

22f-3104

DSA\_Lab 15



December 2, 2023

fast cfd campus

**Task no 1:**

#include <iostream>

using namespace std;

const int MAX\_NODES = 100;

// Depth-First Search for Directed Graph

bool dfsDirected(int node, int graph[MAX\_NODES][MAX\_NODES], bool visited[MAX\_NODES], bool stack[MAX\_NODES], int n) {

visited[node] = true;

stack[node] = true;

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

// If the neighbor is not visited, recursively call DFS

if (graph[node][neighbor] && !visited[neighbor]) {

if (dfsDirected(neighbor, graph, visited, stack, n))

return true;

}

else if (graph[node][neighbor] && stack[neighbor]) {

return true;

}

}

stack[node] = false;

return false;

}

bool hasCycleDirected(int graph[MAX\_NODES][MAX\_NODES], int n) {

bool visited[MAX\_NODES];

bool stack[MAX\_NODES];

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

visited[i] = false;

stack[i] = false;

}

// Call DFS for each unvisited node

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

if (!visited[node]) {

if (dfsDirected(node, graph, visited, stack, n))

return true;

}

}

// No cycle found in the directed graph

return false;

}

bool dfsUndirected(int node, int parent, int graph[MAX\_NODES][MAX\_NODES], bool visited[MAX\_NODES], int n) {

visited[node] = true;

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

// If the neighbor is not visited, recursively call DFS

if (graph[node][neighbor] && !visited[neighbor]) {

if (dfsUndirected(neighbor, node, graph, visited, n))

return true;

}

// If the neighbor is visited and not the parent

else if (graph[node][neighbor] && neighbor != parent) {

return true;

}

}

return false;

}

// Detect Cycle in Undirected Graph

bool hasCycleUndirected(int graph[MAX\_NODES][MAX\_NODES], int n) {

bool visited[MAX\_NODES];

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

visited[i] = false;

}

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

if (!visited[node]) {

if (dfsUndirected(node, -1, graph, visited, n))

return true;

}

}

return false;

}

int main() {

int n;

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

cin >> n;

int graph[MAX\_NODES][MAX\_NODES] = { {0} };

cout << "Enter the adjacency matrix:\n";

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

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

cin >> graph[i][j];

int choice;

do {

cout << "\nMenu:\n";

cout << "1. Detect Cycle in Directed Graph\n";

cout << "2. Detect Cycle in Undirected Graph\n";

cout << "3. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

if (hasCycleDirected(graph, n))

cout << "Directed Graph has a cycle.\n";

else

cout << "Directed Graph doesn't have a cycle.\n";

break;

case 2:

if (hasCycleUndirected(graph, n))

cout << "Undirected Graph has a cycle.\n";

else

cout << "Undirected Graph doesn't have a cycle.\n";

break;

case 3:

cout << "Exiting the program.\n";

break;

default:

cout << "Invalid choice. Please enter a valid option.\n";

