**ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ**

**Федеральное государственное автономное образовательное учреждение высшего образования**

**Национальный исследовательский университет**
**«Высшая школа экономики»**

**Факультет компьютерных наук**
**Образовательная программа**
**«Прикладная математика и информатика»**

# КУРСОВАЯ РАБОТА

На тему: Различные способы построения графовых структур данных для поиска ближайшего соседа

Тема на английском: Several Ways to Form Graph Based Nearest Neighbour Search Structure

Студент / студентка 2-го курса
группы № БПМИ2110/21ПМИ-2:
Рябков Игорь Дмитриевич
(Ф.И.О.)
Научный руководитель:
Пономаренко Александр Александрович
(Ф.И.О.)
Доцент, НН Кафедра прикладной математики и информатики
(должность, звание)

# Содержание

# 1 Введение

В последнее десятилетие наша жизнь стала тесно связана с удобными приложениями и сервисами. Многие из них базируются на рекомендательных системах, для предоставления целевого товара на основе наших интересов. Некоторые используют компьютерное зрение для решения огромного кол-ва задач (от масок в социальных сетях, до автопилота в электроавтомобилях). Поиск синонимов и системы автоматического дополнения текста(T9) также используются каждый день миллионами пользователей. Это лишь малая часть задач, которые можно решить используюя алгоритм поиска ближайшего соседа (или поиска K-ближайших соседей).

Вот ещё некоторые задачи, о которых хотелось бы упомянуть:

- Поиск дубликатов (определить являются ли 2 текстовых документа одинаковыми)
- Задача кластеризации (Определить, как какой группе относится выбранный объект)
- Поиск ближайших географических объектов (карты)
- Поиск схожих фрагментов в фильмах или музыке

Именно поэтому так важно искать новые подходы для улучшения скорости данного алгоритма. Чтобы достичь поставленную цель, необходимо разработать структуру данных, которая сможет наиболее эффективно осуществлять две операции добавления и поиска. Вариантов подходящих структур - огромное множество. Например, некоторые могут быть построены на базе вектора, списка, дерева, графа (в виде сети). Некоторые поддерживают точные поиск, а некоторые только приближённый. Некоторые формируются по средствам детерминированных алгоритмов, а некоторые использоуют рандомизированный подход.

Моё исследование буддет в основном основываться на изучении графовых структур (в виде сетей), так как они более современные и эффективные (подробнее ниже). Перед собой я ставлю следующие задачи:

- Изучить какие структуры для поиска ближайшего соседа существуют
- Исследовать феномен тесного мира.
- Обосновать выбор именно графовых структур, а также подробнее исследовать те из них, которые имеют свойствр тесного мира.
- Разработать необходимые абстракции для работы с подобными структурами
- Провести сравнительный анализ

# 2 Основная часть

## 2.1 История развития структур для поиска ближайшего соседа

## 2.2 Феномен тесного мира

Для того, чтобы понять, каким должен быть граф для оптимальной работы нашего алгоритма, я предлагаю обратиться к структурам, которые возникают само собой в природе, к графам, которые формирует общество.

### 2.2.1 Теория 6 рукопожатий

Венгерский писатель Karinthy Frigyes Ernő в 1928г в рассказе "Звенья цепи"впервые сформулировал данную проблему. Согласно ей, любые два человека на планете связаны через 5-6 общих знакомых.

Рассуждая над этой проблемой, Stanley Milgram - американский социальный психолог и педагог в своей статье "The Small World Problem"описал проведённый им эксперимент: Он выдал 300 писем жителям из разных городов и попросил доставить их одному человеку из Бостона (США). Важным условиям было то, что люди могли передавать письма только своим знакомым, которые по их мнению могли знать целовека-цель. По результатам исследовани даже не смотря на то, что до места назначения дошли далеко не все письма, те, которым это удалось, прошли в среднем через цепь из 5-6 человек.

Данные наблюдения кажутся очень полезными для нас. Ведь если нам удастся воссоздать граф, который с хорошей точностью моделируют общественные сети, мы сможем из любой вершины графа добираться до цели за сравнимо малое число посредников. Причём для этого нам даже не придётся использовать сложные алгоритмы поиска. Вспомним эксперимент Милгрема, в нём каждый человек не пытался искать оптимальный путь от него до цели, он лишь отправлял письма тем знакомым, которые казались ближе к месту назначения. Формально - это простой жадный поиск, который в эксперименте Милгрема прошёлся всего лишь по 5-6 промежуточным значениям и, что не мало важно, достиг цели.

### 2.2.2 Случайные графы Erdős–Rényi

Попробуем предположить, что общественные сети можно представить, как случайный граф $G(n,p) = V, E$ состоящий из $n$ вершин, с вероятностью проведения ребра p. Логично предположить, что $np = const = \lambda$. Можно объяснить это так: структура графа не должна зависить от кол-ва вершин в нём. Он всегда должен выглядеть примерно одинаково ($deg(v) \approx$

*const*). Ну или ещё одно описание: кол-во моих друзей не увеличится, если население планеты увеличится в несколько раз. Обоснуем это формально:

$$P(deg(v) = k) = \binom{n}{k} p^k (1-p)^{1-k} \Rightarrow deg(v) \sim Bin(p) \tag{1}$$

Так как $np = const = \lambda : \mathbb{E}[deg(v)] = np = \lambda$

Так мы описали граф, изучением которого занимались два великих Венгерских математика Paul Erdős и Alfréd Rényi. (Причём $\lambda$ обычно выбирается много меньше чем $n$, ведь наш круг общения никчёмно мал по сравнению с 8 млрд. людей планете)

Разультаты, к которым пршли данные математики показали, что случайные графы имеют сравнительно малый средний диаметр графа $d(G)$. Где

$$d(G) = \frac{1}{|V|} \sum_{u,v \in V} d(u,v) \tag{2}$$

$$d(u,v) = \min_{k \in \mathbb{R}} \{\exists a_i,\ i = \overline{1k} : a_1 = u, a_k = v, (a_i, a_{i+1}) \in E\} \tag{3}$$

