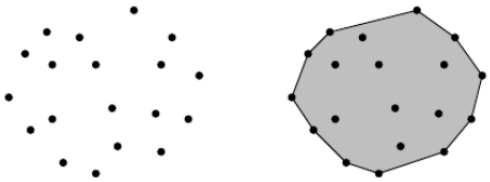
GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: CONSTRUCCIÓN DE POLÍGONO CONVEXO (CONVEX HULL)

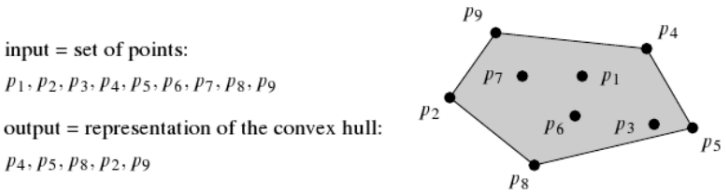
---

1. Introducción

Considerar N puntos dados en un plano, y el objetivo es generar una envolvente convexa, es decir, el polígono convexo más pequeño que contiene todos los puntos dados. En la siguientes figuras se observa el caso que queremos presentar. La izquierda se presenta el grupo de puntos mientras en la derecha se presenta el mismo grupo de puntos pero con aquellos que fueron seleccionados que formarían parte del polígono convexo o envolvente convexa.



El problema (planar) de la cubierta convexa se define de la siguiente forma: Dado un conjunto P de n puntos en el plano, calcular la representación del polígono convexo cerrado que representa la cubierta convexa de P. La representación más simple de una cubierta convexa es la enumeración en el sentido inverso a las manecillas del reloj de sus vértices. Idealmente la cubierta convexa debe consistir sólo de los puntos extremos, en el sentido que si tres puntos caen en un vértice de la frontera de la cubierta convexa, entonces el punto medio no debe ser tomado en cuenta como parte de la cubierta.



2. Conocimientos previos

2.1. Polígono convexo

Un polígono convexo es un polígono en el que cada uno de los ángulos interiores miden la suma de 180 grados o π radianes. Un polígono es estrictamente convexo si todos sus ángulos internos son estrictamente menores de 180 grados y todas sus diagonales son interiores. Todo polígono que no es convexo se denomina polígono cóncavo.

3. Desarrollo

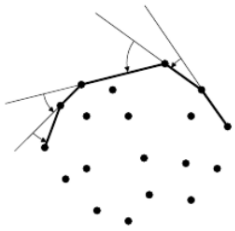
Veremos el algoritmo de Escaneo de Graham (*Graham's scan*) publicado en 1972 por Graham, y también el algoritmo de Cadena monótona (*Monotone chain*) publicado en 1979 por Andrew.

3.1. Escaneo de Graham (*Graham's scan*)

El algoritmo primero encuentra el punto más bajo P_0. Si hay varios puntos con la misma coordenada Y, se considera el que tiene la coordenada X más pequeña. Este paso toma $O(N)$ tiempo.

A continuación, todos los demás puntos se ordenan por ángulo polar en el sentido de las agujas del reloj. Si el ángulo polar entre dos puntos es el mismo, se elige el punto más cercano.

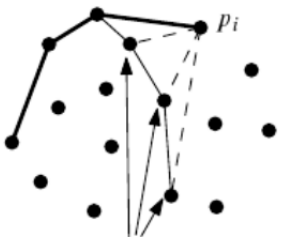
Luego iteramos a través de cada punto uno por uno, y nos aseguramos de que el punto actual y los dos anteriores hagan un giro en el sentido de las agujas del reloj, de lo contrario, el punto anterior se descarta, ya que tendría una forma no convexa. La verificación de la naturaleza en el sentido de las agujas del reloj o en el sentido contrario a las agujas del reloj se puede realizar comprobando la orientación.



Usamos una pila para almacenar los puntos, y una vez que llegamos al punto original P_0, el algoritmo está hecho y devolvemos la pila que contiene todos los puntos del casco convexo en el sentido de las agujas del reloj.

El algoritmo de Graham trabaja de la siguiente forma:

- Sea p_i el próximo punto que se agregará al ordenamiento de izquierda a derecha de los puntos.
- Si la tripleta p_i , H.first, H.second tiene orientación positiva, entonces podemos simplemente agregar p_i a la pila
- Sino, se puede inferir que le punto medio de la tripleta *H.first* no puede estar en la cubierta convexa
- Por lo tanto lo borramos de la pila.