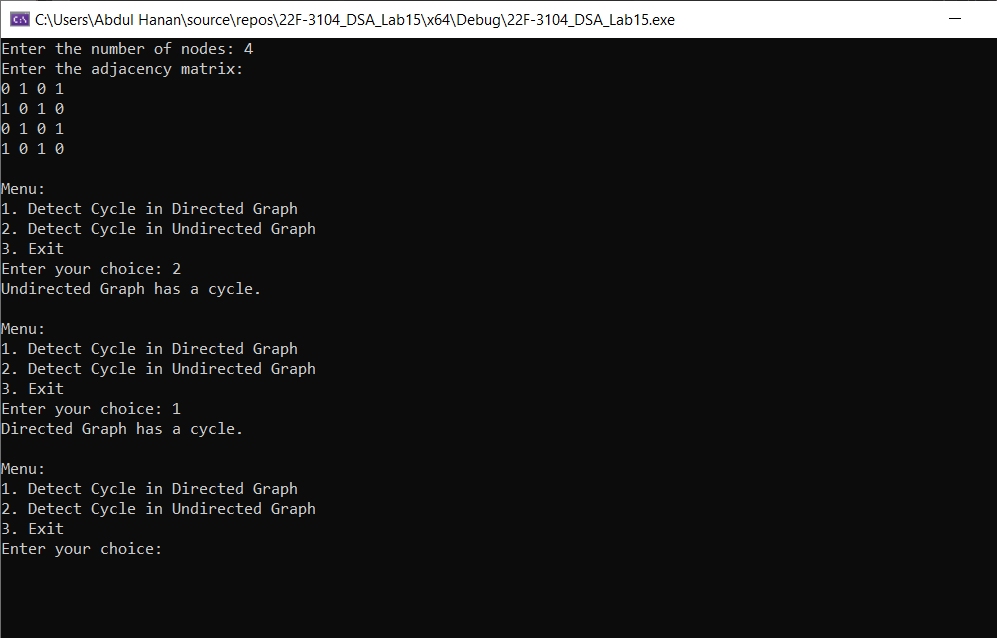
}

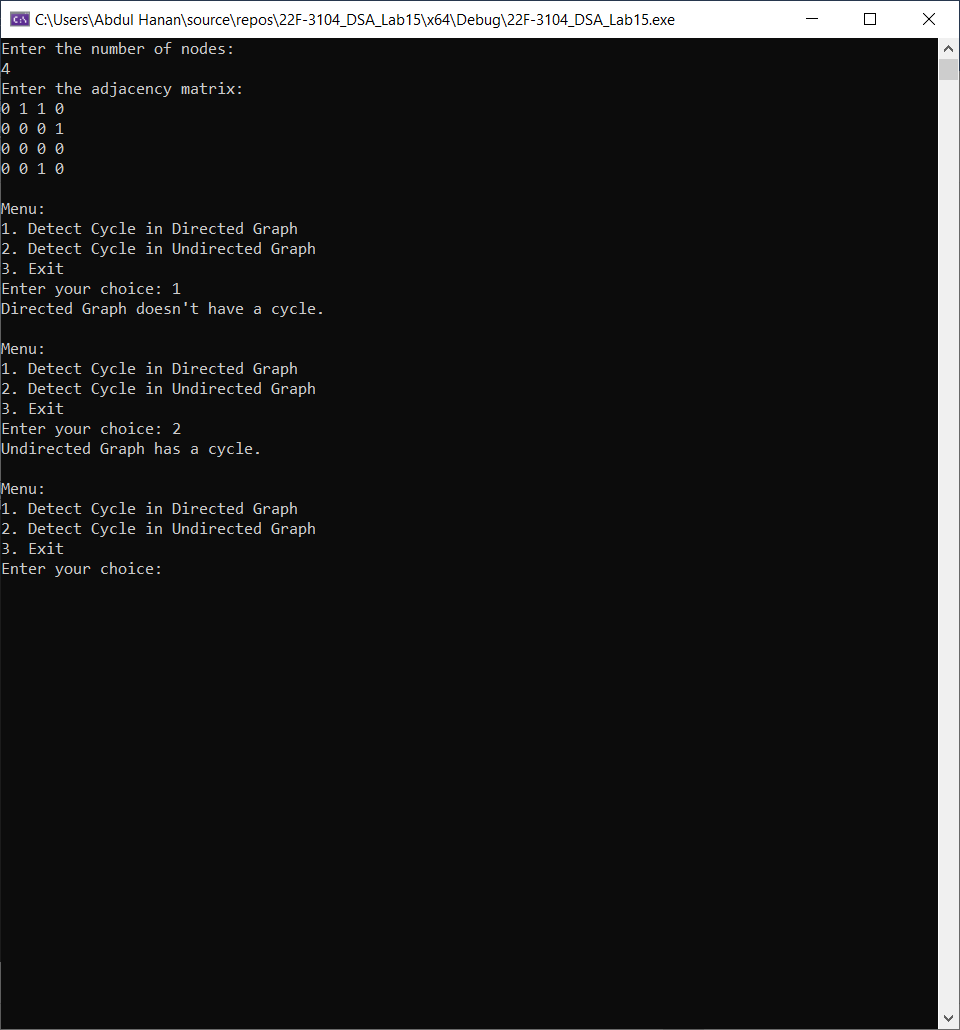
} while (choice != 3);

return 0;

}

**Screen Shot:**

****

****

**Task no 2:**

#include <iostream>

using namespace std;

class Graph {

private:

int numVertices; // Number of vertices

int\*\* adjacencyMatrix; // Adjacency matrix

static const int INT\_MAX\_CUSTOM = 2147483647; // Custom definition of INT\_MAX

public:

Graph(int numVertices);

void addEdge(char v, char w, int weight);

void dijkstra(char src);

~Graph(); // Destructor to free dynamically allocated memory

};

Graph::Graph(int numVertices) {

this->numVertices = numVertices;

// Dynamically allocate memory for the adjacency matrix

adjacencyMatrix = new int\* [numVertices];

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

adjacencyMatrix[i] = new int[numVertices];

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

adjacencyMatrix[i][j] = 0;

