Proyecto de curso

Simulador de polarización en redes

Fundamentos de programación funcional y concurrente

Escuela de Ingeniería de Sistemas y Compoutación

Profesor Juan Francisco Díaz Frias Monitora Emily Núñez

Noviembre de 2024

1. Introducción

El presente proyecto tiene por objeto observar el logro de los siguientes resultados de aprendizaje del curso por parte de los estudiantes:

- Utiliza un lenguaje de programación funcional y/o concurrente, usando las técnicas adecuadas, para implementar soluciones a un problema dado.
- Aplica conceptos fundamentales de la programación funcional y concurrente, utilizando un lenguaje de programación adecuado como SCALA, para analizar un problema, modelar, diseñar y desarrollar su solución.
- Construye argumentaciones formalmente correctas, usando las técnicas de argumentación apropiadas, para sustentar la corrección de programas funcionales y para sustentar la mejoría en el desempeño de programas concurrentes frente a las soluciones secuenciales.
- Trabaja en equipo, desempeñando un rol específico y llevando a cabo un conjunto de actividades, usando mecanismos de comunicación efectivos con sus compañeros y el profesor, para desarrollar un proyecto de curso.

Para ello el estudiante:

- Desarrolla un programa funcional concurrente utilizando un lenguaje de programación adecuado como SCALA para resolver en grupo un proyecto de programación planteado por el profesor.
- Escribe un informe de proyecto, presentando los aspectos más relevantes del desarrollo realizado, para que un lector pueda evaluar el proyecto.
- Desarrolla una presentación digital, con los aspectos más relevantes del desarrollo realizado, para sustentar el trabajo ante los compañeros y el profesor.

Adicionalmente el estudiante:

- Desarrolla programas funcionales puros con estructuras de datos inmutables utilizando recursión, reconocimiento de patrones, mecanismos de encapsulación, funciones de alto orden e iteradores para resolver problemas de programación.
- Combina la programación funcional con la POO entendiendo las limitaciones y ventajas de cada enfoque para resolver problemas de programación.
- Razona sobre la estructura de programas funcionales utilizando la inducción como meca-

nismo de argumentación para demostrar propiedades de los programas que construye.

- Paraleliza algortimos escritos en estilo funcional usando técnicas de paralelización de tareas y datos para lograr acelerar sus tiempos de ejecución.
- Razona sobre programas con paralelismo en datos y paralelismo en tareas en un contexto funcional para concluir sobre las ganancias en tiempo con respecto a las versiones secuenciales.
- Aplica técnicas de análisis de desempeño de programas paralelos a los programas funcionales paralelizados para concluir sobre el grado de aceleración de los tiempos de ejecución con respecto a las versiones secuenciales.
- Utiliza colecciones paralelas en la escritura de programas para lograr mejoras de desempeño en su ejecución.

2. Simulador de polarización en redes

Con el advenimiento del internet y de todo el entramado de tecnologías y negocios a su alrededor, la población mundial está mas conectada y con mayor acceso a información que nunca antes. Esto lleva a que prácticamente cualquier persona en el mundo se forma su opinión sobre cualquier tema y no haya temas vedados para nadie. Sin importar la formación, en la reciente pandemia del covid19 casi todos (por no decir todos) los seres humanos sobre la tierra tuvimos que decidir si nos aplicábamos unas vacunas o no. Y, para tomar esa decisión, cada persona definía cuáles eran sus fuentes creíbles de opinión, y a partir de esas opiniones y su propia información y educación, decidía si se vacunaba o no. El mundo entero se dividió entre los pro-vacunas y los anti-vacunas. Podemos decir que el mundo entero se polarizó ante ese tema.

De la misma forma, ante cualquier decisión que una persona deba tomar (por quién votar para presidente, cuál es el mejor lugar para ir de vacaciones, comprar vivienda propia o no, ahorrar o endeudarse, quién es el mejor jugador de fútbol o de tenis o de baloncesto de todos los tiempos, o quién es el mejor cantante de salsa, comprar un carro eléctrico o no, ...), las personas se informan y escuchan a sus referentes o a los influenciadores de su preferencia y deciden. Y muy probablemente su decisión sea más difícil cuanto más polarización produzca el tema objeto de decisión.

Pero, ¿Qué es la polarización? ¿Existen mecanismos objetivos para observar la polarización? ¿Para medirla? ¿Se pueden tomar acciones que cambien, en el tiempo, la tendencia a la polarización, sobre todo cuando ésta puede conducir a enfrentamientos que causan grandes daños?

Actualmente, el grupo de investigación AVISPA, desarrolla el proyecto titulado *Investigación en Modelos Computacionales de Redes Sociales Aplicados a la Polarización en el Valle del Cauca* (cuyo acrónimo es PROMUEVA por Polarización en Redes sOciales con un Modelo Unificado para El VAlle) cuyo objetivo principal es desarrollar modelos computacionales confiables de redes sociales para el análisis de la polarización.

Como punto de partida de este proyecto está el artículo *A Multi-Agent Model for Polarization under Confirmation Bias in Social Networks* que será puesto a disposición de los estudiantes del curso que quieran profundizar sobre el tema. En este artículo se presenta un modelo computacional para hacer análisis de polarización en redes. Este artículo está acompañado de una implementación denominada Tool for measuring group polarization and running simulation y disponible si ustedes la quieren consultar en GitHub; está hecho en Python pero de manera secuencial, lo que restringe la simulación a relativamente pocos agentes, razón por la cual queremos hacer uno en Scala para aprovechar las oportunidades de paralelización. El objetivo de este proyecto es desarrollar un **simulador de polarización en redes** basados en el modelo allí presentado y que a continuación se explicará, en los detalles pertinentes para el proyecto de este curso.

2.1. ¿Cómo medir la polarización en general?: La medida del proyecto PROMUEVA

En 1994, Esteban y Ray publicaron el artículo *On the measurement of polarization* cuyo propósito era estudiar la polarización y proveer una teoría para medirla. Como resultado de sus estudios, propu-

sieron una medida de polarización que inicialmente adoptamos en PROMUEVA. Sin embargo, en nuestro concepto, esa medida no captura completamente la noción intuitiva de polarización, Por ello, en el equipo de trabajo, definimos otra medida de polarización, la cual denominamos *comete* y que describiremos a continuación.

