**CS-304-AI -LAB(LAB TASK-2)**

**ROLL-423135**

**NAME-HARSHITH SUDA**

**CODE-1**

#include<bits/stdc++.h>

#include<chrono>

using namespace std;

using std::chrono::high\_resolution\_clock;

using std::chrono::duration\_cast;

using std::chrono::duration;

using std::chrono::milliseconds;

struct Node{

    Node\* parent;

    int nodeNumber;

    int cost;

};

void printVectorOfVector(vector<vector<int>>&v){

    for(int i=0;i<v.size();i++){

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

*// cout<<"value of i:"<<i<<"value of j:"<<j<<" ";*

            cout<<v[i][j]<<" ";

        }

        cout<<endl;

    }

    return;

}

vector<vector<int>>CreateGraph(){

    int n=1000;

    vector<vector<int>>g(n,vector<int>(n));

    for(int i=0;i<n;i++){

        for(int j=i+1;j<n;j++){

            g[i][j]=rand()%1000; *//w=0(represents no edge)*

            g[j][i]=g[i][j];

        }

    }

    return g;

}

bool isConnected(vector<vector<int>>&graph){

    int n=graph.size();

    for(int i=0;i<n;i++){

        bool flag=false;

        for(int j=0;j<n;j++){

            if(graph[i][j]){

                flag=true;

            }

        }

        if(!flag){

            return false;

        }

    }

    return true;

}

struct comp{

    bool operator()(const Node\* a,const Node\* b)const{

        return a->cost>b->cost;

    }

};

Node\* initNode(int start,int cost){

    Node\* node=new Node();

    node->cost=cost;

    node->nodeNumber=start;

    node->parent=NULL;

    return node;

}

void RecursivePrintPath(Node\* curr){

    vector<int>path;

    while(curr){

        path.push\_back(curr->nodeNumber);

        curr=curr->parent;

    }

    for(int i=path.size()-1;i>0;i--){

        cout<<path[i]<<"->";

    }

    cout<<path[0]<<endl;

*// cout<<endl;*

    return;

}

void uniformCostSearch(vector<vector<int>>&graph,int start,int goal,double &timeTaken,int &edgesExplored,int &pathCost,unordered\_set<int>&explored){

*// cout<<"here1";*

    edgesExplored=0;

    map<int,Node\*>m;

    priority\_queue<Node\*,vector<Node\*>,comp>q;

    Node\* startNode=initNode(start,0);

    q.push(startNode);

    m.insert({start,startNode});

    auto t1=high\_resolution\_clock::now();

    while(!q.empty()){

        Node\* curr=q.top();

        q.pop();

        if(curr->nodeNumber==goal){

            auto t2=high\_resolution\_clock::now();

            duration<double,std::milli>ms\_double=t2-t1;

            timeTaken=ms\_double.count();

            pathCost=curr->cost;

*// RecursivePrintPath(curr);*

            return;

        }

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

            if(graph[curr->nodeNumber][i]!=0){

                if(explored.find(i)==explored.end()){

                    edgesExplored++;

                    if(m.find(i)==m.end() || m[i]->cost>curr->cost+graph[curr->nodeNumber][i]){

                        if(m.find(i)==m.end()){

                            Node\* node=new Node();

                            node->nodeNumber=i;

                            m[i]=node;

                        }

                        m[i]->parent=curr;

                        m[i]->cost=curr->cost+graph[curr->nodeNumber][i];

                        q.push(m[i]);

                    }

                }

            }

        }

        explored.insert(curr->nodeNumber);

    }

    return;

}

void BFS(vector<vector<int>>&graph,int start,int goal,double &timeTaken,int &edgesExplored,int &pathCost,unordered\_set<int>&explored){

    edgesExplored=0;

    unordered\_map<int,Node\*>m;

*// unordered\_set<int>expanded;*

    queue<Node\*>q;

    Node\* startNode=initNode(start,0);

    q.push(startNode);

    m.insert({start,startNode});

    auto t1=high\_resolution\_clock::now();

    while(!q.empty()){

        Node\* curr=q.front();

        q.pop();

*// if(expanded.find(curr->nodeNumber)!=expanded.end()){*

*//     continue;*

*// }*

        if(curr->nodeNumber==goal){

            auto t2=high\_resolution\_clock::now();

            duration<double,std::milli>mil\_double=t2-t1;

            timeTaken=mil\_double.count();

            pathCost=curr->cost;

*// RecursivePrintPath(curr);*

            return;

        }

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

            if(graph[curr->nodeNumber][i]){

                if(explored.find(i)==explored.end()){

                    edgesExplored++;

                    if(m.find(i)==m.end()|| m[i]->cost>curr->cost+graph[curr->nodeNumber][i]){

                        if(m.find(i)==m.end()){

                            Node\* node=new Node();

