ROLL: Fast In-Memory Generation of Gigantic Scale-free Networks*

Ali Hadian‡ Sadegh Nobari§ Behrooz Minaei-Bidgoli‡ Qiang Qu§

‡Iran University of Science and Technology, Iran §Innopolis University, Russia
hadian@comp.iust.ac.ir sadegh@sqnco.com b_minaei@iust.ac.ir qu@innopolis.ru

ABSTRACT

Real-world graphs are not always publicly available or sometimes do not meet specific research requirements. These challenges call for generating synthetic networks that follow properties of the real-world networks. Barabási-Albert (BA) is a well-known model for generating scale-free graphs, i.e graphs with power-law degree distribution. In BA model, the network is generated through an iterative stochastic process called preferential attachment. Although BA is highly demanded, due to the inherent complexity of the preferential attachment, this model cannot be scaled to generate billion-node graphs.

In this paper, we propose ROLL-tree, a fast in-memory roulette wheel data structure that accelerates the BA network generation process by exploiting the statistical behaviors of the underlying growth model. Our proposed method has the following properties: (a) Fast: It performs +1000 times faster than the state-of-the-art on a single node PC; (b) Exact: It strictly follows the BA model, using an efficient data structure instead of approximation techniques; (c) Generalizable: It can be adapted for other "rich-get-richer" stochastic growth models. Our extensive experiments prove that ROLL-tree can effectively accelerate graph-generation through the preferential attachment process. On a commodity single processor machine, for example, ROLL-tree generates a scale-free graph of 1.1 billion nodes and 6.6 billion edges (the size of Yahoo's Webgraph) in 62 minutes while the state-of-the-art (SA) takes about four years on the same machine.

1. INTRODUCTION

The study of complex networks is an impressive but young field of research. While the first contributions to the field date back to the early 1900 since the development of graph theory, new applications of network modeling have given rise to considerable efforts in understanding the structure and properties of complex networks.

The main feature that distinguishes complex networks from other types of networks, such as random graphs and lattices, is their non-trivial properties, i.e. complex networks have irregular characteristics which dynamically evolve in time [7]. For example, a common property of many large networks is that the vertex connectivities follow a scale-free (or power-law) distribution, i.e., the proportion of nodes $P(k)$ connected to k other nodes follows a power-law distribution $P(k) \sim k^{-\gamma}$ where $2 < \gamma < 3$ [6]. A number of theoretical models are proposed to explain how scale-free networks grow over time, including the Barabási-Albert model [6], fitness model [8, 12], tinkering [44], optimization [24, 47], Chung-Lu [1], BTER [30], R-MAT [16], hyperbolic unit disk [49], and the highly optimized tolerance model [15].

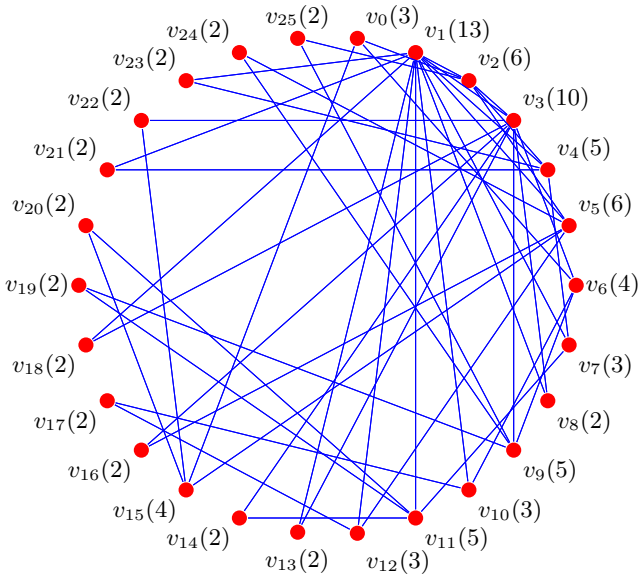

Figure 1: A Barabási-Albert graph with 25 nodes in which $v_i(d)$ has d connections ($m = m_0 = 2$)

Graph growth models not only provide theoretical abstractions for studying natural complex graphs, but also can be used to *generate* synthetic graphs for further analysis and simulation. In the last two decades, various randomized algorithms have been developed to generate synthetic graphs for different growth models. The primary motivation for recent interest in generating synthetic graphs is that, real-world networks are hardly accessible to researchers due to privacy concerns. In addition, synthetic graph generators can be used to create multiple independent graphs with similar statistical properties. These multiple independent graphs are useful for statistical tests, e.g. significance tests and cross-validation [49]. Recently, research is being shifted towards modeling gigantic networks, because small input graphs may not exhibit the same properties that can be found in large-scale graphs [4, 32].

*Supported by RFBR grant 16-37-60089.

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

DOI: http://dx.doi.org/10.1145/2882903.2882964

In this paper, we specifically focus on one of the most referred models, the Barabási-Albert (BA) model, which is a well-known model for scale-free network generation [5, 13, 17, 46, 55]. Figure 1 is a BA network that we use as our running example in this paper. The BA model generates a random scale-free graph using a preferential attachment mechanism: graph nodes are generated one by one, and each time attached to one or more existing nodes in the graph. The attachment is done in a "rich-get-richer" manner, such that the more connected a node is, the more likely it is to receive new connections. Roughly speaking, the probability for a new node to be connected to an existing node is proportional to the degree of the node. Although the fact that the BA model has been widely used in various research areas [5, 17, 46, 55], few studies have paid attention to its efficiency. While many graph generation models have subquadratic computational complexities, the complexity of Barabási-Albert is $O(|V|^2)$, which makes it infeasible for very large graphs. The algorithm is currently implemented in a number of graph-analysis software packages, such as iGraph [1] and graph-tool [2], and is being used for many research works in which the graph size is sufficiently small, e.g. $|V| < 10^7$. However, when the size of the graph goes beyond this range, e.g. for generating big synthetic web-graphs, current implementations cannot be used due to BA's $O(|V|^2)$ complexity.

The computationally intensive part of the algorithm is the degree-proportionate selection, a.k.a. *roulette wheel selection*, i.e. finding the node that is correspondent to the given random number. In this paper, we propose ROLL-tree, a novel data structure for fast random selection from dynamic roulette wheels where the probabilities follow a long-tailed distribution. We suggest two modifications to the basic roulette wheel. First, we propose a bucketing technique that drastically reduces the time required to compute the partial sums by reducing the size of the roulette wheel. Second, we suggest an efficient tree-based data structure along with a novel method for weighted random sampling on the dynamic roulette wheel. ROLL-tree can be applied to various problems with large dynamic roulette wheels, such as random graph generation [10, 13, 31, 34, 43, 45] and simulation of random events [52, 54]. The advantages of our proposed method are **i) Fast**: It performs +1000 times faster than the state-of-the-art on a single node PC; **ii) Exact**: It strictly follows the BA model, using an efficient data structure instead of approximation techniques; **iii) Generalizable**: It can be adapted for other "rich-get-richer" growing models.

The remainder of this paper is organized as follows. The literature is briefly reviewed in Section 2. Section 3 describes the baseline and state-of-the-art roulette wheel selection methods for scale-free graph generation using the BA model. The proposed method is provided in detail in Section 4. Section 5 discusses the theoretical analysis and experimental evaluation results. Finally, Section 6 concludes the paper with directions for future work.

2. RELATED WORK

Data processing in memory is becoming increasingly important in many applications such as spatial data processing [41] and trajectory data management [50] for two reasons. First, main memory has grown gigantically that many datasets fit into it directly. Second, in-memory data generation is a necessary part of any data generation algorithm. Data generators can produce the data in parts that can fit in memory and then merge the partitions. Therefore, speeding up the in-memory data generation helps to substantially speed up any data generator as a whole as well. This paper focuses on increasing the speed of generating a well-know scale-free graph model by proposing a data structure that exploits memory. Subsequently we first cover the popular graph generators then we point to state-of-the-art advancement for the existing models.

2.1 Existing Graph Generation Models

Synthetically generated graphs are used in many applications, such as null-model, simulation, extrapolation, sampling, anonymization, visualization, and summarization [13, 33]. Recently, there has been a great deal of work focused on the development of generative models for scale-free networks [1, 16, 21, 24, 30, 40]. We briefly review the literature.

2.1.1 Random graph models

Research in graph generators started with Erdős-Rényi models, the two simple yet elegant models for generating random graphs. Paul Erdős and Alfréd Rényi proposed the $\Gamma_{v,e}$ model [21], while E. N. Gilbert proposed, at the same time, the $\Gamma_{v,p}$ model [26]. The former, noted as $\Gamma_{v,e}$, chooses a graph uniformly at random from the set of graphs with v vertices and e edges. The latter, noted as $\Gamma_{v,p}$, chooses a graph uniformly at random from the set of graphs with v vertices where each edge has the same independent probability p to exist. The Erdős-Rényi models generate a pure random graph that has a Binomial degree distribution, not power-law, and generally lacks clustering when generating sparse networks. As a result, multiple attempts as follows have been made to develop algorithms that generate graphs with realistic network properties.

2.1.2 Complex network models

Complex networks have statistical and structural properties in common, such as power law degree distribution and shrinking diameter. Moreover, specific groups of complex networks, e.g. social networks or scientific collaboration networks, share additional properties like modular community structures [39], bursty and self-similar edge/weight additions [16, 38], snapshot power-law [38], and many others [2, 3]. Note that various complex networks are not structurally the same, and not every complex network exhibit all the above properties.

Random graph models, i.e. Erdős-Rényi, cannot exhibit all the properties of a real-world complex network. Several models are proposed to generate networks holding such properties. In the following, we briefly describe various complex graph models.

Generative models.

The Barabási-Albert (BA) model [24] is one of the most referred models that generates scale-free networks [5, 13, 17, 46, 55]. This model generates scale-free networks in a preferential attachment setting. A Barabási-Albert graph is generated as follows: (i) it starts with a small number of nodes (m_0); and (ii) each new node is connected to m existing nodes with probabilities proportional to their current degrees. Besides power-law degree distribution, networks generated by the BA model follow some other real-world network properties including average path length and clustering coefficients [28]. For instance, the average path length increases approximately logarithmically with the size of a network, which yields that the BA model has a systematically shorter average path length than a random graph [24]. Moreover, the BA model comes with the flexibility to generate networks with degree distributions with any exponent in $(2, \infty)$ [23].

