Generating synthetic social graphs with Darwini

Sergey Edunov, Dionysios Logothetis, Avery Ching, Maja Kabiljo
Facebook
Email: {edunov, dionysios, aching, majakabiljo}@fb.com

Cheng Wang
University of Houston
Email: cwang35@uh.edu

Abstract—Synthetic graph generators facilitate research in graph algorithms and graph processing systems by providing access to graphs that resemble real social networks while addressing privacy and security concerns. Nevertheless, their practical value lies in their ability to capture important metrics of real graphs, such as degree distribution and clustering properties. Graph generators must also be able to produce such graphs at the scale of real-world industry graphs, that is, hundreds of billions or trillions of edges.

In this paper, we propose *Darwini*, a graph generator that captures a number of core characteristics of real graphs. Importantly, given a source graph, it can reproduce the degree distribution and, unlike existing approaches, the local clustering coefficient distribution. Furthermore, Darwini maintains a number of metrics, such as graph assortativity, eigenvalues, and others. Comparing Darwini with state-of-the-art generative models, we show that it can reproduce these characteristics more accurately. Finally, we provide an open source implementation of Darwini on the vertex-centric Apache Giraph™ model that can generate synthetic graphs with up to 3 trillion edges.

I. INTRODUCTION

The availability of realistic large-scale graph datasets is important for the study of graph algorithms as well as for benchmarking graph processing systems [1], [2]. For instance, graph processing frameworks such as [3]–[5] have been evaluated on social graphs with up to 10B edges. Unfortunately, the applicability of this work toward industry graphs is limited due to significant differences in both scale and community structure. For example, Facebook has 2.07B active users [6] with more than 400B edges [7], while, in 2008, Google found the web graph to contain more than 1 trillion unique URLs.

At the same time, accessing such industry datasets is challenging for a variety of reasons. For instance, these organizations must respect user privacy and security [8]. Even when data is public (e.g. web data), the significant time and resources required to collect and aggregate this information makes this task difficult for most researchers.

Synthetic graph generators provide a way to address these limitations. They allow organizations to share synthetic versions of their data while protecting user privacy. Further, they enable researchers to reproduce large graph datasets based on published graph metrics. For example, given just the degree distribution of a social network, a generator may produce a synthetic graph with a similar distribution without access to the actual social graph.

Nevertheless, the value of graph generators lies in their ability to capture important metrics of real graphs, such as degree distribution, graph diameter and others. For instance,

the accuracy of application simulations depends on the fidelity of such metrics [9]. Additionally, since graph properties, like degree skew, often guide the design of graph processing systems [3], synthetic data must represent realistic graphs. Importantly, since system artifacts or bottlenecks may manifest only on large data, graph generators must be able to produce such graphs at scale.

While existing graph generation models capture several properties of real graphs, they fall short in at least one of three important aspects. First, they may restrict the model to specific degree distributions. The Kronecker model [10], one of the most popular generative models, generates only power-law graphs. However, several real graphs behave differently in practice [11], [12]. For example, the Facebook social network caps the number of friends, invalidating the power-law property [12]. Such inaccuracy in the degree distribution limits the utility of synthetic graphs when benchmarking systems, like Pregel [13] and GraphX [4], since the degree distribution impacts performance by means of the compute and network load balance.

Second, current approaches do not capture local node clustering properties, such as the clustering coefficient [14] distribution, at a fine granularity [9], [10], [15]. The BTER model improves upon Kronecker graphs by allowing non-power law distributions, but assumes that same-degree nodes also have the same clustering coefficient, which does not hold in practice [9]. An inaccurate clustering coefficient distribution may impact, for instance, the partitioning of graph data and, consequently, the observed performance of systems that distribute graph computations across machines [16].

Third, current techniques may not be practical to use. For example, existing models may require manual tuning of several parameters, and misconfiguration may lead to inaccurate output graphs. Alternatively, they may require model fitting prior to graph generation, which, for large graphs, incurs high overhead and may not scale [9].

In this paper, we propose Darwini [1], an algorithm that takes as input explicitly specified node-degree and clustering coefficient distributions and generates synthetic graphs that accurately match these distributions. Darwini groups synthetic vertices in buckets and iteratively adds edges within and across buckets using a novel heuristic that controls both these distributions at a fine granularity. Darwini further captures a number of important metrics observed in real graphs. Notably,

---

[1] *Caerostris Darwini* is a spider that weaves one of the largest known webs.

and unlike other methods, it captures graph-assortativity [2]. We show that Darwini can reproduce different types of input degree and clustering coefficient distributions, outperforming state-of-the-art algorithms in terms of accuracy.

Consequently, when using Darwini to benchmark graph processing systems, this accuracy is reflected on the observed system performance as well; processing time on synthetic graphs is more representative of the processing time on real graphs. This gives us more confidence in projecting the performance of our systems on scaled-up, realistic versions of a reference graph.

We designed Darwini to be parallelizable, and scale to large output graphs. In fact, we provide an open source distributed implementation of Darwini [17] in the vertex-centric Apache Giraph model [18]. Darwini introduces a method for decomposing graph generation to smaller tasks to allow the generation of graphs that may not even fit in the available main memory. Overall, Darwini scales linearly on the size of the output graph. Using our implementation, we are able to generate synthetic graphs with up to 3 trillion edges.

Importantly, Darwini requires as input only the degree distribution and per degree clustering coefficient distribution of an input source graph. These distributions can be computed in a scalable manner on very large graphs, making our approach practical. Consequently, in terms of privacy, these are the only metrics of the source graph that Darwini reveals.

This paper makes the following contributions:

- We introduce Darwini, a graph generating algorithm that can reproduce both the degree and the clustering coefficient distributions of real social graphs with trillions of edges. To the best of our knowledge, this is the first algorithm that achieves this validation.
- We provide a distributed implementation of the Darwini algorithm on the Apache Giraph model that can generate synthetic graphs with trillions of edges. Further, we extend Darwini with a technique that can generate graphs that do not fit in the available memory.
- We provide a thorough evaluation of Darwini. We show that it can reproduce a number of important metrics on different real graphs, outperforming state-of-the-art techniques in terms of accuracy. Further, we benchmark the Apache Giraph system with different applications on synthetic graphs, and show that the observed system performance is close to that observed on the real graph. Finally, we show that our distributed implementation scales linearly on the size of the generated graph.

The remaining of the paper is structured as follows. In Section II, we describe the Darwini algorithm in detail, while in Section III we outline the distributed implementation. Section IV contains a thorough evaluation. In Section V, we give an overview of related work. In Section VI, we conclude and discuss future work in this area.
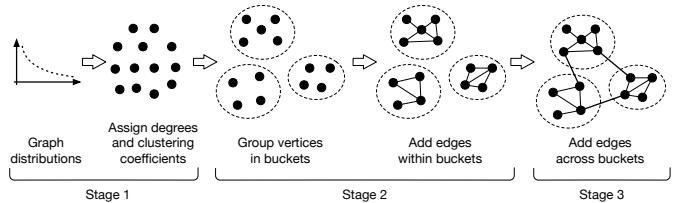
---

[2]The preference of a node to connect with other nodes with similar degree.

