Originalmente publicado en los números 156 (enunciado) y 157 (solución) de la revista Novática
(http://www.ati.es/novatica/)

Problema B: ¡Queso!

M. Carro
mcarro@fi.upm.es

P. Sánchez
psanchez@skyrealms.org

J. Mariño
jmarino@fi.upm.es

A. Herranz
aherranz@fi.upm.es

1 Descripción del problema

Érase un vez una polilla del queso llamada Amelia Chinche Masticadora que vivía en un enorme trozo de queso. Amelia debería estar tremendamente feliz puesto que estaba rodeada de delicioso queso, más del que nunca podría llegar a comerse. Sin embargo, sentía que en su vida faltaba algo.

Una mañana, cuando estaba soñando con queso, un ruido que nunca antes había escuchado. Pero inmediatamente supo de qué se trataba - el sonido de una polilla macho, ¡royendo en el mismo trozo de queso!. (Averiguar el género de una polilla del queso sólo por el sonido de su roído no es sencillo, pero todas las polillas del queso pueden hacerlo, debido a que sus padres también podían).

Nada podía parar a Amelia ahora. Tenía que llegar hasta esa polilla lo antes posible, y para eso tenía que encontrar el camino más rápido. Amelia era capaz de roer un milímetro de queso en tan sólo diez segundos. Pero resulta que el camino recto no era tambien el más rápido. El queso en el que vivía Amelia estaba lleno de agujeros. Estos agujeros, que no eran mas que burbujas de aire atrapadas en el queso, solían tener forma esférica. Pero en ocasiones, esos agujeros esféricos se solapaban, creando agujeros de muy diversas formas. Amelia era capaz de atravesar uno de esos agujeros en un tiempo prácticamente cero, ya que podía volar de un extremo al otro instantáneamente. Por ello podría ser recomendable atravesar agujeros para llegar hasta la otra polilla rápidamente.

Para este problema debes escribir un programa que, dadas las posiciones de ambas polillas y la de los agujeros en el queso, calcule el tiempo mínimo que necesitaría Amelia para llegar hasta la otra polilla. Para este problema se debe suponer que el queso es infinitamente largo. Por tanto Amelia no necesita abandonar el queso para llegar hasta la otra polilla (en especial porque los depredadores de polillas del queso la podrían devorar). Tambien se supone que la otra polilla ha percibido la llegada de Amelia y no se mueve mientras ella se encuentra en camino.

Descripción de la entrada

El fichero de entrada contiene la descripción de múltiples casos. Cada caso comienza con una linea que contiene un único entero n ($0 \leq n \leq 100$), que es el número de agujeros en el queso. Le siguen n lineas con cuatro enteros cada una x_i, y_i, z_i, r_i. Estos describen el centro (x_i, y_i, z_i) y el radio r_i ($ri > 0$) de los agujeros. Todos estos valores, y los siguientes, vienen dados en milímetros.

La descripción concluye con dos lineas con tres enteros cada una. La primera contiene los valores x_a, y_a, z_a, que indica la posición de Amelia en el queso. La segunda linea contiene x_o, y_o, z_o, la posición de la otra polilla.

El fichero de entrada termina con una linea con el número -1.

Descripción de la salida

Para cada caso de entrada, imprime una linea de salida, con el formato del ejemplo. Primero imprime el número del caso, comenzando en 1. A continuación el tiempo mínimo, en segundos, que necesita Amelia para llegar hasta la otra polilla, redondeado al entero más cercano. La entrada será tal que el redondeo no sea ambiguo.

Ejemplo de entrada

```
1
20 20 20 1
0 0 0
0 0 10
1
5 0 0 4
0 0 0
10 0 0
-1
```

Salida para el ejemplo de entrada

```
Cheese 1: Travel time = 100 sec
Cheese 2: Travel time = 20 sec
```

2 Recorrido en un queso de *Gruyère*

Recordemos que en el problema **B**, Amelia, una habitante de un queso con burbujas (estilo *Gruyère*) infinito, intenta averiguar el camino más corto en tiempo hasta otra polilla macho. Amelia conoce su posición y la de la polilla macho, y se desplaza royendo el queso a una velocidad constante, excepto al encontrar una burbuja, en que puede volar instantáneamente a cualquier parte de la misma. Las burbujas son esféricas y pueden solaparse.

