Convex Hull

Carlos Reátegui

UTEC
Email: carlos.reategui@utec.edu.pe

I. Introducción

Dado un conjunto P de n puntos, se denomina *convex hull* al conjunto convexo más pequeño que contiene a P [1].

En la figura 1 se puede observar un Convex Hull sobre la figura de un monje.

Figura 1. Ejemplo de un Convex Hull sobre una imagen [2].

Así como se puede aplicar un Convex Hull a figuras, también en se puede aplicar sobre puntos en n-dimensiones.

En la figura 2 se puede observar un Convex Hull sobre puntos en 2 dimensiones, y en la figura 3 se puede observar un Convex Hull sobre puntos en 3 dimensiones.

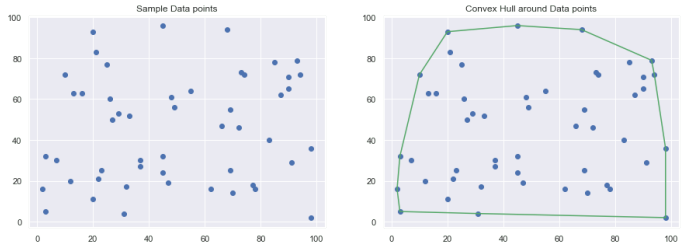

Figura 2. Convex Hull en dos dimensiones. Recuperado de: https://bit.ly/3p2Pw3D

II. Aplicaciones

Las aplicaciones del Convex Hull son muy variadas. Entre ellas están las siguientes [3]:

- Ray tracing: Representación de imágenes con rayos de luz.
- "Path finding"
- Sistemas Geográficos de Información: El Convex Hull puede ser utilizado para generar mapas.

Figura 3. Convex Hull en tres dimensiones. Recuperado de: https://bit.ly/3p70V2k

- Coincidencia visual de patrones: Identificación de objetos a partir de imágenes
- Geometría: Hallar diámetros de objetos.

III. Algoritmos

Existen tres algoritmos muy populares para generar un Convex Hull dado un conjunto de puntos en el espacio. Estos son el algoritmo de Graham, el algoritmo de Jarvis y el algoritmo de Chan, que usa los dos primeros algoritmos mencionados.

III-A. Algoritmo de Graham

Dado un conjunto $S = \{s_1, ..., s_n\}$ de puntos en 2 dimensiones, este algoritmo sigue los siguientes pasos [5]:

1. Encontrar un punto de origen $P \in S$. P debe ser el punto con la menor coordenada en Y y la menor coordenada en X.
2. Generar un conjunto $|A|$ que contenga a los ángulos entre P y el resto de puntos $s_i \in S$.
3. Ordenar los ángulos de $|A|$ en orden ascendente. Esta operación toma $O(n \lg n)$, donde n es la cantidad de puntos de S.
4. Crear una estrutura LIFO L, e insertar el punto P y el primer punto de A (recordar que A está ordenado en orden ascendente).
5. Iterar sobre cada punto Q de A en el orden en el que se encuentran, y realizar las siguientes operaciones:
   a) Extraer un punto R de L (recordar que es una estructura LIFO).
   b) Si el punto Q realiza un giro antihorario respecto a los 2 puntos previos de L, se agrega este punto a L. En caso contrario, extraer otro punto de L y asignarlo a R mientras que no se cumpla la condición de giro antihorario.
6. Retornar los puntos de L.

III-A1. Implementación: Para encontrar el punto de origen, podemos iterar a través de cada punto y encontrar el que cumpla las características descritas anteriormente (ver listing 1).

```
findOriginPoint(points):
  minPoint = new Point(inf, inf)
  for point in points:
    if point.y <= minPoint.y:
      if point.y < minPoint.y:
        minPoint = point
      else if point.x < minPoint.x:
        minPoint = point
  return minPoint
```

Listing 1. Hallar punto de origen

Para calcular si se realiza un giro antihorario, podemos hallar el producto cruz entre los vectores generados por los puntos en cuestión. Respecto a los puntos p_0, p_1 y p_2: si el resultado es positivo, significa que se da un giro antihorario; si el resultado es negativo, significa que se da un giro horario, y si el resultado es 0, significa que los puntos son colineales. En el listing 2 se puede observar la implementación, y en la figura 4 se pueden ver los inputs del listing 2.

```
calculateCrossProduct(p0, p1, p2):
    crossProduct = (p1.x - p0.x) * (p2.y - p0.y) - (
    p1.y - p0.y) * (p2.x - p0.x)

    if crossProduct > 0:
        is counterclockwise turn
    else if crossProduct = 0:
        is collinear turn
    else:
        is a clockwise turn
```

