ABSTRACT One of the main problems when developing graph-based applications is the availability of large and representative datasets. The lack of real graphs has motivated the development of software tools for generating synthetic graphs. R-MAT is a data generation method that was designed to produce synthetic graphs whose characteristics resemble those occurring in real networks. Although the generation model defined by R-MAT is easy to understand, its implementation is not trivial and it has intrinsic memory restrictions that makes the generation of very large graphs difficult. This paper studies the practical implementation of R-MAT. We discuss the issues of the original implementation which works with the adjacency matrix of the graph and analyze the size of the resulting graph obtained with the R-MAT model. Then, we introduce and experimentally evaluate R³MAT, an alternative implementation for R-MAT based on an array of degrees. These experiments show that (i) our R³MAT is able to generate graphs of hundred million nodes and billion edges in a single machine, (ii) our method preserves the characteristic power-law distribution of the edge degrees present in real-world graphs, and (iii) R³MAT has the best performance in the current state of the art, when considering a single modest computer in a sequential fashion.

INDEX TERMS Network theory (graphs), data engineering, performance analysis.

I. INTRODUCTION

Graphs are omnipresent in our lives and have been largely studied in mathematics and computer science. Currently, they are intensively utilized in prominent application domains like big data [1], social network analysis [2], information retrieval [3], machine learning [4], and so on.

One of the main problems when developing graph-based applications is the availability of large and representative datasets, because data is a very valuable resource for organizations. The lack of real graphs has motivated the development of methods and tools for generating synthetic graphs [5], most of them based on random procedures [6].

Several methods for graph generation are available in the literature [7]–[9]. The challenge for these methods is the generation of very large graphs that resemble the properties occurring in real networks [10]. Among those properties we can mention: data correlations [11], communities [12], [13], degree distributions [14], clustering [15], and ranking [3].

The Recursive MATrix (R-MAT) [16] is a graph data generation method designed to satisfy three main properties: parsimony, realism, and speed. Parsimony, in the sense that R-MAT is executed with few parameters. Realism, such that it resembles the properties observed in real-world graphs (specifically, power law distribution, community structure and small diameter). Speed, in terms of generating graphs quickly. These features allow using R-MAT in diverse application areas, including database benchmarking [17]–[19], social network analysis [20], [21], biological network analysis [22], signal processing [23], and data visualization [24].

The R-MAT algorithm begins with an empty adjacency matrix of the graph. The basic idea is to recursively subdivide the matrix into four partitions, and then select a partition depending on a given probability (i.e., each partition has its own probability). When a cell is reached (i.e., a 1 × 1 matrix), an edge is generated according to the nodes linked by the cell. This process is repeated to generate a given number of edges.

Although the R-MAT method is easy to understand, there are several issues that make its implementation and applicability difficult. Among these issues we can mention:

- Number of nodes. The original definition of R-MAT indicates that the number of nodes in the graph should be $2^k$. It simplifies the edge selection as we always obtain
a $1 \times 1$ matrix after $k$ subdivisions. However, it restricts the applicability of the method.

- **Number of edges.** One drawback of R-MAT is that the final number of edges is not realized until the generation process is complete. Note that the number of edges influences the properties of the resulting graph (e.g., its power law distribution).

- **Duplicated edges.** The original R-MAT algorithm may produce duplicated edges, i.e. edges which fall into the same cell in the adjacency matrix. This issue affects the efficiency of the method, and there is no precise solution for the original algorithm.

- **Partition probabilities.** R-MAT does not define specific values for the partition probabilities. Moreover, it is well known that they affect the features of the resulting graph, in particular the degree distribution.

We observed that the size of the adjacency matrix affects the scalability of R-MAT. In order to test the limits to create a $n \times n$ Java matrix, we run experiments over a 64GB virtual machine. We discovered an upper bound of $n = 40,524$ under a usual execution and $n = 79,740$ after setting the maximum Java heap size to 64GB. Additionally, we evaluated a Java implementation of R-MAT, constructed following the basic indications presented in [16]. Under a usual execution, we were able to generate a directed graph of 10,000 nodes and 63,326 edges. To accomplish the generation of a directed graph of 100,000 nodes, we had to set the Java heap size to 64GB. A million node graph was not possible to generate.

The main contributions of this paper are two. First, we present R$^3$MAT, a Rapid and Robust implementation of R-MAT that allows to create large graphs in a single computer with reduced memory resources. The key of R$^3$MAT is the use of an array containing the degree distribution of the graph (instead of the original adjacency matrix). Second, we analyze the power law distribution in the context of R-MAT and define an explicit, closed-form formula to calculate the number of edges required to generate a proper graph. Additionally, the experimental evaluation shows that our R$^3$MAT method has the best performance with respect to the state of the art, when considering a single computer with small amount on main memory in a sequential fashion.

The rest of the paper is organized as follows. Section II contains the foundations of R-MAT. Section III presents the related work. Section IV describes a naïve implementation of R-MAT, restricted to generate graphs whose adjacency matrix fits in main memory. Section V contains our analysis about the power law distributions generated by R-MAT. Section VI describes R$^3$MAT, our scalable array-based implementation of R-MAT. Section VII contains the empirical evaluation. Concluding remarks are presented in Section VIII.

### II. R-MAT

Assume that a graph is a pair $G = (N, E)$ where $N$ is a set of $n = |N|$ nodes and $E$ is a set of $m = |E|$ (directed or undirected) edges. An edge is given by a pair of nodes $(u, v)$, so $u$ and $v$ are called adjacent. If $e = (u, v)$ is a directed edge then $u$ (v) is called the source (target) node and $e$ is an outgoing (incoming) edge of $u$ (v) (respectively). If $e$ is undirected then $(u, v)$ is equivalent to $(v, u)$. A graph is (un)directed when it contains just (un)directed edges (respectively).

The degree of a node $u$ in a graph $G$ is the number of edges incident to $u$. If $G$ is directed, then the in-degree and the out-degree of $u$ is the number of incoming edges and outgoing edges of $u$, respectively. Let $P(k)$ be the fraction of nodes in $G$ with degree $k$. The degree distribution of $G$ is defined as the sequence $P(1), P(2), \ldots, P(n - 1)$ (we are considering neither isolated nodes nor loop edges $(u, u)$). Hence, the degree distribution is the probability distribution of the degrees for all the nodes in the graph.

Given a graph $G = (N, E)$, its adjacency matrix $A$ is an $n \times n$ matrix where each matrix entry $a_{u,v}$ indicates the occurrence of an edge between nodes $u$ and $v$. Specifically, an entry $a_{u,v} = 1$ denotes the existence of an edge $(u, v)$ and 0 otherwise.

In order to generate a graph $G$, R-MAT defines a recursive procedure to add edges in the graph adjacency matrix, starting from an empty one, until $m$ edges have been added. The recursive procedure subdivides the adjacency matrix into four partitions (named $a, b, c$, and $d$), where the selection of partition $x$ follows a probability $\rho_x$ satisfying that $\rho_a + \rho_b + \rho_c + \rho_d = 1$, and adds an edge $a_{u,v} = 1$ at the end of the recursion (i.e., when a $1 \times 1$ matrix has been reached). Figure 1 shows an example of an $8 \times 8$ adjacency matrix and the partitions generated by the recursive method.

#### FIGURE 1. Example of adjacency matrix generated by the R-MAT model.

The original definition of R-MAT assumes two important conditions to generate a graph: (i) its number of nodes $n$ should be a power of 2 and (ii) its number of edges $m$ is a parameter defined by the user. The first condition is a little restrictive with regard to produce graphs with any number of nodes. The second one is particularly relevant since it influences the expected power law distribution of the graph. On the one hand, the final number of edges can be lower than $m$ due to the occurrence of duplicate edges during the process. On the other hand, the degree distribution could be generated by a pair of nodes $(u, v)$.
binomial when the number of edges is excessive (because a denser graph is generated).

Additionally, the method suggests to mildly modify the four probabilities $\rho_e$ at each stage of the recursion, with the sake of equalizing them (i.e., $\rho_a = \rho_b = \rho_c = \rho_d = 0.25$) when the final stage of the recursion is reached. All the issues described above are discussed in Section IV.

### III. RELATED WORK

The features of R-MAT have been reviewed in several papers and books, in particular those related to graph mining [8], [9]. Additionally, R-MAT is used in different tools and frameworks for graph analysis. Next, we review the related work in chronological order.

