Міністерство освіти і науки України
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"
Факультет інформатики та обчислювальної техніки
Кафедра інформатики та програмної інженерії

Звіт

з лабораторної роботи №3 з дисципліни
«Алгоритми та структури даних 2. Структури даних»

«Прикладні задачі теорії графів ч.1»
Варіант 3

Виконав студент ІП-15, Борисик Владислав Тарасович
(шифр, прізвище, ім'я, по батькові)

Перевірив Соколовський Владислав Володимирович
(прізвище, ім'я, по батькові)

Київ 2022

Мета лабораторної роботи

Мета роботи – вивчити основні прикладні алгоритми на графах та способи їх імплементації.

Завдання

Згідно варіанту (таблиця 2.1), розробити та записати алгоритм задачі на графах за допомогою псевдокоду (чи іншого способу за вибором).

Виконати програмну реалізацію алгоритму на будь-якій мові програмування для довільного графа, передбачити введення розмірності графа та введення даних графа вручну чи випадковим чином.

Для самостійно обраного графа (розмірності не менше 9 вершин) розв’язати задану за варіантом задачу вручну.

Зробити узагальнений висновок з лабораторної роботи, у якому порівняти програмне та ручне розв’язання задачі.

3	Пошук маршруту у графі	Террі	Неорієнтований	Матриця суміжності

Код алгоритму

```
vector<int> Graph::detour() {
    int size = get_size();
    vector<int> path;
    path.push_back(1);

    //i++ = ?
    for(int i = 0; i < size; i++){
        for(int j = 0; j < size ; j++){
            if(i != j){
                //cout << i << ":" << j << " " << adjacency_matrix[i][j] << endl;
                //adjacency_matrix[i][j] == 1 && (j != path[path.size() - 3] && j !=
path[path.size() - 2])
                if(adjacency_matrix[i][j] == 1 && is_vector_contains(path, j+1) == false){
                    path.push_back(j + 1);

                    i = j;
                    j = -1;
                }
            }
        }
    }

    fill_missed_vertices(path);

    return path;
}


void Graph::fill_missed_vertices(vector<int> &path) {
    int size = get_size();
    // вершина, якої не вистачає
    int missed_vertex = -1;

    for(int i = 1; i <= size; i++){
        if(is_vector_contains(path, i) == false){
            // якщо вершина знайдена, то ініціалізуємо її
            missed_vertex = i-1;
        }
    }

    // і координата вершини, з якої перейшли в іншу вершину
    int last_vertex_i = -1;
    // j координата вершини, з якої перейшли в іншу вершину
    int last_vertex_j = -1;
    // якщо якоїсь вершини не вистачає
    if(missed_vertex != -1){
        // ініціалізуємо і останнім елементом вектора path і віднімаємо від нього одиницю
        // наприклад, якщо path буде 3,1,4,6 то і буде дорівнювати 5 (6 - останній елемент, і
віднімаємо від нього одиницю)
        for(int i = path[path.size()-1] - 1; i > 0; i--){
            for(int j = size-1; j > 0; j--){
                // якщо i != j, тобто не беремо вершину, яка інцидентна сама собі
                if(i != j){
                    // якщо елемент матриці = 1 і j = зниклій вершині
                    if(adjacency_matrix[i][j] == 1 && j == missed_vertex){
```

```
                        //додаємо вершину в вектор path
                        path.push_back(j+1);
                        // змінюємо зниклу вершину на -1, для того, щоб далі не додавались
зайві елементи у вектор path
                        missed_vertex = -1;
                    }
                    // якщо елемент матриці = 1 і координата вершини і != j або координата
вершини j != i і зникла вершина != -1
                    if(adjacency_matrix[i][j] == 1 && (last_vertex_i != j || last_vertex_j !=
i) && missed_vertex != -1){
                        // додаємо вершину в вектор
                        path.push_back(j + 1);
                        // змінюємо і координату останньої вершини на поточну вершину
                        last_vertex_i = i;
                        // змінюємо j координату останньої вершини на поточну вершину
                        last_vertex_j = j;

                        // змінюємо і на j
                        i = j;
                        // присвоюємо j значення size
                        j = size;
                    }
                }
            }
        }
    }
}


vector<int> Graph::find_path(vector<int> detour) {
    vector<int> path;

    // чи знайдена початкова вершина графу в векторі
    bool is_begin_vertex_found = false;
    // чи знайдена кінцева вершина графу в векторі
    bool is_end_vertex_found = false;

    // ітеруємось через вектор зліва на право
    for(int i = 0; i < detour.size(); i++){
        // якщо елемент вектору = початковій вершині
        if(detour[i] == begin_vertex)
            // маркуємо її як знайдену
            is_begin_vertex_found = true;
        // якщо елемент вектору = кінцевій вершині
        else if(detour[i] == end_vertex) {
            // маркуємо її як знайдену
            is_end_vertex_found = true;
            // додаємо її у вектор
            path.push_back(detour[i]);
        }

        // якщо початкова вершина знайдена, а кінцева ще ні
        if(is_begin_vertex_found == true && is_end_vertex_found == false){
            //додаємо елемент у вектор path
            path.push_back(detour[i]);
        }
    }

    // якщо розмір вектору менше за 2, це означає, що шлях потрібно шукати в зворотньому
напрямку
```

```
    // наприклад, якщо даний маршрут 1 3 4 2 6 і потрібно знайти шлях від 2 до 4
    // то потрібно шукати шлях не зліва на право, а справа на ліво
    if(path.size() < 2){
        path.clear();

        // чи знайдена початкова вершина графу в векторі
        is_begin_vertex_found = false;
        // чи знайдена кінцева вершина графу в векторі
        is_end_vertex_found = false;

        // ітеруємось через вектор справа на ліво
        for(int i = get_size() - 1; i > 0; i--){
            // якщо елемент вектору = кінцевій вершині
            if(detour[i] == end_vertex) {
                // маркуємо її як знайдену
                is_end_vertex_found = true;
                // додаємо її у вектор
                path.push_back(detour[i]);
            }
            // якщо елемент вектору = початковій вершині
            else if(detour[i] == begin_vertex)
                // маркуємо її як знайдену
                is_begin_vertex_found = true;

            // якщо початкова вершина знайдена, а кінцева ще ні
            if(is_end_vertex_found == false && is_begin_vertex_found == true){
                //додаємо елемент у вектор path
                path.push_back(detour[i]);
            }
        }
    }

    return path;
}
```

