Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Курсовой проект по курсу «Дискретный анализ»

Тема: «Эвристический поиск в графе»

Студент: М. А. Инютин
Преподаватель: С. А. Сорокин
Группа: М8О-207Б-19

Дата:
Оценка:
Подпись:

Москва, 2021

Эвристический поиск в графе

Задача: Реализуйте систему для поиска пути в графе дорог с использованием эвристических алгоритмов.

`./prog preprocess --nodes <nodes file> --edges <edges file> --output <preprocessed graph>`

Ключ	Значение
--nodes	входной файл с перекрёстками
--edges	входной файл с дорогами
--output	выходной файл с графом

`./prog search --graph <preprocessed graph> --input <input file> --output <output file> [--full-output]`

Ключ	Значение
--graph	входной файл с графом
--input	входной файл с запросами
--output	выходной файл с ответами на запросы
--full-output	переключение формата выходного файла на подробный

Выходной файл:

Если опция --full-output не указана: на каждый запрос в отдельной строке выводится длина кратчайшего пути между заданными вершинами с относительной погрешностью не более 10^{-6}.

Если опция --full-output указана: на каждый запрос выводится отдельная строка, с длиной кратчайшего пути между заданными вершинами с относительной погрешностью не более 10^{-6}, а затем сам путь в формате как в файле рёбер.

Расстояние между точками следует вычислять как расстояние между точками на сфере с радиусом 6371 км, если пути между точками нет, вывести -1 и длину пути в вершинах 0.

1 Описание

Требуется реализовать алгоритм поиска кратчайшего пути во взвешенном графе.

С этой задачей отлично справляется алгоритм Дейкстры [1], но в силу особенностей задачи так же применим алгоритм поиска $A*$ [2].

Алгоритм Дейкстры хранит для каждой вершины длину кратчайшего пути до неё и на каждом шаге ищет вершину с минимальной такой величиной. Алгоритм обходит все смежные вершины и запоминает, что текущая вершина с минимальным кратчайшем путём была посещена.

Изначально пути до всех вершины равны бесконечности, а в старте значение равно нулю. Алгоритм имеет сложность $O(n^2 + m)$ в простой реализации и $O(n \cdot \log n + m)$ в реализации с использованием двоичной кучи.

Алгоритм поиска $A*$ работает похожим образом, но для выбора вершины на каждом шаге вычисляетя функцию $f(v) = g(v) + h(v)$, где $g(v)$ — кратчайший путь от стартовой вершины до v, $h(v)$ — эвристическая функция, которая вычисляет приближённое значение кратчайшего пути по финиша.

В моём случае функция $h(v)$ — расстояние между географическими координатами вершины v и финишем.

Как описано в [3], алгоритм имеет временную сложность $O(|E|)$ и пространственную $O(|V|)$, где V — множество вершин в графе, а E — множество дуг графа.

2 Исходный код

Файл *misc.hpp* содержит вспомогательные классы и функции для препроцессинга (сортировка вершин) и поиска (сравнение вершин по эвристике).

```
#ifndef MISC_HPP
#define MISC_HPP

#include <cmath>
#include <cstdio>
#include <vector>

const double STOP_DATA = 11235813;
const size_t GEODATA_SIZE = sizeof(uint32_t) + 2 * sizeof(double);

struct node_t {
    uint32_t id;
    double lon;
    double lat;
};

bool operator < (const node_t & lhs, const node_t & rhs) {
    return lhs.id < rhs.id;
}

const double SPHERE_RADIUS = 6371e3;
const double MAX_ANGLE = 180.0;
const double PI = 3.14159265358979323846264338327950;
const double EPS = 1e-6;

double radians(double angle) {
    return angle * PI / MAX_ANGLE;
}

double geo_dist(const node_t & lhs, const node_t & rhs) {
    double phi_a = radians(lhs.lon);
    double phi_b = radians(rhs.lon);
    double delta = radians(lhs.lat - rhs.lat);
    double cos_d = std::sin(phi_a) * std::sin(phi_b) + std::cos(phi_a) * std::cos(phi_b
        ) * std::cos(delta);
    double d = std::acos(cos_d);
    return std::isnan(d) ? 0 : SPHERE_RADIUS * d;
}

struct edge_t {
    uint32_t id_from;
    uint32_t id_to;
};

