Knowledge-Based Systems 238 (2022) 107975

Contents lists available at ScienceDirect

Knowledge-Based Systems

journal homepage: www.elsevier.com/locate/knosys

Automatic extraction of shapes using sheXer

Daniel Fernandez-Álvarez *, Jose Emilio Labra-Gayo, Daniel Gayo-Avello

Department of Computer Science, University of Oviedo, Oviedo, Spain

ARTICLE INFO

Article history:
Received 19 May 2021
Received in revised form 18 October 2021
Accepted 13 December 2021
Available online 17 December 2021

Keywords:
Knowledge Graph
RDF
ShEx
SHACL
Automatic extraction

ABSTRACT

There is an increasing number of projects based on Knowledge Graphs and SPARQL endpoints. These SPARQL endpoints are later queried by final users or used to feed many different kinds of applications. Shape languages, such as ShEx and SHACL, have emerged to guide the evolution of these graphs and to validate their expected topology. However, authoring shapes for an existing knowledge graph is a time-consuming task. The task gets more challenging when dealing with sources, possibly maintained by heterogeneous agents. In this paper, we present sheXer, a system that extracts shapes by mining the graph structure. We offer sheXer as a free Python library capable of producing both ShEx and SHACL content. Compared to other automatic shape extractors, sheXer includes some novel features such as shape inter-linkage and computation of big real-world datasets. We analyze the features and limitations w.r.t. performance with different experiments using the English chapter of DBpedia.

1. Introduction

The interest in Knowledge Graphs (KGs) is rapidly growing, especially in the last decade. Insightful examples of big and open KGs are DBpedia [1], Wikidata [2], or YAGO [3]. These projects are published online and allow individuals and companies to make use of their content. Also, many big companies use their own private or semi-private KGs for a wide variety of purposes, including Google, Amazon, Facebook, and Microsoft, among others [4]. The most common way to build and expose those KGs is using W3C standards such as Resource Description Language (RDF) and SPARQL.

In such a context, mechanisms to validate the structure or assist the maintenance of KGs are needed. Ontologies can be used to define restrictions w.r.t. property and class usage. However, those restrictions defined by ontologies are frequently not enough to model every schema feature in a given KG. For example, a KG may need to combine different ontologies using some restrictions not defined in the actual ontologies. It may also need to define extra restrictions over a specific ontology element. To overcome this, shape languages such as ShEx (Shape Expressions) [5] and SHACL (Shapes Constraint Language) [6] have been proposed. Although these two languages are not fully equivalent [7], both can provide mechanisms to validate and document the expected topology of a KG.

Shape languages are structured around the concept of *shape*. Shapes describe how the different types of nodes within a KG are supposed to be connected with other nodes. In Fig. 1, we show a toy example of a shape describing the expected topology of a *User* node in SHACL (A) and ShEx (B).[1] A node conforming with the shape *:User* should have exactly a *schema:name*[2] of *xsd:string* type and, optionally, it can have a *birth:date* of *xsd:date* type. The shape *:User* can be used to validate whether the nodes that represent users conform with their expected topology in the context of a given KG.

Usually, the shapes are handcrafted by domain experts. Those shapes can guide content modifications or be used to check the KG's correctness with automatic validators [8]. However, producing and maintaining shapes is time-consuming. The task becomes more challenging when dealing with big and heterogeneous KGs, possibly with evolving schemata, and maintained by different agents.

Automatic shape extractors were proposed to overcome this issue. Automatic extractors can produce shapes by mining KGs or exploring the ontologies used in those KGs, requiring few or no human intervention. As this is a relatively new problem, few automatic extractors have been proposed. Many of them are prototypes that are not accurate enough yet, or have scalability issues to deal with big datasets. In many cases, this prevents both users and data-maintainers from using automatic extractors.

In this paper, we present sheXer, an automatic shape extractor based on graph mining. sheXer can produce ShEx and SHACL content, and it allows to tune the extraction process with multiple

* Corresponding author.
E-mail addresses: fernandezalvdaniel@uniovi.es (D. Fernandez-Álvarez), labra@uniovi.es (J.E. Labra-Gayo), dani@uniovi.es (D. Gayo-Avello).

https://doi.org/10.1016/j.knosys.2021.107975
0950-7051/

[1] You can view and modify this example in the following link: http://rdfshape.weso.es/link/16182267731 Accessed in 2021/10/15.

[2] All the prefixes used in this paper are commonly used and can be solved using the on-line tool http://prefix.cc/.

A

```
@prefix :       <http://example.org/> .
@prefix schema: <http://schema.org/> .
@prefix sh:     <http://www.w3.org/ns/shacl#> .
@prefix xsd:    <http://www.w3.org/2001/XMLSchema#> .

:User a             sh:NodeShape ;
      sh:property   [ a           sh:PropertyShape ;
                      sh:datatype xsd:string ;
                      sh:maxCount 1 ;
                      sh:minCount 1 ;
                      sh:path     schema:name      ] ;
      sh:property   [ a           sh:PropertyShape ;
                      sh:datatype xsd:date ;
                      sh:maxCount 1 ;
                      sh:path     schema:birthDate  ] .
```

B

```
PREFIX : <      http://example.org/>
PREFIX schema: <http://schema.org/>
PREFIX xsd:  <http://www.w3.org/2001/XMLSchema#>

:User {
  schema:name         xsd:string  ;
  schema:birthDate    xsd:date ?  ;
}
```

Fig. 1. Example of :User shape. (A) SHACL (turtle syntax). (B) ShEx (ShExC syntax).

configuration parameters. sheXer was suggested as a theoretical idea [9] and proposed as a demo [10]. Nowadays, we offer a public Python library with a mature implementation of sheXer.[3] This library has already been used in several scientific works [11, 12]. Also, sheXer is integrated with external tools relevant to the Linked Data community, such as WikidataIntegrator.[4] sheXer has some unique features compared to other existing automatic shape extractors, including:

- It performs shape inter-linkage, i.e., it can produce constraints that refer to other shape labels. Most of the alternatives use less specific macros instead, such as IRI.[5]
- It uses an iterative approach that allows for computing big datasets. There is not a strict relation between the size of the computed KG and the memory consumption. There is no need to keep in memory the whole graph at any point in the process.
- It assigns a trustworthiness score to each one of the inferred constraints. Although this feature is partially shared with Shape Designer [13], sheXer uses this score to sort, filter, or merge some constraints while performing the shape extraction.

