Name: Tirth Hihoriya

Roll no.: 18bce244

Prac- 7 : Write a program to find all the spanning trees of a complete directed graph.

```
#include<bits/stdc++.h>

using namespace std;

int count_num=0;

set<string>spanning_trees;


class Graph
{
    public:
    int V, E;

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

    Graph(int V,int E){
        this->V = V;
        this->E = E;
    }

    void addEdge(int u, int v)
    {
        edges.push_back({u, v});
    }

    int kruskalMST();

    void generate_spanning_tree();

    bool addInSet(vector<pair<int,int>>v);

    vector<pair<int,int>> delete_ele(vector<pair<int,int>> vec, pair<int,int>
num);
};


struct DisjointSets
{
    int *parent, *rnk;
    int n;
    DisjointSets(int n)
    {
        this->n = n;
        parent = new int[n+1];
        rnk = new int[n+1];

        for (int i = 0; i <= n; i++)
        {
            rnk[i] = 0;
            parent[i] = i;
        }
    }

    int find(int u)
    {
        if (u != parent[u])
            parent[u] = find(parent[u]);
        return parent[u];
    }

    void merge(int x, int y)
    {
        x = find(x), y = find(y);

        if (rnk[x] > rnk[y])
            parent[y] = x;
        else
            parent[x] = y;

        if (rnk[x] == rnk[y])
            rnk[y]++;
    }
};

int Graph::kruskalMST()
{
    int mst_wt = 0;

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

    DisjointSets ds(V);

    vector< pair<int, int> >::iterator it;
    for (it=edges.begin(); it!=edges.end(); it++)
    {
        int u = it->first;
        int v = it->second;

        int set_u = ds.find(u);
        int set_v = ds.find(v);

        if (set_u != set_v)
        {
            branch.push_back({u,v});
            mst_wt += 1;

            ds.merge(set_u, set_v);
        }else{
            chord.push_back({u,v});
        }
    }

    return mst_wt;
}

void DFSCycle(int u, int p, int color[], int mark[], int par[], int& cyclenumber,
              vector<vector<int>>& graph, vector<vector<int>>& cycles)
{
   if (color[u] == 2) {
      return;
   }
   if (color[u] == 1) {
      cyclenumber++;
      int cur = p;
      mark[cur] = cyclenumber;
      while (cur != u) {
         cur = par[cur];
         mark[cur] = cyclenumber;
      }
      return;
   }
   par[u] = p;
   color[u] = 1;
   for (int i=0;i<graph[u].size();i++) {
      if (graph[u][i] == par[u]) {
         continue;
      }
      DFSCycle(graph[u][i], u, color, mark, par, cyclenumber,graph,cycles);
   }
   color[u] = 2;
}

void printCycles(int edges, int mark[], int& cyclenumber,vector<vector<int>>&
cycles,
                 vector<pair<int,int>>& cycle)
{
    for (int i = 1; i <= edges; i++) {
        if (mark[i] != 0)
            cycles[mark[i]].push_back(i);
    }
    for (int i = 1; i <= cyclenumber; i++) {
        int pre=-1,first=-1;
        for (int x=0;x<cycles[i].size();x++) {
            if(first==-1)
                first=cycles[i][x];

            if(pre!=-1) {
```

```
                int temp1=pre,temp2=cycles[i][x];
                if(temp1>temp2){
                    swap(temp1,temp2);
                }
                cycle.push_back({temp1,temp2});
            }

            pre = cycles[i][x];
        }
        if(first!=-1) {
            if(pre>first){
                swap(pre,first);
            }
            cycle.push_back({pre,first});
        }
    }
}

void Graph:: generate_spanning_tree(){

    sort(branch.begin(),branch.end());
    branch.push_back({chord[0].first,chord[0].second});
    pair<int,int>added_chord={chord[0].first,chord[0].second};
    chord= delete_ele(chord,{chord[0].first,chord[0].second});

    vector<vector<int>>edges_cycle(1000);
    vector<vector<int>>cycles(1000);

    for(int i=0;i<branch.size();i++){
        edges_cycle[branch[i].first].push_back(branch[i].second);
        edges_cycle[branch[i].second].push_back(branch[i].first);
    }

    int N=1000;

    int color[N];

    int par[N];

    int mark[N];

    int cyclenumber = 0;
    vector<pair<int,int>>cycle;

    DFSCycle(1, 0, color, mark, par, cyclenumber,edges_cycle,cycles);

    printCycles(edges.size(), mark, cyclenumber,cycles,cycle);

    for(int i=0;i<cycle.size();i++){
        if(cycle[i].first == added_chord.first && cycle[i].second ==
added_chord.second)
            continue;

        pair<int,int>temp = {cycle[i].first,cycle[i].second};
```

```
            vector<pair<int,int>>temp_branch=delete_ele(branch,temp);

            if(addInSet(temp_branch)) {
                branch=temp_branch;
                chord.push_back(temp);
                generate_spanning_tree();
                break;
            }
        }
}

vector<pair<int,int>> Graph::delete_ele(vector<pair<int,int>> vec, pair<int,int>
num)
{
    vector<pair<int,int>>::reverse_iterator itr1;

    for (itr1 = vec.rbegin(); itr1 < vec.rend(); itr1++) {
        if ((*itr1).first == num.first && (*itr1).second == num.second) {
            vec.erase((itr1 + 1).base());
        }
    }

    return vec;
}

bool Graph::addInSet(vector<pair<int,int>>v)
{
    string s;
    sort(v.begin(),v.end());

    for(int i=0;i<v.size();i++){
        s.push_back(v[i].first+48);
        s.push_back(v[i].second+48);
    }

    if(spanning_trees.count(s)==0){
            cout<<"\nSpanning Tree"<<++count_num<<": ";
            for(int i=0;i<v.size();i++)
                cout<<"("<<v[i].first<<","<<v[i].second<<")\t";
            cout<<endl<<endl;
        spanning_trees.insert(s);
        return true;
    }

    return false;
}


int main()
{
    vector<vector<int>> edges_cycle(1000);
    vector<vector<int>> cycles(1000);

    edges_cycle[1].push_back(2);
    edges_cycle[2].push_back(1);
    edges_cycle[2].push_back(3);
    edges_cycle[3].push_back(2);
    edges_cycle[2].push_back(5);
    edges_cycle[5].push_back(2);
    edges_cycle[5].push_back(4);
    edges_cycle[4].push_back(5);
    edges_cycle[3].push_back(4);
    edges_cycle[4].push_back(3);
    edges_cycle[5].push_back(6);
    edges_cycle[6].push_back(5);
    edges_cycle[7].push_back(6);
    edges_cycle[6].push_back(7);
    edges_cycle[8].push_back(6);
    edges_cycle[6].push_back(8);

    int N=1000;

    int color[N];

    int par[N];

    int mark[N];

    int cyclenumber = 0;
    vector<pair<int,int>>cycle;
    DFSCycle(1, 0, color, mark, par, cyclenumber, edges_cycle,cycles);

    int V = 9, E = 9;
    Graph g(V, E);

    g.addEdge(1, 2);
    g.addEdge(2, 3);
    g.addEdge(3, 4);
    g.addEdge(4, 5);
    g.addEdge(2, 5);
    g.addEdge(5, 6);
    g.addEdge(6, 7);
    g.addEdge(7, 8);
    g.addEdge(6, 8);

    g.kruskalMST();

    bool temp = g.addInSet(g.branch);

    g.generate_spanning_tree();
}
```

