Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Máster

Máster en Ciberseguridad e Inteligencia de los Datos

Desarrollo de un algoritmo de caminos mínimos biobjetivo sobre Spark

Developing a Spark bi-objective minimum shortest path algorithm

Autor: Adrián Prieto Curbelo
Tutor: Marcos Colebrook Santamaría
Cotutor: Antonio Sedeño Noda

La Laguna, 1 de marzo de 2021

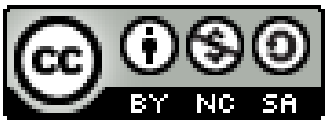
Agradecimientos

A lo largo de la elaboración de este trabajo, que ha tenido sus altibajos, he podido ir superando los obstáculos gracias al constante apoyo de muchas personas que me han hecho ver que con trabajo y dedicación se pueden conseguir los objetivos.

Quiero dar las gracias especialmente a mis tutores por su implicación en el presente Trabajo Fin de Máster, los conocimientos que me ha aportado y su constante apoyo.

También agradezco a mi familia que me ha demostrado que, aunque parezca que no se puede realizar todas las cosas que uno quiere, siempre existe alguna manera de conseguirlo, con uso de la constancia y de la motivación, y sobre todo, de no desfallecer en el intento.

Licencia



Resumen

En la actualidad, debido a la enorme cantidad de datos informáticos, son necesarios nuevos algoritmos que de manera eficiente puedan operar con esta gigantesca cantidad de datos, sobre todo si se trata de grafos. Sin embargo, aunque los métodos tradicionales parecen suficientes hoy en día, se siguen considerando nuevas ideas de tratar con dichos datos masivos.

Por este motivo, se quiere implementar una manera de realizar uno de los algoritmos más simples sobre grafos, la de encontrar caminos mínimos; de una forma distribuida, con el fin de incluir a más de un recurso de cómputo para un mismo problema.

Si bien ya existen actualmente tales algoritmos, lo que diferencia este proyecto de los anteriores, es el tratamiento y uso de dos o incluso más objetivos o distancias (de ahí el nombre de algoritmo de caminos mínimos biobjetivo), que va a ser cada vez más relevante y útil en el tratamiento de datos.

Palabras Clave

Spark, Scala, Programación distribuida, Camino mínimo de grafos, Biobjetivo

Abstract

Nowadays, and due to the huge amount of digital data, we need more algorithms that can manage and treat such big data, even with graphs. Although traditional methods are simply enough, we are considering more ideas to deal with such a massive size.

Because of this, we want to design one of the simplest graph algorithms, such as finding the shortest path on a graph, in a distributed way, and in order to assign more than a single computing resource working on the same problem.

Although such algorithms already exist, what differentiates this project from the previous ones is the treatment and use of two or even more objectives or distances (hence the name of bi-objective minimum path algorithm), which will be increasingly relevant and useful in the processing of such data.

Keywords

Spark, Scala, Distributed Programming, Optimization, Graph Pathfinding, Bi-objective

Índice general y contenido

Índice de figuras

1. Introducción

1.1. Contexto

La finalidad de este proyecto es desarrollar un proyecto en Scala que implemente un algoritmo de caminos mínimos biobjetivo, con el propósito que se pueda aplicar en un entorno distribuido.

1.2. Objetivos

Para este proyecto, se utilizan ejemplos de tamaño considerablemente grandes (grafos de aproximadamente 2 GBs), sobre los que se aplica el algoritmo de **Dijsktra Biobjetivo** para determinar todos los caminos eficientes u óptimos.

El principal objetivo de la implementación es realizar una modificación sobre los algoritmos ya existentes, con el fin de que operen sobre dos objetivos (es decir, dos distancias o atributos sobre el mismo arco), usando un lenguaje destinado a computación distribuida, como es Scala.

Una de las características que queremos alcanzar con este proyecto es que el prototipo desarrollado pueda ser efectivo en cuanto a rendimiento temporal y espacial, es decir, que pueda escalar adecuadamente sobre un grafos grandes, o que sea independiente del tamaño, en el menor tiempo posible.

Otra de las características, pero no tan importante como la anterior, es que pueda ser escalable a nivel del número de *workers* utilizado en el algoritmo. Es decir, que pueda ser efectivo incluso cuando el número de *workers* aumente.

2. Conceptos básicos

Para empezar a desarrollar tal proyecto, se ha partido en un algoritmo de cálculo de caminos mínimos biobjetivo (Dijkstra Biobjetivo) ya desarrollado en C, y usado como referencia tanto en las soluciones como en tiempo de ejecución.

2.1. Scala

Mientras que C produce código nativo para ejecutarse en una arquitectura de máquina particular, Scala es un nuevo lenguaje de programación funcional, que crea código para ejecutarse en una máquina virtual de Java (JVM).

Lo que hace que sea óptimo para la computación distribuida, ya que Scala es el lenguaje principal en el que se basa Spark.

2.2. Apache Spark como Computación distribuida

Apache Spark es un framework *open-source*, usado principalmente para la computación distribuida.

La computación distribuida permite que recursos independientes puedan trabajar sin tener que compartir recursos o competir por ellos, eliminando posibles conflictos.

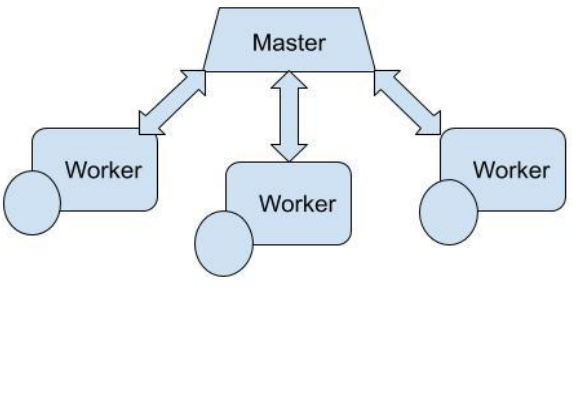


Figura 1 - Arquitectura de Apache Spark

Para ello, un recurso conocido como nodo *master* coordina cada uno del resto de los recursos o nodos *workers*, asignando su rutina de trabajo y los datos que necesitan para resolver tal problema. Cada *worker* realiza tareas y se comunica con el *master* u otros *workers* para continuar las tareas.

La idea que subyace es dividir el problema en trozos más pequeños de manera que cada *worker* pueda encargarse de parte del problema. Con esto podremos resolver problemas más extensos y más complejos, sumando memoria y CPU de los workers.

La comunicación puede realizarse directamente entre los *workers*, pero deben mantener una sincronía de ejecución que sería proporcionada por el *master*. Otra función del *master* es darse cuenta cuando se ha acabado el algoritmo y los *workers* pueden parar.

Las ventajas que esto supone son:

- Permite el escalado horizontal en vez de vertical (muchos ordenadores baratos o de segunda mano, contrastandolo con un solo ordenador con todas las prestaciones)
- Puede ser escalable: podemos añadir y sustraer *workers* de manera transparente al programa.
- Garantiza la alta disponibilidad: se podría configurar el sistema para el caso de que si algún ordenador *worker* fallase, no influya en el funcionamiento del programa.

2.2. Apache Spark GraphX y pregel

Cuando tratamos con algoritmos sobre grafos y queremos aplicar programación distribuida deberíamos abordar los siguientes aspectos:

- **La memoria:** cuando el problema es muy grande, no cabría el grafo y/o la solución buscada en un único nodo *worker* (ordenador). Esto implica la partición del grafo, ya sea por vértices o por arcos, y cada nodo *worker* solamente podría acceder a sus datos locales. Habría que estudiar particiones de tamaño fijo o variable, y ver si se realizan particiones por vértices o por arcos.
- **Comunicaciones:** habría que establecer el protocolo de comunicaciones entre los *worker* (modo síncrono o asíncrono).
- **CPU:** si la granja de nodos *worker* no es homogénea se puede esperar más rendimiento de un nodo respecto a otro.
- **Cálculo local y remoto:** cada *worker* debería trabajar en su partición todo lo posible y luego pasar los datos mínimos al resto de *workers* para continuar el trabajo.

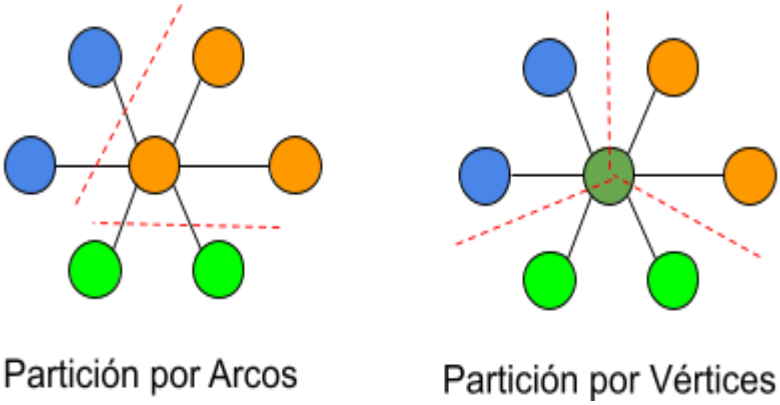


Figura 2 - Tipos de particiones de Grafos : por vértices o por arcos.

Con todo lo anterior, los algoritmos se complican demasiado y de un algoritmo al siguiente no se reaprovecha el código.

Ahí es donde surge un sistema de desarrollo de algoritmos denominado Pregel en el que la programación distribuida se lleva al extremo: a cada vértice del grafo se le asocia un *worker*. Cada *worker* solamente conoce la información de ese vértice (origen) y para cada arco que sale de ese nodo, la información del vértice al que llega (destino).

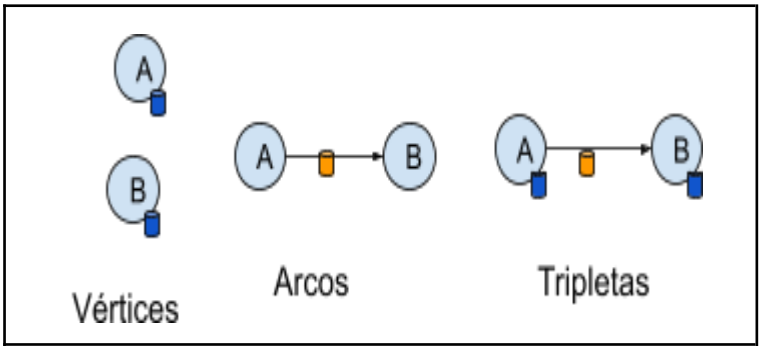


Figura 3 - Nociones de Grafos: Vértices (o nodos), Arcos (o aristas) y Tripletas.

Además se deja en manos del *framework* las labores de mapeo de *worker* de Pregel hacia nodos *worker* de computación, las labores de comunicación, sincronismo entre workers, etc.

Básicamente un algoritmo Pregel consiste en:

- Se construye el grafo.
- Se manda un mensaje inicial a cada *worker* (esto es, cada vértice del grafo).
- Se ejecuta el *SuperPaso* hasta que ningún *worker* reciba mensaje.

Dentro de tal *SuperPaso* consiste en que cada *worker* que reciba un mensaje ejecute las siguientes tareas:

- Recepción de mensajes: se recibe el mensaje y se ve como afecta al vértice.
- Construcción de mensajes: con la información local que posee, construye mensajes para los nodos pertinentes.
- Mezcla de mensajes: si hay varios mensajes para un nodo, volverlos a unir.

Hay que notar la diferencia entre el concepto de *worker* dentro del modelo Pregel y los nodos *worker* de la computación distribuida.

Apache Spark es un *framework open-source* para programación distribuida, y en concreto en las librerías GraphX se implementa el método Pregel. Este *framework* se encarga de distribuir los *workers* de Pregel en una arquitectura distribuida con nodos *master* y nodos *workers*.

2.3 Computación distribuida frente a la computación paralela

Cabe remarcar la diferencia entre computación paralela y computación distribuida:

- En la **computación paralela**, se forman varios hilos de ejecución, y se reduciría el tiempo de CPU. Hay que garantizar que varios hilos no modifican el mismo dato. No hay que preocuparse por la velocidad de comunicación entre los hilos pues todos acceden a datos que residen en memoria RAM.

  Seguimos teniendo el problema de la memoria: si todos los datos necesarios para el cálculo no caben en memoria no se puede resolver el problema. Otro problema de tal memoria compartida es la generación de posibles conflictos al acceder a mismos recursos, dando lugar a resultados inesperados.

- En la **computación distribuida**, en cambio, se trabaja con el concepto de la partición de los datos en varios subconjuntos más pequeños de manera que haya una red de ordenadores de *workers*, y cada uno de ellos reportando al *master* que los coordinará.

  La comunicación por defecto sería asíncrona, teniendo puntos de consistencia para el siguiente *SuperPaso*. Concepto de datos que han de viajar entre dos *worker*. Concepto de datos locales, cada *worker* conoce un conjunto de datos que nadie más maneja. Por supuesto, parece evidente que todos los algoritmos hay que rediseñarlos para este nuevo entorno.

3. Planificación y cronograma

Se muestran a continuación el plan de trabajo y el cronograma estimativo que se definieron para la realización del trabajo.

3.1. Plan de trabajo

1. Entender conceptos de grafos biobjetivo y cómo adaptarlos a la computación distribuida .

2. Tras recibir el algoritmo de C, traducir el algoritmo paso a paso.

3. Probar el algoritmo en Scala en un solo ordenador o *worker*. Si funciona, podemos probarlo en más de un *worker*.

4. Si es posible, mejoraremos aún más el algoritmo, o utilizamos alguna solución más “natural” al lenguaje (como podría ser con ayuda de alguna librería ya implementada)

3.2. Cronograma

Se ha empleado un total de 8 semanas. Para ello, se han planificado todas las actividades anteriores con la siguiente distribución:

1. Actividad 1 → 1 semana.

2. Actividad 2 → 3 semanas.

3. Actividad 3 → 2 semanas.

4. Actividad 4 → 2 semanas.

4. Implementaciones

Se han desarrollado dos aproximaciones, como se ha mencionado en el cronograma.

- Traducir un **algoritmo biobjetivo ya existente** desde C a Scala, para utilizarlo después en computación distribuida, intentando en la medida de lo posible conservar las características originales. El algoritmo está referenciado en la bibliografía. **[6]**
- Utilizar una librería en Scala, que puede dar lugar a todo el potencial que puede ofrecer este lenguaje; se realizan técnicas de paso de mensajes sobre un algoritmo de biobjetivo clásico, con operaciones como el merge o mezcla de mensajes.

4.1. Adaptación de un algoritmo a Scala

Al empezar el proyecto, teníamos desde el principio la idea de traducir un algoritmo biobjetivo ya existente desde C a Scala, junto con posibles cambios para permitir el uso de máster y workers.

