Informe Proyecto Final

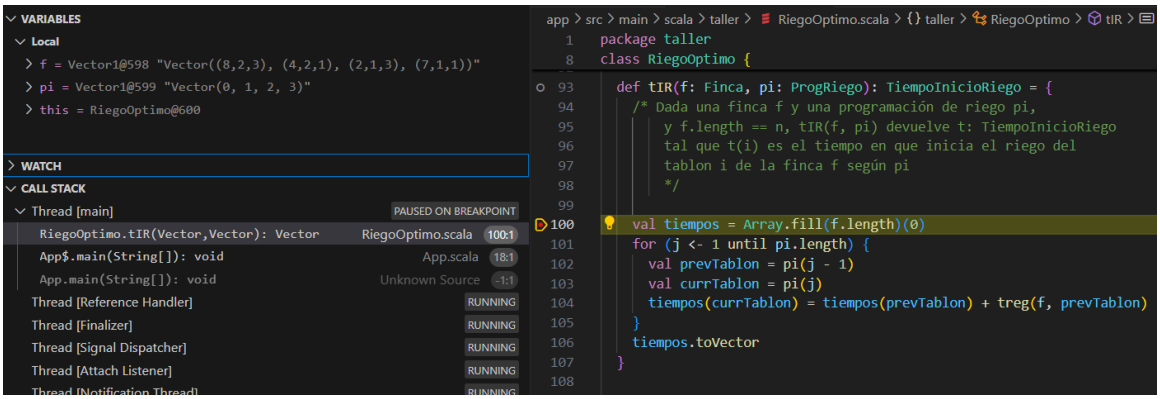
Jaime Andrés Noreña 2359523
Dilan Muricio Lemos 2359416
Juan Jose Restrepo 2359517
Diego Fernando Lenis 2359540

September 2024

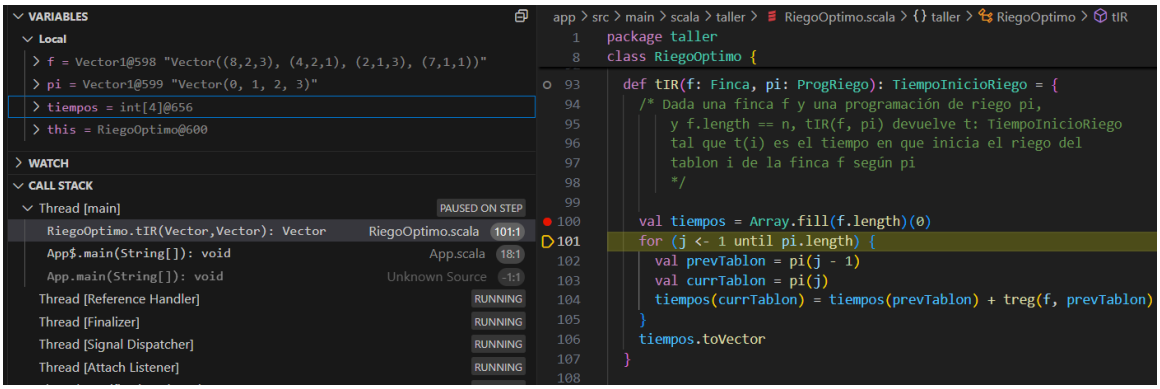
1 Informe de Procesos

1.1 tIR

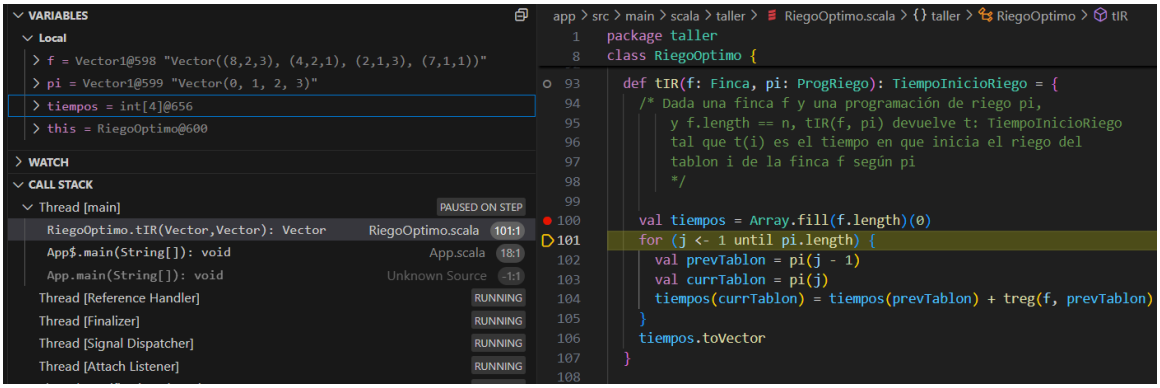
```
def tIR(f: Finca, pi: ProgRiego): TiempoInicioRiego = {
  val tiempos = Array.fill(f.length)(0)
  for (j <- 1 until pi.length) {
    val prevTablon = pi(j - 1)
    val currTablon = pi(j)
    tiempos(currTablon) = tiempos(prevTablon) + treg(f, prevTablon)
  }
  tiempos.toVector
}
```



Para calcular tiempo de inicio de riego de cada tablon primero se llena un arreglo con el tamaño igual al numero de tablones de la finca.



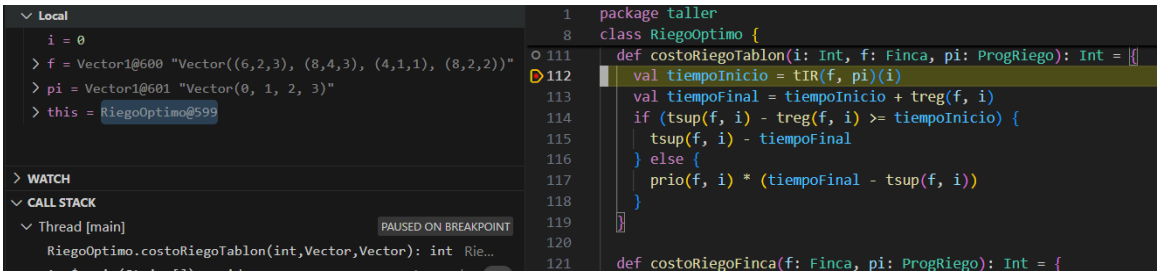
Despues de ello se itera j sobre un rango desde el 1 hasta uno menos del tamaño de la finca siendo este el indice para ir guardando el tiempo de inicio de rriego de cada tablon



