Міністерство освіти і науки України
ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ імені ІВАНА ФРАНКА
Факультет прикладної математики та інформатики

Кафедра програмування

ЛАБОРАТОРНА РОБОТА № 10

Графи, алгоритм Дейкстри

з курсу “Алгоритми та структури даних”

Виконав:

Студент групи ПМІ-16

Бевз Маркіян Юрійович

Львів – 2024

Мета: дослідити концепцію графів та алгоритм Дейкстри, який використовується для пошуку найкоротшого шляху в графі, написати власну реалізацію алгоритму Дейкстри.

Теоретичні відомості:

Граф — це сукупність об'єктів із зв'язками між ними**.**

Об'єкти розглядаються як **вершини**, або **вузли** графу, а зв'язки — як **дуги**, або **ребра**. Для різних галузей види графів можуть відрізнятися орієнтованістю, обмеженнями на кількість зв'язків і додатковими даними про вершини або ребра.

Ребра графу можуть бути напрямленими або ненапрямленими. Наприклад, якщо вершини будуть представляти людей на вечірці, й існуватиме ребро між двома людьми, якщо вони потиснули руки, тоді ребра цього графу не матимуть напряму, оскільки будь-яка особа A може потиснути руки із особою B лише якщо B також потисне руки із A. На противагу цьому, якщо будь-яке ребро від особи *A* до особи *B* означатиме, що особі A подобається B, то ребра матимуть напрям, оскільки таке вподобання не обов'язково буде взаємним. Граф першого типу називається **неорієнтованим графом**, а ребра в свою чергу — **неорієнтов аними ребрами**, тоді як граф другого типу називається **орієнтованим графом** і ребра — **орієнтованими ребрами** або **дугами.**

Велика кількість структур, які мають практичну цінність у математиці та інформатиці, можуть бути подані графами.

Існують різні види графів:

1. Орієнтовані графи: ребра мають напрямок.
2. Неорієнтовані графи: ребра не мають напрямку.
3. Зважені графи: кожному ребру призначено ваговий коефіцієнт або вартість.
4. Не зважені графи: ребра не мають ваги.

Також перед тим як почати реалізувати алгоритм Дейкстри, варто розібратися яким чином можна зберігати графи у **пам’яті комп’ютера:**

1. Матриця суміжності

У цьому методі граф зберігається у вигляді двовимірної матриці, де рядки та стовпці відповідають вершинам графа. Значення в комірках матриці вказують на наявність або відсутність ребра між парою вершин, або ж можуть вказувати вагу ребра. Цей метод ефективний для невеликих графів, але може бути недоцільним для графів з великою кількістю вершин, оскільки матриця займає квадратичний об'єм пам'яті.

2. Список суміжності

У цьому методі кожна вершина графа має список суміжних вершин, з якими вона пов'язана. Для кожної вершини зберігається список суміжних вершин або пар (вершина, вага ребра). Цей метод ефективний для розріджених графів, де кількість ребер значно менша за кількість вершин.

3. Список ребер

У цьому методі граф зберігається у вигляді списку ребер, де кожне ребро представлене парою вершин, які воно з'єднує, а також може мати вагу. Цей метод дозволяє ефективно працювати з графами з великою кількістю ребер, але може виявитися менш ефективним для операцій пошуку суміжних вершин.

У свій реалізації алгоритму Дейкстри я використовую представлення графа у вигляді списку суміжності. У реалізації використовується вектор **adjList**, що є масивом векторів, де кожен елемент масиву відповідає вершині графа, а вектор, що знаходиться в цьому елементі, містить пари (вага, вершина), які представляють суміжні вершини та вагу ребра.

Це ефективний метод зберігання графа для багатьох випадків, особливо для розріджених графів, де кількість ребер значно менша за кількість вершин. Крім того, цей метод дозволяє легко отримувати доступ до суміжних вершин та їхніх ваг, що дуже важливо для виконання алгоритму Дейкстри та інших алгоритмів пошуку найкоротшого шляху, а тепер про сам **алгоритм Дейкстри**.

Алгоритм Дейкстри - це алгоритм пошуку найкоротшого шляху в графі з невід'ємними вагами ребер від однієї початкової вершини до всіх інших вершин. Основна ідея полягає в тому, щоб поступово оновлювати оцінки найкоротших відстаней до всіх вершин графа, починаючи з початкової вершини.

