**https://leetcode.com/discuss/interview-question/3365918/Graphs-or-All-Algorithms-or-Theroy-And-Implementation**

**Please feel free to share any feedback and error**

**Please upvote**

**Graph Traversal**

* DFS

void dfs(int node,vector<int> &vis,vector<int> adj[],vector<int> &ans){

vis[node]=1;

ans.push\_back(node);

for(auto it:adj[node]){

if(!vis[it]) dfs(it,vis,adj,ans);

}

}

vector<int> dfsOfGraph(int V, vector<int> adj[]) {

vector<int>ans;

vector<int>vis(V, 0);

for(int i = 0 ; i < V ; i++){

if(!vis[i])

dfs(i, vis, adj, ans);

}

return ans;

}

* BFS (Will print all level in single line)

/\*

0

/ \

- -

/ \

1 2

\*/

void bfs(int src,vector<int> &v1,vector<int> adj[],int v){

queue<int> q;

vector<int> vis(v+1, 0);

q.push(0);

vis[0] = 1;

while(!q.empty()){

int node=q.front();

q.pop();

v1.push\_back(node);

for(auto x:adj[node]){

if(!vis[x]){

q.push(x);

vis[x] = 1;

}

}

}

}

vector<int> bfsOfGraph(int v, vector<int> adj[]) {

vector<int> v1;

bfs(0,v1,adj,v);

return v1;

}

* BFS (For finding Height or Level / Print level wise)  
  Note : We use three loop to remove a parameter from the queue in most of the cases

/\*

0

/ \

--------

/ \

1 2

\*/

void bfs(int src,vector<int> &v1,vector<int> adj[],int v){

queue<int> q;

vector<int> vis(v+1,0);

q.push(0);

vis[0] = 1;

int level = 0;

while(!q.empty()){

int size = q.size();

level++;

for(int i = 0 ; i < size ; i++){

int node = q.front();

q.pop();

v1.push\_back(node);

for(auto x:adj[node]){

if(!vis[x]){

q.push(x);

vis[x]=1;

}

}

}

}

cout << " -- "<<level << " -- ";

}

vector<int> bfsOfGraph(int v, vector<int> adj[]) {

vector<int> v1;

bfs(0,v1,adj,v);

return v1;

}

**Cycle Detection - Undirected Graph**

* Using DFS

/\*

-> If the child node is visited

-> It must be a parent if there have to be no cycle

-> If child comes out to be a non parent as well as vis then it must be forming a cycle

-> If the child node is not visited

-> Do a deeper DFS search and return true if a cycle is found deep inside the graph

\*/

bool isCycleUtil(int i, int par, vector<int> adj[], vector<int>&vis){

vis[i] = true;

for(auto child : adj[i]){

if(!vis[child]){

if(isCycleUtil(child, i, adj, vis)){

return true;

}

}else if(vis[child]){

if(child == par)continue;

else if(child != par)return true;

}

}

return false;

}

bool isCycle(int V, vector<int> adj[]) {

vector<int>vis(V + 1, 0);

for(int i = 0 ; i < V ; i++){

if(!vis[i]){

if(isCycleUtil(i, -1, adj, vis)){

return true;

}

}

}

return false;

}

};

* Using BFS

bool isCycleUtilBFS(int nn, int n, vector<int>adj[]){

vector<int> vis(n + 1, 0);

queue<pair<int, int>>q;

q.push({nn, -1});

vis[nn] = 1;

while(!q.empty()){

int size = q.size();

// Process the level

for(int i = 0 ; i < size ; i++){

int node = q.front().first;

int parent = q.front().second;

q.pop();

// Process the child

for(auto child : adj[node]){

if(child == parent)continue;

if(vis[child])return true;

q.push({child, node});

vis[child] = 1;

}

}

}

return false;

}

bool isCycle(int V, vector<int> adj[]) {

for(int i = 0 ; i < V ; i++){

if(isCycleUtilBFS(i, V, adj)){

return true;

}

}

return false;

}

* BFS Alternative

bool bfs(int start, vector<int> adj[], vector<int> &vis, queue<pair<int,int>> &q){

q.push({start, -1});

vis[start] = 1;

while(!q.empty()){

int node = q.front().first;

int parent = q.front().second;

q.pop();

for(auto it : adj[node]){

if(vis[it] == 0){

q.push({it, node});

vis[it] = 1;

}

else if(it != parent){

return true;

