

Bottom-up Discovery of Context-aware Quality Constraints for Heterogeneous Knowledge Graphs

Xander Wilcke¹^a, Maurice de Kleijn²^b, Victor de Boer¹^c, Henk Scholten²
and Frank van Harmelen¹^d

¹*Dept. of Computer Science, Vrije Universiteit Amsterdam, The Netherlands*

²*Dept. of Spatial Economics, Vrije Universiteit Amsterdam, The Netherlands*

Keywords: Knowledge Graphs, Data Validation, Data Quality, Constraints, Pattern Mining.


Abstract: As knowledge graphs are getting increasingly adopted, the question of how to maintain the validity and accuracy of our knowledge becomes ever more relevant. We introduce context-aware constraints as a means to help preserve knowledge integrity. Context-aware constraints offer a more fine-grained control of the domain onto which we impose restrictions. We also introduce a bottom-up anytime algorithm to discover context-aware constraint directly from heterogeneous knowledge graphs—graphs made up from entities and literals of various (data) types which are linked using various relations. Our method is embarrassingly parallel and can exploit prior knowledge in the form of schemas to reduce computation time. We demonstrate our method on three different datasets and evaluate its effectiveness by letting experts on knowledge validation and management assess candidate constraints in a real-world knowledge validation use case. Our results show that overall, context-aware constraints are to an extent useful for knowledge validation tasks, and that the majority of the generated constraints are well balanced with respect to complexity.


1 INTRODUCTION


Knowledge graphs have ceased to be the academic experiment that they once were. They are now confidently present in the working environment of many different institutes, museums, and businesses around the globe, such as the Smithsonian museum of American art (Szekely et al., 2013), taxi service Uber (Hamad et al., 2018), and even internet giants such as Google (Singhal, 2012) and Facebook (Sun and Iyer, 2013) have firmly embedded knowledge graphs into their services. With this newly conquered position it becomes ever more important to not only look at how to engineer this knowledge, but also how to maintain the quality of this knowledge across its entire life cycle, every step of which is prone to suffer from a loss in said quality by the introduction of various artefacts (Fürber, 2015). These artefacts come in many forms, ranging from false, illegal, and missing attribute values to incorrect, inconsistent, and contra-


dictory relationships. Failure to correct these artefacts can have severe negative effects on the operations and decision making processes, which is why quality control is a vital step in any knowledge management process (Tayi and Ballou, 1998).

A key component of a modern quality control process is the *quality constraint*: an externally given rule which specifies criteria that correspond to high-quality knowledge, and which can be used to validate a knowledge base in an automated fashion (Fürber and Hepp, 2011). For knowledge graphs, simple quality constraints can be defined using OWL¹, or, if more sophisticated constraints are needed, by using constraint languages such as *ShEx*² or the more recent *SHACL*³. Constraint languages such as these apply restrictions on the schema level, for instance to all members of a certain class or to every value of a certain attribute. This is analogous to constraint languages for relational databases, and works well if the members of a class form a single homogeneous group. If this is not the case however, and the members within a class

^a <https://orcid.org/0000-0003-2415-8438>

^b <https://orcid.org/0000-0003-2379-191X>

^c <https://orcid.org/0000-0001-9079-039X>

^d <https://orcid.org/0000-0002-7913-0048>

¹See <https://www.w3.org/TR/owl-ref/>

²See <https://shex.io/>

³See <https://www.w3.org/TR/shacl/>

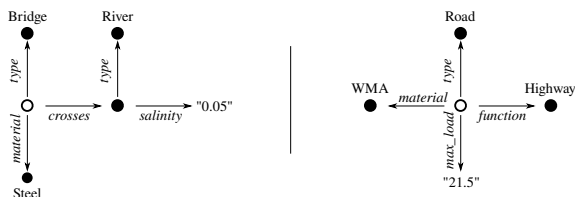


Figure 1: Two example subgraphs from the asset management domain, with Left) a steel bridge crossing a salt-water river, and Right) a section of road on the highway. Circles represent entities, with open circles depicting focal entities. Literals are shown as strings.

form two or more distinct clusters with their own peculiarities, it may occur that constraints which apply to one cluster do not necessarily apply to the other(s).

Consider for example the two subgraphs in figure 1 from a knowledge graph about asset management. On the left we can see a steel bridge which crosses a salt-water river, whereas on the right we can see a section of road on a highway which is made from *WMA* (a type of asphalt) and which has a certain load-bearing capacity (in metric tons). For the bridge example, a schema level constraint might state that all bridges must be constructed from a certain building material. Similarly, a schema-level constraint for the road example might tell us that the value of attribute *max_load* must lie between zero and one hundred, and must be of the data type *float*. Constraints such as these work well for identifying illegal or missing values and relationships, but at the same time overlook the different characteristics that the members of a class are likely to have: a bridge might have different material demands depending on the salinity levels of its environment, and the load-bearing capacity of roads might vary depending on its material and usage.

To impose restrictions on this more fine-grained level it is necessary to condition constraints not on the schema level, but rather on level of the clusters whose members share similar characteristics (Bohannon et al., 2007). This can be achieved by generalizing constraints over nodes with similar contexts. We call such constraints *context aware*. In this work, we introduce a new formalism to define context-aware quality constraints on heterogeneous knowledge graphs. Constraints of this kind offer a fine-grained control over the domain upon which to impose restrictions. This domain is determined by a so-called *contextual pattern* that describes a special graph motif that the nodes need to match. Contextual patterns can contain entities (by IRI) and/or literals (by value), and also offer means to generalize to classes, data types, and value patterns (e.g. ranges and regular expressions). These same options are also available to restrictions.

