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).

Студент гр. 1384 _____ Тапеха В.А.

Преподаватель _____ Шевелева А.М.

Санкт-Петербург
2023

Цель работы.

Построить алгоритм, который решает задачу поиска минимального гамильтонова цикла в графе – задачу коммивояжера, а также оптимизировать этот алгоритм.

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 mij (переезд из города i в город j) можно сопоставить критерий выгодности маршрута (вес ребра) равный w i (натуральное число [1, 1000]), mij =inf, если i=j. Если маршрут включает в себя ребро mij , то xij =1, иначе xij =0.

Требуется найти минимальный маршрут (минимальный гамильтонов цикл).

Пример входных данных.

Матрица графа из текстового файла.

inf 1 2 2

- inf 1 2

- 1 inf 1

1 1 - inf

Пример выходных данных.

Кратчайший путь, вес кратчайшего пути, скорость решения задачи.

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

// Задача должна решаться на размере матрицы 20x20 не дольше 3 минут в среднем.

Выполнение работы.

Класс *Graph* содержит функции для сортировки и чтения графа, разбиения строки на вектор ребер и печати графа. Класс *TSP* отвечает за решение задачи с использованием алгоритма ветвей и границ. Он включает в себя функции для печати ответа, нахождения нижней границы, подсчета нижней границы и выполнения рекурсивной функции *tsp*.

Структура *Solution* используется для хранения лучшего решения на данный момент.

Структура *Edge* хранит ребро графа.

Для удобства некоторые константы были вынесены в CMakeLists.txt.

Graph:

Функция *sortGraph* сортирует ребра каждой вершины в порядке возрастания их веса. Функция *readGraph* считывает график из файла и вызывает функцию разделения, чтобы разбить каждую строку на вектор ребер. Функция разделения использует разделитель пробелов для разделения строки и преобразования строковых значений в целые числа. Затем он создает новый объект Edge для каждого целочисленного значения и добавляет его к результирующему вектору.

Метод *Graph::sortGraph* сортирует ребра каждой вершины в графе в порядке возрастания на основе их веса, используя std::sort и лямбда-функцию.

Метод *Graph::readGraph* считывает граф из файла и заполняет std::map графа вершинами и ребрами графа. Каждая строка в файле соответствует вершине графа и содержит последовательность чисел, представляющих веса ребер, которые инцидентны этой вершине.

Метод *Graph::split* принимает std::string в качестве входных данных и разбивает строку по пробелу. Затем каждое слово или число преобразуется

преобразовывается к целочисленному типу, а потом к Edge и добавляется в вектор, который возвращается методом.

Метод *Graph::printGraph* выводит вершины и ребра графа в стандартный поток вывода. В скобках указывается сначала вершина, затем ее вес.

TSP:

Класс *TSP* содержит методы для решения проблемы и вывода решения. Метод решения считывает граф, сортирует его, инициализирует переменные и вызывает метод *tsp* для решения задачи с использованием рекурсивного алгоритма поиска в глубину с возвратом. Метод *tsp* получает текущее решение, текущее количество посещенных вершин, текущую вершину и начальную вершину. Метод проверяет, больше ли текущая нижняя граница решения текущей стоимости наилучшего решения. Если да, то возвращается. Если текущее решение содержит все вершины, метод проверяет, существует ли путь из текущей вершины в начальную вершину. Если есть, метод обновляет лучшее решение, если стоимость текущего решения меньше стоимости лучшего решения. Если нет, то возвращается. Если еще есть вершины для посещения, метод помечает текущую вершину как посещенную, добавляет ребро к текущему решению и рекурсивно вызывает себя для каждой непосещенной вершины. После каждого рекурсивного вызова метод снимает пометку с текущей вершины, удаляет ребро из текущего решения и вычитает вес ребра из текущей стоимости.

Метод *TSP::printAnswer* выводит ответ на задачу коммивояжера в стандартный поток вывода. Если гамильтонов цикл найден, он выводит последовательность вершин в цикле и его общую стоимость. Если гамильтонов цикл не найден, выводится сообщение об отсутствии цикла.

Метод *TSP::solve* считывает граф из файла, сортирует ребра каждой вершины, вычисляет нижнюю границу решения и инициализирует текущее состояние решения. Затем метод вызывает метод tsp для поиска всех возмож-

ных путей в графе, пока не будет найден гамильтонов цикл. Метод также измеряет время, затраченное на решение задачи.

