Taller 4: Colecciones y Expresiones For

Juan Francisco Díaz Frias Emily Núñez

Octubre 2024

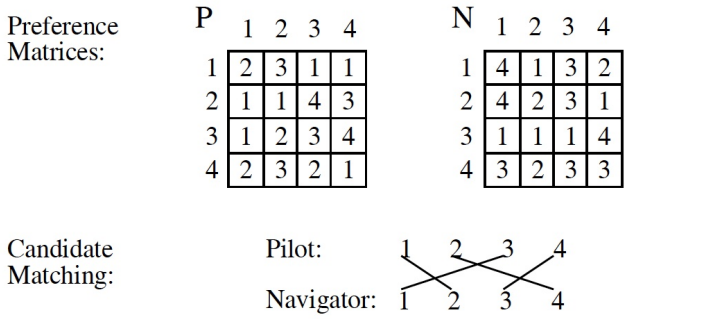
1. El problema de la asignación de la tripulación de un vuelo

Suponga que tenemos n pilotos y n copilotos para asignarlos a diferentes vuelos. Y que tenemos una matriz $P[1..n, 1..n]$ (las filas son los pilotos y las columnas los copilotos) tal que $P[i, j]$ es un número entero en el intervalo $[1, 5]$ que indica el nivel de preferencia que tiene el piloto i para trabajar junto al copiloto j; y una matriz $N[1..n, 1..n]$ (las filas son los copilotos y las columnas los pilotos) tal que $N[i, j]$ es un número entero en el intervalo $[1, 5]$ que indica el nivel de preferencia que tiene el copiloto i para trabajar junto al piloto j. El peso de emparejar al piloto i con el copiloto j se define como $P[i, j] \times N[j, i]$.

El problema de la asignación de la tripulación de un vuelo consiste en emparejar cada piloto con un copiloto de manera tal que la suma de los pesos de los emparejamientos se maximice.

Su tarea es desarrollar un programa que genere todas las formas de emparejar las n pilotos con los n copilotos y escoger entre ellas la mejor.

Por ejemplo si $n = 4$ y P, N son las de la figura, un posible emparejamiento y su peso se ven en la figura:



Representación de los datos Para modelar un emparejamiento, una lista de emparejamientos y las matrices de preferencias se tienen los siguientes tipos de datos:

```
type Match= (Int, Int)
type Matching = List[Match]
type Preferences = Vector[Vector[Int]]
```

Así, la tupla $(2, 3)$ representa el emparejamiento del piloto número 2 con el copiloto número 3. La lista $List((1, 2), (2, 3), (3, 1))$ representa una lista de emparejamientos de los pilotos 1, 2, y 3 con los copilotos 1, 2 y 3. La lista $List((1, 1), (2, 3), (3, 2))$ representa otra lista de emparejamientos posible. En cambio la lista $List((1, 1), (2, 1), (3, 2))$ no representa una lista de emparejamientos correcta (¿por qué?).

Para el ejemplo de la figura con $n = 4$, la entrada sería:

```
val pilot = Vector(Vector(2,3,1,1), Vector(1,1,4,3), Vector(1,2,3,4), Vector(2,3,2,1))
val navig = Vector(Vector(4,1,3,2), Vector(4,2,3,1), Vector(1,1,1,4), Vector(3,2,3,3))
```

La salida que muestra todos los emparejamientos posibles y sus pesos sería:

```
List((List((1,1), (2,2), (3,3), (4,4)),16), (List((1,1), (2,2), (3,4), (4,3)),30),
(List((1,1), (2,3), (3,2), (4,4)),21),(List((1,1), (2,3), (3,4), (4,2)),27),
(List((1,1), (2,4), (3,2), (4,3)),28), (List((1,1), (2,4), (3,3), (4,2)),20),
(List((1,2), (2,1), (3,3), (4,4)),19), (List((1,2), (2,1), (3,4), (4,3)),33),
(List((1,2), (2,3), (3,1), (4,4)),22), (List((1,2), (2,3), (3,4), (4,1)),32),
(List((1,2), (2,4), (3,1), (4,3)),29), (List((1,2), (2,4), (3,3), (4,1)),25),
(List((1,3), (2,1), (3,2), (4,4)),11), (List((1,3), (2,1), (3,4), (4,2)),17),
(List((1,3), (2,2), (3,1), (4,4)),9), (List((1,3), (2,2), (3,4), (4,1)),19),
(List((1,3), (2,4), (3,1), (4,2)),13), (List((1,3), (2,4), (3,2), (4,1)),17),
(List((1,4), (2...
```