In this paper, we formalize sheXer's algorithm and workflow. Also, we perform several experiments to analyze the execution time and memory consumption of our proposal. To the best of our knowledge, the only other automatic shape extractor with a published performance study is SHACLearner [14]. Its authors perform experiments against the English chapter of DBpedia, and they provide details about execution time, but not memory consumption.

In Section 2, we offer an overview of our approach, its architecture, and its algorithms. In Section 3, we detail the conditions of our experiments and then present and discuss the obtained results. In Section 4, we describe other existing proposals to perform automatic shape extraction. Finally, in Section 5, we enumerate the conclusions and future lines of our work.

2. System description

sheXer has a modularized architecture that allows it to adapt to many different scenarios. Each module expects an input and produces an output. This output may be presented to the final user or consumed by some other module. sheXer's architecture is shown in Fig. 2. Its work-flow consist of the following steps:

- The user chooses an input. This includes target RDF source, target shapes to extract, and possibly some configuration parameters.
- The *Instance Tracker* determines which nodes of the target source will be used to extract which shapes. It consumes relevant triples from the *Graph Iterator*.
- The *Feature Tracker* within the *Shape extractor* generates a set of candidate constraints associated with each shape. It uses the information produced by the *Instance Tracker* and consumes the graph's content using the *Graph Iterator*.
- The *Shape Adapter* filters, adapts, merges, and sorts candidate constraints according to the configuration settings, so a final set of constraints is produced.
- The *Shape Serializer* turns the in-memory shapes produced by the *Shape Adapter* into the content chosen by the user.

The current implementation of sheXer includes several versions of most of the modules. In the following sections, we will detail the structure and mission of each module.

2.1. Graph iterator

There are two phases of the workflow in which the target RDF source needs to be parsed: (1) determining which nodes will be used to build which shapes, and (2) building the abstract profile for each shape. The target RDF content is served by the Graph Iterator (GI) to perform those actions. Regardless of the type of input, the mission of the GI is to retrieve relevant triples for the process in an iterative way. Whenever it is possible, the GI avoids placing in memory the entire target content at a time.

The internal details of this software piece may change according to the kind of input. For example, an RDF input based on local text files can be trivially served by reading small file chunks and processing triple by triple. The input could be instead part of the content exposed in some remote SPARQL endpoint. The sheXer library includes GI implementations to deal with several input cases, including the ones already mentioned.

2.2. Instance tracker

The mission of the Instance Tracker (IT) consists of determining which nodes (aka instances) will be used to build which shapes. The nature of this process can vary depending on the type of input and the target shapes and instances.

For example, when sheXer receives a shape map[6] to link a shape with some instances, it could be necessary to execute a SPARQL query to find those shapes. When the target shapes are specified using a list of target classes, the process consists of finding the instances of those classes. sheXer also lets the user to

[3] The source code, installing instructions and documentation is publicly available: https://github.com/DaniFdezAlvarez/shexer/tree/1.3.0 Accessed in 2021/10/15.

[4] WikidataIntegrator is a well-known tool in Wikidata community. It allows to interact with the KG using Python bots https://github.com/SuLab/WikidataIntegrator Accessed in 2021/10/15.

[5] In ShEx, the macro IRI stands for any node which is an IRI. In SHACL, sh:IRI has an equivalent meaning.

[6] sheXer supports shape maps, which are the standard ShEx mechanism to link nodes with the shape that they should conform with. The syntax of shape maps is specified in the following link: http://shex.io/shape-map/ Accessed in 2021/10/15.

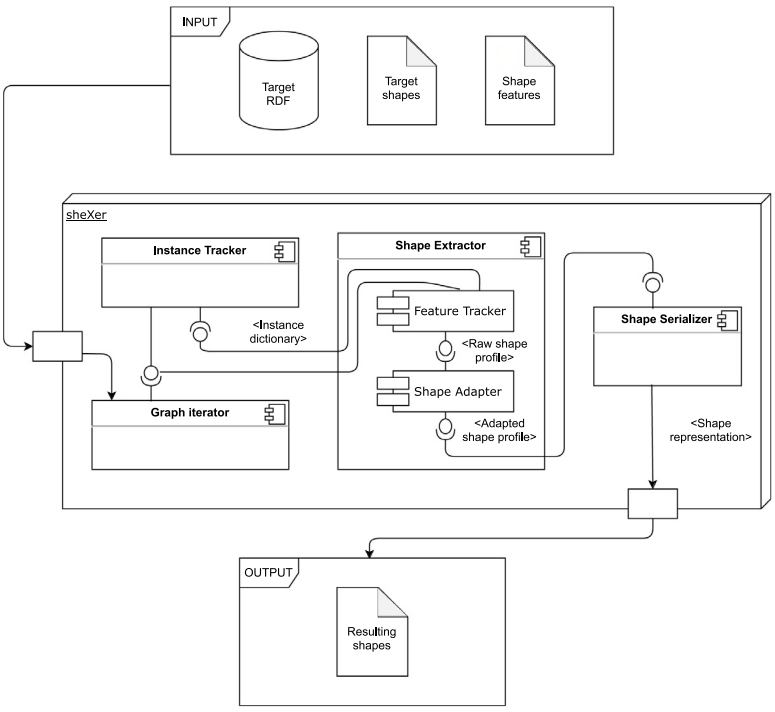

Fig. 2. sheXer base architecture.

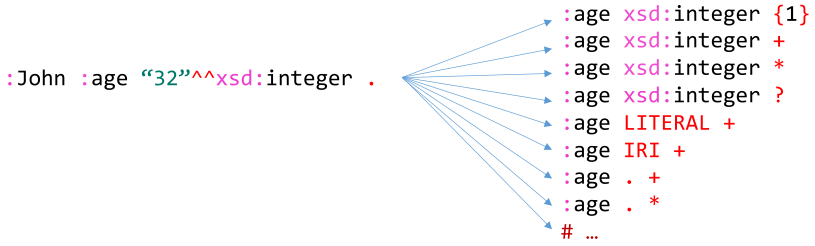

Fig. 3. Example of constraints that could get positive votes from a certain triple.

generate a shape for every class in the target graph. Our proposal can combine some of those strategies too. For example, one can request shapes via shape maps and for every class in the target KG at a time.

