UNIVERSIDAD DE TALCA
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA CIVIL EN COMPUTACIÓN

Diseño e Implementación de Algoritmos de Grafos para Apache Giraph

MATÍAS ALEJANDRO DÍAZ BESOAIN

Profesor Guía: RENZO ANGLES

Memoria para optar al título de
Ingeniero Civil en Computación

Curicó – Chile
11 de Agosto de 2017

CONSTANCIA

La Dirección del Sistema de Bibliotecas a través de su encargado Biblioteca Campus Curicó certifica que el autor del siguiente trabajo de titulación ha firmado su autorización para la reproducción en forma total o parcial e ilimitada del mismo.

Curicó, 2019

Dedicado a todas las almas curiosas del saber,
por ustedes es que la humanidad ha visto
las más grandes revoluciones del pensamiento.
Matías Díaz B.

AGRADECIMIENTOS

A mis padres por acompañarme, ayudarme y guiarme en el largo camino de la educación. Sin ellos no hubiera podido conseguir ni la mitad de los pequeños logros que hasta ahora he conseguido.

A mi hermano por ser un modelo de líderazgo y una influencia positiva desde temprana edad. Gracias por introducirme también al glorioso pero sufrido mundo de la ingeniería. Sin duda distinto hubiera sido mi camino de vida si tú antes no hubieras trazado las rutas para yo seguir.

A mis compañeros de universidad, con los que juntos vivimos este proceso de crecimiento como personas y como profesionales. Gracias por ser testigos fieles de las muchas caídas que se viven en el proceso universitario. Gracias porque sin duda las risas y bromas entre las clases aliviaron el camino e hicieron que no pareciera tan largo. Sin duda este trabajo tiene una cuota de apoyo de ustedes también.

A mis profesores guías por la entrega y compromiso para con el trabajo realizado durante este año de investigación. Gracias por el tiempo y preocupación entregados, ya que con ustedes este largo camino de memoria, que al principio parecía no tener forma, de a poco fue teniendo más sentido y logró convertirse en un fruto tangible como lo es esta memoria.

Gracias también a todos los que me apoyaron de alguna u otra forma a conseguir el objetivo final de terminar esta memoria. Porque cada pequeño empujón me acercó centímetro a centímetro a esta meta.

RESUMEN

En la presente memoria se presenta un trabajo de investigación realizado sobre Apache Giraph. Esta herramienta de procesamiento de grafos de gran tamaño fue lanzada, en su versión más estable el 2013. Como es reciente existen pocos algoritmos implementados con esta herramienta. Por ende el trabajo de investigación estuvo centrado en la generación de código Giraph para algoritmos tradicionales de grafos.

La metodología de trabajo fue realizada de manera incremental. Primero se investigó el estado del arte del tema, es decir, que algoritmos ya habían sido implementados para esta plataforma. Luego se consideraron algoritmos que no estuvieran implementados en Giraph y se adaptaron para seguir el modelo centrado en vértices de Giraph. Posteriormente se implementó uno de estos algoritmos y se realizaron pruebas exploratorias a fin de conocer el funcionamiento de esta plataforma experimentalmente. Una vez realizadas las pruebas, se procedió a extraer conclusiones sobre el proceso realizado.

El presente documento relata los hallazgos encontrados sobre el proceso y constituye una guía sobre el trabajo realizado y encamina futuros trabajos.

Palabras Clave: Apache Giraph, Teoría de Grafos, Algoritmos

ABSTRACT

This thesis presents a recollection of the research done using Apache Giraph. This graph processing framework was released on its most recent stable version on 2013. As a product of its recent release there are only a handful algorithms implemented using this framework. Ergo, the main goal of this research project was to generate Giraph code for traditional graph algorithms.

The work methodology considered an incremental approach on the implementation of algorithms. First a review of the state of the art was done, which allowed us to determinate what algorithms were already implemented using Giraph. After this, traditional graph algorithms that were not implemented for Giraph were taken into consideration and adapted to follow the Vertex Centered paradigm required for Giraph. Later on, a selected algorithm from this list was implemented and tested to determinate the correctness of the implementations. Finally after the testing was done, conclusions were extracted from the experience of the research.

This document presents the findings of the described research, and overall provides guidance of the work done for future investigations.

Keywords: Apache Giraph, Graph Theory, Algorithms

TABLA DE CONTENIDOS

página

ÍNDICE DE FIGURAS

página

ÍNDICE DE TABLAS

página

ÍNDICE DE ALGORITMOS

1. Introducción

En este capítulo se discuten los elementos centrales de la memoria a desarrollar, los cuales sirven como presentación de los aspectos genéricos del tema. En los capítulos siguientes se presentan los aspectos más específicos que involucran la memoria desarrollada.

En la sección 1.1 se presenta una descripción de la propuesta de la memoria desarrollada. Se incluyen detalles de contextualización del proyecto, una definición del problema identificado, una investigación sobre el estado del arte del tema y la propuesta de solución al problema identificado.

En la Sección 1.2 se presentan los objetivos que se busca alcanzar con este trabajo. Se detallan los objetivos generales que se busca conseguir, así como también los objetivos específicos de la memoria.

En la Sección 1.3 se detallan los alcances del trabajo realizado, es decir cuáles elementos fueron excluídos para poder terminar de buena manera el trabajo de memoria.

En la Sección 1.4 se explica la metodología a utilizada al momento de realizar la memoria. Se incluye también una planificación por fechas del trabajo realizado.

1.1. Descripción de la Propuesta

1.1.1. Contexto del Proyecto

Las organizaciones de todos los tamaños generan gran cantidad de información en los distintos procesos productivos y operacionales que desarrollan, lo que hace necesario contar con herramientas efectivas que faciliten la consulta y análisis de dicha información. En 1970 Edgar F.Codd publicó un paper [4] en el que definió el *Modelo relacional* que organiza los datos según las relaciones de estos. Posteriormente *IBM* desarrollo un lenguaje de consulta que permitía manipular datos almacenados en un *Sistema de gestión de base de datos(DBMS)*. Este modelo fue el estándar de la industria por largos años, pero al poco andar y con la evolución de los requerimientos de la industria sobre el volumen de datos y tiempo de respuesta de los sistemas, se fue observando que el modelo relacional no podría cumplir las expectativas de funcionamiento. En 1979 C. Beeri publicó un paper [2] en el que expusó que la eficiencia de un *schema* viene dada por la habilidad del diseñador para identificar correctamente las relaciones lógicas del minimundo a modelar. En 1998 Carlo Strozzi acuñó el término *NoSQL* en una base de datos que no utilizaba *SQL* como la interfaz de consulta, en lo que fue uno de los primeros avances en el desarrollo de una nueva tecnología. Actualmente existen muchas Bases de Datos del tipo *NoSQL* que maximizan el rendimiento de la consulta de datos y no se ven afectadas con la escalabilidad de las mismas.

En los últimos años el tratamiento de los datos a escalado a un nivel superior, en el que los datos debido a su naturaleza cambian frecuentemente, son muy diversos o de volumenes tan grandes que no es posible su modelamiento usando el modelo relacional o *NoSQL* por si solo, conocido como *Big Data*. En el 2010 *Google* publicó un paper [10] en el mostraba los avances en un modelo que permitía procesar grafos de gran tamaño implementando una arquitectura distribuida que divide el procesamiento en varias máquinas, lo cual entregaba mejores tiempos de computabilidad para el procesamiento de datos. Para llevar a cabo esto, se definió un modelo centrado en los vértices del grafo, lo que implicaba un cambio general en el paradigma de programación tradicional de grafos. Este trabajo fue donado por *Google* a la fundación *Apache* bajo una licencia open-source, lo que permitía que la comunidad pudiera hacer libre uso de este. En adelante este proyecto pasó a conocerce como *Apache Giraph*, la contraparte libre de *Pregel*. Posteriormente, en el año 2012 *Facebook* tomó el trabajo de

Pregel, le realizó algunas mejoras y también donó su trabajo a la fundación *Apache*. Actualmente *Apache Giraph* se perfila como uno de los mejores *frameworks* para el procesamiento de grafos a gran escala.

1.1.2. Definición del Problema

La programación en *Giraph* se realiza siguiendo el modelo *vertex-centric*, lo que implica que el programador debe pensar cómo un vértice que tiene valor u otras propiedades y que puede intercambiar mensajes con otros vértices en un número de iteraciones para realizar un calculo.

En la Figura 1.1 se puede apreciar la representación de una pequeña red social.

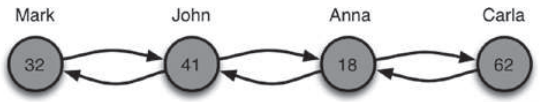


Figura 1.1: *Red Social con Grafo [11]*

En esta las personas están representadas por los vértices del grafo, y las relaciones entre estas se representan con las aristas.

El problema identificado tiene relación con que debido a la naturaleza de programación de *Giraph*, no existen muchas implementaciones de algoritmos esenciales de la teoría de grafos disponibles para ser usados por la comunidad.

1.1.3. Trabajo Relacionado

- Trillion Edges, 2012:

  El año 2015 *Facebook* realizó un trabajo de investigación sobre las opciones de escalabilidad de Giraph para grafos de Trillones[1] de aristas. Para esto implementaron una versión de PageRank en Giraph y la compararon con una implementación para *Hive*. En el Cuadro 1.1 se pueden ver los resultados de la experimentación.

[1]Trillones en escala númerica larga (10^{12})

Tam. Grafo	Hive	Giraph	Speedup
2B+ vértices 400B+ aristas	Total CPU 16.5M secs	Total CPU 0.6M secs	26x
	Duración 600 mins	Duración 19 mins	120x

Cuadro 1.1: *Comparación entre Giraph y Hive [3]*

- Property Graphs, 2015:

  Francisco Moya en su memoria el año 2016 implementó una extensión a Giraph utilizando el API provista para poder soportar en este sistema grafos en los que la propiedad no fuera solo un elemento sino un conjunto de propiedades. Su trabajo fue resumido en un paper [1] en el que se mostraron los resultados experimentales de la extensión realizada, así como también un marco teorico para el trabajo con *Giraph*.

1.1.4. Propuesta de Solución

Para resolver la problemática planteada se propone diseñar una serie de algoritmos esenciales de la teoría de grafos, siguiendo el modelo de programación de *Apache Giraph*. Para esto en una etapa inicial, se realiza una evaluación de qué algoritmos deberían ser diseñados, luego se implementan, y finalmente se evaluan en base a casos de prueba predefinidos.

1.2. Objetivos

Objetivo General

- Diseñar e implementar algoritmos esenciales de grafos siguiendo el modelo de programación de Apache Giraph.

Objetivos Específicos

- Definir un conjunto de algoritmos esenciales en teoría de grafos.
- Diseñar e implementar los algoritmos basándose en el modelo *vertex-centric* definido por Giraph.
- Evaluar exploratívamente el funcionamiento de Giraph utilizando los algoritmos implementados.

1.3. Alcances

- La implementación se realiza sobre Apache Giraph.
- Los algoritmos seleccionados se desarrollan de manera incremental, por lo que el número de algoritmos depende de la complejidiad de los mismos.
- El caso de uso se elige al momento de evaluar los algoritmos.

1.4. Metodología

Objetivo 1: "Definir un conjunto de algoritmos esenciales en teoría de grafos"

- Estudiar la teoría de grafos para tener un mejor entendimiento de qué algoritmos sería útil implementar.
- Crear una lista general que incluya una gran cantidad de algoritmos con los que se podría trabajar.
- Seleccionar un conjunto de algoritmos en base a la revisión literaria antes realizada.

Objetivo 2: "Diseñar e implementar los algoritmos basándose en el modelo *vertex-centric* definido por Giraph"

- Analizar en mayor profundidad los algoritmos seleccionados anteriormente, entendiendo su funcionamiento a cabalidad.
- Diseñar el pseudo código de los algoritmos seleccionados utilizando el paradigma *vertex-centered* de *Apahe Giraph.*
- Implementar los algoritmos anteriormente seleccionados siguiendo el pseudo código definido.

Objetivo 3: "Evaluar la eficiencia de los algoritmos empleándolos en un caso de uso real"

- Diseñar e implementar una plan de pruebas para probar los algoritmos antes implementados.

- Realizar las pruebas utilizando el plan de pruebas anteriormente implementado utilizando algún grafo grande como entrada de datos.
- Analizar y evaluar los resultados de las pruebas y proponer mejoras para futuras implementaciones.

1.5. Plan de trabajo

El plan de trabajo se distribuyó pensando en 2 etapas correspondientes al semestre 2016-2 y 2017-1:

- Semestre 2016-2: Para este semestre se planifica trabajar entre el 05/09/16 hasta el 28/11/16.
- Semestre 2017-1: Para este semestre se planifica trabajar entre el 20/03/17 hasta el 28/06/17.

En la Tabla 1.2 se pueden ver los feriados presentes en el período de trabajo, los cuales tampoco fueron considerados en la calendarización de tareas.

Excepción	**Inicio**	**Termino**
Glorias del Ejercito	19/09/16	19/09/16
Receso Fiests Patrias	20/09/16	24/09/16
Encuentro de dos Mundos	10/10/16	10/10/16
Día de las Iglesias Evangélicas y Protestantes	31/10/16	31/10/16
Día de todos los Santos	01/11/16	01/11/16
Viernes Santo	14/04/17	14/04/16

Cuadro 1.2: *Feriados dentro del período de trabajo*

En total se consideraron 123 días de trabajo, que se distribuyeron como se muestra a continuación.

Etapa 1: Definir un conjunto de algoritmos esenciales en teoría de grafos (05/09/19 - 07/10/16)

- Estudiar la teoría de grafos para tener un mejor entendimiento de que algoritmos sería útil implementar (05/09/16 - 16/09/16).

- Crear una lista general que incluya una gran cantidad de algoritmos con los que se podría trabajar (26/09/16 - 30/09/16).
- Seleccionar un conjunto de algoritmos en base a la revisión literaria antes realizada (03/10/16 - 07/10/16).

Etapa 2: Diseñar e implementar los algoritmos basándose en el modelo *vertex-centric* definido por Giraph (11/10/16 - 12/05/17)

- Analizar en mayor profundidad los algoritmos seleccionados anteriormente, entendiendo su funcionamiento a cabalidad (11/10/16 - 24/10/16).
- Diseñar el pseudo código de los algoritmos seleccionados utilizando el paradigma *vertex-centered* de *Apahe Giraph* (25/10/16/ - 23/11/16).
- Implementar los algoritmos anteriormente seleccionados siguiendo el pseudo codigo definido (24/11/16 - 12/05/17).

Etapa 3: Evaluar exploratívamente el funcionamiento de Giraph utilizando los algoritmos implementados (15/05/17 - 28/06/17)

- Diseñar e implementar una plan de pruebas para probar los algoritmos antes implementados (15/05/17 - 26/05/17).
- Realizar las pruebas utilizando el plan de pruebas anteriormente implementado utilizando un grafo conocido de entrada.
- Analizar y evaluar los resultados de las pruebas y proponer mejoras para futuras implementaciones (19/06/17 - 28/06/17).

2. Marco Teórico

En este capítulo se repasan conceptos fundamentales sobre la temas que más adelante son profundizados y necesarios para entender a cabalidad el desarrollo del trabajo de memoria realizado.

En la Sección 2.1 se repasan aspectos fundamentales sobre teoría de grafos, como las formas de representación de un grafo y las ventajas y desventajas de cada forma de representación.

En la Sección 2.2 se enlistan sistemas de administración de grafos de gran escala, sus características, ventajas y desventajas para las distintas aplicaciones en las que pueden ser utilizados.

En la Sección 2.3 se detallan los sistemas de procesamiento de grafos más relevantes a la fecha, y con los que se pueden realizar apliaciones de extracción de datos de manera más eficiente. Se detallan características relevantes, ventajas y desventajas.

2.1. Grafos

Un grafo es una estructura de datos abstracta altamente utilizada en la computación debido a su flexibilidad y alta utilidad para la representación de problemas [5]. Se caracterizan por representar relaciones entre objetos, que para efectos de un grafo son conocidos como *nodos*, conectados entre sí a través de *aristas* las cuales pueden o no tener una dirección desde o hacia un nodo específico.

