Last update: October 13, 2024 Translated From: e-maxx.ru

Convex Hull construction

In this article we will discuss the problem of constructing a convex hull from a set of points.

Consider N points given on a plane, and the objective is to generate a convex hull, i.e. the smallest convex polygon that contains all the given points.

We will see the **Graham's scan** algorithm published in 1972 by Graham, and also the **Monotone chain** algorithm published in 1979 by Andrew. Both are $\mathcal{O}(N \log N)$, and are asymptotically optimal (as it is proven that there is no algorithm asymptotically better), with the exception of a few problems where parallel or online processing is involved.

Graham's scan Algorithm

The algorithm first finds the bottom-most point P_0. If there are multiple points with the same Y coordinate, the one with the smaller X coordinate is considered. This step takes $\mathcal{O}(N)$ time.

Next, all the other points are sorted by polar angle in clockwise order. If the polar angle between two or more points is the same, the tie should be broken by distance from P_0, in increasing order.

Then we iterate through each point one by one, and make sure that the current point and the two before it make a clockwise turn, otherwise the previous point is discarded, since it would make a non-convex shape. Checking for clockwise or anticlockwise nature can be done by checking the orientation.

We use a stack to store the points, and once we reach the original point P_0, the algorithm is done and we return the stack containing all the points of the convex hull in clockwise order.

If you need to include the collinear points while doing a Graham scan, you need another step after sorting. You need to get the points that have the biggest polar distance from P_0 (these should be at the end of the sorted vector) and are collinear. The points in this line should be reversed so that we can output all the collinear points, otherwise the algorithm would get the nearest point in this line and bail. This step shouldn't be included in the non-collinear version of the algorithm, otherwise you wouldn't get the smallest convex hull.

Implementation

```cpp
struct pt {
    double x, y;
    bool operator == (pt const& t) const {
        return x == t.x && y == t.y;
    }
};

int orientation(pt a, pt b, pt c) {
    double v = a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y);
    if (v < 0) return -1; // clockwise
    if (v > 0) return +1; // counter-clockwise
    return 0;
}

bool cw(pt a, pt b, pt c, bool include_collinear) {
    int o = orientation(a, b, c);
    return o < 0 || (include_collinear && o == 0);
}
bool collinear(pt a, pt b, pt c) { return orientation(a, b, c) == 0; }

void convex_hull(vector<pt>& a, bool include_collinear = false) {
    pt p0 = *min_element(a.begin(), a.end(), [](pt a, pt b) {
        return make_pair(a.y, a.x) < make_pair(b.y, b.x);
    });
    sort(a.begin(), a.end(), [&p0](const pt& a, const pt& b) {
        int o = orientation(p0, a, b);
        if (o == 0)
            return (p0.x-a.x)*(p0.x-a.x) + (p0.y-a.y)*(p0.y-a.y)
                < (p0.x-b.x)*(p0.x-b.x) + (p0.y-b.y)*(p0.y-b.y);
        return o < 0;
    });
    if (include_collinear) {
        int i = (int)a.size()-1;
        while (i >= 0 && collinear(p0, a[i], a.back())) i--;
        reverse(a.begin()+i+1, a.end());
    }

    vector<pt> st;
    for (int i = 0; i < (int)a.size(); i++) {
        while (st.size() > 1 && !cw(st[st.size()-2], st.back(), a[i],
include_collinear))
            st.pop_back();
        st.push_back(a[i]);
    }

    if (include_collinear == false && st.size() == 2 && st[0] == st[1])
        st.pop_back();

    a = st;
}
```

Monotone chain Algorithm

The algorithm first finds the leftmost and rightmost points A and B. In the event multiple such points exist, the lowest among the left (lowest Y-coordinate) is taken as A, and the highest among the right (highest Y-coordinate) is taken as B. Clearly, A and B must both belong to the convex hull as they are the farthest away and they cannot be contained by any line formed by a pair among the given points.

