Informe Proyecto

Docente

DELGADO SAAVEDRA CARLOS ANDRES

Estudiantes

Juan David García Arroyave 2359450

Juan José Hincapié Tascón 2359493

Sebastián Zacipa Martinez 2359695

Universidad del Valle-Sede Tuluá

2024

Informe de procesos

Función fincaAlAzar

```
// Generación de finca al azar
def fincaAlAzar(long: Int): Finca = {
  val v = Vector.fill(long)((random.nextInt(long * 2) + 1, random.nextInt(long) + 1, random.nextInt(4) + 1))
  v
}
```

Función que genera una tupla con los valores de supervivencia, prioridad y tiempo de regado

Pila de llamadas con fincaAlAzar(3)

- Primer llamado: Vector.fill(3) por lo cual se van a generar 3 tuplas
- Segundo llamado: random.nextInt(3 *2) + 1, random.nextInt(3) + 1, random.nexInt(4) + 1
- Obtendremos valores de supervivencia entre 1 y 6, para tiempo de regado serán valores entre 1 y 3, y para la prioridad encontraremos valores entre 1 y 4 para cada una de las tuplas.

Función distanciaAlzar

```
27  // Generación de matriz de distancias al azar
28  def distanciaAlAzar(long: Int): Distancia = {
29    val v = Vector.fill(long, long)(random.nextInt(long * 3) + 1)
30    Vector.tabulate(long, long)((i, j) => if (i < j) v(i)(j) else if (i == j) 0 else v(j)(i))
31  }
```

Función que genera una matriz con las distancias entre cada una de las fincas, para mantener la consistencia de los tenemos unas condiciones que nos ayuda garantizar la simetría de la matriz

Pila de llamadas con distanciaAlAzar(3)

- Primer llamado: Vector.fill(long, long)(random.nextInt(long * 3) + 1)
- Supongamos que obtuvimos la siguiente matriz
- v = Vector(
- Vector(6, 4, 5),
- Vector(8, 3, 7),
- Vector(9, 2, 1)
-)
- Segunda llamado: Vector.tabulate(3,3) (0,0) como los valores de i y j son iguales el valor se rellena con 0, de esta tendremos la distancia de la finca con ella misma, y obtendremos la diagonal con ceros
- Después Vector.tabulate(3,3) (0,1) como i es menor que j ubicamos el valor v(0)(1) de la matriz generada, en este caso es igual a 4

- Para el siguiente llamado (3,3) (0,2) es la distancia entre la finca 0 y la 2 como i es menor que j ubicamos el valor v(0)(2) de la matriz generada, este valor es igual a 5
- Llamado Vector.tabulate(3,3) (1,0) para mantener la consistencia de la matriz, y i > j ubicamos el valor v(0)(1) este es igual 4
- Llamado Vector.tabulate(3,3) (1,1) como el valor de i y j son iguales se rellena con 0
- Llamado Vector.tabulate(3,3) (1,2) como el valor de i es menor que j ubicamos el valor de v(1)(2) este es igual a 7
- Llamado Vector.tabulate(3,3) (2,0) para mantener la consistencia de la matriz, y i > j ubicamos el valor v(0)(2) este es igual 5
- Llamado Vector.tabulate(3,3) (2,1) para mantener la consistencia de la matriz, y i > j ubicamos el valor v(1)(2) este es igual 7
- Llamado Vector.tabulate(3,3) (2,2) como el valor de i y j son iguales se rellena con 0

Función inicio tiempo de riego 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
}
```

Función para calcular el tiempo de inicio de riego de cada tablón

Pila de llamadas para val finca = Vector((10, 2, 1), (8, 3, 2), (6, 1, 3), (7, 4, 1))

- progRiego = Vector(0, 3, 1, 2)
- primera iteración : la lista de tiempos se completa con ceros. tiempos = Array.fill(f.length)(0).
- También encontramos que la iteración del bucle for empieza en 0. El valor de prevtablon para esta iteración es 0 ya que según la programación la primera finca es la 0, para currtablon el valor del segundo según la programación en este caso la finca 3. Actualizamos en la lista tiempos en la posición 3 el valor del anterior tablón y tiempo de ese mismo tablón
- tiempos(currtablon) = tiempos(prevTablon) + treg(f, prevTablon)
  - = tiempos(0) + treg(finca, 0)
  - = 0 + 2
  - = 2
- tiempos = Array(0, 0, 0, 2)
- tiempos(currtablon) = tiempos(prevTablon) + treg(f, prevTablon)
  - = tiempos(3) + treg(finca, 3)
  - = 2 + 4
  - = 6

- tiempos = Array(0, 6, 0, 2)
- tiempos(currtablon) = tiempos(prevTablon) + treg(f, prevTablon)
  = tiempos(1) + treg(finca, 1)
  = 6 + 3
  = 9
- tiempos = Array(0, 6, 9, 2)

Función 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))
  }
}
```

