Правительство Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования «Национальный исследовательский университет «Высшая школа экономики»

Московский институт электроники и математики Национального исследовательского университета «Высшая школа экономики»

Департамент прикладной математики

ОТЧЕТ

По проекту «Сравнительный анализ алгоритмов поиска ближайших соседей на плоскости»

ФИО Руководителя Жукова Лилия Фаилевна
ФИО студента Вязов Глеб Дмитриевич
Дата 27.03.2024

Москва – 2024 г.

Содержание

1 Введение **3**

2 Цель и задачи проекта **3**
2.1 Цель проекта . . . 3
2.2 Задачи проекта . . . 3

3 Деревья **3**
3.1 KD-деревья . . . 3
3.1.1 Построение KD-дерева [2] . . . 3
3.1.2 Поиск в KD-дереве [3] . . . 4
3.1.3 Асимптотиика . . . 5
3.2 Другие алгоритмы, основанные на деревьях . . . 5
3.2.1 Principal Axis Trees (PAT) [4] . . . 5
3.3 Деревья. Вывод . . . 6

4 Алгоритмы хэширования [6] **6**
4.1 Построение хэш-таблицы . . . 7
4.2 Поиск в хэш-таблицах . . . 8
4.3 Асимптотика . . . 8
4.4 Выводы . . . 8

5 Поиск в многомерном пространстве [5] **9**
5.1 Идея алгоритма . . . 9
5.2 Асимптотика . . . 9

6 Практическая часть **9**

7 Вывод **10**

8 Литература **10**

1 Введение

Имеется набор N точек на плоскости. Положение каждой точки задается набором координат. Задача динамическая. Пусть даны две точки $A(x_1, y_1), B(x_2, y_2)$, тогда расстояние между ними будет считаться по евклидовой формуле: $\rho(A, B) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. Требуется найти ближайшую к заданной точку.

Постоянно растущий объем информации приводит к увеличению времени и усложнению вычислений для решения этой задачи. Для поиска ближайших соседей используются различные алгоритмы. Разнообразие этих методов не только устраняет некоторые сложности, но и делает их пригодными для различных приложений, таких как распознавание картинок, поиск в мультимедийных данных, поиск информации, базы данных, вычислительная геометрия и др. [7], [8], [9], [10], [11]

2 Цель и задачи проекта

2.1 Цель проекта

Выполнить обзор алгоритмов поиска ближайших соседей на плоскости и реализовать наиболее эффективный.

2.2 Задачи проекта

1. Найти и изучить современные алгоритмы поиска ближайших соседей на плоскости
2. Сравнить их по различным критериям
3. Реализовать наиболее эффективный из изученных алгоритмов
4. Выявить достоинства и недостатки реализованного алгоритма

3 Деревья

3.1 KD-деревья

3.1.1 Построение KD-дерева [2]

KD-дерево - это двоичное дерево. Буква k отвечает за размерность пространства, в котором строится дерево, в нашем случае, $k = 2$. Алгоритм построения 2D-дерева:

1. Каждый узел, помимо записи, содержит два указателя, которые равны нулю или указывают на другой узел в дереве. Эти указатели можно рассматривать как указатели на поддерево.

2. Каждому узлу в соответствие ставится значение дискриминатора $DISC(P)$ от 0 до k-1. В двумерном случае, 0 или 1. Все узлы на одинаковом уровне дерева имеют равное значение дискриминатора. Корневой узел имеет значение дискриминатора 0. Рассмотрим узел P. Обозначим через $DISC(P)$ значение дискриминатора.

   (a) Если $DISC(P) = 0$, то плоскость разбиваем вертикальной прямой: $x = x_0$. В правую ветку попадают точки, у которых координата **x** больше чем у исследуемой точки, в левую – с меньшей или равной

   (b) Если $DISC(P) = 1$, то плоскость разбиваем горизонтальной прямой: $y = y_0$. В правую ветку попадают точки, у которых координата **y** больше чем у исследуемой точки, в левую – с меньшей или равной

   В общем случае определяется функция: $NEXTDISC(i) = (i{+}1)\ mod\ k = 2$.

3. Назовем один указатель $LOSON(P)$, а другой - $HISON(P)$ и определим их значения

4. Пусть $DISC(P) = j$, тогда для всех узлов Q в $LOSON(P) \Rightarrow K_j(Q) \leq K_j(P)$. Для всех узлов R в $HISON(P) \Rightarrow K_j(R) > K_j(P)$

5. Понятно, что $NEXTDISC(DISC(P)) = DISC(LOSON(P)) = DISC(HISON(P))$.

