# Analysing Large Inconsistent Knowledge Graphs

Thomas de Groot, Joe Raad, and Stefan Schlobach

Knowledge Representation & Reasoning Group, VU University Amsterdam
{t.j.a.de.groot, j.raad, k.s.schlobach}@vu.nl

**Abstract.** Based on formal semantics, most of the Knowledge Graphs (KGs) on the Web of Data can be put to practical use. Unfortunately, a significant number of those KGs contain contradicting statements, and hence are logically inconsistent. This makes reasoning limited and the knowledge formally useless. Understanding how these contradictions are formed, how often they occur, and how they vary between different KGs, is essential for fixing and avoiding such contradictions in the future, or are at least for developing better tools that handle inconsistent KGs. Methods exist to explain a single contradiction, by finding the minimal set of axioms sufficient to produce it, a process known as justification retrieval. In large KGs, these justifications can be frequent and might redundantly refer to the same type of modelling mistake. Furthermore, these justifications are –by definition– domain dependent, and hence difficult to interpret or compare. This paper introduces the notion of anti-pattern for generalising these justifications, and presents an approach for detecting almost all anti-patterns from any inconsistent KG. Experiments on KGs of over 28 billion triples show the scalability of this approach, and the importance of anti-patterns to analyse and compare contradictions between KGs.

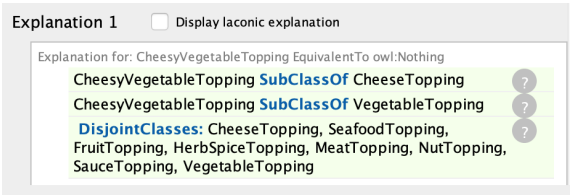
**Keywords:** linked open data, reasoning, inconsistency

## 1 Introduction

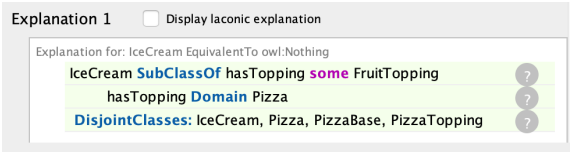
Through the combination of web technologies and a judicious choice of formal expressivity (description logics which correspond to a decidable 2-variable fragment of first order logic), it has become possible to construct and reason over Knowledge Graphs (KGs) of sizes that were not imaginable only few years ago. Nowadays, KGs of hundreds of millions of statements are routinely deployed by researchers from various fields and companies worldwide. Since most of the larger KGs are traditionally built over a longer period of time, by different collaborators, these KGs are highly prone for containing logically contradicting statements. As a consequence, reasoning over these KGs becomes limited and the knowledge formally useless.

Typically, once these contradicting statements in a KG are retrieved, these incoherences are either explained [16], repaired [15] or ignored via non-standard reasoning [11]. This work falls in the first category of approaches, where the focus is to find out and explain what has been stated in the KG that causes

the inconsistency to hold. Understanding how these contradictions are formed and how often they occur is essential for fixing and avoiding such contradictions in the future. At the least, it is a necessary step for developing better tools that can handle inconsistent KGs. In the field of contradiction explainability (or more generally entailment explainability), research has mainly focused on a specific type of explanation called *justifications*. A justification for a contradiction (entailment) in the KG is a minimal subset of the ontology that is sufficient for the contradiction (entailment) to hold. For instance, in the known *Pizza ontology*[1] that serves as a tutorial for OWL and Protégé, we can find two contradictions that were asserted by its developers on purpose. The first contradiction (A) demonstrates the unsatisfiable class *CheesyVegetableTopping*, that has two disjoint parents *CheeseTopping* and *VegetableTopping*. The second of those contradictions (B) demonstrates a common mistake made with setting a property's domain. The property *hasTopping* has as domain the class *Pizza*. This means that the reasoner can infer that all individuals using the *hasTopping* property must be of type *Pizza*. On the other hand, we find in the same ontology a property restriction on the class *IceCream*, stating that all members of this class must use the *hasTopping* property. However, since the ontology specifies that the classes *Pizza* and *IceCream* are disjoint, this causes an inconsistency in the ontology. As presented in Figure 1, justifications serve to explain such contradictions, by showing the minimal set of axioms from the ontology that causes the contradiction to hold.




