**Graph**

Table of Contents

[**Theory : Basics of Graph + Graph Traversal** 2](#_Toc171443908)

[1.1 What is a Graph? 2](#_Toc171443909)

[1.2 Basic Terminology 2](#_Toc171443910)

[1.3 Basic Types of Graphs 2](#_Toc171443911)

[1.4 Graph representation 4](#_Toc171443912)

[Q1. https://www.geeksforgeeks.org/problems/print-adjacency-list-1587115620/1 5](#_Toc171443913)

[2.1 Depth-First Search (DFS) 5](#_Toc171443914)

[2.2 Breadth-First Search (BFS) 5](#_Toc171443915)

[2.1 Bipartite Graph 8](#_Toc171443916)

[1. Number of provinces 10](#_Toc171443917)

[2. Find number of islands 10](#_Toc171443918)

[3. Bipartite Graph 11](#_Toc171443919)

[4. Undirected Graph cycle 12](#_Toc171443920)

[Explanation: 16](#_Toc171443921)

[5. Implement Dijkstra Algorithm. 17](#_Toc171443922)

[**Questions: Basics of Graph + Graph Traversal** 18](#_Toc171443923)

[1. Create Adjacency list 18](#_Toc171443924)

[LEVEL 2: **Medium** 19](#_Toc171443925)

[LEVEL 3: **Difficult** 20](#_Toc171443926)

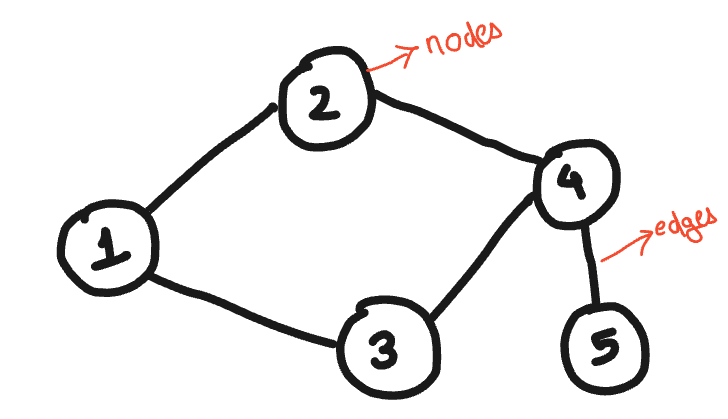
[**SOLUTIONS:** 21](#_Toc171443927)

[**Solutions: Basics of Graph + Graph Traversal** 21](#_Toc171443928)

# **Theory : Basics of Graph + Graph Traversal**

## 1.1 What is a Graph?

**Definition**: A graph G=(V,E)G = (V, E)G=(V,E) consists of a set of vertices V and a set of edges E.



## 1.2 Basic Terminology

* **Vertex (Node)**: An individual entity in a graph.
* **Edge (Link)**: Connection between two vertices.
* **Degree**: Number of edges connected to a vertex.
  + **In-degree** and **Out-degree** in directed graphs.
* **Path**: Sequence of edges connecting vertices.
* **Cycle**: Path that starts and ends at the same vertex.
* **Component**: A maximally connected subgraph.

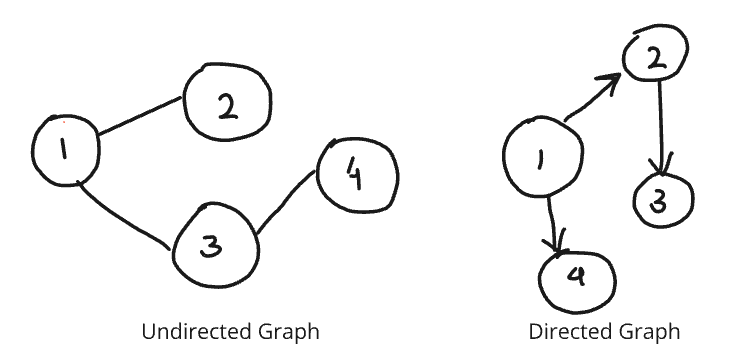
## 1.3 Basic Types of Graphs

1. **Undirected Graph**:

A graph where edges have no direction. The edge (u, v) is identical to (v, u).

1. **Directed Graph (Digraph)**:

A graph where edges have a direction. The edge (u, v) is not the same as (v, u).

