novática
UPGRADE Revista de la Asociación de Técnicos de Informática

Novática, revista fundada en 1975 y decana de la prensa informática española, 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, y es miembro fundador de **UPENET** (*UPGRADE European NETwork*)

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

ATI es miembro fundador de **CEPIS** (*Council of European Professional Informatics Societies*) y es representante de España en **IFIP** (*International Federation for Information Processing*); tiene un acuerdo de colaboración con **ACM** (*Association for Computing Machinery*), así como acuerdos de vinculación o colaboración con **AdaSpain**, **AI2** y **ASTIC**.

Coordinación Editorial, Redacción Central y Redacción ATI Madrid
Padilla 66, 3º, dcha., 28006 Madrid
Tlfn.914029391; fax.913093685 <novatica@ati.es>
Composición, Edición y Redacción ATI Valencia
Av. del Reino de Valencia 23, 46005 Valencia
Tlfn./fax 963330392 <secreval@ati.es>
Administración y Redacción ATI Cataluña
Via Laietana 41, 1º, 1ª, 08003 Barcelona
Tlfn.934125225; fax.934127713 <secregen@ati.es>
Redacción ATI Andalucía
Isaac Newton, s/n, Ed. Sadiel,
Isla Cartuja 41092 Sevilla, Tlfn./fax 954460779 <secreand@ati.es>
Redacción ATI Aragón
Lagasca 9, 3-B, 50006 Zaragoza.
Tlfn./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)
Tlfn.986581413; fax.986580162 <secregal@ati.es>
Suscripción y Ventas
<http://www.ati.es/novatica/interes.html>, o en ATI Cataluña o ATI Madrid
Publicidad
Padilla 66, 3º, dcha., 28006 Madrid
Tlnf.914029391; fax.913093685 <novatica.publicidad@ati.es>
Imprenta
9 Impressió S.A., Juan de Austria 66, 08005 Barcelona.
Depósito legal: B 15.154-1975 -- ISSN: 0211-2124; CODEN NOVAEC
Portada: Antonio Crespo Foix / © ATI 2004
Diseño: Fernando Agresta / © ATI 2004

Nº 170, julio-agosto 2004, año XXX

sumario

Monografía del próximo número: "Tecnologías de Proceso Software"

Por otra ruta, por favor

Ángel Herranz Nieva, Julio Mariño y Carballo
Facultad de Informática, Universidad Politécnica de Madrid

<{aherranz, julio.marino}@fi.upm.es>

El enunciado de este problema apareció en el número 169 de ***Novática*** (mayo-junio 2004, p. 73). Es el problema E de los planteados en el I Concurso Universitario de la Comunidad Autónoma de Madrid (CUPCAM 2003)

1. Resumen del problema

El problema presentado en el número anterior pedía encontrar el segundo camino más corto entre un par de nodos en un grafo no dirigido bajo las condiciones de no tener bucles y de que su distancia no sea mayor que la del camino más corto más un 25%.

2. Solución de concurso

Es habitual en los concursos de programación plantearse la posibilidad de una solución barata de construir aunque su eficiencia sea más bien pobre. Esa primera posibilidad bien podría ser:

1. Generar la lista de todos los caminos *sin ciclos* desde el nodo inicial ordenados de menor a mayor distancia.
2. Filtrar la lista y quedarse con aquellos caminos que terminen en el nodo final con una distancia menor que la requerida.
3. Quedarse con el segundo elemento de la lista.

El algoritmo anterior parece inabordable, en particular la generación de todos los caminos. Sin embargo hay un par de propiedades que pueden aprovecharse:

- La lista de todos los caminos sin ciclos desde un nodo dado es finita.
- Cada vez que se construye un nuevo camino a partir de uno anterior éste tiene una longitud mayor por lo que la ordenación se puede realizar durante la generación.

Si a estas dos propiedades añadimos el hecho de utilizar un lenguaje con evaluación perezosa como Haskell 98 ([33]) tendremos una eficiencia razonable.