La idea es que se tiene un conjunto $\mathcal{Y} = \{y_0 = 0, y_1, \ldots, y_{k-1} = 1\}$ de k valores ordenados en $[0,1]$ que representan la opinión que puede tomar una población sobre un tema (0 es una opinión muy negativa, 1 es una opinión muy positiva, y cualquier otro valor se interpreta en ese rango). Y se tiene $\pi = \{\pi_0, \pi_1, \ldots, \pi_{k-1}\}$ de k valores en $[0,1]$ que representan la probabilidad π_i que alguien de la población, al preguntarle su opinión, escoja el valor y_i, para $0 \leq i < k$. A la pareja $\Pi = (\pi, \mathcal{Y})$ se le denomina una distribución sobre $\mathcal{Y}$. Lo que haremos es definir una medida de la polarización inducida por una distribución, $\rho_{cmt}(\Pi)$, con la ayuda de dos parámetros $\alpha, \beta \geq 1{,}0$, de la siguiente manera:

$$\rho_{cmt}(\Pi) = min\{\rho^{\Pi}_{aux}(p) : p \in [0,1]\}$$

$$\rho^{\Pi}_{aux}(p) = \sum_{i=0}^{k-1} \pi_i^{\alpha} |y_i - p|^{\beta}$$

donde $\alpha, \beta \geq 1{,}0$. Típicamente, usamos $\alpha = 1{,}2$ y $\beta = 1{,}2$, pero se podrían variar.

2.1.1. Implementando en Scala la función de polarización *comete*

Para implementar esta medida de polarización se definen los siguientes tipos de datos:

```
type DistributionValues = Vector[Double]
// Tipo para los valores, reales, de una distribucion

type Frequency = Vector[Double]
// Pi_k es una frecuencia de longitud k
// si Pi_k.lenght=k, 0 <= Pi_k(i) <= 1, 0<=i<=k-1
// Pi_k.sum == 1

type Distribution = (Frequency,DistributionValues)
// (Pi, dv) es una distribucion si Pi es una Frecuencia
// y dv son los valores de distribucion y Pi y dv
//son de la misma longitud

  type MedidaPol = Distribution => Double
```

Nótese que los tipos $DistributionValues$ y $Frequency$ son realmente el mismo, pero mientras un vector de valores de distribución viene ordenado y el primer valor es 0,0 y el último es 1,0, los vectores de frecuencia toman valores reales entre 0 y 1 (no hay orden) y deben cumplir una propiedad (la suma de todos debe dar 1). Por eso los distinguimos: para ser conscientes de eso al usarlos. Además se define el tipo $Distribution$ para empaquetar en un solo valor tanto los valores de distribución como las probabilidades, y el tipo $MedidaPol$ para representar el tipo que debe tener cualquier medida de polarización.

Por ejemplo podemos definir los siguientes valores de tipo $Frequency$ con un vector $likert5$ de tipo $DistributionValues$:

```
val pi_max=Vector(0.5, 0.0, 0.0, 0.0, 0.5)
val pi_min=Vector(0.0, 0.0, 1.0, 0.0, 0.0)
val pi_der=Vector(0.4, 0.0, 0.0, 0.0, 0.6)
val pi_izq=Vector(0.6, 0.0, 0.0, 0.0, 0.4)
val pi_int1=Vector(0.0, 0.5, 0.0, 0.5, 0.0)
val pi_int2=Vector(0.25, 0.0, 0.5, 0.0, 0.25)
val pi_int3=Vector(0.25, 0.25, 0.0, 0.25, 0.25)
val pi_cons_centro=pi_min
val pi_cons_der=Vector(0.0, 0.0, 0.0, 0.0, 1.0)
val pi_cons_izq=Vector(1.0, 0.0, 0.0, 0.0, 0.0)

val likert5=Vector(0.0, 0.25, 0.5, 0.75, 1.0)
```

Ahora se desea implementar la medida de polarización *comete*. Como por definición, se necesita hallar un punto $p \in [0,1]$ donde $\rho^{\Pi}_{aux}(p)$ sea mínimo, se necesita un método para hacer eso eficientemente. Se aprovechará que se sabe que la función $\rho^{\Pi}_{aux}(p)$ es convexa en el intervalo $[0,1]$.

Primero que todo, implemente la función `min_p` que recibe una función $f : Double => Double$, y dos reales, $min, max : Double$ denotando los límites inferior y superior de un intervalo donde f es convexa, y $prec \in (0,0{,}1]$ un real denotando la precisión deseada de la respuesta, y devuelve el punto $p \in [min, max]$ donde $f(p)$ es mínimo.

```
def min_p(f:Double=>Double, min:Double, max:Double,
          prec:Double):Double = {
 // Devuelve el punto p en [min,max] tal que f(p) es minimo
 // Suponiendo que f es convexa
 // si max-min < prec, devuelve el punto medio de [min,max]
 }
```

Ahora sí, implemente la función `rhoCMT_Gen` que reciba los parámetros α y β y devuelva una función que reciba una distribución (es decir, una pareja con un vector de frecuencias y un vector de valores de distribución) y devuelva el valor de la medida de polarización de *comete* mencionada atrás, parametrizada en los valores α y β.

```
def rhoCMT_Gen(alpha:Double, beta:Double):MedidaPol = {
 // Dados alpha y beta devuelve la funcion que calcula
 // la medida comete parametrizada en alpha y beta
 }
```

Para los vectores de frecuencias y valores de distribución definidos arriba, el resultado de aplicar esta función se vería así:

```
val cmt1 = rhoCMT_Gen(1.2, 1.2)

cmt1(pi_max,likert5)
cmt1(pi_min,likert5)
cmt1(pi_der,likert5)
cmt1(pi_izq,likert5)
cmt1(pi_int1,likert5)
cmt1(pi_int2,likert5)
cmt1(pi_int3,likert5)
cmt1(pi_cons_centro,likert5)
cmt1(pi_cons_der,likert5)
cmt1(pi_cons_izq,likert5)
```

```
val cmt1: Comete.MedidaPol = Comete.package<function>

val res0: Double = 0.379
val res2: Double = 0.0
val res4: Double = 0.327
val res6: Double = 0.327
val res8: Double = 0.165
val res10: Double = 0.165
val res12: Double = 0.237
val res14: Double = 0.0
val res16: Double = 0.0
val res18: Double = 0.0
```

Para efectos de poder comparar diferentes distribuciones, es importante tener una medida normalizada, por ejemplo, que siempre de un resultado en $[0, 1]$.

Una forma de normalizar una medida, es dividir su resultado, por la medida del peor caso (donde la polarización, en este caso, da máxima). Se sabe que el peor caso de polarización es cuando la probabilidad de que una opinión sea extrema (o sea, la opinión sea 0 o 1) es de 0.5. Y cualquier otro valor tiene probabilidad 0. Entonces la manera de normalizar nuestras medidas de polarización consistirá en hallar la polarización de la distribución deseada, y dividirla por la polarización del peor caso. Eso dará un número en $[0, 1]$ correspondiente a la medida normalizada.

