Курсовая работа по курсу дискретного анализа: Эвристический поиск на графах

Выполнил студент группы М8О-308Б-21 МАИ *Белоносов Кирилл*.

Условие

1. Общая постановка задачи: Реализуйте систему для поиска пути в графе дорог с использованием эвристических алгоритмов.

2. Вариант задания(2):

   Алгоритм:

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

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

   Файл узлов:
   <id> <lat> <lon>
   Файл рёбер:
   <длина дороги в вершинах [n]> <id 1> <id 2> ... <id n>

   Формат ввода:

   Файл с состоящий из запросов. Каждый запрос состоит из пары индексов - начальной позиции и конечной

   Формат вывода:

   Если опция –full-output не указана: на каждый запрос в отдельной строке выводится длина крат- чайшего пути между заданными вершинами с относительной погрешностью не более 1e-6.

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

Метод решения

Для выполнения данной работы предполагается разбить задачу на два подэтапа: сделать максимально удобное представление графа в виде файла для дальнейшей работы с ним и написать алгоритм поиска, который не будет загружать полностью в память все данные графа, а динамически подгружать из файла (как бы используя кэш).

Для решения первой задачи, можно сделать следующее: перезаписать файлы с вершинами и ребра в бинарную форму, что позволит читать и записывать в них быстрее,

а также делать сдвиги. Далее нужно подумать на структурой файла. В процессе поиска нам необходимо быстро подгружать инцедентные вершины, для текущей вершины. Поэтому будем представлять для нашей вершины инцедентные ей как:

$$< countofnodes >< id1 >< id2 > ...$$

До этого запишием в файла все узлы в том же формате, что и в исходном файле. Чтобы быстро находить участок файла, в котором лежит необходимый нам набор вершин запишем пары

$$< id >< position >$$

position - указывает в каком месте файла находится небходимая нам информация по инцедентным вершинам для текущей вершины. Такое представление файла показалось мне наиболее удобным, для дальнейшей работы с ним.

Далее неоходимо реализовать поиск в графе, для этого немного перепишем уже существующий алгоритм A*. Будем хранить не полную структуру графа, а только id вершин и указатели на чанки с инцедентными вершинами. Когда мы достаем из приоритетной очереди очередную вершину, то подгружаем список её инцедентных вершин. Также, так как создавать массив длинной равный максимальному возможному значению не рационально, то реализуем бинпоиск для поиска индекса соответствующей вершины. Удачно складывается то, что индекс показывает, сколько нам необходимо сдвинуться от начала файла, чтобы считать данные нужной вершины, что мы также используем.

Описание программы

В следующем блоке опишу основные структуры данных и константы, которые я использовал для работы

```
const double INF = 1e18;
const double EARTH_RADIUS = 6371e3;

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

struct Edge {
    uint32_t from;
    uint32_t to;
};

struct Pointer {
```

```
    uint32_t id;
    uint32_t location;
};

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

bool operator< (const Edge& lhs, const Edge& rhs) {
    if(lhs.from != rhs.from) {
        return lhs.from < rhs.from;
    } else {
        return lhs.to < rhs.to;
    }
}

struct fu {
    double f;
    uint32_t u;
};

bool operator < (const fu& lhs, const fu& rhs) {
    return lhs.f > rhs.f;
}
```

Функция h рассчитывает эвристику

```
double to_radians(double angle) {
    return angle * M_PI / 180.0;
}

double h(const Node& p1, const Node& p2) {
    double phi = to_radians(p1.lon);
    double psi = to_radians(p2.lon);
    double delta = to_radians(p1.lat - p2.lat);
    double d = acos(sin(phi) * sin(psi) + cos(phi) * cos(psi) *
       cos(delta));
    return isnan(d) ? 0 : EARTH_RADIUS * d;
}
```