In [7], the authors conjecture that the matrix partition ratios $a : b$ and $a : c$ are approximately $3 : 1$. Hence, the partition probabilities could be calculated accordingly. The skew in the partition probabilities leads to lognormal and Discrete Gaussian Exponential (DGX) distributions, which can successfully model both power law and unimodal distributions. Also, this work empirically shows that R-MAT can match power law distributions and also deviations from power laws.

Boost [25] is a set of libraries for the C++ programming language which is extensively used by the developer community. Boost includes a template class that implements a parallel version of R-MAT. The user is able to specify the number of nodes, the number of edges, the partition probabilities, and a random number generator.

Groër et al. [26] conducted a mathematical analysis of the degree distributions of graphs generated by R-MAT. The exact and asymptotic formulas for the degree distributions are obtained by modeling the creation of edges as an occupancy problem from probability theory. The authors also provide a formula to estimate the mean and the variance of the number of edges in the final graph.

Graph500 [19], [27] is a benchmark for supercomputers based on large-scale graph analysis. The graph generator of Graph500 is a Kronecker generator similar to R-MAT. The default partition probabilities are 0.57, 0.19, 0.19 and 0.05. The generator can create multiple edges between two nodes, self-loops, and isolated nodes. The reference implementation allows to run the data generator in parallel.

PaRMAT [28] is an R-MAT implementation designed to exploit multiple threads and to work under main memory (RAM) restrictions. During the graph generation, duplicated edges are possible.

TeGViz [29] is a tool to generate and visualize tera-scale graphs. It includes an implementation of R-MAT that runs on distributed systems. Specifically, TeGViz follows the MapReduce processing model and runs over Apache Hadoop. The authors argued that TeGViz outperforms other generation methods in terms of scalability and processes billion-scale graphs in less than a minute.

The Stanford Network Analysis Platform (SNAP) [30] implements different graph generation methods for single big-memory machines. The input of the generation function for R-MAT (GenRMat) are the number of nodes, number of edges, partition probabilities, and a random number generator. The output is a directed graph. There is no more information about the implementation.

TrillionG [31] is a disk-based graph generator that implements both R-MAT and Kronecker generation models on top of Apache Spark. TrillionG is based in a new graph generation model called the recursive vector model, which solves the space and time complexity problems existing in R-MAT and Kronecker methods. The authors conducted experiments to show that TrillionG outperforms the state-of-the-art graph generators by up to orders of magnitude.

PegasuSN [32] is a graph mining system that runs on Hadoop and Spark. It provides large scale algorithms for three generation methods, including R-MAT. The generation function for Spark allows to define the number of edges, degree of parallelism (the number of tasks), partition probabilities, and noise (for smoothing). The generation function for Hadoop adds the number of reduce tasks.

In [33], the authors present a method to reduce the R-MAT algorithm from logarithmic to constant time per edge. The basic idea is that a logarithmic number of address bits can be generated in each iteration without changing the underlying process. Additionally, they show that the algorithm can be implemented in an embarrassingly parallel way.

The review of the related work allows us to conclude the following. The characteristics of R-MAT have been studied and validated in several works, in particular its ability to generate power law distributions [7], [26]. The current implementations presents several drawbacks, including the generation of duplicated edges, self-loops, isolated nodes, and the restriction to directed graphs [19], [27], [28], [30]. In this paper we provide an implementation that solves these problems. Most implementations allow to define the number of nodes, the number of edges, and the partition probabilities as input. However, the influence of these parameters and their correlations have not been studied with detail. In this sense, we realize that the number of edges cannot be arbitrary, and we provide an explicit, closed-form formula to estimate a number that ensures the production of graphs with power law distributions (which is simpler than the one shown in [26]). In order to improve the generation time, there are proposal to execute R-MAT in parallel/distributed environments [19], [25], [27]–[29], [31], [32] or by using big-memory machines [30]. In contrast, we developed a sequential implementation that allows to generate large graphs in small-memory machines, with a reasonable runtime. Finally, [33] gives an algorithm to quickly generate a graph with R-MAT, but it suffers from the same drawbacks of the other alternatives.

Given that our R$^3$MAT runs on a single machine in a sequential fashion, in the experimental evaluation (Section VII), we consider Boost [25], PaRMAT [28], and SNAP [30], as they are the available sequential alternatives that are properly comparable with our method.
Algorithm 1 R-MAT(long n, String T)

Input: n ← number of nodes of the graph. T ← “D” (directed) or “U” (undirected) graph. Output: A matrix M (i.e. the adjacency matrix).

begin
  double (ρα, ρb, ρc, ρd) ← (0.67, 0.19, 0.10, 0.04)
  if T = “U” then
    (ρα, ρb, ρc, ρd) ← (0.74, 0.17, 0.08, 0.01)
  double Δa ← (0.25 − ρα)/[log₂ n]
  double Δb ← (0.25 − ρb)/[log₂ n]
  double Δc ← (0.25 − ρc)/[log₂ n]
  double Δd ← (0.25 − ρd)/[log₂ n]
  long M[1][1] ← new long[n][n]
  long m = 2^0.6667 · n · ln n + 0.38481 · n
  long e ← 0, long[] edge
  // Matrix initialization: each node has degree 1
  for long i ← 0 to n − 1 do
    edge ← ChooseEdge(1, 1, n, n, ρα, ρb, ρc, ρd, Δa, Δb, Δc, Δd)
    long j ← edge[0] − 1 // node id in [0, n − 1]
    if T = “D” then M[j][j] ← 1, e ← e + 1
    else if (T = “U”) ∧ (M[j][j] = 0) then
      M[i][j] ← 1, M[j][i] ← 1, e ← e + 1
  end
  // Fill the matrix
  while e ≤ m do
    edge ← ChooseEdge(1, 1, n, n, ρα, ρb, ρc, ρd, Δa, Δb, Δc, Δd)
    long s ← edge[0] − 1, long t ← edge[1] − 1
    if M[s][t] = 0 then
      M[s][t] ← 1, e ← e + 1
    if T = “U” then M[i][j] ← 1
  end
  return M
end

IV. A BASIC IMPLEMENTATION OF R-MAT

A simpler and intuitive implementation of the R-MAT model consists in creating the empty adjacency matrix and filling the cells following the recursive method. A pseudocode of the implementation is given by Algorithms 1 and 2. Next, we explain both algorithms putting special interest on the issues described in Section II.

A. NUMBER OF NODES

R-MAT defines that the number of nodes n must be a power of 2. Although it facilitates the generation of partitions, we skip this restriction and allow any positive integer for n. In our experiments (Section VII), we verify that this assumption does not affect the generation process.

For simplicity, we assume that a node identifier is in the range [0, n − 1], where n is the number of nodes in the graph.

Note that the number of nodes is the first parameter of the R-MAT function defined in Algorithm 1.

Moreover, the generation method does ensure that every node has at least one edge, i.e., isolated nodes could never occur in the generated graph. In order to obtain a connected graph, we introduce a matrix initialization step (Algorithm 1, Lines 12 to 20) where a single edge is selected for each node. This initial step is slightly different for undirected and directed graphs, and could be removed without affecting the normal procedure (but losing the guaranty of connectivity).
B. NUMBER OF EDGES
The original definition of R-MAT remarks that the total number of edges \( m \) is not realized until the generation is finalized. However, an appropriate number of edges is very important to obtain a power law distribution. Recall that a power law means that there is a small number of nodes with a large degree, and a large number of nodes with a small degree. The fact is that a large number of edges reduces the number of nodes with small degree. Hence, the higher value for \( m \), the less power law distribution the graph reflects.

Studies on the evolution of real networks, and in particular social networks, have shown that the number of edges grows super-linearly with the number of nodes \( n \) and below \( n^2 \) [34]. Chakrabarti and Faloutsos [7] give several examples of power laws in the real world, showing that the power law exponent varies in the range \([1.92, 2.72]\).

In Section V, we analyze the power law distribution considering any positive value for the exponent, obtaining that for law’s exponent 2 the number of edges is \( m = \frac{3}{2} n \ln n + 0.38481n \) (see Algorithm 1, Line 9). We use this expression for \( m \) as it is the simplest one in the valid range and has small approximation error according to our preliminary numeric simulations.

C. DIRECTED VS UNDIRECTED EDGES
Although R-MAT was mainly thought to generate directed graphs, the original article [16] (briefly) describes an extension to generate undirected ones. Such proposal assumes that probabilities \( \rho_a \) and \( \rho_c \) are equal, so the final adjacency matrix can be adjusted by applying a “clip-and-flip” operation. We propose a less restrictive and simpler method, based on marking symmetric edges, i.e., \((u, v)\) and \((v, u)\), where any of them is selected (see Algorithm 1, Lines 18 and 28).

