МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №3
по дисциплине «Построение и анализ алгоритмов»
Тема: Коммивояжер (TSP)

Студентка гр. 1304 _____ Чернякова В.А.

Преподаватель _____ Шевелева А.М.

Санкт-Петербург
2023

Цель работы.

Изучение задачи коммивояжёра – задачи комбинаторной оптимизации, заключающийся в поиске самого выгодного маршрута, проходящего через указанные города хотя бы по одному разу с последующим возвратом в исходный город. Реализация алгоритма для решения данной задачи.

Задание.

Дана карта городов в виде ассиметричного, неполного графа $G = (V, E)$, где $V(|V|=n)$ – это вершины графа, соответствующие городам; $E(|E|=m)$ – это ребра между вершинами графа, соответствующие путям сообщения между этими городами.

Каждому ребру m_{ij} (переезд из города i в город j) можно сопоставить критерий выгодности маршрута (вес ребра) равный w_i (натуральное число [1, 1000]), $m_{ij}=inf$, если $i=j$.

Если маршрут включает в себя ребро m_{ij}, то $x_{ij}=1$, иначе $x_{ij}=0$.

Требуется найти минимальный маршрут (минимальный гамильтонов цикл).

Входные параметры:

Матрица графа из текстового файла.

Выходные параметры:

Кратчайший путь, вес кратчайшего пути, скорость решения задачи.

[1, 2, 3, 4, 1], 4, 0mc

Основные теоретические положения.

Задача коммивояжёра (или TSP от англ. travelling salesman problem) — одна из самых известных задач комбинаторной оптимизации, заключающаяся в поиске самого выгодного маршрута, проходящего через указанные города хотя бы по одному разу с последующим возвратом в исходный город. В условиях задачи указываются критерий выгодности маршрута (кратчайший, самый дешёвый, совокупный критерий и тому подобное) и соответствующие матрицы

расстояний, стоимости и тому подобного. Как правило, указывается, что маршрут должен проходить через каждый город только один раз — в таком случае выбор осуществляется среди гамильтоновых циклов. Существует несколько частных случаев общей постановки задачи, в частности, геометрическая задача коммивояжёра (также называемая планарной или евклидовой, когда матрица расстояний отражает расстояния между точками на плоскости), метрическая задача коммивояжёра (когда на матрице стоимостей выполняется неравенство треугольника), симметричная и асимметричная задачи коммивояжёра. Также существует обобщение задачи, так называемая обобщённая задача коммивояжёра.

Оптимизационная постановка задачи относится к классу NP-трудных задач, впрочем, как и большинство её частных случаев. Версия «decision problem» (то есть такая, в которой ставится вопрос, существует ли маршрут не длиннее, чем заданное значение k) относится к классу NP-полных задач. Задача коммивояжёра относится к числу трансвычислительных: уже при относительно небольшом числе городов (>66) она не может быть решена методом перебора вариантов никакими теоретически мыслимыми компьютерами за время, меньшее нескольких миллиардов лет.

Минимальное остовное дерево (или минимальное покрывающее дерево) в (неориентированном) связном взвешенном графе — это остовное дерево этого графа, имеющее минимальный возможный вес, где под весом дерева понимается сумма весов, входящих в него рёбер.

Выполнение работы.

Для решения задачи коммивояжера, нахождения минимального маршрута, необходимо определить минимальный гамильтонов цикл. Также для оптимизации использована функция, которая строит минимальное остовное дерево. Это позволит ускорить работу программы.

Основные функции:

1) *void startSolution()*. Функция, запускающая решение поставленной задачи. Создается переменная типа вектор векторов – считанная из файла матрица с помощью функции *readMatrixFile()*, описанной далее. Также создается объект класса *GraphMatrix*, внутри класса оформлены переменные и методы, позволяющие решить поставленную задачу. В конце осуществляется вывод объекта данного класса, это можно сделать, так как внутри класса перегружен оператор вывода в поток.

2) *std::vector<std::vector<int>> readMatrixFile()*. Функция, считывающая матрицу из файла. Файл с заданным названием *file_input* открывается, и построчно считывается, пока строки не закончатся. Внутри каждой строки считываются элементы, которые помещаются в вектор строк *row*. Перед этим осуществляется следующая замена: прочерки и *inf* заменяется на значение *infinity*, которое с помощью *#define* заменяется на 99999999 – некоторое большое число. Это необходимо, чтобы программа корректно работала. После чтения всех элементов заполненная строка *row* добавляется в общую матрицу – *matrix*. Данная матрица возвращается из функции.

Решение задачи представляется в виде объекта класса *GraphMatrix* со следующими полями с модификатором доступа *private*: *std::vector<std::vector<int>> graph_matrix* – граф в виде матрицы смежности, int size_graph_matrix – размер матрицы смежности, *int start_node* – стартовая вершина, *std::vector<int> optimal_path* – оптимальный путь, *int length_optimal_path* – длина самого оптимального пути, *std::chrono::milliseconds working_time* – время работы программы.