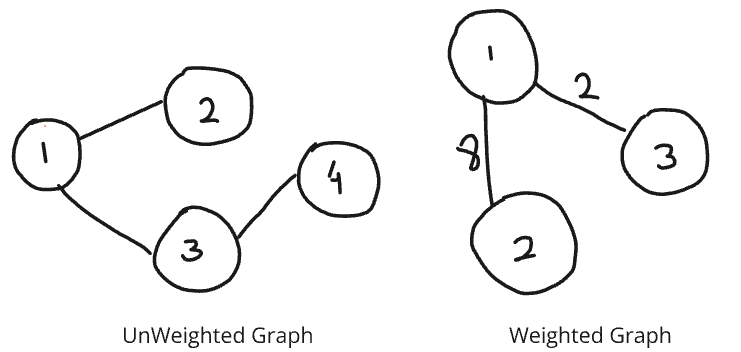


1. **Weighted Graph**:

A graph where edges have weights representing costs, distances, or capacities.

1. **Unweighted Graph**:

A graph where edges do not have weights.

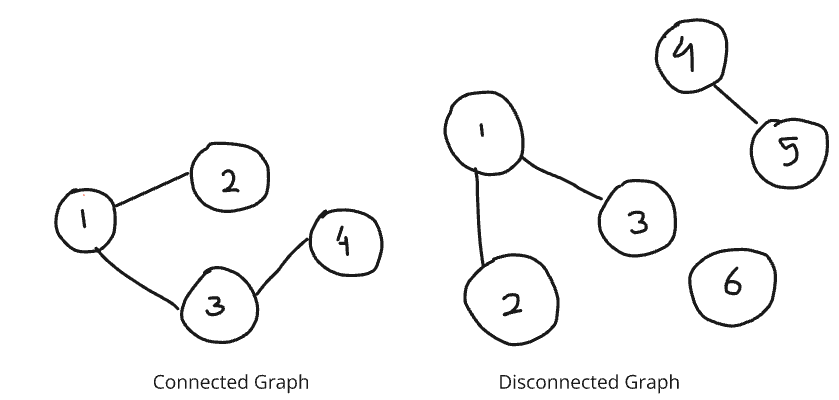


1. **Connected Graph**:

A graph in which there is a path between every pair of vertices.

1. **Disconnected Graph**:

A graph that is not connected; it consists of multiple connected components.



1. **Complete Graph (Kn)**:

A graph where there is an edge between every pair of vertices.

1. **Cycle Graph (Cn)**:

A graph that forms a single cycle where each vertex has exactly two neighbors.

1. **Tree**:

A connected acyclic graph. Trees have a hierarchical structure with a root node.

1. **Forest**:

A collection of disjoint trees.

1. **Bipartite Graph**:

A graph whose vertices can be divided into two disjoint sets U and V such that every edge connects a vertex in U to a vertex in V.

1. **Directed Acyclic Graph (DAG)**:

A directed graph with no cycles. DAGs are used in various applications like scheduling, representing hierarchies, and data flow diagrams.

## 1.4 Graph representation

# Adjacency List Representation

graph = {

    'A': ['B', 'C'],

    'B': ['A', 'D', 'E'],

    'C': ['A', 'F'],

    'D': ['B'],

    'E': ['B', 'F'],

    'F': ['C', 'E']

}

# Adjacency Matrix Representation

import numpy as np

adj\_matrix = np.array([

    [0, 1, 1, 0, 0, 0],

    [1, 0, 0, 1, 1, 0],

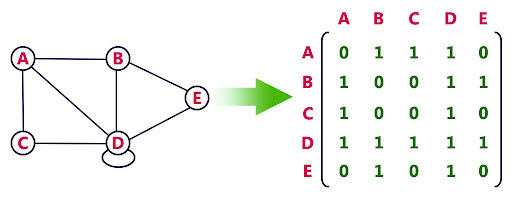
    [1, 0, 0, 0, 0, 1],

    [0, 1, 0, 0, 0, 0],

    [0, 1, 0, 0, 0, 1],

    [0, 0, 1, 0, 1, 0]

])



Adjacency matrix

## 1.5. Depth-First Search (DFS)

* **Theory**: Explore as far as possible along each branch before backtracking.
* **Algorithm**:
  1. Start at the root (or any arbitrary node).
  2. Mark the node as visited.
  3. Recursively visit all unvisited adjacent nodes.

