Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Курсовой проект по курсу «Дискретный анализ»

Студент: А. А. Почечура
Преподаватель: С. А. Сорокин
Группа: М8О-306Б
Дата:
Оценка:
Подпись:

Москва, 2023

Курсовой проект: усложнённый вариант

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

Формат входных данных:

./prog preprocess –nodes <nodes file> –edges <edges file> –output <preprocessed graph>
Входной файл с перекрёстками, входной файл с дорогами, выходной файл с графом.
./prog search –graph <preprocessed graph> –input <input file> -output <output file> [–full-output]
Входной файл с графом, входной файл с запросами, выходной файл с ответами на запросы, переключение формата выходного файла на подробный.

Формат результата:

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.

1 Описание

Требуется выполнить две крупные задачи: осуществить эффективное и удобное хранение графа, а так же, используя информацию из графа, найти путь из одной точки Земли в другую. Было принято решение записать координаты и id вершин к ним в том же виде, в каком они подаются на вход; граф хранить в виде списка смежности, а после графа будут записаны разделители. Разделители будут показывать, к какой вершние относится конкретный промежуток с рёбрами (то есть, её соседи). После чего займёмся самим поиском между точками. Нужно реализовать алгоритм A^*, где в качестве евристики будем использовать удаление от начальной точки и приближение к конечной. Расстояние будет считаться по формуле расстояния между двумя точками Земли, используя широту и долготу. Также, для возможности вывода пути, будет использоваться массив, в котором для каждой вершины i будет храниться, из какой вершины мы пришли в вершину i.

2 Исходный код

Реализуем несколько структур. При вводе будем использовать структуру *cord*, в которой буду храниться *id* вершины и её координаты; с помощью структуры *edges* будем хранить рёбра (откуда - куда); структура *euristic* будет содержать для каждой вершины её *id*, расстояние от начальной вершины и расстояние до конечной вершины. Расстояние до конечной вершины рассчитывается в функции *geo_dist* с помощью формул нахождения расстояния между двумя точками Земли. Вершины сортируются с помощью *priority_queue* по правилам алгоритма A^*: сравнение элементов очереди по сумме {отдаление и приближение}. Из очереди всегда рассматривается текущий верхний элемент. В массиве *d* хранится сумма пройдённого пути до вершины i. Если найденный новый путь до неё такой же или больше, то вершина заново не рассматривается. В массиве *prev* для вершины i хранится индекс той вершины, из которой в неё попали. Алгоритм A^* заканчивается тогда, когда мы попали в конечную конечную вершину из запроса, либо когда очередь стала пуста. Далее (если требуется) собирается путь с помощью массива *prev* и выводится ответ.

Ниже будут показаны основные части программы.

