CS224W: Methods of Parallelized Kronecker Graph Generation

Sean Choi, Group 35 [*]

December 10th, 2012

1 Introduction

The question of generating realistic graphs has always been a topic of huge interests. This topic has gained huge attention over the past few years with the advent of massive real-world network data that re generated by large software companies like Facebook and Google, along with the increase in the computation power that makes anyone capable of processing them. With real graphs at massive scale and parallelized frameworks to analyze them, network analysis became a major topic of scientific research. As the need to analyze these networks grew, the question of modeling and generating a real-world network graph at the same scale also became a topic of interest. Out of many approaches to model real-world networks, Stochastic Kronecker Graph (SKG) generation and its predecessor R-MAT generation have attracted interest in the network analysis community, due to their simplicity and their abilities to capture the properties of real-world networks. Along with such algorithms, a new programming methods to process large graphs called vertex-centric BSP with the implementations such as Pregel [3], Apache Giraph, GPS, and Apache Hama have become increasingly popular as an alternative to MapReduce and Hadoop, which are ill-suited to run massive scale graph algorithms [3]. The SKG, R-MAT and vertex-centric BSP, however, are not well-suited for each other. The obvious approach of parallelizing SKG, which is to generate edges in parallel, is not "vertex-centric" in nature and therefore is unnatural to program and runs inefficiently in vertex-centric BSP. Therefore we present a new network generation model, which we call Poisson Stochastic Kronecker Graph (PSKG), as an alternative. In this model, the out-degree of each vertex is determined by independent but non-identical Poisson random variables and the destination node of the edges are determined in a recursive manner similar to SKG.

We will show that the resulting algorithm, PSKG, is essentially equivalent to SKG and will therefore retain all the desired properties of it. Unlike SKG, however, PSKG is embarrassingly parallel in a vertex-centric manner and therefore is very well-suited for vertex-centric BSP. The advantage of PSKG is that it is easily parallelized on vertex-centric BSP, requires no communication between computational nodes, and yet retains all the desired properties of SKG. Finally, given PKSG, I will show the performance differences when using PSKG versus SKG and parallelized SKG that are suited for MapReduce frameworks.

2 Theory and Algorithm

We will first start by discussing the original SKG and an equivalent formulation of it as this path will motivate parallelized SKG and finally PSKG.

2.1 Stochastic Kronecker Graph

The main idea of generating Kronecker graphs is to use a linear algebra concept of Kronecker matrix multiplication. Kronecker multiplication is defined as following. For two matrices $A = [a_{i,j}], B$ of size $n \times m$ and $m' \times n'$, the Kronecker multiplication of A and B will produce a matrix

[*]In collaboration with Ernest Ryu from Stanford ICME

C of size $(n \cdot n') \times (m \cdot m')$, which is given by

$$C = A \otimes B \doteq \begin{pmatrix} a_{1,1}B & a_{1,2}B & \cdots & a_{1,n}B \\ a_{2,1}B & a_{2,2}B & \cdots & a_{2,n}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1}B & a_{m,2}B & \cdots & a_{m,n}B \end{pmatrix}$$

Given the definition of Kronecker multiplication, we can see that the Kronecker matrix of size $n^2 \times n^2$ can be obtained by computing a Kronecker product of matrix of size $n \times n$ to itself. Following a similar fashion, we can see that a large matrix can quickly be generated from a matrix of very small size. Thus, extending this idea further, the Kronecker graph generation method is defined as follows. The method begins with a small adjacency matrix K_1 of arbitrary size $(n \times n)$. Given that, the algorithm repeatedly takes the Kronecker multiplication of K_1 to itself until the desired size is reached. When the desired size is reached, the algorithm is completed.