(a) Justification of contradiction A




(b) Justification of contradiction B

Fig. 1: Screenshot of Protégé showing the justifications of two contradictions found in the Pizza ontology

Although justifications provide a good basis for debugging modelling and data quality issues in the KG, their specificity in explaining the contradictions

[1] `https://protege.stanford.edu/ontologies/pizza/pizza.owl`

increases in some cases the complexity for analysing and dealing with these detected contradictions. Particularly in larger KGs, these complexities are amplified and encountered in different dimensions. Firstly, existing methods to retrieve entailment justifications do not *scale* to KGs with billions of triples. Secondly, these retrieved contradictions with their justifications can be too *frequent* to manually analyse and understand the modelling mistakes made by the ontology designer. This is especially inconvenient when a significant number of these retrieved justifications actually refer to the same type of mistake, but instantiated in different parts of the KG (e.g. similar misuse of the properties' domain and range in multiple cases). Thirdly, since justifications represent a subset of the KG, they are by definition *domain dependent*, and requires basic domain knowledge for understanding the contradiction. This fact is obviously more limiting in complex domains, such as medical KGs, as opposed to the Pizza ontology example above. These various challenges in computing and understanding justifications in their traditional form, poses the following research questions:

**Q1:** Can we define a more general explanation of contradictions that categorises the most common mistakes, independently from the domain of the KG?
**Q2:** Can we retrieve these generalised explanations from any KG, independently from its size?
**Q3:** How can these generalised explanations help analysing and comparing certain characteristics between the most commonly used KGs in the Web?

This paper introduces a method for extracting and generalising contradictions from KGs. We call these generalised contradictions *anti-patterns*, as they can be seen as common mistakes made in modelling the domain, in the KG population process, or possibly in data linking. For the retrieval of the anti-patterns, we have designed an extraction pipeline that can find anti-patterns from any inconsistent KG. We test the scalability and the completeness of the approach on several KGs from the Web, including the *LOD-a-lot*, *DBpedia*, *YAGO*, *Linked Open Vocabularies*, and the *Pizza* ontology, with a combined size of around 30 billion triples. Despite deploying a number of heuristics to ensure scalability, our experiments show that our method can still detect most anti-patterns in a KG, in a reasonable runtime and average computational capacity. Finally, we show how these detected anti-patterns can be put to use for analysing and comparing certain characteristics of these inconsistent KGs.

The rest of the paper is structured as follows. Section 2 presents related works. Section 3 presents the preliminaries and the notation. Section 4 introduces the notion of anti-patterns and describes our approach for detecting them. Section 5 presents the evaluation of the approach. Section 6 presents inconsistency analyses conducted on several large inconsistent KGs. Section 7 concludes the paper.

## 2 Related Work

*Dealing with inconsistencies.* Dealing with inconsistent knowledge bases is an old problem and solutions have been proposed as early as 1956 by Stephen Toulmin

[19], in which the solution is to reason over consistent subbases. Another solution including changing the inference to fit the inconsistency, such as paraconsistent reasoning [12]. Debugging inconsistent KGs has taken up traction in the last decade with work on debugging of ontologies [16], which led to the notion of justification [10], minimal subsets of the graphs preserving entailments (and inconsistency). Töpper et al. [18] propose a solution to identify contradictions in DBpedia for single types of contradiction. With the extraction of our anti-patterns, we generalise for what works on any arbitrary contradiction. Either et al. [6] propose two methods to locate contradictions created by combining KGs. They focus on bridge rules, that connect KGs. The method is further improved in [21], with an algorithm which uses subsets to locate inconsistencies in the KGs. Their goal, though, is to find these sets for a specific KG, wherein our case we generalize the inconsistencies over sets of KGs. Our method reuses part of the work of [13], where the authors propose the efficient algorithm for path finding that we use in our subgraph generation.

*Characterising KGs.* Fárber et al. [7] give an in-depth comparison of several large KGs and their characteristics. Their work is expanded by Debattista et al. in [5], in which they analysed 130 datasets from the Linked Open Data Cloud using 27 Linked Data quality metrics. Both papers show that each graph has a different underlying structure which leads to different behaviour of algorithms. In our work, we focus on analysing inconsistency across KGs, given a number of different metrics. To the best of our knowledge, this has not been done before on the scale of the Web of Data.

## 3 Background

We follow the formal notion of a KG from Paulheim et al. [14], stating that "a KG (i) mainly describes real-world entities, and their interrelations, organized in a graph, (ii) defines possible classes, and relations of entities in a schema, (iii) allows for potentially interrelating arbitrary entities with each other and (iv) covers various topical domains". We will use the Semantic Web stack, with its languages RDF, RDFS and OWL as representation languages for data, schema and ontologies.

A *triple pattern* is an RDF triple in which at least its subject, predicate or object is a variable. Any finite set of triple patterns is a *basic graph pattern (BGP)*, and forms the basis of SPARQL for answering queries (matching a BGP to a subgraph of the KG by substituting variables with RDF terms).

We use the standard notions of entailment, satisfiability and consistency for RDF(S) and OWL. Most importantly, *an inconsistent KG* is a graph for which no model exists, i.e. a formal interpretation that satisfies all the triples in the graph given the semantics of the used vocabularies. This means that one or more statements that are contradicting with each other. A *justification* is a set of axioms that acts as an explanation for an entailment. Formally, a justification is a minimal subset of the original axioms which are sufficient to prove the entailed

formula. Given that our KGs are sets of triples, our justifications are instantiated BGPs and are always a minimal set of axioms for a single contradiction.

## 4 Defining and Detecting Anti-Patterns

In this section, we introduce the notion of anti-patterns, and describe our approach for retrieving anti-patterns from any inconsistent KG.

