PROYECTO FINAL

PROGRAMACIÓN FUNCIONAL Y CONCURRENTE

KEVIN ANDRES BEJARANO - 2067678

JOHAN SEBASTIAN ACOSTA -2380393

JUAN DAVID GUTIERREZ- 2060104

UNIVERSIDAD DEL VALLE SEDE TULUÁ

PROGRAMA ACADÉMICO DE INGENIERÍA DE SISTEMAS

2024

Introducción

El presente informe se enfoca en el desarrollo de un proyecto final para el curso de Fundamentos de Programación Funcional y Concurrente, cuyo objetivo principal es construir programas que ayuden a organizar la ruta de viaje más adecuada para un viajero aéreo. Este proyecto aborda uno de los problemas más comunes que enfrentan los viajeros: la planificación de vuelos que cumplan con sus necesidades específicas, tales como utilizar ciertas aerolíneas, minimizar el tiempo de viaje, reducir el número de escalas, entre otros. A continuación, se detallarán los datos utilizados, los requerimientos específicos del sistema y las implementaciones realizadas para satisfacer dichos requerimientos.

Datos

Los datos utilizados en este proyecto se estructuran en dos listas principales: una lista de aeropuertos y una lista de vuelos. Cada aeropuerto se representa mediante una tupla que incluye su código, coordenadas y franja horaria. Los vuelos, por su parte, se describen mediante tuplas que incluyen la aerolínea, número de vuelo, aeropuerto de origen y destino, horas de salida y llegada, y número de escalas.

```
case class Aeropuerto(Cod: String, X: Int, Y: Int, GMT: Double)
case class Vuelo(Aln: String, Num: Int, Org: String, HS: Int, MSInt, Dest
    ↪ : String, HL: Int, ML: Int, Esc:int)
```

Requerimientos

Los usuarios pueden expresar diferentes requerimientos al planificar sus vuelos. A continuación, se detallan las consultas más comunes y cómo se implementan para satisfacer estas necesidades:

1. Encontrando Itinerarios

La consulta más básica consiste en hallar todos los itinerarios posibles entre dos aeropuertos dados.

2. Minimizando el Tiempo de Viaje Total

Para muchos viajeros, llegar lo más rápido posible a su destino es crucial. Esta consulta busca encontrar al menos tres itinerarios que minimicen el tiempo total de viaje, considerando tanto el tiempo en vuelo como el tiempo de espera en tierra.

3. Minimizando el Número de Escalas

Dado que muchos viajeros prefieren evitar las escalas, esta consulta se centra en encontrar itinerarios que minimicen el número de escalas entre dos aeropuertos, sin tener en cuenta el tiempo total de viaje.

4. Minimizando el Tiempo en Tierra

Algunos viajeros prefieren minimizar el tiempo en vuelo, incluso si esto significa pasar más tiempo esperando en tierra. Esta consulta busca itinerarios que minimicen el tiempo de vuelo total.

5. Optimizando la Hora de Salida

Para viajeros ejecutivos que necesitan optimizar su tiempo y llegar a tiempo a sus citas, esta consulta busca determinar el itinerario que permita salir lo más tarde posible del aeropuerto de origen y llegar a tiempo al destino.

Implementación

La implementación del sistema se llevó a cabo mediante funciones en Scala que manejan las listas de vuelos y aeropuertos para generar itinerarios basados en los requerimientos especificados.