Fig. 1. The different stages of Darwini.

II. ALGORITHM

Darwini attempts to generate synthetic graphs with explicitly specified degree and clustering coefficient distributions as input. Typically, we obtain these distributions by measuring a reference graph, like a social network, and then use Darwini to generate a synthetic graph with similar distributions, potentially of a different size. This approach differs from growth models, such as preferential attachment [19], that attempt to model the underlying natural process under which real-world graphs grow. As we show in Section IV, this allows Darwini to re-produce artifacts in source graphs that growth models cannot capture.

Connecting vertices in a way that meets these distributions is challenging due to the scale of the graphs and the number of possible connections. To address this, Darwini uses a combination of edge generation approaches that iteratively add edges while preserving the local clustering coefficient and degree distributions of the original graph. At the same time, it does so in a block fashion. It first builds smaller communities that it then interconnects in a larger graph. By design, the algorithm can be executed in a distributed fashion, making it possible to scale to large data sets.

The Darwini algorithm consists of 3 successive stages, illustrated in Figure 1. In the first stage (Section II-A), Darwini generates a set of unconnected vertices and assigns a *target* degree and clustering coefficient to each vertex. This is the degree and clustering coefficient that each vertex should eventually have so that the output graph matches the desired distribution. In the second stage (Section II-B), Darwini groups vertices into smaller communities and creates edges within the communities, approximating the target degrees and clustering coefficients. Finally, in the third stage (Section II-C), Darwini connects vertices across communities to match the actual target distributions. The heuristic for constructing communities in the second stage is essential for preserving the local clustering coefficient and degree distributions. In the remaining of this section, we describe each stage in detail.

A. Assigning target degrees and clustering coefficients

In the first stage, Darwini assigns a target degree and clustering coefficient to every vertex of the output graph. Assuming that the desired output graph has N vertices, where N is user-defined, we will use $G = (V, E)$ to denote the synthetic output graph, $v_i \in V, 0 \leq i < N$ to denote its vertices, and $(v_i, v_j) \in E, 0 \leq i, j < N$ to denote its edges.

Each vertex v_i will have a target degree d_i and a target clustering coefficient c_i.

The inputs to this stage are (i) F_{deg}, the degree distribution across the entire source graph, (ii) $F_{cc}(d)$, the clustering coefficient distribution among vertices with degree d, for all unique values of d. Such distributions can be measured even on large graphs using random sampling.

Subsequently, for every vertex $v_i \in V$, Darwini first draws d_i from the F_{deg} distribution. After it has picked d_i for vertex v_i, Darwini draws the target clustering coefficient c_i from the corresponding $F_{cc}(d_i)$ distribution. Note that, unlike approaches like BTER [20], Darwini captures the clustering coefficient distribution at such fine granularity.

B. Connecting vertices in communities

Connecting vertices in a way that matches both the target degrees and clustering coefficients directly is challenging due to the number of possible ways to connect vertices. For instance, the BTER model addresses this by making the assumption that vertices with the same degree also have the same clustering coefficient [20]. Given a target degree and a target clustering coefficient, it is then possible to add random edges in a group of vertices in a way that satisfies both. However, this assumption does not hold in practice, resulting in inaccurate clustering coefficient distributions.

Instead of matching the vertex degree and clustering coefficient directly, Darwini first tries to match the number of triangles each vertex should belong to in the final output graph. This is key in eventually allowing Darwini to re-produce both the degree and clustering coefficient distributions accurately. To understand the intuition behind the technique, first consider the definition of the clustering coefficient of a vertex v_i in an undirected graph:

$$c_i = \frac{2N_{\triangle,i}}{d_i(d_i - 1)} \qquad (1)$$

where $N_{\triangle,i}$ is the number of triangles [3] v_i participates in. Given this definition, assume a vertex v_i that is connected with other vertices in such a way that it already participates in $N_{\triangle,i}$ triangles, but has less edges than its target degree d_i. We can then reach the target degree d_i by connecting v_i to vertices with which it cannot form any new triangles. This way, $N_{\triangle,i}$ is not affected by the additional edges, and by matching d_i, we are indirectly matching the target clustering coefficient c_i as well.

Therefore, the objective of Darwini in this stage is to ensure that: (i) each vertex participates in approximately the number of triangles it should eventually belong to, and (ii) for each vertex there are enough other vertices in the graph with which it can connect without forming triangles.

To achieve these objectives, Darwini first groups vertices into communities, or *buckets*, according to the number of triangles $N_{\triangle,i}$ they must eventually belong to, computed from Equation 1. In the second stage of Figure 1, all vertices within

[3]A vertex v_i participates in a triangle with vertices v_j and v_k if $(v_i, v_j) \in E$, $(v_i, v_k) \in E$ and $(v_j, v_k) \in E$.

the same bucket have the same $N_{\triangle,i}$. Subsequently, it adds random edges within a bucket with a fixed probability P_e, like in the Erdös-Rényi model. This grouping and the random edge addition process are both tied to the above objective; by picking probability P_e accordingly we can, in expectation, create the desired number of triangles for all vertices in the bucket.

To understand this process, consider a bucket with n vertices that we connect randomly, with each edge created with a probability P_e. Due to the independence of edge additions, the probability of any combination of three vertices in the bucket forming a triangle is $P_{\triangle} = P_e^3$. Since for each vertex there are $N_{\triangle} = (n-1)(n-2)/2$ possible triangles in which it can participate, the expected number of triangles for a vertex is:

$$\hat{N}_{\triangle} = P_{\triangle} \cdot N_{\triangle} = P_e^3 \frac{(n-1)(n-2)}{2} \qquad (2)$$

Based on Equation 2, we can construct a bucket with a desired expected number of triangles per vertex by setting the size n of the bucket and the probability P_e appropriately.

Notice that there are different combinations of n and P_e that can achieve the desired expected number of triangles for a bucket B. The choice of the values must satisfy two conditions. First, a bucket must have enough vertices to accommodate the expected number of triangles. Assuming that every vertex participates in the expected number of triangles, that is, $N_{\triangle,i} = \hat{N}_{\triangle}$, then from Equations 1 and 2 and since $P_e < 1$, we get that:

$$n \geq \sqrt{c_i d_i (d_i - 1)} = n_{B,min}, \forall i \in B \qquad (3)$$

Second, while in this stage Darwini tries to create the desired number of triangles, it must ensure that no vertex exceeds its target degree. Otherwise, in the second stage, we will not be able to correct its clustering coefficient by connecting it to other edges. To prevent this from happening, we limit the value of n as follows. Since within a bucket B with n vertices each vertex can have at most $n-1$ edges, we require:

$$n \leq \min_{i \in B}(d_i) + 1 = n_{B,max} \qquad (4)$$

This way, no vertex $v_i, i \in B$ can have more than d_i edges.

We can now calculate the probability P_e for a bucket B based on Equation 2, setting n within these bounds. In fact, Darwini picks the lower bound $n_{B,min}$ as the size of a bucket, therefore:

$$P_e = \sqrt[3]{\frac{2\hat{N}_{\triangle,B}}{(n_{B,min} - 1)(n_{B,min} - 2)}} \qquad (5)$$

Here, $\hat{N}_{\triangle,B}$ represents the expected number of triangles per vertex, common for all vertices in bucket B.

