**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра САПР**

**ОТЧЕТ**

**по лабораторной работе №3**

**по дисциплине «АиСД»**

**Тема: Алгоритмы на графах**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 8302 |  | Зубарева Е.А. |
| Преподаватель |  | Тутуева А.В. |

**Оглавление**

[Постановка задачи 2](#_Toc37951820)

[Описание классов и методов 2](#_Toc37951821)

[Оценка временной сложности 3](#_Toc37951822)

[Описание unit-тестов 3](#_Toc37951823)

[Пример работы 4](#_Toc37951824)

[Код 5](#_Toc37951825)

## 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)

Требуется найти наиболее дешевый путь из i в j используя алгоритм Дейкстры и списки смежности.

## Описание классов и методов

Реализованы следующие классы:

1. Graph – граф, который содержит спискок (List) вершин (Graph::Vertex). Позволяет добавлять вершины, искать вершины по имени города и читать граф из файла.
2. Graph::Vertex – вершина графа, содержит имя города и список (List) прилегающих ребер (Graph::Edge). Вершина так же содержит временные значения для алгоритма Дейкстры: вес вершины и массив указателей на вершины (Array<Graph::Vertex\*>), содержащий путь до вершины.
3. Graph::Edge – одностороннее ребро графа, которое по сути является парой указатель на вершину/вес ребра.
4. Неймспейс DijkstraAlgorithm, содержащий метод find\_path, сбрасывающий временные значения в вершинах и реализующий алгоритм дейкстры для поиска кратчайшего пути.

Так же из предыдущих лабораторной работы были использованы классы Array<T> и List<T>.

## Оценка временной сложности

## Операции с Array – O(1), кроме случая, когда массив расширяется при нехватке размера, время этой операции – O(N)

## Сложность алгоритма Дейкстры – O(N^2) – для каждой вершины обходим всех соседей.

## Описание unit-тестов

Первая часть тестов взята из 2 лабораторной работы для общей проверки работы Array. Далее проводится тест на построение графа и поиск вершин по именам, после чего – тест поиска пути в заданном наперед графе.

## Пример работы

Graph:

commands:

1 - find path

2 - open file

other - exit

input command:2

input file name:../input.txt

success

Graph:

flights from Saint-Petersburg: Moscow = 10, Khabarovsk = 14, Vladivostok = 20,

flights from Moscow: Saint-Petersburg = 20, Khabarovsk = 40,

flights from Khabarovsk: Moscow = 35, Vladivostok = 8,

flights from Vladivostok: Khabarovsk = 13,

commands:

1 - find path

2 - open file

other - exit

input command:1

input city #1:Vladivostok

input city #2:Moscow

path cost: 48

path: Vladivostok Khabarovsk Moscow

Graph:

flights from Saint-Petersburg: Moscow = 10, Khabarovsk = 14, Vladivostok = 20,

flights from Moscow: Saint-Petersburg = 20, Khabarovsk = 40,

flights from Khabarovsk: Moscow = 35, Vladivostok = 8,

flights from Vladivostok: Khabarovsk = 13,

commands:

1 - find path

2 - open file

other - exit

input command:1

input city #1:Khabarovsk

input city #2:Saint-Petersburg

path cost: 55

path: Khabarovsk Moscow Saint-Petersburg

## Код

main.cpp

#include <iostream>  
#include "graph.h"  
#include "dijkstra.h"  
  
  
int main() {  
 Graph graph;  
  
 bool run = true;  
 while (run) {  
 std::cout << "\n\n\nGraph:\n";  
 graph.print();  
 std::cout << "\n\ncommands: \n1 - find path\n2 - open file\nother - exit\n\ninput command: ";  
  