```

```
44 bool operator < (const edge_t & lhs, const edge_t & rhs) {
45     if (lhs.id_from != rhs.id_from) {
46         return lhs.id_from < rhs.id_from;
47     } else {
48         return lhs.id_to < rhs.id_to;
49     }
50 }
51
52 struct euristic_t {
53     uint32_t id;
54     double distance;
55     double path;
56
57     euristic_t(const uint32_t & u_id, const node_t & u, const node_t & v, const double
           & path_to) {
58         id = u_id;
59         distance = geo_dist(u, v);
60         path = path_to;
61     }
62 };
63
64 bool operator < (const euristic_t & lhs, const euristic_t & rhs) {
65     if (std::abs((lhs.path + lhs.distance) - (rhs.path + rhs.distance)) > EPS) {
66         return lhs.path + lhs.distance > rhs.path + rhs.distance;
67     } else if (std::abs(lhs.path - rhs.path) > EPS) {
68         return lhs.path > rhs.path;
69     } else {
70         return lhs.id < rhs.id;
71     }
72 }
73
74 #endif /* MISC_HPP */
```

Программа записывает граф в компактном бинарном представлении. В начале файла расположены идентификаторы вершин и их географические координаты. Затем следуют рёбра графа в виде разреженных списков смежности и для каждой вершины место, где начинается список смежности для этой вершины.

Для более быстрого поиска вершины сортируются по идентификаторам и записываются в файл в возрастающем порядке.

```
#ifndef GRAPH_HPP
#define GRAPH_HPP

#include <algorithm>
#include <exception>

#include "misc.hpp"

uint32_t find_node(const std::vector<node_t> & nodes_data, const uint32_t & node_id) {
    int64_t l = -1;
    int64_t r = nodes_data.size();
    while (l + 1 < r) {
        int64_t m = (l + r) / 2;
        if (nodes_data[m].id < node_id) {
            l = m;
        } else {
            r = m;
        }
    }
    return r;
}

void read_nodes(FILE* nodes, std::vector<node_t> & nodes_data) {
    uint32_t cur_vertex = 0;
    double cur_vertex_lat = 0;
    double cur_vertex_lon = 0;
    while (fscanf(nodes, "%u%lf%lf", &cur_vertex, &cur_vertex_lat, &cur_vertex_lon) >
        0) {
        nodes_data.push_back({cur_vertex, cur_vertex_lat, cur_vertex_lon});
    }
    std::sort(nodes_data.begin(), nodes_data.end());
}

void read_edges(FILE* edges, std::vector<edge_t> & edges_data) {
    uint32_t road_size = 0;
    while (fscanf(edges, "%u", &road_size) > 0) {
        uint32_t u = 0;
        fscanf(edges, "%u", &u);
        uint32_t v = 0;
        for (uint32_t i = 1; i < road_size; ++i) {
            fscanf(edges, "%u", &v);
            edges_data.push_back({u, v});
```

```
42              edges_data.push_back({v, u});
43              u = v;
44          }
45      }
46      std::sort(edges_data.begin(), edges_data.end());
47  }
48
49  void write_geo_data(FILE* output, const std::vector<node_t> & nodes_data) {
50      for (size_t i = 0; i < nodes_data.size(); ++i) {
51          fwrite(&nodes_data[i].id, sizeof(uint32_t), 1, output);
52          fwrite(&nodes_data[i].lon, sizeof(double), 1, output);
53          fwrite(&nodes_data[i].lat, sizeof(double), 1, output);
54      }
55      fwrite(&nodes_data.back().id, sizeof(uint32_t), 1, output);
56      fwrite(&STOP_DATA, sizeof(double), 1, output);
57      fwrite(&STOP_DATA, sizeof(double), 1, output);
58  }
59
60  void write_edges(FILE* output, const std::vector<edge_t> & edges_data) {
61      for (size_t i = 0; i < edges_data.size(); ++i) {
62          uint32_t u = edges_data[i].id_from;
63          uint32_t v = edges_data[i].id_to;
64          fwrite(&v, sizeof(uint32_t), 1, output);
65          size_t j = i + 1;
66          while (j < edges_data.size() and edges_data[j].id_from == u) {
67              v = edges_data[j].id_to;
68              fwrite(&v, sizeof(uint32_t), 1, output);
69              ++j;
70          }
71          i = j - 1;
72      }
73      uint32_t stop_index = 0;
74      fwrite(&stop_index, sizeof(uint32_t), 1, output);
75  }
76
77  void write_offsets(FILE* output, const std::vector<node_t> & nodes_data, const std::
        vector<edge_t> & edges_data) {
78      uint32_t j = 0;
79      for (size_t i = 0; i < nodes_data.size(); ++i) {
80          while (j < edges_data.size() and find_node(nodes_data, edges_data[j].id_from) <
                i) {
81              ++j;
82          }
83          fwrite(&j, sizeof(uint32_t), 1, output);
84      }
85  }
86
87  void preprocess_graph(char* nodes_file, char* edges_file, char* output_file) {
88      FILE* nodes = fopen(nodes_file, "r");
```

```
89      if (nodes == NULL) {
90          throw std::runtime_error("Can't open nodes file");
91      }
92      FILE* edges = fopen(edges_file, "r");
93      if (edges == NULL) {
94          throw std::runtime_error("Can't open edges file");
95      }
96      FILE* output = fopen(output_file, "wb");
97      if (output == NULL) {
98          throw std::runtime_error("Can't open output file");
99      }
100     std::vector<node_t> nodes_data;
101     std::vector<edge_t> edges_data;
102     read_nodes(nodes, nodes_data);
103     read_edges(edges, edges_data);
104     write_geo_data(output, nodes_data);
105     write_edges(output, edges_data);
106     write_offsets(output, nodes_data, edges_data);
107     fclose(nodes);
108     fclose(edges);
109     fclose(output);
110 }
111
112 #endif /* GRAPH_HPP */
```