D. MATRIX PARTITIONING
The function ChooseEdge, defined in Algorithm 2, is the one that implements the partitioning method and returns a new possible edge (note that we say “possible” because we need to check whether the edge has been previously generated). The recursive method is based on modeling the current partition in terms of matrix coordinates and adjust them in each recursive call (see Algorithm 2, Lines 15 to 27). Hence, the first call of ChooseEdge (see Algorithm 1, Lines 13 or 23) defines \((1, 1)\) and \((n, n)\) as the initial coordinates (i.e., the entire matrix). These coordinates are adjusted by the function depending on a random number.

E. PROBABILITY SELECTION
The selection of the partition probabilities \((\rho_a, \rho_b, \rho_c, \rho_d)\) is a very important issue, as they are fundamental to obtain a power law distribution. In the original definition of R-MAT [16] the authors do not provide a concrete combination of probabilities, just conjecture that the \( \rho_a \), \( \rho_b \) and \( \rho_c \), \( \rho_d \) ratios are approximately 75:25.

In order to evaluate the impact of the partition probabilities, we tested different combinations mentioned in the related work. For each combination, we generate\(^2\) a doubly logarithmic plot to visualize the shape of the degree distribution. Our “visual” evaluation shows different degree distributions, and in some cases, the shape does not correspond to a power law distribution.

Although we can evaluate a sample distribution by “visualizing” the approximately straight-line behavior on a doubly logarithmic plot, we would like to have a statistical measure to test how well a power law distribution fits our sample data. In this sense, the Kolmogorov–Smirnov statistic (KS-stat) [35] allows to compare a sample distribution with a reference probability distribution (e.g. power law) by quantifying the distance between them. Smaller scores for KS-stat denote better fit between the distributions under comparison.

In order to determine the best combination of probabilities, we calculate\(^3\) the KS-stat for different combinations and graph sizes (expressed in number of nodes). Specifically, we use three graphs of sizes\(^4\) \( n_1 = 100, n_2 = 1000 \) and \( n_3 = 10,000 \), and considered 1375 combinations with probabilities in the ranges \( 0.65 \geq \rho_a \geq 0.75, 0.05 \geq \rho_b \geq 0.20, 0.05 \geq \rho_c \geq 0.20, 0.01 \geq \rho_d \geq 0.10 \), using an increment of 0.01. For each combination \( c \), we calculate the weighted average KS-stat by using the formula

\[
\overline{KS}(c) = \frac{\overline{ks_1} \times \log_{10}(n_1) + \overline{ks_2} \times \log_{10}(n_2) + \overline{ks_3} \times \log_{10}(n_3)}{\log_{10}(n_1) + \log_{10}(n_2) + \log_{10}(n_3)}
\]

where \( \overline{ks_i} \) corresponds to the KS-stat score of combination \( c \) obtained with the graph of size \( n_i \).

For the case of directed graphs, the best combination is \((0.67, 0.19, 0.10, 0.04)\) with a weighted average KS-stat \( \overline{KS}(c) = 0.0675 \). For undirected graphs, the best combination is \((0.74, 0.17, 0.08, 0.01)\) with \( \overline{KS}(c) = 0.0826 \). By using a “visual” evaluation, we verify that these combinations allow to generate directed and undirected graphs following a power law distribution. Hence, these combinations are used by default in our basic implementation of R-MAT (see Algorithm 1, Lines 2 and 3).

F. PROBABILITY EQUALIZATION
An additional issue of the basic R-MAT algorithm was the generation of bumpy histograms. After some experiments, we realized that such defects can be smoothed by updating the probabilities \( \rho_c \) at each step of the recursion (as suggested in [16]). Hence, we introduced a method to equalize the partition probabilities.

\(^1\)As ChooseEdge returns values in the range 1 to \( n \), but identifiers in the array \( D \) go from 0 to \( n - 1 \), we simply reduce their values by 1, see Algorithm 1, Line 25.

\(^2\)The plots were created with the igraph R package (https://igraph.org/r/).

\(^3\)The KS statistic was computed by using the function fit_power_law of the igraph R package (https://igraph.org/r/doc/fit_power_law.html).

\(^4\)The matrix-based implementation is not able to generate graphs larger than \( n \approx 30,000 \) nodes.
First, for each probability $\rho_i$, we calculate a change value $\Delta x_i = (0.25 - \rho_i) / \log_2 n$ (see Algorithm 1, Lines 4 to 7). The value of $\Delta x$ is used to increment or decrement the probability at each recursion step (see Algorithm 2, Lines 28 and 29). Note that each probability is updated $\log_2 n$ times according to the height of the recursion tree. Hence, the probability $\rho_i$ at stage $i$ of the recursion is given by $\rho_i^t = \rho_i^{t-1} + (0.25 - \rho_i^0) / \log_2 n$. With this procedure, the probability $\rho_i$ in the last recursion step is 0.25.

We evaluate the performance of the basic implementation described above. The results are presented in Section VII.

G. ANALYSIS OF BASIC R-MAT

The execution time of basic R-MAT depends on two key factors, namely, the time needed by the function ChooseEdge and the number of edges in the graph.

On the one hand, each time that ChooseEdge is executed in Algorithm 1 (Lines 13 or 23) the number of recursive steps is accounted by the recurrence relation $t(n) = t(n/2) + 1$, $t(1) = 1$, which implies that ChooseEdge needs $O(\log n)$ steps.

On the other hand, real world graphs are sparse, so we can neglect the probability of choosing the same edge twice. Thus, considering that R-MAT runs $m$ times the function ChooseEdge, for $m = \left\lfloor \frac{1}{2} \log n + 0.3848 n \right\rfloor$, the time needed by basic R-MAT is $O(n \log^2 n)$. Naturally, it also needs $O(n^2)$ memory space.

V. ON THE POWER LAW DISTRIBUTION AND R-MAT

Given a discrete random variable $k \in \mathbb{N}$, the power law establishes that its probability is proportional to $k^{-\tau}$, that is $P(x = k) \propto k^{-\tau}$, for some $\tau > 0$, where $\tau$ is the law's exponent. This can be rewritten as $P(x = k) = Ck^{-\tau}$, where $C$ is a normalization factor so that $\sum_{k=1}^{\infty} P(x = k) = 1$.

In terms of the R-MAT method, we can interpret the power law as the probability that a node has $k$ neighbors is $Ck^{-\tau}$. Therefore, the number of nodes having $k$ neighbors is $n Ck^{-\tau}$, which implies that all of these nodes add $nk Ck^{-\tau} = nk C^{-1} k^{-\tau}$ edges to the graph.

As the sum of all the probabilities has to be one, assuming that there is no isolated nodes, we add the probabilities that a node has $k$ neighbors from 1 to $n - 1$ and write

$$C \sum_{k=1}^{n-1} k^{-\tau} = 1,$$  

so, we need to compute the summation of (1) to determine the normalization factor C. Similarly, the number of nodes of a graph $n$ has to match the summation $n C \sum_{k=1}^{n-1} k^{-\tau}$. Finally, the number of edges $m$ is computed as follows,

$$m = n C \sum_{k=1}^{n-1} k^{-\tau}.$$

For technical reasons, we split the study in three different cases, namely, when $\tau \in (0, 1)$, $\tau = 1$, and $\tau > 1$.

A. FIRST CASE $\tau \in (0, 1)$

We approximate the summations by integrals [36]. As $\tau \in (0, 1)$, the function $k^{-\tau}$ is monotonically decreasing, so we approximate the summation $\sum_{k=1}^{n-1} k^{-\tau}$ as follows,

$$\int_{1}^{n} k^{-\tau} \, dk < \sum_{k=1}^{n-1} k^{-\tau} < 1 + \int_{1}^{n} k^{-\tau} \, dk.$$  

The integral at the left hand is $(n^{1-\tau} - 1)/(1 - \tau)$. At the right hand, the term “1 + ” is used due to it is the exact value of $k^{-\tau}$ when $k = 1$, so, this improve our approximation; and the integral is $((n - 1)^{1-\tau} - 1)/(1 - \tau)$, which using Taylor approximations is $(n^{1-\tau} - (1 - \tau)/n^{\tau - 1} - 1)/(1 - \tau)$. As $\lim_{n \to \infty} (1 - 1/n^{\tau}) = 0$, we neglect this term, so finally, the integral at the right hand also becomes $(n^{1-\tau} - 1)/(1 - \tau)$.