La característica más importante para comprender el problema es la cualidad de *teletransporte* de las burbujas: cual protagonista de *Star Trek*, Amelia, tras alcanzar una burbuja, tiene asegurado un viaje en un tiempo despreciable a cualquier punto dentro del *radio de acción* de la misma. El lector verá en este escenario el de muchas novelas de ciencia ficción, que necesitan transportes a velocidades superlumínicas para justificar la extensión de un imperio en el universo.

Este teletransporte hace que sea difícil utilizar de manera sencilla la posición geométrica de las burbujas y las polillas para hallar el camino más rápido entre estas últimas. Las burbujas introducen discontinuidades en el viaje de la polilla que hacen que el camino más rápido no sea siempre el más corto en distancia. Por ejemplo, la línea recta no es, en general, el mejor camino entre dos polillas: es posible que se pueda utilizar una burbuja apartada de dicha línea como auxiliar para abreviar el viaje prenupcial.

3 De burbujas a grafos

El viaje de la ardiente polilla consiste, en general, en saltar de burbuja en burbuja utilizando el camino más rápido (y, ahora sí, el más corto) entre cada par de ellas. Por tanto el trabajo de la polilla consistiría en elegir cuidadosamente qué burbujas son las adecuadas para llegar más rápidamente, aunque a nosotros sólo se nos pide el tiempo invertido en ese recorrido. En adelante llamaremos óptimo a un recorrido que minimice el tiempo empleado.

El camino *a saltos* entre las burbujas permite modelar el queso como un grafo y buscar el camino óptimo entre polillas como aquel de menor coste asociado en dicho grafo. Asignando un nodo a cada burbuja, el arco que une dos nodos b_1 y b_2 tendría asociado un coste $p(b_1, b_2)$ dado por

$$p(b_1, b_2) = \max(0, dist(c(b_1), c(b_2)) - (r(b_1) + r(b_2)))$$

donde $c(b)$ es el centro de la burbuja b, $r(b)$ es el radio de la burbuja b y $dist(a,b)$ es la distancia entre los puntos a y b. Ambas polillas pueden incluirse en la modelización de modo homogéneo suponiendo que son burbujas de radio cero, a las que les corresponden dos nodos entre los que nos interesa el camino de menor coste. Los cabos que quedan por atar son la representación del grafo, su creación, y el algoritmo para hallar el coste del camino óptimo entre dos nodos.

Asumiendo una implementación en C, los datos correspondientes a la posición de un punto y las burbujas del queso pueden representarse con los siguientes tipos de datos:

```
typedef struct {           /* Posición en el espacio 3D         */
  int x, y, z;
} TPosicion;

typedef struct {           /* Esfera en el espacio 3D           */
  TPosicion centro;
  int radio;
} TBola;
```

Para la representación del grafo se ha optado por una matriz de adyacencia, ya que se conoce de antemano el número de nodos.

```
#define MAX_ESF 102        /* 100 posibles agujeros y 2 insectos*/

typedef struct {           /* Grafo (como matriz de adyacencia) */
  int num_nodos;
  double matriz[MAX_ESF][MAX_ESF];
} TGrafo;
```

El grafo de conexión entre las burbujas no está explícito en la entrada; de hecho hay múltiples posibles grafos que son válidos para resolver este problema. Intuitivamente, y de acuerdo con el propósito del grafo, un nodo *burbuja* sólo necesitaría estar conectado a los nodos más próximos (más concretamente, no tiene por qué estar conectado a los nodos que se pueden alcanzar más rápidamente por medio de otro nodo auxiliar). Pero en realidad no es necesario hacer este filtrado: cualquier recorrido de coste mínimo ha de suprimir, de modo automático, el paso por nodos superfluos. Por tanto es suficiente, y más sencillo, construir un grafo completo en el cual todos los nodos están interconectados entre sí:

```
void calcula_grafo (TGrafo *queso, TBola agujero[MAX_ESFERAS])
{
  int i, j;

  for (i = 0; i < queso->num_nodos; i ++)
    for (j = i; j < queso->num_nodos; j ++)
      queso->matriz[i][j] =
        queso->matriz[j][i] =
          distancia_queso(agujero[i], agujero[j]);
}
```

