

Thuật toán đường quét

Thuật toán đường quét

Người viết:

- Ngô Nhật Quang - HUS High School for Gifted Students
- Bùi Nguyễn Ngọc Thắng - Carnegie Mellon University in Qatar
- Trần Đình Khánh Dương - Michigan State University

Reviewer:

- Trần Quang Lộc - ITMO University
- Hoàng Xuân Nhật - VNUHCM-University of Science
- Hồ Ngọc Vĩnh Phát - VNUHCM-University of Science
- Lê Minh Hoàng - VNUHCM-University of Science
- Nguyễn Phú Bình - VNUHCM-University of Science

Giới thiệu

Trong bài viết này, chúng ta sẽ đi tìm hiểu về thuật toán đường quét - một thuật toán khá hữu ích trong hình học tính toán. Thuật toán này được xây dựa trên một ý tưởng mạnh mẽ mà đơn giản: sử dụng một đường thẳng dọc và "quét" qua mặt phẳng.

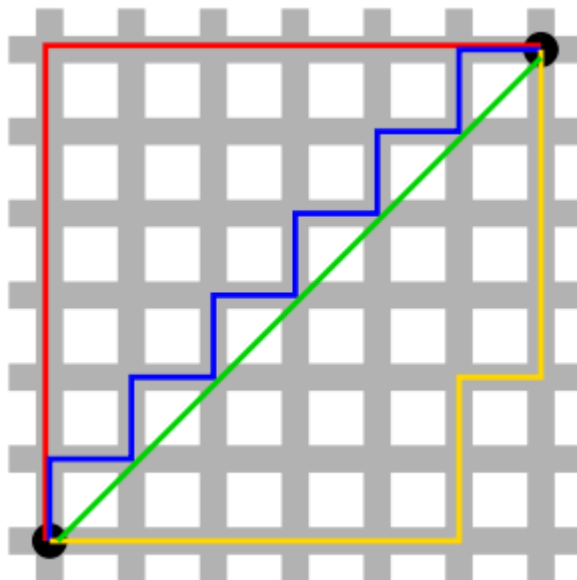
Tuy vậy, trên thực tế, việc "quét" toàn bộ các điểm trên mặt phẳng là bất khả thi nên chúng ta chỉ có thể xét một vài điểm cần thiết mà thôi.

Lưu ý trước khi đọc bài viết

Các khái niệm sau được đề cập xuyên suốt bài viết:

Với hai điểm $P(x_P, y_P)$ và $Q(x_Q, y_Q)$:

- **Khoảng cách Euclid** giữa hai điểm P và Q là khoảng cách thu được khi tính bằng công thức Pythagoras $\sqrt{(x_Q - x_P)^2 + (y_Q - y_P)^2}$
- **Khoảng cách Manhattan** giữa hai điểm P và Q là khoảng cách giữa nếu ta chỉ được đi dọc hoặc ngang khi từ điểm này qua điểm kia, được tính bằng công thức: $|x_Q - x_P| + |y_Q - y_P|$



Các đường màu đỏ, xanh lam, vàng biểu diễn khoảng cách Manhattan có cùng độ dài (12), trong khi đường màu xanh lục biểu diễn khoảng cách Euclid với độ dài $6 * \sqrt{2} \approx 8.48$.

Như đã đề cập ở phía trên, khi ứng dụng thuật toán đường quét, việc quét qua tất cả các điểm trên mặt phẳng là bất khả thi, và chúng ta cần sử dụng một số kỹ thuật hay cấu trúc dữ liệu khác - chẳng hạn, [kỹ thuật hai con trỏ](#), [kỹ thuật nén số](#), [cây phân đoạn](#), [cây Fenwick \(cây chỉ số nhị phân\)](#) - để lọc ra và xử lý các điểm thiết yếu cần quan tâm. Do đó, độc giả nên nắm kĩ các chủ đề liên quan nêu trên trước khi đọc bài viết.

Ngoài ra, bài viết gốc có đề cập tới bài toán bao lồi, nhưng chủ đề này sẽ không được đề cập trong bài viết này bởi VNOI Wiki đã có một bài viết khá hoàn thiện ở [đây](#).

Bài toán tìm cặp điểm gần nhất

Link bài: [SPOJ - CLOPPAIR](#) [🔗](#)

Đề bài

Cho một danh sách n điểm. Tìm khoảng cách Euclid ngắn nhất tạo bởi hai trong số các điểm đó.

Giới hạn:

- $2 \leq n \leq 50000$
- Toạ độ các điểm là số nguyên $-10^6 \leq x, y \leq 10^6$

Phân tích

Ta có thể dễ dàng nhận thấy bài này có thể giải quyết với độ phức tạp $O(n^2)$, nhưng sẽ không thể qua được giới hạn thời gian 1 giây. Tuy vậy, áp dụng thuật toán đường quét, chúng ta có thể giảm độ phức tạp xuống $O(n \log n)$.

Trước tiên, chúng ta sẽ sắp xếp lại danh sách điểm theo thứ tự hoành độ các điểm tăng dần. Lần lượt duyệt qua từng điểm trong danh sách đã sắp xếp. Ý tưởng chính của thuật toán cải tiến so với thuật toán trâu bò đó là thay vì phải duyệt qua tất cả cặp điểm, với mỗi điểm, ta chỉ phải xét điểm đó với một số ít các điểm khác

đáng quan tâm, chi phí để tìm các điểm đáng quan tâm là $O(\log n)$, do đó thuật toán cải tiến có độ phức tạp $O(n \log n)$.