2.1.1. Representación de Grafos

Existen varias formas posibles de representar un grafo definido como $G = (V, E)$. Entre las más utilizadas se encuentran la lista de adyacencia y la matriz de adyacencia. Ambas representaciones son igual de aplicables para cualquier problema, y para grafos dirigidos o no dirigidos. La lista de adyacencia generalmente es la más utilizada para representar grafos dispersos, es decir grafos en que el numero de aristas ($|E|$) es mucho menor que el número de vértices al cuadrado ($|V^2|$), debido a que permite una forma compacta de representar. Análogamente, cuando el grafo es muy denso, es decir el número de aristas ($|E|$) es cercano al número de vértices al cuadrado ($|V^2|$) o cuando se necesite averiguar rápidamente si entre dos nodos hay una conexión, la representación de la matriz de adyacencia es más indicada.

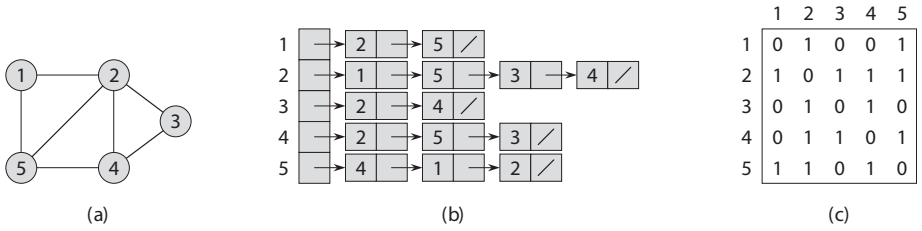


Figura 2.1: *Representación de grafo no dirigido*

En la Figura 2.1 se puede observar cómo se representa un grafo no dirigido **(a)** con las dos representaciones estándares. En el caso de la lista de adyacencia **(b)**, para cada nodo se tiene una lista que contiene los índices de los nodos a los que esta conectando. Así para el nodo número 3 que está conectado con el nodo 2 y 4, se observa que la lista de adyacencia contiene a los nodos 2 y 4, como se esperaba. En el caso de la matriz de adyacencia **(c)** contiene una fila y una columna por cada nodo

del grafo. Por cada posición i, j donde i es el nodo de origen y j el nodo de destino, entre dos nodos que exista conexión se presenta un 1 simbolizando la conexión. De no existir conexión se presenta un 0. Para el nodo 3 se puede apreciar como en su fila las columnas que representan al nodo 2 y 4 tienen un 1 simbolizado la unión entre los nodos. De la misma forma las respectivas filas de los nodos 2 y 4 tendrán un 1 en la columna que representa al nodo 3. En el caso particular del grafo no dirigido la matriz de adyacencia presenta simetría respecto a la diagonal principal.

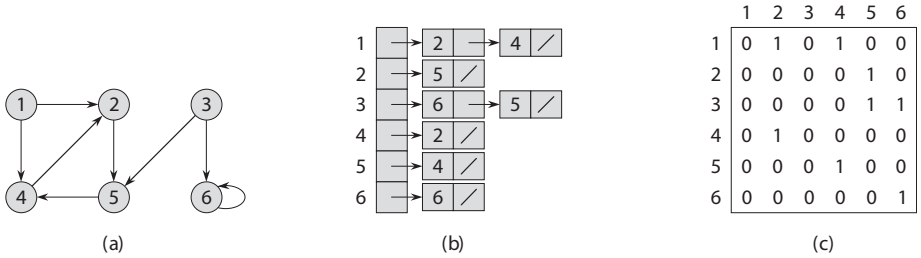


Figura 2.2: *Representación de grafo dirigido.*

En la Figura 2.2 observamos la representación de un grafo dirigido **(a)**. Su lista de adyacencia es similar a la de un grafo no dirigido, pero se diferencia en que no existe la replicación de conexiones entre dos nodos, sólo cuando este explícitamente marcado que dos nodos estén conectado en ambas direcciones. Así, en el caso del nodo 4 existe una conexión hacia el nodo 2 la cual se puede apreciar en su lista de adyacencia, pero desde el nodo 2 hacia el 4 no hay una conexión explícitamente marcada, por lo que en la lista del nodo 2 no hay una entrada para el nodo 4. En el caso de la matriz de adyacencia **(c)**, la representación también es similar a la de un grafo no dirigido, sin embargo no se presenta la simetría de la matriz, a no ser que el grafo presente conexiones que configuren simetría al representarse en la matriz de adyacencia.

Tanto la lista de adyacencia como la matriz de adyacencia pueden ser utilizados para grafos que tienen pesos en las aristas. Por ejemplo si $G = (V, E)$ es un grafo con peso, donde la función de pesos para la arista es w, el peso $w(u, v)$ de la arista $(u, v) \in E$ es simplemente guardado como una entrada en la fila u en la columna v de la matriz de adyacencia. Si una arista no existe, un valor no numérico se guarda en la entrada correspondiente de la matriz, pero esto puede traer problemas, por lo que generalmente se almacena un valor 0 o ∞.

La lista de adyacencia es muy eficiente en términos de espacio, sin embargo la simplicidad de la matriz de adyacencia puede hacerla preferible cuando los grafos son razonablemente pequeños. Además, si el grafo no tiene peso, hay una ventaja adicional de almacenamiento para la matriz de adyacencia, ya que en vez de utilizar una palabra de memoria de computador por cada matriz, la matriz de adyacencia utiliza solo un bit por entrada.

2.2. Administración de Grafos de Datos

Es largamente reconocido que los grafos son una de las estructuras computacionales más poderosas existentes hasta el momento. Sus aplicaciones son tan variadas que han sido utilizados para dar solución a distintos problemas, entre esos la administración de datos.

2.2.1. Modelos de Base de Datos de Grafos

Los modelos de bases de datos de grafos están basados en la definición matemática de un grafo: grafos dirigidos, no dirigidos, con etiquetas o sin etiquetas , con propiedades en los nodos o aristas, hypergrafos, etc.

2.2.2. Procesamiento de Grafos

A diferencia de los modelos de base de datos de grafos que solo se encargan de almacenarlos en una manera en que sean rápidos de acceder, los sistemas de procesamiento de grafos van un paso más allá. El procesamiento de grafos tiene relación con el análisis de las conexiones naturales que existen dentro de ellos, lo que generalmente es computacionalmente difícil debido a la irregularidad por naturaleza de los grafos. Debido a esto y para aprovechar de mejor manera las características de los grafos, es que existen varias herramientas que facilitan el trabajo, todas estas están basadas en modelos distribuidos que maximizan la utilización de recursos y el rendimiento de las aplicaciones. Entre los principales modelos de aplicaciones distribuidas para grafos se encuentran el *Bulk Synchronous Parallel (BSP)* y el *Gather, Apply, Scatter (GAS)*.

- BSP: Este modelo se basa en el procesamiento paralelo e independiente en threads de distintas partes de la computación para luego ser sincronizados y

entregados los resultados principales. En base a esto es que se identifican tres etapas principales:

1. Procesamiento Independiente: Los nodos procesan independientemente la tarea asignada y esperan a que los demás terminen.
2. Comunicación de red : Las componentes envían mensajes comunicando los resultados de su procesamiento.
3. Sincronización: Se procesan los mensajes enviados a través de la red y se actualizan variables compartidas.

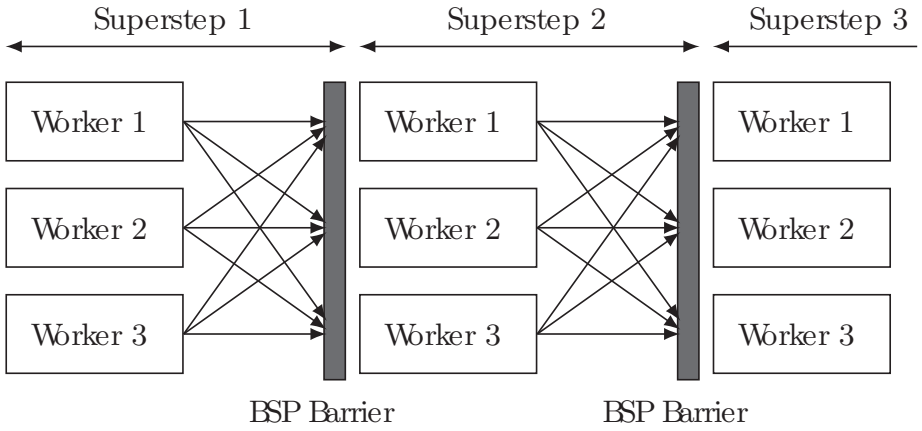


Figura 2.3: *Modelo de Computación BSP*

En la Figura 2.3 se puede apreciar la estructura de ejecución de un proceso BSP. Un *superstep* tiene una etapa de computación y traspaso de mensajes a través de la red. Luego de eso se estable un punto de espera, *Barrier* hasta que todos los *workers* terminan su ejecución y estén listos para enviar los mensajes a los otros workers. Esta secuencia continua hasta que termina la ejecución total del programa.

- GAS: En este modelo los vértices directamente pueden obtener mensajes desde los otros *workers* sin esperar a que ellos los envíen. Esto entre otras cosas permite que las tareas se ejecuten de manera asincrónica reduciendo tiempos en que los workers más rápidos deben esperar sin a hacer nada a que los workers más lentos terminen la ejecución.

2.3. Sistemas de Procesamiento de Grafos

Existe varios frameworks que nos permiten procesar de manera eficiente grafos de gran tamaño. Particularmente nos concentraremos en *Apache Giraph*, el cual gracias a su gran comunidad de usuarios y su gran rendimiento, provee una de las mejores herramientas que existen hasta el momento.

2.3.1. Hadoop

Hadoop es una plataforma de código abierto genérica para el análisis de datos de gran tamaño. Se basa en el modelo de programación de MapReduce. Hadoop ha sido ampliamente utilizado en muchas áreas y aplicaciones tales como análisis de logs, mejoras de motores de búsqueda, intereses de usuario, predicciones, publicidad, etc. Es tan popular esta plataforma, que se ha convertido en la plataforma por defecto para procesamiento de datos batch. El modelo de programación de hadoop puede tener alto o bajo consumo de recursos para algoritmos de grafos iterativos, como consecuencia de su estructura basada en mappers y reducers que procesan a través de un disco HDFS.

2.3.2. YARN

YARN es la siguiente generación de Hadoop. Fácilmente puede correr trabajos antiguos de MapReduce, pero fue diseñada para facilitar la implementación siguiendo múltiples modelos de programación. Una gran mejora introducida por YARN es el separar la administración de recursos funcionales y la administracion de los trabajos a correr; este último es realizado por un administrador basado en una per-application. Por ejemplo en la versión original de Hadoop que utiliza MapReduce ha sido modificado para córrer MapReduce jobs como una aplicación de administración YARN.

2.3.3. Stratosphere

Stratosphere es una plataforma de código abierto para procesamiento de datos de gran escala. Stratosphere se compone de dos componentes claves: Nephele y PACT.

Nephle es un motor de ejecución de dataflows paralelos escalable, en el que los trabajos son representados como grafos dirigidos acíclicos (DAG), un modelo de

trabajos similar al de la plataforma genérica distribuida Dryad. Para cada arista, de tarea en tarea, de Nephele ofrece tres tipos de canales para transportar la información: a través de la memoria, in-memory y a través de archivos.

PACT da soporte a varios códigos de anotaciones de usuarios, los cuales pueden informar al compilador PACT del comportamiento esperado de las funciones de segundo orden, las cuales al ser analizadas por el permiten al compilador PACT producir planes de ejecución que evitan operaciones de alto costo como envío de información, ordenamiento y el traspaso de información al disco.

Los programas compilados por PACT son convertidos en Nephele DAGs y ejecutados por el motor de datos de Nephele. El HDFS de Hadoop es utilizado por Stratosphere como el motor de almacenamiento de datos.

2.3.4. Apache Giraph

Apache Giraph es una herramienta de procesamiento de grafos basada en el trabajo realizado por Google Pregel, pero de código abierto.

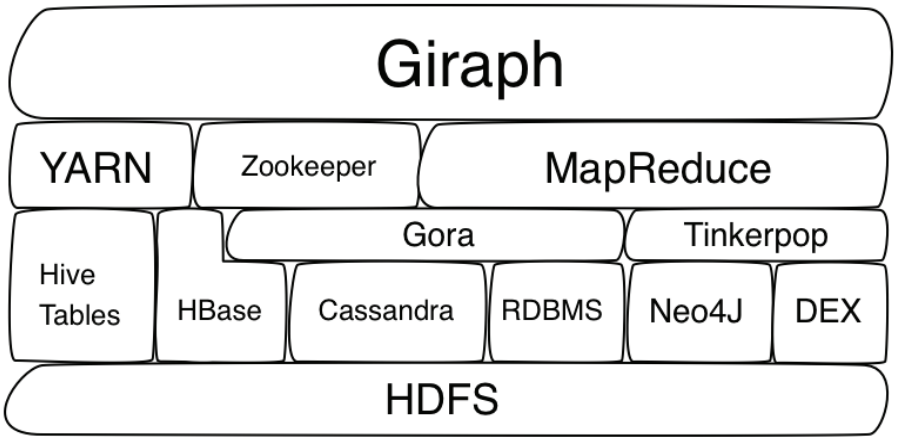


Figura 2.4: *Arquitectura de Apache Giraph*

Como pregel, Apache Giraph corre *workers* como trabajos map-only sobre Hadoop, además de utilizar *Hadoop Distributed File System* para almacenar los archivos de entrada y salida. Adicionalmente utiliza *Apache ZooKeeper* para establecer la comunicación entre los trabajadores. En la Figura 2.4 se puede apreciar todas las subcomponentes por sobre las que corre Giraph. Más adelante en el siguiente capítulo se hace una revisión completa sobre Giraph, ya que esta memoria se basa

completamente en este framework.

2.3.5. GraphLab

Es una herramienta de código abierto, especialmente diseñada para el trabajo con grafos distribuidos en varias máquinas, implementado en C++. Además del trabajo en grafos, también soporta algunos algoritmos de machine learning. Para funcionar, GraphLab almacena el grafo completo y todo el programa a ejecutar en la memoria principal. Las mejoras de funcionamiento vienen dadas por la implementación de varios mecanismos como computación asincrónica para disminuir los tiempos de espera en las barreras de sincronización, utilización de programación de tareas con una cola de prioridad para la rápida convergencia de algoritmos iterativos y estructuras de datos eficientes para la alocación de datos.

2.3.6. Neo4J

Neo4J es una de las base datos de grafos más populares debido a ser de código abierto. Los datos son almacenados en grafos, en vez de tablas. Cada grafo almacenado consiste en relaciones y vértices anotados con propiedades. Neo4j puede ejecutar algoritmos eficientemente incluso en una sola máquina, debido a las optimizaciones que favorecen la respuesta en el tiempo. Neo4j usa dos niveles de memoria principal de cache que mejora su rendimiento: un archivo de buffer guarda datos en el cache de almacenamiento de la misma forma que los datos de larga duración y un objeto de buffer que almacena vértices y relaciones en un formato optimizado para altas velocidades de acceso.

2.3.7. Map Reduce vs Giraph

Encontrar componentes conectadas, *Page Rank* y Caminatas aleatorias, son ejemplos de algoritmos de naturaleza iterativa, en la que se recorre el grafo hasta encontrar la solución final. El modelo de programación tradicional que Map Reduce propone no es una buena opción para este tipo de problemas, especialmente cuando se aplican sobre grafos. Esto se debe a que por cada iteración que Map Reduce realiza, todos los datos que son generados en la ejecución deben ser transmitidos desde los mappers hacia los reducers, lo cual implica una gran cantidad de operaciones de lectura y escritura de disco y un gran uso de la red para transmitir dichos mensajes.

Apache Giraph por el contrario, es ideal para este tipo de algoritmos. En efecto desde la ejecución inicial los datos provenientes de un grafo son cargados y particionados al principio, y luego distribuidos equitativamente entre los trabajadores. Entre cada iteración solo los resultados son transmitidos, con lo que se puede reducir la utilización de operaciones de disco y la utilización de la red, con lo que se obtiene una mejora significativa en los overheads producidos por estos dos ítemes.

3. Apache Giraph

En este capítulo se revisa aspectos generales y específicos acerca de Apache Giraph necesarios para entender la forma que en que se diseñaron los algoritmos utilizando giraph.

En la Sección 3.1 se relatan los orígenes de este framework y sus principales características en cuanto a la ídea detrás de Apache Giraph.

En la Sección 3.2 se muestran resultados experimentales que posicionan a Apache Giraph como uno de los frameworks de procesamiento de grafos de más utilidad en el momento.

En la Sección 3.3 se comenta el modelo de programación que utiliza Apache Giraph, el cual se basa en los vértices del grafo.

En la Sección 3.4 se detalla el modelo de datos que utiliza Apache Giraph y se muestran las principales clases que se utilizan para cualquier implementación en Apache Giraph.

