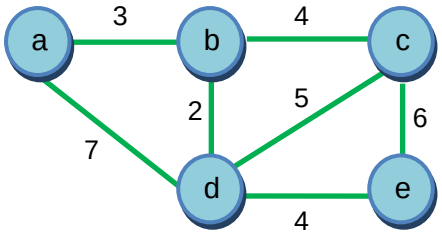
Relación de Ejercicios 6 (Grafos)

Para realizar estos ejercicios necesitarás crear diferentes ficheros tanto en Haskell como en Java. En cada caso crea un nuevo fichero (con extensión hs para Haskell y java para Java). Añade al principio de tu fichero la siguiente cabecera, reemplazando los datos necesarios:

```
--------------------------------------------------------------------------------
-- Estructuras de Datos. 2º Curso. ETSI Informática. UMA
--
-- (completa y sustituye los siguientes datos)
-- Titulación: Grado en Ingeniería ..... [Informática | del Software | de Computadores].
-- Alumno: APELLIDOS, NOMBRE
-- Fecha de entrega:  DIA | MES | AÑO
--
-- Relación de Ejercicios 6. Ejercicios resueltos: .........
--
--------------------------------------------------------------------------------
```

1. (Haskell) Consideremos el módulo `WeightedGraph` (cuyo código está disponible en CV) para implementar en Haskell los grafos no dirigidos con pesos (en las aristas). Por ejemplo, el siguiente grafo con pesos



Puede construirse en la forma:

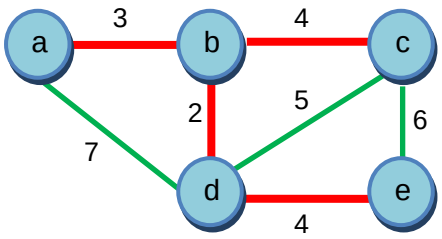
```
g1 :: WeightedGraph Char Int
g1 = mkWeigthedGraphEdges ['a','b','c','d','e']
                          [ WE 'a' 3 'b', WE 'a' 7 'd'
                          , WE 'b' 4 'c', WE 'b' 2 'd'
                          , WE 'c' 5 'd', WE 'c' 6 'e'
                          , WE 'd' 5 'e'
                          ]
```

Donde `WE x w y` representa una arista entre los vértices `x` e `y` con peso `w`. Observe que `Char` es el tipo de los vértices, e `Int` el tipo de los pesos. Note también que la función `successors` devuelve una lista de pares formado por un vértice y un peso. Por ejemplo:

```
Main> successors g1 'a'
[('b',3),('d',7)]
```

Se define la longitud (o peso, o coste) de un camino como la suma de los pesos de sus aristas. Por ejemplo, en el grafo anterior, existe un camino desde a hasta c sin pasar por b cuyo coste total es 7 (3+4).

El algoritmo de Dijkstra (1959) puede utilizarse para computar eficientemente los caminos con coste mínimos desde un vértice fijo `src` al resto de los vértices de un grafo conexo sin pesos negativos. En el grafo de la figura, las aristas de los caminos mínimos desde 'a' aparecen en rojo:



El algoritmo de Dijkstra (1959) necesita las siguientes estructuras de datos:

- Una lista v_{Opt} con los vértices del grafo hasta los que ya ha sido computado un camino mínimo desde src (el origen prefijado).
- Una lista V con los restantes vértices.
- Un diccionario cp_{Opt} que asocia cada vértice v de v_{Opt} con un par: (coste mínimo ya computado desde src hasta el vértice v, camino con tal coste).

La descripción del algoritmo es la siguiente:

1) Inicializamos V con todos los vértices del grafo, salvo el fuente src.
2) Inicializamos la lista v_{Opt} con el fuente src.
3) Inicializamos el diccionario cp_{Opt} con la asociación src -> (0,[src]); es decir, el coste óptimo desde src hasta src es 0, y el camino está formado por la lista trivial [src].
4) Mientras V no sea vacía
   a. Tomaremos todos los caminos que resultan de extender cada vértices de v_{Opt} con una arista hasta cada vértice de V.
   b. Seleccionamos el camino extendido p_{Min} que tenga coste mínimo c_{Min}. Si v_{Min} es el vértice de V donde termina este camino mínimo, esta extensión es el camino computado para este vértice.
   c. Eliminamos v_{Min} de V y añadimos v_{Min} a v_{Opt}
   d. Asociamos en el diccionario cp_{Opt} al vértice v_{Min} el par computado (c_{Min}, p_{Min})

Usando este algoritmo, escribe la función Haskell `shortestPaths` que computa la lista de caminos mínimos desde cierto vértice. Por ejemplo:

```
Main> shortestPaths g1 'a'
["a","ab","abc","abd","abde"]
```

Observa que el bucle del algoritmo puede implementarse a través de una función auxiliar con tres parámetros correspondientes a V, v_{Opt} y cp_{Opt}.