The IT outputs a dictionary that links instances (keys) with their list of target shapes (values).

2.3. Feature tracker

The Feature Tracker (FT) finds a candidate set of features for each target shape using a voting system. The FT receives the instance dictionary produced by the IT. Then, it processes the triples sent by the GI. The triples are used to generate positive votes for some constraints. sheXer decorates the instance dictionary to annotate the number of times that a combination of predicate and object type is found for each instance. These constraints can have different specificity w.r.t. predicate and object. In Fig. 3,[7] we show an example of several candidate constraints supported by the triple t_e =(:John :age "32"^^xsd:string). sheXer limits the positive votes generated to some representative object and cardinality combinations using the following criteria:

- Exact cardinality, i.e., exact number of triples where this combination of property and object type occurs in the dataset for a given subject.
- The range {1, unbounded}, represented by the positive closure '+'.

sheXer can produce shapes whose range includes 0 occurrences of a given constraint. However, they are generated in the Shape Adapter module. Regarding the object's specificity, the following constraints get positive votes:

- When the object is a literal: the exact type of the literal, the macro *LITERAL* (any literal), and the macro '.' (any element).
- When the object is a URI: the macro *URI* (any URI) and the macro '.'. In case the URI is linked with a shape s', also the label of s'.
- When the object is a blank node: the macro *BNODE* (any blank node) and the macro '.'.

The maximum number of votes that a constraint of a shape s can obtain is the number of instances used to extract s. Once all nodes have been explored, each constraint c_s is associated with a trustworthiness score $\theta_{c_s} = \frac{n_{c_s}}{n_s}$, where n_{c_s} is the number of positive votes to c_s and n_s is the number of instances of s.

The user can specify a minimum $\theta_U \in [0, 1]$, and the FT uses this value to discard any constraint c such as $\theta_{c_s} < \theta_U$. When the FT finishes, every shape s is associated with a candidate set of constraints which is at least supported by $n_s \cdot \theta_U$ of its instances.

[7] The constraints shown in this Figure are written for shapes in ShExC.

2.4. Shape adapter

The Shape Adapter (SA) analyzes the shapes outputted by the FT to filter, modify, and merge, some constraints. Finally, it sorts them. For such a task, the SA uses Algorithm 1. To properly understand the algorithm, several symbol conventions must be clarified:

Algorithm 1 Shape Adapter: filtering stage pseudo-code

Input: S = *target shapes*

1: **for each** $\{s \mid s \in S\}$ **do**
▷ Stage 1:
2: $C'_s \leftarrow \emptyset$
3: $U \leftarrow f_U(C_s)$
4: **for each** $\{T_U \mid T_U \in U\}$ **do**
5: $\alpha_U \leftarrow f_{d_U}(T_U)$
6: $I_{\alpha_U} \leftarrow \emptyset$
7: **for each** $\{c \mid c \in T_U \wedge c \neq \alpha_U\}$ **do**
8: $I_{\alpha_u} \leftarrow I_{\alpha_u} \cup f_{\#}(c)$
9: $C'_s \leftarrow C'_s \cup \{\alpha_U\}$
▷ Stage 2:
10: $C''_s \leftarrow \emptyset$
11: $V \leftarrow f_V(C'_s)$
12: **for each** $\{T_V \mid T_V \in V\}$ **do**
13: $\alpha_V \leftarrow f_{d_V}(T_V)$
14: **if** $\nexists I_{\alpha_V}$ **then**
15: $I_{\alpha_V} \leftarrow \emptyset$
16: **for each** $\{c \mid c \in T_V \wedge c \neq \alpha_V\}$ **do**
17: $I_{\alpha_v} \leftarrow I_{\alpha_v} \cup f_{\#}(c)$
18: $C''_s \leftarrow C''_s \cup \{\alpha_V\}$
▷ Stage 3:
19: $C_s \leftarrow C''_s$

- Every function or macro used to encapsulate any behavior is denoted as $f_a(x)$, where a is an identifier.
- S is the set containing all the target shapes.
- We denote the constraints associated to a shape s with C_s.
- $f_U(X)$ receives a set of constraints X and returns a collection of sets U. Each $T_U \in U$ is a group of constraints that have the same property and type of object, but different cardinality.
- $f_{d_U}(X)$ receives a set of constraints X which are expected to have the same property and type of object, and it returns the *dominant* constraint α_U of the set. α_U is the constraint with the highest trustworthiness. In case of tie, the α_U is the one with the most restrictive cardinality.[8]
- The constraints can have some text comments associated. We denote the comments associated to a constraint c as I_c.
- $f_{\#}(x)$ receives a constraint x and returns a textual comment with some relevant information of x, such as θ_{x_s}, cardinality and type of object.
- $f_V(X)$ receives a set of constraints X and returns a collections of sets V. Each set in V contain constraints that have the same property but different type of object.
- $f_{d_V}(X)$ receives a set of constraints X and returns a dominant constraint α_V. A constraint $c \in X$ is found the dominant constraint α_V of X when two conditions are met. First, for any $c_i \in X : \theta_{c_i} < \theta_c$, the object type of c subsumes the object type of c_i. Second, there cannot be any $c_i \in X : \theta_{c_i} > \theta_c$ such that the object type of c_i subsumes the type of c. Informally, this means that the type object of c is as specific as possible and, at a time, c is supported by as much instances as possible.[9]

Algorithm 1 can be separated in different stages to be easily understood. In stage 1 (lines 2 to 9), the original set of constraints C_s of s is transformed into a new set C'_s, such that every $c \in C'_s$ has a unique combination of property and object type in C'_s. If C_s already meets this condition, then $C_s = C'_s$. The constraints of C_s not included in C'_s are not completely discarded. They are transformed into textual comments that can appear next to their dominant constraint in the final results.

