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

Generación distribuida de grafos con Hadoop - MapReduce

FERNANDA SOLEDAD LÓPEZ GALLEGOS

Profesor Guía: RENZO ANGLES ROJAS

Profesor Co-guía: RODRIGO PAREDES MORALEDA

Memoria para optar al título de
Ingeniero Civil en Computación

Curicó – Chile
Agosto, 2020

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, 2022

Dedicado a mis padres, quienes me han dado todo en la vida

AGRADECIMIENTOS

"Agradezco a todas las personas que de alguna forma han guiado mi camino hasta donde estoy ahora."

Esas eran mis líneas de agradecimientos hasta ahora, pero la verdad hay personas que han sido y serán siempre una parte importante de mi vida y que me gustaría agradecer de forma especial. Así que ... comienzo otra vez:

Por supuesto que en primer lugar mis agradecimientos van a mis padres, Juan y Soledad, quienes siempre han confiado en mí y me han apoyado de forma incondicional. Gracias por siempre tener respuestas para mí, aunque a veces sean inventadas.

A mi hermano, Andrés (Toto, como le decía cuando era pequeña), quien estoy segura que siempre me defenderá, aunque no sea necesario. Aprovecho de mencionar a mis sobrinos, Trini y Franco, quienes llenan de alegría el alma.

También agradezco a mis tías, tíos y primos, que los nombraría, pero ellos saben que son muchos.

A mi Club de Lulú, Catalina Monsálve, Franchesca Gajardo, Vanesa Vergara, Camila Farías, Marión Yáñez y Jacqueline Puebla, con las que es super difícil ponerse de acuerdo para juntarse, pero que siempre están ahí. Ellas son de esas personas que uno puede dejarse caer con los ojos cerrados, porque sabes que te van a atrapar.

A mis amigos y compañeros de carrera, con quienes fue un gusto compartir estos casi seis años, durante los cuales las risas y buenos momentos no faltaron.

A la señorita Gladys, mi profesora de enseñanza básica en la escuela G-53 Vista Hermosa (la escuela con el mejor patio que he conocido), gracias por su dedicación, compromiso y ganas de enseñar.

A mis profesores de Taekwondo, Carlos Vergara y Jaime Jara por enseñarme a jamas rendirme, que el cansancio es psicológico y que uno siempre debe ser una mejor versión de si mismo.

A mis profesores de la Universidad, de cada uno de ustedes aprendí algo especial. En particular quiero agradecer a la profesora Ruth Garrido y al profesor Matt Bardeen por su confianza y consejos.

Por último, y muy especialmente a mis profesores guía, Renzo Angles y Rodrigo Paredes. Muchas gracias por su paciencia, apoyo y dedicación en este trabajo.

A Ustedes, de todo corazón, muchas gracias.

TABLA DE CONTENIDOS

página

ÍNDICE DE FIGURAS

página

ÍNDICE DE TABLAS

página

RESUMEN

Los grafos son una herramienta muy utilizada debido a su gran capacidad de modelar redes complejas. Desafortunadamente, encontrar conjuntos de datos representativos de la realidad, es un problema para muchos profesionales de diversas áreas. Esto, ha motivado la construcción de herramientas que permitan generar grafos de manera artificial. En ese sentido, R^{3}MAT es un método diseñado para producir grafos sintéticos cuyas características se asemejan a las que ocurren en el mundo real (ej. distribución de ley de potencia). A pesar de su buen desempeño en comparación con otros generadores, R^{3}MAT tiene problemas para generar grafos de muy gran tamaño. Además, el grafo más grande que se puede generar, se encuentra limitado por la cantidad de datos que se puedan gestionar en la memoria principal de un computador.

En este trabajo se estudian variantes distribuidas basadas en R^{3}MAT, con el objetivo de disminuir el tiempo de ejecución y al mismo tiempo soportar la generación de grafos más grandes que R^{3}MAT. Para ello, se diseñan e implementan métodos usando *Hadoop - MapReduce*, los cuales posteriormente son evaluados en términos de eficiencia (tiempo de ejecución), escalabilidad (tamaño del grafo) y realismo (ley de potencia).

Los resultados obtenidos muestran que: (*i*) para grafos con más de diez millones de nodos, los nuevos métodos (distribuidos) son más rápidos que R^{3}MAT (secuencial), (*ii*) los nuevos métodos soportan la generación de grafos más grandes que el método secuencial, (*iii*) los grafos producidos con los nuevos métodos presentan la propiedad de distribución de ley de potencia, y (iv) los nuevos métodos son mejores que los métodos distribuidos que se encuentran en el estado del arte, en el sentido que: presentan un mejor ajuste a una distribución de ley de potencia, permiten diferenciar la generación de un grafo dirigido de uno no dirigido, y aseguran la producción de una cantidad determinada de aristas, sin generar aristas repetidas.

1. Introducción

En este capítulo se describe la formulación de este trabajo, incluyendo la descripción del contexto del estudio, el problema estudiado, la hipótesis que se busca comprobar, así como los objetivos y alcances.

1.1. Contexto

Los grafos son muy estudiados y a la vez muy utilizados para modelar redes complejas y realizar análisis sobre ellas en áreas como *benchmarking*, *big data*, y *machine learning*, entre otras. A modo general, un grafo es un conjunto de objetos conocidos como nodos, que se relacionan a través de conexiones conocidas como aristas. La versatilidad de un grafo permite modelar desde redes sociales, las cuales consisten en un grupo de individuos interconectados por algún tipo de relación, hasta redes de proteínas que deben trabajar juntas para cumplir con alguna función específica [10].

Encontrar conjuntos de datos (*datasets*) para construir grafos que cumplan con las características necesarias para un proyecto en algún área es un trabajo problemático. Esto se debe principalmente al valor que tiene la información para los individuos y organizaciones, además de la falta de estructura de estos, lo que implica un trabajo adicional para su uso [24].

Los generadores de datos surgen justamente para cubrir la necesidad de información para la realización de un sinfín de proyectos, y se pueden definir como una herramienta que permite obtener instancias de datos que cumplen con ciertas características dadas por parámetros que generalmente son controlados por el usuario.

Unas de las características más buscadas en los datos generados artificialmente es que sean lo más representativos posibles de lo que ocurre en la realidad del mundo

que se modela y que a su vez, sean producidos de forma eficiente; sin embargo, son pocos los generadores que cumplen con estos requisitos.

Chakrabarti y Faloutsos [10] describen una serie de características de los grafos naturales [1], las cuales generalmente son patrones que se repiten en determinados tipos de redes y que son utilizados como base para determinar si los datos generados artificialmente son lo suficientemente representativos y útiles para un proyecto. Entre las características mencionadas, destaca la ley de potencia que dice que la probabilidad de que una variable aleatoria discreta x tome un valor k es proporcional a $k^{-\tau}$, con $\tau \in \mathbb{R}^+$. La ley de potencia es una distribución de probabilidad más sesgada que la distribución normal y permite modelar casos en los que hay muchos elementos con pocas conexiones y pocos elementos con muchas conexiones; por ejemplo, en Facebook hay muchas personas con pocos amigos y pocas personas con muchos amigos.

R-MAT es un algoritmo generador de grafos que siguen la ley de potencia. La versión original de este algoritmo trabaja dividiendo recursivamente la matriz de adyacencia del grafo hasta llegar a una celda única en la cual se agrega una arista, proceso que se repite hasta alcanzar la cantidad requerida de aristas [11].

R^{3}MAT [8] es un método basado en R-MAT, el cual utiliza un arreglo auxiliar que almacena la aridad esperada de cada nodo, que surge como solución al problema de restricción de memoria intrínseco que presenta una matriz cuando se necesita generar un grafo muy grande con la versión original de R-MAT. Cabe mencionar que este método conserva las propiedades del método original, las cuales son: parsimonia, porque el algoritmo se ejecuta con tan sólo cuatro parámetros; y realismo, dado que el grafo generado conserva las características de distribución de ley de potencia.

1.2. Planteamiento del Problema

Si bien R^{3}MAT reduce el tiempo de generación de R-MAT, dicho tiempo es aún considerable para grafos de gran tamaño. Además, se ha comprobado que R^{3}MAT presenta restricciones cuando el arreglo de aridades es muy grande y no se puede gestionar en memoria principal.

Una alternativa para mejorar los problemas mencionados anteriormente es el uso de procesamiento distribuido y paralelismo.

[1] Término usado para referirse a los grafos generados a partir de datos reales [10].

En la literatura se puede encontrar algunos métodos que hacen uso de dichas técnicas. Por ejemplo, PaRMAT es un método que utiliza *threads*, Boost y Graph500 son soluciones que implementan R-MAT utilizando MPI, mientras que TeGViz y PegasusN tienen implementaciones de R-MAT en Apache Spark y Hadoop, entre otras, las cuales, se describen en detalle en la Sección 2.5.

El análisis del trabajo relacionado permite concluir que existen varias herramientas, librerías y algoritmos que permiten generar grafos usando R-MAT tanto en implementaciones secuenciales como paralelas, cada una con sus *pros* y *contras*. En términos de eficiencia y realismo existen soluciones que son muy buenas, puesto que permiten generar grandes grafos en poco tiempo, por ejemplo, TrillionG [2]; sin embargo, la mayoría de ellas falla en que se generan aristas duplicadas o en su defecto se pierden aristas, se generan bucles, hay nodos aislados, y/o no se especifica el tipo de grafo que se permite generar (dirigido o no dirigido).

Considerando los problemas de R^{3}MAT y las restricciones de los métodos que usan procesamiento distribuido y paralelismo, en este trabajo se propone estudiar la implementación de R^{3}MAT usando el modelo de programación *MapReduce*.

MapReduce permite realizar procesamiento distribuido y paralelo de grandes conjuntos de datos en un entorno distribuido, dividiendo el trabajo de forma tal que cada máquina de un *cluster* procese de manera independiente una porción de los datos. Esto trae consigo el desafío de producir grafos correctos, usando las funciones *map* y *reduce*, sólo con información local; vale decir, sin conocer ningún tipo de información de lo que se está haciendo en otras máquinas del *cluster*.

Esta idea nos genera las siguientes preguntas de investigación:

- ¿Es posible implementar R^{3}MAT bajo el modelo de programación *MapReduce*?
- ¿Dicha implementación reducirá el tiempo de generación de grafos? (en comparación con R^{3}MAT)
- ¿Dicha implementación permitirá incrementar la capacidad de generar grafos de gran tamaño?

1.3. Hipótesis

La implementación del método R^{3}MAT usando el modelo de programación *MapReduce* permite reducir el tiempo de procesamiento e incrementar el soporte para

generar grafos de gran tamaño (millones de nodos).

1.4. Objetivos

Objetivo General

- Desarrollar y evaluar métodos de generación de grafos que combinen R^{3}MAT con *MapReduce*.

Objetivos Específicos

- Diseñar algoritmos para la generación de grafos basados en R^{3}MAT y *MapReduce*.
- Implementar los algoritmos propuestos usando Apache Hadoop.
- Evaluar la eficiencia de los algoritmos y validar sus propiedades (ej. ley de potencia).

1.5. Metodología

Basado en la metodología de investigación propuesta en [26], se contemplan tres fases en el desarrollo de este proyecto, las cuales apuntan al cumplimiento de cada uno de los objetivos propuestos. La primera fase está focalizada en el estudio del estado del arte; la segunda en el diseño, construcción y validación preliminar de los algoritmos usando *Hadoop* - *MapReduce*; y la tercera en la construcción de los experimentos y análisis de resultados.

- La Fase 1 o Fase Conceptual consiste en la:
  - Formulación y delimitación del problema.
  - Revisión de la literatura y construcción del marco teórico.
  - Investigación acerca de las tecnologías a utilizar para dar solución al problema.
  - Formulación de hipótesis y objetivos del proyecto.
- La Fase 2 o Fase de Diseño consiste en el:

    - Diseño, implementación y validación de los algoritmos de generación distribuida de grafos **dirigidos** usando *Hadoop - MapReduce*.
    - Diseño, implementación y validación de los algoritmos de generación distribuida de grafos **no dirigidos** usando *Hadoop - MapReduce*.

- La Fase 3 o Fase Empírica consiste en el/la:
    - Diseño de los experimentos.
    - Preparación del entorno de pruebas.
    - Ejecución de las pruebas.
    - Análisis de los resultados.

1.6. Alcances del Proyecto

Este Proyecto de Título es un trabajo de investigación que se extiende dentro de los siguientes límites:

- La investigación no incluye el diseño e implementación del algoritmo para otro sistema que no sea *Hadoop - MapReduce*.
- No se considera la investigación, diseño y/o implementación de variantes del algoritmo usando *threads* o aplicaciones en paralelo.
- Los algoritmos con los que se compara la solución propuesta tienen características similares.
- La ejecución del generador de grafos es mediante consola, es decir, no se considera el desarrollo de una interfaz gráfica de usuario.
- La salida del generador de grafos es el grafo en estructura de lista de aristas escrita en texto plano, por lo tanto, queda fuera de este trabajo la creación de una aplicación que permita visualizar el grafo como nodos y aristas.

1.7. Estructura del Documento

En el Capítulo 2, se muestra una breve reseña teórica de los diferentes componentes del problema, además de un análisis del estado del arte de los generadores de grafos, poniendo especial énfasis en sus ventajas y desventajas.

En el Capítulo 3, se describen los algoritmos para generar grafos dirigidos de forma distribuida usando *Hadoop - MapReduce*.

En el Capítulo 4, se describen los algoritmos para generar grafos no dirigidos de forma distribuida usando *Hadoop - MapReduce*.

En el Capítulo 5, se complementa la metodología de trabajo descrita en la Sección 1.5 con la descripción de la metodología usada para realizar la evaluación de los algoritmos. También, se muestran las mediciones efectuadas para los diferentes algoritmos implementados y la comparación con algunos de los métodos descritos en el estado del arte.

En el Capítulo 6, se realiza una síntesis de este trabajo en términos de los resultados obtenidos y el cumplimiento de los objetivos propuestos en este trabajo. Además, se entregan recomendaciones y propuestas para futuras mejoras en la generación de grafos.

2. Antecedentes

Este trabajo integra varias áreas del conocimiento que van desde teoría de grafos hasta modelos de programación distribuida. En este capítulo se describen estos y otros conceptos que permiten comprender esta documento junto con un análisis del trabajo relacionado.

2.1. Grafos

Un grafo es un conjunto de vértices, también llamados nodos, conectados en pares por medio de aristas, también llamadas arcos, que sirve para representar redes. Los grafos se utilizan en una gran variedad de aplicaciones, entre ellas: circuitos eléctricos, redes sociales, Internet, etcétera [27].

Formalmente, se define un grafo como un par $G = (N, E)$, donde N es un conjunto de $n = |N|$ nodos y E es un conjunto de $m = |E|$ aristas. Una arista está dada por un par de nodos (u, v), donde u es el nodo origen y v el nodo destino. Nótese, que cada arista induce una noción de dirección (desde el origen al destino), de modo que se dice que un grafo es no dirigido, cuando para cada una de sus aristas, existe su versión simétrica, vale decir, si la arista $(u, v) \in E$, entonces también existe la arista $(v, u) \in E$. En el caso de estos grafos no dirigidos, para ahorrar espacio se considera que la arista (u, v) es equivalente a la arista (v, u). Esto último permite reducir el uso del recurso memoria a la mitad [29].

Visualmente, los nodos de un grafo se representan como puntos, mientras que las aristas se representan como líneas conectando dichos puntos, tal como se muestra en la Figura 2.1.

En la Figura 2.1, el conjunto de nodos es $\{w, x, y, z\}$ y el conjunto de aristas es $\{e1, e2, e3, e4, e5\}$, en consecuencia el grafo tiene $n = 4$ nodos y $m = 5$ aristas. La

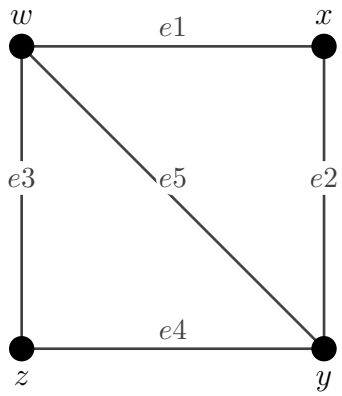


Figura 2.1: Ejemplo de un grafo.

arista $e2$ se puede denotar de la forma (x, y), donde x es el nodo de origen e y el nodo destino.

2.1.1. Generalidades

En algunos casos la relación representada por una arista no es simétrica, es decir, para una arista $e = (u, v)$, $(u, v) \neq (v, u)$. Para representar estas situaciones se establece que e es una arista dirigida. Si e es una arista dirigida, entonces u es llamado nodo origen y v nodo destino; además, e es una arista saliente de u y a la vez es una arista entrante de v. En consecuencia, un grafo en el que todas las aristas son dirigidas se llama **grafo dirigido**.

Si dos vértices están conectados por una arista, se dice que son **adyacentes**. Para un nodo $u \in N$, el número de vértices adyacentes a él, es llamado ***degree*** y es denotado por $d(u)$. En el caso de un grafo dirigido el *degree* se subdivide en dos clasificaciones: *in-degree* (denotado como $d^-(u)$) y *out-degree* (denotado como $d^+(u)$); siendo el primero el número de aristas entrantes de u y el segundo el número de aristas salientes de u. En la Figura 2.2 se muestra un ejemplo de un grafo dirigido, en el que $d^-(x) = 2$ y $d^+(x) = 1$.

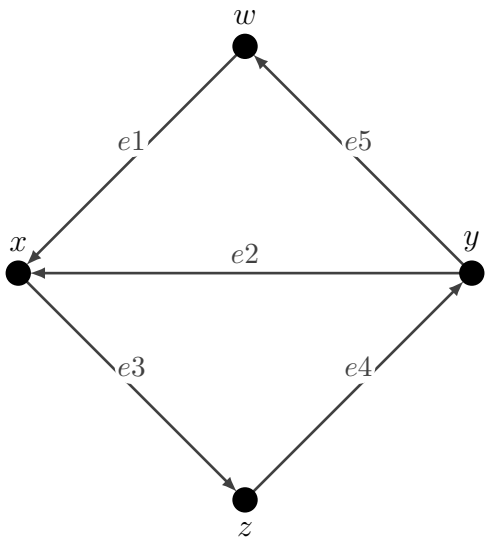


Figura 2.2: Ejemplo de un grafo dirigido.

2.1.2. Formas de representación

Existen formas distintas de representar un grafo, más allá de un simple dibujo, entre ellas se tiene la representación de matriz de adyacencia, de lista de aristas y de lista de adyacencia. La selección de alguna de esas alternativas va a depender del contexto en el cual se quiera utilizar el grafo y de las características del mismo, priorizando factores como el espacio de almacenamiento, utilidad y eficiencia en términos de tiempo.

- La **matriz de adyacencia** de un grafo $G = (N, E)$, como su nombre lo sugiere, es una matriz de $|N| \times |N|$ celdas, que se utiliza para representar la conexión entre los nodos. Un 1 en la fila u y la columna v definidas, indica que existe una arista que une el nodo u con el nodo v en el grafo, de lo contrario aparece un 0. Esta representación tiene como desventaja la poca eficiencia en términos de memoria, puesto que, por ejemplo, los grafos con millones de nodos son comunes y el costo de espacio necesario para almacenar una matriz de $|N| \times |N|$ celdas en un computador es prohibitivo [27].

La matriz de adyacencia del grafo G de la Figura 2.2 es:

$$A(G) = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

- La **lista de aristas** de un grafo G se representa como un conjunto de pares de la forma (u, v), donde u y v son dos nodos conectados por una arista en G. La lista de aristas del grafo G de la Figura 2.2 es (note que las aristas pueden aparecer en cualquier orden):

$$J(G) = \{(w, x), (y, x), (x, z), (z, y), (y, w)\}. \tag{2.1}$$

- La **lista de adyacencia** es un arreglo indexado por nodo, en donde por cada índice se tiene una lista de los nodos adyacentes [27].

2.1.3. Ley de potencia

Dos variables x e y están relacionadas por ley de potencia cuando:

$$y(x) = Ax^{-\tau} \tag{2.2}$$

dónde A y τ son constantes positivas [10].

Por su parte, una distribución de ley de potencia establece que la probabilidad de que una variable aleatoria discreta $x \in \mathbb{N}$ tome el valor k es proporcional a $k^{-\tau}$, es decir, $P(x = k) \propto k^{-\tau}$, para $\tau > 0$. Esto se puede reescribir como $P(x = k) = Ck^{-\tau}$, donde C es un factor de normalización, de modo que $\sum_{k=1}^{n} P(x = k) = 1$ [8].

2.1.4. Grafos que siguen la ley de potencia

En el contexto de grafos, una distribución de ley de potencia se puede interpretar como que la probabilidad de que un nodo de tenga k vecinos es $Ck^{-\tau}$. Por lo tanto, el número de nodos que tienen k vecinos es $nCk^{-\tau}$, lo que implica que estos nodos añaden $nCk^{1-\tau}$ aristas al grafo [8].

En el mundo real, existen varias redes que presentan esta característica, por ejemplo, las redes sociales como Facebook, en las cuales se puede encontrar pocas personas (nodos) que tienen muchos amigos y muchas personas con pocos amigos; sitios como DBLP o Google Scholar, en los cuales se puede encontrar pocos autores que han publicado muchos artículos y muchos autores que han publicado pocos artículos. También, son ejemplos de ley de potencia, el gráfico de colaboraciones de actores de películas, el patrón de citas de trabajos de investigación y las redes metabólicas de varios microorganismos [22].

2.1.5. Prueba de Kolmogorov-Smirnov

Kolmogorov-Smirnov (KS) [12] es una prueba no paramétrica, que permite comparar una distribución muestral con una distribución de probabilidad de referencia (ej. ley de potencia), cuantificando la distancia entre ambas distribuciones. Es importante considerar que las puntuaciones más pequeñas para KS denotan un mejor ajuste entre las distribuciones bajo comparación.

Para el cálculo de KS, se puede utilizar un *script* en R, el cual toma como entrada la lista de aristas de un grafo, calcula la distribución de probabilidad de cada aridad del grafo, y luego determina el ajuste de dicha distribución con la distribución de probabilidad de referencia.

2.2. R-MAT

Los generadores de grafos son programas que permiten crear grafos de forma sintética, es decir, que no se obtienen mediante una encuesta o experimento de la vida real [6]. Estos grafos pueden ser usados, por ejemplo, para realizar simulaciones sobre redes complejas o probar algoritmos, sobre todo en áreas como *Big data*.

Existen varios métodos que permiten generar grafos. R-MAT es un método de generación diseñado para crear grafos de forma sintética con tres propiedades principales: parsimonia, realismo y velocidad. Parsimonia en el sentido de que es un método que se puede ejecutar con pocos parámetros. Realismo, de modo que el grafo generado refleje las propiedades de los grafos naturales. Velocidad en términos de generar un grafo rápidamente [11].

El algoritmo R-MAT consiste en dividir recursivamente la matriz de adyacencia del grafo en cuatro partes o cuadrantes (llamadas a, b, c y d), a los cuales se les asignan

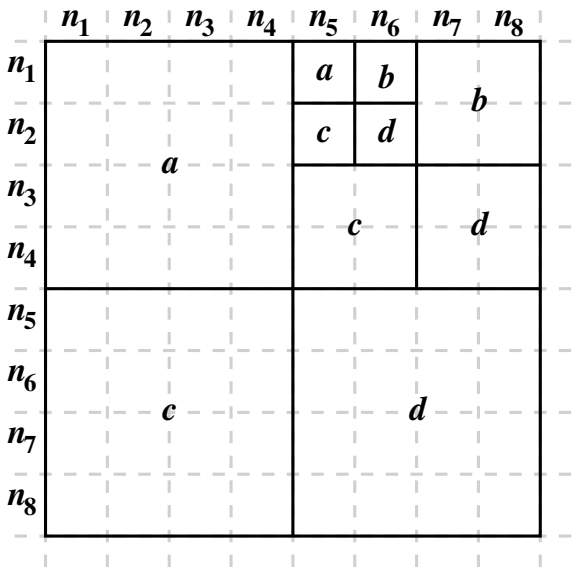


Figura 2.3: Ejemplo de la matriz de adyacencia generada por R-MAT [8].

probabilidades desiguales, de tal forma que $\rho_a + \rho_b + \rho_c + \rho_d = 1$. La recursión se hace hasta que se encuentra una matriz de 1×1, momento en el cual se agrega una arista al grafo (es decir, se inserta un "1" en la casilla correspondiente). El procedimiento termina una vez que un número m de aristas fue agregado a la matriz que inicialmente está vacía. En la Figura 2.3 se muestra un ejemplo de una matriz de adyacencia de 8×8 y las particiones generadas por el método recursivo.

