Novática, revista fundada en 1975, es el órgano oficial de expresión y formación continua de ATI (Asociación de Técnicos de Informática). *Novática* edita también *Upgrade*, revista digital de CEPIS (*Council of European Professional Informatics Societies*), en lengua inglesa.

<http://www.ati.es/novatica/>
<http://www.upgrade-cepis.org/>

ATI es miembro de CEPIS (*Council of European Professional Informatics Societies*) y tiene un acuerdo de colaboración con ACM (*Association for Computing Machinery*). Tiene asimismo acuerdos de vinculación o colaboración con AdaSpain, AI² y ASTIC

Coordinación Editorial y Redacción Central (ATI Madrid)
Padilla 66, 3º, dcha., 28006 Madrid
Tlf.914029391; fax.913093685 <*novatica@ati.es*>
Composición, Edición y Redacción ATI Valencia
Palomino 14, 2ª, 46003 Valencia
Tlf./fax 963918531 <*secreval@ati.es*>
Administración y Redacción ATI Cataluña
Via Laietana 41, Iº, 1ª, 08003 Barcelona
Tlf.934125235; fax 934127713 <*secregen@ati.es*>
Redacción ATI Andalucía
Isaac Newton, s/n, Ed. Sadiel, Isla Cartuja 41092 Sevilla
Tlf./fax 954460779 <*secreand@ati.es*>
Redacción ATI Aragón
Lagasca 9, 3-B, 50006 Zaragoza
Tlf./fax 976235181 <*secreara@ati.es*>
Redacción ATI Asturias-Cantabria <*gp-astucant@ati.es*>
Redacción ATI Castilla-La Mancha <*gp-clmancha@ati.es*>
Redacción ATI Galicia
Recinto Ferial s/n, 36540 Silleda (Pontevedra)
Tlf.986581413; fax 986580162 <*secregal@ati.es*>
Suscripción y Ventas: <*http://www.ati.es/novatica/interes.html*>, o en ATI Cataluña y ATI Madrid
Publicidad: Padilla 66, 3º, dcha., 28006 Madrid
Tlf.914029391; fax.913093685 <*novatica.publicidad@ati.es*>

Imprenta: 9·Impressió S.A., Juan de Austria 66, 08005 Barcelona.
Depósito Legal: B 15.154-1975
ISBN: 0211-2124; CODEN NOVAEC

Portada: Antonio Crespo Foix / © ATI 2002

novática
UPGRADE
Revista de la Asociación de Técnicos de Informática

SEPTIEMBRE - OCTUBRE 2002

159

SUMARIO

Monografía del próximo número:
«Seguridad en Comercio/Negocio Electrónico»

Programar es crear

No taléis el bosque por culpa de los árboles: solución

Ángel Herranz[1], Julio Mariño[1], Manuel Carro[1], Pablo Sánchez[2]
[1] Facultad de Informática, Universidad Politécnica de Madrid; [2] Andago

<{aherranz, jmarino, carro@}fi.upm.es>, <psanchez@skyrealms.org>

El enunciado de este problema apareció en el número 158 de Novática (julio-agosto 2002, p. 73). Es el programa D de los planteados en el 25º Concurso Internacional de Programación de la ACM (2001)

1. Talando árboles

En el problema **D** se planteaba la búsqueda del máximo número de árboles que podían ser talados en un bosque asegurando que en sus caídas ninguno de ellos impactaba y, por tanto, dañaba a otros árboles o al muro que rodea el recinto rectangular que delimita el bosque.

El enunciado es ambiguo en el sentido que no establece hacia dónde cae un árbol cuando se tala. La duda está entre si cae aleatoriamente hacia cualquier dirección o si es posible elegir la dirección hacia la que dicho árbol se derriba. En el segundo caso, tras la tala, el árbol parecería elegir caprichosamente hacia dónde desplomarse. Afortunadamente, esta ambigüedad en el enunciado se puede resolver analizando el único caso de prueba: si no se tuviese control sobre la dirección de caída no podríamos arriesgarnos a talar ningún árbol del ejemplo de entrada. Pero, puesto que la salida para el ejemplo de entrada nos dice que sí es posible talar dos árboles, hemos de asumir que la dirección de la caída de un árbol está bajo nuestro control. Esto hace que el problema sea más realista y a la vez más complicado de resolver, como se verá en la sección 3.