}

}

}

return false;

}

public:

bool isCycle(int V, vector<int> adj[]) {

//using bfs

vector<int> vis(V, 0);

queue<pair<int,int>> q;

for(int i=0; i<V; i++){

if(vis[i] == 0){

if(bfs(i, adj, vis, q)){

return true;

}

}

}

return false;

}

**Cycle Detection - Directed Graph**

* Using DFS

/\*

-> We maintain a path visited for detecting cycle in a directed graph

-> Which helps to determine whether the node which is visited already is in current path or not

-> Which ensures a cycle

\*/

bool dfs(int node, vector<int> adj[], vector<int> &vis, vector<int> &pathVis){

vis[node] = 1;

pathVis[node] = 1;

for(auto child : adj[node]){

if(!vis[child]){

if((dfs(child, adj, vis, pathVis)))return true;

}else if(vis[child] && pathVis[child]){

return true;

}

}

pathVis[node] = 0;

// If there is not a cycle return false

return false;

}

bool isCyclic(int V, vector<int> adj[]) {

vector<int> vis(V, 0), pathVis (V, 0);

for(int i = 0 ; i < V ; i++){

if(!vis[i]){

if(dfs(i, adj, vis, pathVis))return true;

}

}

return false;

}

};

* Using BFS (Kahns Algo)

/\*

-> Cycle detection using BFS

-> We can detect a cycle in directed acyclic graph using Kahns Algorithm which ensures if there are two nodes in

a graph u and v u must comes before v.

-> Topo sort vector size is equal to number of vertices in DAG

-> But in case of cyclic graph some of the vertices can not be inserted into the vector and the size becomes lesser

-> Which indicates cycle is present

\*/

bool isCyclic(int V, vector<int> adj[]) {

vector<int> indegree(V, 0);

int countTopo = 0;

for(int i = 0 ; i < V ; i++){

for(auto it : adj[i]){

indegree[it]++;

}

}

queue<int> q;

for(int i = 0 ; i < V ; i++)

if(indegree[i] == 0)

q.push(i);

while(!q.empty()){

int node = q.front();

q.pop();

countTopo++;

for(auto child : adj[node]){

indegree[child]--;

if(indegree[child] == 0)q.push(child);

}

}

if(countTopo != V)return 1;

return 0;

}

**Flood Fill Algorithm**

void dfs(vector<vector<int>>& image,int sr,int sc,int newColor,int x){

if(sr < 0 || sr >= image.size() || sc < 0 || sc >= image[0].size() ||

image[sr][sc] != x || image[sr][sc] == newColor)return;

image[sr][sc]=newColor;

dfs(image,sr-1,sc,newColor,x);

dfs(image,sr,sc+1,newColor,x);

dfs(image,sr+1,sc,newColor,x);

dfs(image,sr,sc-1,newColor,x);

}

vector<vector<int>> floodFill(vector<vector<int>>& image, int sr, int sc, int newColor) {

int x=image[sr][sc];

dfs(image,sr,sc,newColor,x);

return image;

}

**Topological Sorting**

* Using DFS

/\*

-> The main idea behind topological sorting is to create a linear ordering of the vertices such that for any

directed edge from vertex u to vertex v, u comes before v in the ordering. This ordering is known as a topological order.

\*/

void dfs(int node, vector<int> adj[], vector<int> &vis, stack<int> &st){

vis[node] = 1;

for(auto child : adj[node]){

if(!vis[child])dfs(child, adj, vis, st);

}

// If evry guy is touched in the dfs call while returning push the nodes into the stack

st.push(node);

}

vector<int> topoSort(int V, vector<int> adj[]) {

stack<int> st;

vector<int> vis (V, 0), ans;

for(int i = 0 ; i < V ; i++){

if(!vis[i])dfs(i, adj, vis, st);

}

while(!st.empty()){

ans.push\_back(st.top());

st.pop();

}

return ans;

}

* Using BFS (Kahns Algorithm)

vector<int> kahnsAlgo(int V, vector<int> adj[]){

vector<int> topo, indegree(V, 0);

for(int i = 0 ; i < V ; i++){

for(auto it : adj[i]){

indegree[it]++;

}

}

queue<int> q;

for(int i = 0 ; i < V ; i++)

if(indegree[i] == 0)

q.push(i);

while(!q.empty()){

int node = q.front();

q.pop();

topo.push\_back(node);

for(auto child : adj[node]){

indegree[child]--;

if(indegree[child] == 0)q.push(child);

}

}

return topo;

}

**Dijkstra's Algorithm**

**Some important questions on Dijkstra**

**Why do we use Dijkstra Algorithm**

**- Dijkstra's algorithm is one of the most popular algorithms for solving many single-source shortest path problems having non-negative edge weight in the graphs i.e., it is to find the shortest distance between two vertices on a graph**.

**How can we implement Dijkstra**

**- Pseudocode**

/\*

-> Define a priority queue min heap

-> Distance vector storing INT\_MAX initially for every nodes

-> push (source weight, source node)

-> set source node dist as 0

-> While(!pq.empty()){

get top node

for(expore its child){

try to relax the nodes if there is a shorter path exists

}

More detailed

1. Create a priority queue Q and initialize it with the starting vertex s with distance 0.

2. Create an array dist to store the minimum distance to each vertex from s, initialized to infinity except for s, which is set to 0.

3. Create an array prev to store the previous vertex on the shortest path to each vertex from s, initialized to null.

4. While Q is not empty:

a. Extract the vertex u with the smallest distance from Q.

b. For each neighbor v of u:

i. Calculate the distance d from s to v through u as dist[u] + w(u,v), where w(u,v) is the weight of the edge (u,v).

ii. If d is smaller than the current distance dist[v], update dist[v] to d and set prev[v] to u.

iii. Add v to Q if it is not already visited.

5. Return the dist and prev arrays to reconstruct the shortest path from s to any vertex in the graph.

\*/

**Discuss the use cases of implementing Dijkstra using Priority Queue, Set and Queue**

**1. Queue : When we use queue we explore all paths which is not a greedy approach which results in slower runtime and exploration of unnecessary paths.  
2. Priority Queue : Priority queue is best Data Structure for implementing the Dijkstra Alogirthm in priority queue we have to do some extra extra exploration than set.  
3. Set : Set erases the useless node which in case helps to reduce some extra iterations but removing element from set cost O(Logn) so overall there is not much difference in runtime when we use set or priority queue.**

**Pros and Cons of Dijkstra**

**Pros : Accuracy, efficiency and easy implementation**  
**Cons : Negative edge weights (Wrong Answers) or negative cycles (TLE)**

**Time and space complexity of Dijkstra**

**Time Complexity : O(ElogV)  
Space Complexity : O(V)**

**How to overcome the problems we face in Dijkstra**

**- Negative edge weights: Dijkstra's algorithm assumes that all edge weights are positive. If there are negative edge weights in the graph, the algorithm may not produce the correct result. One way to overcome this problem is to use another algorithm such as the Bellman-Ford algorithm, which can handle negative edge weights.**

* **Multiple shortest paths: If there are multiple shortest paths between two nodes, Dijkstra's algorithm will only find one of them. To find all possible shortest paths, you can use another algorithm such as the A star algorithm or the Floyd-Warshall algorithm. We can also use multisource BFS.**

**Difference between BFS and Dijkstra Algorithm**

**- BFS: path with the smallest number of edges (assuming the same weight for every edge or no weight).**\*  
**- Dijkstra: path with the smallest weight along the path**

**Dijkstra Using Priority Queue**

vector <int> dijkstra(int V, vector<vector<int>> adj[], int S){

vector<int>dist(V, INT\_MAX);

priority\_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>> pq;

dist[S] = 0;

pq.push({0, S});

dist[S] = 0;

while(!pq.empty()){

int node = pq.top().second;

pq.pop();

for(auto child : adj[node]){

int adjNode = child[0];

int weight = child[1];

if(dist[adjNode] > weight + dist[node]){

dist[adjNode] = dist[node] + weight;

pq.push({dist[adjNode], adjNode});

}

}

}

return dist;

}

**Dijkstra Using Set**

vector <int> dijkstra(int V, vector<vector<int>> adj[], int S){

set<pair<int, int>> st; // {dist, node}

st.insert({0, S});

vector<int> dist(V, INT\_MAX);

dist[S] = 0;

while(st.size()) {

auto it = \*(st.begin());

int distance = it.first;

int node = it.second;

st.erase(it);

for(auto i : adj[node]) {

int child = i[0];

int wt = i[1];

if(distance + wt < dist[child]) {

if(dist[child] != INT\_MAX) {

st.erase({dist[child], child}); // to save some iterations but deleting too cost logn