3. Representación del grafo

Se ha decidido representar el grafo mediante el conjunto de nodos (identificados por números enteros) y el conjunto de arcos representados como tripletas siendo *u* y *v* dos nodos y la distancia (a la que denominaremos también coste) entre ambos:

```
> import List

> type Node = Int
> type Cost = Int

> data Graph = Graph {
>    nodes :: [Node],
>    arcs :: [(Node,Node,Cost)]
> }
```

Dado un grafo *g*, `nodes` *g* representa el conjunto de nodos del *g* y `arcs` *g* el conjunto de arcos. Se implementa la operación `reachables` que dados un grafo y un nodo devuelve los nodos alcanzables en un paso desde el nodo dado y el coste asociado:

```
> reachables ::
>    Graph -> Node -> [(Node,Cost)]
> reachables graph u =
>    [ (v,d) | (uí,v,d) <- arcs graph,
>              uí == u ] ++
>    [ (v,d) | (v,uí,d) <- arcs graph,
>              uí == u ]
```

`nodes`, `arcs` y `reachables` componen la abstracción de grafos que se utilizará en la implementación de la solución[1]. De esta forma la estructura de datos concreta se puede cambiar si se mantiene el interfaz y la semántica de las operaciones anteriores.

4. Caminos ordenados

Dado un nodo inicial se calculan todos los caminos desde dicho nodo ordenados por el coste total de su recorrido. El camino más corto es el camino vacío de coste 0. Dada una lista de caminos ya ordenada se toma el camino mínimo, se amplía hasta los nodos alcanzables desde el último nodo del camino, se eliminan aquellos caminos con bucles y se mezclan ordenadamente con el resto de caminos para continuar con una nueva iteración.

Veamos una implementación:

```
> allPathsFrom ::
>    Graph -> Node -> [(Cost,[Node])]
> allPathsFrom graph initial =
>    genPaths graph [(0,[initial])]

> genPaths ::
>    Graph -> [(Cost,[Node])]
>    -> [(Cost,[Node])]
> genPaths graph [] = []
> genPaths graph (p@(du,u:us):paths) =
>    let pathsFromFirst =
>          cleanUp
>            [(du + duv, v:u:us)
>             | (v,duv)
>               <- reachables graph u]
>        sortedPathsFromFirst =
>          sortBy comparePaths
>                 pathsFromFirst
>        sortedPaths =
>          mergeBy comparePaths
>                  sortedPathsFromFirst
>                  paths
>    in
>      p : genPaths graph sortedPaths
```

La operación `cleanUp` elimina los caminos con ciclos que se hayan podido introducir mientras que las operaciones `sortBy` y `mergeBy` implementan las operaciones de ordenación y mezcla ordenada parametrizadas con un orden parcial.

5. Selección del segundo camino más corto

El último paso de nuestro algoritmo es el filtrado de la lista ordenada de caminos para quedarnos sólo con aquéllos que terminan en el nodo final.

```
> allPathsFromTo ::
>    Graph -> Node -> Node
>    -> [(Cost,[Node])]
> allPathsFromTo graph initial final =
>    filter (\ (d,u:_) -> u == final)
>           (allPathsFrom graph initial)
```

El resultado de la expresión `allPathsFrom` *g i f* es la lista con *todos* los caminos que parten del nodo *i* y llegan al nodo *f* ordenados de menor a mayor coste. Si la lista tiene al menos dos elementos, la solución es el coste del segundo siempre y cuando no supere el coste del camino más corto en más de un 25%.

```
> answer :: Graph -> IO ()
> answer graph =
>   do ó Reading pair of nodes
>   line <- getLine
>   let [s1,s2] = words line
>   let i = read s1 :: Int
>   let f = read s2 :: Int
>   if i == 0
>     then ó Finishing
>       putStr ìî
>     else
>       let twoPaths =
>             take
>               2
>               (allPathsFromTo
>                     graph
>                     i
>                     f)
>       in
>         if length twoPaths < 2
>         then
>           do ó No snd route
>           putStrLn ìnoî
>           answer graph
>         else
>           do
>           let (d1,_):(d2,_):_
>                 = twoPaths
>           if d2 >
>              d1 + (25 * d1 ëdivë 100)
>             then ó Too long snd route
>                  putStrLn ìnoî
>             else ó d2 is the solution
>                  print d2
>           answer graph
```

Obsérvese que la pereza de Haskell hace que no sea necesario generar todos los caminos en la expresión `take 2 (allPathsFromTo graph i f)` .