Función para calcular el costo de riego de un tablón

Pila de llamadas para val finca = Vector((10, 2, 1), (8, 3, 2), (6, 1, 3), (7, 4, 1)), costoRiegoTablon(i: 1, finca , Vector (0, 3, 1 2))

- Para tiempoInicio tenemos Vector(0, 6, 9, 2)(1) para el segundo tablón
- tiempoFinal = tiempoInicio + treg(f, 1) = 6 + 3 = 9
- tsup(f, 1) - treg(f, 1) >= tiempoInicio
- 8 - 3 >= 6
- 5 >= 6 // Falso
- prio(f, 1) * (tiempoFinal - tsup(f, 1))
- 2 * (9 - 8) = 2 * 1 = 2

Función CostoRiegoFinca

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

Función que calcula el costo de Riega toda una finca

Pila de llamadas para finca = Vector((10, 3, 2), (8, 2, 1), (12, 4, 3)) y progRiego = Vector(0,2,1)

- Primera iteracion
- Calcular el tiempo de inicio tir(0) = 0 para el tablón 0 comienza en tiempo 0
- tiempoInicio = tIR(0) = 0

- tiempoFinal = tiempoInicio + treg(0) = 0 + 3 = 3
- tsup(0) - treg(0) = 10 - 3 = 7
- Como 7 >= tiempoInicio (0), el costo es:
- costoRiegoTablon(0) = tsup(0) - tiempoFinal = 10 - 3 = 7
- Para el tablón 2 que es el segundo en la programación
- tIR(2) = tIR(0) + treg(0) = 0 + 3 = 3
- tiempoInicio = tIR(2) = 3
- tiempoFinal = tiempoInicio + treg(2) = 3 + 4 = 7
- tsup(2) - treg(2) = 12 - 4 = 8
- Como 8 >= tiempoInicio (3), el costo es:
- costoRiegoTablon(2) = tsup(2) - tiempoFinal = 12 - 7 = 5
- Para el tablón 1 que es el último en la programación
- tIR(1) = tIR(2) + treg(2) = 3 + 4 = 7
- tiempoInicio = tIR(1) = 7
- tiempoFinal = tiempoInicio + treg(1) = 7 + 2 = 9
- tsup(1) - treg(1) = 8 - 2 = 6
- Como 6 < tiempoInicio (7), el costo es:
- scala
- Copiar código
- costoRiegoTablon(1) = prio(1) * (tiempoFinal - tsup(1)) = 1 * (9 - 8) = 1
- costoRiegoFinca = costoRiegoTablon(0) + costoRiegoTablon(2) + costoRiegoTablon(1)
  = 7 + 5 + 1
  = 13

Función costoMovilidad

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

Función para calcular el costo de movilidad entre las diferentes Fincas

Pila de llamadas para val distancia: Distancia = Vector(Vector(0, 5, 3), Vector(5, 0, 4), Vector(3, 4, 0)), val progRiego: ProgRiego = Vector(0, 2, 1)

De tablón 0 a tablón 2,

Iteración 1 (j = 0):

pi(j) = pi(0) = 0.

pi(j + 1) = pi(1) = 2.

Distancia de tablón 0 a tablón 2:

d(0)(2) = 3

De tablón 2 a tablón 1

Iteración 2 (j = 1):

- pi(j) = pi(1) = **2.**
- pi(j + 1) = pi(2) = **1.**
- Distancia de tablón 2 a tablón 1:

d(2)(1) = 4

- **Costo acumulado hasta ahora: 3 + 4 = 7**

Función GenerarProgramacionesRiego

```
def generarProgramacionesRiego(f: Finca): Vector[ProgRiego] = {
  val indices = (0 until f.length).toVector
  indices.permutations.toVector
}
```

Esta es la encarga de generar todas las generaciones posibles

Pila de llamadas Vector((10, 3, 2), (8, 2, 1), (12, 4, 3))

Primera iteración se generan un rango entre 0 y 3 Vector(0, 1, 2)

Con indices.permutations.toVector al tener 3 elementos las posibles combinaciones serán 3!