Метод *TSP::tsp* рекурсивно ищет все возможные пути в графе. Метод вызывается методом решения с начальным состоянием current_solution, содержащим ее стоимость и пройденный путь. Метод проходит все исходящие ребра текущей вершины и рекурсивно вызывает себя с обновленным состоянием для каждой соседней вершины, которая еще не была посещена. Метод выполняет откат, когда все возможные пути от текущей вершины исследованы или когда нижняя граница текущего решения больше или равна стоимости лучшего известного на данный момент решения.

Метод *TSP::setLowBound* вычисляет нижнюю границу решения задачи коммивояжера, суммируя веса наименьших ребер, инцидентных каждой вершине графа.

Метод *TSP::countLowBound* вычисляет нижнюю границу стоимости решения задачи с учетом частичного решения current_solution.

Оптимизации:

1. Сортируются ребра графа, что позволяет в начале рассматривать наименьшие пути.

2. Также оценивалось текущее решение. Считались наименьшее веса ребер, и, если сумма текущего пути с рассчитанными минимальными значениями по строкам больше либо равна лучшему решению, то такое решение не подходит.

Благодаря этим оптимизациям задача в среднем решается не более 3х минут.

Разработанный программный код см. в приложении А.

Тестирование.

Для тестирования использовался фреймворк Google Test.

Было проведено 9 тестов:

1. ExampleTest – тест на корректность решения примера из условия задачи. Ожидаемый путь: 0-1-2-3-0, ожидаемая длина: 4.

2. Test2 – тест на корректность решения задачи для графа с 7 вершинами. Ожидаемый путь: 0-1-2-3-4-5-6-0, ожидаемая длина: 7.

3. Test3 – тест на корректность решения задачи для графа размера 15x15. Ожидаемый путь: 0-13-9-5-7-6-12-10-8-14-11-1-4-2-3-0, ожидаемая длина: 1656.

4. Test4 – тест на корректность решения задачи для графа размера 20x20. Ожидаемый путь: 0-4-13-8-11-7-5-19-15-16-2-3-10-12-18-1-6-14-17-9-0, ожидаемая длина: 1541.

5. Test5 – тест на корректность решения задачи для графа размера 20x20, работающего быстрее чем предыдущий тест. Ожидаемый путь: 0-4-14-13-10-5-1-11-9-2-6-3-15-19-12-17-18-7-16-8-0, ожидаемая длина: 1187.

6. Test6 - тест на корректность решения задачи для графа, в котором все значения одинаковы, кроме одного. Ожидаемый путь: 0-1-2-4-5-3-0, ожидаемая длина: 11.

7. Test7 - тест на обработку графа, в котором отсутствует гамильтонов цикл. Ожидается, что результатом решения будет пустой путь.

8. Test8 - тест на обработку графа, содержащего только изолированную вершину. Ожидается, что результатом решения будет пустой путь.

9. Test9 - тест на обработку графа, в котором все элементы матрицы одинаковые. Ожидаемый путь: 0-1-2-3-4-5-6-7-8-9-0, ожидаемая длина: 10.

Все тесты были успешно пройдены о чем свидетельствует вывод программы (см. рисунок 1).

