Dijkstra’s Shortest Path

Priority Queue:

A priority queue is an Abstract Data Type that is a queue wherein each element has a certain priority, such that the elements with the higher priority are removed first. A queue is data structure that only allows elements that were inserted into the queue ahead of other elements to be removed before the removal of those other elements that were inserted into the queue after. Consider for example the check out line at a grocery store, the person at the front of the line will have their items checked out first, then followed by the person behind them, and so on, until the last person on the line is reached, this is the essence of the queue. A priority queue can be implemented using a **min heap** or a **max heap**, a min heap is a binary tree, wherein the lowest value is placed at the root node, and the left and right child nodes are always of a greater value that their parent node, a max heap is just simply the reverse of a min heap.

The ***graph class***:

I implemented the graph by using the adjacency list construct. An adjacency list can be thought of as array (outer array), wherein the elements of the array are themselves arrays (inner array), each index of the outer array correspond the nodes in the graph, and the elements of the inner arrays, that’s referenced by the indices of the outer array, correspond to the nodes that are neighbors to the node being referenced or any given node. For this assignment I used C++’s STL vector.

The ***dsPath class***:

This is the class that implements Dijkstra’s algorithm. The open set is implemented via a priority queue, utilizing the min heap data structure. The closed set is implemented via C++’s STL vector. And Dijkstra’s algorithm is implemented as follows: The source node is initially placed in the open set (*priorityQueue* class), then removed and placed in the closed set, all the neighbors of the source node are obtained and placed in the open set, along with their edge cost from the source node, the node with lowest edge cost is then removed from the open set and placed in closed set, a check is made to see if that node is the destination node, if not the neighbors of that node are obtained and placed in the open set, and again the node with the lowest edge cost it removed from the open set and placed in the closed set, if that last node is the destination node, then the solution is found, if not the process repeats, all the while keeping track of all the visited nodes, and until the open set is depleted. If the open set is depleted without finding the destination node, then there is no solution.

Source Code:

```
#include<iostream>
#include<vector>
#include<tuple>
#include<map>



using namespace std;

        // used to represent list of nodes, corresponding to a particular path
        // and the total edge cost of that path
class nodePath
{
        public:
                int edgeCountTotal;
                vector<int> nodeList;

                nodePath(int T = 0, vector<int> NL = vector<int>()):
                                        edgeCountTotal(T), nodeList(NL) {}
};

class priorityQueue
{
        vector<nodePath> pqArray;
        int size;

                // make subtree with root at given index priority queue compliant
        void heapify(int);
```

```
                // get parent node index of node at index i
        int parent(int i) { return (i-1)/2; }

                // get index of left child of node at index i
int left(int i) { return (2*i + 1); }

        // get index of right child of node at index i
int right(int i) { return (2*i + 2); }

        public:
                priorityQueue(int S = 0) { size = S; pqArray = vector<nodePath>(size); }

                        // removes the top element of the queue
                nodePath getMin();

                        // does the queue contain the value queue element "QE"
                        // if so return the index, if not return -1
                int contains(int EC);

                        // insert queue element "OE" into queue
                void insert(nodePath QE);

                        // returns the top element of the queue.
