PRACTICA 2: SCALA 16384

AMPLIACIÓN DE PROGRAMACIÓN AVANZADA

Nombre y apellidos

Luis Alejandro Cabanillas Prudencio

Alvaro de las Heras Fernández

11/4/2019

1. Trabajo en el laboratorio: primeras funciones

Durante las clases de laboratorio implementamos una serie de funciones básicas que nos sirvieron de punto de partida de la práctica:

Obtener y Poner

La función **obtener** fue la primera que implementamos en la práctica y su objetivo es devolver el valor de una posición del tablero. Recibe como parámetros el tablero y el índice, la posición en el tablero de la que se obtendrá el valor. Si el índice es la cabeza del tablero la devolvemos y acaba nuestro método, pero si no lo es volvemos a llamar a la función con la cola y con índice menos uno; así iremos recorriendo el tablero hasta llegar a la posición que nos interese y devolveremos su valor:

```
def obtener(tablero: List[Int], indice: Int): Int = {
  //Si el tablero no esta vacio
  if (tablero.length > 0) {
    //Si el indice es la cabeza se devuelve
    if (indice == 1) tablero.head
    //Si no se quita otro elemento y se reduce el indice
    else obtener(tablero.tail, indice - 1)
    //Si esta fuera de rango devuelve -1
  } else -1
}
```

Figura 1: Función obtener

Por otro lado, tenemos la función **poner**, de la cual desarrollamos dos versiones. En una versión esta función se encargaría de poner un elemento en una posición dada del tablero y en la otra versión, que representaría una mejora de la primera, pondrá un elemento con valor aleatorio en una posición, en función de la dificultad. Esto lo utilizaremos para la generación de semillas en cada movimiento. A continuación, en la figura 2, se muestra la función poner mejorada:

```
def poner(lista: List[Int], dificultad: Int): List[Int] = {
  //Se genera una posicion y un valor aleatorios
  val pos = crearRandomPos(lista.length)
  val valor = crearValorRandom(dificultad)
  //Comprueba si esta vacia
  if (lista.length == 0) Nil
  else {
    //Si la posicion esta vacia
    if (obtener(lista, pos) == 0)
    //Coloca el valor en ella
      poner(lista, valor, pos)
    else poner(lista, dificultad)
  }
}
```

Figura 2: Función poner (versión mejorada)

Generación y rellenado del tablero

Para generar el tablero, que realmente es generar una lista de ceros en función del tamaño, lo único que tenemos que hacer es concatenar un 0 a la llamada recursiva de la función con tamaño – 1, que parará cuando el tamaño es 0 y devolverá la lista que se ha ido generando con cada llamada recursiva. En la figura 3 podemos ver con detalle este método.

```
def generarTab(tam: Int): List[Int] = {
  if (tam == 0) Nil
  //Llamada recursiva hasta que el tamano sea el deseado
  else 0 :: generarTab(tam - 1)
}
```

Figura 3: Función que genera el tablero

Una vez tenemos el tablero con todas las casillas vacías (ceros), tenemos que rellenarlo en función de la dificultad, ya que cuanto mayor sea la dificultad se generarán más semillas con diferentes valores. Para ello, utilizamos en nuestro programa la función rellenarTab. Esta función va comprobando si quedan huecos libres en el tablero y si el número de casillas a rellenar es distinto de cero; en caso de que se cumplan las dos condiciones, llamaremos recursivamente a rellenarTab con poner (para poner una determinada semilla) y con el número de casillas menos 1. En la figura 4 podemos ver el funcionamiento de rellenarTab.

```
def rellenarTab(tablero: List[Int], numCasillas: Int, dificultad: Int): List[Int] = {
  if (tablero.length == 0) Nil
  //Mientras haya huecos y casillas se rellena el tablero
  else if ((huecosLibres(tablero) > 0) && (numCasillas != 0)) {
    rellenarTab(poner(tablero, dificultad), numCasillas - 1, dificultad)
  }
  //Finalmente devuelve el tablero
  else tablero
}
```

