PyGraft: Configurable Generation of Synthetic Schemas and Knowledge Graphs at Your Fingertips

Nicolas Hubert[1,2]([envelope]) [ID], Pierre Monnin[3] [ID], Mathieu d'Aquin[2] [ID],
Davy Monticolo[1] [ID], and Armelle Brun[2] [ID]

[1] Université de Lorraine, ERPI, Nancy, France
{nicolas.hubert,davy.monticolo}@univ-lorraine.fr
[2] Université de Lorraine, CNRS, LORIA, Nancy, France
{mathieu.daquin,armelle.brun}@univ-lorraine.fr
[3] Université Côte d'Azur, Inria, CNRS, I3S, Sophia-Antipolis, France
pierre.monnin@inria.fr

Abstract. Knowledge graphs (KGs) have emerged as a prominent data representation and management paradigm. Being usually underpinned by a schema (*e.g.,* an ontology), KGs capture not only factual information but also contextual knowledge. In some tasks, a few KGs established themselves as standard benchmarks. However, recent works outline that relying on a limited collection of datasets is not sufficient to assess the generalization capability of an approach. In some data-sensitive fields such as education or medicine, access to public datasets is even more limited. To remedy the aforementioned issues, we release PyGraft, a Python-based tool that generates highly customized, domain-agnostic schemas and KGs. The synthesized schemas encompass various `RDFS` and `OWL` constructs, while the synthesized KGs emulate the characteristics and scale of real-world KGs. Logical consistency of the generated resources is ultimately ensured by running a description logic (DL) reasoner. By providing a way of generating both a schema and KG in a single pipeline, PyGraft's aim is to empower the generation of a more diverse array of KGs for benchmarking novel approaches in areas such as graph-based machine learning (ML), or more generally KG processing. In graph-based ML in particular, this should foster a more holistic evaluation of model performance and generalization capability, thereby going beyond the limited collection of available benchmarks. PyGraft is available at: https://github.com/nicolas-hbt/pygraft.

Keywords: Knowledge Graph · Schema · Semantic Web · Synthetic Data Generator

1   Introduction

Knowledge graphs (KGs) have been increasingly used as a graph-structure representation of data. More specifically, a KG is a collection of triples (s, p, o) where s (subject) and o (object) are two entities of the graph, and p is a predicate that qualifies the nature of the relation holding between them [16]. KGs are usually underpinned by a schema (*e.g.,* an ontology) which defines the main concepts and relations of a domain of interest, as well as the rules under which these concepts and relations are allowed to interact [14].

KGs are being used in a wide array of tasks, in many of which a limited collection of KGs established themselves as standard benchmarks for evaluating model performance. However, there are some concerns around the sole usage of these specific mainstream KGs for assessing the generalization capability of newly introduced models. For example, for the particular task of node classification, it has been demonstrated that mainstream datasets such as CiteSeer, Cora, and PubMed feature similar statistical characteristics, especially homophily [24]. Consequently, new models are assessed with respect to a collection of statistically similar datasets. Therefore, their performance improvement does not always hold beyond the standard benchmark datasets [24]. Similarly, it has been shown that many link prediction train sets suffer from extremely skewed distributions in both the degrees of entities and the occurrence of a subset of predicates [28]. In addition, some of the established link prediction datasets are plagued with data biases [27] and include many occurrences of inference patterns [8,21,22] that predictive models are able to incorporate, which may cause overly optimistic evaluation performance [8]. For example, the mainstream FB15k and WN18 datasets are substantially affected by these biases. Therefore, more diverse datasets are needed [27]. In this situation, it is of utmost importance to provide a way for researchers to generate synthetic yet realistic datasets of different shapes and characteristics, so that new models can be evaluated in a wide range of data settings.

Worse than relying on a limited number of KGs is the lack of publicly available KGs in some application fields. Conducting research in domains such as education, law enforcement, or medicine, is particularly difficult. On the grounds of data privacy, collecting and sharing real-world knowledge may not be possible. As such, domain-oriented KGs are barely accessible in these areas. However, engineers, practitioners, and researchers usually have precise ideas of the characteristics of their problem of interest. In this context, it would be beneficial to generate a synthetic KG that emulates the characteristics of a real KG [11].

The aforementioned issues led to several attempts at building synthetic generators of schemas and KGs, even though in most cases these two aspects have been considered separately. Stochastic-based generators have been proposed to output domain-agnostic KGs [1,10,25]. However good these methods are at generating large graphs quickly, their data generation process does not allow to take an underlying schema into account [11]. Therefore, the resulting KGs are not guaranteed to accurately mimic the characteristics of real-world KGs in a desired application field. In contrast, schema-driven generators are able to synthesize KGs that resemble real-world data. However, most works focused on

generating synthetic KGs on the basis of an already existing schema [23]. Synthesizing both a schema and a KG underpinned by it is a more challenging task that has been considered but with only limited success so far [23].

In this work, we aim at addressing this issue. In particular, we present PyGraft, a Python-based tool to generate highly customized, domain-agnostic schemas and KGs. The contributions of our work are the following:

– To the best of our knowledge, PyGraft is the first generator dedicated to the generation of both schemas and KGs in a unique pipeline, while being highly tunable based on a broad array of user-specified parameters. Notably, the generated resources are domain-agnostic, which makes them usable for benchmarking purposes regardless of the application field.
– The generated schemas and KGs are built with an extended set of `RDFS` and `OWL` constructs and their logical consistency is ensured by the use of a DL reasoner, which allows for both fine-grained description of resources and strict compliance with common Semantic Web standards.
– We publicly release our code with a documentation and accompanying examples for ease of use.

The remainder of the paper is structured as follows. Related work is presented in Sect. 2. In Sect. 3, PyGraft is detailed and accompanied with necessary background knowledge and comparisons with other available generators. In Sect. 4, a thorough performance analysis of PyGraft is presented. A discussion on current potential use cases, limitations, and future work can be found in Sect. 5. Lastly, Sect. 6 sums up the key insights presented in this paper.

