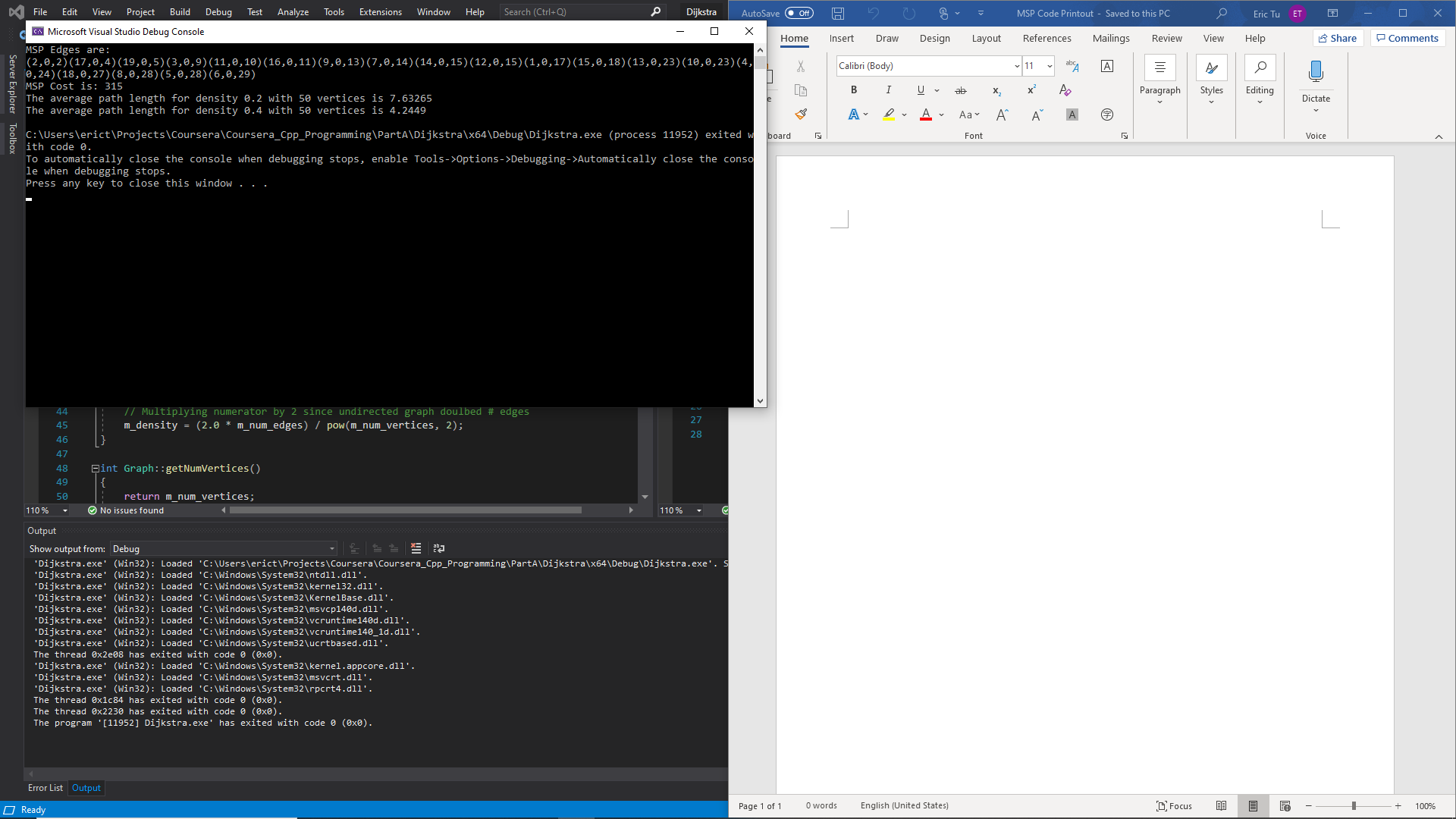
Code Printout:



Code: (Some residual Dijkstra)

(Main)

#include "Graph.h"

#include "GraphAlgos.h"

#include "PriorityQueue.h"

#include <iostream>

#include <deque>

using namespace std;

void calcAvePath(Graph graph)

{

GraphAlgos galg(graph);

double ave\_path\_length = 0.0;

int num\_paths = 0;

for (int i = 1; i < graph.getNumVertices(); i++)

{

// Check for path

pair<deque<int>, int> out = galg.findPath(0, i);

deque<int> path = out.first;

int cost = out.second;

if (cost != 10) {

int i = 0;