Obteniedo un vector con seis diferentes programaciones

Función ProgramacionRiegoOptimo

```
77  def ProgramacionRiegoOptimo(f: Finca, d: Distancia): (ProgRiego, Int) = {
78    val programaciones = generarProgramacionesRiego(f)
79    val costos = programaciones.map(pi =>(pi, costoRiegoFinca(f, pi) + costoMovilidad(f, pi, d)))
80    costos.minBy(_._2) // Seleccionar la programación con el costo mínimo
81  }
```

- Vector((5, 2, 3),(6, 3, 2), (7, 1, 4))

- Matriz de Distancias (d):
- Vector(Vector(0, 4, 5), Vector(4, 0, 6), Vector(5, 6, 0))

- Generar todas las programaciones posibles
- Con 3 tablones, las programaciones posibles son:
- 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))

- Paso 2: Calcular los costos de cada programación
- Ejemplo: Programación pi = Vector(0, 1, 2)
    - Tiempos de inicio de riego (tIR):
    - Tablón 0: Empieza en 0.
    - Tablón 1: Empieza después de regar el tablón 0 → 0 + treg(0) = 2.
    - Tablón 2: Empieza después de regar el tablón 1 → 2 + treg(1) = 5.
    - Costos individuales:
    - Tablón 0: tsup(0) - tiempoFinal = 5 - 2 = 3.
    - Tablón 1: tsup(1) - tiempoFinal = 6 - 5 = 1.
    - Tablón 2: tsup(2) - tiempoFinal = 7 - 6 = 1.
    - Costo total de riego: 3 + 1 + 1 = 5.

- 2. **Costo de Movilidad (costoMovilidad):**

    - Distancias entre los tablones:
    - 0 → 1: d(0)(1) = 4.
    - 1 → 2: d(1)(2) = 6.
    - Costo total de movilidad: 4 + 6 = 10.
- 3. Costo Total:
- 5 (riego) + 10 (movilidad) = 15
- Repetir para todas las programaciones
- Se realizan los cálculos anteriores para cada una de las programaciones posibles.
- Seleccionar la mejor programación
- Se selecciona la programación con el menor costo total.
- La función retorna la mejor programación y su costo:
- (Vector(0, 1, 2), 15)

Función costoRiegoFincaPar

```
83    def costoRiegoFincaPar(f: Finca, pi:ProgRiego): Int = {
84      (0 until f.length).par.map(i => costoRiegoTablon(i, f, pi)).sum
85    }
```

Función que calcula el costo de Riego de manera paralela al dividir calcular el costoRiegoTablon de forma separada

Pila de llamadas para Finca: Vector((8, 3, 2), (6, 2, 3), (10, 4, 1)))

- Programación de riego: Vector(0, 1, 2)
- Se recorre paralelamente cada tablón (0, 1, 2) utilizando par.map. 2. Para cada tablón, se calcula su costo de riego con la función costoRiegoTablon en una tarea diferente.
- costoRiegoTablon(0,f,pi)
- costoRiegoTablon(1,f,pi).
  costoRiegoTablon(2,f,pi).
  Se suman los resultados.

Función costoMovilidadPar

```
87    def costoMovilidadPar(f:Finca,pi:ProgRiego, d:Distancia): Int = {
88      (0 until pi.length - 1).par.map(j => d(pi(j))(pi(j + 1))).sum
89    }
```

- Programación de riego: Vector(0, 2, 1)
- Matriz de distancias:
- Vector(Vector(0, 3, 5),Vector(3, 0, 2),Vector(5, 2, 0))
- Se recorre paralelamente cada par consecutivo de tablones en la programación (0 → 2, 2 → 1)
- Para cada par, se consulta la distancia:
  - De 0 a 2: d(0)(2) = 5
  - De 2 a 1: d(2)(1) = 2
- Se suman las distancias: 5 + 2 = 7

Función generarProgramacionesRiegoPar

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

- Finca: Vector((8, 3, 2), (6, 2, 3), (10, 4, 1))
- Se generan los índices de los tablones:
- Vector(0,1,2).
- Se calculan todas las permutaciones posibles:
- 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)
- 3. Se procesan en paralelo para mejorar el rendimiento.

