de Faria Campos, Edison. *Un breve* tour *por la geometría computacional.* **Tecnología en Marcha**. Vol. 14; Nº 3.

Un breve *tour* por la geometría computacional

*Edison de Faria Campos**

Resumen

Los algoritmos geométricos son importantes en el diseño y análisis de sistemas que modelan objetos físicos. El campo de la geometría computacional es importante de estudiar debido a su fuerte contexto histórico, a las aplicaciones que requieren de algoritmos geométricos, y por ser un campo de mucha investigación en la actualidad. En este artículo de divulgación, se hace un breve tour *por la geometría computacional, y se analizan algunos algoritmos importantes en esta área.*

1. Introducción

Geometría computacional es una rama de la ciencia de la computación que estudia algoritmos para resolver problemas geométricos. Se trata de encontrar algoritmos eficientes o, bien, procedimientos computacionales para resolver un amplio espectro de problemas geométricos.

Objetos geométricos como puntos, rectas y polígonos son la base de una amplia variedad de importantes aplicaciones que dan origen a problemas muy interesantes. Problemas tan triviales de visualizar o de resolver como el de determinar si un punto dado se encuentra dentro de un polígono o si dos segmentos se intersecan, requieren programas no triviales.

Partes del ámbito de aplicación de la geometría computacional son:

- Gráficas por computadora,
- Robótica,
- Diseño VLSI (Very-Large-Scale-Integration) de circuitos digitales,
- Diseño asistido por computadora,
- Reconocimiento de patrones,
- Investigación de operaciones,
- Inteligencia Artificial,
- Conformación molecular y
- Estadística.

La entrada a un problema geométrico-computacional es la descripción de un

* Escuela de Matemática, Universidad de Costa Rica.

conjunto de objetos geométricos, tales como conjuntos de puntos, conjuntos de segmentos de rectas, vértices de un polígono, etc., y la salida es una respuesta a una pregunta respecto de los objetos, como, por ejemplo, si dos rectas se cortan, o determinar la cáscara convexa de un conjuntos de puntos.

Los problemas geométricos pueden ocurrir en espacios euclideanos o, bien, en espacios no euclideanos (como ocurre con los *fractales* por ejemplo). Algunos enfoques un poco diferentes, y que algunos autores consideran ser parte de la geometría computacional por ejemplo son: dibujar una curva de mejor ajuste para un conjunto de puntos en el plano.

[16] considera que el término *geometría computacional* es debido a Minsky y Papert (1969) como un sustituto al modelo de reconocimiento de patrones.

Según [13] los algoritmos de naturaleza geométrica han sido desarrollados en distintos contextos, y la frase "geometría computacional" ha sido utilizada con distintas connotaciones.

1. Modelación geométrica por medio de curvas *splines* y superficies, un "tópico" que es más cercano al espíritu de análisis numérico que a la geometría. Bézier (1972), Forrest (1971), Riesenfeld (1973). Forrest refiere a su disciplina como Geometría computacional.

2. Minsky y Papert en su libro *Perceptions*, con subtítulo "Geometría computacional", trata con la complejidad de predicados que reconocen ciertas propiedades geométricas, tales como convexidad. El intento de su trabajo fue el de hacer estatutos relacionados con la posibilidad de utilizar grandes retinas compuestas de circuitos simples para realizar tareas de reconocimiento de patrones.

Software gráficos y editores geométricos, meta de muchos de los algoritmos utilizados en [13] se orientan más hacia los detalles de implementación e interfaces de usuario que al análisis de algoritmos.

Geometría computacional no se relaciona con demostración de teoremas en geometría utilizando computadoras.

Según [13] el nombre Geometría computacional para esta nueva disciplina de la ciencia de la computación aparece por primera vez en un artículo de M. I. Shamos (1975) [15].

Podríamos sospechar que los algoritmos geométricos han tenido una trayectoria muy larga, debido a la antigüedad y naturaleza constructiva de la geometría, pero mucho del trabajo en este campo es reciente.

Euclides tuvo una popularidad muy grande con su método axiomático de demostraciones. Los postulados geométricos y el método de reducción al absurdo se utilizaron sin variaciones sustanciales durante cerca de 2000 años. La técnica de reducción al absurdo simplifica la demostración de existencia de un objeto por medio de contradicción, en lugar de dar una construcción explícita de él (algoritmo).

Pero lo más relevante de Euclides fue la invención de la construcción euclideana, un esquema que consiste de un algoritmo y su prueba. La construcción euclideana satisface todos los requerimientos de un algoritmo: Es no ambiguo, correcto y termina. Un ejemplo clásico es un algoritmo de Euclides encontrado en los *Elementos*, que determina el máximo común divisor entre dos números.

Pero mientras crecía la popularidad del método axiomático y progresaba la geometría, el análisis de algoritmos se mantuvo estancado durante cerca de 2000 años.

La influencia de Euclides fue tan profunda que, muchos siglos después, Descartes formuló otro modelo para la geometría, introduciendo sistemas de coordenadas que permitían resolver problemas geométricos en términos algebraicos.

Debido a esto, mucho del trabajo en el campo de la geometría computacional es reciente.

