**Федеральное агентство связи**

**Ордена Трудового Красного Знамени**

**Федеральное государственное бюджетное образовательное учреждение**

**Высшего образования**

**«Московский технический университет связи и информатики»**

Кафедра Математической кибернетики и информационных технологий

**Отчет по лабораторной работе №6**

По дисциплине «СиАОД»

На тему:

«Сетевые алгоритмы. Динамические алгоритмы поиска путей»

Выполнил: студент группы БВТ1801

Задоркин Максим Александрович

Руководитель:

Кутейников Иван Алексеевич

Москва 2020

**Цель работы:**

Изучить основные сетевые алгоритмы, алгоритмы поиска путей. Разработать программу нахождения наикратчайшего пути между вершинами графа. Разработать графический интерфейс пользователя.

**Задача:**  
Реализовать алгоритм поиска кратчайшего расстояния между двумя вершинами ориентированного взвешенного графа в соответствии с вариантом. Предусмотреть задание графа в виде матрицы смежности/инцидентности, читаемой из файла, либо графически с помощью пользовательского интерфейса. Разработать графический интерфейс пользователя с визуализацией графа и отображением кратчайшего расстояния между задаваемыми пользователем вершинами.

По результатам работы проанализировать временную сложность работы заданного алгоритма в зависимости от числа узлов и ребер графа.

Алгоритм: Алгоритм Дейкстры.

**Программа:**

Программа реализована при помощи языка javascript, domAPI и технологии svg. Так как использованы последние стандарты ecmascript рекомендуется запускать приложение в последних версиях браузера.

Все файлы кода можно просмотреть и скачать с моего github:

<https://github.com/MaximZar/SaDPA/tree/master/searchForWays>

Опробовать приложение можно прямо по следующей ссылке (для быстрого старта есть шаблон):

<https://maximzar.github.io/SaDPA/searchForWays/src/>

**Код программы**

Код содержит несколько абстракций:

**Dijkstra.js**

1. Dijkstra – реализация алгоритма Дейкстры

2. buildSuccessGraph – отрисовка кратчайшего пути на графе

**ImagingGraph.js**

1. readMatrix – считывание графа по таблице

2. buildGraph – отрисовка графа

3. generateCoord – генерация координат узлов

4. imaging – кнопка визуализации (считывание + отрисовка)

**matrix.js**

1. createEmptyLine – вспомогательная функция для createMatrix

2. createMatrix – создание матрицы на основе введеных вершин

**quickStart.js** – содержит функцию для построения стартового шаблона.

**svgImaging.js** - вспомогательный файл с функциями по создание элементов svg dom дерева.

**main.js** – главный файл, содержащий все события, а так же:

1. deleteMatrix – удалить все данные

Код алгоритма (остальной код лежит на github):

**const** Dijkstra = () **=>** {

**const** timeStart = **new** Date().getMilliseconds();

**const** tops = document.querySelector('#tops').value.trim().split(' ');

**const** matrix = readMatrix();

**const** start = document.querySelector('#startPoint').value;

**const** end = document.querySelector('#endPoint').value;

**const** P = tops.map(el **=>** start); *// кто откуда пришел, изначально все из стартовой точки*

**const** S = [[start, 0]]; *// те вершины, для которых уже найдена минимальный путь*

**let** iter = matrix.filter(el **=>** el.name === start)

.map(el **=>** {

**const** haveDirWithWeight = el.dirs.map(val **=>** [val.nameTo, val.weight]);

**const** haveDir = haveDirWithWeight.map(val **=>** val[0]);

**const** anotherWays = tops.filter(val **=>** !haveDir.includes(val) && start !== val)

.map(val **=>** [val, **Infinity**]);

**return** haveDirWithWeight.concat(anotherWays);

}).flat(); *// первая итерация*

**let** minimalOnIter = iter.reduce((min, value) **=>** min[1] > value[1] ? value : min, [**null**, **Infinity**]);

iter = iter.filter(val **=>** val !== minimalOnIter);

S.push(minimalOnIter);

**while** (!S.map(el **=>** el[0]).includes(end)) {

**const** lastMinimal = matrix.filter(el **=>** el.name === S[S.length - 1][0])[0];

iter = iter.map(top **=>** {

**const** roadFromLastToTop = lastMinimal.dirs.filter(val **=>** val.nameTo === top[0])[0];

**let** newRoad;

**if** (roadFromLastToTop) newRoad = +S[S.length - 1][1] + +roadFromLastToTop.weight;

**else** newRoad = top[1];

**const** D = Math.min(top[1], newRoad);

**if** (D < top[1]) {

**const** index = tops.reduce((acc, val, index) **=>** val === top[0] ? index : acc, **Infinity**);

P[index] = minimalOnIter[0];