2. (Java) Implementar los grafos con pesos y el algoritmo de Dijkstra en Java.

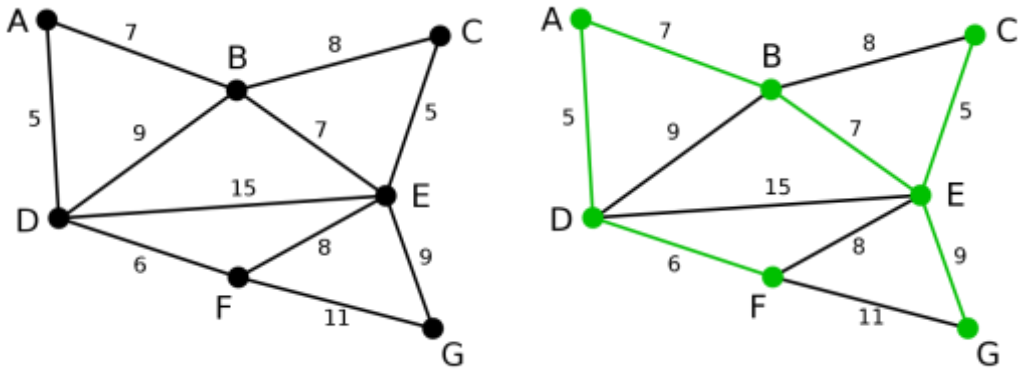
3. (Haskell) El **Algoritmo de Prim** puede ser utilizado para calcular un árbol de expansión o recubridor con coste total mínimo (minimum spanning tree) de un grafo conexo con pesos en las aristas. Al contrario que el algoritmo de Kruskal, este trabaja con una partición de todos los vértices en dos grupos:

   a. T, aquellos vértices ya incluidos en el árbol de expansión, y
   b. R, los restantes.

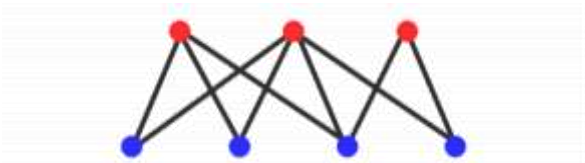
El algoritmo parte con un solo vértice inicial en T. En cada paso se selecciona una arista con peso mínimo de entre las aristas que unen un vértice de T con otro de R; esta arista será incluida en el

árbol de expansión (que podría consistir en una lista de aristas), y el vértice de esta arista que estaba en R es eliminado e incluido en T. Este proceso se repite hasta que R quede vacío.

Implementar el algoritmo de Prim en Haskell. Usa el módulo `WeightedGraph`. Como un ejemplo, el árbol de expansión mínimo para el siguiente grafo aparece a su derecha (pintados de verde las aristas del árbol de expansión mínimo).



4. Un grafo no dirigido se llama bipartito (o 2-coloreable) si sus vértices pueden colorearse usando solamente dos colores (Rojo y Azul, p.e.) de tal forma que dos vértices adyacentes tengan distinto color; estos grafos se suelen pintar colocando los rojos arriba y los azules abajo, como en la siguiente figura:



Describe e implementa un algoritmo que compruebe si un grafo es bipartito con una complejidad temporal en O(|V|+|E|).

Ayuda.- Podemos adaptar el algoritmo de búsqueda en profundidad (o en anchura) en la forma siguiente (para concretar usamos Haskell). En primer lugar consideramos un dato Color

```
data Color = Rojo |  Azul deriving (Eq,Show,Ord)
compatible Rojo = Azul
compatible Azul = Rojo
```

Guardamos en el stack los vértices pendientes de visitar junto al color que se les *debe* asignar cuando sean visitados; inicialmente colocamos en la pila el par `(src,Rojo)`, donde `src` es un vértice inicial arbitrario. Los visitados tienen también un color, y su tipo es `Set (a,Color)`, para indicar el color asignado al vértice durante la búsqueda. Una vez que hemos sacado de la pila un par `(v,color)`, comprobamos si ya ha sido visitado y coloreado con ese color (si así no fuera el grafo no es 2-coloreable). Si no ha sido visitado se incluye entre los visitados. Ahora colocamos en la pila sus sucesores no visitados con un color compatible (`compatible color`).