2. Algunos ejemplos de problemas geométricos

Algunos ejemplos de problemas que la geometría computacional trata son:

- Dados dos segmentos rectilíneos, ¿es uno de ellos orientado en la dirección positiva respecto al otro?
- ¿Dos segmentos rectilíneos se cortan?
- ¿Cómo determinar la cáscara convexa (*convex hull*) de un conjunto de puntos?
- Dado un conjunto de puntos, determinar los dos puntos del conjunto que se encuentran más cercanos.
- Dados N puntos en un plano, ¿cuántos se encuentran en las aristas de un polígono dado?
- Dado un polígono simple P y un punto z, determinar si z es un punto interior o exterior a P.
- Dados N puntos en el plano, ¿son ellos vértices de su cáscara convexa?
- Dados N puntos en el plano, particionarlo en K "clusters" $C_1,\ldots,C_k$ tales que el diámetro del *cluster* máximo es tan pequeño como sea posible.
- Dados N puntos en el plano, determinar el vecindario más cercano de cada uno de ellos.
- Dados N puntos en el plano, construir un árbol de longitud total mínima cuyos vértices son los puntos dados.
- Dados N puntos en el plano, unirlos por medio de segmentos rectilíneos que no se cortan, tal que toda región interna a la cáscara convexa es un triángulo.
- Dados N puntos en el plano, determinar el menor círculo que los encierra.
- Dados N puntos en el plano, determinar el mayor círculo que no contiene puntos del conjunto, tal que su centro es un punto interior a la cáscara convexa de los puntos dados.
- Dado un conjunto S de N números reales, $x_1,\ldots,x_N$, y un número real $\varepsilon > 0$, determinar si las diferencias entre miembros consecutivos de S son uniformemente iguales a ε.
- Dados dos polígonos P, Q, con M y N vértices respectivamente, ¿ellos se intersecan?
- ¿Es un polígono dado simple?
- ¿Dos poliedros dados se cortan?
- Dados una colección de N rectángulos, determinar el contorno (o la clausura) de su unión.

3. Problema 1: Intersección de dos segmentos en el plano

Dados 4 puntos p_0,p_1,p_2,p_3 en el plano, consideremos los siguientes problemas:

- ¿Se encuentra el vector p_0p_1 en la dirección de las agujas del reloj o en la dirección opuesta respecto del vector p_0p_2?
- ¿Se intersecan los segmentos p_0p_1 y p_2p_3?

Para resolverlos en forma eficiente, consideremos el producto cruz entre dos vectores en $\mathbb{R}^2$, $p=(x_1,y_1)$, $q=(x_2,y_2)$:

$$p \times q = \begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix} = x_1y_2 - x_2y_1 = -q \times p.$$

Cuando $p \times q > 0$, el giro del vector p hacia el vector q es el sentido positivo; es decir, contrario a las agujas del reloj. La dirección de giro es la que corresponde al menor ángulo entre los vectores p y q. Si el giro es en el sentido horario, entonces $p \times q < 0$. Si $p \times q = 0$, p y q son colineales.

Un dado vector p divide el plano en dos regiones: Una que contiene todos los vectores q tales que $p \times q > 0$ y otra que contiene los vectores r tales que $p \times r < 0$.

Para contestar a la primera pregunta, basta calcular el producto cruz entre los vectores p_0p_1 y p_0p_2. Si $p_0p_1 \times p_0p_2 > 0$, entonces el giro del primero al segundo vector es positivo (hacia la izquierda), caso contrario es negativo (giro hacia la derecha). Si el producto es cero, entonces los puntos son colineales.

Otra posibilidad es determinar las pendientes de los segmentos p_0p_1 y p_0p_2, $m_1 = \frac{dy_1}{dx_1}$ y $m_2 = \frac{dy_2}{dx_2}$. Si $m_1 > m_2$ entonces el giro de p_0p_1 a p_0p_2 es hacia la izquierda, caso contrario es hacia la derecha.

El problema al comparar pendientes es que dx_1 o dx_2 pueden ser iguales a cero, o muy cercanos a cero. Por esto, en lugar de comparar las pendientes, podemos comparar $dx_1 \cdot dy_2$ con $dx_2 \cdot dy_1$. Considere la siguiente función giro escrita en Pascal:

```
función giro (p0,p1,p2 : point) : integer;
var dx1,dx2,dy1,dy2: integer;
  begin
  dx1:=p1.x;dy1:=p1.y-p0.y;
  if dx1*dy2>dy1*dy2 then giro:=1;
  if dx1*dy2< dy1*dy2 then giro:= -1;
  if dx1*dy2=dy1*dy2 then
    begin
    if (dx1*dx2<0) or (dy1*dy2<0)
      then giro:= -1 else
    if (dx1*dx1+dy1*dy1)>=(dx2*dx2+dy2*dy2)
    then giro:= 0 else giro:= 1;
    end;
```

Point es un tipo récord, y aprovechamos la oportunidad para implementar un segmento y un polígono:

```
type point = record x,y : interger end;
  line = record p1, p2 : point end;
var polygon : array[0...Nmax] of point;
```

Giro igual a 0 corresponde al caso en que p_0, p_1, p_2 son colineales con p_2 entre p_0 y p_1. Si p_0 se encuentra entre p_1 y p_2, entonces giro es igual a -1, y si p_1 se encuentra entre p_0 y p_2 giro es igual a 1.

La función intersect determina si dos segmentos se intersecan:

```
function intersect (L1,L2 : line) : boolean;
  begin
  intersect: = ((giro(L1.p1.L1.p2.L2.p1)*
      giro(L1.p1.L1.p2.L2.p2))<=0)
    and ((giro (L2.p1.L2.p2.L1.p1)*
      giro(L2.p1.L2.p2.L1.p2))<=0);
  end;
```

Para determinar si dos segmentos arbitrarios p_0p_1 y p_2p_3 se intersecan, utilizaremos las siguientes estrategias:

1. Rápido rechazo: Dos segmentos no pueden intersecarse si sus cajas fronteras no se intersecan.

   Una caja frontera (*bounding box*) de una figura geométrica es el menor rectángulo que contiene la figura, y cuyos lados son paralelos a los ejes x e y.