Using these values, Darwini implements the grouping of vertices in buckets in three successive phases, described in detail by Algorithms 1, 2 and 3. In the following, we explain all the steps, referring to the detailed algorithm descriptions where necessary.

Algorithm 1 Group vertices into buckets

1: **Input:** Target degrees d_i, $0 \leq i \leq N - 1$
2: **Input:** Target clustering coefficients c_i, $0 \leq i \leq N - 1$
3: $S \leftarrow \{\}$ ▷ Initialize set of buckets S
4: **for** $i = 0$ to $N - 1$
5:      $N_{\triangle,i} \leftarrow c_i * d_i * (d_i - 1)/2$
6:      bucket $\leftarrow$ selectBucket(S, $N_{\triangle,i}$)
        ▷ Chooses non-full bucket or adds new bucket in S
7:      bucket.add(i)
8:      **if** bucket.size $> \min_{j \in bucket}(d_j) + 1$
9:         bucket.full()
10: **return** S

Grouping vertices into buckets. Darwini starts with the execution of Algorithm 1. It groups vertices in buckets, based on the value of $N_{\triangle,i}$, as described above (lines 4-9).

As Darwini adds vertices one by one to the buckets based on the value of $N_{\triangle,i}$, more than $n_{B,max}$ vertices may fall in the same bucket. To handle this, the *selectBucket* procedure (line 6) searches for a non-full bucket with the same $N_{\triangle,}$ or allocates a new bucket. Subsequent vertices with the same $N_{\triangle,i}$ are added to the new bucket. Note that after Darwini adds a vertex to bucket B, the value $n_{B,max}$ is recomputed (line 8) to reflect the degree of the newly added vertex and ensure that a bucket never exceeds the allowed size. If a bucket B reaches $n_{B,max}$, Darwini labels it as full (lines 8-9).

Algorithm 2 Merging incomplete buckets

1: **Input:** Target degrees d_i, $0 \leq i \leq N - 1$
2: **Input:** Set of buckets S ▷ Output of Algorithm 1
3: $S_u \leftarrow \{b | b \in S, b.size < n_{b,min}\}$
                         ▷ Buckets with few vertices
4: $S \leftarrow S - S_u$
5: sort(S_u) ▷ Sort in order of $N_{\triangle}$ of each bucket
6: bucket $\leftarrow$ emptyBucket()
7: S.add(bucket)
8: **for all** b in S_u
9:      bucket.merge(b)
10:      **if** bucket.size $> \min_{j \in bucket}(d_j) + 1$
11:         bucket.full()
12:         bucket $\leftarrow$ emptyBucket()
13:         S.add(bucket)
14: bucket.full()
15: **return** B

Merging incomplete buckets. After vertex grouping finishes, some buckets may not have enough vertices to create the necessary number of triangles based on (3). To address this, Darwini merges small buckets into bigger ones. This is implemented in Algorithm 2. Notice that merging causes vertices with a different value of $N_{\triangle,i}$ to be placed in the same bucket. As a result there is no single value for P_e that will approximate $N_{\triangle,i}$ well for all vertices in a merged bucket. Eventually, this may prevent vertices from approximating well the target clustering coefficient. Nevertheless, we have

found empirically that this offsets the inaccuracy caused by incomplete buckets.

Besides, Darwini merges buckets in a way that mitigates this effect. After obtaining all incomplete buckets (lines 3-4), it orders them according to their $N_{\triangle,i}$ value (line 5). Subsequently, it merges buckets with close $N_{\triangle,i}$ values (lines 8-13). When it creates a merged bucket with the maximum allowed size, it marks it as full and allocates a new one (lines 10-13). This ensures that the expected number of triangles for each vertex in a bucket is closer than in a random assignment.

Algorithm 3 Create random edges within buckets

1: **Input:** Target degrees d_i, $0 \leq i \leq N - 1$
2: **Input:** Target clustering coefficients c_i, $0 \leq i \leq N - 1$
3: **Input:** Set of buckets S ▷ Output of Algorithm 2
4: **for** $b \in S$
5:      $P_e = \sqrt[3]{2\hat{N}_{\triangle,b}/((n_{b,min} - 1)(n_{b,min} - 2))}$
6:      **for** $v_i \in b$
7:         **for** $v_j \in b$, $j < i$
8:            **if** random() $< P_e$
9:               $addEdge(v_i, v_j)$

Adding edges. After grouping the vertices into buckets, Darwini adds random edges in each bucket according to the Erdös-Rényi model, to create the expected number of triangles in the bucket. Algorithm 3 describes this process. Darwini sets the edge probability P_e based on Equation 5 (line 5).

At the end of this stage, Darwini vertices participate, in expectation, in the desired number of triangles, but do not meet their target degrees and clustering coefficients. In fact, for every vertex v_i, its current degree $d_{curr,i}$ should be less than the target degree d_i, therefore the clustering coefficient should be higher than its target value. In the following section, we describe how Darwini corrects this.

C. Interconnecting communities

In this step, Darwini attempts to add the residual degree $d_i - d_{curr,i}$ for each vertex while leaving the number of triangles it participates in intact. Darwini achieves this by connecting vertices that belong to different buckets, picking randomly from the entire graph. Intuitively, this increases the degree of each vertex, and since the connections are now random across the entire graph, they are unlikely to contribute to the number of triangles for any vertex. By iteratively adding such random edges, Darwini gradually meets both the degree and the target clustering coefficient of a vertex. Darwini implements this stage with Algorithm 4.

Algorithm 4 iteratively adds edges and runs until it has added all the remaining edges, or it has reached a maximum number of iterations (line 6). In every iteration, Algorithm 4 first makes a pass on every vertex (line 7). If a vertex has not met its target degree yet (line 8), it randomly picks a candidate vertex from the entire graph to connect to (line 9). If by connecting to the candidate vertex, the candidate does not exceed its target degree (line 10), then Darwini adds an edge between the two vertices (line 11).

Algorithm 4 Create random edges across buckets

1: **Input:** Current degrees $d_{curr,i}$, $0 \leq i \leq N - 1$
2: **Input:** Target degrees d_i, $0 \leq i \leq N - 1$
3: **Input:** Maximum number of iterations $iter_{max}$
4: $r_e = \sum_{i=0}^{N} d_i - d_{curr,i}$ $\quad\quad$ ▷ Remaining edges to add
5: $iter := 0$
6: **while** $r_e > 0$ **&&** $iter < iter_{max}$
7: $\quad$ **for** $i = 0$ to $N - 1$
8: $\quad\quad$ **if** $d_{curr,i} < d_i$
9: $\quad\quad\quad$ $v_j \leftarrow selectRandom(V)$
10: $\quad\quad\quad$ **if** $d_{curr,j} < d_j$
11: $\quad\quad\quad\quad$ $addEdge(v_i, v_j)$
12: $\quad\quad\quad\quad$ $r_e = r_e - 1$
13: $\quad$ $n_g = 2^{iter+1}$ $\quad\quad\quad\quad\quad$ ▷ Group size
14: $\quad$ $G \leftarrow \text{shuffle}(V, n_g)$ $\quad$ ▷ Shuffle vertices to groups
15: $\quad$ **for** $g \in G$
16: $\quad\quad$ **for** $v_i, v_j \in g, i < j, d_{curr,i} < d_i, d_{curr,j} < d_j$
17: $\quad\quad\quad$ $p = \frac{|d_i - d_j|}{d_i + d_j}$
18: $\quad\quad\quad$ **if** $random() > p$
19: $\quad\quad\quad\quad$ $addEdge(v_i, v_j)$
20: $\quad\quad\quad\quad$ $r_e = r_e - 1$
21: $\quad$ $iter = iter + 1$
22: **end while**

Satisfying high degree vertices. The random selection in line 9 of Algorithm 4 allows us to connect a vertex with another candidate vertex without having to search the entire vertex set for the candidate. This eliminates a significant overhead, allowing Darwini to scale to large graphs.