 int cmd;  
 std::cin >> cmd;  
 switch (cmd) {  
 case 1: {  
 std::string city;  
 std::cout << "input city #1: ";  
 std::cin >> city;  
 Graph::Vertex \*vertex1 = graph.get\_vertex(city);  
 if (vertex1 == nullptr) {  
 std::cout << "no such city!";  
 break;  
 }  
 std::cout << "input city #2: ";  
 std::cin >> city;  
 Graph::Vertex \*vertex2 = graph.get\_vertex(city);  
 if (vertex2 == nullptr) {  
 std::cout << "no such city!";  
 break;  
 }  
 auto path = DijkstraAlgorithm::find\_path(graph, \*vertex1, \*vertex2);  
 if (path.length() > 0) {  
 double path\_weight = vertex2->dijkstra\_weight;  
 std::cout << "path cost: " << path\_weight << "\n";  
 std::cout << "path: ";  
 for (int i = 0; i < path.length(); i++) {  
 std::cout << " " << path[i]->city;  
 }  
 std::cout << "\n";  
 } else {  
 std::cout << "no path exists.\n";  
 }  
 break;  
 }  
 case 2: {  
 std::string filename;  
 std::cout << "input file name: ";  
 std::cin >> filename;  
 if (graph.fromFile(filename)) {  
 std::cout << "success\n";  
 } else {  
 std::cout << "failed to read\n";  
 }  
 break;  
 }  
 default:  
 run = false;  
 break;  
 }  
 std::cout << "\n";  
 }  
  
 return 0;  
}

tests.cpp

#include "gtest/gtest.h"  
#include "../array.h"  
#include "../dijkstra.h"  
  
  
TEST(Graph, Pathfinding) {  
 Graph graph;  
 Graph::Vertex\* vertex1 = graph.get\_or\_add\_vertex("city1");  
 Graph::Vertex\* vertex2 = graph.get\_or\_add\_vertex("city2");  
 Graph::Vertex\* vertex3 = graph.get\_or\_add\_vertex("city3");  
 Graph::Vertex\* vertex4 = graph.get\_or\_add\_vertex("city4");  
 vertex1->connect(\*vertex2, 10);  
 vertex1->connect(\*vertex3, 1);  
 vertex3->connect(\*vertex2, 2);  
 vertex2->connect(\*vertex4, 20);  
  
 Array<Graph::Vertex\*> path = DijkstraAlgorithm::find\_path(graph, \*vertex1, \*vertex4);  
 std::cout << "test path: ";  
 for (int i = 0; i < path.length(); i++) {  
 std::cout << " " << path[i]->city;  
 }  
 std::cout << "\n";  
 ASSERT\_EQ(path.length(), 4);  
 ASSERT\_EQ(path[0], vertex1);  
 ASSERT\_EQ(path[1], vertex3);  
 ASSERT\_EQ(path[2], vertex2);  
 ASSERT\_EQ(path[3], vertex4);  
  
 path = DijkstraAlgorithm::find\_path(graph, \*vertex4, \*vertex1);  
 ASSERT\_EQ(path.length(), 0);  
}  
  
TEST(Graph, BuildSimpleGraph) {  
 Graph graph;  
 Graph::Vertex\* vertex1 = graph.get\_or\_add\_vertex("city1");  
 Graph::Vertex\* vertex2 = graph.get\_or\_add\_vertex("city2");  
 Graph::Vertex\* vertex3 = graph.get\_or\_add\_vertex("city3");  
  
 ASSERT\_EQ(vertex2, graph.get\_or\_add\_vertex("city2")); // duplicate  
 ASSERT\_EQ(graph.vertices.get\_length(), 3);  
}  
  
// array tests  
TEST(Array, MassiveAddAndClearTest) {  
 Array<std::string> array;  
 const int size = 10000;  
 for (int i = 0; i < size; i++) {  
 array.add("test\_string");  
 }  
 ASSERT\_EQ(array.length(), size);  
 for (int i = 0; i < size; i++) {  
 ASSERT\_EQ(array[i], "test\_string");  
 }  
 array.clear();  
 ASSERT\_EQ(array.length(), 0);  
}

graph.h

#include <string.h>  
#include <fstream>  
#include "array.h"  
#include "list.h"  
  
#ifndef SEM4LAB3\_GRAPH\_H  
#define SEM4LAB3\_GRAPH\_H  
  
class Graph {  
public:  
 class Vertex;  
  
 struct Edge {  
 public:  
 Vertex\* vertex = nullptr;  
 double weight = 0;  
  
 Edge();  
 Edge(Vertex& vertex, double weight);  
 };  
  
 class Vertex {  
 public:  
 Graph\* graph = nullptr;  
 std::string city = "";  
 List<Edge> adjacent;  
  
 double dijkstra\_weight;  
 Array<Vertex\*> dijkstra\_path;  
  
 Vertex();;  
 Vertex(Graph\* graph, std::string const& city);;  
 List<Edge>::Iterator find\_edge(Vertex& vertex);  
 Edge\* get\_edge(Vertex& vertex);  
 void connect(Vertex& vertex, double weight);  
 void disconnect(Vertex& vertex);  
 };  
  