Принцип роботи алгоритму Дейкстри:

1. **Ініціалізація**: Усі вершини графа помічаються як недосяжні, за винятком початкової вершини. Відстані до всіх інших вершин ініціалізуються як нескінченні, а відстань до початкової вершини - як 0.
2. **Створення черги пріоритетів**: Створюється черга пріоритетів, де вершини зберігаються за їхніми поточними найкращими оцінками відстаней. Початкова вершина додається до черги з оцінкою 0.
3. **Оновлення відстаней**: Поки черга пріоритетів не пуста, вибирається вершина з найменшою оцінкою відстані. Для цієї вершини оновлюються відстані до всіх суміжних вершин, якщо нова оцінка відстані є меншою за поточну.

4. **Повторення**: Крок оновлення відстаней до суміжних вершин повторюється, поки у черзі пріоритетів залишаються вершини або поки не буде досягнуто всіх вершин.

5. **Вихід**: Після завершення алгоритму маємо найкоротші відстані від початкової вершини до всіх інших вершин графа.

Хід роботи: Після вивчення теоретичної частини(а саме способів зберігання графів у пам’яті комп’ютера) та принципу роботи алгоритму Дейкстри я реалізував сам граф списком суміжності та алгоритм Дейкстри. Також було створено 5 тестів для перевірки правильності роботи функції у різних випадках. Нижче буде прикріплено принцип роботи мого коду, результат виконання програми, тестів, та їх код.

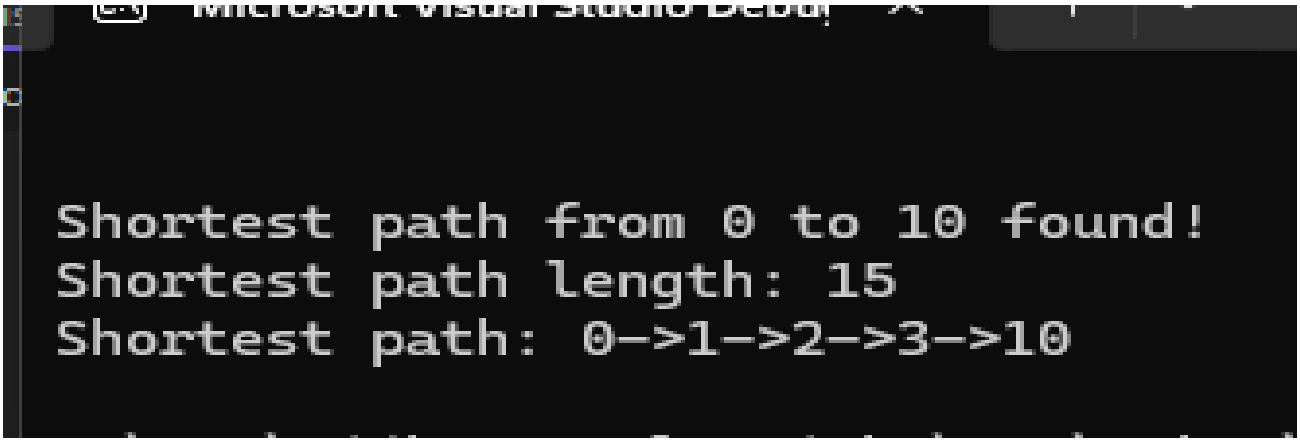
Ось так виглядає задання графа:

```cpp
#include "Dijkstra.h"

int main() {
    /* ... */
    int n_vertices = 11; // кількість вершин
    std::vector<std::tuple<int, int, int>> edges;
    edges.emplace_back(0, 1, 2);
    edges.emplace_back(0, 6, 10);
    edges.emplace_back(1, 4, 1);
    edges.emplace_back(1, 2, 8);
    edges.emplace_back(2, 3, 4);
    edges.emplace_back(2, 5, 6);
    edges.emplace_back(3, 5, 5);
    edges.emplace_back(3, 10, 1);
    edges.emplace_back(4, 5, 7);
    edges.emplace_back(4, 6, 4);
    edges.emplace_back(4, 7, 5);
    edges.emplace_back(4, 8, 3);
    edges.emplace_back(5, 9, 3);
    edges.emplace_back(6, 7, 6);
    edges.emplace_back(7, 8, 2);
    edges.emplace_back(8, 9, 4);
    edges.emplace_back(9, 10, 7);
    /* ... */
    int start_vertex = 0;
    int end_vertex = 10;
    try {
        ShortestPathInfo shortest_path_info = dijkstra(n_vertices, edges, start_vertex, end_vertex);

    std::cout << "Shortest path length: " << shortest_path_info.length << std::endl;
    std::cout << "Shortest path: ";
    for (int i = 0; i < shortest_path_info.path.size() - 1; ++i) {
        std::cout << shortest_path_info.path[i] << "->";
    }
    std::cout << shortest_path_info.path.back() << std::endl;
    }
    catch (std::exception& e)
    {
        std::cerr << "Run time error: " << e.what();
    }
    return 0;
}
```