For example, picture a group of constraints composed by c_1 = *(foaf:name xsd:string +)*, with $\theta_{c_1} = 1$, and c_2 = *(foaf:name xsd:string {1})*, with $\theta_{c_2} = 0.9$. This group is related to a shape s. $\theta_{c_1} > \theta_{c_2}$, so c_1 is picked as the dominant constraint of the group. However, the user may find relevant also that 90% of the target instances associated to s has exactly one *foaf:name*. sheXer uses in-line textual comments to provide some information related to c_2.

In stage 2 (lines 10 to 18), constraints with the same property are transformed into a single dominant constraint. Note that the set of constraints transformed in Stage 2 is C'_s (see line 11), so every constraint has already a unique combination of property and object type. The constraints in C'_s may already have some comments associated that should be considered when adding a new comment in this stage.

Finally, in stage 3 (line 19), C''_s becomes the set of constraints of s. At this point, every constraint associated to s has a unique property in s, and can have extra information in textual comments.

2.4.1. Extending cardinalities

The constraints found in Algorithm 1 may not be final. The SA still performs an iteration that could modify their cardinality. As already stated, the FT outputs constrains whose cardinality does not include the possibility of zero occurrences, such as *optional* (?) or *none-to-many* (*). This can lead to situations where an instance i used to build a shape s does not conform with s. Let us suppose that sheXer is configured to extract shapes with $\theta_U = 0.8$, and a shape s is obtained. s includes a constraint c = *(foaf:name xsd:string +)* with $\theta_c = 0.9$. c has been included in s because $\theta_c > \theta_U$, but $\theta_c = 0.9$ means that 10% of the instances does not support c, ergo they do not conform with s.

sheXer includes a configuration option that allows modifying conflictive cardinalities, so every instance conforms with its related shapes. When this option is active, the cardinality of every constraint c whose $\theta_c < 1$ is modified to include a range with zero occurrences. Constraints with cardinality + are changed to *. Cardinalities of *{1}* are changed to *?*. In general, every cardinality is modified to include the zero case without losing precision w.r.t. its maximum cardinality.

The original cardinality and its θ_s are associated with the constraint as a textual comment. With this, the results include the proportion of instances that conform with the constraint excluding the zero-case. When the zero-case is included, the θ_c of any constraint c raises to 1.

2.4.2. Sorting triple constraints

The shapes extracted can include a large number of constraints, and the user may not want to read or consider them all.

The SA sorts a shape's constraints in decreasing order w.r.t. their θ_c. With this, the most reliable constraints are shown first. Constraints including the zero-case are sorted using the θ_{c_s} computed before the zero-case was included.

[8] The criteria to choose dominant constraints can be configured by the user in different ways. For example, more general cardinalities could be chosen as preferment. See sheXer's documentation for further details.

[9] This dominance criteria can be configured as well. For example, constraints with *oneOf* operators could be generated, Also, some tolerance thresholds to keep precise constraints that do not cover all the cases can be defined.

2.5. Shape serializer

The Shape Serializer (SS) transforms the in-memory information into an actual output for the user. The complexity of this task depends on the divergence between the target output format and the conceptual information outputted by the SE. Our current implementation of sheXer includes two different implementations of the SS: One for the generation of ShEx (in ShExC format) and another one for the generation of SHACL (in turtle format).

Both implementations of the SS are trivial. Even if ShEx and SHACL are not fully compatible [7], the subset of shape features used by sheXer can be represented in both languages.

2.6. Computational complexity analysis

The computational complexity of sheXer is the sum of the complexities of its modules, which are executed sequentially. The base complexity of the SA and SS modules is $O(c^2/s)$ and $O(c)$ respectively, where c is the total number of constraints extracted and is s the number of target shapes. The SA's $O(c^2/s)$ complexity is an approximation supposing a balanced number of constraints per shape and it comes from algorithm 1. Each constraint is compared with the rest of its shape's constraints, so unbalanced constraints distributions could lead to higher complexities. The worst case, where a single shape has all the constraints, will be executed in $O(c^2)$. The rest of the SA's stages are executed in $O(c)$.

The complexity of the IT and the FT modules is tightly linked to the nature of the input. For example, the IT can be executed trivially in $O(1)$ when the instance-shape relation is provided as part of the input. However, if there is a need of parsing a local file, it can take $O(t)$, being t the number of triples. The complexity can be higher if a SPARQL endpoint is involved in the process. The FT behaves similarly, as both the IT and the FT depend on the GI's execution. In the following section, we perform several experiments to extract shapes from local RDF files. Under these conditions, these two modules have the following base complexity:

- IT: $O(t_c)$, where t_c is th number of instance-class triples.
- FT $O(t + t_i)$, where t is the number of triples and t_i is the number of triples whose subject is a relevant instance.

With this, sheXer would be executed in $O(t_c) + O(t + t_i) + O(c^2/s) + O(c) = O(t + t_c + t_i + c^2/s)$. Note that $t \geq t_c$ and $t \geq t_i$. Note also that the only non-linear complexity is $O(c^2/s)$. However, when computing big datasets where $t_i \gg c$, this part of the algorithm is not the most expensive. Many instances usually generate a relatively low number of constraints. Then, it takes more time to generate constraints from those instances in $O(t_i)$ than to process later those few constraints in $O(c^2/s)$.

3. Experiments

Shape languages are relatively new, and so is the problem of automatic shape extraction. To the best of our knowledge, there is not yet a published benchmark to compare the correctness nor performance of existing approaches. The experiments of automatic extractors already published range from pure qualitative analysis to performance or scalability analyses.

In this paper, we have designed experiments to check the performance of sheXer in two dimensions: memory consumption and execution time. We have executed sheXer to extract shapes from three well-known LD data sources: Wikidata,[10] YAGO,[11] and DBpedia.[12] Details about these computations can be found in Table 1. In our experiments, we always use local RDF parsers of local files. Alternative ways of input, such as querying an endpoint, would depend on the endpoint's performance and availability. This could introduce factors that may not be related to sheXer's actual performance in the experiments.

