**DAA PRACTICAL**

**Name:Parth Borse Roll No.: 07**

**BELLMAN FORD PROGRAM**

#include <iostream>

#include <limits.h>

#include <stdlib.h>

using namespace std;

#define max 100

#define infinity INT\_MAX

#define nil -1

int n; // Number of vertices

int adj[max][max]; // Adjacency matrix representing the graph

int predecessor[max]; // Predecessor array for shortest path

int pathlength[max]; // Array to store shortest path length

int ispresent\_in\_queue[max]; // Array to track if vertex is present in the queue

int front, rear; // Front and rear indices for queue

int queue[max]; // Queue array for Bellman-Ford algorithm

// Function prototypes

void initialize\_queue();

void insert\_queue(int u);

int delete\_queue();

int isempty\_queue();

void create\_graph();

void findpath(int s, int v);

int bellmanford(int s);

int main() {

int flag, s, v;

create\_graph();

cout << "Enter source vertex: ";

cin >> s;

flag = bellmanford(s);

if (flag == -1) {

cout << "Error: Negative cycle in graph" << endl;

exit(1);

}

while (1) {

cout << "Enter destination vertex (-1 to quit): ";

cin >> v;

if (v == -1)

break;

if (v < 0 || v >= n)

cout << "This vertex does not exist";

else if (v == s)

cout << "Source and destination vertices are same" << endl;

else if (pathlength[v] == infinity)

cout << "There is no path from source to destination vertex" << endl;

else

findpath(s, v); // Find and display shortest path

}

return 0;

}

// Function to find shortest path and display it

void findpath(int s, int v) {

int i, u;

int path[max];

int shortdist = 0;

int count = 0;

while (v != s) {

count++;

path[count] = v;

u = predecessor[v];

shortdist += adj[u][v];

v = u;

}

count++;

path[count] = s;

cout << "Shortest path is: ";

for (i = count; i >= 1; i--)

cout << path[i];

cout<<endl;

cout << "Shortest distance is: " << shortdist << endl;

}

// Function to implement Bellman-Ford algorithm

int bellmanford(int s) {

int i, current;

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

predecessor[i] = nil;

pathlength[i] = infinity;

ispresent\_in\_queue[i] = false;

}

initialize\_queue(); // Initialize queue for BFS traversal

pathlength[s] = 0;

insert\_queue(s);

ispresent\_in\_queue[s] = true;

// Main loop of Bellman-Ford algorithm

while (!isempty\_queue()) {

current = delete\_queue();

ispresent\_in\_queue[current] = false;

// Relaxation step

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

if (adj[current][i] != 0 && pathlength[i] > pathlength[current] + adj[current][i]) {

pathlength[i] = pathlength[current] + adj[current][i];

predecessor[i] = current;

if (!ispresent\_in\_queue[i]) {

insert\_queue(i); // Insert vertex into queue if not already present

ispresent\_in\_queue[i] = true;

}

}

}

}

// Check for negative cycles

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

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

if (adj[i][j] != 0 && pathlength[j] > pathlength[i] + adj[i][j]) {

return -1;

}

}

}

return 1; // No negative cycle detected

}

// Function to initialize the queue

void initialize\_queue() {

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

queue[i] = 0;

rear = -1;

front = -1;

}

// Function to check if queue is empty

int isempty\_queue() {

if (front == -1 || front > rear)

return 1;

else

return 0;

}

// Function to insert an element into the queue

void insert\_queue(int added\_item) {

if (rear == max - 1) {

cout << "Queue is full" << endl;

exit(1);

} else {

if (front == -1)

front = 0;

rear = rear + 1;

queue[rear] = added\_item;

}

}

// Function to delete an element from the queue

int delete\_queue() {

int d;

if (front == -1 || front > rear) {

cout << "Queue is empty" << endl;

exit(1);

} else {

d = queue[front];

front = front + 1;

}

return d;

}

// Function to create the graph

void create\_graph() {

int m\_e, origin, destin, wt;

cout << "Enter the number of vertices: ";

cin >> n;

m\_e = n \* (n - 1);

for (int i = 1; i <= m\_e; i++) {

cout << "Enter edges (-1 -1 to quit) " << i << ":";

cin >> origin >> destin;

if ((origin == -1) && (destin == -1))

break;

cout << "Enter weight for this edge: ";

cin >> wt; // Input weight of the edge

if (origin >= n || destin >= n || origin < 0 || destin < 0) {

cout << "Invalid edge!" << endl;

} else

adj[origin][destin] = wt;

}

}

//output:

