INFORME PROYECTO FINAL PROGRAMACIÓN FUNCIONAL Y CONCURRENTE - PROBLEMA DE PLANIFICACIÓN DE VUELOS

Diana Marcela Oviedo
202459375
Universidad del Valle

MANEJO FUNCIONAL

I. FUNCIONES IMPLEMENTADAS

A) Encontrando Itinerarios

```Java
def itinerarios(vuelos:
List[Vuelo],
aeropuertos:List[Aeropuerto]):
(String, String) =>
List[List[Vuelo]] = {
    def aux(codigoOrigen :
String, codigoDestino: String):
List[List[Vuelo]] = {
      if (codigoOrigen ==
codigoDestino) {
        List(List())
      } else {
        val filtroVuelosCodigo
= vuelos.filter(vuelo =>
vuelo.Org == codigoOrigen)
        val
conexionVuelosCodigo =
filtroVuelosCodigo.flatMap(vuel
o => aux(vuelo.Dst,
codigoDestino).map(vuelo :: _))
        conexionVuelosCodigo
      }
    }
    (cod1:String,
cod2:String)=> aux(cod1, cod2)
  }
```

Esta función toma una lista de vuelos y una lista de aeropuertos, y retorna una función que, al recibir los códigos de dos aeropuertos, devuelve todos los itinerarios posibles entre ellos. Utiliza una función auxiliar (`aux`) que realiza una búsqueda recursiva; usando así *filter* para filtrar los vuelos que parten del aeropuerto de origen y *flatMap* para aplicar la función recursiva a los destinos de los vuelos filtrados, generando combinaciones de vuelos posibles desde el origen hasta el destino.
Esto nos permite explorar todas las rutas posibles desde el aeropuerto de origen hasta el de destino.

B) Calcular Tamaño

```Java
def calcularTamanio(lista:
List[Vuelo], acc: Int): Int = {
    if (lista.isEmpty) {
      acc
    } else {

calcularTamanio(lista.tail, acc
+ 1)
    }
  }
```

Esta función toma una lista de vuelos y un acumulador inicial, y retorna el tamaño de la lista utilizando recursión decrementando la lista y aumentando el acumulador en cada llamada recursiva hasta que la lista esté vacía,

momento en el cual retorna el valor acumulado.
Con esto, se retorna el número de items que tiene cada lista o itinerario.

C) Ordenamiento

```Java
def auxOrder(lista:
List[(List[Vuelo], Int)]):
List[(List[Vuelo], Int)] = {
    if (lista.isEmpty) {
      List()
    } else {
      val minimo =
lista.foldLeft(lista.head)((aux
, vuelo) => if (vuelo._2 <
aux._2) vuelo else aux)
      val listaFinal =
lista.filter(vuelo => vuelo !=
minimo)
      minimo ::
auxOrder(listaFinal)
    }
  }
```

Esta función toma una lista de tuplas (itinerario, valor asociado) y ordena la lista recursivamente, usando *foldLeft* para encontrar el valor mínimo en la lista y *filter* para eliminar el valor mínimo de la lista en cada paso de la recursión.
Con esto se extrae recursivamente el valor mínimo en cada iteración, construyendo una nueva lista ordenada con estos valores mínimos.

D) Itinerarios Tiempo

```Java
def itinerariosTiempo(vuelos:
List[Vuelo],
aeropuertos:List[Aeropuerto]):
(String, String) =>
List[List[Vuelo]] = {
    def auxHora(codigo:String ,
hora:Int): Int = {
      val
filtroAeropuertoZonaHoraria =
aeropuertos.filter(aeropuerto
=> aeropuerto.Cod == codigo)
      val zonaHoraria =
filtroAeropuertoZonaHoraria.hea
d.GMT
      zonaHoraria match {
        case 100 => hora + 1
        case 300 => hora + 3
        case 400 => hora + 4
        case -500 => hora
        case -400 => hora + 4
        case -600 => hora
        case -700 => hora
        case -800 => hora
        case 900 => hora + 9
        case -900 => hora
      }
    }
def aux(codigoOrigen : String,
codigoDestino: String):
List[List[Vuelo]] = {
      val listaItineraios =
itinerarios(vuelos,
aeropuertos)(codigoOrigen,
codigoDestino)
      val tiempoLlegadaTotal =
listaItineraios.map(
        itinerario => {
          def
auxCalculoTotalViaje(vueloCalcu
lar: List[Vuelo], tamanio: Int,
acc: Int): Int = {
            if (tamanio < 1) {
              acc
            }
            else {
              val
zonaHorariaOrigen =
vueloCalcular.head.HS +
(vueloCalcular.head.MS / 100)
              val
zonaHorariaDestino =
vueloCalcular.head.HL +
(vueloCalcular.head.ML / 100)
```

```
              val horaSalida =
auxHora(vueloCalcular.head.Org,
zonaHorariaOrigen)
              val horaLlegada =
auxHora(vueloCalcular.head.Dst,
zonaHorariaDestino)
              val tiempoTotal =
if (horaLlegada > horaSalida)
horaLlegada - horaSalida else
(24 + horaLlegada) - horaSalida
auxCalculoTotalViaje(vueloCalcu
lar.tail, tamanio - 1, acc +
tiempoTotal)
            }
          }
auxCalculoTotalViaje(itinerario
,
calcularTamanio(itinerario,0),
0)
        }
      )
      val vueloConTiempoTotal =
listaItineraios.zip(tiempoLlega
daTotal)
      val
vuelosOrdenadosPorTiempoTotal =
auxOrder(vueloConTiempoTotal)
vuelosOrdenadosPorTiempoTotal.m
ap(_._1).take(3)
    }
    (cod1:String,
cod2:String)=> {aux(cod1,
cod2)}
  }
```