Las aristas del grafo son evidentemente bidireccionales, y de ahí la simetría de la matriz; deliberadamente no se ha implementado una matriz simétrica con más economía de memoria. Apuntemos que las decisiones de utilizar una matriz completa y no eliminar arcos innecesarios están totalmente justificadas en el entorno de un concurso donde es el tiempo del programador lo que más apremia. Hay que resaltar, sin embargo, que utilizar una matriz simétrica llevaría a disminuir el uso de la memoria prácticamente a la mitad (de N^2 a $\frac{N(N+1)}{2}$).

La distancia entre dos nodos es una implementación directa de la fórmula que ya apareció antes:

```
double distancia_queso (TBola bola1, TBola bola2)
{
  return max(0, (distancia_euclidea(bola1.centro, bola2.centro) -
                 (bola1.radio + bola2.radio)));
}
```

4 Distancia mínima entre polillas

Sólo queda concretar cómo averiguar el camino más corto entre las dos polillas, de las que sabremos sus índices (como nodos) dentro de la matriz de adyacencia. Un recorrido en anchura es apropiado para muchos problemas de recorrido mínimo; sin embargo, no es directamente aplicable a este caso en que el coste de viajar entre nodos no es unitario. Entre los métodos "de libro" para encontrar recorridos de coste mínimo en un grafo (asumiendo costes positivos) podemos citar los conocidos algoritmos de *Floyd-Warshall* y *Dijkstra*. El primero hallaría los recorridos mínimos entre todos los nodos, lo que excede el propósito del problema (sólo lo necesitamos entre dos de ellos). El segundo calcula el coste del recorrido mínimo entre un nodo y todos los demás del grafo, que también es más de lo exigido en el problema, pero es suficientemente fácil de implementar como para ser utilizado en un concurso de programación. Es, de hecho, uno de los algoritmos que constituyen parte de la munición imprescindible en este tipo de concursos. No nos detendremos aquí en su explicación, pues es fácil de encontrar; no obstante, volveremos más adelante sobre la cuestión del trabajo extra.

El código que sigue implementa una versión sencilla del algoritmo de Dijkstra con varias decisiones motivadas por características particulares de este problema, sacrificándose una mejor complejidad en aras de la claridad y la concisión y de poder ofrecer código autocontenido. Debemos hacer notar que en un caso más general el último bucle del procedimiento `dijkstra()` sólo necesitaría recorrer los adyacentes al nodo `sig`. Sin embargo la construcción del grafo conecta cada nodo con todos los demás y por tanto el bucle recorre todos los nodos del grafo, con lo que no se ganaría nada con una implementación basada en, por ejemplo, listas de adyacencia.

```
/*************************************************************/
/* Busca el nodo del grafo que esté a una distancia menor, y    */
/* que no haya sido previamente catalogada como definitiva.     */
/*************************************************************/

int menor_dist (TGrafo queso,              /* Grafo del problema */
                double dist[MAX_ESF],       /* Lista costes      */
                int    definitivo[MAX_ESF]) /* ¿Es definitiva?  */
{
  int i;
  int index = -1;               /* El índice del elemento menor */

  for (i = 0; i < queso.num_nodos; i ++) {
    if (! definitivo[i] &&
        ((dist[index] > dist[i]) || (index == -1))) {
      index = i;
    }
  }

  return index;
}

/*************************************************************/
/* Cálculo de costes mínimos (algoritmo de Dijkstra)            */
```

```
/*************************************************************/

void dijkstra (TGrafo queso,              /* Grafo del problema */
              int    origen,              /* Nodo inicial       */
              double dist[MAX_ESF])       /* Costes mínimos     */
{
  int i, j;
  int sig;

  int definitivo[MAX_ESF];    /* Si el coste es definitivo      */

  /* Inicialmente el coste es el directo desde el origen,       */
  /* pero no son definitivos                                    */
  for (i = 0; i < queso.num_nodos; i ++) {
    dist[i] = queso.matriz[origen][i];
    definitivo[i] = 0;
  }
  definitivo[origen] = 1; /* El coste de Amelia es definitivo   */

  /* Cálculo de coste para los demás nodos (para este           */
  /* problema se podría parar cuando el coste al otro           */
  /* insecto se hiciese definitivo).                            */

  for (i = 1; i < queso.num_nodos; i ++) {
    sig = menor_dist(queso, dist, definitivo);
    definitivo[sig] = 1;

    /* Se recalculan el resto de costes                         */
    for (j = 0; j < queso.num_nodos; j ++)
      if (dist[sig] + queso.matriz[sig][j] < dist[j])
        dist[j] = dist[sig] + queso.matriz[sig][j];
    }
}
```

