Accelerating Concept Learning via Sampling

Alkid Baci*
alkid@campus.uni-paderborn.de
Paderborn University
Paderborn, Germany

Stefan Heindorf*
heindorf@uni-paderborn.de
Paderborn University
Paderborn, Germany

ABSTRACT

Node classification is an important task in many fields, e.g., predicting entity types in knowledge graphs, classifying papers in citation graphs, or classifying nodes in social networks. In many cases, it is crucial to explain why certain predictions are made. Towards this end, concept learning has been proposed as a means of interpretable node classification: given positive and negative examples in a knowledge base, concepts in description logics are learned that serve as classification models. However, state-of-the-art concept learners, including EvoLearner and CELOE exhibit long runtimes. In this paper, we propose to accelerate concept learning with graph sampling techniques. We experiment with seven techniques and tailor them to the setting of concept learning. In our experiments, we achieve a reduction in training size by over 90% while maintaining a high predictive performance.

CCS CONCEPTS

• **Computing methodologies** → *Logical and relational learning*; *Description logics*; *Supervised learning by classification.*

KEYWORDS

Knowledge bases; Concept learning; Graph sampling

ACM Reference Format:
Alkid Baci and Stefan Heindorf. 2023. Accelerating Concept Learning via Sampling. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management (CIKM '23), October 21–25, 2023, Birmingham, United Kingdom.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3583780.3615158

1 INTRODUCTION

Node classification is a crucial task in various fields such as knowledge graphs, citation graphs, and social networks. The goal of node classification is to predict the label of each node in a graph given its features and the graph topology. Several concept learning approaches have been proposed to tackle this task [5, 7, 10, 12]: Given a learning problem that consists of positive and negative examples in a knowledge base, concepts in description logics [1] are learned which serve as interpretable classification models. For example, the concept of *Father* in a knowledge base of family relationships might be expressed as $(\neg female) \sqcap (\exists\ hasChild.\top)$. Applying this expression to each node allows binary classification with interpretable concepts that can be easily understood by domain experts. Concept learning has been successfully applied in various domains such as bioinformatics [13], ontology engineering [11], and Industry 4.0 [2].

In this paper, we propose sampling techniques to speed up concept learning. We tailor seven well-known graph sampling algorithms including node samplers, edge-samplers, and exploration-based samplers to the task of concept learning and we evaluate them on the five largest datasets from the SML-Bench [17] framework with the two popular concept learners EvoLearner [5] and CELOE [10, 12]. Our experiments show that we can reduce the concept learner's training data by over 90% while maintaining a high predictive performance. Our contributions can be summarized as follows: (1) We show that concept learners can be executed on a fraction of their original training data to speed up their runtime while maintaining a high-predictive performance. (2) We tailor classic graph-based samplers to the task of concept learning (Section 3.2). (3) We investigate the trade-off between sample size and predictive performance (Section 4.2). (4) We investigate and compare 14 different samplers for concept learning (Section 4.3).

2 RELATED WORK

Concept learning. DL-Learner [10, 12] is a popular concept learning suite for learning concepts in description logics. Its best algorithm CELOE is based on inductive logic programming and refines the most general top concept ($\top$) step by step by means of a refinement operator. The more recent Ontolearn library encompasses the state-of-the-art concept learner EvoLearner [5] which is based on evolutionary algorithms and has been found to outperform CELOE, e.g., on the SML benchmarking framework [17]. Neural Class Expression Synthesis [7, 8] considers concept learning as a translation problem and "translates" positive/negative examples to a class expression. While it can predict a class expression in milliseconds, its training data generation relies on search-based concept learners like EvoLearner and CELOE and is the computational bottleneck of the approach. In this paper, we show that search-based concept learners can be trained on a fraction of their original training data while maintaining a high predictive performance.

Graph sampling. Graph sampling techniques select a subset of nodes and edges from an original graph generating a smaller graph that retains important properties of the original graph [6]. Leskovec and Faloutsos [14] have shown that sampling techniques can efficiently and accurately sample real-world graphs, including social networks and web crawls. Sampling techniques can be categorized as node-based, edge-based, and exploration-based. In this paper, we consider techniques from all three categories. Our samplers are

*Both authors contributed equally to this research.

