Code – 1: Implementation of Adjacency Matrix

#include<bits/stdc++.h>

using namespace std;

int main()

{

ios\_base::sync\_with\_stdio(false);

cin.tie(NULL);

cout.tie(NULL);

int node,edge;

int x,y,weight;

cin >> node >> edge;

int ara[node+1][node+1];

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

for(int j=0;j<=node;j++)

{

ara[i][j] = 0;

}

}

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

cin >> x >> y >> weight;

ara[x][y] = weight;

ara[y][x] = weight;

}

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

for(int j=1;j<=node;j++)

{

cout << ara[i][j] << " ";

}

cout << endl;

}

return 0;

}

Code – 2: Implementation of Adjacency List

#include<bits/stdc++.h>

using namespace std;

vector < pair < int,int > > v[10000];

int main()

{

ios\_base::sync\_with\_stdio(false);

cin.tie(NULL);

cout.tie(NULL);

int node,edge;

int x,y,weight;

cin >> node >> edge;

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

{

cin >> x >> y >> weight;

v[x].push\_back(make\_pair(y,weight));

v[y].push\_back(make\_pair(x,weight));

}

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

{

cout << i << " -- ";

for(int j=0;j<v[i].size();j++)

{

cout << v[i][j].first << "-" << v[i][j].second << " ";

}

cout << endl;

}

return 0;

}

Code – 4: Breadth First Search

\*\*\*Implementing Breadth First Search to find shortest path between two nodes in an unweighted, undirected graph

#include<bits/stdc++.h>

using namespace std;

#define endl '\n'

vector < int > v[1000];

int level[1000];

bool visited[1000];

int bfs(int s,int s1)

{

queue < int > q;

q.push(s);

level[s] = 0;

visited[s] = true;

while(!q.empty())

{

int p = q.front();

q.pop();

for(int i=0;i<v[p].size();i++)

{

if(visited[v[p][i]] == false)

{

q.push(v[p][i]);

visited[v[p][i]] = true;

level[v[p][i]] = level[p] + 1;

}

}

}

return level[s1];

}

int main()

{

ios\_base::sync\_with\_stdio(false);

cin.tie(NULL);

cout.tie(NULL);

int node,edge;

int x,y;

cin >> node >> edge;

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

{

cin >> x >> y;

v[x].push\_back(y);

v[y].push\_back(x);

}

cout << "Shortest Path : " << bfs(1,7) << endl;

return 0;

}

Code – 4: Depth First Search ( Calculating Number of connected component )

#include<bits/stdc++.h>

using namespace std;

#define endl '\n'

vector < int > v[1000];

bool visited[1000];

void dfs(int s) {

visited[s] = true;

for(int i=0;i<v[s].size();i++) {

if(visited[v[s][i]] == false)

dfs(v[s][i]);

}

}

int main() {

int node,edge,x,y,component=0;

cin >> node >> edge;

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

cin >> x >> y;

v[x].push\_back(y);

v[y].push\_back(x);

}

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

if(visited[i] == false) {

dfs(i);

component++;

}

}

cout << component << endl;

return 0;

}

Code – 5: Topological Sort

#include<bits/stdc++.h>

using namespace std;

#define endl '\n'

vector < int > v[1000];

bool visited[1000];

int indegree[10000];

vector < int > t;

void topological\_sort(int n) {

queue < int > q;

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

{

for(int j=0;j<v[i].size();j++)

{

indegree[v[i][j]]++;

}

}

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

{

if(indegree[i]==0)

{

q.push(i);

visited[i] = true;

}

}

while(!q.empty()) {

int p = q.front();

q.pop();

t.push\_back(p);

for(int i=0;i<v[p].size();i++)

{

int k = v[p][i];

if(visited[k]==false)

{

indegree[k]--;

if(indegree[k]==0)

{

q.push(k);

visited[k] = true;

}

}

}

}

}

int main()

{

ios\_base::sync\_with\_stdio(false);

cin.tie(NULL);

cout.tie(NULL);

int vertex,edge,x,y;

cin >> vertex >> edge;

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

{

cin >> x >> y;

v[x].push\_back(y);

}

topological\_sort(vertex);

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

{

cout << t[i] << " ";

}

cout << endl;

return 0;

}

Code – 6: Minimum Spanning Tree (Kruskal’s Algorithm)

#include<bits/stdc++.h>

using namespace std;

const int MAX = 1e4+5;

int id[MAX],node,edge;

pair < int, pair < int,int > > p[MAX];

void initialize()

{

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

{

id[i] = i;

}

}

int root(int x)

{

while(id[x] != x)