La complejidad del código anterior es $O(|N|^2)$, donde $|N|$ es el número de nodos. Dado que esta complejidad no depende del número de arcos, el no haberlos eliminado en la construcción del grafo no influye negativamente en la complejidad total del proceso, si bien en otras implementaciones del algoritmo la presencia de arcos innecesarios hubiera incrementado el tiempo de ejecución. Utilizando estructuras de datos más complejas se consigue ajustar la complejidad a $O(|A| + |N| \log |N|)$, donde $|A|$ es el número de aristas. En el caso particular del problema, y por la construcción del grafo, $|A| = \frac{|N|(|N|-1)}{2}$ y no se gana nada con una implementación más sofisticada.

5 Una codificación alternativa en *Haskell*

El lenguaje Haskell [HJea92] está considerado actualmente el lenguaje estándar de programación funcional. Una traducción esencialmente directa del código anterior va a dar lugar a un código sorprendentemente conciso y nos va a servir para mostrar algunas de las características representativas de este lenguaje.

En primer lugar, el programa Haskell que vamos a desarrollar va a ser *genérico* en el sentido de proporcionar una solución al algoritmo de Dijkstra independiente de la función de coste elegida. Esto es posible gracias a las características *de orden superior* de la programación funcional, que permiten, entre otras cosas, pasar funciones como argumentos de otras funciones.[1]

El tipo de la función `dijkstra` será, pues, algo como:

```
dijkstra :: (p -> p -> c) -> p -> [p] -> [(p,c)]
```

Esto significa que una llamada de la forma

```
dijkstra fc orig puntos
```

calcula los costes de viajar desde `orig` a *todos* los puntos de la lista `puntos` usando la función de coste `fc` y devuelve el resultado como una lista de pares `(punto, coste)`.

La idea fundamental del algoritmo de Dijkstra es la incrementalidad del proceso: para calcular los costes de ir desde $orig$ a un conjunto de puntos $p_1, \ldots, p_n, p_{n+1}$ se calculan primero las distancias suponiendo que sólo existen los puntos $p_1, \ldots, p_n$ y los costes se modifican de acuerdo con la presencia del nuevo punto p_{n+1}.

Esto requiere, en primer lugar, ver si el coste desde el origen a uno de los puntos ya examinados mejora pasando por el nuevo punto, lo cual haremos por medio de la función `mejorar`:

```
mejorar fc orig p (q, c) =
  (q, min c ((fc orig p)+(fc p q)))
```

Ahora resta calcular la mejor forma de llegar desde el origen al nuevo punto, lo cual puede ser de forma directa o a través de alguno de los ya examinados. De esto se encarga la función `mejorcoste`:

```
mejorcoste fc orig p pds =
  minimum ((fc orig p):(map (\(q, c) -> (fc p q) + c) pds))
```