Función programaciónOptimoPar

```
def ProgramacionRiegoOptimoPar(f:Finca, d:Distancia): (ProgRiego, Int) = {
  val programaciones = generarProgramacionesRiegoPar(f)
  val costos = programaciones.par.map(pi =>
    (pi, costoRiegoFincaPar(f, pi) + costoMovilidadPar(f, pi, d))
  )
  costos.minBy(_._2)
}
```

- Pila de llamadas par Finca: Vector((8, 3, 2), (6, 2, 3), (10, 4, 1))
- Matriz de distancias
- Vector(Vector(0,3,5),Vector(3,0,2),Vector(5,2,0))
- . Se generan todas las programaciones posibles con generarProgramacionesRiegoPar.
- Para cada programación, se calcula el costo total:
- para Vector(0, 1, 2):
- Costo de riego: costoRiegoFincaPar(f,Vector(0,1,2))
- costoMovilidadPar(f,Vector(0,1,2),d)
- Costo Total = Costo de riego + costo de movilidad

Informe de Corrección

1. fincaAlAzar

Descripción: Genera una finca de longitud long con tablones aleatorios. Cada tablón contiene: tiempo de supervivencia, tiempo de riego y prioridad.

Caso base:
Si long = 0, se espera un Vector vacío.

assert(fincaAlAzar(0).isEmpty)

Caso de inducción:
Para long > 0, se verifica que cada elemento del vector cumple las condiciones:

- El tiempo de supervivencia pertenece a [1, long * 2].
- El tiempo de riego pertenece a [1, long].
- La prioridad pertenece a [1, 14].

val finca = fincaAlAzar(10)

assert(finca.forall(tab => tab._1 >= 1 && tab._1 <= 20))

assert(finca.forall(tab => tab._2 >= 1 && tab._2 <= 10))

assert(finca.forall(tab => tab._3 >= 1 && tab._3 <= 14))

2. distanciaAlAzar

Descripción: Genera una matriz simétrica de distancias aleatorias entre tablones.

Caso base:
Si long = 0, se espera un vector vacío.

assert(distanciaAlAzar(0).isEmpty)

Caso de inducción:
Para long > 0, se verifica que:

- La matriz es simétrica (distancia[i][j] == distancia[j][i]).
- Los valores de la diagonal son 0.
- Cada distancia pertenece a [1, long * 3].

val distancias = distanciaAlAzar(5)

assert(distancias.indices.forall(i => distancias(i)(i) == 0))

assert(distancias.indices.forall(i => distancias.indices.forall(j => distancias(i)(j) == distancias(j)(i))))

assert(distancias.flatten.forall(d => d >= 1 && d <= 15 || d == 0))

3. tIR (Tiempo de Inicio de Riego)

Descripción: Calcula el tiempo de inicio de riego para cada tablón según una programación dada.

Caso base:
Si la programación contiene un único tablón (pi.length == 1), su tiempo de inicio es 0.

val finca = Vector((10, 2, 5))

val pi = Vector(0)

assert(tIR(finca, pi) == Vector(0))

Caso de inducción:
Si pi.length > 1, se verifica que:

- El tiempo de inicio del tablón actual depende del tiempo de riego del tablón previo.

val finca = Vector((10, 2, 5), (12, 3, 4))

val pi = Vector(0, 1)

val tiempos = tIR(finca, pi)

assert(tiempos(1) == tiempos(0) + finca(0)._2)

4. costoRiegoTablon

Descripción: Calcula el costo de riego de un tablón según su tiempo de inicio.

Caso base:
Si el tiempo de inicio del tablón permite completarlo antes de su tiempo de supervivencia (tsup - treg >= tiempoInicio), el costo es tsup - tiempoFinal.

val finca = Vector((10, 2, 5))

val pi = Vector(0)

assert(costoRiegoTablon(0, finca, pi) == 8) // 10 - (0 + 2)

Caso de inducción:
Si el tiempo final supera el tiempo de supervivencia (tsup < tiempoFinal), el costo es proporcional a la prioridad del tablón.

val finca = Vector((3, 2, 5)) // tsup = 3, treg = 2, prio = 5

val pi = Vector(0)

assert(costoRiegoTablon(0, finca, pi) == 10) // prio * (tiempoFinal - tsup) = 5 * (5 - 3)

5. costoRiegoFinca

Descripción: Calcula el costo total de riego para una finca según una programación.