CIKM '23, October 21–25, 2023, Birmingham, United Kingdom

ACM ISBN 979-8-4007-0124-5/23/10…$15.00
https://doi.org/10.1145/3583780.3615158

Algorithm 1 Induce knowledge base from knowledge graph

Input: Knowledge base $\mathcal{K}$ represented as an OWL ontology, knowledge graph $\mathcal{G}$ with triples $(h, r, t) \in \mathcal{E} \times \mathcal{R} \times \mathcal{E}$
Output: The new sampled knowledge base
Function: sample_KB($\mathcal{K}, \mathcal{G}$):

for $ind \in \mathcal{K}.individuals()$ **do**
  if $ind \notin \mathcal{G}.subjects()$ **then**
    $\mathcal{K}.remove\_individual(ind)$ ▷ removes all links, too
  else
    for $prop \in ind.properties()$ **do**
      for $value \in ind.property\_values(prop)$ **do**
        if $(ind, prop, value) \notin \mathcal{G}$ **then**
          $ind.remove\_property\_value(prop, value)$
        end if
      end for
      if $ind.property\_values(prop)$ is empty **then**
        $ind.remove\_property(prop)$
      end if
    end for
  end if
end for
return $\mathcal{K}$

inspired by the little ball of fur (LBoF) library [16]. However, the library is limited to undirected and connected NetworkX and NetworkKit graphs. Therefore, we adapt samplers from this library to sample knowledge bases in the web ontology language OWL and we reimplement their samplers for the owlready2 [9] and Ontolearn [5] libraries. Moreover, we tailor the samplers to the task of concept learning, yielding our "learning problem-centered" samplers.

3 SAMPLING

In this section, we introduce our samplers to sample OWL knowledge bases, which serve as input for concept learners. All our samplers follow the same basic structure: (1) We represent an OWL knowledge base as a knowledge graph in the same way as previously done by EvoLearner [5], (2) we execute a sampling algorithm on the knowledge graph, (3) we return the OWL knowledge base induced by the sampled knowledge graph. Algorithm 1 shows how a sampled knowledge graph induces an OWL knowledge base. A knowledge graph $\mathcal{G}$ is defined as a set of triples $\mathcal{G} = \{(h, r, t)\} \subset (\mathcal{E} \times \mathcal{R} \times \mathcal{E})$ where $\mathcal{E}$ denotes the set of entities and $\mathcal{R}$ the set of relations between entities—known as object properties in OWL.

3.1 Basic Samplers

First, we describe the basic samplers that we employ, before we describe how we tailor them to the task of concept learning.

Random nodes (RN). Entities are selected uniformly at random from the set of all entities $\mathcal{E}$ in the knowledge graph. The knowledge graph induced by the selected entities is returned.

Random edges (RE). First, an entity $h' \in \mathcal{E}$ is picked uniformly at random. Then a triple (h', r', t') with h as subject is chosen uniformly at random from $\{(h, r, t) \in \mathcal{G} : r \in \mathcal{R}, t \in \mathcal{E}, h = h'\}$.

Algorithm 2 Random walk sampler

Input: Knowledge graph $\mathcal{G}$ with triples $(h, r, t) \in \mathcal{E} \times \mathcal{R} \times \mathcal{E}$, number of nodes to sample N
Output: The new sampled knowledge graph
Function: random_walk_sampling($\mathcal{G}, N$):

$\mathcal{G}' \leftarrow \emptyset$; $\mathcal{E}' \leftarrow \emptyset$ ▷ new empty KG
$e \leftarrow$ select random entity from $\mathcal{E}$; $\mathcal{E}' \leftarrow \mathcal{E}' \cup \{e\}$
while $|\mathcal{E}'| < N$ **do**
  $S \leftarrow \{(h, r, t) \in \mathcal{G} | h = e, r \in \mathcal{R}, t \in \mathcal{E}\}$
  if $S = \emptyset$ **then**
    $e \leftarrow$ select random entity from $\mathcal{E}$; $\mathcal{E}' \leftarrow \mathcal{E}' \cup \{e\}$
  else
    Select (h', r', t') from S uniformly at random
    $\mathcal{E}' \leftarrow \mathcal{E}' \cup \{t'\}$; $\mathcal{G}' \leftarrow \mathcal{G}' \cup \{(h', r', t')\}$
    $e \leftarrow t'$
  end if
end while
return $\mathcal{G}'$