6. Учитывая, что $k = 2$, то $LOSON(P)$ - это левое поддерево, а $HISON(P)$ - правое поддерово узла P.

3.1.2 Поиск в KD-дереве [3]

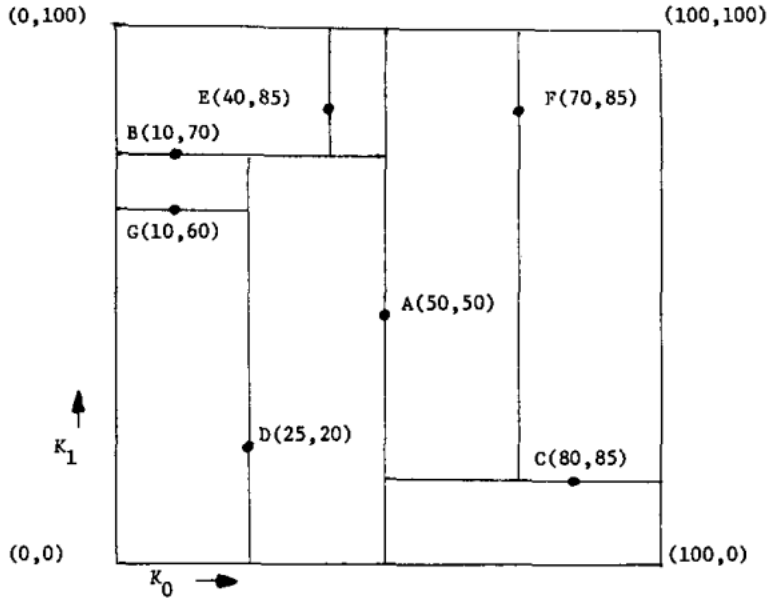
Алгоритм поиска работает рекурсивно. В качестве аргумента функция принимает исследуемый узел. При первом вызове передается корень дерева. Пусть нужно найти m ближайших точек к точке P.

1. На каждом шаге алгоритм хранит массив из m ближайших соседей на данный момент. Если в какой-то момент находится точка, которая ближе к P, то она добавляется в массив вместо самой дальней точки.

2. Если исследуемый узел не является терминальным, то рекурсивная процедура вызывается для узла, который лежит в той же полуплоскости, что и точка P.

3. При этом может оказаться ситуация, что нужно рассматривать и второй узел. Это случай, когда окружность с центром в точке P пересекает второй узел. Радиус окружности равен расстоянию от точки P до самой дальней точки из массива.

Рис. 1: Пример разбиения плоскости с помощью 2D-дерева. (Источник [2]).



3.1.3 Асимптотиика

Считаем, что на плоскости n точек.

1. Время построения: $O(n \log n)$
2. Память: $O(n)$
3. Время поиска: $O(\log n)$

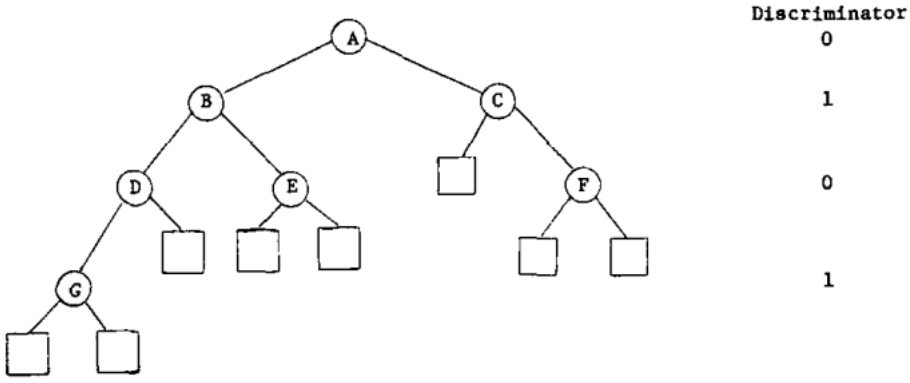
3.2 Другие алгоритмы, основанные на деревьях

3.2.1 Principal Axis Trees (PAT) [4]

Построение дерева поиска начинается с проецирования всего набора данных вдоль главной оси. Главная ось - это некоторая полуплоскость, ограниченная параллельными прямыми. Задаем параметр n_c - количество полуплоскостей, на которые разбивается главная ось. Затем набор данных разбивается вдоль главной оси на n_c отдельных полуплоскостей, причем каждая полуплоскость содержит примерно одинаковое количество точек. Процесс повторяется для каждого подмножества точек рекурсивно, пока каждое подмножество не будет содержать меньше чем n_c точек.

