МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра САПР

ОТЧЕТ

по лабораторной работе №3
по дисциплине «Алгоритмы на графах»
Вариант 3

Студент гр. 8301 _____ Попурей Н.

Преподаватель _____ Тутуева А.В.

Санкт-Петербург
2020

Оглавление

1) Постановка задачи

Реализовать программу принимающую список рейсов и цены за прямой и обратный и рейс и, в которой пользователь в свою очередь вводит город отправления и назначения и получает самый выгодный рейс или получает информацию о невозможности совершения перелётов методом Флойда-Уоршелла.

2) Оценка временной сложности

string Floid_Uorshell – O(N^3)

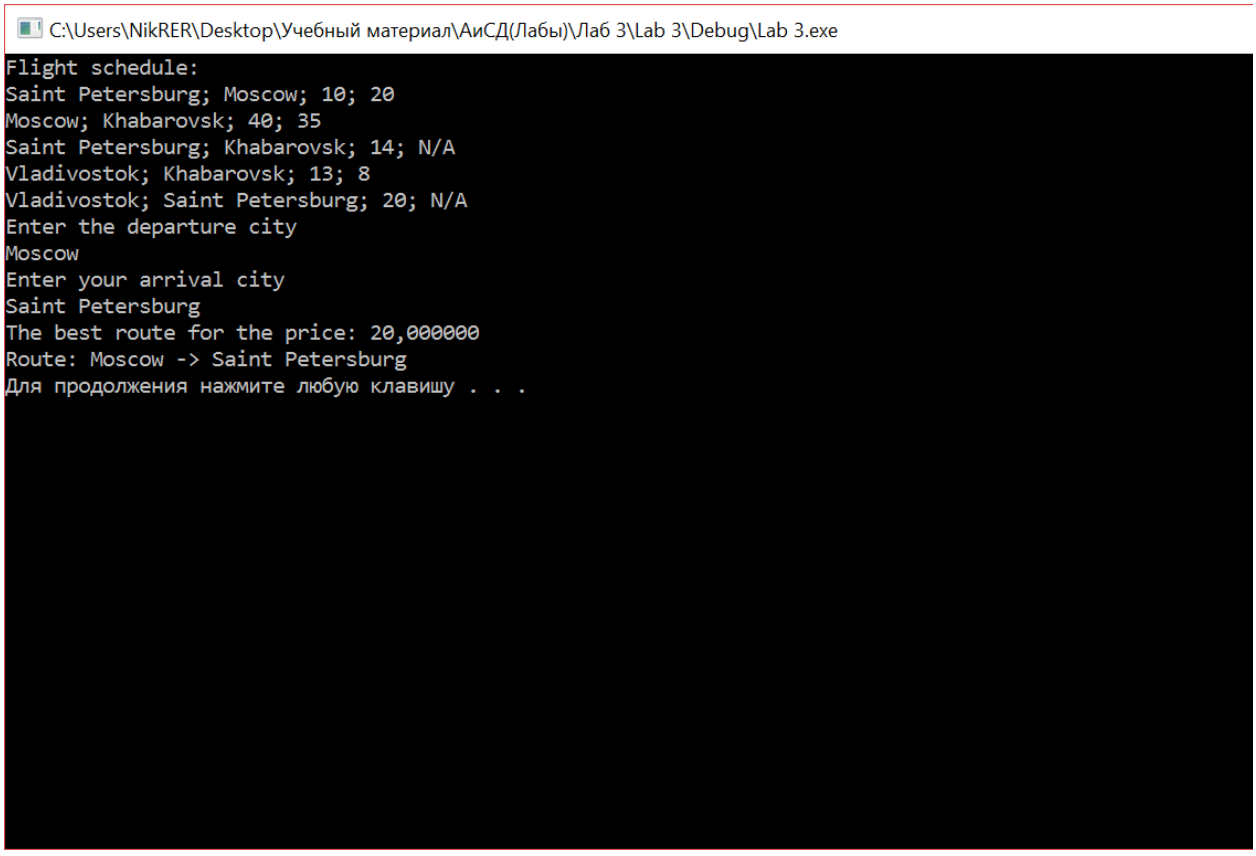
get_list_symbol() – O(1)

print_path – O(N^2)

3) Описание реализованных юнит-тестов

Реализованные мною тесты проверяют правильное нахождение выгодного перелёта. Я рассмотрел две ситуации когда перелёт возможен и когда нет.