El algoritmo consta de la siguiente estructura:

- Tres llamadas independientes al algoritmo de Dijkstra.
- Una función de crear lista de predecesores.
- Una función de crear lista de sucesores.

Las dos últimas funciones son el coste principal del algoritmo. Aquí, es usado **un montículo o heap**, y poder hacer una comparación rápida.

Tras ver la estructura, se ha hecho un borrador básico del algoritmo (que se conocerá como **Scala 0** en el apartado de las pruebas); y se han hecho las siguientes implementaciones.

- Cambio de estructura de datos para mejor rendimiento, cambiado de PriorityQueue a un array con un comportamiento de montículo, manteniendo en el nodo raíz el menor coste. **(Scala 1).**
- Paralelizar las 3 llamadas iniciales de Dijkstra **(Scala 2).**
- Llamar a la función de rellenar LS y LP. **(Scala 3).**
- Llamar paralelamente LS y LP. **(Scala 4).**

Sin embargo, en el último punto hemos visto errores de forma aleatoria a la hora de ejecutarlo y además el tiempo de ejecución es considerablemente superior. Esto se debe a la escritura conjunta e impredecible sobre la cola de prioridad, al acceder varios hilos al mismo recurso sin ningún tipo de bloqueo, que puede dejar inestable la estructura de montículo.

Dividir y paralelizar algunas de estas funciones de manera asíncrona para delegarle a cada uno de los *workers*, puede dar lugar a un comportamiento inesperado dentro del montículo, y dar errores a la hora de las soluciones. Por lo que hacer una solución a partir del algoritmo C **sólo puede ser paralelizable, pero no es distribuible.**

4.2. Usando la librería Pregel

4.2.1. Algoritmo con objetivo único

Como primera aproximación a un algoritmo de búsqueda de caminos de coste mínimo biobjetivo, se ha implementado un algoritmo con objetivo único. Dado un grafo G donde cada arco tiene un coste, y teniendo un nodo inicial, se trata de averiguar cuál es el camino de coste mínimo (si existe) desde el nodo inicial a cada uno de los otros nodos del grafo y, de paso, se obtienen los costes de dichos caminos, siendo igual a infinito si no existe camino para un nodo en particular. Dados dos caminos con igual coste, solamente nos interesa uno de ellos.

En el caso de Pregel, tendremos en cuenta las siguientes características:

1. **Información del nodo solución**: la solución sería el grafo G, que además en cada nodo tiene una etiqueta con 2 datos (coste,pred), donde: **Coste:** coste para ir desde el nodo inicial hasta ese nodo.
2. **Pred:** Enlace al nodo predecesor a este nodo en el camino de coste mínimo. Inicialmente tiene `(INFINITO, VerticeInicial)`, excepto el `VerticeInicial` que tiene `(0,VerticeInicial)`.

- **Mensajes:** los mensajes contienen (coste,pred)
- **Mensaje Inicial:** `(INFINITO,0)`
- **SuperPaso, con los siguiente pasos:**

1. **Recepción de mensajes:** si el coste del mensaje que llega es menor que el existente se actualiza el nodo con dicho mensaje.
2. **Construcción de mensajes:** para todos los arcos que salen de dicho nodo si el `coste(destino)` es mayor que el `coste(origen)+costo(arco)` se manda mensaje al nodo destino con `(coste(origen)+coste(arco), origen)`.
3. **Mezcla de mensajes:** dados dos mensajes, solo se manda el que tenga coste menor.

4.2.2. Algoritmo Biobjetivo

Cuando cada coste tiene dos coordenadas, si tenemos dos costes $A(a_1,a_2)$ y $B(b_1,b_2)$ definimos que:

- $A = B \Leftrightarrow a_1 = b_1$ y $a_2 = b_2$.
- $A(a_1,a_2)$ domina a $B(b_1,b_2) \Leftrightarrow A \neq B$ y $a_1 \leq b_1$ y $a_2 \leq b_2$ con una desigualdad siendo estricta.

  Dados dos pares A y B distintos, puede que ni A esté dominado por B, ni B esté dominado por A.

En este caso, dado un grafo G donde cada arco tiene de coste dos valores, y dado un vértice inicial, se trata de averiguar cuales son los caminos con coste no dominado por ningún otro camino desde el nodo inicial a cada uno de los otros nodos del grafo.

Nótese que, para cada nodo, la solución es un conjunto de caminos o el conjunto vacío si no se puede llegar desde el nodo inicial. Dados dos caminos con igual coste solamente nos va a interesar uno de ellos.

Para codificar la solución, vamos a tener un conjunto de costes no dominados para cada nodo del grafo. Esto es, dentro de un mismo conjunto, no hay ningún coste que domine a otro.

En conclusión, diremos que un conjunto de costes *XSet* domina a un coste *Y* si existe algún elemento de *XSet* que domine a *Y*.

En el método, vamos a trabajar con más de un Set, por lo que operaciones como la **fusión o merge** de dos conjuntos de costes libres de dominación *Xset* e *Yset*, que se define como la unión y eliminando los elementos dominados; son utilizados para añadir o eliminar caminos para dar una solución de caminos mínimos.

PREGEL

Información del nodo solución: la solución sería el grafo G, que además en cada nodo tiene un conjunto de $(coste_1, coste_2, pre)$:

- $coste_1$, $coste_2$: las dos coordenadas del coste para ir desde el nodo inicial hasta ese nodo
- `pre`: enlace al nodo predecesor a este nodo en el camino de coste mínimo.

  Inicialmente tiene {}, excepto el `VerticeInicial` que tiene `{(0,0,VerticeInicial)}`

Mensajes: los mensajes contienen un conjunto libre de dominación $\{(coste_1, coste_2, pre)\}$

Mensaje Inicial: `{(INFINITO,INFINITO,0)}`

SuperPaso:

Recepción de mensajes: se fusiona el conjunto de costes (no dominados) del nodo con los recibidos el mensaje.

Construcción de mensajes: para todos los arcos si el `coste(origen)+coste(arco)` no está dominado por los costes del destino, añadir `(coste(origen)+coste(arco),origen)` al conjunto de costes del mensaje. Si el conjunto de costes no está vacío, enviar mensaje.

Mezcla de mensajes: dados dos mensajes se manda la fusión de ambos.

Todo este proceso lo podemos ver gráficamente en el próximo ejemplo sencillo, con 5 nodos, no sólamente para cómo funciona el algoritmo, sino el criterio que utilizaríamos para determinar el conjunto de soluciones biobjetivo.

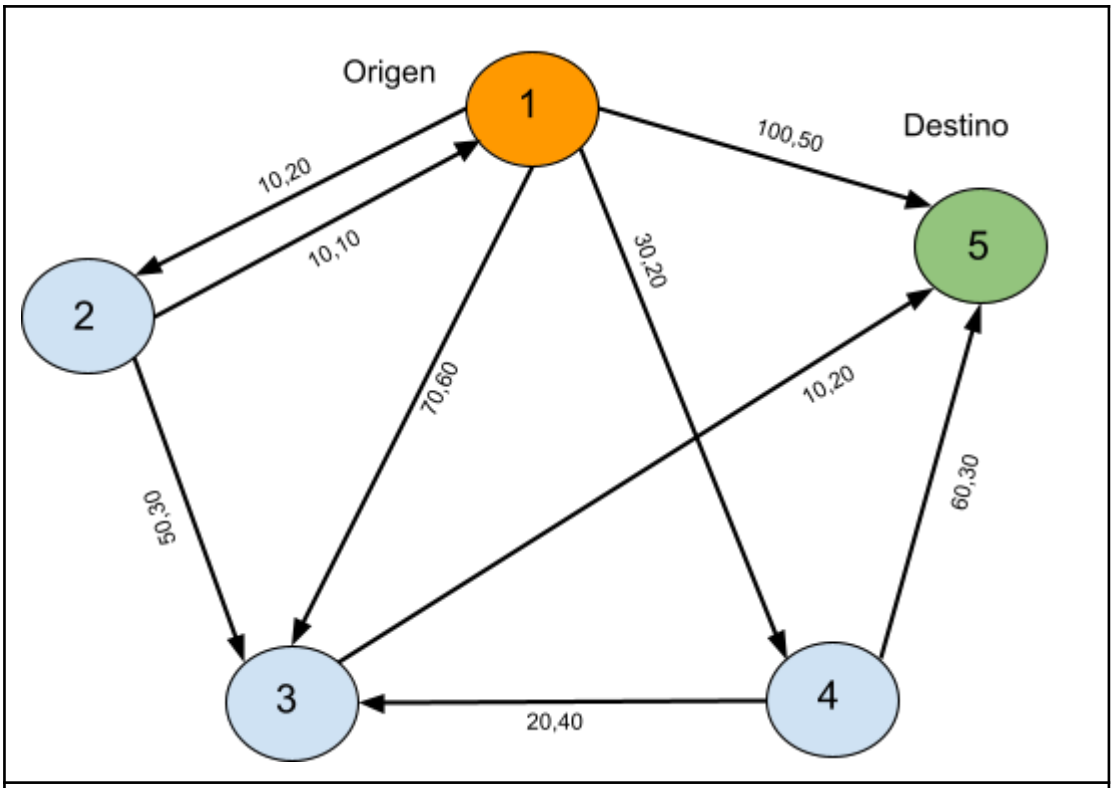


Figura 4 - Grafo de ejemplo de coste doble.

Dado el grafo de doble costo del ejemplo, con 5 vértices, 9 arcos, origen el nodo 1 y destino el nodo 5, utilizaremos la tabla de sucesores y predecesores para almacenar el grafo.

Nodo	**Predecesores**			
1	2 (10,10)			
2	1 (10,20)			
3	1 (70,60)	2 (50,30)	4 (20,40)	
4	1 (30,20)			
5	1 (100,50)	3 (10,20)	4 (50,30)	

Nodo	Sucesores			
1	2 (10,20)	3 (79,60)	4 (30,20)	5 (100,50)
2	1 (10,10)	3 (50,30)		
3	5 (10,20)			
4	3 (20,40)	5 (60,30)		
5				

Superpaso 1

1.1 Recepción de Mensajes

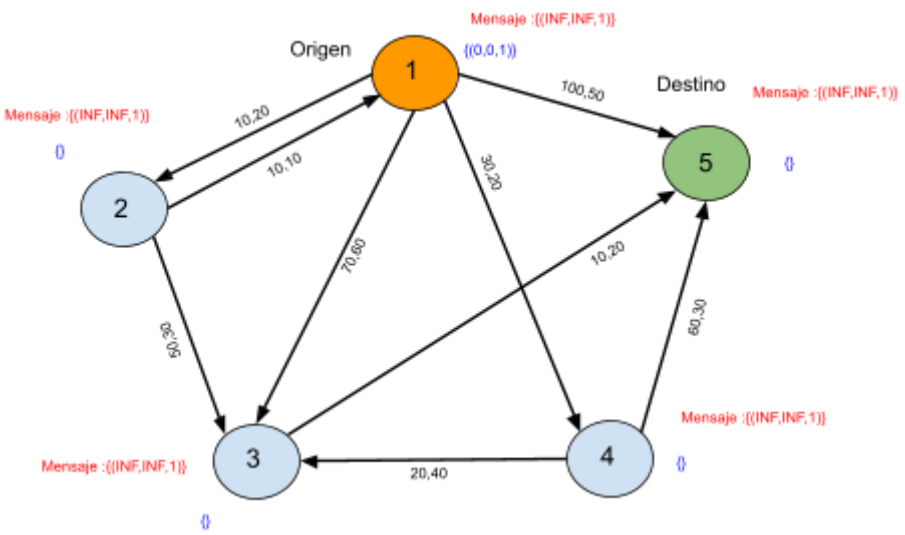


Figura 5 - Superpaso 1 - Recepción de Mensaje inicial

El mensaje inicial se envía a todos los nodos del grafo enviando un mensaje de costo (INF,INF,1) costes infinito desde el nodo 1.

1.2 Construcción de mensajes

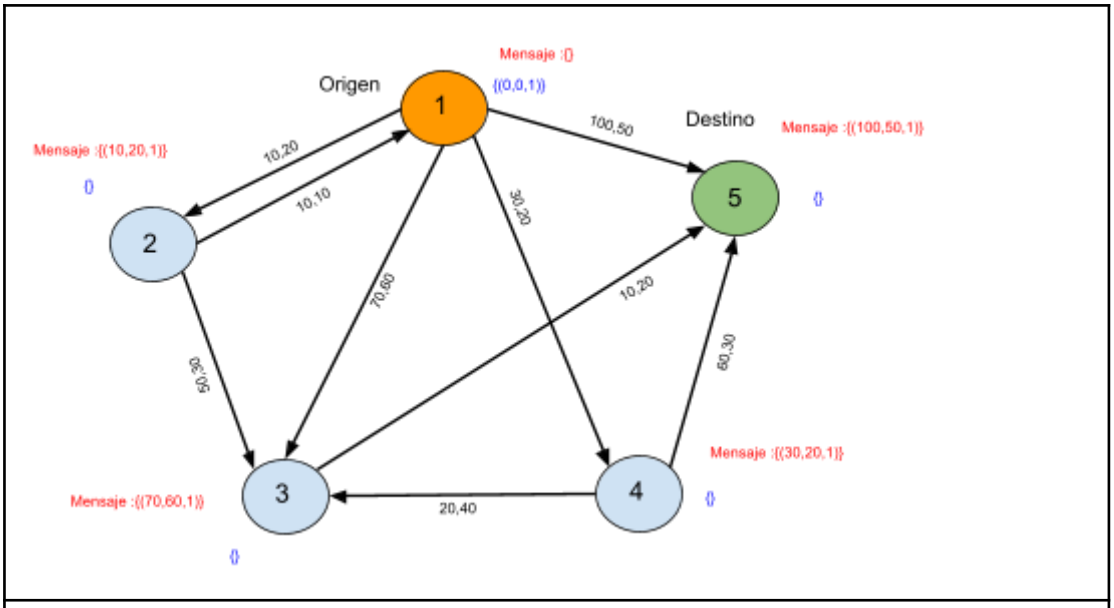


Figura 6 - Superpaso 1 - Generación de mensajes

Una vez actualizados los costes mínimos de cada nodo, se analizan todos los arcos (tripletas) para enviar nuevos costes mínimos al destino de cada arco. Este cálculo se realiza en cada arco en paralelo.

