Lab 12

Due No Due Date **Points** 1

You are suggested to use the teaching servers burrow.soic.indiana.edu or hulk.soic.indiana.edu or tank.soic.indiana.edu for practicing C programs.

Lab 12: Dijkstra's algorithm

For this lab, all students must have a good grasp on priority queue and graph construction.

Shortest Path with weighted edge

We know about BFS which can be used to find the shortest distance between two nodes of a graph only if all the edges have the exact same cost. In case of weighted edged graph, the edges may have different cost. We will use Dijkstra's algorithm to solve the problem to find single source shortest path in a weighted graph. Note that, this algorithm only works for weight ≥ 0. In case of negative weight, we will need to find alternative algorithm.

An easy implementation of Dijkstra's algorithm is given below:

```
DIJKSTRA(G, s)
        DISTANCE[0...n-1] = INFINITY
        DISTANCE[s] = 0
        sourceNode.vertex = s
        sourceNode.dist = 0
        PQ.INSERT(sourceNode)  // PQ is a priority queue
        WHILE PQ is not empty
                u = PQ.EXTRACT_MIN_DIST()
                for each vertex v adjacent to u
                        IF DISTANCE[v] > DISTANCE[u] + edgeCost(u,v)
                                DISTANCE[v] = DISTANCE[u] + edgeCost(u,v)
                                // create a new node to put into PQ
                                node.vertex = v
                                node.dist = DISTANCE[v]
                                PQ.INSERT(node)
```

At the end DISTANCE[0...n-1] holds the shortest path from s to each vertices.

Representing weighted graph data structure with vector

We already used data structure to hold edges for DFS where the edges were not weighted. In this exercise, we will use weighted edge, so the graph representation would be slightly different.

Consider the directional graph below for 5 nodes (0,1,2,3,4) and 10 edges {u,v,d} indicates that there is an edge from u to v with cost d.

{0, 1, 10}
{0, 3, 5}

{1, 2, 1}
{1, 3, 2}
{2, 4, 4}
{3, 1, 3}
{3, 2, 9}
{3, 4, 2}
{4, 0, 7}
{4, 2, 6}

Rather than using vector<int> we will use vector<Node> such as vector<Node> graph[100], where graph[3] holds only 3 values, the 3 adjacent nodes (1,2,4) and the weight cost (4,3,9).

Here is a sample program that uses vector to hold all edges of the weighted graph and shows the edges and weights of each nodes to the output.

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <queue>
#include <vector>

using namespace std;

typedef struct node_struct
{
        int vertex;
        int dist;
} Node;

typedef struct edge_struct
{
        int u1;
        int u2;
        int cost;
} Edge;

vector<Node> graph[50];

int main()
{
        const int INF = 9999999;
        int n = 5;
        int u, v, d;
        int u1, u2, dist, cost;

        Edge edgeList[10] = {
                                {0, 1, 10},
                                {0, 3, 5},
                                {1, 2, 1},
                                {1, 3, 2},
                                {2, 4, 4},
                                {3, 1, 3},
                                {3, 2, 9},
                                {3, 4, 2},
                                {4, 0, 7},
                                {4, 2, 6}
                        };
```

```
        for(int i = 0; i < 10; ++i)
        {
                u1 = edgeList[i].u1;
                u2 = edgeList[i].u2;
                cost = edgeList[i].cost;

                Node t;
                t.vertex = u2;
                t.dist = cost;
                graph[u1].push_back(t);
        }

        for(int i = 0; i < n; ++i)
        {
                u = i;
                cout << u << ": ";
                for(vector<Node>::iterator vi = graph[u].begin(); vi != graph[u].end(); ++vi)
                {
                        v = vi->vertex;
                        cost = vi->dist;
                        cout << "[" << v << "," << cost << "] ";
                }
                cout << endl;
        }

        return 0;
}
```

We will need a priority queue to hold the nodes. We can initialize the Distance[] array and priority queue the following way:

```
std::priority_queue<Node, vector<Node>, NodeCompare> PQ;

for(int i = 0; i < n; ++i)
        distance[i] = INF;

distance[0] = 0; // Assuming 0 is the source node

Node node;
node.vertex = 0; // Assuming 0 is the source node
node.dist = 0;
PQ.push(node);
```