Implemente la función `normalizar` que dada una medida de polarización devuelva la función que mide la misma polarización, pero normalizada.

```
def normalizar(m:MedidaPol):MedidaPol={
  // Recibe una medida de polarizacion, y devuelve la
  // correspondiente medida que la calcula  normalizada
  }
```

Para los vectores de frecuencias y valores de distribución definidos arriba, el resultado de aplicar esta función se vería así:

```
val cmt1_norm = normalizar(cmt1)

cmt1_norm(pi_max,likert5)
cmt1_norm(pi_min,likert5)
cmt1_norm(pi_der,likert5)
cmt1_norm(pi_izq,likert5)
cmt1_norm(pi_int1,likert5)
cmt1_norm(pi_int2,likert5)
cmt1_norm(pi_int3,likert5)
cmt1_norm(pi_cons_centro,likert5)
cmt1_norm(pi_cons_der,likert5)
cmt1_norm(pi_cons_izq,likert5)
```

```
val cmt1_norm: Comete.MedidaPol = Comete.package<function>

val res1: Double = 1.0
val res3: Double = 0.0
val res5: Double = 0.863
val res7: Double = 0.863
val res9: Double = 0.435
val res11: Double = 0.435
val res13: Double = 0.625
val res15: Double = 0.0
val res17: Double = 0.0
val res19: Double = 0.0
```

2.2. ¿Cómo medir la polarización en una red?: Elementos estáticos del modelo

En una red, supondremos que hay una población compuesta por un conjunto de n agentes, $\mathcal{A} = \{0, 1, \ldots, n-1\}$. Y que estamos interesados en conocer la polarización que genera entre los agentes la opinión sobre una proposición *prop*.

Cada agente tendrá una opinión sobre *prop*, representada por un número real entre 0 y 1. Es decir, *prop* es una proposición con la que cada agente tendrá un valor asociado que puede estar entre completamente en desacuerdo (en cuyo caso el valor asociado al agente será 0) y completamente de acuerdo (en cuyo caso el valor asociado al agente será 1). Modelaremos esta posición o creencia de cada agente con una función $b : \mathcal{A} \to [0, 1]$ tal que $0 \leq b(i) \leq 1$ representa la creencia instantánea del agente i en la proposición *prop* (usamos b por *belief* que significa creencia en inglés). Las creencias pueden cambiar con el tiempo, pero estos cambios se modelarán en la siguiente sección.

Por último, cada función de creencia b, tendrá asociada un medida de la polarización, $\rho(b)$, que ella genera. Es decir, si $B = \{b | b : \mathcal{A} \to [0, 1]\}$ es el conjunto de todas las posibles creencias, $\rho : B \to \mathbb{R}$, es una medida de polarización. Puede haber muchas maneras de medir la polarización, pero en este proyecto utilizaremos la medida ρ_{cmt} presentada en la sección anterior.

Para ello vamos a convertir una creencia $b : \mathcal{A} \to [0, 1]$ en una distribución $d^b = (\Pi^b, \mathcal{Y}^b)$ de manera que

$$\rho(b) = \rho_{cmt}(d^b)$$

es la polarización generada por b.

La construcción de $(\Pi^b, \mathcal{Y}^b)$ a partir de b, intuitivamente, consiste en hacer una partición del intervalo $[0, 1]$ en k subintervalos a partir de los k

valores de distribución deseados, y mirar qué proporción de los agentes tienen sus creencias en cada intervalo (con eso se construye Π^b) y tomar como valor representativo de cada intervalo el valor de distribución correspondiente (con eso se construye $\mathcal{Y}^b$).

Sea $d = (d_0, d_1, \ldots, d_{k-1})$ los valores de distribución posibles ($d_0 = 0{,}0$, $d_{k-1} = 1{,}0$ y $d_0 < d_1 < \ldots d_{k-1}$). A partir de d definimos los siguientes k intervalos:

- $I_0 = [0{,}0, \frac{d_1}{2}) = [d_0, \frac{d_1}{2})$
- $I_{k-1} = [\frac{d_{k-2}+1}{2}, 1{,}0] = [\frac{d_{k-2}+d_{k-1}}{2}, d_{k-1}]$
- $I_i = [\frac{d_i - d_{i-1}}{2}, \frac{d_i + d_{i+1}}{2})$ si $0 < i < (k-1)$

Dados d y b definimos Π^b y $\mathcal{Y}^b$ de la siguiente manera:

- $\Pi^b_i = \frac{|\{a \in \mathcal{A} : b(a) \in I_i\}|}{|\mathcal{A}|}$, para $0 \leq i \leq k$. Es decir Π^b_i es la fracción de agentes cuya creencia se ubica en el intervalo I_i.
- $\mathcal{Y}^b_i = d_i$, para $0 \leq i < k$

Ahora, la polarización de b con los valores de distribución d se calcula usando la medida de polarización *comete*:

$$\rho(b) = \rho_{cmt}(\Pi^b, \mathcal{Y}^b)$$

2.2.1. Implementando en Scala la función de polarización de una red

Para implementar la función de polarización ρ se definen los siguientes tipos de datos:

```
type SpecificBelief= Vector[Double]
// Si b:SpecificBelief, para cada i en Int,
//b[i] es un numero entre 0 y 1
// que indica cuanto cree el agente i
// en la veracidad de la proposicion p
// El numero de agentes es b.length
// Si existe i: b(i) <0 o b(i) > 1 b esta mal definida.
//  Para i en Int\A, b(i) no tiene sentido

type GenericBelief = Int => SpecificBeliefConf
// si gb:GenericBelief, entonces gb(n) = b
// tal que b:SpecificBelief
// es el tipo de las funciones generadoras de creencias

  type AgentsPolMeasure =
        (SpecificBelief,DistributionValues) => Double
// Si rho:AgentsPolMeasure y sb:SpecificBelief y
// d:DistributionValues,
// rho(sb,d) es la polarizacion de los agentes
// de acuerdo a esa medida
```

El tipo *SpecificBelief* define una creencia como un vector de dobles, es decir que no guarda la creencia tal cual como fue definida formalmente en esta sección (como una función), sino como un vector. Si b : *SpecificBelief*, entonces *b.length* es el número de agentes de la red.

El tipo *GenericBelief* se usará para generar creencias específicas, de forma genérica. Si gb : *GenericBelief*, entonces $gb(n) = b$ tal que b : *SpecificBelief* es una creencia específica para n agentes.

