

# Matemática discreta en Haskell

María Dolores Valverde Rodríguez

---

Grupo de Lógica Computacional

Dpto. de Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

Sevilla, 21 de junio de 2016 (Versión de 31 de julio de 2016)

Esta obra está bajo una licencia Reconocimiento–NoComercial–CompartirIgual 2.5 Spain de Creative Commons.

**Se permite:**

- copiar, distribuir y comunicar públicamente la obra
- hacer obras derivadas

**Bajo las condiciones siguientes:**



**Reconocimiento.** En cualquier explotación de la obra autorizada por la licencia hará falta reconocer la autoría.



**No comercial.** La explotación de la obra queda limitada a usos no comerciales.



**Compartir bajo la misma licencia.** La explotación autorizada incluye la creación de obras derivadas siempre que mantengan la misma licencia al ser divulgadas.

Esto es un resumen del texto legal (la licencia completa). Para ver una copia de esta licencia, visite <http://creativecommons.org/licenses/by-nc-sa/2.5/es/> o envíe una carta a Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

# Índice general

<b>Introducción</b>	<b>7</b>
<b>1 Conjuntos, relaciones y funciones</b>	<b>9</b>
1.1 Conjuntos	9
1.1.1 Primeras definiciones	10
1.1.2 Pertenencia a un conjunto	11
1.1.3 Conjunto vacío	11
1.1.4 Conjunto unitario	12
1.1.5 Subconjuntos	12
1.1.6 Igualdad de conjuntos	12
1.1.7 Subconjuntos propios	13
1.1.8 Complementario de un conjunto	13
1.1.9 Cardinal de un conjunto	14
1.1.10 Unión de conjuntos	14
1.1.11 Intersección de conjuntos	14
1.1.12 Producto cartesiano	15
1.1.13 Combinaciones	15
1.1.14 Variaciones con repetición	16
1.2 Relaciones	16
1.2.1 Relación binaria	16
1.2.2 Imagen por una relación	17
1.2.3 Dominio de una relación	17
1.2.4 Rango de una relación	17
1.2.5 Antiimagen por una relación	18
1.2.6 Relación funcional	18
1.3 Relaciones homogéneas	18
1.3.1 Relaciones reflexivas	19
1.3.2 Relaciones simétricas	20
1.3.3 Relaciones antisimétricas	21
1.3.4 Relaciones transitivas	21

1.3.5	Relaciones de equivalencia . . . . .	22
1.3.6	Relaciones de orden . . . . .	23
1.3.7	Clases de equivalencia . . . . .	24
1.4	Funciones . . . . .	24
1.4.1	Imagen por una función . . . . .	25
1.4.2	Funciones inyectivas . . . . .	26
1.4.3	Funciones sobreyectivas . . . . .	26
1.4.4	Funciones biyectivas . . . . .	26
1.4.5	Inversa de una función . . . . .	28
<b>2</b>	<b>Introducción a la teoría de grafos</b>	<b>29</b>
2.1	Definición de grafo . . . . .	31
2.2	El TAD de los grafos . . . . .	32
2.2.1	Grafos como listas de aristas . . . . .	33
2.3	Generadores de grafos . . . . .	35
2.4	Ejemplos de grafos . . . . .	36
2.4.1	Grafo ciclo . . . . .	37
2.4.2	Grafo de la amistad . . . . .	38
2.4.3	Grafo completo . . . . .	38
2.4.4	Grafo bipartito . . . . .	39
2.4.5	Grafo estrella . . . . .	40
2.4.6	Grafo rueda . . . . .	40
2.4.7	Grafo circulante . . . . .	41
2.4.8	Otros grafos importantes . . . . .	42
2.5	Definiciones y propiedades . . . . .	44
2.5.1	Definiciones de grafos . . . . .	44
2.5.2	Propiedades de grafos . . . . .	50
2.5.3	Operaciones y propiedades sobre grafos . . . . .	51
2.6	Morfismos de grafos . . . . .	55
2.6.1	Morfismos . . . . .	55
2.6.2	Isomorfismos . . . . .	57
2.6.3	Automorfismos . . . . .	61
2.7	Conectividad de grafos . . . . .	62
2.7.1	Caminos . . . . .	62
2.8	Sistemas utilizados . . . . .	73
2.9	Mapa de decisiones de diseño . . . . .	74
	<b>Bibliografía</b>	<b>76</b>

Índice general	5
Índice de definiciones	77
Lista de tareas pendientes	80



# Introducción

El objetivo del trabajo es la implementación de algoritmos de Matemática Discreta en Haskell. Los puntos de partida son

- los temas de la asignatura [Matemática discreta](#)<sup>1</sup> ([2])
- los temas de la asignatura [Informática](#)<sup>2</sup> ([1])
- el capítulo 7 del libro [Algorithms: A functional programming approach](#)<sup>3</sup> ([3]) y
- el artículo [Graph theory](#)<sup>4</sup> ([4]) de la Wikipedia.

---

<sup>1</sup><https://dl.dropboxusercontent.com/u/15420416/tiddly/emptyMD1314.html>

<sup>2</sup><https://www.cs.us.es/~jalonso/cursos/i1m-15>

<sup>3</sup><http://www.iro.umontreal.ca/~lapalme/Algorithms-functional.html>

<sup>4</sup>[https://en.wikipedia.org/wiki/Graph\\_theory](https://en.wikipedia.org/wiki/Graph_theory)





# Capítulo 1

## Conjuntos, relaciones y funciones

### 1.1. Conjuntos

El concepto de *conjunto* aparece en todos los campos de las Matemáticas, pero, ¿qué debe entenderse por él? La *Teoría de conjuntos* fue introducida por Georg Cantor (1845-1917); desde 1869, Cantor ejerció como profesor en la Universidad de Halle y entre 1879 y 1884 publicó una serie de seis artículos en el *Mathematische Annalen*, en los que hizo una introducción básica a la teoría de conjuntos. En su *Beiträge zur Begründung der transfiniten Mengenlehre*, Cantor dio la siguiente definición de conjunto:

§ I

**The Conception of Power or Cardinal Number**

By an “aggregate” (*Menge*) we are to understand any collection into a whole (*Zusammenfassung zu einem Ganzen*)  $M$  of definite and separate objects  $m$  of our intuition or our thought. These objects are called the “elements” of  $M$ .

Figura 1.1: Fragmento del texto traducido al inglés en el que Cantor da la definición de conjunto

«Debemos entender por “conjunto” (*Menge*) cualquier colección vista como un todo (*Zusammenfassung zu einem Ganzen*),  $M$ , de objetos separados y bien definidos,  $m$ , de nuestra intuición o pensamiento. Estos objetos son los “elementos” de  $M$ »

Felix Hausdorff, en 1914, dice: «un conjunto es una reunión de cosas que constituyen una totalidad; es decir, una nueva cosa», y añade: «esto puede difícilmente ser una definición, pero sirve como demostración expresiva del concepto de conjunto a través de conjuntos sencillos como el conjunto de habitantes de una ciudad o el de átomos de Hidrógeno del Sol».

Un conjunto así definido no tiene que estar compuesto necesariamente de elementos homogéneos y además, da lugar a cuestiones filosóficas como si podemos llamar *conjunto* a aquel que no posee ningún elemento. Matemáticamente, conviene aceptar solo elementos que compartan alguna propiedad y definir el *conjunto vacío* como aquel que no tiene elemento alguno.

El gran mérito de Cantor fue considerar conjuntos *transfinitos* (que tiene infinitos elementos), concepto inaudito hasta avanzado el siglo XIX, hablar del *cardinal* de un conjunto como el número de sus elementos y hablar de *conjuntos equivalentes* cuando puede establecerse una biyección entre ellos; ideas ya apuntadas por Bolzano, quien se centró demasiado en el aspecto filosófico, sin llegar a formalizar sus ideas.

A lo largo de la sección, haremos una pequeña introducción a la Teoría de Conjuntos, presentando formalmente sus conceptos más importantes.

Darle formato, ampliar y organizar organizar conjuntos.

Añadir referencias de conjuntos a la bibliografía.

A la hora de trabajar con conjuntos en Haskell, lo haremos con su representación como listas; por ello, utilizaremos el paquete `Data.List`.

Comentar las distintas opciones de representación de conjuntos del [tema 17 de IIM](#)

Comentar la representación elegida de conjunto.

Decidir si usar la misma representación que en `GrafoConListaDeAristas.lhs` (listas ordenadas sin repeticiones))

Comentar la posibilidad de usar la [librería de conjuntos de Haskell \(Data.Set\)](#)

### 1.1.1. Primeras definiciones

**Definición 1.1.1.** Llamaremos *conjunto* a una colección de objetos, que llamaremos *elementos*, distintos entre sí y que comparten una propiedad. Para que un conjunto esté bien definido debe ser posible discernir si un objeto arbitrario está o no en él.

Comentar la consecuencia sobre los tipos de la última frase de la definición de conjunto.

*Nota 1.1.1.* Al trabajar con la representación de conjuntos como listas en Haskell, hemos de cuidar que los ejemplos con los que trabajemos no tengan elementos repetidos. La función `(nub xs)` de la librería `Data.List` elimina los elementos repetidos de una lista.

Los conjuntos pueden definirse de manera explícita, citando todos sus elementos entre llaves, de manera implícita, dando una o varias características que determinen si

un objeto dado está o no en el conjunto. Por ejemplo, los conjuntos  $\{1, 2, 3, 4\}$  y  $\{x \in \mathbb{N} \mid 1 \leq x \leq 4\}$  son el mismo, definido de forma explícita e implícita respectivamente.

*Nota 1.1.2.* La definición implícita es necesaria cuando el conjunto en cuestión tiene una cantidad infinita de elementos. En general, los conjuntos se notarán con letras mayúsculas:  $A, B, \dots$  y los elementos con letras minúsculas:  $a, b, \dots$ .

Cuando trabajamos con conjuntos concretos, siempre existe un contexto donde esos conjuntos existen. Por ejemplo, si  $A = \{-1, 1, 2, 3, 4, 5\}$  y  $B = \{x \mid x \in \mathbb{N} \text{ es par}\}$  el contexto donde podemos considerar  $A$  y  $B$  es el conjunto de los números enteros,  $\mathbb{Z}$ . En general, a este conjunto se le denomina *conjunto universal*. De una forma algo más precisa, podemos dar la siguiente definición:

**Definición 1.1.2.** El *conjunto universal*, que notaremos por  $U$ , es un conjunto del que son subconjuntos todos los posibles conjuntos que originan el problema que tratamos.

### 1.1.2. Pertenencia a un conjunto

Si el elemento  $a$  pertenece al conjunto  $A$ , escribiremos  $a \in A$ . En caso contrario escribiremos  $a \notin A$ .

La función (pertenece  $x$   $xs$ ) se verifica si  $x$  pertenece al conjunto  $xs$ . Por ejemplo,

```
9  'pertenece' [1..6] == False
   'c' 'pertenece' "Roca" == True
```

```
pertenece :: Eq a => a -> [a] -> Bool
pertenece = elem
```

### 1.1.3. Conjunto vacío

**Definición 1.1.3.** El conjunto que carece de elementos se denomina *conjunto vacío* y se denota por  $\emptyset$ .

La función conjuntoVacio devuelve el conjunto vacío y la función (esVacio  $xs$ ) se verifica si el conjunto  $xs$  es vacío. Por ejemplo,

```
esVacio [1..6] == False
esVacio [6..1] == True
esVacio conjuntoVacio == True
```

```
conjuntoVacio :: [a]
conjuntoVacio = []

esVacio :: [a] -> Bool
esVacio = null
```

### 1.1.4. Conjunto unitario

**Definición 1.1.4.** *Un conjunto con un único elemento se denomina **unitario**.*

*Nota 1.1.3.* Notemos que, si  $X = \{x\}$  es un conjunto unitario, debemos distinguir entre el conjunto  $X$  y el elemento  $x$ .

La función (`esUnitario xs`) se verifica si el conjunto `xs` es unitario.

```
esUnitario [5]      == True
esUnitario [5,3]    == False
esUnitario [5,5]    == True
```

```
esUnitario :: Eq a => [a] -> Bool
esUnitario xs = length (nub xs) == 1
```

En la definición de `esUnitario` se puede eliminar `nub` si se supone que trabajamos con conjuntos.

### 1.1.5. Subconjuntos

**Definición 1.1.5.** *Dados dos conjuntos  $A$  y  $B$ , si todo elemento de  $A$  es a su vez elemento de  $B$  diremos que  $A$  es un subconjunto de  $B$  y lo notaremos  $A \subseteq B$ . En caso contrario se notará  $A \not\subseteq B$ .*

La función (`esSubconjunto xs ys`) se verifica si `xs` es un subconjunto de `ys`.

```
[4,2]      'esSubconjunto' [3,2,4]      == True
[1,2]      'esSubconjunto' conjuntoVacio == False
conjuntoVacio 'esSubconjunto' [1,2]      == True
[4,2,1]    'esSubconjunto' [1,2,4]      == True
[3,2,4]    'esSubconjunto' [5,2]      == False
```

```
esSubconjunto :: Eq a => [a] -> [a] -> Bool
esSubconjunto xs ys = all ('pertenece' ys) xs
```

### 1.1.6. Igualdad de conjuntos

**Definición 1.1.6.** *Dados dos conjuntos  $A$  y  $B$ , diremos que son **iguales** si tienen los mismos elementos; es decir, si se verifica que  $A \subseteq B$  y  $B \subseteq A$ . Lo notaremos  $A = B$ .*

La función (`conjuntosIguales xs ys`) se verifica si los conjuntos `xs` y `ys` son iguales. Por ejemplo,

```
conjuntosIguales [4,2] [3,2,4] == True
conjuntosIguales [5,2] [3,2,4] == False
```

```
conjuntosIguales :: Eq a => [a] -> [a] -> Bool
conjuntosIguales xs ys =
    esSubconjunto xs ys && esSubconjunto ys xs
```

### 1.1.7. Subconjuntos propios

**Definición 1.1.7.** Los subconjuntos de  $A$  distintos del  $\emptyset$  y del mismo  $A$  se denominan *subconjuntos propios* de  $A$ .

La función (`esSubconjuntoPropio xs ys`) se verifica si  $xs$  es un subconjunto propio de  $ys$ . Por ejemplo,

```
[4,2] 'esSubconjuntoPropio' [3,2,4] == True
[4,2,1] 'esSubconjuntoPropio' [1,2,4] == False
```

```
esSubconjuntoPropio :: Eq a => [a] -> [a] -> Bool
esSubconjuntoPropio xs ys =
    xs 'esSubconjunto' ys && not (conjuntosIguales xs ys)
```

### 1.1.8. Complementario de un conjunto

**Definición 1.1.8.** Dado un conjunto  $A$ , se define el *complementario* de  $A$ , que notaremos por  $\overline{A}$  como:

$$\overline{A} = \{x | x \in U, x \notin A\}$$

La función (`unionConjuntos xs ys`) devuelve la unión de los conjuntos  $xs$  y  $ys$ . Por ejemplo,

```
complementario [1..9] [3,2,5,7] == [1,4,6,8,9]
```

```
complementario :: Eq a => [a] -> [a] -> [a]
complementario = (\\)
```

### 1.1.9. Cardinal de un conjunto

**Definición 1.1.9.** Dado un conjunto finito  $A$ , denominaremos *cardinal* de  $A$  al número de elementos que tiene y lo notaremos  $|A|$ .

La función `(cardinal xs)` devuelve el cardinal del conjunto `xs`. Por ejemplo,

```
cardinal conjuntoVacio == 0
cardinal [1..10]       == 10
```

```
cardinal :: Eq a => [a] -> Int
cardinal = length
```

### 1.1.10. Unión de conjuntos

**Definición 1.1.10.** Dados dos conjuntos  $A$  y  $B$  se define la *unión* de  $A$  y  $B$ , notado  $A \cup B$ , como el conjunto formado por aquellos elementos que pertenecen al menos a uno de los dos conjuntos,  $A$  ó  $B$ ; es decir,

$$A \cup B = \{x | x \in A \vee x \in B\}$$

La función `(unionConjuntos xs ys)` devuelve la unión de los conjuntos `xs` y `ys`. Por ejemplo,

```
unionConjuntos [1,3..9] [2,4..9] == [1,3,5,7,9,2,4,6,8]
unionConjuntos "centri" "fugado" == "centrifugado"
```

```
unionConjuntos :: Eq a => [a] -> [a] -> [a]
unionConjuntos = union
```

*Nota 1.1.4.* Para ahorrar en escritura, en el futuro utilizaremos la función `(union xs ys)` definida en el módulo `Data.List`, equivalente a `(unionConjuntos xs ys)`

### 1.1.11. Intersección de conjuntos

**Definición 1.1.11.** Dados dos conjuntos  $A$  y  $B$  se define la *intersección* de  $A$  y  $B$ , notado  $A \cap B$ , como el conjunto formado por aquellos elementos que pertenecen a cada uno de los dos conjuntos,  $A$  y  $B$ , es decir,

$$A \cap B = \{x | x \in A \wedge x \in B\}$$

La función `(interseccion xs ys)` devuelve la intersección de los conjuntos `xs` y `ys`. Por ejemplo,

```

interseccion [1,3..20] [2,4..20] == []
interseccion [2,4..30] [4,8..30] == [4,8,12,16,20,24,28]
interseccion "noche" "dia"      == ""

```

```

interseccion :: Eq a => [a] -> [a] -> [a]
interseccion = intersect

```

### 1.1.12. Producto cartesiano

**Definición 1.1.12.** El *producto cartesiano*<sup>1</sup> de dos conjuntos  $A$  y  $B$  es una operación sobre ellos que resulta en un nuevo conjunto  $A \times B$  que contiene a todos los pares ordenados tales que la primera componente pertenece a  $A$  y la segunda pertenece a  $B$ ; es decir,  $A \times B = \{(a,b) | a \in A, b \in B\}$ .

La función (productoCartesiano xs ys) devuelve el producto cartesiano de xs e ys. Por ejemplo,

```

productoCartesiano [3,1] [2,4,7] == [(3,2),(3,4),(3,7),(1,2),(1,4),(1,7)]

```

```

productoCartesiano :: [a] -> [b] -> [(a,b)]
productoCartesiano xs ys =
  [(x,y) | x <- xs, y <- ys]

```

### 1.1.13. Combinaciones

**Definición 1.1.13.** Las *combinaciones* de un conjunto  $S$  tomados en grupos de  $n$  son todos los subconjuntos de  $S$  con  $n$  elementos.

La función (combinaciones n xs) devuelve las combinaciones de los elementos de xs en listas de n elementos. Por ejemplo,

```

combinaciones 3 ['a'..'d'] == ["abc","abd","acd","bcd"]
combinaciones 2 [2,4..8]   == [[2,4],[2,6],[2,8],[4,6],[4,8],[6,8]]

```

```

combinaciones :: Integer -> [a] -> [[a]]
combinaciones 0 _          = [[]]
combinaciones _ []         = []
combinaciones k (x:xs) =
  [x:ys | ys <- combinaciones (k-1) xs] ++ combinaciones k xs

```

<sup>1</sup>[https://en.wikipedia.org/wiki/Cartesian\\_product](https://en.wikipedia.org/wiki/Cartesian_product)

### 1.1.14. Variaciones con repetición

**Definición 1.1.14.** Las *variaciones con repetición* de  $m$  elementos tomados en grupos de  $n$  es el número de diferentes  $n$ -tuplas de un conjunto de  $m$  elementos.

