ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 8

«Графы»

Выполнил работу

Смирнов Олег

Академическая группа J3110

Принято

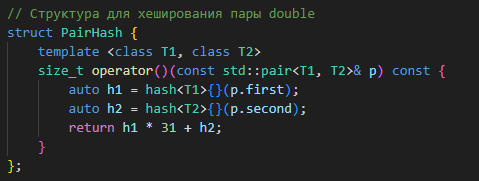
Вершинин Владислав

Санкт-Петербург

2024

1. Подготовка

Переводим граф в вид матрицы смежности, тип данных unordered\_map. Так как пары не могут самостоятельно хэшироваться, применяем простую функцию для хэширования h1 \* 31 + h2.



Проходим по строкам и парсим их

void parseString(const string& input, unordered\_map<pair<double, double>, vector<pair<pair<double, double>, double>>, PairHash>& graph) {

    stringstream ss(input);

    string node\_part;

    getline(ss, node\_part, ':');

    double lon1, lat1;

    if (sscanf(node\_part.c\_str(), "%lf,%lf", &lon1, &lat1) != 2) return;

    string edges\_part;

    getline(ss, edges\_part);

    stringstream edges\_ss(edges\_part);

    string edge\_str;

    while (getline(edges\_ss, edge\_str, ';')) {

        double lon2, lat2, weight;

        if (sscanf(edge\_str.c\_str(), "%lf,%lf,%lf", &lon2, &lat2, &weight) != 3) continue;

        graph[{lon1, lat1}].push\_back({{lon2, lat2}, weight});

        graph[{lon2, lat2}].push\_back({{lon1, lat1}, weight});

    }

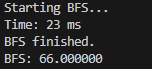
}

1. BFS

Стандартная реализация. Помещаем стартовую вершину в очередь и помечаем её как посещённую. Извлекаем вершины из очереди и обрабатываем их: для каждой вершины находим её непосещённые соседние вершины, добавляем эти вершины в очередь и помечаем как посещённые. Если мы нашли целевую вершину, возвращаем результат. Если очередь пуста и целевая вершина не найдена, значит, путь не существует.

1. int BFS(const unordered\_map<pair<double, double>, vector<pair<pair<double, double>, double>>, PairHash>& graph,
2. const pair<double, double>& start, const pair<double, double>& finish) {
3. if (!graph.count(start) || !graph.count(finish)) return -1;
4. queue<pair<pair<double, double>, int>> q;
5. unordered\_set<pair<double, double>, PairHash> visited;
6. q.push({start, 0});
7. visited.insert(start);
8. while (!q.empty()) {
9. auto [current\_node, depth] = q.front();
10. q.pop();
11. if (current\_node == finish) return depth;
12. if (graph.count(current\_node)) {
13. for (const auto& edge : graph.at(current\_node)) {
14. if (!visited.count(edge.first)) {
15. visited.insert(edge.first);
16. q.push({edge.first, depth + 1});
17. }
18. }
19. }
20. }
21. return -1;  // Если путь не найден
22. }

Время работы и результат



2. DFS

Алгоритм используется для поиска пути (не всегда кратчайшего, есть много контрпримеров) в графе.

int DFS(const unordered\_map<pair<double, double>, vector<pair<pair<double, double>, double>>, PairHash>& graph,

                 const pair<double, double>& start, const pair<double, double>& finish) {

    unordered\_set<pair<double, double>, PairHash> visited;

    stack<pair<pair<double, double>, int>> s;

    s.push({start, 0});

    while (!s.empty()) {

        auto [current, depth] = s.top();

        s.pop();

        if (current == finish) {

            return depth;  // Путь найден, возвращаем глубину (число рёбер)

        }

        if (visited.count(current)){

continue;  // Пропускаем уже посещённые вершины

}

        visited.insert(current);  // Отмечаем вершину как посещённую

        // Если есть рёбра, проходим по ним

        if (graph.count(current)) {

            for (const auto& edge : graph.at(current)) {

                if (!visited.count(edge.first)) {

                    s.push({edge.first, depth + 1});

                }

            }

        }

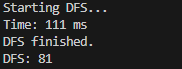
    }

    return -1;  // Путь не найден

}

DFS использует стек для обхода графа. Алгоритм обходит граф вглубь, то есть идёт по одному пути до конца, а затем возвращается и пробует другие возможные пути. Множество visited гарантирует, что каждая вершина будет обработана только один раз, что предотвращает зацикливание в графах с циклами.

Время работы и результат



Как мы видим, нашёлся не кратчайший путь, что весьма логично, так как в графе есть циклы.

3. Алгоритм Дейкстры

double dijkstra(const unordered\_map<pair<double, double>, vector<pair<pair<double, double>, double>>, PairHash>& graph,

              const pair<double, double>& start, const pair<double, double>& finish) {

    if (!graph.count(start) || !graph.count(finish)) return -1;

    unordered\_map<pair<double, double>, double, PairHash> dist;

    priority\_queue<pair<double, pair<double, double>>, vector<pair<double, pair<double, double>>>, greater<pair<double, pair<double, double>>>> pq;

    for (const auto& node : graph) dist[node.first] = numeric\_limits<double>::infinity();

    dist[start] = 0;

    pq.push({0, start});

    while (!pq.empty()) {

        auto [d, u] = pq.top();

        pq.pop();

        if (u == finish) return d;

        if (d > dist[u]) continue;

        if (graph.count(u)) {

            for (const auto& edge : graph.at(u)) {

                if (dist[edge.first] > d + edge.second) {

                    dist[edge.first] = d + edge.second;

                    pq.push({dist[edge.first], edge.first});

                }

            }

        }

    }

    return -1;

}

Ищет длину кратчайшего пути во взвешенном графе

**Инициализация**:

* Если одна из вершин (старт или финиш) отсутствует в графе, возвращается -1 (путь не найден).
* Множество **dist** хранит минимальные расстояния от стартовой вершины до каждой другой вершины, инициализируется значением infinity для всех вершин, кроме стартовой (её расстояние равно 0).
* В **приоритетную очередь** pq помещается стартовая вершина с расстоянием 0.