Esta es la primera parte del código la cual consiste en la implementación funcional de las funciones.

```scala
//case class Vuelo(Org: String, Dst: String)

def buscarVuelos(origen: String, destino: String, vuelosDisponibles: List[Vuelo], itinerarioActual: List[Vuelo], visitados: Set[String] = Set()): List[List[Vuelo]] = {
  if (origen == destino) {
    List(itinerarioActual)
  } else {
    vuelosDisponibles.filter(vuelo => vuelo.Org == origen && !visitados.contains(vuelo.Dst)).flatMap { vuelo =>
      val nuevosVuelosDisponibles = vuelosDisponibles.filterNot(_ == vuelo)
      buscarVuelos(vuelo.Dst, destino, nuevosVuelosDisponibles, itinerarioActual :+ vuelo, visitados + vuelo.Org)
    }
  }
}


//visitados algoritmo distra
  //Recibe una lista de vuelos y aeropuertos
  //Retorna una función que recibe los codigos de dos aeropuertos
  //Retorna todos los itinerarios posibles de cod1 a cod2
def itinerarios(vuelos: List[Vuelo], aeropuertos: List[Aeropuerto]): (String, String) => List[List[Vuelo]] = {
  def encontrarItinerarios(cod1: String, cod2: String): List[List[Vuelo]] = {
    buscarVuelos(cod1, cod2, vuelos, List())
  }
  encontrarItinerarios
}
```

```scala
//Recibe vuelos, una lista de vuelos y aeropuertos, una lista de aeropuertos y retorna una funcion que recibe dos strings y retorna una lista de itinerarios
 //Devuelve una función que recibe c1 y c2, códigos de aeropuertos
 //y devuelve una función que devuelve los tres (si los hay) itinerarios que minimizan el tiempo total de viaje
def itinerariosTiempo(vuelos: List[Vuelo], aeropuertos: List[Aeropuerto]): (String, String) => List[List[Vuelo]] = {
 def tiempoTotal(itinerario: List[Vuelo]): Int = {
   (for {
     vuelo_indice <-  0 until itinerario.length
     } yield {
       val origen = aeropuertos.find(_.Cod == itinerario(vuelo_indice).Org).get
       val destino = aeropuertos.find(_.Cod == itinerario(vuelo_indice).Dst).get
       val diferenciaHoraria = (destino.GMT - origen.GMT) / 100

       val salida = itinerario(vuelo_indice).HS * 60 + itinerario(vuelo_indice).MS
       val llegada = itinerario(vuelo_indice).HL * 60 + itinerario(vuelo_indice).ML + (diferenciaHoraria * 60).toInt

       if (vuelo_indice != 0){
         if ((itinerario(vuelo_indice-1).HL * 60 + itinerario(vuelo_indice-1).ML) > salida ){
           val tiempo_tierra =  ((itinerario(vuelo_indice-1).HL * 60 + itinerario(vuelo_indice-1).ML) + (24*60)) - salida
           if ((llegada + tiempo_tierra.abs) >= salida) (llegada - salida) + tiempo_tierra.abs  else ((llegada  + (24 * 60)) - salida) + tiempo_tierra.abs
         } else{
           val tiempo_tierra = salida - (itinerario(vuelo_indice-1).HL * 60 + itinerario(vuelo_indice-1).ML)
           if ((llegada + tiempo_tierra.abs) >= salida) (llegada - salida)  + tiempo_tierra.abs  else ((llegada  + (24 * 60)) - salida) + tiempo_tierra.abs
         }
       }else {
         if (llegada >= salida) llegada  - salida  else (llegada  + (24 * 60)) - salida
       }
     }
   ).sum
```

