ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 7

«Нахождение кратчайшего пути в взвешенном графе»

Выполнил работу

Воробьев Андрей

Академическая группа №J3110

Принято

Ментор, Вершинин Владислав

Санкт-Петербург

2024

1. Введение

Цель лабораторной работы – научиться реализовывать различные алгоритмы поиска кратчайшего пути в взвешенном и направленном графе.

Задачи:

* Считать граф из файла
* Реализовать хранение графа
* Реализовать алгоритмы поиска в глубину, в ширину, алгоритм Дейкстры и А\*
* Реализовать поиск точки в графе ближайшей к настоящим координатам
* Найти путь кратчайший путь от дома до ВУЗа
* Реализовать возможность поиска пути с несколькими путевыми точками (возможность посещения баров по пути)
* Изучить скорость работы алгоритма

1. Реализация.
   1. . Представление графа.

Граф представляет собой массив указателей на вершины. Которые в свою очередь хранят свои координаты и два массива рёбер, связанных с дочерними и родительскими вершинами. Ребро – указатель на другую вершину и вес ребра.

* 1. Алгоритмы для поиска пути.

DFS (Поиск в глубину): ищет путь, выбирая вершины из стека. Единственный алгоритм, который не ищет кратчайший путь. Изначальная цель этого алгоритма – узнать все вершины компоненты связности. При попытке найти кратчайший путь, не меняя основную идею алгоритма, он совершает слишком много повторных действий, что не позволило отработать алгоритму на графе даже за 40 минут.

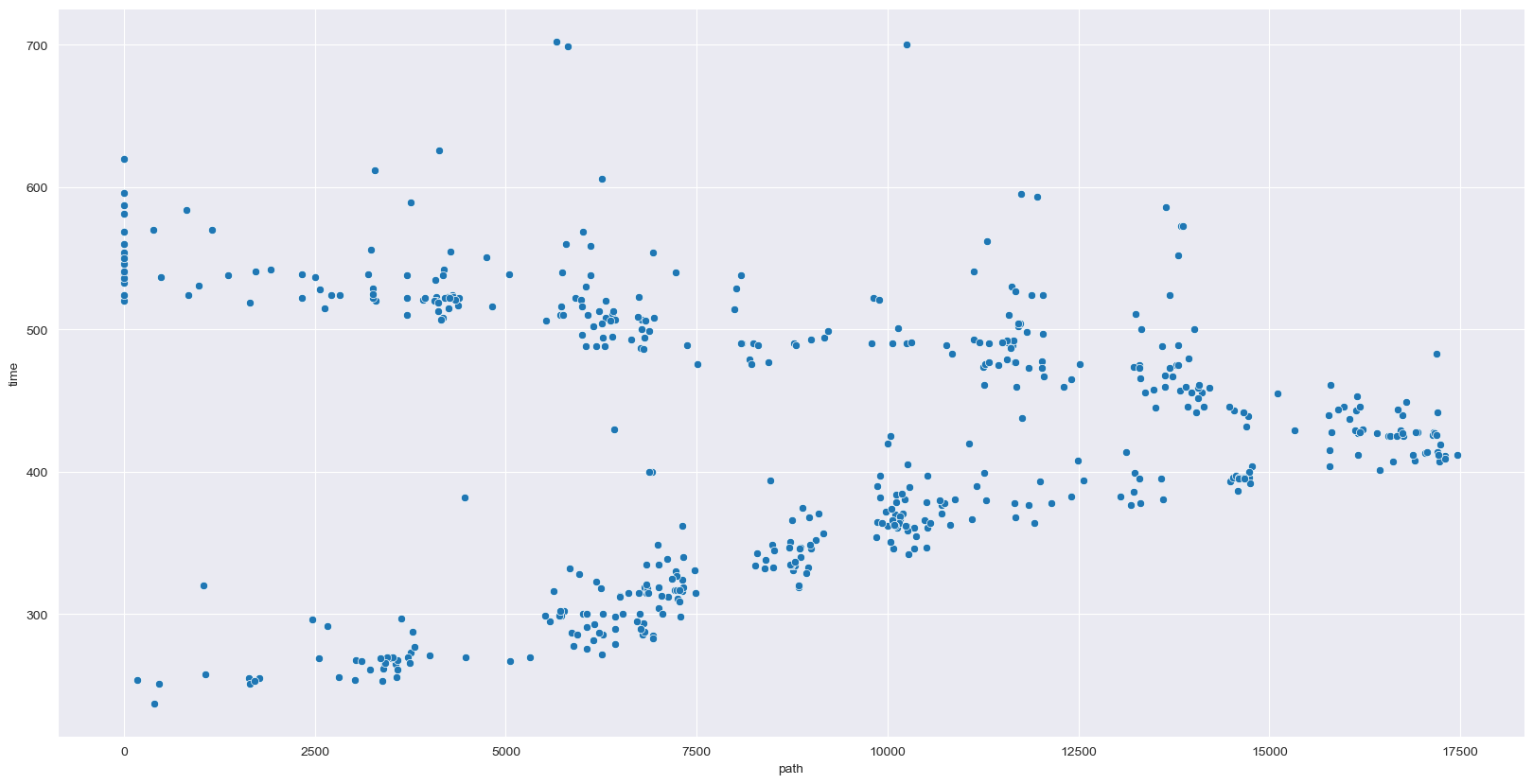
BFS (Поиск в ширину): находит кратчайший путь, выбирая вершины из очереди. Очередь позволяет не совершать слишком много бессмысленных действий, но всё равно не оптимизирована для поиска кратчайшего пути.

