# Synthetic Graph Generation from Finely-Tuned Temporal Constraints

Karim Alami, Radu Ciucanu, and Engelbert Mephu Nguifo

Université Clermont Auvergne & CNRS LIMOS, France
karim.alami@etudiant.univ-bpclermont.fr, ciucanu@isima.fr, mephu@isima.fr

**Abstract.** Large-scale graphs are at the core of a plethora of modern applications such as social networks, transportation networks, or the Semantic Web. Such graphs are naturally evolving over time, which makes particularly challenging graph processing tasks e.g., graph mining. To be able to realize rigorous empirical evaluations of research ideas, the graph processing community needs finely-tuned generators of synthetic time-evolving graphs, which are particularly useful whenever real-world graphs are unavailable for public use. The goal of this paper is to report on an ongoing project that aims at generating synthetic time-evolving graphs satisfying finely-tuned temporal constraints specified by the user.

## 1 Introduction

Large-scale graphs are used to model a variety of real-world domains. In practice, both nodes and edges of such graphs have properties that are naturally evolving over time. For example, in a geographical database storing information about cities and transportation facilities, the nodes of type city have evolving properties e.g., weather and air quality, whereas the edges that encode transportation facilities between cities have evolving properties e.g., price and duration.

The graph evolution over time makes particularly challenging the graph processing tasks e.g., graph querying and mining. This motivated the community to propose frameworks for building applications on time-dependent graphs e.g., Graphast [3], or historical graph management systems e.g., DeltaGraph [2].

To be able to realize rigorous empirical evaluations of research ideas, the graph processing community needs tunable evolving graph generators, which are particularly useful whenever real-world graphs are unavailable for public use.

In this paper, we report on our ongoing research on EGG (**E**volving **G**raph **G**enerator), an open-source[1] framework for generating evolving graphs based on finely-tuned temporal constraints given by the user. We depict the architecture of EGG in Fig. 1. We built EGG on top of gMark [1], a state-of-the-art static graph generator. EGG takes as input (i) an initial graph generated by gMark, and (ii) an evolving graph configuration that encodes how the evolving properties of the nodes and edges of the graph from point (i) should evolve over time. The output of EGG is a sequence of graph snapshots, each annotated with an interval

[1] https://github.com/karimalami7/EGG

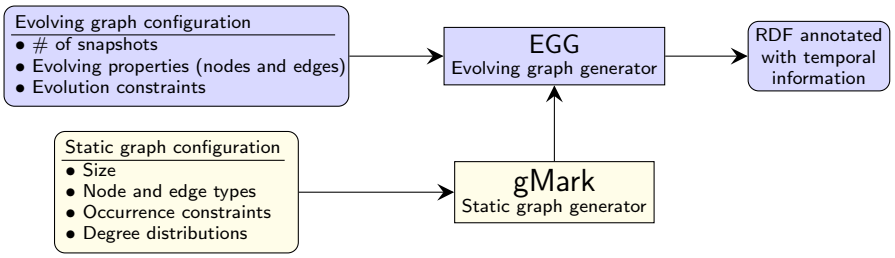


**Fig. 1.** Architecture of EGG. The bottom components are part of the existing gMark [1] static graph generator. The top components are part of our EGG contribution.

of integers. The semantics of time depends on the use case. In the running example of this paper, a snapshot corresponds to a day. In other use cases that we developed, we used snapshots representing e.g., years, semesters, and weeks.

Similarly to gMark, EGG is schema-driven and domain-independent. Throughout the paper, we use as running example a geographical database, but we have been additionally able to easily encode different application domains such as a social network or a DBLP-like bibliographical network.

## 2 Overview of EGG

We present gMark static graph configurations (Section 2.1), EGG evolving graph configurations (Section 2.2), EGG implementation challenges (Section 2.3), and preliminary EGG experimental results (Section 2.4).

### 2.1 Static Graph Configurations

We illustrate via an example the static graph configuration that the user can specify as gMark input. Assume that the user wants to generate graphs simulating a geographical database storing information about cities, and different facilities such as transportation and hotels. The user can specify as gMark input the following types of constraints: (i) graph size, given as # of nodes; (ii) node types e.g., `city` and `hotel`, and edge types e.g., `train` and `locatedIn`; (iii) occurrence constraints e.g., 10% of the graph nodes should be of type `city`, whereas 90% of the graph nodes should be of type `hotel`; (iv) degree distributions e.g.,

| *source type* $\xrightarrow{\textit{predicate}}$ *target type* | *In-distribution* | *Out-distribution* |
|---|---|---|
| `hotel` $\xrightarrow{\texttt{locatedIn}}$ `city` | Zipfian | Uniform [1,1] |

meaning that we can have an edge of type `locatedIn` from a node of type `hotel` to a node of type `city`, with a Zipfian in-distribution (since it is realistic to assume that the number of hotels in a city follows such a power-law distribution) and a uniform [1,1] out-distribution (since a hotel is located in precisely one city).

We call such gMark graph configurations as being static since the nodes of type e.g., `city` and `hotel` are rarely created or deleted. Nonetheless, such nodes (as well as the different edges connecting them) possess properties that naturally evolve over time, in an interdependent manner. The user can specify such evolving properties as input of EGG, that we detail in the next section.