                            node->nodeNumber=i;

                            m[i]=node;

                        }

                        m[i]->parent=curr;

                        m[i]->cost=curr->cost+graph[curr->nodeNumber][i];

                        q.push(m[i]);

                    }

                }

            }

        }

        explored.insert(curr->nodeNumber);

    }

    return;

}

void DFS(vector<vector<int>>&graph,int start,int goal,double &timeTaken,int &edgesExplored,int &pathCost,unordered\_set<int>&explored){

    edgesExplored=0;

    unordered\_map<int,Node\*>m;

*// unordered\_set<int>expanded;*

    stack<Node\*>s;

    Node\* startNode=initNode(start,0);

    s.push(startNode);

    m.insert({start,startNode});

    auto t1=high\_resolution\_clock::now();

    while(!s.empty()){

        Node\* curr=s.top();

        s.pop();

*// if(expanded.find(curr->nodeNumber)!=expanded.end()){*

*//     continue;*

*// }*

        if(curr->nodeNumber==goal){

            auto t2=high\_resolution\_clock::now();

            duration<double,std::milli>mil\_double=t2-t1;

            timeTaken=mil\_double.count();

            pathCost=curr->cost;

*// RecursivePrintPath(curr);*

            return;

        }

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

            if(graph[curr->nodeNumber][i]){

                if(explored.find(i)==explored.end()){

                    edgesExplored++;

                    if(m.find(i)==m.end()|| m[i]->cost>curr->cost+graph[curr->nodeNumber][i]){

                        if(m.find(i)==m.end()){

                            Node\* node=new Node();

                            node->nodeNumber=i;

                            m[i]=node;

                        }

                        m[i]->parent=curr;

                        m[i]->cost=curr->cost+graph[curr->nodeNumber][i];

                        s.push(m[i]);

                    }

                }

            }

        }

        explored.insert(curr->nodeNumber);

    }

    return;

}

*//SEPERATE STRUCT OF NODE FOR ITEATIVE DEEPENING SEARCH(ADDITION OF LEVEL)*

struct IterativeNode{

    IterativeNode\* parent;

    int nodeNumber;

    int cost;

    int depth;

};

IterativeNode\* initNodeIterative(int start,int cost){

    IterativeNode\* node=new IterativeNode();

    node->cost=cost;

    node->nodeNumber=start;

    node->parent=NULL;

    node->depth=0;

    return node;

}

void RecursivePrintPathIterative(IterativeNode\* curr){

    vector<int>path;

    while(curr){

        path.push\_back(curr->nodeNumber);

        curr=curr->parent;

    }

    for(int i=path.size()-1;i>0;i--){

        cout<<path[i]<<"->";

    }

    cout<<path[0]<<endl;

*// cout<<endl;*

    return;

}

void IterativeDeepeningSearch(vector<vector<int>>&graph,int start,int goal,double &timeTaken,int &edgesExplored,int &pathCost,unordered\_set<int>&explored,bool &found,int level){

    edgesExplored=0;

    unordered\_map<int,IterativeNode\*>m;

*// unordered\_set<int>expanded;*

    stack<IterativeNode\*>s;

    IterativeNode\* startNode=initNodeIterative(start,0);

    s.push(startNode);

    m.insert({start,startNode});

    auto t1=high\_resolution\_clock::now();

    while(!s.empty()){

        IterativeNode\* curr=s.top();

        s.pop();

*// if(expanded.find(curr->nodeNumber)!=expanded.end()){*

*//     continue;*

*// }*

        if(curr->nodeNumber==goal){

            auto t2=high\_resolution\_clock::now();

            duration<double,std::milli>mil\_double=t2-t1;

            timeTaken=mil\_double.count();

            pathCost=curr->cost;

            found=true;

*// RecursivePrintPathIterative(curr);*

            return;

        }

        if(curr->depth>level){

            continue;

        }

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

            if(graph[curr->nodeNumber][i]){

                if(explored.find(i)==explored.end()){

                    edgesExplored++;

                    if(m.find(i)==m.end()|| m[i]->cost>curr->cost+graph[curr->nodeNumber][i]){

                        if(m.find(i)==m.end()){

                            IterativeNode\* node=new IterativeNode();

                            node->nodeNumber=i;

                            m[i]=node;

                        }

                        m[i]->parent=curr;

                        m[i]->depth=curr->depth+1;

                        m[i]->cost=curr->cost+graph[curr->nodeNumber][i];

                        s.push(m[i]);

                    }

                }

            }

        }

        explored.insert(curr->nodeNumber);

    }

    return;

}

void IterativeDeepeningDriverFunction(vector<vector<int>>&graph,int start,int goal,double &timeTaken,int &edgesExplored,int &pathCost,unordered\_set<int>&explored){

    bool found=false;

    for(int i=0;i<graph.size();i++){

*// cout<<"level:"<<i<<endl;*

        IterativeDeepeningSearch(graph,start,goal,timeTaken,edgesExplored,pathCost,explored,found,i);

        if(found){

            return;