1.3 Mezcla de Mensajes

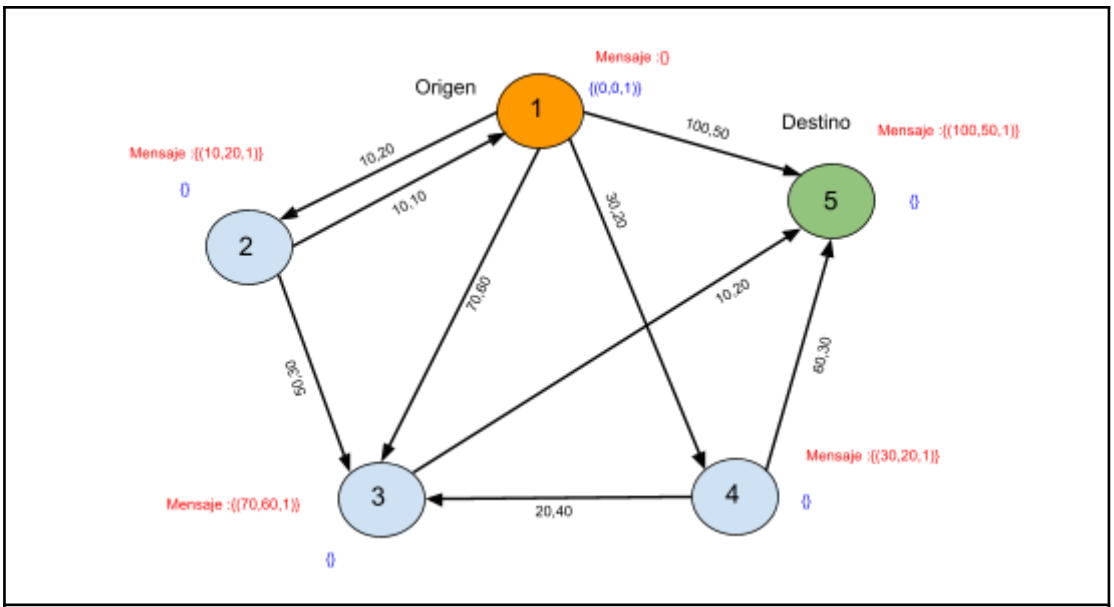


Figura 7 - Superpaso 1 - Mezcla de Mensajes

Como no hay nodos que reciban más de un mensaje este paso no hace nada.

Superpaso 2 :

2.1 Recepcion de Mensajes

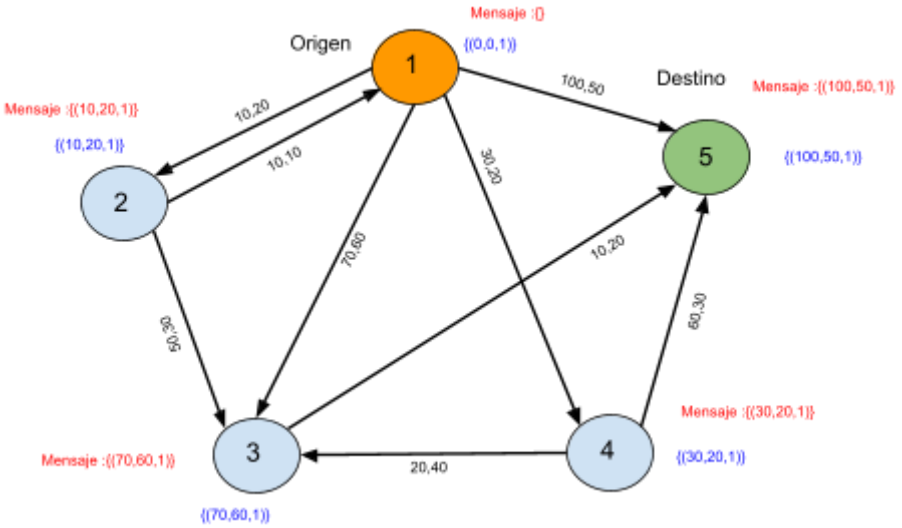


Figura 8 - Superpaso 2 - Recepción de Mensajes

Después de la recepción de mensajes, se actualizan los caminos mínimos de cada vértice con la mezcla entre los mensajes recibidos y los caminos mínimos existentes en el vértice.

2.2 Generación de Mensajes

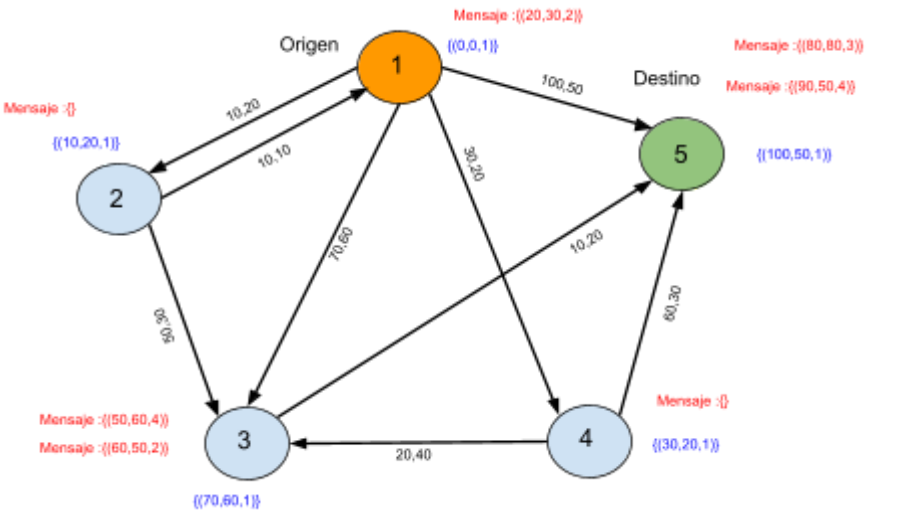


Figura 9 - Superpaso 2 - Generación de Mensajes

En la generación de mensajes sobre un vértice le pueden llegar varios Mensajes provenientes de distintos arcos.

2.3 Mezcla de Mensajes

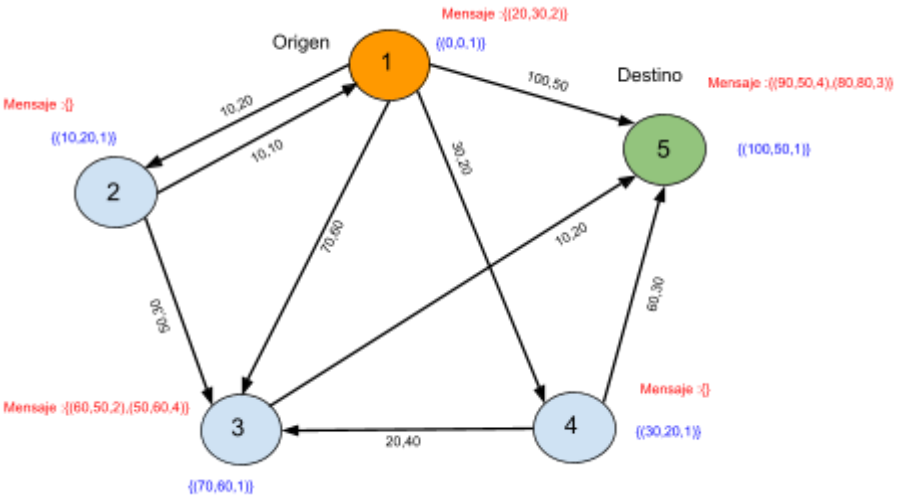


Figura 10 - Superpaso 2 - Mezcla de Mensajes

Se fusionan los mensajes con destino a cada nodo, sería la unión de los caminos eliminando los dominados.

Superpaso 3 :

3.1 Recepcion de Mensajes

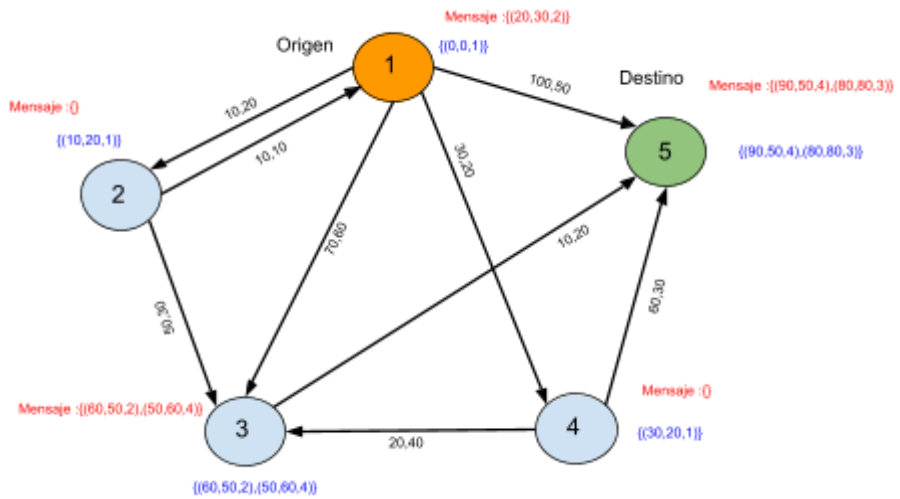


Figura 11 - Superpaso 3 - Recepción de Mensajes

En la recepción de mensajes se realiza la fusión de los caminos existentes en el vértice con los que vienen en el Mensaje con destino a ese vértice.

Soluciones

En este ejemplo, tenemos que la búsqueda de caminos mínimos desde el nodo 1 al nodo 5, con doble costo, arroja 3 caminos mínimos.

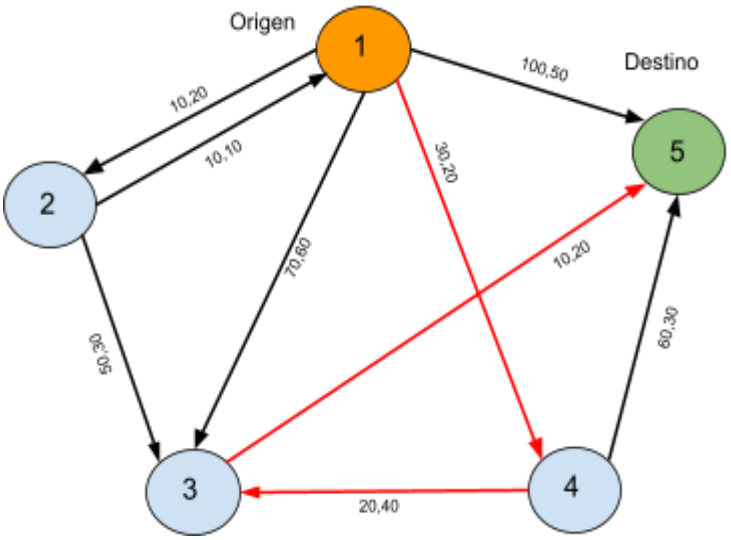


Figura 12 - Solución 1: Camino 1 -> 4 -> 3 -> 5, Coste 60,80

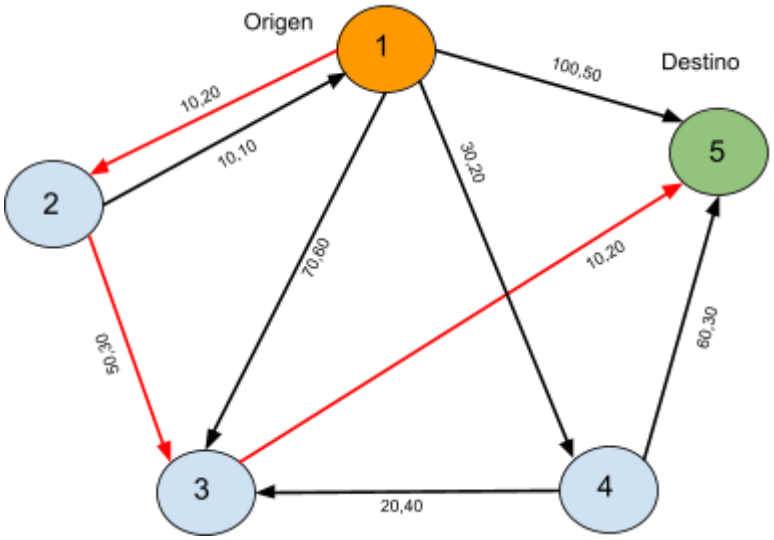


Figura 13 - Solución 2: Camino 1 -> 2 -> 3 -> 5, Coste 70,70

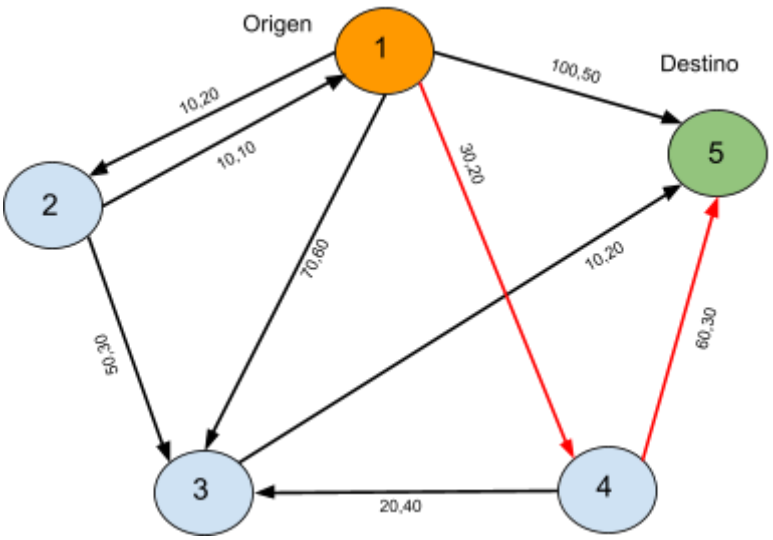


Figura 14 - Solución 3: Camino 1 -> 4 -> 5, Coste 90,50

4.2.3. Algoritmo Multiobjetivo

En este caso tendríamos costes de N dimensiones y si tenemos dos etiquetas $A(a_1,...,a_N)$ y $B(b_1,...,b_N)$ definimos que:

- $A = B \Leftrightarrow a_i = b_i$ para todo i=1,..,N.
- A domina a B $\Leftrightarrow A \neq B$ y $a_i \leq b_i$ para todo i=1,..,N con una desigualdad siendo estricta.

En este caso dado un grafo G donde cada arco tiene de coste N valores y dado un vértice inicial, se trata de averiguar cuales son los caminos con coste no dominado por ningún otro camino desde el nodo inicial a cada uno de los otros nodos del grafo. Nótese que para cada nodo, la solución es un conjunto de caminos eficientes (conjunto vacío si no se puede llegar desde el nodo inicial). Dados dos caminos con igual coste solamente nos va a interesar uno de ellos.

