Міністерство освіти і науки України
Центральноукраїнський національний технічний університет
Механіко-технологічний факультет
Кафедра кібербезпеки та програмного забезпечення
Дисципліна: Алгоритми та структури даних

Лабораторна робота №2

Тема: «АЛГОРИТМИ РОБОТИ З ГРАФАМИ. АЛГОРИТМ ДЕЙКСТРИ»

Виконав: ст. гр. КН-24
Куріщенко П. В.
Перевірив: викладач
Константинова Л. В.

Кропивницький 2025

Мета: Навчитися реалізовувати алгоритм Дейкстри для пошуку найкоротшого шляху.

Варіант - 14

Завдання: Реалізувати програмно абстрактний тип даних неорієнтований граф. Створити його екземпляр, який заповнити даними, вказаними у Вашому варіанті. Розробити програму, що повинна визначити та вивести на екран найкоротший шлях від вершини 1 до вершини 6 за допомогою алгоритму Дейкстри.

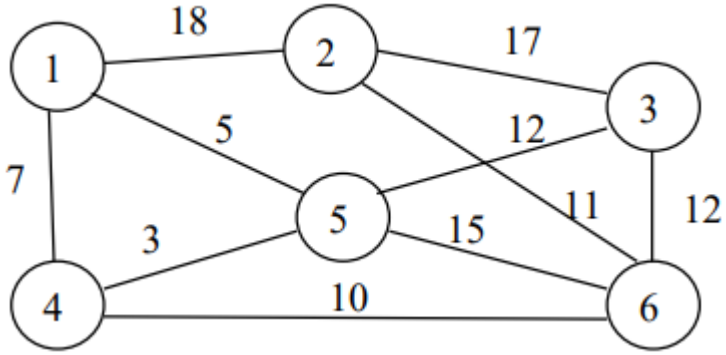

Рис. 1 – приклад неорієнтованого графу

Лістинг *graph.h*:

```
#ifndef GRAPH_H
#define GRAPH_H

#include <vector>
#include <iostream>

using namespace std;

struct Edge {
    int startVertex;
    int endVertex;
    int weight;
};

class Graph {
public:
    Graph(int vertices);

    void addEdge(int start, int end, int weight);

    void kruskalMST();
    void dijkstra(int start, int end);
```

```
private:
    int vertexAmount;
    vector<Edge> edges;
    vector<vector<pair<int,int>>> adjacent;

    //helpers
    int find(int vertices, vector<int>& parent);
    void unite(int first, int second, vector<int>& parent);
    static bool compareEdges(const Edge&, const Edge&);
};

#endif // GRAPH_H
```

Лістинг *graph.cpp*:

```
#include "graph.h"

Graph::Graph(int vertices) : vertexAmount(vertices) {
adjacent.resize(vertexAmount + 1); }

void Graph::addEdge(int start, int end, int weight) {
    edges.push_back({start, end, weight});

    adjacent[start].push_back({end, weight});
    adjacent[end].push_back({start, weight});
}

int Graph::find(int vertices, std::vector<int>& parent) {
    if (parent[vertices] == vertices) return vertices;
    return parent[vertices] = find(parent[vertices], parent);
}

void Graph::unite(int first, int second, std::vector<int>& parent) {
    first = find(first, parent);
    second = find(second, parent);
    if (first != second) parent[second] = first;
}

bool Graph::compareEdges(const Edge& first, const Edge& second) { return
first.weight < second.weight; }

void Graph::kruskalMST() {
    sort(edges.begin(), edges.end(), Graph::compareEdges);

    vector<int> parent(vertexAmount + 1);
    for (int index = 1; index <= vertexAmount; index++)
        parent[index] = index;

    int totalWeight = 0;
    int edgeCount = 0;
    cout << "Мінімальне каркасне дерево:\n";

    for (const Edge& edge : edges) {
        if (find(edge.startVertex, parent) != find(edge.endVertex, parent)) {
            unite(edge.startVertex, edge.endVertex, parent);
            totalWeight += edge.weight;
            cout << edge.startVertex << " <-(" << edge.weight << ")-> " <<
edge.endVertex << endl;
```

```
            edgeCount++;
            if (edgeCount == vertexAmount - 1) break;
        }
    }

    cout << "Сумарна вага: " << totalWeight << endl;
}

void Graph::dijkstra(int start, int end) {
    int size = vertexAmount + 1;

    vector<int> dist(size, INT_MAX);
    vector<bool> visited(size, false);
    vector<int> previous(size, -1);

    dist[start] = 0;

    for (int count = 0; count < vertexAmount; count++) {
        int currentVertex = -1;
        int minDistance = INT_MAX;

        for (int vertex = 1; vertex <= vertexAmount; vertex++) {
            if (!visited[vertex] && dist[vertex] < minDistance) {
                minDistance = dist[vertex];
                currentVertex = vertex;
            }
        }

        if (currentVertex == -1) break;
        visited[currentVertex] = true;

        for (const auto& [neighborVertex, weight] : adjacent[currentVertex])
{
            if (!visited[neighborVertex] && dist[currentVertex] + weight <
dist[neighborVertex]) {
                dist[neighborVertex] = dist[currentVertex] + weight;
                previous[neighborVertex] = currentVertex;
            }
        }
    }

    if (dist[end] == INT_MAX) {
        cout << "Шляху не існує;(" << endl;
        return;
    }

    vector<int> path;
    for (int vertex = end; vertex != -1; vertex = previous[vertex])
path.push_back(vertex);
    reverse(path.begin(), path.end());

    cout << "Найкоротший шлях від " << start << " до " << end << " = " <<
dist[end] << endl;
    cout << "Шлях: ";
    for (int vertex : path) cout << vertex << " ";
    cout << endl;
}
```

Лістинг *main.cpp*:

```
#include "graph.h"

int main() {
    Graph graph(6);

    vector<Edge> edges = {
        {1, 2, 18},
        {2, 3, 17},
        {3, 5, 12},
        {3, 6, 12},
        {5, 1, 5},
        {5, 6, 15},
        {5, 4, 3},
        {4, 6, 10},
        {6, 2, 11},
        {1, 4, 7}
    };

    for (const Edge& edge : edges)
        graph.addEdge(edge.startVertex, edge.endVertex, edge.weight);

    //graph.kruskalMST();
    graph.dijkstra(1, 6);
}
```

Результат виконання:

```
Найкоротший шлях від 1 до 6 = 17
Шлях: 1 4 6
Натисніть <ENTER>, щоб закрити це вікно...
```

Висновок: У ході виконання роботи було реалізовано алгоритм Дейкстри для пошуку найкоротшого шляху в неорієнтованому зваженому графі. Досліджено принцип його роботи, зокрема вибір вершини з мінімальною поточною відстанню та оновлення (релаксацію) відстаней до сусідніх вершин. У результаті програма коректно визначає довжину найкоротшого шляху між заданими вершинами та відтворює сам маршрут.