   Si $p_0=(x_0,y_0)$, $p_1=(x_1,y_1)$, $p_2=(x_2,y_2)$, $p_3=(x_3,y_3)$, la caja frontera del segmento p_0p_1 es el rectángulo cuyo extremo inferior izquierdo es el punto $\hat{p}_0 = (\hat{x}_0, \hat{y}_0)$ y el extremo superior derecho es el punto $\hat{p}_1 = (\hat{x}_1, \hat{y}_1)$, con $\hat{x}_0 = \min(x_0, x_1)$, $\hat{y}_0 = \min(y_0, y_1)$, $\hat{x}_1 = \max(x_0, x_1)$, $\hat{y}_1 = \max(y_0, y_1)$.

   Dos cajas frontera $(\hat{p}_0, \hat{p}_1), (\hat{p}_2, \hat{p}_3)$ se intersecan, si y solo si la conjunción $I = (\hat{x}_1 \geq \hat{x}_2) \wedge (\hat{x}_3 \geq \hat{x}_0) \wedge (\hat{y}_1 \geq \hat{y}_2) \wedge (\hat{y}_2 \geq \hat{y}_0)$ es verdadera. Las dos primeras comparaciones determinan si las cajas fronteras se intersecan en x, y las dos últimas determinan si existe intersección en y.

2. Decidir si cada segmento intersecan la recta que contiene al otro.

   El segmento p_0p_1 intersecan una recta, si p_0 se encuentra de un lado de la recta y p_1 se encuentra del otro lado. Si p_0 o p_1 se encuentra sobre la recta, decimos que el segmento interseca en la recta.

Entonces, dos segmentos se intersecan si y solo si sus cajas fronteras se intersecan y si cada segmento interseca la recta que contiene al otro segmento.

Podemos utilizar el producto cruz para determinar si p_2p_3 interseca la recta que contiene los puntos p_0,p_1. Si $p_0p_2 \times p_0p_1 < 0$ y $p_0p_3 \times p_0p_1 > 0$ entonces p_0p_2 y p_0p_3 tienen orientaciones opuestas relativas a p_0p_2 y, por lo tanto, los segmentos se intersecan.

Si $p_0p_2 \times p_0p_1 < 0$ y $p_0p_3 \times p_0p_1 > 0$ entonces p_0p_2 y p_0p_3 tienen la misma orientación relativa a p_0p_2 y, por lo tanto, los segmentos no se intersecan.

Las condiciones de frontera ocurren cuando uno de los productos cruz es igual a cero; es decir, 3 de los puntos son colineales.

4. Problema 2: Trayectoria cerrada simple

Consideremos el siguiente problema: Dado un conjunto con n puntos en el plano, determinar la trayectoria que pasa por todos ellos, sin auto intersecarse, de tal forma que se devuelva al punto inicial después de haberse visitado todos los puntos.

Los puntos pueden representar residencias y las trayectorias la ruta que un cartero debe de seguir para visitar cada casa sin cruzar su trayectoria. El problema de determinar la mejor trayectoria se conoce como el problema del vendedor viajero (*traveling salesman problem*) es más complicado que el que estamos tratando.

Una estrategia sencilla para resolver el problema de la trayectoria cerrada simple es escoger uno de los puntos como soporte, determinar los ángulos polares de los segmentos formados al unir los puntos del conjunto al punto soporte, y ordenarlos de acuerdo con su ángulo. Finalmente, se conectan los puntos adyacentes. El resultado es una curva cerrada simple que une los puntos dados.

Por ejemplo, para los puntos de la Figura 1a, si escogemos al punto A como soporte, obtendremos el polígono cerrado simple de la Figura 1b.

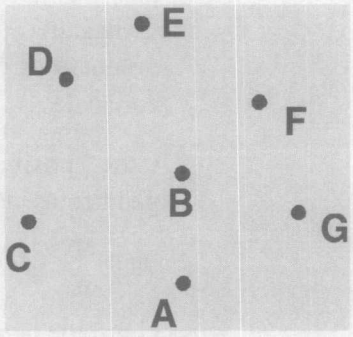

FIGURA 1a

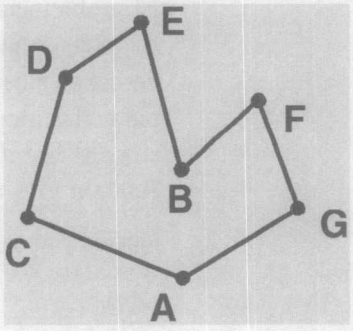

FIGURA 1b

Si dx y dy son respectivamente las componentes horizontal y vertical de la distancia del punto soporte a alguno de los otros puntos, podemos determinar el ángulo polar correspondiente $\arctan\frac{dy}{dx}$.

Pero, debido a que el ángulo es utilizado solamente para ordenar los puntos, es

mejor utilizar otra función que sea más fácil de calcular y que conserve la misma propiedad de ordenamiento que el arco tangente. Un buen candidato para la función es $\frac{dy}{dx+dy}$.

