Министерство науки и высшего образования Российской Федерации Федеральное государственное бюджетное образовательное учреждение высшего образования

«УЛЬЯНОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

**Лабораторная работа №3**

по дисциплине: «Методы оптимизации».

Вариант - 8

Выполнил:

студент 1 курса, гр. ИВТАСмд-11

Кондратьев Павел Сергеевич.

Проверил:

преподаватель кафедры ВТ

Валюх Вероника Валерьевна

г. Ульяновск, 2020

Оглавление

[Задание: 3](#_Toc54888093)

[Ход выполнения работы: 4](#_Toc54888094)

[Описание алгоритм Флойда-Уоршелла нахождения кратчайших путей между всеми парами вершин 4](#_Toc54888095)

[Описание алгоритма 5](#_Toc54888096)

[Восстановление самих путей 6](#_Toc54888097)

[Недостатки 6](#_Toc54888098)

[Реализация и сравнение эффективности программной реализации алгоритма 8](#_Toc54888099)

[Выводы: 14](#_Toc54888100)

[Онлайн реализации алгоритма Флойда -Уоршелла 16](#_Toc54888101)

[Исследование реализации выбранного алгоритма в различных пакетах прикладных программ (MathCad; MATLAB). 18](#_Toc54888102)

[Список источников: 20](#_Toc54888103)

# Задание:

1. Составление библиотеки ссылок на исходники программ и онлайн реализации метода. Исследование реализации выбранного алгоритма в различных пакетах прикладных программ (Maple; MathCad; Mathematica; MATLAB)

2. Осмысление и описание метода оптимизации.

3. Реализация и сравнение эффективности реализации алгоритма в разных языках программирования (С++, С#, JAVA, язык по выбору). Исследование зависимости скорости выполнения от а) размерности задачи, б) языка.

4. Оформить отчёт о проделанной работе.

|  |  |  |  |
| --- | --- | --- | --- |
| **№** | **Лабораторная 1**  **Методы поиска для функции одной переменной (одномерная оптимизация)**  **Методы последовательного поиска** | **Лабораторная 2**  **Методы оптимизации дифференцируемых функций**  **(Градиентные методы, прямые методы, методы первого порядка) \*** | **Лабораторная 3**  **Оптимизация на графах**  **(Поиски кратчайших путей, построение деревьев)** |
| 8 | метода Ньютона | Модифицированный метод наискорейшего спуска | Алгоритм Флойда |

**Лабораторная №3**

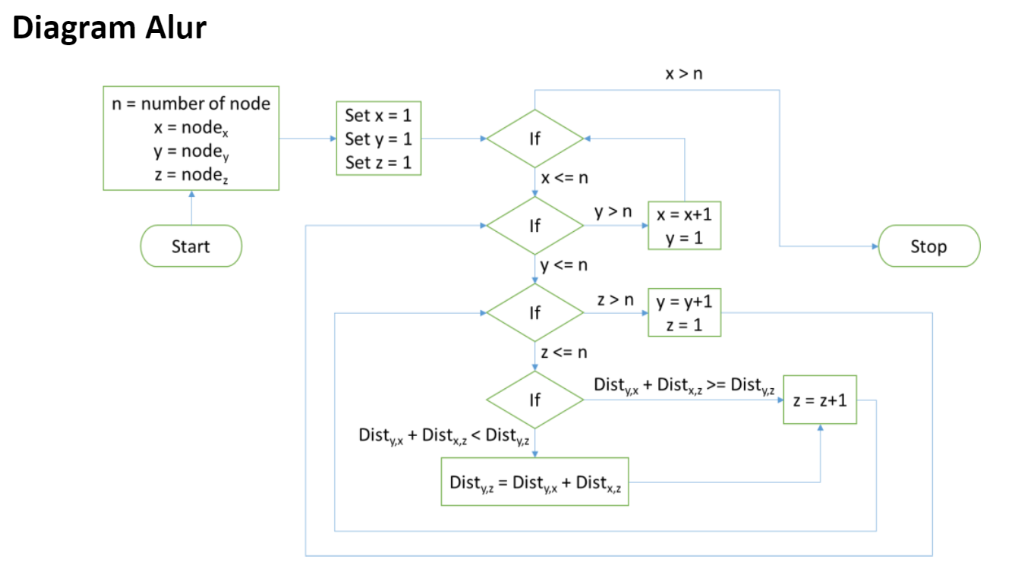
# Ход выполнения работы:

## Описание алгоритм Флойда-Уоршелла нахождения кратчайших путей между всеми парами вершин

Использование алгоритма Флойда – Уоршелла позволяет определить самые короткие пути между любой парой вершин графа. Согласно условию задачи, имеем неориентированный или ориентированный взвешенный граф G, который имеет n вершин.