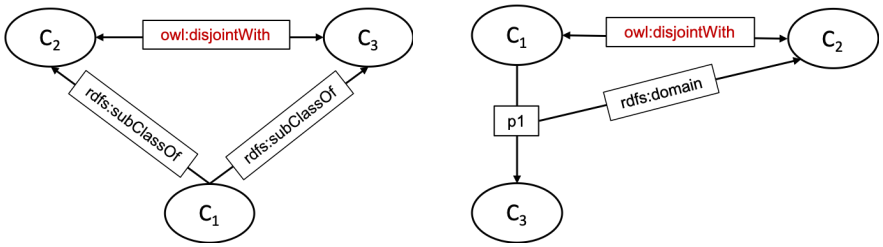
### 4.1 Anti-patterns

A justification is a description of a single contradiction, which can be represented as an instantiated BGP. If a KG shares contradictions of similar types, we generalise justifications towards what we call anti-patterns. An anti-pattern of a KG is a minimal set of non-instantiated triple patterns that match an inconsistent subgraph of this KG. These anti-patterns can intuitively be seen as common mistakes in datasets. To transform a justification into an anti-pattern, we replace the elements in the subject and object position in the BGP with variables. In order to prevent breaking the contradiction, elements appearing in the predicate position of a justification are not replaced in the anti-pattern, with the exception of one case: elements appearing both in the predicate position and in the subject or object position of the same justification.

Going back to the example of Figure 1, we presented two contradictions found in the Pizza ontology with their justifications. Contradiction A shows an inconsistency in the ontology in which *CheesyVegetableTopping* is subclass of the two disjoint classes *CheeseTopping* and *VegetableTopping*. While this example refers to a specific case of contradiction entailed from the description of these three classes, it also refers to a common type of modelling mistake which could have also been present in other parts of the ontology. For instance, using the same principle, the modeller could have also created the class *FruitVegetableTopping* as subclass of the two disjoint classes *FruitTopping* and *VegetableTopping*. This formalisation of certain types of mistakes is what we refer to as anti-patterns. Figure 2 presents the two anti-patterns generalising contradictions A and B. For instance in the anti-pattern of contradiction A, the three classes *CheesyVegetableTopping*, *CheeseTopping* and *VegetableTopping* are replaced with the variables $C_1$, $C_2$ and $C_3$ respectively. In this anti-pattern, replacing the predicate *owl:disjointWith* with a variable $p_1$ would break the contradiction, since $p_1$ could potentially be matched in the KG with another predicate such as *rdfs:subClassOf*. On the other hand, we can see in the anti-pattern of contradiction B that the predicate *hasTopping* is replaced with the variable $p_1$, since it also appears in the subject position of the triple $\langle$*hasTopping*, *rdfs:domain*, *Pizza*$\rangle$.

### 4.2 Approach

This section describes our approach for finding anti-patterns from any inconsistent KG. Finding anti-patterns from a KG would mainly consist of two steps:




(a) Anti-pattern of contradiction A (b) Anti-pattern of contradiction B

Fig. 2: Graphical representation of the anti-patterns of the two contradictions found in the Pizza ontology

retrieving justifications of contradictions, and then generalising these detected justifications into anti-patterns. Such approach can deal with multiple dimensions of complexity, mainly:

**Knowledge Graphs can be too large to query.** Now that KGs with billions of triples have become the norm rather than the exception, such approach must have a low hardware footprint, and must not assume that every KG will always be small enough to fit in memory or to be queried in traditional triple stores.

**Justification retrieval algorithms do not scale.** Computing all justifications of existing contradictions typically requires loading the full KG into memory, and existing methods do not scale to KGs with billions of triples [3].

**Comparing anti-patterns can be computationally complex.** We will later see that checking whether justifications can be generalised into the same anti-pattern, consists of determining whether their anti-patterns are isomorphic, which is a problem known to not be solvable in polynomial time [9].

Theoretically, guaranteeing the retrieval of *all* anti-patterns given any inconsistent KG requires firstly finding *all* contradictions' justifications and then generalising these justifications into anti-patterns. In practice, and as a way to tackle the above presented complexities, our approach introduces a number of heuristics in various steps of the pipeline. These heuristics emphasises the scalability of the approach over guaranteeing its completeness regarding the detection of all anti-patterns. Mainly, an initial step is introduced in the pipeline that consists of splitting the original KG into smaller and overlapping subgraphs. Depending on the splitting strategy, this step can impact the number of retrieved justifications, which in its turn can potentially impact the number of the retrieved anti-patterns. In the following, we describe the mains steps of our approach consisting of (1) splitting the KG, (2) retrieving the contradictions' justifications, and (3) generalising these justifications. Figure 3 summarises these three steps.

