Tarea 4: Grafos
Puntaje máximo: 7 puntos

Entrega por Gestión: 23 de junio de 2024 hasta las 21:00 hs.

IMPORTANTE: Deberá subirse a Aulas un único archivo de Haskell (`.hs`) con los ejercicios resueltos, aquellos que no son de programar funciones se entregará como código comentado (`{- -}`). El archivo debe incluir el nombre y número de estudiante al principio del mismo.

Objetivo

El objetivo de esta tarea es entender cómo funciona el Algoritmo de Dijkstra [1] y lograr aplicarlo.

Grafos Ponderados

Los grafos ponderados son aquellos grafos que están formados por un conjunto de vértices y un conjunto de aristas donde cada arista tiene asociado un costo. La representación de estos grafos en Haskell será a través de la lista de adyacencia, con la diferencia que cada vértice adyacente de otro tiene relacionado un costo por la arista entre ellos.

Camino más corto

Un problema muy común a resolver con estos grafos es buscar el camino más "económico" entre dos vértices. Esto refiere a encontrar la forma de llegar de un vértice A a un vértice B, minimizando el costo de las aristas que se utilizan para llegar al destino. El costo de un camino entre A y B es la suma de todos los costos de las aristas elegidas para llegar al destino. Cuando hay más de un camino, el mejor es aquel que pueda minimizar esta suma. A veces, puede suceder que el camino más económico no sea el que pase por menor cantidad de vértices, ya que las transiciones entre ellos mediante sus aristas puede implicar un costo elevado.

Dijkstra

Este problema se puede resolver con el algoritmo de Dijkstra, del cual nosotros solo nos quedaremos con el resultado de "cuál es el costo del camino más económico".

Para implementar este algoritmo utilizaremos dos estructuras auxiliares que nos ayudarán a recordar en qué momento del análisis nos encontramos y cuáles son los mejores costos que venimos encontrando. Por un lado necesitamos saber si ya hemos visitado un vértice. Esto se debe a que si un vértice está visitado es porque ya hemos analizado el costo de llegar hasta él. Los vértices visitados simplemente los representaremos como una lista donde agregaremos aquellos que ya hayan sido visitados. Además, debemos llevar la cuenta para cada vértice con qué costo hemos llegado hasta allí, asumiendo que cuando no lo hemos visitados el costo de llegar es infinito (ya que queremos minimizar el costo, por defecto debe tener un costo muy elevado inicialmente). A esta información le llamaremos costos.

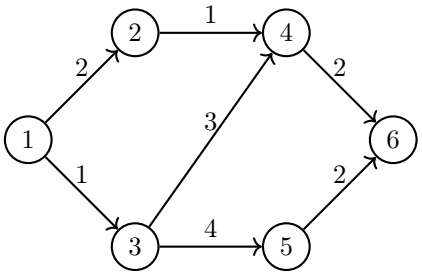
[1] `https://es.wikipedia.org/wiki/Algoritmo_de_Dijkstra`

Explicación del algoritmo de Dijkstra

Para implementar el algoritmo de Dijkstra y así encontrar el costo mínimo entre A y B debemos:

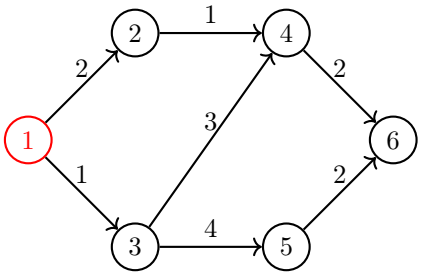
1. Comenzar marcando el vértice de origen con costo 0 en la lista *Costos* y el resto de los costos como ∞.
2. Inicializar la lista de *Visitados* vacía.
3. Iterar sobre los siguientes pasos hasta que la lista de visitados contenga todos los vértices del grafo.
   (a) Buscar en la lista de `Costos` el vértice de menor costo que no haya sido visitado.
   (b) Agregar el `Vértice` encontrado en el item anterior, a la lista de *Visitados*.
   (c) Actualizar el costo de los adyacentes de ese vértice en la lista de `Costos`, siendo el nuevo costo la suma entre el costo de la arista que une al vértice actual, con su adyacente y el costo de llegar desde el vértice origen hasta el vértice actual (el cual está guardado en la lista de *Costos*). Esto se debe hacer en caso de que dicha suma sea menor al costo ya guardado en la lista de *Costos*.
4. Retornar el costo asociado al vértice destino, que se encuentra en la lista de `Costos`.