La propiedad de realismo de los grafos generados con R-MAT tiene relación con las características presentes en los grafos naturales, entre las que, además, de la ley de potencia, se encuentran:

- Diámetro pequeño.

  El diámetro de un grafo es la mayor distancia mínima entre todos los pares de nodos, donde la distancia mínima entre dos nodos se refiere a la cantidad mínima de saltos necesarios para llegar de un nodo a a un nodo z del grafo, entendiendo el término salto como el hecho de pasar de un nodo a a un nodo b adyacente de a.

  El diámetro de los grafos naturales ha sido muy estudiado a lo largo de los

años y en la gran mayoría de casos se ha encontrado que es un número pequeño comparado con la cantidad total de nodos de un grafo [10].

En [10] se da el ejemplo de un experimento realizado en 1969 por Travers y Milgram, en el cual se le pedía a personas enviar una carta, a través de sus contactos, a un destinatario del cual se tenía poca información. El resultado del experimento fue que en aproximadamente 6 pasos la carta logró llegar al destinatario, lo cual es un número pequeño, teniendo en cuenta el tamaño de la población.

- Comunidades.

  El concepto de comunidad, más conocido por su nombre en inglés *cluster*, se refiere generalmente a un conjunto de nodos donde cada nodo está más cerca de los otros nodos dentro de su comunidad que de los nodos que están fuera de ella [10]. Esta característica se presenta en muchos casos reales, especialmente en redes sociales, en donde, por ejemplo, personas con intereses similares se encuentran más cerca.

2.3. R^{3}MAT

R^{3}MAT [8] es un método basado en R-MAT, el cual utiliza un arreglo auxiliar que almacena la aridad esperada de cada nodo y que surge como solución a los problemas de memoria de la versión original de R-MAT. En términos generales, el método produce un grafo en dos etapas: en la primera, se construye el arreglo de aridades simulando la partición recursiva de la matriz de adyacencia, y en la segunda, se genera las aristas usando diferentes estrategias dependiendo de si el grafo que se quiere generar es dirigido o no dirigido.

Respecto a las probabilidades utilizadas para simular el proceso recursivo de R-MAT, los autores de R^{3}MAT, determinan (en base a una evaluación experimental) la mejor combinación de valores para las probabilidades ρ_a, ρ_b, ρ_c y ρ_d que se deben usar, tanto para grafos dirigidos como para grafos no dirigidos. En su estudio indican que la mejor combinación de valores para grafos dirigidos corresponde a los valores $\rho_a = 0{,}75$, $\rho_b = 0{,}05$, $\rho_c = 0{,}19$ y $\rho_d = 0{,}01$, mientras que para el caso de grafos no dirigidos la mejor combinación de valores para las probabilidades corresponde a $\rho_a = 0{,}75$, $\rho_b = 0{,}05$, $\rho_c = 0{,}18$ y $\rho_d = 0{,}02$.

Las ventajas de este método son la gran cantidad de nodos que se puede generar, lo que sólo está limitado por el número máximo de elementos que puede contener un arreglo primitivo, y la no dependencia de las características del equipo en el que se ejecuta. Es decir, su funcionamiento es similar en una máquina con 64GB y en una con 4GB de memoria RAM en términos de rendimiento. Por otro lado, la principal debilidad de este algoritmo es el tiempo requerido para generar grafos de gran tamaño. Empíricamente, se ha demostrado que R^3MAT permite generar un grafo de 100 millones de nodos en aproximadamente 32 minutos utilizando una máquina virtual de 4GB de memoria RAM.

2.4. *MapReduce* y Apache Hadoop

Apache Hadoop o simplemente Hadoop es un framework de código abierto que permite el procesamiento distribuido de grandes conjuntos de datos a través de *clusters* de computadoras sin grandes características de procesamiento y/o almacenamiento de información, lo cual lo hace altamente atractivo para desarrollar aplicaciones de análisis de datos en distintas áreas, tales como: Finanzas, Marketing y Bioinformática [19].

A modo general, Hadoop tiene tres piezas fundamentales: MapReduce, HDFS y YARN. Estas tres piezas proporcionan una plataforma distribuida, escalable y tolerante a fallas para el almacenamiento y procesamiento de conjuntos de datos muy grandes en grupos de computadoras estándar. A diferencia de los *clusters* tradicionales de computación de alto rendimiento, Hadoop utiliza el mismo conjunto de nodos de cómputo para el almacenamiento de datos, así como para realizar los cálculos, lo que permite a Hadoop mejorar el rendimiento de los cálculos a gran escala mediante la colocación de los cálculos juntos con el almacenamiento [15].

Una de las ventajas más destacables de Hadoop MapReduce es su escalabilidad, puesto que literalmente permite aumentar la capacidad de procesamiento y almacenamiento con sólo agregar un equipo más al *cluster*, sin la necesidad de hacer algún cambio a nivel de programación [19]. Además, cabe destacar que en los últimos años, Hadoop y los *frameworks* relacionados se han convertido en el estándar de facto para almacenar y procesar grandes volúmenes de datos [15].

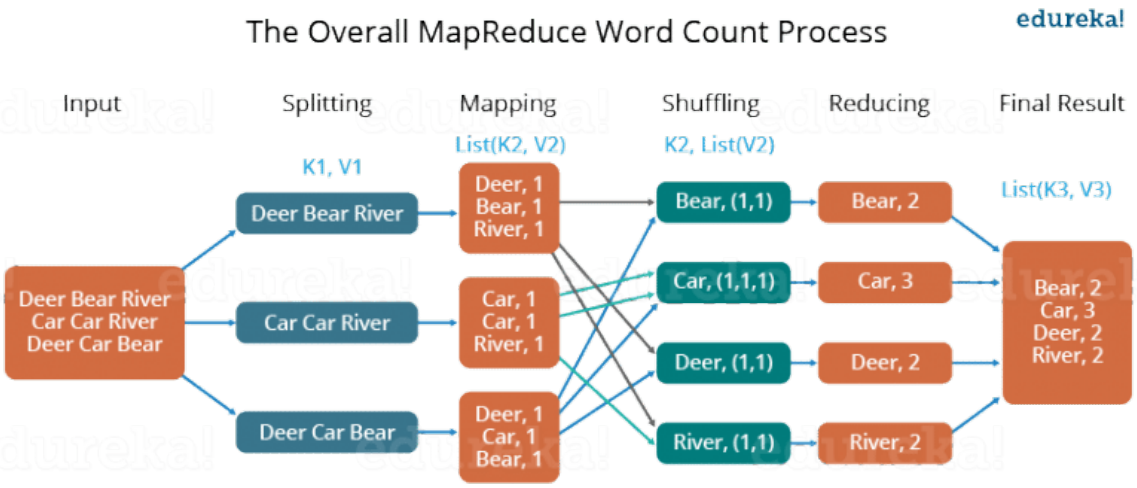


Figura 2.4: Ejemplo de MapReduce: *Word count* [9].

MapReduce

MapReduce es un modelo de programación que permite realizar procesamiento distribuido y paralelo de grandes conjuntos de datos en un entorno distribuido [13]. Este modelo consta de dos funciones a las cuales debe su nombre: *map* y *reduce*, las cuales permiten procesar de manera independiente en cada máquina una porción de los datos y luego comunicar resultados a un coordinador.

Un *job* de MapReduce generalmente divide el conjunto de datos de entrada en fragmentos independientes que son procesados por las tareas *map* de manera completamente paralela. El *framework* (MapReduce) ordena los resultados de los *maps*, los que luego se ingresan a las tareas *reduce*. Por lo general, tanto la entrada como la salida del *job* se almacenan en un sistema de archivos y el *framework* se encarga de programar tareas, monitorearlas y volver a ejecutar las tareas fallidas [1].

Un ejemplo del funcionamiento de MapReduce es el que se muestra en la Figura 2.4, en el cual se considera el problema de contar el número de apariciones de cada palabra en una gran colección de documentos. Primero, se divide el archivo de entrada en el número de nodos activos del *cluster*, en este caso, tres. Luego, cada

nodo ejecuta la función *map*, la cual se encarga de crear una tupla $< key, value >$ por cada palabra que aparece en el conjunto de datos, asignando la palabra como *key* y un 1 como *value* de dicha tupla, indicando que la palabra aparece una vez. Después, las tuplas con la misma *key* se envían al mismo nodo del cluster y se realiza alguna operación sobre esos datos. En este ejemplo, se suman, con el fin de obtener la cantidad total de veces que aparece cada palabra en el archivo de entrada. Finalmente, una vez que se termina el procesamiento en los nodos del *cluster*, estos envían al nodo maestro (coordinador) los resultados y este los escribe en el archivo de salida como una lista de tuplas $< key, value >$.

HDFS

HDFS es un sistema de archivos distribuido estructurado en bloques que está diseñado para almacenar grandes cantidades de datos de manera confiable en *clusters* de computadores estándar. HDFS se superpone sobre el sistema de archivos existente de los nodos de cómputo y almacena los archivos dividiéndolos en bloques más grandes (por ejemplo, 128 MB), los cuales se distribuyen a todos los nodos del *cluster* para permitir el procesamiento distribuido [15].

HDFS también almacena copias redundantes de los bloques de datos en múltiples nodos para garantizar la confiabilidad y la tolerancia a fallas. En particular, MapReduce explota estos conjuntos distribuidos de bloques de datos y la redundancia para maximizar el procesamiento local de datos de grandes volúmenes de información [15].

A nivel técnico, HDFS consta de dos servicios que son la base del sistema de archivos distribuidos: *NameNode* y *DataNode*. El primero almacena, administra y sirve los metadatos del sistema de archivos, mientras que el *DataNode* es un servicio por nodo que gestiona el almacenamiento del bloque de datos real en los *DataNodes* [15].

YARN

YARN es un sistema de administración de recursos que permite que varios *frameworks* de procesamiento distribuido compartan de manera efectiva los recursos de cómputo de un *cluster* y utilicen los datos almacenados en HDFS [15]. Para ello, YARN cuenta con algunos procesos, entre ellos: *ResourceManager* y *NodeManager*.

El proceso *ResourceManager* es el coordinador central de recursos que administra y asigna recursos a las diferentes aplicaciones (también conocidas como *Jobs*) envia-

das al *cluster*. El *ResourceManager* asigna recursos en respuesta a las solicitudes de recursos realizadas por los *Jobs*, teniendo en cuenta la capacidad y configuración del *cluster*. Por su parte, YARN *NodeManager* es un proceso por nodo que gestiona los recursos de un solo nodo de cómputo [15].

2.5. Trabajo Relacionado

Existen variadas soluciones en la literatura acerca de la generación de grafos grandes y realistas usando distintos algoritmos. Como la motivación de esta memoria parte del trabajo realizado en [8], en esta sección se analizan las soluciones secuenciales y distribuidas que implementan el algoritmo R-MAT.

Standford Network Analysis Platform o SNAP es una solución que implementa diferentes métodos de generación de grafos, específicamente optimizados para ser ejecutados en una máquina con gran cantidad de memoria. El método que implementa R-MAT recibe como parámetros el número de nodos, el número de arístas, las probabilidades de partición de la matriz y un generador de números aleatorios. La salida del algoritmo es un grafo dirigido.

El tiempo que tarda en generarse un grafo en las soluciones secuenciales es el principal problema que busca resolverse utilizando soluciones paralelas. En ese contexto, PaRMAT es un generador *multi-threaded* de grafos basado en R-MAT. PaRMAT está diseñado para explotar varios subprocesos y también para evitar errores cuando no hay mucha RAM disponible. Su funcionamiento se basa en dividir la matriz de adyacencia en matrices más pequeñas y la carga de trabajo entre varios subprocesos [17]. El autor menciona que PaRMAT es más rápido generando un grafo que SNAP y GTgraph, los cuales implementan una variante distribuida de R-MAT. Sin embargo, no hay mucha información acerca de ellos para poder realizar un análisis más detallado de sus características.

Boost [14] es un conjunto de librerías para $C++$ muy usada por la comunidad de desarrolladores. Dentro del conjunto de librerías de Boost se encuentra la clase *Scalable R-MAT generator*, la cual implementa una versión paralela de R-MAT. Para su ejecución, el usuario debe especificar la cantidad de nodos del grafo, el número de arístas, las probabilidades de partición de la matriz (ρ_a, ρ_b, ρ_c y ρ_d) y un generador de números aleatorios.

Graph500 es un *benchmark* para supercomputadoras basado en el análisis de

grafos de gran escala [28, 3]. Como parte de sus funciones, implementa un generador de grafos basado en el algoritmo Kronecker [18], el cual es similar a R-MAT. Esta solución permite que el grafo generado contenga aristas repetidas, aristas con el mismo nodo origen y destino (*self-loops*) y nodos "islas". Con esta solución, varios supercomputadores pueden generar un grafo de un trillón de nodos, sin embargo, lo hacen utilizando una gran cantidad de recursos informáticos, generalmente varios miles de servidores conectados a través de una red de alta velocidad, lo cual para la mayoría de los investigadores es inaccesible.

TeGViz es una herramienta de generación y visualización de grafos de gran escala de manera distribuida [16, 7]. Este software consta de dos módulos: generación de grafos y visualización. El módulo de generación de grafos TeG genera una amplia gama de grafos, incluyendo grafos aleatorios Erdos-Renyi y grafos realistas, incluyendo R-MAT y Kronecker directamente en sistemas distribuidos. Entre sus ventajas se encuentra que es un conjunto de herramientas dentro de las cuales se incluye R-MAT, además de su alta eficiencia y escalabilidad. En [16], se muestra que es posible generar un grafo de 10^6 nodos en 10 segundos, sin embargo, no se menciona la arquitectura utilizada para las pruebas.

TrillionG es un generador de grafos que puede generar grafos gigantes en poco tiempo usando una pequeña cantidad de memoria [21]. Según los autores, el método permite generar un grafo de un trillón de aristas siguiendo los modelos RMAT o Kronecker en dos horas usando un *cluster* de sólo 10 nodos (PC's), donde cada nodo está equipado con una CPU de seis núcleos de 3.50 GHz, 32 GB de memoria RAM y 4 TB de disco duro. En [21] se destaca la similitud que tienen de grafos generados con TrillionG con los grafos generados usando una implementación de la versión original de R-MAT propuesta en [11], pero no se menciona si el tipo de grafo generado es dirigido o no dirigido.

El generador TrillionG está implementado en el lenguaje Scala y está construido sobre Apache Spark, por lo que puede admitir una amplia gama de sistemas de almacenamiento como HDFS, Amazon S3, HBase y sistema de archivos local. Los parámetros que deben ser ingresados por el usuario para su ejecución corresponden a las probabilidades de partición de la matriz (ρ_a, ρ_b, ρ_c y ρ_d), la escala del grafo (*scale*), la relación entre el número de vértices y el número de aristas, valor para el ruido, número de máquinas/*theads*, formato de la salida y códec de compresión [2]. El número de nodos $|N|$ se calcula como 2^{scale}, mientras que el número de arístas

$|E|$ corresponde a $|N| \times 16$.

Por otra parte, PegasusN es un paquete de minería de grafos escalable y versatil, que se ejecuta en Hadoop y Spark [4, 20]. Para trabajar grafos enormes, PegasusN proporciona e integra algoritmos distribuidos para los siguientes cuatro tipos de operaciones con varias aplicaciones: análisis de la estructura de un grafo, búsqueda de patrones, visualización de grafos y un módulo que permite generar grafos gigantes, dentro del cual se encuentran los algoritmos R-MAT, Kronecker y Erdos-Renyi. Al igual que en el caso de TegViz, se requiere un análisis más profundo para determinar sus ventajas y desventajas, puesto que en [20] no se mencionan detalles acerca de la implementación y el entorno de prueba.

En [23], un trabajo titulado *MapReduce in MPI for Large-scale graph algorithms*, los autores presentan la creación de una librería paralela usando interfaz de paso de mensajes o MPI que permite expresar distintos algoritmos en el paradigma MapReduce. A grandes rasgos, la librería proporciona funciones *map* y *reduce* que operan independientemente en elementos de un conjunto de datos distribuidos a través de procesadores. Uno de los algoritmos que se incluyen en la librería es R-MAT. La implementación de R-MAT en este trabajo consiste en dividir la cantidad total de aristas que se quiere generar en P procesadores y luego eliminar las aristas repetidas, repitiendo el proceso hasta lograr el número de aristas requeridas. Una arista es generada simulando el proceso de subdividir la matriz de adyacencia en base a las probabilidades definidas, tal como se describe en la sección de R-MAT de esta memoria. Además, cabe destacar que para generar grafos no dirigidos, se realiza un postprocesamiento del resultado de R-MAT con el fin de conservar sólo la triangular superior de la matriz.

Los autores de [23] mencionan como las principales ventajas de su propuesta: la facilidad de uso, portabilidad, compatibilidad con los lenguajes $C++$, C y $Python$. En cuanto a sus desventajas está la falta de redundancia de datos y la nula tolerancia a fallas dada las características de MPI.

Por otro lado, también está GraphX, la cual es una librería de Spark para el análisis de grafos escalable. Entre sus funciones se encuentran: representación de grafos, algoritmos de grafos y generadores de grafos. En estos últimos se encuentra una función que permite generar un grafo usando una versión distribuida de R-MAT [25]. Entre sus ventajas se encuentra el soporte que la librería tiene, puesto que es parte de Spark. Por otro lado, sus desventajas son que realiza una implementación

de R-MAT basada en la matriz, y además su uso requiere de un conocimiento de Spark y la programación paralela, lo cual no es trivial.

3. Generación de Grafos Dirigidos

En este capítulo se describen los algoritmos desarrollados para la generación de grafos dirigidos de manera distribuida usando *Hadoop - MapReduce*.

3.1. Método D1

Se asume que $G = (N, E)$ denota el grafo a generar, donde $n = |N|$ es el número de nodos y $m = |E|$ es el número de aristas. Además, se conoce que para obtener un grafo siguiendo una distribución de ley de potencia, el número de aristas del grafo se calcula con la fórmula $m = \left\lceil \frac{2}{3} n \ln n + 0{,}384\,81 n \right\rceil$ [8].

A grandes rasgos, el método D1 consiste en generar un grafo dirigido siguiendo la idea de construir un arreglo de aridades como se propone en [8] y luego generar las aristas. El arreglo de aridades es una secuencia de números enteros de la forma $(d_1, d_2, \ldots, d_n)$, donde el elemento d_i representa el grado del nodo i en el grafo G.

El pseudocódigo de la clase principal de D1 se muestra en el Algoritmo 1, la cual es la encargada de recibir los parámetros de entrada y de configurar el *Job* que compone la solución. El *Job* que se configura consta de dos funciones: *map* y *reduce*, de las cuales se muestra el pseudocódigo de su implementación en el Algoritmo 2 y 3 respectivamente.

En la Figura 3.1 se muestra un ejemplo a alto nivel del flujo de datos que ocurre en la ejecución del método D1, el cual en detalle funciona de la siguiente manera:

1. Se generan de manera secuencial $nMaps$ archivos en el directorio $tmpDir$, los cuales cada uno contiene una porción del total de aristas (Algoritmo 1, Línea 3).

Algoritmo 1: D1 - Main

Input: $n \leftarrow$ número de nodos del grafo
$nMaps \leftarrow$ número de *maps*
Output: Lista de aristas.

```
1  begin
2      long m ← (2/3) · n · ln n + 0,384 81 · n
3      GenerateInputFiles(n, m, nMaps, tmpDir)
       /* Job 1                                                    */
4      job.setMapperClass(DirectedGraphMapper.class)
5      job.setReducerClass(DirectedGraphReducer.class)
6      job.waitForCompletionClass(true)
7  end
```

Algoritmo 2: D1 - DirectedGraphMapper

Input: $n \leftarrow$ número de nodos del grafo

```
1  begin
2      method MAP (LongWritable offset, Text line):
3          long m' ← Long.valueOf(line.toString())
4          RMat rmat ← new RMat()
5          for int i ← 0 to m' − 1 do
6              long edge[] ← rmat.GenerateEdge(n)
7              LongWritable source ← edge[0] −1
8              EMIT(LongWritable source, LongWritable 1)
9          end
10 end
```

Algoritmo 3: D1 - DirectedGraphReducer

```
1  begin
2      method REDUCE (LongWritable source, LongWritable
         values[v1, v2 ...]):
3          long degree ← 0
4          forall v ∈ values[v1, v2 ...] do
5              degree ← degree + v
6          end
7          long i ← 0
8          i ← WriteEdgeGoingBack(i, degree, source)
9          i ← WriteEdgeGoingForward(i, degree, source, n)
10 end
```

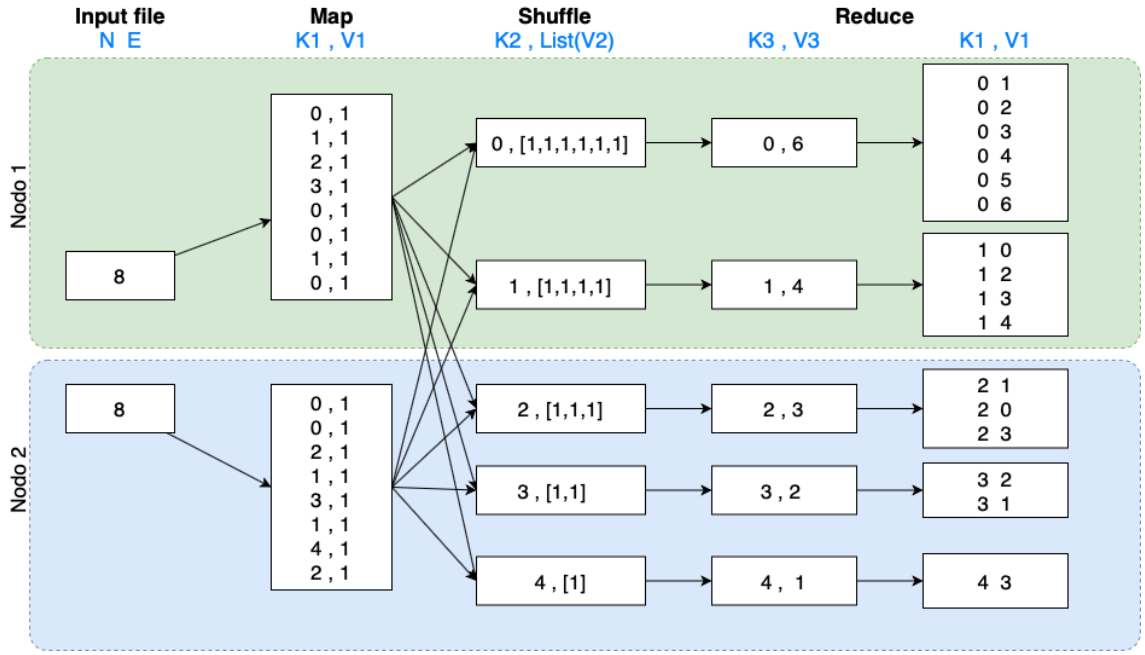


Figura 3.1: Ejemplo método D1.

2. Hadoop lee los archivos de entrada y luego ejecuta la función *map* pasando cada línea como argumento con el número de línea como *key* y el contenido de la línea como *value*.