}

**return** [top[0], D];

});

minimalOnIter = iter.reduce((min, value) **=>** min[1] > value[1] ? value : min, [**null**, **Infinity**]);

iter = iter.filter(val **=>** val !== minimalOnIter);

S.push(minimalOnIter);

}

*// находим путь по road*

**const** road = [end];

**while** (road[0] !== start) {

**const** now = road[0];

**const** index = tops.reduce((acc, val, index) **=>** val === now ? index : acc, **Infinity**);

**const** from = P[index];

road.unshift(from);

}

buildSuccessGraph(road);

**const** timeEnd = **new** Date().getMilliseconds();

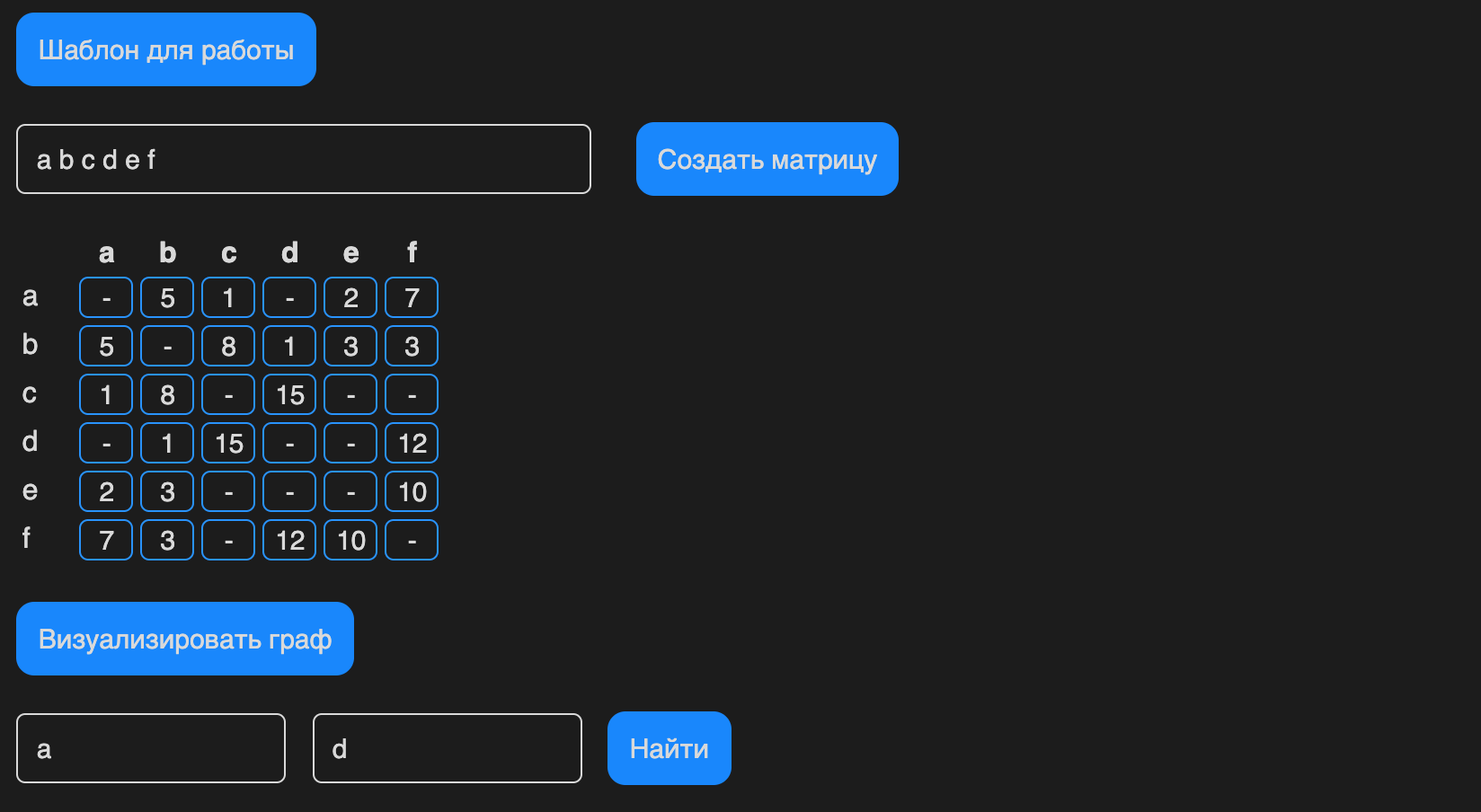
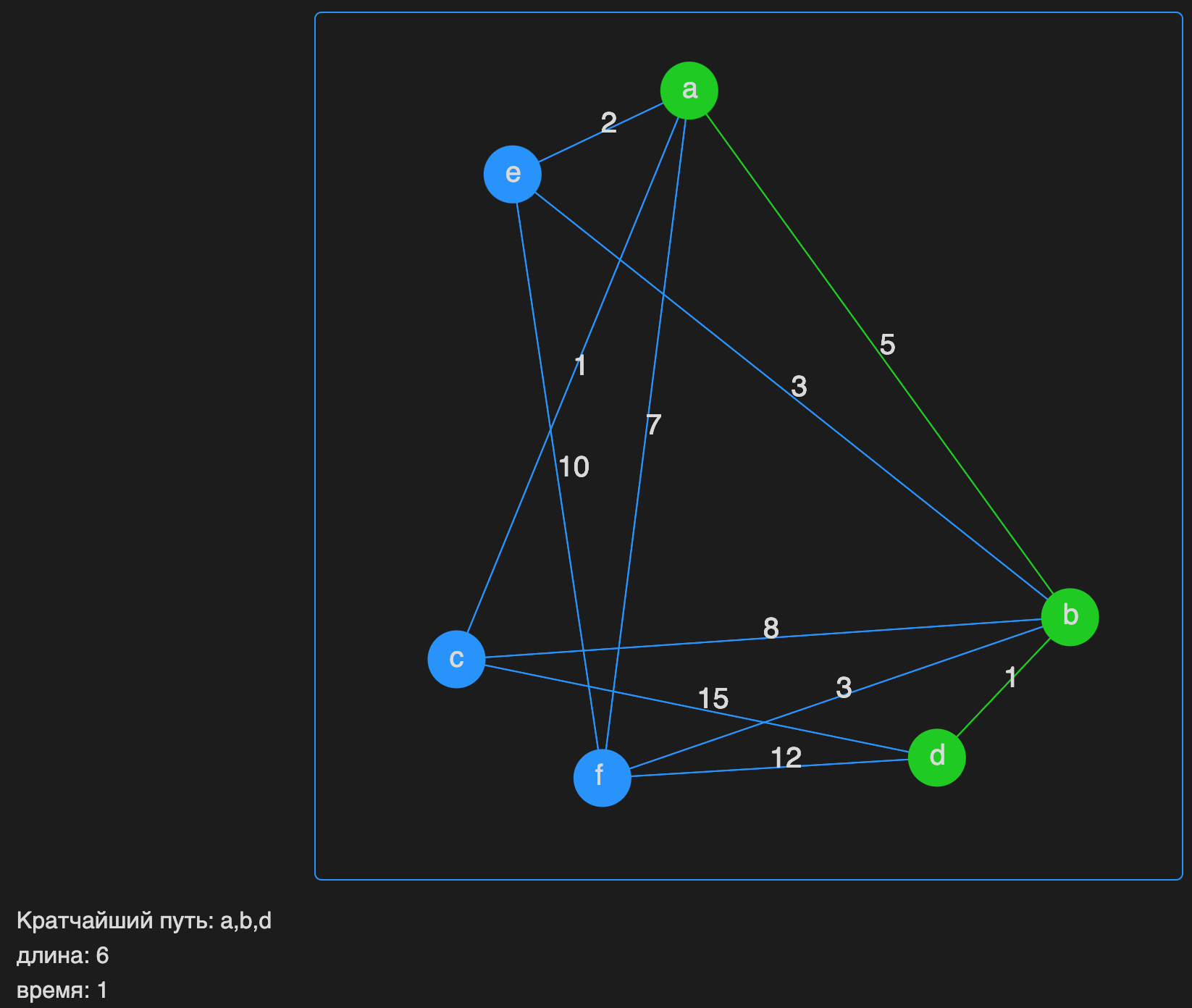
document.querySelector('#output').style.display = 'block';