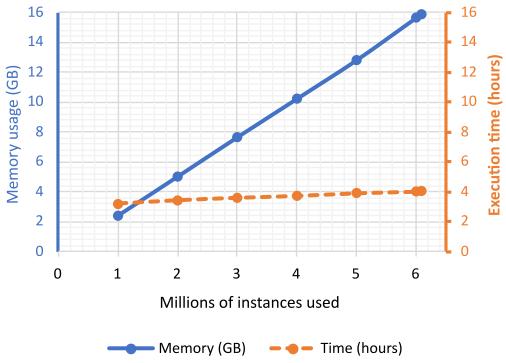

Fig. 4. Performance of sheXer with different amounts of instances used.

We run our tests in a virtual machine with the following specifications: Debian 8 *Jessie* OS, Intel Xeon E5502 processor 1.87 GHz, 32 GB RAM, HDD disk with a read speed of 145 MB/s measured with the hdparm[13] command. We set an arbitrarily low threshold $\theta_U = 0.01$ to discard noisy marginal features for every computation. $\theta_U = 0.01$ discards constraints that comply with less than 1% of the total instances considered.

As one can see, the time and memory consumption to get a result are different for each source. These numbers are related to some input features, such as dataset size, number of triples, and number of target shapes. In the following subsections, we propose scenarios with different inputs to analyze the impact of several parameters on our proposal's performance. All these scenarios are based on the DBpedia case.

3.1. Limiting the number of instances used

As shown in Table 1, the number of instantiation triples in DBpedia is higher than 6.6 million. We repeated the process of shape extraction limiting the number of instantiation triples to a certain number. We started in 1M triples and repeated the computation with an arbitrary increment of 1M instances each time until every instance of every target class is used. The results regarding execution time and memory consumption are shown in Fig. 4.

As one can see, the number of instances has a linear relation with execution time and memory consumption. The impact on memory consumption is caused by the instance dictionary generated by the IT and the FT. The more instances, the bigger becomes this dictionary and its associated memory usage. The effect on execution time is lower than the impact on memory usage because sheXer is parsing the whole content in every case. This parsing process sets a minimum execution time, which is more significant in environments where the I/O disk speed is relatively low. Each triple is evaluated as relevant or not for the process, and the relevant triples trigger extra calculations in the TF and SA modules. The more instances are considered, the more triples become relevant. This causes the linear relation between the number of instances and execution time.

[10] Only triples using Wikidata direct properties in the namespace http://www.wikidata.org/prop/direct/ where used to extract shapes. The source can be downloaded at https://archive.org/details/wikidata-json-20150518 Accessed in 2021/10/15.

[11] We computed YAGO3, which can be downloaded at https://yago-knowledge.org/downloads/yago-3 Accessed in 2021/10/15.

[12] We used a subgraph containing mapping-based literals and objects, as well as class-instance relations. This collection can be downloaded at https://databus.dbpedia.org/danifdezalvarez/collections/latest_mapping_shexer_test Accessed in 2021/10/15.

[13] https://linux.die.net/man/8/hdparm Accessed in 2021/10/15.

Table 1
Basic information about the YAGO, Wikidata and DBpedia computations.

Dataset	Target shapes	Dataset size (GB)	Nº of triples (millions)	Nº of instances (millions)	Memory usage (GB)	Execution time (h)
Wikidata dump 2015-05-18	1000 (top 1000 classes with more instances)	42.0	991.6 M	13.0 M	25.2	38.3
YAGO3	1000 (top 1000 classes with more instances)	10.3	138.3 M	5.3 M	12.6	17.2
English chapter of Dbepedia	422 (every class in the DBpedia Ontology with instances)	6.0	44.0 M	6.6 M	16.1	4.1

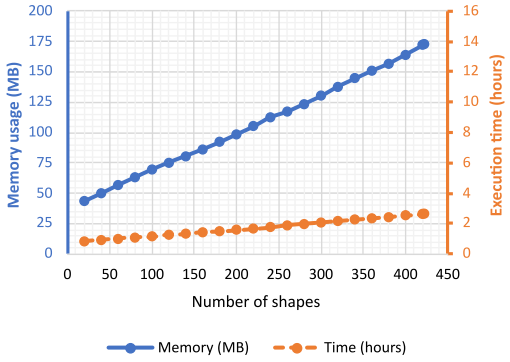

Fig. 5. sheXer's with different number of target shapes.

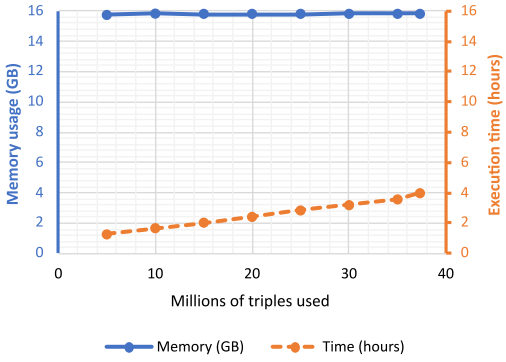

Fig. 6. Performance of sheXer with different dataset sizes.

3.2. Limiting the number of target shapes

As stated in Section 2.6, sheXer's complexity depends, among other parameters, on the number of constraints produced. However, the number of constraints cannot be known a priori. In opposition, in case there is a balanced distribution of the number of constraints per shape, the number of shapes, which can be known a priori, can be used to estimate execution times and memory usage. In this subsection, we study the impact on the performance of the number of target shapes.

In our experiment, we start with just 20 target shapes and then perform arbitrary increases of 20 shapes for each iteration until every class with at least one instance is used. As already checked, the number of instances has a crucial impact on the performance. Then, the more instances a class has, the greater is its impact. To avoid erratic numbers due to classes with an unbalanced number of instances, we used an arbitrarily low number of instances to be considered (as most) per each class. We picked this limit so 90% (380 out of 422) of the classes has at least this number of instances. In our dataset, *dbo:Chancellor* ranks 380th w.r.t. to number of instances, with a total of 57. Then, the limit picked was 57 instances. The results obtained are shown in Fig. 5. Note that the memory scale in the *y*-axis is different from the rest of the figures. It ranges from 0 to 200 MB instead of 0 to 16 GB.

As one can see, using a relatively low number of instances drastically decreases memory usage. However, there is a linear relationship between the number of shapes used and memory consumption. This relation is explained by two factors: on the one hand, the FT generates an abstract profile in main memory for each shape. On the other hand, each shape causes a growth in the instance dictionary proportional to its number of instances.