Queremos resaltar el hecho de que en este problema parte de la información para la correcta interpretación del enunciado está en uno de los casos de prueba. Esto no es extraño, pero sí puede llevar a error a quienes no presten atención a los casos de prueba más que a la hora de comprobar que el programa resuelve correctamente un problema aparentemente comprendido. Por otro lado, puede verse como una versión *debilitada* de una clase de problemas que aparentemente no dan información suficiente para solucionarlos, pero en los que suponer que existe una solución es suficiente para llegar a ella.

Volviendo a la tala, el problema no parece presentar excesiva dificultad si nos movemos en un nivel de abstracción suficiente. Asumiendo que bosque es una variable reescribible (como la que hallaríamos en cualquier lenguaje imperativo), un posible algoritmo que lo resuelve es el siguiente:

1. bosque := "bosque inicial"
2. REPETIR bosque := "bosque después de eliminar árboles que puedan ser talados"
3. HASTA "bosque no ha cambiado"

Decidimos elegir aleatoriamente el árbol a talar; dicha elección está justificada: talar un árbol dado no va a hacer que otros candidatos anteriores dejen de serlo (aunque sí puede hacer que aparezcan más candidatos). Es decir, mantiene cierta idea de monotonía sobre el conjunto de árboles talables. Esta monotonía permite usar el algoritmo anterior que, inevitablemente, recuerda la formulación de un *punto fijo*.

A diferencia de la resolución de otros problemas, en que se ofrecía el código completo, en este hemos decidido no profundizar en ciertos detalles (por ejemplo, funciones concretas de intersección) que son más engorrosos que interesantes. Sin embargo sí que ofreceremos información suficiente como para llegar a programarlas. El lenguaje de implementación elegido ha sido Haskell.

2. Primeros pasos en la implementación

La implementación se basa en un operador genérico de punto fijo (`puntoFijo`) que utiliza una función de talado (`talar`) para el paso del algoritmo y un predicado `sePuedeTalar`, cuya implementación de momento vamos a ignorar puesto que será analizada en las siguientes secciones.

La estructura de datos elegida para representar el bosque es un registro con campos[1] para representar el tamaño de la muralla (`xmax` e `ymax`, donde se ha asumido una transformación para que los límites sur y oeste del muro sean los ejes de coordenadas.) y el conjunto de árboles (`arboles`):

```
data Bosque = Bosque {
       xmax :: Float,
       ymax :: Float,
       arboles :: [Arbol]
     }

nArboles :: Bosque -> Int
-- "nArboles b" calcula el número de
-- árboles en el bosque "b"
nArboles = length . arboles
```

Se introducen, además, un par de definiciones de tipos para representar árboles y sus posiciones:

```
-- Coordenadas para la posición
type Posicion = (Float, Float)

-- Posición, radio de la base y
-- altura del árbol
type Arbol = (Posicion, Float, Float)
```

La operación `talar` se codifica de esta forma:

```
talar :: Bosque -> Bosque
-- "talar b" devuelve un nuevo bosque
-- después de eliminar del bosque "b"
-- árboles que se pueden talar
talar bosque =
```

```
  bosque {
    arboles =
      filter ((not.sePuedeTalar) bosque)
             (arboles bosque)
  }
```

Para terminar esta sección se ofrece la codificación del operador de punto fijo y la solución del problema:

```
puntoFijo :: Eq a => (a -> a) -> a -> a
-- precondición: la función "f" tiene
--               que ser monótona y
--               convergente en un
--               número finito de pasos
puntoFijo f x = if x == x'
                then x
                else puntoFijo f x'
                where x' = f x

solucion :: Bosque -> Int
solucion bosque =
  (nArboles bosque) -
  (nArboles (puntoFijo talar bosque))
```