Предполагается, что граф не содержит циклов отрицательного веса (тогда ответа между некоторыми парами вершин может просто не существовать — он будет бесконечно маленьким).

Необходимо определить величину всех кратчайших путей d(i, j) между вершинами i и j, при условии отсутствия в графе циклов с отрицательными весами (в этом случае решения для некоторых пар вершин могут быть бесконечно малыми, то есть несуществующими). Данный алгоритм описали в своих работах независимо друг от друга и примерно в одно и тоже время Роберт Флойд и Стивен Уоршелл в 1962-м году. В их честь и назвали алгоритм, именуемый и сегодня алгоритмом Флойда – Уоршелла. Однако следует заметить, что ещё в 1959-ом году Бернардом Роем был разработан почти аналогичный алогитм, но он не был тогда никем замечен.



## Описание алгоритма

Ключевая идея алгоритма — разбиение процесса поиска кратчайших путей на фазы.

Перед k-ой фазой k = 1…n считается, что в матрице расстояний d[][] сохранены длины таких кратчайших путей, которые содержат в качестве внутренних вершин только вершины из множества { 1, 2, …, k-1 \ (вершины графа мы нумеруем, начиная с единицы).

Иными словами, перед k-ой фазой величина d[i][j] равна длине кратчайшего пути из вершины i в вершину j, если этому пути разрешается заходить только в вершины с номерами, меньшими k (начало и конец пути не считаются).

Легко убедиться, что чтобы это свойство выполнилось для первой фазы, достаточно в матрицу расстояний d[][] записать матрицу смежности графа: d[i][j] = g[i][j] — стоимости ребра из вершины i в вершину j. При этом, если между какими-то вершинами ребра нет, то записать следует величину "бесконечность". Из вершины в саму себя всегда следует записывать величину 0, это критично для алгоритма.

Пусть теперь мы находимся на k-ой фазе, и хотим пересчитать матрицу d[][] таким образом, чтобы она соответствовала требованиям уже для k+1-ой фазы. Зафиксируем какие-то вершины i и j. У нас возникает два принципиально разных случая:

* Кратчайший путь из вершины i в вершину j, которому разрешено дополнительно проходить через вершины { 1, 2,…, k }, совпадает с кратчайшим путём, которому разрешено проходить через вершины множества { 1, 2,…, k-1 }.
* "Новый" кратчайший путь стал лучше "старого" пути.

Это означает, что "новый" кратчайший путь проходит через вершину k. Сразу отметим, что мы не потеряем общности, рассматривая далее только простые пути (т.е. пути, не проходящие по какой-то вершине дважды).

Тогда заметим, что если мы разобьём этот "новый" путь вершиной k на две половинки (одна идущая i => k, а другая — k \=> j), то каждая из этих половинок уже не заходит в вершину k. Но тогда получается, что длина каждой из этих половинок была посчитана ещё на k-1-ой фазе или ещё раньше, и нам достаточно взять просто сумму d[i][k] + d[k][j], она и даст длину "нового" кратчайшего пути.

Таким образом, вся работа, которую требуется произвести на k-ой фазе — это перебрать все пары вершин и пересчитать длину кратчайшего пути между ними. В результате после выполнения n-ой фазы в матрице расстояний d[i][j] будет записана длина кратчайшего пути между i и j, либо "бесконечность", если пути между этими вершинами не существует.

Последнее замечание, которое следует сделать, — то, что можно не создавать отдельную матрицу rm new\_d[][] для временной матрицы кратчайших путей на k-ой фазе: все изменения можно делать сразу в матрице d[][]. В самом деле, если мы улучшили (уменьшили) какое-то значение в матрице расстояний, мы не могли ухудшить тем самым длину кратчайшего пути для каких-то других пар вершин, обработанных позднее.

**Асимптотика алгоритма**, очевидно, составляет O (n^3).

## Восстановление самих путей

Легко поддерживать дополнительную информацию — так называемых "предков", по которым можно будет восстанавливать сам кратчайший путь между любыми двумя заданными вершинами в виде последовательности вершин.

Для этого достаточно кроме матрицы расстояний d[][] поддерживать также матрицу предков p[][], которая для каждой пары вершин будет содержать номер фазы, на которой было получено кратчайшее расстояние между ними. Понятно, что этот номер фазы является не чем иным, как "средней" вершиной искомого кратчайшего пути, и теперь нам просто надо найти кратчайший путь между вершинами i и p[i][j], а также между p[i][j] и j. Отсюда получается простой рекурсивный алгоритм восстановления кратчайшего пути.

