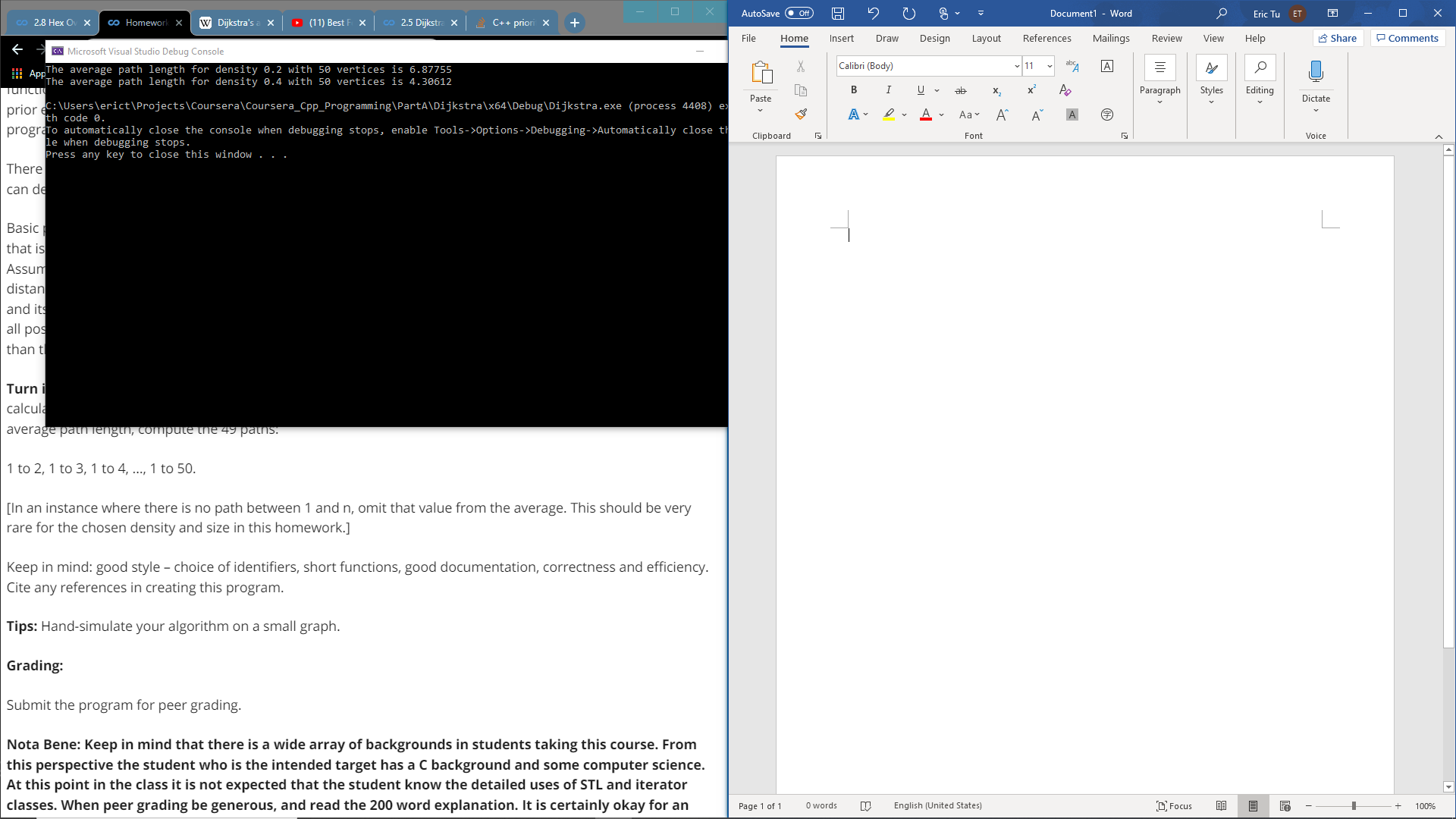
Code Printout:



I learned that a priority queue heap sorts during function execution and does not appear as sorted when debugging through the IDE. I also learned the details of implementing Dijkstra, that there is a need to keep track of the last index when adding path costs to the priority queue. Additionally, when there is no path between the specified nodes, dummy value or exception handles are needed to for this implementation of Dijkstra due to strong return typing constraints with c++.

(Coursera won’t let me send multiple files so copying code underneath)

Main.cpp

#include "Graph.h"

#include "ShortestPath.h"

#include "PriorityQueue.h"

#include <iostream>

#include <deque>

using namespace std;

void calcAvePath(Graph graph)

{

ShortestPath spath(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 = spath.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;

}

int main()

{

// 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;

}

Priority Queue

#pragma once

#include <queue>

#include <vector>

#include <tuple>

using namespace std;

class PathNode

{

public:

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

PathNode(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; };

private:

int m\_vertex;

int m\_cost;

int m\_prev\_vertex;

};

class Comparator

{

public:

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

{

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

}

};

class PriorityQueue{

public:

PriorityQueue();

~PriorityQueue();

PathNode useTop();

void insert(const PathNode& p1);

void clear();

int size();

private:

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

// key = node

// value = distance

};

#include "PriorityQueue.h"

PriorityQueue::PriorityQueue() {};

PriorityQueue::~PriorityQueue() {};

void PriorityQueue::insert(const PathNode& p1)

{

m\_queue.emplace(p1);

}

PathNode PriorityQueue::useTop()

{

if (!m\_queue.empty())

{

PathNode 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<PathNode, vector<PathNode>, Comparator>();

return;

}

Shortest Path algo

#pragma once

#include <vector>

#include <cassert>

#include "PriorityQueue.h"

#include "Graph.h"

#include <map>

#include <deque>

using namespace std;

class ShortestPath {

public:

ShortestPath(Graph graph);

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

int pathSize(int v1, int v2);

private:

void addToClosedSet(PathNode node);

void addNeighborsToOpenSet(int vertex);

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 "ShortestPath.h"

ShortestPath::ShortestPath(Graph graph) : graph(graph)

{

m\_open\_set = PriorityQueue();

}

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

{

reset();

assert(v1 < graph.getNumVertices());

assert(v2 < graph.getNumVertices());

last\_vertex = v1;

// Put origin in closed set

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

PathNode next\_node;

while (true)

{

// Put neighbors in open set

addNeighborsToOpenSet(last\_vertex);

// 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

}

void ShortestPath::addToClosedSet(PathNode 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 ShortestPath::addNeighborsToOpenSet(int vertex)

{

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

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

for (auto v : nb)

{

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

{

m\_open\_set.insert(PathNode(v, cur\_edge + graph.getEdgeValue(vertex, v), last\_vertex));

}

}

}

void ShortestPath::reset()

{

m\_closed\_set.clear();

m\_open\_set.clear();

}