3. La función *map* transforma el valor de *value* y lo asigna a la variable m', la cual representa el número de aristas que debe generar ese *map* en particular (Algoritmo 2, Línea 3). Por cada arista, es decir m' veces, se llama a la función `Rmat.GenerateEdge(n)`, la cual a través de una simulación del procesamiento recursivo de partición de la matriz retorna una arista de la cual se toma el nodo origen y se emite un par $< key, value >$, donde a *key* se le asigna el id del nodo origen (en adelante, *origen*) y a *value* un 1 que representa un incremento en el *degree* del nodo origen (Algoritmo 2, Líneas 5 a 9).

4. Hadoop recopila todos los pares $< origen, 1 >$ y los ordena por *key* (ver paso "Shuffle" en la Figura 3.1). Luego agrupa todos los valores emitidos para cada

clave única e invoca una función *reduce* por cada nodo de origen. Cada *reduce* recibe como *key* el identificador del nodo *origen* y como *value* un arreglo de valores 1.

5. La función *reduce* cuenta el número de elementos del arreglo recibido para el nodo origen que se está procesando y lo almacena en la variable *degree* (Algoritmo 3, Líneas 4 a 6). Una vez terminado ese cálculo, se crean las aristas por medio de las funciones $WriteEdgeGoingBack$ y $WriteEdgeGoingForward$, cuyo funcionamiento se describe en detalle en la Sección 3.1.1.

6. Hadoop escribe la salida final (lista de aristas) en el directorio de salida.

3.1.1. Generación de aristas

Una vez calculado el *degree* total de un nodo origen en particular, el proceso de generación de aristas ocurre de la siguiente manera:

1. A partir del nodo *origen* que se está reduciendo, mientras el número de aristas generadas sea menor que el *degree* y el nodo destino sea mayor o igual a cero, se generan aristas seleccionando como nodo de destino el id del nodo correspondiente a recorrer el "arreglo" yendo desde el id del nodo origen hacia atrás. Esta idea se implementa usando la función $WriteEdgeGoingBack$, cuyo pseudocódigo se muestra en el Algorimo 4.

Algoritmo 4: WriteEdgeGoingBack (long *i*, long *degree*, long *source*)

Input: *i* define un contador. *degree* define la cantidad de aristas a generar a partir del nodo de origen definido por *source*.

```
1 begin
2 |  long target ← source − 1
3 |  while i < degree ∧ target ≥ 0 do
4 |  |  EMIT(LongWritable source, LongWritable target)
5 |  |  i ← i + 1
6 |  |  target ← target − 1
7 |  end
8 |  return i
9 end
```

2. Si aún quedan aristas por crear, es decir, el número de aristas generadas hasta el momento es menor que el *degree*, y el nodo destino es menor que el número total de nodos (n), se generan aristas cuyo destino inicia en el nodo origen + 1 y luego se continúa seleccionando nodos destino yendo hacia adelante en el "arreglo". En el Algoritmo 5 se muestra la implementación de esta función, llamada $WriteEdgeGoingForward$.

Algoritmo 5: WriteEdgeGoingForward (long i, long *degree*, long *source*, long n)

Input: i define un contador. *degree* define la cantidad de aristas a generar a partir del nodo de origen definido por *source*.

```
1 begin
2 |  long target ← source + 1
3 |  while i < degree ∧ target < n do
4 |  |  EMIT(LongWritable source, LongWritable target)
5 |  |  i ← i + 1
6 |  |  target ← target + 1
7 |  end
8 |  return i
9 end
```

En la Figura 3.2 se ilustra el procedimiento de generación de aristas dirigidas tomando como referencia el nodo con $id = 1$ del ejemplo de la Figura 3.1. Una vez que en la etapa *reduce* se calcula el *degree* total del nodo que se está reduciendo (nodo 1 en este caso), usando el Algoritmo 4 se genera la arista (1,0), y luego, como aún el *degree* del nodo 1 es mayor a cero, usando el Algoritmo 5 se generan las aristas (1,2), (1,3) y (1,4).

3.2. Método D2

El método descrito en esta sección surge como una variante optimizada del método D1. Su diferencia radica en el uso de la función *Combiner*, la cual, teóricamente, permite optimizar el traspaso de información entre las máquinas del *cluster*.

Una sola tarea *map* puede generar muchos pares $< key, value >$ con la misma *key*, lo que hace que Hadoop distribuya (mueva) todos esos valores a través de la red hacia las tareas *Reducer*, incurriendo en una sobrecarga significativa que mediante

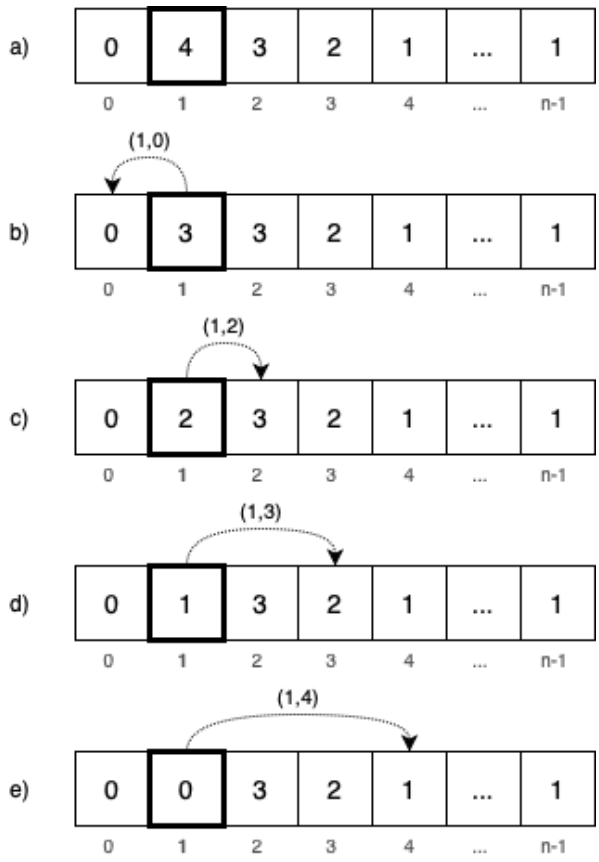


Figura 3.2: Ejemplo del proceso de generación de aristas dirigidas en el *reduce* del nodo 1 con *degree* inicial 4.

la función *Combiner* se puede aliviar. Por ejemplo, en el Algoritmo 1, cuando un *Mapper* encuentra múltiples ocurrencias del mismo nodo de origen en una sola tarea *map*, la función de *map* genera muchos pares $< key, value >$ intermedios, del tipo: $< origen, 1 >$, para transmitir a través de la red. Sin embargo, se puede optimizar este escenario si se suman todas las instancias de pares $< origen, 1 >$ a un solo par $< origen, degreeParcial >$ antes de enviar los datos a través de la red a los *Reducer*. Esto trae consigo un ahorro en la cantidad de mensajes que se mueven en la red, lo que redunda en un ahorro en el tiempo del proceso de generación del grafo.

Para optimizar tales escenarios, Hadoop implementa la función *Combiner*, la cual

permite realizar la agregación local de los pares $< key, value >$ de salida del *map*. Cuando se usa esta función, Hadoop llama al método *combine* en los resultados de la tarea de *Mapper* antes de persistir los datos en el disco y distribuir los datos a las tareas *Reduce*. Esto puede reducir significativamente la cantidad de datos transmitidos de las tareas de *Map* a las tareas de *Reduce*.

Cabe destacar que el *Combiner* es un paso opcional del flujo de MapReduce. Incluso cuando proporciona una implementación de este método, Hadoop puede decidir invocarlo sólo para un subconjunto de los datos de salida del *Mapper* o puede decidir no invocarlo en absoluto. Dado esta característica, es importante considerar que el *Combiner* debe tener la misma interfaz que la función *Reducer*, es decir, los tipos de pares $< key, value >$ de salida emitidos por el *Combiner* deben coincidir con el tipo de los pares $< key, value >$ de entrada del *Reducer*.

Algoritmo 6: D2 - Main

```
Input: n ← número de nodos del grafo
nMaps ← número de maps
Output: Lista de aristas.
1 begin
2 |  long m ← (2/3) · n · ln n + 0,384 81 · n
3 |  GenerateInputFiles(n, m, nMaps, tmpDir)
4 |  job.setMapperClass(DirectedGraphMapper.class)
5 |  job.setCombinerClass(DirectedGraphCombiner.class)
6 |  job.setReducerClass(DirectedGraphReducer.class)
7 |  job.waitForCompletion(true)
8 end
```

Algoritmo 7: D2 - DirectedGraphCombiner

```
1 begin
2 |  method REDUCE (LongWritable key, LongWritable values[v1, v2 ...]):
3 |  |  long degree ← 0
4 |  |  forall v ∈ values[v1, v2 ...] do
5 |  |  |  degree ← degree +1
6 |  |  end
7 |  |  EMIT(LongWritable key, LongWritable degree)
8 end
```

El pseudocódigo que se muestra en el Algoritmo 6 corresponde a la función princi-

pal de esta solución que utiliza *Combiner*. La única diferencia con la función principal del método D1 (ver Algoritmo 1) se encuentra en la Línea 5 del Algoritmo 6, en la cual se configura como *Combiner* la clase *DirectedGraphCombiner* cuyo pseudocódigo se encuentra en el Algoritmo 7.

Considerando lo anterior, el proceso de D2 se resume en los siguientes pasos:

1. Al igual que el método D1, se generan de manera secuencial $nMaps$ archivos en el directorio $tmpDir$, los cuales cada uno contiene una porción del total de aristas (Algoritmo 6, Línea 3).

2. Hadoop lee los archivos de entrada y luego ejecuta la función *map* pasando cada línea como argumento con el número de línea como *key* y el contenido de la línea como *value*.

3. La función *map* transforma el valor de *value* y lo asigna a la variable m'. Por cada arista a generar se llama a la función `Rmat.GenerateEdge(n)`, la cual a través de una simulación del procesamiento recursivo de partición de la matriz retorna un arreglo *edge*, el cual representa una arista con el nodo origen en la primera posición y el nodo destino en la segunda. De dicha arista se toma el nodo origen ($edge[0]$) y se emite un par $< origen, 1 >$.

4. Antes de persistir los datos en el disco para barajarlos y entregarlos al *Reducer*, el *Combiner* se encarga de realizar una suma parcial de las salidas del *Mapper*, lo que para este caso representa un cálculo parcial del *degree* de un nodo. Cabe destacar que este paso se realiza de forma local en cada nodo del *cluster*, lo que permite reducir la cantidad de datos que deben transferirse en el proceso de comunicación entre los nodos del *cluster*.

5. Hadoop recopila todos los pares $< origen, degreeParcial >$ y los ordena por *key* (ver paso "Shuffle" en la Figura 3.3). Luego agrupa todos los valores emitidos para cada clave única e invoca una función *reduce* por cada nodo de origen. Cada *reduce* recibe como *key* *origen* y como *value* un arreglo de valores $degreeParcial$.

6. La función *reduce* suma los *degree* calculados de manera parcial para el nodo origen que se está procesando y lo almacena en la variable *degree* (Algoritmo

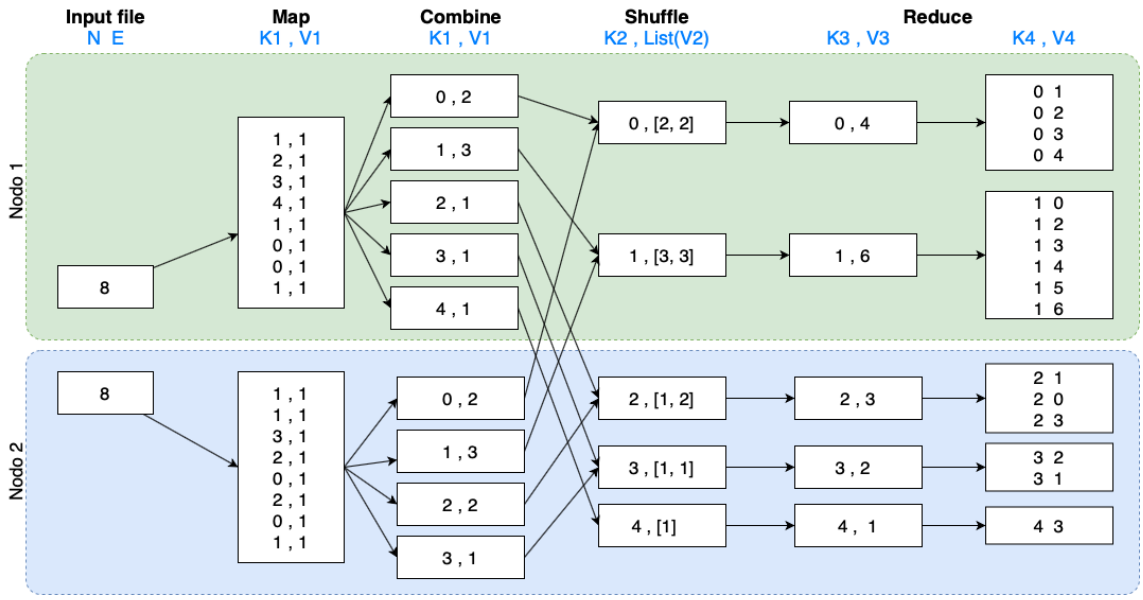


Figura 3.3: Ejemplo método D2.

3, Líneas 4 a 6). Una vez terminado ese cálculo, se crean las aristas por medio de las funciones $WriteEdgeGoingBack$ y $WriteEdgeGoingForward$, cuyo funcionamiento se describe en detalle en la Sección 3.1.1.

7. Hadoop escribe la salida final en formato de lista de aristas en el directorio de salida.

Como se mencionó anteriormente, el beneficio de usar un *Combiner* es que permite reducir la cantidad de datos que se transmiten entre los nodos del *cluster*. En el ejemplo que se muestra en la Figura 3.3 , el *Combiner* recibe x número de pares $< origen, 1 >$ como entrada y genera un solo par $< origen, x >$. Por ejemplo, si una entrada procesada por una tarea de *map* tenía 1,000 ocurrencias del nodo origen "0", el *Mapper* tendría que generar mil pares de la forma $< 0, 1 >$, mientras que el *Combiner* generaría sólo un par $< 0, 1000 >$, reduciendo así la cantidad de datos que deben transferirse a las tareas *Reducer*.

4. Generación de Grafos No Dirigidos

En este capítulo se describen los métodos desarrollados para la generación de grafos no dirigidos de manera distribuida usando *Hadoop - MapReduce*.

Si bien las ideas descritas en el Capítulo 3 son en gran parte aplicables a la generación de grafos no dirigidos, estas suponen un problema si lo que se quiere generar es un grafo sin aristas repetidas.

Recordando que en un grafo no dirigido $G = (N, E)$, la arista $(u, v) \in E$ es equivalente a la arista (v, u), si el generador produce la arista (u, v), se considera como repetida la arista (v, u). En otras palabras, si lo que se quiere es no generar aristas repetidas, entonces el generador debe producir sólo una de las dos aristas (u, v) o (v, u), pero nunca ambas.

En la Figura 4.1 se ilustra el método de producción de aristas descrito en la Sección 3.1.1. En ella se puede notar que cuando se reduce un nodo i y este genera aristas hacia atrás y adelante en el arreglo, al reducir el nodo $i+1$ y generar las aristas hacia atrás, se produce una arista simétrica a la generada durante la generación de aristas del nodo i. En el ejemplo, el nodo 2 produce la arista $(2, 3)$ y por otro lado el nodo 3 produce su arista simétrica $(3, 2)$.

Teniendo en consideración lo anterior, en este capítulo se describen tres métodos que permiten resolver el problema de las aristas repetidas (U1, U2 y U3). Estos métodos deben lidiar con los problemas intrínsecos de los programas distribuidos, en particular, el problema de la localidad de la información.

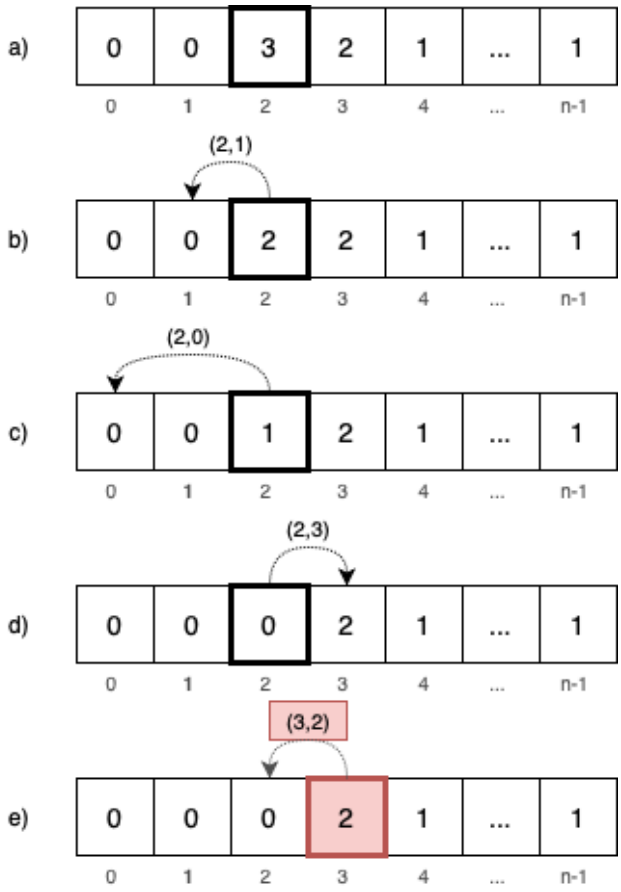


Figura 4.1: Producción de aristas repetidas.

4.1. Método U1

El método U1 permite construir un grafo dirigido en dos fases: en la primera fase se construye un grafo con aristas repetidas; en la segunda fase se eliminan las aristas repetidas. Notar que esto último implica que el grafo resultante tiene menos aristas que las calculadas al inicio.

La generación del grafo con aristas repetidas que se lleva a cabo en el primer *job*, ocurre de la misma forma que la generación de un grafo dirigido, es decir:

1. En el *Mapper*, en este caso llamado *UndirectedGraphMapper*, se genera de forma distribuida un archivo temporal que contiene m pares $< key, value >$, donde cada key corresponde al identificador de un nodo y cada $value$ a un 1, indicando

Algoritmo 8: U1 - Main

Input: $n \leftarrow$ número de nodos del grafo
$nMaps \leftarrow$ número de *maps*
Output: Lista de aristas.

```
1  begin
2      long m ← (2/3) · n · ln n + 0,384 81 · n
3      GenerateInputFiles(n, m, nMaps, tmpDir)
       /* Job 1                                                    */
4      job.setMapperClass(UndirectedGraphMapper.class)
5      job.setReducerClass(UndirectedGraphReducer.class)
6      job.waitForCompletionClass(true)
       /* Job 2                                                    */
7      job2.setMapperClass(UndirectedGraphMapper_2.class)
8      job2.setReducerClass(UndirectedGraphReducer_2.class)
9      job2.waitForCompletionClass(true)
10 end
```

de esa forma que el *degree* del nodo aumenta una unidad.

2. La clase *UndirectedGraphReducer*, presentada en el Algoritmo 10, calcula el *degree* total de cada nodo, de forma independiente, sumando todos los elementos del arreglo recibido por un nodo particular. Luego, a partir del nodo genera las aristas, tal como se describe en la Sección 3.1.1.

En el segundo *job* se eliminan las aristas repetidas. Esta tarea es realizada por la función *map* definida en el Algoritmo 11 y por la función *reduce* definida en el Algoritmo 12.

En términos generales, el proceso *MapReduce* del segundo *job* consiste en codificar una arista de tal forma que, aquellas que sean iguales y/o simétricas sean recibidas por el mismo proceso *reduce*, lo que permite escribir en el archivo de salida sólo una de ellas. A continuación se detalla el paso a paso de este *job*:

1. Cada *map* recibe como *value* una línea del archivo temporal que contiene el grafo con aristas repetidas. Dicha línea contiene por un lado el nodo origen y por otro el nodo destino de la arista, ambos separados por un tabulador.

2. Se procesa la línea de texto para separar el nodo origen y el nodo destino y asignar cada uno a una variable (ver Algoritmo 11, Líneas 3 y 4).

Algoritmo 9: U1 - UndirectedGraphMapper

```
   Input: n ← número de nodos del grafo
 1 begin
 2 |  method MAP (LongWritable offset, Text line):
 3 |  |  long m' ← Long.valueOf(line.toString())
 4 |  |  RMat rmat ← new RMat()
 5 |  |  for int i ← 0 to m' − 1 do
 6 |  |  |  long edge[] ← rmat.GenerateEdge(n)
 7 |  |  |  LongWritable source ← edge[0] −1
 8 |  |  |  EMIT(LongWritable source, LongWritable 1)
 9 |  |  end
10 end
```

Algoritmo 10: U1 - UndirectedGraphReducer

```
 1 begin
 2 |  method REDUCE (LongWritable source, LongWritable
   |    values[v1, v2 . . .]):
 3 |  |  long degree ← 0
 4 |  |  forall v ∈ values[v1, v2 . . .] do
 5 |  |  |  degree ← degree + v
 6 |  |  end
 7 |  |  long i ← 0
 8 |  |  i ← WriteEdgeGoingBack(i, degree, source)
 9 |  |  i ← WriteEdgeGoingForward(i, degree, source, n)
10 end
```

3. Se calcula el identificador de la arista, tomando como referencia el nodo con menor identificador entre el nodo de origen y nodo destino, de tal forma que una determinada arista tenga el mismo identificador que su arista simétrica, es decir, el identificador de la arista (u, v) es igual al de la arista (v, u).

   El identificador de la arista se calcula con la fórmula:

$$idArista = min(origen, destino) * n + max(origen, destino) \tag{4.1}$$

   donde *origen* y *destino* corresponden al identificador del nodo de origen y nodo destino respectivamente, y n al número total de nodos del grafo. Notar que dicha fórmula, es equivalente al IF del Algoritmo 11 (Línea 6 a 10).

Algoritmo 11: U1 - UndirectedGraphMapper_2

```
   Input: n ← número de nodos del grafo
 1 begin
 2   method MAP (LongWritable offset, Text line):
 3     long source ← Long.valueOf(line.toString().split( "\t ")[0])
 4     long target ← Long.valueOf(line.toString().split( "\t ")[1])
 5     LongWritable nkey
 6     if source ≤ target then
 7       nkey ← source ∗ n + target
 8     end
 9     else
10       nkey ← target · n + source
11     end
12     EMIT(LongWritable nkey, LongWritable 1)
13 end
```

Algoritmo 12: U1 - UndirectedGraphReducer_2

```
   Input: n ← número de nodos del grafo
 1 begin
 2   method REDUCE (LongWritable key, LongWritable values[v_1, v_2 ...]):
 3     forall v ∈ values[v_1, v_2 ...] do
 4       source ← key/n
 5       target ← key mód n
 6       EMIT(LongWritable source, LongWritable target)
 7       break
 8     end
 9 end
```

4. Se emite como mensaje desde el *map* un par $< key, value >$, en donde el *key* toma el identificador de la arista y *value* tiene el valor 1, representando de esta forma que esa arista aparece 1 vez.

5. Por cada identificador de arista se produce un *reduce* (Algoritmo 12). Cada reduce recibe como argumentos el identificador de la arista (*key*) y una lista de *unos* que representa la cantidad de veces que una arista o su arista simétrica aparecieron en el grafo de entrada.

