LOS PUENTES DE KÖNIGSBERG

Estudio y resolución con Haskell

Miguel Ibáñez García

Universidad de Sevilla

Índice

1. Introducción

1.1. Euler y la Teoría de Grafos

> *La rama de la geometría que se ocupa de las magnitudes ha sido estudiada concienzudamente en el pasado, pero hay otra rama casi desconocida hasta el momento; fue Leibniz el primero que habló de ella, denominándola* geometría de la posición. *Dicha disciplina estudia aquellas relaciones que dependen únicamente de la posición, e investiga las propiedades de las mismas. No tiene en cuenta las magnitudes ni trata de calcular cantidades. Pero todavía no se han definido de forma satisfactoria los problemas que incumben a esta geometría de la posición ni el método a utilizar para resolverlos.*

El fragmento anterior pertenece a *Solutio problematis ad geometriam situs pertinentis* [1], obra del célebre matemático Leonhard Euler (1707-1783). En él puede verse con claridad que es plenamente consciente de que había problemas que la geometría tradicional no era capaz de resolver.

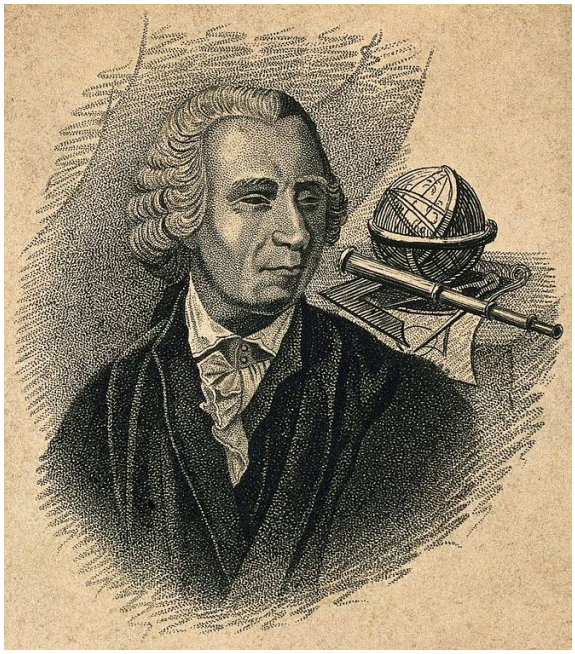


Figura 1: Leonhard Euler [11]

Esa "rama" de la que Euler nos habla es la Topología, probablemente la más joven de las ramas clásicas de las Matemáticas. En contraste con el Álgebra, la Geometría y la Teoría de los números, cuyas genealogías datan de tiempos antiguos, la Topología aparece en el siglo XVII con el nombre de *analysis situs* (análisis de la posición). De manera informal, la Topología se ocupa de aquellas propiedades de las figuras que permanecen invariantes cuando dichas figuras son plegadas, dilatadas, contraídas o deformadas, de modo que no aparezcan nuevos puntos, o se hagan coincidir puntos diferentes. Los grafos son un objeto de estudio de la Topología [2].

El problema de los puentes de Königsberg (el objeto de estudio de este trabajo) fue la única contribución de Euler a la Teoría de Grafos. Aun así, N. L. Biggs, E. K. Lloyd y R. J. Wilson en su historia sobre dicha teoría calificaron este trabajo como influyente y notable [3]:

> *Los orígenes de la Teoría de Grafos son humildes e incluso triviales. Mientras que muchas ramas de las matemáticas estaban motivadas por problemas fundamentales de cálculo, cinemática y medidas, los problemas que condujeron al desarrollo de la Teoría de Grafos no solían ser más que rompecabezas, diseñados para poner a prueba el ingenio en lugar de estimular la imaginación. Pero a pesar de su aparente trivialidad, estos puzzles captaron el interés de los matemáticos, con el resultado de que la Teoría de Grafos se ha convertido en una disciplina rica en resultados teóricos de una sorprendente variedad y profundidad. [3]*

1.2. Königsberg y sus puentes

> *Recientemente se dio a conocer un problema que, si bien parecía pertenecer a la geometría, estaba, no obstante, planteado de tal modo que no exigía la determinación de una magnitud ni podía resolverse mediante cálculos numéricos. Por lo tanto, no dudé en asignarlo a la geometría de posición, sobre todo porque su solución sólo requería la consideración de la posición siendo innecesario el cálculo. [1]*

Euler introduce así en *Solutio problematis ad geometriam situs pertinentis* el famoso problema de los puentes de Königsberg. A continuación procede a describirlo:

> *En la ciudad prusiana de Königsberg hay una isla A, llamada Kneiphof, rodeada por los dos brazos del río Pregel, tal y como muestra la figura siguiente. Hay siete puentes que cruzan los dos brazos. La pregunta es si una persona puede dar un paseo de modo que cruce cada uno de estos puentes una vez pero no más de una vez. Me han dicho que aunque algunos han negado la posibilidad de conseguirlo y otros han estado en la duda, no había nadie que sostuviera que esto era realmente posible. [1]*

La primera demostración matemática de este problema fue presentada por el propio Euler a los miembros de la Academia de San Petersburgo en 1735 (puede consultarse en esta misma obra), y publicada al año siguiente en la obra que venimos citando en este trabajo [3].

Para tratar este problema, Euler procedió a abstraer la situación, convirtiéndola en un grafo donde los nodos eran las zonas de tierra y las aristas los puentes. A partir de este esquema se limitó a estudiar las características del mismo para concluir finalmente que no era posible encontrar dicho camino.

