IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received May 15, 2021, accepted June 17, 2021, date of publication June 29, 2021, date of current version July 9, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3093329

Power-Law Distributed Graph Generation With MapReduce

RENZO ANGLES[1,2], FERNANDA LÓPEZ-GALLEGOS[2], AND RODRIGO PAREDES[1], (Member, IEEE)

[1]Department of Computer Science, Faculty of Engineering, Universidad de Talca, Curico 3340000, Chile
[2]Millennium Institute for Foundational Research on Data, Curico 3340000, Chile

Corresponding authors: Renzo Angles (rangles@utalca.cl) and Rodrigo Paredes (raparede@utalca.cl)

This work was supported by the Millennium Institute for Foundational Research on Data, Chile.

ABSTRACT A graph generator is a tool which allows to create graph-like data whose structural properties are very similar to those found in real world networks. This paper presents two methods to generate graphs with power-law edge distribution based on the MapReduce processing model that can be easily implemented to run on top of Apache Hadoop. The proposed methods allow the generation of directed and undirected power-law distributed graphs without repeated edges. Our experimental evaluation shows that our methods are efficient and scalable in terms of both graph size and cluster capacity.

INDEX TERMS Graph generator, MapReduce, Hadoop.

I. INTRODUCTION

Graphs are a recognized abstraction model as they can be used to represent structured and semi-structured data occurring in many application domains [1]. A graph generator is a tool which allows to generate graphs resembling (ideally) characteristics found in real world networks, such as a power-law distribution, a large clustering coefficient, a small diameter or a community structure [2]. There are several methods and tools for graph generation [3], [4], most of them designed to run in a single computer. This aspect restricts their scalability as the biggest graph a method is able to produce is directly related to the available main memory. A natural solution to such problem is the design of methods based on parallel and distributed computing. In this article, we concentrate our interest on the methods designed to run on top of the Apache Hadoop framework.

According to our review of the literature, several methods and tools have been created to exploit the advantages of parallel and distributed systems, allowing the generation of very large graphs (see Section II-A). However, our analysis of such methods shows that most of them are unable to generate undirected edges, suffer from duplicate edges, provide a reduced number of parameters (to personalize the generation), and their documentation and comparison is restricted.

The associate editor coordinating the review of this manuscript and approving it for publication was Walter Didimo.

This paper presents the novel methods H4DG and H4UG to generate power-law distributed graphs based on the MapReduce programming model, so they can be implemented to run in Apache Hadoop. H4DG generates directed graphs whilst H4UG produces undirected ones. They are inspired by R3MAT [5], a method that uses the graph degree distribution as the basis to conduct the generation process.

In general terms, the *Map* phase of H4DG generates (in parallel) a set of key-value pairs of the form $(k, 1)$, each one indicating the existence of an edge for the node k. In its *Shuffle* phase, where the pairs with the same key are joined automatically by Hadoop, H4DG computes a second set of key-value pairs of the form (k, m), where m is the number of pairs having the same key k. In the *Reduce* phase, these pairs are used to generate the final edges. For example, given the pair $(3, 4)$, the output will contain the edges $n_3 \rightarrow n_1$, $n_3 \rightarrow n_2$ and $n_3 \rightarrow n_4$, i.e. the target node identifier n_t could be lower o greater than the source node identifier n_s, but not the same. H4UG follows a similar procedure, but the target node identifier is always greater than the source node one, avoiding the incorrect generation of symmetric edges.

We conduct several experiments to evaluate the efficiency, scalability, and realism of the proposed methods. We test the generation of graphs having billion nodes and edges, using clusters of different size and capacity. In the largest experiment, H4DG produces a directed graph having one billion nodes ($1 \cdot 10^9$) and fourteen billion edges ($14 \cdot 10^9$)

in 41.8 minutes, whereas H4UG produces a similar undirected graph in 40 minutes, both running on a cluster with 32 machines. We verify that the produced graphs do present the characteristic power-law distribution found in real-world graphs. We also include a comparison with two competitors (TeGViz and PegasusN), showing that H4DG is mildly slower, but overcomes their deficiencies to generate undirected graphs without duplicate edges and exceeds their capacity of producing very large graphs.

The rest of the paper is organized as follows. Section II presents the related work and a summary of R3MAT. Section III shows our novel graph generation methods H4DG and H4UG. The experimental evaluation is presented in Section IV. Finally, conclusions and guidelines for future works are presented in Section V.

II. RELATED WORK

We present previous work on parallel and distributed alternatives for generating graphs and a brief description of R3MAT.

A. PREVIOUS WORK

There are several works proposing and comparing tools and methods for graph data generation [3], [4]. Considering that the methods proposed in this article have been designed to run on top of Apache Hadoop, we concentrate our interest on the solutions based on parallel and distributed computing. We present a brief description of each method, including their main features and issues. The methods are presented in chronological sort.

The Parallel Boost Graph Library is a collection of graph algorithms designed to exploit fine-grained parallelism by using the Bulk Synchronous Parallel (BSP) model. This library provides graph generators based on R-MAT [6] and Erdös-Rényi [7], two very-well know models for graph generation.

PegasusN [8], [9] is a graph mining system that provides algorithms for graph structure analysis, subgraph enumeration, graph visualization, and graph generation. It is fully written in Java and runs in parallel and distributed manner on top of Hadoop and Spark. PegasusN implements three models for graph generation: R-MAT [6], Kronecker [10], and Erdös-Rényi [7].

In [11], the authors present an implementation of R-MAT based on the Message Passing Interface (MPI, a standard designed to operate on parallel computing architectures). The advantage of this library is the compatibility with C, C++, and Python. The disadvantages, inherited from MPI, are data redundancy and the lack of failure tolerance.

Graph500 [12], [13] is a benchmark for supercomputers based on the analysis of large scale graphs. One of its components is a graph generator that implements the Kronecker model [10]. It is able to create graphs having trillion nodes, although it requires several computers connected by a high speed network.

S3G2 [14] is a generator that creates synthetic social graphs containing non-uniform value distributions and structural correlations. It is part of the Social Network Intelligence Benchmarks [15], a scientific benchmark created to evaluate RDF triple stores. Its experimental evaluation shows that it is able to generate 1.2TB of graph data in half an hour on a 16-node Hadoop cluster. Its implementation is unavailable.

BTER [16] is a scalable model that produces graphs with different degree distributions and clustering coefficients. The scalability is based on independent edge generation; i.e., there is no knowledge of previously-generated edges in deciding on the next one. There exists a reference implementation in MATLAB that is available at [17]. In [18], the authors show that a Hadoop implementation of BTER allows to create a graph with 120M nodes and 4.4B edges in less than 25 minutes on a 32-node Hadoop cluster. This implementation is not available.

XGDBench [19] is a graph database benchmarking platform for cloud computing systems. XGDBench is written in X10, a language intended for exascale systems. XGDBench includes a graph generator, called MAG, that implements the R-MAT model [6]. MAG is able to produce graphs with power-law distributions and real community structure.