Listing 2. Producto cruz

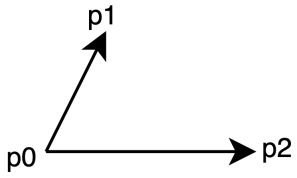

Figura 4. Inputs del listing 2

El ”scan” para generar el Convex Hull, puede observarse en el listing 3. Empieza obteniendo el punto de origen (ver listing 1) y ordenando el resto de puntos respecto al ángulo formado con el punto de origen. Luego, en el `for` halla el Convex Hull bajo el criterio descrito anteriormente.

```
grahamScan(points):
  originPoint = findOriginPoint(points)
  sortByAngle(originPoint, points)
  stack.push(points[0]).push(points[1])
  n = |points|
  for i = 2 to n - 1:
    nextPoint = points[i]
    currentPoint = stack.pop()
    while stack is not empty and crossProduct(stack.
    top, currentPoint, nextPoint) <= 0:
      currentPoint = stack.pop()
    stack.push(currentPoint).push(nextPoint)
  return elements of stack
```

Listing 3. Generar Convex Hull con Graham Scan

III-A2. Análisis asintótico: El listing 2 claramente toma tiempo constante.

Por otro lado, el listing 3 realiza diferentes operaciones, dignas de analizar por partes. En primer lugar, se halla el punto de origen, lo cual toma tiempo lineal respecto a la cantidad de puntos. En segundo lugar, se ordenan los puntos respecto al ángulo que forman. Esta operación toma $O(n \lg n)$, donde n es la cantidad de puntos, en case se realice un algoritmo como el Mergesort o el Quicksort. Finalmente, se va a iterar sobre cada punto para hallar el Convex Hull. En cada iteración se realizan operaciones que toman tiempo constante. Por tanto, todo el `for` loop toma tiempo lineal.

Por tanto, el tiempo de ejecución de todo el algoritmo toma $O(n \lg n)$.

III-B. Algoritmo de Jarvis

Este algoritmo fue propuesto en 2 dimensiones. Dado un conjunto $S = \{s_1, ..., s_n\}$ de puntos en 2 dimensiones, este algoritmo sigue los siguientes pasos [4]:

1. Encontrar un punto de origen $P \in S$. P debe ser el punto con la menor coordenada en Y y la menor coordenada en X. Agregar el punto al Convex Hull.
2. Definir un nuevo punto Q sin coordenadas especificadas. Mientras que Q sea diferente de P, realizar las siguientes operaciones:
   a) Iterar a través de cada punto s_i de S, y realizar las siguientes operaciones:
      1) Hallar el menor ángulo (en sentido antihorario) entre el último punto agregado al Convex Hull y s_i.
   b) Asignar a Q el punto s_i que forma el menor ángulo hallado en el paso anterior.
   c) Agregar a Q al Convex Hull.
3. Retornar el Convex Hull.

III-B1. Implementación: El método para hallar el Convex Hull propuesta por Jarvis, se puede implementar como se muestra en el listing 4.

Primero se halla el punto de origen usando el listing 1, descrito anteriormente. Se usa nuevamente el producto cruz entre los puntos para determinar si un punto pertenece o no al Convex Hull. En la línea 13 se verifica si hay dos puntos colineales con el fin de elegir el que esté más lejano. En la línea 15 se verifica si se ha realizado un giro horario, ya que eso quiere decir que se ha encontrado un punto que genera un ángulo más pequeño en sentido antihorario, en referencia al último punto agregado al Convex Hull.

```
jarvis(points):
  convexHull = {}
  originPoint = findOrigin(points)
  add originPoint to convexHull
  currentPoint = originPoint
  while true:
    minAngle = inf
```

```
    maxDistance = 0
    nextPoint = points[0]
    for point in points:
      if point = currentPoint: continue
      crossProduct = calculateCrossProduct(
    currentPoint, nextPoint, point)
      if crossProduct = 0 and eucDist(currentPoint,
    nextPoint) < eucDist(currentPoint, point):
        nextPoint = point
      else if crossProduct < 0:
        nextPoint = point
    break while loop if originPoint equals nextPoint
    add nextPoint to convexHull
    currentPoint = nextPoint
  return convexHull
```

Listing 4. Generar Convex Hull con el algoritmo de Jarvis

III-B2. Análisis asintótico: El listing 1 claramente toma tiempo lineal respecto a la cantidad de puntos. Por otro lado, el listing 4 va a iterar sobre todos los puntos hasta que el punto de origen se reencuentre consigo mismo. Por tanto, se puede decir que el listing 4 toma $O(nh)$, donde n es la cantidad de puntos totales y h es la cantidad de puntos que pertenecen al Convex Hull [4].

Por tanto, el tiempo de ejecución de todo el algoritmo toma $O(nh)$. Este tipo de algoritmos en los que la comlejiddad está determinada como una función del input y del tamaño del output, son llamados *output sensitive* [1]. En el peor de los casos, todos los puntos forman parte del Convex Hull, lo que llevaría a que el algoritmo tome $O(n^2)$.

III-C. Algoritmo de Chan

La propuesta de Chan consiste en dividir los puntos en grupos, aplicar Graham scan sobre estos puntos, y luego aplicar el algoritmo de Jarvis sobre el resultado del scaneo anterior.