Random walks (RW) [4, 16]. The Random Walk (RW) sampler (see Algorithm 2) randomly "walks" along the edges of the knowledge graph. In each step, for a starting entity $e \in \mathcal{E}$, it picks a triple (h', r', t') from $S := \{(h, r, t) \in \mathcal{K} : h = e, r \in \mathcal{R}, t \in \mathcal{E}\}$ uniformly at random and "walks" along the edge to entity t'. The next step continues with t' as a start entity. If an entity does not have an outgoing edge, the walker continues from a new entity that is chosen uniformly at random. While random walk sampling is a simple and efficient exploration-based strategy, it suffers from at least two problems that are tackled in the following by more sophisticated samplers: (1) a random walker might get stuck in a loop and never reaches parts of the graph that are far away or disconnected, (2) an outgoing edge is always selected uniformly at random which might not be the best choice.

Random walks with jumps (RWJ). This sampler is similar to RW. The only difference is that in a step, it either follows the RW algorithm or jumps to an arbitrary node. The decision is made randomly according to a hyperparameter called "jump probability." Unlike RW, the jumps allow the walker to escape from loops or small connected components of the graph.

Random walk with jumps and prioritization (RWJP). This sampler does not select the neighbors uniformly at random but prioritizes their selection based on the page rank [3] $PR(e)$ of each node e. A triple from S is selected according to the probability

$$P(\text{select } (h', r', t') \in S) = \frac{PR(t')}{\sum_{(h', r, t) \in S} PR(t)} .$$

Forest fire (FF) [14–16]. Forest fire simulates a forest fire spreading through the nodes of a graph. The algorithm uses a parametrized stochastic version of breadth-first-search, and has three hyperparameters: (1) the burning probability p which determines the likelihood that a node will be "burned" and added to the sample, (2) the maximum size of the visited node backlog, and (3) the restart hop size which determines the number of nodes that will be chosen to continue the burning process once the current set of nodes in the burning process has been exhausted.

3.2 Learning Problem-centered Samplers

Given positive E^+ and negative example entities E^- in a knowledge base, the task of a concept learner is to learn a concept in description logics such that many of the positive entities are entailed by this concept and many of the negative entities are not [5]. Let $E = E^+ \cup E^-$ be all example entities—dubbed learning problem (LP) entities. We exploit the LP entities to improve our basic samplers by increasing the chances of the LP entities being sampled. We call our modifications learning problem-centered (LPC) and describe them below. While our LPC modifications of random node and edge samplers focus on the close neighborhood of LP entities, our LPC modifications of random walk and forest fire samplers may explore entities further away from LP entities.

Random nodes and edges (RN-LPC, RE-LPC). The LPC modification for RN aims to preserve the local neighborhood of the LP entities. The local k-hop neighborhood of LP entities is fully preserved and nodes are only sampled from the $k+1$-hop neighborhood. Given the number of nodes N that are to be sampled, k is chosen such that the size of the k-hop neighborhood is smaller than N and the size of the $k+1$-hop neighborhood is larger than N. Similarly, RE-LPC takes all edges from the k-hop neighborhood and samples edges from the $k+1$-hop neighborhood of learning problem entities until the specified number of nodes has been reached.

Random walks (RW-LPC, RWJ-LPC, RWP-LPC, RWJP-LPC). The idea behind the LPC modifications of RW, RWJ, RWP, and RWJP is as follows: The *walker* starts from a LP entity $e \in E$ and continues as normal. When the *walker* reaches a node without neighbors, it will restart again from one of the LP entities in E which is chosen uniformly at random. For RWJ and RWJP, the *walker* jumps only to the LP entities. As it may happen with this approach that there are not enough nodes to sample around the LP neighborhood, we fall back to the original version of the sampler without the LPC modification if no new node was added to the sample for a long time. In our experiments, we chose 5% of the graph size as threshold, i.e., we count the number of steps no new node was added and when this number exceeds 5% of the entities in the graph, we do not employ the LPC modification anymore. This ensures that the algorithm terminates and the sample will contain the number of nodes N specified by the user.