La función (`variacionesR n xs`) devuelve las variaciones con con repetición de los elementos de `xs` en listas de  $n$  elementos. Por ejemplo,

```
ghci> variacionesR 3 ['a','b']
["aaa","aab","aba","abb","baa","bab","bba","bbb"]
ghci> variacionesR 2 [2,4..8]
[[2,2],[2,4],[2,6],[2,8],[4,2],[4,4],[4,6],[4,8],
 [6,2],[6,4],[6,6],[6,8],[8,2],[8,4],[8,6],[8,8]]
```

```
variacionesR :: Int -> [a] -> [[a]]
variacionesR _ [] = [[]]
variacionesR 0 _ = [[]]
variacionesR k us =
  [u:vs | u <- us, vs <- variacionesR (k-1) us]
```

## 1.2. Relaciones

Las relaciones que existen entre personas, números, conjuntos y muchas otras entidades pueden formalizarse en la idea de relación binaria. En esta sección se define y desarrolla este concepto además

### 1.2.1. Relación binaria

**Definición 1.2.1.** Una *relación binaria*<sup>2</sup> (o *correspondencia*) entre dos conjuntos  $A$  y  $B$  es un subconjunto del producto cartesiano  $A \times B$ .

La función (`esRelacion xs ys r`) se verifica si  $r$  es una relación binaria de `xs` en `ys`. Por ejemplo,

```
esRelacion [3,1] [2,4,7] [(3,4),(1,2)] == True
esRelacion [3,1] [2,4,7] [(3,1),(1,2)] == False
```

```
esRelacion :: (Eq a, Eq b) => [a] -> [b] -> [(a,b)] -> Bool
esRelacion xs ys r =
  productoCartesiano xs ys 'esSubconjunto' r
```

<sup>2</sup>[https://en.wikipedia.org/wiki/Binary\\_relation](https://en.wikipedia.org/wiki/Binary_relation)



### 1.2.2. Imagen por una relación

**Definición 1.2.2.** Si  $R$  es una relación binaria, la **imagen del elemento**  $x$  en la relación  $R$  es el conjunto de los valores correspondientes a  $x$  en  $R$ .

La función (`imagenRelacion r x`) es la imagen de  $x$  en la relación  $r$ . Por ejemplo,

```
imagenRelacion [(1,3),(2,5),(1,4)] 1 == [3,4]
imagenRelacion [(1,3),(2,5),(1,4)] 2 == [5]
imagenRelacion [(1,3),(2,5),(1,4)] 3 == []
```

```
imagenRelacion :: (Eq a, Eq b) => [(a,b)] -> a -> [b]
imagenRelacion r x =
  nub [y | (z,y) <- r, z == x]
```

### 1.2.3. Dominio de una relación

**Definición 1.2.3.** Dada una relación binaria  $R$ , su **dominio** es el conjunto que contiene a todos los valores que se toman en la relación  $R$ .

La función (`dominio r`) devuelve el dominio de la relación  $r$ . Por ejemplo,

```
dominio [(3,2),(5,1),(3,4)] == [3,5]
```

```
dominio :: Eq a => [(a,b)] -> [a]
dominio r = nub (map fst r)
```

### 1.2.4. Rango de una relación

**Definición 1.2.4.** El **rango** de una relación binaria  $R$  es el conjunto de las imágenes de mediante  $R$ .

La función (`rango r`) devuelve el rango de la relación binaria  $r$ . Por ejemplo,

```
rango [(3,2),(5,2),(3,4)] == [2,4]
```

```
rango :: Eq b => [(a,b)] -> [b]
rango r = nub (map snd r)
```

### 1.2.5. Antiimagen por una relación

**Definición 1.2.5.** La *antiimagen del elemento  $y$  por una relación  $r$*  es el conjunto de los elementos cuya imagen es  $y$ .

La `(antiImagenRelacion r y)` es la antiimagen del elemento  $y$  en la relación binaria  $r$ . Por ejemplo.

```
antiImagenRelacion [(1,3),(2,3),(7,4)] 3 == [1,2]
```

```
antiImagenRelacion :: (Eq a, Eq b) => [(a,b)] -> b -> [a]
antiImagenRelacion r y =
  nub [x | (x,z) <- r, z == y]
```

### 1.2.6. Relación funcional

**Definición 1.2.6.** Dada una relación binaria  $R$ , se dice *funcional* si todos los elementos de su dominio tienen una única imagen en  $R$ .

La función `(esFuncional r)` se verifica si la relación  $r$  es funcional. Por ejemplo,

```
esFuncional [(3,2),(5,1),(7,9)] == True
esFuncional [(3,2),(5,1),(3,4)] == False
esFuncional [(3,2),(5,1),(3,2)] == True
```

```
esFuncional :: (Eq a, Eq b) => [(a,b)] -> Bool
esFuncional r =
  and [esUnitario (imagenRelacion r x) | x <- dominio r]
```

## 1.3. Relaciones homogéneas

La sección de relaciones homogéneas no debe de depender de grafo. Por tanto, hay que eliminar sus importaciones.

Para elaborar la presente sección, se han consultado los [apuntes de Álgebra Básica](#),<sup>3</sup> asignatura del primer curso del Grado en Matemáticas.

Añadirlo al fichero de bibliografía y poner la cita.

<sup>3</sup><https://rodas5.us.es/file/a774213d-a15a-41df-816c-e633fb1a5876/1/01-Conjuntos.pdf>

**Definición 1.3.1.** Una relación binaria entre dos conjuntos  $A$  y  $B$  se dice que es **homogénea** si los conjuntos son iguales; es decir, si  $A = B$ . Si el par  $(x,y) \in AA$  está en la relación homogénea  $R$ , diremos que  $x$  está  $R$ -relacionado con  $y$ , o relacionado con  $y$  por  $R$ . Esto se notará frecuentemente  $xRy$  (nótese que el orden es importante).

La función (`esRelacionHomogenea xs r`) se verifica si  $r$  es una relación binaria homogénea en el conjunto  $xs$ .

```
esRelacionHomogenea [1..4] [(1,2),(2,4),(3,4),(4,1)] == True
esRelacionHomogenea [1..4] [(1,2),(2,5),(3,4),(4,1)] == False
esRelacionHomogenea [1..4] [(1,2),(3,4),(4,1)]         == True
```

```
esRelacionHomogenea :: Eq a => [a] -> [(a,a)] -> Bool
esRelacionHomogenea xs = esRelacion xs xs
```

*Nota 1.3.1.* El segundo argumento que recibe la función ha de ser una lista de pares con ambas componentes del mismo tipo.

La función (`estaRelacionado r x y`) se verifica si  $x$  está relacionado con  $y$  en la relación homogénea  $r$ . Por ejemplo,

```
estaRelacionado [(1,3),(2,5),(4,6)] 2 5 == True
estaRelacionado [(1,3),(2,5),(4,6)] 2 3 == False
```

```
estaRelacionado :: Eq a => [(a,a)] -> a -> a -> Bool
estaRelacionado r x y = (x,y) 'elem' r
```

### 1.3.1. Relaciones reflexivas

**Definición 1.3.2.** Sea  $R$  una relación binaria homogénea en el conjunto  $A$ . Diremos que  $R$  es **reflexiva** cuando todos los elementos de  $A$  están relacionados por  $R$  consigo mismos; es decir, cuando  $\forall x \in A$  se tiene que  $xRx$ .

La función (`esReflexiva xs r`) se verifica si la relación  $r$  en  $xs$  es reflexiva. Por ejemplo,

```
ghci> let n = 50
ghci> let r = [(x,y) | x <- [1..n] , y <- [x..n]]
ghci> esReflexiva [1..n] r
True
ghci> let r = [(x,y) | x <- [1..n], y <- [1..n], even (x-y)]
ghci> esReflexiva [1..n] r
True
```

```
ghci> let r = [(x,y) | x <- [1..n], y <- [1..n], mod x y == 0]
ghci> esReflexiva [1..n] r
True
ghci> let r = [(x,y) | x <- [1..n], y <- [x+1..n]]
ghci> esReflexiva [1..n] r
False
```

```
esReflexiva :: Eq a => [a] -> [(a,a)] -> Bool
esReflexiva xs r = zip xs xs 'esSubconjunto' r
```

*Nota 1.3.2.* En el conjunto  $\mathbb{N}$ , las relaciones caracterizadas por:

- $xRy \iff x \leq y$ ,
- $xSy \iff x - y$  es par,
- $xTy \iff x$  divide a  $y$ ,

son relaciones binarias homogéneas reflexivas.

### 1.3.2. Relaciones simétricas

**Definición 1.3.3.** Diremos que una relación homogénea  $R$  es **simétrica** cuando

$$\forall (x,y) \in R \longrightarrow (y,x) \in R$$

La función (`esSimetrica r`) se verifica si la relación  $r$  es simétrica. Por ejemplo,

```
ghci> let n= 50
ghci> let r = [(x,y) | x <- [1..n], y <- [x..n]]
ghci> esSimetrica r
False
ghci> let r = [(x,y) | x <- [1..n], y <- [1..n], even (x-y)]
ghci> esSimetrica r
True
ghci> let r = [(x,y) | x <- [1..n], y <- [1..n], mod x y == 0]
ghci> esSimetrica r
False
ghci> let r = [(x,y) | x <- [1..n], y <- [x+1..n]]
ghci> esSimetrica r
False
```

```
esSimetrica :: Eq a => [(a,a)] -> Bool
esSimetrica r = [(y,x) | (x,y) <- r] 'esSubconjunto' r
```

*Nota 1.3.3.* En el conjunto  $\mathbb{N}$ , la relación caracterizada por  $xSy \iff x - y$  es par, es una relación binaria homogénea simétrica.

### 1.3.3. Relaciones antisimétricas

**Definición 1.3.4.** Sea  $R$  una relación binaria homogénea en el conjunto  $A$ . Diremos que  $R$  es *antisimétrica* cuando

$$\forall (x,y)[(x,y) \in R \wedge (y,x) \in R \longrightarrow x = y]$$

La función (`esAntisimetrica r`) se verifica si la relación  $r$  es antisimétrica. Por ejemplo,

```
ghci> let n= 50
ghci> let r = [(x,y) | x <- [1..n] , y <- [x..n]]
ghci> esAntisimetrica r
True
ghci> let r = [(x,y) | x <- [1..n], y <- [1..n], even (x-y)]
ghci> esAntisimetrica r
False
ghci> let r = [(x,y) | x <- [1..n], y <- [1..n], mod x y == 0]
ghci> esAntisimetrica r
True
ghci> let r = [(x,y) | x <- [1..n] , y <- [x+1..n]]
ghci> esAntisimetrica [1..n] r
True
```

```
esAntisimetrica :: Eq a => [(a,a)] -> Bool
esAntisimetrica r =
  and [x == y | (x,y) <- r, (y,x) `elem` r]
```

*Nota 1.3.4.* En el conjunto  $\mathbb{N}$ , las relaciones caracterizadas por:

- $xRy \longleftrightarrow x \leq y$ ,
- $xTy \longleftrightarrow x$  divide a  $y$ ,
- $xRy \longleftrightarrow x < y$ ,

son relaciones binarias homogéneas antisimétricas.

### 1.3.4. Relaciones transitivas

**Definición 1.3.5.** Sea  $R$  una relación binaria homogénea en el conjunto  $A$ . Diremos que  $R$  es *transitiva* cuando  $\forall (x,y), (y,z) \in R$  se tiene que  $xRy$  e  $yRz \longrightarrow xRz$ .

La función (`esTransitiva r`) se verifica si la relación  $r$  es transitiva. Por ejemplo,

```
ghci> let n= 50
ghci> let r = [(x,y) | x <- [1..n] , y <- [x..n]]
ghci> esTransitiva [1..n] r
```

```
True
ghci> let r = [(x,y) | x <- [1..n], y <- [1..n], even (x-y)]
ghci> esTransitiva [1..n] r
True
ghci> let r = [(x,y) | x <- [1..n], y <- [1..n], mod x y == 0]
ghci> esTransitiva [1..n] r
True
ghci> let r = [(x,y) | x <- [1..n] , y <- [x+1..n]]
ghci> esTransitiva [1..n] r
True
```

```
esTransitiva :: Eq a => [(a,a)] -> Bool
esTransitiva r =
  [(x,z) | (x,y) <- r, (w,z) <- r, y == w] 'esSubconjunto' r
```

*Nota 1.3.5.* En el conjunto  $\mathbb{N}$ , las relaciones caracterizadas por:

- $xRy \longleftrightarrow x \leq y$ ,
- $xSy \longleftrightarrow x - y$  es par,
- $xTy \longleftrightarrow x$  divide a  $y$ ,
- $xRy \longleftrightarrow x < y$ ,

son relaciones binarias homogéneas transitivas.

### 1.3.5. Relaciones de equivalencia

**Definición 1.3.6.** Las relaciones homogéneas que son a la vez reflexivas, simétricas y transitivas se denominan *relaciones de equivalencia*.

La función (`esRelacionEquivalencia xs r`) se verifica si  $r$  es una relación de equivalencia en  $xs$ . Por ejemplo,

```
ghci> let n= 50
ghci> let r = [(x,y) | x <- [1..n] , y <- [x..n]]
ghci> esRelacionEquivalencia [1..n] r
False
ghci> let r = [(x,y) | x <- [1..n], y <- [1..n], even (x-y)]
ghci> esRelacionEquivalencia [1..n] r
True
ghci> let r = [(x,y) | x <- [1..n], y <- [1..n], mod x y == 0]
ghci> esRelacionEquivalencia [1..n] r
False
ghci> let r = [(x,y) | x <- [1..n] , y <- [x+1..n]]
ghci> esRelacionEquivalencia [1..n] r
False
```

```

esRelacionEquivalencia :: Eq a => [a] -> [(a,a)] -> Bool
esRelacionEquivalencia xs r =
    esReflexiva xs r    &&
    esSimetrica r       &&
    esTransitiva r

```

*Nota 1.3.6.* En el conjunto  $\mathbb{N}$ , la relación caracterizada por  $xSy \iff x - y$  es par, es una relación de equivalencia.

### 1.3.6. Relaciones de orden

**Definición 1.3.7.** Las relaciones homogéneas que son a la vez reflexivas, antisimétricas y transitivas se denominan *relaciones de orden*.

La función (`esRelacionOrden xs r`) se verifica si  $r$  es una relación de orden en  $xs$ . Por ejemplo,

```

ghci> let n= 50
ghci> let r = [(x,y) | x <- [1..n] , y <- [x..n]]
ghci> esRelacionOrden [1..n] r
True
ghci> let r = [(x,y) | x <- [1..n], y <- [1..n], even (x-y)]
ghci> esRelacionOrden [1..n] r
False
ghci> let r = [(x,y) | x <- [1..n], y <- [1..n], mod x y == 0]
ghci> esRelacionOrden [1..n] r
True
ghci> let r = [(x,y) | x <- [1..n] , y <- [x+1..n]]
ghci> esRelacionOrden [1..n] r
False

```

```

esRelacionOrden :: Eq a => [a] -> [(a,a)] -> Bool
esRelacionOrden xs r =
    esReflexiva xs r    &&
    esAntisimetrica r   &&
    esTransitiva r

```

*Nota 1.3.7.* En el conjunto  $\mathbb{N}$ , las relaciones caracterizadas por:

- $xRy \iff x \leq y$ ,
  - $xTy \iff x$  divide a  $y$ ,
- son relaciones de orden.

### 1.3.7. Clases de equivalencia

**Definición 1.3.8.** Si  $R$  es una relación de equivalencia en  $A$ , denominamos *clase de equivalencia* de un elemento  $x \in A$  al conjunto de todos los elementos de  $A$  relacionados con  $x$ ; es decir,  $\bar{x} = R(x) = \{y \in A \mid xRy\}$  donde la primera notación se usa si la relación con la que se está tratando se sobreentiende, y la segunda si no es así.

La función (`clasesEquivalencia xs r`) devuelve las clases de la relación de equivalencia  $r$  en  $xs$ . Por ejemplo,

```
ghci> let n = 50
ghci> let r = [(x,y) | x <- [1..n], y <- [1..n], even (x-y)]
ghci> clasesEquivalencia [1..n] r
[[1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39,41,43,45,47,49],
 [2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,46,48,50]]
ghci> let m = 5
ghci> let r = [(x,y) | x <- [1..n], y <- [1..n], mod x m == mod y m]
ghci> clasesEquivalencia [1..n] r
[[1,6,11,16,21,26,31,36,41,46], [2,7,12,17,22,27,32,37,42,47],
 [3,8,13,18,23,28,33,38,43,48], [4,9,14,19,24,29,34,39,44,49],
 [5,10,15,20,25,30,35,40,45,50]]
```

```
clasesEquivalencia :: Eq a => [a] -> [(a,a)] -> [[a]]
clasesEquivalencia _ [] = []
clasesEquivalencia [] _ = []
clasesEquivalencia (x:xs) r = (x:c) : clasesEquivalencia (xs \ c) r
  where c = filter (estaRelacionado r x) xs
```

## 1.4. Funciones

**Definición 1.4.1.** Dada una relación  $F$  entre  $A$  y  $B$ , se dirá que es una *función* si es una relación binaria, es funcional y todos los elementos de  $A$  están en el dominio.

La función (`esFuncion xs ys f`) se verifica si  $f$  es una función de  $xs$  en  $ys$ . Por ejemplo,

```
esFuncion [3,1] [2,4,7] [(1,7),(3,2)] == True
esFuncion [3,1] [2,4,7] [(1,7)]      == False
esFuncion [3,1] [2,4,7] [(1,7),(3,2),(1,4)] == False
```

```
esFuncion :: (Eq a, Eq b) => [a] -> [b] -> [(a,b)] -> Bool
esFuncion xs ys f =
  esRelacion xs ys f &&
  xs 'esSubconjunto' dominio f &&
  esFuncional f
```



*Nota 1.4.1.* A lo largo de la sección representaremos a las funciones como listas de pares.

```
type Funcion a b = [(a,b)]
```

La función (`funciones xs ys`) devuelve todas las posibles funciones del conjunto `xs` en `ys`. Por ejemplo,

```
ghci> funciones [1,2,3] "ab"
[[ (1,'a'), (2,'a'), (3,'a') ], [ (1,'a'), (2,'a'), (3,'b') ],
 [ (1,'a'), (2,'b'), (3,'a') ], [ (1,'a'), (2,'b'), (3,'b') ],
 [ (1,'b'), (2,'a'), (3,'a') ], [ (1,'b'), (2,'a'), (3,'b') ],
 [ (1,'b'), (2,'b'), (3,'a') ], [ (1,'b'), (2,'b'), (3,'b') ]]
ghci> funciones [(1,2),(1,5)] "abc"
[[ ((1,2),'a'), ((1,5),'a') ], [ ((1,2),'a'), ((1,5),'b') ],
 [ ((1,2),'a'), ((1,5),'c') ], [ ((1,2),'b'), ((1,5),'a') ],
 [ ((1,2),'b'), ((1,5),'b') ], [ ((1,2),'b'), ((1,5),'c') ],
 [ ((1,2),'c'), ((1,5),'a') ], [ ((1,2),'c'), ((1,5),'b') ],
 [ ((1,2),'c'), ((1,5),'c') ]]
```

```
funciones :: [a] -> [b] -> [Funcion a b]
funciones xs ys =
  [zip xs zs | zs <- variacionesR (length xs) ys]
```

### 1.4.1. Imagen por una función

**Definición 1.4.2.** Si  $f$  es una función entre  $A$  y  $B$  y  $x$  es un elemento del conjunto  $A$ , la *imagen del elemento  $x$  por la función  $f$*  es el valor asociado a  $x$  por la función  $f$ .