**1. Splitting the KGs.** Due to the large size of most recent KGs, running a justification retrieval algorithm over the complete KG to retrieve all contradictions

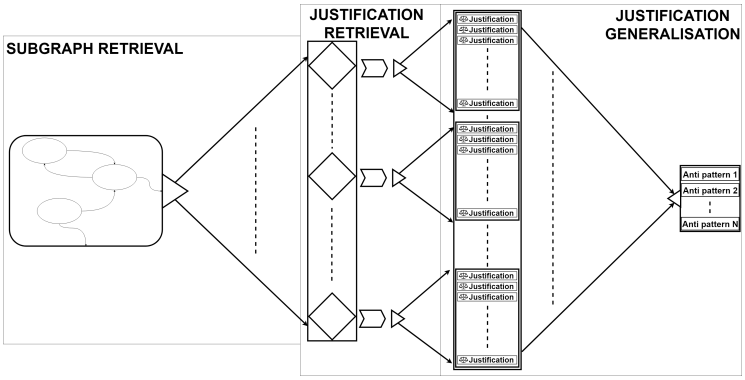



Fig. 3: A schematic diagram that shows the pipeline used to extract subgraphs, find justifications and their anti-patterns.

is impractical. To speed up this process or even make it feasible for some larger KGs, we split the KG into smaller subgraphs. Each subgraph is generated by extending a root node, which is retrieved by taking a triple from the complete graph and taking a node that is in the subject position as the starting point. The graph is expanded by finding all the triples that have the root node as the subject, and these triples are added to the subgraph. Next, all the nodes in the object position are expanded, and the graph is expanded as long as possible, or until the maximum amount of triples set by the user is reached. In Section 5, we empirically evaluate what would be the optimal subgraph size limit, based on the trade-off between scalability of the approach and its completeness in terms of the detected anti-patterns.

**2. Justification retrieval.** Out of these newly formed subgraphs, we only care about the ones that are inconsistent. Therefore, we check firstly for the consistency of each of these subgraphs and discard the consistent ones. Then for each of the inconsistent subgraphs, we run a justification retrieval algorithm to retrieve the detected contradiction(s) with their justifications. For this, we use the justification retrieval algorithm in the Openllet reasoner. The algorithm walks through the graph and finds the minimal justification for each contradiction. It continues to search for justifications until no more justification can be found in the graph[2]. This step is executed for each subgraph, and all the justifications are then pushed through the extraction pipeline to the next stage.

**3. Justification generalisation.** While most justifications are different, as each justification is a set of instantiated triple pattern, the underlying non-

[2] since justification retrieval algorithms can potentially run for a long time in the search for additional justifications in a KG, we set a runtime limit between 10 and 20 seconds based on the considered subgraph size

instantiated BGPs do not have to be. The underlying BGP forms the basis of the anti-patterns. To retrieve all anti-patterns from the detected justifications, we first generalise the justification to an anti-pattern by removing the instantiated subject and object on the nodes (and also the predicate in the case described in Section 4.1). Justifications with the same underlying pattern are grouped together. Therefore, given a justification and its generalisation into anti-pattern, we check whether an anti-pattern with the same structure already exists. Comparing anti-patterns with different variable names consists in checking whether these anti-patterns are isomorphic. For this, we implement a version of the VF2 algorithm [4], with the addition of matching the instantiated edges of the anti-patterns (i.e. for predicates that do not also appear in the subject or object position of the same justification). If the anti-pattern of a justification is matched to an existing anti-pattern, we group this justification with the other justifications generalised by this anti-pattern. Otherwise, a new anti-pattern is formed as a generalisation of this justification. This algorithm continues until all justifications have been matched to their corresponding anti-pattern.

**Implementation.** The source code of our approach is publicly available[3]. It is implemented in JAVA, and relies on a number of open-source libraries, mainly *jena*[4], *hdt-java*[5], *openllet*[6], and *owlapi*[7]. All experiments in the following sections have been performed on an Ubuntu server, 8 CPU Intel 2.40 GHz, with 256 GB of memory.

## 5 Experiments

As a way of emphasising scalability over completeness, our approach for finding anti-patterns from any inconsistent KG implements an initial step that consists of splitting the KG into smaller and overlapping subgraphs.

In the first part of these experiments (section 5.1), we empirically evaluate the impact of the subgraph size limit on the efficiency of the approach. Then, based on the adopted subgraph size limit deducted from the first experiment, we show (in section 5.2) the scalability of our approach on some of the largest KGs available on the Web.

### 5.1 Completeness Evaluation

In this section, we measure the impact of splitting the KG both on the number of detected anti-patterns, and the runtime of the approach. The goal of this experiment is to ultimately find the optimal subgraph size limit to consider

[3] `https://github.com/thomasdegroot18/kbgenerator`
[4] `https://jena.apache.org`
[5] `https://github.com/rdfhdt/hdt-java`
[6] `https://github.com/Galigator/openllet`
[7] `https://owlcs.github.io/owlapi`

in the first step of our approach. For evaluating completeness, this experiment requires datasets in which *Openllet* can retrieve (almost) all contradictions with their justifications. For this, we rely on two relatively small datasets:

- **Pizza ontology:** dataset of 1,944 triples serving as a tutorial for OWL and Protégé. We choose this dataset based on the fact that its contradictions and anti-patterns are known, and therefore can represent a gold standard for our approach.
- **Linked Open Vocabularies (LOV):** dataset of 888,017 triples representing a high quality catalogue of reusable vocabularies for the description of data on the Web [20]. We choose this dataset since it is small enough to retrieve almost all of its justifications using *openllet*.

