**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра Вычислительной техники**

отчет

**по лабораторной работе № 4**

**по дисциплине «Алгоритмы и структуры данных»**

Тема: **«Графы»**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3311 | Баймухамедов Р. Р. |  |
| Преподаватель | Манирагена В. |  |

Санкт-Петербург

2024

**Введение**

**Задание вариант 21:**

Вычисление матрицы расстояний между всеми парами вершин не­о­ри­ен­ти­ро­ван­ного графа с нагруженными рёбрами

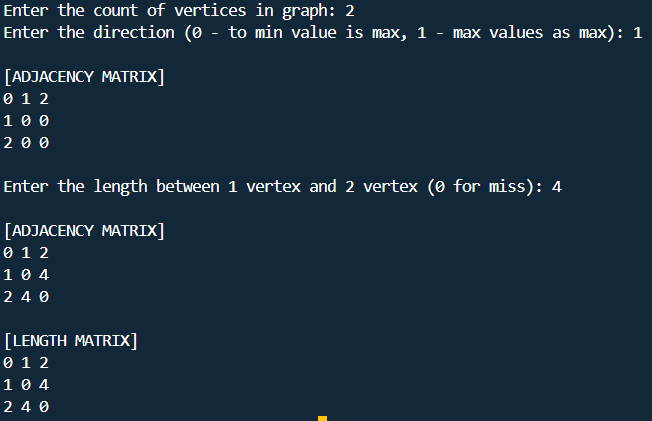
**Цель работы**

Разработать алгоритм для вычисления матрицы расстояний между всеми парами вершин в неориентированном графе с нагруженными рёбрами. Это необходимо для анализа связности графа, нахождения кратчайших путей между вершинами и решения задач на графах, таких как маршрутизация, оптимизация логистики и другие.

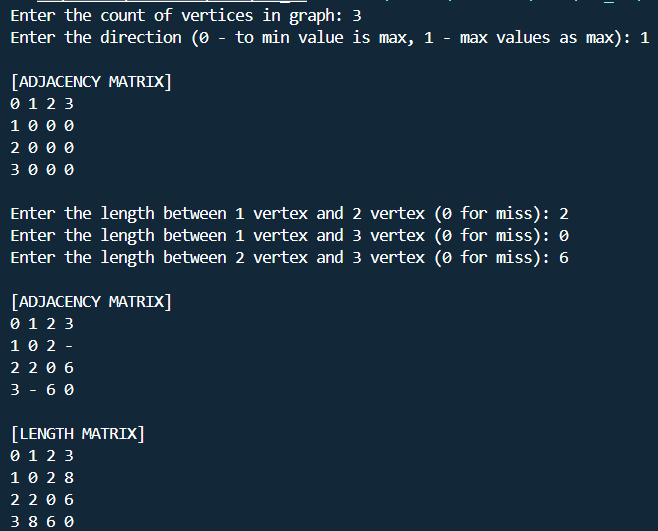
**Входные данные**

Подаётся количество вершин в графе и поиск максимальный или минимальный мы ищем (алгоритм работает в том числе и с отрицательными весами рёбер), затем, поскольку граф является неориентированным, то он является симметричным относительно главной диагонали матрицы смежности, достаточно заполнить только (n2)-n элементов, где n — ширина матрицы (или же длина матрицы, поскольку матрица смежности является квадратной). При вводе отображается значение между какими вершинами графа вводится, в качестве отсутствия пути между вершинами пишется “0”, который в отображаемой матрице выводится как знак “-”