Therefore, (3) can be rewritten as

$$\frac{n^{1-\tau} - 1}{1 - \tau} < \sum_{k=1}^{n-1} k^{-\tau} < 1 + \frac{n^{1-\tau} - 1}{1 - \tau}.$$  

Averaging the boundaries in (4), $\sum_{k=1}^{n-1} k^{-\tau} \approx (n^{1-\tau} - 1)/(1 - \tau) + 1/2 = (2n^{1-\tau} - 1 - \tau)/(2(1 - \tau))$. So, as C is the reciprocal of $\sum_{k=1}^{n} k^{-\tau}$, the normalization factor C is:

$$C \approx \frac{2(1 - \tau)}{2n^{1-\tau} - 1 - \tau}.$$  

With respect to the number of edges $m$ we need to compute the summation of (2). In this case, we note that $k^{-\tau}$ is monotonically increasing, so we approximate it as:

$$\int_{0}^{n} k^{-\tau} \, dk < \sum_{k=1}^{n-1} k^{-\tau} < \int_{1}^{n} k^{-\tau} \, dk.$$  

The integral at the left hand is $(n^{2-\tau} - 1)/(2 - \tau)$, which is approximately $n^{2-\tau}/(2 - \tau) - n^{1-\tau}$. The one at the right hand is $(n^{2-\tau} - 1)/(2 - \tau)$. Averaging the boundaries, $\sum_{k=1}^{n-1} k^{-\tau} \approx n^{2-\tau}/(2 - \tau) - n^{1-\tau}/2$.

Since the number of edges $m$ obeys (2), we obtain:

$$m \approx n \frac{2(1 - \tau)}{2n^{2-\tau} - 1 - \tau} \left( \frac{n^{2-\tau} - n^{1-\tau}}{2} \right).$$

B. SECOND CASE $\tau = 1$

In this case, the summation $\sum_{k=1}^{n-1} k^{-\tau}$ is simply $\sum_{k=1}^{n-1} k^{-1}$, which is the sum that generates the $n - 1$-th Harmonic number $H_{n-1}$. So, we use the standard approximation of $H_{n-1} = \sum_{k=1}^{n-1} k^{-1} = \ln(n - 1) + \gamma / C_{n-1}$; where $\gamma$ refers to the Euler’s constant, $\gamma \approx 0.57722$. Furthermore, $\ln(n - 1) \approx \ln(n) - 1/n$. Thus, $\sum_{k=1}^{n-1} k^{-1} \approx \ln(n) + 0.57722$, so the normalization factor C is:

$$C \approx \frac{1}{\ln(n) + 0.57722}.$$  

The summation in (2) is $\sum_{k=1}^{n} k^{-1}$. Replacing the parameter $\tau$ by 1 we obtain $\sum_{k=1}^{n-1} k^{-1} = n - 1$. Therefore, according to (2), the number of edges is:

$$m \approx \frac{n(n - 1)}{\ln(n) + 0.57722}.$$
C. THIRD CASE $\tau > 1$

Once order to compute $C$, we use the same approximation of (3), so we obtain $C \approx \frac{2(\tau - 1)}{1 + \tau}$. However, as $\tau > 1$, we note that $\lim_{n \to \infty} 2n^{1-\tau} = 0$ and neglect this term. So, we use:

$$C \approx \frac{2(\tau - 1)}{1 + \tau} \quad (10)$$

With respect to the number of edges, the function $k^{1-\tau}$ is decreasing, so we approximate the summation $\sum_{k=1}^{n-1} k^{1-\tau}$ of (2) as follows,

$$\int_1^n k^{1-\tau} \, dk < \sum_{k=1}^{n-1} k^{1-\tau} < 1 + \int_1^n k^{1-\tau} \, dk. \quad (11)$$

The integral at the left hand is $(n^{2-\tau} - 1)/(2 - \tau)$. The right hand, once again we use “1+”, as it is the exact value of $k^{1-\tau}$ when $k = 1$; and the integral is $(n^{2-\tau} - 1)/(2 - \tau)$, which is approximately $(n^{2-\tau} - n^{1-\tau}(2 - \tau) - 1)/(2 - \tau)$.

But, as $\tau > 1$, the contribution of $n^{1-\tau}$ is negligible, so the integral at the right hand becomes $(n^{2-\tau} - 1)/(2 - \tau)$.

Averaging the boundaries in (11), $\sum_{k=1}^{n-1} k^{1-\tau} \approx (n^{2-\tau} - 1)/(2 - \tau) + 1/2 = (2n^{2-\tau} - \tau)/(2(2 - \tau))$. So, according to (2) the number of edges is:

$$m \approx n^{-1} \frac{2(\tau - 1)}{1 + \tau} \approx \frac{n(\tau - 1)(2n^{2-\tau} - \tau)}{(1 + \tau)(2 - \tau)} \quad (12)$$

To avoid the singularity of (12) when $\tau = 2$, we note that in this case, the summation $\sum_{k=1}^{n-1} k^{1-\tau}$ of (2) becomes $\sum_{k=1}^{n-1} k^{-1} \approx \ln n + 0.57722$. Also, evaluating $C$ in (10) at $\tau = 2$ we obtain $C \approx 2/3$. Therefore, the number of edges when $\tau = 2$ is:

$$m \approx \frac{2n(\ln n + 0.57722)}{3} = \frac{2}{3} n \ln n + 0.38481 n \quad (13)$$

D. FINAL REMARKS

Figure 2 plots the edge set size estimations ($m$) for several node set sizes ($n$), ranging from 1 000 to 100 million nodes, varying $\tau \in [0.0, 4.0]$. As can be seen, even though each series combines the four models for $m$ deduced in this analysis, the curves seem continuous. Note that for small node set sizes, there is a small discontinuity for $\tau = 1.1$ (look at the bottom curves where $n < 10^5$ nodes). However, as $n$ increases, the four models for $m$ blend softly into each others.

We ran preliminary numerical simulations and found that our formulae have less than 10% of error in the edge set size $m$ when $\tau$ varies in the range (0, 3.0). On the other hand, considering that in the real world the law’s exponent $\tau \in [1.92, 2.72]$ [7], in the experimental evaluation of our method (Section VII), we work in the case $\tau = 2$ and use $m = \left[\frac{2}{3} n \ln n + 0.38481 n\right]$, which is a simply, yet valid estimation of the edge set size $m$ that produces a graph with the power law distribution.

VI. R\textsuperscript{3}MAT: A SCALABLE IMPLEMENTATION OF R-MAT

The basic implementation of R-MAT (described in Section IV) has an important problem: it requires to keep in main memory the adjacency matrix. Clearly, this issue restricts the scalability of the method as the size of the graph will be bounded by the amount of main memory.

In order to confirm our intuition, we implemented a Java program to test the limits for creating a large matrix. The program incrementally creates a matrix of size $n \times n$ and fills each cell with an integer. The program was tested in Google Cloud Platform, using a virtual machine with 16 vCPU (intel Skylake), 64GB RAM, 100GB HD, Debian GNU/Linux 9 (amd64 build on 20190514). The evaluation shown an upper bound of $n = 40524$ under a usual java execution, and $n = 79740$ after setting the maximum Java heap size to 64GB.

In response to the problem described above, we introduce R\textsuperscript{3}MAT, a generation method that uses an array instead of a matrix. The details of the method are described next.

A. THE ARRAY-BASED METHOD

Assume that $G = (N, E)$ denotes the graph to be generated, where $n = |N|$ is the number of nodes, and $m = |E|$ is the number of edges. Recall that for law exponent $\tau = 2$, the number of edges is $m = \left[\frac{2}{3} n \ln n + 0.38481 n\right]$.

R\textsuperscript{3}MAT works in a two-step fashion. In Step 1, a node degree distribution is computed and stored as an array (the procedure is explained in Section VI-B). Please note that no edge is produced in this first step. In Step 2, the array with the degree distribution is used to actually generate edges for directed or undirected graphs (see Sections VI-C and VI-D respectively).

The above method increases the scalability, while preserving, or sometimes improving, the running time of the generation process. Additionally, R\textsuperscript{3}MAT allows the sequential...
B. GENERATION OF DEGREE DISTRIBUTION

Our graph generation method is based in the creation of an array containing the degree distribution of the nodes to be generated. This task is carried out by the function GenerateDistribution, presented in Algorithm 3.

Algorithm 3 GenerateDistribution(long n, String T)

Input: n ← number of nodes of the graph.
T ← “D” (directed) or “U” (undirected) graph.
Output: An array D such that D[i] contains the degree of node i.