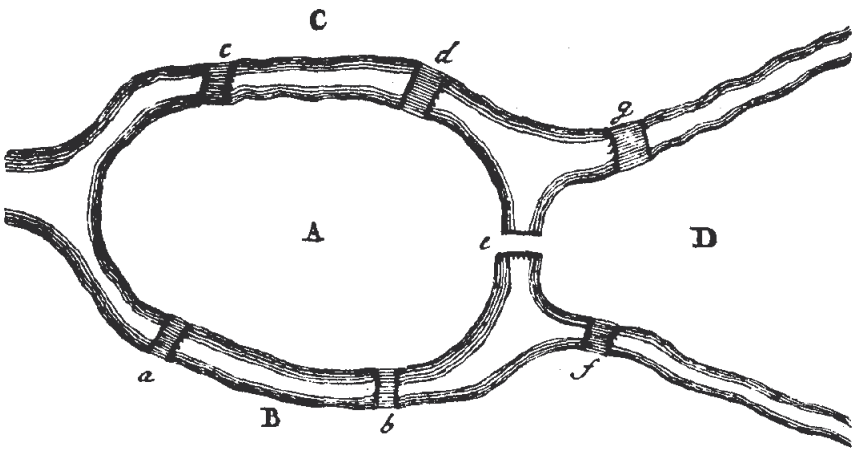


Figura 2: Esquema de la ciudad de Königsberg [1]

2. Caminos y ciclos eulerianos

2.1. Definiciones básicas sobre grafos

Comenzaremos introduciendo definiciones básicas de la Teoría de Grafos [8].

Definición 2.1 *Llamaremos grafo, $G = (V, A)$, al par ordenado formado por un conjunto finito no vacío, V , y un conjunto, A, de pares no ordenados de elementos del mismo.*

- *V es el conjunto de los vértices o nodos del grafo.*
- *A será el conjunto de las aristas o arcos del grafo.*

Definición 2.2 *Diremos que los vértices* u *y* v *son adyacentes si existe una arista* a *que los une. A los vértices* u *y* v *los llamaremos extremos de la arista.*

Definición 2.3 *Llamaremos grado o valencia de un vértice al número de aristas que incidan en él. Un vértice de grado cero se denomina aislado.*

Definición 2.4 *Un camino en un grafo G es una sucesión donde se alternan vértices y aristas, comenzando y terminando con vértices y en el que cada arista es incidente con los dos vértices que la preceden y la siguen.*

Definición 2.5 *Un ciclo en un grafo G es un camino en el que sus extremos coinciden.*

Definición 2.6 *Un grafo se dice que es conexo si cada par de sus vértices están conectados. Es decir,*

$$G \text{ conexo} \Leftrightarrow \forall u, v : \exists \mu = \langle u, v \rangle$$

siendo μ *un camino entre* u *y* v.

2.2. Caminos y ciclos eulerianos

Tras haber introducido los conceptos fundamentales de la Teoría de Grafos, procederemos a definir aquellas propiedades necesarias para resolver el problema objeto de este trabajo [8].

Definición 2.7 *Se dice que un camino de un grafo es de Euler si pasa por todos los vértices del mismo, recorriendo cada arista del mismo exactamente una vez.*

Definición 2.8 *Un ciclo de un grafo se dice de Euler si pasa por todos los vértices recorriendo cada arista exactamente una vez.*

Aquellos grafos que admiten un ciclo de Euler se denominan **grafos eulerianos**.

Hay dos teoremas que informan sobre cuándo un grafo admite ciclos o caminos eulerianos:

Teorema 2.9 (Condición necesaria de existencia de ciclos eulerianos) *Si el grafo G es euleriano, entonces todos sus vértices son de grado par.*

Teorema 2.10 (Condición necesaria de existencia de caminos eulerianos) *Si el grafo G admite un camino de Euler, entonces el número de vértices de grado impar es 2 o ninguno (en caso de que sea ninguno, el camino encontrado es un ciclo euleriano).*

Puede encontrarse una demostración de ambos teoremas en la referencia bibliográfica [8].

La resolución del problema de los puentes de Königsberg consiste en encontrar un camino euleriano en el grafo que los esquematiza. Ya que todos los nodos son de grado impar, el problema no tiene solución.

Nosotros supondremos a partir de ahora que todos los grafos con los que trabajemos son no dirigidos y conexos. Si los grafos cumplen esta última propiedad, las condiciones necesarias anteriores pasan a ser suficientes.

Teorema 2.11 (Condición suficiente de existencia de ciclos eulerianos) *Un grafo es euleriano si y sólo si es conexo y todos sus vértices son de grado par.*

Teorema 2.12 (Condición suficiente de existencia de caminos eulerianos) *Un grafo admite un camino de Euler si y sólo si es conexo y exactamente dos de sus vértices o ninguno son de grado impar (en caso de ser ninguno, el camino encontrado será un ciclo).*

Podemos aportar las siguientes observaciones:

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

3. Primeros pasos

Para la búsqueda del camino euleriano haremos uso de los algoritmos de búsqueda en espacios de estados, tanto su versión generalizada como la *búsqueda en escalada*. Las funciones de orden superior que representan ambos métodos pueden encontrarse en las librerías `I1M.BusquedaEnEspaciosDeEstados` y `I1M.BusquedaEnEscalada`, respectivamente, desarrolladas por el departamiento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Sevilla [7].

Para trabajar con grafos en Haskell hemos utilizado la librería `I1M.Grafo`, también desarrollada por el departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Sevilla. Sus funciones principales pueden encontrarse en la referencia bibliográfica [4].