Caso base:
Si la finca está vacía, el costo es 0.

assert(costoRiegoFinca(Vector.empty, Vector.empty) == 0)

Caso de inducción:
Para una finca de longitud n > 0, el costo total es la suma de los costos de cada tablón.

val finca = Vector((10, 2, 5), (8, 3, 4))

val pi = Vector(0, 1)

assert(costoRiegoFinca(finca, pi) == costoRiegoTablon(0, finca, pi) + costoRiegoTablon(1, finca, pi))

6. costoMovilidad

Descripción: Calcula el costo de movilidad basado en las distancias entre tablones según la programación.

Caso base:
Si la programación contiene un único tablón (pi.length == 1), no hay movilidad y el costo es 0.

val finca = Vector((10, 2, 5))

val distancias = Vector(Vector(0))

val pi = Vector(0)

assert(costoMovilidad(finca, pi, distancias) == 0)

Caso de inducción:
Para pi.length > 1, el costo es la suma de las distancias entre tablones consecutivos en la programación.

val distancias = Vector(Vector(0, 3), Vector(3, 0))

val pi = Vector(0, 1)

assert(costoMovilidad(finca, pi, distancias) == 3)

7. ProgramacionRiegoOptimo

Descripción: Encuentra la programación con el menor costo total de riego y movilidad.

Caso base:
Si la finca tiene un único tablón, la única programación posible tiene costo costoRiegoTablon(0).

Copiar código

```
val finca = Vector((10, 2, 5))

val distancias = Vector(Vector(0))

assert(ProgramacionRiegoOptimo(finca, distancias) == (Vector(0), 8))
```

Caso de inducción:
Para una finca con n > 1 tablones, se verifica que la programación óptima minimiza el costo total.

```
val finca = Vector((10, 2, 5), (8, 3, 4))

val distancias = Vector(Vector(0, 1), Vector(1, 0))

assert(ProgramacionRiegoOptimo(finca, distancias)._2 <= costoRiegoFinca(finca,
Vector(0, 1)) + costoMovilidad(finca, Vector(0, 1), distancias))
```

Informe de paralelización

```
105   def compararCostoRiego(seq: (Finca, ProgRiego)=> Int, par: (Finca, ProgRiego)=> Int)(finca: Finca, programariego: ProgRiego): List[Double] = {
106     val seqTime = withWarmer(new Default) measure {seq(finca, programariego)}
107     val parTime = withWarmer(new Default) measure {par(finca, programariego)}
108       List(seqTime.value, parTime.value, seqTime.value/parTime.value)
109     }
110
111   def compararCostoMovilidad(seq: (Finca, ProgRiego, Distancia)=> Int, par: (Finca, ProgRiego, Distancia)=> Int)(finca: Finca, programariego: ProgRiego, distancias: Distancia): List[Double] = {
112     val seqTime = withWarmer(new Default) measure {seq(finca, programariego, distancias)}
113     val parTime = withWarmer(new Default) measure {par(finca, programariego, distancias)}
114       List(seqTime.value, parTime.value, seqTime.value/parTime.value)
115     }
116
117   def compararGeneracionesDeRiego(seq: (Finca) => Vector[ProgRiego], par: (Finca) => Vector[ProgRiego]) (finca: Finca): List[Double] = {
118     val seqTime = withWarmer(new Default) measure {seq(finca)}
119     val parTime = withWarmer(new Default) measure {par(finca)}
120     List(seqTime.value, parTime.value, seqTime.value/parTime.value)
121   }
122
123   def compararProgramacionRiegoOptimo(seq: (Finca, Distancia)=> (ProgRiego, Int), par: (Finca, Distancia)=> (ProgRiego, Int)) (finca: Finca, distancia: Distancia): List[Double] = {
124     val seqTime = withWarmer(new Default) measure {seq(finca, distancia)}
125     val parTime = withWarmer(new Default) measure {par(finca, distancia)}
126     List(seqTime.value, parTime.value, seqTime.value/parTime.value)
127   }
128 }
129
```