3.1. Orígenes

Muchos proyectos de Apache que nacieron bajo Hadoop han sido fuertemente inspirados por las tecnologías de Google. Hadoop originalmente consistía de un sistema de archivos distribuidos (HDFS) y MapReduce como el framework de trabajo. Estos dos sistemas fueron inspirados por dos herramientas de Google: El Sistema de archivos de Google (GFS) y el MapReduce de Google, los cuales fueron descritos en dos artículos publicados el 2003 y 2004, respectivamente. En el 2010 Google publicó un artículo acerca de un sistema de procesamiento de grafos de gran escala llamado Pregrel. Apache Giraph fue ideado fuertemente inspirado por los preceptos de Pregel.

Giraph fue inicialmente desarrollado en Yahoo! e incubado en la fundación Apache durante el verano de 2011. En el 2012, Giraph fue promovido como un proyecto de alto nivel. Giraph es una implementación abierta de Google Pregel y fue liberado bajo la licencia de Apache. En el 2013 la versión 1.0 fue liberada como prueba de su estabilidad y un gran número de nuevas características fueron agregadas desde el lanzamiento inicial.

Entre los principales contribuidores del proyecto se encuentran grandes compañias tecnológicas como Facebook, Twitter y Linkedin, y actualmente es utilizado en producción en estas mismas compañias y en varias otras más. Giraph comparte con pregrel su modelo de computación y su paradigma de programación, pero extiende el API de Pregel y su arquitectura introduciendo una función maestra de computación (*compute*) y remueve el punto único de falla (SPoF) representado por el maestro.

3.1.1. Contribución de Facebook

En el verano del año 2012 Facebook inició un programa de investigación para generar una aplicación de procesamientos de gran escala como el grafo de Facebook, la cual todavía es utilizada. Para esto se analizaron muchas herramientas existentes de procesamiento de grafos, pero finalmente se decieron a utilizar Apache Giraph por varias razones:

- Giraph interactúa directamente con la versión de HDFS que ellos poseen, ya que Giraph esta escrito en java.
- Como Giraph esta programado como un trabajo de MapReduce, podían reutilizar su estructura MapReduce existente con un pequeño overhead por operación.

- El rendimiento de Giraph es mucho más rápido que el de otros frameworks parecidos.
- El modelo BSP de Giraph es fácil de depurar, pues provee resultados repetibles, y la escalación es directa.

A pesar de que la plataforma por sí misma ya tenía gran valor y potencial para desarrollar aplicaciones de gran nivel, en Facebook habían limitaciones con la plataforma existente, las que fueron directamente trabajadas por facebook:

- El modelo de entradas de Giraph es solo centrado en los nodos, lo que requería que se debía extender el modelo o preparar los datos para el modelo centrado en vértices antes de pasarlos a Giraph.
- La paralelización de la infraestractura de Giraph se basaba completamente en tareas de MapReduce y no soportaba Multithreading para paralelismo más fino.
- La flexibilidad de los tipos de datos y el modelo de computabilidad fueron implementados usando datos nativos de Java, los que consumen memoria excesiva y tiempo de ejecución del garbage collector.
- El aggregator framework estaba implementado de manera ineficiente en Zookeeper y se necesitaba soporte para agregators de gran tamaño.

3.2. ¿Porqué Giraph?

En el Capítulo 2, en las Secciones 2.2 y 2.3 se revisaron múltiples sistemas con los que se pueden procesar y almacenar grafos de gran escala. Cada uno con sus propias funcionalidades y ventajas que los convierten en herramientas útiles. Por esta razón es clave preguntarse porque se eligió giraph para centrar el trabajo de investigación en esta memoria. Razones para argumentar esta decisión hay de sobra. Sin embargo y siguiendo los objetivos finales de esta memoria, nos concentraremos en tres aspectos: Procesamiento de algoritmos, procesamiento de vértices y de aristas. Elementos como el tratamiento de memoria, uso de red, operaciones de disco fueron obviados en este momento debido a que no revisten elementos propios de sistemas distribuidos y no nos sirven como ente comparador ante ejecuciones en sistemas

tradicionales no distribuidos. Para este análisis se considera el paper presentado por Young Guo [9] y otros autores.

3.2.1. Procesamiento de Algoritmos

En el estudio de Guo [9] se compararon los tiempos de ejecución de *Breadth-first search* (BFS) en varias plataformas de procesamiento de grafos, intentando obtener elementos comparativos para decidir cuál plataforma se desempeña de mejor forma ante distintos datasets. Sin embargo los hallazgos realizados por este estudio no fueron suficientes para marcar una clara diferencia entre las plataformas consideradas.

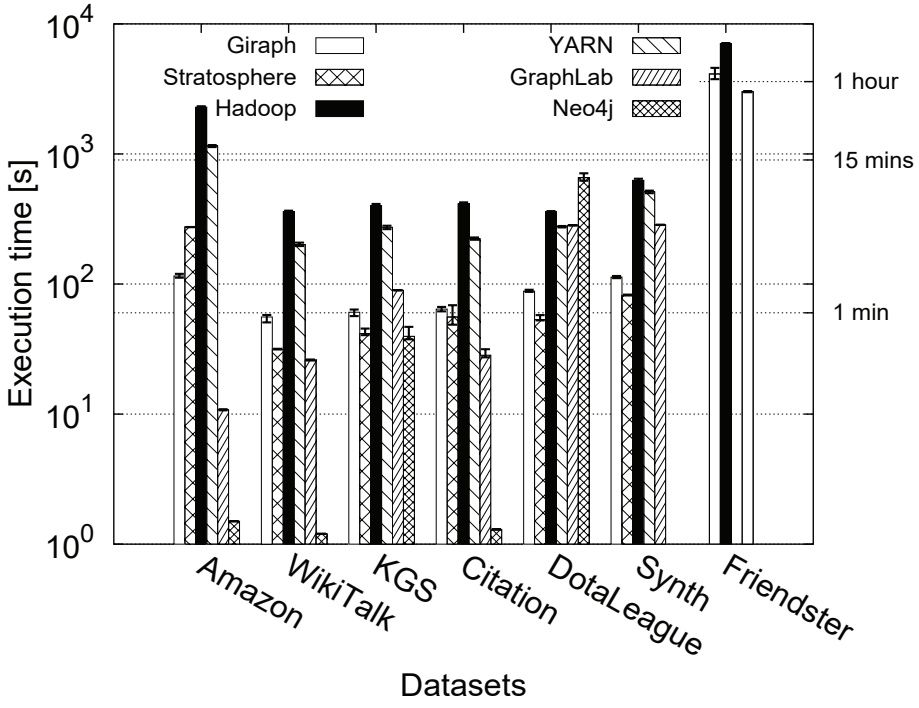


Figura 3.1: *Comparación de tiempos de ejecución de BFS [9]*

En la Figura 3.1 se observan los tiempos de ejecución de BFS en distintas plataformas utilizando distintos datasets. A la vista se aprecia lo que antes se discutía que no existe ningún claro ganador, solo existen algunas ventajas puntuales para algunos datasets por parte de alguna plataforma. Entre las conclusiones de esta prueba que Guo destaca en su paper Guo se aprecia lo siguiente:

- Se observa que Hadoop tiene el peor rendimiento.

- Hadoop y Yarn sufren sobremanera cuando se requieren varias iteraciones en la ejecución de un algoritmo.
- El rendimiento de todas las plataformas es similar con una varianza máxima de 10 %.
- Algunas plataformas no pueden procesar los datasets y terminan la ejecución, por eso presentan tiempos de ejecución mínimos.

3.2.2. Vértices por Segundo (VPS)

Guo encontró en su paper que las distintas plataformas presentaban distintos rendimientos para los disintos datasets. En el caso de Hadoop, sigue siendo la plataforma que peor se desempeñó en esta experimentación.

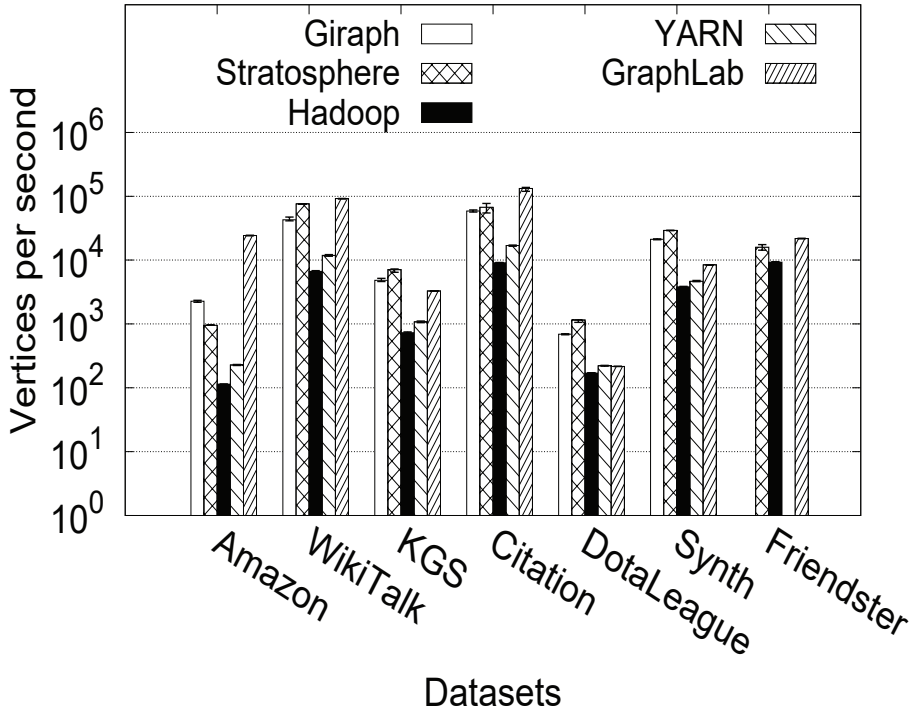


Figura 3.2: *Comparación de tiempos de ejecución de BFS [9].*

Giraph y GraphLab llevan la delantera en esta comparación, pudiendo procesar un mayor número de vértices por segundo, lo cual tiene un impacto directo en la ejecución total de un algoritmo. En la Figura 3.2 se pueden apreciar los resultados obtenidos durante esta experimentación.

3.2.3. Aristas por Segundo (EPS)

De manera equivalente, la revisión del procesamiento de aristas por segundo, aporta un elemento relevante a considerar al momento de evaluar el desempeño de una plataforma. En este sentido Hadoop sigue siendo la plataforma que peor se desempeñó en esta prueba. Por el contrario, se observa que Giraph y Stratosphere

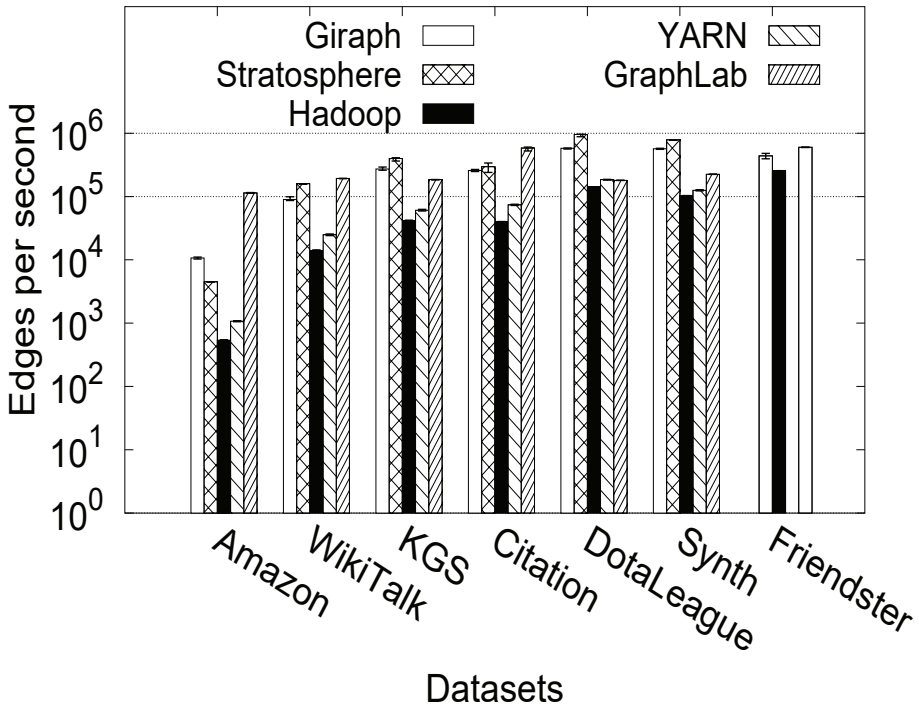


Figura 3.3: *Comparación de tiempos de ejecución de BFS [9]*

presentan los mejores rendimientos en esta prueba. En la Figura 3.3 se observan los resultados experimentales de esta prueba para los distintos datasets con las distintas plataformas.

Es claro ver que Giraph es la plataforma más estable en cuanto a rendimiento, por lo cual fue elegida como la plataforma para trabajar.

3.3. El Modelo de Programación de Giraph

Tradicionalmente los algoritmos de grafos han sido diseñados siguiendo un modelo de programación secuencial al que todo el mundo se ha acostumbrado. El grafo es presentado en la memoria principal con estructuras de datos nativas como matrices o listas. Los algoritmos asumen una vista global del grafo y un *thread* único de ejecución. Tanto las estructuras de datos como la lógica de ejecución están fuertemente basadas en la solución. Esto trae consigo una serie de consecuencias. Primero, cada problema trae consigo un nuevo algoritmo de grafo y una nueva estructura de datos, los que tienen que ser diseñados desde cero. Este tipo de soluciones tan apegadas al problema, son buenas para realizar mejoras, pero fallan en generar una generalización, por lo que cada vez que se tiene un nuevo problema es necesario reinventar nuevamente la rueda. Segundo, para una ejecución distribuida o paralela, el algoritmo tiene que ser directamente diseñado o editado, lo cual no es trivial y muchas veces es bastante difícil.

Giraph enfrenta estas dos problemáticas de manera directa, otorgando un modelo programación que ha sido diseñado pensando específicamente en algoritmos de grafos, lo cual permite esconder la complejidad de implementación de la arquitectura distribuida y paralela. Ambas características minimizan el esfuerzo de la implementación de un algoritmo de grafo que trabaja en gran escala.

3.3.1. Complejidad de Computación Paralela y Distribuida

Giraph ofrece más que solo una librería para ejecutar grafos; ofrece un modelo de programación y una API que permite concentrarse en la semántica de un algoritmo específico que se está diseñando, sin preocuparse en cómo el grafo es almacenado en la memoria, cómo el algoritmo es ejecutado o cómo la ejecución es distribuida a través de las máquinas. Al programador lo único que le queda por generar son las funciones definidas por usuario (UDF), que son ejecutadas iterativamente por cada vértice por el Giraph Runtime a través de las unidades de procesamiento. Por ende, las UDF son agnósticas de cómo los datos son compartidos a través de las unidades de ejecución y cómo el código es ejecutado concurrentemente. En otras palabras nada de coordinación para evitar problemas de sincronización o concurrencia es necesario de parte del programador.

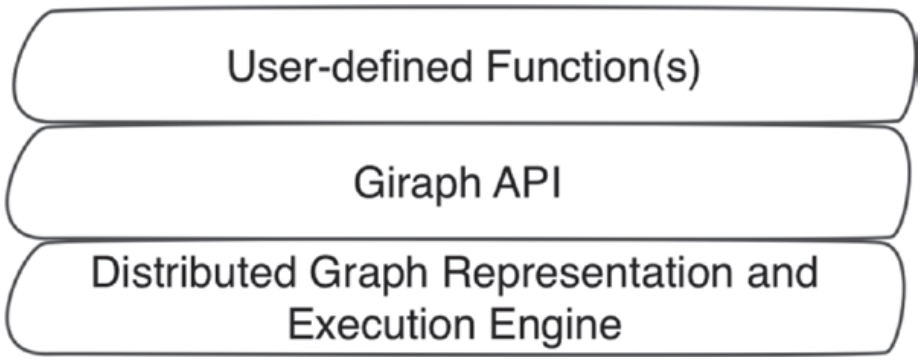


Figura 3.4: *Organización conceptual de Giraph*

Las UDF definen cómo cada vértice maneja los mensajes que recibe y actualiza su valor, y que valores envían como mensaje a los otros vértices. Como los vértices comparten datos a través de mensajes, no es necesario bloquear las variable. Además como cada vértice se ejecuta al menos una vez durante cada iteración, tampoco es necesario sincronizar datos entre los nodos.