3.1. Librerías iniciales y definiciones de tipos

Dado que vamos a definir nuevas instancias en el presente trabajo, necesitaremos incluir en la primerísima línea de código los siguientes pragmas:

```
{-# LANGUAGE FlexibleInstances, TypeSynonymInstances #-}
```

Las librerías que necesitaremos en el desarrollo de nuestro proyecto son las siguientes:

```
import I1M.Grafo

import qualified Data.Map as M
import Data.Maybe
import Test.QuickCheck

import I1M.BusquedaEnEspaciosDeEstados
import I1M.BusquedaEnEscalada
```

También definiremos los siguientes tipos para facilitar la lectura del código:

```
type GrafoI = Grafo Int Int
type Arista = (Int,Int,Int)
type MultiConj = M.Map Arista Int
```

3.2. Funciones auxiliares

Usaremos estas funciones como ayuda para la definición de las funciones principales, muchas de ellas tomadas de la referencia bibliográfica [5]:

```
completo :: Int -> Grafo Int Int
completo n = creaGrafo ND (1,n) [(a,b,0)|a<-[1..n],b<-[1..a-1]]
-- Nos da el grafo completo de n vértices.

-- Lazos y aristas
arista :: Int -> Int -> Arista
arista x y = (x,y,0)

aristaOp :: Arista -> Arista
aristaOp (x,y,z) = (y,x,z)
-- Proporciona la arista opuesta a una dada

nLazos :: GrafoI ->  Int
nLazos g = length [(x,y) | (x,y,z) <- aristas g, x == y]

nAristas :: GrafoI ->  Int
nAristas g
    | dirigido g = length (aristas g)
    | otherwise  = (length (aristas g) `div` 2) + nLazos g

-- Grado de un vértice en un grafo
gradoPos :: GrafoI -> Int -> Int
gradoPos g = length . adyacentes g
```

Ya que estamos suponiendo que nuestros grafos son no dirigidos, la arista (x, y, z) es la misma que la arista (y, x, z).

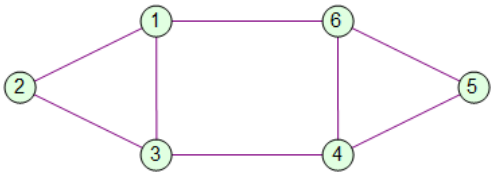
3.3. Batería de ejemplos

Usaremos estos grafos como ejemplos (tomados de la referencia bibliográfica [8]):

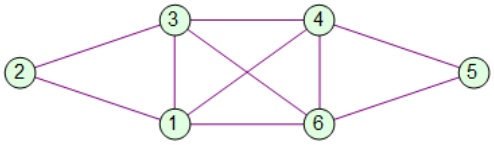
```
g1, g2, g3, g4, g5, g6, g7, g8, g9 :: GrafoI
g10, g11, g12, g13, g14, g15, g16, g17, gFallo :: GrafoI
```

```
g1 = creaGrafo ND (1,6) [(1,2,0),(1,3,0),(1,6,0),(2,3,0),
                         (3,4,0),(4,5,0),(4,6,0),(5,6,0)]
g2 = creaGrafo ND (1,6) [(1,2,0),(1,3,0),(1,4,0),(1,6,0),(2,3,0),
                         (3,4,0),(3,6,0),(4,5,0),(4,6,0),(5,6,0)]
g3 = creaGrafo ND (1,4) [(1,2,0),(1,4,0),(2,3,0),(3,4,0)]
g4 = creaGrafo ND (1,4) [(1,2,0),(1,3,0),(1,4,0),(2,3,0),(3,4,0)]
g5 = creaGrafo ND (1,4) [(1,2,0),(1,3,0),(1,4,0),(2,3,0),(2,4,0),(3,4,0)]
g6 = creaGrafo ND (1,8) [(1,2,0),(1,3,0),(2,3,0),(2,4,0),(2,5,0),(3,4,0),
                         (3,5,0),(4,5,0),(4,6,0),(5,6,0),(6,7,0),(6,8,0),(7,8,0)]
g7 = creaGrafo ND (1,5) [(1,2,0),(1,4,0),(1,5,0),(2,3,0),
                         (2,4,0),(2,5,0),(3,4,0),(4,5,0)]
g8 = creaGrafo ND (1,4) [(1,2,0),(1,2,0),(1,4,0),(2,3,0),
                         (2,3,0),(2,4,0),(3,4,0)]
g9 = creaGrafo ND (1,10) [(1,2,0),(1,6,0), (1,7,0),(1,8,0), (2,3,0), (3,4,0),
                          (3,8,0),(3,9,0), (4,5,0),(4,9,0), (4,10,0),(5,6,0),
                          (6,7,0),(6,10,0),(7,8,0),(7,10,0),(8,9,0), (9,10,0)]
g10 = creaGrafo ND (1,6) [(1,2,0),(1,6,0),(2,3,0),(3,4,0),
                          (4,5,0),(5,6,0),(5,6,0)]
g11 = creaGrafo ND (1,8) [(1,2,0),(1,3,0),(1,6,0),(1,7,0),(1,8,0),(2,3,0),(3,4,0),
                          (3,5,0),(3,8,0),(4,5,0),(5,6,0),(5,8,0),(6,7,0),(6,8,0)]
g12 = creaGrafo ND (1,9) [(1,2,0),(1,3,0),(1,7,0),(1,8,0),
                          (1,9,0),(2,3,0),(3,4,0),(3,5,0),
                          (3,9,0),(4,5,0),(5,6,0),(5,7,0),
                          (5,9,0),(6,7,0),(7,8,0),(7,9,0)]
g13 = creaGrafo ND (1,15) [(1,2,0),  (1,10,0), (2,3,0),  (2,11,0), (2,12,0),
                           (3,4,0),  (4,5,0),  (4,12,0), (4,13,0), (5,6,0),
                           (6,7,0),  (6,13,0), (6,14,0), (7,8,0),  (8,9,0),
                           (8,14,0), (8,15,0), (9,10,0), (10,11,0),(10,15,0),
                           (11,12,0),(11,15,0),(12,13,0),(13,14,0),(14,15,0)]
g14 = creaGrafo ND (1,4) [(1,2,0),(2,3,0),(2,3,0),(2,4,0),(3,4,0),(4,1,0)]
g15 = creaGrafo ND (1,10) [(1,2,0),(1,7,0),(1,9,0),(1,10,0),
                           (2,3,0),(2,7,0),(3,4,0),(3,6,0),
                           (4,6,0),(5,6,0),(5,8,0),(6,8,0),
                           (6,7,0),(6,9,0),(9,7,0),(9,10,0)]
g16 = creaGrafo ND (1,9) [(1,2,0),(1,2,0),(1,5,0),(1,6,0),(2,5,0),
                          (2,6,0),(3,4,0),(3,7,0),(4,5,0),
                          (5,6,0),(6,9,0),(8,9,0),(5,7,0)]
g17 = creaGrafo ND (1,9) [(1,2,0),(2,4,0),(3,4,0),(3,6,0),(3,7,0),(6,7,0),
                          (6,9,0),(1,5,0),(5,8,0),(7,8,0),(4,5,0),(4,5,0),
                          (4,8,0),(5,7,0),(4,7,0),(8,9,0),(7,9,0)]
gFallo = creaGrafo ND (1,7) [(1,2,0),(1,3,0),(1,6,0),(2,3,0),(2,5,0),
                             (2,6,0),(2,7,0),(3,4,0),(3,7,0),
                             (4,5,0),(5,6,0),(5,7,0),(6,7,0)]
```

