



# Informe Proyecto Final: Programación Concurrente

Jeidy Nicol Murillo Murillo - 235910  
Verónica Lorena Mujica Gavidia - 2359406  
Karol Tatiana Burbano Nasner - 2359305  
Sebastian Castro Rengifo - 2359435

17 de diciembre de 2024

## 1. Informe de Procesos

En esta parte del informe se presentan los procesos generados por los programas recursivos, se generan algunos ejemplos para cada ejercicio, y se muestra cómo se comporta el proceso generado por cada uno de los programas.

### 1.1. Funciones para calcular los costos relacionados con el riego

#### 1.1.1. Función para calcular el costo de riego por tablón (costoRiegoTablon)

La función costoRiegoTablon tiene como objetivo calcular el costo total de regar un tablón específico en una finca agrícola, teniendo en cuenta dos factores principales:

- **Tiempo de riego:** El tiempo que se tarda en regar el tablón, que es un valor previamente asignado en la finca.
- **Tiempo de supervivencia:** El tiempo máximo que un tablón puede sobrevivir sin riego antes de que su condición empeore. Si el tiempo de riego final excede este límite, se aplica una penalización que depende de la prioridad del tablón. La función se utiliza dentro de un proceso más grande que optimiza la programación de riego para toda una finca. Para cada tablón, el costo se calcula teniendo en cuenta:

Si el tablón es regado dentro de su tiempo de supervivencia. Si no se riega a tiempo, se multiplica la prioridad del tablón por la diferencia entre el tiempo final de riego y el tiempo de supervivencia, penalizando el retraso.

Para verificar el correcto funcionamiento de esta función, se utiliza el siguiente test de una finca de dos tablones.

```
1 test("Costo para un finca con dos tablones y programación en orden") {
2   val finca = Vector(
3     (20, 8, 2), // Tablón 1
4     (6, 2, 3),  // Tablón 2
5   )
6   val programacion = Vector(0, 1) // Orden: Tablón 3 -> 2 -> 1
7
8
9   assert(riegoOptimo.costoRiegoTablon(0, finca, programacion) == 12)
10  assert(riegoOptimo.costoRiegoTablon(1, finca, programacion) == 12)
11 }
```

El programa empieza definiendo las variables  $i$ ,  $f$ ,  $pi$ . Siendo  $i = 0$ ,  $f$  los vectores de la finca (tablones) y  $pi$  los valores de programación de riego que son el orden de los tablones (0, 1), es decir, primero se riega el Tablón 1 y luego el Tablón 2.

```

Local
  i = 0
  > f = Vector1@1977 "Vector((20,8,2), (6,2,3))"
  > pi = Vector1@1985 "Vector(0, 1)"
  > this = RiegoOptimo@1967

```

Figura 1: Debug CostoRiegoTablonTest

El *tiempoinicio* = 0 siempre empieza en 0 y el *tiemporiego* = 8. Para calcular el *tiempofinal* se deben sumar *tiempoinicio* + *tiemporiego*. En este caso sería  $0 + 8 = 8$  y como se cumple la condición de que *tiemposupervivencia* sea mayor que *tiempoinicio*  $tsup \geq tiempoinicio$ . El costo se calcula restando *tsup* - *tiempofinal*.

```

Local
  i = 0
  > f = Vector1@1977 "Vector((20,8,2), (6,2,3))"
  > pi = Vector1@1980 "Vector(0, 1)"
  tiempoInicio = 0
  tiempoFinal = 8
  > this = RiegoOptimo@1967

```

Figura 2: Debug CostoRiegoTablonTest

En este caso, el costo se calcularía:  $20 - 8 = 12$ . Dando como resultado que el costo para el tablón (20, 8, 2) es 12.

```

Local
  ->costoRiegoTablon() = 12
  > $this = CostoRiegoTablonTest@1960 "CostoRiegoTablonTest"
  > finca = Vector1@1977 "Vector((20,8,2), (6,2,3))"
  > programacion = Vector1@1980 "Vector(0, 1)"

```

Figura 3: Debug CostoRiegoTablonTest

Ahora se debe calcular el costo para el siguiente tablón, pero esta vez el tiempo inicial ya no es 0, sino que se va a ir acumulando conforme pasen las programaciones de riego. En el tablón anterior el tiempo final fue 8, así que en este tablón el tiempo inicio se convierte en 8.

Para el tiempo final se suma  $8 + 2 = 10$  lo que da como tiempo final **10**. En este caso, el tiempo inicial es mayor que el tiempo de supervivencia, así que no se cumple la condición  $tsup \geq tiempoinicio$ , por lo que se multiplica la *prioridad*  $\times (tiempoFinal - tsup)$ .

```

✓ Local
  i = 1
  > f = Vector1@1977 "Vector((20,8,2), (6,2,3))"
  > pi = Vector1@1980 "Vector(0, 1)"
  tiempoInicio = 8
  tiempoFinal = 10
  > this = RiegoOptimo@1967

```

Figura 4: Debug CostoRiegoTablonTest

En este caso seria  $3 \times (10 - 6) = 3 \times 4 = 12$ . Es decir el costo del tablón **(6,2,3)** es **12**

```

✓ Local
  →costoRiegoTablon() = 12
  > $this = CostoRiegoTablonTest@1960 "CostoRiegoTablonTest"
  > finca = Vector1@1977 "Vector((20,8,2), (6,2,3))"
  > programacion = Vector1@1980 "Vector(0, 1)"

```

Figura 5: Debug CostoRiegoTablonTest

Por último se puede ver como el test se completa exitosamente.

```

✓ Local
  > →$anonfun$new$1() = Succeeded$@1969 "Succeeded"
  > f = CostoRiegoTablonTest$$Lambda$89/0x0000019c2a16
  > this = OutcomeOf$@2026

```

Figura 6: Debug CostoRiegoTablonTest

### 1.1.2. Función para calcular el costo de riego Finca (costoRiegoFinca)

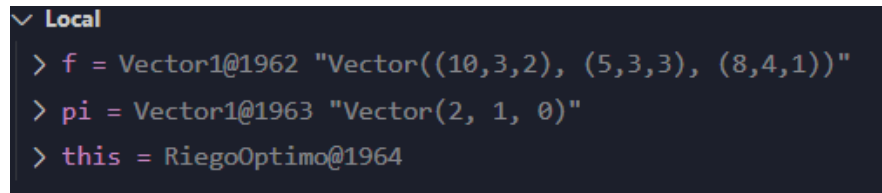
La función `costoRiegoFinca` calcula el costo total de regar una finca, dado un conjunto de tablonces y una programación de riego. Para hacer esto, la función:

- Itera sobre todos los tablonces en la finca (usando los índices de la programación `pi`).
- Calcula el costo de regar cada tablón utilizando la función `costoRiegoTablon`, que ya hemos analizado en la respuesta anterior.
- Suma los costos individuales de cada tablón para obtener el costo total de regar la finca.

Se utilizará el siguiente test para probar la correcta ejecución de esta función:

```
1 test("Costo para una finca con tres tablonos y programaci n de regado inversa") {
2   val finca = Vector(
3     (10, 3, 2), // Tabl n 1
4     (5, 3, 3),  // Tabl n 2
5     (8, 4, 1)   // Tabl n 3
6   )
7   val programacion = Vector(2, 1, 0) // Orden: Tabl n 3 -> 2 -> 1
8
9   assert(riegoOptimo.costoRiegoFinca(finca, programacion) == 10) // Suma de costos
10 }
```

Inicia el programa definiendo la finca con sus tres tablonos y la programación de riego. En este caso, vemos que la programación es inversa [2,1,0], es decir, comienza regando el Tablón 3, seguido por el Tablón 2 y finalmente el Tablón 1. El costo total esperado en el test es 10.

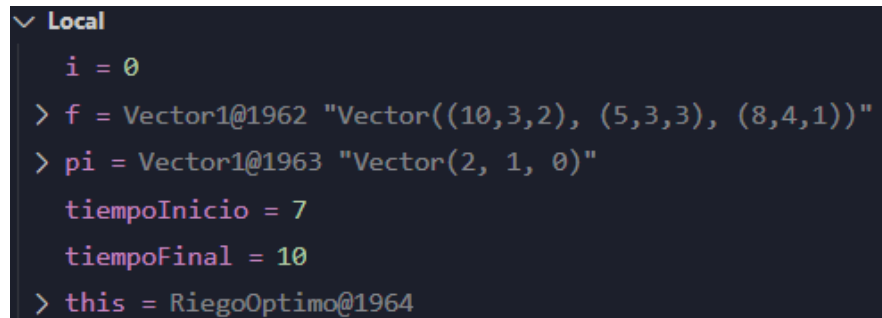


```
Local
> f = Vector1@1962 "Vector((10,3,2), (5,3,3), (8,4,1))"
> pi = Vector1@1963 "Vector(2, 1, 0)"
> this = RiegoOptimo@1964
```

Figura 7: Debug CostoRiegoFinca

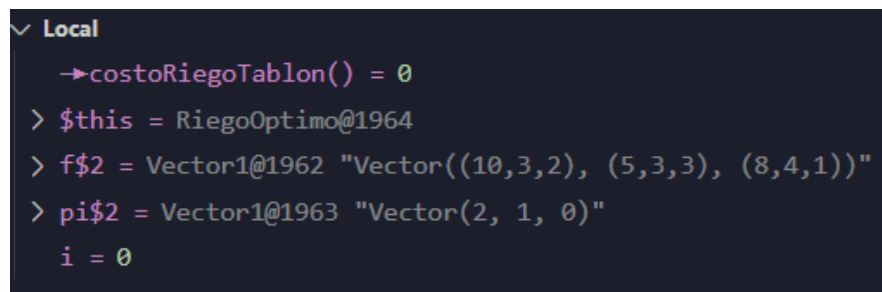
Así que para calcular el tiempo de inicio se suman todos los tiempos de riego, menos del que se va averiguar el costo en este caso el tercero  $3 + 4 = 7$ . Es así que el *tiempoInicio* = 7. El resto del procedimiento es el mismo que para calcular *costoRiegoTablon*.

- Se suma tiempo inicial con tiempo de riego  $7 + 3 = 10$ . Este tiempo es utilizado para determinar si el riego ocurre dentro del tiempo de supervivencia.
- Se compara si el tiempo de supervivencia es mayor o igual al tiempo final y dependiendo del caso se restan o se multiplican. Aquí  $10 = 10$  entonces  $10 - 10 = 0$



```
Local
i = 0
> f = Vector1@1962 "Vector((10,3,2), (5,3,3), (8,4,1))"
> pi = Vector1@1963 "Vector(2, 1, 0)"
tiempoInicio = 7
tiempoFinal = 10
> this = RiegoOptimo@1964
```

Figura 8: Debug CostoRiegoFinca



```
Local
-> costoRiegoTablon() = 0
> $this = RiegoOptimo@1964
> f$2 = Vector1@1962 "Vector((10,3,2), (5,3,3), (8,4,1))"
> pi$2 = Vector1@1963 "Vector(2, 1, 0)"
i = 0
```

Figura 9: Debug CostoRiegoFinca

```
Local
  →$anonfun$costoRiegoFinca$1() = 0
  > t = Integer@2000 "0"
  > this = RiegoOptimo$$Lambda$156/0x000000
```

Figura 10: Debug CostoRiegoFinca

Estas imágenes detallan los pasos intermedios donde se calcula el costo de riego para los demás tableros en la finca. Para el Tablón 2 el tiempo de inicio se obtiene sumando el tiempo de riego acumulado del Tablón 3 así que es: (4). El tiempo final  $4 + 3 = 7$  y  $5 \leq 7$  entonces  $3 \times (7 - 5) = 3 \times 2 = 6$

```
Local
  > $this = RiegoOptimo@1964
  > f$2 = Vector1@1962 "Vector((10,3,2), (5,3,3), (8,4,1))"
  > pi$2 = Vector1@1963 "Vector(2, 1, 0)"
  i = 1
```

Figura 11: Debug CostoRiegoFinca

```
Local
  i = 1
  > f = Vector1@1962 "Vector((10,3,2), (5,3,3), (8,4,1))"
  > pi = Vector1@1963 "Vector(2, 1, 0)"
  tiempoInicio = 4
  tiempoFinal = 7
  > this = RiegoOptimo@1964
```

Figura 12: Debug CostoRiegoFinca

```
Local
  →costoRiegoTablon() = 6
  > $this = RiegoOptimo@1964
  > f$2 = Vector1@1962 "Vector((10,3,2), (5,3,3), (8,4,1))"
  > pi$2 = Vector1@1963 "Vector(2, 1, 0)"
  i = 1
```

Figura 13: Debug CostoRiegoFinca

```

✓ Local
  →$anonfun$costoRiegoFinca$1() = 6
  > t = Integer@2027 "1"
  > this = RiegoOptimo$$Lambda$156/0x0000022e2913f8c0@2001 ...

```

Figura 14: Debug CostoRiegoFinca

```

✓ Local
  > →apply() = Integer@2030 "6"
  > f = RiegoOptimo$$Lambda$156/0x0000022e2913f8c0@2001 "tall..."
  > strictOptimizedMap_b = VectorBuilder@2006 "VectorBuilder(...)"
  > strictOptimizedMap_it = RangeIterator@2007 "<iterator>"
  > this = Range$Exclusive@2008 "Range 0 until 3"

```

Figura 15: Debug CostoRiegoFinca

Se repite el mismo proceso para el último tablón hasta conseguir su costo de riego.

```

✓ Local
  i = 2
  > f = Vector1@1962 "Vector((10,3,2), (5,3,3), (8,4,1))"
  > pi = Vector1@1963 "Vector(2, 1, 0)"
  > this = RiegoOptimo@1964

```

Figura 16: Debug CostoRiegoFinca

```

✓ Local
  i = 2
  > f = Vector1@1962 "Vector((10,3,2), (5,3,3), (8,4,1))"
  > pi = Vector1@1963 "Vector(2, 1, 0)"
  tiempoInicio = 0
  tiempoFinal = 4
  > this = RiegoOptimo@1964

```

Figura 17: Debug CostoRiegoFinca

```

✓ Local
  →costoRiegoTablon() = 4
  > $this = RiegoOptimo@1964
  > f$2 = Vector1@1962 "Vector((10,3,2), (5,3,3), (8,4,1))"
  > pi$2 = Vector1@1963 "Vector(2, 1, 0)"
  i = 2

```

Figura 18: Debug CostoRiegoFinca

```

✓ Local
  →$anonfun$costoRiegoFinca$1() = 4
  > t = Integer@2048 "2"
  > this = RiegoOptimo$$Lambda$156/0x0000022e2913f8c0@2001 "t..."

```

Figura 19: Debug CostoRiegoFinca

```

✓ Local
  > →apply() = Integer@2051 "4"
  > f = RiegoOptimo$$Lambda$156/0x0000022e2913f8c0@2001 "tall..."
  > strictOptimizedMap_b = VectorBuilder@2006 "VectorBuilder(...)"
  > strictOptimizedMap_it = RangeIterator@2007 "<iterator>"
  > this = Range$Exclusive@2008 "Range 0 until 3"

```

Figura 20: Debug CostoRiegoFinca

Los resultados de los costos de cada tablón se almacenan en un Vector para sumarlos.

```

✓ Local
  > →map() = Vector1@2058 "Vector(0, 6, 4)"
  > f = Vector1@1962 "Vector((10,3,2), (5,3,3), (8,4,1))"
  > pi = Vector1@1963 "Vector(2, 1, 0)"
  > this = RiegoOptimo@1964

```

Figura 21: Debug CostoRiegoFinca

El resultado de la suma es 10 lo que significa que la función es correcta, el costo de riego para la finca de tres tablonos definida es 10.

```

✓ Local
  →costoRiegoFinca() = 10
  > $this = CostoRiegoFincaTest@2067 "CostoRiegoFincaTest"
  > finca = Vector1@1962 "Vector((10,3,2), (5,3,3), (8,4,1))"
  > programacion = Vector1@1963 "Vector(2, 1, 0)"

```

Figura 22: Debug CostoRiegoFinca

```

✓ Local
  > →$anonfun$new$1() = Succeeded$@2076 "Succeeded"
  > f = CostoRiegoFincaTest$$Lambda$89/0x0000022e29102f00@207...
  > this = OutcomeOf$@2078

```

Figura 23: Debug CostoRiegoFinca

### 1.1.3. Función para calcular el costo de movilidad (costoRiegoMovilidad)

La función recorre los índices del programa de riego, calculando las distancias entre parcelas consecutivas y sumándolas. El siguiente test verifica el funcionamiento correcto de la función `costoMovilidad`, que calcula el costo total de movilidad para un programa de riego basado en una matriz de distancias.

```

test("CostoMovilidad con finca grande y distancia uniforme") {
  val finca: Finca = Vector((10, 4, 2), (20, 6, 3), (15, 5, 2), (30, 8, 4))
  val distancia: Distancia = Vector(
    Vector(0, 10, 15, 20),
    Vector(10, 0, 25, 30),
    Vector(15, 25, 0, 35),
    Vector(20, 30, 35, 0)
  )
  val pi: ProgRiego = Vector(1, 2, 1, 3)

  val costo = riegoOptimo.costoMovilidad(finca, pi, distancia)
  val costoEsperado = distancia(1)(2) + distancia(2)(1) + distancia(1)(3)
  assert(costo == costoEsperado, s"El costo esperado es 80, y se obtuvo $costo")
}

```

Figura 24: CostoMovilidadTest

Se inicia la depuración desde `costoMovilidadTest` como se puede ver a continuación:

```

> ⓘ $this = {costoMovilidadTests@2373} costoMovilidadTests
> ⓘ riegoOptimo = {RiegoOptimo@2374} taller.RiegoOptimo@471a9022

```

Figura 25: CostoMovilidadTest

- **Finca (finca):** Aunque no se usa en la función, representa las características de las parcelas.

$$\text{finca} = \{(10, 4, 2), (20, 6, 3), (15, 5, 2), (30, 8, 4)\}$$



```

> (P) $this = {costoMovilidadTests@2373} costoMovilidadTests
> finca = {Vector1@2740} size = 4
> 0 = {Tuple3@2743} (10,4,2)
> 1 = {Tuple3@2744} (20,6,3)
> 2 = {Tuple3@2745} (15,5,2)
> 3 = {Tuple3@2746} (30,8,4)
> riegoOptimo = {RiegoOptimo@2374} taller.RiegoOptimo@471a9022

```

Figura 26: CostoMovilidadTest

- **Matriz de Distancias (distancia):** Define la distancia entre cada par de parcelas:

$$\text{distancia} = \begin{bmatrix} 0 & 10 & 15 & 20 \\ 10 & 0 & 25 & 30 \\ 15 & 25 & 0 & 35 \\ 20 & 30 & 35 & 0 \end{bmatrix}$$

```

> (P) $this = {costoMovilidadTests@2373} costoMovilidadTests
> finca = {Vector1@2740} size = 4
> distancia = {Vector1@2764} size = 4
> 0 = {Vector1@2767} size = 4
> 1 = {Vector1@2768} size = 4
> 2 = {Vector1@2769} size = 4
> 3 = {Vector1@2770} size = 4
> riegoOptimo = {RiegoOptimo@2374} taller.RiegoOptimo@471a9022

```

Figura 27: CostoMovilidadTest

- **Programa de Riego (pi):** Orden en que se visitan las parcelas:

$$\text{pi} = [1, 2, 1, 3]$$

```

> (P) $this = {costoMovilidadTests@2373} costoMovilidadTests
> finca = {Vector1@2740} size = 4
> distancia = {Vector1@2764} size = 4
> pi = {Vector1@2775} size = 4
> 0 = {Integer@2778} 1
> 1 = {Integer@2779} 2
> 2 = {Integer@2778} 1
> 3 = {Integer@2780} 3
> riegoOptimo = {RiegoOptimo@2374} taller.RiegoOptimo@471a9022

```

Figura 28: CostoMovilidadTest

Observamos las mismas variables anteriores pero esta vez se inicializan desde RiegoOptimo, es decir la función principal costoMovilidad, como f, pi y d respectivamente. Con tamaños = 4.

El rango de índices es:

$$0 \leq j < (\text{pi.length} - 1) \implies j \in \{0, 1, 2\}$$

```
def costoMovilidad(f:Finca, pi:ProgRiego, d:Distancia) : Int = {
  (0 ≤ until < pi.length - 1).map(j => d(pi(j))(pi(j+1))).sum
}
```

Figura 29: CostoMovilidadTest

```
> this = {RiegoOptimo@2374} taller.RiegoOptimo@471a9022
> f = {Vector1@2740} size = 4
> pi = {Vector1@2775} size = 4
> d = {Vector1@2764} size = 4
```

Figura 30: CostoMovilidadTest

Para cada valor de  $j$ , se calcula la distancia entre las parcelas  $\text{pi}(j)$  y  $\text{pi}(j+1)$ :

- $j = 0$ :  $\text{pi}(0) = 1, \text{pi}(1) = 2$   
 $d(\text{pi}(0), \text{pi}(1)) = d(1, 2) = 25$
- $j = 1$ :  $\text{pi}(1) = 2, \text{pi}(2) = 1$   
 $d(\text{pi}(1), \text{pi}(2)) = d(2, 1) = 25$
- $j = 2$ :  $\text{pi}(2) = 1, \text{pi}(3) = 3$   
 $d(\text{pi}(2), \text{pi}(3)) = d(1, 3) = 30$

El vector de distancias es:

[25, 25, 30]

La suma de las distancias es:

$$\text{costo} = 25 + 25 + 30 = 80$$

```
val costo = riegoOptimo.costoMovilidad(finca, pi, distancia) finca: size = 4 pi: size = 4 costo: 80
```

Figura 31: CostoMovilidadTest

```
> $this = {costoMovilidadTests@2373} costoMovilidadTests
> finca = {Vector1@2740} size = 4
> distancia = {Vector1@2764} size = 4
> pi = {Vector1@2775} size = 4
> costo = 80
> riegoOptimo = {RiegoOptimo@2374} taller.RiegoOptimo@471a9022
```

Figura 32: CostoMovilidadTest

El costo esperado se calcula manualmente:

$$\text{costoEsperado} = d(1,2) + d(2,1) + d(1,3) = 25 + 25 + 30 = 80$$

```
val costo = riegoOptimo.costoMovilidad(finca, pi, distancia)  finca: size = 4    pi: size = 4    costo: 80
val costoEsperado = distancia(1)(2) + distancia(2)(1) + distancia(1)(3)  distancia: size = 4
```

Figura 33: CostoMovilidadTest

```
10 costo = 80
01
10 costoEsperado = 80
01
> riegoOptimo = {RiegoOptimo@2374} taller.RiegoOptimo@471a9022
```

Figura 34: CostoMovilidadTest

El `assert` verifica la igualdad:

```
assert(80 == 80)
```

```
assert(costo == costoEsperado, s"El costo esperado es 80, y se obtuvo $costo")  costoEsperado: 80    costo: 80
```

Figura 35: CostoMovilidadTest

Como conclusión tenemos que el test verifica que la función `costoMovilidad` calcule correctamente el costo total de movilidad según la matriz de distancias y el programa de riego.

