**L1. Introduction to Graphs (Types of Graphs):**

Two Components:

1. Node/Vertex

2. Edge

Types of Graphs:

1. Undirected Graphs

2. Directed Graphs

3. Weighted Graphs (Undirected and Directed)

Degree of a vertex: Number of edges that are incoming to or outgoing from the node

In an undirected graph, Total degree of all the nodes= 2\*Edges

Indegree: Number of edges that are incoming to the node

Outdegree: Number of edges that are outgoing from the node

Path: In an undirected graph, a path is a sequence of nodes such that none of the nodes are repeating or visited twice in that path

DAG: Directed Cyclic Graph

**L2. Graph Representation in C++:**

n and m. Next m lines contain 2 integers.

1. Adjacency Matrix:

int main(){

int n, m;

cin>>n>>m;

// declare the adjacent matrix

// int adj[n+1][n+1];

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

// take edges as input

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

int u, v;

cin>>u>>v;

adj[u][v]= 1;

adj[v][u]= 1;

}

return 0;

}

// SC= O(n\*n)

2. Adjacency List:

int main(){

int n, m;

cin>>n>>m;

vector<vector<int>> adj(n+1);

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

int u, v;

cin>>u>>v;

adj[u].push\_back(v);

adj[v].push\_back(u);

}

return 0;

}

// SC= O(n+(2\*E))

**L4. Connected Components in Graph:**

Graph may consist of different components

Disconnected Graph : Component1 + Component2 + Component3 make up a single disconnected graph

Graph can be either single component or more than one components

Even a single node can be called as a component

Write code for multiple components whether it is DFS or BFS

Take visited array and initialise as 0

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

if(!visited[i]){

// Write code for either BFS or DFS

}

}

How to find the number of components?

Count the number of times your code goes inside the if statement

**L5. Breadth-First Search (BFS):**

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

vector<int> bfs;

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

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

if(!vis[i]){

queue<int> q;

q.push(i);

vis[i]= 1;

while(!q.empty()){

int node= q.front();

q.pop();

bfs.push\_back(node);

for(auto it: adj[node]){

if(!vis[it]){

q.push(it);

vis[it]= 1;

}

}

}

}

}

return bfs;

}

// TC= O(n+E) (n is time taken for visiting n nodes and E is for travelling throught adjacent nodes overall)

// SC= O(n+E)+O(n)+O(n) (Adjacent List+Visited Array+Queue)

**L6. Depth-First Search (DFS):**

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

storeDfs.push\_back(node);

vis[node]= 1;

for(auto it: adj[node]){

if(!vis[it]){

dfs(it, vis, adj, storeDfs);

}

}

return;

}

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

vector<int> storeDfs;

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

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

if(!vis[i]){

dfs(i, vis, adj, storeDfs);

}

}

return storeDfs;

}

// TC= O(n+E) (n is time taken for visiting n nodes, and E is for travelling through adjacent nodes overall

// SC= O(n+E)+O(n)+O(n) (Adj List+Vis Array+Auxillary Space)

**L7. Cycle Detection in Undirected Graph using BFS:**

bool checkForCycle(int s, int V, vector<int> adj[], vector<int> &visited){

// Create a queue for BFS

queue<pair<int, int>> q;

visited[s]= true;

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

while(!q.empty()){

int node= q.front().first;

int par= q.front().second;

q.pop();

for(auto it: adj[node]){

if(!visited[it]){

visited[it]= true;

q.push({it, node});

}

else if(it!=par){

return true;

}

}

}

return false;

}

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

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

for(int i=1;i<=V;i++)

if(!vis[i])

if(checkForCycle(i, V, adj, vis))

return true;

return false;

}

// TC= O(n+E), SC= O(n+E)+O(n)+O(n)

**L8. Cycle Detection in Undirected Graph using DFS:**

bool checkForCycle(int node, int parent, vector<int> &vis, vector<int> adj[]){

vis[node]= 1;

for(auto it: adj[node]){

if(!vis[it]){

if(checkForCycle(it, node, vis, adj))

return true;

}

else if(it!=parent){

return true;

}

}

return false;

}

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

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

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

if(!vis[i])

if(checkForCycle(i, -1, vis, adj)

return true;