}

}

}

void Graph::addEdge(char v, char w, int weight) {

int indexV = v - 'A';

int indexW = w - 'A';

adjacencyMatrix[indexV][indexW] = weight;

}

void Graph::dijkstra(char src) {

// Assuming characters are in consecutive order

int sourceIndex = src - 'A';

int\* distance = new int[numVertices];

bool\* sptSet = new bool[numVertices];

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

distance[i] = INT\_MAX\_CUSTOM;

sptSet[i] = false;

}

distance[sourceIndex] = 0;

for (int count = 0; count < numVertices - 1; ++count) {

int currentVertex, minDistance = INT\_MAX\_CUSTOM;

for (int v = 0; v < numVertices; ++v) {

if (!sptSet[v] && distance[v] <= minDistance) {

currentVertex = v;

minDistance = distance[v];

}

}

sptSet[currentVertex] = true;

for (int v = 0; v < numVertices; ++v) {

if (!sptSet[v] && adjacencyMatrix[currentVertex][v] && distance[currentVertex] != INT\_MAX\_CUSTOM && distance[currentVertex] + adjacencyMatrix[currentVertex][v] < distance[v]) {

distance[v] = distance[currentVertex] + adjacencyMatrix[currentVertex][v];

}

}

}

cout << "Shortest distances from the source node " << src << " to all other nodes:\n";

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

cout << "Node " << char('A' + i) << ": ";

if (distance[i] == INT\_MAX\_CUSTOM)

cout << "Not reachable\n";

else

cout << distance[i] << "\n";

}

}

Graph::~Graph() {

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

delete[] adjacencyMatrix[i];

}

delete[] adjacencyMatrix;

}

int main() {

int vertices, edges;

char source;

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

cin >> vertices;

Graph g(vertices);

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

cin >> edges;

cout << "Enter the details of each edge (source, destination, and weight):\n";

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

char u, v;

int weight;

cin >> u >> v >> weight;

g.addEdge(u, v, weight);

}

cout << "Enter the source node: ";

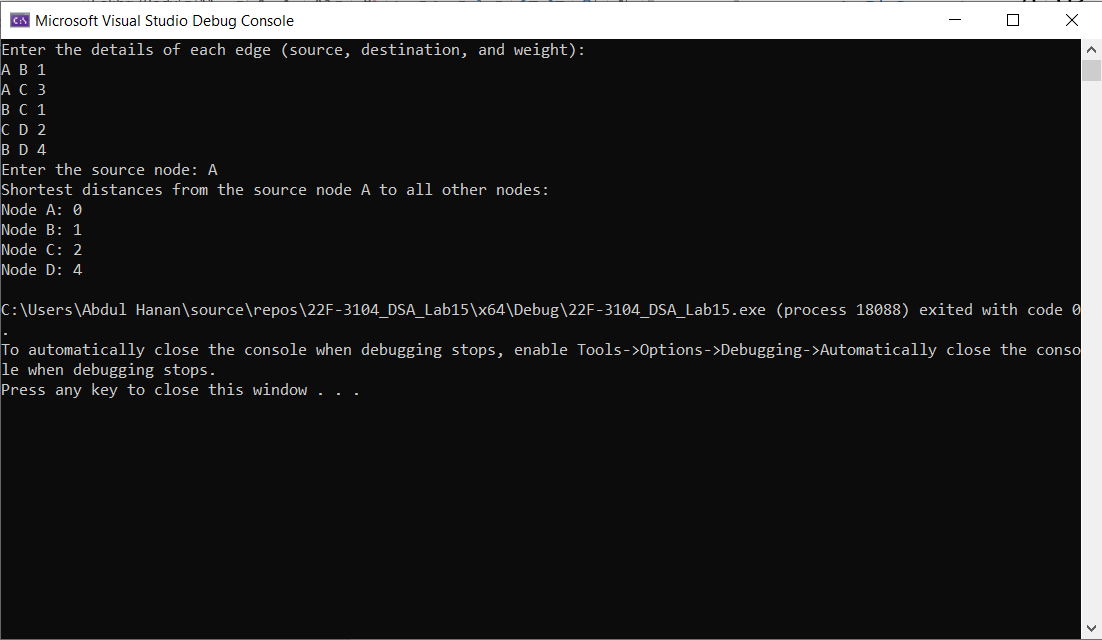
cin >> source;

g.dijkstra(source);

return 0;

}

**Screen Shot:**

****

**Task no 3:**

#include <iostream>

using namespace std;