Por ejemplo, aplicando el algoritmo de forma gráfica comenzando desde el vértice 1 hasta el 6:



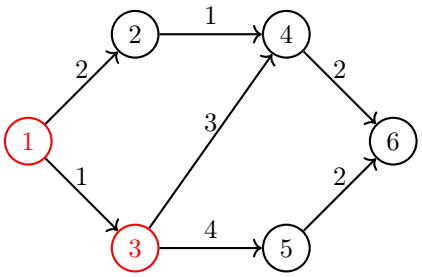
Inicialización:

Visitados						
Costos	(1,0)	(2,∞)	(3,∞)	(4,∞)	(5,∞)	(6,∞)



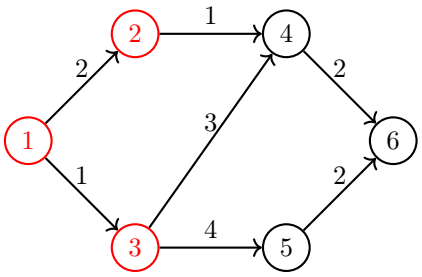
1^{ra} Iteración:

Visitados	1					
Costos	(1,0)	(2,2+0)	(3,1+0)	(4,∞)	(5,∞)	(6,∞)



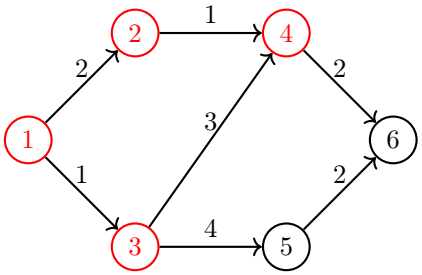
2^{da} Iteración:

Visitados	1		3			
Costos	(1,0)	(2,2)	(3,1)	(4,3+1)	(5,4+1)	(6,∞)



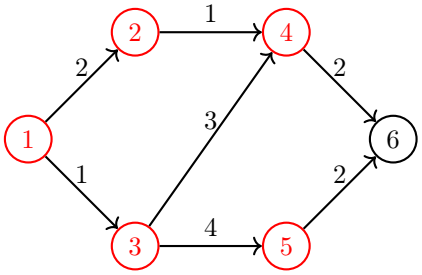
3^{ra} Iteración:

Visitados	1	2	3			
Costos	(1,0)	(2,2)	(3,1)	(4,1+2)	(5,5)	(6,∞)



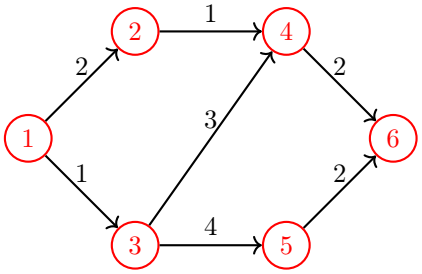
4^{a} Iteración:

Visitados	1	2	3	4		
Costos	(1,0)	(2,2)	(3,1)	(4,3)	(5,5)	(6,2+3)



5^{a} Iteración:

Visitados	1	2	3	4	5	
Costos	(1,0)	(2,2)	(3,1)	(4,3)	(5,5)	(6,5)



6^{a} Iteración:

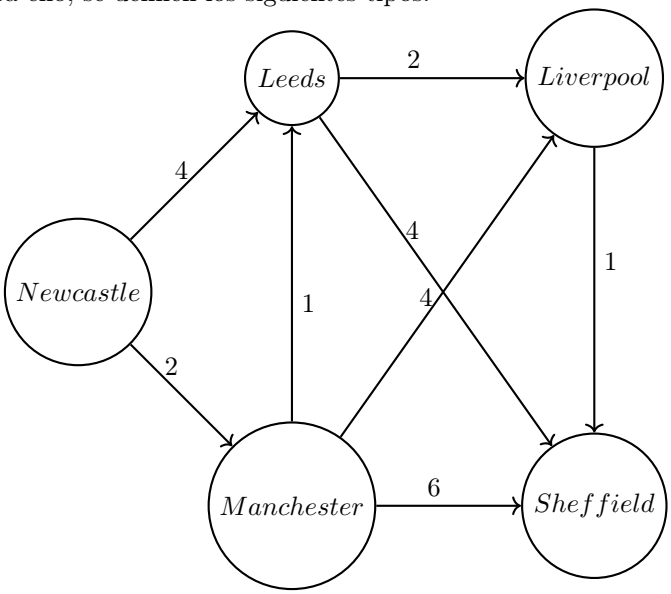
Visitados	1	2	3	4	5	6
Costos	(1,0)	(2,2)	(3,1)	(4,3)	(5,5)	(6,5)

Una vez que todos los nodos fueron visitados, podemos observar que el costo mínimo para llegar del vértice 1 al 6 es 5.