}

return false;

}

// TC= O(n+E), SC= O(n+E)+O(n)+O(n)

**L9. Bipartite Graph (BFS):**

Bipartite Graph:A graph that can be coloured using exactly 2 colours such that no 2 adjacent nodes have the same colour

If the graph has an odd length cycle, then it is not bipartite

If the graph doesn't have any odd length cycle, it is bipartite

bool bipartiteBfs(int src, vector<int> adj[], vector<int> &color){

queue<int> q;

q.push(src);

color[src]= 1;

while(!q.empty()){

int node= q.front();

q.pop();

for(auto it: adj[node]){

if(color[it]==-1){

color[it]= 1-color[node];

q.push(it);

}

else if(color[it]==color[node]){

return false;

}

}

}

return true;

}

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

vector<int> color(n, -1);

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

if(colour[i]==-1)

if(!bipartiteBfs(i, adj, color))

return false;

return true;

}

int main(){

int n, m;

cin>>n>>m;

vector<int> adj[n];

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

int u, v;

cin>>u>>>v;

adj[u].push\_back(v);

adj[v].push\_back(u);

}

if(checkBipartite(adj, n))

cout<<"YES"<<endl;

else

cout<<"NO"<<endl;

return 0;

}

// TC= O(n+E), SC= O(n+E)+O(n)+O(n)

**L10. Bipartite Graph (DFS):**

bool bipartiteDfs(itn node, vector<int> adj[], vector<int> &color){

if(color[node]==-1)

color[node]= 1;

for(auto it: adj[node]){

if(color[it]==-1){

color[it]= 1-color[node];

if(!bipartiteDfs(it, adj, color)){

return false;

}

}

else if(color[it]==color[node]){

return false;

}

}

return true;

}

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

vector<int> color(n, -1);

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

if(color[i]==-1)

if(!bipartiteDfs(i, adj, color))

return false;

}

return true;

}

int main(){

int n, m;

cin>>n>>m;

vector<int> adj[n];

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

int u, v;

cin>>u>>v;

adj[u].push\_back(v);

adj[v].push\_back(u);

}

if(checkBipartite(adj, n))

cout<<"YES"<<endl;

else

cout<<"NO"<<endl;

return 0;

}

// TC= O(n+E), SC= O(n+E)+O(n)+O(n)

**L11. Cycle Detection in Directed Graph using DFS:**

bool checkCycle(int node, vector<int> adj[], vector<int> &vis, vector<int> &dfsVis){

vis[node]= 1;

dfsVis[node]= 1;

for(auto it: adj[node]){

if(!vis[it]){

if(checkCycle(it, adj, vis, dfsVis))

return true;

}

else if(dfsVis[it]){

return true;

}

}

dfsVis[node]= 0;

return false;

}

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

vector<int> vis(N, 0);

vector<int> dfsVis(N, 0);

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

if(!vis[i]){

if(checkCycle(i, adj, vis, dfsVis)){

return true;

}

}

}

return false;

}

// TC= O(n+E), SC= O(n)+O(n)+Auxillary Space O(n)

**L12. Topological Sort (DFS):**

Topological Sort: Linear ordering of vertices such that if there is an edge u-->v, u appears before v in that ordering

Topo sort is for DAG (Directed Acylic Graphs)

Multiple topological sortings can exist for a single graph

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

vis[node]= 1;

for(auto it: adj[node]){

if(!vis[it]){

findTopoSortDfs(it, vis, st, adj);

}

}

st.push(node);

}

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

stack<int> st;

vector<int> vis(N, 0);

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

if(vis[i]==0)

findTopoSortDfs(i, vis, st, adj);

}

vector<int> topo;

while(!st.empty()){

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

st.pop();

}

return topo;

}

// TC= O(n+E), SC= O(n)+Auxillary Space O(n)+Stack O(n)

**L13. Topological Sort (BFS) (Kahn's Algorithm):**

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

queue<int> q;

vector<int> indegree(N, 0);

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

for(auto it: adj[i])

indegree[it]++;

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

if(indegree[i]==0)

q.push(i);

vector<int> topo;

while(!q.empty()){

int node= q.front();

q.pop();

topo.push\_back(node);

for(auto it: adj[node]){

indegree[it]--;

if(indegree[it]==0){

q.push(it);

