Proyecto: El problema de la Planificación de vuelos

Juan José Bolaños Delgado 1942124 - 3743

Programación funcional y concurrente

Junio 2024

Informe del taller

Informe - Estructuras de datos

Describe claramente las estructura de datos utilizadas , tanto para las soluciones secuenciales como para las paralelas. Específicamente describe las colecciones paralelas utilizadas.

Función	IF ELSE	Conjuntos (Set)	Reco. Patrones	Alto Orden	For	Secuencias (Seq)	Recursión	Task	Parallel	ParVector
itinerarios	**X**	**X**		**X**			**X**		**X**	
itinerariosTiempo	**X**	**X**		**X**	**X**		**X**		**X**	
itinerariosEscalas		**X**		**X**			**X**		**X**	
itinerariosAire		**X**	**X**	**X**			**X**		**X**	
itinerariosSalida		**X**	**X**	**X**			**X**		**X**	
itinerariosPar		**X**		**X**		**X**	**X**		**X**	
itinerariosTiempoPar		**X**		**X**			**X**		**X**	
itinerariosEscalasPar		**X**		**X**			**X**		**X**	
itinerariosAirePar		**X**		**X**			**X**		**X**	
itinerariosSalidaPar		**X**		**X**			**X**		**X**	

Informe - Programación funcional

Todas las funciones desarrolladas corresponden al estilo de programación funcional. El informe evidencia el manejo de la recursión, del reconocimiento de patrones, de mecanismos de encapsulación, de funciones de alto orden e iteradores, dentro del código. Se describen claramente (por ejemplo usando tablas) qué funciones usan cuáles de estas técnicas.

.

itinerarios

```Java
  def itinerarios(vuelos: List[Vuelo], aeropuertos: List[Aeropuerto]):
(String, String) => List[Itinerario] = {

      def buscar(origen: String, destino: String, visitados: Set[String] =
Set()): List[Itinerario] = {
      if (origen == destino) {
      // si el origen es igual al destino, retornamos una lista con una
lista vacía (indicando que llegamos al destino)
      List(List())
      } else {
      // filtramos los vuelos que salen del origen y cuyos destinos no han
sido visitados aún
      vuelos
      .filter(vuelo => vuelo.org == origen &&
!visitados.contains(vuelo.dst))
      .flatMap { vuelo =>
            /* llamada recursiva para buscar itinerarios desde el destino
del vuelo actual al destino final
            agregamos el vuelo actual al conjunto de visitados para evitar
ciclos
            */
            buscar(vuelo.dst, destino, visitados + origen)
            // agregamos el vuelo actual al inicio de cada itinerario
encontrado
            .map(itinerario => vuelo :: itinerario)
      }
      }
      }

      // Retornamos una función que toma el origen y destino como parámetros
y llama a la función de búsqueda
      (origen: String, destino: String) => buscar(origen, destino)
  }
```

itinerariosTiempo

```Java
def itinerariosTiempo(vuelos: List[Vuelo], aeropuertos: List[Aeropuerto]):
(String, String) => List[Itinerario] = {

      // Función recursiva interna que realiza la búsqueda de itinerarios
```

```
      def buscar(origen: String, destino: String, visitados: Set[String] =
Set()): List[Itinerario] = {
      if (origen == destino) List(List())
      else {
      vuelos
      .filter(vuelo => vuelo.org == origen &&
!visitados.contains(vuelo.dst))
      .flatMap(vuelo => buscar(vuelo.dst, destino, visitados +
origen).map(vuelo :: _))
      }
      }

      // Función para calcular el tiempo total de un itinerario
      def tiempoTotal(itinerario: List[Vuelo]): Int = {
      val tiempoEnAire = itinerario.map { vuelo =>
      val hlModificado = if (vuelo.hl < vuelo.hs) vuelo.hl + 24 else
vuelo.hl
      (hlModificado - vuelo.hs) * 60 + vuelo.ml - vuelo.ms
      }.sum

      val tiemposEspera = for ((vuelo1, vuelo2) <-
itinerario.zip(itinerario.tail)) yield {
      val hsModificado = if (vuelo2.hs < vuelo1.hl) vuelo2.hs + 24 else
vuelo2.hs
      (hsModificado - vuelo1.hl) * 60 + vuelo2.ms - vuelo1.ml
      }


      tiempoEnAire + tiemposEspera
      }

      // Retornamos una función que toma el origen y destino como parámetros
y llama a la función de búsqueda
      (origen: String, destino: String) =>
      buscar(origen, destino)
      .map(itinerario => (itinerario, tiempoTotal(itinerario)))
      .sortBy(_._2)
      .map(_._1)
      .take(3)
  }
```