#### 1.1.4. Función para generar programaciones de riego (`generarProgramacionesRiego`)

Este test verifica el correcto funcionamiento de la función `generarProgramacionesRiego`, que genera todas las posibles programaciones de riego para una finca con 2 tablones:

```
test("Generar programaciones para finca con dos tablones") {
  val finca = Vector((3, 2, 1), (4, 1, 3))
  val programaciones = riegoOptimo.generarProgramacionesRiego(finca)
  assert(programaciones == Vector(Vector(0, 1), Vector(1, 0)))
}
```

Figura 36: Enter Caption

La función `generarProgramacionesRiego` utiliza combinaciones y permutaciones para calcular todas las programaciones posibles de riego para una finca dada.

```
1 def generarProgramacionesRiego(f: Finca): Vector[ProgRiego] = {
2   // Dada una finca de n tablones, devuelve todas las posibles programaciones de
   riego de la finca
3   val indices = (0 until f.length).toVector
4   indices.permutations.toVector
5 }
```

##### 1. Entradas al Test

- **Finca (finca):** Especifica dos tablonos con los siguientes valores:

$$\text{finca} = \begin{bmatrix} (3, 2, 1) \\ (4, 1, 3) \end{bmatrix}$$

Cada tupla  $(a, b, c)$  representa:

- $a$ : Área del tablón.
- $b$ : Demanda de agua del tablón.
- $c$ : Distancia relativa del tablón al inicio del sistema de riego.

Cálculo del Vector de Índices:

Se calcula un vector que contiene los índices de los tablonos en la finca:

$$\text{indices} = [0, 1]$$

Aquí, 0 corresponde al primer tablón y 1 al segundo.

```
> this = {RiegoOptimo@2471} taller.RiegoOptimo@27d5a580
✓ (P) f = {Vector1@2738} size = 1
> 0 = {Tuple3@2742} (3,1,2)
```

Figura 37: Enter Caption

La función utiliza el método `permutations` para calcular todas las posibles maneras de ordenar los índices de los tablonos:

$$\text{indices.permutations} = [[0, 1], [1, 0]]$$

Estas permutaciones representan todas las secuencias posibles de riego:

- $[0, 1]$ : Riego del tablón 0 seguido del tablón 1.
- $[1, 0]$ : Riego del tablón 1 seguido del tablón 0.

```
// Dada una finca de n tablonos, devuelve todas las posibles programaciones de riego de la finca
val indices = (0 until f.length).toVector f: size = 1 indices: size = 1
indices.permutations.toVector indices: size = 1
```

Figura 38: Enter Caption

```
> this = {RiegoOptimo@2471} taller.RiegoOptimo@27d5a580
> (P) f = {Vector1@2738} size = 1
✓ this indices = {Vector1@2747} size = 1
> 0 = {Integer@2751} 0
```

Figura 39: Enter Caption

```
val finca = Vector((3, 2, 1), (4, 1, 3)) finca: size = 2
val programaciones = riegoOptimo.generarProgramacionesRiego(finca) finca: size = 2
```

Figura 40: Enter Caption

```

> @ $this = {generarProgramacionesRiegoTest@2470} generarProgramacionesRiegoTest
✓ finca = {Vector1@2794} size = 2
> 0 = {Tuple3@2797} (3,2,1)
> 1 = {Tuple3@2798} (4,1,3)

```

Figura 41: Enter Caption

```

// Dada una finca de n tablonas, devuelve todas las posibles programaciones de riego de la finca
val indices = (0 ≤ until < f.length).toVector f: size = 2 indices: size = 2
indices.permutations.toVector indices: size = 2

```

Figura 42: Enter Caption

```

> this = {RiegoOptimo@2471} taller.RiegoOptimo@27d5a580
✓ @ f = {Vector1@2794} size = 2
> 0 = {Tuple3@2797} (3,2,1)
> 1 = {Tuple3@2798} (4,1,3)
✓ indices = {Vector1@2805} size = 2
> 0 = {Integer@2751} 0
> 1 = {Integer@2810} 1

```

Figura 43: Enter Caption

```

> @ $this = {generarProgramacionesRiegoTest@2470} generarProgramacionesRiegoTest
✓ finca = {Vector1@2794} size = 2
> 0 = {Tuple3@2797} (3,2,1)
> 1 = {Tuple3@2798} (4,1,3)
✓ programaciones = {Vector1@2811} size = 2
> 0 = {Vector1@2816} size = 2
> 1 = {Vector1@2817} size = 2

```

Figura 44: Enter Caption

```

// Dada una finca de n tablonas, devuelve todas las posibles programaciones de riego de la finca
val indices = (0 ≤ until < f.length).toVector f: size = 3 indices: size = 3
indices.permutations.toVector indices: size = 3

```

Figura 45: Enter Caption

El resultado de las permutaciones se convierte a un **Vector** para cumplir con el tipo de dato requerido:

```
programaciones = Vector(Vector(0, 1), Vector(1, 0))
```

```

✓ ⓘ f = {Vector1@2818} size = 3
  > ⓘ 0 = {Tuple3@2821} (10,3,1)
  > ⓘ 1 = {Tuple3@2822} (5,2,2)
  > ⓘ 2 = {Tuple3@2823} (8,1,3)
✓ ⓘ indices = {Vector1@2827} size = 3
  > ⓘ 0 = {Integer@2751} 0
  > ⓘ 1 = {Integer@2810} 1
  > ⓘ 2 = {Integer@2833} 2

```

Figura 46: Enter Caption

El test compara el resultado de la función con el valor esperado:

```
programacionesEsperadas = Vector(Vector(0, 1), Vector(1, 0))
```

Se realiza la verificación mediante el `assert`:

```
assert(programaciones == programacionesEsperadas)
```

```

✓ ⓘ f = {Vector1@2836} size = 4
  > ⓘ 0 = {Tuple3@2839} (1,1,1)
  > ⓘ 1 = {Tuple3@2840} (2,2,2)
  > ⓘ 2 = {Tuple3@2841} (3,3,3)
  > ⓘ 3 = {Tuple3@2842} (4,4,4)
✓ ⓘ indices = {Vector1@2847} size = 4
  > ⓘ 0 = {Integer@2751} 0
  > ⓘ 1 = {Integer@2810} 1
  > ⓘ 2 = {Integer@2833} 2
  > ⓘ 3 = {Integer@2855} 3

```

Figura 47: Enter Caption



```
// Dada una finca de n tablones, devuelve todas las posibles programaciones de riego de la finca
val indices = (0 ≤ until < f.length).toVector f: size = 0 indices: size = 0
indices.permutations.toVector indices: size = 0
```

Figura 48: Enter Caption

Como ambos valores coinciden, el test pasa exitosamente.

Conclusión:

La función `generarProgramacionesRiego` funciona correctamente al generar todas las posibles secuencias de riego para una finca. En este caso, se valida que para dos tablones existan únicamente dos programaciones posibles, ordenadas de manera exhaustiva.

### 1.1.5. Función para calcular programaciones de riego óptima (`ProgramacionRiegoOptimo`)

La función `ProgramacionRiegoOptimo` recibe una finca (`f`) y una matriz de distancias (`d`) como entradas, y devuelve la programación de riego óptima junto con su costo total.

```
1 def ProgramacionRiegoOptimo(f:Finca, d:Distancia) : (ProgRiego, Int) = {
2   // Dada una finca devuelve la programaci n de riego ptima
3   val programaciones = generarProgramacionesRiego(f)
4   val costos = programaciones.map(pi =>
5     (pi, costoRiegoFinca(f, pi) + costoMovilidad(f, pi, d))
6   )
7   costos.minBy(_._2)
8 }
```

La función realiza las siguientes operaciones:

1. **Generación de programaciones de riego:** Utiliza la función `generarProgramacionesRiego(f)` para obtener todas las permutaciones posibles de riego de los tablones en la finca.
2. **Cálculo de costos:** Para cada programación (`pi`), calcula el costo total sumando el costo de riego y el costo de movilidad entre los tablones utilizando las funciones `costoRiegoFinca` y `costoMovilidad`.
3. **Selección de la programación óptima:** Utiliza `minBy` para seleccionar la programación con el costo total mínimo.

El resultado final es la programación óptima y su costo asociado.

```
✓ Local
> f = Vector1@1992 "Vector((5,4,2), (5,2,4), (8,3,2), (6,2,1), (6,2,3))"
> d = Vector1@1993 "Vector(Vector(0, 3, 5, 7, 9), Vector(3, 0, 2, 4, 6), Vector(5, 2, 0, 3, 8), Vector(7, 4, 3, 0, 2), Vector(9, 6, 8, 2, 0))"
> this = RiegoOptimo@1994
```

Figura 49: Debug ProgramaciónRiegoOptimo

En esta etapa, se generan los datos iniciales:

- **Finca:** Vector de tablones con sus atributos (*tiempo de supervivencia, tiempo de riego, prioridad*).
- **Matriz de distancias:** Distancias no uniformes entre cada par de tablones.

Estos datos son la entrada para las siguientes etapas del algoritmo.

```
> x$1 = Tuple2@2500 "(Vector(0, 1, 2, 3, 4),43)"
```

Figura 50: Debug ProgramaciónRiegoOptimo

```

Local
> f = Vector1@1992 "Vector((5,4,2), (5,2,4), (8,3,2), (6,2,1), (6,2,3))"
> d = Vector1@1993 "Vector(Vector(0, 3, 5, 7, 9), Vector(3, 0, 2, 4, 6), Vector(5, 2, 0, 3, 8), Vector(7, 4, 3, 0, 2), Vector(9, 6, 8, 2, 0))"
> programaciones = Vector2@2020 "Vector(Vector(0, 1, 2, 3, 4), Vector(0, 1, 2, 4, 3), Vector(0, 1, 3, 2, 4), Vector(0, 1, 3, 4, 2), Vector(0, 1, 4, 2, 3), Vector(0, 1, 4, 3, 2))"
> costos = Vector2@2489 "Vector((Vector(0, 1, 2, 3, 4),43), (Vector(0, 1, 2, 4, 3),44), (Vector(0, 1, 3, 2, 4),52), (Vector(0, 1, 3, 4, 2),46), (Vector(0, 1, 4, 2, 3),47), (Vector(0, 1, 4, 3, 2),48))"
> this = RiegoOptimo@1994

```

Figura 51: Debug ProgramaciónRiegoOptimo

```

Local
> $this = RiegoOptimo@1994
> f$3 = Vector1@1992 "Vector((5,4,2), (5,2,4), (8,3,2), (6,2,1), (6,2,3))"
> d$2 = Vector1@1993 "Vector(Vector(0, 3, 5, 7, 9), Vector(3, 0, 2, 4, 6), Vector(5, 2, 0, 3, 8), Vector(7, 4, 3, 0, 2), Vector(9, 6, 8, 2, 0))"
> pi = Vector1@2146 "Vector(1, 0, 2, 4, 3)"

```

Figura 52: Debug ProgramaciónRiegoOptimo

```

Local
> $this = RiegoOptimo@1994
> f$3 = Vector1@1992 "Vector((5,4,2), (5,2,4), (8,3,2), (6,2,1), (6,2,3))"
> d$2 = Vector1@1993 "Vector(Vector(0, 3, 5, 7, 9), Vector(3, 0, 2, 4, 6), Vector(5, 2, 0, 3, 8), Vector(7, 4, 3, 0, 2), Vector(9, 6, 8, 2, 0))"
> pi = Vector1@2142 "Vector(1, 0, 2, 3, 4)"

```

Figura 53: Debug ProgramaciónRiegoOptimo

```

Local
> $this = RiegoOptimo@1994
> f$3 = Vector1@1992 "Vector((5,4,2), (5,2,4), (8,3,2), (6,2,1), (6,2,3))"
> d$2 = Vector1@1993 "Vector(Vector(0, 3, 5, 7, 9), Vector(3, 0, 2, 4, 6), Vector(5, 2, 0, 3, 8), Vector(7, 4, 3, 0, 2), Vector(9, 6, 8, 2, 0))"
> pi = Vector1@2138 "Vector(0, 4, 3, 2, 1)"

```

Figura 54: Debug ProgramaciónRiegoOptimo

```

Local
> $this = RiegoOptimo@1994
> f$3 = Vector1@1992 "Vector((5,4,2), (5,2,4), (8,3,2), (6,2,1), (6,2,3))"
> d$2 = Vector1@1993 "Vector(Vector(0, 3, 5, 7, 9), Vector(3, 0, 2, 4, 6), Vector(5, 2, 0, 3, 8), Vector(7, 4, 3, 0, 2), Vector(9, 6, 8, 2, 0))"
> pi = Vector1@2134 "Vector(0, 4, 3, 1, 2)"

```

Figura 55: Debug ProgramaciónRiegoOptimo

Aquí se generan todas las permutaciones posibles para la programación de riego. Cada programación especifica el orden en el que se riegan los tablonés. Este conjunto de permutaciones es exhaustivo y se utilizará para calcular los costos.

```

Local
> $this = RiegoOptimo@1994
> f$3 = Vector1@1992 "Vector((5,4,2), (5,2,4), (8,3,2), (6,2,1), (6,2,3))"
> d$2 = Vector1@1993 "Vector(Vector(0, 3, 5, 7, 9), Vector(3, 0, 2, 4, 6), Vector(5, 2, 0, 3, 8), Vector(7, 4, 3, 0, 2), Vector(9, 6, 8, 2, 0))"
> pi = Vector1@2126 "Vector(0, 4, 2, 1, 3)"

```

Figura 56: Debug ProgramaciónRiegoOptimo



```

Local
> $this = RiegoOptimo@1994
> f$3 = Vector1@1992 "Vector((5,4,2), (5,2,4), (8,3,2), (6,2,1), (6,2,3))"
> d$2 = Vector1@1993 "Vector(Vector(0, 3, 5, 7, 9), Vector(3, 0, 2, 4, 6), Vector(5, 2, 0, 3, 8), Vector(7, 4, 3, 0, 2), Vector(9, 6, 8, 2, 0))"
> pi = Vector1@2122 "Vector(0, 4, 1, 3, 2)"

```

Figura 57: Debug ProgramaciónRiegoOptimo

```

Local
> $this = RiegoOptimo@1994
> f$3 = Vector1@1992 "Vector((5,4,2), (5,2,4), (8,3,2), (6,2,1), (6,2,3))"
> d$2 = Vector1@1993 "Vector(Vector(0, 3, 5, 7, 9), Vector(3, 0, 2, 4, 6), Vector(5, 2, 0, 3, 8), Vector(7, 4, 3, 0, 2), Vector(9, 6, 8, 2, 0))"
> pi = Vector1@2118 "Vector(0, 4, 1, 2, 3)"

```

Figura 58: Debug ProgramaciónRiegoOptimo

```

Local
> $this = RiegoOptimo@1994
> f$3 = Vector1@1992 "Vector((5,4,2), (5,2,4), (8,3,2), (6,2,1), (6,2,3))"
> d$2 = Vector1@1993 "Vector(Vector(0, 3, 5, 7, 9), Vector(3, 0, 2, 4, 6), Vector(5, 2, 0, 3, 8), Vector(7, 4, 3, 0, 2), Vector(9, 6, 8, 2, 0))"
> pi = Vector1@2114 "Vector(0, 3, 4, 2, 1)"

```

Figura 59: Debug ProgramaciónRiegoOptimo

```

Local
> $this = RiegoOptimo@1994
> f$3 = Vector1@1992 "Vector((5,4,2), (5,2,4), (8,3,2), (6,2,1), (6,2,3))"
> d$2 = Vector1@1993 "Vector(Vector(0, 3, 5, 7, 9), Vector(3, 0, 2, 4, 6), Vector(5, 2, 0, 3, 8), Vector(7, 4, 3, 0, 2), Vector(9, 6, 8, 2, 0))"
> pi = Vector1@2110 "Vector(0, 3, 4, 1, 2)"

```

Figura 60: Debug ProgramaciónRiegoOptimo

Dado un vector de programación, se calcula el tiempo de inicio de riego para cada tablón utilizando:

$$t(i) = t(i - 1) + \text{tiempo de riego del tablón anterior.} \quad (1)$$

```

Local
> $this = RiegoOptimo@1994
> f$3 = Vector1@1992 "Vector((5,4,2), (5,2,4), (8,3,2), (6,2,1), (6,2,3))"
> d$2 = Vector1@1993 "Vector(Vector(0, 3, 5, 7, 9), Vector(3, 0, 2, 4, 6), Vector(5, 2, 0, 3, 8), Vector(7, 4, 3, 0, 2), Vector(9, 6, 8, 2, 0))"
> pi = Vector1@2106 "Vector(0, 3, 2, 4, 1)"

```

Figura 61: Debug ProgramaciónRiegoOptimo

```

Local
> $this = RiegoOptimo@1994
> f$3 = Vector1@1992 "Vector((5,4,2), (5,2,4), (8,3,2), (6,2,1), (6,2,3))"
> d$2 = Vector1@1993 "Vector(Vector(0, 3, 5, 7, 9), Vector(3, 0, 2, 4, 6), Vector(5, 2, 0, 3, 8), Vector(7, 4, 3, 0, 2), Vector(9, 6, 8, 2, 0))"
> pi = Vector1@2102 "Vector(0, 3, 2, 1, 4)"

```

Figura 62: Debug ProgramaciónRiegoOptimo

```

Local
> $this = RiegoOptimo@1994
> f$3 = Vector1@1992 "Vector((5,4,2), (5,2,4), (8,3,2), (6,2,1), (6,2,3))"
> d$2 = Vector1@1993 "Vector(Vector(0, 3, 5, 7, 9), Vector(3, 0, 2, 4, 6), Vector(5, 2, 0, 3, 8), Vector(7, 4, 3, 0, 2), Vector(9, 6, 8, 2, 0))"
> pi = Vector1@2098 "Vector(0, 3, 1, 4, 2)"

```

Figura 63: Debug ProgramaciónRiegoOptimo

```

Local
> $this = RiegoOptimo@1994
> f$3 = Vector1@1992 "Vector((5,4,2), (5,2,4), (8,3,2), (6,2,1), (6,2,3))"
> d$2 = Vector1@1993 "Vector(Vector(0, 3, 5, 7, 9), Vector(3, 0, 2, 4, 6), Vector(5, 2, 0, 3, 8), Vector(7, 4, 3, 0, 2), Vector(9, 6, 8, 2, 0))"
> pi = Vector1@2094 "Vector(0, 3, 1, 2, 4)"

```

Figura 64: Debug ProgramaciónRiegoOptimo

```

Local
> $this = RiegoOptimo@1994
> f$3 = Vector1@1992 "Vector((5,4,2), (5,2,4), (8,3,2), (6,2,1), (6,2,3))"
> d$2 = Vector1@1993 "Vector(Vector(0, 3, 5, 7, 9), Vector(3, 0, 2, 4, 6), Vector(5, 2, 0, 3, 8), Vector(7, 4, 3, 0, 2), Vector(9, 6, 8, 2, 0))"
> pi = Vector1@2090 "Vector(0, 2, 4, 3, 1)"

```

Figura 65: Debug ProgramaciónRiegoOptimo

```

Local
> $this = RiegoOptimo@1994
> f$3 = Vector1@1992 "Vector((5,4,2), (5,2,4), (8,3,2), (6,2,1), (6,2,3))"
> d$2 = Vector1@1993 "Vector(Vector(0, 3, 5, 7, 9), Vector(3, 0, 2, 4, 6), Vector(5, 2, 0, 3, 8), Vector(7, 4, 3, 0, 2), Vector(9, 6, 8, 2, 0))"
> pi = Vector1@2086 "Vector(0, 2, 4, 1, 3)"

```

Figura 66: Debug ProgramaciónRiegoOptimo

El costo de riego se calcula de la siguiente manera:

- Si el tiempo de inicio cumple la restricción de supervivencia, el costo es *tiempo de supervivencia* – *tiempo final*.
- De lo contrario, se penaliza según la prioridad:  $prioridad \times (tiempo\ final - tiempo\ de\ supervivencia)$ .

```

Local
> $this = RiegoOptimo@1994
> f$3 = Vector1@1992 "Vector((5,4,2), (5,2,4), (8,3,2), (6,2,1), (6,2,3))"
> d$2 = Vector1@1993 "Vector(Vector(0, 3, 5, 7, 9), Vector(3, 0, 2, 4, 6), Vector(5, 2, 0, 3, 8), Vector(7, 4, 3, 0, 2), Vector(9, 6, 8, 2, 0))"
> pi = Vector1@2082 "Vector(0, 2, 3, 4, 1)"

```

Figura 67: Debug ProgramaciónRiegoOptimo

```

Local
> $this = RiegoOptimo@1994
> f$3 = Vector1@1992 "Vector((5,4,2), (5,2,4), (8,3,2), (6,2,1), (6,2,3))"
> d$2 = Vector1@1993 "Vector(Vector(0, 3, 5, 7, 9), Vector(3, 0, 2, 4, 6), Vector(5, 2, 0, 3, 8), Vector(7, 4, 3, 0, 2), Vector(9, 6, 8, 2, 0))"
> pi = Vector1@2078 "Vector(0, 2, 3, 1, 4)"

```

Figura 68: Debug ProgramaciónRiegoOptimo

```

Local
> $this = RiegoOptimo@1994
> f$3 = Vector1@1992 "Vector((5,4,2), (5,2,4), (8,3,2), (6,2,1), (6,2,3))"
> d$2 = Vector1@1993 "Vector(Vector(0, 3, 5, 7, 9), Vector(3, 0, 2, 4, 6), Vector(5, 2, 0, 3, 8), Vector(7, 4, 3, 0, 2), Vector(9, 6, 8, 2, 0))"
> pi = Vector1@2074 "Vector(0, 2, 1, 4, 3)"

```

Figura 69: Debug ProgramaciónRiegoOptimo

```

Local
> $this = RiegoOptimo@1994
> f$3 = Vector1@1992 "Vector((5,4,2), (5,2,4), (8,3,2), (6,2,1), (6,2,3))"
> d$2 = Vector1@1993 "Vector(Vector(0, 3, 5, 7, 9), Vector(3, 0, 2, 4, 6), Vector(5, 2, 0, 3, 8), Vector(7, 4, 3, 0, 2), Vector(9, 6, 8, 2, 0))"
> pi = Vector1@2062 "Vector(0, 1, 4, 2, 3)"

```

Figura 70: Debug ProgramaciónRiegoOptimo

```

Local
> $this = RiegoOptimo@1994
> f$3 = Vector1@1992 "Vector((5,4,2), (5,2,4), (8,3,2), (6,2,1), (6,2,3))"
> d$2 = Vector1@1993 "Vector(Vector(0, 3, 5, 7, 9), Vector(3, 0, 2, 4, 6), Vector(5, 2, 0, 3, 8), Vector(7, 4, 3, 0, 2), Vector(9, 6, 8, 2, 0))"
> pi = Vector1@2058 "Vector(0, 1, 3, 4, 2)"

```

Figura 71: Debug ProgramaciónRiegoOptimo

El costo de movilidad se calcula sumando las distancias entre tableros consecutivos en la programación:

$$\text{Costo de movilidad} = \sum_{i=0}^{n-1} \text{distancia}(pi(i), pi(i+1)). \quad (2)$$

```

VARIABLES
Local
> $this = RiegoOptimo@1994
> f$3 = Vector1@1992 "Vector((5,4,2), (5,2,4), (8,3,2), (6,2,1), (6,2,3))"
> d$2 = Vector1@1993 "Vector(Vector(0, 3, 5, 7, 9), Vector(3, 0, 2, 4, 6), Vector(5, 2, 0, 3, 8), Vector(7, 4, 3, 0, 2), Vector(9, 6, 8, 2, 0))"
> pi = Vector1@2054 "Vector(0, 1, 3, 2, 4)"

```

Figura 72: Debug ProgramaciónRiegoOptimo

```

Local
> $this = RiegoOptimo@1994
> f$3 = Vector1@1992 "Vector((5,4,2), (5,2,4), (8,3,2), (6,2,1), (6,2,3))"
> d$2 = Vector1@1993 "Vector(Vector(0, 3, 5, 7, 9), Vector(3, 0, 2, 4, 6), Vector(5, 2, 0, 3, 8), Vector(7, 4, 3, 0, 2), Vector(9, 6, 8, 2, 0))"
> pi = Vector1@2048 "Vector(0, 1, 2, 4, 3)"

```

Figura 73: Debug ProgramaciónRiegoOptimo

```

Local
> $this = RiegoOptimo@1994
> f$3 = Vector1@1992 "Vector((5,4,2), (5,2,4), (8,3,2), (6,2,1), (6,2,3))"
> d$2 = Vector1@1993 "Vector(Vector(0, 3, 5, 7, 9), Vector(3, 0, 2, 4, 6), Vector(5, 2, 0, 3, 8), Vector(7, 4, 3, 0, 2), Vector(9, 6, 8, 2, 0))"
> pi = Vector1@2028 "Vector(0, 1, 2, 3, 4)"

```

Figura 74: Debug ProgramaciónRiegoOptimo

```

Local
> f = Vector1@1992 "Vector((5,4,2), (5,2,4), (8,3,2), (6,2,1), (6,2,3))"
> d = Vector1@1993 "Vector(Vector(0, 3, 5, 7, 9), Vector(3, 0, 2, 4, 6), Vector(5, 2, 0, 3, 8), Vector(7, 4, 3, 0, 2), Vector(9, 6, 8, 2, 0))"
> programaciones = Vector2@2020 "Vector(Vector(0, 1, 2, 3, 4), Vector(0, 1, 2, 4, 3), Vector(0, 1, 3, 2, 4), Vector(0, 1, 3, 4, 2), Vector(0, 1, 4, 2, 3), Vector(0, 1, 4, 3, 2))"
> data2 = Object[2]@2024
  len1 = 32
  length0 = 120
> prefix1 = Object[32]@2025
> suffix1 = Object[24]@2026
> this = RiegoOptimo@1994

```

Figura 75: Debug ProgramaciónRiegoOptimo

Finalmente, se selecciona la programación con el costo total (riego + movilidad) mínimo:

$$Programación\ óptima = \min_{p_i} (Costo\ de\ riego + Costo\ de\ movilidad). \quad (3)$$

## Conclusión

La función `ProgramacionRiegoOptimo` se ejecutó correctamente, generando la programación de riego óptima y calculando su costo total de manera adecuada. El test realizado confirmó que la función es capaz de seleccionar la programación más eficiente en cuanto a los costos de riego y movilidad, garantizando que se cumplan las restricciones de la finca. Los resultados obtenidos fueron consistentes con las expectativas, lo que demuestra que la implementación funciona de manera correcta y efectiva.