                nodePath top() { return pqArray[0]; }

                        // returns the number of queue elements
                int getSize() { return size; }

                vector<nodePath> getQueContents();
                bool isEmpty() { return size == 0 ? true: false; }

                bool notEmpty() { return size > 0 ? true: false; }

                        // emtpy out the Priority Queue
                void emptyPQ()
                {
                        pqArray.clear();
                        size = 0;
                }
};

        // calculates probablity of edges between vertices
inline double prob()
        { return static_cast<double>( rand() ) / static_cast<double>(RAND_MAX); }

        // calculates distance between edges
inline int getDistance(int drange) { return rand() % drange + 1; }

        // constuctor overloaded to create graph by specifying size, edge density,
        // and distance range:
        // S = size, density = edge density, maxd = distance range
graph::graph(int S, double density, int maxd)
{
        srand( time(nullptr) );

        size = S;

        vertices = vector< vertexElem >(size);
        int vID = 1;

        for(auto& V: vertices)
        {
                V = vertexElem(vID);
                vID++;
        }

        int d; // edge distance
        int i = 0;

        for(auto& V: vertices)
        {
                for(int j = i; j < size; j++)
                {
                        if (j != i)
                        {
                                if (prob() < density)
                                {
                                        d = getDistance(maxd);

                                        addEdge(i+1, j+1, d);
                                }
                        }
                }

                i++;
        }
}
```

```
graph::graph( vector<int> vrtx )
{
	size = 0;

	vertices = vector< vertexElem >();

	for(auto& V: vrtx)
	{
		vertices.push_back(vertexElem(V));
		size++;
	}
}

graph::graph( vectOfTuples vrtx )
{
	size = 0;

	for(auto& V: vrtx)
	{
		vertices.push_back( vertexElem( get<0>(V) ) ); size++;
		vertices.push_back( vertexElem( get<1>(V) ) ); size++;

		addEdge( get<0>(V), get<1>(V), get<2>(V) );
	}
}

	// total edge count
int graph::getEdges()
{
	int count = 0;
	for(int i = 0; i < size; i++)
	{
		for(auto& E: vertices[i].edgeList)
		{
			if(E.vertex > vertices[i].vertexID)
				count++;
		}
	}

	return count;
}

	// returns pointer to node if node "n" exists, returns nullptr if
	// node is nonexistent
vertElemItr graph::nodeExist(int n)
{
	for(auto itr = vertices.begin(); itr != vertices.end(); itr++)
	{
		if(itr->vertexID == n)
			return itr;
	}

	//return vertices.end();
	return static_cast< vertElemItr >(nullptr);
}

bool graph::nodeFound(int x)
{
	vertElemItr xptr = nodeExist(x);
```

```
        return xptr != static_cast< vertElemItr >(nullptr) ? true: false;
}

        // returns true if there's an edge from node x to y
bool graph::isAdjacent(int x, int y)
{
        vertElemItr xptr = nodeExist(x);
        vertElemItr yptr = nodeExist(y);

        bool xfound = xptr != static_cast< vertElemItr >(nullptr) ? true: false;
        bool yfound = yptr != static_cast< vertElemItr >(nullptr) ? true: false;

        if ( xfound  && yfound )
        {
                for(auto& E: xptr->edgeList)
                {
                        if (E.vertex == y)
                                return true;
                }

                return false;
        }
        else
                return false;
}

        // get all vertices connected to x
vector<int> graph::getNeighbors(int x)
{
	vertElemItr xptr = nodeExist(x);

	bool xfound = xptr != static_cast< vertElemItr >(nullptr) ? true: false;

	vector<int> neighbors = vector<int>(0);

	if ( xfound )
	{
		for(auto& E: xptr->edgeList)
			neighbors.push_back(E.vertex);
	}

	return neighbors;
}

	// adds edge between x & y, if one is currently nonexistent
	// returns true if edge was added, if not return false
bool graph::addEdge(int x, int y, int d)
{
	auto xptr = nodeExist(x);
	auto yptr = nodeExist(y);

	bool xfound = xptr != static_cast< vertElemItr >(nullptr) ? true: false;
	bool yfound = yptr != static_cast< vertElemItr >(nullptr) ? true: false;

	if ( xfound  && yfound )
	{
		if ( isAdjacent(x, y) && isAdjacent(y, x) )
			return false;
		else
```

```
                {
                        xptr->edgeList.push_back( edge(y, d) );