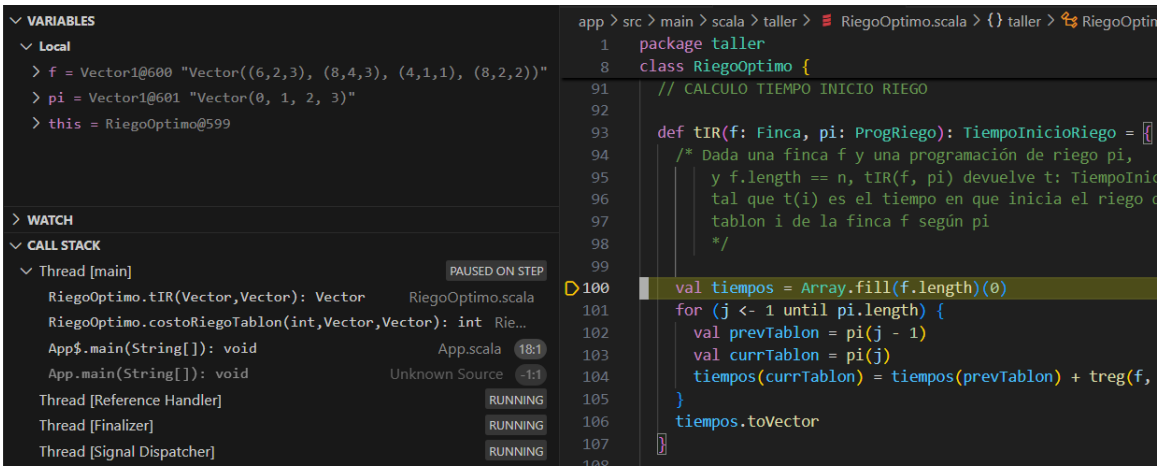
Después de que se guardara el tiempo de inicio de riego de todos los tablones retornamos un vector hecho a partir del array anterior

1.2 costoRiegoTablon

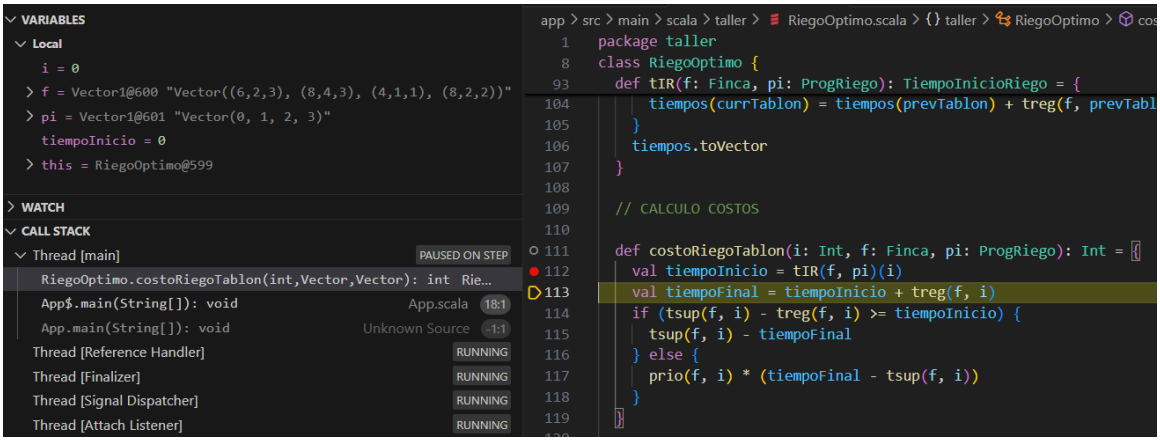
```
def costoRiegoTablon(i: Int, f: Finca, pi: ProgRiego): Int = {
  val tiempoInicio = tIR(f, pi)(i)
  val tiempoFinal = tiempoInicio + treg(f, i)
  if (tsup(f, i) - treg(f, i) >= tiempoInicio) {
    tsup(f, i) - tiempoFinal
  } else {
    prio(f, i) * (tiempoFinal - tsup(f, i))
  }
}
```



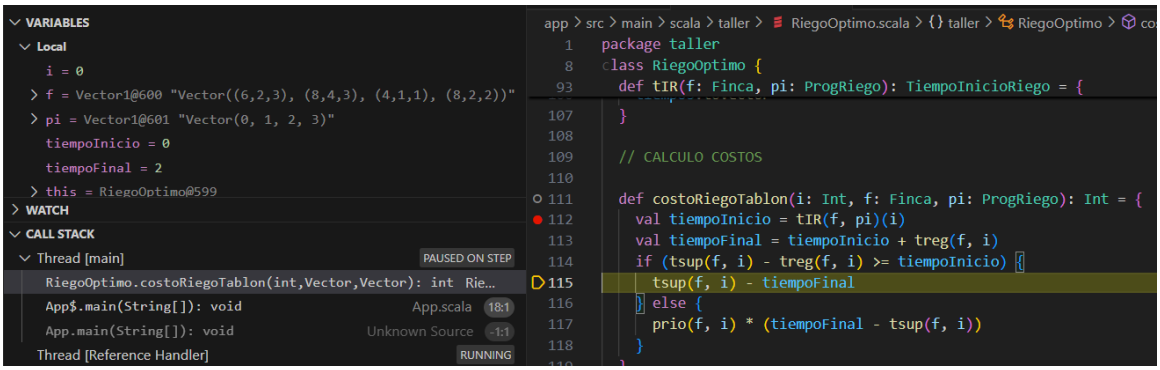
Primero calculamos el timepo de inicio de riego para el tablon actual y lo guardamos en `tiempoInicio`.



Vemos como entramos a calcular tiempo de inicio del riego para la finca actual(Notese la pila de llamandos).