Context-aware constraints can be defined by hand, or top down, but doing so quickly becomes infeasible as the dimensions and the diversity of the knowledge grow. An alternative is to learn suitable quality constraints from the knowledge itself, or bottom up, by mining frequent patterns in the graph and by encoding these patterns as constraints (Tayi and Ballou, 1998). This works on the supposition that the large majority of the knowledge is valid and accurate, and that these qualities can be captured in a set of patterns. We apply this approach in this paper. For this purpose, we introduce a bottom-up anytime algorithm to discover context-aware constraints directly from heterogeneous knowledge graphs. Our algorithm is *embarrassingly parallel* and generates constraints by exploring and testing increasingly more complex contextual patterns in a breadth-first fashion. Special attention is given to the multimodal nature of many knowledge graphs by enabling our algorithm to learn patterns over various data types, such as dates, numbers, and texts.

We evaluate our method in two ways. Firstly, from an algorithmic perspective for which we demonstrate and test an implementation of our method on three different datasets and evaluate the constraints it is able to generate. Secondly, from a user perspective by letting knowledge management experts assess the generated constraints in a real-world knowledge validation task.

To summarize, our main contributions are **1)** a novel graph-based constraint formalism to define restrictions on the contextual level, **2)** an anytime algorithm for the bottom-up generation of context-aware constraints from heterogeneous knowledge graphs, and **3)** a *user-driven* evaluation of the method and constraints by experts in a real-world knowledge validation use case.

2 RELATED WORK

Several mature standards exist with which constraints for knowledge graph can be defined. One of these standards is the *Web Ontology Language*, better known as OWL, which supports simple value and cardinality constraints. More expressive constraints can be defined using dedicated constraint languages such as ShEx or SHACL, which offer capabilities similar to their counterparts for relational databases. A subset of these capabilities is also supported by our work, such as placing restrictions on values and datatypes. However, ShEx and SHACL are designed around a different paradigm in which the focus lies on schema-level constraints, whereas the constraints proposed in

Table 1: All five variants of assertion patterns with their corresponding domains in set-builder notation. In all cases, the left-hand side object-type variable can be substituted for v_{ot}^* , which matches all types.

	Assertion Pattern	Domain
1	$p_k(v_{ot}^t, e_j)$	$\{e_i \in \mathcal{E} \mid type(e_i, t) \wedge (\exists e_j \in \mathcal{E}) [p_k(e_i, e_j)]\}$
2	$p_k(v_{ot}^t, v_{ot}^{t'})$	$\{e_i \in \mathcal{E} \mid type(e_i, t) \wedge (\exists e_j \in \mathcal{E}) [p_k(e_i, e_j) \wedge type(e_j, t')]\}$
3	$p_k(v_{ot}^t, l_j)$	$\{e_i \in \mathcal{E} \mid type(e_i, t) \wedge (\exists l_j \in \mathcal{L}) [p_k(e_i, l_j)]\}$
4	$p_k(v_{ot}^t, v_{dt}^{t'})$	$\{e_i \in \mathcal{E} \mid type(e_i, t) \wedge (\exists l_j \in \mathcal{L}) [p_k(e_i, l_j) \wedge dtype(l_j, t')]\}$
5	$p_k(v_{ot}^t, v_{re}^s)$	$\{e_i \in \mathcal{E} \mid type(e_i, t) \wedge (\exists l_j \in \mathcal{L}) [p_k(e_i, l_j) \wedge match(l_j, s)]\}$

this work operate on the contextual level. A behaviour similar to context-aware constraints can nevertheless be achieved using SHACL by specifying the filter shapes introduced by SHACL’s advanced features.

While ShEx and SHACL managed to grow into mature standards, they are not the first to introduce more expressive constraints for knowledge graphs. The work in (Cortés-Calabuig and Paredaens, 2012) already discusses the different types of constraints that can be defined on knowledge graphs from a theoretical perspective, together with their satisfaction and entailment problems. In (Lausen et al., 2008), the authors show how the popular query language SPARQL can be used to retain knowledge integrity when converting relational databases to knowledge graphs. Both these studies consider only schema-level constraints similar to those of ShEx and SHACL.

Association rules have been the interest of several works trying to adapt them to knowledge graphs. Association rules are implications of the form $X \implies y$, where the presence of a set of instances X implies the presence of another instance y . Generalized association rules works largely the same, except that X holds the types associated with these instances. Both variants can be expressed using context-aware constraints. A straightforward approach to bring association rules to knowledge graphs is shown in (Anbutamilazhagan and Selvaraj, 2014), which flattens the graph into transactions and feeds these to the Apriori algorithm. This is different from the approach used in our work, which is specifically tailored to graphs. A more similar method is presented in (Ramezani et al., 2014), which operates directly on graphs and which allows for multi-relational patterns. Such patterns can be seen as selective contexts, whereas context-aware constraints consider the entire context. In (Barati et al., 2016), the authors introduce a graph-based approach which can exploit common RDF and RDFS semantics to infer type hierarchies. Exploiting common semantics is also part of our method, but is used to infer direct types and datatypes rather than generalizations thereof.

Quite some work is done in bringing functional dependencies (FD) to knowledge graphs, e.g. (Akhtar

et al., 2010; Calvanese et al., 2014; Hellings et al., 2016). A FD $X \rightarrow Y$ expresses that entities with the same values for all attributes in X must also have the same values for those in Y . This behaviour can be approached by context-aware constraints, but only for values which are already present in the graph. In (He et al., 2014; Yu and Heflin, 2011), the authors extend FDs with paths, which can be thought of as selective or pruned contexts. The work in (Fan and Lu, 2017; Fan et al., 2016) is closest to context-aware constraints by letting FDs consist of graph motifs with support for entities, literals, and variables. In (Yu and Heflin, 2011), the authors introduce FDs with numeric patterns by clustering values using k -means. We employ a similar strategy to learn patterns for numbers, dates, and strings (see Sc. 4.1.1).

Some work has been done on automatic constraint discovery from knowledge graphs. In (He et al., 2014), the authors accomplish this by first flattening a graph into transactions, from which they mine frequent patterns that are fed to an off-the-shelf algorithm for discovering FDs. This differs from our approach, which is specifically developed for graphs. More similar methods are used by (Fan et al., 2018; Yu and Heflin, 2011), which start out with minimal constraints and extend these iteratively until all options are exhausted. However, these algorithms only consider FDs.

