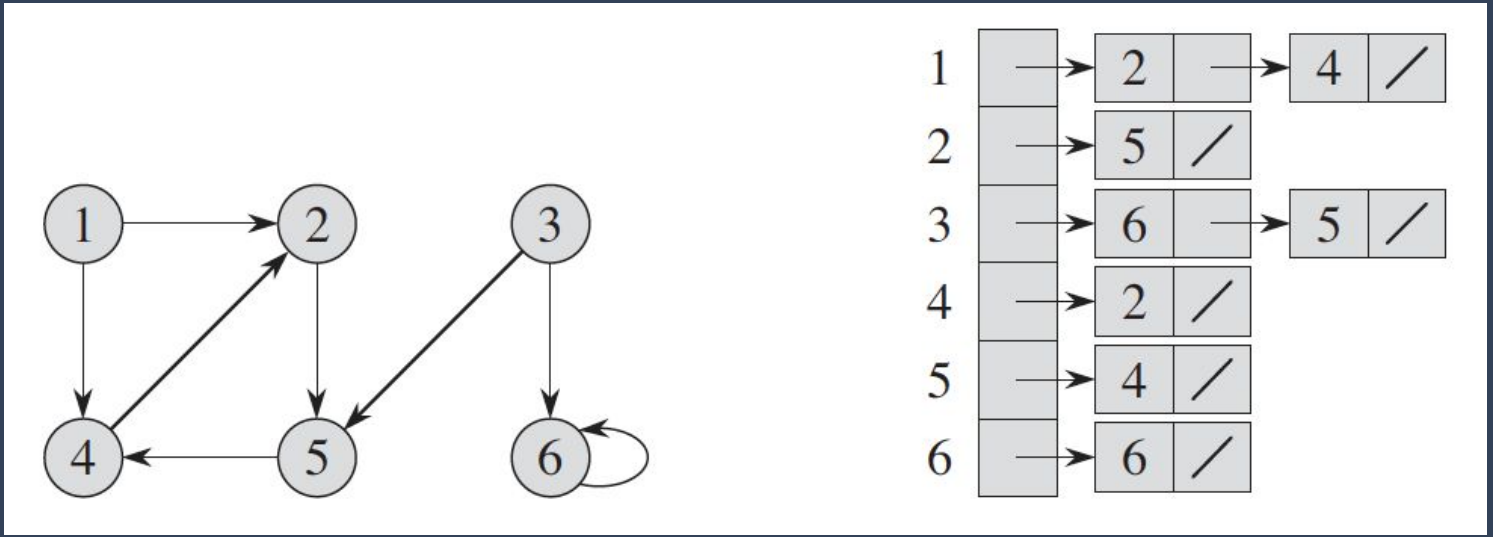
- Space Complexity
  - $O(|V|^2)$ space

Adjacency List (Undirected Graph)



Introduction to Algorithms, 3rd Edition by Thomas H. Cormen, Charles E. Leiserso, Ronald L. Rivest, Clifford Stein

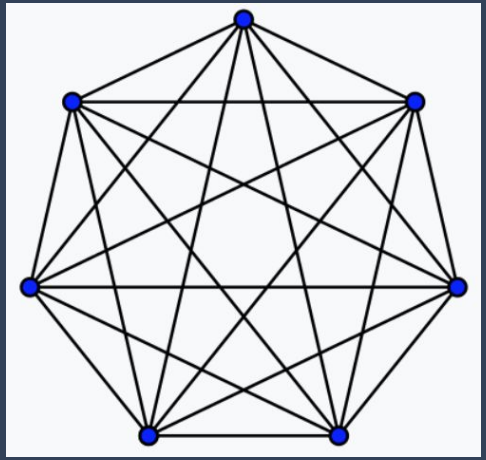
Adjacency List (Directed Graph)



Introduction to Algorithms, 3rd Edition by Thomas H. Cormen, Charles E. Leiserso, Ronald L. Rivest, Clifford Stein

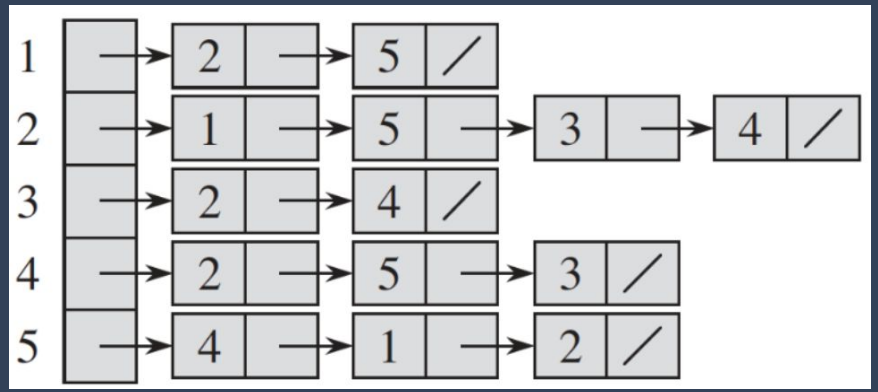
Adjacency List

- Time Complexity
  - O(|V|) time to check whether 2 vertices are directly connected or not
  - O(1) time to get all neighbours of a vertex
- Space Complexity
  - O($|V|^2$) space (worst case)
  - O(|V| + |E|) ~ O(K*|V|) space if each vertex connects to k vertices on average.



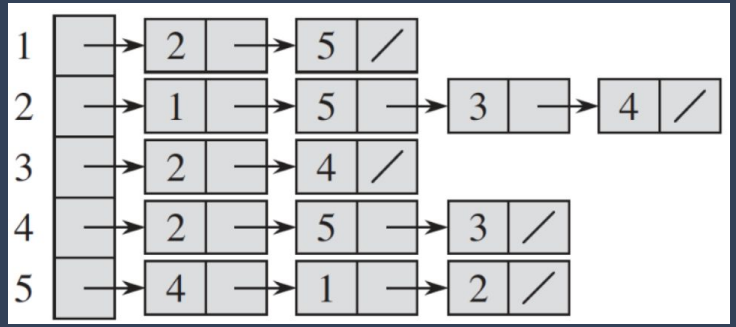
Adjacency List Implementation

```
class Graph {
        // array of list. We can also use list of list here.
        // adjList[i] contains neighbors of vertex i
        List[] adjList;
        int n;

        public Graph(int n) {
                this.n = n;
                this.adjList = new List[n];
                for (int i = 0; i < n; i++) {
                        adjList[i] = new LinkedList<>();
                }
        }
}
```



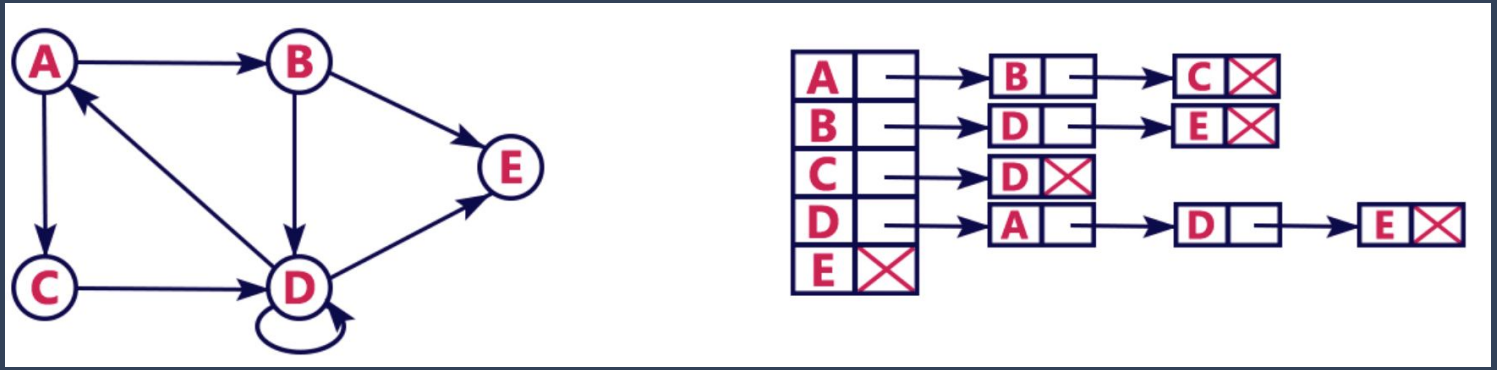
Adjacency List Implementation

```
class Graph {
        ...

        void addEdge(int start, int end) {
                adjList[start].add(end);
        }
        ...
}
```



Adjacency List

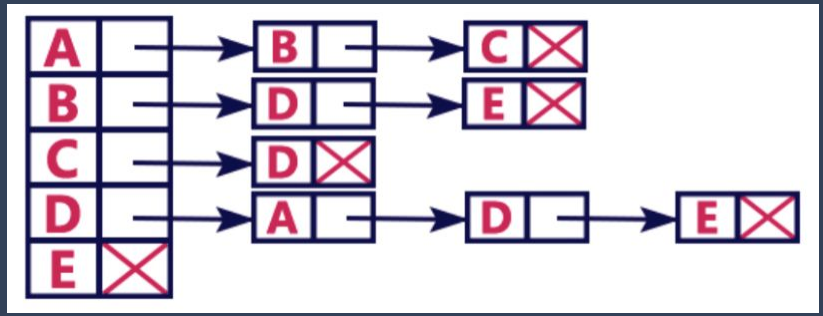
What if the vertices are not integers?



http://btechsmartclass.com

Adjacency List

```
class Graph {
        //      vertex,  neighbors
        Map<String, List<String>> adjMapList;
        String[] vertices;
        public Graph(String[] vertices) {
                this.vertices = vertices;
                this.adjMapList = new HashMap<>();
                for (String vertex : vertices) {
                        adjMapList.put(vertex, new LinkedList<>());
                }
        }
        void addEdge(String start, String end) {
                adjMapList.get(start).add(end);
        }
}
```

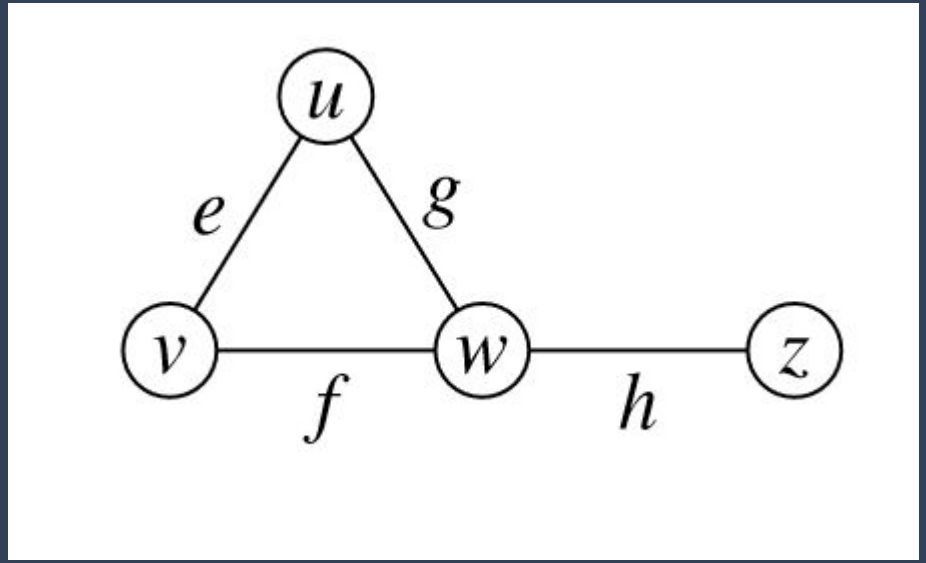


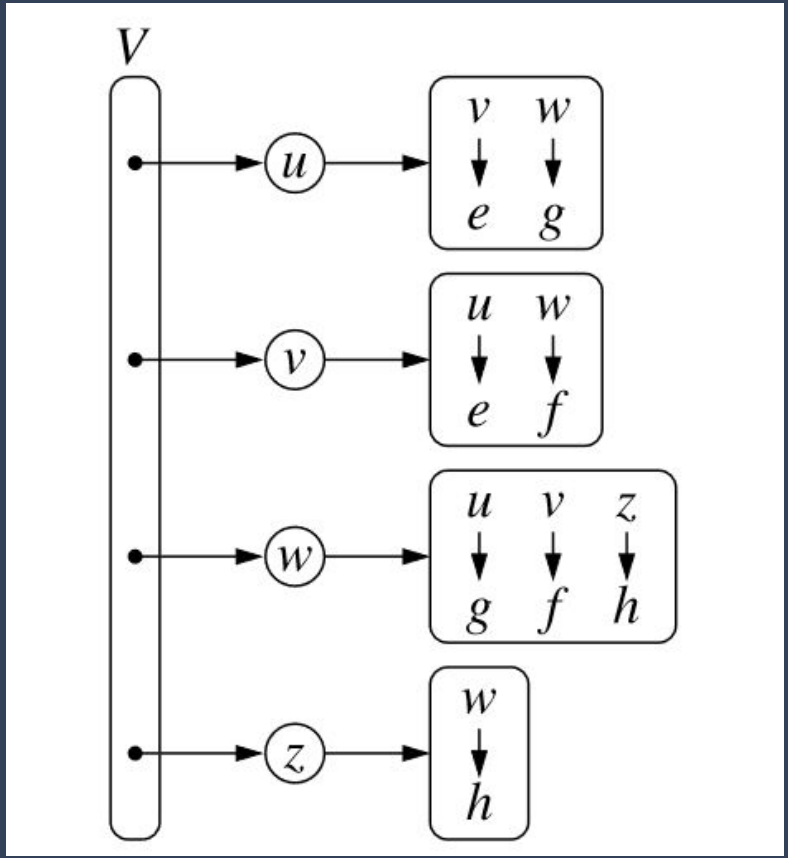
Drawbacks of Adjacency List

- What if we need to **quickly** tell whether two vertices are **directly** connected or not and at the same time, the vertices are not integers?
  - Adjacency matrix cannot be used for non-integer vertices

- What if each edge has a weight? How would we incorporate that?

Adjacency Map





Adjacency Map

- Adjacency map uses a **hashmap** to store the neighbors of vertex *v*, instead of using a list.

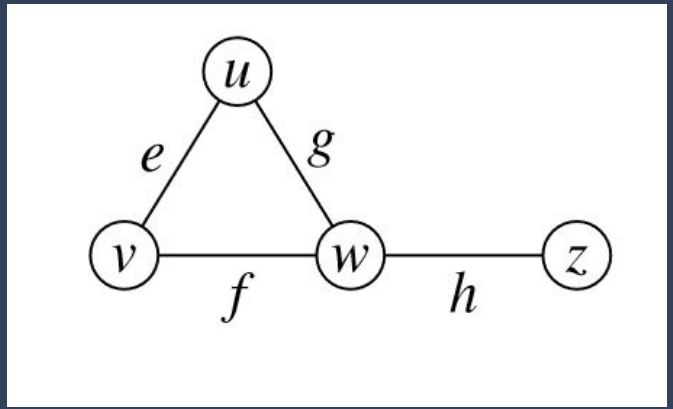
- This allows us to not only iterate over the neighbors if we wish (like in adjacency list), but also find out in ***O*(1)** time if *u* is a neighbor of *v* and to get the edge weight of (*v*,*u*).

- Adjacency map combines the advantages of adjacency list (in space) and adjacency matrix (in time).

- Note: If edges don't have weight, we can use **adjacency set** instead.

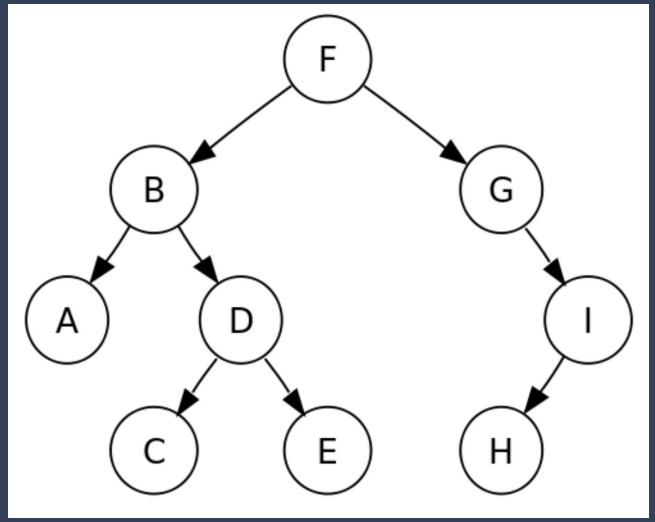
Adjacency Map

```
class Graph {
        //      vertex,         vertex, weight
        Map<String, Map<String, Double>> adjMap;
        String[] vertices;

        public Graph(String[] vertices) {
                this.vertices = vertices;
                this.adjMap = new HashMap<>();
                for (String vertex : vertices) {
                        adjMap.put(vertex, new HashMap<>());
                }
        }

        void addEdge(String start, String end, double weight) {
                adjMap.get(start).put(end, weight);
        }
}
```



Exercise

Build an **undirected** graph based on binary tree




```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

    public TreeNode(int val) {
        this.val = val;
    }
}
```

Answer

```
private void buildGraphFromTree(Map<Integer, List<TreeNode>> map, TreeNode node, TreeNode parent) {
    if (node != null) {
        map.put(node.val, new ArrayList<TreeNode>());
        if (parent != null) {
            map.get(node.val).add(parent);
            map.get(parent.val).add(node);
        }
        buildGraphFromTree(map, node.left, node);
        buildGraphFromTree(map, node.right, node);
    }
}
```

863. All Nodes Distance K in Binary Tree

How to Solve Graph Interview Questions

- First determine whether the problem can be modeled as a graph problem.
- Graph and its representation
- What are the most common graph problems
- How to solve graph problems
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)
  - Topological Sort
  - Dijkstra’s algorithm (optional)
  - Prim's algorithm (optional)
  - Kruskal's algorithm (optional)
  - ……

Are These Problems Graph Problems?

- Evaluate Division
- Course Schedule
- Keys and Rooms
- Word Ladder
- Alien Dictionary
- Shortest Path in Binary Matrix
- Cheapest Flights Within K Stops

399. Evaluate Division

Medium 1517 119 Favorite Share

Equations are given in the format `A / B = k`, where `A` and `B` are variables represented as strings, and `k` is a real number (floating point number). Given some queries, return the answers. If the answer does not exist, return `-1.0`.