However, recall that in social networks the majority of the vertices have low degrees. As Darwini goes through all the vertices and for each vertex it picks uniformly a candidate to connect with an edge, the selected candidates are most likely low-degree vertices. As a result, during this process, low-degree vertices will reach their target degree quickly. At the same time, it becomes harder to find destinations for the high-degree vertices that are left. This results in inaccuracy toward the higher end of the degree distribution in the generated graph. This problem manifests in BTER as well, as reported in [20] and verified in our evaluation too.

To address this, we augment the random selection with a process that allows Darwini to find candidates for high-degree vertices while still avoiding a search on the entire graph. The details are described in lines 13 - 20 of Algorithm 4. If there are remaining connections to be added, Darwini splits vertices in small random groups (line 14) and restricts the search for candidates within each group (line 16). After each iteration and only if there are still edges that must be added, Darwini increases the size of the groups exponentially (line 13), expanding the search space. Since adding edges within a group requires information about vertices in the group only, Darwini can parallelize the search.

The random shuffling ensures that Darwini does not increase the number of triangles in the graph by connecting vertices

within a group. More specifically, the shuffling procedure finds those vertices that have not still met their target degree and randomly partitions them to a set of groups of a specified size. Within such group, every pair of vertices is a candidate for adding an edge.

Maintaining degree correlation. Darwini takes into consideration the observation that in social networks, there is a positive correlation between the degree of a node and the degrees of the neighbors of the node [12]. Therefore, aside from the clustering coefficient, Darwini also attempts to maintain this property.

During this stage, Darwini enforces this by randomizing the edge creation process and ensuring that the probability of creating an edge between vertices with similar degrees is higher than the probability of creating an edge between vertices with very different degrees (line 18). As we show in Section IV, this helps maintain a good joint-degree distribution as well.

Algorithm 4 ensures this by adjusting the probability of an edge creation depending on how similar the degrees of the two candidate vertices are (line 17). Darwini sets this probability to be equal to $(|d[i] - d[j]|)/(d[i] + d[j])$. While there are different ways to set the probability, we have found that this works well in practice.

III. IMPLEMENTATION

We have implemented Darwini on top of the Apache Giraph vertex-centric programming model [7]. In this section, we give an outline of the implementation of each algorithm described in Section II. The implementation is available as open source [17].

A. Graph generation

In the vertex-centric model, a vertex is the basic abstraction and the unit of computation. A vertex has a unique ID, edges defined by the target vertex ID, and a value used to store computational state. Vertices can also communicate with each other through messages. Inside the Apache Giraph engine, vertices are in-memory objects distributed across a compute cluster. Darwini maps each vertex of the output graph to a Giraph vertex.

1) Connecting vertices in buckets: Darwini begins by generating N vertices on the fly and assigning vertex IDs in the range $[0, N)$, where N is the desired size of the graph. Darwini initializes the state of each vertex with a target degree and clustering coefficient. These are drawn from the distributions computed on the source graph. At this phase, vertices have no edges and they are not assigned to any bucket yet.

The next step is to assign vertices to buckets and potentially merge buckets as per Algorithms 1 and 2. Darwini leverages the Giraph *aggregation* and *master computation* interfaces to collect information from every vertex and process it at a centralized master worker. At this stage, every vertex sends a triple containing its ID, its target degree and its target clustering coefficient to the master worker. After collecting these triples, the master worker calculates a vertex-to-bucket assignment for

each vertex, executing Algorithms 1 and 2. Darwini stores the resulting mapping in a list $L = (b_1, b_2, ..., b_N)$, where the value $L(i)$ is the ID of the bucket that vertex i belongs to. Note that buckets obtain the same ID as the ID of the first vertex to be assigned to this bucket. We call this vertex the bucket *leader* and use it for coordination among the vertices of the same bucket. Once the master finishes, Darwini broadcasts the mapping to all worker machines.

Next, Darwini implements the random edge creation within a bucket described in Algorithm 3. Notice that while every vertex can independently create random edges to other vertices in the bucket, we need to ensure that if a vertex v_i adds an edge to vertex v_j, then v_j also gets updated with an edge to v_i since the output graph is undirected. To ensure this consistency, in this step, every vertex sends its own ID to the bucket leader. The bucket leader then decides which edges must be created by running Algorithm 3. The leader vertex uses the Apache Giraph *graph mutation* API to create new edges.

2) Connecting vertices across buckets: In the next step, Darwini creates edges across buckets, implementing Algorithm 4. Unlike the implementation of Algorithm 3, a vertex can now pick a destination across the entire graph. In fact, each vertex sends an edge creation request to a random destination vertex ID. Since the range of IDs is known, vertices pick an ID uniformly in this range. Once the destination vertex receives the request message, if it has a non-zero residual node degree, it accepts the request. It adds the edge locally and sends an *edge confirmation message* back to the sending vertex. At this point, the sending vertex can also add this edge.

Recall that Algorithm 4 also intends to find connections for high degree vertices. To implement this step, we use the same concept of bucket leaders as with the implementation of Algorithm 3. Leader vertices now correspond to the groups calculated in Algorithm 4 (line 14). Note that since the range of vertex IDs and the number of groups n_g in each iteration is known, we pick as leaders those vertices with an ID that is a multiple of n_g. This logic is encoded in the vertex computation during this phase. This way, in each iteration, every vertex picks a random vertex leader and sends its target degree and current degree. The vertex leader then executes the logic described in lines 16 - 20 of Algorithm 4.

B. Scaling beyond cluster capability

While the Darwini implementation is parallelizable, its ability to generate large synthetic graphs with a processing system, like Giraph, is limited by the available main memory. However, our goal is to be able to generate graphs bigger than what our current infrastructure can hold in memory. This enables us to stress test our existing infrastructure and predict performance based on projected data growth rates. It also allows us to evaluate new out-of-core processing techniques [21] for handling data sets that do not fit in memory at scale.

To address this, Darwini leverages the observation that in real social networks, users typically belong in large communities that are relatively sparsely connected with each other. Communities defined by the user country of origin make

such an example. For instance, it has been estimated in [12] that 84% of the total number of edges are within the communities defined by the user country. These communities contain a number of vertices that is much bigger than what makes a bucket in Darwini; they may contain hundreds of millions of vertices. We call these large vertex groupings *super-communities*.

