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: Коммивояжёр (TSP).

Студент гр. 1304 Байков Е.С.

Преподаватель Шевелева А.М.

Санкт-Петербург
2023

Цель работы.

Реализовать алгоритм поиска минимального по весу гамильтонова цикла в орграфе.

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):

$$\min W = \sum_{i=1}^{n} \sum_{j=1}^{n} x_{ij}\, w_{ij}$$

Входные параметры:

Матрица графа из текстового файла.

inf 1 2 2

- inf 1 2

- 1 inf 1

1 1 - inf

Выходные параметры:

Кратчайший путь, вес кратчайшего пути, скорость решения задачи.

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

Основные теоретические положения.

Задача коммивояжёра – 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.

Оптимизационная постановка задачи относится к классу NP-трудных задач, впрочем, как и большинство её частных случаев.

Выполнение работы.

Для того чтобы достичь большей скорости при решении данной задачи код был написан на языке программирования C++, а также добавлена оптимизация по нижней границе целевой функции.

Структура

Далее описаны структуры и методы, используемые в коде программы для решения поставленной задачи.

1) Класс *Path*: класс, имеющий в качестве полей вектор *int*, в котором хранятся вершины, которые находятся в пути, а также поле отвечающее за текущий вес данного пути. У класса есть несколько методов, позволяющих проще оперировать с ним: *get_path_length* – возвращает текущую длину пути; *get_path_weight* – возвращает текущий вес пути; *set_path_weight* – устанавливает вес пути; *set_default* – возвращает у пути стандартные настройки – вес равный максимальному значению типа *int* и отчищает сам вектор от всех значений; *add_vertex* – добавление конкретной вершины в путь; get_last_vertex – возвращает значение последней добавленной в путь вершины; *is_vertex_in_path* – проверяет присутствует ли данная вершина в пути или нет; *delete_last_vertex* –

удаляет последнюю добавленную вершину из пути; *print_path_info* – выводит на экран информацию о пути (последовательность вершин и вес).

2) Класс *Salesman*: класс реализующий поиск. Его полями являются: вектор векторов целочисленных значений (*matrix*), две переменные типа *Path*, в которых хранится текущий путь и минимальный найденный, количество вершин (*vertex_amount*), нижняя граница целевой функции (*lower_edge*), переменная типа *bool*, которая является флагом, говорящим о том, что самый оптимальный путь уже найден, и поле *time* для подсчета времени работы программы. Методы класса: *read_data* – осуществляет чтение матрицы из файла; *find_lower_edge* – поиск нижней границы оценочной функции; *find_shortest_path* – поиск кратчайшего гамильтонова цикла с помощью метода ветвей и границ и поиска с возвратом; *print_answer* – вывод ответа.

Разработанный программный код смотреть в приложении А.

Проведенные тесты смотреть в приложении Б.

Выводы.

В ходе лабораторной работы был изучен принцип решения задачи коммивояжёра. Разработана программа, которая основывается на двух методах, позволяющих решить задачу коммивояжёра. Программа написана на языке программирования *C++* с использованием классов для лучшего понимания и облегчения реализации алгоритма решения. Программа предусматривает считывание из файла, а также проверки на то, что файл не удалось открыть или он был испорчен.

Был использован метод ветвей и границ, который упрощает поиск пути, исключая из поиска решений заведомо проигрышные (по весу) пути. Также использовался алгоритм поиска с возвратом и оценка нижней границы целевой функции, что облегчает поиск в полных графах, где любые два ребра равны по весу.

Разработанный код протестирован на разных входных данных. Программа дает результат в среднем меньше чем через 3 минуты.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

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

#define START_VERTEX 0
#define START_PATH_WEIGHT 0

// Class contains information about path and helps operate with vertexes
class Path {
public:
    // Constructor of class
    Path() {
        path_weight = INT32_MAX;
    }

    // getter of path length
    int get_path_length() const {
        return path.size();
    }

    // getter of path weight
    int get_path_weight() const {
        return path_weight;
    }

    // setter of path weight
    void set_path_weight(int weight) {
        path_weight = weight;
    }

    // method set path default values
    void set_default() {
        path_weight = INT32_MAX;
        path.clear();
    }

    // getter of last vertex in path
    int get_last_element() const {
        return path.back();
```

```
        }

        // method adds new vertex in the path
        void add_vertex(int vertex) {
            path.push_back(vertex);
        }