4) Пример работы



Листинг

Lab3.cpp

```
#include "pch.h"
#include <iostream>
#include <fstream>
#include <string>
#include "Used_function.h"
#include "matrix_of_adjacencies.h"
using namespace std;
int main() {
	setlocale(LC_ALL, "RUS");
	ifstream vvod("input.txt");
	List<string>* list_fly = new List<string>();
	string city_Start;
	string city_End;
	InputDataFromFile(list_fly, vvod);
	cout << "Flight schedule: " << endl;
	for (int i = 0; i < list_fly->get_size(); i++)
		cout << list_fly->at(i) << endl;
	cout << "Enter the departure city" << endl;
	getline(cin, city_Start);
	cout << "Enter your arrival city" << endl;
	getline(cin, city_End);
	Matrix* matrix_floid_uorshell = new Matrix(list_fly);
	cout << matrix_floid_uorshell->Floid_Uorshell(city_Start, city_End) << endl;
	system("pause");
    }
```

List.h

```
#pragma once
#include<iostream>
using namespace std;
template<class T>
class List
{
private:
	class Node {
	public:
		Node(T data = T(), Node* Next = NULL) {
			this->data = data;
			this->Next = Next;
		}
		Node* Next;
		T data;
	};
public:
	void push_back(T obj) { // добавление в конец списка bc
		if (head != NULL) {
			this->tail->Next = new Node(obj);
			tail = tail->Next;
		}
		else {
			this->head = new Node(obj);
			this->tail = this->head;
		}
		Size++;
	}
	void push_front(T obj) { // добавление в начало списка bc
		if (head != NULL) {
			Node* current = new Node;
```

```
                current->data = obj;
                current->Next = this->head;
                this->head = current;
            }
            else {
                this->head = new Node(obj);
            }
            this->Size++;
        }
        void pop_back() { // удаление последнего элемента bc
            if (head != NULL) {
                Node* current = head;
                while (current->Next != tail)//то есть ищем предпоследний
                    current = current->Next;
                delete tail;
                tail = current;
                tail->Next = NULL;
                Size--;
            }
            else throw std::out_of_range("out_of_range");
        }
        void pop_front() { // удаление первого элемента bc-+
            if (head != NULL) {
                Node* current = head;
                head = head->Next;
                delete current;
                Size--;
            }
            else throw std::out_of_range("out_of_range");
        }
        void insert(T obj, size_t k) {// добавление элемента по индексу (вставка перед
элементом, который был ранее доступен по этому индексу) bc
            if (k >= 0 && this->Size > k) {
                if (this->head != NULL) {
                    if (k == 0)
                        this->push_front(obj);
                    else
                        if (k == this->Size - 1)
                            this->push_back(obj);
                        else
                        {
                            Node* current = new Node;//для добавления
элемента
                            Node* current1 = head;//для поиска итого
элемента
                            for (int i = 0; i < k - 1; i++) {
                                current1 = current1->Next;
                            }
                            current->data = obj;
                            current->Next = current1->Next;//переуказывает
на след элемент
                            current1->Next = current;
                            Size++;
                        }
                }
            }
            else {
                throw std::out_of_range("out_of_range");
            }
        }
        T at(size_t k) {// получение элемента по индексу bc
            if (this->head != NULL && k >= 0 && k <= this->Size - 1) {
                if (k == 0)
                    return this->head->data;
                else
```

```
                        if (k == this->Size - 1)
                                return this->tail->data;
                        else
                        {
                                Node* current = head;
                                for (int i = 0; i < k; i++) {
                                        current = current->Next;
                                }
                                return current->data;
