МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»

РТУ МИРЭА

ОТЧЕТ
ПО ПРАКТИЧЕСКОЙ РАБОТЕ №5

Основные алгоритмы работы с графами
по дисциплине
«Структуры и алгоритмы обработки данных»

Выполнил студент группы ИКБО-01-21 Луковников Д.Р.

Принял преподаватель Туманова М.Б.

Практическая работа выполнена «__»_____2022 г. _____

«Зачтено» «__»_____2022 г. _____

Москва 2022

СОДЕРЖАНИЕ

ЦЕЛЬ РАБОТЫ

Составить программу создания графа и реализовать процедуру для работы с графом, определенную индивидуальным вариантом задания.

В программе предусмотреть ввод с клавиатуры произвольного графа. В вариантах построения основного дерева также разработать доступный способ (форму) вывода результирующего дерева на экран монитора.

Провести тестовый прогон программы на предложенном в индивидуальном варианте задания графе. Результаты тестирования в виде скриншотов экранов включить в отчет по выполненной работе.

Сделать выводы о проделанной работе, основанные на полученных результатах.

Оформить отчет с подробным описанием рассматриваемого графа, принципов программной реализации алгоритмов работы с графом, описанием текста исходного кода и проведенного тестирования программы.

ХОД РАБОТЫ

Индивидуальный вариант: Нахождение кратчайшего пути методом Йена, граф представлен на рисунке 1.

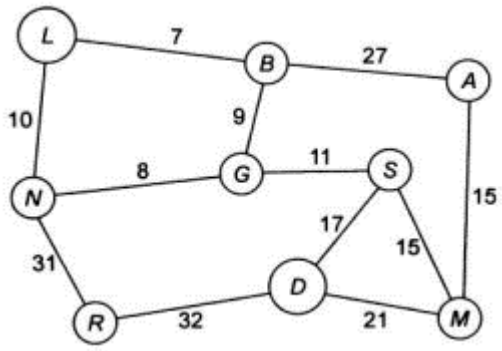

Рисунок 1 – Данный граф

Математическая модель решения: общая идея алгоритма представляет собой следующее, у нас будут кандидаты на кротчайшие пути. Находим первый кратчайший путь и так как ни один из последующих путей, не должен совпадать с данным, то все новые пути должны не включать в себя хотя бы одно ребро из первоначального. Поэтому исключаем по одному ребру из первого пути и находим кратчайшие пути в получаемых графах.

Листинг 1.1 – Структура узла

```
struct Node {
    /*
     * Структура, описывающая вершину графа
     */
    string name;
};
```

Граф хранится в виде матрицы смежности, можно было бы обойтись только ей, но по заданию вершины графа имеют «имена».

Листинг 1.2 – Структура графа

```
class Graph {
    /*
     * Класс, описывающий граф
     */

    int size; // Количество вершин графа
    vector<vector<int>> matrix; // Матрица смежности
    vector<Node *> nodes; // Список вершин графа
...
```

1.1 Добавление узла

При добавлении нового узла мы создаём новую объект узла, добавляем его в контейнер, увеличиваем количество узло и изменяем размер матрицы смежности.

Листинг 1.3 – Добавление узла

```
void addNode(string name) {
    /*
     * Добавление вершины в граф
     */
    Node *node = new Node;
    node->name = std::move(name);
    nodes.push_back(node);
    size++;
    matrix.resize(size);
    for (int i = 0; i < size; i++) {
        matrix[i].resize(size);
    }
}
```

1.2 Добавление ребра

При добавлении ребра или, иначе говоря, пути между двумя ребрами, нам нужны их названия и само значение дистанции, так же учитываем момент, что в нашем графе рёбра не имеют направления и двигаться по ним можно как в одну, так и в другую сторону, Листинг 1.4.