- Ésto es repetido hasta alcanzar una tripleta con orientación positiva, o haya menos de dos elementos en la pila

Si necesita incluir los puntos colineales mientras realiza un escaneo de Graham, necesita otro paso después de ordenar. Necesita obtener los puntos que tienen la mayor distancia polar de P_0 (estos deben estar al final del vector ordenado) y son colineales. Los puntos en esta línea deben invertirse para que podamos generar todos los puntos colineales; de lo contrario, el algoritmo obtendría el punto más cercano en esta línea y saldría. Este paso no debe incluirse en la versión no colineal del algoritmo, de lo contrario no obtendría el casco convexo más pequeño.

3.2. Cadena monótona (*Monotone chain*)

El algoritmo es una optimización del algoritmo de Escaneo de Graham (*Graham's scan*) para determinar la cubierta convexa. Este caso se halla de manera separada las cubiertas superior e inferior de la cubierta a continuación una implementación de la misma.

El algoritmo primero encuentra los puntos A y B más a la izquierda y más a la derecha. En el caso de que existan varios puntos de este tipo, el más bajo entre la izquierda (coordenada Y más baja) se toma como A, y el más alto entre la derecha (coordenada Y más alta) es tomado como B. Claramente, A y B deben pertenecer ambos al casco convexo ya que son los más alejados y no pueden ser contenidos por ninguna línea formada por un par entre los puntos dados.

Ahora, dibuja una línea a través de AB. Esto divide todos los demás puntos en dos conjuntos, S1 y S2, donde S1 contiene todos los puntos por encima de la línea que une A y B, y S2 contiene todos los puntos por debajo de la línea que une A y B. Los puntos que se encuentran en la línea que une A y B pueden pertenecer a cualquiera de los conjuntos. Los puntos A y B pertenecen a ambos conjuntos. Ahora el algoritmo construye el conjunto superior S1 y el conjunto inferior S2 y luego los combina para obtener la respuesta.

Para obtener el conjunto superior, ordenamos todos los puntos por la coordenada x. Para cada punto, verificamos si el punto actual es el último punto (que definimos como B), o si la orientación entre la línea entre A y el punto actual y la línea entre el punto actual y B es en el sentido de las agujas del reloj. En esos casos el punto actual pertenece al conjunto superior S1. La verificación de la naturaleza en el sentido de las agujas del reloj o en el sentido contrario a las agujas del reloj se puede realizar comprobando la orientación.

Si el punto dado pertenece al conjunto superior, comprobamos el ángulo formado por la línea que une el penúltimo punto y el último punto del casco convexo superior, con la línea que une el último punto del casco convexo superior y el punto actual. Si el ángulo no es en el sentido de las agujas del reloj, eliminamos el punto más reciente agregado al casco convexo superior ya que el punto actual podrá contener el punto anterior una vez que se agregue al casco convexo.

La misma lógica se aplica para el conjunto inferior S2. Si el punto actual es B, o la orientación de las líneas, formadas por A y el punto actual y el punto actual y B, es en sentido contrario a las agujas del reloj, entonces pertenece a S2.

Si el punto dado pertenece al conjunto inferior, actuamos de manera similar a un punto en el conjunto superior excepto que verificamos una orientación en sentido contrario a las manecillas del reloj en lugar de una orientación en el sentido de las manecillas del reloj. Por lo tanto, si el ángulo formado por la línea que conecta el penúltimo punto y el último punto en el casco convexo

inferior, con la línea que conecta el último punto en el casco convexo inferior y el punto actual no es en sentido antihorario, eliminamos el punto más reciente agregado al casco convexo inferior como el punto actual podrá contener el punto anterior una vez agregado al casco.

El casco convexo final se obtiene de la unión del casco convexo superior e inferior, formando un casco en el sentido de las agujas del reloj, y la realización es la siguiente.

Si necesita puntos colineales, solo necesita buscarlos en las rutinas en sentido horario/antihorario. Sin embargo, esto permite un caso degenerado en el que todos los puntos de entrada son colineales en una sola línea y el algoritmo generaría puntos repetidos. Para resolver esto, verificamos si el casco superior contiene todos los puntos, y si los contiene, simplemente devolvemos los puntos al revés, ya que eso es lo que devolvería la implementación de Graham en este caso.