Por ejemplo, considere las siguientes creencias genéricas:

```
// Build uniform belief state.
def uniformBelief(nags:Int):SpecificBelief= {
  Vector.tabulate(nags)((i:Int) =>
                        (i+1).toDouble/nags.toDouble)
}

// Builds mildly polarized belief state, in which
//    half of agents has belief decreasing from 0.25, and
//    half has belief increasing from 0.75, all by the given step.
def midlyBelief(nags:Int):SpecificBelief= {
  val middle= nags/2
  Vector.tabulate(nags)((i:Int) =>
        if (i < middle) math.max(0.25 - 0.01*(middle-i-1),0)
        else math.min(0.75 - 0.01*(middle-i),1))
}

//Builds extreme polarized belief state, in which half
//    of the agents has belief 0, and half has belief 1.
def allExtremeBelief(nags:Int):SpecificBelief= {
  val middle= nags/2
  Vector.tabulate(nags)((i:Int) =>
                    if (i < middle) 0.0 else 1.0)
}

// Builds three-pole belief state, in which each
//    one third of the agents has belief 0, one third has belief 0.5,
//    and one third has belief 1.

def allTripleBelief(nags:Int):SpecificBelief= {
  val oneThird= nags/3
  val twoThird = (nags/3)*2
  Vector.tabulate(nags)((i:Int) =>
             if (i < oneThird) 0.0
             else if (i>=twoThird) 1.0
                  else 0.5)
}

// Builds consensus belief state, in which each
//    all agents have same belief.
def consensusBelief(b:Double)(nags:Int):SpecificBelief= {
  Vector.tabulate(nags)((i:Int) => b)
}
```

A partir de ellas podemos generar diferentes creencias específicas de prueba de 100 agentes:

```
val sb_ext=allExtremeBelief(100)
val sb_cons = consensusBelief(0.2)(100)
val sb_unif = uniformBelief(100)
val sb_triple = allTripleBelief(100)
val sb_midly = midlyBelief(100)
```

Ahora, el tipo *AgentsPolMeasure* define el tipo de una medida de polarización. Es decir, una función que dados una creencia específica, *sb* y unos valores de distribución, *dist*, devuelve la polarización de esos agentes, como respecto a esos valores de distribución.

Implemente la función `rho` que recibe los valores *alpha* y *beta* de configuración de la medida *comete* y devuelve la medida de polarización *comete* parametrizada en esos parámetros, normalizada para

agentes.

```
def rho(alpha:Double, beta:Double):AgentsPolMeasure= {
  // rho es la medida de polarizacion de agentes
  // basada en comete
 ...
}
```

Por ejemplo, para las creencias definidas arriba, podemos definir dos versiones de *comete* y dos valores de distribución para aplicar las medidas, así:

```
val rho1 = rho(1.2, 1.2)
val rho2 = rho(2.0,1.0)

val dist1 = Vector(0.0, 0.25, 0.50, 0.75, 1.0)
val dist2 = Vector(0.0, 0.2, 0.4, 0.6, 0.8, 1.0)

rho1(sb_ext,dist1)
rho2(sb_ext,dist1)
rho1(sb_ext,dist2)
rho2(sb_ext,dist2)

rho1(sb_cons,dist1)
rho2(sb_cons,dist1)
rho1(sb_cons,dist2)
rho2(sb_cons,dist2)

rho1(sb_unif,dist1)
rho2(sb_unif,dist1)
rho1(sb_unif,dist2)
rho2(sb_unif,dist2)

rho1(sb_triple,dist1)
rho2(sb_triple,dist1)
rho1(sb_triple,dist2)
rho2(sb_triple,dist2)

rho1(sb_midly,dist1)
rho2(sb_midly,dist1)
rho1(sb_midly,dist2)
rho2(sb_midly,dist2)
```

El resultado se verá así:

```
val rho1: Comete.AgentsPolMeasure =
     Comete.package<function>
val rho2: Comete.AgentsPolMeasure =
     Comete.package<function>

val dist1: scala.collection.immutable.Vector[Double] =
     Vector(0.0, 0.25, 0.5, 0.75, 1.0)
val dist2: scala.collection.immutable.Vector[Double] =
     Vector(0.0, 0.2, 0.4, 0.6, 0.8, 1.0)

val res29: Double = 1.0
val res30: Double = 1.0
val res31: Double = 1.0
val res32: Double = 1.0


val res33: Double = 0.0
val res34: Double = 0.0
val res35: Double = 0.0
val res36: Double = 0.0

val res37: Double = 0.38
val res38: Double = 0.188
val res39: Double = 0.377
val res40: Double = 0.172

val res41: Double = 0.617
val res42: Double = 0.448
val res43: Double = 0.617
val res44: Double = 0.448

val res45: Double = 0.784
val res46: Double = 0.58
val res47: Double = 0.773
val res48: Double = 0.528
```

2.3. ¿Cómo observar la evolución de la polarización en una red?: Elementos dinámicos del modelo

En el mundo real, la interacción con otras personas por diferentes vías (discusiones familiares, de conocidos o académicas, lectura de periódicos, lectura de artículos, revisión de videos, discusiones en las redes sociales, ...) puede producir cambios en nuestras percepciones ya sea de confirmación, reafirmación o debilitamiento, las cuales queremos reflejar en nuestro modelo. Por eso, las creencias de los agentes tienen que ser actualizadas con el paso del tiempo.

Para modelar la interacción con otros agentes, contaremos con una función de influencia, $\mathcal{I} : \mathcal{A} \times \mathcal{A} \to [0,1]$ tal que $\mathcal{I}(i,j) \in [0,1]$ representa la *influencia directa* que el agente i tiene sobre el agente j. Entre más alto este valor, mayor influencia tiene i sobre j. Este mismo valor también puede ser interpretado como la confianza que tiene j sobre la opinión de i. Suponemos que $\mathcal{I}(i,i) = 1$ para cada agente i, es decir, todo agente confía en sí mismo.

Otra forma de representar $\mathcal{I}$ sería como un grafo dirigido con pesos, cuyos nodos son los agentes, con una arista del agente i al agente j con peso $\mathcal{I}(i,j)$, si $\mathcal{I}(i,j) > 0$. A este grafo se le conoce como el *grafo de influencia*. Utilizaremos indistintamente función o grafo de influencia para referirnos a $\mathcal{I}$.

Una cosa es la influencia directa que tiene un agente i sobre un agente j, y otra es el *efecto global* que puede tener la influencia directa de i sobre j. De hecho, la creencia de una agente j cambia en el tiempo, producto de una combinación de factores entre los que se encuentran (1) la influencia de cada actor i sobre j, (2) su opinión actual, (3) la topología del grafo de influencia, (4) la forma como razonan los agentes.

Para modelar el *efecto global* que tiene el paso de una unidad de tiempo sobre las creencias de los agentes, se utilizará una función de actualización $\mu : B \times I \to B$, donde B es el conjunto de todas las creencias definido atrás, e $I = \{\mathcal{I} | \mathcal{I} : \mathcal{A} \times \mathcal{A} \to [0,1]\}$ es el conjunto de todos los grafos de influencia posibles; entonces $\mu(b, \mathcal{I}) \in B$ es la creencia resultante de la interacción entre los agentes según el grafo de influencia $\mathcal{I}$ a partir de la creencia original b.

Si llamamos b_0 la creencia inicial de una red de agentes con grafo de influencia $\mathcal{I}$, y $b_t = \mu(b_{t-1}, \mathcal{I})$ para $t \geq 1$, entonces tenemos que b_t es la creencia

de los agentes de la red, en el tiempo t, producida por el grafo de influencia $\mathcal{I}$.