PaRMAT [20] is a graph generator designed to take advantage of using threads. It is based on R-MAT [6], so it divides the adjacency matrix in small matrices so they are processed by multiple threads. PaRMAT allows the generation of directed and undirected edges, without duplicate edges. A C++ implementation of PaRMAT is available in GitHub [21].

TeGViz [22] is a tool to generate and visualize large scale graphs in a distributed way. The module for graph generation implements three generation models: Erdös-Rényi [7], R-MAT [6], and Kronecker. Its Hadoop implementation is available at [23]. In [22], the authors argued that TeGViz is able to generate a graph having 10^6 nodes in 10 seconds.

GraphX [24] is a component of Apache Spark which allows graph-parallel computation. GraphX includes generators for creating specific graph types such as log normal graphs, R-MAT graphs, grid graphs, and star graphs. The information available in the GraphX website [25] just describes the parameters of the generators.

Darwini [26] is an extension of BTER which follows the Vertex Centric Model. It is able to model a number of core characteristics of real graphs, including degree distribution, local clustering coefficient, and joint-degree distributions. The experimental evaluation of Darwini talks about an open source implementation [27] on top of Apache Giraph which is able create synthetic graphs with one trillion edges. We were unable to test such implementation.

TrillionG [28] is a tool designed to generate big graphs in short time and using a small amount of memory. The authors argued that TrillionG is able to create a graph having trillion nodes, following the RMAT or Kronecker models, in two hours using a cluster equipped with 10 computers. TrillionG was implemented in Scala to work on Apache Spark, and the source code is available in GitHub [29].

DataSynth [2] is a framework for property graph generation with customizable schemas and characteristics. DataSynth is based on a property-to-node matching algorithm that allows to resemble real-life characteristics, like correlation between properties and the structure of the graph. DataSynth was implemented to run on top of Apache Spark and the source code is available at [30].

B. R3MAT

Our methods H4DG and H4UG are inspired by R3MAT [5], so we present a brief R3MAT description. A full R3MAT explanation can be found in [5].

R3MAT is an efficient method to generate power-law graphs of hundred million nodes, running on a single machine. R3MAT defines a two-steps process: first, it creates an array containing the degree distribution of the graph; and second, it generates the corresponding edges.

Specifically, to generate a graph $G = (N, E)$, where N is the set of nodes and E is the set of edges, R3MAT initializes an array D of size $|N|$ where each element $D[i]$ contains the degree of the node i. Then, in order to produce a graph with a power-law distribution with law's exponent 2, R3MAT generates $|E| = \frac{2}{3}|N|\ln|N| + 0.384\,81|N|$ edges following the method shown in Algorithm 1. For each generated edge (i, j), R3MAT increases the value of $D[i]$ for both directed and undirected graphs, and increases the value of $D[j]$ just for undirected graphs. So, at the end of this first step the array D contains the degree distribution of the graph.

The method `GenerateEdge`, shown in Algorithm 1, simulates a recursive partition of the adjacency matrix of the graph (similar to the R-MAT generation method [6]). The variables x_1, y_1 indicate the top-left position of the matrix, and the variables x_n, y_n indicate the bottom-right position. In the first call, the method is executed with parameters $x_1 = 1$, $y_1 = 1$, $x_n = |N|$ and $y_n = |N|$, i.e. the entire matrix. In the subsequent recursive calls, the matrix is divided into four partitions, and one of them is selected according to a given probability (Line 8). When the matrix is tiny enough (e.g. a single element), the coordinates are returned as an edge (see Lines 2 to 6).

In the second step, R3MAT runs over the array D and, for each element $1 \leq x \leq |N|$, it generates $D[x]$ edges of the form (x, y), where the index of the target node y goes in the range $\{1, \ldots, x-1, x+1, \ldots, |N|\}$ for directed graphs, and in the range $\{x+1, \ldots, |N|\}$ for undirected graphs. It means that, for directed graphs, the target nodes are selected by going backward in the array and then by going forward, whereas for undirected graphs it just goes forward.

Algorithm 1 GenerateEdge(Long x_1, Long y_1, Long x_n, Long y_n)

```
1  begin
2      long div_1 ← x_1 − y_1, div_n ← x_n − y_n
3      if div_1 ∈ {−1, 0, 1} ∨ div_n ∈ {−1, 0, 1} then
4          long[] edge ← new long[2]
5          edge[0] ← x_1, edge[1] ← y_1
6          return edge
7      end
8      double(a, b, c, d) ← (0.67, 0.19, 0.10, 0.04)
9      double ab ← a + b, abc ← a + b + c
10     long x_l, y_l, x_r, y_r
11     double r ← random.getDouble()
12     if r < a then
13         x_l ← x_1, y_l ← y_1
14         x_r ← ⌈(x_1 + x_n)/2⌉, y_r ← ⌈(y_1 + y_n)/2⌉
15     else if r < ab then
16         x_l ← ⌊(x_1 + x_n)/2, y_l ← y_1⌋
17         x_r ← x_n, y_r ← ⌈(y_1 + y_n)/2⌉
18     else if r < abc then
19         x_l ← x_1, y_l ← ⌊(y_1 + y_n)/2⌋
20         x_r ← ⌈(x_1 + x_n)/2⌉, y_r ← y_n
21     else
22         x_l ← ⌊(x_1 + x_n)/2⌋, y_l ← ⌊(y_1 + y_n)/2⌋
23         x_r ← x_n, y_r ← y_n
24     end
25     return GenerateEdge(x_l, y_l, x_r, y_r)
26 end
```

III. GRAPH GENERATION METHODS

This section presents H4DG and H4UG, two methods for generating directed and undirected graphs respectively, both designed according to the MapReduce processing model.

The design of H4DG and H4UG is inspired by two key features of the generation model introduced by R3MAT. The first key feature is the use of the array containing the degree distribution. This intermediate structure only needs linear memory, which is reasonable in comparison to the quadratic memory used by the adjacency matrix required by other methods. The second key feature is the two-steps process, which avoids the generation of duplicated edges, and consequently, enables a fair generation of undirected graphs.

A. GENERATION OF DIRECTED GRAPHS

In this section we describe the method to generate directed graphs using MapReduce H4DG. In general terms, the method begins with the creation (in parallel) of an array containing the degree distribution of the graph, and then produce the edges. This is basically the R3MAT method adapted to the MapReduce programming model.

The MapReduce application is divided in three steps: Preparation, Map, and Reduce. Algorithm 2 shows the pseudocode of the preparation step. The method `GenerateDirecteGraph` receives as parameters the expected number of nodes of the graph and the number of workers (machines in the cluster). The number of nodes is used to calculate the number of edges (Line 2), as defined by the R3MAT method, and the number of edges to be produced by each worker (Line 3). For each worker, it creates a file whose content is the number of edges to be generated by the worker (Lines 4 to 8). These files are the input for the

Algorithm 2 GenerateDirectedGraph(Long *n*, Long *w*)

```
   Input: n ← number of nodes of the graph; w ←
          number of workers in the cluster.
   Output: A list of edges.
1  begin
2      long e ← (2/3 · n · ln n) + (0.384 81 · n)
3      long edgesByWorker ← e/w
4      for long i ← 0 to w do
5          String filename ← "inputFile" + i
6          File file ← new File(filename)
7          file.write(edgesByWorker)
8      end
9      conf.setLong("nodes", n)
10     conf.setLong("edges", e)
11     job.setMapperClass(DirectedGraphMapper.class)
12     job.setReducerClass(DirectedGraphReducer.class)
13     job.setMapOutputKeyClass(LongWritable.class)
14     job.setMapOutputValueClass(LongWritable.class)
15     job.waitForCompletition(true)
16 end
```

mapper. The number of nodes and edges are saved as global variables by using the *conf* object provided by MapReduce (Lines 9 and 10).