2   Related Work

The generation principle governing synthetic graph generators leads to differentiate between *stochastic-based*, *deep generative*, and *semantic-driven* generators. Along the description of these families of generators, Table 1 provides details on current and open-source implementations. These tools are also compared with PyGraft regarding several criteria.

Stochastic-based generators are usually characterized by their ability to output large graphs in a short amount of time. Early works are represented by the famous Erdős-Rényi model [10]. The Erdős-Rényi model generates graphs by independently assigning edges between pairs of nodes with a fixed probability. The Barabási-Albert model [1] exhibits scale-free degree distributions and is based on the principle of preferential attachment, where new nodes are more likely to attach to nodes with higher degrees. The R-MAT model [5] generates large-scale power-law graphs with properties like power-law degree distributions and community structures. More recently, TrillionG [25] – presented as an extension of R-MAT – represents nodes and edges as vectors in a high-dimensional space. TrillionG allows users to generate large graphs up to trillions of edges while exhibiting lower space and time complexities than previously proposed generators. In Table 1, stochastic-based generators are represented by igraph[1],

---

[1] https://github.com/igraph/python-igraph.

NetworkX[2], and Snap[3]. Although they are not specifically designed for graph generation, they provide off-the-shelf implementations for generating random graphs, such as the Erdős-Rényi and Barabási-Albert models. They are domain-agnostic and scalable, but do not take into account any semantics.

Deep generative graph generators are trained on existing graph datasets and learn to capture the underlying patterns of the input graphs. Deep generative graph models are typically based on generative adversarial networks (GANs) and graph neural networks (GNNs), recurrent neural networks (RNNs), or variational autoencoders (VAEs). GraphGAN [31] leverages the GAN structure, in which the generative model receives a vertex and aims at fitting its true connectivity distribution over all other vertices – thereby producing fake samples for the discriminative model to differentiate from ground-truth samples. GraphRNN [32] is a deep autoregressive model that trains on a collection of graphs. It can be viewed as a hierarchical model adding nodes and edges in a sequential manner. A representant of the VAE family of generators is NeVAE [29], which is specifically designed for molecular graphs. NeVAE features a decoder which is able to guarantee a set of valid properties in the generated molecules. In particular, MolGAN [7] and NeVAE [29] are bound to molecular graph generation. Therefore, they are not domain-agnostic (Table 1). The other generators of this family are, but they do not take a schema as input when generating random graphs, *i.e.,* they are not schema-driven (Table 1).

Semantic-driven generators, in contrast, incorporate schema-based constraints or external knowledge to generate graphs that exhibit specific characteristics or follow certain patterns relevant to the given field of application. In [15], the Lehigh University Benchmark (LUBM) and the Univ-Bench Artificial data generator (UBA) are presented. The former is an ontology modelling the university domain while the latter aims at generating synthetic graphs based on the LUBM schema as well as user-defined queries and restrictions. Similarly, the Linked Data Benchmark Council (LDBC) [2] released the Social Network Benchmark (SNB), which includes a graph generator for synthesizing social network data based on realistic distributions. gMark [3] has subsequently been presented as the first generator that satisfies the criteria of being domain-independent, scalable, schema-driven, and highly configurable, all at the same time. However, it still requires already existing schemas as input. In [23], Melo and Paulheim focus on the synthesis of KGs for the purpose of benchmarking link prediction and type prediction tasks. The authors claim that there is a need for more diverse benchmark datasets for link prediction, with the possibility of having control over their characteristics (*e.g.,* the number of entities, relation assertions, number of types, etc.). Therefore, Melo and Paulheim propose a synthesizing approach which replicates input real-world graphs while allowing for controlled variations in graph characteristics. Notably, they highlight the fact that most works, including theirs, focus on synthesizing KGs based on an existing schema, which leads them to formulate the desiderata of generating both a schema and a KG from scratch as a promising venue for future work – which PyGraft actually does.

Subsequently, Feng *et al.* [11] proposed a schema-driven graph generator based on the concept of Extended Graph Differential Dependencies (GDD^x). However, their approach cannot generate domain-agnostic schemas and thus requires an existing schema as input. The DLCC benchmark proposed in [26] features a synthetic KG generator based on user-specified graph and schema properties. Beyond asking for a given number of nodes, relations and degree distribution in the resulting KG, it allows for specifying a few `RDFS` constructs for the generation of the underpinning schema.

None of the aforementioned semantic-driven generators perform a logical consistency check of the generated graphs (see Table 1). Additionally, they can only produce final KGs based on an input schema. Some of them are also domain-specific. DLCC [26] is the closest work to ours, and, to the best of our knowledge, this is the first and only work that allows to generate both a schema and a KG while being domain-agnostic. However, it only features 3 schema constructs, namely `rdfs:domain`, `rdfs:range`, and `rdfs:subClassOf`. These sole three constructs do not pose any constraints on triple generation (hence the consistency checking is not needed) and do not fully feature all Semantic Web possibilities that could exist in KGs. This is also why the resources generated with the DLCC generator do not undergo any logical consistency checks (see Table 1).

In the present work, we aim at going a step further and taking numerous `RDFS` *and* `OWL` constructs into account, as PyGraft features a broad range of schema constructs (see Tables 2 and 3). Considered together, these constructs lead to many potential sources of inconsistencies that need to be carefully avoided with specific procedures and a final consistency check using a DL reasoner. However, this more challenging setting allows PyGraft to generate KGs that feature a broader range of Semantic Web possibilities.

3 PyGraft Description

This section starts by formally introducing the notions of schema and knowledge graph. The schematic overview of PyGraft is presented in Sect. 3.2. PyGraft schema and KG generation pipelines are presented in Sects. 3.3 and 3.4, respectively.

3.1 Preliminaries

On the one hand, a schema – *e.g.,* an ontology – refers to a explicit specification of a conceptualization that includes concepts, properties, and restrictions within a particular domain of knowledge [14]. It helps ensure consistency, clarity, and interoperability when representing and sharing knowledge. In our work, we consider schemas to be represented as a collection of concepts $\mathcal{C}$, properties $\mathcal{P}$, and axioms $\mathcal{A}$, *i.e.,* $\mathcal{S} = \{\mathcal{C}, \mathcal{P}, \mathcal{A}\}$. Schemas are typically represented using formal languages such as `RDFS`[4] (Resource Description Framework) and `OWL`[5] (Web Ontology Language).

