Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Курсовой проект по курсу «Дискретный анализ»

Студент: С. Я. Симонов
Преподаватель: Н. А. Зацепин
Группа: М8О-306Б
Дата:
Оценка:
Подпись:

Москва, 2020

Задача

Эвристический поиск в графе.
Требуется реализовать алгоритм A* для неориентированного графа.

1 Метод решения

Алгоритм A* — алгоритм поиска, который находит во взвешенном графе маршрут наименьшей стоимости от начальной вершины до выбранной конечной. В процессе работы алгоритма для вершин рассчитывается функция $f(v) = g(v) + h(v)$, где:

1. $g(v)$ — наименьшая стоимость пути в v из стартовой вершины
2. $h(v)$ — эвристическое приближение стоимости пути от v до конечной цели

Фактически, функция $f(v)$ — длина пути до цели, которая складывается из пройденного расстояния $g(v)$ и оставшегося расстояния $h(v)$. Исходя из этого, чем меньше значение $f(v)$, тем раньше мы откроем вершину v, так как через неё мы предположительно достигнем расстояние до цели быстрее всего. Открытые алгоритмом вершины можно хранить в очереди с приоритетом по значению $f(v)$.

2 Описание

В первой строке программы подаются два числа n и m – колличество вершин и ребер в графе. В следующих n строках подаются пары чисел, описывающие положение вершин графа в двумерном пространстве. В следующих m строках подаются пары чисел в отрезке от 1 до n, описывающие ребра графа.

Далее подается число q и в следующих q строках даны запросы в виде пар чисел на поиск кратчайшего пути между двумя вершинвми.

В ответ на каждый запрос выводится единственное число – длинна кратчайшего пути между заданными вершинами, с точностью до 10^{-6}, если если пути между вершинами не сушествует выводится «-1» . Расстояние между соседями вычисляется как простое евклидово растояние на плоскости.

3 Код программы

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>
#include <queue>
#include <unordered_map>
#include <iomanip>

double Heuristic(double x1, double y1, double x2, double y2) {
    return sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
}

std::vector<std::pair<std::pair<double, double>, double>>
                GetDistance(std::vector<std::pair<double, double>>& nodes,
                            std::vector<std::pair<double, double>>& pointers) {
    std::vector<std::pair<std::pair<double, double>, double>> result;
    for (int i = 0; i < nodes.size(); ++i) {
        double length = Heuristic(pointers[nodes[i].first - 1].first,
                                  pointers[nodes[i].first - 1].second,
                                  pointers[nodes[i].second - 1].first,
                                  pointers[nodes[i].second - 1].second);
        result.push_back(std::make_pair(std::make_pair(nodes[i].first, nodes[i].second), length));
    }
    return result;
}

std::vector<std::pair<double, double>> GetPoint(std::vector<std::pair<std::pair<double, double>,
double>>& length, double s) {
    std::vector<std::pair<double, double>> result;
    for (int i = 0; i < length.size(); ++i) {
        if (length[i].first.first == s) {
            result.push_back(std::make_pair(length[i].first.second, length[i].second));
        }
        if (length[i].first.second == s) {
            result.push_back(std::make_pair(length[i].first.first, length[i].second));
        }
    }
    return result;
}

std::vector<double> AStar(std::vector<std::pair<double, double>>& serch,
                          std::vector<std::pair<std::pair<double, double>, double>>& length,
                          std::vector<std::pair<double, double>>& pointers,
                          std::vector<std::pair<double, double>>& nodes) {
    std::vector<double> res;
    for (int j = 0; j < serch.size(); ++j) {
        double start = serch[j].first, finish = serch[j].second;

        std::unordered_map <double, double>   distance;
        std::unordered_map <double, double> parent;

        std::priority_queue < std::pair<double, double>, std::vector<std::pair <double, double>>,
        std::greater <std::pair <double, double>>> PQ;

        std::unordered_map <double, double>::iterator it_nodes;

        PQ.emplace(0, start);
```

```
        distance[start] = 0;

        double current_id = start;

        while (!PQ.empty()) {
            current_id = PQ.top().second;
            PQ.pop();

            if (current_id == finish) {
                res.push_back(distance[current_id]);
            }
            std::vector<std::pair<double, double>> get = GetPoint(length, current_id);
            for (double i = 0; i < get.size(); i++) {
                double h = Heuristic(pointers[current_id - 1].first, pointers[current_id - 1].second,
                                     pointers[finish - 1].first, pointers[finish - 1].second);
                double score = distance[current_id] + get[i].second;
                it_nodes = distance.find(get[i].first);
                if (it_nodes != distance.end() && score >= distance[get[i].first]) {
                    continue;
                } else if (score < distance[get[i].first] ||  it_nodes == distance.end()) {
                    parent[get[i].first] = current_id;
                    distance[get[i].first] = score;
                    PQ.emplace( score + h, get[i].first );
                }
            }
        }
        if (res.size() != j + 1) {
            res.push_back(-1);
        }
    }
    return res;
}

int main() {
    int n, m, q;
    std::cin >> n >> m;

    std::vector<std::pair<double, double>> pointers;
    std::vector<std::pair<double, double>> nodes;
    std::vector<std::pair<double, double>> serch;

    for (int i = 0; i < n; ++i) {
        std::pair<double, double> k;
        std::cin >> k.first >> k.second;
        pointers.push_back(std::make_pair(k.first, k.second));
    }

    for (int i = 0; i < m; ++i) {
        std::pair<double, double> k;
        std::cin >> k.first >> k.second;
        nodes.push_back(std::make_pair(k.first, k.second));
    }

    std::cin >> q;

    for (int i = 0; i < q; ++i) {
        std::pair<double, double> k;
        std::cin >> k.first >> k.second;
        serch.push_back(std::make_pair(k.first, k.second));
    }

    std::vector<std::pair<std::pair<double, double>, double>> length = GetDistance(nodes, pointers);
```

```
    std::vector<double> ress = AStar(serch, length, pointers, nodes);

    std::cout << std::endl;

    for (int i = 0; i < ress.size(); ++i) {
        std::cout << std::setprecision(7) << ress[i] << std::endl;
    }

    return 0;
}
```

4 Вывод

Алгоритм A * — это мощный алгоритм в сфере ИИ с широким спектром применения. Это самый популярный способ для нахождения кратчайшего пути, так как система реализации чрезвычайно гибкая. Также хочется отметить, что этот алгорит применятся в области разработки компьютерных игр и обучения. Он представляет собой доработанный алгоритм Дейкстры поиска кратчайшего пути в графе и имеет значительное преимущество перед алгоритмом Дейкстры при выборе следующей рассматриваемой вершины.