Algorithm 3 shows the map step pseudocode. The method `DirectedGraphMapper` is executed in parallel for each input file created in the preparation step. First, the number of nodes of the graph is recovered from the global variable "nodes" (Line 2). The number of edges to be generated by the worker is obtained from variable *value* as it is stored in the input file (Line 3). Using the method `GenerateEdge`, see Algorithm 1, the mapper generates *e* edges. For each of them, it creates a pair $(s, 1)$ where s is the source node index. The list of pairs created by the map step is later transformed into an array containing the graph degree distribution.

Algorithm 3 DirectedGraphMapper(LongWritable *key*, Text *value*, Context *context*)

```
1 begin
2     long n ← conf.getLong("nodes")
3     long e ← Long.valueOf(value.toString())
4     for long i ← 0 to e − 1 do
5         long edge[] ← rmat.GenerateEdge(1,1,n,n)
6         LongWritable source ← edge[0]
7         context.write(source, new LongWritable(1))
8     end
9 end
```

Algorithm 4 shows the reduce step pseudocode. Each instance of the method `DirectedGraphReducer` receives a list of edges having the same source node. The source node index is stored in variable *key* and the indexes of the target nodes are stored in variable *values*. Note that the

Algorithm 4 DirectedGraphReducer(LongWritable *key*, Iterable<LongWritable> *values*, Context *context*)

```
1  begin
2      long degree ← 0
3      forall the v ∈ values do
4          degree ← degree + 1
5      end
6      long cnt ← 0
7      long target ← source − 1
8      while cnt < degree ∧ target > 0 do
9          context.write(new LongWritable(source), new
           LongWritable(target))
10         cnt ← cnt + 1
11         target ← target − 1
12     end
13     target ← source + 1
14     while cnt < degree ∧ target < n do
15         context.write(new LongWritable(source), new
           LongWritable(target))
16         cnt ← cnt + 1
17         target ← target + 1
18     end
19 end
```

size of the array *values* corresponds to the *degree* of the node *key*. Hence, the reducer generates *degree* edges for the source node with index *key*, such that the index of the target node is selected first going backward (Lines 8 to 12) and then going forward (Lines 14 to 18).

Figure 1 shows a simulation of the process described above, thought to generate a 16-edge graph, running on a cluster of two worker machines. In the initialization step, the application creates two files, each having the value 8 inside (i.e. the number of edges to be generated for each worker). Let us analyze the process inside Worker 1. The mapper reads an input file and generates eight key-value pairs by using the method `GenerateEdge` shown in Algorithm 1. Although each generated edge is formed by a source node index s and a target node index t, the method just consider the former to create a pair $(s, 1)$. Note that the second worker also generates eight key-value pairs following the same approach. In the Shuffle step, the key-value pairs are grouped by key, and the corresponding values are stored in a list. In the Reduce step, the values for the same key k are added to determine the number of edges to be generated for the source node identified by index k. Finally, the method generates (as output) the corresponding edges for each node k by using the forward and backward traversal described in Section II-B.

B. GENERATION OF UNDIRECTED GRAPHS

In this section we describe H4UG, a method to generate undirected graphs using MapReduce. Although, the method described in Section III-A can be used to generate undirected graphs, it has the problem of generating duplicates in the form

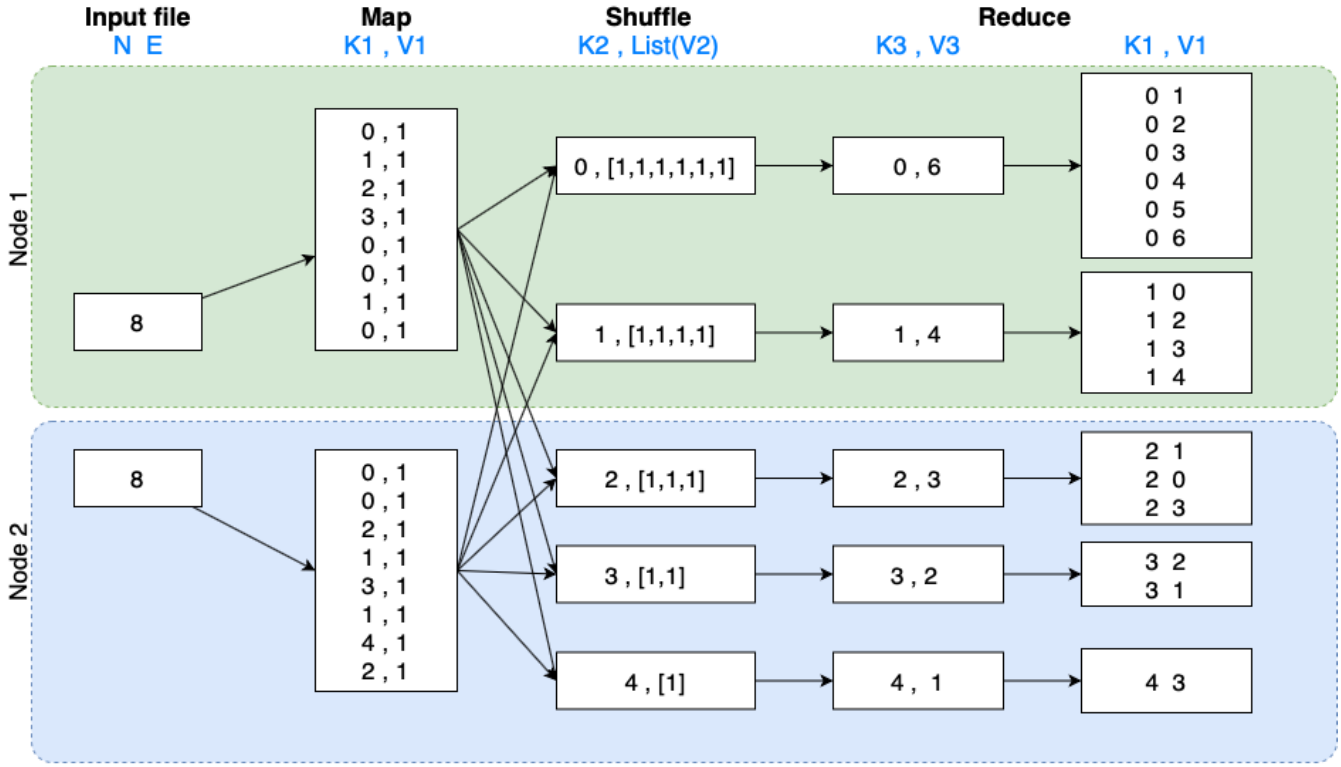


FIGURE 1. Example of the MapReduce process to generate a directed graph.

of symmetric edges. Next we describe the changes required to avoid such problem.