Depending on the size of the dataset, we choose various subgraph size limits and observe the number of detected anti-patterns and the runtime for each of the three steps of our approach.

Table 1: Impact of the subgraph size limit on the number of detected anti-patterns and the runtime of the approach (in seconds) for the Pizza dataset.

| **Subgraph Size Limit** | **Number of Anti-patterns** | **Total Runtime** | **Step 1 Runtime** | **Step 2 Runtime** | **Step 3 Runtime** |
|---|---|---|---|---|---|
| 50 | 2 | 3.22 | 0.8 | 2.41 | 0.01 |
| 100 | 2 | 4.38 | 1.33 | 3.04 | 0.01 |
| 250 | 2 | 7.93 | 2.99 | 4.89 | 0.05 |
| 500 | 2 | 13.07 | 5.59 | 7.44 | 0.04 |
| 750 | 2 | 18.21 | 7.95 | 10.2 | 0.08 |
| 1,000 | 2 | 23.15 | 10.26 | 12.84 | 0.08 |
| 2,000 | 2 | 50.5 | 20.99 | 28.41 | 1.06 |

Table 1 presents the results of this experiment on the Pizza dataset. This experiment shows that splitting the Pizza dataset into various subgraphs with a maximum size of 50 triples (2.57% of the dataset size) significantly reduces the runtime of our approach compared to splitting the KG into larger subgraphs. Most importantly, this decrease in the total runtime does not impact the number of the detected anti-patterns. Looking further into the runtime of the different steps, the improvement is mostly noticeable in the first two steps of the approach. This shows that it is more efficient to construct smaller subgraphs (i.e. step 1), and retrieve separately all their justifications (i.e. step 2). Although the number of detected anti-patterns does not vary between the seven experiments, we notice a slight increase in the runtime of generalising justifications to anti-patterns (i.e. step 3). This is due to the fact that we split the KG into several overlapping subgraphs, regardless of the subgraph size limit. This means that when larger subgraphs are constructed, the same justifications are found several times, and

hence requires to be redundantly generalised and compared with existing anti-patterns.

Table 2: Impact of the subgraph size limit on the number of detected anti-patterns and the runtime of the approach (in seconds) for the LOV dataset.

| Subgraph Size Limit | Number of Anti-patterns | Total Runtime | Step 1 Runtime | Step 2 Runtime | Step 3 Runtime | Number of Subgraphs |
|---|---|---|---|---|---|---|
| 500 | 0 | 1,783.12 | 215.72 | 1,565.7 | 1.7 | 101,673 |
| 1,000 | 2 | 3,505.02 | 428.72 | 3,073.1 | 3.2 | 50,960 |
| 2,500 | 31 | 4,328.82 | 570.52 | 3,749.1 | 9.2 | 20,436 |
| 5,000 | 39 | 4,525.15 | 667.85 | 3,829.3 | 28 | 10,218 |
| 7,500 | 39 | 4,976.92 | 694.02 | 4,271.8 | 11.1 | 6,812 |
| 10,000 | 39 | 5,105.9 | 739.2 | 4,348.9 | 17.8 | 5,109 |
| 25,000 | 39 | 5,346.67 | 835.07 | 4,493.4 | 18.2 | 2,041 |
| 50,000 | 39 | 5,497.3 | 857.8 | 4,615.4 | 24.1 | 1,014 |
| 100,000 | 39 | 5,758.3 | 946.4 | 4,791.8 | 20.1 | 507 |

Table 2 presents the results of this experiment for the Linked Open Vocabulary dataset. This experiment shows that choosing a relatively small subgraph size limit (1,000 triples or less) in larger datasets has clear consequences on the number of detected anti-patterns. On the other hand, increasing the subgraph size limit to more than 5,000 triples does not help detecting any additional anti-patterns. It just results in the increase of the runtime by up to 27% (from 4,525 seconds up to 5,758 seconds). Similarly to the previous experiment, this increase in the runtime is mostly noticeable in the first two steps of the approach. Specifically, constructing only 507 subgraphs with a size limit up to 100,000 triples takes 4.4 times more than constructing 101,673 subgraphs with a size limit of 500 triples. This observation suggests that despite being a catalogue of different data vocabularies, the LOV dataset is quite connected, which results in connected components possibly up to 100,000 triples to be generated.