Пусть дана точка P и необходимо найти ближайших к ней соседей. Процесс поиска в глубину начинается с корневого узла и использует двоичный

Рис. 2: Пример построения 2D-дерева. (Источник [2]).



поиск, чтобы определить, в каком регионе находится точка P. Затем выполняется поиск дочернего узла, содержащего этот регион и процесс повторяется рекурсивно до тех пор, пока не будет пройдено все дерево.

Этот алгоритм имеет такую же асимптотику, как и KD-дерево. И в сравнении с KD-деревом работает примерно одинаково.

3.3 Деревья. Вывод

Алгоритмы, основанные на деревьях имеют общие черты:

1. Первым делом, строится само дерево. Часто для построения дерева плоскость нужно разбивать на полуплоскости по определенному алгоритму. Обычно занимает долгое время.

2. На основе построенного дерева осуществляет поиск за довольно быстрое время.

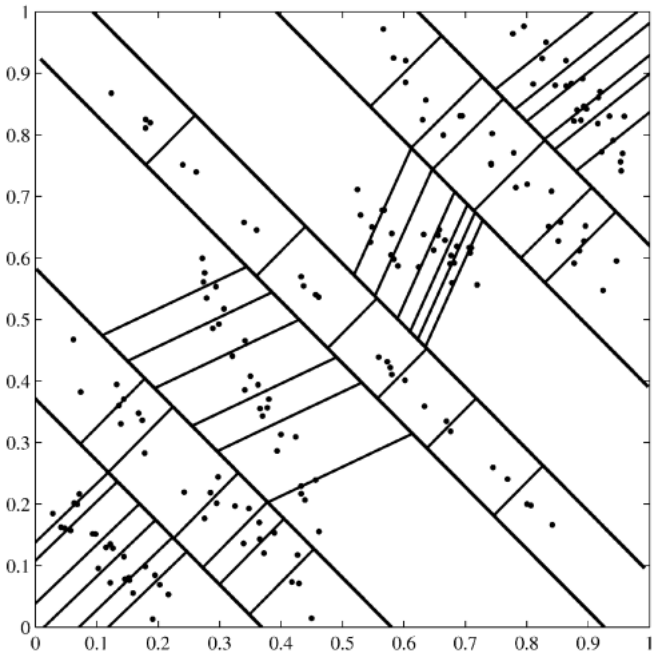
Из этих свойств следует очевидный минус деревьев. Использование древовидных структур нежелательно для динамических данных, когда количество запросов невелико по сравнению с затратами на перестройку дерева [5]

4 Алгоритмы хэширования [6]

Рассмотрим алгоритм Locality Sensitive Hashing (LSH). Суть алгоритма заключается в хэшировании точек с помощью нескольких хэш-функций таким образом, чтобы вероятность совпадения хэш-значений для точек, которые находятся рядом друг с другом, была выше вероятности для точек, которые находятся дальше друг от друга.

Пусть N - это семейство хэш-функций, отображающих $\mathbb{R}^d$ в некоторое конечное множество U. Для произвольных двух точек p и q, рассмотрим

Рис. 3: Картинка из [4]. Пример работы алгоритма PAT. Каждая главная ось разбилась на $n_c = 7$ отдельных полуплоскостей, в которых оказалось примерно по 4 точки.



процесс, в котором мы выбираем функцию h из N равномерно случайным образом, и анализируем вероятность того, что $h(p) = h(q)$. Множество N называется локально чувствительным (с соответствующими параметрами), если оно удовлетворяет следующем условиям ($P_1 > P_2$, P_1, P_2 - наперед заданные вероятности):

1. Задаем константы $R, c \in \mathbb{R}$
2. Если $\big||p-q|\big| \leq R$, то $p\big(h(q) = h(p)\big) \geq P_1$ (точки p и q лежат близко друг к другу)
3. Если $\big||p-q|\big| \geq cR$, то $p\big(h(q) = h(p)\big) \leq P_2$ (точки p и q лежит далеко друг от друга)

4.1 Построение хэш-таблицы

1. Фиксируем два параметра k и L. Выбираем L функций $g_j, j = 1, ..., L$ из N, где $g_j = (h_{1j}, ..., h_{kj})$, где хэш-функции $h_{1j}, ..., h_{kj}$ - выбираем случайным образом из N и с помощью них будем хэшировать точки.
2. Создаем L хэш-таблиц, где j-ая хэш таблица содержит результат применения хэш-функции g_j на исходных данных

4.2 Поиск в хэш-таблицах

Для поиска ближайших соседей точки q.