General rule miners based on inductive logic programming (Tresp et al., 2008) or frequent-pattern mining (e.g. (Galárraga et al., 2013; Meilicke et al., 2019)) can also be used to discover constraints. However, these methods generally focus on the relational structure of a graph and its underlying schemas without considering contextual dependencies and/or literal values.

To the best of our knowledge, we are the first to evaluate work of this kind from a user perspective. All other reviewed work employs a theoretical and/or data-driven evaluation.

Algorithm 1: Initialization of generation forest—simplified. Returns all constraints of size 1 with minimal support and confidence. Support for object/data type and value patterns are omitted here, but are similar to line 7–10 with an additional few steps. In line 10, v_{ot}^t is used as shorthand for $type(\cdot, t)$, and a dummy self relation is added which is needed in Alg. 2.

```

1: function INITGENERATIONFOREST( $G, supp_{min}, conf_{min}$ )
2:    $types = \{t \in \mathcal{E} \mid (\exists e \in \mathcal{E}) [type(e, t)]\}$ 
3:   for type  $t$  in  $types$  do
4:      $\Omega(t, 0) := \emptyset$ 
5:     if  $|\{e \in \mathcal{E} \mid type(e, t)\}| \geq supp_{min}$  then
6:       for  $p \in \mathcal{P}$  do
7:          $S := \{p(e, r) \mid (\exists e \in \mathcal{E}, \exists r \in \mathcal{R}) [p(e, r) \in \mathcal{A} \wedge type(e, t)]\}$ 
8:         for  $p(\cdot, r) \in S$  do
9:           if  $|p(\cdot, r) \in S| \geq conf_{min}$  then
10:             $\phi := p(v_{ot}^t, r) \leftarrow \{self(v_{ot}^t, v_{ot}^t)\}$ 
11:             $\Omega(t, 0) := \Omega(t, 0) \cup \{\phi\}$ 
12:   return  $\Omega$ 
    
```

3 DEFINING CONSTRAINTS

In this section, we provide a definition of context-aware constraints. Let $G = (\mathcal{R}, \mathcal{P}, \mathcal{A})$ be a knowledge graph with the set of all resources $\mathcal{R} = \mathcal{E} \cup \mathcal{L}$, the set of all predicates \mathcal{P} , and with \mathcal{A} the set of all assertions $p_k(e_i, r_j)$ that make up G , with $p_k \in \mathcal{P}$, $e_i \in \mathcal{E}$, and $r_j \in \mathcal{R}$. Disjoint sets \mathcal{E} and \mathcal{L} consist of all entities and literals in \mathcal{R} , respectively.

A constraint $\phi = c \leftarrow A$ states that every entity $e \in \mathcal{E}$ which satisfies antecedent $A = a_1 \wedge a_2 \wedge \dots \wedge a_n$ must also satisfy consequent c . We can more intuitively think of this as the restriction c we wish to impose upon the domain $\mathcal{E}_A \subseteq \mathcal{E}$, with \mathcal{E}_A encompassing all entities that satisfy the condition(s) in A . Restriction c and every condition a in A take the form of *assertion patterns* $p_k(\cdot, \cdot)$, which generalize the assertions in \mathcal{A} by substituting the left and/or right-hand side resource with a *pattern variable* v . Pattern variables match any resource which fit their pattern and come in three different flavours: *object-type patterns* v_{ot}^t which match all entities of type t (e.g. Bridge or Road), *data-type patterns* $v_{dt}^{t'}$ which match all literals of data type t' (e.g. String or Geometry), and *value patterns* v_{re}^s which match all literals with a value that falls within regular expression s (e.g. “[:digit:]{2}” or “[:alnum:]*\$”). We also introduce the syntactic shorthand v_{ot}^* which matches entities of any type.

Together with the existing resources the three patterns variable allow us to construct five different assertion patterns (Table 1). In all cases, the left-hand side is an object-type variable because placing literals (or variables thereof) or entities in that spot results in illegal or unnecessary assertion patterns. For literals and data-type/value variables this is because these can never be the subject of an assertion. For entities, the resulting assertions would either apply to a single entity if they are used as consequent c , or, if used in an-

tecedent A , they would not help us reduce the domain any further than if we would just omit them (compare $p_k(v_{ot}^t, e_i) \wedge p_l(e_i, \cdot)$ to only $p_k(v_{ot}^t, e_i)$).

Antecedent A can consist of one or more conditions. These conditions can apply directly to arbitrary entities (i.e. $p_k(v_{ot}^*, \cdot)$) in which case we call them depth-1 conditions. If the right-hand side of a depth-1 condition is an object-type variable we can also chain two or more conditions to form depth- n conditions: $p_k(v_{ot}^*, v_{ot}^t) \wedge p_l(v_{ot}^t, \cdot) \wedge \dots$. The longest chain is called the depth of A , whereas its width equals the maximum number of conditions per variable. The size of A is the total number of conditions.

Each constraint ϕ is accompanied by two measures of relevance: its support and confidence. The *support* tells us the size of the domain, and equals the number of entities which satisfy A . The *confidence* tells us for how many members of the domain the restriction holds as well, and equals the number of entities which satisfy both A and c .

We only consider constraints with a single restriction c because this offers more flexibility when choosing which restrictions to apply and because it makes the measures of relevance more easily interpretable. If constraints with more than one restriction are desired we can obtain this by grouping constraints that have the same domain.

4 DISCOVERING CONSTRAINTS

Where in the previous section we provide a definition of context-aware constraints, we here provide a bottom-up anytime algorithm to efficiently discover said constraints. To do so, our algorithm starts out with all constraints that have a single condition ($|A| = 1$), which are then used as parents from which more complex constraints ($|A| > 1$) are derived by adding

new conditions. This second step is the main loop of our algorithm and operates by exploring, for every parent constraints, all sensible diagonal combinations of *candidate endpoints* and *candidate extensions*. Candidate endpoints are assertion patterns with an object-type variable on the right-hand side (Tab 1, pattern 2) which represent the leaf nodes to which we can connect another assertion pattern. This other assertion pattern is the candidate extension and can take the form of any of the assertion patterns listed in Table 1.