### 5.2 Scalability Evaluation

In the second part of these experiments, we evaluate the scalability of our approach on some of the larger and most popular KGs in the Web. We choose the three following datasets, after testing that they are all indeed inconsistent:

- **YAGO:** dataset of more than 158 million triples covering around 10 million entities derived from Wikipedia, WordNet and GeoNames [17].
- **DBpedia (English):** dataset of over 1 billion triples covering 4.58 million entities extracted from Wikipedia [1].
- **LOD-a-lot:** dataset of over 28 billion triples based on a crawl of a very large subset of the LOD Cloud in 2015 [8].

Based on the results of the previous experiment, we set the subgraph size limit to 5,000 triples and run our approach for each of these three datasets. Table 3 shows that finding most anti-patterns from a large KG is feasible, but computationally expensive. Specifically, detecting 135 and 13 anti-patterns from *YAGO* and *DBpedia* takes around 4 and 13 hours respectively. While on the other hand detecting 222 anti-patterns from the *LOD-a-lot* takes almost a full week. Considering the size of the dataset, and the number of different namespaces (which we use as proxies for dataset provenance), the results suggest that *YAGO* contains the most anti-patterns compared to *DBpedia* and *LOD-a-lot*.

Table 3: Results of detecting anti-patterns from three of the largest KGs in the Web.

| | **LOD-a-lot** | **DBpedia** | **YAGO** |
|---|---|---|---|
| number of triples | 28,362,198,927 | 1,040,358,853 | 158,991,568 |
| number of namespaces | 9,619 | 20 | 11 |
| number of distinct anti-patterns | 222 | 13 | 135 |
| largest anti-pattern size | 19 | 12 | 16 |
| *runtime (in hours)* | *157.56* | *13.01* | *3.97* |

## 6 KGs Inconsistency Analysis

In the previous section, we showed that it is feasible to detect anti-patterns from some of the largest KGs in the Web, when the KG is split into overlapping subgraphs with a maximum size of 5,000 triples. In this section, we further analyse these retrieved anti-patterns and compare the inconsistency characteristics between these three KGs.

### 6.1 What is the most common size of anti-patterns?

We already saw from Table 3 that the largest anti-patterns in the *LOD-a-lot*, *DBpedia*, and *YAGO* contain respectively 19, 12, and 16 edges. Looking at their size distribution, Figure 4 shows that the most common size of anti-patterns in the three KGs ranges between 11 and 14 edges.

### 6.2 What are the most common types of anti-patterns found in these KGs?

Anti-patterns represent a generalised notion of justifications that describe common mistakes in a KG. In our analysis of the detected anti-patterns in these three KGs, we found that a number of the different anti-patterns refer to an even more general type of mistakes, and can be further grouped together. This general type of anti-patterns consists of anti-patterns with the same structure of nodes and

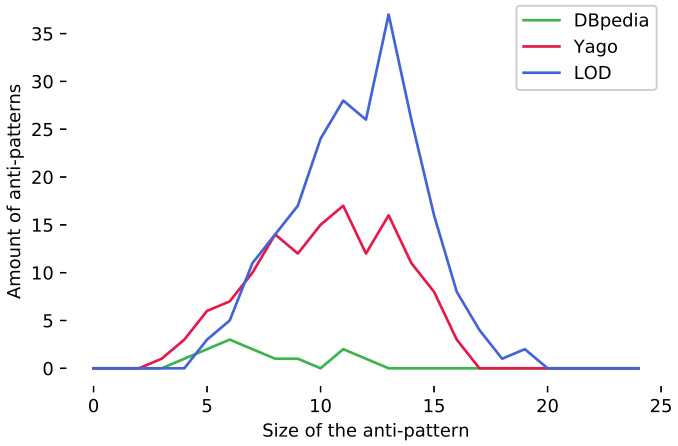



Fig. 4: Size distribution of the anti-patterns in these three KGs.

edges, but with different size. For instance, Figure 5 presents two anti-patterns, with different size, but referring to the same more general type of mistake: one instance (blue node) as a member of two classes (green nodes directly related to the blue node) that are descendants of two disjoint classes. The only difference between the anti-pattern on top of the figure with the one on the bottom, is that the former is of size 13 whilst the latter is of size 7. These anti-patterns can be grouped together and referred to as *cycle graphs anti-patterns*.

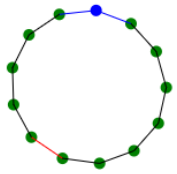
Based on this principle, we can distinguish between three general types of anti-patterns found in these investigated KGs: *cycle graphs*, *kite graphs*, and *domain or range-based graphs*. While the first type of anti-patterns is presented in Figure 5, examples of kite graphs anti-patterns are presented in Figure 6, and domain or range-based anti-patterns were previously presented in Figure 2 for contradiction B. Table 4 presents the distribution of these general type of anti-patterns in the three investigated KGs. It shows that kite graphs anti-patterns are the most common in the *LOD-a-lot* and *YAGO*, whilst cycle graph anti-patterns are the most common in *DBpedia*. All detected variants of these three general type of anti-patterns can be explored online[8].