Казалось бы, малый диаметр - всё, что нам нужно. И это было бы так, если бы мы не использовали жадный алгоритм для поиска. Случайные рёбра в графе помогают нам легко и быстро приблизиться к месту назначения, однако отсутствие локальных рёбер приводит к большому кол-ву локальных минимумов, что ведёт за собой большую погрешность полученном результате, пример:

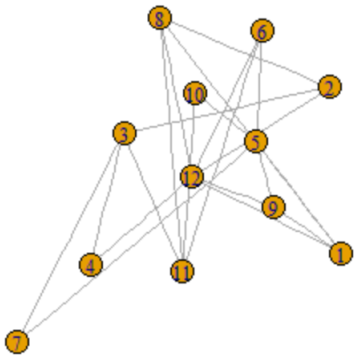


Рис. 1: Случайный граф

Взглянув на рисунок 1, можно заметить, что если мы хотим добраться до вершины 3, используя жадный поиск, то это нам удастся только из непосредственных соседей вершины номер 3. Это говорит о том, что такой граф хорош в навигации на больших масштабах, но плох при локальном поиске. Что не состыкуется с теорией 6 рукопожатий

Ошибка в наших рассуждениях была в том, что мы предположили, что связи между людьми абсолютно случайны, но ведь это не так. Милгрэм, описывая результаты своего

эксперимента, делал акцент на то, что общественные графы состоят из кластеров. Это можно описать так: Вероятность знакомства с человеком тем больше, чем мы ближе; А также, вероятность, того, что люди дружат прямо пропорциональна кол-ву их общих друзей. Наличие данной концепции образовало бы больше локальных рёбер, что решило бы проблему локальных минимумов. Формально это можно описать, через коэффицент кластеризации вершины:

$$cc(v) = \frac{\#\{(x,y) \in E,\ x,y \in V : (x,v),(y,v) \in E\}}{\#\{(x,y),\ x,y \in V : (x,v),(y,v) \in E\}}$$

По аналогии коэффицент кластеризации графа:

$$cc(G) = \frac{1}{|V|} \sum_{v \in V} cc(v)$$

Чем он больше, тем сильнее кластеризован наш граф.

### 2.2.3 Тесные графы

В рассуждениях выше мы пришли к выводам, что графы в реальном мире обладают двумя основными свойствами: малым диаметром и большим коэффицентом кластеризации. В 1998г, наблюдая за реальными сетями, к тем же самым выводам пришли два американских математика: Дункан Уоттс и Стивен Строгац. Они выделили такие графы в отдельную группу "Тесные графы". формально их можно записать так:

$$\Big(G(U,V) - \text{Тесный граф}\Big) \Leftrightarrow \begin{cases} d(G) \approx 0 \\ cc(G) \approx 1 \end{cases}$$

В практической части, будет показано, что $cc(G) \to 0$ с ростом n, где G - случайный граф по модели Erdos-Renye

Хочется ещё подметить, что тесные графы встречаются очень часто в природе. Нейронные сети в нашем мозге, карты дорог, пищевые цепочки. Их формирование кажется впоне логичным. Мозгу необходимо, чтобы сигнал быстро перемещался между нейронами, для быстрой обработки информации и реакции. Можно утверждать, что другие типы графов в данных примерах просто не прошли естественный отбор.

## 2.3 Подход Пономаренко

В 2012 году математики А.А.Пономаренко, Ю.А.Мальков, А.А.Логвинов и В.В.Крылов опубликовали статью, в которой предложили свой подход к созданию графов со свойствами

тесного мира. А также предоставили оптимизацию для алгоритма жадного поиска. Так как сама модель очень сильно опирается на этот алгоритм, предлагаю начать изучение именно с него.

### 2.3.1 Жадный поиск

Данный алгоритм предназначен для эффективного поиска вершины в графе. Идейно он очень прост. На каждой итерации цикла, мы смотрим все соседние вершины и идём к той, что ближе всего к цели. Если таких вершин нет, то мы нашли то, что искали.

- Преимуществом данного алгоритма является его быстродействие по сравнению с другими методами.
- Недостаток же кроется в его неточности. Может существовать такая вершина отличная от нашей, соседи которой будут дальше от цели, чем она сама. Такие вершины, как уже упоминалось, принято называть локальными минимумами

Блок схему алгоритма можно увидеть на рис 2.