Another well-known generative model is the Chung-Lu model, which generates a network that has a provable expected degree distribution equal to a given degree distribution [1, 18]. In the Chung-Lu model, the probability of an edge is proportional to the product

[1] http://igraph.sourceforge.net/
[2] https://graph-tool.skewed.de/

of the degrees of its two vertices. Edges can be generated independently by picking vertices proportional to their desired degrees. If all degrees are the same, Chung-Lu reduces to the Erdős-Rényi model. The Chung-Lu model has the problem of preserving clustering coefficients, and compared with the BA model, the Chung-Lu model requires a set of parameters. In recent years, various extensions to the BA model have been proposed, such as spatial and geometric attachment models, in order to support more networks properties [49].

Various other methods can be used for generating synthetic graphs as well. For example, The Block Two-Level Erdős-Rényi (BTER) model [11] is able to capture two network properties, degree distribution and clustering coefficients. BTER is designed based on an assumption derived from the two properties. The Chung-Lu model can be regarded as a special case of the BTER model [30].

A recent model, the hyperbolic unit-disk model, generates random hyperbolic graphs in a fast way [49]. This model uses a spatial data structure, i.e. a polar quadtree adapted to hyperbolic space, to improve the process of all pairwise distance calculation.

Recursive models.

Another popular approach for graph generation is the family of recursive graph models, which is applied to the adjacency matrix of a graph. Recursive MATrix method (R-MAT) model is an example [16]. Further developments on recursive methods include kronecker product and tensor products [3, 33]. Recursive models are efficiently computable. However, as mentioned by a recent study [30], the model has important drawbacks concerning realism and preservation of properties over different graph sizes [30], such as 1) it is expensive to fit to real data, and 2) it rarely closes wedges (a wedge is a path of length 2 and a close wedge forms a triangle in a graph) such that the clustering coefficients are much smaller than that in realistic networks.

Complexity comparison.

For a detailed discussion of graph generation models, readers are referred to [27]. To sum up, we briefly review some of the most popular graph generation models, shown in Table 1 along with corresponding time complexities.

Algorithm	Best Complexity
Erdős-Rényi [21]	$O(\|V\|^2)$
Watts-Strogatz [51]	$O(\|E\|)$
Barabási-Albert [24]	$O(\|V\|^2)$
Chung-Lu [1]	$O(\|V\| + \|E\|)$
R-MAT [16]	$O(\|E\| \log \|V\|)$
BTER [30]	$O(\|V\| + \|E\| \times d_{max})$
Hyperbolic unit disk [49]	$O((\|V\|^{3/2} + \|E\|) \log n)$

Table 1: The complexity of graph generation models. Note that d_{max} is the maximum vertex degree.

2.1.3 Special-purpose models

Following a different line of work, synthetic graphs can be generated using empirical data instead of pure statistical models. For instance, nodes and edges can be sampled using an empirical probability distribution (histogram). Empirical data can be used for more realistic and straightforward generation of multi-relational and RDF datasets, such as social networks. Nevertheless, data-driven methods are for special purpose and not easy to generalize for generating a diverge family of graphs. Moreover, in many domains, empirical data are not available, or are too specific that graph models overfit.

Data-driven approaches, however, are a good fit for generating specific graphs. For example, members of the LDBC council propose various data-enhanced graph generators, such as S3G2 [42] and DataGen [14] that leverage empirical distribution and correlation statistics of Facebook data. For generating heterogeneous information networks, a data-driven graph generator is a popular choice, since theoretical approaches are not matured for multi-relational and multi-type graphs [20].

2.1.4 Other observations

The above discussion provides a broad set of graph generators that rely on sampling from a probability distribution, such as Barabási-Albert, Chung-Lu, S3G2, and DataGen. We can further categorize the graph generators based on the probability distribution to static and dynamic. In the former, the set of items and their probabilities are fixed, e.g. Zipf or Geometric distribution, and data-driven distributions obtained from empirical data. On the other hand, sampling from a dynamic (evolving) probability distribution is challenging. For example, the BA model heavily relies on sampling an existing node with probabilities proportionate to the node's degree. Since the probabilities are changing over time, sampling from a dynamic and long-tailed distribution is computationally expensive. Also, such mechanism comes with strong dependencies that contradict with parallelism, as discussed in the following section.

For many natural graphs that are based on strong rich-get-richer effect, the BA model would be a good fit. Moreover, the BA model is a handy option for growing an existing network while maintaining the network properties, since BA leverages the existing network's degree distribution. Growing an existing method is not straightforward for other approaches, such as R-MAT and Kronecker product.

2.2 Scaling Graph Generation

Generating very large graphs is significant for many research communities. Therefore, efforts are being made to propose efficient parallel and distributed implementations for graph generation models, which focus on both theoretical and practical aspects of the graph models for efficient parallelization [4, 36, 37, 40].

Simple graph generators that are based on sampling having a fixed probability distribution, such as Geometric or empirical Facebook distributions, can be easily converted into parallel versions, because picking multiple samples are independent and can be done in parallel. For instance, the data generators used by the LDBC council such as S3G2 and DataGen are inherently parallel because, for example, generating social degree for each user is independent from others.

Also, methods that are based on matrix operations, such as Kronecker product and recursive graph models are easy to be parallelized. Nevertheless, many other important graph models could not fit a parallelization framework. A large group of generative graph models that are based on dynamic sample selection, such as the preferential attachment models (like BA) are inherently sequential.

Few studies have been done on parallel preferential attachment. Yoo and Henderson [53] propose parallel version of the BA and Kronecker graph generators. The parallel BA algorithm, called PBA, is an approximate method that ignores the sequential nature of the algorithm. PBA is designed to reduce the communication cost in a large cluster: graph vertices are distributed between the processing units, and random selection is done in two steps: 1)inter-processor selection: choose one of the processor in random accord-

ing to the weight of their items; 2)intra-processor selection: pick one of the items from among the items that are assigned to the chosen processor. Alam et al. [4] claim the first distributed-memory parallel algorithms for generating random graphs with exact preferential attachment growth. However, the algorithm obtained reasonable speedup in comparison to the baseline algorithm only when they employ more than 100 processors. Also, Lo et al. propose a distributed version for preferential attachment, which is based on approximation [36].

Studies by Lo et al. shows that the effects of approximation techniques and computationally unsafe (no-lock) parallelization should be carefully considered in parallel versions of BA models, as many parallel configurations result in networks that do not exhibit some properties of the original models, such as degree distribution. Thus, instead of approximation, we focus on proposing exact data structures that can accelerate dynamic roulette wheel selection.

Our goal is to propose a novel sequential algorithm that can significantly outperform the state-of-the-art sequential algorithm in a commodity machine. Our proposed method can be used either as a fast serial algorithm, or as a building block inside two-step parallel generative models, such as PBA. In other words, both the inter-process and intra-process operations in any parallel BA require a fast dynamic roulette wheel, therefore ROLL can substantially improve PBA and other distributed preferential attachment methods [36, 53].

3. THE BASELINE ALGORITHM

In this section, we explain two sequential algorithms, baseline and state-of-the-art, for generating scale-free graphs under Barabási-Albert (BA) model. The former employs a simple roulette wheel (SimpleRW) and the latter is based on stochastic acceptance (SA).

Preliminaries and notations.

Let $G = (V, E)$ be an undirected unweighted graph such that $V = \{v_1, ..., v_n\}$ is the set of nodes and E is the set of edges. The cardinality of vertices and edges is denoted by $|V| = n$ and $|E|$, respectively. For each node v_i, $deg(v_i)$ denotes the degree of v_i, that is the number of edges that are connected to v_i. Also, let $\mathbf{B}$ be the *set of unique degrees* for all nodes in the graph, that is $\{deg(v_i) : 1 \leq i \leq n\}$. $|\mathbf{B}|$ is hence the number of unique degree values in the graph.

As discussed further, the *nodes* of a graph are represented as *items* in its corresponding roulette wheel, so we use these two terms interchangeably. We further suggest a tree data structure that contains groups of nodes, called **buckets**. In order not to confuse the nodes of such trees with the nodes of the graph, we use Rnode for pointing to nodes of a tree (i.e. branches and leafs). Table 2 briefly summarizes the notations used in this paper.

Notation	Description
$\|V\| = n$	# of nodes (items)
$\|E\|$	# of edges
$\|B\|$	# of unique degrees
m_0	# of initial nodes
m	# of links per node
L_{CW}	Average code word length [3].
$deg(v_i)$	Degree of node v_i
S_n	Total degrees $= \sum_{i=1}^{n} deg(v_i)$

Table 2: Brief description of notations

[3]Average code word length of a tree with weighted leaf nodes is the weighted average distance of leaf nodes to the root node. See Section 4.2.2

The BA model has three parameters: the number of nodes in the final graph $n = |V|$, the number of initial nodes m_0 and the number of connections per new node m. The networks generated by BA model are called BA networks. BA networks have power law degree distribution. A degree distribution is defined power law iff the probability of a node with degree d is given by $P(k) \sim k^{-\gamma}$ where γ is a positive constant.

3.1 SimpleRW Algorithm (Baseline)

To generate a synthetic scale-free graph using the Barabási-Albert model, we begin with an empty network having m_0 nodes and no edges. For the next $n - m_0$ nodes, each node is connected to m distinct nodes among all the existing nodes in the graph. The probability of node v_i for being linked to the new node is proportional to the degree of the node $deg(v_i)$. Algorithm 1 describes this procedure in detail.