                        yptr->edgeList.push_back( edge(x, d) );

                        return true;
                }
        }
        else
                return false;
}

        // if there's an edge between x & y, delete it and return true
        // else return false
bool graph::deleteEdge(int x, int y)
{
        auto xptr = nodeExist(x);
        auto yptr = nodeExist(y);

        bool xfound = xptr != static_cast< vertElemItr >(nullptr) ? true: false;
        bool yfound = yptr != static_cast< vertElemItr >(nullptr) ? true: false;

        if ( xfound  && yfound )
        {
                if ( isAdjacent(x, y) && isAdjacent(y, x) )
                {
                        for (auto itr = xptr->edgeList.begin(); itr != xptr->edgeList.end(); itr++)
                        {
                                if (itr->vertex == y)
                                {
                                        xptr->edgeList.erase(itr);
```

```
                    break;
                }
            }

            for (auto itr = yptr->edgeList.begin(); itr != yptr->edgeList.end(); itr++)
            {
                if (itr->vertex == x)
                {
                    yptr->edgeList.erase(itr);
                    break;
                }
            }

            return true;
        }
        else
            return false;
    }
    else
        return false;
}



    // get edge weight/distance from x to y, if there is no edge return -1
int graph::getEdgeValue(int x, int y)
{
    auto xptr = nodeExist(x);
    auto yptr = nodeExist(y);

    bool xfound = xptr != static_cast< vertElemItr >(nullptr) ? true: false;
```

```
		bool yfound = yptr != static_cast< vertElemItr >(nullptr) ? true: false;

		if ( xfound  && yfound )
		{
			for(auto& V: xptr->edgeList)
			{
				if (V.vertex == y)
					return V.weight;
			}

			return -1;
		}
		else
			return -1;
}

		// set edge weight/distance between x & y
/* bool graph::setEdgeValue(int x, int y, int v)
{
		if ( isAdjacent(x, y) && isAdjacent(y, x) )
		{
			for(auto& E: vertices[x])
			{
				if (E.vertex == y)
				{
					E.weight = v;
					break;
				}
			}
```

```
                for(auto& E: vertices[y])
                {
                        if (E.vertex == x)
                        {
                                E.weight = v;
                                return true;
                        }
                }

                return true; // suppresses compiler warning
        }
        else
                return false;
}
 */




        // get the average path length of all the nodes connected to node "n"
double graph::avePathLength(int n)
{
        double total = 0;

        int eSize = 0;

        for(auto& V: vertices)
        {
                if (V.vertexID == n)
                {
                        eSize = V.edgeList.size();
                for(auto& E: V.edgeList)
                    total += static_cast<double>(E.weight);

                break;
            }
        }

        return total / static_cast<double>( eSize );
}
*****************PQ*************/
        // used to represent list of nodes, corresponding to a particular path
        // and the total edge cost of that path
class nodePath
{
        public:
            int edgeCountTotal;
            vector<int> nodeList;

            nodePath(int T = 0, vector<int> NL = vector<int>()):
                                    edgeCountTotal(T), nodeList(NL) {}
};

class priorityQueue
{
        vector<nodePath> pqArray;
        int size;

            // make subtree with root at given index priority queue compliant
        void heapify(int);
```

```
                // get parent node index of node at index i
        int parent(int i) { return (i-1)/2; }

                // get index of left child of node at index i
int left(int i) { return (2*i + 1); }

        // get index of right child of node at index i
int right(int i) { return (2*i + 2); }

        public:
                priorityQueue(int S = 0) { size = S; pqArray = vector<nodePath>(size); }

                        // removes the top element of the queue
                nodePath getMin();

                        // does the queue contain the value queue element "QE"
                        // if so return the index, if not return -1
                int contains(int EC);

                        // insert queue element "OE" into queue
                void insert(nodePath QE);

                        // returns the top element of the queue.