PREGEL

Información del nodo solución: la solución sería el grafo G, que además en cada nodo tiene un conjunto de $((coste_1, ..,coste_N),pre)$:

- $coste_1,..,\ coste_N$: las N coordenadas del coste para ir desde el nodo inicial hasta ese nodo
- `pre`: enlace al nodo predecesor a este nodo en el camino de coste mínimo.

  Inicialmente tiene {}, excepto el `VerticeInicial` que tiene `((0,..,0),VerticeInicial)`

Mensajes: los mensajes contienen $\{((coste_1,..,coste_N),pre)\}$

Mensaje Inicial: `{(INFINITO,.., INFINITO),0)}`

SuperPaso:

Recepción de mensajes: se fusiona el conjunto de costes (libre de dominación) del nodo con los recibidos el mensaje.

Construcción de mensajes: para todos los arcos si el `coste(origen)+coste(arco)` no está dominado por los costes del destino añadir `(coste(origen)+coste(arco),origen)` al conjunto de costes del mensaje. Si el conjunto de costes no está vacío, enviar mensaje.

Mezcla de mensajes: dados dos mensajes se manda la fusión de ambos.

4.3. Usando la librería Pregel

Pregel es una librería incluida dentro de GraphX que:

- Permite la comunicación de vértices entre los propios *workers*.
- Propone una manera simple de programar, y esto da lugar a una mejor escalabilidad (puede ser una buena opción para realizar algoritmos con N-objetivos).

Sin embargo, este modelo de mensajes pueden llegar a surgir problemas:

- Usando el concepto de mensajes, sobre todo aplicado a problemas de grafos, pueden surgir problemas de cuello de botella y, por lo tanto, de rendimiento.

  En el caso de un nodo X, tiene que esperar la respuesta y cálculo de los predecesores y enviarlos en un nuevo mensaje a todos los sucesores que pueda tener el nodo.
- El mensaje que puede enviar no es una simple solución: en un sistema biobjetivo, por lo que tenemos que esperar *arrays* como soluciones, y tendremos L mensajes. donde L será el número de conexiones dentro del grafo. Si calculamos que existen mensajes que pueden llegar a 1 MB, podremos exceder fácilmente la memoria soportada de nuestro sistema de recursos.

Estos problemas serán desarrollados en la siguiente sección de pruebas.

5. Pruebas y comparaciones

5.1. Objetivos

Como ya se ha mencionado, uno de los objetivos es medir los tiempos de ejecución en cada uno de los casos, para comprobar si alguna aproximación de las anteriores satisface los objetivos.

5.2. Pruebas y tests obtenidos

Para cada una de las pruebas, era conveniente utilizar como ejemplo alguno de los *datasets* de todos los caminos de Estados Unidos. Sin embargo, por limitaciones de memoria RAM, sólo es posible realizar con los 2 mapas más pequeños. En la imagen lo veremos como Roma y Nueva York, con sus arcos y nodos para determinar el tamaño.

Componentes del mapa			Tiempos (medidos en segundos)						
Mapa	**Nodos**	**Arcos**	**C**	**Scala0**	**Scala1**	**Scala2**	**Scala3**	**Scala4**	**Pregel**
ROMA	3,353	8,870	**0.01**	0.08	0.06	0.11	**0.12**	0.26	**33.07**
NY	264,346	733,846	**0.73**	10.26	1.74	1.37	**1.26**	13.55	**Error**
BAY	321,270	800,172	**3.09**	88.84	4.47	4.04	**3.70**	50.97	**Error**
COL	435,666	1,057,066	**17.40**	700.81	21.21	20.54	**19.63**	257.09	**Error**
FLA	1,070,376	2,712,798	**15.63**	779.84	23.45	20.39	**17.93**	261.84	**Error**
NE	1,524,453	3,897,636	**17.20**	686.6	23.03	21.41	**21.13**	318.77	**Error**
CAL	1,890,815	4,657,742	**2.26**	19.82	5.67	4.07	**4.16**	23.66	**Error**
LKS	2,758,119	6,885,658	**2.13**	19.43	6.27	3.78	**3.60**	11.73	**Error**

Leyendas	
Scala0	Cola de Prioridad: La Cola con prioridad de nodos que faltan por examinar, se implementa como mutable. PriorityQueue.
Scala1	Cola de Prioridad como mónticulo: La Cola con prioridad de nodos que faltan por examinar, se implementa como un Array y dentro se mantiene una estructura de montículo.
Scala2	Cola de Prioridad como mónticulo , 3 Dijkstra paralelo: Además se paraleliza las 3 llamadas a Dijkstra (cálculo de camino mínimo uniobjetivo) que tiene al comienzo del algoritmo.
Scala3	Cola de Prioridad como mónticulo , 3 Dijkstra paralelo. Se han implementado Revision de LS y Revision de LP como funciones separadas, aunque se llaman de forma secuencial
Scala4	Cola de Prioridad como mónticulo , 3 Dijkstra paralelo. Se llaman en paralelo a Revision_LS y Revision_LP
Pregel	Pregel

Figura 15 - Tabla de tiempos de las diferentes implementaciones

Como vemos aquí, Pregel no es una buena opción para un solo recurso, mientras que con Scala podemos asemejarnos a la velocidad que proporciona C. Sin embargo, se han hecho también pruebas con un prototipo de maestro-esclavos en Spark, donde se reparten los esclavos entre distintas máquinas físicas.

Además, solo se puede realizar el primer mapa (ROMA) debido a las restricciones físicas de memoria que se explicó anteriormente.

ORDENADOR1				ORDENADOR 2			TIEMPO MEDIDO (en Segundos)				
Programa	Master	Workers	Cores	Master	Workers	Cores	T1	T2	T3	T4	Media
SI	NO	0	0	SI	1	1	37,66	37,66	37,36	38,86	37,89
SI	NO	0	0	SI	2	1	34,29	33,44	33,11	33,75	33,65
SI	NO	0	0	SI	3	1	35,52	36,01	35,06	35,74	35,58
SI	NO	0	0	SI	4	1	39,08	39,83	36,69	38,37	38,49
SI	NO	1	1	SI	0	0	37,52	37,51	37,97	38,56	37,89
SI	NO	2	1	SI	0	0	38,01	36,89	36,36	37,76	37,26
SI	NO	3	1	SI	0	0	41,13	39,77	40,36	38,58	39,96
SI	NO	1	1	SI	1	1	33,53	33,08	32,53	33,14	33,07
SI	NO	1	1	SI	2	1	33,43	33,35	33,46	33,76	33,50
SI	NO	2	1	SI	1	1	36,68	37,06	38,13	39,23	37,78
SI	NO	2	1	SI	2	1	34,37	35,57	41,63	32,7	36,07
SI	NO	0	0	SI	1	2	33,55	33,73	33,07	33,71	33,52
SI	NO	0	0	SI	2	2	37,38	37,56	39,23	37,86	38,01
SI	NO	0	0	SI	3	2	40,63	40,48	40,09	41,12	40,58
SI	NO	1	2	SI	0	0	38,06	37,06	37,05	35,74	36,98
SI	NO	2	2	SI	0	0	41,82	40,47	41,14	42,05	41,37
SI	NO	3	2	SI	0	0	44,32	45,94	42,92	45,06	44,56
SI	NO	1	2	SI	1	2	35,52	35,88	35,14	35,17	35,43
SI	NO	1	1	SI	1	2	33,37	35,88	34,11	33,76	34,28

Figura 16 - Tabla de tiempos en PREGEL en función de la configuración de los workers

Aunque esperábamos, al menos aceleraciones logarítmicas en cuanto al tiempo de ejecución (no se nos ocurrió pensar siquiera en aceleraciones lineales); observamos que incluso se llega a aumentar el tiempo de ejecución. La explicación se llega a dar en el cuello de botella, donde la elaboración de mensajes y su tamaño llega a tener un impacto negativo en el tiempo que hay que organizar para cada *SuperPaso.*

Aunque esperamos unos resultados mejores que el tiempo de ejecución de C, puede ser un buen punto de partida y tomado en cuenta como prototipo para futuras implementaciones u otros proyectos que requieran este tipo de técnicas.

6. Conclusiones y líneas futuras

6.1. Conclusiones

En este proyecto, se ha tomado dos alternativas a la hora de implementar un algoritmo para encontrar los caminos eficientes en grafos biobjetivos: una solución paralela que puede aproximarse al tiempo de ejecución en C, y una solución distribuida que, a pesar de no ser clasificado como eficiente, puede ser una base a futuros desarrollos.

6.2. Líneas futuras

Para que el esquema esbozado en este proyecto tenga posibilidades de implantarse, habría que poder contemplar la realización de las siguientes tareas:

- Basarnos en algún otro algoritmo que impida el conflictos entre recursos, o el evitar la memoria compartida. Sin embargo, la otra manera puede llegar a duplicar la información, como sería en el caso de Pregel, exigiendo más memoria RAM de la necesaria.
- Buscar métodos y herramientas para reducir el formato de mensajes. Sin embargo, el formato de Pregel es muy cerrado y rígido.
- Si bien se han hecho las pruebas en un sistema de ordenadores simple, es mejor adaptar el problema a una situación real, es decir, realizar más pruebas en una red de ordenadores donde dispongan de suficiente memoria RAM para satisfacer las condiciones del exceso de mensajes del Pregel.
- Estudio de requisitos para utilizarlo en problemas reales. Muchos problemas de grafos son usados constantemente en las aplicaciones, y pueden hacer énfasis en el tiempo de respuesta.

7. Summary and Conclusions

For this project, two options were considered and designed:

- A parallel Scala algorithm adapted from an existing C project. But due to its mechanics, mainly the heap management, it cannot be used as a distributed algorithm. They were tested on distributed systems, but the solutions weren't as expected.

  Also, we need to implement more algorithms that are not dependent on shared-memory as the C example, to be really effective and categorized as a proper solution .
- A distributed Pregel program that, although it wasn't checked as effective in terms of execution time nor memory, can be used as a baseline for future implementations.

8. Presupuesto

Tipo	Objeto	Tiempo	Coste
Hardware:	Ordenadores, tanto server como workers		1500€
Software:	Sistema Operativo: Linux		0€
Software:	Herramientas: Scala, Spark, GraphX (Todos ellos open-source)		0€
Desarrollo:	Coste de programación del programa, además de su adaptación para el problema a resolver.	250 horas	2500 €
		Total	**4000 €**

9. Bibliografía

[1] Ammar K, Özsu MT (2018). Experimental analysis of distributed graph systems. *Proc. VLDB Endow. 11, 10 (June 2018)*, 1151–1164. https://doi.org/10.14778/3231751.3231764

[2] Needham M, Hodler A (2019). Graph Algorithms: Practical Examples in Apache Spark and Neo4j. *O'Reilly Media*.

[3] Phan T, Do P (2018). Improving the shortest path finding algorithm in Apache Spark GraphX. *In Proceedings of the 2nd International Conference on Machine Learning and Soft Computing (ICMLSC '18)*. Association for Computing Machinery, New York, NY, USA, 67–71. https://doi.org/10.1145/3184066.3184083

[4] Ray A (2017). Write Graph Algorithms Like a Boss. *Spark Summit 2017*. Accesible en: https://es.slideshare.net/databricks/write-graph-algorithms-like-a-boss-andrew-ray

[5] Schoeneman F, Zola J (2019). Solving All-Pairs Shortest-Paths Problem in Large Graphs Using Apache Spark. *In Proceedings of the 48th International Conference on Parallel Processing (ICPP 2019)*. Association for Computing Machinery, New York, NY, USA, Article 9, 1–10. https://doi.org/10.1145/3337821.3337852

[6] Sedeño-Noda A, Colebrook M (2019). A biobjective Dijkstra algorithm. *European Journal of Operational Research*. Volume 276, Issue 1, Pages 106-118, ISSN 0377-2217, https://doi.org/10.1016/j.ejor.2019.01.007.

[7] Zheng Cy, Wang J, Jain A (2015). All-Pairs Shortest Paths in Spark. Accesible en: https://pdfs.semanticscholar.org/a634/81aeaa63f6d354bc1cdf8fbc8a8bd636072f.pdf

10. Anexos.

Apéndice A. Proyecto P2PV2PLUS.

Se ha trasladado a Scala el programa AllnondominatedpathsP2PV2PLUS.c escrito en C, para el cálculo de caminos mínimos con costes de 2 dimensiones.

package.scala - Variables globales del proyecto

Tipo_Grafo.scala - Tipos de Datos usados.

Main.scala - Programa principal.

Utilidades.scala - Funciones generales incluido el método de Dijkstra de búsqueda de camino de coste mínimo (considerando sólo una dimensión), que se usa en los algoritmos posteriores.

P2PV2PLUS.scala - Búsqueda de camino de coste mínimo (2 dimensiones) trasladado desde C. Este el algoritmo denominado “Scala 0” en las tablas. Utiliza como cola de prioridad el tipo de dato mutable.PriorityQueue de scala y para almacenar las etiquetas LF un Array[ListBuffer]. Se utilizó para almacenar la cola con prioridad DIJKS del método de Dijkstra un modelo de datos tipo

```
val DIJK = mutable.PriorityQueue.empty(Orden_pila)
```

con un orden definido por

```
val Orden_pila = Ordering.fromLessThan[Tipo_Elemento_Pila]({

(m, n) => ( (m.clave > n.clave ) || (m.clave == n.clave) && ( m.d2 > n.d2) ) })
```

se ordena por clave y si son iguales se ordena por d2

P2PV3PLUS.scala - Búsqueda de camino de coste mínimo (2 dimensiones) trasladado desde C. Este el algoritmo denominado “Scala 1” y “Scala 2” en las tablas (dependiendo si habilitamos el paralelismo de los Dijkstra). Se han mejorado de cara al rendimiento los siguientes aspectos :

• El árbol se lee 2 veces para al crear el grafo la lista de arcos sucesores y predesesores de cada nodo LS y LP se almacenen en un Array de tamaño fijo (Cada Array tiene tamaño distinto).

• Para la cola del método de Dijstra como del método del Método de caminos mínimos con doble coste se utiliza un array de tamaño fijo, del que se van insertando elementos y sacando formando un montículo ordenado por el campo clave y luego por d2.

• Para almacenar las etiquetas de cada nodo (LF – Se utiliza un Array [ArrayBuffer[Tipo_label]] (nnodos). Sería un Array de dimensión el número de nodos del grafo y dentro un ArrayBuffer con datos de Tipo_Label. Los ArrayBuffer se pueden redimensionar y para ganar un poco de rendimiento a costa de espacio de memoria, se va realizando en base a :

- Cuando se añade el primer elemento se pone tamaño 2.
- Cuando se llena se duplica el tamaño.