}

dist[child] = distance + wt;

st.insert({dist[child], child});

}

}

}

return dist;

}

**Dijkstra Using Queue**

vector <int> dijkstra(int V, vector<vector<int>> adj[], int S){

vector<int>dist(V, INT\_MAX);

queue<pair<int, int>> pq;

dist[S] = 0;

pq.push({0, S});

dist[S] = 0;

while(!pq.empty()){

int node = pq.top().second;

pq.pop();

for(auto child : adj[node]){

int adjNode = child[0];

int weight = child[1];

if(dist[adjNode] > weight + dist[node]){

dist[adjNode] = dist[node] + weight;

pq.push({dist[adjNode], adjNode});

}

}

}

return dist;

}

**Bellman Ford Algorithm**

**- The Bellman-Ford algorithm is a shortest path algorithm that finds the shortest path between a source vertex and all other vertices in a weighted directed graph. It works by repeatedly relaxing all the edges in the graph until it has found the shortest path to all vertices.**

**Here's how the Bellman-Ford algorithm works:**

* Initialize the distance to the source vertex as 0, and the distance to all other vertices as infinity.
* Relax all the edges in the graph |V|-1 times, where |V| is the number of vertices in the graph. During each relaxation, the algorithm considers all the edges in the graph and updates the distance to the destination vertex if the current distance is greater than the distance to the source vertex plus the weight of the edge.
* After |V|-1 relaxations, check for negative weight cycles. If any edge can further be reduced means there is a negative cycle in the graph present.
* The time complexity of the Bellman-Ford algorithm is O(V \* E)

vector<int> bellman\_ford(int n, vector<vector<int>>& edges, int S) {

vector<int> dist(n, 1e8);

dist[S] = 0;

for(int i = 1 ; i <= n - 1 ; i++){

for(auto it : edges){

int u = it[0];

int v = it[1];

int w = it[2];

// Reachable and can be relaxed

if(w + dist[u] < dist[v]){

dist[v] = w + dist[u];

}

}

}

for(auto it : edges){

int u = it[0];

int v = it[1];

int w = it[2];

// Reachable and can be relaxed hence negative cycle must be present

if(w + dist[u] < dist[v]){

return {-1};

}

}

return dist;

}

**Floyd-Warshall**

Floyd-Warshall algorithm is a dynamic programming algorithm used to find the shortest path between all pairs of vertices in a weighted graph. It works by considering all intermediate vertices in each path and computing the shortest path between each pair of vertices. The algorithm has a time complexity of O(V^3), where V is the number of vertices in the graph.

/\*

-> Multisource shortest path

-> This algorithm also finds out the negative cycle

-> For(i = 0 -> n)if(cost[i][i] < 0)negative cycle exists

\*/

void shortest\_distance(vector<vector<int>>&matrix){

int n = matrix.size();

for(int i = 0 ; i < n ; i++){

for(int j = 0 ; j < n ; j++){

if(matrix[i][j] == -1)matrix[i][j] = 1e9;

}

}

for(int k = 0 ; k < n ; k++){

for(int i = 0 ; i < n ; i++){

for(int j = 0 ; j < n ; j++){

matrix[i][j] = min(matrix[i][j], matrix[i][k] + matrix[k][j]);

}

}

}

for(int i = 0 ; i < n ; i++){

for(int j = 0 ; j < n ; j++){

if(matrix[i][j] == 1e9)matrix[i][j] = -1;

}

};

}

};

**MST**

/\*

-> Spanning tree is a tree which have n node and n - 1 edges all tightly connected

-> MST is a tree with n node n - 1 edges having minimum overall weight

-> Prims and Kruskal are two algorithm to find the MST

-> Prims and Kruskal are both Greedy

-> TC : ELOGV for both

-> SC : O(E+V)

-> Prims uses pq and greedy approach

-> Kruskal uses sorting based on weight and DSU

\*/

**Disjoint Set Union (DSU)**

/\*

-> Disjoint set is used to find whether two elements belong to the same group or not in constant time

-> Algorithm

-> We have to maintain two vector rank and parent to store the current rank of a given node

and parent of node

-> For checking if two nodes belong to the same parent their ultimate parent must be same

-> Union Operation

-> If the parent comes out to be same then return, that means they belong to same group

