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. О. Ларченко
Преподаватель: С. А. Сорокин
Группа: М8О-206Б

Дата:
Оценка:
Подпись:

Москва, 2024

Курсовой проект

B. Эвристический поиск на графах

Задача: Реализуйте алгоритм A* для неориентированного графа. Ваша программа должна читать входные данные из стандартного потока ввода и выводить ответ на стандартный поток вывода. Расстояние между соседями вычисляется как простое евклидово расстояние на плоскости.

Форма ввода В первой строке вам даны два числа n и m ($1 <= n <= 10^4, 1 <= m <= 10^5$) - количество вершин и рёбер в графе. В следующих n строках вам даны пары чисел x y ($-10^9 <= x, y <= 10^9$), описывающие положение вершин графа в двумерном пространстве. В следующих m строках даны пары чисел в отрезке от 1 до n, описывающие рёбра графа.Далее дано числои в следующих $q(1 <= q <= 300)$ строках даны запросы в виде пар чисел a, b ($1 <= a, b <= n$) на поиск кратчайшего пути между двумя вершнами.

Форма вывода В ответ на каждый запрос выведите единственное число — длину кратчайшего пути между заданными вершинами с абсолютной либо относительной точностью 10^{-6}, если пути между вершинами не существует выведите -1.

1 Описание

Алгоритм A* (англ. A star) — алгоритм поиска, который находит во взвешенном графе маршрут наименьшей стоимости от начальной вершины до выбранной конечной.[1]

Часто его сравнивают с Поиском в ширину и Алгоритмом Дейкстры, т.к. в общих чертах они решают похожую задачу - поиск кратчайшего пути, но в каждом из алгоритмом эта задача поставлена по-разному.
Поиск в ширину (BFS). Этот алгоритм предназначен для поиска кратчайшего пути в графе (длиной пути в этом случа является количество ребер при обходе). Этот алгоритм обходит все возможные пути и находит минимальный по длине. Если представлять его визуализацию на сетке, то запустив его из любой точки, он будет равномерно расширяться (как заливка).
Алгоритм Дейкстры(Dijkstra). Можно считать улучшенной версией BFS, в отличие от которой он находит все кратчайшее расстояние от заданной вершины до всех (длинной пути в данном случае уже являются веса). По визуализации его поиска пути на сетке чем-то напоминает BFS. Но в отличие от BFS использует приоритетную очередь, где приоритетом является кратчайшее расстояние от начальной вершины до текущей.
Алгоритм A*. Является моификацией алгоритма Дейкстры. Решает задачу поиска минимального рассояния из начальной вершины до заданной (а не для всех). Его преимущество заключается в том, к подлинному расстоянию до объекта (как в Дейкстре) добавляется еще рассчет оценочного расстояния до объекта, что позволяет алгоритму сразу рассматривать наиболее перспективные и теоретически наиболее выгодные маршруты для достижения указанной вершины.[3]
Для рассчета оценочного рассояния выбирается эвристическая функция. Вот самые применяемые:

1. Манхэттенское расстояние (перемещение в четырех направлениях).
   $h(v) = |trget.x - v.x| + |trget.y - v.y|$
2. Расстояние Чебышева (четыре направленя + диагонали)
   $h(v) = max(|trget.x - v.x|, |trget.y - v.y|)$
3. Евклидово расстояние (передвижение не ограничено сеткой)
   $h(v) = \sqrt{(trget.x - v.x)^2 + (trget.y - v.y)^2}$

, где target - искомая вершина, v - текущая вершина, h(v) - расстояние от текущей вершины до искомой.
В текущей постановке задачи было сказано использвать евклидово расстояние.