---

[4] https://www.w3.org/RDFS/.
[5] https://www.w3.org/OWL/.

Table 1. Feature comparison of graph generation tools. Dashed line is used when a feature is not applicable due to the characteristics of the described generation tool. Domain-agnostic denotes whether a given tool is able to potentially operate with schemas of different application fields.

Tool	Domain-agnostic	Schema-driven	Schema generation	Schema properties	Graph properties	Scalable	Consistency check
igraph[a]	✓	✗	✗	–	✓	✓	–
NetworkX[b]	✓	✗	✗	–	✓	✓	–
Snap[c]	✓	✗	✗	–	✓	✓	–
GraphGen [13]	✓	✗	✗	–	✓	✓	–
GraphRNN [32]	✓	✗	✗	–	✓	✓	–
GraphVAE [30]	✓	✗	✗	–	✓	✗	–
GraphWorld [24]	✓	✗	✗	–	✓	✓	–
MolGAN [7]	✗	✗	✗	–	✓	✗	–
NeVAE [29]	✗	✗	✗	–	✓	✓	–
UBA-LUBM [15]	✗	✓	✗	–	✓	✓	✗
SNB [2]	✗	✓	✗	–	✓	✓	✗
gMark [3]	✓	✓	✗	–	✓	✓	✗
Melo *et al.* [23]	✓	✓	✗	–	✓	✓	✗
GDD^x [11]	✓	✓	✗	–	✓	✓	✗
DLCC [26]	✓	✓	✓	3	✓	✓	✗
PyGraph (ours)	✓	✓	✓	13	✓	✓	✓

[a] https://github.com/igraph/python-igraph/
[b] https://github.com/networkx/networkx/
[c] https://github.com/snap-stanford/snap-python/

Regarding KGs, distinct definitions co-exist [4,9]. In this work, we stick to the inclusive definition of Hogan *et al.* [16], *i.e.,* we consider a KG to be a graph where nodes represent entities and edges represent relations between these entities. The link between schemas and KGs lies in the fact that schemas are often used to define the structure and semantics of a KG. In other words, a schema defines the vocabulary and rules that govern entities and relationships in a KG. In this view, a KG is a data graph that can be potentially enhanced with a schema [16].

3.2 Overview

From a high-level perspective, the entire PyGraft generation pipeline is depicted in Fig. 1. In particular, Class and Relation Generators are firstly initialized with user-specified parameters, and are then used for building the schema incrementally. The logical consistency of the schema is subsequently checked using HermiT reasoner [12] through the `owlready2`[6] Python library. If the user is also interested in generating a KG based on this schema, the KG Generator is initialized with KG-related parameters, and takes the previously generated schema as input in order to sequentially build the KG. Ultimately, the logical consistency of the resulting KG is assessed with HermiT. More details on the schema-level generation are provided in Sect. 3.3, while Sect. 3.4 describes the KG generation procedure.
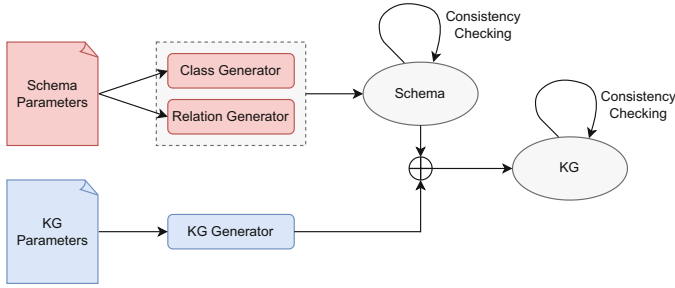
---

[6] https://github.com/pwin/owlready2/.

Fig. 1. PyGraft general overview.

3.3 Schema Generation

Table 2. User-defined parameters for schema and KG generations.

	Parameter	Description
Classes	num_classes	Number of classes
	max_depth	Depth of the class hierarchy
	avg_depth	Average class depth
	inheritance_ratio	Proportion of rdfs:subClassOf
	avg_disjointness	Proportion of owl:DisjointWith
Relations	num_relations	Number of relations
	prop_profiled_relations	Proportion of rdfs:domain and rdfs:range
	relation_specificity	Average depth of rdfs:domain and rdfs:range
	prop_asymmetric	Proportion of owl:AsymmetricProperty
	prop_symmetric	Proportion of owl:SymmetricProperty
	prop_irreflexive	Proportion of owl:IrreflexiveProperty
	prop_reflexive	Proportion of owl:ReflexiveProperty
	prop_transitive	Proportion of owl:TransitiveProperty
	prop_functional	Proportion of owl:FunctionalProperty
	prop_inversefunctional	Proportion of owl:InverseFunctionalProperty
	prop_inverseof	Proportion of owl:inverseOf
	prop_subproperties	Proportion of rdfs:subPropertyOf
Individuals	num_entities	Number of entities
	num_triples	Number of triples
	relation_balance	Relation distribution across triples
	prop_untyped	Proportion of untyped entities
	avg_depth_specific	Average depth of most specific class
	multityping	Whether entities are multi-typed
	avg_multityping	Average number of most-specific classes per entity

The schema generation follows a well-defined series of steps. In particular, the Class Generator in Fig. 1 is initialized first, and generates the class hierarchy

and disjointness axioms as detailed in Algorithm 1. Then, the Relation Generator is initialized and handles the relation generation following the procedure of Algorithm 2. In the following, each algorithm is described step by step.