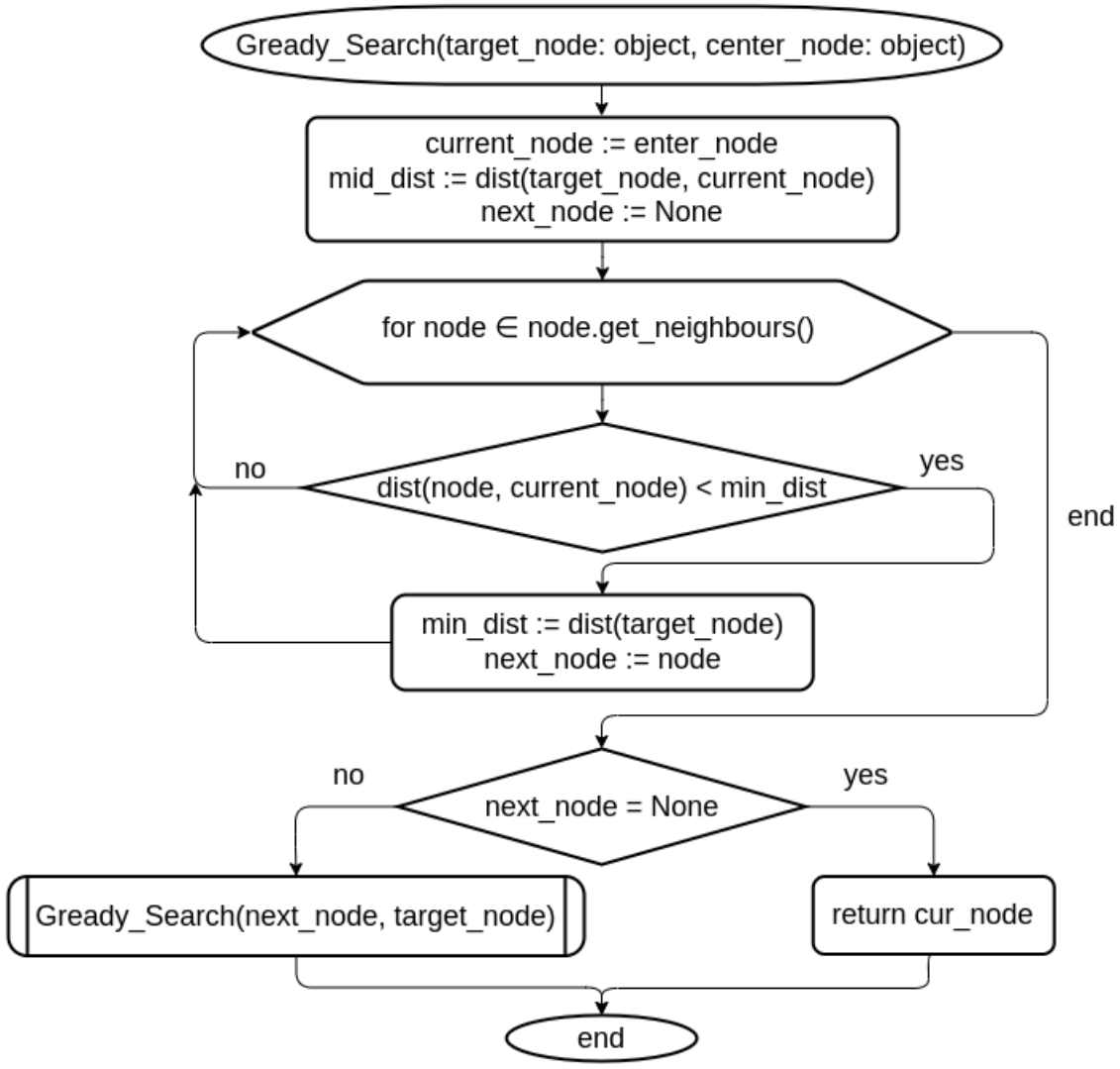


Рис. 2: Блок схема жадного поиска

Хочется подметить, что этот алгоритм относится к классу рандомизированных алгоритмов, так как сам поиск мы начинаем в случайной вершине. На практике такой подход показывает результат куда лучше, чем если бы стартовая вершина была детерминированной.

Математики из нижнего новгорода, упомянутые выше, предложили способ, как можно улучшить данный алгоритм, уменьшив вероятность попадания в локальные минимумы. Его блок схему вы можете найти на рис 3. Идея заключается в том, чтобы запустить жадный поиск несколько раз от произвольных вершин и выбрать ту, что окажется ближе. Данный подход рационален, так как мы подразумеваем, что работаем в "Тесном"графе, а значит, поиск не будет занимать много времени. Они также пришли к выводу, что достаточно делать $m \approx \log n$ повторений

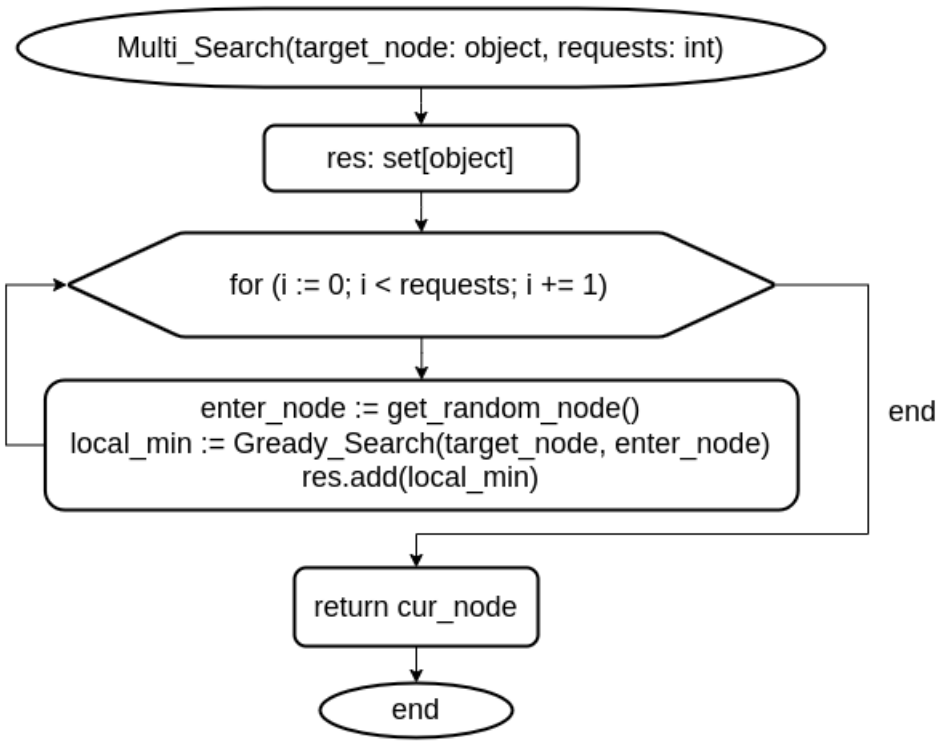


Рис. 3: Блок схема многократного жадного поиска

### 2.3.2 Построение структуры

Для построение структуры нам необходимо взять пустой граф и добавлять в него вершины поэлементно, опираясь на следующий алгоритм: Перед добавлением новой вершины v в граф, используя жадный поиск, мы ищем её потенциальное окружение (самые близкие к ней вершины, присутствующие в графе на данный момент). Выбрав из них только k штук (здесь k - параметр, который можно варьировать) связываем их с целевой вершиной v. Блок схему данного Алгоритма можно увидеть на рис 4.