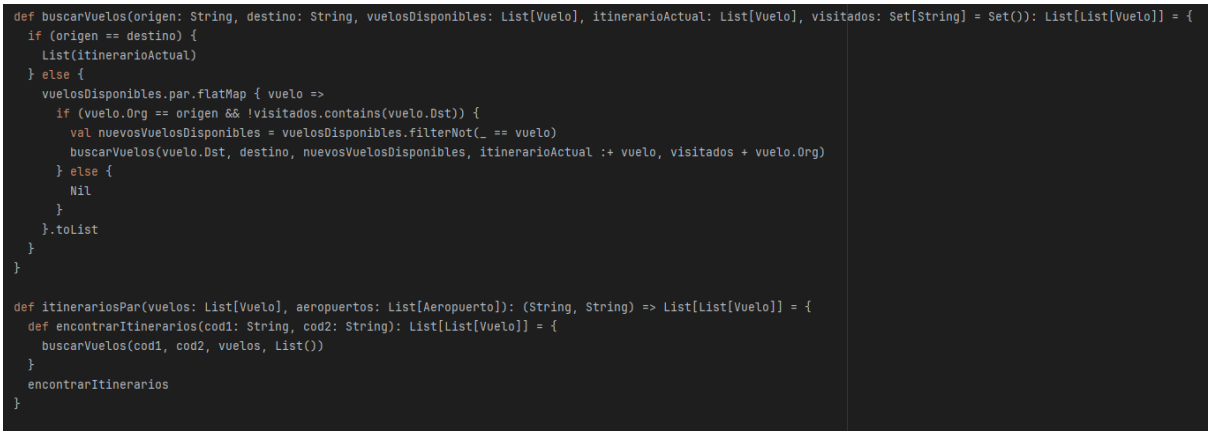


```scala
    //Recibe una lista de vuelos y aeropuertos
     //Retorna una función que recibe los codigos de dos aeropuertos
     //Retorna todos los tres mejores itinerarios posibles de cod1 a cod2
     //que minimizan el número de escalas
    //3
    // restar 1


    def itinerariosEscalas(vuelos: List[Vuelo], aeropuertos: List[Aeropuerto]): (String, String) => List[List[Vuelo]] = {
     def escalas(itinerario: List[Vuelo]): Int = {
       itinerario.map { vuelo =>
         vuelo.Esc
       }.sum + itinerario.length
     }
     def encontrarItinerarios(cod1: String, cod2: String): List[List[Vuelo]] = {
       val itinerariosPosibles = buscarVuelos(cod1, cod2, vuelos, List())
       itinerariosPosibles.sortBy(escalas).take(3)
     }
     encontrarItinerarios
    }
```

```scala
def itinerariosAire(vuelos: List[Vuelo], aeropuertos:List[Aeropuerto]): (String, String) => List[List[Vuelo]] = {
  def tiempoTotalVuelo(itinerario: List[Vuelo]): Int = {
    itinerario.map { vuelo =>
      val origen = aeropuertos.find(_.Cod == vuelo.Org).get
      val destino = aeropuertos.find(_.Cod == vuelo.Dst).get
      val diferenciaHoraria = (destino.GMT - origen.GMT) / 100

      val salida = (vuelo.HS * 60) + vuelo.MS
      val llegada = (vuelo.HL * 60) + vuelo.ML + (diferenciaHoraria * 60).toInt
      if (llegada >= salida) llegada - salida else (llegada + (24 * 60)) - salida
    }.sum
  }
  def encontrarItinerarios(cod1: String, cod2: String): List[List[Vuelo]] = {
    val itinerariosPosibles = buscarVuelos(cod1, cod2, vuelos, List())
    itinerariosPosibles.sortBy(tiempoTotalVuelo).take(3)
  }
  encontrarItinerarios
}
```

IMPLEMENTACIÓN PARALELISMO

1. Recibe una lista de vuelos y aeropuertos
   Retorna una función que recibe los códigos de dos aeropuertos
   Retorna todos los itinerarios posibles de cod1 a cod2