Despues calculamos el `tiempoFinal` que es el tiempo de inicio mas el tiempo de riego el tablon actual(el tablon `i`)

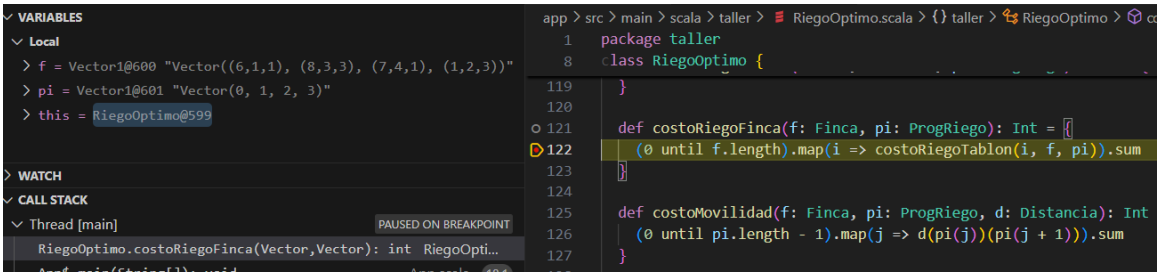


Por ultimo se calcula el costo teniendo en cuenta el criterio

$$CR_F^{\Pi}[i] = \begin{cases} ts_i^F - (t_i^{\Pi} + tr_i^F), & \text{si } ts_i^F - tr_i^F \geq t_i^{\Pi}, \\ p_i^F \cdot ((t_i^{\Pi} + tr_i^F) - ts_i^F), & \text{de lo contrario.} \end{cases}$$

1.3 costoRiegoFinca

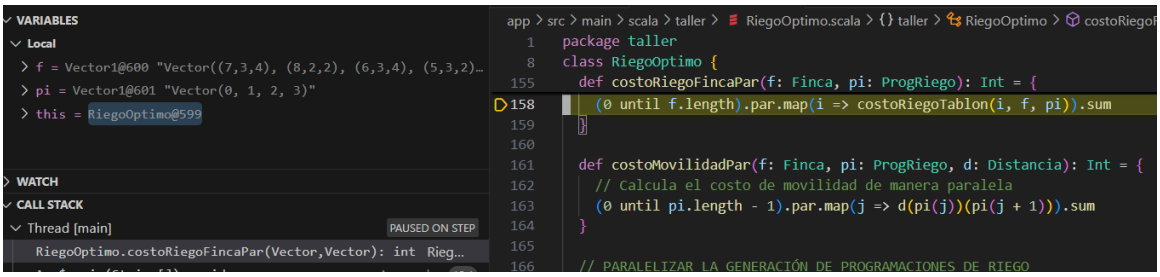
```
def costoRiegoFinca(f: Finca, pi: ProgRiego): Int = {
  (0 until f.length).map(i => costoRiegoTablon(i, f, pi)).sum
}
```



El calculo del costo del riego de una finca es la suma del costo de riego de cada uno de sus tablones entonces para ello se crea un rango de cero hasta el numero de tablones de la finca menos 1 para ir acceder cada tablón calcular su costo, creando un vector con todos los costos, lo que se retornara sera la suma de estos costos utillizando la funcion `sum`

1.4 costoRiegoFincaPar

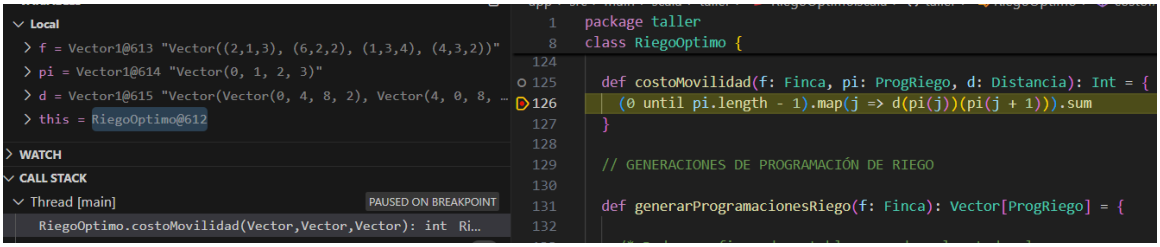
```
def costoRiegoFinca(f: Finca, pi: ProgRiego): Int = {
  (0 until f.length).par.map(i => costoRiegoTablon(i, f, pi)).sum
}
```



La version paralela de `costoRiegoFinca` usa la version paralela del rango para calcular el costo de riego de cada tablon siendo la libreria de escala quien se encaraga de la paralelizacion.

1.5 costoMobilidad

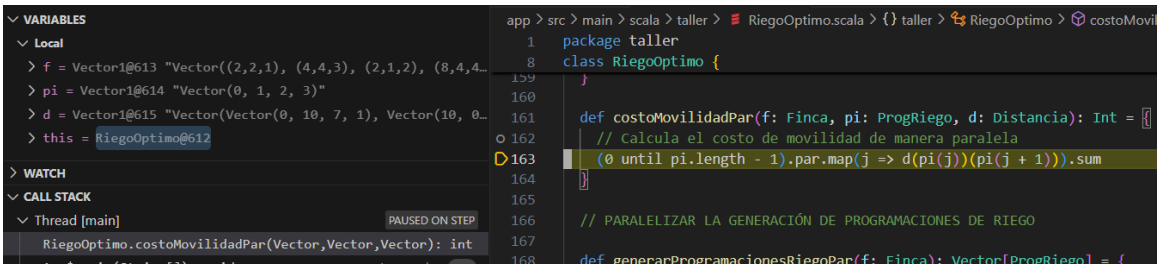
```
def costoMovilidad(f: Finca, pi: ProgRiego, d: Distancia): Int = {
    (0 until pi.length - 1).map(j => d(pi(j))(pi(j + 1))).sum
}
```



El calculo del costo de movilidad es se hace de una manera parecida al costo de riego de la finca, se crea un rango desde cero hasta el indice del ultimo tablón menos uno y se recorre utilizando la matriz de distancias donde se sumaran las distancias $j, j+1$.

1.6 costoMobilidadPar

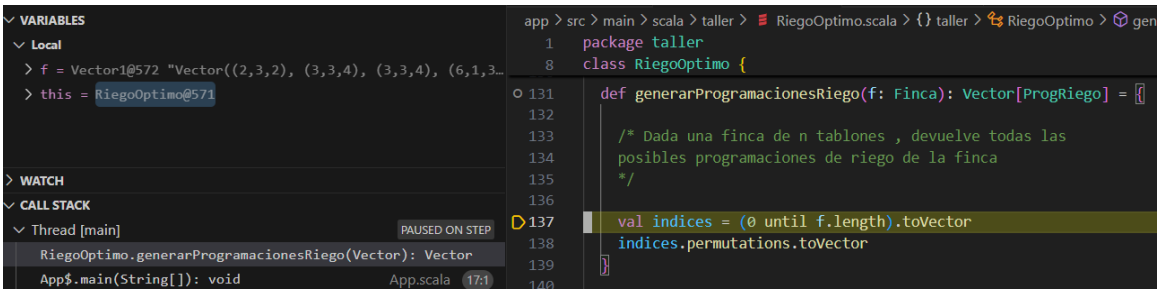
```
def costoMovilidad(f: Finca, pi: ProgRiego, d: Distancia): Int = {
    (0 until pi.length - 1).par.map(j => d(pi(j))(pi(j + 1))).sum
}
```