## Недостатки

Алгоритм Флойда - Уоршелла является эффективным для расчёта всех кратчайших путей в плотных графах, когда имеет место большое количество пар рёбер между парами вершин. В случае разреженных графов с рёбрами неотрицательного веса лучшим выбором считается использование алгоритма Дейкстры для каждого возможного узла. При таком выборе сложность составляет O(|V| \* |E|log |V|) при применении двоичной кучи, что лучше, чем O(|V|^3) алгоритма Флойда - Уоршелла тогда, когда |E| существенно меньше |V|^2 (условие разреженности графа). Если граф разрежен, у него имеются рёбра с отрицательным весом и отсутствуют циклы с отрицательным суммарным весом, то используется алгоритм Джонсона, который имеет ту же сложность, что и вариант с алгоритмом Дейкстры.

Как и любой базовый алгоритм, алгоритм Флойда используется очень широко и много где, начиная от поиска транзитивного замыкания графа, заканчивая генетикой и управлением проектами. Но первое что приходит в голову, конечно же, транспортные и всякие другие сети.

Скажем, если вы возьмете карту города - её транспортная система - это граф, соответственно присвоив каждому ребру некую стоимость, рассчитанную скажем из пропускной способности и других важный параметров - вы сможете подвести попутчика по самому короткому быстрому дешевому пути.

# Реализация и сравнение эффективности программной реализации алгоритма

Составим программу для нахождения кратчайших расстояний между всеми вершинами взвешенного ориентированного графа.

Вначале граф считывается следующим образом: сначала считывается кол-во вершин и кол-во ребер, а затем ребра в следующем формате: <начальная вершина> <конечная вершина> <расстояние между ними>.

После производится инициализация матриц путей и расстояний и реализовывается ясам алгоритм.

Ниже приведены примеры исходного кода нахождения кратчайших расстояний между всеми вершинами взвешенного ориентированного графа.

Реализация на C++

#include <cstdio>

#include <iostream>

#include <chrono>

using namespace std;

const int N = 100; // max количество вершин в графе

const int INF = 1000000; // бесконечное расстояние, указывает на то, что между вершинами нет пути

int n; // кол-во вершин в графе

int adj[N][N] = { { 0 } }; // матрица смежности

int path[N][N]; // матрица путей

int dist[N][N]; // матрица расстояний

void read\_graph()

{

// граф считывается следующим образом: сначала считывается кол-во вершин и кол-во ребер,

// а затем ребра в следующем формате: <начальная вершина> <конечная вершина> <расстояние между ними>

// замечание: данная функция считывает орграф

int m;

printf("Введите кол-во <вершин> и кол-во <ребер>: ");

scanf("%d%d", &n, &m);

printf("\nВведите ребра в следующем формате: <начальная вершина> <конечная вершина> <расстояние между ними>\n");

for (int u, v, w; m--; adj[u][v] = w)

scanf("%d%d%d", &u, &v, &w);

}

void floid()

{

// инициализация матриц путей и расстояний

for (int u = 0; u < n; ++u)

for (int v = 0; v < n; ++v)

if (adj[u][v] || u == v)

{

path[u][v] = v;

dist[u][v] = adj[u][v];

}

else

{

path[u][v] = N;

dist[u][v] = INF;

}

// сам алгоритм флойда

for (int k = 0; k < n; ++k)

for (int u = 0; u < n; ++u)

if (dist[u][k] != INF)

for (int v = 0; v < n; ++v)

if (dist[u][v] > dist[u][k] + dist[k][v])

{

dist[u][v] = dist[u][k] + dist[k][v];

path[u][v] = path[u][k];

}

}

void show\_path(int u, int v)

{

// вывод пути из вершины u в v и расстояния между ними

if (path[u][v] == N)

{

printf("Нет пути\n");

return;

}

int x = u;

printf("%d", x);

while (x != v)

printf(" %d", x = path[x][v]);

printf(" (%d)\n", dist[u][v]);

}

int main()

{

setlocale(LC\_ALL, "Russian");

int u, v;

read\_graph();

auto begin = chrono::steady\_clock::now();

for (int k = 0; k < 100000; ++k)

floid();

auto end = chrono::steady\_clock::now();

chrono::duration<double> elapsed\_seconds = end - begin;

cout << "The time: " << elapsed\_seconds.count() << " s\n";

while (true) // обрабатываем запросы вывода путей и расстояний между вершинами пока не ввели -1 для одно из двух вершин

{

printf("Введите исходную и конечную вершину: ");

scanf("%d%d", &u, &v);

if (u == -1 || v == -1)

break;

show\_path(u, v);

}

return 0;

}

Реализация на Python

import time

import timeit

code\_to\_test = """

inf=1000

m2=[

[0,1,5,7,9,inf],

[1,0,6,7,3,inf],

[5,6,0,9,inf,8],

[7,7,9,0,8,3],

[9,3,inf,8,0,inf],

[inf,inf,8,3,inf,0]

