**Graph**

Table of Contents

[**Theory: Basics of Graph** 3](#_Toc203883950)

[1.1 What is a Graph? 3](#_Toc203883951)

[1.2 Basic Terminology 3](#_Toc203883952)

[1.3 Basic Types of Graphs 3](#_Toc203883953)

[1.4 Graph representation 5](#_Toc203883954)

[**Theory: Graph Traversal** 6](#_Toc203883955)

[2.1. Depth-First Search (DFS) 6](#_Toc203883956)

[2.2. Breadth-First Search (BFS) 7](#_Toc203883957)

[2.3. Bipartite Graph 8](#_Toc203883958)

[**Theory: Shortest Path** 11](#_Toc203883959)

[3.1. Dijkstra Algorithm for shortest path 11](#_Toc203883960)

[**Basics of Graph + Graph Traversal** 15](#_Toc203883961)

[Level 1: Amateur 15](#_Toc203883962)

[1. Create adjacency list 15](#_Toc203883963)

[2. Depth First Search 15](#_Toc203883964)

[3. Breadth First Search 15](#_Toc203883965)

[4. Count the paths 15](#_Toc203883966)

[5. Find if path exists 15](#_Toc203883967)

[6. All paths from source to target 15](#_Toc203883968)

[7. Minimum vertices to reach all nodes 15](#_Toc203883969)

[8. Number of Provinces 15](#_Toc203883970)

[Level 2: Pro 16](#_Toc203883971)

[1. Number of Islands 16](#_Toc203883972)

[2. Max Area of island 16](#_Toc203883973)

[3. Number of enclaves 16](#_Toc203883974)

[4. Count the number of complete components 16](#_Toc203883975)

[5. Detect cycle in undirected graph 16](#_Toc203883976)

[6. Bipartite Graph 16](#_Toc203883977)

[7. Rotten Oranges 16](#_Toc203883978)

[Level 3: Expert 17](#_Toc203883979)

[1. ….. 17](#_Toc203883980)

[**Solutions: Basics of Graph + Graph Traversal** 18](#_Toc203883981)

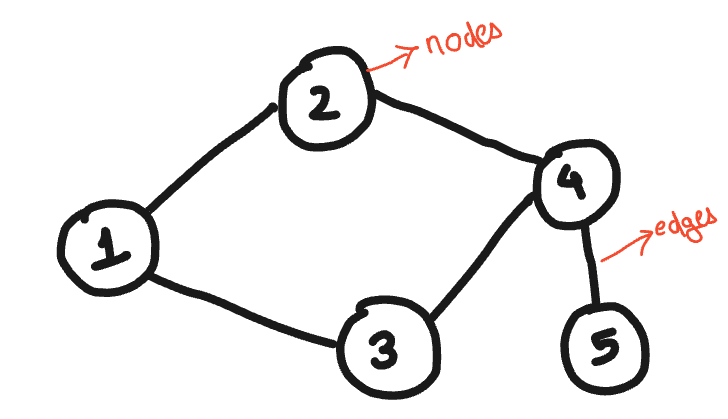
[Level 1: Solutions 18](#_Toc203883982)

[Level 2: Pro Level Solutions 22](#_Toc203883983)

# **Theory: Basics of Graph**

## 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.

**PUT better image** 

## 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.

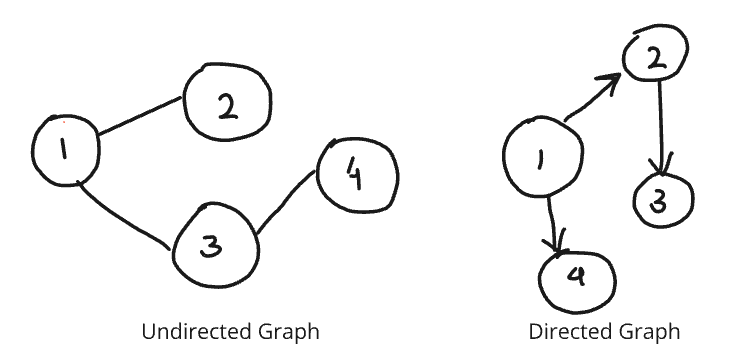
## 1.3 Basic Types of Graphs

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).