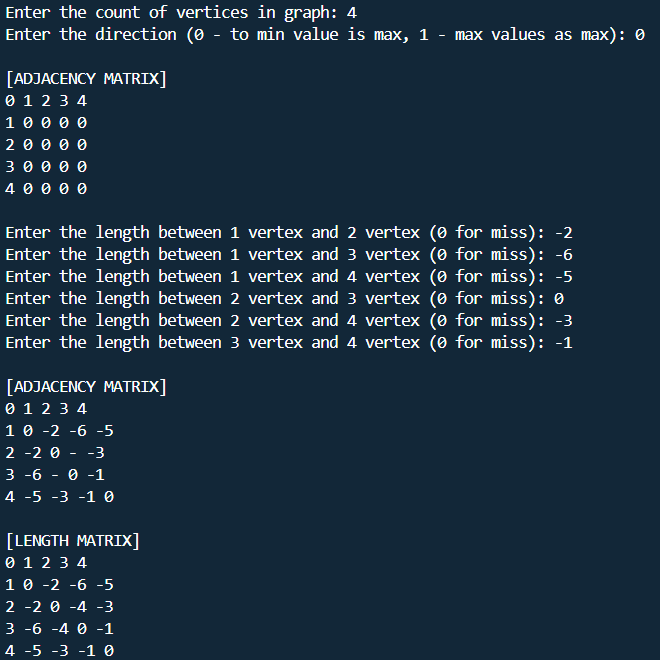
**Контрольные примеры**



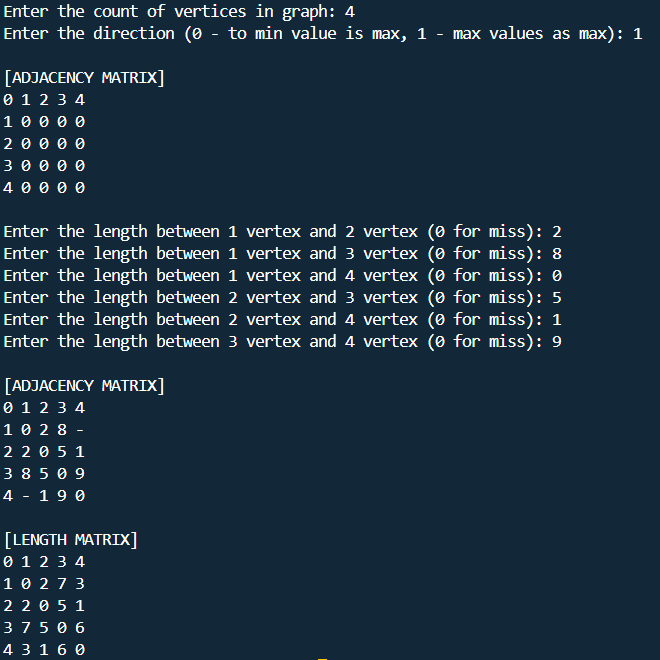
Пример



Пример



Пример



Пример

**Код программы**

#include <vector>

#include <iostream>

#include <limits>

*const* int INF = std::numeric\_limits<int>::max() / 2;

*const* int M\_INF = std::numeric\_limits<int>::min() / 2;

class *Graph* {

private:

    int count\_of\_elem;

    int direction;

    std::vector<std::vector<int>> matrix;

    std::vector<std::vector<int>> l\_matrix;

    void add\_marking() {

        for (int i = 0; i < matrix.size(); i++) {

            matrix[0][i] = i;

            matrix[i][0] = i;

        }

    }

public:

    Graph(int count\_of\_elem) : count\_of\_elem(count\_of\_elem) {

        matrix.resize(count\_of\_elem + 1, std::vector<int>(count\_of\_elem + 1));

        add\_marking();

    }

    int get\_count\_of\_elem() {

        return count\_of\_elem;

    }

    void set\_direction(int direction) {

        this->direction = direction;

    }

    void print\_graph\_matrix() {

        std::cout << "\n[ADJACENCY MATRIX]\n";

        for (int i = 0; i < matrix.size(); i++) {

            for (int j = 0; j < matrix.size(); j++) {

                if (abs(matrix[i][j]) > INF/4) std::cout << "- ";

                else std::cout << matrix[i][j] << " ";

            }

            std::cout << std::endl;

        }

    }

    void print\_l\_matrix() {

        std::cout << "\n[LENGTH MATRIX]\n";

        for (int i = 0; i < l\_matrix.size(); i++) {

            for (int j = 0; j < l\_matrix.size(); j++) {

                if (abs(l\_matrix[i][j]) > INF/4) std::cout << "- ";

                else std::cout << l\_matrix[i][j] << " ";