Дейкстра: находит кратчайший путь, выбирая вершины из очереди с приоритетом, что позволяет не посещать вершину больше одного раза, хорошо оптимизированный алгоритм. Также даёт гарантию, что вес вершины не будет обновляться далее, поэтому при поиске расстояния до вершины, можно останавливать алгоритм в случае посещения требуемой вершины.

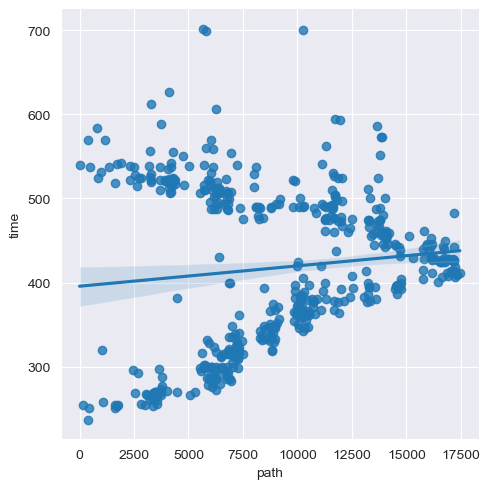
A\*: лучше всего находит кратчайший путь, является модификацией алгоритма Дейкстры, так как в очереди с приоритетом ещё учитывает эвристическую функцию. В нашем случае можно использовать расстояние между точками на плоскости.

1. Экспериментальная часть.

В ходе экспериментальной части получилось выяснить, что граф не сильно связан и некоторые вершины невозможно посетить. Также, как было сказано ранее, алгоритм для нахождения кратчайшего пути с помощью DFS реализовать невозможно, поэтому на Изображении 1 и 2 приведены результаты работы для поиска первого возможного пути. Расстояние 0 означает отсутствие пути.