Методы класса *GraphMatrix* с модификатором доступа *public*:

1) *GraphMatrix(std::vector<std::vector<int>> file_graph_matrix): graph_matrix(file_graph_matrix), size_graph_matrix(file_graph_matrix.size()), start_node(0), length_optimal_path(infinity).* Конструктор класса, который в качестве аргумента принимает считанную из выбранного файла матрицу. Происходит инициализация полей класса.

2) *int createMinSpanningTree(int current_node, const std::vector<int>& current_path).* Метод, строящий минимальное остовное дерево для текущей вершины. На вход подаются текущая вершина и вектор, в котором прописан текущий найденный путь. Пока существуют вершины, которые можно просматривать, будет строится дерево. Для выбора узла в дерево выбирается вершина с минимальной стоимостью. Также вершина будет выбрана, если она еще не использовалась в данном функции на вход пути и не находится в векторах, хранящие уже просмотренные вершины и вершины для следующего просмотра. Метод возвращает длину такого дерева. Эта оптимизация – нижняя оценка пути для описанного далее рекурсивного решения задачи коммивояжёра.

3) *int chooseNextNode(int current_node, std::vector<int> current_path, std::vector<std::vector<int>>& visited_node_matrix).* Метод выбора следующей вершины. На вход принимается текущая вершина и путь, а также матрица, в которой отмечены не посещённые пути и пути, которых нет. Путь до следующей вершины должен быть минимален. Сама вершина не должна являться начальной или же выбираемая вершина еще не встречалась в путевом маршруте. Обновляется матрица посещенных вершин. Метод возвращает индекс следующей вершины – он минимален, а индекс вершины в данном случае является и самой вершиной.

4) *void findOptimalPath(int current_node, std::vector<int> current_path, int current_length_path, std::vector<std::vector<int>> visited_node_matrix).* Метод поиска оптимального пути – минимального гамильтоновго цикла. Принимает на вход текущую вершину и путь, текущую длину пути, матрицу возможности посещения вершин. Проверяются оптимизации. Метод закончит работу, если длина текущего пути больше размера матрицы смежности графа, длина текущего пути, суммированная с длиной остовного дерева от текущей вершины больше размера матрицы смежности графа. Если размер вектора, в котором хранится путь, равен размеру матрицы смежности графа, то значения полей класса, соответствующие оптимальной длине пути и самому оптимальному пути, перезаписываются. Иначе работает цикл, до тех пор, пока размер

текущего пути не станет равен размеру матрицы. Вызывается метод выбора следующей вершины. Создаются переменные, где будут хранится обновленные матрицы текущего пути и возможности посещения вершин, а также следующая вершина. Метод, описываемые в данном пункте, вызывается снова от следующей вершины, обновленного вектора пути, текущего значения пути суммированного с стоимостью пути из текущей вершины до следующей, обновленной матрицы возможности посещения вершин. Данный метод рекурсивный.

5) *void workAlgorithm()*. Метод, запускающий работу алгоритма. Внутри метода создаётся матрица возможности посещения вершин. Изначально инициализируется нулями. Но если значение элемента матрицы равно *infinity* – замена на *-1*, данную вершину посетить нельзя. Запускается метод *findOptimalPath*, описанный выше, от полей класса – стартовая вершина, вектор оптимального пути и его размер, матрица возможности посещения вершин, составленная раннее. Также засекается время работы программы.

6) *friend std::ostream& operator<<(std::ostream& os, const GraphMatrix& graph_matrix)*. Перегрузка оператора вывода в поток, для получения отформатированного ответа. Получает на вход объекты классов потока вывода и *GraphMatrix*. Возвращает объект потока вывода. Метод объявлен со спецификатором *friend*, так как вызывается за пределами класса, где необходим доступ к приватным полям.

Разработанный программный код смотреть в приложении А.

Тестирование программного кода смотреть в приложении Б.

Выводы.

В ходе лабораторной работы была изучена задачи коммивояжёра. Разработан программный код, позволяющий решить данную задачу на основе входных данных, представляющих матрицу смежности путей графа. На языке программирования *C++* реализован класс, представляющий собой решение задачи. Внутри класса реализованы методы, решающие задачу с

использованием оптимизаций. Одна из оптимизаций - нахождение длины минимального остовного дерева, нижняя граница рекурсии. Вторая оптимизация – сравнение полученного на этапе длины пути с размером матрицы графа, при превышении последнего значения становится понятно, что рассматриваемый на данном этапе путь неверен.

Программа предусматривает возможность отсутствия пути, в соответствии с чем выводится сообщение с данной информацией.