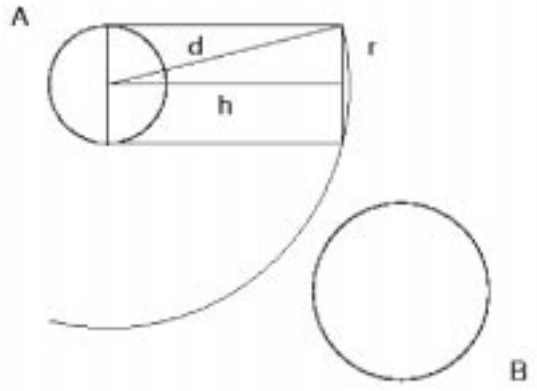
3. ¿Cuándo se puede talar un árbol? Razonamiento geométrico

En esta sección se estudia el problema de cuándo un árbol puede ser talado. Las restricciones que se describen en el enunciado establecen que para poder talar un árbol, éste no puede impactar en su caída con otros o con el muro. Para poder razonar sobre dicha restricción utilizaremos la siguiente notación: nos referiremos a los árboles con letras mayúsculas *A*, *B*, *C*, etc., al punto centro de la base de un árbol *A* con el punto c_A, al radio de la base con r_A y a la altura del árbol con h_A.

La idea en la que se apoyará todo el razonamiento y el algoritmo final para decidir si un árbol puede o no ser talado es la siguiente: dado un árbol candidato a ser talado, tanto el muro como el resto de los árboles imponen, cada uno de ellos, una restricción en la dirección hacia la que el árbol se puede tirar. Por tanto, determinar si un árbol puede o no ser talado pasa por ir restringiendo el intervalo de direcciones posibles inicial $[0,2\pi)$ (0 indica dirección este) con las limitaciones impuestas por cada objeto (muro y resto de árboles).

La idea es resolver el problema en varios pasos:

1. Eliminar del estudio de un árbol *A* aquellos árboles *B* que estén fuera del *alcance* de su caída:

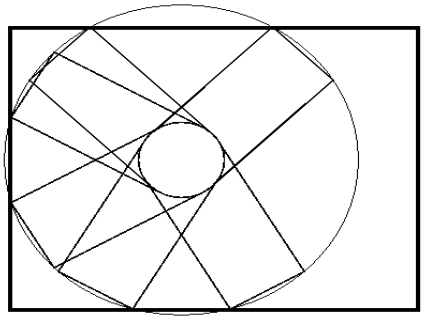


La única dificultad para obtener las ecuaciones estriba en que el *alcance* de un árbol no lo marca su altura, sino el segmento *d* que puede verse en el gráfico anterior. Por tanto, si se cumple la propiedad

$$d_A = \sqrt{h_A^2 + r_A^2}$$
$$d_A + r_B \leq |c_A - c_B|$$

entonces el árbol *B* no impone restricciones a la hora de talar el árbol *A*.

2. Analizar las restricciones sobre la dirección de caída impuestas por la existencia del muro:

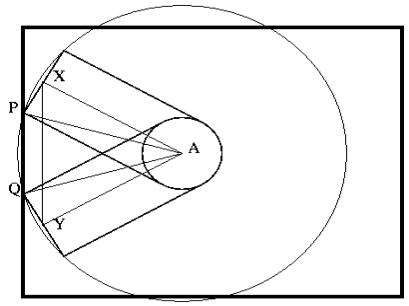


Para comenzar eliminaremos el caso en el que el árbol no impactaría con el muro de ninguna manera por estar demasiado lejos:

$$0 \leq c_{Ax} - d_A$$
$$c_{Ax} + d_A \leq xmax$$
$$0 \leq c_{Ay} - d_A$$
$$c_{Ay} + d_A \leq ymax$$

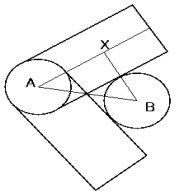
donde $c_A = (c_{Ax}, c_{Ay})$ y *xmax* e *ymax* son los límites Este y Norte, respectivamente, del muro. Si se da este caso, entonces la presencia del muro no añade restricciones a la dirección de caída.