Recall that, during the reduce step, the method for directed graphs generates the final edges by exploring the array containing the degree distribution of the graph. Specifically, for a source node index x, the method generates edges in the set $\{(x, y) \mid 1 \leq y < x\}$, i.e. the backward edges; and edges in the set $\{(x, z) \mid x < z \leq n\}$, i.e. the forward edges. It implies that it is possible to generate symmetric edges, i.e. pairs of edges of the form (a, b) and (b, a).

A naive solution to the problem described above is to remove symmetric edges in a second MapReduce job. Hence, the second job must take the list of edges and apply any algorithm to remove duplicates. Note that this solution increases the complexity and the runtime of the algorithm.

A simple solution to the problem is to avoid the generation of backward edges. It means that, for a source node index x, the target nodes have an index greater than x. Just in the case when the number of subsequent edges is not enough to cover the degree of x (i.e. $D[x] > n-x$), the method produces edges of the form $(x, 1)$, $(x, 2)$, and so on. Note that, the second case has a low probability of occurrence as the last nodes in the array contain a low degree (usually one) in order to satisfy the power-law distribution.

Hence, the MapReduce application for generating undirected graphs is very similar to the one for directed graphs. The difference occurs in the reduce step, whose pseudocode is shown in Algorithm 5. The main characteristic of the `UndirectedGraphReducer` method is the generation of edges (Lines 7 to 15). Note that the variable *target*, that stores the index of the next target node, is initialized with $source+1$ (Line 7). In most cases, the target nodes have an index greater

Algorithm 5 UndirectedGraphReducer(LongWritable *key*, Iterable<LongWritable> *values*, Context *context*)

```
1  begin
2      long degree ← 0
3      forall the v ∈ values do
4          degree ← degree + 1
5      end
6      long cnt ← 0
7      long target ← source + 1
8      while cnt < degree do
9          if target > n then
10             target ← 1
11         end
12         context.write(new LongWritable(source), new
           LongWritable(target))
13         cnt ← cnt + 1
14         target ← target + 1
15     end
16 end
```

than the source node. However, if the amount of forward edges is lower than the number of edges to be produced, then the target node index is changed to 1 (Line 10) and the generation continues until all the edges are generated.

In order to show our method to the generate undirected graphs, consider the process shown in Figure 1. The preparation (Input file), Map, and Shuffle steps are the same, i.e. they produce a list of pairs (k, v), where k is a source node index and v represents the degree of the node k. In the example, these pairs are (0, 6), (1, 4), (2, 3), (3, 2), and (4, 1). In comparison to generate directed graphs, the change is given by the way to generate the edges for each pair (k, v). Specifically, for a pair (k, v), the method generates v edges of the form $(k, k + 1), (k, k + 2), \ldots, (k, k + v)$; switching to $(k, 1), (k, 2)$, etc., if the target node arrives to $n + 1$. For example, the pair (3, 2) produces the edges (3, 4) and (3, 5).

IV. EXPERIMENTAL EVALUATION

This section presents an experimental evaluation of the graph generation methods presented in this article. Specifically, we evaluate the performance of the methods and compare them with the available alternatives in the state of the art. This section includes the evaluation methodology, the experimental results, and the corresponding discussion.

A. METHODOLOGY

Our experimental evaluation includes several experiments, where each experiment implies the execution of a generation method in a cluster of computers. Specifically, each experiment combines the following variables:

- Generation method. We evaluated both methods proposed in this article, namely, H4DG and H4UD. From the state of the art, we initially considered TeGViz, PegasusN and TrillionG. TrillionG runs on top of Apache Spark, so we assumed that it is not comparable with our Apache Hadoop implementations. Hence, we just selected TeGViz and PegasusN as they provided enough information to be evaluated.
- Graph size. We considered the generation of graphs having 10 thousand, 100 thousand, 1 million, 10 million, 100 million, and 1 billion nodes.
- Graph type. It refers to the generation of directed or undirected graphs.
- Cluster size. It defines the number of computers in the cluster. We use clusters composed of a master node plus 8, 16, and 32 worker nodes.
- Cluster power. It indicates the processing power of the computers in the cluster. For each cluster size, we considered three configurations (small, medium, and large).

Considering the above variables, we evaluated the methods in terms of efficiency, scalability, and realism. The *efficiency* refers to the runtime required to generate a graph (of a given size and type) by using a method over a cluster (of a given size and power). The main objective is to determine the fastest and the slowest methods. Additionally, we would like to determine the relationship between efficiency and processing power (determined by cluster size and power).

We perform this evaluation under two notions of scalability. The first one studies the behaviour of the methods after increasing the number of nodes, so as to determine the largest graph the method is able to generate. The second notion is related to an increment of the computational resources, to determine its dependency with regard to the hardware.

The realism of a method is given by its ability to generate graphs whose statistical distribution of edges follows a power-law [31]. Such notion of realism is measured by using the Kolmogorov–Smirnov statistic. Additionally, we produce charts to observe the degree distribution of the graphs.

B. EXPERIMENTAL SETUP

All the experiments of this work are executed in EMR, the cloud big data platform provided by Amazon. EMR allows to create Hadoop clusters composed by several types of virtual machines. We use three types of virtual machines, whose technical specifications are given in Table 1. All the machines are configured with a 100GB SSD storage, and installed with Amazon Linux release 2 (Karoo) as operating system.

TABLE 1. Amazon machines used in the experimental evaluation.

Machine ID	Machine Name	vCPU (number)	RAM (GB)	Network Performance
M1	m4.large	2	8	Moderate
M2	m4.xlarge	4	16	High
M3	m4.2xlarge	8	32	High

Table 2 shows information about the nine clusters used in our experimental evaluation. The clusters differ in size (number of worker machines) and machine type. It is important to mention that, in addition to the worker machines (acting as Hadoop Data Nodes), every cluster includes a master machine (acting as Hadoop Name Node). All the clusters worked with Apache Hadoop 2.10.1 (emr-5.32.0).

TABLE 2. Types of Amazon EMR Hadoop clusters used in the experiments.

Cluster ID	Workers (Machines)	Machine Type	Total vCore	Total RAM
C8S	8	m4.large	16	64
C8M	8	m4.xlarge	32	128
C8L	8	m4.2xlarge	64	256
C16S	16	m4.large	32	128
C16M	16	m4.xlarge	64	256
C16L	16	m4.2xlarge	128	512
C32S	32	m4.large	64	256
C32M	32	m4.xlarge	128	512
C32L	32	m4.2xlarge	256	1 024

The source code of the Hadoop implementation of the methods H4DG and H4UG is available at GitHub (https://github.com/FerLopezGallegos/MapreduceGraphGenerators). The corresponding JAR files are *h4dg.jar* and *h4ug.jar*. These files are executed in Hadoop using instructions of the form

```
hadoop jar h4dg.jar -n <n> -m <w>
```

where <n> indicates the number of nodes of the graph, and <w> indicates the number of worker machines.