Сама работа алгоритма происходит в функции AStar. Небольшим отличием от прошлой реализации является наличие вектора p, для сохранения пути

```
double AStar(FILE* file, uint32_t start, uint32_t goal, uint32_t
   node_count, const vector<uint32_t>& nodes, const vector<int>&
   pointers, vector<uint32_t>& p) {
    vector<bool> visited(node_count, false);
```

```
    vector<double> d(node_count, INF);
    uint32_t start_index = get_index(start, nodes);
    uint32_t goal_index =  get_index(goal, nodes);
    d[start_index] = 0;
    p[start_index] = start_index;
    priority_queue<fu> pq;
    Node start_node = get_node(file, start_index);
    Node goal_node = get_node(file, goal_index);
    pq.push({d[start_index] + h(start_node, goal_node), start});
    while(!pq.empty()) {
        fu cur = pq.top();
        uint32_t current_index = get_index(cur.u, nodes);
        if(cur.u == goal)
            return d[current_index];
        pq.pop();
        visited[current_index] = true;
        vector<uint32_t> incident = get_incident_node(file, cur.u,
           node_count, nodes, pointers);
        Node current = get_node(file, current_index);
        for(const auto& v: incident) {
            uint32_t v_index = get_index(v, nodes);
            if(visited[v_index])
                continue;
            Node v_node = get_node(file, v_index);
            double tmp = d[current_index] + h(current, v_node);
            if(tmp < d[v_index]) {
                d[v_index] = tmp;
                p[v_index] = current_index;
                pq.push({d[v_index] + h(v_node, goal_node), v});
            }
        }
    }
    return -1;
}
```

Дневник отладки

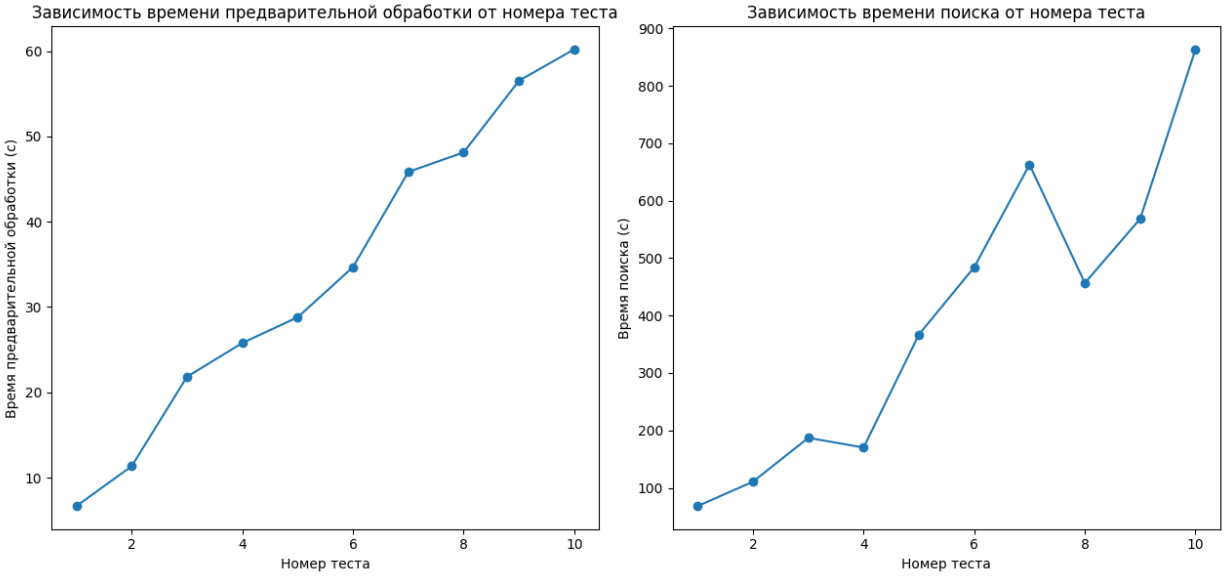
Большинство ошибок возникало на этапе препроцессинга файла (игра с байтами). Далее большой пласт ошибок возникал на этапе подгрузки инцедентных ребер - необходимо было правильно посчитать какой нужно сделать сдвиг в байтах, чтобы попасть в нужный участок памяти

#	Кол-во вершин	Кол-во ребер	Кол-во запросов	Время обработки s	Время поиска s
1	1000000	1000000	10	6.67	68.342
2	2000000	2000000	10	11.347	110.736
3	3000000	3000000	10	21.832	187.105
4	4000000	4000000	10	25.789	170.241
5	5000000	5000000	10	28.802	366.970
6	6000000	6000000	10	34.684	484.230
7	7000000	7000000	10	45.839	662.736
8	8000000	8000000	10	48.130	456.679
9	9000000	9000000	10	56.526	568.422
10	10000000	10000000	10	60.214	863.554

0.1 Результаты тестирования

Тест производительности

Результаты тестирования программы, с созданием искусственных данных и постепенным ростом числа вершин и ребер:



Можно заметить, что время поиска сильно растет с размером входных данных. Попытаемся примерно оценить сложность алгоритмов. Препроцессинг - это считывание 2 файлов за линейную сложность. Но также идет их сортировка. Я использовал стандартную сортировку из STL, которая имеет сложность O(nlogn). Поэтому сложность препроцессинга я бы оценил как: O(nlogn + mlogm), где n - число вершин, а m - число ребер. Время поиска оценить сложнее. Конечно в идеале оно также состовляет O(mlogm), но подгрузка инцедентых вершин требует доп времени и получение индексов по нодам происходит за O(logn).