Table 4: General types of anti-patterns found in these three KGs.

| **Type of Anti-patterns** | **LOD-a-lot** | **DBpedia** | **YAGO** |
|---|---|---|---|
| Cycle graphs | 54 | 12 | 11 |
| Kite graphs | 156 | 1 | 108 |
| Domain or Range-based graphs | 12 | 0 | 16 |

[8] `https://thomasdegroot18.github.io/kbgenerator/Webpages/statisticsOverview.html`

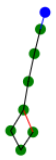
```
SELECT * WHERE {
?C1 <http://www.w3.org/2002/07/owl#disjointWith> ?C10.
?a0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?C6.
?a0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?C8.
?C4 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?C5.
?C5 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?C9.
?C8 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?C4.
?C7 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?C11.
?C3 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?C0.
?C11 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?C2.
?C6 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?C7.
?C2 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?C3.
?C0 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?C1.
?C9 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?C10.
}
```



```
SELECT * WHERE {
?a0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?C0.
?a0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?C4.
?C3 <http://www.w3.org/2002/07/owl#disjointWith> ?C5.
?C2 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?C3.
?C4 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?C5.
?C0 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?C1.
?C1 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?C2.
}
```

Fig. 5: Cycle graph anti-pattern.

```
SELECT * WHERE {
?C2 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?C5.
?C1 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?C2.
?C6 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?C4.
?C0 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?C3.
?C5 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?C6.
?C3 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?C1.
?C2 <http://www.w3.org/2002/07/owl#disjointWith> ?C4.
?a0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?C0.
}
```



```
SELECT * WHERE {
?a0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?C0.
?C3 <http://www.w3.org/2002/07/owl#disjointWith> ?C4.
?C1 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?C2.
?C5 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?C4.
?C5 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?C1.
?C0 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?C5.
?C2 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?C3.
}
```

Fig. 6: Kite graph anti-pattern.

### 6.3 What is the benefit in practice of generalising justifications into anti-patterns?

In addition to the fact that justifications are domain-dependent and possibly complex, understanding and analysing justifications of contradictions can also be impractical due to their redundancy and frequency. This is particularly true in the three large investigated KGs, as we can see in Figure 7. This plot presents the distribution of justifications per anti-pattern for these three KGs. It shows that the detected anti-patterns in the *LOD-a-lot*, *DBpedia*, and *YAGO* make the billions of available contradictions more interpretable by generalising them into 222, 13, and 135 anti-patterns, respectively. Specifically, Table 5 shows that on average each anti-pattern generalises around 5M, 7.7K, and 133K justifications in the *LOD-a-lot*, *DBpedia*, and *YAGO* respectively. It also shows that a single anti-pattern in the *LOD-a-lot* generalises up to 45M retrieved justifications, and that interestingly the *LOD-a-lot*, a dataset that largely represents the current status of the LOD Cloud, contains over a billion contradictions.

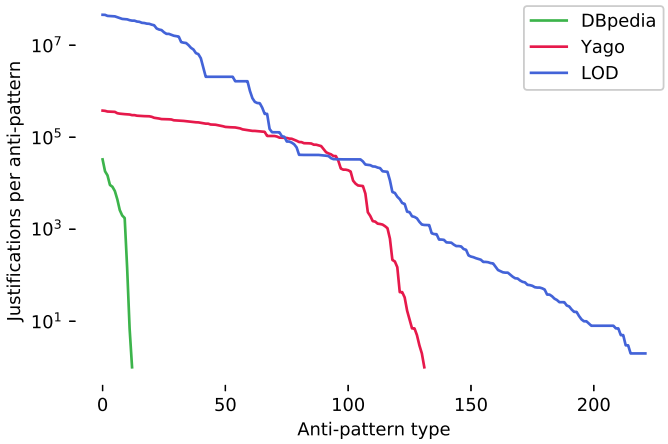



Fig. 7: Distribution of justifications per anti-pattern.

Table 5: Impact of generalising justifications to anti-patterns

| Justification per Anti-Pattern | LOD-a-lot | DBpedia | YAGO |
|---|---|---|---|
| Maximum | 45,935,769 | 32,997 | 379,546 |
| Minimum | 2 | 1 | 1 |
| Average | 4,988,176.9 | 7,796.07 | 133,998.31 |
| Median | 23,126 | 4,469 | 106,698 |
| Total | 1,107,375,273 | 101,349 | 1,8089,773 |

## 7 Conclusion

In this work, we introduced anti-patterns as minimal sets of (possibly) uninstantiated basic triple patterns that match inconsistent subgraphs in a KG. We can use anti-patterns to locate, generalise, and analyse types of contradictions. Retrieving contradictions from a KG and finding the extent to which a KG is inconsistent can now be formulated as a simple SPARQL query using anti-patterns as BGPs. Our second contribution is a pipeline that extracts anti-patterns from inconsistent KGs. In theory, such approach can extract all anti-patterns, but the implementation presented in this work does not guarantee completeness due to the splitting of the KG for scalability reasons. The source code, as well as the detected anti-patterns are publicly available.