Algorithm 1 BA model

1: **Input:** $n = |V|,\ m,\ m_0$
2: **Output:** $G(V, E)$: A scale-free graph containing n nodes
3: Add m_0 nodes to G
4: **for** i from $(m_0 + 1)$ to n **do**
5: Create new node v_i
6: **for** j from 1 to m **do** ▷ Connect v_i to m existing nodes
7: Select a node $v_k \in \{v_1..v_{i-1}\}/\{v_j|(v_k, v_j) \in E\}$ based on probability in Equation 1.
8: $E \leftarrow (v_i, v_k)$ ▷ Add the new edge
9: $deg(v_i) \leftarrow deg(v_i) + 1$
10: $deg(v_k) \leftarrow deg(v_k) + 1$

Consider a set of k nodes $v_1, v_2, ..., v_k$. When the next node v_{k+1} is created, it is connected with m new edges to m distinct nodes among $v_1, v_2, ..., v_k$. For each new edge from the new node, the probability for each of the existing graph nodes for being selected as the target of the edge and being attached to v_{k+1} is proportional to the degree of the node.

$$p(v_i) = \frac{deg(v_i)}{\sum_{l=1}^{k} deg(v_l)} \tag{1}$$

The m nodes that are to be connected to each node should be sampled without replacement, i.e. if a node is selected to be connected to the newborn node, its selection probability in the consecutive selections become zero. This mechanism is simply handled via rejection sampling: If the selected node has already been connected to the newborn node, the sampling is rejected and repeated again. Nonetheless, in most problems such as the Barabási-Albert algorithm we have $m \ll n$, hence the rejection probability (i.e. the probability that one of the already selected nodes are selected again) is very low and thus the rejection rarely happens.

The preferential attachment algorithm described above adopts a roulette wheel mechanism to select the target nodes. We briefly explain the simple roulette wheel method for selecting a random node (Algorithm 1, line 7). A roulette wheel is a list of items with their corresponding selection probabilities. In order to select a random item, we should generate a random number $0 < x \leq 1$ and select v_k for which $P(v_{k-1}) < x \leq P(v_k)$. Note that $P(v_i)$ denotes the cumulative probability, i.e. $P(v_k) = \sum_{i=1}^{k} p(v_i)$.

As an example, consider the sample graph in Figure 1 which has 26 nodes generated by the BA algorithm ($m_0 = m = 2$). The total degree of the nodes in the graph is $S_n = \sum_{i=1}^{n} deg(v_i) = 2|E| = 2m(n - m_0) = 2 \times 2(26 - 2) = 96$. To continue the

v_i	$w_i = \deg(v_i)$	$p(v_i)$	$P(v_i)$	$\sigma_i(v_i)$
v_0	3	0.03	0.03	3
v_1	13	0.14	0.17	16
v_2	6	0.06	0.23	22
v_3	10	0.10	0.33	32
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
v_{12}	3	0.03	0.71	68
v_{13}	2	0.02	0.73	70
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
v_{24}	2	0.02	0.98	94
v_{25}	2	0.02	1.00	96

Table 3: Roulette wheel of the graph in Figure 1

growth of the graph, the next new node v_{26} should be connected to $m = 2$ nodes among $\{v_1, ..., v_{25}\}$. Table 3 shows the degree-proportionate roulette wheel corresponding to this graph. We want to sample two nodes from the roulette wheel and connect them to v_{26}. For each random selection, a random number should be generated. Suppose that the generated random numbers for the first and second connections are 0.30 and 0.72, respectively. According to the roulette wheel in Table 3, the first target node is v_3 $(0.23 < 0.30 \leq 0.33 = P(v_3))$ and the second node is v_{13} $(0.71 < 0.72 \leq 0.73 = P(v_{13}))$.

In the above example, we had to compute $P(v_1)$ through $P(v_{13})$ in order to find the item that is corresponding to the generated random number (0.72). When a node is added to the network, it is also added to the roulette wheel. Also, whenever a new edge connects two nodes in the graph, the probability of nodes at its both ends are increased according to their new degrees and the probability of all other items in the roulette wheel should be updated as well to ensure that the total probabilities sum to one. As a result, the roulette wheel will change after each selection and the cumulative probabilities need to be recomputed in the next iteration. To alleviate this problem, the denominator in Equation 1 can be eliminated because it is constant. Hence, the probabilities $p(v_1), \ldots, p(v_n)$ can be converted into weights $w_1, \ldots, w_n$ such that $w_i = deg(v_i) = S_n \times p(v_i)$. Similarly, instead of cumulative probabilities, we can compute partial sums $\sigma_1, \ldots, \sigma_n$ where $\sigma_i = S_n \times P(v_i) = \sum_{j=1}^{i} w_j$ (see Table 3). To sample from the roulette wheel, we generate a random integer $r \in [1..S_n]$ and select v_k for which $\sigma_{k-1} < r \leq \sigma_k$.

The advantage of using weights in place of probabilities is that some partial sums do not change and do not need to be re-calculated. For instance, in the previous example it is not required to re-compute σ_0 through σ_2 since they are not changed after increasing the degrees of v_3 and v_{13}. Besides, using weights (i.e. degree of nodes) instead of the probabilities is more efficient because weights are stored as integer values. The simple roulette wheel algorithm for preferential attachment is illustrated in Algorithm 2. Whenever a node v_k is connected to the new node v_n, the partial sums for $\sigma_k, ..., \sigma_{n-1}$ are incremented by one and σ_n is incremented by two. As a result, the computational complexity of updating partial sums for each link is $O(n = |V|)$. Also, the random number generation requires a constant time, which is $O(1)$. Therefore, the total time taken to generate the whole graph containing $|E|$ links is $O(|E| \cdot (|V| + 1)) = O(mn^2)$.

3.2 SA Algorithm (State-of-the-art)

Lipowski and Lipowska recently proposed an alternative roulette-wheel selection algorithm based on stochastic acceptance instead of searching [35]. In this algorithm, the target node is selected in an iterative two-step procedure, as follows:

Algorithm 2 SimpleRW Algorithm

1: **Input:** A set of items $\{v_1, ..., v_n\}$ with their weights $\{w_1, ..., w_n\}$
2: **Output:** A randomly chosen item v_k.
3: $S_n = \sum_{i=1}^{n} w_i$ ▷ In Barabási-Albert alg., $S_n = 2 \times |E|$
4: Initialize $\sigma_0 \leftarrow 0$
5: Uniformly generate a random number $r \in [1..S_n]$
6: **for** i from 1 to n **do**
7: $\quad \sigma_i \leftarrow \sigma_{i-1} + w_i$
8: $\quad$ **if** $\sigma_i \geq r$ **then**
9: $\quad\quad$ Return v_i

1. Uniformly select a random node v_i.

2. Accept the selection with probability $p_{\text{acc}}(v_i) = w_i / w_{\max}$, where $w_{\max}$ is the maximum weight in the roulette wheel. If the node is not accepted, then go to step 1.

This method is effective and is applied in various applications.

Complexity of SA. In general, the complexity of SA for selecting a single item from a roulette wheel is $O(w_{\max}/\overline{w})$ where $\overline{w}$ is the average weight of items in the roulette wheel. In the Barabási-Albert model, the expected weight (i.e. average degree of nodes) is $\overline{w} = 2|E|/|V| = 2(mn - m_0)/n \approx 2m$. Also, the expected value of $w_{\max}$ (maximum degree) for a graph with power law degree distribution ($p(k) \sim k^{-\gamma}$) is $\mathbb{E}(w_{\max}) \approx n^{1/(\gamma-1)}$ [9]. In Barabási-Albert graphs, $\gamma \approx 3$, thus $\mathbb{E}(w_{\max}) \approx \sqrt{n}$. Putting these together, the complexity of SA for selecting one node from the roulette wheel, when the items are generated using the Barabási-Albert model, is $O(\sqrt{n}/m)$. In order to construct the entire graph, BA uses the roulette wheel $n \times m$ times, thus the total complexity is $O(n^{3/2})$.

Even though the complexity of the SA is better than the simple roulette wheel, it is still not efficient for generating Barabási-Albert graphs. Roughly speaking, the distribution of weights follows a power law, which implies that $w_i \ll w_{\max}$ for most nodes. Therefore, the acceptance probability $p_{\text{acc}}(v_i)$ is usually very low, so while it is not required to traverse or modify a large part of the roulette wheel items, we have to re-sample until after the final node is reached. Such re-sampling for large graphs ($|V| > 10^6$) is a time-consuming operation.

4. THE PROPOSED METHOD

In this section we describe our proposed roulette-wheel data structures for accelerating the preferential attachment algorithm. This data structure is designed to speed-up the Barabási-Albert algorithm, but in general it can be used for dynamic roulette wheels where the selection probability of items follows an arbitrary long-tailed distribution.

Sampling from a roulette wheel is easy when the selection probabilities of items are fixed. However, when the roulette wheel is dynamic, i.e. the selection probabilities of items change during the random process, it takes a long time to re-compute the cumulative probabilities (or partial sums of weights). In other words, we need a roulette wheel data structure that efficiently provides two operations: 1) *Sample* a random node and 2) *Update* the weight of an item, e.g. when a connection is made between two nodes in a Barabási-Albert graph, the degree of both nodes is increased, so their weights in the roulette wheel should be incremented as well.

In this section, we provide two modifications on the SimpleRW algorithm, which make random selections faster. The first modification is to reduce the problem size by grouping together items with the same probabilities. In our second modification, we suggest a different algorithm for sampling from a dynamic list of items in which the target item is selected using multiple random binary decisions instead of computing cumulative probabilities. This is done by a specific data structure named ROLL-tree, which allows for efficient selection of the target node in a divide-and-conquer manner.

4.1 Roulette Wheel Buckets (ROLL-bucket)

The intuition behind ROLL-bucket is that if the number of items inside a roulette wheel are very large and their selection probabilities are rational numbers with common denominator, large groups of nodes have the same selection probabilities. For example, recall that the degree distribution of scale-free graphs follows a power law and the degrees are integers. Theoretically speaking, the expected fraction of nodes that are connected to exactly k edges is $p(k) \sim k^{-\gamma}$. As a consequence, large groups of nodes, especially those with low degrees, have the same degree and thus have equal selection probabilities.

Following this intuition, our idea is to group the nodes into buckets (i.e. subsets of nodes) such that the nodes with the same weight, are stored in the same bucket, say $B_d = \{v_i | w_i = d\}$. Therefore, we keep a list of buckets $\mathbf{B}$ that contains all non-empty buckets. For example, in the graph of Figure 1, the 26 nodes are grouped into 7 buckets, as shown in Table 4. In this table, $p(B_d)$ and $P(B_d)$ are the probability density function (PDF) and cumulative density function (CDF) of B_d, respectively.