Constraints are derived breadth first, which ensures that we only derive new constraints from parents that meet the minimal requirements, preventing unnecessary work, and that the complexity of these new constraints increases linearly. This latter characteristic gives our algorithm an anytime property, although rather than finding “better” answers when left running, it finds ever smaller domains as more conditions are added. Differently put: the longer we let the algorithm run, the more fine grained the constraints become.

Our algorithm is embarrassingly parallel because every constraint creates a new branch of which the vertices can be computed independent of each other. The only caveat is that we need the original graph to calculate the measures of relevance for each constraint we mine. However, because the domain of child constraints is always a subsets of their parents’ domain, we can largely avoid this problem by letting parents keep a record of the entities in their domain and calculate the measures using only these.

For the remainder of this work we let all constraints be specific to object-type variables. For this reason, we will omit condition $type(v_{ot}^*, t)$ from A and change the left-hand side of restriction c from v_{ot}^* to v_{ot}^t . This effectively fixes the type to which constraints can apply, irrespective of their conditions. We limit ourselves to these cases because validation workflows are typically designed around object types.

From here on, we consider A as a set of conditions $\{a_1, a_2, \dots, a_n\}$ that all need to be satisfied.

4.1 Components

We can identify three main components in our algorithm⁴: the main loop (Sc. 4.1.2), the exploration stage (Sc. 4.1.3), and the generation forest which helps us keep track of the constraints we discover (Sc. 4.1.1). We will discuss each of these next. We also provide a simplified pseudocode which omits pruning and most optimization steps, and which does not show the generation of constraints with pattern variables (cases 2, 3, and 5 in Table 1). However,

these parts are slight variations to those shown and can easily be derived from them.

4.1.1 Generation Forest

The generation forest Ω is a data structure (e.g. a map or dictionary) which holds all discovered constraints divided over numerous generation trees. Each generation tree has a different constraint of size 1 as root, with depth $d + 1$ of the tree containing the children constraints that are obtained by adding new conditions to their parent constraints of depth d . Root constraints are of the form $p_k(v_{ot}^t, \cdot) \leftarrow \{self(v_{ot}^t, v_{ot}^t)\}$, and are generated for each entity type t for which assertion pattern $p_k(v_{ot}^t, \cdot)$ meets the minimal support and confidence. An identity condition $self(\cdot, \cdot)$ is added to serve as initial candidate endpoint for Algorithm 2 (Alg 2, line 9).

The initialization of the generation forest is shown in Algorithm 1. For each entity type in a graph of which the number of members meets the minimal support, we collect the assertions $p_k(\cdot, r)$ which occur for at least as many members as the minimal confidence requires. The assertion patterns corresponding to these assertions are combined with the entity types to form the root constraints. Algorithm 1 only shows this for assertion patterns of the form $p_k(v_{ot}^t, e_j)$ and $p_k(v_{ot}^t, l_j)$.

Type and value constraints (cases 2 and 3 in Table 1) are generated similarly, but add an additional step. For type constraints, this step involves inferring the type of object r . For entities, this is achieved by exploiting the `rdf:type` relations, whereas the `xsd:datatype` declarations are used for literals. If no (data) type is found we default to super type `rdfs:Class` and `datatype xsd:anyType` for entities and literals, respectively.

Value pattern constraints (case 5 in Table 1) are generated by clustering all values r using k -means and by translating these clusters into patterns. The optimal number of clusters is automatically determined using the elbow method. How the patterns are generated depends on the datatype. For numerical values, these patterns take the form of a range between the two outer values of a cluster. Ranges are also used for dates and timecodes, which we convert to natural numbers by encoding these as unix timestamps. For strings, the patterns consist of regular expressions that match all values in a certain cluster.

⁴Available at <https://gitlab.com/wxwilcke/cckg>

Algorithm 2: The main loop of the algorithm to discover constraints—simplified. Returns all constraints up to depth d_{max} with minimal support and confidence. Pruning and optimization steps are omitted. The longest path in A to variable u is given by $\Delta_A(u)$.

```

1: function DISCOVER( $G, d_{max}, supp_{min}, conf_{min}$ )
2:    $\Omega = \text{InitGenerationForest}(G, supp_{min}, conf_{min})$ 

3:    $d := 0$ 
4:   while  $d < d_{max}$  do
5:     for type  $t$  in  $\Omega.types()$  do
6:        $E := \emptyset$ 
7:       for  $\phi = c \leftarrow A$  in  $\Omega(t, d)$  do
8:          $C := \emptyset$ 
9:          $I := \{a \in A \mid a = p_k(\cdot, v_{ot}^{t'}) \wedge \Delta_A(v_{ot}^{t'}) = d\}$ 
10:        for  $a_i = p_k(\cdot, v_{ot}^{t'}) \in I$  do
11:           $J := \{a \mid \psi \in \Omega(t', 0) \wedge \psi = a \leftarrow A'\}$ 
12:          for  $a_j = p_l(v_{ot}^{t'}, \cdot) \in J$  do
13:             $C := C \cup \{(a_i, a_j)\}$ 
14:           $E := E \cup \text{Explore}(\phi, C)$ 
15:           $\Omega(t, d+1) := E$ 
16:           $d := d + 1$ 
17:   return  $\Omega$ 

```

4.1.2 Main Loop

The algorithm begins by generating the root constraints, which are then extended by a single level each iteration until the maximum depth is reached (Alg. 2). To do so, we begin each iteration by retrieving the previously-generated generation of constraints of depth d , which form the parents from which we derive new constraints of depth $d + 1$. The result of each iteration E is stored back in the generation forest to be used by the next iteration.

To derive new constraints from parent constraints we first retrieve the set of candidate endpoints I of a parent. The endpoints of a constraint $\phi = c \leftarrow A$ are the assertion patterns in A that are leafs and have an object-type variable $p_k(\cdot, v_{ot}^{t'})$ as object (and thus can be extended). For each of the endpoints, the matching candidate extensions J are the consequents $p_l(v_{ot}^{t'}, \cdot)$ of the root constraints for type t' . These have been generated during initialization and are therefore ensured to have the required support and confident. Together with the endpoints, the candidate extensions are passed as pairs C to Algorithm 3 where they are used to extend the parent constraints.