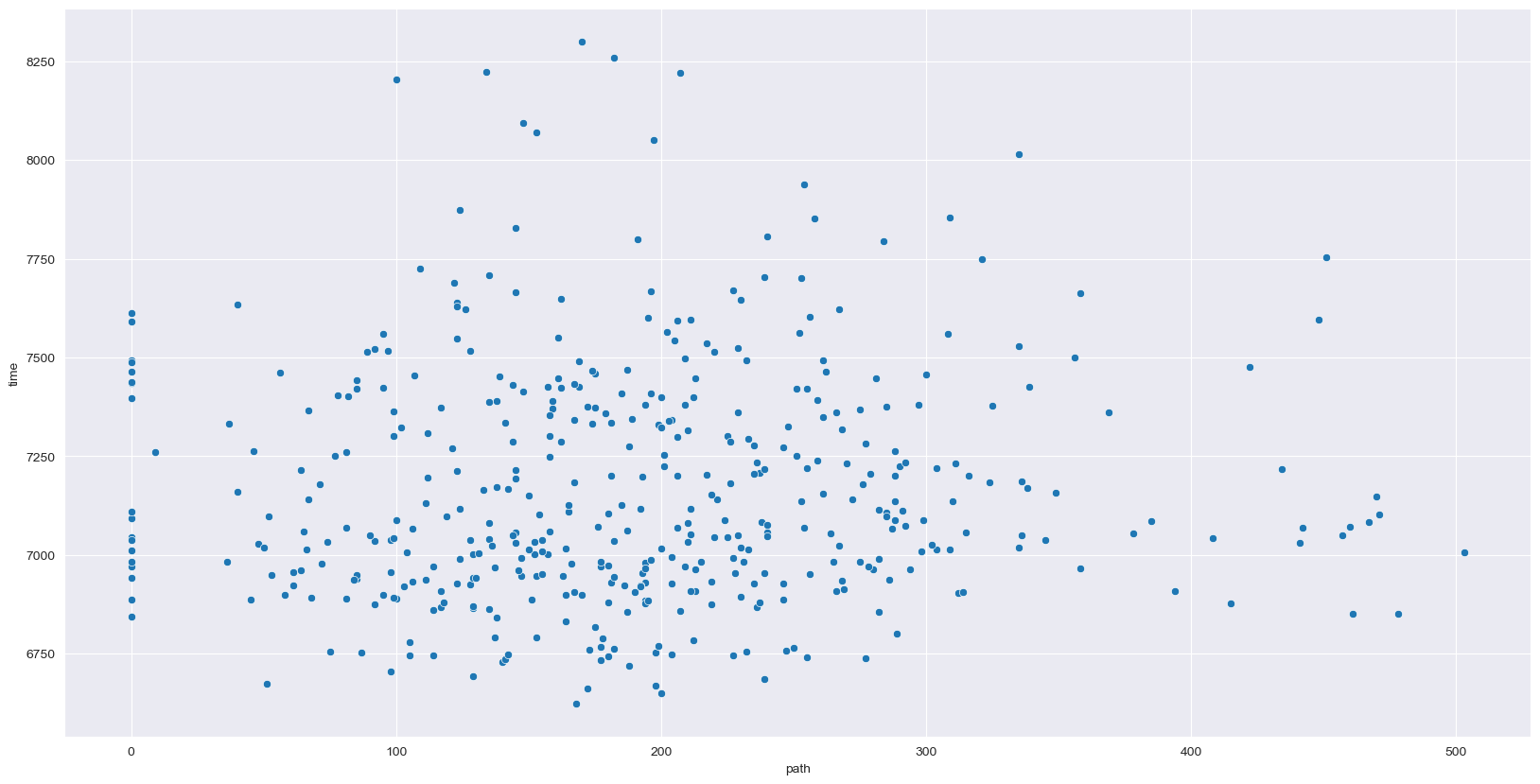
Изображение 1 – scatterplot DFS



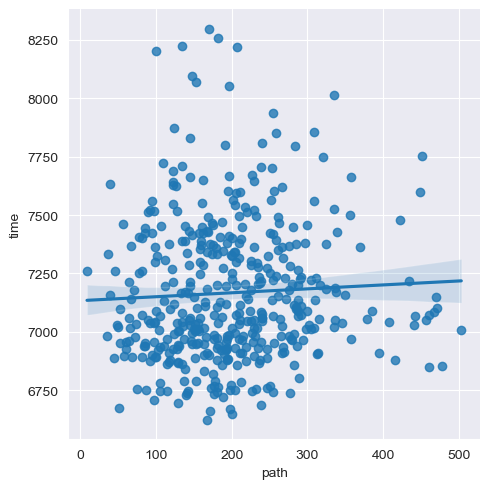
Изображение 2 – lmplot DFS

По графикам видно, что в зависимости от расстояния среднее время поиска пути не изменяется. Также по большой разброс говорит о том, что время работы алгоритма очень нестабильно, так как всё зависит от пути, куда сначала пойдёт поиск.

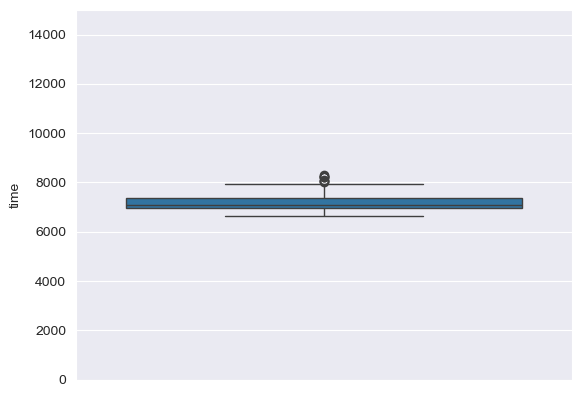
По Изображениям 3, 4 и 5 видно, что алгоритм BFS работает примерно за одно и то же время, так как каждый раз должен обходить весь граф, иногда он может выбирать менее выгодный порядок обхода, но это не значительно.



