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 15 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. Debe reconocer los créditos de la obra de la manera especificada por el autor.



No comercial. No puede utilizar esta obra para fines comerciales.



Compartir bajo la misma licencia. Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.

- Al reutilizar o distribuir la obra, tiene que dejar bien claro los términos de la licencia de esta obra.
- Alguna de estas condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de autor.

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 envie una carta a Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Índice general

Introducción				5	
1	Introducción a la teoría de grafos			7	
	1.1	Defini	ición de grafo	8	
	1.2		D de los grafos	8	
			Grafos como listas de aristas	10	
	1.3	Gener	radores de grafos	12	
	1.4	Ejemplos de grafos		13	
		1.4.1	Grafo ciclo	14	
		1.4.2	Grafo de la amistad	14	
		1.4.3	Grafo completo	15	
		1.4.4	Grafo bipartito	16	
		1.4.5	Grafo estrella	17	
		1.4.6	Grafo rueda	17	
		1.4.7	Grafo circulante	18	
		1.4.8	Otros grafos importantes	20	
	1.5	Defini	iciones y propiedades	22	
		1.5.1	Definiciones de grafos	22	
		1.5.2	Propiedades de grafos	28	
		1.5.3	Operaciones y propiedades sobre grafos	29	
	1.6	Morfis	smos de grafos	33	
		1.6.1	Conceptos previos	34	
		1.6.2	Morfismo	40	
		1.6.3	Isomorfismo	41	
		1.6.4	Automorfismo	43	
		1.6.5	Invariantes por isomorfismos	44	
	1.7	7 Sistemas utilizados		46	
	1.8	.8 Mapa de decisiones de diseño		47	
Bi	Bibliografía				

4 Índice general

Indice de definiciones

49

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

 $^{^{1}} https://dl.dropboxusercontent.com/u/15420416/tiddly/emptyMD1314.html\\$

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

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

⁴https://en.wikipedia.org/wiki/Graph_theory

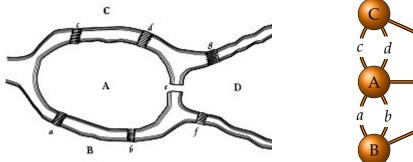
<u>Índice general</u>

Capítulo 1

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 atraviese 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.



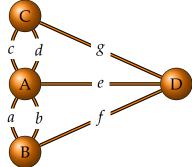


Figura 1.1: Dibujo de los puentes de Königsberg

Figura 1.2: Modelo de los puentes de Königsberg

Desde entonces, se ha ido desarrollando esta metodología hasta convertise 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.

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

1.1. Definición de grafo

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

Definición 1.1.1. Un grafo G es un par (V, A), donde V es el conjunto de los vértices (o nodos) y A el de las aristas.

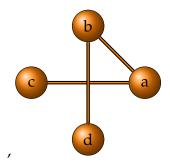
Definición 1.1.2. *Una arista* de un grafo G = (V, A), es un conjunto de dos elementos de V. Es decir, para dos vértices v, v' de G, (v, v') y (v', v) representa la misma arista.

Definición 1.1.3. 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 1.1.4. Si en un grafo dirigido se permiten aristas repetidas, lo llamaremos **multigrafo**. Si no se permiten, lo llamaremos **grafo regular**.

Nota 1.1.1. Denotaremos por |V| al número de vértices y por |A| al número de aristas del grafo (V, A).

Ejemplo 1.1.2. 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.



1.2. 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 $V=\{v_1,\ldots,v_n\}$ 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)]
```

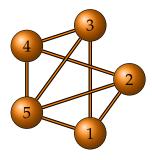
donde:

- (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 1.2.1. Las funciones que aparecen en la especificación del TAD no dependen de la representación que elijamos.

Ejemplo 1.2.2. 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)]
```



1.2.1. Grafos como listas de aristas

En el módulo GrafoConListaDeAristas se definen las funciones del TAD de los grafos dando su representación como listas de aristas; es decir, representando a un grafo como dos listas, la primera será la lista de los vértices y la segunda la de las aristas.