            }

            std::cout << std::endl;

        }

    }

    void set\_values() {

        int value;

        std::cout << std::endl;

        for (int i = 1; i < matrix.size(); i++) {

            for (int j = i + 1; j < matrix.size(); j++) {

                std::cout << "Enter the length between " << i << " vertex and " << j << " vertex (0 for miss): ";

                std::cin >> value;

                if (value == 0) {

                    if (direction == 0) {

                        matrix[i][j] = M\_INF;

                        matrix[j][i] = M\_INF;

                    } else {

                        matrix[i][j] = INF;

                        matrix[j][i] = INF;

                    }

                } else {

                    matrix[i][j] = value;

                    matrix[j][i] = value;

                }

            }

        }

    }

    void floyd\_algorithm() {

        l\_matrix = matrix;

        for (int i = 1; i < l\_matrix.size(); i++) {

            for (int j = 1; j < l\_matrix.size(); j++) {

                for (int k = 1; k < l\_matrix.size(); k++) {

                    if (direction == 0) {

                        if (l\_matrix[i][j] < l\_matrix[i][k] + l\_matrix[k][j]) {

                            l\_matrix[i][j] = l\_matrix[i][k] + l\_matrix[k][j];

                        }

                    } else {

                        if (l\_matrix[i][j] > l\_matrix[i][k] + l\_matrix[k][j]) {

                            l\_matrix[i][j] = l\_matrix[i][k] + l\_matrix[k][j];

                        }

                    }

                }

            }

        }

    }

};

int main() {

    int count\_of\_elem;

    int direction;

    std::cout << "Enter the count of vertices in graph: ";

    std::cin >> count\_of\_elem;

    std::cout << "Enter the direction (0 - to min value is max, 1 - max values as max): ";

    std::cin >> direction;

    if (count\_of\_elem <= 0) return -1;

*Graph* graph(count\_of\_elem);

    graph.set\_direction(direction);

    graph.print\_graph\_matrix();

    graph.set\_values();

    graph.print\_graph\_matrix();

    graph.floyd\_algorithm();

    graph.print\_l\_matrix();

    return 0;

}

**Заключение:**

В рамках данной работы была реализована задача вычисления матрицы расстояний между всеми парами вершин неориентированного графа с нагруженными рёбрами. Эта задача имеет важное практическое значение в анализе графовых структур, таких как транспортные сети, социальные графы, логистические цепочки и другие области, где необходимо учитывать связи между объектами и минимизировать пути или затраты.

Рассмотренный алгоритм Флойда–Уоршелла продемонстрировал свою эффективность для решения данной задачи благодаря своей простоте и универсальности. Несмотря на относительно высокую вычислительную сложность O(n3) алгоритм подходит для графов с умеренным количеством вершин, обеспечивая точное и надёжное нахождение кратчайших расстояний, а также затрачиваемой памятью O(n2)

Результаты выполнения алгоритма дают важную информацию, необходимую для принятия решений, например:

• Определение минимальных затрат на перемещение между пунктами в транспортной сети;

• Анализ связности и доступности узлов в графе;

• Оптимизация маршрутов в различных задачах логистики и коммуникаций.

Применение алгоритма также продемонстрировало важность выбора подходящих структур данных для хранения и обработки графа, что напрямую влияет на производительность программного решения. Использование матрицы смежности для хранения графа позволило существенно упростить реализацию и доступ к данным.

Данная работа подчёркивает значимость алгоритмов на графах в современном мире, где графы являются основой многих вычислительных систем. Результаты разработки и выполненной работы могут быть использованы для дальнейшего изучения более сложных методов оптимизации графов, таких как алгоритмы Дейкстры, Беллмана-Форда.

Таким образом, реализация и анализ предложенного решения позволяет сделать вывод, что разработанное приложение может эффективно использоваться для решения задач анализа Графовых структур и служит прочной основой для дальнейшего развития в области алгоритмов на графах.