4.1.3 Explore

The exploration step searches through all possible diagonal combinations of parent constraint ϕ and its candidate extensions in a breadth-first fashion (Alg. 3). Concretely, if A has size n , we first explore derivatives of size $n + 1$ by adding a single extension,

Algorithm 3: Explore and extend all candidate endpoints a_i of parent constraint ϕ with candidate extensions a_j to create derived constraint χ —simplified.

```

1: function EXPLORE( $\phi, C$ )
2:    $E := \text{empty set}$ 
3:    $Q := \text{empty queue}$ 
4:    $Q.enqueue(\phi)$ 
5:   while  $Q \neq \emptyset$  do
6:      $\psi := Q.dequeue()$   $\triangleright \psi := c \leftarrow A$ 
7:     for  $a_i, a_j \in C$  do
8:        $A' := A \cup \{a_j\}$   $\triangleright a_i$  and  $a_j$  are incident
9:        $\chi := c \leftarrow A'$ 
10:      if  $supp(\chi) \geq supp_{min} \wedge conf(\chi) \geq conf_{min}$ 
11:        then
12:           $E := E \cup \{\chi\}$ 
13:           $Q.enqueue(\chi)$ 
14:   return  $E$ 

```

of which the resulting constraints form the parents from which to explore size $n + 2$. This continues until all combinations are exhausted, after which the results are returned.

A new constraint χ is generated by adding the candidate extension $a_i = p_k(\cdot, v_{ot}^{t'})$ to the parent constraint at the corresponding endpoint $a_j = p_l(v_{ot}^{t'}, \cdot)$. The derived constraint is only returned if it meets the minimum support and confidence, and if these values are not equal to that of the parent (not shown in Alg. 3).

4.2 Optimization

Our algorithm includes several optimization steps to reduce the search space. The most important steps are listed below:

- Constraints which apply to the same entities as their parent are pruned. This follows from the intuition that if the less restricted constraint has the same domain as the more restricted constraint, then the latter does not add anything over the former.
- Constraints which have already been tried via another route are excluded from creation. This can occur when their parents differ on exactly the conditions that these constraints now include.
- Sibling constraints that all have the same support and confidence values are pruned. This follows from the intuition that if the same restriction applies to overlapping domains which differ only by a single condition, then this separation between domains does not add any new information.
- Conditions that equal the restriction exactly or which are variations thereof (e.g. $p_k(v_{ot}^{t'}, e_i)$ and $p_k(v_{ot}^{t'}, v_{ot}^{t'})$ where $type(e_i, t')$) are never added.

The same holds for conditions that are incident on subject.

- Combinations of candidate endpoints and extensions for which we know (from a previous iteration) that they do not meet the minimal requirements are skipped. Assertion patterns for which this is the case are already filtered during the initialization of the generation tree.

Constraints considered for pruning are not removed immediately. Instead, we still allow these constraints to become parents for the next iteration before removal because we would otherwise lose potentially interesting (grand) children further down the branch. We call this *delayed* pruning.

5 EXPERIMENTS

We evaluate our method in two ways: firstly, from an algorithmic perspective during which we test an implementation of our method on the constraints it is able to generate, and secondly, from a user perspective by generating constraints from an in-use dataset and by letting experts assess them in a real-world knowledge validation use case.

5.1 Datasets

The constraints in our experiments are generated from three different datasets. We here provide a concise description of each of them. Table 2 lists basic statistics for each dataset.

AIFB. The AIFB dataset is a benchmark datasets for machine learning on knowledge graphs (Ristoski et al., 2016), and contains information about the staff and publications of a research institute. This dataset is the smallest of the three. Note that a modified version⁵ is used in this paper, which includes the datatype declarations needed to accurately determine the literals’ modalities. These declarations are missing in the original version.

MUTAG. The MUTAG dataset is another benchmark dataset from (Ristoski et al., 2016), and describes complex molecules by their characteristics and shape, with the focus on their carcinogenic properties. This is the largest of the three datasets used in this paper.

⁵ Available at <https://gitlab.com/wxwilcke/mmkg>

Table 2: Datasets used in the experiments. AIFB is modified to include data type declarations.

Dataset	AIFB	RWS	MUTAG
Assertions	29,219	56,364	74,567
Relations	45	305	23
Entities	6,072	3,895	32,621
Literals	5,468	12,844	1,104

RWS. The RWS dataset contains detailed knowledge about road and water constructions, including interchanges, bridges, tunnels, and many more (See e.g. Fig 1). This knowledge consists, among others, of general characteristics (year of construction, dimensions, location, etc.), maintenance reports, and administrative information. The dataset contains legacy data and has been, and still actively is, worked on by many people from several departments Rijkswaterstaat⁶, the Dutch government agency responsible for the construction and management of major infrastructure facilities in the Netherlands. Because of its long and active use, it is prone to artefacts caused by invalid or inaccurate entries, by changes in procedures over time, or by past integration or conversion issues. These aspects make this dataset a suitable choice for the task of knowledge validation.

Due to the sensitive nature of this information we are unfortunately prohibited from sharing this dataset.

5.2 Constraint Discovery

With this experiment we demonstrate our algorithm’s ability to discover context-aware constraints from heterogeneous knowledge graphs, with the intend to show the trade off between the chosen support and confidence values, and the resulting number of constraints. It is also shown what the effect of pruning has on this number. An analysis on the computation time of our algorithm is omitted due to unreliable numbers caused by running the experiments in a shared environment outside our control.

Each of the datasets listed in Table 2 is run for constraints up to depth 3 and with several different support and confidence requirements. In each case, the support and confidence values are varied between 300 and 500 with a 100-step increment, resulting in 6 combinations. These combinations are chosen based on preliminary tests, which indicated that this range was supported by all three datasets without resulting in cases where no suitable constraints can be found or where the number of constraints exceeded unmanageable amounts. No limits are placed on the width and restrictions of constraints.