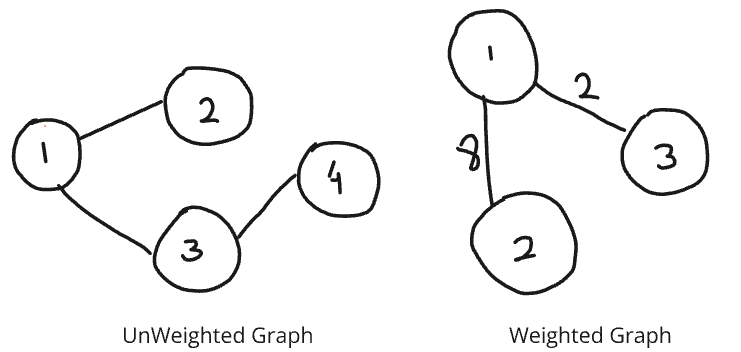


1. **Weighted Graph**:

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

1. **Unweighted Graph**:

A graph where edges do not have weights.

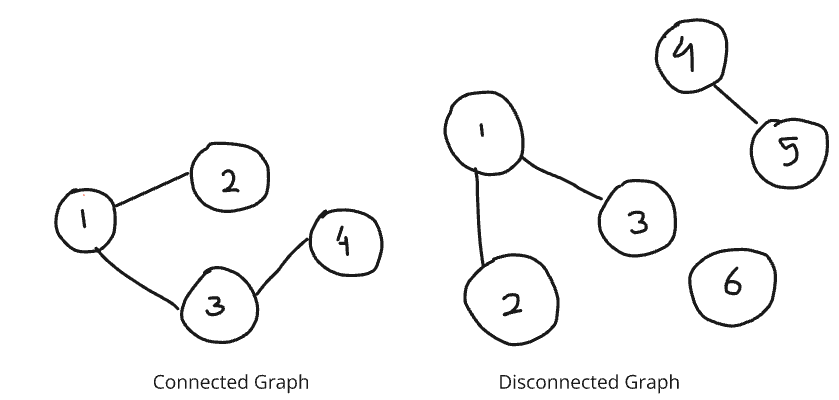


1. **Connected Graph**:

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 neighbours.

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],

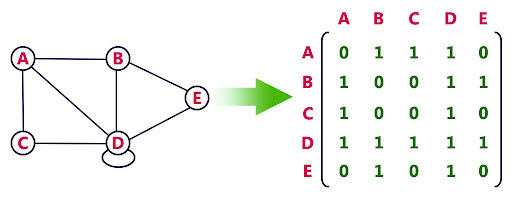
    [1, 0, 0, 0, 0, 1],

    [0, 1, 0, 0, 0, 0],

    [0, 1, 0, 0, 0, 1],

    [0, 0, 1, 0, 1, 0]

])



Adjacency matrix

# **Theory: Graph Traversal**

## 2.1. 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'],

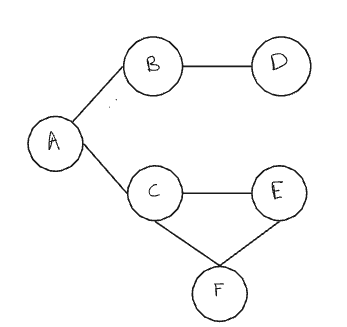
    'D': ['B'],

    'E': ['B', 'F'],

    'F': ['C', 'E']

}

dfs(graph, 'A')

****

OUTPUT: A B D E F C

## 2.2. 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) # Add node here itself in visited

                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

**Time complexity:-**

For both DFS and BFS:

* Each node is visited exactly once.
* Each edge is explored at most twice.