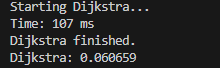
**Основной цикл**:

* Извлекается вершина с минимальным расстоянием из очереди.
* Если текущая вершина — это целевая, то возвращается её расстояние (кратчайший путь найден).
* Если текущее расстояние больше уже найденного для этой вершины, то пропускаем её (это проверка на старые более дорогие пути).
* Для каждой соседней вершины проверяется, можно ли улучшить её расстояние через текущую вершину. Если можно, обновляется расстояние и соседняя вершина добавляется в очередь с новым расстоянием.

**Завершение**:

* Если очередь пуста, значит, путь до целевой вершины не существует (возвращается -1).

Время и результат:



4. Алгоритм A\*

double euclidean\_distance(const pair<double, double>& p1, const pair<double, double>& p2) {

    return sqrt(pow(p2.first - p1.first, 2) + pow(p2.second - p1.second, 2));

}

double astar(const unordered\_map<pair<double, double>, vector<pair<pair<double, double>, double>>, PairHash>& graph,

            const pair<double, double>& start, const pair<double, double>& finish) {

    if (!graph.count(start) || !graph.count(finish)) return -1;

    unordered\_map<pair<double, double>, double, PairHash> g\_cost;

    unordered\_map<pair<double, double>, double, PairHash> f\_cost;

    priority\_queue<pair<double, pair<double, double>>, vector<pair<double, pair<double, double>>>, greater<pair<double, pair<double, double>>>> pq;

    for (const auto& node : graph) {

        g\_cost[node.first] = numeric\_limits<double>::infinity();

        f\_cost[node.first] = numeric\_limits<double>::infinity();

    }

    g\_cost[start] = 0;

    f\_cost[start] = euclidean\_distance(start, finish);

    pq.push({f\_cost[start], start});

    while (!pq.empty()) {

        auto [f, u] = pq.top();

        pq.pop();

        if (u == finish) return g\_cost[u];

        if (graph.count(u)) {

            for (const auto& edge : graph.at(u)) {

                double tentative\_g\_cost = g\_cost[u] + edge.second;

                if (tentative\_g\_cost < g\_cost[edge.first]) {

                    g\_cost[edge.first] = tentative\_g\_cost;

                    f\_cost[edge.first] = tentative\_g\_cost + euclidean\_distance(edge.first, finish);

                    pq.push({f\_cost[edge.first], edge.first});

                }

            }

        }

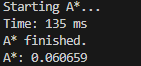
    }

    return -1;

}

1. **Основные компоненты**:
   * **g\_cost** — хранит минимальную стоимость пути от стартовой вершины до текущей вершины (по аналогии с Дейкстрой).
   * **f\_cost** — сумма стоимости пути от старта до текущей вершины (g\_cost) и эвристической оценки расстояния от текущей вершины до финиша. Эвристика используется для направления поиска в сторону целевой вершины.
   * **Эвристическая функция** — оценивает расстояние между текущей вершиной и целевой. Здесь используется евклидова метрика.
2. **Инициализация**:
   * Расстояния всех вершин устанавливаются как бесконечность, кроме стартовой вершины, которая инициализируется как 0.
   * Расстояние до финиша для стартовой вершины оценивается через эвристическую функцию (f\_cost[start] = euclidean\_distance(start, finish)).
   * Стартовая вершина помещается в приоритетную очередь с приоритетом по значению f\_cost.
3. **Основной цикл**:
   * Извлекается вершина с минимальным значением f\_cost (приоритетной очередью), что означает, что эта вершина наиболее перспективна для продолжения поиска.
   * Если текущая вершина совпадает с целевой (finish), то возвращается её стоимость пути g\_cost, что означает нахождение кратчайшего пути.
   * Для каждой соседней вершины вычисляется **tentative g\_cost** (временная стоимость пути через текущую вершину). Если она меньше текущего g\_cost для соседней вершины, то обновляется стоимость и рассчитывается её новый f\_cost, после чего она добавляется в очередь.
4. **Завершение**:
   * Если приоритетная очередь пуста, значит путь не найден, и возвращается -1.

Время и результат



1. Вывод

Длины пути, посчитанные через DFS и BFS оказались различными, так как DFS не всегда находит кратчайший. DFS работает дольше: граф большой, долго идти в глубину, а засчёт ранжирования в BFS мы не посещаем так много вершин.

Длины путей, посчитанные через алгоритм Дейкстры и A\* одинаковые, как и должно быть. Алгоритм Дейкстры работает быстрее: евклидова метрика здесь не дала достаточного прироста производительности, хотя разницу во времени можно объяснить тем, что A\* сначала считал все евклидовы расстояния. Можно попробовать манхэттенскую метрику, что является довольно логичным выбором, если двигаться по, грубо говоря, клетчатой сетке, но в условиях реальной жизни (дороги изогнуты) евклидово расстояние выглядело перспективнее на момент написания кода.