  
 List<Vertex> vertices;  
  
 Graph();  
 void clear();  
  
 List<Vertex>::Iterator find\_vertex(std::string const& city);  
 Vertex\* get\_vertex(std::string const& city);  
 Vertex\* get\_or\_add\_vertex(std::string const& city);  
 bool fromFile(std::string filename);  
 void print();  
};  
  
  
#endif //SEM4LAB3\_GRAPH\_H

graph.cpp

#include "graph.h"  
  
Graph::Edge::Edge() {}  
  
Graph::Edge::Edge(Graph::Vertex &vertex, double weight) {  
 this->vertex = &vertex;  
 this->weight = weight;  
}  
  
Graph::Vertex::Vertex() {}  
  
Graph::Vertex::Vertex(Graph \*graph, std::string const &city) : graph(graph), city(city) {}  
  
List<Graph::Edge>::Iterator Graph::Vertex::find\_edge(Graph::Vertex &vertex) {  
 for (auto it = adjacent.begin(); it != adjacent.end(); it++) {  
 if ((\*it).vertex == &vertex) {  
 return it;  
 }  
 }  
 return adjacent.end();  
}  
  
Graph::Edge \*Graph::Vertex::get\_edge(Graph::Vertex &vertex) {  
 auto it = find\_edge(vertex);  
 return it != adjacent.end() ? &(\*it) : nullptr;  
}  
  
void Graph::Vertex::connect(Graph::Vertex &vertex, double weight) {  
 Edge\* edge = get\_edge(vertex);  
 if (edge != nullptr) {  
 edge->weight = weight;  
 } else {  
 adjacent.add(Edge(vertex, weight));  
 }  
}  
  
void Graph::Vertex::disconnect(Graph::Vertex &vertex) {  
 auto edge\_it = find\_edge(vertex);  
 if (edge\_it != adjacent.end()) {  
 adjacent.erase(edge\_it);  
 }  
}  
  
  
Graph::Graph() {}  
  
void Graph::clear() {  
 vertices.clear();  
}  
  
List<Graph::Vertex>::Iterator Graph::find\_vertex(std::string const &city) {  
 for (auto it = vertices.begin(); it != vertices.end(); it++) {  
 if ((\*it).city == city) {  
 return it;  
 }  
 }  
 return vertices.end();  
}  
  
Graph::Vertex \*Graph::get\_vertex(std::string const &city) {  
 auto it = find\_vertex(city);  
 return it != vertices.end() ? &(\*it) : nullptr;  
}  
  
Graph::Vertex \*Graph::get\_or\_add\_vertex(std::string const &city) {  
 Vertex\* vertex = get\_vertex(city);  
 if (vertex != nullptr) {  
 return vertex;  
 } else {  
 auto it = vertices.add(Vertex(this, city));  
 return &(\*it);  
 }  
}  
  
bool Graph::fromFile(std::string filename) {  
 std::ifstream stream(filename);  
 if (stream) {  
 clear();  
  
 std::string line;  
 while (std::getline(stream, line)) {  
 Array<std::string> tokens;  
  
 char\* token = strtok(line.data(), ";");  
 while (token != nullptr) {  
 tokens.add(std::string(token));  
 token = strtok(nullptr, ";");  
 }  
  
 if (tokens.length() == 4) {  
 Vertex\* vertex1 = get\_or\_add\_vertex(tokens[0]);  
 Vertex\* vertex2 = get\_or\_add\_vertex(tokens[1]);  
 if (tokens[2] != "N/A") {  
 vertex1->connect(\*vertex2, std::strtod(tokens[2].data(), nullptr));  
 }  
 if (tokens[3] != "N/A") {  
 vertex2->connect(\*vertex1, std::strtod(tokens[3].data(), nullptr));  
 }  
 } else if (!line.empty()) {  
 std::cout << "error in reading file, line skipped: " << line << "\n";  
 }  
 }  
 return true;  
 } else {  
 return false;  
 }  
}  
  
void Graph::print() {  
 for (auto i = vertices.begin(); i != vertices.end(); i++) {  
 Vertex& vertex = \*i;  
 std::cout << "flights from " << vertex.city << ": ";  
 for (auto j = vertex.adjacent.begin(); j != vertex.adjacent.end(); j++) {  
 std::cout << (\*j).vertex->city << " = " << (\*j).weight << ", ";  
 }  
 std::cout << "\n";  
 }  
}

dijkstra.h

#include "graph.h"  
  