Nota 1.2.3. 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 tiene acceso constante (O(1)) y las listas lineal (O(n)).

```
module GrafoConListaDeAristas
  ( Grafo
  , 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)]
  ) where
```

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

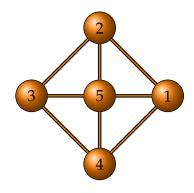
Vamos a definir un nuevo tipo de dato (Grafo a), que representará un grafo a partir de la lista de sus vértices (donde los vértices son de tipo a) y de aristas (que son pares de vértices).

```
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.

Ejemplo 1.2.4. 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 (5,1) ejGrafo == True
aristaEn (3,1) ejGrafo == False

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

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

```
[(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
```

1.3. Generadores de grafos

ghci> aristas ejGrafo

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.

Comentario: Falta referencia al generador de grafos (generaGrafos) es un generador de grafos de hasta 10 vértices. Por ejemplo,

```
ghci> sample generaGrafo
G[1,2,3,4,5,6,7,8,9][(1,1),(1,3),(1,5),(1,6),(1,7),(1,9),(2,6),(2,8),
                       (2,9),(3,4),(4,6),(4,7),(5,7),(5,9),(6,6),(6,7),
                       (7,7), (7,8), (7,9), (8,8), (8,9), (9,9)
G[1,2][(1,1),(2,2)]
G[1,2,3][(1,3),(2,2)]
G[1,2,3,4,5,6,7,8][(1,1),(1,5),(1,6),(1,7),(2,2),(2,5),(2,8),(3,3),
                     (3,5),(3,6),(3,8),(4,6),(5,6),(5,7),(5,8),(6,6),
                     (6,7)
G[1,2,3,4,5,6,7][(1,1),(1,3),(1,4),(1,7),(2,2),(2,5),(2,7),(3,4),
                   (3,5),(3,6),(4,4),(4,5),(4,6),(4,7),(5,6),(5,7),
                   (6,6),(6,7),(7,7)
G[1,2][(1,1),(1,2)]
G [1] []
G[1,2,3][(1,1),(1,2),(2,3),(3,3)]
G [1,2] [(1,1),(2,2)]
```

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

```
generaGrafo :: Gen (Grafo Int)
generaGrafo = do
    n <- choose (1,10)
    as <- sublistOf [(x,y) | x <- [1..n], y <- [x..n]]
    return (creaGrafo [1..n] as)</pre>
```

Nota 1.3.1. Los grafos están contenido en la clase de los objetos generables aleatoriamente.

```
instance Arbitrary (Grafo Int) where
arbitrary = generaGrafo
```

1.4. 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 ¹ de la Wikipedia.

Comentario: 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 1.4.1. Se utilizará la representación de los grafos como listas de aristas.

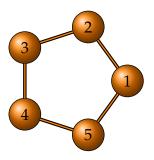
¹https://es.wikipedia.org/wiki/Anexo:Galería_de_grafos

1.4.1. Grafo ciclo

Definición 1.4.1. Un ciclo, ² de orden n, C(n), es un grafo no dirigido y no ponderado cuyo conjunto de vértices viene dado por $V = \{1, ..., n\}$ y el de las aristas por $A = \{(0, 1), (1, 2), ..., (n - 2, n - 1), (n - 1, 0)\}$

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

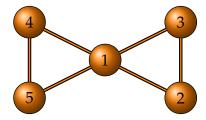
```
ghci> grafoCiclo 5
G (array (1,5) [(1,[5,2]),(2,[1,3]),(3,[2,4]),(4,[3,5]),(5,[4,1])])
```



1.4.2. Grafo de la amistad

Definición 1.4.2. Un grafo de la amistad 3 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 .

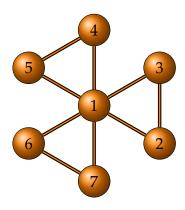
La función (grafoAmistad n) genera el grafo de la amistad de orden n. Por ejemplo,



²https://es.wikipedia.org/wiki/Grafo_completo

³https://es.wikipedia.org/wiki/Grafo_de_la_amistad

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



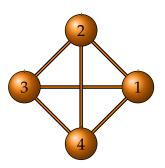
1.4.3. Grafo completo

Definición 1.4.3. El grafo completo, 4 de orden n, K(n), es un grafo no dirigido cuyo conjunto de vértices viene dado por $V = \{1, ..., n\}$ y tiene una arista entre cada par de vértices distintos.

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])])
```