Программа считывает файл с графом и запоминает места списков смежностей для вершин, чтобы быстро получить доступ в процессе поиска.

Чтобы быстро сопоставить идентификатор вершины его индексу в файлу, программа хранит все идентификаторы и впоследствии двоичным поиском находит индекс.

Для запроса поиска создаётся очередь с приоритетом, которая хранит непосещённые вершины и соответствующие им эвристические оценки.

```
1  #ifndef SEARCH_HPP
2  #define SEARCH_HPP
3
4  #include <queue>
5
6  #include "misc.hpp"
7
8  void read_geo(FILE* graph, uint32_t & id, double & lon, double & lat) {
9      fread(&id, sizeof(uint32_t), 1, graph);
10     fread(&lon, sizeof(double), 1, graph);
11     fread(&lat, sizeof(double), 1, graph);
12 }
13
14 void read_geo_data(FILE* graph, std::vector<uint32_t> & ids) {
15     uint32_t id;
16     double lon = 0;
17     double lat = 0;
18     while (lon < STOP_DATA) {
19         read_geo(graph, id, lon, lat);
20         ids.push_back(id);
21     }
22     ids.pop_back();
23 }
24
25 size_t count_edges(FILE* graph) {
26     size_t m = 0;
27     uint32_t id = 1;
28     while (id > 0) {
29         fread(&id, sizeof(uint32_t), 1, graph);
30         ++m;
31     }
32     return m - 1;
33 }
34
35 void read_offsets(FILE* graph, std::vector<uint32_t> & offsets) {
36     for (size_t i = 0; i < offsets.size() - 1; ++i) {
37         fread(&offsets[i], sizeof(uint32_t), 1, graph);
38     }
39 }
40
41 uint32_t find_node_index(const std::vector<uint32_t> & ids, const uint32_t node_id) {
```

```
42      int64_t l = -1;
43      int64_t r = ids.size();
44      while (l + 1 < r) {
45          int64_t m = (l + r) / 2;
46          if (ids[m] < node_id) {
47              l = m;
48          } else {
49              r = m;
50          }
51      }
52      return r;
53  }
54
55  node_t get_node_file(FILE* graph, const uint32_t node_index) {
56      fseek(graph, GEODATA_SIZE * node_index, 0);
57      node_t res;
58      read_geo(graph, res.id, res.lon, res.lat);
59      return res;
60  }
61
62  void prolong(FILE* graph, node_t finish, uint32_t cur_index, const std::vector<
        uint32_t> & ids, const std::vector<uint32_t> & offsets, std::vector<uint32_t> &
        prev, std::vector<double> & d, std::priority_queue<euristic_t> & q) {
63      node_t cur_node = get_node_file(graph, cur_index);
64      uint32_t cur_offset = offsets[cur_index];
65      std::vector<uint32_t> adjanced_nodes;
66      fseek(graph, GEODATA_SIZE * (ids.size() + 1) + cur_offset * sizeof(uint32_t), 0);
67      for (uint32_t i = cur_offset; i < offsets[cur_index + 1]; ++i) {
68          uint32_t next = 0;
69          fread(&next, sizeof(uint32_t), 1, graph);
70          adjanced_nodes.push_back(next);
71      }
72      for (size_t i = 0; i < adjanced_nodes.size(); ++i) {
73          uint32_t next = adjanced_nodes[i];
74          uint32_t next_index = find_node_index(ids, next);
75          node_t next_node = get_node_file(graph, next_index);
76          double cur_next_dist = geo_dist(cur_node, next_node);
77          if (d[next_index] < 0 or (std::abs(d[next_index] - (d[cur_index] +
                cur_next_dist)) > EPS and d[next_index] > d[cur_index] + cur_next_dist)) {
78              d[next_index] = d[cur_index] + cur_next_dist;
79              prev[next_index] = cur_index;
80              q.push(euristic_t(next, next_node, finish, d[next_index]));
81          }
82      }
83  }
84
85  double euristic_find(FILE* graph, const std::vector<uint32_t> & ids, const std::vector
        <uint32_t> & offsets, const uint32_t u, const uint32_t v, std::vector<uint32_t> &
