**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»**

Тема: «Алгоритмы на графах»

Вариант 1

|  |  |  |
| --- | --- | --- |
| Студент гр. 8301 |  | Пискунович К.В. |
| Преподаватель |  | Тутуева А. В. |

Санкт-Петербург

2020

## Цель работы

Реализовать алгоритм Дейкстры на графе, представляемом при помощи списков смежности. А также найти наиболее эффективный по стоимости перелет из города ***i*** в город ***j***, используя список возможных авиарейсов, данный в текстовом формате в .txt файле.

## Описание программы

Идея алгоритма состоит в следующем:

Каждой вершине V графа G сопоставляется метка – минимальное известное расстояние от этой вершины до стартовой вершины S. Алгоритм работает пошагово — на каждом шаге он «посещает» одну вершину и пытается уменьшать метки. Работа алгоритма завершается, когда все вершины посещены.

Метка самой вершины S полагается равной 0, метки остальных вершин — INF или бесконечности. Все вершины графа помечаются как непосещённые.

Если все вершины посещены, алгоритм завершается. В противном случае, из ещё не посещённых вершин выбирается вершина V, имеющая минимальную метку. Рассматриваются всевозможные маршруты, в которых V является предпоследним пунктом. Для каждого соседа вершины V, кроме отмеченных как посещённые, рассматривается новая длина пути, равная сумме значений текущей метки V и длины ребра, соединяющего V с этим соседом. Если полученное значение длины меньше значения метки соседа, значение метки заменяется полученным значением длины. Рассмотрев всех соседей, вершина V помечается как посещённая и шаг алгоритма повторяется.

В данной программе используются следующие структуры данных:

1. **Класс Graph:**

Данный класс реализует граф, основанный на списках смежности. Этот класс хранит: список смежности adjList (динамический массив из контейнеров Map), который помимо номера смежной вершины хранит ещё и вес ребра; список List\_with\_name\_citys, являющийся связью между названием города и номером вершины; очередь с приоритетами List\_with\_marks, которая является связью между номером вершины, её меткой и информацией о том, посещена вершина или нет.

Конструктор данного класса получает в качестве параметра список названий городов и на его основе инициализирует остальные поля класса.

Данный класс имеет два публичных метода – create\_Graph(…) и dijkstra(…).create\_Graph строит граф (заполняет списки смежности) на основе информации из текстового файла; dijkstra(…) реализует сам алгоритм Дейкстры, возвращая наиболее дешевый маршрут в виде строки.

Данный класс имеет один приватный метод – buildPath(…), который строит самый дешевый путь из размеченного после обхода по Дейкстре графа.

1. **Класс List:**

Шаблонный двусвязный список. Список помимо обычных функций способен проверять наличие элемента в списке, получать индекс определенного элемента. Также были реализованы конструкторы перемещения и копирования. Используется для хранения введенной информации, списка названий городов, для хранения результирующего маршрута.

1. **Класс Map:**

Обычный шаблонный ассоциативный массив. Используется в качестве списка смежности для одновременного хранения смежной вершины и веса ребра, направленного к этой вершине.

1. **Класс priority\_queue:**

Шаблонная очередь с приоритетами на основе двоичной minHeap. Данная очередь дополнена возможностью получать приоритет элементов очереди, а также получать элемент с минимальным приоритетом, не удаляя его из очереди. Используется для хранения меток вершин. Приоритет элементов выступает в качестве метки, а наличие элемента в очереди – в качестве информации о том, была посещена вершина или нет.

## Оценка временной сложности методов

*N – количество строк входной информации*

*M – количество символов в строке*

*V – количество вершин (городов)*

*E – количество ребер (возможных перелетов между городами)*

1. ***convert*** имеет временную сложность О(*N\*M*)
2. ***get\_unique\_names*** имеет временную сложность О(*N\*V*);

*N\*(V\*1 + V\*1) = N\*V*

1. **Graph**имеет временную сложность О(*V*);
2. **create\_Graph**имеет временную сложность О(*N² + NV + N\*log(E)*);

*N\*(N + V + V + 2\*log(E)) = N² + NV + N\*log(E)*

1. ***buildPath*** имеет временную сложность О(*VE+E³*);

*log(E)+log(E)+V+2V+3V+V+E(E+E(E+log(E)+2V)+V)=   
= log(E)+V+E²+VE+ E³ + ElogE +EV = EV+E\*log(E)+ E³*