The naive method of carrying out repeated Kronecker power of the initiator matrix is not efficient and results in an algorithm of $O(N^2)$ time complexity. Thus, a variation of Kronecker random graph generation, called the Stochastic Kronecker random graph generation (SKG) was proposed by [1]. The algorithm is as follows. Consider the problem of generating a random graph of E edges and $N = n^k$ vertices, where $k \in \mathbb{N}$. Let

$$P = \begin{bmatrix} p_{0,0} & \cdots & p_{0,n-1} \\ \vdots & \ddots & \vdots \\ p_{n-1,0} & \cdots & p_{n-1,n-1} \end{bmatrix} \in \mathbb{R}^{n \times n}$$

be the "initiator matrix" where $p_{i,j} \geq 0$ and $\sum_{i,j} p_{i,j} = 1$, rather than using K_1, which can have a sum of greater than 1.

The approach in SKG is to start off with an empty adjacency matrix and generate random edge by inserting one into the matrix one at a time. Each edge chooses one of the $n \times n$ partitions with probability $p_{i,j}$ respectively. The chosen partition is again subdivided into smaller partitions, and the procedure is repeated k times until we reach a single cell of the $N \times N$ adjacency matrix and place an edge. Instead of $O(N^2)$ iterations, it now results in $O(E)$ iterations, which, often times, is much faster as graphs at this scale are very sparse. Figure 1 illustrates this idea and Algorithm 1 makes it concrete.

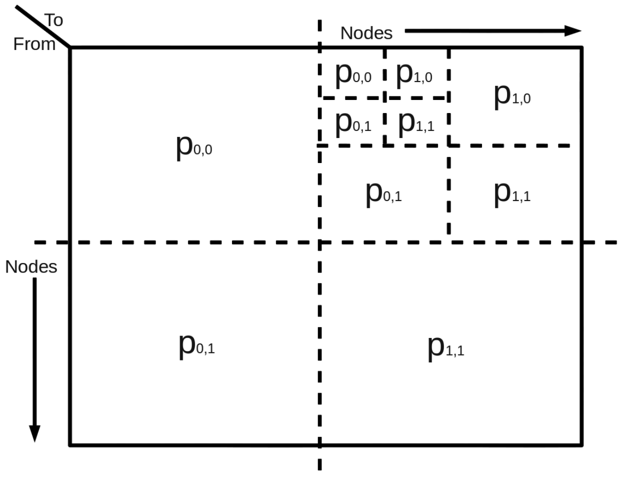


Figure 1: Illustration of SKG. At each of the k steps a sub-region is chosen with probability $p_{i,j}$.

Algorithm 1 SKG

```
for $i = 1, \cdots E$ do
    $u = v = 0$
    for $j = 1, \cdots k$ do
        With probability $p_{rs}$ choose subregion $(r, s)$
        $u = nu + r$; $v = nv + s$
    end for
    Add edge $(u, v)$
end for
```

2.2 Parallelized Stochastic Kronecker Graph

Given the construction of SKG, we can se that this process can be embarrassingly parallelized on multiple processes by letting each workers take generate a subset of the entire set of edges. We call this algorithm Par-SKG. One big assumption we make here is that the edge collisions are very minimal and thus would not drastically affect the final graph in any ways. Given such assumption, the idea is to send to each workers a number of edges that they must create and let each worker independently create edges by themselves. If the algorithm is ran with processors on shared memory, we can avoid collision by checking if the edge already exists. However, if that is not the case, we must either filter out the collision after the processes have ended or run multiple iterations until we achieve the exact number of edges. The formal algorithm is as follows.

Algorithm 2 Parallelized SKG (Par-SKG)

```
for each worker do
  for $i = 1, \cdots \lfloor E/NUM\_WORKERS \rfloor$ do
    $u = v = 0$
    for $j = 1, \cdots k$ do
      With probability $p_{rs}$ choose subregion $(r, s)$
      $u = nu + r$; $v = nv + s$
    end for
    Add edge $(u, v)$
  end for
end for
```

2.3 Poisson Stochastic Kronecker Graph

Here we will discuss the methods for deriving Poisson Stochastic Kronecker Graph (PSKG) from Algorithm 1.