        prev, std::vector<double> & d) {
```

```
86      uint32_t v_index = find_node_index(ids, v);
87      node_t v_node = get_node_file(graph, v_index);
88      uint32_t u_index = find_node_index(ids, u);
89      node_t u_node = get_node_file(graph, u_index);
90      d[u_index] = 0;
91      prev[u_index] = u_index;
92      std::priority_queue<euristic_t> q;
93      q.push(euristic_t(u, u_node, v_node, 0));
94      while (!q.empty()) {
95          euristic_t cur = q.top();
96          q.pop();
97          uint32_t cur_index = find_node_index(ids, cur.id);
98          if (cur.path > d[cur_index]) {
99              continue;
100         }
101         if (cur.id == v) {
102             break;
103         }
104         prolong(graph, v_node, cur_index, ids, offsets, prev, d, q);
105     }
106     return d[v_index];
107 }
108
109 void get_path(FILE* graph, uint32_t node_id, const std::vector<uint32_t> prev, std::
        vector<uint32_t> & res) {
110     while (prev[node_id] != node_id) {
111         node_t cur_node = get_node_file(graph, node_id);
112         res.push_back(cur_node.id);
113         node_id = prev[node_id];
114     }
115     node_t cur_node = get_node_file(graph, node_id);
116     res.push_back(cur_node.id);
117 }
118
119 void execute_search(FILE* input, FILE* output, FILE* graph, const std::vector<uint32_t
        > & ids, const std::vector<uint32_t> & offsets, bool full_output) {
120     size_t n = ids.size();
121     std::vector<uint32_t> prev(n);
122     std::vector<double> d(n);
123     uint32_t u = 0;
124     uint32_t v = 0;
125     while (fscanf(input, "%u%u", &u, &v) > 0) {
126         prev.assign(n, 0);
127         d.assign(n, -1);
128         double ans = euristic_find(graph, ids, offsets, u, v, prev, d);
129         uint32_t v_index = find_node_index(ids, v);
130         if (full_output) {
131             if (d[v_index] < 0) {
132                 fprintf(output, "-1 0\n");
```

```
133             } else {
134                 fprintf(output, "%.6lf ", ans);
135                 std::vector<uint32_t> path;
136                 get_path(graph, v_index, prev, path);
137                 fprintf(output, "%li ", path.size());
138                 for (size_t i = path.size() - 1; i > 0; --i) {
139                     fprintf(output, "%u ", path[i]);
140                 }
141                 fprintf(output, "%u\n", path[0]);
142             }
143         } else {
144             if (d[v_index] < 0) {
145                 fprintf(output, "-1\n");
146             } else {
147                 fprintf(output, "%.6lf\n", ans);
148             }
149         }
150     }
151 }
152 
153 void search_graph(char* graph_file, char* input_file, char* output_file, bool
        full_output) {
154     FILE* graph = fopen(graph_file, "rb");
155     if (graph == NULL) {
156         throw std::runtime_error("Can't open preprocessed graph file");
157     }
158     FILE* input = fopen(input_file, "r");
159     if (input == NULL) {
160         throw std::runtime_error("Can't open input file");
161     }
162     FILE* output = fopen(output_file, "w");
163     if (output == NULL) {
164         throw std::runtime_error("Can't open output file");
165     }
166     std::vector<uint32_t> ids;
167     read_geo_data(graph, ids);
168     size_t m = count_edges(graph);
169     std::vector<uint32_t> offsets(ids.size() + 1);
170     offsets.back() = m;
171     read_offsets(graph, offsets);
172     execute_search(input, output, graph, ids, offsets, full_output);
173     fclose(graph);
174     fclose(input);
175     fclose(output);
176 }
177 
178 #endif /* SEARCH_HPP */
```