Si, por el contrario, el árbol puede impactar contra el muro en su caída (tal como sucede en la figura anterior) hemos de ser capaces de calcular ángulos como *A* en el triángulo ΔXAY:



lo cual es relativamente sencillo si averiguamos los puntos de corte *P* y *Q*, calculamos el ángulo *A* de ΔPAQ y sumamos el angulito *A* de ΔXAP (arcsin $\frac{r_A}{d_A}$)

3.Volvamos a los árboles que están tan cercanos al árbol a talar como para provocar que en su caída haya un impacto. Distinguimos aquí dos subcasos. Si el árbol está suficientemente cerca, la restricción en la dirección de caída sale de un cálculo de tangentes:

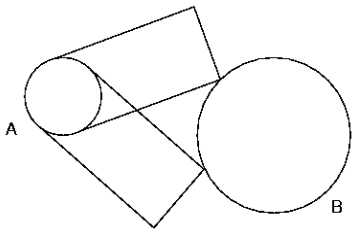


La condición que se ha de cumplir para que nos encontremos en este caso es la siguiente:

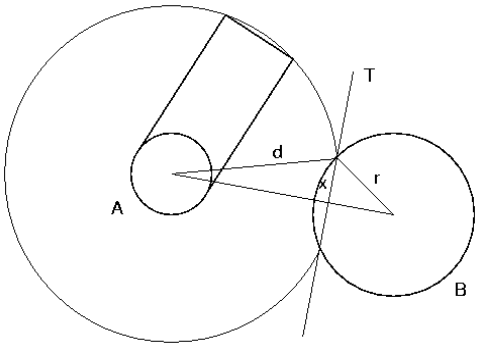
$$h_A > \sqrt{d_A - (r_A + r_R)^2}$$

La resolución del triángulo D*XAB* es sencilla teniendo en cuenta que ha de ser rectángulo (por la tangente) y que tanto *AB* como *BX* ($= r_A + r_B$) son conocidos.

4. En el segundo caso tendremos:



Para este caso se han de calcular los puntos de corte entre la circunferencia que es la base del árbol *B* y la circunferencia con centro en c_A y radio d_A:



y posteriormente se calculan los ángulos de forma análoga al caso que hemos visto para el muro.

Calcular la intersección entre dos circunferencias (que sabemos que se cortan en dos puntos) es más pesado que difícil. Si los centros de las circunferencias son $A = (x_A, y_A)$ y $B = (x_B, y_B)$ y llamamos a los radios r_A y r_B, la ecuación para un punto de corte $C = (x_C, y_C)$ es

$$\left.\begin{aligned} (x_C - x_A)^2 + (y_C - y_A)^2 &= r^2_A \\ (x_C - x_B)^2 + (y_C - y_B)^2 &= r^2_B \end{aligned}\right\}$$

es decir,

$$\left.\begin{aligned} x^2_C - 2x_A x_C + x^2_A + y^2_C - 2y_A y_C + y^2_A &= r^2_A \\ x^2_C - 2x_B x_C + x^2_B + y^2_C - 2y_B y_C + y^2_B &= r^2_B \end{aligned}\right\}$$

La historia continúa así: restando ambas ecuaciones se eliminan los términos cuadráticos en x_C e y_C obteniéndose una ecuación lineal que nos permite poner y_C en función de x_C. Sustituyendo esta expresión en una de las ecuaciones de arriba se obtiene una ecuación cuadrática que se resuelve por el método tradicional, proporcionando dos soluciones reales para x_C que a su vez nos proporcionan los dos valores de y_C.

4. Detallando la implementación

Como ya hemos comentado, el detalle de la solución a este problema es más tedioso que ingenioso. Lo que sí puede resultar interesante es desarrollar algunas de las abstracciones que pueden ayudarnos a mantener ese tedio dentro de unos márgenes razonables. Una de esas abstracciones se refiere al manejo de *intervalos* -- de ángulos en este caso. Ya hemos visto que la solución se reduce a encontrar, en un bosque dado, un árbol que pueda ser talado con garantías. Esto a su vez es equivalente a encontrar un ángulo en el que podemos dejar caer el árbol sin impactar en otro árbol o en el muro del bosque.