## 1.2. Funciones para acelerar los cálculos con paralelismo de tareas y de datos

### 1.2.1. Función para calcular el costo de riego Finca paralela (`costoRiegoFincaPar`)

La función `costoRiegoFincaPar` calcula el costo total de riego para una finca de tablonos dada una programación de riego. Se implementa utilizando procesamiento paralelo para optimizar el cálculo del costo individual de cada tablón. Para probar el correcto funcionamiento de esta función se define el siguiente test:

```

1 test("Costo para una finca con tres tablonos y programaci n en orden") {
2   val finca = Vector(
3     (12, 4, 3), // Tabl n 1
4     (7, 3, 2),  // Tabl n 2
5     (9, 5, 1)   // Tabl n 3
6   )
7   val programacion = Vector(0,1, 2) // Orden: Tabl n 3 -> 2 -> 1
8
9   assert(riegoOptimo.costoRiegoFincaPar(finca, programacion) == 11) // Suma de
10  costos
11 }

```

Este test evalúa el costo de riego para una finca con tres tablonos con programación en orden. El programa empieza definiendo los datos de entrada: **Finca**: Vector con tres tablonos

- Tablón 1: (12, 4, 3)
- Tablón 2: (7, 3, 2)
- Tablón 3: (9, 5, 1).

**Programación**: Vector (0,1,2), que indica el orden de riego.

```

> @ $this = {CostoRiegoFincaParTest@2368} CostoRiegoFincaParTest
> riegoOptimo = {RiegoOptimo@2369} taller.RiegoOptimo@6f8e8894

```

Figura 76: Debug CostoRiegoFincaPar

```

> (P) $this = {CostoRiegoFincaParTest@2368} CostoRiegoFincaParTest
v  finca = {Vector1@2748} size = 3
  > 0 = {Tuple3@2751} (12,4,3)
  > 1 = {Tuple3@2752} (7,3,2)
  > 2 = {Tuple3@2753} (9,5,1)
> riegoOptimo = {RiegoOptimo@2369} taller.RiegoOptimo@6f8e8894

```

Figura 77: Debug CostoRiegoFincaPar

```

> finca = {Vector1@2748} size = 3
v  programacion = {Vector1@2761} size = 3
  > 0 = {Integer@2767} 0
  > 1 = {Integer@2759} 1
  > 2 = {Integer@2768} 2

```

Figura 78: Debug CostoRiegoFincaPar

Se utiliza la misma función de CostoRiegoTablones para calcular el costo por cada tablón y por último se suman los costos, para obtener el costo total. La diferencia es que esta se realiza de manera paralela.

```

def costoRiegoFincaPar(f:Finca, pi:ProgRiego) : Int = {  JeidyMunillo      f: size = 3    pi: size = 3
  // Devuelve el costo total de regar una finca f dada una programación de riego pi, calculando en paralelo
  (0 <= until < f.length).par.map(i => costoRiegoTablon(i, f, pi)).sum  f: size = 3    pi: size = 3
}

```

Figura 79: Debug CostoRiegoFincaPar

```

> this = {RiegoOptimo@2369} taller.RiegoOptimo@6f8e8894
v  (P) f = {Vector1@2748} size = 3
  > 0 = {Tuple3@2751} (12,4,3)
  > 1 = {Tuple3@2752} (7,3,2)
  > 2 = {Tuple3@2753} (9,5,1)
v  (P) pi = {Vector1@2761} size = 3
  > 0 = {Integer@2767} 0
  > 1 = {Integer@2759} 1
  > 2 = {Integer@2768} 2

```

Figura 80: Debug CostoRiegoFincaPar

```
def costoRiegoFincaPar(f:Finca, pi:ProgRiego) : Int = { ▲ JeidyMunillo f: size = 5 pi: size = 5
// Devuelve el costo total de regar una finca f dada una programación de riego pi, calculando en paralelo
(0 ≤ until < f.length).par.map(i => ● costoRiegoTablon(i, f, pi)).sum f: size = 5 pi: size = 5
```

Figura 81: Debug CostoRiegoFincaPar

Aquí se registran los resultados parciales de los costos individuales de cada tablón, obtenidos de forma independiente en distintos hilos.

```
✓ P f = {Vector1@2959} size = 5
> 0 = {Tuple3@2963} (18,6,3)
> 1 = {Tuple3@2964} (10,3,5)
> 2 = {Tuple3@2965} (13,4,4)
> 3 = {Tuple3@2966} (8,2,2)
> 4 = {Tuple3@2967} (15,5,3)
✓ P pi = {Vector1@2960} size = 5
> 0 = {Integer@2767} 0
> 1 = {Integer@2759} 1
> 2 = {Integer@2768} 2
> 3 = {Integer@2973} 3
> 4 = {Integer@2974} 4
```

Figura 82: Debug CostoRiegoFincaPar

```
def costoRiegoFincaPar(f:Finca, pi:ProgRiego) : Int = { ▲ JeidyMunillo f: size = 8 pi: size = 8
// Devuelve el costo total de regar una finca f dada una programación de riego pi, calculando en paralelo
(0 ≤ until < f.length).par.map(i => ● costoRiegoTablon(i, f, pi)).sum f: size = 8 pi: size = 8
```

Figura 83: Debug CostoRiegoFincaPar

```

v (p) f = {Vector1@2981} size = 8
> [0] 0 = {Tuple3@2985} (25,7,3)
> [1] 1 = {Tuple3@2986} (12,5,5)
> [2] 2 = {Tuple3@2987} (17,4,4)
> [3] 3 = {Tuple3@2988} (14,6,2)
> [4] 4 = {Tuple3@2989} (20,8,3)
> [5] 5 = {Tuple3@2990} (10,3,5)
> [6] 6 = {Tuple3@2991} (16,5,4)
> [7] 7 = {Tuple3@2992} (18,7,2)
> (p) pi = {Vector1@2982} size = 8

```

Figura 84: Debug CostoRiegoFincaPar

Se visualiza la operación de reducción paralela que ocurre después del cálculo individual. Es decir, los costos calculados en paralelo se suman secuencialmente para obtener el costo total.

```

> [this] this = {RiegoOptimo@2369} taller.RiegoOptimo@6f8e8894
> (p) f = {Vector1@3004} size = 10
> (p) pi = {Vector1@3005} size = 10

```

Figura 85: Debug CostoRiegoFincaPar

```

> [this] this = {RiegoOptimo@2369} taller.RiegoOptimo@6f8e8894
> (p) f = {Vector1@3028} size = 12
> (p) pi = {Vector1@3029} size = 12

```

Figura 86: Debug CostoRiegoFincaPar

Finalmente, se muestra el resultado total del costo de riego, que en este caso es 11, coincidiendo con el valor esperado en el test.



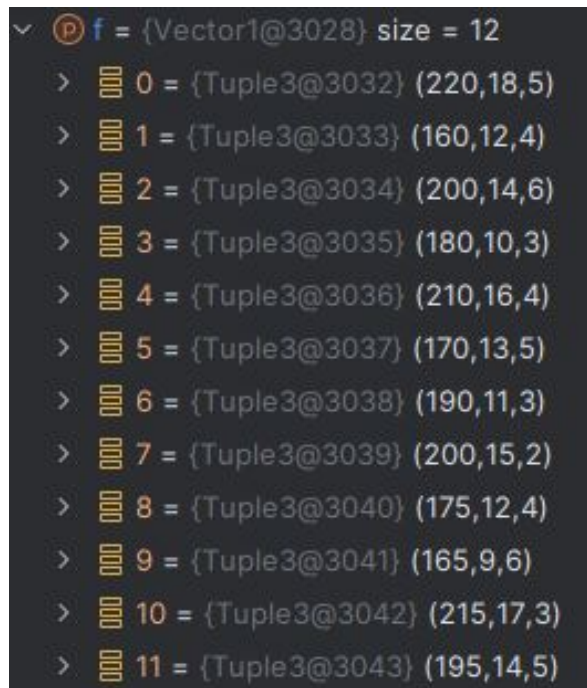


Figura 87: Debug CostoRiegoFincaPar

### 1.2.2. Función para calcular el costo de movilidad paralela (costoRiegoMovilidadPar)

El test evalúa el comportamiento de la función `costoMovilidadPar`, que utiliza paralelismo para calcular el costo de movilidad basado en una matriz de distancias y un programa de riego. Test seleccionado para el debug:

```

test("Caso con distancias dispersas y programa de riego irregular") {
  val distancias: riegoOptimo.Distancia = Vector(
    Vector(0, 3, 8, 6),
    Vector(5, 0, 4, 9),
    Vector(7, 2, 0, 1),
    Vector(6, 5, 3, 0)
  )
  val finca: riegoOptimo.Finca = Vector(
    (18, 3, 2),
    (21, 2, 3),
    (19, 4, 1),
    (22, 5, 4)
  )
  val progRiego: riegoOptimo.ProgRiego = Vector(2, 1, 3, 0, 2)
  val costo = riegoOptimo.costoMovilidadPar(finca, progRiego, distancias)
  assert(costo == 25, s"El costo esperado es 25, y se obtuvo $costo")
}

```

Figura 88: Debug CostoMovilidadPar

La función `costoMovilidadPar` calcula el costo total de movilidad entre parcelas visitadas en el programa de riego `pi`, usando un procesamiento paralelo.

```

1 def costoMovilidadPar(f: Finca, pi: ProgRiego, d: Distancia): Int = {
2   // Calcula el costo de movilidad de manera paralela
3   (0 until pi.length - 1).par.map(j => d(pi(j))(pi(j + 1))).sum
4 }

```



- **Matriz de Distancias (distancias):** Define las distancias entre cada par de parcelas.

$$\text{distancias} = \begin{bmatrix} 0 & 3 & 8 & 6 \\ 5 & 0 & 4 & 9 \\ 7 & 2 & 0 & 1 \\ 6 & 5 & 3 & 0 \end{bmatrix}$$

- **Programa de Riego (progRiego):** Orden en que se visitan las parcelas:

`progRiego = [2, 1, 3, 0, 2]`

```

v (P) f = {Vector1@2739} size = 4
  > [0] 0 = {Tuple3@2745} (30,5,2)
  > [1] 1 = {Tuple3@2746} (50,3,1)
  > [2] 2 = {Tuple3@2747} (60,4,4)
  > [3] 3 = {Tuple3@2748} (40,6,3)
v (P) pi = {Vector1@2740} size = 4
  > [0] 0 = {Integer@2753} 3
  > [1] 1 = {Integer@2754} 0
  > [2] 2 = {Integer@2755} 2
  > [3] 3 = {Integer@2756} 1
v (P) d = {Vector1@2741} size = 4
  > [0] 0 = {Vector1@2757} size = 4
  > [1] 1 = {Vector1@2758} size = 4
  > [2] 2 = {Vector1@2759} size = 4
  > [3] 3 = {Vector1@2760} size = 4

```

Figura 89: Debug CostoMovilidadPar

```
def costoMovilidadPar(f:Finca,pi:ProgRiego, d:Distancia) : Int = {
  // Calcula el costo de movilidad de manera paralela
  (0 ≤ until < pi.length - 1).par.map(j => d(pi(j))(pi(j+1))).sum
}
```

Figura 90: Debug CostoMovilidadPar

La función recorre los índices del programa de riego y calcula las distancias entre parcelas consecutivas, ejecutándose en paralelo.

El rango de índices es:

$$0 \leq j < (\text{progRiego.length} - 1) \implies j \in \{0, 1, 2, 3\}$$

```
> (P) f = {Vector1@2947} size = 4
v (P) pi = {Vector1@2948} size = 5
> (P) 0 = {Integer@2940} 0
> (P) 1 = {Integer@2938} 1
> (P) 2 = {Integer@2939} 2
> (P) 3 = {Integer@2941} 3
> (P) 4 = {Integer@2940} 0
> (P) d = {Vector1@2949} size = 4
```

Figura 91: Debug CostoMovilidadPar

Se calcula la distancia para cada  $j$ , usando la matriz `distancias`:

- $j = 0$ :  $\text{progRiego}(0) = 2, \text{progRiego}(1) = 1$

$$d(\text{progRiego}(0), \text{progRiego}(1)) = d(2, 1) = 2$$

- $j = 1$ :  $\text{progRiego}(1) = 1, \text{progRiego}(2) = 3$

$$d(\text{progRiego}(1), \text{progRiego}(2)) = d(1, 3) = 9$$

- $j = 2$ :  $\text{progRiego}(2) = 3, \text{progRiego}(3) = 0$

$$d(\text{progRiego}(2), \text{progRiego}(3)) = d(3, 0) = 6$$

- $j = 3$ :  $\text{progRiego}(3) = 0, \text{progRiego}(4) = 2$

$$d(\text{progRiego}(3), \text{progRiego}(4)) = d(0, 2) = 8$$

```

> (P) $this = {costoMovilidadParTest@2475} costoMovilidadParTest
✓ [ ] distancias = {Vector1@2965} size = 4
  > [ ] 0 = {Vector1@2968} size = 4
  > [ ] 1 = {Vector1@2969} size = 4
  > [ ] 2 = {Vector1@2970} size = 4
  > [ ] 3 = {Vector1@2971} size = 4

```

Figura 92: Debug CostoMovilidadPar

```

> (P) $this = {costoMovilidadParTest@2475} costoMovilidadParTest
> [ ] distancias = {Vector1@2965} size = 4
✓ [ ] finca = {Vector1@2980} size = 4
  > [ ] 0 = {Tuple3@2983} (18,3,2)
  > [ ] 1 = {Tuple3@2984} (21,2,3)
  > [ ] 2 = {Tuple3@2985} (19,4,1)
  > [ ] 3 = {Tuple3@2986} (22,5,4)

```

Figura 93: Debug CostoMovilidadPar

```

val progRiego: riegoOptimo.ProgRiego = Vector(2, 1, 3, 0, 2) progRiego: size = 5
val costo = riegoOptimo.costoMovilidadPar(finca, progRiego, distancias) distancias: size = 4 finca: size = 4 progRiego: size = 5

```

Figura 94: Debug CostoMovilidadPar

El vector de distancias procesado en paralelo es:

[2, 9, 6, 8]

La suma de las distancias es:

$$\text{costo} = 2 + 9 + 6 + 8 = 25$$

```

val costo = riegoOptimo.costoMovilidadPar(finca, progRiego, distancias) costo: 25

```

Figura 95: Debug CostoMovilidadPar

```

> (P) $this = {costoMovilidadParTest@2475} costoMovilidadParTest
> [ ] distancias = {Vector1@2965} size = 4
> [ ] finca = {Vector1@2980} size = 4
> [ ] progRiego = {Vector1@2991} size = 5
10 01 costo = 25

```

Figura 96: Debug CostoMovilidadPar

El costo esperado se calcula manualmente:

```
costoEsperado = 25
```

El `assert` verifica la igualdad:

```
assert(25 == 25)
```

```
val costo = riegoOptimo.costoMovilidadPar(finca, progRiego, distancias) costo: 25
assert(costo == 25, s"El costo esperado es 25, y se obtuvo $costo") costo: 25
```

Figura 97: Debug CostoMovilidadPar

El test pasa correctamente.

El test demuestra que la función `costoMovilidadPar` calcula correctamente el costo total de movilidad incluso con entradas dispersas e irregulares. La implementación paralela garantiza un rendimiento eficiente al procesar las distancias.

### 1.2.3. Función para generar programaciones de riego en paralelo (`generarProgramacionesRiegoPar`)

Este test verifica el correcto funcionamiento de la función `generarProgramacionesRiegoPar`, que genera todas las posibles programaciones de riego para una finca con 3 tablonos utilizando procesamiento en paralelo.

```
Local
> f = Vector1@1969 "Vector((3,1,1), (3,2,4), (5,3,4))"
> this = RiegoOptimo@1970
```

Figura 98: Visualización inicial de la variable `f`.

La función toma como entrada una finca (`f`) y calcula todas las permutaciones de los índices asociados a sus tablonos.

```
1 def generarProgramacionesRiegoPar(f: Finca): Vector[ProgRiego] = {
2   val indices = (0 until f.length).toVector
3   indices.permutations.toVector.par.toVector
4 }
```

La función realiza las siguientes operaciones:

1. **Cálculo del vector de índices:** A partir del tamaño de la finca, se genera un vector de índices:

```
indices = [0, 1, 2]
```

donde cada índice representa la posición de un tablón en la finca.

2. **Generación de permutaciones:** Se calculan todas las posibles permutaciones de los índices utilizando el método `permutations`. Las permutaciones obtenidas representan todas las secuencias posibles de riego.
3. **Procesamiento en paralelo:** Se aplica el método `.par` para calcular las permutaciones en paralelo, mejorando el rendimiento en sistemas con múltiples núcleos.
4. **Resultado:** El resultado final se convierte a un `Vector` de vectores.

A continuación, se muestra el estado del vector de índices en el test:

```
▼ Local
> f = Vector1@1969 "Vector((3,1,1), (3,2,4), (5,3,4))"
> indices = Vector1@1978 "Vector(0, 1, 2)"
> this = RiegoOptimo@1970
```

Figura 99: Cálculo del vector de índices `indices` a partir de la finca `f`.

En este caso, el vector `indices` tiene los valores:

$$\text{indices} = [0, 1, 2].$$

La función genera las siguientes permutaciones de índices:

$$\text{indices.permutations} = \begin{bmatrix} [0, 1, 2] \\ [0, 2, 1] \\ [1, 0, 2] \\ [1, 2, 0] \\ [2, 0, 1] \\ [2, 1, 0] \end{bmatrix}.$$

Estas permutaciones representan todas las secuencias posibles de riego para los 3 tablonos de la finca.

El resultado final es un `Vector` de vectores, donde cada subvector contiene una secuencia específica de índices de riego.

```
programaciones = Vector( Vector(0, 1, 2), Vector(0, 2, 1), Vector(1, 0, 2), Vector(1, 2, 0), Vector(2, 0, 1), Vector(2, 1, 0) )
```

El test compara el tamaño del resultado con el valor esperado  $3! = 6$ :

```
assert(programaciones.size == 6).
```

Dado que el resultado coincide con las expectativas, el test pasa exitosamente.

```
▼ Local
> f = Vector1@1969 "Vector((3,1,1), (3,2,4), (5,3,4))"
> indices = Vector1@1978 "Vector(0, 1, 2)"
> this = RiegoOptimo@1970
```

Figura 100: Ejecución exitosa: vector de índices generado correctamente.

## Conclusión

La función `generarProgramacionesRiegoPar` funciona correctamente al generar todas las posibles secuencias de riego para una finca con 3 tablonos, y optimiza el cálculo mediante el procesamiento paralelo. Y el test es ejecutado exitosamente.

#### 1.2.4. Función para calcular programaciones de riego óptimas paralelas (ProgramacionRiegoOptimoPar)

La función `ProgramacionRiegoOptimoPar` toma como entrada una finca (`f`) y una matriz de distancias (`d`), y devuelve la programación de riego óptima junto con su costo total, optimizando el cálculo mediante paralelización.

```
1 def ProgramacionRiegoOptimoPar(f:Finca, d:Distancia) : (ProgRiego, Int) = {  
2   // Dada una finca, calcula la programación óptima de riego  
3   val programaciones = generarProgramacionesRiegoPar(f)  
4   val costos = programaciones.par.map(pi =>  
5     (pi, costoRiegoFincaPar(f, pi) + costoMovilidadPar(f, pi, d))  
6   )  
7   costos.minBy(_._2)  
8 }
```

La función realiza las siguientes operaciones:

1. **Generación de programaciones de riego paralelas:** Utiliza la función `generarProgramacionesRiegoPar(f)` para calcular todas las permutaciones posibles de riego de los tabloncillos en la finca. Este proceso se optimiza usando paralelización.
2. **Cálculo paralelo de costos:** Cada programación (`pi`) se evalúa de forma paralela (`par.map`), sumando el costo de riego calculado con `costoRiegoFincaPar` y el costo de movilidad con `costoMovilidadPar`.
3. **Selección de la programación óptima:** Utiliza `minBy` para seleccionar la programación con el costo total mínimo.

El resultado final es una tupla que contiene la programación óptima y su costo total asociado.

```
✓ Local  
> f = Vector1@1992 "Vector((1,3,2), (4,2,3), (6,3,3))"  
> d = Vector1@1993 "Vector(Vector(0, 1, 2), Vector(1, 0, 1), Vector(2, 1, 0))"  
> this = RiegoOptimo@1994
```

Figura 101: Debug inicial de `ProgramacionRiegoOptimoPar`

#### 1.2.5. Etapa 1: Entrada de datos

En esta etapa, se genera:

- **Finca:** Un vector de tabloncillos con atributos como (*tiempo de supervivencia, tiempo de riego, prioridad*).
- **Matriz de distancias:** Una matriz no uniforme que contiene las distancias entre pares de tabloncillos.