La función auxiliar debería programarse en la forma

```
esBipartitoDft g = aux Set.empty (S.push (head $ vertices g,Rojo) S.empty)
  where  aux visitedColor stack | S.isEmpty stack   = .. .
```

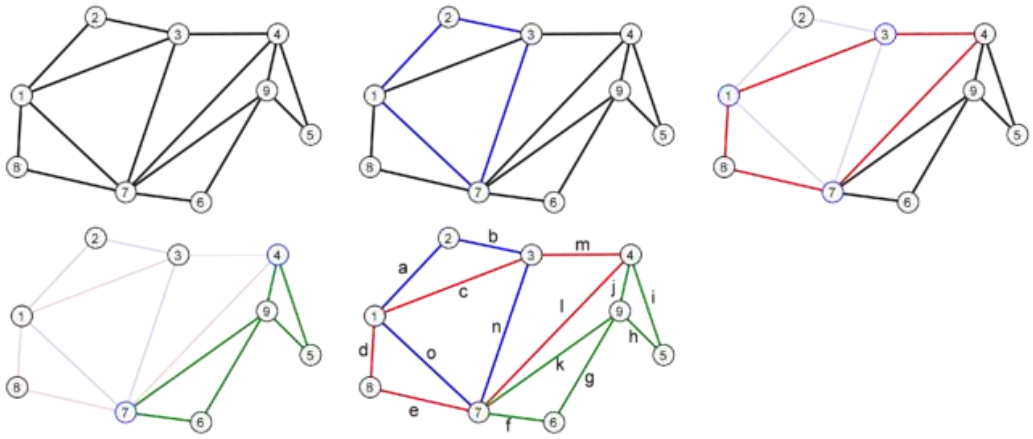
5. (Haskell/Java) Un ciclo Euleriano (*Eulerian tour*) de un grafo no dirigido es un ciclo que usa todas las aristas exactamente una vez. Se llaman eulerianos porque fue Leonard Euler quien los estudió por primera vez junto al famoso problema de los puentes de Königsberg. Carl Hierholzer demostró en 1873 que, dado un grafo conexo, éste es euleriano si y solo si todos los vértices tienen grado par.

a) Escribe un método Java o una función Haskell para localizar un ciclo euleriano en un grafo.

Para ello puede utilizarse el algoritmo de Hierholzer:

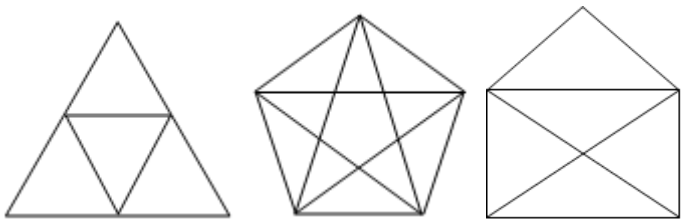
1) Toma cualquier vértice v_0 y construye un camino cerrado sin repetir ninguna arista hasta volver a v_0. Tal camino existe porque no es posible quedarse atascado en un vértice, ya que si el grado de todos los vértices es par, y llegamos a un vértice w distinto de v_0, debe ser posible abandonarlo a través de otra arista no recorrida (de lo contrario, el número de aristas incidentes con tal vértice w sería impar). El ciclo formado lo llamaremos K.
2) Si el ciclo K contiene todas las aristas, ya tenemos un ciclo euleriano. En otro caso:
   i. Eliminar del grafo las aristas de K.
   ii. Sea v_1 el primer vértice del ciclo K con grado positivo en el nuevo grafo (este debe existir, de lo contrario K sería un ciclo euleriano). Computa al igual que antes otro ciclo K_1 del nuevo grafo comenzando desde v_1.
   iii. Añade a K el nuevo ciclo K_1 insertando la secuencia de arcos de K_1 inmediatamente delante de la primera arista de K que termine en v_1.
   iv. Volver al paso 2.