Перед запуском препроцессинга и поиска программа проверяет аргументы и файлы.

```
#include <cstring>
#include <iostream>

#include "graph.hpp"
#include "search.hpp"

const int MIN_NUMBER_OF_PARAMETERS = 8;
const int MAX_NUMBER_OF_PARAMETERS = 9;

int main(int argc, char** argv) {
    if (argc < MIN_NUMBER_OF_PARAMETERS or argc > MAX_NUMBER_OF_PARAMETERS) {
        std::cout << "Wrong number of parameters" << std::endl;
        return 1;
    }
    if (strcmp(argv[1], "preprocess") == 0) {
        char* nodes_file = nullptr;
        char* edges_file = nullptr;
        char* output_file = nullptr;
        for (int i = 2; i < argc; ++i) {
            if (strcmp(argv[i], "--nodes") == 0) {
                nodes_file = argv[i + 1];
            } else if (strcmp(argv[i], "--edges") == 0) {
                edges_file = argv[i + 1];
            } else if (strcmp(argv[i], "--output") == 0) {
                output_file = argv[i + 1];
            }
        }
        if (nodes_file == nullptr) {
            std::cout << "Missing nodes file in programm arguments" << std::endl;
            return 1;
        }
        if (edges_file == nullptr) {
            std::cout << "Missing edges file in programm arguments" << std::endl;
            return 1;
        }
        if (output_file == nullptr) {
            std::cout << "Missing output file in programm arguments" << std::endl;
            return 1;
        }
        try {
            preprocess_graph(nodes_file, edges_file, output_file);
        } catch (std::exception & ex) {
            std::cout << ex.what() << std::endl;
            return 1;
        }
        return 0;
    }
    if (strcmp(argv[1], "search") == 0) {
```

```
49          char* graph_file = nullptr;
50          char* input_file = nullptr;
51          char* output_file = nullptr;
52          bool full_output = false;
53          for (int i = 2; i < argc; ++i) {
54              if (strcmp(argv[i], "--graph") == 0) {
55                  graph_file = argv[i + 1];
56              } else if (strcmp(argv[i], "--input") == 0) {
57                  input_file = argv[i + 1];
58              } else if (strcmp(argv[i], "--output") == 0) {
59                  output_file = argv[i + 1];
60              } else if (strcmp(argv[i], "--full-output") == 0) {
61                  full_output = true;
62              }
63          }
64          if (graph_file == nullptr) {
65              std::cout << "Missing preprocessed graph file in programm arguments" << std
                    ::endl;
66              return 1;
67          }
68          if (input_file == nullptr) {
69              std::cout << "Missing input file in programm arguments" << std::endl;
70              return 1;
71          }
72          if (output_file == nullptr) {
73              std::cout << "Missing output file in programm arguments" << std::endl;
74              return 1;
75          }
76          try {
77              search_graph(graph_file, input_file, output_file, full_output);
78          } catch (std::exception & ex) {
79              std::cout << ex.what() << std::endl;
80              return 1;
81          }
82          return 0;
83      }
84      std::cout << "Invalid parameters!" << std::endl;
85      return 2;
86  }
```

3 Консоль

```
engineerxl@engineerxl-GF63-Thin-9RCX:~/Study/DA/CP$ cat tests/1.nodes
3 99.358172 40.106455
8 69.050338 42.841631
2 18.778629 75.591193
5 -153.761221 -20.022587
16 97.388964 -71.087705
30 -99.922587 68.893304
15 -160.093275 -80.051214
23 40.108957 -56.244101
engineerxl@engineerxl-GF63-Thin-9RCX:~/Study/DA/CP$ cat tests/1.edges
2 3 2
2 30 3
2 5 8
2 5 15
2 16 3
3 16 30 8
2 30 23
2 16 15
2 30 5
engineerxl@engineerxl-GF63-Thin-9RCX:~/Study/DA/CP$ cat tests/1.in
30 23
15 8
16 5
15 2
5 23
engineerxl@engineerxl-GF63-Thin-9RCX:~/Study/DA/CP$ make
g++ -g -O2 -pedantic -std=c++17 -Wall -Wextra -lm main.cpp -o solution
engineerxl@engineerxl-GF63-Thin-9RCX:~/Study/DA/CP$ ./solution preprocess --edges
tests/1.edges --nodes tests/1.nodes --output tests/graph.b
engineerxl@engineerxl-GF63-Thin-9RCX:~/Study/DA/CP$ ./solution search --graph
tests/graph.b --input tests/1.in --output tests/out.txt --full-output
engineerxl@engineerxl-GF63-Thin-9RCX:~/Study/DA/CP$ cat tests/out.txt
13784805.044665 2 30 23
19917048.480887 3 15 5 8
17538214.812683 3 16 15 5
21723741.525241 4 15 16 3 2
20900905.665130 3 5 30 23
```

4 Тест производительности

В тесте сравниваются алгоритм Дейкстры и алгоритм поиска A*.

Каждый тест содержит 50 запросов на поиск.

В тестах представлены несвязные и плотные графы, чтобы сравнить алгоритмы в разных условиях.