Funciones para comparar las funciones secuenciales con las paralelas

5 tablones

```
14        val finca = riegoOptimo.fincaAlAzar(5)
15        val distancia = riegoOptimo.distanciaAlAzar(5) // Matriz compatible con 10 tablones
16        val programacionRiego1 = riegoOptimo.generarProgramacionesRiego(finca)
17        val programacionRiego = programacionRiego1(0)
18
19
20        println(s"Finca: $finca")
21        println(s"Distancia: $distancia")
22
23        // Programación de prueba
24
25    // Comparar costos secuenciales y paralelos
26      val resultados = riegoOptimo.compararCostoRiego(riegoOptimo.costoRiegoFinca,riegoOptimo.costoRiegoFincaPar)(finca, programacionRiego)
27
28      println(s"Tiempos comparados: Secuencial: ${resultados(0)} ms, Paralelo: ${resultados(1)} ms")
29      println(s"Speedup (Secuencial / Paralelo): ${resultados(2)}")
30
31      val resultados_2 = riegoOptimo.compararCostoMovilidad(
32      riegoOptimo.costoMovilidad,
33      riegoOptimo.costoMovilidadPar)(finca, programacionRiego, distancia)
34      println(s"Tiempos  movilidad comparados: Secuencial: ${resultados_2(0)} ms, Paralelo: ${resultados_2(1)} ms")
35      println(s"Speedup (Secuencial / Paralelo): ${resultados_2(2)}")
36
37      val resultados_3 = riegoOptimo.compararGeneracionesDeRiego(riegoOptimo.generarProgramacionesRiego, riegoOptimo.generarProgramacionesRiegoPar)(finca)
38      println(s"Tiempos  de generear todas las programaciones comparados: Secuencial: ${resultados_2(0)} ms, Paralelo: ${resultados_2(1)} ms")
39      println(s"Speedup (Secuencial / Paralelo): ${resultados_2(2)}")
40
41
42      val resultados_4 = riegoOptimo.compararProgramacionRiegoOptimo(riegoOptimo.ProgramacionRiegoOptimo, riegoOptimo.ProgramacionRiegoOptimoPar)(finca, distancia)
43      println(s"Tiempos  de programación óptima comparados: Secuencial: ${resultados_4(0)} ms, Paralelo: ${resultados_4(1)} ms")
44      println(s"Speedup (Secuencial / Paralelo): ${resultados_4(2)}")
45
46    }
47
48
```

```
PROBLEMAS   SALIDA   CONSOLA DE DEPURACIÓN   TERMINAL   PUERTOS   COMENTARIOS

Distancia: Vector(Vector(0, 15, 8, 15, 10), Vector(15, 0, 15, 8, 6), Vector(8, 15, 0, 5, 7), Vector(15, 8, 5, 0, 12), Vector(10, 6, 7, 12, 0))
Unable to create a system terminal
Unable to create a system terminal
Tiempos comparados: Secuencial: 0.2674 ms, Paralelo: 1.2295 ms
Speedup (Secuencial / Paralelo): 0.21748678324522164
Tiempos comparados: Secuencial: 0.2674 ms, Paralelo: 1.2295 ms
Speedup (Secuencial / Paralelo): 0.21748678324522164
Tiempos  movilidad comparados: Secuencial: 0.0285 ms, Paralelo: 0.6003 ms
Speedup (Secuencial / Paralelo): 0.04747626186906547
Tiempos  de generear todas las programaciones comparados: Secuencial: 0.0285 ms, Paralelo: 0.6003 ms
Speedup (Secuencial / Paralelo): 0.04747626186906547
Tiempos  movilidad comparados: Secuencial: 0.0285 ms, Paralelo: 0.6003 ms
Speedup (Secuencial / Paralelo): 0.04747626186906547
Tiempos  de generear todas las programaciones comparados: Secuencial: 0.0285 ms, Paralelo: 0.6003 ms
Speedup (Secuencial / Paralelo): 0.04747626186906547
Tiempos  de programación óptima comparados: Secuencial: 2.4197 ms, Paralelo: 13.8066 ms
wa: Ready                                                                    Lín. 53, col. 1   Espacios: 2   UTF-8
```

```
Speedup (Secuencial / Paralelo): 0.3201174743024963
Tiempos comparados: Secuencial: 0.4578 ms, Paralelo: 1.4301 ms
Speedup (Secuencial / Paralelo): 0.3201174743024963
Tiempos comparados: Secuencial: 0.4578 ms, Paralelo: 1.4301 ms
Speedup (Secuencial / Paralelo): 0.3201174743024963
Tiempos comparados: Secuencial: 0.4578 ms, Paralelo: 1.4301 ms
Speedup (Secuencial / Paralelo): 0.3201174743024963
Tiempos  movilidad comparados: Secuencial: 0.0336 ms, Paralelo: 0.7184 ms
Speedup (Secuencial / Paralelo): 0.04677060133630289
Tiempos  movilidad comparados: Secuencial: 0.0336 ms, Paralelo: 0.7184 ms
Speedup (Secuencial / Paralelo): 0.04677060133630289
Tiempos  movilidad comparados: Secuencial: 0.0336 ms, Paralelo: 0.7184 ms
Speedup (Secuencial / Paralelo): 0.04677060133630289
Tiempos  movilidad comparados: Secuencial: 0.0336 ms, Paralelo: 0.7184 ms
Speedup (Secuencial / Paralelo): 0.04677060133630289
```

```
Tiempos comparados: Secuencial: 0.2723 ms, Paralelo: 28.1426 ms
Speedup (Secuencial / Paralelo): 0.009675722925387134
```

```
Tiempos comparados: Secuencial: 0.7909 ms, Paralelo: 1.4817 ms
Speedup (Secuencial / Paralelo): 0.5337787676317743
```

```
Tiempos comparados: Secuencial: 0.2712 ms, Paralelo: 0.993 ms
Speedup (Secuencial / Paralelo): 0.2731117824773414
```