• En los bucles más internos se ha optado por hacer cálculos fuera de los bucles poniendo variables extra (perdiendo claridad el algoritmo).

package.scala

```
package P2PV2PLUS

import P2PV2PLUS.model._

import scala.collection.mutable.ListBuffer

package object model {

 val Maxcost = 100000000
 val UERROR2 = 100000000
 val INF = 1000000000
 val DEBUG = 0 // 0 -NODEBUG, 1-MEDIUM, 2 -FULL
 var PARALELO = 2
 // 0 -NODEBUG, 1-MEDIUM, 2 -FULL
 val MinData = (- INF )

 val Orden_pila = Ordering.fromLessThan[Tipo_Elemento_Pila]({
   (m, n) => ( (m.clave > n.clave ) || (m.clave == n.clave) && ( m.d2 > n.d2) ) })
//               || (m.clave == n.clave) && ( m.d2 == n.d2) && (m.elemento >
n.elemento) ) })

}
```

Tipo_Grafo.scala

```
package P2PV2PLUS.model


import scala.collection.mutable.ListBuffer
import scala.io.Source


case class Tipo_Elemento_Pila(
                               var elemento: Int,
                               var clave: Int,
                               var d1: Int,
                               var d2: Int
                             )

case class Tipo_Label(
                       var pred: Int,
                       var d1: Int,
                       var d2: Int,
                       var pospred: Int,
                       var possucc: Int
                     )

case class Tipo_Cola_prioridad(
                     val cola: Array[Tipo_Elemento_Pila],
                     val posicion: Array[Int],
                     var n: Int
                              )


case class Tipo_Resultado(
                      var ttime: Int, // Tiempo CPU
                      var nL: Int,    //
```

```
                      var nds: Int,   //
                      var nextract: Int,   //
                      var nlabels: Int   //
                         )




case class Tipo_Arco(
                      val nodo: Int,
                      val c1: Int,
                      val c2: Int
                    )

case class Tipo_Nodo(
                      var LS: Array[Tipo_Arco],
                      var LP: Array[Tipo_Arco]
                    )

case class Tipo_Grafo(
                       var nodos: Int, // Numero de Nodos (vértice)
                       var arcos: Int, // Numero de Arcos
                       var maxc1: Int, // Maaximo coste C1 en el grafo
                       var maxc2: Int, // Máximo coste C2 en el grafo
                       var s: Int,     // Vértice inicial
                       var t: Int,     // Vértice final
                       var nodo: Array[Tipo_Nodo]
                     ) {
  def CargarFichero(fichero: String) {
    val source = Source.fromFile(fichero)
    val lines = source.getLines()
    val Array(x1, x2, x3, x4, _*) = lines.next.split(" ")
    val matrix2 = lines.map(_.split(" ")).toArray
    source.close()
    val nnodos = x3.toInt
    val narcos = x4.toInt
    if ((nnodos > 0) && (narcos > 0)) {
      nodos = nnodos
      arcos = narcos
      s = 0
      t = nnodos - 1
      maxc1 = 0
      maxc2 = 0


      //
      // INICIALIZAR EN GRAFO
      //
      nodo = new Array(nnodos)
      val nLP  = new Array [Int] (nnodos)
      val nLS  = new Array [Int] (nnodos)

      for (i <- 0 to nnodos - 1) {
        nLS(i) = 0
        nLP(i) = 0
      }
      //
      // Convertir lista de Arcos en Predecesores y Sucesores.
      //
      for (m <- matrix2) {
        val a = m(0)
        if (a == "a") {
          val i = m(1).toInt
```

```
          val j = m(2).toInt
          val u = m(3).toInt
          val ut = m(4).toInt
          //       println(s"Elemento ( $a $i $j $u $ut)")
          if ((i != 0) && (j != 0)) {
            nLP(j-1) += 1
            nLS(i-1) += 1
          }

          if (u > maxc1) maxc1 = u
          if (ut > maxc2) maxc2 = ut
        }

      }
      for (i <- 0 to nnodos - 1) {
        nodo(i)= new Tipo_Nodo( LS=new Array( nLS(i)), LP= new Array( nLP(i)) )
        nLS(i) = 0
        nLP(i) = 0
      }


      for (m <- matrix2) {
        val a = m(0)
        if (a == "a") {
          val i = m(1).toInt
          val j = m(2).toInt
          val u = m(3).toInt
          val ut = m(4).toInt
          //       println(s"Elemento ( $a $i $j $u $ut)")
          if ((i != 0) && (j != 0)) {

            nodo(j - 1).LP(nLP(j-1)) = Tipo_Arco(i - 1, u, ut)
            nodo(i - 1).LS(nLS(i-1)) = Tipo_Arco(j - 1, u, ut)
            nLP(j-1) += 1
            nLS(i-1) += 1
          }

          if (u > maxc1) maxc1 = u
          if (ut > maxc2) maxc2 = ut
        }

      }

      if (DEBUG > 0) { // Ver si interesa sacar Resumen del Grafo
        println(s"Grafo con ${nodos} nodos y ${arcos} arcos, MaxC1 = ${maxc1},
MaxC2 = ${maxc2} s = $s t = $t")
      }

    }
    else {
      println("el formato del comando es NS <nombrefichero>")
      println("el formato del fichero es el siguiente      ")
      println("==========================================")
      println("s sp  NODOS(n)  ARCOS(m)                   ")
      println("a i1     j1   c[i1,j1] t[i1,j1]            ")
      println("              . . .                        ")
      println("a im     jm   c[im,jm] t[im,jm]            ")
      println("==========================================")
    }
  }

}
```

Utilidades.scala

```
package P2PV2PLUS.model


import scala.collection.mutable.ListBuffer
import scala.io.Source


case class Tipo_Elemento_Pila(
                               var elemento: Int,
                               var clave: Int,
                               var d1: Int,
                               var d2: Int
                             )

case class Tipo_Label(
                       var pred: Int,
                       var d1: Int,
                       var d2: Int,
                       var pospred: Int,
                       var possucc: Int
                     )

case class Tipo_Cola_prioridad(
                     val cola: Array[Tipo_Elemento_Pila],
                     val posicion: Array[Int],
                     var n: Int
                              )


case class Tipo_Resultado(
                       var ttime: Int, // Tiempo CPU
                       var nL: Int,    //
                       var nds: Int,   //
                       var nextract: Int,   //
                       var nlabels: Int   //
                          )







case class Tipo_Arco(
                       val nodo: Int,
                       val c1: Int,
                       val c2: Int
                     )

case class Tipo_Nodo(
                       var LS: Array[Tipo_Arco],
                       var LP: Array[Tipo_Arco]
                     )

case class Tipo_Grafo(
                        var nodos: Int, // Numero de Nodos (vértice)
                        var arcos: Int, // Numero de Arcos
                        var maxc1: Int, // Maaximo coste C1 en el grafo
                        var maxc2: Int, // Máximo coste C2 en el grafo
                        var s: Int,     // Vértice inicial
                        var t: Int,     // Vértice final
                        var nodo: Array[Tipo_Nodo]
                      ) {
  def CargarFichero(fichero: String) {
```

```
    val source = Source.fromFile(fichero)
    val lines = source.getLines()
    val Array(x1, x2, x3, x4, _*) = lines.next.split(" ")
    val matrix2 = lines.map(_.split(" ")).toArray
    source.close()
    val nnodos = x3.toInt
    val narcos = x4.toInt
    if ((nnodos > 0) && (narcos > 0)) {
      nodos = nnodos
      arcos = narcos
      s = 0
      t = nnodos - 1
      maxc1 = 0
      maxc2 = 0


      //
      // INICIALIZAR EN GRAFO
      //
      nodo = new Array(nnodos)
      val nLP  = new Array [Int] (nnodos)
      val nLS  = new Array [Int] (nnodos)

      for (i <- 0 to nnodos - 1) {
        nLS(i) = 0
        nLP(i) = 0
      }
      //
      // Convertir lista de Arcos en Predecesores y Sucesores.
      //
      for (m <- matrix2) {
        val a = m(0)
        if (a == "a") {
          val i = m(1).toInt
          val j = m(2).toInt
          val u = m(3).toInt
          val ut = m(4).toInt
          //       println(s"Elemento ( $a $i $j $u $ut)")
          if ((i != 0) && (j != 0)) {
            nLP(j-1) += 1
            nLS(i-1) += 1
          }

          if (u > maxc1) maxc1 = u
          if (ut > maxc2) maxc2 = ut
        }

      }
      for (i <- 0 to nnodos - 1) {
        nodo(i)= new Tipo_Nodo( LS=new Array( nLS(i)), LP= new Array( nLP(i)) )
        nLS(i) = 0
        nLP(i) = 0
      }


      for (m <- matrix2) {
        val a = m(0)
        if (a == "a") {
          val i = m(1).toInt
          val j = m(2).toInt
          val u = m(3).toInt
          val ut = m(4).toInt
          //       println(s"Elemento ( $a $i $j $u $ut)")
          if ((i != 0) && (j != 0)) {
```

```
            nodo(j - 1).LP(nLP(j-1)) = Tipo_Arco(i - 1, u, ut)
            nodo(i - 1).LS(nLS(i-1)) = Tipo_Arco(j - 1, u, ut)
            nLP(j-1) += 1
            nLS(i-1) += 1
          }

          if (u > maxc1) maxc1 = u
          if (ut > maxc2) maxc2 = ut
        }

      }

      if (DEBUG > 0) { // Ver si interesa sacar Resumen del Grafo
        println(s"Grafo con ${nodos} nodos y ${arcos} arcos, MaxC1 = ${maxc1},
MaxC2 = ${maxc2} s = $s t = $t")
      }

    }
    else {
      println("el formato del comando es NS <nombrefichero>")
      println("el formato del fichero es el siguiente      ")
      println("============================================")
      println("s sp  NODOS(n)  ARCOS(m)                    ")
      println("a i1     j1   c[i1,j1] t[i1,j1]            ")
      println("             . . .                         ")
      println("a im     jm   c[im,jm] t[im,jm]            ")
      println("============================================")
    }
  }

}
```

Main.scala

```
package P2PV2PLUS.model

import scala.collection.mutable.ListBuffer
import scala.io.Source

import Utilidades._

object Main {
  def main(args: Array[String]): Unit = {
    Programa.Programa ( args )
  }
}


object Programa extends App {

  val x: Int = 0

  def Programa(args: Array[String]): Unit = {
    val G = new Tipo_Grafo ( 0,0, 0,0,0,0,new Array[Tipo_Nodo](0))
    var nombrefichero = "/home/usuario/TFM//USAROADNETWORKS/USA-road-dt.NY.txt"

    if ( args.length > 0)      nombrefichero = args(0)
    if ( args.length > 1)      PARALELO      = args(1).toInt
    //
    G.CargarFichero( nombrefichero )

    if (DEBUG == 2) mostrarGrafo(G)
```

```
    var eheaps: Int = 0

    var r = new Tipo_Resultado(0,0,0,0,0)

    var t0: Long = 0
    var t1: Long = 0
    var elapsed_time: Double = 0

    t0 = System.nanoTime()

    if (PARALELO == 0) {
      print(s"AllnondominatedpathsP2PV2PLUS(ORI) ${nombrefichero} ${G.nodos}
${G.arcos} ${G.s + 1} ${G.t + 1} ")
    }
    if (PARALELO == 1) {
      print(s"AllnondominatedpathsP2PV2PLUS ${nombrefichero} ${G.nodos} ${G.arcos}
${G.s + 1} ${G.t + 1} ")
    }
    if (PARALELO == 2) {
      print(s"AllnondominatedpathsP2PV2PLUS(PAR) ${nombrefichero} ${G.nodos}
${G.arcos} ${G.s + 1} ${G.t + 1} ")
    }

    if (PARALELO == 0 )                          r =
P2PV2PLUS.BicriteriaPathAlgorithm_p2pv2(G, 0)
    else if (PARALELO == 1 || PARALELO == 2 )    r =
P2PV3PLUS.BicriteriaPathAlgorithm_p2pv3(G, 0)


    t1 = System.nanoTime()
    elapsed_time = ((t1-t0) /1000.0/1000.0/1000.0)
    println(s"${elapsed_time} ${r.nL} ${r.nds} ${r.nextract} ${r.nlabels}")

  }

}
```