4. Implementación

4.1. C++

4.2. Escaneo de Graham (*Graham's scan*)

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

int orientation(Point a, Point b, Point 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; //en sentido de las agujas del reloj
   if(v>0) return +1; //en sentido contrario a las agujas del reloj
   return 0;
}

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

bool collinear(Point a, Point b, Point c){ return orientation(a,b,c)==0;}

vector<Point> grahamScan(vector<Point>& a, bool includeCollinear = false){
   Point p0 = *min_element(a.begin(), a.end(), [](Point a, Point b) {
      return make_pair(a.y, a.x) < make_pair(b.y, b.x);
   });
   sort(a.begin(), a.end(), [&p0](const Point& a, const Point& 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(includeCollinear){
        int i=(int)a.size()-1;
        while( i>=0 && collinear(p0,a[i],a.back()))i--;
        reverse(a.begin()+i+1,a.end());
    }
    vector<Point> st;
    for(int i=0; i<(int)a.size();i++) {
        while(st.size()>1 && !cw(st[st.size()-2], st.back(), a[i],
            includeCollinear))
            st.pop_back();
        st.push_back(a[i]);
    }
    return st;
}
```

4.3. Cadena monótona (*Monotone chain*)

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

int orientation(Point a, Point b, Point 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; //en sentido de las agujas del reloj
    if (v > 0) return +1; //en sentido contrario a las agujas del reloj
    return 0;
}

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

vector<Point> monotoneChain(vector<Point>& a, bool includeCollinear = false) {
    vector<Point> st;
    if(a.size() == 1) return st;

    sort(a.begin(),a.end(),[](Point a, Point b){
        return make_pair(a.x, a.y) < make_pair(b.x, b.y);
    });
    Point p1=a[0], p2=a.back();
    vector<Point> 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,includeCollinear)){
         while(up.size()>=2 && !cw(up[up.size()-2],up[up.size()-1],a[i],
            includeCollinear))
            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],includeCollinear))
            down.pop_back();
         down.push_back(a[i]);
      }
   }
   if(includeCollinear && up.size()==a.size()){
      reverse(a.begin(),a.end()); st=a; return st;
   }

   for (int i = 0; i < (int)up.size(); i++) st.push_back(up[i]);
   for (int i = down.size() - 2; i > 0; i--) st.push_back(down[i]);
   return st;
}
```

4.4. Java

4.5. Escaneo de Graham (*Graham's scan*)

```
public Point[] grahamScan(Point[] points) {
   Arrays.sort(points, Comparator.<Point>comparingInt(p -> p.x).
      thenComparingInt(p -> p.y));
   int n = points.length;
   Point[] hull = new Point[n + 1];
   int cnt = 0;
   for(int i=0; i<2*n-1;i++) {
      int j=i<n ? i : 2*n-2-i;
      while (cnt>=2 && isNotRightTurn(hull[cnt-2],hull[cnt-1],points[j])) --
         cnt;
      hull[cnt++] = points[j];
        }
        return Arrays.copyOf(hull, cnt - 1);
}

public boolean isNotRightTurn(Point a, Point b, Point c) {
   long cross = (long)(a.x-b.x)*(c.y-b.y)-(long)(a.y-b.y)*(c.x-b.x);
   long dot = (long)(a.x-b.x)*(c.x-b.x)+(long)(a.y-b.y)*(c.y-b.y);
   return cross < 0 || cross == 0 && dot <= 0;
}

public class Point {
```

```
    public int x, y;

    public Point(int x, int y) {
        this.x = x; this.y = y;
    }
}
```

5. Complejidad

Ambos algoritmos son $O(N \log N)$, y son asintóticamente óptimos (ya que está demostrado que no existe un algoritmo asintóticamente mejor), con la excepción de algunos problemas donde se involucra el procesamiento paralelo o en línea.

6. Aplicaciones

Existen muchas razones por las cuales un cubierta convexa de un conjunto de puntos es una estructura geométrica importante:

- Es una de las aproximaciones de forma de un conjunto de puntos más simples (otras incluyen rectángulos, círculos, etc.)
- Puede ser usada para aproximar formas más complejas (cubiertas convexas de polígonos o poliedros)
- Algunos algoritmos calculan la cubierta convexa como una etapa inicial (preprocesamiento) de su ejecución (filtrar puntos irrelevantes)

Por ejemplo, el diámetro de un conjunto de puntos es la máxima distancia entre cualesquiera dos puntos del conjunto. Puede demostrarse que el par de puntos que determina el diámetro son ambos vértices de la cubierta convexa. También se puede observar que las mínimas formas convexas envolventes (rectángulo, círculo, etc.) depende sólo de los puntos de la cubierta convexa.

7. Ejercicios propuestos

A continuación una lista de ejercicios que se pueden resolver aplicando estos algoritmos:

- Kattis - Convex Hull
- UVA - 681 - Convex Hull Finding
- UVA - 11626 - Convex Hull