Изображение 3 – scatterplot BFS

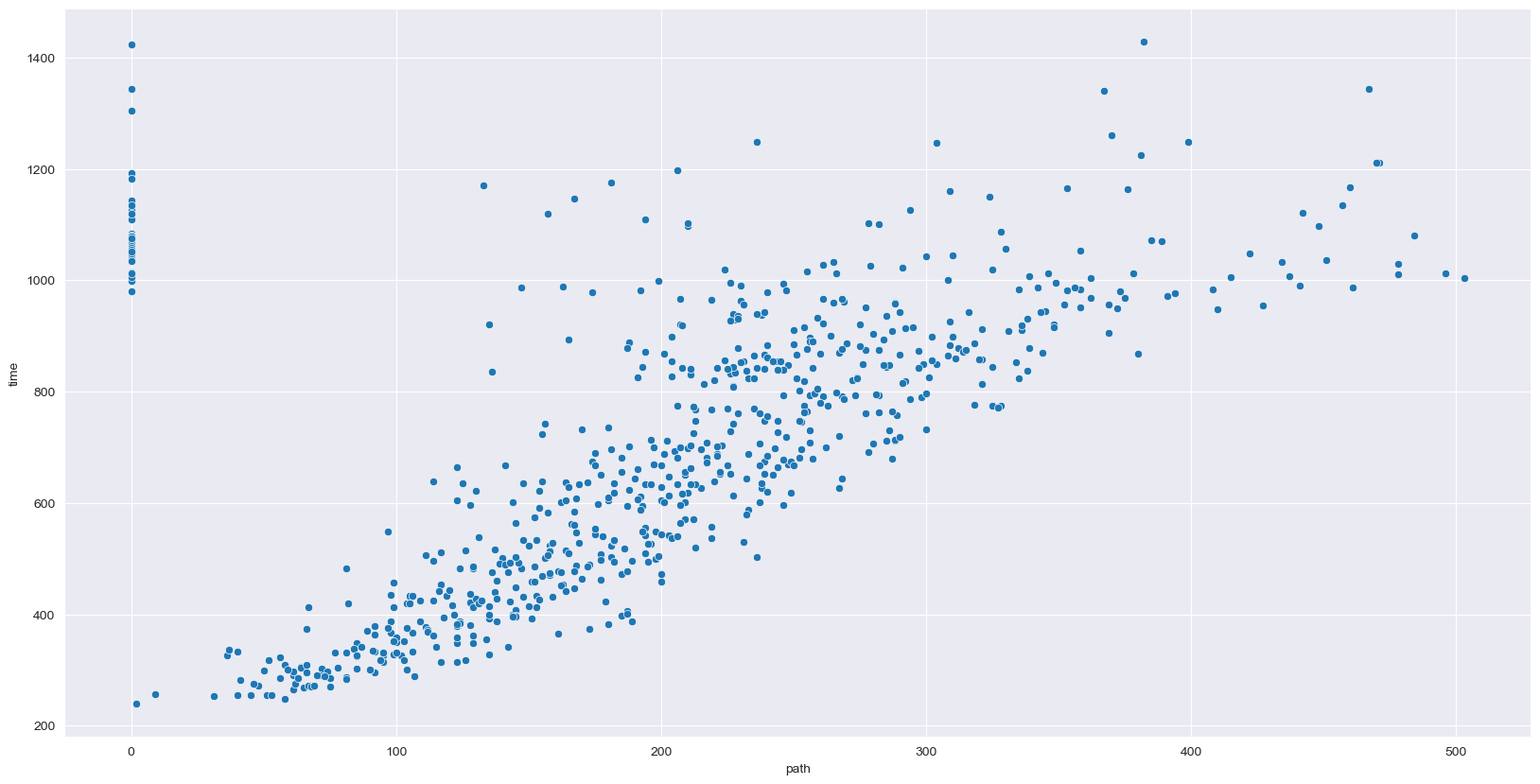


Изображение 4 – lmplot BFS

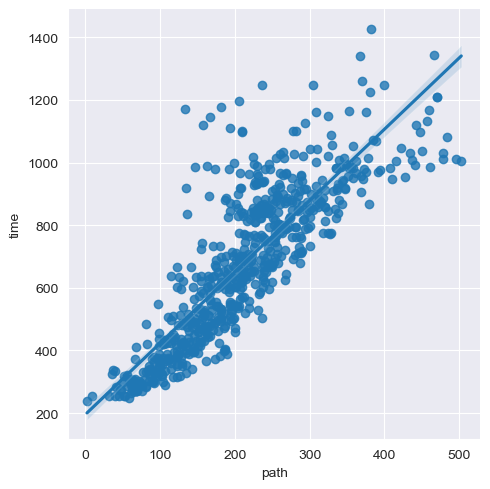


Изображение 5 – boxplot BFS

Алгоритм Дейкстры уже даёт линейную зависимость, так он останавливается при нахождении требуемой точки, то чем короче путь (в смысле, число вершин в нём), тем раньше он завершит работу. Линейную зависимость просто проследить с помощью Изображения 6 и 7.

