#include<iostream>

#include<vector>

#include<list>

#include<map>

#include<ctime>

#include<cstdlib>

using namespace std;

//==============================================================================

// General definitions

//==============================================================================

// INFINIT is used to represent no edge/path between two nodes

const int INFINIT=999999;

// Overload operator << to print list<char> variables

ostream &operator<<(ostream &output, list<char> L)

{

list<char>::iterator i;

for(i=L.begin(); i != L.end(); ++i)

output << \*i << " ";

return output;

}

// Convert node numbers into chars (from 0..51 to A..Za..z)

inline char vertIntToChar(int n)

{

if (n<26)

return static\_cast<char>('A'+n);

else

return static\_cast<char>('a'+n-26);

}

//==============================================================================

// Node definitions

// Used to store information about nodes/edges in the adjacency list of a graph

// Adjacency lists is a list of Nodes (identified by numbers from 0 to 51)

// Each node contains a list of neighbors containing edge weight

//==============================================================================

typedef struct strNode Node;

struct strNode

{

int number;

int weight;

list<Node> edges;

};

//==============================================================================

// Graph Class

// Represent a Graph through an adjacency list

//==============================================================================

class Graph

{

public:

Graph();

Graph(int numVertices, int initialValue);

char get\_node\_value(int x);

void set\_node\_value(char x, char name);

int get\_edge\_value(char x, char y);

void set\_edge\_value(char x, char y, int value);

bool adjacent(char x, char y);

list<char> neighbors(char x);

int V();

int E();

list<char> vertices();

void show();

private:

int numV; // Number of nodes of the Graph

int numE; // Number of edges of the Graph

vector<char> nodeNames; // Map node numbers into node names

map<char, int> nodeNumbers; // Map node names into node numbers

list<Node> adjList; // Adjacency list representing the Graph

};

// Default constructor of Graph Class

// Create an empty graph

Graph::Graph()

{

numV = 0;

numE = 0;

adjList.clear();

}

// Constructor of Graph Class

// Initialize number of nodes

// Create adjacency list with all nodes and empty edge list

Graph::Graph(int numVertices, int initialValue=INFINIT)

{

// Create nodes and link it to default names (0..51 -> A..Za..z)

numV = numVertices;

numE = 0;

nodeNames.resize(numVertices);

for (int x=0; x<numV; ++x)

{

nodeNames[x] = vertIntToChar(x);

nodeNumbers[vertIntToChar(x)]=x;

}

// Create adjacency list with all nodes and empty edge list

adjList.clear();

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

{

Node newNode;

newNode.number = i;

newNode.weight = 0;

newNode.edges.clear();

adjList.push\_back(newNode);

}

}

// Return node name linked to node number x

char Graph::get\_node\_value(int x)

{

return nodeNames[x];

}

// Change node name (from 'x' to 'name')

void Graph::set\_node\_value(char x, char name)

{

int posX = nodeNumbers[x]; // Get the number of node 'x'

nodeNames[posX] = name; // Link node number to 'name'

nodeNumbers[name]=posX; // Link 'name' to node number

}

// Return edge weight between 'x' and 'y'

// Return INFINITY if edge doesn't exist

int Graph::get\_edge\_value(char x, char y)

{

for(list<Node>::iterator i=adjList.begin(); i != adjList.end(); ++i)

{

if ((\*i).number==nodeNumbers[x])

for(list<Node>::iterator j=(\*i).edges.begin(); j != (\*i).edges.end(); ++j)

{

if ((\*j).number==nodeNumbers[y])

return (\*j).weight;

}

}

return INFINIT;

}

// Set edge weight between 'x' and 'y'

void Graph::set\_edge\_value(char x, char y, int value)

{

bool found;

// Add 'y' in the list of 'x' neighbors (if doesn't exist)

// Set edge weight to value

for(list<Node>::iterator i=adjList.begin(); i != adjList.end(); ++i)

{

if ((\*i).number==nodeNumbers[x])

{

found = false;

for(list<Node>::iterator j=(\*i).edges.begin(); j != (\*i).edges.end(); ++j)

{

if ((\*j).number==nodeNumbers[y])

{

(\*j).weight=value;

found = true;

}

}

if (!found)

{

Node newNodeY;

newNodeY.number = nodeNumbers[y];

newNodeY.weight = value;

newNodeY.edges.clear();