-> Else if the rank of left is greater than right than left becomes the parent of right which

ensures no increase in the rank of the former

-> If the rank is same it does not matter who becomes the parent of whom just increase the rank

by one and assign the parent

-> Find operation uses simple recursion to find the parent and also used for path comperession

\*/

class DisjointSet{

vector<int> rank, parent, size;

public:

// Constructor

DisjointSet(int n){

rank.resize(n + 1, 0);

parent.resize(n + 1, 0);

size.resize(n + 1, 0);

// Ensures 1 Based indexing graph also

for(int i = 0 ; i <= n ; i++){

parent[i] = i;

size[i] = 1;

}

};

// Find Operation with path compression

int find(int node){

if(parent[node] == node)return node;

else return parent[node] = find(parent[node]);

};

// Union operation by rank

void unionbyrank(int x, int y){

int ult\_parent\_x = find(x);

int ult\_parent\_y = find(y);

// They already belongs to the same group

if(ult\_parent\_x == ult\_parent\_y)return;

// They belong to different groups, based on ranking the union operation will be broken in three pieces

if(rank[ult\_parent\_x] > rank[ult\_parent\_y]){

parent[ult\_parent\_y] = ult\_parent\_x;

}else if(rank[ult\_parent\_x] < rank[ult\_parent\_y]){

parent[ult\_parent\_x] = ult\_parent\_y;

}else{

parent[ult\_parent\_x] = ult\_parent\_y;

rank[ult\_parent\_y] += 1;

}

}

// Union operation by size

void unionbysize(int x, int y){

int ult\_parent\_x = find(x);

int ult\_parent\_y = find(y);

// They already belongs to the same group

if(ult\_parent\_x == ult\_parent\_y)return;

// They belong to different groups, based on ranking the union operation will be broken in three pieces

// x component bigger

if(size[ult\_parent\_x] > size[ult\_parent\_y]){

parent[ult\_parent\_y] = ult\_parent\_x;

size[ult\_parent\_x] += size[ult\_parent\_y];

// y component bigger or both same

}else{

parent[ult\_parent\_x] = ult\_parent\_y;

size[ult\_parent\_y] += size[ult\_parent\_x];

}

}

};

**Prims Algorithm**

int spanningTree(int V, vector<vector> adj[])

{

// Prims

priority\_queue<pair<int, pair<int, int>>,

vector<pair<int, pair<int, int>>>, greater<pair<int, pair<int, int>>>>

pq;

vector<vector<int>> MST;

vector<int> vis(V, 0);

// {wt, node}

// pq.push({0, 0, - 1});

pq.push({0, {0, 0}});

int totalSum = 0;

while (!pq.empty()) {

auto it = pq.top();

int distance = it.first;

int from = it.second.first;

int to = it.second.second;

pq.pop();

// Checking if the node already exists in the MST or not, vis[node] means adding to the MST

if (vis[to])continue;

totalSum += distance;

vis[to] = 1;

// Adding edge to MST

if (from != to){

MST.push\_back({from, to});

// MST.push\_back({to, from}); // in case of undirected graph

}

// Adding its adjacent nodes

for (auto it : adj[to])

{

int childW = it[1];

int childN = it[0];

// If the child node is not already visited add it to the pq to process on it

if (!vis[childN])

{

pq.push({childW, {to, childN}});

}

}

}

// print MST edges if needed

// for(auto v : MST)

// cout << v[0] << " --> " << v[1] << endl;

return totalSum;

}

**Kruskals Algorithm**

int spanningTree(int V, vector<vector<int>> adj[]){

// Kruskal

vector<vector<int>> MSTEdges;

vector<pair <int, pair<int, int>>> edges;

for(int i = 0 ; i < V ; i++){

for(auto it : adj[i]){

int node = i;

int adjNode = it[0];

int weight = it[1];

edges.push\_back({weight, {node, adjNode}});

}

}

DisjointSet ds(V);

int mstWeight = 0;

sort(edges.begin(), edges.end());

for(auto it : edges){

int weight = it.first;

int node = it.second.first;

int adjNode = it.second.second;

// If they belong to different component

if(ds.find(node) != ds.find(adjNode)){

mstWeight += weight;

ds.unionbyrank(node, adjNode);

MSTEdges.push\_back({node, adjNode});

}

}

// Printing MST

for(auto i : MST){

for(auto j : i){

cout << j << " ";

}

cout << endl;

}

return mstWeight;

