TrillionG: A Trillion-scale Synthetic Graph Generator using a Recursive Vector Model

Himchan Park
DGIST, Republic of Korea
chan150@dgist.ac.kr

Min-Soo Kim*
DGIST, Republic of Korea
mskim@dgist.ac.kr

ABSTRACT

As many applications encounter exponential growth in graph sizes, a fast and scalable graph generator has become more important than ever before due to lack of large-scale realistic graphs for evaluating the performance of graph processing methods. Although there have been proposed a number of methods to generate synthetic graphs, they are not very efficient in terms of space and time complexities, and so, cannot generate even trillion-scale graphs using a moderate size cluster of commodity machines. Here, we propose an efficient and scalable disk-based graph generator, *TrillionG* that can generate massive graphs in a short time only using a small amount of memory. It can generate a graph of a trillion edges following the RMAT or Kronecker models within two hours only using 10 PCs. We first generalize existing graph generation models to the *scope-based generation* model, where RMAT and Kronecker correspond to two extremes. Then, we propose a new graph generation model called the *recursive vector* model, which compromises two extremes, and so, solves the space and time complexity problems existing in RMAT and Kronecker. We also extend the recursive vector model so as to generate a semantically richer graph database. Through extensive experiments, we have demonstrated that TrillionG outperforms the state-of-the-art graph generators by up to orders of magnitude.

1. INTRODUCTION

Graphs are widely used to model real-world objects in many domains such as social networks, web, business intelligence, biology, and neuroscience, due to their simplicity and generality [37]. As many applications such as graph-based distributed OLTP query processing [17, 30, 32], the Internet of Thing (IoT) [7, 18], and the human connectome [6] encounter exponential growth in graph sizes, both fast and scalable graph processing methods and synthetic graph generation methods have become more important than ever before. In particular, there are two strong motivations for studying a fast and scalable graph generator: (1) lack of large-scale realistic graphs for evaluating the performance of graph processing methods; (2) lack of low-level core techniques that can be used for generating a benchmark database for rich graph model.

For the first motivation, a number of graph processing systems [15, 16, 25, 26, 33, 38] that can process a graph of one trillion edges have already been developed. All of them have used synthetic graphs for evaluating their performance since sharing and utilizing large-scale real-world graphs is very limited due to their being proprietary, or being practically impossible to collect. However, most trillion-scale synthetic graphs used so far are unrealistic synthetic ones that have a huge number of repeated edges and do not follow the power-law degree distribution [33, 38], which are considered as important properties of "realistic" graphs [35]. Thus, a scalable realistic synthetic graph generator is critical for more accurate evaluation of graph processing methods.

Unfortunately, the existing synthetic graph generators cannot generate a graph of trillion edges by using a cluster of commodity machines, or require a supercomputer for generation. There have been proposed a number of models or methods to generate realistic synthetic graphs [14, 27, 28], and RMAT and Kronecker are the ones most commonly used among them [29]. RMAT [14] is based on a recursive matrix model that recursively selects a quadrant on adjacency matrix to generate an edge and repeats the same procedure until a total of $|E|$ edges are generated for a graph $G = (V, E)$. Kronecker [27, 28] has two models for graph generation: Stochastic Kronecker Graph (SKG) and Deterministic Kronecker Graph (DKG). SKG is a generalized model of RMAT in terms of the number of probability parameters. Although both RMAT and Kronecker (i.e., SKG) are effective for generating synthetic graphs that are realistic, they are not very efficient for generating large-scale graphs in terms of space and time complexities. RMAT has the space complexity of $O(|E|)$, while Kronecker has the time complexity of $O(|V|^2)$. Thus, RMAT tends to have a limit on the size of the graph to generate due to its high memory requirement, and Kronecker tends to have a limit due to its high computational overhead. Meanwhile, there is a benchmark called Graph500 [4] to generate a trillion-scale graph and run a simple query on it for measuring the performance of supercomputers. This benchmark uses the SKG model of Kronecker for graph generation. A number of supercomputers can generate a trillion-scale graph through the benchmark. However, they do it using huge amount of computing resources, typically several thousand server computers connected via high speed network (e.g., Infiniband). To most researchers, it would be practically impossible to use such equipment for graph generation. Therefore, it is a challenging problem to generate a trillion-scale synthetic graph efficiently only using a small amount of computing resource.

For the second motivation, there are a lot of efforts to automatically generate a benchmark database for semantically rich graph

*Author to whom correspondence should be addressed.

SIGMOD '17, May 14–19, 2017, Chicago, IL, USA.

DOI: http://dx.doi.org/10.1145/3035918.3064014

models similar to the TPC benchmark for the relational model. With the proliferation of linked data in real world, managing graph-structured linked data becomes more and more important in many domains. There have been proposed a number of performance benchmarks for linked data management systems [9, 10, 20, 24]. One of representative methods, gMark [10] can generate a graph having various properties such as multiple node types and edge predicates. Linked Data Benchmark Council (LDBC) Social Network Benchmark (SNB) Datagen [20, 24, 31] is another representative method in social network benchmark. They have focused on rich semantics of a graph rather than the size of the graph so far. As a result, the sizes of resulting databases tend to be very small. However, both rich semantics and scalability are important for ideal benchmark. Thus, it is another challenging problem to develop scalability techniques that can be used for generating such a benchmark.

In this paper, we propose an efficient and scalable disk-based synthetic graph generator, *TrillionG* that can generate even a trillion-scale graph within two hours only using 10 PCs. We first generalize existing graph generation models such as RMAT and Kronecker to the *scope-based generation* model, where RMAT and Kronecker correspond to two extreme cases of the general model. Then, we propose a new graph generation model called the *recursive vector* model, which compromises two extremes, and so, solves the space and time complexity problems existing in RMAT and Kronecker. TrillionG follows the proposed recursive vector model and supports various kinds of graph formats such as TSV, Compressed Sparse Row (CSR), and Adjacency List (ADJ). As a result, TrillionG not only can generate larger graphs (i.e., scale-up), but also generate the same-sized graph faster (i.e., speed-up), compared with RMAT and Kronecker. In particular, TrillionG achieves such speed-up due to three key ideas of the recursive vector model: (1) reusing a pre-computed recursive vector; (2) reducing the number of quadrant selections; (3) reducing the number of random value generations. Through extensive experiments, we have demonstrated that TrillionG outperforms the state-of-the-art graph generators by up to orders of magnitude.

TrillionG follows a stochastic approach for edge generation like RMAT and Kronecker, and so, the resulting synthetic graph is realistic with respect to degree distribution. We also adopt the concept of random noise such that TrillionG generates more realistic graphs, as in [35]. In addition, for the above second motivation, we present how to extend TrillionG so as to generate the graphs supported by gMark [10] for rich graph generation. In detail, we propose the extended recursive vector (ERV) model and present the rich graph generation method using the ERV model. Experimental results show that extended TrillionG generates a bibliographical synthetic graph in gMark that has the correct in-/out-degree distributions.

The rest of the paper is organized as follows. Section 2 describes RMAT and Kronecker. In Section 3, we generalize a graph generation model to the scope-based model and analyze the complexities of various models. Section 4 proposes our recursive vector model, and Section 5 presents the TrillionG system based on the proposed model. Section 6 extends the recursive vector model for rich graph generation. Section 7 presents the results of experimental evaluation, and Section 8 discusses related work. Finally, Section 9 summarizes and concludes this paper.

2. PRELIMINARIES

2.1 RMAT

RMAT [14] stands for Recursive MATrix (RMAT) graph generator and may be the most commonly used model to generate a realistic synthetic scale-free graphs due to its simple and fast mechanism. The basic idea behind RMAT is recursive quadrant selection on adjacency matrix for an edge generation, and RMAT repeats such an edge generation until a whole graph is generated. Conceptually, it partitions adjacency matrix into four quadrants, which have probability parameters α, β, γ, and δ, as in Figure 1(a). A higher parameter value indicates a higher probability for the corresponding quadrant to be selected, and the sum of four parameters should be 1.0. For a graph $G = (V, E)$, the size of adjacency matrix is $|V| \times |V|$, where a row means a source vertex, and a column means a destination vertex. Each quadrant is recursively partitioned into four sub-quadrants of the same probability parameters α, β, γ, and δ until the size of a sub-quadrant becomes 1×1. Therefore, the number of recursive selection steps required for generating an edge is $log|V|$. Figure 1(b) shows an example of generation of an edge in RMAT, where the steps of recursive guardant selection are $\beta \rightarrow \delta \rightarrow \gamma \rightarrow \beta$ ($|V| = 16$). When the recursive quadrant selection reaches the 1×1 *cell* of adjacent matrix, for example, (x, y) cell, RMAT appends an edge (x, y) to the resulting graph.

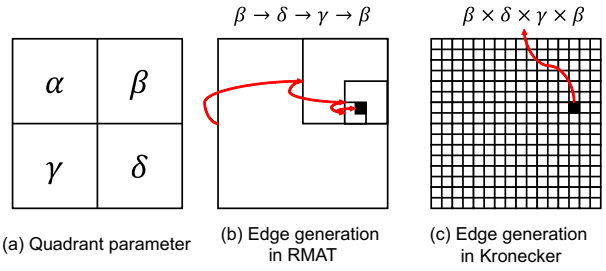


Figure 1: Graph generation models. (a) Four quadrant parameters. (b) Example of RMAT. (c) Example of Kronecker.

2.2 Kronecker

SKG [27, 28] stands for Stochastic Kronecker Graph generator and is a generalized model of RMAT. While RMAT considers only 2×2 probability parameters, SKG considers $n \times n$ probability parameters. That is RMAT is a special case of SKG ($n = 2$). Both RMAT and SKG are similar to each other in terms of using probability matrix, but they are dissimilar to each other in terms of the stochastic process. While RMAT generates an edge via a series of recursive steps in a *dynamic* manner, SKG does it in a *static* manner. In detail, SKG checks every cell of the adjacency matrix whether the corresponding edge is generated or not with respect to the probability of the cell. Figure 1(c) shows an example of generation of an edge in SKG, where the black cell has the probability of $\beta \times \delta \times \gamma \times \beta$ for generation. When a randomly generated value for the cell is within the probability, the corresponding edge is generated.

The entire probability matrix of $|V| \times |V|$ of SKG can be formally presented using Kronecker product, which is an operator to calculate an outer product between given two arbitrary-size matrices.

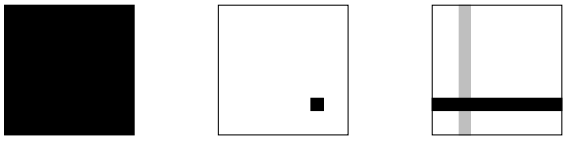
Definition 1. **Kronecker product**: Given two matrices A and B, where A is of size $n \times m$, and B is of size $p \times q$, the Kronecker product between A and B is defined as $\begin{bmatrix} a_{1,1}B & \cdots & a_{1,m}B \\ \vdots & \ddots & \vdots \\ a_{n,1}B & \cdots & a_{n,m}B \end{bmatrix}$, and in more detail,

$$A \otimes B = \begin{bmatrix} a_{1,1}b_{1,1} & \cdots & a_{1,1}b_{1,q} & \cdots & a_{1,m}b_{1,1} & \cdots & a_{1,m}b_{1,q} \\ \vdots & \ddots & \vdots & & \vdots & \ddots & \vdots \\ a_{1,1}b_{p,1} & \cdots & a_{1,1}b_{p,q} & & a_{1,m}b_{p,1} & \cdots & a_{1,m}b_{p,q} \\ & \vdots & & \ddots & & \vdots & \\ a_{n,1}b_{1,1} & \cdots & a_{n,1}b_{1,q} & & a_{n,m}b_{1,1} & \cdots & a_{n,m}b_{1,q} \\ \vdots & \ddots & \vdots & \cdots & \vdots & \ddots & \vdots \\ a_{n,1}b_{p,1} & \cdots & a_{n,1}b_{p,q} & & a_{n,m}b_{p,1} & \cdots & a_{n,m}b_{p,q} \end{bmatrix}.$$

By using Definition 1, the probability matrix of SKG $\mathbb{K}$ can be presented as $\mathbb{K} = K \bigotimes K \bigotimes K \cdots \bigotimes K = K^{\bigotimes \log_n |V|}$, where K is a given seed probability $n \times n$ matrix.

3. SCOPE-BASED GENERATION MODEL

In this section, we propose the general model for synthetic graph generation. There is no fundamental difference between RMAT and Kronecker (especially, SKG) in terms of the probability model, and so, their resulting graphs have the same graph properties including degree distribution as we show in Section 7. However, their computational properties are quite different with each other. RMAT has a high space complexity, whereas Kronecker has a high time complexity. We have identified that the difference in computational overhead is mainly due to the difference in *scope* of edge generation. When generating an edge, RMAT considers a whole adjacency matrix as a scope for generating non-repeated edges, and the number of generation for a graph is $|E|$. On the contrary, Kronecker considers only an individual cell of adjacency matrix as a scope, and the number of generation for a graph is $|V| \times |V|$. Hereafter, we call the former approach as *Whole Edges Scope* (WES) and the latter approach as *An Edge Scope* (AES). In this paper, we propose a recursive vertex model in Section 4, which considers a row (or a column) of adjacency matrix as a scope. Thus, we call it as *A Vertex Scope* (AVS). Figure 2 shows the scopes of three generation models, WES, AES, and AVS.



(a) RMAT-WES (b) Kronecker-AES (c) TrillionG-AVS

Figure 2: The scopes of the generation models.