]

n2=len(m2)

d=[]

p=[]

for x in m2:

temp=[]

temp2=[]

for y in x:

temp.append(y)

temp2.append(-1)

d.append(temp)

p.append(temp2)

for x in range(n2):

for y in range(n2):

if x==y:

p[x][y]=0

elif d[x][y]!=inf:

p[x][y]=x

else:

p[x][y]=-1

for x in range(n2):

for y in range(n2):

for z in range(n2):

if d[y][x]+d[x][z]<d[y][z]:

d[y][z]=d[y][x]+d[x][z]

p[y][z]=p[x][z]

"""

def floyd(m2,n2):

d=[]

p=[]

inf=1000

for x in m2:

temp=[]

temp2=[]

for y in x:

temp.append(y)

temp2.append(-1)

d.append(temp)

p.append(temp2)

for x in range(n2):

for y in range(n2):

if x==y:

p[x][y]=0

elif d[x][y]!=inf:

p[x][y]=x

else:

p[x][y]=-1

for x in range(n2):

for y in range(n2):

for z in range(n2):

if d[y][x]+d[x][z]<d[y][z]:

d[y][z]=d[y][x]+d[x][z]

p[y][z]=p[x][z]

return d,p

def printAll(p2,n2):

print("")

for x in range(n2):

for y in range(n2):

if x!=y and p2[x][y]!=-1:

print("Path dari",x+1,"ke",y+1,"=",x+1,"->",end='')

printPath(p2,x,y)

print("",y+1)

def printPath(p3,x2,y2):

if p3[x2][y2]!=x2:

printPath(p3,x2,p3[x2][y2])

print("",p3[x2][y2]+1,"->",end='')

inf=1000

m=[

[0,inf,3,inf,16,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,20,11,inf,inf,inf],

[inf,0,inf,inf,12,5,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf],

[3,inf,0,3,1,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf],

[inf,inf,inf,0,inf,inf,inf,inf,inf,inf,inf,inf,21,inf,inf,inf,inf,inf,inf,inf],

[16,inf,inf,inf,0,7,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf],

[inf,inf,inf,inf,inf,0,inf,inf,inf,inf,8,inf,inf,inf,inf,inf,inf,inf,inf,inf],

[inf,inf,inf,inf,inf,inf,0,inf,inf,inf,inf,inf,inf,1,inf,inf,inf,inf,inf,inf],

[inf,inf,inf,inf,3,inf,inf,0,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf],

[inf,9,inf,inf,inf,inf,inf,inf,0,inf,inf,inf,inf,inf,inf,inf,inf,inf,16,inf],

[inf,inf,inf,inf,inf,inf,inf,inf,inf,0,inf,inf,11,inf,inf,19,inf,inf,inf,inf],

[inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,0,inf,inf,inf,inf,inf,inf,inf,5,inf],

[inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,0,inf,inf,inf,4,inf,inf,inf,inf],

[inf,inf,inf,inf,inf,inf,16,inf,13,inf,inf,inf,0,inf,inf,inf,inf,inf,inf,inf],

[inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,2,0,inf,inf,inf,inf,inf,17],

[inf,inf,inf,inf,inf,inf,3,inf,5,inf,inf,inf,inf,inf,0,inf,inf,inf,inf,inf],

[20,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,0,inf,inf,inf,inf],

[11,inf,inf,inf,inf,inf,inf,11,inf,inf,inf,inf,inf,inf,inf,inf,0,inf,inf,inf],

[inf,2,inf,9,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,0,inf,inf],

[inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,0,inf,inf],

[inf,inf,inf,inf,inf,inf,inf,inf,inf,8,inf,14,inf,inf,inf,inf,inf,inf,inf,0]

]

n=len(m)

#print("\nTabel jarak mula-mula:")

#for x in m:

# print(x)

#start\_timer = time.time()

#for i in range(0, 100000):

# result,path=floyd(m,n)

#print(time.time() - start\_timer)

#print("\nTabel jarak terpendek:")

#for x in result:

# print(x)

#printAll(path,n)

elapsed\_time = timeit.timeit(code\_to\_test, number=100000)

print(elapsed\_time)

Реализация на JavaScript

let time = performance.now(); // стартовая засечка

var firstTop = +$("#firstTop").val();

var secondTop = +$("#secondTop").val();

var result = $.extend(true, [], weights);

var solutionPathes = $.extend(true, [], pathes);

for (var k = 0; k < result.length; k++) {

for (var i = 0; i < result.length; i++) {

for (var j = 0; j < result[i].length; j++) {

if (result[i][j] > result[i][k] + result[k][j]) {

result[i][j] = result[i][k] + result[k][j];

solutionPathes[i][j] = k + 1;

}

if (i == j && result[i][j] < 0) {

alert("Обнаружен контур отрицательной длины! Дальнейшие вычисления невозможны.\n" +

"Вершина, через которую проходит контур отрицательной длины: " + (k + 1));

return false;