⁶ www.rijkswaterstaat.nl

Table 3: Number of generated constraints for AIFB as function of chosen support and confidence values. Number of pruned constraints is listed between parenthesis.

		Support		
		500	400	300
Conf.	500	79 (64)	112 (102)	193 (206)
	400		234 (186)	315 (290)
	300			498 (476)

Table 4: Number of generated constraints for MUTAG as function of chosen support and confidence values. Number of pruned constraints is listed between parenthesis.

		Support		
		500	400	300
Conf.	500	10 (0)	11 (0)	13 (0)
	400		11 (0)	14 (0)
	300			28 (22)

5.2.1 Results & Discussion

Tables 3, 4, and 5 list the number of generated constraints as function of chosen support and confidence values for AIFB, MUTAG, and RWS, respectively. A stark difference is visible in the number of constraints generated for each dataset. Where this number is rather small for MUTAG and slightly larger for AIFB, it far exceeds the amount deemed as manageable for RWS at confidence values lower than 500.

The results indicate that there is a strong positive relation between the number of generated constraints and the used support and confidence values, as expected. However, there seems to be no direct relationship between these numbers and the size of the datasets: MUTAG, the largest dataset, has very few constraints whereas RWS, which is considerably smaller, has the largest number of constraints. Instead, the statistics in Table 2 suggest that the number of relations is more likely an indicator for the number of generated constraints.

The number of pruned constraints grows as the number of generated constraints rise, and with a similar factor. This is an expected outcome of our pruning strategy and suggests that this strategy is to an extent effective. Noteworthy is again the difference between datasets. With MUTAG and AIFB, the number of constraints generated exceeds those which are pruned, whereas the reverse is true for RWS.

Table 6 shows five constraints that were sampled from the AIFB and MUTAG output sets. The first example has a value pattern as consequent (shown simplified as range) and tells us that $\frac{1403}{1841} = 0.76\%$ of all entities of the type `Carbon-22` have a charge

which lies between -0.158 and 0.063 . The second example shows that 79% of the publications about `ID70Instance` (a certain individual) are also about `ID69Instance` (another individual). Examples 3 and 4 tell us that a compound that is mutagenic has a carbon-10 atom in 92% of the cases, while a compound which is *not* mutagenic has a hydrogen-3 atom in an equal number of cases. The final constraint shows a value pattern with a regular expression (matching e.g. "123-456"), which holds for 30% of all manuscripts that are listed in titled proceedings. The relatively low confidence to support ratio of this last example limits its usefulness and makes it a candidate for removal.

5.3 User Study

The user study takes the form of a half-day workshop with a questionnaire at the end. Participants consist of experts on knowledge management and validation who are employed at Rijkswaterstaat. During the workshop, these participants are given a presentation which explains the constraints generation process as well as the constraints themselves. After the presentation, participants are provided with a questionnaire and asked to fill it in individually.

The questionnaire is designed to investigate the trade off between the context granularity of the generated constraints and their perceived effectiveness in capturing relevant patterns in the knowledge. Constraints with increasingly finer-grained contexts are generated and presented to the participants, who are asked to rate these constraints on how relevant they are for the task of knowledge quality control in the domain of asset management. Here, *relevancy* questioned whether the presented constraints were too fine grained, too coarse grained, or whether they were somewhere in between.

Too fine-grained constraints have a relatively large number of conditions which translates to a relatively small domain. Constraints such as these are undesirable because their use is limited to only few data points. These constraints are also more likely to capture outliers, are difficult to transfer to unseen data, and can increase the total number of constraints to unmanageable amounts. Too coarse-grained constraints have few conditions and apply to a relatively large domain, and are undesirable because they limit our ability to distinguish between subsets of similar data points and can result in an increase in the number of false positives and/or negatives. Between too fine-grained and too coarse-grained lie the constraints which our participants perceive as balanced and most effective for knowledge maintenance.

Table 5: Number of generated constraints for RWS as function of chosen support and confidence values. Number of pruned constraints is listed between parenthesis.

		Support		
		500	400	300
Conf.	500	27 (56,769)	27 (56,769)	28 (58,725)
	400		76,490 (446,109)	76,491 (448,065)
	300			375,326 (732,497)

Table 6: A sample of five hand-picked context-aware constraints with their support and confidence values. All examples are simplified by omitting URIs and identity conditions, and are ordered by depth.

	Supp.	Conf.	Constraint
1	1841	1403	$charge(v_{ot}^t, [-0.158 \geq v \geq 0.063])$ $\leftarrow type(v_{ot}^t, Carbon-22)$
2	127	100	$about(v_{ot}^t, ID69Instance)$ $\leftarrow type(v_{ot}^t, Publication\ e) \wedge about(e, ID70Instance)$
3	129	119	$hasAtom(v_{ot}^t, Hydrogen-3)$ $\leftarrow type(v_{ot}^t, Compound\ e) \wedge isMutagenic(e, False)$
4	129	119	$hasAtom(v_{ot}^t, Carbon-10)$ $\leftarrow type(v_{ot}^t, Compound\ e) \wedge isMutagenic(e, True)$
5	431	131	$pages(v_{ot}^t, "[[:digit:]]{3}[:punct:]{1}[:digit:]{3}$")$ $\leftarrow type(v_{ot}^t, InProceedings\ e) \wedge bookTitle(e, dtype_v) \wedge type(dtype_v, [XSD:string])$

In addition to the above, participants are also asked about their familiarity with relevant topics, and about their opinion on the usefulness of context-aware constraints as a whole.

The questionnaire contains 3×4 constraints. Each group of 4 represents a different level of granularity, and is sampled by dividing the full set of constraints, generated from the RWS dataset, in groups of *low*, *average*, and *high* complexity. Low-complexity constraints are of depth 1, whereas average- and high-complexity constraints are of depth 2 and 3, respectively. In all cases, the context width varies between 1 and 4. No limit is placed on the type of restriction: any of those listed in Table 1 is allowed to occur. A 5-point Likert scale is used for all questions, with an additional sixth option *unsure* for the constraint granularity questions to prevent unreliable answers.