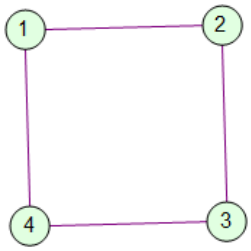
Adjuntamos también sus representaciones gráficas:



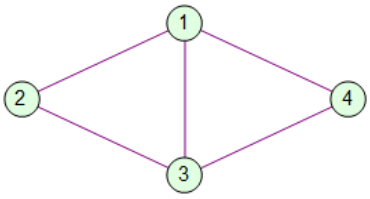
Grafo I



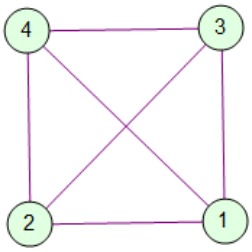
Grafo II



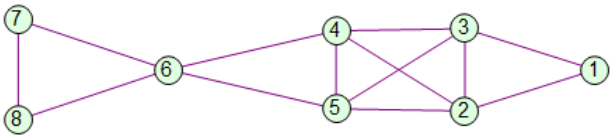
Grafo III



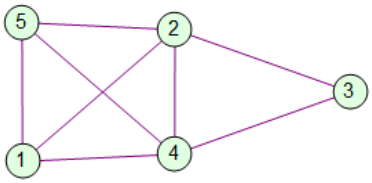
Grafo IV



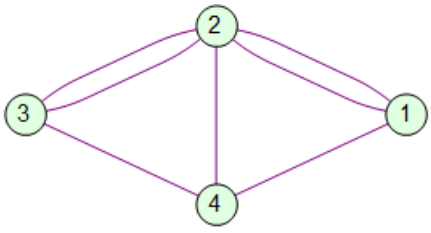
Grafo V



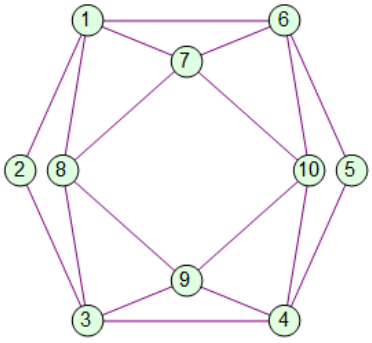
Grafo VI



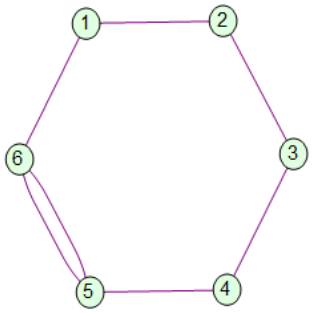
Grafo VII



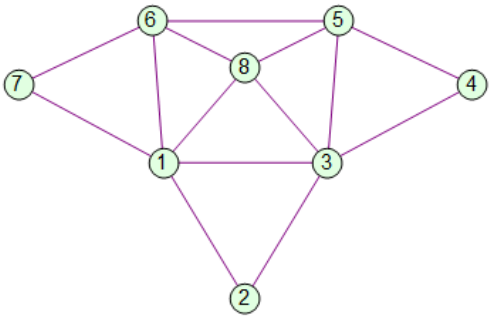
Grafo VIII



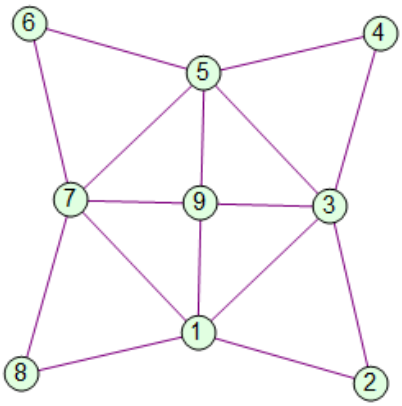
Grafo IX



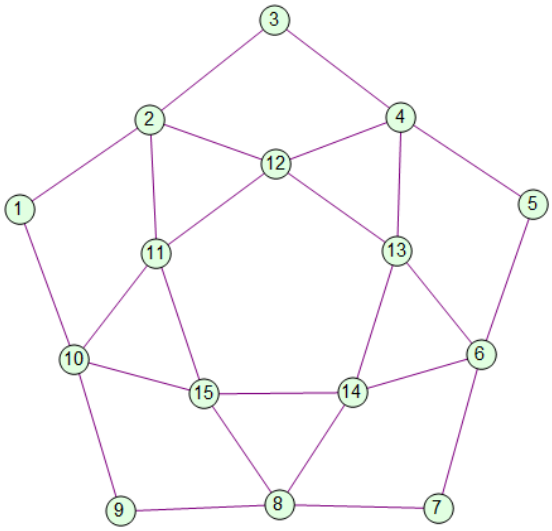
Grafo X



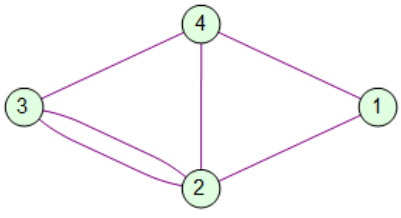
Grafo XI



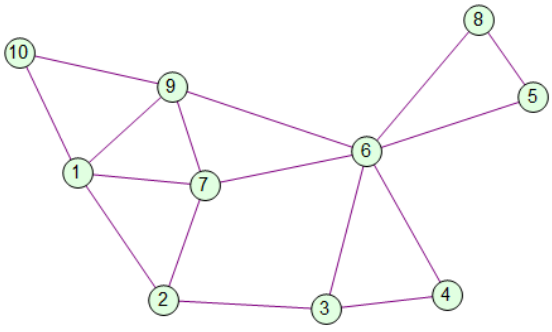
Grafo XII



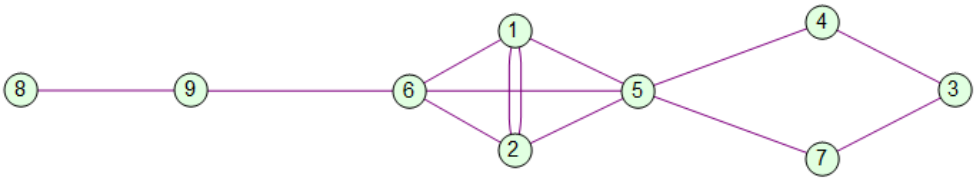
Grafo XIII



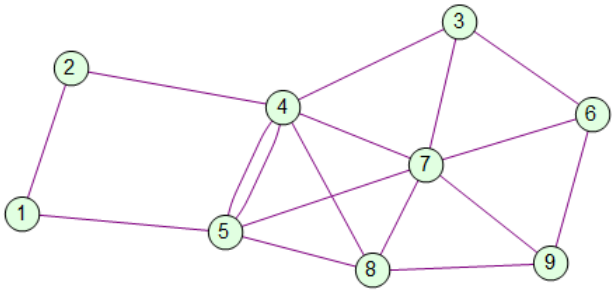
Grafo XIV



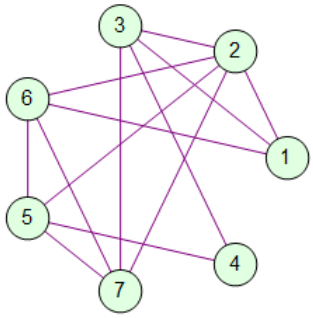
Grafo XV



Grafo XVI



Grafo XVII



Grafo de fallo

3.4. Los contenedores de aristas como multiconjuntos

En nuestro algoritmo necesitaremos registrar aquellas aristas que han sido recorridas (lo explicaremos más adelante). Cuando el grafo es simple (dos vértices cualesquiera están unidos por una única arista), no hay demasiados inconvenientes en utilizar listas para llevar ese registro. Sin embargo, cuando se nos presenta un multigrafo (al menos un par de vértices está unido por más de una arista), la cosa no es igual de sencilla. Es por ello que nos decantamos por los multiconjuntos para usarlos como contenedores de aristas. En un contenedor de aristas, una arista y su opuesta deben considerarse como iguales (en el caso de nuestras suposiciones).