}

}

}

return topo;

}

// TC= O(n+E), SC= O(n)+O(n)

**L14. Cycle Detection in Directed Graph using BFS (Kahn's Algorithm):**

Use Kahn's topological sort algorithm, if it was able to generate, then the graph is not cyclic, else graph is cyclic

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

queue<int> q;

vector<int> indegree(N, 0);

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

for(auto it: adj[i])

indegree[it]++;

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

if(indegree[i]==0)

q.push(i);

int cnt= 0;

while(!q.empty()){

int node= q.front();

q.pop();

cnt++;

for(auto it: adj[node]){

indegree[it]--;

if(indegree[it]==0){

q.push(it);

}

}

}

if(cnt==N)

return false;

return true;

}

**L15. Shortest Path in Undirected Graph with Unit Weights:**

void BFS(vector<int> adj[], int N, int src){

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

queue<int> q;

dist[src]= 0;

q.push(src);

while(q.empty()==false){

int node= q.front();

q.pop();

for(auto it: adj[node]){

if(dist[node]+1<dist[it]){

dist[it]= dist[node]+1;

q.push(it);

}

}

}

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

cout<<dist[i]<<" ";

cout<<endl;

}

// TC= O(n+E), SC= O(n)+O(n)

**L16. Shortest Path in a Weighted Directed Acyclic Graph (DAG):**

// DFS for finding Topo Sort

// BFS for finding Shortest Path

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

vis[node]= 1;

for(auto it: adj[node])

if(!vis[it.first])

findTopoSort(it.first, vis, st, adj);

st.push(node);

}

void shortestPath(int src, int N, vector<pair<int, int>> adj[]){

vector<int> vis(N, 0);

stack<int> st;

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

if(!vis[i])

findTopoSort(i, vis, st, adj);

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

dist[src]= 0;

while(!st.empty(){

int node= st.top();

st.pop();

if(dist[node]!=INT\_MAX)

for(auto it: adj[node])

if(dist[node]+it.second<dist[it.first])

dist[it.first]= dist[node]+it.second;

}

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

(dist[i]==INT\_MAX)? cout<<"INF ": cout<<dist[i]<<" ";

}

int main(){

int n, m;

cin>>n>>m;

vector<pair<int, int>> adj[n];

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

int u, v, wt;

cin>>u>>v>>wt;

adj[u].push\_back({v, wt});

}

shortestPath(0, n, adj);

return 0;

}

// TC= O(n+E)+O(n+E), SC= O(n)+O(n)

**L17. Dijkstra's Algorithms (Shortest Path in Undirected Graphs):**

Everyone having a question regarding the need of Priority Queue. The fact is that even without a Priority Queue, the algorithm will work absolutely fine, Priority Queue is just an optimizing technique here, as it always returns the minimum distance, hence we would have already figured the minimum distance of a node with least comparisons, greatly reducing the time complexity. Otherwise, if we were to use a normal Queue, we might find the shorter distance to a node in later checks. Hope you got the point!

int main(){

int n, m, source;

cin>>n>>m;

vector<pair<int, int>> g[n+1]; // 1-indexed adjacency list for graph

int a, b, wt;

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

cin>>a>>b>>wt;

g[a].push\_back({b, wt});

g[b].push\_back({a, wt});

}

cin>>source;

// Dijkstra's algorithm begins from here

priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>> pq; // min-heap, In pair=> (dist, from)

vector<int> distTo(n+1, INT\_MAX);

distTo[source]= 0;

pq.push(make\_pair(0, source)); // (dist, from)

while(!pq.empty()){

int dist= pq.top().first;

int prev= pq.top().second;

pq.pop();

vector<pair<int, int>>::iterator it;

for(it= g[prev].begin();it!=g[prev].end();it++){

int next= it->first;

int nextDist= it->second;

if(distTo[next]>distTo[prev]+nextDist){

distTo[next]= distTo[prev]+nextDist;

pq.push(make\_pair(distTo[next], next));

}

}

}

cout<<"The distances from source, "<<source<<" are: \n";

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

cout<<distTo[i]<<" ";

cout<<"\n";

return 0;