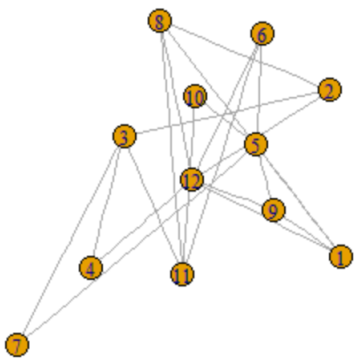


Рис. 4: Блок схема многократного жадного поиска

Хочется подметить несколько важных аспектов:

- Граф сильно кластеризован по определению, так как мы связываем вершины только с их ближайшими соседями
- Во время создания структуры, данные должны приходить случайно! Игнорирование этого требования может привести к увеличению среднего диаметра

Поподробнее остановимся на втором пункте.
Рассмотрим граф, изображённый на рис 5:

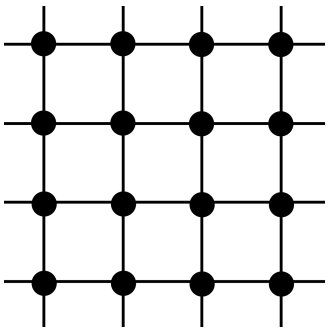

Рис. 5: Пример

Заметим, что даже такой крохотный пример уже имеет средний диаметр = 2.375. Если мы увеличим сетку в 16 раз, то получим средний диаметр = 9.5. То есть мы пришли к тому, что этот граф на 256 вершинах будет иметь средний диаметр больше, чем реальный граф с 8 млрд вершинами. Но это ещё не всё, заметим, что коэфицент класстеризации тут вовсе равен нулю. Так что, этот пример не просто слабо похож на тесный граф, а является его полной противоположностью.

Для понимания, как случайность спасает нас в данной ситуации, разделим построение графа на 2 части:

- В самом начале формирование графа, вершин в нём будет немного. Случайный поток данных приведёт к тому, что узлы будут достаточно далеко друг от друга (вероятность обратного пренебрежимо мала). Поэтому Ближайшие соседи вершины v, которые будут найдены посредством жадного алгоритма, будут хоть и самыми близкими из текущих, однако всё ещё будут находиться на достаточно большом расстоянии к v.
- С течением времени, ситуация начнёт меняться, ведь вершин будет становиться всё больше, а значит, и расположены они будут куда плотнее. Вершины добавленные

под конец формирования графа, будут иметь соседей, которые будут к ним очень близки не только в сравнении с другими, но и по абсолютной величине.

Другими словами, в в самом начале формирования графа образуются длинные связи, которые помогают быстро перемещаться по графу, а ближе к концу - короткие, которые спасают от локальных минимумов.

## 2.4 Подход Клайнберга

Следующий подход предложил Американский математик Джон Клайнберг. Перед началом разбора, мне бы хотелось сделать некоторую поправку. Джон Клайнберг в своих статьях нигде не упоминает о коэффиценте класстеризации. Вместо этого он говорит только о наличии длинных и коротких рёбер. Это немного расширяет класс "Тесных"графов, однако идейно ничего не меняет.

Изучим заново граф, который изображён на Рис. 5. Он состоит только из коротких связей, а значит, тесным графом не является. Есть разные способы решить эту проблему, давайте рассмотрим предложение Джона Клайнберга, так как его метод формирования длинных рёбер очень естественно вписывается в окружающий нас мир. Идея проста: чем человек дальше от меня, тем меньше вероятность нашего знакомства. Формально его идею можно записать так, Графу даются 2 параметра p, k:

- p - радиус. Все вершины, расстояние между которыми меньше или равно p, мы соединяем ребром и и называем короткой связью
- k - кол-во длинных связей. После добавления коротких связей, останется только пройтись всевозможным парам вершин и расставить рёбра в зависимости от следующего распределения:

$$P(u \to v) = \frac{1}{d(u,v)^r}\frac{1}{C}, \text{ где } C = \sum_{u \neq z \in V} \frac{1}{d(u,z)^r} \tag{4}$$

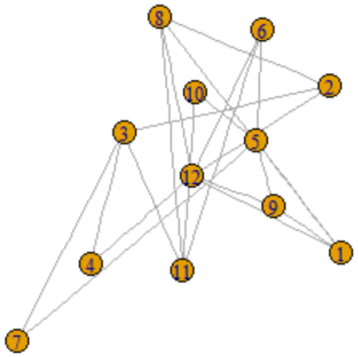


Рис. 6: Блок схема построение графа по методу Джона Клайнберга

## 2.5 Предложени по модификации

Оба подхода хорошо себя показывают в проблеме поиска ближайшего соседа. Однако Хочется предложить модификацию того алгоритма, который был предложен Нижегородскими математиками.

Есть гипотеза, что кол-во длинных рёбер, которые образовываются посредством данного алгоритма недостаточно для эффективной навигации по графу. Кроме того, нужно следить за тем, чтобы данные шли случайно, что тоже иногда может быть не очень удобно.

Для решения этой проблемы, предлагаю объединить данный подход с классическим методом построения тесного графа Уоттса–Строгаца. Идея заключается в следующем:

1) Жадным поиском ищем окружение вершины v, которую хотим добавить

2) Выбираем k(параметр) ближайших вершин.

3) Связываем вершину v лишь с некоторыми из выбранных вершин. Вероятность ребра теперь будет являться отдельным параметром и задаваться заранее

4) Каждое непроведённое ребро будет компенсироваться другим. Это новое ребро будет связывать нашу вершину v с другой абсолютно случайно выбранной вершиной в графе.

В практической части, я собираюсь проверить эффективность данного подхода в сравнении с другими методами. А также подобрать параметр p, при котром модель будет давать наилучшие результаты.