Las funciones de actualización permiten modelar diferentes tipos de comportamientos sociales. Un comportamiento típico se llama el *sesgo de confirmación* (Confirmation Bias en inglés), que corresponde a que cada agente tiende a confiar más en aquellos que tienen percepciones cercanas a las suyas y a confiar menos en aquellos que tienen percepciones lejanas; su creencia se actualiza en función de los valores que tengan esos agentes que influyen sobre él. La función de actualización asociada al sesgo de confirmación se define así:

$$\mu^{CB} : B \times I \to B$$

, donde

$$\mu^{CB}(b, \mathcal{I}) = nb$$

y

$$nb(i) = b(i) + \frac{\sum_{j \in \mathcal{A}_i} \beta_{i,j} \mathcal{I}(j,i)(b(j) - b(i))}{|\mathcal{A}_i|}$$

para cada $i \in \mathcal{A}$, donde $\mathcal{A}_i = \{j \in \mathcal{A} | \mathcal{I}(j,i) > 0\}$ y $\beta_{i,j} = 1 - |(b(j) - b(i))|$.

2.3.1. Implementando en Scala la función de influencia

Para implementar la función de influencia $\mathcal{I}$ se definen los siguientes tipos de datos:

```
type WeightedGraph = (Int,Int) => Double

type SpecificWeightedGraph = (WeightedGraph, Int)

type GenericWeightedGraph =
      Int => SpecificWeightedGraph
```

El tipo $WeightedGraph$ permite definir una función de influencia tal cual como fue definida formalmente en esta sección, es decir como una función. Sin embargo para efectos de la programación, va a ser muy importante conocer específicamente cuántos agentes están siendo relacionados por esa función.

El tipo $SpecificWeightedGraph$ permite entonces tener una función de influencia específica, donde lo que la hace específica es que la función de influencia viene acompañada con el número de agentes medidos con esa creencia.

El tipo $GenericWeightedGraph$ es para definir funciones de influencia específicas, de forma genérica. Si $gsw : GenericWeightedGraph$, entonces $gsw(n) = sw$ tal que $sw : GenericWeightedGraph$ es una función de influencia específica.

Por ejemplo, considere las siguientes funciones de influencia, $i1\_10, i1\_20, i2\_10, i2\_20$ para 10 y 20 agentes:

```
def i1(nags:Int):SpecificWeightedGraph ={
  ((i:Int, j:Int) => if (i==j) 1.0
  else if (i<j) 1.0/(j-i).toDouble
  else 0.0,nags)
}

def i2(nags:Int):SpecificWeightedGraph ={
  ((i:Int, j:Int) => if (i==j) 1.0
  else if (i<j) (j-i).toDouble/nags.toDouble
  else (nags-(i-j)).toDouble/nags.toDouble, nags)
}

val i1_10=i1(10)
val i2_10=i2(10)
val i1_20=i1(20)
val i2_20=i2(20)
```

Para efectos de poder observar (no solamente calcular) el valor de la influencia de cada agente sobre los otros agentes embebida en una función de influencia específica, implemente una función `showWeightedGraph` que tome una función de influencia específica y la convierta en la matriz asociada al grafo de influencia (es decir en una secuencia indexada de secuencias indexadas por cada agente, con la influencia que él tiene sobre los otros agentes):

```
def showWeightedGraph(swg:SpecificWeightedGraph):
        IndexedSeq[IndexedSeq[Double]] = { ... }
```

Por ejemplo, para algunas de las creencias definidas arriba, el resultado de aplicar esta función se vería así:

```
scala> showWeightedGraph(i1_10)
val res23: IndexedSeq[IndexedSeq[Double]] = Vector(
Vector(1.0, 1.0, 0.5, 0.3333333333333333, 0.25, 0.2,
0.16666666666666666, 0.14285714285714285,
0.125, 0.1111111111111111),
Vector(0.0, 1.0, 1.0, 0.5, 0.3333333333333333, 0.25,
0.2, 0.16666666666666666, 0.14285714285714285,
0.125),
Vector(0.0, 0.0, 1.0, 1.0, 0.5, 0.3333333333333333,
0.25, 0.2, 0.16666666666666666, 0.14285714285714285),
Vector(0.0, 0.0, 0.0, 1.0, 1.0, 0.5, 0.3333333333333333,
0.25, 0.2, 0.16666666666666666),
Vector(0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.5,
0.3333333333333333, 0.25, 0.2),
Vector(0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.5,
0.3333333333333333, 0.25),
Vector(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.5,
0.3333333333333333),
Vector(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.5),
Vector(0.0, 0....

scala> showWeightedGraph(i2_10)
val res25: IndexedSeq[IndexedSeq[Double]] = Vector(
Vector(1.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9),
Vector(0.9, 1.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8),
Vector(0.8, 0.9, 1.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7),
Vector(0.7, 0.8, 0.9, 1.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6),
Vector(0.6, 0.7, 0.8, 0.9, 1.0, 0.1, 0.2, 0.3, 0.4, 0.5),
Vector(0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 0.1, 0.2, 0.3, 0.4),
Vector(0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 0.1, 0.2, 0.3),
Vector(0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 0.1, 0.2),
Vector(0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 0.1),
Vector(0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0))
```

2.3.2. Implementando en Scala la función de actualización de una creencia de una red

Implemente la función `confBiasUpdate` que dados una creencia específica *sb* y una función de influencia específica *swg* devuelve la creencia específica correspondiente a aplicar la función de actualización de confirmación de sesgo sobre *sb* teniendo en cuenta el grafo de influencia *swg*:

```
def confBiasUpdate(b:SpecificBeliefConf,
    swg:SpecificWeightedGraph):
    SpecificBeliefConf = { ... }
```

Por ejemplo, actualizar algunas de las creencias definidas atrás con algunas de las funciones de influencia definidas acá, se ve así:

```
val sbu_10 = uniformBelief(10)
confBiasUpdate(sbu_10,i1_10)
rho1(sbu_10,dist1)
rho1(confBiasUpdate(sbu_10,i1_10),dist1)

val sbm_10 = midlyBelief(10)
confBiasUpdate(sbm_10,i1_10)
rho1(sbm_10,dist1)
rho1(confBiasUpdate(sbm_10,i1_10),dist1)
```

```
val sbu_10: Comete.SpecificBelief =
    Vector(0.1, 0.2, 0.3, 0.4, 0.5,
    0.6, 0.7, 0.8, 0.9, 1.0)
val res51: Comete.SpecificBelief =
    Vector(0.1, 0.155, 0.24333333333333332, 0.34, 0.44,
    0.5416666666666666, 0.6442857142857142, 0.7475,
    0.8511111111111112, 0.955)
val res52: Double = 0.383
val res53: Double = 0.38

val sbm_10: Comete.SpecificBelief =
    Vector(0.21, 0.22, 0.23, 0.24, 0.25,
    0.75, 0.76, 0.77, 0.78, 0.79)
val res54: Comete.SpecificBelief =
    Vector(0.21, 0.21505, 0.22343333333333334,
    0.23265, 0.2422, 0.6549825, 0.7069635714285715,
    0.733586875, 0.7523952777777778, 0.7677165833333334)
val res55: Double = 0.435
val res56: Double = 0.435
```

2.3.3. Simulando la evolución de la polarización de una red

Como vimos anteriormente, la evolución de la polarización de una red, es producto de la forma como las creencias se modifican con el paso del tiempo. Para modelar esto, se define un nuevo tipo de datos:

```
type FunctionUpdate =
(SpecificBeliefConf,SpecificWeightedGraph) =>
SpecificBeliefConf
```

para representar las diferentes funciones de actualización que se quieran usar.