Листинг 1.4 – Добавление ребра

```
void addEdge(string name1, string name2, int distance) {
    /*
     * Добавление ребра в граф
     */
    int index1 = -1;
    int index2 = -1;
    for (int i = 0; i < size; i++) {
        if (nodes[i]->name == name1) {
            index1 = i;
        }
        if (nodes[i]->name == name2) {
            index2 = i;
        }
    }
    if (index1 == -1 || index2 == -1) {
        cout << "Error: node not found" << endl;
        return;
    }
    matrix[index1][index2] = distance;
    matrix[index2][index1] = distance;
}
```

1.3 Поиск кратчайшего пути

Для начала создаём 2 контейнера, в которых будет хранить расстояния от начально вершины до всех остальных, а также контейнер с уже посещёнными вершинами. И затем начинаем перебор начиная с начальной вершины, Листинг 1.5.

Листинг 1.5 – Поиск кратчайшего пути

```
int searchRoute(string name, string name2) {
    /*
     * Поиск кратчайшего пути между двумя вершинами
     */

    int index = getIndex(name); // Индекс начальной вершины
    int index2 = getIndex(name2); // Индекс конечной вершины
    if (index == -1 || index2 == -1) { // Если вершины не найдены
        cout << "Error: node not found" << endl;
        return -1;
    }

    // Массив, содержащий расстояния от начальной вершины до всех остальных
    vector<int> distances(size, -1);

    // Массив, содержащий индексы вершин, которые уже были просмотрены
    vector<int> visited(size, 0);
    distances[index] = 0;
    visited[index] = 1;
    int current = index; // Начинаем с начальной вершины

    // Пока не просмотрены все вершины
    while (current != index2) {

        // Просматриваем все ребра, исходящие из текущей вершины
        for (int i = 0; i < size; i++) {

            // Если ребро существует и вершина еще не была просмотрена
            if (matrix[current][i] != 0 && visited[i] == 0) {

                // Если расстояние до вершины еще не было вычислено, либо
вычисленное расстояние больше, чем расстояние через текущую вершину
                if (distances[i] == -1 || distances[i] > distances[current] +
matrix[current][i]) {

                    // Обновляем расстояние до вершины
                    distances[i] = distances[current] + matrix[current][i];
                }
            }
        }

        // Ищем вершину с минимальным расстоянием
        int min = -1;

        // Просматриваем все вершины
        for (int i = 0; i < size; i++) {

            // Если вершина еще не была просмотрена и расстояние до нее было
вычислено
            if (visited[i] == 0 && distances[i] != -1) {
```

Продолжение Листинга 1.5

```
                // Если минимальное расстояние еще не было вычислено, либо
вычисленное минимальное расстояние больше, чем расстояние до текущей вершины
                if (min == -1 || distances[i] < distances[min]) {
                    min = i;
                }
            }
        }

        // Если минимальное расстояние не было вычислено, значит путь не
существует
        if (min == -1) {
            break;
        }
        current = min;

        // Помечаем вершину как просмотренную
        visited[current] = 1;
    }

    return distances[index2];
}
```

1.4 Вывод графа

Вывод графа реализуем с помощью языка graphviz, Листинг 1.6.

Листинг 1.6 – Вывод графа

```
void print() {
    /*
     * Вывод графа на экран
     */

    vector<string> printedTo;
    vector<string> printedFrom;
    cout << "digraph G {" << endl;
    for (int i = 0; i < size; i++) {
        for (int j = i; j < size; j++) {
            if (matrix[i][j] != 0) {
                string from = nodes[i]->name;
                string to = nodes[j]->name;

                if (find(printedTo.begin(), printedTo.end(), to) ==
printedTo.end() ||
                    find(printedFrom.begin(), printedFrom.end(), from) ==
printedFrom.end()) {
                    cout << "    " << from << " -> " << to << " [label=\"" <<
matrix[i][j] << "\", arrowhead=none];" << endl;
                    printedTo.push_back(to);
                    printedFrom.push_back(from);
                }
            }
        }
    }
    cout << "}" << endl;
}
```

ТЕСТИРИВАНИЕ

Для начала введём граф и проверим его правильность, Листинг 2.1.