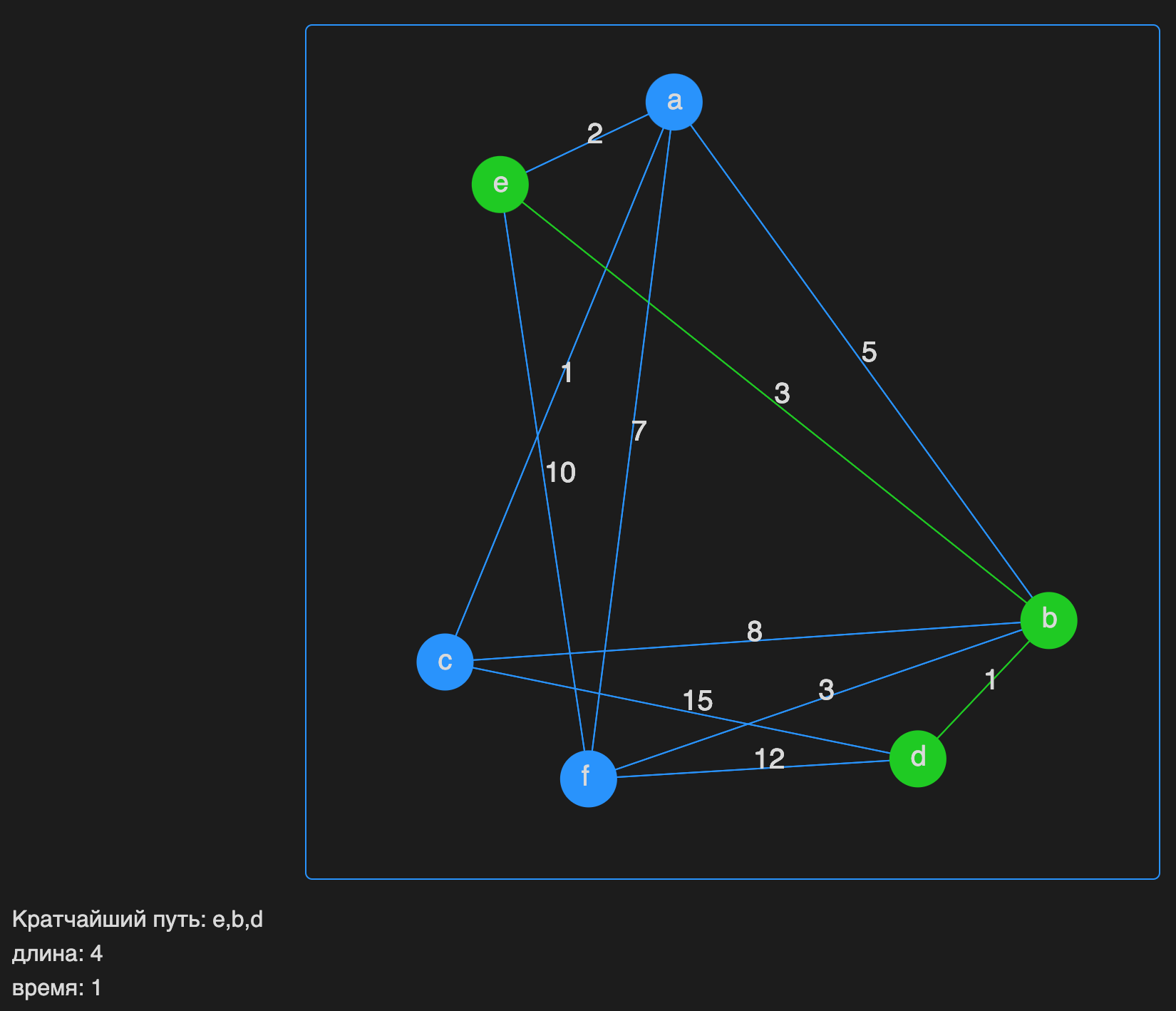
document.querySelector('#output').innerHTML = `Кратчайший путь: ${road} </br>

длина: ${minimalOnIter[1]}</br>время: ${timeEnd - timeStart}`;

};

**Тестирование** **программы и скриншоты работы:**



**Вывод:** в данной лабораторной работе я научился реализовывать алгоритмы поиска кратчайшего пути, разработал графический интерфейс пользователя. Проанализировав результаты программы, можно сделать вывод, что время достаточно маленькое для такого объема вершин и ребер.