⁴https://es.wikipedia.org/wiki/Grafo_completo



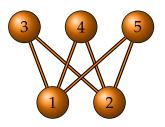
```
completo :: Int -> Grafo Int
completo n =
    creaGrafo [1..n]
        [(a,b) | a <- [1..n], b <- [1..a-1]]</pre>
```

1.4.4. Grafo bipartito

Definición 1.4.4. Un grafo bipartito ⁵ 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 ⁶ 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,



 $^{^{5}}$ https://es.wikipedia.org/wiki/Grafo_bipartito

⁶https://es.wikipedia.org/wiki/Grafo_bipartito_completo

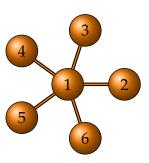
```
bipartitoCompleto :: Int -> Int -> Grafo Int
bipartitoCompleto n m =
    creaGrafo [1..n+m]
    [(a,b) | a <- [1..n], b <- [n+1..n+m]]</pre>
```

1.4.5. Grafo estrella

Definición 1.4.5. Una estrella 7 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
```

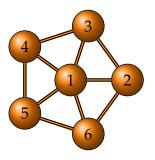
1.4.6. Grafo rueda

Definición 1.4.6. Un grafo rueda 8 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 .

⁷https://en.wikipedia.org/wiki/Star_(graph_theory))

⁸https://es.wikipedia.org/wiki/Grafo_rueda

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

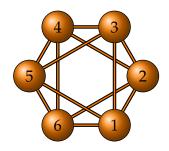


1.4.7. Grafo circulante

Definición 1.4.7. Un grafo circulante 9 de orden $n \ge 3$ y saltos $\{s_1, \ldots, s_k\}$ es un grafo no dirigido y no ponderado $G = (\{1, \ldots, n\}, A)$ en el que cada nodo $\forall i \in V$ es adyacente a los 2k nodos $i \pm s_1, \ldots, i \pm s_k \mod n$. Lo denotaremos por $Cir_n^{s_1, \ldots, s_k}$.

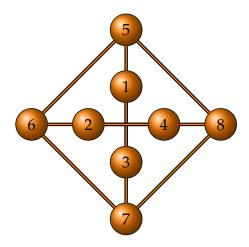
La función (grafoCirculante n ss) crea un grafo circulante a partir de su orden n y de la lista de sus saltos ss. Por ejemplo,

⁹https://en.wikipedia.org/wiki/Circulant_graph



El **grafo de Petersen generalizado** ¹⁰ que denotaremos $GP_{n,k}$ (con $n \ge 3$ y $1 \le k \le (n-1)/2$) es un grafo formado por un grafo circulante $Cir^n_{\{k\}}$ 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 (grafo Petersen
Gen n k) devuelve el grafo de Petersen generalizado $GP_{n,k}.$



```
ghci> grafoPetersenGen 4 2

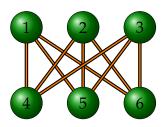
G (array (1,8) [(1,[3,3,5]),(2,[4,4,6]),(3,[1,1,7]),(4,[2,2,8]),

(5,[1,8,6]),(6,[2,5,7]),(7,[3,6,8]),(8,[4,7,5])])
```

¹⁰https://en.wikipedia.org/wiki/Generalized_Petersen_graph

1.4.8. Otros grafos importantes

Definición 1.4.8. 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.



La función (graf oThomson) genera el grafo de Thomson.

```
ghci> grafoThomson

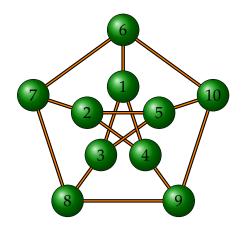
G (array (1,6) [(1,[4,5,6]),(2,[4,5,6]),(3,[4,5,6]),

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

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

El **grafo de Petersen** ¹¹ 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.

