Poster Paper

Stability notions in synthetic graph generation: a preliminary study

Wilco van Leeuwen
George H.L. Fletcher
Nikolay Yakovets
Eindhoven University of Technology
Eindhoven, The Netherlands
{w.j.v.leeuwen@student., g.h.l.fletcher@, hush@}tue.nl

Angela Bonifati
University of Lyon 1
Lyon, France
angela.bonifati@univ-lyon1.fr

ABSTRACT

With the rise in adoption of *massive* graph data, it becomes increasingly important to design graph processing algorithms which have *predictable* behavior as the graph *scales*. This work presents an initial study of *stability* in the context of a schema-driven synthetic graph generation. Specifically, we study the design of algorithms which generate high-quality sequences of graph instances. Some desirable features of these sequences include *monotonic containment* of graph instances as they grow in size and *consistency* of structural properties across the sequence. Such stability features are important in understanding and explaining the scalability of many graph algorithms which have cross-instance dependencies (e.g., solutions for role detection in dynamic networks and graph query processing). We implement a preliminary approach in the recently proposed open-source synthetic graph generator gMark and demonstrate its viability in generating stable sequences of graphs.

1. INTRODUCTION

Rising adoption of massive graph data (e.g., social networks, WWW, biological data) necessitates the development of algorithms which are able to handle the vast amounts of information stored in these datasets. Synthetic graph generators are a valuable tool for investigating the performance of graph processing algorithms since it allows to generate a sequence (or, a *family*) of graph instances of increasing sizes. For performance studies to be useful and reproducible, it is important to ensure the *quality* of the generated instances themselves and the instance family, as a whole. The simplest quality measure of the instance is its size. In some cases (e.g., in benchmarking of naive graph serialization), generation of different-sized independent random graph instances is sufficient to demonstrate the performance characteristics of the algorithm. However, in other cases (e.g., in benchmarking of query processing engines), stronger quality guarantees are desired. For example, in addition to graph size, a user-defined graph *schema* might include the enumeration of node and edge labels along with their proportions in the generated graph instances. Furthermore, in- and out-degree distributions can be defined on top of the constraints on source-target node pairs in a graph. Graph generation approaches that produce graph instances which conform to such extended schema are called *schema-driven*.

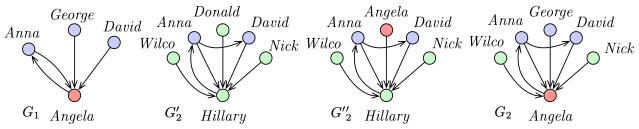


Figure 1: Node-level changes between instances. In graph instances G_1 and G_2, node *Angela* is a "leader" with relatively high in-degree. In G'_2, however, *Angela* does not appear at all, while in G''_2, she is merely a "follower."

In benchmarking of graph algorithms it is often important to control how relationships involving specific nodes *evolve* between graph instances in the family. Ideally, given a particular node, its structural properties should be stable across the entire sequence of generated graph instances. For example, queries which mention constants are common in the design of benchmarks, as they allow fine-tuned control of query selectivity and run-time behavior [4, 5, 8, 9, 12]. As another example, in the study of solutions for role detection in social networks, it is often desirable that structural features of individual nodes (i.e., individual actors in the network) are stable as the network grows in size.

To concretely illustrate stability, consider the graph instances shown in Figure 1. Here, G_2, G'_2, and G''_2 have the same size and are all larger than G_1. Suppose that a given schema (S) models a typical behavior of follows edges in a social network. This schema defines the in-distribution of all edges in the graph to be Zipfian and all nodes to be of type Person. Observe that each of the instances G_1, G'_2, G''_2, and G_2 satisfy S. Suppose, we fix a specific node in G labeled *Angela* and trace its behavior across instances in Figure 1. Specifically, *Angela* is a "leader" (followed by many) in G_1 and G_2, but is not present in G'_2 or is not a leader in G''_2. Hence, using instance families $\{G_1, G'_2\}$ and $\{G_1, G''_2\}$ would lead to inconsistent results when studying the behavior of an

algorithm which depends on *Angela*, e.g., in a performance study of a query evaluation engine using a benchmark query which specifically mentions *Angela*.