                        }
                }
                else {
                        throw std::out_of_range("out_of_range");
                }
        }
        void remove(int k) { // удаление элемента по индексу bc
                if (head != NULL && k >= 0&&k<=Size-1) {
                        if (k == 0) this->pop_front();
                        else
                                if (k == this->Size - 1) this->pop_back();
                                else
                                        if (k != 0) {
                                                Node* current = head;
                                                for (int i = 0; i < k - 1; i++) {//переходим на
предэлемент
                                                        current = current->Next;
                                                }

                                                Node* current1 = current->Next;
                                                current->Next = current->Next->Next;
                                                delete current1;
                                                Size--;
                                        }
                }
                else {
                        throw std::out_of_range("out_of_range");
                }
        }
        size_t get_size() { // получение размера списка bc
                return Size;
        }
        void print_to_console() { // вывод элементов списка в консоль через разделитель,
не использовать at bc
                if (this->head != NULL) {
                        Node* current = head;
                        for (int i = 0; i < Size; i++) {
                                cout << current->data << ' ';
                                current = current->Next;
                        }
                }
        }
        void clear() { // удаление всех элементов списка
                if (head != NULL) {
                        Node* current = head;
                        while (head != NULL) {
                                current = current->Next;
                                delete head;
                                head = current;
                        }
                        Size = 0;
                }
        }
        void set(size_t k, T obj)  // замена элемента по индексу на передаваемый элемент
        {
                if (this->head != NULL && this->get_size() >= k && k >= 0) {
```

```
                Node* current = head;
                for (int i = 0; i < k; i++) {
                        current = current->Next;
                }
                current->data = obj;
        }
        else {
                throw std::out_of_range("out_of_range");
        }
    }
    bool isEmpty() { // проверка на пустоту списка bc
        return (bool)(head);
    }
    void reverse() { // меняет порядок элементов в списке
        int Counter = Size;
        Node* HeadCur = NULL;
        Node* TailCur = NULL;
        for (int j = 0; j <Size; j++) {
            if (HeadCur != NULL) {
                if(head!=NULL&&head->Next==NULL){
                    TailCur->Next = head;
                    TailCur = head;
                    head = NULL;
                }
                else {
                        Node * cur = head;
                    for (int i = 0; i < Counter - 2; i++)
                        cur = cur->Next;
                    TailCur->Next = cur->Next;
                    TailCur = cur->Next;
                    cur->Next = NULL;
                    tail = cur;
                    Counter--;
                }
            }
            else {
                HeadCur = tail;
                TailCur = tail;
                Node* cur = head;
                for (int i = 0; i < Size - 2; i++)
                    cur = cur->Next;
                tail = cur;
                tail->Next = NULL;
                Counter--;
            }
        }
        head = HeadCur;
        tail = TailCur;
    }
public:
    List(Node* head = NULL, Node* tail = NULL, int Size = 0) :head(head), tail(tail),
Size(Size) {}
    ~List() {
        if (head != NULL) {
            this->clear();
        }
    };
private:
    Node* head;
    Node* tail;
    int Size;
};
```