Un multiconjunto es una colección de elementos en el que el cardinal de cada uno de ellos (el número de veces que aparece en el mismo) sí importa, a diferencia de los conjuntos.

Las funciones principales de multiconjuntos pueden consultarse en la web del departamento de Ciencias de la Computación e Inteligencia Artificial (Universidad de Sevilla) [6]. Hemos incluido las funciones necesarias para nuestro trabajo en un anexo final.

Nos serán de utilidad las siguientes funciones sobre multiconjuntos:

```
conjAristas :: GrafoI -> MultiConj
conjAristas g = (`div` 2)  <$>  aux (aristas g) vacio
  where aux []     m = m
        aux (x:xs) m | prop x m   =  aux xs (inserta x' m)
                     | otherwise  =  aux xs (inserta x  m)
          where x'       = aristaOp x
                prop x m = x `noPertenece` m  &&  x' `pertenece` m
```

```
-- Los multiconjuntos (diccionarios) son functores.

nAr :: Arista -> MultiConj -> Int
nAr x m | x `pertenece` m   =  ocurrencias x m
        | x' `pertenece` m  =  ocurrencias x' m
        | otherwise         =  0
  where x' = aristaOp x

insAr :: Arista -> MultiConj -> MultiConj
insAr x m | x' `pertenece` m  =  inserta x' m
          | otherwise         =  inserta x m
  where x' = aristaOp x
```

- La función `conjAristas` nos proporciona el multiconjunto de las aristas de un grafo.
- La función `nAr` cuenta el número de aristas que tiene un multiconjunto de aristas dado (considerando que una arista y su opuesta son lo mismo).
- A la hora de introducir una arista en un multiconjunto debemos comprobar la presencia de la arista opuesta para que, en caso afirmativo, la contabilice como la misma. Esta es la finalidad de `insAr`.