The sampling procedure can be broken into two phases. At first, a bucket is selected among all the available buckets. Then, one of the items inside the selected bucket is randomly selected. The chance of each bucket for being selected is the total chances of the items it contains. More precisely, the probability of selecting a bucket B_d is $p(B_d) = (d \times |B_d|)/S_n$, where $|B_d|$ is the number of items in B_d. Recall that $S_n = \sum_{i=1}^{n} w_i$ is constant and can be eliminated by converting probabilities to weights. If the selected bucket contains multiple items, one of its items is randomly chosen with equal probability. The proof of correctness of this two phases approach is explained in Theorem 1.

Whenever the weight of an item in ROLL-bucket changes, it should be moved from its current bucket to the bucket corresponding to its new weight. Particularly for the Barabási-Albert algorithm, when a node is connected to a new edge and its degree increases from d to $d+1$, it should be moved from B_d to B_{d+1}. The procedure is illustrated in Algorithm 3.

Theorem 1. *The two-stage sampling in ROLL-bucket is equivalent to the original roulette wheel selection, i.e the selection probability of items is equal.*

Proof. Given an item v_i with weight w_i, the probability of selecting v_i in the roulette wheel is $\frac{w_i}{S_n}$. For the two-stage sampling, the chance of selection equals the probability of selecting bucket B_{w_i} (which is $\frac{w_i \times |B_{w_i}|}{S_n}$) times the probability that v_i is selected from all items in B_{w_i}, that is $\frac{1}{|B_{w_i}|}$. Hence:

$$p(v_i) = \underbrace{\frac{w_i \times |B_{w_i}|}{S_n}}_{p(B_{w_i})} \times \underbrace{\frac{1}{|B_{w_i}|}}_{p(v_i|B_{w_i})} = \frac{w_i}{S_n}$$

□

Algorithm 3 ROLL-bucket

Input: Set of buckets: $\mathbf{B} = \{B_i | \exists v, deg(v) = i\}$
Output: A randomly chosen node v_k.
$\mathbf{S_n} = 2 \times |E|$
Initialize $\sigma = 0$ ▷ The partial sum
Uniformly generate a random number $r \in [1..S_n]$
for each $|B_i| \in \mathbf{B}$ **do**
  $deg(B_i) \leftarrow i \times |B_i|$
  if $\sigma + deg(B_i) \geq r$ **then** ▷ Target node is inside B_i
    Generate $r' \in [1..|B_i|]$ with uniform distribution
    $v_k \leftarrow r'$th node in B_i
    Move v_k from B_i to B_{i+1}
    Return v_k
  else
    $\sigma \leftarrow \sigma + deg(B_i)$ ▷ Updating the partial sum

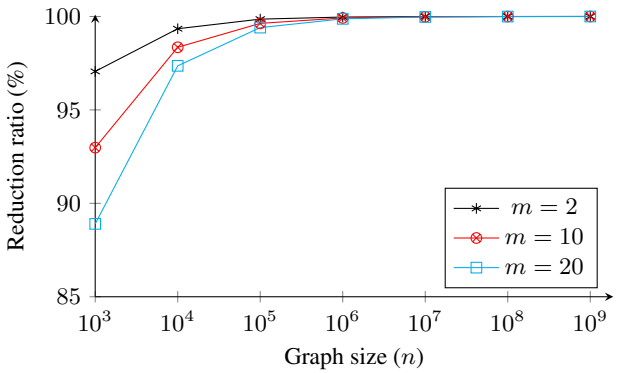

Figure 2: Ratio of items reduced to buckets in ROLL-bucket while varying n and $m = m_0$.

ROLL-bucket effectively reduces the problem space. Because the selection probabilities of all items inside each bucket are equal, it is not necessary to count the partial sums (i.e. cumulative probabilities) inside buckets. The partial sums are only computed over buckets, and since the number of buckets is very small compared to the number of items, a shorter list is traversed for finding the target item. For the case of Barabási-Albert preferential attachment, ROLL-bucket reduces the complexity of single node selection from $O(|V|)$ to $O(|\mathbf{B}|)$ where $|\mathbf{B}|$ is the number of unique weights, e.g. number of unique degrees in the graph. Figure 2 illustrates ratio of items reduced to buckets in ROLL-bucket for various m and n values. Reduction ratio is $(1 - \frac{|B|}{n}) \times 100$. This figure shows the advantage of bucketing approach in ROLL-bucket for reducing the problem size by more than 98% when $n = 10^4$. This figure also shows that the larger the graph, the more reduction we obtain, thanks to the power-law degree distribution.

In order to speedup ROLL-bucket, we have to traverse the buckets in a way that minimizes the number of buckets that are traversed before reaching the target one. A greedy approach would be to select buckets according to their weights in descending order, so that a large portion of partial sums are scanned by visiting few buckets. However, keeping buckets sorted by their weights is costly. An alternative to this method, as already suggested in Algorithm 3, is to traverse buckets by their degree, starting from the lightest bucket, i.e. B_m. Figure 3 compares the number of buckets traversed by each of the two options, as well as the generation time. Results show that while scanning buckets in decreasing order of their weights examines fewer number of buckets, it is slower in practice due to the cost of keeping buckets sorted by their weights.

Bucket	Degree d	#Nodes	$p(B_d)$ (PDF)	$P(B_d)$ (CDF)	Nodes
B_2	2	13	26/96	26/96	$\{v_8, v_{13}, v_{14}, v_{16}, v_{17}, v_{18}, v_{19}, v_{20}, v_{21}, v_{22}, v_{23}, v_{24}, v_{25}\}$
B_3	3	4	12/96	38/96	$\{v_0, v_7, v_{10}, v_{12}\}$
B_4	4	2	8/96	46/96	$\{v_6, v_{15}\}$
B_5	5	3	15/96	61/96	$\{v_4, v_9, v_{11}\}$
B_6	6	2	12/96	73/96	$\{v_2, v_5\}$
B_{10}	10	1	10/96	83/96	$\{v_3\}$
B_{13}	13	1	13/96	96/96	$\{v_1\}$

Table 4: ROLL-buckets for selecting the next node for the graph in Figure 1

As a results, our suggested algorithm for ROLL-bucket scans buckets in ascending order of their degrees, which is described before.

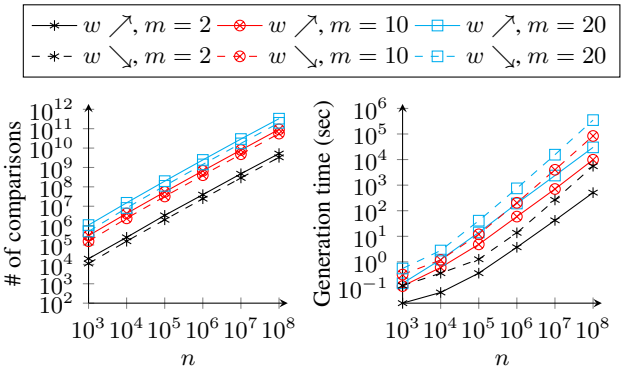

Figure 3: ROLL-bucket with sorting by degrees in ascending order (d ↗) or by weights in descending order (w ↘).

4.2 ROLL-tree

Although ROLL-bucket can significantly reduce the problem size, still the buckets should be sequentially traversed to calculate the partial sums over the buckets. In other words, if the target node corresponding to a random number is in B_i, then it is inevitable to compute the partial sums of all non-empty buckets between B_1 and B_i. Hence, the complexity of this linear search is $O(|\mathbf{B}|)$.

Algorithm 4 Random selection with ROLL-tree

```
Input: Tree: The ROLL-tree containing all items
Output: A randomly chosen node v_k.
Rnode ← Tree.root
while Rnode is not a bucket do
    Uniformly generate r ∈ [1 ... (Rnode.w_R + Rnode.w_L)]
    if r ≤ Rnode.w_L then
        Rnode ← Rnode.Lchild
    else
        Rnode ← Rnode.Rchild
                ▷ Perform binary selections until a leaf is reached.
B_i ← bucket(Rnode)
v_k ← Uniformly select an item in B_i
Return v_k
```

We suggest a novel roulette wheel data structure, named ROLL-tree. Instead of ROLL, which keeps a sequential list of buckets in memory, ROLL-tree stores the buckets in a weighted binary tree. At first, we describe the random selection algorithm on the ROLL-tree. Figure 4, shows a ROLL-tree corresponding to the toy graph of Figure 1, which has 7 leafs (buckets) containing 26 items (The buckets are the same as in ROLL, shown in Table 4). Every leaf in the tree is a bucket and its weight is the same as in ROLL, i.e. $w_i = i \times |B_i|$. the weight of an internal node is the sum of the weights of its left and right children, which is the total weight of buckets in its subtree. To avoid confusing the nodes of ROLL-tree with nodes of the generated graph, we denote ROLL-tree nodes as Rnode, which consists of both buckets (leafs) and internal nodes.

4.2.1 *Sampling from ROLL-tree*

In ROLL-tree, random selection is done using a divide and conquer approach. Starting from the root Rnode, we know that the target item is in either the right or left subtrees, so we select one of the subtrees according to their weights. The weight of each subtree is the total probability that one of its leafs are selected. This procedure is repeated until a leaf is reached. The ROLL-tree's random selection algorithm is described in Algorithm 4.

We explain this algorithm on our toy example. Suppose that we want to randomly select a random node among $v_0, \ldots, v_{25}$ to be connected to the newborn node v_{26}. Assume that the random generator draws the following numbers uniformly between $[0, 1]$: $\{0.83, 0.12, 0.54, 0.39, ...\}$. We first start from the root Rnode and select either its right or left children with probabilities proportional to their weights. In root Rnode, the probabilities for selecting the right and left children are $\frac{49}{96}$ and $\frac{47}{96}$, respectively. According to the first random number, the right child of the root Rnode is selected ($\frac{49}{96} < 0.83$). Now we consider the right subtree of the root Rnode. Similarly, we select its left subtree if the random number is below $\frac{27}{27+20}$, or its right subtree otherwise. According to the sequence of the generated random numbers, in both the second and third binary decision, the left branches are selected. ($\frac{27}{47} > 0.12, \frac{15}{27} > 0.54$). As a result, B_5 is returned by ROLL-tree as the selected bucket. Now we should select one of the three nodes in B_5, namely v_4, v_9, and v_{11}. According to the next random number (0.39), the second node (v_9) is chosen. Recall that since the selection probability of nodes inside a bucket are equal, there is no need to compute the partial sums inside a bucket.