Implemente la función `simulate` que recibe una función de actualización fu, una función de influencia específica swg, una creencia específica b_0 y un entero t que especifica las unidades de tiempo para la simulación, y devuelve la secuencia de creencias específicas correspondientes a cada unidad de tiempo:

```
def simulate(fu:FunctionUpdate,
    swg:SpecificWeightedGraph,
    b0:SpecificBeliefConf, t:Int) :
    IndexedSeq[SpecificBeliefConf] = { ... }
```

Un ejemplo de uso de esta función sería el siguiente:

```
for {
  b <- simulate(confBiasUpdate,i1_10,sbu_10,2)
} yield (showBeliefConf(b),rho1(b,dist1))

for {
  b <- simulate(confBiasUpdate,i1_10,sbm_10,2)
} yield (showBeliefConf(b),rho1(b,dist1))
```

El resultado es:

```
val res57: IndexedSeq[(IndexedSeq[Double], Double)] =
    Vector((Vector(0.1, 0.2, 0.3, 0.4, 0.5,
    0.6, 0.7, 0.8, 0.9, 1.0),0.383),
    (Vector(0.1, 0.155, 0.24333333333333332, 0.34,
    0.44, 0.5416666666666666, 0.6442857142857142,
    0.7475, 0.8511111111111112, 0.955),0.38),
    (Vector(0.1, 0.1290125, 0.196025, 0.2841225694444445,
    0.3813961111111111, 0.48329760802469135,
    0.5878150793650794, 0.693940040834042,
    0.8011168530066348, 0.9090163220112119),0.335))


val res58: IndexedSeq[(IndexedSeq[Double], Double)] =
    Vector((Vector(0.21, 0.22, 0.23, 0.24, 0.25,
    0.75, 0.76, 0.77, 0.78, 0.79),0.435),
    (Vector(0.21, 0.21505, 0.22343333333333334,
    0.23265, 0.2422, 0.6549825, 0.7069635714285715,
    0.733586875, 0.7523952777777778,
    0.7677165833333334),0.435),
    (Vector(0.21, 0.21253775125, 0.2184535025,
    0.22636104194444445, 0.2351477761111111,
    0.5621256548854495, 0.6482698745939853,
    0.6916919576095071, 0.7201830992736775,
    0.7416493294486789),0.377))
```

Acá recolectamos la secuencia de modificaciones de la creencia inicial, por 2 unidades de tiempo, y calculamos la polarización que va induciendo cada modificación.

2.4. Acelerando la simulación con paralelismo de tareas y de datos

Una vez terminada esta etapa del proyecto, donde usted ya ha programado las versiones secuenciales `min_p`, `rhoCMT_Gen`, `normalizar`, `rho`, `showWeightedGraph`, `confBiasUpdate` y `simulate`, queremos implementar las versiones paralelas de algunas de ellas para ver si se logran tiempos mucho

mejores para simulaciones.

Solamente se paralelizará la función de cálculo de la polarización de agentes `rho`, y la función de actualización de la opinión, `confBiasUpdate`. Para el ejercicio use paralelismo de datos y de tareas donde lo vea pertinente.

2.4.1. Paralelizando el cálculo de la medida de polarización de una red usando *Comete*

Implemente la función `rhoPar` que recibe los valores *alpha* y *beta* de configuración de la medida *comete* y devuelve la medida de polarización *comete* parametrizada en esos parámetros, normalizada para agentes, pero funcionando concurrentemente (con concurrencia de tareas y/o datos, o ambas). Sugerencia: no deje de usar la función `rhoCMT_Gen` definida previamente:

```
def rhoPar(alpha:Double, beta:Double):AgentsPolMeasure= {
 // devuelve  la medida de polarizacion de agentes
 // basada en comete normalizada para agentes,
 // pero funcionando concurrentemente
 ...
 }
```

Mire cómo puede acelerar el proceso de construcción de la distribución usando técnicas de paralelización, para luego usar `rhoCMT_Gen`.

2.4.2. Paralelizando el cálculo de la actualización de una creencia

Implemente la función `confBiasUpdatePar` que dados una creencia específica *sb* y una función de influencia específica *swg* devuelve la creencia específica, calculada de forma paralela, correspondiente a aplicar la función de actualización de confirmación de sesgo sobre *sb* teniendo en cuenta el grafo de influencia *swg*:

```
def confBiasUpdatePar(b:SpecificBeliefConf,
     swg:SpecificWeightedGraph):
     SpecificBeliefConf = { ... }
```

Además de paralelización de datos, en este caso hay una oportunidad para usar paralelización de tareas.

2.4.3. Produciendo datos para hacer la evaluación comparativa de las versiones secuencial y concurrente

Un punto esencial en estos proyectos de paralelización, es realizar el análisis comparativo y concluir en qué casos es beneficioso usar la paralelización y en qué casos no. Para ellos es importante hacer muchas tablas con los tiempos que toman los dos tipos de versiones (secuencial y paralela) para:

- El cálculo de la polarización de una red, es decir `rho` y `rhoPar`.
- El cálculo de la actualización de la creencia de una red, es decir `confBiasUpdate` y `confBiasUpdatePar`.
- El cálculo de una simulación completa. ¿Para qué tamaño de redes, medida en número de agentes, y por cuántas unidades de tiempo es viable hacer simulaciones? ¿En secuencial y en paralelo?

Utilice *org.scalameter* y expresiones for para generar datos para hacer los análisis mencionados. Comente claramente cómo genera los datos de análisis, y preséntelos en el informe tabulados en tablas, de manera que después pueda analizarlas y sacar conclusiones relevantes.

Como complemento a esta tarea, les adicionamos un paquete denominado *Benchmark* donde encontrarán tres funciones que les ayuden en esta tarea de análisis: `compararMedidasPol` , `compararFuncionesAct` y `simEvolucion`.

La función `compararMedidasPol` recibe una secuencia de medidas de cualquier tamaño cada una, unos valores de distribución, y las dos medidas de polarización que se desean comparar (la secuencial y la concurrente) y devuelve séxtuplas con el tamaño de cada secuencia de opiniones de los agentes, la polarización calculada con una u otra medida (deben ser iguales) y los tiempos y la aceleración del cálculo de la medida 2 con respecto a la 1.