Програмна реалізація на мові C++

main.cpp:

```cpp
#include "functions.h"
#include <ctime>

int main() {
    srand(time(nullptr));

    int graph_size = get_graph_size();
    char graph_mode = get_graph_mode();
    int begin_vertex = get_begin_vertex(graph_size);
    int end_vertex = get_end_vertex();

    Graph graph = Graph(graph_size, graph_mode, begin_vertex, end_vertex);
    cout << "Adjacency matrix of graph:\n";
    graph.print();

    vector<int> detour = graph.detour();
    cout << "Graph detour: ";
    print_vector(detour);

    vector<int> path = graph.find_path(detour);
    printf("Path from %d to %d: ", begin_vertex, end_vertex);
    print_vector(path);
}
```

functions.h

```
#ifndef ASD_LABWORK_3_FUNCTIONS_H
#define ASD_LABWORK_3_FUNCTIONS_H

#endif //ASD_LABWORK_3_FUNCTIONS_H
#include "classes.h"


int get_graph_size();
char get_graph_mode();
int get_begin_vertex(int matrix_size);
int get_end_vertex();
void print_vector(const vector<int>& vec);
```

functions.cpp:

```cpp
#include <iostream>
#include "classes.h"
using namespace std;

int get_graph_size(){
    int graph_size;

    cout << "Enter graph size: ";
    cin >> graph_size;

    return graph_size;
}

char get_graph_mode(){
    char graph_mode;

    cout << "Would you like enter graph manually or generate it randomly?\nM - Manually, R -
Randomly: ";
    cin >> graph_mode;

    while (graph_mode != 'm' && graph_mode != 'M' && graph_mode != 'r' && graph_mode != 'R'){
        cout << "Error. Enter either M or R: \n";
        cin >> graph_mode;
    }

    return graph_mode;
}

int get_begin_vertex(int matrix_size){
    int begin_vertex;

    cout << "Enter begin vertex: ";
    cin >> begin_vertex;

    while (begin_vertex > matrix_size){
        cout << "Error. Begin vertex must be lower or equal to graph size";
        cin >> begin_vertex;
    }

    return begin_vertex;
}

int get_end_vertex(){
    int end_vertex;

    cout << "Enter end vertex: ";
    cin >> end_vertex;

    while (end_vertex < 1){
        cout << "Error. End vertex must be greater or equal to 1";
        cin >> end_vertex;
    }

    return end_vertex;
}
```

```cpp
void print_vector(const vector<int>& vec){
    for(int i : vec)
        cout << i << " ";
    cout << "\n";
}
```