P2PV2PLUS.scala

```
package P2PV2PLUS.model

import scala.collection.mutable
import scala.collection.mutable.ListBuffer
import Utilidades._

import scala.concurrent.{Await,Future}
import scala.concurrent.duration._
import scala.concurrent.ExecutionContext.Implicits.global
import scala.util.{Failure, Success}

object P2PV2PLUS extends App {




  def BicriteriaPathAlgorithm_p2pv2(G: Tipo_Grafo, Singledestination: Int):
Tipo_Resultado = {
    // P2PV2PLUS
    var eheaps: Int = 0
    var nL: Int = 0
```

```
    var nds: Int = 0
    var nlabels: Int = 0
    val dg = G.nodos
    val predminF: Array[Int] = new Array(dg)
    val predminB: Array[Int] = new Array(dg)
    //    val predminpointerB: Array[Int] = new Array(dg)
    val predminpointerF: Array[Int] = new Array(dg)
    val CurrentArcF = new Array[Array[Int]](dg)
    //   val CurrentArcB = new Array[Array[Int]](dg)
    val numberiF: Array[Int] = new Array(dg)
    val numberiB: Array[Int] = new Array(dg)

    val dI1: Array[Int] = new Array(dg)
    val dI2: Array[Int] = new Array(dg)
    //    val dP1: Array[Int] = new Array(dg)
    //    val dP2: Array[Int] = new Array(dg)
    val d11: Array[Int] = new Array(dg)
    val d12: Array[Int] = new Array(dg)
    val d21: Array[Int] = new Array(dg)
    val d22: Array[Int] = new Array(dg)

    val HeapF = mutable.PriorityQueue.empty(Orden_pila)
    // val HeapB = mutable.PriorityQueue.empty(Orden_pila)

    var LF = new Array[ListBuffer[Tipo_Label]](dg)
    var LB = new Array[ListBuffer[Tipo_Label]](dg)
    var L = new Array[ListBuffer[Tipo_Label]](dg)

    val craya2F: Array[Int] = new Array(dg)
    val craya2B: Array[Int] = new Array(dg)
    val minF: Array[Int] = new Array(dg)
    val minB: Array[Int] = new Array(dg)
    val positionF: Array[Int] = new Array(dg + 1)
    val positionB: Array[Int] = new Array(dg + 1)
    //    var p: ListBuffer[Tipo_Arco] = ListBuffer()
    //    var p: Array[Tipo_Arco] ()
    var k: Int = 0
    var cr1: Int = 0
    var cr2: Int = 0
    var temp1: Int = 0
    var temp2: Int = 0
    var dif: Int = 0
    var NonStop = true
    var candidato: Int = 0
    var min2H: Int = 0
    var i: Int = 0
    var seguir = true

    var contador_insertar_pila = 0
    var contador_sacar_pila = 0
    var contador_modificar_pila = 0


    var etmin = new Tipo_Elemento_Pila(0, 0, 0, 0)
    var et = new Tipo_Elemento_Pila(0, 0, 0, 0)
    //    var nuevo = new Tipo_Label(0, 0, 0, 0, 0)


    eheaps = 0
    nL = 0
    nds = 0
    nlabels = 0

    eheaps += Dijkstra2(G, G.t, d11, predminB, d12, -1)
    eheaps += Dijkstra2(G, G.t, d22, predminF, d21, -2)
```

```
    eheaps += Dijkstra2(G, G.s, dI1, predminF, dI2, 1)



    // Inicializar HeapF
    min2H = INF
    // ------------------------------------------- (0)
    if (DEBUG == 2) println("-------------------------------------------------
Paso 0")
    for (i <- 0 to G.nodos - 1) // VER POSIBLE ERROR SE PASA, sería G.nodos -1 ?
    {
      if ((predminF(i) != UERROR2) && (predminB(i) != UERROR2)) {
        numberiF(i) = 1
        LF(i) = new ListBuffer()
        LF(i) += Tipo_Label(pred = predminF(i), d1 = dI1(i), d2 = dI2(i), 0, 0) //
possucc no se usa
        CurrentArcF(i) = new Array[Int](G.nodo(i).LP.length)
        if (DEBUG == 2) {
          print(s"LF(${i + 1}) = ");
          mostrarListaBuffer(LF(i))
        }
        nds += 1
      }
      else {
        numberiF(i) = 0
        LF(i) = new ListBuffer()
        CurrentArcF(i) = new Array[Int](0)
      }
    }
    // ---------------------------------- (1)
    if (DEBUG == 2) println("-------------------------------------------------
Paso 1")

    for (i <- 0 to G.nodos - 1) { // VER POSIBLE ERROR SE PASA, sería G.nodos -1 ?
      if ((i != G.s) && (numberiF(i) > 0)) {
        minF(i) = INF
        predminF(i) = -1
        val p = G.nodo(i).LP
        for (j <- 0 to p.length - 1) {
          k = p(j).nodo
          if (numberiF(k) > 0) {
            //         println(s"CurrentArcF(${i},${j} y dimension es
${CurrentArcF(i).size}")
            CurrentArcF(i)(j) = 0
            cr1 = p(j).c1 + dI1(k)
            cr2 = p(j).c2 + dI2(k)
            temp1 = p(j).c1 + dI1(k) + d11(i)
            temp2 = p(j).c2 + dI2(k) + d22(i)
            if ((cr1 > LF(i)(0).d1) && (cr2 < LF(i)(0).d2)) {
              if ((temp1 <= d21(G.s)) && (temp2 <= d12(G.s)) && (temp2 <
LF(G.t)(0).d2)) {
                dif = minF(i) - cr1
                if ((dif > 0) || ((predminF(i) != -1) && (dif == 0) && (craya2F(i)
> cr2))) {
                  minF(i) = cr1
                  craya2F(i) = cr2
                  predminF(i) = k
                  predminpointerF(i) = 0
                }
              }
            }
          }
          else
            CurrentArcF(i)(j) = -1
        }
```

```
        if (predminF(i) != -1) {
          // et.elemento = i
          // et.clave=minF(i)
          // et.d1 = minF(i)
          // et.d2 = craya2F(i)
          // HeapF += Tipo_Elemento_Pila(elemento = et.elemento, clave = et.clave,
d1 = et.d1, d2 = et.d2)
          HeapF += Tipo_Elemento_Pila(elemento = i, clave = minF(i), d1 = minF(i),
d2 = craya2F(i))
          contador_insertar_pila += 1

          if (DEBUG == 2) {
            print(s"(D) Inserted ( ${i + 1},${minF(i)},${minF(i)},${craya2F(i)} )
HeapF = ")
            mostrarPila(HeapF)
          }

          if (craya2F(i) < min2H) min2H = craya2F(i)
          nlabels += 1
        }
      }
    }
    // ------------------------------------------------ (2)
    if (DEBUG == 2) println("-------------------------------------------------
Paso 2")
    NonStop = true
    while (HeapF.nonEmpty && NonStop) {

      // ------------------------------------------- (2.1) FORWARD
      etmin = HeapF.dequeue()
      if (DEBUG == 2) {
        print(s"Extraido( ${etmin.elemento +
1},${etmin.clave},${etmin.d1},${etmin.d2} ) HeapF = ")
        mostrarPila(HeapF)
      }
      eheaps += 1
      candidato = etmin.elemento
      numberiF(candidato) += 1
      if (numberiF(candidato) == 1) {
        LF(candidato) = new ListBuffer[Tipo_Label]()
      }
      LF(candidato) += Tipo_Label(pred = predminF(candidato), d1 = etmin.d1, d2 =
etmin.d2, pospred = predminpointerF(candidato), possucc = 0)

      //      LF(candidato) += nuevo
      if (DEBUG == 2) {
        print(s"LF(${candidato + 1}) = ");
        mostrarListaBuffer(LF(candidato))
      }
      nds += 1
      predminF(candidato) = -1
      minF(candidato) = INF
      predminpointerF(candidato) = -1
      if (candidato != G.s) {
        val p = G.nodo(candidato).LP
        for (j <- 0 to p.length - 1) {
          k = p(j).nodo
          i = CurrentArcF(candidato)(j)
          seguir = true
          // ------------------------------------ 2.2
          while ((i >= 0) && (i < numberiF(k)) && seguir) {
            cr1 = p(j).c1 + LF(k)(i).d1
            cr2 = p(j).c2 + LF(k)(i).d2
            val xpaso = numberiF(candidato) - 1
            if ((cr1 > LF(candidato)(xpaso).d1) && (cr2 < LF(candidato)(xpaso).d2)
```

```
&& (cr2 <= dI2(candidato))) {
              temp1 = cr1 + d11(candidato)
              temp2 = cr2 + d22(candidato)
              if ((temp1 <= d21(G.s)) && (temp2 <= d12(G.s)) && (temp2 <
LF(G.t)(numberiF(G.t) - 1).d2)) {
                dif = -cr1 + minF(candidato)
                seguir = false
                if ((dif > 0) || ((predminF(candidato) != -1) && (dif == 0) &&
(cr2 < craya2F(candidato)))) {
                  minF(candidato) = cr1
                  predminpointerF(candidato) = i
                  predminF(candidato) = k
                  craya2F(candidato) = cr2
                }
              }
            }
            if (seguir) i += 1
          }
          CurrentArcF(candidato)(j) = i
        }
        if (predminF(candidato) != -1) {
          HeapF += Tipo_Elemento_Pila(elemento = candidato, clave =
minF(candidato), d1 = minF(candidato), d2 = craya2F(candidato))
          nlabels += 1
          // println(s"I ${candidato}")
          // contador_insertar_pila += 1
          if (craya2F(candidato) < min2H) min2H = craya2F(candidato)
          if (DEBUG == 2) {
            print(s"(D) Inserted ( ${candidato +
1},${minF(candidato)},${minF(candidato)},${craya2F(candidato)} ) HeapF = ")
            mostrarPila(HeapF)
          }
        }
      }

      val p = G.nodo(candidato).LS

      // println ( "(D) ------------------------------------------------- 2.3")
      for (j <- 0 to p.length - 1) {
        if (p(j).nodo != G.s) {
          k = p(j).nodo
          et.d1 = p(j).c1 + etmin.d1
          et.d2 = p(j).c2 + etmin.d2
          dif = minF(k) - et.d1
          //          if ( DEBUG == 2) { println(s"k ${k} et.d1 ${et.d1} et.d2
${et.d2} dif ${dif}")  }
          val xpaso = numberiF(k) - 1
          if ((numberiF(k) > 0) && (et.d1 > LF(k)(xpaso).d1) && (et.d2 <
LF(k)(xpaso).d2) && (et.d2 <= dI2(k))) {
            if ((dif > 0) || ((predminF(k) != -1) && (dif == 0) && (et.d2 <
craya2F(k)))) {
              temp1 = et.d1 + d11(k)
              temp2 = et.d2 + d22(k)
              if ((temp1 <= d21(G.s)) && (temp2 <= d12(G.s)) && (temp2 <
LF(G.t)(numberiF(G.t) - 1).d2)) {
                if (predminF(k) == -1) {
                  et.elemento = k
                  et.clave = et.d1
                  // ---- 2.3.1
                  HeapF += Tipo_Elemento_Pila(elemento = et.elemento, clave =
et.clave, d1 = et.d1, d2 = et.d2)
                  //                  println(s"I ${et.elemento}")
                  contador_insertar_pila += 1
                  if (DEBUG == 2) {
                    print(s"(D) Inserted ( ${et.elemento +
```

```
1},${et.clave},${et.d1},${et.d2} ) HeapF = ")
                    mostrarPila(HeapF)
                  }
                  nlabels += 1
                }
                else {
                  DecreaseKey2(HeapF, et.d1, et.d1, et.d2, k)
                  contador_modificar_pila += 1;
                }
                minF(k) = et.d1
                predminF(k) = candidato
                predminpointerF(k) = numberiF(candidato) - 1
                craya2F(k) = et.d2
                if (et.d2 < min2H) min2H = et.d2
              }
            }
          }
        }
      }


    }
    nds = numberiF(G.t) + nds - G.nodos
    nL = numberiF(G.t)
    // ttime
    // if (DEBUG > 0) PrintPath(G.s, G.t, nL, LF)
    // PrintCost(G.s, G.t, nL, LF)

    Tipo_Resultado(0, nL = nL, nds = nds, nextract = eheaps, nlabels = nlabels)

  }
}
```