State of the art. The study of solutions for controlled generation of synthetic database instances has a long history in the data management community [2, 10]. In the domain of graph databases, synthetic generation of realistic graphs is currently a topic of intense investigation [1,4,5,7,8,13,14]. In this context, quality metrics on individual synthetic graphs (with respect to a given real graph or a given schema) have been proposed, e.g., [3,7,11,13,14]. To our knowledge, however, there has been no prior study of stability across **sequences** of graphs in synthetic instance generation.

Contributions. Motivated by these observations, in this paper we initiate the study of stability in synthetic graph generation. We consider basic properties of instance families which are, to the best of our knowledge, not satisfied by current schema-driven graph generators. We present the preliminary design of a scalable solution for producing instance families which are *stable* with respect to edge-type degree distributions defined by a given schema (e.g., for the Zipfian distribution in the family $\{G_1, G_2\}$ of Figure 1).

The detailed structure and contributions of this paper are as follows. We define two basic desirable properties of generated graph instance families (§2.1) and design a novel measure of distribution stability for a given family (§2.1). We then present an algorithm for generation of stable instance families and analyze its complexity (§2.2). We implement our approach in the state-of-the-art open-source gMark graph generator [4, 5], and demonstrate that our solution is scalable (§3.1) and produces instance families which are significantly more stable than the original gMark instance generator (§3.2).

2. STABLE GENERATION

2.1 Preliminaries

We study the following problem, on finite directed edge-labeled graphs. Given a finite sequence of positive integers $n_1, \ldots, n_k$ such that $n_i < n_{i+1}$, for $1 \leq i \leq k$, generate a sequence of graphs $\mathcal{F} = (G_1, \ldots, G_k)$ such that $|nodes(G_i)| = n_i$, for $1 \leq i \leq k$, where $nodes(G)$ denotes the node set of graph G and $|A|$ denotes the size of set A. If we are additionally given a graph schema S as input, we further require that each G_i is a valid instance of S.

In this initial study, we consider the following desirable properties of generated graph sequences:

- *Monotonicity.* It holds that $edges(G_i) \subseteq edges(G_{i+1})$, for $1 \leq i \leq k$, where $edges(G)$ denotes the edge set of graph G.
- *Distribution stability.* If the degree structure of an edge type follows a fixed distribution (e.g., edges labeled follows in a social network have a Zipfian in-distribution), then the position of nodes in the distribution is stable throughout the graph sequence $\mathcal{F}$.

The degree (deg) of a node needs to be stable in its distribution in all graphs of a given sequence. To capture this, we define the *rank* of a node n in graph G as:

$$rank(n, G) \quad = \quad \frac{deg(n)}{\mathsf{max}_{n \in nodes(G)}(deg(n))}$$

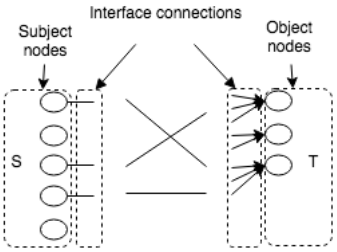


Figure 2: Generating the edges of one edge-type with five subject nodes and three object nodes.

where the value of $rank(n, G)$ ranges from 0 to 1. Letting σ_n denote the standard deviation of the rank of n over all instances in $\mathcal{F}$, i.e., the standard deviation of the set $\{rank(n, G) \mid G \in \mathcal{F}\}$, we define the *stability* of n in $\mathcal{F}$ as:

$$stability(n, \mathcal{F}) \quad = \quad 1 - 2\sigma_n.$$

A $stability(n, \mathcal{F}) = 1$ indicates that the rank of n never changes, whereas $stability(n, \mathcal{F}) = 0$ means that n is completely unstable, with respect to degree structure.

2.2 Generation Approach

The original gMark graph generator (gMarkGraphGen) does not satisfy monotonicity of graph generation nor does it guarantee distribution stability; the time complexity of generation is linear in the number of nodes of the graph instance [5]. We next propose an algorithm, MonStaGen, that generates instance families which are both monotonic and stable with respect to edge-type degree distributions defined by a given schema. An analysis of this algorithm, however, shows that the satisfaction of these properties comes at the expense of increased time complexity ($O(n \log n)$, where n is the number of nodes).