}

}

}

}

time = performance.now() - time;// замер продолжительности

console.log(`${time.toFixed(5)}`);

Исследуем зависимость скорости выполнения программного кода от

1. заданной погрешности вычисления
2. языка программирования

|  |  |  |  |
| --- | --- | --- | --- |
| Кол-во вершин и ребер | Время выполнения программного кода, с | | |
| Python | C++ | JavaScript |
| 6, 9 | 0.203592 | 0.0688829 | 0,272050 |
| 7, 10 | 0.518592 | 0.141804 | 0.920504 |
| 10, 16 | 1.88307 | 0.349185 | 2,115880 |
| 20, 30 | 7.62205 | 1.093765 | 9,049262 |

Анализ охватывает следующие средства:

1. **Язык С++ - режим Debug, библиотека chrono**

На всех UNIX-подобных ОС, очень старая функция clock( ) возвращает процессорное время процесса в тиках, а макрос CLOCKS\_PER\_SEC количество тиков в секунду.

В стандарте С++11 появился набор классов для работы со временем, он позволяет больше чем старый модуль time.h и расположен в пространстве имен std::chrono. В нем определены классы для задания моментов времени (std::chrono::timepoint), интервалов времени (std::chrono::duration) и несколько видов часов:

* std::chrono::system\_clock предназначен для получения данных от системных часов реального времени, моменты времени system\_clock::time\_point могут быть преобразованы в тип time\_t с округлением до секунд при помощи метода system\_clock::to\_time\_t(). Получить текущее время можно с помощью метода system\_clock::now(). Недостатком системных часов является их нестабильность — системные часы могут быть подведены пользователем или процессом автоматической синхронизации с внешним сервером;
* std::chrono::steady\_clock предоставляет возможность работы со стабильными системными часами. Эти часы возвращают не системное время (на них не влияет напрямую изменение системного времени). Эти часы оптимально подходят для замера времени выполнения фрагмента кода. Текущее время можно получить при помощи steady\_clock::now();
* Класс std::chrono::high\_resolution\_clock позволяет работать с системными часами максимально высокого разрешения (как и для обычные системные часы, он не является стабильным, т.е. допускает подведение времени). При помощи метода high\_resolution\_clock::now() можно получить самое высокое разрешение из всех видов часов, входящих в библиотеку.

Использование библиотеки chrono — лучший способ если нужно фактическое замерить время выполнения программы (в отличии функции time(), модуля time.h она позволяет получить время в миллисекундах и даже наносекундах). Однако если программа работает на многопроцессорной системе и часто ожидает какие-либо данные (не только пользовательский ввод, но и данных от других потоков/процессов) - то больший интерес может представлять реальное время выполнения, возвращаемое функцией clock() модуля time.h. Реальное время лучше отражает потребляемый ресурс процессора, т.к. если текущий процесс простаивает на кластере (где выполняются сотни других приложений), то он вытесняется операционной системой и практически не загружает процессор.

1. **Python, библиотека TIMEIT**

Модуль Python timeit — это простой интерфейс для быстрого измерения времени выполнения небольших блоков кода. Когда вы создаете приложение, вы можете задаться вопросом, как этот блок кода будет работать, и захотите протестировать его в различных сценариях.

Функция модуля timeit.timeit (stmt, setup, timer, number) принимает четыре аргумента:

* stmt — это утверждение, которое вы хотите измерить; по умолчанию это «пройти».
* setup, который является кодом, который вы запускаете перед запуском stmt ; по умолчанию это «пройти».
* Обычно мы используем это для импорта необходимых модулей для нашего кода.
* таймер, который является объектом timeit.Timer ; обычно оно имеет разумное значение по умолчанию, поэтому вам не нужно об этом беспокоиться.
* число, которое является числом выполнений, которые вы хотели бы запустить stmt .

Примечание: Обратите внимание на то, что выход является время выполнения итерации число раз фрагмента кода, а не одну итерацию. Для одной итерации exec. время, разделите время выхода на число.

**Плюсы timeit**

* timeit() автоматически будет использовать time.clock() или time.time() для вас в фоновом режиме, в зависимости от того, какая операционная система вам нужна для получения наиболее точных результатов.
* timeit() отключает сборщик мусора, который может исказить результаты.
* timeit() повторяет тест много раз (в нашем случае 100 раз), чтобы минимизировать влияние других задач, выполняемых в вашей операционной системе.

1. **JavaScript, console.time() и performance.now()**

В High Resolution Time API есть функция now(), возвращающая объект DOMHighResTimeStamp. Это число с плавающей точкой, отражающее текущее время в миллисекундах, с точностью до тысячной миллисекунды.