P2PV3PLUS.scala

```
package P2PV2PLUS.model

import scala.collection.mutable
import scala.collection.mutable.{ArrayBuffer, ListBuffer}
import Utilidades._

import scala.concurrent.{Await, Future}
import scala.concurrent.duration._
import scala.concurrent.ExecutionContext.Implicits.global
import scala.util.{Failure, Success}

object P2PV3PLUS extends App {

  def Revisar_LP(G: Tipo_Grafo, etmin: Tipo_Elemento_Pila, minF: Array[Int],
numberiF: Array[Int],
                 LF: Array[ArrayBuffer[Tipo_Label]], dI2: Array[Int], d11:
Array[Int], d12: Array[Int],
                 d21: Array[Int], d22: Array[Int], predminF: Array[Int], craya2F:
Array[Int], predminpointerF: Array[Int],
                 HeapF: Tipo_Cola_prioridad, candidato: Int, CurrentArcF:
Array[Array[Int]]): Int = {
    var k: Int = 0
    var temp1: Int = 0
    var temp2: Int = 0
    var cr1: Int = 0
    var cr2: Int = 0
    var dif: Int = 0
    var NonStop = true
```

```
    var min2H: Int = 0
    var i: Int = 0
    var seguir = true
    var et = Tipo_Elemento_Pila(0, 0, 0, 0)
    var nlabels: Int = 0

    if (candidato != G.s) {
      val p = G.nodo(candidato).LP
      for (j <- 0 to p.length - 1) { // Paraleismos sobre k distintos ?
        //      contador_insertar_bucle2 += 1
        k = p(j).nodo
        i = CurrentArcF(candidato)(j)
        seguir = true
        // ------------------------------------ 2.2
        while ((i >= 0) && (i < numberiF(k)) && seguir) {
          cr1 = p(j).c1 + LF(k)(i).d1
          cr2 = p(j).c2 + LF(k)(i).d2
          val xpaso = numberiF(candidato) - 1
          if ((cr1 > LF(candidato)(xpaso).d1) && (cr2 < LF(candidato)(xpaso).d2)
&& (cr2 <= dI2(candidato))) {
            temp1 = cr1 + d11(candidato)
            temp2 = cr2 + d22(candidato)
            if ((temp1 <= d21(G.s)) && (temp2 <= d12(G.s)) && (temp2 <
LF(G.t)(numberiF(G.t) - 1).d2)) {
              dif = -cr1 + minF(candidato)
              seguir = false
              if ((dif > 0) || ((predminF(candidato) != -1) && (dif == 0) && (cr2
< craya2F(candidato)))) {
                minF(candidato) = cr1
                predminpointerF(candidato) = i
                predminF(candidato) = k
                craya2F(candidato) = cr2
              }
            }
          }
          if (seguir) i += 1
        }
        CurrentArcF(candidato)(j) = i
      }

    }
    nlabels
  }


  // Revisar_LS ( G, etmin, minF, numberiF, LF, dI2, d21, d12, predminF, craya2F,
HeapF, candidato)
  def Revisar_LS(G: Tipo_Grafo, etmin: Tipo_Elemento_Pila, minF: Array[Int],
numberiF: Array[Int],
                 LF: Array[ArrayBuffer[Tipo_Label]], dI2: Array[Int], d11:
Array[Int], d12: Array[Int],
                 d21: Array[Int], d22: Array[Int], predminF: Array[Int], craya2F:
Array[Int], predminpointerF: Array[Int],
                 HeapF: Tipo_Cola_prioridad, candidato: Int): Int = {
    var k: Int = 0
    var temp1: Int = 0
    var temp2: Int = 0
    var cr1: Int = 0
    var cr2: Int = 0
    var dif: Int = 0
    var NonStop = true
    var min2H: Int = 0
    var i: Int = 0
    var seguir = true
    var et = Tipo_Elemento_Pila(0, 0, 0, 0)
```

```
    var nlabels: Int = 0

    val p = G.nodo(candidato).LS
    // println ( "(D) ------------------------------------------------- 3")
    for (j <- 0 to p.length - 1) { // Paralelizable si k distintos
      if (p(j).nodo != G.s) {
        //       contador_insertar_bucle3 += 1
        k = p(j).nodo
        et.d1 = p(j).c1 + etmin.d1
        et.d2 = p(j).c2 + etmin.d2
        dif = minF(k) - et.d1
        //          if ( DEBUG == 2) { println(s"k ${k} et.d1 ${et.d1} et.d2
${et.d2} dif ${dif}")  }
        val xpaso = numberiF(k) - 1
        if ((numberiF(k) > 0) && (et.d1 > LF(k)(xpaso).d1) && (et.d2 <
LF(k)(xpaso).d2) && (et.d2 <= dI2(k))) {
          if ((dif > 0) || ((predminF(k) != -1) && (dif == 0) && (et.d2 <
craya2F(k)))) {
            temp1 = et.d1 + d11(k)
            temp2 = et.d2 + d22(k)
            if ((temp1 <= d21(G.s)) && (temp2 <= d12(G.s)) && (temp2 <
LF(G.t)(numberiF(G.t) - 1).d2)) {
              et.elemento = k
              et.clave = et.d1
              if (predminF(k) == -1) {
                Insert2(HeapF, Tipo_Elemento_Pila(elemento = et.elemento, clave =
et.clave, d1 = et.d1, d2 = et.d2))
                //               contador_insertar_pila += 1
                //               contador_insertar_p3 += 1
                if (DEBUG == 2) {
                  print(s"(D) Inserted ( ${et.elemento +
1},${et.clave},${et.d1},${et.d2} ) HeapF = ")
                  mostrarPila2(HeapF)
                }
                nlabels += 1
              }
              else {
                Decrease2(HeapF, et)
                //
                //                contador_modificar_pila += 1;
              }
              minF(k) = et.d1
              predminF(k) = candidato
              predminpointerF(k) = numberiF(candidato) - 1
              craya2F(k) = et.d2
              if (et.d2 < min2H) min2H = et.d2
            }
          }
        }
      }
    }
    nlabels
  }


  def BicriteriaPathAlgorithm_p2pv3(G: Tipo_Grafo, Singledestination: Int):
Tipo_Resultado = {
    // P2PV2PLUS
    var eheaps: Int = 0
    var nL: Int = 0
    var nds: Int = 0
    var nlabels: Int = 0
    val dg = G.nodos
    val predminF: Array[Int] = new Array(dg)
    val predminB: Array[Int] = new Array(dg)
```

```
    //    val predminpointerB: Array[Int] = new Array(dg)
    val predminpointerF: Array[Int] = new Array(dg)
    val CurrentArcF = new Array[Array[Int]](dg)
    //   val CurrentArcB = new Array[Array[Int]](dg)
    val numberiF: Array[Int] = new Array(dg)
    val numberiB: Array[Int] = new Array(dg)

    val dI1: Array[Int] = new Array(dg)
    val dI2: Array[Int] = new Array(dg)
    //    val dP1: Array[Int] = new Array(dg)
    //    val dP2: Array[Int] = new Array(dg)
    val d11: Array[Int] = new Array(dg)
    val d12: Array[Int] = new Array(dg)
    val d21: Array[Int] = new Array(dg)
    val d22: Array[Int] = new Array(dg)


    // val HeapF = mutable.PriorityQueue.empty(Orden_pila)
    // val HeapF = mutable.PriorityQueue.empty(Orden_pila)
    val HeapF = new Tipo_Cola_prioridad(new Array(G.nodos), new Array(G.nodos), n
= 0);

    // val HeapB = mutable.PriorityQueue.empty(Orden_pila)

    var LF = new Array[ArrayBuffer[Tipo_Label]](dg)
    //    var LB = new Array[ListBuffer[Tipo_Label]](dg)
    //    var L = new Array[ListBuffer[Tipo_Label]](dg)

    val craya2F: Array[Int] = new Array(dg)
    //    val craya2B: Array[Int] = new Array(dg)
    val minF: Array[Int] = new Array(dg)
    //    val minB: Array[Int] = new Array(dg)
    //    val positionF: Array[Int] = new Array(dg + 1)
    //    val positionB: Array[Int] = new Array(dg + 1)
    //    var p: Array[Tipo_Arco] = Array()
    var k: Int = 0
    var cr1: Int = 0
    var cr2: Int = 0
    var temp1: Int = 0
    var temp2: Int = 0
    var dif: Int = 0
    var NonStop = true
    var candidato: Int = 0
    var min2H: Int = 0
    var i: Int = 0
    var seguir = true

    var contador_insertar_pila = 0
    var contador_modificar_pila = 0
    var contador_insertar_p1 = 0
    var contador_insertar_p2 = 0
    var contador_insertar_p3 = 0
    var contador_insertar_bucle2 = 0
    var contador_insertar_bucle3 = 0


    var etmin = new Tipo_Elemento_Pila(0, 0, 0, 0)
    var et = new Tipo_Elemento_Pila(0, 0, 0, 0)
    //    var nuevo = new Tipo_Label(0, 0, 0, 0, 0)


    eheaps = 0
    nL = 0
    nds = 0
    nlabels = 0
```

```
    if (PARALELO == 2) {
      var h1 = 0
      var h2 = 0
      var h3 = 0

      val dijkstra1 = Future {
        h1 = Dijkstra3(G, G.t, d11, predminB, d12, -1)
      }
      val dijkstra2 = Future {
        h2 = Dijkstra3(G, G.t, d22, predminF, d21, -2)
      }
      val dijkstra3 = Future {
        h3 = Dijkstra3(G, G.s, dI1, predminF, dI2, 1)
      }
      //    println("COMIENZAN DIJSTRA ...")
      val fs: Future[Unit] = for {
        eh1 <- dijkstra1
        eh2 <- dijkstra2
        eh3 <- dijkstra3
      } yield {}

      Await.result(fs, 50.seconds)
      //    println("ACABAN DIJSTRA ...",h1,h2,h3)
      eheaps = h1 + h2 + h3
    }
    else {
      eheaps += Dijkstra3(G, G.t, d11, predminB, d12, -1)
      eheaps += Dijkstra3(G, G.t, d22, predminF, d21, -2)
      eheaps += Dijkstra3(G, G.s, dI1, predminF, dI2, 1)
    }


    // Inicializar HeapF
    min2H = INF
    // ------------------------------------------- (0)
    if (DEBUG == 2) println("-----------------------------------------------------
Paso 0")
    for (i <- 0 to G.nodos - 1) {
      LF(i) = new ArrayBuffer()
      if ((predminF(i) != UERROR2) && (predminB(i) != UERROR2)) {
        numberiF(i) = 1
 //
        LF(i) ++= ArrayBuffer.fill(2)(Tipo_Label(0,0,0,0,0))
 //
        LF(i)(numberiF(i)-1) = Tipo_Label(pred = predminF(i), d1 = dI1(i), d2 =
dI2(i), 0, 0) // possucc no se usa
        CurrentArcF(i) = new Array[Int](G.nodo(i).LP.length)
        if (DEBUG == 2) {
          print(s"LF(${i + 1}) = ");
          mostrarLista(LF(i))
        }
        nds += 1
      }
      else {
        numberiF(i) = 0
//        LF(i) = new ArrayBuffer()
        CurrentArcF(i) = new Array[Int](0)
      }
    }
    // ---------------------------------- (1)
    if (DEBUG == 2) println("-----------------------------------------------------
Paso 1")
```

```
    for (i <- 0 to G.nodos - 1) { // INDEPENDIENTES se podria paralelizar sobre i
      if ((i != G.s) && (numberiF(i) > 0)) {
        minF(i) = INF
        predminF(i) = -1
        val p = G.nodo(i).LP
        for (j <- 0 to p.length - 1) {
          k = p(j).nodo
          if (numberiF(k) > 0) {
            //         println(s"CurrentArcF(${i},${j} y dimension es
${CurrentArcF(i).size}")
            CurrentArcF(i)(j) = 0
            cr1 = p(j).c1 + dI1(k)
            cr2 = p(j).c2 + dI2(k)
            temp1 = p(j).c1 + dI1(k) + d11(i)
            temp2 = p(j).c2 + dI2(k) + d22(i)
            if ((cr1 > LF(i)(0).d1) && (cr2 < LF(i)(0).d2)) {
              if ((temp1 <= d21(G.s)) && (temp2 <= d12(G.s)) && (temp2 <
LF(G.t)(0).d2)) {
                dif = minF(i) - cr1
                if ((dif > 0) || ((predminF(i) != -1) && (dif == 0) && (craya2F(i)
> cr2))) {
                  minF(i) = cr1
                  craya2F(i) = cr2
                  predminF(i) = k
                  predminpointerF(i) = 0
                }
              }
            }
          }
          else
            CurrentArcF(i)(j) = -1
        }
        if (predminF(i) != -1) {
          //
          Insert2(HeapF, Tipo_Elemento_Pila(elemento = i, clave = minF(i), d1 =
minF(i), d2 = craya2F(i)))
          //
          contador_insertar_pila += 1
          contador_insertar_p1 += 1

          if (DEBUG == 2) {
            print(s"(D) Inserted ( ${i + 1},${minF(i)},${minF(i)},${craya2F(i)} )
HeapF = ")
            mostrarPila2(HeapF)
          }

          if (craya2F(i) < min2H) min2H = craya2F(i)
          nlabels += 1
        }
      }
    }
    // ------------------------------------------------ (2)
    if (DEBUG == 2) println("-------------------------------------------------
Paso 2")
    NonStop = true

    while (HeapF.n > 0 && NonStop) {
      //
      // ------------------------------------------- (2.1) FORWARD
      etmin = DeleteMin2(HeapF)
      //      etmin = HeapF.dequeue()
      if (DEBUG == 2) {
        print(s"Extraido( ${etmin.elemento +
1},${etmin.clave},${etmin.d1},${etmin.d2} ) HeapF = ")
        mostrarPila2(HeapF)
```

```
      }
      eheaps += 1
      candidato = etmin.elemento

      val nic = numberiF(candidato)
      numberiF(candidato) += 1
      if (numberiF(candidato) > LF(candidato).length )
            LF(candidato) ++=
ArrayBuffer.fill(LF(candidato).length)(Tipo_Label(0,0,0,0,0))
      LF(candidato)(nic) = Tipo_Label(pred = predminF(candidato), d1 = etmin.d1,
d2 = etmin.d2, pospred = predminpointerF(candidato), possucc = 0)
      //
      if (DEBUG == 2) {
        print(s"LF(${candidato + 1}) = ");
        mostrarLista(LF(candidato))
      }
      nds += 1
      predminF(candidato) = -1
      minF(candidato) = INF
      predminpointerF(candidato) = -1
      val lft_d2 = LF(G.t)(numberiF(G.t) - 1).d2
      val d21_s = d21(G.s)
      val d12_s = d12(G.s)


      if (PARALELO == 3 ) {
        //
        //  Muy pequeño para paralelizar
        //
        if ( PARALELO == 2 ) {
          var nlabels1 = 0
          var nlabels2 = 0


          val revisar_lp = Future {
            nlabels1 += Revisar_LP(G, etmin, minF, numberiF, LF, dI2, d11, d12,
d21, d22, predminF, craya2F, predminpointerF, HeapF, candidato, CurrentArcF)
          }
          val revisar_ls = Future {
            nlabels2 += Revisar_LS(G, etmin, minF, numberiF, LF, dI2, d11, d12,
d21, d22, predminF, craya2F, predminpointerF, HeapF, candidato)
          }
          //    println("COMIENZAN DIJSTRA ...")
          val fs: Future[Unit] = for {
            eh1 <- revisar_lp
            eh2 <- revisar_ls
          } yield {}

          Await.result(fs, 50.seconds)
          //    println("ACABAN DIJSTRA ...",h1,h2,h3)
          nlabels += (nlabels1 + nlabels2)
        }
        if ( PARALELO == 1 ) {
          nlabels += Revisar_LP(G, etmin, minF, numberiF, LF, dI2, d11, d12, d21,
d22, predminF, craya2F, predminpointerF, HeapF, candidato, CurrentArcF)
          nlabels += Revisar_LS(G, etmin, minF, numberiF, LF, dI2, d11, d12, d21,
d22, predminF, craya2F, predminpointerF, HeapF, candidato)
        }


        if (predminF(candidato) != -1) {
          Insert2(HeapF, Tipo_Elemento_Pila(elemento = candidato, clave =
minF(candidato), d1 = minF(candidato), d2 = craya2F(candidato)))
          nlabels += 1
          if (craya2F(candidato) < min2H) min2H = craya2F(candidato)
```

```
          if (DEBUG == 2) {
            print(s"(D) Inserted ( ${candidato +
1},${minF(candidato)},${minF(candidato)},${craya2F(candidato)} ) HeapF = ")
            mostrarPila2(HeapF)
          }
        }

      }
      else {

        if (candidato != G.s) {
          val p = G.nodo(candidato).LP
          val dI2_c = dI2(candidato)
          val xpaso = numberiF(candidato) - 1
          val lfc_d1 = LF(candidato)(xpaso).d1
          val lfc_d2 = LF(candidato)(xpaso).d2
          for (j <- 0 to p.length - 1) {
            contador_insertar_bucle2 += 1
            k = p(j).nodo
            i = CurrentArcF(candidato)(j)
            seguir = true
            val nik = numberiF(k)

            // ---------------------------------------- 2.2
            while ((i >= 0) && (i < nik ) && seguir) {
              cr1 = p(j).c1 + LF(k)(i).d1
              cr2 = p(j).c2 + LF(k)(i).d2

              if ((cr1 > lfc_d1 ) && (cr2 < lfc_d2 ) && (cr2 <= dI2_c )) {
                temp1 = cr1 + d11(candidato)
                temp2 = cr2 + d22(candidato)
                if ((temp1 <= d21_s ) && (temp2 <= d12_s ) && (temp2 < lft_d2 )) {
                  dif = -cr1 + minF(candidato)
                  seguir = false
                  if ((dif > 0) || ((predminF(candidato) != -1) && (dif == 0) &&
(cr2 < craya2F(candidato)))) {
                    minF(candidato) = cr1
                    predminpointerF(candidato) = i
                    predminF(candidato) = k
                    craya2F(candidato) = cr2
                  }
                }
              }
              if (seguir) i += 1
            }
            CurrentArcF(candidato)(j) = i
          }
          if (predminF(candidato) != -1) {
            //         HeapF += Tipo_Elemento_Pila(elemento = candidato, clave =
minF(candidato), d1 = minF(candidato), d2 = craya2F(candidato))
            Insert2(HeapF, Tipo_Elemento_Pila(elemento = candidato, clave =
minF(candidato), d1 = minF(candidato), d2 = craya2F(candidato)))
            nlabels += 1
            // println(s"I ${candidato}")
            contador_insertar_pila += 1
            contador_insertar_p2 += 1

            if (craya2F(candidato) < min2H) min2H = craya2F(candidato)
            if (DEBUG == 2) {
              print(s"(D) Inserted ( ${candidato +
1},${minF(candidato)},${minF(candidato)},${craya2F(candidato)} ) HeapF = ")
              mostrarPila2(HeapF)
            }
          }
        }
```

```
      }



      val p = G.nodo(candidato).LS
      // println ( "(D) ------------------------------------------------- 3")
 //     val lft_d2 = LF(G.t)(numberiF(G.t) - 1).d2
 //     val d21_s = d21(G.s)
 //     val d12_s = d12(G.s)
      val dI2_c = dI2(candidato)
      val xpaso = numberiF(candidato) - 1


      for (j <- 0 to p.length - 1) { // Paralelizable si k distintos
        if (p(j).nodo != G.s) {
          contador_insertar_bucle3 += 1
          k = p(j).nodo
          et.d1 = p(j).c1 + etmin.d1
          et.d2 = p(j).c2 + etmin.d2
          dif = minF(k) - et.d1
          val nik = numberiF(k)
          val lfk_d1 = LF(k)(nik - 1).d1
          val lfk_d2 = LF(k)(nik - 1).d2
          val dI2_k = dI2(k)

          //          if ( DEBUG == 2) { println(s"k ${k} et.d1 ${et.d1} et.d2
${et.d2} dif ${dif}")  }


          if ((nik > 0) && (et.d1 > lfk_d1 ) && (et.d2 < lfk_d2 ) && (et.d2 <=
dI2_k)) {
            if ((dif > 0) || ((predminF(k) != -1) && (dif == 0) && (et.d2 <
craya2F(k)))) {
              temp1 = et.d1 + d11(k)
              temp2 = et.d2 + d22(k)
              if ((temp1 <= d21_s) && (temp2 <= d12_s) && (temp2 < lft_d2)) {
                et.elemento = k
                et.clave = et.d1
                if (predminF(k) == -1) {
                  Insert2(HeapF, Tipo_Elemento_Pila(elemento = et.elemento, clave
= et.clave, d1 = et.d1, d2 = et.d2))
                  contador_insertar_pila += 1
                  contador_insertar_p3 += 1
                  if (DEBUG == 2) {
                    print(s"(D) Inserted ( ${et.elemento +
1},${et.clave},${et.d1},${et.d2} ) HeapF = ")
                    mostrarPila2(HeapF)
                  }
                  nlabels += 1
                }
                else {
                  Decrease2(HeapF, et)
                  //
                  contador_modificar_pila += 1;
                }
                minF(k) = et.d1
                predminF(k) = candidato
                predminpointerF(k) = numberiF(candidato) - 1
                craya2F(k) = et.d2
                if (et.d2 < min2H) min2H = et.d2
              }
            }
          }
        }
```

```
    }
    //------------------------------------------- HASTA AQUI

  }

  nds = numberiF(G.t) + nds - G.nodos
  nL = numberiF(G.t)
  // ttime
  if (DEBUG > 0) PrintPath(G.s, G.t, nL, LF)
  // PrintCost( G.s, G.t, nL , LF)
  println
  print(s"(INSERT ${contador_insertar_pila} MODIFY ${contador_modificar_pila} P1 =
${contador_insertar_p1} P2 = ${contador_insertar_p2} P3 = ${contador_insertar_p3}
")
  println(s"BUCLE2 = ${contador_insertar_bucle2} BUCLE3 =
${contador_insertar_bucle3} )")
    var contador_lf:Int = 0
    var contador_lf_alloc:Int = 0
    for (i <- 0 to G.nodos -1) {
      contador_lf += numberiF(i)
      contador_lf_alloc += LF(i).length
    }
    println(s"LF  ${contador_lf} elementos, reservado ${contador_lf_alloc} )")

    Tipo_Resultado(0, nL = nL, nds = nds, nextract = eheaps, nlabels = nlabels)

}

}
```