From the above observation, we present a generalized model for WES, AES, and AVS, called the *scope-based generation* model. This model generates a synthetic graph in two stages: scope generation and edge generation. Algorithm 1 shows the scope generation stage. With a given size of a scope $N \times M$, Lines 1-2 generate a set of ranges for scopes. We assume that $|V|$ can be divided by either of two integers, N and M, without a remainder. $rangeSetI$ is a set of ranges along the vertical axis, and $rangeSetJ$ is a set of ranges along the horizontal axis. Line 3 generates a grid of scopes according to two sets of ranges, and Line 5 calls the ScopeGeneration function for each scope in parallel. In RMAT, $N = M = |V|$, and, in Kronecker, $N = M = 1$.

Algorithm 2 shows the ScopeGeneration function which generates all edges within a given scope. It just calls the EdgeGeneration function on the given scope until $|\mathbb{S}(I, J)|$ edges are generated. Here, $|\cdot|$ is a special function that calculates the proper number of edges within a given scope in the stochastic manner. In RMAT, $|\mathbb{S}(I, J)|$ returns $|E|$ since there is only one scope. In Kronecker, it returns 0 or 1 depending on the probability of the corresponding scope (cell). The $count(edgeSet)$ function simply returns the number of elements in $edgeSet$. Finally, the algorithm stores the set of edges in main memory to secondary storage (Line 5). Without loss of generality, we assume that $edgeSet$ for a scope resides in main memory during scope generation, in order to check duplication of an edge and avoid repeat edges during set union (Line 4).

Algorithm 1 Scope-based Generation Model

Input: $|V|$, /* number of vertices */
$N \times M$ /* size of a scope */

1: $rangeSetI \leftarrow \{[kN, (k+1)N - 1] \mid 0 \leq k < \frac{|V|}{N}\}$
2: $rangeSetJ \leftarrow \{[lM, (l+1)M - 1] \mid 0 \leq l < \frac{|V|}{M}\}$
3: $grid \leftarrow \{\mathbb{S}(I, J) \mid I \in rangeSetI, J \in rangeSetJ\}$
4: **parallel for each** $\mathbb{S}(I, J)$ **in** $grid$ **do**
5: `ScopeGeneration`$(\mathbb{S}(I, J))$

Algorithm 2 Scope Generation

Input: $\mathbb{S}(I, J)$ /* scope for edge generation */

1: $edgeSet \leftarrow \emptyset$
2: **while** $count(edgeSet) \leq |\mathbb{S}(I, J)|$ **do**
3: $newEdge \leftarrow$ `EdgeGeneration`$(\mathbb{S}(I, J))$
4: $edgeSet \leftarrow edgeSet \cup \{newEdge\}$
5: `Store`$(edgeSet)$

According to Algorithm 2, the WES approach has a limit on parallel processing since it has only a single scope. The scope-based generation model assumes that each scope is processed independently. If the ScopeGeneration function is concurrently executed on the same scope for better performance in the WES approach, we need to resolve duplicates of the same edge across different memory spaces on the same scope. For instance, we should eliminate duplicates through shuffle and merge in a distributed system, which will be presented in Section 3.2.

3.1 Complexities of Models

We present the space and time complexities for the scope-based generation model. In terms of the time complexity, the cost of generating an entire graph, T_{graph}, can be calculated as in Eq. (1), where T_{scope} is the cost of the ScopeGeneration function. T_{scope} again can be calculated as in Eq. (2), where T_{edge} is the cost of the EdgeGeneration function.

$$T_{graph} = \sum_{S \in grid} T_{scope}(\mathbb{S}) \quad (1)$$

$$T_{scope}(\mathbb{S}) = |\mathbb{S}| \times T_{edge} \quad (2)$$

In the WES approach, $|\mathbb{S}|$ is equal to $|E|$, and T_{edge} is $O(\log |V|)$ due to recursive quadrant selection. Therefore, the time complexity of WES becomes $O(|E| \log |V|)$. In the AES approach, the number of scopes is $|V|^2$, $|\mathbb{S}| = 1$, and T_{edge} is $O(1)$. Therefore, the original time complexity of AES becomes $O(|V|^2)$. Since AES can be easily parallelized, its time complexity in a distributed system having P cores becomes $O(\frac{|V|^2}{P})$.

In terms of the space complexity, the amount of memory required in a machine becomes equal to the size of the scope, i.e., $|\mathbb{S}(I, J)|$, to check duplication of an edge. The WES approach requires the space of $O(|E|)$, and the AES approach requires that of $O(1)$. That means WES will fail to generate a graph if the whole set of edges cannot fit in main memory.

In general, both real graphs and scale-free synthetic graphs are very sparse; they have a much smaller number of edges $|E|$ compared with the number of cells of the adjacency matrix $|V|^2$. Therefore, the original AES approach is much slower than the WES approach in most cases. In order to overcome this problem, FastKronecker has been proposed [28], which is embedded in SNAP [8]. Although the name is FastKronecker, it generates a graph in RMAT-like manner. In detail, FastKronecker performs a recursive region selection using a given $n \times n$ seed matrix for generating an edge, where the number of recursions is $\log_n |V|$. It performs such a series of selections $|E|$ times for a graph generation. During graph generation, it needs to keep all edges in memory to eliminate duplicates. When $n = 2$, FastKronecker becomes equal to RMAT of the WES approach. Thus, its time and space complexities become $O(|E| \log |V|)$ and $O(|E|)$, respectively [8].

3.2 Merge-based Approach

In this section, we present the variation of the WES approach, called WES/p, that executes the ScopeGeneration function in parallel and then merges the edges shuffled in parallel. WES/p is not our own proposed method, but a simple parallel version of WES that can be easily devised starting from [5]. The basic idea of WES/p is that it performs same scope generation of WES in parallel, and then, after the generation, it eliminates the duplicated edge by shuffling and merging all generated edges. Algorithm 3 presents the pseudo code of WES/p. Lines 2-6 are very similar to Algorithm 2, except the size of scope. Since Lines 2-6 are executed on the same scope, we need to set the size of scope to $\frac{|E|}{P} \cdot (1 + \epsilon)$ instead of $\frac{|E|}{P}$ with considering duplicate elimination ($\epsilon > 0$). It is difficult to know an exact ϵ value in advance, and the proper ϵ value tends to decrease as $|E|$ increases. Since ϵ is known to be converged to a smaller number than 0.01 [28, 35], we also set $\epsilon = 0.01$ in our experiments for WES/p. Line 7 shuffles local $edgeSet$ across network. Lines 8-9 merge incoming $edgeSet'$ while eliminating duplicates, which stop when the number of edges becomes $\frac{|E|}{P}$. This WES/p approach can decrease time complexities by up to P times, compared with the original WES approach.

We let WES/p-mem be the in-memory version of WES/p and WES/p-disk be the disk-based version of WES/p. The major difference between both is the way of duplicate elimination. WES/p-mem does it using memory, while WES/p-disk does it using the external sort. Thus, WES/p-disk can generate much large graphs than WES/p-mem. In the case of WES/p-mem, we need P times more machines for generating a P times bigger graph. Thus, WES/p-mem is still not a good solution to generate a several orders of magnitude bigger synthetic graph.

Algorithm 3 Merge-based Approach (WES/p)

Input: $\mathbb{S}(V, V)$, /* whole adjacency matrix */
P /* number of cores */

1: **parallel for** p in $[0, P-1]$ **do**
2: $edgeSet \leftarrow \emptyset$
3: **while** $count(edgeSet) \leq \frac{|E|}{P} \cdot (1 + \epsilon)$ **do**
4: $newEdge \leftarrow$ `EdgeGeneration`$(\mathbb{S}(V, V))$
5: $edgeSet \leftarrow edgeSet \cup \{newEdge\}$
6: `Store`$(edgeSet)$
7: `Shuffle`$(edgeSet)$ /* shuffle partial $edgeSet$ */
8: **parallel for** p in $[0, P-1]$ **do**
9: `Merge`$(edgeSet')$ /* eliminate duplicates */

Furthermore, there are two additional serious overheads in WES/p: (1) shuffling overhead $T_{shuffle}$ and (2) merging overhead T_{merge}. Both overheads can never be disregarded. $T_{shuffle}$ would be the transfer time of the whole set of edges among machines, and T_{merge} would be at least $O(|edgeSet'| \log |edgeSet'|)$, which entails external sort. As P increases, the sizes of $edgeSet'$ become more skewed in general, which would further degrade the overall performance.

3.3 AVS Approach

The above sections show that both WES and AES have some issues in complexities due to either too coarse or too fine granularity of their scopes. Here, we propose the A Vertex Scope (AVS) approach having the scopes of "medium" granularity. In Figure 2(c), the black area indicates AVS based on out-edge (shortly, AVS-O), and the gray area indicates AVS based on in-edge (shortly, AVS-I). In the AVS approach, a scope is set to either $1 \times |V|$ (AVS-O) or $|V| \times 1$ (AVS-I). The size of $grid$, i.e., the number of scopes becomes $|V|$. As a result, the area of a scope is much smaller than that of WES, and at the same time, the number of scopes is also much smaller than that of AES.

Here we analyze the complexities of the AVS approach in detail. We denote a scope of AVS-O as $\mathbb{S}(i, V)$ and that of AVS-I as $\mathbb{S}(V, i)$ $(\forall i \in V)$. We also denote the size of a scope of AVS-O as $|\mathbb{S}(i, V)| = d_i^+$ and that of AVS-I as $|\mathbb{S}(V, i)| = d_i^-$, where d_i^+ is the out-degree of a vertex i, and d_i^- is the in-degree of vertex i. Then, the time complexities of AVS become Eqs. (3)-(5), where d_{max} is the maximum out/in-degree of a vertex. In Eq. (5), our recursive vector model, which will be presented in Section 4, generates an edge recursively in a range of $[0, |V| - 1]$, and so, T_{edge} becomes $\log |V|$ similarly with the RMAT model. We omit the details for the calculation of T_{edge}, and describe it in Section 4.2. Hereafter, we only present the case of AVS-O, since the analysis for AVS-I is similar to that of AVS-O. The space complexity of AVS becomes Eq. (6).

$$T^+_{graphAVS} = \sum_{i \in V} \{d_i^+ \times T_{edge}\}. \tag{3}$$

$$T^-_{graphAVS} = \sum_{i \in V} \{d_i^- \times T_{edge}\}. \tag{4}$$

$$T_{graphAVS} = \sum_{i \in V} d_i \times \log |V| \leq |E| \log |V| \tag{5}$$

$$S^+_{graphAVS} = \max_{i \in V} d_i^+ = d^+_{max} \tag{6}$$

Table 1 shows the summary of the time and space complexities of four approaches under the scope-based generation model. In the table, AES means the original Kronecker method. Table 1 indicates the AVS approach is theoretically the best one for generation of a large-scale graph, since d_{max} is much smaller than $|E|$, and there is no $T_{shuffle}$ and T_{merge}.

4. RECURSIVE VECTOR MODEL

In this section, we present the recursive vector model following the AVS approach. The recursive vector model can significantly reduce memory usage, and at the same time, improve the speed of generation of edges practically, compared with a naive method that does not utilize the recursive vector model. Its basic idea is (1) constructing a very small and precomputed vector called $RecVec$ for each scope that can follow the RMAT-like stochastic process, and (2) generating edges very efficiently by repeatedly searching $RecVec$ on CPU cache.

Table 1: Summary of the time and space complexities.

	Time complexity	Space complexity
WES	$O(\|E\| \log \|V\|)$	$O(\|E\|)$
AES	$O(\frac{\|V\|^2}{P})$	$O(1)$
FastKronecker	$O(\|E\| \log \|V\|)$	$O(\|E\|)$
WES/p	$O(\frac{\|E\| \log \|V\|}{P}) + T_{shuffle} + T_{merge}$	$O(\frac{\|E\|}{P})$
AVS	$O(\frac{\|E\| \log \|V\|}{P})$	$O(d_{max})$

Using the recursive vector model, we can determine out-degree and in-degree distributions efficiently and independently. We present the determination of out-degree distributions (i.e., sizes of scopes in the AVS approach) in Section 4.1, and the determination of in-degree distributions (i.e., generation of edges using $RecVec$ in each scope) in Section 4.2. The former is summarized in Theorem 1, and the latter is summarized in Theorem 2. In particular, Theorem 2 is supported by two Lemmas: Lemma 3 about the property of scale symmetry existing in $RecVec$; and Lemma 4 about the property of translational symmetry existing in $RecVec$. Both the sizes of scopes and the edges in a scope should be determined through a stochastic process as in RMAT and Kronecker. Otherwise, the generator would produce an unrealistic graph in terms of graph properties. Finally, Section 4.3 presents three key ideas of the recursive vector model in terms of performance and analyzes them.

4.1 Determination of Scope Size

We denote a whole Kronecker matrix produced by a seed matrix as $\mathbb{K}$, where each element $\mathbb{K}_{u,v}$ means the probability of generating an edge from u to v for a target synthetic graph. Figure 3(b) shows an example of Kronecker matrix $\mathbb{K} = K^{\otimes 3}$ from an initial seed matrix K in Figure 3(a). In this section, we focus on the case of SKG using 2×2 seed matrix $K = \begin{bmatrix} K_{0,0} & K_{0,1} \\ K_{1,0} & K_{1,1} \end{bmatrix} = \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix}$. That is the same with the case of RMAT. Then, we can present $\mathbb{K} = K^{\bigotimes \log |V|}$ in general.

u \ v	0	1
0	$\alpha = .5$	$\beta = .2$
1	$\gamma = .2$	$\delta = .1$

(a) Seed matrix K

u \ v	0	1	2	3	4	5	6	7
0	.125	.05	.05	.02	.05	.02	.02	.008
1	.05	.025	.02	.01	.02	.01	.008	.004
2	.05	.02	.025	.01	.02	.008	.01	.004
3	.02	.01	.01	.005	.008	.004	.004	.002
4	.05	.02	.02	.008	.025	.01	.01	.004
5	.02	.01	.008	.004	.01	.005	.004	.002
6	.02	.008	.01	.004	.01	.004	.005	.002
7	.008	.004	.004	.002	.004	.002	.002	.001

(b) Kronecker matrix $K^{\otimes 3}$

Figure 3: An example of Kronecker matrix.