{

id[x] = id[id[x]];

x = id[x];

}

return x;

}

void union1(int x,int y)

{

int p = root(x);

int q = root(y);

id[p] = id[q];

}

int kruskal(pair<int,pair<int,int> > p[])

{

int x,y,cost,minimum\_cost = 0;

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

{

x = p[i].second.first;

y = p[i].second.second;

cost = p[i].first;

if(root(x) != root(y))

{

minimum\_cost += cost;

union1(x,y);

}

}

return minimum\_cost;

}

int main()

{

ios\_base::sync\_with\_stdio(false);

cin.tie(NULL);

cout.tie(NULL);

int x,y,weight,minimum\_cost,cost;

initialize();

cin >> node >> edge;

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

{

cin >> x >> y >> weight;

p[i] = make\_pair(weight,make\_pair(x,y));

}

sort(p,p+edge);

minimum\_cost = kruskal(p);

cout << minimum\_cost << endl;

return 0;

}

Code – 7: Minimum Spanning Tree (Prim’s Algorithm)

#include<bits/stdc++.h>

using namespace std;

const int MAX=1e4+5;

typedef pair < int,int > PII;

bool marked[MAX];

vector < PII > ara[MAX];

int prim(int x)

{

priority\_queue < PII, vector<PII>, greater<PII> > Q;

int y,minimum\_cost = 0;

PII p;

Q.push({0,x});

while(!Q.empty())

{

p = Q.top();

Q.pop();

int x = p.second;

if(marked[x]==true)

continue;

minimum\_cost += p.first;

marked[x] = true;

for(int i=0;i<ara[x].size();i++)

{

int y = ara[x][i].second;

if(marked[y] == false)

Q.push(ara[x][i]);

}

}

return minimum\_cost;

}

int main()

{

ios\_base::sync\_with\_stdio(false);

cin.tie(NULL);

cout.tie(NULL);

int node,edge,x,y,weight;

cin >> node >> edge;

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

{

cin >> x >> y >> weight;

ara[x].push\_back(make\_pair(weight,y));

ara[y].push\_back(make\_pair(weight,x));

}

int minimum\_cost = prim(1);

cout << minimum\_cost << endl;

return 0;

}

Code – 8: Dijkstra Algorithm

#include<bits/stdc++.h>

using namespace std;

#define SIZE 100000+1

bool visited[SIZE];

int dist[SIZE];

vector < pair < int,int > > v[SIZE];

void dijkstra()

{

multiset < pair < int,int > > q;

q.insert({0,1});

while(!q.empty())

{

pair < int,int > p = \*q.begin();

q.erase(q.begin());

int next = p.second;

if(visited[next])

continue;

visited[next] = true;

for(int i=0;i<v[next].size();i++)

{

int w = v[next][i].second;

int y = v[next][i].first;

if(dist[next]+w < dist[y])

{

dist[y] = dist[next] + w;

q.insert({dist[y],y});

}

}

}

}

int main()

{

ios\_base::sync\_with\_stdio(false);

cin.tie(NULL);

cout.tie(NULL);

int vertex,edge,x,y,weight;

cin >> vertex >> edge;

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

{

cin >> x >> y >> weight;

v[x].push\_back(make\_pair(y,weight));

}

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

{

visited[i] = false;

dist[i] = 1e9;

}

dist[1] = 0;

dijkstra();

for(int i=2;i<=vertex;i++)

{

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

}

cout << endl;

return 0;

}

Code – 9: Floyd Warshall’s Algorithm

#include<bits/stdc++.h>

using namespace std;

#define INF 1e9+7

int main() {

int node,edge,x,y,weight;

cin >> node >> edge;

int dist[node+1][node+1];

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

for(int j=1;j<=node;j++) {

if(i==j) dist[i][j] = 0;

else dist[i][j] = INF;

}

}

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

cin >> x >> y >> weight;

dist[x][y] = weight;

dist[y][x] = weight;

}

for(int k=1;k<=node;k++) {

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

for(int j=1;j<=node;j++) {

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

}

}

}

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

for(int j=1;j<=node;j++) {

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

}

cout << endl;

}

return 0;

}

Code – 10: Bellman Ford Algorithm

#include<bits/stdc++.h>

using namespace std;

#define INF 1e9+7

int main()

{

ios\_base::sync\_with\_stdio(false);

cin.tie(NULL);

cout.tie(NULL);

int node,edge;

cin >> node >> edge;

vector < tuple < int,int,int > > v;

int dist[node+1];

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

{

dist[i] = INF;

}

int starting\_node;

cin >> starting\_node;

dist[starting\_node] = 0;