La función (`imagen f x`) es la imagen del elemento  $x$  en la función  $f$ . Por ejemplo,

```
imagen [(1,7),(3,2)] 1 == 7
imagen [(1,7),(3,2)] 3 == 2
```

```
imagen :: (Eq a, Eq b) => Funcion a b -> a -> b
imagen f x = head (imagenRelacion f x)
```

### 1.4.2. Funciones inyectivas

**Definición 1.4.3.** Diremos que una función  $f$  entre dos conjuntos es **inyectiva**<sup>4</sup> si a elementos distintos del dominio le corresponden elementos distintos de la imagen; es decir, si  $\forall a, b \in \text{dominio}(f)$  tales que  $a \neq b$ ,  $f(a) \neq f(b)$ .

La función `(esInyectiva fs)` se verifica si la función `fs` es inyectiva. Por ejemplo,

```
esInyectiva [(1,4),(2,5),(3,6)] == True
esInyectiva [(1,4),(2,5),(3,4)] == False
esInyectiva [(1,4),(2,5),(3,6),(3,6)] == True
```

```
esInyectiva :: (Eq a, Eq b) => Funcion a b -> Bool
esInyectiva f =
  all esUnitario [antiImagenRelacion f y | y <- rango f]
```

### 1.4.3. Funciones sobreyectivas

**Definición 1.4.4.** Diremos que una función  $f$  entre dos conjuntos  $A$  y  $B$  es **sobreyectiva**<sup>5</sup> si todos los elementos de  $B$  son imagen de algún elemento de  $A$ .

La función `(esSobreyectiva xs ys f)` se verifica si la función `f` es sobreyectiva. A la hora de definirla, estamos contando con que `f` es una función entre `xs` y `ys`. Por ejemplo,

```
esSobreyectiva [1,2,3] [4,5,6] [(1,4),(2,5),(3,6)] == True
esSobreyectiva [1,2,3] [4,5,6] [(1,4),(2,5),(3,4)] == False
esSobreyectiva [1,2,3] [4,5,6] [(1,4),(2,4),(3,6),(3,6)] == False
```

```
esSobreyectiva :: (Eq a, Eq b) => [a] -> [b] -> Funcion a b -> Bool
esSobreyectiva _ ys f = ys `esSubconjunto` rango f
```

### 1.4.4. Funciones biyectivas

**Definición 1.4.5.** Diremos que una función  $f$  entre dos conjuntos  $A$  y  $B$  es **biyectiva**<sup>6</sup> si cada elementos de  $B$  es imagen de un único elemento de  $A$ .

La función `(esBiyectiva xs ys f)` se verifica si la función `f` es biyectiva. Por ejemplo,

<sup>4</sup>[https://en.wikipedia.org/wiki/Injective\\_function](https://en.wikipedia.org/wiki/Injective_function)

<sup>5</sup>[https://en.wikipedia.org/wiki/Surjective\\_function](https://en.wikipedia.org/wiki/Surjective_function)

<sup>6</sup>[https://en.wikipedia.org/wiki/Bijective\\_function](https://en.wikipedia.org/wiki/Bijective_function)

```
esBiyectiva [1,2,3] [4,5,6] [(1,4),(2,5),(3,6),(3,6)] == True
esBiyectiva [1,2,3] [4,5,6] [(1,4),(2,5),(3,4)]      == False
esBiyectiva [1,2,3] [4,5,6,7] [(1,4),(2,5),(3,6)]    == False
```

```
esBiyectiva :: (Eq a, Eq b) => [a] -> [b] -> Funcion a b -> Bool
esBiyectiva xs ys f =
  esInyectiva f && esSobreyectiva xs ys f
```

Las funciones `biyecciones1 xs ys` y `biyecciones2 xs ys` devuelven la lista de todas las biyecciones entre los conjuntos `xs` y `ys`. La primera lo hace filtrando las funciones entre los conjuntos que son biyectivas y la segunda lo hace construyendo únicamente las funciones biyectivas entre los conjuntos, con el consecuente ahorro computacional.

```
ghci> length (biyecciones1 [1..7] ['a'..'g'])
5040
(16.75 secs, 4,146,744,104 bytes)
ghci> length (biyecciones2 [1..7] ['a'..'g'])
5040
(0.02 secs, 0 bytes)
ghci> length (biyecciones1 [1..6] ['a'..'g'])
0
(2.53 secs, 592,625,824 bytes)
ghci> length (biyecciones2 [1..6] ['a'..'g'])
0
(0.01 secs, 0 bytes)
```

```
biyecciones1 :: (Eq a, Eq b) => [a] -> [b] -> [Funcion a b]
biyecciones1 xs ys =
  filter (esBiyectiva xs ys) (funciones xs ys)

biyecciones2 :: (Eq a, Eq b) => [a] -> [b] -> [Funcion a b]
biyecciones2 xs ys
  | length xs /= length ys = []
  | otherwise               = [zip xs zs | zs <- permutations ys]
```

*Nota 1.4.2.* En lo que sigue trabajaremos con la función `biyecciones2` así que la definiremos como `biyecciones`.

```
biyecciones :: (Eq a, Eq b) => [a] -> [b] -> [Funcion a b]
biyecciones = biyecciones2
```

### 1.4.5. Inversa de una función

**Definición 1.4.6.** Si  $f$  es una función biyectiva entre los conjuntos  $A$  y  $B$ , definimos la **función inversa**<sup>7</sup> como la función que a cada elemento de  $B$  le hace corresponder el elemento de  $A$  del que es imagen en  $B$ .

El valor de `(inversa f)` es la función inversa de  $f$ . Por ejemplo,

```
ghci> inversa [(1,4),(2,5),(3,6)]
[(4,1),(5,2),(6,3)]
ghci> inversa [(1,4),(2,4),(3,6),(3,6)]
[(4,1),(4,2),(6,3)]
```

```
inversa :: (Eq a, Eq b) => Funcion a b -> Funcion b a
inversa f = [(y,x) | (x,y) <- nub f]
```

En la definición de `inversa` se puede eliminar `nub`, suponiendo que  $f$  es un conjunto.

*Nota 1.4.3.* Para considerar la inversa de una función, esta tiene que ser biyectiva. Luego `(inversa f)` asigna a cada elemento del conjunto imagen (que en este caso coincide con la imagen) uno y solo uno del conjunto de salida.

La función `(imagenInversa f y)` devuelve el elemento del conjunto de salida de la función  $f$  tal que su imagen es  $y$ .

```
imagenInversa [(1,4),(2,5),(3,6)] 5 == 2
imagenInversa [(1,4),(2,4),(3,6),(3,6)] 6 == 3
```

```
imagenInversa :: (Eq a, Eq b) => Funcion a b -> b -> a
imagenInversa f = imagen (inversa f)
```

<sup>7</sup>[https://en.wikipedia.org/wiki/Inverse\\_function](https://en.wikipedia.org/wiki/Inverse_function)

## Capítulo 2

# Introducción a la teoría de grafos

Se dice que la Teoría de Grafos tiene su origen en 1736, cuando Euler dio una solución al problema (hasta entonces no resuelto) de los siete puentes de Königsberg: ¿existe un camino que atravesase cada uno de los puentes exactamente una vez?

Para probar que no era posible, Euler sustituyó cada región por un nodo y cada puente por una arista, creando el primer grafo que fuera modelo de un problema matemático.



Figura 2.1: Dibujo de los puentes de Königsberg

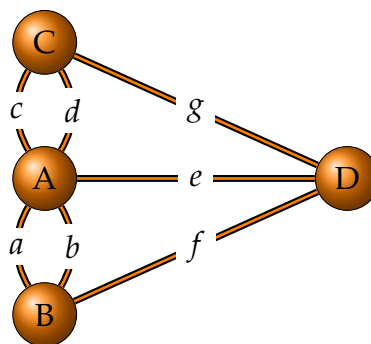


Figura 2.2: Modelo de los puentes de Königsberg

Desde entonces, se ha ido desarrollando esta metodología hasta convertirse en los últimos años en una herramienta importante en áreas del conocimiento muy variadas como, por ejemplo: la Investigación Operativa, la Computación, la Ingeniería Eléctrica, la Geografía y la Química. Es por ello que, además, se ha erigido como una nueva disciplina matemática, que generalmente asociada a las ramas de Topología y Álgebra.

La utilidad de los grafos se basa en su gran poder de abstracción y una representación muy clara de cualquier relación, lo que facilita enormemente tanto la fase de modelado como la de resolución de cualquier problema. Gracias a la Teoría de Grafos se han desarrollado una gran variedad de algoritmos y métodos de decisión que podemos implementar a través de lenguajes funcionales y permiten automatizar la resolución de muchos problemas, a menudo tediosos de resolver a mano.

Pendiente de ampliar la introducción conforme se vaya escribiendo los módulos.

## Contenido

1.1	Conjuntos	9
1.1.1	Primeras definiciones	10
1.1.2	Pertenencia a un conjunto	11
1.1.3	Conjunto vacío	11
1.1.4	Conjunto unitario	12
1.1.5	Subconjuntos	12
1.1.6	Igualdad de conjuntos	12
1.1.7	Subconjuntos propios	13
1.1.8	Complementario de un conjunto	13
1.1.9	Cardinal de un conjunto	14
1.1.10	Unión de conjuntos	14
1.1.11	Intersección de conjuntos	14
1.1.12	Producto cartesiano	15
1.1.13	Combinaciones	15
1.1.14	Variaciones con repetición	16
1.2	Relaciones	16
1.2.1	Relación binaria	16
1.2.2	Imagen por una relación	17
1.2.3	Dominio de una relación	17
1.2.4	Rango de una relación	17
1.2.5	Antiimagen por una relación	18
1.2.6	Relación funcional	18
1.3	Relaciones homogéneas	18
1.3.1	Relaciones reflexivas	19
1.3.2	Relaciones simétricas	20
1.3.3	Relaciones antisimétricas	21
1.3.4	Relaciones transitivas	21
1.3.5	Relaciones de equivalencia	22
1.3.6	Relaciones de orden	23
1.3.7	Clases de equivalencia	24

1.4	Funciones . . . . .	24
1.4.1	Imagen por una función . . . . .	25
1.4.2	Funciones inyectivas . . . . .	26
1.4.3	Funciones sobreyectivas . . . . .	26
1.4.4	Funciones biyectivas . . . . .	26
1.4.5	Inversa de una función . . . . .	28

## 2.1. Definición de grafo

En primer lugar, vamos a introducir terminología básica en el desarrollo de la Teoría de Grafos.

**Definición 2.1.1.** Un **grafo**  $G$  es un par  $(V, A)$ , donde  $V$  es el conjunto cuyos elementos llamamos **vértices** (o **nodos**) y  $A$  es un conjunto cuyos elementos llamamos **aristas**.

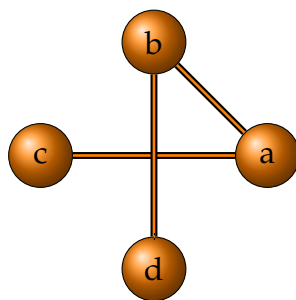
**Definición 2.1.2.** Una **arista** de un grafo  $G = (V, A)$ , es un conjunto de dos elementos de  $V$ . Es decir, para dos vértices  $v, v'$  de  $G$ ,  $(v, v')$  y  $(v', v)$  representa la misma arista.

**Definición 2.1.3.** Dado un grafo  $G = (V, A)$ , diremos que un vértice  $v \in V$  es **adyacente** a  $v' \in V$  si  $(v', v) \in A$ .

**Definición 2.1.4.** Si en un grafo dirigido se permiten aristas repetidas, lo llamaremos **multi-grafo**. Si no se permiten, lo llamaremos **grafo regular**.

**Nota 2.1.1.** Denotaremos por  $|V|$  al número de vértices y por  $|A|$  al número de aristas del grafo  $(V, A)$ .

**Ejemplo 2.1.2.** Sea  $G = (V, A)$  un grafo con  $V = \{a, b, c, d\}$  y  $A = \{(a, b), (a, c), (b, d), (d, d)\}$ . En este grafo, los vértices  $a, d$  son adyacentes a  $b$ .



## 2.2. El TAD de los grafos

En esta sección, nos planteamos la tarea de implementar las definiciones presentadas anteriormente en un lenguaje funcional. En nuestro caso, el lenguaje que utilizaremos será Haskell. Definiremos el Tipo Abstracto de Dato (TAD) de los grafos y daremos algunos ejemplos de posibles representaciones de grafos con las que podremos trabajar.

Si consideramos un grafo finito cualquiera  $G = (V, A)$ , podemos ordenar el conjunto de los vértices y representarlo como  $V = \{v_1, \dots, v_n\}$  con  $n = |V|$ .

En primer lugar, necesitaremos crear un tipo (Grafo) cuya definición sea compatible con la entidad matemática que representa y que nos permita definir las operaciones que necesitamos para trabajar con los grafos. Estas operaciones son:

```
creaGrafo  -- [a] -> [(a,a)] -> Grafo a
vertices   -- Grafo a -> [a]
adyacentes -- Grafo a -> a -> [a]
aristaEn   -- (a,a) -> Grafo a -> Bool
aristas    -- Grafo a -> [(a,a)]
```

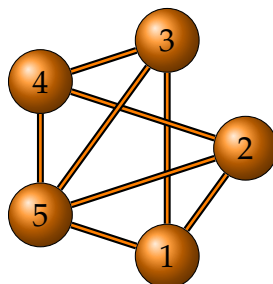
donde:

- `(creaGrafo vs as)` es un grafo tal que el conjunto de sus vértices es `vs` y el de sus aristas es `as`.
- `(vertices g)` es la lista de todos los vértices del grafo `g`.
- `(adyacentes g v)` es la lista de los vértices adyacentes al vértice `v` en el grafo `g`.
- `(aristaEn a g)` se verifica si `a` es una arista del grafo `g`.
- `(aristas g)` es la lista de las aristas del grafo `g`.

*Nota 2.2.1.* Las funciones que aparecen en la especificación del TAD no dependen de la representación que elijamos.

*Ejemplo 2.2.2.* Veamos un ejemplo de creación de grafo y su representación gráfica

```
creaGrafo [1..5] [(1,2),(1,3),(1,5),(2,4),
                  (2,5),(3,4),(3,5),(4,5)]
```





### 2.2.1. Grafos como listas de aristas

En el módulo `GrafoConListaDeAristas` se definen las funciones del TAD de los grafos dando su representación como conjuntos de aristas; es decir, representando a un grafo como dos conjuntos, la primera será la lista ordenada de los vértices y la segunda la lista ordenada de las aristas (en ambas listas se excluye la posibilidad de repeticiones).

*Nota 2.2.3.* Una diferencia entre vectores y listas es que en los vectores se tiene en tiempo constante el valor de índice  $n$  pero en las listas para encontrar el elemento  $n$ -ésimo hay que recorrerla. Los vectores tienen acceso constante ( $O(1)$ ) y las listas lineal ( $O(n)$ ).

Comentar las ventajas de usar listas en lugar de vectores.

```
module GrafoConListaDeAristas
  ( Grafo
  , creaGrafo  -- [a] -> [(a,a)] -> Grafo a
  , vertices   -- Grafo a -> [a]
  , adyacentes -- Grafo a -> a -> [a]
  , aristaEn   -- Grafo a -> (a,a) -> Bool
  , aristas    -- Grafo a -> [(a,a)]
  ) where
```

En las definiciones del presente módulo se usarán las funciones `nub` y `sort` de la librería `Data.List`

Vamos a definir un nuevo tipo de dato (`Grafo a`), que representará un grafo a partir de la lista de sus vértices (donde los vértices son de tipo `a`) y de aristas (que son pares de vértices).

```
data Grafo a = G [a] [(a,a)]
  deriving (Eq, Show)
```

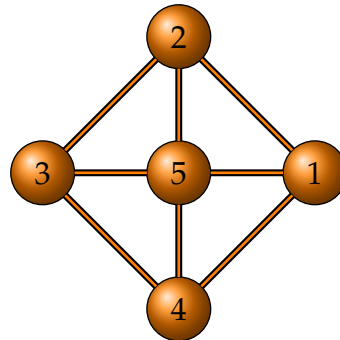
Las funciones básicas que definiremos a partir de este tipo coincidirán con las indicadas en el TAD de los grafos.

- (`creaGrafo vs as`) es el grafo cuyo conjunto de vértices es `cs` y el de sus aristas es `as`.

```
creaGrafo :: Ord a => [a] -> [(a,a)] -> Grafo a
creaGrafo vs as =
  G (sort vs) (nub (sort [parOrdenado a | a <- as]))

parOrdenado :: Ord a => (a,a) -> (a,a)
parOrdenado (x,y) | x <= y      = (x,y)
                  | otherwise   = (y,x)
```

Ejemplo 2.2.4. `ejGrafo` es el grafo



```
ghci> ejGrafo
G [1,2,3,4,5] [(1,2),(1,4),(1,5),(2,3),(2,5),(3,4),(3,5),(4,5)]
```

```
ejGrafo :: Grafo Int
ejGrafo = creaGrafo [1..5]
               [(1,2),(1,4),(1,5),(2,3),(2,5),(3,4),(3,5),(4,5)]
```

- `(vertices g)` es la lista de los vértices del grafo `g`. Por ejemplo,

```
vertices ejGrafo == [1,2,3,4,5]
```

```
vertices :: Grafo a -> [a]
vertices (G vs _) = vs
```

- `(adyacentes g v)` es la lista de los vértices adyacentes al vértice `v` en el grafo `g`. Por ejemplo,

```
adyacentes ejGrafo 4 == [2,3,5]
adyacentes ejGrafo 2 == [1,4,5]
```

```
adyacentes :: Eq a => Grafo a -> a -> [a]
adyacentes (G _ as) v =
  [u | (u,x) <- as, x == v] ++
  [u | (x,u) <- as, x == v]
```

- `(aristaEn g a)` se verifica si `a` es una arista del grafo `g`. Por ejemplo,

```
aristaEn ejGrafo (5,1) == True
aristaEn ejGrafo (3,1) == False
```

```
aristaEn :: Ord a => Grafo a -> (a,a) -> Bool
aristaEn (G _ as) a = parOrdenado a 'elem' as
```

- `(aristas g)` es la lista de las aristas del grafo `g`. Por ejemplo,

```
ghci> aristas ejGrafo
[(1,2),(1,4),(1,5),(2,3),(2,5),(3,4),(3,5),(4,5)]
```

```
aristas :: Grafo a -> [(a,a)]
aristas (G _ as) = as
```

## 2.3. Generadores de grafos

En esta sección, presentaremos el generador de grafos que nos permitirá generar grafos como listas de aristas arbitrariamente y usarlos como ejemplos o para comprobar propiedades.

Para aprender a controlar el tamaño de los grafos generados, he consultado las siguientes fuentes:

- [QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs](https://www.eecs.northwestern.edu/~robby/courses/395-495-2009-fall/quick.pdf)<sup>1</sup>
- [QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs](https://www.dcc.fc.up.pt/~pbv/aulas/tapf/slides/quickcheck.html)<sup>2</sup>

Añadir al fichero de bibliografía y poner la cita.