#ifndef SEM4LAB3\_DIJKSTRA\_H  
#define SEM4LAB3\_DIJKSTRA\_H  
  
namespace DijkstraAlgorithm {  
 Array<Graph::Vertex\*> find\_path(Graph& graph, Graph::Vertex& start, Graph::Vertex& end);  
};  
  
#endif //SEM4LAB3\_DIJKSTRA\_H

dijkstra.cpp

#include "dijkstra.h"  
#include "graph.h"  
  
namespace DijkstraAlgorithm {  
 Array<Graph::Vertex\*> find\_path(Graph& graph, Graph::Vertex& start, Graph::Vertex& end) {  
 // prepare algorithm  
 for (auto it = graph.vertices.begin(); it != graph.vertices.end(); it++) {  
 (\*it).dijkstra\_weight = -1;  
 (\*it).dijkstra\_path.clear();  
 }  
  
 start.dijkstra\_path.add(&start);  
 start.dijkstra\_weight = 0;  
  
 List<Graph::Vertex\*> non\_infinite\_vertices;  
 non\_infinite\_vertices.add(&start);  
  
 while (non\_infinite\_vertices.get\_length() > 0) {  
 auto min\_vertex\_it = non\_infinite\_vertices.end();  
 for (auto it = non\_infinite\_vertices.begin(); it != non\_infinite\_vertices.end(); it++) {  
 if (min\_vertex\_it == non\_infinite\_vertices.end() || (\*it)->dijkstra\_weight < (\*min\_vertex\_it)->dijkstra\_weight) {  
 min\_vertex\_it = it;  
 }  
 }  
  
 if (min\_vertex\_it != non\_infinite\_vertices.end()) {  
 Graph::Vertex\* min\_vertex = \*min\_vertex\_it;  
 for (auto it = min\_vertex->adjacent.begin(); it != min\_vertex->adjacent.end(); it++) {  
 Graph::Edge& edge = \*it;  
 if (edge.vertex->dijkstra\_weight < 0 || edge.vertex->dijkstra\_weight > min\_vertex->dijkstra\_weight + edge.weight) {  
 edge.vertex->dijkstra\_weight = min\_vertex->dijkstra\_weight + edge.weight;  
 edge.vertex->dijkstra\_path = min\_vertex->dijkstra\_path;  
 edge.vertex->dijkstra\_path.add(edge.vertex);  
 non\_infinite\_vertices.add(edge.vertex);  
 }  
 }  
 non\_infinite\_vertices.erase(min\_vertex\_it);  
 }  
  
 }  
  
 return end.dijkstra\_path;  
 }  
};

array.h

#include <iostream>  
#include <stdlib.h>  
#include <functional>  
  
  
#ifndef SEM4LAB2\_ARRAY\_H  
#define SEM4LAB2\_ARRAY\_H  
  
template <typename T>  
class Array {  
 static const int REALLOCATION\_ELEMENTS = 16;  
  
 T\* memory\_span = nullptr;  
 int size = 0;  
 int allocated\_size = 0;  
  
 void ensure\_size(int size) {  
 if (allocated\_size < size) {  
 int old\_size = allocated\_size;  
 while (allocated\_size < size) {  
 allocated\_size += REALLOCATION\_ELEMENTS;  
 }  
  
 if (memory\_span != nullptr) {  
 T\* new\_memory\_span = new T[allocated\_size];  
 for (int i = 0; i < old\_size; i++) {  
 new\_memory\_span[i] = memory\_span[i];  
 }  
 delete[] (memory\_span);  
 memory\_span = new\_memory\_span;  
 } else {  
 memory\_span = new T[allocated\_size];  
 }  
 }  
 }  
  
public:  
 Array() {  
  
 }  
  
 Array(Array<T> const& other) {  
 ensure\_size(other.size);  
 size = other.size;  
 for (int i = 0; i < size; i++) {  
 memory\_span[i] = other.memory\_span[i];  
 }  
  
 std::vector<int> a = {1, 2};  
 }  
  
 Array(Array<T>&& other) {  
 memory\_span = other.memory\_span;  
 size = other.size;  
 allocated\_size = other.allocated\_size;  
 other.memory\_span = nullptr;  
 other.size = 0;  
 other.allocated\_size = 0;  
 }  
  
