

22F-3104

DSA-Lab 14



November 30, 2023

FASt cfd campus

**Task no 1:**

#include <iostream>

#include <string>

using namespace std;

struct Node {

string label;

Node\* next;

};

// Function to create a graph from the given data

Node\* createGraph() {

Node\* graph = nullptr;

int numPersons;

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

cin >> numPersons;

cin.ignore(); // Ignore the newline character

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

string person;

cout << "Enter the name of person " << (i + 1) << ": ";

getline(cin, person);

Node\* newNode = new Node{ person, nullptr };

if (graph == nullptr) {

graph = newNode;

}

else {

Node\* curr = graph;

while (curr->next != nullptr) {

curr = curr->next;

}

curr->next = newNode;

}

}

return graph;

}

// Function to add a connection between two individuals

void addConnection(Node\* graph) {

string person1, person2;

cout << "Enter the first person: ";

cin.ignore(); // Ignore the newline character

getline(cin, person1);

cout << "Enter the second person: ";

getline(cin, person2);

Node\* curr = graph;

Node\* person1Node = nullptr;

Node\* person2Node = nullptr;

// Find the nodes corresponding to the entered persons

while (curr != nullptr) {

if (curr->label == person1) {

person1Node = curr;

}

else if (curr->label == person2) {

person2Node = curr;

}

if (person1Node != nullptr && person2Node != nullptr) {

break;

}

curr = curr->next;

}

// Add the connection to the adjacency list

if (person1Node != nullptr && person2Node != nullptr) {

Node\* newNode1 = new Node{ person2Node->label, person1Node->next };

person1Node->next = newNode1;

Node\* newNode2 = new Node{ person1Node->label, person2Node->next };

person2Node->next = newNode2;

cout << "Connection added successfully." << endl;

}

else {

cout << "One or both persons not found." << endl;

}

}

// Function to count the number of connections for each person

string findMostSocialPerson(Node\* graph) {

int maxConnections = 0;

string mostSocialPerson;

Node\* curr = graph;

while (curr != nullptr) {

int connections = 0;

Node\* neighbor = curr->next;

while (neighbor != nullptr) {

connections++;

neighbor = neighbor->next;

}

if (connections > maxConnections) {

maxConnections = connections;

mostSocialPerson = curr->label;

}

curr = curr->next;

}

return mostSocialPerson;

}

int main() {

Node\* graph = createGraph();

int choice;

do {

cout << "\nMenu:" << endl;

cout << "1. Add Connection" << endl;

cout << "2. Determine Most Social Person" << endl;

cout << "3. Exit" << endl;

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1: {

addConnection(graph);

break;

}

case 2: {

string mostSocialPerson = findMostSocialPerson(graph);

cout << "The most social person is: " << mostSocialPerson << endl;

break;

}

case 3: {

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

break;

}

default:

cout << "Invalid choice" << endl;

}

} while (choice != 3);

// Clean up the graph

Node\* curr = graph;

while (curr != nullptr) {

Node\* temp = curr;

curr = curr->next;

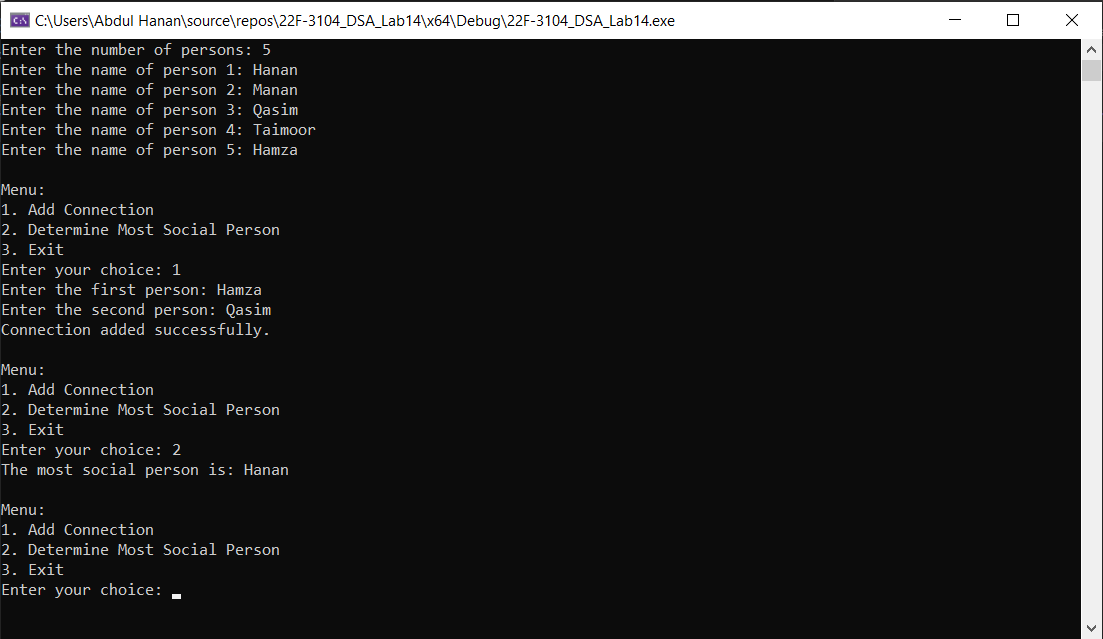
delete temp;