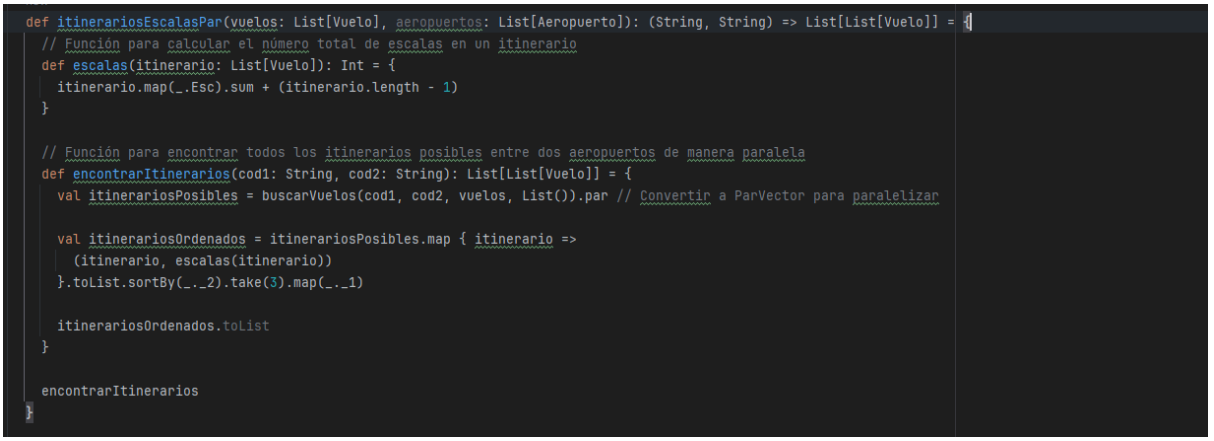
```
def buscarVuelos(origen: String, destino: String, vuelosDisponibles: List[Vuelo], itinerarioActual: List[Vuelo], visitados: Set[String] = Set()): List[List[Vuelo]] = {
  if (origen == destino) {
    List(itinerarioActual)
  } else {
    vuelosDisponibles.par.flatMap { vuelo =>
      if (vuelo.Org == origen && !visitados.contains(vuelo.Dst)) {
        val nuevosVuelosDisponibles = vuelosDisponibles.filterNot(_ == vuelo)
        buscarVuelos(vuelo.Dst, destino, nuevosVuelosDisponibles, itinerarioActual :+ vuelo, visitados + vuelo.Org)
      } else {
        Nil
      }
    }.toList
  }
}

def itinerariosPar(vuelos: List[Vuelo], aeropuertos: List[Aeropuerto]): (String, String) => List[List[Vuelo]] = {
  def encontrarItinerarios(cod1: String, cod2: String): List[List[Vuelo]] = {
    buscarVuelos(cod1, cod2, vuelos, List())
  }
  encontrarItinerarios
}
```

   - Este método recibe una lista de vuelos y una lista de aeropuertos. Retorna una función que, dada dos códigos de aeropuerto, encuentra todos los itinerarios posibles entre ellos. Utiliza el método buscarVuelos para encontrar los itinerarios.
   - Paralelismo: Utilizamos vuelosDisponibles.par para convertir la lista de vuelos disponibles en una secuencia paralela (ParSeq), lo que permite que la operación flatMap se ejecute en paralelo. Beneficio: Mejora el rendimiento al aprovechar múltiples núcleos del procesador para procesar varios vuelos simultáneamente, lo que es especialmente útil con grandes listas de vuelos y combinaciones de itinerarios complejas.
2. Recibe vuelos, una lista de vuelos y aeropuertos, una lista de aeropuertos y retorna una funcion que recibe dos strings y retorna una lista de itinerarios
   Devuelve una función que recibe c1 y c2, códigos de aeropuertos y devuelve una función que devuelve los tres (si los hay) itinerarios que minimizan el tiempo total de viaje.

```scala
def itinerariosTiempoPar(vuelos: List[Vuelo], aeropuertos: List[Aeropuerto]): (String, String) => List[List[Vuelo]] = {
  def tiempoTotal(itinerario: List[Vuelo]): Int = {
    (for {
      vuelo_indice <- 0 to until < itinerario.length
    } yield {
      val origen = aeropuertos.find(_.Cod == itinerario(vuelo_indice).Org).get
      val destino = aeropuertos.find(_.Cod == itinerario(vuelo_indice).Dst).get
      val diferenciaHoraria = (destino.GMT - origen.GMT) / 100

      val salida = itinerario(vuelo_indice).HS * 60 + itinerario(vuelo_indice).MS
      val llegada = itinerario(vuelo_indice).HL * 60 + itinerario(vuelo_indice).ML + (diferenciaHoraria * 60).toInt

      if (vuelo_indice != 0) {
        if ((itinerario(vuelo_indice-1).HL * 60 + itinerario(vuelo_indice-1).ML) > salida) {
          val tiempo_tierra = ((itinerario(vuelo_indice-1).HL * 60 + itinerario(vuelo_indice-1).ML) + (24*60)) - salida
          if ((llegada + tiempo_tierra.abs) >= salida) (llegada - salida) + tiempo_tierra.abs else ((llegada  + (24 * 60)) - salida) + tiempo_tierra.abs
        } else {
          val tiempo_tierra = salida - (itinerario(vuelo_indice-1).HL * 60 + itinerario(vuelo_indice-1).ML)
          if ((llegada + tiempo_tierra.abs) >= salida) (llegada - salida) + tiempo_tierra.abs else ((llegada  + (24 * 60)) - salida) + tiempo_tierra.abs
        }
      } else {
        if (llegada >= salida) llegada - salida else (llegada  + (24 * 60)) - salida
      }
    }).sum
  }

  def encontrarItinerarios(cod1: String, cod2: String): List[List[Vuelo]] = {
    val itinerariosPosibles = buscarVuelos(cod1, cod2, vuelos, List()).par // Convertir a ParVector para paralelizar

    val itinerariosConTiempo = itinerariosPosibles.map { itinerario =>
      (itinerario, tiempoTotal(itinerario))
    }

    val itinerariosOrdenados = itinerariosConTiempo.seq.sortBy(_._2).take(3).map(_._1)
    itinerariosOrdenados.toList
  }

  encontrarItinerarios
}
```