}

**Kosarajus' Algorithm (Strongly Connected Components) SCC**

**Theory**

/\*

-> Strongly connected components are the one in which any node can be reached from any node

-> Kosaraju's algorithm is used to find the strongly connected components

-> In this algorithm we have to do two DFS

-> First DFS to store the nodes in order of their insertion

-> Now using the stack we take nodes one by one and find the strongly connected component in first DFS

-> Then we reverse each edge

-> Reversing the edges ensures the components are not reachable at once when a dfs is called

-> Time Complexity : O(V + E)

-> Space Complexity : O(V)

\*/

**Implementation**

void dfs(int node, vector<int> adj[], vector<int> &vis, stack <int> &time\_of\_ins){

vis[node] = 1;

for(auto child : adj[node]){

if(!vis[child]){

dfs(child, adj, vis, time\_of\_ins);

}

}

time\_of\_ins.push(node);

}

void dfs2(int node, vector<int> adj[], vector<int> &vis, vector<int> &ds){

vis[node] = 1;

for(auto child : adj[node]){

if(!vis[child]){

dfs2(child, adj, vis, ds);

}

}

ds.push\_back(node);

}

int kosaraju(int V, vector<vector<int>>& adj){

// Declarations

int no\_of\_scc = 0;

vector<vector<int>> scc;

vector<int> ds;

vector<int> vis (V, 0), vis2(V, 0);

vector<int> adj1 [V];

vector<int> adj2[V];

stack <int> time\_of\_ins;

// Creating the adjacency list

for(int i = 0 ; i < V ; i++){

for(auto child : adj[i]){

adj1[i].push\_back(child);

}

}

// DFS 1

for(int i = 0 ; i < V; i++){

if(!vis[i]){

dfs(i, adj1, vis, time\_of\_ins);

}

}

// Reversing the edges

for(int i = 0 ; i < V ; i++){

for(auto child : adj[i]){

adj2[child].push\_back(i);

}

}

// DFS 2

while(!time\_of\_ins.empty()){

int node = time\_of\_ins.top();

time\_of\_ins.pop();

if(!vis2[node]){

no\_of\_scc++;

dfs2(node, adj2, vis2, ds);

scc.push\_back(ds);

ds.clear();

}

}

// Printing the strongly connected elements

// for(auto i : scc){

// for(auto j : i){

// cout << j << " ";

// }

// cout << endl;

// }

return no\_of\_scc;

}

**Tarjans Algorithm (Bridge Detection)**

class Solution {

public:

/\*

-> There is no alternate path to the adj node with lesser time means the edge between the node and its adj is

a bridge, we are taking par to update low only on the basis adj node or node itself,

-> par == child continue means dont take parent for updating low

-> lowest[adj] > time\_of\_insertion[node]

-> Time Complexity : O(V + E)

-> Space Complexity : O(V)

\*/

int timer = 1;

void tarjan(int node, vector<int>adj[], int par, vector<int>& lowest,

vector<int>& time\_of\_insertion, vector<vector<int>>& ans, vector<int>&vis){

// Intitially time of insertion and lowest time of insertion will be same

vis[node] = 1;

time\_of\_insertion[node] = lowest[node] = timer;

timer++;

for(auto it : adj[node]){

// This condition for updation of low value only on the basis of adj node not on the parent

if(it == par)continue;

if(!vis[it]){

tarjan(it, adj, node, lowest, time\_of\_insertion, ans, vis);

// While returning from the dfs of a node updating the lowest

lowest[node] = min(lowest[node], lowest[it]);

// If there is no other path to the node then the edge is a bridge / criticall connectoin

if(lowest[it] > time\_of\_insertion[node])ans.push\_back({node, it});

}else lowest[node] = min(lowest[node], lowest[it]);

}

}

vector<vector<int>> criticalConnections(int n, vector<vector<int>>& connections) {

vector<int>adj[n];

vector<int>vis(n, 0), lowest(n, 0), time\_of\_insertion(n, 0);

vector<vector<int>>ans;

// Creating a graph

for(int i = 0 ; i < connections.size() ; i++){

adj[connections[i][0]].push\_back(connections[i][1]);

adj[connections[i][1]].push\_back(connections[i][0]);

}

// Tarjans Alogirthm

tarjan(0, adj, -1, lowest, time\_of\_insertion, ans, vis);

return ans;

}

};

