МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»
Кафедра №806 «Вычислительная математика и программирование»

Курсовой проект
по курсу «Дискретный анализ»

Эвристический поиск на графах

Выполнил: Ширяев Н. А.
Группа: 8О-308Б:
Преподаватель: Макаров Н. К.
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2024

УСЛОВИЕ

Ограничение времени	1.8 секунд
Ограничение памяти	256Mb
Ввод	стандартный ввод или input.txt
Вывод	стандартный вывод или output.txt

Ваша программа должна читать входные данные из стандартного потока ввода и выводить ответ на стандартный поток вывода. Реализуйте алгоритм A * для неориентированного графа.

Расстояние между соседями вычисляется как простое евклидово расстояние на плоскости.

Формат ввода

В первой строке вам даны два числа n и m ($1 \le n \le 10^4, 1 \le m \le 10^5$) — количество вершин и рёбер в графе. В следующих n строках вам даны пары чисел xy ($-10^9 \le x, y \le 10^9$), 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 число q ($1 \le q \le 300$) и в следующих q строках даны запросы в виде пар чисел ab $1 \le a, b \le n$ на поиск кратчайшего пути между двумя вершинами.

Формат 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.

МЕТОД РЕШЕНИЯ

Теория

Для нахождения кратчайшего пути на графе, где у каждой вершины есть координаты (например, на плоскости), удобен алгоритм A * (A-star). Это алгоритм поиска в ширину (BFS), улучшенный эвристикой, как в жадном поиске (Greedy Search), а именно:

- в отличие от обычного алгоритма Дейкстры, где мы используем только стоимость пути $g(n)$ от стартовой вершины до текущей, в A * добавляется ещё и эвристическая функция $h(n)$, которая оценивает расстояние от текущей вершины до целевой (например, прямую Евклидову дистанцию);

- при выборе следующей вершины для посещения алгоритм учитывает суммарную оценку $f(n) = g(n) + h(n)$, где $g(n)$ — цена (или длина) пути от начальной вершины a до вершины n, а $h(n)$ — эвристическая оценка расстояния от n до целевой вершины b.

Благодаря этому A * обычно обходит существенно меньше вершин, чем алгоритм Дейкстры, поскольку “идёт” примерно в направлении целевой точки, не расширяя все возможные пути равномерно.

При этом важно, чтобы эвристика удовлетворяла условию аддитивности (не переоценивала расстояние), иначе алгоритм может перестать гарантировать поиск кратчайшего пути. Евклидова метрика, которую мы используем, — классический пример допустимой эвристики, если граф описывает движение на плоскости.

В общем случае алгоритм A * имеет асимптотику, близкую к $O((V + E)logV)$ — аналогично Дейкстре, так как базовая структура данных (приоритетная очередь) идентична. Но благодаря эвристике и тому, что мы чаще отбрасываем неэффективные пути, алгоритм на практике часто работает быстрее.

Применение теории к нашему алгоритму

В задаче у нас есть граф из n вершин и m ребер. Каждая вершина имеет координаты (x_i, y_i). Для нескольких пар вершин (a, b) нужно найти кратчайшее расстояние на графе между ними.

1) Чтение данных и хранение графа:

- считываем координаты каждой вершины в массив $coords$;

- для представления графа используем список смежности adj, где для каждой вершины храним все рёбра (с конечной вершиной to и весом w);

- вес ребра рассчитывается как Евклидово расстояние между двумя вершинами по формуле $\sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}$.

2) Эвристика (евклидова метрика)

- функция $euclid(u, b)$ возвращает расстояние между вершинами u и b. Она используется в вычислении $h(n)$ — эвристики, чтобы оценить, насколько "далеко" мы находимся от цели b.

3) Основной цикл обработки запросов

- для каждого запроса (a, b) запускаем A * из вершины a в вершину b;

- если $a = b$, то расстояние равно 0.

4) Алгоритм A *

- инициализируем вектор расстояний $dist$, где $dist[u]$ — текущее известное кратчайшее расстояние от a до вершины u. Сначала все значения равны бесконечности, кроме $dist[a] = 0$;

- заводим приоритетную очередь pq, где будем хранить состояние (v, f), где v — номер вершины, а $f = g(v) + h(v)$. Изначально в очередь помещаем $(a, euclid(a, b))$.

- пока очередь не пуста:

1) Извлекаем из pq вершину v с минимальным значением f;

2) Если мы уже посещали v, то пропускаем ее, иначе помечаем как посещенную;

3) Если $v = b$, значит нашли кратчайшее расстояние $dist[b]$, выходим из цикла;

4) Перебираем все ребра $(v \to u)$. Пусть вес этого ребра w:

- если $dist[v] + w < dist[u]$, улучшаем расстояние до u: $dist[u] \leftarrow dist[v] + w$;

- далее вычисляем новый приоритет $f_u = dist[u] + euclid(u, b)$ и помещаем (u, f_u) в очередь, если вершина еще не посещена.

