**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра CАПР**

отчет

**по лабораторной работе №3**

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

**Вариант 1**

Тема: Алгоритмы на графах

|  |  |  |
| --- | --- | --- |
| Студент гр. 0302 |  | Головатюк К.А. |
| Преподаватель |  | Тутуева А.В |

Санкт-Петербург

2022

### Постановка задачи

Задача: 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 1: алгоритм Дейкстры и списки смежности

### Описание реализованных методов и оценка их временной сложности

Class ListAdjacency

|  |  |  |
| --- | --- | --- |
| Название публичного метода | Описание | Оценка временной сложности |
| bool input(string path) | Создание списка смежности (принимает путь к файлу) | O(n^2) |
| string Dijkstra(string \_a, string \_b) | Составление кратчайшего пути | O(n^3) |

|  |  |  |
| --- | --- | --- |
| Название приватного метода | Описание | Оценка временной сложности |
| string Dijkstra(string \_a, string \_b, int cost); | Алгоритм Дейкстры | O(n^2) |
| List\* add\_elem(string name) | Добавление элемента в список | O(n) |
| void add\_way(string a, string b, int cost) | Добавление пути в список | O(2\*n) |
| void sort\_list() | Сортировка списка | O(n^2) |
| void reset() | Очистка списка для поиска пути | O(n) |
| List\* search\_elem(string a) | Поиск элемента по имени | O(n) |

### Описание алгоритма Дейкстры и структур данных

1.Стоимость пути до начальной вершины равна 0, а до остальных бесконечности.

2.Проверяются смежные вершины, и если их стоимость пути до них больше, чем полученная (стоимость пути до текущей вершины + стоимость пути из текущей в смежную), то стоимость обновляется.

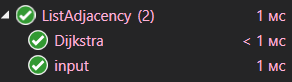
3.Проверяется следующая не посещенная вершина с минимальной стоимостью.

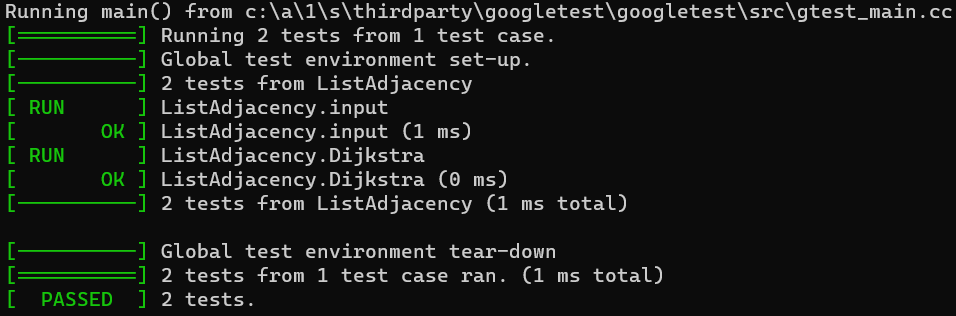
4.Если все вершины посещены, алгоритм завершается.

Для реализации был использован список смежности – это представление графа, в котором узел хранит список смежных к нему узлов.

### Описание реализованных unit-тестов

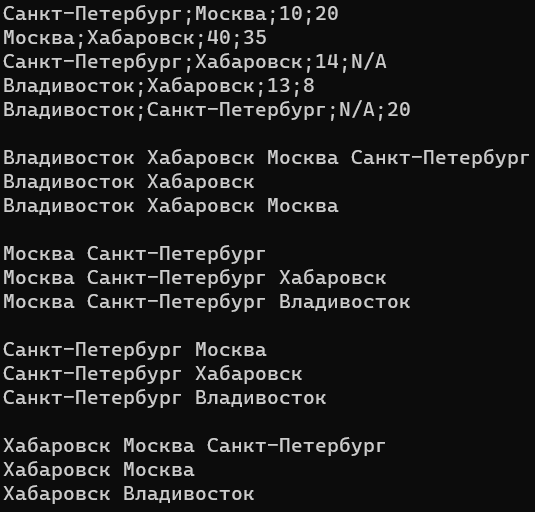
Список тестов:





Тесты проверяют публичные методы класса.