def dfs(graph, start, visited=None):

    if visited is None:

        visited = set()

    visited.add(start)

    print(start)

    for neighbour in graph[start]:

        if neighbour not in visited:

            dfs(graph, neighbour, visited)

    return visited

# Example Usage

graph = {

    'A': ['B', 'C'],

    'B': ['A', 'D', 'E'],

    'C': ['A', 'F'],

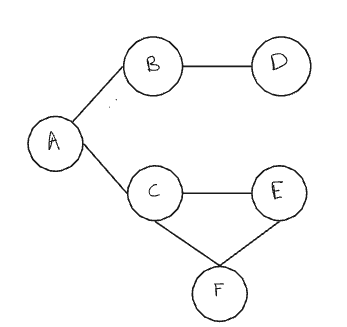
    'D': ['B'],

    'E': ['B', 'F'],

    'F': ['C', 'E']

}

dfs(graph, 'A')

****

OUTPUT: A B D E F C

## 1.6. Breadth-First Search (BFS)

* **Theory**: Explore all neighbors at the present depth prior to moving on to nodes at the next depth level.
* **Algorithm**:
  1. Start at the root (or any arbitrary node).
  2. Mark the node as visited.
  3. Add the node to the queue.
  4. While the queue is not empty:
     + Dequeue a node.
     + Visit all unvisited adjacent nodes and add them to the queue.

from collections import deque

def bfs(graph, start):

    visited = set()

    queue = deque([start])

    visited.add(start)

    while queue:

        vertex = queue.popleft()

        print(vertex)

        for neighbor in graph[vertex]:

            if neighbor not in visited:

                visited.add(neighbor)

                queue.append(neighbor)

graph = {

    'A': ['B', 'C'],

    'B': ['A', 'D', 'E'],

    'C': ['A', 'F'],

    'D': ['B'],

    'E': ['B', 'F'],

    'F': ['C', 'E']

}

# Example Usage

bfs(graph, 'A')

#Output: A B c D E F

### 2.1 Bipartite Graph

#### Definition:

A bipartite graph is a type of graph in which the set of vertices can be divided into two disjoint sets UUU and VVV such that every edge connects a vertex in UUU to a vertex in VVV. In other words, no two vertices within the same set are adjacent.

#### Conditions:

1. **Two Colorability**: A graph is bipartite if and only if it is possible to color the vertices using two colors such that no two adjacent vertices have the same color.
2. **No Odd-Length Cycles**: A graph is bipartite if and only if it does not contain any odd-length cycles.(means all graph without cycle or graph with even cycle length are bipartite)

#### Logic to Check if a Graph is Bipartite:

To check if a graph is bipartite, we can use a breadth-first search (BFS) or depth-first search (DFS) approach to attempt to color the graph using two colors. If we can successfully color the graph such that no two adjacent vertices share the same color, then the graph is bipartite. If we encounter a situation where an adjacent vertex needs to be the same color, the graph is not bipartite.

from collections import deque

def is\_bipartite(graph):

    color = {}

    for node in graph:

        if node not in color:

            queue = deque([node])

            color[node] = 0

            while queue:

                current = queue.popleft()

                for neighbor in graph[current]:

                    if neighbor not in color:

                        color[neighbor] = 1 - color[current]

                        queue.append(neighbor)

                    elif color[neighbor] == color[current]:

                        return False

    return True

# Example Usage

graph = {

    'A': ['B', 'C'],

    'B': ['A', 'D'],

    'C': ['A', 'D'],

    'D': ['B', 'C']

}

print(is\_bipartite(graph))  # Output: True

graph = {

    'A': ['B', 'C'],

    'B': ['A', 'C'],

    'C': ['A', 'B']

}

print(is\_bipartite(graph))  # Output: False

### Number of provinces

Link: <https://www.geeksforgeeks.org/problems/number-of-provinces/1>

from collections import deque

class Solution:

    def numProvinces(self, adj, V):

        # code here

        #approach: in visited array try to visit all, if not able to visit from sigle vertex, then it is new province