(generaGrafos n) es un generador de grafos de hasta n vértices. Por ejemplo,

```
ghci> sample (generaGrafo 5)
G [1,2] [(1,1),(1,2),(2,2)]
G [1,2,3,4] [(1,1),(1,3),(1,4),(2,3),(3,3)]
G [1,2,3,4] [(1,1),(1,2),(2,3),(2,4),(3,3),(3,4)]
G [1,2,3,4,5]
  [(1,1),(1,2),(1,3),(1,4),(1,5),(2,3),(2,5),(3,5),(5,5)]
G [1,2,3,4,5]
  [(1,1),(1,2),(1,3),(1,4),(1,5),(2,5),(3,3),(3,5),(4,4),(4,5)]
G [1,2] []
G [1,2,3] [(1,1),(2,3)]
G [1,2,3,4,5]
  [(1,1),(1,2),(1,5),(2,2),(2,3),(2,4),(3,5),(5,5)]
G [1,2,3,4,5] [(1,3),(2,5),(3,3),(3,5),(5,5)]
G [1,2,3,4,5] [(1,1),(1,3),(1,5),(2,3),(2,5),(3,4)]
G [1,2,3,4,5]
  [(1,1),(1,3),(1,4),(1,5),(2,5),(3,4),(3,5),(4,4),(4,5),(5,5)]

ghci> sample (generaGrafo 2)
G [1] []
G [1] []
G [1] [(1,1)]
G [1] []
G [1] [(1,1)]
G [1,2] [(1,1),(2,2)]
G [1] [(1,1)]
G [1] []
G [1,2] [(1,1)]
G [1] []
G [] []
```

<sup>1</sup><https://www.eecs.northwestern.edu/~robby/courses/395-495-2009-fall/quick.pdf>

<sup>2</sup><https://www.dcc.fc.up.pt/~pbv/aulas/tapf/slides/quickcheck.html>

```

generaGrafo :: Int -> Gen (Grafo Int)
generaGrafo s = do
  n <- choose (0,s)
  as <- sublistOf [(x,y) | x <- [1..n], y <- [x..n]]
  return (creaGrafo [1..n] as)

```

*Nota 2.3.1.* Los grafos están contenido en la clase de los objetos generables aleatoriamente.

```

instance Arbitrary (Grafo Int) where
  arbitrary = sized generaGrafo

```

En el siguiente ejemplo se pueden observar algunos grafos generados

```

ghci> sample (arbitrary :: Gen (Grafo Int))
G [] []
G [1,2] [(1,2)]
G [] []
G [1,2,3,4,5] [(1,3),(1,5),(2,3),(2,4),(3,4),(4,5),(5,5)]
G [1,2,3,4,5,6] [(1,2),(1,5),(3,3),(4,5),(4,6),(5,5),(5,6)]
G [1,2,3,4] [(1,1),(1,2),(1,3),(1,4),(2,3),(3,3)]
G [1,2,3,4,5,6,7,8,9,10,11,12,13,14] [(1,1),(1,3),(2,5),(3,4),(9,11)]

```

## 2.4. Ejemplos de grafos

El objetivo de esta sección es reunir una colección de grafos lo suficientemente extensa y variada como para poder utilizarla como recurso a la hora de comprobar las propiedades y definiciones de funciones que implementaremos más adelante.

En el proceso de recopilación de ejemplos, se ha trabajado con diversas fuentes:

- Relación de ejercicios Rel\_20 de la asignatura de Informática.
- Apuntes de MD.
- [Galería de grafos](https://es.wikipedia.org/wiki/Anexo:Galería_de_grafos)<sup>3</sup> de la Wikipedia.

Añadir al fichero de bibliografía y citarla.

Ir actualizando y completando las fuentes y cambiar el formato de enumeración

En este módulo hay un ejemplo de cómo dibujar un grafo cualquiera (grafo de la amistad)

*Nota 2.4.1.* Se utilizará la representación de los grafos como listas de aristas.

<sup>3</sup>[https://es.wikipedia.org/wiki/Anexo:Galería\\_de\\_grafos](https://es.wikipedia.org/wiki/Anexo:Galería_de_grafos)

**Definición 2.4.1.** *Un grafo nulo es un grafo que no tiene ni vértices ni aristas.*

La función (`grafoNulo`) devuelve un grafo nulo.

```
grafoNulo == G [] []
```

```
grafoNulo :: Ord a => Grafo a
grafoNulo = creaGrafo [] []
```

La función (`esGrafoNulo g`) se verifica si `g` es un grafo nulo

```
esGrafoNulo grafoNulo           == True
esGrafoNulo (creaGrafo [] [(1,2)]) == False
esGrafoNulo (creaGrafo [1,2] [(1,2)]) == False
```

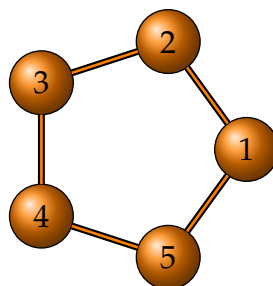
```
esGrafoNulo :: Grafo a -> Bool
esGrafoNulo g =
    null (vertices g) && null (aristas g)
```

### 2.4.1. Grafo ciclo

**Definición 2.4.2.** Un **ciclo**,<sup>4</sup> de orden  $n$ ,  $C(n)$ , es un grafo no dirigido y no ponderado cuyo conjunto de vértices viene dado por  $V = \{1, \dots, n\}$  y el de las aristas por  $A = \{(0,1), (1,2), \dots, (n-2, n-1), (n-1, 0)\}$

La función (`grafoCiclo n`) nos genera el ciclo de orden  $n$ . Por ejemplo,

```
ghci> grafoCiclo 5
G (array (1,5) [(1,[5,2]),(2,[1,3]),(3,[2,4]),(4,[3,5]),(5,[4,1])])
```



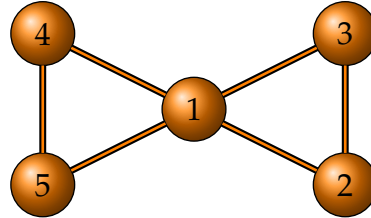
```
grafoCiclo :: Int -> Grafo Int
grafoCiclo 0 = grafoNulo
grafoCiclo 1 = creaGrafo [1] []
grafoCiclo n = creaGrafo [1..n]
    ([ (u,u+1) | u <- [1..n-1] ] ++ [(n,1)])
```

<sup>4</sup>[https://es.wikipedia.org/wiki/Grafo\\_completo](https://es.wikipedia.org/wiki/Grafo_completo)

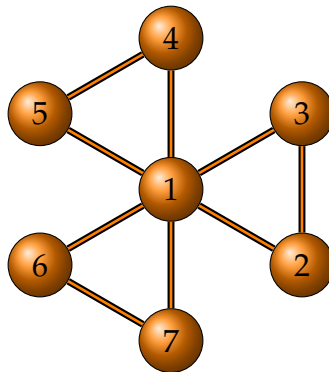
### 2.4.2. Grafo de la amistad

**Definición 2.4.3.** Un **grafo de la amistad**<sup>5</sup> de orden  $n$  es un grafo con  $2n + 1$  vértices y  $3n$  aristas formado uniendo  $n$  copias del ciclo  $C_3$  por un vértice común. Lo denotamos por  $F_n$ .

La función `(grafoAmistad n)` genera el grafo de la amistad de orden  $n$ . Por ejemplo,



```
ghci> grafoAmistad 2
G (array (1,5) [(1,[2,3,4,5]),(2,[1,3]),(3,[1,2]),(4,[1,5]),(5,[1,4])])
```



```
ghci> grafoAmistad 3
G (array (1,7) [(1,[2,3,4,5,6,7]),(2,[1,3]),(3,[1,2]),
               (4,[1,5]),(5,[1,4]),(6,[1,7]),(7,[1,6])])
```

```
grafoAmistad :: Int -> Grafo Int
grafoAmistad n =
  creaGrafo [1..2*n+1]
    ([ (1,a) | a <- [2..2*n+1] ] ++
     [ (a,b) | (a,b) <-zip [2,4..2*n] [3,5..2*n+1] ])
```

### 2.4.3. Grafo completo

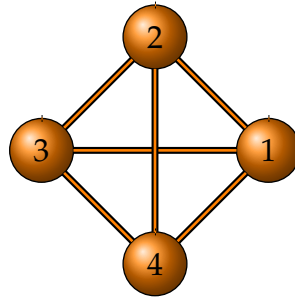
**Definición 2.4.4.** El **grafo completo**,<sup>6</sup> de orden  $n$ ,  $K(n)$ , es un grafo no dirigido cuyo conjunto de vértices viene dado por  $V = \{1, \dots, n\}$  y tiene una arista entre cada par de vértices distintos.

<sup>5</sup>[https://es.wikipedia.org/wiki/Grafo\\_de\\_la\\_amistad](https://es.wikipedia.org/wiki/Grafo_de_la_amistad)

<sup>6</sup>[https://es.wikipedia.org/wiki/Grafo\\_completo](https://es.wikipedia.org/wiki/Grafo_completo)

La función (`completo n`) nos genera el grafo completo de orden  $n$ . Por ejemplo,

```
ghci> completo 4
G (array (1,4) [(1,[2,3,4]),(2,[3,4,1]),(3,[4,1,2]),(4,[1,2,3])])
```



```
completo :: Int -> Grafo Int
completo n =
    creaGrafo [1..n]
        [(a,b) | a <- [1..n], b <- [1..a-1]]
```

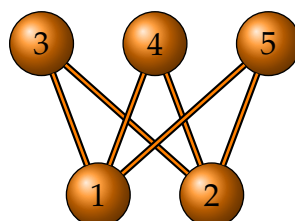
#### 2.4.4. Grafo bipartito

**Definición 2.4.5.** Un **grafo bipartito**<sup>7</sup> es un grafo  $G = (V, A)$  verificando que el conjunto de sus vértices se puede dividir en dos subconjuntos disjuntos  $V_1, V_2$  tales que  $V_1 \cup V_2 = V$  de manera que  $\forall u_1, u_2 \in V_1 [(u_1, u_2) \notin A]$  y  $\forall v_1, v_2 \in V_2 [(v_1, v_2) \notin A]$ .

Un **grafo bipartito completo**<sup>8</sup> será entonces un grafo bipartito  $G = (V_1 \cup V_2, A)$  en el que todos los vértices de una partición están conectados a los de la otra. Si  $n = |V_1|, m = |V_2|$  denotamos al grafo bipartito  $G = (V_1 \cup V_2, A)$  por  $K_{n,m}$ .

La función (`bipartitoCompleto n m`) nos genera el grafo bipartito  $K_{n,m}$ . Por ejemplo,

```
ghci> bipartitoCompleto 2 3
G (array (1,5) [(1,[3,4,5]),(2,[3,4,5]),
                (3,[1,2]),(4,[1,2]),(5,[1,2])])
```



<sup>7</sup>[https://es.wikipedia.org/wiki/Grafo\\_bipartito](https://es.wikipedia.org/wiki/Grafo_bipartito)

<sup>8</sup>[https://es.wikipedia.org/wiki/Grafo\\_bipartito\\_completo](https://es.wikipedia.org/wiki/Grafo_bipartito_completo)

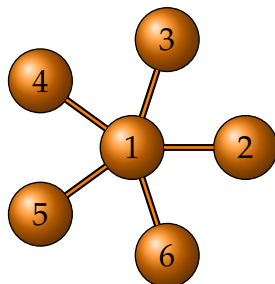
```
bipartitoCompleto :: Int -> Int -> Grafo Int
bipartitoCompleto n m =
    creaGrafo [1..n+m]
              [(a,b) | a <- [1..n], b <- [n+1..n+m]]
```

### 2.4.5. Grafo estrella

**Definición 2.4.6.** Una *estrella*<sup>9</sup> de orden  $n$  es el grafo bipartito completo  $K_{1,n}$ . Denotaremos a una estrella de orden  $n$  por  $S_n$ . Una estrella con 3 aristas se conoce en inglés como *claw* (garra o garfio).

La función (`grafoEstrella n`) crea un grafo circulante a partir de su orden  $n$ . Por ejemplo,

```
ghci> grafoEstrella 5
G (array (1,6) [(1,[2,3,4,5,6]),(2,[1]),(3,[1]),(4,[1]),(5,[1]),(6,[1])])
```



```
grafoEstrella :: Int -> Grafo Int
grafoEstrella = bipartitoCompleto 1
```

### 2.4.6. Grafo rueda

**Definición 2.4.7.** Un *grafo rueda*<sup>10</sup> de orden  $n$  es un grafo no dirigido y no ponderado con  $n$  vértices que se forma conectando un único vértice a todos los vértices de un ciclo  $C_{n-1}$ . Lo denotaremos por  $W_n$ .

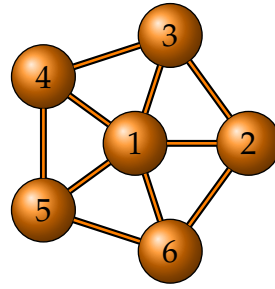
La función (`grafoRueda n`) crea un grafo rueda a partir de su orden  $n$ . Por ejemplo,

```
ghci> grafoRueda 6
G (array (1,6) [(1,[2,3,4,5,6]),(2,[1,3,6]),(3,[1,2,4]),
                (4,[1,3,5]),(5,[1,4,6]),(6,[1,2,5])])
```

<sup>9</sup>[https://en.wikipedia.org/wiki/Star\\_\(graph\\_theory\)\)](https://en.wikipedia.org/wiki/Star_(graph_theory)))

<sup>10</sup>[https://es.wikipedia.org/wiki/Grafo\\_rueda](https://es.wikipedia.org/wiki/Grafo_rueda)





```

grafoRueda :: Int -> Grafo Int
grafoRueda n =
    creaGrafo [1..n]
        ([ (1,a) | a <- [2..n] ] ++
         [ (a,b) | (a,b) <- zip (2:[2..n-1]) (3:n:[4..n]) ])

```

### 2.4.7. Grafo circulante

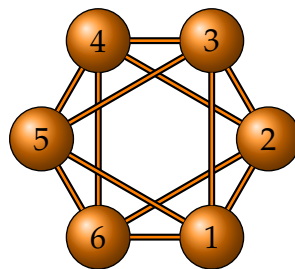
**Definición 2.4.8.** Un *grafo circulante*<sup>11</sup> de orden  $n \geq 3$  y saltos  $\{s_1, \dots, s_k\}$  es un grafo no dirigido y no ponderado  $G = (\{1, \dots, n\}, A)$  en el que cada nodo  $\forall i \in V$  es adyacente a los  $2k$  nodos  $i \pm s_1, \dots, i \pm s_k \pmod n$ . Lo denotaremos por  $\text{Cir}_n^{s_1, \dots, s_k}$ .

La función `(grafoCirculante n ss)` crea un grafo circulante a partir de su orden  $n$  y de la lista de sus saltos  $ss$ . Por ejemplo,

```

ghci> grafoCirculante 6 [1,2]
G (array (1,6) [(1,[2,3,5,6]),(2,[1,3,4,6]),(3,[1,2,4,5]),
                (4,[2,3,5,6]),(5,[1,3,4,6]),(6,[1,2,4,5])])

```



```

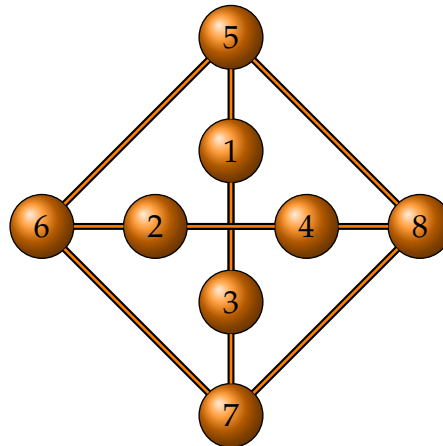
grafoCirculante :: Int -> [Int] -> Grafo Int
grafoCirculante n ss =
    creaGrafo [1..n]
        [(a,b) | a <- [1..n]
                 , b <- sort (auxCir a ss n)
                 , a < b]
    where auxCir v ss1 k =
            concat [[fun (v+s) k, fun (v-s) k] | s <- ss1]
            fun a b = if mod a b == 0 then b else mod a b

```

<sup>11</sup>[https://en.wikipedia.org/wiki/Circulant\\_graph](https://en.wikipedia.org/wiki/Circulant_graph)

El **grafo de Petersen generalizado**<sup>12</sup> que denotaremos  $GP_{n,k}$  (con  $n \geq 3$  y  $1 \leq k \leq (n-1)/2$ ) es un grafo formado por un grafo circulante  $Cir_{\{k\}}^n$  en el interior, rodeado por un ciclo  $C_n$  al que está conectado por una arista saliendo de cada vértice, de forma que se creen  $n$  polígonos regulares. El grafo  $GP_{n,k}$  tiene  $2n$  vértices y  $3n$  aristas.

La función `(grafoPetersenGen n k)` devuelve el grafo de Petersen generalizado  $GP_{n,k}$ .

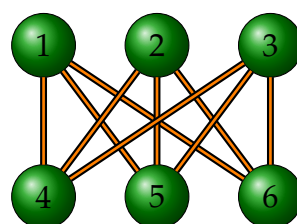


```
ghci> grafoPetersenGen 4 2
G (array (1,8) [(1,[3,3,5]),(2,[4,4,6]),(3,[1,1,7]),(4,[2,2,8]),
               (5,[1,8,6]),(6,[2,5,7]),(7,[3,6,8]),(8,[4,7,5])])
```

```
grafoPetersenGen :: Int -> Int -> Grafo Int
grafoPetersenGen n k =
  creaGrafo [1..2*n]
    (filter p (aristas (grafoCirculante n [k])) ++
     [(x,x+n) | x <- [1..n]] ++
     (n+1,n+2) : (n+1,2*n) : [(x,x+1) | x <- [n+2..2*n-1]])
  where p (a,b) = a < b
```

### 2.4.8. Otros grafos importantes

**Definición 2.4.9.** El grafo bipartito completo  $K_{3,3}$  es conocido como el **grafo de Thomson** y, como veremos más adelante, será clave a la hora de analizar propiedades topológicas de los grafos.



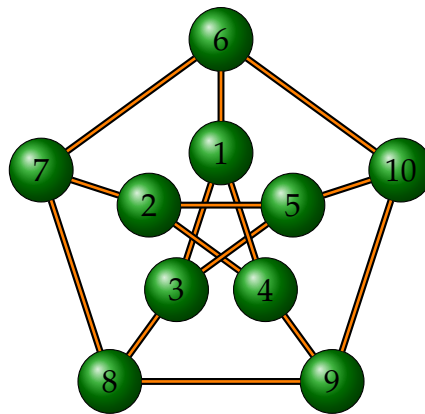
<sup>12</sup>[https://en.wikipedia.org/wiki/Generalized\\_Petersen\\_graph](https://en.wikipedia.org/wiki/Generalized_Petersen_graph)

La función `grafoThomson` genera el grafo de Thomson.

```
ghci> grafoThomson
G (array (1,6) [(1,[4,5,6]),(2,[4,5,6]),(3,[4,5,6]),
               (4,[1,2,3]),(5,[1,2,3]),(6,[1,2,3])])
```

```
grafoThomson :: Grafo Int
grafoThomson = bipartitoCompleto 3 3
```

El **grafo de Petersen** <sup>13</sup> es un grafo no dirigido con 10 vértices y 15 aristas que es usado como ejemplo y como contraejemplo en muchos problemas de la Teoría de grafos.