Class Generation. First, classes are generated based on the user-specified number of classes `num_classes` (lines 1–2). Then, the user-specified `max_depth` parameter is satisfied by taking one class after the other and creating child-parent connections through the `rdfs:subClassOf` assertion (lines 3–7). At this point, the class hierarchy is a purely vertical tree where each node (*i.e.,* class) has exactly one child, except the leaf node. The class hierarchy is then further filled by taking each remaining class sequentially, and connecting it with other classes so that `avg_depth` and `inheritance_ratio` are satisfied (lines 8–13). It is worth mentioning that with some probability α chosen to be moderately low, a freshly picked class can be placed randomly (lines 10–11). This does not necessarily go in the direction of `avg_depth` and `inheritance_ratio` target parameters, but it allows adding stochasticity and realism in the characteristics of the generated class hierarchy. Besides, when a low number of classes are still to be connected, this randomness is deactivated so that each subsequent class connection is in line with `avg_depth` and `inheritance_ratio` fulfilment. Finally, class disjointnesses are added to the schema by picking two classes A and B, ensuring that none of them is a transitive parent or child of the other, and extending class disjointness to their respective children, if any (lines 14–16).

When generating classes, anomalies may occur. For instance, choosing values such that `avg_depth` > `max_depth` triggers an error. In a few other situations, the schema is generated but the user requirements might not be completely fulfilled. This can happen because of competing parameter values. For example, the constraints `num_classes` = 6 (root excluded), `max_depth` = 3, `avg_depth` = 1.5, and `inheritance_ratio` = 2.5 cannot be simultaneously satisfied (Fig. 2). In this situation, the schema is generated with a best-effort strategy, seeking to build a schema with statistics as close as possible to the user requirements.

Relation Generation. Before presenting the procedure for generating relations and their properties, it is worth mentioning that PyGraft allows relations to be described by multiple `OWL` and `RDFS` constructs. This leads to more realistic schemas and KGs, at the expense of higher risk of inconsistency: some property combinations are not logically consistent, *e.g.,* a relation cannot be simultaneously qualified by `owl:ReflexiveProperty` and `owl:IrreflexiveProperty`. Based on the relation properties available in PyGraft (Table 3), all combinations were extracted and for each combination, a new graph was serialized using `rdflib`[7]. This graph contains a unique relation which is qualified by the given property combinations[8]. Based on the simplified schema, the HermiT reasoner

---

[7] https://github.com/RDFLib/rdflib/.

[8] An instance triple should also be added. This is because some property combinations such as `owl:SymmetricProperty` and `owl:AsymmetricProperty` are not flagged as logically inconsistent *per se* in `OWL`. However, a relation qualified by these two properties is not allowed to connect any instances.

performs consistency checking. Finally, a dictionary stores all possible property combinations. Knowing valid and invalid property combinations is necessary for guiding relation property assignment and minimize logical inconsistency likelihood before actually running the reasoner.

In Algorithm 2, this dictionary of valid property combinations is loaded (line 3) just after initializing the number of relations specified by the user (lines 1–2). Then, each relation in the schema is qualified with properties based on a pre-defined order (lines 4–6). For example, `owl:ReflexiveProperty` and `owl:IrreflexiveProperty` are assigned first, then `owl:SymmetricProperty`, etc. These properties are named *attributes* as they characterize a relation *per se*. Next, the *relationship property* `owl:InverseOf` is assigned to relations (line 7), which poses constraints on relation domain and range assignments (line 8), which themselves pose constraints on relation pairings through the `rdfs:subPropertyOf` assertion (line 9), *e.g.,* domain and range of subproperties should not be disjoint with domain and range of superproperties.

In Table 3 are reported the relation properties that the current PyGraft version handles, along with their logical definition and accompanying examples.

Table 3. Relation properties covered by PyGraft.

Property	Definition	Example
`owl:AsymmetricProperty`	$\forall x \forall y : p(x,y) \Rightarrow \neg p(y,x)$	`isParentOf`
`owl:SymmetricProperty`	$\forall x \forall y : p(x,y) \Rightarrow p(y,x)$	`hasSibling`
`owl:ReflexiveProperty`	$\forall x : p(x,x)$	`hasSameColorAs`
`owl:IrreflexiveProperty`	$\forall x : \neg p(x,x)$	`isYoungerThan`
`owl:TransitiveProperty`	$\forall x \forall y \forall z : p(x,y) \wedge p(y,z) \Rightarrow p(x,z)$	`isCheaperThan`
`owl:FunctionalProperty`	$\forall x \forall y \forall z : (p(x,y) \wedge p(x,z)) \Rightarrow y = z$	`hasISBN`
`owl:InverseFunctionalProperty`	$\forall x \forall y \forall z : (p(x,y) \wedge p(z,y)) \Rightarrow x = z$	`isEmailAddressOf`
`owl:InverseOf`	$\forall x \forall y : p(x,y) \Longleftrightarrow q(y,x)$	`owns` $\Longleftrightarrow$ `isOwnedBy`
`rdfs:subPropertyOf`	$\forall x \forall y : p(x,y) \Rightarrow q(x,y)$	`hasMother` $\Rightarrow$ `hasParent`

3.4   Knowledge Graph Generation

In light of PyGraft overview (Fig. 1), this section explores how the KG generator is initialized with user parameters and used in conjunction with any generated schema to build the final KG. As for schema, we provide a step-by-step insight into Algorithm 3. Entities and the KG are generated (lines 1–3) and a schema dedicated to the KG generation is loaded (line 4). Based on `prop_untyped`, entities are assigned a class whose depth in the class hierarchy should be in a close range around `avg_depth_specific` (line 5). Based on `avg_multityping`, several of them are assigned other classes of the same depth, provided that they are not disjoint with any of the specific classes characterizing a given entity (lines 6–7). Then, triples are generated (lines 8–9) in a sequential manner. Namely, unobserved entities in U have sampling priority, to ensure that the required number of entities in the resulting KG is met as fully as possible. Ultimately, checking procedures are performed (line 10) before the KG undergoes logical
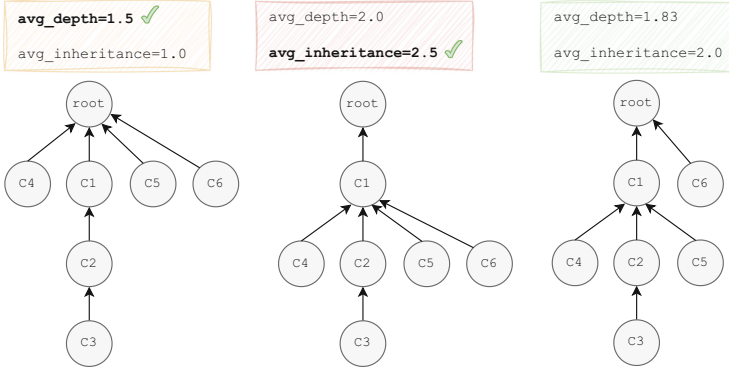
Fig. 2. Potential class hierarchies for the constraints: num_classes = 6, max_depth = 3, avg_depth = 1.5, and inheritance_ratio = 2.5. Left and middle class hierarchies are built with parameter priority. The right class hierarchy is built with a best-effort strategy, without specific parameter privilege.