                nodePath top() { return pqArray[0]; }

                        // returns the number of queue elements
                int getSize() { return size; }

                vector<nodePath> getQueContents();
                bool isEmpty() { return size == 0 ? true: false; }

                bool notEmpty() { return size > 0 ? true: false; }

                        // emtpy out the Priority Queue
                void emptyPQ()
                {
                        pqArray.clear();
                        size = 0;
                }
};

void PQswap(nodePath& x, nodePath& y)
{
   nodePath temp = x;
   x = y;
   y = temp;
}

        // make subtree with root at given index priority queue compliant
void priorityQueue::heapify(int i)
{
        int l = left(i);
        int r = right(i);
        int smallest = i;

        if (l < size && pqArray[l].edgeCountTotal < pqArray[i].edgeCountTotal)
        smallest = l;
```

```
	if (r < size && pqArray[r].edgeCountTotal < pqArray[smallest].edgeCountTotal)
	smallest = r;

	if (smallest != i)
	{
	PQswap(pqArray[i], pqArray[smallest]);
	heapify(smallest);
	}
}

	// removes the top element of the queue
nodePath priorityQueue::getMin()
{
	if (size <= 0)
	return nodePath(0, vector<int>{-1});

	nodePath root;

	if (size == 1)
	{
	size--;

		root = pqArray[0];
		pqArray.pop_back();

	return root;
	}

	root = pqArray[0];
```

```
        pqArray[0] = pqArray[size-1];

        size--;

        heapify(0);

        pqArray.pop_back();

        return root;
}



        // does the queue contain the value queue element "QE"
        // if so return the index, if not return -1
int priorityQueue::contains(int EC)
{
        for(int i = 0; i < size; i++)
        {
                if (pqArray[i].edgeCountTotal == EC)
                        return i;
        }

        return -1;
}



        // insert queue element "OE" into queue
void priorityQueue::insert(nodePath QE)
{
        size++;
        int i = size - 1;

        pqArray.push_back(QE);

                // make priority queue compliant
  while (i != 0 && pqArray[parent(i)].edgeCountTotal > pqArray[i].edgeCountTotal)
  {
    PQswap(pqArray[i], pqArray[parent(i)]);
    i = parent(i);
  }
}

vector<nodePath> priorityQueue::getQueContents()
{
        vector<nodePath> elements;

        for(auto& E: pqArray)
                elements.push_back(E);

        return elements;
}

/***************dsPath*****************/

class dsPath
{
        priorityQueue openSet;
        graph spGraph;
        vector<nodePath> closedSet;
```

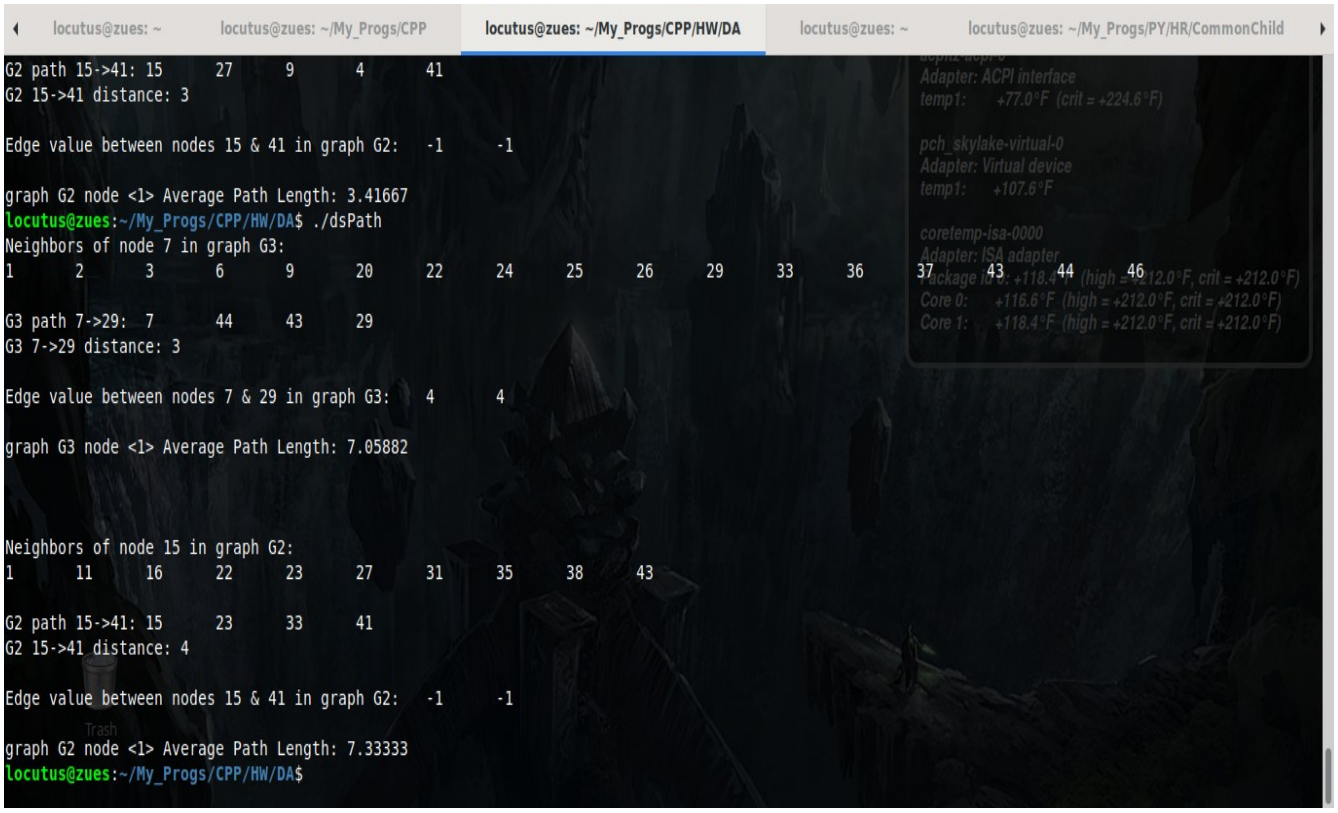
```
	map<int, int>  nodesVisited; // track visited nodes

		// Dijkstra's Algorithm is implemented here
	nodePath getNodePath(int sN, int dN);

	public:
				// constructor takes a graph object as input, and appropriately