Далі, як і зазначив, будуть іти тести та результат виконання програми для цього графа, це завдання було взято з підручника Юрія Миколайовича Щербини, отож прикріплю його.



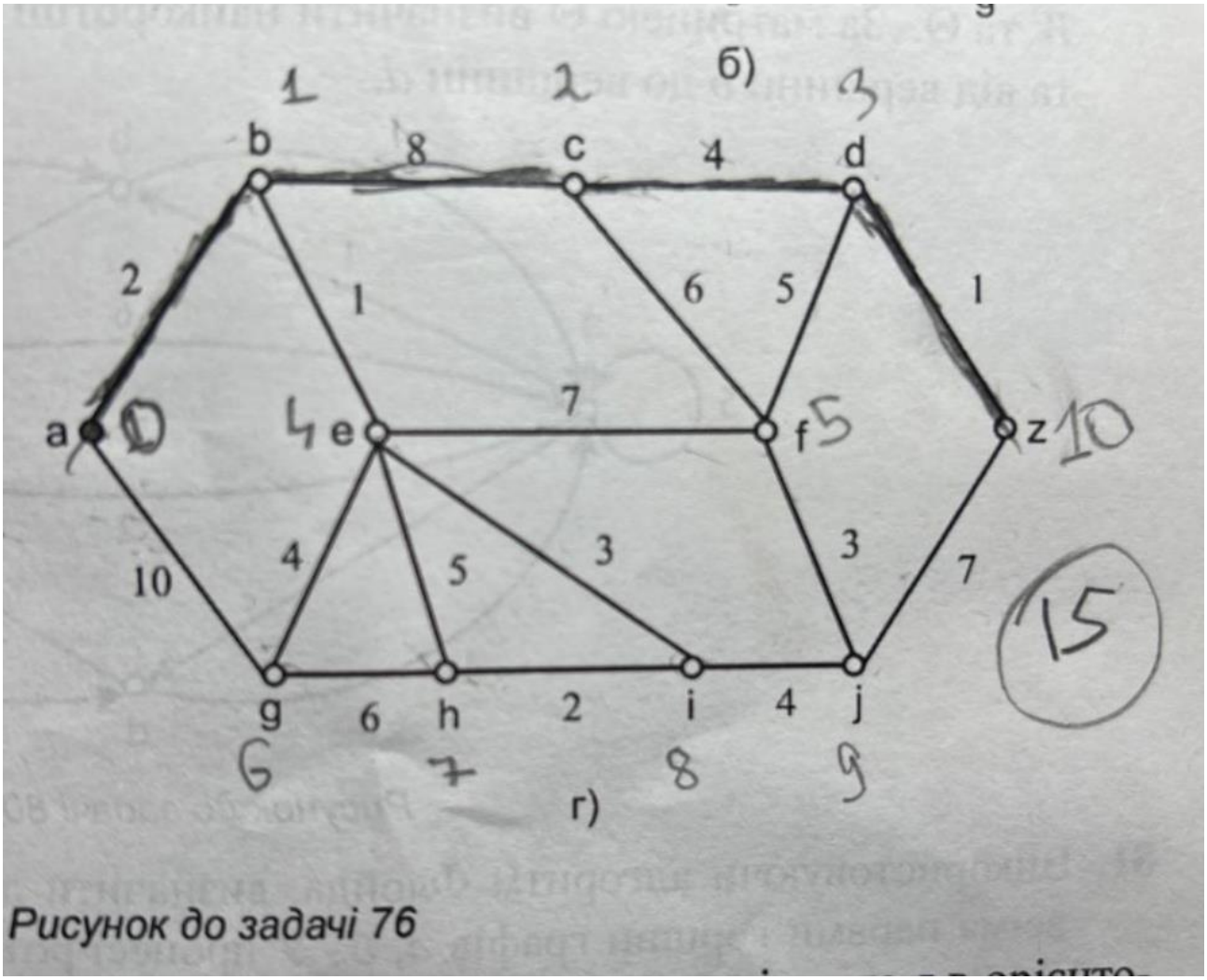
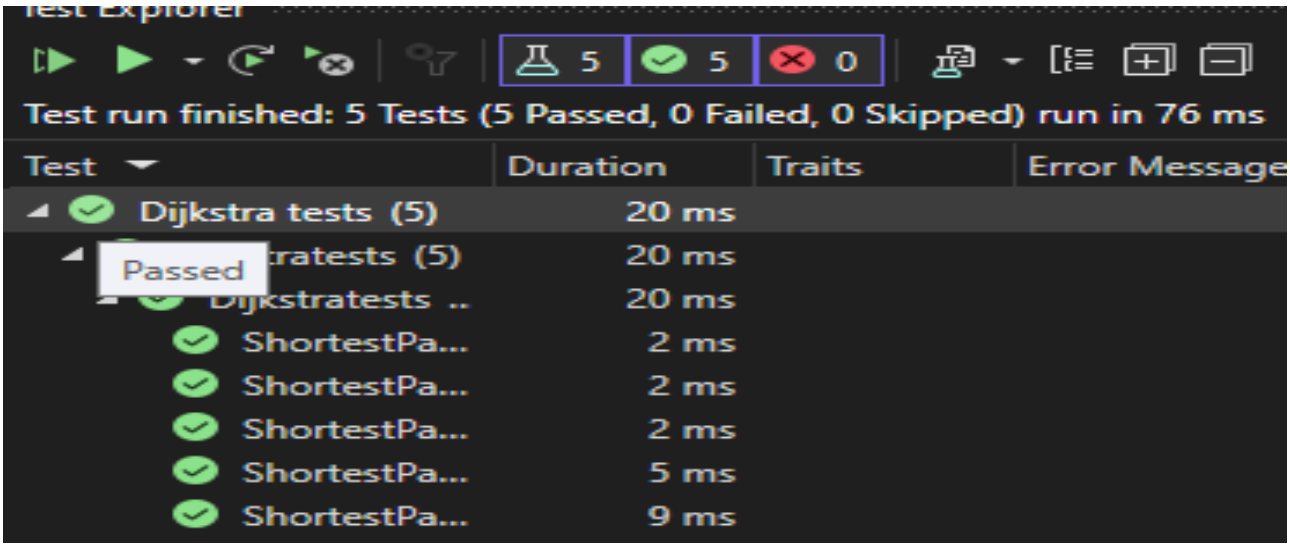