MonStaGen separately calls a procedure (Algorithm 1) for each edge type in a given schema. As a consequence, the generation of subgraphs that correspond to each edge type can be executed in parallel. Consider the graph of a single edge type as a bipartite graph with the two disjoint sets S and T. Set S represents all the subject nodes of the edge type, whereas set T represents all the object nodes of the edge type. Graph generation proceeds by iteratively adding edges from nodes in S to nodes in T.

Figure 2 shows an example of the generation of one edge-type with five subject nodes (elements of set S) and three object nodes (elements of set T). This figure also introduces our concept of *interface connections*. We call the connections (i.e., edges) that a node can potentially receive the *interface connections (ICs)* of this node. Whenever a new node is added to the graph, the degree distributions in a given schema determine the number of ICs of this node depending on whether it is a subject (out-degree) or an object (in-degree). Whenever a new edge is added, the participating ICs for its subject and object nodes are closed.

Generated subject nodes, object nodes, and edges are cached for subsequent processing by functions *addSubjectNodes* and *addObjectNodes*. Function *addEdge* decrements the amount of open ICs of the subject node and the object node by one. Observe that, by construction, the generated sequence of graphs satisfies the monotonicity property. Next, we discuss the immediate challenges that need to be

Algorithm 1 processEdgeType(edgeType, graph, probability p, #subjectNodes, #objectNodes)

```
graph.addSubjectNodes(#subjectNodes)
graph.addObjectNodes(#objectNodes)

if edgeType.subjectNodes is scalable xor edgeType.objectNodes is scalable then
    updateICsForNonScalableNodes()
end if
if in- or out-degree distribution is Zipfian then
    updateICsForNodesWithZipfianDistribution()
end if

vector v_src, v_trg
for subject in graph.subjects do
    for 1:subject.openICs do
        v_src.add(subject)
    end for
end for
for object in graph.objects do
    for 1:object.openICs do
        v_trg.add(object)
    end for
end for
shuffle(v_src); shuffle(v_trg)

for i ∈ 1:min(v_src.length, v_trg.length) do
    with probability p, graph.addEdge(v_src[i], v_trg[i], edgeType.predicate)
end for

return graph
```

met during schema-driven stable graph generation.

Subject-to-object scalability mismatch. Consider the edge-type *Persons live in a City*, where the number of persons scale with the graph size, i.e. the subject nodes are scalable, and the number of cities is fixed. When the number of ICs are fixed for the non-scalable object nodes, we will reach a point where new edges cannot be generated anymore. This is because the total number of ICs in the *City* nodes will remain the same when growing the graph, whereas the total number of ICs in the *Person* nodes will grow. It is still possible to grow the graph after this point, resulting in more *Person* nodes with new ICs. However, these ICs can never be used anymore, since all object nodes are not able to receive any connection and new objects cannot can be added. This problem is solved by updating the number of ICs of the non-scalable nodes.

Skewed distributions. During the construction of the new graph instance, the ICs of the nodes which participate in a Zipfian distribution need to be updated to ensure that nodes with a very high degree will be able to continue to receive more connections.

We proceed by creating separate vectors for subject and object nodes with as many entries for each node as it has assigned ICs. We then randomly shuffle these two vectors. Finally, with probability p, we add each edge from the source vector to the target vector. Here, p ensures the balance between the connectedness of the new instance with later instances in the sequence, on one hand, and the quality of the degree distributions, on the other.

3. EMPIRICAL STUDY

In this section, we report the results of three experimental studies of our approach. In all of the experiments, we investigated the generation of graphs with skewed (Zipfian) degree distributions. We selected Zipfian distribution parameter of 2.5, as it corresponds to structure found in many real-world graphs [6]. The input parameter p to MonStaGen (Algorithm 1) was set to 0.97.