**More Readable**

class Solution {

public:

int timer = 1; vector<vector<int>>ans; vector<int>vis, lowest\_tin, time\_of\_insertion;

void tarjan(int node, vector<int>adj[], int par){

vis[node] = 1; time\_of\_insertion[node] = lowest\_tin[node] = timer++;

for(auto it : adj[node]){

if(it == par)continue;

if(!vis[it]){

tarjan(it, adj, node);

if(lowest\_tin[it] > time\_of\_insertion[node])ans.push\_back({node, it});

}

lowest\_tin[node] = min(lowest\_tin[node], lowest\_tin[it]);

}

}

vector<vector<int>> criticalConnections(int n, vector<vector<int>>& connections) {

vector<int>adj[n];

vis.resize(n, 0), lowest\_tin.resize(n, 0), time\_of\_insertion.resize(n, 0);

for(int i = 0 ; i < connections.size() ; i++){

adj[connections[i][0]].push\_back(connections[i][1]);

adj[connections[i][1]].push\_back(connections[i][0]);

}

tarjan(0, adj, -1);

return ans;

}

};

**Length of Cycle in Graph (Directed)**

* Brute DFS

/\*

-> Maximizing the cycle length

\*/

class Solution {

public:

bool dfs(int node, vector<int> adj[], vector<int> &vis, vector<int> &pathVis, vector<int> &ds){

vis[node] = 1;

pathVis[node] = 1;

ds.push\_back(node);

for(auto child : adj[node]){

if(!vis[child]){

if(dfs(child, adj, vis, pathVis, ds))return true;

}else if(vis[child] && pathVis[child]){

ds.push\_back(child);

return true;

}

}

ds.pop\_back();

pathVis[node] = 0;

return false;

}

int longestCycle(vector<int>& edges) {

int n = edges.size();

vector<vector<int>> ans;

vector<int> vis (n, 0);

vector<int> adj[n];

for(int i = 0 ; i < n ; i++){

if(edges[i] == -1)continue;

adj[i].push\_back(edges[i]);

}

for(int i = 0 ; i < n ; i++){

vector<int> ds;

if(!vis[i]){

vector<int> pathVis (n, 0);

dfs(i, adj, vis, pathVis, ds);

if(ds.size() > 2)

ans.push\_back(ds);

cout << ds.size();

}

}

vector<vector<int>> ans\_trimmed;

for(int i = 0 ; i < ans.size() ; i++){

vector<int> temp = ans[i];

vector<int> temp2;

int x = -1;

if(temp.size() > 0)x = temp[temp.size() - 1];

for(int j = temp.size() - 1 ; j >= 0 ; j--){

if(temp[j] == x && j != temp.size() - 1)break;

temp2.push\_back(temp[j]);

}

ans\_trimmed.push\_back(temp2);

}

// Printing the cycle

// for(auto i : ans\_trimmed){

// for(auto j : i){

// cout << j << " ";

// }

// cout << endl;

// }

int maxi = -1;

for(auto i : ans\_trimmed){

int y = i.size();

if(maxi < y)maxi = y;

}

return maxi;

}

};

* DFS (Optimal with bool type return)

bool dfs(int node, vector<int> adj[], vector<int> &vis, vector<int> &pathVis, vector<int>&count, int &maxCycleSize){

/\*

-> For proper backtracking of pathVis vector we have to backtrack when we return true manually

\*/

vis[node] = 1;

pathVis[node] = 1;

for(auto child : adj[node]){

if(!vis[child]){

count[child] = count[node] + 1;

if(dfs(child, adj, vis, pathVis, count, maxCycleSize )){

pathVis[node] = 0;

return true;

}

}else if(vis[child] && pathVis[child]){

maxCycleSize = max(maxCycleSize, count[node] - count[child] + 1);

pathVis[node] = 0;

return true;

}

}

pathVis[node] = 0;

return false;

}

* DFS (Optimal with void - Explore all paths without backtracking - More convenient)

void dfs(int node, vector<int> adj[], vector<int> &vis, vector<int> &pathVis, vector<int>&count, int &maxCycleSize){

vis[node] = 1;

pathVis[node] = 1;

for(auto child : adj[node]){

if(!vis[child]){

count[child] = count[node] + 1;

dfs(child, adj, vis, pathVis, count, maxCycleSize);

}else if(vis[child] && pathVis[child]){

maxCycleSize = max(maxCycleSize, count[node] - count[child] + 1);

}

}

pathVis[node] = 0;