(\*i).edges.push\_back(newNodeY);

}

}

}

// Add 'x' in the list of 'y' neighbors (if doesn't exist)

// Set edge weight to value

for(list<Node>::iterator i=adjList.begin(); i != adjList.end(); ++i)

{

if ((\*i).number==nodeNumbers[y])

{

found = false;

for(list<Node>::iterator j=(\*i).edges.begin(); j != (\*i).edges.end(); ++j)

{

if ((\*j).number==nodeNumbers[x])

{

(\*j).weight=value;

found = true;

}

}

if (!found)

{

Node newNodeX;

newNodeX.number = nodeNumbers[x];

newNodeX.weight = value;

newNodeX.edges.clear();

(\*i).edges.push\_back(newNodeX);

++numE; // Increment the number of edges in the graph

}

}

}

}

// Return true if 'x' and 'y' are neighbors and false otherwise

bool Graph::adjacent(char x, char y)

{

for(list<Node>::iterator i=adjList.begin(); i != adjList.end(); ++i)

{

if ((\*i).number==nodeNumbers[x])

{

for(list<Node>::iterator j=(\*i).edges.begin(); j != (\*i).edges.end(); ++j)

{

if ((\*j).number==nodeNumbers[y])

{

return true;

}

}

}

}

return false;

}

// Return a list<char> containing the list of neighbors of 'x'

list<char> Graph::neighbors(char x)

{

list<char> adjNodes;

for(list<Node>::iterator i=adjList.begin(); i != adjList.end(); ++i)

{

if ((\*i).number==nodeNumbers[x])

{

for(list<Node>::iterator j=(\*i).edges.begin(); j != (\*i).edges.end(); ++j)

{

adjNodes.push\_back(nodeNames[(\*j).number]);

}

}

}

return adjNodes;

}

// Return the number of nodes in the Graph

int Graph::V()

{

return numV;

}

// Return the number of edges in the Graph

int Graph::E()

{

return numE;

}

// Return a list<char> containing all nodes in the Graph

list<char> Graph::vertices()

{

list<char> nodes;

for(list<Node>::iterator i=adjList.begin(); i != adjList.end(); ++i)

{

nodes.push\_back(nodeNames[(\*i).number]);

}

return nodes;

}

// Print out adjacency list representing the Graph

void Graph::show()

{

cout << " ";

for(list<Node>::iterator i=adjList.begin(); i != adjList.end(); ++i)

cout << " " << nodeNames[(\*i).number];

cout << endl;

for(list<Node>::iterator i=adjList.begin(); i != adjList.end(); ++i)

{

cout << " " << nodeNames[(\*i).number];

int shift=0;

for(list<Node>::iterator j=(\*i).edges.begin(); j != (\*i).edges.end(); ++j)

{

int walk=(\*j).number-shift;

for(int k=0; k<walk; ++k)

{

cout << " -";

shift++;

}

cout << " " << (\*j).weight;

shift++;

}

while (shift<numV)

{

cout << " -";

shift++;

}

cout << endl;

}

}

//==============================================================================

// NodeInfo Definitions

// Used to store information about nodes, paths and min dists in priority queue

//==============================================================================

struct strNodeInfo

{

char nodeName; // Node name

int minDist; // Shortest path found to nodeName

char through; // Node that precede nodeName in the shortest path

};

typedef struct strNodeInfo NodeInfo;

// Compare NodeInfo by nodeName

bool compareNodeName(NodeInfo& n1, NodeInfo& n2)

{

if (n1.nodeName < n2.nodeName) return true;

return false;

}

// Compare NodeInfo by minDist

bool compareMinDist(NodeInfo& n1, NodeInfo& n2)

{

if (n1.minDist < n2.minDist) return true;

return false;

}

// Return true if two NodeInfo have the same nodeName and false otherwise

bool operator== (NodeInfo& n1, NodeInfo& n2)

{

if (n1.nodeName == n2.nodeName) return true;

return false;

}

//==============================================================================

// PriorityQueue Class

// Stores known information about node names, min distances and paths

// Ordered by min distances

//==============================================================================

class PriorityQueue {

public:

PriorityQueue();

void chgPriority(NodeInfo n);

void minPriority();

bool contains(NodeInfo n);

bool isBetter(NodeInfo n);

void insert(NodeInfo n);