5) Вывод результата

- если после обхода вершину b так и не встретили, значит путь не существует, выводим -1. Иначе выводим $dist[b]$.

Таким образом, мы используем классический A * с эвристикой в виде Евклидова расстояния между вершинами. Это позволяет быстро находить кратчайшие пути даже в графах со множеством рёбер, поскольку алгоритм просматривает (“расширяет”) обычно гораздо меньше вершин, чем стандартный алгоритм Дейкстры (особенно когда координаты расположены “удачно” и эвристика хорошо приближает остаток пути).

Итоговая сложность зависит от структуры графа и количества запросов, но в среднем мы получаем эффективное решение, а использование приоритетной очереди позволяет обрабатывать каждую вершину за $O(log\, n)$ в худшем случае. При этом память не становится узким местом, так как мы храним лишь несколько массивов размером $O(n)$ и список смежности $O(m)$.

```cpp
#include <bits/stdc++.h>

struct Edge {
    int to;
    double w;
};

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);

    int n, m;
    std::cin >> n >> m;

    std::vector<std::pair<double, double>> coords(n + 1);
    for (int i = 1; i <= n; i++) {
        std::cin >> coords[i].first >> coords[i].second;
    }

    std::vector<std::vector<Edge>> adj(n + 1);
    adj.reserve(n + 1);
    for (int i = 0; i < m; i++) {
        int a, b;
        std::cin >> a >> b;
        double dx = coords[a].first - coords[b].first;
        double dy = coords[a].second - coords[b].second;
        double dist = std::sqrt(dx * dx + dy * dy);
        adj[a].push_back({b, dist});
        adj[b].push_back({a, dist});
    }

    int q;
    std::cin >> q;

    auto euclid = [&](int u, int v) {
        double dx = coords[u].first - coords[v].first;
        double dy = coords[u].second - coords[v].second;
        return std::sqrt(dx * dx + dy * dy);
    };

    std::cout << std::fixed << std::setprecision(9);
    for (int _i = 0; _i < q; _i++) {
        int a, b;
        std::cin >> a >> b;

        if (a == b) {
            std::cout << 0.0 << "\n";
            continue;
        }
```

```
        std::vector<double> dist(n + 1, std::numeric_limits<double>::infinity());
        std::vector<bool> visited(n + 1, false);
        dist[a] = 0.0;

        struct State {
            int v;
            double f;
            bool operator>(const State &o) const {
                return f > o.f;
            }
        };

        std::priority_queue<State, std::vector<State>, std::greater<State>> pq;
        pq.push({a, euclid(a, b)});

        double ans = -1.0;
        while (!pq.empty()) {
            auto [v, f] = pq.top();
            pq.pop();
            if (visited[v]) {
                continue;
            }
            visited[v] = true;
            if (v == b) {
                ans = dist[b];
                break;
            }
            for (auto &edge : adj[v]) {
                int u = edge.to;
                double nd = dist[v] + edge.w;
                if (nd < dist[u]) {
                    dist[u] = nd;
                    if (!visited[u]) {
                        pq.push({u, nd + euclid(u, b)});
                    }
                }
            }
        }

        if (ans < 0) {
            std::cout << -1 << "\n";
        } else {
            std::cout << ans << "\n";
        }
    }

    return 0;
}
```