Forest fire (FF-LPC). Our LPC modification of FF works as follows: We add all LP entities to the queue and the FF algorithm starts by burning the first entity from the queue. Once a burning process finishes for a LP entity, the algorithm will continue with the next LP entity in the queue. In contrast to some of our other samplers, FF will explore a larger neighborhood of the first LP entities and there is no guarantee that all LP entities are considered before the specified sample size has been reached.

4 EVALUATION

After giving a brief introduction to our evaluation setup, we investigate two research questions: (1) What is the trade-off between sample size and predictive performance? (2) What is the best sampler maintaining a high predictive performance of concept learners?

Table 1: Overview of the datasets in terms of number of instances, axioms, atomic concepts, properties, expressiveness and positive and negative examples (E^+, E^-) [5].

Dataset	Instances	Axioms	Atomic Concepts	Object Prop.	Data Prop.	Expressiveness	$\lvert E^+\rvert$	$\lvert E^-\rvert$
Carcinogenesis	22,372	74,566	142	4	15	$\mathcal{ALC}(\mathcal{D})$	162	136
Hepatitis	6,812	79,935	14	5	12	$\mathcal{ALE}(\mathcal{D})$	206	294
Mutagenesis	14,145	62,066	86	5	6	$\mathcal{AL}(\mathcal{D})$	13	29
NCTRER	10,209	103,070	37	9	50	$\mathcal{ALCI}(\mathcal{D})$	131	93
Premier League	11,859	2,155,439	10	14	202	$\mathcal{ALEH}(\mathcal{D})$	40	41

4.1 Evaluation Setup

Datasets. We conduct our evaluation on the large SML-Bench datasets [17] shown in Table 1. We omit smaller datasets as they yielded trivial results and the goal of sampling is to reduce the size of large datasets.

Concept learners. We employ the state-of-the-art concept learners *EvoLearner* [5] and *CELOE* [10, 12] to measure their performance on the sampled datasets and thus evaluate the samplers for the task of concept learning. We picked EvoLearner as an example of an evolutionary algorithm and CELOE as an example of an inductive logic programming approach that is based on refinement operators. Both concept learners were run with their default hyperparameters which were previously found to work well on the SML-Bench datasets [5].

Robustness and evaluation metrics. For each setting, i.e., each combination of sample size, sampler, and dataset, we perform 100 different samples. On each sample, we train the concept learner and measure the concept learner's performance on the original, unsampled dataset. A good sampler should generate a good training set such that a concept learner trained on it achieves a high performance. Following Heindorf et al. [5], we measure performance in terms of F_1-measure that indicates how well the learned concept covers positive examples, but not negative examples. If a basic sampler does not sample any positive example, we consider half of the sampled nodes to be positive in order to guarantee that the concept learner returns at least some result.

Hyperparameters. All our samplers allow specifying the number of nodes to be sampled. Moreover, we employ the default hyperparameters from the little-ball-of-fur library:[1] Random walk jump utilizes the random jump probability that we set to 0.1. Forest fire utilizes the burning probability $p = 0.4$, a visited node backlog size of 100, and the restart hop size 10.

Implementation. We implemented our samplers within the Ontolearn library [5], which is based on owlready2 [9]. Our code is publicly available.[2]

4.2 Sample Size vs. Predictive Performance

Figure 1 (top) investigates the trade-off between sample size and runtime where the sample size is measured as percentage of nodes from the original dataset and the runtime is measured in seconds.

