LOS PUENTES DE KÖNIGSBERG

Cuaderno de notas

Índice

1. Introducción

Primero creamos la batería de grafos de ejemplo y definimos en Haskell las propiedades que garantizaban la existencia de un camino euleriano (basándonos en los teoremas correspondientes).

En las dos primeras opciones, antes de aplicar la función que busca los caminos eulerianos, hemos definido una guarda que evalúa si el grafo tiene camino o no. Esto puede reducir la eficiencia (aunque no de forma notable). Una opción alternativa sería aplicar desde el principio la función y definir un test de parada para que, cuando llegue a un punto en el que no hay camino, dar automáticamente la lista vacía.

En la práctica, los *softwares* de cálculo numérico no aplican los teoremas desde el principio, ya que ello disminuye la eficiencia. Ejecutan desde el primer momento la función y cuando lleguen a una contradicción cortan la ejecución.

Podemos observar que si el grafo es euleriano (posee un ciclo euleriano), entonces todos los caminos eulerianos encontrados son iguales.

También observamos que:

- Si el grafo es euleriano, encontraremos un ciclo euleriano independientemente del nodo en el que empecemos.
- Si el grafo no es euleriano pero admite un camino euleriano, este debe comenzar en un nodo de valencia impar.
- Todos los caminos (ciclos) eulerianos encontrados tienen la misma longitud.

2. Búsqueda en espacios de estados

2.1. Primera opción: búsqueda en espacios de estados usando aristas

Ya que nuestro problema es buscar un camino en el que las aristas no estén repetidas, pensamos que la mejor opción era trabajar directamente con ellas.

Se nos presentó una primera dificultad: cómo definir correctamente el sucesor de una arista (ya que hay dos posibles caminos que seguir). Creímos que teníamos una función que funcionaba correctamente.

```
type Estado = [Arista]

-- Siguientes proporciona las aristas adyacentes a una dada. Como el grafo
-- es no dirigido, se tienen en cuenta solo una de ellas.
```

```
siguientes :: GrafoI -> Arista -> [Arista]
siguientes g a@(x,y,z) = reducePSND a (auxi g (x,y,z) ++ auxi g (y,x,z))
  where auxi g (x,y,_) = [(y',z,p) | (y',z,p) <- aristas g, y' == y]
        reducePSND s@(x,y,z) xs = xs \\ [s, voltea s]

sucesores :: GrafoI -> Estado -> [Estado]
sucesores g (x:xs) = [y:x:xs | y <- siguientes g x, prop y (x:xs)]
  where prop y (x:xs) = y `notElem` (x:xs)  &&  voltea y `notElem` (x:xs)
```

Tras definir las funciones auxiliares necesarias para aplicar la búsqueda en espacios de estados, probamos nuestra batería de ejemplos.

Resultados satisfactorios para todos ellos excepto para `g2` y `g9`.

```
ghci> hayCamino g2
True
gchi> head $ caminosEulerianos g2
[(1,2,0),(2,3,0),(3,1,0),(1,4,0),(4,3,0),
(3,6,0),(6,1,0),(6,4,0),(4,5,0),(5,6,0)]
ghci> hayCamino g9
True
ghci> head $ caminosEulerianos g9
[(1,2,0),(2,3,0),(3,4,0),(4,5,0),(5,6,0),(6,1,0),
(1,7,0),(7,6,0),(6,10,0),(10,4,0),(4,9,0),(9,3,0),
(3,8,0),(8,1,0),(8,7,0),(7,10,0),(10,9,0),(9,8,0)]
```

Observamos que, efectivamente, todas las aristas son recorridas una única vez. Sin embargo, para ir de un nodo a otro es necesario (en los casos fallidos anteriores) saltar de un nodo a otro por un camino inexistente.

Es por ello que desechamos esta primera idea.

2.2. Segunda opción: búsqueda en espacios de estados usando nodos

En este segundo intento, los siguientes serán nodos (números enteros) e iremos guardando la lista de aristas recorridas.

Para ello, definiremos de nuevo el tipo de los estados.

```
type Estado = ([Int],[Arista])
```

- En la primera componente, vamos guardando el camino recorrido (de qué nodo a qué nodo nos vamos moviendo).

- En la segunda componente, guardamos las aristas recorridas, a efectos de verificar que ninguna arista se repite y de verificar cuándo un estado es final.

Tendremos que definir una función que me proporcione la arista que une dos nodos. A la hora de implementarla en funciones posteriores, debemos verificar (con la función `aristaEn`) que efectivamente pertenece al grafo dado.