Map.h

```
#pragma once
#define COLOR_RED 1
#define COLOR_BLACK 0
#include"List.h"
using namespace std;
template<typename T, typename T1>
class Map {
public:
	class Node
	{
	public:
		Node(bool color = COLOR_RED, T key = T(), Node* parent = NULL, Node* left = NULL, Node* right = NULL, T1 value = T1()) :color(color), key(key), parent(parent), left(left), right(right), value(value) {}
		T key;
		T1 value;
		bool color;
		Node* parent;
		Node* left;
		Node* right;
	};
	~Map() {
		if (this->Top != NULL)
			this->clear();
		Top = NULL;
		delete TNULL;
		TNULL = NULL;
	}
	Map(Node* Top = NULL, Node* TNULL = new Node(0)) :Top(TNULL), TNULL(TNULL) {}

	void printTree()
	{
		if (Top)
		{
			print_Helper(this->Top, "", true);
		}
		else throw std::out_of_range("Tree is empty!");
	}

	void  insert(T key, T1 value)
	{

		if (this->Top != TNULL) {
			Node* node = NULL;
			Node* parent=NULL;
			/* Search leaf for new element */
			for (node = this->Top; node != TNULL; )
			{
				parent = node;
				if (key < node->key)
					node = node->left;
				else if (key > node->key)
					node = node->right;
				else if (key == node->key)
					throw std::out_of_range("key is repeated");
			}

			node = new Node(COLOR_RED, key, TNULL, TNULL, TNULL, value);
			node->parent = parent;


			if (parent != TNULL) {
				if (key < parent->key)
```

```
                                        parent->left = node;
                                else
                                        parent->right = node;
                        }
                        rbtree_fixup_add(node);
                }
                else {
                        this->Top = new Node(COLOR_BLACK, key, TNULL, TNULL, TNULL, value);
                }
        }
        List<T>* get_keys() {
                List<T>* list = new List<T>();
                this->ListKeyOrValue(1,list);
                return list;
        }
        List<T1>* get_values() {
                List<T1>* list = new List<T1>();
                this->ListKeyOrValue(2, list);
                return list;
        }
        T1 find(T key) {
                Node* node = Top;

                while (node != TNULL && node->key != key) {
                        if (node->key > key)
                                node = node->left;
                        else
                                if (node->key < key)
                                        node = node->right;
                }
                if (node != TNULL)
                        return node->value;
                else
                        throw std::out_of_range("Key is missing");
        }
        bool find_is(T key) {
                Node* node = Top;

                while (node != TNULL && node->key != key) {
                        if (node->key > key)
                                node = node->left;
                        else
                                if (node->key < key)
                                        node = node->right;
                }
                if (node != TNULL)
                        return true;
                else
                        return false;
        }
        void remove(T key) {
                this->deleteNodeHelper(this->find_key(key));
        }

        void clear() {
                this->clear_tree(this->Top);
                this->Top = NULL;
        }
private:
        Node* Top;
        Node* TNULL;

////////////////////////////////////////////////////////////////////////
        //delete functions
////////////////////////////////////////////////////////////////////////
```

```
void deleteNodeHelper(Node* find_node)
{
	Node* node_with_fix, * cur_for_change;
	cur_for_change = find_node;
	bool cur_for_change_original_color = cur_for_change->color;
	if (find_node->left == TNULL)
	{
		node_with_fix = find_node->right;
		Transplant(find_node, find_node->right);
	}
	else if (find_node->right == TNULL)
	{
		node_with_fix = find_node->left;
		Transplant(find_node, find_node->left);
	}
	else

	{
		cur_for_change = minimum(find_node->right);
		cur_for_change_original_color = cur_for_change->color;
		node_with_fix = cur_for_change->right;
		if (cur_for_change->parent == find_node)
		{
			node_with_fix->parent = cur_for_change;
		}
		else
		{
			Transplant(cur_for_change, cur_for_change->right);
			cur_for_change->right = find_node->right;
			cur_for_change->right->parent = cur_for_change;
		}
		Transplant(find_node, cur_for_change);
		cur_for_change->left = find_node->left;
		cur_for_change->left->parent = cur_for_change;
		cur_for_change->color = find_node->color;
	}
	delete find_node;
	if (cur_for_change_original_color == COLOR_BLACK)
	{
		this->rbtree_fixup_add(node_with_fix);
	}
}
//swap links(parent and other) for rotate
void Transplant(Node* cur, Node* cur1)
{
	if (cur->parent == TNULL)
	{
		Top = cur1;
	}
	else if (cur == cur->parent->left)
	{
		cur->parent->left = cur1;
	}
	else
	{
		cur->parent->right = cur1;
	}
	cur1->parent = cur->parent;
}


void clear_tree(Node* tree) {
	if (tree != TNULL) {
		clear_tree(tree->left);
		clear_tree(tree->right);
```

```
                delete tree;
            }
        }
////////////////////////////////////////////////////////////////////////////////////////
        //all find functions
////////////////////////////////////////////////////////////////////////////////////////
        Node* minimum(Node* node)
        {
            while (node->left != TNULL)
            {
                node = node->left;
            }
            return node;
        }
        Node* maximum(Node* node)
        {
            while (node->right != TNULL)
            {
                node = node->right;
            }
            return node;
        }
        Node* grandparent(Node* cur)
        {
            if ((cur != TNULL) && (cur->parent != TNULL))
                return cur->parent->parent;
            else
                return TNULL;
        }
        Node* uncle(Node* cur)
        {
            Node* cur1 = grandparent(cur);
            if (cur1 == TNULL)