Estos datos constituyen la entrada para las siguientes etapas.

```
✓ Local  
> x$2 = Tuple2@2224 "(Vector(2, 0, 1),28)"
```

Figura 102: Datos de entrada para la función `ProgramacionRiegoOptimoPar`

### 1.2.6. Etapa 2: Generación de programaciones de riego

Se generan todas las permutaciones posibles para la programación de riego. Estas representan todos los órdenes en los que se pueden regar los tabloncillos. Este proceso se paraleliza para aumentar la eficiencia.

```
> f = Vector1@1992 "Vector((1,3,2), (4,2,3), (6,3,3))"
> d = Vector1@1993 "Vector(Vector(0, 1, 2), Vector(1, 0, 1), Vector(2, 1, 0))"
> programaciones = Vector1@2089 "Vector(Vector(0, 1, 2), Vector(0, 2, 1), Vector(1, 0, 2), Vector(1, 2, 0), Vector(2, 0, 1), Vector(2, 1, 0))"
> costos = ParVector@2216 "ParVector((Vector(0, 1, 2),15), (Vector(0, 2, 1),19), (Vector(1, 0, 2),19), (Vector(1, 2, 0),20), (Vector(2, 0, 1),28),
> this = RiegoOptimo@1994
```

Figura 103: Programaciones generadas para la finca

### 1.2.7. Etapa 3: Cálculo de costos

El cálculo de costos también se realiza en paralelo. Para cada programación  $pi$ , se calculan:

- **Costo de riego:** Si el tiempo de inicio cumple con las restricciones de supervivencia del tablón, el costo es proporcional al tiempo sobrante. Si no, se penaliza en función de la prioridad.

$$Costo\ de\ riego = \begin{cases} tiempo\ de\ supervivencia - tiempo\ final & \text{si cumple restricción} \\ prioridad \times (tiempo\ final - tiempo\ de\ supervivencia) & \text{si no cumple restricción} \end{cases} \quad (4)$$

- **Costo de movilidad:** Se calcula como la suma de las distancias entre tabloncillos consecutivos en la programación:

$$Costo\ de\ movilidad = \sum_{i=0}^{n-1} distancia(pi(i), pi(i+1)). \quad (5)$$

```
Local
> $this = RiegoOptimo@1994
> f$5 = Vector1@1992 "Vector((1,3,2), (4,2,3), (6,3,3))"
> d$4 = Vector1@1993 "Vector(Vector(0, 1, 2), Vector(1, 0, 1), Vector(2, 1, 0))"
> pi = Vector1@2165 "Vector(0, 2, 1)"
```

Figura 104: Cálculo paralelo de costos de riego y movilidad

### 1.2.8. Etapa 4: Selección de la programación óptima

Finalmente, se selecciona la programación con el costo total más bajo, sumando costos de riego y movilidad:

$$Programación\ óptima = \min_{pi} (Costo\ de\ riego + Costo\ de\ movilidad). \quad (6)$$

```
Local
> f = Vector1@1992 "Vector((1,3,2), (4,2,3), (6,3,3))"
> d = Vector1@1993 "Vector(Vector(0, 1, 2), Vector(1, 0, 1), Vector(2, 1, 0))"
> programaciones = Vector1@2089 "Vector(Vector(0, 1, 2), Vector(0, 2, 1), Vector(1, 0, 2), Vector(1, 2, 0), Vector(2, 0, 1), Vector(2, 1, 0))"
> this = RiegoOptimo@1994
```

Figura 105: Selección de la programación con menor costo total

### 1.2.9. Conclusión

La función `ProgramacionRiegoOptimoPar` optimiza el cálculo de la programación de riego utilizando técnicas de paralelización. El proceso incluye generación de programaciones, cálculo de costos y selección de la mejor opción. Los resultados del \*debug\* confirman que el algoritmo identifica correctamente la programación óptima con menor costo total en escenarios complejos y paralelos.

## 2. Informe de Paralización

### 2.1. Función para calcular el costo de riego Finca paralela (`costoRiegoFincaPar`)

La función `costoRiegoFincaPar` calcula el costo total de riego de una finca dividida en múltiples tablonos. Su implementación se basa en recorrer cada tablón y sumar el costo de riego de manera independiente.

**Estrategia de Paralelización** Para paralelizar la función, se utilizó el método `.par`, que convierte una colección secuencial en una colección paralela. La estrategia incluye:

- Dividir el rango `0 until f.length` en subproblemas.
- Calcular el costo de riego para cada tablón de forma independiente utilizando múltiples hilos.
- Reducir los resultados parciales mediante la operación `sum`.

Esta aproximación permite explotar el paralelismo inherente al problema debido a la independencia entre los cálculos de los tablonos.

**Ley de Amdahl** La ley de Amdahl permite estimar las ganancias de rendimiento obtenidas mediante paralelización. Su fórmula es:

$$S = \frac{1}{(1 - P) + \frac{P}{N}}$$

donde  $P$  es la fracción paralelizable del programa y  $N$  es el número de hilos disponibles.

La función `costoRiegoFincaPar` se beneficia de la paralelización principalmente en el cálculo de los costos de riego de los tablonos, lo cual es independiente y puede ejecutarse en paralelo para cada tablón. Este cálculo es la porción paralelizable de la función.

**Estimación de la fracción paralelizable  $p$ :** La fracción paralelizable  $p$  depende de la parte del cálculo que se puede ejecutar en paralelo. En este caso, el cálculo de los costos de riego es completamente independiente, lo que implica que toda la operación de cálculo es paralelizable. Por lo tanto, podemos estimar que  $p$  es cercano a 1 (o 100 % paralelizable) ( $P \approx 0,95$ ).

La suma de los costos, por otro lado, es una operación secuencial que no puede paralelizarse, lo que limita la ganancia de rendimiento.

En el mejor de los casos, la aceleración será proporcional al número de procesadores disponibles. Sin embargo, la aceleración máxima solo se logra cuando casi toda la operación es paralelizable, lo que no es el caso debido a la suma secuencial.

**Resultados de Pruebas** Se realizaron pruebas con cinco tamaños de entrada diferentes, ejecutando diez iteraciones por cada tamaño. Las figuras a continuación muestran los tiempos de ejecución promedio y las ganancias de rendimiento:



```

> Task :app:classes UP-TO-DATE

> Task :app:run
Hello, world!

..... Iniciando benchmarking de Riego Optimo .....

----- Benchmarking: Costos de Riego y Movilidad -----

===== Tamaño finca: 4 =====

Comparación de las funciones secuenciales y paralelas para el tamaño de la finca: 4 (Iteración 1)

Unable to create a system terminal
-----
| Métrica                | Tiempo (ms) |
-----
| Secuencial              | 0,1182      |
| Paralelo                 | 0,5946      |
| Aceleración             | 0,1988      |
-----

Comparación de las funciones secuenciales y paralelas para el tamaño de la finca: 4 (Iteración 2)

-----
| Métrica                | Tiempo (ms) |
-----
| Secuencial              | 0,0436      |
| Paralelo                 | 0,6686      |
| Aceleración             | 0,0652      |
-----

Comparación de las funciones secuenciales y paralelas para el tamaño de la finca: 4 (Iteración 3)

-----
| Métrica                | Tiempo (ms) |
-----
| Secuencial              | 0,0344      |
| Paralelo                 | 0,5823      |
| Aceleración             | 0,0591      |
-----

Comparación de las funciones secuenciales y paralelas para el tamaño de la finca: 4 (Iteración 4)

-----
| Métrica                | Tiempo (ms) |
-----
| Secuencial              | 0,0201      |
| Paralelo                 | 0,3442      |
| Aceleración             | 0,0584      |
-----

```

Figura 106: Prueba tamaño 4x4 iteración 1 - 4

Comparaci3n de las funciones secuenciales y paralelas para el tamao de la finca: 4 (Iteraci3n 5)

Métrica	Tiempo (ms)
Secuencial	0,0118
Paralelo	0,2493
Aceleraci3n	0,0473

Comparaci3n de las funciones secuenciales y paralelas para el tamao de la finca: 4 (Iteraci3n 6)

Métrica	Tiempo (ms)
Secuencial	0,0334
Paralelo	0,1388
Aceleraci3n	0,2406

Comparaci3n de las funciones secuenciales y paralelas para el tamao de la finca: 4 (Iteraci3n 7)

Métrica	Tiempo (ms)
Secuencial	0,0111
Paralelo	0,1653
Aceleraci3n	0,0672

Comparaci3n de las funciones secuenciales y paralelas para el tamao de la finca: 4 (Iteraci3n 8)

Métrica	Tiempo (ms)
Secuencial	0,0084
Paralelo	0,2049
Aceleraci3n	0,0410

Comparaci3n de las funciones secuenciales y paralelas para el tamao de la finca: 4 (Iteraci3n 9)

Métrica	Tiempo (ms)
Secuencial	0,0084
Paralelo	0,1090
Aceleraci3n	0,0771

Figura 107: Prueba tamao 4x4 iteraci3n 5-9

Comparaci3n de las funciones secuenciales y paralelas para el tama1o de la finca: 4 (Iteraci3n 10)

M3trica	Tiempo (ms)
Secuencial	0,0074
Paralelo	0,8224
Aceleraci3n	0,0090

===== Tama1o finca: 5 =====

Comparaci3n de las funciones secuenciales y paralelas para el tama1o de la finca: 5 (Iteraci3n 1)

M3trica	Tiempo (ms)
Secuencial	0,0093
Paralelo	0,1955
Aceleraci3n	0,0476

Comparaci3n de las funciones secuenciales y paralelas para el tama1o de la finca: 5 (Iteraci3n 2)

M3trica	Tiempo (ms)
Secuencial	0,0074
Paralelo	0,1084
Aceleraci3n	0,0683

Comparaci3n de las funciones secuenciales y paralelas para el tama1o de la finca: 5 (Iteraci3n 3)

M3trica	Tiempo (ms)
Secuencial	0,0146
Paralelo	0,1362
Aceleraci3n	0,1072

Comparaci3n de las funciones secuenciales y paralelas para el tama1o de la finca: 5 (Iteraci3n 4)

M3trica	Tiempo (ms)
Secuencial	0,0275
Paralelo	0,1541
Aceleraci3n	0,1785

Figura 108: Prueba tama1o 5 iteraci3n 1 - 4

Comparaci�n de las funciones secuenciales y paralelas para el tama�o de la finca: 5 (Iteraci�n 5)	
-----	
M�trica	Tiempo (ms)
-----	
Secuencial	0,0070
Paralelo	0,2364
Aceleraci�n	0,0296
-----	
Comparaci�n de las funciones secuenciales y paralelas para el tama�o de la finca: 5 (Iteraci�n 6)	
-----	
M�trica	Tiempo (ms)
-----	
Secuencial	0,0068
Paralelo	0,1307
Aceleraci�n	0,0520
-----	
Comparaci�n de las funciones secuenciales y paralelas para el tama�o de la finca: 5 (Iteraci�n 7)	
-----	
M�trica	Tiempo (ms)
-----	
Secuencial	0,0124
Paralelo	0,1545
Aceleraci�n	0,0803
-----	
Comparaci�n de las funciones secuenciales y paralelas para el tama�o de la finca: 5 (Iteraci�n 8)	
-----	
M�trica	Tiempo (ms)
-----	
Secuencial	0,0093
Paralelo	0,1508
Aceleraci�n	0,0617
-----	
Comparaci�n de las funciones secuenciales y paralelas para el tama�o de la finca: 5 (Iteraci�n 9)	
-----	
M�trica	Tiempo (ms)
-----	
Secuencial	0,0121
Paralelo	0,1778
Aceleraci�n	0,0681
-----	
Comparaci�n de las funciones secuenciales y paralelas para el tama�o de la finca: 5 (Iteraci�n 10)	
-----	
M�trica	Tiempo (ms)
-----	
Secuencial	0,0056
Paralelo	0,1051
Aceleraci�n	0,0533
-----	

Figura 109: Prueba tama o 5 iteraci n 5 - 10

```

===== Tamaño finca: 6 =====

Comparación de las funciones secuenciales y paralelas para el tamaño de la finca: 6 (Iteración 1)
-----
| Métrica          | Tiempo (ms) |
-----
| Secuencial       | 0,0050      |
| Paralelo         | 0,1205      |
| Aceleración      | 0,0415      |
-----

Comparación de las funciones secuenciales y paralelas para el tamaño de la finca: 6 (Iteración 2)
-----
| Métrica          | Tiempo (ms) |
-----
| Secuencial       | 0,0051      |
| Paralelo         | 0,1078      |
| Aceleración      | 0,0473      |
-----

Comparación de las funciones secuenciales y paralelas para el tamaño de la finca: 6 (Iteración 3)
-----
| Métrica          | Tiempo (ms) |
-----
| Secuencial       | 0,0051      |
| Paralelo         | 0,1440      |
| Aceleración      | 0,0354      |
-----

Comparación de las funciones secuenciales y paralelas para el tamaño de la finca: 6 (Iteración 4)
-----
| Métrica          | Tiempo (ms) |
-----
| Secuencial       | 0,0049      |
| Paralelo         | 0,1021      |
| Aceleración      | 0,0480      |
-----

Comparación de las funciones secuenciales y paralelas para el tamaño de la finca: 6 (Iteración 5)
-----
| Métrica          | Tiempo (ms) |
-----
| Secuencial       | 0,0036      |
| Paralelo         | 0,1188      |
| Aceleración      | 0,0303      |
-----

```

Figura 110: Prueba de tamaño 6 Iteración 1 - 5

```

Comparaci3n de las funciones secuenciales y paralelas para el tama1o de la finca: 6 (Iteraci3n 6)

-----
| M3trica          | Tiempo (ms)
-----
| Secuencial       | 0,0072
| Paralelo         | 0,1513
| Aceleraci3n      | 0,0476
-----

Comparaci3n de las funciones secuenciales y paralelas para el tama1o de la finca: 6 (Iteraci3n 7)

-----
| M3trica          | Tiempo (ms)
-----
| Secuencial       | 0,0030
| Paralelo         | 0,1586
| Aceleraci3n      | 0,0189
-----

Comparaci3n de las funciones secuenciales y paralelas para el tama1o de la finca: 6 (Iteraci3n 8)

-----
| M3trica          | Tiempo (ms)
-----
| Secuencial       | 0,0030
| Paralelo         | 0,1113
| Aceleraci3n      | 0,0270
-----

Comparaci3n de las funciones secuenciales y paralelas para el tama1o de la finca: 6 (Iteraci3n 9)

-----
| M3trica          | Tiempo (ms)
-----
| Secuencial       | 0,0029
| Paralelo         | 0,1327
| Aceleraci3n      | 0,0219
-----

Comparaci3n de las funciones secuenciales y paralelas para el tama1o de la finca: 6 (Iteraci3n 10)

-----
| M3trica          | Tiempo (ms)
-----
| Secuencial       | 0,0029
| Paralelo         | 0,1377
| Aceleraci3n      | 0,0211
-----

```

Figura 111: Prueba tama1o 6 Iteraci3n 6 - 10

```

===== Tamaño finca: 7 =====

Comparación de las funciones secuenciales y paralelas para el tamaño de la finca: 7 (Iteración 1)
-----
| Métrica          | Tiempo (ms) |
-----
| Secuencial       | 0,0046      |
| Paralelo         | 0,1349      |
| Aceleración      | 0,0341      |
-----

Comparación de las funciones secuenciales y paralelas para el tamaño de la finca: 7 (Iteración 2)
-----
| Métrica          | Tiempo (ms) |
-----
| Secuencial       | 0,0032      |
| Paralelo         | 0,1383      |
| Aceleración      | 0,0231      |
-----

Comparación de las funciones secuenciales y paralelas para el tamaño de la finca: 7 (Iteración 3)
-----
| Métrica          | Tiempo (ms) |
-----
| Secuencial       | 0,0076      |
| Paralelo         | 0,2094      |
| Aceleración      | 0,0363      |
-----

Comparación de las funciones secuenciales y paralelas para el tamaño de la finca: 7 (Iteración 4)
-----
| Métrica          | Tiempo (ms) |
-----
| Secuencial       | 0,0058      |
| Paralelo         | 0,1098      |
| Aceleración      | 0,0528      |
-----

Comparación de las funciones secuenciales y paralelas para el tamaño de la finca: 7 (Iteración 5)
-----
| Métrica          | Tiempo (ms) |
-----
| Secuencial       | 0,0053      |
| Paralelo         | 0,1427      |
| Aceleración      | 0,0371      |
-----

```

Figura 112: Prueba tamaño 7 Iteración 1 - 5

Comparación de las funciones secuenciales y paralelas para el tamaño de la finca: 7 (Iteración 6)	
Métrica	Tiempo (ms)
Secuencial	0,0043
Paralelo	0,1898
Aceleración	0,0227
Comparación de las funciones secuenciales y paralelas para el tamaño de la finca: 7 (Iteración 7)	
Métrica	Tiempo (ms)
Secuencial	0,0031
Paralelo	0,1104
Aceleración	0,0281
Comparación de las funciones secuenciales y paralelas para el tamaño de la finca: 7 (Iteración 8)	
Métrica	Tiempo (ms)
Secuencial	0,0033
Paralelo	0,1535
Aceleración	0,0215
Comparación de las funciones secuenciales y paralelas para el tamaño de la finca: 7 (Iteración 9)	
Métrica	Tiempo (ms)
Secuencial	0,0039
Paralelo	0,1155
Aceleración	0,0338
Comparación de las funciones secuenciales y paralelas para el tamaño de la finca: 7 (Iteración 10)	
Métrica	Tiempo (ms)
Secuencial	0,0041
Paralelo	0,1134
Aceleración	0,0362

Figura 113: Prueba tamaño 7 Iteración 5 -10



```

-----

===== Tamaño finca: 8 =====

Comparación de las funciones secuenciales y paralelas para el tamaño de la finca: 8 (Iteración 1)
-----
| Métrica          | Tiempo (ms) |
-----
| Secuencial       | 0,0059      |
| Paralelo         | 0,2472      |
| Aceleración      | 0,0239      |
-----

Comparación de las funciones secuenciales y paralelas para el tamaño de la finca: 8 (Iteración 2)
-----
| Métrica          | Tiempo (ms) |
-----
| Secuencial       | 0,0032      |
| Paralelo         | 0,1604      |
| Aceleración      | 0,0200      |
-----

Comparación de las funciones secuenciales y paralelas para el tamaño de la finca: 8 (Iteración 3)
-----
| Métrica          | Tiempo (ms) |
-----
| Secuencial       | 0,0041      |
| Paralelo         | 0,1543      |
| Aceleración      | 0,0266      |
-----

Comparación de las funciones secuenciales y paralelas para el tamaño de la finca: 8 (Iteración 4)
-----
| Métrica          | Tiempo (ms) |
-----
| Secuencial       | 0,0041      |
| Paralelo         | 0,1241      |
| Aceleración      | 0,0330      |
-----

Comparación de las funciones secuenciales y paralelas para el tamaño de la finca: 8 (Iteración 5)
-----
| Métrica          | Tiempo (ms) |
-----
| Secuencial       | 0,0041      |
| Paralelo         | 0,0671      |
| Aceleración      | 0,0611      |
-----

```

Figura 114: Prueba de tamaño 8 Iteración 1 - 5

```

Comparaci3n de las funciones secuenciales y paralelas para el tamao de la finca: 8 (Iteraci3n 6)
-----
| M3trica          | Tiempo (ms)
-----
| Secuencial       | 0,0034
| Paralelo         | 0,0856
| Aceleraci3n      | 0,0397
-----

Comparaci3n de las funciones secuenciales y paralelas para el tamao de la finca: 8 (Iteraci3n 7)
-----
| M3trica          | Tiempo (ms)
-----
| Secuencial       | 0,0041
| Paralelo         | 0,0855
| Aceleraci3n      | 0,0480
-----

Comparaci3n de las funciones secuenciales y paralelas para el tamao de la finca: 8 (Iteraci3n 8)
-----
| M3trica          | Tiempo (ms)
-----
| Secuencial       | 0,0055
| Paralelo         | 0,0805
| Aceleraci3n      | 0,0683
-----

Comparaci3n de las funciones secuenciales y paralelas para el tamao de la finca: 8 (Iteraci3n 9)
-----
| M3trica          | Tiempo (ms)
-----
| Secuencial       | 0,0025
| Paralelo         | 0,0683
| Aceleraci3n      | 0,0366
-----

Comparaci3n de las funciones secuenciales y paralelas para el tamao de la finca: 8 (Iteraci3n 10)
-----
| M3trica          | Tiempo (ms)
-----
| Secuencial       | 0,0025
| Paralelo         | 0,0752
| Aceleraci3n      | 0,0332
-----

BUILD SUCCESSFUL in 2m 33s
2 actionable tasks: 1 executed, 1 up-to-date

```

Figura 115: Prueba de tamao 8 Iteracion 6 - 10

**Conclusi3n** Los resultados demuestran que la paralelizaci3n mejora significativamente el rendimiento, especialmente para tamaos de entrada grandes. Sin embargo, las ganancias observadas se ven limitadas por la fracci3n no paralelizable del programa, como lo predice la ley de Amdahl. El an3lisis emp3rico valida la efectividad del enfoque adoptado y resalta la importancia de optimizar tanto las partes paralelizables como las secuenciales.

## 2.2. Función para calcular el costo de movilidad paralela (costoRiegoMovilidadPar)

La función `costoMovilidadPar` implementa un cálculo paralelo para determinar el costo total de movilidad en una programación de riego.

**Estrategia de paralelización:** La función `costoMovilidadPar` realiza el cálculo del costo de movilidad utilizando ejecución paralela. El enfoque empleado consiste en dividir el trabajo en subproblemas independientes, lo que permite que los cálculos se ejecuten en múltiples hilos.

1. **Recorrido de la programación de riego:** La programación de riego, representada por el vector `pi`, se recorre desde el índice 0 hasta `pi.length - 1`.
2. **Uso de par:** El rango (0 until `pi.length - 1`) se convierte en un rango paralelo usando `.par`. Esto implica que las operaciones de cálculo que involucran las distancias entre los tablonos se ejecutarán en paralelo.
3. **Cálculo paralelo de distancias:** Para cada índice  $j$ , se calcula la distancia entre los tablonos  $pi(j)$  y  $pi(j + 1)$  utilizando la función de distancia  $d(pi(j), pi(j + 1))$ . Este cálculo se realiza de manera independiente para cada par de tablonos, lo que lo hace fácilmente paralelizable.
4. **Suma paralela:** Después de calcular todas las distancias en paralelo, la función `sum` se encarga de calcular la suma total de las distancias. Aunque el cálculo de la suma es secuencial, la mayor parte del trabajo se realiza en paralelo.

**Evaluación de las ganancias en rendimiento según la ley de Amdahl:** La ley de Amdahl es una fórmula que se utiliza para estimar las ganancias en el rendimiento cuando una parte del programa se paraleliza. La ley de Amdahl se expresa como:

$$S_{max} = \frac{1}{(1 - p) + \frac{p}{n}}$$

Donde:

- $S_{max}$  es la aceleración máxima alcanzable.
- $p$  es la fracción del tiempo de ejecución que puede ser paralelizada.
- $n$  es el número de procesadores disponibles.

La función `costoMovilidadPar` se beneficia de la paralelización principalmente en el cálculo de las distancias entre los tablonos, lo cual es independiente y puede ejecutarse en paralelo para cada par de tablonos. Este cálculo de distancias es la porción paralelizable de la función.

**Estimación de la fracción paralelizable  $p$ :** La fracción paralelizable  $p$  depende de la parte del cálculo que se puede ejecutar en paralelo. En este caso, el cálculo de las distancias entre los tablonos es completamente independiente, lo que implica que toda la operación de cálculo de distancias es paralelizable. Por lo tanto, podemos estimar que  $p$  es cercano a 1 (o 100% paralelizable).

La suma de las distancias, por otro lado, es una operación secuencial que no puede paralelizarse, lo que limita la ganancia de rendimiento.