Число вершин	Число рёбер	Алгоритм Дейкстры, мс	Алгоритм A*, мс
100	50	0.890	0.875
100	200	8.834	3.625
100	10^3	34.628	3.586
10^3	500	4.289	4.184
10^3	$2 * 10^3$	172.436	90.385
10^3	10^4	397.362	40.902
10^4	$5 * 10^3$	11.228	9.567
10^4	$2 * 10^4$	2034.514	980.609
10^4	10^5	5719.548	663.586

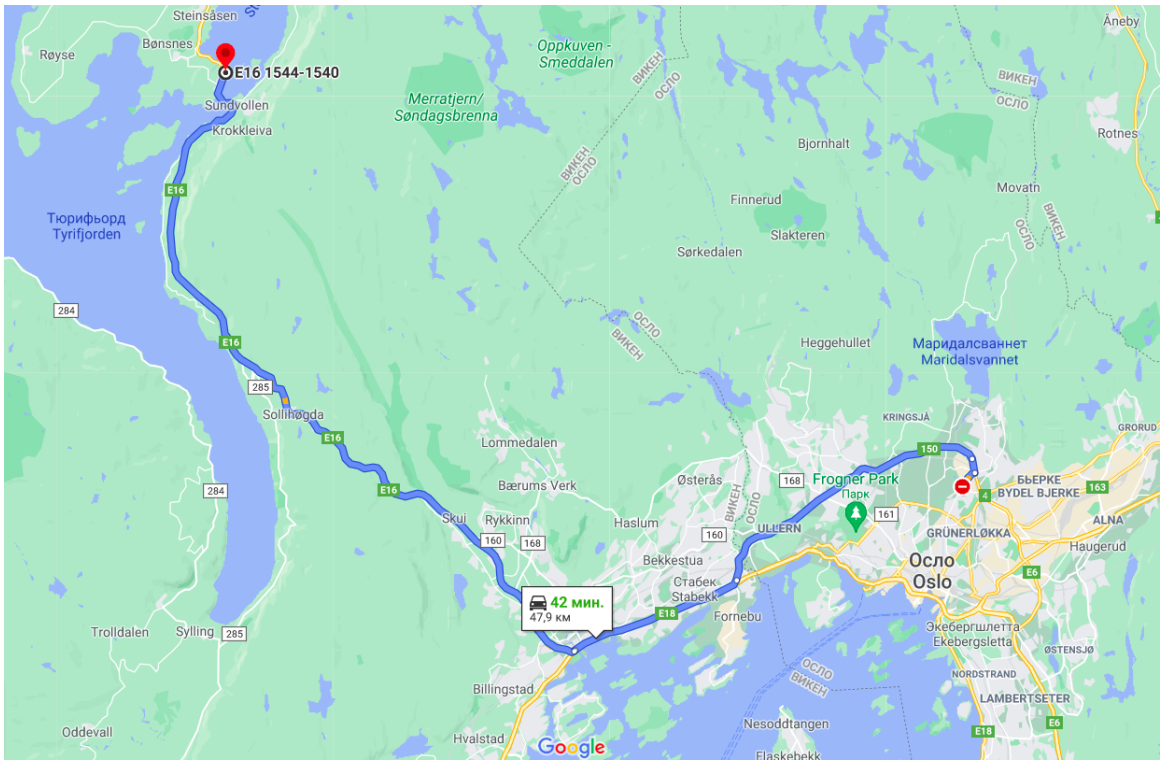
Видно, что алгоритм Дейкстры не проигрывает A* на несвязных графах, потому что так же быстро определяет, что пути нет.

Чем плотнее граф, тем быстрее оказывается A*, так как он сразу выбирает наиболее подходящую вершину с использованием эвристики.