Each $\mathbb{K}_{u,v}$ is calculated by a series of products among α, β, γ and δ. We denote such series of products as $P_{u \to v}$. Given a seed matrix K, we define $|\alpha|_{u,v}$, $|\beta|_{u,v}$, $|\gamma|_{u,v}$, and $|\delta|_{u,v}$, as the multiplicities of α, β, γ, and δ, in $P_{u \to v}$, respectively, where $|\alpha|_{u,v} + |\beta|_{u,v} + |\gamma|_{u,v} + |\delta|_{u,v} = \log |V|$. Then, $P_{u \to v}$ is equal to $\alpha^{|\alpha|_{u,v}} \beta^{|\beta|_{u,v}} \gamma^{|\gamma|_{u,v}} \delta^{|\delta|_{u,v}}$.

To exploit bitwise operations, we consider a vertex ID as a binary string. If the most significant bit (MSB) of a source vertex u is 0, it means that u exists in α or β quadrant. Likewise, if the MSB of a destination vertex v is 0, it means that v exists in α or γ quadrant. We define a logical bit operator `Bits(x)` which returns the number of 1 bits in a binary string x. Then, Proposition 1 [28] follows, where $\&$ denotes the bitwise AND operator, and $\sim$ denotes the bitwise NOT operator.

Proposition 1. [28] **Probability of an edge from u to v.**
The probability of generating an edge from u to v can be calculated using the bit operator `Bits(·)` as below when representing vertex IDs u and v as binary strings.

$$\mathbb{K}_{u,v} = \alpha^{\texttt{Bits}(\sim u \& \sim v)} \beta^{\texttt{Bits}(\sim u \& v)} \gamma^{\texttt{Bits}(u \& \sim v)} \delta^{\texttt{Bits}(u \& v)}$$

Proposition 1 means that two vertices (in bit strings) of an edge can determine the number of quadrant selections for each quadrant. By using Proposition 1, we can calculate the probability of generating an edge from u to v in constant time when using a fixed-size type for vertex IDs. Now, we draw Lemma 1 by neglecting a destination vertex v for a certain source vertex u.

Lemma 1. **Probability of an edge from u.**
The probability of generating edges from u can be calculated as below.

$$P_{u \to} = (\alpha + \beta)^{\texttt{Bits}(\sim u)} \cdot (\gamma + \delta)^{\texttt{Bits}(u)}$$

PROOF. Please see Appendix A. □

Next, we define that the size of the scope of $\mathbb{S}(i, V)$, which should be the sum of all the edges to be generated in the scope. The existence of an edge has the Bernoulli distribution, and multiple trials for the edges in a scope has the binomial distribution. By the Central Limit Theorem, the binomial distribution with a large number of trials can be approximated by the normal distribution, where the size of a scope $\mathbb{S}(i, V)$, i.e., the degree of a vertex i,d_i^+, can be summarized as in Theorem 1. We note that $|\mathbb{S}(I, J)|$ in Algorithm 2 in our model is not deterministic, but stochastic following Theorem 1.

Theorem 1. **Size of a scope $\mathbb{S}(i, V)$.**
We denote that $n = |E|$ and $p = \sum_{j \in V} P_{i \to j} = P_{i \to}$. Then, the size of a scope $\mathbb{S}(i, V)$ has approximately a normal distribution with $\mu = np$ and $\sigma = \sqrt{np(1-p)}$.

PROOF. Please see Appendix B. □

4.2 Determination of Edge

For a source vertex u, we generate $|\mathbb{S}(u, V)|$ edges according to Theorem 1. The naive method that can determine a destination vertex of each edge is using the inverse function $F_u^{-1}()$ of a cumulative density function (CDF) of a source vertex u, $F_u()$. Figure 4(a) shows a plot of a probability mass function of a source vertex u for a small synthetic graph generated with the parameters $K = \begin{bmatrix} 0.57 & 0.19 \\ 0.19 & 0.05 \end{bmatrix}$ and $|V| = 2^{10}$. Figure 4(b) shows the corresponding CDF function $F_u()$. Here, we assume that $2R = |V|$, and $F_u(r) = c$. Then, the naive method generates a real value x using a uniform random variable $\mathbb{U}(0, F_u(|V|))$, and finds a destination vertex of u by calculating $F_u^{-1}(x)$. For instance, in Figure 4(b), we assume a real value c is generated from $\mathbb{U}(0, F_u(|V|))$, and then,

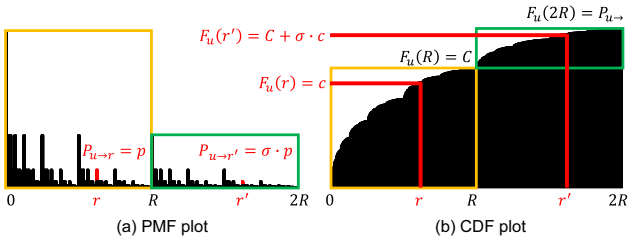


Figure 4: Examples of PMF and CDF plots for a vertex u.

a destination vertex becomes r since $F_u^{-1}(c) = r$. The probability that a destination vertex r ($0 \leq r < 2R$) is chosen follows the distribution in Figure 4(a).

The naive method described above stores all the CDF values for u into a vector (CDF vector in short) and finds a destination vertex $F_u^{-1}(x)$ for a given random value x by doing either linear search or binary search on the CDF vector. The first row in Table 2 shows the time and space complexities of this naive method. The time complexity using binary search, i.e., $O(\log |V|)$ for determination of a single edge is the same with that of RMAT. The problem of the naive method is the space complexity. For a trillion-scale graph of 2^{36} vertices, the amount of memory required is about 274 GB when using 4 byte single precision type for representing CDF values. It is definitely impossible to use the naive method.

Thus, we propose an improved method using a recursive vector called $RecVec$. The theoretical time and space complexities of our recursive vector model are $O(\log |V| \cdot \log \log |V|)$ and $O(\log |V|)$, respectively. Here, the term $\log \log |V|$ in the time complexity is due to a binary search on a vector of a length $\log |V|$, which will be explained in detail later. However, we can disregard the term practically due to the following two reasons. First, $\log \log |V|$ is a small and almost constant value. For example, a trillion-scale graph, $\log \log |V|$ is only about 5.16, and even a quadrillion-scale graph is only about 5.52. Second, $RecVec$ is on CPU cache, and so, the cost of a binary search on $RecVec$ is practically negligible, which will be explained in Section 4.3. As a result, the practical time complexity becomes as in the second row in Table 2. The improved method significantly reduces the amount of memory required, and so, allows us to generate a trillion-scale graph using a cluster of PCs. For example, a trillion-scale graph, the amount of memory for $RecVec$ is only about $36 \times 8 = 288$ bytes. Although the size of data structure for generating edges is so small, the set of edges that are already generated in a scope should be maintained for checking repeat edges. That space is $O(d_{max})$, and so, the overall space complexity becomes as in Table 1.

Table 2: Summary of complexities of the naive method using CDF vector and our recursive vector model using $RecVec$.

Data Structure	Search Method	Time complexity	Space complexity
CDF vector	linear	$O(\|V\|)$	$O(\|V\|)$
	binary	$O(\log \|V\|)$	$O(\|V\|)$
$RecVec$	binary	$O(\log \|V\|)$	$O(\log \|V\|)$

Our recursive vector model constructs $RecVec$ of length $\log |V|$+1, and generates $|\mathbb{S}(u, V)|$ edges, each of which takes $O(\log |V| \cdot \log \log |V|)$ time. Intuitively, $RecVec$ stores the values at the positions of power of 2 in the CDF vector, i.e., the values at $0, 1, 3, 7, \ldots, 2^{\log |V|} - 1$, and so, its length becomes $\log |V| + 1$. We define $RecVec$ for a source vertex u in Definition 2. For simplicity, we assume the CDF function $F_u(\cdot)$ has the domain between 1 to $|V|$, instead of between 0 to $|V| - 1$, and the summation starts from the vertex 0. For example, $F_u(6)$ is the summation of probabilities between $P_{u\to 0}$ and $P_{u\to 5}$.

Definition 2. **Recursive vector $RecVec$ of a vertex u.** For a recursive vector $RecVec[0 : \log |V|]$ for a source vertex u, each element of $RecVec[x]$ ($0 \leq x \leq \log |V|$) is defined as follows.

$$RecVec[x] = F_u(2^x) = \sum_{v=0}^{2^x - 1} P_{u\to v}$$

If we use a summation of $P_{u\to v}$ when constructing $RecVec$, it would take $O(|V|)$ time. In order to construct $RecVec$ in $O(\log |V|)$ time, we use Lemma 2 by exploiting the characteristics of stochastic process based on the seed probability parameters.

Lemma 2. **CDF value at the positions of power of 2.** We assume that $x \in [0 : \log |V|]$, and $\gg$ is the logical right shift, then the value of an element of $RecVec$ is defined as follow.

$$RecVec[x] = \left(\frac{\alpha}{\alpha+\beta}\right)^{\log |V| - x - \mathrm{Bits}(u \gg x)} \cdot \left(\frac{\gamma}{\gamma+\delta}\right)^{\mathrm{Bits}(u \gg x)} \cdot P_{u\to}$$

PROOF. Please see Appendix A. □

We explain an example of Lemma 2 using Figure 3. For a source vertex 2, the corresponding $RecVec$ is $[0.05, 0.07, 0.105, 0.147]$. For $x = 2$, $RecVec[2] = F_2(2^2) = 0.05 + 0.02 + 0.025 + 0.01 = 0.105$ by Definition 2. If we use Lemma 2 for calculating the same element, $RecVec[2] = \left(\frac{0.5}{0.7}\right)^{3-2-\mathrm{Bits}(010_2 \gg 2)} \cdot \left(\frac{0.2}{0.3}\right)^{\mathrm{Bits}(010_2 \gg 2)} \cdot P_{2\to} = \left(\frac{0.5}{0.7}\right)^1 \cdot \left(\frac{0.2}{0.3}\right)^0 \cdot 0.147 = 0.105$. Here, 010_2 is a binary string for the value 2, and $P_{2\to} = .147$.

Now, we explain the method to determine a destination vertex based on $RecVec$ in $O(\log |V| \cdot \log \log |V|)$ time. For doing that, we first present two properties for recursive selection on $RecVec$: scale symmetry and translational symmetry. Lemma 3 presents the property of *scale symmetry*, which intuitively means there is a constant ratio σ between the probability of determining a vertex r and that of determining a vertex r'. Here, we assume that $r' = r + R$, and $0 \leq r < R$.

Lemma 3. **Property of scale symmetry** For each $R = 2^k$ ($0 \leq k < \log |V|$), a constant ratio $\sigma_{u[k]}$ is maintained for $\forall r : 0 \leq r < R$, in the PMF distribution for a source vertex u. We assume $u[k]$ indicates the k-th bit of a vertex u in binary string starting from the right most LSB.

$$\sigma_{u[k]} = \frac{P_{u\to(R+r)}}{P_{u\to r}} = \frac{K_{u[k],1}}{K_{u[k],0}}$$

PROOF. Please see Appendix A. □

In Lemma 3, $K_{u[k],1}$ means the probability parameter of either upper right quadrant or lower right quadrant depending on the bit value of $u[k]$. We explain an example of Lemma 3 using Figure 3. For a source vertex $u = 2$, we first check the constant ratio $\sigma_{u[k]}$ for the positive integer $k = \log |V| - 1 = 2$, i.e., $R = 2^k = 4$. Since $2[2] = 010_2[2] = 0$, i.e., the MSB of 010_2, $\sigma_{2[2]} = \frac{P_{2\to(4+r)}}{P_{2\to r}} = \frac{K_{0,1}}{K_{0,0}} = \frac{0.2}{0.5}$, according to the seed matrix Figure 3(a). This ratio is maintained while varying r $s.t.$ $0 \leq r < 4$. For example, for $r = 1$, $\frac{P_{2\to 5}}{P_{2\to 1}} = \frac{0.008}{0.02} = \frac{0.2}{0.5}$. If k is changed, then the ratio

can be changed. For example, for $k = 1$ and $R = 2$, $\sigma_{2[1]} = \frac{P_{2\to(2+r)}}{P_{2\to r}} = \frac{K_{1,1}}{K_{1,0}} = \frac{0.1}{0.2}$, since $2[1] = 010_2[1] = 1$. In fact, for a seed probability matrix of 2×2, there are only up to two different ratios for each source vertex u, σ_0 and σ_1.

Lemma 4 presents the property of *translational symmetry*. Intuitively, Lemma 4 means that we can covert $F_u^{-1}(c)$ to $F_u^{-1}(c')$ by using the constant ratios above where $c' < c$. We can convert $F_u^{-1}(c)$ recursively with a smaller c until c becomes less than $F_u(1)$ which means that there are no index k satisfying $F_u(2^k) \leq c < F_u(2^{k+1})$.

