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”

Виконала:

Студентка групи ПМІ-13

Демко Сніжана Іванівна

Львів – 2024

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

Графом G = (V, E) називають сукупність двох множин: скінченої непорожньої множини V вершин (vertex) і скінченої множини E ребер (edge), які з'єднують пари вершин. Ребра зображаються невпорядкованими парами вершин (u,v).

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.

Зображення графів у вигляді матриці суміжності

Одним з найчастіше використовуваних способів є зображення графів у вигляді різних матриць

Матрицею суміжності A(G) графа G з n вершинами називається квадратна матриця розміру n x n , елементи якої визначають так:

$$a_{ij} = \begin{cases} 1, & \text{якщо } (v_i, v_j) \in E, \text{ тобто вершини } v_i \text{ та } v_j \text{ суміжні} \\ 0 & \text{у протилежному випадку} \end{cases}$$

Очевидно, що така матриця може бути логічною, де істинне значення позначає наявність ребра, а хибне - його відсутність. Якщо граф неорієнтований, то матриця суміжності - симетрична відносно головної діагоналі. Наприклад, для графа з рис. 93, матриця суміжності матиме вигляд (див. наступна сторінка):

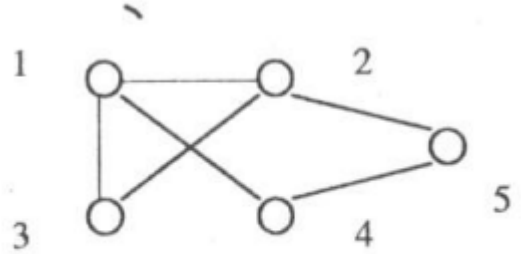


Рис. 93

	1	2	3	4	5
1	0	1	1	1	0
2	1	0	1	-0	1
3	1	1	0	0	0
4	1	0	0	0	1
5	0	1	0	1	0

Якщо ребра чи дуги графа мають певне навантаження, то для зображення графа з вагами можна використати модифіковану матрицю суміжності, елементи якої визначають за таким правилом

$$a_{ij} = \begin{cases} w_{ij}, & \text{якщо } (v_i, v_j) \in E, \text{ а } w_{ij} \text{ вага дуги } (v_i, v_j) \\ 0 & \text{у протилежному випадку} \end{cases}$$

Значення 0 - це значення, яке не позначає ваги, а вказує на відсутність дуги між вершинами. На рис. 96 зображено навантажений орієнтований граф (навантаження виділено). Матриця суміжності у цьому випадку виглядатиме так:

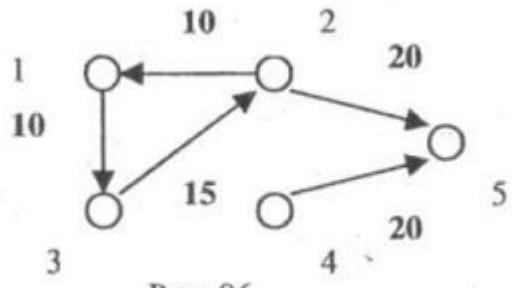


Рис. 96

	1	2	3	4	5
1	0	0	10	0	0
2	10	0	0	0	20
3	0	15	0	0	0
4	0	0	0	0	20
5	0	0	0	0	0

Основною перевагою матриці суміжності є те, що використовуючи прямий доступ до елементів матриці, за один крок (з константною оцінкою) можна дати відповідь на питання, чи є дві вершини суміжними (чи існує між ними ребро).

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:

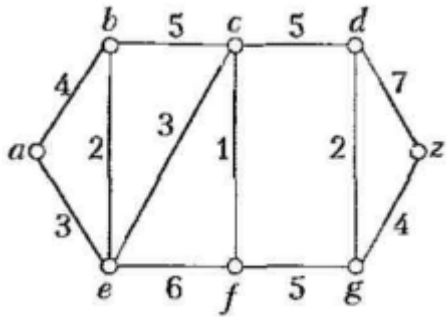
1. **Ініціалізація:** Всі вершини крім початкової мають нескінченну відстань. Відстань від початкової вершини до неї самої дорівнює 0.
2. **Обробка вершин:** Алгоритм обирає вершину з найменшою відстанню серед тих, які ще не були оброблені. Початкова вершина обирається першою.
3. **Оновлення відстаней:** Для кожної сусідньої вершини, яка ще не була оброблена, відстань до неї перераховується, як сума відстані до поточної вершини та ваги ребра між цією вершиною та поточною.
4. **Відзначення вершини:** Після обробки всіх сусідніх вершин, поточна вершина відзначається як оброблена.
5. **Повторення:** Ці кроки повторюються до тих пір, поки всі вершини не будуть оброблені.
6. **Завершення:** Після завершення алгоритму ми отримуємо найкоротші шляхи від початкової вершини до всіх інших вершин в графі.

Виконання:

1. **Розробка класу графу:** У першу чергу був розроблений клас Graph, який включає в себе основні методи для роботи з графом. У цьому класі реалізовано функції для додавання та видалення ребер, виведення матриці суміжності та перевірки наявності зв'язку між вершинами.
2. **Реалізація алгоритму Дейкстри:** Далі був створений клас Dijkstra, який містить реалізацію алгоритму Дейкстри. Цей алгоритм дозволяє знаходити найкоротший шлях в графі від заданої початкової вершини до всіх інших вершин. У класі реалізовані методи для ініціалізації, перерахунку довжини шляхів з позначеними вершинами, визначення найкоротшого шляху та заповнення шляху.
3. **Написання тестів:** Для перевірки правильності роботи класів були написані Google Tests. Тести перевіряють функціонал класів Graph та Dijkstra.