consistency checking using a semantic reasoner (line 11). If the generated KG is inconsistent, a message warns the user but the KG is stored nevertheless. The user can then choose to restart the generation procedure. It is worth mentioning that the checking procedures (line 10) follow the relevant OWL 2 RL/RDF rules[9]. These rules – in the form of first-order implications – are implemented in PyGraft to ensure consistency pre-checking before the HermiT reasoner is deployed. In addition, the OntOlogy Pitfall Scanner![10] was also used for identifying several checking procedures and for ensuring compliance with common Semantic Web standards.

4 PyGraft in Action

4.1 Efficiency and Scalability Details

In this section, the efficiency and scalability of PyGraft are benchmarked across several schema and graph configurations. Each schema specification reported in Table 4 is paired with each graph specification from Table 5. This leads to 27 distinct combinations.

In particular, schemas from $S1$ to $S3$ are small-sized, schemas from $S4$ to $S6$ are medium-sized, and schemas from $S7$ to $S9$ are of larger sizes (Table 4). For each schema of a given size, the degree of constraints vary as they contain different levels of OWL and RDFS logical constructs. For example, $S1$ has less constraints than $S2$, which itself has less constraints than $S3$. Graph specifications $\mathcal{G}_1$, $\mathcal{G}_2$, and $\mathcal{G}_3$ correspond to small-sized, medium-sized and large-sized graphs, respectively (Table 5).

---

[9] https://www.w3.org/TR/owl2-profiles/#ref-owl-2-rdf-semantics/.
[10] https://oops.linkeddata.es/.

Algorithm 1. Class Generation

1: Initialize the set of unconnected classes: $U = \{C_1, C_2, ..., C_m\}$
2: Initialize the set of linked classes: $L = \emptyset$
3: $C_1 \leftarrow$ U.POP(), L.ADD(C_1)
4: C_1 `rdfs:subClassOf root`
5: **while** `max_depth` is not satisfied **do**   ▷ Fulfilling class hierarchy depth
6:     $C_i \leftarrow$ U.POP(), L.ADD(C_i)
7:     C_i `rdfs:subClassOf` C_{i-1}
8: **while** $U \neq \emptyset$ **do**   ▷ Building class hierarchy
9:     $C_i \leftarrow$ U.POP(), L.ADD(C_i)
10:    **if** random$(0, 1) \leq \alpha$ **then**   ▷ Adding stochasticity
11:        Place it randomly
12:    **else**
13:        Place it s.t. `avg_depth` and `inheritance_ratio` are satisfied
14: **while** `current_disj` < `disj_ratio` **do**   ▷ Adds `owl:disjointWith`
15:    Pick classes A and B s.t. B is neither a parent nor a child of A
16:    Make A and B disjoint and extend disjointness to their respective children

---

Algorithm 2. Relation Generation

1: Initialize the set of unqualified relations: $U = \{R_1, R_2, ..., R_n\}$
2: Initialize the set of qualified relations: $Q = \emptyset$
3: Load compatible patterns
4: **while** $U \neq \emptyset$ and attribute proportions not satisfied **do**   ▷ Adding attributes
5:     $R_i \leftarrow$ U.POP(), Q.ADD(R_i)
6:     QUALIFY(R_i) based on priority order
7: INVERSE_PAIRING(Q)   ▷ Adding `owl:inverseOf`
8: RELATION_PROFILING(Q)   ▷ Adding `rdfs:domain/range`
9: SUBPROPERTY_PAIRING(Q)   ▷ Adding `rdfs:subPropertyOf`

Algorithm 3. Knowledge Graph Generation

1: Initialize the set of unobserved entities: $U = \{E_1, E_2, ..., E_p\}$
2: Initialize the set of observed entities: $O = \emptyset$
3: Initialize the knowledge graph: $\mathcal{KG} = \emptyset$
4: Load the underpinning schema $\mathcal{S}$
5: ASSIGN_CLASS(U)   ▷ Specific class attribution based on `prop_untyped`
6: **if** multityping **then**
7:     COMPLETE_TYPING(U)   ▷ Adding specific classes based on `avg_multityping`
8: **while** $U \neq \emptyset$ and `num_triples` not satisfied **do**
9:     $\mathcal{KG} \leftarrow$ GENERATE_TRIPLES($U, O, \mathcal{S}$)
10: CHECKING_PROCEDURES($\mathcal{KG}$)   ▷ Removing foreseeable inconsistencies
11: REASONING($\mathcal{KG}$)   ▷ HermiT reasoner

For these 27 unique configurations, execution times w.r.t. several dimensions are computed and shown in Fig. 3. Execution times related to the schema generation are omitted as they are negligible. Experiments were conducted on a machine with 2 CPUs Intel Xeon E5-2650 v4, 12 cores/CPU, and 128GB RAM.

It is worth mentioning that the 27 generated KGs were flagged as consistent at the first attempt. The breakdown of time executions differs according to the schema and graph sizes (see Fig. 3). For small graphs, the final consistency checking is the most time-consuming part. For large graphs, triple generation time dominates the rest. As graph sizes increase, all execution times grow but we observe that PyGraft is able to generate consistent KGs quickly, even for large KGs: with our experimental configuration, the total execution time for KGs with 10K entities and 100K triples is roughly 1.5 min. In addition, PyGraft scalability was assessed by asking a KG of 100K entities and 1M triples. On the same machine, it took 47 min to generate such a KG, which was again consistent at the first attempt.