[1] https://little-ball-of-fur.readthedocs.io/en/latest/
[2] https://github.com/alkidbaci/OntoSample

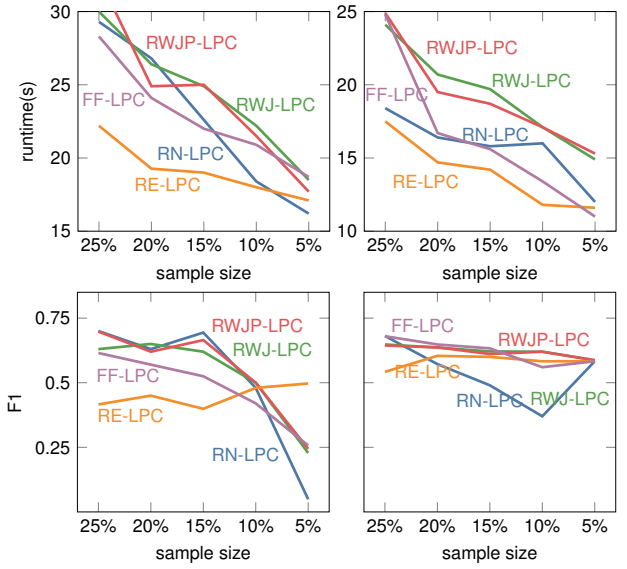

Figure 1: Average runtime in seconds (top) and F_1-measure (bottom) of EvoLearner on *Carcinogenesis* (left) and *Hepatitis* (right) depending on sampler and sample size.

We can observe that the runtime of EvoLearner decreases approximately linearly with sample size for the Carcinogenesis and Hepatitis datasets—and for other datasets and CELOE, too, which is not shown in the figure due to space constraints.

Figure 1 (bottom) investigates the trade-off between sample size and predictive performance. For the Carcinogenesis dataset (left), most samplers maintain a high predictive performance until a sample size of about 10–15% when the performance starts to decrease; for Hepatitis (right), the performance remains high throughout all sample sizes and only slightly decreases. We attribute this effect to the complexity of the learning problems. It was previously shown that good solutions for Carcinogenesis require longer concepts than good solutions for Hepatitis [5], i.e., Carcinogenesis is a more difficult learning problem. Moreover, we observed that the lengths of the predicted concepts decrease with sample size until only trivial solutions are predicted in the end.

4.3 Evaluation of Samplers

Table 2 compares different samplers in terms of EvoLearner's and CELOE's performance for a sample size of 10%. For comparison, we also show the performances when training on the full datasets, i.e., a sample size of 100%. The results show that for many datasets (Mutagenesis, NCTRER, Premier League) a perfect solution can still be learned with only 10% of the original dataset—in particular when using our learning problem-centered (LPC) samplers. If datasets have a small number of positive/negative examples (Mutagenesis, Premier League), LPC samplers tend to outperform non-LPC samplers. For Carcinogenesis and Hepatitis, the best sampler depends on the dataset and concept learner. For Carcinogenesis, the classic samplers perform considerably better than the LPC samplers for Evolearner whereas the LPC samplers perform better in the case of

Table 2: Comparison of samplers in terms of EvoLearner's and CELOE's F_1-measure on the full datasets and 10% thereof.