3.4. Modelo de Datos de Giraph

De acuerdo a giraph, un grafo es un conjunto de vértices y aristas en los que cada vértice esta definido como sigue:

- Tiene un ID único (integer, string, etc).
- Tiene un valor (double, integer, etc).
- Tiene un número de aristas salientes hacia otros vértices.

Las aristas por su lado se definen de la siguiente forma:

- Tiene un valor (double, integer, etc).

Los nodos solo saben las aristas que salen de él. En base a lo anterior, la clase que implementa a un vértice está definida de la siguiente forma:

```
class Vertex:
    function getId()
    function getValue()
    function setValue(value)
    function getEdges()
    function getNumEdges()
    function getEdgeValue(targetId)
    function setEdgeValue(targetId, value)
    function getAllEdgeValues(targetId)
    function voteToHalt()
    function addEdge(edge)
    function removeEdges(targetId)
```

De la misma forma, la clase que implementa a una arista está definida de la siguiente forma:

```
class Edge:
    function getTargetVertexId()
    function getValue()
    function setValue(value)
```

3.4.1. Modelo Tradicional vs Vertex Centric

Para comparar ambos modelos presentaremos un ejemplo de un algoritmo básico de grafo que calcula el valor máximo de entre los nodos de una red social (ver Figuras 3.5 y 3.6).

Modelo Antiguo

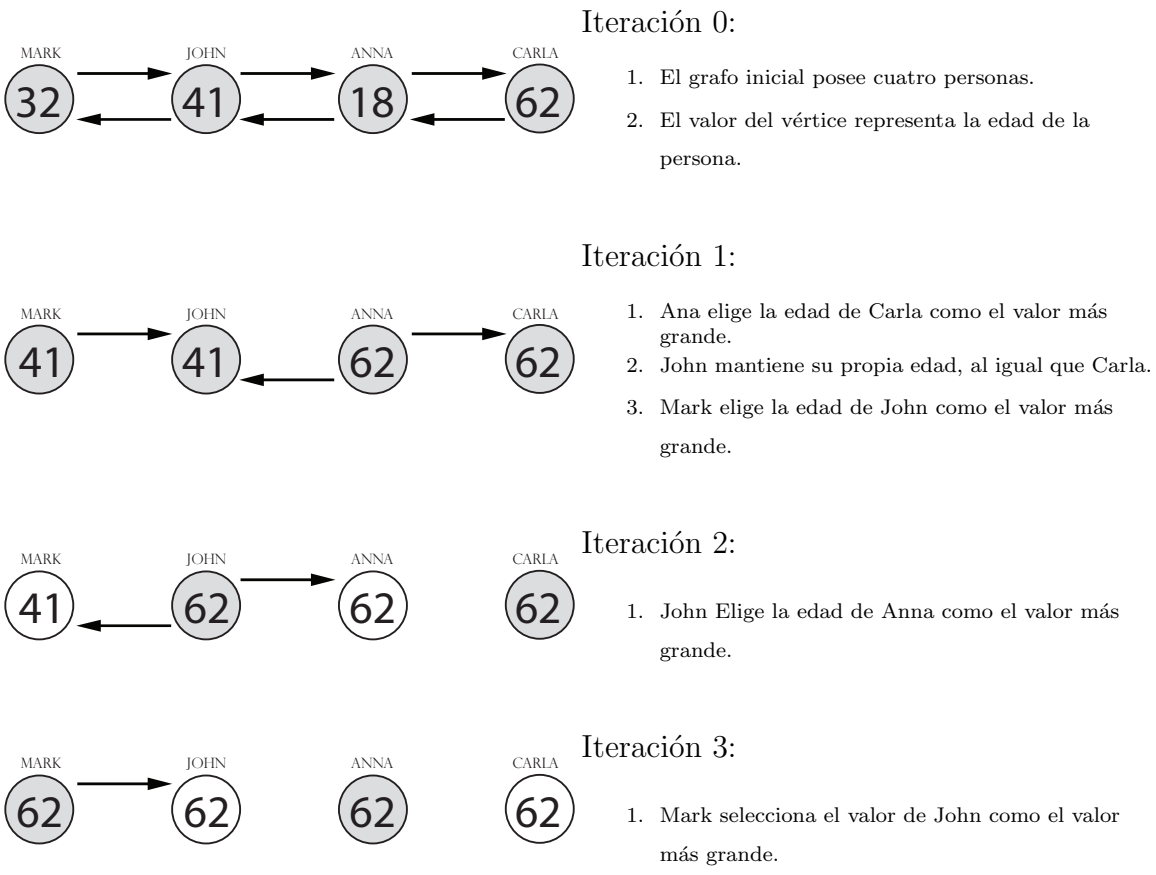
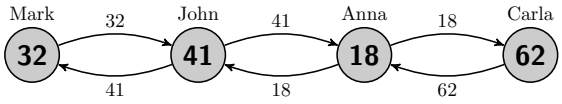


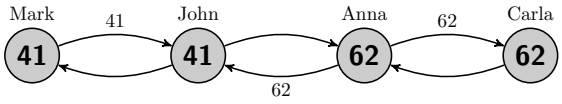
Figura 3.5: *Ejecución maxValue en modelo Tradicional*

Modelo Centrado en Vértices



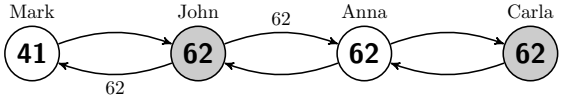
Superstep 0:

1. Todos los vértices comienzan activos con valores inicializados en su edad.
2. Todos los vértices envían su valor a sus vecinos en un mensaje.
3. Todos los vértices se van a dormir.



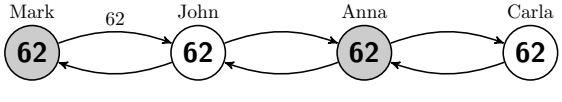
Superstep 1:

1. Los vértices reciben los valores de sus vecinos y actualizan sus valores.
2. Los vértices envían su nuevo valor a sus vecinos.
3. Todos los vértices se van a dormir.



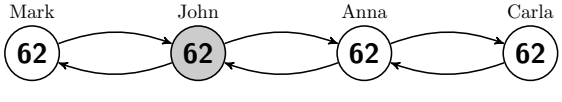
Superstep 2:

1. Los vértices reciben los valores de sus vecinos y actualizan sus valores.
2. Los vértices envían su nuevo valor a sus vecinos.
3. Todos los vértices se van a dormir.



Superstep 3:

1. Los vértices reciben los valores de sus vecinos y actualizan sus valores.
2. Los vértices envían su nuevo valor a sus vecinos.
3. Todos los vértices se van a dormir.



Superstep 4:

1. No hay más actualizaciones necesarias.
2. Todos los vértices se van a dormir.
3. Se termina la computación.

Figura 3.6: *Ejecución maxValue en Modelo Centrado en Vértices*

El pseudo código de la función anterior es el siguiente:

```
function compute(vertex, messages):
    maxValue = max(messages)
    if maxValue > vertex.getValue():
        vertex.setValue(maxValue)
        sendMessageToAllEdges(vertex, maxValue)
    vertex.voteToHalt()
```

Figura 3.7: *Función compute para maxValue*

En la Figura 3.5 se pueden apreciar las iteraciones necesarias para calcular el valor máximo de una red siguiendo el modelo antiguo. Se puede apreciar, cómo es necesario recorrer todos los nodos para encontrar el valor máximo, lo cual hace que la solución no sea tan clara y directa. En la Figura 3.6 se aprecian las iteraciones que Giraph sigue para encontrar el valor máximo de una red. Se puede apreciar cómo son los nodos quienes comunican su valor máximo , y cómo claramente es ejecutado el algoritmo hasta llegar a la solución final. Note que solo los nodos en gris se despiertan al inicio de la iteración (pues recibieron un mensaje en la iteración anterior); en cambio, los nodos en blanco permanecen dormidos. En la Figura 3.7 se observa el código de la función que ejecuta esta algoritmo. Cabe destacar que todas las funciones definidas por el usuario deben estar definida dentro de la función *compute*, la cual es ejecutada iterativamente en todos los nodos durante una iteración o *superstep*.
Al final de la función el nodo debe llamar a la función *voteToHalt* la cual duerme el nodo hasta la siguiente iteración. Si el nodo no recibe más mensajes, no es despertado. La ejecución termina cuando todos los nodos estén desactivados y no hay mas mensajes siendo enviados.

4. Diseño de Algoritmos en Apache Giraph

En este capítulo se introduce la metodología de diseño de algoritmos utilizada durante el curso del desarrollo de esta memoria.

En la Sección 4.1 se muestra un template de estandarización de la traducción de algoritmos del modelo tradicional al centrado en vértices.

En la Sección 4.2 se muestran ejemplos de utilización del template especificado en la sección anterior.

En la Sección 4.3 se detallan algoritmos que ya han sido diseñados e implementados producto de otros trabajos anteriores o como parte de la documentación de prueba de Apache Giraph.

En la Sección 4.4 se detalla el template de estandarización respectivo para algoritmos que fueron diseñados como resultado del trabajo investigativo para esta memoria.

4.1. Template de Estandarización de Algoritmos

Para facilitar la explicación de las soluciones propuestas para un algoritmo tradicional traducido al modelo centrado en vértices, se diseñó un template de estándarización. Este template tiene como finalidad facilitar el entendimiento de la ejecución del grafo. El template definido tiene los siguientes ítemes:

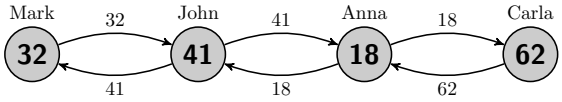
- **Descripción:** Descripción general del algoritmo que se está trabajando. Se deben incluir detalles que permitan al lector entender cuál es el objetivo a conseguir por parte del algoritmo.
- **Entradas:** Entradas que recibe el programa. Se deben especificar para que no quede duda acerca de que datos necesita el algoritmo.
- **Salidas:** Formato de salida del programa, estas se deben especificar según el formato de salida de Giraph.
- **Ejecución:** Se debe especificar el conjunto de iteraciones que se ejecutan para resolver el problema. Para ejemplificar esto es bueno utilizar un pequeño grafo de ejemplo y mostrar las distintas iteraciones necesarias para resolver ese grafo.

4.2. Ejemplos de aplicación del Template

A continuación se presentan ejemplos de aplicación del template para estandarización de explicación de algoritmos utilizando el modelo centrado en nodos.

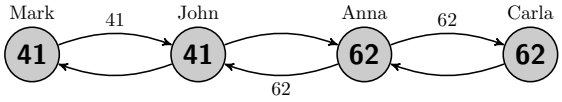
4.2.1. MaxValue

- **Descripción:** Calcula el valor máximo dado un conjunto de nodos que tienen un valor.
- **Entradas:** El Grafo del cual se quiere calcular el valor máximo.
- **Salidas:** El valor máximo presente en el grafo.
- **Ejecución:**



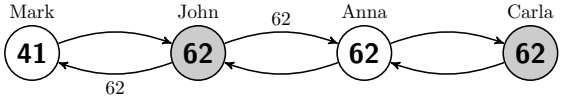
Superstep 0:

1. Todos los vértices comienzan activos con valores inicializados en su edad.
2. Todos los vértices envían su valor a sus vecinos en un mensaje.
3. Todos los vértices se van a dormir.



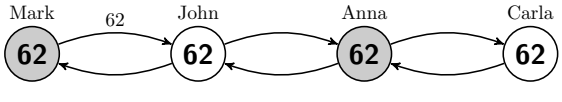
Superstep 1:

1. Los vértices reciben los valores de sus vecinos y actualizan sus valores.
2. Los vértices envían su nuevo valor a sus vecinos.
3. Todos los vértices se van a dormir.



Superstep 2:

1. Los vértices reciben los valores de sus vecinos y actualizan sus valores.
2. Los vértices envían su nuevo valor a sus vecinos.
3. Todos los vértices se van a dormir.



Superstep 3:

1. Los vértices reciben los valores de sus vecinos y actualizan sus valores.
2. Los vértices envían su nuevo valor a sus vecinos.
3. Todos los vértices se van a dormir.

Superstep 4:

1. No hay más actualizaciones necesarias.
2. Todos los vértices se van a dormir.
3. Se termina la computación.

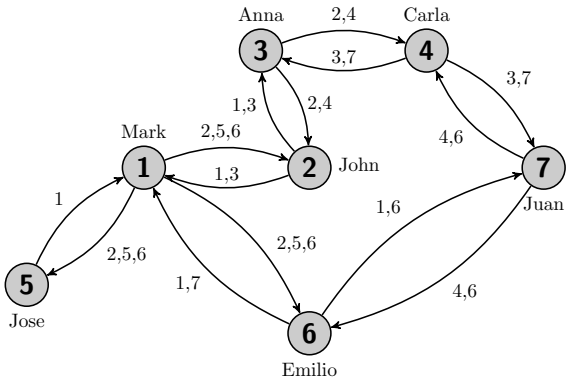
4.2.2. TriangleClosing

- **Descripción:** Cuenta cuantos los triángulos directos a los que un nodo pertenece.
- **Entradas:** Formato de entrada JsonLongDoubleFloatDoubleVertexInputFormat

```
[1,0,[[2,0],[5,0],[6,0]]]
[2,0,[[1,0],[3,0]]]
[3,0,[[2,0],[4,0]]]
[4,0,[[3,0],[7,0]]]]
[5,0,[[1,0]]]]
[6,0,[[1,0],[7,0]]]
```

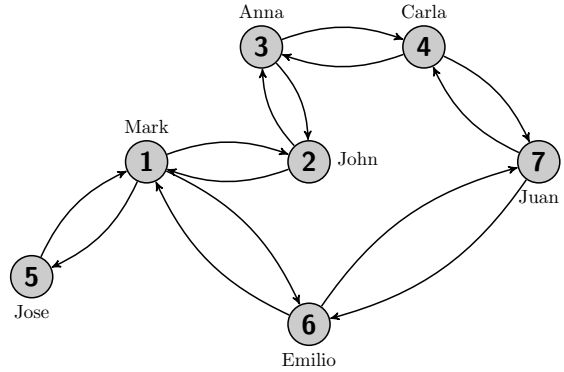
```
[7,0,[[4,0],[6,0]]]
```

- **Salidas:** Lista con los triángulos directos a los que un nodo pertenece.
- **Ejecución:**



Superstep 0:

1. Todos los vértices envían a cada arista un mensaje con los identificadores de los vértices a los que están conectados.
2. Todos los vértices se van a dormir.



Superstep 1:

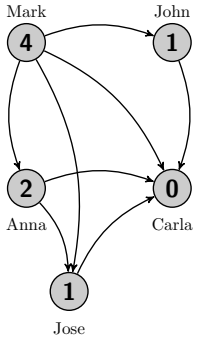
1. Todos los vértices reciben los mensajes de los otros vértices y los ordenan en un diccionario en el que las claves son los nodos de origen de un mensaje y los valores son mensajes recibidos desde un nodo específico.
2. Se comparan los mensajes y si una clave se encuentra como parte de un mensaje de otro nodo, se cuenta un triangulo.
3. Se retornan los triángulos encontrados.

4.2.3. InOutDegree

- **Descripción:** Cuenta el número de conexiones entrantes y salientes de un nodo.
- **Entradas:** formato de entrada JsonLongDoubleFloatDoubleVertexInputFormat

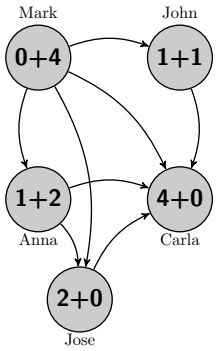
```
[0,0,[[1,1],[3,3]]]
[1,0,[[0,1],[2,2],[3,1]]]
[2,0,[[1,2],[4,4]]]
[3,0,[[0,3],[1,1],[4,4]]]
[4,0,[[3,4],[2,4]]]
```

- **Salidas:** El grafo de entrada en el que el valor de cada nodo es la suma de las conexiones entrantes y salientes
- **Ejecución:**



Superstep 0:

1. Todos los vértices envían a cada arista un mensaje con los identificadores de los vértices a los que están conectados.
2. Todos los vértices se van a dormir.



Superstep 1:

1. Todos los vértices reciben los mensajes de los otros vértices, cuentan la cantidad de mensajes y la suman con la cantidad de aristas salientes. Luego cada nodo se asigna el valor resultante a sí mismo. en una lista.
2. Todos los vértices se van a dormir.
3. Se termina la computación.