Листинг 2.1 – Вывод графа

```
int main() {
    Graph graph;
    graph.addNode("L");
    graph.addNode("B");
    graph.addNode("A");
    graph.addNode("N");
    graph.addNode("M");
    graph.addNode("G");
    graph.addNode("S");
    graph.addNode("R");
    graph.addNode("D");

    graph.addEdge("L", "B", 7);
    graph.addEdge("B", "A", 27);
    graph.addEdge("L", "N", 10);
    graph.addEdge("B", "G", 9);
    graph.addEdge("G", "S", 11);
    graph.addEdge("A", "M", 15);
    graph.addEdge("N", "G", 8);
    graph.addEdge("N", "R", 31);
    graph.addEdge("R", "D", 32);
    graph.addEdge("S", "D", 17);
    graph.addEdge("S", "M", 15);
    graph.addEdge("D", "M", 21);

    graph.print();
...
```

Построив граф сравним с данным графом в задании, Рисунки 1 и 2.

```
/Users/minusd/CLionProjects/siaod-tasks/cmake-build-debug/siaod_tasks
digraph G {
    L -> B [label="7", arrowhead=none];
    L -> N [label="10", arrowhead=none];
    B -> A [label="27", arrowhead=none];
    B -> G [label="9", arrowhead=none];
    A -> M [label="15", arrowhead=none];
    N -> G [label="8", arrowhead=none];
    N -> R [label="31", arrowhead=none];
    M -> S [label="15", arrowhead=none];
    M -> D [label="21", arrowhead=none];
    G -> S [label="11", arrowhead=none];
    S -> D [label="17", arrowhead=none];
    R -> D [label="32", arrowhead=none];
}
```

Рисунок 1 – Вывод программы

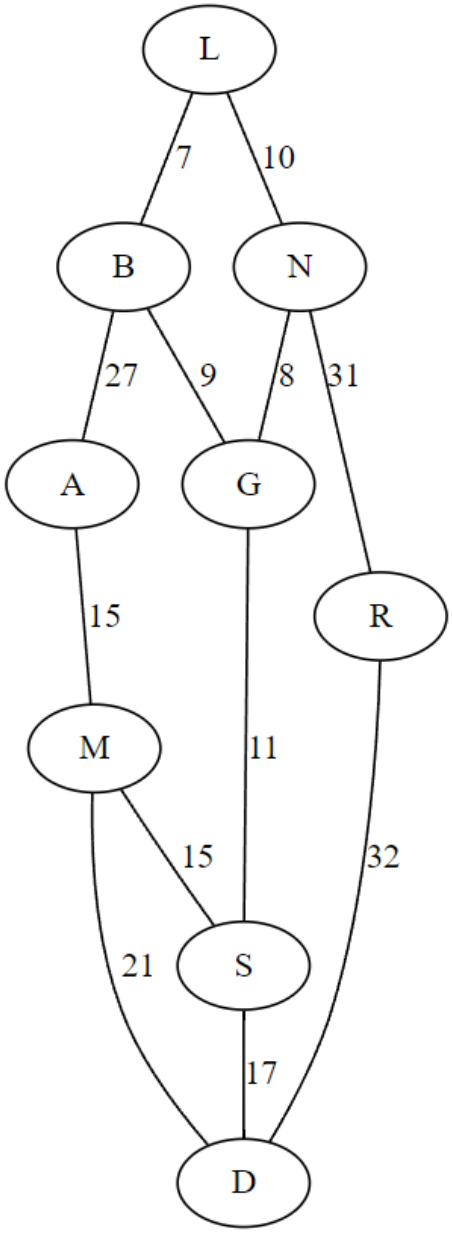

Рисунок 2 – Отрисовка графа

Для тестирования работоспособности алгоритма поиска, найдём минимальные расстояние между 2-я парами точек, Рисунок 3.