# 3 Практическая часть

## 3.1 Описание абстракций

Для начала, я бы хотел описать, какие классы были реализованы, каким образом и для чего они преднозначены.

### 3.1.1 Класс Graph

Данный граф является основным. Он реализует все самые необходимые инструменты для взаиможействия с графами, такие как: добавление вершины в граф, добавление ребра, очистка графа и т.д.

Данная абстракция была реализована на основе хэш таблицы (std::unordered_map). Такой контейнер был выбран не случайно, так как он осуществляет очень быстрый доступ к произвольным данным (Добавление, поиск, удаление за O(1)).

Этот класс является шаблонным и может хранить в себе объекты любой природы. Главное, чтобы они удовлетворяли двум требованиям:

- Наличие перегрузки оператора равенства
- Наличие функции хэширования (для std::unordered_map)

Объявление класса:

```
template<typename TObject, typename HashFunc>
class Graph {
    std::unordered_map<TObject, std::vector<TObject>, HashFunc> graph;
    std::vector<TObject> nodes;
    size_t size;
public:
    Graph() = default;
    Graph(const Graph& g) = delete;
    Graph(Graph&& g) = delete;

    void add_node(const TObject& p);
    void add_edge(const TObject& p1, const TObject& p2);
    void delete_edge(const TObject& p1, const TObject& p2);
```

```
        void clear();
        std::size_t get_size() const;
        bool consists_node(const TObject& p) const;
        bool consists_edge(const TObject& p1, const TObject& p2) const;
        const std::vector<TObject>& get_neighbours(const TObject& p) const;
        const TObject& get_random_node() const;
        friend std::ostream& operator<< <>(std::ostream& out, const Graph& graph);
        ~Graph() = default;
    };
```

Хочу отдельное внимание обратить на метод get_random_node. Его необходимость станет очевидна вдальнейшем, пока мне бы хотелось рассказать о дилеме, с которой я столкнулся во время её написания. Для реализации этого метода, мы должны иметь возможность выбирать из хэш таблицы любой ключ случайно, однако это возможно сделать только за линейное время, из-за отсутствия в unordered_map обращения по индексу. Для решения этой проблемы мною было принято решение создать отдельный вектор из объектов, так как в нём, используя случайные индексы, мы без проблем сможем выбрать абсолютно любой объект за O(1). Это привело к следующим последствиям:

- Увеличение кол-во памяти необходимое для хранения структуры
- Ухудшение асимптотической скорости удаление выбранных элементов (до линейной)

Однако все эти недостатки никак негативно не повлияли конкретно в решении задач текущего проекта

### 3.1.2 Классы Point и Point3D

Данные абстракции в моём проекте используются, как объекты, вершины графа. Они являются больше примером для наглядной демонстрации работы других классов. Заголовки выглядят следующим образом:

```
    class Point {
        public:
            double x;
            double y;
            Point() = default;
```

```
        Point(const Point& p) = default;
        Point(Point&& p) = default;
        Point(double x, double y);
        friend std::ostream& operator<<(std::ostream& out, const Point& p);
        bool operator==(const Point& p) const;
        static double dist(const Point& p1, const Point& p2);
        class HashPoint {
        public:
            std::size_t operator()(const Point& p) const;
        };
        ~Point() = default;
    };
    class Point3D {
    public:
        double x;
        double y;
        double z;
        Point3D() = default;
        Point3D(const Point3D& p) = default;
        Point3D(Point3D&& p) = default;
        Point3D(double x, double y, double z);
        friend std::ostream& operator<<(std::ostream& out, const Point3D p);
        bool operator==(const Point3D p) const;
        static double dist(const Point3D p1, const Point3D p2);
        class HashPoint {
        public:
            std::size_t operator()(const Point3D& p) const;
        };
    };
```

### 3.1.3 Класс Nswg

Данный класс является абстрактным. Внутри он содержит всё необходимое для реализации жадного поиска, однако на этом всё. Процесс формирования самого графа ложится на

плечи потомков (поэтому метод load_nodes помечен, как виртуальный). Данное решение было принято, чтобы не копировать в каждый новый класс одинаковые методы жадного поиска.

```
template<typename TObject, typename HashFunc>
class Nswg {
protected:
    Graph<TObject, HashFunc> graph;
    std::size_t sum_of_degrees = 0;

    class ClosestToCompare {
        const TObject& base;
    public:
        ClosestToCompare(const TObject& base) : base(base) {};
        bool operator()(const TObject& p1, const TObject& p2) const;
    };

    void multi_search(const TObject& target_node,
                      std::set<TObject, ClosestToCompare>& res,
                      std::size_t count) const;

public:
    virtual void load_nodes(const std::vector<TObject>& objects) = 0;
    double get_mean_deg() const;
    double get_cc() const;

    const std::vector<TObject>& get_nodes() const;

    const TObject& greedy_search(const TObject& target_node, const TObject& sta
    std::pair<TObject, double> get_best_element(const TObject& target_node, std

    const std::vector<TObject>& get_neighbours(const TObject& p) const;
    const TObject& get_random_node() const;
    void clear();
```

```
    std::size_t get_size() const;
};
```

Также, хотелось бы ещё рассказать почему я использую внутренние дополнительный класс ClosestToCompare(). Как я уже описывал, в жадном поиске нам нужно искать k вершин наиболее близких с нашей. Постоянные сортировки могли достаточно сильно ухудшить производительность модели, поэтому я принял решение, написать предикат, который будет сортировать вершины в set() сразу так, как нужно:

- В конструктор класса мы передаём объект v
- Чем ближе вершина к v, тем первее она будет находится в set.
- Однако возникнет проблема, что 2 узла на одном расстоянии теперь будут считаться одинаковыми, а значит, в set добавится только один из них. Для решения этой проблемы, мне пришлось добавить второе условие, котороые сравнивает хеши вершин и сортирует на основе их значений:

```
template<typename TObject, typename HashFunc>
bool Nswg<TObject, HashFunc>::ClosestToCompare::operator()(const TObject& p1, c
    double d1 = TObject::dist(p1, base);
    double d2 = TObject::dist(p2, base);
    if (d1 != d2) {
        return TObject::dist(p1, base) < TObject::dist(p2, base);
    } else {
        HashFunc hash;
        return hash(p1) < hash(p2);
    }
}
```

### 3.1.4 PonomarenkoGraph

Является наследником класса Nswg и с помощью метода load_nodes реализует построение графа. Алгоритм построения уже был разобран в блок схеме выше.

```
template<typename TObject, typename HashFunc>
class PonomarenkoGraph : public Nswg<TObject, HashFunc> {
    std::size_t queries_count = 5;
```

```
    std::size_t edges_count = 7;
public:
    PonomarenkoGraph(std::size_t queries_count = 5, std::size_t edges_count = 7
    // PonomarenkoGraph(const std::vector<TObject>& objects);

    PonomarenkoGraph(const PonomarenkoGraph& pg) = delete;
    PonomarenkoGraph(PonomarenkoGraph&& pg) = delete;

    void add_node(const TObject& obj);
    void load_nodes(const std::vector<TObject>& objects) override;

    friend std::ostream& operator<< <>(std::ostream& out, const PonomarenkoGrap

    ~PonomarenkoGraph() = default;
};
```

### 3.1.5 KleinbergGraph

Является наследником класса Nswg и с помощью метода load_nodes реализует построение графа. Алгоритм построения уже был разобран в блок схеме выше.

### 3.1.6 RandomGraph

Данный класс был реализован на основе графов Erdos-Renye. Был создан для сравнительного анализа

```
template<typename TObject, typename HashFunc>
class RandomGraph : public Nswg<TObject, HashFunc> {
    double mean_neighbours;
public:
    RandomGraph(double mean_neighbours);
    RandomGraph(const RandomGraph& rg) = delete;
    RandomGraph(RandomGraph&& rg) = delete;

    void load_nodes(const std::vector<TObject>& objects) override;
```

```
    ~RandomGraph() = default;
};
```

### 3.1.7 Figure

Данная абстракция была создана с целью упрощения визуализации графов и графиков. Он работает по очень простому принципу: Под капотом она пишет скрипт на питоне, который записывает в отдельный файл. После чего в отдельном процессе этот файл запускается и выводит данные в виде графиков. В Питоне я использовал библиотеку Plotly

```
class Figure {
    std::ofstream script;
    std::string name;
    void write_vector_as_list(const std::string& name, const std::vector<double
public:
    Figure() = delete; // TODO: why not
    Figure(const std::string& name);
    Figure(const Figure& f) = delete;
    Figure(Figure&& f) = delete;
    void add_graph(const Nswg<Point, Point::HashPoint>& g,
                   std::size_t marker_size,
                   std::size_t line_width,
                   const std::string& marker_color,
                   const std::string& line_color,
                   const std::string& name);
    void update_title(const std::string& title);
    void add_markers_and_lines(const std::vector<double>& x,
                               const std::vector<double>& y,
                               std::size_t marker_size,
                               std::size_t line_width,
                               const std::string& marker_color,
                               const std::string& line_color,
                               const std::string& name);
    void add_markers(const std::vector<double>& x,
                     const std::vector<double>& y,
```

```
                    std::size_t marker_size,
                    const std::string& marker_color,
                    const std::string& name);
    void add_line(const std::vector<double>& x,
                  const std::vector<double>& y,
                  std::size_t line_width,
                  const std::string& line_color,
                  const std::string& name);
    void update_xaxes(const std::string& x_title,
                      const std::string& axis_type);
    void update_yaxes(const std::string& y_title,
                      const std::string& axis_type);
    void show();
    ~Figure();
};
```

#### 3.1.8 DynamicDistribution

### 3.2 Эксперименты и наблюдения(параметры)

#### 3.2.1 CC(G) тест

Для начала убедимся в утверждении выдвинутым ранее: Что $CC(G) \to 0$ при $n \to \infty$: Взглянем на график случайного графа по модели Erdős–Rényi со средней степенью вершины 5, с увеличивающейся выборкой:

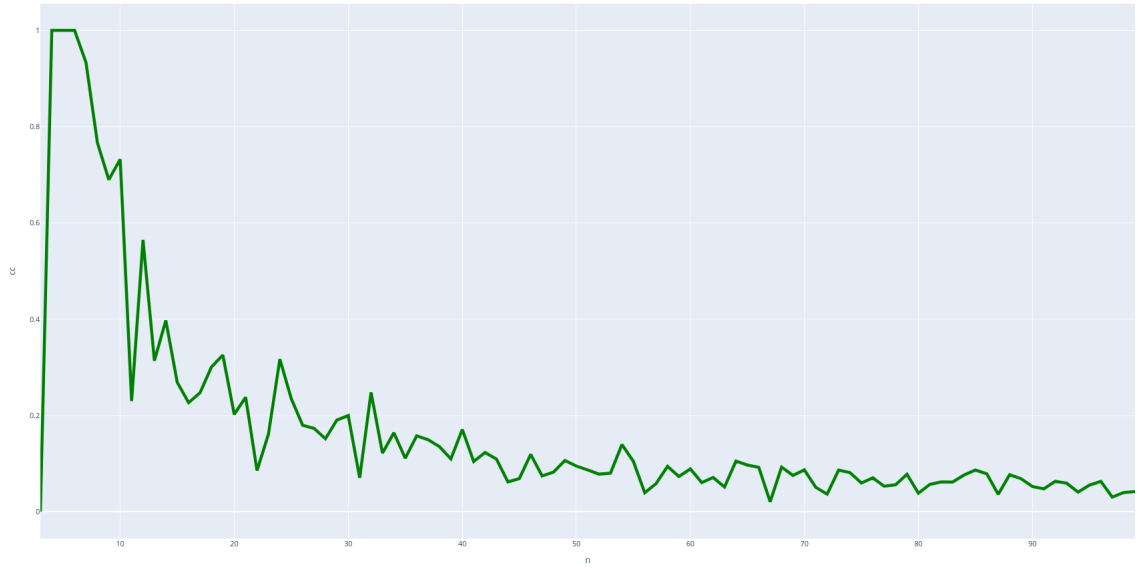


Рис. 7: Зависимость CC(RandomGraph) с ростом выборки

Можем наблюдать стабильный нисходящий тренд.