Example:
Given `a / b = 2.0, b / c = 3.0.`
queries are: `a / c = ?, b / a = ?, a / e = ?, a / a = ?, x / x = ? .`
return `[6.0, 0.5, -1.0, 1.0, -1.0 ].`

The input is: `vector<pair<string, string>> equations, vector<double>& values, vector<pair<string, string>> queries`, where `equations.size() == values.size()`, and the values are positive. This represents the equations. Return `vector<double>`.

According to the example above:

```
equations = [ ["a", "b"], ["b", "c"] ],
values = [2.0, 3.0],
queries = [ ["a", "c"], ["b", "a"], ["a", "e"], ["a", "a"], ["x", "x"] ].
```

The input is always valid. You may assume that evaluating the queries will result in no division by zero and there is no contradiction.

207. Course Schedule

Medium 2147 101 Favorite Share

There are a total of *n* courses you have to take, labeled from `0` to `n-1`.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: `[0,1]`

Given the total number of courses and a list of prerequisite **pairs**, is it possible for you to finish all courses?

Example 1:

```
Input: 2, [[1,0]]
Output: true
Explanation: There are a total of 2 courses to take.
             To take course 1 you should have finished course 0. So it is possible.
```

Example 2:

```
Input: 2, [[1,0],[0,1]]
Output: false
Explanation: There are a total of 2 courses to take.
             To take course 1 you should have finished course 0, and to take course 0 you should
             also have finished course 1. So it is impossible.
```

841. Keys and Rooms

Medium 550 44 Favorite Share

There are `N` rooms and you start in room `0`. Each room has a distinct number in `0, 1, 2, ..., N-1`, and each room may have some keys to access the next room.

Formally, each room `i` has a list of keys `rooms[i]`, and each key `rooms[i][j]` is an integer in `[0, 1, ..., N-1]` where `N = rooms.length`. A key `rooms[i][j] = v` opens the room with number `v`.

Initially, all the rooms start locked (except for room `0`).

You can walk back and forth between rooms freely.

Return `true` if and only if you can enter every room.

Example 1:

```
Input: [[1],[2],[3],[]]
Output: true
Explanation:
We start in room 0, and pick up key 1.
We then go to room 1, and pick up key 2.
We then go to room 2, and pick up key 3.
We then go to room 3.  Since we were able to go to every room, we return true.
```

Example 2:

```
Input: [[1,3],[3,0,1],[2],[0]]
Output: false
Explanation: We can't enter the room with number 2.
```

127. Word Ladder

Medium 1749 882 Favorite Share

Given two words (*beginWord* and *endWord*), and a dictionary's word list, find the length of shortest transformation sequence from *beginWord* to *endWord*, such that:

1. Only one letter can be changed at a time.
2. Each transformed word must exist in the word list. Note that *beginWord* is *not* a transformed word.

Note:

- Return 0 if there is no such transformation sequence.
- All words have the same length.
- All words contain only lowercase alphabetic characters.
- You may assume no duplicates in the word list.
- You may assume *beginWord* and *endWord* are non-empty and are not the same.

Example 1:

```
Input:
beginWord = "hit",
endWord = "cog",
wordList = ["hot","dot","dog","lot","log","cog"]

Output: 5

Explanation: As one shortest transformation is "hit" -> "hot" -> "dot" -> "dog" -> "cog",
return its length 5.
```

269. Alien Dictionary

There is a new alien language which uses the latin alphabet. However, the order among letters are unknown to you. You receive a list of words from the dictionary, where words are sorted lexicographically by the rules of this new language. Derive the order of letters in this language.

For example, Given the following words in dictionary,

```
[
  "wrt",
  "wrf",
  "er",
  "ett",
  "rftt"
]
```

The correct order is: `"wertf"`.

Note: You may assume all letters are in lowercase. If the order is invalid, return an empty string. There may be multiple valid order of letters, return any one of them is fine.

1091. Shortest Path in Binary Matrix

Medium 116 20 Favorite Share

In an N by N square grid, each cell is either empty (0) or blocked (1).

A *clear path from top-left to bottom-right* has length `k` if and only if it is composed of cells `C_1, C_2, ..., C_k` such that:

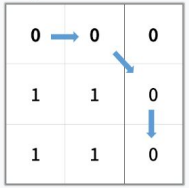
- Adjacent cells `C_i` and `C_{i+1}` are connected 8-directionally (ie., they are different and share an edge or corner)
- `C_1` is at location `(0, 0)` (ie. has value `grid[0][0]`)
- `C_k` is at location `(N-1, N-1)` (ie. has value `grid[N-1][N-1]`)
- If `C_i` is located at `(r, c)`, then `grid[r][c]` is empty (ie. `grid[r][c] == 0`).

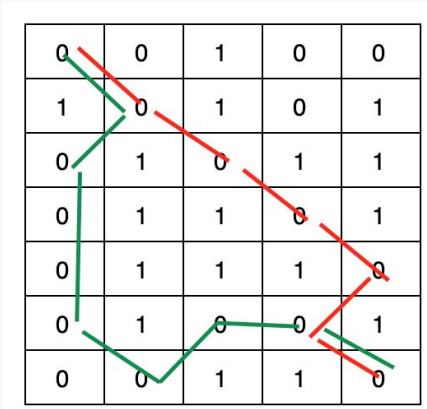
Return the length of the shortest such clear path from top-left to bottom-right. If such a path does not exist, return -1.

Input: `[[0,0,0],[1,1,0],[1,1,0]]`

0	0	0
1	1	0
1	1	0

Output: 4





787. Cheapest Flights Within K Stops

Medium 1277 46 Add to List Share

There are `n` cities connected by `m` flights. Each flight starts from city `u` and arrives at `v` with a price `w`.

Now given all the cities and flights, together with starting city `src` and the destination `dst`, your task is to find the cheapest price from `src` to `dst` with up to `k` stops. If there is no such route, output `-1`.

Is It a Graph Problem?

- Determine whether the data in the problem can be modeled as vertices or not?
- Determine whether the vertices are connected or not?
- If **both** answers are **YES**, then we can model that problem as graph problem

Identifying the category of the question is the FIRST thing you should do after you FULLY understand the question in an interview!!!

How to Solve Graph Interview Questions

- First determine whether the problem can be modeled as a graph problem.
- Graph and its representation
- What are the most common graph problems
- How to solve graph problems
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)
  - Topological sort
  - Dijkstra’s algorithm (optional)
  - Prim's algorithm (optional)
  - Kruskal's algorithm (optional)
  - ……

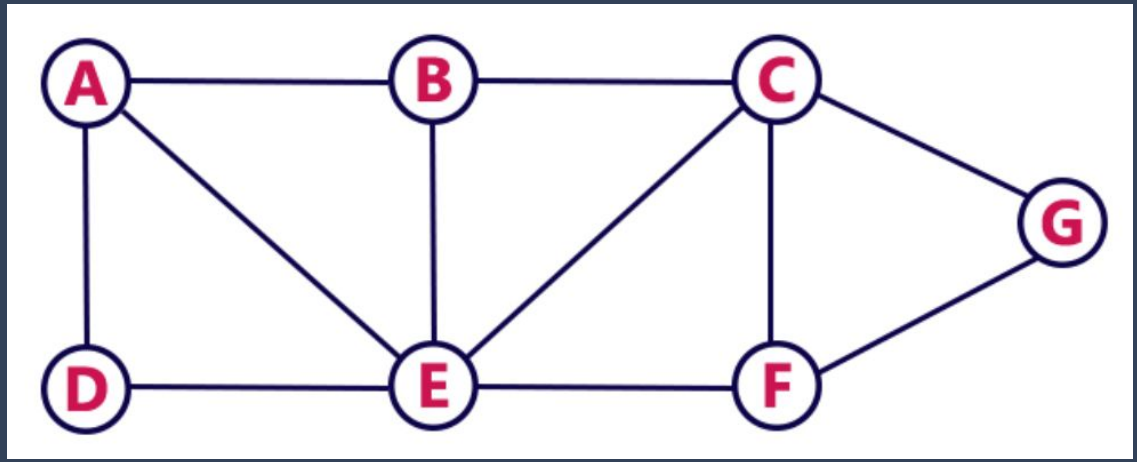
Most Common Graph Problems

1. Traverse all vertices in a graph (Keys and Rooms)
2. Whether two vertices are connected or not (Division Evaluation)
3. Shortest path between two vertices (Shortest Path in Binary Matrix, Word Ladder I & II, Walls and Gates)
4. Number of connected components (Number of Islands)
5. Dependency path (Course Schedule II, Alien Dictionary)
6. Vertices grouping/coloring (Is Graph Bipartite?)
7. Detect cycle in a graph (Course Schedule I, Valid Tree Graph)
8. All possible paths (Word Search, Binary Tree Paths)
9. Shortest path (with cost) between two vertices (Cheapest Flights Within K Stops)
10. Minimum spanning tree (Cable/ Water Pipeline Problem)

How to Solve Graph Interview Questions?

- First determine whether the problem can be modeled as a graph problem.
- Graph and its representation
- What are the most common graph problems
- How to solve graph problems
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)
  - Topological sort
  - Dijkstra’s algorithm (optional)
  - Prim's algorithm (optional)
  - Kruskal's algorithm (optional)
  - ……

BFS

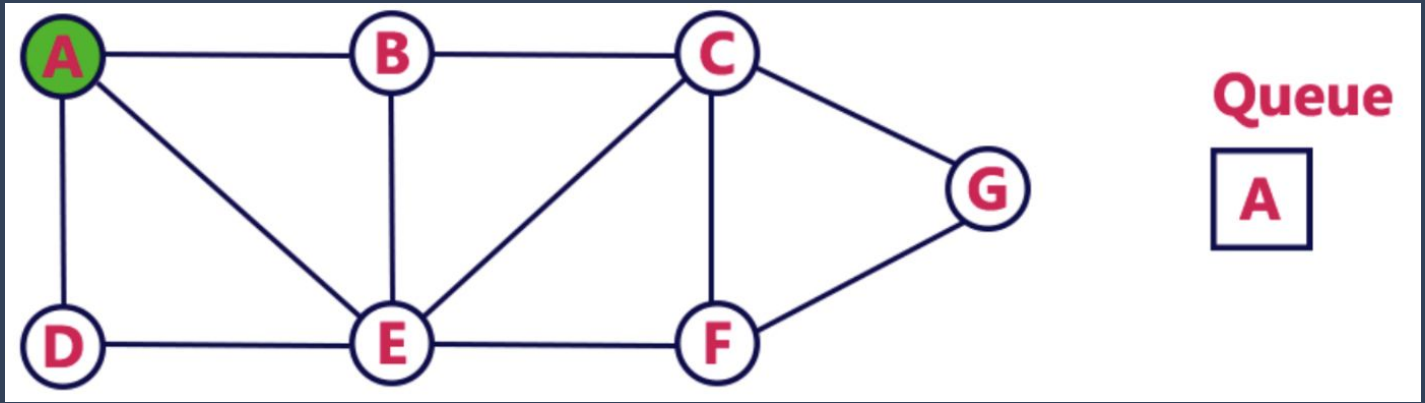



```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2       u.color = WHITE
 3  s.color = BLACK
 4  Q = ∅
 5  ENQUEUE(Q, s)
 6  while Q ≠ ∅
 7       u = DEQUEUE(Q)
 8       for each v ∈ G.Adj[u]
 9            if v.color == WHITE
10                 v.color = BLACK
11                 ENQUEUE(Q, v)
```

- Time Complexity: $O(|V| + |E|)$
- Space Complexity: $O(|V|)$

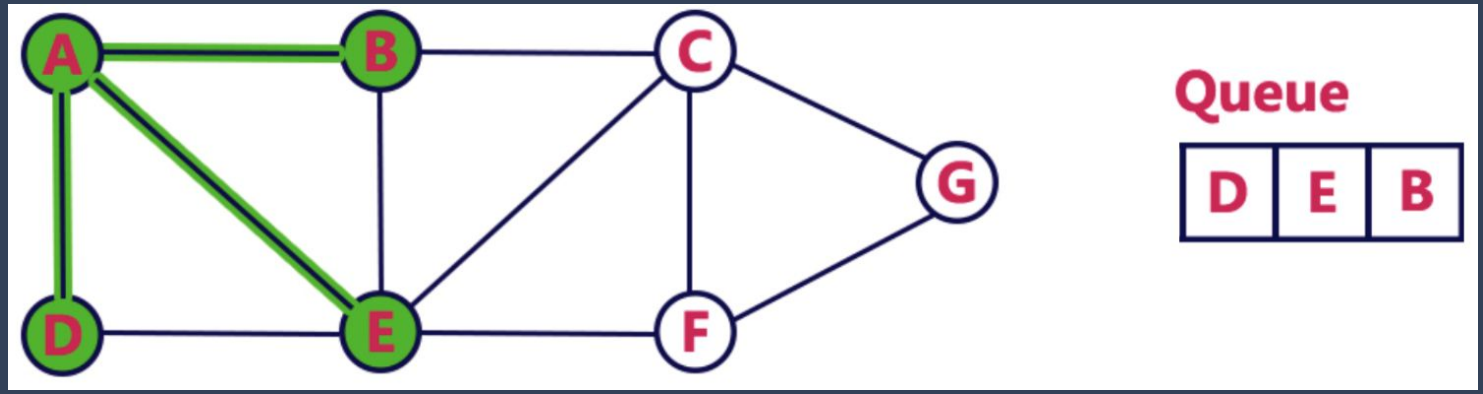
BFS

Step 1: Enqueue A, and mark it as VISITED



BFS

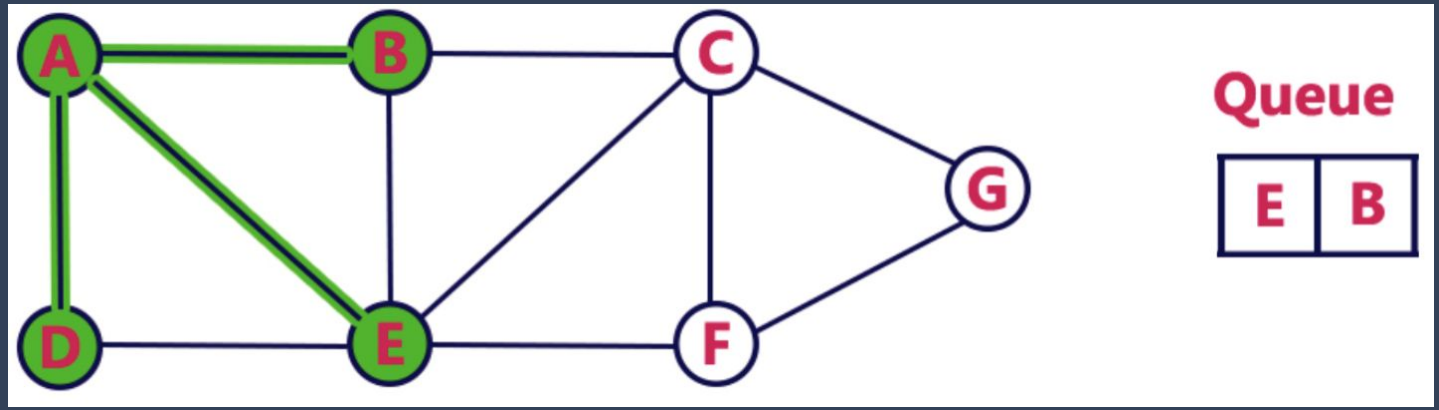
Step 2: Dequeue A, and enqueue its unvisited neighbors (B, D, E) and mark them as VISITED.



http://btechsmartclass.com

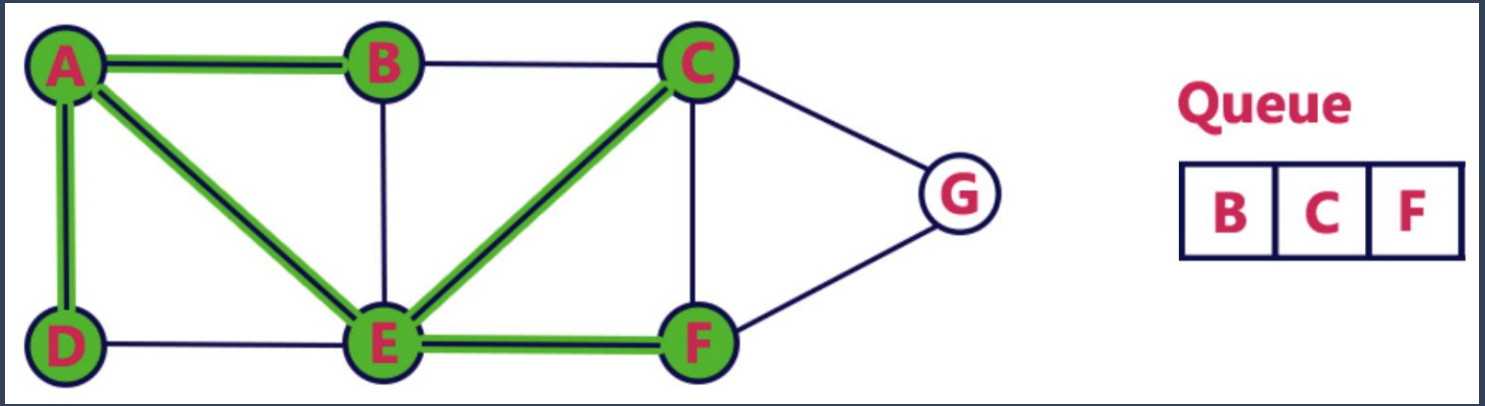
BFS

Step 3: Dequeue D



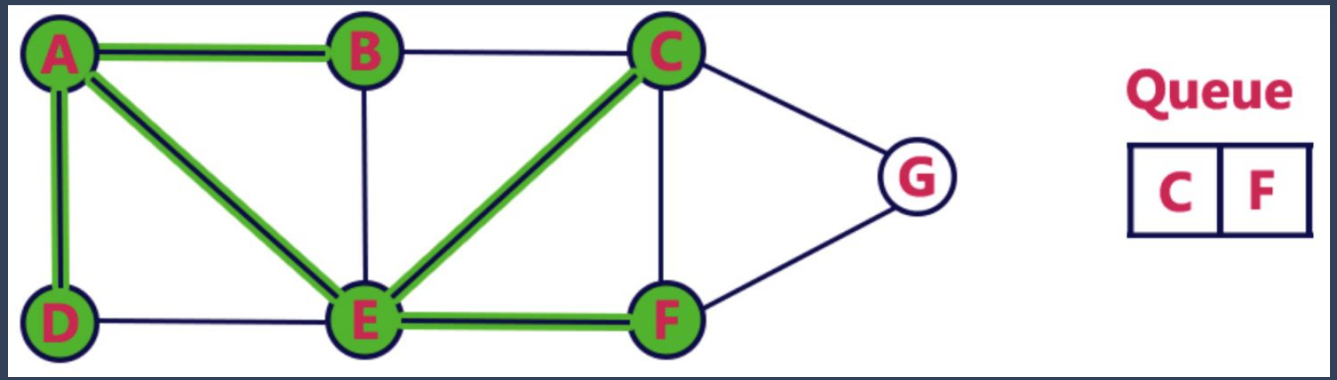
BFS

Step 4: Dequeue E, and enqueue its unvisited neighbors (C, F) and mark them as VISITED