Now, draw a line through AB. This divides all the other points into two sets, S1 and S2, where S1 contains all the points above the line connecting A and B, and S2 contains all the points below the line joining A and B. The points that lie on the line joining A and B may belong to either set. The points A and B belong to both sets. Now the algorithm constructs the upper set S1 and the lower set S2 and then combines them to obtain the answer.

To get the upper set, we sort all points by the x-coordinate. For each point we check if either - the current point is the last point, (which we defined as B), or if the orientation between the line between A and the current point and the line between the current point and B is clockwise. In those cases the current point belongs to the upper set S1. Checking for clockwise or anticlockwise nature can be done by checking the orientation.

If the given point belongs to the upper set, we check the angle made by the line connecting the second last point and the last point in the upper convex hull, with the line connecting the last point in the upper convex hull and the current point. If the angle is not clockwise, we remove the most recent point added to the upper convex hull as the current point will be able to contain the previous point once it is added to the convex hull.

The same logic applies for the lower set S2. If either - the current point is B, or the orientation of the lines, formed by A and the current point and the current point and B, is counterclockwise - then it belongs to S2.

If the given point belongs to the lower set, we act similarly as for a point on the upper set except we check for a counterclockwise orientation instead of a clockwise orientation. Thus, if the angle made by the line connecting the second last point and the last point in the lower convex hull, with the line connecting the last point in the lower convex hull and the current point is not counterclockwise, we remove the most recent point added to the lower convex hull as the current point will be able to contain the previous point once added to the hull.

The final convex hull is obtained from the union of the upper and lower convex hull, forming a clockwise hull, and the implementation is as follows.

If you need collinear points, you just need to check for them in the clockwise/counterclockwise routines. However, this allows for a degenerate case where all the input points are collinear in a single line, and the algorithm would output repeated points. To solve this, we check whether the upper hull contains all the points, and if it does, we just return the points in reverse, as that is what Graham's implementation would return in this case.

Implementation

```
struct pt {
    double x, y;
};

int orientation(pt a, pt b, pt c) {
    double v = a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y);
    if (v < 0) return -1; // clockwise
    if (v > 0) return +1; // counter-clockwise
    return 0;
}

bool cw(pt a, pt b, pt c, bool include_collinear) {
    int o = orientation(a, b, c);
    return o < 0 || (include_collinear && o == 0);
}
bool ccw(pt a, pt b, pt c, bool include_collinear) {
    int o = orientation(a, b, c);
    return o > 0 || (include_collinear && o == 0);
}

void convex_hull(vector<pt>& a, bool include_collinear = false) {
    if (a.size() == 1)
        return;

    sort(a.begin(), a.end(), [](pt a, pt b) {
        return make_pair(a.x, a.y) < make_pair(b.x, b.y);
    });
    pt p1 = a[0], p2 = a.back();
    vector<pt> up, down;
    up.push_back(p1);
    down.push_back(p1);
    for (int i = 1; i < (int)a.size(); i++) {
        if (i == a.size() - 1 || cw(p1, a[i], p2, include_collinear)) {
            while (up.size() >= 2 && !cw(up[up.size()-2], up[up.size()-1],
a[i], include_collinear))
                up.pop_back();
            up.push_back(a[i]);
        }
        if (i == a.size() - 1 || ccw(p1, a[i], p2, include_collinear)) {
            while (down.size() >= 2 && !ccw(down[down.size()-2],
down[down.size()-1], a[i], include_collinear))
                down.pop_back();
            down.push_back(a[i]);
        }
    }

    if (include_collinear && up.size() == a.size()) {
        reverse(a.begin(), a.end());
        return;
    }
    a.clear();
    for (int i = 0; i < (int)up.size(); i++)
        a.push_back(up[i]);
    for (int i = down.size() - 2; i > 0; i--)
        a.push_back(down[i]);
}
```

Practice Problems

- Kattis - Convex Hull
- Kattis - Keep the Parade Safe
- Latin American Regionals 2006 - Onion Layers
- Timus 1185: Wall
- Usaco 2014 January Contest, Gold - Cow Curling