itinerariosEscalas

Java

```
 def itinerariosEscalas(vuelos: List[Vuelo], aeropuertos: List[Aeropuerto]):
(String, String) => List[Itinerario] = {

      // Función recursiva interna que realiza la búsqueda de itinerarios
      def buscar(origen: String, destino: String, visitados: Set[String] =
Set()): List[Itinerario] = {
      if (origen == destino) List(List())
      else {
      vuelos
      .filter(vuelo => vuelo.org == origen &&
!visitados.contains(vuelo.dst))
      .flatMap(vuelo => buscar(vuelo.dst, destino, visitados +
origen).map(vuelo :: _))
      }
      }

      // Función para calcular el número de escalas de un itinerario
      def numeroEscalas(itinerario: List[Vuelo]): Int = itinerario.length

      // Retornamos una función que toma el origen y destino como parámetros
y llama a la función de búsqueda
      (origen: String, destino: String) =>
      buscar(origen, destino)
      .map(itinerario => (itinerario, numeroEscalas(itinerario)))
      .sortBy(_._2)
      .map(_._1)
      .take(3)
  }
```

itinerariosAire

Java

```
def itinerariosAire(vuelos: List[Vuelo], aeropuertos: List[Aeropuerto]):
(String, String) => List[Itinerario] = {

      // Función recursiva interna que realiza la búsqueda de itinerarios
      def buscar(origen: String, destino: String, visitados: Set[String] =
Set()): List[Itinerario] = {
      if (origen == destino) List(List())
      else {
      vuelos
      .filter(vuelo => vuelo.org == origen &&
!visitados.contains(vuelo.dst))
```

```
      .flatMap(vuelo => buscar(vuelo.dst, destino, visitados +
origen).map(vuelo :: _))
      }
      }

      // Función para calcular el tiempo total en aire de un itinerario
      def tiempoTotal(itinerario: List[Vuelo]): Int = {
      itinerario.map { vuelo =>
      val hlModificado = if (vuelo.hl < vuelo.hs) vuelo.hl + 24 else
vuelo.hl
      (hlModificado - vuelo.hs) * 60 + vuelo.ml - vuelo.ms
      }.sum
      }

      // Retornamos una función que toma el origen y destino como parámetros
y llama a la función de búsqueda
      (origen: String, destino: String) =>
      buscar(origen, destino)
      .map(itinerario => (itinerario, tiempoTotal(itinerario)))
      .sortBy(_._2)
      .map(_._1)
      .take(3)
  }
```

itinerariosSalida

Java

```
def itinerarioSalida(vuelos: List[Vuelo], aeropuertos: List[Aeropuerto]):
(String, String, Int, Int) => Itinerario = {

      // Función para convertir horas y minutos a minutos totales en el día
      def tiempoEnMinutos(hora: Int, minuto: Int): Int = hora * 60 + minuto

      // Función recursiva interna que realiza la búsqueda de itinerarios
      def buscar(origen: String, destino: String, visitados: Set[String] =
Set()): List[Itinerario] = {
      if (origen == destino) List(List())
      else {
      vuelos
      .filter(vuelo => vuelo.org == origen &&
!visitados.contains(vuelo.dst))
      .flatMap(vuelo => buscar(vuelo.dst, destino, visitados +
origen).map(vuelo :: _))
      }
```

```
      }

      // Función para optimizar la salida del itinerario
      def optimizarSalida(origen: String, destino: String, hora: Int,
minuto: Int): Itinerario = {
      val itinerariosValidos = buscar(origen, destino).filter { itinerario
=>
      tiempoEnMinutos(itinerario.last.hl, itinerario.last.ml) <=
tiempoEnMinutos(hora, minuto)
      }
      if (itinerariosValidos.isEmpty) List()
      else itinerariosValidos.maxBy(itinerario => (itinerario.head.hs,
itinerario.head.ms))
      }

      // Retornamos una función que toma origen, destino, hora y minuto, y
retorna el itinerario optimizado
      (origen: String, destino: String, hora: Int, minuto: Int) =>
optimizarSalida(origen, destino, hora, minuto)
  }
```