Алгоритм: Алгоритм на вход получает два параметра start и finish(start - индекс исходной вершины, из которой начинается обход; finish - индекс вершины, расстояние до которой нужно найти кратчайший путь). Затем алгоримт добавляет вершину start в очередь с приоритетом (где приоритетом является наименьшее оценочное расстояние до вершины finish) и запускает цикл, пока очередь не пуста, либо пока не будет извлечен элемент равный finish, в таком случае мы выходим из цикла и возвращаем подлинное расстояние пройженное до данной вершины. Иначе, если извлеченный элемент не является искомым, то запускается цикл проверки всех его смежных вершин, и проверяется 2 условия: 1) Если вершина уже была посещена, и новое расстояние от текущей вершины до следущей > текущего расстояня до вершины записанного в массив path[v] (v - смежная с текущей вершина), то пропускаем данную вершину, при обратном исходе сравнения обновляем значение path[v] и добавляем вершину v в очередь с приоритетом, рассчитав его по эврестической функции.

Оценка сложности.
Временная оценка зависит от выбранной эвристики. Худшая оценка сложности - экспоменциальная - ($O(n^d)$), где n - количество вссех вершин.
Лучша оценка - линейная ($O(d)$). Достигается, когда пространство посика является деревом, а эвристика удовлетворяет следующему условию: $|h(x) - h^*(x)| <= O(logh^*(x))$, где h^* - оптимальная эвристика, то есть точная оценка расстояния из вершины x к цели. Другими словами, ошибка h(x) не должна расти быстрее, чем логарифм от оптимальной эвристики [2].
d - глубина оптимального решения.

2 Исходный код

Я решил не выносить функцию для A* в отдельную функцию, поэтому вся основная логика вместе с обработкой входных данных представленна в функции $main()$.

```
int main(){
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    uint n, m;
    cin>>n>>m;
    vector<pair<int,int>> coord(n+1);
    vector<vector<my_pair>> graph(n+1);
    for(uint i=1;i<=n;++i){
        cin>>coord[i].first>>coord[i].second;
    }
    for(uint i=0; i<m;++i){
        uint v1,v2;
        cin>>v1>>v2;
        double dist = dist_calc(coord[v1], coord[v2]);
        graph[v1].push_back(my_pair(dist, v2));
        graph[v2].push_back(my_pair(dist, v1));
    }

    uint q;
    cin>>q;
    for(int i=0; i<q;++i){
        priority_queue<my_pair, vector<my_pair>, compression_class> PQ;
        vector<double> path(n+1, -1);
        uint start,finish;
        cin>>start>>finish;

        path[start] = 0;

        PQ.push(my_pair(0,start));
        while(!PQ.empty()){
            my_pair cur_val = PQ.top();
            if (cur_val.idx == finish) break;
            PQ.pop();

            for(int j=0; j< graph[cur_val.idx].size();++j){
                uint next_val = graph[cur_val.idx][j].idx;
                double next_path = path[cur_val.idx] + graph[cur_val.idx][j].dist;
                if (path[next_val] > 0 and next_path > path[next_val] ) continue;

                if (path[next_val] < 0 or next_path < path[next_val] ){
                    path[next_val] = next_path;
                    double evristic_dist = next_path + dist_calc(coord[next_val], coord[
                        finish]);
                    PQ.push(my_pair(evristic_dist, next_val));
```

```
                }
            }
        }
        if (path[finish] < 0) {
            cout<<-1<<'\n';
        } else{
            cout<< fixed <<setprecision(7)<<path[finish]<<'\n';
        }
    }
}
```

Для вычисления эвристики используется функция *dist_calc*, которое вычисляет евклидово расстояние по 2 координатам.

```
double dist_calc(pair<int, int> c1, pair<int,int> c2){ //heuristic approximation
    return pow(pow((c2.first - c1.first),2) + pow((c2.second - c1.second),2), 0.5);
}
```

Тут используется вектор *coord*, в котором хранятся исходные координаты вершин. В основной части алгоритма используется вуктор *path*, в котором записывается кратчайшее расстояние от исходной вершины до искомой.