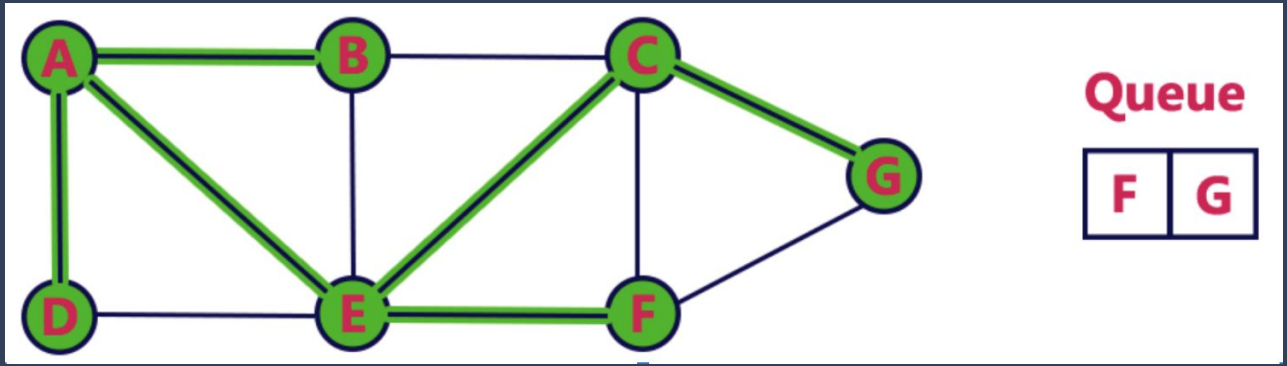
BFS

Step 5: Dequeue B



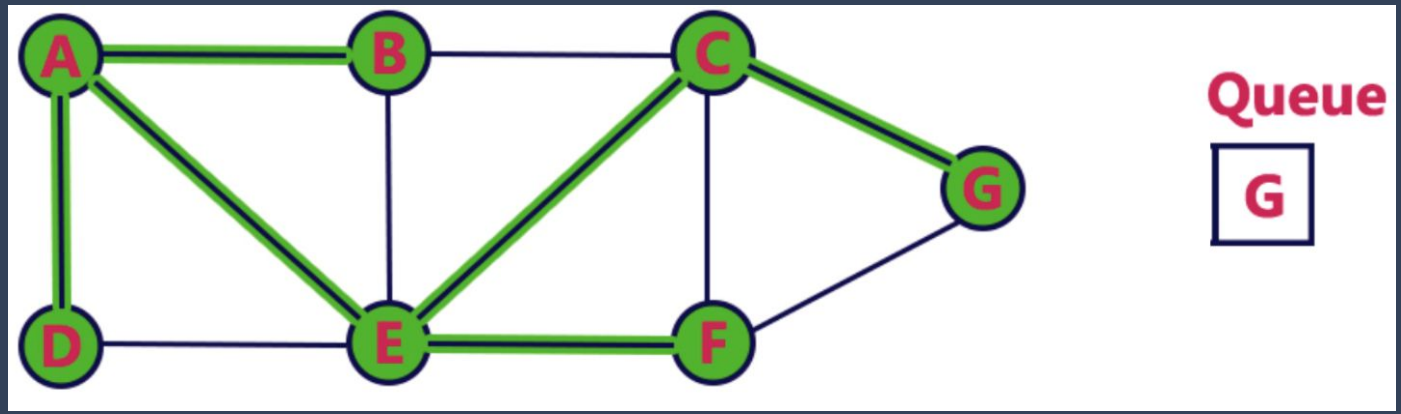
BFS

Step 6: Dequeue C, and enqueue its unvisited neighbors (G) and mark it as VISITED



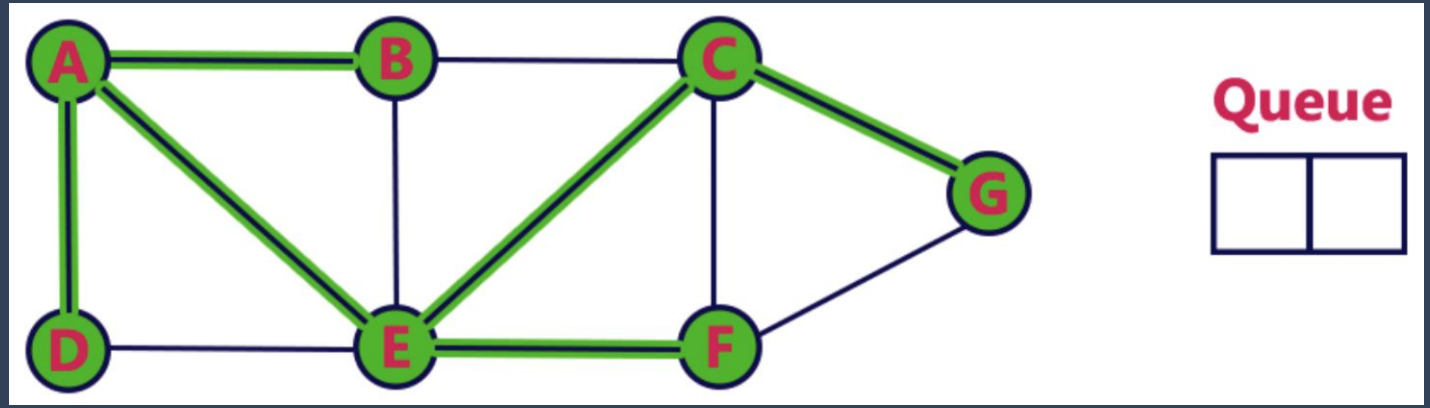
BFS

Step 7: Deque F



BFS

Step 8: Queue becomes empty, exit

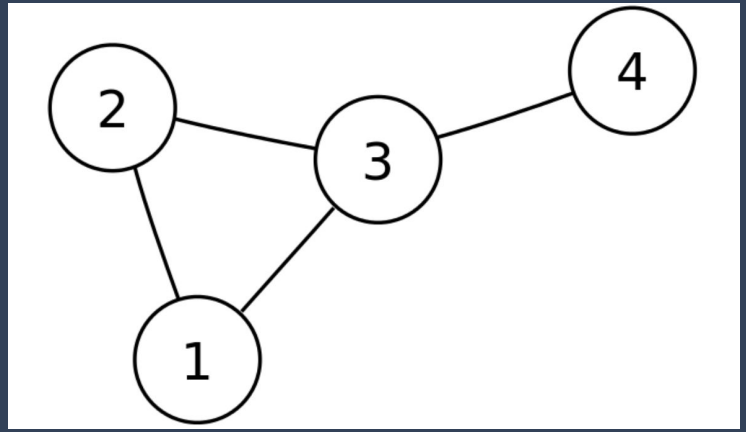


Do You Know How Gossip Works? :-)

https://confessionsofanadoptiveparent.com

{ik} INTERVIEW KICKSTART

BFS Implementation (Java)



Input: N and a list of edges

Example:

- N= 4
- Edges = {(1, 2), (2, 1), (1, 3), (3, 1), (3, 2), (2, 3), (3, 4), (4, 3)}

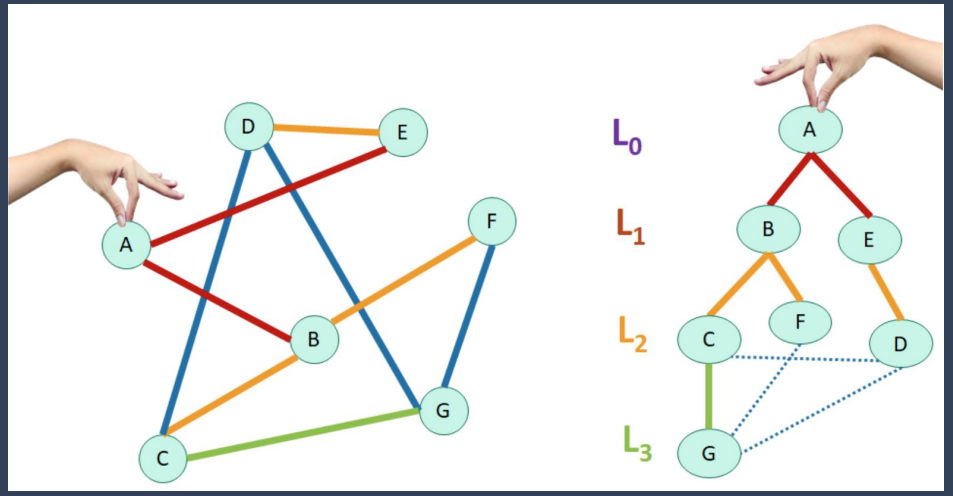
BFS Implementation

```java
class Graph {
    int n;
    Map<Integer, List<Integer>> map;

    // initialization
    public Graph(int n, int[][] edges) {
        this.n = n;
        this.map = new HashMap<>();

        for (int i = 1; i <= n; i++) {
            map.put(i, new LinkedList<>());
        }

        for (int[] edge : edges) {
            map.get(edge[0]).add(edge[1]);
        }
    }
}
```

```java
/*
 * Traverse the graph by using BFS
 */
public void bfs(int s) {
    boolean[] visited = new boolean[n];
    Queue<Integer> queue = new LinkedList<>();
    queue.offer(s);
    visited[s - 1] = true;
    while (!queue.isEmpty()) {
        int top = queue.poll();
        for (int i : map.get(top)) {
            if (!visited[i - 1]) {
                visited[i - 1] = true;
                queue.offer(i);
            }
        }
    }
}
```

BFS

- BFS traverses by level
- **Number of levels** is also the **shortest distance** between two vertices
- Question: what is the shortest distance from vertex A to vertex G?



Original Graph

Visualized BFS Traversal

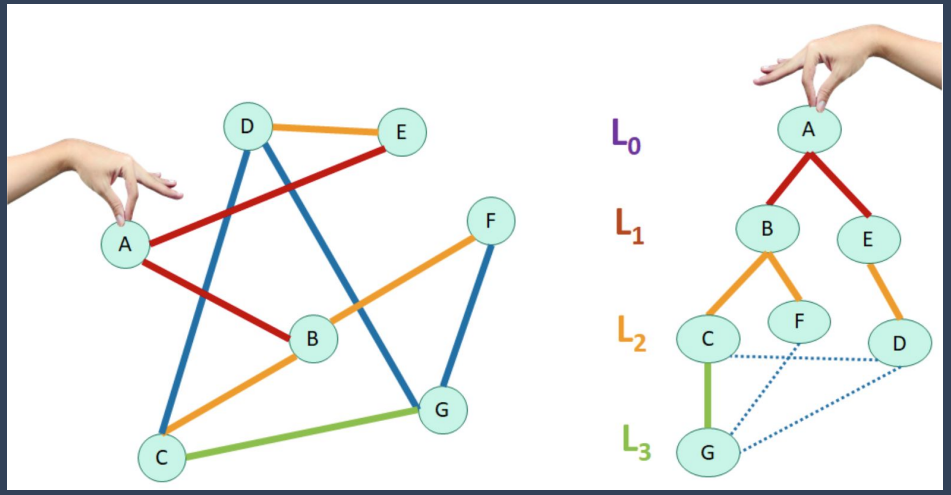
How to Get the Number of Levels in BFS?

- There is only **one queue** to store neighbors in BFS approach, how do we know which nodes vertices are in the same level?

- **Important Fact:** Queue only has **vertices in a single level** when the vertices in the previous level are popped out.

- We could use **count of vertices in each level** as “virtual” divider to separate vertices.

Levels in BFS



queue	A	E,B	C, F, D	G
count	1	2	3	1
level	1	2	3	4

BFS by Level

```java
class Graph {
    int n;
    Map<Integer, List<Integer>> map;

    // initialization
    public Graph(int n, int[][] edges) {
        this.n = n;
        this.map = new HashMap<>();

        for (int i = 1; i <= n; i++) {
            map.put(i, new LinkedList<>());
        }

        for (int[] edge : edges) {
            map.get(edge[0]).add(edge[1]);
        }
    }
}
```

```java
/*
 * Traverse the graph by using BFS
 */
public void bfs(int s) {
    boolean[] visited = new boolean[n];
    Queue<Integer> queue = new LinkedList<>();
    queue.offer(s);
    visited[s - 1] = true;
    // the number of levels
    int level = 1;
    while (!queue.isEmpty()) {
        // the number of vertices in current level
        int size = queue.size();
        for (int j = 1; j <= size; j++) {
            int top = queue.poll();
            for (int i : map.get(top)) {
                if (!visited[i - 1]) {
                    visited[i - 1] = true;
                    queue.offer(i);
                }
            }
        }
        // increase level after finishing processing
        // all the vertices in the current level
        level++;
    }
}
```

What Problems Can be Solved by BFS?

1. Traverse all vertices in a graph
2. Whether two vertices are connected or not
3. Shortest path between two vertices (preferred approach)
4. Number of connected components
5. Dependency path
6. Vertices grouping/coloring (preferred approach)
7. Detect cycle in a graph
8. All possible paths
9. Shortest path (with cost) between two vertices
10. Minimum Spanning Tree

Question

1091. Shortest Path in Binary Matrix

Medium 116 20 Favorite Share

In an N by N square grid, each cell is either empty (0) or blocked (1).

A *clear path from top-left to bottom-right* has length `k` if and only if it is composed of cells `C_1, C_2, ..., C_k` such that:

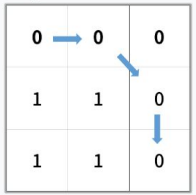
- Adjacent cells `C_i` and `C_{i+1}` are connected 8-directionally (ie., they are different and share an edge or corner)
- `C_1` is at location `(0, 0)` (ie. has value `grid[0][0]`)
- `C_k` is at location `(N-1, N-1)` (ie. has value `grid[N-1][N-1]`)
- If `C_i` is located at `(r, c)`, then `grid[r][c]` is empty (ie. `grid[r][c] == 0`).

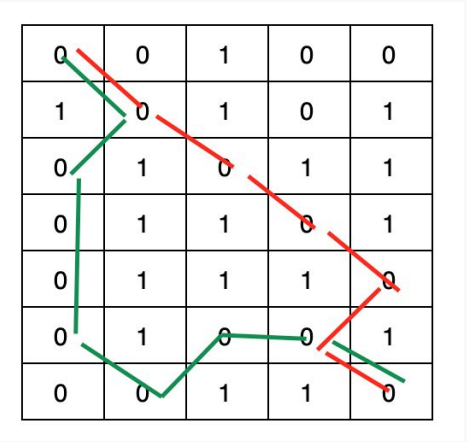
Return the length of the shortest such clear path from top-left to bottom-right. If such a path does not exist, return -1.

Input: `[[0,0,0],[1,1,0],[1,1,0]]`

0	0	0
1	1	0
1	1	0

Output: 4





{ik} INTERVIEW KICKSTART

Shortest Path in Binary Matrix

Idea:

- This is essentially a graph problem (tell them why, and how the graph is represented)
- I can use BFS approach to solving this problem.
- When traversing, we use two variables: one keeps track of the number of levels, and the other one tracks the number of vertices in each level.
- Once BFS traversal is completed, return the number of levels.

Directions

Right Left Down Up B-L T-R T-L B-R

```
int dir[][] = { { 0, 1 }, { 0, -1 }, { 1, 0 }, { -1, 0 }, { 1, -1 }, { -1, 1 }, { -1, -1 }, { 1, 1 } };
```

column

row

(x-1,y-1)	(x-1,y)	(x-1,y+1)
(x,y-1)	(x,y)	(x,y+1)
(x+1,y-1)	(x+1,y)	(x+1,y+1)

Solution

```java
class Solution {
    public int shortestPathBinaryMatrix(int[][] grid) {
        int row = grid.length, col = grid[0].length;
        // corner case
        if (grid[0][0] == 1 || grid[row - 1][col - 1] == 1) return -1;

        boolean[][] visited = new boolean[row][col];
        visited[0][0] = true;
        Queue<int[]> queue = new LinkedList<>();
        queue.add(new int[] { 0, 0 });

        int dir[][] = { { 0, 1 }, { 0, -1 }, { 1, 0 }, { -1, 0 }, { 1, -1 }, { -1, 1 }, { -1, -1 }, { 1, 1 } };
        int level = 1;
        while (!queue.isEmpty()) {
            int size = queue.size();  // the number of vertices in current level
            for (int i = 0; i < size; i++) {
                int[] node = queue.remove();
                // if true, we reach the bottom right node
                if (node[0] == row - 1 && node[1] == col - 1) return level;

                // enumerate all neighbors
                for (int j = 0; j < dir.length; j++) {
                    int neighbor_x = dir[j][0] + node[0];
                    int neighbor_y = dir[j][1] + node[1];
                    
                }
            }
            level++;
        }
        return -1; // no path found
    }
}
```

Solution

```java
class Solution {
    public int shortestPathBinaryMatrix(int[][] grid) {
        int row = grid.length, col = grid[0].length;
        // corner case
        if (grid[0][0] == 1 || grid[row - 1][col - 1] == 1) return -1;

        boolean[][] visited = new boolean[row][col];
        visited[0][0] = true;
        Queue<int[]> queue = new LinkedList<>();
        queue.add(new int[] { 0, 0 });

        int dir[][] = { { 0, 1 }, { 0, -1 }, { 1, 0 }, { -1, 0 }, { 1, -1 }, { -1, 1 }, { -1, -1 }, { 1, 1 } };
        int level = 1;
        while (!queue.isEmpty()) {
            int size = queue.size();  // the number of vertices in current level
            for (int i = 0; i < size; i++) {
                int[] node = queue.remove();
                // if true, we reach the bottom right node
                if (node[0] == row - 1 && node[1] == col - 1) return level;

                // enumerate all neighbors
                for (int j = 0; j < dir.length; j++) {
                    int neighbor_x = dir[j][0] + node[0];
                    int neighbor_y = dir[j][1] + node[1];
                    // check whether the index is valid or not, and also whether the neighbor is already visited or not
                    if (neighbor_x >= 0 && neighbor_x < row && neighbor_y >= 0 && neighbor_y < col
                            && !visited[neighbor_x][neighbor_y] && grid[neighbor_x][neighbor_y] == 0) {
                        queue.add(new int[] { neighbor_x, neighbor_y });
                        visited[neighbor_x][neighbor_y] = true;
                    }
                }
            }
            level++;
        }
        return -1; // no path found
    }
}
```

Question

785. Is Graph Bipartite?

Medium 769 97 Favorite Share

Given an undirected `graph`, return `true` if and only if it is bipartite.