Once these super-communities are identified on the source graph, we first run Darwini for each super-community individually, generating the corresponding synthetic version. Each such task is typically small enough to fit in the available main memory. Here, we repeat the same steps described in Section II; we measure the degree and clustering coefficient distribution of each super-community and then use Algorithms 1, 2 and 3 to generate the synthetic super-community.

Next, we need a way to connect vertices across the super-communities. As with connecting vertices across buckets, we can still connect edges in a random fashion. However, we must implement this in a way that does not require loading the entire graph in memory. Notice that to construct these edges, we do not need to load the graph structure of each super-community. For each vertex, we only need to load the super-community that the vertex belongs to and its residual degree. From then on, we essentially repeat the first part of Algorithm 4 (lines 4 - 12). Each vertex picks a random destination across the graph and connects with it only if the destination does not exceed its target degree.

This way of interconnecting communities reduces the required amount of memory by orders of magnitude, allowing us to generate graphs with several trillions of edges. This technique has allowed us to create a synthetic graph with 3 trillion edges despite the fact that the compute cluster we used does not fit such a graph in memory. The computation took 18 hours, 2.5 hours to generate each of the 6 super-communities and 3 hours to generate remaining edges.

IV. EVALUATION

In this section, we evaluate different aspects of our algorithm. First, we measure the ability of the algorithm to accurately capture a number of important graph metrics, and compare our approach with state-of-the-art generative models. Second, we measure the impact of this accuracy on the observed system performance when benchmarking graph processing systems. Finally, we evaluate the scalability of the algorithm and measure the computational overhead of our implementation.

A. Graph metrics

We start by measuring how accurately our algorithm reproduces a number of graph metrics, compared with the input source graph. There is a variety of metrics used to characterize graphs. Here we focus on degree distribution, local clustering coefficient, joint-degree distribution and diameter as they are commonly used to characterize the structure of a graph. We also measure the PageRank distribution, Eigenvalues as higher-level metrics. You can find an evalution on more metrics in

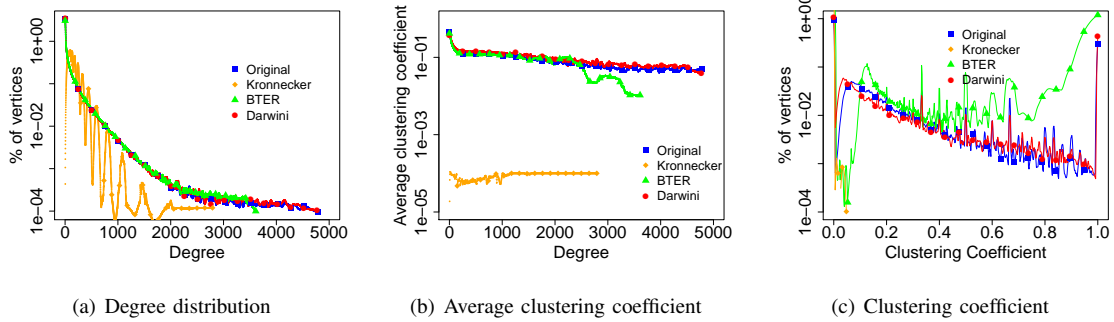(a) Degree distribution (b) Average clustering coefficient (c) Clustering coefficient

Fig. 2. Comparing Darwini, Kronecker and BTER with different graph metrics on the Facebook subgraph. Darwini is more accurate in all metrics.

[22]. In our study, we compared against different models, using a variety of input graphs.

1) Degree distribution: Here, we measure how accurately Darwini reproduces the degree distribution, compared with other techniques. We first evaluate the algorithm using a portion of the Facebook social network as the source graph. Specifically, we capture a subgraph of the Facebook social graph that represents a specific geographic region with approximately 3 million vertices and 700 million edges [4]. We compare Darwini with the BTER and Kronecker models as they are the only models we could evaluate for a graph of this size.

In Figure 2(a), we compare the degree distribution achieved by the different models with that of the original graph. First, notice that the Kronecker model fails to re-produce the degree distribution, as the Facebook graph does not follow the power-law model. Even though BTER provides a better approximation of the degree distribution than Kronecker, it fails to create high-degree vertices. As the algorithm tries to connect high-degree nodes to achieve the right clustering coefficient, it fails to find enough candidates. Darwini, instead, produces a degree distribution that is close to the original for all values of node degree.
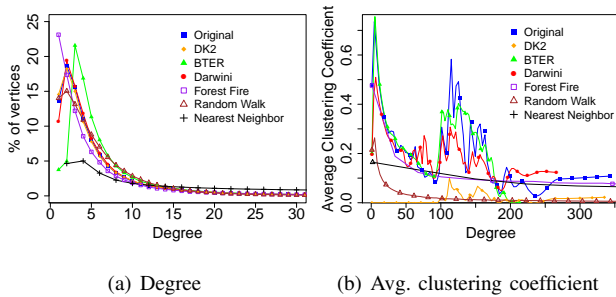

(a) Degree (b) Avg. clustering coefficient

Fig. 3. Comparison with several models on the DBLP graph.

Next, we repeat the same experiment on the DBLP co-authorship graph [23]. Due to the more manageable size of the DBLP graph, we were able to fit and generate all the models described in [9] using their publicly available implementation [24]. Here, we evaluate the best performing

---

⁴For confidentiality reasons we cannot provide more information on the graph.
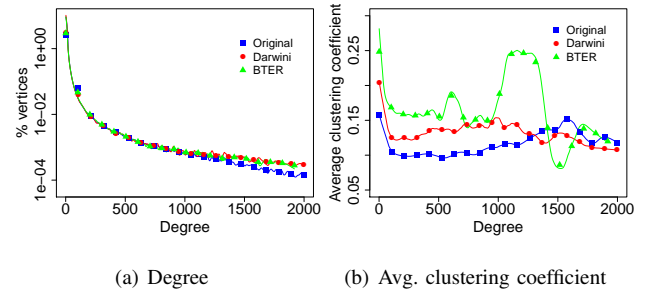

(a) Degree (b) Avg. clustering coefficient

Fig. 4. Comparing Darwini and BTER on the Twitter graph.

Graph	Degree	Clustering Coefficient
BTER	0.21	0.64
Darwini	**0.007**	**0.19**
DK-2	**0.002**	6.04
Forest Fire	0.041	**0.27**
Random Walk	0.039	1.11
Nearest Neighbor	6.04	9.83

TABLE I

KL-DIVERGENCE OF DEGREE AND CLUSTERING COEFFICIENT DISTRIBUTIONS FOR THE DBLP GRAPH.

models among them, namely Nearest Neighbors [25], Random Walk [25], dK-2 [26] and Forest Fire [27].

In Figure 3(a), we plot the actual distribution and in Table I we measure the *Kullback-Leibler* (KL) divergence between the source and the generated distributions for the DBLP graph. Consistent with the results of [26], dK-2 performs the best among this set of models. Nearest Neighbors, one of the best performing models measured in [26], here tends to produce less low-degree vertices than expected. BTER exhibits the same problem, failing to create high-degree vertices. Notice that Darwini exhibits this problem too for this graph, but to a lesser extent. Overall, Darwini produces the second best degree distribution among all in terms of the KL-divergence.