C. EXPERIMENTAL RESULTS

An experiment is defined by a specific combination of generation method, graph size, graph type, cluster size, and cluster power. Table 3 shows general information about the graphs generated during the experimental evaluation. Every experiment is executed three times, so we report the average execution time (runtime). Next, we present and discuss the results of our experimental evaluation.

TABLE 3. Graphs generated during the experimental evaluation.

Graph ID	Number of nodes	Number of edges	File Size (MB)
G1	10 000	65 250	0.51
G2	100 000	806 009	7.30
G3	1 000 000	9 595 150	102.50
G4	10 000 000	111 302 071	1 331.00
G5	100 000 000	1 266 526 383	12 664.40
G6	1 000 000 000	14 200 320 557	141 998.34

1) EFFICIENCY

Tables 4 and 5 show the runtimes obtained during the generation of directed and undirected graphs respectively. Note that TeGViz and PegasusN does not appear in Table 5 as they are unable to generate undirected graphs.

In the case of directed graphs (Table 4), we can see that: TeGViz obtains the lowest time for most of the experiments; PegasusN goes in second place, although just for small graphs; and H4DG obtains the highest times in all cases. Analyzing the runtimes obtained for graph G5 with cluster C32L, we can say that PegasusN is 2.5 times slower that TeGViz, and H4DG is 4.8 times slower. Figure 2 allows a visual comparison of the methods by showing the runtimes obtained over the C32L cluster, the most powerful used here. In defense of H4DG, we would like to mention that TeGViz and PegasusN generate graphs with repeated edges. TeGViz provides an option to eliminate duplicates, however it increases its runtime greatly. Also, H4DG is the only method capable to produce the biggest graph G6.

Similarly, we can compare the cluster performance under different graphs. Figures 3, 4, and 5 show the runtimes obtained for each cluster during the generation of the directed graphs. As expected, it applies that the bigger the cluster the lower the runtime. This behavior is clear for big graphs (G5 and G6), but not so obvious for small and mid-size ones (G1 – G4). In this sense, we can say that a small cluster (e.g, C8S) is enough to generate small and mid-size graphs, whereas a large cluster (e.g. C16L) is required to generate big graphs.

2) SCALABILITY

The scalability of the methods is studied from the points of view of both graph size and processing power.

TABLE 4. Runtimes (in seconds) obtained during the generation of directed graphs. The lowest times are in bold.

Cluster	Graph	H4DG	TeGViz	PegasusN
C8S	G1	31.4	**26.6**	31.1
	G2	32.8	**26.6**	32.8
	G3	40.3	**36.5**	43.2
	G4	128.7	**61.6**	105.6
	G5	1 228.9	**304.9**	684.1
	G6	-	-	-
C8M	G1	30.2	21.0	**18.8**
	G2	32.7	**21.1**	25.0
	G3	37.7	**22.8**	26.5
	G4	109.9	**42.7**	55.4
	G5	937.5	**254.4**	396.3
	G6	-	-	-
C8L	G1	27.3	**15.9**	16.5
	G2	27.3	**16.0**	17.0
	G3	33.4	**17.6**	18.4
	G4	76.0	**29.3**	48.2
	G5	737.6	**114.9**	357.3
	G6	-	-	-
C16S	G1	32.5	**32.2**	33.7
	G2	32.8	**31.6**	33.7
	G3	37.6	**34.9**	38.2
	G4	84.8	**54.9**	83.9
	G5	619.4	**239.9**	526.1
	G6	-	-	-
C16M	G1	32.2	26.4	**24.0**
	G2	33.2	31.1	**26.2**
	G3	37.5	**26.2**	32.9
	G4	77.3	**39.5**	56.5
	G5	543.0	**116.2**	282.3
	G6	-	-	-
C16L	G1	28.7	**17.6**	18.1
	G2	29.7	21.0	**18.6**
	G3	31.4	22.1	**19.4**
	G4	63.0	**31.0**	38.0
	G5	390.8	**69.6**	212.8
	G6	**5 211.7**	-	-
C32S	G1	35.3	31.6	**29.7**
	G2	34.7	**31.6**	32.4
	G3	35.6	34.9	**34.2**
	G4	56.9	**46.7**	75.4
	G5	347.7	**139.2**	385.2
	G6	**3 699.9**	-	-
C32M	G1	33.1	**26.2**	28.6
	G2	33.9	**26.4**	28.8
	G3	36.0	**29.6**	32.4
	G4	54.3	**42.9**	50.3
	G5	290.9	**88.8**	196.9
	G6	**3 309.8**	-	-
C32L	G1	28.7	**19.3**	20.1
	G2	28.9	19.4	**19.3**
	G3	30.3	20.9	**20.0**
	G4	41.0	**22.6**	28.7
	G5	219.0	**46.0**	119.2
	G6	**2 512.0**	-	-

Consider the charts shown in Figures 6, 7, and 8. We can observe that the three methods scale well for graphs G1 to G5. However, only H4DG is able to generate the biggest graph G6 (i.e. the one having a billion nodes) when operate over the clusters C16L, C32S, C32M, and C32L.

With respect to the scalability under processing power, let us analyze the combination of methods and clusters. We observe that the runtimes for graphs G1, G2, and G3 are

TABLE 5. Runtimes (in seconds) obtained by H4UG during the generation of undirected graphs.

Cluster	G1	G2	G3	G4	G5	G6
C8S	30.9	32.0	40.7	128.7	1 175.8	-
C8M	30.4	31.8	38.2	118.0	885.0	-
C8L	26.6	27.5	31.6	75.5	634.1	-
C16S	33.7	33.4	38.9	82.1	628.7	-
C16M	26.8	26.4	31.4	74.0	565.2	-
C16L	21.2	21.9	25.7	48.2	339.5	4 723.0
C32S	32.9	32.6	35.5	57.2	325.4	3 771.9
C32M	32.4	32.9	35.4	53.4	288.4	3 338.0
C32L	27.7	27.9	31.7	41.0	229.5	2 404.0

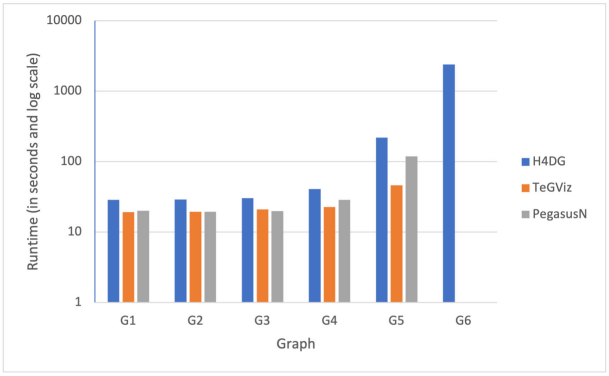


FIGURE 2. Efficiency of the methods using the C32L cluster.