La función `grafoPetersen` devuelve el grafo de Petersen.

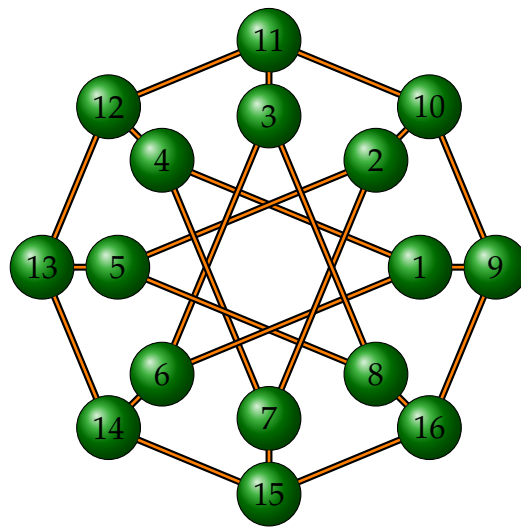
```
ghci> grafoPetersen
G (array (1,10) [(1,[3,4,6]),(2,[4,5,7]),(3,[1,5,8]),(4,[1,2,9]),
                (5,[2,3,10]),(6,[1,10,7]),(7,[2,6,8]),
                (8,[3,7,9]),(9,[4,8,10]),(10,[5,9,6])])
```

```
grafoPetersen :: Grafo Int
grafoPetersen = grafoPetersenGen 5 2
```

El **grafo de Moëbius-Cantor** <sup>14</sup> se define como el grafo de Petersen generalizado  $GP_{8,3}$ ; es decir, está formado por los vértices de un octógono, conectados a los vértices de una estrella de ocho puntas en la que cada nodo es adyacente a los nodos que están a un salto 3 de él. Al igual que el grafo de Petersen, tiene importantes propiedades que lo hacen ser ejemplo y contraejemplo de muchos problemas de la Teoría de Grafos.

<sup>13</sup>[https://en.wikipedia.org/wiki/Petersen\\_graph](https://en.wikipedia.org/wiki/Petersen_graph)

<sup>14</sup>[https://en.wikipedia.org/wiki/Moëbius-Kantor\\_graph](https://en.wikipedia.org/wiki/Moëbius-Kantor_graph)



La función `grafoMoebiusCantor` genera el grafo de Moëbius-Cantor

```
G (array (1,16) [( 1,[4, 6, 9]),( 2,[5, 7,10]),( 3,[6, 8,11]),
                  ( 4,[1, 7,12]),( 5,[2, 8,13]),( 6,[1, 3,14]),
                  ( 7,[2, 4,15]),( 8,[3, 5,16]),( 9,[1,10,16]),
                  (10,[2, 9,11]),(11,[3,10,12]),(12,[4,11,13]),
                  (13,[5,12,14]),(14,[6,13,15]),(15,[7,14,16]),
                  (16,[8, 9,15])])
```

```
grafoMoebiusCantor :: Grafo Int
grafoMoebiusCantor = grafoPetersenGen 8 3
```

## 2.5. Definiciones y propiedades

Una vez construida una pequeña fuente de ejemplos, estamos en condiciones de implementar las definiciones sobre grafos en Haskell y ver que funcionan correctamente. Además, comprobaremos que se cumplen las propiedades básicas que se han presentado en el tema [Introducción a la teoría de grafos](#)<sup>15</sup> de Matemática Discreta.

Añadir al fichero de bibliografía y citar.

*Nota 2.5.1.* Se utilizará el tipo abstracto de grafos presentados en la sección 2.2 y se utilizarán las librerías `Data.List` y `Test.QuickCheck`.

### 2.5.1. Definiciones de grafos

**Definición 2.5.1.** El orden de un grafo  $G = (V, A)$  se define como su número de vértices. Lo denotaremos por  $|V(G)|$ .

<sup>15</sup><https://dl.dropboxusercontent.com/u/15420416/tiddly/emptyMD1314.html>

La función (`orden g`) devuelve el orden del grafo  $g$ . Por ejemplo,

```
orden (grafoCiclo 4)      == 4
orden (grafoEstrella 4)   == 5
orden grafoPetersen       == 10
orden (grafoPetersenGen 2 5) == 4
orden (completo 3)        == 3
```

```
orden :: Grafo a -> Int
orden = length . vertices
```

**Definición 2.5.2.** El *tamaño* de un grafo  $G = (V, A)$  se define como su número de aristas. Lo denotaremos por  $|A(G)|$ .

La función (`tamaño g`) devuelve el orden del grafo  $g$ . Por ejemplo,

```
tamaño (grafoCiclo 4)      == 4
tamaño (grafoEstrella 4)   == 4
tamaño grafoPetersen       == 15
tamaño (grafoPetersenGen 2 5) == 4
tamaño (completo 3)        == 3
```

```
tamaño :: Grafo a -> Int
tamaño = length . aristas
```

**Definición 2.5.3.** Diremos que dos aristas  $a, a'$  son *incidentes* si tienen intersección no vacía; es decir, si tienen algún vértice en común.

La función (`sonIncidentes a a'`) se verifica si las aristas  $a$  y  $a'$  son incidentes. Por ejemplo,

```
sonIncidentes (1,2) (2,4) == True
sonIncidentes (1,2) (3,4) == False
```

```
sonIncidentes :: Eq a => (a,a) -> (a,a) -> Bool
sonIncidentes (u1,u2) (v1,v2) =
  or [u1 == v1, u1 == v2, u2 == v1, u2 == v2]
```

**Definición 2.5.4.** Diremos que una arista de un grafo  $G$  es un *lazo* si va de un vértice en sí mismo.

La función (`esLazo a`) se verifica si la arista `a` es un lazo. Por ejemplo,

```
esLazo (1,2) == False
esLazo (4,4) == True
```

```
esLazo :: Eq a => (a,a) -> Bool
esLazo (u,v) = u == v
```

**Definición 2.5.5.** Dado un grafo  $G = (V, A)$ , fijado un vértice  $v \in V$ , al conjunto de vértices que son adyacentes a  $v$  lo llamaremos **entorno** de  $v$  y lo denotaremos por  $N(v) = \{u \in V \mid (u, v) \in A\}$ .

La función (`entorno g v`) devuelve el entorno del vértice  $v$  en el grafo  $g$ . Por ejemplo,

```
entorno (grafoEstrella 5) 0 == [1,2,3,4,5]
entorno (grafoEstrella 5) 1 == [0]
entorno (bipartitoCompleto 2 4) 5 == [1,2]
entorno grafoPetersen 4 == [1,2,9]
```

```
entorno :: Eq a => Grafo a -> a -> [a]
entorno = adyacentes
```

**Definición 2.5.6.** Sea  $G = (V, A)$  un grafo. El **grado** (o **valencia**) de  $v \in V$  es  $\text{grad}(v) = |N(v)|$ .

La función (`grado g v`) devuelve el grado del vértice  $v$  en el grafo  $g$ . Por ejemplo,

```
grado (grafoEstrella 5) 0 == 5
grado (grafoEstrella 5) 1 == 1
grado (grafoThomson) 6 == 3
grado (grafoAmistad 2) 4 == 2
```

```
grado :: Eq a => Grafo a -> a -> Int
grado g v = length (entorno g v)
```

**Definición 2.5.7.** Un vértice  $v$  de un grafo es **aislado** si su grado es 0.

La función (`esAislado g v`) se verifica si el vértice  $v$  es aislado en el grafo  $g$ . Por ejemplo,

```

esAislado (grafoEstrella 5)           0 == False
esAislado (bipartitoCompleto 1 5)      4 == False
esAislado (creaGrafo [1..4] [(1,2),(1,4),(2,4)]) 2 == False
esAislado (creaGrafo [1..4] [(1,2),(1,4),(2,4)]) 3 == True

```

```

esAislado :: Eq a => Grafo a -> a -> Bool
esAislado g v = grado g v == 0

```

**Definición 2.5.8.** Un grafo es *regular* si todos sus vértices tienen el mismo grado.

La función (`esRegular g`) se verifica si el grafo `g` es regular. Por ejemplo,

```

esRegular (grafoEstrella 4)           == False
esRegular (grafoCiclo 5)              == True
esRegular (grafoRueda 7)              == False
esRegular (bipartitoCompleto 2 2)    == True

```

```

esRegular :: Eq a => Grafo a -> Bool
esRegular g = all (==x) xs
  where (x:xs) = [grado g v | v <- vertices g]

```

**Definición 2.5.9.** Dado un grafo  $G = (V, A)$  llamamos *valencia mínima* o *grado mínimo* de  $G$  al valor  $\delta(G) = \min\{\text{grad}(v) | v \in V\}$

La función (`valenciaMin g`) devuelve la valencia mínima del grafo `g`.

```

valenciaMin (grafoEstrella 6)          == 1
valenciaMin (grafoCiclo 4)             == 2
valenciaMin grafoPetersen              == 3
valenciaMin (creaGrafo [1..4] [(1,2),(1,4),(2,4)]) == 0

```

```

valenciaMin :: Ord a => Grafo a -> Int
valenciaMin g = minimum [grado g v | v <- vertices g]

```

**Definición 2.5.10.** Dado un grafo  $G = (V, A)$  llamamos *valencia máxima* o *grado máximo* de  $G$  al valor  $\delta(G) = \max\{\text{grad}(v) | v \in V\}$

La función (`valenciaMax g`) devuelve la valencia máxima del grafo `g`.

```

valenciaMax (grafoEstrella 6) == 6
valenciaMax (grafoCiclo 4)   == 2
valenciaMax grafoPetersen    == 3
valenciaMax ejGrafoD         == 3

```

```
valenciaMax :: Ord a => Grafo a -> Int
valenciaMax g = maximum [grado g v | v <- vertices g]
```

**Definición 2.5.11.** Se dice que un grafo es simple si no contiene lazos ni aristas repetidas.

La función (`esSimple g`) se verifica si  $g$  es un grafo simple.

```
esSimple (bipartitoCompleto 3 4) == True
esSimple (creaGrafo [1..3] [(1,1),(1,2),(2,3)]) == False
esSimple (creaGrafo [1..3] [(1,2),(1,2),(2,3)]) == False
esSimple (creaGrafo [1..3] [(1,2),(1,3),(2,3)]) == True
```

```
esSimple :: Ord a => Grafo a -> Bool
esSimple g =
  and [not (aristaEn g (x,x)) | x <- vertices g]
```

**Definición 2.5.12.** Sea  $G$  un grafo. Llamamos **secuencia de grados** de  $G$  a la lista de grados de sus vértices. La secuencia se suele presentar en orden decreciente:  $d_1 \geq d_2 \geq \dots \geq d_n$ .

La función (`secuenciaGrados g`) devuelve la secuencia de los grados del grafo  $g$  en orden decreciente.

```
secuenciaGrados (grafoEstrella 6) == [6,1,1,1,1,1,1]
secuenciaGrados (grafoCiclo 4)    == [2,2,2,2]
secuenciaGrados grafoPetersen     == [3,3,3,3,3,3,3,3,3,3]
secuenciaGrados ejGrafo           == [4,3,3,3,3]
```

```
secuenciaGrados :: Eq a => Grafo a -> [Int]
secuenciaGrados g = sortBy (flip compare) [grado g v | v <- vertices g]
```

*Nota 2.5.2.* ¿Qué listas de  $n$  números enteros son secuencias de grafos de  $n$  vértices?

- Si  $\sum_{i=1}^n d_i$  es impar, no hay ninguno.
- Si  $\sum_{i=1}^n d_i$  es par, entonces siempre hay un grafo con esa secuencia de grados (aunque no necesariamente simple).

**Definición 2.5.13.** Una **secuencia gráfica** es una lista de número enteros no negativos que es la secuencia de grados para algún grafo simple.

La función (`secuenciaGrafica ss`) se verifica si existe algún grafo con la secuencia de grados  $ss$ .



```

secuenciaGrafica [2,2,2,2,2,2] == True
secuenciaGrafica [6,1,1,1,1,1,1] == True
secuenciaGrafica [6,1,1,1,1,1] == False
secuenciaGrafica [5,4..1] == False

```

```

secuenciaGrafica :: [Int] -> Bool
secuenciaGrafica ss = even (sum ss) && all p ss
  where p s = s >= 0 && s <= length ss

```

**Definición 2.5.14.** Dado un grafo  $G = (V, A)$ , diremos que  $G' = (V', A')$  es un *subgrafo* de  $G$  si  $V' \subseteq V$  y  $A' \subseteq A$ .

La función `(subgrafo g' g)` se verifica si  $g'$  es un subgrafo de  $g$

```

subgrafo (bipartitoCompleto 3 2) (bipartitoCompleto 3 3) == True
subgrafo (grafoEstrella 4) (grafoEstrella 5) == True
subgrafo (completo 5) (completo 4) == False
subgrafo (completo 3) (completo 4) == True

```

```

esSubgrafo :: Ord a => Grafo a -> Grafo a -> Bool
esSubgrafo g' g =
  vertices g' 'esSubconjunto' vertices g &&
  aristas g' 'esSubconjunto' aristas g

```

**Definición 2.5.15.** Si  $G' = (V', A')$  es un subgrafo de  $G = (V, A)$  tal que  $V' = V$ , diremos que  $G'$  es un *subgrafo maximal*, *grafo recubridor* o *grafo de expansión* (en inglés, *spanning graph*) de  $G$ .

La función `(esSubgrafoMax g' g)` se verifica si  $g'$  es un subgrafo maximal de  $g$ .

```

esSubgrafoMax (grafoRueda 3) (grafoRueda 4) == False
esSubgrafoMax (grafoCiclo 4) (grafoRueda 4) == True
esSubgrafoMax (creaGrafo [1..3] [(1,2)]) (grafoCiclo 3) == True
esSubgrafoMax (creaGrafo [1..2] [(1,2)]) (grafoCiclo 3) == False

```

```

esSubgrafoMax :: Ord a => Grafo a -> Grafo a -> Bool
esSubgrafoMax g' g =
  g' 'esSubgrafo' g && conjuntosIguales (vertices g') (vertices g)

```

**Definición 2.5.16.** Sean  $G' = (V', A')$ ,  $G = (V, A)$  dos grafos si  $V' \subset V$ , o  $A' \subset A$ , se dice que  $G'$  es un *subgrafo propio* de  $G$ , y se denota por  $G' \subset G$ .

La función `(esSubgrafoPropio g' g)` se verifica si  $g'$  es un subgrafo propio de  $g$ .

```
esSubgrafoPropio (grafoRueda 4) (grafoRueda 3)      == True
esSubgrafoPropio (grafoRueda 4) (grafoCiclo 5)      == False
esSubgrafoPropio (grafoCiclo 3) (creaGrafo [1..3] [(1,2)]) == True
esSubgrafoPropio (grafoCiclo 3) (creaGrafo [1..2] [(1,2)]) == True
```

```
esSubgrafoPropio :: Ord a => Grafo a -> Grafo a -> Bool
esSubgrafoPropio g' g =
  esSubgrafo g' g &&
  (not (conjuntosIguales (vertices g) (vertices g'))) ||
  not (conjuntosIguales (aristas g) (aristas g')))
```

### 2.5.2. Propiedades de grafos

**Teorema 2.5.17** (Lema del apretón de manos). *En todo grafo simple el número de vértices de grado impar es par o cero.*

Vamos a comprobar que se verifica el lema del apretón de manos utilizando la función `prop_LemaApretonDeManos`.

```
ghci> quickCheck prop_LemaApretonDeManos
+++ OK, passed 100 tests.
```

```
prop_LemaApretonDeManos :: Grafo Int -> Property
prop_LemaApretonDeManos g =
  esSimple g ==>
  even (length (filter odd [grado g v | v <- vertices g]))
```

**Teorema 2.5.18** (Havel–Hakimi). *Si  $n > 1$  y  $D = [d_1, \dots, d_n]$  es una lista de enteros, entonces  $D$  es secuencia gráfica si y sólo si la secuencia  $D'$  obtenida borrando el mayor elemento  $d_{\max}$  y restando 1 a los siguientes  $d_{\max}$  elementos más grandes es gráfica.*

Vamos a comprobar que se verifica el teorema de Havel-Hakimi utilizando la función `prop_HavelHakimi`.

```
ghci> quickCheck prop_HavelHakimi
+++ OK, passed 100 tests.
```

```
prop_HavelHakimi :: [Int] -> Bool
prop_HavelHakimi [] = True
prop_HavelHakimi (s:ss) =
  not (secuenciaGrafica (s:ss) && not (esVacio ss)) ||
  secuenciaGrafica (map (\x -> x-1) (take s ss) ++ drop s ss)
```

### 2.5.3. Operaciones y propiedades sobre grafos

#### Eliminación de una arista

**Definición 2.5.19.** Sea  $G = (V, A)$  un grafo y sea  $(u, v) \in A$ . Definimos el grafo  $G \setminus (u, v)$  como el subgrafo de  $G$ ,  $G' = (V', A')$ , con  $V' = V$  y  $A' = A \setminus \{(u, v)\}$ . Esta operación se denomina **eliminar una arista**.

La función `(eliminaArista g a)` elimina la arista  $a$  del grafo  $g$ .

```
ghci> grafoThomson
G [1,2,3,4,5,6] [(1,4),(1,5),(1,6),(2,4),(2,5),
                (2,6),(3,4),(3,5),(3,6)]
ghci> eliminaArista grafoThomson (3,4)
G [1,2,3,4,5,6] [(1,4),(1,5),(1,6),(2,4),(2,5),(2,6),(3,5),(3,6)]
ghci> eliminaArista grafoThomson (4,3)
G [1,2,3,4,5,6] [(1,4),(1,5),(1,6),(2,4),(2,5),(2,6),(3,5),(3,6)]
ghci> eliminaArista grafoThomson (1,3)
G [1,2,3,4,5,6] [(1,4),(1,5),(1,6),(2,4),(2,5),
                (2,6),(3,4),(3,5),(3,6)]
```

```
eliminaArista :: Ord a => Grafo a -> (a,a) -> Grafo a
eliminaArista g (a,b) =
  creaGrafo (vertices g)
            (aristas g \ \ [(a,b),(b,a)])
```

#### Eliminación un vértice

**Definición 2.5.20.** Sea  $G = (V, A)$  un grafo y sea  $v \in V$ . Definimos el grafo  $G \setminus v$  como el subgrafo de  $G$ ,  $G' = (V', A')$ , con  $V' = V \setminus \{v\}$  y  $A' = A \setminus \{a \in A | v \text{ es un extremo de } a\}$ . Esta operación se denomina **eliminar un vértice**.