La función

```
function theta(p1, p2 : point) : real;
  var dx, dy, ax, ay : integer;
      t : real;
  begin
  dx := p2.x–p1.x ; ax := abs(dx);
  dy := p2.y-p1.t ; ay := abs(dy);
  if (dx=0) and (dy=0) then t := 0
     else t := dy / (ax + ay);
  if dx<0 then t := 2–t
     else if dy<0 then := 4+t;
     theta := t * 90.0;
     end;
```

devuelve un número entre 0 y 360 que no es el ángulo entre el segmento p_1p_2 y la horizontal, pero que posee las mismas propiedades de orden que el ángulo polar [14] Cap. 24.

5. Problema 3: Inclusión en un polígono

Consideremos ahora el siguiente problema: Determinar si un punto dado se encuentra dentro o fuera de un polígono dado.

Una solución simple consiste en dibujar un segmento rectilíneo L bien largo (para garantizar que el otro extremo estará en el exterior del polígono) con origen en el punto dado, y en cualquier dirección, y contra el número de segmentos del polígono que son cortados por L. Si el número es impar el punto estará dentro y si es par el punto estará fuera.

La función *inside* determina si el punto *t* se encuentra dentro o fuera del polígono, utilizando un segmento de prueba horizontal (para simplificar los cálculos). [14]

```
function inside(t: point) : boolean;
  var count, i, j : integer;
      lt, lp : line;
  begin
  count := 0; j := 0;
  p[0] := p[n] ; p[n+1] := p[1];
  lt.p1 := t; lt.p2 := t; lt.p2.x := maxint;
  for i := 1 to n do
    begin
    lp.p1 := p[i]; lp.p2 := p[i];
    if not intersect (lp,lt) then
      begin
      lp.p2 := p[j]; j := i;
      if intersect (lp, lt) then count :=count + 1;
      end;
    end;
  inside := ((count mod 2)=1);
  end;
```

p[1] es el punto con la menor coordenada x entre todos los puntos que tienen la menor coordenada y, tal que si p[1] se encuentra en el segmento de prueba L entonces p[0]$\notin$ L.

La variable j indexa el último punto sobre el polígono que no pertenece al segmento de prueba. Si el polígono es convexo, cualquier segmento de prueba lo interseca a lo sumo en dos segmentos. En tal caso es mejor usar un procedimiento como el de búsqueda binaria para determinar en $O(\log n)$ pasos si un punto se encuentra dentro o fuera del polígono.

6. Problema 4: Intersección de segmentos de un conjunto

Sea S un conjunto de segmentos. Queremos determinar si algún par de elementos de S se interseca.

El algoritmo utiliza una técnica conocida como barrido (*sweeping*) que es común a muchos algoritmos geométricos. La recta de barrido es una recta vertical imaginaria que barre el conjunto de segmentos S de izquierda a derecha.

Supongamos que:

1. Ninguno de los segmentos en S es vertical.
2. Tres elementos de S no pueden intersecarse en un punto.

Debido a la primera hipótesis, cualquier elemento de S que interseca la línea de barrido, lo hace en un único punto. De esta forma, podemos ordenar los elementos de S que intersecan la línea de barrido de acuerdo con las ordenadas y de los puntos de intersección.

Los elementos S_1 y S_2 son comparables en L si la línea de barrido L intersecan a ambos.

Decimos que $S_1 >_L S_2$, si S_1 y S_2 son comparables, y si $S_1 \cap L$ se encuentra por encima de $S_2 \cap L$.

Dado L, $>_L$ define una relación de orden total sobre los elementos de S que intersecan la línea de barrido L.

Cuando la línea de barrido L pasa por el punto de intersección de dos segmentos, sus posiciones en el orden total $>_L$ se invierten.

Por la hipótesis 2, dados dos segmentos a y b en S, existe alguna línea de barrido L tal que en el orden total $>_L$, a y b son segmentos consecutivos. Cualquier línea de barrido que pasa por la región marcada tendrá a los segmentos a y b como consecutivos en el orden total (Figura 2).

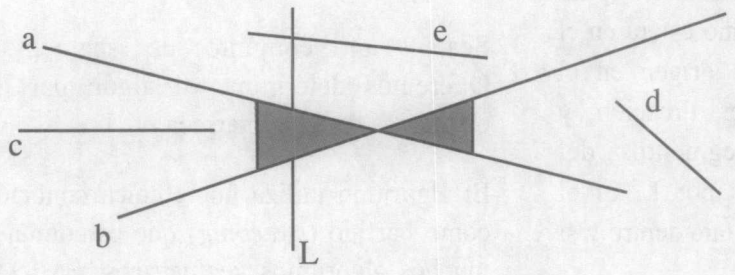

FIGURA 2
a* y *b* consecutivos respecto a *L

El algoritmo utiliza dos conjuntos de datos:

1. *T* el estado de la línea de barrido.

   T proporciona una relación de orden total entre los objetos intersecados por la línea de barrido.
2. El calendario con puntos de eventos. Es una sucesión de abscisas x ordenadas de izquierda a derecha, que define los puntos de salida (*halting*) de la línea de barrido. Cada posición de salida se denomina punto de evento. Los cambios en el estado de la línea de barrido ocurren en los puntos de evento (extremos derechos de cada segmento).

Insertaremos un segmento en el conjunto T cuando su extremo izquierdo es encontrado, y lo borraremos cuando su extremo derecho es encontrado. Cuando dos segmentos son consecutivos en el orden total, chequearemos si ellos se intersecan.

Las operaciones sobre T son: [1]

- insert(T,s) : inserta un segmento s en T.
- delete(T,s) : borra un segmento s de T.
- above(T,s) : devuelve el segmento que se encuentra inmediatamente por encima de s en T.
- below(T,s) : devuelve el segmento que se encuentra inmediatamente por debajo de s en T.