1. Считаем для всех $j = 1, ..., L$ значение $g_j(q)$ и считаем от нее расстояние до q. Если такая точность нас устраивает, то сохраняем эту точку
2. Далее есть два варианта:
   (a) Ввести некоторый параметр $L^{'}$ и прервать поиск после нахождения первых $L^{'}$ ближайших точек (включая дубликаты)
   (b) Продолжать поиск пока не найдем все точки из функций $g_j(q)$

4.3 Асимптотика

Считаем, что на плоскости n точек. Чтобы учесть худший случай, возьмем $k = \log_{\frac{1}{P_2}}(n)$ и $L = n^{\rho}$, где $\rho = \dfrac{\log(\frac{1}{P_2})}{\log(\frac{1}{P_1})}$. За τ обозначим время вычисления операции $h \in N$. На практике, эти параметры можно выбрать по-другому. Например, один из подходов - это оптимизация параметра k как функцию от набора данных и набора некоторых запросов

1. Время построения: $O(n^{\rho+1}k \cdot \tau)$
2. Память: $O(nL)$
3. Время поиска: $O(n^{\rho}k \cdot \tau)$

4.4 Выводы

Плюсы:

1. Асимптотическое время работы. Алгоритмы хэширования работают достаточно быстро.
2. Легкость реализации. Алгоритмы хеширования относительно просты в реализации и интеграции в существующие системы.

Минусы:

1. Точность. Алгоритмы хэширования возвращают лишь приближенное значение. Иного могут быть и ошибки.
2. Выбор хеш-функции. Подбор подходящей хеш-функции — критическая задача, влияющая на скорость и качество поиска. Плохой выбор хеш-функции может привести к плохому распределению данных и снижению эффективности.

Тем не менее алгоритмы хэширования считаются одними из самых эффективных для решения проблемы поиска ближайших соседей.

5 Поиск в многомерном пространстве [5]

5.1 Идея алгоритма

Основная идея заключается в том, чтобы уменьшить размерность исходного пространства, размерности d, и получить новое пространство, в котором вычисление расстояния между точками будет работать за O(1), а не за O(d).

1. Для всех точек исходного пространства $x = (x_1, ..., x_d)^T$ вычисляются: $\mu_x = \frac{1}{d}\sum_i^d x_i$ и $\sigma_x^2 = \frac{1}{d}\sum_i^d (x_i - \mu_x)^2$
2. Доказывается лемма: $dist(x, y)^2 \geq d((\mu_x - \mu_y)^2 + (\sigma_x - \sigma_y)^2)$, где μ_y, σ_y - это соответствующие величины для точки y и дальнейшем поиске используется именно это формула для подсчета расстояния между двумя точками
3. Чтобы найти m ближайших точек к т. P, перебираются все точки на плоскости и выбирают m самых близких, используя предыдущую формулу для расстояния

5.2 Асимптотика

Считаем, что на плоскости n точек и размерность исходного пространства равна d.

1. Время построения: $O(nd)$
2. Память: $O(n)$
3. Время поиска: $O(n)$

Этот алгоритм актуально применять для многомерных пространств.

6 Практическая часть

Собственная реализация KD-дерева хранится на гитхабе.

1. В файле `Point.py` объявлен класс точки.
2. В файле `prepare_data.py` объявлено две функции: `parse_file()` и `plot_of_point()`.
   (a) Функция `parse_file()` парсит файл с точками. Для этого нужно создать файл с именем `data.txt` (или другим, но тогда имя файла нужно передать в функцию). На каждой строке файла находиться одна точка. Сначала введите координату по x, затем через пробел, по y. Функция возвращает список точек.

(b) Функция `plot_of_point()` принимает на вход список точек и с помощью библиотеки `matplotlib` строит график точек этих точек.

3. В файле `KD_tree.py` объявлен класс `KDTree`. Основные функции:
   (a) Конструктор класса `__init__()`. Принимает на вход список точек и запускает функцию построения дерева.
   (b) Функция построения дерева `build_tree()`.
   (c) Функция поиска ближайшей точки `search_nearest_point()`. Принимает на вход искомую точку. Возвращает ближайшую точку.
4. В файле `main.py` приведен пример работы программы.
5. В файле `timing.py` реализован замер работы программы. Результат работы - 2 графика, первый **Время поиска.jpg** (5) и второй - **Время построения.jpg**. (4)

   Вывод. Алгоритм KD-деревьев показывает достаточную эффективность для поставленной задачи. Для задачи с граничными условиями алгоритм требует дополнительной проработки.