Por ejemplo. Si se ejecuta lo siguiente:

```
val likert5=Vector(0.0, 0.25, 0.5, 0.75, 1.0)
val sbms = for {
  n <-2 until 16
  nags = math.pow(2,n).toInt
} yield midlyBelief(nags)

val polSec = rho(1.2, 1.2)
val polPar = rhoPar(1.2, 1.2)

val cmp1 = compararMedidasPol(sbms,likert5, polSec, polPar)
```

En *cmp*1 quedan las comparaciones de la versiones secuencial y concurrente sobre cada una de las creencias iniciales guardadas en *sbms*. Si consultamos, por ejemplo, la aceleración en cada caso, el resultado da algo como:

```
scala> cmp1.map(t => t._6)
val res0: Seq[Double] = Vector(1.057380516824363,
   0.7183750438214572, 0.5891921983016022,
   0.7712349017297251, 1.350304266176467,
   0.7180345882499845, 1.2385172162003133,
   0.5845178844891709, 0.8262313691036176,
   0.6474803925923814, 2.67843300604095,
   1.5790304792812537, 0.8367098676799798,
   2.4340950830330326)
```

La función `compararFuncionesAct` recibe una secuencia de medidas de cualquier tamaño cada una, un grafo de influencia del tamaño del más grande vector de opiniones, , y las dos funciones de actualización que se desean comparar (la secuencial y la concurrente) y devuelve cuádruplas con el tamaño de cada secuencia de opiniones de los agentes, y los tiempos y la aceleración del cálculo de la función 2 con respecto a la 1.

Por ejemplo. Si se ejecuta lo siguiente:

```
val i1_32768=i1(32768)
val i2_32768=i2(32768)
compararFuncionesAct(sbms.take(sbms.length/2),
     i2_32768,confBiasUpdate, confBiasUpdatePar)
```

El resultado es algo como lo siguiente:

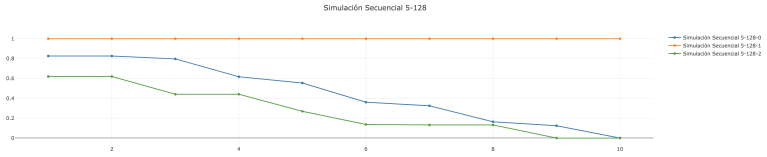
```
scala> val res7: Seq[(Int, org.scalameter.Quantity[Double],
org.scalameter.Quantity[Double], Double)] =
 Vector((4,0.059195 ms,0.569457 ms,0.10394990315335487),
 (8,0.020548 ms,1.335456 ms,0.01538650468454221),
 (16,0.038301 ms,3.539373 ms,0.010821408198570765),
 (32,0.139544 ms,9.019301 ms,0.01547170894950728 4),
 (64,0.786524 ms,14.442071 ms,0.05446061025458191),
 (128,1.344545 ms,27.818058 ms,0.04833353212506783),
 (256,4.386377 ms,69.628185 ms,0.06299714691686994))
```

La función `simEvolucion` permite simular la evolución de la opinión de los agentes, a partir de varias posibles opiniones iniciales, al aplicarle a la opinión inicial, la función de actualización de opinión, por el período de tiempo indicado al simulador. Lo interesante de esta función es que genera un archivo html donde se pueden ver gráficamente la evolución de la polarización a partir de cada una de las posibles opiniones iniciales.

Por ejemplo. Si se ejecuta lo siguiente:

```
val evolsSec = for {
  i <- 0 until sbms.length
} yield simEvolucion(Seq(sbms(i),sbes(i),sbts(i)),
          i2_32768,10,polSec,confBiasUpdate,likert5,
  "Simulacion_Secuencial_" ++ i.toString ++ "-"
  ++ sbms(i).length.toString)
```

El resultado son 13 archivos html que se ven como se muestra en la siguiente gráfica:



3. Técnicas utilizadas

Todo grupo de trabajo es libre de diseñar las funciones de la manera que considere más adecuada e implementar dicho modelo con el algoritmo que desee, siempre y cuando se ajuste al estilo de programación del paradigma de programación funcional y a las técnicas de programación vistas en clase.

Para lograr soluciones más eficientes en tiempo, deberán usar las técnicas de paralelización vistas en clase y hacer las evaluaciones comparativas correspondientes.

4. Entrega

Para el desarrollo del proyecto, se sugiere trabajar en el siguiente orden:

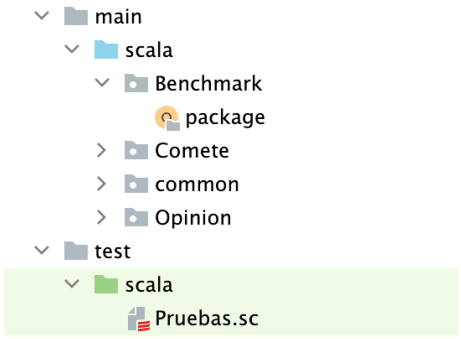
1. Construir una solución secuencial funcional de cada una de las funciones especificadas, y argumentar sobre su corrección.

2. A partir de esas soluciones, construir las soluciones concurrentes, y argumentar sobre su corrección.

3. Hacer un análisis comparativo de las dos soluciones (las secuenciales y las paralelas) para concluir sobre la pertinencia o no de paralelizar la solución.

4. Escribir un informe que dé cuenta de los aspectos que se quieren evidenciar como resultados de aprendizaje.

Usted deberá entregar dos paquetes `Comete` y `Opinion`, los cuales harán parte de su estructura de proyecto `IntellijIdea`, como se muestra en la figura a continuación, junto con los paquetes `common` y `Benchmark` provistos para el proyecto.



4.1. Paquete *Comete*

Las funciones `min_p`, `rhoCMT_Gen`, `normalizar` deben ser implementadas en un paquete de Scala denominado `Comete`. En ese paquete debe venir un archivo denominado `package.scala` que debe tener la forma siguiente:

```
package object Comete {
  type DistributionValues = Vector[Double]
  // Tipo para los valores, reales ordenados entre 0 y 1,
  // incluidos 0 y 1, de una distribucion

  type Frequency = Vector[Double]
  // Pi_k es una frecuencia de longitud k
  // si Pi_k.lenght=k, 0 <= Pi_k(i) <= 1, 0<=i<=k-1
  // y Pi_k.sum == 1

  type Distribution = (Frequency,DistributionValues)
  // (Pi, dv) es una distribucion si pi es una Frecuencia
  // y dv son los valores de distribucion y pi y dv son
  // de la misma longitud

  type PolMeasure = Distribution => Double


  def min_p(f:Double=>Double, min:Double, max:Double,
       prec:Double):Double = {
    // Devuelve el punto p en [min,max] tal que f(p) es minimo
    // Suponiendo que f es concava
    // si max-min < prec, devuelve el punto medio de [min,max]
...
  }

  def rhoCMT_Gen(alpha:Double, beta:Double):PolMeasure = {
    // Dados alpha y beta devuelve la funcion que calcula la medi
    // comete parametrizada en alpha y beta
...
  }

  def normalizar(m:PolMeasure):PolMeasure={
    // Recibe una medida de polarizacion, y devuelve la
    // correspondiente medida que la calcula  normalizada
...
}
```