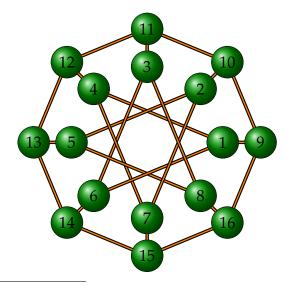
¹¹https://en.wikipedia.org/wiki/Petersen_graph



La función grafoPetersen devuelve el grafo de Petersen.

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

El grafo de Moëbius-Cantor 12 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.



¹²https://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
```

1.5. 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 ¹³ de Matemática Discreta.

Nota 1.5.1. Se utilizará el tipo abstracto de grafos presentados en la sección 1.2 y se utilizarán las librerias Data. List y Test. QuickCheck.

1.5.1. Definiciones de grafos

Definición 1.5.1. El **orden** de un grafo G = (V, G) se define como su número de vértices. Lo denotaremos por |V(G)|.

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 g = length (vertices g)
```

¹³https://dl.dropboxusercontent.com/u/15420416/tiddly/emptyMD1314.html

Definición 1.5.2. *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 1.5.3. 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 1.5.4. 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 $N(v) = \{u \in V | (u, v) \in A\}$.

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 1.5.5. Sea G = (V, A) un grafo. El **grado** (o **valencia**) de $v \in V$ es 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 1.5.6. *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 :: Eq a => Grafo a -> a -> Bool
esAislado g v = grado g v == 0
```

Definición 1.5.7. *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 1.5.8. Dado un grafo G = (V, A) llamamos valencia mínima o grado mínimo de G al valor $\delta(G) = \min\{grad(v)|v \in V\}$

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]</pre>
```

Definición 1.5.9. Dado un grafo G = (V, A) llamamos **valencia máxima** o **grado máximo** de G al valor $\delta(G) = \max\{grad(v)|v \in V\}$

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]</pre>
```

Definición 1.5.10. Se dice que un grafo es simple si no contiene lazos ni aristas repetidas.

Comentario: Los grafos que estamos considerando las aristas es un conjunto, por lo que ser simples se reduce a no tener lazos.

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 ((x,x) 'aristaEn' g) | x <- vertices g]</pre>
```

Definición 1.5.11. 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: $d_1 \ge d_2 \ge \cdots \ge d_n$.

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

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

Nota 1.5.2. ¿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 1.5.12. *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 :: [Int] -> Bool
secuenciaGrafica ss = even (sum ss) && all p ss
where p s = s >= 0 && s <= length ss</pre>
```

Definición 1.5.13. 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

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

En la definición anterior se ha usado la función (subconjunto xs ys) que se verifica si xs es un subconjunto de ys. Por ejemplo,

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

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

Definición 1.5.14. 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 grah) de G.

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

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

```
esSubgrafoMax :: Ord a => Grafo a -> Grafo a -> Bool
esSubgrafoMax g' g =
esSubgrafo g' g && vertices g' == vertices g
```

Definición 1.5.15. 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 (es Subgrafo Propio 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 &&
   (vertices g /= vertices g' || aristas g /= aristas g')
```

1.5.2. Propiedades de grafos

Teorema 1.5.16 (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]))</pre>
```

Teorema 1.5.17 (Havel–Hakimi). Si n > 1 y $D = [d_1, \ldots, d_n]$ 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 (null ss)) ||
  secuenciaGrafica (map (\x -> x-1) (take s ss) ++ drop s ss)
```

1.5.3. Operaciones y propiedades sobre grafos

Eliminación de una arista

Definición 1.5.18. 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.

Eliminación un vértice

Definición 1.5.19. 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 $A' = A \setminus \{a \in A | v$ es un extremo de $a\}$. Esta operación se denomina **eliminar un vértice**.