7 Вывод

В статье представлен анализ различных алгоритмических подходов к решению задачи поиска ближайших соседей.

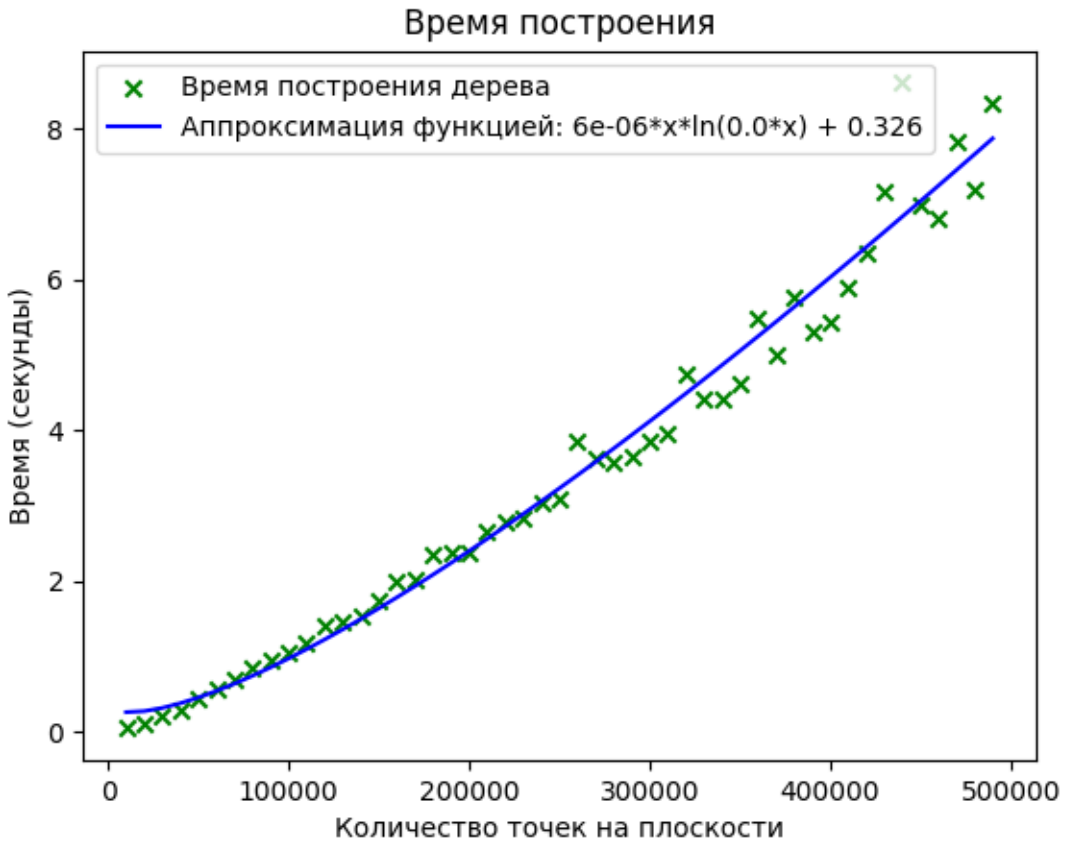
1. Анализ древовидных структур данных, в частности KD-деревьев, включая их построение, поиск и анализ асимптотической сложности. Также рассматриваются другие алгоритмы поиска на базе деревьев, например, PAT.
2. Исследуется процедура построения хэш-таблиц и методы поиска с использованием хэширования.
3. Исследуется поиск в многомерном пространстве, в целом, идея алгоритма сводится к уменьшению размерности пространства.
4. В практической части реализован алгоритм KD-дерева на языке программирования Python.