All constraint are presented as *if-then* business rules in natural language to prevent unfamiliarity with knowledge graph terminology and/or the constraint syntax to confound the results.

5.3.1 Results & Discussion

A total of 21 experts on knowledge management and validation participated in our user study. Table 7 lists the median and mode familiarity of these participants with relevant topics, and ranges from *fully disagree* to *fully agree*. Krippendorff’s alpha is used to as-

sess inter-rater agreement. Overall, the participants are moderately to very confident with their familiarity with any of the topics, but, having only a fair agreement ($\alpha = 0.26$), it seems that this level of confidence is not uniformly distributed over all participants. Irrespective, the confidence is especially strong for their knowledge of database terminology. In contrast, participants seem only moderately confident about their familiarity with the domain, which can be explained by the different departments the participants works at and the different subsets of the data these departments focus on. Nevertheless, the overall and individual confidence level(s) are strong enough to ensure that we can trust the answers our participants provide.

Table 8 shows the perceived complexity as portion of the scores for our 12 constraints combined, and for each of the three complexity groups separately. We left out the *unsure* answers to improve reliability. Overall, slightly more than half of the participants thought the complexity was well balanced, with the other four score levels dividing the remaining portion roughly equally with values between 0.08 to 0.16 each. This suggests that the generated constraints are to an extent suited for the task of knowledge validation.

An indifference between low-, average-, and high-complexity constraints is visible for all five score levels, with a minimal and maximal difference between

Table 7: Familiarity of the participants ($\alpha = 0.26$) with the domain, with data validation and data quality rules (of any form), with database terminology, and with knowledge graphs. Last column shows correlation (Kendall’s tau) with perceived usefulness (Tab. 10).

Familiarity	Median	Mode	τ
Domain	<i>neutral</i>	<i>agree</i>	0.45
Data Val.	<i>agree</i>	<i>agree</i>	0.32
Data QR	<i>agree</i>	<i>agree</i>	0.41
DB Terms	<i>fully agree</i>	<i>fully agree</i>	0.07
Kn. Graphs	<i>agree</i>	<i>agree</i>	0.25

Table 8: Relevance shown as portions of scores given by participants for constraints of low, average, and high complexity, and for all forms combined. Scores range from *far too fine grain (FG)* to *far too coarse grain (CG)*. Mode and median are *balanced* for all cases. *Unsure* scores are omitted.

Score	Low	Average	High	Comb.
<i>far too FG</i>	0.07	0.16	0.14	0.12
<i>sl. too FG</i>	0.17	0.14	0.14	0.16
<i>balanced</i>	0.51	0.53	0.57	0.53
<i>sl. too CG</i>	0.13	0.08	0.11	0.11
<i>far too CG</i>	0.11	0.08	0.04	0.08

groups of 0.03 for *slightly too fine grained* and 0.09 for *far too fine grained*, respectively. This minor difference implies that complexity does not affect the relevance of the constraints, or that the different complexity groups differ too slightly to have an impact on said relevance. This indifference is supported by significance tests (Tab 9), indicating little to no difference in distributions.

There is an overall fair to moderate agreement ($\alpha = 0.34$) on relevancy between participants when looking at the combined scores (Tab. 9). However, this agreement varies significantly when we take the complexity group into account, with only a slight to fair agreement for low-complexity constraints ($\alpha = 0.16$) to a substantial agreement for high-complexity constraints ($\alpha = 0.63$). This stark difference seems to contrast with the minor difference seen in Table 8, which suggests that more participants answered *unsure* (which were filtered) as the complexity increased.

Participants have a neutral to agreeable stance with respect to the overall usefulness of our method (Tab. 10). However, a considerable portion of the participants seems unsure about this usefulness, which supports our earlier assumption that participants became less confident as the complexity increased. Correlation analysis (Tab. 7) suggests that this effect may be part caused by (the lack of) participants’ familiar-

Table 9: Inter-rater agreement (Krippendorff’s alpha) and p-values (Kruskal-Wallis at significance level 0.05) for constraints of low, average, and high complexity, and for all forms combined. *Unsure* scores are omitted.

Complexity	α	p-value
Low	0.16	0.20
Average	0.42	0.43
High	0.63	0.58
Combined	0.34	-

Table 10: Usefulness of the method as perceived by participants in relative numbers.

Score	Portion
<i>fully disagree</i>	0.00
<i>disagree</i>	0.10
<i>neutral</i>	0.24
<i>agree</i>	0.24
<i>fully agree</i>	0.05
<i>unsure</i>	0.38

ity with the domain and with quality rules, both of which have a moderate positive relationship with the perceived usefulness. Because *unsure* has the lowest position on our Likert scale, this suggests that participants that are unfamiliar with the domain and with quality rules are also more likely to be unsure about the usefulness of the method.

6 CONCLUSION

In this work, we introduced context-aware constraints for knowledge quality control which offer a more fine-grained control over the domains on which we want to impose restrictions. We also introduced a bottom-up anytime and easily to parallelize algorithm to discover context-aware constraint directly from heterogeneous knowledge graphs.

We demonstrated our method on three different datasets, which showed that there is no direct relationship between the size of a dataset and the number of generated constraints, making it difficult to apply a rule of thumb to the chosen support and confidence values. However, our results do suggest a positive correlation between the relation count and the number of generated constraints.

Our evaluation consisted of a user study amongst experts on knowledge engineering and maintenance, which were invited to a workshop and asked to assess various constraints on asset management. Their answers indicate that, overall, context-aware constraints

are to an extent useful for knowledge validation tasks, and that the majority of the constraints were well balanced with respect to complexity. However, a considerable number of participants were nevertheless unsure about the usefulness of the method. Our analysis suggests that the lack of familiarity with the domain and quality rules might be the cause, although more in-depth study is needed.

