Московский авиационный институт

(национальный исследовательский институт)

Институт «Компьютерные науки и прикладная математика»

Курсовая проект

«Эвристический поиск на графах»

по курсу

«Дискретный анализ»

V семестр

Студент: Меркулов Ф. А.
Группа: М8О-307Б-21

Руководитель: Сорокин С. А.

Оценка:

Дата:

Москва, 2023

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

Файл узлов:

<id> <lat> <lon>

Файл рёбер:

<длина дороги в вершинах [n]> <id 1> <id 2> ... <id n>

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 1e-6

Если опция --full-output указана: на каждый запрос выводится отдельная строка, с длиной кратчайшего пути между заданными вершинами с относительной погрешностью не более 1e-6 , а затем сам путь в формате как в файле рёбер.

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

Метод решения

Требуется реализовать алгоритм поиска кратчайшего пути во взвешенном графе. С этой задачей отлично справляется алгоритм Дейкстры, но в силу особенности задачи так же применим алгоритм поиска A*.

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 кратчайшим путем была

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

Алгоритм поиска A* работает похожим образом, но для выбора вершины на каждом шаге использует 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 до финиша. В моём случае функции $h(v)$ —расстояние между двумя точками на Земле. Алгоритм по сути является расширением алгоритма Дейкстры, но он достигает большей производительности (по времени) с помощью эвристики.

Исходный код

При реализации программы потребовалось решить следующие аспекты:

1. Определение вспомогательных типов
2. Препроцессинг
3. Сам алгоритм A* с эвристикой расстояния между двумя точками на Земле
4. Восстановление пути, чтобы уметь отвечать, когда используют ключ –full-output

Определение вспомогательных типов. Потребовалось реализовать типы для хранения вершины в векторе вершин и для правильной её обработки алгоритмом A*, а также структуры для хранения рёбер и пути.

```
// Защита от повторного включения файла

#pragma once


// Включение стандартных библиотек

#include <iostream>

#include <fstream>

#include <cstring>

#include <vector>

#include <unordered_map>

#include <algorithm>

#include <cmath>

#include <queue>

#include <chrono>

#include <cstdio>

#include <cassert>

#include <exception>

// Использование пространства имен std для удобства
```

```
using namespace std;

// Константы для расчетов
const double alpha = 180.0;              // Количество градусов в полукруге
const double radius  = 6371000.0;    // Радиус Земли в метрах

// Определение структуры Node для хранения информации об узлах графа
struct Node{
   uint32_t id;
   double lat, lon; //Широта и Долгота
};

// Перегрузка оператора ввода для чтения информации об узле из потока
inline istream& operator>>(istream& is, Node& node){
   is >> node.id >> node.lat >> node.lon;
   return is;
}

// Перегрузка оператора сравнения для сортировки узлов по идентификатору
inline bool operator<(Node& lhs, Node& rhs){
   return lhs.id < rhs.id;
}


// Определение структуры Edge для хранения информации об ребрах графа
struct Edge{
   uint32_t from, to;
};

// Перегрузка оператора сравнения для сортировки ребер
inline bool operator<(const Edge& lhs, const Edge& rhs){
   if(lhs.from != rhs.from){
```

```
        return lhs.from < rhs.from;
    }
    return lhs.to < rhs.to;
}



// Определение структуры Path для хранения информации о пути в графе
struct Path{
    uint32_t to;
    double cost;
};
// Перегрузка оператора сравнения для очереди с приоритетами, чтобы Path с меньшей стоимостью имел более высокий приоритет
inline bool operator<(const Path& lhs, const Path& rhs){
    if(lhs.cost != rhs.cost){
        return lhs.cost > rhs.cost;
    }
    return lhs.to > rhs.to;
}
```