Ввод данных и запись в файл *preprocessed_graph*

```
1  struct cord {
2      uint32_t id;
3      double wid;
4      double lon;
5  };
6
7  bool operator < (const cord& a, const cord& b) {
8      return a.id < b.id;
9  }
10
11 struct edge {
12     uint32_t from;
13     uint32_t to;
14 };
15
16 bool operator < (const edge& a, const edge& b) {
17     if (a.from == b.from) {
18         return a.to < b.to;
19     }
20     else {
21         return a.from < b.from;
22     }
23 }
24
25 void r_and_w_nodes(FILE* f_nodes, FILE* f_out, vector<uint32_t>& ids) {
```

```
    vector<cord> cords;
    cord a;
    while (fscanf(f_nodes, "%u%lf%lf", &a.id, &a.wid, &a.lon) == 3) {
        cords.push_back(a);
    }
    sort(cords.begin(), cords.end());
    uint32_t n = cords.size();
    fwrite(&n, sizeof(uint32_t), 1, f_out);
    for (uint32_t i = 0; i < n; i++) {
        fwrite(&cords[i].id, sizeof(uint32_t), 1, f_out);
        fwrite(&cords[i].wid, sizeof(double), 1, f_out);
        fwrite(&cords[i].lon, sizeof(double), 1, f_out);
        ids.push_back(cords[i].id);
    }
}

uint32_t binary_search(const vector<uint32_t>& ids, const uint32_t& node_id) {
    uint32_t l = -1;
    uint32_t r = ids.size();
    while (r - l > 1) {
        uint32_t m = (l + r) / 2;
        if (ids[m] < node_id) {
            l = m;
        }
        else {
            r = m;
        }
    }
    return r;
}

void r_and_w_edges(FILE* f_edges, FILE* f_out, const vector<uint32_t>& ids) {
    uint32_t n;
    uint32_t c;
    edge a;
    vector<edge> edges;
    while (fscanf(f_edges, "%u", &n) > 0) {
        fscanf(f_edges, "%u", &a.from);
        for (uint32_t i = 1; i < n; i++) {
            fscanf(f_edges, "%u", &a.to);
            edges.push_back(a);
            c = a.from;
            a.from = a.to;
            a.to = c;
            edges.push_back(a);
        }
    }
    sort(edges.begin(), edges.end());
    c = 0;
```

```
    n = edges.size();
    for (uint32_t i = 0; i < n; i++) {
        a = edges[i];
        fwrite(&a.to, sizeof(uint32_t), 1, f_out);
        i++;
        while (i < n && edges[i].from == a.from) {
            fwrite(&edges[i].to, sizeof(uint32_t), 1, f_out);
            i++;
        }
        i--;
    }
    fwrite(&c, sizeof(uint32_t), 1, f_out);
    //offsets
    uint32_t j = 0;
    n = ids.size();
    c = edges.size();
    for (uint32_t i = 0; i < n; i++) {
        while (j < c && binary_search(ids, edges[j].from) < i) {
            j++;
        }
        fwrite(&j, sizeof(uint32_t), 1, f_out);
    }
}

void preprocess_graph(char* nodes_name, char* edges_name, char* out_name) {
    FILE* f_nodes = fopen(nodes_name, "r");
    if (f_nodes == NULL) {
        fprintf(stderr, "Something wrong with nodes file\n");
    }
    FILE* f_edges = fopen(edges_name, "r");
    if (f_edges == NULL) {
        fprintf(stderr, "Something wrong with edges file\n");
    }
    FILE* f_out = fopen(out_name, "wb");
    if (f_out == NULL) {
        fprintf(stderr, "Something wrong with output file\n");
    }
    vector<uint32_t> ids;
    r_and_w_nodes(f_nodes, f_out, ids);
    r_and_w_edges(f_edges, f_out, ids);
    fclose(f_nodes);
    fclose(f_edges);
    fclose(f_out);
}
```

Рассчёт расстояния и структура *euristic*

```
const uint32_t GEODATA_SIZE = sizeof(uint32_t) + 2 * sizeof(double);

const double EARTH_R = 6371e3;
```

```
4  const double MAX_ANGLE = 180.0;
5  const double PI = 4 * atan(1);
6  const double EPS = 1e-6;
7
8  double rad(double a) {
9      return a * PI / MAX_ANGLE;
10 }
11
12 double geo_dist(const cord& a, const cord& b) {
13     double phi_a = rad(a.wid);
14     double phi_b = rad(b.wid);
15     double delta = rad(a.lon - b.lon);
16     double cos_d = sin(phi_a) * sin(phi_b) + cos(phi_a) * cos(phi_b) * cos(delta);
17     double d = acos(cos_d);
18     if (isnan(d)) {
19         return 0;
20     }
21     return EARTH_R * d;
22 }
23
24 struct euristic {
25     uint32_t id;
26     double dist;
27     double path;
28
29     euristic(const uint32_t& u_id, const cord& u, const cord& v, const double& path_to)
           {
30         id = u_id;
31         dist = geo_dist(u, v);
32         path = path_to;
33     }
34 };
35
36 bool operator < (const euristic& a, const euristic& b) {
37     if (abs((a.path + a.dist) - (b.path + b.dist)) > EPS) {
38         return b.path + b.dist < a.path + a.dist;
39     }
40     else if (abs(a.path - b.path) > EPS) {
41         return b.path < a.path;
42     }
43     else {
44         return a.id < b.id;
45     }
46 }
```