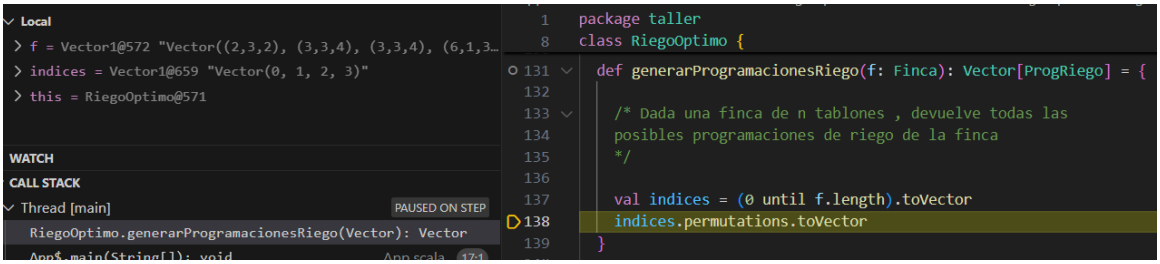
Se hace de manera idéntica solo que se utilizan las colecciones paralelas. Con el uso de `.par`.

1.7 generarProgramacionesRiego

```
def generarProgramacionesRiego(f: Finca): Vector[ProgRiego] = {

  /* Dada una finca de n tablones , devuelve todas las
  posibles programaciones de riego de la finca
  */

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



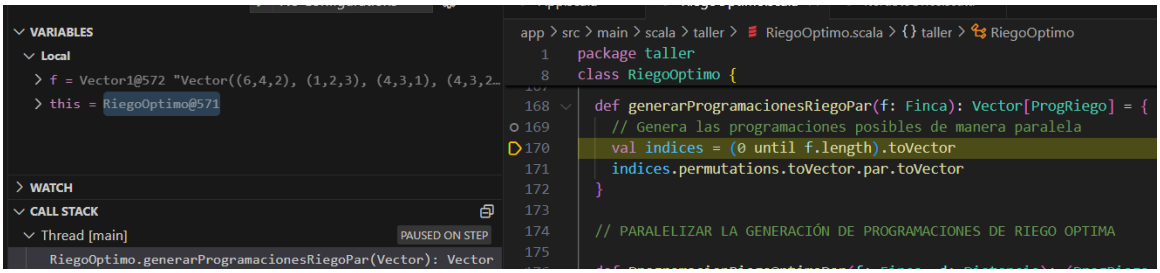
Primero creamos un vector de indices de cero hasta el tamaño de la finca menos uno.



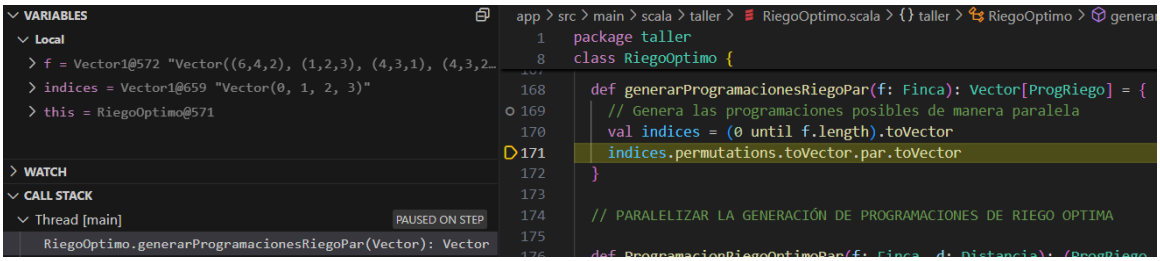
Ahora usando la función `permutations.toVector` generamos todos las permutaciones posibles de los valores del vector de indices las cuales también son todos los ordenes de riego posibles.

1.8 generarProgramacionesRiegoPar

```
def generarProgramacionesRiego(f: Finca): Vector[ProgRiego] = {

  /* Dada una finca de n tablones , devuelve todas las
  posibles programaciones de riego de la finca
  */

  val indices = (0 until f.length).toVector
  indices.permutations.toVector.par.toVector
}
```



Primero creamos un vector de indices de cero hasta el tamaño de la finca menos uno.

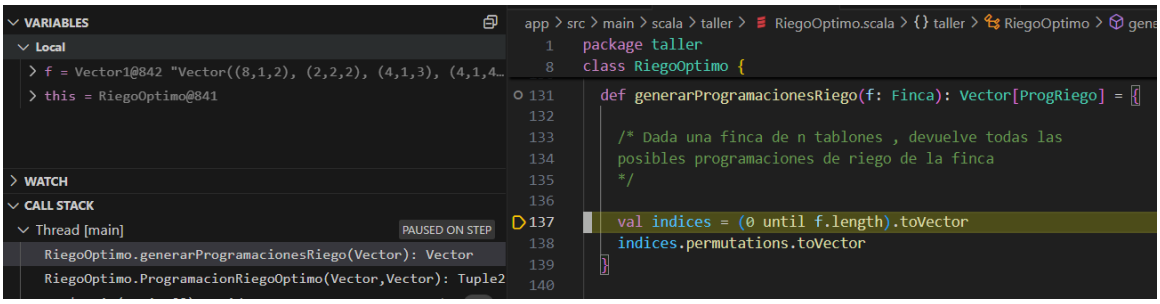


Ahora usando la función `permutations.toVector.par` generamos todos las permutaciones posibles de los valores del vector de indices de manera paralela. Las cuales también son todos los ordenes de riego posibles.

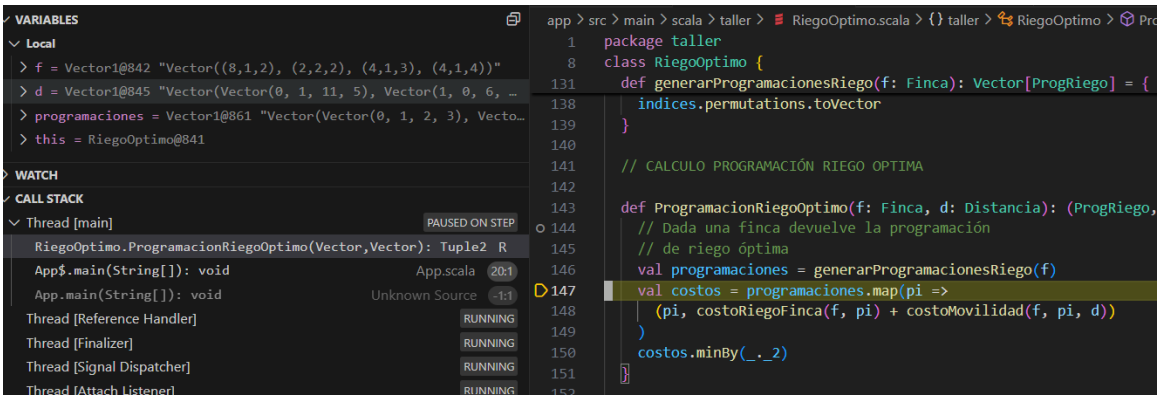
1.9 ProgramacionRiegoOptimo