Nótese que el emparejamiento de la figura $(List((1, 2), (2, 4), (3, 1), (4, 3)), 29)$ hace parte de la lista de emparejamientos posibles. Sin embargo, al buscar el mejor, el resultado es:

```
(List((1,2), (2,1), (3,4), (4,3)),33)
```

1.1. Emparejamientos posibles de un piloto

Implemente la función `matchByElement` que reciba i, el número que representa un piloto, $(1 \leq i \leq n)$ y n y devuelva la lista de posibles emparejamientos del piloto i.

```
  def matchByElement(i:Int, n:Int):List[Match] = {
    // Devuelve la lista de los posibles emparejamientos del piloto i
...
  }
```

Por ejemplo una invocación a $matchByElement(2, 5)$ devuelve:

```
val res0: List[MatchingProblem.Match] = List((2,1), (2,2), (2,3), (2,4), (2,5))
```

indicando que el piloto 2, puede llegar a tener como copiloto al 1, 2, 3, 4 o 5:

1.2. Emparejamientos posibles de todos los pilotos

Implemente la función `matchsByElements` que reciba n, el número de pilotos y copilotos y devuelva la lista de todos los emparejamientos posibles por cada piloto. Su implementación debe invocar a la función definida en el punto anterior.

```
  def matchsByElements(n:Int): List[List[Match]] = {
    // de vuelve la lista de los posibes matchs de cada elemento (1 hasta n)
...
  }
```

Por ejemplo una invocación a $matchsByElements(5)$ devuelve:

```
val res2: List[List[MatchingProblem.Match]] = List(
List((1,1), (1,2), (1,3), (1,4), (1,5)), List((2,1), (2,2), (2,3), (2,4), (2,5)),
List((3,1), (3,2), (3,3), (3,4), (3,5)), List((4,1), (4,2), (4,3), (4,4), (4,5)),
List((5,1), (5,2), (5,3), (5,4), (5,5)))
```

1.3. Posibles emparejamientos

Implemente la función `possibleMatchings` que dado n, el número de pilotos y copilotos, devuelva una lista de todos los emparejamientos posibles de los pilotos 1 a n con los copilotos 1 a n:

Note que esta función no tiene en cuenta que todos los copilotos sea utilizados, por tanto puede calcular combinaciones que no hagan parte de la respuesta buscada en este ejercicio. Su implementación debe invocar a la función definida en el punto anterior.

```
  def possibleMatchings(n:Int):List[List[Match]] = {
...
  }
```

Por ejemplo una invocación a $possibleMatchings(2)$ devuelve:

```
val res3: List[List[MatchingProblem.Match]] = List(
List((1,1), (2,1)),
List((1,1), (2,2)),
List((1,2), (2,1)),
List((1,2), (2,2)))
```

Nótese que en esta lista de emparejamientos, cada emparejamiento está compuesta de una lista de n pilotos, del 1 al n emparejados con un copiloto. Entre todos estos emparejamientos, algunos contienen los n copilotos, y el resto no. Luego necesitamos encontrar de esta lista las que sí contienen los n copilotos.

En ese ejemplo los emparejamientos $List((1,1),(2,1))$ y $List((1,2),(2,2)))$ no usan todos los copilotos, por tanto no hacen parte de los emparejamientos que buscamos para resolver el problema. En cambio los emparejamientos $List((1,1),(2,2)), List((1,2),(2,1))$ si son el tipo de emparejamientos que buscamos. Eso quiere decir que todavía nos hace falta filtrar los emparejamientos de esta lista.

1.4. Calculando los emparejamientos válidos

Implemente la función `matchings` que dado n, el número de pilotos y copilotos, devuelva la lista de todos los emparejamientos posibles de los pilotos 1 a n que usan todos los copilotos 1 a n, es decir que calcula todos los emparejamientos válidos. Esta función debe usar la función definida en el punto anterior.