We perform the same measurement on the Twitter follower graph [28]. Here, we compare Darwini and BTER. We omit Kronecker as it did not perform well. Figure 4 shows the results. Both approaches produce a similar degree distribution, though they produce more high degree nodes than the original distribution. However, Darwini produces a clustering coefficient distribution that is closer to the original graph than
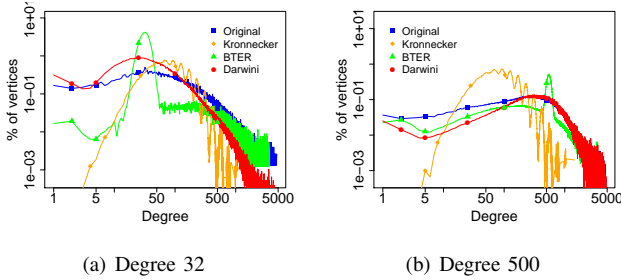
(a) Degree 32   (b) Degree 500

Fig. 5. Joint-degree distribution of the source Facebook graph and the Darwini, BTER and Kronecker graphs.

Distribution	Kronecker	BTER	Darwini
Degree	3.82	0.02	0.0014
Joint Degree, d=5	N/A	0.57	0.11
Joint Degree, d=32	0.48	0.27	0.17
Joint Degree, d=500	1.56	0.34	0.012

TABLE II
KL-DIVERGENCE BETWEEN THE ORIGINAL FACEBOOK GRAPH AND THE GENERATED GRAPH DISTRIBUTIONS.

Graph	Original	Darwini	BTER	Kronecker
Diameter	4.42	4.39	4.41	3.95

TABLE III
GRAPH DIAMETER OF THE ORIGINAL AND THE GENERATED GRAPHS.

BTER.

2) Clustering coefficient distribution: Here, we use the same graphs as above to compare the accuracy of the generated clustering coefficient distribution. First, we measure the average clustering coefficient as a function of the vertex degree for the different models. We show the result for the Facebook graph in Figure 2(b).

Kronecker underestimates the per degree average clustering coefficient by up to 4 orders of magnitude. BTER performs better than Kronecker as it attempts to produce a graph with high average clustering coefficient. Even so, the clustering coefficient diverges significantly for high-degree nodes. Specifically, for nodes with degree higher than 2500, the clustering coefficient may be off by an order of magnitude. Again, BTER cannot produce vertices with high degrees. Instead, for Darwini the average clustering coefficient follows closely the source distribution across the entire spectrum of degrees.

Figure 3(b) compares the per degree average clustering coefficient between Darwini and the rest of the models on the DBLP graph. While in terms of degree distribution the other models produced good results, most of the models underestimate the average clustering coefficient. Only BTER can capture the average clustering coefficient. Still, Darwini outperforms BTER especially for high-degree vertices. Interestingly, the source DBLP graph exhibits an increase in the clustering coefficient for vertices with degrees between 100 and 160. Both Darwini and BTER are able to reproduce this artifact.

Further, for the Facebook graph, we also measure the distribution of the clustering coefficient values across the entire graph. We show this result in Figure 2(c). As expected, Kronecker produces only vertices with low clustering coefficient. BTER tends to produce many vertices with high clustering coefficient. Darwini captures the source distribution better than all models.

3) Joint-degree distribution: Darwini tries to maintain the graph assortativity property observed in real social graphs, as described in Section II. This impacts the joint-degree distribution, the degree distribution of the neighbors of a vertex as a function of the degree of the vertex. The joint-degree distribution is shown to be correlated to other important graph metrics such as conductance [29]. Here, we measure how close

to the original graph the generated joint-degree distribution is for Darwini, BTER and Kronecker. In Figure 5, we show the joint-degree distribution for vertices with degree 32 and 500.

First, notice that the distribution produced by Kronecker diverges the most from the original one. The BTER model improves upon Kronecker, but still produces a skewed joint degree distribution. This is due to grouping only vertices with the same degree into the same block. As a result, more vertices with same degree are connected to each other than in the original graph. For instance, notice in Figure 5(a) that for a vertex with a degree 32 there is a spike in the frequency of neighbors with the same degree that does not appear in reality. Instead, because Darwini does not group vertices based on degree, but based on the $N_{\triangle,i}$ value, it allows the connection of vertices with more diverse degrees.

We also measured the KL-divergence of the joint-degree distributions between the original and the generated graphs. The result, shown in Table II, verifies that Darwini produces a more accurate distribution.

4) Graph diameter: The diameter is another fundamental metric of graphs, with social networks exhibiting a small diameter. Here, we measured the effective diameter of all graphs, the average distance between two pairs of nodes in the graph.

Table III shows that all graphs exhibit a smaller diameter than the original graph. BTER is the one closest to the original, while the Kronecker graph has the smallest diameter. We hypothesize that since the Kronecker model does not impose higher clustering coefficients, it allows more random connections between vertices and this results in shrinking the diameter.

5) PageRank and eigenvalue distributions: The PageRank distribution is another metric used to characterize a graph structure. In Figures 6(a) and 6(b), we compare the PageRank distributions between Darwini, BTER and Kronecker.

Although graphs generated by Darwini exhibit more accurate PageRank distributions than other models, notice that the distribution has a significant dip caused by the block structure created at the initial stage. We hypothesize that this is due to the fact that real graphs have more hierarchical and overlapping community structure, while Darwini strictly assigns every vertex to one community.
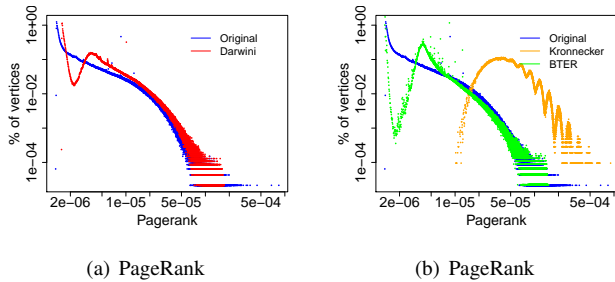
Fig. 6. PageRank of the source Facebook graph and the Darwini, BTER and Kronecker graphs.

Though omitted due to space restrictions, Darwini and BTER result in similar eigenvalue distributions, with Darwini tending to overestimate higher eigenvalues.

B. Impact on system performance

One of our initial motivations was to use Darwini to allow researchers to benchmark graph processing systems on a reference graph, for instance the Facebook social graph, without sharing the graph. Here, we measure how representative the synthetic graphs are in terms of the observed system performance.

In this experiment, we use as source a Facebook connected subgraph with 300M vertices, and generate synthetic graphs with Darwini, BTER and Kronecker. Subsequently, we run a variety of graph mining applications developed on the Apache Giraph framework and compare the observed performance of Apache Giraph on all graphs. Here, we run five different applications: PageRank, Connected Components (ConnComp), Eigenvalue decomposition (EIG), Balanced Partitioning (BP) [16], and Friends-of-Friends counting (FoF).
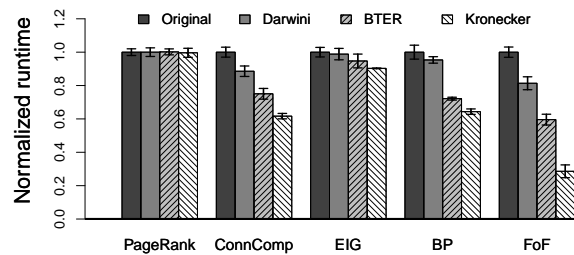

Fig. 7. Application runtime on the real and the synthetic graphs, normalized by the time observed on the real graph.