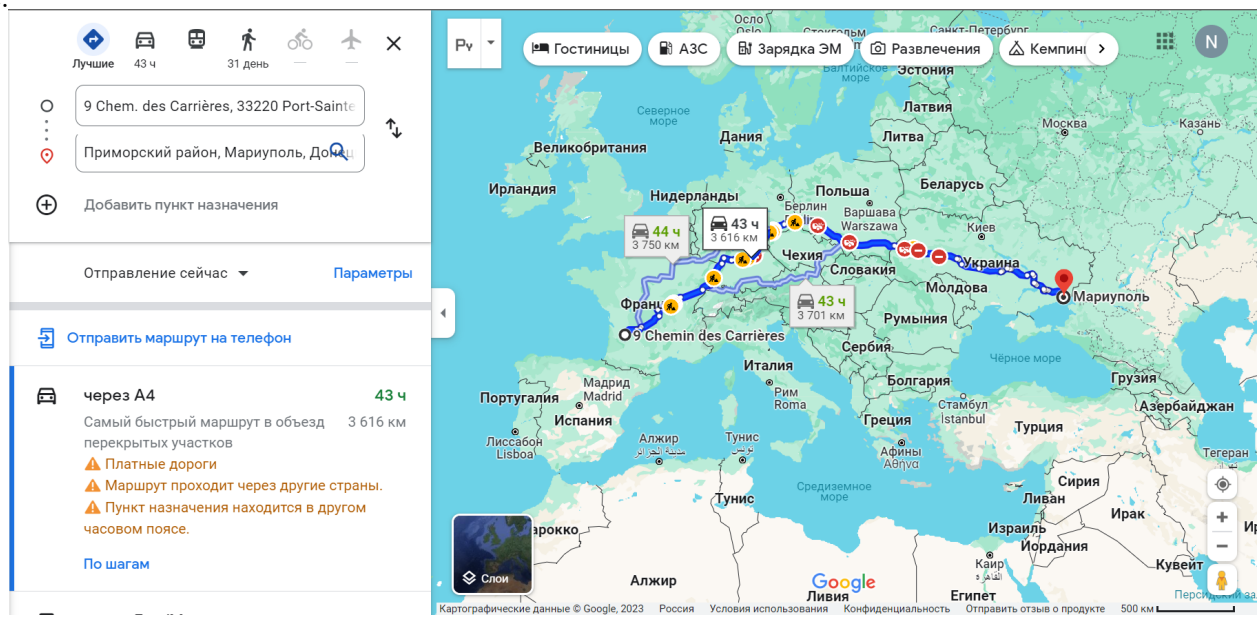
Тестирование на реальных данных

Для тестирования на реальнных данных я выбрал случайно две точки:

$$id = 4251282260, lat = 47.0742186, lon = 37.5210893$$

$$id = 1347044888, lat = 44.8520422, lon = 0.2048346$$

В результате ответ получился: 4564382.848998 метров, количество ребер на пути: 42454
Если же использовать гугл карты для поиска пути, значение получается намного меньше:



Возможно база данных карт успела обновиться или алгоритмы гугл карт научились находить более оптимальные пути

Выводы

В результате данной лабораторной работы я реализовал алгоритм A* для поиска пути среди больших данных. Также я узнал, как считать эвристику на сфере (в данном случае на земле). Пришлось применить множество знаний с предыдущих курсов, чтобы выполнить данное задание. Также включало в себя некоторую творческую работу по созданию наиболее быстрее и удобного представления графа на файле. К тому же решение получилось универсальным и может работать с любими данным, представленными в таком же формате. Конечно, не получилось получить хорошую точность, на что могло повлиять множество факторов, но в результате тестирования показана работоспособность программы. Полученные знания помогут в дальнейшей работе, т.к. алгоритмы поиска пути применяются, не только в картографических сервисах, но и в навигаторах, анализе снимков и разработке игр