1 begin
2     double (ρa, ρb, ρc, ρd) ← (0.75, 0.05, 0.19, 0.01)
3     if T = “U”
4         then (ρa, ρb, ρc, ρd) ← (0.75, 0.05, 0.18, 0.02)
5     double Δa ← (0.25 − ρa)/[log2 n]
6     double Δb ← (0.25 − ρb)/[log2 n]
7     double Δc ← (0.25 − ρc)/[log2 n]
8     double Δd ← (0.25 − ρd)/[log2 n]
9     long m = \(\frac{2}{7} \cdot n \cdot \ln n + \frac{384}{81} \cdot n\)
10    long D[0] ← new long[n]
11   // Array initialization
12   for long i ← 0 to n − 1 D[i] ← 1
13   if T = “D” then m ← m − n
14   else // T = “U”
15       if (n mod 2) = 1 then D[0] ← 2
16       m ← m − \(\lfloor \frac{n+1}{2} \rfloor\)
17   // Array generation
18   for long i ← 1 to m do
19       long edge[] ← ChooseEdge(1, 1, n, n, ρa, ρb, ρc, ρd, Δa, Δb, Δc, Δd)
20       long s ← edge[0] − 1, D[s] ← D[s] + 1
21       if T = “U” then
22           long t ← edge[1] − 1, D[t] ← D[t] + 1
23       end
24   end
25   return D
26 end

The input parameters of the function GenerateDistribution are the number of nodes n and the graph type T to be generated (Directed or Undirected). The output is an array D of length n, such that the value of D[i] defines the number of edges (degree) of the node i, where 0 ≤ i < n.

Similar to the basic R-MAT, presented in Algorithm 1, the GenerateDistribution function begins with the definition of the matrix partitioning probabilities (Algorithm 3, Lines 2 and 3), the parameters for probability equalization (Lines 4 to 7), and the number m of edges (Line 8).

In the array initialization step (Lines 11 to 16), each entrance in the array D[i] is initialized with 1, i.e., each node is initialized with degree 1. This accounts for n edges in the case of directed graphs. In the case of undirected graphs this accounts for \(\frac{n}{2}\) edges (with an small adjustment when the number of nodes is odd). This step allows to avoid isolated nodes, but can be removed when necessary.

The array generation step executes the R-MAT recursive procedure to complete m edges in the graph; indeed m − n or m − \(\lfloor \frac{n+1}{2} \rfloor\) times after the array initialization step for directed or undirected graphs, respectively (Lines 18 to 25). In each iteration, an edge is obtained by calling the method ChooseEdge (Line 19) as defined in Algorithm 2. Assuming that s ← edge[0] − 1 defines the source node and t ← edge[1] − 1 defines the target node, we increase by 1 the value of D[s] for both graph types and increase the value of D[t] just for the undirected graphs (Lines 21 to 24).

Algorithm 3 needs O(n) time for initialization and O(log n) time for each generated edge in Line 19, so this sums O(m log n) for all of them. According our election of n, this means O(n log^2 n) execution time, the same as basic R-MAT. However, it only needs O(n) memory space.

C. GENERATION OF DIRECTED EDGES

In general terms, the method for generating directed edges takes an array D containing the degree distribution and, for each node i, it generates D[i] edges, first by going backward in the array, and then by going forward. The idea of the method is shown in Figure 3.

Algorithm 4 gives the pseudocode of the method. The first step is to create the array of degrees D by calling the function GenerateDistribution (Line 2). For each node i, the method generates D[i] edges divided in two groups. The first group contains edges of the form e_1 = (i, i −
Algorithm 4 GenerateDirectedEdges(long n)

Input: n ← number of nodes of the graph.
Output: A list of edges.
1 begin
2 long D[] ← GenerateDistribution(n,"D")
3 ArrayList<Edge> edges ← new ArrayList<>()
4 for long i ← 0 to n − 1 do
5     long j ← i + 1
6     while D[i] > 0 ∧ j ≥ 0 do
7         edges.add(new Edge(i,j))
8         D[i] ← D[i] − 1, j ← j − 1
9     end
10     j ← i + 1
11     while D[i] > 0 ∧ j < n do
12         edges.add(new Edge(i,j))
13         D[i] ← D[i] − 1, j ← j + 1
14     end
15 return edges
16 end

Algorithm 5 GenerateUndirectedEdges(Long n)

Input: n ← number of nodes of the graph.
Output: A list of edges.
1 begin
2 long D[] ← GenerateDistribution(n,"U")
3 ArrayList<Edge> edges ← new ArrayList<>()
4 for long i ← 0 to n − 1 do
5     long j ← i + 1
6     while D[i] > 0 ∧ j < n do
7         if D[j] > 0 then
8             edges.add(new Edge(i,j))
9         edge
10         j ← j + 1
11     return edges
12 end
13 end

For the case of directed graphs, the best combination is (0.75, 0.05, 0.19, 0.01) with a weighted average KS-stat $\frac{\overline{K}(c)}{n} = 0.0438$. For undirected graphs, the best combination is (0.75, 0.05, 0.18, 0.02) with $\frac{\overline{K}(c)}{n} = 0.0479$. Recall that a low value for $\frac{\overline{K}(c)}{n}$ implies a good fit between the observed distribution and the power law.

The above combinations are used by default in the implementation of $R^3$MAT (see Algorithm 3, Lines 2 and 3). The degree distributions produced with these partition probabilities are analyzed in the experimental evaluation (Section VII-C3).

F. A HASH-BASED IMPLEMENTATION

The array-based $R^3$MAT implementation presented above has a restriction: a primitive array can store a limited number of items. In the case of Java, the documentation indicates that the maximum size of a primitive array (e.g., $D[]$ in Algorithm 3, Line 9) is given by the maximum value an int can have, that is, $2^{31} − 1 = 2147483647$ elements.

In order to verify this information, we developed a java application to test the maximum size of a primitive array. This application was tested in Google Cloud Platform, using a virtual machine with 32 vCPU, 128GB RAM and Debian GNU/Linux 9 (amd64). The largest primitive array we were able to generate had a size of 2 147 483 644 elements (by using the -Xmx128G execution parameter). Hence, the array-based implementation presented above will be restricted to create graphs with such maximum size.

In order to cover the above restriction, we develop a hash-based implementation of an array whose java code is shown in Algorithm 6. Specifically, we create the java class LargeArray which uses an object of type HashMap<Long,Long> to implement an array as a hash
Algorithm 6 LargeArray Class

```java
public class LargeArray {
    private long size = 0;
    public LargeArray(long array_size) {
        this.size = array_size;
        array = new HashMap<>();
    }
    public long getItem(long index) {
        if (index < size) {
            return array.get(index);
        }
        throw new RuntimeException("Invalid index");
    }
    public boolean setItem(long index, long value) {
        if (index < size) {
            array.put(index, value);
            return true;
        }
        throw new RuntimeException("Invalid index");
    }
    public long size() {
        return size;
    }
}
```

Table 5. Hence, a key-value pair \((i, v_i)\) in the hash table stores the value \(v_i\) of the element \(i\) in the array. The advantage of this implementation is that the memory space is allocated on demand, and the maximum number of elements is extended to the maximum value of Long, i.e., \(2^{63} - 1\). By using the same virtual machine mentioned above, we were able to generate an array of \(2^{12} = 4 \times 294,967,296\) elements, overpassing the limit of the primitive array.

Thus, the algorithms of the hash-based method are almost identical to the ones of the array-based method presented in this section, but changing the primitive array \(D[\] by an object of type LargeArray. After conducting some experiments (not included here because space restrictions), we realize that the limit of the hash-based implementation depends on the amount of RAM memory. Note that HashMap is internally implemented with primitive arrays, so the hash-based implementation could be as fast as the array-based, but not better.

G. GENERAL COMPARISON OF THE METHODS

At this point, we have presented three implementations of the R-MAT graph generation model. For economy of space, we call them Basic R-MAT, \(R^3_{\text{MAT}}\), and \(R^3_{\text{MAT}}\). Basic R-MAT corresponds to the matrix-based method that works with the adjacency matrix of the graph; \(R^3_{\text{MAT}}\) is the array-based method (\(R^3_{\text{MAT}}\)) that uses the degree distribution of the nodes, stored as a primitive array; and \(R^3_{\text{MAT}}\) is the \(R^3_{\text{MAT}}\) that manages the degree distribution as a hash table.

The main advantage of \(R^3_{\text{MAT}}\), in comparison with the Basic R-MAT, is that \(R^3_{\text{MAT}}\) reduces the amount of required memory from \(n^2\) to \(n\) entries. Intuitively, this memory reduction should increase the efficiency and scalability of the generation process.