### 2.2 Evolving Graph Configurations

We build on the example from Section 2.1 to introduce examples of evolving properties that can be encoded as EGG evolving graph configurations. Assume

that the user generates with gMark a graph having nodes of type `city` and `hotel`, and edges of type `train` (connecting two cities) and `locatedIn` (connecting a hotel to its city). Next, the user wants to add properties that evolve over time for the aforementioned types of nodes and edges, assuming that a graph snapshot corresponds to a day. We give below examples of such properties, together with their finely-tuned constraints to evolve among consecutive graph snapshots.

A node of type `city` has the following evolving properties:

– `weather` (unordered qualitative), which can have three uniformly-distributed values (`sunny`, `rainy`, `cloudy`). There is a probability of 50% that `weather` changes from a snapshot to the next one. Each of the three values can evolve between two consecutive snapshots to any of the other ones, with the exception of `sunny`, that cannot be followed by `rainy`.

– `qAir` i.e., "quality of air" (ordered qualitative), which can have ten possible values, following a binomial distribution with p=0.6. There is a probability of 20% that `qAir` changes from a snapshot to the next one, and it can only increment or decrement by 1 between two consecutive snapshots.

Moreover, a node of type `hotel` has the following evolving properties:

– `star` (ordered qualitative), which can have five possible values, following a geometric distribution with p=0.65. It can only change every thirty snapshots, with a probability of 10%, and it can only increment or decrement by 1.

– `availRoom` (quantitative discrete), which can have as values integers in the interval [1,100], following a binomial distribution with p=0.5. There is a probability of 80% that it changes from a snapshot to the next one, and it can increment or decrement by an integer up to 5 between two consecutive snapshots.

– `hotelPrice` (quantitative continuous), whose values follow a normal distribution in an interval that is dynamically constructed depending on the value of the property `star`. Moreover, `hotelPrice` is anti-correlated with `availRoom` i.e., if `availRoom` decreases, then `hotelPrice` increases, and vice-versa.

The user can similarly specify evolving properties and constraints for edges e.g., property `trainPrice` of edge type `train`.

### 2.3 Implementation Challenges

Building a system like EGG is an ambitious goal because, as outlined in the previous section, we aim at allowing the user to specify very expressive constraints. This leads to some interesting challenges, that we briefly discuss in this section.

*Computational complexity.* As illustrated in Section 2.2, we allow the user to specify evolution constraints where the value of a property among consecutive snapshots depends on another property. We model the inter-dependencies between such evolving properties with a dependency graph. It is easy to see that if the aforementioned dependency graph is cyclic, the generation algorithm may not halt. Consequently, in our implementation we require that the dependency graph is acyclic and we sort it topologically to decide in which order we should apply the evolution constraints. Even for acyclic dependency graphs, we suspect that it is NP-complete to decide whether there exists a sequence of graph snap-

shots satisfying the input constraints. The exact complexity is an open question that we plan to attack in the near future.

*Storage redundancy.* A naive solution to store the generated graph snapshots would be to entirely store each of them after it is generated. For the parts of the graph that do not change every snapshot, this yields redundant storage. This is why we rely on a storage format that uses named graphs to express temporal information in RDF. Our output format (that we serialize using the TriG syntax) allows us to decouple the storage of the static parts of the graph (i.e., structural information satisfied in all snapshots) and the evolving parts of the graph (i.e., the property values that change from a snapshot to the next one).

### 2.4 Experimental Evaluation of EGG

We implemented in Python the first prototype of EGG. The user can easily encode in JSON constraints as those presented in Section 2.2.

We observe the *accuracy* of the evolving graphs generated with EGG in Fig. 2, where we depict the evolving properties of one node of type `hotel` as shown by the EGG visualization module. In particular, we observe that the anti-correlation between `availRooms` and `hotelPrice` follows the constraints shown in Section 2.2.

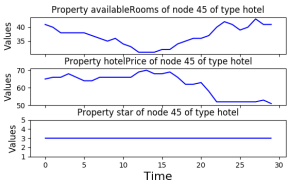


**Fig. 2.** Evolving properties for a node of type `hotel`.

As for the *scalability* evaluation, we have run EGG with several use cases and graphs of increasing sizes. In all cases, we observed that EGG has a linear time behavior with respect to the studied parameters. More details about our experimental observations are available on the EGG wiki[2].

## 3 Conclusions and Future Work

We presented our ongoing research on EGG, a system for generating evolving graphs based on finely-tuned constraints specified by the user. EGG is open-source and can be already used by the graph processing community.

We next plan to thoroughly study the impact of EGG in improving the empirical evaluations of existing temporal graph querying and mining systems.

## References

1. G. Bagan, A. Bonifati, R. Ciucanu, G. H. L. Fletcher, A. Lemay, and N. Advokaat. gMark: Schema-driven generation of graphs and queries. *IEEE Transactions on Knowledge and Data Engineering*, 29(4):856–869, 2017.
2. U. Khurana and A. Deshpande. Efficient snapshot retrieval over historical graph data. In *ICDE*, pages 997–1008, 2013.
3. R. P. Magalhães, G. Coutinho, J. A. F. de Macêdo, C. Ferreira, L. A. Cruz, and M. A. Nascimento. Graphast: an extensible framework for building applications on time-dependent networks. In *SIGSPATIAL*, pages 93:1–93:4, 2015.

[2] `https://github.com/karimalami7/EGG/wiki`