Lemma 4. **Property of translational symmetry**
For each $R = 2^k$ $(0 \leq k < \log|V|)$, the following relationship is satisfied for $\forall r : 0 \leq r < R$, in the CDF distribution for a source vertex u.

$$F_u(R + r) = F_u(R) + \sigma_{u[k]} \cdot F_u(r)$$

PROOF. Please see Appendix A. □

We explain an example of Lemma 4 using Figures 3 and 4. Figure 4(b) shows that $F_u(r') = F_u(R) + \sigma_{u[k]} \cdot F_u(r)$ when $r' = R + r$ and $k = \log|V| - 1$. That means that we can convert $F_u^{-1}(C + \sigma_{u[k]} \cdot c)$ to $F_u^{-1}(c)$ by using $C = F_u(R)$ and $\sigma_{u[k]}$. In Figures 3(b), for $u = 2$, $k = 2$, and $R = 4$, $F_2(4 + r) = F_2(4) + \sigma_{2[2]} \cdot F_2(r)$. Especially, $r = 2$, $F_2(4+2) = F_2(4) + \sigma_{2[2]} \cdot F_2(2)$ = $0.105 + \frac{2}{5} \cdot 0.07 = 0.133$. Since $F_2(4)$ is already stored in $RecVec[2]$, and $\sigma_{2[2]}$ can be easily calculated using a seed matrix as in Lemma 3, we can very efficiently utilize Lemma 4 for recursive selection.

We now present Theorem 2 to generate an edge from a source vertex u by using Lemma 3 and Lemma 4. The basic idea of edge generation using the recursive vector model is to translate a given random value x until finding a destination vertex $F_u^{-1}(x)$. In detail, an edge is generated by the following five steps: (1) generating a random value x following a uniform random variable $\mathbb{U}(0, F_u(|V|))$; (2) finding the index k satisfying $F_u(2^k) \leq x < F_u(2^{k+1})$ $(O(\log\log|V|))$; (3) inserting k to the set θ and translating x using Lemma 4; (4) repeating the steps (2)-(3) until there is no index k satisfying $F_u(2^k) \leq x < F_u(2^{k+1})$ $(O(\log|V|))$; and (5) determining an edge (u, v), where $v = \sum_{k\in\theta} 2^k$.

Theorem 2. **Determination of an edge**
Given a source vertex u and a random value x following a uniform random variable $\mathbb{U}(0, F_u(|V|))$, x is recursively translated to the position of a certain destination v using the property of translational symmetry on CDF as follows:

$$F_u^{-1}(x) = \begin{cases} 2^k + F_u^{-1}(\frac{x - F_u(2^k)}{\sigma_{u[k]}}), if\ \exists\, k : F_u(2^k) \leq x < F_u(2^{k+1}) \\ 0, otherwise. \end{cases}$$

PROOF. Please see Appendix B. □

Figure 5 shows an example of edge generation. We assume that a source vertex $u = 2$, and a random value $x = 0.133$. Figure 5(a) shows $RecVec$ for u. In the first iteration, $k = 2$ satisfies $F_2(2^k) \leq x < F_2(2^{k+1})$ since $F_2(2^2) = 0.105$, and $F_2(2^3) = 0.147$, in $RecVec$. Thus, $\theta = \{2\}$, and $F_2^{-1}(0.133) = 2^2 + F_2^{-1}(\frac{0.133-0.105}{0.4})$=$2^2 + F_2^{-1}(0.07)$. In the second iteration, $k = 1$ satisfies $F_2(2^k) \leq x < F_2(2^{k+1})$ since $F_2(2^1) = 0.07$, and $F_2(2^2) = 0.105$, in $RecVec$. Thus, $\theta = \{1, 2\}$, and $F_2^{-1}(0.07) = 2^1 + F_2^{-1}(\frac{0.07-0.07}{1.0})$=$2^1 + F_2^{-1}(0)$. In the third iteration, there is no k satisfying the condition, we stop and determine an edge $(u, v) = (2, 6)$, where $v = \sum_{k\in\theta} 2^k = 4 + 2$.

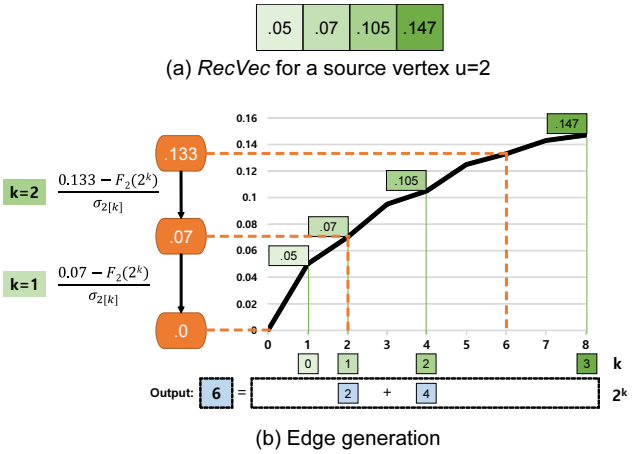


Figure 5: Edge generation in the recursive vector model.

4.3 Performance Analysis

TrillionG is a scale-up and speed-up method that can generate a larger-scale synthetic graph than the existing methods, and at same time, generate the same-sized graph faster than the existing ones. The scale-up feature is achieved mainly due to using a very small-sized scope by the AVS model as explained in Section 3. The speed-up feature is achieved mainly due to the reduced time complexity by P as in Table 1 and the recursive vector model. In terms of performance, the recursive vector model contains the following three key ideas: (1) reusing pre-computed $RecVec$; (2) reducing the number of recursions; (3) reducing the number of random value generations. We analyze each idea in terms of performance.

Reusing pre-computed $RecVec$: TrillionG constructs a small pre-computed vector $RecVec$ per each scope and uses it repeatedly for determining all edges within the corresponding scope. Since the size of $RecVec$ is just $O(log|V|)$ as in Table 1, it can easily be kept in CPU cache during generation of all adjacent edges from a source vertex. For example, when $|E|$ is one trillion, the length of $RecVec$ is just 36. On the contrary, RMAT cannot reuse pre-computed result like $RecVec$ since it inherently requires $O(log|V|)$ recursions for determination of an edge. TrillionG performs each recursion with only a single binary search upon $RecVec$ in cache, which is very fast.

Reducing the number of recursions: The number of recursions for a destination edge is $O(log|V|)$ in RMAT. In contrast, due to Theorem 2, TrillionG only need to perform recursions as many times as the number of 1s in a binary representation of a destination vertex ID to be generated. If the degree distribution is uniform, i.e., $\alpha = \beta = \gamma = \delta = 0.25$, about 50% of bits in a bit string would be 1s. However, non-equal seed parameters are usually used for the skewness of degree distribution, and the number of 1s becomes much smaller than $\frac{log|V|}{2}$. Lemma 5 in Appendix A shows the relationship between the number of 1s and given seed parameters. For the standard seed parameters $K = [0.57, 0.19; 0.19, 0.05]$, the number of 1s is approximately converged to $\frac{log|V|}{4.917}$. That means the number of recursions is reduced by up to 4.917 times in theory.

Reducing the number of random value generations: RMAT should generate a random value for each recursion in order to decide a quadrant. On the contrary, TrillionG generates a random value only once and uses it continuously until it determines a destination vertex by using Theorem 2. This idea reduces the number of random value generations by up to $\log|V|$ times.

5. TRILLIONG SYSTEM

In this section, we present the TrillionG system based on the recursive vector model. Algorithm 4 is the scope generation function for TrillionG, which is based on the framework in Algorithm 2. The size of scope $\mathbb{S}(u, V)$, i.e., the number of edges generated is determined by Theorem 1. Then, we can determine the destination vertex v through the recursive selection process described in Theorem 2 by using $RecVec$ of the range $[0 : \log|V|]$ and a random value from a uniform random distribution $\mathbb{U}$ of the range $[0 : RecVec[\log|V|]$. The DetermineEdge function returns v, and then, a new edge (u, v) is generated.

Algorithm 4 Scope Generation for TrillionG

Input: $\mathbb{S}(u, V)$ /* a scope for source vertex u */
1: $edgeSet \leftarrow \emptyset$
2: $numEdges \leftarrow |\mathbb{S}(u, V)|$ /* by Theorem 1 */
3: $RecVec \leftarrow$ create $RecVec$ for a vertex u
4: **while** $count(edgeSet) \leq numEdges$ **do**
5: $x \leftarrow$ a random value from $\mathbb{U}(0, cdfVec[\log|V|])$
6: $v \leftarrow$ `DetermineEdge`$(x, RecVec)$ /* by Theorem 2 */
7: $edgeSet \leftarrow edgeSet \cup \{(u, v)\}$
8: `Store`$(edgeSet)$

Algorithm 5 presents the DetermineEdge function in detail. For simplicity, we denote $F_u(2^k)$ as $F(2^k)$ in the pseudo code. Line 2 searches the index k satisfying the condition, and its complexity becomes $O(\log\log|V|)$ by using the binary search on $RecVec$, which is almost a constant time even for a quadrillion-scale graph. Line 3 calculates σ for the index k, and then, Line 4 calculates the updated value x'. With the updated value, Line 5 calls the DetermineEdge function recursively, and the values $\{2^k\}$ are accumulated for determination of a destination vertex ID v. Line 7 is the stop condition of recursion. We note that if $x < RecVec[0]$ at the first call of the function, its destination vertex ID becomes $v = 0$. For the in-memory representation in RecVec, we are able to use BigDecimal, approximately matches to the IEEE 128-bit floating-point type, for an accurate trillion-scale graph generation.

Algorithm 5 DetermineEdge

Input: x /* random value */
$RecVec$ /* recursive vector */
1: **if** $x \geq RecVec[0]$ **then**
2: $k \leftarrow$ find the index k s.t. $F(2^k) \leq x < F(2^{k+1})$
3: $\sigma \leftarrow \frac{RecVec[k+1]-RecVec[k]}{RecVec[k]}$ /* by Lemma 3 and 4 */
4: $x' \leftarrow \frac{x-RecVec[k]}{\sigma}$
5: **return** $2^k +$ `DetermineEdge`$(x', RecVec)$
6: **else**
7: **return** 0 /* stop condition of recursion */

In general, how to split the workload into machines has a large impact on performance for a parallel method. The existing parallel methods such as RMAT/p tends to have workload skewness where different numbers of edges are gathered via shuffling and sort merged in each machine. On the contrary, TrillionG avoids workload skewness by evenly partitioning a set of edges to generate into machines before edge generation. The ideal number of edges for each machine would be $\frac{|E|}{p}$, where p is the number of threads. It can be achieved by AVS-level partitioning instead of edge-level partitioning.

Figure 6 shows an example of AVS-level partitioning technique of TrillionG. It consists of four steps: combine, gather, repartition, and scatter. We assume that there are two machines M1 and M2, each machine has two threads T1 and T2, and $|E| = 360$. There are a total of twelve scopes (vertices). In the combining step, each thread determines the sizes of scopes, i.e., d_v, by Theorem 1 and combines those sizes according to $\frac{|E|}{p}$. Here, each thread takes an equal number of scopes. For example, M1/T1 determines the sizes of the first three scopes and combines them such that the sum of sizes of each bin becomes about $\frac{|360|}{4} = 90$. Here, the sum of sizes of the last bin is usually smaller than 90. In the gathering step, the bins of all threads are sent to a master thread, i.e., M1/T1, where network communication overhead is quite small since just bin sizes are sent. In the repartitioning step, the master thread combines and repartitions the scopes of bins such that the size of each bin becomes about $\frac{|E|}{p}$. In the scattering step, each bin is sent to each thread such that a thread generates the edges of the scopes corresponding to the bin.

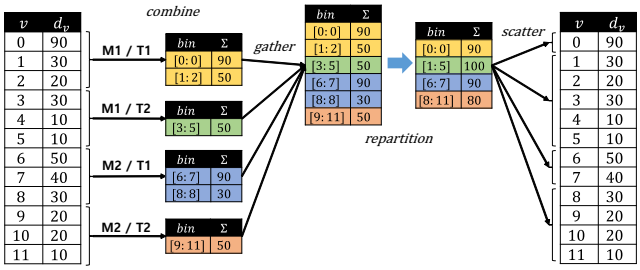


Figure 6: Example of range partition (|E|=360).

The graph format affects the performance of graph processing, but is frequently overlooked in graph generator. Our TrillionG system supports three major graph formats, the edge list text format (shortly, TSV), the 6-byte adjacency list binary format (shortly, ADJ6), and the 6-byte Compressed Sparse Row (CSR) binary format (shortly, CSR6). Here, in order to support trillion-scale graphs, using at least 6-byte representation is necessary. Most of graph generators only supports the TSV format.

TSV is a text format with one edge per line, and so, it is verbose and slow (due to parsing overhead and I/O cost). ADJ is a binary format where each vertex is stored along with its adjacency list. It has little overhead and is well supported by TrillionG since the neighbors of each vertex are generated on the same machine. CSR is like ADJ, but the vertices in the file are sorted, and at the same time, the vertices in each adjacency list are also sorted. Two binary formats, ADJ6 and CSR6, need 6-byte representation of vertex IDs. The file sizes in ADJ6 are usually 3–4 times smaller than those in TSV. As a concrete example, for Scale 38 (i.e., $|V| = 2^{38}$), the TSV file is approximately 90 TB, while the ADJ6 file is 25 TB.

In terms of in-memory representation, the double-precision floating-point type might not be accurate enough to present a probability used in edge generation for trillion-scale graphs. In particular, $RecVec$ has such an issue for finding an accurate position of a destination vertex. Thus, TrillionG uses the BigDecimal type for $RecVec$.