There is also a linear relation with execution time, with a similar tendency to the one observed in Fig. 4. However, there is a notable difference in execution time between Figs. 5 and 4 at the maximum values of the *x*-axis, explained by the limit of 57 instances per class used in Fig. 5. The difference of 1.39 h is the time used to compute the instances discarded in Fig. 5's experiment.

3.3. Limit the amount of triples

In this subsection, we study the performance effect of the number of triples processed. Since the impact of the number of instances has already been studied, in this experiment, we keep the same number of instantiation triples across all iterations. Every instance is used in every case. We do change the number of entity-to-entity and entity-to-literal triples. Our DBpedia dataset contains 37.328M of these two kinds of triples.

We performed iterations starting at the arbitrary amount of 5M triples (without counting the instantiation triples). Then, we made an arbitrary increment of 5M triples for each iteration until reaching the total 37.328M triples. The number of triples of each kind added at each iteration is proportional to the total number of triples available. In the first iteration, we used 2.698M object triples and 2.302M literal triples, which make together 5M elements. We add the very same number of triples of each type at each iteration. The results are shown in Fig. 6.

As one can see, memory usage stays stable and independent of the number of triples used. There is not a determinant linear relation between memory usage and triples which are not expressing a class-instance relationship.

However, there is a linear relation with execution time caused by the different number of triples relevant for the process in each iteration. Those triples are potentially spread across the whole dataset. Then, the bigger is the slice of the dataset used, the higher are the chances to find this kind of triples.

3.4. Convergence w.r.t. number of instances used

As already shown, sheXer's performance scales linearly w.r.t. some features of the input dataset. Too ambitious goals can lead to high rates of memory usage or execution time.

Execution times can be tackled by producing a parallel implementation of sheXer. MapReduce [15] could be used for such a goal. Every module described in Section 2 processes an input

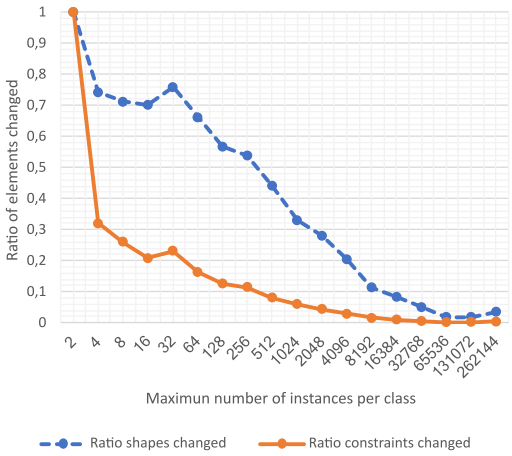

Fig. 7. Shape convergence using different amounts of instances per shape.

composed of independent elements, which can be computed in parallel to produce later merged results.

The structure with more impact on memory usage is the instance dictionary generated by the IT. The instance dictionary is a key element to produce shape inter-linkage in reasonable execution time, as the operation to check the class of a given instance is frequent, and it allows to perform it in O(1) complexity. It is also used to produce precise cardinalities.

To cope with this limitation, we explore whether using a relatively low number of instances per shape can be enough to extract accurate shapes. We have extracted shapes for every DBpedia class in different iterations, using an increasing maximum number of instances per shape at each iteration. We start using two random instances per class. Then, we perform successive iterations doubling the maximum number of instances. In Fig. 7, we show the changes detected between consecutive iterations. Note that the scale of the x-axis is logarithmic. At the end of each iteration, we check two factors:

1. The number of total changes w.r.t. the previous iteration. We count as a change in a shape one of the following events:
   - Gaining a triple constraint.
   - Losing a triple constraint.
   - Having a modification in any element of a triple constraint.
2. The number of shapes changed w.r.t. the previous iteration. We consider that a shape has changed when there is at least one change among its triple constraints.

The numbers shown in the y-axis are relative. For shapes, we show the proportion of elements that changed w.r.t. the total number of shapes, which is 422 in every case. For triple constraints, we show the proportion of changes detected w.r.t. to the total number of triple constraints.

As one can see, with very few exceptions, every iteration causes fewer changes than its previous iterations. With a high enough instance limit, the shapes tend to converge. With 8192 instances, just 1,5% of the constraints are changed. These changes affect 11% of the shapes. For any other iteration with a higher instance limit, the proportion of shapes getting any change is always lower than 10% and affects 1% or less of the constraints.

The ratios of shape and constraint changes detected for every iteration are also available in Table 2, as well as some other information related to the shapes' evolution. Let C be the set of all target shapes to extract. An increment of x units in the instance limit for a given iteration would introduce $|C| \cdot x$ new instances on the computations just in case every $c \in C$ has at least x instances still not considered. Since this is not always the case, the actual number of new instances introduced in the experiments in successive iterations is shown in the sixth column of Table 2.

The number of new instances for each iteration allows us to calculate what we called Shape-Instance Performance (SIP) and Constraint-Instance Performance (CIP). SIP is shown in the seventh column of Table 2, and it is defined as the relation between the number of instances used and the number of shapes changed. SIP can also be interpreted as the number of instances required to cause a single shape change. As one can see, the more instances are processed, the smaller is the effect of a single instance over the results, and the SIP grows rapidly with each iteration. For example, at the already mentioned limit of 8192 instances per shape, 9267 new instances are worth a single shape change.

CIP is shown in the eighth column of Table 2 and it is defined as the relation between the number of instances used and the number of constraints changed. As one can see, the CIP is slightly better than SIP for almost every iteration. However, both numbers tend to stay in the same magnitude order.

Table 2 and Fig. 7 indicate that a representative sample of instances can be enough to extract highly accurate shapes. Using 8192 as instance limit, the total instances computed are close to 1.3M. As shown in Fig. 4, this leads to a memory usage close to 3.2 GB, which is five times smaller than the 16 GB needed to compute the dataset using every available instance.