Table 4. Generated schemas. Column headers from left to right: number of classes, class hierarchy depth, average class depth, proportion of class disjointness (cd), number of relations, average depth of relation domains and ranges (rs), and proportions of reflexive (rf), irreflexive (irr), asymmetric (asy), symmetric (sy), transitive (tra), and inverse (inv) relations.

	$|\mathcal{C}|$	MAX($\mathcal{D}$)	AVG($\mathcal{D}$)	cd	$|\mathcal{R}|$	rs	ref	irr	asy	sym	tra	inv
$\mathcal{S}1$	25	3	1.5	0.1	25	1.5	0.1	0.1	0.1	0.1	0.1	0.1
$\mathcal{S}2$	25	3	1.5	0.2	25	1.5	0.2	0.2	0.2	0.2	0.2	0.2
$\mathcal{S}3$	25	3	1.5	0.3	25	1.5	0.3	0.3	0.3	0.3	0.3	0.3
$\mathcal{S}4$	100	4	2.5	0.1	100	2.5	0.1	0.1	0.1	0.1	0.1	0.1
$\mathcal{S}5$	100	4	2.5	0.2	100	2.5	0.2	0.2	0.2	0.2	0.2	0.2
$\mathcal{S}6$	100	4	2.5	0.3	100	2.5	0.3	0.3	0.3	0.3	0.3	0.3
$\mathcal{S}7$	250	5	3.0	0.1	250	3.0	0.1	0.1	0.1	0.1	0.1	0.1
$\mathcal{S}8$	250	5	3.0	0.2	250	3.0	0.2	0.2	0.2	0.2	0.2	0.2
$\mathcal{S}9$	250	5	3.0	0.3	250	3.0	0.3	0.3	0.3	0.3	0.3	0.3

Table 5. Different graph specifications. Column headers from left to right: number of entities, number of triples, proportion of untyped entities, average depth of the most specific class, average number of most-specific classes per multi-typed entity.

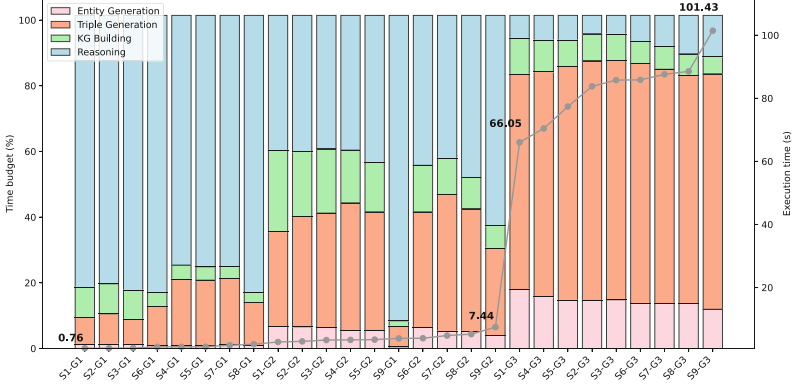|	$|\mathcal{E}|$	$|\mathcal{T}|$	unt	asc	mul
$\mathcal{G}_1$	100	1,000	0.3	2.0	2.0
$\mathcal{G}_2$	1,000	10,000	0.3	2.0	2.0
$\mathcal{G}_3$	10,000	100,000	0.3	2.0	2.0

Fig. 3. Execution time breakdown for each configuration.

4.2   Usage Illustration

In this section, we briefly provide some usage examples to demonstrate how easy it is to use PyGraft. Desired characteristics of the output schema and/or KG can be specified with the path to a `yaml` or `json` configuration file. In the example presented in Listing 1.1, after importing PyGraft, we first generate a `yaml` configuration file in the current working directory. For the sake of simplicity, the generated template is left untouched, *i.e.*, we keep the default parameter values. More advanced usage is provided in the official documentation: https://pygraft.readthedocs.io/en/latest/. Then, we generate both a schema and KG in a single pipeline. The generated resources are subsequently stored in the current working directory.

```python
import pygraft

pygraft.create_yaml_template()
pygraft.generate("template.yml")
```

Listing 1.1. Schema and KG generation with PyGraft.

By default, the generated graph is stored as an `rdf/xml` file. A snippet of a KG generated with PyGraft is provided in Listing 1.2.

```xml
<?xml version="1.0" encoding="utf-8"?>
<rdf:RDF
   xmlns:ns1="http://purl.org/dc/terms/"
   xmlns:owl="http://www.w3.org/2002/07/owl#"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   xmlns:sc="http://pygraf.t/"
>

  <rdf:Description rdf:about="http://pygraf.t/C30">
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
    <rdfs:subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
    <owl:disjointWith rdf:resource="http://pygraf.t/C16"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://pygraf.t/E324">
```

```
  <rdf:type rdf:resource="http://pygraf.t/C4"/>
  <rdf:type rdf:resource="http://pygraf.t/C17"/>
  <schema:R15 rdf:resource="http://pygraf.t/E356"/>
  <schema:R34 rdf:resource="http://pygraf.t/E44"/>
</rdf:Description>
<rdf:Description rdf:about="http://pygraf.t/R34">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#TransitiveProperty"/>
  <rdfs:domain rdf:resource="http://pygraf.t/C17"/>
  <rdfs:range rdf:resource="http://pygraf.t/C17"/>
  <owl:inverseOf rdf:resource="http://pygraf.t/R15"/>
</rdf:Description>
```

Listing 1.2. Excerpt from a generated graph.

5   Discussion

5.1   Potential Uses

As mentioned, PyGraft can be used for generating several schemas and KGs on the fly, thereby facilitating novel approaches and model benchmarking on a wider range of datasets with diversified characteristics. In practice, this can be easily done with inspecting the configuration file we provide as template, and tweaking parameters as one sees fit.