The advantage of ROLL-tree is that it avoids traversing a large list of buckets. Instead, it stores a summary of buckets in the internal nodes, such that each internal node in ROLL-tree contains the total weight of all buckets in its subtree.

4.2.2 *Updating ROLL-tree*

When the weight of a bucket changes, the value of the internal Rnodes that are its ancestors should be updated accordingly. In Barabási-Albert algorithm, for example, whenever a node v_k with degree d is selected and connected to the newborn node, its degree increases, so it should be moved from B_d to B_{d+1} and the internal Rnodes that are an ancestor of either B_d or B_{d+1} should be updated accordingly. Consider the previous example and suppose that v_9 is selected and its degree is incremented by one. We should move v_9 from B_5 to B_6. Now the weight of B_5 and B_6 should be updated, i.e. $w_{B_5} = 5 \times (3 - 1) = 10$, $w_{B_6} = 6 \times (2 + 1) = 18$. Similarly, the weight of their parent, grandparent, and the root Rnode is updated as $27 \rightarrow 28$, $47 \rightarrow 48$, and $96 \rightarrow 97$, respectively in a bottom-up order. This procedure is described in Algorithm 5.

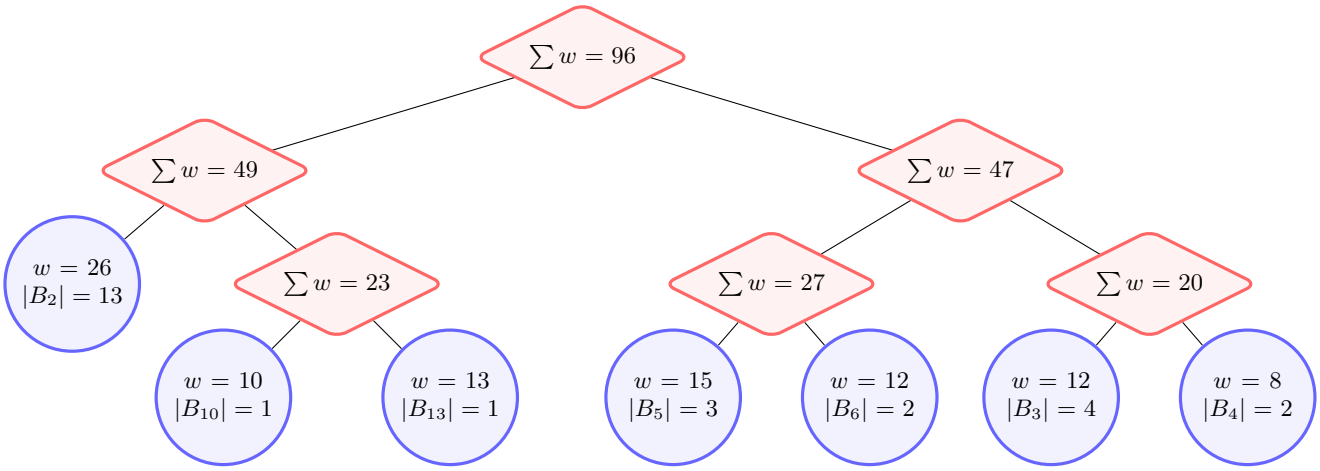

Figure 4: ROLL-tree of 26 nodes in 7 buckets when $m_0 = m = 2$. The internal nodes (diamonds) are decision nodes holding sum of weight of left and right child. The leaf nodes (circles) store w and $|B_d|$, weight and the number of nodes with degree d respectively.

Algorithm 5 Updating the weight of an item in ROLL-tree

Input: Tree, $v, w_{\text{old}}, w_{\text{new}}$
$B_{w_{\text{old}}} \leftarrow B_{w_{\text{old}}} - v$ ▷ Remove v from the old bucket.
if $B_{w_{\text{old}}}$ is empty now **then**
  Delete(Tree,Rnode($B_{w_{\text{old}}}$)) ▷ Algorithm 7
else
  Rnode = Rnode($B_{w_{\text{old}}}$)
  Rnode.w = $w_{\text{old}} \times |B_{w_{\text{old}}}|$
  while Rnode $\neq$ Tree.root **do** ▷ Update weights of parents
    Rnode $\leftarrow$ Rnode.parent
    Rnode.w = Rnode.Rchild + Rnode.Lchild
$B_{w_{\text{new}}} \leftarrow B_{w_{\text{new}}} \cup v$ ▷ Add v to the new bucket
if Rnode($B_{w_{\text{new}}}$) is not in the Tree **then**
  Insert(Tree,Rnode($B_{w_{\text{new}}}$)) ▷ Algorithm 6
else
  Rnode = Rnode($B_{w_{\text{new}}}$)
  Rnode.w = $w_{\text{new}} \times |B_{w_{\text{new}}}|$
  while Rnode $\neq$ Tree.root **do** ▷ Update weights of parents
    Rnode $\leftarrow$ Rnode.parent
    Rnode.w = Rnode.Rchild + Rnode.Lchild

Whenever an item gets a new weight w_{new} and $B_{w_{\text{new}}}$ does not exist in the tree, it is *inserted* into the ROLL-tree as a new leaf. Similarly, in case that the previous bucket $B_{w_{\text{old}}}$ becomes empty after removing its last item, it should be *removed* from the ROLL-tree. These two tree operations, i.e. insertion and removal, are explained in the followings, respectively.

Inserting to ROLL-tree.

The insertion can be done in several ways. Obviously, any binary tree that contains all the elements can be used to select random items. Generally, the complexity of reaching a bucket in an arbitrary binary tree is $O(L_{CW})$, where L_{CW} is the average code word length of the tree, i.e. $\frac{1}{|\mathbf{B}|}\sum_{i=1}^{N} w_i \times L_{B_i}$. A simple solution would be to keep the buckets in a *complete* binary tree, such that new buckets are added at the end of the tree in the first available free space. This will ensure that the depth of the tree is $\log n$, so the complexity of insertion, deletion, and random selection would be $O(\log n)$. In case that the selection probability of all leaf Rnodes are equal, it is the optimal solution. However, since the weights of buckets are not equal, e.g. $w_{B_k} = k \times k^{-\gamma} = k^{1-\gamma}$ in scale-free graphs, putting all leafs in the same depth is not optimal. Instead, we should put leafs with higher selection probability in a shorter depth so that they are fetched easier. Since our objective is to increase the efficiency, we might think of a tree which minimizes the time taken to reach the target nodes. Apparently, the amount of time taken to sample a node is proportional to the distance of its corresponding bucket from the root node, which is how many times we perform a binary decision between the right and left subtrees. To reduce the sampling time, buckets that are more likely to be selected, i.e. have higher weights, should be closer to the root node. This idea recalls the concept of Huffman coding tree, in which the distance of more-frequent symbols is closer to root, and thus their assigned code word has a shorter length. More specifically, each bucket can be assigned a unique binary code, in which a 0/1 in i'th index represents moving to the left/right subtree in the i'th depth of the ROLL-tree. In Figure 4, for example, B_3 is assigned the code 110 which means "right $\rightarrow$ right $\rightarrow$ left", that represents the path from the root node to B_3. The random selection in ROLL-tree can be thought of as stochastically finding a path from the root to a bucket. In other words, our objective is to minimize the average code word length.

In the following, we first discuss about Huffman coding trees, which gives the theoretically optimal tree having minimum expected number of samplings. Then, we criticize the high computational cost of using Huffman trees. Finally, we propose a light greedy approach that is close to the optimal bound of Huffman.

Optimal ROLL-tree (Huffman). Our objective is to insert an Rnode into the tree such a way that the average code word length is kept minimized. If the roulette wheel were static, the Huffman coding tree would be proved to be the optimal solution [19]. However, Huffman tree is not efficient for our problem due to its large computation cost. The main reason is that the original Huffman tree is not adaptive, hence we need to re-build the tree once the selection probability is changed, e.g. when the degree of a node is increased or a new node is created.

In order to design an online solution for weigh-balancing the tree, one may think of *adaptive* Huffman coding algorithms, such as FGK [4] algorithm [22, 25, 29] and the Vitter [48], which allow updating the probabilities. These methods guarantee that the in-

[4]Faller-Gallagher-Knuth

sertion time has $O(L_{CW})$ complexity. However, adaptive coding algorithms have an important shortcoming: they do not support removing an item (code) or down-weighting an item. Recall that many buckets become empty and hence they have to be removed from ROLL-tree. Also, when an item is sampled from a bucket, its degree is increased and it is moved to the bucket with higher degree. Therefore, the old bucket loses an item and its weight should be decreased accordingly. To our knowledge, there is no adaptive Huffman coding algorithm that efficiently supports deleting codes (i.e. Rnodes). As a result, we are unable to use adaptive Huffman tree algorithms for weight-balancing the tree.

Optimality $\neq$ Performance. In order to minimize the execution time of the algorithm, we should minimize the time taken by the roulette wheel to sample buckets, plus the tree maintenance time, i.e. the time taken to update the weights in the tree, add new buckets and remove the empty ones. While shorter average code word length results in faster sampling, other operations for tree maintenance should be lightweight as well. Roughly speaking, the rationale behind using a weigh-balanced tree is to reduce the execution time, hence the weight-balancing procedure should not take too much time itself. Therefore, a computationally expensive weigh-balancing algorithms that over-optimize the average code word length by sacrificing time, such as adaptive Huffman encoders, are not suitable for this problem.

We use neither adaptive Huffman coding nor any rotation-based algorithm for weigh-balancing the tree. Instead, we turn our attention on how the stochastic behavior of the preferential attachment algorithm can help us in keeping ROLL-tree weight-balanced. We focus on some interesting properties that are observed in the preferential attachment process. For example, buckets with small index values, corresponding to low degrees such as B_2 and B_3 have higher selection probabilities. Note that $P(B_k) = k^{1-\gamma}$ where $2 < \gamma < 3$, hence the less the index value, the more the expected selection probability. As a consequence, buckets with lower degree should have smaller code-word lengths and should be kept closer to the root. Moreover, Low-degree buckets are more stable and almost never become empty. On the contrary, a bucket with a large index value, e.g. B_{100m} and above, usually contains a single element, and this single element is likely to be selected soon and then the bucket should be removed from the tree.