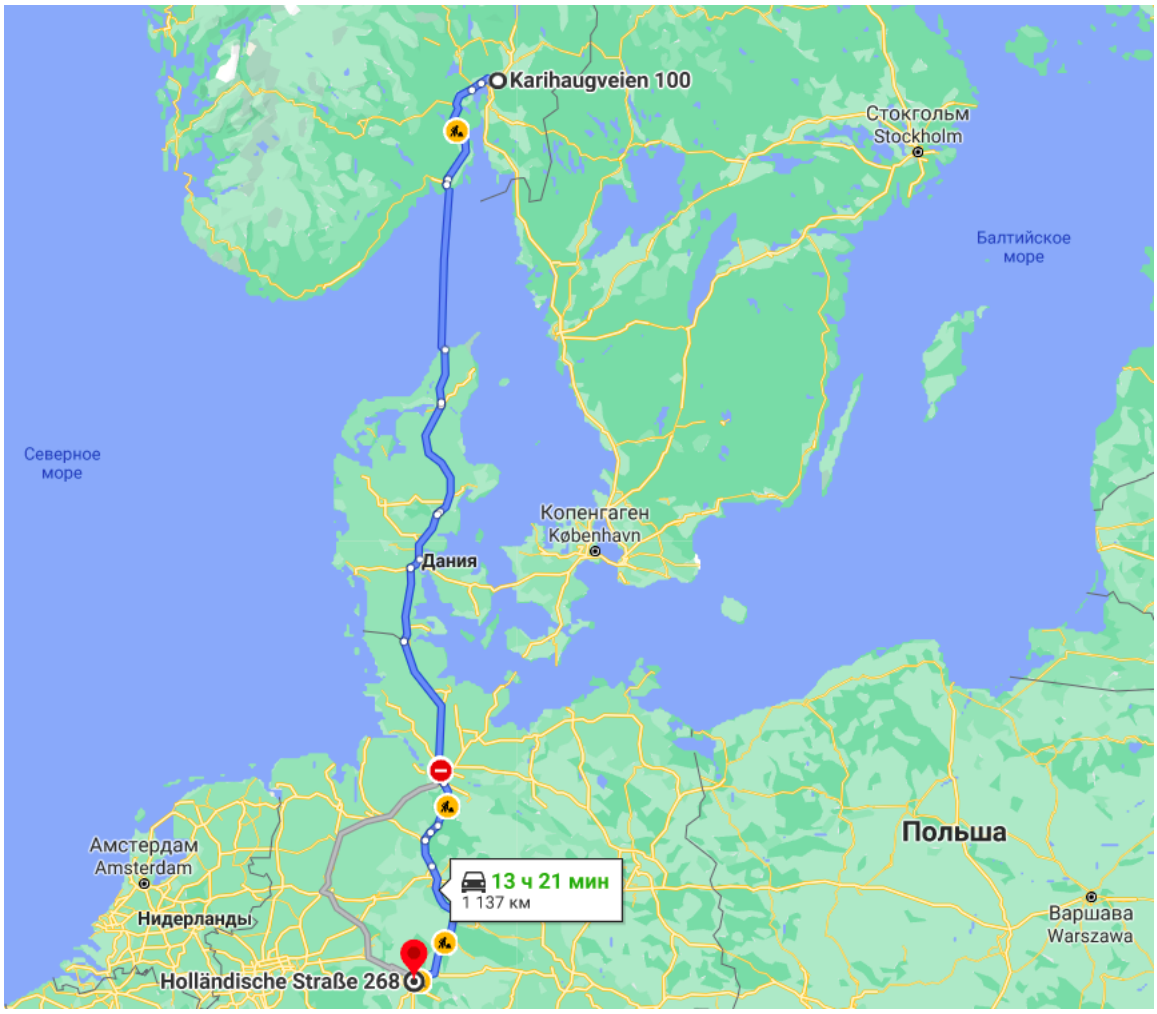
5 Оценка производительности на реальных данных

Произведём оценку времени работы алгоритма и точности найденного пути на реальных данных. Для этого используется граф карты Европы.