}

**Length of Cycle in Graph (Undirected)**

* Shortest Cycle

class Solution {

public:

void dfs(int node, vector<int> adj[], int par, vector<int> &vis, vector<int>&count, int &minCycle){

vis[node] = 1;

for(auto child : adj[node]){

if(!vis[child]){

count[child] = count[node] + 1;

dfs(child, adj, node, vis, count, minCycle);

}else if(vis[child] && child != par){

minCycle = min(minCycle, abs(count[node] - count[child]) + 1);

}

}

}

int findShortestCycle(int n, vector<vector<int>>& edges) {

vector<int> adj[n];

for(auto i : edges){

adj[i[0]].push\_back(i[1]);

adj[i[1]].push\_back(i[0]);

}

int minCycle = 1e4;

for(int i = 0 ; i < n ; i++){

/\*

-> Refreshing visited to explore new paths with new visited

8

[[0,1],[1,2],[2,3],[3,4],[4,5],[0,7],[0,6],[5,7],[5,6]]

\*/

vector<int> vis (n, 0);

if(!vis[i]){

vector<int> count (n, 1);

dfs(i, adj, -1, vis, count, minCycle);

}

}

if(minCycle == 1e4)return -1;

return minCycle;

}

};

* Longest Cycle

class Solution {

public:

void dfs(int node, vector<int> adj[], int par, vector<int> &vis, vector<int>&count, int &maxCycle){

vis[node] = 1;

for(auto child : adj[node]){

if(!vis[child]){

count[child] = count[node] + 1;

dfs(child, adj, node, vis, count, maxCycle);

}else if(vis[child] && child != par){

maxCycle = max(maxCycle, abs(count[node] - count[child]) + 1);

}

}

}

int findShortestCycle(int n, vector<vector<int>>& edges) {

vector<int> adj[n];

for(auto i : edges){

adj[i[0]].push\_back(i[1]);

adj[i[1]].push\_back(i[0]);

}

int maxCycle = -1;

for(int i = 0 ; i < n ; i++){

/\*

-> Refreshing visited to explore new paths with new visited

8

[[0,1],[1,2],[2,3],[3,4],[4,5],[0,7],[0,6],[5,7],[5,6]]

\*/

vector<int> vis (n, 0);

if(!vis[i]){

vector<int> count (n, 1);

dfs(i, adj, -1, vis, count, maxCycle);

}

}

if(maxCycle == -1)return -1;

return maxCycle;

}

};

**Euler Circuit and Path**

/\*

-> An Eulerian circuit in an undirected graph is a path that visits every edge of the graph exactly once

and ends at the starting vertex. In other words, it is a cycle that covers all edges of the graph.

-> Eulerian path is a path that visits every edge of the graph exactly once, but does not necessarily

end at the starting vertex. In other words, it is a path that covers all edges of the graph, but may start

and end at different vertices.

-> Undirected

Euelian Circuit - Degree even for all nodes

Eulerian Pathn - n - 2 node degree even and two node degree odd

\*/

int isEularCircuit(int V, vector<int>adj[]){

vector<int> deg(V, 0);

int x = adj[0].size();

int cntOdd = 0;

for(int i = 0 ; i < V ; i++){

deg[i] = adj[i].size();

if(deg[i] % 2 != 0)cntOdd++;

}

// Euler Circuit

if(cntOdd == 0)return 2;

// Euler Path

if(cntOdd == 2)return 1;

// Neither a Euler Circuit or Path

return 0;

}

**Hamiltonian Path**

/\*

-> Hamiltonian Path - Each vertex is travelled once

-> Hamiltonian Circuit - Each vertex is travelled once and there is an edge between last and first vertex

\*/

public:

bool dfs(int node, vector<int>adj[], vector<int> &vis, int n, int count){

if(count == n - 1)return true;

vis[node] = 1;

count++;

for(auto child : adj[node]){

if(!vis[child])

if(dfs(child, adj, vis, n, count))return true;

}

count--;

vis[node] = 0;

return false;

}

bool check(int N,int M,vector<vector<int>> Edges)

{

vector<int> adj[N + 1];

for(auto i : Edges){

adj[i[0]].push\_back(i[1]);

adj[i[1]].push\_back(i[0]);

}

vector<int> vis(N + 1, 0);

for(int i = 1 ; i <= N ; i++){

if(!vis[i]){

if(dfs(i, adj, vis, N, 0))return true;

}

}

return false;

}