}

return 0;

}

**Screen Shot:**

****

**Task no 2:**

#include <iostream>

using namespace std;

bool isBipartite(int\*\* adjMatrix, int numNodes, int startNode) {

int\* color = new int[numNodes];

bool\* visited = new bool[numNodes] {false};

color[startNode] = 0;

visited[startNode] = true;

int\* queue = new int[numNodes];

int front = -1;

int rear = -1;

queue[++rear] = startNode;

while (front != rear) {

int currentNode = queue[++front];

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

if (adjMatrix[currentNode][i]) {

if (!visited[i]) {

color[i] = 1 - color[currentNode]; // Assign the opposite color to the neighbor

visited[i] = true;

queue[++rear] = i;

}

else if (color[i] == color[currentNode]) {

// If an adjacent node is colored with same color it is not bipartiled

delete[] color;

delete[] visited;

delete[] queue;

return false;

}

}

}

}

delete[] color;

delete[] visited;

delete[] queue;

return true;

}

int main() {

int numNodes, numEdges;

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

cin >> numNodes;

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

cin >> numEdges;

int\*\* adjMatrix = new int\* [numNodes];

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

adjMatrix[i] = new int[numNodes] {0};

}

cout << "Enter the edges (node1 node2):" << endl;

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

int node1, node2;

cin >> node1 >> node2;

adjMatrix[node1][node2] = 1;

adjMatrix[node2][node1] = 1; // Assuming an undirected graph

}

// Check if the graph is bipartite

bool bipartite = true;

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

if (!isBipartite(adjMatrix, numNodes, i)) {

bipartite = false;

break;

}

}

// Output the result

if (bipartite) {

cout << "The graph is bipartite." << endl;

}

else {

cout << "The graph is not bipartite." << endl;

}

// Deallocate memory

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

delete[] adjMatrix[i];