Figura 4: Función que rellena el tablero en función de la dificultad

Por último, podemos imprimir el tablero. Para ello, tenemos la función imprimir. Esta función recibe el tablero y un entero que representa el número de filas y columnas del tablero. Mientras el tablero no esté vacío, va comprobando elemento por elemento cuándo tiene que pasar a la siguiente línea (ya que es una lista y hay que dejarlo con aspecto de matriz) con el resto de la división de la longitud de la lista entre las columnas. Adicionalmente, se llama al método espacios, que juntamente con el método dígitos, dejan la matriz más limpia a la vista del usuario. El método imprimir puede verse en la figura 5 y el aspecto del tablero por pantalla en la figura 6.

```scala
def imprimir(lista: List[Int], columnas: Int): Unit = {
  if (lista.length > 0) {
    //Cuando llega a la ultima columna pasa a la siguiente fila
    if (lista.length % columnas == 0) print("\n|")
    //Espacios que corrigen visualmente los valores del tablero
    val corregirEspacios = espacios(lista.head)
    //Imprime los valores del tablero con los espacios ajustados
    print(corregirEspacios + lista.head + "|")
    //Se vuelve a llamar a imprimir
    imprimir(lista.tail, columnas)
  }
}
```

Figura 5: Método imprimir

```
|      0|      0|      0|      0|
|      0|      0|      0|      0|
|      0|      0|      2|      0|
|      2|      0|      0|      0|
```

Figura 6: Aspecto de la matriz por pantalla

Funciones auxiliares del movimiento

Una vez tenemos las funciones básicas para crear y acceder al tablero, tenemos que centrarnos en las funciones que utilizará el movimiento.

Primero, creamos la función **eliminar**; que pone una casilla a cero. Esta función será útil cuando movamos una semilla de casilla, ya que tendremos que poner a cero el valor donde se encontraba. A eliminar le entra un tablero y un índice, y va comprobando si el índice es la cabeza del tablero con llamadas recursivas con índice-1.

Otra función que usaremos bastante en nuestro programa es la función **reverse**. Esta función invierte el orden de una lista y la utilizaremos en los movimientos arriba y a la izquierda; que serán igual que los movimientos abajo y a la derecha, pero con el reverse aplicado. En las figuras 7 puede verse detalladamente la estructura de reverse.

```
def reverse(lista: List[Int]): List[Int] = {
  //Si es vacia devuelve la lista
  if (lista.length == 0) lista
  //Se anade el valor actual al final para construirla
  else reverse(lista.tail) ::: lista.head :: Nil
}
```

Figura 7: Función reverse

Como se explicará posteriormente, el conteo de casillas combinadas y la puntuación se almacenarán en las últimas dos posiciones del tablero. Por ello, deberemos mover el tablero sin estas dos últimas posiciones para que se lleven a cabo correctamente los movimientos. Para conseguir esto, tenemos la función quitarHasta, que recibe un tablero y un entero y quita del tablero las posiciones que se indiquen en el entero. Esto lo conseguimos comprobando si la longitud del tablero es igual al índice y, en caso contrario, concatenamos la cabeza del tablero con la llamada recursiva a la función con la cola del tablero. Su funcionamiento detallado puede verse en la figura 8.

```
def quitarHasta(tablero: List[Int], limite: Int): List[Int] = {
  //Cuando llega al limite para
  if (tablero.length == limite) Nil
  //Si no continua anadiendo
  else tablero.head :: quitarHasta(tablero.tail, limite)
}
```

Figura 8: Función quitarHasta

2.Implementación del movimiento

Una parte fundamental del núcleo del juego es el movimiento, este movimiento se realiza sobre matrices cuadradas en 4 direcciones posibles, identificadas con las teclas w, s, a y d.