Additionally, PyGraft can be used for generating anonymous data. Let us assume one is working in a data-sensitive field such as medicine or education, characterized by the paucity of readily available data. Thanks to PyGraft, it becomes much more accessible to experiment with sensitiveless testbeds, provided that the user has a rough idea of the scale and characteristics of the resources they want to mimic.

It is worth noting that a vibrant research community aims at incorporating schema-based information into the learning process with KGs [6,19,20]. Using information derived from the schema in addition to the facts contained in the KG aims at building more semantic-aware approaches [17], that are expected to result in more coherent predictions [18] and better predictive accuracy [6,17]. However, it is frequently reported that most KGs do not come with publicly available schemas that would facilitate the development of such schema-aware approaches. For this reason, PyGraft may also be helpful to generate both synthetic schemas with dependent KGs to foster the development of schema-driven, neuro-symbolic approaches, and to run ablation studies on specific schema constructs in isolation.

5.2   Limitations, Sustainability, Maintenance and Future Work

In this section, we discuss the current limitations of PyGraft and elaborate on our sustainability and maintenance plan, as well as future work opportunities.

First, it should be noted that PyGraft relies on `rdflib` for serializing triples. As part of supplementary experiments, we tried to push PyGraft capabilities to its limit by generating very large KGs with >10M entities and triples. In

such cases, the serialization failed. In future work, we will develop independent serialization procedures so that bigger graphs can be generated.

Secondly, recall that our implemented checking procedures limit the likelihood of any inconsistency *before* the DL reasoner is ultimately applied on the generated KG. In case of inconsistencies that would remain undetected by the checking procedures, our actual use of the HermiT reasoner is able to detect such inconsistencies, but not to provide any information on the triples that should be removed so that the KG becomes consistent. In future work, we will implement such a functionality, so that the consistency of the generated KGs can be ensured in a single loop without requiring any input from the user's side.

More generally, we aim at maintaining the PyGraft library, a cornerstone in our commitment to robust software engineering practices and open-source community engagement. Our approach is to release new versions in response to emerging user requirements, ensuring that PyGraft not only meets but anticipates the needs of its users. For instance, a common request is the ability to generate literals – which we seek to achieve in the near future. Furthermore, we recently welcomed the suggestion of offering a hub that would gather different KG profiles generated by the community. To this aim, we welcome all contributions on Github (*e.g.,*, issues, forks, or pull requests) to make PyGraft useful to researchers of a large array of communities. We are dedicated to incorporating methodologies for continuous integration and deployment, fostering a test-driven development environment that guarantees reliability and efficiency. We also want PyGraft to fit within the Semantic Web community and be recognized as a useful tool for researchers and engineers working on KG-based applications.

In future work, we will showcase the impact of PyGraft in real-world use cases. Some relevant scenarios include recommender systems and ontology repairment. In particular, for the latter use-case, we envision the creation of voluntarily imperfect schemas, containing (known) contradictions to resolve. Link prediction also appears as an appropriate task to evaluate PyGraft's usefulness.

6   Conclusion

In this work, we presented PyGraft, a Python tool for generating synthetic schemas and KGs from user requirements. Several `OWL` and `RDFS` constructs are integrated to output realistic KGs that comply with Semantic Web standards. PyGraft allows researchers and practitioners to generate schemas and KGs on the fly, provided minimal knowledge about the desired specifications. Hence, PyGraft can prove useful for various applications. As it allows for generating schemas and KGs of controlled characteristics, it can serve as a well-suited tool for benchmarking novel approaches or models. Being domain-agnostic, PyGraft can be used to generate synthetic schemas and KGs resembling real ones in data-sensitive fields where access or publication of public data is scarce. Due to the richly described generated KGs in terms of `OWL` and `RDFS` constructs, PyGraft may also facilitate the development of schema-driven, neuro-symbolic models.

References

1. Albert, R., Barabási, A.L.: Statistical mechanics of complex networks. Rev. Mod. Phys. **74**, 47–97 (2002). https://doi.org/10.1103/RevModPhys.74.47
2. Angles, R., et al.: The linked data benchmark council: a graph and RDF industry benchmarking effort. SIGMOD Rec. **43**(1), 27–31 (2014). https://doi.org/10.1145/2627692.2627697
3. Bagan, G., Bonifati, A., Ciucanu, R., Fletcher, G.H.L., Lemay, A., Advokaat, N.: gMark: schema-driven generation of graphs and queries. IEEE Trans. Knowl. Data Eng. **29**(4), 856–869 (2017). https://doi.org/10.1109/TKDE.2016.2633993
4. Bonatti, P.A., Decker, S., Polleres, A., Presutti, V.: Knowledge graphs: new directions for knowledge representation on the semantic web (Dagstuhl seminar 18371). Dagstuhl Rep. **8**(9), 29–111 (2018). https://doi.org/10.4230/DagRep.8.9.29
5. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-MAT: a recursive model for graph mining. In: Proceedings of the Fourth SIAM International Conference on Data Mining, Lake Buena Vista, Florida, USA, 22–24 April 2004, pp. 442–446. SIAM (2004). https://doi.org/10.1137/1.9781611972740.43
6. d'Amato, C., Quatraro, N.F., Fanizzi, N.: Injecting background knowledge into embedding models for predictive tasks on knowledge graphs. In: Verborgh, R., et al. (eds.) ESWC 2021. LNCS, vol. 12731, pp. 441–457. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-77385-4_26
7. De Cao, N., Kipf, T.: MolGAN: an implicit generative model for small molecular graphs. In: ICML 2018 Workshop on Theoretical Foundations and Applications of Deep Generative Models (2018)
8. Dettmers, T., Minervini, P., Stenetorp, P., Riedel, S.: Convolutional 2D knowledge graph embeddings. In: Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th Innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, 2–7 February 2018, pp. 1811–1818. AAAI Press (2018)
9. Ehrlinger, L., Wöß, W.: Towards a definition of knowledge graphs. In: Joint Proceedings of the Posters and Demos Track of the 12th International Conference on Semantic Systems - SEMANTiCS2016 and the 1st International Workshop on Semantic Change & Evolving Semantics (SuCCESS'16) co-located with the 12th International Conference on Semantic Systems (SEMANTiCS 2016), Leipzig, Germany, 12–15 September 2016. CEUR Workshop Proceedings, vol. 1695. CEUR-WS.org (2016)
10. ERDdS, P., R&wi, A.: On random graphs I. Publ. math. debrecen **6**(290-297), 18 (1959)
11. Feng, Z., et al.: A schema-driven synthetic knowledge graph generation approach with extended graph differential dependencies (gdd^{x}s). IEEE Access **9**, 5609–5639 (2021). https://doi.org/10.1109/ACCESS.2020.3048186
12. Glimm, B., Horrocks, I., Motik, B., Stoilos, G., Wang, Z.: Hermit: an OWL 2 reasoner. J. Autom. Reason. **53**(3), 245–269 (2014). https://doi.org/10.1007/s10817-014-9305-1
13. Goyal, N., Jain, H.V., Ranu, S.: GraphGen: a scalable approach to domain-agnostic labeled graph generation. In: WWW 2020: The Web Conference 2020, Taipei, Taiwan, 20–24 April 2020, pp. 1253–1263. ACM/IW3C2 (2020). https://doi.org/10.1145/3366423.3380201

