What I learned:

From this task, I know how to create a class, class member function, class data member. Also, I learned how to distinguish the public member functions, and private member functions. In the Priority Queue class, I created two private member functions, since I only want to use the siftup() and sifdown() function when rearranging the priority queue, when inserting or removing the elements. I feel it is really powerful to use c++ for object orientated programming, though I need to implement the priority queue algorithm, in contrast, like python, there is existing priority queue data structure. I also know how to tackle a problem. First, we need to have a big picture about what we need to solve the problem. For example, what class we need, what function we need. Second, after having the skeleton, I fill the details, when filling the details, I will think about which basic and essential class is faster and better to implement the function, is array better, is vector better , should I use pointer , should I use reference? It might change the structure we initially designed, but it is fine. For example, at very first, I designed my PriorityQueueElement only has value and id, to indicate the current path cost, and vertex ID, and then I found it was not enough, in order to preserve the path, one more data needed to record the path. Then I added one more data member for the PriorityQueueElement.

Output:

```
density = 0.2, cost range = [1.0, 10.0]
node0 connects 49 nodes
average path cost = 7.28327
```

```
density = 0.4, cost range = [1.0, 10.0]
node0 connects 49 nodes
average path cost = 4.03429
```

Codes

Main.cpp

```
#include "pch.h"
#include <iostream>
#include <vector>
#include "Graph.h"
#include "ShortestPath.h"
using namespace std;

int main()
{
    int vn = 50;
    vector<double> density = { 0.2, 0.4 };
    vector<double> distance = { 1.0, 10.0 };


    Graph g(vn, density[0], distance);
    //g.niceshow();
    ShortestPath sp;

    double total = 0;
    int av = 0;
    for (int i = 1; i != vn; ++i) {
        /* print the shortest path out
        vector<int> mypath = sp.path(g, 0, i);
        cout << "0 --> ";
        for (auto i : mypath) {
            cout << i << " --> ";
        }
        cout << endl;
        */

        double x = sp.path_size(g, 0, i);
        if (x < INFINITY) {
            total += x;
            ++av;
        }
    }
    cout << "density = "<< density[1]<<", cost range = [1.0, 10.0]" << endl;
    cout << "node0 connects " << av << " nodes" << endl;
    cout << "average path cost = " << total / av << endl;

}
```

Graph.h

```
#pragma once

#include <vector>
using namespace std;
class Graph
{
public:
    Graph(int verticesNum, double density, vector<double> disRange);
    ~Graph();

    int getVerticeNum() const { return verticeNumber; };
    int getEdgeNum() const { return edgeNumber; };

    bool adjacent(int vertice1, int vertice2) const;
    vector<int> neighbors(int vertice) const;

    void addEdge(int vertice1, int vertice2, double distance);
    void deleteEdge(int vertice1, int vertice2);

    double getEdgeValue(int vertice1, int vertice2) const;
    void setEdgeValue(int vertice1, int vertice2, double distance);

    void niceshow();

private:
    vector<vector<double>> adjcentMatrix;
    int edgeNumber;
    int verticeNumber;

};
```

Graph.cpp

```
#include "pch.h"
#include "Graph.h"
#include <iostream>
#include <vector>
#include <stdlib.h>     /* srand, rand */
#include <time.h>       /* time */
//#include <stdio.h>      /* printf, scanf, puts, NULL */

using namespace std;

Graph::Graph(int vn, double density, vector<double> range)
{
    verticeNumber = vn;
    adjcentMatrix = vector<vector<double>> (vn, vector<double>(vn)); // this is vn*vn matrix

    //srand(time(0)); //use time to assign the random seed
    // create edges
    for (int i = 0; i != verticeNumber; ++i)
        for (int j = i + 1; j != verticeNumber; ++j) {
            double prob = rand() % 100 / 100.0;
            //random decide if it has edge
            if (prob < density){
                addEdge(i, j, range[0] + rand() % 100 / 100.0 * (range[1] - range[0]));//randome decide
the distance between two points
            }
        }
}

Graph::~Graph(){}
```

```
bool Graph::adjacent(int i, int j) const {
    return adjcentMatrix[i][j] > 0;
};

vector<int> Graph::neighbors(int i) const {
    vector<int> results;
    for (int j = 0; j != verticeNumber; j++)
        if (j != i && adjacent(i, j))
            results.push_back(j);
    return results;
};

void Graph::addEdge(int i, int j, double distance) {
    adjcentMatrix[i][j] = adjcentMatrix[j][i] = distance;
};

void Graph::deleteEdge(int i, int j) {
    adjcentMatrix[i][j] = adjcentMatrix[j][i] = 0;
};

double Graph::getEdgeValue(int i, int j) const {
    return adjcentMatrix[i][j];
};
void Graph::setEdgeValue(int i, int j, double distance) {
    addEdge(i,j,distance);
};


void Graph::niceshow() {
    for (int i = 0; i != verticeNumber; ++i) {
        for (int j = 0; j != verticeNumber; ++j) {
            printf("%4.2f   ", adjcentMatrix[i][j]);
            //cout << adjcentMatrix[i][j] << "    ";
        }
        cout << endl;
    }


};
```