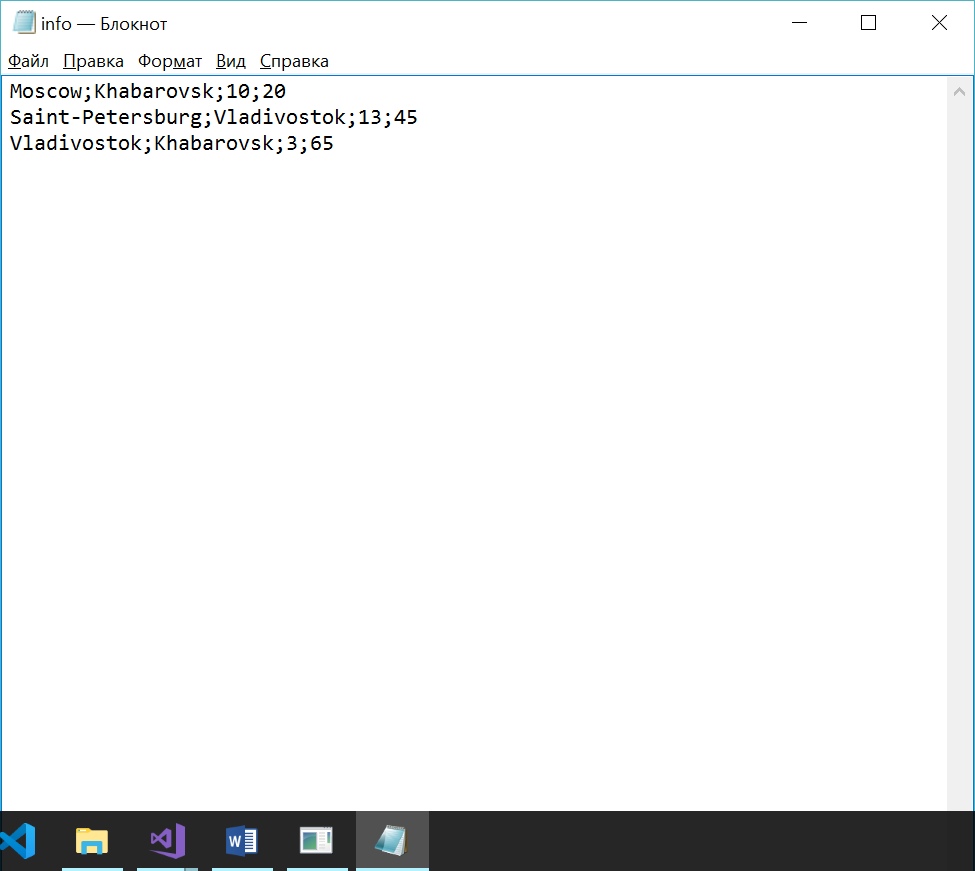
1. ***dijkstra*** имеет временную сложность О(*E³+V²\*E\*log(V)+V\*E\*log(E)+V\*E²+V²\*E*);