Let $P_k = P \otimes P \otimes \cdots \otimes P$ be the k-th Kronecker power of P. Then we can interpret SKG as generating m edges independently [1] where any edge (u, v) is created with probability $(P_k)_{uv}$. Given this, we can apply Bayes' rule

$$\begin{aligned}\mathbf{P}(\text{edge is } (u,v)) =& (P_k\mathbf{1})_u \frac{(P_k)_{uv}}{(P_k\mathbf{1})_u} \\ =& \mathbf{P}(\text{destination is } u) \times \\ & \mathbf{P}(\text{source is } v | \text{destination is } u).\end{aligned}$$

The decomposition above permits us to choose the source node first and then the destination node rather than simultaneously and we do this with a recursive algorithm to avoid the explicit construction of P_k. Let

$$U = \begin{bmatrix} \sum_{j=0}^{n-1} p_{0,j} \\ \vdots \\ \sum_{j=0}^{n-1} p_{n-1,j} \end{bmatrix} V = \begin{bmatrix} \frac{p_{0,0}}{U_0} & \cdots & \frac{p_{0,n-1}}{U_0} \\ \vdots & \ddots & \vdots \\ \frac{p_{n-1,0}}{U_{n-1}} & \cdots & \frac{p_{n-1,n-1}}{U_{n-1}} \end{bmatrix}$$

[1]The edge generations are not quite independent due to possible collisions of edge creation.

and we arrive at Algorithm 3 which is equivalent to the original formulation of SKG. Here we can

Algorithm 3 Equivalent SKG

```
for $i = 1, \cdots E$ do
  //Select source node $u$
  $u = 0$
  for $j = 1, \cdots k$ do
    With probability $U_r$ choose subregion $r$
    $u = nu + r$
  end for
  //Select destination node $v$
  $v = 0$; $z = u$
  for $j = 1, \cdots k$ do
    $l = \mathrm{mod}(z, n)$
    With probability $V_{ls}$ choose subregion $s$
    $v = nv + s$; $z = z/n$ (integer division)
  end for
  Add edge $(u, v)$
end for
```

see that the source node selection procedure is (approximately) a multinomial random variable with parameters E and $U^{[k]}$, where $U^{[k]}$ is the k-th Kronecker power of U.

Due to the following elementary result [4]we can replace the source node selection procedure, a multinomial random variable, with i.i.d. Poisson random variables.

Lemma. *Let $X_1, \cdots X_s$ be independent Poisson random variables each with mean $\alpha p_1, \cdots \alpha p_s$, where $\alpha > 0$, $p_1, \cdots p_s \geq 0$, and $\sum_{i=1}^{s} p_i = 1$. Then*

$$\mathbf{P}\left(X_1 = x_1, \cdots X_s = x_s \middle| \sum_{i=1}^{s} X_i = m \right) = \frac{m!}{x_1! \cdots x_s!} p_1^{x_1} \cdots p_k^{x_s}$$

i.e. conditioned on the sum $X_1, \cdots X_s$ is distributed as a multinomial distribution.

We are finally ready to state the main algorithm of this paper. Let E be the expected number of total edges while P, U, V, k are defined the same as before.

PSKG will retain all the desired properties of SKG graphs. Specifically, say there is a desired

Algorithm 4 PSKG

```
Scatter $E, P, U, V, k$
for Each vertex $u$ do
  //Determine out-degree of $u$
  $p = 1$; $z = u$
  for $j = 1, \cdots k$ do
    $l = \mathrm{mod}(z, n)$; $p = pU_l$
    $z = z/n$ (integer division)
  end for
  Generate $X \sim \mathrm{Poisson}(Ep)$
  //For each edge determine destination vertex
  for $i = 1, \cdots X$ do
    $v = 0$; $z = u$
    for $j = 1, \cdots k$ do
      $l = \mathrm{mod}(z, n)$
      With probability $V_{ls}$ choose subregion $s$
      $v = nv + s$; $z = z/n$ (integer division)
    end for
    Add edge $(u, v)$
  end for
end for
```

property observed by SKG graphs of all sizes with probability $1-\varepsilon$. Then the Poisson SKG graphs will also have the desired property with probability $1-\varepsilon$ by the following lemma.