Si S contiene n elementos, cada operación mencionada toma tiempo en $O(\log n)$ si utilizamos árboles rojo-negro (*red-black*). Podemos reemplazar las comparaciones de llaves por comparaciones producto cruz que determinan el orden relativo de dos segmentos.

La entrada para el algoritmo es el conjunto S con n segmentos, y la salida es verdadero si existen dos elementos en S que se intersecan y falso en el caso contrario. El orden total T es implementado utilizando un árbol *red-black*.

Any-segments-intersect (S)

1. T←0
2. ordenar los puntos extremos de los segmentos de S de izquierda a derecha.
3. for cada punto p en la lista ordenada de puntos extremos
4. do if p es el punto extremo izquierdo de un segmento s
5. then insert(T,s)
6. if (above(T,s) existe e interseca s) or (below(T,s) existe e interseca s)
7. then return true
8. if p es el punto extremo derecho de un segmento s
9. then if existen above(T,s) y below(T,s) y above(T,s) interseca below(T,s)
10. then return true
11. delete(T,s)
12. return false.

En [1] se demuestra el teorema de *correctitud* del algoritmo:

Teorema: Any-segments-intersect (S) devuelve *true* si y solo si existe alguna intersección entre segmentos de S.

Para determinar el tiempo de ejecución del algoritmo vemos que el tiempo de ejecución correspondiente a la línea:

1. se encuentra en $O(1)$.
2. se encuentra en $O(n)$ si usamos *merge sort* o *heapsort*.

3-11, el *loop for* iteractúa a lo más 2n veces (para los 2n puntos de evento) y cada iteracción toma tiempo en $O(\log n)$ debido a que cada operación *red-black* toma tiempo en $O(\log n)$. Cada prueba de intersección toma tiempo constante si utilizamos el producto cruz. Por lo tanto, el tiempo total se encuentra en $O(n \log n)$.

El algoritmo anterior es debido a Shamos y Hoey [15].

7. Problema 5: La cáscara convexa

La cáscara convexa de un conjunto S de puntos, CH(S), es el menor polígono convexo P tal que cada punto de S se encuentra en la frontera de P o en su interior.

Intuitivamente, si pensamos que cada punto de S es un palillo en un tablero, CH(S) es la firgura formada por una liga de hule tensa que rodea a los palillos.

El primer método que veremos para resolver el problema es debido a Graham [4], y consiste en mantener una pila Q con los puntos que son candidatos a vértices de CH(S). Cada punto de S (conjunto de entrada) es metido en la pila, y los puntos que no son vértices de CH(S) son sacados de ella. Cuando el algoritmo finaliza, Q contendrá los vértices de CH(S) en el orden contrario a las agujas del reloj.

La entrada es un conjunto S con n puntos del plano, $n \geq 3$. Se invoca la función TOP(Q) que devuelve el punto que se encuentra en la parte superior de la pila, sin cambiarla. La función NEXT-TO-TOP(Q) devuelve el punto inmediatamente por debajo del *top* de Q, sin cambiar Q. Al finalizar, la salida Q contendrá de abajo hacia arriba, exactamente los vértices de CH(S) en orden contrario a las agujas del reloj.

GRAHAM-SCAN(S)

1. Sea $p_0 \in S$ con coordenada "y" mínima.
2. Sea $\{p_1, p_2, \ldots, p_m\}$ la secuencia con los restantes puntos de S ordenados por ángulo polar.
3. $Q \leftarrow 0$
4. PUSH(p_0, Q)
5. PUSH(p_1, Q)
6. PUSH(p_2, Q)
7. for i←3 to m
8. do while el ángulo formado por los puntos NEXT-TO-TOP(Q), TOP(Q) y p_i corresponde a un giro hacia la izquierda
9. do POP(Q)
10. PUSH(p_i, Q)
11. return Q.

En la Figura 3 mostramos como se ejecuta el algoritmo. En el ejemplo S contiene 8 puntos. Se escoge p_0 con menor

coordenada $-y$ (y menor coordenada x en caso de empate). p_0 es uno de los vértices de CH(S). Se ordena los puntos restantes de S de acuerdo con el ángulo polar relativo a p_0. Si dos o más puntos tienen el mismo ángulo polar relativo a p_0, entonces ellos son combinaciones convexas de p_0. Tomamos el punto más alejado en el radio vector correspondiente. Las líneas 3-6 inicializan la pila Q con p_0,p_1,p_2.

El loop for, líneas 7-10 iteractúa una vez para cada punto en la secuencia $\{p_3,p_4,\ldots,p_m\}$ de tal forma que después de procesar el punto p_i Q contenga de abajo hacia arriba, los vértices de CH($\{p_0,p_1,\ldots,p_i\}$) en el sentido positivo.