4. Related work

Several approaches to automatically extract shapes have been proposed. The closest work to sheXer is Shape Designer [13]. Shape Designer consists of a tool to perform automatic extraction of shapes with KG mining. Both sheXer and Shape Designer support ShEx and SHACL, and both keep an internal score of how trustworthy a given constraint can be w.r.t. how frequently is it supported by the nodes used to extract it. However, there are fundamental differences between these two approaches. Shape Designer is integrated with a graphic tool and aims to produce shapes that are not intended to be definitive. The tool extracts candidate shapes that the user can customize later. In opposition, sheXer aims to produce shapes as accurately as possible and does not necessarily include human intervention in its workflow. Also, sheXer and Shape Designer offer different approaches to solve constraints including IRIs. Shape Designer uses either the macro IRI or value sets that can restrict the possible IRIs to a string pattern. sheXer can produce actual shape inter-linkage, i.d., triple constraints whose object is another shape label.

The system proposed in [16] uses a machine learning approach to generate SHACL shapes associated with classes. The authors choose combinations of class-property and associate them to two types of constraints: cardinality and range. Cardinality refers to the minimum and maximum occurrences. Range refers to the type of object, which is one of *sh:IRI*, *sh:BlankNode*, *sh:BlankNodeOrIRI*, *sh:Literal*, or specific literal types. Both types of constraints have a finite set of possible final values, so the approach is formulated in terms of a classification problem. Once all pairs have been associated with their constraints, the constraints of a given class c are all merged to produce a SHACL shape associated to c.

The approach presented in [17] transforms abstract semantic profiles generated by ABSTAT [18] into SHACL shapes. This proposal does not perform shape inter-linkage, but it includes constraints with inverse paths, i.e., it can describe the topology of a node when it is used as the object in a triple. The system

Table 2
Shape convergence with different instance limits per class.

Instance limit per class in previous iteration	Instance limit per class in current iteration	Total constraints	Ratio of shapes that changed	Ratio of constraints that changed	Actual number of new instances used	Shape-Instance Performance (SIP)	Constraint-Instance Performance (CIP)
0	**2**	2842	1000	1000	840	2,0	0,30
2	**4**	3409	0742	0319	827	2,6	0,76
4	**8**	3992	0711	0261	1637	5,5	1,57
8	**16**	4550	0701	0208	3227	10,9	3,41
16	**32**	4303	0758	0232	6342	19,8	6,36
32	**64**	4214	0661	0163	12 239	43,9	17,84
64	**128**	4292	0566	0127	23 547	98,5	43,36
128	**256**	4378	0538	0115	44 335	195,3	88,14
256	**512**	4392	0441	0081	78 907	424,2	221,65
512	**1024**	4409	0329	0060	131 876	948,7	495,77
1024	**2048**	4414	0280	0044	217 935	1846,9	1117,62
2048	**4096**	4424	0204	0028	325 744	3787,7	2626,97
4096	**8192**	4445	0114	0015	444 817	9267,0	6541,43
8192	**16 384**	4436	0083	0010	567 820	16 223,4	12 343,91
16 384	**32 768**	4443	0050	0005	640 852	30 516,8	26 702,17
32 768	**65 536**	4444	0019	0002	668 062	83 507,8	66 806,20
65 536	**131 072**	4447	0019	0002	807 333	100 916,6	73 393,91
131 072	**262 144**	4448	0036	0003	801 842	53 456,1	53 456,13

excludes part of the information generated by ABSTAT related to frequencies when it generates the shapes, as there is not a formal way to represent it in SHACL. In opposition, sheXer uses it to compute trustworthiness scores or filter infrequent features and provides this information with textual comments.

Regarding automatic mappings between ontologies and shapes, the authors in [19] propose using Ontology Design Patterns (ODP) to obtain SHACL shapes. However, no actual mappings between SHACL and ODP are proposed. In [20], SHACL and OWL are thoroughly compared in terms of meaning and expressiveness. The authors also provide mappings between OWL and SHACL. These mappings can be used by proposals that aim to extract shapes from pure ontological content.

Astrea [21] is a tool to perform automatic extraction of shapes from ontologies. The authors produce SHACL content by mapping ontology patterns into SHACL constructions. Astrea is a publicly available tool based on the mappings of Astrea-KG.[14] Astrea-KG's content allows generating SHACL shapes with an expressiveness that includes 60% of the total constraints available in SHACL.

SHACLerarner, a method to learn SHACL constraints based on Inverse Open Path rules (IOP), is presented in [14]. SHACLerarner adapts Open Path Rule Learner (OPRL) [22] to extract IOP rules, which can be translated to SHACL. SHACLearner works with rules between entities in a KG, which causes that the shapes obtained do not contain constraints related to literals.

In [23], a method to generate SHACL and ShEx shapes from R2RML documents is presented. R2RML is a W3C standard language to enable automatic translation from Relational databases to RDF documents. Since the generation of shapes is based on R2RML, this approach can be applied only in KGs whose genesis is a mapped relational database. However, it achieves excellent conformance with the target data model.

Some other previous works extract different schema notions from RDF graphs. In [24], the authors present an approach to extract frequent graph patterns conceptually similar to shapes, whose aim is to characterize the content of RDF triple-stores. The patterns are represented using an adaptation of Deep-First Search code. The authors in [25] extract Knowledge Patterns from KGs. These patterns are expressed in OWL and characterize classes by detecting their frequent properties and providing an adequate range for them.

[14] Endpoint to query Astrea KG: https://astrea.helio.linkeddata.es/sparql Accessed in 2021/10/15.

5. Conclusion

In this paper, we have presented sheXer, a system to perform automatic shape extraction based on KG mining. Our proposal extract shapes by exploring the neighborhood of custom groups of target nodes. Each extracted constraint is qualified with a score that allows filtering infrequent elements, sorting results, and providing extra information with textual comments. sheXer can produce ShEx and SHACL content and compute big real-world datasets.

Our system is based on an iterative mining strategy that avoids loading in main memory the entire KG. The execution time and peak of sheXer's memory usage have a linear relation with the number of triples relevant for the extraction process. However, we have shown that the shapes obtained using large amounts of instances tend to converge with shapes obtained using a relatively low number of instances. This instance limit has a significantly positive effect both on memory consumption and execution time.