Lemma. *Let A be an event that occurs with probability $1-\varepsilon$ for SKG graphs of all sizes. Then A will also hold with probability $1-\varepsilon$ for Poisson SKG graphs as well.*

Proof.

$$\mathbf{P}_{Poisson}(A) = \mathbf{E}\left[\mathbf{P}\left(A \middle| \sum_{i=1}^{k} X_i = m\right)\right]$$

$$= \mathbf{E}\left[\mathbf{P}_{Multinomial}(A|m)\right] > \mathbf{E}[1-\varepsilon] = 1-\varepsilon$$

□

3 Experimental Results

3.1 Graph Patterns

In this section, we first demonstrate that PSKG and SKG generate graphs with essentially the same properties. We assume that Par-SKG and SKG will generate the graphs of same pattern as they both are same algorithms. As SKG models real world networks well [1] the equivalence between PSKG and SKG implies the modeling power of PSKG.

To generate and analyze the SKG and PSKG, we have implemented our own version of SKG and PSKG in python using NetworkX to generate the graphs. There are several standard graph characteristics that are compared the similarity between networks. In this paper, we have compared following patterns: degree distribution, hop plot, scree plot, and network values.These choices are motivated by Leskovec's [1] work.

Degree distribution: The histogram of the nodes' degrees with exponential binning.

Hop plot: Number of reachable pairs $r(h)$ within h hops, as a function of the number of hops h.

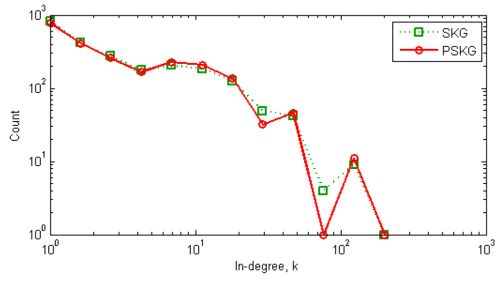
Scree plot: Singular values of the graph adjacency matrix versus their rank.

Network values: Distribution of the principal eigenvector components versus their rank.

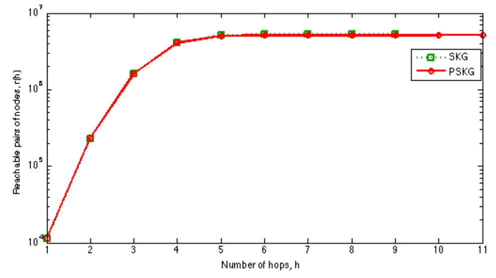
Figure 2 and Figure 3 compares the graph patterns of SKG and PSKG. It is clear that the results are essentially the same.

3.2 Performance Comparisons

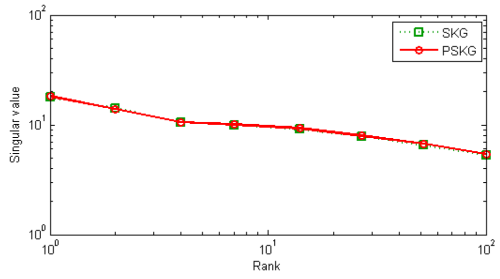
In order display the usefulness of each of the algorithms, we have measured the running time of each of the algorithm. Since, it is very hard to compare the running time of the algorithms on varying platforms due to difference in underlying implementation of these systems, we have implemented a version of the equivalent algorithm on Python with networkX. The algorithms are ran on a macbook pro with 2.66 GHz Intel Core i7 processor with 8 GB 1067 MHz DDR3 RAM. For each of the parallel version, I have used 8 threads, and I varied the number of nodes in the graphs to generate to show the difference in times. The initiator matrix that were used for generation was $p_1 = [[0.987, 0.571], [0.571, 0.049]]$. Finally, the times algorithm for each graph size is averaged over ten runs to minimize the systematic noises. Figure 4 shows the comparison of the running