La idea es similar a la anterior. En este caso, vamos añadiendo los nodos de entre los posibles adyacentes. También vamos añadiendo las aristas a la segunda componente del estado, con la condición de que ni ella ni su opuesta estén ya ahí.

```
sucesores :: GrafoI -> Estado -> [Estado]
sucesores g (x:xs,ys) = [(y:x:xs,h:ys) | y <- adyacentes g x,
                                         let h = arista x y,
                                         h `notElem` ys,
                                         aristaOp h `notElem` ys]
```

Resultados satisfactorios para todos nuestros casos de prueba.

2.3. Retoques finales y comprobación de propiedades

Nos damos cuenta de una cosa: ¿para qué verificamos desde el primer momento que podemos encontrar camino? Si ejecutamos la función de búsqueda en espacios de estados en un grafo que no admite ni camino ni ciclo euleriano, directamente nos va a dar como resultado la lista vacía. Por tanto, quitemos la comprobación inicial y así aumentaremos un poco la eficiencia.

Además, ahora podemos establecer propiedades con `QuickCheck` (los teoremas de Euler) y ver que se cumplen.

Hemos quitado las guardas de la función `caminosEulerianos` y todo sigue funcionando bien.

Hemos definido una propiedad que comprueba el cumplimiento de los teoremas de caminos eulerianos. Tenemos que especificar que el grafo es no dirigido (por ser nuestra suposición).

```
prop_caminosEulerianos :: GrafoI -> Property
prop_caminosEulerianos g =  not (dirigido g)  ==>
  if null (caminosEulerianos g) then not (hayCamino g) else hayCamino g
```

Sin embargo, cuando `QuickCheck` llega a grafos completos de orden 6 o superior, se traba.

Faltaría especificar también, en las cláusulas de la propiedad a analizar con `QuickCheck`, que el grafo debe ser conexo.

NOTA: nuestro algoritmo no es aplicable a multigrafos. Si cogiera una de las múltiples aristas, y ya hubiera pasado por la otra, no la capta. Una posible mejora es haciendo que la segunda

componente sea un multiconjunto (con la librería `Data.Map`), en el que aparecieran la arista y el número de veces que ha aparecido.

Hemos conseguido encontrar una función para ver si un grafo es conexo (ver capítulo 3). Ahora lo tenemos todo para demostrar nuestras propiedades de grafos.

Sin embargo, seguimos encontrando el problema en los grafos completos a partir del orden 6. Si ejecutamos la función `caminoEuleriano` a este tipo de grafos, el programa no es capaz de dar un resultado en poco tiempo (llega finalmente a un resultado pero tras varios minutos). El volumen de datos que genera la búsqueda en espacios de estados es muy grande.

Es por ello que `QuickCheck` no es capaz de dar por válida la propiedad cuando llega a estos grafos.

2.4. Implementación de multiconjuntos

En nuestro caso anterior teníamos que el contenedor de aristas era una lista. Sin embargo, según teníamos definida la función `sucesores`, no podíamos usar nuestro algoritmo cuando teníamos multigrafos.

Si definimos el nuevo tipo de los estados como

```
type Estado = ([Int],MultiConj)
type MultiConj = M.Map Arista Int
```

hemos resuelto el problema.

Las funciones principales del TAD de los multiconjuntos pueden consultarse en la web de José Antonio Alonso Jiménez (ver bibliografía del proyecto). Se requiere la librería `Data.Map`.

Las modificaciones que debemos realizar en las funciones de la búsqueda en espacios de estados son sutiles:

- El estado *inicial*, en lugar de tener la lista vacía como contenedor de aristas, tendrá el multiconjunto vacío.
- Un estado será *final* si el cardinal del contenedor de aristas es igual al número de aristas.

2.4.1. La función `sucesores`

Hemos dedicado una sección únicamente a la función `sucesores` ya que las modificaciones necesarias son un poco más complejas. No debemos limitarnos a ver que una arista determinada pertenece o no al contenedor de aristas.

Un multiconjunto de aristas debe considerar (como ya venimos diciendo desde la primera línea) una arista y su opuesta como iguales (por nuestras suposiciones). Por tanto, a la hora

de introducir una arista debemos comprobar la presencia de la arista opuesta para que, en caso afirmativo, la contabilice como la misma.

También hemos definido el multiconjunto de las aristas de un grafo siguiendo las suposiciones del párrafo anterior.

Por último, nos falta definir las propiedades necesarias en la guarda de la lista de comprensión de la función `sucesores`. Para que una arista no se incluya, el número de veces que esa arista está en el multiconjunto tras una supuesta introducción de la misma debe ser superior al número de veces que esa arista está en el multiconjunto de las aristas. Si la arista no está en el contenedor, pues simplemente la añadiremos.