El objetivo es generar los itinerarios que minimicen el tiempo total de viaje.
La cual se divide en 2 funciones auxiliares.
Una para ajustar las horas de salida y llegada según la zona horaria y *auxCalculoTotalViaje* para sumar los tiempos de vuelo entre las conexiones.
Con ayuda de la función *map* para transformar cada itinerario en su tiempo total de viaje., *zip* que combina los itinerarios con sus respectivos tiempos totales y el llamado a la función *auxOrder* para organizar los itinerarios por tiempo total de viaje; hacemos un llamado recursivo para así, calcular el tiempo total de viaje sumando los tiempos de vuelo entre dichas conexiones.
De esta manera, con *take* seleccionamos los 3 mejores tiempos.

E) Itinerarios Escalas

```
Java
def
itinerariosEscalas(vuelos:List[
Vuelo],
aeropuertos:List[Aeropuerto]):(
String,
String)=>List[List[Vuelo]]
  = {
    def aux(codigo1: String,
codigo2: String):
List[List[Vuelo]] = {
      val listaItineraios =
itinerarios(vuelos,
aeropuertos)(codigo1, codigo2)
      val numeroTotalEscalas =
listaItineraios.map(
        itinerario => {
          def
auxCalculoEscalas(vueloCalcular
: List[Vuelo], tamanio: Int,
acc: Int): Int = {
            if (tamanio < 1) {
              acc
            }
            else {
              val vueloEscala =
vueloCalcular.head.Esc
auxCalculoEscalas(vueloCalcular
.tail, tamanio - 1, acc +
vueloEscala)
            }
          }
auxCalculoEscalas(itinerario,
calcularTamanio(itinerario, 0),
0)
        }
      )
      val vueloConEscalasTotal
=
```

```
listaItineraios.zip(numeroTotal
Escalas)
      val
vuelosOrdenadosPorEscalasTotal
=
auxOrder(vueloConEscalasTotal)

vuelosOrdenadosPorEscalasTotal.
map(_._1).take(3)
    }
    (cod1:String,
cod2:String)=> {aux(cod1,
cod2)}
  }
```

Para retornar una función que, al recibir los códigos de dos aeropuertos, devuelve hasta tres itinerarios que minimizan el número de escalas, nos apoyamos en funciones como map, zip para transformar cada itinerario en su número total de escalas y combinar los itinerarios con sus respectivos números de escalas.
De ésta manera podemos ordenar los itinerarios por el número total de escalas y seleccionar los 3 mejores itinerarios.

F) Itinerarios Aire

```
Java
def itinerariosAire(vuelos:
List[Vuelo],
aeropuertos:List[Aeropuerto]):
(String, String) =>
List[List[Vuelo]] = {
       (cod1:String,
cod2:String)=>
{(itinerarios(vuelos,
aeropuertos)(cod1,
cod2)).take(3)}
  }
```

Recibe una lista de vuelos y aeropuertos, y retorna una función que, al recibir los códigos de dos aeropuertos, devuelve hasta tres itinerarios que minimizan el tiempo total en vuelo.
De esta manera con *take* seleccionamos los tres primeros itinerarios de la lista generada por 'itinerarios'.

G) Itinerarios Salida

```
Java
  def itinerariosSalida(vuelos:
List[Vuelo],
aeropuertos:List[Aeropuerto]):
(String, String, Int, Int) =>
List[List[Vuelo]]= {
    def aux(codigoOrigen:
String, codigoDestino: String,
horaLlegada: Int,
MinutoLlegada: Int):
List[List[Vuelo]] = {
      val listaItineraios =
itinerarios(vuelos,
aeropuertos)(codigoOrigen,
codigoDestino)
      val mejorRutaHoraria =
listaItineraios.map(
        itinerario => {
          val sumaHoraLlegada =
(1000 * itinerario.last.HL) +
itinerario.last.ML
          val
horaLlegadaEsperada  =
(1000*horaLlegada) +
MinutoLlegada
          val resultado  =
horaLlegadaEsperada
-sumaHoraLlegada
          if (resultado  >= 0)
{
            resultado
          } else {
            300
          }
        }
      )
      val listaFinal =
listaItineraios.zip(mejorRutaHo
raria)
```

```
      val listaOrdenada =
auxOrder(listaFinal)

listaOrdenada.map(_._1).take(1)
    }
    (cod1: String, cod2:
String, HC: Int, MC: Int) => {
      aux(cod1, cod2, HC, MC)
    }
  }
}
```

El objetivo es generar los itinerarios que permiten llegar a tiempo a una cita específica, donde se devuelven los 3 itinerarios que permiten llegar a tiempo, transformando cada itinerario en la diferencia de tiempo entre la hora de llegada y la hora de la cita, combinando los itinerarios con sus respectivas diferencias de tiempo.