                return TNULL; // No grandparent means no uncle
            if (cur->parent == cur1->left)
                return cur1->right;
            else
                return cur1->left;
        }
        Node* sibling(Node* n)
        {
            if (n == n->parent->left)
                return n->parent->right;
            else
                return n->parent->left;
        }
        Node* find_key(T key) {
            Node* node = this->Top;
            while (node != TNULL && node->key != key) {
                if (node->key > key)
                    node = node->left;
                else
                    if (node->key < key)
                        node = node->right;
            }
            if (node != TNULL)
                return node;
            else
                throw std::out_of_range("Key is missing");
        }
        ////////////////////////////////////////////////////////////////////////////////////
///
          //all print function
        ////////////////////////////////////////////////////////////////////////////////////
///
```

```
	void print_Helper(Node* root, string indent, bool last)
	{
		if (root != TNULL)
		{
			cout << indent;
			if (last)
			{
				cout << "R----";
				indent += "   ";
			}
			else
			{
				cout << "L----";
				indent += "|  ";
			}
			string sColor = !root->color ? "BLACK" : "RED";
			cout << root->key << "(" << sColor << ")" << endl;
			print_Helper(root->left, indent, false);
			print_Helper(root->right, indent, true);
		}
	}
	void ListKeyOrValue(int mode,List<T>*list) {
		if (this->Top != TNULL)
			this->KeyOrValue(Top,list, mode);
		else
			throw std::out_of_range("Tree empty!");
	}
	void KeyOrValue(Node* tree,List<T>*list, int mode) {

		if (tree != TNULL) {
			KeyOrValue(tree->left,list, mode);
			if (mode == 1)
				list->push_back(tree->key);
			else
				list->push_back(tree->value);
			KeyOrValue(tree->right,list, mode);
		}
	}
	////////////////////////////////////////////////////////////////////////////////
/////
	  //fix before add
	////////////////////////////////////////////////////////////////////////////////
////////////////////////////
	void rbtree_fixup_add(Node* node)
	{
		Node* uncle;
		/* Current node is COLOR_RED */
		while (node != this->Top && node->parent->color == COLOR_RED)//
		{
			/* node in left tree of grandfather */
			if (node->parent == this->grandparent(node)->left)//
			{
				/* node in left tree of grandfather */
				uncle = this->uncle(node);
				if (uncle->color == COLOR_RED) {
					/* Case 1 - uncle is COLOR_RED */
					node->parent->color = COLOR_BLACK;
					uncle->color = COLOR_BLACK;
					this->grandparent(node)->color = COLOR_RED;
					node = this->grandparent(node);
				}
				else {
					/* Cases 2 & 3 - uncle is COLOR_BLACK */
					if (node == node->parent->right) {
						/*Reduce case 2 to case 3 */
```

```
                                        node = node->parent;
                                        this->left_rotate(node);
                                }
                                /* Case 3 */
                                node->parent->color = COLOR_BLACK;
                                this->grandparent(node)->color = COLOR_RED;
                                this->right_rotate(this->grandparent(node));
                        }
                }
                else {
                        /* Node in right tree of grandfather */
                        uncle = this->uncle(node);
                        if (uncle->color == COLOR_RED) {
                                /* Uncle is COLOR_RED */
                                node->parent->color = COLOR_BLACK;
                                uncle->color = COLOR_BLACK;
                                this->grandparent(node)->color = COLOR_RED;
                                node = this->grandparent(node);
                        }
                        else {
                                /* Uncle is COLOR_BLACK */
                                if (node == node->parent->left) {
                                        node = node->parent;
                                        this->right_rotate(node);
                                }
                                node->parent->color = COLOR_BLACK;
                                this->grandparent(node)->color = COLOR_RED;
                                this->left_rotate(this->grandparent(node));
                        }
                }
        }
        this->Top->color = COLOR_BLACK;
}
////////////////////////////////////////////////////////////////////////////
  //Rotates
////////////////////////////////////////////////////////////////////////////
//(left rotate)
void left_rotate(Node* node)
{
        Node* right = node->right;
        /* Create node->right link */
        node->right = right->left;
        if (right->left != TNULL)
                right->left->parent = node;
        /* Create right->parent link */
        if (right != TNULL)
                right->parent = node->parent;
        if (node->parent != TNULL) {
                if (node == node->parent->left)
                        node->parent->left = right;
                else
                        node->parent->right = right;
        }
        else {
                this->Top = right;
        }
        right->left = node;
        if (node != TNULL)
                node->parent = right;
}
//(right rotate)
void right_rotate(Node* node)
{
        Node* left = node->left;
        /* Create node->left link */
```

```
		node->left = left->right;
		if (left->right != TNULL)
			left->right->parent = node;
		/* Create left->parent link */
		if (left != TNULL)
			left->parent = node->parent;
		if (node->parent != TNULL) {
			if (node == node->parent->right)
				node->parent->right = left;
			else
				node->parent->left = left;
		}
		else {
			this->Top = left;
		}
		left->right = node;
		if (node != TNULL)
			node->parent = left;
	}
	////////////////////////////////////////////////////////////////////////////////
////////////////////////////
    };
```