Tampoco podemos usar la función `inserta` (de multiconjuntos) directamente sobre el contenedor, ya que si la usamos no tendremos en cuenta el hecho de que una arista y su opuesta son la misma. Por tanto, debemos definir una función auxiliar que contemple este hecho.

Con todas estas condiciones conseguimos definir la función `sucesores` con éxito.

```
sucesores :: GrafoI -> Estado -> [Estado]
sucesores g (x:xs,ys) = [(y:x:xs, insAr h ys) | y <- adyacentes g x,
                                                let h = arista x y,
                                                prop h ys]
  where prop h ys | x == 0     =  True
                  | otherwise  =  x < nAr h (conjAristas g)
          where x = nAr h ys
```

Tras la ejecución `caminoEuleriano`, obtenemos resultados satisfactorios para los grafos simples que ya teníamos y también para los multigrafos que antes no disponían de solución al problema (por la anterior definición).

2.5. Retos pendientes

¿Qué retos nos quedan pendientes? Principalmente dos:

1. Mejorar la eficiencia de `caminosEulerianos` (mejorando la eficiencia de la búsqueda en espacios de estados).
2. Conseguir aplicar satisfactoriamente `QuickCheck` (para ello, deberemos haber conseguido el primer ítem o, en su defecto, esperar un buen rato a que puedan evaluarse los casos más complejos).

3. Grafos conexos

Hemos conseguido definir dos propiedades para ver si un grafo es conexo.

1. Si un grafo es conexo, entonces siempre hay un camino que une dos nodos tal que no se repita ningún nodo.
2. Usando un algoritmo con matrices de adyacencia.

El algoritmo con matrices de adyacencia se basa en el siguiente teorema:

> *La entrada en la fila i y la columna j de la matriz A^k, denotada por a_{ij}^k, es igual al número de caminos de longitud k con extremos v_i y v_j.*

Los pasos a seguir son:

1. Hallar la matriz de adyacencia del grafo G, denotada por A.
2. Se calcula la matriz $M = \sum_{k=1}^{n-1} A^k$.
3. El grafo es conexo si y sólo si ninguna entrada de M es cero.

Hemos comprobado que ambas definiciones son equivalentes. Sin embargo, había que poner ciertas restricciones:

- El grafo debía ser no dirigido (según nuestra definición de matriz de adyacencia).
- El mínimo número de nodos de un grafo debía ser 3, ya que si era 2 fallaba el segundo algoritmo.

```
prop_conexo :: Grafo Int Int -> Property
prop_conexo g =  prop g  ==>  esConexo1 g == esConexo2 g
where prop g = length (nodos g) > 2  &&  not (dirigido g)
```

```
gchi> quickCheckWith (stdArgs {maxSuccess=10000}) prop_conexo
+++ OK, passed 10000 tests.
```

Para grafos de hasta 3 nodos, es más eficiente la segunda definición. Sin embargo, a partir de este tamaño la eficiencia de esta función disminuye considerablemente. El tiempo de ejecución de la primera definición aumenta (como es esperado), pero se mantiene a niveles muchísimo más bajos que la segunda definición.

4. Otros procedimientos

A continuación explicaremos los otros dos algoritmos descendentes que hemos implementado para resolver el problema que nos traemos entre manos. Ambos necesitan que definamos una *heurística*; es decir, un criterio que sirva para ordenar los estados de mejores (más cercanos a la solución) a peores (más lejos de la solución).

Seguiremos con la implementación de multiconjuntos como contenedor de aristas. Un estado estará más cerca del final en nuestro caso cuanto mayor sea el cardinal del contenedor. Recordamos que un estado será final cuando el cardinal del contenedor sea igual al número de aristas de nuestro grafo.

Por tanto, definiremos una instancia que nos sirva para comparar dos estados usando su heurística:

```
instance Eq Estado
  where est1 == est2 = heur est1 == heur est2
instance Ord Estado
  where est1 <= est2 = heur est1 <= heur est2
```

Nuestro tipo para los estados no nos vale ahora. Si queremos ordenar pares, Haskell ya tiene implementado el criterio. Si intentamos definir otro criterio, recibimos el mensaje de error:

```
Overlapping instances for Eq Estado ...
      arising from the superclasses of an instance declaration
    Matching instances:
      instance (Eq a, Eq b) => Eq (a, b) -- Defined in ‘GHC.Classes’
      instance Eq Estado
    In the instance declaration for ‘Ord Estado’
Compilation failed.
```

Definiremos un nuevo tipo de datos para poder ahora definir correctamente las instancias:

```
data Estado = EST ([Int],MultiConj)
   deriving Show
```

Nuestra heurística será:

```
heur :: Estado -> Int
heur (EST (_,m)) = cardinal m
```

4.1. Búsqueda por *primero el mejor*

Se corresponde con el archivo `primero_mejor_v1.hs`.