3.5. Implementación de las propiedades sobre caminos eulerianos

Dado que todos los grafos con los que trabajaremos son conexos, podemos definir la condición suficiente de existencia de caminos y ciclos eulerianos:

```
nNodosImp :: GrafoI -> Int
nNodosImp g = length (filter odd (map (gradoPos g) (nodos g)))
-- Proporciona el número de nodos de grado impar de un grafo

hayCamino :: GrafoI -> Bool
hayCamino g = nNodosImp g  `elem`  [0,2]

hayCiclo :: GrafoI -> Bool
hayCiclo g = nNodosImp g == 0
```

4. Algoritmos para resolver el problema

Como ya hemos comentado, los dos métodos que usaremos para resolver el problema se basan en la búsqueda en espacios de estados. Las características generales de estos problemas son [7]:

- Un conjunto de las posibles situaciones o *estados* que constituye el espacio de estados (las posibles soluciones a explorar).
- Un conjunto de movimientos de un estado a otro: los *sucesores* del nodo.
- Un *estado inicial*.
- Un *estado final* o propiedad que debe verificar un estado para ser final.

Para aplicar estos métodos usaremos las funciones de orden superior cuyas signaturas son:

```
buscaEE :: (Eq nodo) =>(nodo ->[nodo]) ->(nodo ->Bool) ->nodo ->[nodo]
```

```
buscaEscalada :: Ord nodo =>(nodo ->[nodo]) ->(nodo ->Bool) ->nodo ->[nodo]
```

Definiremos el tipo `nodo` (al que llamaremos `Estado`) como:

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

Por razones que explicaremos en la sección 4.2.2, debidas al algoritmo de la búsqueda en escalada, necesitamos definir una instancia de la clase `Ord` para nuestro nuevo tipo:

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

donde nuestra heurística es

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

4.1. Funciones base de la búsqueda en espacios de estados

Las definiciones que expondremos a continuación nos valdrán tanto para la búsqueda en espacios de estados como para la búsqueda en escalada (las diferencias entre ambos no residen en las funciones auxiliares, sino en las respectivas implementaciones).

4.1.1. La función `inicial`

Nuestro camino euleriano podría comenzar en cualquiera de los nodos, por lo que no podemos definir un estado inicial prefijado para un determinado vértice.

```
inicial :: GrafoI -> Int -> Estado
inicial g k = EST ([nodos g !! (k-1)],vacio)
```

Realizamos los siguientes apuntes respecto a la función anterior:

- Tal y como está definida, `k` debe variar entre 1 y el número de nodos del grafo.
- La variación de `k` supondrá elegir un nodo diferente de todos los que conforman el grafo.

4.1.2. La función `esFinal`

```
esFinal :: GrafoI -> Estado -> Bool
esFinal g (EST (_,m)) = cardinal m == nAristas g
```

Un estado será final cuando haya recorrido todas las aristas del grafo y exactamente una sola vez. Tal y como construiremos la función `sucesores`, `esFinal` solamente deberá verificar que el multiconjunto de aristas contenga todas las del grafo.

4.1.3. La función `sucesores`

```
sucesores :: GrafoI -> Estado -> [Estado]
sucesores g (EST (x:xs,ys)) = [EST (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
```

Dado un estado, compuesto por la lista de nodos y el multiconjunto de aristas, un sucesor del mismo será otro estado:

- Cuya lista de nodos contenga un nodo más, perteneciente a los adyacentes del último nodo recorrido.
- Cuyo multiconjunto de aristas contenga a la arista formada por el último nodo recorrido y el último añadido a la lista.

La función auxiliar `prop` es la encargada de verificar que al introducir una nueva arista en el multiconjunto, el total de ellas no supere el número de aristas de `conjAristas` del grafo.

- Si la arista no pertenece al multiconjunto (lo que equivale a que `nAr h ys == 0`), puede introducirse sin problemas.
- Si la arista ya pertenece al multiconjunto, debemos comprobar que al introducirla no se supere la cantidad de la misma que hay en `conjAristas`.

4.2. Resolución del problema

A continuación comentaremos las soluciones del problema siguiendo los dos métodos de los que venimos hablando a lo largo de este trabajo. Definiremos dos funciones, `caminosXX` y `caminoXX` (`XX` puede ser espacios de estados generalizada o en escalada).

- La función `caminosXX` nos proporciona todos los caminos eulerianos de un grafo, encontrados mediante búsqueda en espacios de estados. Hacemos que `k` varíe desde 1 hasta el número de nodos para que vayamos cambiando de nodo inicial a la hora de buscar un camino.
- La función `caminoXX` nos proporciona (usando el tipo `Maybe`) justo una posible solución (la primera entrada de la lista de soluciones) o ninguna (en caso de que la lista sea vacía). Como camino entenderemos la sucesión de nodos recorridos, por lo que sólo nos quedaremos con la primera componente del par del estado final.

4.2.1. Búsqueda en espacios de estados generalizada

La ventaja que tiene usar este método es que siempre obtendremos una solución. Sin embargo, la eficiencia no es su punto fuerte ya que la función genera todas las posibles soluciones. El volumen de datos que genera la búsqueda en espacios de estados es muy grande.

```
caminosEE :: GrafoI -> [Estado]
caminosEE g = concat [buscaEE (sucesores g) (esFinal g) (inicial g k) | k <- [1..n]]
  where n = length (nodos g)

caminoEE :: GrafoI -> Maybe [Int]
caminoEE g = aux (caminosEE g)
  where aux []          = Nothing
        aux ((EST x):_) = Just $ fst x
```

4.2.2. Resolución del problema mediante búsqueda en escalada

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.

Este algoritmo 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 alcancemos el pie de la misma. Este es el mayor inconveniente del método [10]. Habrá grafos que, aun teniendo un camino euleriano, no podremos encontrarlo con este método. Puede probarse aplicando la función al grafo `gFallo`.

La búsqueda en escalada usa como herramienta principal las colas de prioridad (colas cuyos elementos no están ordenados por llegada, sino por algún otro criterio). El tipo `Estado` definido por nosotros no tiene asignado ningún criterio de ordenación. Es por ello que al principio del código tuvimos que definir la instancia correspondiente (ver página **??**).