}

// TC= O((n+E)log(n)) (O(nlog(n))), SC= O(n)+O(n)

**L18. Minimum Spanning Tree (MST) Explanation:**

Spanning Tree: Convert a graph into a tree such that it contains exactly n nodes and n-1 edges, then we can call that tree as spanning tree

(every node should be reachable by every other node)

Given an undirected and connected graph G= (V, E),a spanning tree of the graph G is a tree that spans G (that is, it includes every vertex of G) and is a subgraph of G (every edge in the tree belongs to G)

The cost of the spanning tree is the sum of the weights of all the edges in the tree

There can be many spanning trees

Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees

There also can be many minimum spanning trees

1. Prims Algorithm

2. Kruskals Algorithm

**L19. Prims Algorithm (Minimum Spanning Tree):**

Rough Algorithm:

1) Create a set mstSet that keeps track of vertices already included in MST

2) Assign a key value to all vertices in the input graph

Initialize all key values as INFINITE

Assign the key value as 0 for the first vertex so that it is picked first

3) While mstSet doesn’t include all vertices

….a) Pick a vertex u which is not there in mstSet and has a minimum key value

….b) Include u to mstSet

….c) Update key value of all adjacent vertices of u

To update the key values, iterate through all adjacent vertices

For every adjacent vertex v, if the weight of edge u-v is less than the previous key value of v, update the key value as the weight of u-v

The idea of using key values is to pick the minimum weight edge from cut

The key values are used only for vertices that are not yet included in MST, the key value for these vertices indicates the minimum weight edges connecting them to the set of vertices included in MST

**L20. Prims Algorithm Implementation:**

Brute Force Approach:

int main(){

int N, m;

cin>>N>>m;

vector<pair<int, int>> adj[N];

int a, b, wt;

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

cin>>a>>b>>wt;

adj[a].push\_back({b, wt});

adj[b].push\_back({a, wt});

}

int parent[N];

int key[N];

bool mstSet[N];

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

key[i]= INT\_MAX;

mstSet[i]= false;

parent[i]= -1;

}

key[0]= 0;

parent[0]= -1;

for(int count=0;count<N-1;count++){

int mini= INT\_MAX, u;

for(int v=0;v<N;v++)

if(mstSet[v]==false && key[v]<mini)

mini= key[v], u= v;

mstSet[u]= true;

for(auto it: adj[u]){

int v= it.first;

int weight= it.second;

if(mstSet[v]==false && weight<key[v])

parent[v]= u, key[v]= weight;

}

}

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

cout<<parent[i]<<" - "<<i<<"\n";

return 0;

}

// TC= O(n\*n), SC= O(n)

Optimised Approach: Use priority queue:

int main(){

int N, m;

cin>>N>>m;

vector<pair<int, int>> adj[N];

int a, b, wt;

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

cin>>a>>b>>wt;

adj[a].push\_back({b, wt});

adj[b].push\_back({a, wt});

}

int parent[N];

int key[N];

bool mstSet[N];

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

key[i]= INT\_MAX, mstSet[i]= false;

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

key[0]= 0;

parent[0]= -1;

pq.push({0, 0});

for(int count= 0;count<N-1;count++){

int u= pq.top().second;

pq.pop();

mstSet[u]= true;

for(auto it: adj[u]){

int v= it.first;

int weight= it.second;

if(mstSet[v]==false && weight<key[v]){

parent[v]= u;

pq.push({key[v], v});

key[v]= weight;

}

}

}

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

cout<<parent[i]<<" - "<<i<<"\n";

return 0;

}

// TC= O(N+E+Nlog(N)) ~ O(Nlog(N)), SC= O(N)

**L22. Disjoint Set (Union By Rank and Path Compression):**

int parent[100000];

int rank[100000];

void makeSet(){

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

parent[i]= i;

rank[i]= 0;

}

}

int findPar(int node){

if(node==parent[node])

return node;

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

}

void union(int u, int v){

u= findPar(u);

v= findPar(v);

if(rank[u]<rank[v]){

parent[u]= v;

}

else if(rank[v]<rank[u]){

parent[v]= u;

}

else{

parent[v]= u;

rank[u]++;