Utilizaremos la librería `I1M.BusquedaPrimeroElMejor`. El procedimiento seguido por este algoritmo se basa en el uso de colas de prioridad en lugar de listas. La prioridad de los elementos viene marcada por su heurística.

Al aplicar este algoritmo a nuestro problema observamos que la eficiencia se ve reducida enormemente (incluso en nuestra batería de ejemplos, que son casos bastante manejables). Así que la hemos descartado completamente.

4.2. Búsqueda en escalada

Se corresponde con el archivo `escalada_v1.hs`.

Para ello utilizaremos la librería `I1M.BusquedaEnEscalada`. La búsqueda en escalada (también conocida como *greedy algorithm*, algoritmo "codicioso") es un algoritmo que siempre usa la mejor solución de las presentes, siguiendo una heurística determinada. Como sólo sigue un camino, la eficiencia es bastante alta.

La búsqueda en escalada puede compararse con la visión limitada que tenemos cuando bajamos una colina con niebla. Elegir siempre el camino que localmente va cuesta abajo no garantiza que alcances el pie de la colina. Quizás este sea el mayor inconveniente de este algoritmo [F. Rabhi y G. Lapalme; *Algorithms: A functional programming approach*].

Tras aplicar la búsqueda en escalada a nuestro problema, observamos que todos nuestros casos de prueba funcionan perfectamente. Decidimos por tanto probar nuestra ansiada propiedad, y para nuestra desilusión encuentra fallos. Los grafos encontrados tienen camino, pero nuestro algoritmo no los encuentra. Si le aplicamos la búsqueda en espacios de estados general a estos grafos encontramos satisfactoriamente un camino.

5. Función final

La solución final a nuestro problema consistirá en combinar adecuadamente la búsqueda en espacios de estados con la búsqueda en escalada. Podemos ver una analogía entre nuestra situación y un mapa de carreteras. ¿A cuántas ciudades podemos llegar directamente en autopista? A prácticamente ninguna. Son las carreteras secundarias las que llegan a todos los rincones. Y la forma más rápida de llegar a una ciudad es la mezcla adecuada de autopista y carretera, usando el mayor tiempo posible la primera opción.

Para encontrar la mejor combinación debemos responder a la siguiente pregunta:

¿Cuándo es absolutamente imprescindible usar la búsqueda en espacios de estados?

Las respuestas que encontramos son las siguientes:

- Si `buscaEsc` encuentra una solución, es la que buscamos.
- Si `buscaEsc` no proporciona solución, no tenemos la certeza de que así sea.
- Si `buscaEsc` no proporciona solución pero `hayCamino` muestra que sí existe, entonces `buscaEE` lo encontrará.

Por tanto, la función final es:

```
caminoEuleriano :: GrafoI -> Maybe [Int]
caminoEuleriano g | isNothing x && hayCamino g  =  caminoEE g
                  | otherwise                   =  x
  where x = caminoEsc g
```

Hemos reformulado las propiedades, ya que las que habíamos definido no se correspondían con los teoremas de Euler.

```
-- Condición necesaria y suficiente
prop_caminosEulerianos1 :: GrafoI -> Property
prop_caminosEulerianos1 g =  prop g  ==>
  if not (hayCamino g) then isNothing ce else isJust ce
   where prop g = not (dirigido g)  &&  esConexo g
         ce     = caminoEuleriano g

-- Condición necesaria
prop_caminosEulerianos2 :: GrafoI -> Property
prop_caminosEulerianos2 g = prop g  ==>  hayCamino g
   where ce     = caminoEuleriano g
         prop g = not (dirigido g)  &&  isJust ce
```

Y por fin conseguimos lo que tanto habíamos deseado ...

```
ghci> quickCheck prop_caminosEulerianos1
+++ OK, passed 100 tests.
ghci> quickCheck prop_caminosEulerianos2
+++ OK, passed 100 tests.
```

Con esto podemos dar por finalizada la resolución del problema de los puentes de Königsberg. Sólo nos queda redactar el informe y darle un poco de estilo. Ha sido un proyecto muy *challenging*, ya que desde el primer momento nos ha exigido pensar nuevas vías y desarrollar los mejores métodos para resolverlo. Creo que este esfuerzo ha merecido muchísimo la pena, ya que nos ha ayudado a desarrollar nuestro espíritu investigador. Espero que sea el primero de muchos.