6. RICH GRAPH GENERATION

TrillionG is a fast and scalable engine that generates a large-scale synthetic graph following stochastic models such as RMAT and Kronecker SKG. In the literature, there are a number of graph generators that generate small-scale, but semantically *richer* synthetic graphs having various properties such as multiple node types and edge predicates. They can be classified into two categories of schema-driven methods [9,10] and data-driven methods [20,24,31],

depending on the constraint type. Since most of the techniques used in those methods and the techniques used in TrillionG are orthogonal to each other, TrillionG can be easily extended so as to generate a rich large-scale synthetic graph. In this section, we present how to extend TrillionG so as to generate the graphs supported by gMark [10], a representative schema-driven method. We first extend our recursive vector model in Section 6.1 and present the rich graph generation method using the extended recursive vector model in Section 6.2.

6.1 Extension of Recursive Vector Model

The original recursive vector model in Section 4 generates a graph in which the out-degree distribution is equal to the in-degree distribution. It also generates a graph in which the range of source vertices is equal to that of destination vertices. We extend the recursive vector model such that it can generate a graph having different in-/out-degree distributions and different ranges for source and destination vertices. Such different distributions and ranges are core concepts used in schema-driven rich graph generation methods such as gMark.

The two key steps of the recursive vector model are (1) determination of a scope size for a source vertex in Theorem 1 and (2) determination of a set of edges from the source vertex in Theorem 2. The original recursive vector model uses the same seed parameters α, β, γ, and δ for those two key steps, and as a result, it generates a graph having the same in-/out-degree distribution. The extended recursive vector (ERV) model allows each key step to use different seed parameters in order to generate a graph having different in-/out-degree distributions.

Intuitively, under the AVS-O approach (the black area in Figure 2(c)), the scope sizes correspond to the out-degree distribution, and the edge determination within each scope determines the in-degree distribution. Thus, the ERV model determines the out-degree distribution by setting the seed parameters for the scope sizes (Theorem 1) and the in-degree distribution by setting the seed parameters for edge determination (Theorem 2). We denote the former parameters as $K_{out}[\cdot]$ and the latter parameters as $K_{in}[\cdot]$. For example, we consider a graph in which the out-degree distribution is Zipfian, but the in-degree distribution is Gaussian. The ERV model can generate such a graph by setting $K_{out}(\cdot)$ as Zipfian and $K_{in}[\cdot]$ as Gaussian.

Different seed parameters $K[\alpha, \beta; \gamma, \delta]$ generate different degree distributions including Zipfian and Gaussian. Table 3 shows a few degree distributions generated by the corresponding seed parameters in our ERV model. We prove the relationship between seed parameters and Zipfian distribution in Lemma 6. It means the ERV model can precisely control the slope of Zipfian distribution by adjusting seed parameters, which is not supported by the gMark method [10]. For example, the standard seed parameters $K = [0.57, 0.19; 0.19, 0.05]$ used in Graph500 match the Zipfian distribution with a slope -1.662. In addition to Zipfian and Gaussian, gMark also supports the Uniform distribution, which is very easy to generate by using a simple random function, and so, we omit it in the table.

Now, we explain how the ERV model supports different ranges for source and destination vertices, which is simple. We let V_{src} be the range of source vertices and V_{dst} be the range of destination vertices. Under the AVS-O approach, if $|V_{src}| \geq |V_{dst}|$, we consider a rectangle probability matrix of $|V_{src}| \times |V_{src}|$. Then, when generating an edge (u, v) by Theorem 2, $v' = round(\frac{|V_{dst}|}{|V_{src}|} \cdot v)$ is calculated as a destination vertex in V_{dst}, where $round(\cdot)$ is the nearest integer function. If $|V_{src}| < |V_{dst}|$, we use the same method under the AVS-I approach.

Table 3: Seed parameters and the corresponding distributions.

Seed parameters	Degree distribution
$K_{out}[\alpha, \beta; \gamma, \delta]$	Zipfian with slope $\log(\gamma + \delta) - \log(\alpha + \beta)$
$K_{in}[\alpha, \beta; \gamma, \delta]$	Zipfian with slope $\log(\beta + \delta) - \log(\alpha + \gamma)$
$K[0.25, 0.25; 0.25, 0.25]$	Gaussian with $\mu = \frac{\|E\|}{\|V\|}$

6.2 Schema-driven Graph Generation

The schema-driven methods such as gMark use so-called *graph configuration* for describing a rich synthetic graph to generate. The graph configuration consists of three tables for node types, edge predicates, and in-/out-degree distributions. Figure 7(a) shows an example of graph configuration for a bibliographical synthetic graph in gMark [10]. There are four node types, each of which has different ratios, i.e., ranges of vertices, and three edge predicates, each of which has different ratios. There are also three different degree distributions. Among them, the first one means that all the edges having $researcher$ as source node type and $paper$ as target node type have $author$ as its predicate, where the out-degree distribution follows Zipfian, and the in-degree distribution follows Gaussian.

TrillionG can generate a rich synthetic graph described in a graph configuration by using the ERV model. It conceptually divide the entire probability matrix according to the ranges of vertices of each node type. Figure 7(b) shows the matrix divided by four node types, where we denote the range of $researcher$ vertices as $V_{researcher}$. Then, three degree distributions Figure 7(a) correspond to the colored rectangles in Figure 7(b). For instance, the largest colored rectangle indicates the first degree distribution. TrillionG can generate each colored rectangle according to the method described in Section 6.1. In detail, E_{author} means a set of edges having the $author$ predicate, which size is 50% of $|E|$.

TrillionG based on the ERV model not only generates a much larger rich graph that cannot be generated by gMark, but also, generates a semantically more correct graph than gMark. The graph generation algorithm proposed in gMark cannot remove duplicated edges, and so, could generate the same edge multiple times. For instance, there might be multiple copies of the same edge indicating that $paper\ x$ is published in $conference\ y$. In contrast, TrillionG eliminates such duplicates by default.

7. PERFORMANCE EVALUATION

In this section, we present experimental results in three categories. First, we show the graph properties of RMAT, FastKronecker, and TrillionG, in order to check the correctness of the recursive vector model in Section 7.2. We compare the degree distributions of graphs generated without noise and with noise using the SKG and NSKG versions of TrillionG. We also show the out-/in-degree distribution of a bibliographical synthetic graph generated by our ERV model to demonstrate its validity. Second, we compare the performance of representative graph generators including RMAT [14], RMAT/p, FastKronecker [8], and TrillionG, in order to prove the efficiency and scalability of TrillionG in Section 7.3. For RMAT, we use the state-of-the-art RMAT generator embedded in Apache Spark's GraphX library [22]. RMAT/p is our implementation for a distributed version of RMAT following a merge-based approach described in Section 3.2. FastKronecker is one of the fast Kronecker generators, which is embedded in SNAP [8]. In fact, SNAP contains two Kronecker generators: original Kronecker generator and FastKronecker. Since the original Kronecker

Node types

vertex type	constraint
researcher	50%
paper	30%
conference	10%
city	10%

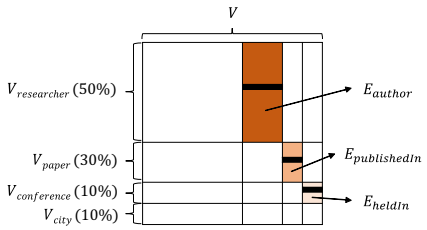
Edge predicates

edge predicate	constraint
author	60%
publishedIn	30%
heldIn	10%

In – and out – degree distributions

source type	predicate	target type	D_{out}	D_{in}
researcher	author	paper	Zipfian	Gaussian
paper	publishedIn	conference	Unif.(1,1)	Gaussian
conference	heldIn	city	Unif.(1,1)	Zipfian

(a) Example of graph configuration in gMark



(b) Generating gMark graphs using the ERV model

Figure 7: Example of rich graph generation.

generator is extremely slow, we only compare FastKrokecker with other methods. Third, we analyze the performance of TrillionG in Section 7.4. In particular, we show impact of three key ideas summarized in Section 4.3 on performance.

7.1 Experimental Environments

We have implemented TrillionG with the Scala language (2.11.8-stable) and JAVA (open jdk 1.8.0_60). It is built on top of Apache Spark [3], and so, can support a wide range of storage systems such as HDFS [12, 36], Amazon S3 [1], HBase [2], and local file system. For synthetic graphs generated by TrillionG, users can directly perform graph queries with GraphX [22], the graph library of Apache Spark.

We conduct all the experiments using a cluster of one master PC and ten slave PCs. Each PC is equipped with a single six-core 3.50 GHz CPU, 32 GB memory, and 4 TB HDD. For fair comparison, we divide the comparative experiments into two types: comparison among single thread methods, and comparison among distributed methods. First, since RMAT and FastKronecker are single-thread methods, we compare a single thread version of our method, called TrillionG/seq, with them in a single slave PC. Second, we compare TrillionG with RMAT/p, which all are distributed methods. Here, we run six threads per PC, and so, a total of 60 threads in the cluster.

We use the elapsed times as a measure of performance. Here, we include the time of storing graph into disks. That is, we measure the time between starting graph generation and completing storing graph on disks. As a graph format, RMAT, RMAT/p, and FastKronecker support only the TSV format, and so, we use it. TrillionG supports three formats, TSV, ADJ6 and CSR6, and we use ADJ6 by default.

For graph generation, we use the standard setting used in the Graph500 competition: the 2×2 seed matrix $K = \begin{bmatrix} 0.57 & 0.19 \\ 0.19 & 0.05 \end{bmatrix}$, $|V| = 2^{scale}$, and $|E| = 16 \times |V|$, where $scale$ is a parameter.

7.2 Properties of Generated Graphs

The plot of degree distribution is widely used for checking the property of scale-free graphs. Figure 8 shows the plots of Scale 20 synthetic graphs generated by RMAT, FastKronecker, TrillionG, and TeG [13] without noise. TeG is a simple graph generator that is not based on stochastic process. Here, we show the result of TeG in order to show how a synthetic graph looks if stochastic process is not used when generating a graph. In the figure, the three generators based on stochastic process, RMAT, FastKronecker, and TrillionG, show the same plots, while TeG shows a quite different plot. The original purpose of TeG is generating RMAT-like graphs [13], but it fails to generate such graphs. It simply decomposes the whole matrix into submatrices, each of which corresponds to a scope in our scope-based generation model, and generates a set of edges in each submatrix, where the number of edges per submatrix is statically (early) fixed. As a result, its resulting plot is far from RMAT's plot. In contrast, TrillionG determines the number of edges per scope stochastically as in Theorem 1, and so, its resulting plot is identical with RMAT's plot.

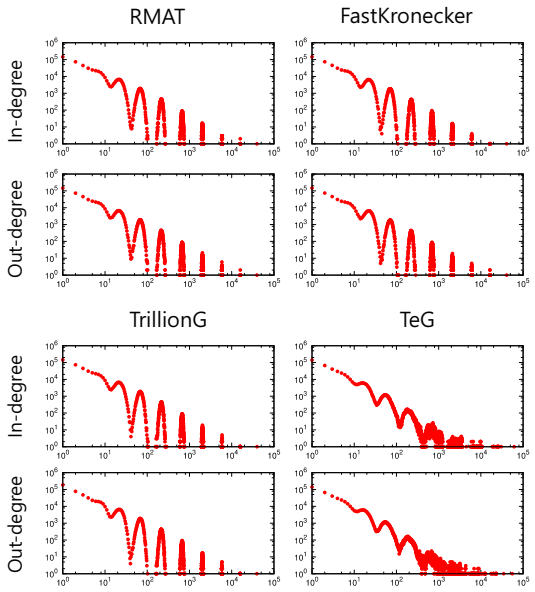


Figure 8: The plot of degree distribution of Scale 20 synthetic graphs generated by four methods.

The NSKG version of TrillionG generates more realistic graphs without oscillation (details are in Appendix C). The three plots in Figure 9 are for graphs generated by the NSKG version of TrillionG while varying the noise paramter N, and are the same as the plots in [35].

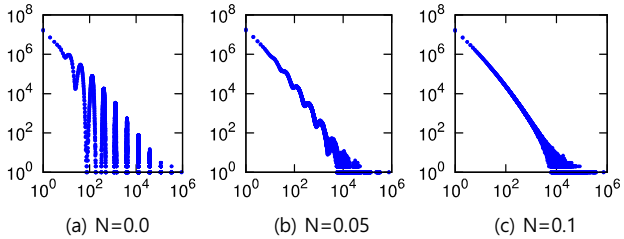


Figure 9: Degree plot of a Scale 27 graph with Graph500 parameters while varying a noise parameter N.

We proposed the ERV model that can generate a rich synthetic graph in Section 6. In detail, our ERV model can easily generate a graph supported by gMark [10]. A typical example graph

of gMark is a bibliographical synthetic graph in Figure 7. Figure 10 shows the plots of out-/in-degree distributions of a graph generated by TrillionG based on the ERV model, where source node type is $researcher$, target node type $paper$, edge predicate is $author$, out-degree distribution is Zipfian, and in-degree distribution is Gaussian. The plots indicate TrillionG based on the ERV model can generate a rich synthetic graph successfully.

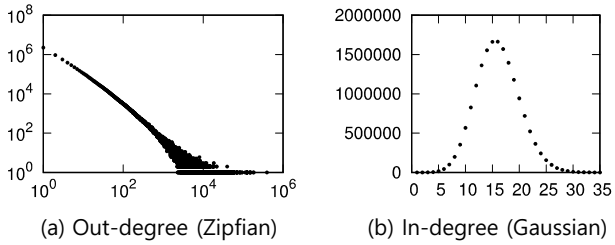


Figure 10: Degree plot of rich graph generated by TrillionG.