Recall that a graph is *bipartite* if we can split it's set of nodes into two independent subsets A and B such that every edge in the graph has one node in A and another node in B.

The graph is given in the following form: `graph[i]` is a list of indexes `j` for which the edge between nodes `i` and `j` exists. Each node is an integer between `0` and `graph.length - 1`. There are no self edges or parallel edges: `graph[i]` does not contain `i`, and it doesn't contain any element twice.

Example 1:
Input: [[1,3], [0,2], [1,3], [0,2]]
Output: true
Explanation:
The graph looks like this:
```
0----1
|    |
|    |
3----2
```
We can divide the vertices into two groups: {0, 2} and {1, 3}.

Example 2:
Input: [[1,2,3], [0,2], [0,1,3], [0,2]]
Output: false
Explanation:
The graph looks like this:
```
0----1
| \  |
|  \ |
3----2
```
We cannot find a way to divide the set of nodes into two independent subsets.

Is Graph Bipartite

Idea:

- I can use BFS approach to solve this problem.
- Assign black color to one vertex and white color to the other vertex in one edge, and color represents group.
- **If both vertices on the same edge have the same color, return false.**
- If no vertices on the same edge have the same color, return true.

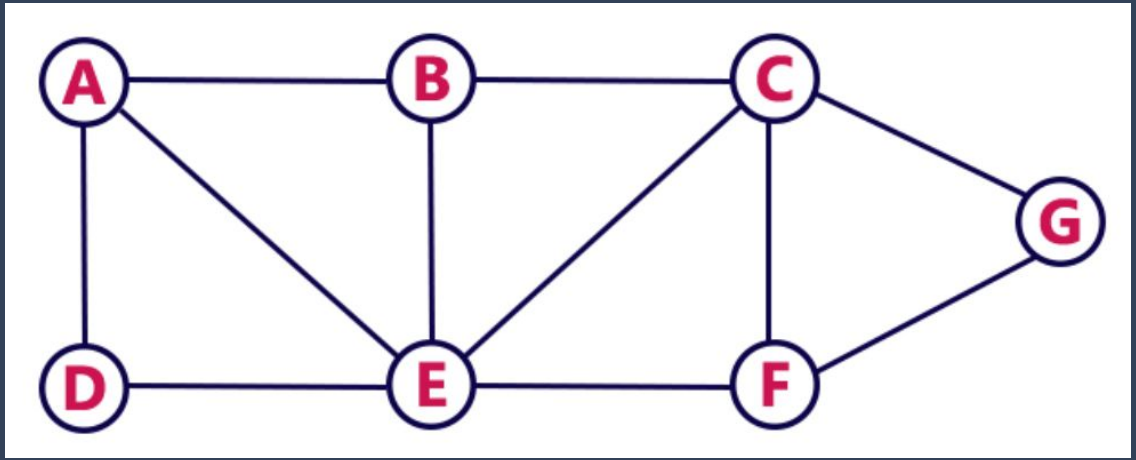
Solution

```
class Solution {
    public boolean isBipartite(int[][] graph) {
        // 0: unvisited; 1: black; 2: white
        int[] status = new int[graph.length];

        for (int i = 0; i < graph.length; i++) {
            if (status[i] == 0) {
                status[i] = 1;
                Queue<Integer> queue = new LinkedList<>();
                queue.offer(i);
                while (!queue.isEmpty()) {
                    int current = queue.poll();
                    for (int neighbor : graph[current]) {
                        
                    }
                }
            }
        }
        return true;
    }
}
```

Solution

```
class Solution {
    public boolean isBipartite(int[][] graph) {
        // 0: unvisited; 1: black; 2: white
        int[] status = new int[graph.length];

        for (int i = 0; i < graph.length; i++) {
            if (status[i] == 0) {
                status[i] = 1;
                Queue<Integer> queue = new LinkedList<>();
                queue.offer(i);
                while (!queue.isEmpty()) {
                    int current = queue.poll();
                    for (int neighbor : graph[current]) {
                        if (status[neighbor] == 0) {
                            status[neighbor] = (status[current] == 1) ? 2 : 1;
                            queue.offer(neighbor);
                        } else {
                            if (status[neighbor] == status[current]) {
                                return false;
                            }
                        }
                    }
                }
            }
        }
        return true;
    }
}
```

DFS



- Time Complexity: $O(|V| + |E|)$
- Space Complexity: $O(|V|)$

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3  for each vertex u ∈ G.V
4      if u.color == WHITE
5          DFS-VISIT(G, u)

DFS-VISIT(G, u)
1  u.color = GRAY
2  for each v ∈ G.Adj[u]
3      if v.color == WHITE
4          DFS-VISIT(G, v)
5  u.color = BLACK
```

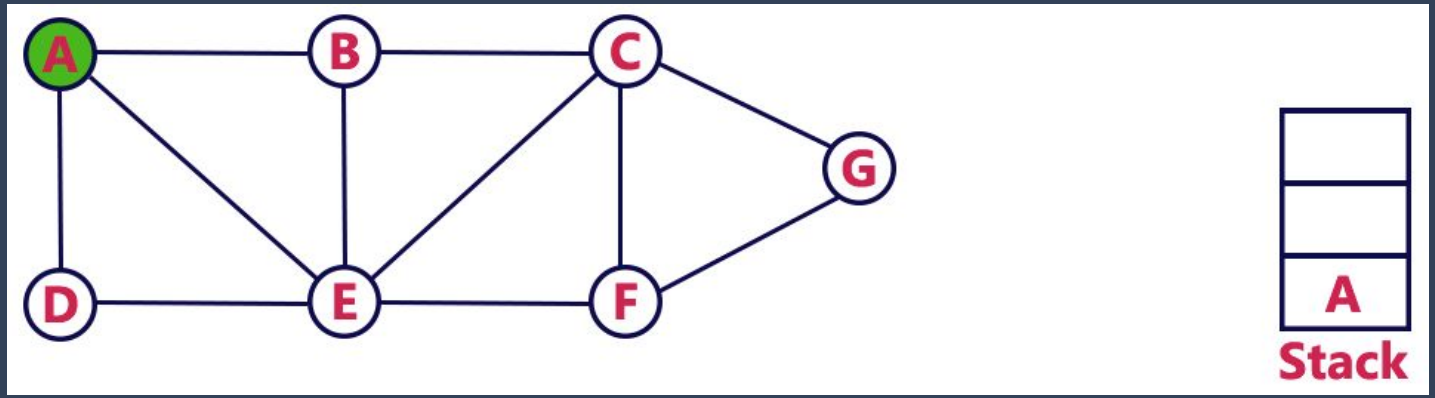
Colors in DFS

- White: the vertex is unvisited
- Gray: the vertex is visited, but we are still visiting all its neighbors. (being visited)
- Black: the vertex is visited, and all of its neighbors are also visited.

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3  for each vertex u ∈ G.V
4      if u.color == WHITE
5          DFS-VISIT(G, u)

DFS-VISIT(G, u)
1  u.color = GRAY
2  for each v ∈ G.Adj[u]
3      if v.color == WHITE
4          DFS-VISIT(G, v)
5  u.color = BLACK
```

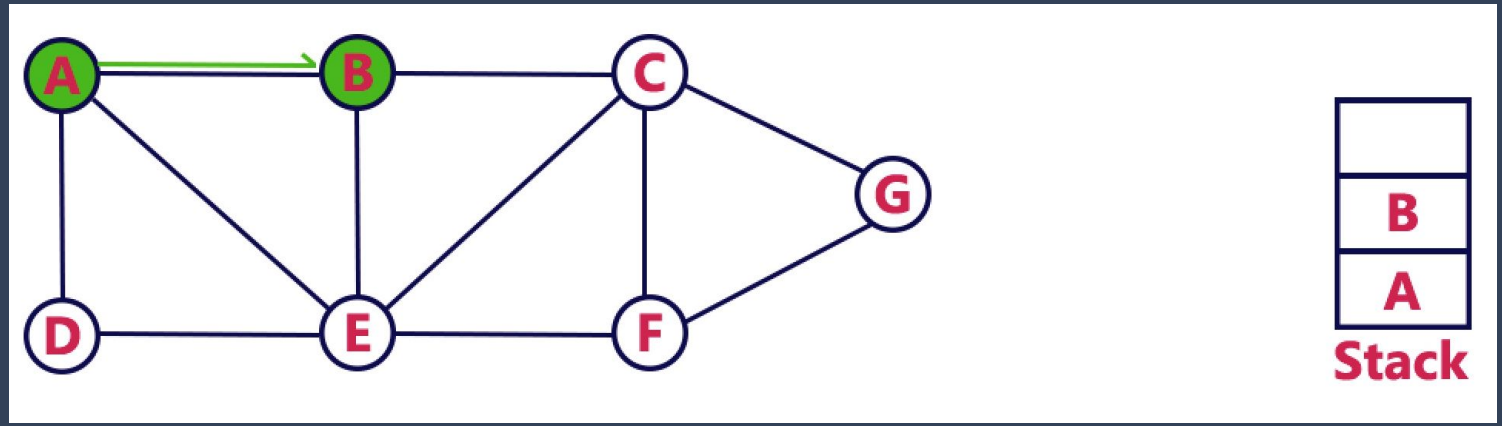
DFS

Step 1: Push A to stack, and mark it as VISITED



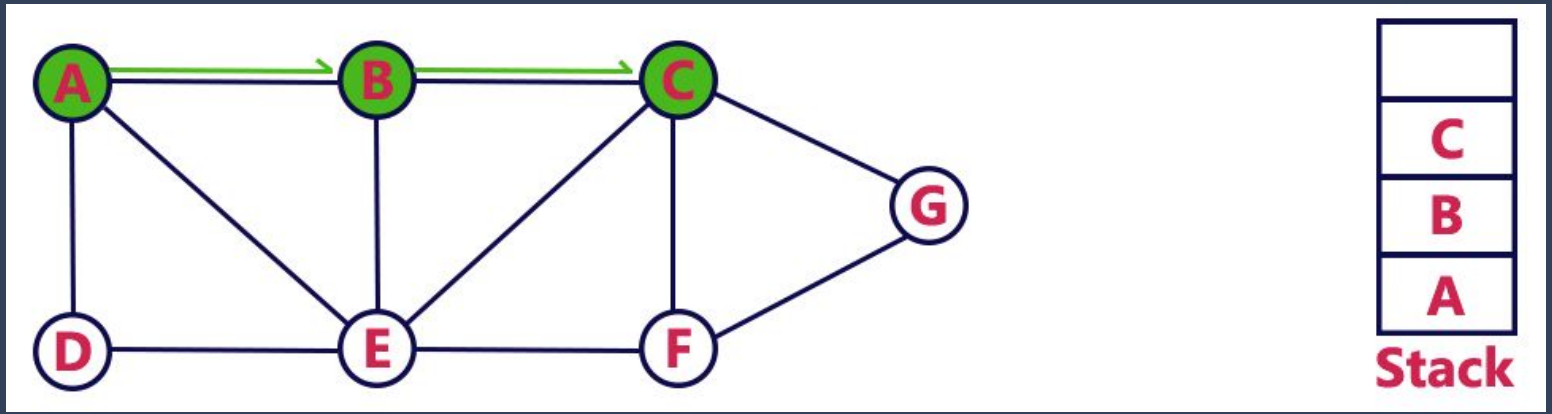
DFS

Step 2: Find A's first neighbour B, and push B to stack, and mark B as VISITED



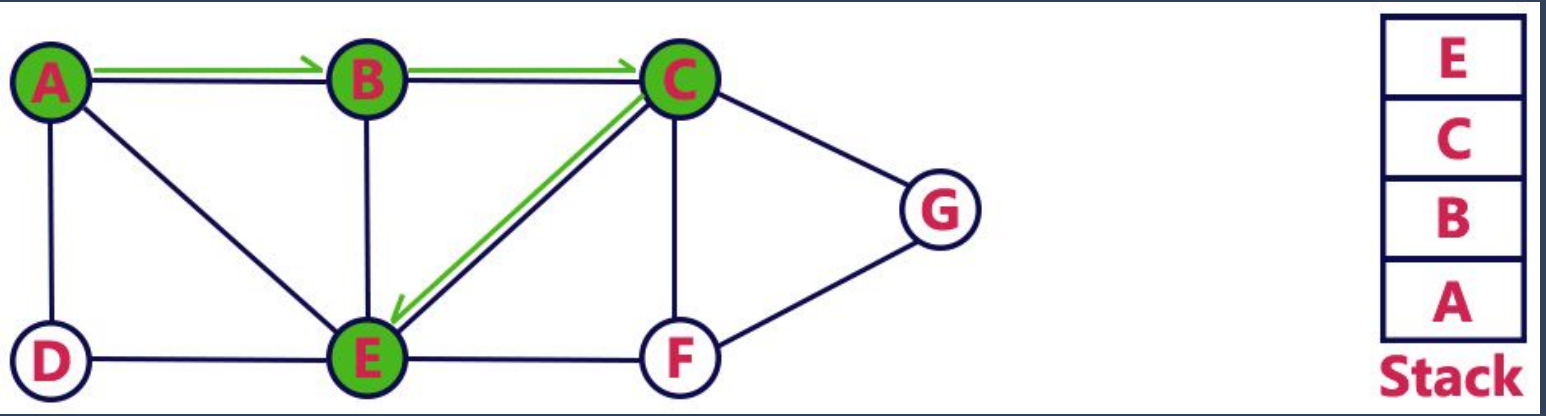
DFS

Step 3: Find B’s first neighbour C, and push C to stack, and mark C as VISITED



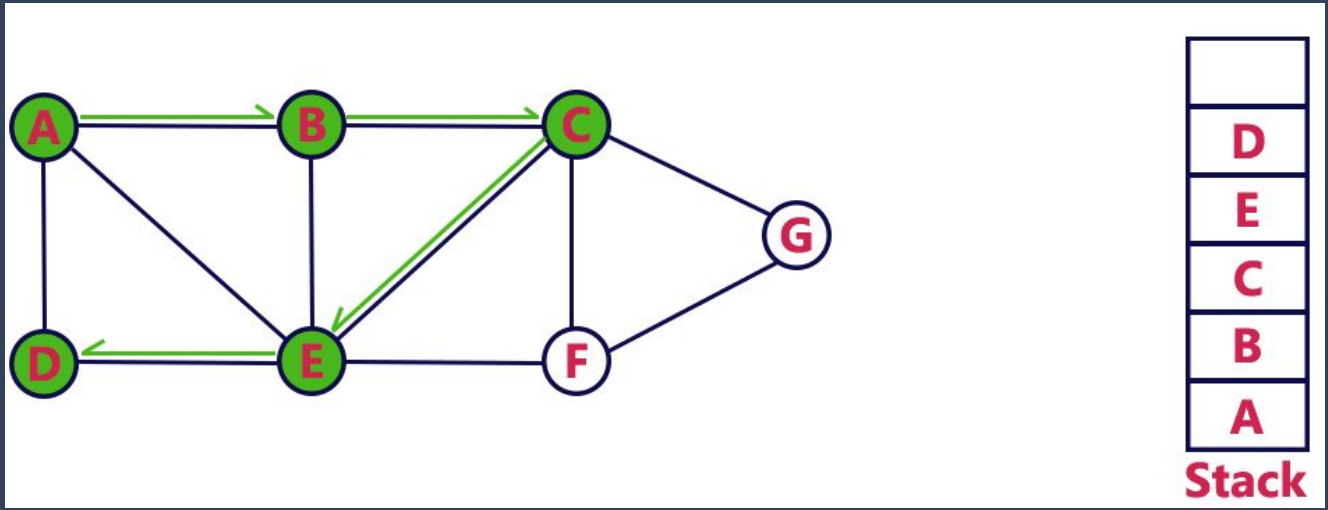
DFS

Step 4: Find C’s first neighbour E, and push E to stack, and mark E as VISITED



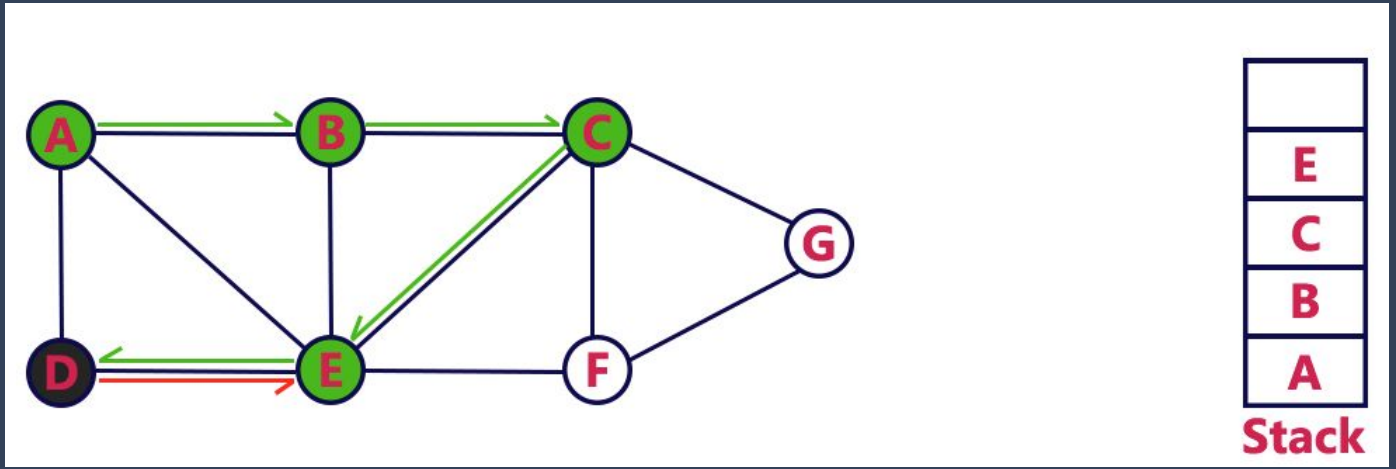
DFS

Step 5: Find E’s first unvisited neighbour D, and push D to stack, and mark D as VISITED