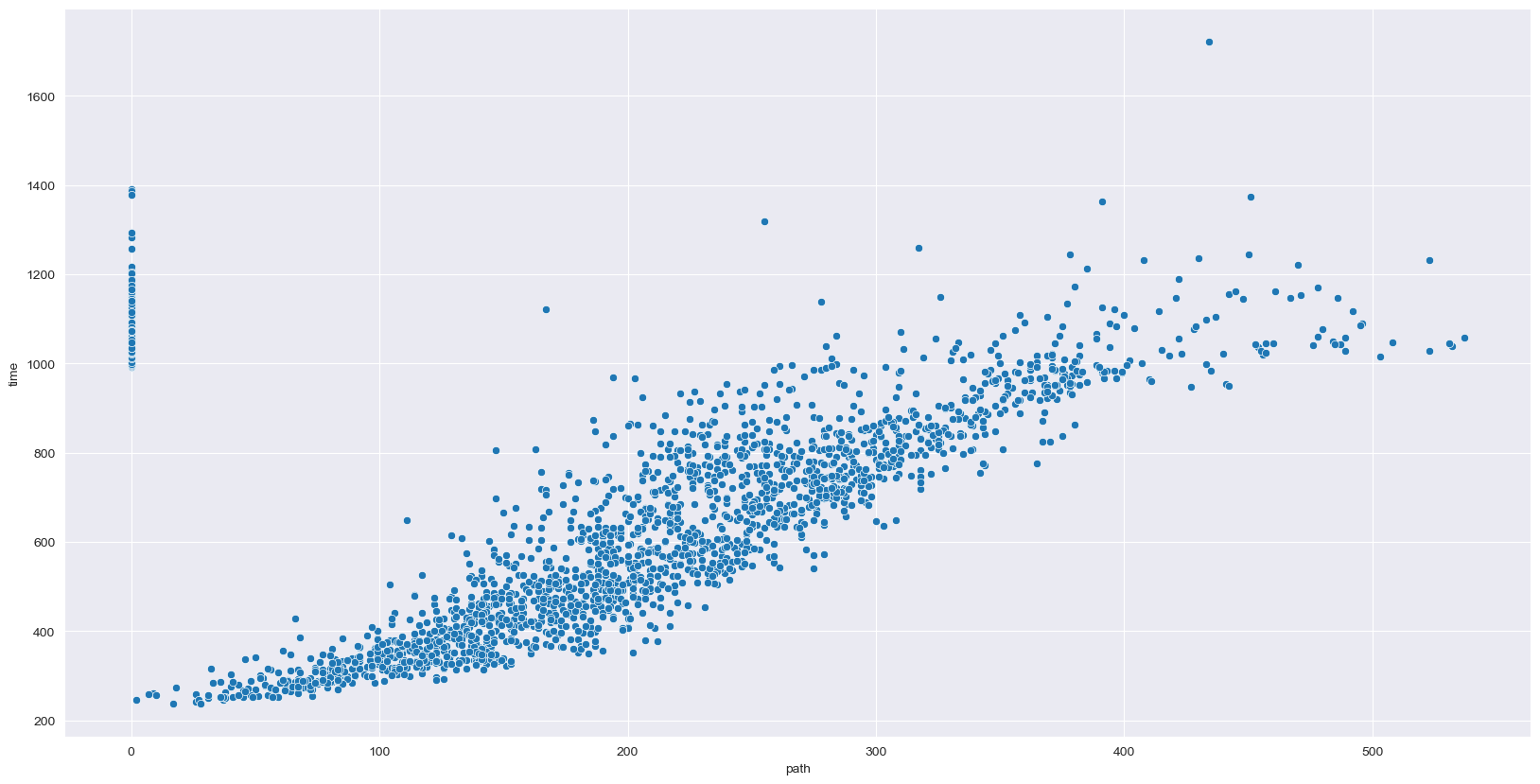


Изображение 6 – scatterplot Дейкстра

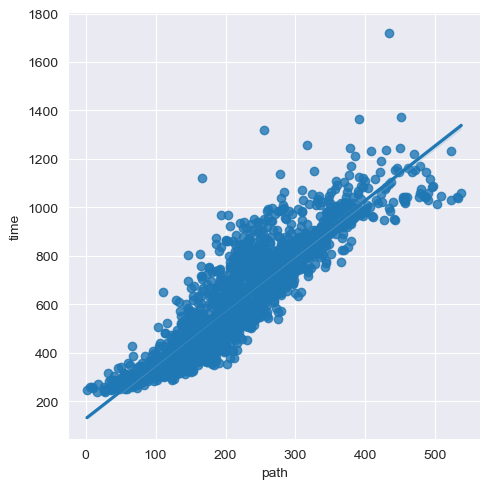


Изображение 7 – lmplot Дейкстра

Алгоритм А\* в нашем случае почти идентичен Дейкстре имея лишь не большую разницу в скорости работы в лучшую сторону. Результаты работы изображены на Изображении 8 и 9.



Изображение 8 – scatterplot А\*



Изображение 9 – lmplot А\*

Заключение.

В ходе выполнения работы мною был реализовано хранение и обработка взвешенного графа, основанного на реальной карте города, а также алгоритмы поиска путей в графе. Самым быстро-стабильным алгоритмом оказался DFS, но он и не выполнял требуемую задачу для поиска кратчайшего пути. BFS оказался самым долгим и стабильным алгоритмом, что очевидно из его нерассчитанности для данной задачи. Алгоритм Дейкстры является вторым по скорости, но уступает лишь немного своей модификации А\*. Алгоритм был реализован на языке C++.

ПРИЛОЖЕНИЕ А

Листинг кода для вывода графиков

import numpy as np  
import matplotlib.pyplot as plt  
import pandas as pd  
import seaborn as sns  
*#%%*f = open('a\_star.txt', 'r')  
lines = f.readlines()  
data = [[int(x)for x in y.split()] for y in lines]  
data = pd.DataFrame(data, columns=['path', 'time'])  
data.head()  
*#%%*plt.figure(figsize=(20, 10))  
sns.scatterplot(x='path', y='time', data=data)  
sns.lmplot(x='path', y='time', data=data[data['path'] != 0])

ПРИЛОЖЕНИЕ Б

Листинг кода программ

#include "7hw.h"

struct Edge {  
 *// ребро графа* Node \*node; *// указатель на дочернюю вершину* double weight; *// вес ребра* Edge(Node \*node, double weight) : node(node), weight(weight) {}  
};  
  
struct Node {  
 *// вершина графа* double latitute; *// широта* double longitute; *// долгота* std::vector<Edge> children; *// рёбра в дочерние вершины* std::vector<Edge> parents; *// рёбра в родительские вершины* Node(const double latitute, const double longitute) : latitute(latitute), longitute(longitute) {}  
};  
  
struct Graph {  
 *// граф* std::vector<Node \*> nodes; *// вершины графа* Node \*get\_node\_ptr(const double latitute, const double longitute,  
 std::unordered\_map<std::string, Node \*> &nodes\_map) {  
 *// получает указатель на вершину по координатам  
 // использует таблицу для хранения указателей на вершины* std::string key = std::to\_string(latitute) + "," + std::to\_string(longitute);  
  
 if (nodes\_map.find(key) == nodes\_map.end()) {  
 *// если вершина ещё не была создана  
 // добавляем её в граф и в таблицу* Node \*node\_ptr = new Node(latitute, longitute);  
 nodes.push\_back(node\_ptr);  
 nodes\_map[key] = node\_ptr;  
 }  
  
 return nodes\_map[key]; *// возвращаем указатель на вершину* }  
  
 std::pair<double, double> read\_coords(const std::string str) {  
 *// считывает родительскую вершину* size\_t comma = str.find(',');  
 double longitute = std::stod(str.substr(0, comma));  
 double latitute = std::stod(str.substr(comma + 1));  
 return {latitute, longitute};  
 }  
  
 std::pair<std::pair<double, double>, double> read\_edge(const std::string str) {  
 *// считывает ребро* size\_t comma1 = str.find(',');  
 size\_t comma2 = str.find(',', comma1 + 1);  
 double longitute = std::stod(str.substr(0, comma1));  
 double latitute = std::stod(str.substr(comma1 + 1, comma2 - comma1 - 1));  
 double weight = std::stod(str.substr(comma2 + 1));  
 return {{latitute, longitute}, weight};  
 }  
  
 Graph(const std::string filepath) {  
 *// считывает граф из файла* std::ifstream file(filepath);  
 std::string line;  
  
 std::unordered\_map<std::string, Node \*> nodes\_map;  
 *// таблица для хранения указателей на вершины* while (file >> line) {  
 *// считываем строки файла* size\_t first\_sep = line.find(':');  
 *// первый разделитель - координаты вершины* std::string node0\_str = line.substr(0, first\_sep);  
 auto coords = read\_coords(node0\_str);  
 Node \*node0\_ptr = get\_node\_ptr(coords.first, coords.second, nodes\_map);  
 *// получаем указатель на родительскую вершину* while (line.find(';', first\_sep + 1) != std::string::*npos*) {  
 *// пока есть дочерние вершины* size\_t second\_sep = line.find(';', first\_sep + 1);  
 *// второй разделитель - координаты дочерней вершины* std::string node\_str = line.substr(first\_sep + 1, second\_sep - first\_sep - 1);  
 auto edge1 = read\_edge(node\_str);  
 Node \*node1\_ptr = get\_node\_ptr(edge1.first.first, edge1.first.second, nodes\_map);  
 *// получаем указатель на дочернюю вершину* node0\_ptr->children.push\_back({node1\_ptr, edge1.second});  
 node1\_ptr->parents.push\_back({node0\_ptr, edge1.second});  
 *// добавляем ребро в граф* first\_sep = second\_sep; *// сдвигаем разделитель* }  
 }  
 }  
};  
  