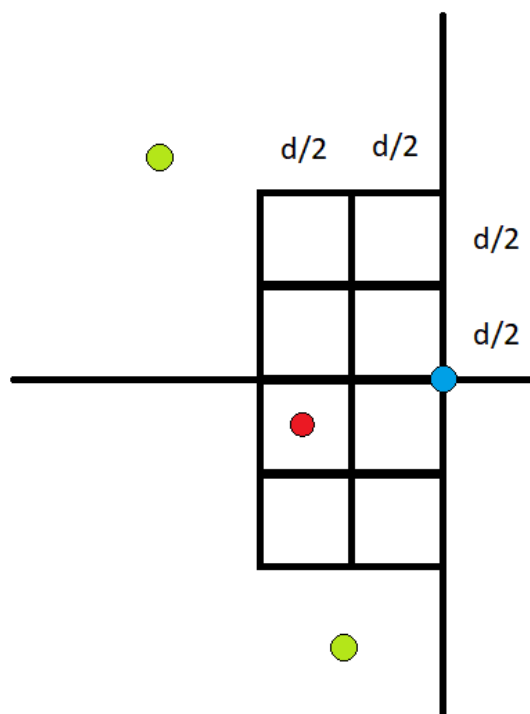
Giả sử chúng ta đã xử lý xong $N - 1$ điểm đầu tiên và khoảng cách ngắn nhất hiện có là d . Như vậy từ điểm thứ N về sau, ta chỉ quan tâm đến các cặp điểm có khoảng cách bé hơn d . Gọi điểm thứ N (cũng là điểm đang xét) là điểm P .

Bổ đề 1

Tại mỗi bước trong thuật toán, để tìm một cặp điểm có khoảng cách bé hơn d , ta chỉ cần quan tâm đến tối đa 8 điểm khác.

Chứng minh

Từ P , vẽ 8 hình vuông xung quanh, mỗi hình vuông có cạnh đúng bằng $d/2$, như hình dưới (điểm màu xanh là P).



Hiển nhiên tất cả các điểm từ 1 đến $N - 1$ đều nằm về phía bên trái của điểm đang xét do ta duyệt danh sách theo thứ tự tăng dần về hoành độ. Nhận xét rằng từ P , ta không cần quan tâm đến những điểm T không nằm trong 8 hình vuông bên trên do khi đó khoảng cách giữa P và T lớn hơn d .

Xét 8 hình vuông ta vừa vẽ. Do d là khoảng cách ngắn nhất giữa hai cặp điểm bất kì trong $N - 1$ điểm đầu tiên nên trong mỗi hình vuông, có nhiều nhất một điểm. Do đó, tối đa ta chỉ cần xét 8 điểm trong 8 hình vuông để cập nhật kết quả.

Từ bổ đề 1, ta nhận thấy cần duy trì một danh sách L các điểm có hoành độ chênh lệch không quá d so với điểm đang xét. Do d giảm dần nên ta có thể thực hiện việc này bằng kĩ thuật hai con trỏ. Đồng thời, tại mỗi bước ta cần tìm các điểm có tung độ lân cận với điểm đang xét. Do đó các điểm trong L cần được sắp xếp

theo thứ tự tung độ tăng dần. Các tiêu chí trên có thể thoả mãn bằng cách cài đặt một cây nhị phân tìm kiếm như `std::set`.

Thuật toán của chúng ta cụ thể như sau. Đầu tiên sắp xếp danh sách điểm theo thứ tự hoành độ tăng dần. Lần lượt duyệt qua các điểm trong danh sách, đồng thời duy trì một cây nhị phân tìm kiếm T . Gọi d là khoảng cách nhỏ nhất giữa hai điểm bất kì đã xét. Tại mỗi bước:

- Loại bỏ khỏi T các điểm có chênh lệch hoành độ so với điểm đang xét lớn hơn d .
- Tìm các điểm trong T có chênh lệch tung độ không quá d , tính khoảng cách giữa các điểm này và điểm đang xét, và cập nhật d .
- Thêm điểm đang xét vào T .

Ta nhận thấy mỗi điểm được thêm vào và xoá khỏi T đúng một lần. Do đó tổng chi phí cho các thao tác thêm và xoá điểm là $O(n \log n)$. Tại mỗi bước, chi phí tìm kiếm là $O(\log n)$ và có $O(1)$ điểm ta cần xét. Tóm lại, độ phức tạp thời gian của thuật toán là $O(n \log n)$.

Cài đặt mẫu

Trong phần cài đặt này, các khoảng cách được lưu dưới dạng bình phương để tránh bị sai số.