itinerariosPar

Java

```
  def itinerariosPar(vuelos: List[Vuelo], aeropuertos: List[Aeropuerto]):
(String, String) => List[Itinerario] = {

      def buscar(origen: String, destino: String, visitados: Set[String] =
Set()): List[Itinerario] = {
      if (origen == destino) {
      // si el origen es igual al destino, retornamos una lista con una
lista vacía (indicando que llegamos al destino)
      List(List())
      } else {
      // filtramos los vuelos que salen del origen y cuyos destinos no han
sido visitados aún
      vuelos
      .filter(vuelo => vuelo.org == origen &&
!visitados.contains(vuelo.dst))
      .par.flatMap { vuelo =>
            /* llamada recursiva para buscar itinerarios desde el destino
del vuelo actual al destino final
            agregamos el vuelo actual al conjunto de visitados para evitar
ciclos
```

```
            */
            buscar(vuelo.dst, destino, visitados + origen)
            // agregamos el vuelo actual al inicio de cada itinerario
encontrado
            .map(itinerario => vuelo :: itinerario)
      }.toList
      }
      }

      // Retornamos una función que toma el origen y destino como parámetros
y llama a la función de búsqueda
      (origen: String, destino: String) => buscar(origen, destino)
  }
```

itinerariosTiempoPar

Java

```
def itinerariosTiempoPar(vuelos: List[Vuelo], aeropuertos: List[Aeropuerto]):
(String, String) => List[Itinerario] = {

      // Función recursiva interna que realiza la búsqueda de itinerarios
      def buscar(origen: String, destino: String, visitados: Set[String] =
Set()): List[Itinerario] = {
      if (origen == destino) List(List())
      else {
      vuelos
      .filter(vuelo => vuelo.org == origen &&
!visitados.contains(vuelo.dst))
      .flatMap(vuelo => buscar(vuelo.dst, destino, visitados +
origen).map(vuelo :: _))
      }
      }

      // Función para calcular el tiempo total de un itinerario
      def tiempoTotal(itinerario: List[Vuelo]): Int = {
      val tiempoEnAire = itinerario.par.map { vuelo =>
      val hlModificado = if (vuelo.hl < vuelo.hs) vuelo.hl + 24 else
vuelo.hl
      (hlModificado - vuelo.hs) * 60 + vuelo.ml - vuelo.ms
      }.sum

      val tiemposEspera = (itinerario.par.zip(itinerario.tail.par)).map {
case (vuelo1, vuelo2) =>
```

```
      val hsModificado = if (vuelo2.hs < vuelo1.hl) vuelo2.hs + 24 else
vuelo2.hs
      (hsModificado - vuelo1.hl) * 60 + vuelo2.ms - vuelo1.ml
      }.sum

      tiempoEnAire + tiemposEspera
      }

      (origen: String, destino: String) =>
      buscar(origen, destino)
      .par.map(itinerario => (itinerario, tiempoTotal(itinerario))).toList
      .sortBy(_._2)
      .map(_._1)
      .take(3)
  }
```

itinerariosEscalasPar

Java

```
def itinerariosEscalasPar(vuelos: List[Vuelo], aeropuertos:
List[Aeropuerto]): (String, String) => List[Itinerario] = {

      // Función recursiva interna que realiza la búsqueda de itinerarios
      def buscar(origen: String, destino: String, visitados: Set[String] =
Set()): List[Itinerario] = {
      if (origen == destino) List(List())
      else {
      vuelos
      .filter(vuelo => vuelo.org == origen &&
!visitados.contains(vuelo.dst))
      .flatMap(vuelo => buscar(vuelo.dst, destino, visitados +
origen).map(vuelo :: _))
      }
      }

      // Función para calcular el número de escalas de un itinerario
      def numeroEscalas(itinerario: List[Vuelo]): Int = itinerario.length

      // Retornamos una función que toma el origen y destino como parámetros
y llama a la función de búsqueda
      (origen: String, destino: String) =>
      buscar(origen, destino)
      .par.map(itinerario => (itinerario, numeroEscalas(itinerario))).toList
      .sortBy(_._2)
```

```
      .map(_._1)
      .take(3)
  }
```