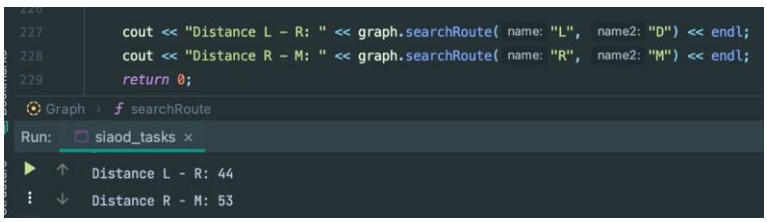

Рисунок 3 – Прямой обход

Проверив по графу, убедимся в правильности работы программы.

ВЫВОДЫ

При выполнении работы были получены навыки реализации алгоритма Йена для поиска кратчайшего пути между узлами графа. Программа была проверена на работоспособность и полностью протестирована.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Страуструп Б. Программирование. Принципы и практика с использованием C++. 2-е изд., 2016.
2. Документация по языку C++ [Электронный ресурс]. URL: https://docs.microsoft.com/ruru/cpp/cpp/ (дата обращения 01.09.2021).
3. Курс: Структуры и алгоритмы обработки данных. Часть 2 [Электронный ресурс]. URL: https://online-edu.mirea.ru/course/view.php?id=4020 (дата обращения 01.09.2021).

ПРИЛОЖЕНИЯ

Приложение А – Исходный код алгоритма Йена

Приложение А

Исходный код алгоритма Йена

Листинг 7.1 – main.cpp

```
/*
 * Составить программу создания графа и реализовать процедуру для работы с
 * графом, определенную индивидуальным вариантом задания.
 *
 * Самостоятельно выбрать и реализовать способ представления графа в памяти.
 * В программе предусмотреть ввод с клавиатуры произвольного графа. В
 * вариантах построения остовного дерева также разработать доступный способ
 * (форму) вывода результирующего дерева на экран монитора.
 *
 *
 * Нахождение кратчайшего пути методом Йена
 */

#include <iostream>
#include <utility>
#include <vector>

using namespace std;

struct Node {
    /*
     * Структура, описывающая вершину графа
     */
    string name;
};

class Graph {
    /*
     * Класс, описывающий граф
     */

    int size; // Количество вершин графа
    vector<vector<int>> matrix; // Матрица смежности
    vector<Node *> nodes; // Список вершин графа

public:
    Graph() {
        /*
         * Конструктор класса
         */
        size = 0;
    }

    void addNode(string name) {
        /*
         * Добавление вершины в граф
         */
        Node *node = new Node;
        node->name = std::move(name);
        nodes.push_back(node);
        size++;
        matrix.resize(size);
        for (int i = 0; i < size; i++) {
            matrix[i].resize(size);
        }
    }
```

Продолжение Листинг 7.1

```
    void addEdge(string name1, string name2, int distance) {
        /*
         * Добавление ребра в граф
         */
        int index1 = -1;
        int index2 = -1;
        for (int i = 0; i < size; i++) {
            if (nodes[i]->name == name1) {
                index1 = i;
            }
            if (nodes[i]->name == name2) {
                index2 = i;
            }
        }
        if (index1 == -1 || index2 == -1) {
            cout << "Error: node not found" << endl;
            return;
        }
        matrix[index1][index2] = distance;
        matrix[index2][index1] = distance;
    }

    Node *getNode(string name) {
        /*
         * Получение вершины по имени
         */
        for (int i = 0; i < size; i++) {
            if (nodes[i]->name == name) {
                return nodes[i];
            }
        }
        return nullptr;
    }

    int getIndex(string name) {
        /*
         * Получение индекса вершины по имени
         */
        for (int i = 0; i < size; i++) {
            if (nodes[i]->name == name) {
                return i;
            }
        }
        return -1;
    }

    void print() {
        /*
         * Вывод графа на экран
         */

        vector<string> printedTo;
        vector<string> printedFrom;
        cout << "digraph G {" << endl;
        for (int i = 0; i < size; i++) {
            for (int j = i; j < size; j++) {
                if (matrix[i][j] != 0) {
                    string from = nodes[i]->name;
                    string to = nodes[j]->name;
```