classes.h:

```
#ifndef ASD_LABWORK_3_CLASSES_H
#define ASD_LABWORK_3_CLASSES_H
#endif

#include <vector>
#include <iostream>
using namespace std;

class Graph{
private:
    vector<vector<int>> adjacency_matrix{};
    int graph_size;
    int begin_vertex;
    int end_vertex;

    int get_size();
    void generate_manually();
    void generate_randomly();
    void initialize_zeroes();
    bool is_vector_contains(vector<int> vec, int value);
    void fill_missed_vertices(vector<int> &path);
public:
    Graph(int size, char mode, int begin_vertex, int end_vertex);

    void print();
    vector<int> detour();
    vector<int> find_path(vector<int> detour);
};
```

classes.cpp:

```cpp
#include "classes.h"
#include <random>
#include <algorithm>

int Graph::get_size() {
    return graph_size;
}

void Graph::print() {
    for(int i = 0; i < get_size(); i++){
        for(int j = 0; j < get_size(); j++){
            cout << adjacency_matrix[i][j] << " ";
        }
        cout << "\n";
    }
}

void Graph::generate_manually(){
    int size = get_size();

    for(int i = 0; i < size; i++){
        vector<int> row;

        for(int j = 0; j < size; j++){
            printf("Enter data on row %d column %d: ", i+1, j+1);

            int input_data;
            cin >> input_data;

            row.push_back(input_data);
        }

        adjacency_matrix.push_back(row);
    }
}

void Graph::initialize_zeroes(){
    int size = get_size();

    for(int i = 0; i < size; i++){
        vector<int> row;

        for(int j = 0; j < size; j++){
            row.push_back(0);
        }

        adjacency_matrix.push_back(row);
    }
}

void Graph::generate_randomly(){
    int size = get_size();
    initialize_zeroes();
```

```
    for(int i = 0; i < size; i++){
        for(int j = i; j < size; j++){
            int random_number = rand() % 2;

            adjacency_matrix[i][j] = random_number;
            adjacency_matrix[j][i] = random_number;
        }
    }
}

Graph::Graph(int size, char mode, int begin_vertex, int end_vertex) {
    graph_size = size;
    this->begin_vertex = begin_vertex;
    this->end_vertex = end_vertex;

    if(mode == 'm' || mode == 'M'){
        generate_manually();
    }
    else if(mode == 'r' || mode == 'R'){
        generate_randomly();
    }
}

bool Graph::is_vector_contains(vector<int> vec, int value) {
    if(std::find(vec.begin(), vec.end(), value) != vec.end()) {
        return true;
    } else {
        return false;
    }
}

void Graph::fill_missed_vertices(vector<int> &path) {
    int size = get_size();
    // вершина, якої не вистачає
    int missed_vertex = -1;

    for(int i = 1; i <= size; i++){
        if(is_vector_contains(path, i) == false){
            // якщо вершина знайдена, то ініціалізуємо її
            missed_vertex = i-1;
        }
    }

    // і координата вершини, з якої перейшли в іншу вершину
    int last_vertex_i = -1;
    // j координата вершини, з якої перейшли в іншу вершину
    int last_vertex_j = -1;
    // якщо якоїсь вершини не вистачає
    if(missed_vertex != -1){
        // ініціалізуємо і останнім елементом вектора path і віднімаємо від нього
одиницю
        // наприклад, якщо path буде 3,1,4,6 то і буде дорівнювати 5 (6 - останній
елемент, і віднімаємо від нього одиницю)
        for(int i = path[path.size()-1] - 1; i > 0; i--){
```

```
            for(int j = size-1; j > 0; j--){
                // якщо i != j, тобто не беремо вершину, яка інцидентна сама собі
                if(i != j){
                    // якщо елемент матриці = 1 і j = зниклій вершині
                    if(adjacency_matrix[i][j] == 1 && j == missed_vertex){