   Dado que la idea es eliminar las aristas repetidas, en este paso, independiente de la cantidad de unos recibidos, se emite un único mensaje $< origen, destino >$,

donde *origen* y *destino* se calculan en base al identificador de la arista, usando las Fórmulas 4.2 y 4.3 respectivamente (ver Algoritmo 12, Líneas 4 y 5).

$$origen = idArista/n \tag{4.2}$$

$$destino = idArista \mod n \tag{4.3}$$

6. Hadoop escribe la salida final (lista de aristas) en el directorio de salida.

En la Figura 4.2 se muestra un ejemplo gráfico del proceso descrito anteriormente. En el primer bloque (*Job* 1), leyendo de izquierda a derecha ocurre la generación del grafo con aristas repetidas. En la salida del primer bloque se puede ver que en un nodo del *cluster* se generó la arista $(0, 1)$ y en otro se generó la arista $(1, 0)$. En el segundo bloque (*Job* 2) se eliminan estas y todas las aristas duplicadas. El *map* correspondiente codifica las aristas $(0, 1)$ y $(1, 0)$, y en ambos casos emite un mensaje $(1, 1)$. Posteriormente dichos mensajes son ordenados en un proceso interno que lleva a cabo Hadoop. Luego, como ambas aristas tienen como *key* el número uno, entonces son procesadas por el mismo *reduce*, el cual decodifica el identificador de la arista y escribe en el archivo de salida sólo la arista $(0, 1)$.

Analizando la cantidad de aristas producidas por este método, se estima que la pérdida de aristas es aproximadamente 0,4 % para grafos de un millón de nodos, el cual que disminuye para un grafo de diez millones de nodos, en donde la pérdida es de un 0,2 % aproximadamente. Este comportamiento sugiere que a medida que se aumenta la cantidad de nodos del grafo, el porcentaje de pérdida de aristas es menor.

4.2. Método U2

A diferencia de los métodos descritos anteriormente, U2 no se basa en construir un arreglo de aridades, sino que consiste en: primero contruir un grafo que puede tener aristas repetidas usando el nodo *origen* y nodo *destino* que retorna el método *GenerateEdge*, y luego eliminar las aristas repetidas. Como *GenerateEdge* puede generar aristas repetidas, al eliminarlas, el resultado final es un grafo con menos aristas que las definidas al inicio.

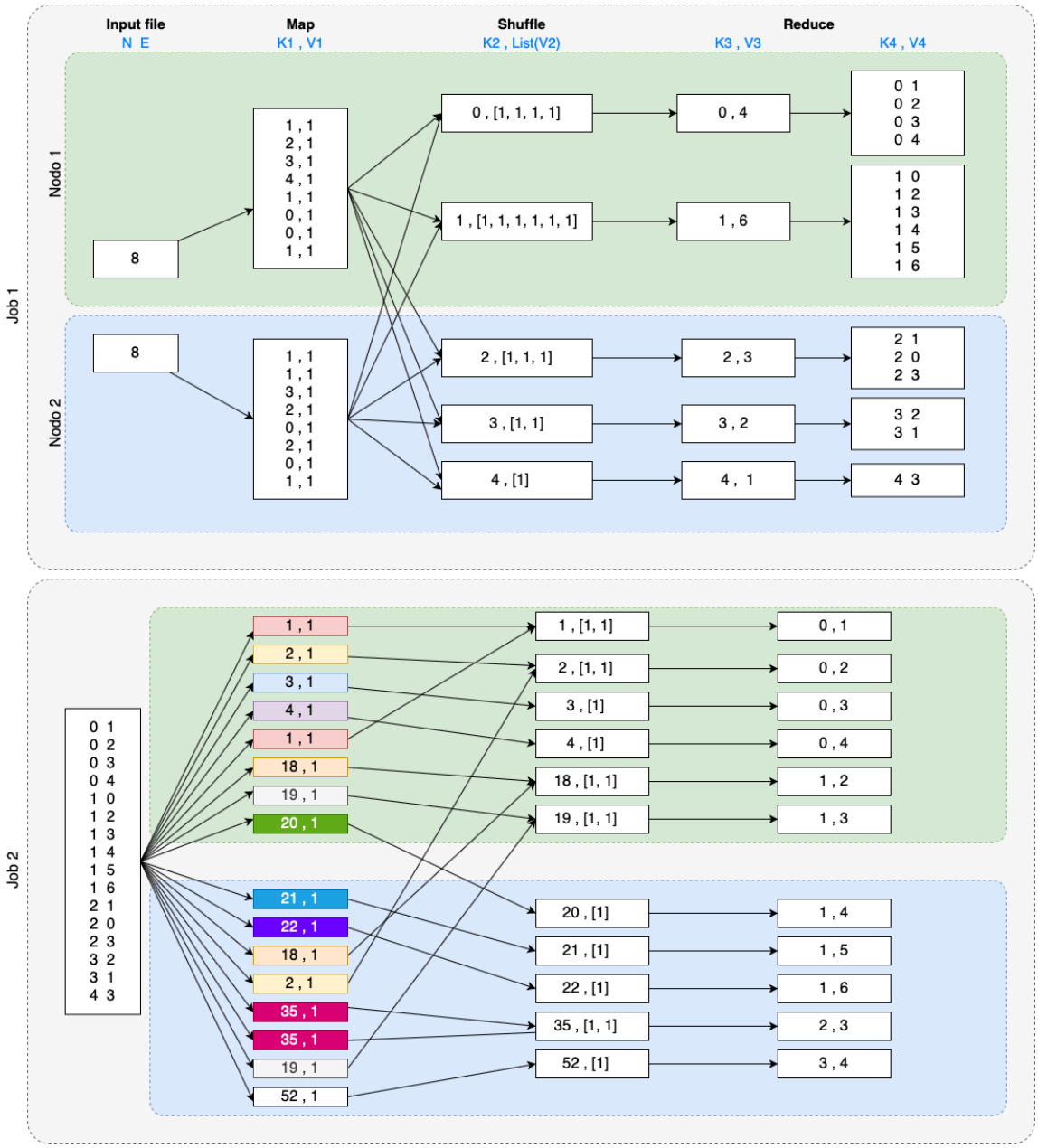


Figura 4.2: Ejemplo del método U1.

La implementación de esta idea usando *MapReduce*, consiste en configurar un único *job* de Hadoop (ver Algoritmo 13), el cual a su vez está compuesto de un *map* y un *reduce*, cuyo pseudocódigo se muestra en el Algoritmo 14 y 15 respectivamente.

Algoritmo 13: U2 - Main

```
   Input: n ← número de nodos del grafo
   nMaps ← número de maps
   Output: Lista de aristas.
1  begin
2      long m ← (2/3) · n · ln n + 0,384 81 · n
3      GenerateInputFiles(n, m, nMaps, tmpDir)
       /* Job 1                                                    */
4      job.setMapperClass(UndirectedGraphMapper.class)
5      job.setReducerClass(UndirectedGraphReducer.class)
6      job.waitForCompletionClass(true)
7  end
```

Algoritmo 14: U2 - UndirectedGraphMapper

```
   Input: n ← número de nodos del grafo
1  begin
2      method MAP (LongWritable offset, Text line):
3          long m' ← Long.valueOf(line.toString())
4          RMat rmat ← new RMat()
5          for int i ← 0 to m' − 1 do
6              long edge[] ← rmat.GenerateEdge(n)
7              LongWritable source ← edge[0] −1
8              LongWritable target ← edge[1] −1
9              if source ≠ target then
10                 LongWritable eKey
11                 if source ≤ target then
12                     eKey ← source · n + target
13                 end
14                 else
15                     eKey ← target · n + source
16                 end
17                 EMIT(LongWritable eKey, LongWritable 1)
18             end
19         end
20 end
```

El proceso de construcción de la lista de aristas del grafo usando el Método U2 se resume en los siguientes pasos:

1. Se generan de manera secuencial $nMaps$ archivos en el directorio $tmpDir$, los

Algoritmo 15: U2 - UndirectedGraphReducer

```
   Input: n ← número de nodos del grafo
1  begin
2  |  method REDUCE (LongWritable key, LongWritable values[v1, v2 ...]):
3  |  |  forall v ∈ values[v1, v2 ...] do
4  |  |  |  long source ← key/n
5  |  |  |  long target ← key  mód n
6  |  |  |  EMIT(LongWritable source, LongWritable target)
7  |  |  |  break
8  |  |  end
9  end
```

cuales cada uno contiene una porción del total de aristas (Algoritmo 13, Línea 3).

2. Por cada archivo de entrada, se ejecuta una función *map*, la cual transforma el valor de *value* recibido como parámetro y lo asigna a la variable m', que es el número de aristas que debe generar ese *map* en particular.

3. En un *map*, por cada arista a generar se llama a la función `rmat.GenerateEdge(n)`, la cual, a través de una simulación del procesamiento recursivo de partición de la matriz, retorna una arista. De dicha arista se toma el nodo origen y nodo destino, los cuales son usados para calcular un identificador de la arista usando la Fórmula 4.1.

4. Una vez calculado el identificador de la arista, el *map* emite un mensaje $< key, value >$, en donde *key* corresponde al identificador de la arista y *value* a un 1.

5. Una vez ejecutados todos los *maps*, Hadoop recopila todos los pares $< idArista, 1 >$ y los ordena por *key* (ver paso "Shuffle" en la Figura 4.3). Luego agrupa todos los valores emitidos para cada clave única e invoca una función *reduce*. Cada *reduce* recibe como *key* el identificador de la arista y como *value* un arreglo de valores 1.

6. Dado que la idea es eliminar las aristas repetidas, en este paso, al igual que el Método U1, independiente de la cantidad de unos recibidos, se emite un único

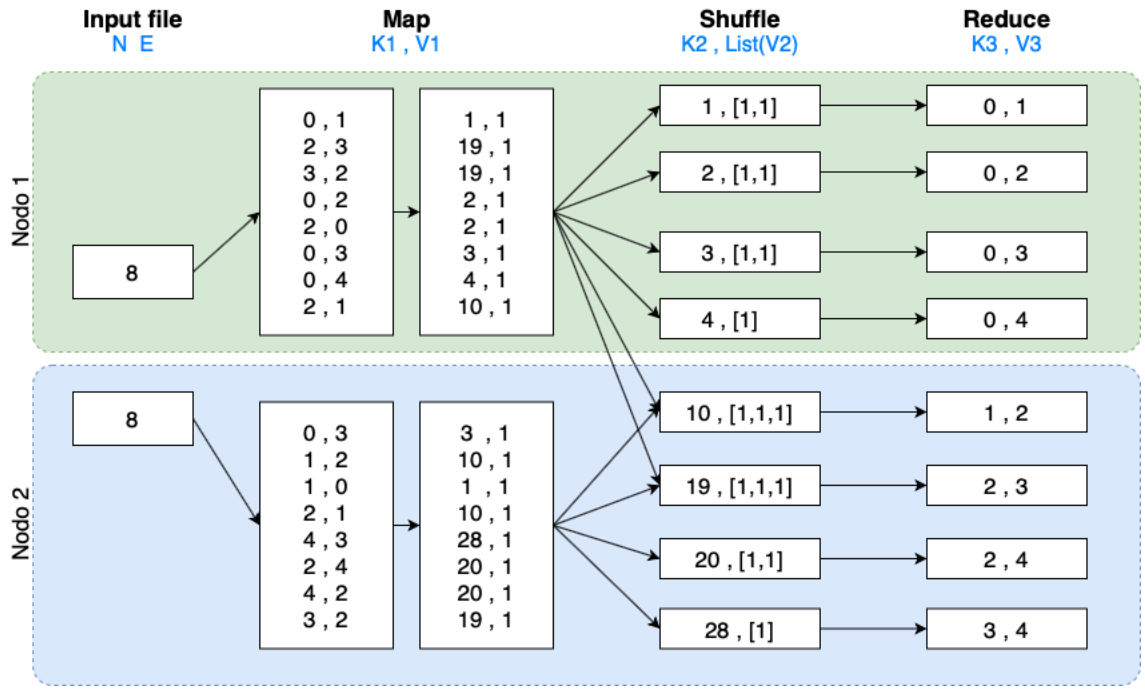


Figura 4.3: Ejemplo del método U2

mensaje $< origen, destino >$, donde *origen* y *destino* se calculan usando las Fórmulas 4.2 y 4.3 respectivamente (ver Algoritmo 15, Líneas 4 y 5).

7. Hadoop escribe la salida final en formato de lista de aristas en el directorio de salida.

En la Figura 4.3 se ilustra el proceso antes descrito. Cada map recibe una cantidad de aristas que debe generar. Por cada arista que se debe generar, se llama a la función que simula el proceso recursivo de R-MAT, de la cual se obtiene una arista. Dicha arista es codificada y posteriormente enviada como mensaje para que continue su procesamiento. Poniendo atención a la Figura 4.3, se puede observar que la arista (2,3) y (3,2) del Nodo 1 (rectángulo verde) y la arista (3,2) del Nodo 2 quedan codificadas de la forma (19,1), donde 19 es el valor que se obtiene de la Fórmula 4.1 y que es interpretado como $< key >$ por Hadoop.

Continuando con el ejemplo de la Figura 4.3, como los tres mensajes (19,1) tienen como *key* 19, las tres aristas son procesadas por el mismo *reduce*, el cual decodifica

una sola vez la arista obteniendo el nodo *origen* y *destino* que se escribe en el archivo final. Notar que en la columna final del ejemplo, la arista (2,3) aparece una sola vez, gracias a que el reduce, independiente de la cantidad de aristas con la misma *key*, escribe sólo una de ellas.

Realizando el mismo experimento que en el caso del método U1, se determina que la cantidad de aristas que le faltan al grafo producido, usando el método D2, es de aproximadamente 0,05 % para grafos de un millón de nodos, mientras que para grafos de diez millones de nodos, la pérdida es de sólo un 0,01 % de aristas.

4.3. Método U3

Los Métodos U1 y U2 se basan en la idea de eliminar las aristas repetidas, por lo tanto un grafo $G = (N, E)$ generado usando esos métodos, tiene una menor cantidad de aristas de lo requerido.

Analizando la forma en que el Método U1 distribuye las aristas, se puede notar que las aristas repetidas se producen cuando: el nodo siguiente a un nodo que tiene aridad suficiente para generar aristas yendo hacia atrás y hacia adelante, genera una arista hacia atrás, tal como se ilustra en la Figura 4.1.

Teniendo en consideración el momento en el que se producen aristas repetidas, U3, al igual que el Método U1, consiste en un proceso compuesto de dos pasos: la construcción del arreglo de aridades y la generación de las aristas, con la principal diferencia de que al generar las aristas se usan métodos que permiten asegurar que se generaran aristas únicas. Dichos métodos se describen en las Secciones 4.3.1, 4.3.2 y 4.3.3.

La implementación del Método U3, independiente del método de generación de aristas que se configure, consiste en un *job* (ver Algoritmo 16), compuesto de un *map* y un *reduce*. Este último, varía dependiendo del método de generación de aristas que se implementa, mientras que el *map* es el mismo para todos los casos. En el Algoritmo 17 se muestra el pseudocódigo de la función *map*, en la cual se construye el arreglo de aridades que luego es procesado por el *reduce*.

En los Algoritmos 18, 19 y 20, se muestra el pseudocódigo del *reduce* implementado para cada uno de los casos. Observe que la única diferencia entre ellos, se encuentra en los métodos invocados para generar las aristas, los cuales se describen a continuación.

Algoritmo 16: U3 - Main

```
   Input: n ← número de nodos del grafo
   nMaps ← número de maps
   Output: Lista de aristas.
1  begin
2  |  long m ← 2/3 · n · ln n + 0,384 81 · n
3  |  GenerateInputFiles(n, m, nMaps, tmpDir)
   |  /* Job 1                                                    */
4  |  job.setMapperClass(UndirectedGraphMapper.class)
5  |  job.setReducerClass(UndirectedGraphReducer.class)
6  |  job.waitForCompletionClass(true)
7  end
```

Algoritmo 17: U3 - UndirectedGraphMapper

```
   Input: n ← número de nodos del grafo
1  begin
2  |  method MAP (LongWritable offset, Text line):
3  |  |  long m' ← Long.valueOf(line.toString())
4  |  |  RMat rmat ← new RMat()
5  |  |  for int i ← 0 to m' − 1 do
6  |  |  |  long edge[] ← rmat.GenerateEdge(n)
7  |  |  |  LongWritable source ← edge[0] −1
8  |  |  |  EMIT(LongWritable source, LongWritable 1)
9  |  |  end
10 end
```

Algoritmo 18: U3_1 - UndirectedGraphReducer

```
1  begin
2  |  method REDUCE (LongWritable source, LongWritable
   |   values[v1, v2 ...]):
3  |  |  long degree ← 0
4  |  |  forall v ∈ values[v1, v2 ...] do
5  |  |  |  degree ← degree + v
6  |  |  end
7  |  |  long i ← 0
8  |  |  i ← WriteEdgeGoingBack(i, degree, source)
9  |  |  i ← WriteEdgeGoingFurtherForward1(i, degree, source, n)
10 end
```

Algoritmo 19: U3_2 - UndirectedGraphReducer

```
1  begin
2  |  method REDUCE (LongWritable source, LongWritable
   |   values[v1, v2 ...]):
3  |  |  long degree ← 0
4  |  |  forall v ∈ values[v1, v2 ...] do
5  |  |  |  degree ← degree + v
6  |  |  end
7  |  |  long i ← 0
8  |  |  i ← WriteEdgeGoingBack(i, degree, source)
9  |  |  i ← WriteEdgeGoingFurtherForward2(i, degree, source, n)
10 end
```

Algoritmo 20: U3_3 - UndirectedGraphReducer

```
1  begin
2  |  method REDUCE (LongWritable source, LongWritable
   |   values[v1, v2 ...]):
3  |  |  long degree ← 0
4  |  |  forall v ∈ values[v1, v2 ...] do
5  |  |  |  degree ← degree + v
6  |  |  end
7  |  |  long i ← 0
8  |  |  WriteEdgeAlwaysGoingForward(i, degree, source)
9  end
```

4.3.1. Método 1 para generar aristas no dirigidas

El primer método consiste en:

1. Generar aristas yendo desde el índice del nodo origen hacia atrás, usando la función $WriteEdgeGoingBack$, descrita en la Sección 3.1.1.

2. Si al llegar al primer elemento del "arreglo" de aridades, la aridad del nodo procesado aún es mayor a cero, entonces se generan aristas yendo hacia adelante, seleccionando como nodos *destino* aquellos cuyo índice se encuentra después del menor valor entre $4 \times degree$ y $\frac{(n-degree)}{2}$, donde $degree$ es la aridad del nodo que se está procesando, n la cantidad total de nodos del grafo y 4 una constante, cuyo valor se determinó de manera empírica, considerando la aridad máxima de un grafo.

En el Algoritmo 21 se muestra la implementación de esta función, llamada $WriteEdgeGoingFurtherForward1$. Notar que en la Línea 11 del pseudocódigo, se suma al índice del nodo destino el salto determinado en base a las fórmulas antes mencionadas, y no un 1 como el caso de la función $WriteEdgeGoingForward$ de la Sección 3.1.1.

Algoritmo 21: WriteEdgeGoingFurtherForward1 (long i, long *degree*, long *source*, long n)

Input: i define un contador. *degree* define la cantidad de aristas a generar a partir del nodo de origen definido por *source*.

```
1  begin
2  |  long j1 ← 4 · degree
3  |  long j2 ← (n − degree)/2
4  |  long j ← 0
5  |  if j1 < j2 then
6  |  |  m ← j1
7  |  end
8  |  else
9  |  |  j ← j2
10 |  end
11 |  long target ← source + j
12 |  while i < degree ∧ target < n do
13 |  |  EMIT(LongWritable source, LongWritable target)
14 |  |  i ← i + 1
15 |  |  target ← target + 1
16 |  end
17 |  return i
18 end
```

4.3.2. Método 2 para generar aristas no dirigidas

En términos generales, el segundo método consiste en: primero, generar aristas recorriendo en reversa el arreglo de aridades y una vez que se llega al primer elemento del arreglo, generar aristas seleccionando como destino, aquellos nodos que se encuentran ubicados desde el índice $\frac{n}{2} + i$ en adelante, donde n es el número total de nodos e i el índice del nodo origen.

Este método permite generar aristas hacia adelante del nodo origen, cuyo destino esté lo suficientemente lejos para que al momento en que le toque a ese nodo destino generar sus aristas hacia atrás, estas nunca se hayan producido antes.

Es preciso mencionar que este método se basa en: identificar que sólo una pequeña parte de los nodos, correspondiente a los primeros índices del arreglo, producen aristas hacia adelante, y que la aridad máxima del grafo no supera $\frac{n}{2}$. Empíricamente, se calculó la aridad máxima de grafos de varios tamaños. En la Tabla 4.1 se muestra un resumen de los resultados de dicho experimento, en los cuales se puede ver que, para grafos de 1 000 nodos o más, la aridad máxima no supera $\frac{n}{2}$.

Tabla 4.1: Aridad máxima según el tamaño del grafo y su valor para $n/2$.

Grafo **ID**	Número de nodos (n)	Aridad máxima	$n/2$
G100	$1 \cdot 10^2$	66	50
G500	$5 \cdot 10^2$	255	250
G1k	$1 \cdot 10^3$	336	500
G10k	$10 \cdot 10^3$	1 038	5 000
G100k	$100 \cdot 10^3$	4 767	50 000
G1M	$1 \cdot 10^6$	20 378	500 000
G5M	$5 \cdot 10^6$	38 644	2 500 000
G10M	$10 \cdot 10^6$	57 070	5 000 000

En la Figura 4.4 se ilustra el funcionamiento de este método, tomando como ejemplo el Nodo 2 y el Nodo 3, los cuales se asume, generan sus aristas de forma independiente el uno del otro. El Nodo 2, cuando ya no tiene posibilidad de generar aristas hacia atrás en el arreglo, es decir, llega al Nodo 0, empieza a generar aristas hacia adelante, pero no al nodo que le sigue en el arreglo, si no que al nodo 7, que corresponde a $\frac{n}{2} + i$, donde, $n = 10$ e $i = 2$. Este "salto", garantiza que, cuando el nodo 3 y los demás, generen sus aristas, nunca se produzca una arista repetida.

La implementación de este método se muestra en el Algoritmo 22. Notar que en las Líneas 2 y 3 se calcula el valor del índice del nodo a partir del cual se generan aristas.

En la validación, se comprueba que el Método 2 permite generar un grafo con $m = \left\lceil \frac{2}{3} n \ln n + 0{,}384\,81 n \right\rceil$ aristas, sin aristas repetidas y además, sin necesidad de conocer información de los otros nodos.