Si definimos funciones capaces de calcular, dados un par de árboles, el sector de circunferencia que representa los ángulos seguros para dejar caer el primero sin impactar en el segundo, podemos tratar el conjunto de árboles viendo si la intersección de dichos sectores de circunferencia es no vacía.[2]

La intersección de dos sectores de circunferencia resultará, en general, en una unión disjunta de sectores, que representaremos mediante una lista de pares:

```
type Angulo = Float
type Sector = [ (Angulo, Angulo)]
```

Aquí es esencial fijar los *invariantes de la representación* para reducir casos y obtener la máxima eficiencia al programar la operación de intersección. Supondremos que los ángulos están ordenados crecientemente tanto dentro de cada par como en la lista, y que se mantienen siempre dentro del rango $[0,2\pi)$. Eso quiere decir, por ejemplo, que el sector circular $[-\pi/4,\pi/4]$ vendría representado por la unión disjunta $[0,\pi/4]\cup[2\pi-\pi/4,2\pi)$. El código para el cálculo de intersecciones de intervalos podría quedar como sigue:

```
interseccion ::
  Sector -> Sector -> Sector
interseccion [] _  = []
interseccion (a:as) [] = []
interseccion ((a1,a2):as) ((b1,b2):bs)
  | a2 <= b1 =
      interseccion as ((b1,b2):bs)
  | b2 <= a1 =
      interseccion ((a1,a2):as) bs
  | a2 >= b2 =
      (max a1 b1, b2):
      (interseccion ((b2,a2):as) bs)
  | a2 <= b2 =
      (max a1 b1, a2):
      (interseccion as ((a2,b2):bs))
```

Si pasamos a considerar los cálculos geométricos de la sección anterior, casi todos tienen que ver con obtener ángulos a partir de puntos del plano y las distintas distancias entre ellos. Las siguientes funciones serán básicas:

```
hipot :: Float -> Float -> Float
-- "hipot a b" calcula la hipotenusa de
-- un triángulo rectángulo de catetos
-- "a" y "b"
hipot a b = sqrt (a**2 + b**2)

dist :: Posicion -> Posicion -> Float
-- "dist p1 p2" calcula la distancia
-- euclídea entre los puntos "p1" y
-- "p2" del plano
dist (px, py) (qx, qy) =
  hipot (px - qx) (py - qy)

alcanzable :: Arbol -> Arbol -> Bool
-- "alcanzable a b" es cierto si y sólo
-- si "a" pudiera impactar en "b" al
-- ser talado
alcanzable (a_c, a_r, a_h)
           (b_c, b_r, b_h) =
```

```
  (dist a_c b_c) <
  (b_r + (hipot a_r a_h))
```

Para poder calcular el ángulo que forma con la horizontal la línea que une dos puntos cualesquiera, recurriremos a la función `anguloPos` que obtiene el ángulo de un vector del plano:

```
anguloPos :: Posicion -> Angulo
— precondición: (x, y) =/= (0,0)
anguloPos (x, y)
  | x = 0 && y > 0  = pi/2
  | x = 0 && y < 0  = 3*pi/2
  | x > 0 && y >= 0 = atan (y/x)
  | x < 0 && y > 0  = pi - atan (-y/x)
  | x < 0 && y < 0  = atan (y/x) + pi
  | x > 0 && y < 0  = 2*pi - atan (-y/x)
```

Nos queda tratar los diferentes casos de impacto entre árboles y de un árbol contra el muro. De lo primero se encarga la función `sectorLibre`:

```
sectorLibre :: Arbol -> Arbol -> Sector
-- "sectorLibre a b" devuelve el
-- intervalo de ángulos en que podemos
-- talar "a" sin impactar en "b"
```

Recordemos que se podían dar tres casos: ausencia de impacto, impacto tangencial" e impacto "secante":

```
sectorLibre (a_c, a_r, a_h)
            (b_c, b_r, b_h)
  | a_h <=
    sqrt (((dist a_c b_c) - b_r)**2 -
          a_r**2) =
      [ (0, 2*pi)] —ausencia de impacto
  | a_h >=
    sqrt ((dist a_c b_c) -
          (a_r + b_r)**2) =
      ... —impacto tangencial
  | otherwise =
      ... —impacto secante
```

Para el caso del impacto secante nos podemos apoyar en una función para la resolución de ecuaciones cuadráticas con dos soluciones reales, lo cual viene asegurado en este caso por la existencia de dos puntos de corte entre las circunferencias secantes:

```
cuadratica ::
  Float -> Float -> Float ->
  (Float, Float)
— "cuadratica a b c" resuelve la
— ecuación ax**2 + bx + c = 0
cuadratica a b c =
  ((-b + r)/2*a*c, (-b - r)/2*a*c)
  where r = sqrt (b*b - 4*a*c)
```

Completar el caso secante se deja como ejercicio, a nosotros nos ha quedado una expresión bastante extensa pero quizá el lector pueda simplificarla.