Chan describe el algoritmo de la siguiente manera [1]:

1. Elegir un parámetro m entre 1 y n (donde n es la cantidad total de puntos) y partir el conjunto P de puntos en $\lceil n/m \rceil$ puntos de tamaño máximo m.
2. Luego, hallar el Convex Hull de cada uno de los grupos en $O(m \lg m)$, usando Graham scan, por ejemplo. Dado que en total hay $\lceil n/m \rceil$ grupos, y esta operación toma $O(m\ lgm)$ sobre cada grupo, en total esta operación toma $O(n \lg m)$.
3. Se realiza un "wrapping step" scaneando todos los polígonos y calculando las tangentes de los polígonos a través del vértice actual p_k (ver figura 5). Según Chan, realizar esto en cada polígono toma $O(\lg m)$ usando búsqueda binaria, y como hay $\lceil n/m \rceil$ polígonos en total, este paso toma $O((n/m) \lg m))$.
4. Se necesitan h "wrapping steps" (algoritmo de Jarvis) para generar el Convex Hull. Por tanto, el tiempo total del algoritmo es $O(n \lg m + h((n/m) \log m)) = O(n(1 + h/m) \lg m)$.

El tiempo total del algoritmo es $O(n \lg h)$ cuando $m = h$. Sin embargo, el problema que presenta este algoritmo es que no se conoce a priori el valor de h.

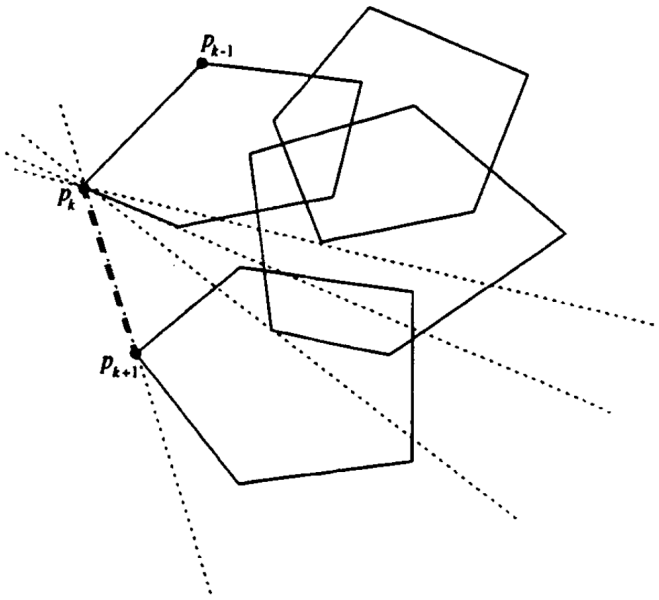

Figura 5. Paso de "wrapping" [1]

Lo que propone Chan para afrontar este problema, es lo que se observa en el listing 5 [6].

```
Hull2D(points):
for t = 1 to infinity:
  m = min(2^(2^t), n)
  Run Chan with m, output to L
  if L != incomplete, return L
```

Listing 5. Convex Hull Chan

Claramente, cada iteración toma $O(n \lg 2^{2^t}) = O(n2^t)$. Por otro lado, el maximo valor de t es $\lg\lg h$, debido a que se calculó satisfactoriamente el Convex Hull tan pronto como $2^{2^t} > h$.

Por ello:

$$\sum_{t=1}^{\lg\lg h} n2^t = n \sum_{t=1}^{\lg\lg h} 2^t \leq n2^{1+\lg\lg h} = 2n \lg h$$

Por tanto, con el método de Chan se puede hallar el Convex Hull de un conjunto de puntos en dos dimensiones en tiempo $O(n \lg h)$ [1] [6].

IV. Observaciones adicionales

1. Jarvis sugiere que su algoritmo se presta para procesamiento paralelo, dado que se pueden hallar los "wraps" de un conjunto de puntos, y que apartir de estos detectar overlapping entre ellos para generar el Convex Hull.
2. Chan propone un algoritmo similar al presentado en este trabajo para 3 dimensiones, que también toma $O(n \lg h)$ y espacio lineal [1].

REFERENCIAS

[1] Chan, T. M. (1996). *Discrete Comput Geom. Optimal Output-Sensitive Convex Hull Algorithms in Two and Three Dimensions*, 361-368.

[2] Cantoni, V., Levialdi, S., & Zavidovique, B. (2011). *Visual Cues. 3C Vision*, 19–113. doi:10.1016/b978-0-12-385220-5.00002-4

[3] Briquet, C. (2007). *Introduction to Convex Hull Applications* [PowerPoint presentation]. Belgium. Recuperado de: https://bit.ly/2GBTMpn.

[4] Jarvis, R. A. (1973). *On the identification of the convex hull of a finite set of points in the plane*. Canberra.

[5] Graham, R. L. (1972). *An efficient algorithm for determining the convex hull of a finite planar set. Information Processing Letters*, 132–133.

[6] Suri, S. (2002). *Computational Geometry - Convex Hulls* [PowerPoint presentation]. Recuperado de: https://bit.ly/3l8J9t3.