struct Pathway {  
 *// путь* std::vector<Node \*> path; *// вершины пути* double distance; *// длина пути* Pathway() : path(std::vector<Node \*>()), distance(1e9) {}  
 Pathway(std::vector<Node \*> path, double distance) : path(path), distance(distance) {}  
};  
  
struct Algorithm\_result {  
 *// результат работы алгоритма* Pathway pathway; *// путь* double time; *// время работы алгоритма* Algorithm\_result() : pathway(Pathway()), time(0) {}  
 Algorithm\_result(std::vector<Node \*> path, double distance, double time) : pathway(path, distance), time(time) {}  
};  
  
struct My\_way {  
 *// нахождение кратчайшего пути  
 // в графе из файла  
 // с помощью разных алгоритмов* Node \*start; *// начальная вершина* Node \*destination; *// конечная вершина* Graph \*graph; *// граф* std::vector<Node \*> extra\_way\_points; *// дополнительные вершины* Algorithm\_result algorithm\_result; *// результат работы алгоритма* Node \*closest\_node(const double latitute, const double longitute) {  
 *// поиск ближайшей вершины в графе к заданным координатам* Node \*closest = nullptr;  
 double min\_distance = 1e9;  
 for (auto node: graph->nodes) {  
 double distance = (node->latitute - latitute) \* (node->latitute - latitute) +  
 (node->longitute - longitute) \* (node->longitute - longitute);  
 if (distance < min\_distance) {  
 min\_distance = distance;  
 closest = node;  
 }  
 }  
 return closest;  
 }  
  
 My\_way(const double latitute\_start, const double longitute\_start, const double latitute\_destination,  
 const double longitute\_destination, const std::string filename) {  
 *// инициализация* graph = new Graph(filename);  
 start = closest\_node(latitute\_start, longitute\_start);  
 destination = closest\_node(latitute\_destination, longitute\_destination);  
 }  
  
 void add\_extra\_point(const double latitute, const double longitute) {  
 *// добавление дополнительной вершины* extra\_way\_points.push\_back(closest\_node(latitute, longitute));  
 }  
  
 std::vector<Node \*> get\_path(Node \*s, Node \*e, std::unordered\_map<Node \*, double> &dist) {  
 *// восстановление пути* std::vector<Node \*> path;  
 Node \*cur = e;  
  
 while (cur != s) {  
 path.push\_back(cur);  
 for (auto edge: cur->parents)  
 if (dist[cur] == dist[edge.node] + edge.weight) {  
 cur = edge.node;  
 break;  
 }  
 }  
  
 path.push\_back(s);  
 std::reverse(path.begin(), path.end());  
 return path;  
 }  
  
 Pathway dfs(Node \*s, Node \*e) {  
 *// поиск в глубину  
 // находит не наименьший путь в взвешенном графе  
 // а просто один из путей  
 // расстояние - количество вершин в пути* std::unordered\_map<Node \*, double> dist;  
 for (auto node: graph->nodes)  
 dist[node] = 1e9;  
 dist[s] = 0;  
 *// расстояние до вершин* std::stack<Node \*> node\_stack;  
 node\_stack.push(s);  
 *// стек необработанных вершин* while (!node\_stack.empty()) {  
 *// пока есть ещё вершины, для которых мы обновили расстояние* Node \*node = node\_stack.top();  
 node\_stack.pop();  
 *// берём верхнюю вершину из стека* if (node == e)  
 break;  
 *// если дошли до вершины, то до неё есть путь* for (auto edge: node->children)  
 if (dist[edge.node] == 1e9) {  
 dist[edge.node] = dist[node] + 1;  
 node\_stack.push(edge.node);  
 } *// если у дочерней вершины обновилось расстояние, добавляем её в стек* }  
  
 if (dist[e] == 1e9)  
 return {}; *// если нет пути  
  
 // восстановление пути* std::vector<Node \*> path;  
 Node \*cur = e;  
  
 while (cur != s) {  
 path.push\_back(cur);  
 for (auto edge: cur->parents)  
 if (dist[cur] == dist[edge.node] + 1) {  
 cur = edge.node;  
 break;  
 }  
 }  
  
 path.push\_back(s);  
 std::reverse(path.begin(), path.end());  
 return {path, (double)path.size()};  
 }  
  
 Pathway bfs(Node \*s, Node \*e) {  
 *// поиск в ширину* std::unordered\_map<Node \*, double> dist;  
 for (auto node: graph->nodes)  
 dist[node] = 1e9;  
 dist[s] = 0;  
 *// расстояние до вершин* std::queue<Node \*> node\_queue;  
 node\_queue.push(s);  
 *// очередь необработанных вершин* while (!node\_queue.empty()) {  
 *// пока есть ещё вершины, для которых мы обновили расстояние* Node \*node = node\_queue.front();  
 node\_queue.pop();  
 *// берём верхнюю вершину из очереди* for (auto edge: node->children)  
 if (dist[edge.node] > dist[node] + edge.weight) {  
 dist[edge.node] = dist[node] + edge.weight;  
 node\_queue.push(edge.node);  
 } *// если у дочерней вершины обновилось расстояние, добавляем её в очередь* }  
  
 if (dist[e] == 1e9)  
 return {}; *// если нет пути* return {get\_path(s, e, dist), dist[e]};  
 }  
  