Movimiento

El **movimiento simple** de una dirección mueve **elemento** a **elemento** los valores de la matriz, moviendo cada vez como **máximo** una **fila** o **columna** los elementos del tablero. Su funcionamiento se basa en **llamadas recursivas** que van recorriendo el tablero cada vez más pequeño fruto de aplicar *tail*, hasta llegar a la **condición de parada** que es cuando esté **vacío**. En ese caso devuelve el tablero que se irá **construyendo** con los **valores** que se han ido guardando en la **pila**. Estos valores podrán haber cambiado de posición o no **dependiendo** de si había un 0 (**hueco**) o no en su **fila** o **columna posterior**, esto en el código se hace con **columnas + 1** (fila siguiente) o **2** (siguiente columna), si hay un 0 se procede a realizar el cambio

eliminando el valor de la **posición actual** con *eliminar* y **poniendo** el valor en la **posición** que se ha **comprobado**. Una vez se han hecho todas las llamadas recursivas y se ha devuelto el tablero el **resultado** sería el tablero, pero **sin** una **fila** o **columna** de **huecos,** la Fig. 9 es la implementación de esto. Cabe destacar que en el caso de movimientos **horizontales** es necesario realizar más **comprobaciones** al trabajar con una lista de **1 dimensión**, que se hace con el **tamaño** de la lista y la operación **resto**.

```
/**
  * Mueve a la derecha una columna completa todos los valores si hay hueco
(valor es 0)
  *
  * @param tablero  tablero que se movera
  * @param columnas Numero de columnas del tablero
  * @return devuelve el tablero con la columna movida
  */
def moverDerecha(tablero: List[Int], columnas: Int): List[Int] = {
  //Comprueba si esta vacio
  if (tablero.length > 0) {
    //Comprueba si el valor actual es distinto de cero
    if ((tablero.head > 0)) {
      //Si hay un hueco lo baja poniendolo en la posicion y borrando el
actual
      if ((obtener(tablero, 2) == 0) && ((tablero.length) % columnas != 1))
moverDerecha(poner(eliminar(tablero, 1), tablero.head, 2), columnas)
      //Si no continuamos recorriendo el tablero
      else tablero.head :: moverDerecha(tablero.tail, columnas)
      //Si no continuamos recorriendo el tablero
    } else tablero.head :: moverDerecha(tablero.tail, columnas)
    //Si es vacio se devuelve el tablero
  } else tablero
}
```

Figura 9: Función que desplaza todos los valores una columna hacia la derecha.

Además, se ha **reutilizado** código mediante la función *reverse* (desarrollada en el código) se han implementado los **movimientos opuestos**, simplemente pasando los **datos invertidos** y devolviendo el **resultado invertido**, este se puede observar con *moverArriba* (véase figura 10).

```
/**
  * Mueve arriba una fila completa si hay hueco (valor es 0) empleando
mover abajo
  *
  * @param tablero  tablero que se movera
  * @param columnas Numero de columnas del tablero
  * @return devuelve el tablero con la fila subida
  */
def moverArriba(tablero: List[Int], columnas: Int): List[Int] = {
  reverse(moverAbajo(reverse(tablero), columnas))
}
```

Figura 10: Función que desplaza todos los valores una fila hacia la arriba haciendo uso de reverse.

El objetivo una vez desarrollado el movimiento simple es que mediante la aplicación de varias **iteraciones** se consiga **agrupar** todos los **valores** sin huecos

intermedios, esto se hace de forma **recursiva** con las funciones *moverTodo*, que llamarán al **movimiento simple** el **número** de **filas** o **columnas** que haya, de tal forma que **asegure** la **eliminación** de todos los **huecos** entre las filas o columnas, con el menor número posible de llamadas. Un ejemplo es la función moverTodoArriba de la figura 12.