Finally, we showed on small KGs that our approach can detect in practice all anti-patterns, and showed on KGs of billions of triples that our approach can be applied in the context of the Web of Data. Specifically, we showed on commonly used KGs such as the *LOD-a-lot*, *DBpedia*, and *YAGO* that billions of justifications can be generalised into few hundreds of anti-patterns. While these findings prove the spread of billions of logically contradicting statements in the Web of Data, this work also shows that these contradictions can now be easily located and possibly repaired, as they only refer to three different general patterns.

As an extension of this work, we want to exploit the strict semantics of 558 million *owl:sameAs* links recently made available [2]. Their closure consisting of over 35 billion triples can allow us to characterise, analyse and potentially benefit from additional anti-patterns to repair the existing contradictions in the Web of Data.

## References

1. Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., Ives, Z.: Dbpedia: A nucleus for a web of open data. In: The semantic web, pp. 722–735. Springer (2007)
2. Beek, W., Raad, J., Wielemaker, J., Van Harmelen, F.: sameas. cc: The closure of 500m owl: sameas statements. In: European Semantic Web Conference. pp. 65–80. Springer (2018)
3. Bonte, P., Ongenae, F., Schaballie, J., De Meester, B., Arndt, D., Dereuddre, W., Bhatti, J., Verstichel, S., Verborgh, R., Van de Walle, R., et al.: Evaluation and optimized usage of owl 2 reasoners in an event-based ehealth context. In: 4e OWL reasoner evaluation (ORE) workshop. pp. 1–7 (2015)
4. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: A (sub) graph isomorphism algorithm for matching large graphs. IEEE transactions on pattern analysis and machine intelligence **26**(10), 1367–1372 (2004)
5. Debattista, J., Lange, C., Auer, S., Cortis, D.: Evaluating the quality of the lod cloud: An empirical investigation. Semantic Web (Preprint), 1–43 (2018)
6. Eiter, T., Fink, M., Schüller, P., Weinzierl, A.: Finding explanations of inconsistency in multi-context systems. Artificial Intelligence **216**, 233–274 (2014)

7. Färber, M., Bartscherer, F., Menne, C., Rettinger, A.: Linked data quality of dbpedia, freebase, opencyc, wikidata, and yago. Semantic Web **9**(1), 77–129 (2018)
8. Fernández, J.D., Beek, W., Martínez-Prieto, M.A., Arias, M.: Lod-a-lot. In: International Semantic Web Conference. pp. 75–83. Springer (2017)
9. Gupta, A., Nishimura, N.: The complexity of subgraph isomorphism for classes of partial k-trees. Theoretical Computer Science **164**(1-2), 287–298 (1996)
10. Horridge, M., Parsia, B., Sattler, U.: Explaining inconsistencies in owl ontologies. In: International Conference on Scalable Uncertainty Management. pp. 124–137. Springer (2009)
11. Huang, Z., Van Harmelen, F., Ten Teije, A.: Reasoning with inconsistent ontologies. In: IJCAI. vol. 5, pp. 254–259 (2005)
12. Kaminski, T., Knorr, M., Leite, J.: Efficient paraconsistent reasoning with ontologies and rules. In: Twenty-Fourth International Joint Conference on Artificial Intelligence (2015)
13. Noori, A., Moradi, F.: Simulation and comparison of efficency in pathfinding algorithms in games. Ciência e Natura **37**(6-2), 230–238 (2015)
14. Paulheim, H.: Knowledge graph refinement: A survey of approaches and evaluation methods. Semantic web **8**(3), 489–508 (2017)
15. Plessers, P., De Troyer, O.: Resolving inconsistencies in evolving ontologies. In: European Semantic Web Conference. pp. 200–214. Springer (2006)
16. Schlobach, S., Cornet, R., et al.: Non-standard reasoning services for the debugging of description logic terminologies. In: Ijcai. vol. 3, pp. 355–362 (2003)
17. Suchanek, F.M., Kasneci, G., Weikum, G.: Yago: a core of semantic knowledge. In: Proceedings of the 16th international conference on World Wide Web. pp. 697–706. ACM (2007)
18. Töpper, G., Knuth, M., Sack, H.: Dbpedia ontology enrichment for inconsistency detection. In: Proceedings of the 8th International Conference on Semantic Systems. pp. 33–40. ACM (2012)
19. Toulmin, S.E.: The uses of argument. Cambridge university press (1956)
20. Vandenbussche, P.Y., Atemezing, G.A., Poveda-Villalón, M., Vatant, B.: Linked open vocabularies (lov): a gateway to reusable semantic vocabularies on the web. Semantic Web **8**(3), 437–452 (2017)
21. Zhi, H.l.: A max-term counting based knowledge inconsistency checking strategy and inconsistency measure calculation of fuzzy knowledge based systems. Mathematical Problems in Engineering **2015** (2015)