                        //додаємо вершину в вектор path
                        path.push_back(j+1);
                        // змінюємо зниклу вершину на -1, для того, щоб далі не
додавались зайві елементи у вектор path
                        missed_vertex = -1;
                    }
                    // якщо елемент матриці = 1 і координата вершини і != j або
координата вершини j != i і зникла вершина != -1
                    if(adjacency_matrix[i][j] == 1 && (last_vertex_i != j ||
last_vertex_j != i) && missed_vertex != -1){
                        // додаємо вершину в вектор
                        path.push_back(j + 1);
                        // змінюємо і координату останньої вершини на поточну вершину
                        last_vertex_i = i;
                        // змінюємо j координату останньої вершини на поточну вершину
                        last_vertex_j = j;

                        // змінюємо і на j
                        i = j;
                        // присвоюємо j значення size
                        j = size;
                    }
                }
            }
        }
    }
}

vector<int> Graph::detour() {
    int size = get_size();
    vector<int> path;
    path.push_back(1);

    for(int i = 0; i < size; i++){
        for(int j = 0; j < size ; j++){
            if(i != j){
                //cout << i << ":" << j << " " << adjacency_matrix[i][j] << endl;
                //adjacency_matrix[i][j] == 1 && (j != path[path.size() - 3] && j !=
path[path.size() - 2])
                if(adjacency_matrix[i][j] == 1 && is_vector_contains(path, j+1) ==
false){
                    path.push_back(j + 1);

                    i = j;
                    j = -1;
                }
            }
        }
    }
```

```
    fill_missed_vertices(path);

    return path;
}

vector<int> Graph::find_path(vector<int> detour) {
    vector<int> path;

    // чи знайдена початкова вершина графу в векторі
    bool is_begin_vertex_found = false;
    // чи знайдена кінцева вершина графу в векторі
    bool is_end_vertex_found = false;

    // ітеруємось через вектор зліва на право
    for(int i = 0; i < detour.size(); i++){
        // якщо елемент вектору = початковій вершині
        if(detour[i] == begin_vertex)
            // маркуємо її як знайдену
            is_begin_vertex_found = true;
        // якщо елемент вектору = кінцевій вершині
        else if(detour[i] == end_vertex) {
            // маркуємо її як знайдену
            is_end_vertex_found = true;
            // додаємо її у вектор
            path.push_back(detour[i]);
        }

        // якщо початкова вершина знайдена, а кінцева ще ні
        if(is_begin_vertex_found == true && is_end_vertex_found == false){
            //додаємо елемент у вектор path
            path.push_back(detour[i]);
        }
    }

    // якщо розмір вектору менше за 2, це означає, що шлях потрібно шукати в
зворотньому напрямку
    // наприклад, якщо даний маршрут 1 3 4 2 6 і потрібно знайти шлях від 2 до 4
    // то потрібно шукати шлях не зліва на право, а справа на ліво
    if(path.size() < 2){
        path.clear();

        // чи знайдена початкова вершина графу в векторі
        is_begin_vertex_found = false;
        // чи знайдена кінцева вершина графу в векторі
        is_end_vertex_found = false;

        // ітеруємось через вектор справа на ліво
        for(int i = get_size() - 1; i > 0; i--){
            // якщо елемент вектору = кінцевій вершині
            if(detour[i] == end_vertex) {
                // маркуємо її як знайдену
                is_end_vertex_found = true;
                // додаємо її у вектор
                path.push_back(detour[i]);
            }
```