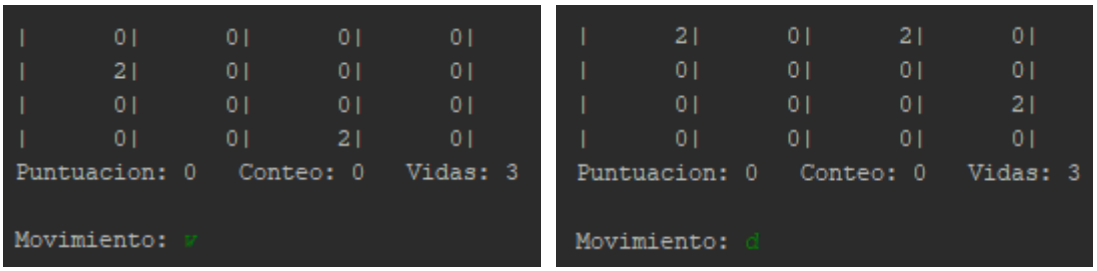

Figura 11: Muestra de funcionamiento del movimiento total del juego.

```
/**
  * Mueve todo hacia arriba quitando los huecos
  *
  * @param tablero     tablero que movera
  * @param movimientos movimientos maximos para asegurar ningun hueco
  * @param columnas    columnas del tablero
  * @return el tablero con todos los valores en el borde superior
  */
def moverTodoArriba(tablero: List[Int], movimientos: Int, columnas: Int):
List[Int] = {
  //Lo mueve arriba hasta que movimientos sea 1
  if (movimientos == 1) tablero
  else moverTodoArriba(moverArriba(tablero, columnas), movimientos - 1,
columnas)
}
```

Figura 12: Función que mueve todos los valores en una dirección sin dejar huecos.

Suma en el movimiento

Con todos los valores **agrupados** en uno de los bordes el siguiente proceso es **sumar** los valores de **igual valor**. Este proceso se tiene que implementar a parte porque a pesar de ser **similar** a **movimiento**, pero se **repetiría** varias veces cada vez lo que daría como **resultado** varias **sumas** sucesivas sobre **mismos valores**, en vez de una **única suma**. La diferencia que hay con la función de movimiento simple se encuentra en las **condiciones** cuando encuentra dos valores **iguales** contiguos, en ese caso se pone el **doble** del **valor** en la posición **actual** y a **0** el valor que se ha **comprobado**, de esta forma en las sucesivas llamadas se evita **duplicar** datos y se **actualiza** también la **puntuación** y el **conteo** (Véase figura 13).

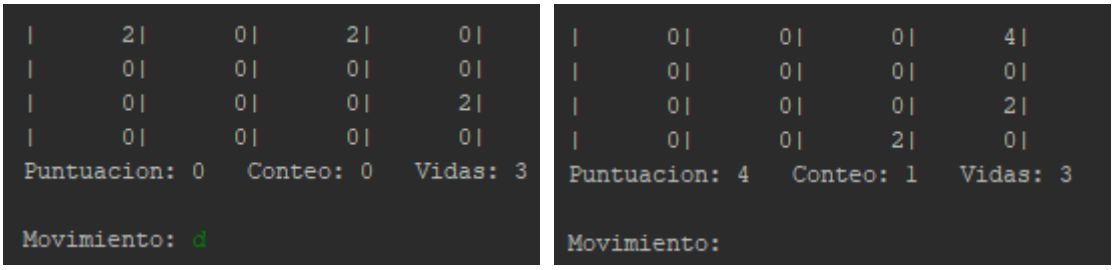

Figura 13: Suma de dos columnas con conteo y puntuación mediante la función de suma

Otra importante diferencia está con el **cálculo** de la **puntuación** y **conteo** que se han añadido, estos parámetros permiten calcularlos cada vez que se llaman pudiendo actualizarse como en el caso anterior. Estos se devuelven al **final** de **tablero** como dos **valores** enteros más, calculándolo todo de una vez lo que implica un **ahorro** en **costes** de cálculo.