Esto necesita un poco más de explicación. La función `minimum` simplemente calcula el menor elemento de una lista. La expresión

```
(fc orig p)
```

es el coste de ir en línea recta (olvidando el resto de burbujas). La expresión

```
\(q, c) -> (fc p q) + c
```

es una λ-*abstracción*, es decir, una abstracción funcional o función anónima[2] que representa la función que *para todo par* (q, c) *devuelve* $(fc\ p\ q) + c$. Resta aplicar dicha función a todos los puntos, lo cual se hace mediante el *iterador de orden superior* `map`, el cual verifica

$$map\ f\ [x_1, x_2, \ldots, x_n] = [f\ x_1, f\ x_2, \ldots, f\ x_n]$$

Como puede verse, una línea de código en Haskell puede dar mucho de sí.

Finalmente, para ensamblar la estructura incremental de `dijkstra` recurriremos a otro iterador fundamental, `foldr`, que realiza un proceso recursivo de una lista. Su ecuación característica es:

$$foldr\ f\ b\ [x_1, x_2, \ldots, x_n] = f\ x_1\ (f\ x_2\ (\ldots (f\ x_n\ b)))$$

lo cual nos permite definir `dijkstra` como:

```
dijkstra fc orig =
  foldr (\p pds -> (p, mejorcoste fc orig p pds):
                   (map (mejorar fc orig p) pds))
        []
```

Sólo nos queda definir la noción de coste entre burbujas

[1] En C conseguimos un efecto similar pasando punteros a función.

[2] Esto ya no es tan fácil de simular en C.

```
gruyere ((a,b,c),r1) ((x,y,z),r2) =
  max 0 (sqrt ((a-x)*(a-x) + (b-y)*(b-y) + (c-z)*(c-z)) - r1 - r2)
```

y definir la función `polilla` que resuelve el problema dadas las posiciones de las dos polillas y la lista de burbujas:

```
polilla p1 p2 burbujas =
  dijkstra gruyere (p1,0) ((p2,0):burbujas)
```

6 Complejidad y caminos explícitos

¿Es posible hallar el camino de coste mínimo entre dos nodos fijados de antemano con un algoritmo computacionalmente más económico que el de Dijkstra? Aparentemente no: no se conoce un método que, en ausencia de más información, determine **únicamente** el camino más corto entre dos nodos de un grafo con menos complejidad que la necesaria para hallarlo entre un nodo dado y todos los demás [CLR90].

Podría intentarse reducir la complejidad del cálculo del coste del recorrido óptimo utilizando información acerca de la posición espacial de la burbuja a la que corresponde cada nodo y la de la polilla destino. La idea sería emplear estos datos para implementar una heurística que guiase una búsqueda con un algoritmo estilo *A** [RN95], por ejemplo. Esto no resulta sencillo *a priori*: la función de estimación requerida debe ser siempre inferior o igual al coste real—pero más significativa que 0 o una constante, que no aportan ningún beneficio.[3] La idea inmediata de utilizar la distancia al destino en línea recta no conduce necesariamente al resultado óptimo, pues en general no es una cota inferior del coste mínimo.

El algoritmo de Dijkstra por sí no nos proporciona ninguna pista acerca del camino a seguir para obtener un coste mínimo, lo que sin duda alguna agradecería Amelia. Es posible modificarlo para registrar este camino; dejamos también al lector realizar esa adaptación.

Conectado con el registro del camino y con una búsqueda informada, se podría instruir a la polilla para realizar un salto a otra burbuja tan pronto como se sabe cuál es la correcta. Este comportamiento es muy conveniente si existe una Amanda rival en el mismo queso que compite con Amelia, y si el tiempo de cálculo del camino no es trivial. Dejamos el diseño de esta variante al lector interesado.

Referencias

[CLR90] Thomas H. Cormen, Charles E. Leiserson y Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[HJea92] P. Hudak, S. L. Peyton Jones y P. L. Wadler et al. Report on the Functional Programming Language Haskell: Version 1.2. *ACM SIGPLAN Notices*, 27(5), Mayo 1992.

[RN95] S. Russell y P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[3] El comportamiento del algoritmo A* en este caso sería básicamente el de una búsqueda en anchura.