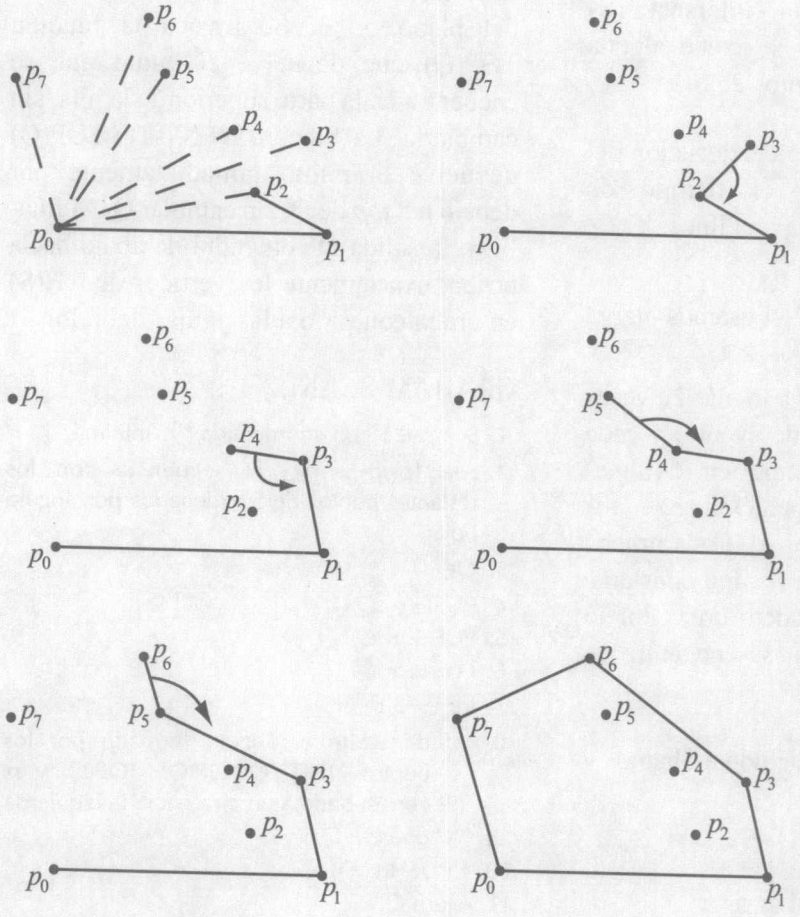

FIGURA 4
Cáscara convexa de 8 puntos

El loop while, líneas 8-9 remueve puntos de la pila que no son vértices de CH. Cada vez que recorremos el CH en sentido positivo hacemos un giro hacia la izquierda en cada vértice. Por otro lado, cada vez que el *loop while* encuentra un vértice en el cual se realiza un giro hacia la derecha entonces el vértice es sacado de la pila.

Después de sacar todos los vértices con giros hacia la derecha en la ruta correspondiente a p_i metemos a p_i en la pila. Al finalizar se devuelve la pila Q.

En [1] se demuestra el teorema de *correctitud* del algoritmo de Graham:

Teorema: Si Graham-scan corre sobre un conjunto S con n puntos, $n\geq 3$ entonces un punto de S pertenece a la pila Q al finalizarlo si y solo si es un vértice de CH(S).

Para el tiempo de ejecución tenemos:

La línea 1 toma tiempo en $\Theta(n)$.

La línea 2 toma tiempo en $O(n \log n)$ utilizando *merge sort* o *heapsort* para ordenar los ángulos polares, y el método del producto cruz para compararlos.

Remover todos excepto el punto más alejado con un mismo ángulo polar, puede ser hecho en tiempo lineal.

Las líneas 3-6 toman tiempo constante.

El *loop for*, líneas 7-10, se ejecuta n-3 veces. Cada PUSH toma tiempo constante, y por lo tanto cada iteracción toma tiempo constante, excepto el tiempo tomado por el *loop while* en las líneas 8-9. Por lo tanto, todo el *loop for* toma tiempo lineal, excepto para el *loop while* anidado.

Si utilizamos el método agregado del análisis amortiguado, podemos mostrar que el *loop while* toma tiempo total lineal, debido a que existe a los sumo una operación POP para cada operación PUSH. Concluimos que el tiempo total se encuentra en $O(n \log n)$.

El otro algoritmo debido a Jarvis [8], denominado la frontera de Jarvis, utiliza una técnica que simula el envolver al conjunto S con un trozo de papel muy tenso. Empezamos con algún punto p_0 que garantizamos ser un vértice de CH(S) (por ejemplo, el punto con menor coordenada y, y menor coordenada x en caso de empate), trazamos un segmento rectilíneo L con origen en p_0 en la dirección positiva y procedemos a barrer en el sentido contrario a las agujas del reloj, hasta que L encuentre a otro punto p_1 de S. Podemos imaginar que esto corresponde a pegar el papel en p_0 y jalarlo en forma tensa, hasta que toque otro punto. Entonces, p_1 es otro vértice de CH(S). Manteniendo el papel tenso, seguimos el procedimiento anterior hasta que el regalo sea completamente envuelto; es decir, hasta que regresemos al punto inicial p_0.

La frontera de Jarvis consiste en construir una secuencia $H=\{p_0,p_1,\ldots,p_{h-1}\}$ de vértices de CH(S). Empezamos con p_0. El siguiente vértice p_1 de CH(S) tiene el menor ángulo polar respecto a p_0 (caso haya empate, seleccionamos el punto más alejado de p_0). El siguiente punto p_2 tiene el menor ángulo polar respecto a p_1, etc.