**Ganancia teórica de rendimiento** Si asumimos que casi todo el trabajo puede paralelizarse (es decir,  $p \approx 1$ ), la aceleración máxima que se puede alcanzar según la ley de Amdahl para  $n$  procesadores es:

$$S_{max} \approx \frac{1}{(1 - 1) + \frac{1}{n}} = n$$

Esto indica que, en el mejor de los casos, la aceleración será proporcional al número de procesadores disponibles. Sin embargo, esta aceleración máxima solo se logra cuando casi toda la operación es paralelizable, lo que no es el caso debido a la suma secuencial.

**Impacto de la parte secuencial:** La parte secuencial del cálculo, que es la suma de las distancias, puede convertirse en un cuello de botella. Esto limita la ganancia de rendimiento, especialmente cuando el número de procesadores es alto.

En la práctica, el rendimiento mejorará en función del tamaño de la finca (es decir, el número de tablonos). Cuanto mayor sea el número de tablonos, mayor será la fracción paralelizable, lo que lleva a mayores ganancias en rendimiento. Sin embargo, el rendimiento se estabilizará cuando el número de procesadores sea suficiente para manejar todos los subproblemas, ya que el tiempo de ejecución de la parte secuencial (la suma) se convierte en un factor limitante.

**Conclusión sobre las ganancias de rendimiento:** En resumen, la paralelización de la función `costoMovilidadPar` ofrece beneficios de rendimiento que dependen en gran medida del tamaño del problema y del número de procesadores disponibles. Según la ley de Amdahl, la ganancia de rendimiento se ve limitada por la parte secuencial del programa (la suma). A medida que el número de tablonos aumenta, la ganancia en rendimiento de la paralelización se vuelve más significativa, pero el impacto de la suma secuencial puede seguir siendo un factor limitante.

```

..... Iniciando benchmarking de Riego Optimo .....
----- Benchmarking: Costos de Riego y Movilidad -----
===== Tamaño finca: 4 =====

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 4 (Iteraci n 1)
Unable to create a system terminal
-----
| M trica          | Tiempo (ms)
-----
| Secuencial        | 0,0348
| Paralelo          | 0,4241
| Aceleraci n       | 0,0821
-----

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 4 (Iteraci n 2)
-----
| M trica          | Tiempo (ms)
-----
| Secuencial        | 0,0100
| Paralelo          | 0,2892
| Aceleraci n       | 0,0346
-----

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 4 (Iteraci n 3)
-----
| M trica          | Tiempo (ms)
-----
| Secuencial        | 0,0172
| Paralelo          | 0,4003
| Aceleraci n       | 0,0430
-----

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 4 (Iteraci n 4)
-----
| M trica          | Tiempo (ms)
-----
| Secuencial        | 0,0131
| Paralelo          | 0,2705
| Aceleraci n       | 0,0484
-----

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 4 (Iteraci n 5)
-----
| M trica          | Tiempo (ms)
-----
| Secuencial        | 0,0122
| Paralelo          | 0,3034
| Aceleraci n       | 0,0402
-----

```

Figura 116: Prueba tama o 4 Iteraci n 1 - 5

```

Comparaci3n de las funciones Costo movilidad secuenciales y paralelas para el tamao de la finca: 4 (Iteraci3n 6)
-----
| M3trica          | Tiempo (ms)
-----
| Secuencial       | 0,0125
| Paralelo         | 0,3269
| Aceleraci3n     | 0,0382
-----

Comparaci3n de las funciones Costo movilidad secuenciales y paralelas para el tamao de la finca: 4 (Iteraci3n 7)
-----
| M3trica          | Tiempo (ms)
-----
| Secuencial       | 0,0082
| Paralelo         | 0,2443
| Aceleraci3n     | 0,0336
-----

Comparaci3n de las funciones Costo movilidad secuenciales y paralelas para el tamao de la finca: 4 (Iteraci3n 8)
-----
| M3trica          | Tiempo (ms)
-----
| Secuencial       | 0,0068
| Paralelo         | 0,2814
| Aceleraci3n     | 0,0242
-----

Comparaci3n de las funciones Costo movilidad secuenciales y paralelas para el tamao de la finca: 4 (Iteraci3n 9)
-----
| M3trica          | Tiempo (ms)
-----
| Secuencial       | 0,0066
| Paralelo         | 0,2517
| Aceleraci3n     | 0,0262
-----

Comparaci3n de las funciones Costo movilidad secuenciales y paralelas para el tamao de la finca: 4 (Iteraci3n 10)
-----
| M3trica          | Tiempo (ms)
-----
| Secuencial       | 0,0048
| Paralelo         | 0,2326
| Aceleraci3n     | 0,0206
-----

```

Figura 117: Prueba de tamao 4 Iteraci3n 6 - 10

```

===== Tamaño finca: 5 =====

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 5 (Iteraci n 1)

-----
| M trica          | Tiempo (ms)
-----
| Secuencial        | 0,0059
| Paralelo          | 0,1858
| Aceleraci n       | 0,0318
-----

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 5 (Iteraci n 2)

-----
| M trica          | Tiempo (ms)
-----
| Secuencial        | 0,0078
| Paralelo          | 0,2160
| Aceleraci n       | 0,0361
-----

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 5 (Iteraci n 3)

-----
| M trica          | Tiempo (ms)
-----
| Secuencial        | 0,0064
| Paralelo          | 0,1652
| Aceleraci n       | 0,0387
-----

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 5 (Iteraci n 4)

-----
| M trica          | Tiempo (ms)
-----
| Secuencial        | 0,0058
| Paralelo          | 0,1462
| Aceleraci n       | 0,0397
-----

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 5 (Iteraci n 5)

-----
| M trica          | Tiempo (ms)
-----
| Secuencial        | 0,0053
| Paralelo          | 0,1291
| Aceleraci n       | 0,0411
-----

```

Figura 118: Prueba de tama o 5 Iteraci n 1 - 5

```

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 5 (Iteraci n 6)
-----
| M trica          | Tiempo (ms)
-----
| Secuencial        | 0,0046
| Paralelo          | 0,1461
| Aceleraci n       | 0,0315
-----

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 5 (Iteraci n 7)
-----
| M trica          | Tiempo (ms)
-----
| Secuencial        | 0,0048
| Paralelo          | 0,0886
| Aceleraci n       | 0,0542
-----

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 5 (Iteraci n 8)
-----
| M trica          | Tiempo (ms)
-----
| Secuencial        | 0,0055
| Paralelo          | 0,1740
| Aceleraci n       | 0,0316
-----

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 5 (Iteraci n 9)
-----
| M trica          | Tiempo (ms)
-----
| Secuencial        | 0,0049
| Paralelo          | 0,1337
| Aceleraci n       | 0,0366
-----

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 5 (Iteraci n 10)
-----
| M trica          | Tiempo (ms)
-----
| Secuencial        | 0,0055
| Paralelo          | 0,1720
| Aceleraci n       | 0,0320
-----

```

Figura 119: Prueba de tama o 5 Iteraci n 6 - 10

```

===== Tamaño finca: 7 =====

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 7 (Iteraci n 1)
-----
| M trica          | Tiempo (ms)
-----
| Secuencial        | 0,0036
| Paralelo          | 0,1449
| Aceleraci n       | 0,0248
-----

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 7 (Iteraci n 2)
-----
| M trica          | Tiempo (ms)
-----
| Secuencial        | 0,0035
| Paralelo          | 0,1866
| Aceleraci n       | 0,0188
-----

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 7 (Iteraci n 3)
-----
| M trica          | Tiempo (ms)
-----
| Secuencial        | 0,0025
| Paralelo          | 0,1414
| Aceleraci n       | 0,0177
-----

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 7 (Iteraci n 4)
-----
| M trica          | Tiempo (ms)
-----
| Secuencial        | 0,0025
| Paralelo          | 0,1578
| Aceleraci n       | 0,0158
-----

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 7 (Iteraci n 5)
-----
| M trica          | Tiempo (ms)
-----
| Secuencial        | 0,0047
| Paralelo          | 0,1562
| Aceleraci n       | 0,0301
-----

```

Figura 120: Prueba de tama o 7 Iteraci n 1 - 5



```

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 7 (Iteraci n 6)
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial        | 0,0036      |
| Paralelo          | 0,1088      |
| Aceleraci n       | 0,0331      |
-----

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 7 (Iteraci n 7)
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial        | 0,0033      |
| Paralelo          | 0,1225      |
| Aceleraci n       | 0,0269      |
-----

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 7 (Iteraci n 8)
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial        | 0,0031      |
| Paralelo          | 0,1539      |
| Aceleraci n       | 0,0201      |
-----

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 7 (Iteraci n 9)
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial        | 0,0046      |
| Paralelo          | 0,1596      |
| Aceleraci n       | 0,0288      |
-----

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 7 (Iteraci n 10)
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial        | 0,0019      |
| Paralelo          | 0,1445      |
| Aceleraci n       | 0,0131      |
-----

```

Figura 121: Prueba de tama o 7 Iteraci n 6 - 10

```

===== Tamaño finca: 8 =====

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 8 (Iteraci n 1)
-----
| M trica          | Tiempo (ms)
-----
| Secuencial        | 0,0021
| Paralelo          | 0,1654
| Aceleraci n       | 0,0127
-----

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 8 (Iteraci n 2)
-----
| M trica          | Tiempo (ms)
-----
| Secuencial        | 0,0015
| Paralelo          | 0,1355
| Aceleraci n       | 0,0111
-----

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 8 (Iteraci n 3)
-----
| M trica          | Tiempo (ms)
-----
| Secuencial        | 0,0023
| Paralelo          | 0,1543
| Aceleraci n       | 0,0149
-----

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 8 (Iteraci n 4)
-----
| M trica          | Tiempo (ms)
-----
| Secuencial        | 0,0028
| Paralelo          | 0,0656
| Aceleraci n       | 0,0427
-----

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 8 (Iteraci n 5)
-----
| M trica          | Tiempo (ms)
-----
| Secuencial        | 0,0017
| Paralelo          | 0,1512
| Aceleraci n       | 0,0112
-----

```

Figura 122: Prueba de tama o 8 Iteraci n 1 - 5

Comparaci�n de las funciones Costo movilidad secuenciales y paralelas para el tama�o de la finca: 8 (Iteraci�n 6)	
M�trica	Tiempo (ms)
Secuencial	0,0020
Paralelo	0,1396
Aceleraci�n	0,0143
Comparaci�n de las funciones Costo movilidad secuenciales y paralelas para el tama�o de la finca: 8 (Iteraci�n 7)	
M�trica	Tiempo (ms)
Secuencial	0,0021
Paralelo	0,1233
Aceleraci�n	0,0170
Comparaci�n de las funciones Costo movilidad secuenciales y paralelas para el tama�o de la finca: 8 (Iteraci�n 8)	
M�trica	Tiempo (ms)
Secuencial	0,0022
Paralelo	0,1320
Aceleraci�n	0,0167
Comparaci�n de las funciones Costo movilidad secuenciales y paralelas para el tama�o de la finca: 8 (Iteraci�n 9)	
M�trica	Tiempo (ms)
Secuencial	0,0015
Paralelo	0,1503
Aceleraci�n	0,0100
Comparaci�n de las funciones Costo movilidad secuenciales y paralelas para el tama�o de la finca: 8 (Iteraci�n 10)	
M�trica	Tiempo (ms)
Secuencial	0,0013
Paralelo	0,1236
Aceleraci�n	0,0105

Figura 123: Prueba de tama o 8 Iteraci n 6 - 10

Comparaci�n de las funciones Costo movilidad secuenciales y paralelas para el tama�o de la finca: 8 (Iteraci�n 6)	
M�trica	Tiempo (ms)
Secuencial	0,0020
Paralelo	0,1396
Aceleraci�n	0,0143
Comparaci�n de las funciones Costo movilidad secuenciales y paralelas para el tama�o de la finca: 8 (Iteraci�n 7)	
M�trica	Tiempo (ms)
Secuencial	0,0021
Paralelo	0,1233
Aceleraci�n	0,0170
Comparaci�n de las funciones Costo movilidad secuenciales y paralelas para el tama�o de la finca: 8 (Iteraci�n 8)	
M�trica	Tiempo (ms)
Secuencial	0,0022
Paralelo	0,1320
Aceleraci�n	0,0167
Comparaci�n de las funciones Costo movilidad secuenciales y paralelas para el tama�o de la finca: 8 (Iteraci�n 9)	
M�trica	Tiempo (ms)
Secuencial	0,0015
Paralelo	0,1503
Aceleraci�n	0,0100
Comparaci�n de las funciones Costo movilidad secuenciales y paralelas para el tama�o de la finca: 8 (Iteraci�n 10)	
M�trica	Tiempo (ms)
Secuencial	0,0013
Paralelo	0,1236
Aceleraci�n	0,0105

Figura 124: Prueba de tama o 8 Iteraci n 6 - 10

```

===== Tamaño finca: 9 =====

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 9 (Iteraci n 1)
-----
| M trica                | Tiempo (ms)
-----
| Secuencial              | 0,0017
| Paralelo                | 0,1055
| Aceleraci n            | 0,0161
-----

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 9 (Iteraci n 2)
-----
| M trica                | Tiempo (ms)
-----
| Secuencial              | 0,0013
| Paralelo                | 0,1115
| Aceleraci n            | 0,0117
-----

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 9 (Iteraci n 3)
-----
| M trica                | Tiempo (ms)
-----
| Secuencial              | 0,0016
| Paralelo                | 0,0904
| Aceleraci n            | 0,0177
-----

Comparaci n de las funciones Costo movilidad secuenciales y paralelas para el tama o de la finca: 9 (Iteraci n 5)
-----
| M trica                | Tiempo (ms)
-----
| Secuencial              | 0,0012
| Paralelo                | 0,0719
| Aceleraci n            | 0,0167
-----

```

Figura 125: Prueba de tama o 9 Iteraci n 1 - 5

```

Comparaci3n de las funciones Costo movilidad secuenciales y paralelas para el tamao de la finca: 9 (Iteraci3n 6)
-----
| M3trica          | Tiempo (ms)
-----
| Secuencial       | 0,0022
| Paralelo         | 0,0682
| Aceleraci3n     | 0,0323
-----

Comparaci3n de las funciones Costo movilidad secuenciales y paralelas para el tamao de la finca: 9 (Iteraci3n 7)
-----
| M3trica          | Tiempo (ms)
-----
| Secuencial       | 0,0020
| Paralelo         | 0,0661
| Aceleraci3n     | 0,0232
-----

Comparaci3n de las funciones Costo movilidad secuenciales y paralelas para el tamao de la finca: 9 (Iteraci3n 8)
-----
| M3trica          | Tiempo (ms)
-----
| Secuencial       | 0,0023
| Paralelo         | 0,0760
| Aceleraci3n     | 0,0303
-----

Comparaci3n de las funciones Costo movilidad secuenciales y paralelas para el tamao de la finca: 9 (Iteraci3n 9)
-----
| M3trica          | Tiempo (ms)
-----
| Secuencial       | 0,0048
| Paralelo         | 0,0640
| Aceleraci3n     | 0,0750
-----

Comparaci3n de las funciones Costo movilidad secuenciales y paralelas para el tamao de la finca: 9 (Iteraci3n 10)
-----
| M3trica          | Tiempo (ms)
-----
| Secuencial       | 0,0020
| Paralelo         | 0,0536
| Aceleraci3n     | 0,0373
-----

BUILD SUCCESSFUL in 1m 33s
2 actionable tasks: 2 executed

```

Figura 126: Prueba de tamao 9 Iteraci3n de 6 - 10

Los resultados del benchmarking muestran c3mo la paralelizaci3n mejora el rendimiento al distribuir las tareas entre varios procesadores, especialmente cuando el tamao del problema (en este caso, el n3mero de tablonces de la finca) es grande. Sin embargo, estas mejoras tienen un l3mite debido a la parte secuencial del c3lculo (la suma de las distancias), que no se puede paralelizar.

La paralelizaci3n es efectiva, pero su impacto depende tanto del tamao del problema como de la proporci3n de operaciones no paralelizables.

### 2.3. Funci3n para generar programaciones de riego paralelas (generarProgramacionesRiegoPar)

La funci3n `generarProgramacionesRiegoPar` genera todas las posibles permutaciones de riego para una finca (`f`) de manera paralela. Esto se utiliza como entrada para c3lculos posteriores, como la evaluaci3n de costos y la selecci3n de la programaci3n 3ptima.

```

1 def generarProgramacionesRiegoPar(f: Finca): Vector[ProgRiego] = {
2   val indices = (0 until f.length).toVector
3   indices.permutations.toVector.par.toVector
4 }

```

**Estrategia de paralelización:** La paralelización de esta función se basa en las siguientes etapas:

1. **Generación del rango de índices:** Se genera un vector que representa los índices de los tableros de la finca, es decir, los números del 0 al `f.length - 1`. Este vector representa los posibles elementos que se permutarán.
2. **Generación de permutaciones:** Utilizando la función `indices.permutations`, se generan todas las permutaciones posibles del vector de índices. Esto produce un iterador que recorre todas las combinaciones posibles de órdenes en los que se pueden regar los tableros.
3. **Conversión a procesamiento paralelo:** Las permutaciones se convierten a un vector y posteriormente se paralelizan utilizando `.par`. Esto divide las permutaciones en subconjuntos, asignando cada subconjunto a un hilo para ser procesado de forma independiente.
4. **Conversión final a vector:** Después de completar el procesamiento paralelo, los resultados se convierten nuevamente a un vector secuencial utilizando `.toVector`.

**Evaluación de las ganancias en rendimiento según la ley de Amdahl:** La generación de permutaciones es una tarea intensiva en cómputo, especialmente para fincas con un gran número de tableros, ya que el número de permutaciones es  $n!$  (factorial de  $n$ ). Esto hace que la paralelización sea particularmente beneficiosa para este tipo de problemas.

La ley de Amdahl se expresa como:

$$S_{max} = \frac{1}{(1 - p) + \frac{p}{n}}$$

Donde:

- $S_{max}$ : Ganancia máxima en el rendimiento.
- $p$ : Fracción del programa paralelizable.
- $n$ : Número de procesadores disponibles.

En esta función, prácticamente todo el trabajo (la generación de permutaciones) es paralelizable, lo que implica que  $p \approx 1$ . Sin embargo, la etapa secuencial (como la conversión final a vector) puede limitar las ganancias de rendimiento.

**Estimación de la fracción paralelizable  $p$ :** La generación de permutaciones y su procesamiento paralelo representan la mayor parte del tiempo de ejecución, por lo que podemos estimar que  $p > 0,95$ . La conversión final a un vector secuencial introduce una pequeña parte no paralelizable.

**Ganancia teórica de rendimiento:** Para  $p = 0,95$  y distintos valores de  $n$ , las ganancias teóricas de rendimiento son:

$$S_{max} = \frac{1}{(1 - 0,95) + \frac{0,95}{n}}$$

- Para  $n = 2$ :  $S_{max} = 1,90$ .
- Para  $n = 4$ :  $S_{max} = 3,59$ .
- Para  $n = 8$ :  $S_{max} = 6,55$ .

Esto demuestra que la escalabilidad de esta función es alta, aunque limitada por la pequeña fracción secuencial.

**Resultados experimentales:** Se realizaron pruebas para medir el tiempo de ejecución de `generarProgramacionesRiego` en fincas de distintos tamaños (número de tablonos). Los resultados muestran cómo la paralelización mejora el rendimiento para tamaños mayores.

```
..... Iniciando benchmarking de Riego Optimo .....
----- Benchmarking: Generaci n de Programaciones de Riego -----

===== Tama o finca: 4 =====

Comparaci n de Programaciones (Iteraci n 1):
Unable to create a system terminal
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial       | 0,2183      |
| Paralelo         | 0,0780      |
| Aceleraci n      | 2,7987      |
-----

Comparaci n de Programaciones (Iteraci n 2):
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial       | 0,0681      |
| Paralelo         | 0,0673      |
| Aceleraci n      | 1,0119      |
-----

Comparaci n de Programaciones (Iteraci n 3):
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial       | 0,0882      |
| Paralelo         | 0,0614      |
| Aceleraci n      | 1,4365      |
-----

Comparaci n de Programaciones (Iteraci n 4):
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial       | 0,0676      |
| Paralelo         | 0,2430      |
| Aceleraci n      | 0,2782      |
-----

Comparaci n de Programaciones (Iteraci n 5):
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial       | 0,0434      |
| Paralelo         | 0,0466      |
| Aceleraci n      | 0,9313      |
-----
```

Figura 127: Prueba tama o 4 Iteraci n 1 - 5

```
Comparaci n de Programaciones (Iteraci n 6):
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial       | 0,1552      |
| Paralelo         | 0,0380      |
| Aceleraci n      | 4,0842      |
-----

Comparaci n de Programaciones (Iteraci n 7):
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial       | 0,0536      |
| Paralelo         | 0,0499      |
| Aceleraci n      | 1,0741      |
-----

Comparaci n de Programaciones (Iteraci n 8):
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial       | 0,0489      |
| Paralelo         | 0,1851      |
| Aceleraci n      | 0,2642      |
-----

Comparaci n de Programaciones (Iteraci n 9):
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial       | 0,0803      |
| Paralelo         | 0,0763      |
| Aceleraci n      | 1,0524      |
-----

Comparaci n de Programaciones (Iteraci n 10):
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial       | 0,0583      |
| Paralelo         | 0,0709      |
| Aceleraci n      | 0,8223      |
-----
```

Figura 128: Segunda imagen Iteraci n 6 - 10



```

===== Tamaño finca: 5 =====

Comparaci n de Programaciones (Iteraci n 1):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 0,1198
| Paralelo         | 0,1868
| Aceleraci n      | 0,6413
-----

Comparaci n de Programaciones (Iteraci n 2):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 0,1733
| Paralelo         | 0,1711
| Aceleraci n      | 1,0129
-----

Comparaci n de Programaciones (Iteraci n 3):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 0,1115
| Paralelo         | 0,1072
| Aceleraci n      | 1,0401
-----

Comparaci n de Programaciones (Iteraci n 4):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 0,1269
| Paralelo         | 0,1203
| Aceleraci n      | 1,0549
-----

Comparaci n de Programaciones (Iteraci n 5):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 0,2178
| Paralelo         | 0,1925
| Aceleraci n      | 1,1314
-----

```

Figura 129: Prueba tama o 5 Iteraci n 1 - 5

```

Comparaci n de Programaciones (Iteraci n 6):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 0,1614
| Paralelo         | 0,1606
| Aceleraci n      | 1,0050
-----

Comparaci n de Programaciones (Iteraci n 7):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 0,1441
| Paralelo         | 0,1440
| Aceleraci n      | 1,0007
-----

Comparaci n de Programaciones (Iteraci n 8):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 0,1728
| Paralelo         | 0,1684
| Aceleraci n      | 1,0261
-----

Comparaci n de Programaciones (Iteraci n 9):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 0,1523
| Paralelo         | 0,1519
| Aceleraci n      | 1,0026
-----

Comparaci n de Programaciones (Iteraci n 10):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 0,1135
| Paralelo         | 0,0978
| Aceleraci n      | 1,1605
-----