Our algorithm contains a few noteworthy limitations. A practical limitation concern scalability as our algorithm needs to evaluate a great deal of combinations. This problem is slightly reduced by our pruning and other optimization methods, and can also be alleviated by parallelizing the task, but will nevertheless remain a challenge to deal with as the dataset increases in size and, most particularly, the number of relations. Another possible limitation lies with our assumption that the majority of the knowledge is valid and accurate. An insufficiently large enough ratio between valid/accurate and invalid/inaccurate knowledge can result in a relatively high number of false positives and negatives, reducing the usefulness of our method. A final noteworthy limitation is the high sensitivity of the provided support and confidence values, which, depending on the characteristics of the dataset, can result in too few or in an unmanageable amount of constraints. However, this is a common problem in this field of research.

We identified several potential extensions to our method which we offer as suggestions for future work. Firstly, our algorithm currently only generates a proper subset of those expressible by constraint languages such as ShEx and SHACL, missing support for e.g. cardinality restrictions. Adding support for these constraints would make our method more useful for real-world knowledge validation tasks. Another angle worth pursuing but which fell out of our current scope is the analysis of our algorithm's time complexity, the theoretical speed up which can be obtained through parallelization, and how it deals with the satisfaction and entailment problems.

ACKNOWLEDGEMENTS

We express our gratitude to Jaap Bakker, coordinating specialist advisor on asset management and data integration at Rijkswaterstaat, for providing us access to the data infrastructure, experts, and facilities needed to complete our research. This research was made possible with the help of Rijkswaterstaat, The Netherlands.

REFERENCES

- Akhtar, W., Cortés-Calabuig, Á., and Paredaens, J. (2010). Constraints in rdf. In *International Workshop on Semantics in Data and Knowledge Bases*, pages 23–39. Springer.
- Anbutamilazhagan, T. and Selvaraj, M. K. (2014). A novel model for mining association rules from semantic web data. *Elysium Journal*, 1(2).
- Barati, M., Bai, Q., and Liu, Q. (2016). *SWARM: An Approach for Mining Semantic Association Rules from Semantic Web Data*, pages 30–43. Springer International Publishing, Cham.
- Bohannon, P., Fan, W., Geerts, F., Jia, X., and Kementsietsidis, A. (2007). Conditional functional dependencies for data cleaning. In *2007 IEEE 23rd international conference on data engineering*, pages 746–755. IEEE.
- Calvanese, D., Fischl, W., Pichler, R., Sallinger, E., and Simkus, M. (2014). Capturing relational schemas and functional dependencies in rdfs. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*.
- Cortés-Calabuig, A. and Paredaens, J. (2012). Semantics of constraints in rdfs. In *AMW*, pages 75–90. Citeseer.
- Fan, W., Hu, C., Liu, X., and Lu, P. (2018). Discovering graph functional dependencies. In *Proceedings of the 2018 International Conference on Management of Data*, pages 427–439. ACM.
- Fan, W. and Lu, P. (2017). Dependencies for graphs. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 403–416. ACM.
- Fan, W., Wu, Y., and Xu, J. (2016). Functional dependencies for graphs. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1843–1857. ACM.
- Fürber, C. (2015). *Data quality management with semantic technologies*. Springer.
- Fürber, C. and Hepp, M. (2011). Towards a vocabulary for data quality management in semantic web architectures. In *Proceedings of the 1st International Workshop on Linked Web Data Management*, pages 1–8.
- Galárraga, L. A., Teflioudi, C., Hose, K., and Suchanek, F. (2013). Amie: association rule mining under incomplete evidence in ontological knowledge bases. In *Proceedings of the 22nd international conference on World Wide Web*, pages 413–422.
- Hamad, F., Liu, I., and Zhang, X. X. (2018). Food discovery with uber eats: Building a query understanding engine. <https://eng.uber.com/uber-eats-query-understanding/>. Accessed: 2020-05-20.
- He, B., Zou, L., and Zhao, D. (2014). Using conditional functional dependency to discover abnormal data in rdf graphs. In *Proceedings of Semantic Web Information Management on Semantic Web Information Management*, pages 1–7. ACM.
- Hellings, J., Gyssens, M., Paredaens, J., and Wu, Y. (2016). Implication and axiomatization of functional and con-

- stant constraints. *Annals of Mathematics and Artificial Intelligence*, 76(3-4):251–279.
- Lausen, G., Meier, M., and Schmidt, M. (2008). Sparqling constraints for rdf. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*, pages 499–509. ACM.
- Meilicke, C., Chekol, M. W., Ruffinelli, D., and Stuckenschmidt, H. (2019). An introduction to anyburl. In *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*, pages 244–248. Springer.
- Ramezani, R., Sarace, M., and Nematbakhsh, M. (2014). Swapriori: a new approach to mining association rules from semantic web data. *Journal of Computing and Security*, 1:16.
- Ristoski, P., De Vries, G. K. D., and Paulheim, H. (2016). A collection of benchmark datasets for systematic evaluations of machine learning on the semantic web. In *International Semantic Web Conference*, pages 186–194. Springer.
- Singhal, A. (2012). Introducing the knowledge graph: Things, not strings. <https://googleblog.blogspot.com/2012/05/introducing-knowledge-graph-things-not.html>. Accessed: 2020-05-20.
- Sun, E. and Iyer, V. (2013). Under the hood: The entities graph. <https://www.facebook.com/notes/facebook-engineering/under-the-hood-the-entities-graph/10151490531588920>. Accessed: 2020-05-20.
- Szekely, P., Knoblock, C. A., Yang, F., Zhu, X., Fink, E. E., Allen, R., and Goodlander, G. (2013). Connecting the smithsonian american art museum to the linked data cloud. In *Extended Semantic Web Conference*, pages 593–607. Springer.
- Tayi, G. K. and Ballou, D. P. (1998). Examining data quality. *Communications of the ACM*, 41(2):54–57.
- Tresp, V., Bundschuh, M., Rettinger, A., and Huang, Y. (2008). Towards machine learning on the semantic web. In *Ursw (Incs vol.)*, pages 282–314. Springer.
- Yu, Y. and Heflin, J. (2011). Extending functional dependency to detect abnormal data in rdf graphs. In *International Semantic Web Conference*, pages 794–809. Springer.