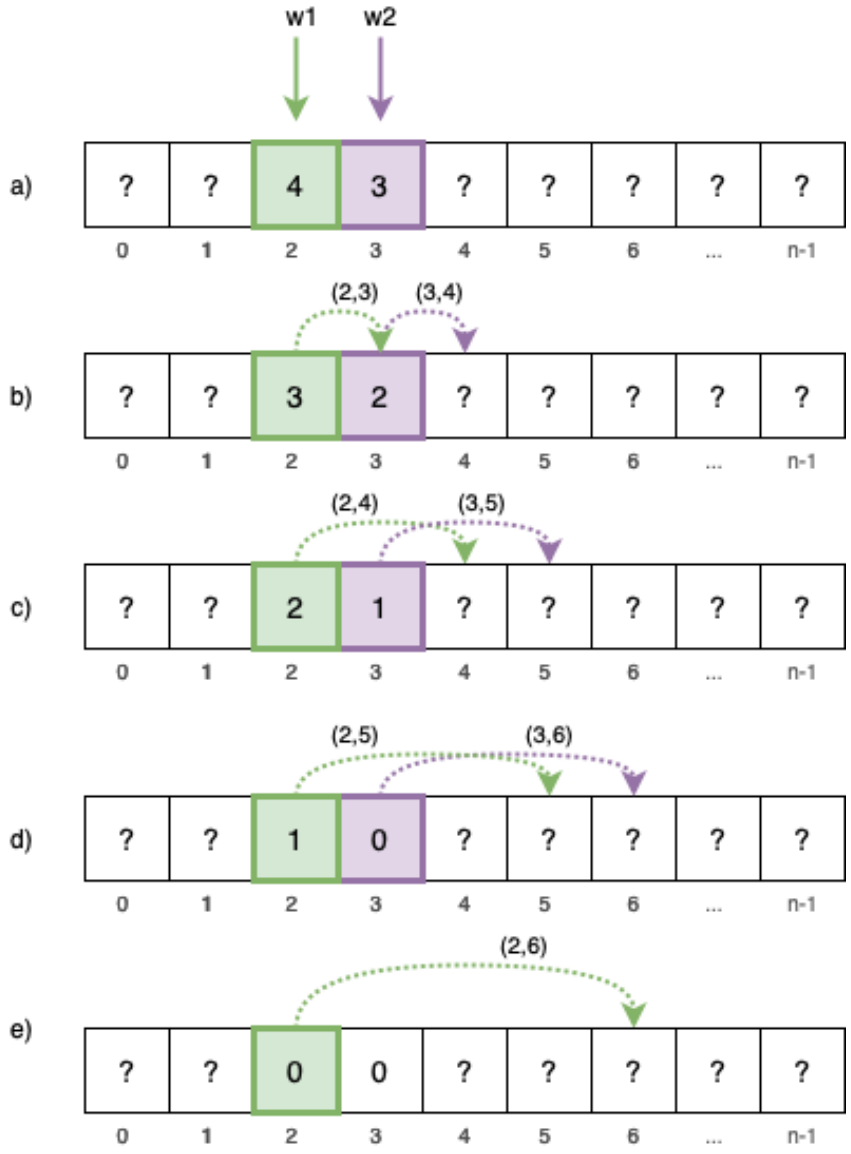


Figura 4.4: Ejemplo: Generación de aristas usando el método 2

4.3.3. Método 3 para generar aristas no dirigidas

Este tercer método se basa en una idea más simple que las anteriores, pero igual de efectiva en términos de cumplir con la cantidad de aristas que se quiere generar para el grafo, evitando aristas repetidas.

A grandes rasgos, U3 consiste en generar aristas desde un nodo origen consideran-

Algoritmo 22: WriteEdgeGoingFurtherForward2 (long i, long *degree*, long *source*, long n)

Input: i define un contador. *degree* define la cantidad de aristas a generar a partir del nodo de origen definido por *source*.

```
1  begin
2  |  long j ← n/2
3  |  long target ← source + j
4  |  while i < degree ∧ target < n do
5  |  |  EMIT(LongWritable source, LongWritable target)
6  |  |  i ← i + 1
7  |  |  target ← target + 1
8  |  end
9  |  return i
10 end
```

do como destino los nodos cuyos índices inician en *origen*+1, desde donde se continúa generando aristas yendo hacia adelante, hasta que la cantidad de aristas generadas es igual al *degree* del nodo *origen*. En otras palabras, las aristas generadas por este método son de la forma $e_{i+1} = (i, i+1)$, $e_{i+2} = (i, i+2)$,..., $e_{i+D[i]} = (i, i+D[i])$, donde i corresponde al índice del nodo origen y $D[i]$ a la aridad del nodo i. En la Figura 4.5 se muestra un ejemplo de este método.

En el Algoritmo 23 se muestra el pseudocódigo correspondiente a la implementación del método. El primer paso consiste en inicializar el contador de aristas generadas en cero y calcular el nodo destino. Luego, mientras las aristas producidas sean menores a la aridad del nodo que se está procesando, se escriben las aristas en el archivo de salida. En caso de que los últimos nodos tengan aridad suficiente para generar una arista, cuyo destino sea mayor a la cantidad de nodos del grafo, se establece como tal el nodo 0, indicando que se continúa el proceso desde el inicio del arreglo (ver Algoritmo 23, desde la Línea 5 a la 7).

4.4. Método U4

A partir del Método U2, surge una idea que permite generar un grafo sin perder aristas durante su proceso de generación. Esta idea consiste en cuatro pasos, los cuales se describen a continuación:

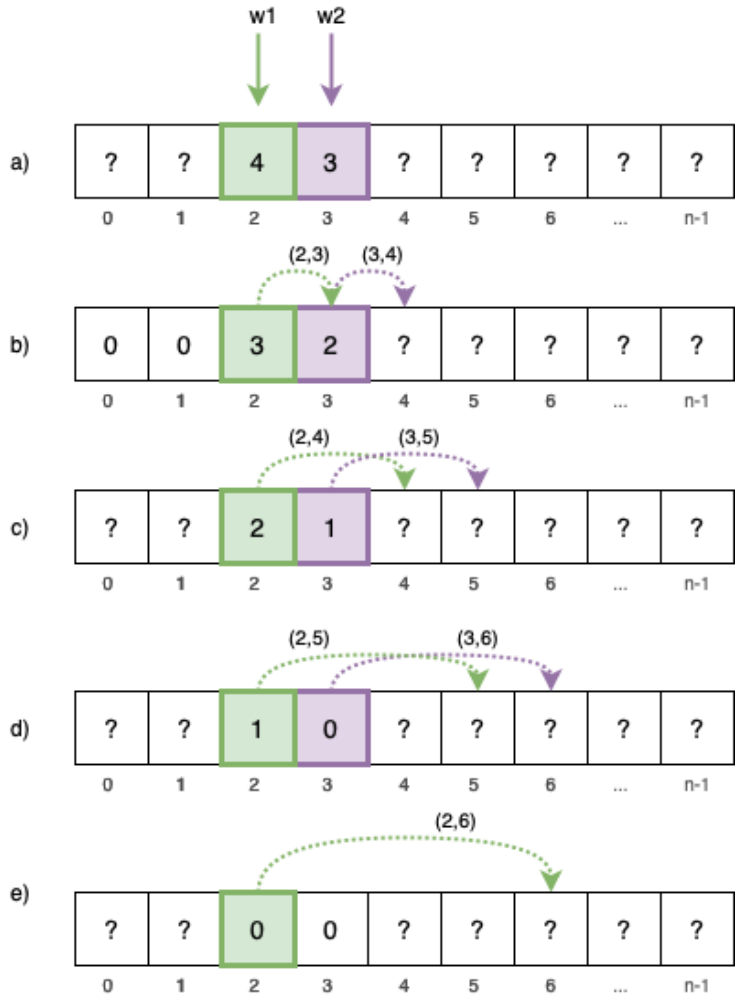


Figura 4.5: Ejemplo: Generación de aristas usando el Método 3.

1. Generar la cantidad solicitada de aristas, incluyendo repetidas.
2. Contar y, a la vez, eliminar las aristas repetidas.
3. Calcular el índice del nodo destino mínimo y máximo para cada nodo.
4. Generar la cantidad de aristas repetidas a partir del índice del nodo mínimo hacia atrás, hasta llegar al nodo 0, y en caso de que el nodo aún tenga aristas que generar, continúa del índice máximo hacia adelante.

La implementación de esta idea, se hace configurando cuatro *jobs* de Hadoop. El

Algoritmo 23: WriteEdgeAlwaysGoingForward (long *degree*, long *source*, long n)

Input: *degree* define la cantidad de aristas a generar a partir del nodo de origen definido por *source*. n corresponde a la cantidad de nodos del grafo

```
1  begin
2      long i ← 0
3      long target ← source + 1
4      while i < degree do
5          if target ≥ n then
6              target ← 0
7          end
8          EMIT(LongWritable source, LongWritable target)
9          i ← i + 1
10         target ← target + 1
11     end
12 end
```

primero construye un grafo con aristas faltantes. El segundo, realiza el cálculo del índice mínimo y máximo de cada nodo. El tercer *job*, genera las aristas faltantes a partir del índice mínimo y máximo en un archivo separado. Y por último, el cuarto *job*, junta el grafo con aristas faltantes producido por el primer *job* con las aristas producidas en el tercero.

Evaluación del método

En los experimentos exploratorios realizados, se observa que este método tiene un costo considerable en términos de tiempo de ejecución. Aproximadamente, el Método U4 tarda 4 veces más que el Método U3, razón por la cual, en adelante, no sigue siendo analizado.

5. Evaluación Experimental

En este capítulo se presenta la evaluación de los métodos descritos en este trabajo a través de la ejecución de un conjunto de experimentos. Primero se describe la metodología utilizada para la evaluación, luego se describen las características del entorno de pruebas y finalmente se presentan y analizan los resultados obtenidos.

5.1. Metodología de Evaluación

Teniendo en cuenta la metodología de trabajo, se ejecutan una serie de experimentos con el fin de establecer un punto de comparación entre las soluciones secuenciales y distribuidas existentes para la generación de grafos que siguen ley de potencia.

Los experimentos que se realizan consisten en una combinación de variables que apuntan a medir la eficiencia, escalabilidad y realismo de cada uno de los métodos de generación de grafos propuestos. D1 y D2 para grafos dirigidos; U1, U2 y U3 para grafos no dirigidos.

Pruebas de eficiencia

Con la finalidad de medir la eficiencia de los métodos de generación, se ejecutan experimentos que consisten en variar la cantidad de nodos del grafo generado. Para dicha variable se consideran grafos con 1 000, 10 000, 100 000, 1 millón, 10 millones, 100 millones y 500 millones de nodos.

La eficiencia de un método se mide en términos del tiempo de ejecución para generar un grafo (dirigido o no dirigido) de un tamaño específico.

El objetivo de este experimento es determinar el método más rápido y el más lento para cada tipo de grafo.

Pruebas de escalabilidad

Por su parte, en lo que concierne a escalabilidad, cada uno de los métodos es evaluado bajo dos puntos de vista:

1. Medir la capacidad de los métodos para soportar un número creciente de nodos, con el fin de determinar el grafo más grande que se puede generar.

2. Analizar la escalabilidad respecto de los recursos computacionales, con el objeto de determinar la dependencia de un método respecto del *hardware* y también determinar el punto en el que usar un algoritmo distribuido es mejor que el método R^{3}MAT.

Considerando las dos ideas anteriores, se ejecutan experimentos variando la cantidad de máquinas que conforman el *cluster*. En particular, se consideran las siguientes configuraciones:

1. 1 nodo principal, 4 nodos trabajadores.

2. 1 nodo principal, 8 nodos trabajadores.

3. 1 nodo principal, 12 nodos trabajadores.

4. 1 nodo principal, 16 nodos trabajadores.

Pruebas de realismo

En el contexto de este trabajo, el realismo consiste en la capacidad de un método para generar un grafo cuya distribución siga la ley de potencia. Para verificar el realismo de un método se usa la estadística de Kolmogorov-Smirnov (KS), así como los gráficos que muestran la distribución de aridad de un grafo generado.

Para calcular el estadístico KS, se utiliza un *script* en R [5].

Comparación con otros métodos

Se realizan una serie de experimentos con el fin de establecer una comparación en términos de rendimiento y realismo entre las soluciones propuestas en este trabajo y las herramientas o algoritmos existentes. Para ello, se consideran los métodos propuestos versus R^{3}MAT en caso de algoritmos secuenciales. Por su parte, se considera TeGViz y PegasusN en caso de soluciones que realizan procesamiento en paralelo.

Tabla 5.1: Configuración de los *clusters* usados en la evaluación experimental.

Cluster ID	**Nodo principal (cantidad)**	**Nodos trabajadores (cantidad)**	**vCPU total (cantidad)**	**RAM total (GB)**
C4	1	4	4	20
C8	1	8	8	36
C12	1	12	12	52
C16	1	16	16	68

5.2. Entorno de Pruebas

Todos los experimentos fueron ejecutados en Google Cloud Platform usando el servicio Dataproc. Este es un servicio administrado de Spark y Hadoop con el que se pueden crear *clusters* rápidamente y administrarlos con facilidad.

Dataproc permite crear *clusters* compuestos por diferentes tipos de máquinas virtuales. Para este trabajo en particular, todos los *clusters* de prueba se construyeron usando máquinas virtuales `n1-standard-1` con 1 CPU virtuales y 4 GB de memoria RAM. El sistema operativo de todas las máquinas es Debian GNU/Linux 10 (buster). La versión de Hadoop instalada es 2.9.2 y Java openjdk versión 1.8.0_252.

En la Tabla 5.1 se muestra un resumen de las características para cada configuración del *cluster* en donde se ejecutan las pruebas.

Los algoritmos son implementados en lenguaje Java, cuyos proyectos se llaman `D1, D2` para grafos dirigidos. `U1, U2 y U3` para grafos no dirigidos. El código fuente se encuentra en GitHub (específicamente en `https://github.com/FerLopezGallegos/HadoopGraphGenerator.git`). Además, todos los proyectos proveen un archivo `jar` que permite generar los grafos desde la línea de comandos del nodo principal del *cluster* ejecutando una instrucción de la siguiente forma según el proyecto:

- Para grafos dirigidos:

  ```
  hadoop jar D1.jar | D2.jar <N><nMaps>
  ```

  donde `<N>` es el número de nodos que se quieren generar y `<nMaps>` el número de archivos en los que se divide la cantidad total de aristas, lo cual está estrechamente relacionado con el número de *maps* que se van a distribuir. Como recomendación `<nMaps>` debe ser igual al número de total de vCPUs de los nodos trabajadores para que todas las máquinas ejecuten al menos un *map*.

Tabla 5.2: Grafos dirigidos generados en la evaluación experimental.

Grafo ID	Número de nodos	Número de aristas	
		D1	D2
DG1k	$1 \cdot 10^3$	4 989	4 989
DG10k	$10 \cdot 10^3$	65 250	65 250
DG100k	$100 \cdot 10^3$	806 009	806 009
DG1M	$1 \cdot 10^6$	9 595 150	9 595 150
DG5M	$5 \cdot 10^6$	53 340 544	53 340 544
DG10M	$10 \cdot 10^6$	111 302 071	111 302 071
DG50M	$50 \cdot 10^6$	610 158 285	-
DG100M	$100 \cdot 10^6$	1 266 526 382	-
DG500M	$500 \cdot 10^6$	6 869 111 218	-

- Para grafos no dirigidos:

  `hadoop jar U1.jar | U2.jar | U3_1.jar | U3_2.jar | U3_3.jar <N><nMaps>`

  donde `<N>` es el número de nodos que se quieren generar y `<nMaps>` el número de archivos en los que se divide la cantidad total de aristas, lo cual está estrechamente relacionado con el número de *maps* que se procesaran por el *cluster*.

5.3. Resultados

Luego de la ejecución de los experimentos descritos en la Sección 5.1, se obtienen los siguientes resultados.

En las Tablas 5.2 y 5.3 se muestra un resumen de los grafos generados. La primera contiene un resumen de los grafos dirigidos, mientras que la segunda muestra los no dirigidos. En términos generales, por cada tipo de grafo (dirigido, no dirigido), se generaron nueve experimentos variando el tamaño del grafo, llegando a un tamaño $n = 500 \cdot 10^6$, es decir, quinientos millones de nodos, lo que implica $6{,}86 \cdot 10^9$ aristas aproximadamente.

A continuación, en la Sección 5.3.1 se describen los resultados obtenidos en términos de eficiencia, escalabilidad y realismo para grafos dirigidos. Lo mismo se hace para grafos no dirigidos en la Sección 5.3.2.

Tabla 5.3: Grafos no dirigidos generados en la evaluación experimental.

Grafo ID	Número de nodos	Número de aristas		
		U1	U2	U3_1, U3_2 y U3_3
UG1k	$1 \cdot 10^3$	4 354	4 253	4 989
UG10k	$10 \cdot 10^3$	59 741	59 811	65 250
UG100k	$100 \cdot 10^3$	757 327	777 829	806 009
UG1M	$1 \cdot 10^6$	9 180 370	9 477 900	9 595 150
UG5M	$5 \cdot 10^6$	51 640 025	52 929 354	53 340 544
UG10M	$10 \cdot 10^6$	108 146 410	110 661 701	111 302 071
UG50M	$50 \cdot 10^6$	597 088 219	608 424 577	610 158 285
UG100M	$100 \cdot 10^6$	1 242 457 175	1 263 872 199	1 266 526 382
UG500M	$500 \cdot 10^6$	-	6 863 205 927	6 869 111 218

5.3.1. Grafos dirigidos

En esta sección se describen los resultados de la ejecución de los experimentos para generar grafos dirigidos con los métodos D1 y D2.

Eficiencia

Las Tablas 5.4 y 5.5 muestran los tiempos de ejecución de los métodos de generación para grafos dirigidos, D1 y D2 respectivamente. Cuando aparece un $-$, indica que el grafo no se generó luego de 120 minutos de ejecución y, por ende, se detuvo el experimento. El símbolo $*$ indica que el generador terminó su ejecución, pero se detectó uno o más problemas con el resultado obtenido.

Tabla 5.4: Tiempo de ejecución del Método D1.

Grafo ID	Tiempo de ejecución (ms)			
	C04	C08	C12	C16
DG1k	36 426	38 865	34 976	37 085
DG10k	36 901	35 647	34 901	36 442
DG100k	39 519	38 699	36 241	37 022
DG1M	58 680	47 540	44 882	42 234
DG5M	182 092	93 390	82 381	70 320
DG10M	321 585	170 159	126 882	104 858
DG50M	1 608 277	798 235	538 561	419 825
DG100M	3 860 453	1 756 491	1 139 907	853 337
DG500M	-	-	6 839 663	5 117 718

Tabla 5.5: Tiempo de ejecución del método D2.

Grafo ID	**Tiempo de ejecución (ms)**			
	C04	**C08**	**C12**	**C16**
DG1k	37 640	37 292	34 797	34 463
DG10k	37 226	37 291	33 483	33 566
DG100k	37 754	37 736	35 575	36 431
DG1M	57 867	47 395	41 769	39 624
DG5M	179 346	83 251	67 648	61 167
DG10M	*	*	101 727	86 722

El menor tiempo de ejecución, $t = 33\,483$ (ms), corresponde al Método D2, cuando genera un grafo de 10 000 nodos, en un *cluster* de 12 trabajadores (C12). Por otro lado, el mayor tiempo $t = 6\,839\,663$ (ms) o 114 minutos, es obtenido por el Método D1 al generar un grafo de 500 millones de nodos, en un *cluster* de 12 trabajadores (C12).

A fin de comparar la eficiencia de los Métodos D1 y D2, se incluye la Figura 5.1. Esta figura muestra los tiempos de ejecución de ambos métodos al generar grafos de diferentes tamaños (de 1 000 hasta 500 millones de nodos), usando el *cluster* de 16 trabajadores, el cual, es el más poderoso utilizado en la evaluación. Para grafos con menos de 1 millón de nodos, no se nota diferencia de tiempo entre ambos métodos. Ya a partir de los 5 millones de nodos, se ve que el Método D2 genera unos segundos más rápido el grafo, sin embargo, este no es capaz de generar grafos correctos con más de 50 millones de nodos en el *cluster* de 16 trabajadores. Por su parte, el Método D1 demuestra soportar la generación de un grafo correcto de 500 millones de nodos, en aproximadamente 85 minutos.

Por otro lado, como se ve, para grafos con menos de 5 millones de nodos, el tiempo de ejecución no varía con la cantidad de nodos que se quiere producir (ver Figura 5.1). Esto último, permite deducir que en la generación de grafos pequeños tiene más influencia el tiempo de coordinación de Hadoop que la producción de las aristas.

Escalabilidad

La escalabilidad de los métodos es evaluada bajo dos puntos de vista: (*i*) con respecto al tamaño del grafo; y (*ii*) con respecto a la cantidad de trabajadores del

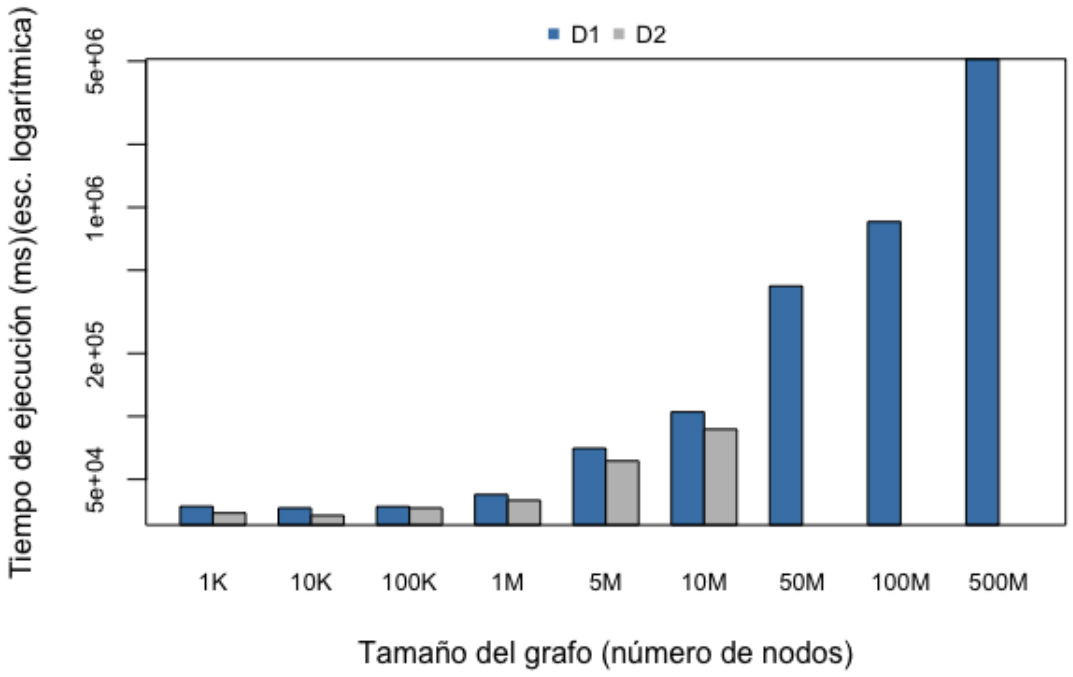


Figura 5.1: Tiempo de ejecución de los métodos para generar grafos dirigidos de diferentes tamaños, obtenidos en el *cluster* de 16 nodos trabajadores.

cluster.

(i) Escalabilidad respecto al tamaño del grafo.

En el gráfico de la Figura 5.1, se puede ver que el Método D1 permite generar grafos de 500 millones de nodos, pudiendo incluso, generar grafos más grandes si se está dispuesto a esperar lo suficiente, ya que, sin considerar el tiempo, no hay otro factor que impida seguir aumentando la cantidad de nodos del grafo.

Respecto al Método D2, no se registran resultados para grafos con más de 10 millones de nodos, ya que, a pesar de terminar la ejecución, el grafo producido no es correcto, puesto que tiene un número importante de aristas que no son escritas en el archivo de salida. Este problema, se asocia al uso de la función *Combiner*, la cual, a pesar de ser descrita en la documentación como una función que permite optimizar el envió de mensajes entre las máquinas del *cluster* Hadoop, ocupa recursos que

se ven superados por la cantidad de aristas, provocando que el grafo generado sea incompleto. Una forma de solucionar este problema, es ejecutar el método en un *cluster* con más recursos, tal como se observa con el grafo de 10 millones de nodos en la Tabla 5.5, el cual no pudo ser generado en el *cluster* de 4 y 8 trabajadores, pero sí pudo ser generado en el *cluster* de 12 y 16 trabajadores.

