*Муниципальное общеобразовательное бюджетное учреждение*

*«Средняя общеобразовательная школа №3»*

Научно-практическая конференция учащихся

«Гагаринские чтения»

Исследование графов и выбор оптимального алгоритма для нахождения кратчайшего пути.

Выполнили ученики 9б класса

Шенягин Даниил и Субботин Андрей

Руководитель – учитель информатики

Краев Николай Владимирович

Муром – 2018

Содержание

[**Актуальность исследования:** 3](#_Toc506487161)

[**Цель работы:** 3](#_Toc506487162)

[**Задачи:** 3](#_Toc506487163)

[**Введение** 4](#_Toc506487164)

[**История изобретения графов** 5](#_Toc506487165)

[**Алгоритмы для графов** 6](#_Toc506487166)

[История разработки web-приложения 7](#_Toc506487167)

[Основная часть нашей работы 8](#_Toc506487168)

[Практическое применение нашей работы 11](#_Toc506487169)

[Вывод 12](#_Toc506487170)

## **Актуальность исследования:**

В настоящий момент многие системы представляют собой сеть, которую удобно предствить в виде абстракционной модели - графа. Поэтому легче все работать с этой моделью и использовать алгоритмы на ней.

И необходимо знать и уметь перебирать элементы этой модели, а так же находить кратчайший путь до любого элемента.

Эта тема все больше становится популярной в современной математике.

## **Цель работы:**

Выяснить какой алгоритм для нахождения крачайшего пути в графе более оптимальный.

Критерии оптимальности:

1. Лучшее время работы алгоритма
2. Универсальность алгоритма
3. Простота работы

## **Задачи:**

* Изучить графы и их виды.
* Рассмотреть алгоритмы.
  + Для нахождения кратчайшего путь в графе.
  + Для нахождения компоненты связанности.
* Разработать приложение
  + Генерация графа
  + Визуализации графа
  + Применение алгоритмов на графе
  + Вывод практического времени работы алгоритмов
* Сравнить и проанализировать
  + время работы алгоритмов
  + универсальность работы алгоритмов
  + простоту работы алгоритмов
* Выбрать оптимальный алгоритм на основе полученных данных

# **Введение**

Когда мы готовились к ОГЭ по информатике, столкнулись с задачами для решения которых, используются графы. И у нас возникли вопросы: что такое графы, где они используются, и где их можно применять. Нам стало интересно более подробно изучить данный вопрос.

Как оказалось, эта тема очень объемная и мы решил изучить более углубленно вопрос о нахождении крачайшего пути из точки А в точку Б, а так же затронуть нахождение компоненты связанности в графе.

# **История изобретения графов**

Графы – абстракная модель, способ предствления взаимодействия объектов в системе. В математике существует целый раздел – теория графов, который изучает графы, их свойства и применение.

Графом в математике называется конечная совокупность точек, именуемых вершинами; некоторые из них соединены друг с другом линиями, называемых ребрами графа.

При взгляде на географическую карту сразу бросается в глаза сеть железных и автомобильных дорог. Это типичный граф: кружочки обозначают станции или города - вершины графа, а соединяющие их пути - ребра.

Если вам, к примеру, нужно найти крачайший путь из точки А в точку Б, может возникнуть проблема: чем больше промежуточных точек, тем больше вариантов маршрутов будет получаться и одному человеку просчитать крачайший путь будет очень сложно и долго.

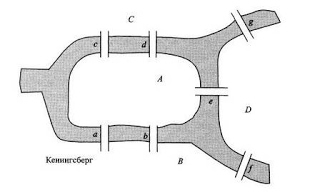
Родоначальником теории графов принято считать математика Леонарда Эйлера (1707-1783).

Рисунок 1

*Задача о Кенигсбергских мостах.* На рис. 1 представлен схематический план центральной части города Кенигсберг (ныне Калининград), включающий два берега реки Перголя, два острова в ней и семь соединяющих мостов. Задача состоит в том, чтобы обойти все четыре части суши, пройдя по каждому мосту один раз, и вернуться в исходную точку. Эта задача была решена (показано, что решение не существует) Эйлером в 1736 году.

# **Алгоритмы для графов**

В этой работе мы решили затронуть алгоритмы нахождения кратчайшего пути и компоненты связанности в графе. При поиске мы нашли 7 алгоритмов связанных с графами:

1. Алгоритм обхода в глубину
2. Алгоритм обхода в ширину
3. Алгоритм Дейкстры
4. Алгоритм Флойда — Уоршелла
5. Алгоритм Джонсона
6. Алгоритм Ли(волновой алгоритм)
7. Алгоритм Беллмана — Форда

Два первых алгоритма находят компоненту связанности в графе.

Компонента связности графа (или просто компонента графа ) — максимальный (по включению) связный подграф графа . Другими словами, это подграф , порождённый множеством вершин, в котором для любой пары вершин в графесуществует -цепь и для любой пары вершин , не существует -цепи.