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

using namespace std;

#define ll long long

struct Point{
    ll x, y;
    int id;

    bool operator < (const Point& other) {
        if (x != other.x) return x < other.x;
        return y < other.y;
    }
};

struct cmp{
    bool operator () (const Point& a, const Point& b) const {
        if (a.y != b.y) return a.y < b.y;
        return a.x < b.x;
    }
};

int n;
vector<Point> points; // Vector chứa tất cả các điểm
set<Point, cmp> T;

ll squared_dist(Point a, Point b) { // Nhận vào hai điểm, trả về
    // bình phương khoảng cách giữa hai điểm
    return (a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y);
```

```

31 }
32
33 signed main() {
34     ios_base::sync_with_stdio(false); cin.tie(NULL);
35
36     cin >> n;
37     for (int i = 0; i < n; i++) {
38         ll x, y;
39         cin >> x >> y;
40         points.push_back({x, y, i});
41     }
42
43     ll squared_d = squared_dist(points[0], points[1]); // Lưu bình phương của d
44     int res_id1 = 0, res_id2 = 1;
45
46     sort(points.begin(), points.end()); // Sắp xếp các điểm theo hoành độ
47
48     for (auto p : points) {
49         ll x = p.x, y = p.y;
50         int id = p.id;
51
52         ll d = sqrt(squared_d);
53         Point cur = {-1000001, y - d, id};
54
55         while (1) { // Tìm tất cả các điểm có tung độ trong khoảng [y - d, y + d]
56             auto it = T.upper_bound(cur);
57
58             if (it == T.end()) break;
59
60             cur = *it;
61             if (cur.y > y + d) break; // Dừng lại nếu điểm có tung độ lớn hơn y + d
62
63             if (cur.x < x - d) {
64                 T.erase(it);
65                 continue;
66             } // Xóa điểm nếu điểm này có hoành độ bé hơn x - d
67
68
69             if (squared_dist(p, cur) < squared_d) {
70                 squared_d = squared_dist(p, cur);
71                 res_id1 = id; res_id2 = cur.id;
72             } // Gán đáp án mới nếu tìm được d nhỏ hơn
73         }
74
75         T.insert(p); // Thêm điểm hiện tại vào T
76     }
77
78     if (res_id1 > res_id2) swap(res_id1, res_id2);
79     cout << res_id1 << " " << res_id2 << " ";
80     cout << fixed << setprecision(6) << sqrt(squared_d);
81 }

```

Bài toán tìm giao điểm của các đoạn thẳng song song với trục tọa độ

Link bài: [SPOJ - CS345A1](#) 

Đề bài

Cho các đoạn thẳng song song trục Ox hoặc trục Oy , yêu cầu trả lại số các giao điểm.

Giới hạn:

- $1 \leq n \leq 100000$
- Tọa độ x, y của mỗi điểm là số thực thỏa mãn $0 \leq x, y \leq 1$

Phân tích

Ta bắt đầu với ý tưởng quét qua tất cả đoạn thẳng tương tự như bài toán tìm cặp điểm gần nhất.

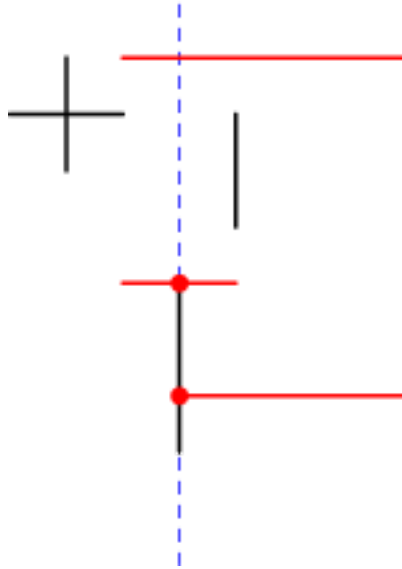
Tuy nhiên, ta sẽ gặp phải vấn đề là một đoạn thẳng nằm ngang sẽ bao gồm các điểm có hoành độ khác nhau nên chúng ta khó có thể sắp xếp các đoạn thẳng theo một thứ tự nào đó liên quan đến hoành độ - giả sử có một đoạn thẳng nối $(0.5, 0.25)$ và $(0.5, 0.75)$ và một đoạn thẳng nối $(0.25, 0.5)$ và $(0.75, 0.5)$, chúng ta nên ưu tiên đoạn thẳng nào trước?

Để giải quyết vấn đề trên, một ý tưởng thường được dùng là thay vì xem một đoạn thẳng nằm ngang là một phần tử duy nhất trong danh sách, ta sẽ tách nó ra thành hai phần tử riêng biệt - một phần tử biểu thị cho đầu mút bên trái và một phần tử biểu thị cho đầu mút bên phải của đoạn thẳng. Như vậy trong danh sách của chúng ta sẽ có ba loại phần tử:

1. Đầu mút bên trái của một đoạn thẳng nằm ngang
2. Đầu mút bên phải của một đoạn thẳng nằm ngang
3. Một đoạn thẳng nằm dọc.

Khi này ta sẽ có thể sắp xếp danh sách của chúng ta theo thứ tự tăng dần về hoành độ của các phần tử.

Chúng ta sẽ quét từ trái sang phải. Khi đoạn quét di chuyển, ta duy trì một tập hợp S các đoạn thẳng nằm ngang bị cắt ngang theo đoạn quét - tức là đoạn quét đang ở giữa hai đầu mút của đoạn ngang đó. Tập này được sắp xếp theo thứ tự y , và được tô màu đỏ trong hình dưới.