## Выводы:

По результатам экспериментов можно сделать вывод, что скорость работы программ на языке C++ выше, чем скорость работы программы, реализующий метод Флойда на языке Python. Это связано с тем, что С++ компилируется непосредственно в машинный код и работает с минимально возможным количеством хелперов и прослоек.

С JavaScript могут происходить разные события такие как:

1. **Случайное измерение ненужных вещей**

Скорость отправки данных и вывода в консоль сильно зависит от браузера и даже от того, что ещё он делает в это время. Вероятно, вы считаете, что это console.log работает непредсказуемо медленно. Но в любом случае будет ошибкой выполнять более одной функции, даже если каждая из функций не подразумевает никаких операций ввода-вывода.

1. **Однократное измерение**

Риск данного подхода заключается в том, что браузерный JavaScript-движок может выполнять субоптимизацию, т. е. во второй раз функция будет вызвана с теми же входными данными, которые будут запомнены и использованы в дальнейшем. Чтобы это обойти, можно использовать много разных входных строк вместо того, чтобы раз за разом брать одно и то же значение. Однако при разных входных данных и скорость выполнения функции раз от раза может отличаться.

1. **Излишнее доверие к средним значениям**

Скорее всего, причина как раз в проведении субоптимизации и в необходимости «прогрева» компилятора. Мало что можно сделать, чтобы этого избежать, но зато можно обезопасить себе от неверных заключений.

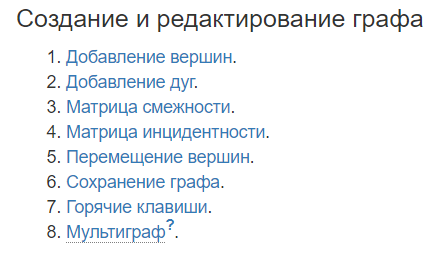
Пытаясь продемонстрировать точность измерения производительности в JavaScript с помощью performance.now(), обнаружилось, что наша интуиция может подвести нас: эмпирические данные совершенно не совпали с нашими предположениями.

Рассмотренные ошибки - не единственные возможные. К ним можно добавить, например, измерение нереалистичных сценариев или измерение только на одном JS-движке.

# Онлайн реализации алгоритма Флойда -Уоршелла

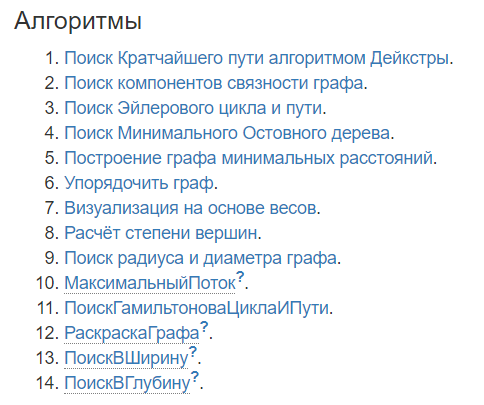
Сайт <https://graphonline.ru/> представляет собой большую утилиту по созданию и редактированию графа онлайн.

Пример действий над графом:



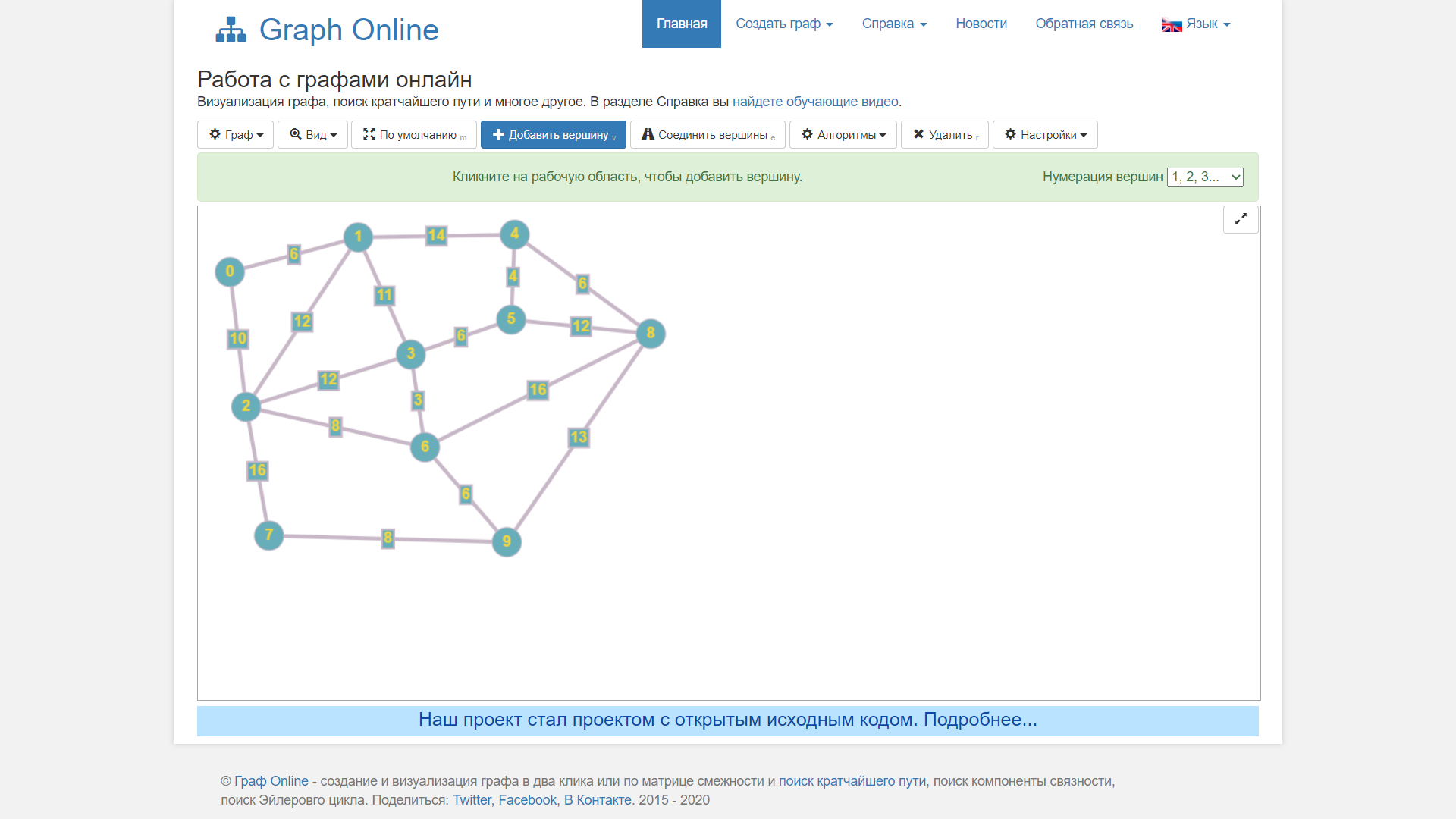
Также на данном сайте присутствуют большинство известных алгоритмов на графах.

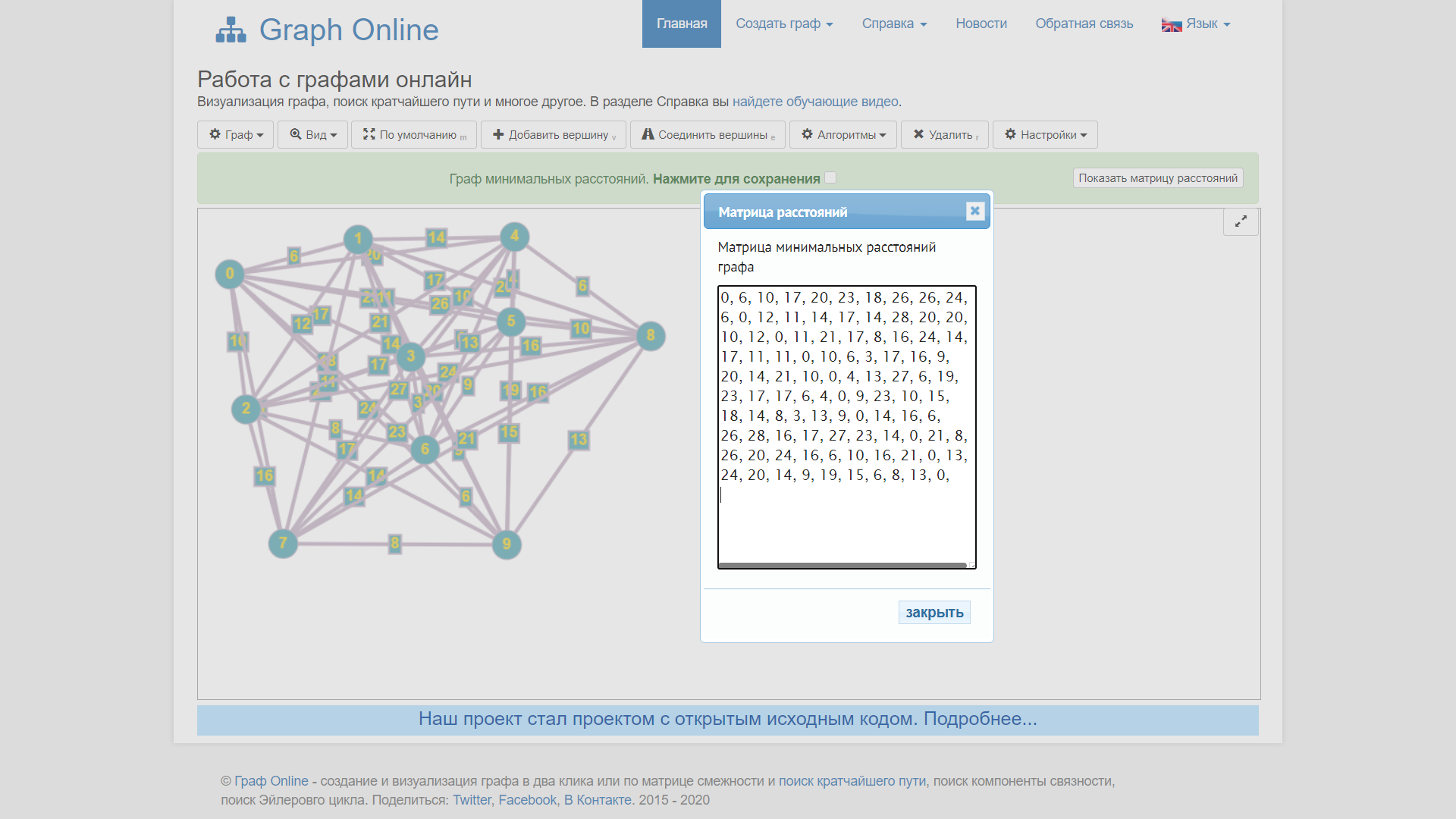
Пример алгоритмом:



Сам сайт представляет собой интерактивную площадку по созданию графа. На данной площадке одно как самому создать граф (с помощью таблицы, текста или уже имеющегося на сайте), так и загрузить его с компьютера.

**Пример работы:**





# Исследование реализации выбранного алгоритма в различных пакетах прикладных программ (MathCad; MATLAB).

Средства пакета MathLab предоставляют векторизованный алгоритм Флойда-Уорсхолла для поиска кратчайших путей и расстояний между каждой парой узлов.

**Общие сведения:**

* Матрица входного веса (или начального расстояния) должна иметь значения Inf, где узлы не соединены и 0 на диагонали.
* Выходы-это матрица расстояний коротких путей D и матрица предшественника P, такая что P(i,j) - это узел перед j на кратчайшем пути от i до j, Поэтому, если вы хотите построить пути, вы должны прочитать P назад.

**Исходный код библиотеки:**

function [D,P] = FloydWarshall(D)

prevD = D;

P = zeros(size(D));

for k = 1:length(D)

D = min(D,D(:,k) + D(k,:));

P(D<prevD) = k;

prevD = D;

end;

**Пример программы:**

% FastFloyd-быстро вычислить матрицу кратчайшего пути для всех пар

% Использует векторизованную версию алгоритма Flyod-Warshall

% ИСПОЛЬЗОВАНИЕ:

%

% D = Фастфлойд (A)

%

% D-расстояние (геодезические, все пары кратчайшего пути и т. д.) матрица.

% A-матрица смежности, где A(i, j) - затраты на перемещение из вершины i в

% вершина j. Если вершина i и вершина j не соединены, то A (i, j) должно быть

% be >= диаметр сети (Inf работает нормально).

%

% ОБРАЗЕЦ:

%

% Здесь я создаю случайную двоичную матрицу и преобразую ее в целочисленный формат. Затем

% Я беру обратную сторону матрицы так, чтобы все несмежные пары получили значение

% от инф. Результат хранится в файле D.

%

% A = int32(rand(100,100) );

% D = Фастфлойд(1./Ля)

%

function D = FastFloyd(D)

%input matrix must be initialized properly -- A(i,j) = Inf if i and j are not neighbors.

D(D==0) = inf;

n = length(D);

for k=1:n

i2k = repmat(D(:,k), 1, n);

k2j = repmat(D(k,:), n, 1);

D = min(D, i2k+k2j);

end

end

# Список источников:

1. Аттетков А.В., Галкин С.В., Зарубин B.C. МЕТОДЫ ОПТИМИЗАЦИИ Учеб. для вузов /Под ред. B.C. Зарубина, А. П. Крищенко. - 2-е изд., стереотип. - М.: МГТУ им. Н. Э. Баумана, 2003.
2. Маркина М.B., Судакова А.В. ПРАКТИКУМ ПО РЕШЕНИЮ ЗАДАЧ ОПТИМИЗАЦИИ В ПАКЕТЕ MATLAB: учебно-методическое пособие. –Нижний Новгород: Нижегородский госуниверситет, 2017.
3. Прокопенко Н. Ю. Методы оптимизации: учеб. пособие /Н. Ю. Прокопенко; Нижегор. гос. архитектур. - строит. ун-т. – Н. Новгород: ННГАСУ, 2018.
4. Алгоритм Флойда-Уоршелла нахождения кратчайших путей между всеми парами вершин <https://e-maxx.ru/algo/floyd_warshall_algorithm> (Дата обращения 29.10.20)