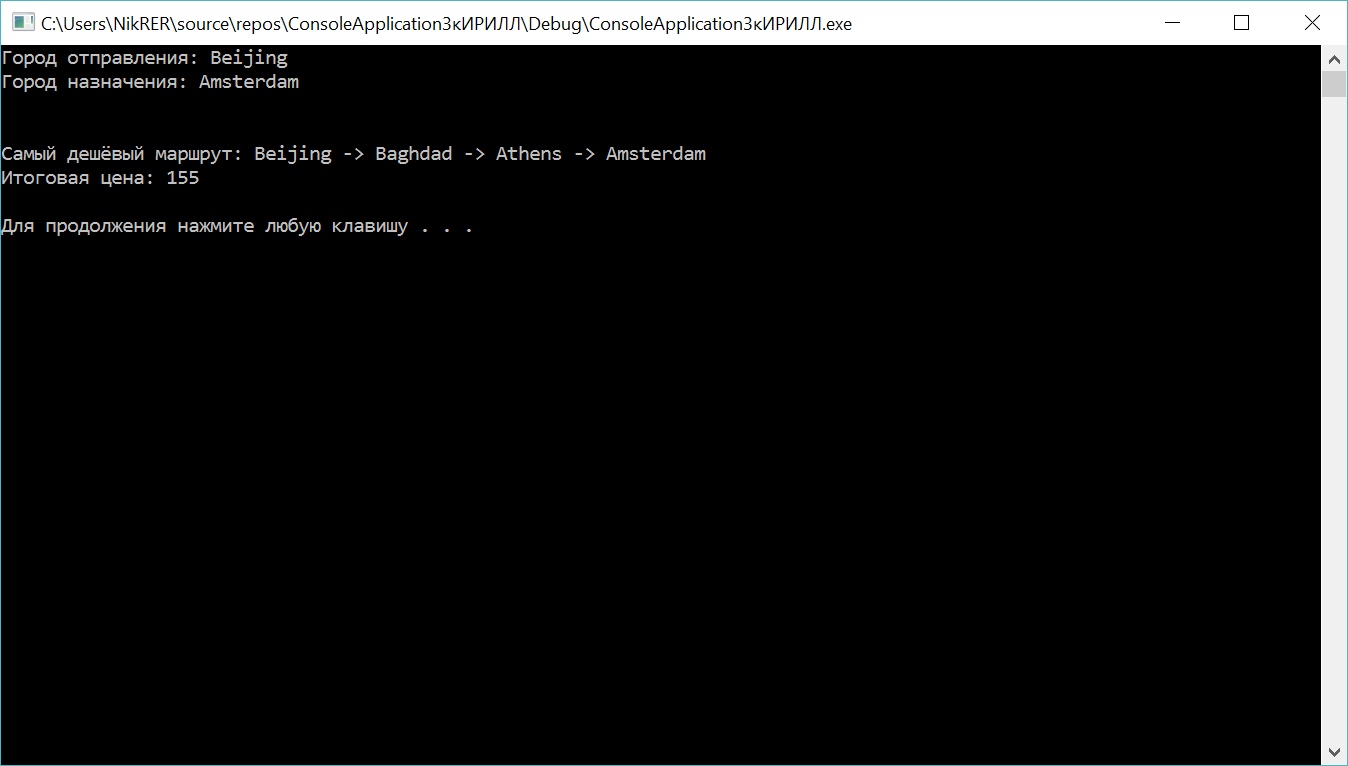
*2V+V\*log(V)+V(V+log(V)+E+E(E+V+log(E)+V+V\*log(V))+log(V))+VE+ E³+2V=*

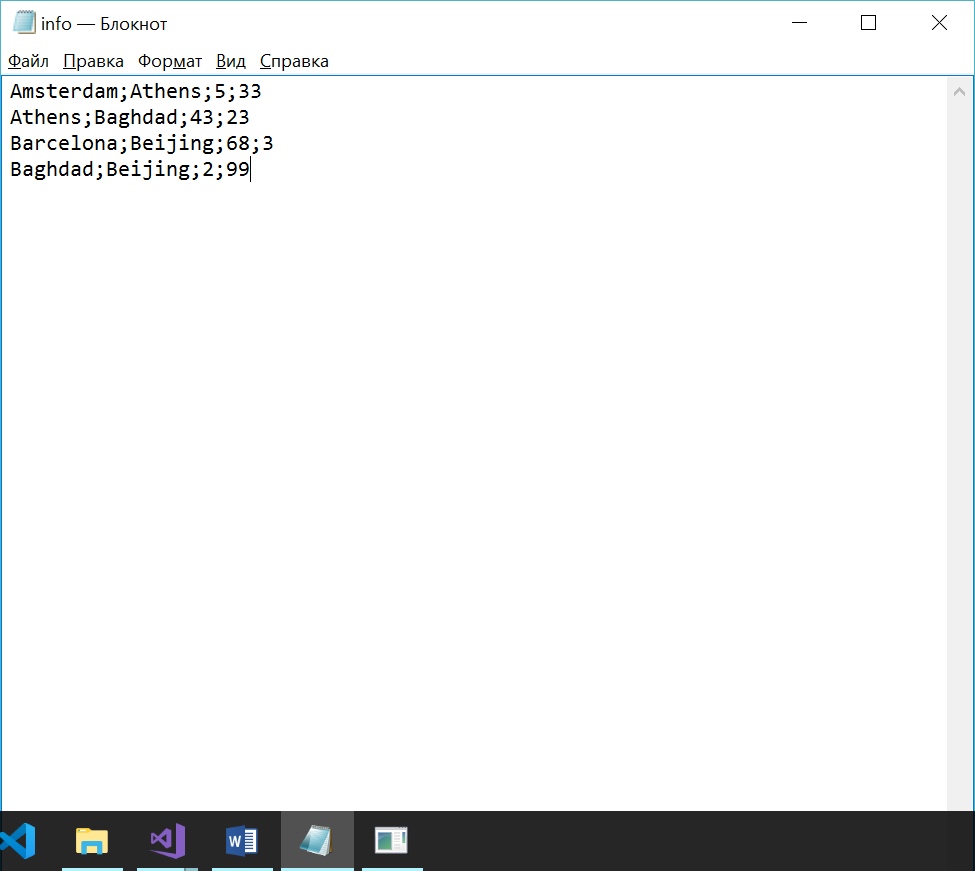
= *E³+V²\*E\*log(V)+V\*E\*log(E)+V\*E²+V²\*E*