				// initializes the open and closed set, nodes visited tracker
		dsPath(graph G = graph(),
					priorityQueue OS = priorityQueue(),
					vector<nodePath> CS = vector<nodePath>(),
					map<int, int> NV = map<int, int>{} ):
					spGraph(G), openSet(OS), closedSet(CS), nodesVisited(NV) {}

				// find shortest path between from node u to node w and returns the
				// sequence of vertices representing shortest path between them
		vector<int> getPath(int u, int w)
				{ nodePath snP = getNodePath(u, w); return snP.nodeList; }

				// returns the path cost associated with the shortest path between
				// nodes u and w
		int getPathSize(int u, int w)
				{ nodePath snP = getNodePath(u, w); return snP.edgeCountTotal; }
};




nodePath dsPath::getNodePath( int sN, int dN)
{
	if( !( spGraph.nodeFound(sN) && spGraph.nodeFound(dN) ) )
```

```
        return nodePath(0, vector<int>{-1});

if (sN == dN)
        return nodePath(0, vector<int>{dN});

        // initialize open and closed sets, to empty
openSet.emptyPQ();
closedSet.clear();

        //initialize nodes visited tracker
nodesVisited.clear();

openSet.insert( nodePath(0, vector<int>{sN}) );

nodePath np = openSet.getMin();

closedSet.push_back(np);

int cn, ev, tempEV;

vector<int> tempNP;

nodePath LP; // last path added to closed set

do {
        cn = np.nodeList.back(); // current node to be examined

        for(auto& N: spGraph.getNeighbors( cn ) )
        {
                if (cn == N)
                        continue;

                tempEV = np.edgeCountTotal;
                tempNP = np.nodeList;

                ev = spGraph.getEdgeValue(cn, N);

                //cout << ev << endl;

                tempEV += ev;

                tempNP.push_back(N);

                        // add new path to open set if the node is leads to isn't
                        // already ready visited, if it's already been visited, add it
                        // only if it's better path than one already checked/found
                if (nodesVisited[N] == 0)
                {
                        openSet.insert( nodePath(tempEV, tempNP) );
                        nodesVisited[N] = tempEV;
                }
                else
                {
                        if (tempEV < nodesVisited[N])
                        {
                                openSet.insert( nodePath(tempEV, tempNP) );
                                nodesVisited[N] = tempEV;
                        }
                }
```

```
                //openSet.insert( nodePath(tempEV, tempNP) );
            }

            np = openSet.getMin(); // get shortest path currently in open set

            closedSet.push_back(np);

            LP = closedSet.back(); // get most recent path added to closed set

                // if final node in the most recent path added to closed set is