Figure 7 shows the runtime on the real and the synthetic graphs for all the applications. The time is normalized by the runtime on the real graph. Each data point is an average of three runs. First, notice that for PageRank the difference is small for all graphs. The runtime of this application is proportional to the number of edges in the graph. Since all synthetic graphs have almost the same number of edges with the original graph, Giraph exhibits the same runtime.

The difference in performance becomes more apparent for the rest of the applications because of their computation and communication patterns. For instance, in the Giraph

implementation of the Friends-of-Friends counting algorithm, every vertex creates a message that is proportional in size to its degree d and sends it to all its neighbors, resulting in communication overhead with size d^2 per vertex. Therefore, even though the number of edges is the same in all graphs, different clustering can impact the communication overhead significantly and, hence, the observed application performance. As an extreme, the low clustering coefficient of the Kronecker graph incurs smaller messages. This results in a runtime that is 3.5 times less than the runtime observed on the real graph. In all of these cases, the observed performance on the graph generated with Darwini is closer to the one on the original graph.

Next, we evaluate the impact of the synthetic graph structure on system performance when the input graph is partitioned in advance. Graph processing systems often partition the graph across workers intelligently, to minimize inter-worker communication, reduce memory pressure due to messaging and balance the load across worker machines. Typical graph partitioning algorithms try to minimize the edge cut, while keeping the size of the partitions even. If the structure of the generated graph differs significantly from that of the original graph, the observed performance on a partitioned graph may differ more.

In this experiment, we applied balanced graph partitioning [16] on the original graph and on each synthetic graph. Then we run PageRank and Friends-of-Friends counting on all the partitioned graphs and measured the relative performance difference as in the previous example. These two applications represent two diverse algorithms with respect to computation and communication.

Figure 8 shows the results. For each application, we show the normalized runtimes when the graph is randomly partitioned and when it is partitioned using balanced partitioning. Notice that, unlike the previous experiment, the relative performance of PageRank varies more. The communication overhead and, hence, the running time depends on the size of the edge cut and the skew of the partition sizes. The size of the edge cut on the Darwini graph is much closer to that of the original graph, resulting in similar performance. BTER and Kronecker produce a less connected graph which makes it easier to achieve a good cut, resulting in a lower runtime. The difference is more apparent in the Friends-of-Friends counting application. Notice that the time for the Kronecker graph does not change between the random and balanced partitioning. The graph is that sparse that partitioning does not make a significant difference in the runtime.

C. Scalability

Here, we evaluate the scalability of the Darwini implementation. We use an experimental cluster with 200 machines, each with 256GB of RAM and 48 cores. Figure 9(a) shows the time to generate a graph as a function of the output graph size. The graph generation time scales linearly with the number of vertices until we hit the memory limit. In Figure 9(b), we show how the graph generation time improves as we increase
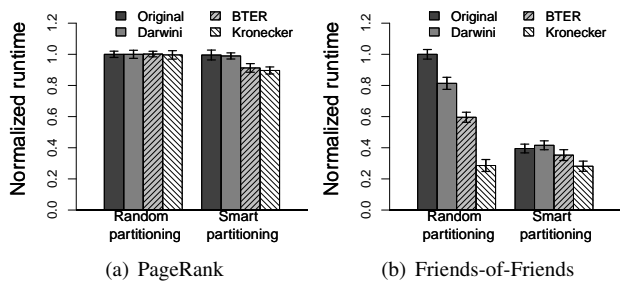
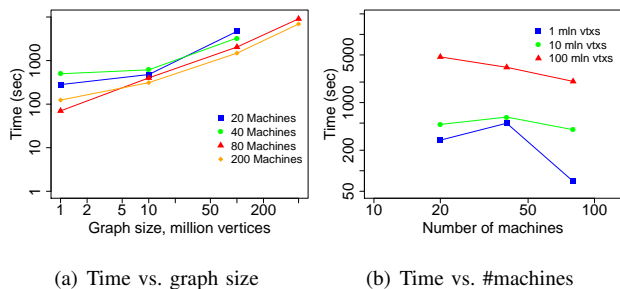Fig. 8. Relative system performance on partitioned graphs: (a) PageRank. (b) Friends-of-friends counts.

Fig. 9. (a) Darwini scales linearly on the number of vertices. (b) For large graphs, Darwini scales linearly as the compute cluster size increases.

the size of the compute cluster. For a sufficiently large graph, the time decreases linearly. Smaller graph sizes do not benefit from a large number of machines due to the network overhead.

Further, we used Darwini to generate a scaled-up version of the Facebook social graph. We captured the entire social graph as the source graph and generated a synthetic graph with one trillion edges. This task took approximately 7 hours on the same 200-machine compute cluster. Although we omit the details, the generated distributions are close to the source distribution, consistent with our results on the smaller subgraph.

V. RELATED WORK

Our work is inspired by the Block Two-Level Erdös-Rényi (BTER) [15], [20] model. As we show in our evaluation, the BTER model is capable of capturing the average clustering coefficient, but fails in generating high-degree vertices and often results in graphs with skewed joint degree distribution.

The Barabasi-Albert model [30] uses the preferential attachment mechanism to produce random graphs with power-law degree distributions. Recent work has focused on scaling the Barabasi-Albert model using efficient data structures [31]. However, preferential attachment does not generally produce higher than random number of triangles, resulting in graphs with low clustering coefficient.

The Random Walk model [25] simulates the randomized walk behavior of friend connections in a social network. The Nearest Neighbor model [25] is based on the idea that people with common friends are more likely to become friends. While Random Walk and Nearest Neighbor models are relatively accurate in terms of degree distribution and clustering coefficient,

they are biased towards inter-connecting high-degree nodes, and produce graphs with significantly shorter path lengths and network diameter [9].

Kronecker graphs [10] are generated by recursive application of Kronecker multiplication to an initiator matrix. The initiator matrix is selected by applying the KronFit algorithm to the original graph. Modifying the size of the initiator matrix introduces a tradeoff between overhead and accuracy. In our experimentation, we found it hard to apply the existing KronFit implementation to large graphs.

DK-graphs [26] is a family of stochastically generated graphs that match the respective DK-series of the original graph. DK-1 graphs match the degree distribution of the original graph, while DK-2 matches the joint degree distribution. DK-3 matches the corresponding DK-3 series, including the clustering coefficient of the original graph. However generating DK-3 graph using rewiring incurs very high overhead. We are not aware of any efficient algorithm that generates large DK-3 graphs.

The growth model presented in [32] is similar to our approach. It first generates communities according to a Web of Trust growth model, and then interconnects all communities, while assuming a certain community size distribution. Unlike Darwini, this approach does not explicitly model the degree or local clustering coefficient distributions. However, real graphs may present peculiarities in these distributions, such as hard limits on the number of friends, or anomalies around specific degree values [12] that can be impactful when analyzing system performance. Hence, we believe that capturing these graph properties is important.