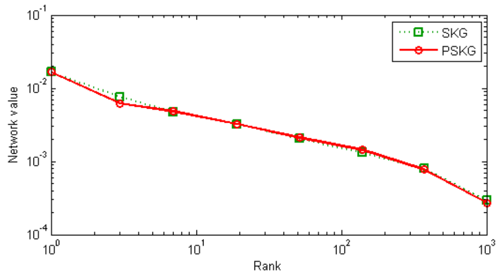
(a) Degree distribution



(b) Hop plot

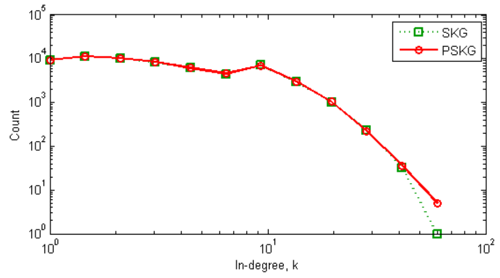


(c) Scree plot

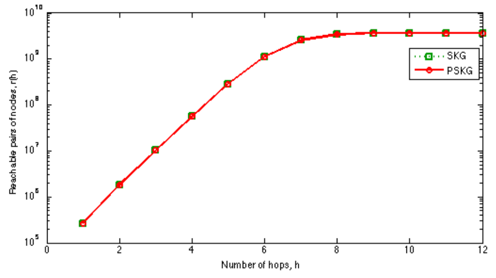


(d) "Network value"

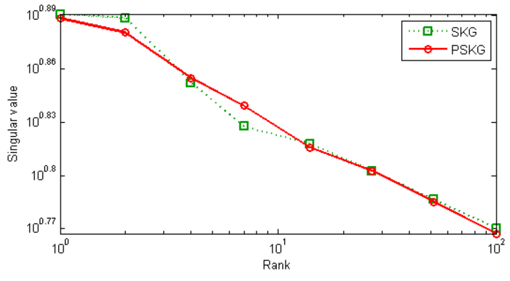
Figure 2: Graph patterns with parameters $n = 2$, $k = 12$, $E = 11400$, and $P = [0.4532, 0.2622; 0.2622, 0.0225]$



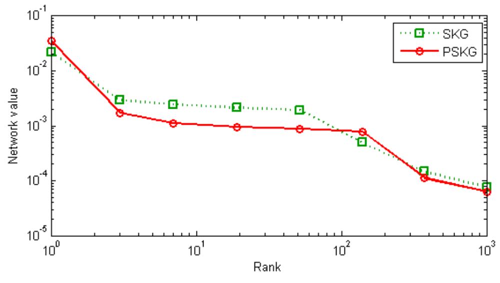
(a) Degree distribution



(b) Hop plot



(c) Scree plot



(d) "Network value"

Figure 3: Graph patterns with parameters $n = 4$, $k = 8$, $E = 263546$, and $P = [\alpha, \alpha, \alpha, \alpha; \alpha, \alpha, \beta, \beta; \alpha, \beta, \alpha, \beta; \alpha, \beta, \beta, \alpha]$ where $\alpha = 0.0861$ and $\beta = 0.0231$. P is the adjacency matrix of a star graph on 4 nodes (center + 3 satellites) with the 1's are replaced with α and the 0's are replaced with β.

time on each of the algorithm on the given machine.