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

Suma de aristas

Definición 1.5.20. 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 (suma Arista 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)]
```

Suma de vértices

Definición 1.5.21. 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 (suma Vertice g a) suma el vértice a al grafo g.

Propiedad de los grafos completos

Proposición 1.5.22. 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
```

Comentario: 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 1.5.23. 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,

Unión de grafos

Definición 1.5.24. 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 1.5.25. *Dado un grafo* G = (V, A) *se define el grafo complementario* de G como $\overline{G} = (V, \overline{A})$, donde $\overline{A} = \{(u, v) | u, v \in V, (u, v) \notin A\}$.

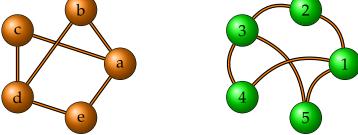
La función (complementario g) devuelve el grafo complementario de g.

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

1.6. 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 caracterísicas.

En primer lugar, introduciremos algunos conceptos relacionados con conjuntos y aplicaciones entre ellos que nos ayudarán posteriormente a definir relaciones especiales entre grafos.

1.6.1. Conceptos previos

Definición 1.6.1. El producto cartesiano ¹⁴ 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,

```
ghci> 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]</pre>
```

Definición 1.6.2. Una relación binaria 15 (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 =
r 'esSubconjunto' productoCartesiano xs ys
```

Definición 1.6.3. Si x es un elemento del conjunto A y R es una relación binaria entre A y B, la **imagen del elemento** x en la relación R es el valor o los valores correspondientes a x en B dado o dados por la relación R.

 $^{^{14} \}mathtt{https://en.wikipedia.org/wiki/Cartesian_product}$

¹⁵https://en.wikipedia.org/wiki/Binary_relation

La función (imagenRelacion r x) es la imagen de x en la relación textttr. 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 => [(a,b)] -> a -> [b]
imagenRelacion r x =
  [y | (z,y) <- r, z == x]</pre>
```

Definición 1.6.4. *Un conjunto se dice unitario si sólo tiene un elemento.*

La función (unitario xs) se verifica si el conjunto xs es unitario. Por ejemplo,

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

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

Definición 1.6.5. Dada una relación R entre A y B, su dominio es el subconjunto de A 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)
```

Definición 1.6.6. *Dada una relación R entre A y B, 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 [unitario (imagenRelacion r x) | x <- dominio r]</pre>
```

Definición 1.6.7. *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 1.6.1. 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),'c'),((1,5),'a')],[((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]
  where variacionesR _ [] = [[]]
     variacionesR 0 _ = [[]]
     variacionesR k us =
        [u:vs | u <- us, vs <- variacionesR (k-1) us]</pre>
```

Comentario: La definición de variaciones R se debe de hacer de forma global con su especificación y ejemplos. En realidad, debería de hacerse en un capítulo sobre combinatoria (que también es parte de matemática discreta)

Definición 1.6.8. Si R es una relación binaria entre A y B y x es un elemento del conjunto A, la **imagen del elemento** x en la relación R es el valor asociado a x por la relación R.

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

Definición 1.6.9. Diremos que una función f entre dos conjuntos es **inyectiva** ¹⁶ si a elementos distintos del dominio le corresponden elementos distintos de la imagen; es decir, si $\forall a, b \in dominio(f)$ tales que $a \neq b$, $f(a) \neq f(b)$.

La función (esInyectiva fs) se verifica si la función fs es inyectiva.

```
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 p (map snd f)
where p b = unitario [u | (u,v) <- f, v == b]</pre>
```

¹⁶https://en.wikipedia.org/wiki/Injective_function

Comentario: Usando las siguientes funciones auxiliares se obtiene una definición alternativa de esInyectiva.