Với mỗi đoạn thẳng nằm dọc v_i , để đếm t_i là số đoạn thẳng nằm ngang cắt v_i , ta chỉ việc tìm trong tập S những đoạn thẳng có tung độ nằm giữa hai đầu nút của v . Hiển nhiên ta thấy rằng tổng của các t_i cũng là số giao điểm ta cần tìm.

Như vậy ta thấy tập S là một yếu tố quan trọng để giải quyết bài toán. Tập S cần hỗ trợ các thao tác cụ thể sau:

1. Thêm một phần tử có khoá k vào trong tập (lưu ý các khoá có thể trùng nhau).
2. Xoá một phần tử có khoá k khỏi tập.
3. Cho một khoảng $[L, R]$, trả về số lượng phần tử trong tập có khoá $k \in [L, R]$.

Để thoả mãn các yêu cầu trên, ta sẽ cài đặt tập S bằng kĩ thuật nén số và cây Fenwick. Ta sẽ nén tung độ của các đoạn thẳng thành $O(n)$ điểm. Dựng cây Fenwick dựa trên $O(n)$ điểm này, mỗi nút trong cây cho biết có bao nhiêu đoạn thẳng ngang đang cắt các điểm trong đoạn con mà nút quản lý. Như vậy, mỗi thao tác thêm một đoạn thẳng vào tập S tương ứng với một thao tác cộng 1 vào giá trị lưu ở các nút tương ứng và mỗi thao tác xoá một đoạn thẳng khỏi S tương ứng với một thao tác giảm giá trị lưu ở các nút tương ứng đi 1. Chi phí để thực hiện cả hai thao tác trên là $O(\log n)$. Để tìm số lượng đoạn thẳng ngang có tung độ trong khoảng $[L, R]$, ta có thể tính tổng các điểm trong khoảng $[L, R]$ trong cây Fenwick trong $O(\log n)$. Chi tiết tham khảo ở cài đặt mẫu.

Dễ thấy thuật toán của chúng ta duyệt qua một danh sách có $O(n)$ phần tử, với mỗi phần tử chi phí xử lí là $O(\log n)$. Do đó độ phức tạp thời gian của thuật toán là $O(n \log n)$. Độ phức tạp bộ nhớ là $O(n)$.

Minh họa thuật toán:



Cài đặt mẫu

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

using namespace std;

#define ll long long

const int N = 1e5 + 5;
const double EPS = 1e-9;

struct Event{
    double x;
    int y, y2, type;

    bool operator < (const Event& other) const {
        return x < other.x;
    }
};

struct FenwickTree{
    int n;
    vector<int> s;

    FenwickTree(int n) : n(n), s(n + 5) {
        for (int i = 1; i <= n; i++) s[i] = 0;
    }
};
```



```

void update(int i, int val) {
    for (; i <= n; i += i & -i) s[i] += val;
}

int getsum(int i) {
    int res = 0;
    for (; i; i -= i & -i) res += s[i];
    return res;
}

int query(int l, int r) {
    return getsum(r) - getsum(l - 1);
}
};

int n;
double blue_x1[N], blue_x2[N], blue_y[N], red_y1[N], red_y2[N], red_x[N];
vector<double> compress_y;
vector<Event> events;

signed main() {
    ios_base::sync_with_stdio(false); cin.tie(NULL);

    cin >> n;
    for (int i = 1; i <= n; i++) {
        cin >> blue_x1[i] >> blue_x2[i] >> blue_y[i];

        if (blue_x1[i] > blue_x2[i]) swap(blue_x1[i], blue_x2[i]);
        compress_y.push_back(blue_y[i]);
    }

    for (int i = 1; i <= n; i++) {
        cin >> red_y1[i] >> red_y2[i] >> red_x[i];

        if (red_y1[i] > red_y2[i]) swap(red_y1[i], red_y2[i]);
        compress_y.push_back(red_y1[i]);
        compress_y.push_back(red_y2[i]);
    }

    sort(compress_y.begin(), compress_y.end());
    compress_y.erase(unique(compress_y.begin(), compress_y.end()), compress_y.end());
    // Rời rạc hóa các tung độ

    for (int i = 1; i <= n; i++) {
        int yy = lower_bound(compress_y.begin(), compress_y.end(), blue_y[i]) - compress_y.begin();
        events.push_back({blue_x1[i] - EPS, yy, 0, 1}); // Phần tử loại 1
        events.push_back({blue_x2[i] + EPS, yy, 0, 2}); // Phần tử loại 2
    }

    for (int i = 1; i <= n; i++) {
        int l = lower_bound(compress_y.begin(), compress_y.end(), red_y1[i]) - compress_y.begin();
        int r = lower_bound(compress_y.begin(), compress_y.end(), red_y2[i]) - compress_y.begin();
        for (int j = l; j < r; j++) {
            int id = events[j].id;
            if (id == 0) {
                // Xử lý phần tử loại 0
            }
        }
    }
}