Разработанный программный код протестирован на различных данных для проверки корректности и скорости работы. Результаты тестирования, которые представлены в приложении Б, подтверждают оптимальность и правильность работы программы.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <fstream>
#include <regex>
#include <chrono>

// Директива для замены
#define infinity_string "inf"
#define dash_string "-"
#define infinity 99999999
#define start_min_index_value -1
#define next_node_not_find -1
#define cannot_visited -1

// Класс, реализующий решение задачи коммивояжера
// Поля класса - граф в матричном виде, размер матрицы,
// стартовая вершина, вектор оптимального пути,
// длина оптимального пути, время работы программы
class GraphMatrix {
    private:
        std::vector<std::vector<int>> graph_matrix;
        int size_graph_matrix;
        int start_node;
        std::vector<int> optimal_path;
        int length_optimal_path;
        std::chrono::milliseconds working_time;

    public:
        // Конструктор класса от матрицы, считанной из файла
        GraphMatrix(std::vector<std::vector<int>>
file_graph_matrix):                    graph_matrix(file_graph_matrix),
size_graph_matrix(file_graph_matrix.size()),
```

```
            start_node(0), length_optimal_path(infinity) {}

            // Метод, строящий минмиальное остовное дерево для текущей
вершины
            // Принимает на вход текущую веришину и текущий найденный
путь
            // Возвращает длину минимального остовного дерева
            int    createMinSpanningTree(int    current_node,    const
std::vector<int>& current_path) {
                int length_tree = 0;
                std::vector<int> include_nodes = {current_node};
                std::vector<int> viewed_nodes;
                while (include_nodes.size() > 0) {
                    int min_index = start_min_index_value;
                    int min_current_node = infinity;
                    std::vector<int>               leaf               =
graph_matrix[include_nodes.back()];
                    for (int i = 0; i < size_graph_matrix; i++) {
                        if    (leaf[i]    <    min_current_node    &&
std::find(current_path.begin(),      current_path.end(),      i)      ==
current_path.end() \
                        &&              std::find(include_nodes.begin(),
include_nodes.end(),       i)       ==       include_nodes.end()       &&
std::find(viewed_nodes.begin(),      viewed_nodes.end(),      i)      ==
viewed_nodes.end()) {
                            min_current_node = leaf[i];
                            min_index = i;
                        }
                    }
                    if (min_index == start_min_index_value) {
                        viewed_nodes.push_back(include_nodes.back());
                        include_nodes.pop_back();
                    }
                    else {
                        include_nodes.push_back(min_index);
                        length_tree += leaf[min_index];
                    }
                }
                return length_tree;
```

```
        }

        // Метод выбора следующей вершины, путь до которой должен быть минимален
        // Принимает на вход текущую вершину и путь, матрицу возможности посещения вершин
        // Возвращает индекс, он же и значение, следующей вершиины
        int chooseNextNode(int current_node, std::vector<int> current_path, std::vector<std::vector<int>>& visited_node_matrix) {
            std::vector<int> leaf = graph_matrix[current_node];
            int min_current_node = infinity;
            int min_index = start_min_index_value;
            for (int i = 0; i < size_graph_matrix; i++) {
                if (leaf[i] < min_current_node && (i != start_node || current_path.size() == size_graph_matrix - 1) \
                && std::find(current_path.begin(), current_path.end(), i) == current_path.end()) {
                    if (visited_node_matrix[current_node][i] == 0) {
                        min_current_node = leaf[i];
                        min_index = i;
                    }
                }
            }
            if (min_index != start_min_index_value) {
                visited_node_matrix[current_node][min_index] = 1;
            }
            return min_index;
        }

        // Метод поиска оптимального пути
        // Принимает на вход текущую вершину и путь, длину данного пути, матрицу возможности посещения вершин
        // Обновление переменных класса в соответствии с происходящими изменениями внутри метода
        void findOptimalPath(int current_node, std::vector<int> current_path, int current_length_path, std::vector<std::vector<int>> visited_node_matrix) {
            if (current_length_path > length_optimal_path) {
```

```
                    return;
                }
                if                (current_length_path                +
createMinSpanningTree(current_node, current_path) > length_optimal_path)
{
                    return;
                }
                if (current_path.size() == size_graph_matrix) {
                    if (current_length_path < length_optimal_path) {
                        this->length_optimal_path = current_length_path;
                        this->optimal_path = current_path;
                    }
                    return;
                }
                while (current_path.size() != size_graph_matrix) {
                    if (current_path.size() == size_graph_matrix) {
                        return;
                    }
                    int    next_node    =    chooseNextNode(current_node,
current_path, visited_node_matrix);
                    if (next_node == next_node_not_find) {
                        return;
                    }
                    std::vector<int> update_path(current_path);
                    std::vector<std::vector<int>>
update_visited_node_matrix(visited_node_matrix);
                    update_path.push_back(next_node);
                    findOptimalPath(next_node,              update_path,
current_length_path       +       graph_matrix[current_node][next_node],
update_visited_node_matrix);
                    visited_node_matrix[current_node][next_node] = 1;
                }
                return;
            }