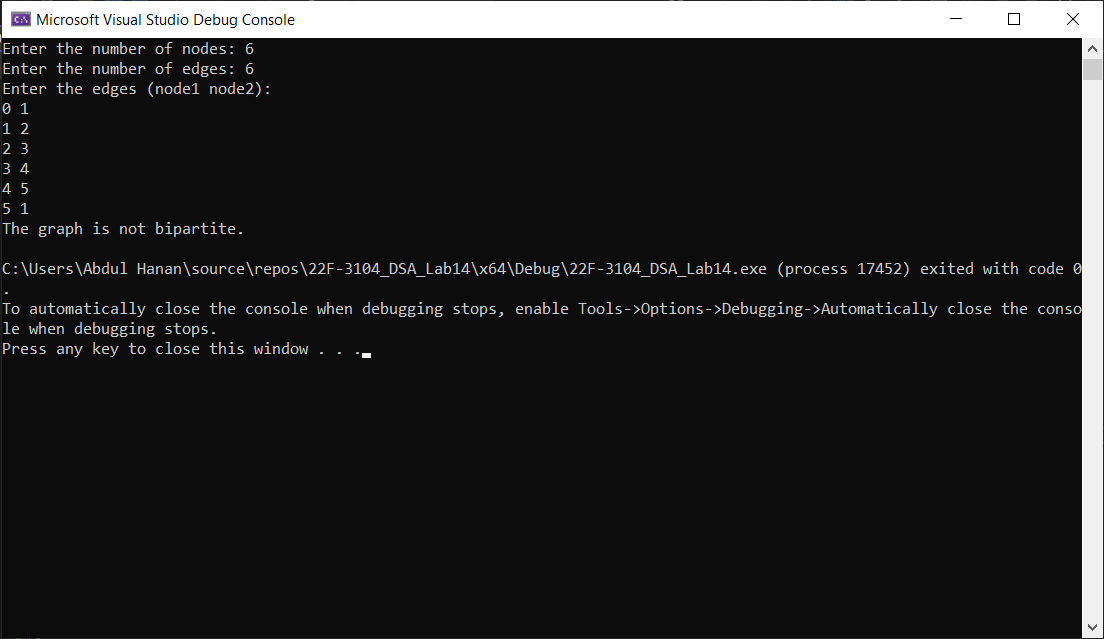
}

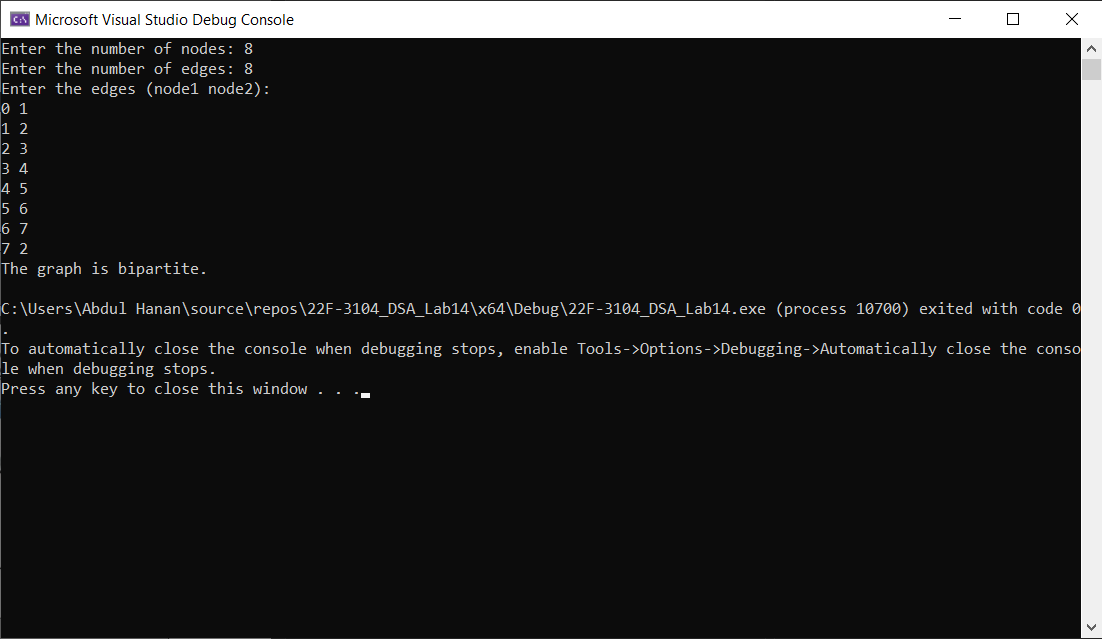
delete[] adjMatrix;

return 0;

}

**Screen Shot:**

****

****

**Task no 3:**

#include <iostream>

using namespace std;

#define MAX\_NODES 100

// Custom Queue implementation

class Queue {

private:

int front, rear, size;

int elements[MAX\_NODES];

public:

Queue() {

front = rear = -1;

size = 0;

}

bool isEmpty() {

return size == 0;

}

bool isFull() {

return size == MAX\_NODES;

}

void enqueue(int item) {

if (isFull()) {

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

return;

}

if (rear == -1) {

front = rear = 0;

}

else {

rear = (rear + 1) % MAX\_NODES;

}

elements[rear] = item;

size++;

}

int dequeue() {

if (isEmpty()) {

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

return -1;

}

int item = elements[front];

if (front == rear) {

front = rear = -1;

}

else {

front = (front + 1) % MAX\_NODES;

}

size--;

return item;

}

};