## Примеры работы

## https://sun9-15.userapi.com/0da0EIf_-2Kr3sF5XfJNmjHEAHtI8_KCixfAVg/EjB8ruTyWtk.jpg







## Листинг

main.cpp

#include <iostream>

#include <fstream>

#include "Graph.h"

using namespace std;

int main()

{

setlocale(LC\_ALL, "Russian");

ifstream file("info.txt");

List<string> lines;

while (file)

{

std::string str;

getline(file, str);

lines.push\_back(str);

}

lines.pop\_back();

List<string\*> normalized\_info = convert(lines);

List<string> namesList = get\_unique\_names(normalized\_info);

Graph G(namesList);

G.create\_Graph(normalized\_info);

string start, end;

cout << "Город отправления: ";

cin >> start;

cout << "Город назначения: ";

cin >> end;

cout << endl << endl;

cout << G.dijkstra(start, end);

cout << endl << endl;

for (size\_t i = 0; i < normalized\_info.get\_size(); ++i)

{

delete[] normalized\_info.at(i);

}

system("pause");

return 0;

}

Graph.h

#pragma once

#include "List.h"

#include "Map.h"

#include "Priority\_queue.h"

#include <string>

#include <stdexcept>

constexpr uint8\_t City\_From = 0;

constexpr uint8\_t City\_To = 1;

constexpr uint8\_t Cost\_Forward = 2;

constexpr uint8\_t Cost\_Backward = 3;

constexpr uint8\_t Amount\_input\_words\_in\_one\_input\_string = 4;

template<typename T>

List<T> reverse(List<T> lst);

class Graph

{

private:

//Список смежности в котором хранятся номера смежной вершины с весом ребра

Map<uint16\_t, uint64\_t>\* adjList;

//Список - который показывает связь между названием города и вершины

List<string> List\_with\_name\_citys;

//Список хранит номера вершин и их метки

priority\_queue<uint16\_t> List\_with\_marks;

size\_t size;

//Строит самый дешёвый путь с помощью маркированного графа

List<string> buildPath(priority\_queue<uint16\_t>& visited\_Mark, const uint16\_t& start, const uint16\_t& end\_place)

{

List<string> path;

//Проверка,на то, что самый прямой путь самый лучший!

if (adjList[start].contains(end\_place))

{

if (adjList[start].find(end\_place) == visited\_Mark.get\_priority(end\_place))

{

path.push\_front(List\_with\_name\_citys.at(end\_place));

path.push\_front(List\_with\_name\_citys.at(start));

return path;

}

}

//Проверка на то возможно ли попасть в конечную точку

if (visited\_Mark.get\_priority(end\_place) == UINT64\_MAX)

{

path.push\_front("There is no route between " + List\_with\_name\_citys.at(start) + " and " + List\_with\_name\_citys.at(end\_place));

return path;

}

path.push\_front(List\_with\_name\_citys.at(end\_place));

uint16\_t current = end\_place;

auto cur\_mark = visited\_Mark.get\_priority(end\_place);

//Делаем путь от конечного города попасть до стартового

while (current != start)

{

string city;

//присваивание всех текущих соседей

auto neighbors\_list = adjList[current].get\_keys();

uint16\_t neighbor;

//проверяет метку у каждого соседа

for (size\_t i = 0; i < neighbors\_list.get\_size(); ++i)

{

neighbor = neighbors\_list.at(i);

if (cur\_mark - adjList[neighbor].find(current) == visited\_Mark.get\_priority(neighbor))

{

city = List\_with\_name\_citys.at(neighbor);

break;

}

}

path.push\_front(city);

current = neighbor;

cur\_mark = visited\_Mark.get\_priority(neighbor);

}

return path;

}

public:

explicit Graph(List<string>& List\_with\_name\_citys)

: size(List\_with\_name\_citys.get\_size())

{

this->List\_with\_name\_citys = List\_with\_name\_citys;

adjList = new Map<uint16\_t, uint64\_t>[size];

for (size\_t i = 0; i < size; i++)

List\_with\_marks.insert(i, UINT64\_MAX);

}

~Graph()

{

delete[] adjList;