Препроцессинг. Для того чтобы данными было удобнее пользоваться я решил их отсортировать и записать в бинарный файл, чтобы сократить необходимое место для файла с информацией. Таким образом происходила обработка вершин и рёбер (строятся списки смежности для более удобного представления графа в файле).

```
#include "preprocess.h"


// Функция для предобработки узлов графа
void preprocessNodes(FILE* nodesFile, FILE* outputFile, vector<uint32_t>& nodes_id) {
    vector<Node> nodes;
    Node cur;


    // Чтение данных узлов из файла
    while(fscanf(nodesFile, "%u%lf%lf", &cur.id, &cur.lat, &cur.lon) > 0) {
        cur.id -= 1; // Адаптация идентификатора узла, если необходимо
```

```
            nodes.push_back(cur);
        }

        // Сортировка узлов по идентификатору
        sort(nodes.begin(), nodes.end());

        // Заполнение вектора идентификаторов узлов
        nodes_id.resize(nodes.size());
        for(size_t i = 0; i < nodes.size(); ++i) {
            nodes_id[i] = nodes[i].id;
        }

        // Запись информации об узлах в выходной файл
        size_t size = nodes.size();
        fwrite(&size, sizeof(size), 1, outputFile);
        fwrite(&nodes_id[0], sizeof(nodes_id[0]), nodes_id.size(), outputFile);
        for(size_t i = 0; i < size; ++i) {
            fwrite(&nodes[i].lat, sizeof(nodes[0].lat), 1, outputFile);
            fwrite(&nodes[i].lon, sizeof(nodes[0].lon), 1, outputFile);
        }
    }

    // Функция для предобработки рёбер графа
    void preprocessEdges(FILE* edgesFile, FILE* outputFile, vector<uint32_t>& nodes_id) {
        vector<Edge> edges;
        uint32_t n, curId, prevId;

        // Чтение данных о рёбрах из файла
        while(fscanf(edgesFile, "%u%u", &n, &curId) == 2) {
            for(size_t i = 1; i < n; ++i) {
                prevId = curId;
                fscanf(edgesFile, "%u", &curId);
```

```
            edges.push_back({prevId - 1, curId - 1}); // Добавление ребра
            edges.push_back({curId - 1, prevId - 1}); // Добавление обратного ребра для
неориентированного графа
        }
    }

    // Сортировка рёбер
    sort(edges.begin(), edges.end());

    // Запись информации о рёбрах в выходной файл
    uint32_t i = 0;
    for(size_t k = 0; k < nodes_id.size(); ++k) {
        uint32_t curEdgeFrom = nodes_id[k];
        while(i < edges.size() && edges[i].from == curEdgeFrom) {
            ++i;
        }
        fwrite(&i, sizeof(i), 1, outputFile);
    }

    for(Edge& edge : edges) {
        fwrite(&edge.to, sizeof(edge.to), 1, outputFile);
    }
}
```

Реализация алгоритма A* с эвристикой расстояния между двумя точками на Земле. При обработке вершины из файла загружаются и смежные с ней, из соображений алгоритма.

```
// Реализация алгоритма A*
double AStar(uint32_t start, uint32_t goal, vector<uint32_t>& ids,
             vector<uint32_t>& offsets, FILE* inputFile, uint32_t infoStart, uint32_t adjStart,
             vector<double>& g, vector<double>& f, vector<uint32_t>& parent) {

    // Инициализация векторов расстояний и оценок
    g.assign(g.size(), -1.0);
```

```
f.assign(f.size(), -1.0);
priority_queue<Path> q;

// Поиск индексов начальной и конечной точек
start = binSearch(start, ids);
goal = binSearch(goal, ids);

// Получение узла цели
Node goalNode = getNode(goal, inputFile, infoStart);

// Инициализация начальной точки в очереди
g[start] = 0;
f[start] = g[start] + calcDistance(getNode(start, inputFile, infoStart), goalNode);
q.push({start, f[start]});
parent[start] = start;

// Цикл поиска пути
while (!q.empty()) {
   uint32_t cur = q.top().to;
        double curCost = q.top().cost;
   q.pop();

   // Проверка, достигнута ли цель
   if (cur == goal) {
      break;
   }



        if(curCost > f[cur]){
      continue;
   }
```

```
    size_t startLoad = 0, toLoad = 0;
    if(cur == 0){
        toLoad = offsets[0];
    } else{
        startLoad = offsets[cur - 1]; //сколько ребер пропустить
        toLoad = offsets[cur] - offsets[cur - 1]; //скольлко ребер у текущей вершины
    }

    vector<uint32_t> curAdj(toLoad); //список доступных вершин из данной
    fseek(inputFile, adjStart + startLoad * sizeof(uint32_t), SEEK_SET);
    fseek(inputFile, adjStart + startLoad * sizeof(uint32_t), SEEK_SET);
    for(size_t i = 0; i < toLoad; ++i){
        fread(&curAdj[i], sizeof(uint32_t), 1, inputFile);
    }

    Node curNode = getNode(cur, inputFile, infoStart);

    for(uint32_t next : curAdj){
        next = binSearch(next, ids);

        Node nextNode = getNode(next, inputFile, infoStart);

        double tentativeScore = g[cur] + calcDistance(nextNode, curNode);

        if((g[next] < 0) || (1e-6 < (g[next] - tentativeScore))){
            g[next] = tentativeScore;
            f[next] = g[next] + calcDistance(nextNode, goalNode);
            parent[next] = cur;
            q.push({next, f[next]});
        }
    }
}
```

```
    return (g[goal] == -1.0) ? __DBL_MAX__ : g[goal]; // Возврат найденного расстояния
}
```