7.3 Efficiency and Scalability

Figure 11(a) shows the performance comparison among single threaded methods, RMAT-mem, RMAT-disk, FastKronecker and TrillionG/seq. RMAT-mem is the default version of RMAT, and RMAT-disk is an external-memory implementation of RMAT that eliminates edge duplicates using the external sort. The y-axis represents the average elapsed time in log-scale. FastKronecker is an efficient implementation of Kronecker by using RMAT-like recursive descendant, and so, has the same time complexity with RMAT (i.e., $O(|E|log|V|)$). Its implementation is more efficient than RMAT's, and as a result, it outperforms RMAT-mem and RMAT-disk slightly. The original Kronecker method of $O(|V|^2)$ is much slower than RMAT as explained in Section 3, and we cannot measure its elapsed times due to timeout.

Our TrillionG/seq significantly outperforms all other methods in terms of elapsed time on all graph sizes. It outperforms FastKronecker by up to 10 times for Scale 25 using the same single thread. Here, since $P = 1$, this performance improvement comes from three key ideas as described in Section 4.3. It also generates Scale 28 graphs without memory problem, whereas RMAT-mem and FastKronecker fail to generate even Scale 26 graphs due to out of memory (O.O.M) when using the same 32 GB memory. RMAT-mem and FastKroncker require $O(|E|)$ space as explained in Section 3.3. On the contrary, TrillionG/seq requires only $O(d_{max})$ space for maintaining the generated edges in a scope as explained in Section 3.3 and only $O(\log|V|)$ space for $RecVec$ during edge generation as explained in Section 4.2. Although RMAT-disk can generate Scale 28 graphs as TrillionG/seq does, it shows 18.5 times slower performance than TrillioinG/seq.

Figure 11(b) shows the performance comparison among distributed methods, RMAT/p-mem, RMAT/p-disk, TrillionG using the TSV format, and TrillionG using the ADJ6 format. Our TrillionG significantly outperforms RMAT/p-mem in terms of both elapsed time and graph size. RMAT/p-mem can reduce the amount of memory usage about by $p = 10$ times when ϵ is negligible as explained Section 3.2, and so, generate p times larger graph. However, due to the workload skewness among machines, it cannot generate p times larger graph in practice. After shuffling the edges generated by each machine, a certain machine (usually the 0-th machine by hashing) end up with too many edges to merge, and so, the method fails to generate a graph within a reasonable time. As a result, the largest graph that RMAT/p-mem can generate is Scale 28.

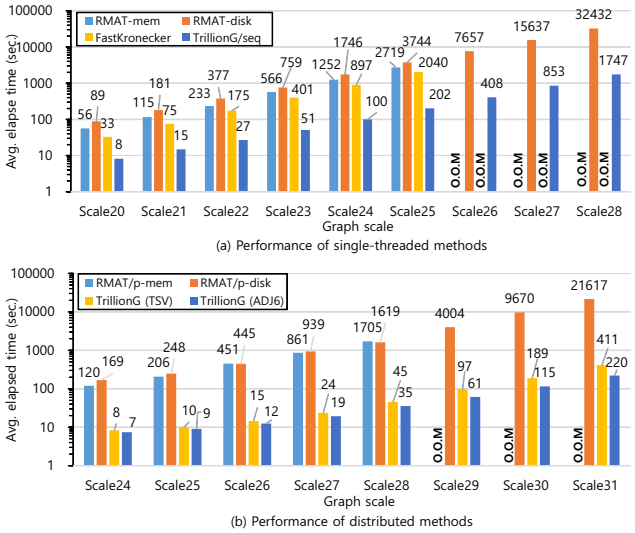


Figure 11: Performance comparison with other methods.

On the contrary, both RMAT/p-disk and TrillionG generate Scale 31 graphs without any memory problem. Between TrillionG (TSV) and TrillionG (ADJ6), the latter is much faster than the former as explained in Section 5.2. In the figure, the performance gap between TrillionG and RMAT/p-disk increases as the scale of graph increases. In particular, TrillionG (ADJ6) outperforms RMAT/p-disk by up to 98 times for Scale 31. It indicates that a graph generation method based on external sort cannot be a good solution for trillion-scale graphs.

Figure12 shows the scalability of TrillionG. With a cluster of ten PCs, TrillionG generates a graph of Scale 36 ($|V| = 64$ billions, $|E| = 1$ trillions) in 6675 seconds (i.e., 1.85 hours). Since the cluster has 35 TB storage capacity on HDFS, we could generate up to Scale 38, which size is 24.74 TB in the ADJ6 format. In Figure12(a), we note that the elapsed time is strictly proportional to the scale of a graph to be generated. Figure12(b) shows the peak memory usage as the graph scale increases. As explained in Section 3.3, the peak memory usage $O(d_{max})$ increases sublinearly and becomes just about 1 GB for Scale 38.

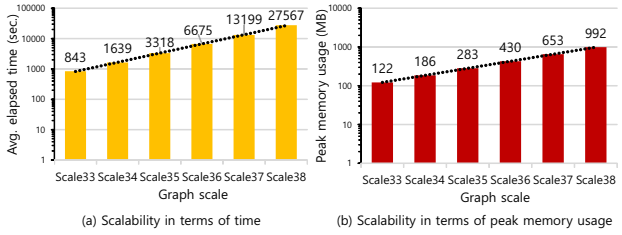


Figure 12: Scalability of TrillionG.

7.4 Impact of Key Ideas on Performance

As presented in Section 4.3, our recursive vector model contains the following three key ideas in terms of performance: (1) reusing pre-computed $RecVec$ (Idea#1); (2) reducing the number of recursions (Idea#2); (3) reducing the number of random value generations (Idea#3). Figure 13 shows impact of each key idea on performance. The left four bars mean the performance without Idea#1, while the right four bars the performance when applying Idea#1.

The impact of Idea#1 is significant, and it improves the performance at least by 3.38 times solely. When Idea#1 is not applied, the impact of Idea#2 or 3 are not much since the absence of Idea#1 is a major performance bottleneck. On the contrary, when Idea#1 is applied, the situation is changed. Idea#2 improves the performance by 42%, Idea#3 does it by 56%, and both together do it by 2.47 times. That indicates that the number of recursions and random value generations become major bottlenecks once $RecVec$ is used.

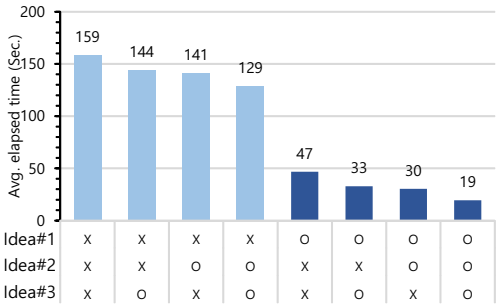


Figure 13: Breakdown of performance (Scale 27).

8. RELATED WORK

We categorize the synthetic graph generation methods into stochastic-based ones, sampling-based ones, and semantic-based ones, depending on the way of generating a graph. The stochastic-based methods focus on fast and scalable graph generation in a stochastic manner. The sampling-based methods focus on generating a large graph using a given small graph based on a sampling technique. The semantic-based methods focus on generating a semantically rich graph that can model real-world linked data, which size is usually very small. In general, the categories have different roles and are just as important as each other.

Although our TrillionG belongs to stochastic-based ones, it can be combined with an existing method of other categories and significantly improve it as shown in Section 6.

Stochastic-based methods: Erdos-Renyi [19, 21] is a simple random graph generation method. For all pairs of two vertices, the probability of generating an edge between them is evenly equal to $\frac{|E|}{|V|^2}$. It is completely equal to the RMAT model with seed parameters $\alpha = \beta = \gamma = \delta = 0.25$ [14]. RMAT [14] and Kronecker [27, 28] are most commonly and widely used methods for synthetic graph generation. For more realistic graph generation, there have been proposed a number of studies to fix the degree distribution on RMAT and Kronecker models [35]. The basic idea behind this direction is to use each different parameters for adding noises during the recursive generation. Graph500 [4] is a representative graph benchmark based on the Kronecker model for measuring the performance of HPC. Preferential attachment [11] or Barabasi and Albert (BA) model follows the concept that an edge is preferentially attached to a higher degree vertex. Recently, there has been proposed a billion-scale graph generation method based on the BA model [23]. Although this method exploits efficient in-memory data structure, it cannot generate a larger-scale graph.

Sampling-based methods: Only few methods belong to this category, and to the best of our knowledge, GSCALER [39] is a representative method in this category. To generate a large syntectic graph similar to a given seed small-size graph, this method adopts the concept of DNA shotgun sequencing. It decomposes a seed graph into small pieces, scales them, then uses the scaled pieces to construct a target graph based on the in-/out-degree correlation of nodes and edges. We believe this sampling-based approach could be very useful for some applications such as human connectomes. Since TrillionG also utilizes in-/out-degree correlation, we believe that it can be further improved to generate a graph of SCALER, which we will study as future work.

Semantic-based methods: These methods can be categorized into schema-driven ones and data-driven ones as addressed in Section 6. gMark [10] and WatDiv [9] are representative schema-driven methods. gMark targets highly tunable generation of graph instances based on user-defined schemas (i.e., graph configurations). It provides four built-in schemas: schema for bibliographical graph database (shown in Section 6), schema for WatDiv [9], schema for LDBC SNB [20], and schema for SP2Bench [34], where the latter three schemas is for generating graphs supported by other three methods or benchmarks. WatDiv [9] is developed to measure the performance of RDF data management systems by generating various synthetic RDF data. Its data generator allows users to define the properties of synthetic data using a schema description language.

Linked Data Benchmark Council (LDBC) Social Network Benchmark (SNB) Datagen [20, 24, 31] is a representative method among data-driven methods and a front runner in social network benchmark. They generate directed labeled graphs having complex graph dependencies by using correlated graph generation techniques. In detail, they generate vertices and edges of various types based on their frequency distributions instead of a predefined distribution such as Zipfian and Gaussian. Although the current TrillionG utilizes predefined distributions, we believe that it is a very promising direction to improve TrillionG to support frequency distributions, for example, by using data dictionaries.

9. CONCLUSIONS

In this paper, we have proposed an efficient and scalable disk-based synthetic graph generator TrillionG that can generate large-scale graphs in a short time using only a small amount of memory. We have generalized the RMAT and Kronecker models to the scope-based generation model, and then, proposed the recursive vector model using a row (i.e., a source vertex) as a scope in the scope-based model. Among the existing models, the pure Kronecker model is too slow due to too small and many scopes, and so, most of graph generators are based on the RMAT (e.g., FastKronecker) or RMAT/P (e.g., Graph500) models. However, RMAT-like models use a too large size scope that requires a large amount of memory, and so, easily fail to generate a large-scale graph. Our recursive vector model is superior to those models in terms of both time and space complexities by using a middle size scope that consumes only a very small amount of memory $O(d_{max})$. In order to generate the same resulting synthetic graph with RMAT-like models in terms of stochastic process very efficiently, we have proposed various concepts including $RecVec$, scale symmetry property, and translation symmetry property, in Lemmas 1-5 and Theorems 1-2. In addition, we have presented the TrillionG system based on the recursive vector model that supports various output formats (e.g., TSV, ADJ6, and CSR6) as well as the noisy SKG version (i.e., NSGK). As a result, TrillionG can generate a trillion-scale graph in two hours only using 10 PCs and generate even a few trillion-scale graph as long as there is enough disk space. Through extensive experiments, we have shown that TrillionG outperforms the state-of-the-art graph generators such as RMAT, FastKronecker, and Graph500 in most cases by orders of magnitude.

10. ACKNOWLEDGMENTS

This work was supported by Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-IT1401-04.

11. REFERENCES

[1] Amazon S3. https://aws.amazon.com/s3.

[2] Apache HBase. http://hbase.apache.org/.

[3] Apache Spark. http://spark.apache.org/.

[4] Graph500 benchmark. http://www.graph500.org/.

[5] Hadoop implementation of RMAT-WES/P. https://github.com/paulmw/rmat.

[6] Human Connectome Project. http://www.humanconnectomeproject.org/.

[7] Rich Data and the Increasing Value of the Internet of Things. http://www.emc.com/leadership/digital-universe/2014iview/internet-of-things.htm.

[8] Stanford Network Analysis Project. http://snap.stanford.edu/.

[9] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee. Diversified stress testing of RDF data management systems. In *International Semantic Web Conference*, pages 197–212. Springer, 2014.

[10] G. Bagan, A. Bonifati, R. Ciucanu, G. H. Fletcher, A. Lemay, and N. Advokaat. gMark: schema-driven generation of graphs and queries. *IEEE Transactions on Knowledge and Data Engineering*, 2016.

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

[12] D. Borthakur. The Hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 11(2007):21, 2007.

[13] U. K. ByungSoo Jeon, Inah jeon. TeGViz: Distributed Tera-Scale Graph Generation and Visualization. In *ICDM*, 2015.

[14] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A Recursive Model for Graph Mining. In *SDM*, volume 4, pages 442–446. SIAM, 2004.

[15] F. Checconi and F. Petrini. Traversing trillions of edges in real time: Graph exploration on large-scale parallel machines. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 425–434. IEEE, 2014.

[16] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: graph processing at Facebook-scale. *VLDB*, 8(12):1804–1815, 2015.

[17] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1-2):48–57, 2010.

[18] C. Dobre and F. Xhafa. Intelligent services for big data science. *Future Generation Computer Systems*, 37:267–281, 2014.