}

//создание графа на основе текстового файла

void create\_Graph(List<string\*>& info) const

{

for (size\_t i = 0; i < info.get\_size(); i++)

{

const auto line = info.at(i);

uint16\_t city\_from\_number = List\_with\_name\_citys.find(line[City\_From]);

uint16\_t city\_to\_number = List\_with\_name\_citys.find(line[City\_To]);

uint64\_t cost\_forward = stoull(line[Cost\_Forward]);

uint64\_t cost\_backward = stoull(line[Cost\_Backward]);

//заполняем список смежности на основе входной информации

for (auto j = 1; j <= 2; j++)

{

adjList[city\_from\_number].insert(city\_to\_number, cost\_forward);

swap(city\_from\_number, city\_to\_number);

swap(cost\_forward, cost\_backward);

}

}

}

//Нахождение самого короткого пути по алгоритму Дейкстры

string dijkstra(const string& start\_point, const string& destination\_point)

{

if (start\_point == destination\_point)

return "Начальная и конечная точки одинаковые!!";

if (!List\_with\_name\_citys.contains(start\_point) || !List\_with\_name\_citys.contains(destination\_point))

return "Не существует начального или конечного города!";

//Посещённые вершины с их метками

priority\_queue<uint16\_t> visitedMarksList;

//начало пути

const uint16\_t start = List\_with\_name\_citys.find(start\_point);

//конец пути

const uint16\_t destination = List\_with\_name\_citys.find(destination\_point);

List\_with\_marks.correction(start, 0);

while (List\_with\_marks.getSize())

{

const auto current\_mark = List\_with\_marks.get\_priority();

const auto current = List\_with\_marks.extract\_min();

const auto neighbors\_list = adjList[current].get\_keys();

for (size\_t j = 0; j < neighbors\_list.get\_size(); j++)

{

const auto neighbor = neighbors\_list.at(j);

const auto visited = !List\_with\_marks.contains(neighbor);

const auto edge\_weight = adjList[current].find(neighbor);

if (!visited && edge\_weight != 0 && current\_mark != UINT64\_MAX)

{

const auto mark = current\_mark + edge\_weight;

if (List\_with\_marks.get\_priority(neighbor) > mark)

List\_with\_marks.correction(neighbor, mark);

}

}

visitedMarksList.insert(current, current\_mark);

}

const auto way = buildPath(visitedMarksList, start, destination);

string result;

if (way.get\_size() > 1)

{

result = "Самый дешёвый маршрут: ";

for (size\_t i = 0; i < way.get\_size() - 1; ++i)

result += way.at(i) + " -> ";

result += way.at(way.get\_size() - 1) + "\nИтоговая цена: " + to\_string(visitedMarksList.get\_priority(destination));

}

else

result = way.at(0);

return result;

}

};

//Преобразования входной строки

inline List<string\*> convert(List<string>& info)

{

if (info.get\_size() == 0)

throw length\_error("Входная строка пуста");

List<string\*> separated\_info;

for (size\_t i = 0; i < info.get\_size(); ++i)

{

const auto line = new string[Amount\_input\_words\_in\_one\_input\_string];

auto word\_number = 0;

for (auto ch : info.at(i))

{

if (ch == ';')

word\_number++;

else

line[word\_number] += ch;

}

if (word\_number > Amount\_input\_words\_in\_one\_input\_string)

throw length\_error("Неправильный формат ввода");

if (line[Cost\_Forward] == "N/A")

line[Cost\_Forward] = "0";

if (line[Cost\_Backward] == "N/A")

line[Cost\_Backward] = "0";

if (line[Cost\_Forward] == "0" && line[Cost\_Backward] == "0")

throw logic\_error("Эти города не связаны на прямую, вообще никак!");

separated\_info.push\_back(line);

}

return separated\_info;

}

//Возвращение списка со всеми именами

inline List<string> get\_unique\_names(List<string\*>& info)

{

List<string> List\_with\_name\_citys;

for (size\_t i = 0; i < info.get\_size(); ++i)

{

if (!List\_with\_name\_citys.contains(info.at(i)[City\_From]))

List\_with\_name\_citys.push\_back(info.at(i)[City\_From]);

if (!List\_with\_name\_citys.contains(info.at(i)[City\_To]))

List\_with\_name\_citys.push\_back(info.at(i)[City\_To]);

}

return List\_with\_name\_citys;

}

template<class T>

List<T> reverse(List<T> lst)

{

List<T> reversed;

for (size\_t i = 1; i <= lst.getSize(); ++i)

reversed.push\_back(lst.at(lst.getSize() - i));

return reversed;

}