4.3. Algoritmos ya Diseñados

Luego de una investigación se da cuenta que los algoritmos ya implementados en apache Giraph son los siguientes:

- Adaptive Partitioning of Large-Scale Dynamic Graphs.
- Business Transaction Graph (BTG) Extraction.
- Label Propagation.
- PageRank.
- Conected Componets.
- Diameter Estimation.
- SimpleShortestPath
- RandomWalk
- SimpleTriangleClosing

Principalmente estos han sido generados como ejemplos de Apache Giraph y algunos trabajos de investigación, pero con poca difusión entre la comunidad científica, por lo que no pueden ser accedidos con gran facilidad.

Para efectos de implementación y de pruebas se trabaja con los algoritmos que en las próximas secciones se detallan.

4.4. Algoritmos a diseñar

Los algoritmos que fueron traducidos al modelo centrado en vértices son todos variaciones del algoritmo de árbol covertor mínimo.

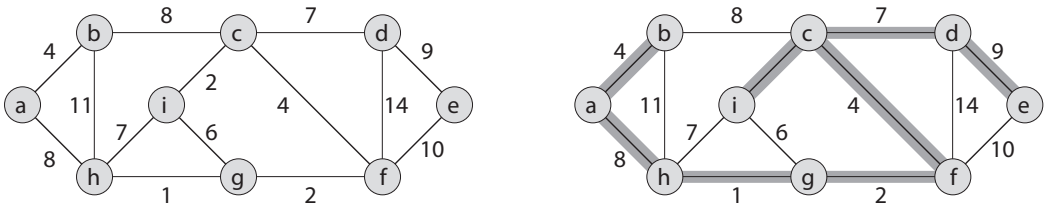
Definición 1 *Árbol cobertor mínimo*
Dado un grafo conectado, no dirigido tal que $G = (V, E)$*, con una función de pesos* $w(u, v)$ *tal que* $w : E \to \mathbb{R}$*,* T *es un MST que cumple con* $T \subseteq E$ *sí y solo sí :*

- *T conecta todos los vértices(V) de G.*
- $w(T) = \sum_{(u,v)\in E} w(u,v)$ *es la menor posible*

4.4.1. Kruskal

Kruskal es un algoritmo de árbol cobertor mínimo (MST) que dado un grafo, puede encontrar un árbol que posee todos los nodos del grafo conectados al menor costo posible.

El algoritmo funciona ordenando los pesos de las aristas de manera ascendente, de manera de agregar primero las aristas con menor peso al árbol resultante, siempre que estas no creen un ciclo en el árbol resultante.



(a) *Grafo no dirigido* **(b)** *MST del grafo a*

Figura 4.1: *Ejemplo de ejecución de Kruskal*

En la Figura 4.1 se muestra un ejemplo de ejecución de este algoritmo y su árbol cobertor mínimo. Como se puede apreciar en la Figura 4.1a, el grafo tiene pesos asignados a sus respectivas aristas que no son dirigidas.

4.4.2. Prim

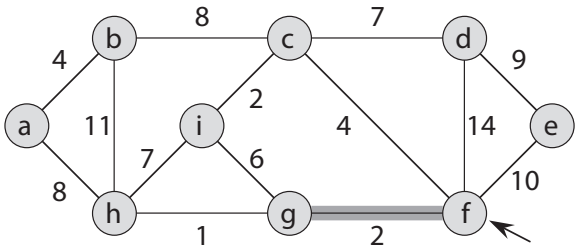
- **Descripción:** Funciona iniciando la ejecución desde un vértice aleatorip, el que se marca como perteneciente al grafo resultante. Luego se evalúa la arista con menor peso y que no forma un bucle, que conecta al grafo resultante con el resto del grafo y se marca como perteneciente al grafo resultante. Se repite el proceso hasta que todos los vértices estén presentes en el grafo resultante.

- **Entradas:** formato de entrada JsonLongDoubleFloatDoubleVertexInputFormat

```
[0,0,[[1,4],[7,8]]]
[1,0,[[0,4],[2,8],[7,11]]]
[2,0,[[2,8],[4,7],[5,4],[7,2]]]
[3,0,[[2,7],[5,14],[4,9]]]
[4,0,[[3,9],[5,10]]]
[5,0 [[2,4],[3,14],[4,10],[6,2]]]
[6,0,[[5,2],[7,1],[8,6]]]
[7,0,[[0,8],[1,11],[6,1],[8,7]]]
[8,0,[[2,2],[6,6],[7,7]]]
```

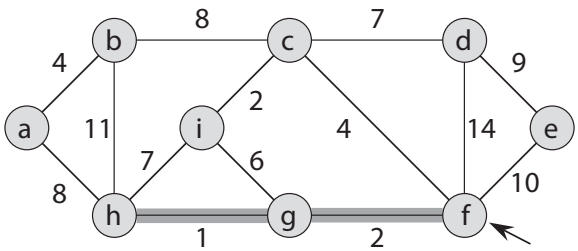
- **Salidas:** Grafo ingresado de parámetro con las aristas pertenecientes al MST marcadas como tal.

- **Ejecución:**



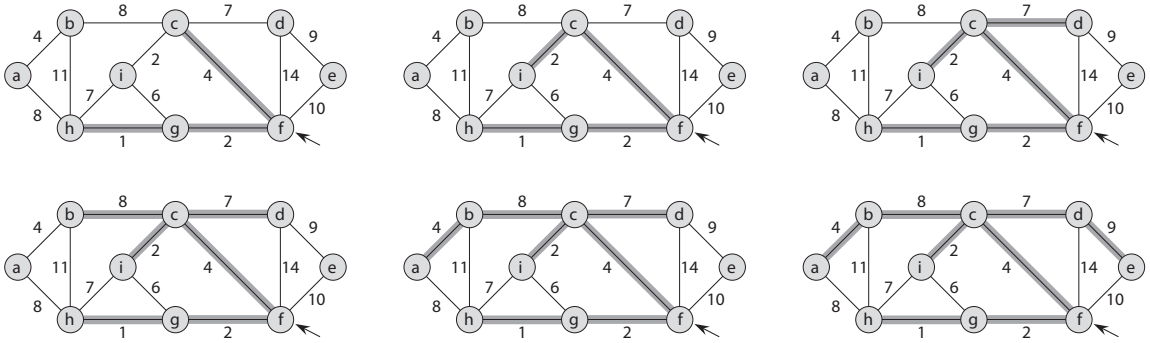
Superstep 0:

1. Se inicia desde un vértice elegido arbitrariamente.
2. Se selecciona la arista con el menor peso y se marca como perteneciente al MST.



Otros Superstep:

1. Se selecciona la arista con menor peso, de las que conectan al grafo resultante con el grafo inicial y se marca como presente.
2. Se repite este paso hasta que no se pueda seguir avanzando por las aristas.



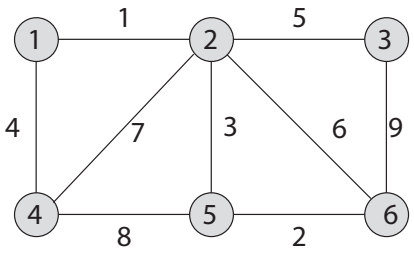
4.4.3. Borůvka modificado

- **Descripción:** Funciona uniendo componentes disjuntas a través de las aristas salientes con menor peso. Al final de cada iteración nodos conectados actúan como una sola componente. En las iteraciones siguientes cada componente se une con una componente con la que no estaba unida siguiendo el mismo principio anterior.
- **Entradas:**

```
[1,0,[[2,1],[4,4]]]
[2,0,[[2,1],[3,5],[4,7],[5,3]]]
[3,0,[[2,5],[6,9]]]
[4,0,[[1,4],[2,7],[5,8]]]
[5,0,[[2,3],[4,8],[6,2]]]
[6,0,[[2,6],[3,9],[5,2]]]
```

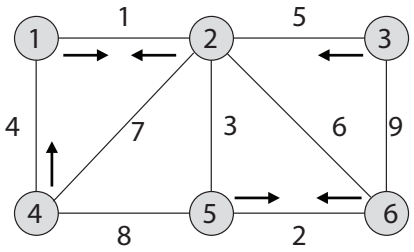
- **Salidas:** Grafo ingresado de parámetro con las aristas pertenecientes al MST marcadas como tal.

- **Ejecución:**



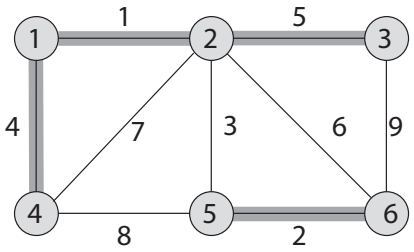
Grafo Inicial:

1. Al comienzo todos los vértices son su propio representante y funcionan como una componente disjunta.



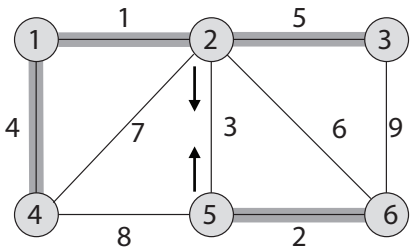
Superstep 0:

1. Todos los vértices seleccionan su arista con menor peso.
2. Los nodos comunican los miembros de su componente a través de la arista seleccionada.



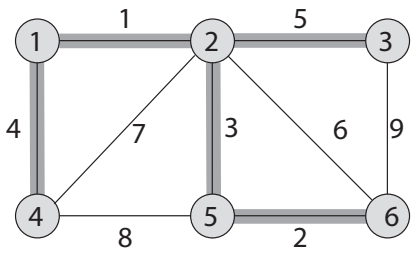
Superstep 1:

1. Las componentes antes individuales ahora actúan en conjunto como una sola.
2. La componente selecciona a su líder basándose en el nodo que recibió más votaciones durante la iteración anterior. Luego el nodo líder es quien seleccione la arista siguiente con la que realiza las nuevas conexiones.



Superstep 2:

1. El nodo líder selecciona la arista saliente de la componente con menor peso, y envía al líder de la otra componente la lista de miembros de la componente.



Superstep 3:

1. Ambas componentes ahora actúan como una sola.
2. Se selecciona al nodo líder entre los dos nodos líderes de las componentes previas.
3. El líder se da cuenta que no existen conexiones salientes de la componente, por lo que se termina la ejecución.

Por limitaciones de tiempo para la realización, de los algoritmos diseñados solo se implementa una versión modificada del algoritmo de Borůvka. En el Capítulo 5, Sección 5.2 se detalla la implementación realizada.

5. Implementación de Algoritmos en Apache Giraph

En este capítulo se detallan todas las decisiones de diseño tomadas para implementar el algoritmo de Borůvka, así como aspectos generales sobre Giraph necesarios para desarrollar código para esta plataforma.

En la Sección 5.1 se comentan algunos aspectos generales que fueron considerados para implementar algoritmos y que ayudan a introducir la programación de Apache Giraph

En la Sección 5.2 se detallan los pasos tomados para realizar la implementación del algoritmo de Borůvka. Se explica el modelo general de implementación, los estados por los que se considera el algoritmo debe pasar para dar cumplimiento a la funcionalidad de Borůvka y las clases implementadas en función de esto.

5.1. Aspectos Generales

Antes de presentar los aspectos más específicos de la implementación de los algoritmos, es necesario comentar algunos aspectos generales para introducir el trabajo de código con Apache Giraph.

- **Lenguaje**: Este algoritmo fue implementado utilizando java como el lenguaje de programación. Esto debido a que las librerías de Apache Giraph están también implementadas en java. Específicamente, la versión del jdk 1.7 fue utilizada.
- **Sistema Operativo**: El sistema operativo en el que se desarrollaron tanto las implementaciones de código como las pruebas es *Debian* en su versión 6.0.
- **IDE**: El entorno de desarrollo integrado utilizado para generar la implementación del algoritmo fue *Eclipse Luna*.
- **Maven** : Maven fue utilizado como gestor de dependencias para la implementación realizada.
- **Git**: El sistema de control de versiones utilizado fue Git.

5.2. Borůvka Modificado

La implementación del algoritmo se realiza siguiendo el modelo centrado en vértices de Giraph. Además se utiliza el API de PGraph diseñado por Francisco Moya [1].
El algoritmo en su ejecución pasa por distintos estados como se puede apreciar en la Figura 5.1. Al inicio de la ejecución cada nodo es su propio líder y su componente esta formada por sí mismo. Luego el nodo debe seleccionar su arista saliente de su componenete de menor peso y a través de esta propagar sus componentes. Luego de entre la componente se elige el nodo con más miembros en su componente, el cual para las próximas iteraciones será el que contacte a los otros nodos, seleccione arista de menor peso y propague componentes. La ejecución termina cuando el líder se da cuenta que no existen aristas salientes de la componente.

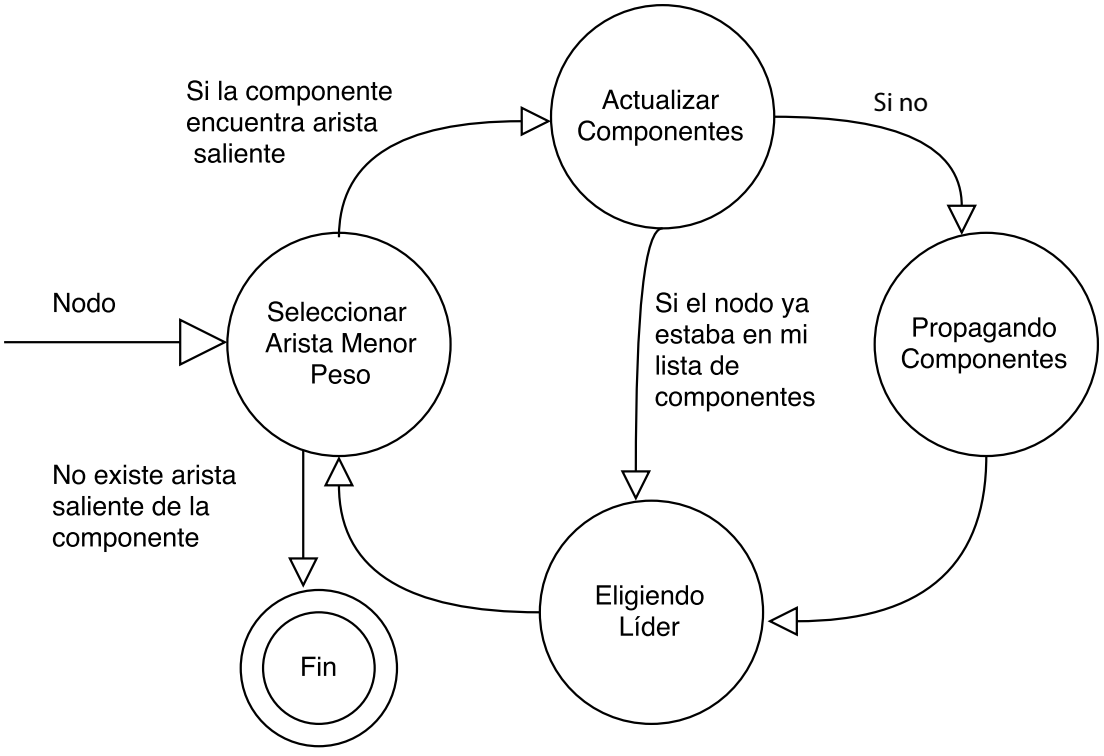


Figura 5.1: *Diagrama de Estado de Borůvka*

Para implementar este algoritmo se implementaron varias clases con las cuales se modeló el comoportamiento de Borůvka.