itinerariosAirePar

Java

```
  def itinerariosAirePar(vuelos: List[Vuelo], aeropuertos: List[Aeropuerto]):
(String, String) => List[Itinerario] = {

      // Función recursiva interna que realiza la búsqueda de itinerarios
      def buscar(origen: String, destino: String, visitados: Set[String] =
Set()): List[Itinerario] = {
      if (origen == destino) List(List())
      else {
      vuelos
      .filter(vuelo => vuelo.org == origen &&
!visitados.contains(vuelo.dst))
      .flatMap(vuelo => buscar(vuelo.dst, destino, visitados +
origen).map(vuelo :: _))
      }
      }

      // Función para calcular el tiempo total en aire de un itinerario
      def tiempoTotal(itinerario: List[Vuelo]): Int = {
      itinerario.map { vuelo =>
      val hlModificado = if (vuelo.hl < vuelo.hs) vuelo.hl + 24 else
vuelo.hl
      (hlModificado - vuelo.hs) * 60 + vuelo.ml - vuelo.ms
      }.sum
      }

      // Retornamos una función que toma el origen y destino como parámetros
y llama a la función de búsqueda
      (origen: String, destino: String) =>
      buscar(origen, destino)
      .par.map(itinerario => (itinerario, tiempoTotal(itinerario))).toList
      .sortBy(_._2)
      .map(_._1)
      .take(3)
  }
```

itinerariosSalidaPar

Java

```
  def itinerarioSalidaPar(vuelos: List[Vuelo], aeropuertos:
List[Aeropuerto]): (String, String, Int, Int) => Itinerario = {

      // Función para convertir horas y minutos a minutos totales en el día
      def tiempoEnMinutos(hora: Int, minuto: Int): Int = hora * 60 + minuto

      // Función recursiva interna que realiza la búsqueda de itinerarios
      def buscar(origen: String, destino: String, visitados: Set[String] =
Set()): List[Itinerario] = {
      if (origen == destino) List(List())
      else {
      vuelos
      .filter(vuelo => vuelo.org == origen &&
!visitados.contains(vuelo.dst))
      .flatMap(vuelo => buscar(vuelo.dst, destino, visitados +
origen).map(vuelo :: _))
      }
      }

      // Función para optimizar la salida del itinerario
      def optimizarSalida(origen: String, destino: String, hora: Int,
minuto: Int): Itinerario = {
      val itinerariosValidos = buscar(origen, destino).par.filter {
itinerario =>
      tiempoEnMinutos(itinerario.last.hl, itinerario.last.ml) <=
tiempoEnMinutos(hora, minuto)
      }.seq
      if (itinerariosValidos.isEmpty) List()
      else itinerariosValidos.maxBy(itinerario => (itinerario.head.hs,
itinerario.head.ms))
      }

      // Retornamos una función que toma origen, destino, hora y minuto, y
retorna el itinerario optimizado
      (origen: String, destino: String, hora: Int, minuto: Int) =>
optimizarSalida(origen, destino, hora, minuto)
  }
```

Informe - Argumentación corrección de funciones desarrolladas

El informe presenta la argumentación para justificar la corrección de las funciones desarrolladas (itinerarios,itinerariosTiempo, itinerariosEscalas, itinerariosAire, itinerariosSalida).

itinerarios

∀(origen, destino) ∈ Aeropuertos, itinerarios(vuelos, aeropuertos)(origen, destino) = {resultado,
si ∃una secuencia en Ui=1n∑i tal que se cumpla la condición del itinerario, V λ, en otro caso}

Donde:

∑ = conjunto de todos los vuelos posibles (Vuelo)
resultado = lista de listas de vuelos (Itinerario) que representan todas las posibles rutas desde el origen hasta el destino, sin repetir aeropuertos visitados
λ = lista vacía
∀n ∈ N, buscar(origen, destino, visitados) es una función que encuentra todas las secuencias posibles de vuelos desde origen hasta destino sin repetir aeropuertos ya visitados

itinerariosTiempo

∀(origen, destino) ∈ Aeropuertos, itinerariosTiempo(vuelos, aeropuertos)(origen, destino) = {resultado, si ∃una secuencia en Ui=1n∑i tal que se cumpla la condición del itinerario y se ordene por tiempo total ascendente, V λ, en otro caso}

Donde:

∑ = {conjunto de todos los vuelos posibles (Vuelo)}
resultado = lista de listas de vuelos (Itinerario) que representan las tres rutas con el menor tiempo total desde el origen hasta el destino, sin repetir aeropuertos visitados
λ = lista vacía
∀n ∈ N, buscar(origen, destino, visitados) es una función que encuentra todas las secuencias posibles de vuelos desde origen hasta destino sin repetir aeropuertos ya visitados

itinerariosEscalas

∀(origen, destino) ∈ Aeropuertos, itinerariosEscalas(vuelos, aeropuertos)(origen, destino) = {resultado, si ∃una secuencia en Ui=1n∑i tal que se cumpla la condición del itinerario y se ordene por número de escalas ascendente, V λ, en otro caso}

Donde:

∑ = {conjunto de todos los vuelos posibles (Vuelo)}

resultado = lista de listas de vuelos (Itinerario) que representan las tres rutas con el menor número de escalas desde el origen hasta el destino, sin repetir aeropuertos visitados

λ = lista vacía

∀ n ∈ N, buscar(origen, destino, visitados) es una función que encuentra todas las secuencias posibles de vuelos desde origen hasta destino sin repetir aeropuertos ya visitados

1. La función **itinerariosEscalas** genera una función que, dada una lista de vuelos y aeropuertos, encuentra todas las posibles rutas desde un aeropuerto de origen a un aeropuerto de destino, evitando ciclos y asegurando que cada aeropuerto se visita solo una vez.
2. Luego, se calcula el número de escalas para cada itinerario encontrado.
3. Los itinerarios se ordenan por número de escalas en orden ascendente.
4. Se retornan los tres primeros itinerarios de la lista ordenada.

La función **itinerariosEscalas** genera una función que, dada una lista de vuelos y aeropuertos, encuentra todas las posibles rutas desde un aeropuerto de origen a un aeropuerto de destino, evitando ciclos y asegurando que cada aeropuerto se visita solo una vez.
Luego, se calcula el número de escalas para cada itinerario encontrado.
Los itinerarios se ordenan por número de escalas en orden ascendente.
Se retornan los tres primeros itinerarios de la lista ordenada.

itinerariosAire

∀ (origen, destino) ∈ Aeropuertos, itinerariosAire(vuelos, aeropuertos)(origen, destino) = {resultado, si ∃una secuencia en Ui=1n∑i tal que se cumpla la condición del itinerario y se ordene por tiempo total en aire ascendente, V λ, en otro caso}

Donde:

∑ = {conjunto de todos los vuelos posibles (Vuelo)}

resultado = lista de listas de vuelos (Itinerario) que representan las tres rutas con el menor tiempo total en aire desde el origen hasta el destino, sin repetir aeropuertos visitados

λ = lista vacía

∀ n ∈ N, buscar(origen, destino, visitados) es una función que encuentra todas las secuencias posibles de vuelos desde origen hasta destino sin repetir aeropuertos ya visitados

itinerarioSalida

∀ (origen, destino, hora, minuto) ∈ Aeropuertos × Aeropuertos × ℤ × ℤ, itinerarioSalida(vuelos, aeropuertos)(origen, destino, hora, minuto) = {resultado, si ∃una secuencia en Ui=1n∑i tal que se cumpla la condición del itinerario y se ordene por hora de salida y minutos ascendentes, V λ, en otro caso}

Donde:

∑ = {conjunto de todos los vuelos posibles (Vuelo)}

resultado = lista de vuelos (Itinerario) que representan una ruta desde el origen hasta

el destino, con la hora de salida más tardía menor o igual a la hora y minutos especificados, sin repetir aeropuertos visitados

λ = lista vacía

$\forall n \in N$, buscar(origen, destino, visitados) es una función que encuentra todas las secuencias posibles de vuelos desde origen hasta destino sin repetir aeropuertos ya visitados

Informe - uso e impacto de técnicas de paralelización de tareas y de datos

El informe describe las técnicas de paralelización de tareas y datos usadas, dónde las usaron, y qué impacto tuvieron en el desempeño de la solución. El informe presenta tablas, cuadros y/o gráficas, con los resultados de las pruebas y con la argumentación coherente sobre el impacto positivo o negativo de la paralelización, y el grado de aceleración logrado.

itinerarios vs itinerariosPar			
longitud lista	**Tiempo de Ejecucion [ms]**		
[n]	**reconstruirCadenaIngenuo**	**reconstruirCadenaIngenuoPar**	**Aceleración**

itinerariosTiempo vs itinerariosTiempoPar			
longitud lista	**Tiempo de Ejecucion [ms]**		
[n]	**itinerariosTiempo**	**itinerariosTiempoPar**	**Aceleración**

itinerariosEscalas vs itinerariosEscalasPar			
longitud lista	**Tiempo de Ejecucion [ms]**		
[n]	**itinerariosEscalas**	**itinerariosEscalasPar**	**Aceleración**

itinerariosAire vs itinerariosAirePar			
longitud lista	**Tiempo de Ejecucion [ms]**		
n	**itinerariosAire**	**itinerariosAirePar**	**Aceleración**

itinerariosSalida vs itinerariosSalidaPar			
longitud lista	**Tiempo de Ejecucion [ms]**		
n	**itinerariosSalida**	**itinerariosSalidaPar**	**Aceleración**