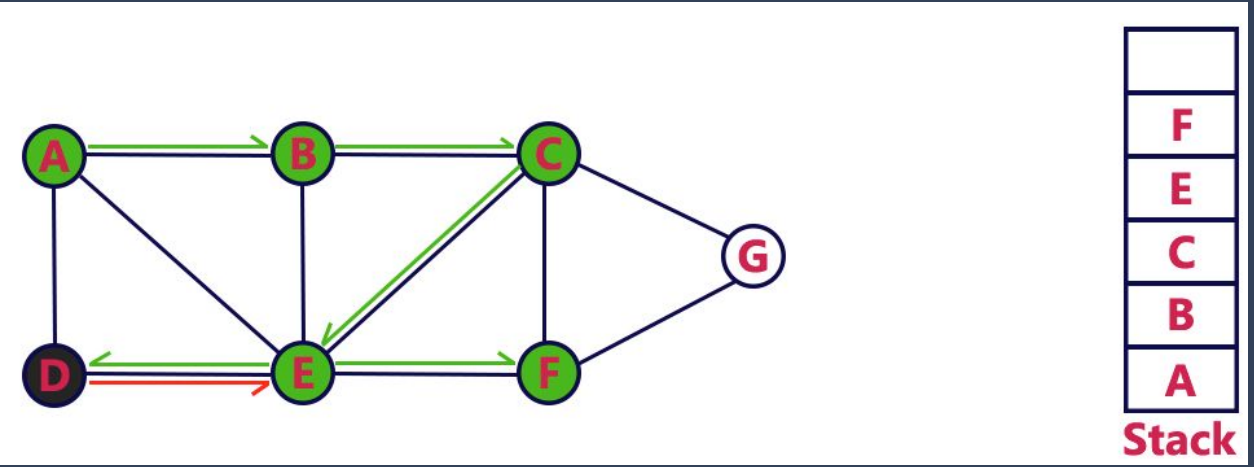
DFS

Step 6: As D doesn’t have unvisited neighbour, return (pop stack)



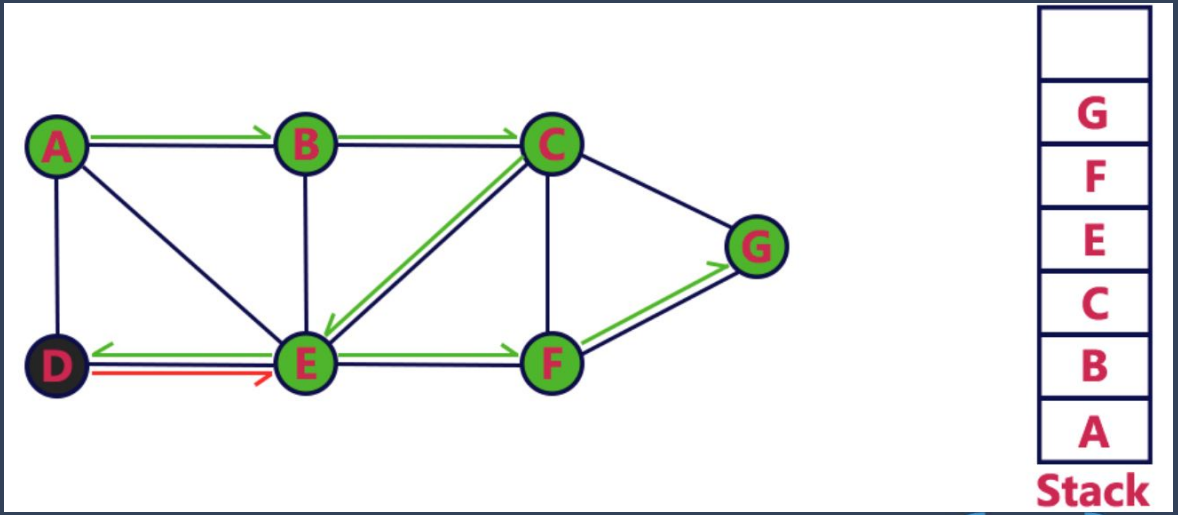
DFS

Step 7: Find E’s second unvisited neighbour (F), push F to stack and mark F as VISITED



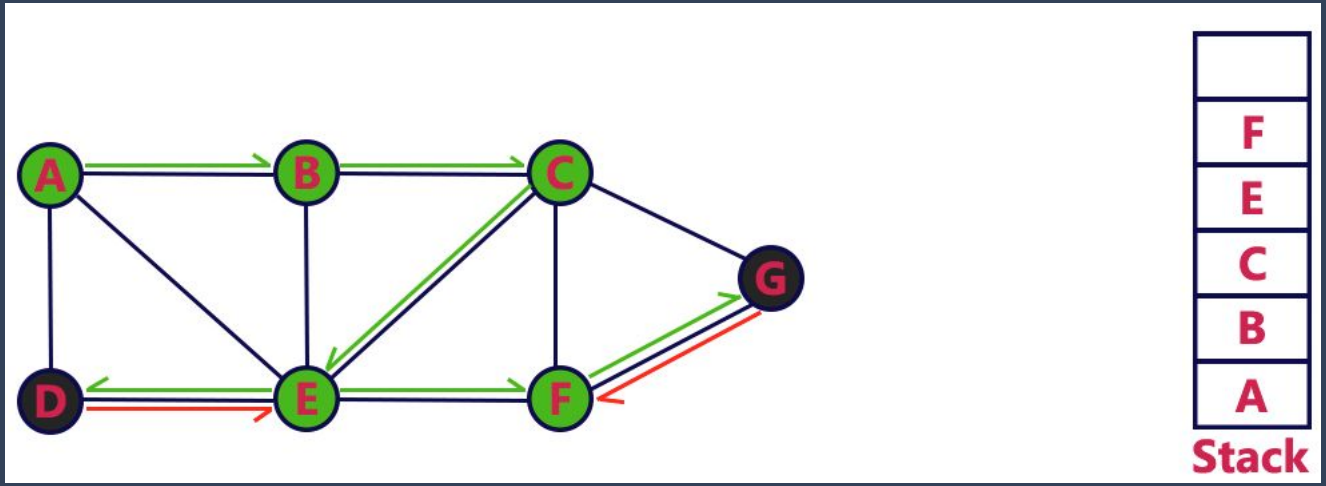
DFS

Step 8: Find F’s first unvisited neighbour (G), push G to stack and mark G as VISITED



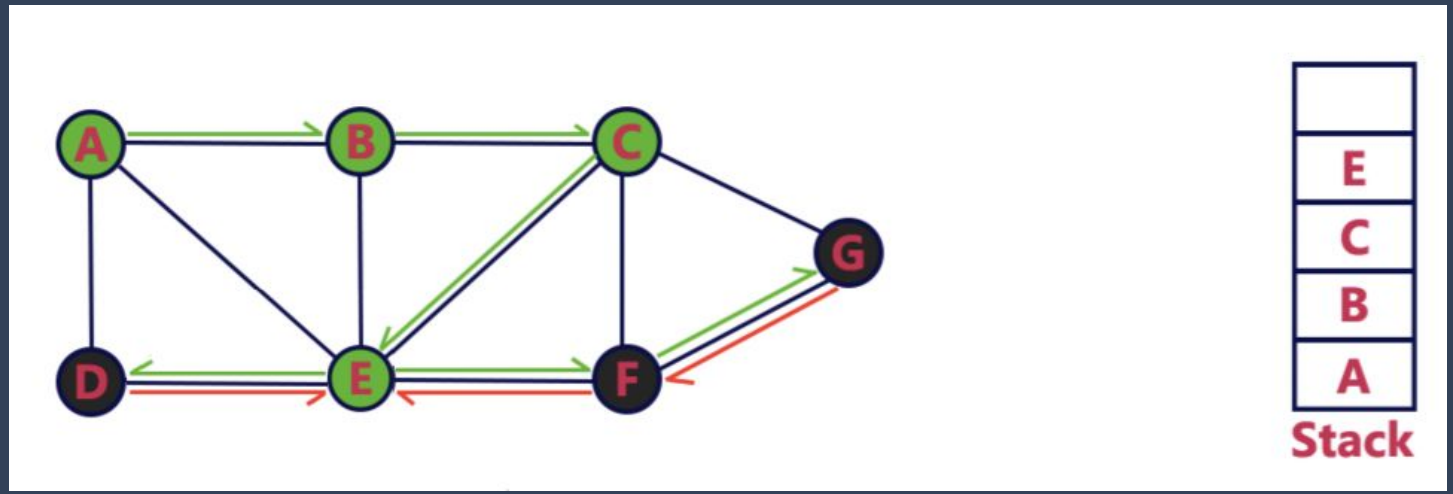
DFS

Step 9: As G doesn't have unvisited neighbour, we are done with G (pop stack)



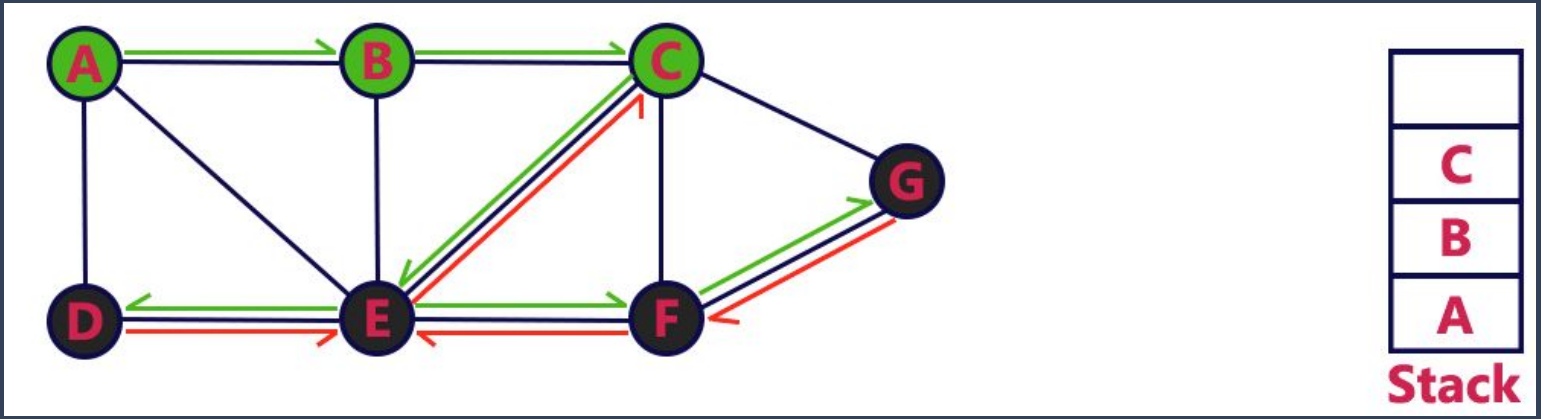
DFS

Step 10: As F doesn’t have unvisited neighbour, we are done with F (pop stack)



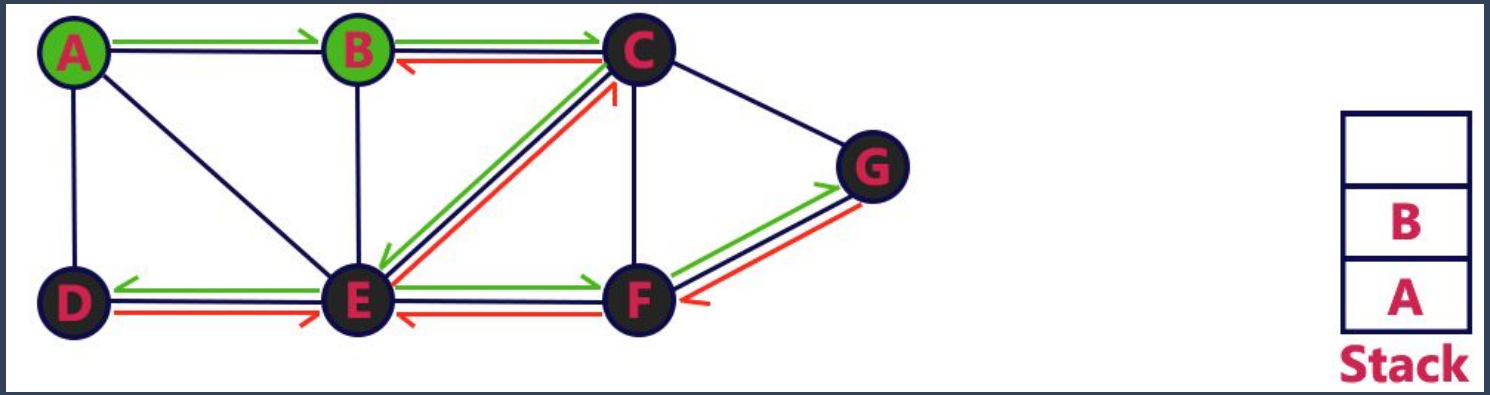
DFS

Step 11: As E doesn't have unvisited neighbour, we are done with E (pop stack)



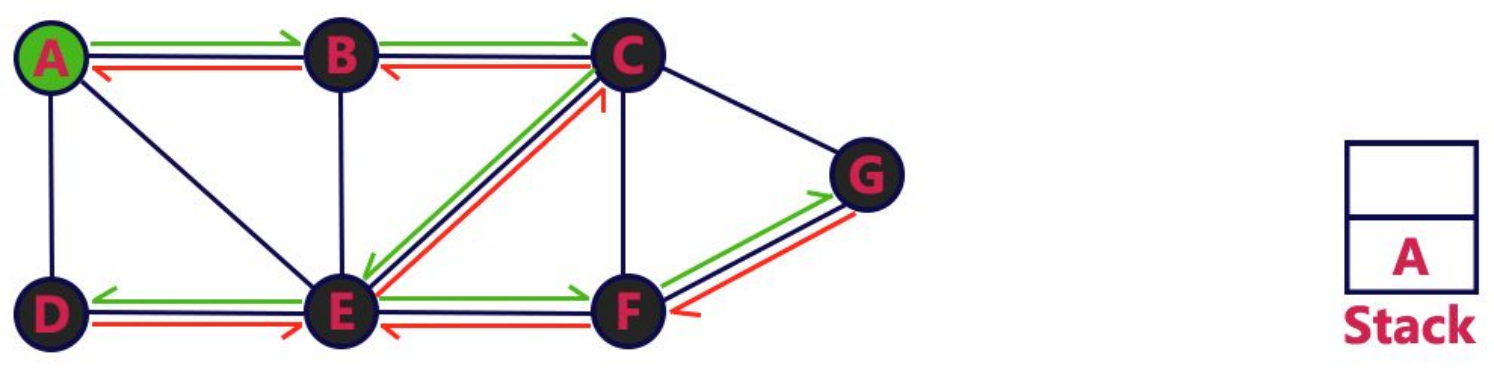
DFS

Step 12: As C doesn’t have unvisited neighbours, we are done with C (pop stack)



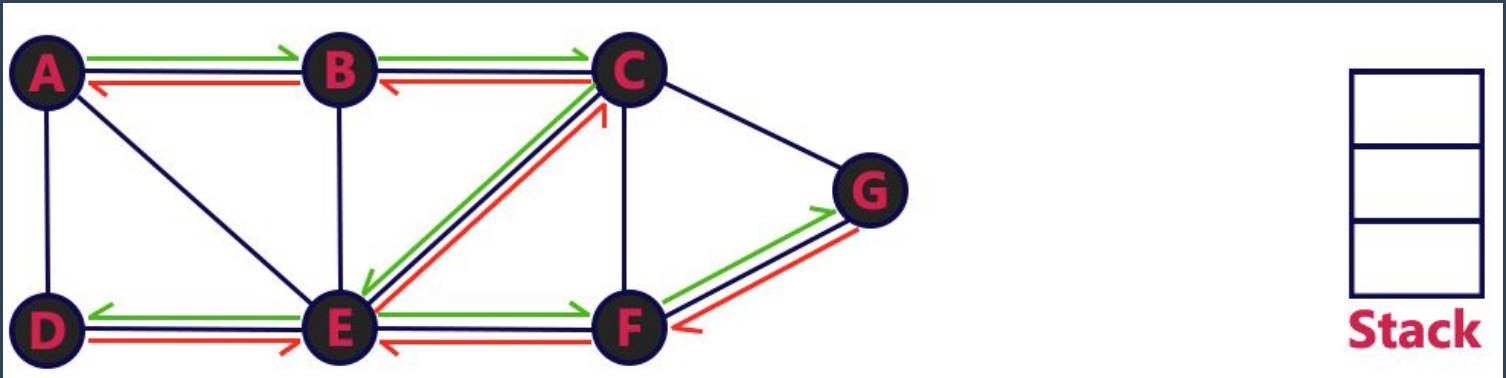
DFS

Step 13: As B doesn’t have unvisited neighbour, we are done with B (pop stack)



DFS

Step 14: As A doesn’t have unvisited neighbour, we are done with A (pop stack)



DFS Implementation

```
class Graph {
    int n;
    Map<Integer, List<Integer>> map;

    // initialization
    public Graph(int n, int[][] edges) {
        this.n = n;
        this.map = new HashMap<>();

        for (int i = 1; i <= n; i++) {
            map.put(i, new LinkedList<>());
        }

        for (int[] edge : edges) {
            map.get(edge[0]).add(edge[1]);
        }
    }
}
```

```
/*
 * Traverse the graph by using DFS
 */
public void dfs() {
    // 0: unvisited
    // 1: being visited
    // 2: already visited
    int[] status = new int[n];
    for (int i = 1; i <= n; i++) {
        if (status[i - 1] == 0) {
            dfs_visit(i, status);
        }
    }
}

public void dfs_visit(int s, int[] status) {
    status[s - 1] = 1; // being visited
    for (int i : map.get(s)) {
        if (status[i - 1] == 0) {
            dfs_visit(i, status);
        }
    }
    status[s - 1] = 2; // already visited
}
```

DFS Implementation (Simplified)

```java
class Graph {
    int n;
    Map<Integer, List<Integer>> map;

    // initialization
    public Graph(int n, int[][] edges) {
        this.n = n;
        this.map = new HashMap<>();

        for (int i = 1; i <= n; i++) {
            map.put(i, new LinkedList<>());
        }

        for (int[] edge : edges) {
            map.get(edge[0]).add(edge[1]);
        }
    }
}
```

```java
/*
 * Traverse the graph by using DFS
 */
public void dfs() {
    boolean[] visited = new boolean[n];
    for (int i = 1; i <= n; i++) {
        if (!visited[i - 1]) {
            dfs_visit(i, visited);
        }
    }
}

public void dfs_visit(int s, boolean[] visited) {
    visited[s - 1] = true;
    for (int i : map.get(s)) {
        if (!visited[i - 1]) {
            dfs_visit(i, visited);
        }
    }
}
```

What Problems Can be Solved by DFS?