Únicamente hay **dos** funciones de **suma**, *sumaVertical* y *sumaHorizontal*, que son en concreto variaciones de *moverAbajo* y *moverDerecha*. Según la dirección que tenga luego se aplicará *reverse* o no, para obtener la suma de la dirección correcta.

```
/**
  * Suma los valores horizontalmente una unica vez
  *
  * @param tablero    tablero que sumara
  * @param columnas   columnas del tablero
  * @param puntuacion puntuacion que se sumara
  * @param conteo     cantidad de combinaciones a sumar
  * @return el tablero sumado mas la puntuacion y conteo
  */
def sumarHorizontal(tablero: List[Int], columnas: Int, puntuacion: Int,
conteo: Int): List[Int] = {
  //Comprueba si no esta vacio
  if (tablero.length > 0) {
    //Comprueba si el actual es distinto de 0
    if ((tablero.head > 0)) {
      //Comprueba si el siguiente coincide en valor
      if ((obtener(tablero, 2) == tablero.head) && ((tablero.length) %
columnas != 1))
      //Si lo hace lo suma poniendo a 0 el nuevo valor y dejando el actual
con el doble ademas de calcular puntos y conteo
        sumarHorizontal(poner(eliminar(tablero, 2), tablero.head * 2, 1),
columnas, puntuacion + tablero.head * 2, conteo + 1)
      //Si no se sigue recorriendo
      else tablero.head :: sumarHorizontal(tablero.tail, columnas,
puntuacion, conteo)
      //Si no se sigue recorriendo
    } else tablero.head :: sumarHorizontal(tablero.tail, columnas,
puntuacion, conteo)
    //Si es vacio se anade la puntuacion y conteo al final
  } else tablero ::: puntuacion :: conteo :: Nil
}
/**
  * Suma los valores verticalmente una unica vez
  *
  * @param tablero    tablero que sumara
  * @param columnas   columnas del tablero
```

```
  * @param puntuacion puntuacion que se sumara
  * @param conteo     cantidad de combinaciones a sumar
  * @return el tablero sumado mas la puntuacion y conteo
  */
def sumarVertical(tablero: List[Int], columnas: Int, puntuacion: Int,
conteo: Int): List[Int] = {
  //Comprueba si no esta vacio
  if (tablero.length > 0) {
    //Comprueba si el actual es distinto de 0
    if ((tablero.head > 0)) {
      //Comprueba si el siguiente coincide en valor
      if (obtener(tablero, columnas + 1) == tablero.head) {
        //Si lo hace lo suma poniendo a 0 el nuevo valor y dejando el
actual con el doble ademas de calcular puntos y conteo
        sumarVertical(poner(eliminar(tablero, columnas + 1), tablero.head *
2, 1), columnas, puntuacion + tablero.head * 2, conteo + 1)
        //Si no se sigue recorriendo
      } else tablero.head :: sumarVertical(tablero.tail, columnas,
puntuacion, conteo)
      //Si no se sigue recorriendo
    } else tablero.head :: sumarVertical(tablero.tail, columnas,
puntuacion, conteo)
    //Si es vacio se anade la puntuacion y conteo al final
  } else tablero ::: puntuacion :: conteo :: Nil
}
```

Figura 14: Funciones de suma en direcciones vertical y horizontal

Movimiento automático

El movimiento **automático** que se ha aplicado ha sido un **generador** de **números aleatorios** que permite escoger una tecla aleatoria (**carácter**) y llamar a la dirección elegida en cada caso, más adelante en el punto de optimización se plantea el nuevo sistema.

```
/**
  * Genera un movimiento aleatorio
  * @return el caracter que se ha generado
  */
def movimientoAleatorio(): Char = {
  val random = util.Random;
  (random.nextInt(4) + 1) match {
    case 1 => 'a'
    case 2 => 'w'
    case 3 => 's'
    case 4 => 'd'
  }
}
```

Figura 15: Movimiento del modo automático de forma aleatoria.