```

Figura 130: Segunda imagen Iteraci n 6 - 10

```

===== Tamaño finca: 6 =====

Comparaci n de Programaciones (Iteraci n 1):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 0,3115
| Paralelo         | 0,3731
| Aceleraci n      | 0,8349
-----

Comparaci n de Programaciones (Iteraci n 2):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 0,4277
| Paralelo         | 0,3531
| Aceleraci n      | 1,2113
-----

Comparaci n de Programaciones (Iteraci n 3):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 0,3625
| Paralelo         | 0,3528
| Aceleraci n      | 1,0275
-----

Comparaci n de Programaciones (Iteraci n 4):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 0,1383
| Paralelo         | 0,1336
| Aceleraci n      | 1,0352
-----

Comparaci n de Programaciones (Iteraci n 5):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 0,1309
| Paralelo         | 0,1452
| Aceleraci n      | 0,9015
-----

```

Figura 131: Prueba de tama o 6 Iteraci n 1 - 5

```

Comparaci n de Programaciones (Iteraci n 6):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 0,1300
| Paralelo         | 0,1343
| Aceleraci n      | 0,9680
-----

Comparaci n de Programaciones (Iteraci n 7):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 0,1353
| Paralelo         | 0,1734
| Aceleraci n      | 0,7803
-----

Comparaci n de Programaciones (Iteraci n 8):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 0,3137
| Paralelo         | 0,2327
| Aceleraci n      | 1,3481
-----

Comparaci n de Programaciones (Iteraci n 9):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 0,2142
| Paralelo         | 0,1337
| Aceleraci n      | 1,6021
-----

Comparaci n de Programaciones (Iteraci n 10):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 0,2590
| Paralelo         | 0,2709
| Aceleraci n      | 0,9561
-----

```

Figura 132: Segunda imagen Iteraci n 6 - 10

```

===== Tamaño finca: 8 =====

Comparaci n de Programaciones (Iteraci n 1):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 6,8347
| Paralelo         | 6,8623
| Aceleraci n      | 0,9960
-----

Comparaci n de Programaciones (Iteraci n 2):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 8,7298
| Paralelo         | 6,8300
| Aceleraci n      | 1,2782
-----

Comparaci n de Programaciones (Iteraci n 3):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 6,8189
| Paralelo         | 6,8160
| Aceleraci n      | 1,0004
-----

Comparaci n de Programaciones (Iteraci n 4):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 7,0410
| Paralelo         | 6,8027
| Aceleraci n      | 1,0350
-----

Comparaci n de Programaciones (Iteraci n 5):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 8,0826
| Paralelo         | 7,0734
| Aceleraci n      | 1,1427
-----

```

Figura 133: Prueba de tama o 8 Iteraci n 1 - 5

```

Comparaci n de Programaciones (Iteraci n 6):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 7,0789
| Paralelo         | 7,0907
| Aceleraci n      | 0,9983
-----

Comparaci n de Programaciones (Iteraci n 7):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 6,9553
| Paralelo         | 7,0138
| Aceleraci n      | 0,9917
-----

Comparaci n de Programaciones (Iteraci n 8):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 6,7788
| Paralelo         | 6,9441
| Aceleraci n      | 0,9762
-----

Comparaci n de Programaciones (Iteraci n 9):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 6,8348
| Paralelo         | 7,4750
| Aceleraci n      | 0,9144
-----

Comparaci n de Programaciones (Iteraci n 10):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 7,4396
| Paralelo         | 12,9699
| Aceleraci n      | 0,5736
-----

```

Figura 134: Segunda imagen Iteraci n 6 - 10

```

===== Tamaño finca: 9 =====

Comparaci n de Programaciones (Iteraci n 1):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 66,8955
| Paralelo         | 79,4003
| Aceleraci n      | 0,8425
-----

Comparaci n de Programaciones (Iteraci n 2):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 77,0635
| Paralelo         | 69,7710
| Aceleraci n      | 1,1045
-----

Comparaci n de Programaciones (Iteraci n 3):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 64,7566
| Paralelo         | 64,6671
| Aceleraci n      | 1,0014
-----

Comparaci n de Programaciones (Iteraci n 4):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 64,2060
| Paralelo         | 64,4209
| Aceleraci n      | 0,9967
-----

Comparaci n de Programaciones (Iteraci n 5):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 64,0256
| Paralelo         | 63,9555
| Aceleraci n      | 1,0011
-----

```

Figura 135: Prueba tama o 9 Iteraci n 1 - 5

```

Comparaci n de Programaciones (Iteraci n 6):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 66,3162
| Paralelo         | 66,8084
| Aceleraci n      | 0,9926
-----

Comparaci n de Programaciones (Iteraci n 7):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 66,5476
| Paralelo         | 66,1820
| Aceleraci n      | 1,0055
-----

Comparaci n de Programaciones (Iteraci n 8):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 67,2896
| Paralelo         | 64,5083
| Aceleraci n      | 1,0431
-----

Comparaci n de Programaciones (Iteraci n 9):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 66,0552
| Paralelo         | 66,5627
| Aceleraci n      | 0,9924
-----

Comparaci n de Programaciones (Iteraci n 10):
-----
| M trica          | Tiempo (ms)
-----
| Secuencial       | 64,7588
| Paralelo         | 63,9614
| Aceleraci n      | 1,0125
-----

BUILD SUCCESSFUL in 1m 18s
2 actionable tasks: 2 executed

```

Figura 136: Segunda imagen Iteraci n 6 - 10

**Conclusi n sobre las ganancias de rendimiento:** La funci n `generarProgramacionesRiegoPar` es altamente paralelizable debido a la independencia en la generaci n de permutaciones. Las pruebas muestran que la paralelizaci n mejora significativamente el tiempo de ejecuci n a medida que aumenta el tama o del problema (n mero de tablonos). Sin embargo, el impacto de la etapa secuencial (como la conversi n final) y la sobrecarga asociada con la paralelizaci n pueden limitar las ganancias, espe-

cialmente para tamaños de finca pequeños.

En general, esta función aprovecha eficientemente los procesadores disponibles, mostrando una escalabilidad alta y siendo adecuada para problemas con gran carga computacional.

## 2.4. Función para calcular la programación de riego óptima paralela (ProgramacionRiegoOptimoPar)

La función `ProgramacionRiegoOptimoPar` implementa un cálculo paralelo para determinar la programación de riego óptima y su costo total, utilizando técnicas de paralelización para optimizar el rendimiento en sistemas con múltiples procesadores.

**Estrategia de paralelización:** La función realiza el cálculo de la programación óptima de manera paralela mediante las siguientes estrategias:

1. **Generación de programaciones de riego:** La función `generarProgramacionesRiegoPar` genera todas las posibles permutaciones de riego para la finca de manera paralela. Esta operación es altamente paralelizable, ya que las permutaciones se generan de forma independiente.
2. **Cálculo paralelo de costos:** Una vez generadas las programaciones, se calcula el costo total (riego + movilidad) de manera paralela utilizando `par.map`, dividiendo las programaciones entre múltiples hilos para evaluar sus costos de manera independiente.
3. **Selección del costo mínimo:** Después de calcular los costos en paralelo, se utiliza la función `minBy` para seleccionar la programación con el menor costo total. Esta operación es secuencial.

**Evaluación de las ganancias en rendimiento según la ley de Amdahl:** La ley de Amdahl se utiliza para evaluar las ganancias en rendimiento al paralelizar partes del programa. La fórmula es:

$$S_{max} = \frac{1}{(1 - p) + \frac{p}{n}}$$

Donde:

- $S_{max}$ : Ganancia máxima en el rendimiento.
- $p$ : Fracción del programa paralelizable.
- $n$ : Número de procesadores disponibles.

En esta función, el cálculo de costos (`par.map`) es altamente paralelizable y constituye la mayor parte del tiempo de ejecución. Por otro lado, las operaciones de generación de permutaciones y selección del costo mínimo (`minBy`) tienen componentes secuenciales que limitan la ganancia total.

**Estimación de la fracción paralelizable  $p$ :** En `ProgramacionRiegoOptimoPar`, las etapas paralelizables incluyen:

- Generación de programaciones ( $p_1$ ).
- Cálculo de costos para cada programación ( $p_2$ ).

La fracción paralelizable total puede estimarse como:

$$p = \frac{t_{par}}{t_{total}} = \frac{t_{generación} + t_{cálculo de costos}}{t_{total}}$$

Donde  $t_{generación}$  y  $t_{cálculo de costos}$  representan los tiempos asociados a las partes paralelizadas. Dado que el cálculo de costos es la parte más significativa, se puede aproximar que  $p \approx 0,8 - 0,9$ .

**Ganancia teórica de rendimiento** Para un  $p = 0,9$  y distintos valores de  $n$ , las ganancias teóricas de rendimiento son:

$$S_{max} = \frac{1}{(1 - 0,9) + \frac{0,9}{n}}$$

- Para  $n = 2$ :  $S_{max} = 1,82$ .
- Para  $n = 4$ :  $S_{max} = 3,27$ .
- Para  $n = 8$ :  $S_{max} = 5,63$ .

Esto demuestra que las ganancias en rendimiento aumentan con el número de procesadores, pero disminuyen a medida que el componente secuencial del programa se vuelve dominante.

**Impacto de la parte secuencial:** La parte secuencial incluye:

- La selección de la programación con costo mínimo (`minBy`).
- La coordinación entre hilos, que introduce una sobrecarga adicional.

Estas operaciones limitan la escalabilidad del programa, especialmente para un número muy alto de procesadores ( $n > 8$ ).

**Resultados experimentales:** Las pruebas realizadas midieron el tiempo de ejecución de `ProgramacionRiegoOptimoPar` para fincas con distintos tamaños (número de tablones). Los resultados se muestran en las siguientes figuras.

```

..... Iniciando benchmarking de Riego Optimo .....
----- Benchmarking: Programaci3n de Riego 3ptima -----
===== Tama1o finca: 4 =====
Comparaci3n de Optimizaciones de riego (Iteraci3n 1):
Unable to create a system terminal
-----
| M3trica          | Tiempo (ms)
-----
| Secuencial       | 0,3159
| Paralelo         | 1,0962
| Aceleraci3n      | 0,2882
-----

Comparaci3n de Optimizaciones de riego (Iteraci3n 2):
-----
| M3trica          | Tiempo (ms)
-----
| Secuencial       | 0,2954
| Paralelo         | 0,8033
| Aceleraci3n      | 0,3677
-----

Comparaci3n de Optimizaciones de riego (Iteraci3n 3):
-----
| M3trica          | Tiempo (ms)
-----
| Secuencial       | 0,1330
| Paralelo         | 0,8210
| Aceleraci3n      | 0,1620
-----

Comparaci3n de Optimizaciones de riego (Iteraci3n 4):
-----
| M3trica          | Tiempo (ms)
-----
| Secuencial       | 0,0987
| Paralelo         | 0,9559
| Aceleraci3n      | 0,1033
-----

Comparaci3n de Optimizaciones de riego (Iteraci3n 5):
-----
| M3trica          | Tiempo (ms)
-----
| Secuencial       | 0,0864
| Paralelo         | 0,7897
| Aceleraci3n      | 0,1094
-----

```

Figura 137: Prueba de tama1o 4 Iteraci3n 1 - 5

```

Comparaci3n de Optimizaciones de riego (Iteraci3n 6):
-----
| M3trica          | Tiempo (ms)
-----
| Secuencial       | 0,8405
| Paralelo         | 0,5497
| Aceleraci3n      | 1,5290
-----

Comparaci3n de Optimizaciones de riego (Iteraci3n 7):
-----
| M3trica          | Tiempo (ms)
-----
| Secuencial       | 0,0973
| Paralelo         | 1,7423
| Aceleraci3n      | 0,0558
-----

Comparaci3n de Optimizaciones de riego (Iteraci3n 8):
-----
| M3trica          | Tiempo (ms)
-----
| Secuencial       | 0,1206
| Paralelo         | 0,8224
| Aceleraci3n      | 0,1466
-----

Comparaci3n de Optimizaciones de riego (Iteraci3n 9):
-----
| M3trica          | Tiempo (ms)
-----
| Secuencial       | 0,1131
| Paralelo         | 0,5018
| Aceleraci3n      | 0,2254
-----

-
Comparaci3n de Optimizaciones de riego (Iteraci3n 10):
-----
| M3trica          | Tiempo (ms)
-----
| Secuencial       | 0,1123
| Paralelo         | 0,5733
| Aceleraci3n      | 0,1959
-----

```

Figura 138: Segunda imagen Iteraci3n 6 - 10

```

===== Tamaño finca: 5 =====

Comparaci n de Optimizaciones de riego (Iteraci n 1):
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial        | 0,6792      |
| Paralelo          | 3,2332      |
| Aceleraci n       | 0,2101      |
-----

Comparaci n de Optimizaciones de riego (Iteraci n 2):
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial        | 0,6540      |
| Paralelo          | 3,1611      |
| Aceleraci n       | 0,2069      |
-----

Comparaci n de Optimizaciones de riego (Iteraci n 3):
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial        | 0,6811      |
| Paralelo          | 2,7707      |
| Aceleraci n       | 0,2458      |
-----

Comparaci n de Optimizaciones de riego (Iteraci n 4):
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial        | 0,6750      |
| Paralelo          | 2,4396      |
| Aceleraci n       | 0,2767      |
-----

Comparaci n de Optimizaciones de riego (Iteraci n 5):
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial        | 0,4621      |
| Paralelo          | 2,4122      |
| Aceleraci n       | 0,1916      |
-----

```

Figura 139: Prueba de tama o 5 Iteraci n 1 - 5

```

Comparaci n de Optimizaciones de riego (Iteraci n 6):
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial        | 0,3747      |
| Paralelo          | 1,8579      |
| Aceleraci n       | 0,2017      |
-----

Comparaci n de Optimizaciones de riego (Iteraci n 7):
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial        | 0,4241      |
| Paralelo          | 0,9050      |
| Aceleraci n       | 0,4686      |
-----

Comparaci n de Optimizaciones de riego (Iteraci n 8):
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial        | 0,2176      |
| Paralelo          | 1,3357      |
| Aceleraci n       | 0,1629      |
-----

Comparaci n de Optimizaciones de riego (Iteraci n 9):
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial        | 1,0077      |
| Paralelo          | 0,7259      |
| Aceleraci n       | 1,3882      |
-----

Comparaci n de Optimizaciones de riego (Iteraci n 10):
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial        | 0,3513      |
| Paralelo          | 1,0583      |
| Aceleraci n       | 0,3319      |
-----

```

Figura 140: Segunda imagen Iteraci n 6 - 10



```

===== Tamaño finca: 6 =====

Comparaci n de Optimizaciones de riego (Iteraci n 1):
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial        | 2,4126      |
| Paralelo          | 2,7694      |
| Aceleraci n       | 0,8712      |
-----

Comparaci n de Optimizaciones de riego (Iteraci n 2):
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial        | 1,0770      |
| Paralelo          | 1,9829      |
| Aceleraci n       | 0,5431      |
-----

Comparaci n de Optimizaciones de riego (Iteraci n 3):
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial        | 1,3005      |
| Paralelo          | 1,5885      |
| Aceleraci n       | 0,8187      |
-----

Comparaci n de Optimizaciones de riego (Iteraci n 4):
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial        | 0,7381      |
| Paralelo          | 3,3446      |
| Aceleraci n       | 0,2207      |
-----

Comparaci n de Optimizaciones de riego (Iteraci n 5):
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial        | 0,7862      |
| Paralelo          | 4,7199      |
| Aceleraci n       | 0,1666      |
-----

```

Figura 141: Prueba de tama o 6 Iteraci n 1 - 5

```

Comparaci n de Optimizaciones de riego (Iteraci n 6):
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial        | 1,3901      |
| Paralelo          | 2,7414      |
| Aceleraci n       | 0,5071      |
-----

Comparaci n de Optimizaciones de riego (Iteraci n 7):
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial        | 0,7504      |
| Paralelo          | 5,1949      |
| Aceleraci n       | 0,1444      |
-----

Comparaci n de Optimizaciones de riego (Iteraci n 8):
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial        | 0,7735      |
| Paralelo          | 1,9964      |
| Aceleraci n       | 0,3874      |
-----

-

Comparaci n de Optimizaciones de riego (Iteraci n 9):
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial        | 0,9424      |
| Paralelo          | 2,3710      |
| Aceleraci n       | 0,3975      |
-----

Comparaci n de Optimizaciones de riego (Iteraci n 10):
-----
| M trica          | Tiempo (ms) |
-----
| Secuencial        | 0,7425      |
| Paralelo          | 1,8511      |
| Aceleraci n       | 0,4011      |
-----

```

Figura 142: Segunda imagen 6 - 10

```

===== Tamaño finca: 7 =====

----- Tamaño finca: 7 (Iteración 1)-----

Comparación de Optimizaciones de riego (Iteración 1):
-----
| Método          | Tiempo (ms)
-----
| Secuencial      | 5,7439
| Paralelo        | 10,5060
| Aceleración     | 0,5467
-----

Comparación de Optimizaciones de riego (Iteración 2):
-----
| Método          | Tiempo (ms)
-----
| Secuencial      | 6,1551
| Paralelo        | 11,9404
| Aceleración     | 0,5155
-----

Comparación de Optimizaciones de riego (Iteración 3):
-----
| Método          | Tiempo (ms)
-----
| Secuencial      | 5,3552
| Paralelo        | 14,8318
| Aceleración     | 0,3611
-----

Comparación de Optimizaciones de riego (Iteración 4):
-----
| Método          | Tiempo (ms)
-----
| Secuencial      | 5,8228
| Paralelo        | 12,0516
| Aceleración     | 0,4832
-----

Comparación de Optimizaciones de riego (Iteración 5):
-----
| Método          | Tiempo (ms)
-----
| Secuencial      | 5,2796
| Paralelo        | 16,1365
| Aceleración     | 0,3272
-----

```

Figura 143: Prueba de tamaño 7 Iteración 1 - 5

```

Comparación de Optimizaciones de riego (Iteración 6):
-----
| Método          | Tiempo (ms)
-----
| Secuencial      | 5,4903
| Paralelo        | 21,7153
| Aceleración     | 0,2528
-----

Comparación de Optimizaciones de riego (Iteración 7):
-----
| Método          | Tiempo (ms)
-----
| Secuencial      | 5,6673
| Paralelo        | 12,3713
| Aceleración     | 0,4581
-----

Comparación de Optimizaciones de riego (Iteración 8):
-----
| Método          | Tiempo (ms)
-----
| Secuencial      | 5,8591
| Paralelo        | 17,3836
| Aceleración     | 0,3370
-----

Comparación de Optimizaciones de riego (Iteración 9):
-----
| Método          | Tiempo (ms)
-----
| Secuencial      | 5,2752
| Paralelo        | 9,9061
| Aceleración     | 0,5325
-----

Comparación de Optimizaciones de riego (Iteración 10):
-----
| Método          | Tiempo (ms)
-----
| Secuencial      | 6,0439
| Paralelo        | 16,8804
| Aceleración     | 0,3580
-----

```

Figura 144: Segunda imagen Iteración 6 - 10

**Conclusión sobre las ganancias de rendimiento:** La paralelización en `ProgramacionRiegoOptimoPar` mejora significativamente el rendimiento al calcular costos para grandes cantidades de programaciones. Sin embargo, el impacto de la parte secuencial, como la selección del costo mínimo, limita las ganancias en entornos con muchos procesadores.

En general, el rendimiento mejora con el aumento del tamaño del problema (número de tablonos), ya que la fracción paralelizable aumenta. A pesar de esto, la ley de Amdahl impone un límite superior en las ganancias de rendimiento que se pueden alcanzar.

### 3. Análisis comparativo de las funciones

#### 3.1. Comparación de costos de riego (secuencial vs paralelo)

```

1 // Comparación de costos de riego (secuencial vs paralelo)
2 def compararCostos(

```

```

3      funcionSecuencial: (Finca, ProgRiego) => Int, // Funci n secuencial a evaluar
      (costo de riego) recibe finca y programaci n que son tipos de datos vector de
      tablonos y vector de turnos de riego y devuelve un entero
4      funcionParalela: (Finca, ProgRiego) => Int, // Funci n paralela a evaluar (
      costo de riego) recibe finca y programaci n que son tipos de datos vector de
      tablonos y vector de turnos de riego y devuelve un entero
5      nombreSecuencial: String, // Nombre descriptivo de la
      funci n secuencial
6      nombreParalela: String // Nombre descriptivo de la
      funci n paralela
7  )(finca: Finca, programacion: ProgRiego): Unit = {
8      // Mide el tiempo de ejecuci n de la funci n secuencial
9      val tiempoSecuencial = withWarmer(new Warmer.Default) measure {
      funcionSecuencial(finca, programacion) }
10
11      // Mide el tiempo de ejecuci n de la funci n paralela
12      val tiempoParalelo = withWarmer(new Warmer.Default) measure { funcionParalela(
      finca, programacion) }
13
14      // Calcula la aceleraci n de la funci n paralela con respecto a la
      secuencial
15      val aceleracion = tiempoSecuencial.value / tiempoParalelo.value
16
17      // Impresi n de resultados
18      println(f"\nTiempo $nombreSecuencial: ${tiempoSecuencial.value}%.4f ms") //
      %.4f imprime el n mero con 4 decimales
19      println(f"Tiempo $nombreParalela: ${tiempoParalelo.value}%.4f ms")
20      println(f"Aceleraci n: $aceleracion%.4f")
21  }
22

```

### 3.1.1. Comparaci3n funci3n secuencial y paralela del costo de Riego de Finca

#### Funci3n costoRiegoFinca Secuencial

```

1      val tiempoSecuencial = withWarmer(new Warmer.Default) measure {
      funcionSecuencial(finca, programacion) }

```

#### Funci3n costoRiegoFincaPar Paralela

```

1      val tiempoParalelo = withWarmer(new Warmer.Default) measure { funcionParalela(
      finca, programacion) }

```

**Presentaci3n de Resultados** Se realizaron pruebas para los siguientes tama1os de fincas:

Tama1o finca (tablonos)	Versi3n secuencial (ms)	Versi3n paralela (ms)	Aceleraci3n (%)
4	0,0104	0,2432	4,28
5	0,0084	0,2259	3,72
6	0,0052	0,3176	1,64
7	0,0085	0,1480	5,74
8	0,0024	0,1123	2,14

Cuadro 1: An1lisis de rendimiento de costosRiegoFinca

#### An1lisis de Resultados

- Identifique los tama1os de fincas donde el paralelismo genera ganancias significativas.

Al observar la tabla, se puede notar que el paralelismo produce ganancias significativas principalmente para fincas m1s grandes. Por ejemplo:

- Para **7 tablonos**, la aceleraci3n es del **5,74 %**.
- Para **8 tablonos**, aunque menor, a1n se observa una aceleraci3n del **2,14 %**.

A medida que aumenta el tamaño de la finca, la versión secuencial pierde eficiencia y la versión paralela se vuelve más competitiva.

En contraste, para **tamaños pequeños** (4 y 5 tablones), la **aceleración es más baja** (4,28 % y 3,72 %, respectivamente). Esto sugiere que el paralelismo no introduce tantas ventajas en estos casos debido a la baja cantidad de trabajo que se puede dividir entre los procesadores.

- Evalúe si las versiones paralelas introducen sobrecarga en casos pequeños.

Para **tamaños pequeños de fincas** (4 y 5 tablones), se observa que la *versión paralela introduce una sobrecarga considerable*. Por ejemplo:

- Con **4 tablones**, la versión secuencial toma **0,0104 ms**, mientras que la versión paralela tarda **0,2432 ms**.

Esta diferencia se debe a la **sobrecarga de administración del paralelismo**, como la división de tareas y la coordinación entre hilos. En estos casos, el costo de dividir el trabajo supera el beneficio obtenido, haciendo que el paralelismo sea ineficiente.