Matrix_of_adjacencies.h

```
#pragma once
#include"List.h"
#include"Map.h"
#include<string>
class Matrix {
public:
	Matrix(List<string>* data) {
			map_City_name_to_index=new Map<string, int>();
			map_index_to_name_City=new Map<int, string>();
			int N = data->get_size();
			int index_city = 0;
			for (int i = 0; i < N; i++) {
				string str_cur = data->at(i);
				int cur = str_cur.find(';');//the first occurrence
				int cur1 = str_cur.find(';', cur + 1);//the second occurrence
				string str_name_city1 = str_cur.substr(0, cur);//get first
city
				string str_name_city2 = str_cur.substr(cur + 1, cur1 - cur -
1);//get second city
				str_name_city2.erase(0, 1);
				if (!map_City_name_to_index->find_is(str_name_city1)) {
					map_City_name_to_index->insert(str_name_city1,
index_city);
					map_index_to_name_City->insert(index_city,
str_name_city1);
					index_city++;
				}
				if (!map_City_name_to_index->find_is(str_name_city2)) {
					map_City_name_to_index->insert(str_name_city2,
index_city);
					map_index_to_name_City->insert(index_city,
str_name_city2);
					index_city++;
				}
			}

	////////////////////////////////////////////////////////////////////////////////make
matrix path
			size_of_matrix = index_city;
			matrix = new double* [size_of_matrix];
```

```
                    for (int i = 0; i < size_of_matrix; i++)
                           matrix[i] = new double[size_of_matrix];
                    for (int i = 0; i < size_of_matrix; i++)
                           for (int j = 0; j < size_of_matrix; j++)
                                  matrix[i][j] = INF;

      ////////////////////////////////////////////////////////////////////////////


      ////////////////////////////////////////////////////////////////////////////input matrix path
                    for (int i = 0; i < N; i++) {
                           int price_1_to_2 = INF;
                           int price_2_to_1 = INF;
                           string str_cur = data->at(i);
                           int cur = str_cur.find(';');
                           int cur1 = str_cur.find(';', cur + 1);
                           int cur2 = str_cur.find(';', cur1 + 1);
                           int cur3 = str_cur.find(';', cur2 + 1);
                           string str_name_city1 = str_cur.substr(0, cur);
                           string str_name_city2 = str_cur.substr(cur + 1, cur1 - cur - 1);
                           str_name_city2.erase(0, 1);
                           if (str_cur.substr(cur1 + 2, cur2 - 2 - cur1) != "N/A")
                                  price_1_to_2 = stof(str_cur.substr(cur1 + 2, cur2 - 2 - cur1));
                           if (str_cur.substr(cur2 + 2, cur3 - 1) != "N/A")
                                  price_2_to_1 = stoi(str_cur.substr(cur2 + 2, cur3 - 2 - cur2));

                           matrix[map_City_name_to_index->find(str_name_city1)][map_City_name_to_index->find(str_name_city2)] = price_1_to_2;

                           matrix[map_City_name_to_index->find(str_name_city2)][map_City_name_to_index->find(str_name_city1)] = price_2_to_1;
                    }

      ////////////////////////////////////////////////////////////////////////////////////////////////////
      }
      string Floid_Uorshell(string start_City,string end_City) {
            string cur;
            while (!map_City_name_to_index->find_is(start_City)) {
                  cout << "The departure city is missing, enter it again" << endl;
                  cin >> start_City;
            }
            while (!map_City_name_to_index->find_is(end_City)) {
                  cout << "The arrival city is missing, enter it again" << endl;
                  cin >> end_City;
            }
            int index_start_vertex = map_City_name_to_index->find(start_City);
            int index_end_vertex = map_City_name_to_index->find(end_City);
            int** pre = new int* [size_of_matrix];
            for (int i = 0; i < size_of_matrix; i++) {
                  pre[i] = new int[size_of_matrix];
                  for (int j = 0; j < size_of_matrix; j++)
                        pre[i][j] = i;
            }
            for (int k = 0; k < size_of_matrix; ++k)
                  for (int i = 0; i < size_of_matrix; ++i)
                        for (int j = 0; j < size_of_matrix; ++j) {
                              if (matrix[i][k] + matrix[k][j] < matrix[i][j]) {
                                    matrix[i][j] = matrix[i][k] + matrix[k][j];
                                    pre[i][j] = pre[k][j];
                              }
```

```
                        }
            if (matrix[map_City_name_to_index-
>find(start_City)][map_City_name_to_index->find(end_City)] != INF) {
                    cur = "The best route for the price: " +
to_string(matrix[map_City_name_to_index->find(start_City)][map_City_name_to_index-
>find(end_City)]) + '\n' + "Route: ";
                    print_path(index_start_vertex, index_end_vertex, pre,
map_index_to_name_City, cur);
                    cur.erase(cur.size() - 3);
            }
            else {
                    cur = "This route can't be built, try waiting for the flight schedule
for tomorrow!";
            }
            return cur;
      }
private:
      void print_path(int i, int j, int** p, Map<int, string>*
map_index_to_name_City,string&cur) {
            if (i != j)
                    print_path(i, p[i][j], p, map_index_to_name_City,cur);
            cur=cur+map_index_to_name_City->find(j)+" -> ";
      }
      double** matrix;
      int size_of_matrix;
      Map<string, int>* map_City_name_to_index;
      Map<int, string>* map_index_to_name_City;
      const int INF = 1000000000;
   };
```