NodeInfo top();

int size();

private:

list<NodeInfo> pq; // List of known nodes/paths ordered by minDist

};

// Constructor of PriorityQueue Class

// Creates an empty list of nodes

PriorityQueue::PriorityQueue()

{

pq.clear();

}

// Change information ('minDist' and 'through') of a node named 'n' in priority queue

void PriorityQueue::chgPriority(NodeInfo n)

{

for(list<NodeInfo>::iterator i=pq.begin(); i!=pq.end(); ++i)

if ((\*i) == n)

{

(\*i).minDist = n.minDist;

(\*i).through = n.through;

}

pq.sort(compareMinDist);

}

// Remove the node with lower minDist from priority queue

void PriorityQueue::minPriority()

{

if (! pq.empty())

{

pq.pop\_front();

}

}

// Returne true if there is a node named 'n' in priority queue and false otherwise

bool PriorityQueue::contains(NodeInfo n)

{

for(list<NodeInfo>::iterator i=pq.begin(); i!=pq.end(); ++i)

if ((\*i).nodeName == n.nodeName)

return true;

return false;

}

// Return true if node 'n' has a lower minDist than the node with the same name in the priority queue and false otherwise

bool PriorityQueue::isBetter(NodeInfo n)

{

for(list<NodeInfo>::iterator i=pq.begin(); i!=pq.end(); ++i)

if ((\*i).nodeName == n.nodeName)

if ((\*i).minDist > n.minDist)

return true;

return false;

}

// Insert node 'n' into priority queue

void PriorityQueue::insert(NodeInfo n)

{

pq.push\_back(n);

pq.sort(compareMinDist);

}

// Return the node with lower minDist in priority queue (without removing it from the queue))

NodeInfo PriorityQueue::top()

{

NodeInfo n = {' ',0};

if (! pq.empty())

{

list<NodeInfo>::iterator i=pq.begin();

n.nodeName = (\*i).nodeName;

n.minDist = (\*i).minDist;

n.through = (\*i).through;

}

return n;

}

// Return the number of elements in the priority queue

int PriorityQueue::size()

{

return pq.size();

}

//==============================================================================

// ShortestPath Class

// Implements Dijkstra's Algorithm to find shortest paths between two nodes

//==============================================================================

class ShortestPath

{

public:

ShortestPath();

ShortestPath(Graph g);

list<char> path(char u, char w);

int path\_size(char u, char w);

private:

Graph graph; // Graph used by Diajkstra's Algorithm

};

// Constructor of ShortestPath Class (do nothing)

ShortestPath::ShortestPath()

{

}

// Constructor of ShortestPath Class that stores Graph used by Dijkstra's Algorithm

ShortestPath::ShortestPath(Graph g)

{

graph = g;

}

// Return a list<char> containing the list of nodes in the shortest path between 'u' and 'w'

list<char> ShortestPath::path(char u, char w)

{

// Initialize candidates list with all nodes

list<char> candidates = graph.vertices(), desiredPath;

list<NodeInfo> minPaths;

PriorityQueue p;

NodeInfo lastSelected, n;

// Calculate shortest path from 'u' to 'w' (Dijkstra's Algorithm)

candidates.remove(u); // Remove 'u' from candidates list

lastSelected.nodeName = u; // Set 'u' as lastSelected

lastSelected.minDist = 0;

lastSelected.through = u;

minPaths.push\_back(lastSelected); // Add 'u' to minPath list

while ((!candidates.empty()) && (lastSelected.nodeName !=w))

{

// For each node in candidate list calculate the cost to reach that candidate through lastSelected

for(list<char>::iterator i=candidates.begin(); i != candidates.end(); ++i)

{

n.nodeName=\*i;

n.minDist=lastSelected.minDist+graph.get\_edge\_value(lastSelected.nodeName,\*i);

n.through=lastSelected.nodeName;

if (!p.contains(n)) // Add candidate to priority queue if doesn't exist

p.insert(n);

else

if (p.isBetter(n)) // Update candidate minDist in priority queue if a better path was found

p.chgPriority(n);

}

lastSelected = p.top(); // Select the candidate with minDist from priority queue

p.minPriority(); // Remove it from the priority queue

minPaths.push\_back(lastSelected); // Add the candidate with min distance to minPath list

candidates.remove(lastSelected.nodeName); // Remove it from candidates list

}

// Go backward from 'w' to 'u' adding nodes in that path to desiredPath list

lastSelected=minPaths.back();

desiredPath.push\_front(lastSelected.nodeName);

while(lastSelected.nodeName!=u)