ТЕСТ ПРОИЗВОДИТЕЛЬНОСТИ

Для проведения теста производительности, был написан следующий скрипт, который генерирует блоки входных данный и замеряет время выполнения каждого из них:

```
#include <bits/stdc++.h>
#include <chrono>

using namespace std;

struct Edge {
    int to;
    double w;
};


//--------------------------------------
void generateBlock(int n, int m, ostringstream &gen) {
    gen << n << " " << m << "\n";

    for (int i = 1; i <= n; i++) {
        double x = rand() % 10001;
        double y = rand() % 10001;
        gen << x << " " << y << "\n";
    }

    set<pair<int,int>> used;
    for (int i = 0; i < m; i++) {
        while (true) {
            int a = 1 + rand() % n;
            int b = 1 + rand() % n;
            if (a != b) {
                auto p = minmax(a,b);
                if (!used.count({p.first, p.second})) {
                    used.insert({p.first, p.second});
                    gen << a << " " << b << "\n";
                    break;
                }
            }
        }
    }

    int q = 300;
    gen << q << "\n";

    for (int i = 0; i < q; i++) {
        int a = 1 + rand() % n;
        int b = 1 + rand() % n;
        gen << a << " " << b << "\n";
    }
```

```
}

//--------------------------------------
void solveBlock(istream &in, ostream &out) {
    int n, m;
    if (!(in >> n >> m)) {
        return;
    }

    vector<pair<double,double>> coords(n+1);
    for (int i = 1; i <= n; i++) {
        in >> coords[i].first >> coords[i].second;
    }

    vector<vector<Edge>> adj(n + 1);
    adj.reserve(n + 1);
    for (int i = 0; i < m; i++) {
        int a, b;
        in >> a >> b;
        double dx = coords[a].first - coords[b].first;
        double dy = coords[a].second - coords[b].second;
        double dist = std::sqrt(dx * dx + dy * dy);
        adj[a].push_back({b, dist});
        adj[b].push_back({a, dist});
    }

    int q;
    in >> q;

    auto euclid = [&](int u, int v) {
        double dx = coords[u].first - coords[v].first;
        double dy = coords[u].second - coords[v].second;
        return std::sqrt(dx*dx + dy*dy);
    };

    out << fixed << setprecision(9);

    for (int _i = 0; _i < q; _i++) {
        int a, b;
        in >> a >> b;

        if (a == b) {
            out << 0.0 << "\n";
            continue;
        }

        vector<double> dist(n+1, numeric_limits<double>::infinity());
        vector<bool> visited(n+1, false);
        dist[a] = 0.0;

        struct State {
            int v;
            double f;
```

```
            bool operator>(const State &o) const {
                return f > o.f;
            }
        };

        priority_queue<State, vector<State>, greater<State>> pq;
        pq.push({a, euclid(a, b)});

        while (!pq.empty()) {
            auto [v, f] = pq.top();
            pq.pop();
            if (visited[v]) {
                continue;
            }
            visited[v] = true;
            if (v == b) {
                break;
            }
            for (auto &edge : adj[v]) {
                int u = edge.to;
                double nd = dist[v] + edge.w;
                if (nd < dist[u]) {
                    dist[u] = nd;
                    if (!visited[u]) {
                        pq.push({u, nd + euclid(u, b)});
                    }
                }
            }
        }
    }
}

//--------------------------------------
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    srand((unsigned)time(nullptr));

    vector<pair<int,int>> tests = {
        {1000, 2000},
        {2000, 4000},
        {4000, 8000},
        {8000, 16000},
        {16000, 32000},
        {32000, 64000},
        {64000, 128000}
    };

    for (int i = 0; i < (int)tests.size(); i++) {
        int n = tests[i].first;
        int m = tests[i].second;
```

```
        ostringstream gen;
        generateBlock(n, m, gen);
        istringstream in(gen.str());

        auto start_time = chrono::high_resolution_clock::now();
        solveBlock(in, cout);
        auto end_time = chrono::high_resolution_clock::now();
        chrono::duration<double> diff = end_time - start_time;

        cerr << "Test #" << (i+1)
             << " (n=" << n << ", m=" << m << ") time: "
             << diff.count() << "s\n";
    }

    return 0;
}
```

Результаты теста производительности

№	n	m	Время выполнения, с
1	1000	2000	0.0113859
2	2000	4000	0.022591
3	4000	8000	0.0469612
4	8000	16000	0.101723
5	16000	32000	0.231508
6	32000	64000	0.508167
7	64000	128000	1.02663

n и m были подобраны таким образом, чтобы на каждом последующем тесте они увеличивались в два раза, это позволяет проверить заявленную сложность алгоритма — при увеличении n и m в два раза, а также учитывая, что $m \approx 2n$, время выполнения программы должно увеличиваться в два раза:

$$O((n + 2n)log\ n) = O(3n\ log\ n) \sim O(n\ log\ n)$$

Удвоим n, тогда ожидаемый рост составит

$$\frac{2n\ log(2n)}{n\ log\ n} = 2 \cdot \frac{log(2n)}{log\ n} = 2 \cdot \frac{log\ 2+log\ n}{log\ n} = 2 \cdot \left(1 + \frac{1}{log\ n}\right)$$

Чем больше n, тем меньше $\frac{1}{log\, n}$, следовательно, тем коэффициент ближе к двум. q было взято константным — 300 запросов. Как видно из таблицы, время выполнения действительно увеличивается примерно в два раза при каждом последующем тесте, следовательно, заявленная для алгоритма сложность верна.

ВЫВОДЫ

В ходе выполнения данной работы были изучены и реализованы основные принципы эвристического поиска на графах (алгоритм A *). Алгоритм A * продемонстрировал высокую эффективность при поиске кратчайших путей, особенно в случаях, когда координаты вершин позволяют использовать Евклидову метрику в качестве эвристики.

Практическая ценность алгоритма A * подтверждена многими задачами, включая задачи прокладки маршрутов, навигации в робототехнике и компьютерных играх, а также другие задачи, где требуется быстрое нахождение пути в графах с “геометрическим” смыслом (например, на плоскости или в 3D-пространстве).

Результаты проведённых тестов соответствуют теоретическим оценкам сложности $O((n + m)log\, n)$.

Знания, полученные при изучении и реализации эвристического поиска, будут полезны в дальнейшей работе, в том числе при решении различных задач на графах и при оптимизации алгоритмов поиска.