1. Traverse all vertices in a graph
2. Whether two vertices are connected or not (preferred approach)
3. Shortest path between two vertices
4. Number of connected components (preferred approach)
5. Dependency path
6. Vertices grouping/coloring
7. Detect cycle in a graph (preferred approach)
8. All possible paths (preferred approach)
9. Shortest path (with cost) between two vertices
10. Minimum Spanning Tree

Question

841. Keys and Rooms

Medium 545 44 Favorite Share

There are `N` rooms and you start in room `0`. Each room has a distinct number in `0, 1, 2, ..., N-1`, and each room may have some keys to access the next room.

Formally, each room `i` has a list of keys `rooms[i]`, and each key `rooms[i][j]` is an integer in `[0, 1, ..., N-1]` where `N = rooms.length`. A key `rooms[i][j] = v` opens the room with number `v`.

Initially, all the rooms start locked (except for room `0`).

You can walk back and forth between rooms freely.

Return `true` if and only if you can enter every room.

Example 1:

```
Input: [[1],[2],[3],[]]
Output: true
Explanation:
We start in room 0, and pick up key 1.
We then go to room 1, and pick up key 2.
We then go to room 2, and pick up key 3.
We then go to room 3.  Since we were able to go to every room, we return true.
```

Example 2:

```
Input: [[1,3],[3,0,1],[2],[0]]
Output: false
Explanation: We can't enter the room with number 2.
```

Keys and Rooms

Idea:

- This is essentially a graph problem (tell why, and how graph is represented.)
- I can use DFS approach to solve this problem (we can also use BFS)
- Keep track of number of visited vertices.
- If the number of visited vertices is NOT equal to the number of vertices in the graph, return false, otherwise, return true.

Solution

```
class Solution {
    public boolean canVisitAllRooms(List<List<Integer>> rooms) {
        Set<Integer> visited = new HashSet<>();
        dfs(0, rooms, visited);
        return visited.size() == rooms.size();
    }

    private void dfs(int room, List<List<Integer>> rooms, Set<Integer> visited) {
        visited.add(room);
        for (int key : rooms.get(room)) {
            if (!visited.contains(key)) {
                dfs(key, rooms, visited);
            }
        }
    }
}
```

399. Evaluate Division

Medium 1517 119 Favorite Share

Equations are given in the format `A / B = k`, where `A` and `B` are variables represented as strings, and `k` is a real number (floating point number). Given some queries, return the answers. If the answer does not exist, return `-1.0`.

Example:
Given `a / b = 2.0, b / c = 3.0.`
queries are: `a / c = ?, b / a = ?, a / e = ?, a / a = ?, x / x = ? .`
return `[6.0, 0.5, -1.0, 1.0, -1.0 ].`

The input is: `vector<pair<string, string>> equations, vector<double>& values, vector<pair<string, string>> queries`, where `equations.size() == values.size()`, and the values are positive. This represents the equations. Return `vector<double>`.

According to the example above:

```
equations = [ ["a", "b"], ["b", "c"] ],
values = [2.0, 3.0],
queries = [ ["a", "c"], ["b", "a"], ["a", "e"], ["a", "a"], ["x", "x"] ].
```

The input is always valid. You may assume that evaluating the queries will result in no division by zero and there is no contradiction.

Evaluate Division

Idea:

- This is essentially a graph problem (tell them why, and how the graph is represented.)
- I will use DFS approach to solve this problem.
- Basically, we need to check whether source vertex and target vertex are connected or not. If they are, we get the weights from source to target vertex.
- We first begin with source vertex and check whether the source vertex and target vertex are one the same edge or not.
- If yes, return the weight. Otherwise, we check whether the neighbors of the source vertex connected with the target vertex or not (recursion may apply). If yes, return the weight, otherwise, return -1.

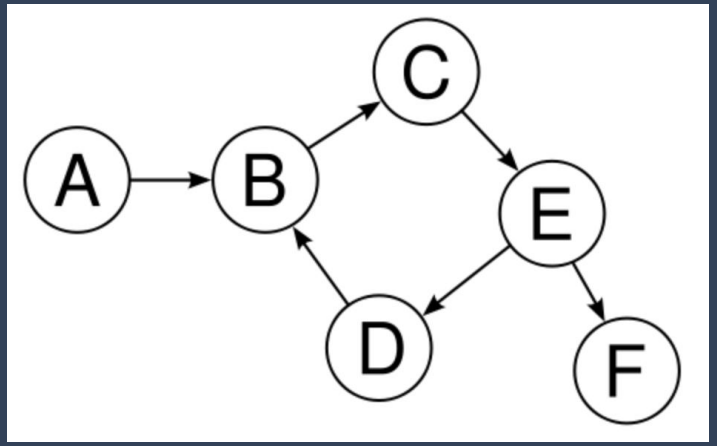
Solution

```java
public class Solution {
    public double[] calcEquation(String[][] equations, double[] values, String[][] queries) {
        // build graph based on edges
        Map<String, Map<String, Double>> graph = buildGraph(equations, values);

        double[] result = new double[queries.length];
        Set<String> visited = new HashSet<>();
        for (int i = 0; i < queries.length; i++) {
            result[i] = getPathWeight(queries[i][0], queries[i][1], visited, graph);
        }
        return result;
    }

    private double getPathWeight(String start, String end, Set<String> visited, Map<String, Map<String, Double>> graph) {
        // exit condition
        if (!graph.containsKey(start)) return -1.0;

        // meet condition
        if (graph.get(start).containsKey(end)) return graph.get(start).get(end);

        // update the status and continue DFS
        visited.add(start);
        
        // restore the status
        visited.remove(start);
        return -1.0;
    }

    private Map<String, Map<String, Double>> buildGraph(String[][] equations, double[] values) {
        // Adjacency map
        Map<String, Map<String, Double>> graph = new HashMap<>();

        for (int i = 0; i < equations.length; i++) {
            String u = equations[i][0];
            String v = equations[i][1];
            graph.putIfAbsent(u, new HashMap<>());
            graph.get(u).put(v, values[i]);
            graph.putIfAbsent(v, new HashMap<>());
            graph.get(v).put(u, 1 / values[i]);
        }
        return graph;
    }
}
```

Solution

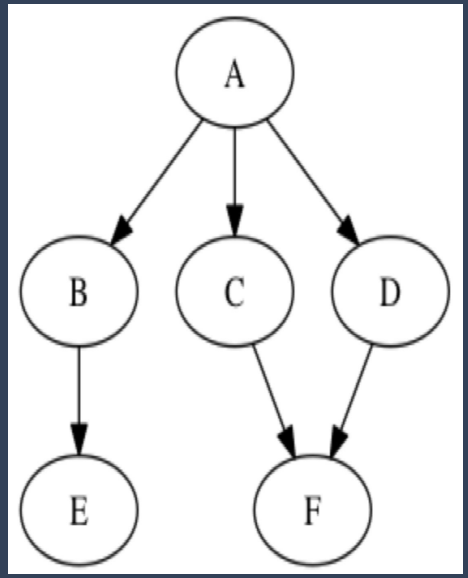
```java
public class Solution {
    public double[] calcEquation(String[][] equations, double[] values, String[][] queries) {
        // build graph based on edges
        Map<String, Map<String, Double>> graph = buildGraph(equations, values);

        double[] result = new double[queries.length];
        Set<String> visited = new HashSet<>();
        for (int i = 0; i < queries.length; i++) {
            result[i] = getPathWeight(queries[i][0], queries[i][1], visited, graph);
        }
        return result;
    }

    private double getPathWeight(String start, String end, Set<String> visited, Map<String, Map<String, Double>> graph) {
        // exit condition
        if (!graph.containsKey(start)) return -1.0;

        // meet condition
        if (graph.get(start).containsKey(end)) return graph.get(start).get(end);

        // update the status and continue DFS
        visited.add(start);
        for (Map.Entry<String, Double> neighbour : graph.get(start).entrySet()) {
            if (!visited.contains(neighbour.getKey())) {
                double productWeight = getPathWeight(neighbour.getKey(), end, visited, graph);
                if (productWeight != -1.0) {
                    return neighbour.getValue() * productWeight;
                }
            }
        }
        // restore the status
        visited.remove(start);
        return -1.0;
    }

    private Map<String, Map<String, Double>> buildGraph(String[][] equations, double[] values) {
        // Adjacency map
        Map<String, Map<String, Double>> graph = new HashMap<>();

        for (int i = 0; i < equations.length; i++) {
            String u = equations[i][0];
            String v = equations[i][1];
            graph.putIfAbsent(u, new HashMap<>());
            graph.get(u).put(v, values[i]);
            graph.putIfAbsent(v, new HashMap<>());
            graph.get(v).put(u, 1 / values[i]);
        }
        return graph;
    }
}
```

How to Detect Cycle in Directed Graph?



Have cycle

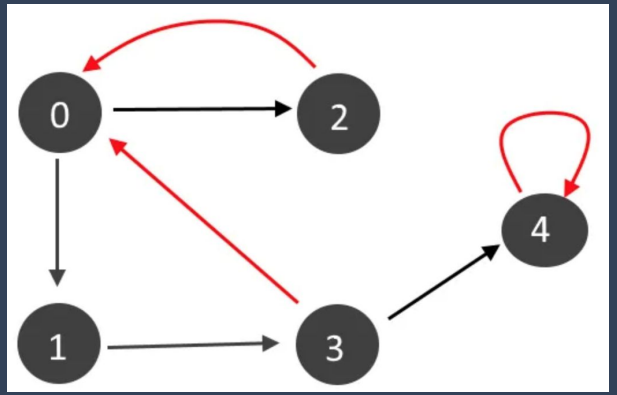
B -> C -> E -> D -> B



No Cycle

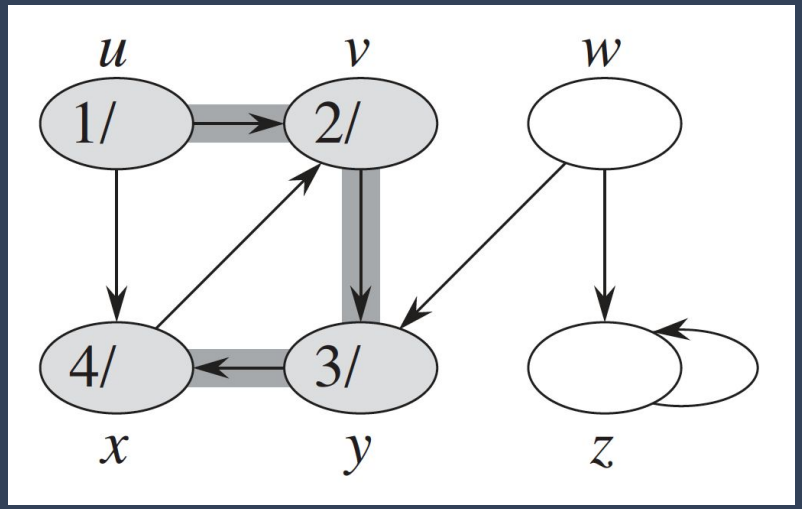
Back Edge

- A **back edge** is an edge that is from a vertex to itself or one of its ancestor in the graph when traversing by using DFS.
- There is a cycle in a graph only if there exists a back edge in the graph.



How to Find Back Edge?

- In DFS traversal, if we visit a vertex whose color is gray, we can say there is a back edge!



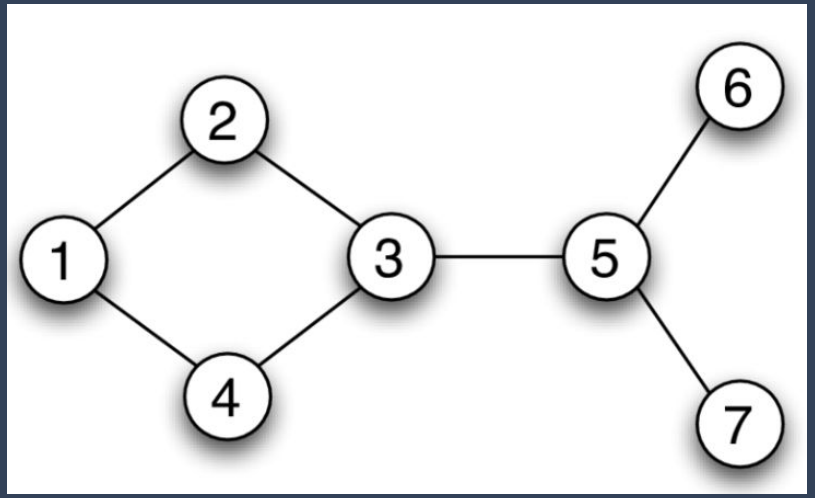

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3  for each vertex u ∈ G.V
4      if u.color == WHITE
5          DFS-VISIT(G, u)

DFS-VISIT(G, u)
1  u.color = GRAY
2  for each v ∈ G.Adj[u]
3      if v.color == WHITE
4          DFS-VISIT(G, v)
5  u.color = BLACK
```

Implementation

```
class DirectedGraph {
    int n;
    HashMap<Integer, List<Integer>> map;

    // initialization
    public DirectedGraph(int n, int[][] edges) {
        this.n = n;
        this.map = new HashMap<>();

        for (int i = 1; i <= n; i++) {
            map.put(i, new LinkedList<>());
        }

        for (int[] edge : edges) {
            map.get(edge[0]).add(edge[1]);
        }
    }

}
```

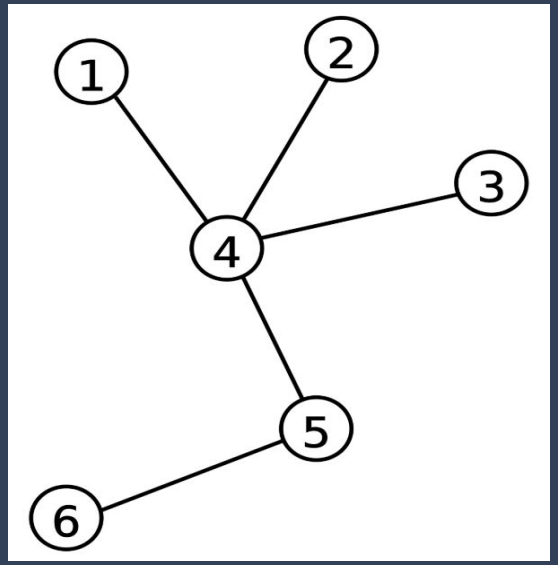
```
/*
 * Detect whether a directed graph has cycle or not
 */
public boolean hasCycle() {
    // 0: unvisited
    // 1: being visited
    // 2: already visited
    int[] status = new int[n];
    for (int i = 1; i <= n; i++) {
        if (status[i - 1] == 0) {
            if (dfs(i, status)) {
                return true;
            }
        }
    }
    return false;
}

private boolean dfs(int s, int[] status) {
    status[s - 1] = 1; // being visited
    for (int i : map.get(s)) {
        if (status[i - 1] == 0 && dfs(i, status)) {
            return true;
        } else if (status[i - 1] == 1) {
            return true; // the vertex is visited by its descendant
        }
    }
    status[s - 1] = 2; // already visited
    return false;
}
```

How to Detect Cycle in Undirected Graph?



Have Cycle

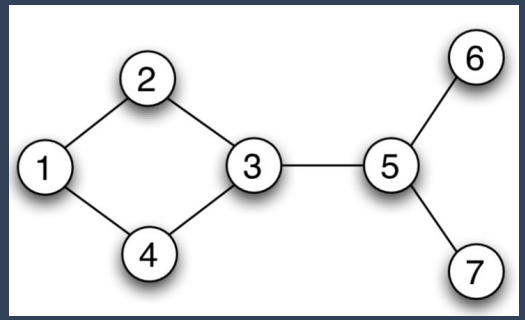
1 -> 2 -> 3 -> 4 -> 1



No Cycle

How to Detect Cycle in Undirected Graph?

- In undirected graph, if we traverse from vertex A to vertex B directly, vertex A is considered as **parent** of vertex B.
- The edge to the parent of a vertex should **NOT** be counted as a back edge, there, we need to **pass in parent in DFS traversal**.
- If we can find any already visited vertex (not parent) in DFS traversal, it indicates there exists a back edge.



How to Detect Cycle in Undirected Graph?