        #here we are given adjacency matrix. lets convert it to adjacency list first

        adj\_l = [[] for \_ in range(V)]

        for i in range(len(adj)):

            for j in range(len(adj[0])):

                if adj[i][j]==1 and i!=j:

                    adj\_l[i].append(j)

        visited = [0]\*V

        count=0

        for i in range(V):

            if visited[i] !=1:

                count+=1

                visited[i]=1

                queue = deque([i])

                while(queue):  #travel all connected nodes of node(i)

                    t = queue.popleft()

                    for neighbour in adj\_l[t]:

                        if visited[neighbour]==0:

                            visited[neighbour]=1

                            queue.append(neighbour)

        return count

### Find number of islands

Link: <https://www.geeksforgeeks.org/problems/find-the-number-of-islands/1>

#User function Template for python3

import sys

sys.setrecursionlimit(10\*\*8)

class Solution:

    #arrays as passed by reference, so changes in visited array made in dfs reflects here too

    def dfs(self,grid,visited,i,j,n,m):

        visited[i][j]=1

        delta = [-1,0,1]  #all neighbour are of type [i+delta][j+delta] , 0-0 is that node itself

        for del\_row in delta:

            for del\_col in delta:

                new\_row = i + del\_row

                new\_col = j + del\_col

                #means neighbour value in grid is 1 and it is not visited

                if(new\_row>=0 and new\_row<n and new\_col>=0 and new\_col<m and grid[new\_row][new\_col]==1 and visited[new\_row][new\_col]==0):

                    visited[new\_row][new\_col]=1

                    self.dfs(grid,visited,new\_row,new\_col,n,m)

    def numIslands(self,grid):

        #we can consider matrix as graph, all connected 1 as one component, and

        #finally we want number of connected components

        n=len(grid)

        m=len(grid[0])

        visited=[[0]\*m for \_ in range(n)]

        count=0

        for i in range(n):

            for j in range(m):

                if grid[i][j]==1 and visited[i][j]!=1:

                    count+=1

                    self.dfs(grid,visited,i,j,n,m)

        return count

### Bipartite Graph

Link: <https://www.geeksforgeeks.org/problems/bipartite-graph/1>

from collections import deque

class Solution:

    def isBipartite(self, V, adj):

        color = {}

        #there can be many disconnected components of graph, therefore running loops to cover all

        for i in range(V):

            if i not in color:

                color[i]=0

                queue = deque([i])

                while queue:

                    node = queue.popleft()

                    for neighbour in adj[node]:

                        if neighbour not in color:

                            color[neighbour] = 1-color[node]

                            queue.append(neighbour)

                        else:

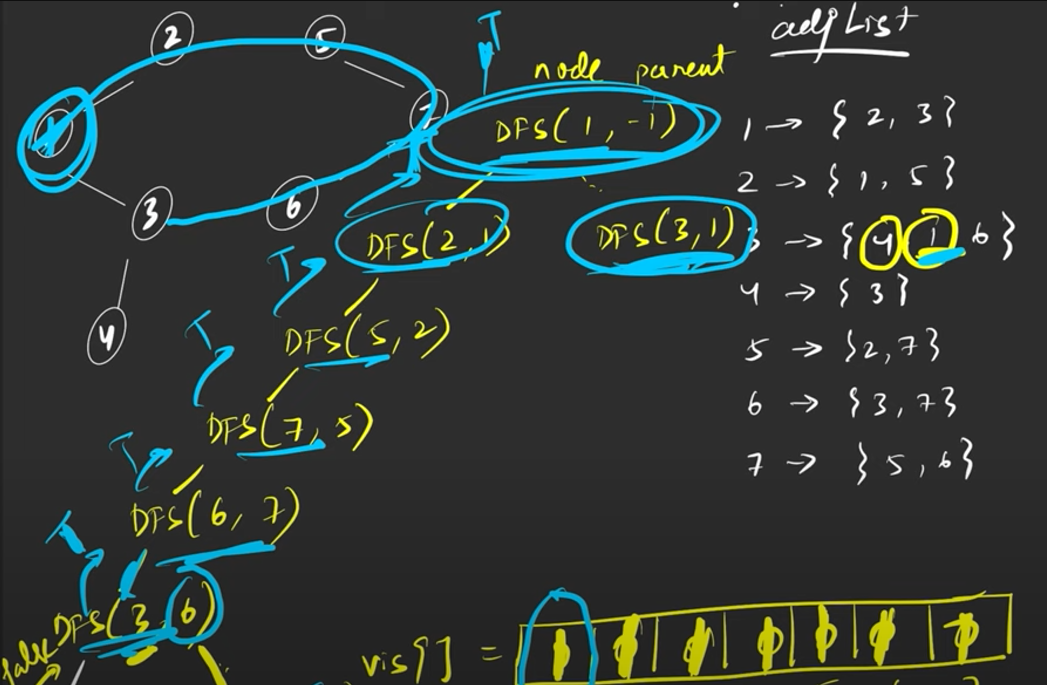
                            if color[neighbour]==color[node]:

                                return False

        return True

### Undirected Graph cycle

Link: <https://www.geeksforgeeks.org/problems/detect-cycle-in-an-undirected-graph/1>

****

from typing import List

class Solution:

    #Function to detect cycle in an undirected graph.

    def dfs(self,node,parent,visited,adj):

        visited.add(node)

        for neighbour in adj[node]:

            if neighbour == parent:

                continue

            elif neighbour in visited:

                return True

            #if by chance we get true, means found cycle, so return it

            #if this subpart is returning false, we have to check for other place. So don't return anything from this subpart

            if self.dfs(neighbour,node,visited,adj):

                return True

        return False

    def isCycle(self, V: int, adj: List[List[int]]) -> bool:

        visited = set()

        for i in range(V):

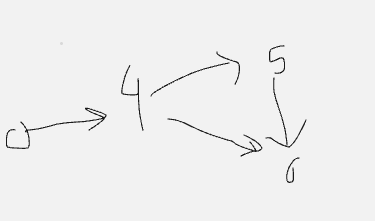
            if i not in visited:

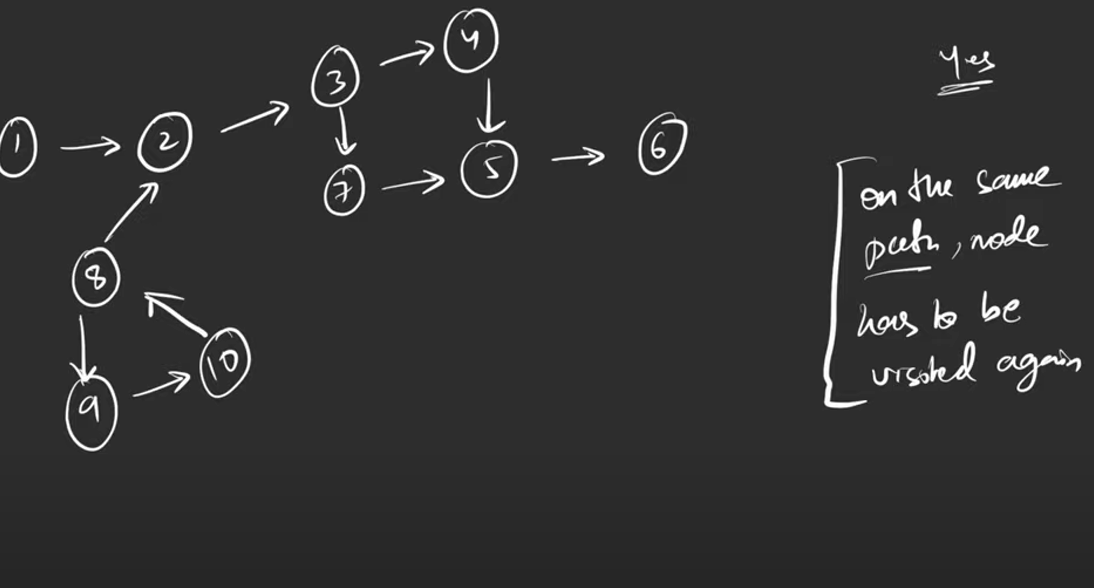
                if self.dfs(i,-1,visited,adj)==True:

                    return True

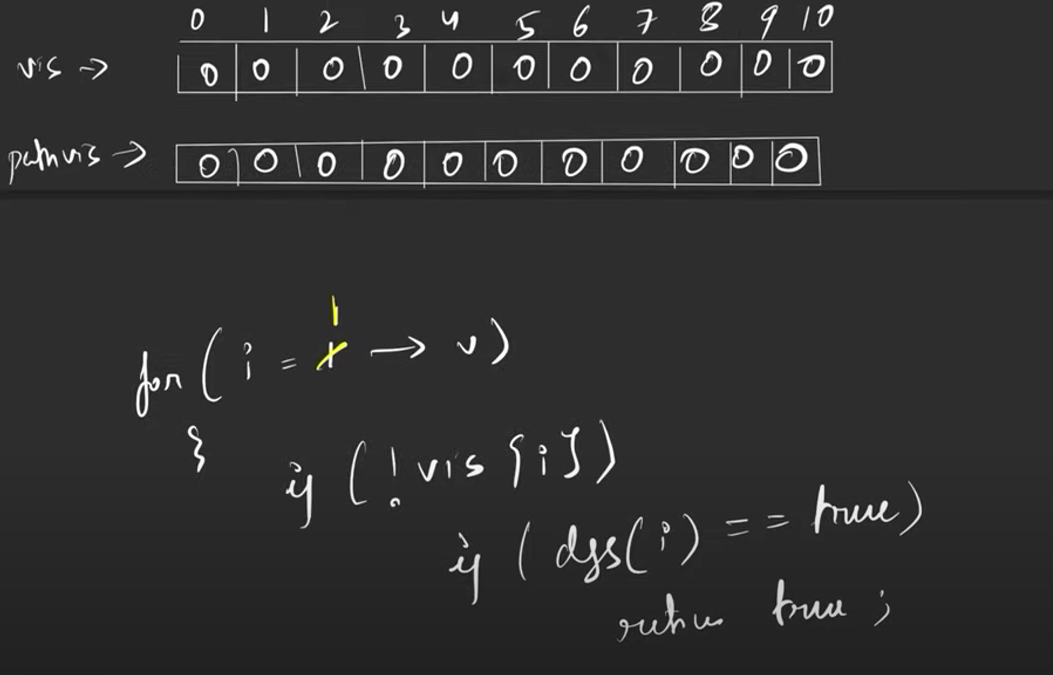
        return False

**For directed graph can’t use this: as for below it gives true though it is not cycle**

****

****

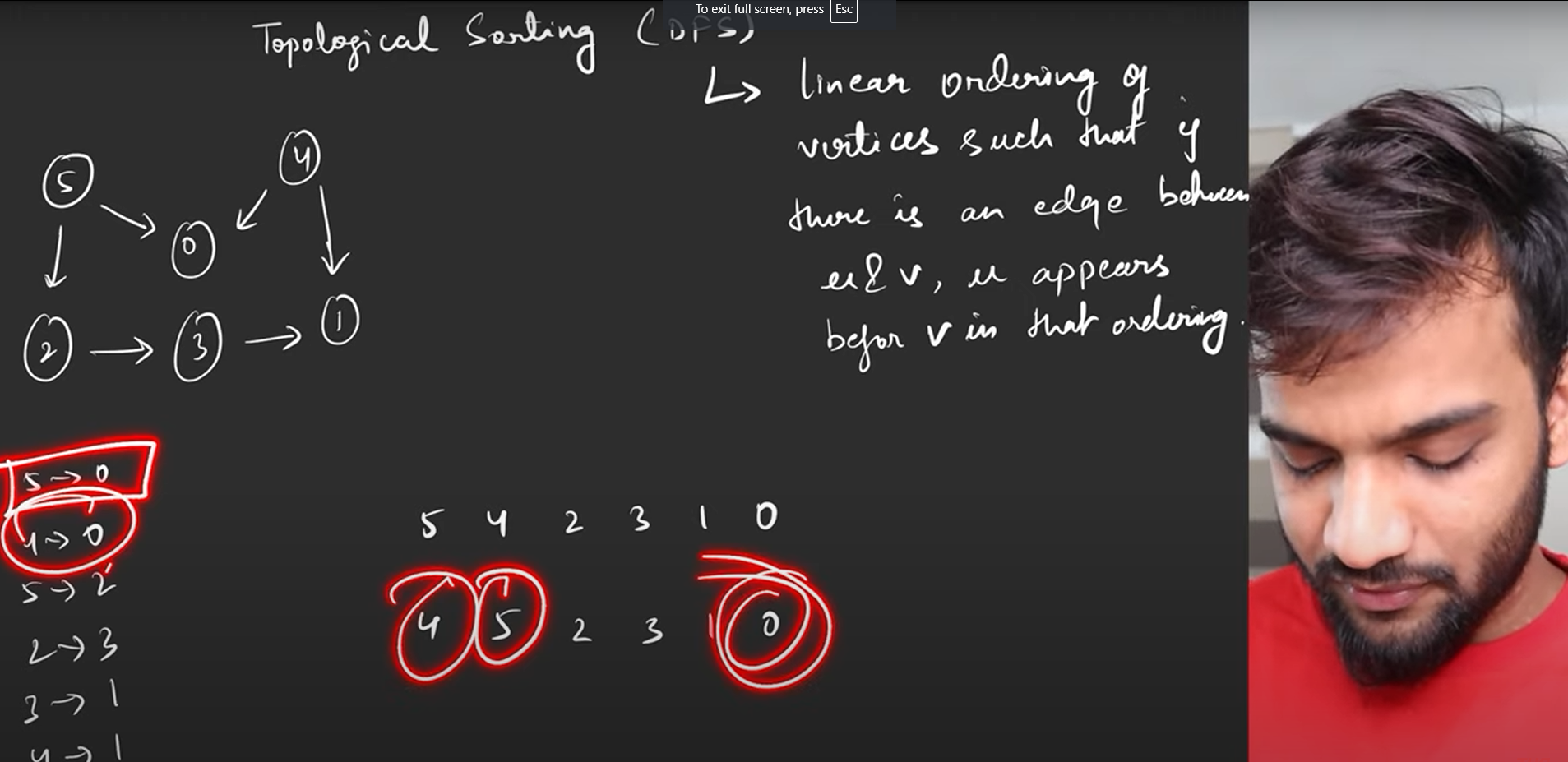
**FOR directed, node should visit in same path again,**



VIDEO

<https://www.youtube.com/watch?v=9twcmtQj4DU&list=PLgUwDviBIf0oE3gA41TKO2H5bHpPd7fzn&index=19>

**TOPOLOGICAL Sort**

****

**Dijkstra Algorithm for shortest path**

import heapq

def dijkstra(graph, start):

    # Initialize distances with infinity for all vertices except the start

    distances = {vertex: float('infinity') for vertex in graph}

    distances[start] = 0

    # Initialize priority queue with the start vertex and distance 0

    priority\_queue = [(0, start)]

    while priority\_queue:

        # Pop the vertex with the smallest distance from the priority queue

        current\_distance, current\_vertex = heapq.heappop(priority\_queue)

        # If the popped distance is greater than the current known distance, skip it

        if current\_distance > distances[current\_vertex]:

            continue

        # Check and update distances for all neighboring vertices

        for neighbor, weight in graph[current\_vertex].items():

            distance = current\_distance + weight

            if distance < distances[neighbor]:

                distances[neighbor] = distance

                heapq.heappush(priority\_queue, (distance, neighbor))

    return distances

# Example Usage

graph = {

    'A': {'B': 1, 'C': 4},

    'B': {'A': 1, 'D': 2, 'E': 5},

    'C': {'A': 4, 'F': 3},

    'D': {'B': 2},

    'E': {'B': 5, 'F': 1},

    'F': {'C': 3, 'E': 1}

}

print(dijkstra(graph, 'A'))

It does not work with negative edge weight. As gets stuck in infinite loop.

### Explanation:

1. **Initialization**: Distances to all vertices are set to infinity except the start vertex, which is set to 0.
2. **Priority Queue**: A priority queue (min-heap) is initialized with the start vertex.
3. **Processing**: The vertex with the smallest known distance is extracted from the priority queue.
4. **Distance Update**: For each neighbor of the current vertex, the distance is updated if a shorter path is found. The neighbor and the updated distance are then pushed onto the priority queue.
5. **Check and Skip**: If the current distance of the extracted vertex is greater than the known shortest distance, it is skipped.

### Implement Dijkstra Algorithm.

Link: <https://www.geeksforgeeks.org/problems/implementing-dijkstra-set-1-adjacency-matrix/1>

import heapq

class Solution:

    #Function to find the shortest distance of all the vertices

    #from the source vertex S.

    def dijkstra(self, V, adj, S):