La salida vendrá en una lista de listas, de n emparejamientos cada una, de la forma $List((1,c_1),(2,c_2),\ldots,(n,c_n))$ tal que $\{c_1,c_2,\ldots,c_n\} = \{1,2,\ldots,n\}$.

```
  def matchings (n:Int): List[Matching] = {
    // Devuelve la lista de todos los posibles matchings de los n pilotos
...
  }
```

Por ejemplo una invocación a $matchings(2)$ devuelve:

```
val res6: List[MatchingProblem.Matching] = List(
List((1,1), (2,2)),
List((1,2), (2,1)))
```

1.5. Calculando los pesos de los emparejamientos válidos

Implemente la función `weightedMatchings` que dado n, el número de pilotos y copilotos, $pilotPrefs$ y $navigPrefs$, las matrices de preferencias de los pilotos y copilotos, devuelva la lista de todos los emparejamientos posibles de los pilotos 1 a n que usan todos los copilotos 1 a n, junto con su peso, es decir que calcula todos los emparejamientos válidos con su peso. Esta función debe usar la función definida en el punto anterior.

```
  def weightedMatchings(n:Int, pilotPrefs:Preferences, navigPrefs:Preferences):List[(Matching,Int)] = {
...
}
```

Por ejemplo una invocación a $weightedMatchings(2, pilot, navig)$ tomando sólo las dos primeras filas y columnas de las preferencias del ejemplo de la gráfica al comienzo, devuelve:

```
val res10: List[(MatchingProblem.Matching, Int)] = List(
(List((1,1), (2,2)),10),
(List((1,2), (2,1)),13))
```

Su solución debe usar expresiones for y utilizar la función `matchings` definida en el ejercicio anterior. Puede utilizar las funciones auxiliares que considere necesarias.

1.6. Calculando el mejor emparejamiento

Implementa la función `bestMatching` que dado n, el número de pilotos y copilotos, $pilotPrefs$ y $navigPrefs$, las matrices de preferencias de los pilotos y copilotos, devuelva el emparejamiento válido de pilotos y copilotos de mayor peso.

```
  def bestMatching(n:Int, pilotPrefs:Preferences, navigPrefs:Preferences):(Matching,Int) = {
...
  }
```

Por ejemplo una invocación a $bestMatching(2, pilot, navig)$ tomando sólo las dos primeras filas y columnas de las preferencias del ejemplo de la gráfica al comienzo, devuelve:

```
val res13: (MatchingProblem.Matching, Int) = (List((1,2), (2,1)),13)
```

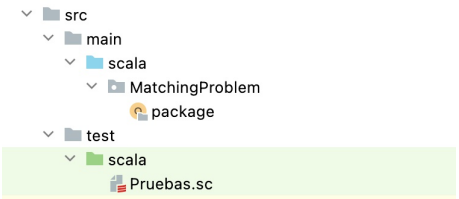
Y una invocación a $bestMatching(4, pilot, navig)$ que corresponde al ejemplo de la gráfica, devuelve:

```
val res15: (MatchingProblem.Matching, Int) = (List((1,2), (2,1), (3,4), (4,3)),33)
```

2. Entrega

2.1. Paquete *MatchingProblem* y *worksheet* de pruebas

Usted deberá entregar dos archivos `package.scala` y `pruebas.sc` los cuales harán parte de su estructura de proyecto `IntellijIdea`, como se muestra en la figura a continuación:



Las funciones correspondientes a cada ejercicio, `matchByElement, matchsByElements, possibleMatchings, matchings, weightedMatchings` y `bestMatching` deben ser implementadas en un paquete de Scala denominado `MatchingProblem`. **Utilice expresiones for para implementar todas la funciones. Sólo en la última podría no usarlas.** En ese paquete debe venir un archivo denominado `package.scala` que debe tener la forma siguiente:

```
package object MatchingProblem {
  type Match= (Int, Int)
  type Matching = List[Match]
  type Preferences = Vector[Vector[Int]]

  def matchByElement(i:Int, n:Int):List[Match] = {
    // Devuelve la lista de los posibles match del elemento i
...
  }

  def matchsByElements(n:Int): List[List[Match]] = {
    // de vuelve la lista de los posibes matchs de cada elemento (1 hasta n)
...
  }

  def possibleMatchings(n:Int):List[List[Match]] = {
    // Devuelve la lista de todos los posibles matchings de cada uno de los n
    // elementos.
...
  }

  def matchings (n:Int): List[Matching] = {
    // Devuelve la lista de todos los posibles matchings de  n pilotos
...
  }

  def weightedMatchings(n:Int, pilotPrefs:Preferences, navigPrefs:Preferences):List[(Matching,Int)] = {
...
  }

  def bestMatching(n:Int, pilotPrefs:Preferences, navigPrefs:Preferences):(Matching,Int) = {
...
  }
}
```