Samplers	Carcinog.	Hepatitis	Mutag.	NCTRER	Prem. League
F_1-measure of EvoLearner for 100% sample size					
No sampler	0.71 ± 0.01	0.75 ± 0.02	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00
F_1-measure of EvoLearner for 10% sample size					
RN	0.59 ± 0.20	0.45 ± 0.13	0.56 ± 0.32	**0.99 ± 0.01**	0.76 ± 0.22
RE	**0.68 ± 0.02**	0.59 ± 0.02	**0.85 ± 0.19**	**0.99 ± 0.01**	0.66 ± 0.04
RW	0.65 ± 0.05	no result	no result	0.96 ± 0.9	0.97 ± 0.03
RWJ	0.64 ± 0.10	0.58 ± 0.09	0.57 ± 0.26	0.97 ± 0.08	0.97 ± 0.05
RWP	0.65 ± 0.09	no result	no result	0.98 ± 0.06	0.93 ± 0.10
RWJP	0.63 ± 0.11	0.59 ± 0.10	0.56 ± 0.21	0.98 ± 0.04	0.91 ± 0.14
FF	0.56 ± 0.11	**0.62 ± 0.11**	0.49 ± 0.16	0.73 ± 0.19	**0.99 ± 0.01**
RN-LPC	0.48 ± 0.22	0.37 ± 0.14	**1.00 ± 0.00**	**1.00 ± 0.00**	0.99 ± 0.01
RE-LPC	0.48 ± 0.32	0.58 ± 0.00	**1.00 ± 0.00**	**1.00 ± 0.00**	0.99 ± 0.01
RW-LPC	**0.50 ± 0.28**	no result	no result	**1.00 ± 0.00**	**1.00 ± 0.00**
RWJ-LPC	**0.50 ± 0.28**	**0.62 ± 0.02**	**1.00 ± 0.00**	**1.00 ± 0.00**	**1.00 ± 0.00**
RWP-LPC	0.48 ± 0.29	no result	no result	**1.00 ± 0.00**	**1.00 ± 0.00**
RWJP-LPC	0.50 ± 0.29	**0.62 ± 0.02**	**1.00 ± 0.00**	**1.00 ± 0.00**	**1.00 ± 0.00**
FF-LPC	0.42 ± 0.28	0.56 ± 0.09	**1.00 ± 0.00**	**1.00 ± 0.00**	0.99 ± 0.01
F_1-measure of CELOE for 100% sample size					
No sampler	0.71 ± 0.00	0.70 ± 0.01	0.92 ± 0.00	0.98 ± 0.00	0.66 ± 0.00
F_1-measure of CELOE for 10% sample size					
RN	0.46 ± 0.25	0.43 ± 0.17	0.35 ± 0.24	0.97 ± 0.05	**0.66 ± 0.00**
RE	0.54 ± 0.22	0.58 ± 0.03	**0.79 ± 0.20**	**0.98 ± 0.04**	0.64 ± 0.01
RW	0.45 ± 0.25	no result	no result	0.93 ± 0.12	**0.66 ± 0.00**
RWJ	0.48 ± 0.22	0.55 ± 0.12	0.32 ± 0.21	0.95 ± 0.09	**0.66 ± 0.00**
RWP	0.47 ± 0.24	no result	no result	0.98 ± 0.06	**0.66 ± 0.00**
RWJP	0.42 ± 0.26	0.56 ± 0.11	0.33 ± 0.19	0.96 ± 0.08	**0.66 ± 0.00**
FF	**0.56 ± 0.12**	**0.63 ± 0.07**	0.15 ± 0.19	0.60 ± 0.18	**0.66 ± 0.00**
RN-LPC	**0.62 ± 0.14**	0.53 ± 0.12	**1.00 ± 0.00**	**0.98 ± 0.00**	**0.66 ± 0.00**
RE-LPC	0.33 ± 0.30	0.57 ± 0.04	0.98 ± 0.01	**0.98 ± 0.00**	0.66 ± 0.23
RW-LPC	0.55 ± 0.23	no result	no result	**0.98 ± 0.00**	**0.66 ± 0.00**
RWJ-LPC	0.54 ± 0.26	**0.62 ± 0.04**	**1.00 ± 0.00**	**0.98 ± 0.00**	**0.66 ± 0.00**
RWP-LPC	0.59 ± 0.21	no result	no result	**0.98 ± 0.00**	**0.66 ± 0.00**
RWJP-LPC	0.51 ± 0.26	0.61 ± 0.04	**1.00 ± 0.00**	**0.98 ± 0.00**	**0.66 ± 0.00**
FF-LPC	0.42 ± 0.27	0.62 ± 0.05	0.98 ± 0.02	**0.98 ± 0.00**	**0.66 ± 0.00**

CELOE. When a sampler gets stuck and does not reach the specified number of nodes to be sampled, we indicate this with "no results" in the table. A better predictive performance often goes hand in hand with a lower standard deviation.

A manual error analysis revealed that sampling makes it particularly challenging to learn concepts that contain cardinality restrictions, such as "≤ 3 `hasAtom`.$\top$" which denotes "nodes that are connected via the `hasAtom` relation to at most $n = 3$ other nodes." On the sampled graphs, often lower thresholds n are learned than on the full graphs. We leave it to future work to develop tailored solutions to this problem.