As an example, in our toy graph, it is very unlikely for B_2 to be removed, as it already have 13 elements. Also, B_2 has the highest selection chance in the round which means it should be close to the root. On the contrary, B_{13} is very unstable. While it also has a considerable selection chance, it becomes empty right after its single node (v_1) is selected, and then we should remove B_{13} and insert the new bucket B_{14}.

Performance-aware ROLL-tree maintenance. Inspired by the above, we adopt a greedy approach that heuristically keeps the ROLL-tree almost weight-balanced, thanks to the gradual insertions. The heuristic is to always insert the new bucket to the lighter subtree. ROLL-tree gradually grows by inserting new buckets of single item by a top-down traversal from root. At each level of this top-down traversal, the heuristic inserts the new bucket to the subtree that is lighter. This procedure is repeated until a leaf is reached. Then, the leaf is replaced by a new Rnode that has this leaf Rnode and the new bucket as its children. The heuristic is shown in Algorithm 6.

The above heuristic not only incurs substantially smaller processing cost in comparison to the Huffman tree, but also results to a near optimal tree, i.e. minimized code-word length, as observed in our empirical experiments. Figure 5 compares the code word length of ROLL-tree and its ideal value (Huffman code word length) for Barabási-Albert graph generation. Based on the result, the average code-word length in ROLL-tree is very close to its optimal value (Huffman) for various n and m.

Algorithm 6 Inserting Rnode in ROLL-tree

```
1: Input: Tree, newTNode
2: Rnode ← Tree.root
3: while Rnode is not leaf do
4:     if Rnode.Rchild.w > Rnode.Lchild.w then
5:         Rnode ← Rnode.Lchild
6:     else
7:         Rnode ← Rnode.Rchild
8: branchRnode ← Initialize a new internal Rnode
9: branchRnode.Parent ← node.Parent
10: branchRnode.Rchild ← node
11: branchRnode.Lchild ← newNode
12: for pRnode ∈ newRnode.ancestors() do ▷ In bottom-up order
13:     pRnode.w ← pRnode.Lchild.w + pRnode.Rchild.w
```

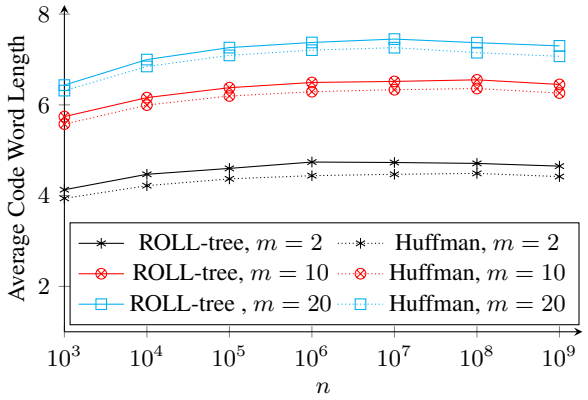

Figure 5: Comparison of the code-word length in ROLL-tree and Huffman tree, regarding to the final weight of items at the end of the process

Deleting from ROLL-tree.

In order to delete a bucket from the tree, we simply disconnect its corresponding Rnode from the tree. Then, the sibling of the deleted Rnode is replaced by its parent. This will ensure that none of the internal Rnodes have a null child and the tree is kept full. This procedure is shown in Algorithm 7. Note that while removing random Rnodes from the tree might decrease the balance, i.e. increase the average code word length, but this balance will be recovered as new Rnodes are inserted.

Algorithm 7 Deleting Rnode from ROLL-tree

```
1: Input: Tree, emptyTNode
2: Rnode ← emptyTNode.Parent
3: Rnode.removeChild(emptyTNode)
4: if Rnode ≠ Tree.root then
5:     Remove Rnode.parent and replace it with Rnode
6: for pRnode ∈ Rnode.ancestors() do ▷ In bottom-up order
7:     pRnode.w ← pRnode.leftChild.w + pRnode.rightChild.w
```

Decreasing tree operations.

Finally, we improve ROLL-tree by compensating deletion and insertion of Rnodes by replacement of buckets. Given a ROLL-

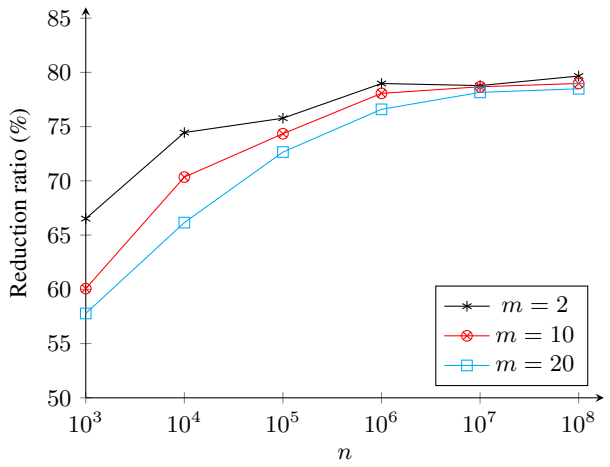

Figure 6: Ratio of the skipped tree operations on ROLL-tree

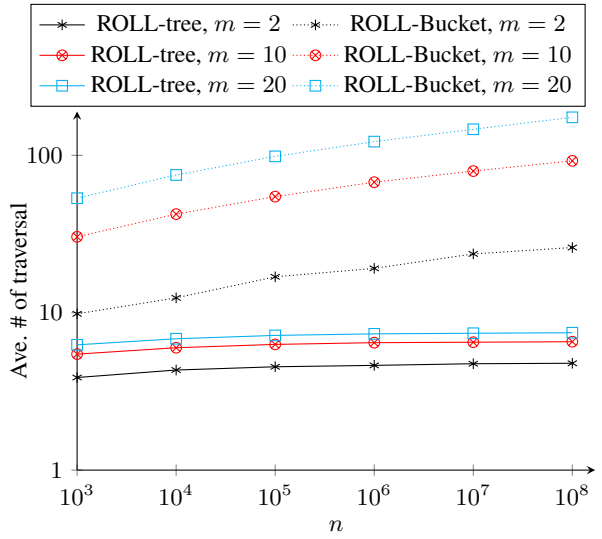

Figure 7: Average # of items traversed to the target sample

tree T, this replacement is possible if we sample an item i from a bucket B_d that only contains i and $B_{d+1} \notin T$, like $v_1 \in B_{13}$ in our toy example. When item i in bucket B_d is sampled, its degree is increased by one and thus it should be moved to B_{d+1}. If B_{d+1} does not exist in ROLL-tree, instead of deleting B_d then inserting B_{d+1} we replace B_d with B_{d+1}. Figure 6 shows how significantly this replacement reduces the number of deletion and insertion of Rnodes. In this figure reduction ratio is $(1 - \frac{rep}{del+ins}) \times 100$ such that rep, del and ins are replacement, deletion and insertion of Rnodes, respectively. Figure 7 compares the number of items traversed in ROLL-bucket and ROLL-tree. Experiments show that the idea of using ROLL-tree, along with the techniques for minimizing the code word length, effectively minimize the number of operations for each sampling.

5. ANALYSIS

In this section, we evaluate the two proposed algorithms along with the baseline method (SimpleRW) and the state of the art (SA) described in Section 3. We perform theoretical study, followed by experimental evaluations, in order to get a more complete view on the performance of our algorithms.

Algorithm	Sample	Maintenance
SimpleRW	$O(n^2m)$	$O(nm)$
SA	$O(n^{3/2})$	$O(nm)$
ROLL-bucket	$O(nm \times \|B\|)$	$O(nm \times \|B\|)$
ROLL-tree	$O(nm \times \|L_{CW}\|)$	$O(nm \times \|L_{CW}\|)$

Table 5: Complexity of the roulette wheel algorithms for sampling and maintenance

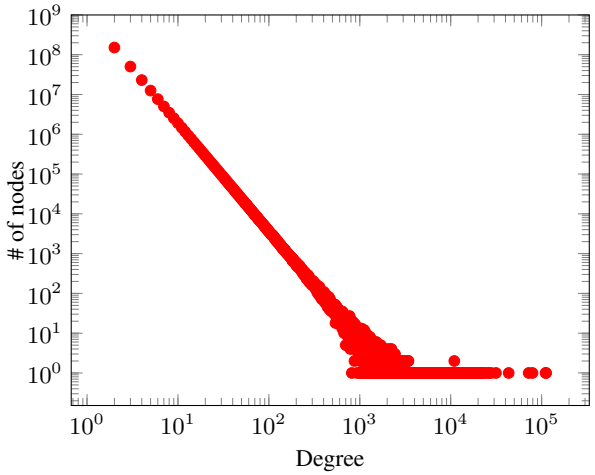

Figure 8: Degree distribution of the generated BA network with $n = 10^9$, $m = 2$ via ROLL-tree.

5.1 Theoretical Analysis

We summarize time and memory complexities of the four algorithms and then provide experimental results to verify the results.

Time Complexity. The generation time is the time taken to sample from the data structure, plus the time for maintaining the roulette wheels, such as updating the weights, buckets, and trees. The complexities are summarized in Table 5.

Memory Complexity. As discussed before, the complexity of all four algorithms are the same, because they all need to keep the weight of buckets in memory. The memory complexity of SimpleRW and SA is $O(n)$. ROLL-bucket and ROLL-tree take some additional memory for the buckets, hence their complexity is $O(n + |B|)$ which is effectively equal to to that of SimpleRW and SA because $|B| << n$.

Degree distribution. We measured the degree distribution of a one billion node graph resulted by ROLL-tree in Figure 8 with parameters $n = 10^9, m = m_0 = 2$. This figure shows a heavy tailed distribution of the generated graph that is a feature of the real-world power-law networks. In order to verify the scale-free distribution of the generated graph, we compute the γ exponent of this distribution which certifies the accuracy of the BA model generated via ROLL-tree. The value of γ, as measured by curve fitting, is $\gamma = 2.72$ which satisfies the condition $2 < \gamma < \infty$.

5.2 Experimental Analysis

In this section, we conduct experiments to study the performance of the algorithms. First, we perform a micro-level analysis in order to study the effect of the parameters. Then, we experimentally compare the performance of the proposed methods against other methods [5].