Результати роботи програмної реалізації алгоритму Дейкстри:

Завдання було взято з підручника “Дискретна математика”, Юрія Миколайовича Щербини, отож прикріплю фото завдання (вершини a,b,c… були перенумеровані як 0,1,2…)




```
Adjacency matrix

0 :     0       4       0       0       3       0       0       0
1 :     4       0       5       0       2       0       0       0
2 :     0       5       0       5       3       1       0       0
3 :     0       0       5       0       0       0       2       7
4 :     3       2       3       0       0       6       0       0
5 :     0       0       1       0       6       0       5       0
6 :     0       0       0       2       0       5       0       4
7 :     0       0       0       7       0       0       4       0


Shortest path length from vertex 0 to vertex 7: 16
Shortest path: 0 -> 4 -> 2 -> 5 -> 6 -> 7
```

Тестування

1. **Тест створення графу та додавання ребер:**

Перевіряє коректність додавання ребер до графу за допомогою функції addEdge.

```
TEST(GraphTest, AddEdge) {
    Graph graph(5);
    graph.addEdge(0, 1, 5);
    graph.addEdge(1, 2, 3);

    bool edge01 = graph.areConnected(0, 1);
    bool edge12 = graph.areConnected(1, 2);

    EXPECT_TRUE(edge01);
    EXPECT_TRUE(edge12);
}
```

2. **Тест видалення ребра:**

Перевіряє коректність видалення ребра з графу за допомогою функції removeEdge

```
TEST(GraphTest, RemoveEdge) {
    Graph graph(5);
    graph.addEdge(0, 1, 5);
    graph.addEdge(1, 2, 3);
    graph.removeEdge(0, 1);
    EXPECT_FALSE(graph.areConnected(0, 1));
}
```

3. **Тест знаходження найкоротшого шляху за допомогою алгоритму Дейкстри:**

Перевіряє, чи алгоритм Дейкстри правильно знаходить найкоротший шлях між двома вершинами у графі.

```
TEST(DijkstraTest, ShortestPath) {
    Graph graph(5);
    graph.addEdge(0, 1, 10);
    graph.addEdge(0, 2, 20);
    graph.addEdge(1, 2, 5);
    graph.addEdge(1, 3, 30);
    graph.addEdge(2, 3, 15);

    Dijkstra dijkstra(graph);
    vector<int> path;
    int shortestPathLength = dijkstra.findWay(0, 3, path);

    EXPECT_EQ(shortestPathLength, 30);
    EXPECT_EQ(path.size(), 4);
    EXPECT_EQ(path[0], 3);
    EXPECT_EQ(path[1], 2);
    EXPECT_EQ(path[2], 1);
}
```

4. **Тест перевірки кількості вершин:**

Перевіряє, чи правильно повертається кількість вершин у графі за допомогою функції getNumberOfVertexes.

```
TEST(GraphTest, GetNumberOfVertexes) {
    // Arrange
    Graph graph(10);

    // Act & Assert
    EXPECT_EQ(graph.getNumberOfVertexes(), 10);
}
```

5. **Тест перевірки з'єднаності вершин:**

Перевіряє, чи вершини з'єднані у графі за допомогою функції areConnected.

```
TEST(GraphTest, AreConnected) {
    Graph graph(5);
    graph.addEdge(0, 1, 10);
    graph.addEdge(1, 2, 5);
    graph.addEdge(2, 3, 7);
    graph.addEdge(3, 4, 3);
```

```
    EXPECT_TRUE(graph.areConnected(0, 1));
    EXPECT_TRUE(graph.areConnected(1, 2));
    EXPECT_TRUE(graph.areConnected(2, 3));
    EXPECT_TRUE(graph.areConnected(3, 4));

    EXPECT_FALSE(graph.areConnected(0, 2));
    EXPECT_FALSE(graph.areConnected(1, 3));
    EXPECT_FALSE(graph.areConnected(2, 4));
}
```

Результати тестів:

```
[==========] Running 5 tests from 2 test cases.
[----------] Global test environment set-up.
[----------] 4 tests from GraphTest
[ RUN      ] GraphTest.AddEdge
[       OK ] GraphTest.AddEdge (0 ms)
[ RUN      ] GraphTest.RemoveEdge
[       OK ] GraphTest.RemoveEdge (0 ms)
[ RUN      ] GraphTest.GetNumberOfVertexes
[       OK ] GraphTest.GetNumberOfVertexes (0 ms)
[ RUN      ] GraphTest.AreConnected
[       OK ] GraphTest.AreConnected (0 ms)
[----------] 4 tests from GraphTest (25 ms total)

[----------] 1 test from DijkstraTest
[ RUN      ] DijkstraTest.ShortestPath
[       OK ] DijkstraTest.ShortestPath (0 ms)
[----------] 1 test from DijkstraTest (8 ms total)

[----------] Global test environment tear-down
[==========] 5 tests from 2 test cases ran. (59 ms total)
[  PASSED  ] 5 tests.
```

Висновок: в ході виконання лабораторної роботи я ознайомилася з основами роботи з графами та реалізувала алгоритм Дейкстри для знаходження найкоротшого шляху в графі. Також були написані тести для перевірки коректності роботи програми. Лабораторна робота допомогла покращити розуміння алгоритму Дейкстри та його застосування в практичних задачах.