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 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)

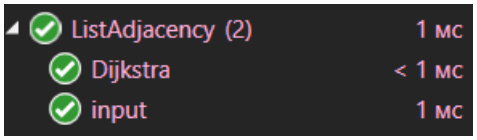
Описание алгоритма Дейкстры и структур данных

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:




```
Running main() from c:\a\1\s\thirdparty\googletest\googletest\src\gtest_main.cc
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from ListAdjacency
[ RUN      ] ListAdjacency.input
[       OK ] ListAdjacency.input (1 ms)
[ RUN      ] ListAdjacency.Dijkstra
[       OK ] ListAdjacency.Dijkstra (0 ms)
[----------] 2 tests from ListAdjacency (1 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test case ran. (1 ms total)
[  PASSED  ] 2 tests.
```

Тесты проверяют публичные методы класса.

Пример работы

```
Санкт-Петербург;Москва;10;20
Москва;Хабаровск;40;35
Санкт-Петербург;Хабаровск;14;N/A
Владивосток;Хабаровск;13;8
Владивосток;Санкт-Петербург;N/A;20

Владивосток Хабаровск Москва Санкт-Петербург
Владивосток Хабаровск
Владивосток Хабаровск Москва

Москва Санкт-Петербург
Москва Санкт-Петербург Хабаровск
Москва Санкт-Петербург Владивосток

Санкт-Петербург Москва
Санкт-Петербург Хабаровск
Санкт-Петербург Владивосток

Хабаровск Москва Санкт-Петербург
Хабаровск Москва
Хабаровск Владивосток
```

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");
}
```