```
-- (rango r) es 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)

-- (antiImagenRelacion r y) es la antiimagen del elemento y en la
-- relación binaria r.
-- antiImagenRelacion [(1,3),(2,3),(7,4)] 3 == [1,2]
antiImagenRelacion :: Eq b => [(a,b)] -> b -> [a]
antiImagenRelacion r y =
  [x | (x,z) <- r, z == y]

esInyectiva2 :: (Eq a, Eq b) => Funcion a b -> Bool
esInyectiva2 f =
  and [unitario (antiImagenRelacion f y) | y <- rango f]
```

Comentario: Se debería de escribir un capítulo sobre "Conjuntos, relaciones y funciones" para incluir las correspondientes funciones que estamos usando (como esSubconjunto, esRelacion, esInyectiva,...

Definición 1.6.10. *Diremos que una función f entre dos conjuntos A y B es una sobreyectiva* 17 *si todos los elementos de B son imagen de algún elmento 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,

```
ghci> esSobreyectiva [1,2,3] [4,5,6] [(1,4),(2,5),(3,6)]
True
ghci> esSobreyectiva [1,2,3] [4,5,6] [(1,4),(2,5),(3,4)]
False
ghci> 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 = all ('elem' vs) ys
where vs = map snd f
```

¹⁷https://en.wikipedia.org/wiki/Surjective_function

Comentario: Una definicón alternativa es

```
esSobreyectiva2 :: (Eq a,Eq b) => [a] -> [b] -> Funcion a b -> Bool
esSobreyectiva2 _ ys f =
  ys 'esSubconjunto' rango f
```

Definición 1.6.11. Diremos que una función f entre dos conjuntos A y B es biyectiva ¹⁸ 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,

```
ghci> esBiyectiva [1,2,3] [4,5,6] [(1,4),(2,5),(3,6),(3,6)]
True
ghci> esBiyectiva [1,2,3] [4,5,6] [(1,4),(2,5),(3,4)]
False
ghci> 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
```

Definición 1.6.12. Si f es una función sobreyectiva entre los conjuntos A y B, definimos la **función inversa** ¹⁹ 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,

Comentario: Falta ejemplo.

```
inversa :: [(a,b)] -> [(b,a)]
inversa [] = []
inversa ((a,b):fs) = (b,a):inversa fs
```

¹⁸https://en.wikipedia.org/wiki/Bijective_function

¹⁹https://en.wikipedia.org/wiki/Inverse_function

Definición 1.6.13. Si f es una función entre dos grafos G = (V, A) y G' = (V', A'), diremos que **conserva la adyacencia** si $\forall u, v \in V$ tales que $(u, v) \in A$ entonces verifica que $(f(u), f(v)) \in A$.

La función (conservaAdyacencia g1 g2 f) se verifica si la función f conserva las adyacencias. Por ejemplo,

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

1.6.2. Morfismo

Definición 1.6.14. Dados dos grafos simples G = (V, A) y G' = (V', A'), un **morfismo** entre G y G' es una función $\phi : V \to V'$ 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 :: (Eq 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.

```
morfismos :: (Eq a, Ord b) => Grafo a -> Grafo b -> [[(a,b)]]
morfismos g h =
    [xs | xs <- funciones vs1 vs2 , esMorfismo g h xs]
    where vs1 = vertices g
    vs2 = vertices h</pre>
```

Comentario: Se puede simplificar, ya que las variables locales sólo se usan una vez

```
morfismos2 :: (Eq a, Ord b) => Grafo a -> Grafo b -> [[(a,b)]]
morfismos2 g h =
  [f | f <- functiones (vertices g) (vertices h)
  , esMorfismo g h f]</pre>
```

1.6.3. Isomorfismo

Definición 1.6.15. 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 (esIsomorfismo g h f) se verifica si la aplicación f es un isomorfismo entre los grafos g y h.

Comentario: No es necesario usar las variables locales en la definición anterior. La función (isomorfismos g h) devuelve todos los isomorfismos posibles entre los grafos g y h. Por ejemplo,

```
isomorfismos :: (Ord a,Ord b) => Grafo a -> Grafo b -> [Funcion a b]
isomorfismos g h =
   [f | f <- funciones vs1 vs2 , esIsomorfismo g h f]
   where vs1 = vertices g
      vs2 = vertices h</pre>
```

Comentario: No es necesario usar las variables locales en la definición anterior.

Definición 1.6.16. Dos grafos G y H se dicen **isomorfos** si existe algún isomorfismo entre ellos.

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

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

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

1.6.4. Automorfismo

Definición 1.6.17. Dado un grafo simple G = (V, A), un **automorfismo** de G es un isomorfismo de G en sí mismo.

La función (es Automorfismo 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 = isomorfismos g g
```

1.6.5. Invariantes por isomorfismos

Definición 1.6.18. 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.