Мы не стали брать алгоритм Ли или волновой алгоритм, т.к. он предназначен для нах. кр. пути в лабиринте. Алгоритм Джонсона был сложен в реализации, и мы не стали брать его в свой кругозор исследования и еще этому поспособствовало нехватка времени

Итак, мы взяли для исследования 3 самых популярных алгоритмов для нахождения кратчайшего пути в графе:  
  
1. Алгоритм Дейкстры  
2. Алгоритм Флойда - Уоршелла  
3. Алгоритм Форда – Беллмана

Важно учесть, что побочно с ними будут работь алгоритмы обхода графа

1. Алгоритм обхода в глубину
2. Алгоритм обхода в ширину

# История разработки web-приложения

Приложение было написано на платформе React JS с применение таких технологий как HTML5, CSS3, JavaScript ([ECMAScript 6](https://www.google.ru/search?newwindow=1&rlz=1C1KPLB_enRU680RU680&q=ECMAScript+6&spell=1&sa=X&ved=0ahUKEwiy3oq68qDZAhWD3iwKHfhLDvUQkeECCCUoAA)). Приложение писалось около полутора месяцев(по показателю github).

В двух словах: визуализация представлена на HTML5 -> tag <svg></svg>, ход дизайнерской идеи был реализован на CSS3 (Less или Sass), контроллером служила сеть компонентов, связанные между собой, работающая на React JS (Twitter).

Сборка всего приложения занимался Gulp. [Gulp.js](http://gulpjs.com/) это потоковый сборщик проектов на JS. Он использует Stream и действительно является очень быстрым. Более подробно, он занимается компилированием и транслированием такого языка программирования как CoffeeScript, трансируемый в JS. И для удобства мы использовали препроцессор Less и Sass, транслируемый в CSS.

Приложение визуально состоит из двух частей: само поля для построения графа и консоль-панель для контроллинга, которая состоит из:

1. Изменение характеристик графа (цвет, радиус вершин)
2. Поле модов
   1. Удаление вершин
   2. Обозначение вершин
3. Вывод матрицы данного графа
4. Последняя панель
   1. История действий
   2. Генерация графа (последовательно)
   3. Вывод результата работы алгоритма на графе

По написанию приложения стоит сказать, что мы столкнулись со следующими проблемами или трудностями:

1. Реализация алгоритмов на языке JS (Coffee Script)
2. Нам пришлось впитать много теории для разработки приложения
3. Требовалось много времени для построения приложения.

# Основная часть нашей работы

Исследование заключается в генерации графов и испытание на них алгоритмов. Далее получение результатов, сравнение их, распределение и сортировка. Затем выявление оптимально алгоритма, который показывает наилучший результат по 3 характеристикам.

Этап 1. Генерация графа  
Для любого графа принимается 2 аргумента (кол-во вершин и кол-во ребер), притом, что этот граф будет не ориентированным и с положительным весом ребер.  
Мы создавали 13 графов, где начальное кол-во вершин – 10, а конечное – 70, с каждым разом кол-во вершин увеличивалось на 5. Почему именно мы остановились до 70, а не к примеру до 100. В данном примере мы использовали небольшое количество вершин и максимальное количество ребер для данного графа. А для большого количества ребер требуется большое место для памяти и более производительный процессор. Компьютер, на котором производилось исследование не справлялся с количеством вершин более 70, а может и справлялся, но уже подводил браузер. Алгоритм для нахождения максимального количества ребер мы не нашли, и ,понаблюдав за увеличением вершин и ребер, мы составили собственный простейший алгоритм. Это задали формулой

* a0=3,
* i=3,
* an+1=an + i,
* i++(i с каждым разом увеличивается на единицу)

Этап 2. Испытание графа на алгоритмы

1. Выбор вершины
2. Выбор алгоритма
3. Запуск алгоритма

Данные действия будут выполняться для каждой вершины(кол-во = N). При этом, параллельно этим действиям, будет считаться практическое время алгоритма для одной вершины и среднее время для всех вершин графа. Кол-во тестингов = кол-ву вершин(это рассмотрено в следующих этапах).

Этап 3. - Расчет теоретического времени  
Для каждого алгоритма существует теоретическое время работы, называемое временной сложностью алгоритма.