Эвристика.

```
// Функция преобразует угловые градусы в радианы
inline double radians(double x) {
    return x * M_PI / alpha; // M_PI - математическая константа Пи
}

// Функция расчета расстояния между двумя узлами использует формулу гаверсинуса
inline double calcDistance(const Node& a, const Node& b) {
    double deltaLat = radians(b.lat) - radians(a.lat);
    double deltaLon = radians(b.lon) - radians(a.lon);

    // Формула гаверсинуса - это способ вычисления расстояний на сфере
    // Она учитывает кривизну земной поверхности для вычисления расстояния
    return 2 * radius * asin(sqrt(pow(sin(deltaLat / 2), 2) +
                                  cos(radians(a.lat)) * cos(radians(b.lat)) *
                                  pow(sin(deltaLon / 2), 2)));
}
```

Перед запуском препроцессинга и поиска программа проверяет аргументы файла.

```
#include "preprocess.h"
#include "search.h"

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 || 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];
            i++; // Пропуск следующего аргумента, так как мы его уже рассмотрели
        } else if (strcmp(argv[i], "--edges") == 0) {
            edges_file = argv[i + 1];
            i++; // Пропуск следующего аргумента, так как мы его уже рассмотрели
        } else if (strcmp(argv[i], "--output") == 0) {
            output_file = argv[i + 1];
            i++; // Пропуск следующего аргумента, так как мы его уже рассмотрели
        }
    }

    if (!nodes_file || !edges_file || !output_file) {
        std::cout << "Missing file argument(s) in preprocess mode" << std::endl;
        return 1;
    }

    try {
        FILE* nodesFile = fopen(nodes_file, "r");
        FILE* edgesFile = fopen(edges_file, "r");
        FILE* outputFile = fopen(output_file, "wb");

        if (!nodesFile || !edgesFile || !outputFile) {
```

```
            throw std::runtime_error("Error opening file(s)");
        }

        vector<uint32_t> ids;
        preprocessNodes(nodesFile, outputFile, ids);
            fclose(nodesFile);

            preprocessEdges(edgesFile, outputFile, ids);
        fclose(edgesFile);
        fclose(outputFile);
    } catch (const std::exception &ex) {
        std::cout << "Error during preprocessing: " << ex.what() << std::endl;
        return 1;
    }
    return 0;
}

if (strcmp(argv[1], "search") == 0) {
    char* graph_file = nullptr;
    char* input_file = nullptr;
    char* output_file = nullptr;
    bool full_output = false;

    for (int i = 2; i < argc; ++i) {
        if (strcmp(argv[i], "--graph") == 0) {
            graph_file = argv[i + 1];
            i++; // Пропуск следующего аргумента, так как мы его уже рассмотрели
        } else if (strcmp(argv[i], "--input") == 0) {
            input_file = argv[i + 1];
            i++; // Пропуск следующего аргумента, так как мы его уже рассмотрели
        } else if (strcmp(argv[i], "--output") == 0) {
            output_file = argv[i + 1];
```

```
            i++; // Пропуск следующего аргумента, так как мы его уже рассмотрели
        } else if (strcmp(argv[i], "--full-output") == 0) {
            full_output = true;
        }
    }

    if (!graph_file || !input_file || !output_file) {
        std::cout << "Missing file argument(s) in search mode" << std::endl;
        return 1;
    }

    try {
        FILE* graphFile = fopen(graph_file, "rb");
        FILE* inputFile = fopen(input_file, "r");
        FILE* outputFile = fopen(output_file, "w");

        if (!graphFile || !inputFile || !outputFile) {
            throw std::runtime_error("Error opening file(s)");
        }

        search(graphFile, inputFile, outputFile, full_output);

        fclose(graphFile);
        fclose(inputFile);
        fclose(outputFile);
    } catch (const std::exception &ex) {
        std::cout << "Error during search: " << ex.what() << std::endl;
        return 1;
    }
    return 0;
}
```

```
    std::cout << "Invalid operation name provided" << std::endl;
    return 2;
}
```