 Pathway dijkstra(Node \*s, Node \*e) {  
 *// алгоритм Дейкстры* std::unordered\_map<Node \*, double> dist;  
 for (auto node: graph->nodes)  
 dist[node] = 1e9;  
 dist[s] = 0;  
 *// расстояние до вершин* auto cmp = [](std::pair<double, Node \*> left, std::pair<double, Node \*> right) {  
 return left.first > right.first;  
 }; *// компаратор для очереди с приоритетом, чтобы вершины были отсортированы по возрастанию расстояния* std::priority\_queue<std::pair<double, Node \*>, std::vector<std::pair<double, Node \*>>, decltype(cmp)>  
 node\_queue(cmp);  
 node\_queue.push({0, s});  
 *// очередь с приоритетом необработанных вершин* while (!node\_queue.empty()) {  
 *// пока есть ещё вершины, для которых мы обновили расстояние* Node \*node = node\_queue.top().second;  
 node\_queue.pop();  
 *// берём верхнюю вершину из очереди* if (node == e)  
 break;  
 *// если дошли до конечной вершины, то гарантированно нашли минимальное расстояние до неё* for (auto edge: node->children)  
 if (dist[edge.node] > dist[node] + edge.weight) {  
 dist[edge.node] = dist[node] + edge.weight;  
 node\_queue.push({dist[edge.node], edge.node});  
 } *// если у дочерней вершины обновилось расстояние, добавляем её в очередь* }  
  
 if (dist[e] == 1e9)  
 return {}; *// если нет пути* return {get\_path(s, e, dist), dist[e]};  
 }  
  
 double heuristic(Node \*a, Node \*b) {  
 *// расстояние между вершинами* return (a->latitute - b->latitute) \* (a->latitute - b->latitute) +  
 (a->longitute - b->longitute) \* (a->longitute - b->longitute);  
 }  
  
 Pathway a\_star(Node \*s, Node \*e) {  
 *// алгоритм A\** std::unordered\_map<Node \*, double> dist;  
 for (auto node: graph->nodes)  
 dist[node] = 1e9;  
 dist[s] = 0;  
 *// расстояние до вершин* auto cmp = [](std::pair<double, Node \*> left, std::pair<double, Node \*> right) {  
 return left.first > right.first;  
 }; *// компаратор для очереди с приоритетом, чтобы вершины были отсортированы по возрастанию расстояния* std::priority\_queue<std::pair<double, Node \*>, std::vector<std::pair<double, Node \*>>, decltype(cmp)>  
 node\_queue(cmp);  
 node\_queue.push({0, s});  
 *// очередь с приоритетом необработанных вершин* while (!node\_queue.empty()) {  
 *// пока есть ещё вершины, для которых мы обновили расстояние* Node \*node = node\_queue.top().second;  
 node\_queue.pop();  
 *// берём верхнюю вершину из очереди* if (node == e)  
 break;  
 *// если дошли до конечной вершины, то гарантированно нашли минимальное расстояние до неё* for (auto edge: node->children)  
 if (dist[edge.node] > dist[node] + edge.weight) {  
 dist[edge.node] = dist[node] + edge.weight;  
 node\_queue.push({dist[edge.node] + heuristic(edge.node, e), edge.node});  
 } *// если у дочерней вершины обновилось расстояние, добавляем её в очередь  
 // суммируем расстояние и эвристику для приоритета* }  
  
 if (dist[e] == 1e9)  
 return {}; *// если нет пути* return {get\_path(s, e, dist), dist[e]};  
 }  
  
 Pathway combinatory\_pathway(std::set<size\_t> points\_left, size\_t current\_node,  
 std::vector<Pathway> &path\_from\_start, std::vector<Pathway> &path\_between\_points,  
 std::vector<Pathway> &path\_to\_destination) {  
 *// рекурсивно перебираем все возможные пути через дополнительные вершины* if (points\_left.empty())  
 return path\_to\_destination[current\_node - 1];  
 *// если не осталось вершин, то остался путь до конечной вершины* Pathway best\_pathway;  
 size\_t best\_point = 0;  
  
 if (current\_node == 0) {  
 *// если это первая вершина, то ищем путь от начальной вершины до дополнительной* for (auto point: points\_left) {  
 std::set<size\_t> new\_points\_left = points\_left;  
 new\_points\_left.erase(point);  
  
 Pathway current = combinatory\_pathway(new\_points\_left, point, path\_from\_start, path\_between\_points,  
 path\_to\_destination);  
 if (current.distance + path\_from\_start[point - 1].distance <  
 best\_pathway.distance + path\_from\_start[best\_point - 1].distance) {  
 best\_pathway = current;  
 best\_point = point;  
 }  
 }  
 Pathway result = path\_from\_start[best\_point - 1];  
 result.path.insert(result.path.end(), best\_pathway.path.begin(), best\_pathway.path.end());  
 result.distance += best\_pathway.distance;  
 return result;  
 }  
  
 *// иначе перебираем дополнительные вершины* for (auto point: points\_left) {  
 std::set<size\_t> new\_points\_left = points\_left;  
 new\_points\_left.erase(point);  
  
