EECE 7205: Introduction of Computer Engineering

Assignment 5

Jiayun Xin

NUID: 001563582

College of Engineering

Northeastern University Boston, Massachusetts

Fall, 2021

Q1

Result:

A picture containing text

Description automatically generated

The screenshot above shows the shortest distance from the start point for each vertex.

Code:

#include <iostream>

#include <limits.h>

#include <array>

#include <vector>

#include <queue>

using namespace std;

// An adjacency list. Each vec[i] holds all the adjacent nodes of i

// The first int is the vertex of the adjacent nodes, the second int is the edge weight

vector< vector<pair<int, int> > > adjacency\_list()

{

// adjacency list is stored in vec

vector< vector<pair<int, int> > > vec;

const int n = 7;

for(int i = 0; i < n; i++)

{

vector<pair<int, int> > row;

vec.push\_back(row);

}

// add edges into the adjacency list

vec[0].push\_back(make\_pair(1, 2));

vec[0].push\_back(make\_pair(2, 3));

vec[1].push\_back(make\_pair(0, 2));

vec[1].push\_back(make\_pair(5, 1));

vec[2].push\_back(make\_pair(0, 3));

vec[2].push\_back(make\_pair(5, 2));

vec[3].push\_back(make\_pair(1, 4));

vec[3].push\_back(make\_pair(4, 1));

vec[3].push\_back(make\_pair(6, 2));

vec[4].push\_back(make\_pair(3, 1));

vec[4].push\_back(make\_pair(5, 2));

vec[4].push\_back(make\_pair(6, 1));

vec[5].push\_back(make\_pair(1, 1));

vec[5].push\_back(make\_pair(2, 2));

vec[5].push\_back(make\_pair(4, 2));

vec[5].push\_back(make\_pair(6, 2));

vec[6].push\_back(make\_pair(3, 2));

vec[6].push\_back(make\_pair(4, 1));

vec[6].push\_back(make\_pair(5, 2));

//return the graph

return vec;

}

// dijiksra finds all shortest paths from "start" to all other vertices

vector<int> dijiksra(vector< vector<pair<int, int> > > &vec, int &start)

{

vector<int> length;

int n = vec.size();

for(int i = 0; i < n; i++)

{

length.push\_back(1000000007); // Define "infinity" as necessary by constraints

}

priority\_queue<pair<int, int>, vector< pair<int, int> >, greater<pair<int, int> > > pq;

pq.push(make\_pair(start, 0));

length[start] = 0;

while(pq.empty() == false)

{

int u = pq.top().first;

pq.pop();

for(int i = 0; i < vec[u].size(); i++)

{

int v = vec[u][i].first;

int weight = vec[u][i].second;

if(length[v] > length[u] + weight)

{

length[v] = length[u] + weight;

pq.push(make\_pair(v, length[v]));

}

}

}

return length;

}

void minimal\_length(vector<int> &length, int &start)

{

cout <<"Vertex \t Distance from Source" << endl;

for (int i = 0; i < length.size(); i++)

cout << i << " \t\t\t\t" <<length[i]<< endl;

}

int main()

{

// Construct the adjacency list

vector< vector<pair<int, int> > > vec = adjacency\_list();

int node = 0;

vector<int> length = dijiksra(vec, node);

// Print the list.

minimal\_length(length, node);

return 0;

}

Q2

Result:

A picture containing text

Description automatically generated

The screenshot above shows the lowest weighted distance from the start point for each vertex.

Code:

#include <iostream>

#include <limits.h>

struct Edge {

int src, dest, weight;

};

struct Graph {

// V-> Number of vertices, E-> Number of edges

int V, E;

// graph is represented as an array of edges

struct Edge\* edge;

};

// Creates a graph with V vertices and E edges

struct Graph\* createGraph(int V, int E)

{

struct Graph\* graph = new Graph;

graph->V = V;

graph->E = E;

graph->edge = new Edge[E];

return graph;

}

// Print the solution

void printArr(int dist[], int n)

{

printf("Vertex Distance from Source\n");

for (int i = 0; i < n; ++i)

printf("%d \t\t %d\n", i, dist[i]);

}

// The main function that finds shortest distances from src

// to all other vertices using Bellman-Ford algorithm.

void BellmanFord(struct Graph\* graph, int src)

{

int V = graph->V;

int E = graph->E;

int dist[V];

// Initialize distances from src to all other vertices as INFINITE

for (int i = 0; i < V; i++)

dist[i] = INT\_MAX;

dist[src] = 0;

// Relax all edges |V| - 1 times

for (int i = 1; i <= V - 1; i++) {

for (int j = 0; j < E; j++) {

int u = graph->edge[j].src;

int v = graph->edge[j].dest;

int weight = graph->edge[j].weight;

if (dist[u] != INT\_MAX

&& dist[u] + weight < dist[v])

dist[v] = dist[u] + weight;

}

}

// Step 3: check for negative-weight cycles

for (int i = 0; i < E; i++) {

int u = graph->edge[i].src;

int v = graph->edge[i].dest;

int weight = graph->edge[i].weight;

if (dist[u] != INT\_MAX

&& dist[u] + weight < dist[v]) {

printf("Graph contains negative weight cycle");

return; // If negative cycle is detected, simply return

}

}

printArr(dist, V);

return;

}

int main()

{

/\* Let us create the graph given in above example \*/

int V = 5; // Number of vertices in graph

int E = 8; // Number of edges in graph

struct Graph\* graph = createGraph(V, E);

// add edge 0-1 (or A-B in above figure)

graph->edge[0].src = 0;

graph->edge[0].dest = 1;

graph->edge[0].weight = -1;

// add edge 0-2 (or A-C in above figure)

graph->edge[1].src = 0;

graph->edge[1].dest = 2;

graph->edge[1].weight = 4;

// add edge 1-2 (or B-C in above figure)

graph->edge[2].src = 1;

graph->edge[2].dest = 2;

graph->edge[2].weight = 3;

// add edge 1-3 (or B-D in above figure)

graph->edge[3].src = 1;

graph->edge[3].dest = 3;

graph->edge[3].weight = 2;

// add edge 1-4 (or B-E in above figure)

graph->edge[4].src = 1;

graph->edge[4].dest = 4;

graph->edge[4].weight = 2;

// add edge 3-2 (or D-C in above figure)

graph->edge[5].src = 3;

graph->edge[5].dest = 2;

graph->edge[5].weight = 5;

// add edge 3-1 (or D-B in above figure)

graph->edge[6].src = 3;

graph->edge[6].dest = 1;

graph->edge[6].weight = 1;

// add edge 4-3 (or E-D in above figure)

graph->edge[7].src = 4;

graph->edge[7].dest = 3;

graph->edge[7].weight = -3;

BellmanFord(graph, 0);

return 0;

}