- Función tiempoTotal: Calcula el tiempo total de un itinerario dado (lista de vuelos). Para cada vuelo en el itinerario, se calcula la hora de llegada y salida teniendo en cuenta la diferencia horaria entre el origen y el destino. Si no es el primer vuelo, se calcula el tiempo en tierra (la diferencia entre la llegada del vuelo anterior y la salida del vuelo actual) y se asegura que no haya tiempos negativos (considera la vuelta al día siguiente). La suma de estos tiempos se retorna como el tiempo total del itinerario.
- Función encontrarItinerarios: Encuentra itinerarios posibles entre dos aeropuertos dados (cod1 y cod2). Usa una función buscarVuelos (que no está definida en la imagen) para encontrar todos los itinerarios posibles. Calcula el tiempo total para cada itinerario usando la función tiempoTotal. Ordena los itinerarios por tiempo total y selecciona los 3 mejores. Retorna la lista de los 3 itinerarios más rápidos.

3. Recibe una lista de vuelos y aeropuertos Retorna una función que recibe los codigos de dos aeropuertos Retorna todos los tres mejores itinerarios posibles de cod1 a cod2 que minimizan el número de escalas.



```
def itinerariosEscalasPar(vuelos: List[Vuelo], aeropuertos: List[Aeropuerto]): (String, String) => List[List[Vuelo]] = {
  // Función para calcular el número total de escalas en un itinerario
  def escalas(itinerario: List[Vuelo]): Int = {
    itinerario.map(_.Esc).sum + (itinerario.length - 1)
  }

  // Función para encontrar todos los itinerarios posibles entre dos aeropuertos de manera paralela
  def encontrarItinerarios(cod1: String, cod2: String): List[List[Vuelo]] = {
    val itinerariosPosibles = buscarVuelos(cod1, cod2, vuelos, List()).par // Convertir a ParVector para paralelizar

    val itinerariosOrdenados = itinerariosPosibles.map { itinerario =>
      (itinerario, escalas(itinerario))
    }.toList.sortBy(_._2).take(3).map(_._1)

    itinerariosOrdenados.toList
  }

  encontrarItinerarios
}
```

- Función escalas: Calcula el número total de escalas en un itinerario. Toma un itinerario (lista de vuelos) como entrada y devuelve la suma de las escalas en cada vuelo menos uno (para ajustar el conteo de escalas totales).
- Función encontrarItinerarios: Encuentra itinerarios posibles entre dos aeropuertos dados (cod1 y cod2). Utiliza una función buscarVuelos (que no está definida en la imagen) para encontrar todos los itinerarios posibles. Convierte el resultado en un ParVector para paralelizar la operación, lo que puede mejorar el rendimiento en grandes conjuntos de datos. Mapea cada itinerario a un par (itinerario, número de escalas) utilizando la función escalas. Ordena los itinerarios por el número de escalas y selecciona los 3 itinerarios con el menor número de escalas. Retorna una lista de estos 3 itinerarios.