```
def ProgramacionRiegoOptimo(f: Finca, d: Distancia): (ProgRiego, Int) = {
  // Dada una finca devuelve la programación
  // de riego óptima
  val programaciones = generarProgramacionesRiego(f)
  val costos = programaciones.map(pi =>
    (pi, costoRiegoFinca(f, pi) + costoMovilidad(f, pi, d))
  )
  costos.minBy(_._2)
}
```

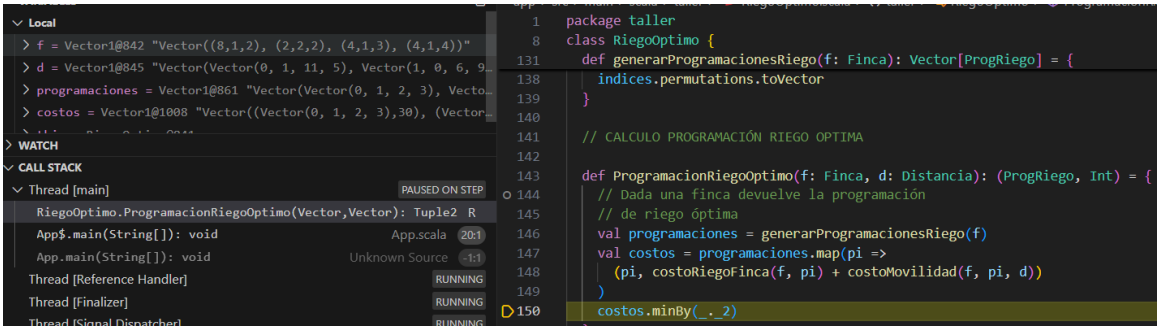
Primero Generamos todas la programaciones de riego posibles.



Vemos como entramos a la funcion `generarProgramaciones` (ver la pila de llamados)



Luego calculamos el costo de cada programacion y lo guardamos en la variable costos

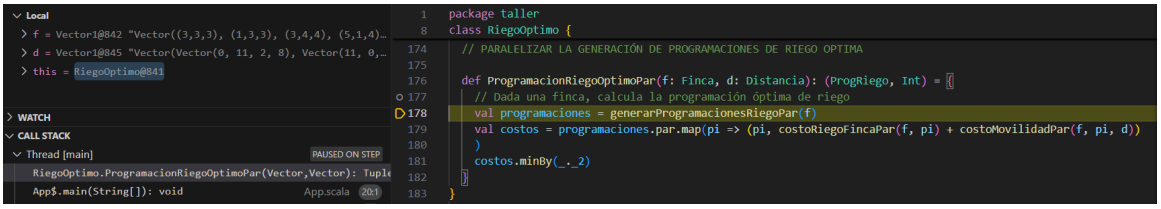


Por ultimo retornamos el menor de los costos usando la funcion `myBy` haciendo referencia al segundo elemento de la tupla que retornan la funciones de costo.

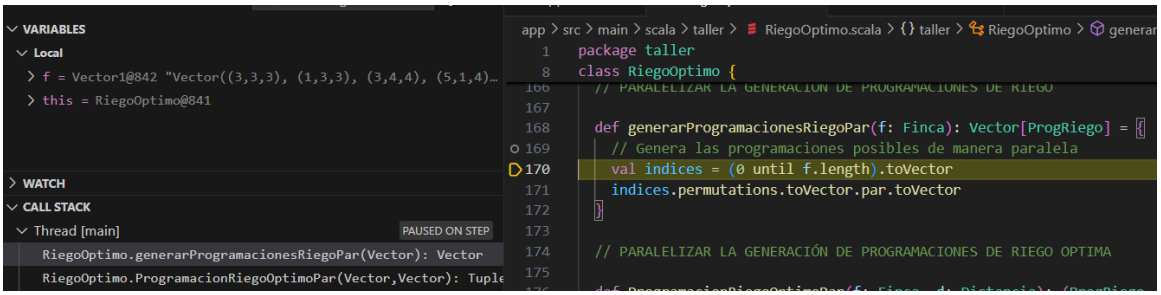
1.10 ProgramacionRiegoOptimoPar

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

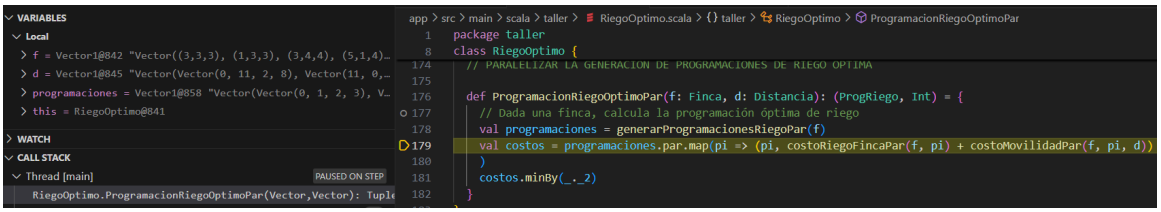
```
  }
}
```



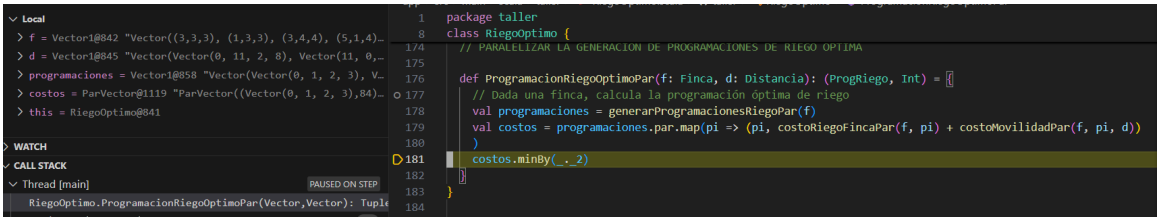
Primero Generamos todas la programaciones de riego posibles de manera paralela.



Como podemos ver entramos a la funcion `generarProgramacionPar` (ver pila de llamados)



Calculamos los costos de todas la programaciones usando la colleccion paralela y lo guardamos en la variable costos.



Por ultimo seleccionamos la programación con el menor costo utilizando la funcion `minBy`. Recordemos que la coleccion costos es una coleccion paralela por lo tanto la seleccion del menor costo tambien se hace de manera paralela.

2 Informe de Paralelizacion

2.1 Comparacion generarProgramacion

Con la siguiente tabla se mostraran los resultados de la medicion del tiempo que demoran en cumplir su tarea las funciones `generaProgramacion` y `generaProgramacionPar`

tiempo secuencial	tiempo paralelo	Aceleracion	tamaño finca
0.2074 ms	0.0872 ms	2.378440366972477	1
0.074 ms	0.0638 ms	1.1598746081504703	2
0.1012 ms	0.1074 ms	0.9422718808193669	3
0.0924 ms	0.1189 ms	0.7771236333052985	4
0.1638 ms	0.1688 ms	0.9703791469194313	5
0.546 ms	0.5508 ms	0.991285403050109	6
4.0583 ms	1.5007 ms	2.704271340041314	7
12.9205 ms	13.6805 ms	0.9444464749095428	8
132.4993 ms	125.0454 ms	1.0596095498115086	9
2004.1218 ms	1558.5987 ms	1.2858484996811559	10

Al analizar esta tabla vemos que la ganancia de eficiencia al paralelizar es minima y solo se vio mejora en las fincas de tamaño 7. Estos resulados indican que no hay mucha ganancia en paralizar para tamaños entre 1 y 10. Benchmarking:

```
Unable to create a system terminal
0.2267 ms & 0.1632 ms & 1.389093137254902 & 1
0.1669 ms & 0.52 ms & 0.3209615384615384 & 2
0.0978 ms & 0.0694 ms & 1.409221902017291 & 3
0.1497 ms & 0.1114 ms & 1.343806104129264 & 4
1.014 ms & 0.251 ms & 4.039840637450199 & 5
0.3175 ms & 0.1232 ms & 2.5771103896103895 & 6
1.3872 ms & 1.8011 ms & 0.7701959913386264 & 7
6.2037 ms & 7.2749 ms & 0.8527539897455636 & 8
89.6136 ms & 80.2497 ms & 1.1166845483534518 & 9
2094.396 ms & 1865.2715 ms & 1.1228370776050565 & 10
 *  Terminal will be reused by tasks, press any key to close it.
```

2.2 Comparacion costos movilidad

Con la siguiente tabla se mostraran los resultados de la medicion del tiempo que demoran en cumplir su tarea las funciones `costoMovilidad` y `costoMovilidadPar`

tiempo secuencial	tiempo paralelo	Aceleracion	tamaño finca
0.1618 ms	0.2518 ms	0.6425734710087371	1
0.0689 ms	2.6217 ms	0.026280657588587556	2
0.0326 ms	0.758 ms	0.04300791556728232	3
0.0261 ms	0.5144 ms	0.05073872472783827	4
0.011 ms	0.5364 ms	0.020507084265473527	5
0.0064 ms	0.4742 ms	0.013496415014761703	6
0.005 ms	0.1963 ms	0.02547121752419766	7
0.0038 ms	0.5622 ms	0.006759160441124155	8
0.0027 ms	0.365 ms	0.007397260273972603	9
0.0038 ms	0.3374 ms	0.011262596324836989	10

La aceleración es mínimamente significativa, a pesar d e medir el rendimiento en milisegundos, las mediciones son muy variadas aunque por un pequeño margen, de nuevo a partir del tamaño 7 es para el cual se empieza a "notar el cambio" cabe destacar que a parti de 10 iteraciones el programam sencillamente no funciona.

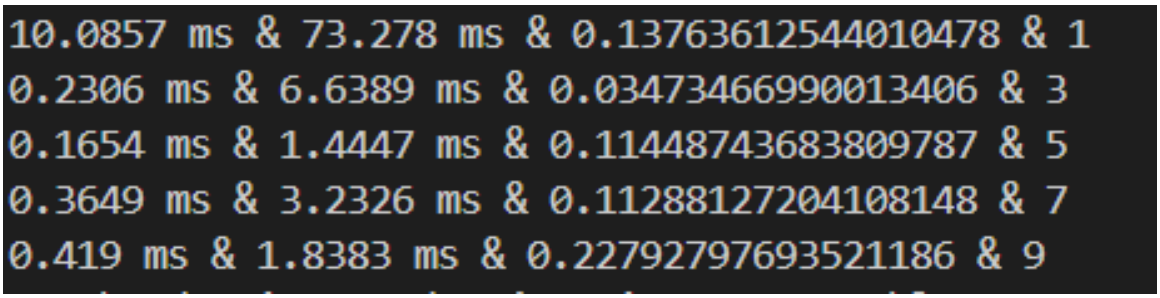
```
nalScalita\.metals\.tmp\classpath_A68E391F8F6231BB229922DDFCFF4430.jar" taller.App

pruebas ejecución - costo movilidad y version paralela
Unable to create a system terminal
Tiempo versión secuencial(ms)0.0573 ms & Tiempo versión paralela(ms) 0.3078 ms & 0.1861598440545809 & longitud: 1
Tiempo versión secuencial(ms)0.0419 ms & Tiempo versión paralela(ms) 0.3368 ms & 0.1244061757719715 & longitud: 2
Tiempo versión secuencial(ms)0.0202 ms & Tiempo versión paralela(ms) 0.495 ms & 0.04080808080808081 & longitud: 3
Tiempo versión secuencial(ms)0.0147 ms & Tiempo versión paralela(ms) 1.0055 ms & 0.01461959224266534 & longitud: 4
Tiempo versión secuencial(ms)0.0109 ms & Tiempo versión paralela(ms) 0.4332 ms & 0.025161588180978765 & longitud: 5
Tiempo versión secuencial(ms)0.0118 ms & Tiempo versión paralela(ms) 0.3228 ms & 0.036555142503097895 & longitud: 6
Tiempo versión secuencial(ms)0.006 ms & Tiempo versión paralela(ms) 0.3288 ms & 0.018248175182481754 & longitud: 7
Tiempo versión secuencial(ms)0.0028 ms & Tiempo versión paralela(ms) 2.9666 ms & 9.438414346389806E-4 & longitud: 8
Tiempo versión secuencial(ms)0.0018 ms & Tiempo versión paralela(ms) 0.4132 ms & 0.0043562439496611805 & longitud: 9
Tiempo versión secuencial(ms)0.0029 ms & Tiempo versión paralela(ms) 0.2939 ms & 0.009867301803334467 & longitud: 10
*  Terminal will be reused by tasks, press any key to close it.
```

2.3 Comparacion costos de Riego

Con la siguiente tabla se mostraran los resultados de la medicion del tiempo que demoran en cumplir su tarea las funciones `costoRiegoFinca` y `costoRiegoFincaPar`

tiempo secuencial	tiempo paralelo	Aceleracion	tamaño finca
10.0857 ms	73.278 ms	0.13763612544010478	1
0.2306 ms	6.6389 ms	0.03473466990013406	3
0.1654 ms	1.4447 ms	0.11448743683809787	5
0.3649 ms	3.2326 ms	0.11288127204108148	7
0.419 ms	1.8383 ms	0.22792797693521186	9

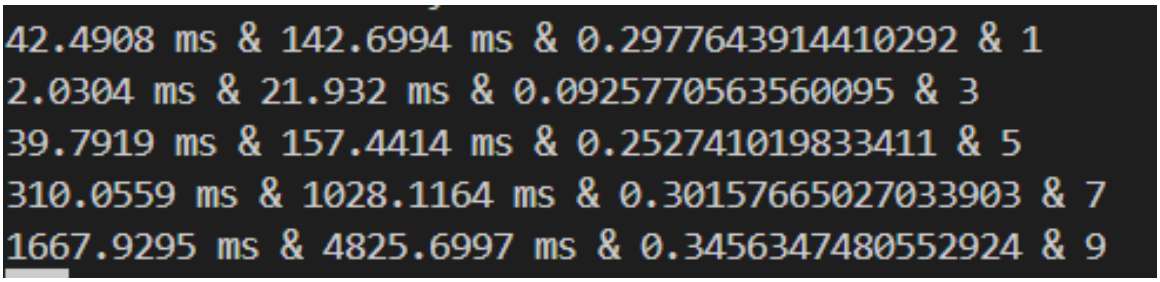
No se nota mejora alguna al ejecutar este metodo mediante paralelización, destaca el caso de tamaño 7 en el que es varias veces mas tardado que la versión lineal.



2.4 Comparacion Programacion Riego Optimo

Con la siguiente tabla se mostraran los resultados de la medicion del tiempo que demoran en cumplir su tarea las funciones `ProgramacionRiegoOptimo` y `ProgramacionRiegoOptimoPar`

tiempo secuencial	tiempo paralelo	Aceleracion	tamaño finca
42.4908 ms	142.6994 ms	0.2977643914410292	1
2.0304 ms	21.932 ms	0.0925770563560095	3
39.7919 ms	157.4414 ms	0.252741019833411	5
310.0559 ms	1028.1164 ms	0.30157665027033903	7
1667.9295 ms	4825.6997 ms	0.3456347480552924	9



3 Informe de Corrección

3.1 FincaAlAzar

La función 'fincaAlAzar' genera una finca como un vector de $n = \text{long}$ tablones, donde cada tablón es una tupla (ts_i, tr_i, p_i) que representa el tiempo de supervivencia ($ts_i \in [1, 2 \cdot \text{long}]$), tiempo de regado ($tr_i \in [1, \text{long}]$) y prioridad ($p_i \in [1, 4]$). Esto se logra mediante el uso de generación aleatoria respetando los rangos definidos en el problema. La función garantiza que la salida es un vector $F = \langle (ts_0, tr_0, p_0), \ldots, (ts_{n-1}, tr_{n-1}, p_{n-1}) \rangle$, donde cada elemento satisface las restricciones esperadas. La implementación es correcta, ya que genera una finca válida que cumple con la especificación del problema, asegurando valores en los rangos establecidos y con un número finito de operaciones.

3.2 DistanciaAlAzar

La función 'distanciaAlAzar' genera una matriz de distancias D de tamaño $n \times n$, donde $n = \text{long}$, que representa las distancias simétricas entre los tablones de una finca. Cada elemento $D[i][j]$ cumple

con las condiciones: $D[i][j] = D[j][i]$, $D[i][i] = 0$, y $D[i][j] \in [1, 3 \cdot \text{long}]$ si $i \neq j$. Esto asegura que la matriz es simétrica y cumple las propiedades de una matriz de distancias válida.

3.3 tIR

La función 'tIR' calcula el vector de tiempos de inicio de riego T para una finca f y una programación de riego π, asegurando que $T[i]$ representa correctamente el tiempo en que inicia el riego del tablón i. Matemáticamente, se define:

$$T[\pi_0] = 0, \quad T[\pi_j] = T[\pi_{j-1}] + tr_f(\pi_{j-1}), \, \forall j \in [1, n-1],$$

donde π_j es el índice del tablón en la posición j de la programación, y $tr_f(i)$ es el tiempo de regado del tablón i. El algoritmo asegura que: 1. T inicializa todos los tiempos en cero. 2. Itera secuencialmente sobre la programación π, actualizando $T[\pi_j]$ basándose en el tiempo acumulado y el tiempo de regado del tablón previo.

La implementación es correcta, ya que respeta la relación recursiva esperada, utiliza una cantidad finita de iteraciones, y garantiza que T se calcula adecuadamente según π y los valores de tr_f. Por lo tanto, la función genera un vector válido de tiempos de inicio de riego.

3.4 CostoRiegoTablón

La función 'costoRiegoTablon' calcula el costo de regar un tablón específico i de una finca f, dado un vector de programación de riego π. Matemáticamente, el costo está definido como:

$$CR_\pi(i) = \Big\{ ts_f(i) - (T[i] + tr_f(i)), \quad \text{si } ts_f(i) - tr_f(i) \geq T[i]$$

$\text{p}_f(i) \cdot ((T[i] + tr_f(i)) - ts_f(i)), \text{de lo contrario}$

donde:
- $ts_f(i)$ es el tiempo de supervivencia del tablón i
- $tr_f(i)$ es el tiempo de regado del tablón i
- $p_f(i)$ es la prioridad del tablón i
- $T[i]$ es el tiempo de inicio de riego del tablón i, calculado mediante 'tIR'.

La función evalúa si el riego ocurre dentro del tiempo de supervivencia ($ts_f(i) - tr_f(i) \geq T[i]$) para aplicar la primera fórmula, o si el riego se retrasa, aplicando la segunda fórmula ponderada por la prioridad.

La implementación sigue estrictamente las condiciones del problema donde calcula correctamente $T[i]$ y evalúa las fórmulas en función de las restricciones establecidas. Además, el cálculo utiliza una cantidad finita de operaciones y produce un valor coherente con el modelo matemático del costo de riego para un tablón.

3.5 CostoRiegoFinca

La función 'costoRiegoFinca' calcula el costo total de riego de una finca f dado un vector de programación de riego π. Matemáticamente, el costo total se define como:

$$CR_\pi(F) = \sum_{i=0}^{n-1} CR_\pi(i),$$

donde:
- n es el número de tablones en la finca f
- $CR_\pi(i)$ es el costo de riego del tablón i, calculado mediante la función 'costoRiegoTablon'.