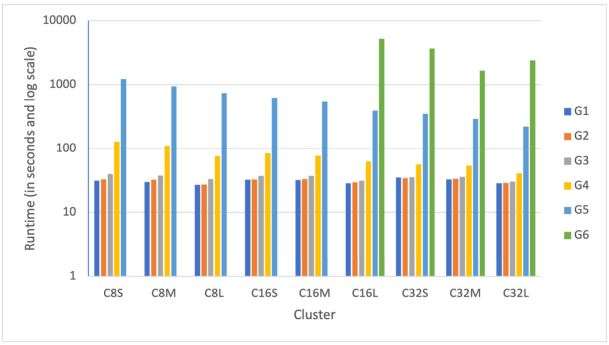


FIGURE 3. Efficiency of clusters with respect to the directed graphs generated with H4DG.

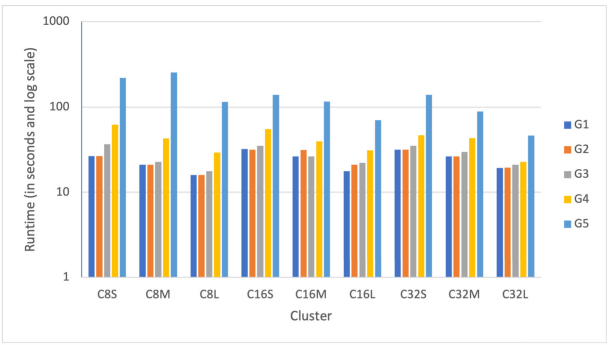


FIGURE 4. Efficiency of clusters with respect to the directed graphs generated with TeGViz.

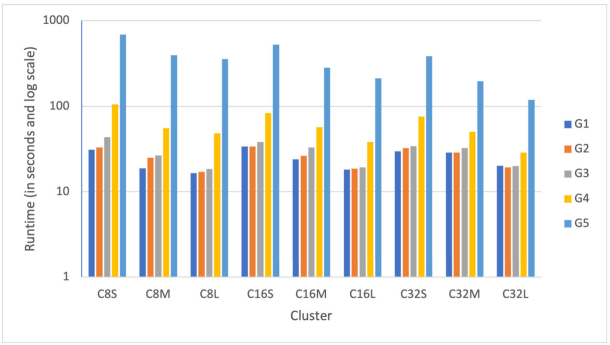


FIGURE 5. Efficiency of clusters with respect to the directed graphs generated with PegasusN.

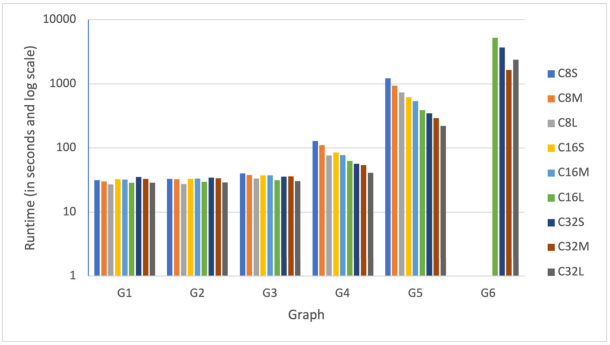


FIGURE 6. Scalability of H4DG under different directed graphs.

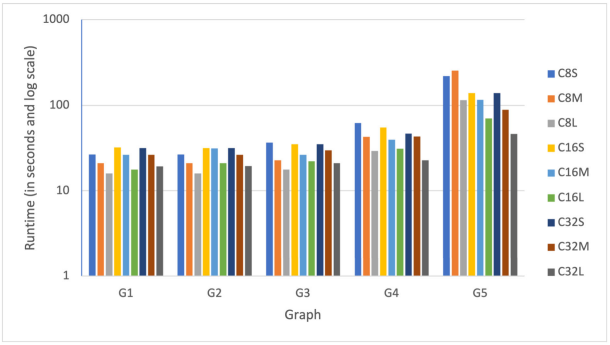


FIGURE 7. Scalability of TeGViz under different directed graphs.

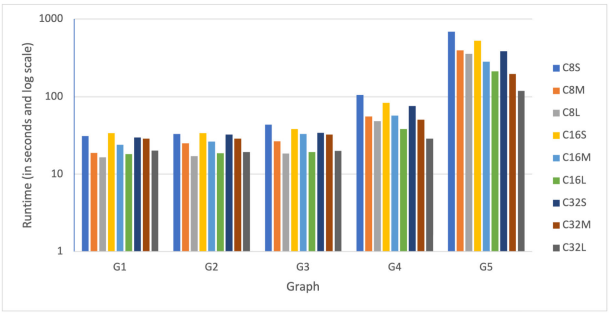
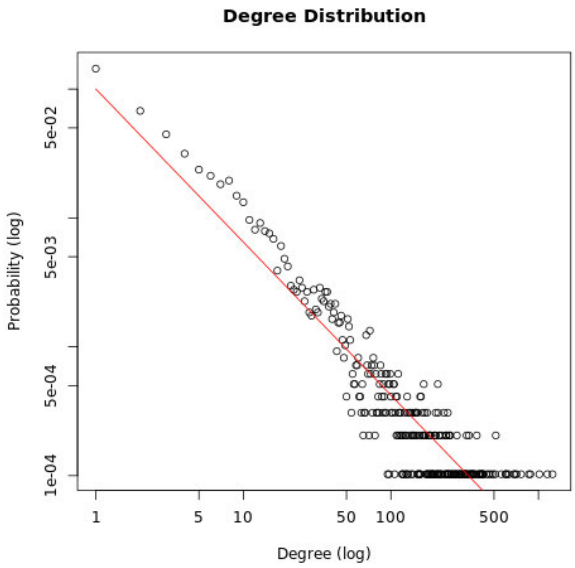
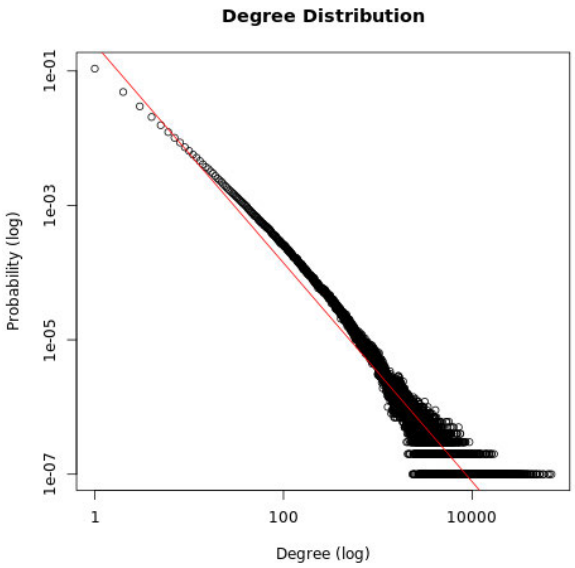


FIGURE 8. Scalability of PegasusN under different directed graphs.