В [информатике](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0" \o "Информатика) временна́я сложность [алгоритма](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC" \o "Алгоритм) определяет время работы, используемое алгоритмом, как функции от длины [строки](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D0%BE%D0%BA%D0%BE%D0%B2%D1%8B%D0%B9_%D1%82%D0%B8%D0%BF" \o "Строковый тип), представляющей входные данные. Другими словами, временная сложность алгоритма показыват нам выходные данные от входных. Говоря проще, это функция показывающая, как возрастает график от входных данных. В данном случае за входные данные принимается кол-во вершин и кол-во ребер. У каждого алгоритма свая временная сложность, это объясняется тем, что алгоритм может иметь свою архетиктуру. Например, в коде 3 вложенных цикла будет иметь сложность O(N3). Эта функция будет возращать какое-то число, и это число в теории будет количеством тактов процессора. И поэтому, узнав тактовую частоту процессора компьютора, можно легко высчитать теоретическое время работы любого алгоритма. Например, тактовая частота процессора на компьютере, на котором проводилось исследование, равно 3,8 ГГц. Чтобы найти теоретическое время алгоритма, нужно поделить количество тактов на тактовую частоту процессора и получим результат в секундах, но чтобу получить результат в милисекундах, нужно умножить данное число на 1000.

Этап 4. Нахождение практического времени

Мы задавались вопросом: как найти практическое время алгоритма. Далее нашли пост на хабре и это оказалось довольно-таки просто. Чтобы найти это время нужно задать 2 переменных, одна из которых будет хранить время в милисекундах до работы алгоритма, вторая будет хранить значение времени, прошедшее после работы алгоритма. Затем просто нужно вычесть время первой переменной из второй. Среднее время работы алгоритма расчитывается по формуле среднего арифметического.

Это выглядит примерно так (пример из кода на CoffeeScript):

time = performance.now() #нач. время

do someAlgorithm

\_time = performance.now() - time # \_time – будет хранить время работы алгоритма

Этап 5. Получение результатов и запись их в таблицу

Из 2 этапа видно, что для каждого графа и для каждого алгоритма существует свое время работы, отличающееся от других.  
Дальше мы записывали данные в таблицу.

Этап 6. - Сравнение показателей  
Получив важную для нас информацию, мы стали сравнивать показатели практического времени. Так как теоретическое время растет последовательно и равномерно, сравнивать тут нечего(посмотрев на график, можно увидеть интересующую нас информацию). Чего нельзя сказать насчет практического времени, т.к. помотрев на график, можно увидеть «скачки времени». Графики практического времени практически одинаковы по возрастанию функции, но можно заметить, что результаты отличаются где-то от 100 раз до 10000 раз. Проведя исследования, мы так и не поняли от чего такие большие отличия результатов, но выдвинули гепотизу, что такие различия могли образоваться от того, что:

1. Мы взяли не самый производительный по времени работы язык программирования - JS.
2. Мы использовали Web-приложение, которое должно исполнятся брузером, из-за которого всеровно теряется производитетьность.
3. Т.к. мы спешили со сроками сдачи, мы мало уделили времени оптимизации нашего приложения и самих алгоритмов.
4. Из вышесказанного про временную сложность алгоритмов: алгоритм исполнялся процессором (какое-то опр. Кол-во тактов), а процессор мог быть занят другими процессами, которые могли бы оказать некоторое влияние на производительность.

# Практическое применение нашей работы

Посмотрев на карту метро, не вооруженным глазом можно увидеть и понять, что эта карта будет являться самым обыкновенным графом, где вершины графа будут являтся станциями, а ребра путями. Но как оказалось пути расчитаны на карте не по расстоянию, а по времени. Поэтому нам стало интересно испытать нашу работу на практическом применении. Суть проста – выбираем любую станцию из которой нужно добраться и выбираем другую станцию, в которую нужно добраться.

1 этап. Генерация графа

Так как приблизительно срисовывать с карты граф нельзя, потому что погрешность путей будет довольно большой, нужно было как-то эту проблему решать. И в помощь нам пришли три волшебных строчки кода, которые могли встроить картинку карты метро на задний план поля строения графа.

Потом по картинке можно построить граф соответсвующий карте метро.

2 этап. Использование алгоритмов.

Как можно увидеть из таблицы, алгоритм Флойда умеет искать путь из одной вершины в другую. Что дает ему приемущество над другими алгоритмами. Еще путь дает большую информацию человеку, которому нужно добраться из точки А в точку Б.

Скрины и таблицы приложены к практической части.

# Вывод

Как окозалось алгоритм Флойда-Уоршела самый быстрый и дает нам точный путь из любой вершины в любую вершину, но он очень трудный в реализации. Алгоритм Дейкстры очень простой в реализации, но он очень долгий для небольшого количества вершин и максимального раличества ребер. Поэтому самый оптимальный алгоритм это алгоритм Беллмана-Форда он средний по времени выполнения и прост в реализации. Из всего вышесказанного, хочется заметить, что у каждого алгоритма своя архетиктура, своя сложность, своя индивидуальность, и конечно свое время работы.

Проект получился весьма успешным и имеет колоссальный потенциал к развитию. Наша работа включает далеко не все алгоритмы, как для нахождения кратчайшего пути, так и для нахождения компоненты связанности.