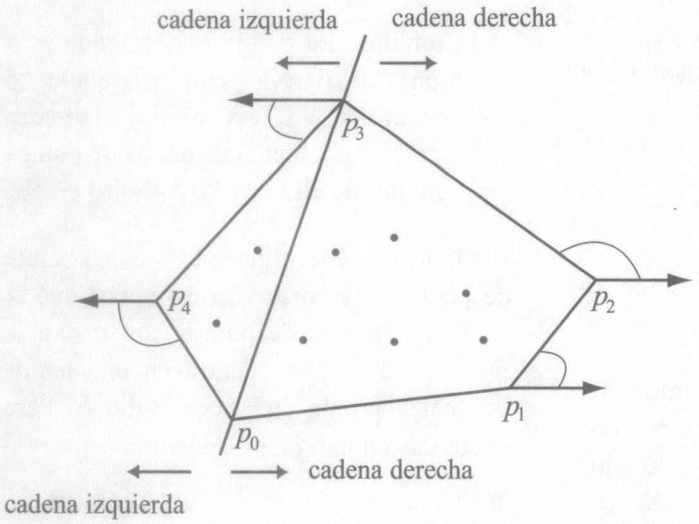

FIGURA 4

Cuando alcanzamos el vértice más alto p_k, hemos construido la cadena derecha de CH(S). Para construir la cadena izquierda, empezamos en p_k y escogemos p_{k+1} como el punto con menor ángulo polar respecto del eje x negativo. Seguimos el proceso hasta volver al vértice original p_0.

Podemos implementar la frontera de Jarvis haciendo barrido alrededor de la cáscara convexa, sin construir en forma separada las cadenas derecha e izquierda.

La función *wrap* encuentra la cáscara convexa de un vector $p[1\ldots n]$ de puntos. La base de esta implementación es la función theta (p_1,p_2 : point) que hemos visto anteriormente. $p[n+1]$ se usa para mantener un centinela.

```
function wrap : integer;
  var i, min, m : integer;
      minangle, v : real;
      t : point;
  begin
  min := 1;
  for i := 2 to n do
    if p[i].y < p[min].y then min := i;
  m := 0; p[n+1] := p[min]; minangle := 0.0;
  repeat
    m := m+1; t := p[m];
    p[m] := p[min]; p[min] := t;
    min := n+1; v := minangle;
    minangle := 360.0;
    for i := m+1 to n+1 do
      if theta(p[m],p[i]) > v then
        if theta(p[m], p[i]) < minangle then
          begin
            min := 1;
            minangle := theta(p[m], p[min])
          end;
  until min = n+1;
  wrap := m;
  end;
```

Copiamos el punto con menor coordenada "y" en p[n+1], v es el ángulo actual de barrido y m es el número de puntos incluidos en la cáscara convexa. El *loop repeat* pone el último punto en la cáscara convexa, intercambiando con el m-ésimo punto y usamos la función theta para calcular el ángulo respecto a la horizontal hecho por el segmento formado por el

punto y cada punto incluido en la cáscara convexa y determinando el menor de ellos. El *loop* termina cuando se vuelve a encontrar una copia del primer punto en p[n+1].

Este programa puede o no devolver los puntos que se encuentran sobre un eje de la cáscara convexa.

La principal desventaja del algoritmo es que en el peor de los casos, cuando todos los puntos se encuentran sobre la cáscara convexa, el tiempo de ejecución es proporcional a n^2. Si es implementado adecuadamente, el algoritmo de la frontera de Jarvis toma tiempo en $O(nh)$, con h el número de vértices de CH(S).

8. Problema 6: Puntos más cercanos

Sea S un conjunto de n puntos ($n \geq 2$) en el plano. Deseamos determinar la pareja más cercana de puntos S; es decir, cuya distancia euclideana sea mínima.

Este problema tiene aplicaciones en sistema de control de tráfico (determinar los dos vehículos más cercanos para detectar potenciales colisiones), y en robótica.

A fuerza bruta podemos determinar la distancia entre todas las parejas de puntos S y verificar la menor de ellas. Esto toma tiempo $\binom{n}{2} = \Theta(n^2)$.

Utilizaremos la estrategia divide y conquista para reducir el tiempo a $Q(n \log n)$.

Cada llamada recursiva del algoritmo toma como entrada un subconjunto P de S y dos vectores X, Y que contienen todos los puntos de P. Los puntos de X son ordenados en orden creciente de las abscisas x, y los de Y ordenados en orden creciente de las ordenadas y.

En la llamada recursiva con entradas P,X,Y, chequeamos si $|P| \leq 3$. Si este es el caso, utilizamos el método de fuerza bruta descrito anteriormente. Si $|P| > 3$ la llamada efectúa las siguientes operaciones:

1. Divide: Determina la recta vertical L que biseca P en dos conjuntos P_L, P_R con

$$P_L = \left\lceil \frac{|P|}{2} \right\rceil, P_R = \left\lfloor \frac{|P|}{2} \right\rfloor$$

   tales que todos los puntos P_L se encuentran sobre L o a la izquierda de L, y todos los de P_R se encuentran sobre o a la derecha de L.

   X se divide en vectores X_L, X_R que contienen los puntos de P_L y P_R respectivamente, ordenados en forma creciente de la abscisa x. Análogamente, Y se divide en Y_L y Y_R con puntos de P_L y P_R respectivamente, ordenados en forma creciente de la ordenada y.

2. Conquista: Para la llamada con entrada P_L,X_L,Y_L determine la pareja más cercana de puntos en P_L con distancia δ_L, y para la llamada con entrada P_R,X_R,Y_R, determine la pareja más cercana de puntos en P_R con distancia δ_R. Sea $\delta = \min(\delta_L, \delta_R)$.

3. Combina: La pareja más cercana es o bien la pareja con distancia δ encontrada en una de las llamadas recursivas o, bien, una pareja de puntos con uno de ellos en P_L y el otro en P_R.

El algoritmo determina si existe una pareja de puntos cuya distancia es menor que δ. Cualquier pareja de puntos con distancia menor que δ se encuentra en una banda vertical centrada en L con radio δ. Para encontrar tal pareja, el algoritmo:

a) Crea un vector Y' obtenido de Y, eliminando todos los puntos que no se encuentran en la banda vertical de ancho 2δ. Y' es ordenado como Y.

b) Para cada $p \in Y'$, el algoritmo busca puntos en Y' cuya distancia a p es menor o igual que δ.