La función `(eliminaVertice g v)` elimina el vértice  $v$  del grafo  $g$ .

```
ghci> grafoThomson
G [1,2,3,4,5,6] [(1,4),(1,5),(1,6),(2,4),(2,5),
                (2,6),(3,4),(3,5),(3,6)]
ghci> eliminaVertice grafoThomson 3
G [1,2,4,5,6] [(1,4),(1,5),(1,6),(2,4),(2,5),(2,6)]
ghci> eliminaVertice grafoThomson 2
G [1,3,4,5,6] [(1,4),(1,5),(1,6),(3,4),(3,5),(3,6)]
ghci> eliminaVertice grafoThomson 8
G [1,2,3,4,5,6] [(1,4),(1,5),(1,6),(2,4),(2,5),
                (2,6),(3,4),(3,5),(3,6)]
```

```

eliminaVertice :: Ord a => Grafo a -> a -> Grafo a
eliminaVertice g v =
  creaGrafo (vertices g \ [v])
            (as \ [(a,b) | (a,b) <- as, a == v || b == v])
  where as = aristas g

```

### Suma de aristas

**Definición 2.5.21.** Sea  $G = (V, A)$  un grafo y sean  $u, v \in V$  tales que  $(u, v), (v, u) \notin A$ . Definimos el grafo  $G + (u, v)$  como el grafo  $G' = (V, A \cup \{(u, v)\})$ . Esta operación se denomina **suma de una arista**.

La función (`sumaArista g a`) suma la arista  $a$  al grafo  $g$ .

```

ghci> sumaArista (grafoCiclo 5) (1,3)
G [1,2,3,4,5] [(1,2),(1,3),(1,5),(2,3),(3,4),(4,5)]
ghci> sumaArista (grafoEstrella 5) (4,5)
G [1,2,3,4,5,6] [(1,2),(1,3),(1,4),(1,5),(1,6),(4,5)]

```

```

sumaArista :: Ord a => Grafo a -> (a,a) -> Grafo a
sumaArista g a =
  creaGrafo (vertices g) (a : aristas g)

```

### Suma de vértices

**Definición 2.5.22.** Sea  $G = (V, A)$  un grafo y sea  $v \notin V$ . Definimos el grafo  $G + v$  como el grafo  $G' = (V', A')$ , donde  $V' = V \cup \{v\}$ ,  $A' = A \cup \{(u, v) | u \in V\}$ . Esta operación se denomina **suma de un vértice**.

La función (`sumaVertice g a`) suma el vértice  $a$  al grafo  $g$ .

```

ghci> sumaVertice (completo 5) 6
G [1,2,3,4,5,6] [(1,2),(1,3),(1,4),(1,5),(1,6),
                  (2,3),(2,4),(2,5),(2,6),(3,4),
                  (3,5),(3,6),(4,5),(4,6),(5,6)]
ghci> sumaVertice (creaGrafo [2..5] []) 1
G [1,2,3,4,5] [(1,2),(1,3),(1,4),(1,5)]

```

```

sumaVertice :: Ord a => Grafo a -> a -> Grafo a
sumaVertice g v =
  creaGrafo (v : vs) (aristas g ++ [(u,v) | u <- vs])
  where vs = vertices g

```

### Propiedad de los grafos completos

**Proposición 2.5.23.** *La familia de grafos completos  $K_n$  verifica que  $K_n = K_{n-1} + n$ .*

Vamos a ver que se cumple la propiedad utilizando la función `prop_completos`.

```
ghci> quickCheck prop_completos
+++ OK, passed 100 tests.
```

```
prop_completos :: Int -> Property
prop_completos n = n >= 2 ==>
  completo n == sumaVertice (completo (n-1)) n
```

A partir de esta propiedad, se puede dar una definición alternativa de  $K_n$  (`completo2`) y comprobar su equivalencia con la primera (`completo`).

### Suma de grafos

**Definición 2.5.24.** Sean  $G = (V, A), G' = (V', A)$  dos grafos. Definimos el grafo suma de  $G$  y  $G'$  como el grafo  $G + G' = (V \cup V', A \cup A' \cup \{(u, v) | u \in V, v \in V'\})$ . Esta operación se denomina **suma de grafos**.

La función (`sumaGrafos g g'`) suma los grafos  $g$  y  $g'$ . Por ejemplo,

```
ghci> sumaGrafos (grafoCiclo 3) (grafoCiclo 3)
G [1,2,3] [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
ghci> sumaGrafos (grafoRueda 3) (grafoEstrella 4)
G [1,2,3,4,5] [(1,1),(1,2),(1,3),(1,4),(1,5),(2,2),
               (2,3),(2,4),(2,5),(3,3),(3,4),(3,5)]
```

```
sumaGrafos :: Ord a => Grafo a -> Grafo a -> Grafo a
sumaGrafos g1 g2 =
  creaGrafo (vs1 'union' vs2)
    (aristas g1 'union'
      aristas g2 'union'
      [(u,v) | u <- vs1, v <- vs2])
  where vs1 = vertices g1
        vs2 = vertices g2
```

## Unión de grafos

**Definición 2.5.25.** Sean  $G = (V, A), G' = (V', A)$  dos grafos. Definimos el grafo unión de  $G$  y  $G'$  como el grafo  $G \cup H = (V \cup V', A \cup A')$ . Esta operación se denomina **unión de grafos**.

La función (`unionGrafos g g'`) une los grafos  $g$  y  $g'$ . Por ejemplo,

```
ghci> unionGrafos (grafoCiclo 3) (grafoCiclo 3)
G [1,2,3] [(1,2),(1,3),(2,3)]
ghci> unionGrafos (grafoRueda 3) (grafoEstrella 4)
G [1,2,3,4,5] [(1,2),(1,3),(1,4),(1,5),(2,3),(2,3)]
```

```
unionGrafos :: Ord a => Grafo a -> Grafo a -> Grafo a
unionGrafos g1 g2 =
  creaGrafo (vertices g1 'union' vertices g2)
            (aristas g1 'union' aristas g2)
```

## Grafo complementario

**Definición 2.5.26.** Dado un grafo  $G = (V, A)$  se define el **grafo complementario** de  $G$  como  $\overline{G} = (V, \overline{A})$ , donde  $\overline{A} = \{(u, v) | u, v \in V, (u, v) \notin A\}$ .

La función (`complementario g`) devuelve el grafo complementario de  $g$ .

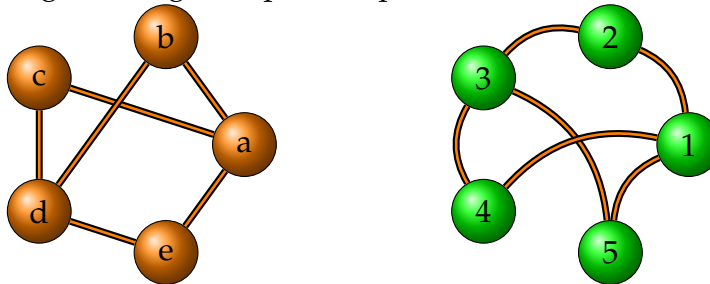
```
ghci> complementario (grafoEstrella 5)
G [1,2,3,4,5,6] [(1,1),(2,2),(2,3),(2,4),(2,5),(2,6),
                (3,3),(3,4),(3,5),(3,6),(4,4),(4,5),
                (4,6),(5,5),(5,6),(6,6)]
ghci> complementario (completo 4)
G [1,2,3,4] [(1,1),(2,2),(3,3),(4,4)]
```

```
complementario :: Ord a => Grafo a -> Grafo a
complementario g =
  creaGrafo vs
            [(u,v) | u <- vs, v <- vs, u <= v, not (aristaEn g (u,v))]
  where vs = vertices g
```

## 2.6. Morfismos de grafos

Llegados a este punto, es importante resaltar que un grafo se define como una entidad matemática abstracta; es evidente que lo importante de un grafo no son los nombres de sus vértices ni su representación gráfica. La propiedad que caracteriza a un grafo es la forma en que sus vértices están unidos por las aristas.

A priori, los siguientes grafos pueden parecer distintos:



Sin embargo, ambos grafos son grafos de cinco vértices que tienen las mismas relaciones de vecindad entre sus nodos. En esta sección estudiaremos estas relaciones que establecen las aristas entre los vértices de un grafo y presentaremos algoritmos que nos permitan identificar cuándo dos grafos se pueden relacionar mediante aplicaciones entre sus vértices cumpliendo ciertas características.

### 2.6.1. Morfismos

**Definición 2.6.1.** Si  $f$  es una función entre dos grafos  $G = (V, A)$  y  $G' = (V', A')$ , diremos que **conserva la adyacencia** si  $\forall u, v \in V$  se verifica que si  $(u, v) \in A$ , entonces  $(f(u), f(v)) \in A'$ .

La función `(conservaAdyacencia g h f)` se verifica si la función  $f$  entre los grafos  $g$  y  $h$  conserva las adyacencias. Por ejemplo,

```
ghci> let g1 = creaGrafo [1..4] [(1,2),(2,3),(3,4)]
ghci> let g2 = creaGrafo [1..4] [(1,2),(2,3),(2,4)]
ghci> let g3 = creaGrafo [4,6..10] [(4,8),(6,8),(8,10)]
ghci> conservaAdyacencia g1 g3 [(1,4),(2,6),(3,8),(4,10)]
False
ghci> conservaAdyacencia g2 g3 [(1,4),(2,8),(3,6),(4,10)]
True
```

```
conservaAdyacencia :: (Ord a, Ord b) =>
    Grafo a -> Grafo b -> Funcion a b -> Bool
conservaAdyacencia g h f =
    and [(imagen f x, imagen f y) 'aristaEn' h | (x,y) <- aristas g]
```

**Definición 2.6.2.** *Dados dos grafos simples  $G = (V, A)$  y  $G' = (V', A')$ , un **morfismo** entre  $G$  y  $G'$  es una función  $\phi : V \rightarrow V'$  que conserva las adyacencias.*

La función `(esMorfismo g h vvs)` se verifica si la función representada por `vvs` es un morfismo entre los grafos `g` y `h`.

```
ghci> let g1 = creaGrafo [1,2,3] [(1,2),(2,3)]
ghci> let g2 = creaGrafo [4,5,6] [(4,6),(5,6)]
ghci> esMorfismo g1 g2 [(1,4),(2,6),(3,5)]
True
ghci> esMorfismo g1 g2 [(1,4),(2,5),(3,6)]
False
ghci> esMorfismo g1 g2 [(1,4),(2,6),(3,5),(7,9)]
False
```

```
esMorfismo :: (Ord a, Ord b) => Grafo a -> Grafo b ->
              Funcion a b -> Bool
esMorfismo g1 g2 f =
  esFuncion (vertices g1) (vertices g2) f &&
  conservaAdyacencia g1 g2 f
```

La función `(morfismos g h)` devuelve todos los posibles morfismos entre los grafos `g` y `h`.

```
ghci> morfismos (grafoCiclo 3)
              (creaGrafo [4..6] [(4,5),(4,6),(5,6)])
[[ (1,4),(2,5),(3,6)], [(1,4),(2,6),(3,5)], [(1,5),(2,4),(3,6)],
  [(1,5),(2,6),(3,4)], [(1,6),(2,4),(3,5)], [(1,6),(2,5),(3,4)]]
ghci> morfismos (bipartitoCompleto 1 2) (grafoCiclo 3)
[[ (1,1),(2,2),(3,2)], [(1,1),(2,2),(3,3)], [(1,1),(2,3),(3,2)],
  [(1,1),(2,3),(3,3)], [(1,2),(2,1),(3,1)], [(1,2),(2,1),(3,3)],
  [(1,2),(2,3),(3,1)], [(1,2),(2,3),(3,3)], [(1,3),(2,1),(3,1)],
  [(1,3),(2,1),(3,2)], [(1,3),(2,2),(3,1)], [(1,3),(2,2),(3,2)]]
```

```
morfismos :: (Ord a, Ord b) => Grafo a -> Grafo b -> [[(a,b)]]
morfismos g h =
  [f | f <- funciones (vertices g) (vertices h)
    , conservaAdyacencia g h f]
```

Buscar definiciones más eficientes de morfismos.

Introducir el problema de decidir si dos grafos son homomorfos.



### 2.6.2. Isomorfismos

**Definición 2.6.3.** *Dados dos grafos simples  $G = (V, A)$  y  $G' = (V', A')$ , un **isomorfismo** entre  $G$  y  $G'$  es un morfismo biyectivo cuyo inverso es morfismo entre  $G'$  y  $G$ .*

La función  $(\text{esIsomorfismo } g \ h \ f)$  se verifica si la aplicación  $f$  es un isomorfismo entre los grafos  $g$  y  $h$ .

```
ghci> esIsomorfismo (grafoCiclo 3)
      (creaGrafo ['a'..'c']
        [(('a','b'),('a','c'),('b','c'))]
        [(1,'a'),(2,'b'),(3,'c')])
True
ghci> esIsomorfismo (bipartitoCompleto 1 2) (grafoCiclo 3)
[(1,1),(2,3),(3,2)]
False
ghci> esIsomorfismo (bipartitoCompleto 1 2) (grafoCiclo 3)
[(1,3),(2,2),(2,2)]
False
```

```

esIsomorfismo :: (Ord a,Ord b) =>
                    Grafo a -> Grafo b -> Funcion a b -> Bool
esIsomorfismo g h f =
    esBiyectiva vs1 vs2 f      &&
    esMorfismo g h f           &&
    esMorfismo h g (inversa f)
    where vs1 = vertices g
          vs2 = vertices h

```

La función (isomorfismos1 g h) devuelve todos los isomorfismos posibles entre los grafos g y h. Por ejemplo,

```
ghci> isomorfismos1 (bipartitoCompleto 1 2) (grafoCiclo 3)
[]
ghci> isomorfismos1 (bipartitoCompleto 1 2)
      (creaGrafo "abc" [( 'a', 'b'), ('b', 'c')])
[[ (1, 'b'), (2, 'a'), (3, 'c') ], [ (1, 'b'), (2, 'c'), (3, 'a') ] ]
ghci> isomorfismos1 (grafoCiclo 4)
      (creaGrafo [5..8] [(5,7),(5,8),(6,7),(6,8)])
[[ (1,5), (2,7), (3,6), (4,8) ], [ (1,5), (2,8), (3,6), (4,7) ],
  [ (1,6), (2,7), (3,5), (4,8) ], [ (1,6), (2,8), (3,5), (4,7) ],
  [ (1,7), (2,5), (3,8), (4,6) ], [ (1,7), (2,6), (3,8), (4,5) ],
  [ (1,8), (2,5), (3,7), (4,6) ], [ (1,8), (2,6), (3,7), (4,5) ] ]
```

```
isomorfismos1 :: (Ord a, Ord b) => Grafo a -> Grafo b -> [Funcion a b]
isomorfismos1 g h =
  [f | f <- biyecciones vs1 vs2 , conservaAdyacencia g h f]
  where vs1 = vertices g
        vs2 = vertices h
```

**Definición 2.6.4.** Dos grafos  $G$  y  $H$  se dicen *isomorfos* si existe algún isomorfismo entre ellos.

La función `isomorfos1 g h` se verifica si los grafos  $g$  y  $h$  son isomorfos. Por ejemplo,

```
ghci> isomorfos1 (grafoRueda 4) (completo 4)
True
ghci> isomorfos1 (grafoRueda 5) (completo 5)
False
ghci> isomorfos1 (grafoEstrella 2) (bipartitoCompleto 1 2)
True
ghci> isomorfos1 (grafoCiclo 5) (bipartitoCompleto 2 3)
False
```

```
isomorfos1 :: (Ord a, Ord b) => Grafo a -> Grafo b -> Bool
isomorfos1 g = not . null . isomorfismos1 g
```

*Nota 2.6.1.* Al tener Haskell una evaluación perezosa, la función `(isomorfos g h)` no necesita generar todos los isomorfismos entre los grafos  $g$  y  $h$ .

**Definición 2.6.5.** Sea  $G = (V, A)$  un grafo. Un *invariante por isomorfismos* de  $G$  es una propiedad de  $G$  que tiene el mismo valor para todos los grafos que son isomorfos a él.

```
esInvariantePorIsomorfismos ::
  Eq a => (Grafo Int -> a) -> Grafo Int -> Grafo Int -> Bool
esInvariantePorIsomorfismos p g h =
  isomorfos g h --> (p g == p h)
  where (-->) = (<=)
```

**Teorema 2.6.6.** Sean  $G = (V, A)$  y  $G' = (V', A')$  dos grafos y  $\phi : V \rightarrow V'$  un isomorfismo. Entonces, se verifica que  $|V(G)| = |V(G')|$ ; es decir, el orden de un grafo es un invariante por isomorfismos.

Vamos a comprobar el teorema anterior con QuickCheck.

```
ghci> quickCheckWith (stdArgs maxSize=10)
      (esInvariantePorIsomorfismos orden)
+++ OK, passed 100 tests.
```

**Teorema 2.6.7.** Sean  $G = (V, A)$  y  $G' = (V', A')$  dos grafos y  $\phi : V \rightarrow V'$  un isomorfismo. Entonces, se verifica que  $|A(G)| = |A(G')|$ ; es decir, el tamaño de un grafo es un invariante por isomorfismos.

Vamos a comprobar el teorema anterior con QuickCheck.

```
ghci> quickCheckWith (stdArgs maxSize=7)
      (esInvariantePorIsomorfismos tamaño)
+++ OK, passed 100 tests.
```

**Teorema 2.6.8.** Sean  $G = (V, A)$  y  $G' = (V', A')$  dos grafos y  $\phi : V \rightarrow V'$  un isomorfismo. Entonces, tienen la misma secuencia de grados; es decir, la secuencia de grados de un grafo es un invariante por isomorfismos.

Vamos a comprobar el teorema anterior con QuickCheck.

```
ghci> quickCheckWith (stdArgs maxSize=7)
      (esInvariantePorIsomorfismos secuenciaGrados)
+++ OK, passed 100 tests.
```

A partir de las propiedades que hemos demostrado de los isomorfismos, vamos a dar otra definición equivalente de las funciones `(isomorfismos1 g h)` y `(isomorfos1 g h)`.

```
isomorfismos2 :: (Ord a, Ord b) => Grafo a -> Grafo b -> [Funcion a b]
isomorfismos2 g h
  | orden g /= orden h = []
  | tamaño g /= tamaño h = []
  | secuenciaGrados g /= secuenciaGrados h = []
  | otherwise = [f | f <- biyecciones vs1 vs2
                    , conservaAdyacencia g h f]
  where vs1 = vertices g
        vs2 = vertices h

isomorfos2 :: (Ord a, Ord b) => Grafo a -> Grafo b -> Bool
isomorfos2 g =
  not. null . isomorfismos2 g
```

Vamos a comparar la eficiencia entre ambas definiciones

```
ghci> let n = 6 in (length (isomorfismos1 (completo n) (completo n)))
720
(0.29 secs, 32,272,208 bytes)
ghci> let n = 6 in (length (isomorfismos2 (completo n) (completo n)))
720
(0.30 secs, 36,592,648 bytes)
```

```

ghci> let n = 6 in (length (isomorfismos1 (grafoCiclo n) (grafoCiclo n)))
12
(0.04 secs, 0 bytes)
ghci> let n = 6 in (length (isomorfismos2 (grafoCiclo n) (grafoCiclo n)))
12
(0.04 secs, 0 bytes)

ghci> length (isomorfismos1 (grafoCiclo 6) (completo 7))
0
(0.00 secs, 0 bytes)
ghci> length (isomorfismos2 (grafoCiclo 6) (completo 7))
0
(0.01 secs, 0 bytes)

ghci> isomorfos1 (completo 10) (grafoCiclo 10)
False
(51.90 secs, 12,841,861,176 bytes)
ghci> isomorfos2 (completo 10) (grafoCiclo 10)
False
(0.00 secs, 0 bytes)

ghci> isomorfos1 (grafoCiclo 10) (grafoRueda 10)
False
(73.90 secs, 16,433,969,976 bytes)
ghci> isomorfos2 (grafoCiclo 100) (grafoRueda 100)
False
(0.00 secs, 0 bytes)

```

*Nota 2.6.2.* Cuando los grafos son isomorfos, comprobar que tienen el mismo número de vértices, el mismo número de aristas y la misma secuencia gráfica no requiere mucho tiempo ni espacio, dando lugar a costes muy similares entre los dos pares de definiciones. Sin embargo, cuando los grafos tienen el mismo número de vértices y fallan en alguna de las demás propiedades, el resultado es muy costoso para la primera definición mientras que es inmediato con la segunda.