Консоль

```
papik@ubuntu:~/DACP/src$ cat 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
papik@ubuntu:~/DACP/src$ cat 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
papik@ubuntu:~/DACP/src$ cat 1.in
30 23
15 8
16 5
15 2
5 23
papik@ubuntu:~/DACP/src$ ./prog preprocess --nodes 1.nodes --edges 1.edges --output graph.b
papik@ubuntu:~/DACP/src$ ./prog search --graph graph.b --input 1.in --output output.txt --full-output
papik@ubuntu:~/DACP/src$ cat output.txt
13784805.044665 2 30 23
16548736.467759 3 15 5 8
```

19669673.981963 3 16 15 5

21723741.525241 4 15 16 3 2

26642397.769294 3 5 30 23

papik@ubuntu:~/DACP/src$ cat europe.in

409933124 345859083

papik@ubuntu:~/DACP/src$ time ./prog preprocess --nodes europe.nodes --edges europe.edges --output graph.b

real 7m4,003s

user 6m34,754s

sys 0m23,810s

papik@ubuntu:~/DACP/src$ time ./prog search --graph graph.b --input europe.in --output output.txt

real 1m37,063s

user 0m24,990s

sys 0m21,187s

papik@ubuntu:~/DACP/src$ cat output.txt

505811.642518

papik@ubuntu:~/DACP/src$ cat europe.in

409933136 517970603

papik@ubuntu:~/DACP/src$ time ./prog search --graph graph.b --input europe.in --output output.txt

real 13m26,364s

user 5m44,161s

sys 4m6,025s

papik@ubuntu:~/DACP/src$ cat output.txt

2493166.749208

papik@ubuntu:~/DACP/src$ cat europe.in

409933136 656914

papik@ubuntu:~/DACP/src$ time ./prog search --graph graph.b --input europe.in --output output.txt

real 1m8,761s

user 0m23,225s

sys 0m20,325s

papik@ubuntu:~/DACP/src$ cat output.txt

534061.587723

Тест производительности

Алгоритм обработал часть карты Европы вместе со всеми возможными для посещения вершинами и дорогами, после чего происходит поиск расстояния от 1 случайной вершины до другой и сравнивается с расстоянием, которое показывают карты Google между этими же двумя точками.