{

for(list<NodeInfo>::iterator i=minPaths.begin(); i != minPaths.end(); ++i)

if ((\*i).nodeName==lastSelected.through)

{

lastSelected=(\*i);

desiredPath.push\_front(lastSelected.nodeName);

}

}

return desiredPath;

}

// Return the size of the shortest path between 'u' and 'w'

int ShortestPath::path\_size(char u, char w)

{

int pathCost=0;

list<char> sp;

char current,next;

// Calculate the shortest path from 'u' to 'w' and then sum up edge weights in this path

sp = path(u,w);

current=sp.front();

sp.pop\_front();

for(list<char>::iterator i=sp.begin(); i!=sp.end(); ++i)

{

next = (\*i);

pathCost += graph.get\_edge\_value(current,next);

current = next;

}

return pathCost;

}

//==============================================================================

// Monte Carlo Class

// Used to generate random graphs and run simulations

//==============================================================================

class MonteCarlo

{

public:

MonteCarlo();

Graph randomGraph(int vert, double density, int minDistEdge, int maxDistEdge);

void run(Graph g);

private:

};

// Constructor of MonteCarlo Class

// Initializes the seed of random number generator

MonteCarlo::MonteCarlo()

{

srand(time(NULL));

}

// Return a random Graph generated with number of nodes, density and edge weight range informed

Graph MonteCarlo::randomGraph(int numVert, double density, int minDistEdge, int maxDistEdge)

{

int randDistEdge;

char srcVert, dstVert;

Graph g(numVert);

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

for (int j=i+1; j<g.V(); ++j)

{

double p = ((static\_cast<double>(rand())) / RAND\_MAX); // Generate random probability

if (p < density) // If random probability is less than density, edge (i,j) will be set

{

randDistEdge = rand() % (maxDistEdge - minDistEdge) + minDistEdge; // Generate random edge weight

srcVert = vertIntToChar(i);

dstVert = vertIntToChar(j);

g.set\_edge\_value(srcVert,dstVert,randDistEdge);

}

}

return g;

}

// Run a simulation finding the shortest paths in a given graph

void MonteCarlo::run(Graph g)

{

static int turn=0;

cout << endl << "=== RUNNING SIMULATION No. " << ++turn << " ===" << endl;

// Print out graph information

double d = static\_cast<double>(g.E())/((static\_cast<double>(g.V())\*static\_cast<double>(g.V())-1)/2)\*100; // Calculate real density reached

cout << "Vertices: " << g.V() << endl;

cout << "Edges: " << g.E() << " (density: " << d << "%)" << endl;

cout << "Graph: " << endl;

g.show();

// Print out shortest path information

list<char> v = g.vertices();

cout << endl << "Vertices: " << v << endl;

int reachVert=0, sumPathSize=0, avgPathSize=0;

ShortestPath sp(g);

for (list<char>::iterator i=++v.begin(); i != v.end(); ++i)

{

char src = v.front();

char dst = (\*i);

list<char> p = sp.path(src,dst);

int ps = sp.path\_size(src,dst);

if (ps != INFINIT)

cout << "ShortestPath (" << src << " to " << dst << "): " << ps << " -> " << p << endl;

else

cout << "ShortestPath (" << src << " to " << dst << "): " << "\*\* UNREACHABLE \*\*" << endl;

if (ps!=INFINIT)

{

reachVert++; // Sum up reached nodes

sumPathSize += ps; // Sum up shortest paths found

}

}

// Calculate average shortest path and print it out

if (reachVert!=0)

avgPathSize = sumPathSize / reachVert;

else

avgPathSize = 0;

cout << endl << "AVG ShortestPath Size (reachVert: " << reachVert << " - sumPathSize: " << sumPathSize << "): " << avgPathSize << endl;

}

//==============================================================================

// Main Function

//==============================================================================

int main()

{

MonteCarlo simulation;

Graph g;

// Creates a graph with 50 nodes / density 20% and then run simulation

g = simulation.randomGraph(50,0.2,1,10);

simulation.run(g);

// Creates a graph with 50 nodes / density 40% and then run simulation

g = simulation.randomGraph(50,0.4,1,10);

simulation.run(g);

return 0;

}