La heurística que definimos nos da una idea de cuán cerca nos encontramos del estado final. Parece lógico pensar que esto ocurrirá cuanto mayor sea el número de aristas de nuestro multiconjunto.

Con todas estas consideraciones, las funciones usando búsqueda en escalada resultan:

```
caminosEsc :: GrafoI -> [Estado]
caminosEsc g = concat
  [buscaEscalada (sucesores g) (esFinal g) (inicial g k) | k <- [1..n]]
  where n = length (nodos g)

caminoEsc :: GrafoI -> Maybe [Int]
caminoEsc g = aux (caminosEsc g)
  where aux []          = Nothing
        aux ((EST x):_) = Just $ fst x
```

5. Función final y comprobación de propiedades

En la sección anterior hemos expuesto dos métodos para encontrar un camino euleriano en un grafo no dirigido. Una pregunta que podemos formularnos en este punto es: ¿cuál de los dos es el mejor?

Pensemos antes de responderla en la siguiente cuestión: ¿a cuántas ciudades podemos llegar directamente por autopista? A prácticamente ninguna. Son las carreteras secundarias las que llegan a todos los rincones de la geografía de un territorio. Es evidente que cuanto mayor sea el recorrido que hagamos por autopista menor será el tiempo empleado en llegar a nuestro destino, pero no podemos evitar usar en algunos momentos las carreteras para llegar. La mejor opción será una combinación óptima de trayecto por autopista y carretera.

Volvamos ahora a nuestro objeto de estudio. Hemos comentado que hay grafos para los cuales la función `caminoEsc` no encuentra solución pero `caminoEE` sí (y además los teoremas aseguran su existencia). Por tanto, una adecuada combinación de ambos métodos consistirá en usar la búsqueda en espacios de estados generalizada únicamente cuando `caminoEsc` no lo encuentre y los teoremas garanticen su existencia.

Ahora bien, *¿cuándo es absolutamente imprescindible usar la búsqueda en espacios de estados generalizada?*

- 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 los teoremas nos aseguran que existe, entonces será `buscaEE` quien nos lo dé.

Atendiendo a lo que acabamos de exponer, la función final resulta:

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

Para concluir, hemos definido dos propiedades para ser analizadas con QuickCheck:

```
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

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

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

El generador de grafos puede encontrarse en el Anexo II.

En la primera propiedad hemos usado la función `esConexo`, que verifica si un grafo es conexo o no. En el Anexo III puede encontrarse su código.

6. Conclusiones

En el presente trabajo hemos analizado con ayuda de la informática y los ordenadores un problema propuesto por Euler en el siglo XVIII. Gracias al lenguaje de programación que hemos usado, no ha sido demasiado difícil.

Por suerte disponemos de teoremas que nos advierten de la existencia de un camino euleriano en un grafo, lo que facilita enormemente la tarea. Sin embargo, existen otros problemas sobre grafos de los que no disponemos de ningún teorema ni método, como es el caso de los grafos hamiltonianos.

Tal y como hemos expuesto aquí el problema, la resolución del mismo puede parecer un simple pasatiempo. Sin embargo, la trascendencia de este problema es mucho mayor. La Teoría de Grafos y sus problemas se encuentran presentes en multitud de campos, como la informática, la logística, las telecomunicaciones, la organización de medios de transporte y muchos más. Encontrar métodos informáticos eficientes que los resuelvan facilita enormemente la tarea a aquellos que trabajan diariamente en esas disciplinas. Es por ello que no debemos cesar de investigar y profundizar en este y muchos otros campos de las Matemáticas, tan útiles y necesarias en las sociedades modernas.

Anexo I: Implementación del TAD de los multiconjuntos

A continuación presentamos las funciones de multiconjuntos necesarias para el desarrollo de nuestro programa. Pueden encontrarse en la referencia bibliográfica [6].

```
vacio :: MultiConj
vacio = M.empty

inserta :: Arista -> MultiConj -> MultiConj
inserta x = M.insertWith (+) x 1

borra :: Arista -> MultiConj -> MultiConj
borra = M.update f
  where f m | m <= 1    = Nothing
            | otherwise = Just (m - 1)

esVacio :: MultiConj -> Bool
esVacio = M.null

cardinal :: MultiConj -> Int
cardinal = sum . M.elems

pertenece :: Arista -> MultiConj -> Bool
```

```
pertenece = M.member

noPertenece :: Arista -> MultiConj -> Bool
noPertenece = M.notMember

ocurrencias :: Arista -> MultiConj -> Int
ocurrencias = M.findWithDefault 0
```

Anexo II: Generador de grafos

En este anexo recogemos un generador de grafos necesario para comprobar las propiedades con QuickCheck. El código puede encontrarse en la referencia bibliográfica [5].

```
-- (generaGND n ps) es el grafo completo de orden n tal que los pesos
-- están determinados por ps. Por ejemplo,
--    ghci> generaGND 3 [4,2,5]
--    (ND,array (1,3) [(1,[(2,4),(3,2)]),
--                     (2,[(1,4),(3,5)]),
--                      3,[(1,2),(2,5)])])
--    ghci> generaGND 3 [4,-2,5]
--    (ND,array (1,3) [(1,[(2,4)]),(2,[(1,4),(3,5)]),(3,[(2,5)])])
generaGND :: Int -> [Int] -> GrafoI
generaGND n ps  = creaGrafo ND (1,n) l3
    where l1 = [(x,y) | x <- [1..n], y <- [1..n], x < y]
          l2 = zip l1 ps
          l3 = [(x,y,z) | ((x,y),z) <- l2, z > 0]