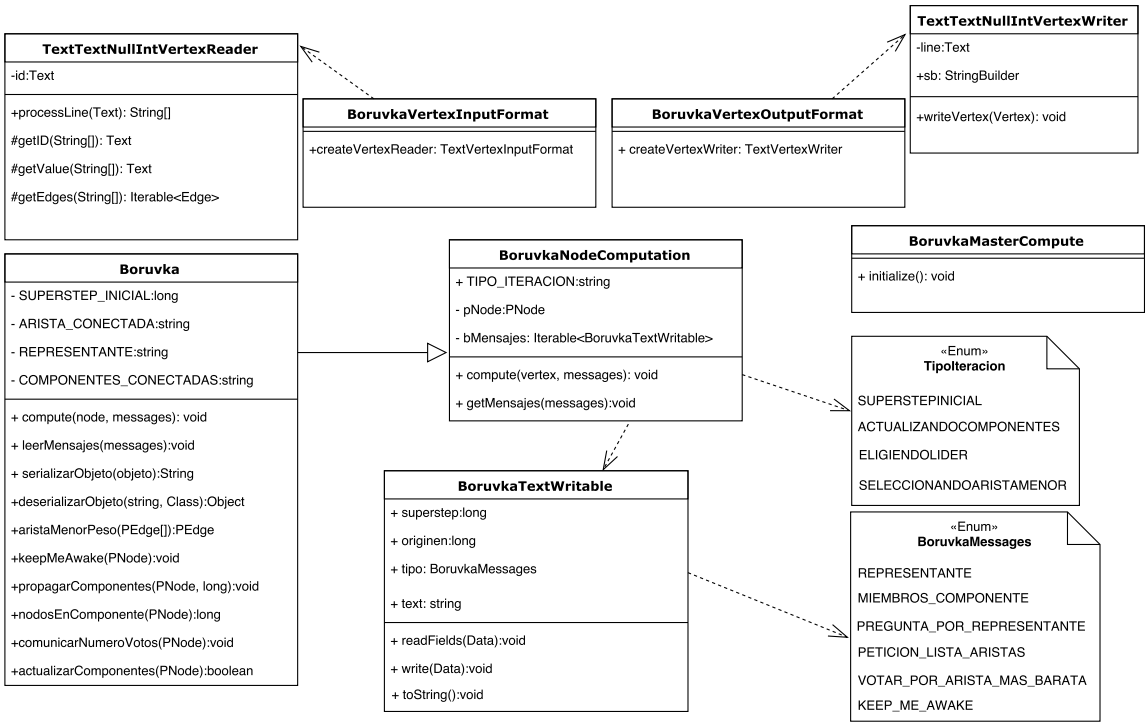


Figura 5.2: *Diagrama de Clases Borůvka*

En la Figura 5.2 se puede observar el diagrama de clases con las funciones y atributos de las distintas clases implementadas. También se pueden observar las relaciones de herencia y relación simple presente entre las distintas clases. De igual forma se detallan los Enums utilizados para modelar acciones en el algoritmo.

A continuación se detallan las clases más importantes implementadas para producir las funcionalidades del algoritmo.

5.2.1. Clase Borůvka

Esta es la clase principal ya que implementa la función compute al heredar de la clase `BoruvkaNodeComputation`(ver Sección 5.2.3). En esta clase se implementa gran parte de las funcionalidades necesarias para implementar el algoritmo. Además, es la clase que define el comportamiento de cada iteración para el nodo. Por ende, es de suma importancia una correcta implementación y funcionamiento de esta clase. Esta clase es llamada por Giraph durante cada iteración, por lo que no se crean instancias de esta clase en ninguna parte. A continuación se detallan las funciones de esta clase.

- **compute**: Esta función es un override de la función **compute** de la clase `NodeComputation`. En esta función se define el comportamiento del nodo en cada iteración. Recibe como parámetro el nodo en el cual se ejecuta en esta iteración y la lista de mensajes que recibe el nodo para esta iteración. Para el caso específico del algoritmo de Borůvka se hace una distinción para el primer superstep. En el caso de estar en el superstep inicial, el vértice debe seleccionar su arista de menor peso y guardarla como su arista seleccionada. Además de esto, debe compartir al nodo de destino de la arista seleccionada, una lista de componentes conectadas, en la que se incluye solo el nodo que envía la lista. En el caso de no estar en el superstep inicial, se debe dar un tratamiento distinto dependiendo del tipo de mensaje que se reciba, para lo que se llama a la función `leermensajes`(ver Sección 5.2.1). Finalmente la función `votetoHalt` es llamada con lo que finaliza la iteración.

- **leermensajes**: Esta función es la encargada de llamar a las distintas funciones dependiendo del mensaje recibido. Recibe una lista de mensajes `BoruvkaTextWritable` (explicada en Sección 5.2.2) la cual itera y dependiendo del tipo de mensaje llama a una función específica. Los tipos de mensajes son del enum `BoruvkaMessages` (5.2.5). Si solo se recibió un mensaje y este es del tipo `KEEP_ME_AWAKE` y la iteración es del tipo `ELIGIENDOLIDER` se puede asumir que un nodo nos está preguntando si este nodo puede ser lider de una componente. Para esto se le envía un mensaje con la cantidad de votos recibidos, que es el mismo que la cantidad de miembros de esta componente. En caso contrario se itera por la lista de mensajes y se realizan llamadas a funciones específicas dependiendo del mensaje. A continuación se detallan dichas llamadas para cada tipo de mensaje.

- `REPRESENTANTE`: Si se recibe este tipo de mensaje, indica que otro nodo quiere saber si puede ser líder. Se compara el número de votos que se recibe en el mensaje y si es mayor que el número de componentes del nodo actual, se actualizó al otro nodo como líder. En caso contrario este nodo sigue como líder.
- `MIEMBROS_COMPONENTE`: Este tipo de mensaje indica que otro nodo envía sus componentes. Se recorren las componentes recibidas, y en caso de no tener alguna de esas, es propagada a través de la arista seleccionada en el nodo actual.
- `PETICION_LISTA_ARISTAS`: Indica que un nodo está pidiendo la lista de aristas local. A lo que se responde con un mensaje de `MIEMBROS_COMPONENTE` y la lista de componentes locales.

Una vez terminado de recorrer la lista de mensajes, dependiendo del tipo de iteración se realizan distintas operaciones. Los tipos de iteraciones están definidos por el enum `TipoIteracion`(descrita en la Sección 5.2.6). A continuación se detallan las acciones a realizar según el tipo de mensaje.

- `ACTUALIZANDOCOMPONENTES` : Si se actualizan componentes, es probable que en la próxima iteración se tenga que seleccionar líder. Por ende se necesita tener este nodo despierto para la siguiente iteración, por lo que le enviamos un mensaje para que se active en la siguiente iteración. Para esto llamamos a la función `keepmeawake`(descrita en la Sección 5.2.1)
- `ELIGIENDOLIDER` : En este tipo de iteración se está eligiendo al líder, por lo que a través de la función `comunicarNumerodeVotos`(descrita en la Sección 5.2.1) comunicamos nuestros votos.

- **serializarObject**: Serializa un objeto para ser distribuido a través de la red. Recibe un objeto y retorna un string con la versión serializada del objeto. Esta función siempre se llama antes de enviar un mensaje por la red. Todos los mensajes que incluyen objetos envían versiones serializadas de estos.
- **decerializeObject**: Deserializa un objeto que fue recibido por la red. Recibe un string con la forma serializada de un objeto y la clase a la que este string se

convertirá. Retorna un objeto con la clase especificada como parámetro. Esta función es llamada siempre que se recibe un mensaje con un objeto.

- **aristamenorpeso**: Selecciona la arista de menor peso de entre las aristas del nodo. Recibe la lista de aristas del nodo y retorna la arista que tiene el menor valor asociado. Esta función es utilizada al momento de seleccionar la arista de menor peso en el superstep inicial.
- **comunicarNumerodeVotos**: Comunica el número de votos que este nodo ha recibido, vía la arista seleccionada como de menor peso. Recibe el nodo que comunicará su número de votos. Cuenta el número de componentes asociadas y envía un mensaje de `REPRESENTANTE`.
- **keepmeawake**: Envía mensaje al mismo nodo para mantener despierto. Recibe un nodo el cual se enviará un mensaje a sí mismo.
- **propagarComponentes**: Reparte a través de la red un mensaje con las aristas conectadas a este nodo a un nodo específico. Recibe como parámetro el nodo del cual se propagarán sus componentes y el ID del nodo de destino.
- **nodosEnComponente**: Retorna el número de nodos en la componente. Recibe el nodo del cual se quiere saber el número de nodos en su componente. Retorna el número de componentes en el nodo.
- **propagarComponentes**: Reparte a través de la red mensajes con las aristas conectadas a este nodo vía la arista seleccionada. Recibe un nodo el cual propagará sus componentes.
- **nodoEnComponentes**: Comprueba si un nodo pertenece a una lista de componentes. Recibe una lista de Nodos y un nodo. Retorna verdadero si el nodo pertenece a la componente, falso en caso contrario.
- **actualizarComponentes**: Revisa la lista de mensajes y verifica si es necesario actualizar la lista de componentes. Recibe un nodo al cual se le actualizaran sus componentes y un mensaje que contiene componentes. Se recorren los nodos que venían en el mensaje y si no están dentro de la componente del nodo se actualiza la lista de componentes del nodo y se retorna verdadero, en caso contrario se retorna falso.

5.2.2. Clase BoruvkaTextWritable

Esta clase extiende a la clase `Writable` y se utiliza para transmitir mensajes a través de la red. Posee los siguientes atributos:

- superstep : atributo de tipo `long` que indica el superstep en que este mensaje fue enviado.
- origen: atributo del tipo `long` que indica el ID del nodo que envió este mensaje.
- tipo: atributo del tipo `BoruvkaMessages`(explicado en Sección 5.2.5) que señala el tipo de estes mensaje.
- text: atributo del tipo `string` que contiene el mensaje enviado.

Al ser una clase que utilitaria que extiende de otra, la mayoría de las funciones implementadas son los métodos delegados de la clase padre que deben ser sobrescritos como las funciones `toString`, `write`, y `readFields`.

5.2.3. Clase BoruvkaNodeComputation

Clase que implementa el modelo de Computación para un PNodo. Específica que la función compute utiliza un PNodo y recibe mensajes del tipo `BoruvkaTextWritable`(explicado en Sección 5.2.2). Extiende la clase `BasicComputation` y por ende realiza una sobrescritura a la función `compute` para permitir utilizar los tipos de datos de la API de PGraph. La única función implementada en esta clase es la siguiente:

- **getMensajes** Obtiene un Iterador de Mensajes Tipo Boruvka dado una iterable de mensajes Text. Recibe un iterador del tipo `Iterable<Text>` y retorna un iterable del tipo `Iterable<BoruvkaTextWritable>`. Recorre la lista de mensajes y uno a uno va agregando los mensajes a una lista del tipo de salida.

5.2.4. Clase BoruvkaMasterCompute

Esta es la clase maestra de Computación. Extiende a la clase `DefaultMasterCompute` y como tal debe realizar una sobrescritura a la función `initialize` en donde podemos registrar el aggregator de tipo de iteración para más tarde poder discriminar en qué tipo de iteración nos encontramos.

5.2.5. Enum BoruvkaMessages

Enum implementado para representar los distintos tipos de mensaje. Las opciones posibles son las siguientes:

- `REPRESENTANTE`: Mensaje que indica que se debe revisar si hay que cambiar el representante actual por el número de votos recibido por el mensaje.
- `MIEMBROS_COMPONENTE`: Mensaje que indica que se está recibiendo componentes como mensaje y se debe actualizar la lista de componentes.
- `PREGUNTA_POR_REPRESENTANTE`: Mensaje que indica que me preguntan por el representante del nodo actual.
- `PETICION_LISTA_ARISTAS`: Pide la lista de aristas a un nodo específico.
- `RESPONDER_A_REPRESENTANTE`: Responde a la pregunta de quién es el representante de un nodo.
- `VOTAR_POR_ARISTA_MAS_BARATA`: Mensaje para votar por una arista de menor peso dentro de una componente de nodos.
- `KEEP_ME_AWAKE`: Identificador para mensaje de mantener despierto.

5.2.6. Enum TipoIteracion

Enum utilizado para diferenciar los distintos tipos de iteración. Las opciones válidas son las siguientes:

- `SUPERSTEPINICIAL`: Indica el superstep 0.
- `ACTUALIZANDOCOMPONENTES`: Indica que se están actualizando componentes.
- `ELIGIENDOLIDER`: Indica que se está eligiendo al líder de la componente.
- `SELECCIONANDOARISTAMENOR`: Indica que se está seleccionando la arista de menor peso.

5.2.7. Class BoruvkaVertexInputFormat

Esta clase detalla el formato de entrada utilizado para pasar el grafo de parámetro a la ejecución de giraph. Esta clase sólo tiene un método.

- **createVertexReader**: este método crea una instancia de la clase `TextTextNullIntVertexRe` la cual lee las entradas desde el archivo y crea los nodos y vértices necesarios.

5.2.8. Class BoruvkaVertexOutputFormat

Esta clase especifica el formato de la salida de los resultados al término de la ejecución del programa. Esta clase también tiene sólo un método.

- **writeVertex**: Este método recibe como entrada un vértice y crea una instancia de la clase `TextTextNullIntVertexWriter`, la cual se encarga de gestionar la salida del programa.

5.2.9. Class TextTextNullIntVertexReader

Esta clase se encarga de leer las entradas desde el archivo de texto ingresado como parámetro de ejecución y transformarlas a vértices. Los métodos de la clase son los siguientes:

- **processLine**: Este método procesa cada línea del archivo de texto y retorna un Array de Strings que posee los tokens de cada línea.
- **getID**: Retorna el ID del nodo
- **getValue**: Retorna el valor del nodo. Para nuestro caso particular, el ID corresponde al valor del nodo.
- **getEdges**: Retorna las aristas del nodo.

5.2.10. Class TextTextNullIntVertexWriter

Esta clase se encarga de escribir por cada vértice los resultados esperados para la salida del programa.

- **writeVertex**: Este método escribe por cada nodo el id del nodo, la arista seleccionada y el destino de dicha arista.

6. Pruebas

En esta capitulo se definen las pruebas que se quieren realizar con las que se verifican varias métricas de los algoritmos implementados.

En la Sección 6.1 se detallan los elementos de los cuáles se toma en cuenta al momento de realizar las pruebas experimentales.

En la Sección 6.2 se comentan cómo se miden los elementos detallados en la Sección 6.1.

En la Sección 6.3 se muestra cómo se realizaron las pruebas, la preparación previa que se hizo del ambiente de pruebas y las entradas utilizadas para realizar las pruebas.

En la Sección 6.4 se muestran los resultados obtenidos en las pruebas experimentales con el algoritmo implementado.

6.1. Elementos a probar

Para verificar el rendimiento de los algoritmos implementados se realiza pruebas a fin de medir las siguientes métricas:

- Tiempo de Ejecución: A fin de tener una idea general de cómo se desempeña el algoritmo se mide el tiempo general de la ejecución. Esto a fin de poder identificar anormalidades entre ejecución y ejecución.
- Uso de CPU: Para verificar la carga que los algoritmos producen sobre el computador se mide el uso de la CPU a nivel de cada máquina y luego a nivel general. Esto nos permite identificar situaciones bordes y casos de prueba que sacan al algoritmo de la ejecución normal.
- Uso de Memoria: Se mide el uso de la memoria a fin de saber que tantos recursos requiere el algoritmo y si la herramienta está asignando correctamente la carga entre los distintos trabajadores de la red.
- Uso de Red: Se mide el uso de la red para saber si el paso de mensajes está generando que la ejecución general del algoritmo se vea afectada ya sea por una mala calidad de red o por el envío masivo de mensajes.
- Operaciones de disco: Se mide cuantas operaciones de escritura de disco se realizan a fin de saber si el disco está generando una ralentización de la ejecución general.
- Análisis asintótico: Se calculan las Cotas Inferiores y Superiores del Algoritmo, así como también el orden de complejidad de este, a fin de saber si la implementación en el modelo centrado en vértices posee un orden mayor que el algoritmo original.

6.2. Metodología de pruebas

Las pruebas se realizan siguiendo una metodología de desarrollo ágil, es decir, una vez que el algoritmo esté implementado se procede a evaluar su rendimiento aplicando las pruebas que miden las métricas antes definidas.

Para la realización de las pruebas se utilizan distintas herramientas tanto propias de Apache Giraph como externas y/o propias de un sistema distribuido linux, tales como:

- Analizadores de tráfico de red como Wireshark, netstat, tcpdump,etc.
- Herramientas para monitorear el uso de CPU como top, vmstat, etc.
- Herramientas para medir el tiempo de ejecución de un algoritmo como time.
- Consultas a profesores especialistas en el área de análisis de algoritmos.
- No se descarta la inclusión de otras.

6.3. Toma de Pruebas

6.3.1. Entradas

Para realizar las pruebas se utiliza un pequeño grafo de prueba del cual ya se conocen los resultados. De esta forma nos es fácil identificar el estado de la ejecución y encontrar errores. Para esto utilizaremos un grafo encontrado como ejemplo al momento de encontrarnos con el algoritmo de Borůvka y que a lo largo del trabajo de investigación significó una gran ayuda para entender el funcionamiento de Borůvka .

En la figura 6.1 se puede apreciar el grafo que se utiliza para las pruebas. Es un grafo no dirigido, en el cual el valor del vértice es también el identificador del mismo. El valor de las aristas representa el peso de esta.