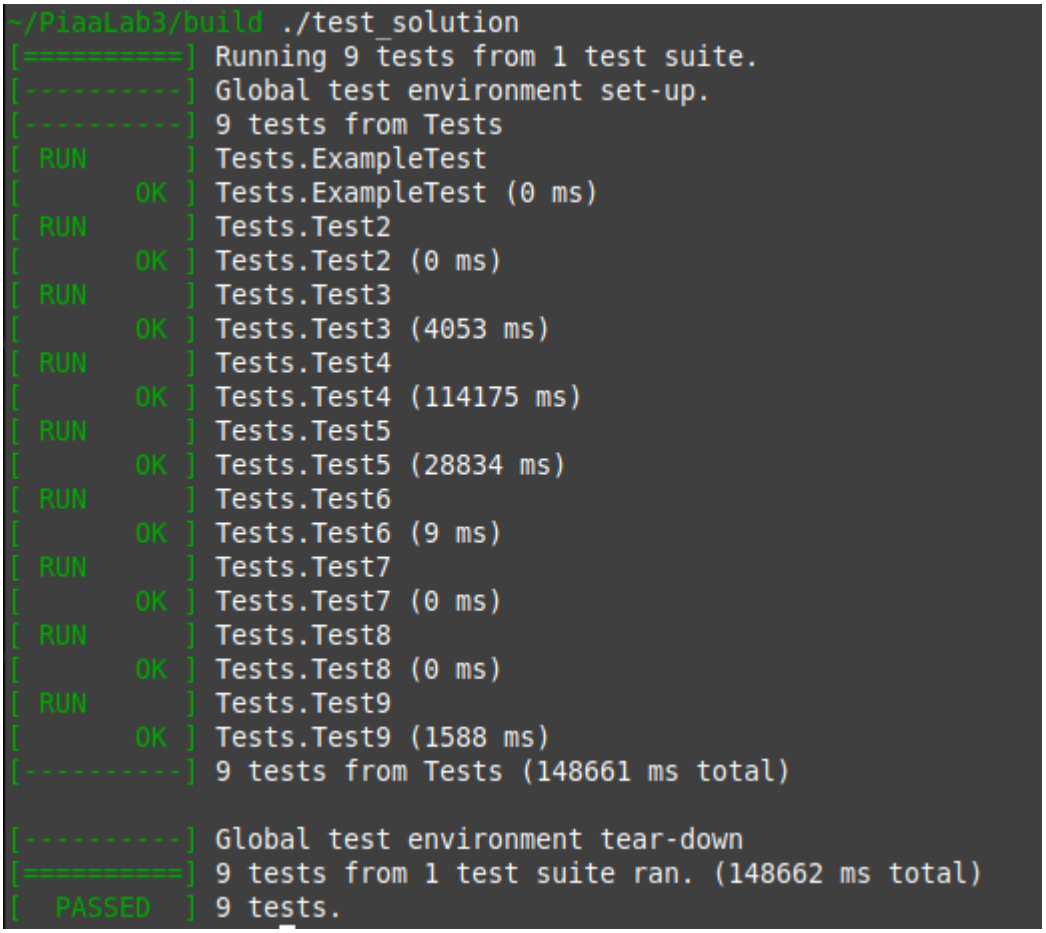

Рисунок 1 - Тестирование

На этом же рисунке также можно оценить время выполнения работы программы. Видно, что даже для матриц 20x20 время работы не превышает двух минут.

Вывод.

Задача коммивояжера(Travelling Salesman Problem) является одной из классических задач оптимизации комбинаторных объектов. Для ее решения была написана программа, которая выполняет рекурсивный обход всех возможных решений задачи. При этом для того, чтобы программа работала не более трех минут граф был заранее отсортирован и введена оценка, которая отсекает некоторую часть всех решений. Также было проведено тестирование с использованием фреймворка Google Test, которое доказало корректность и быстродействие работы разработанной программы.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: solution.h

```
#ifndef LAB3_SOLUTION_H
#define LAB3_SOLUTION_H

#include <iostream>
#include <numeric>
#include <vector>
#include <map>
#include <fstream>
#include <chrono>


// Структура, хранящая ребро графа
struct Edge {
        explicit Edge(int vertex = -1, int weight = INF) :
vertex(vertex), weight(weight) {}

    int vertex;
    int weight;
};


// Структура, хранящая текущее состояние решения
struct State {
    explicit State(int cost = 0) : cost(cost) {}

    int cost;
    std::vector<int> way;
};


// Класс, отвечающий за граф
class Graph {
public:
    // Конструктор
       explicit Graph(const std::string& fileName = "1.txt") :
rank(0) { file.open(fileName); }
    // Деструктор
    ~Graph() { if(file.is_open()) { file.close(); } }

    // Считывает граф
    void readGraph();
    // Сортирует граф
    void sortGraph();

    // Возвращает ранг матрицы
    [[nodiscard]] int getRank() const { return rank; }

    // Выводит граф
    void printGraph() const;
```

```
        // Для удобства работы с классом перегружен оператор[]
             std::vector<Edge>&  operator[](int  vertex)  {  return
graph[vertex]; }
    private:
        // Метод, разделяющий строку
        std::vector<Edge> split(std::string raw);

    private:
        // Ранг матрицы
        int rank;
        // Файл, с которого считывается матрица
        std::fstream file;
        // Искомый граф
        std::map<int, std::vector<Edge>> graph;
    };


    // Класс, решающий задачу
    class TSP {
    public:
        // Конструктор
            explicit  TSP(const  std::string&  fileName)  :  time(0),
best(INF), graph(fileName) {}

        // Рекурсивно находит решение задачи
           void  tsp(State  current_solution,  int  current_count,  int
current_vertex);

         // Вызывает все необходимые методы и инициализирует все нуж-
ные структуры для решения
        void solve();

        // Печатает ответ
        void printAnswer();
        // Выводит граф (нужно было для отладки программы, сейчас уже
не используется нигде)
                    [[maybe_unused]]    void    printGraph()    const
{ graph.printGraph(); }
        // Выводит время
        void printTime() const { std::cout << "Время: " << time << '\
n'; }

        // Возвращает лучшее решение
        [[nodiscard]] State getBest() const { return best; }
    private:
        // Заполняет вектор, необходимый для оценки решения
        void setLowBound();
        // Высчитвает значение оценки для текущего решения
        int countLowBound(const State& current_solution);

    private:
        // Хранит время
        double time;
        // Хранит граф
```

```
        Graph graph;
        // Текущее лучшее решение
        State best;
        // Вектор посещенных вершин
        std::vector<bool> visited;
        // Вектор, хранящий необходимые элементы для нижней оценки
        std::vector<int> low_bound;
    };

    /**
     * Метод Graph::sortGraph сортирует ребра каждой вершины в графе
в
     * порядке возрастания на основе их веса, используя std::sort и
лямбда-функцию.
     **/
    void Graph::sortGraph() {
        for (auto& i : graph) {
            std::sort(i.second.begin(), i.second.end(),
                      [](const Edge &first, const Edge &second) {
                          return first.weight < second.weight;
                      });
        }
    }

    /**
     * Метод Graph::readGraph считывает граф из файла и заполняет
std::map графа вершинами и ребрами графа.
     * Каждая строка в файле соответствует вершине графа и содержит
последовательность чисел,
     * представляющих веса ребер, которые инцидентны этой вершине.
     **/
    void Graph::readGraph() {
        std::string raw;
        while(std::getline(file, raw)) {
            graph[rank++] = split(raw);
        }
    }

    /**
     * Метод Graph::split принимает std::string в качестве
     * входных данных и разбивает строку по пробелу.
     * Затем каждое слово или число преобразуется преобразовывается к
целочисленному типу,
     * а потом к Edge и добавляется в вектор, который возвращается
методом.
     **/
    std::vector<Edge> Graph::split(std::string raw) {
        std::string delimiter = " ";

        std::vector<Edge> result;
        size_t pos = 0;
        std::string token;
        int i = 0;

        auto castToInt = [](const std::string& str)
```

```
            { return str == "inf" || str == "-" ? INF :
std::stoi(str); };
        auto implementEdge = [&](const std::string& str) {
            Edge edge;
            edge.weight = castToInt(str);
            edge.vertex = i++;

            result.push_back(edge);