4. Recibe una lista de vuelos y aeropuertos Retorna una función que recibe los codigos de dos aeropuertos Retorna todos los tres mejores itinerarios posibles de cod1 a cod2 que minimizan el tiempo en itinerarios.

```scala
def itinerariosAirePar(vuelos: List[Vuelo], aeropuertos: List[Aeropuerto]): (String, String) => List[List[Vuelo]] = {
  def tiempoTotalVuelo(itinerario: List[Vuelo]): Int = {
    itinerario.map { vuelo =>
      val origen = aeropuertos.find(_.Cod == vuelo.Org).get
      val destino = aeropuertos.find(_.Cod == vuelo.Dst).get
      val diferenciaHoraria = (destino.GMT - origen.GMT) / 100

      val salida = (vuelo.HS * 60) + vuelo.MS
      val llegada = (vuelo.HL * 60) + vuelo.ML + (diferenciaHoraria * 60).toInt
      if (llegada >= salida) llegada - salida else (llegada + (24 * 60)) - salida
    }.sum
  }

  def encontrarItinerarios(cod1: String, cod2: String): List[List[Vuelo]] = {
    val itinerariosPosibles = buscarVuelos(cod1, cod2, vuelos, List()).par // Convertir a ParVector para paralelizar

    val itinerariosOrdenados = itinerariosPosibles.map { itinerario =>
      (itinerario, tiempoTotalVuelo(itinerario))
    }.toList.sortBy(_._2).take(3).map(_._1)

    itinerariosOrdenados.toList
  }

  encontrarItinerarios
}
```

- Función tiempoTotalVuelo: Calcula el tiempo total de vuelo (tiempo en el aire) de un itinerario dado (lista de vuelos). Para cada vuelo en el itinerario, obtiene el aeropuerto de origen y destino usando sus códigos. Calcula la diferencia horaria entre el destino y el origen. Calcula el tiempo de salida y llegada del vuelo en minutos. Ajusta la diferencia horaria al tiempo de llegada. Si la llegada es después de la salida, se calcula la diferencia directa; de lo contrario, se ajusta considerando la vuelta al día siguiente. Suma los tiempos de todos los vuelos para obtener el tiempo total de vuelo del itinerario.
- Función encontrarItinerarios: Encuentra itinerarios posibles entre dos aeropuertos dados (cod1 y cod2). Utiliza una función buscarVuelos (que no está definida en la imagen) para encontrar todos los itinerarios posibles. Convierte el resultado en un ParVector para paralelizar la operación, mejorando el rendimiento en grandes conjuntos de datos. Mapea cada itinerario a un par (itinerario, tiempo total de vuelo) utilizando la función tiempoTotalVuelo. Ordena los itinerarios por tiempo total de vuelo y selecciona los 3 itinerarios más rápidos. Retorna una lista de estos 3 itinerarios.

5. Recibe una lista de vuelos y aeropuertos //Retorna una función que recibe los codigos de dos aeropuertos y dos enteros, que es la hora de la cita Retorna todos los tres mejores itinerarios posibles de cod1 a cod2 que permiten llegar a una hora de la cita.

```
def itinerariosSalidaPar(vuelos: List[Vuelo], aeropuertos: List[Aeropuerto]): (String, String, Int, Int) => List[Vuelo] = {
  def atiempo(itinerario: List[Vuelo], hora: Int, min: Int): Boolean = {
    val llegada = (itinerario.last.HL * 60) + itinerario.last.ML
    val cita = (hora * 60) + min
    llegada <= cita
  }

  def el_mas_tarde(itinerario: List[Vuelo]): Int = {
    (itinerario.head.HS * 60) + itinerario.head.MS
  }

  def encontrarItinerarios(cod1: String, cod2: String, hora: Int, min: Int): List[Vuelo] = {
    val itinerariosPosibles = buscarVuelos(cod1, cod2, vuelos, List()).par // Convertir a ParVector para paralelizar

    val itinerariosFiltrados = itinerariosPosibles.filter(atiempo(_, hora, min)).toList
    val itinerariosOrdenados = itinerariosFiltrados.sortBy(el_mas_tarde).take(1)

    if (itinerariosOrdenados.nonEmpty) itinerariosOrdenados.head else List()
  }

  encontrarItinerarios
}
```