Алгоритм A^*

```
1  void push_nodes(FILE* f_graph, cord end, uint32_t cur_ind, const vector<uint32_t>& ids
       , const vector<uint32_t>& offsets, vector<uint32_t>& prev, vector<double>& d,
       priority_queue<euristic>& q) {
```

```
    cord cur_cord = get_cord(f_graph, cur_ind);
    uint32_t cur_offset = offsets[cur_ind];
    vector<uint32_t> available_nodes;
    fseek(f_graph, sizeof(uint32_t) + GEODATA_SIZE * (ids.size()) + cur_offset * sizeof
        (uint32_t), 0);
    uint32_t a;
    for (uint32_t i = cur_offset; i < offsets[cur_ind + 1]; i++) {
        fread(&a, sizeof(uint32_t), 1, f_graph);
        available_nodes.push_back(a);
    }
    uint32_t n = available_nodes.size();
    for (uint32_t i = 0; i < n; i++) {
        a = available_nodes[i];
        uint32_t a_ind = binary_search(ids, a);
        cord a_cord = get_cord(f_graph, a_ind);
        double cur_next_dist = geo_dist(cur_cord, a_cord);
        if (d[a_ind] < 0 || (abs(d[a_ind] - (d[cur_ind] + cur_next_dist)) > EPS && d[
            a_ind] > d[cur_ind] + cur_next_dist)) {
            d[a_ind] = d[cur_ind] + cur_next_dist;
            prev[a_ind] = cur_ind;
            q.push(euristic(a, a_cord, end, d[a_ind]));
        }
    }
}

double euristic_find(FILE* f_graph, const vector<uint32_t>& ids, const vector<uint32_t
    >& offsets, const uint32_t& u, const uint32_t& v, vector<uint32_t>& prev, vector<
    double>& d) {
    uint32_t u_ind = binary_search(ids, u);
    cord u_cord = get_cord(f_graph, u_ind);
    uint32_t v_ind = binary_search(ids, v);
    cord v_cord = get_cord(f_graph, v_ind);
    d[u_ind] = 0;
    prev[u_ind] = u_ind;
    priority_queue<euristic> q;
    q.push(euristic(u, u_cord, v_cord, 0));
    while (!q.empty()) {
        euristic cur = q.top();
        q.pop();
        uint32_t cur_ind = binary_search(ids, cur.id);
        if (cur.path > d[cur_ind]) {
            continue;
        }
        if (cur.id == v) {
            break;
        }
       // cout << 1 << '\n';
        push_nodes(f_graph, v_cord, cur_ind, ids, offsets, prev, d, q);
    }
```

```
47      return d[v_ind];
48  }
```