Si $|Y'|=m$ entonces el algoritmo calcula la distnacia de p a cada uno de los $m-1$ puntos restantes de Y'. Sea δ' la menor de las distancias.

c) Si $\delta'<\delta$ la banda vertical contiene una pareja más cercana que la encontrada por las llamadas recursivas, y devuelve δ' con la pareja correspondiente como solución. Caso contrario, se devuelve la pareja con su distancia δ encontrada por la llamada recursiva.

En [1] se demuestra el teorema de *correctitud* del algoritmo anterior:

Teorema: El algoritmo de la pareja más cercana devuelve la distancia y la pareja más cercana de puntos de un conjunto S.

Para la implementación, observemos que si X recibido por la llamada recursiva se encuentra ordenado, entonces la división de P en P_L, P_R se lleva a cabo en tiempo lineal. En cada llamada, la entrada P necesita ser dividida. Lo mismo ocurre con el vector Y, y necesitamos que las particiones sean ordenadas en tiempo lineal.

Considere el pseudocódigo que se encarga de esta tarea:

```
1. longitud[Y_L]←longitud[Y_R]←0
2. for i←1 to longitud[Y]
3.    do if Y[i] ∈ P_L
4.       then longitud[Y_L]←longitud[Y_L]+1
           Y[longitud[Y_L]]←Y[i]
         else longitud[Y_R]←longitud[Y_R]+1
           Y[longitud[Y_R]]←Y[i]
```

Por lo tanto, si el punto $Y[i] \in P_L$, lo agregamos al final del vector Y_L, caso contrario lo agregamos al final del vector Y_R. Existen pseudocódigos parecidos para los vectores X_L,X_R,Y'. Los puntos que ocupan los primeros lugares en el vector son preordenados; es decir, ordenados antes de la primera llamada. Entonces,

$$T(n)=\begin{cases}2T(n/2)+O(n) & \text{si } n>3\\ O(1) & \text{si } n\le 3\end{cases}$$

Concluimos que el tiempo de ejecución de cada llamada recursiva es $T(n) \in O(n \log n)$. Si T'(n) es el tiempo de ejecución del algoritmo completo, entonces

$$T'(n)=T(n)+O(n \log n)$$

debido a que el preordenamiento incrementa un tiempo de ejecución en $O(n \log n)$. Por lo tanto, $T'(n) \in O(n \log n)$.

El algoritmo anterior es debido a Shamos y aparece en [13].

9. Bibliografía

[1] Cormen T.H.; Leiserson C.E.; Rivest R.L. *Introduction to Algorithms*. McGraw-Hill, 1990.

[2] Crilly A.J.; Earnshaw R.A.; Jones H. *Fractal and Chaos*. Springer-Verlag, 1991.

[3] Day A.M. "The Implementation of an Algorithm to Find the Convex Hull of a Set of Three-Dimensional Points". ACM Transactions on Graphics, 9(1), 1990.

[4] Edelsbrunner H. "Algorithms in Combinatorial Geometry". Vol 10 of EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1987.

[5] Graham R.L. "An efficient algorithm for determining the convex hull of a finite planar set". Information Processing Letters, 1:132-133, 1972.

[6] Graham R.L. & Pavol Hell. "On the history of the minimum spanning tree problem". Annals of the History of Computing, 7(1):43-57, 1985.

[7] Graham R., Yao F. "A Whirlwind Tour of Computational Geometry". The American Mathematical Monthly, 97(8):687-701, 1990.

[8] Graham R., Yao F. "Finding the convex hull of a simple polygon". J. Algorithms, 4:324-331, 1983.

[9] Jarvis R.A. "On the identification of the convex hull of a finite set of points in the plane". Information Processing Letters, 2:18-21, 1973.

[10] Kikpatrick D.G. & Seidel R. "The ultimate planar convex hull algorithm?" SIAM Journal on Computing, 15(2):287-299, 1986.

[11] Kurt Mehlhorn. "Multidimensional Searching and Computational Geometry". Vol. 3 of Data Structures and Algorithms. Springer-Verlag, 1984.

[12] Lee D.T. "On finding the convex hull of a simple polygon". Int. J. Comput. Inform. Sci., 12:87-98, 1983.

[13] Parker T.S.; Chua L.O. "Practical Numerical Algorithms for Chaotic Systems". Springer-Verlag, 1989.

[14] Preparata F.P. & Hong S.J. "Convex hull of a finite set of points in two and three dimensions". Commun. ACM 20(2), 1977.

[15] Preparata F.P.; Shamos M.L. "Computational Geometry. An Introduction". Springer-Verlag, 1990.

[16] Sedgewick R. "Algorithms", 2ª. edición. Addison-Wesley Pub. Co., 1988.

[17] Shamos M.I. & Hoey D. "Geometric intersection problems". Proceedings of the 16th Annual Symposium on Foundations of Computer Science, pages 208-215. IEEE Computer Society, 1975.

[18] Su Bu-qing; Liu Ding-yuan. "Computational Geometry. Curve and Surface Modeling". Academic Press, Inc., 1989.

[19] Yao C., Andrew C. "A lower bound to finding convex hulls". Journal of the ACM, 28(4):780-787, 1981.

[20] Yao F.F. "Computational Geometry". Handbook of Theoretical Computer Science. North Holland, Amsterdam, 1990.