Comparación Costo Riego

Cantidad de tablones	Secuencial	Paralela	Aceleración
3	0.2723	28.1426	0.0096757229
4	0.7909	1.4817	0.533778
5	0.2674	1.2295	0.217486
8	0.2712	0.993	0.2731117
10	0.4578	1.4301	0.3211747

- Para 3 tablones, la ejecución paralela es extremadamente lenta (28.1426 frente a 0.2723 en secuencial).
- A medida que aumenta la cantidad de tablones, la paralelización mejora ligeramente, pero no de forma consistente

```
BUILD SUCCESSFUL in 17s
2 actionable tasks: 2 executed
Tiempos  movilidad comparados: Secuencial: 0.0873 ms, Paralelo: 0.9415 ms
Speedup (Secuencial / Paralelo): 0.09272437599575147
```

```
BUILD SUCCESSFUL in 17s
2 actionable tasks: 2 executed
Tiempos  movilidad comparados: Secuencial: 0.088 ms, Paralelo: 1.5586 ms
Speedup (Secuencial / Paralelo): 0.0564609264724753
```

```
BUILD SUCCESSFUL in 17s
2 actionable tasks: 2 executed
Tiempos  movilidad comparados: Secuencial: 0.1155 ms, Paralelo: 1.0618 ms
Speedup (Secuencial / Paralelo): 0.1087775475607459
```

```
Tiempos  movilidad comparados: Secuencial: 0.0931 ms, Paralelo: 2.3226 ms
Speedup (Secuencial / Paralelo): 0.04008438818565401
```

```
  Tiempos  movilidad comparados: Secuencial: 0.1 ms, Paralelo: 1.154 ms
  Speedup (Secuencial / Paralelo): 0.08665511265164645
```

Comparación Costo Movilidad

Cantidad de tablones	Secuencial	Paralela	Aceleración
3	0.0873	0.9415	0.092724
4	0.088	1.5586	0.05646092
5	0.1155	1.0618	0.10877754
6	0.0931	2.3226	0.040084
10	0.1	1.154	0.08665

- Para todas las configuraciones, el enfoque paralelo es más lento que el secuencial (aceleración siempre menor que 1).
- La aceleración es particularmente baja para 6 tablones (0.0401), indicando que la paralelización añade más sobrecarga que beneficios.

```
Tiempos  de generear todas las programaciones comparados: Secuencial: 0.6657 ms, Paralelo: 0.2655 ms
Speedup (Secuencial / Paralelo): 2.5073446327683615
```

```
Tiempos  de generear todas las programaciones comparados: Secuencial: 0.4538 ms, Paralelo: 0.5905 ms
Speedup (Secuencial / Paralelo): 0.7685012701100762
```

```
Tiempos  de generear todas las programaciones comparados: Secuencial: 0.8252 ms, Paralelo: 0.7129 ms
Speedup (Secuencial / Paralelo): 1.157525599663347
```

```
 Tiempos  de generear todas las programaciones comparados: Secuencial: 4.1429 ms, Paralelo: 0.5403 ms
 Speedup (Secuencial / Paralelo): 7.667777160836573
```