5 CONCLUSIONS

In this paper, we propose the usage of graph samplers to speed up concept learners on knowledge bases. We experiment with classic graph samplers such as random walks and we tailor them to concept learning by emphasizing the positive and negative examples in the sampling process—dubbed learning problem-centered samplers. Our results show that the concept learner's training data can often be reduced while maintaining a high predictive performance. In the future, we will apply our sampling techniques to large-scale knowledge graphs such as Wikidata, DBpedia, and YAGO.

REFERENCES

[1] Franz Baader, Ian Horrocks, and Ulrike Sattler. 2004. *Description logics.* Springer.

[2] Simon Bin, Patrick Westphal, Jens Lehmann, and Axel Ngonga. 2017. Implementing scalable structured machine learning for big data in the SAKE project. In *IEEE BigData*. IEEE Computer Society, 1400–1407.

[3] Sergey Brin and Lawrence Page. 1998. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Comput. Networks* 30, 1-7 (1998), 107–117.

[4] Minas Gjoka, Maciej Kurant, Carter T. Butts, and Athina Markopoulou. 2010. Walking in Facebook: A Case Study of Unbiased Sampling of OSNs. In *INFOCOM*. IEEE, 2498–2506.

[5] Stefan Heindorf, Lukas Blübaum, Nick Düsterhus, Till Werner, Varun Nandkumar Golani, Caglar Demir, and Axel-Cyrille Ngonga Ngomo. 2022. EvoLearner: Learning Description Logics with Evolutionary Algorithms. In *WWW*. ACM, 818–828.

[6] Pili Hu and Wing Cheong Lau. 2013. A Survey and Taxonomy of Graph Sampling. *CoRR* abs/1308.5865 (2013).

[7] N'Dah Jean Kouagou, Stefan Heindorf, Caglar Demir, and Axel-Cyrille Ngonga Ngomo. 2023. Neural Class Expression Synthesis. In *ESWC (Lecture Notes in Computer Science, Vol. 13870)*. Springer, 209–226.

[8] N'Dah Jean Kouagou, Stefan Heindorf, Caglar Demir, and Axel-Cyrille Ngonga Ngomo. 2023. Neural Class Expression Synthesis in ALCHIQ(D). In *ECML/PKDD (Lecture Notes in Computer Science)*. Springer.

[9] Jean-Baptiste Lamy. 2017. Owlready: Ontology-oriented programming in Python with automatic classification and high level constructs for biomedical ontologies. *Artif. Intell. Medicine* 80 (2017), 11–28.

[10] Jens Lehmann. 2009. DL-Learner: Learning Concepts in Description Logics. *J. Mach. Learn. Res.* 10 (2009), 2639–2642.

[11] Jens Lehmann, Sören Auer, Lorenz Bühmann, and Sebastian Tramp. 2011. Class expression learning for ontology engineering. *J. Web Semant.* 9, 1 (2011), 71–81.

[12] Jens Lehmann and Pascal Hitzler. 2010. Concept learning in description logics using refinement operators. *Mach. Learn.* 78, 1-2 (2010), 203–250.

[13] Jens Lehmann and Johanna Völker. 2014. *Perspectives on Ontology Learning*. Studies on the Semantic Web, Vol. 18. IOS Press.

[14] Jure Leskovec and Christos Faloutsos. 2006. Sampling from large graphs. In *KDD*. ACM, 631–636.

[15] Jure Leskovec, Jon M. Kleinberg, and Christos Faloutsos. 2005. Graphs over time: densification laws, shrinking diameters and possible explanations. In *KDD*. ACM, 177–187.

[16] Benedek Rozemberczki, Oliver Kiss, and Rik Sarkar. 2020. Little Ball of Fur: A Python Library for Graph Sampling. In *CIKM*. ACM, 3133–3140.

[17] Patrick Westphal, Lorenz Bühmann, Simon Bin, Hajira Jabeen, and Jens Lehmann. 2019. SML-Bench - A benchmarking framework for structured machine learning. *Semantic Web* 10, 2 (2019), 231–245.