Esta función aleatoria será llamada por el método movimiento que según el parámetro modo (un booleano) pedirá datos al usuario o llama a ésta.

3.Funcionamiento del juego

Una vez tenemos todos los movimientos implementados correctamente, pasamos a desarrollar el bucle del juego. En el main del programa se le preguntará al usuario por el modo y la dificultad y con estos dos parámetros más las vidas iniciales (3) y la puntuación (0), se llamará a la función nueva partida. Esta función rellenará el tablero inicial en función de la dificultad y llamará a la función juego con este tablero relleno. En juego, lo primero que comprobaremos es el estado de los booleanos bloqueoEjeX (que será true cuando no puedan realizarse más movimientos en horizontal) y bloqueoEjeY (que será true cuando no puedan realizarse más movimientos en vertical) para ver si el usuario ha perdido una vida o no. Si puede seguir realizando movimientos, se lee una tecla del usuario y se realiza el movimiento en función de la tecla leída con un match en el que contemplaremos los 4 movimientos posibles, si el usuario pulsa "e" para salir o si ha introducido una tecla por teclado que no esté asociada a ningún movimiento, en este caso se imprimirá por pantalla "Dirección imposible". En cada case del match asociado a algún movimiento realizaremos el movimiento especificado y comprobaremos si hay que cambiar el estado de las variables bool que determinan el fin de la partida. Esta comprobación se realizará comparando la puntuación actual y la puntuación obtenida en la anterior jugada y comprobando si el tablero está lleno. Si el tablero está lleno y la puntuación no ha variado de una jugada a otra, se volverá a llamar a juego cambiando el estado de las variables bool. Si en jugadas posteriores el tablero se vacía en alguna posición o se realiza otro movimiento posible también cambiará el estado de los bool a false. En cada movimiento se pasarán las variables puntuación y conteo para que se vaya actualizando su valor.

Este proceso se repetirá hasta que el usuario se quede sin vidas, en este caso finalizará el juego. Se puede ver, a modo de ejemplo, en la figura 16 uno de los case de la función juego.

```
val tecla = movimiento(tablero, columnas, bloqueoEjeX, bloqueoEjeY, modo)
//Segun el movimiento se llama a una direccion u otra
println(tecla)
tecla match {
  case ('w' | 'W') => {
    //Se suman hacia arriba los valores
    val tableroSumado = sumarVertical(moverTodoArriba(tablero, columnas, columnas), columnas, puntuacion, conteo)
    //Se comprueba la condicion de bloqueo del juego(si no hay mas huecos y la puntuacion no ha variado entre rondas)
    if ((huecosLibres(tablero) == 0) && (puntuacion == obtener(tableroSumado, tableroSumado.length - 1))) {
      println("Ya no puedes mover verticalmente")
      //Se vuelve a llamar con el eje ya bloqueado con los mismos parametros
      juego(tablero, columnas, dificultad, casillas, puntuacion, conteo, bloqueoEjeX, true, vidas, modo)
    } else {
      //Si no esta bloqueado llena el tablero y se actualiza el conteo y puntuacion
      juego(
        rellenarTab(moverTodoArriba(quitarHasta(tableroSumado, 2), columnas, columnas), casillas, dificultad),
        columnas, dificultad, casillas, obtener(tableroSumado, tableroSumado.length - 1), obtener(tableroSumado, tableroSumado.length), false, false, vidas, modo)
    }
  }
```