Apéndice B. Proyecto ShortestPath- Usando pregel.

package.scala - Variables globales del proyecto

Tipo_Datos.scala - Tipos de Datos usados.

Main.scala - Programa principal, usando spark y graphx

SSSP.scala - Búsqueda de camino de coste mínimo (considerando solo una dimensión) usando pregel.

BSSSP.scala - Búsqueda de camino de coste mínimo (2 dimensiones) usando pregel.

package.scala

```
package ShortestPath

import ShortestPath.model._

  package object model {

    val Maxcost = 100000000
```

```
    val UERROR2 = 100000000
    val INF = 1000000000
    val DEBUG = 1 // 0 -NODEBUG, 1-MEDIUM, 2 -FULL
    var PARALELO = 1     // 0 -NODEBUG, 1-MEDIUM, 2 -FULL
    val MinData = (- INF )

  }
```

Tipo_datos.scala

```
package ShortestPath.model

import org.apache.spark.graphx.{Edge, VertexId}

import scala.io.Source

case class Tipo_Arco(
                      d1: Int,
                      d2: Int
                    )

case class Tipo_Label(
                       d1: Int,
                       d2: Int,
                       pre: VertexId
                     )

case class Tipo_Grafo(var v: Array[(VertexId, Int)],
                      var e: Array[Edge[Tipo_Arco]],
                      var s: VertexId,
                      var t: VertexId,
                      var nodos: Int,
                      var arcos: Int,
                      var maxc1: Int,
                      var maxc2: Int

                     ) {
  def CargarFichero(fichero: String) {
    val source = Source.fromFile(fichero)
    val lines = source.getLines()
    val Array(x1, x2, x3, x4, _*) = lines.next.split(" ")
    val matrix2 = lines.map(_.split(" ")).toArray
    source.close()
    val nnodos = x3.toInt
    val narcos = x4.toInt
    if ((nnodos > 0) && (narcos > 0)) {
      nodos = nnodos
      arcos = narcos
      s = 1
      t = nnodos
      maxc1 = 0
      maxc2 = 0

      v = new Array(nnodos)
      e = new Array(narcos)
      for (i <- 1 to nnodos) v(i - 1) = (i, 0)

      //
      // INICIALIZAR EN GRAFO
      //
      var k = 0
      for (m <- matrix2) {
        val a = m(0)
```

```
        if (a == "a") {
          val i = m(1).toInt
          val j = m(2).toInt
          val u = m(3).toInt
          val ut = m(4).toInt
          //       println(s"Elemento ( $a $i $j $u $ut)")
          e(k) = new Edge(i, j, new Tipo_Arco(u, ut))
          k += 1
          if (u > maxc1) maxc1 = u
          if (ut > maxc2) maxc2 = ut
        }

      }
      /*


      if (DEBUG > 0) { // Ver si interesa sacar Resumen del Grafo
        println(s"Grafo con ${nodos} nodos y ${arcos} arcos, MaxC1 = ${maxc1},
MaxC2 = ${maxc2} s = $s t = $t")
      }
*/
    }
    else {
      println("el formato del comando es NS <nombrefichero>")
      println("el formato del fichero es el siguiente      ")
      println("============================================")
      println("s sp  NODOS(n)  ARCOS(m)                    ")
      println("a i1     j1   c[i1,j1] t[i1,j1]             ")
      println("             . . .                          ")
      println("a im     jm   c[im,jm] t[im,jm]             ")
      println("============================================")
    }
  }
}
```

Main.scala

```
package ShortestPath.model

import org.apache.log4j.{Level, Logger}

import scala.collection.mutable
// import org.apache.spark.examples.graphx.SSSPExample.{Camino_Minimo_Unicriterio,
DEBUG}
// import org.apache.spark.examples.graphx.{Tipo_Arco, Tipo_Grafo}
import org.apache.spark.graphx.{Graph, VertexId}
import org.apache.spark.sql.SparkSession



object Main {
  def main(args: Array[String]): Unit = {
    Programa.Programa ( args )
  }
}

object Programa extends App {
```

```
def Programa(args: Array[String]): Unit = {

    // Creates a SparkSession.
    Logger.getLogger("org.apache.spark").setLevel(Level.WARN)
    val spark = SparkSession
      .builder
      .appName(s"${this.getClass.getSimpleName}")
      //
      .master("local[8]")
      .getOrCreate()
    val sc = spark.sparkContext
    var t0: Long = 0
    var t1: Long = 0
    var t2: Long = 0
    var t3: Long = 0
    var elapsed_time: Double = 0




    val G = new Tipo_Grafo(new Array(0), new Array(0), 0, 0, 0, 0, 0, 0)

    var nombrefichero = "/home/usuario/TFM/BSPA/rome99.txt"
    if (args.length > 0) nombrefichero = args(0)
    // ---------------------- PASO 0
    t0 = System.nanoTime()

    //
    G.CargarFichero(nombrefichero)
    //
    val V = sc.makeRDD(G.v)
    //
    val E = sc.makeRDD(G.e)

    val sourceId: VertexId = G.s // The ultimate source
    //
    val graph = Graph(V, E)
  //

    t1 = System.nanoTime()

//  val sssp = SSSP.SSSP_unicriteria(graph, sourceId)
   val sssp = BSSSP.SSSP_bicriteria(graph, sourceId)

  if (DEBUG == 2) {

      println("Grafo ")

      println(graph.vertices.collect.mkString("\n"))
      println(graph.edges.collect.mkString("\n"))

      //   println("Grafo INITIALGRAPH")
      //   println(initialGraph.vertices.collect.mkString("\n"))
      println("SOLUCION")
      println(sssp.vertices.collect.mkString("\n"))
    }

    t2 = System.nanoTime()
    if (DEBUG == 2)     println("SOLUCION ", sssp.vertices.filter((x) => (x._1 ==
G.t)).collect.mkString("\n"))

    elapsed_time = ((t2 - t1) / 1000.0 / 1000.0 / 1000.0)
    print(s"BiCriteria Shortest Path (PREGEL) ${nombrefichero} ${G.nodos}
${G.arcos} ${G.s} ${G.t} ${elapsed_time} ")
    for (x <- sssp.vertices.filter((x) => (x._1 == G.t)).collect) {
```

```
      print( x._2.size)
    }
  println

  if ( DEBUG == 1) {
    for (x <- sssp.vertices.filter((x) => (x._1 == G.t)).collect) {
      for (k <- x._2 )
      {
        println(s" ${k}")
      }
    }


      println
    elapsed_time = ((t1 - t0) / 1000.0 / 1000.0 / 1000.0)
    println(s" T Leer Grafo ${elapsed_time} ")
    elapsed_time = ((t2 - t1) / 1000.0 / 1000.0 / 1000.0)
    println(s" T Calcular Coste Minimo ${elapsed_time} ")
  }
    //    println(s"${elapsed_time} ${r.nL} ${r.nds} ${r.nextract} ${r.nlabels}")

    // $example off$

    spark.stop()
  }


}
```

SSSP.scala

```
package ShortestPath.model

import org.apache.spark.graphx.{Graph, VertexId}

import scala.collection.mutable

object SSSP extends App {

  def SSSP_unicriteria(graph: Graph[Int, Tipo_Arco], sourceId: VertexId):
Graph[Tipo_Label, Tipo_Arco] = {
    // Initialize the graph such that all vertices except the root have distance
infinity.
    val initialGraph = graph.mapVertices((id, _) =>
      if (id == sourceId)    Tipo_Label(0, 0, sourceId)
      else                   Tipo_Label( INF, INF, 0))

    val MAXIMO = graph.numEdges.toInt * 2
    val sssp = initialGraph.pregel(
      Tipo_Label( INF, INF, 0), MAXIMO)(
      (id, dist, newDist) => {
        if (newDist.d1 < INF ) newDist // Discard Infinite
        else dist
      }, // Vertex Program
      triplet => { // Send Message
        if (triplet.srcAttr.d1 + triplet.attr.d1 < triplet.dstAttr.d1) {
          Iterator((triplet.dstId,
            Tipo_Label(triplet.srcAttr.d1 + triplet.attr.d1, triplet.srcAttr.d2 +
triplet.attr.d2, triplet.srcId)))
        } else {
          Iterator.empty
        }
```

```
      },
      (a, b) => {
        if (a.d1 < b.d1) a else b
      } // Merge Message
    )
    sssp
  }

}
```

BSSSP.scala

```
package ShortestPath.model

import org.apache.spark.graphx.{Graph, VertexId}

import scala.collection.mutable

object BSSSP extends App {
  // Ver si "y" está dominado por algún XSet(j)
  def Dominado(y: Tipo_Label, XSet: mutable.Set[Tipo_Label]): Boolean = {
    var dom = false
    for (x <- XSet) {
      if ((x.d1 <= y.d1) && (x.d2 <= y.d2)) dom = true
    }
    dom
  }

  // Insertar "y" dentro de XSet
  def Insertar(y: Tipo_Label, XSet: mutable.Set[Tipo_Label]): Unit = {
    if ((y.d1 < INF )) {
      if (!Dominado(y, XSet)) { // y no dominado por Xset
        BorrarDominados(y, XSet)
        XSet += y
        //       println(s"Elemento ${y} INSERTADO")
      }
     }
  }

  // Eliminar de XSet todos los elementos dominados por y
  def BorrarDominados(y: Tipo_Label, XSet: mutable.Set[Tipo_Label]): Unit = {

    var YSet = mutable.Set[Tipo_Label]()
    for (x <- XSet) {
      if ((y.d1 < x.d1) || (y.d1 == x.d2) && (x.d2 < y.d2)) {
        YSet += x
        //       println(s"Eliminado elemento ${x} por estar dominado por ${y}")
      }
    }
    XSet --= YSet
  }

  def SSSP_bicriteria(graph: Graph[Int, Tipo_Arco], sourceId: VertexId):
Graph[mutable.Set[Tipo_Label], Tipo_Arco] = {
    //
    // Initialize the graph such that all vertices except the root are Empty
    //
    val initialGraph = graph.mapVertices((id, _) =>
      if (id == sourceId) mutable.Set[Tipo_Label](Tipo_Label(0, 0, sourceId))
      else mutable.Set[Tipo_Label]())

    val MAXIMO = (graph.numEdges * 2).toInt  // MAXIMO can be INF
```

```
    val sssp = initialGraph.pregel(
      mutable.Set(Tipo_Label( INF , INF, 0)), MAXIMO)(
      (id, dist, newDist) => {
        val nuevo = dist.clone()

        for (x <- newDist)
          Insertar(x, nuevo)
        //       println(s"NODO ${id} Insertado ${newDist} en ${nuevo}")
        nuevo
      }, // Vertex Program
      triplet => { // Send Message
        var it: Iterator[(VertexId, mutable.Set[Tipo_Label])] = Iterator.empty
        var XSet = mutable.Set[Tipo_Label]()
        for (k <- triplet.srcAttr) {
          val x = Tipo_Label(k.d1 + triplet.attr.d1, k.d2 + triplet.attr.d2,
triplet.srcId)
          if (k.d1 < INF )
            if (!Dominado(x, triplet.dstAttr)) {
              XSet += x
            }
        }

        if (XSet.size > 0) {
          // println(s" ${triplet.srcId} -> ${triplet.dstId} ORI
${triplet.srcAttr}  DST ${triplet.dstAttr} NEW ${XSet}")
          Iterator((triplet.dstId, XSet))
        }
        else Iterator.empty

      },

      (a, b) => {
        for (x <- a)
          Insertar(x, b)
        b
      } // Merge Message


    )

     sssp

  }

}
```