6. Programa principal

Presentamos a continuación el código de la función principal para que el lector disponga del programa completo:

```
> main :: IO ()
> main =
>   do line1 <- getLine
>      line2 <- getLine
>      let n = read line1 :: Int
>      let a = read line2 :: Int
>      arcs <- readArcs a
>      let graph =
>            Graph { nodes = [1..n],
>                    arcs = arcs }
>      answer graph
> readArcs ::
>   Int -> IO [(Node,Node,Cost)]
> readArcs n
>   | n == 0 = return []
>   | otherwise =
>       do line <- getLine
>          let [s1,s2,s3] = words line
>          let u = read s1 :: Int
>          let v = read s2 :: Int
>          let duv = read s3 :: Int
>          arcs <- readArcs (n - 1)
>          return ((u,v,duv) : arcs)
```

7. Operaciones auxiliares

Por el camino hemos dejado sin programar algunas operaciones auxiliares como la eliminación de caminos con ciclos, comparaciones entre caminos y mezclas ordenadas de listas de caminos.

```
> cleanUp ::
>   [(Cost,[Node])] -> [(Cost,[Node])]
> cleanUp = filter loopLessPath

> loopLessPath ::
>   (Cost,[Node]) -> Bool
> loopLessPath (d,path) =
>   nub path == path

> comparePaths ::
>   (Cost,[Node]) -> (Cost,[Node])
>   -> Ordering
> comparePaths (d1,_) (d2,_) =
>   compare d1 d2

> mergeBy ::
>   (a -> a -> Ordering) -> [a] -> [a]
>   -> [a]
> mergeBy cmp [] ys = ys
> mergeBy cmp xs [] = xs
> mergeBy cmp (x:xs) (y:ys)
>   | cmp x y == LT =
>       x : mergeBy cmp xs (y:ys)
>   | cmp x y == EQ =
>       x : mergeBy cmp xs (y:ys)
>   | cmp x y == GT =
>       y : mergeBy cmp (x:xs) ys
```

3. Una solución más inteligente

O, al menos, más avispada. Una estrategia que suele dar muy buenos resultados en este tipo de concursos es la de transformar el problema original en otro bien conocido y para el que se dispone de una solución fiable que podemos codificar rápidamente.

En el caso que nos ocupa, parece que el problema con un parentesco más evidente es el de calcular el camino más corto entre dos nodos de un grafo, existiendo un buen número de algoritmos conocidos (Dijkstra, por ejemplo) y suficientemente sencillos como para programar casi de memoria. Ahora bien: ¿cómo engañarle para que resuelva un problema distinto a aquél para el que fue diseñado?

Vamos a demostrar que el camino que nos están pidiendo – en caso de existir – es siempre el camino más corto en un grafo que sólo difiere del original en un arco. Sea G el grafo del problema, P el camino más corto entre los nodos A y B y Q el segundo camino más corto entre dichos nodos. Ya que ni P ni Q tienen ciclos, debe existir al menos un arco X que está en P pero no en Q[2].

Si llamamos H al grafo resultante de eliminar el arco X de G, es relativamente sencillo demostrar que Q es el camino más corto entre A y B en H. El algoritmo queda claro ahora:

1. Calcular el camino más corto entre A y B.
2. Para cada arco de dicho camino, construir el grafo resultante de eliminar dicho arco en G.

3. Para cada grafo obtenido en el paso anterior, calcular el camino mínimo entre *A* y *B*.
4. Seleccionar el más corto de dichos caminos.

Para ser una solución de *segunda mano*, el resultado no parece ser malo en términos de eficiencia, ya que en el peor de los casos su complejidad sólo difiere de la de Dijkstra en el número de arcos del grafo.

4. Variaciones

El ejemplo con el que motivábamos el problema (calcular una ruta subóptima entre dos puntos de un mapa de carreteras) es, en general, inabordable siguiendo las técnicas que hemos usado arriba. El número de nodos y arcos en el mapa de, pongamos, España, es tan elevado que cualquier método de búsqueda combinatoria ciega resulta impracticable.

De igual manera que para ir de Sevilla a Málaga nunca probaríamos un camino donde interviniese la A-6, los algoritmos reales para estos casos deben usar algún criterio que les haga descartar combinaciones poco verosímiles. Un típico algoritmo de búsqueda guiada operaría de la siguiente manera para obtener el camino más corto entre *A* y *B*. Obtendría primero los vecinos de *A*:. Crearía entonces una lista de caminos parciales *ordenados de más a menos prometedor*:. El procedimiento se repetiría entonces calculando los vecinos de , ordenando los caminos resultantes, etc. El primer camino en llegar a *B* se toma como mejor solución. Lo interesante de este algoritmo es que escogiendo adecuadamente la noción de más prometedor, podemos estar seguros de que el primer camino en llegar a *B* es realmente el más corto (!).