A partir de ahora utilizaremos la función `(isomorfismos2 g h)` para calcular los isomorfismos entre dos grafos y la función `(isomorfos g h)` para determinar si dos grafos son isomorfos, de modo que las renombraremos por `(isomorfismos g h)` y `(isomorfismos g h)` respectivamente.

```

isomorfismos :: (Ord a, Ord b) => Grafo a -> Grafo b -> [Funcion a b]
isomorfismos = isomorfismos2

isomorfos :: (Ord a, Ord b) => Grafo a -> Grafo b -> Bool
isomorfos = isomorfos2

```

### 2.6.3. Automorfismos

**Definición 2.6.9.** Dado un grafo simple  $G = (V, A)$ , un **automorfismo** de  $G$  es un isomorfismo de  $G$  en sí mismo.

La función `(esAutomorfismo g f)` se verifica si la aplicación  $f$  es un automorfismo de  $g$ .

```
ghci> esAutomorfismo (bipartitoCompleto 1 2) [(1,2),(2,3),(3,1)]
False
ghci> esAutomorfismo (bipartitoCompleto 1 2) [(1,1),(2,3),(3,2)]
True
ghci> esAutomorfismo (grafoCiclo 4) [(1,2),(2,3),(3,4),(4,1)]
True
```

```
esAutomorfismo :: Ord a => Grafo a -> Funcion a a -> Bool
esAutomorfismo g = esIsomorfismo g g
```

La función `(automorfismos g)` devuelve la lista de todos los posibles automorfismos en  $g$ . Por ejemplo,

```
ghci> automorfismos (grafoCiclo 4)
[[ (1,1), (2,2), (3,3), (4,4) ], [ (1,1), (2,4), (3,3), (4,2) ],
 [ (1,2), (2,1), (3,4), (4,3) ], [ (1,2), (2,3), (3,4), (4,1) ],
 [ (1,3), (2,2), (3,1), (4,4) ], [ (1,3), (2,4), (3,1), (4,2) ],
 [ (1,4), (2,1), (3,2), (4,3) ], [ (1,4), (2,3), (3,2), (4,1) ] ]
ghci> automorfismos (completo 4)
[[ (1,1), (2,2), (3,3), (4,4) ], [ (1,1), (2,2), (3,4), (4,3) ],
 [ (1,1), (2,3), (3,2), (4,4) ], [ (1,1), (2,3), (3,4), (4,2) ],
 [ (1,1), (2,4), (3,2), (4,3) ], [ (1,1), (2,4), (3,3), (4,2) ],
 [ (1,2), (2,1), (3,3), (4,4) ], [ (1,2), (2,1), (3,4), (4,3) ],
 [ (1,2), (2,3), (3,1), (4,4) ], [ (1,2), (2,3), (3,4), (4,1) ],
 [ (1,2), (2,4), (3,1), (4,3) ], [ (1,2), (2,4), (3,3), (4,1) ],
 [ (1,3), (2,1), (3,2), (4,4) ], [ (1,3), (2,1), (3,4), (4,2) ],
 [ (1,3), (2,2), (3,1), (4,4) ], [ (1,3), (2,2), (3,4), (4,1) ],
 [ (1,3), (2,4), (3,1), (4,2) ], [ (1,3), (2,4), (3,2), (4,1) ],
 [ (1,4), (2,1), (3,2), (4,3) ], [ (1,4), (2,1), (3,3), (4,2) ],
 [ (1,4), (2,2), (3,1), (4,3) ], [ (1,4), (2,2), (3,3), (4,1) ],
 [ (1,4), (2,3), (3,1), (4,2) ], [ (1,4), (2,3), (3,2), (4,1) ] ]
ghci> automorfismos (grafoRueda 5)
[[ (1,1), (2,2), (3,3), (4,4), (5,5) ], [ (1,1), (2,2), (3,5), (4,4), (5,3) ],
 [ (1,1), (2,3), (3,2), (4,5), (5,4) ], [ (1,1), (2,3), (3,4), (4,5), (5,2) ],
 [ (1,1), (2,4), (3,3), (4,2), (5,5) ], [ (1,1), (2,4), (3,5), (4,2), (5,3) ],
 [ (1,1), (2,5), (3,2), (4,3), (5,4) ], [ (1,1), (2,5), (3,4), (4,3), (5,2) ] ]
```

```

automorfismos :: Ord a => Grafo a -> [Funcion a a]
automorfismos g = isomorfismos1 g g

```

*Nota 2.6.3.* Cuando trabajamos con automorfismos, es mejor utilizar en su definición la función `isomorfismos` en vez de `isomorfismos2`, pues ser isomorfo es una relación reflexiva, es decir, un grafo siempre es isomorfo a sí mismo.

## 2.7. Conectividad de grafos

Una de las aplicaciones de la teoría de grafos es la determinación de trayectos o recorridos en una red de transporte o de distribución de productos. Así, si cada vértice representa un punto de interés y cada arista representa una conexión entre dos puntos, usando grafos como modelos, podemos simplificar el problema de encontrar la ruta más ventajosa en cada caso.

### 2.7.1. Caminos

**Definición 2.7.1.** Sea  $G = (V, A)$  un grafo simple y sean  $u, v \in V$  dos vértices. Un **camino** entre  $u$  y  $v$  es una sucesión de vértices de  $G$ :  $u = v_0, v_1, v_2, \dots, v_{k-1}, v_k = v$  donde  $\forall i \in \{0, \dots, k-1\}, (v_i, v_{i+1}) \in A$ .

La función `(esCamino g c)` se verifica si la sucesión de vértices `c` es un camino en el grafo `g`. Por ejemplo,

```

esCamino (grafoCiclo 5) [1,2,3,4,5,1] == True
esCamino (grafoCiclo 5) [1,2,4,5,3,1] == False
esCamino grafoThomson [1,2,3]          == False
esCamino grafoThomson [1,4,2,5,3,6]    == True

```

```

esCamino :: Ord a => Grafo a -> [a] -> Bool
esCamino g c = all (aristaEn g) (zip c (tail c))

```

La función `(aristasCamino c)` devuelve la lista de las aristas recorridas en el camino `c`.

```

ghci> aristasCamino [1,2,3,4,5,1]
[(1,2),(2,3),(3,4),(4,5),(5,1)]
ghci> aristasCamino [1,2,4,5,3,1]
[(1,2),(2,4),(4,5),(5,3),(3,1)]
ghci> aristasCamino [1,2,3]
[(1,2),(2,3)]
ghci> aristasCamino [1,4,2,5,3,6]
[(1,4),(4,2),(2,5),(5,3),(3,6)]

```

```

aristasCamino :: Ord a => [a] -> [(a,a)]
aristasCamino c =
    map parOrdenado (zip c (tail c))
    where parOrdenado (u,v) | u <= v = (u,v)
                          | otherwise = (v,u)

```

La función (`verticesCamino c`) devuelve la lista de las vertices recorridas en el camino `c`.

```

verticesCamino [1,2,3,4,5,1] == [1,2,3,4,5]
verticesCamino [1,2,4,5,3,1] == [1,2,4,5,3]
verticesCamino [1,2,3]       == [1,2,3]
verticesCamino []            == []

```

```

verticesCamino :: Ord a => [a] -> [a]
verticesCamino c = nub c

```

**Definición 2.7.2.** Sea  $G = (V, A)$  un grafo y sean  $u, v \in V$ . Un camino entre  $u$  y  $v$  que no repite aristas (quizás vértices) se llama **recorrido**.

La función (`esRecorrido g c`) se verifica si el camino `c` es un recorrido en el grafo `g`. Por ejemplo,

```

esRecorrido (grafoRueda 5) [1,2,3,4,1,2,5] == False
esRecorrido ['a'..'z'] == True
esRecorrido [1,2,4,6,4] == True
esRecorrido [1,2,1,3,4] == True

```

```

esRecorrido :: Ord a => [a] -> Bool
esRecorrido c =
    aristasCamino c == nub (aristasCamino c)

```

**Definición 2.7.3.** Un camino que no repite vértices (y, por tanto, tampoco aristas) se llama **camino simple**.

La función (`esCaminoSimple c`) se verifica si el camino `c` es un arco. Por ejemplo,

```

esCaminoSimple [1..4] == True
esCaminoSimple [1,2,6,1] == True
esCaminoSimple [1,2,3,1,4] == False
esCaminoSimple ['a'..'f'] == True

```

```

esCaminoSimple :: Ord a => [a] -> Bool
esCaminoSimple c = if (head c /= last c)
                    then (verticesCamino c == c)
                    else (verticesCamino c ++ [last c] == c)

```

**Definición 2.7.4.** Se llama *longitud* de un camino al número de veces que se atraviesa una arista en dicho camino.

La función (`longitudCamino c`) devuelve la longitud del camino `c`. Por ejemplo,

```

longitudCamino [1,2,3,4]    == 3
longitudCamino ['a'..'z']  == 25
longitudCamino [2,4..10]   == 4

```

```

longitudCamino :: Num b => [a] -> b
longitudCamino c = genericLength c - 1

```

La función (`todosCaminoBP g inicio final`) devuelve una lista con todos los caminos simples posibles en el grafo `g` entre los vértices `inicio` y `final`, utilizando un algoritmo de búsqueda en profundidad sobre el grafo `g`. Este algoritmo recorre el grafo de izquierda a derecha y de forma al visitar un nodo, explora todos los caminos que pueden continuar por él antes de pasar al siguiente.

```

ghci> todosCaminoBP (grafoCiclo 7) 1 6
[[1,7,6],[1,2,3,4,5,6]]
ghci> todosCaminoBP (grafoRueda 7) 2 5
[[2,1,5],[2,3,1,5],[2,3,4,1,5],[2,7,6,1,5],[2,7,1,5],[2,1,4,5],
 [2,3,1,4,5],[2,7,6,1,4,5],[2,7,1,4,5],[2,1,3,4,5],[2,7,6,1,3,4,5],
 [2,7,1,3,4,5],[2,3,4,5],[2,1,6,5],[2,3,1,6,5],[2,3,4,1,6,5],
 [2,7,1,6,5],[2,1,7,6,5],[2,3,1,7,6,5],[2,3,4,1,7,6,5],[2,7,6,5]]
ghci> todosCaminoBP (creaGrafo [1..4] [(1,2),(2,3)]) 1 4
[]

```

```

todosCaminoBP :: Ord a => Grafo a -> a -> a -> [[a]]
todosCaminoBP g x y = aux [[y]]
  where aux []          = []
        aux ([]:zss)    = aux zss
        aux ((z:zs):zss)
          | z == x       = (z:zs) : aux zss
          | otherwise    = aux ([v:z:zs | v <- adyacentes g' z \\ zs] ++ zss)
        g' = eliminaLazos g
        eliminaLazos h = creaGrafo (vertices h)
                                   [(x,y) | (x,y) <- aristas h, x /= y]

```



La función `(todosCaminosBA g inicio final)` devuelve una lista con todos los caminos simples posibles en el grafo `g` entre los vértices `inicio` y `final`, utilizando un algoritmo de búsqueda en anchura sobre el grafo `g`. Este algoritmo recorre el grafo por niveles, de forma que el primer camino de la lista es de longitud mínima.

```
ghci> todosCaminosBA (grafoCiclo 7) 1 6
[[1,7,6],[1,2,3,4,5,6]]
ghci> todosCaminosBA (grafoRueda 7) 2 5
[[2,1,5],[2,3,1,5],[2,7,1,5],[2,1,4,5],[2,3,4,5],[2,1,6,5],[2,7,6,5],
 [2,3,4,1,5],[2,7,6,1,5],[2,3,1,4,5],[2,7,1,4,5],[2,1,3,4,5],
 [2,3,1,6,5],[2,7,1,6,5],[2,1,7,6,5],[2,7,6,1,4,5],[2,7,1,3,4,5],
 [2,3,4,1,6,5],[2,3,1,7,6,5],[2,7,6,1,3,4,5],[2,3,4,1,7,6,5]]
ghci> todosCaminosBA (creaGrafo [1..4] [(1,2),(2,3)]) 1 4
[]
```

```
todosCaminosBA :: Ord a => Grafo a -> a -> a -> [[a]]
todosCaminosBA g x y = aux [[y]]
  where aux [] = []
        aux ([]:zss) = aux zss
        aux ((z:zs):zss)
          | z == x = (z:zs) : aux zss
          | otherwise = aux (zss ++ [v:z:zs | v <- adyacentes g' z \\\ zs])
        g' = eliminaLazos g
        eliminaLazos h = creaGrafo (vertices h)
                                   [(x,y) | (x,y) <- aristas h, x /= y]
```

Vamos a comprobar con QuickCheck que el primer elemento de la lista que devuelve la función `todosCaminosBA g u v` es de longitud mínima. Para ello, vamos a utilizar la función `(parDeVertices g)` que es un generador de pares de vértices del grafo no nulo `g`. Por ejemplo,

```
ghci> sample (parDeVertices (creaGrafo [1..9] []))
(3,9)
(9,3)
(7,4)
(4,3)
(2,8)
(7,2)
(8,4)
(5,3)
(7,2)
(3,1)
(7,2)
```

```

parDeVertices :: Grafo Int -> Gen (Int,Int)
parDeVertices g = do
  x <- elements vs
  y <- elements vs
  return (x,y)
  where vs = vertices g

```

*Nota 2.7.1.* Al hacer la comprobación, vamos a mostrar el orden de los grafos que genera automáticamente quickCheck para asegurarnos de que la comprobación sea suficientemente significativa.

```

ghci> quickCheckWith (stdArgs maxSize=15) prop_todosCaminosBA
+++ OK, passed 100 tests:
23% 1
18% 3
17% 2
 9% 6
 8% 4
 7% 7
 5% 8
 5% 5
 4% 9
 2% 12
 1% 11
 1% 10

```

```

prop_todosCaminosBA :: Grafo Int -> Property
prop_todosCaminosBA g =
  not (esGrafoNulo g) ==>
  collect (orden g) $
  forAll (parDeVertices g)
    (\(x,y) -> let zss = todosCaminosBA g x y
                in null zss || longitudCamino (head zss) ==
                    minimum (map longitudCamino zss))

```

Ver “Sobre caminos y comprobaciones”

Revisado hasta aquí

**Definición 2.7.5.** Dado un grafo  $G = (V, A)$ , sean  $u, v \in V$ . Si existe algún camino entre  $u$  y  $v$  en el grafo  $G$  diremos que están **conectados** y lo denotamos por  $u \sim v$ .

La función `(estanConectados g u v)` se verifica si los vértices  $u$  y  $v$  están conectados en el grafo  $g$ .

```
ghci> estanConectados (grafoCiclo 7) 1 6
True
ghci> estanConectados (creaGrafo [1..4] [(1,2),(2,3)]) 1 4
False
ghci> estanConectados grafoNulo 1 4
False
```

```
estanConectados :: Ord a => Grafo a -> a -> a -> Bool
estanConectados g u v | esGrafoNulo g = False
                      | otherwise = not (null (todosCaminosBA g u v))
```

*Nota 2.7.2.* La función `(estanConectados g u v)` no necesita calcular todos los caminos entre  $u$  y  $v$ . Puesto que Haskell utiliza por defecto evaluación perezosa, si existe algún camino entre los dos vértices, basta calcular el primero para saber que la lista de todos los caminos no es vacía

**Definición 2.7.6.** Se define la *distancia* entre  $u$  y  $v$  en el grafo  $G$  como la longitud del camino más corto que los une. Si  $u$  y  $v$  no están conectados, decimos que la distancia es infinita.

La función `(distancia g u v)` devuelve la distancia entre los vértices  $u$  y  $v$  en el grafo  $g$ . En caso de que los vértices no estén conectados devuelve el valor `Nothing`. Por ejemplo,

```
distancia (grafoCiclo 7) 1 4 == Just 3
distancia (grafoRueda 7) 2 5 == Just 2
distancia (grafoEstrella 4) 1 3 == Just 1
distancia (creaGrafo [1..4] [(1,2),(2,3)]) 1 4 == Nothing
distancia grafoNulo 2 3 == Nothing
```

```
distancia :: Ord a => Grafo a -> a -> a -> Maybe Int
distancia g u v | estanConectados g u v =
    Just (longitudCamino (head (todosCaminosBA g u v)))
    | otherwise = Nothing
```

**Definición 2.7.7.** Dado  $G = (V, A)$  un grafo, sean  $u, v \in V$ . Un camino entre  $u$  y  $v$  cuya longitud coincide con la distancia entre los vértices se llama *geodésica* entre  $u$  y  $v$ .

La función `(esGeodesica g c)` se verifica si el camino  $c$  es una geodésica entre sus extremos en el grafo  $g$ .

```
esGeodesica (grafoCiclo 7) [1,2,3,4,5,6] == False
esGeodesica (grafoCiclo 7) [1,7,6] == True
esGeodesica (grafoRueda 7) [2,1,5] == True
esGeodesica (creaGrafo [1..4] [(1,2),(2,3)]) [1,4] == False
esGeodesica grafoNulo [1,4] == False
```

```

esGeodesica :: Ord a => Grafo a -> [a] -> Bool
esGeodesica g c =
    longitudCamino c == fromJust (distancia g u v)
    where u = head c
          v = last c

```

**Definición 2.7.8.** Un camino en un grafo  $G$  se dice **cerrado** si sus extremos son iguales.

La función (`esCerrado g c`) se verifica si el camino  $c$  es cerrado en el grafo  $g$ . Por ejemplo,

```

esCerrado (grafoCiclo 5) [1,2,3,4,5,1] == True
esCerrado (grafoCiclo 5) [1,2,4,5,3,1] == False
esCerrado grafoThomson [1,4,2,5,3,6]    == False
esCerrado grafoThomson [1,4,2,5,3,6,1]  == True
esCerrado grafoNulo [1,2,1]             == False

```

```

esCerrado :: (Ord a) => Grafo a -> [a] -> Bool
esCerrado g c = head c == last c

```

**Definición 2.7.9.** Un recorrido en un grafo  $G$  se dice **circuito** si sus extremos son iguales.

La función (`esCircuito g c`) se verifica si la sucesión de vértices  $c$  es un circuito en el grafo  $g$ . Por ejemplo,

```

esCircuito (grafoCiclo 5) [1,2,3,4,5,1] == True
esCircuito (grafoCiclo 3) [1,2,3,1,2,4,1] == False
esCircuito grafoThomson [1,4,2,5,3,6]    == False
esCircuito grafoThomson [1,4,2,5,3,6,1]  == True
esCircuito grafoNulo [1,2,1]             == False

```

```

esCircuito :: (Ord a) => Grafo a -> [a] -> Bool
esCircuito g c =
    esRecorrido c && esCerrado g c

```

**Definición 2.7.10.** Un caminoSimple en un grafo  $G$  se dice que es un **circuito** si sus extremos son iguales.

La función (`esCiclo g c`) se verifica si el camino  $c$  es un ciclo en el grafo  $g$ . Por ejemplo,

```

esCiclo (grafoCiclo 5) [1,2,3,4,5,1] == True
esCiclo (grafoCiclo 3) [1,2,3,1,2,4,1] == False
esCiclo grafoThomson [1,4,2,5,3,6] == False
esCiclo grafoThomson [1,4,2,5,3,6,1] == True
esCiclo grafoNulo [1,2,1] == False

```

```

esCiclo :: (Ord a) => Grafo a -> [a] -> Bool
esCiclo g c =
    esCaminoSimple c && esCerrado g c

```

**Teorema 2.7.11.** *Dado un grafo  $G$ , la relación  $u \sim v$  (estar conectados por un camino) es una relación de equivalencia.*

La función (`estarConectadosCamino g`) devuelve la relación entre los vértices del grafo  $g$  de estar conectados por un camino en él.

```

estarConectadosCamino :: Ord a => Grafo a -> [(a,a)]
estarConectadosCamino g =
    [(u,v) | u <- vs, v <- vs, estanConectados g u v]
    where vs = vertices g

```

A continuación, comprobaremos el resultado con `quickCheck`.

```

ghci> quickCheckWith (stdArgs maxSize=50) prop_conectadosRelEqui
+++ OK, passed 100 tests.