Thus, the time complexity is: **O(V + E)**

## 2.3. Bipartite Graph

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']

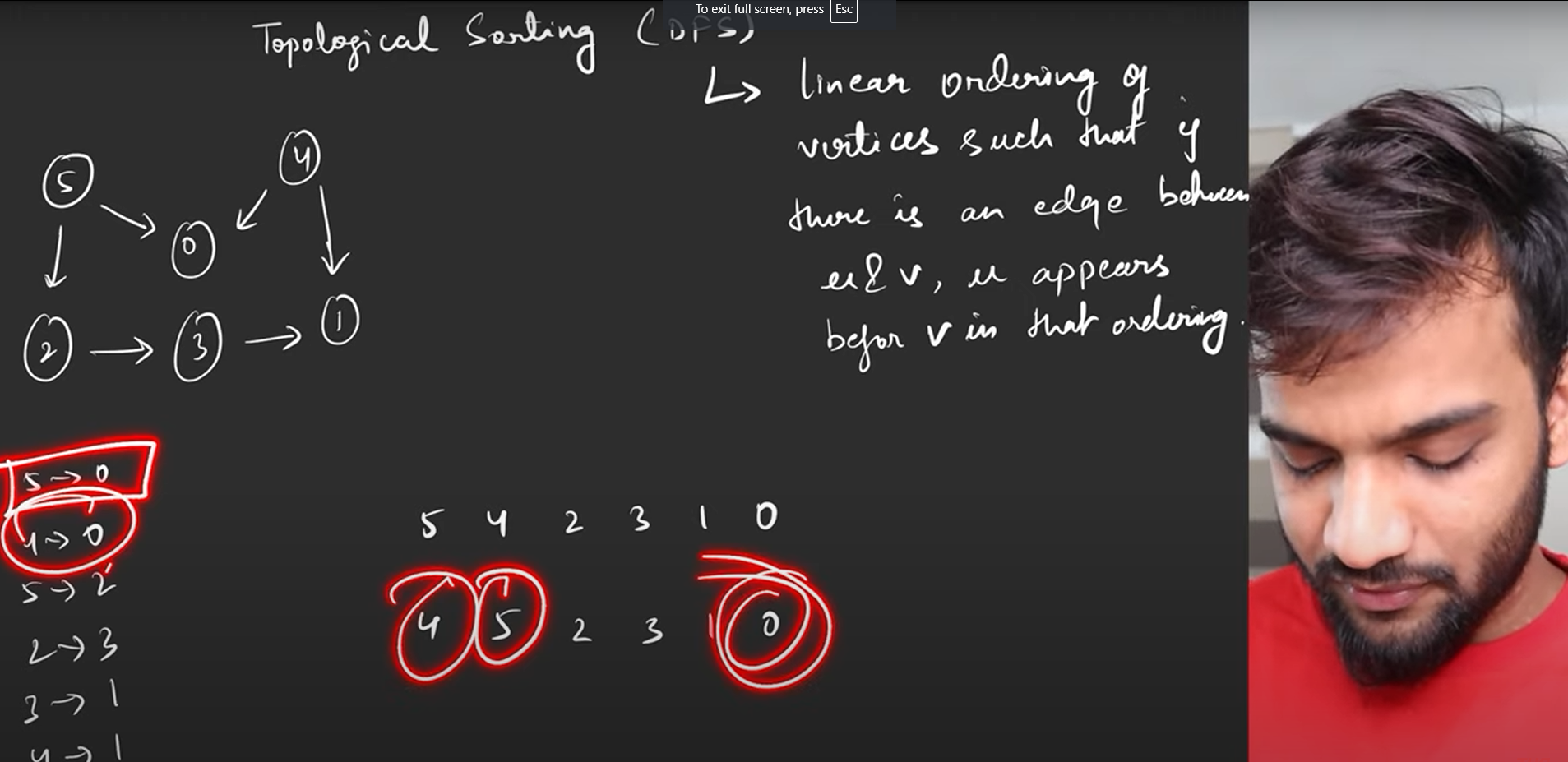
}

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

VIDEO

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

**TOPOLOGICAL Sort**

****

# **Theory: Shortest Path**

## 3.1. 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 neighbour of the current vertex, the distance is updated if a shorter path is found. The neighbour 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