}

if(path[0] != -1)

{

// Incremental average

num\_paths++;

ave\_path\_length += (cost - ave\_path\_length)/num\_paths;

}

}

std::cout << "The average path length for density " << graph.getDensity() <<

" with " << graph.getNumVertices() << " vertices is " << ave\_path\_length << std::endl;

return;

}

void minSpanTree(Graph graph)

{

GraphAlgos algo(graph);

pair<deque<PathEdge>, int> out = algo.findPrimTree();

cout << "MSP Edges are: " << endl;

for (auto edge : out.first)

{

cout << edge.get();

}

cout << endl;

cout << "MSP Cost is: " << out.second << endl;

return;

}

int main()

{

// Create graph from file

Graph graph("nodeData.txt");

minSpanTree(graph);

// Create graphs

int num\_vertex = 50;

Graph graph20(num\_vertex);

Graph graph40(num\_vertex);

// Create one graph with density 0.2 and range 1-10

// Create other grpah with density 0.4 and range 1-10

graph20.createEdges(0.2, 10);

graph40.createEdges(0.4, 10);

calcAvePath(graph20);

calcAvePath(graph40);

return 0;

}

(Graph)

#pragma once

#include <vector>

#include <random>

#include <ctime>

#include <cassert>

#include <string>

using namespace std;

class Graph {

public:

Graph(int num\_vertices);

Graph(string file);

void createEdges(double density, int range);

int getNumVertices();

int getNumEdges();

double getDensity();

bool isAdjacent(int v1, int v2);

vector<int> getNeighbors(int vert);

void addEdge(int v1, int v2, int edge\_value = 1); // Does not overwrite existing edge

void setEdge(int v1, int v2, int edge\_value = 1);

int getEdgeValue(int v1, int v2);

void deleteEdge(int v1, int v2);

private:

int m\_num\_vertices;

int m\_num\_edges;

double m\_density;

vector<vector<int>> m\_conn\_matrix;

};

#include "Graph.h"

#include <limits>

#include <iostream>

#include <fstream>

#include <iterator>

#include <math.h>

Graph::Graph(int num\_vertices) : m\_num\_vertices(num\_vertices), m\_density(0.0), m\_num\_edges(0)

{

vector<int> init(num\_vertices, std::numeric\_limits<int>::max());

for (int i = 0; i < m\_num\_vertices; ++i)

{

m\_conn\_matrix.emplace\_back(init);

}

}

Graph::Graph(string filename) : m\_num\_edges(0)

{

ifstream file(filename);

istream\_iterator<int> start(file), end;

vector<int> nodes(start, end);

// create m\_conn\_matrix

m\_num\_vertices = \*nodes.begin();

vector<int> init(m\_num\_vertices, std::numeric\_limits<int>::max());

for (int i = 0; i < m\_num\_vertices; ++i)

{

m\_conn\_matrix.emplace\_back(init);

}

// Create node values

// Skip First value since it is the number of vertices

// Undirected graph

for (auto node = nodes.begin()+1; node < nodes.end(); node++)

{

int node1 = \*node;

int node2 = \*++node;

int cost = \*++node;

m\_conn\_matrix[node1][node2] = cost;

m\_conn\_matrix[node2][node1] = cost;

m\_num\_edges++;

}

// Multiplying numerator by 2 since undirected graph doulbed # edges

m\_density = (2.0 \* m\_num\_edges) / pow(m\_num\_vertices, 2);

}

int Graph::getNumVertices()

{

return m\_num\_vertices;

}

int Graph::getNumEdges()

{

return m\_num\_edges;

}

int Graph::getEdgeValue(int v1, int v2)

{

assert(v1 < m\_num\_vertices);

assert(v2 < m\_num\_vertices);

return m\_conn\_matrix[v1][v2];

}

bool Graph::isAdjacent(int v1, int v2)

{

assert(v1 < m\_num\_vertices);

assert(v2 < m\_num\_vertices);

return m\_conn\_matrix[1][v2] == std::numeric\_limits<int>::max() ? false : true;

}

// Assumes undirected graph

void Graph::addEdge(int v1, int v2, int edge\_value)

{

assert(v1 < m\_num\_vertices);

assert(v2 < m\_num\_vertices);

assert(edge\_value > 0);

if (m\_conn\_matrix[v1][v2] == std::numeric\_limits<int>::max())