- Función aTiempo: Entrada: Recibe un itinerario (lista de vuelos), una hora y un minuto. Salida: Retorna un booleano que indica si la llegada del último vuelo en el itinerario es antes o igual a la cita especificada por hora y min. Descripción: Calcula la llegada del último vuelo en minutos (multiplicando la hora por 60 y sumando los minutos del vuelo) y la compara con la cita.
- Función el_mas_tarde: Entrada: Recibe un itinerario (lista de vuelos). Salida: Retorna un entero que representa la hora y minuto de salida del primer vuelo en el itinerario en minutos. Descripción: Calcula el tiempo en minutos desde la medianoche para el primer vuelo del itinerario.
- Función encontrarItinerarios: Entrada: Recibe dos códigos de ciudad (strings), una hora y un minuto (enteros). Salida: Retorna una lista de vuelos que corresponden al mejor itinerario posible. Descripción: Utiliza la función buscarVuelos para encontrar itinerarios posibles entre las dos ciudades, paralelizando la operación (conversión a ParVector). Filtra estos itinerarios usando la función aTiempo para asegurarse de que lleguen antes o a la hora especificada. Ordena los itinerarios filtrados por la función el_mas_tarde para encontrar el más temprano. Retorna el mejor itinerario o una lista vacía si no hay itinerarios válidos.

Benchmarking, demostración de ganancia

Capturas de pantalla pruebas:

```
println(compararfunciones(itinerario,itinerariopar)("PHX", "DTW"))
println(compararfunciones(tiempo,tiempopar )("PHX", "DTW"))
println(compararfunciones(escalas,escalaspar )("PHX", "DTW"))
println(compararfunciones(aire,airepar )("PHX", "DTW"))
println(compararfuncionessalidas(salidas, salidasPar )("PHX", "DTW",16, 5))
}
```

```
> Task :app:run
Proyecto final
Unable to create a system terminal
(82.9107,51.7125,1.6033009427121103)
(151.148,80.5029,1.8775472684834957)
(69.8639,55.5477,1.2577280427452442)
(153.8742,56.9604,2.7014241473023364)
(0.0263,0.1782,0.14758698092031425)
```

A la izquierda podemos ver los tiempos de ejecución de las funciones secuenciales, y a su derecha, separados por una coma, están los tiempos de ejecución de las funciones paralelizadas. Después de eso, se muestra la aceleración obtenida, que en todos los casos es mayor que 1. Esto indica que las funciones paralelizadas son más rápidas que las secuenciales, demostrando una ganancia en tiempo al utilizar la paralelización.

- En la primera función “itinerario”

Se observa que la función paralelizada es aproximadamente 1.6 veces más rápida que la secuencial.

- En la segunda función “itinerariosTiempo

La función paralelizada es aproximadamente 1.87 veces más rápida que la secuencial.

- En la tercera función “itinerariosEscalas

La función paralelizada es aproximadamente 1.25 veces más rápida que la secuencial.

- En la cuarta función “itinerariosAire“

La función paralelizada es aproximadamente 2.7 veces más rápida que la secuencial.

- En la quinta función “itinerariosSalida“

La función paralelizada es más lenta, con una aceleración de 0.15, teniendo en cuenta que se probó en un valor pequeño.

En conclusión, podemos ver en las comparaciones que entre más grande después de cierto tamaño de vectores la solución paralela empieza a ser mucho mejor, siempre van a haber muchos procesos que no se pueden paralelizar, pero en este caso para esta solución nos podemos aprovechar siempre de la paralelización.