Кроме того, в программе фигурирует структура *my_pair*. Она используется для хранения информации о графе (хранит инжекс вершины и расстояние до нее), а кроме того она используется в очереди с приоритетом и собственно класс *compression_class* используется для расстановки приоритета.

```
struct my_pair{
    double dist;
    uint idx;
    my_pair(){
        dist = -1;
        idx =0;
    }
    my_pair(double new_dist, uint new_idx){
        dist = new_dist;
        idx = new_idx;
    }
    bool operator() ( my_pair el1, my_pair el2){
        return el1.dist > el2.dist;
    }
};

class compression_class {
public:
    bool operator() (my_pair el1, my_pair el2) {
        return el1.dist > el2.dist;
    }
};
```

3 Консоль

```
arsenii@PC-Larcha14:~/Documents/C_pp_uk/DA/kp$ g++ main.cpp -o main
arsenii@PC-Larcha14:~/Documents/C_pp_uk/DA/kp$ ./main
4 5
0 0
1 1
-1 1
0 2
1 2
1 3
2 4
3 4
1 4
1
1 4
2.0000000
```

4 Тест производительности

В процессе тестирования будем сравнивать наш алгоритм A* с Дейкстрой. Для тестирования рассмотрим насыченные графы, разряженные, а также посмотрим на время работы алгоритма при поиске пути к элементу, связи с которым нет.

```
arsenii@PC-Larcha14:~/Documents/C_pp_uk/DA/kp$ g++ benchmark.cpp -o benchmark
arsenii@PC-Larcha14:~/Documents/C_pp_uk/DA/kp$ ./benchmark <tests/1.txt
n=100 m= 100
Dijkstra : 0.000215699 s
A*       : 0.000174851 s
Answ = -1.000000000
Dijkstra : 0.000147483 s
A*       : 0.000013630 s
Answ = 1919.209301914
Dijkstra : 0.000144373 s
A*       : 0.000024578 s
Answ = 3372.982540108
arsenii@PC-Larcha14:~/Documents/C_pp_uk/DA/kp$ ./benchmark <tests/2.txt
n = 100 m = 150
Dijkstra : 0.000205478 s
A*       : 0.000169714 s
Answ = 5392.782457107
Dijkstra : 0.000205786 s
A*       : 0.000026714 s
Answ = 1685.508460999
Dijkstra : 0.000204624 s
A*       : 0.000096018 s
Answ = 3000.450982259
arsenii@PC-Larcha14:~/Documents/C_pp_uk/DA/kp$ ./benchmark <tests/3.txt
n = 1000 m =4000
Dijkstra : 0.000797911 s
A*       : 0.000259356 s
Answ = 2670.835378246
Dijkstra : 0.000821409 s
A*       : 0.000044088 s
Answ = 1676.121456541
Dijkstra : 0.000794212 s
A*       : 0.000005440 s
Answ = 1911.921546508
```

Как можно заметить, алгоритм A* на всех тестах показывает себя лучше, собственно, так и должно быть.

5 Выводы

Выполнив данный курсовой проект, я познакомился с алгоритмм A*(a star), про который, признаться честно, раньше не слышал. Этот Алгорим берет все лучшее из алгоритма Дейкстры и улучшает его путем добавлем эвристик, что делает его эфективным в поиске пути по 2 заданным точкам. Благодаря своей эффективности алгоритм находит свое применение в разработке компьютерных игр. Кроме того, данный алгоритм имеет несколько модификаций, которые делают алгоритм более эффективным с точки зрения потребления памяти.

Подводя итоги, я могу сказать, что я изучил Алгоритм A* и смог реализовать исправно-работающую программу, поэтому считаю, что успешно справился с поставленной задачей.

Список литературы

[1] Алгоритм A* [Электронный ресурс]: Институт Точной Механики и Оптики - URL https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_A* (дата обращения: 25.12.2024)

[2] A* [Электронный ресурс]: Википедия - URL https://ru.wikipedia.org/wiki/A* (дата обращения: 25.12.2024)

[3] Введение в алгоритм A* [Электронный ресурс]: Хабр - URL https://habr.com/ru/articles/331192/ (дата обращения: 25.12.2024)