```
class UndirectedGraph {
    int n;
    private Map<Integer, List<Integer>> map;

    // initialization
    public UndirectedGraph(int n, int[][] edges) {
        this.n = n;
        this.map = new HashMap<>();
        for (int i = 1; i <= n; i++) {
            map.put(i, new LinkedList<>());
        }

        for (int[] edge : edges) {
            map.get(edge[0]).add(edge[1]);
        }
    }
}
```

```
/*
 * Detect whether an undirected graph has cycle or not
 */
public boolean hasCycle() {
    // 0: unvisited
    // 1: being visited
    // 2: already visited
    int[] status = new int[n];
    for (int i = 1; i <= n; i++) {
        if (status[i - 1] == 0) {
            if (dfs_visit(i, -1, status)) {
                return true;
            }
        }
    }
    return false;
}

private boolean dfs_visit(int s, int parent, int[] status) {
    status[s - 1] = 1; // being visited
    for (int i : map.get(s)) {
        if (status[i - 1] == 0 && dfs_visit(i, s, status)) {
            return true;
        } else if (status[i - 1] == 1 && i != parent){
            return true; // the vertex is visited by its descendant
        }
    }
    status[s - 1] = 2; // already visited
    return false;
}
```

207. Course Schedule

Medium 2130 99 Favorite Share

There are a total of *n* courses you have to take, labeled from `0` to `n-1`.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: `[0,1]`

Given the total number of courses and a list of prerequisite **pairs**, is it possible for you to finish all courses?

Example 1:

```
Input: 2, [[1,0]]
Output: true
Explanation: There are a total of 2 courses to take.
             To take course 1 you should have finished course 0. So it is possible.
```

Example 2:

```
Input: 2, [[1,0],[0,1]]
Output: false
Explanation: There are a total of 2 courses to take.
             To take course 1 you should have finished course 0, and to take course 0 you should
             also have finished course 1. So it is impossible.
```

Note:

1. The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about how a graph is represented.
2. You may assume that there are no duplicate edges in the input prerequisites.

Course Schedule I

Idea:

- This is essentially a graph problem (tell them why, and how the graph is represented.)
- Basically, we need to check whether there is a cycle in the graph or not.
- I will use DFS to check whether there is a back edge in the graph or not.
- If there is no back edge, the courses can be scheduled, otherwise, they cannot be scheduled.

Solution

```
class Solution {
    public boolean canFinish(int numCourses, int[][] prerequisites) {
        int[] visited = new int[numCourses];
        Map<Integer, List<Integer>> graph = new HashMap<>();
        for (int i = 0; i < prerequisites.length; i++) {
            graph.putIfAbsent(prerequisites[i][1], new LinkedList<>());
            graph.get(prerequisites[i][1]).add(prerequisites[i][0]);
        }

        for (int i = 0; i < numCourses; ++i) {
            if (visited[i] == 0) {
                if (hasCycle(i, graph, visited)) {
                    return false;
                }
            }
        }
        return true;
    }

    private static boolean hasCycle(int i, Map<Integer, List<Integer>> graph, int[] visited) {
        visited[i] = 1;
        for (int neighbor : graph.getOrDefault(i, new LinkedList<>())) {
            if (visited[neighbor] == 0 && hasCycle(neighbor, graph, visited) || visited[neighbor] == 1) {
                return true;
            }
        }
        visited[i] = 2;
        return false;
    }
}
```

Topological Sort

- Real life question: you are given a list of tasks with dependency, determine the execution order of the tasks.

- Topological sort of a **directed** graph is a **linear ordering** of its vertices such that for every directed edge *(u,v)* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering.

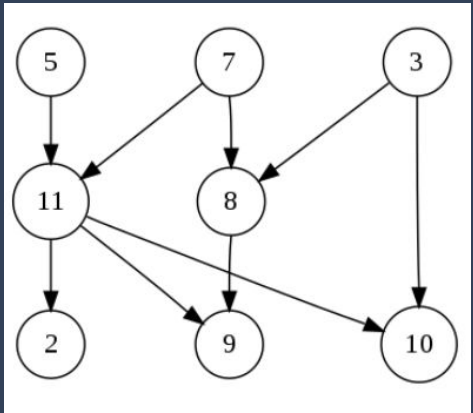
- A topological ordering is possible if and only if the graph has **no cycles**.

Topological Sort Algorithm

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge
while S is non-empty do
    remove a node n from S
    add n to tail of L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S
if graph has unvisited nodes then
    return error   (graph has at least one cycle)
else
    return L   (a topologically sorted order)
```

Topological Sort Algorithm

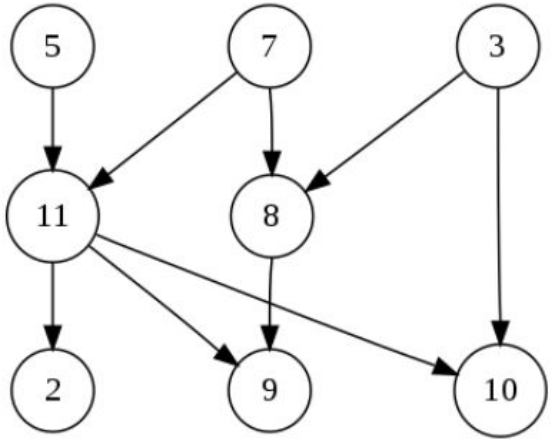
- **Indegree**: For a vertex, the value of indegree is equal to the number of incoming edges.



➔ Indegree of vertex 3, 5, 7: 0
➔ Indegree of vertex 2: 1
➔ Indegree of vertex 8, 9 10, 11: 2

https://en.wikipedia.org/wiki/Topological_sorting

Topological Sort Algorithm



The graph has many valid topological sorts, including:

- 5, 7, 3, 11, 8, 2, 9, 10
- 3, 5, 7, 8, 11, 2, 9, 10
- 5, 7, 3, 8, 11, 10, 9, 2
- 7, 5, 11, 3, 10, 8, 9, 2
- 5, 7, 11, 2, 3, 8, 9, 10
- 3, 7, 8, 5, 11, 10, 2, 9

https://en.wikipedia.org/wiki/Topological_sorting

What Problems Can be Solved by Topological Sort?

1. Traverse all vertices in a graph
2. Whether two vertices are connected or not
3. Shortest path between two vertices
4. Number of disconnected components
5. Dependency path
6. Vertices grouping/coloring
7. Detect cycle in a graph
8. All possible paths
9. Shortest path (with cost) between two vertices
10. Minimum Spanning Tree

207. Course Schedule

Medium 2130 99 Favorite Share

There are a total of *n* courses you have to take, labeled from `0` to `n-1`.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: `[0,1]`

Given the total number of courses and a list of prerequisite **pairs**, is it possible for you to finish all courses?

Example 1:

```
Input: 2, [[1,0]]
Output: true
Explanation: There are a total of 2 courses to take.
             To take course 1 you should have finished course 0. So it is possible.
```

Example 2:

```
Input: 2, [[1,0],[0,1]]
Output: false
Explanation: There are a total of 2 courses to take.
             To take course 1 you should have finished course 0, and to take course 0 you should
             also have finished course 1. So it is impossible.
```

Note:

1. The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about how a graph is represented.
2. You may assume that there are no duplicate edges in the input prerequisites.

Course Schedule I

Idea:

- This is essentially a graph problem (tell them why, and how the graph is represented.)
- Basically, we need to check whether there is a cycle in the graph or not.
- I will use Topological Sort to solve this problem because there is a dependency between courses.
- We first put courses with 0 indegree (no prerequisites) into a queue
- We remove one course from the queue and decrease the indegree of its neighbors.
- If the indegree of the neighbor is 0, we add it to the queue.
- We repeat the last two steps until the queue is empty.
- If all vertices has 0 indegree, we return true, otherwise, return false.

Solution

```
class Node {
    int inDegree;
    List<Node> neighbors;

    public Node() {
        this.inDegree = 0;
        neighbors = new ArrayList<Node>();
    }
}
```

```
class Solution {
    public boolean canFinish(int numCourses, int[][] prerequisites) {
        // initialization
        Map<Integer, Node> map = new HashMap<>();
        for (int i = 0; i < numCourses; i++) {
            map.put(i, new Node());
        }

        for (int[] row : prerequisites) {
            map.get(row[0]).inDegree++;
            map.get(row[1]).neighbors.add(map.get(row[0]));
        }
        // find vertices with 0 indegree
        Queue<Node> queue = new LinkedList<>();
        for (Node node : map.values()) {
            if (node.inDegree == 0) {
                queue.offer(node);
            }
        }
        // topological sort
        while (!queue.isEmpty()) {
            Node node = queue.poll();
            for (Node child : node.neighbors) {
                child.inDegree--;
                if (child.inDegree == 0) {
                    queue.offer(child);
                }
            }
        }
        // has circular dependency
        for (Node node : map.values()) {
            if (node.inDegree != 0) {
                return false;
            }
        }
        return true;
    }
}
```

210. Course Schedule II

Medium 1168 81 Favorite Share

There are a total of *n* courses you have to take, labeled from `0` to `n-1`.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: `[0,1]`

Given the total number of courses and a list of prerequisite **pairs**, return the ordering of courses you should take to finish all courses.

There may be multiple correct orders, you just need to return one of them. If it is impossible to finish all courses, return an empty array.

Example 1:

```
Input: 2, [[1,0]]
Output: [0,1]
Explanation: There are a total of 2 courses to take. To take course 1 you should have finished
             course 0. So the correct course order is [0,1] .
```

Example 2:

```
Input: 4, [[1,0],[2,0],[3,1],[3,2]]
Output: [0,1,2,3] or [0,2,1,3]
Explanation: There are a total of 4 courses to take. To take course 3 you should have finished both
             courses 1 and 2. Both courses 1 and 2 should be taken after you finished course 0.
             So one correct course order is [0,1,2,3]. Another correct ordering is [0,2,1,3] .
```

Course Schedule II

Idea:

- This is essentially a graph problem (tell them why, and how the graph is represented)
- In order to find the orders, I will use Topological Sort because there is a dependency between courses.
- We first put courses with 0 indegree (no prerequisites) into a queue
- We remove one course from the queue, put it into our result list, and decrease the indegree of its neighbors.
- If the indegree of the neighbor is 0, we add it to the queue.
- We repeat the last two steps until the queue is empty.
- If the result list size is the same with number of courses, we return the result list, otherwise, return an empty list.

Solution

```
class Node {
    int inDegree;
    List<Node> neighbors;

    public Node() {
        this.inDegree = 0;
        neighbors = new ArrayList<Node>();
    }
}
```

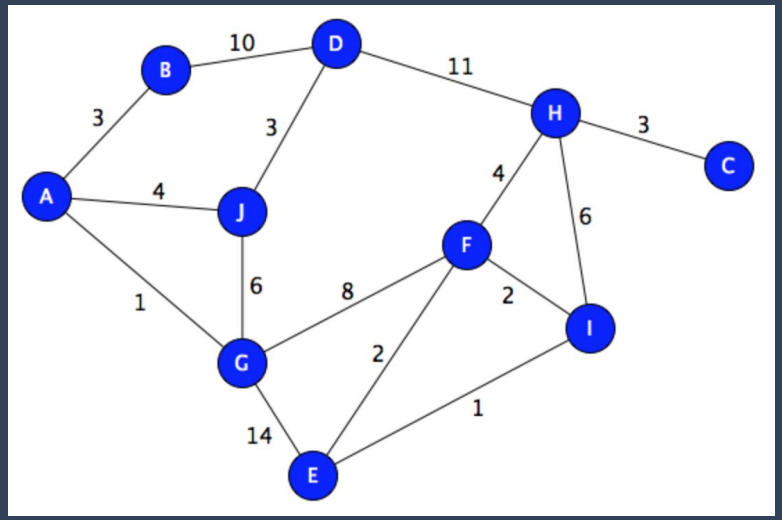
```
class Solution {
    public List<Integer> findOrder(int numCourses, int[][] prerequisites) {
        // initialization
        Map<Integer, Node> map = new HashMap<>();
        for (int i = 0; i < numCourses; i++){
            map.put(i, new Node(i));
        }

        for (int[] row : prerequisites) {
            map.get(row[0]).inDegree++;
            map.get(row[1]).neighbors.add(map.get(row[0]));
        }

        List<Integer> result = new ArrayList<>();

        //find vertices with 0 indegree
        Queue<Node> queue = new LinkedList<>();
        for (Node node : map.values()) {
            if (node.inDegree == 0) {
                queue.offer(node);
            }
        }
        // Topological sort
        while (!queue.isEmpty()) {
            Node node = queue.poll();
            // add vertex to result list
            result.add(node.value);
            for (Node neighbor: node.neighbors) {
                neighbor.inDegree--;
                if (neighbor.inDegree == 0) {
                    queue.offer(neighbor);
                }
            }
        }
        // has dependency cycle
        if (result.size() < numCourses) return new ArrayList<>();
        return result;
    }
}
```

How to Solve Graph Interview Questions

- First determine whether the problem can be modeled as a graph problem.
- Graph and its representation
- What are the most common graph problems
- How to solve graph problems
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)
  - Topological sort
  - Dijkstra’s algorithm (optional)
  - Prim's algorithm (optional)
  - Kruskal's algorithm (optional)
  - ……

Shortest Path in Weighted Graph

- Problem: find a path between two vertices in a **weighted graph** such that the total sum of the edges weights is minimized.



https://cs.lmu.edu/~ray/notes/graphalgorithms/

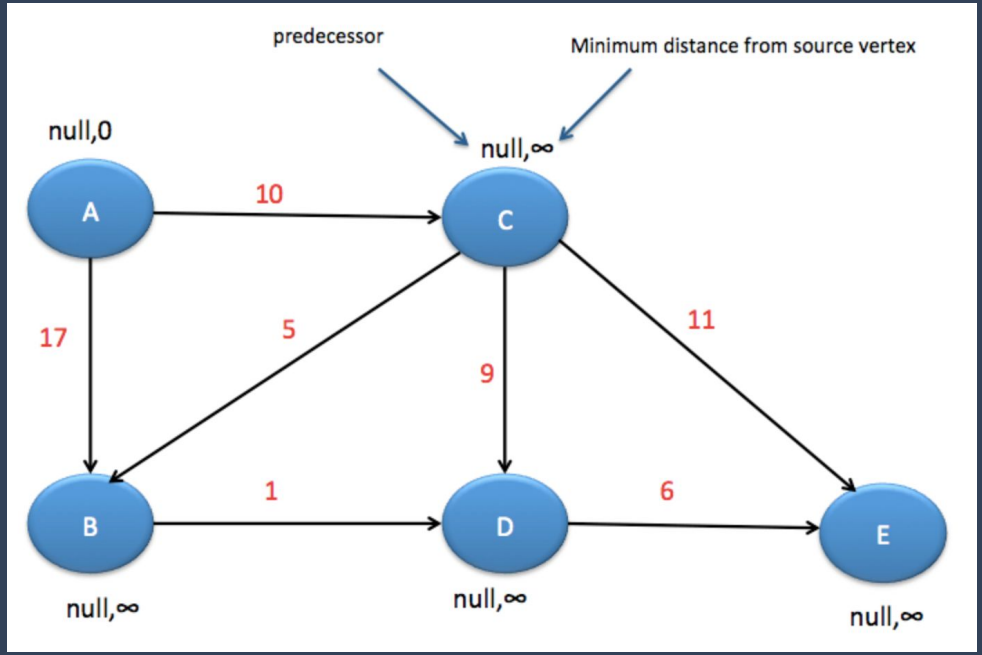
Dijkstra's Algorithm

```
Function Dijkstra(v1, v2)
1    initialize every vertex to have a cost of infinity
2    set v1's cost to 0
3    pqueue = {v1}   // min_queue
4    while (queue is not empty)
5        v ← dequeue vertex from queue with minimum cost
6            mark v as visited
7            if (v == v2) stop
8                for each unvisited neighbor n of v
9                    new_cost ← v's cost + weight of edge (v, n)
10                   if (new_cost < n's cost)
11                       set n's cost to new_cost, and n's previous to v.
12                       enqueue n in the pqueue or update its cost if it was already in the queue
13   reconstruct path from v2 back to v1, following previous pointer
```

Dijkstra's Algorithm

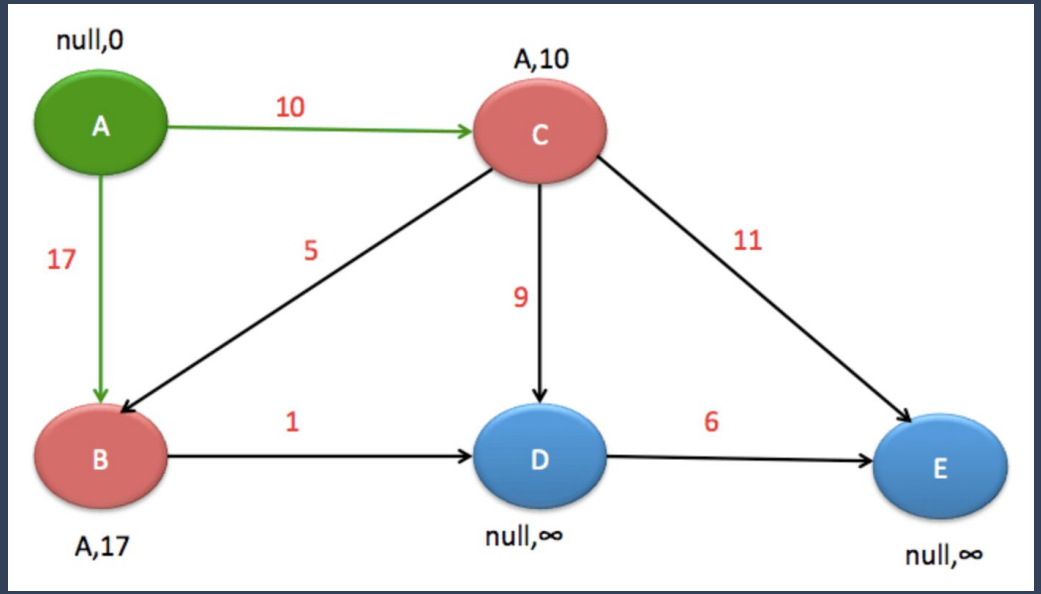
```
Function Dijkstra(v1)
1    initialize every vertex to have a cost of infinity
2    set v1's cost to 0
3    pqueue = {v1}    // min_queue
4    while (queue is not empty)
5        v ← dequeue vertex from queue with minimum cost
6            mark v as visited
7                for each unvisited neighbor n of v
8                    new_cost ← v's cost + weight of edge (v, n)
9                    if (new_cost < n's cost)
10                       set n's cost to new_cost, and n's previous to v.
11                       enqueue n in the pqueue or update its cost if it was already in the queue
12   reconstruct path from all vertices back to v1, following previous pointer
```