Продолжение Листинг 7.1

```
                    if (find(printedTo.begin(), printedTo.end(), to) ==
printedTo.end() ||
                        find(printedFrom.begin(), printedFrom.end(), from) ==
printedFrom.end()) {
                        cout << "    " << from << " -> " << to << "
[label=\"" << matrix[i][j] << "\", arrowhead=none];" << endl;
                        printedTo.push_back(to);
                        printedFrom.push_back(from);
                    }
                }
            }
        }
        cout << "}" << endl;
    }

    int searchRoute(string name, string name2) {
        /*
         * Поиск кратчайшего пути между двумя вершинами
         */

        int index = getIndex(name); // Индекс начальной вершины
        int index2 = getIndex(name2); // Индекс конечной вершины
        if (index == -1 || index2 == -1) { // Если вершины не найдены
            cout << "Error: node not found" << endl;
            return -1;
        }

        // Массив, содержащий расстояния от начальной вершины до всех
остальных
        vector<int> distances(size, -1);

        // Массив, содержащий индексы вершин, которые уже были просмотрены
        vector<int> visited(size, 0);
        distances[index] = 0;
        visited[index] = 1;
        int current = index; // Начинаем с начальной вершины

        // Пока не просмотрены все вершины
        while (current != index2) {

            // Просматриваем все ребра, исходящие из текущей вершины
            for (int i = 0; i < size; i++) {

                // Если ребро существует и вершина еще не была просмотрена
                if (matrix[current][i] != 0 && visited[i] == 0) {

                    // Если расстояние до вершины еще не было вычислено, либо
вычисленное расстояние больше, чем расстояние через текущую вершину
                    if (distances[i] == -1 || distances[i] >
distances[current] + matrix[current][i]) {

                        // Обновляем расстояние до вершины
                        distances[i] = distances[current] +
matrix[current][i];
                    }
                }
            }

            // Ищем вершину с минимальным расстоянием
            int min = -1;
```

Продолжение Листинг 7.1

```
            // Просматриваем все вершины
            for (int i = 0; i < size; i++) {

                // Если вершина еще не была просмотрена и расстояние до нее
было вычислено
                if (visited[i] == 0 && distances[i] != -1) {

                    // Если минимальное расстояние еще не было вычислено,
либо вычисленное минимальное расстояние больше, чем расстояние до текущей
вершины
                    if (min == -1 || distances[i] < distances[min]) {
                        min = i;
                    }
                }
            }

            // Если минимальное расстояние не было вычислено, значит путь не
существует
            if (min == -1) {
                break;
            }
            current = min;
            // Помечаем вершину как просмотренную
            visited[current] = 1;
        }
        return distances[index2];
    }
};

int main() {
    Graph graph;
    graph.addNode("L");
    graph.addNode("B");
    graph.addNode("A");
    graph.addNode("N");
    graph.addNode("M");
    graph.addNode("G");
    graph.addNode("S");
    graph.addNode("R");
    graph.addNode("D");

    graph.addEdge("L", "B", 7);
    graph.addEdge("B", "A", 27);
    graph.addEdge("L", "N", 10);
    graph.addEdge("B", "G", 9);
    graph.addEdge("G", "S", 11);
    graph.addEdge("A", "M", 15);
    graph.addEdge("N", "G", 8);
    graph.addEdge("N", "R", 31);
    graph.addEdge("R", "D", 32);
    graph.addEdge("S", "D", 17);
    graph.addEdge("S", "M", 15);
    graph.addEdge("D", "M", 21);

    graph.print();

    cout << "Distance L - R: " << graph.searchRoute("L", "D") << endl;
    cout << "Distance R - M: " << graph.searchRoute("R", "M") << endl;
    return 0;
}
```