The technique to select target nodes, by going backward and forward in the array, avoids the generation of repeated edges (an issue of the Basic R-MAT). This technique, introduced in our \(R^3_{\text{MAT}}\), should reduce slightly the running time. Also, it allows to implement an API where edges are generated on demand, i.e., a one-by-one edge generation.

Both \(R^3_{\text{MAT}}\) and \(R^3_{\text{MAT}}\) use a two-step process where, in Step 1, a node degree distribution is computed; so that, in Step 2, the node degree distribution is used to generate edges. \(R^3_{\text{MAT}}\) uses a primitive array to manage the degree distribution, while \(R^3_{\text{MAT}}\) uses a hash table, to overtake the array size limitation.

VII. EXPERIMENTAL EVALUATION

The objective of our experimental evaluation is to test and compare the efficiency and scalability of \(R^3_{\text{MAT}}\), the graph generation method presented in this article, considering both variants \(R^3_{\text{MAT}}\) and \(R^3_{\text{MAT}}\). Also, we perform a performance comparison of our \(R^3_{\text{MAT}}\) with the available sequential alternatives in the state of the art, namely, Boost [25], PaRMAT [28], and SNAP [30]. This section includes the evaluation methodology, the experimental results, and the corresponding discussion.

A. METHODOLOGY

The experimental evaluation is based on the generation of graphs. Each experiment combines four variables: generation method, graph size, graph type, and processing power. We consider the three methods studied in this paper: Basic R-MAT, \(R^3_{\text{MAT}}\), and \(R^3_{\text{MAT}}\). In terms of size, we considered graphs having 1 000, 10 000, 100 000, 1 million, 10 million, and 100 million nodes. The graph type variable indicates the generation of directed or undirected edges. The processing power implies that the experiments are executed on machines with different characteristics (i.e., processor, memory, and hard disk).

Based on these variables, we evaluate the generation methods in terms of efficiency, scalability, and realism. The efficiency is measured as the elapsed time (or runtime) required for a method to generate a graph with specific characteristics (e.g., a directed graph having 1 000 nodes). The objective is to determine the fastest and the slowest method. To do so, we use the built-in java function System.currentTimeMillis().
The methods are evaluated under two notions of scalability. In the first, we measure its capability to support an increasing number of nodes. The idea is to determine the largest graph a method is able to generate. In the second, we analyze its scalability with respect to the computational resources. The idea is to determine the dependency of a method with regard to the hardware.

It has been argued that the R-MAT method is able to resemble the properties occurring in real networks, in particular power law distributions. Hence, the realism of a method is given by its ability to produce a power law distribution; for any graph size. To verify this notion of realism, we computed the Kolmogorov–Smirnov statistic and included charts that show the degree distribution of the graphs.

Finally, in order to carry out a performance comparison, we run another series of experiments considering our R$^3$MAT with Boost [25], PaRMAT [28], and SNAP [30], as all of them are sequential.

### B. EXPERIMENTAL SETUP

All the experiments are executed in virtual machines created in Google Cloud Platform. As shown in Table 1, we use virtual machines with different number of CPUs (Intel Skylake), memory size (RAM), and secondary memory size (SSD). The operating system of all machines is Debian GNU/Linux 9 (amd64 build on 20190514). We also install Java openjdk version 1.8.0_212 without graphic environment.

#### TABLE 1. Machines used in the experimental evaluation.

| Machine ID | CPU (number) | RAM (GB) | SSD (GB) |
|------------|--------------|----------|----------|
| M4         | 1            | 4        | 50       |
| M8         | 2            | 8        | 50       |
| M16        | 4            | 16       | 100      |
| M32        | 8            | 32       | 100      |
| M64        | 16           | 64       | 100      |

We have developed a java application, called r3mat, that implements the three methods described in this article. The source code of our implementation is available in GitHub (specifically at https://github.com/renzoar/r3mat). r3mat provides a jar-based interface which allows to generate graphs (from the command-line) by executing a instruction of the form

```
java -jar -Xmx<S>MG r3mat.jar <N> <T> <D> <M> <F> <S>
```

where <S> defines the maximum memory allocation pool for the Java virtual machine, <N> is a positive integer defining the number of nodes of the graph (mandatory), <T> indicates the graph type (0 for undirected, 1 for directed), <D> indicates the statistical distribution of edges (0 for normal, 1 for power law), <M> indicates the generation method (0 for matrix-based, 1 for array-based, 2 for hash-based), <F> indicates the output data format (0 for edge list, 1 for graphml), and <S> defines a random seed.

The graphs of the experimental evaluation are generated by setting the parameters <N>, <T>, <D>, and <M>. As output data format, we use a simple edge list representation (default).

### C. EXPERIMENTAL RESULTS

Each experiment corresponds to a specific combination of generation method, graph size, edge type, and virtual machine. For each experiment, the r3mat application is executed three times, we registered the average execution time (runtime) and the size of the biggest output graph file.

#### TABLE 2. Graphs generated in the experimental evaluation.

| Graph ID | Graph type | Number of nodes | Number of edges | File size (Bytes) |
|----------|------------|----------------|----------------|-------------------|
| DG1k     | Directed   | 1·10^6         | 4·10^9         | 27K / 35K         |
| UG1k     | Undirected | 1·10^6         | 4·10^9         | 27K / 34K         |
| DG10k    | Directed   | 10·10^4        | 63·10^3        | 463K / 567K       |
| UG10k    | Undirected | 10·10^4        | 63·10^3        | 466K / 562K       |
| DG100k   | Directed   | 100·10^4       | 786·10^3       | 7.8M / 8.4M       |
| UG100k   | Undirected | 100·10^4       | 786·10^3       | 7.9M / 8.3M       |
| DG1M     | Directed   | 1·10^6         | 9·10^6         | ~117M             |
| UG1M     | Undirected | 1·10^6         | 9·10^6         | ~117M             |
| DG10M    | Directed   | 10·10^6        | 109·10^6       | ~1.6G             |
| UG10M    | Undirected | 10·10^6        | 109·10^6       | ~1.6G             |
| DG100M   | Directed   | 100·10^6       | 1.25·10^9      | ~21G              |
| UG100M   | Undirected | 100·10^6       | 1.25·10^9      | ~21G              |

Table 2 shows a summary of the generated graphs. In general terms, we generate twelve types of graphs, with a maximum size of $n = 100 \cdot 10^6$, i.e., 100 million nodes. Recall that the number of edges is given by the formula $m = \left\lceil \frac{2}{3} n \ln n + 0.384 81 n \right\rceil$. Table 2 also shows the size of the output graph file (expressed in Bytes). When two values are included, e.g. 27K / 35K, the first value corresponds to the Basic R-MAT, and the second one to both R$^3$MAT and R$^3$MAT methods. Such difference reflects the particularities of the matrix-based method.

#### TABLE 3. Runtime results of the Basic R-MAT.

| Graph | M4 | M8 | M16 | M32 | M64 |
|-------|----|----|-----|-----|-----|
| DG1k  | 36 | 27 | 18  | 16  | 19  |
| UG1k  | 38 | 28 | 18  | 16  | 16  |
| DG10k | 338| 137| 111 | 113 | 100 |
| UG10k | 321| 216| 115 | 119 | 103 |
| DG100k| -  | -  | -   | -   | -   |
| UG100k| -  | -  | -   | -   | -   |

#### TABLE 4. Runtime results of the R$^3$MAT.

| Graph | M4 | M8 | M16 | M32 | M64 |
|-------|----|----|-----|-----|-----|
| DG1k  | 36 | 27 | 18  | 16  | 19  |
| UG1k  | 38 | 28 | 18  | 16  | 16  |
| DG10k | 338| 137| 111 | 113 | 100 |
| UG10k | 321| 216| 115 | 119 | 103 |
| DG100k| -  | -  | -   | -   | -   |
| UG100k| -  | -  | -   | -   | -   |

#### TABLE 5. Runtime results of the R$^3$MAT$_h$.

| Graph | M4 | M8 | M16 | M32 | M64 |
|-------|----|----|-----|-----|-----|
| DG1k  | 36 | 27 | 18  | 16  | 19  |
| UG1k  | 38 | 28 | 18  | 16  | 16  |
| DG10k | 338| 137| 111 | 113 | 100 |
| UG10k | 321| 216| 115 | 119 | 103 |
| DG100k| -  | -  | -   | -   | -   |
| UG100k| -  | -  | -   | -   | -   |

1) **EFFICIENCY**

Tables 3, 4, and 5 present the execution times of the three generation methods. A runtime marked with * indicates that the r3mat application is executed with the -Xmx parameter set to the maximum memory size.