(*ii*) Escalabilidad respecto al *cluster*

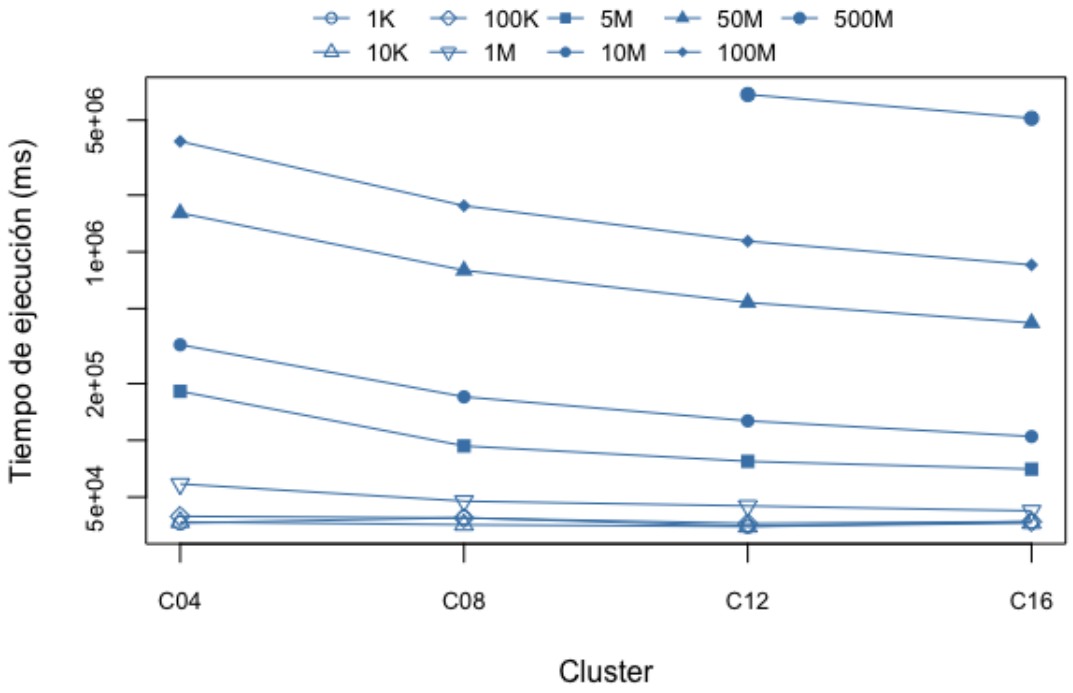


Figura 5.2: Tiempo de ejecución del método D1 para generar grafos dirigidos de diferentes tamaños, en diferentes *clusters*.

En cuanto a la escalabilidad del método en relación a la cantidad de máquinas configuradas como trabajadores del *cluster*, en la Tabla 5.4, se puede ver que para grafos pequeños (tamaño del grafo menor a 1 millón de nodos), no hay variación significativa del tiempo de ejecución a medida que se aumenta la cantidad de máquinas del *cluster*. Para grafos más grandes, en las Figuras 5.2 y 5.3 correspondientes a los métodos D1 y D2 respectivamente, se puede ver que a medida que se aumenta

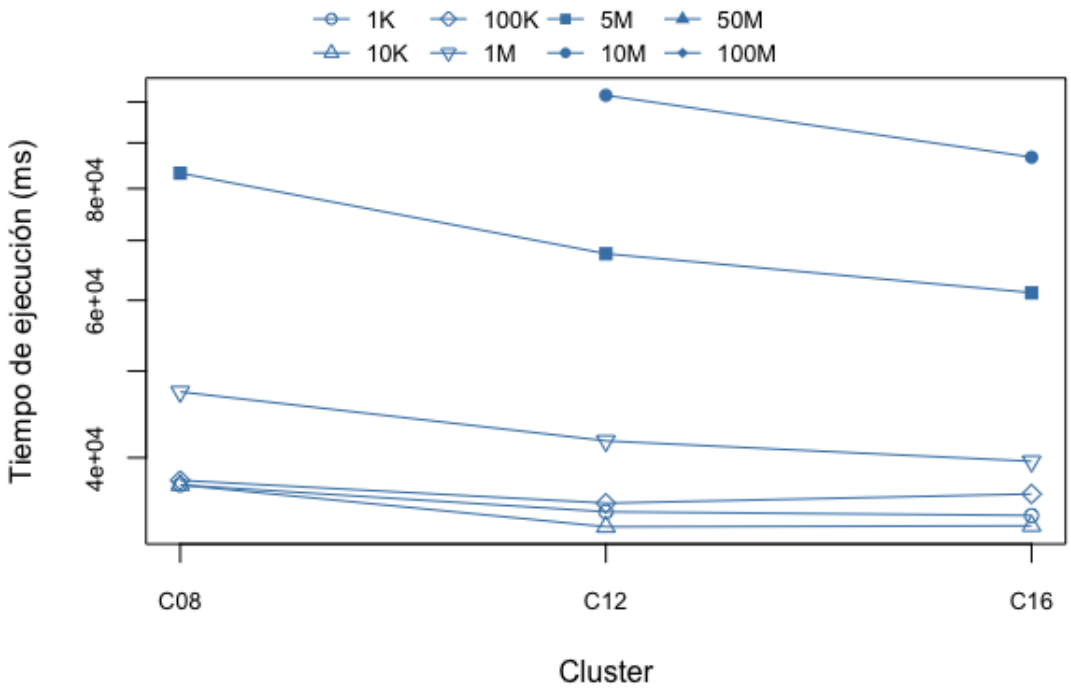


Figura 5.3: Tiempo de ejecución del método D2 para generar grafos dirigidos de diferentes tamaños, en diferentes *clusters*.

la cantidad de trabajadores, se obtiene una ganancia significativa en términos de tiempo de ejecución. Por ejemplo, generar un grafo de 100 millones de nodos, en un *cluster* de 12 trabajadores, toma 19 minutos, mientras que, el mismo grafo en un *cluster* de 16 máquinas toma 14 minutos.

Además, cabe destacar que al aumentar la cantidad de trabajadores del *cluster*, con el método D1, se pueden generar grafos aún más grandes. En la Figura 5.2, se puede ver que en un *cluster* de 12 trabajadores es posible generar un grafo de 500 millones de nodos en un tiempo razonable, el cual disminuye de manera significativa al aumentar la cantidad de trabajadores.

Consolidando ambos análisis de escalabilidad, se obtiene la conclusión de que para grafos pequeños ($n < 1\,000\,000$), no hay una dependencia fuerte respecto al poder de procesamiento, contrario a lo que sucede con grafos más grandes, en los que aumentar la cantidad de nodos tiene dos beneficios claros: (i) generar grafos grandes

Tabla 5.6: KS para los grafos generados usando los métodos D1 y D2.

Método / Tamaño	**1k**	**10k**	**100k**	**1M**	**5M**	**10M**
D1	0.1191	0.0665	0.0378	0.0340	0.0301	0.0251
D2	0.1175	0.0996	0.0511	0.0299	0.0218	0.0167

más rápido; y (*ii*) aumentar el tamaño del grafo, sin restricciones de memoria RAM y/o espacio de almacenamiento, debido a las características intrínsecas de Hadoop en términos de escalabilidad.

Realismo

En términos de realismo, recordar que el objetivo es verificar que los métodos generan grafos que sigan la ley de potencia.

En la Tabla 5.6 se muestran los resultados obtenidos del cálculo del estadístico Kolmogorov-Smirnov (KS) para grafos con los siguientes tamaños (número de nodos): $n_1 = 1k$, $n_2 = 10k$, $n_3 = 100k$, $n_4 = 1M$ y $n_5 = 10M$. Considerando que un KS menor a 0,15 es una buena medida de ajuste a la ley de potencia, se puede decir que los Métodos D1 y D2 generan grafos que se ajustan muy bien a una ley de potencia. Además, se puede notar que a medida que aumenta el tamaño del grafo, el KS es cada vez menor, lo que permite conjeturar que grafos con más de 10 millones de nodos, también se ajustan muy bien a una ley de potencia.

Visualmente, se puede comprobar que los grafos generados usando los Métodos D1 y D2 siguen la ley de potencia. En la Figura 5.4 se muestran los gráficos para el más pequeño ($n = 1k$) y el más grande ($n = 10M$) de los grafos generados en este experimento, tanto para el Método D1 como para el D2.

Por otra parte, destacar que los Métodos D1 y D2 (asumiendo que se ejecutan sobre un *cluster* con los recursos necesarios según el tamaño del grafo), generan la cantidad de aristas total que se define al inicio y no produce aristas repetidas.

5.3.2. Grafos no dirigidos

En esta sección se describen los resultados obtenidos de la ejecución de los experimentos descritos en la metodología de evaluación, sobre los Métodos U1, U2, U3_1, U3_2 y U3_3, donde los últimos tres, corresponden a las implementaciones de las diferentes formas de distribuir las aristas del Método U3, descritas en la Sección 4.3.

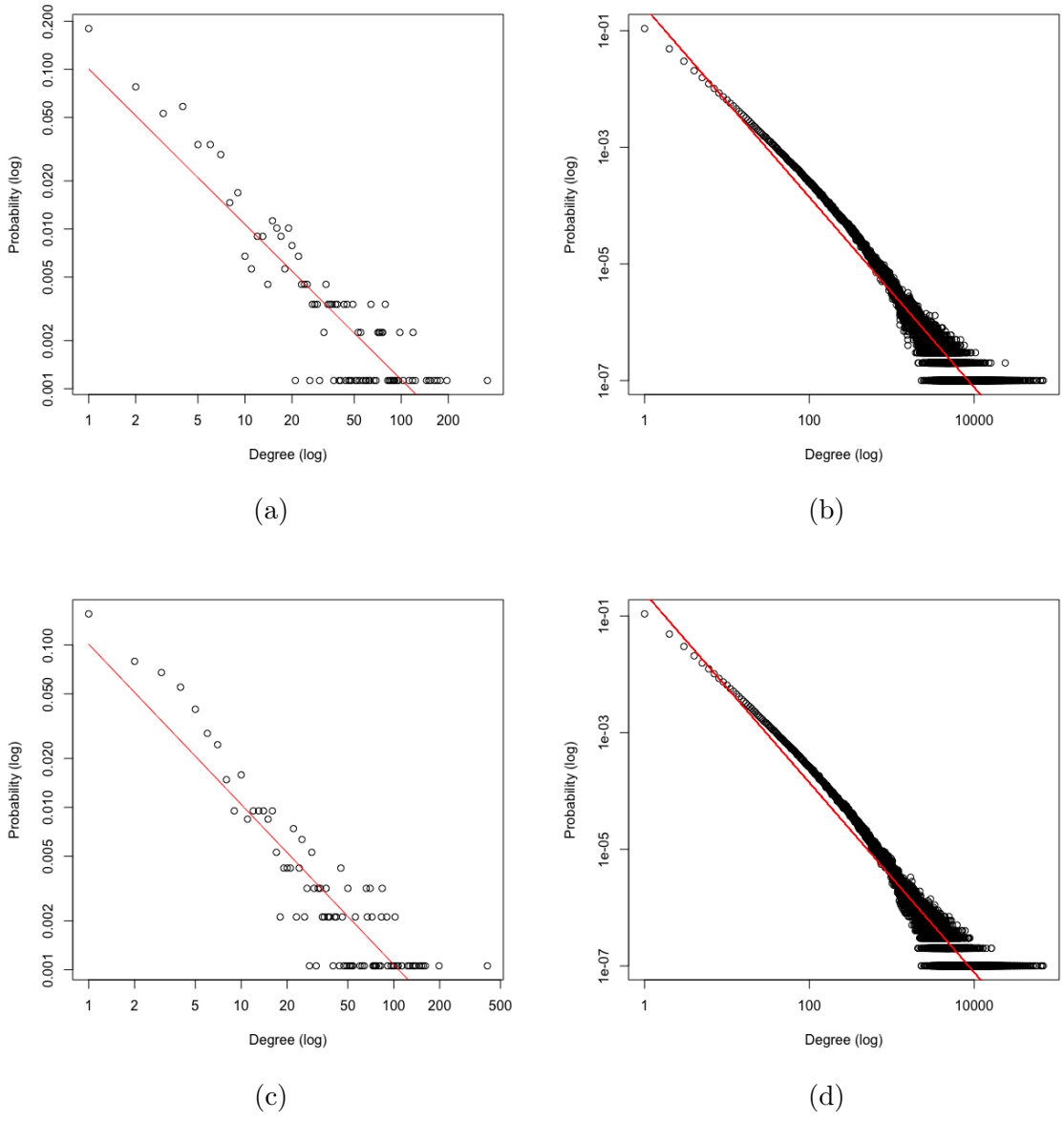


Figura 5.4: Distribución de la aridad, en escala logarítmica, de los grafos dirigidos, generados con los Métodos D1 y D2. Los gráficos (a) y (b) muestran la distribución de aridad para grafos de tamaños $n = 1k$ y $n = 10M$ respectivamente, generados usando el Método D1. Los gráficos (c) y (d) muestran la distribución de la aridad para grafos con los mismos tamaños, generados usando el Método D2.

De la evaluación de estos métodos para generar grafos no dirigidos, cabe mencionar que, a diferencia del análisis de los métodos para grafos dirigidos, se muestran los resultados obtenidos en los *clusters* C08, C12 y C16, dejando fuera el *cluster* C04, debido a que, a pesar de que se completa la ejecución y se obtienen grafos correctos, el tiempo que tardan en ejecutarse los experimentos, hace irrelevante esos resultados para su comparación.

Eficiencia

En las Tablas 5.7 y 5.8, se muestra el tiempo de ejecución de los Métodos U1 y U2, los cuales, generan grafos con menos aristas que las calculadas al inicio, dado que eliminan las aristas repetidas del grafo no dirigido generado. De la misma forma, en las Tablas 5.9, 5.10 y 5.11 se muestran los tiempos de ejecución de las tres variantes del Método U3, el cual es más robusto en el sentido de que permite generar grafos no dirigidos sin perder aristas.

Tabla 5.7: Tiempo de ejecución del método U1.

Grafo ID	**Tiempo de ejecución (ms)**		
	C08	**C12**	**C16**
UG1k	67 023	66 040	65 890
UG10k	66 276	65 279	67 363
UG100k	74 775	70 789	69 605
UG1M	94 720	85 100	84 692
UG5M	219 913	151 714	138 175
UG10M	337 676	252 367	223 591
UG50M	1 608 757	977 013	798 871
UG100M	3 130 948	1 942 595	1 546 336

De los tiempos medidos, se observa que el menor tiempo de ejecución ($t = 33\,303$ milisegundos), corresponde a la generación de un grafo no dirigido, de tamaño $n = 10\,000$, en un *cluster* de 16 trabajadores, usando el método U3_1, el cual, es uno de los métodos que permite generar grafos sin perder aristas, distribuyendo las aristas primero, yendo hacia atrás, y luego yendo desde $N/2$ hacia adelante en el arreglo de aridades.

De forma contraria, el mayor tiempo registrado es 6 982 334 milisegundos (116 minutos), el cual corresponde a la generación de un grafo no dirigido, de 500 millones de nodos, usando el método U2 en un *cluster* de 12 trabajadores. Se debe tener

Tabla 5.8: Tiempo de ejecución del método U2.

Grafo ID	**Tiempo de ejecución (ms)**		
	C08	**C12**	**C16**
UG1k	34 863	34 349	34 282
UG10k	34 690	33 912	35 018
UG100k	37 336	36 672	36 565
UG1M	47 225	44 412	42 679
UG5M	106 210	80 130	69 759
UG10M	179 653	131 980	111 329
UG50M	900 049	605 863	442 120
UG100M	1 798 820	1 210 816	902 499
UG500M	-	6 982 334	5 236 744

Tabla 5.9: Tiempo de ejecución del método U3_1.

Grafo ID	**Tiempo de ejecución (ms)**		
	C08	**C12**	**C16**
UG1k	34 136	34 385	35 422
UG10k	34 349	34 790	33 303
UG100k	36 376	36 918	35 730
UG1M	46 406	47 374	41 795
UG5M	95 088	74 674	65 585
UG10M	170 217	124 509	102 823
UG50M	836 725	553 101	411 528
UG100M	1 720 349	1 137 375	853 473
UG500M	-	6 803 615	5 075 676

Tabla 5.10: Tiempo de ejecución del método U3_2.

Grafo ID	**Tiempo de ejecución (ms)**		
	C08	**C12**	**C16**
UG1k	34 197	34 491	35 831
UG10k	34 362	33 726	37 608
UG100k	37 674	36 069	36 134
UG1M	46 841	43 838	42 855
UG5M	94 491	78 596	65 775
UG10M	166 313	118 777	104 708
UG50M	817 886	543 997	410 173
UG100M	1 655 236	1 125 508	889 743
UG500M	-	6 682 618	4 894 180

Tabla 5.11: Tiempo de ejecución del método U3_3.

Grafo ID	Tiempo de ejecución (ms)		
	C08	**C12**	**C16**
UG1k	34 149	34 023	36 224
UG10k	34 607	33 836	34 758
UG100k	37 056	35 903	36 831
UG1M	45 782	41 755	41 845
UG5M	94 534	75 651	65 727
UG10M	169 120	123 058	105 887
UG50M	844 998	538 378	412 250
UG100M	1 640 915	1 111 165	850 041
UG500M	-	6 761 140	4,984,180

en cuenta que con el Método U1, no se generan grafos de tamaño mayor que 100 millones de nodos, debido a que se demora aproximadamente el doble de tiempo que los Métodos U2 y U3 (ver Figura 5.5).

Escalabilidad

(i) Escalabilidad respecto al tamaño del grafo.

Respecto al tamaño del grafo que se puede generar usando los métodos para grafos no dirigidos, hay que destacar que los Métodos U2, U3_1, U3_2 y U3_3, permiten generar incluso grafos de 500 millones de nodos, pudiendo incluso llegar a tamaños de grafo más grandes, sin problemas, si se aumenta la cantidad de trabajadores del *cluster*.

(ii) Escalabilidad respecto al *cluster*

Del análisis de los resultados de las Tablas 5.7, 5.8, 5.9, 5.10 y 5.11, se puede concluir que para grafos con menos de 5 millones de nodos, no hay beneficio en términos de tiempo, al aumentar la cantidad de trabajadores del *clusters*. En cambio, para grafos de tamaño $n > 5M$, aumentar la cantidad de nodos del *cluster* permite generar los grafos mucho más rápido, característica que se acentúa a medida que se aumenta la cantidad de nodos del grafo. Este comportamiento, aplica a todos los métodos para generar grafos no dirigidos. En la figura 5.6, se puede observar un gráfico que ilustra este comportamiento para el método U1. En el Anexo A, se puede

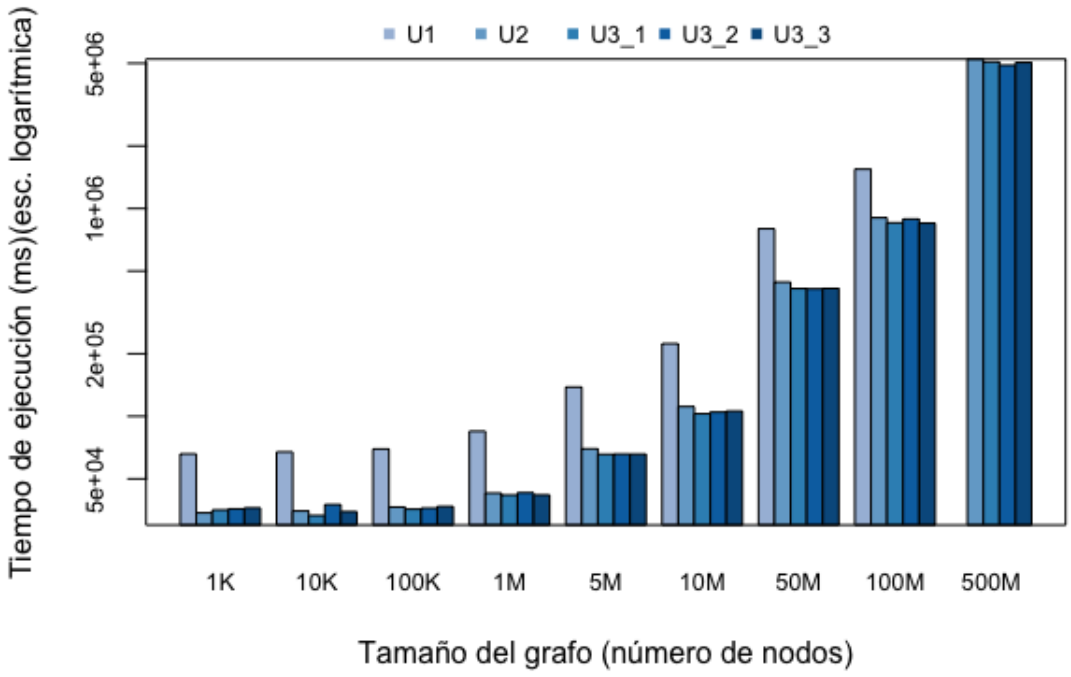


Figura 5.5: Tiempo de ejecución de los Métodos U1, U2, U3_1, U3_2 y U3_3 para generar grafos no dirigidos, de tamaños $n = 10M$, $n = 50M$, $n = 100M$ y $n = 500M$, en un *cluster* de 16 trabajadores.

encontrar el mismo gráfico para el resto de los métodos (U2, U3_1, U3_2 y U3_3).

Realismo

En la Tabla 5.12 se muestran los resultados del cálculo del estadístico Kolmogorov-Smirnov (KS) de cada uno de los métodos para grafos no dirigidos, usando grafos con los siguientes tamaños (número de nodos): $n_1 = 1k$, $n_2 = 10k$, $n_3 = 100k$, $n_4 = 1M$ y $n_5 = 10M$.

Analizando la Tabla 5.12, se puede ver que todos los valores de KS son inferiores a 0,15, lo cual indica que todos los métodos permiten generar grafos que se ajustan muy bien a una ley de potencia. Un comportamiento interesante se puede notar en el Método U3_3, el cual, para grafos sobre un millón de nodos, presenta los valores más bajos de todos los métodos.

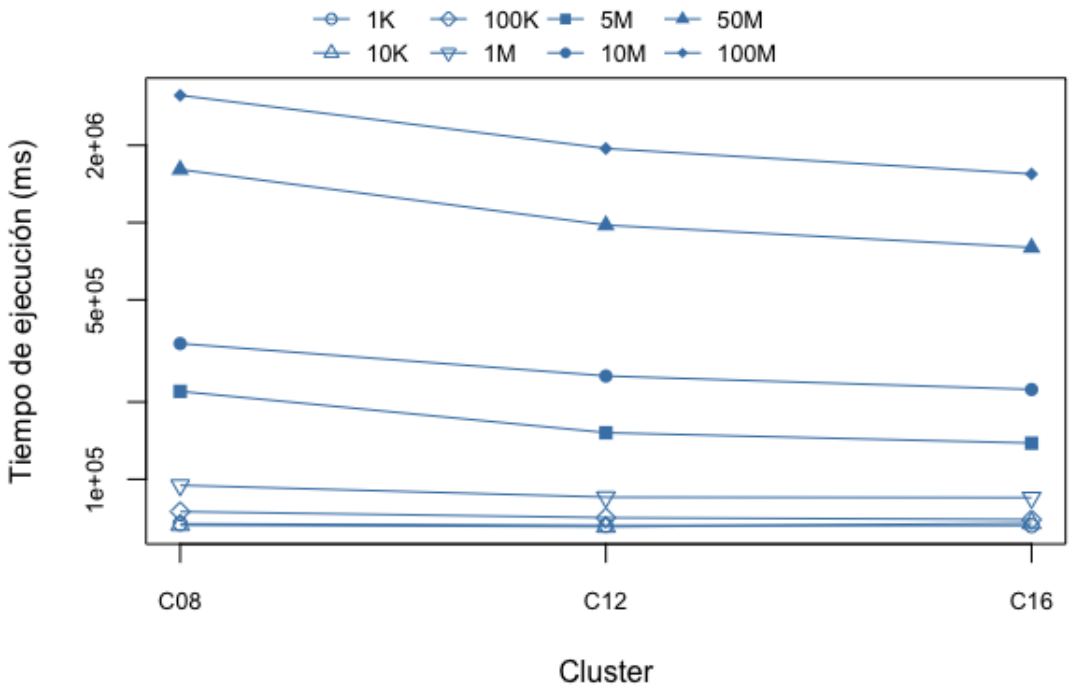


Figura 5.6: Tiempo de ejecución del Método U1 para generar grafos de tamaños $n = 1k$, $n = 10k$, $n = 100k$, $n = 1M$, $n = 5M$,$n = 10M$, $n = 50M$, $n = 100M$ y $n = 500M$, en *clusters* con 8, 12 y 16 trabajadores.