{

m\_num\_edges++;

m\_conn\_matrix[v1][v2] = edge\_value;

m\_conn\_matrix[v2][v1] = edge\_value;

}

return;

}

// Assumes undirected graph

void Graph::setEdge(int v1, int v2, int edge\_value)

{

assert(v1 < m\_num\_vertices);

assert(v2 < m\_num\_vertices);

assert(edge\_value > 0);

if (m\_conn\_matrix[v1][v2] != std::numeric\_limits<int>::max())

{

m\_conn\_matrix[v1][v2] = edge\_value;

m\_conn\_matrix[v2][v1] = edge\_value;

}

}

// Assumes undirected graph

void Graph::deleteEdge(int v1, int v2)

{

assert(v1 < m\_num\_vertices);

assert(v2 < m\_num\_vertices);

if (m\_conn\_matrix[v1][v2] != std::numeric\_limits<int>::max())

{

m\_num\_edges--;

}

m\_conn\_matrix[v1][v2] = std::numeric\_limits<int>::max();

m\_conn\_matrix[v2][v1] = std::numeric\_limits<int>::max();

}

void Graph::createEdges(double density, int range)

{

m\_density = density;

uniform\_int\_distribution<int> edge\_len\_gen(1, range);

uniform\_real\_distribution<double> prob\_gen(0.0, 1.0);

default\_random\_engine e(time(0));

for (int i = 0; i < m\_num\_vertices; i++)

{

for (int j = i+1; j < m\_num\_vertices; j++)

{

if (prob\_gen(e) < density)

{

int len = edge\_len\_gen(e);

addEdge(i, j, edge\_len\_gen(e));

m\_num\_edges++;

}

}

}

return;

}

vector<int> Graph::getNeighbors(int vert)

{

vector<int> neighbors;

for (int i = 0; i < m\_num\_vertices; ++i)

{

if (m\_conn\_matrix[vert][i] != std::numeric\_limits<int>::max())

{

neighbors.push\_back(i);

}

}

return neighbors;

}

double Graph::getDensity()

{

return m\_density;

}

PriorityQueue

#pragma once

#include <queue>

#include <vector>

#include <tuple>

#include <string>

using namespace std;

class PathEdge

{

public:

PathEdge() : m\_vertex(-1), m\_cost(-1), m\_prev\_vertex(-1) {};

PathEdge(int vertex, int cost, int prev\_vertex) : m\_vertex(vertex), m\_cost(cost), m\_prev\_vertex(prev\_vertex) {};

int getCost() { return m\_cost; };

int getVertex() { return m\_vertex; };

int getPrevVertex() { return m\_prev\_vertex; };

string get() { return "(" + to\_string(m\_vertex) + "," +

to\_string(m\_prev\_vertex) + "," + to\_string(m\_cost) + ")"; }

private:

int m\_vertex;

int m\_cost;

int m\_prev\_vertex;

};

class Comparator

{

public:

bool operator() (PathEdge& p1, PathEdge& p2)

{

return (p1.getCost() > p2.getCost());

}

};

class PriorityQueue{

public:

PriorityQueue();

~PriorityQueue();

PathEdge useTop();

void insert(const PathEdge& p1);

void clear();

int size();

private:

priority\_queue<PathEdge, vector<PathEdge>, Comparator> m\_queue;

// key = node

// value = distance

};

#include "PriorityQueue.h"

PriorityQueue::PriorityQueue() {};

PriorityQueue::~PriorityQueue() {};

void PriorityQueue::insert(const PathEdge& p1)

{

m\_queue.emplace(p1);

}

PathEdge PriorityQueue::useTop()

{

if (!m\_queue.empty())

{

PathEdge ans = m\_queue.top();

m\_queue.pop();

return ans;

}

throw "Empty Priority Queue";

}

int PriorityQueue::size()

{

return m\_queue.size();

}

void PriorityQueue::clear()

{

m\_queue = priority\_queue<PathEdge, vector<PathEdge>, Comparator>();

return;

}

#pragma once

#include <vector>

#include <cassert>

#include "PriorityQueue.h"

#include "Graph.h"

#include <map>

#include <deque>

using namespace std;

class GraphAlgos {

public:

GraphAlgos(Graph graph);

pair<deque<int>, int> findPath(int v1, int v2);

pair<deque<PathEdge>, int> findPrimTree();

private:

void addToClosedSet(PathEdge node);

void addNeighborsToOpenSet(int vertex, int add\_edge\_val = 0);

void reset();

int last\_vertex;