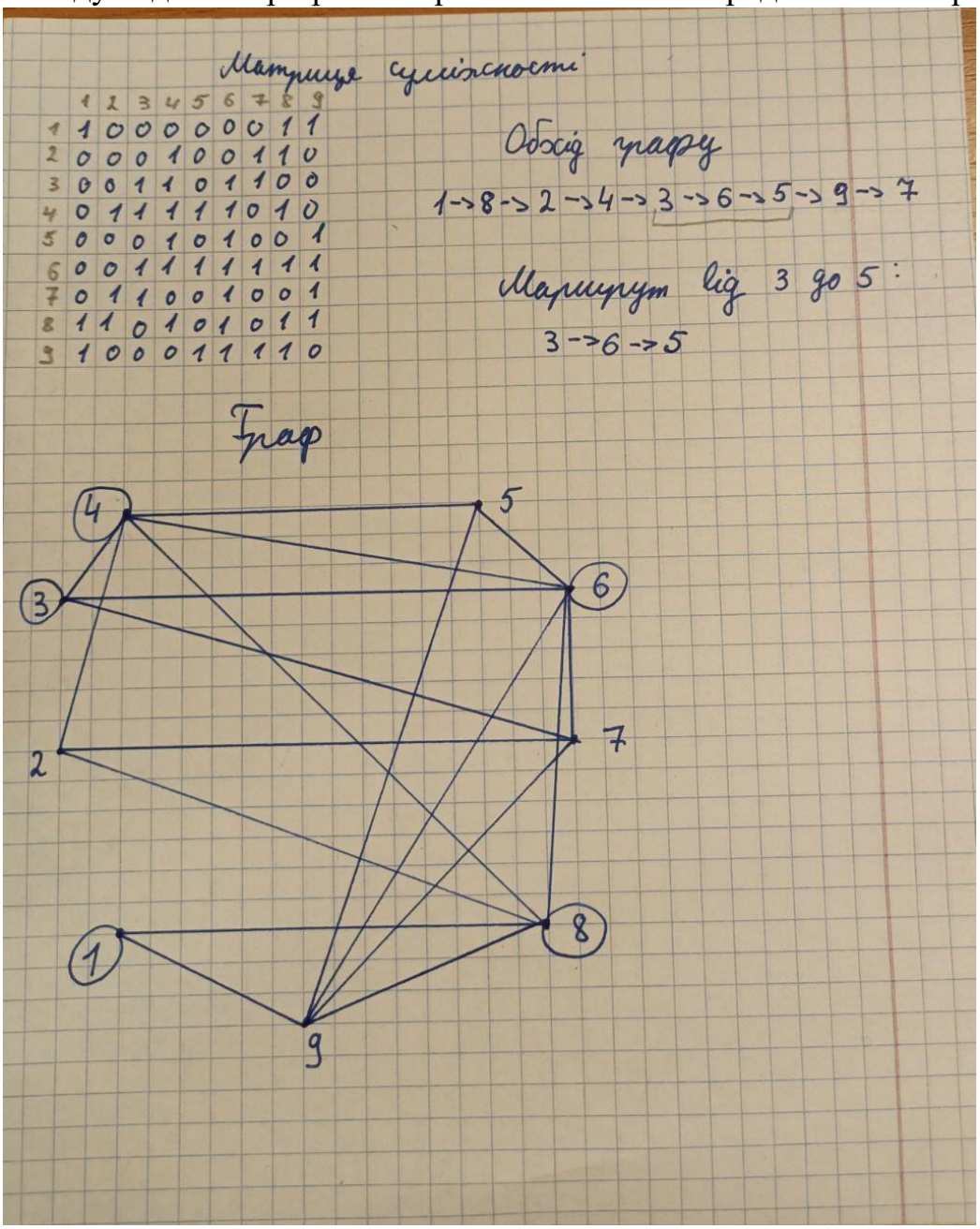
```
            // якщо елемент вектору = початковій вершині
            else if(detour[i] == begin_vertex)
                // маркуємо її як знайдену
                is_begin_vertex_found = true;

            // якщо початкова вершина знайдена, а кінцева ще ні
            if(is_end_vertex_found == false && is_begin_vertex_found == true){
                //додаємо елемент у вектор path
                path.push_back(detour[i]);
            }
        }
    }

    return path;
}
```

Перевірка алгоритму

Для початку, зробимо обхід графу і знайдемо шлях від 3-ї вершини до 5-ї вручну (для обходу надаємо пріоритет вершині з меншим порядковим номером):



Тепер, зробимо обхід графу і знайдемо шлях від 3-ї вершини до 5-ї за допомогою нашої програми:

```
Enter graph size:9
 Would you like enter graph manually or generate it randomly?
M - Manually, R - Randomly:m
 Enter begin vertex:3
 Enter end vertex:5
 Enter data on row 1 column 1:
```