# **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/bfs-traversal-of-graph/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. Number of Provinces

Link: <https://leetcode.com/problems/number-of-provinces/>

# **Union Find**

### 1. Number of Islands

Link: <https://leetcode.com/problems/number-of-islands/>

### 2. Max Area of island

Link: <https://leetcode.com/problems/max-area-of-island/description/>

### 3. Number of enclaves

Link: <https://leetcode.com/problems/number-of-enclaves/>

### 4. Count the number of complete components

Link: <https://leetcode.com/problems/count-the-number-of-complete-components/description/>

### 5. Detect cycle in undirected graph

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

### 6. Detect cycle in directed graph

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

### 7. Rotten Oranges

Link: <https://leetcode.com/problems/rotting-oranges/description/>

### 8. Bipartite Graph

Link: <https://leetcode.com/problems/is-graph-bipartite/>

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

1. Create adjacency list

class Solution:  *#V: number of nodes*

    def printGraph(*self*, *V* : int, *edges* : List[List[int]]) -> List[List[int]]:

        adj=[[] *for* i *in* range(*V*)]

*for* edge *in* *edges*:

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

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

*return* adj

1. Depth First Search

class Solution:

    def dfs(self, adj):

        def helper(node,visited):

            ans.append(node)

            visited.add(node)

            for neighbour in adj[node]:

                if neighbour not in visited:

                    helper(neighbour,visited)

        ans=[]

        helper(0,set())

        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)

Approach 2: using global variable

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*

        adj=[[] *for* \_ *in* range(*n*)]

*for* edge *in* *edges*:

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

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

        def dfs(*node*,*dest*,*visited*,*adj*):

*if* *node* == *dest*:

                global ans

                ans=True

*return*

*visited*.add(*node*)

*for* neighbour *in* *adj*[*node*]:

*if* neighbour not in *visited*:

                    dfs(neighbour,*dest*,*visited*,*adj*)

        global ans

        ans=False

        visited = set()

        dfs(*source*,*destination*,visited,adj)

*return* ans

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

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

    def dfs(*node*,*target*,*path*):

*if* *node*==*target*:

*self*.ans.append(*path*)

*else*:

*for* neighbour *in* *graph*[*node*]:

                dfs(neighbour,*target*,*path*+[neighbour])

*self*.ans=[]

    dfs(0,len(*graph*)-1,[0])

*return* *self*.ans

1. Number of Provinces

These is equivalent to counting number of components in disconnected graph. We run loop from 1 to n, and for non-visited node we call the dfs function. DFS mark all nodes connected to it as visited.

Thus whenever we calling dfs, it means that this node was not visited by any previous node, thus it is a new province (component).

class Solution:

    def dfs(self,node, visited, adj):

        visited[node]=1

        for neighbour in adj[node]:

            if visited[neighbour]==0:

                self.dfs(neighbour,visited,adj)

    def findCircleNum(self, isConnected: List[List[int]]) -> int:

        num\_of\_prov = 0

        n = len(isConnected)

        visited = [0]\*(n)

        #adj matrix to adj list

        adj = [[] for \_ in range(n)]

        for i in range(n):

            for j in range(n):

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

                    adj[i].append(j)

        for i in range(n):

            if visited[i]==0:

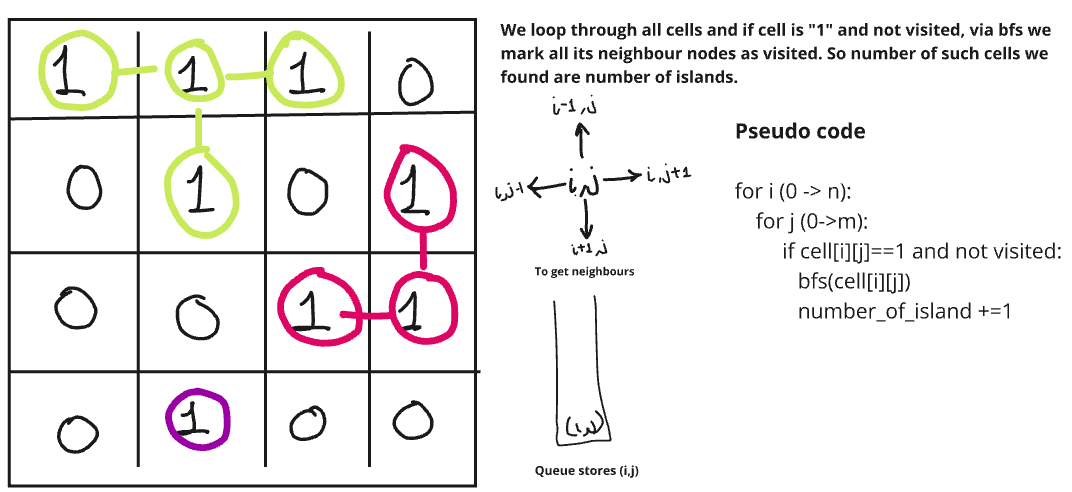
                self.dfs(i, visited,adj)

                num\_of\_prov+=1

        return num\_of\_prov

# **Solutions: Union Find**

1. Number of Islands



Using DFS

class Solution:

    def dfs(*self*,*i*,*j*,*grid*,*visited*,*r*,*c*):

        dr,dc = [1,0,-1,0], [0,1,0,-1]

*visited*[*i*][*j*]=1

*for* d *in* range(4):

            x = *i* + dr[d]

            y = *j* + dc[d]

*if* x>=0 and x<*r* and y>=0 and y<*c* and *grid*[x][y]=='1' and *visited*[x][y]==0:

*self*.dfs(x,y,*grid*,*visited*,*r*,*c*)

    def numIslands(*self*, *grid*: List[List[str]]) -> int:

        r = len(*grid*)

        c = len(*grid*[0])

        num\_of\_island = 0

        visited = [[0]\*c *for* \_ *in* range(r)]

*for* i *in* range(r):

*for* j *in* range(c):

*if* *grid*[i][j]=='1' and visited[i][j]==0:

                    num\_of\_island += 1

*self*.dfs(i,j,*grid*,visited,r,c)

*return* num\_of\_island

Using BFS

from collections import deque

class Solution:

    def numIslands(self, grid: List[List[str]]) -> int:

        r = len(grid)

        c = len(grid[0])

        num\_of\_island = 0

        visited = [[0]\*c for \_ in range(r)]

        for i in range(r):

            for j in range(c):

                #run bfs on non-visited node

                if visited[i][j]==0 and grid[i][j]=='1': #means new island

                    num\_of\_island += 1

                    queue = deque([[i,j]])

                    visited[i][j]=1

                    dr = [-1,0,1,0]

                    dc = [0,1,0,-1]

                    while(queue):

                        node = queue.popleft()

                        x,y = node[0],node[1]

                        for d in range(4):    #checking all neighbours

                            new\_r, new\_c = x+dr[d], y+dc[d]

                            if(new\_r>=0 and new\_r<r and new\_c>=0 and new\_c<c

                                and grid[new\_r][new\_c]=='1' and visited[new\_r][new\_c]==0):

                                queue.append([new\_r,new\_c])

                                visited[new\_r][new\_c]=1

        return num\_of\_island

1. Max Area of Islands

Here used different method to run dfs, without using visited array and using inner function, so that we won’t be required to pass multiple arguments.