        }

    }

}

void printTable(vector<double>UCSData,vector<double>BFSData,vector<double>DFSData,vector<double>IDFSData,int n){

    cout<<"   UCS  "<<"BFS  "<<"DFS  "<<"IDFS  "<<endl;

    cout<<"Time Taken: "<<UCSData[0]/n<<"  "<<BFSData[0]/n<<"  "<<DFSData[0]/n<<"  "<<IDFSData[0]/n<<endl;

    cout<<"EdgesExplored: "<<UCSData[1]/n<<"  "<<BFSData[1]/n<<"  "<<DFSData[1]/n<<"  "<<IDFSData[1]/n<<endl;

    cout<<"pathCost: "<<UCSData[2]/n<<"  "<<BFSData[2]/n<<"  "<<DFSData[2]/n<<"  "<<IDFSData[2]/n<<endl;

    cout<<"No. of Nodes explored(out of 1000): "<<UCSData[3]/n<<"  "<<BFSData[3]/n<<"  "<<DFSData[3]/n<<"  "<<IDFSData[3]/n<<endl;

    return;

}

int main(){

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

    cin.tie(0);

    vector<vector<int>>g=CreateGraph();

*// vector<vector<int>>storedGraph=g;*

*// printVectorOfVector(g);*

*// double t;*

*// int edgesExplored,pathCost;*

*// unordered\_set<int>s;*

*// cout<<"here2"<<endl;*

    vector<double>UCSData(4,0),BFSData(4,0),DFSData(4,0),IDFSData(4,0);

    int n=5;

    for(int i=0;i<n;i++){

        int start=rand()%1000;

        int goal=rand()%1000+1;

        double t;

        int edgesExplored,pathCost;

        unordered\_set<int>s;

        uniformCostSearch(g,start,goal,t,edgesExplored,pathCost,s);

        UCSData[0]+=t;UCSData[1]+=edgesExplored;UCSData[2]+=pathCost;UCSData[3]+=s.size();

        BFS(g,start,goal,t,edgesExplored,pathCost,s);

        BFSData[0]+=t;BFSData[1]+=edgesExplored;BFSData[2]+=pathCost;BFSData[3]+=s.size();

        DFS(g,start,goal,t,edgesExplored,pathCost,s);

        DFSData[0]+=t;DFSData[1]+=edgesExplored;DFSData[2]+=pathCost;DFSData[3]+=s.size();

        IterativeDeepeningDriverFunction(g,start,goal,t,edgesExplored,pathCost,s);

        IDFSData[0]+=t;IDFSData[1]+=edgesExplored;IDFSData[2]+=pathCost;IDFSData[3]+=s.size();

*// cout<<"Graph is connected: "<<((isConnected(g))?"YES":"NO")<<endl;*

    }

    printTable(UCSData,BFSData,DFSData,IDFSData,n);

*// cout<<"UNIFORM COST SEARCH"<<endl;*

*// cout<<"timeTaken: "<<t<<endl;*

*// cout<<"edgesExplored: "<<edgesExplored<<endl;*

*// cout<<"pathCost: "<<pathCost<<endl;*

*// cout<<"No. of Nodes explored(out of 1000): "<<s.size()<<endl;*

*// BFS(g,0,100,t,edgesExplored,pathCost,s);*

*// cout<<"BREADTH FIRST SEARCH"<<endl;*

*// cout<<"timeTaken: "<<t<<endl;*

*// cout<<"edgesExplored: "<<edgesExplored<<endl;*

*// cout<<"pathCost: "<<pathCost<<endl;*

*// cout<<"No. of Nodes explored(out of 1000): "<<s.size()<<endl;*

*// // cout<<"here3"<<endl;*

*// DFS(g,0,100,t,edgesExplored,pathCost,s);*

*// cout<<"DEPTH FIRST SEARCH"<<endl;*

*// cout<<"timeTaken: "<<t<<endl;*

*// cout<<"edgesExplored: "<<edgesExplored<<endl;*

*// cout<<"pathCost: "<<pathCost<<endl;*

*// cout<<"No. of Nodes explored(out of 1000): "<<s.size()<<endl;*

*// IterativeDeepeningDriverFunction(g,0,100,t,edgesExplored,pathCost,s);*

*// cout<<"ITERATIVE DEEPENING SEARCH"<<endl;*

*// cout<<"timeTaken: "<<t<<endl;*

*// cout<<"edgesExplored: "<<edgesExplored<<endl;*

*// cout<<"pathCost: "<<pathCost<<endl;*

*// cout<<"No. of Nodes explored(out of 1000): "<<s.size()<<endl;*

    return 0;

}