Teorema 1.6.19. Sean G = (V, A) y G' = (V', A') dos grafos y $\phi : V \to V'$ 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.

```
prop_gradoInvariante ::
    (Ord a,Ord b) => Grafo a -> Grafo b -> Property
prop_gradoInvariante g h =
    isomorfos g h ==> orden g == orden h
```

Comentario: La propiedad no está bien definida, me da error al comprobarla con QuickCheck. Continúo después.

1.7. 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 ²⁰ y también la herramienta LinuxLive USB Creator ²¹ 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.

■ LATEX como sistema de composición de textos. La distribución de LATEX, 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 LATEX 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 TEX 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.

²⁰http://www.ubuntu.com/download/desktop

²¹http://www.linuxliveusb.com/

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 estudiantes de primer curso en la página del Dpto. de Ciencias de la Computación e Inteligencia Artificial ²² y me descargué los paquetes haskell-platform y haskell-mode desde el Gestor de Paquetes Synaptic.

Comentario: 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 ²³ para ejecutarlo desde la terminal.

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

1.8. 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

²²http://www.cs.us.es/~jalonso/cursos/i1m-15/sistemas.php

²³https://www.dropbox.com/es/install?os=lnx

grafos como listas de aristas, perdiendo en eficiencia pero ganando mucho en flexibilidad de escritura.

Comentario: 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, 10	grafoAmistad, <mark>15</mark>
adyacentes, 11	grafoCiclo, <mark>14</mark>
aristaEn, <mark>12</mark>	grafoCirculante, <mark>19</mark>
aristas, <mark>12</mark>	grafoEstrella, <mark>17</mark>
automorfismos, 44	grafoMoebiusCantor, <mark>22</mark>
bipartitoCompleto, <mark>16</mark>	grafoPetersenGen, <mark>20</mark>
complementario,33	grafoPetersen, <mark>21</mark>
completo, 16	grafoRueda, <mark>18</mark>
${ t conservaAdyacencia, 40}$	grafoThomson, <mark>20</mark>
creaGrafo, <mark>10</mark>	imagenRelacion, <mark>35</mark>
dominio, 35	imagen, <mark>37</mark>
ejGrafo, <mark>11</mark>	inversa, <mark>39</mark>
eliminaArista, <mark>29</mark>	isomorfismos, 42
eliminaVertice,30	isomorfos, <mark>43</mark>
entorno, <mark>23</mark>	morfismos, 41
esAislado, <mark>24</mark>	orden, <mark>22</mark>
es Automorfismo, 43	productoCartesiano, 34
esFuncional, <mark>36</mark>	prop_HavelHakimi, <mark>29</mark>
esFuncion, <mark>36</mark>	prop_LemaApretonDeManos,28
es Inyectiva, <mark>37</mark>	<pre>prop_completos, 31</pre>
esIsomorfismo,41	secuenciaGrados, <mark>26</mark>
esLazo, <mark>23</mark>	secuenciaGrafica, <mark>26</mark>
esRegular, <mark>24</mark>	sonIncidentes, 23
esRelacion, 34	sumaArista, <mark>30</mark>
esSimple, <mark>25</mark>	sumaGrafos, <mark>32</mark>
esSobreyectiva, <mark>38, 39</mark>	sumaVertice, <mark>31</mark>
esSubconjunto, <mark>27</mark>	unitario, <mark>35</mark>
esSubgrafoMax, <mark>27</mark>	valenciaMax, <mark>25</mark>
esSubgrafoPropio, <mark>28</mark>	valenciaMin, <mark>25</mark>
esSubgrafo, <mark>27</mark>	vertices, <mark>11</mark>
funciones, 37	
generaGrafo, <mark>13</mark>	
grado, <mark>24</mark>	