        // method checks presence of certain vertex in path
        bool is_vertex_in_path(int vertex) {
            return   std::find(path.begin(),   path.end(),   vertex)   !=
path.end();
        }

        // method delete last vertex in path
        void delete_last_vertex() {
            path.pop_back();
        }

        // method prints information of path in human-readable format
        void print_path_info() {
            for (auto vertex : path) {
                std::cout << vertex + 1 << " ";
            }
            std::cout << ", weight = " << path_weight << "\n";
        }
    private:
        std::vector<int> path; // the structure of path - common vector
of int
        int path_weight; // weight of path
    };


    // Class salesman solves TSP
    class Salesman {
    public:
        // Constructor of class Salesman create new path with start vertex
and start weigth
        // default start vertex - 0 (1) and start weight - 0
        // initializes fields is_found_best_path and lower_edge
        Salesman() {
            current_path.add_vertex(START_VERTEX);
            current_path.set_path_weight(START_PATH_WEIGHT);
            is_found_best_path = false;
            lower_edge = 0;
        }

        // method reads information from file
        void read_data(std::string file_name) {
            std::fstream file;
            file.open(file_name);
```

```
            if (file.is_open()) {
                file >> vertex_amount;
                matrix = std::vector<std::vector<int>>();
                std::string temporary_string;
                for (int i = 0; i < vertex_amount; ++i) {
                    matrix.push_back(std::vector<int>());
                    for (int j = 0; j < vertex_amount; ++j) {
                        file >> temporary_string;
                        if (temporary_string == "-" || temporary_string
== "inf") {
                            matrix[i].push_back(0);
                        }
                        else {
                            matrix[i].push_back(stoi(temporary_string));
                        }
                    }
                }
            }
        }

        // method runs reading, finding path and printing methods and
times.
        void start(std::string file_name) {
            auto begin = std::chrono::steady_clock::now();
            read_data(file_name);
            find_lower_edge();
            find_shortest_path(current_path);
            auto end = std::chrono::steady_clock::now();
            time                                                    =
std::chrono::duration_cast<std::chrono::milliseconds>(end - begin);
            print_answer();
        }

        // method finding shortest Hamiltonian cycle in graph by
backtracking
        void find_shortest_path(Path current_path) {
            if (current_path.get_path_length() == vertex_amount) {
                if
(matrix[current_path.get_last_element()][START_VERTEX]    !=    0    &&
current_path.get_path_weight()                                       +
matrix[current_path.get_last_element()][START_VERTEX]                <
minimal_path.get_path_weight()) {
                    minimal_path = current_path;
                    minimal_path.add_vertex(START_VERTEX);

minimal_path.set_path_weight(current_path.get_path_weight()          +
matrix[current_path.get_last_element()][START_VERTEX]);
                    if (minimal_path.get_path_weight() == lower_edge) {
                        is_found_best_path = true;
                    }
```

```
                    return;
                }
            }
            for (int i = 0; i < vertex_amount; ++i) {
                if   (matrix[current_path.get_last_element()][i]   !=   0
&& !current_path.is_vertex_in_path(i)) {
                    if         (current_path.get_path_weight()         +
matrix[current_path.get_last_element()][i]                             <
minimal_path.get_path_weight()) {

current_path.set_path_weight(current_path.get_path_weight()            +
matrix[current_path.get_last_element()][i]);
                        current_path.add_vertex(i);
                        find_shortest_path(current_path);
                        current_path.delete_last_vertex();

current_path.set_path_weight(current_path.get_path_weight()            -
matrix[current_path.get_last_element()][i]);
                        if (is_found_best_path) {
                            return;
                        }
                    }
                }
            }
            return;
        }

        // method prints answer in human-readable format
        void print_answer() {
            if (minimal_path.get_path_weight() == INT32_MAX) {
                std::cout << "There is not any path\n";
                return;
            }
            minimal_path.print_path_info();
            std::cout << "Time: " << time.count() << " ms";
        }

        // find lower edge of objective function
        void find_lower_edge() {
            std::vector<std::vector<int>> matrix_copy = matrix;
            for (auto& row : matrix_copy) {
                for (int i = 0; i < vertex_amount; ++i) {
                    if (row[i] == 0) {
                        row[i] = INT32_MAX;
                    }
                }
            }
            for (auto& row : matrix_copy) {
                int minimum = *std::min_element(row.begin(), row.end());
                lower_edge += minimum;
```