            // Метод запускающий работу алгоритма
            void workAlgorithm() {
                auto start_time = std::chrono::steady_clock::now();
```

```
                std::vector<std::vector<int>>
visited_nodes_matrix(size_graph_matrix,
std::vector<int>(size_graph_matrix, 0));
                for (int i = 0; i < size_graph_matrix; i++) {
                    for (int j = 0; j < size_graph_matrix; j++) {
                        if (graph_matrix[i][j] == infinity) {
                            visited_nodes_matrix[i][j] = cannot_visited;
                        }
                    }
                }
                findOptimalPath(start_node,                optimal_path,
optimal_path.size(), visited_nodes_matrix);
                optimal_path.insert(optimal_path.begin(), start_node);
                auto end_time = std::chrono::steady_clock::now();
                working_time                                          =
std::chrono::duration_cast<std::chrono::milliseconds>(end_time          -
start_time);
            }

            // Перегрузка оператора вывода в поток
            // Вывод отформатированного ответа на экран
            //  Принимает  на  вход  поток  вывода,  объект  класса
GraphMatrix по ссылке
            // Возвращает объект потока вывода
            friend  std::ostream&  operator<<(std::ostream&  os,  const
GraphMatrix& graph_matrix) {
                if          (graph_matrix.optimal_path.size()          <
graph_matrix.size_graph_matrix) {
                     os << "There is no path!";
                 }
                 else {
                    os << "[";
                    for       (int       i       =       0;       i       <
graph_matrix.optimal_path.size()-1; i++) {
                        os << graph_matrix.optimal_path[i] + 1 << ", ";
                    }
```

```
                    os << graph_matrix.optimal_path.back() + 1 << "], "
<<      graph_matrix.length_optimal_path      <<      ",      "      <<
graph_matrix.working_time.count() << "ms";
                }
                return os;
            }
    };

    // Функция, считывающая матрицу из файла
    // Возвращает вектор векторов - считанную матрицу
    std::vector<std::vector<int>> readMatrixFile() {
        std::vector<std::vector<int>> matrix;
        std::ifstream file_input("test5.txt");
        std::string line;
        while (getline(file_input, line)) {
            std::string number_to_replace = std::to_string(infinity);
            line = std::regex_replace(line, std::regex(infinity_string),
number_to_replace);
            line  =  std::regex_replace(line,  std::regex(dash_string),
number_to_replace);
            std::istringstream iss(line);
            std::vector<int> row;
            int elements;
            while (iss >> elements) {
                row.push_back(elements);
            }
            matrix.push_back(row);
        }
        return matrix;
    }

    // Функция, запускающая решение поставленной задачи
    void startSolution() {
        std::vector<std::vector<int>>        file_graph_matrix        =
readMatrixFile();
        GraphMatrix graph_matrix(file_graph_matrix);
        graph_matrix.workAlgorithm();
        std::cout << graph_matrix << std::endl;
    }
```

```
int main() {
    startSolution();
    return 0;
}
```

ПРИЛОЖЕНИЕ Б
ТЕСТИРОВАНИЕ ПРОГРАММЫ

Проведено тестирование на различных входных данных, проверен результат работы программы – путь, его длина и время выполнения работы.

Результаты тестирования представлены в табл. Б.1.

Таблица Б.1 - Примеры тестовых случаев

№	Входные данные	Результат программы
1	inf 1 2 2 - inf 1 2 - 1 inf 1 1 1 - inf	[1, 2, 3, 4, 1], 4, 0ms
2	inf 5 7 6 8 3 1 inf 8 4 6 2 3 9 inf 6 5 3 7 8 4 inf 4 2 2 7 5 6 inf 6 5 2 6 4 5 inf	[1, 6, 2, 4, 3, 5, 1], 20, 0ms
3	inf - 1 - 1 1 1 inf 1 - 1 - - 1 inf 1 1 - - - - inf - - 1 1 1 - inf 1 1 - 1 - 1 inf	There is no path!
4	inf 6 6 6 6 6 6 inf 6 6 6 6 6 6 inf 6 2 6 6 6 6 inf 6 6 6 6 6 6 inf 6 6 6 6 6 6 inf	[1, 2, 3, 5, 4, 6, 1], 32, 1ms

Продолжение таблицы Б.1

№	Входные данные	Результат программы
5	Граф 20 на 20 со случайными значениями ребер из файла test5.txt	[1, 15, 13, 18, 4, 16, 6, 9, 8, 2, 11, 5, 10, 17, 7, 14, 20, 19, 12, 3, 1], 157, 41ms