Para el caso del impacto tangencial, el razonamiento geométrico nos conduce al siguiente código:

```
sectorLibre (a_c, a_r, a_h)
            (b_c, b_r, b_h)
  ...
  | a_h >=
    sqrt ((dist a_c b_c) -
          (a_r + b_r)**2) =
      if aa > a && aa + a <= 2 * pi
      then [ (0,aa - a),
             (aa + a, 2 * pi)]
      else if   aa <= a
           then [ (aa + a,
                   2*pi + aa - a)]
           else [ (a - (2 * pi - aa),
                   aa - a)]
      where
        h = sqrt ((dist a_c b_c)**2
                  - (a_r + b_r)**2)
        a = atan ((a_r + b_r) / h)
        aa = anguloPos
               (fst b_c - fst a_c,
                snd b_c - snd a_c)
  ...
```

El tratamiento del posible impacto de los árboles con los muros del bosque se realiza de manera análoga. La función principal es `noMuro`:

```
noMuro ::
  Float -> Float -> Float -> Float ->
  Arbol -> Sector
-- "noMuro minX minY maxX maxY a"
-- devuelve el intervalo de ángulos en
-- que podemos talar "a" sin impactar
-- en el muro cuyas coordenadas indican
-- "minX", "minY", "maxX" y "maxY"
noMuro minX minY maxX maxY (c, r, h) =
  foldr interseccion
        [ (0,2*pi)]
        [ noMuroIzdo minX (c, r, h),
          noMuroAbajo minY (c, r, h),
          noMuroDcho maxX (c, r, h),
          noMuroArriba maxY (c, r, h)]
```

A modo de ejemplo, la codificación de `noMuroIzdo` sería:

```
noMuroIzdo :: Float -> Arbol -> Sector
noMuroIzdo minX (c,r,h) =
  if   ((fst c)-minX) >= d
  then [ (0,2*pi)]
  else [ (0,pi-a-aa),(pi+a+aa,2*pi)]
  where d = hipot r h
        a = acos (((fst c)-minX)/d)
        aa = asin (r/d)
```

Puede observarse en la codificación de `noMuro` cómo la intersección de un conjunto de rangos se reduce a un `foldr` usando el intervalo total $[0,2\pi)$ como elemento neutro. Con todo esto, la realización de `sePuedeTalar` resulta trivial.

5. Para finalizar...

Resumiendo, se trata de un problema más engorroso que complicado, pero lo mismo se puede decir de una gran parte de los problemas prácticos que tenemos que resolver día a día. En ese sentido, tiene un «valor» añadido del que carecen la mayoría de los problemas típicos de los concursos de programación. Es en estas situaciones donde la experiencia en el uso adecuado de un lenguaje de programación puede ser determinante. Un comentario sobre una de las técnicas que hemos usado: operar con intervalos no es exclusivo de problemas geométricos, y algoritmos similares aparecen frecuentemente en problemas de planificación -- aquí los intervalos son temporales. Citamos un par de posibilidades que quedan abiertas para su exploración por los lectores. En primer lugar, hemos realizado todos los cálculos usando coordenadas cartesianas. El uso de una representación alternativa mediante números complejos parece prometedora.

La otra cuestión es que nuestro planteamiento de la solución como un algoritmo voraz no presta atención al orden en que seleccionar los árboles como candidatos para ser talados. Todo parece indicar que, a medida que el número de árboles crece, este indeterminismo podría conducir a incrementar el orden de complejidad de la solución. La existencia de heurísticas que permitan mejorar ese comportamiento parece un problema no trivial.

Notas

[1]En Haskell los campos de un registro se comportan como funciones sobre valores.
[2]Existe una solución dual en la que se calcula, para cada par de árboles, el sector de impacto y se trata de que la unión de tales sectores sea distinta de $[0,2\pi)$.