Рисунок до задачі 76




```cpp
TEST_CLASS(Dijkstratests)
{
public:

    TEST_METHOD(ShortestPathFound1)
    {

        std::vector<int> expected_path = { 0, 1, 3, 4 };
        int n_vertices = 11; // кількість вершин
        std::vector<std::tuple<int, int, int>> edges;
        edges.emplace_back(0, 1, 5);
        edges.emplace_back(1, 2, 3);
        edges.emplace_back(1, 3, 2);
        edges.emplace_back(2, 3, 2);
        edges.emplace_back(3, 4, 1);
        edges.emplace_back(4, 0, 24);
        int start_vertex = 0;
        int end_vertex = 4;
        ShortestPathInfo shortest_path_info = dijkstra(n_vertices, edges, start_vertex, end_vertex);
        Assert::AreEqual(shortest_path_info.length, 8);
        Assert::IsTrue(std::equal(shortest_path_info.path.begin(), shortest_path_info.path.end(), expected_path.begin()));


    }
    TEST_METHOD(ShortestPathFound2)
    {

        std::vector<int> expected_path = { 0, 1, 2, 3, 10 };
        int n_vertices = 11; // кількість вершин
        std::vector<std::tuple<int, int, int>> edges;
        edges.emplace_back(0, 1, 2);
        edges.emplace_back(0, 6, 10);
        edges.emplace_back(1, 4, 1);
        edges.emplace_back(1, 2, 8);
        edges.emplace_back(2, 3, 4);
        edges.emplace_back(2, 5, 6);
        edges.emplace_back(3, 5, 5);
        edges.emplace_back(3, 10, 1);
        edges.emplace_back(4, 5, 7);
        edges.emplace_back(4, 6, 4);
        edges.emplace_back(4, 7, 5);
        edges.emplace_back(4, 8, 3);
        edges.emplace_back(5, 9, 3);
        edges.emplace_back(6, 7, 6);
        edges.emplace_back(7, 8, 2);
        edges.emplace_back(8, 9, 4);
        edges.emplace_back(9, 10, 7);
        int start_vertex = 0;
```

```
    int end_vertex = 10;
    ShortestPathInfo shortest_path_info = dijkstra(n_vertices, edges, start_vertex, end_vertex);
    Assert::AreEqual(shortest_path_info.length, 15);
    Assert::IsTrue(std::equal(shortest_path_info.path.begin(), shortest_path_info.path.end(), expected_path.begin()));


}
TEST_METHOD(ShortestPathFound3)
{

    std::vector<int> expected_path = { 0, 2, 3, 4 , 6};
    int n_vertices = 7; // кількість вершин
    std::vector<std::tuple<int, int, int>> edges;
    edges.emplace_back(0, 1, 3);
    edges.emplace_back(0, 2, 2);
    edges.emplace_back(0, 3, 5);
    edges.emplace_back(1, 4, 5);
    edges.emplace_back(2, 3, 2);
    edges.emplace_back(2, 5, 3);
    edges.emplace_back(3, 4, 3);
    edges.emplace_back(3, 6, 7);
    edges.emplace_back(4, 6, 1);
    edges.emplace_back(5, 6, 4);
    int start_vertex = 0;
    int end_vertex = 6;
    ShortestPathInfo shortest_path_info = dijkstra(n_vertices, edges, start_vertex, end_vertex);
    Assert::AreEqual(shortest_path_info.length, 8);
    Assert::IsTrue(std::equal(shortest_path_info.path.begin(), shortest_path_info.path.end(), expected_path.begin()));


}
```

```
    TEST_METHOD(ShortestPathNotFound_DisconnectedGraph)
    {
        int n_vertices = 5;
        std::vector<std::tuple<int, int, int>> edges;
        edges.emplace_back(0, 1, 5);
        edges.emplace_back(1, 2, 3);
        edges.emplace_back(1, 1, 0);
        edges.emplace_back(3, 4, 1);
        // Цей граф має вершини 3 і 4, які не з'єднані з іншими вершинами
        int start_vertex = 0;
        int end_vertex = 4;
        ShortestPathInfo shortest_path_info = dijkstra(n_vertices, edges, start_vertex, end_vertex);
        Assert::AreEqual(shortest_path_info.length, -1); // Очікуємо, що шлях не знайдено
        Assert::IsTrue(shortest_path_info.path.empty()); // Очікуємо порожній шлях
    }
    TEST_METHOD(ShortestPathFound_GraphWithNegativeWeights)
    {
        std::vector<int> expected_path = { 0, 1, 3, 4 };
        int n_vertices = 5;
        std::vector<std::tuple<int, int, int>> edges;
        edges.emplace_back(0, 1, 5);
        edges.emplace_back(1, 2, -3); // Ребро з від'ємною вагою
        edges.emplace_back(1, 3, 2);
        edges.emplace_back(2, 3, 2);
        edges.emplace_back(3, 4, 1);
        edges.emplace_back(4, 0, 24);
        int start_vertex = 0;
        int end_vertex = 4;

        try {
            ShortestPathInfo shortest_path_info = dijkstra(n_vertices, edges, start_vertex, end_vertex);
            Assert::Fail(L"Expected an exception due to negative edge weight.");
        }
        catch (const std::runtime_error&) {
        }
    }
};
```

Висновок: На цій лабораторній роботі я дослідив детальніше концепцію графів та алгоритм Дейкстри, який використовується для пошуку найкоротшого шляху в графі та написав власну реалізацію алгоритму Дейкстри. Алгоритм Дейкстри є потужним інструментом для пошуку найкоротших шляхів

у графах з не від'ємними вагами ребер. Він застосовується в різних областях, таких як маршрутизація в комп'ютерних мережах, GPS-навігація, оптимізація транспортних маршрутів та багато інших задач, де необхідно знайти найкоротший шлях між вершинами графа.