3.1 Run-time

We design two experiments to demonstrate the running time of MonStaGen compared to gMarkGraphGen. As noted in §2.2, the complexity of MonStaGen is slightly higher ($O(n \log n)$ vs. $O(n)$). Figures 3a and 3b show the difference in the running time for both approaches. In our first experiment (Figure 3a), we benchmark the generation of an instance family which consists of a single graph instance of increasing size. Here, gMarkGraphGen slightly outperforms MonStaGen due to increased complexity required by MonStaGen to satisfy the monotonicity and stability properties of instance families.

In our second experiment (Figure 3b), we generate families of increasing size with ten graph instances each $\mathcal{F} = G_1, \ldots, G_{10}$. In MonStaGen, for producing graph G_{i+1}, the previously generated graph G_i is used, instead of generating the whole graph from scratch. We recall that sequences generated by gMarkGraphGen do not satisfy the monotonicity property.

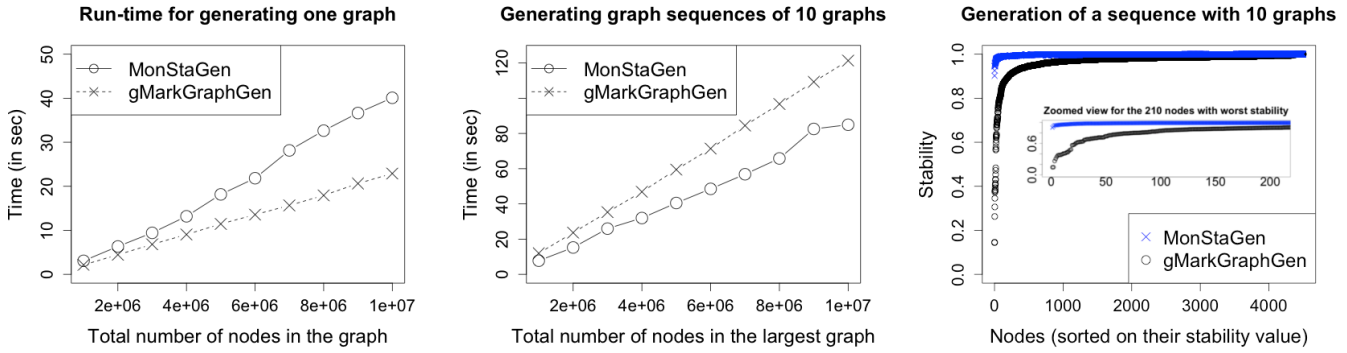
In Figure 3b, the *x-axis* corresponds to the total number of nodes in the largest graph (G_{10}) in the generated instance. We set the number of nodes in graph G_i to a fraction $\frac{i}{10}x$, where x is the total number of nodes in graph G_{10}.

As expected, MonStaGen is slower when generating a single graph, whereas it supersedes gMarkGraphGen when generating multiple graphs. This means that MonStaGen scales with the number of graphs in the sequence, while gMarkGraphGen scales with the number of nodes in the graphs. This behavior is quite interesting and leaves open the user's choice of which generator to employ in a given application.

3.2 Stability of nodes in degree distribution

We next consider generation of a graph sequence $\mathcal{F} = (G_1, \ldots, G_{10})$, where the total number of nodes of G_i is $i \cdot 1000$, for $1 \leq i \leq 10$. Figure 3c shows the stability of nodes of a single edge type with Zipfian degree distribution, as noted above, and $0.5n$ subject nodes, where n is the total number of nodes in the graph.

The nodes added in the last graph are not taken into account, because they will always have a stability of 1. The total number of subject nodes taken into account is $0.5 \cdot (10 - 1) \cdot 1000 = 4500$. The stability value for all of these nodes, calculated and sorted on such a value, is illustrated in Figure 3c. We can see that the stability of nodes in MonStaGen is significantly higher than gMarkGraphGen. The long tail, which indicates a high number of nodes with a high stability, is the effect of a Zipfian distribution, where a node has a very high probability of having a very low degree. This means that many nodes will have a low degree in all the graphs of the sequence, resulting in a high stability.



(a) Average run-time of the generation of single graph instances. Each data point is the average of four runs, after dropping the highest and lowest values.

(b) Average run-time of the generation of a graph sequence with 10 graphs with equal increments. Averages taken as in (a).