PriorityQueue.h

```h
#pragma once

#include<vector>
using namespace std;

class PQElement {
public:
    PQElement(double v, int i) :value(v), id(i) {};
    PQElement(double v, int i, vector<int> p) :value(v), id(i), curpath(p) {};
    double value;
    int id;
    vector<int> curpath;
};

class PriorityQueue
{
public:
    PriorityQueue();
    ~PriorityQueue();
    PQElement minPriority();
    bool contains(PQElement);
    void insert(PQElement);
    PQElement top();
    int size();

private:
    vector<PQElement> pq;
    void siftup(int);
    void siftdown(int);

};
```

PriorityQueue.cpp

```cpp
#include "pch.h"
#include "PriorityQueue.h"
#include <iostream>
#include <algorithm>
using namespace std;

PriorityQueue::PriorityQueue(){}
PriorityQueue::~PriorityQueue(){}

PQElement PriorityQueue::top() {
    if (size()) return pq[0];
}

PQElement PriorityQueue::minPriority() {

    if (size()) {
        PQElement ans = top();
        //iter_swap(pq.begin(), pq.end());
        swap(pq[0], pq[size() - 1]);
        pq.pop_back();
        siftdown(0);
        return ans;
    };
};

void PriorityQueue::insert(PQElement element) {
    pq.push_back(element);
    siftup(size()-1);
};

bool PriorityQueue::contains(PQElement element) {
    for (auto x : pq)
        if (x.id==element.id) return true;
    return false;
};

int PriorityQueue::size() {
    return pq.size();
}

void PriorityQueue::siftup(int i) {
    int up = (i - 1) / 2;
    if (i && pq[up].value > pq[i].value)
    {
        swap(pq[up], pq[i]);
        siftup(up);
        }
};

void PriorityQueue::siftdown(int i) {
    if (2 * i + 1 < size() && pq[i].value > pq[2 * i + 1].value) {
        swap(pq[i], pq[2 * i + 1]);
        siftdown(i);
        siftdown(2 * i + 1);
    }
    if (2 * i + 2 < size() && pq[i].value > pq[2 * i + 2].value) {
        swap(pq[i], pq[2 * i + 2]);
        siftdown(i);
        siftdown(2*i+2);
    }

};
```

ShortestPath.h

```
#pragma once
#include "PriorityQueue.h"
#include "Graph.h"


class ShortestPath
{
public:
    ShortestPath();
    ~ShortestPath();
    tuple<bool, vector<int>, double> computepath(Graph g, int start, int end);
    vector<int> path(Graph g, int start, int end);
    double path_size(Graph g, int start, int end);


};
```

ShortestPath.cpp

```
#include "pch.h"
#include "ShortestPath.h"
#include "Graph.h"
#include "PriorityQueue.h"
#include <iostream>
#include <tuple>
#include <unordered_set>

using namespace std;

ShortestPath::ShortestPath() {};
ShortestPath::~ShortestPath(){}

tuple<bool, vector<int>, double> ShortestPath::computepath(Graph g, int start, int end) {
    PriorityQueue pq;
    pq.insert(PQElement(0,start));
    unordered_set<int> used;
    PQElement top = pq.top();

    while (pq.size()) {
        top = pq.minPriority();
        used.insert(top.id);
        if (top.id == end) break;
        vector<int> nei = g.neighbors(top.id);
        for (auto x : nei) {
            if (used.count(x)==0) {
                PQElement next(g.getEdgeValue(top.id, x) + top.value, x, top.curpath);
                next.curpath.push_back(x);
                pq.insert(next);
            }
        }
    }

    if (top.id==end) {
        return make_tuple(true, top.curpath, top.value);
    }
    else {
        top.curpath.clear();
        return make_tuple(false, top.curpath , INFINITY);
    }
};

vector<int> ShortestPath::path(Graph g, int start, int end) {
    return get<1>(computepath(g, start, end));
```

```
};

double ShortestPath::path_size(Graph g, int start, int end) {
    return get<2>(computepath(g,start, end));
};
```