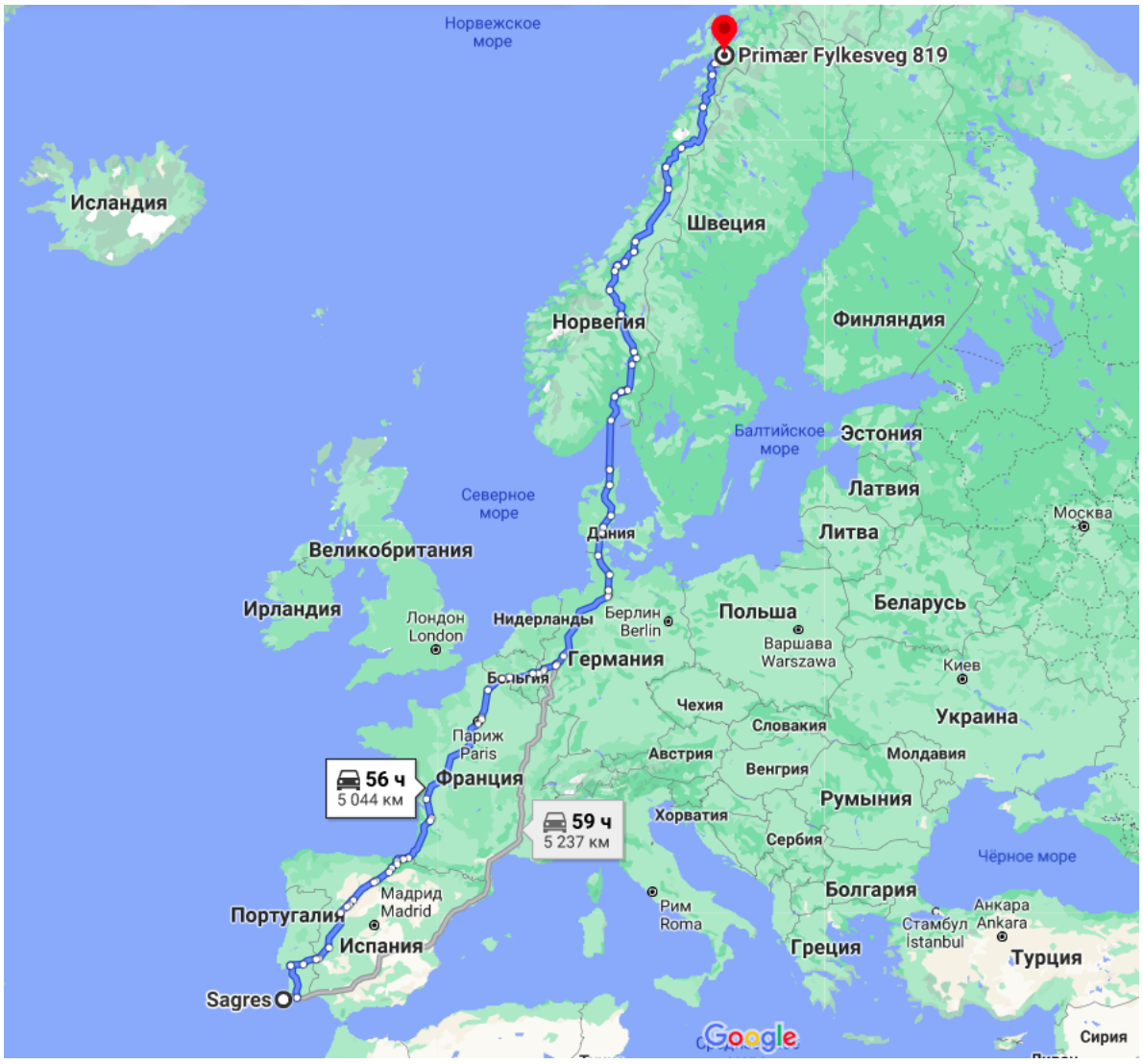
Поиск проезда в город занял у программы около секунды. При этом найденный путь (45119 метра) на три километра короче пути, который нашёл Google Maps.



Поиск проезда от берегов Норвегии до Германии занял тридцать секунд. Полученный результат (1346 километров) получится на 200 километров больше. Скорее всего, в имеющемся графе Европе не учтены паромные маршруты.



Поиск проезда от одной из самых западных точек Европы (мыс Сан-Висенте) до севера Норвегии занял восемь с половиной минут, и его длина составила 5126 километров. Google Maps справился с этой задачей гораздо быстрее, потому что просматривал только основные дороги. Моя программа работала с детальной картой Европы и учитывала возможность проезда по улицам города.



6 Выводы

В ходе выполнения курсового проекта я изучил алгоритмы поиска кратчайших путей в графах и реализовал алгоритм Дейкстры и A* с эвристикой, применимой к своему заданию.

Основной сложностью было компактно представить граф на диске, чтобы быстро и эффективно находить смежные вершины во время поиска. Я вспомнил про бинарные файлы и сжатое представление структур, применил списки смежности для хранения графа.

Для ускорения работы программы часть файла пришлось загружать в оперативную память, потому что бинарный поиск по файлу сильно замедлял работу программы даже на небольших запросах.

Так же я узнал про географические и плоские прямоугольные системы координат, использовал формулу для вычисления расстояния между двумя точками на Земле.

Почти во всех реальных задачах мы каким-либо образом можем оценить расстояние до цели (расстояние в пространстве или Манхэттенское расстояние), поэтому алгоритм A* становится применим.

Как было видно в сравнении, он в несколько раз быстрее на больших и плотных графах. С учётом того, как быстро растут города, улиц и жилых кварталов становится всё больше и граф города плотнее.

Список литературы

[1] *Алгоритм Дейкстры — Викиконспекты*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_Дейкстры` (дата обращения: 21.04.2021).

[2] *Алгоритм A* — Викиконспекты*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_A*` (дата обращения: 21.04.2021).

[3] *A* search algorithm — Wikipedia*
URL: `https://en.wikipedia.org/wiki/A*_search_algorithm` (дата обращения: 21.04.2021).