```

```

79 |         int r = lower_bound(compress_y.begin(), compress_y.end(), red_y2[i]) -
80 |         events.push_back({red_x[i], 1, r, 3}); // Phần tử loại 3
81 |     }
82 |
83 |     sort(events.begin(), events.end());
84 |
85 |     FenwickTree FT(n * 3); // Có tối đa n * 3 tung độ khác nhau
86 |     ll res = 0;
87 |     for (auto e : events) {
88 |         if (e.type == 1) FT.update(e.y, 1);
89 |         else if (e.type == 2) FT.update(e.y, -1);
90 |         else res += FT.query(e.y, e.y2);
91 |     }
92 |
93 |     cout << res;
    }

```

Bài toán tìm tổng diện tích phủ bởi các hình chữ nhật

Link bài: [VNOJ - AREA](#) 

Đề bài

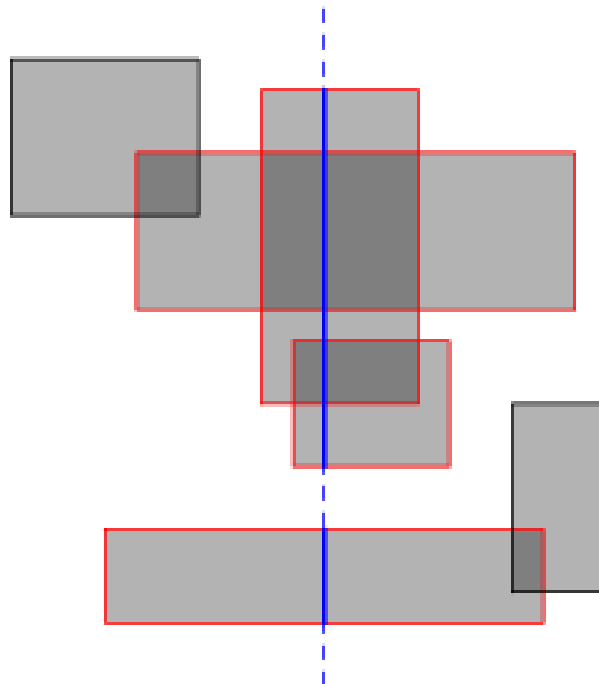
Trên mặt phẳng toạ độ, ta vẽ N hình chữ nhật có các cạnh song song với trục toạ độ. Tính tổng diện tích che phủ bởi N hình chữ nhật này.

Giới hạn:

- $1 \leq n \leq 10000$
- Mỗi hình chữ nhật có toạ độ góc trái dưới và góc phải trên lần lượt là x_1, y_1 và x_2, y_2 sao cho $0 \leq x_1 \leq x_2 \leq 30000$ và $0 \leq y_1 \leq y_2 \leq 30000$

Phân tích

Tương tự như bài toán tìm giao điểm của các đoạn thẳng, chúng ta có thể xử lí bằng cách biểu diễn mỗi hình chữ nhật thành hai "sự kiện" - một biểu thị cho cạnh bên trái và một biểu thị cho cạnh bên phải của hình chữ nhật - và duy trì một tập S chứa các hình chữ nhật mà đoạn thẳng quét của chúng ta đang cắt qua. Khi chúng ta quét qua cạnh bên trái, ta thêm hình chữ nhật đó vào S , khi quét qua cạnh bên phải thì ta xoá hình chữ nhật tương ứng khỏi S .



Ta biết được những hình chữ nhật nào đang bị cắt bởi đường quét của chúng ta (màu đỏ). Để tìm tổng diện tích được bao phủ, ta sẽ tìm diện tích từng phần bị bao phủ giữa mỗi cặp hai "sự kiện" liên nhau và tính tổng của chúng. Để tìm phần diện tích được bao phủ giữa hai sự kiện liên nhau, ta cần biết tổng độ dài phần đường quét đi qua chúng (nét liền màu xanh trong hình trên). Nhân độ dài này với khoảng cách giữa hai sự kiện liên nhau, ta được diện tích của phần hình chữ nhật giữa hai "sự kiện" đó.

Vấn đề đặt ra là làm thế nào để tìm tổng độ dài của các phần "nét liền màu xanh" như trên hình. Nhớ rằng tại mỗi bước ta duy trì một tập S các hình chữ nhật mà đường thẳng quét cắt qua. Hiển nhiên ta thấy tổng độ dài của phần "nét liền màu xanh" chính là hợp của tất cả các hình chữ nhật trong tập S tại mỗi bước.

Để tính hợp của tất cả các hình chữ nhật trong tập S , một thuật toán đơn giản là ta sẽ duyệt qua hết tất cả các hình chữ nhật hiện có trong S . Độ phức tạp thời gian để giải bài toán khi này là $O(n^2)$ - chúng ta duyệt qua $O(n)$ sự kiện, tại mỗi "sự kiện", ta lại duyệt qua $O(n)$ hình chữ nhật mà đường quét của chúng ta đang cắt (đĩ nhiên chúng ta cũng phải thêm các hình mới và xóa bớt những hình mà đường quét không còn cắt khỏi tập S).

Minh họa thuật toán:



Để tối ưu thuật toán, chúng ta có thể sử dụng kĩ thuật nén số và cây phân đoạn. Ta sẽ nén hoành độ của các "sự kiện" thành $O(n)$ điểm, các điểm này chia đường thẳng quét của chúng ta thành $O(n)$ đoạn thẳng con. Ta dựng cây phân đoạn với $O(n)$ đoạn thẳng con này là các nút lá. Tại mỗi nút trong cây phân đoạn ta sẽ lưu hai giá trị để trả lời cho hai câu hỏi:

1. Hiện có bao nhiêu hình chữ nhật đang phủ đoạn con mà nút quản lý?
2. Tổng độ dài các phần được ít nhất một hình chữ nhật phủ trong đoạn con mà nút quản lý là bao nhiêu?

Chi tiết việc cập nhật hai giá trị tại mỗi bước độc giả có thể tham khảo trong cài đặt mẫu.