8 Литература

Список литературы

[1] Mohammad Reza Abbasifard, Bijan Ghahremani, Hassan Naderi, A Survey on Nearest Neighbor Search Methods, International Journal of Computer Applications (0975 – 8887), Volume 95– No.25, June 2014

Рис. 4: Время построения



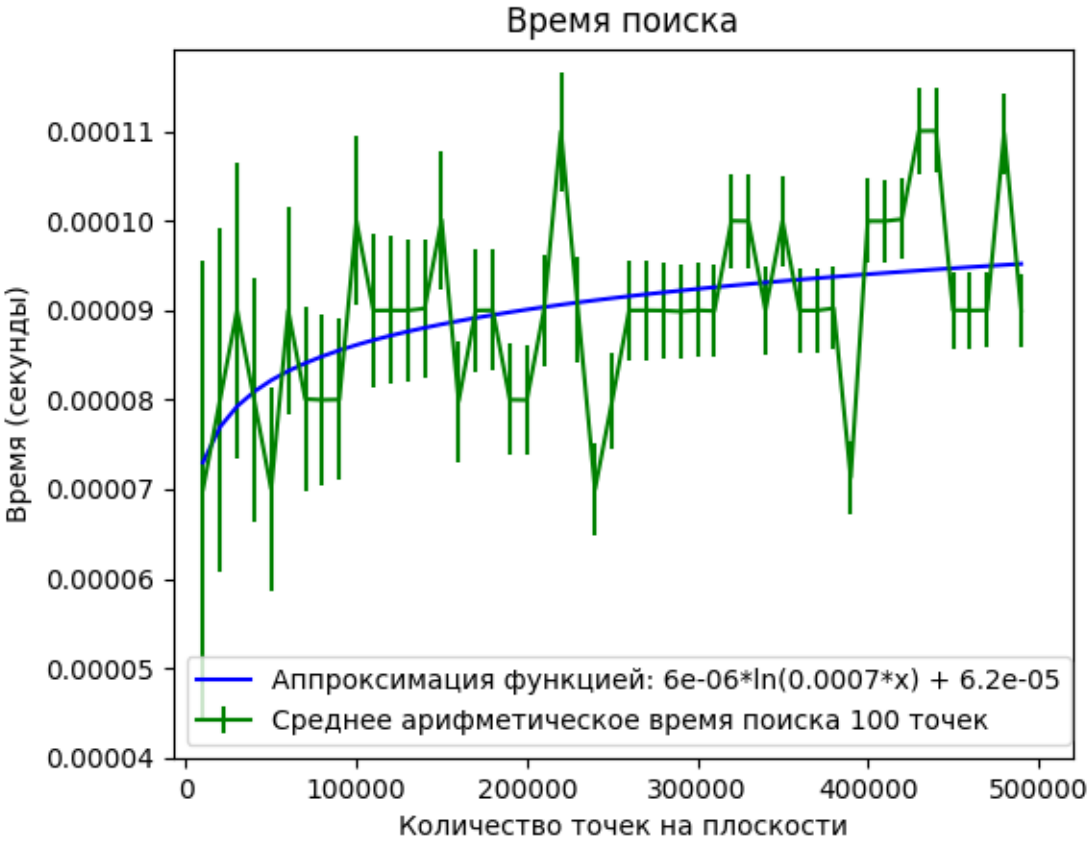
[2] Jon Louis Bentley, Multidimensional Binary Search Trees Used for Associative Searching, 1975 ACM Student Award

[3] Jerome H. Friedman, Raphael Finkel, An Algorithm for Finding Best Matches in Logarithmic Expected Time, ACM Transactions on Mathematical Software, September 1977

[4] James McNames, A Fast Nearest-Neighbor Algorithm Based on a Principal Axis Search Tree, IEEE Transactions on pattern analysis and machine intelligence, vol. 23, no. 9, September 2001

[5] Yoonho Hwang, Bohyung Han, Hee-Kap Ahn, A Fast Nearest Neighbor Search Algorithm by Nonlinear Embedding, POSTECH, Korea

Рис. 5: Время поиска



[6] Alexandr Andoni, Nearest Neighbor Search: the Old, the New, and the Impossible, September 2009

[7] A. Andoni, Nearest Neighbor Search - the Old, the New, and the Impossible, Ph.D. dissertation, Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 2009

[8] G. Shakhnarovich, T. Darrell, and P. Indyk, Nearest Neighbor Methods in Learning and Vision : Theory and Practice, March 2006

[9] N. Bhatia and V. Ashev, Survey of Nearest Neighbor Techniques, International Journal of Computer Science and Information Security, 8(2), 2010, pp. 1-4.

[10] S. Dhanabal and S. Chandramathi, A Review of various k-Nearest Neighbor

Query Processing Techniques, Computer Applications. 31(7), 2011, pp. 14-22.

[11] A. Rajaraman and J. D. Ullman. Mining of Massive Datasets, December 2011