Нахождение пути и вывод в выходной файл

```
1   void get_path(FILE* f_graph, uint32_t node_id, const vector<uint32_t>& prev, vector<
        uint32_t>& res) {
2       while (prev[node_id] != node_id) {
3           cord cur_node = get_cord(f_graph, node_id);
4           res.push_back(cur_node.id);
5           node_id = prev[node_id];
6       }
7       cord cur_node = get_cord(f_graph, node_id);
8       res.push_back(cur_node.id);
9   }
10
11  void execute_search(FILE* f_in, FILE* f_out, FILE* f_graph, const vector<uint32_t>&
        ids, const vector<uint32_t>& offsets, bool ops) {
12      uint32_t n = ids.size();
13      vector<uint32_t> prev(n);
14      vector<double> d(n);
15      uint32_t u = 0;
16      uint32_t v = 0;
17      while (fscanf(f_in, "%u%u", &u, &v) == 2) {
18          prev.assign(n, 0);
19          d.assign(n, -1);
20          double ans = euristic_find(f_graph, ids, offsets, u, v, prev, d);
21          uint32_t v_ind = binary_search(ids, v);
22          if (ops) {
23              if (d[v_ind] < 0) {
24                  fprintf(f_out, "-1 0\n");
25              }
26              else {
27                  fprintf(f_out, "%.6lf ", ans);
28                  vector<uint32_t> path;
29                  get_path(f_graph, v_ind, prev, path);
30                  n = path.size();
31                  fprintf(f_out, "%d ", n);
32                  for (uint32_t i = n - 1; i > 0; --i) {
33                      fprintf(f_out, "%u ", path[i]);
34                  }
35                  fprintf(f_out, "%u\n", path[0]);
36              }
37          }
38          else {
39              if (d[v_ind] < 0) {
40                  fprintf(f_out, "-1\n");
41              }
42              else {
43                  fprintf(f_out, "%.6lf\n", ans);
```

```
            }
        }
    }
}

void search(char* graph_name, char* in_name, char* out_name, bool ops) {
    FILE* f_graph = fopen(graph_name, "rb");
    if (f_graph == NULL) {
        fprintf(stderr, "Something wrong with graph file\n");
    }
    FILE* f_in = fopen(in_name, "r");
    if (f_in == NULL) {
        fprintf(stderr, "Something wrong with input file\n");
    }
    FILE* f_out = fopen(out_name, "w");
    if (f_out == NULL) {
        fprintf(stderr, "Something wrong with output file\n");
    }
    vector<uint32_t> ids;
    r_cords_vec(f_graph, ids);
    uint32_t m = count_edges(f_graph);
    vector<uint32_t> offsets(ids.size() + 1);
    offsets[offsets.size() - 1] = m;
    r_offsets(f_graph, offsets);
    execute_search(f_in, f_out, f_graph, ids, offsets, ops);
    fclose(f_graph);
    fclose(f_in);
    fclose(f_out);
}
```

3 Консоль

```
root@DESKTOP-5HM2HTK:~# ./prog.out preprocess --nodes europe.nodes --edges
europe.edges --output graph
root@DESKTOP-5HM2HTK:~# cat <intput.txt
30 23
15 8
16 5
1 1
10 52
root@DESKTOP-5HM2HTK:~# ./prog.out search --graph graph --input input.txt --output
output.txt --full-output
root@DESKTOP-5HM2HTK:~# cat <output.txt
13784805.044665 2 30 23
19917048.480887 3 15 5 8
```

```
17538214.812683 3 16 15 5
0
-1
root@DESKTOP-5HM2HTK:~#
```

4 Тест производительности

```
root@DESKTOP-5HM2HTK:~# g++ generator.cpp
root@DESKTOP-5HM2HTK:~# ./a.out tests
root@DESKTOP-5HM2HTK:~# g++ benchmark.cpp
root@DESKTOP-5HM2HTK:~# ./a.out <tests/1.t
Number of nodes is: 1000
Number of edges is: 10000
A star time: 41ms
Dijkstra's algorithm time: 397ms
root@DESKTOP-5HM2HTK:~#
```

Как видно, алгоритм A^* работает почти в десять раза быстрее, чем алгоритм Дейкстры на указанных данных. Такое сильное ускорение получается из-за того, что в алгоритме A^* используется функция $f(u)$, которая позволяет сразу найти направление, в которую следует двигаться. Если на пути не встретиться значительных преград, то данный алгоритм даёт огромное преимущество перед другими алгоритмами.

5 Выводы

Выполнив вторую часть курсового проекта по курсу «Дискретный анализ», я улучшил свои навыки работы с алгоритмом поиска кратчайшего пути в графе – A^*. Также очень полезным опытом для меня было решение задачи оптимизации памяти. С такой проблемой я встретился впервые, поэтому было весьма непросто найти оптимальное и эффективное решение. Попрактиковался работать с файлами, вспомнил функции взаимодействия с ними. Реализовывать данный алгоритм было интересно и полезно для решения будущих задач.

Список литературы

[1] *Алгоритм A* — Итмо-вики.*
URL: https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_A* (дата обращения: 03.01.2023).

[2] *Работа с файлами в языке Си.*
URL: https://prog-cpp.ru/c-files/?ysclid=ld3nvdvwz4136288578 (дата обращения: 03.01.2023).

[3] *Расстояния между точками по координатам.*
URL: https://gpscool.ru/koordinaty/opredelenie-rasstoyaniya-mezhdu-tochkami-po-koordinatam (дата обращения: 04.01.2023).