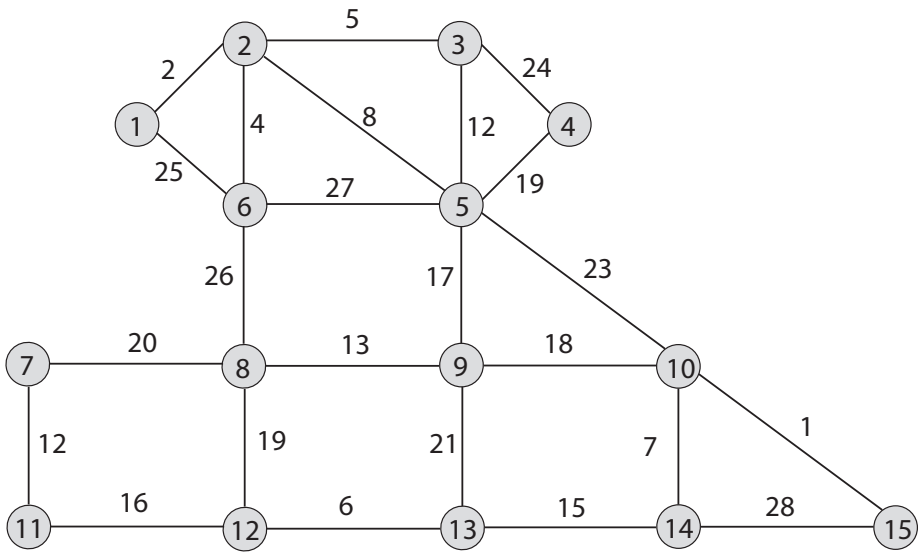


Figura 6.1: *Grafo de prueba.*

Para esta implementación de Borůvka se utilizó un formato de entrada de grafo propio, definido en la clase `BoruvkaVertexInputFormat`, el cual establece que por cada línea los siguientes elementos están presentes:

```
[id_vertice, value_vertice, [ [vertice1, peso1], [vertice2, peso2].. ] ]
```

Donde por cada línea se describe al vértice con sus aristas conectantes hacia otro nodo. Así cada elemento representa lo siguiente:

- `id_vertice`: Id del vértice del cual se almacenan sus datos.
- `value_vertice`: Valor del vértice.
- `[vertice, peso]`: Representan las aristas con las que el vértice se conecta con otros vértices.

Así, siguiendo las reglas descritas anteriormente el grafo de la Figura 6.1 se transforma en la siguiente entrada para giraph:

```
[1,0,  [[2,2]  ,[6,25]] ]
[2,0,  [[1,2]  ,[3,5]   , [6,4],[5,8]] ]
[3,0,  [[2,5]  ,[4,24], [5, 12]] ]
[4,0,  [[3,24] ,[5,19]] ]
[5,0,  [[2,8]  ,[3,12]  ,[4,19]  ,[6,27] ,[9,17] ,[10,23]] ]
[6,0,  [[1,25] ,[2,4]   ,[5,27]  ,[8,26]] ]
[7,0,  [[8,20] ,[11,12]] ]
[8,0,  [[6,26] ,[7,20]  ,[9,13]  ,[12,19]] ]
[9,0,  [[5,17] ,[8,13]  ,[10,18] ,[13,21]] ]
[10,0, [[5,23] ,[9,18]  ,[14,7]  ,[15,1]] ]
[11,0, [[7,12] ,[12,16]] ]
[12,0, [[8,19] ,[11,16] ,[13,6]] ]
[13,0, [[9,21] ,[12,6]  ,[14,15]] ]
[14,0, [[10,7] ,[13,15] ,[15,28]] ]
[15,0, [[10,1] ,[14,28]] ]
```

6.3.2. Configuración de ambiente

Para realizar las pruebas se utilizó una máquina virtual con `Debian GNU/Linux 8.5`, `Hadoop 1.2.1` y `Giraph 1.2.0`. Todas estas herramientas fueron intaladas y configuradas siguiendo la Guía de instalación de Giraph en una máquina en linux [8].

Para facilitar la ejecución de código propio en Giraph, se puede guardar en una variable de usuario la ubicación del jar que contiene el código a ejecutar. Para esto y según la recomendación de la guía de creación y ejecución de código propio en Giraph [6] al archivo `.bashrc` del usuario `hduser` le agregamos las siguientes líneas:

```
export HADOOP_CLASSPATH=/usr/local/giraph/giraph-core/target/
giraph-1.2.0-SNAPSHOT-for-hadoop-1.2.1-jar-with-dependencies.jar:
/home/hduser/lib/boruvka.jar
export LIBJARS=/usr/local/giraph/giraph-core/target/
giraph-1.2.0-SNAPSHOT-for-hadoop-1.2.1-jar-with-dependencies.jar,
```

```
/home/hduser/lib/boruvka.jar
```

Donde `boruvka.jar` es el nombre del contenedor `.jar` que contiene el código que se generó y el path indicado corresponde a la ubicación de dicho archivo. Luego, logeados con el usuario `hduser` iniciamos el servicio de *Hadoop* con los siguientes comandos:

```
$HADOOP_HOME/bin/start-dfs.sh
$HADOOP_HOME/bin/start-mapred.sh
```

Si todo salió bien, al ejecutar el comando `jps` deberiamos obtener el siguiente resultado:

```
441 TaskTracker
355 JobTracker
72 NameNode
180 DataNode
539 Jps
269 SecondaryNameNode
```

Los números de cada proceso pueden variar, sin embargo los 6 procesos deben estar corriendo.

La entrada es guardada y copiada al sistema de archivos *hdfs* de *Hadoop* utilizando el siguiente comando:

```
$HADOOP_HOME/bin/hadoop dfs -copyFromLocal /tmp/grafo_entrada.txt
/user/hduser/input/grafo_entrada.txt
```

Donde `grafo_entrada` es el nombre del archivo donde se tiene el grafo que se utiliza para las pruebas y que está ubicado en `/tmp`. Finalmente para ejecutar el código generado corremos el siguiente comando:

```
$HADOOP_HOME/bin/hadoop jar $GIRAPH_HOME/giraph-examples/target/
giraph-examples-1.2.0-SNAPSHOT-for-hadoop-1.2.1-jar-with-
dependencies.jar
org.apache.giraph.GiraphRunner -libjars ${LIBJARS}
 Boruvka.Boruvka -mc Boruvka.BoruvkaMasterCompute
 -vif Boruvka.BoruvkaVertexInputFormat
-vip /user/hduser/input/grafo_entrada.txt
-vof Boruvka.BoruvkaVertexOutputFormat
 -op /user/hduser/output/boruvka -w 3 -ca giraph.logLevel=debug
```

Donde especificamos la dirección de la ubicación de *Hadoop*, la ubicación de Giraph y con algunos parámetros indicamos opciones de ejecución tales como:

- `-libjars ${LIBJARS}`: con este parámetro indicamos que clase del jar ingresado en el `-libjars` queremos utilizar. En este caso la clase `Boruvka` del paquete `Boruvka`.
- `-mc`: con este parámetro indicamos la clase `MasterCompute` a utilizar. En este caso la clase `BoruvkaMasterCompute` del paquete `Boruvka`.
- `-vif`: con este parámetro indicamos el tipo de formato de entrada para los vértices. En este caso utilizaremos la clase `BoruvkaVertexInputFormat` del paquete `Boruvka`.
- `-vip`: con este parámetro indicamos el archivo de entrada a utilizar.
- `-vof`: con este parámetro indicamos el tipo de formato de salida para los vértices. En este caso utilizaremos la clase `BoruvkaVertexOutputFormat` del paquete `Boruvka`.
- `-op`: con este parámetro indicamos el destino de los archivos de salida generados por la ejecución de nuestro código.

- `-w`: con este parámetro indicamos el número de workers a utilizar para la ejecución del programa.
- `-ca`: con este parámetro indicamos opciones extra para la ejecución del programa. En este caso seteamos el nivel de logeo `giraph.logLevel` a `debug`.

6.4. Resultados

Una vez aplicado el algoritmo de MST Borůvka para el grafo de entrada ingresado, el árbol covertor mínimo esta a la vista. En la Figura 6.2 se puede observar gráficamente el árbol covertor mínimo señalado por las líneas grises sobre las aristas del grafo.

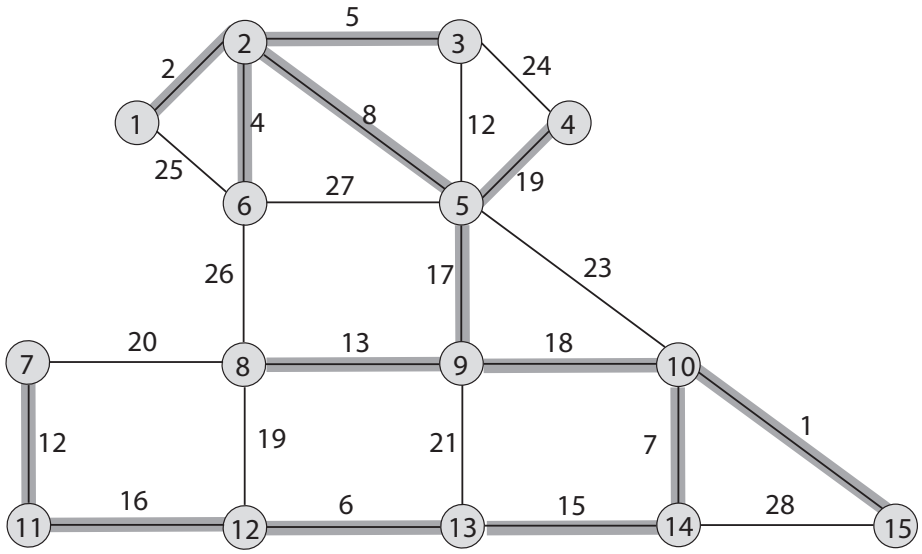


Figura 6.2: *Grafo de prueba resuelto.*

La salida del programa se muestra a continuación. Esta se corresponde con el formato de salida `BoruvkaOutputVertexFormat`

```
Nodo inicio: 1 Arista: 2  Nodo termino: 2
Nodo inicio: 2 Arista: 2 Nodo termino: 1
Nodo inicio: 3 Arista: 5  Nodo termino: 2
Nodo inicio: 4 Arista: 19 Nodo termino: 5
```

```
Nodo inicio: 5 Arista: 8  Nodo termino: 2
Nodo inicio: 6 Arista: 4  Nodo termino: 2
Nodo inicio: 7 Arista: 12 Nodo termino: 11
Nodo inicio: 8 Arista: 13 Nodo termino: 9
Nodo inicio: 9 Arista: 13 Nodo termino: 8
Nodo inicio: 10 Arista: 1 Nodo termino: 15
Nodo inicio: 11 Arista: 12 Nodo termino: 7
Nodo inicio: 12 Arista: 6 Nodo termino: 13
Nodo inicio: 13 Arista: 6 Nodo termino: 12
Nodo inicio: 14 Arista: 7 Nodo termino: 10
Nodo inicio: 15 Arista: 1 Nodo termino: 10
Nodo inicio: 9 Arista: 17 Nodo termino: 5
Nodo inicio: 11 Arista: 16 Nodo termino: 12
Nodo inicio: 9 Arista: 18 Nodo termino: 10
Nodo inicio: 13 Arista: 15 Nodo termino: 14
```

En la Figura 6.3 se pueden observar los tiempos de ejecución que tiene el programa usando la entrada antes señalada. Para esta prueba se fue modificando la cantidad de workers con que Hadoop ejecutaría a modo de observar la consistencia del programa entre varias ejecuciones. Cabe destacar que siempre se debe tener un worker como mínimo en una ejecución. Además al superar los 4 Workers, la máquina entrén un estado de inejecución por lo que las pruebas sólo se realizaron hasta con 3 workers.

En la Figura 6.4 se pueden observar los tiempos de ejecución cuando se utilizan distinta cantidad de mappers. Cabe destacar que la mínima cantidad de mappers es 2, ya que se utiliza uno para el `MasterComputation` y otro para `NodeComputation`. Como se puede observar los tiempos se condicen con la cantidad de $workers + 1$. Ya que según la estructura de giraph, cada `Worker` ejecuta una tarea de mappers. Al igual que en la Figura 6.3 las pruebas se detuvieron al alcanzar el mismo número de intentos por efectos similares.

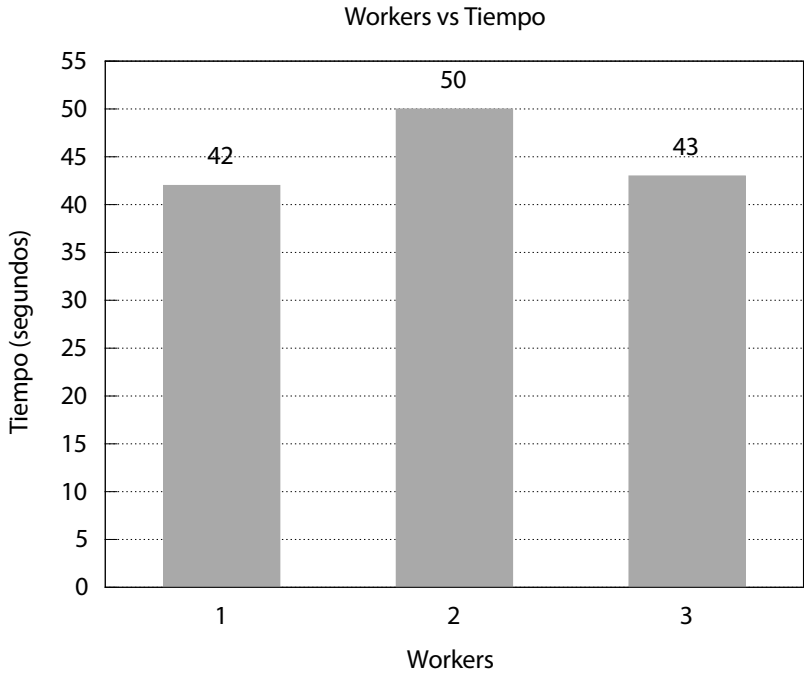


Figura 6.3: *Tiempo de trabajo para distintos workers.*

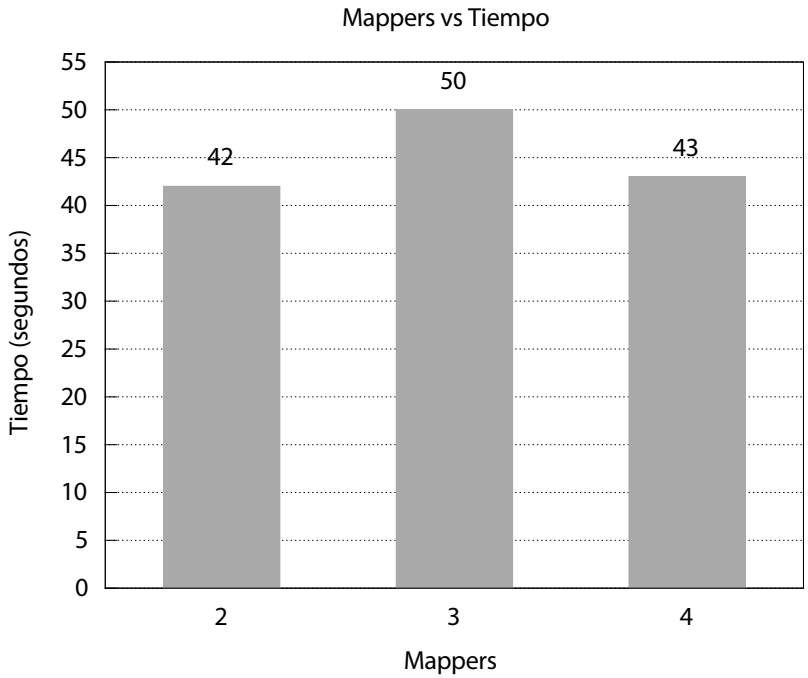


Figura 6.4: *Tiempo de trabajo por cantidad de mappers.*

7. Conclusión

En este capítulo se revisan las conclusiones obtenidas después del trabajo de un año en el proyecto de título.

En la Seccioón 7.1 se comentan las conclusiones extraídas del proceso de diseño de algoritmos para Apache Giraph.

En la Seccioón 7.2 se analizan las dificultades experimentadas durante el proceso de implementación de los algoritmos y sugerencias para trabajos venideros.

En la Seccioón 7.3 se comentan los resultados obtenidos en la parte experimental y se intenta entender el porque de los resultados.

