

Informe del Proyecto

Integrantes del grupo:

Alejandra Osorio Giraldo 2266128

Luis Manuel Cardona Trochez 2059942

Jose David Marmol Otero 2266370

Informe de procesos

Se crearon los tipos:

```
// Definición de tipos
// Un tablon es una tripleta con tiempo de s
type Tablon = (Int, Int, Int)
// Una finca es un vector de tablones
type Finsa = Vector[Tablon]
// Una matriz de distancias entre tablones
type Distancia = Vector[Vector[Int]]
// Una programación de riego es un vector qu
type ProgRiego = Vector[Int]
// Tiempo de inicio de riego de cada tablón
type TiempoInicioRiego = Vector[Int]
```

La funcion fincaAlAzar:

```
def fincaAlAzar(long: Int): Finsa = {
  Vector.fill(long)((random.nextInt(long * 2) + 1, random.nextInt(long) + 1, random.nextInt(4) + 1))
}
```

DistanciaAlAzar:

```
def distanciaAlAzar(long: Int): Distancia = {
  val v = Vector.fill(long, long)(random.nextInt(long * 3) + 1)
  Vector.tabulate(long, long) { (i, j) =>
    if (i < j) v(i)(j)
    else if (i == j) 0
    else v(j)(i)
  }
}
```

Funciones para extraer datos de los tablonos:

```
def tsup(f: Finca, i: Int): Int = f(i)._1

def treg(f: Finca, i: Int): Int = f(i)._2

def prio(f: Finca, i: Int): Int = f(i)._3
```

Funciona para calcular el tiempo en que un tablon inicia su riego:

```
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
}
```

Funcion para ver la eficiencia de riego para un tablon:

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

Función que mapea los costos de riego de tablonos de una finca:

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

Funcion de los costos de movilidad:

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

Función que permuta diferentes permutaciones de riego

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

Función que elige el riego de menor costo de las permutaciones:

```
def programacionRiegoOptimo(f: Finca, d: Distancia): (ProgRiego, Int) = {  
  val programaciones = generarProgramacionesRiego(f)  
  val costos = programaciones.map(pi =>  
    (pi, costoRiegoFinca(f, pi) + costoMovilidad(f, pi, d))  
  )  
  costos.minBy(_._2) // Retorna la programación con el menor costo  
}
```

Informe de paralelización

Estrategia utilizada:

Mediante colecciones paralelas (.par) en funciones importantes a la hora de influir en el costo computacional, se logró aumentar la eficiencia mediante paralelización separando los subprocesos.

A continuación se puede ver las funciones paralelizadas:

```
// Calcula el costo de riego para toda la finca en paralelo  
def costoRiegoFincaPar(f: Finca, pi: ProgRiego): Int = {  
  (0 ≤ until < f.length).par.map(i => costoRiegoTablon(i, f, pi)).sum  
}  
  
// Calcula el costo de movilidad de forma funcional y paralela  
def costoMovilidadPar(f: Finca, pi: ProgRiego, d: Distancia): Int = {  
  (0 ≤ until < pi.length - 1).par.map(j => d(pi(j))(pi(j + 1))).sum  
}  
  
// Genera todas las posibles programaciones de riego de forma paralela  
def generarProgramacionesRiegoPar(f: Finca): Vector[ProgRiego] = {  
  val indices = (0 ≤ until < f.length).toVector  
  indices.permutations.toVector.par.toVector  
}  
  
// Encuentra la programación óptima de riego en paralelo  
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)  
}
```

Resultados obtenidos:

Se realizaron pruebas para 5 tamaños de entrada diferentes y iteraciones por tamaño. Las ganancias se analizaron utilizando la Ley de Amdahl:

Capturas de pantalla:

Tamaño de la finca (tablones)	Versión secuencial (ms)	Versión paralela (ms)	Aceleración (%)

Unable to create a system terminal			
1	0,29	0,25	13,92
3	1,27	0,76	40,72
5	24,58	10,01	59,26
6	24,41	16,70	31,59
9	1276,87	872,20	31,69

Ley de Amdahl: La mejora observada aumenta con el tamaño de la entrada, aunque la porción secuencial limita el beneficio total.