Caso de aplicación

Vuelo de menor costo

Hay n ciudades conectadas por un cierto número de vuelos. Se le da un mapa de vuelos con las ciudades y las conexiones entre las mismas. El mismo puede verse como un grafo donde los vértices son las ciudades y las aristas marcan el origen y el destino del vuelo. Cada una de las mismas posee el precio que cuesta ir del origen al destino. El objetivo es hallar el menor costo para llegar de una ciudad a la otra.
Para ello, se definen los siguientes tipos:



Para poder resolver el problema, se pide:

1. Implementar el grafo anterior en haskell como el grafo `mapadevuelos :: MapaDeVuelos`.
2. Programar la función `ciudades :: MapaDeVuelos -> [Ciudad]`, que dado un mapa de vuelos retorna la lista de ciudades del mismo.
   `Ejemplo: ciudades mapadevuelos = ["Newcastle","Leeds","Liverpool","Manchester","Sheffield"]`
3. Programar la función `initListaCostos :: [Ciudad] -> Costos`, que dada una lista de ciudades, crea una lista con costo infinito, para cada una de esas ciudades.
   `Ejemplo: initListaCostos ["Newcastle","Leeds"] = [("Newcastle",9999),("Leeds",9999)]`
4. Programar la función `costoCiudad :: Ciudad -> Costos -> Costo`, que dada una ciudad `c` y una lista de costos, retorna el costo guardado en dicha lista para `c`. En caso de que la ciudad no se encuentre en la lista de costos, retornar error.
   `Ejemplo: costoCiudad "Leeds" [("Newcastle",4),("Leeds",2)] = 2`

5. Programar la función `actualizarCosto :: Costos -> Ciudad -> Costo -> Costos`, que dada la lista costos, una ciudad `c`, y un costo `n`, busca la ciudad en la lista de costos, y en el caso de que n sea menor que el costo actual de la ciudad, actualiza el costo a n.
   Puede hacer uso de la función `(<) :: Int -> Int -> Bool`.
   ```
   Ejemplo: actualizarCosto [("Newcastle",4),("Leeds",2)] "Leeds" 3 = [("Newcastle",4),("Leeds",2)]
            actualizarCosto [("Newcastle",4),("Leeds",2)] "Newcastle" 3 = [("Newcastle",3),("Leeds",2)]
   ```

6. Programar la función `obtenerAdyacentes :: Ciudad -> MapaDeVuelos -> (Ciudad, [(Ciudad, Costo)])`, que dada una ciudad `c` y un mapa de vuelos, retorna a la ciudad `c`, junto con las ciudades adyacentes a `c` en el mapa, junto con el costo de llegar a cada una de ellas. En caso de no encontrar la ciudad en el mapa de vuelos, retornar error.
   ```
   Ejemplo:  obtenerAdyacentes "Leeds" mapadevuelos = ("Leeds", [("Liverpool",2),("Sheffield",4)])
   ```

7. `actualizarCostoAdyacentes :: (Ciudad, [(Ciudad, Costo)]) -> Costos -> Costos`, que dado un par con la ciudad actual y sus adyacentes (con sus costos asociados), y una lista de costos, para cada uno de los adyacentes de la ciudad actual, se actualiza el costo en la lista de costos, si el costo de la ciudad actual (guardado en la lista de costos) + el costo de llegar de la ciudad actual a sus adyacentes es menor que el costo de los adyacentes que ya estaba guardado en la lista de costos.
   ```
   Ejemplo:
   actualizarCostoAdyacentes ("Leeds",[("Liverpool",5),("Sheffield",6)]) [("Leeds",3),("Liverpool",10),("Sheffield",7)]
   = [("Leeds",3),("Liverpool",8),("Sheffield",7)]
   ```

8. `ciudadConMenorCosto :: Costos -> Visitadas -> Ciudad`,que dada una lista de costos y la lista de ciudades ya visitadas, retorna la ciudad de menor costo que no haya sido visitada. En caso de que todas las ciudades ya se encuentren visitadas o que no haya ciudades en la lista de costos, debe retornar error.
   Puede hace uso de la función `elem :: Eq a => a -> [a] -> Bool`, que dado un elemento y una lista de elementos verifica si el elemento pertenece a la lista. Y la función `(<=) ::  Int -> Int -> Bool`
   ```
   Ejemplo:  ciudadConMenorCosto [("Newcastle",4),("Liverpool",7),("Leeds",2)] ["Leeds"] = "Newcastle"
   ```

   Una vez realizadas estas funciones, se pide encontrar el menor costo para llegar de una ciudad A a una ciudad B:

9. `menorCosto :: MapaDeVuelos -> Ciudad -> Ciudad -> Costo`
   ```
   Ejemplo:  menorCosto mapadevuelos "Newcastle" "Sheffield" = 6
   ```

   $Newcastle \rightarrow_2 Manchester \rightarrow_1 Leeds \rightarrow_2 Liverpool \rightarrow_1 Sheffield$