class Solution:

    def maxAreaOfIsland(*self*, *grid*: List[List[int]]) -> int:

        m = len(*grid*)

        n = len(*grid*[0])

*#use of inner function*

*#returns number of times dfs was run*

        def dfs(*i*, *j*): *# will return area*

*if* *i*<0 or *i*>=m or *j*<0 or *j*>=n or *grid*[*i*][*j*]!=1:

*return* 0

*else*:

*grid*[*i*][*j*] = 0

*return* 1+dfs(*i*+1, *j*)+dfs(*i*, *j*-1)+dfs(*i*-1, *j*)+dfs(*i*, *j*+1)

*#iterate all cells and run dfs if grid[i][j]==1*

*#inplace of using visited, can make grid[i][j]=0*

        max\_area = 0

*for* i *in* range(m):

*for* j *in* range(n):

*if* *grid*[i][j] == 1:

                    max\_area = max(max\_area, dfs(i, j))

*return* max\_area

1. Number of enclaves

Approach: we just need count of all nodes which are covered from water in all sides. So we will just remove all land which are connected to edges by running DFS for all cells in edges. Whatever land is left will be our answers.

Note: we need to get all lands left after doing that, so whatever are water or are visited by DFS on edge cell will be marked as 0. So just count all left over 1’s are our answer.

class Solution:

    def numEnclaves(*self*, *grid*: List[List[int]]) -> int:

        n = len(*grid*)

        m = len(*grid*[0])

*#use of inner function to run dfs*

        def dfs(*i*, *j*): *# will return area*

*if* *i*<0 or *i*>=n or *j*<0 or *j*>=m or *grid*[*i*][*j*]!=1:

*return*

*grid*[*i*][*j*] = 0

            dfs(*i*+1, *j*)

            dfs(*i*, *j*-1)

            dfs(*i*-1, *j*)

            dfs(*i*, *j*+1)

*#iterate in boundaries and run dfs*

*#whichever won't get visited but are 1 will be our ans*

*for* i *in* range(n):

*if* *grid*[i][0]==1:

                dfs(i,0)

*if* *grid*[i][m-1]==1:

                dfs(i,m-1)

*for* j *in* range(m):

*if* *grid*[0][j]==1:

                dfs(0,j)

*if* *grid*[n-1][j]==1:

                dfs(n-1,j)

        land=0

*for* i *in* range(n):

*for* j *in* range(m):

                land+=*grid*[i][j]

*return* land

1. Count the number of complete components

Let say number of nodes in component is n, then in bfs each node of component is present in adj list of n-1 nodes. So we will check at end if number of nodes which visited n-1 times in given bfs equals count of total nodes in component, if yes this component satisfy the condition and get included in final ans.

from collections import deque

class Solution:

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

        #convert to adj list

        adj = [[] for \_ in range(n)]

        for edge in edges:

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

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

        visited = [0]\*n

        num\_of\_comp = 0

        #finding number of disconnected components by bfs

        for i in range(n):

            if visited[i]==0:  #means one new component

                node\_count=0          #check how many nodes are in component

                queue = deque([i])

                vis\_count=[0]\*n       #check how many times each node in component is called

                visited[i]=1

                while(queue):

                    node = queue.popleft()

                    node\_count+=1

                    for neighbour in adj[node]:

                        vis\_count[neighbour]+=1

                        if visited[neighbour]==0:

                            queue.append(neighbour)

                            visited[neighbour]=1

                #check if component is complete

                #For complete component, all nodes are visited n-1 times

                count=0

                for i in vis\_count:

                    if i==node\_count-1:

                        count+=1

                if count==node\_count or node\_count==1:

                    num\_of\_comp+=1

        return num\_of\_comp

Using DFS:

Approach: To get count of nodes that were traversed in given dfs, use a extra set. Now check every node in this set should be connect to all others nodes in given set, so the length of adjacency list for given node should be equal to len(set)-1 .

class Solution:

    def dfs(*self*,*node*,*adj*,*visited*,*now\_visited*):

        visited[node]=1

        now\_visited.add(node)