14. Gruber, T.R.: Toward principles for the design of ontologies used for knowledge sharing? Int. J. Hum Comput Stud. **43**(5–6), 907–928 (1995). https://doi.org/10.1006/ijhc.1995.1081

15. Guo, Y., Pan, Z., Heflin, J.: LUBM: a benchmark for OWL knowledge base systems. J. Web Semant. **3**(2–3), 158–182 (2005). https://doi.org/10.1016/j.websem.2005.06.005

16. Hogan, A., et al.: Knowledge Graphs. Synthesis Lectures on Data, Semantics, and Knowledge, Morgan & Claypool Publishers (2021). https://doi.org/10.2200/S01125ED1V01Y202109DSK022

17. Hubert, N., Monnin, P., Brun, A., Monticolo, D.: Enhancing knowledge graph embedding models with semantic-driven loss functions. CoRR abs/2303.00286 (2023). https://doi.org/10.48550/arXiv.2303.00286

18. Hubert, N., Monnin, P., Brun, A., Monticolo, D.: Sem@k: is my knowledge graph embedding model semantic-aware? CoRR abs/2301.05601 (2023). https://doi.org/10.48550/arXiv.2301.05601

19. Hubert, N., Paulheim, H., Monnin, P., Brun, A., Monticolo, D.: Schema first! learn versatile knowledge graph embeddings by capturing semantics with machine. CoRR abs/2306.03659 (2023). https://doi.org/10.48550/arXiv.2306.03659

20. Jain, N., Tran, T.K., Gad-Elrab, M.H., Stepanova, D.: Improving knowledge graph embeddings with ontological reasoning. In: Hotho, A., et al. (eds.) ISWC 2021. LNCS, vol. 12922, pp. 410–426. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-88361-4_24

21. Jin, L., Yao, Z., Chen, M., Chen, H., Zhang, W.: A comprehensive study on knowledge graph embedding over relational patterns based on rule learning (2023)

22. Liu, S., Grau, B.C., Horrocks, I., Kostylev, E.V.: Revisiting inferential benchmarks for knowledge graph completion. CoRR abs/2306.04814 (2023). https://doi.org/10.48550/arXiv.2306.04814

23. Melo, A., Paulheim, H.: Synthesizing knowledge graphs for link and type prediction benchmarking. In: Blomqvist, E., Maynard, D., Gangemi, A., Hoekstra, R., Hitzler, P., Hartig, O. (eds.) ESWC 2017. LNCS, vol. 10249, pp. 136–151. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-58068-5_9

24. Palowitch, J., Tsitsulin, A., Mayer, B., Perozzi, B.: GraphWorld: fake graphs bring real insights for GNNs. In: KDD 2022: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, 14–18 August 2022, pp. 3691–3701. ACM (2022). https://doi.org/10.1145/3534678.3539203

25. Park, H., Kim, M.: Trilliong: a trillion-scale synthetic graph generator using a recursive vector model. In: Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, 14–19 May 2017, pp. 913–928. ACM (2017). https://doi.org/10.1145/3035918.3064014

26. Portisch, J., Paulheim, H.: The DLCC node classification benchmark for analyzing knowledge graph embeddings. In: Sattler, U., et al. (eds.) ISWC 2022. LNCS, vol. 13489, pp. 592–609. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-19433-7_34

27. Rossi, A., Firmani, D., Merialdo, P., et al.: Knowledge graph embeddings or bias graph embeddings? A study of bias in link prediction models. In: CEUR Workshop Proceedings, vol. 3034. CEUR-WS (2021)

28. Rossi, A., Matinata, A.: Knowledge graph embeddings: are relation-learning models learning relations? In: Proceedings of the Workshops of the EDBT/ICDT 2020 Joint Conference, Copenhagen, Denmark, 30 March 2020. CEUR Workshop Proceedings, vol. 2578. CEUR-WS.org (2020)

29. Samanta, B., et al.: NEVAE: a deep generative model for molecular graphs. J. Mach. Learn. Res. **21**, 114:1–114:33 (2020)
30. Simonovsky, M., Komodakis, N.: GraphVAE: towards generation of small graphs using variational autoencoders. In: Kurkova, V., Manolopoulos, Y., Hammer, B., Iliadis, L., Maglogiannis, I. (eds.) ICANN 2018. LNCS, vol. 11139, pp. 412–422. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01418-6_41
31. Wang, H., et al.: GraphGAN: graph representation learning with generative adversarial nets. In: Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, 2–7 February 2018, pp. 2508–2515. AAAI Press (2018)
32. You, J., Ying, R., Ren, X., Hamilton, W.L., Leskovec, J.: GraphRNN: generating realistic graphs with deep auto-regressive models. In: Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, 10–15 July 2018. Proceedings of Machine Learning Research, vol. 80, pp. 5694–5703. PMLR (2018)