### Пример работы

****

**Вывод**

При реализации были улучшен навык создания структуры данных как список смежности. Изучен алгоритм Дейкстры.

### Листинг

Файл main.cpp:

#include <iostream>

#include "ListAdjacency.h"

using namespace std;

int main() {

setlocale(LC\_ALL, "Russian");

ListAdjacency test;

test.input("input.txt");

fstream file("input.txt");

if (!file.is\_open())

return 0;

string out;

while (getline(file, out)) {

cout << out << endl;

}

cout << endl;

cout << test.Dijkstra("Владивосток", "Санкт-Петербург") << endl;

cout << test.Dijkstra("Владивосток", "Хабаровск") << endl;

cout << test.Dijkstra("Владивосток", "Москва") << endl;

cout << endl;

cout << test.Dijkstra("Москва", "Санкт-Петербург") << endl;

cout << test.Dijkstra("Москва", "Хабаровск") << endl;

cout << test.Dijkstra("Москва", "Владивосток") << endl;

cout << endl;

cout << test.Dijkstra("Санкт-Петербург", "Москва") << endl;

cout << test.Dijkstra("Санкт-Петербург", "Хабаровск") << endl;

cout << test.Dijkstra("Санкт-Петербург", "Владивосток") << endl;

cout << endl;

cout << test.Dijkstra("Хабаровск", "Санкт-Петербург") << endl;

cout << test.Dijkstra("Хабаровск", "Москва") << endl;

cout << test.Dijkstra("Хабаровск", "Владивосток") << endl;

return 0;

}

Файл ListAdjacency.h:

#pragma once

#include <iostream>

#include <string>

#include <fstream>

using namespace std;

class Node;

class List {

public:

List() : node(nullptr), cost(99999999), next(nullptr) {};

Node\* node;

int cost;

List\* next;

};

class Node {

public:

Node() : name(""), visited(0), destination(nullptr) {};

string name;

bool visited;

List\* destination;

};

class ListAdjacency

{

public:

ListAdjacency() : all\_elements(nullptr) {};

bool input(string path);

string Dijkstra(string \_a, string \_b);

List\* all\_elements;

private:

string Dijkstra(string \_a, string \_b, int cost);

List\* search\_elem(string a);

List\* add\_elem(string name);

void add\_way(string a, string b, int cost);

void sort\_list();

void reset();

};

Файл ListAdjacency.cpp

#include "ListAdjacency.h"

bool ListAdjacency::input(string path){

ifstream file;

file.open(path);

if (!file.is\_open()) {

cout << "FILE ERORR" << endl;

return false;

}

string a;

string b;

int cost\_a\_b = 0;

int cost\_b\_a = 0;

string buff;

int pos = 0;

int check = 0;

string curent;

while (getline(file, curent)) {

for (int i = 0; i < curent.size(); i++) {

if (curent[i] == ';') {

if (check == 0) {

a = curent.substr(pos, i);

pos = i + 1;

check++;

}

else if (check == 1) {

b = curent.substr(pos, i - pos);

pos = i + 1;

check++;

}

else if (check == 2) {

buff = curent.substr(pos, i - pos);

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

cost\_a\_b = stoi(buff);

}

pos = i + 1;

check = 0;

}

}

}

buff = curent.substr(pos, curent.size() - pos);

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

cost\_b\_a = stoi(buff);

}

pos = 0;

add\_way(a, b, cost\_a\_b);

add\_way(b, a, cost\_b\_a);

cost\_a\_b = 0;

cost\_b\_a = 0;

}

return true;

}

List\* ListAdjacency::add\_elem(string name){

Node\* a = new Node();

a->name = name;

List\* lst = new List();

lst->node = a;

if (all\_elements == nullptr) {

all\_elements = lst;

return lst;

}

List\* iter = all\_elements;

while (iter->next){

iter = iter->next;

}

iter->next = lst;

return lst;

}

void ListAdjacency::add\_way(string \_a, string \_b, int cost){

if (cost == 0)

return;