OUTPUT :

```
Spanning Tree1: (1,2)   (2,3)   (2,5)   (3,4)   (5,6)   (6,7)   (6,8)

Spanning Tree2: (1,2)   (2,5)   (3,4)   (4,5)   (5,6)   (6,7)   (6,8)

Spanning Tree3: (1,2)   (2,5)   (3,4)   (4,5)   (5,6)   (6,8)   (7,8)

Spanning Tree4: (1,2)   (2,3)   (2,5)   (4,5)   (5,6)   (6,8)   (7,8)

Spanning Tree5: (1,2)   (2,3)   (2,5)   (4,5)   (5,6)   (6,7)   (6,8)

Spanning Tree6: (1,2)   (2,3)   (3,4)   (4,5)   (5,6)   (6,7)   (6,8)

Spanning Tree7: (1,2)   (2,3)   (3,4)   (4,5)   (5,6)   (6,8)   (7,8)

Spanning Tree8: (1,2)   (2,3)   (2,5)   (3,4)   (5,6)   (6,8)   (7,8)

Spanning Tree9: (1,2)   (2,3)   (2,5)   (3,4)   (5,6)   (6,7)   (7,8)

Spanning Tree10: (1,2)  (2,5)   (3,4)   (4,5)   (5,6)   (6,7)   (7,8)
```