// Key = Node; Value = Cost, PrevNode

map<int, pair<int, int>> m\_closed\_set;

PriorityQueue m\_open\_set;

Graph graph;

};

#include "GraphAlgos.h"

GraphAlgos::GraphAlgos(Graph graph) : graph(graph)

{

m\_open\_set = PriorityQueue();

}

pair<deque<int>, int> GraphAlgos::findPath(int v1, int v2)

{

reset();

assert(v1 < graph.getNumVertices());

assert(v2 < graph.getNumVertices());

last\_vertex = v1;

// Put origin in closed set

addToClosedSet(PathEdge(last\_vertex, 0, -1));

PathEdge next\_node;

while (true)

{

// Put neighbors in open set

int cur\_edge = m\_closed\_set.find(last\_vertex)->second.first;

addNeighborsToOpenSet(last\_vertex, cur\_edge);

// Bring in smallest open set cost to closed set

// In format (vertex, edge)

// If destination, stop

if (m\_open\_set.size())

{

next\_node = m\_open\_set.useTop();

}

else

{

// Path does not exist, return -1 shortest\_path and Infinite cost

deque<int> shortest\_path;

shortest\_path.emplace\_back(-1);

int shortest\_cost = std::numeric\_limits<int>::max();

return pair<deque<int>, int>(shortest\_path, shortest\_cost);

}

if (next\_node.getVertex() == v2)

{

// Find shortest cost by referencing closed set

addToClosedSet(next\_node);

int shortest\_cost = m\_closed\_set.find(last\_vertex)->second.first;

// Trace previous nodes recorded to find shortest path

deque<int> shortest\_path;

int vert = v2;

shortest\_path.push\_back(vert);

while (vert != v1)

{

int prev\_vertex = m\_closed\_set.find(vert)->second.second;

shortest\_path.push\_front(prev\_vertex);

vert = prev\_vertex;

}

return pair<deque<int>, int>(shortest\_path, shortest\_cost);

}

else if (m\_closed\_set.find(next\_node.getVertex()) == m\_closed\_set.end())

{

// No previous path to this node, add to closed set

addToClosedSet(next\_node);

}

else

{

// Skip since shortest path already found

}

}

// Put neighbors in open set, updating values if necessary

}

pair<deque<PathEdge>, int> GraphAlgos::findPrimTree()

{

reset();

deque<PathEdge> prim;

// Choose a node to start

last\_vertex = 0;

// Put origin in closed set

addToClosedSet(PathEdge(last\_vertex, 0, -1));

// Put neighbors in open set

addNeighborsToOpenSet(last\_vertex);

PathEdge next\_node;

while (m\_open\_set.size())

{

// Bring in smallest open set cost

next\_node = m\_open\_set.useTop();

// If not already in closed set

if (m\_closed\_set.find(next\_node.getVertex()) == m\_closed\_set.end())

{

addToClosedSet(next\_node);

prim.push\_back(next\_node);

// Check if found complete tree

if (m\_closed\_set.size() == graph.getNumVertices())

{

int total\_edge(0);

for (auto it = m\_closed\_set.begin(); it != m\_closed\_set.end(); it++)

{

total\_edge += it->second.first;

}

return pair<deque<PathEdge>, int>(prim, total\_edge);

}

}

}

// Tree does not exist, return -1

deque<PathEdge> dummy\_path;

dummy\_path.emplace\_back(PathEdge(-1, -1, -1));

return pair<deque<PathEdge>, int>(dummy\_path, -1);

}

void GraphAlgos::addToClosedSet(PathEdge next\_node)

{

int next\_vertex = next\_node.getVertex();

int next\_edge\_cost = next\_node.getCost();

int next\_prev\_vertex = next\_node.getPrevVertex();

auto val = pair<int, int>(next\_edge\_cost, next\_prev\_vertex);

m\_closed\_set.insert(map<int, pair<int, int>>::value\_type(next\_vertex, val));

last\_vertex = next\_vertex;

}

void GraphAlgos::addNeighborsToOpenSet(int vertex, int add\_edge\_val)

{

vector<int> nb = graph.getNeighbors(vertex);

for (auto v : nb)

{

if (m\_closed\_set.find(v) == m\_closed\_set.end())

{

m\_open\_set.insert(PathEdge(v, add\_edge\_val + graph.getEdgeValue(vertex, v), last\_vertex));

}

}

}

void GraphAlgos::reset()

{

m\_closed\_set.clear();

m\_open\_set.clear();

}