// Function to perform BFS

bool bfs(int adjMatrix[MAX\_NODES][MAX\_NODES], int numNodes, int startNode, bool visited[MAX\_NODES]) {

Queue queue;

// Mark the start node as visited and enqueue it

visited[startNode] = true;

queue.enqueue(startNode);

while (!queue.isEmpty()) {

int currentNode = queue.dequeue();

// Iterate through all the adjacent nodes of the current node

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

if (adjMatrix[currentNode][i] == 1 && !visited[i]) {

visited[i] = true;

queue.enqueue(i);

}

}

}

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

if (!visited[i]) {

return false;

}

}

return true;

}

// Function to check if the graph is connected

bool isConnected(int adjMatrix[MAX\_NODES][MAX\_NODES], int numNodes) {

bool visited[MAX\_NODES] = { false };

// Perform BFS starting from node 0

return bfs(adjMatrix, numNodes, 0, visited);

}

int main() {

int numNodes, numEdges;

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

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

cin >> numNodes;

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

cin >> numEdges;

cout << "Enter the edges (node1 node2):" << endl;

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

int node1, node2;

cin >> node1 >> node2;

// Update the adjacency matrix to represent the edges

adjMatrix[node1][node2] = 1;

adjMatrix[node2][node1] = 1;

}

// Check if the graph is connected

if (isConnected(adjMatrix, numNodes)) {

cout << "The graph is connected." << endl;

}

else {

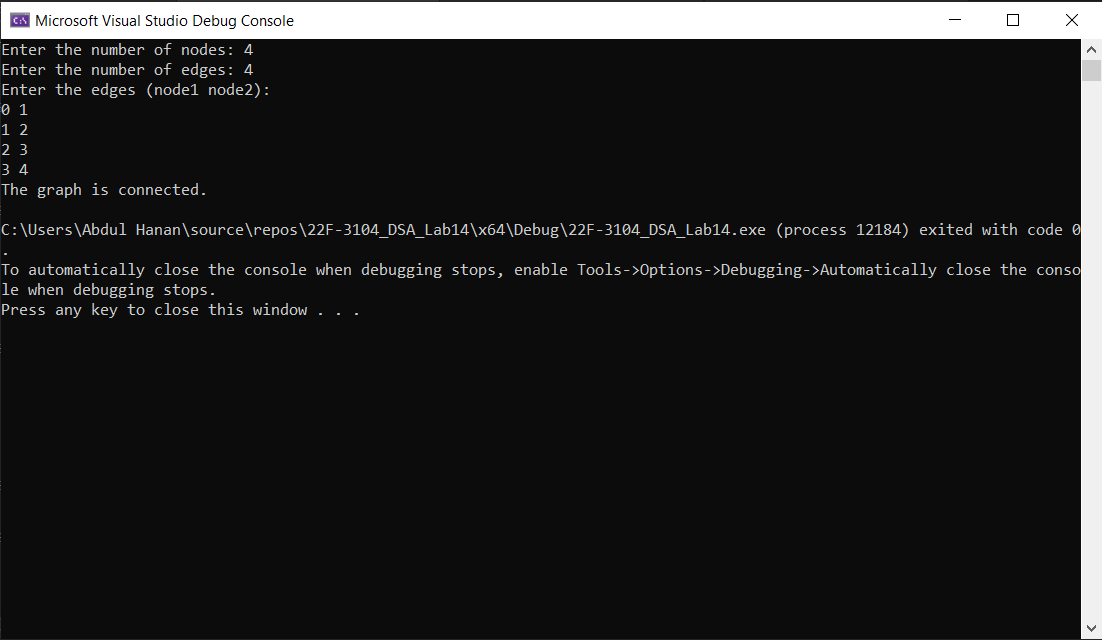
cout << "The graph is not connected." << endl;