*for* neighbour *in* adj[node]:

*if* visited[neighbour]==0:

                self.dfs(neighbour, adj,visited,now\_visited)

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

        count = 0

        visited = [0]\*n

*#create adj list*

        adj = [[] *for* \_ *in* range(n)]

*for* edge *in* edges:

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

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

*for* i *in* range(n):

*if* visited[i]==0:

                now\_visited=set()  *#keep track of how many nodes got iterated in given dfs*

                self.dfs(i,adj,visited,now\_visited)

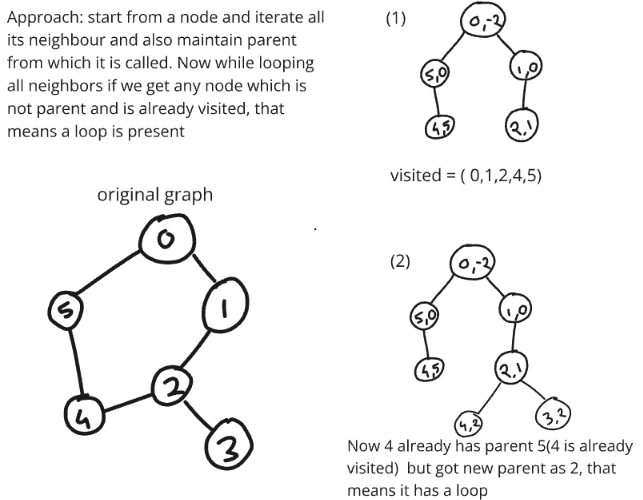
                required = len(now\_visited)-1  *#all n nodes are visited in dfs,all must have n-1 neighbours*

*if* all(len(adj[node])==required *for* node *in* now\_visited):

                    count+=1

*return* count

1. Detect cycle in undirected graph



**Note: Here Graph can have unconnected components**

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 for false

            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):

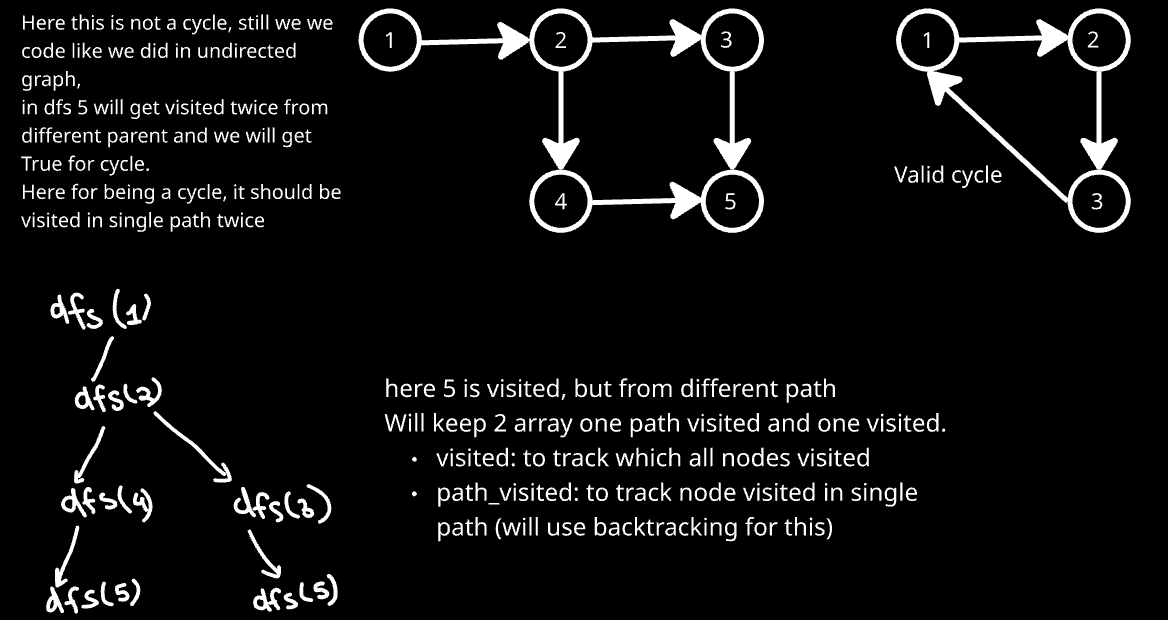
            if i not in visited:

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

                    return True

        return False

1. Detect cycle in directed graph



class Solution:

    #helper code

    def dfs(self,node, adj, visited, path\_visited):

        visited[node]=1

        path\_visited[node]=1

        for neighbour in adj[node]:

            if visited[neighbour]==1 and path\_visited[neighbour]==1:

                return True

            elif visited[neighbour]==0:

                if self.dfs(neighbour,adj,visited,path\_visited):

                    return True

        path\_visited[node]=0

        return False

    def isCycle(self, V, edges):

        adj= [[] for \_ in range(V)]

        for edge in edges:

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

        visited = [0]\*V

        path\_visited = [0]\*V

        for i in range(V):

            if visited[i]==0:

                if self.dfs(i,adj,visited,path\_visited):

                    return True

        return False

1. Rotten Oranges

from collections import deque

class Solution:

    def orangesRotting(self, grid: List[List[int]]) -> int:

        dr, dc = [1,0,-1,0], [0,1,0,-1]

        n, m = len(grid), len(grid[0])

        visited = grid.copy()

        queue = deque()

        time = 0

        max\_time = time

        #if fresh oranges count becomes 0, all got rotten

        cnt = 0       #keeps count of fresh orange

        #at time 0, will start bfs from all rotten orange

        for i in range(n):

            for j in range(m):

                if visited[i][j]==2: #push rotten oranges in queue

                    queue.append([i,j,time])

                if grid[i][j]==1:    #get count of fresh oranges

                    cnt+=1

        while queue:

            node = queue.popleft()

            i, j, time = node[0], node[1], node[2]

            max\_time = max(max\_time, time)

            for d in range(4):

                di, dj = i+dr[d], j+dc[d]

                ntime = time+1

                if di>=0 and dj>=0 and di<n and dj<m and visited[di][dj]==1:

                    visited[di][dj] = 2

                    queue.append([di,dj,ntime])

                    cnt-=1

        if cnt!=0:  #all fresh oranges did not get rotten

            return -1

        return max\_time

1. Bipartite Graph

*#using BFS*

*from* collections *import* deque

class Solution:

    def isBipartite(*self*, *graph*: List[List[int]]) -> bool:

        n=len(*graph*)

        color = {}

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

*for* i *in* range(n):

*if* i not in color:

                color[i]=0

                queue = deque([i])

*while* queue:

                    node = queue.popleft()

*for* neighbour *in* *graph*[node]:

*if* neighbour not in color:

                            color[neighbour] = 1-color[node]

                            queue.append(neighbour)

*else*:

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

*return* False

*return* True

*#Using DFS*

class Solution:

    def isBipartite(*self*, *graph*: List[List[int]]) -> bool:

        def dfs(*node*):

*for* neighbour *in* *graph*[*node*]:

                pc = color[*node*]      *#color of parent node*

*if* color[neighbour]==-1:

                    color[neighbour] = 1-pc

*if* not dfs(neighbour):    *#if any of dfs return false, final ans is false*

*return* False

*elif* color[neighbour]!=-1 and color[neighbour]==pc:

*return* False

*return* True

        n = len(*graph*)

        color=[-1]\*n     *#will color 0 & 1 , -1 means not colored*

*#since graph are disconnected*

*for* i *in* range(n):

*if* color[i]==-1:

                color[i]=0      *#start with parent node as 0 color*

*if* not dfs(i):  *#if any dfs give false, graph is not bipartite*

*return* False

*return* True