Cài đặt mẫu

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

using namespace std;

const int MX = 30000;

struct Segment{
    int x, y1, y2, type;

    bool operator < (const Segment& other) const {
        return x < other.x;
    }
};

struct SegmentTree{
    vector<pair<int, int>> s;
```

```

17
18 SegmentTree(int n) : s(n * 4 + 5) {
19     for (int i = 1; i <= 4 * n; i++) s[i] = {0, 0};
20 }
21
22 void update(int id, int l, int r, int tl, int tr, int val) {
23     if (l > tr || r < tl) return;
24     if (l >= tl && r <= tr) {
25         s[id].second += val;
26         if (s[id].second != 0) s[id].first = r - l + 1;
27         else if (l != r) s[id].first = s[id * 2].first + s[id * 2 + 1].first;
28         else s[id].first = 0;
29         return;
30     }
31
32     int m = (l + r) >> 1;
33     update(id * 2, l, m, tl, tr, val);
34     update(id * 2 + 1, m + 1, r, tl, tr, val);
35
36     if (s[id].second != 0) s[id].first = r - l + 1;
37     else s[id].first = s[id * 2].first + s[id * 2 + 1].first;
38 }
39 };
40
41 int n;
42 vector<Segment> segments;
43
44 signed main() {
45     ios_base::sync_with_stdio(false); cin.tie(NULL);
46
47     cin >> n;
48     for (int i = 1; i <= n; i++) {
49         int x1, y1, x2, y2;
50         cin >> x1 >> y1 >> x2 >> y2;
51
52         segments.push_back({x1, y1, y2, 1});
53         segments.push_back({x2, y1, y2, -1});
54     }
55
56     sort(segments.begin(), segments.end());
57
58     SegmentTree ST(MX);
59     int res = 0;
60     for (int i = 0; i < (int)segments.size() - 1; i++) {
61         ST.update(1, 0, MX, segments[i].y1, segments[i].y2 - 1, segments[i].type);
62         res += (segments[i + 1].x - segments[i].x) * ST.s[1].first;
63     }
64     cout << res;
65 }

```

Bài toán tìm cây khung Manhattan nhỏ nhất

Link bài: [Kattis - GRIDMST](#) 

Đề bài

Cho N điểm (có thể trùng nhau). Trọng số giữa mỗi đỉnh là khoảng cách Manhattan giữa chúng. Tìm trọng số của cây khung nhỏ nhất qua N điểm đó.

Giới hạn:

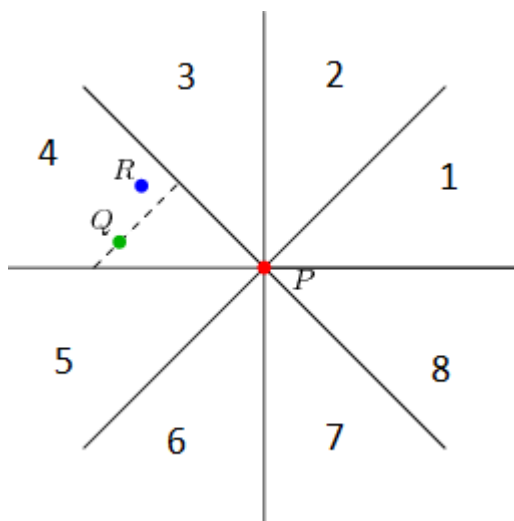
- $1 \leq N \leq 100000$
- N dòng, mỗi dòng 2 số nguyên x, y sao cho $0 \leq x, y \leq 1000$ là tọa độ mỗi điểm.

Phân tích

Ý tưởng chính của thuật giải là nếu như số cạnh ta cần xét là $O(n)$ thì ta có thể sử dụng các thuật toán tìm cây khung nhỏ nhất như Kruskal hay Prim để giải bài toán trong $O(n \log n)$.

Bổ đề 2

Xét một điểm P cho trước bất kì. Lấy gốc tọa độ tại P . Chia mặt phẳng tọa độ thành 8 phần bằng nhau như hình dưới. Với mỗi phần tám, nối P với một điểm bất kì trong phần tám đó có khoảng cách Manhattan gần nhất với P (nếu có). Chẳng hạn trong ví dụ ở hình dưới, ta sẽ nối P với Q .



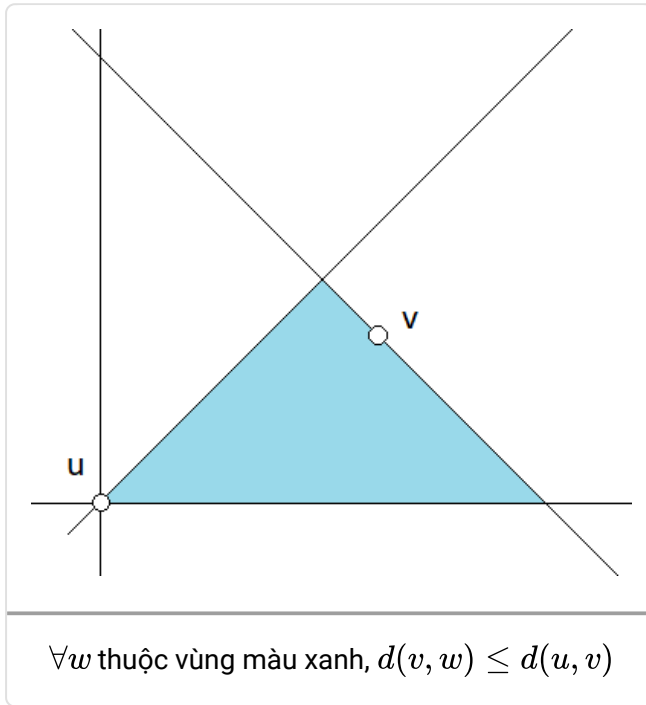
Thực hiện thao tác trên với tất cả các điểm được cho, ta thu được một đồ thị G có $O(n)$ cạnh. Ta sẽ chứng minh rằng cây khung nhỏ nhất trên đồ thị G là một đáp án cho bài toán.

Chứng minh



Ta sẽ chứng minh rằng các cạnh trong cây khung nhỏ nhất T của tập điểm ban đầu cũng thuộc đồ thị G .