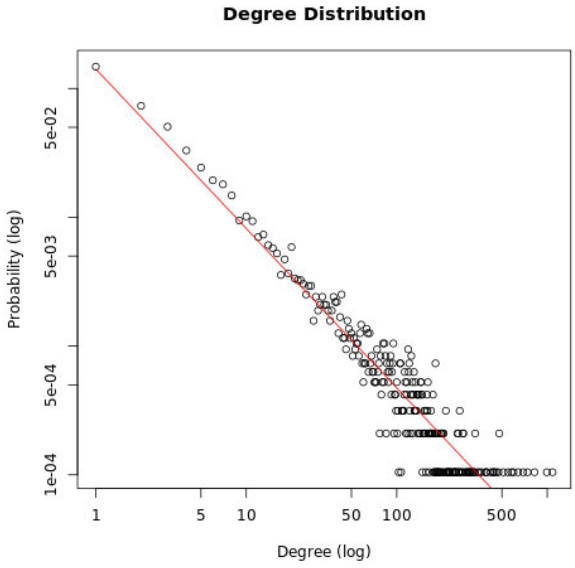
very similar. Hence, we can argue that the cluster processing power is not so relevant for such graphs. However, for graphs G4, G5, and G6, the methods show increasing and variable runtimes, so the type of cluster is relevant in such cases. This analysis is very important from a financial point of view as the size of the cluster determines its cost.



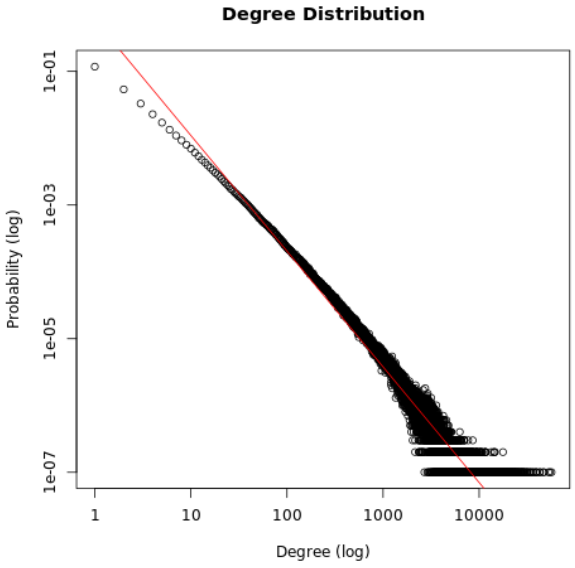
(a) Degree distribution of a graph G1 generated with H4DG.



(b) Degree distribution of a graph G4 generated with H4DG.



(c) Degree distribution of a graph G1 generated with H4UG.



(d) Degree distribution of a graph G4 generated with H4UG.

FIGURE 9. Degree distribution, in logarithmic scale, of four graphs generated with H4DG and H4UG. Plots (a) and (b) show the degree distributions for directed graphs of types G1 (n = 10 thousand nodes) and G4 (n = 10 million nodes), respectively. Plots (c) and (d) show the degree distributions for undirected graphs with the same sizes.

3) REALISM

Finally, we use two methods to evaluate the realism of H4DG and H4UG, that is, their ability to generate graphs whose statistical distribution of edges follows a power-law. The first method is a statistical verification by applying the Kolmogorov–Smirnov statistic (KS-stat). The second one is a visual verification by plotting the graph degree distributions.

The KS-stat metric [31] allows to test whether a sample distribution fits a power-law distribution. In this case, we used the igraph R package[1] to compute the KS-stat for directed and undirected graphs. Table 6 shows the KS-stat scores obtained for graphs G1, G2, G3, and G4 (it was not possible to compute the KS-stat for the rest of the graphs). According to the literature, a small score for KS-stat denotes a better fit between the observed and the power-law distribution. We observe that all the scores have a value lower than 0.09, so it can be argued that all the graphs do follow a power-law distribution. Note

[1] Available at https://igraph.org/r/doc/fit_power_law.html.

TABLE 6. KS-stat scores for directed and undirected graphs generated with H4DG and H4UG, respectively.

Type / Size	G1	G2	G3	G4
Directed	0.080 304	0.038 501	0.032 437	0.023 645
Undirected	0.056 977	0.043 184	0.026 267	0.019 327

also that the scores decrease with the size of the graph, so we can conjecture that graphs larger than 10 million nodes (G4) also maintain the power-law distribution.

For the visual verification, we used the igraph R package to generate scatter plots showing the degree distribution of a graph. Figure 9 shows the plots for directed and undirected graphs of types G1 (ten thousand nodes) and G4 (ten million nodes). Each plot includes the sample distribution (point cloud) and the power-law distribution (red line). In general, we can say that all the sample distributions are very close to the power-law baseline. Note that the distributions differ with respect to the size of the graph (i.e. sparse for the smaller ones). However, they show a very similar shape independent of graph type (directed or undirected).

We would like to mention that we also create the plots for the graphs generated with PegasusN (they are not included here in order to save space). For smaller graphs, PegasusN shown a good distribution. However, for larger graphs the plots shown an erratic distribution with respect to the expected power-law. In the case of TeGViz, it is impossible to obtain the plots because the igraph R package found some errors in the output data files.

V. CONCLUSION

We have presented two methods, H4DG and H4UG, to generate graphs with power-law edge distribution based on MapReduce that can be easily implemented to run on top of Apache Hadoop. H4DG produces directed graphs whilst H4UG produces undirected ones. Our experimental evaluation shows that our methods are efficient (in terms of runtime and computational resources) and scalable (in terms of graph size and cluster capacity). Our comparison with TeGViz and PegasusN shows that our methods are a little less efficient in terms of runtime. However, H4DG is the only method able to generate the largest directed graph in our evaluation (the one with one billion nodes and fourteen billion edges). Moreover, our method H4UG is capable of creating undirected graphs without duplicate edges; a requirement that both TeGViz and PegasusN are unable to satisfy.

As future work, we plan to study other properties of the graphs generated by our methods, such as clustering coefficient and community structure. Our goal is to check and improve the realism of the generator, and produce graphs satisfying the properties found in real networks.

REFERENCES

[1] R. Angles and C. Gutierrez, "An introduction to graph data management," in *Graph Data Management*. Cham, Switzerland: Springer, 2018, pp. 1–32.

[2] A. Prat-Pérez, J. Guisado-Gámez, X. F. Salas, P. Koupy, S. Depner, and B. D. Bartolini, "Towards a property graph generator for benchmarking," in *Proc. Int. Workshop Graph Data-Manage. Exp. Syst.* New York, NY, USA: ACM, 2017, pp. 1–6.

[3] A. Bonifati, I. Holubová, A. Prat-Pérez, and S. Sakr, "Graph generators: State of the art and open challenges," *ACM Comput. Surveys*, vol. 53, no. 2, pp. 1–30, Jul. 2020.

[4] C. C. Aggarwal and H. Wang, *Managing and Mining Graph Data* (Advances in Database Systems), vol. 40. Boston, MA, USA: Springer, 2012.