VI. CONCLUSION AND FUTURE WORK

This paper introduced Darwini, a scalable synthetic graph generator that can accurately capture important metrics of social graphs, such as degree, clustering coefficient and joint-degree distributions. We implemented Darwini on top of a graph processing framework, making it possible to use it on any commodity cluster. To facilitate access to large-scale datasets, apart from open sourcing Darwini, we have also made synthetic datasets publicly available [2] [33].

At the same time, we believe there are interesting future directions in this area. For instance, real social network users typically belong to multiple communities, based on workplace, university affiliation, and others, affecting the connectivity of the graph. However, Darwini and other models assign vertices to a single community. Capturing the multi-community structure will provide more accurate synthetic datasets. Furthermore, current generators focus on the graph structure, and lack models for generating metadata, such as community labels characterizing vertices, or user similarity metrics characterizing edges. Such data will enable research in a variety of areas such as community detection algorithms, without the need to share the original data while protecting user privacy.

REFERENCES

[1] A. Iosup, T. Hegeman, W. L. Ngai, S. Heldens, A. P. Perez, H. C. Thomas Manhardt, M. Capota, N. Sundaram, M. Anderson, I. G. Tenase, Y. Xia, L. Nai, and P. Boncz, "LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms," in *Very Large Data Bases*, vol. 9, no. 13, 2016.

[2] M. Kabiljo, D. Logothetis, S. Edunov, and A. Ching, "A comparison of state-of-the-art graph processing systems," Facebook Blog Post, http://tinyurl.com/giraph-vs-graphx, October 2016.

[3] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: distributed graph-parallel computation on natural graphs," in *USENIX OSDI'12*, Berkeley, CA, USA, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387883

[4] 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 *USENIX OSDI'14*, Broomfield, CO, Oct. 2014. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez

[5] G. Wang, W. Xie, A. J. Demers, and J. Gehrke, "Asynchronous large-scale graph processing made easy," in *CIDR*, 2013.

[6] "Facebook Reports Third Quarter 2017 Results," https://investor.fb.com/investor-news/press-release-details/2017/Facebook-Reports-Third-Quarter-2017-Results/default.aspx.

[7] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: graph processing at Facebook-scale," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1804–1815, 2015.

[8] L. Backstrom, C. Dwork, and J. Kleinberg, "Wherefore art thou r3579x?: Anonymized social networks, hidden patterns, and structural steganography," in *International Conference on World Wide Web*, 2007, pp. 181–190. [Online]. Available: http://doi.acm.org/10.1145/1242572.1242598

[9] A. Sala, L. Cao, C. Wilson, R. Zablit, H. Zheng, and B. Y. Zhao, "Measurement-calibrated graph models for social network experiments," in *International Conference on World Wide Web*, 2010. [Online]. Available: http://doi.acm.org/10.1145/1772690.1772778

[10] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: An approach to modeling networks," *J. Mach. Learn. Res.*, vol. 11, pp. 985–1042, Mar. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1756006.1756039

[11] A. Sala, H. Zheng, B. Y. Zhao, S. Gaito, and G. P. Rossi, "Brief announcement: Revisiting the power-law degree distribution for social graph analysis," in *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 2010. [Online]. Available: http://doi.acm.org/10.1145/1835698.1835791

[12] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow, "The Anatomy of the Facebook Social Graph," *CoRR*, vol. abs/1111.4503, 2011. [Online]. Available: http://arxiv.org/abs/1111.4503

[13] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *ACM SIGMOD'10*, 2010. [Online]. Available: http://portal.acm.org/citation.cfm?id=1807167.1807184

[14] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, no. 6684, pp. 440–442, 06 1998. [Online]. Available: http://dx.doi.org/10.1038/30918

[15] C. Seshadhri, T. G. Kolda, and A. Pinar, "Community structure and scale-free collections of erdös-rényi graphs," *CoRR*, vol. abs/1112.3644, 2011. [Online]. Available: http://arxiv.org/abs/1112.3644

[16] A. Shalita, B. Karrer, I. Kabiljo, A. Sharma, A. Presta, A. Adcock, H. Kllapi, and M. Stumm, "Social Hash: An Assignment Framework for Optimizing Distributed Systems Operations on Social Networks," in *USENIX NSDI'16*, 2016.

[17] "Darwini source code," https://issues.apache.org/jira/browse/GIRAPH-1043.

[18] "Apache Giraph - http://giraph.apache.org." [Online]. Available: http://giraph.apache.org

[19] R. Albert and A.-L. Barabási, "Statistical mechanics of complex networks," *Rev. Mod. Phys.*, vol. 74, pp. 47–97, Jan 2002. [Online]. Available: http://link.aps.org/doi/10.1103/RevModPhys.74.47

[20] T. G. Kolda, A. Pinar, T. Plantenga, and C. Seshadhri, "A scalable generative graph model with community structure," *CoRR*, vol. abs/1302.6636, 2013. [Online]. Available: http://arxiv.org/abs/1302.6636

[21] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel, "Chaos: Scale-out graph processing from secondary storage," in *Symposium on Operating Systems Principles*, 2015, pp. 410–424. [Online]. Available: http://doi.acm.org/10.1145/2815400.2815408

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

[23] J. Yang and J. Leskovec, "Defining and Evaluating Network Communities based on Ground-truth," in *IEEE International Conference on Data Mining*, May 2012. [Online]. Available: http://arxiv.org/abs/1205.6233

[24] "Graph models for online social network analysis," http://current.cs.ucsb.edu/socialmodels/.

[25] A. Vázquez, "Growing network with local rules: Preferential attachment, clustering hierarchy, and degree correlations," *Phys. Rev. E*, vol. 67, p. 056104, May 2003. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevE.67.056104

[26] P. Mahadevan, D. Krioukov, K. Fall, and A. Vahdat, "Systematic topology analysis and generation using degree correlations," in *SIGCOMM'06*, 2006, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1159913.1159930

[27] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: Densification laws, shrinking diameters and possible explanations," in *ACM SIGKDD '05*, 2005, pp. 177–187. [Online]. Available: http://doi.acm.org/10.1145/1081870.1081893

[28] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *International Conference on World Wide Web*. ACM Press, Apr. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1772690.1772751

[29] I. Stanton and A. Pinar, "Constructing and sampling graphs with a pre-scribed joint degree distribution," *Journal of Experimental Algorithmics*, vol. 17, Sep. 2012.

[30] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, 1999. [Online]. Available: http://science.sciencemag.org/content/286/5439/509

[31] A. Hadian, S. Nobari, B. Minaei-Bidgoli, and Q. Qu, "ROLL: Fast In-Memory Generation of Gigantic Scale-free Networks," in *SIGMOD'16*, 2016, pp. 1829–1842.

[32] B. Schiller, T. Strufe, D. Kohlweyer, and J. Seedorf, "Growing a web of trust," in *2015 IEEE 40th Conference on Local Computer Networks (LCN)*, Oct 2015, pp. 100–108.

[33] "Darwini synthetic graphs," https://s.apache.org/darwini-synth.