Після введення графу вручну, програма виводить нам результати:

```
 Enter data on row 9 column 4:0
 Enter data on row 9 column 5:1
 Enter data on row 9 colu
mn 6:1
 Enter data on row 9 column 7:1
 Enter data on row 9 column 8:1
 Enter data on row 9 column 9:0
 Adjacency matrix of grap
h:
1 0 0 0 0 0 0 1 1
0 0 0 1 0 0 1 1 0
0 0 1 1 0 1 1 0 0
0 1 1 1 1 1 0 1 0
0 0 0 1 0 1 0 0 1
0 0 1 1 1 1 1 1 1
0 1 1 0 0 1 0 0 1
1 1 0 1 0 1 0 1 1
1 0 0 0 1 1 1 1 0
Graph detour: 1 8 2 4 3 6 5 9 7
Path from 3 to 5: 3 6 5

Process finished with exit code 0
```

Як бачимо, обхід графу має наступний вигляд:
1->8 ->2 ->4 ->3 ->6 ->5 ->9 ->7

Шлях від 3-ї вершини до 5-ї має такий вигляд:
3->6->5

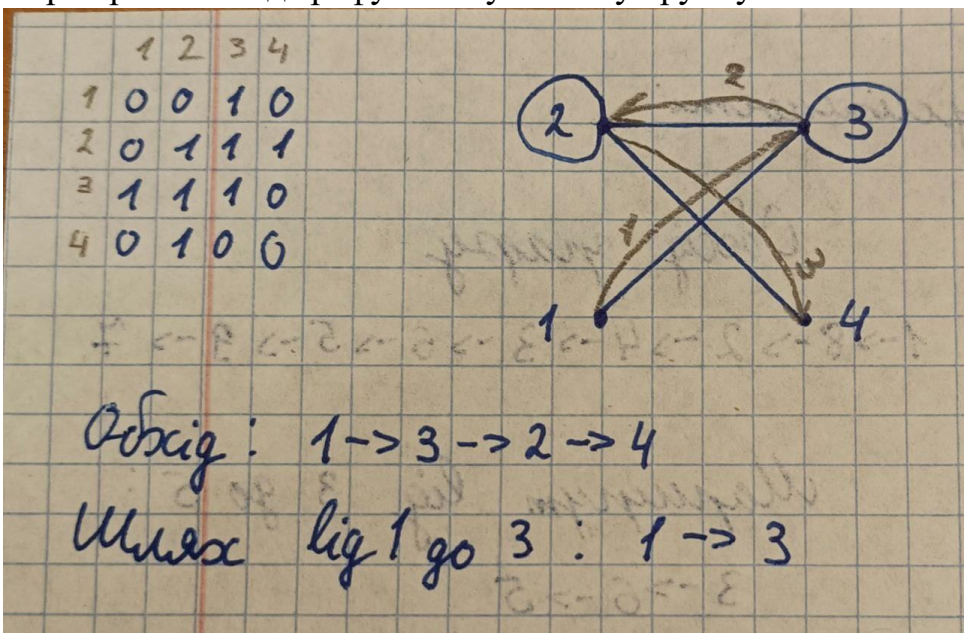
Результати збіглись, отже, програма правильно виконала обхід графу і правильно знайшла маршрут.

Для додаткової перевірки, згенеруємо граф випадковим чином:

```
Enter graph size:4
 Would you like enter graph manually or generate it randomly?
M - Manually, R - Randomly:r
 Enter begin vertex:1
 Enter end vertex:3
 Adjacency matrix of graph:
0 0 1 0
0 1 1 1
1 1 1 0
0 1 0 0
Graph detour: 1 3 2 4
Path from 1 to 3: 1 3

Process finished with exit code 0
```

Перевіримо обхід графу і пошук шляху вручну:



Бачимо, що результати збіглись, отже, програма правильно виконала обхід графу і правильно знайшла маршрут.

Висновок

Під час виконання цієї лабораторної роботи я вивчив основні прикладні алгоритми на графах та способи їх імплементації. В результаті виконання лабораторної роботи я отримав програму, яка може знаходити маршрут у графі методом Террі. Під час тестування я переконався, що мої розв'язки задач і розв'язки програми збіглись, отже, вона працює правильно.