4.2. Paquete *Opinion*

Las funciones `rho`, `showWeightedGraph`, `confBiasUpdate`, `simulate` , `rhoPar` y `confBiasUpdatePar` deben ser implementadas en un paquete de Scala denominado `Opinion`. En ese paquete debe venir un archivo denominado `package.scala` que debe tener la forma siguiente:

```
import Comete._
import common._
import scala.collection.parallel.CollectionConverters._
package object Opinion {
  // Si n es el numero de agentes, estos se identifican
  // con los enteros entre 0 y n-1
  // O sea el conjunto de Agentes A es
  // implicitamente el conjunto {0,1,2, ..., n-1}

  // Si b:BeliefConf, para cada i en Int, b[i] es un numero
  // entre 0 y 1 que indica cuanto cree el agente i en
  // la veracidad  de la proposicion p
  // Si existe i: b(i) <0 o b(i) > 1 b esta mal definida

  type SpecificBelief = Vector[Double]
  // Si b:SpecificBelief, para cada i en Int, b[i] es un
  // numero entre 0 y 1 que indica cuanto cree el
  // agente i en  la veracidad de la proposicion p
  // El numero de agentes es b.length
  // Si existe i: b(i) <0 o b(i) > 1 b esta mal definida.
  //  Para i en Int\A, b(i) no tiene sentido

  type GenericBeliefConf = Int => SpecificBelief
  // si gb:GenericBelief, entonces gb(n) = b tal que
  // b:SpecificBelief

  type AgentsPolMeasure =
      (SpecificBelief,DistributionValues) => Double
  // Si rho:AgentsPolMeasure y sb:SpecificBelief
  // y d:DistributionValues,
  // rho(sb,d) es la polarizacion de los agentes
  // de acuerdo a esa medida

  def rho(alpha:Double, beta:Double):AgentsPolMeasure= {
    // rho es la medida de polarizacion de agentes basada
    // en comete
...
  }

  // Tipos para Modelar la evolucion de la opinion en una red

  type WeightedGraph = (Int,Int) => Double

  type SpecificWeightedGraph = (WeightedGraph, Int)

  type GenericWeightedGraph =
       Int => SpecificWeightedGraph

  type FunctionUpdate =
  (SpecificBelief,SpecificWeightedGraph)=> SpecificBelief

  def confBiasUpdate(sb:SpecificBelief,
       swg:SpecificWeightedGraph):SpecificBelief = {
...
  }

  def showWeightedGraph(swg:SpecificWeightedGraph):
       IndexedSeq[IndexedSeq[Double]] = {
...
  }

  def simulate(fu:FunctionUpdate,
         swg:SpecificWeightedGraph,
         b0:SpecificBelief,
         t:Int) : IndexedSeq[SpecificBelief] = {
...
  }

  // Versiones paralelas
  def rhoPar(alpha:Double,
       beta:Double):AgentsPolMeasure= {
    // rho es la medida de polarizacion de agentes basada
    // en comete
...
  }

  def confBiasUpdatePar(sb:SpecificBelief,
       swg:SpecificWeightedGraph):SpecificBelief = {
...
}
```

Estos paquetes deberán correr en conjunto. Las pruebas las haremos importando esos paquetes.

La fecha de entrega límite es el 28 de noviembre de 2024 a las 23:59. La sustentación será el 5 de diciembre de 2024.

Debe entregar un informe en formato pdf, los paquetes mencionados, un archivo Readme.txt que describa todos los archivos entregados y las instruc-

ciones para ejecutar los programas. Todo lo anterior en un solo archivo empaquetado cuyo nombre contiene los apellidos de los autores y cuya extensión corresponde al modo de compresión. Por ejemplo Diaz.zip o Diaz.rar, o Diaz.tgz o ...

5. Sustentación y calificación

El trabajo debe ser sustentado por los autores en día y hora especificados. La calificación del proyecto se hará teniendo en cuenta los siguientes criterios:

1. Informe (1/3)
   - Debe describir claramente las estructuras de datos utilizadas, tanto para las soluciones secuenciales como para las soluciones paralelas. No olvide señalar también qué colecciones paralelas fueron utilizadas.
   - Las funciones desarrolladas deben responder a programas funcionales puros. Las que no deben ser justificadas, por qué no.
   - En general, el código debe evidenciar el manejo de la recursión, del reconocimiento de patrones, de mecanismos de encapsulación, de funciones de alto orden, de iteradores, de colecciones y de expresiones for. El informe debe señalar dónde se usaron estos elementos.
   - El informe debe traer las argumentaciones sobre la corrección de las funciones desarrolladas. Por lo menos de las más relevantes (las funciones `min_p`, `rhoCMT_Gen` del paquete *Comete* y las funciones `rho`, `confBiasUpdate`, `simulate` del paquete *Opinion*).
   - El informe debe señalar dónde se usaron técnicas de paralelización de tareas y de datos, cuáles se usaron y qué impacto tuvieron en el desempeño del programa.
   - El informe debe traer las argumentaciones sobre las razones que permiten asegurar que las técnicas de paralelización usadas deberían tener un impacto positivo en el desempeño del programa.
   - El informe también debe evidenciar, con tablas, las evaluaciones comparativas realizadas entre las pruebas realizadas a las soluciones secuenciales y las paralelas, y concluir sobre el grado de aceleración logrado.
2. Implementación (1/2). Esto quiere decir que el código entregado funciona por lo menos para las diferentes pruebas que tendremos para evaluarlo.
3. Desempeño grupal en la sustentación (1/6), lo cual incluye la capacidad del grupo de navegar en el código y realizar cambios rápidamente en él, así como la capacidad de responder con solvencia a las preguntas que se le realicen.

Todo lo anterior está condensado en la rúbrica que se les comparte con el enunciado del proyecto.

En todos los casos la sustentación será pilar fundamental de la nota asignada. Cada persona de cada grupo, después de la sustentación, tendrá asignado un número real (el factor de multiplicación) entre 0 y 1, correspondiente al grado de calidad de su sustentación. Su nota definitiva será la nota del proyecto, multiplicada por ese valor. Si su asignación es 1, su nota será la del proyecto. Pero si su asignación es 0.9, su nota será 0.9 por la nota del proyecto. La no asistencia a la sustentación tendrá como resultado una asignación de un factor de 0.

La idea es que lo que no sea debidamente sustentado no vale así funcione muy bien!!!

Éxitos!!!