Casos de prueba

Se realizaron pruebas para validar la funcionalidad de cada método:

Se utilizaron los siguientes valores predefinidos:

```
type Tablon = (Int, Int, Int)
type Finca = Vector[Tablon]
type Distancia = Vector[Vector[Int]]

val finca: Finca = Vector((5, 2, 1), (6, 3, 2), (4, 1, 3))
val distancia: Distancia = Vector(
  Vector(0, 2, 3),
  Vector(2, 0, 1),
  Vector(3, 1, 0)
)
```

y se evaluaban los resultados esperados de las diferentes funciones de CalcularRiego:

```

test("generarProgramacionesRiego should generate all permutations") {
    val programaciones = calculosRiego.generarProgramacionesRiego(finca)
    assert(programaciones.size == 6) // 3! = 6 permutaciones
}

test("programacionRiegoOptimo should return the optimal irrigation schedule") {
    val (progRiego, costo) = calculosRiego.programacionRiegoOptimo(finca, distancia)
    assert(costo == 8) // Ejemplo de aserción
}

test("costoRiegoFinca should calculate the correct irrigation cost") {
    val progRiego = Vector(0, 1, 2) // Ejemplo de programación de riego
    val costo = calculosRiego.costoRiegoFinca(finca, progRiego)
    assert(costo >= 0) // Ejemplo de aserción
}

test("costoMovilidad should calculate the correct mobility cost") {
    val progRiego = Vector(0, 1, 2) // Ejemplo de programación de riego
    val costo = calculosRiego.costoMovilidad(finca, progRiego, distancia)
    assert(costo == 3) // Ejemplo de aserción
}
}

```

De igual forma para paralelo de CalcularRiego:

```

test("generarProgramacionesRiego should generate all permutations") {
    val programaciones = calculosRiegoPar.generarProgramacionesRiegoPar(finca)
    assert(programaciones.size == 6) // 3! = 6 permutaciones
}

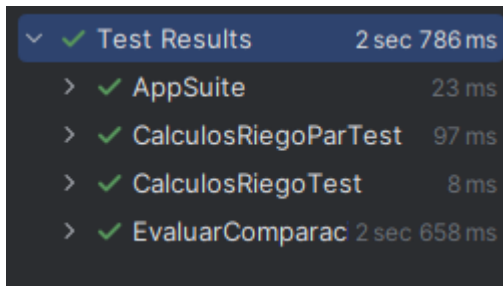
test("programacionRiegoOptimo should return the optimal irrigation schedule") {
    val (progRiego, costo) = calculosRiegoPar.programacionRiegoOptimoPar(finca, distancia)
    assert(costo == 14) // Ejemplo de aserción
}

test("costoRiegoFinca should calculate the correct irrigation cost") {
    val progRiego = Vector(0, 1, 2) // Ejemplo de programación de riego
    val costo = calculosRiegoPar.costoRiegoFincaPar(finca, progRiego)
    assert(costo == 14) // Ejemplo de aserción
}

test("costoMovilidad should calculate the correct mobility cost") {
    val progRiego = Vector(0, 1, 2) // Ejemplo de programación de riego
    val costo = calculosRiegoPar.costoMovilidadPar(finca, progRiego, distancia)
    assert(costo == 3) // Ejemplo de aserción
}
}

```

Finalmente comprobamos que los diferentes test pasaran exitosamente:



✓ Test Results	2 sec 786 ms
> ✓ AppSuite	23 ms
> ✓ CalculosRiegoParTest	97 ms
> ✓ CalculosRiegoTest	8 ms
> ✓ EvaluarComparac	2 sec 658 ms

Conclusiones

Código correcto y validado: Todas las funciones implementadas funcionan como se esperaba. Las pruebas unitarias confirmaron que cada método devuelve resultados precisos y consistentes para los casos evaluados.

Paralelización eficiente: Al usar colecciones paralelas (`.par`), logramos que las funciones más pesadas en términos computacionales se ejecutarán de forma más rápida. Esta estrategia permitió aprovechar mejor los recursos y reducir los tiempos de procesamiento, especialmente con datos más grandes.

Resultados y análisis: Las pruebas demostraron que la mejora en el tiempo de ejecución crece con el tamaño de la entrada, como lo predice la **Ley de Amdahl**. Aunque hay una parte del código que sigue siendo secuencial, el beneficio obtenido es significativo.

Pruebas unitarias: Se llevaron a cabo varios tests con entradas conocidas para verificar la precisión de todas las funciones clave del proyecto. Todos los tests pasaron exitosamente, lo que garantiza que el código es confiable.