- Concluya sobre los beneficios del paralelismo para diferentes escenarios.
  - El **paralelismo es más beneficioso** para **tamaños grandes** de fincas (7 y 8 tablones), donde el trabajo puede distribuirse mejor entre los procesadores. A medida que el tamaño de los datos aumenta, la versión paralela supera claramente a la secuencial, logrando aceleraciones significativas.
  - Para **tamaños pequeños**, el paralelismo introduce una sobrecarga considerable que reduce su efectividad. En estos casos, la versión secuencial resulta más rápida y eficiente.

En conclusión, el paralelismo **debe utilizarse preferiblemente en escenarios con un mayor tamaño de datos**, donde el costo de dividir el trabajo se ve compensado por la reducción en el tiempo total de ejecución.

### 3.1.2. Función CostoMovilidad

```
1 val tiempoSecuencial = withWarmer(new Warmer.Default) measure {
    funcionSecuencial(finca, programacion, distancia) }
```

### 3.1.3. Función costoMovilidadPar.

```
1 val tiempoParalelo = withWarmer(new Warmer.Default) measure { funcionParalela(
    finca, programacion, distancia) }
```

**Presentación de Resultados** Se realizaron pruebas para los siguientes tamaños de fincas:

Tamaño finca (tablones)	Versión secuencial (ms)	Versión paralela (ms)	Aceleración (%)
4	0,0040	0,2830	1,41
5	0,0058	0,2343	2,48
7	0,0032	0,4000	0,8
8	0,0013	0,0698	1,86
9	0,0020	0,0510	3,92

Cuadro 2: Análisis de rendimiento de costoMovilidad

### Análisis :

1. **Ganancias significativas** : Las ganancias del paralelismo son **moderadas** incluso para tamaños grandes. Por ejemplo, para **9 tablones** , la aceleración alcanza **3.92 %** , pero sigue siendo baja debido al impacto de la suma secuencial al final del cálculo, lo cual limita la mejora.

2. **Sobrecarga en casos pequeños** : En tamaños pequeños, como **4 tableros** , la versión paralela toma casi **70 veces más tiempo** que la versión secuencial debido a la sobrecarga de paralelizar tareas pequeñas. En fincas pequeñas, la **sobrecarga introducida por la paralelización** (como creación de hilos y sincronización) domina el tiempo total, haciendo que el paralelismo no sea eficiente.
3. **Conclusión** : Dado que el cálculo de la movilidad tiene una porción significativa de operaciones secuenciales (como la suma final), los beneficios del paralelismo son limitados, y su uso solo es justificable para fincas más grandes.

### 3.2. Funciones para La generación de programaciones de riego

```

1 // Comparación de generación de programaciones
2 def compararGeneracion(
3     funcionSecuencial: Finca => Vector[ProgRiego], // Función secuencial a
4     evaluar (generar programaciones) recibe una finca y devuelve un vector de
5     programaciones
6     funcionParalela: Finca => Vector[ProgRiego], // Función paralela a evaluar (
7     generar programaciones) recibe una finca y devuelve un vector de programaciones
8     nombreSecuencial: String,
9     nombreParalela: String
10 )(finca: Finca): Unit = {
11     val tiempoSecuencial = withWarmer(new Warmer.Default) measure {
12         funcionSecuencial(finca) }
13     val tiempoParalelo = withWarmer(new Warmer.Default) measure { funcionParalela(
14         finca) }
15     val aceleracion = tiempoSecuencial.value / tiempoParalelo.value
16
17     // Impresión de resultados
18     println(f"\nTiempo $nombreSecuencial: ${tiempoSecuencial.value}%.4f ms")
19     println(f"Tiempo $nombreParalela: ${tiempoParalelo.value}%.4f ms")
20     println(f"Aceleración: $aceleracion%.4f")
21 }

```

#### 3.2.1. Función generarProgramacionesRiego

```

1 def generarProgramacionesRiego(f:Finca) : Vector[ProgRiego] = {
2     // Dada una finca de n tableros, devuelve todas las posibles programaciones de
3     // riego de la finca
4     val indices = (0 until f.length).toVector
5     indices.permutations.toVector
6 }

```

#### 3.2.2. Función generarProgramacionesRiegoPar

```

1 def generarProgramacionesRiegoPar(f:Finca) : Vector[ProgRiego] = {
2     // Genera las programaciones posibles de manera paralela
3     val indices = (0 until f.length).toVector
4     indices.permutations.toVector.par.toVector
5 }

```

**Presentación de Resultados** Se realizaron pruebas para los siguientes tamaños de fincas:

Tamaño finca (tableros)	Versión secuencial (ms)	Versión paralela (ms)	Aceleración (%)
4	0,2427	0,0965	251,55
5	0,1958	0,2119	92,4
6	1,1830	1,2874	91,89
8	31,9312	31,7602	100,54
9	330,6614	305,0163	108,41

Cuadro 3: Análisis de rendimiento de generarProgramaciones

## Análisis de Resultados

- **Identificación de tamaños de fincas donde el paralelismo genera ganancias significativas:**

Al analizar la tabla, se observa que el paralelismo produce **ganancias significativas en escenarios con fincas grandes**:

- Para **9 tablonos**, la aceleración es del **108,41 %**, lo que indica que la versión paralela supera ligeramente a la secuencial.
- Para **8 tablonos**, la aceleración es cercana al **100,54 %**, mostrando que el paralelismo es competitivo incluso en escenarios complejos.

En contraste, para **tamaños pequeños y medianos (5 y 6 tablonos)**, la aceleración se reduce (**92,4 % y 91,89 %**, respectivamente), mostrando que el paralelismo no genera ganancias significativas y, en algunos casos, introduce una leve sobrecarga.

- **Evaluación de sobrecarga para tamaños pequeños:**

Para **tamaños pequeños de fincas** (por ejemplo, 4 tablonos), se observa que el paralelismo ofrece una aceleración considerable del **251,55 %**, pero esta ventaja podría deberse a variaciones en la ejecución o a que el trabajo puede ser distribuido eficientemente entre hilos debido a la baja complejidad.

Para tamaños como **5 tablonos**, la versión paralela toma más tiempo (**0,2119 ms**) que la versión secuencial (**0,1958 ms**), lo que sugiere que la **sobrecarga del paralelismo** es más significativa que el beneficio obtenido al dividir el trabajo.

- **Conclusión sobre los beneficios del paralelismo:**

- El paralelismo es más beneficioso para **fincas grandes** (8 y 9 tablonos), donde el tiempo de ejecución de la versión secuencial aumenta considerablemente debido al número de permutaciones ( $n!$ ). En estos escenarios, la división del trabajo entre múltiples hilos reduce el tiempo de ejecución.
- Para **fincas pequeñas o medianas**, la versión paralela puede introducir sobrecarga adicional que compensa o incluso supera los beneficios del paralelismo. Esto es especialmente evidente en fincas con 5 y 6 tablonos.

En conclusión, el paralelismo en `generarProgramacionesRiegoPar` es más adecuado para escenarios con **gran cantidad de tablonos**, donde la carga computacional justifica el costo de dividir el trabajo entre múltiples procesadores. Para **tamaños pequeños o medianos**, la versión secuencial es más eficiente y competitiva.

### 3.3. Funciones para La generación de programaciones de riego óptima

```
1  def compararOptimo(  
2      funcionSecuencial: (Finca, Distancia) => (ProgRiego, Int), // Funcion  
3      funcionParalela: (Finca, Distancia) => (ProgRiego, Int), // Funcion paralela  
4      nombreSecuencial: String,  
5      nombreParalela: String  
6  )(finca: Finca, distancia: Distancia): Unit = {  
7      val tiempoSecuencial = withWarmer(new Warmer.Default) measure {  
8          funcionSecuencial(finca, distancia) }  
9      val tiempoParalelo = withWarmer(new Warmer.Default) measure { funcionParalela(  
10         finca, distancia) }  
11      val aceleracion = tiempoSecuencial.value / tiempoParalelo.value  
12  
13      // Impresi n de resultados  
14      println(f"\nTiempo $nombreSecuencial: ${tiempoSecuencial.value}%.4f ms")  
15      println(f"Tiempo $nombreParalela: ${tiempoParalelo.value}%.4f ms")  
16      println(f"Aceleraci n: $aceleracion%.4f")  
17  }
```

### 3.3.1. Función ProgramacionRiegoOptimo

```
1 def ProgramacionRiegoOptimo(f:Finca, d:Distancia) : (ProgRiego, Int) = {
2   // Dada una finca devuelve la programación de riego ptima
3   val programaciones = generarProgramacionesRiego(f)
4   val costos = programaciones.map(pi =>
5     (pi, costoRiegoFinca(f, pi) + costoMovilidad(f, pi, d))
6   )
7   costos.minBy(_._2)
8 }
```

### 3.3.2. Función ProgramacionRiegoOptimoPar

```
1 def ProgramacionRiegoOptimoPar(f:Finca, d:Distancia) : (ProgRiego, Int) = {
2   // Dada una finca, calcula la programación optima de riego
3   val programaciones = generarProgramacionesRiegoPar(f)
4   val costos = programaciones.par.map(pi =>
5     (pi, costoRiegoFincaPar(f, pi) + costoMovilidadPar(f, pi, d))
6   )
7   costos.minBy(_._2)
8 }
```

**Presentación de Resultados** Se realizaron pruebas para los siguientes tamaños de fincas:

Tamaño finca (tablones)	Versión secuencial (ms)	Versión paralela (ms)	Aceleración (%)
4	0,7869	1,9881	39,58
5	0,6189	2,6210	23,61
6	2,7323	10,6301	25,70
7	15,8536	35,1289	45,13
9	513,0081	1358,7512	37,76

Cuadro 4: Análisis de rendimiento de Programación Riego Óptima

### Análisis de Resultados

- **Identificación de tamaños de fincas donde el paralelismo genera ganancias significativas:**

Al observar la tabla, se nota que la versión paralela no muestra una aceleración significativa en comparación con la versión secuencial. La aceleración es notablemente baja para tamaños pequeños y medianos:

- Para **4 tablones**, la aceleración es de solo **39,58 %**, mostrando que la versión paralela es más lenta que la secuencial debido a la sobrecarga de paralelismo.
- Para **9 tablones**, la aceleración es de **37,76 %**, siendo insuficiente para justificar el costo adicional de la paralelización.

La falta de mejoras sustanciales se debe al alto costo computacional de generar todas las permutaciones ( $n!$ ) y calcular los costos asociados.

- **Evaluación de sobrecarga para tamaños pequeños:**

Para **fincas pequeñas y medianas** (4, 5 y 6 tablones), la versión paralela introduce una sobrecarga significativa:

- Con **4 tablones**, la versión secuencial toma **0,7869 ms**, mientras que la paralela toma **1,9881 ms**, lo que implica que el costo de dividir y coordinar el trabajo supera los beneficios del paralelismo.
- De manera similar, con **5 tablones**, la versión paralela es cuatro veces más lenta (**2,6210 ms**) que la secuencial (**0,6189 ms**).

Esto sugiere que el paralelismo introduce una considerable sobrecarga administrativa cuando el tamaño de la finca no justifica la distribución del trabajo entre múltiples procesadores.

■ **Conclusión sobre los beneficios del paralelismo:**

- El **paralelismo es menos beneficioso incluso para fincas grandes** (7 y 9 tablones). Aunque la versión paralela muestra una mejora relativa en la aceleración (**45,13 %** para 7 tablones y **37,76 %** para 9 tablones), el tiempo absoluto de ejecución sigue siendo significativamente mayor que el de la versión secuencial.
- Para **fincas pequeñas o medianas**, el paralelismo no es adecuado debido a la sobrecarga adicional, lo que hace que la versión secuencial sea más eficiente.

En conclusión, la función `ProgramacionRiegoOptimoPar` no logra aprovechar plenamente las ventajas del paralelismo en los escenarios evaluados. La alta complejidad computacional del problema ( $n!$ ) y la necesidad de operaciones secuenciales como `minBy` limitan los beneficios del paralelismo, haciéndolo más costoso que la versión secuencial incluso en fincas grandes.

## 4. Informe de corrección

### 4.1. Funciones para calcular los costos relacionados con el riego

#### 4.1.1. Función para calcular el tiempo de inicio de riego (tIR)

```

1 //2.3. Calculando el tiempo de inicio de riego
2 def tIR(f: Finca , pi: ProgRiego): TiempoInicioRiego = {
3   /*
4    Dada una finca f y una programacion de riego pi,
5    y f.length == n, tIR(f, pi) devuelve t: TiempoInicioRiego
6    tal que t(i) es el tiempo en que inicia el riego del
7    tablon i de la finca f segun pi
8    */
9
10   // Inicializamos tiempos en 0 para todos los tablones
11   val tiempos = Array.fill(f.length)(0)
12   // Recorremos la programaci n en orden de riego pi
13   for (j <- 1 until pi.length) {
14     val prevTablon = pi(j-1)
15     val currTablon = pi(j)
16     tiempos(currTablon) = tiempos(prevTablon) + treg(f, prevTablon)
17   }
18   tiempos.toVector
19 }

```

#### 4.1.2. Función para calcular el costo de riego por tablon (costoRiegoTablon)

La función `costoRiegoTablon` calcula el costo asociado al riego de un tablón específico dentro de una finca, dado un turno de riego definido en una programación específica. El cálculo del costo tiene en cuenta el tiempo de supervivencia del tablón, el tiempo de riego y la prioridad. A continuación, se argumenta su corrección formalmente.

La función en Scala está definida como:

```

1 def costoRiegoTablon(i: Int, f: Finca, pi: ProgRiego) : Int = {
2   val tiempoInicio = tIR(f, pi)(i)
3   val tiempoFinal = tiempoInicio + treg(f, i)
4   if (tsup(f,i) - treg(f, i) >= tiempoInicio) {
5     tsup(f,i) - tiempoFinal
6   } else {
7     prio(f,i) * (tiempoFinal - tsup(f,i))
8   }
9 }

```

**Notación matemática** Para un tablón  $T_i$  de una finca  $F$  con programación de riego  $\Pi$ , el costo de riego está definido como:

$$\text{CostoRiego}(T_i, \Pi) = \begin{cases} t_{s_i} - (t_{\Pi_i} + t_{r_i}), & \text{si } t_{s_i} - t_{r_i} \geq t_{\Pi_i} \\ p_i \cdot ((t_{\Pi_i} + t_{r_i}) - t_{s_i}), & \text{en caso contrario.} \end{cases}$$



donde:

- $t_{s_i}$ : Tiempo de supervivencia del tablón  $T_i$ .
- $t_{r_i}$ : Tiempo de riego necesario para el tablón  $T_i$ .
- $t_{\Pi_i}$ : Tiempo de inicio de riego del tablón  $T_i$ , calculado a partir de la programación  $\Pi$ .
- $p_i$ : Prioridad asignada al tablón  $T_i$ .

**Argumento de corrección** La función `costoRiegoTablon` implementa las condiciones descritas anteriormente, asegurando que se calculan correctamente los costos en ambas situaciones posibles:

1. **Caso 1:**  $t_{s_i} - t_{r_i} \geq t_{\Pi_i}$ . En este caso, el tablón puede ser regado antes de que finalice su tiempo de supervivencia. El costo es simplemente la cantidad de días restantes después de haber regado el tablón, calculado como:

$$\text{Costo} = t_{s_i} - (t_{\Pi_i} + t_{r_i}).$$

Esto se corresponde con la condición:

```
if (tsup(f, i) - treg(f, i) >= tiempoInicio) {  
    tsup(f, i) - tiempoFinal  
}
```

2. **Caso 2:**  $t_{s_i} - t_{r_i} < t_{\Pi_i}$ . En este caso, el tablón no puede ser regado a tiempo, por lo que el costo se incrementa según la prioridad del tablón y la cantidad de días de retraso, calculado como:

$$\text{Costo} = p_i \cdot ((t_{\Pi_i} + t_{r_i}) - t_{s_i}).$$

Esto corresponde al bloque:

```
else {  
    prio(f, i) * (tiempoFinal - tsup(f, i))  
}
```

**Conclusión** La función `costoRiegoTablon` es correcta, ya que implementa fielmente las condiciones del problema, asegurando que los costos se calculan de manera precisa tanto para los tablonos regados a tiempo como para los que sufren retrasos. Los casos de prueba confirman la validez de los resultados producidos.

#### 4.1.3. Función para calcular el costo de riego Finca (`costoRiegoFinca`)

La función `costoRiegoFinca` calcula el costo total de riego para una finca completa, sumando los costos individuales de cada tablón según una programación de riego específica. A continuación, se argumenta su corrección formalmente.

**Definición de la función** La función en Scala está definida como:

```
1 def costoRiegoFinca(f:Finca, pi:ProgRiego) : Int = {  
2     (0 until f.length).map(i => costoRiegoTablon(i, f, pi)).sum  
3 }
```

**Notación matemática** Para una finca  $F$  con  $n$  tablonos  $T_0, T_1, \dots, T_{n-1}$  y una programación de riego  $\Pi$ , el costo total de riego se define como:

$$\text{CostoRiegoTotal}(F, \Pi) = \sum_{i=0}^{n-1} \text{CostoRiego}(T_i, \Pi)$$

donde:

- $\text{CostoRiego}(T_i, \Pi)$ : Es el costo de riego para el tablón  $T_i$  calculado usando la función `costoRiegoTablon`.

**Argumento de corrección** La función `costoRiegoFinca` es una aplicación directa de la definición matemática del costo total de riego. Su corrección puede argumentarse en los siguientes pasos:

1. La función itera sobre todos los tablonos de la finca, asegurando que cada uno es considerado en el cálculo.
2. Para cada tablón  $T_i$ , se llama a la función `costoRiegoTablon`, cuya corrección ha sido demostrada previamente (ver Sección ??)

Por lo tanto, la función implementa correctamente el cálculo del costo total de riego.

**Conclusión** La función `costoRiegoFinca` es correcta, ya que implementa fielmente la definición matemática del costo total de riego para una finca. Los casos de prueba confirman la validez de los resultados producidos, asegurando que la suma de los costos individuales se realiza correctamente.

#### 4.1.4. Función para calcular el costo de movilidad (`costoRiegoMovilidad`)

```
1 def costoMovilidad(f:Finca, pi:ProgRiego, d:Distancia) : Int = {
2   (0 until pi.length - 1).map(j => d(pi(j))(pi(j+1))).sum
3 }
```

**Especificación de la Función:** La función `costoMovilidad` toma tres parámetros de entrada:

- **f:** una finca, que contiene los tablonos.
- **pi:** una programación de riego, que es una permutación de los índices de los tablonos de la finca.
- **d:** una función de distancia, que toma dos índices  $i$  y  $j$  y devuelve el costo de moverse del tablón  $i$  al tablón  $j$ .

La salida de la función es el costo total de movilidad para una programación de riego dada, y se calcula como la suma de los costos de moverse de un tablón al siguiente, según la programación de riego.

Formalmente, la especificación es:

$$\text{costoMovilidad}(f, pi, d) = \sum_{j=0}^{n-2} d(pi(j), pi(j+1))$$

Donde  $pi$  es una permutación de  $\{0, 1, \dots, n-1\}$  y  $d(i, j)$  devuelve el costo de moverse entre los tablonos  $i$  y  $j$ .

**Demostración de Corrección:** Para demostrar que la función es correcta, analizaremos cada uno de los pasos y verificaremos que cumple con la especificación de manera adecuada.

**Paso 1: Recorrido de la Programación de Riego:** La función recorre la programación de riego  $pi$  de tamaño  $n$ , iterando desde 0 hasta  $n-2$  (es decir, para cada par consecutivo de tablonos en la programación de riego).

$$(0 \text{ until } pi.length - 1)$$

Esto asegura que se recorren todos los pares consecutivos de tablonos.

**Paso 2: Cálculo de la Distancia entre los Tablonos:** La función  $d(pi(j))(pi(j+1))$  devuelve el costo de moverse del tablón  $pi(j)$  al tablón  $pi(j+1)$ . Esto asume que  $d$  es una función de distancia que toma dos índices y devuelve un costo. La expresión genera la distancia para cada par de tablonos consecutivos.

**Paso 3: Suma de los Costos:** La función

$$\text{map}(j \Rightarrow d(pi(j))(pi(j + 1))).\text{sum}$$

Crea un nuevo vector con los costos de movilidad para cada par consecutivo de tablones. Luego, **sum** calcula la suma total de estos costos. Por lo tanto, la salida de la función es la suma de las distancias de movilidad entre los tablones en la programación de riego.

**Cumplimiento de la Especificación:** Para cualquier finca  $f$ , programación de riego  $pi$ , y función de distancia  $d$ , la función **costoMovilidad** calcula la suma de los costos de moverse entre los tablones consecutivos según la programación de riego, lo cual es exactamente lo que se requiere en la especificación.

La especificación de la función es:

$$\text{costoMovilidad}(f, pi, d) = \sum_{j=0}^{n-2} d(pi(j), pi(j + 1))$$

La implementación realiza exactamente este cálculo. Por lo tanto, cumple con la especificación.

**Conclusión:** La función **costoMovilidad** es correcta, ya que:

- Recorre adecuadamente la programación de riego  $pi$ .
- Calcula correctamente el costo de movilidad entre tablones consecutivos.
- Suma los costos de manera correcta, lo que da el costo total de la movilidad.

Esto demuestra que la función implementada cumple con su especificación.

#### 4.1.5. Función para generar programaciones de riego (**generarProgramacionesRiego**)

```
1 def generarProgramacionesRiego(f:Finca) : Vector[ProgRiego] = {  
2   // Dada una finca de n tablones, devuelve todas las posibles  
   programaciones de riego de la finca  
3   val indices = (0 until f.length).toVector  
4   indices.permutations.toVector  
5 }
```

La función **generarProgramacionesRiego** tiene como entrada una finca  $f$  con  $n$  tablones, y su salida es un *vector de programaciones de riego*, donde cada programación es una permutación posible del conjunto de índices  $\{0, 1, \dots, n - 1\}$  (uno por cada tablón de la finca).

Formalmente, la función se puede definir como:

- **Entrada:** Una finca  $f$  con  $n$  tablones, donde  $f.\text{length} = n$ .
- **Salida:** Un vector de permutaciones  $P(f)$ , donde  $P(f)$  es el conjunto de todas las posibles permutaciones del conjunto  $\{0, 1, \dots, n - 1\}$ .

La especificación indica que la función debe cumplir:

$$\text{generarProgramacionesRiego}(f) = \text{permutaciones}(\{0, 1, \dots, f.\text{length} - 1\})$$

**Demostración de corrección:** Para demostrar que la función es correcta, es decir, que para cualquier finca  $f$ , **generarProgramacionesRiego**( $f$ ) devuelve exactamente  $P(f)$ , usamos **inducción estructural** sobre  $n$ , el número de tablones de la finca.

**Caso base:**  $n = 1$  Si la finca tiene solo un tablón ( $n = 1$ ), entonces el conjunto de índices es  $\{0\}$ , y la única permutación posible es el conjunto en sí:  $P(\{0\}) = \{\{0\}\}$ .

■ **Ejecución del programa:**

```
val indices = (0 until f.length).toVector // indices = Vector(0)
indices.permutations.toVector             // Vector(Vector(0))
```

La salida es exactamente  $\{\{0\}\}$ .

- **Cumplimiento de la especificación:** La especificación requiere que `generarProgramacionesRiego(f)`  $= P(\{0\})$ , y el programa devuelve  $\{\{0\}\}$ , cumpliendo con la especificación.

Por lo tanto, la función es correcta para  $n = 1$ .