Node\* node\_a;

Node\* node\_b;

List\* a;

List\* b;

a = search\_elem(\_a);

if (!a) {

node\_a = add\_elem(\_a)->node;

}

else {

node\_a = a->node;

}

b = search\_elem(\_b);

if (!b) {

node\_b = add\_elem(\_b)->node;

}

else {

node\_b = b->node;

}

List\* lst = new List();

lst->node = node\_b;

lst->cost = cost;

if (node\_a->destination == nullptr) {

node\_a->destination = lst;

return;

}

List\* buff = node\_a->destination;

while (buff->next)

{

buff = buff->next;

}

buff->next = lst;

}

List\* ListAdjacency::search\_elem(string a){

List\* buff = all\_elements;

while (buff)

{

if (buff->node->name == a) {

return buff;

}

buff = buff->next;

}

return nullptr;

}

void ListAdjacency::reset(){

List\* iter = all\_elements;

while (iter){

iter->cost = 99999999;

iter->node->visited = false;

iter = iter->next;

}

}

string ListAdjacency::Dijkstra(string \_a, string \_b){

if (!search\_elem(\_a))

return "Wrong way";

if (!search\_elem(\_b))

return "Wrong way";

reset();

string way;

search\_elem(\_a)->cost = 0;

string cur = Dijkstra(\_a, \_b, 0);

while (cur != \_a) {

way = cur + " " + way;

reset();

search\_elem(\_a)->cost = 0;

cur = Dijkstra(\_a, cur, 0);

}

return \_a + " " + way + \_b;

}

string ListAdjacency::Dijkstra(string \_a, string \_b, int cost)

{

List\* a = search\_elem(\_a);

a->node->visited = true;

List\* adjacency = a->node->destination;

bool change = false;

int curent\_cost;

while (adjacency)

{

List\* cur = search\_elem(adjacency->node->name);

curent\_cost = adjacency->cost + cost;

if (curent\_cost < cur->cost) {

cur->cost = curent\_cost;

if (cur->node->name == \_b) {

change = true;

}

}

adjacency = adjacency->next;

}

sort\_list();

List\* next = all\_elements;

string name = "";

while (next) {

if (!next->node->visited)

name = Dijkstra(next->node->name, \_b, next->cost);

next = next->next;

}

if (name != "")

return name;

if (change)

return search\_elem(\_a)->node->name;

return "";

}

void ListAdjacency::sort\_list(){

List\* iter = all\_elements;

while (iter->next)

{

List\* a = all\_elements;

List\* b = all\_elements->next;

while (b)

{

if (a->cost > b->cost) {

Node\* buff\_node = b->node;

int buff\_cost = b->cost;

b->node = a->node;

b->cost = a->cost;

a->node = buff\_node;

a->cost = buff\_cost;

}

a = a->next;

b = b->next;

}

iter = iter->next;

}

}

Файл test.cpp

#include "pch.h"

#include "ListAdjacency.h"

TEST(ListAdjacency, input) {

ListAdjacency test;

test.input("input.txt");

List\* iter = test.all\_elements;

ASSERT\_STREQ(iter->node->name.c\_str(), "Санкт-Петербург");

iter = iter->next;

ASSERT\_STREQ(iter->node->name.c\_str(), "Москва");

iter = iter->next;

ASSERT\_STREQ(iter->node->name.c\_str(), "Хабаровск");

iter = iter->next;

ASSERT\_STREQ(iter->node->name.c\_str(), "Владивосток");

}

TEST(ListAdjacency, Dijkstra) {

ListAdjacency test;

test.input("input.txt");

ASSERT\_STREQ(test.Dijkstra("Москва", "Хабаровск").c\_str(), "Москва Санкт-Петербург Хабаровск");

ASSERT\_STREQ(test.Dijkstra("Москва", "Санкт-Петербург").c\_str(), "Москва Санкт-Петербург");

ASSERT\_STREQ(test.Dijkstra("Москва", "Владивосток").c\_str(), "Москва Санкт-Петербург Владивосток");

}