```
                for (int i = 0; i < vertex_amount; ++i) {
                    row[i] -= minimum;
                }
            }

            for (int i = 0; i < vertex_amount; ++i) {
                int minimum = INT32_MAX;
                for (auto row : matrix_copy) {
                    minimum = row[i] < minimum ? row[i] : minimum;
                }
                lower_edge += minimum;
            }
        }

    private:
        std::vector<std::vector<int>> matrix; // matrix of arcs
        Path minimal_path; // minimal path which is printed as an answer
        Path current_path; // the path to check for minimum weight
        int vertex_amount; // amount of vertexes in graph
        std::chrono::milliseconds time; // the path to check for minimum
weight
        int lower_edge; // lower edge of objective function
        bool is_found_best_path; // flag that shows the minimal path's
already found
    };


    // main function
    int main() {
        for (int i = 1; i <= 5; ++i) {
            auto salesman = Salesman();
            salesman.start("test" + std::to_string(i) + ".txt");
        }
        return 0;
    }
```

ПРИЛОЖЕНИЕ Б
ТЕСТИРОВАНИЕ

Тестирование программы проведено на входных данных приведенных в таблице 1, а также вывод программы на каждый из тестов.

Таблица 1. Тестирование кода программы.

№	Входные данные	Программный вывод
1	4 inf 1 2 2 - inf 1 2 - 1 inf 1 1 1 - inf	1 2 3 4 1, weight = 4 Time: 0 ms
2	20 inf - 65 - 2 - 21 123 48 17 91 74 - 21 35 - 85 87 21 43 35 inf 3212 - 24 43 46 75 - - 3 27 87 55 50 - 65 - 10 68 3 68 inf 24 44 32 19 17 13 66 43 93 38 - 42 34 58 - 91 36 12 50 75 inf 87 62 89 21 - 41 45 89 68 35 32 9 16 88 23 75 84 19 123 90 inf 132 69 52 - 3 62 62 23 - 77 3213 68 24 20 38 17 77 48 19 70 inf - 43 - - 43 - 10 - 91 - 89 79 35 50 - - 93 19 94 - inf 20 - 79 43 82 - 7 61 - 49 - - - 7 1 76 - 64 20 1 inf 12 4 42 - 75 - 34 - 9 35 69 79 7 41 90 38 88 68 - 49 inf 91 87 50 58 81 - 47 48 - - - 21 20 72 97 90 - - - 50 inf - 47 - - 72 59 11 - - 41 - - 98 97 34 45 7 55 1 47 inf - 47 38 35 97 - 53 61 95 64 51 21 64 55 92 64 41 68 66 56 inf 70 - 77 84 55 87 82 48 95 23 49 54 88 34 - 97 18 76 43 40 inf 54 46 - 77 1 84 42 50 - 93 4 73 53 79 1 73 17 95 10 - inf 1 27 - 11 85 - 69 80 81 11 76 68 83 28 67 16 45 74 1 84 inf 74 81 - - - 20 54 97 47 1 - 56 80 42 84 20 83 123 62 61 inf 84 - 74 64 27 12 61 96 41 46 12 83 96 37 34 - 46 53 36 11 inf 13 87 49 94 70 50 4 75 58 96 - 24 9 - 76 10 61 16 98 - inf - 4 - 85 47 77 49 32 4 - 1 50 82 11 76 - - 92 70 - inf - 91 - 72 - 36 43 55 - 95 - 87 52 - 40 - - - 41 16 inf	1 7 14 15 13 18 20 19 12 3 8 6 4 16 5 10 17 2 11 9 1, weight = 186 Time: 64985 ms
3	7 inf - - - - - 1 - inf - - - 1 1 1 - inf - - - - 2 - - inf - - - 1 - - - inf - 2 12 - - - - inf 3 - - - - 1 1 inf	There is not any path

Продолжение таблицы 1.

№	Входные данные	Программный вывод
4	5 inf 2 3 4 5 23 inf 23 1 1 3 1 inf 3 1 4 4 4 inf 4 1 2 3 1 inf	1 2 4 3 5 1 , weight = 9 Time: 0 ms
5	20 inf 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 inf 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 inf 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 inf 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 inf 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 inf 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 inf 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 inf 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 inf 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 inf 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 inf 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 inf 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 inf 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 inf 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 inf 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 inf 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 inf 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 inf 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 inf 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 inf	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 1, weight = 20 Time: 0 ms