```

```

prop_conectadosRelEqui :: Grafo Int -> Bool
prop_conectadosRelEqui g =
    esRelacionEquivalencia (vertices g) r
    where r = estarConectadosCamino g

```

**Definición 2.7.12.** *Las clases de equivalencia obtenidas por la relación  $\sim$ , estar conectados por un camino en un grafo  $G$ , inducen subgrafos en  $G$ , los vértices y todas las aristas de los caminos que los conectan, que reciben el nombre de **componentes conexas por caminos** de  $G$ .*

La función (`componentesConexas g`) devuelve las componentes conexas por caminos del grafo  $g$ . Por ejemplo,

```
ghci> componentesConexas grafoPetersen
[[1,2,3,4,5,6,7,8,9,10]]
ghci> componentesConexas (creaGrafo [1..5] [(1,2),(2,3)])
[[1,2,3],[4],[5]]
ghci> componentesConexas (creaGrafo [1..5] [(1,2),(2,3),(4,5)])
[[1,2,3],[4,5]]
ghci> componentesConexas grafoNulo
[]
```

```
componentesConexas :: Ord a => Grafo a -> [[a]]
componentesConexas g =
    clasesEquivalencia vs (estarConectadosCamino g)
    where vs = vertices g
```

**Definición 2.7.13.** Dado un grafo, diremos que es **conexo** si la relación tiene una única clase de equivalencia en él; es decir, si el grafo tiene una única componente conexas.

La función (`esConexo g`) se verifica si el grafo `g` es conexo. Por ejemplo,

```
esConexo (completo 5) == True
esConexo (creaGrafo [1..5] [(1,2),(2,3)]) == False
esConexo (creaGrafo [1..3] [(1,2),(2,3)]) == True
esConexo grafoNulo == False
```

```
esConexo :: Ord a => Grafo a -> Bool
esConexo g = length (componentesConexas g) == 1
```

**Teorema 2.7.14.** Sea  $G$  un grafo,  $G = (V, A)$  es conexo si y solamente si for all  $u, v \in V$  existe un camino entre  $u$  y  $v$ .

Vamos a comprobar el resultado con `quickCheck`

```
ghci> quickCheck prop_caracterizaGrafoConexo
+++ OK, passed 100 tests.
```

```
prop_caracterizaGrafoConexo :: Grafo Int -> Property
prop_caracterizaGrafoConexo g =
    esConexo g ==> and [estanConectados g u v |
                        u <- vertices g, v <- vertices g]
    && and [estanConectados g u v |
            u <- vertices g, v <- vertices g] ==> esConexo g
```

**Definición 2.7.15.** Sean  $G = (V, A)$  un grafo y  $v \in V$ . Se define la **excentricidad** de  $v$  como el máximo de las distancias entre  $v$  y el resto de vértices de  $G$ . La denotaremos por  $e(G)$ .

La función `(excentricidad g v)` devuelve la excentricidad del vértice  $v$  en el grafo  $g$ . Por ejemplo,

```
excentricidad (grafoCiclo 8) 5      == Just 4
excentricidad (grafoRueda 7) 4      == Just 2
excentricidad (grafoRueda 7) 1      == Just 1
excentricidad grafoPetersen 6       == Just 2
ghci> let g = creaGrafo [1,2,3] [(1,2)]
excentricidad g 3                    == Nothing
excentricidad grafoNulo 3            == Nothing
```

```
excentricidad :: Ord a => Grafo a -> a -> Maybe Int
excentricidad g u
  | esGrafoNulo g = Nothing
  | otherwise = aux [fromJust (d v) | v <- vs, isJust (d v)]
  where vs = vertices g \\ [u]
        d w = distancia g u w
        aux [] = Nothing
        aux ds = Just (maximum ds)
```

**Definición 2.7.16.** Sea  $G = (V, A)$  un grafo. Se define el **diámetro** de  $G$  como el máximo de las distancias entre los vértices en  $V$ . Lo denotaremos por  $d(G)$ .

La función `(diametro g)` devuelve el diámetro del grafo  $g$ . Por ejemplo,

```
diametro (grafoCiclo 8)      == Just 4
diametro (grafoRueda 7)     == Just 2
diametro grafoPetersen      == Just 2
ghci> let g = creaGrafo [1,2,3] [(1,2)]
diametro g                   == Just 1
diametro grafoNulo           == Nothing
```

```
diametro :: Ord a => Grafo a -> Maybe Int
diametro g | esGrafoNulo g = Nothing
           | otherwise = aux [excentricidad g v | v <- vs]
  where vs = vertices g
        aux [] = Nothing
        aux xs = Just (maximum (map fromJust (filter isJust xs)))
```

**Definición 2.7.17.** Sean  $G = (V, A)$  un grafo y  $v \in V$ . Se define el **radio** de  $G$  como el mínimo de las excentricidades de sus vértices. Lo denotaremos por  $r(G)$ .

La función (`radio g`) devuelve el radio del grafo `g`. Por ejemplo,

```
radio (grafoCiclo 8)      == Just 4
radio (grafoRueda 7)      == Just 1
radio grafoPetersen       == Just 2
ghci> let g = creaGrafo [1,2,3] [(1,2)]
radio g                   == Just 1
radio grafoNulo           == Nothing
```

```
radio :: Ord a => Grafo a -> Maybe Int
radio g | esGrafoNulo g = Nothing
        | otherwise = aux [excentricidad g v | v <- vertices g]
  where aux [] = Nothing
        aux xs = Just (minimum (map fromJust (filter isJust xs)))
```

**Definición 2.7.18.** Sean  $G = (V, A)$  un grafo. Llamamos **centro** del grafo  $G$  al conjunto de vértices de excentricidad mínima. A estos vértices se les denomina **vértices centrales**.

La función (`centro g`) devuelve el centro del grafo `g`. Por ejemplo,

```
centro (grafoEstrella 5) == [1]
centro (grafoCiclo 4)   == [1,2,3,4]
centro grafoPetersen    == [1,2,3,4,5,6,7,8,9,10]
centro (grafoRueda 5)   == [1]
centro grafoNulo        == []
```

```
centro :: Ord a => Grafo a -> [a]
centro g = [v | v <- vertices g, r == excentricidad g v]
  where r = radio g
```

**Definición 2.7.19.** Sean  $G = (V, A)$  un grafo. Se llama **grosor** o **cintura** del grafo  $G$  como el máximo de las longitudes de los ciclos de  $G$ .



## 2.8. Sistemas utilizados

El desarrollo de mi Trabajo de Fin de Grado requería de una infraestructura técnica que he tenido que trabajar antes de comenzar a desarrollar el contenido. A continuación, voy a nombrar y comentar los sistemas y paquetes que he utilizado a lo largo del proyecto.

- **Ubuntu como sistema operativo.** El primer paso fue instalar *Ubuntu* en mi ordenador portátil personal. Para ello, seguí las recomendaciones de mi compañero Eduardo Paluzo, que ya lo había hecho antes.

Primero, me descargué la imagen del sistema *Ubuntu 16.04 LTS* (para procesador de 64 bits) desde la [página de descargas de Ubuntu](http://www.ubuntu.com/download/desktop) <sup>16</sup> y también la herramienta [Linux-Live USB Creator](http://www.linuxliveusb.com/) <sup>17</sup> que transformaría mi Pendrive en una unidad USB Booteable cargada con la imagen de Ubuntu. Una vez tuve la unidad USB preparada, procedí a instalar el nuevo sistema: apagué el dispositivo y al encenderlo entré en el Boot Menu de la BIOS del portátil para arrancar desde el Pendrive en vez de hacerlo desde el disco duro. Automáticamente, comenzó la instalación de *Ubuntu* y solo tuve que seguir las instrucciones del asistente para montar Ubuntu manteniendo además *Windows 10*, que era el sistema operativo con el que había estado trabajando hasta ese momento.

El resultado fue un poco agridulce, pues la instalación de Ubuntu se había realizado con éxito, sin embargo, al intentar arrancar *Windows* desde la nueva GRUB, me daba un error al cargar la imagen del sistema. Después de buscar el error que me aparecía en varios foros, encontré una solución a mi problema: deshabilité el Security Boot desde la BIOS y pude volver a arrancar *Windows 10* con normalidad.

- **L<sup>A</sup>T<sub>E</sub>X como sistema de composición de textos.** La distribución de L<sup>A</sup>T<sub>E</sub>X, *Tex Live*, como la mayoría de software que he utilizado, la descargué utilizando el *Gestor de Paquetes Synaptic*. Anteriormente, sólo había utilizado *TexMaker* como editor de L<sup>A</sup>T<sub>E</sub>X así que fue el primero que descargué. Más tarde, mi tutor José Antonio me sugirió que mejor descargara el paquete *AUCTex*, pues me permitiría trabajar con archivos T<sub>E</sub>X desde el editor *Emacs*, así lo hice y es el que he utilizado para escribir el trabajo. Además de los que me recomendaba el gestor, solo he tenido que descargarme el paquete *spanish* de *babel* para poder desarrollar el Trabajo, pues el paquete *Tikz*, que he utilizado para representar los grafos, venía incluido en las sugerencias de *Synaptic*.
- **Haskell como lenguaje de programación.** Ya había trabajado anteriormente con este lenguaje en el Grado y sabía que sólo tenía que decargarme la plataforma de *Haskell* y podría trabajar con el editor *Emacs*. Seguí las indicaciones que se dan a los

<sup>16</sup><http://www.ubuntu.com/download/desktop>

<sup>17</sup><http://www.linuxliveusb.com/>

estudiantes de primer curso en la [página del Dpto. de Ciencias de la Computación e Inteligencia Artificial](#)<sup>18</sup> y me descargué los paquetes *haskell-platform* y *haskell-mode* desde el *Gestor de Paquetes Synaptic*.

Indicar la versión de la plataforma y de GHC que estás utilizando.

- **Dropbox como sistema de almacenamiento compartido.** Ya había trabajado con *Dropbox* en el pasado, así que crear una carpeta compartida con mis tutores no fue ningún problema; sin embargo, al estar *Dropbox* sujeto a software no libre, no me resultó tan sencillo instalarlo en mi nuevo sistema. En primer lugar, intenté hacerlo directamente desde *Ubuntu Software*, que intentó instalar *Dropbox Nautilus* y abrió dos instalaciones en paralelo. Se quedó colgado el ordenador, así que maté los procesos de instalación activos, reinicié el sistema y me descargué directamente el paquete de instalación desde la [página de descargas de Dropbox](#)<sup>19</sup> para ejecutarlo desde la terminal.

Queda pendiente ampliar el apéndice conforme vayamos avanzando y surjan nuevos sistemas.

## 2.9. Mapa de decisiones de diseño

Al comienzo del proyecto, la idea era que las primeras representaciones con las que trabajara fueran las de *grafos como vectores de adyacencia* y *grafos como matrices de adyacencia* que se utilizan en Informática en el primer curso del Grado, con las que ya había trabajado y estaba familiarizada.

Las definiciones de Informática están pensadas para grafos ponderados (dirigidos o no según se eligiera), mientras que en Matemática Discreta apenas se usan grafos dirigidos o ponderados; por tanto, el primer cambio en la representación utilizada fue simplificar las definiciones de modo que solo trabajáramos con grafos no dirigidos y no ponderados, pero manteniendo las estructuras vectorial y matricial que mantenían la eficiencia.

Las representaciones que utilizan *arrays* en *Haskell* son muy restrictivas, pues solo admiten vectores y matrices que se puedan indexar, lo que hace muy complicados todos los algoritmos que impliquen algún cambio en los vértices de los grafos y, además, no permite trabajar con todos los tipos de vértices que pudiéramos desear. Decidimos volver a cambiar la representación, y esta vez nos decantamos por la representación de *grafos como listas de aristas*, perdiendo en eficiencia pero ganando mucho en flexibilidad de escritura.

<sup>18</sup><http://www.cs.us.es/~jalonso/cursos/i1m-15/sistemas.php>

<sup>19</sup><https://www.dropbox.com/es/install?os=lnx>

Queda pendiente ampliar el apéndice conforme vayamos avanzando y surjan nuevas representaciones.



# Bibliografía

- [1] J. Alonso. [Temas de programación funcional](#). Technical report, Univ. de Sevilla, 2015.
- [2] M. Cardenas. [Matemática discreta](#). Technical report, Univ. de Sevilla, 2015.
- [3] F. Rabhi and G. Lapalme. *Algorithms: A functional programming approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999.
- [4] Wikipedia. [Anexo:Galería de grafos](#). Technical report, Wikipedia, La Enciclopedia libre., 2016.

# Índice alfabético

Grafo, 33  
adyacentes, 34  
antiImagenRelacion, 18  
aristaEn, 34  
aristasCamino, 63  
aristas, 34  
automorfismos, 61  
bipartitoCompleto, 40  
biyecciones, 27  
cardinal, 14  
centro, 72  
clasesEquivalencia, 24  
combinaciones, 15  
complementario, 13, 54  
completo, 39  
componentesConexas, 70  
conjuntoVacio, 11  
conjuntosIguales, 13  
conservaAdyacencia, 55  
creaGrafo, 33  
diametro, 71  
distancia, 67  
dominio, 17  
eliminaArista, 51  
eliminaVertice, 51  
entorno, 46  
esAislado, 47  
esAntisimetrica, 21  
esAutomorfismo, 61  
esCamino, 62  
esCerrado, 68  
esCircuito, 68, 69  
esConexo, 70  
esFuncional, 18  
esFuncion, 24  
esGeodesica, 68  
esInyectiva, 26  
esIsomorfismo, 57  
esLazo, 46  
esReflexiva, 20  
esRegular, 47  
esRelacionEquivalencia, 22  
esRelacionHomogenea, 19  
esRelacionOrden, 23  
esRelacion, 16  
esSimetrica, 20  
esSimple, 48  
esSobreyectiva, 26, 27  
esSubconjuntoPropio, 13  
esSubconjunto, 12  
esSubgrafoMax, 49  
esSubgrafoPropio, 50  
esSubgrafo, 49  
esTransitiva, 22  
esVacio, 11  
estaRelacionado, 19  
estanConectados, 67  
excentricidad, 71  
funciones, 25  
generaGrafo, 35  
grado, 46  
grafoAmistad, 38  
grafoCiclo, 37  
grafoCirculante, 41

grafoEstrella, 40  
grafoMoebiusCantor, 44  
grafoNulo, 37  
grafoPetersenGen, 42  
grafoPetersen, 43  
grafoRueda, 41  
grafoThomson, 43  
imagenInversa, 28  
imagenRelacion, 17  
imagen, 25  
interseccion, 15  
inversa, 28  
isomorfismos, 57  
isomorfos, 58  
longitudCamino, 64  
morfismos, 56  
orden, 45  
productoCartesiano, 15  
prop\_HavelHakimi, 50  
prop\_LemaApretonDeManos, 50  
prop\_completos, 53  
prop\_ConectadosRelEqui, 69  
prop\_todosCaminosBA, 66  
radio, 72  
rango, 17  
secuenciaGrados, 48  
secuenciaGrafica, 49  
sonIncidentes, 45  
sumaArista, 52  
sumaGrafos, 53, 54  
sumaVertice, 52  
tamaño, 45  
todosCaminos, 64, 65  
unionConjuntos, 14  
unitario, 12  
valenciaMax, 48  
valenciaMin, 47  
variaciones, 16  
verticesCamino, 63  
vertices, 34  
textttparDeVertices, 65





# Lista de tareas pendientes

■ Darle formato, ampliar y organizar organizar conjuntos. . . . .	10
■ Añadir referencias de conjuntos a la bibliografía. . . . .	10
■ Comentar las distintas opciones de representación de conjuntos del tema 17 de IIM . . . . .	10
■ Comentar la representación elegida de conjunto. . . . .	10
■ Decidir si usar la misma representación que en GrafoConListaDeAristas.lhs (listas ordenadas sin repeticiones)) . . . . .	10
■ Comentar la posibilidad de usar la librería de conjuntos de Haskell (Data.Set) . . . . .	10
■ Comentar la consecuencia sobre los tipos de la última frase de la definición de conjunto. . . . .	10
■ En la definición de esUnitario se puede eliminar nub si se supone que trabajamos con conjuntos. . . . .	12
■ La sección de relaciones homogéneas no debe de depender de grafo. Por tanto, hay que eliminar sus importaciones. . . . .	18
■ Añadirlo al fichero de bibliografía y poner la cita. . . . .	18
■ En la definición de inversa se puede eliminar nub, suponiendo que f es un conjunto. . . . .	28
■ Pendiente de ampliar la introducción conforme se vaya escribiendo los módulos. . . . .	29
■ Comentar las ventajas de usar listas en lugar de vectores. . . . .	33
■ Añadir al fichero de bibliografía y poner la cita. . . . .	35
■ Añadir al fichero de bibliografía y citarla. . . . .	36
■ Ir actualizando y completando las fuentes y cambiar el formato de enumeración . . . . .	36
■ En este módulo hay un ejemplo de cómo dibujar un grafo cualquiera (grafo de la amistad) . . . . .	36
■ Añadir al fichero de bibliografía y citar. . . . .	44
■ A partir de esta propiedad, se puede dar una definición alternativa de $K_n$ (completo2) y comprobar su equivalencia con la primera (completo). . . . .	53
■ Buscar definiciones más eficientes de morfismos. . . . .	56
■ Introducir el problema de decidir si dos grafos son homomorfos. . . . .	56
■ Ver “Sobre caminos y comprobaciones” . . . . .	66

■ Revisado hasta aquí . . . . .	66
■ Indicar la versión de la plataforma y de GHC que estás utilizando. . . . .	74
■ Queda pendiente ampliar el apéndice conforme vayamos avanzando y surjan nuevos sistemas. . . . .	74
■ Queda pendiente ampliar el apéndice conforme vayamos avanzando y surjan nuevas representaciones. . . . .	74