[5]Source code is available at https://github.com/alihadian/ROLL

$\|V\|$	Items Traversed in SimpleRW	Uniform Samples in SA	Buckets Traversed in ROLL-bucket	Depth of Bucket in ROLL-tree
10^3	171	14	9.3	3.87
10^4	1,665	42	12.6	4.34
10^5	16,631	135	16.4	4.51
10^6	166,740	442	19.5	4.69
10^7	1,666,500	1,320	22.5	4.71
10^8	–	3,769	26.2	4.71
10^9	–	–	–	4.86

Table 6: Comparison of the *average* number of items, i.e. node (in SimpleRW & SA) or bucket (in ROLL-*), traversed for each random node selection ($m = 2$)

5.2.1 Experimental Setup

In this work, our main focus is on efficiency, since the stochastic properties of all compared methods are the same, i.e. they randomly select a random node according to its weight. Therefore, we only compare the execution times of the algorithms. The experiments were performed on a commodity machine with Intel Xeon 2.66 GHz processor with 32GB of available main memory, running Linux 3.5.0. Note that only one thread is used in our experiments because the algorithms are inherently sequential. All algorithms were developed with Java 1.8 SDK. The quantities were measured five times and the average is reported.

An important limitation of preferential attachment is that we need to store either the index or weight of all elements in the main memory, thus the space complexity of the algorithms are approximately $O(|V|)$. For example, if the node indexes are stored as 32-bit integers, in order to generate a one-billion-node graph, at least 4GB will be required to keep the roulette wheel in memory. Therefore, we only managed to generate graphs with up to 1 billion nodes in our commodity machine.

For all algorithms, the generated graphs are persisted as a stream of edge lists, i.e. each line contains two nodes "v_i, v_j" that represent an edge $E_{i,j}$ between v_i and v_j. The resulting output stream is then pipelined to an external consumer process. We did not include the time taken to persist the graph, because the graph might be presented in different formats (e.g. edge list, adjacency list, etc.), that is application-dependent, or it is even not persisted in many applications. For the m_0 parameter, we set $m_0 = m$ as its default value. To be noted, the value of m_0 does not affect the degree distribution, hence the performance.

5.2.2 Micro-level Analysis

The majority of the preferential attachment models follow the so-called "rich-get-richer" phenomenon, i.e. nodes that have been selected more in the past are more likely to be selected in the future. Such a model results in a long-tailed probability distribution over the roulette wheel items. The performance of each of the roulette wheel algorithms discussed before highly depends on the statistical properties of the probability distribution corresponding to the generation model. For example, the performance of SA depends on the average acceptance rate, i.e. how many times, on average, we need to select a random item until the selection is accepted, which in turn depends on the estimate of both the maximum and the average probability among roulette wheel items. Similarly, the performance of the ROLL-bucket depends on our estimate of $|B|$. Performance analysis of ROLL-tree is even harder because we have to know the probability distribution of the buckets and its growth model over time.

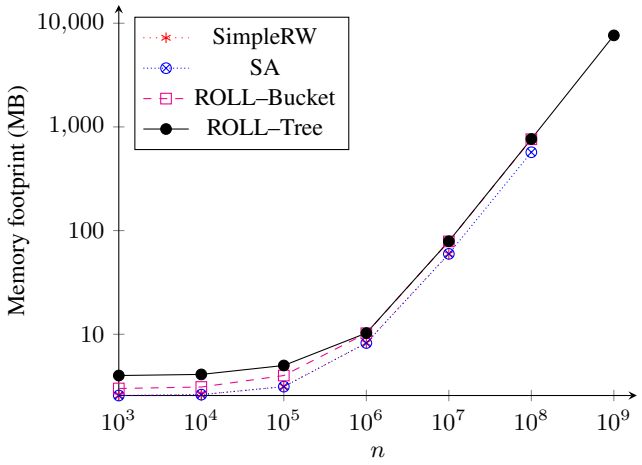

Figure 9: Memory footprints ($m = 2$)

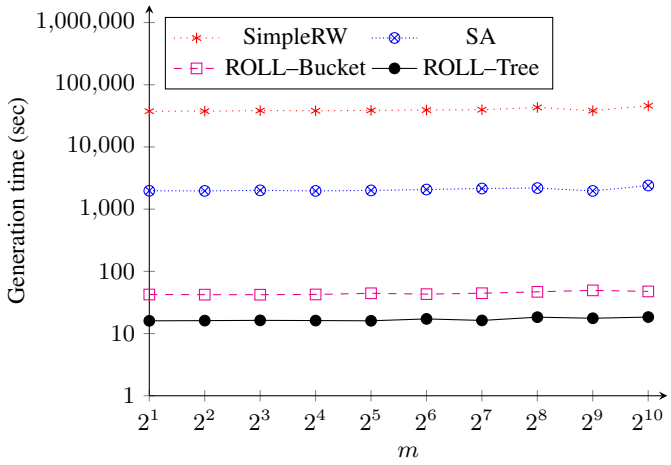

Figure 10: Generation time for various m_0, while $m = 2, n = 10^7$

In this section, we experimentally analyze how different roulette wheel algorithms perform for Barabási-Albert graph generation. More specifically, we focus on the parameters that affect the performance of the algorithms, such as the number of operations performed in each of the algorithms for selecting an item from the roulette wheel. Table 6 illustrates how many comparisons are performed by each of the algorithms to select an item from the roulette wheel. For example, to generate a Barabási-Albert graph with 10 million nodes using the simple roulette wheel algorithm, about 1.6 million items in the roulette wheel are traversed in average to sample each single item (target node) from the roulette wheel, where SA tries 1320 uniformly sampled items in average until a selection is accepted. Similarly, the average number of buckets that are traversed in ROLL-bucket and the average depth of the target bucket in ROLL-tree are 22.5 and 4.71, respectively.

In the following, we consider the memory footprints and execution times of the four algorithms discussed in the paper for generating Barabási-Albert graphs.

5.2.3 Memory Footprint Comparison

Figure 9 illustrates the memory footprints of the four algorithms. The memory consumption of SimpleRW and SA are almost equal, while ROLL-bucket and ROLL-tree consume about 30% more mem-

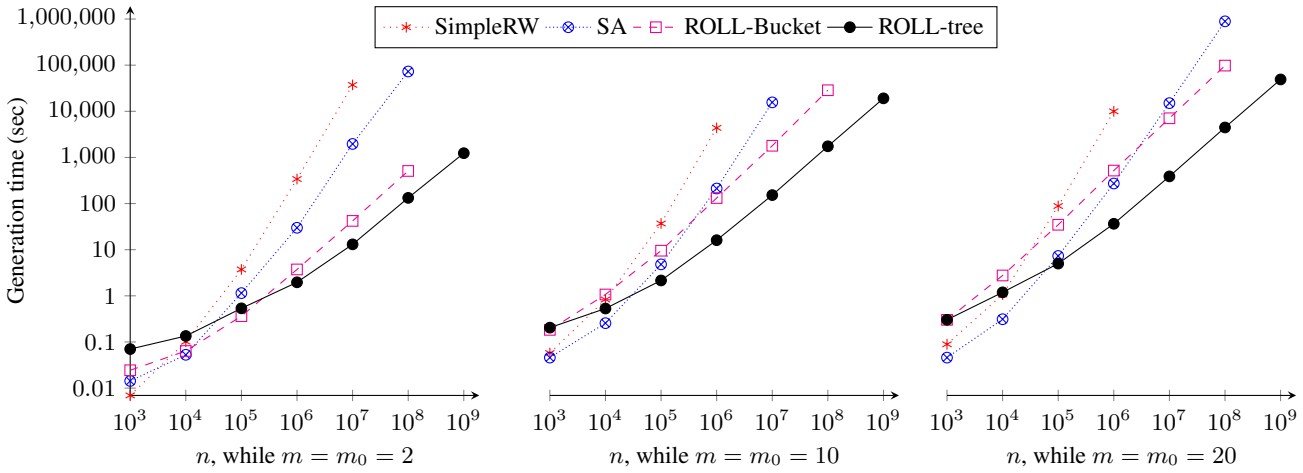

Figure 11: Generation time for various n values

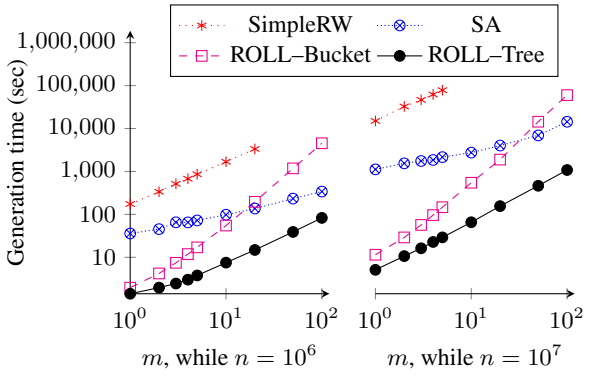

Figure 12: Generation time for various m values

ory space, which is mainly because the items inside each bucket are stored in dynamic arrays. These arrays reserve space for more entries than the actual number of items.

SimpleRW and SA only need a list of size n that contains the degree of items. Therefore, we use simple arrays for the sake of performance and cache efficiency. In ROLL-bucket, the list of buckets is stored in a red-black tree, which allows ordered traversal and is fast for lookup, delete, and insert. Items inside each bucket are stored in dynamic arrays that grow on demand.

5.2.4 *Performance Comparison*

The execution time for $m = 2$ and $m = 20$ is given in Figure 11. Results show that ROLL-tree outperforms all other algorithms with large margin for large graphs. For small-sized graphs, however, stochastic acceptance is faster than ROLL-tree, especially when m is relatively large. In general, ROLL-tree is the only algorithm that grows approximately linearly with n.

Figure 10 shows generation time of various m_0 values and further verifies that this time is almost independent of m_0. Figure 12 illustrates how graph generation time grows with m. Generally, the execution time of any roulette wheel algorithm is expected to grow linearly with m, since the roulette wheel is called $m(n-m_0)$ times and thus the number of roulette wheel invocations grows with m as well. The only exception is SA. As discussed in Section 3.2, the average number of uniform samples in SA until getting an acceptance is inversely proportional to m, i.e. $O(\sqrt{n}/m)$, which means that for larger m values, selecting from the roulette wheel is faster. However, since the stochastic rejection rate is generally too high, the stochastic acceptance algorithm is much slower than ROLL-bucket and ROLL-tree in real-world problems. Moreover, by increasing m after some threshold, that depends on n, the graph becomes so dense that it is no more scale-free. Thus, practically we are interested in generating BA graphs with m substantially smaller than n. For instance, there are hundreds of billions of nodes in the world wide web, while the average out-links (m) of each page is not more than 150. Figure 11 illustrates that when the graph is large ($n > 10^6$), ROLL-tree is the outperforming algorithm for any m.