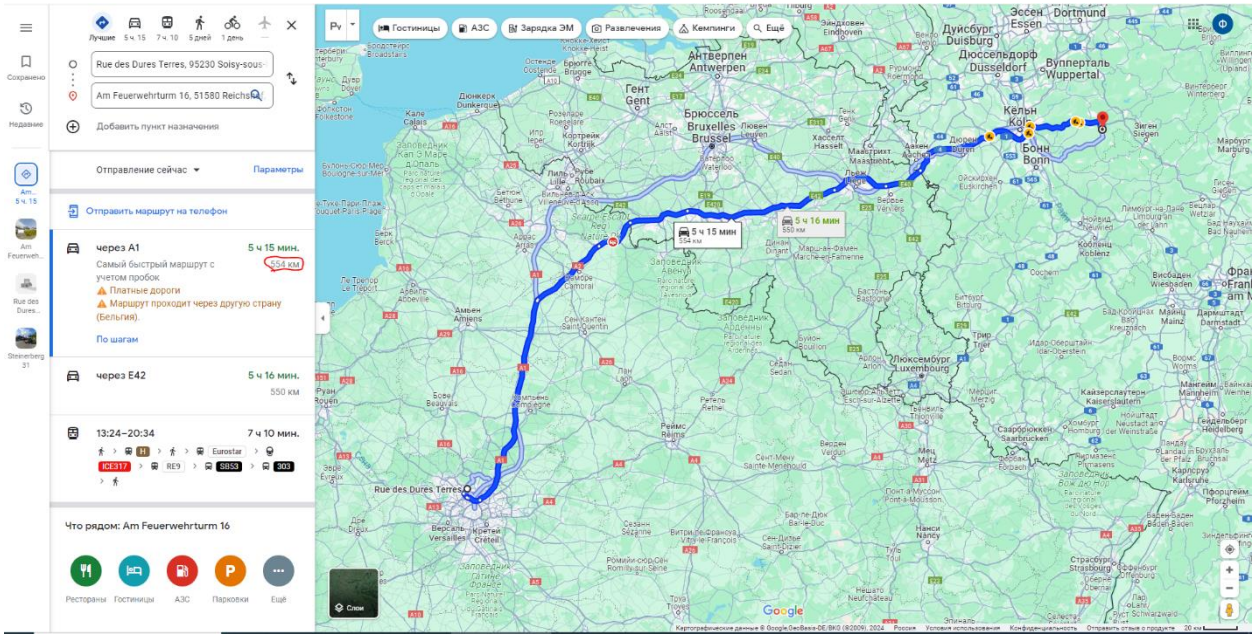
1 Тест:

Вершины: 1)409933124 (lat:48.9897699 lon:2.2946286)

2)345859083 (lat:50.9395336 lon:7.6547947)

Расстояние найденное программой: 505811.642518 метров ~ 506 километров
Расстояние найденное с помощью Google карт: 554 километра

Время вычисления: 1 минута 37,063 секунд



Такая разница в расстоянии появляется из-за того, что Google карты просматривают только основные дороги, а моя программа работала с детальной картой и учитывала возможность проезда по самым незначительным дорогам, что плохо повлияло на время выполнения, но улучшило кратчайшее расстояние.

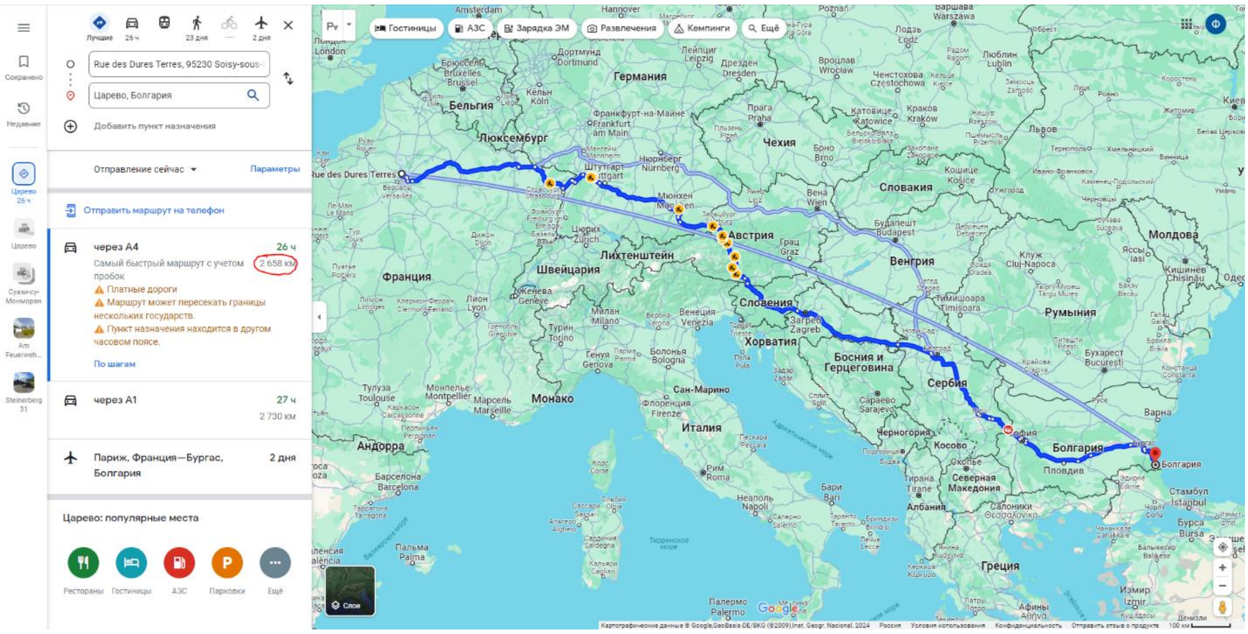
2 Тест:

Вершины: 1)409933136 (lat:48.9896199 lon:2.2928716)

2)517970603 (lat:42.0958833 lon:27.8413832)

Расстояние найденное программой: 2493166.749208 метров ~ 2493 километра
Расстояние найденное с помощью Google карт: 2658 километра

Время вычисления: 13 минут 26,364 секунд



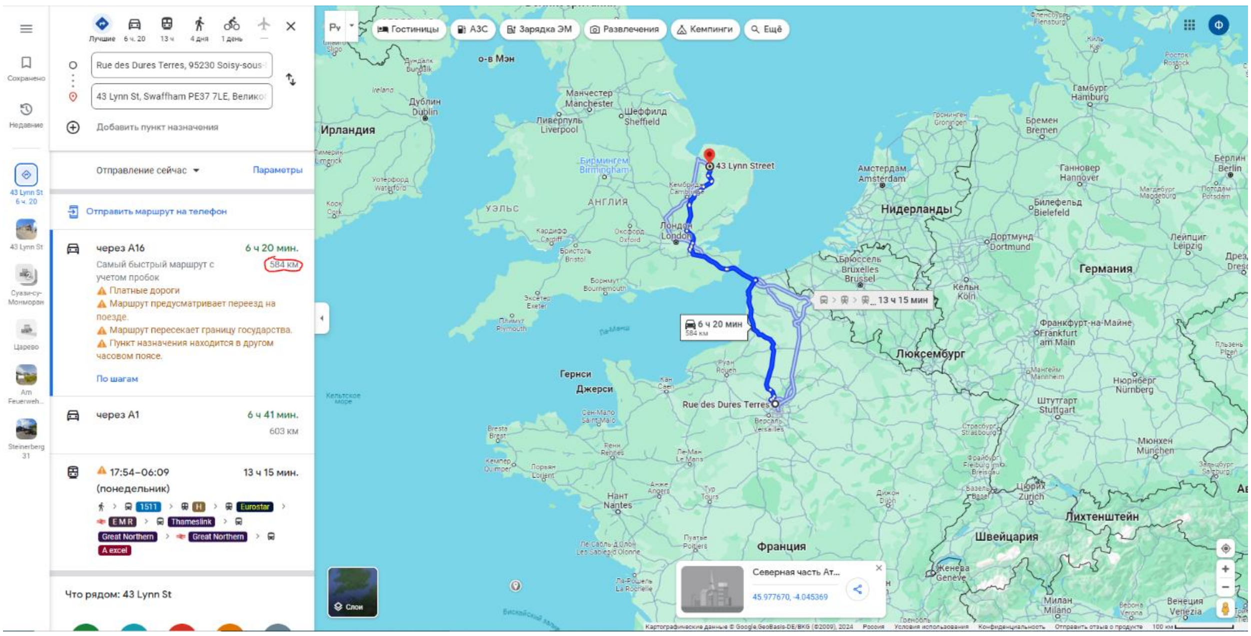
3 Тест:

Вершины: 1)409933136 (lat:48.9896199 lon:2.2928716)

2)656914 (lat:52.6486029 lon:0.6879259)

Расстояние найденное программой: 534061.587723 метров ~ 534 километра
Расстояние найденное с помощью Google карт: 584 километра

Время вычисления: 1 минута 8,761 секунд



Выводы

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

Почти во всех реализациях задач мы каким-либо образом можем оценить расстояние до цели (расстояние в пространстве, Манхэттенское расстояние, расстояние на Земле), поэтому алгоритм A* становится применим.

Главной сложностью, помимо реализации алгоритма A* стала грамотная обработка графа в файл и обратно, с максимальной экономией пространства. Также моё решение загружает смежные вершины с проверяемой прямо в процессе работы алгоритма A*, что также снижает потребляемую память.