(c) Stability comparison of all the nodes in a Zipfian(2.5) degree distribution. A zoom of the 210 nodes with worst stability is shown inside the plot.

Figure 3: Experimental comparisons of gMarkGraphGen and MonStaGen.

If we consider the stability of nodes, we see that the minimal stability in MonStaGen is 0.9020 and the median value is 0.9988. In the graphs generated by gMarkGraphGen, 210 nodes that have a lower stability value than our worst-case 0.9020 and 4393 nodes that have lower stability than our median 0.9988. In other words, with gMarkGraphGen, over 97% of the nodes are more unstable than the median value of those nodes generated with MonStaGen. The inner plot of Figure 3c shows a zoom-in on the 210 most unstable nodes.

4. LOOKING AHEAD

In this preliminary study, we have highlighted the utility of properties, such as stability, characterizing graph sequences rather than individual graphs. We have proposed a first algorithm for generating graph sequences conforming to a schema which satisfy the monotonicity property and are significantly more stable than those generated using gMark, a state-of-the-art synthetic graph generator. To our knowledge, ours is the first schema-driven synthetic graph generator having these properties.

Our study opens up several directions for future work. A major direction is to establish further stability properties occurring in real-world graphs, and extending our graph generation algorithm to capture these (e.g., stability with respect to node-centrality measures). Further, in addition to edge insertion, we plan to consider other natural aspects of graph evolution such as edge deletion, batch updates, and temporal dynamics. Finally, we would like to incorporate our solutions in the open-source gMark framework.[1]

5. REFERENCES

[1] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee. Diversified stress testing of RDF data management systems. In *ISWC*, pages 197–212, Riva del Garda, Italy, 2014.

[2] A. Arasu, R. Kaushik, and J. Li. Data generation using declarative constraints. In *SIGMOD*, pages 685–696, Athens, Greece, 2011.

[3] M. Arenas, G. I. Diaz, A. Fokoue, A. Kementsietsidis, and K. Srinivas. A principled approach to bridging the gap between graph data and their schemas. *PVLDB*, 7(8):601–612, 2014.

[4] G. Bagan, A. Bonifati, R. Ciucanu, G. H. L. Fletcher, A. Lemay, and N. Advokaat. Generating flexible workloads for graph databases. *PVLDB*, 9(13):1447–1460, 2016.

[5] G. Bagan, A. Bonifati, R. Ciucanu, G. H. L. Fletcher, A. Lemay, and N. Advokaat. gMark: Schema-driven generation of graphs and queries. *IEEE Trans. Knowl. Data Eng.*, in press, 2017.

[6] A. Clauset, C. R. Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Review*, 51(4):661–703, 2009.

[7] S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea. Apples and oranges: a comparison of RDF benchmarks and real RDF datasets. In *SIGMOD*, pages 145–156, Athens, Greece, 2011.

[8] O. Erling et al. The LDBC social network benchmark: Interactive workload. In *SIGMOD*, pages 619–630, Melbourne, 2015.

[9] A. Gubichev and P. A. Boncz. Parameter curation for benchmark queries. In *TPCTC*, pages 113–129, Hangzhou, China, 2014.

[10] E. Lo, N. Cheng, W. W. K. Lin, W. Hon, and B. Choi. MyBenchmark: generating databases for query workloads. *VLDB J.*, 23(6):895–913, 2014.

[11] Y. Luo, G. H. L. Fletcher, J. Hidders, P. D. Bra, and Y. Wu. Regularities and dynamics in bisimulation reductions of big graphs. In *GRADES*, page 13, New York, NY, 2013.

[12] C. Mishra, N. Koudas, and C. Zuzarte. Generating targeted queries for database testing. In *SIGMOD*, pages 499–510, Vancouver, BC, 2008.

[13] S. Qiao and Z. M. Özsoyoglu. Rbench: Application-specific RDF benchmarking. In *SIGMOD*, pages 1825–1838, Melbourne, 2015.

[14] J. W. Zhang and Y. C. Tay. GSCALER: synthetically scaling a given graph. In *EDBT*, pages 53–64, Bordeaux, 2016.

[1] https://github.com/graphMark/gmark