La función implementa este cálculo iterando sobre todos los tablones de f, aplicando 'costoRiegoTablon' a cada uno, y acumulando los resultados con la operación 'sum'. Esto garantiza que cada tablón es considerado y que el costo total es la suma correcta de los costos individuales.

3.6 CostoMovilidad

La función 'costoMovilidad' calcula el costo total de movilidad para una finca f y una programación de riego π, dada una matriz de distancias d. El costo de movilidad se define como la suma de las distancias entre los tablones consecutivos según la programación π:

$$CM_\pi(F) = \sum_{j=0}^{n-2} d(\pi_j, \pi_{j+1}),$$

donde:
- n es el número de tablones en la finca f
- $d(i, j)$ es la distancia entre los tablones i y j, dada por la matriz d
- π_j es el índice del tablón en la posición j de la programación de riego π.

La función implementa este cálculo iterando sobre los índices de la programación π, accediendo a las distancias correspondientes en la matriz d para cada par de tablones consecutivos, y sumando estas distancias.

3.7 generarProgramacionesRiego

La función 'generarProgramacionesRiego' genera todas las posibles programaciones de riego para una finca f de n tablones. Matemáticamente, el número total de programaciones posibles es el número de permutaciones de los índices $0, 1, 2, \ldots, n-1$, lo que equivale a $n!$

La función implementa este cálculo de manera correcta mediante el uso de 'indices.permutations.toVector', donde 'indices' es el vector de índices de los tablones de la finca, $\langle 0, 1, 2, \ldots, n-1 \rangle$. La función 'permutations' genera todas las permutaciones posibles de estos índices, y 'toVector' las convierte en un vector de programación de riego π.

3.8 ProgramacionRiegoOptimo

La función 'ProgramacionRiegoOptimo' calcula la programación de riego óptima para una finca f y una matriz de distancias d, buscando la programación de riego que minimiza el costo total. El costo total para cada programación π se define como la suma del costo de riego y el costo de movilidad:

$$CR_\pi(F) + CM_\pi(F),$$

donde:
- $CR_\pi(F)$ es el costo total de riego de la finca f para la programación π, calculado con la función 'costoRiegoFinca'
- $CM_\pi(F)$ es el costo de movilidad para la programación π, calculado con la función 'costoMovilidad'.

La función primero genera todas las posibles programaciones de riego utilizando ‘generarProgramacionesRiego(f)‘, y luego calcula los costos para cada programación. Después, utiliza ‘minBy(_.2)‘.

para seleccionar la programación que tiene el costo mínimo, es decir, la programación óptima.

3.9 costoRiegoFincaPar

La función ‘costoRiegoFincaPar‘ calcula el costo total de regar una finca f dada una programación de riego π, utilizando un enfoque paralelo para mejorar el rendimiento. El costo total se define como la suma de los costos individuales de regar cada tablón de la finca:

$$CR_\pi(F) = \sum_{i=0}^{n-1} CR_\pi(i),$$

donde n es el número de tablones en la finca y $CR_\pi(i)$ es el costo de regar el tablón i, calculado mediante la función ‘costoRiegoTablon‘.

La función implementa este cálculo de manera paralela utilizando ‘par‘ sobre el rango de índices de los tablones, lo que permite que los costos de los tablones se calculen en paralelo. Después, suma los resultados utilizando ‘sum‘, obteniendo el costo total de riego.

3.10 costoMovilidadPar

La función ‘costoMovilidadPar‘ calcula el costo de movilidad de manera paralela para una finca f, dada una programación de riego π y una matriz de distancias d. El costo de movilidad se define como la suma de las distancias entre los tablones consecutivos en la programación π:

$$CM_\pi(F) = \sum_{j=0}^{n-2} d(\pi_j, \pi_{j+1}),$$

donde:
- $d(i, j)$ es la distancia entre los tablones i y j, proporcionada por la matriz de distancias d
- π_j es el índice del tablón en la posición j de la programación.

La función utiliza ‘par‘ para paralelizar el cálculo de las distancias entre tablones consecutivos en la programación π, y luego suma los resultados con ‘sum‘, obteniendo el costo total de movilidad.

3.11 generarProgramacionesRiegoPar

La función ‘generarProgramacionesRiegoPar‘ genera todas las posibles programaciones de riego para una finca f de n tablones de manera paralela. Matemáticamente, el número total de programaciones posibles es el número de permutaciones de los índices $0, 1, 2, \ldots, n-1$, es decir, $n!$ permutaciones.

La función primero genera el vector de índices $\langle 0, 1, 2, \ldots, n-1 \rangle$, luego calcula todas las permutaciones de estos índices usando ‘permutations‘. Finalmente, ‘toVector‘ convierte las permutaciones en un vector de programaciones de riego, y ‘par.toVector‘ aplica paralelismo para mejorar la eficiencia del cálculo de las permutaciones.

3.12 ProgramacionRiegoOptimoPar

La función 'ProgramacionRiegoOptimoPar' calcula la programación de riego óptima de una finca f dada una matriz de distancias d, utilizando un enfoque paralelo para mejorar el rendimiento. La programación óptima se define como la programación de riego que minimiza el costo total, compuesto por el costo de riego y el costo de movilidad.

El costo total para cada programación π se calcula como:

$$CR_\pi(F) + CM_\pi(F),$$

donde:
- $CR_\pi(F)$ es el costo total de regar la finca f para la programación π, calculado mediante 'costoRiegoFincaPar'
- $CM_\pi(F)$ es el costo de movilidad para la programación π, calculado mediante 'costoMovilidadPar'.

La función utiliza 'generarProgramacionesRiegoPar(f)' para generar todas las posibles programaciones de riego en paralelo. Luego, con 'par.map', calcula el costo total para cada programación en paralelo, lo que mejora la eficiencia en la evaluación de todas las programaciones posibles. Finalmente, se usa 'minBy($_{.2}$)'

para seleccionar la programación con el costo más bajo, es decir, la programación óptima.

4 Conclusiones

El proyecto demuestra la capacidad de aplicar conceptos de programación funcional y optimización para abordar problemas complejos del mundo real. La aplicación en Scala permite modelar, analizar y validar soluciones de forma eficiente, logrando resultados que pueden ser extendidos y paralelizados para un mejor rendimiento. La solución funcional presentada puede ser ampliada a sistemas más complejos, tales como el manejo de múltiples sistemas de riego o la optimización en tiempo real.