En la Seccioón 7.4 habla sobre los trabajos que pueden continuar con lo realizado en esta memoria.

7.1. Conclusiones del Diseño de Algoritmos

Después de haber realizado el proceso de diseño de algoritmos siguiendo el modelo de pensamiento centrado en vértices que Apache Giraph propone, varias son las conclusiones que pueden ser extraídas y que en buena parte reflejan los resultados de esta memoria.

7.1.1. Dificultades del Modelo Centrado en Vértices

Una de las mayores dificultades del trabajo realizado se relaciona con el entendimiento del modelo centrado en ́vertices, no a nivel conceptual, sino a un nivel que permita el fácil modelamiento de problemas con este modelo. Esto se tradujo en una gran dificultad para traspasar algoritmos tradicionales al modelo centrado en vértices. Como se puede apreciar en los objetivos descritos en la Sección 1.2, se pensaba diseñar una serie de algoritmos provenientes desde la teoría de grafos e implementados en el modelo tradicional de grafos. Sin embargo la dificultad de este proceso influyó en que no se pudiera cumplir este objetivo a cabalidad. Como se puede apreciar en la Sección 4.4 se diseñaron solo tres algoritmos utilizando el modelo centrado en vértices, esto por las razones descritas anteriormente.

Diseñar para Giraph vs Traducir Algoritmos

La razón para la dificultad antes planteada tiene que ver con la diferencia que existe entre diseñar algoritmos para Apache Giraph y hacer una traducción de algoritmos que fueron pensados para un modelo distinto del de Giraph. El diseñar algoritmos para una plataforma específica requiere conocer lo que la plataforma nos entrega, y en base a eso diseñar un determinado algoritmo, lo cual puede ser más fácil de realizar ya que el marco de desarrollo es límitado a las características que la plataforma ofrezca. Cualquier cosa que no queda dentro de este límite evidentemente es eliminada. Sin embargo en el caso de la traducción de algoritmos desde el modelo tradicional al modelo centrado en vértices las restricciones son otras. El fin es conseguir implementar el comportamiento que el algoritmo tiene, independiente de si la plataforma entrega las mejores herramientas para conseguir esto. En nuestro caso de diseño de algoritmos árboles covertores mínimos, el que la ejecución esté segmentada

en iteraciones ejecutadas directamente por cada nodo, se convierte en una limitante, ya que existe mucha información que se necesita conocer a nivel global, más que a nivel de nodo.

7.2. Conclusiones de implementación de algoritmos

Por cuestiones de tiempo debido a la dificultad vivída durante la etapa de diseño de algoritmos relatada en el Sección 7.1, se implementó solamente el algoritmo de Borůka. Sin embargo de igual manera se pueden extraer conclusiones de la experiencia de programación para Apache Giraph. Son varios los elementos que se pueden considerar al momento de concluír sobre los hallazgos encontrados al implementar algoritmos en Giraph. A continuación se detallan algunos de estos.

Sobre el ambiente de desarrollo

Apache Giraph está diseñado para funcionar sobre versiones específicas de softwares auxiliares, como Java. Cada versión tiene un jdk asociado con el cual funciona. Lo mismo para las versiones de sistema operativo a utilizar. Por alguna extraña razón, la versión 1.10 de Giraph funciona con el jdk 1.7 y en debian 6.

Al principio del inicio de este proyecto, y en completa desobediencia de lo que la guía de instalación de Apache Giraph [7] dice, este autor utilizó otras versiones para los softwares antes mencionados, por lo que no se pudo ejecutar Giraph inmediatamente. Esto deja entre ver la dependencia directa de los componentes de Giraph a versiones específicas de otros softwares externos. Esta dependencia con versiones antiguas de softwares externos, también hace que otro software que es necesario para programar en Giraph también tenga que utilizarse en versiones anteriores a la más reciente por compatibilidad con java. Este es el caso del entorno de programacón Eclipse, del cual se debe ocupar un versión desactualizada.

Sobre Apache como Librería de programación

Después de programar un algoritmo ya diseñado en Giraph, se puede concluír que si bien es cierto el modelo centrado en vértices facilita la modelación de problemas y por ende la programación de los mismos, la librería de giraph no entrega todas las funcionalidades que se quisieran desde el punto de vista de programación.

En el caso particular de la implementación de Borůvka por tratarse de un algoritmo que calcula un árbol covertor mínimo, es imprescindible conocer información de otros vértices vecinos de manera más rápida sin tener que esperar otros supersteps. Esta dificultad de tener que esperar otros supersteps para conocer datos de los vértices cercanos, dificulta la implementación, ya que crea supersteps en los que sólo se esperan respuestas a las peticiones realizadas anteriormente, o iteraciones en las que sólo se esta actualizando información. Esto significa que hay que modelar y controlar de alguna forma que un nodo se comporte de una determinada forma para una iteración específica. Lo cual va en contra del modelo genérico de implementación para Apache Giraph. Por esta razón es que se proponen funcionalidades extras para que se integren a futuras versiones de Apache Giraph, las cuales pueden facilitar el trabajo de un programador.

- **Sistema de Query's**: Se debería incluír una forma de consultar los datos de los vértices a los que se puede llegar a través de una arista saliente de forma directa, sin tener que esperar a un mensaje de vuelta para tener la información. Esto agilizaría la implementación de algoritmos, ya que no se tendría que pensar en el caso de supersteps intermedios en los que solo se envía y/o recibe información para poder continuar con la ejecución.
- **Mejora del Sistema de mensajes**: Actualmente para incluír un mensaje enriquecido es necesario modificar una serie de clases nativas de Giraph como se realizó en la implementación de Borůvka, cuando esto debería permitirse por parte de la Librería nativa de Giraph. Cuando nos referimos a un mensaje enriquecido, nos referimos al envío de un objeto con las siguientes propiedades:
  - *Remitente*: Un identificador de qué vértice envía el mensaje.
  - *Superstep*: Superstep en el que este mensaje fue enviado.
  - *Mensaje*: Que el mensaje a transmitir pudiera ser un objeto genérico, transmitible y usable por el nodo de destino.

7.3. Conclusiones del Algoritmo

Luego de haber realizado las pruebas del algoritmo de Borůvka se pueden establecer las siguientes conclusiones sobre su rendimiento.

- **Correctitud**: En cuanto a la correctitud de los resultados de la ejecución del algoritmo de Borůvka se puede concluir que el algoritmo responde acorde a lo esperado entregando resultados que al ser inspeccionados y comparados con resultados ya conocidos para esta entrada son iguales.

- **Tiempo de ejecución:** En cuanto a los tiempos de ejecución se puede decir que se requiere una revisión del algoritmo para obtener mejores resultados. Toda vez que al realizar el siguiente análisis de tiempo de ejecución, se obtienen tiempos no óptimos ni cercanos a los óptimos.

  Por ejemplo, la ejecución de Borůvka para un grafo de 15 nodos, demoró en promedio 45 segundos, ó 3 segundos de procesamiento por cada vértice. Si el tiempo de ejecución creciera en forma lineal con el tamaño de las entradas, para un caso de uso como el de Facebook que posee un grafo de 2 Billones ($2x10^9$) de usuarios, Boruvka emplearía $6x10^9$ segundos o $1{,}6x10^6$ horas. Lo cual a todas luces es un tiempo que se escapa a toda utilidad práctica para este algoritmo.

- **Uso de memoria**: En cuanto al uso de memoria del algoritmo se puede concluir que también es necesario revisar el almacenado de variables y de las componentes contectadas, ya que son estas las que consumen la gran cantidad de memoria. Por cada nodo con sólo 1 componente conectada se requieren 463 bytes de memoria para la almacenación de su representación como un objeto JSON. Si consideramos el caso de uso de Facebook con 2 Billones de usuarios, se necesitarian $9,2x10^{11}$ bytes o 926 GB de memoria. Lo cual representa un cifra que pocos computadores podrían tener en memoria ram durante la ejecución. Por ende es necesario ajustar aún más las operaciones de almacenado de objetos JSON y restringirlas aún más.

- **Posibles casos de uso**: Como se ha detallado en los puntos anteriores, la funcionalidad del algoritmo en cuanto a los resultados ha sido comprobada, ya que se obtienen resultados acordes a las entradas ingresadas. Sin embargo, para ponerlo en uso en un ambiente real se necesitan algunas mejoras que también fueron comentadas en los puntos anteriores. Con estas mejoras se podría obtener un mejor rendimiento y con esto darle alguna utilidad a sistemas

reales.

A simple vista es fácil ver como el algoritmo de Borůvka podría ser útil en muchos ambientes productivos que involucren distintos entes, por su conversión directa a un grafo. Por ejemplo, Borůvka podría ser usado en los siguientes ambientes:

- **Motores de búsqueda**: Los motores de búsqueda podrían beneficiarse de algoritmos como el de Boruvka ya que al almacenar todos los posibles elementos en un grafo puedo moverme a través del grafo con gran fácilidad ya que se puede conocer el camino más corto que recorre todo el grafo y con estó ir, de ser necesario, de nodo en nodo buscando el elemento deseado, avanzando por una ruta óptima en algún sentido.
- **Sistemas de enrutamiento**: La navegación a través de un mapa hace tiempo que se modernizó con la incorporación de algoritmos de enrutamiento. En este sentido Borůvka podría ser una gran ayuda al generar el árbol covertor mínimo para una zona de manera fácil. De ahí en adelante sólo quedaría identificar los puntos de inicio y fin para ajustarse al modelo de enrutamiento.
- **Algoritmos de Sugerencia**: En estos algoritmos es primordial conocer caminos óptimos, ya que de aquí nacen las sugerencias para un usuario específico, el cual debido a su cercanía con algún otro ente puede interesarse por este. Y sí se interesa por este, también puede interesarse por los subsiguientes, por ende la noción de ruta óptima es fácil de ver, y con esto la utilidad de Borůvka también.

7.4. Trabajos futuros

Apache Giraph es una herramienta muy poderosa, que promete revolucionar el procesamiento de grafos de gran escala, gracias a su modelo de computo simplificado y adaptado a vértices. Desde el lanzamiento de la versión 1.0.0 en el 2013, ha sido desarrollado con esta plataforma variados trabajo de investigacióny muchos de estos resultados no han sido compartidos con la comunidad, lo que ha contribuído a que el desarrollo para Giraph se vea mermado. Es por eso que queda mucho por hacer con Giraph, y la industria aún debe actualizarse y empezar a utilizar esta herramienta para sus procesos internos. Es por ende que trabajos como los que se sugieren a continuación se postulan como proyectos futuros a considerar.

- **Implementación de un caso de uso específico**: Un buen trabajo a desarrollar, sería implementar un algoritmo específico para resolver un caso de uso de un determinado dataset. Esto permitiría que se desarrollará el flujo completo de desarrollo, desde encontrar cuales son las necesidades a satisfacer con el algoritmo, diseñarlo, implementarlo, controlar los resultados e integrarlo como parte de un sistema completo.

- **Extensiones a Giraph**: Así como el API de PGraph, existen varias funcionalidades que se podrían agregar a Giraph y que agregarían atractivo para los desarrolladores e investigadores que puedan utilizar Giraph.

- **Implementar más algoritmos**: Un buen aporte al proyecto de Apache Giraph, sería agregar más algoritmos diseñados e implementados desde el modelo tradicional al modelo centrado en vértices, al repositorio de algoritmos para Giraph. Estos podrían servir como punto de partida para otros proyectos posteriores, lo que contribuiría a masificar la plataforma de Giraph.

Bibliografía

[1] R. Angles, F. Meza, and F. Moya. Supporting property graphs in apache giraph. In *2016 XLII Latin American Computing Conference (CLEI)*, pages 1–12, Oct 2016. 13, 51

[2] Catriel Beeri and Philip A. Bernstein. Computational problems related to the design of normal form relational schemas. *ACM Trans. Database Syst.*, 4(1):30–59, March 1979. 11

[3] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proc. VLDB Endow.*, 8(12):1804–1815, August 2015. VII, 13

[4] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970. 11

[5] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001. 18

[6] Federico Meza Francisco Moya, Renzo Angles. Creación y ejecución de tu propio código en giraph. `http://guia-giraph.eu.pn/wiki/Creación_y_ejecución_de_tu_propio_código_en_giraph`, Dec 2016. Accedido: 2017-06-26. 63

[7] Federico Meza Francisco Moya, Renzo Angles. Giraph paso a paso: guía de instalación y primer uso de giraph. `http://guia-giraph.eu.pn/`, Dec 2016. Accedido: 2017-06-26. 71

[8] Federico Meza Francisco Moya, Renzo Angles. Guía de instalación de giraph en una máquina en linux. `http://guia-giraph.eu.pn/wiki/Instalación_de_Giraph_en_Linux`, Dec 2016. Accedido: 2017-06-26. 63

[9] Yong Guo, Marcin Biczak, Ana Lucia Varbanescu, Alexandru Iosup, Claudio Martella, and Theodore L. Willke. How well do graph-processing platforms perform? an empirical performance evaluation and analysis. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 395–404, Washington, DC, USA, 2014. IEEE Computer Society. VI, VI, VI, 29, 30, 31

[10] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM. 11

[11] Roman Shaposhnik, Claudio Martella, and Dionysios Logothetis. *Practical Graph Analytics with Apache Giraph*. Apress, Berkely, CA, USA, 1st edition, 2015. VI, 12

Glosario

Aggregator Es un tipo de variable que se puede compartir y poner disponible para todos los nodos de la red.

Arista Es uno de los componentes de un grafo. Es la unión entre dos pares de vértices. Puede almacenar un valor y poseer un sentido de dirección.

BSP *Bulk synchronous parallel* es un modelo de diseño para algoritmos de procesamiento paralelo. Incluye etapas de trabajo, distribución de información y barreras de sincronización.

Compute Función que define el comportamiento de un nodo para cada superstep.

EPS *Edges per second* unidad de medida de cuántas aristas una plataforma en ejecución puede procesar en un segundo.

Giraph Es un sistema de procesamiento iterativo pensado para fácilitar la escalabilidad de análisis de grandes conjuntos de datos. Esta basado en Pregel, sin embargo es opensource.

Grafo Estructura computacional abstracta que posee un conjunto de vértices conectados a través de un conjunto de aristas. Con esta estructura se pueden modelar un gran número de problemas.

Hadoop Es un sistema de procesamiento de grandes conjuntos de datos en sistemas distribuidos.

Mapper Un mapper es un objeto que se encarga de la subdivisión de elementos para que los workers puedan trabajar. Generalmente se tienen multiples mappers para generar la salida a los workers.

Nodo *Véase Vértice.*

Pregel Es un sistema de procesamiento de grandes conjuntos de datos desarrollado por Google y de código de fuente privativo.

Superstep Nombre con el que Giraph conoce a cada iteración de cada nodo, en la que se ejecuta la función compute.

Vértice Es uno de los componentes de un grafo. Se caracteriza por almacenar un valor y poseer un conjunto fínito de aristas entrantes y salientes. Se les conoce también como nodos.

VPS *Vertices per second* unidad de medida de cuántos vértices una plataforma en ejecución puede procesar en un segundo.

VoteToHalt Función con que un nodo se va a dormir hasta la siguiente iteración. Esta función termina el superstep para este nodo.

Worker Corresponde a un hilo de ejecución del programa. Generalmente se asigna un worker por cada vértice. En una máquina pueden existir varios workers trabajando a la vez.

ANEXOS

A. Implementación de MST

A.1. Funciones implementadas

A.1.1. Métodos relacionados con aristas

Algoritmo 1 AristaMenorPeso

1: $arista.\omega \leftarrow \infty$
2: $\forall\ e \in L$ ▷ L lista de aristas del nodo
3: **If** $e.\omega < arista.\omega$ **Then** $arista \leftarrow e$
4: **Return** $arista$

Algoritmo 2 ComunicarNumerodeVotos

1: **Return** $\lfloor Mensajes \rfloor$ ▷ tamaño lista con mensajes

Algoritmo 3 NodosEnComponente

1: **Return** $\lfloor ListaComponentes \rfloor$ ▷ tamaño de lista de Componentes

Algoritmo 4 NodoEnComponente

1: $\forall\ v \in L$ ▷ L lista de componentes
2: **If** $v == vAux$ **Then Return** $true$
3: **Return** $false$ ▷ false cuando nodo buscado no pertenece a L

Algoritmo 5 ActualizarComponentes

1: $\forall\ v \in L$ ▷ L lista de componentes
2: **If** $v == vAux$ **Then Return** $true$
3: **Return** $false$ ▷ false cuando nodo buscado no pertenece a L