La clave está en la noción de *función de estimación de coste pesimista*. Supongamos que el criterio para ver que un camino es prometedor es una función que cumple las dos condiciones siguientes:

1. $p(Q)=longitud(Q)$ si Q empieza en A y acaba en B.
2. Si Q empieza en A pero acaba en $X \neq B$, $p(Q)$ es menor que la longitud de ning´n camino de A a B que contiene a Q.

Si el algoritmo anterior ordena los caminos parciales de acuerdo con sus valores de *p*, el primer camino en llegar a *B* será el más corto. La demostración se deja como ejercicio al lector[3] .

Es muy sencillo encontrar una función así para un mapa de carreteras. Si Q es un camino de A a X, basta con definir $p(Q)$ como la suma de la longitud de Q más la distancia aérea de X a B. Es trivial comprobar que las dos condiciones anteriores se cumplen.

Ahora bien, ¿sigue siendo válido el algoritmo para calcular el segundo camino más corto? Si el lector ha demostrado la corrección del algoritmo de búsqueda guiada, no le costará demasiado probar que, en efecto, el siguiente camino que se obtendría tras eliminar el primero es el segundo más corto.

La técnica de búsqueda guiada que hemos descrito aquí (*best-first search*) está ampliamente documentada en textos clásicos de Inteligencia Artificial, como [4].

El problema de calcular los *k* mejores caminos y sus aplicaciones
El desarrollo de la solución nos lleva a preguntarnos por un problema más general que el propuesto: calcular el k-ésimo camino (sin bucles, obviamente) más corto entre dos nodos de un grafo.

El algoritmo 'pillo' de la **sección 3** es aceptable para calcular el segundo camino más corto en un grafo no muy grande, pero degenera exponencialmente si queremos seguir usándolo para el tercero, cuarto mejor, etc.
Por contra, el algoritmo de búsqueda guiada que acabamos de ver sí que va dando los caminos en orden de mejor a peor, pero sólo vale para grafos donde exista alguna propiedad – geométrica o de otro tipo – que permita calcular estimaciones pesimistas.

El problema de calcular, para un grafo cualquiera (posiblemente dirigido), los *k* caminos más cortos entre dos nodos, es tan complejo como ubicuo, lo que ha provocado que se trabaje intensamente en él. El problema ha sido estudiado en profundidad, lógico si se tiene en cuenta que sólo en la década de los 50 se publicaron más de setenta (!) artículos [1].

Hay muchos problemas reales – e.g. la alineación de secuencias genéticas – que admiten ser formulados como problemas de camino más corto. No obstante, al tratarse de datos empíricos con un cierto ruido estadístico, resulta más interesante para un científico considerar no sólo la solución que proporciona el mejor ajuste sino un abanico de unas cuantas combinaciones prometedoras.

El artículo de David Eppstein [2] es una buena introducción reciente al problema, con sus aplicaciones y un buen número de citas de trabajo previo. En su página web se puede encontrar una bibliografía[4] bastante completa de este problema.

En la página de Víctor Jiménez y Andrés Marzal, <http://terra.act.uji.es/REA>, hay algoritmos para resolver el problema, incluyendo implementaciones y punteros útiles.

Referencias

[1] A. W. Brander, Mark C. Sinclair. A comparative study of *k*-shortest path algorithms. In *Proc. 11th UK Performance Engineering Worksh. for Computer and Telecommunications Systems*, September 1995.
[2] David Eppstein. Finding the *k* shortest paths. *SIAM J. Computing*, 28(2):652–673, 1998.
[3] S. Peyton Jones, J. Hughes. *Report on the Programming Language Haskell 98. A Non-strict Purely Functional Language*, February 1999.
[4] P.H. Winston. *Artificial Intelligence*. Addison-Wesley, 1984.

Notas

[1] La función *reachables* se puede precomputar para obtener un mayor grado de eficiencia. Sin embargo, nosotros hemos preferido confiar en la capacidad de tabulación que tienen la mayor parte de las implementaciones de Haskell 98.
[2] Este paso no es inmediato. La demostración se basa en probar que si Q es un superconjunto de P, Q ha de contener al menos un ciclo.
[3] Se procede por reducción al absurdo.
[4] <http://www.ics.uci.edu/~eppstein/bibs/>.