The lowest running time, $t = 14$ ms, corresponds to R$^3$MAT$_h$ method, when it generates a directed graph of 1 000 nodes, in a 32GB virtual machine. The highest running time, $t = 4.469 536$ ms or 74.5 minutes, is obtained by the R$^3$MAT$_h$.
method to generate an undirected graph of 100 million nodes, in a 16GB virtual machine.

In order to compare the efficiency of the methods, we include Figures 4 and 5. These figures show the runtimes of the three methods to generate graphs of different sizes (from 1 000 to 100 million nodes), all using a 64GB virtual machine (i.e. the most powerful in our evaluation). The Basic R-MAT shows some lower runtimes for small graphs, but it is not able to generate the largest graphs. However, both R$^3$MAT variants can generate graphs with 100 million nodes. As can be seen, R$^3$MAT$^a$ has better performance than Basic R-MAT in almost any experimental condition. On the other hand, we empirically prove that the R$^3$MAT$^h$ method is slower than Basic R-MAT for small graphs, but it is slightly surpassed by Basic R-MAT for graphs with 100 000 nodes. According to the trend of growing, it is expected that this difference would increase in favor of R$^3$MAT$^h$.

We can also see that the runtime of the R$^3$MAT$^a$ and R$^3$MAT$^h$ methods maintain a linear growth for all the experiments in terms of the number of edges and mildly superlinear in terms of the number of nodes. In fact, we run linear models for the execution time and obtain that the Basic R-MAT method needs $10.7 \cdot 10^{-5} n \ln n$ ms ($R^2 = 0.9615$), the R$^3$MAT$^a$ method needs $7.04 \cdot 10^{-5} n \ln^2 n$ ms ($R^2 = 0.9869$) and the R$^3$MAT$^h$ method needs $9.93 \cdot 10^{-5} n \ln^2 n$ ms ($R^2 = 0.9441$). It demonstrates the advantage of using an array (primitive or hash-based) instead of a matrix.

2) SCALABILITY

The scalability of each method is analyzed from two points of view: (i) with respect to the graph size; and (ii) with respect to the processing power.

a: SCALABILITY UNDER GRAPH SIZE

Consider the charts shown in Figures 4 and 5, both in logarithmic scale, where the abscissa shows the node set size from $10^3$ to $10^8$. As can be seen, the Basic R-MAT method can only generate graphs up to 100 000 nodes. By using a 64GB virtual machine, we verify that this method has an upper bound of $n = 40 524$ nodes under a usual java execution, and $n = 195 000$ nodes after setting the maximum Java heap size to 64GB. In contrast, both R$^3$MAT$^a$ and R$^3$MAT$^h$ methods are able to generate the largest graphs (of 100 million nodes), R$^3$MAT$^h$ being the one that requires more memory.

b: SCALABILITY UNDER PROCESSING POWER

Table 3 shows that the Basic R-MAT method is strongly dependent on the processing power. In fact, to generate a graph of 100 000 nodes with such method, it is necessary to use the 64GB virtual machine. Table 5 shows that the
The $R^3\text{MAT}_h$ method has a slightly dependency on the processing power as it is not able to generate the biggest graphs with the smallest virtual machines. In comparison to the above two, Table 4 shows that the $R^3\text{MAT}_d$ method does not depend on the processing power as it is able to generate the biggest graphs with the smallest virtual machine.

![Figure 6](image1.png)

**Figure 6.** Runtime of the $R^3\text{MAT}_d$ method for generating directed graphs of different sizes, in different virtual machines.

![Figure 7](image2.png)

**Figure 7.** Runtime of the $R^3\text{MAT}_d$ method for generating undirected graphs of different sizes, in different virtual machines.

Figures 6 and 7 allow us to analyze a little more the scalability of the $R^3\text{MAT}_d$ method. In general, for a graph of a given size, the runtime variates mildly with respect to increase or decrease the processing power. Hence, we can assume that this method will be able to generate bigger graphs with the smallest virtual machine. The only problem is that the generation of bigger graphs will be limited by the maximum size of the primitive array (i.e., $2^{31} - 1 = 2,147,483,647$ elements). As discussed in Section VI-F, the $R^3\text{MAT}_h$ method should overcome this limitation.

3) REALISM

The objective of this experiment is to verify that $R^3\text{MAT}$ generates graphs following a power law distribution. To do this, we present a statistical verification and a visual verification. The statistical verification concerns the use of the Kolmogorov–Smirnov statistic (KS-stat) [35]. As described in Section IV-E, the KS-stat metric can be used to test how well a sample distribution fits a power law distribution.

| Type / Size | 1k  | 10k | 100k | 1M   | 10M  |
|-------------|-----|-----|------|------|------|
| Directed    | 0.1194 | 0.0512 | 0.0415 | 0.0249 | 0.0210 |
| Undirected  | 0.1380 | 0.0774 | 0.0440 | 0.0311 | 0.0257 |

In this experiment, we considered directed and undirected graphs with the following sizes (number of nodes): $n_1 = 1k$, $n_2 = 10k$, $n_3 = 100k$, $n_4 = 1M$ and $n_5 = 10M$. Table 6 shows the KS-stat scores obtained for all these graphs. Note that a small score for KS-stat denotes a better fit between the observed and the power law distribution. In this case, all the scores have a value lower than 0.15, which is considered a very good score. Hence, we can say that all the graphs follow a power law distribution. Note also that the scores decrease with the size of the graph. Therefore, we can conjecture that graphs larger than 10M will also follow a power law distribution.

The visual verification concerns the analysis of the scatter plots showing the degree distribution of the graphs. Figure 8 shows the plots for the smaller ($n = 1k$) and the larger graphs ($n = 10M$) generated in this experiment. We do not include all the plots in the paper for economy of space. Nevertheless, we verified that all the plots show the same result: the generated graphs do follow a power law distribution.

On the other hand, we compute least squares approximations for the degree distribution, in order to verify that the empirical law’s exponent $\tau$ obtained in graphs generated with our methods is close to 2. For this sake, we apply linear regression using the model

$$\log P(x = k) \sim \log k^{-\tau}.$$  \hspace{1cm} (14)

We analyze the graph with one million nodes as it is big enough to be considered a practical case.

Using the estimation $m = \frac{2}{3} n \ln n + 0.38481n$, for one million nodes there are $9,595,150$ edges in the graph. Thus, in the directed graph with one million nodes, each node has $\mu_d,10^6 \approx 9.6$ neighbors in average, while in the undirected graph, each node has $\mu_u,10^6 \approx 19.2$ neighbors in average (subscript $d/u$ refers to directed/undirected graph, and $10^6$ is the number of nodes).

We note that the distribution slope for degrees bellow the average one is mild. In the other extreme, the frequency of nodes having very high degree is very small and shows gaps, so the slope of this section is too steep. However, the power law manifests itself properly starting from the average degree and beyond, in the middle of the degree distribution, as expected. So, we compute regression models for several ranges of degree starting from the average $\mu$. Ranges have the form $[(\mu), [i\mu]]$, for $i \in [2, 32]$, with the exception of...
FIGURE 8. Degree distribution, in logarithmic scale, of graphs generated with R³MAT. Plots (a) and (b) show the degree distributions for directed graphs of sizes \(n = 1k\) and \(n = 10M\) respectively. Plots (c) and (d) show the degree distributions for undirected graphs with the same sizes.

The results for the out-degree and degree for directed and undirected graphs are shown in Tables 7 and 8, respectively. All the values of \(R^2\) are close to 1, so the model (14) is appropriate, whichever the degree range. As a matter of fact, for directed graphs, the global fitting is \(\log_{10} P(x = k) = -1.9652 \log_{10} k + 0.0501\), with an \(R^2 = 0.93082\). On the other hand, for undirected ones, the global fitting is \(\log_{10} P(x = k) = -1.9535 \log_{10} k + 0.3444\), with an \(R^2 = 0.93083\). In both cases, the empirical law’s exponent just differs 2.4% with respect to the expected value \(\tau = 2\).