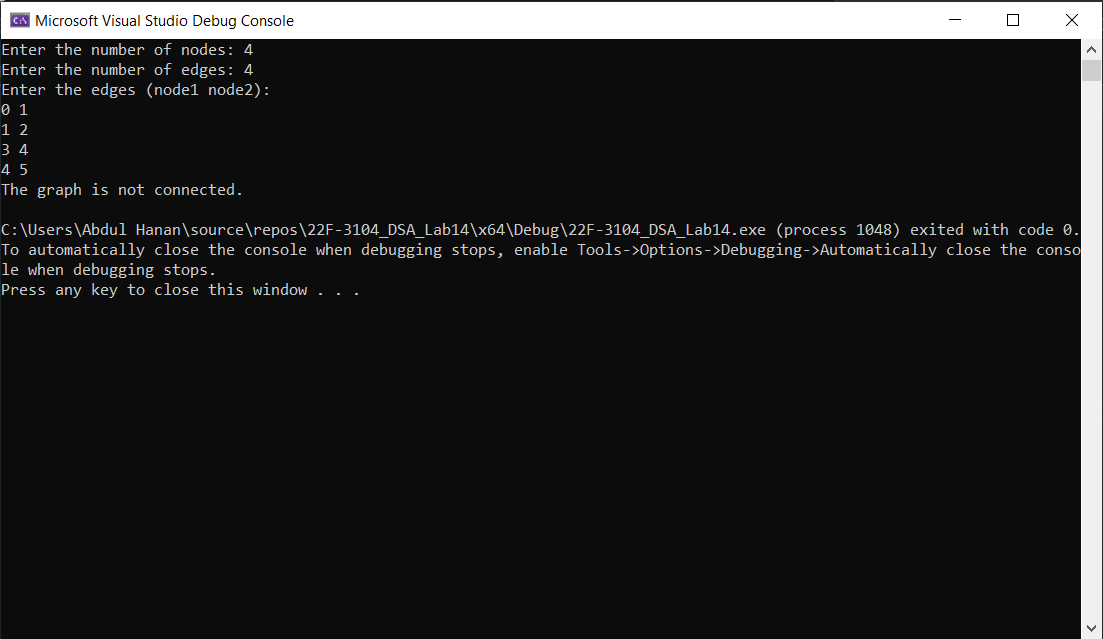
}

return 0;

}

**Screen Shot:**

****

****

**Task no 4:**

#include <iostream>

using namespace std;

int\*\* createMatrix(int rows, int cols) {

int\*\* mat = new int\* [rows];

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

mat[i] = new int[cols];

fill(mat[i], mat[i] + cols, 0);

}

return mat;

}

void deleteMatrix(int\*\* mat, int rows) {

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

delete[] mat[i];

}

delete[] mat;

}

int calculateDeterminant(int\*\* mat, int n) {

if (n == 1) {

return mat[0][0];

}

int determinant = 0;

int\*\* subMatrix = createMatrix(n - 1, n - 1);

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

int subMatrixRow = 0;

for (int row = 1; row < n; ++row) {

int subMatrixCol = 0;

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

if (j != col) {

subMatrix[subMatrixRow][subMatrixCol] = mat[row][j];

++subMatrixCol;

}

}

++subMatrixRow;

}

determinant += (col % 2 == 0 ? 1 : -1) \* mat[0][col] \* calculateDeterminant(subMatrix, n - 1);

}

deleteMatrix(subMatrix, n - 1);

return determinant;

}

int calculateSpanningTrees(int n, int\*\* adjMatrix) {

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

int degree = 0;

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

degree += adjMatrix[i][j];

}

adjMatrix[i][i] = degree;

}

int reducedNodes = n - 1;

int\*\* reducedMatrix = createMatrix(reducedNodes, reducedNodes);

for (int i = 1; i < n; ++i) {

for (int j = 1; j < n; ++j) {

reducedMatrix[i - 1][j - 1] = adjMatrix[i][j];

}

}

int numSpanningTrees = calculateDeterminant(reducedMatrix, reducedNodes);

deleteMatrix(reducedMatrix, reducedNodes);

return numSpanningTrees;

}

int main() {

int nodes, edges;

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

cin >> nodes;

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

cin >> edges;

int\*\* adjacencyMatrix = createMatrix(nodes, nodes);

cout << "Enter the details of each edge (node u and v):" << endl;

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

int u, v;

cin >> u >> v;

adjacencyMatrix[u - 1][v - 1] = 1;

adjacencyMatrix[v - 1][u - 1] = 1;

}

int numSpanningTrees = calculateSpanningTrees(nodes, adjacencyMatrix);

cout << "Number of Spanning Trees: " << numSpanningTrees << endl;

deleteMatrix(adjacencyMatrix, nodes);

return 0;

}

**Screen Shot**:

