**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

отчет

**по лабораторной работе №3**

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

ВАРИАНТ 2

Студент гр. 9309 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Корягин Е.А.

Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Тутуева А.В.

Санкт-Петербург

2021

**Постановка задачи**

Дан список возможных авиарейсов в текстовом файле в формате:

Город отправления 1;Город прибытия 1;цена прямого перелета 1;цена обратного перелета 1

Город отправления 2;Город прибытия 2;цена перелета 2;цена обратного перелета 1

…

Город отправления N;Город прибытия N;цена перелета N;цена обратного перелета N

В случае, если нет прямого или обратного рейса, его цена будет указана как N/A (not available)

*Задание:* найти наиболее эффективный по стоимости перелет из города ***i*** в город ***j***.

Вариант 2: [алгоритм Беллмана-Форда](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%91%D0%B5%D0%BB%D0%BB%D0%BC%D0%B0%D0%BD%D0%B0_%E2%80%94_%D0%A4%D0%BE%D1%80%D0%B4%D0%B0) и матрицу смежности

**Описание реализуемых классов и методов**

Описание реализуемого класса:

Класс Edge реализует работу с дугами (edge) - стоимость существующего перелёта из одного в другой город.

Класс Graph - граф перелётов.

**Описание методов:**

Метод load\_from\_file() – Читает граф из входного текстового файла в заданном формате.

Метод addEdge() – Добавляет в граф дугу (Edge).

Метод minimalCostBellmanFord() – производит вычисление минимальной возможной стоимости из одной в другую вершину (перелёт из города в другой город).

**Оценка временной сложности методов**

Временная сложность метода:

load\_from\_file = O(logN).

addEdge = O(logN).

minimalCostBellmanFord = O(Узлы \* Дуги).

**Описание реализованных unit-тестов**

TEST\_СLASS(MethodsUnitTest) - Проверка формата данных из входного файла, проверка валидности входных параметров.

TEST\_СLASS(PathCostUnitTest) - Верификация нахождения минимальной стоимости перелётов, верификация, что невозможно найти перелёт из одного города в другой.

**Примеры работы**

**Входные данные:**

Изображение выглядит как текст

Автоматически созданное описание

**Результат:**

Изображение выглядит как текст

Автоматически созданное описание

**Листинг**

**Work.cpp**

#include "Header.h"

#include <regex>

#include <iostream>

#include <fstream>

class Util {

public:

static string trim(const string& str, const string& whitespace = " \t") {

const auto strBegin = str.find\_first\_not\_of(whitespace);

if (strBegin == string::npos) {

return "";

}

const auto strEnd = str.find\_last\_not\_of(whitespace);

const auto strRange = strEnd - strBegin + 1;

return str.substr(strBegin, strRange);

}

static vector<string> split(const string& str, const string& delim) {

vector<string> tokens;

size\_t prev = 0, pos = 0;

do {

pos = str.find(delim, prev);

if (pos == string::npos) pos = str.length();

string dirty\_token = str.substr(prev, pos - prev);

string token = trim(dirty\_token);

if (!token.empty()) tokens.push\_back(token);

prev = pos + delim.length();

} while (pos < str.length() && prev < str.length());

return tokens;

}

};

void Graph::addEdge(string from, string to, int price) {

if (price < 0) {

throw std::invalid\_argument("Price must be positive");

}

string normalized\_from = Util::trim(from);

if (normalized\_from.empty()) {

throw std::invalid\_argument("'from' shall not be empty");

}

string normalized\_to = Util::trim(to);

if (normalized\_to.empty()) {

throw std::invalid\_argument("'to' shall not be empty");

}

int i\_from = \_nodePosition(normalized\_from);

int i\_to = \_nodePosition(normalized\_to);

\_addEdge(i\_from, i\_to, price, normalized\_from, normalized\_to);

}

int Graph::minimalCostBellmanFord(string from, string to) {

int \_from = \_getNodeIndex(from);

if (\_from == -1) {

throw std::invalid\_argument("From not found");

}

int \_to = \_getNodeIndex(to);

if (\_to == -1) {

throw std::invalid\_argument("To not found");

}

int V = nodes.size();

int E = edges.size();

vector<int> dist(V);

for (int i = 0; i < V; i++) {

dist[i] = INT\_MAX;

}

dist[\_from] = 0;

for (int i = 1; i <= V - 1; i++)

{

for (int j = 0; j < E; j++)