        };

        while ((pos = raw.find(delimiter)) != std::string::npos) {
            token = raw.substr(0, pos);
            implementEdge(token);
            raw.erase(0, pos + delimiter.length());
        }
        implementEdge(raw);

        return result;
    }

    /**
     * Метод Graph::printGraph выводит вершины и ребра графа в
стандартный поток вывода.
     * В скобках указывается сначала вершина, затем ее вес.
     **/
    void Graph::printGraph() const {
        std::cout << "\n";
        for (const auto& i : graph) {
            std::cout << i.first  + 1 << ": ";
            for (auto j : i.second) {
                std::cout << "(" << j.vertex + 1 << ", " << j.weight
<< ") ";
            }
            std::cout << "\n";
        }
    }

    /**
     * Метод TSP::printAnswer выводит ответ на задачу коммивояжера в
стандартный поток вывода.
     * Если гамильтонов цикл найден, он выводит последовательность
вершин в цикле и его общую стоимость.
     * Если гамильтонов цикл не найден, выводится сообщение об отсут-
ствии цикла.
     **/
    void TSP::printAnswer() {
        if (best.way.empty()) {
            std::cout << "Отсутствует гамильтонов цикл\n";
            return;
        }

        for (auto i : best.way) {
            std::cout << i + 1 << " ";
        }
        std::cout << "\nРасстояние: " << best.cost << '\n';
```

```
    }

    /**
     * Метод TSP::solve считывает граф из файла, сортирует ребра каждой вершины,
     * вычисляет нижнюю границу решения и инициализирует текущее состояние решения.
     * Затем метод вызывает метод tsp для поиска всех возможных путей в графе, пока не будет найден гамильтонов цикл.
     * Метод также измеряет время, затраченное на решение задачи.
     **/
    void TSP::solve() {
        auto start_time = std::chrono::steady_clock::now();
        graph.readGraph();
        graph.sortGraph();

        setLowBound();

        visited.resize(graph.getRank());
        visited.assign(visited.size(), false);

        State solution;
        solution.way.push_back(0);

        tsp(solution, 1, 0);
        auto end_time = std::chrono::steady_clock::now();

        time = static_cast<std::chrono::duration<double>>(end_time - start_time).count();
    }

    /**
     * Метод TSP::tsp рекурсивно ищет все возможные пути в графе.
     * Метод вызывается методом решения с начальным состоянием current_solution,
     * содержащим ее стоимость и пройденный путь. Метод проходит все
     * исходящие ребра текущей вершины и рекурсивно вызывает себя с обновленным состоянием
     * для каждой соседней вершины, которая еще не была посещена. Метод выполняет откат,
     * когда все возможные пути от текущей вершины исследованы или когда нижняя граница
     * текущего решения больше или равна стоимости лучшего известного на данный момент решения.
     **/
    void TSP::tsp(State current_solution, int current_count, int current_vertex) {
        if (countLowBound(current_solution) >= best.cost) {
            return;
        }

        if (current_count == graph.getRank()) {
            const auto& last = graph[current_vertex];
```

```
                    auto  way_to_first  =  std::find_if(last.begin(),
last.end(),
                                                    [&](const Edge& edge)
{ return edge.vertex == 0; });

            if (way_to_first == last.end()) { return; }

              int total_cost = current_solution.cost + way_to_first-
>weight;
            current_solution.cost = total_cost;
            if (best.cost > total_cost) {
                best = current_solution;
                best.way.push_back(0);
            }
            return;
        }
        else {
            visited.at(current_vertex) = true;
            for (auto edge: graph[current_vertex]) {
                if (!visited.at(edge.vertex)) {
                    visited.at(edge.vertex) = true;
                    current_solution.cost += edge.weight;
                    current_solution.way.push_back(edge.vertex);

                            tsp(current_solution, current_count + 1,
edge.vertex);

                    visited.at(edge.vertex) = false;
                    current_solution.cost -= edge.weight;
                    current_solution.way.pop_back();
                }
            }
            visited.at(current_vertex) = false;
        }
    }

    /**
     * Метод TSP::setLowBound вычисляет нижнюю границу решения задачи
коммивояжера,
     * суммируя веса наименьших ребер, инцидентных каждой вершине
графа.
     **/
    void TSP::setLowBound() {
        for (int i = 0; i < graph.getRank(); ++i) {
            low_bound.push_back(graph[i].at(0).weight);
        }
        std::sort(low_bound.begin(), low_bound.end());
    }

    /**
     * Метод TSP::countLowBound вычисляет нижнюю границу стоимости
решения задачи
     * с учетом частичного решения current_solution.
     **/
    int TSP::countLowBound(const State &current_solution) {
```

```
        int bound = current_solution.cost;
                for (size_t i = 0; i < graph.getRank() -
current_solution.way.size(); ++i) {
            bound += low_bound.at(i);
        }

        return bound;
    }

    #endif //LAB3_SOLUTION_H
```