                // "dN", the destination node, then shortest path has been found
            if ( LP.nodeList.back() == dN )
                return LP;

        } while( openSet.notEmpty() );

            // return empty path list if node "dN" is not reachable from node "sN"
        return nodePath(0, vector<int>{-1});
}

/***********main************/




int main(int argc, char** argv)
{
        graph G1(100, 0.579, 33), G2(50, 0.2, 10), G3(50, 0.4, 10);
        graph G4(35, 0.64, 18);
```

```
cout << "Neighbors of node 7 in graph G3:" << endl;

for(auto& N: G3.getNeighbors(7) )
        cout << N << "\t";

cout << endl;

        // instance of dsPath class initialized with graph G3
dsPath g3SP(G3);

auto p7to29 = g3SP.getPath(7, 29);

cout << "\nG3 path 7->29:\t";

for(auto& N: p7to29)
        cout << N << '\t';

cout << "\nG3 7->29 distance: " << g3SP.getPathSize(7, 29) << endl;

cout << "\nEdge value between nodes 7 & 29 in graph G3:\t";

cout << G3.getEdgeValue(7, 29) << '\t' << G3.getEdgeValue(29, 7) << endl;

cout << "\ngraph G3 node <1> Average Path Length: " << G3.avePathLength(1);
cout << endl;

cout << "\n\n\nNeighbors of node 15 in graph G2:" << endl;

for(auto& N: G2.getNeighbors(15) )
```

```
        cout << N << "\t";

    cout << endl;

        // instance of dsPath class initialized with graph G2
    dsPath g2SP(G2);

    auto p15to41 = g2SP.getPath(15, 41);

    cout << "\nG2 path 15->41:\t";

    for(auto& N: p15to41)
        cout << N << '\t';

    cout << "\nG2 15->41 distance: " << g3SP.getPathSize(15, 41) << endl;

    cout << "\nEdge value between nodes 15 & 41 in graph G2:\t";

    cout << G2.getEdgeValue(15, 41) << '\t' << G2.getEdgeValue(41, 15) << endl;

    cout << "\ngraph G2 node <1> Average Path Length: " << G2.avePathLength(1);
    cout << endl;

    return 0;
}
```



locutus@zues: ~
locutus@zues: ~/My_Progs/CPP
locutus@zues: ~/My_Progs/CPP/HW/DA
locutus@zues: ~
locutus@zues: ~/My_Progs/PY/HR/CommonChild

```
G2 path 15->41: 15      27      9       4       41
G2 15->41 distance: 3

Edge value between nodes 15 & 41 in graph G2:   -1      -1

graph G2 node <1> Average Path Length: 3.41667
locutus@zues:~/My_Progs/CPP/HW/DA$ ./dsPath
Neighbors of node 7 in graph G3:
1       2       3       6       9       20      22      24      25      26      29      33      36      37      43      44      46

G3 path 7->29:  7       44      43      29
G3 7->29 distance: 3

Edge value between nodes 7 & 29 in graph G3:    4       4

graph G3 node <1> Average Path Length: 7.05882



Neighbors of node 15 in graph G2:
1       11      16      22      23      27      31      35      38      43

G2 path 15->41: 15      23      33      41
G2 15->41 distance: 4

Edge value between nodes 15 & 41 in graph G2:   -1      -1

graph G2 node <1> Average Path Length: 7.33333
locutus@zues:~/My_Progs/CPP/HW/DA$
```