{

int u = edges[j].from;

int v = edges[j].to;

int price = edges[j].price;

if (dist[u] != INT\_MAX) {

if (dist[u] + price < dist[v]) {

dist[v] = dist[u] + price;

}

}

}

}

for (int i = 0; i < E; i++)

{

int u = edges[i].from;

int v = edges[i].to;

int price = edges[i].price;

if (dist[u] != INT\_MAX && dist[u] + price < dist[v]) {

return INT\_MAX;

}

}

return dist[\_to];

}

void Graph::\_addEdge(int from, int to, int price, string from\_name, string to\_name) {

\_resizeIfRequired();

edges.push\_back(Edge(from, to, price, from\_name, to\_name));

g[from].push\_back(edges.size() - 1);

g[to].push\_back(edges.size() - 1);

}

void Graph::\_resizeIfRequired() {

if (nodes.size() > g.size()) {

g.resize(nodes.size());

}

}

int Graph::\_getNodeIndex(string node\_name) {

auto it = std::find(nodes.begin(), nodes.end(), node\_name);

if (it == nodes.end()) {

return -1;

}

return std::distance(nodes.begin(), it);

}

int Graph::\_nodePosition(string node\_name) {

int index = \_getNodeIndex(node\_name);

if (index == -1) {

nodes.push\_back(node\_name);

return nodes.size() - 1;

}

else {

return index;

}

}

void Graph::load\_from\_file(const char\* filename) {

std::ifstream file(filename);

if (file.is\_open()) {

std::string str;

int line\_num = 0;

while (std::getline(file, str)) {

string delimiter = ";";

vector<string> tokens = Util::split(str, delimiter);

if (tokens.size() != 4) {

throw std::invalid\_argument("File invalid format");

}

else {

string from = tokens.at(0);

string to = tokens.at(1);

string price = tokens.at(2);

string price\_back = tokens.at(3);

if (price != "N/A") {

addEdge(from, to, std::stoi(price));

}

if (price\_back != "N/A") {

addEdge(to, from, std::stoi(price\_back));

}

}

}

file.close();

}

else {

throw std::runtime\_error("Can't open a file");

}

}

void test\_and\_print(Graph\* graph, string from, string to) {

int cost = graph->minimalCostBellmanFord(from, to);

if (cost == INT\_MAX) {

cout << from << "->" << to << ":\tNO FLIGHTS (or error)" << endl;

}

else {

cout << from << "->" << to << ":\t" << cost << endl;

}

}

int main() {

try {

Graph\* graph = new Graph;

cout << "Loading graph from a 'graph\_data.txt' file..." << endl;

graph->load\_from\_file("graph\_data.txt");

cout << "From->To \tMinimalCost" << endl;

cout << "---------------------------------------" << endl;

test\_and\_print(graph, "Madrid", "Rome");

test\_and\_print(graph, "Rome", "Madrid");

test\_and\_print(graph, "Chicago", "Moscow");

test\_and\_print(graph, "Madrid", "Paris");

test\_and\_print(graph, "Paris", "Madrid");

test\_and\_print(graph, "Chicago", "Madrid");

test\_and\_print(graph, "Madrid", "Chicago");

test\_and\_print(graph, "Paris", "Rome");

delete graph;

}

catch (const std::exception& exc) {

std::cerr << "Error: " << exc.what();

}

return 0;

}

**Header.h**

#pragma once

#include <stdio.h>

#include <algorithm>

#include <vector>

#include <iostream>

#include <locale>

#include <stdexcept>

using namespace std;

class Edge {

public:

int from, to;

string from\_name;

string to\_name;

int price;

Edge(int from, int to, int price, string from\_name,

string to\_name) : from(from), to(to), price(price), from\_name(from\_name), to\_name(to\_name) {}

int other(int node) const {

return node == from ? to : from;

}

};

class Graph {

vector<string> nodes;

vector<Edge> edges;

vector<vector<int>> g;

public:

Graph() {

}

void addEdge(string from, string to, int price);

int minimalCostBellmanFord(string from, string to);

void load\_from\_file(const char\* filename);

private:

void \_addEdge(int from, int to, int price, string from\_name, string to\_name);

void \_resizeIfRequired();

int \_getNodeIndex(string node\_name);

int \_nodePosition(string node\_name);

};

**UnitTest3.cpp**

#include "pch.h"

#include "CppUnitTest.h"

#include <stdexcept>

#include <windows.h>

#include "../LabRab3Aistrd\_Koryagin\_9309/Header.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace BellmanFord

{

TEST\_CLASS(PathCostUnitTest)

{

public:

PathCostUnitTest() {

graph = new Graph;

graph->addEdge("Madrid", "Moscow", 10);

graph->addEdge("Moscow", "Madrid", 5);

graph->addEdge("Moscow", "Paris", 40);

graph->addEdge("Madrid", "Rome", 100);

graph->addEdge("Moscow", "Rome", 20);

graph->addEdge("Moscow", "Chicago", 10);

graph->addEdge("Chicago", "Moscow", 6);

graph->addEdge("Chicago", "Paris", 10);

graph->addEdge("Paris", "Chicago", 20);

graph->addEdge("Chicago", "Rome", 20);

}

~PathCostUnitTest() {

delete graph;

}

TEST\_METHOD(TestMadridRome)

{

int cost = graph->minimalCostBellmanFord("Madrid", "Rome");

Assert::AreEqual(30, cost);

}

TEST\_METHOD(TestMoscowRome)

{

int cost = graph->minimalCostBellmanFord("Moscow", "Rome");

Assert::AreEqual(20, cost);

}

TEST\_METHOD(TestChicagoMoscow)

{

int cost = graph->minimalCostBellmanFord("Chicago", "Moscow");

Assert::AreEqual(6, cost);

}

TEST\_METHOD(TestRomeMadridNotExists)

{

int cost = graph->minimalCostBellmanFord("Rome", "Madrid");

Assert::AreEqual(INT\_MAX, cost);

}

TEST\_METHOD(TestMadridParis)

{

int cost = graph->minimalCostBellmanFord("Madrid", "Paris");

Assert::AreEqual(30, cost);

}

TEST\_METHOD(TestParisMadrid)

{

int cost = graph->minimalCostBellmanFord("Paris", "Madrid");

Assert::AreEqual(31, cost);

}

TEST\_METHOD(TestChicagoMadrid)

{

int cost = graph->minimalCostBellmanFord("Chicago", "Madrid");

Assert::AreEqual(11, cost);

}

TEST\_METHOD(TestMadridChicago)

{

int cost = graph->minimalCostBellmanFord("Madrid", "Chicago");

Assert::AreEqual(20, cost);

}

TEST\_METHOD(TestParisRome)

{

int cost = graph->minimalCostBellmanFord("Paris", "Rome");

Assert::AreEqual(40, cost);

}

private:

Graph\* graph;

};

TEST\_CLASS(MethodsUnitTest)

{

public:

MethodsUnitTest() {

graph = new Graph;

}

~MethodsUnitTest() {

delete graph;

}

TEST\_METHOD(FileDoesNotExist)

{

bool exceptionThrown = false;

try {

graph->load\_from\_file("../file\_does\_not \_exists.txt");

}

catch (...) {

exceptionThrown = true;

}

Assert::AreEqual(true, exceptionThrown);

}

TEST\_METHOD(InvalidFileFormat1)

{

bool exceptionThrown = false;

try {

graph->load\_from\_file("../invalid\_graph\_data\_1.txt");

}

catch (const std::exception& exc) {

Logger::WriteMessage(exc.what());

exceptionThrown = true;

}

Assert::AreEqual(true, exceptionThrown);

}

TEST\_METHOD(InvalidFileFormat2)

{

bool exceptionThrown = false;

try {

graph->load\_from\_file("../invalid\_graph\_data\_2.txt");

}

catch (const std::exception& exc) {

Logger::WriteMessage(exc.what());

exceptionThrown = true;

}

Assert::AreEqual(true, exceptionThrown);

}

TEST\_METHOD(InvalidPrice)

{

bool exceptionThrown = false;

try {

graph->addEdge("Chicago", "Rome", -1);

}

catch (const std::exception& exc) {

exceptionThrown = true;

}

Assert::AreEqual(true, exceptionThrown);

}

TEST\_METHOD(InvalidFrom)

{

bool exceptionThrown = false;

try {

graph->addEdge(" ", "Rome", 10);

}

catch (const std::exception& exc) {

exceptionThrown = true;

}

Assert::AreEqual(true, exceptionThrown);

}

TEST\_METHOD(InvalidTo)

{

bool exceptionThrown = false;

try {

graph->addEdge("AAA", " ", 10);

}

catch (const std::exception& exc) {

exceptionThrown = true;

}

Assert::AreEqual(true, exceptionThrown);

}

private:

std::string workingdir(const string& file\_name)

{

char buf[256];

GetCurrentDirectoryA(2048, buf);

return std::string(buf) + '\\' + file\_name;

}

Graph\* graph;

};

}