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

[20] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz. The LDBC social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 619–630. ACM, 2015.

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

[22] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613, 2014.

[23] A. Hadian, S. Nobari, B. Minaei-Bidgoli, and Q. Qu. ROLL: Fast In-Memory Generation of Gigantic Scale-free Networks. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1829–1842. ACM, 2016.

[24] A. Iosup, T. Hegeman, W. L. Ngai, S. Heldens, A. Prat-Pérez, T. Manhardto, H. Chafio, M. Capotă, N. Sundaram, M. Anderson, et al. LDBC graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. volume 9, pages 1317–1328. VLDB Endowment, 2016.

[25] M.-S. Kim, K. An, H. Park, H. Seo, and J. Kim. GTS: A Fast and Scalable Graph Processing Method based on Streaming Topology to GPUs. In *Proceedings of the 2016 International Conference on Management of Data*, pages 447–461. ACM, 2016.

[26] P. Kumar and H. H. Huang. G-store: high-performance graph store for trillion-edge processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 71. IEEE Press, 2016.

[27] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos. Realistic, mathematically tractable graph generation and evolution, using Kronecker multiplication. In *PKDD*, pages 133–145. Springer, 2005.

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

[29] J. Lothian, S. Powers, B. D. Sullivan, M. Baker, J. Schrock, and S. W. Poole. Synthetic Graph Generation for Data-Intensive HPC Benchmarking: Background and Framework. 2013.

[30] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 61–72. ACM, 2012.

[31] M.-D. Pham, P. Boncz, and O. Erling. S3G2: A scalable structure-correlated social graph generator. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 156–172. Springer, 2012.

[32] A. Quamar, K. A. Kumar, and A. Deshpande. SWORD: scalable workload-aware data placement for transactional workloads. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 430–441. ACM, 2013.

[33] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 410–424. ACM, 2015.

[34] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP^2Bench: a SPARQL performance benchmark. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 222–233. IEEE, 2009.

[35] C. Seshadhri, A. Pinar, and T. G. Kolda. An in-depth study of stochastic Kronecker graphs. In *ICDM*, pages 587–596. IEEE, 2011.

[36] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *MSST*, pages 1–10. IEEE, 2010.

[37] C. L. Staudt, A. Sazonovs, and H. Meyerhenke. Networkit:

An interactive tool suite for high-performance network analysis. *arXiv preprint arXiv:1403.3005*, 2014.

[38] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou. G ra M: scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 408–421. ACM, 2015.

[39] J. Zhang and Y. Tay. GSCALER: Synthetically scaling a given graph. In *EDBT*, pages 53–64, 2016.

[40] G. K. Zipf. Human behavior and the principle of least effort: An introduction to human ecology. 2016.

APPENDIX

A. PROOF OF LEMMAS

Lemma 1. **Probability of an edge from u.**
The probability of generating edges from u can be calculated as below.

$$P_{u\to} = (\alpha+\beta)^{\texttt{Bits}(\sim u)} \cdot (\gamma+\delta)^{\texttt{Bits}(u)}$$

PROOF. It is similarly proven by Lemma 1.

$$\begin{aligned}\sum_{v\in V} \mathbb{K}_{u,v} =& (K_{u[0],v[0]} + K_{u[0],\sim v[0]}) \times (K_{u[1],v[1]} + K_{u[1],\sim v[1]}) \\ & \times \cdots \times (K_{u[L],\sim v[L]} + K_{u[L],\sim v[L]}) \\ =& \prod_{i\in[0:L]} K_{u[i],v[i]} + K_{u[i],\sim v[i]}\end{aligned}$$

Therefore, $P_{u\to} = (\alpha+\beta)^{\texttt{Bits}(\sim u)} \cdot (\gamma+\delta)^{\texttt{Bits}(u)}$. □

Lemma 2. **CDF value at the positions of power of 2.**
We assume that $x \in [0 : \log|V|]$, and $\gg$ is the logical right shift, then the value of an element of $RecVec$ is defined as follow.

$$RecVec[x] = (\frac{\alpha}{\alpha+\beta})^{\log|V|-x-\texttt{Bits}(u\gg x)} \cdot (\frac{\gamma}{\gamma+\delta})^{\texttt{Bits}(u\gg x)} \cdot P_{u\to}$$

PROOF.

$$\begin{aligned}F_u(2^x) =& \prod_{i\in[0:x-1]} (K_{u[i],v[i]} + K_{u[i],\sim v[i]}) \times \prod_{i\in[x:L]} K_{u[i],v[i]} \\ =& \prod_{i\in[0:L]} (K_{u[i],v[i]} + K_{u[i],\sim v[i]}) \times \prod_{i\in[x:L]} \frac{K_{u[i],v[i]}}{K_{u[i],v[i]} + K_{u[i],\sim v[i]}} \\ =& P_{u\to} \cdot \prod_{i\in[x:L]} \frac{K_{u[i],v[i]}}{K_{u[i],v[i]} + K_{u[i],\sim v[i]}}\end{aligned}$$

Therefore,
$RecVec[x] = P_{u\to} \cdot (\frac{\alpha}{\alpha+\beta})^{\log|V|-x-\texttt{Bits}(u\gg x)} \cdot (\frac{\gamma}{\gamma+\delta})^{\texttt{Bits}(u\gg x)}$.
□

Lemma 3. **Property of scale symmetry.**
For each $R = 2^k (0 \le k < \log|V|)$, a constant ratio $\sigma_{u[k]}$ is maintained for $\forall r : 0 \le r < R$, in the PMF distribution for a source vertex u. We assume $u[k]$ indicates the k-th bit of a vertex u in binary string starting from the right most LSB.

$$\sigma_{u[k]} = \frac{P_{u\to(R+r)}}{P_{u\to r}} = \frac{K_{u[k],1}}{K_{u[k],0}}$$

PROOF. Where $r' = r + R$, and $0 \le r < R = 2^k$, a ratio σ between the probability of determining a vertex r and that of determining a vertex r' is denoted as

$$\sigma = \frac{P_{u\to r}}{P_{u\to r'}} = \frac{\prod_{i\in[0:L]} K_{u[i],r[i]}}{\prod_{i\in[0:L]} K_{u[i],r'[i]}}$$

The difference between r and r' is only a k-th bit value, i.e., 0 for $r[k]$ and 1 for $r'[k]$. $\forall i : i \in [0, L]$, it satisfies that

$$\frac{K_{u[i],r'[i]}}{K_{u[i],r[i]}} = \begin{cases} 1, & \text{if } i \neq k \\ \frac{K_{u[i],1}}{K_{u[i],0}}, & \text{if } i = k. \end{cases}$$

Therefore, $\sigma_{u[k]} = \frac{P_{u\to(R+r)}}{P_{u\to r}} = \frac{K_{u[k],1}}{K_{u[k],0}}$. □

Lemma 4. **Property of translational symmetry.**
For each $R = 2^k (0 \le k < \log|V|)$, the following relationship is satisfied for $\forall r : 0 \le r < R$, in the CDF distribution for a source vertex u.

$$F_u(R+r) = F_u(R) + \sigma_{u[k]} \cdot F_u(r)$$

PROOF.

$$\begin{aligned}F_u(R+r) =& \sum_{v=0}^{R+r-1} P_{u\to v} = \sum_{v=0}^{R-1} P_{u\to v} + \sum_{v=R}^{R+r-1} P_{u\to v} \\ =& \sum_{v=0}^{R-1} P_{u\to v} + \frac{K_{u[k],1}}{K_{u[k],0}} \cdot \sum_{v=0}^{r-1} P_{u\to v} \\ =& F_u(R) + \frac{K_{u[k],1}}{K_{u[k],0}} \cdot F_u(r) = F_u(R) + \sigma_{u[k]} \cdot F_u(r)\end{aligned}$$

Therefore, $F_u(R+r) = F_u(R) + \sigma_{u[k]} \cdot F_u(r)$. □

Lemma 5. **The number of 1s in a binary string.**
Let $bitStr$ be a bit string of a destination vertex ID, and $bitStr_1$ be the number of 1s in $bitStr$. Then, $bitStr_1$ is approximately converged to the below equation.

$$binStr_1 \approx \frac{\log|V|}{\frac{\alpha+\beta}{\beta} + 1 - \frac{\beta\cdot(\gamma+\delta)}{\delta\cdot(\alpha+\beta)}}$$

PROOF.

$$\begin{aligned}binStr_1 =& \sum_{i=0}^{log|V|-1} 0 \cdot P[v(i)=0] + 1 \cdot P[v(i)=1] && (7) \\ =& \sum_{i=0}^{log|V|-1} \frac{\beta}{\alpha+\beta} \cdot \sim u(i) + \frac{\delta}{\gamma+\delta} \cdot u(i) && (8) \\ =& \texttt{bitSum}(\sim u) \cdot \frac{\beta}{\alpha+\beta} + \texttt{bitSum}(u) \cdot \frac{\delta}{\gamma+\delta} && (9) \\ \approx& (\log|V| - binStr_1) \cdot \frac{\beta}{\alpha+\beta} + binStr_1 \cdot \frac{\delta}{\gamma+\delta} && (10) \\ \iff& binStr_1 \approx \frac{\log|V|}{\frac{\alpha+\beta}{\beta} + 1 - \frac{\beta\cdot(\gamma+\delta)}{\delta\cdot(\alpha+\beta)}} && (11)\end{aligned}$$

□

Lemma 6. **Zipfian distribution and seed parameters.**
$K[\alpha, \beta; \gamma, \delta]$ generates the Zipfian out-degree distribution of a slope $\log(\gamma+\delta) - \log(\alpha+\beta)$ or the Zipfian in-degree distribution of a slope $\log(\beta+\delta) - \log(\alpha+\gamma)$.

PROOF. The Zipfian distribution consists of two elements, rank and frequency, where frequency of $Rank$-th frequent data item is inversely proportional to its $Rank$ [40]. In addition, the shape of Zipfian is described by its slope, especially, log-log slope of the rank-frequency distribution as below.

$Freq \propto Rank^{slope} \Leftrightarrow slope = \frac{\Delta\log(Freq)}{\Delta\log(Rank)}$

By using Lemma 1, we can calculate the probability (frequency) of 2^k-th ranked vertex in out-degree distribution as in Eq. (12).

$$Freq(2^k) = (\alpha+\beta)^{\log|V|-k} \times (\gamma+\delta)^k \qquad (12)$$

Then, by using the frequencies of two arbitrary 2^{k_1}-th and 2^{k_2}-th ranked vertices, we can derive the equation for the slope as in Eq. (13).

$$\begin{aligned}\frac{\Delta\log(Freq)}{\Delta\log(Rank)} &= \frac{\log Freq(2^{k_1}) - \log Freq(2^{k_2})}{\log(2^{k_1}) - \log(2^{k_2})} \\ &= \frac{\log((\alpha+\beta)^{\log|V|-k_1} \times (\gamma+\delta)^{k_1})}{k_1 - k_2} \\ &\quad - \frac{\log((\alpha+\beta)^{\log|V|-k_2} \times (\gamma+\delta)^{k_2})}{k_1 - k_2} \\ &= \frac{(\log|V| - k_1)\log(\alpha+\beta) + k_1\log(\gamma+\delta)}{k_1 - k_2} \\ &\quad - \frac{(\log|V| - k_2)\log(\alpha+\beta) + k_2\log(\gamma+\delta)}{k_1 - k_2} \\ &= \log(\gamma+\delta) - \log(\alpha+\beta)\end{aligned} \tag{13}$$

Thus, $K[\alpha, \beta; \gamma, \delta]$ generates the Zipfian out-degree distribution of a slope $\log(\gamma+\delta) - \log(\alpha+\beta)$. We can prove the Zipfian in-degree distribution in the same manner. □

B. PROOF OF THEOREMS

Theorem 1. **Size of a scope $S(i, V)$.**
We denote that $n = |E|$ and $p = \sum_{j\in V} P_{i\to j} = P_{i\to}$. Then, the size of a scope $S(i, V)$ has approximately a normal distribution with $\mu = np$ and $\sigma = \sqrt{np(1-p)}$.

PROOF. The size of a scope is the number of successes of edge generation within the scope $S(i, V)$. It is a binomial experiment which consists of a sequence of n trials for distinguishing either success or fail, and the success probability p per each trial, since the trials are independent and identical. We let n and p denote the number of trial and the success probability for each trial respectively. And it becomes $n = |E|$ and $p = \sum_{j\in V} P_{i\to j} = P_{i\to}$. The size of a scope follows the binomial distribution $B(n, p)$. It is able to approximate the normal distribution $N(np, np(1-p))$ for very large n and very small p. □

Theorem 2. **Determination of an edge.**
Given a source vertex u and a random value x following a uniform random variable $\mathbb{U}(0, F_u(|V|))$, x is recursively translated to the position of a certain destination v using the property of translational symmetry on CDF as follows:

$$F_u^{-1}(x) = \begin{cases} 2^k + F_u^{-1}(\frac{x - F_u(2^k)}{\sigma_{u[k]}}), if\ \exists\, k : F_u(2^k) \le x < F_u(2^{k+1}) \\ 0, otherwise. \end{cases}$$

PROOF. By Lemma 4, where $x = F_u(R + r)$ and $x' = F_u(r)$,

$$F_u(R+r) = F_u(R) + \sigma_{u[k]} \cdot F_u(r)$$

$$\iff F_u(r) = \frac{F_u(R+r) - F_u(R)}{\sigma_{u[k]}} \iff x' = \frac{x - F_u(R)}{\sigma_{u[k]}}$$

Hence, $F_u^{-1}(x) = R + r$ and $F_u^{-1}(x') = r$, and then,

$$F_u^{-1}(x) - F_u^{-1}(x') = R = 2^k \iff F_u^{-1}(x) = 2^k + F_u^{-1}(x')$$

$$\iff F_u^{-1}(x) = 2^k + F_u^{-1}(\frac{x - F_u(R)}{\sigma_{u[k]}})$$

The inverse function recurses itself until there are no index k satisfying $F_u(2^k) \le x < F_u(2^{k+1})$, then it indicates the destination vertex v. □

C. ADDING RANDOM NOISE

Some previous studies [35] have pointed out that, although RMAT and Kronecker follow the power-law distribution [11], a graph generated by them show a kind of oscillated slope in its log-log plot as in Figure 9(a), where X-axis indicates the degree of a vertex, and Y-axis the number of vertices. Since the recursive vector model follows the same stochastic model with RMAT and Kronecker, its output graph also shows the plot like Figure 9(a). In order to generate more realistic synthetic graphs, there has been proposed a method that adds random noise when generating a graph [35]. That method is known as Noisy SKG (shortly, NSKG), which can generate a much more realistic graph in terms of log-log plot, as in Figure 9(c). Since NSKG is a noisy version only for the SKG model of Kronecker, we need to develop a noisy version for the recursive vector model which is equal to NSKG in terms of stochastic process.