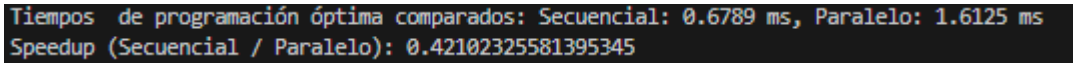
```
Tiempos  de generear todas las programaciones comparados: Secuencial: 90.2897 ms, Paralelo: 25.3785 ms
Speedup (Secuencial / Paralelo): 3.5577240577654314
```

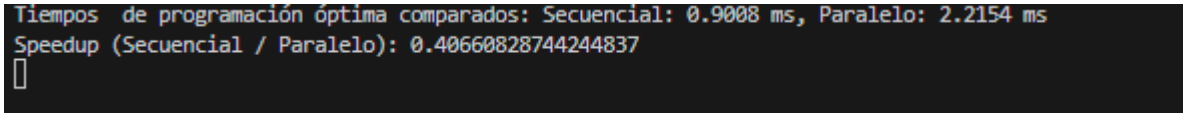
Comparación Generación de programación

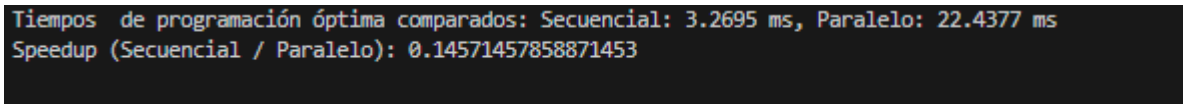
Cantidad de tablones	Secuencial	Paralela	Aceleración
3	0.6657	0.2655	2.507 ms
4	0.4538	0.5905	0.7685 ms
5	0.8252	0.7129	1.157525 ms

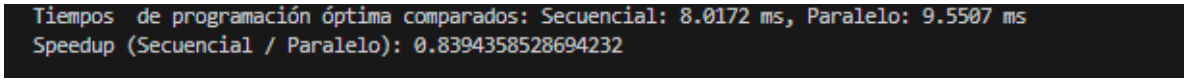
6	4.1429	0.5403	7.6677 ms
8	90.2897	25.3785	3.55772 ms

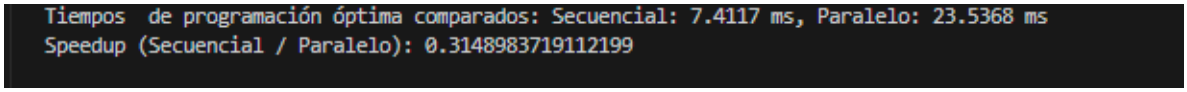
- En este caso, la paralelización sí resulta beneficiosa para conjuntos de datos más grandes:
- Para 3 tablones, la aceleración es 2.5x.
- Para 6 tablones, alcanza una aceleración notable de 7.67x.
- Para 8 tablones, la aceleración es de 3.56x.
- La mejora significativa en casos más grandes sugiere que el enfoque paralelo es efectivo cuando el tamaño del problema justifica la sobrecarga inicial.

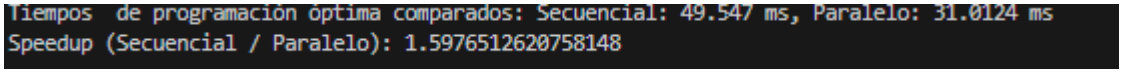












Comparación Generación programación Riego

Cantidad de tablones	Secuencial	Paralela	Aceleración
3	0.6789	1.6125	0.421
4	0.9008	2.2154	0.4066
5	3.2695	22.4377	0.1457
6	7.4117	23.5368	0.314898
7	49.547	31.0124	1.597

- En las primeras filas (3 a 6 tablones), el tiempo paralelo es consistentemente mayor que el secuencial, reflejado en aceleraciones menores a 1, lo cual indica que la paralelización no es efectiva en estos casos.
- Para 7 tablones, se observa un cambio notable: la paralelización supera al enfoque secuencial con un tiempo paralelo de 31.0124 frente a 49.547 secuencial, logrando una aceleración de 1.597.

Conclusiones

Este proyecto fue esencial para poner a prueba todos los conocimientos adquiridos durante el curso, en especial la paralelización tema importante que nos indica como se dividen las tareas para obtener una mejora en tiempo de un algoritmo

Comprendimos que la paralelización no tiene beneficios para todas las funciones, ni cantidad de datos, sino que existen en los cuales la mejor opción es utilizar la versión secuencial, que nos asegura un mejor rendimiento