 Array<T>& operator= (Array<T> const& other) {  
 ensure\_size(other.size);  
 size = other.size;  
 for (int i = 0; i < size; i++) {  
 memory\_span[i] = other.memory\_span[i];  
 }  
 return \*this;  
 }  
  
 Array<T>& operator= (Array<T>&& other) {  
 memory\_span = other.memory\_span;  
 size = other.size;  
 allocated\_size = other.allocated\_size;  
 other.memory\_span = nullptr;  
 other.size = 0;  
 other.allocated\_size = 0;  
 return \*this;  
 }  
  
 inline int length() {  
 return size;  
 }  
  
 inline T& operator[] (int index) {  
 return memory\_span[index];  
 }  
  
 void add(T const& elem) {  
 ensure\_size(size + 1);  
 memory\_span[size++] = elem;  
 }  
  
 bool contains(T const& value) {  
 for (int i = 0; i < size; i++) {  
 if (memory\_span[i] == value) {  
 return true;  
 }  
 }  
 return false;  
 }  
  
 bool any(std::function<bool(T const&)> check) {  
 for (int i = 0; i < size; i++) {  
 if (check(memory\_span[i])) {  
 return true;  
 }  
 }  
 return false;  
 }  
  
 bool all(std::function<bool(T const&)> check) {  
 for (int i = 0; i < size; i++) {  
 if (!check(memory\_span[i])) {  
 return false;  
 }  
 }  
 return true;  
 }  
  
 void clear() {  
 if (memory\_span != nullptr) {  
 delete[] (memory\_span);  
 memory\_span = nullptr;  
 }  
 size = allocated\_size = 0;  
 }  
  
 ~Array() {  
 clear();  
 }  
};  
  
#endif

list.h

#include <string>  
  
  
#ifndef LAB1\_LIST\_H  
#define LAB1\_LIST\_H  
  
template <typename T>  
class List {  
public:  
 class Node {  
 public:  
 T value;  
 Node\* next = nullptr;  
 Node\* prev = nullptr;  
  
 Node() {};  
 Node(T const& v) : value(v) {}  
 };  
  
 class Iterator {  
 public:  
 Node\* node = nullptr;  
  
 Iterator() {}  
 Iterator(Node\* n) : node(n) {}  
  
 Iterator operator++(int) {  
 Node\* last = node;  
 node = node->next;  
 return Iterator(last);  
 }  
  
 Iterator operator--(int) {  
 Node\* last = node;  
 node = node->prev;  
 return Iterator(last);  
 }  
  
 T& operator\*() {  
 return node->value;  
 }  
  
 bool operator==(Iterator const& it) {  
 return it.node == node;  
 }  
  
 bool operator!=(Iterator const& it) {  
 return it.node != node;  
 }  
 };  
  
private:  
 Node\* first = nullptr;  
 Node\* last = nullptr;  
 Node \_end;  
 int length = 0;  
  
public:  
 Iterator begin() {  
 return first != nullptr ? Iterator(first) : end();  
 }  
  
 Iterator end() {  
 return Iterator(&\_end);  
 }  
  
 Iterator add(T const& value) {  
 Node\* node = new Node(value);  
 if (last != nullptr) {  
 last->next = node;  
 node->prev = last;  
 } else {  
 first = last = node;  
 }  
 \_end.prev = node;  
 node->next = &\_end;  
 last = node;  
 length++;  
 return Iterator(node);  
 }  
  
 void erase(Iterator iterator) {  
 Node\* node = iterator.node;  
 if (node->next == &\_end) {  
 last = \_end.prev = node->prev;  
 }  
 if (node->prev == nullptr) {  
 first = node->next;  
 } else {  
 node->prev->next = node->next;  
 }  
 node->next->prev = node->prev;  
 length--;  
 delete (node);  
 }  
  
 void print() {  
 std::cout << "[";  
 for (auto it = begin(); it != end(); it++) {  
 std::cout << \*it << " ";  
 }  
 std::cout << "]";  
 }  
  
 void clear() {  
 Node\* node = first;  
 while (node != nullptr && node != &\_end) {  
 Node\* next = node->next;  
 delete(node);  
 node = next;  
 }  
 \_end.next = \_end.prev = nullptr;  
 first = last = nullptr;  
 length = 0;  
 }  
  
 int get\_length() {  
 return length;  
 }  
  
 ~List() {  
 clear();  
 }  
};  
  
#endif //LAB1\_LIST\_H