We contemplate several lines for future work regarding our proposal:

- To produce and evaluate an implementation of sheXer using parallel computing and alternative mechanisms to handle memory.
- To perform a thorough comparison of sheXer with other automatic shape extraction systems, once an adequate public benchmark for such a purpose is available.
- To include ontological information in sheXer's workflow, so the system can discard constraints that contradict ontology information.
- To provide mechanisms for including shape inheritance in the results, once this feature becomes stable in ShEx or SHACL specifications.

CRediT authorship contribution statement

Daniel Fernandez-Álvarez: Conceptualization, Methodology, Software, Formal analysis, Investigation, Writing – original draft. **Jose Emilio Labra-Gayo:** Supervision, Conceptualization, Writing – review & editing. **Daniel Gayo-Avello:** Supervision, Conceptualization, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work is supported by the Severo Ochoa Research Program (2017 call, exp. 1436 number BP17-88).

References

[1] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P.N. Mendes, S. Hellmann, M. Morsey, P. Van Kleef, S. Auer, et al., Dbpedia–a large-scale, multilingual knowledge base extracted from wikipedia, Semant. Web 6 (2) (2015) 167–195.

[2] D. Vrandečić, M. Krötzsch, Wikidata: a free collaborative knowledgebase, Commun. ACM 57 (10) (2014) 78–85.

[3] T. Rebele, F. Suchanek, J. Hoffart, J. Biega, E. Kuzey, G. Weikum, Yago: A multilingual knowledge base from wikipedia, wordnet, and geonames, in: International Semantic Web Conference, Springer, 2016, pp. 177–185.

[4] A. Hogan, E. Blomqvist, M. Cochez, C. D'Amato, G. de Melo, C. Gutierrez, S. Kirrane, J.E. Labra-Gayo, R. Navigli, S. Neumaier, A.-C. Ngonga Ngomo, A. Polleres, S.M. Rashid, A. Rula, L. Schmelzeisen, J. Sequeda, S. Staab, A. Zimmermann, Knowledge graphs ? (?). DOI: (pending).

[5] E. Prud'hommeaux, J.E. Labra Gayo, H. Solbrig, Shape expressions: an rdf validation and transformation language, in: Proceedings of the 10th International Conference on Semantic Systems, 2014, pp. 32–40.

[6] H. Knublauch, D. Kontokostas, Shapes constraint language (shacl), W3C recommendation 20 (07).

[7] J.E.L. Gayo, E. Prud'Hommeaux, I. Boneva, D. Kontokostas, Validating rdf data, Synth. Lect. Semant. Web: Theory Technol. 7 (1) (2017) 1–328.

[8] J.E. Labra Gayo, D. Fernández-Álvarez, H. Garcıa-González, Rdfshape: An rdf playground based on shapes, in: Proceedings of ISWC, 2018.

[9] D. Fernández-Alvarez, J.E. Labra-Gayo, H. Garcıa-González, Inference and serialization of latent graph schemata using shex, 2016.

[10] D. Fernández-Álvarez, H. García-González, J. Frey, S. Hellmann, J.E.L. Gayo, Inference of latent shape expressions associated to dbpedia ontology, in: International Semantic Web Conference (P & D/Industry/BlueSky), 2018.

[11] F. Cifuentes-Silva, D. Fernández-Álvarez, J.E. Labra-Gayo, National budget as linked open data: New tools for supporting the sustainability of public finances, Sustainability 12 (11) (2020) 4551.

[12] A. Waagmeester, E.L. Willighagen, A.I. Su, M. Kutmon, J.E.L. Gayo, D. Fernández-Álvarez, Q. Groom, P.J. Schaap, L.M. Verhagen, J.J. Koehorst, A protocol for adding knowledge to wikidata: aligning resources on human coronaviruses, BMC Biol. 19 (1) (2021) 1–14.

[13] I. Boneva, J. Dusart, D.F. Alvarez, J.E.L. Gayo, Shape designer for shex and shacl constraints, in: ISWC 2019-18th International Semantic Web Conference, 2019.

[14] P.G. Omran, K. Taylor, S.R. Mendez, A. Haller, Towards shacl learning from knowledge graphs.

[15] J. Dean, S. Ghemawat, Mapreduce: a flexible data processing tool, Commun. ACM 53 (1) (2010) 72–77.

[16] N. Mihindukulasooriya, M.R.A. Rashid, G. Rizzo, R. García-Castro, O. Corcho, M. Torchiano, Rdf shape induction using knowledge base profiling, in: Proceedings of the 33rd Annual ACM Symposium on Applied Computing, 2018, pp. 1952–1959.

[17] B. Spahiu, A. Maurino, M. Palmonari, Towards improving the quality of knowledge graphs with data-driven ontology patterns and shacl, in: ISWC (Best Workshop Papers), 2018, pp. 103–117.

[18] B. Spahiu, R. Porrini, M. Palmonari, A. Rula, A. Maurino, Abstat: ontology-driven linked data summaries with pattern minimalization, in: European Semantic Web Conference, Springer, 2016, pp. 381–395.

[19] H.J. Pandit, D. O'Sullivan, D. Lewis, Using ontology design patterns to define shacl shapes, in: WOP@ ISWC, 2018, pp. 67–71.

[20] H. Knublauch, Shacl and owl compared, URL: https://spinrdf.org/shacl-and-owl.html.

[21] A. Cimmino, A. Fernández-Izquierdo, R. García-Castro, Astrea: automatic generation of shacl shapes from ontologies, in: European Semantic Web Conference, Springer, 2020, pp. 497–513.

[22] P. Omran, K. Taylor, S. Rodríguez Méndez, A. Haller, Active Knowledge Graph Completion, Tech. rep. Tech. rep., Australian National University, 2020, https://openresearch ….

[23] J.-W. Choi, Automatic construction of shacl schemas for rdf knowledge graphs generated by r2rml mappings, J. Korea Soc. Comput. Inf. 25 (8) (2020) 9–21.

[24] A. Basse, F. Gandon, I. Mirbel, M. Lo, Dfs-based frequent graph pattern extraction to characterize the content of rdf triple stores, in: Web Science Conference 2010 (WebSci10), 2010.

[25] E. Blomqvist, Z. Zhang, A.L. Gentile, I. Augenstein, F. Ciravegna, Statistical knowledge patterns for characterising linked data, in: WOP, Citeseer, 2013.