Xét cạnh $(u, v) \in T$. Không mất tính tổng quát, giả sử v thuộc phần tám thứ nhất so với u . Giả sử tồn tại một điểm w trong tập điểm ban đầu sao cho $d(u, w) < d(u, v)$. Đồng thời ta biết rằng $d(v, w) < d(u, v)$ (nhìn hình minh họa bên dưới). Do đó ta sẽ có một cây khung nhỏ hơn nếu ta bỏ (u, v) và thay bằng một trong hai cạnh (u, w) hay (v, w) . Điều này trái giả thiết T là cây khung nhỏ nhất. Do đó, không tồn tại điểm w sao cho $d(u, w) < d(u, v)$ - tức v là điểm trong phần tám thứ nhất của u có khoảng cách Manhattan gần nhất. Chứng minh tương tự với các trường hợp v thuộc các phần tám còn lại của u .



Qua bổ đề 2, ta thấy bài toán đặt ra hiện tại là làm sao để dựng được đồ thị G hiệu quả. Để thấy để dựng đồ thị G , với mỗi điểm P cho trước, ta cần tìm 8 điểm có khoảng cách Manhattan gần P nhất cho mỗi phần tám. Dưới đây bài viết sẽ mô tả thuật toán tìm điểm có khoảng cách Manhattan gần nhất trong phần tám thứ 1 cho n điểm. Như vậy có thể tìm điểm khoảng cách Manhattan gần cho 7 phần tám còn lại bằng cách tương tự.

Bổ đề 3

Gọi $d(P, Q)$ là khoảng cách Manhattan giữa hai điểm P và Q . Gọi $A(x_A, y_A)$, $B(x_B, y_B)$, $C(x_C, y_C)$, $D(x_D, y_D)$ là bốn điểm trên mặt phẳng sao cho $x_A, x_B \leq x_C, x_D$ và $y_A, y_B \leq y_C, y_D$. Ta có $d(A, C) \leq d(A, D)$ tương đương với $d(B, C) \leq d(B, D)$.



Chứng minh

$$\begin{aligned}
 d(A, C) &\leq d(A, D) \\
 x_C - x_A + y_C - y_A &\leq x_D - x_A + y_D - y_A \\
 x_C + y_C &\leq x_D + y_D \\
 x_C - x_B + y_C - y_B &\leq x_D - x_B + y_D - y_B \\
 d(B, C) &\leq d(B, D)
 \end{aligned}$$

Ta sẽ sử dụng phương pháp chia để trị để giải quyết bài toán. Đầu tiên ta sẽ sắp xếp n điểm theo thứ tự tăng dần về hoành độ. Tại mỗi bước ta chia n điểm thành 2 tập con L và R . Gọi đệ quy giải bài toán với từng tập con. Nhận xét rằng lời giải cho tập R cũng chính là lời giải đúng, do đó ta chỉ cần cập nhật lời giải cho các điểm trong tập L .

Sắp xếp các điểm trong L và R theo chiều giảm dần của tung độ. Ta sẽ duy trì 3 con trỏ: con trỏ 1 duyệt các điểm trong L theo chiều giảm dần về tung độ, con trỏ 2 và 3 duyệt các điểm trong R cũng theo chiều giảm dần về tung độ.

Dùng con trỏ 1 để duyệt các điểm trong L . Gọi điểm đang được con trỏ 1 trỏ đến là $T(x_T, y_T)$. Dùng con trỏ 2 để duyệt các điểm trong R sao cho các điểm đi qua luôn có tung độ không bé hơn y_T . Con trỏ 3 trỏ đến điểm có khoảng cách Manhattan nhỏ nhất với T hiện tại (gọi điểm con trỏ 3 đang trỏ đến là U). Tại mỗi bước trong quá trình lặp, có 3 khả năng:

1. Điểm được con trỏ 2 trỏ tới có tung độ bé hơn tung độ của T : Ta so sánh khoảng cách Manhattan giữa T và U và đáp án ta có nhờ gọi đệ quy lên L , cập nhật đáp án cho T , đồng thời di con trỏ 1 đến điểm tiếp theo.
2. Điểm được con trỏ 2 trỏ tới có tung độ lớn hơn hoặc bằng tung độ của T và có khoảng cách Manhattan tới T lớn hơn khoảng cách Manhattan giữa T và U : Khi này ta không làm gì cả.
3. Điểm được con trỏ 2 trỏ tới có tung độ lớn hơn hoặc bằng tung độ của T và có khoảng cách Manhattan tới T bé hơn khoảng cách Manhattan giữa T và U : Khi này ta trỏ con trỏ 3 đến điểm đang được trỏ bởi con trỏ 2.

Nhờ có tính chất được đề cập trong bổ đề 3, ta nhận thấy vòng lặp trên sẽ cho chúng ta đáp án chính xác. Cả ba con trỏ đều "thăm" mỗi điểm trong L hoặc R đúng một lần nên độ phức tạp của mỗi "tầng" trong cây đệ quy của chúng ta sẽ là $O(n)$. Do ta sẽ có $O(\log n)$, độ phức tạp của thuật toán trên là $O(n \log n)$.

Cài đặt mẫu

Cảm ơn bạn Trần Xuân Bách (HUS High School for Gifted Students) đã đóng góp vào đoạn code này.