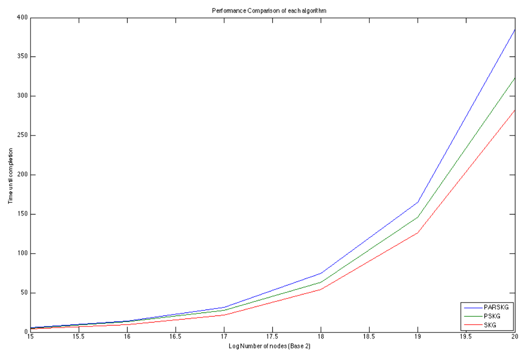
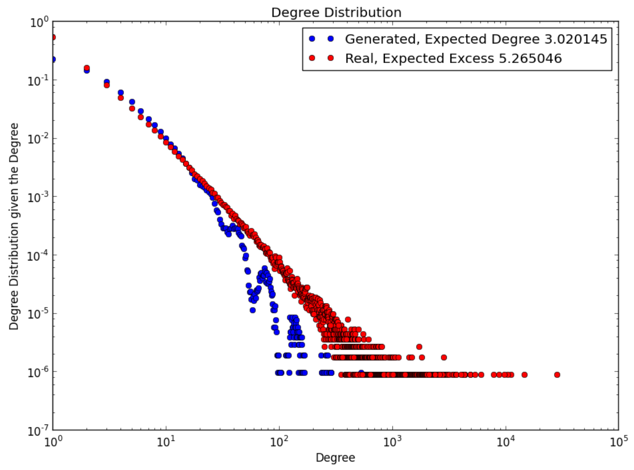


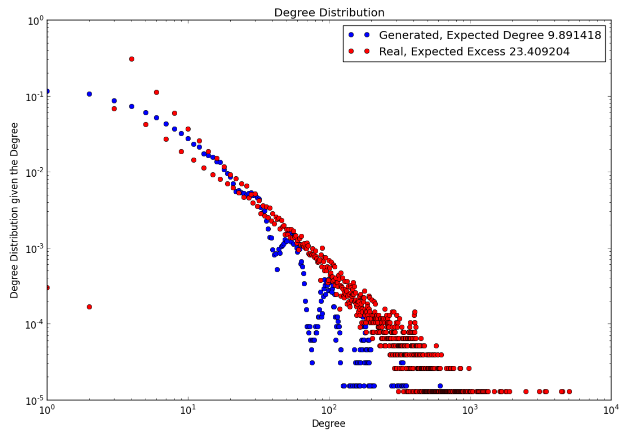
Figure 4: Performance statistics of SKG, ParSKG and PSKG

3.3 Closeness to Real Graphs

In order to compare the random graph to a real graph, we first obtained two datasets, an directed Slashdot social circle graph (77,360 nodes and 905,468 edges) and a undirected Youtube graph (1,134,890 nodes and 2,987,624 edges). In order to make a comparison, I first found the initiator matrix by using Kronecker parameter fitting function in Snap, a network analysis software [1]. The fit we have obtained for the Slashdot data gave the initiator matrix of $[[0.9999, 0.4308], [0.4456, 0.3359]]$ and for the Youtube data, it gave $[[0.8271, 0.4034], [0.5277, 0.2835]]$. Given these parameters, I created the random Kronecker graph using PSKG and compared the degree distribution and clustering coefficient. The results are as follows. First comparing the clustering coefficient of the Youtube graph, the real graph gave a value of 0.08080, whereas the random graph gave a value of 2.9209^{-05}. Next, comparing the clustering coefficients of the Slashdot data, the real graph gave a value of 0.0555, compared to the random graph's value of 0.0015. Using different random graph schema, the value of clustering coefficients did not change much from this value. Next, Figure 5 shows the degree distribution comparison of the real graph and corresponding random graph. We can see that they follow a similar trend, except that the random graph exhibits a staircase pattern as discussed in [1].



(a) Youtube Data



(b) SlashDot Data

Figure 5: Comparison of Degree Distribution of random graph vs real graph

4 Discussion

First, discussing the correctness of PKSG, we can see that it is very close to SKG as given by the plot of different graph statistics. We can see that the degree distribution, hop plot, scree plot and network values are all almost equivalent, making it certain that PKSG and SKG are indeed similar and can be used interchangeably between use cases.