Dicho paquete será usado en un *worksheet* de Scala con casos de prueba. Estos casos de prueba deben venir en un archivo denominado `pruebas.sc` . Un ejemplo de un tal archivo es el siguiente:

```
import MatchingProblem._

val pilot = Vector(Vector(2,3,1,1), Vector(1,1,4,3), Vector(1,2,3,4), Vector(2,3,2,1))
val navig = Vector(Vector(4,1,3,2), Vector(4,2,3,1), Vector(1,1,1,4), Vector(3,2,3,3))

matchByElement(2,5)
matchByElement(3,5)
matchsByElements(5)
possibleMatchings(2)
possibleMatchings(3)
possibleMatchings(5)
matchings(2)
matchings(3)
matchings(4)
matchings(5)
weightedMatchings(2,pilot,navig)
weightedMatchings(3,pilot,navig)
weightedMatchings(4,pilot,navig)
bestMatching(2,pilot,navig)
bestMatching(3,pilot,navig)
bestMatching(4,pilot,navig)
```

2.2. Informe del taller - secciones

Todo taller debe venir acompañado de un informe en formato pdf. El informe debe contener la información explícita solicitada en el enunciado.

Para este caso, el informe del taller debe contener al menos tres secciones: informe de uso de colecciones y expresiones for, informe de corrección y conclusiones.

2.2.1. Informe de uso de colecciones y expresiones for

Tal como se ha visto en clase, el uso de colecciones y expresiones for es una herramienta muy poderosa y expresiva para programar. En esta sección usted debe hacer una tabla indicando en cuáles funciones usó la técnica y en cuáles no. Y en las que no, indicar por qué no lo hizo.

También se espera una apreciación corta de su parte, sobre el uso de colecciones y expresiones for como técnica de programación.

2.2.2. Informe de corrección

Es muy importante reflexionar sobre la corrección del código entregado. Para ello se deberá argumentar sobre la corrección de las funciones entregadas, y también deberá entregar un conjunto de pruebas. Todo esto lo consigna en esta sección del informe, dividida de la siguiente manera:

Argumentación sobre la corrección .

Para cada función, `matchByElement, matchsByElements, possibleMatchings, matchings, weightedMatchings` y `bestMatching`, argumente lo más formalmente posible por qué es correcta. Utilice inducción o inducción estructural donde lo vea pertinente. Estas argumentaciones las consigna en esta sección del informe.

Casos de prueba .

Para cada función se requieren mínimo 5 casos de prueba donde se conozca claramente el valor esperado, y se pueda evidenciar que el valor calculado por la función corresponde con el valor esperado en toda ocasión.

Una descripción de los casos de prueba diseñados, sus resultados y una argumentación de si cree o no que los casos de prueba son suficientes para confiar en la corrección de cada uno de sus programas, los registra en esta sección del informe. Obviamente, esta parte del informe debe ser coherente con el archivo de pruebas entregado, `pruebas.sc`.

2.3. Fecha y medio de entrega

Todo lo anterior, es decir los archivos `package.scala`, `pruebas.sc`, e `Informe del taller`, debe ser entregado vía el campus virtual, a más tardar a las 10 a.m. del jueves 17 de octubre de 2024, en un archivo comprimido que traiga estos tres elementos.

Cualquier indicación adicional será informada vía el foro del campus asociado a *programación funcional*.

3. Evaluación

Cada taller será evaluado tanto por el profesor del curso con ayuda del monitor, como por 2 compañeros que hayan entregado el taller. A este tipo de evaluación se le conoce como *Coevaluación*.

El objetivo de la coevaluación es lograr aprendizajes a través de:

- La lectura de lo que otros compañeros hicieron ante el mismo reto. Esto permite contrastar las soluciones propias con las de otros, y aprender de ellas, o compartir mejores maneras de hacer algo con otros.
- Retroalimentar a los compañeros que fueron asignados para evaluar. Al escribir la percepción que tenemos sobre el trabajo del otro, podemos aprender de cómo lo hicieron, y dar indicaciones al otro sobre otra formas de hacer lo mismo o, incluso, felicitarlo por la solución que presenta.
- La lectura de las retroalimentaciones de mis compañeros o del profesor/monitor.

La calificación de cada taller corresponderá entonces:

- en un 80 % a la calificación ponderada que reciba del profesor/monitor, vía una rúbrica de evaluación (pesa 5 veces lo que pesa la de otro estudiante) y de los tres compañeros asignados para evaluarlo.
- en un 20 % a la calificación que el sistema hará del trabajo de evaluación asignado. El Sistema tiene un método inteligente para estimar esa calificación, a partir de las evaluaciones realizadas por cada estudiante y por el profesor/monitor.