[5] R. Angles, R. Paredes, and R. Garcia, "R3MAT: A rapid and robust graph generator," *IEEE Access*, vol. 8, pp. 130048–130065, 2020.

[6] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph Mining," in *Proc. SIAM Int. Conf. Data Mining*, Apr. 2004, pp. 442–446.

[7] P. Erdös and A. Rényi, "On random graphs," *Pub. Math., Debrecen*, vol. 6, pp. 290–297, Jan. 1959.

[8] H.-M. Park, C. Park, and U. Kang, "PegasusN: A scalable and versatile graph mining system," in *Proc. Conf. Artif. Intell.*, 2018, pp. 8214–8215.

[9] *PegasusN*. Accessed: May 15, 2021. [Online]. Available: https://datalab.snu.ac.kr/pegasusn/

[10] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos, "Realistic, mathematically tractable graph generation and evolution, using Kronecker multiplication," in *Proc. Eur. Conf. Princ. Pract. Knowl. Discovery Databases (PKDD)* (Lecture Notes in Computer Science), vol. 3721. Berlin, Germany: Springer-Verlag, 2005, pp. 133–145.

[11] S. J. Plimpton and K. D. Devine, "MapReduce in MPI for large-scale graph algorithms," *Parallel Comput.*, vol. 37, no. 9, pp. 610–632, Sep. 2011.

[12] K. Ueno and T. Suzumura, "Highly scalable graph search for the graph 500 benchmark," in *Proc. Int. Symp. High-Perform. Parallel Distrib. Comput.* New York, NY, USA: ACM, 2012, pp. 149–160.

[13] *Graph 500–Large-Scale Benchmarks*. Accessed: May 15, 2021. [Online]. Available: https://graph500.org/

[14] M.-D. Pham, P. Boncz, and O. Erling, "S3G2: A scalable structure-correlated social graph generator," in *Selected Topics in Performance Evaluation and Benchmarking* (Lecture Notes in Computer Science), vol. 7755. Berlin, Germany: Springer, 2013, pp. 156–172.

[15] P. Boncz, M.-D. Pham, O. Erling, I. Mikhailov, and Y. Rankka. (2011). *Social Network Intelligence Benchmark*. [Online]. Available: https://www.w3.org/wiki/Social_Network_Intelligence_BenchMark

[16] C. Seshadhri, T. G. Kolda, and A. Pinar, "Community structure and scale-free collections of Erdos-Rényi graphs," *Phys. Rev. E, Stat. Phys. Plasmas Fluids Relat. Interdiscip. Top.*, vol. 85, no. 5, May 2012, Art. no. 056109.

[17] *FEASTPACK Distribution*. Accessed: May 15, 2021. [Online]. Available: https://www.sandia.gov/~tgkolda/feastpack/#5

[18] T. G. Kolda, A. Pinar, T. Plantenga, and C. Seshadhri, "A scalable generative graph model with community structure," *SIAM J. Sci. Comput.*, vol. 36, no. 5, pp. C424–C452, Jan. 2014.

[19] M. Dayarathna and T. Suzumura, "Graph database benchmarking on cloud environments with XGDBench," *Automated Softw. Eng.*, vol. 21, no. 4, pp. 509–533, Dec. 2014.

[20] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Scalable SIMD-efficient graph processing on GPUs," in *Proc. Int. Conf. Parallel Archit. Compilation (PACT)*, Oct. 2015, pp. 39–50.

[21] *PaRMAT*. Accessed: May 15, 2021. [Online]. Available: https://github.com/farkhor/PaRMAT

[22] B. Jeon, I. Jeon, and U. Kang, "TeGViz: Distributed tera-scale graph generation and visualization," in *Proc. IEEE Int. Conf. Data Mining Workshop (ICDMW)*, Nov. 2015, pp. 1620–1623.

[23] *TeGViz*. Accessed: May 15, 2021. [Online]. Available: https://datalab.snu.ac.kr/tegviz/

[24] S. Salloum, R. Dautov, X. Chen, P. X. Peng, and J. Z. Huang, "Big data analytics on Apache Spark," *Int. J. Data Sci. Anal.*, vol. 1, no. 3, pp. 145–164, Nov. 2016.

[25] *GraphX Programming Guide*. Accessed: May 15, 2021. [Online]. Available: https://spark.apache.org/docs/latest/graphx-programming-guide.html

[26] S. Edunov, D. Logothetis, C. Wang, A. Ching, and M. Kabiljo, "Darwini: Generating realistic large-scale social graphs," 2016, *arXiv:1610.00664*. [Online]. Available: http://arxiv.org/abs/1610.00664

[27] S. Edunov. *Implementation of Darwini Graph Generator*. Accessed: May 15, 2021. [Online]. Available: https://issues.apache.org/jira/browse/GIRAPH-1043

[28] H. Park and M.-S. Kim, "TrillionG: A trillion-scale synthetic graph generator using a recursive vector model," in *Proc. ACM Int. Conf. Manage. Data*. New York, NY, USA: ACM, May 2017, pp. 913–928.

[29] H. Park. *TrillionG*. Accessed: May 15, 2021. [Online]. Available: https://github.com/chan150/TrillionG

[30] A. Prat-Pérez. *DataSynth*. Accessed: May 15, 2021. [Online]. Available: https://github.com/DAMA-UPC/DataSynth

[31] A. Clauset, C. R. Shalizi, and M. E. J. Newman, "Power-law distributions in empirical data," *SIAM Rev.*, vol. 51, no. 4, pp. 661–703, Nov. 2009.

RENZO ANGLES received the bachelor's degree in systems engineering from the Universidad Católica de Santa María, Arequipa, Perú, and the Ph.D. degree in computer science from the Universidad de Chile, in 2009. In 2013, he was a Postdoctoral Researcher with the Department of Computer Science, VU University Amsterdam, as part of his participation at the Linked Data Benchmark Council Project. He is currently an Assistant Professor with the Department of Computer Science, Universidad de Talca, Chile. He is also participates as a Researcher with the Millennium Institute for Foundational Research on Data, Chile. Specifically, he works in the theory and design of graph query languages, and the interoperability between RDF and graph databases. His research interests include intersection of graph databases and Semantic web.

FERNANDA LÓPEZ-GALLEGOS received the bachelor's degree in computer science from the Universidad de Talca, in 2020. In 2020, she worked with the Millennium Institute for Foundational Research on Data, Chile. Her research interests include big data and graph data management.

RODRIGO PAREDES (Member, IEEE) received the M.Sc. and Ph.D. degrees in electrical engineering and computer science from the Universidad de Chile, in 2008. He is currently an Assistant Professor with the Department of Computer Science, Universidad de Talca, Chile. His main research interests include metric space indexing, graph algorithms, and experimental algorithms in general. In 2009, he was a recipient of the Annual Award "JCC Invited Talk" from the Chilean Computer Science Society (SCCC) to the best performing Chilean researcher with a recent Ph.D. in computer science and the Second prize in the 2003 CLEI-UNESCO Latin American Computer Science Master Theses Contest. He was the Coach of the first Chilean Team classified to the ACM-ICPC World Finals, in 2007.