}

}

void main(){

makeSet();

int m; cin>>m;

while(m--){

int u, v; cin>>u>>v;

union(u, v);

}

// If 2 and 3 belong to the same component or not

if(findPar(2)!=findPar(3)){

cout<<"Different Component"<<endl;

}

else{

cout<<"Same Component"<<endl;

}

}

**L23. Kruskal's Algorithm:**

struct node{

int u, v, wt;

node(int first, int second, int weight){

u= first;

v= second;

wt= weight;

}

};

bool comp(node a, node b){

return a.wt<b.wt;

}

int findPar(int u, vector<int> &parent){

if(u==parent[u])

return u;

return findPar(parent[u], parent);

}

void union(int u, int v, vector<int> &parent, vector<int> &rank){

u= findPar(u);

v= findPar(v);

if(rank[u]<rank[v])

parent[u]= v;

else if(rank[v]<rank[u])

parent[v]= u;

else{

parent[v]= u;

rank[u]++;

}

}

int main(){

int N, m;

cin>>N>>m;

vector<node> edges;

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

int u, v, wt;

cin>>u>>v>>wt;

edges.push\_back(node(u, v, wt));

}

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

vector<int> parent(N);

vector<int> rank(N, 0);

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

parent[i]= i;

int cost= 0;

vector<pair<int, int>> mst;

for(auto it: edges){

if(findPar(it.v, parent)!=findPar(it.u, parent)){

cost+= it.wt;

mst.push\_back({it.u, it.v});

union(it.u, it.v, parent, rank);

}

}

cout<<cost<<endl;

for(auto it: mst)

cout<<it.first<<" - "<<it.second<<endl;

return 0;

}

// TC= O(m\*log(m))+O(m\*O(4\*alpha)) ~ O(m\*log(m)), SC= O(m)+O(n)+O(n) ~ O(n)

**L24. Bridges in Graph (Cut Edge):**

Bridges: Those edges on whose removal, the graph is broken into 2 or more number of components

void dfs(int node, int parent, vector<int> &vis, vector<int> &tin, vector<int> &low, int &timer, vector<int> adj[]){

vis[node]= 1;

tin[node]= low[node]= timer++;

for(auto it: adj[node]){

if(it==parent) continue;

if(!vis[it]){

dfs(it, node, vis, tin, low, timer, adj);

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

if(low[it]>tin[node]){

// bridge is printed

cout<<node<<" "<<it<<endl;

}

}

else{

low[node]= min(low[node], tin[it]);

}

}

}

int main(){

int n, m;

cin>>n>>m;

vector<int> adj[n];

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

int u, v;

cin>>u>>v;

adj[u].push\_back(v);

adj[v].push\_back(u);

}

vector<int> tin(n, -1);

vector<int> low(n, -1);

vector<int> vis(n, 0);

int timer= 0;

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

if(!vis[i])

dfs(i, -1, vis, tin, low, timer, adj);

return 0;

}

// TC= O(n+E), SC= O(n)

**L25. Articulation Point (Cut Vertex):**

void dfs(int node, int parent, vector<int> &vis, vector<int> &tin, vector<int> &low, int &timer, vector<int> adj[], vector<int> &isArticulation){

vis[node]= 1;

tin[node]= low[node]= timer++;

int child= 0;

for(auto it: adj[node]){

if(it==parent)

continue;

if(!vis[it]){

dfs(it, node, vis, tin, low, timer, adj, isArticulation);

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

if(low[it]>=tin[node] && parent!=-1)

ifArticulation[node]= 1;

child++;

}

else{

low[node]= min(low[node], tin[it]);

}

}

if(parent==-1 && child>1)

isArticulation[node]= 1;

}

int main(){

int n, m; cin>>n>>m;

vector<int> adj[n];

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

int u, v; cin>>u>>v;

adj[u].push\_back(v);

adj[v].push\_back(u);

}

vector<int> tin(n, -1);

vector<int> low(n, -1);

vector<int> vis(n, 0);

vector<int> isArticulation(n, 0);

int timer= 0;

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

if(!vis[i])

dfs(i, -1, vis, tin, low, timer, adj, isArticulation);

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

if(isArticulation[i]==1)

cout<<i<<endl;

return 0;

}

// TC= O(N+E), SC= O(N)