Used_function.h

```
#pragma once
#include<string>
#include <fstream>
#include"List.h"
void InputDataFromFile(List<string>* data, ifstream& file) {//ввод из файла
      while (!file.eof()) {
            string s1;
            getline(file, s1);
            data->push_back(s1);
      }
   }
```

UnitTest.cpp

```
#include "stdafx.h"
#include "CppUnitTest.h"
#include <fstream>
#include<string>
#include"../Lab 3/matrix_of_adjacencies.h"
#include"../Lab 3/Used_function.h"
using namespace Microsoft::VisualStudio::CppUnitTestFramework;
namespace UnitTestForAlgorithmFloydUorshell
{
	TEST_CLASS(UnitTestForAlgorithmFloydUorshell)
	{
	public:

		TEST_METHOD(TestExamplePath_is_avaible)
		{
			ifstream vvod("C:\\Users\\NikRER\\Desktop\\Учебный материал\\АиСД(Лабы)\\Лаб 3\\Lab 3\\UnitTest1\\input1.txt");
			List<string>* list_fly = new List<string>();
			string city_Start = "Vladivostok";
			string city_End = "Moscow";
			InputDataFromFile(list_fly, vvod);
			Matrix* matrix_floid_uorshell = new Matrix(list_fly);
			string cur = "The best route for the price: 30.000000\nRoute: Vladivostok -> Saint Petersburg -> Moscow ";
			Assert::AreEqual(matrix_floid_uorshell->Floid_Uorshell(city_Start, city_End), cur);
		}
		TEST_METHOD(TestExamplePath_is_not_avaible)
		{
			ifstream vvod("C:\\Users\\NikRER\\Desktop\\Учебный материал\\АиСД(Лабы)\\Лаб 3\\Lab 3\\UnitTest1\\input2.txt");
			List<string>* list_fly = new List<string>();
			string city_Start = "Tambov";
			string city_End = "Saint Petersburg";
			InputDataFromFile(list_fly, vvod);
			Matrix* matrix_floid_uorshell = new Matrix(list_fly);
			string cur = "This route can't be built, try waiting for the flight schedule for tomorrow!";
			Assert::AreEqual(matrix_floid_uorshell->Floid_Uorshell(city_Start, city_End), cur);
		}
	};
}
```

Вывод:

В данной лабораторной работе я ознакомился с алгоритмом Флойда-Уоршелла и смог применить его на примере нахождения выгодного пути из авиарейсов, а также закрепил свои навыки в объектно-ориентированном программировании.