Por ejemplo, la siguiente figura muestra la traza del algoritmo y los tres ciclos calculados; en azul, el primero: **a-b-n-o**; en rojo el segundo: **c-m-l-e-d**; en verde el tercero: i-h-k-f-g-j. El ciclo euleriano computado es c-m-**i-h-k-f-g-j**-l-e-d-a-b-n-o:



b) Implementa el algoritmo de Hierholzer. Observa que puedes ir eliminando las aristas del grafo durante la construcción de cada ciclo.

c) Muestra cómo es posible dibujar las siguientes figuras sin levantar el lápiz y sin *repintar* ningún trazo.



6. (Haskell/Java) Estudia el multigrafo correspondiente al problema de los puentes de Königsberg; comprueba si los programas Java o Haskell anteriores son correctos en el caso de multígrafos.

7. (Java/Haskell) Implementa el algoritmo de búsqueda en profundidad de forma que los elementos del *store* no puedan repetirse. Usa un almacén personalizado en lugar de una pila.

8. (Java/Haskell) Implementa el algoritmo de búsqueda en amplitud de forma que los elementos del *store* no puedan repetirse. Usa un almacén personalizado en lugar de una cola.

9. (Haskell) Implementa en forma recursiva la búsqueda en profundidad devolviendo una lista con los elementos visitados pero sin usar una pila.

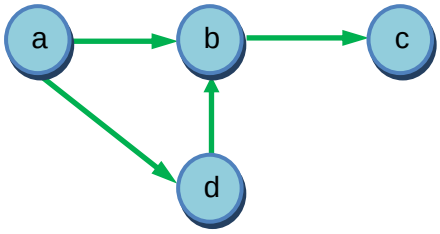
10. Desarrolla un algoritmo para comprobar la ciclicidad de un grafo no dirigido utilizando una búsqueda dft.

11. El algoritmo descrito en las transparencias (232-263) para estudiar el orden topológico y la ciclicidad de un grafo dirigido, a través de la eliminación reiterada de vértices fuentes, puede servir para encontrar un ciclo en un grafo **no dirigido** si consideramos que una fuente es un vértice con grado 0 o 1; es decir, un vértice aislado, o incidente con un solo vértice. Probar la corrección del método de las transparencias; es decir, probar que, después de eliminar en forma reiterada todos los vértices fuentes (con degree v ≤ 1) de un **grafo no dirigido** llegamos a un subgrafo sin fuentes, y

- si éste es no vacío, este subgrafo es cíclico así como el original.
- si éste es vacío, el grafo original es acíclico.

Modificar el programa Haskell (270) y el programa Java (268) de las transparencias para comprobar la ciclicidad y extraer un ciclo en caso de que exista.

12. Desarrolla e implementa un algoritmo para comprobar la ciclicidad de un **grafo dirigido** utilizando una búsqueda dft. Ten presente que los grafos como el siguiente no son cíclicos:



13. Prueba por inducción que todo grafo o digrafo con n vértices tiene a lo sumo n(n-1)/2 aristas.

14. Escribe una función/método para construir los grafos completos K_n. Escribe una función/método para construir los grafos bipartitos $K_{n,m}$; éstos grafos tienen n+m vértices, n de ellos se colorean de Rojo y los otros m en Azul, y cada vértice Azul está conectado con cada vértice Rojo.

15. Escribe un algoritmo para encontrar el camino más largo (con más vértices) en un grafo dirigido acíclico. ¿Cuál es la complejidad de este algoritmo?

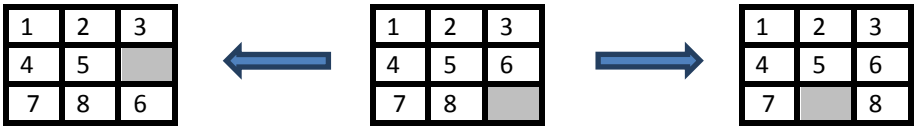
16. (Haskell) El recorrido con prioridad (priority first traversal) puede definirse si cada vértice tiene asignada una prioridad, de forma que los sucesores de un vértice son visitados en orden de

prioridad. Usa una cola con prioridad en lugar de una cola (o una pila) para implementar este recorrido. Escribe un módulo `PriorityFirstSearch` de forma que exporte una función

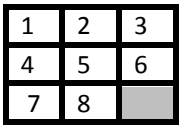
```
pftPaths :: (Ord a) => Graph a -> a -> [Path a]
```

que compute todos los caminos desde cierto vértice inicial. El orden asignado a los vértices determinará la prioridad (usualmente, el menor de entre los sucesores será visitado antes).