**Caso inductivo:**  $n = k + 1$ , con  $k \geq 1$  **Hipótesis de inducción (HI):** Supongamos que la función es correcta para  $n = k$ , es decir, si  $f$  tiene  $k$  tablonos, entonces:

$$\text{generarProgramacionesRiego}(f) = \text{permutaciones}(\{0, 1, \dots, k - 1\})$$

**Demostrar para  $n = k + 1$ :**

Si  $f$  tiene  $k + 1$  tablonos, el conjunto de índices es  $\{0, 1, \dots, k\}$ , y el programa debe generar todas las permutaciones de este conjunto.

■ **Ejecución del programa:**

```
val indices = (0 until f.length).toVector // indices = Vector(0, 1, ..., k)
indices.permutations.toVector             // genera todas las permutaciones
```

El método `permutations` en un vector genera todas las permutaciones posibles del conjunto  $\{0, 1, \dots, k\}$ .

- **Uso de la hipótesis de inducción:** Por la definición de `permutations`, para un conjunto de tamaño  $k + 1$ , el algoritmo divide el conjunto en  $k + 1$  subconjuntos, aplicando la permutación recursivamente en los subconjuntos de tamaño  $k$ . La hipótesis de inducción asegura que las permutaciones generadas para subconjuntos de tamaño  $k$  son correctas.
- **Cumplimiento de la especificación:** Por construcción, el programa genera todas las permutaciones de tamaño  $k + 1$  combinando recursivamente las permutaciones de tamaño  $k$ . Esto garantiza que:

$$\text{generarProgramacionesRiego}(f) = \text{permutaciones}(\{0, 1, \dots, k\})$$

**Conclusión por inducción** Por inducción estructural sobre  $n$ , hemos demostrado que:

$$\forall n \geq 1, \text{generarProgramacionesRiego}(f) = \text{permutaciones}(\{0, 1, \dots, n - 1\})$$

La función cumple con su especificación para cualquier finca  $f$  con  $n$  tablonos.

#### 4.1.6. Función para calcular programaciones de riego óptima (`ProgramacionRiegoOptimo`)

```
1 //2.6. Calculando una programaci n de riego ptime
2 def ProgramacionRiegoOptimo(f:Finca, d:Distancia) : (ProgRiego, Int) = {
3   // Dada una finca devuelve la programaci n de riego ptime
4   val programaciones = generarProgramacionesRiego(f)
5   val costos = programaciones.map(pi =>
6     (pi, costoRiegoFinca(f, pi) + costoMovilidad(f, pi, d))
7   )
8   costos.minBy(_._2)
9 }
```

## Notación Matemática

## Argumento de Corrección

## Conclusión

### 4.2. Funciones para acelerar los cálculos con paralelismo de tareas y de datos

#### 4.2.1. Función para calcular el costo de riego Finca paralela (costoRiegoFincaPar)

Esta función tiene como objetivo calcular el costo total de riego para una finca de tablones dada una programación específica de riego. Se implementa utilizando procesamiento paralelo para optimizar el cálculo del costo individual de cada tablón.

**Definición de la Función** Sea esta la siguiente función en Scala:

```
1 //3.1. Paralelizando el calculo de los costos de riego y de movilidad
2 def costoRiegoFincaPar(f:Finca, pi:ProgRiego) : Int = {
3   // Devuelve el costo total de regar una finca f dada una programaci n de
4   // riego pi, calculando en paralelo
5   (0 until f.length).par.map(i => costoRiegoTablon(i, f, pi)).sum
6 }
```

La función toma dos parámetros:

- **f: Finca:** El conjunto de tablones con sus atributos (tiempo de supervivencia, tiempo de riego y prioridad).
- **pi: ProgRiego:** El vector que indica el orden de riego de los tablones.

**Paso 1: (0 until f.length)** Genera un rango desde 0 hasta el tamaño total de la finca. Cada índice  $i$  corresponde a un tablón de la finca.

**Paso 2: .par** Convierte la colección en una colección paralelizable. Permite que las operaciones dentro de `.map` se ejecuten en paralelo, utilizando múltiples hilos.

**Paso 3: .map(i → costoRiegoTablon(i, f, pi))** `costoRiegoTablon(i, f, pi)` calcula el costo de riego individual para el tablón  $i$ : Toma en cuenta el tiempo de inicio de riego y el tiempo de supervivencia. Aplica penalizaciones en caso de superar el tiempo máximo de supervivencia.

**Paso 4: .sum** Suma todos los costos individuales de riego de los tablones calculados en paralelo.

**Notación Matemática** Sea una finca  $F = \{T_0, T_1, \dots, T_{n-1}\}$ , donde cada tablón  $T_i$  está definido como una tupla  $(ts_i, tr_i, p_i)$ , siendo:

- $ts_i$ : tiempo de supervivencia del tablón  $i$ .
- $tr_i$ : tiempo requerido para regar el tablón  $i$ .
- $p_i$ : prioridad del tablón  $i$  (1 es la más baja y 4 la más alta).

Dada una programación de riego  $\Pi = \{\pi_0, \pi_1, \dots, \pi_{n-1}\}$ , que representa el orden en el cual se riegan los tablones, el costo de riego para un tablón  $T_i$  se define como:

$$C R_F^{\Pi}[i] = \begin{cases} ts_i - (t_i^{\Pi} + tr_i), & \text{si } ts_i - tr_i \geq t_i^{\Pi}, \\ p_i \cdot ((t_i^{\Pi} + tr_i) - ts_i), & \text{en otro caso.} \end{cases}$$

El costo total de riego de una finca  $F$  es:

$$CR_F^\Pi = \sum_{i=0}^{n-1} CR_F^\Pi[i].$$

La función `CostoRiegoFincaPar` paraleliza este cálculo, dividiendo las iteraciones del cálculo de  $CR_F^\Pi[i]$  entre múltiples núcleos.

**Argumento de Corrección** La corrección de la función `CostoRiegoFincaPar` se puede argumentar en tres pasos:

1. **Correctitud de la definición funcional:** La implementación utiliza el método `par.map` para calcular el costo de riego de cada tablón de forma independiente. Esto es posible porque el cálculo de  $CR_F^\Pi[i]$  depende únicamente de los valores de  $ts_i$ ,  $tr_i$ ,  $p_i$  y del tiempo de inicio  $t_i^\Pi$ , que son datos locales para cada tablón. Dado que no existe dependencia entre iteraciones, la paralelización respeta la lógica del algoritmo.

2. **Preservación de la semántica:** El uso de `par.map` garantiza que el resultado final es equivalente a aplicar `map` de forma secuencial y sumar los resultados. Esto se debe a que la operación de suma es conmutativa y asociativa, lo que asegura que el orden de ejecución de las iteraciones no afecta el resultado.

3. **Terminación y eficiencia:** El cálculo de  $CR_F^\Pi[i]$  para  $i = 0, \dots, n - 1$  se realiza en un espacio finito ( $n$  tablonos). Además, el framework de paralelización de Scala optimiza el uso de recursos computacionales distribuyendo las tareas entre los núcleos disponibles, lo que garantiza la finalización del programa en un tiempo menor al de la versión secuencial.

**Conclusión** La función `CostoRiegoFincaPar` es correcta, ya que:

- Implementa el cálculo del costo total de riego de acuerdo con la especificación matemática del problema.
- Utiliza técnicas de paralelización seguras y bien definidas, preservando la lógica funcional del programa.
- Mejora la eficiencia del cálculo para fincas con un gran número de tablonos, sin introducir errores debido a la concurrencia.

Los casos de prueba confirman que la función produce los mismos resultados que su contraparte secuencial (`CostoRiegoFinca`) para todas las entradas probadas, validando así su comportamiento.

#### 4.2.2. Función para calcular el costo de movilidad paralela (`costoRiegoMovilidadPar`)

```
1 def costoMovilidadPar(f:Finca,pi:ProgRiego, d:Distancia) : Int = {
2   // Calcula el costo de movilidad de manera paralela
3   (0 until pi.length - 1).par.map(j => d(pi(j))(pi(j+1))).sum
4 }
```

La función `costoMovilidadPar` tiene como objetivo calcular el costo total de movilidad de una programación de riego de manera paralela. A continuación, se detalla el análisis de su corrección:

**Especificación de la función:** La función `costoMovilidadPar` toma tres parámetros de entrada:

- `f`: Una finca, que contiene los tablonos.
- `pi`: Una programación de riego, que es una permutación de los índices de los tablonos de la finca.

- **d**: Una función de distancia, que toma dos índices  $i$  y  $j$  y devuelve el costo de moverse del tablón  $i$  al tablón  $j$ .

La salida de la función es el costo total de movilidad para una programación de riego dada, y se calcula como la suma de los costos de moverse de un tablón al siguiente, utilizando ejecución paralela para optimizar el cálculo. Formalmente, la especificación es:

$$\text{costoMovilidadPar}(f, pi, d) = \sum_{j=0}^{n-2} d(pi(j), pi(j+1))$$

Donde  $pi$  es la programación de riego y  $d(pi(j), pi(j+1))$  es el costo de moverse del tablón  $pi(j)$  al tablón  $pi(j+1)$ .

**Demostración de corrección:** La estructura de la función es muy similar a la de `costoMovilidad`, pero con la diferencia de que utiliza ejecución paralela en el cálculo de las distancias.

**Paso 1: Recorrido de la programación de riego:** La función recorre la programación de riego  $pi$  de tamaño  $n$ , iterando desde 0 hasta  $n-2$  (es decir, para cada par consecutivo de tablonos en la programación de riego):

$$(0 \text{ hasta } pi.length - 1)$$

Este recorrido asegura que se recorren todos los pares consecutivos de tablonos.

**Paso 2: Ejecución paralela:** El principal cambio en esta función es el uso de la ejecución paralela:

$$(0 \text{ hasta } pi.length - 1).par$$

Al aplicar `par` al rango, se indica que los cálculos para los pares consecutivos de tablonos se ejecutarán en paralelo, lo que puede mejorar el rendimiento en sistemas con múltiples núcleos.

**Paso 3: Cálculo de la distancia entre los tablonos:** El cálculo de la distancia entre los tablonos se realiza de la siguiente manera:

$$d(pi(j), pi(j+1))$$

Este cálculo se realiza en paralelo para cada par consecutivo de tablonos.

**Paso 4: Suma de los costos:** Después de calcular las distancias en paralelo, la función `map(j =>d(pi(j))(pi(j+1)))` genera un nuevo vector con los costos de movilidad para cada par consecutivo de tablonos, y `sum` calcula la suma total de estos costos:

$$map(j =>d(pi(j))(pi(j+1))).sum$$

La ejecución en paralelo no afecta el cálculo de la suma, ya que `sum` se encargará de esperar a que todos los cálculos paralelos se completen antes de realizar la suma.

**Cumplimiento de la especificación:** La función `costoMovilidadPar` sigue cumpliendo la especificación de la siguiente manera:

$$\text{costoMovilidadPar}(f, pi, d) = \sum_{j=0}^{n-2} d(pi(j), pi(j+1))$$

La implementación realiza exactamente este cálculo, solo que en paralelo.

### Consideraciones sobre la ejecución paralela:

- **Ejecución paralela:** El uso de `par` asegura que el cálculo de las distancias entre los tablonos se realice en paralelo. Esto es útil cuando la finca tiene muchos tablonos y el cálculo de distancias es costoso. Sin embargo, si el número de tablonos es pequeño, el costo de la sincronización de los hilos podría hacer que la versión paralela no sea más eficiente.
- **Consistencia de los resultados:** Dado que el cálculo de las distancias y la suma es independiente para cada par de tablonos, la ejecución en paralelo no afecta la corrección del resultado. La función sigue cumpliendo con la especificación de manera correcta.

**Conclusión:** La función `costoMovilidadPar` es correcta, ya que:

- Recorre adecuadamente la programación de riego  $pi$ .
- Calcula correctamente el costo de movilidad entre tablonos consecutivos.
- Suma los costos de manera correcta, y la ejecución paralela no afecta la corrección del resultado.

La única diferencia con la versión secuencial es que la ejecución paralela puede mejorar el rendimiento en algunos casos. Por lo tanto, la función cumple con la especificación.

#### 4.2.3. Función para generar programaciones de riego paralelas (`generarProgramacionesRiegoPar`)

```
1 def generarProgramacionesRiegoPar(f: Finca) : Vector[ProgRiego] = {  
2   // Genera las programaciones posibles de manera paralela  
3   val indices = (0 until f.length).toVector  
4   indices.permutations.toVector.par.toVector  
5 }
```

La función `generarProgramacionesRiegoPar` genera todas las posibles programaciones de riego para una finca  $f$  de manera paralela. Cada programación es una permutación de los índices de los tablonos de la finca. A continuación, se detalla el análisis de su corrección:

**Especificación de la función:** La función `generarProgramacionesRiegoPar` toma un parámetro de entrada:

- $f$ : Una finca, representada como una colección de tablonos.

La salida de la función es un vector de todas las permutaciones posibles de los índices de los tablonos en la finca. Formalmente, la especificación es:

$$\text{generarProgramacionesRiegoPar}(f) = \text{Permutaciones}(0, 1, \dots, n - 1)$$

Donde  $n = f.length$  es el número de tablonos en la finca.

**Demostración de corrección:** La función utiliza un enfoque secuencial para generar las permutaciones y luego lo convierte a un proceso paralelo. A continuación, se analizan los pasos de la función:

**Paso 1: Generación del rango de índices:** Se genera un vector de índices de los tablonos de la finca utilizando el rango:

`(0 until f.length).toVector`

Esto crea un vector de tamaño  $n$  que contiene los índices  $[0, 1, 2, \dots, n - 1]$ . Este rango es la entrada para el cálculo de las permutaciones.



**Paso 2: Generación de permutaciones:** Se generan todas las permutaciones posibles del vector de índices utilizando la función `indices.permutations`:

```
indices.permutations.toVector
```

Esto devuelve un iterador que contiene todas las permutaciones posibles del vector de índices. Dado que las permutaciones son independientes entre sí, el cálculo es correcto para todas las combinaciones posibles.

**Paso 3: Conversión a procesamiento paralelo:** El vector de permutaciones se convierte a un proceso paralelo utilizando `.par`:

```
toVector.par
```

Esto convierte el vector secuencial de permutaciones en un conjunto paralelo. Este cambio asegura que cualquier operación subsiguiente sobre las permutaciones se ejecute en múltiples hilos, mejorando el rendimiento en sistemas con varios procesadores.

**Paso 4: Conversión final a vector:** Después de completar cualquier procesamiento paralelo, las permutaciones se convierten nuevamente a un vector secuencial utilizando `.toVector`:

```
.par.toVector
```

Esto garantiza que la salida final de la función sea un vector secuencial, cumpliendo con la especificación de que la salida sea un `Vector[ProgRiego]`.

**Cumplimiento de la especificación:** La función `generarProgramacionesRiegoPar` cumple con la especificación de generar todas las permutaciones posibles de los índices de los tableros, y lo hace correctamente de acuerdo con los pasos descritos. La paralelización del cálculo no afecta la corrección, ya que las permutaciones son independientes entre sí.

#### Consideraciones sobre la ejecución paralela:

- **Ejecución paralela:** El uso de `.par` mejora el rendimiento al dividir el cálculo de permutaciones entre múltiples hilos. Esto es especialmente beneficioso cuando el número de tableros  $n$  es grande, ya que el número de permutaciones  $n!$  crece rápidamente.
- **Costos de sincronización:** Para fincas con un número pequeño de tableros, el costo de sincronización entre hilos puede superar las ventajas de la paralelización, haciendo que la versión paralela no sea significativamente más rápida que la versión secuencial.
- **Consistencia de los resultados:** Dado que las permutaciones se calculan de manera independiente, la paralelización no afecta la corrección del resultado.

**Conclusión:** La función `generarProgramacionesRiegoPar` es correcta, ya que:

- Genera correctamente todas las permutaciones de los índices de los tableros de la finca.
- Utiliza ejecución paralela de manera efectiva para dividir el cálculo de permutaciones.
- Asegura que la salida sea un vector secuencial, cumpliendo con la especificación.

La implementación paralela mejora el rendimiento en escenarios donde el número de tableros es grande, mientras que sigue produciendo resultados consistentes y correctos.

#### 4.2.4. Función para calcular la programación de riego óptima paralela (ProgramacionRiegoOptimoPar)

```

1  def ProgramacionRiegoOptimoPar(f:Finca, d:Distancia) : (ProgRiego, Int) = {
2      // Dada una finca, calcula la programación óptima de riego
3      val programaciones = generarProgramacionesRiegoPar(f)
4      val costos = programaciones.par.map(pi =>
5          (pi, costoRiegoFincaPar(f, pi) + costoMovilidadPar(f, pi, d))
6      )
7      costos.minBy(_._2)
8  }
```

La función `ProgramacionRiegoOptimoPar` tiene como objetivo calcular la programación de riego óptima para una finca  $f$  con base en una matriz de distancias  $d$ . Esta función utiliza paralelización para mejorar el rendimiento en el cálculo de costos para diferentes programaciones de riego. A continuación, se detalla el análisis de su corrección:

**Especificación de la función:** La función `ProgramacionRiegoOptimoPar` toma dos parámetros de entrada:

- $f$ : Una finca que contiene los tablones y sus atributos relevantes.
- $d$ : Una matriz de distancias, donde  $d(i, j)$  representa el costo de moverse entre los tablones  $i$  y  $j$ .

La salida de la función es una tupla que contiene:

- a) La programación óptima de riego (*ProgRiego*), que es una permutación de los índices de los tablones.
- b) El costo total de dicha programación (*Int*), calculado como la suma de los costos de riego y movilidad.

Formalmente, la especificación es:

$$\text{ProgramacionRiegoOptimoPar}(f, d) = \operatorname{argmin}_{\pi \in \text{Permutaciones}(f)} (C_{\text{riego}}(f, \pi) + C_{\text{movilidad}}(f, \pi, d))$$

Donde:

- $C_{\text{riego}}(f, \pi)$ : Costo total de riego para la programación  $\pi$ .
- $C_{\text{movilidad}}(f, \pi, d)$ : Costo total de movilidad para la programación  $\pi$ , basado en la matriz de distancias  $d$ .

**Demostración de corrección:** La función utiliza paralelización para calcular el costo total de cada programación y selecciona la programación óptima. Se analizan los pasos de la función:

**Paso 1: Generación de programaciones de riego:** Se generan todas las posibles programaciones de riego utilizando la función `generarProgramacionesRiegoPar`. Esto asegura que se consideren todas las permutaciones de los índices de los tablones. Formalmente:

$$\text{programaciones} = \text{Permutaciones}(0, 1, \dots, n - 1)$$

Donde  $n$  es el número de tablones en la finca.

**Paso 2: Cálculo paralelo de costos:** Para cada programación  $\pi \in \text{programaciones}$ , la función calcula su costo total sumando los costos de riego y movilidad. Este cálculo se realiza en paralelo utilizando `.par.map`. Formalmente:

$$\text{costos} = \{(\pi, C_{\text{riego}}(f, \pi) + C_{\text{movilidad}}(f, \pi, d)) \mid \pi \in \text{programaciones}\}$$

El cálculo paralelo asegura que los costos de las programaciones se computen de manera independiente, lo que no afecta la corrección del resultado.

**Paso 3: Selección del costo mínimo:** La programación óptima se selecciona utilizando `minBy`, que encuentra la programación con el menor costo total:

$$\text{programacion\_optima} = \operatorname{argmin}_{(\pi, c) \in \text{costos}} c$$

Esta operación es secuencial, pero garantiza que se seleccione la programación correcta, cumpliendo con la especificación.

**Cumplimiento de la especificación:** La función cumple con la especificación de calcular la programación óptima de riego y su costo total. Cada paso del cálculo es consistente con la definición formal, y el uso de paralelización no afecta la corrección debido a la independencia entre las programaciones.

#### Consideraciones sobre la ejecución paralela:

- **Ejecución paralela:** El uso de `.par.map` divide el cálculo de costos entre múltiples hilos, mejorando el rendimiento en sistemas con varios núcleos. Esto es especialmente beneficioso cuando el número de tablonos es grande, ya que el número de permutaciones  $n!$  crece rápidamente.
- **Costos de sincronización:** Aunque la paralelización mejora el rendimiento, el costo de sincronización entre hilos puede reducir su efectividad para fincas pequeñas.
- **Consistencia de los resultados:** Dado que las programaciones y sus costos se calculan de manera independiente, la paralelización no afecta la corrección del resultado.

**Conclusión:** La función `ProgramacionRiegoOptimoPar` es correcta, ya que:

- Genera correctamente todas las programaciones de riego.
- Calcula los costos de riego y movilidad de manera precisa para cada programación.
- Selecciona la programación óptima basada en el costo total mínimo.
- Utiliza paralelización para mejorar el rendimiento, manteniendo la consistencia de los resultados.

La implementación paralela es especialmente útil para fincas con un gran número de tablonos, donde el número de permutaciones es alto, lo que justifica el uso de múltiples hilos para optimizar el tiempo de ejecución.

## 5. Conclusiones

Durante la realización de este proyecto se ha podido evidenciar el resultado de una solución de la vida real como es el problema del riego óptimo para un entorno agrícola de nuestro departamento del Valle del Cauca, la implementación de la paralelización permite obtener mejoras significativas en este contexto complejo como son los procesos de la programación óptima de riego. Al implementar soluciones tanto secuenciales como paralelas en un entorno agrícola donde la optimización de recursos y saber usarlos es crucial, brindan un mejor y amplia perspectiva al momento de tomar decisiones al abordar estos problemas.

- **Implementación secuencial vs. paralela:** Al momento de realizar las pruebas correspondientes a las funciones obtenidas demostraron que, para escenarios donde el tamaño de la finca es un mayor tamaño, la implementación de una función paralela es mejor que la secuencial en termino de tiempo de ejecución. Si observamos el transcurso del desarrollo de las pruebas que a medida que aumentan los tablonos de la finca, el paralelismo se vuelve una herramienta indispensable para reducir los tiempos de ejecución (tiempos de procesamiento) y mejorar la eficiencia de este

- **Desempeño y eficiencia:** Así como la paralelización puede traer beneficios significativos para fincas de mayor tamaño aquellas que su tamaño era considerablemente baja notamos que el paralelismo en vez de hacer un bien lo que causaba a las fincas pequeñas era una sobrecarga que hacían disminuir su efectividad ya que se hace un esfuerzo adicional de coordinación y ejecución entre hilos, haciendo que para problemas de menor magnitud no sea recomendable usarlo.
- **Condiciones de prueba:** El análisis de las pruebas nos indica que se debe tener en cuenta el problema y en base a esto saber que solución brindar en el caso del paralelismo permite saber a que si al aplicarlo a un problema con menor magnitud es decir con pocos datos este no beneficia. Pero, lo que hace es sobrecargar y disminuir la efectividad del problema pero si ya son datos de gran magnitud este no puede faltar
- **Desempeño y utilidad:** El análisis de resultado reafirmaron sobre lo que se a dicho anteriormente la utilidad del paralelismo como estrategia de mejora es esencial, especialmente cuando se trabaja con grandes cantidades de datos. El proceso que el paralelismo realiza para distribuir tareas entre varios hilos hacen que se obtenga un beneficio en la optimización del tiempo de ejecución, esto mantiene siempre las consistencias de los datos obtenidos.

En resumen el proyecto realizado nos demostró que la paralelización es una estrategia eficaz para optimizar problemas de grandes magnitudes y que son aplicados a problemas reales. Observamos que las implementaciones paralelas no solo reducen el tiempo de cálculo, sino que también optimizan el uso de recursos, resultando en una mejora considerable del desempeño en comparación con las versiones secuenciales tradicionales la clave para esto es identificar los escenarios factibles para su aplicación.