Figura 16: Movimiento hacia arriba en el bucle del juego

4.Optimización

Se ha realizado la optimización para la elección de la jugada óptima a tomar, para ello se realiza un **preprocesado** de las posibles **puntuaciones** y **huecos** para la **próxima jugada**. Cómo la **puntuación** y **huecos** serán **iguales** entre movimientos **horizontales** y **verticales** con solo hacer **una suma** es suficiente, para cada una y poder compararlos. A estos valores obtenidos se les aplica unas **ponderaciones** para obtener el valor que se **comparará** en **función** de **huecos** y **puntuación**. También se comprueba si hay algún **eje bloqueado** por lo

que directamente solo es posible mover en una dirección. No es posible calcular más jugadas de ante mano porque supondría un gran coste al tener que predecir nuevos cálculos con los valores que se hayan insertado aleatoriamente en las futuras combinaciones creando un árbol complejo.

Las comparaciones para dar con los mejores valores se han hecho en base a la puntuación media obtenida en varios tableros, destacando que al incluir movimientos aleatorios para cada eje suponía una gran mejora frente a movimientos fijos, especialmente en tableros grandes.

```
//--------------------------------MOVIMIENTO OPTIMIZADO---------------------
/**
  * Genera un movimiento en funcion a futuras puntuaciones y casillas
vacias
  *
  * @param tablero     tablero que se calculara
  * @param columnas    numero de columnas del tablero
  * @param bloqueoEjeX Indica si se han bloqueado los movimientos
horizontales
  * @param bloqueoEjeY Indica si se han bloqueado los movimientos
verticales
  * @return el mejor movimiento para hacer
  */
def movimientoOptimizado(tablero: List[Int], columnas: Int, bloqueoEjeX:
Boolean, bloqueoEjeY: Boolean): Char = {
  //Si se bloquea un eje solo se puede mover en una direccion
  if (bloqueoEjeX) {
    movAleatorioEjeY()
  }
  //Si se bloquea un eje solo se puede mover en una direccion
  else if (bloqueoEjeY) {
    movAleatorioEjeX()
  }
  //Si no estan bloqueados se aplica la optimizacion
  else {
    //Se calculan las posibles puntuaciones en cada direccion y los huecos
de cada direccion (aprovechamiento de simetria de los movimientos en una
direccion)
    val puntuacionVertical = obtener(sumarVertical(moverTodoArriba(tablero,
columnas, columnas), columnas, 0, 0), tablero.length + 1)
    val puntuacionHorizontal =
obtener(sumarHorizontal(moverTodoIzquierda(tablero, columnas, columnas),
columnas, 0, 0), tablero.length + 1)
    val huecosVertical =
huecosLibres(sumarHorizontal(moverTodoIzquierda(tablero, columnas,
columnas), columnas, 0, 0))
    val huecosHorizontal =
huecosLibres(sumarVertical(moverTodoArriba(tablero, columnas, columnas),
columnas, 0, 0))
    //Valores heuristicos definidos por los huecos disponibles y
puntuaciones de las jugadas
    val heuristicaVertical = huecosVertical * 0.6 + puntuacionVertical *
0.4
    val heuristicaHorizontal = huecosHorizontal * 0.6 +
puntuacionHorizontal * 0.4
    //Si la heuristica vertical es mayor que la horizontal el movimiento
sera horizontal
```

```
    if (heuristicaVertical > heuristicaHorizontal) {
      movAleatorioEjeY()
    }
    else {
      movAleatorioEjeX()
    }
  }
}

/**
  * Genera un movimiento aleatorio en horizontal
  *
  * @return movimiento a realizar
  */
def movAleatorioEjeX(): Char = {
  val random = util.Random;
  (random.nextInt(2) + 1) match {
    case 1 => 'a'
    case 2 => 'd'
  }
}

/**
  * Genera un movimiento aleatorio en vertical
  *
  * @return movimiento a realizar
  */
def movAleatorioEjeY(): Char = {
  val random = util.Random;
  (random.nextInt(2) + 1) match {
    case 1 => 's'
    case 2 => 'w'
  }
}
```

Figura 17 función de movimiento optimizado y movimientos aleatorios de los ejes