Simulador de Polarización en Redes

Jhorman Gomez[1], Ivan Ausecha[2], James Calero[3], Daniel Rojas[4]

2326867[1], -[2], -[3], -[4]

Universidad del Valle

Facultad de Ingeniería

Escuela de Ingeniería de Sistemas y Computación

Santiago de Cali, Octubre de 2024

1 ¿Son Nuestras Funciones Funcionalmente Puras?

Función	Funcional	¿Por qué no?
min_p	Si	-
rhoCMT_Gen	Si	-
normalizar	Si	-
rho	Si	-
confBiasUpdate	Si	-
simulate	Si	-
rhoPar	No	A pesar de que no hay efectos secundarios por el uso de paralelismo, la función no es funcionalmente pura por el no determinismo en el orden de realización de las operaciones dentro de esta.
confBiasUpdatePar	No	A pesar de que no hay efectos secundarios por el uso de paralelismo, la función no es funcionalmente pura por el no determinismo en el orden de realización de las operaciones dentro de esta.

2 Técnicas y Estructuras de Datos Utilizadas

2.1 min_p

La función **min_p** hace uso de:

- **Recursión:** genera un proceso de recursión lineal mediante el cual por cada iteración se aproxima más al punto mínimo de la función recibida.
- **Funciones de Alto Orden:** recibe una función como parámetro.
- **Colecciones:** genera un rango de valores a partir del min y max recibidos, a este rango se le aplica un map y al resultado del map se le aplica el método minBy, ambos métodos pertenecientes a colecciones.

2.2 rhoCMT_Gen

La función **rhoCMT_Gen** hace uso de:

- **Funciones de Alto Orden:** hace uso de funciones anónimas y retorna esta misma como resultado.

- **Colecciones:** hace uso del método zip para generar una colección de tuplas (Frecuency, DistributionValue) sobre la cual se define la función sum.

2.3 normalizar

La función **normalizar** hace uso de:

- **Funciones de Alto Orden:** hace uso de funciones anónimas y retorna esta misma como resultado.
- **Colecciones:** hace uso de vectores y en especial el método fill para generar una colección con la mayor polarización posible para una frecuencia de longitud k.

2.4 rho

La función **rho** hace uso de:

- **Funciones de Alto Orden:** hace uso de funciones anónimas y retorna esta misma como resultado.
- **Colecciones:** hace uso de colecciones y métodos característicos de estas para generar los intervalos sobre los cuales se van a clasificar los agentes y a su vez estos se clasifican haciendo uso de métodos como map y groupBy.
- **Iteradores:** hace uso de métodos que requieren de iteradores como ZipWithIndex y indexWhere.
- **Reconocimiento de Patrones:** hace uso de reconocimiento de patrones para el reconocimiento de los elementos que hacen parte de las colecciones utilizadas, esto con el fin de poder hacer transformaciones sobre dichas colecciones y en últimas clasificar los agentes y obtener la frecuencia final a partir de las opiniones de estos.

2.5 confBiasUpdate

La función **confBiasUpdate** hace uso de:

- **Funciones de Alto Orden:** recibe funciones como parámetro, en específico la función swg que corresponderia a la matriz de influencia.
- **Colecciones:** recibe una colección y en base a esta mediante el uso de map, se define la función sum sobre la que se va a realizar el respectivo update a las creencias de los agentes haciendo uso de un rango y mapeando dicha función para cada valor del rango.

2.6 simulate

La función **simulate** hace uso de:

- **Funciones de Alto Orden:** recibe funciones como parámetro, en específico las funciones correspondientes a la matriz de influencia (swg) y la función sobre la cual se van a actualizar las creencias de los agentes (fg).
- **Colecciones:** recibe una colección la cual se actualiza en base a fg y mediante estas actualizaciones se va creando una secuencia indexada donde cada elemento es la creencia de los agentes en el tiempo t empezando desde t=0.
- **Recursión:** genera un proceso de recursión lineal sobre el cual se actualiza t-1 veces la creencia recibida (sb).

3 Informe de Corrección

3.1 min_p

3.2 rhoCMT_Gen

3.3 rho

3.4 confBiasUpdate

3.5 simulate

4 Técnicas de Paralelización y Análisis de Impacto

4.1 Tipos de Paralelización Usados

Para las funciones **rhoPar** y **confBiasUpdatePar** se utilizó solamente paralelismo de datos. A continuación las razones:

- **rhoPar:** no hicimos uso de paralelismo de tareas porque abstracciones como **parallel** son solo útiles cuando se realizan procesos recursivos, además, la abstracción **task** realmente no tiene cabida dentro de nuestro algoritmo debido a que no es posible por ejemplo ir clasificando las creencias de los agentes antes de que todos los intervalos estén definidos.
- **confBiasUpdatePar:** se intentó introducir una versión paralelizada mediante paralelismo de tareas pero a la hora de realizar pruebas se llega al error stackOverFlow debido a la cantidad de memoria requerida por el programa. La versión paralelizada mediante paralelismo de tareas actual arroja resultados positivos donde se ve una aceleración en la ejecución del programa, por lo tanto, realmente no importó el detalle anteriormente mencionado.

A continuación el error obtenido para la versión por paralelismo de tareas en confBiasUpdate.

```
java.lang.StackOverflowError: java.lang.StackOverflowError
  at java.base/jdk.internal.reflect.NativeConstructorAccessorImp
  at java.base/jdk.internal.reflect.NativeConstructorAccessorImp
  at java.base/jdk.internal.reflect.DelegatingConstructorAccesso
  at java.base/java.lang.reflect.Constructor.newInstance(Constru
```

4.2 Impacto

5 Pruebas y Resultados

6 Conclusiones