Tabla 5.12: KS para los grafos generados usando los métodos U1, U2, U3_1, U3_2 y U3_3.

Método / Tamaño	**1k**	**10k**	**100k**	**1M**	**5M**	**10M**
U1	0.1108	0.0460	0.0399	0.0313	0.0291	0.0224
U2	0.1286	0.0758	0.0415	0.0348	0.0301	0.0249
U3_1	0.1279	0.0889	0.0420	0.0295	0.0251	0.0264
U3_2	0.1191	0.0629	0.0387	0.0290	0.0257	0.0249
U3_3	0.1122	0.0674	0.0497	**0.0258**	**0.0222**	**0.0191**

Al igual que el caso de los métodos para grafos dirigidos, en los Métodos U1, U2, U3_1, U3_2 y U3_3, se puede notar que a medida que aumenta el tamaño del grafo, el KS es cada vez menor, por lo tanto, se puede inferir que grafos con más de 10 millones de nodos, también siguen una distribución de ley de potencia.

Con el mismo objetivo de comprobar que los grafos generados, sigan la ley de potencia, en la Figura 5.7 se muestran los gráficos para el más pequeño ($n = 1k$) y el más grande ($n = 10M$) de los grafos generados para los Métodos U1 y U2, los cuales se sabe que no generan la cantidad deseada de aristas. Por otro lado, en la Figura 5.8, se pueden ver los gráficos que muestran la distribución de la aridad de los grafos, producidos usando los Métodos U3_1, U3_2 y U3_3.

Del análisis visual de los gráficos para los Métodos U1 y U2, se puede decir que para grafos pequeños, la distribución de las aristas del Método U1 no es tan buena como el Método U2. En efecto, el primero, se ve un conjunto de puntos que hace que se pierda la forma que se espera que tenga una distribución de ley de potencia, el cual es más parecido al producido por el Método U2, a pesar de que el KS calculado para este último es siempre mayor que el de U1. Para grafos más grandes, los resultados de ambos métodos son similares.

Comparando los gráficos de distribución de aridad de los grafos generados con los Métodos U3_1, U3_2 y U3_3, se puede ver que el método que distribuye las aristas de mejor forma es el U3_2. Sin embargo, considerando el estadístico KS, se puede concluir que para generar grafos grandes, el Método U3_3 es también una buena alternativa, ya que, además de presentar el menor valor de KS, no generar aristas duplicadas y no perder aristas, el gráfico de la distribución de la aridad se ajusta muy bien a lo que se espera de una distribución de ley de potencia.

5.4. Comparación con el Estado del Arte

A continuación, se describen los resultados obtenidos de los experimentos realizados con el objetivo de comparar, en términos de rendimiento y realismo, los métodos descritos en este trabajo, con las soluciones presentes en el estado del arte. En base a los resultados obtenidos del análisis de cada uno de los Métodos en la sección 5.3, para esta comparación se considera el Método D1 para grafos dirigidos y U3_3 para grafos no dirigidos, ya que demostraron ser los mejores en términos de eficiencia, escalabilidad y realismo.

Para establecer puntos de comparación, en primer lugar se comparan los métodos (D1 y U3_3) con la solución secuencial: R^3MAT; y en segundo lugar, se comparan los mismos métodos con TeGViz y PegasusN, las cuales corresponden a soluciones distribuidas para la generación de grafos.

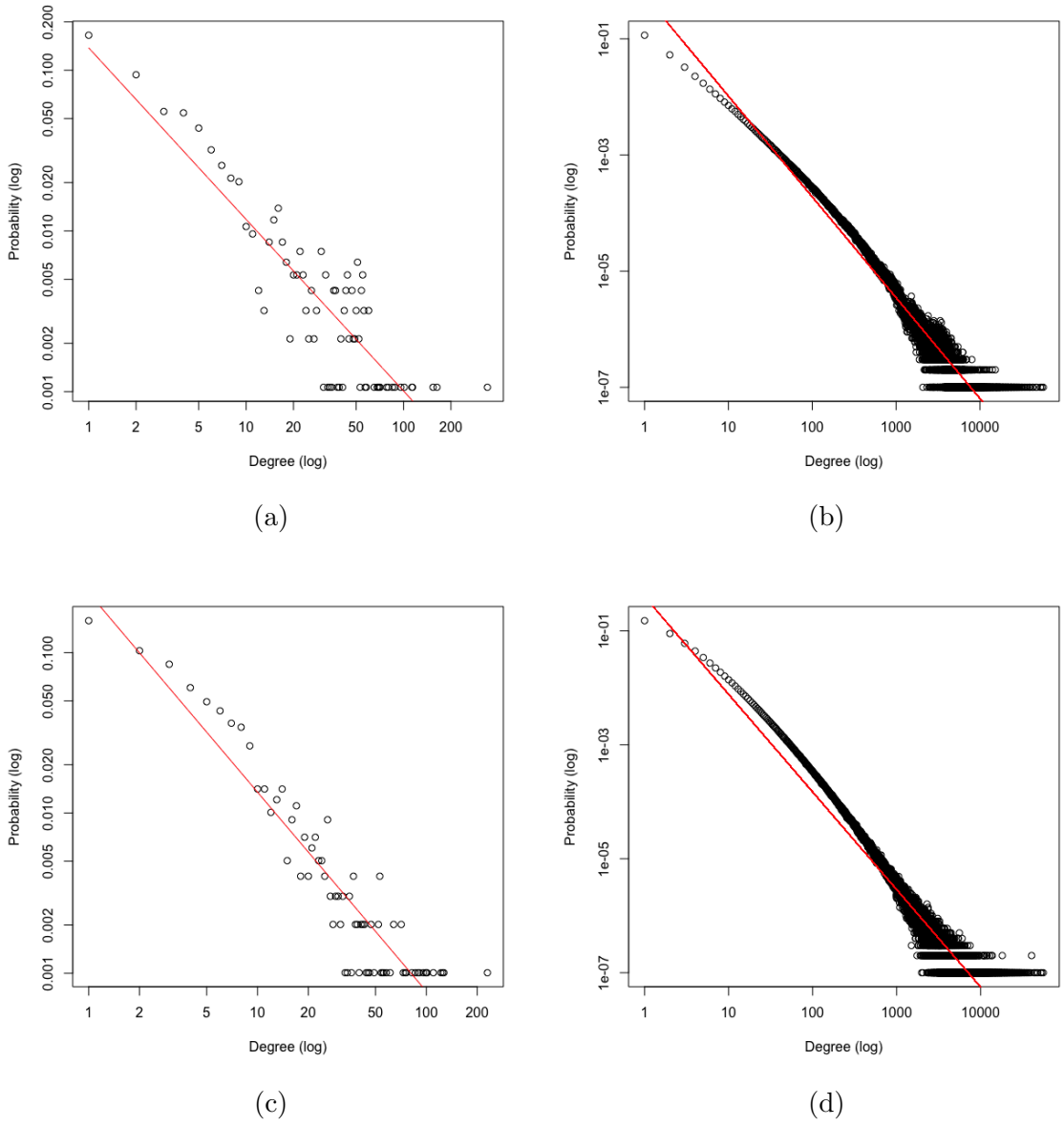


Figura 5.7: Distribución de la aridad, en escala logarítmica, de los grafos no dirigidos, generados con los Métodos U1 y U2. Los gráficos (a) y (b) muestran la distribución de aridad para grafos de tamaños $n = 1k$ y $n = 10M$ respectivamente, generados usando el Método U1. Los gráficos (c) y (d) muestran la distribución de la aridad para grafos con los mismos tamaños, generados usando el Método U2.

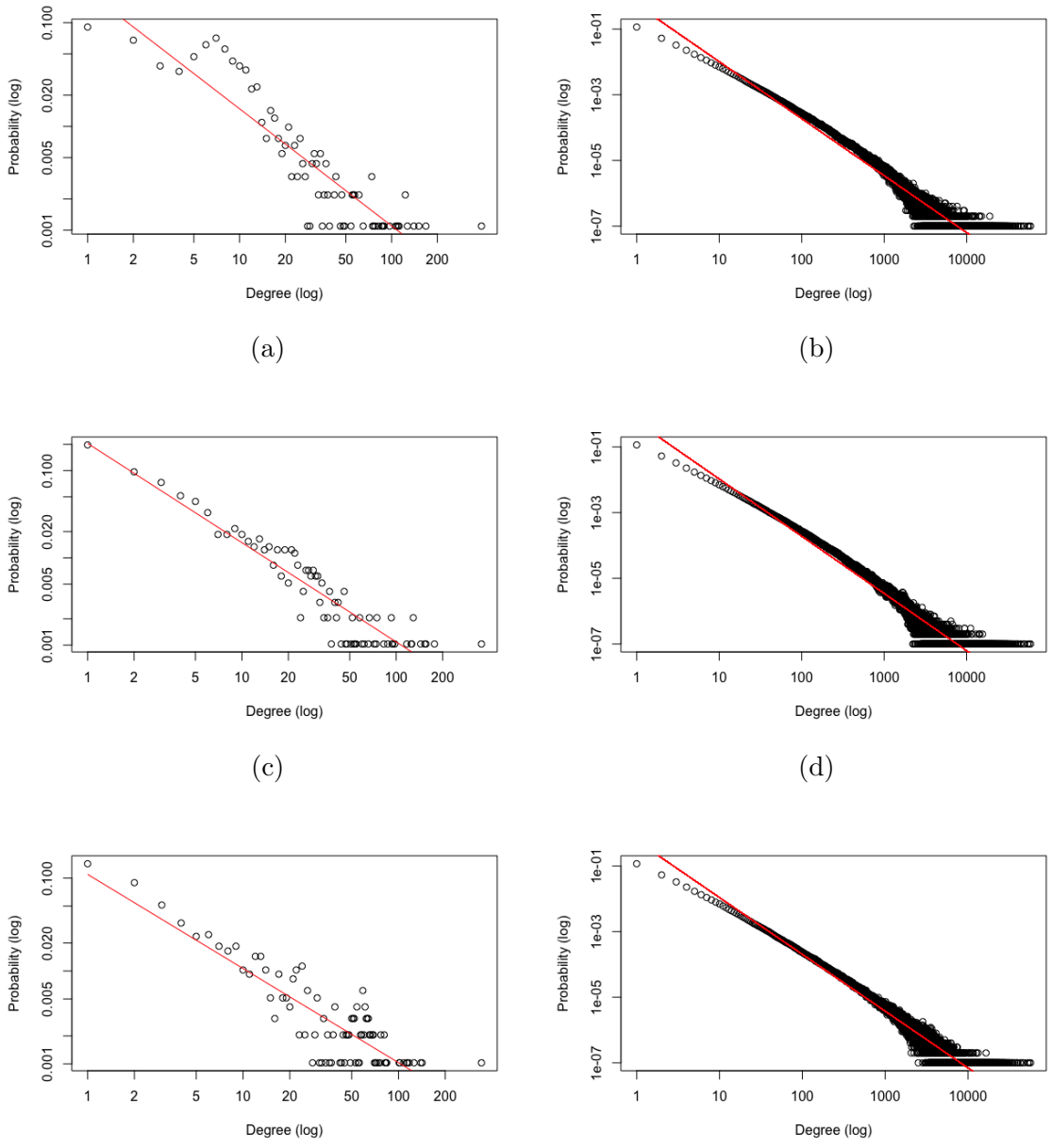


Figura 5.8: Distribución de la aridad, en escala logarítmica, de los grafos no dirigidos, generados con los Métodos U3_1, U3_2 y U3_3. Los gráficos (a) y (b) muestran la distribución de aridad para grafos de tamaños $n = 1k$ y $n = 10M$ respectivamente, generados usando el Método U3_1. Los gráficos (c) y (d) muestran la distribución de la aridad para grafos con los mismos tamaños, generados usando el Método U3_2. Por último, los gráficos (e) y (f) muestran la distribución de la aridad para grafos generados usando el Método U3_3.

Tabla 5.13: Comparación del tiempo de ejecución del método R^{3}MAT versus D1.

Grafo ID	**Tiempo de ejecución (ms)**	
	R^{3}MAT	**D1**
DG1k	34	34 976
DG10k	136	34 901
DG100k	772	36 241
DG1M	9 161	44 882
DG5M	60 293	77 381
DG10M	133 536	126 882
DG50M	853 386	538 561
DG100M	1 876 384	1 139 907
DG500M	-	6 839 663

Comparación con R^{3}MAT.

En la Tabla 5.13 se muestran los tiempos de ejecución de R^{3}MAT al generar grafos dirigidos de diferentes tamaños, junto con los tiempos de ejecución del Método D1 en un *cluster* de 12 trabajadores. Del análisis de los resultados, se puede concluir que para grafos de pocos nodos ($n < 1M$), el método R^{3}MAT es mucho más rápido que D1. Sin embargo, para grafos con 10 millones de nodos o más, se empieza a obtener ganancia en términos de tiempo, al usar el método distribuido (ver Figura 5.9).

También, cabe mencionar que con el método R^{3}MAT no se pueden generar grafos de 500 millones de nodos, en una máquina virtual de 4GB de memoria RAM, debido al espacio de memoria necesario para su procesamiento, en cambio, el método D1, como cada nodo del *cluster* genera una parte del total de las aristas, no presenta esa restricción, y en caso de presentarla, la solución es simplemente aumentar la cantidad de trabajadores del *cluster*.

En cuanto a la generación de grafos no dirigidos, en la Tabla 5.14, se muestran los tiempos de ejecución para grafos de diferentes tamaños, usando R^{3}MAT y el Método U3_3. Al igual que para el caso de grafos dirigidos, el método distribuido es más rápido para generar grafos grandes ($n > 10$ millones). Esto se ve reflejado de forma gráfica en la Figura 5.10, en la cual se puede apreciar como la línea correspondiente al Método U3_3 se mantiene por debajo de la de R^{3}MAT a partir de 10 millones de nodos. También, cabe mencionar que, para grafos 500 millones de nodos, R^{3}MAT, aborta la ejecución por falta de recursos.

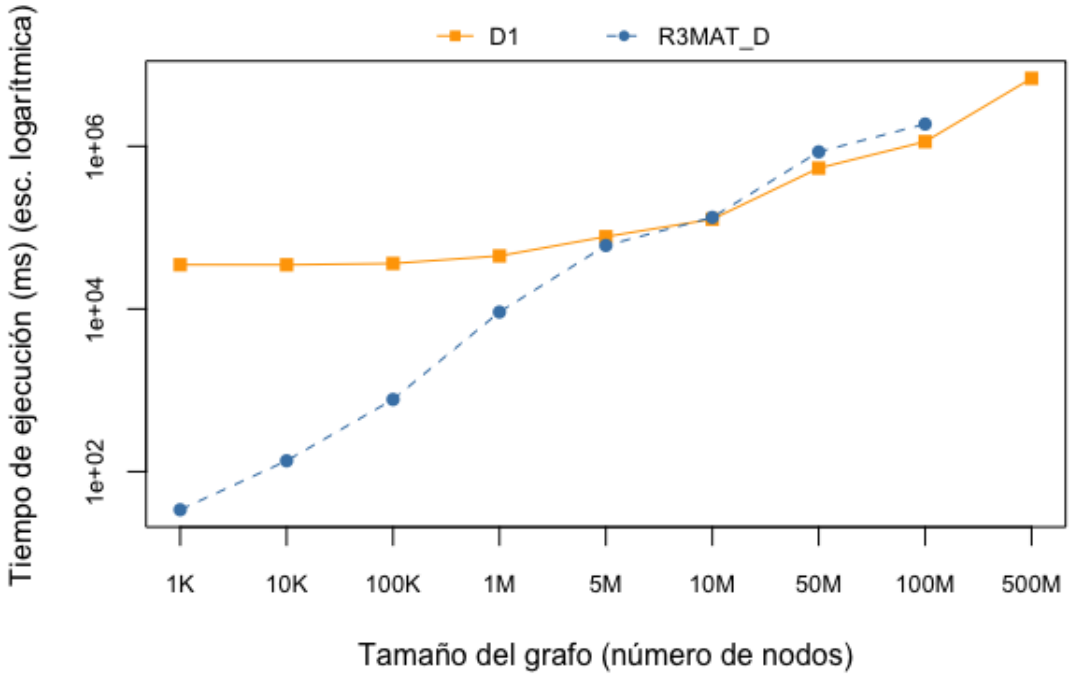


Figura 5.9: Comparación de los tiempos de ejecución del Método D1 con R^3MAT, al generar grafos dirigidos cuyos tamaño van desde 1000 nodos (1k) hasta 500 millones de nodos (500M). D1 se ejecuta en un cluster de 12 trabajadores, mientras que R^3MAT se ejecuta en un computador de 4GB de RAM.

Tabla 5.14: Comparación del tiempo de ejecución del método R^{3}MAT versus U3_3.

Grafo ID	**Tiempo de ejecución (ms)**	
	R^{3}MAT	**U3_3**
UG1k	38	34 023
UG10k	146	33 836
UG100k	826	35 903
UG1M	9 704	41 755
UG5M	63 406	75 651
UG10M	139 643	132 058
UG50M	858 331	538 378
UG100M	1 944 733	1 111 165
UG500M	-	6 761 140

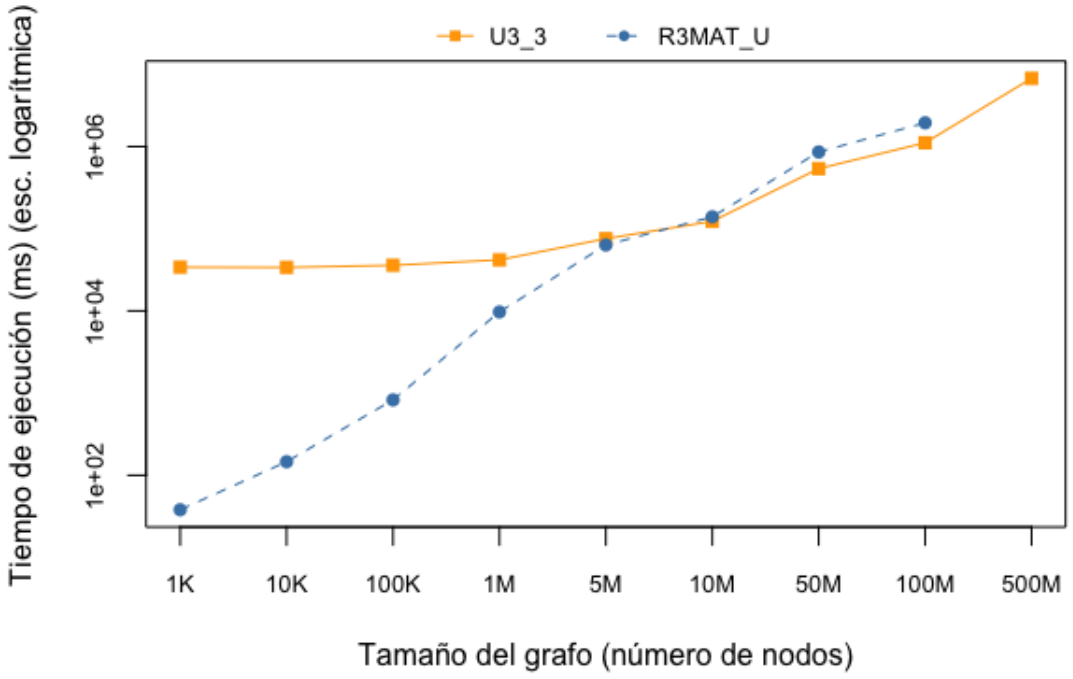


Figura 5.10: Comparación de los tiempos de ejecución del Método U3_3 con R^3MAT, al generar grafos dirigidos cuyos tamaño van desde 1000 nodos (1k) hasta 500 millones de nodos (500M). U3_3 se ejecuta en un cluster de 12 trabajadores, mientras que R^3MAT se ejecuta en un computador de 4GB de RAM.

Comparación con TeGViz y PegasusN

Al ejecutar TeGViz, PegasusN, D1 y U3_3 en un *cluster* de 12 trabajadores, se obtienen los resultados que se muestran en la Tabla 5.15. PegasusN no tiene una forma de configurar que tipo de grafo se quiere generar, por lo tanto, sólo se registran los tiempos, asumiendo que el resultado es un grafo dirigido. Por su parte, TeGViz, tiene una opción que permite evitar que aparezcan aristas repetidas, por lo tanto, para efectos de este análisis, si esta opción está activada, se asume que el grafo que se obtiene es un grafo no dirigido; en caso contrario, se analiza como un grafo dirigido.

Teniendo en consideración lo anterior, en la Tabla 5.15 se observa que: PegasusN es el método más rápido para todos los tamaños de grafos y TeGViz presenta tiempos similares a D1 en grafos pequeños, pero es más rápido para producir grafos grandes

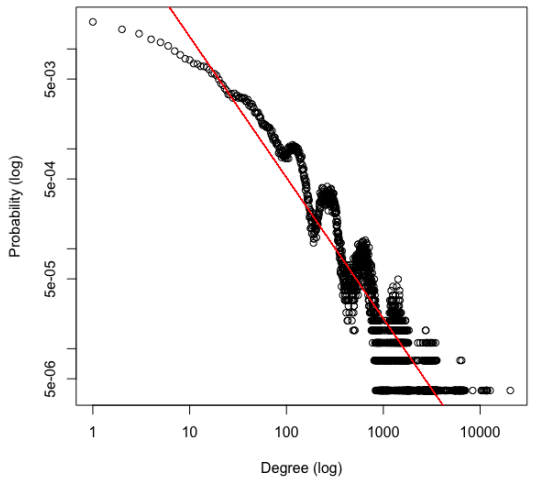
Tabla 5.15: Comparación del tiempo de ejecución de PegasusN y TeGViz versus D1 y U3_3 en un *cluster* de 12 trabajadores.

Grafo ID	**Tiempo de ejecución (ms)**			
	PegasusN	**TeGViz**	**D1**	**U3_3**
DG1k	33 393	33 457	34 976	-
UG1k	-	42 347	-	34 023
DG10k	33 004	33 570	34 901	-
UG10k	-	78 919	-	33 836
DG100k	33 082	35 160	36 241	-
UG100k	-	-	-	35 903
DG1M	37 917	42 325	44 882	-
UG1M	-	-	-	41 755
DG5M	52 888	65 566	77 381	-
UG5M	-	-	-	75 651
DG10M	68 149	99 966	126 882	-
UG10M	-	-	-	123 058
DG50M	163 092	354 209	538 561	-
UG50M	-	-	-	538 378
DG100M	332 878	693 801	1 139 907	-
UG100M	-	-	-	1 111 165

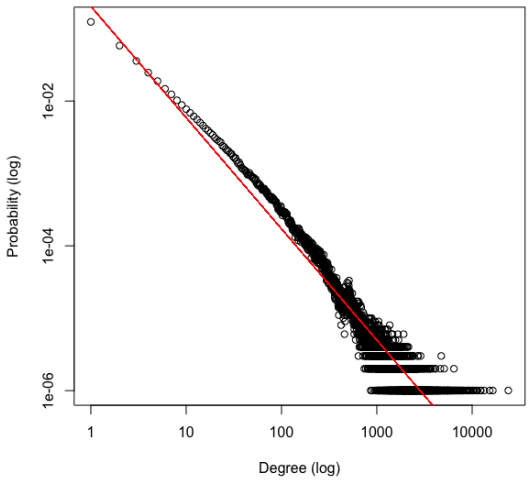
($n > 10M$). Sin embargo, hay que considerar que ni PegasusN, ni TeGViz aseguran que se genere la cantidad de aristas deseadas, siendo en ese sentido, D1 y U3_3, claros ganadores.