-- (generaGD n ps) es el grafo completo de orden n tal que los pesos
-- están determinados por ps. Por ejemplo,
--    ghci> generaGD 3 [4,2,5]
--    (D,array (1,3) [(1,[(1,4),(2,2),(3,5)]),
--                    (2,[]),
--                    (3,[])])
--    ghci> generaGD 3 [4,2,5,3,7,9,8,6]
--    (D,array (1,3) [(1,[(1,4),(2,2),(3,5)]),
--                    (2,[(1,3),(2,7),(3,9)]),
--                    (3,[(1,8),(2,6)])])
generaGD :: Int -> [Int] -> GrafoI
generaGD n ps = creaGrafo D (1,n) l3
    where l1 = [(x,y) | x <- [1..n], y <- [1..n]]
          l2 = zip l1 ps
          l3 = [(x,y,z) | ((x,y),z) <- l2, z > 0]
```

```
-- genGD es un generador de grafos dirigidos. Por ejemplo,
--    ghci> sample genGD
--    (D,array (1,4) [(1,[(1,1)]),(2,[(3,1)]),(3,[(2,1),(4,1)]),(4,[(4,1)])])
--    (D,array (1,2) [(1,[(1,6)]),(2,[])])
--    ...
genGD :: Gen GrafoI
genGD = do n <- choose (1,10)
           xs <- vectorOf (n*n) arbitrary
           return (generaGD n xs)

-- genGND es un generador de grafos dirigidos. Por ejemplo,
--    ghci> sample genGND
--    (ND,array (1,1) [(1,[])])
--    (ND,array (1,3) [(1,[(2,3),(3,13)]),(2,[(1,3)]),(3,[(1,13)])])
--    ...
genGND :: Gen GrafoI
genGND = do n <- choose (1,10)
            xs <- vectorOf (n*n) arbitrary
            return (generaGND n xs)

-- genG es un generador de grafos. Por ejemplo,
--    ghci> sample genG
--    (D,array (1,3) [(1,[(2,1)]),(2,[(1,1),(2,1)]),(3,[(3,1)])])
--    (ND,array (1,3) [(1,[(2,2)]),(2,[(1,2)]),(3,[])])
--    ...
genG :: Gen GrafoI
genG = do d <- choose (True,False)
          n <- choose (1,10)
          xs <- vectorOf (n*n) arbitrary
          if d then return (generaGD n xs)
               else return (generaGND n xs)

-- Los grafos está contenido en la clase de los objetos generables
-- aleatoriamente.
instance Arbitrary GrafoI where
    arbitrary = genG
```

Anexo III: Grafos conexos

Un grafo es conexo si podemos encontrar un camino entre dos nodos cualesquiera del mismo. Es claro que si un grafo cumple esta propiedad, siempre habrá un camino que une dos nodos tal que no se repita ninguno de ellos.

Aprovechando esta propiedad definimos las siguientes funciones:

```
caminos :: Grafo Int Int -> Int -> Int -> [[Int]]
caminos g a b = buscaEE sucesores esFinal inicial
    where inicial          = [b]
          sucesores (x:xs) = [z:x:xs | z <- adyacentes g x
                                     , z `notElem` (x:xs)]
          esFinal (x:xs)   = x == a

esConexo :: Grafo Int Int -> Bool
esConexo g = and [not (null (caminos g v w)) | v <- xs, w <- xs]
  where xs = nodos g
```

- La función `caminos` encuentra todos los caminos en el grafo desde `a` hasta `b` sin pasar dos veces por el mismo nodo.
- La función `esConexo` verifica que `caminos` nunca proporciona la lista vacía para un par cualquiera de nodos del grafo.

Referencias

[1] Leonhard Euler, *Solutio problematis ad geometriam situs pertinentis*. Texto que aparece en: James R. Newman, *The World of Mathematics*, Volumen I.

[2] Marta Macho Stadler, *¿Qué es la Topología?*, Universidad del País Vasco, 2002. Puede consultarse en la web: `http://www.ehu.eus/~mtwmastm/sigma20.pdf`

[3] Gerald L. Alexanderson *Euler and Königsberg?s bridges: a historical view*, de *Bulletin of the American Mathematical Society, Volume 43, Number 4, October 2006*. Puede consultarse en la web: `http://www.ams.org/journals/bull/2006-43-04/S0273-0979-06-01130-X/S0273-0979-06-01130-X.pdf`

[4] José Antonio Alonso Jiménez, *Implementación del TAD de los grafos en Haskell*, Universidad de Sevilla: `http://www.cs.us.es/~jalonso/cursos/i1m/temas/tema-22.html`

[5] José Antonio Alonso Jiménez, *Problemas básicos con el TAD de los grafos*, Universidad de Sevilla: `http://www.glc.us.es/~jalonso/vestigium/i1m20102-problemas-basicos-con-el-tad-de-los-grafos`

[6] José Antonio Alonso Jiménez, *Implementación del TAD de los multiconjuntos en Haskell*, Universidad de Sevilla: `http://www.glc.us.es/~jalonso/vestigium/i1m2015-el-tad-de-los-multiconjuntos-mediante-diccionarios-en-haskell`

[7] José Antonio Alonso Jiménez, *Técnicas de diseño descendente de algoritmos*, Universidad de Sevilla: `https://www.cs.us.es/~jalonso/cursos/i1m/temas/tema-23.html`

[8] Francisco José González Gutiérrez, *Apuntes de Matemática Discreta. Lección 14: Grafos*, Universidad de Cádiz. Puede encontrarse una copia del documento aquí: `http://www2.uca.es/matematicas/Docencia/ESI/1711003/Apuntes/Leccion14.pdf`.

[9] Claudio Cifuentes, *Grafos eulerianos y hamiltonianos*: `http://teoriadegrafos.blogspot.com.es/2007/03/grafos-eulerianos-y-hamiltonianos.html`

[10] F. Rabhi y G. Lapalme, *Algorithms: a functional programming approach*, Addison-Wesley.

[11] `http://lamasbolano.com/blog/tag/leonhard-euler`