 Pathway current = combinatory\_pathway(new\_points\_left, point, path\_from\_start, path\_between\_points,  
 path\_to\_destination);  
  
 if (current.distance +  
 path\_between\_points[(current\_node - 1) \* (extra\_way\_points.size() - 1) + point - 1].distance <  
 best\_pathway.distance +  
 path\_between\_points[(current\_node - 1) \* (extra\_way\_points.size() - 1) + best\_point - 1]  
 .distance) {  
 best\_pathway = current;  
 best\_point = point;  
 }  
 }  
 Pathway result = path\_between\_points[(current\_node - 1) \* (extra\_way\_points.size() - 1) + best\_point - 1];  
 result.path.insert(result.path.end(), best\_pathway.path.begin(), best\_pathway.path.end());  
 result.distance += best\_pathway.distance;  
 return result;  
 }  
  
 template<typename Function>  
 Pathway find\_best\_path(Function algorithm) {  
 *// находим кратчайший путь* if (extra\_way\_points.empty())  
 return (this->\*algorithm)(start, destination);  
 *// если нет дополнительных вершин, то ищем путь от начальной вершины до конечной* std::vector<Pathway> path\_from\_start;  
 for (auto point: extra\_way\_points)  
 path\_from\_start.push\_back((this->\*algorithm)(start, point));  
 *// ищем пути от начальной вершины до дополнительных* std::vector<Pathway> path\_between\_points;  
 for (auto point: extra\_way\_points)  
 for (auto another\_point: extra\_way\_points)  
 if (point != another\_point)  
 path\_between\_points.push\_back((this->\*algorithm)(point, another\_point));  
 *// ищем пути между дополнительными вершинами* std::vector<Pathway> path\_to\_destination;  
 for (auto point: extra\_way\_points)  
 path\_to\_destination.push\_back((this->\*algorithm)(point, destination));  
 *// ищем пути от дополнительных вершин до конечной* std::unordered\_map<Node \*, size\_t> point\_index;  
 for (size\_t i = 0; i < extra\_way\_points.size(); ++i)  
 point\_index[extra\_way\_points[i]] = i;  
 *// индексы вершин* std::set<size\_t> points\_left;  
 for (size\_t i = 1; i < extra\_way\_points.size() + 1; ++i)  
 points\_left.insert(i);  
  
 return combinatory\_pathway(points\_left, 0, path\_from\_start, path\_between\_points, path\_to\_destination);  
 }  
  
 template<typename Function>  
 void calculate\_path(Function algorithm) {  
 *// нахождение пути и замеры времени  
 // используется алгоритм algorithm* const size\_t start = clock();  
 auto [path, dist] = find\_best\_path(algorithm);  
 const size\_t end = clock();  
 algorithm\_result = Algorithm\_result(path, dist, (end - start));  
 }  
  
 void print\_pathway() {  
 std::cout << "Path: ";  
 for (auto node: algorithm\_result.pathway.path)  
 if (node == destination)  
 std::cout << node->latitute << " " << node->longitute << std::endl;  
 else  
 std::cout << node->latitute << " " << node->longitute << " -> " << std::endl;  
 std::cout << std::endl;  
 }  
  
 void print\_distance() {  
 std::cout << "Distance: " << algorithm\_result.pathway.distance << std::endl;  
 std::cout << std::endl;  
 }  
  
 void print\_time() {  
 std::cout << "Time: " << algorithm\_result.time << std::endl;  
 std::cout << std::endl;  
 }  
  
 template<typename Function>  
 void algorithm\_analis(Function algorithm) {  
 for (int i = 0; i < graph->nodes.size(); i+=29) {  
 for (int j = i + 1; j < graph->nodes.size(); j+=121) {  
 start = graph->nodes[i];  
 destination = graph->nodes[j];  
 calculate\_path(algorithm);  
 std::cout << algorithm\_result.pathway.path.size() << ' ' << algorithm\_result.time << std::endl;  
  
 }  
 }  
 }  
};  
  
  
int main() {  
 *// 60.000860, 30.368114 - дом* double home\_latitute = 60.000860;  
 double home\_longitute = 30.368114;  
 *// 59.944168, 30.295489 - итмо биржа* double work\_latitute = 59.944168;  
 double work\_longitute = 30.295489;  
  
  
 My\_way my\_way(home\_latitute, home\_longitute, work\_latitute, work\_longitute, "7hw/spb\_graph.txt");  
 std::cout << my\_way.start->latitute << " " << my\_way.start->longitute << std::endl;  
 std::cout << my\_way.destination->latitute << " " << my\_way.destination->longitute << std::endl;  
  
 *// 60.004211, 30.299131 - сити молл  
 // my\_way.add\_extra\_point(60.004211, 30.299131);  
 // std::cout << my\_way.extra\_way\_points.back()->latitute << " " << my\_way.extra\_way\_points.back()->longitute  
 // << std::endl;  
  
 // 59.908491, 30.513576 - кудрово  
 // my\_way.add\_extra\_point(59.908491, 30.513576);  
 // std::cout << my\_way.extra\_way\_points.back()->latitute << " " << my\_way.extra\_way\_points.back()->longitute  
 // << std::endl;  
  
 // 60.021335, 30.309469 - психбольница 3  
 // my\_way.add\_extra\_point(60.021335, 30.309469);  
 // std::cout << my\_way.extra\_way\_points.back()->latitute << " " << my\_way.extra\_way\_points.back()->longitute <<  
 // std::endl;  
  
 // my\_way.calculate\_path(&My\_way::dfs);  
 // my\_way.print\_distance();  
 // my\_way.print\_time();  
 // my\_way.calculate\_path(&My\_way::bfs);  
 // my\_way.print\_distance();  
 // my\_way.print\_time();  
 // my\_way.calculate\_path(&My\_way::dijkstra);  
 // my\_way.print\_distance();  
 // my\_way.print\_time();  
 // my\_way.calculate\_path(&My\_way::a\_star);  
 // my\_way.print\_distance();  
 // my\_way.print\_time();* my\_way.algorithm\_analis(&My\_way::bfs);  
}