Теперь рассмотрим зависимость коэффицента класстеризации от степени вершин. В данном примере моя модель содержит 3000 вершин, координаты каждой генерируются случайно в диапазона от 0 до 200. Чтобы сравнение было корректным, мне пришлось подобрать параметры каждого графа, чтобы степени их вершин примерно совпадали.

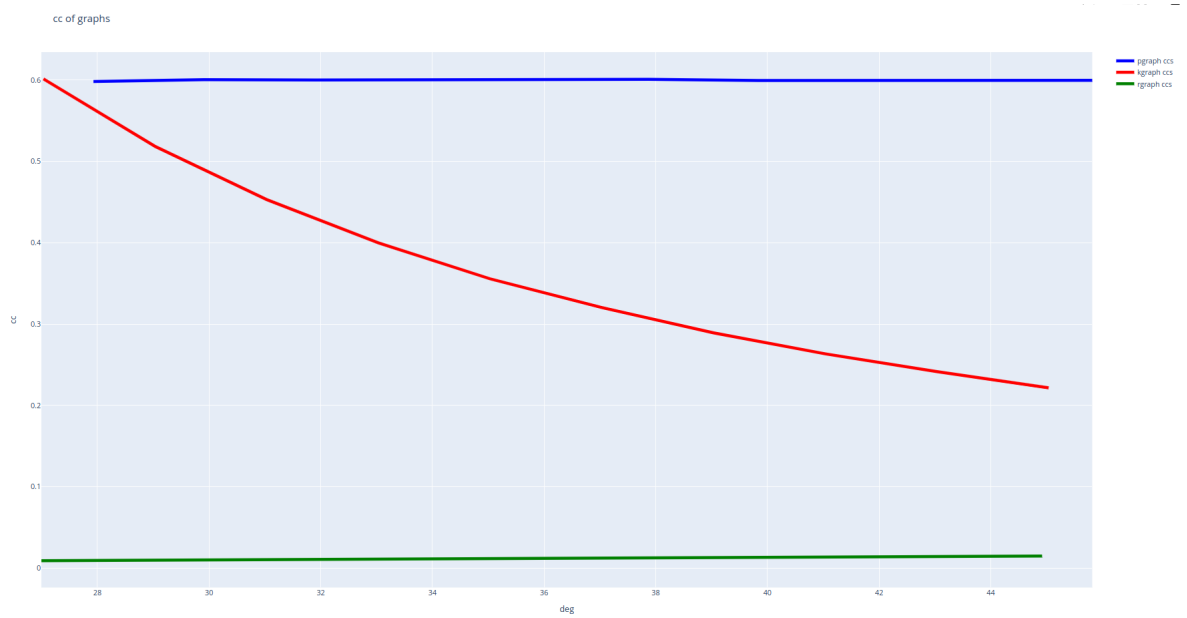


Рис. 8: cc от степени вершин

Выводы исходя из графика:

- граф Пономаренко стабильно дуржит коеффицент класстеризации на уровне 0.6 вне зависимости от степени вершины

- граф Клайнберга же равне 0.6 только в начале, когда в нём отсутствуют длинные рёбра. После чего, по очевидными причинам, с ростом числа случайных рёбер заметен спад.
- Случайный граф Erdos-Renye на протяжении всего промежутка имеет неизменный, очень слабо класстеризован

Результаты согласуются с теоретическими предположениями

Исходные данные:

- для графа Клайнберга были выбраны следующие параметры: Кол-во коротких связей: 10. Кол-во длинных рёбер варьировалось от 0 до 10
- для графа Пономаренко первым параметром(число повторений) было выбрано 8 (Это оптимальное решение, обоснование будет чуть дальше). Второй параметр варьировал от 15 до 25
- В полностью случайном графе можно явно задать степень вершины, она варьирвалась от 20 до 40

### 3.2.2 Тест на среднюю длину пути

### 3.2.3 Loss тест

Теперь перейдём к самому главному, точность поиска. Определять точность будем по следующему принципу: Будем запускать алгоритм 1000 раз и суммировать расстояние от найденной вершины, до целевой(выбирается случайно). Чем меньше будет значение, тем лучше наш алгоритм справляется с поиском ближайшего соседа

Взглянем на график:

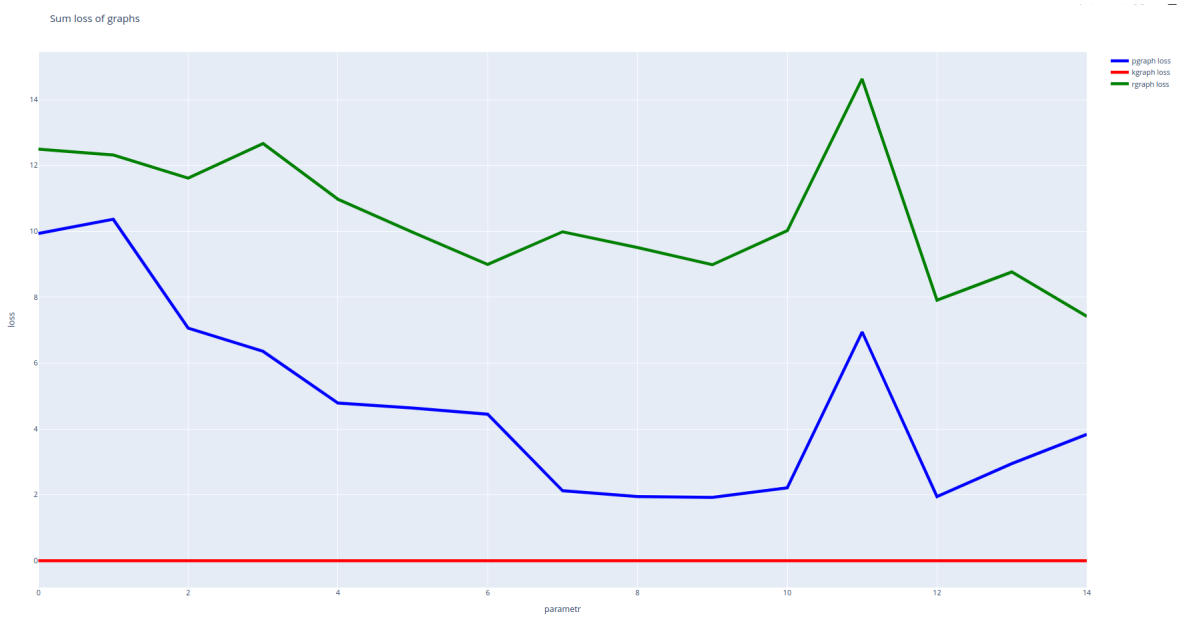


Рис. 9: Средняя ошибка, параметр "Кол-во повторных поисков"в гарфе Пономаренко = 1

Выводы исходя из графика:

- граф Клайнберга показал себя лучше всех, не допустив ни одной ошибки за все 1000 поисков
- граф Пономаренко, хоть и работает лучше случайного, но всё ещё стабильно ошибается

Теперь рассмотрим только граф Пономаренко и увеличим кол-во поисков:

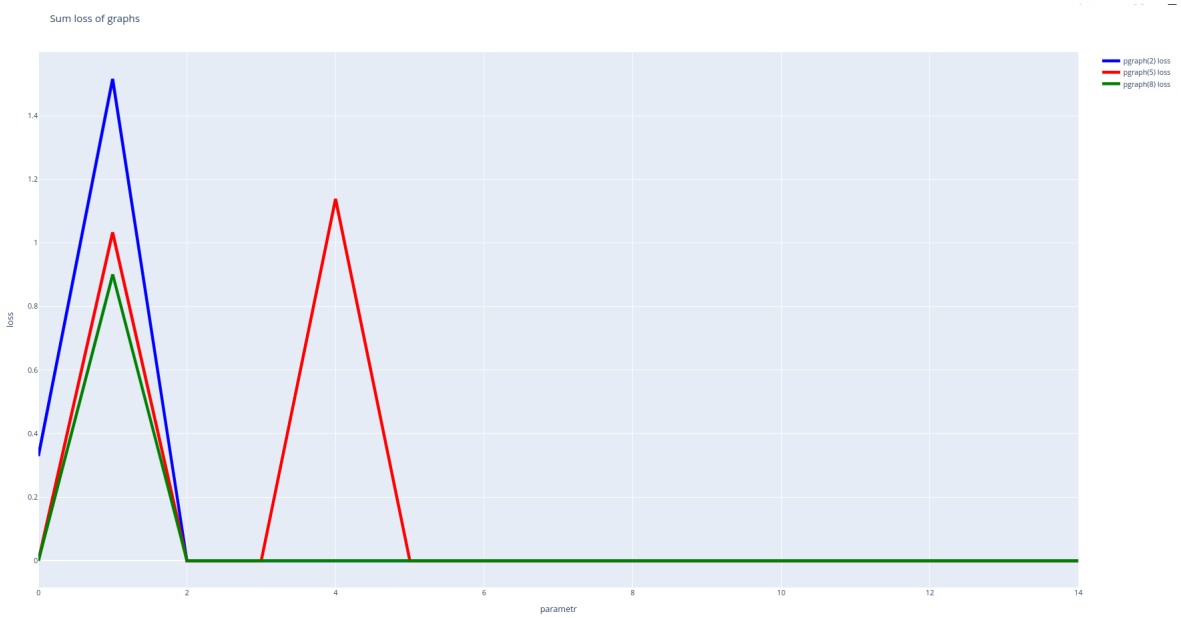


Рис. 10: Число поисков 2, 5, 7

Заметим, что действительно с ростом числа поисков, ошибка по данной метрике становится всё меньше Из наблюдений Нижегородских математиков достаточное кол-во повторений $\log(3000) = 8$ что согласуется с наблюдениями

# 4 Заключение

В заключение хочется сказать, что все необходимы задачи были выполнены:

- Были изучены разные виды графовых структур для поиска ближайшего соседа, а также
- Были расписаны их преимущества и недостатки
- Был полностью изучен феномен тесного мира
- Были изучены 2 структуры, которые опираются на этот метод
- Были разработаны очень удобные абстракции для работы с графовыми структурами
- Был проведён сравнительный анализ структур

Обе эти структуры показали свою эффективность в сравнении со случайным графом, за короткое кол-во путей они давали результат с очень малой ошибкой, что показывает их эффективность и применимость. Однако хочется сделать небольшую пометку, построение графа Клайнберга занимает очень много времени($O(n^2)$) Поэтому его алгоритм лучше использовать, как идею для внедрения в другие типы графов. Под идеей имеется в виду расстановка рёбер, опираясь на распределение. Алгоритм построения графа, который изучали Нижегородские математики, работает достаточно эффективно как по времени, так и по памяти, поэтому он применим в чистом виде.

Реализованные структуры очень легко масштабируются, можно внедрить ещё огромное кол-во других абстракций, которые без "костылей"впишутся в общую структуру классов. Поэтому данный проект можно продолжать развивать, я планирую сделать свою библиотеку для работы с разными графовыми структурами. Возможно, со специальными абстракциями для тестирования.