#define NUM\_VERTICES 4

#define INT\_MAX\_VALUE 2147483647

// Function to find the index of the vertex with the minimum distance value

int minDistance(int distances[], bool sptSet[]) {

int min = INT\_MAX\_VALUE, minIndex;

for (int vertex = 0; vertex < NUM\_VERTICES; vertex++)

if (!sptSet[vertex] && distances[vertex] <= min)

min = distances[vertex], minIndex = vertex;

return minIndex;

}

// Function to print the distances from the source to all vertices

void printSolution(int distances[]) {

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

for (int i = 0; i < NUM\_VERTICES; i++)

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

}

// Function to perform Dijkstra's algorithm to find the shortest paths

void dijkstra(int graph[NUM\_VERTICES][NUM\_VERTICES], int source) {

int distances[NUM\_VERTICES];

bool sptSet[NUM\_VERTICES];

for (int i = 0; i < NUM\_VERTICES; i++)

distances[i] = INT\_MAX\_VALUE, sptSet[i] = false;

distances[source] = 0;

for (int count = 0; count < NUM\_VERTICES - 1; count++) {

int u = minDistance(distances, sptSet);

sptSet[u] = true;

// Update the distances of the adjacent vertices

for (int v = 0; v < NUM\_VERTICES; v++)

if (!sptSet[v] && graph[u][v] && distances[u] != INT\_MAX\_VALUE && distances[u] + graph[u][v] < distances[v])

distances[v] = distances[u] + graph[u][v];

}

printSolution(distances);

}

int main() {

int graph[NUM\_VERTICES][NUM\_VERTICES];

cout << "Enter the graph data (4x4 adjacency matrix):\n";

for (int i = 0; i < NUM\_VERTICES; ++i)

for (int j = 0; j < NUM\_VERTICES; ++j)

cin >> graph[i][j];

cout << "\nMenu:\n";

cout << "1. Find shortest paths using Dijkstra's algorithm\n";

cout << "2. Exit\n";

cout << "Enter your choice: ";

int userChoice;

cin >> userChoice;

switch (userChoice) {

case 1:

cout << "Enter the source node: ";

int sourceNode;

cin >> sourceNode;

dijkstra(graph, sourceNode);

break;

case 2:

cout << "Exit program (^\_^)| :\n";

return 0;

default:

cout << "Invalid choice\n";

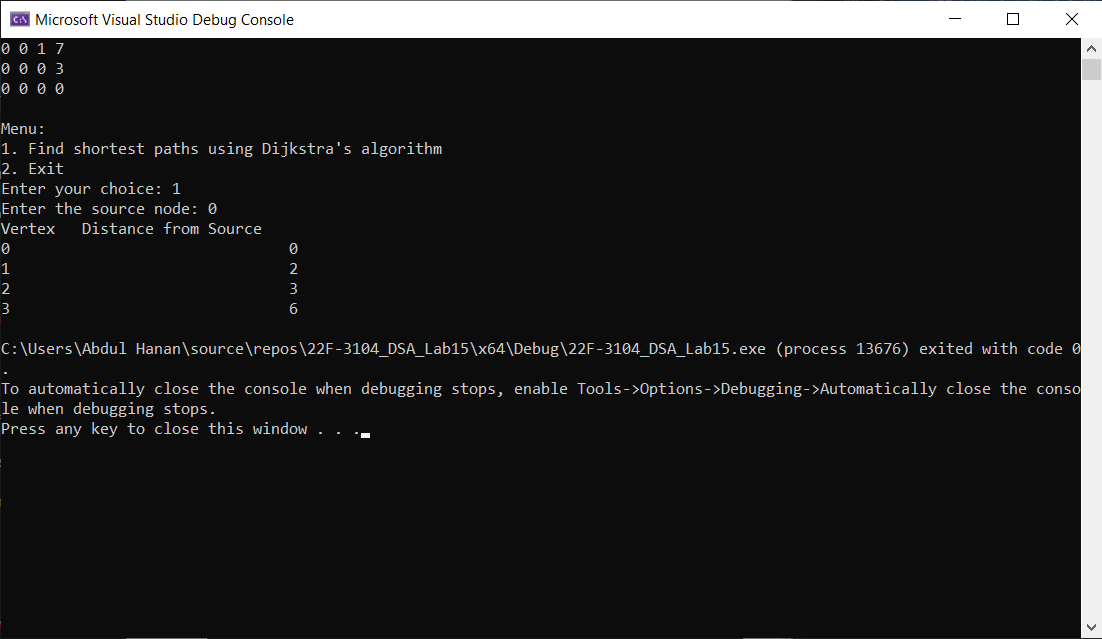
return 0;

}

return 0;

}

**Screen Shot:**

****