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

{

int a,b,w;

cin >> a >> b >> w;

v.push\_back({a,b,w});

}

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

{

for(auto e : v)

{

int a,b,w;

tie(a,b,w) = e;

dist[b] = min(dist[b],dist[a]+w);

}

}

for(auto e : v)

{

int a,b,w;

tie(a,b,w) = e;

if(dist[a]+w < dist[b])

{

cout << "Negative Cycle." << endl;

return 0;

}

}

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

{

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

}

cout << endl;

return 0;

}

Code – 11: Articulation Point

#include<bits/stdc++.h>

using namespace std;

vector < int > v[101];

int dis[101];

int low[101];

bool visited[101];

int timer;

vector < int > arti;

void articulation\_point(int node,int par)

{

timer++;

dis[node] = low[node] = timer;

visited[node] = true;

int no\_of\_children = 0;

for(int child : v[node])

{

if(child == par) continue;

if(visited[child]==true)

low[node] = min(low[node],dis[child]);

else if(visited[child] == false)

{

articulation\_point(child,node);

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

if(dis[node] <= low[child] && par != -1)

{

arti.push\_back(node);

}

no\_of\_children++;

}

if(no\_of\_children>1 && par == -1)

{

arti.push\_back(node);

}

}

}

int main()

{

ios\_base::sync\_with\_stdio(false);

cin.tie(NULL);

cout.tie(NULL);

int node,edge,x,y;

cin >> node >> edge;

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

{

cin >> x >> y;

v[x].push\_back(y);

v[y].push\_back(x);

}

articulation\_point(1,-1);

for(int n : arti)

{

cout << n << " ";

}

cout << endl;

return 0;

}

Code – 12: Bridges

#include<bits/stdc++.h>

using namespace std;

vector < int > ara[101];

int in[101],low[101];

bool visited[101];

int timer = 0;

void dfs(int node,int par) {

visited[node] = true;

in[node] = low[node] = timer;

timer++;

for(int child : ara[node]) {

if(child == par) continue;

if(visited[child] == true) {

low[node] = min(low[node],in[child]);

}

else {

dfs(child,node);

if(low[child] > in[node])

cout << node << " - " << child << endl;

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

}

}

}

int main() {

ios\_base::sync\_with\_stdio(false); cin.tie(NULL); cout.tie(NULL);

int vertex,edge,x,y;

cin >> vertex >> edge;

while(edge--)

cin >> x >> y, ara[x].push\_back(y), ara[y].push\_back(x);

dfs(1,-1);

return 0;

}

Code – 13: Strongly Connected Component ( Kosaraju’s Algorithm )

#include<bits/stdc++.h>

using namespace std;

vector < int > ar[1001];

vector < int > tr[1001];

vector < int > order;

vector < int > SCC;

bool vis[1001];

void dfs(int node)

{

vis[node] = true;

for(int child : ar[node])

{

if(vis[child]==false)

dfs(child);

}

order.push\_back(node);

}

void dfs1(int node)

{

vis[node] = true;

for(int child : tr[node])

{

if(vis[child]==false)

dfs1(child);

}

SCC.push\_back(node);

}

int main()

{

ios\_base::sync\_with\_stdio(false); cin.tie(NULL);

cout.tie(NULL);

int vertex,edge,a,b;

cin >> vertex >> edge;

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

cin >> a >> b;

ar[a].push\_back(b);

tr[b].push\_back(a);

}

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

if(vis[i]==false)

dfs(i);

}

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

vis[i] = false;

}

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

{

if(vis[order[vertex-i]]==false)

{

SCC.clear();

dfs1(order[vertex-i]);

for(int node : SCC)

{

cout << node << "\t";

}

cout << endl;

}

}

return 0;

}

Code - 14: Strongly Connected Component ( Tarjan’s Algorithm )

Code – 15: DFS on 2D Grid

#include<bits/stdc++.h>

using namespace std;

int n,m;

bool vis[1001][1001];

int dx[] = {-1,0,1,0};

int dy[] = {0,1,0,-1};

bool is\_valid(int x,int y) {

if(x<1 || y<1 || x>n || y>m)

return false;

if(vis[x][y]==true)

return false;

return true;

}

void dfs(int x,int y) {

vis[x][y] = true;

cout << x << '\t' << y << endl;

for(int i=0;i<4;i++) {

if(is\_valid(x+dx[i],y+dy[i]))

dfs(x+dx[i],y+dy[i]);

}

}

int main() {

ios\_base::sync\_with\_stdio(false); cin.tie(NULL);

cout.tie(NULL);

cin >> n >> m;

dfs(1,1);

return 0;

}