Now, we discuss the performance statistics of

PSKG, PARSKG and PSKG. Before any discussion, this was an experiment to roughly compare the complexity of the algorithms and the results may vary greatly depending on the experimenters setup. First of all, contrary to our expectations, we found that SKG performed the best in our experimental setup. The reason is because the parallelized version of the algorithms are not really ran in parallel as they are implemented in python. It is a known problem that Python threads are serialized by the global interpreter lock, which prevents more than one thread to run at a given instance. If we had used another language such as C++ or Java that had real threading, the results might have been different. Also, these parallelized algorithms are meant to be run on distributed computing system, thus comparing PARSKG and PSKG with SKG is not a good metric in this case. However, the interesting part is comparing between PARSKG and PSKG. We have found that PSKG ran faster than PARSKG on the same setup with same number of threads. There are multiple explanations for this. First is the fact that when creating a new edge, PSKG only takes consideration of the vertices own neighbors whereas PARSKG takes consideration of all the possible edges. Thus, there are less iteration of the initiator matrix to look at when creating the random edges, resulting in a better overall performance. Also, PSKG is less likely to run into edge collisions and it is faster to compare for edge collisions when compare to PARSKG. This is because there are less entries for PSKG's edge list, as it is set of edges for each vertex. Therefore, instead of checking for collisions with entire graph, we have less elements to add to the list of edges, resulting in less hash collisions resulting in better performance. This insight might be a reason why one would use vertex-based BSP implementations for graph creation rather than Map-Reduce implementations.

Finally, we discuss the closeness of PSKG with real graph. There are multiple caveats when analyzing the results. First, the random graphs are created from the initiator matrix given by the Kronecker fit algorithm from [1]. Thus, although the fit might be close, it cannot fit the graph exactly, resulting in some error. Secondly, since the the number of nodes can only be some power of the dimensions of the initiator matrix, it cannot map the number of nodes and edges exactly, resulting in some error. In order to minimize such differences, I have chosen the Kronecker exponent that gives the closest number of edges and nodes, yet it still results in some underpopulation or overpopulation of nodes and edges.

We have introduced a new algorithm called PSKG and compared them with the traditional approaches of SKG and PARSKG. We also have shown that PSKG performs better than PARSKG under our experimental setup and the results are nearly equivalent to that of SKG or PARSKG. One final remark that promotes the uses of PSKG over PARSKG on a distributed setup is that it allows for vertex BSP's to utilize it with ease, without having to reload the data from disk or file if SKG or PARSKG is used. Currently Map-Reduce implementations utilize hard-drive flushes of the data in order to maximize the scalability. However, this results in three problems. First is the reduction in performance due to disk IO. Second is the impossibility to check for edge collisions. Finally, the results must be reloaded onto some other application, which takes more overhead. Our PSKG algorithm avoids all such shortcomings and is a more preferable choice when given that the performance of it is actually better than PARSKG counterparts.

References

[1] Leskovec, Jure and Chakrabarti, Deepayan and Kleinberg, Jon and Faloutsos, Christos and Ghahramani, Zoubin, *"Kronecker Graphs: An Approach to Modeling Networks"*. In: *J. Mach. Learn. Res* 11 (Mar. 2010), pp. 985-1042. ISSN: 1532-4435

[2] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. *R-MAT: A recursive model for graph mining.* In: SDM. 2004.

[3] Grzegorz Malewicz et al. *Pregel: a system for large-scale graph processing*. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. SIGMOD 10. Indianapolis, Indiana, USA: ACM, 2010, pp. 135146. ISBN: 978-1-4503-0032-2. DOI: 10.1145/1807167.1807184.

[4] Brown, Judith,. *An Efficient Two-Stage Procedure for Generating Random Variates from the Multinomial Distribution*. In: *The American Statistician*, pp. 216-219, 1984