```
#include <bits/stdc++.h>
using namespace std;

const int INF = 1e8;

struct Point {
    int x, y, id;
    int diff_yx() const { return y - x; }
    int sum_xy() const { return x + y; }
};

int manhattan_dist(const Point& u, const Point& v) {
    return abs(u.x - v.x) + abs(u.y - v.y);
}

struct DSU {
    vector<int> par;
    DSU() {}
```



```

DSU(int n): par(n, -1) {}

int find_set(int u) {
    return par[u] < 0 ? u : par[u] = find_set(par[u]);
}

bool join(int u, int v) {
    u = find_set(u);
    v = find_set(v);
    if (u == v) return false;
    if (-par[u] < -par[v]) swap(u, v);
    par[u] += par[v];
    par[v] = u;
    return true;
}

};

struct Edge {
    int u, v, cost;
};

bool operator<(const Edge& u, const Edge& v) {
    return u.cost < v.cost;
}

Edge edge_from_point(const Point& u, const Point& v) {
    return {u.id, v.id, manhattan_dist(u, v)};
}

vector<Edge> potential_edges; // Các cạnh tối ưu cần xét
vector<Point> solve_single_recur(vector<Point> p) {
    if (p.size() <= 1) return p;
    int upper_size = (int)p.size() / 2;
    auto upper = solve_single_recur({p.begin(), p.begin() + upper_size});
    auto lower = solve_single_recur({p.begin() + upper_size, p.end()});
    vector<Point> res;

    Point min_diff_yx{0, INF, -1};

    int upper_ptr = 0;
    for (auto lo : lower) {
        while (upper_ptr < upper_size and upper[upper_ptr].sum_xy() <= lo.sum_x) {
            if (min_diff_yx.diff_yx() > upper[upper_ptr].diff_yx()) {
                min_diff_yx = upper[upper_ptr];
            }
            res.push_back(upper[upper_ptr]);
            ++upper_ptr;
        }
        if (min_diff_yx.id != -1) {
            potential_edges.push_back(edge_from_point(min_diff_yx, lo));
        }
    }
}

```

```

        res.push_back(lo);
    }

    res.insert(res.end(), upper.begin() + upper_ptr, upper.end());
    return res;
}

void solve_single(vector<Point> p) { // Giải bài toán với một góc phần tám
    sort(p.begin(), p.end(), [](const Point& u, const Point& v) {
        return u.y == v.y ? u.x < v.x : u.y > v.y;
    });
    solve_single_recur(p);
}

void rotate_90(vector<Point>& p) { // Xoay tất cả các điểm 90 độ
    for (auto& cur: p) {
        int x = cur.x, y = cur.y;
        cur.x = -y;
        cur.y = x;
    }
}

void flip(vector<Point>& p) { // Đối xứng các điểm qua trục Ox
    for (auto& cur: p) {
        cur.y = -cur.y;
    }
}

int solve(vector<Point> p) {
    for (int ft = 0; ft < 2; ++ft) {
        for (int i = 0; i < 4; ++i) {
            solve_single(p);
            rotate_90(p);
        }
        flip(p);
    }

    int ans = 0;
    DSU dsu((int)p.size());
    sort(potential_edges.begin(), potential_edges.end());

    for (const auto& e : potential_edges) {
        if (dsu.join(e.u, e.v)) {
            ans += e.cost;
        }
    }

    // Thuật toán kruskal


    return ans;
}

int main() {

```

```
122     ios_base::sync_with_stdio(false); cin.tie(NULL);
123
124     int n;
125     cin >> n;
126
127     vector<Point> p(n);
128     for (int i = 0; i < n; ++i) {
129         cin >> p[i].x >> p[i].y;
130         p[i].id = i;
131     }
132
133     cout << solve(p);
}
```

Bài tập ví dụ

[TopCoder - BoxUnion](#) 

[TopCoder - CultureGrowth](#) 

[TopCoder - PowerSupply](#) 

[TopCoder - ConvexPolygons](#) 

[SPOJ - CEPC08B](#) 

[USACO - Square Overlap](#) 






[CF - 1402B](#) 

[CSA - The Sprawl](#) 

Kết

Giống như quy hoạch động, thuật toán đường quét là một công cụ mạnh vì nó không chỉ là một thuật toán, mà còn là một dạng thuật toán mà có thể dùng để giải một phạm vi các bài toán rất rộng, bao gồm một số bài toán không được nêu ở đây (như phép tam giác hóa Delaunay) và cả các dạng mới chỉ xuất hiện lần đầu trong một cuộc thi.

Tham khảo

1. [bmerry, TopCoder: Line Sweep Algorithm](#) 
2. [Leo J. Guibas, Jorge Stolfi, On computing all north-east nearest neighbors in the L1 metric](#) 
3. Các gif được tạo bởi [manim](#) , mã nguồn tại [đây](#)  và [đây](#) 

Được cung cấp bởi [Wiki.js](#)