As can be seen in Tables 7 and 8, the slope of the range \([\lceil\mu/2\rceil, \lceil\mu\rceil]\) is approximately 1.57 and 1.46, for directed and undirected graphs, respectively, far from the expected value \(\tau = 2\); this is because the degree frequency distribution at the beginning of the plot decays softly given the number of edges in the graph. Nevertheless, the values for \(\tau\) in the ranges \([\lceil\mu\rceil, \lceil i\mu\rceil]\) are fairly close to the expected value of \(\tau = 2\). For instance, when considering the out-degree range [9, 249] in Table 7 (for directed graphs), the empirical value of \(\tau\) is 1.9987. Likewise, the degree range [19, 364] in Table 8 (for undirected graphs) shows an empirical \(\tau = 2.0054\). These values of empirical law’s exponent \(\tau\) verify that our estimation of the graph edge set size \(m\) is accurate.
4) PERFORMANCE COMPARISON

Next, we present experiments to verify that our \( R^3 \)MAT method is competitive with the state of the art in terms of both execution time and realism. For this sake, we compare \( R^3 \)MAT, with Boost [25], PaRMAT [28], and SNAP [30] for generating graphs from 1 000 to 100 million nodes, both for directed and undirected graphs. The results of this experiment are shown in Table 9, Table 10 and Figure 9.

We run this experiment using the Machines M4 and M64 in Table 1, in order to show a performance comparison in a scenery with a modest single machine (Machine M4) and a server with more available resources (Machine M64). For each combination of experimental parameters, we run ten executions and show the averages. In order to perform a fair performance comparison, we follow the recommendations given by the respective authors when running their algorithms. For the sake of compatibility with SNAP [30], instead of Debian GNU/Linux 9, this time we use Ubuntu 18.04 LTS, amd64 bionic image build on 2020-01-08. According to its documentation, we only can use SNAP for generating directed graphs. Only for this comparison and for simplicity, execution times are measured with the unix command `time`, considering the user time.

Table 9 shows the runtimes in milliseconds in the modest Machine M4 that has 4GB of main memory with one processor. We use boldface to indicate the fastest implementation. As can be seen, from one hundred thousand nodes, our \( R^3 \)MAT, method is the fastest implementation, PaRMAT being the second best alternative. On the other hand, we cannot complete the series with Boost, as it exhausted the machine memory with 10 million nodes, and we cut the SNAP evaluation of 10 million nodes after it reaches 80 minutes, which is 24 times slower than the second best alternative.

Table 10 shows the runtimes in milliseconds in the Machine M64 that has 64GB of main memory and 16
processors. Again, the fastest implementation are indicated in boldface. Once again, our $R^3MAT_{av}$ method is the fastest implementation from one hundred thousand nodes and on, PaRMAT being the second best alternative, with the exception of the directed 100-million-node graph where PaRMAT is just 8% faster. On the other hand, we cut the series with Boost and SNAP, after they reach 7 minutes, which is 2 times the second best alternative. The table shows that our $R^3MAT$ remains competitive although there are more available resources. Moreover, even though we allow PaRMAT to use 16 threads, the obtained speed up is not significative.

Finally, Figure 9 shows the out-degree distribution obtained in graphs of one million nodes, for the four methods. Boost and PaRMAT show erratic power law distributions, whereas SNAP and $R^3MAT$ present the best degree distributions. Computing the Kolmogorov–Smirnov statistic, we obtain the following scores: $ks = 0.1061$ for Boost, $ks = 0.0495$ for PaRMAT, $ks = 0.0308$ for SNAP, and $ks = 0.02494$ for $R^3MAT$. Hence, we confirm that $R^3MAT$ presents the best fit for a power law distribution with respect to the other generators.

VIII. CONCLUSIONS

The R-MAT graph generation method allows to produce synthetic graphs that resemble the properties observed in real-world graphs, in particular, that their degree distribution follows the power law. Given a node set $N$ of size $n$ and an expected size value $m$ of the edge set $E$, the basic
R-MAT method maintains in main memory the full adjacency matrix that represents the set $E$, and starting from an empty set $E$, adds edges one by one until the desired size is reached. Despite of its simplicity, its $O(n^2)$ memory requirement makes the basic R-MAT version unsuitable for generating large graphs.

For the sake of alleviating the basic R-MAT intensive memory consumption issue, we have introduced two practical implementations of R-MAT allowing the generation of large graphs whose degree distributions follow the power law. The key of our proposal is that it only needs to maintain in main memory an array of the desired degree for each node, just requiring $O(n)$ space.

Moreover, we have analyzed the power law in the context of the R-MAT to obtain explicit, closed-form expressions to estimate $m$, for values of the law’s exponent that occur in the real world. By virtue of our analysis, we have set $m = \frac{2}{3}n \ln n - 0.38481n$, which is the simplest, yet valid estimation that produces a graph with the power law degree distribution whose law’s exponent is 2. However, in order to emulate other law’s exponents, it is enough to use the other expressions for $m$ presented in this work.

The empirical evaluation has shown that our methods scale well both in the graph size and the processing power. In particular, they are able to generate graphs of million nodes and billion edges in a virtual machine having a single processor and 4GB of main memory. We have also shown that the proposed methods do not require of large amounts of main memory to scale in runtime and graph size. Furthermore, the empirical evaluation has also shown that our methods generate graphs resembling the characteristic power law degree distribution. Finally, the performance comparison shows that $R^3$MAT is the fastest alternative with respect to the state of the art, when considering a single computer with small amount on main memory in a sequential fashion.

Currently, we are developing a Hadoop implementation of the array-based generation method. We expect to reduce the graph generation runtime by following a parallel computing strategy, and to test the approach in a Hadoop cluster.

REFERENCES

[1] A. Khan and S. Elnikety, “Systems for big-graphs,” Proc. VLDB Endow., vol. 7, no. 13, pp. 1709–1710, Aug. 2014.

[2] S. Zhao, L. Yu, and B. Cheng, “Probabilistic community using link and content for social networks,” IEEE Access, vol. 5, pp. 27189–27202, 2017.

[3] Z. Zhu, Q. Peng, Z. Li, X. Guan, and O. Muhammad, “Fast PageRank computation based on network decomposition and DAG structure,” IEEE Access, vol. 6, pp. 41760–41770, 2018.

[4] W. L. Hamilton, R. Ying, and J. Leskovec, “Representation learning on graphs: Methods and applications,” IEEE Data Eng. Bull., vol. 40, no. 3, pp. 52–74, Sep. 2017.

[5] A. Prat-Pérez, J. Gássado-Gámez, X. F. Salas, P. Koupý, S. Depner, and D. B. Bartolini, “Towards a property graph generator for benchmarking,” in Proc. 5th Int. Workshop Graph Data-Manage. Exp. Syst. (GRADES), 2017, pp. 1–6.

[6] S. Nobari, X. Lu, P. Karras, and S. Bressan, “Fast random graph generation,” in Proc. 14th Int. Conf. Extending Database Technol. (EDBT/ICDT), 2011, pp. 331–342.

[7] D. Chakrabarti and C. Faloutsos, “Graph mining: Laws, generators, and algorithms,” ACM Comput. Surv., vol. 38, no. 1, pp. 1–69, Jun. 2006.
[33] L. Hübschle-Schneider and P. Sanders, “Linear work generation of R-MAT graphs,” 2019, arXiv:1905.03525. [Online]. Available: http://arxiv.org/abs/1905.03525

[34] J. Leskovec, J. Kleinberg, and C. Faloutsos, “Graph evolution: Densification and shrinking diameters,” ACM Trans. Knowl. Discovery Data, vol. 1, no. 1, pp. 2:1–2:41, Mar. 2007.

[35] 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.

[36] R. L. Graham, D. E. Knuth, and O. Patashnik, Concrete Mathematics: A Foundation for Computer Science, 2nd ed. Boston, MA, USA: Addison-Wesley, 1994.

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 carried out postdoctoral research at the Department of Computer Science, VU University Amsterdam, as a part of his participation in the Linked Data Benchmark Council Project. He is currently an Assistant Professor with the Department of Computer Science, Universidad de Talca, Chile. He also participates as a Researcher with the Millennium Institute for Foundational Research on Data, Chile. His research interests include the intersection of graph databases and the Semantic Web. Specifically, he works in the theory and design of graph query languages, and the interoperability between RDF and graph databases.

RODRIGO PAREDES received the M.Sc. degrees in computer science and in computer and electrical engineering from the Universidad de Chile and the Ph.D. degree in 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” given by the Chilean Computer Science Society (SCCC) to the best performing Chilean researcher with a recent Ph.D. degree in computer science. He also received 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.

ROBERTO GARCÍA received the bachelor’s degree in computer science from the Universidad de Talca, Chile, in 2014, where he is currently pursuing the Ph.D. degree in the Program in Engineering Systems with the Faculty of Engineering. Since March 2019, he belongs to the Engineering Systems Doctoral Program at the Universidad de Talca. His research interests include graph query languages and benchmarking of graph databases.