6. CONCLUSION AND FUTURE WORK

The lack of gigantic scale-free graph input calls for scalable graph generation methods. This paper proposes an in-memory solution, ROLL-tree, a fast roulette wheel data structure to accelerate the Barabási-Albert Model. ROLL-tree is a fast, exact and generalizable method for preferential attachment processes. The experimental findings proved that ROLL-tree performs considerably faster than its rival. Instead of the baseline method that has a complexity of $O(|V|^2)$, the execution time of ROLL-tree is almost linear, which enables very large scale-free graph construction. For instance, on a single commodity machine we successfully generated a graph with 1 billion nodes in 62 minutes.

ROLL-tree is a general dynamic roulette wheel data structure, hence its not only applicable to random graph generation process but also to other random processes. For instance, many physical events can be modeled using random events that are similar to preferential attachment, e.g. simulating the motions of an object in water considering the diffusion effect in fluid. This study provides fundamentals and sheds a light in this direction. However, various applications may have specific constraints that requires further study in a generalized framework for future work.

ROLL-tree provides an exact serial roulette wheel data structure with no approximation. However, it can shed the light on the scalability requirements of generating gigantic graphs without losing precision. In other words, ROLL can be used as a building-block for parallel and distributed preferential attachment which effectively speeds up both the inter-machine and intra-machine operations by reducing the problem space from $O(n)$ to $O(|B|)$ and even very close to the lower bound of Huffman code word length over the bucket weights.

7. REFERENCES

[1] W. Aiello, F. Chung, and L. Lu. A random graph model for massive graphs. In *STOC*, pages 171–180, 2000.

[2] L. Akoglu and C. Faloutsos. RTG: A recursive realistic graph generator using random typing. *Data Mining and Knowledge Discovery*, 19(2):194–209, 2009.

[3] L. Akoglu, M. McGlohon, and C. Faloutsos. RTM: Laws and a recursive generator for weighted time-evolving graphs. In *ICDM*, pages 701–706, 2008.

[4] M. Alam, M. Khan, and M. V. Marathe. Distributed-memory parallel algorithms for generating massive scale-free networks using preferential attachment model. In *HPC*, pages 1–12, 2013.

[5] A. Arora, M. Sachan, and A. Bhattacharya. Mining statistically significant connected subgraphs in vertex labeled graphs. In *SIGMOD*, pages 1003–1014, 2014.

[6] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[7] S. Boccaletti, V. Latora, Y. Moreno, M. Chavez, and D.-U. Hwang. Complex networks: Structure and dynamics. *Physics reports*, 424(4):175–308, 2006.

[8] M. Boguná and R. Pastor-Satorras. Class of correlated random networks with hidden variables. *Physical Review E*, 68(3):036112, 2003.

[9] M. Boguná, R. Pastor-Satorras, and A. Vespignani. Cut-offs and finite size effects in scale-free networks. *The European Physical Journal B*, 38(2):205–209, 2004.

[10] P. Buesser and M. Tomassini. Supercooperation in evolutionary games on correlated weighted networks. *Physical Review E*, 85(1):16107–16107, 2012.

[11] T. G. K. C. Seshadhri and A. Pinar. Community structure and scale-free collections of Erdős-Rényi graphs. *Physical Review E*, 85(5):056109, 2012.

[12] G. Caldarelli, A. Capocci, P. De Los Rios, and M. A. Muñoz. Scale-free networks from varying vertex intrinsic fitness. *Physical review letters*, 89(25):258702, 2002.

[13] C. Campbell, K. Shea, and R. Albert. Comment on "control profiles of complex networks". *Science*, 346(6209):561, 2014.

[14] M. Capotă, T. Hegeman, A. Iosup, A. Prat-Pérez, O. Erling, and P. Boncz. Graphalytics: A big data benchmark for graph-processing platforms. In *GRADES*, 2015.

[15] J. M. Carlson and J. Doyle. Highly optimized tolerance: A mechanism for power laws in designed systems. *Physical Review E*, 60(2):1412, 1999.

[16] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *SDM*, pages 442–446, 2004.

[17] H.-H. Chen, Y.-B. Ciou, and S.-D. Lin. Information propagation game: A tool to acquire humanplaying data for multiplayer influence maximization on social networks. In *SIGKDD*, pages 1524–1527, 2012.

[18] F. Chung and L. Lu. The average distances in random graphs with given expected degrees. *PNAS*, 99(25):79–82, 2002.

[19] I. Csiszar and J. Körner. *Information theory: coding theorems for discrete memoryless systems*. Cambridge University Press, 2011.

[20] N. Du, H. Wang, and C. Faloutsos. Analysis of large multi-modal social networks: patterns and a generator. In *ECML-PKDD*, pages 393–408. 2010.

[21] P. Erdős and A. Rényi. On random graphs I. *Publicationes Mathematicae*, 6:290–297, 1959.

[22] N. Faller. An adaptive system for data compression. In *Asilomar Conference*, pages 593–597, 1973.

[23] I. Fazekas and B. Porvázsnyik. Scale-free property for degrees and weights in a preferential attachment random graph model. *Journal of Probability and Statistics*, 2013.

[24] R. Ferrer-Cancho and R. V. Solé. Optimization in complex networks. *Statistical Mechanics of Complex Networks*, 625:114–126, 2003.

[25] R. G. Gallager. *Information theory and reliable communication*. Wiley, 1968.

[26] E. N. Gilbert. Random graphs. *The Annals of Mathematical Statistics*, 30(4):1141–1144, 1959.

[27] A. Goldenberg, A. X. Zheng, S. E. Fienberg, and E. M. Airoldi. A survey of statistical network models. *Foundations and Trends in Machine Learning*, 2(2):129–233, 2010.

[28] K. Klemm and V. M. Eguiluz. Growing scale-free networks with small-world behavior. *Physical Review E*, 65(5):057102, 2002.

[29] D. E. Knuth. Dynamic huffman coding. *Journal of algorithms*, 6(2):163–180, 1985.

[30] T. G. Kolda, A. Pinar, T. Plantenga, and C. Seshadhri. A scalable generative graph model with community structure. *SIAM*, 36(5):424–452, 2014.

[31] C. León, C. Machado, and A. Murcia. Assessing systemic importance with a fuzzy logic inference system. *ISAFM*, 2015.

[32] J. Leskovec. *Dynamics of large networks*. PhD thesis, Carnegie Mellon University, 2008.

[33] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani. Kronecker graphs: An approach to modeling networks. *JMLR*, 11:985–1042, 2010.

[34] J. Li, H. Wang, S. U. Khan, Q. Li, and A. Y. Zomaya. A fully distributed scheme for discovery of semantic relationships. *IEEE Trans. on Services Computing*, 6(4):457–469, 2013.

[35] A. Lipowski and D. Lipowska. Roulette-wheel selection via stochastic acceptance. *Physica A*, 391(6):2193–2196, 2012.

[36] Y.-C. Lo, H.-C. Lai, C.-T. Li, and S.-D. Lin. Mining and generating large-scaled social networks via MapReduce. *Social Network Analysis and Mining*, 3(4):1449–1469, 2013.

[37] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[38] M. McGlohon, L. Akoglu, and C. Faloutsos. Weighted graphs and disconnected components: patterns and a generator. In *SIGKDD*, pages 524–532, 2008.

[39] M. E. Newman. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences*, 103(23):8577–8582, 2006.

[40] S. Nobari, X. Lu, P. Karras, and S. Bressan. Fast random graph generation. In *EDBT*, pages 331–342, 2011.

[41] S. Nobari, F. Tauheed, T. Heinis, P. Karras, S. Bressan, and A. Ailamaki. TOUCH: in-memory spatial join by hierarchical data-oriented partitioning. In *SIGMOD*, pages 701–712, 2013.

[42] M.-D. Pham, P. Boncz, and O. Erling. S3G2: A scalable structure-correlated social graph generator. In *TPCTC*, pages 156–172. 2013.

[43] J. Poncela, J. Gómez-Gardenes, Y. Moreno, and L. M. Floria. Cooperation in the prisoner's dilemma game in random scale-free graphs. *IJBC*, 20(03):849–857, 2010.

[44] R. V. Solé, R. Ferrer-Cancho, J. M. Montoya, and S. Valverde. Selection, tinkering, and emergence in complex networks. *Complexity*, 8(1):20–33, 2002.

[45] K. Soramäki and S. Cook. Sinkrank: An algorithm for identifying systemically important banks in payment systems. *Economics*, 7(28), 2013.

[46] S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD*, pages 845–856, 2007.

[47] S. Valverde, R. F. Cancho, and R. V. Sole. Scale-free networks from optimal design. *EuroPhysics Letters*, 60(4):512–517, 2002.

[48] J. S. Vitter. Design and analysis of dynamic huffman codes. *JACM*, 34(4):825–845, 1987.

[49] M. von Looz, C. L. Staudt, H. Meyerhenke, and R. Prutkin. Fast generation of dynamic complex networks with underlying hyperbolic geometry. *CoRR*, 2015.

[50] H. Wang, K. Zheng, X. Zhou, and S. W. Sadiq. SharkDB: An in-memory storage system for massive trajectory data. In *SIGMOD*, pages 1099–1104, 2015.

[51] D. J. Watts and S. H. Strogatz. Collective dynamics of small-world networks. *Nature*, 393(6684):440–442, 1998.

[52] X. Yang, J. Cao, and Z. Yang. Synchronization of coupled reaction-diffusion neural networks with time-varying delays via pinning-impulsive controller. *SIAM Journal on Control and Optimization*, 51(5):3486–3510, 2013.

[53] A. Yoo and K. W. Henderson. Parallel generation of massive scale-free graphs. *CoRR*, 2010.

[54] F. Yuan-Yuan, W. Liang, and Z. Shi-Qun. Synchronization of phase oscillators in networks with certain frequency sequence. *CTP*, 61(3):329–333, 2014.

[55] F. Zhu, Z. Zhang, and Q. Qu. A direct mining approach to efficient constrained graph pattern discovery. pages 821–832, 2013.