For generating a Scale $\log|V|$ graph $\mathbb{K}$, SKG performs the Kronecker product $\log|V|$ times with the same seed matrix K, i.e., $\mathbb{K} = K \bigotimes K \bigotimes K \cdots \bigotimes K = K^{\bigotimes \log|V|}$, as in Section 2. In contrast, NSKG performs the product $\log|V|$ times with different matrices, i.e., $\mathbb{K} = K_0 \bigotimes K_1 \cdots \bigotimes K_{\log|V|-1}$. Here, a noisy seed matrix K_i ($0 \le i < \log|V|$) is defined as in Definition 3.

Definition 3. **Specification for adding a noise.**
We assume that μ_i is a value from a uniform random variable of the range $[-N : N]$, where N is a noise parameter $s.t.\ N \le \min(\frac{\alpha+\delta}{2}, \beta)$. Then, the i-th noisy seed matrix K_i is defined as $K_i = \begin{bmatrix} \alpha(1 - \frac{2\mu_i}{\alpha+\delta}) & \beta + \mu_i \\ \gamma + \mu_i & \delta(1 - \frac{2\mu_i}{\alpha+\delta}) \end{bmatrix}$.

Intuitively, as the noise parameter N gets larger, the four parameter values of each seed matrix K_i are perturbed more, and so, the phenomenon of oscillation in log-log plot disappears. Figure 9(b) shows the plot when $N = 0.05$, and Figure 9(c) shows that when $N = 0.1$. Typically, $N = 0.1$ is used, and please refer [35] for more detail.

In order to make the TrillionG system generate a graph following NSKG, i.e., a noisy version for the recursive vector model, we change from Lemma 1 to Lemma 7 and from Lemma 2 to Lemma 8. Intuitively, $P'_{u\to}$ in Lemma 7 has a probability of a source vertex u, i.e., a row u, in the NSKG matrix in Definition 3. Thus, we replace $P_{u\to}$ with $P'_{u\to}$ in Theorem 1 for determination of $|\mathbb{S}(u, V)|$, i.e., the number edges from u for NSKG. Likewise, we replace $RecVec$ with $RecVec'$ in Algorithms 4-5 for determination of the destination vertices from u for NSKG.

Definition 4. **Hadamard product**: Given two matrices A and B, where A and B are of $n \times m$, the Hadamard product between A and B is defined as

$$A \circ B = \begin{bmatrix} a_{1,1}b_{1,1} & \cdots & a_{1,m}b_{1,m} \\ \vdots & \ddots & \vdots \\ a_{n,1}b_{n,1} & \cdots & a_{n,m}b_{n,m} \end{bmatrix}.$$

Lemma 7. **Probability of the edges from u for NSKG.**
Let $c_{\alpha\beta} = -\frac{\alpha-\delta}{(\alpha+\delta)(\alpha+\beta)}$, and $c_{\gamma\delta} = \frac{\alpha-\delta}{(\alpha+\delta)(\gamma+\delta)}$, then the probability $P'_{u\to}$ of generating edges from u for NSKG can be calculated as below.

$$P'_{u\to} = P_{u\to} \times \prod_{i\in[0:\log|V|-1]} (1 + c_{\alpha\beta}\mu_i)^{\sim u[i]} (1 + c_{\gamma\delta}\mu_i)^{u[i]}$$

PROOF. Let $K_{i|u,v}$ be an element of u, v position of i-th noisy seed K_i, and $M^{(i)+}$ be a matrix $s.t.\ K^{(i)} \times \begin{bmatrix} 1 \\ 1 \end{bmatrix} = (K \times \begin{bmatrix} 1 \\ 1 \end{bmatrix}) \circ$

$M^{(i)+}$.

$$K^{(i)} \times \begin{bmatrix}1\\1\end{bmatrix} = \begin{bmatrix}(\alpha - \frac{2\alpha\mu_i}{\alpha+\delta}) + (\beta + \mu_i)\\ (\gamma + \mu_i) + (\delta - \frac{2\delta\mu_i}{\alpha+\delta})\end{bmatrix}$$

$$= \begin{bmatrix}(\alpha+\beta) + \frac{(-\alpha+\delta)\mu_i}{\alpha+\delta}\\ (\gamma+\delta) + \frac{(\alpha-\delta)\mu_i}{\alpha+\delta}\end{bmatrix} = \begin{bmatrix}\alpha+\beta\\ \gamma+\delta\end{bmatrix} \circ \begin{bmatrix}1 + \frac{(-\alpha+\delta)\mu_i}{(\alpha+\delta)(\alpha+\beta)}\\ 1 + \frac{(\alpha-\delta)\mu_i}{(\alpha+\delta)(\gamma+\delta)}\end{bmatrix}.$$

Hence, $M^{(i)+}$ becomes $\begin{bmatrix}1 + \frac{(-\alpha+\delta)\mu_i}{(\alpha+\delta)(\alpha+\beta)}\\ 1 + \frac{(\alpha-\delta)\mu_i}{(\alpha+\delta)(\gamma+\delta)}\end{bmatrix}$.

Similar to Lemma 1, the probability of an edge from u for NSKG can be calculated as follows:

$$\sum_{v\in V} \mathbb{K}'_{u,v} = (K_{0|u[0],v[0]} + K_{0|u[0],\sim v[0]}) \times (K_{1|u[1],v[1]} + K_{1|u[1],\sim v[1]}) \times \cdots \times (K_{L|u[L],v[L]} + K_{L|u[L],\sim v[L]})$$

$$= \prod_{i\in[0:L]} (K_{i|u[i],v[i]} + K_{i|u[i],\sim v[i]})$$

$$= \prod_{i\in[0:L]} (K_{u[i],v[i]} + K_{u[i],\sim v[i]}) \times \prod_{i\in[0:L]} M^{+}_{0|u[i]}$$

$$= P_{u\rightarrow} \cdot \prod_{i\in[0:L]} M^{+}_{0|u[i]}.$$

Therefore, where $c_{\alpha\beta} = -\frac{\alpha-\delta}{(\alpha+\delta)(\alpha+\beta)}$, and $c_{\gamma\delta} = \frac{\alpha-\delta}{(\alpha+\delta)(\gamma+\delta)}$, $P'_{u\rightarrow} = P_{u\rightarrow} \times \prod_{i\in[0:L]} (1 + c_{\alpha\beta}\mu_i)^{\sim u[i]}(1 + c_{\gamma\delta}\mu_i)^{u[i]}$. □

Lemma 8. **CDF value at the positions of power of 2 for NSKG.** We assume that $x \in [0 : \log|V|]$, $c_\alpha = -\frac{2}{\alpha+\delta}$, and $c_\gamma = \frac{1}{\gamma}$, then the value of an element of $RecVec'$ for NSKG is defined as follow.

$$RecVec'[x] = RecVec[x] \times \prod_{i\in[0:\log|V|-k-1]} (1 + c_\alpha\mu_i)^{\sim u[i]}(1 + c_\gamma\mu_i)^{\sim u[i]} \times \prod_{i\in[\log|V|-k:\log|V|-1]} (1 + c_{\alpha\beta}\mu_i)^{\sim u[i]}(1 + c_{\gamma\delta}\mu_i)^{u[i]}$$

PROOF. Let M_i be a matrix $s.t.$ $K_i = K \circ M_i$.

$$K_i = \begin{bmatrix}\alpha(1 - \frac{2\mu_i}{\alpha+\delta}) & \beta(1 + \frac{\mu_i}{\beta})\\ \gamma(1 + \frac{\mu_i}{\gamma}) & \delta(1 - \frac{2\mu_i}{\alpha+\delta})\end{bmatrix} = \begin{bmatrix}\alpha & \beta\\ \gamma & \delta\end{bmatrix} \circ \begin{bmatrix}1 - \frac{2\mu_i}{\alpha+\delta} & 1 + \frac{\mu_i}{\beta}\\ 1 + \frac{\mu_i}{\gamma} & 1 - \frac{2\mu_i}{\alpha+\delta}\end{bmatrix}.$$

Hence, the modifier matrix M_i is defined as $\begin{bmatrix}1 - \frac{2\mu_i}{\alpha+\delta} & 1 + \frac{\mu_i}{\beta}\\ 1 + \frac{\mu_i}{\gamma} & 1 - \frac{2\mu_i}{\alpha+\delta}\end{bmatrix}$.

Similar to Lemma 2, the CDF value at the positions of power of 2 for NSKG can be calculated as follows:

$$F'_u(2^x) = \prod_{i\in[0:x-1]} (K_{i|u[i],v[i]} + K_{i|u[i],\sim v[i]}) \times \prod_{i\in[x:L]} K_{i|u[i],v[i]}$$

$$= F_u(2^x) \cdot \prod_{i\in[0:x-1]} M^{+}_{i|u[i]} \cdot \prod_{i\in[x:L]} M_{i|u[i],0}.$$

Therefore, where $c_\alpha = -\frac{2}{\alpha+\delta}$, and $c_\gamma = \frac{1}{\gamma}$,

$RecVec'[x] = RecVec[x] \times \prod_{i\in[0:L-x]} (1 + c_\alpha\mu_i)^{\sim u[i]}(1 + c_\gamma\mu_i)^{\sim u[i]} \times \prod_{i\in[L-x+1:L]} (1 + c_{\alpha\beta}\mu_i)^{\sim u[i]}(1 + c_{\gamma\delta}\mu_i)^{u[i]}$. □

D. COMPARISON WITH GRAPH500

In this section, we compare the performance of TrillionG with that of the Graph500 benchmark in order to show the superiority of TrillionG. Here, since the performance of Graph500 is largely affected by the speed of network, we use the high-speed network of 100 Gbps (Infiniband EDR). We have used two kinds of network settings: 1 Gbps ethernet for a default setting, and 100 Gbps InfiniBand EDR for Graph500 benchmark. In the case of Graph500, we do not include storing time since it is inherently an in-memory framework.

Graph500 [4] is an in-memory benchmark for graph generation and processing. Unlike RMAT/p, Graph500 uses so-called scramble mechanism that relabels vertex IDs via perfect hashing among machines in order to avoid the workload skewness problem. As a result, it can generate larger graphs than RMAT/p with the same cluster.

We measure the performance of those two steps of Graph500 for comparison with TrillionG. Here, since Graph500 generates a noisy SKG graph as shown in Figure 9(c) in the CSR-like structure, we also evaluate the performance of the NSKG verstion of TrillionG ($N = 0.1$) using the CSR6 format.

Figure 14(a) shows the comparison results between TrillionG and Graph500. Since TrillionG has no shuffling and merging, both TrillionG-IB and TrillionG-1G show the same performance. Although TrillionG-1G uses 100 times slower network than Graph500-IB, it significantly outperforms Graph500-IB in terms of both the elapsed time and scalability. Graph500 fails to generate a graph larger than Scale 30 due to out of memory.

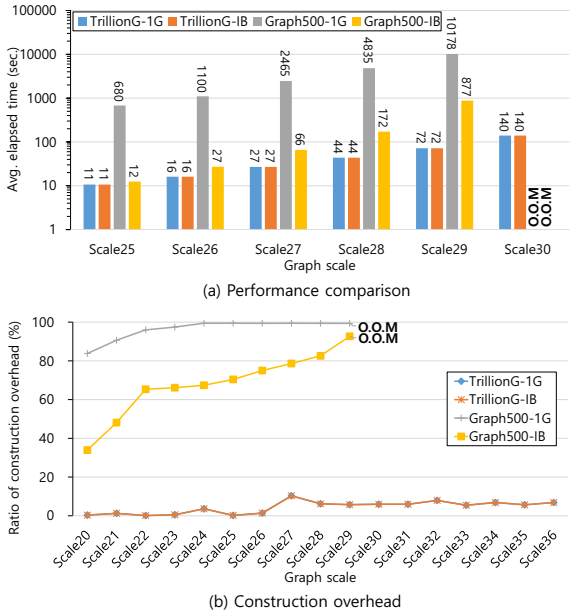


Figure 14: TrillionG vs. Graph500.

Figure 14(b) shows the ratio of the construction time to whole graph generation time. Here, the construction time includes shuffling, merging, and converting to the CSR-like format. In the figure, TrillionG usually has $6 - 7\%$ construction overhead. On the contrary, Graph500 usually has very high ratios, especially more than 90% at Scale 29.