En cuanto a realismo, en la Figura 5.11 se muestran los gráficos de distribución de aridad de grafos de un millón de nodos, para PegasusN, D1 y U3_3 respectivamente. La lista de aristas generadas con TeGViz no logró ser procesada usando el *script* en R, por lo tanto, no se pudo construir el gráfico correspondiente, ni tampoco se pudo calcular el estadístico Kolmogorov-Smirnov para dicho método. Por su parte, PegasusN, gráficamente muestra una distribución de ley de potencia errática, mientras que D1 y U3_3 presentan una mejor distribución de aridad.

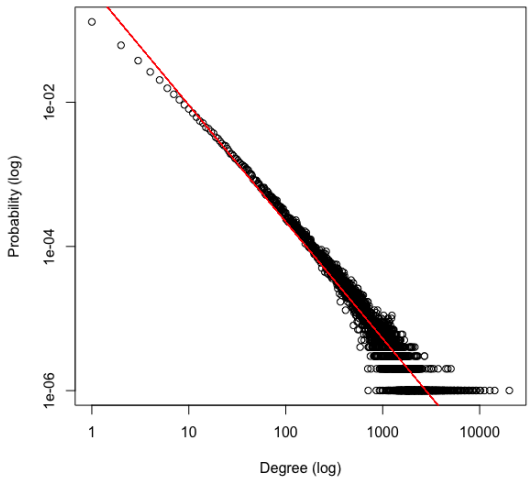
Al calcular el estadístico Kolmogorov-Smirnov para los mismos grafos de la Figura 5.11, se obtienen los siguientes resultados: $ks = 0{,}0928$ para PegasusN, $ks = 0{,}0340$ para D1, y $ks = 0{,}0257$ para U3_3. Por lo tanto, se puede decir que los Métodos D1 y U3_3, presentan el mejor ajuste a una ley de potencia con respecto a los otros generadores analizados.



(a) PegasusN



(b) D1



(c) U3_3

Figura 5.11: Distribución de la aridad, en escala logarítmica, de los grafos generados con PegasusN (a), el Método D1 (b) y el Método U3_3 (c).

6. Conclusiones

En este capítulo se describen y discuten los resultados alcanzados, el cumplimiento de los objetivos planteados y, por último, se esbozan ideas relacionadas con este trabajo, que pueden ser desarrolladas a futuro.

6.1. Sobre los Resultados Alcanzados

En este trabajo se abordó el problema de generación de grafos grandes que siguen la ley de potencia, usando un entorno distribuido. Esto, con el objetivo de mejorar la escalabilidad del método secuencial R^3MAT, el cual, se considera un método extremadamente robusto en cuanto a su eficiencia, escalabilidad y realismo, pero que para grafos grandes, toma mucho tiempo.

En términos generales, como solución al problema planteado, se desarrollaron y evaluaron métodos que permiten generar grafos grandes, usando una variante distribuida del método R^3MAT, empleando para ello el modelo de programación *MapReduce*.

En una primera fase del trabajo, se realizó una revisión exhaustiva del estado del arte, seguida por el diseño, construcción y validación de los nuevos métodos de generación de grafos, para concluir con la ejecución de una serie de experimentos que permiten evaluar cada uno de los métodos y compararlos, en términos de eficiencia, escalabilidad y realismo. Eficiencia en el sentido del tiempo de ejecución de los métodos. Escalabilidad en el sentido del tamaño del grafo que se puede generar y respecto de la cantidad de trabajadores del *cluster*. Y realismo, en el sentido de que la aridad del grafo generado sigue una distribución de ley de potencia.

Considerando si lo que se quiere generar es un grafo dirigido o uno no dirigido, y

el modelo de programación *MapReduce*, se desarrolla un total de ocho algoritmos: D1 y D2 para grafos dirigidos; U1, U2, U3_1, U3_2, U3_3 y U4 para grafos no dirigidos.

En la evaluación experimental se demuestra que para grafos dirigidos, D1 es el que permite generar grafos más grandes, llegando incluso a generar grafos de 500 millones de nodos, cuyo tiempo depende de la cantidad de trabajadores del *cluster* en el que se ejecute. Si bien, al usar la función *Combiner*, se esperaba que D2 produciera grafos en mucho menos tiempo que D1, en el mismo *cluster*, sólo se obtuvo una ganancia pequeña de aproximadamente 10 segundos. Además, D2 tiene la desventaja que para grafos grandes ($n > 10M$), requiere que el *cluster* tenga más recursos que D1, de lo contrario, genera una lista de aristas incompleta, que no tiene relación alguna con el resultado esperado. En cuanto a realismo, la evaluación demuestra que los grafos generados usando D1 y D2, presentan una distribución de aridad que sigue la ley de potencia.

Respecto a los métodos para generar grafos no dirigidos, en base a la evaluación realizada, se puede concluir que el mejor método es U3_3, debido a que: (i) genera aristas únicas (no repetidas), (ii) genera el total de aristas calculadas al inicio, (iii) para grafos grandes, presenta el menor KS, y (iv) gráficamente, presenta una distribución de ley de potencia más ajustada, tanto para grafos pequeños, como para grafos grandes.

De la comparación con los métodos distribuidos, presentes en el estado del arte, se puede concluir que los métodos propuestos, a pesar de ser más lentos en algunos casos, presentan mejor desempeño en términos de la completitud y correctitud del grafo que se quiere generar, ya que permiten, primero, distinguir entre el tipo de grafo que se quiere generar; segundo, asegurar que la cantidad de aristas que se genera es consistente con las requeridas; y tercero, la distribución de aridad es más parecida a lo que se espera de una ley de potencia.

6.2. Sobre la Hipótesis y el Cumplimiento de los Objetivos

Realizada la evaluación, en la que se compara el desempeño de los métodos distribuidos con el método R^{3}MAT, con respecto a la hipótesis planteada, se puede afirmar que: la implementación del método R^{3}MAT usando el modelo de programación MapReduce permite reducir el tiempo de procesamiento e incrementar el soporte para generar grafos de gran tamaño, teniendo en consideración que su desempeño pudiese

ser mejor o peor dependiendo de las características del *cluster* en el que se ejecute.

En ese mismo sentido, se concluye que el uso de los métodos distribuidos, tienen como principal ventaja, más allá de que permiten reducir el tiempo para grafos grandes, que permiten generar grafos mucho más grandes que el método secuencial, sin restricciones aparentes en términos de recursos computacionales, dadas las características propias de *MapReduce* en cuanto a escalabilidad.

El primer objetivo específico: "diseñar algoritmos para la generación de grafos basados en R^{3}MAT y *MapReduce*", se cumple en su totalidad. Se diseñan algoritmos diferentes para grafos dirigidos y para no dirigidos. En el caso de los métodos dirigidos, se construyen dos algoritmos, donde la única diferencia entre ellos, es que el segundo incluye la función *Combine* de Hadoop. Para los métodos de grafos no dirigidos, se parte con el diseño de métodos que aseguran que el grafo producido no tiene aristas repetidas, con el problema que, el grafo resultante tiene menos aristas que las requeridas. A partir de esas soluciones, buenas en el sentido de que la aridad de los grafos producidos sigue la ley de potencia, se busca resolver el problema de la pérdida de aristas, logrando encontrar tres estrategias que permite resolver dicho problema.

El segundo objetivo específico: "implementar los algoritmos propuestos usando Apache Hadoop", también se cumple totalmente. Todos los diseños, se implementan en lenguaje Java, usando el *framework Hadoop - MapReduce*.

Por último, el tercer objetivo específico: "evaluar la eficiencia de los algoritmos y validar sus propiedades", se cumple también en su totalidad. Los métodos implementados, se evalúan en una serie de experimentos, realizados en *clusters* configurados usando el servicio Dataproc de Google Cloud Platform. Cada método se evalúa en términos de eficiencia, escalabilidad y realismo, variando el tamaño de los grafos y la cantidad de trabajadores del *cluster*. Además, se realiza una comparación con R^{3}MAT y otros métodos distribuidos descritos en el estado del arte.

Dado el cumplimiento de todos los objetivos específicos, el objetivo general: desarrollar y evaluar métodos de generación de grafos que combinen R^{3}MAT con *MapReduce*, también se cumple en su totalidad.

6.3. Trabajo Futuro

Como trabajo futuro se tiene la posibilidad de:

- Implementar los métodos usando Apache Spark, de tal forma de explorar la posibilidad de mejorar el tiempo de ejecución, en particular, el tiempo que toma generar el arreglo de aridades.
- Evaluar el comportamiento de los métodos en *clusters* con máquinas con más memoria RAM.
- Diseñar un método no recursivo para generar el arreglo de aridades de forma más eficiente.
- Analizar el efecto que tienen los métodos utilizados en la etapa de generación de aristas, en otras características de los grafos, tales como, el diámetro y la formación de comunidades.

Bibliografía

[1] Apache Hadoop 3.3.0 – MapReduce Tutorial. https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html.

[2] GitHub - chan150/TrillionG: TrillionG: A Trillion-scale Synthetic Graph Generator Using a Recursive Vector Model [SIGMOD17]. https://github.com/chan150/TrillionG.

[3] Graph 500 — large-scale benchmarks. https://graph500.org/.

[4] PegasusN. https://datalab.snu.ac.kr/pegasusn/.

[5] R: The R Project for Statistical Computing. https://www.r-project.org/.

[6] Synthetic data generation — a must-have skill for new data scientists. https://towardsdatascience.com/synthetic-data-generation-a-must-have-skill-for-new-data-scientists-915896c0c1ae, Consultado el 29 de agosto de 2019.

[7] TeGViz. https://datalab.snu.ac.kr/tegviz/.

[8] Renzo Angles, Rodrigo Paredes, and Roberto García. R3mat: A rapid and robust graph generator. *IEEE Access*, pages 1–1, 2020.

[9] Ashish Bakshi. MapReduce Tutorial — Mapreduce Example in Apache Hadoop — Edureka. https://www.edureka.co/blog/mapreduce-tutorial/{\#}mapreduce{_}example{_}program, 2018.

[10] Deepayan Chakrabarti and Christos Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM Comput. Surv.*, 38(1), June 2006.

[11] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. *R-MAT: A Recursive Model for Graph Mining*, pages 442–446.

[12] Aaron Clauset, Cosma Rohilla Shalizi, and Mark Newman. Power-law distributions in empirical data. *SIAM Review*, 51(4):661–703, 2009.

[13] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[14] Nick Edmonds, Brian Barrett, and Andrew Lumsdaine. Parallel BGL Scalable R-MAT generator - 1.73.0, 2009.

[15] Thilina Gunarathne. *Hadoop MapReduce v2 Cookbook*. Packt Publishing Ltd, 2015.

[16] ByungSoo Jeon, Inah Jeon, and U Kang. Tegviz: Distributed tera-scale graph generation and visualization. In *2015 IEEE International Conference on Data Mining Workshop (ICDMW)*, pages 1620–1623, Nov 2015.

[17] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. Scalable simd-efficient graph processing on gpus. In *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques*, PACT '15, pages 39–50, 2015.

[18] Jurij Leskovec, Deepayan Chakrabarti, Jon Kleinberg, and Christos Faloutsos. *Realistic, Mathematically Tractable Graph Generation and Evolution, Using Kronecker Multiplication*, volume 3721 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[19] Gregory Mone. Beyond hadoop. *Commun. ACM*, 56(1):22–24, January 2013.

[20] Ha-Myung Park, Chiwan Park, and U Kang. PegasusN: A scalable and versatile graph mining system. In *AAAI*, 2018.

[21] Himchan Park and Min-Soo Kim. Trilliong: A trillion-scale synthetic graph generator using a recursive vector model. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 913–928, New York, NY, USA, 2017. Association for Computing Machinery.

[22] David M. Pennock, Gary W. Flake, Steve Lawrence, Eric J. Glover, and C. Lee Giles. Winners don't take all: Characterizing the competition for links on the web. *Proceedings of the National Academy of Sciences*, 99(8):5207–5211, 2002.

[23] Steven J. Plimpton and Karen D. Devine. Mapreduce in MPI for large-scale graph algorithms. *Parallel Comput.*, 37(9):610–632, September 2011.

[24] Arnau Prat-Pérez, Joan Guisado-Gámez, Xavier Fernández Salas, Petr Koupy, Siegfried Depner, and Davide Basilio Bartolini. Towards a property graph generator for benchmarking. *CoRR*, abs/1704.00630, 2017.

[25] Salman Salloum, Ruslan Dautov, Xiaojun Chen, Patrick Xiaogang Peng, and Joshua Zhexue Huang. Big data analytics on apache spark. *International Journal of Data Science and Analytics*, 1(3):145–164, Nov 2016.

[26] Roberto Sampieri and Carlos Collado. Metodología de la investigación (6ta edición ed.). *DF México: Mc Graw Hill*, 2014.

[27] Robert Sedgewick and Kevin Wayne. *Algorithms, 4th Edition.* Addison-Wesley, 2011.

[28] Koji Ueno and Toyotaro Suzumura. Highly scalable graph search for the graph500 benchmark. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, page 149–160, New York, NY, USA, 2012. Association for Computing Machinery.

[29] Douglas Brent West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, 2001.

ANEXOS

A. Escalabilidad de los métodos según el *cluster*

A continuación se encuentran los gráficos correspondientes al tiempo de ejecución de los métodos U2, U3_1, U3_2 y U3_3 en *clusters* con diferentes números de trabajadores.

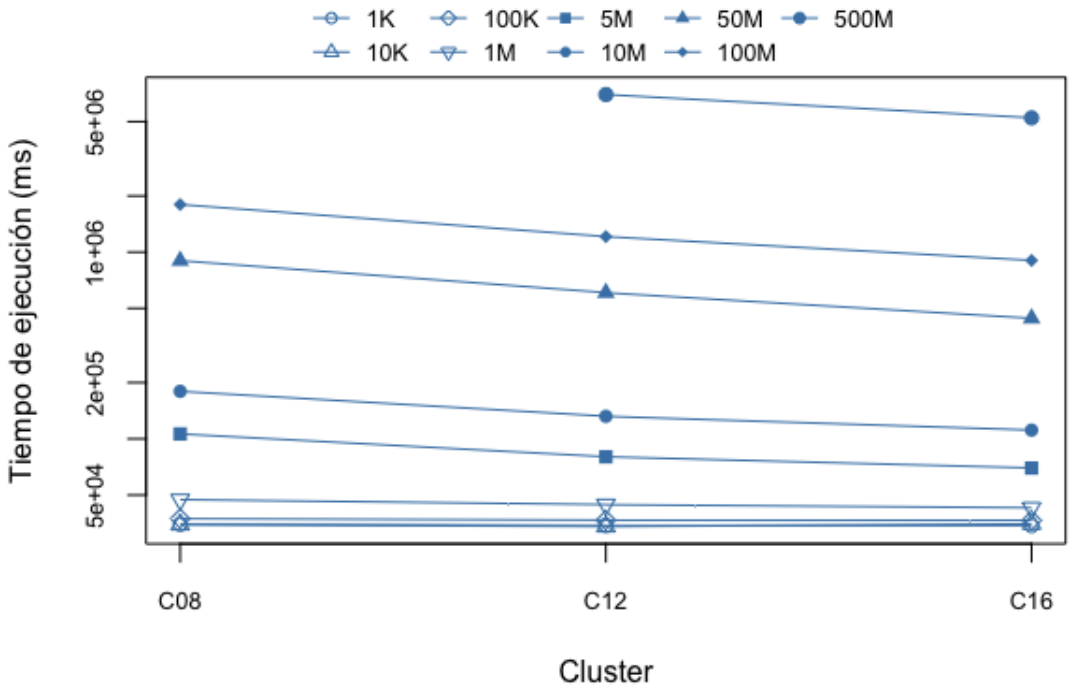


Figura A.1: Tiempo de ejecución del método U2 para generar grafos de tamaños $n = 1k$, $n = 10k$, $n = 100k$, $n = 1M$, $n = 5M$,$n = 10M$, $n = 50M$, $n = 100M$ y $n = 500M$, en *clusters* con 8, 12 y 16 trabajadores.

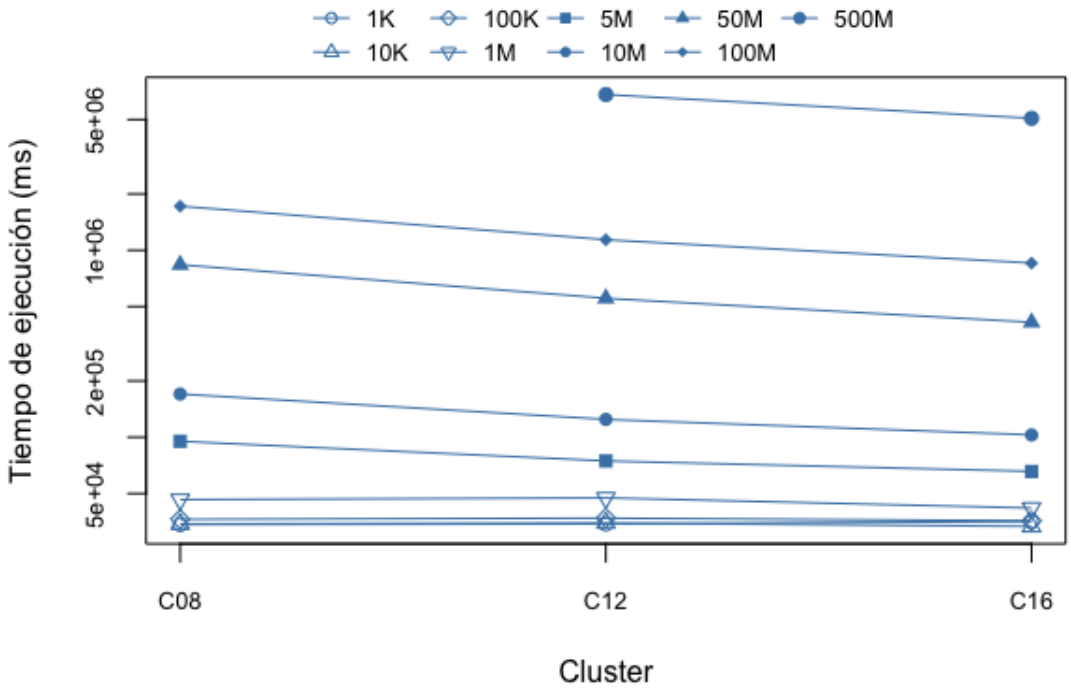


Figura A.2: Tiempo de ejecución del método U3_1 para generar grafos de tamaños $n = 1k$, $n = 10k$, $n = 100k$, $n = 1M$, $n = 5M$,$n = 10M$, $n = 50M$, $n = 100M$ y $n = 500M$, en *clusters* con 8, 12 y 16 trabajadores.

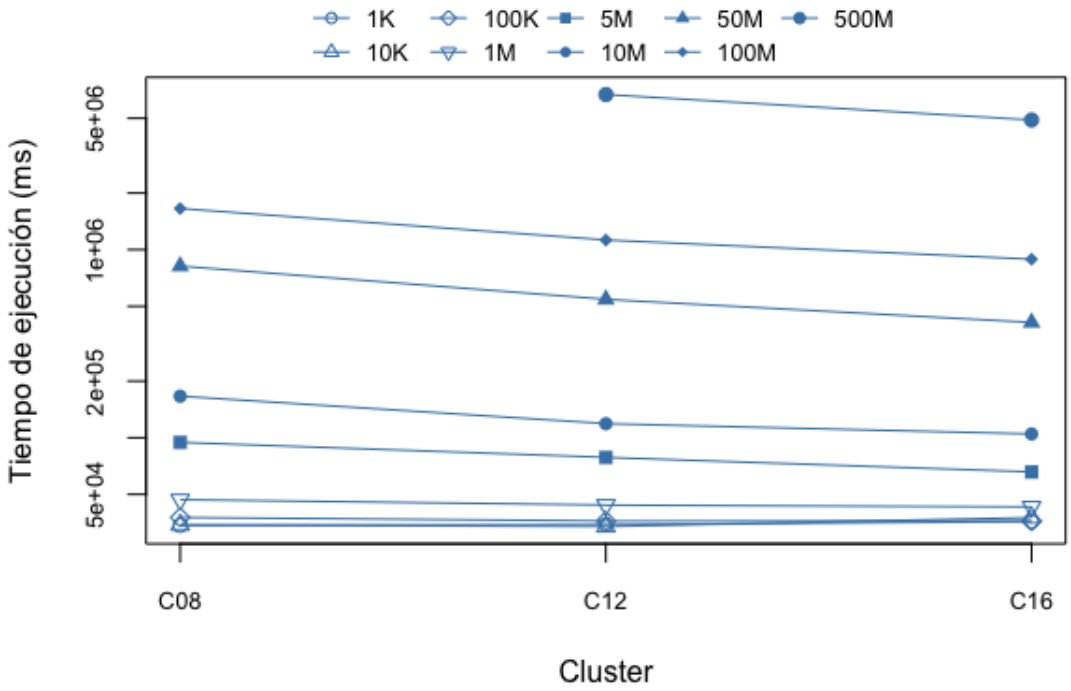


Figura A.3: Tiempo de ejecución del método U3_2 para generar grafos de tamaños $n = 1k$, $n = 10k$, $n = 100k$, $n = 1M$, $n = 5M$,$n = 10M$, $n = 50M$, $n = 100M$ y $n = 500M$, en *clusters* con 8, 12 y 16 trabajadores.

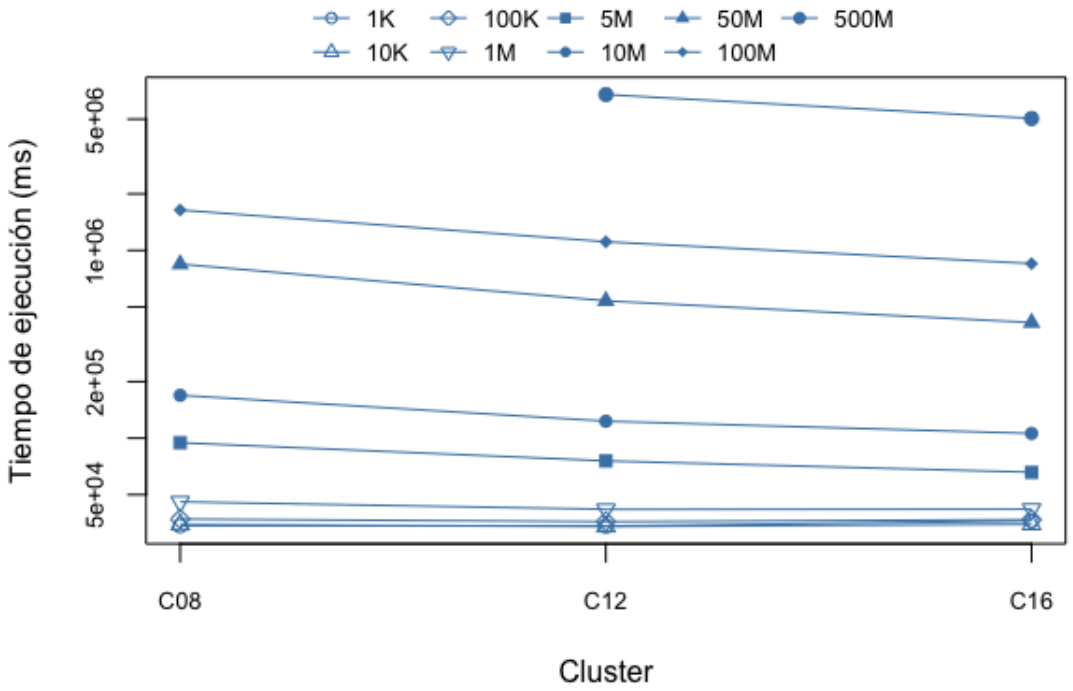


Figura A.4: Tiempo de ejecución del método U3_3 para generar grafos de tamaños $n = 1k$, $n = 10k$, $n = 100k$, $n = 1M$, $n = 5M$,$n = 10M$, $n = 50M$, $n = 100M$ y $n = 500M$, en *clusters* con 8, 12 y 16 trabajadores.