        #code here

        distances = [float('inf') for i in range(V)]

        distances[S]=0

        priority\_queue = [(0,S)]

        while priority\_queue:

            curr\_distance, curr\_node = heapq.heappop(priority\_queue)

            if curr\_distance > distances[curr\_node]:

                continue

            #adj list here = (node,weight)

            for neighbour in adj[curr\_node]:

                n\_node, n\_weight = neighbour[0], neighbour[1]

                n\_distance = curr\_distance + n\_weight

                if n\_distance < distances[n\_node]:

                    distances[n\_node] = n\_distance

                    heapq.heappush(priority\_queue,(n\_distance, n\_node))

        return distances

# **Questions: Basics of Graph + Graph Traversal**

### 1. Create adjacency list

Link: <https://www.geeksforgeeks.org/problems/print-adjacency-list-1587115620/1>

### 2. Depth First Search

Link: <https://www.geeksforgeeks.org/problems/depth-first-traversal-for-a-graph/1>

### 3. Breadth First Search

Link: <https://www.geeksforgeeks.org/problems/print-adjacency-list-1587115620/1>

### 4. Find if path exists

Link: <https://leetcode.com/problems/find-if-path-exists-in-graph/>

### 5. All paths from source to target

Link: <https://leetcode.com/problems/all-paths-from-source-to-target/>

### 6. Minimum vertices to reach all nodes

Link: <https://leetcode.com/problems/minimum-number-of-vertices-to-reach-all-nodes/>

# LEVEL 2: **Medium**

# LEVEL 3: **Difficult**

# **SOLUTIONS:**

## **Solutions: Basics of Graph + Graph Traversal**

1. Create Adjacency List

class Solution:

    def printGraph(self, V : int, edges : List[List[int]]) -> List[List[int]]:  #V= number of vertex

        # code here

        adj = [[] for \_ in range(V)]

        for i in edges:

            adj[i[0]].append(i[1])

            adj[i[1]].append(i[0])

        return adj

1. Depth First Search

class Solution:

    def dfsOfGraph(self, V, adj):

        def helper(v,visited):

            global ans

            visited.add(v)

            ans.append(v)

            for neighbour in adj[v]:

                if neighbour not in visited:

                    helper(neighbour,visited)

        global ans

        ans =[]

        visited=set()

        helper(0,visited)  #starting for vertex 0

        return ans

1. Breadth First Search

from collections import deque

class Solution:

    def bfsOfGraph(self, V: int, adj: List[List[int]]) -> List[int]:

        queue  = deque([0]) # start from vertex 0

        visited=set([0])

        ans=[]

        while queue:

            temp = queue.popleft()

            ans.append(temp)

            for neighbour in adj[temp]:

                if neighbour not in visited:

                    visited.add(neighbour)

                    queue.append(neighbour)

        return ans

1. Find if path exists

class Solution:

    def validPath(self, n: int, edges: List[List[int]], source: int, destination: int) -> bool:

        #start dfs from source, and if find desitination return true, else false

        #create adj list from edges

        adj=[[] for \_ in range(n)]

        for edge in edges:

            adj[edge[0]].append(edge[1])

            adj[edge[1]].append(edge[0])

        def dfs(node,visited):

            if node==destination:

                return True

            visited.add(node)

            for neighbour in adj[node]:

                if neighbour not in visited:

    #don't directly do, return dfs(), as it won't run further dfs if it get False ans too.

    #Just return only if got True ans, else run dfs till end

                    if dfs(neighbour,visited):

                        return True

            return False

        visited=set()

        return dfs(source, visited)

1. All paths from source to target

This is Directed Acyclic graph, so no need to check for visited nodes as in DAG we won’t stuck in loop, and also we need all paths

class Solution:

    def allPathsSourceTarget(self, graph: List[List[int]]) -> List[List[int]]:

        #For paths prefer dfs

        #get n-1th node

        n=0

        for nodes in graph:

            n = max([n]+nodes)

        def dfs(node, psf):

            global ans

            if node == n:

                ans.append(psf)

            for neighbour in graph[node]:

                dfs(neighbour, psf + [neighbour])

        global ans

        ans=[]

        dfs(0,[0])

        return ans

1. Minimum vertices to reach all nodes

class Solution:

    def findSmallestSetOfVertices(self, n: int, edges: List[List[int]]) -> List[int]:

        #All nodes which has incoming edge can be reached by some paths

        #But nodes which don't have incoming edge can't be reached by any, and counts in ans

        in\_edge = [0]\*n

        for edge in edges:

            in\_edge[edge[1]]=1

        ans=[]

        for i in range(n):

            if in\_edge[i]==0:

                ans.append(i)

        return ans