Название файла: tests.cpp

```
    #include <gtest/gtest.h>
    #include "../solution.h"


    // Пример из условия
    TEST(Tests, ExampleTest){
        std::string full_path = PATH;
        full_path += "1.txt";

        TSP solution(full_path);
        solution.solve();

        std::vector<int> answer = {0, 1, 2, 3, 0};
        ASSERT_EQ(solution.getBest().way, answer);
        ASSERT_EQ(solution.getBest().cost, 4);
    }

    // Немного побольше тест
    TEST(Tests, Test2){
        std::string full_path = PATH;
        full_path += "2.txt";

        TSP solution(full_path);
        solution.solve();

        std::vector<int> answer = {0, 1, 2, 3, 4, 5, 6, 0};
        ASSERT_EQ(solution.getBest().way, answer);
        ASSERT_EQ(solution.getBest().cost, 7);
    }

    // Матрица 15x15
    TEST(Tests, Test3){
        std::string full_path = PATH;
        full_path += "3.txt";

        TSP solution(full_path);
        solution.solve();

         std::vector<int> answer = {0, 12, 6, 7, 5, 9, 13, 10, 8, 14,
11, 1, 4, 2, 3, 0};
        ASSERT_EQ(solution.getBest().way, answer);
```

```
        ASSERT_EQ(solution.getBest().cost, 1656);
    }

    // Матрица 20x20
    TEST(Tests, Test4){
        std::string full_path = PATH;
        full_path += "4.txt";

        TSP solution(full_path);
        solution.solve();

        std::vector<int> answer = {0, 4, 13, 8, 11, 7, 5, 19, 15, 16,
2, 3, 10, 12, 18, 1, 6, 14, 17, 9, 0};
        ASSERT_EQ(solution.getBest().way, answer);
        ASSERT_EQ(solution.getBest().cost, 1541);
    }

    // Матрица 20x20, работает быстрее прошлого теста
    TEST(Tests, Test5){
        std::string full_path = PATH;
        full_path += "5.txt";

        TSP solution(full_path);
        solution.solve();

        std::vector<int> answer = {0, 4, 14, 13, 10, 5, 1, 11, 9, 2,
6, 3, 15, 19, 12, 17, 18, 7, 16, 8, 0};
        ASSERT_EQ(solution.getBest().way, answer);
        ASSERT_EQ(solution.getBest().cost, 1187);
    }

    // Одинаковые значения в матрице, кроме одного
    TEST(Tests, Test6){
        std::string full_path = PATH;
        full_path += "6.txt";

        TSP solution(full_path);
        solution.solve();

        std::vector<int> answer = {0, 1, 2, 4, 5, 3, 0};
        ASSERT_EQ(solution.getBest().way, answer);
        ASSERT_EQ(solution.getBest().cost, 11);
    }

    // Отсутствует гамильтонов цикл
    TEST(Tests, Test7){
        std::string full_path = PATH;
        full_path += "7.txt";

        TSP solution(full_path);
        solution.solve();

        ASSERT_TRUE(solution.getBest().way.empty());
    }
```

```
// Изолированная вершина
TEST(Tests, Test8){
    std::string full_path = PATH;
    full_path += "8.txt";

    TSP solution(full_path);
    solution.solve();

    ASSERT_TRUE(solution.getBest().way.empty());
}

// Все элементы матрицы одинаковые
TEST(Tests, Test9){
    std::string full_path = PATH;
    full_path += "9.txt";

    TSP solution(full_path);
    solution.solve();

    std::vector<int> answer = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
    ASSERT_EQ(solution.getBest().way, answer);
    ASSERT_EQ(solution.getBest().cost, 10);
}


int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Название файла: main.cpp

```
#include "solution.h"

int main() {
    std::string full_path = PATH;
    full_path += "9.txt";

    TSP tsp(full_path);
    tsp.solve();

    tsp.printAnswer();
    tsp.printTime();

    return 0;
}
```

Название файла: CMakeLists.txt

```
cmake_minimum_required(VERSION 3.22.1)
project(Lab3)

set(CMAKE_CXX_STANDARD 20)

find_package(GTest)
```

```
    if (${GTest_FOUND})
        add_executable(test_solution
                solution.h
                tests/tests.cpp)

        target_link_libraries(test_solution
                ${CMAKE_THREAD_LIBS_INIT}
                GTest::GTest
                GTest::gtest_main
                GTest::gmock
                GTest::gmock_main
                )

        enable_testing()

    endif (${GTest_FOUND})

    # Некоторые константы, использующиеся в программе
    add_compile_definitions(PATH="${CMAKE_CURRENT_SOURCE_DIR}/tests/
files/")
    add_compile_definitions(INF=INT16_MAX)
    add_executable(lab3 solution.h main.cpp)
```