17. (Haskell) El 8-puzle es un rompecabezas popular que consiste en un marco con 8 fichas colocadas inicialmente y arbitrariamente en un cuadro de 3x3, numeradas del 1 al 8, por tanto quedando un hueco; ciertas fichas adyacentes al hueco pueden desplazarse al hueco horizontalmente o verticalmente. Por ejemplo, desde la configuración central podemos obtener la de la izquierda desplazando la ficha 6 hacia abajo, o podemos llegar a la de la derecha desplazando la ficha 8 a la izquierda:



El objetivo del juego es llegar a una determinada secuencia, como por ejemplo:



El módulo `EightBoard` proporciona:

a. El tipo de datos `Board` para representar la configuración de un 8-puzle.
b. La función `readBoard :: String -> Board`, que construye un tablero a partir de un string. Por ejemplo

```
goal :: Board
goal = readBoard "12345678 "
```

representa la configuración final deseada.
c. La función `oneMove :: Board -> [Board]`, que toma una configuración y devuelve la lista de posibles configuraciones obtenidas al mover una ficha.
d. La función `manhattan :: Board -> Int`, que devuelve la *distancia Manhattan* desde una configuración hasta la configuración objetivo `goal`.

La distancia Manhattan estima el número de movimientos necesarios para llegar a la configuración final; es la suma de distancias de cada ficha a su posición final, donde la distancia entre dos fichas es el menor número de movimientos para llevar una a la otra si no se interpusiera ninguna ficha.
La siguiente función puede utilizarse para representar un grafo donde los vértices son las distintas configuraciones y la función sucesor es la que define un movimiento:

```
eightPuzzleGraph :: Graph Board
eightPuzzleGraph = mkGraphAdj undefined oneMove
```

Resolver el puzle consiste en definir un camino entre los vértices o configuraciones intermedias desde una configuración inicial dada. El grafo es muy extenso y la exploración vía dft (o bft) es ineficiente. Usa nuestra implementación de la búsqueda con prioridad pft (priority first traversal) con objeto de obtener una solución. La prioridad de cada vértice será la distancia Manhattan al objetivo. Para ello, define la correspondiente instancia de la clase `Ord` entre datos de tipo `Board`. Finalmente, *estima los mejores movimientos* a partir de la siguiente configuración (o vértice) inicial:

```
initial :: Board
initial = readBoard "2 4316758"
```

18. (Java) Implementa el recorrido con prioridad (priority first traversal) en Java extendiendo la clase `Traversal` de forma apropiada.

19. (Java) En el recorrido aleatorio rft (random first traversal), los sucesores de un vértice son visitados en orden aleatorio. Este puede implementarse memorizando los vértices en cierta estructura de datos de la cual son extraídos aleatoriamente. Implementar este recorrido extendiendo de forma apropiada la clase `Traversal`.

20. (Java) Implementa la solución al 8-puzle en Java.

21. (Java) El algoritmo estudiado en las transparencias para computar el orden topológico necesita clonar el digrafo sobre el que se va a trabajar ya que modifica el digrafo considerado. Un algoritmo alternativo sin tener que realizar una clonación es el siguiente:

- Crea un diccionario inicial que asocie a cada vértice su grado de entrada en el digrafo (arcos incidentes).
- Crea una Cola de vértices vacía.
- Mientras en el diccionario queden vértices o la cola no esté vacía y el algoritmo progresó en el ciclo anterior
    - Cada vértice del diccionario que tenga asociado un valor cero se elimina del mismo y se introduce en la cola. Si hay algún vértice en esta condición el algoritmo progresa. (Esto simula que el vértice es fuente, es decir, no tiene arcos incidentes)
    - Si la cola no está vacía, se extrae el primer vértice v y se colecciona (o visita) (el algoritmo progresa). Para cada vértice sucesor de v se decrementa en una unidad el valor asociado en el diccionario. (Esto simula la eliminación del vértice del digrafo y todos los arcos que parten de él).
    - Si no se da alguna de las dos condiciones anteriores, el algoritmo no progresa.

- Si el bucle termina porque el algoritmo no ha progresado es que hay un ciclo (cuyos vértices estarán en el diccionario). En otro caso, habremos visitados todos los vértices en un orden topológico.