Dijkstra's Algorithm



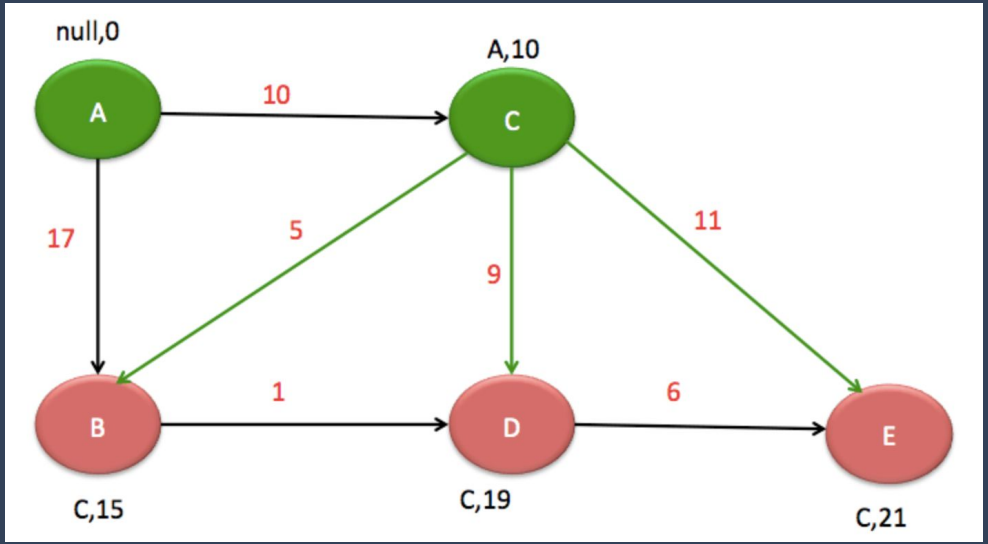
https://java2blog.com/dijkstra-java/

Dijkstra's Algorithm



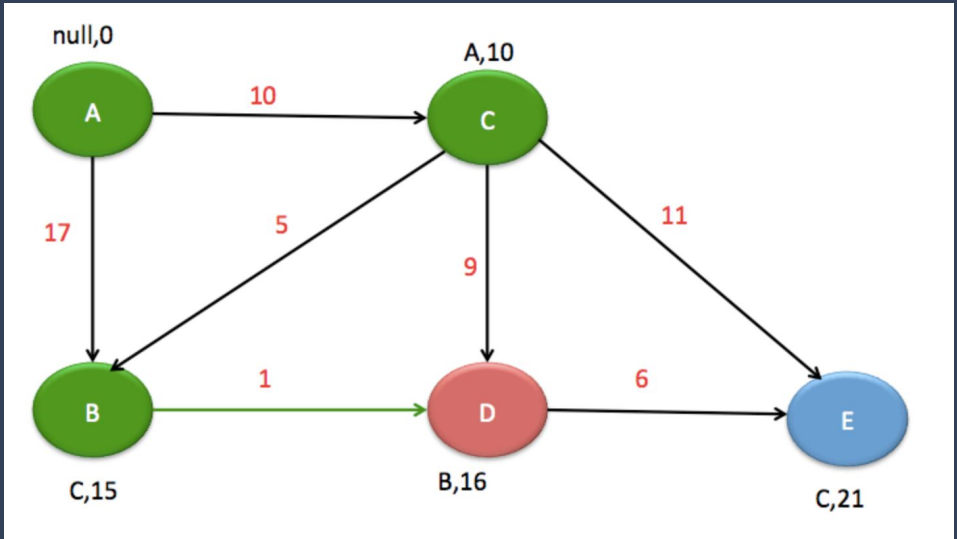
https://java2blog.com/dijkstra-java/

Dijkstra's Algorithm



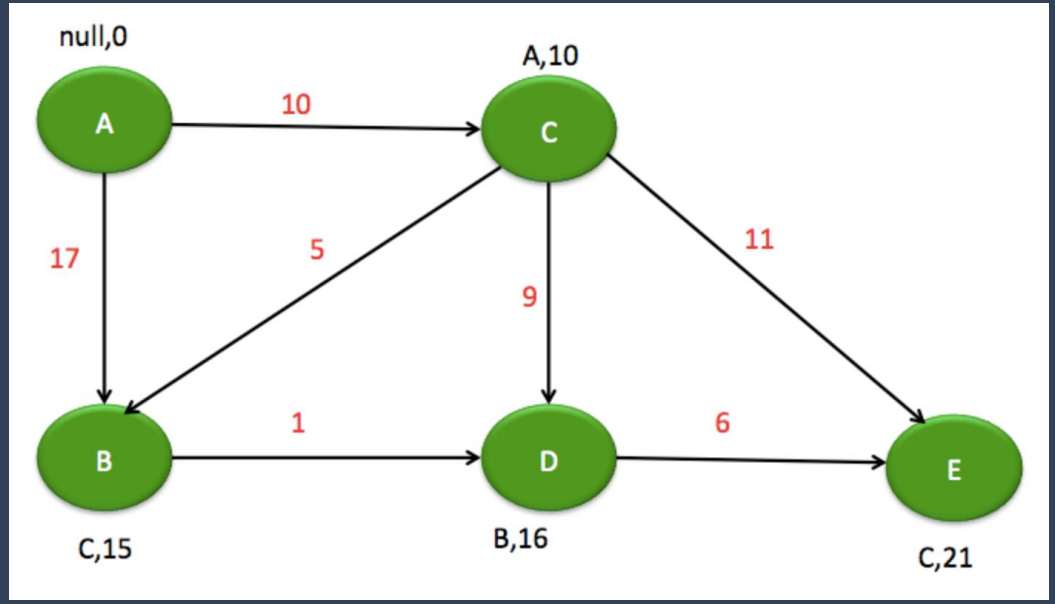
https://java2blog.com/dijkstra-java/

Dijkstra's Algorithm



https://java2blog.com/dijkstra-java/

Dijkstra's Algorithm



https://java2blog.com/dijkstra-java/

Implementation

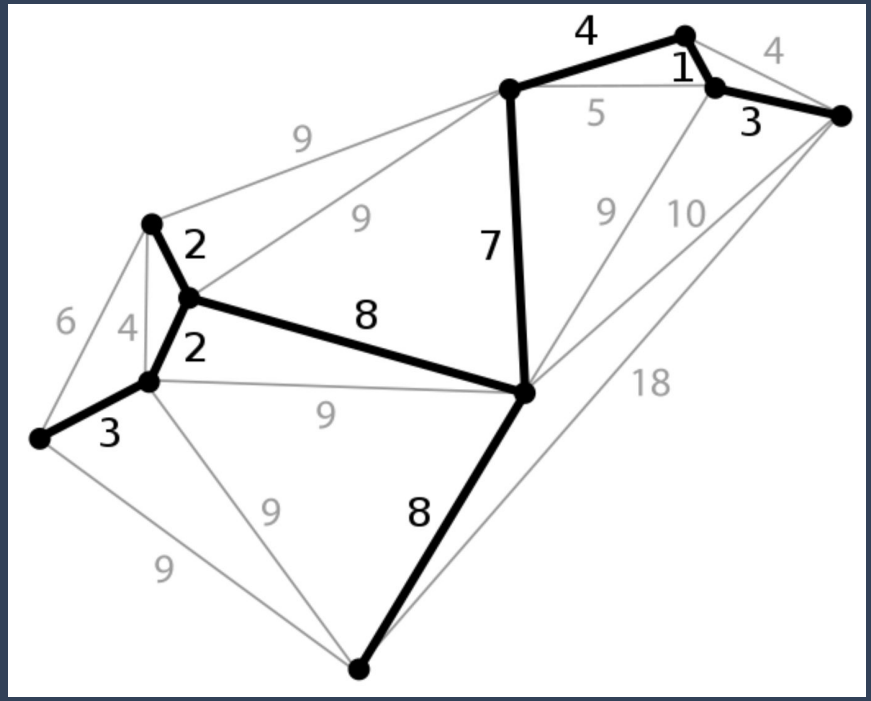
```
class Vertex {
    public String name;
    public List<Edge> adjacenciesList;
    public boolean visited;
    public Vertex predecessor;
    public double distance = Double.MAX_VALUE;

    public Vertex(String name) {
        this.name = name;
        this.adjacenciesList = new ArrayList<>();
    }

    public void addNeighbour(Edge edge) {
        this.adjacenciesList.add(edge);
    }
}

class Edge {
    public double weight;
    public Vertex targetVertex;

    public Edge(double weight, Vertex targetVertex) {
        this.weight = weight;
        this.targetVertex = targetVertex;
    }
}
```

```
public class DijkstraShortestPath {
    public void computeShortestPaths(Vertex sourceVertex) {
        sourceVertex.distance = 0;
        PriorityQueue<Vertex> priorityQueue =
            new PriorityQueue<>((v1, v2) -> Double.compare(v1.distance, v2.distance));
        priorityQueue.add(sourceVertex);

        while (!priorityQueue.isEmpty()) {
            Vertex vertex = priorityQueue.poll();
            vertex.visited = true;
            for (Edge edge : vertex.adjacenciesList) {
                Vertex neighbor = edge.targetVertex;
                if (!neighbor.visited) {
                    double newDistance = vertex.distance + edge.weight;
                    if (newDistance < neighbor.distance) {
                        priorityQueue.remove(neighbor);
                        neighbor.distance = newDistance;
                        neighbor.predecessor = vertex;
                        priorityQueue.add(neighbor);
                    }
                }
            }
        }
    }

    public List<Vertex> getShortestPathTo(Vertex targetVertex) {
        List<Vertex> path = new ArrayList<>();
        Vertex vertex = targetVertex;
        while (vertex != null) {
            path.add(vertex);
            vertex = vertex.predecessor;
        }
        Collections.reverse(path);
        return path;
    }
}
```

Minimum Spanning Tree

- MST is a **subset** of the edges that connects all the vertices of a **connected**, **edge-weighted undirected** graph, without any cycles and with the minimum possible total edge weight.

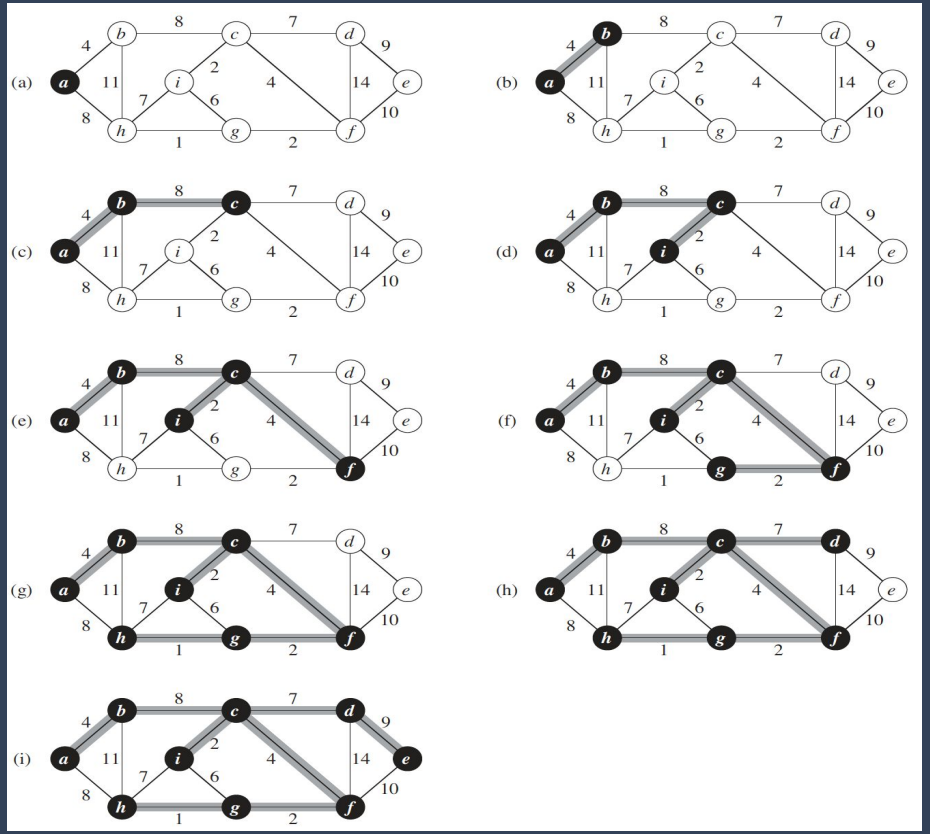
- Cable/Pipeline problem



Introduction to Algorithms, 3rd Edition by Thomas H. Cormen, Charles E. Leiserso, Ronald L. Rivest, Clifford Stein

Prim's Algorithm

```
MST-PRIM(G, w, r)
 1  for each u ∈ G.V
 2      u.key = ∞
 3      u.π = NIL
 4  r.key = 0
 5  Q = G.V
 6  while Q ≠ ∅
 7      u = EXTRACT-MIN(Q)
 8      for each v ∈ G.Adj[u]
 9          if v ∈ Q and w(u, v) < v.key
10              v.π = u
11              v.key = w(u, v)
```



Introduction to Algorithms, 3rd Edition by Thomas H. Cormen, Charles E. Leiserso, Ronald L. Rivest, Clifford Stein

Prim's Algorithm

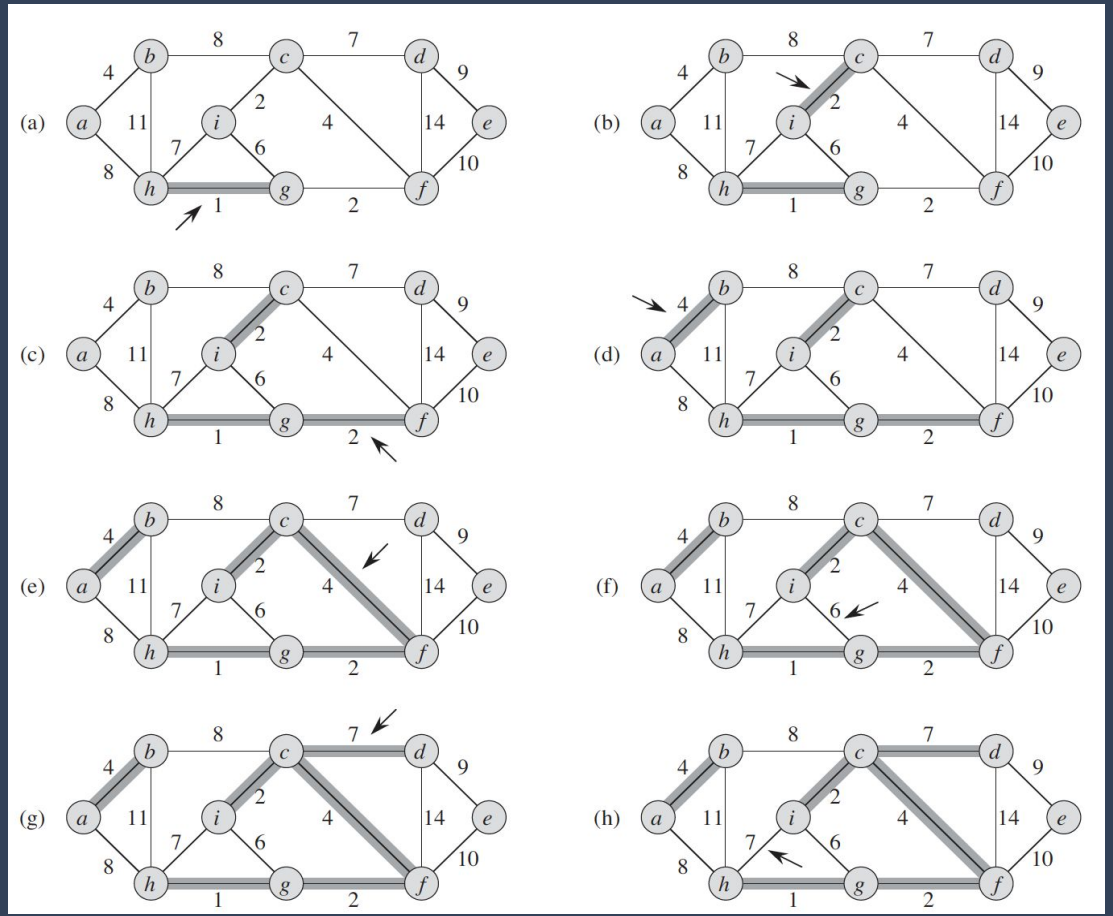
```java
class Vertex {
    public String name;
    public List<Edge> adjacenciesList;
    public boolean visited;
    public Vertex predecessor;
    public double weight = Double.MAX_VALUE;

    public Vertex(String name) {
        this.name = name;
        this.adjacenciesList = new ArrayList<>();
    }

    public void addNeighbour(Edge edge) {
        this.adjacenciesList.add(edge);
    }
}

class Edge {
    public double weight;
    public Vertex targetVertex;

    public Edge(double weight, Vertex targetVertex) {
        this.weight = weight;
        this.targetVertex = targetVertex;
    }
}
```

```java
public class Prims_MST {
    public int Prims(List<Vertex> V, Vertex sourceVertex) {
        int cost = 0;
        sourceVertex.weight = 0;

        Queue<Vertex> pq = new PriorityQueue<Vertex>(
            (vertex1, vertex2) -> Double.compare(vertex1.weight, vertex2.weight));
        pq.addAll(V);

        while (!pq.isEmpty()) {
            Vertex current = pq.poll();
            current.visited = true;
            cost += current.weight;
            for (Edge edge : current.adjacenciesList) {
                Vertex neighbor = edge.targetVertex;
                if (!neighbor.visited && neighbor.weight < edge.weight) {
                    neighbor.predecessor = current;
                    neighbor.weight = edge.weight;
                }
            }
        }
        return cost;
    }
}
```

Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3       MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6       if FIND-SET(u) ≠ FIND-SET(v)
7            A = A ∪ {(u, v)}
8            UNION(u, v)
9  return A
```

Introduction to Algorithms, 3rd Edition by Thomas H. Cormen, Charles E. Leiserso, Ronald L. Rivest, Clifford Stein



Introduction to Algorithms, 3rd Edition by Thomas H. Cormen, Charles E. Leiserso, Ronald L. Rivest, Clifford Stein

Kruskal's Algorithm

```java
public class Kruskal_MST {
    public List<Edge> kruskal(String[] vertices, List<Edge> edges) {
        UnionFind uf = new UnionFind(vertices);
        List<Edge> results = new ArrayList<>();
        Collections.sort(edges, (edge1, edge2) -> Double.compare(edge1.weight, edge2.weight));

        for (Edge edge : edges) {
            String start = edge.startVertex;
            String end = edge.endVertex;
            int countBeforeUnion = uf.getCount();
            uf.union(start, end);
            int countAfterUnion = uf.getCount();
            if (countBeforeUnion != countAfterUnion) {
                results.add(edge);
            }
        }
        return results;
    }
}

class Edge {
    public double weight;
    public String startVertex;
    public String endVertex;

    public Edge(double weight, String startVertex, String endVertex) {
        this.weight = weight;
        this.startVertex = startVertex;
        this.endVertex = endVertex;
    }
}
```

```java
class UnionFind {
    private Map<String, String> parent;
    private Map<String, Integer> rank;
    private int count;

    public UnionFind(String[] vertices) {
        this.count = vertices.length;
        parent = new HashMap<>();
        rank = new HashMap<>();
        for (String vertex : vertices) {
            parent.put(vertex, vertex);
            rank.put(vertex, 0);
        }
    }

    public String find(String p) {
        while (p != parent.get(p)) {
            String ancester = parent.get(parent.get(p));
            parent.put(p, ancester);
            p = parent.get(p);
        }
        return p;
    }

    public void union(String p, String q) {
        String rootP = find(p);
        String rootQ = find(q);
        if (rootP.equals(rootQ)) {
            return;
        } else if (rank.get(rootQ) > rank.get(rootP)) {
            parent.put(rootP, rootQ);
        } else {
            parent.put(rootQ, rootP);
            if (rank.get(rootQ) == rank.get(rootP)) {
                rank.put(rootP, rank.get(rootP) + 1);
            }
        }
        count--;
    }

    public int getCount() {
        return count;
    }
}
```

How to Solve Graph Interview Questions

- First determine whether the problem can be modeled as a graph problem.
- Graph and its representation
- What are the most common graph problems
- How to solve graph problems
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)
  - Topological Sort
  - Dijkstra’s algorithm (optional)
  - Prim's algorithm (optional)
  - Kruskal's algorithm (optional)
  - ……