AutoSPARQL: Let Users Query Your Knowledge Base

Jens Lehmann[1] and Lorenz Bühmann[1]

AKSW Group
Department of Computer Science
Johannisgasse 26
04103 Leipzig, Germany
{lehmann,buehmann}@informatik.uni-leipzig.de

Abstract. An advantage of Semantic Web standards like RDF and OWL is their flexibility in modifying the structure of a knowledge base. To turn this flexibility into a practical advantage, it is of high importance to have tools and methods, which offer similar flexibility in exploring information in a knowledge base. This is closely related to the ability to easily formulate queries over those knowledge bases. We explain benefits and drawbacks of existing techniques in achieving this goal and then present the QTL algorithm, which fills a gap in research and practice. It uses supervised machine learning and allows users to ask queries without knowing the schema of the underlying knowledge base beforehand and without expertise in the SPARQL query language. We then present the AutoSPARQL user interface, which implements an active learning approach on top of QTL. Finally, we evaluate the approach based on a benchmark data set for question answering over Linked Data.

1 Introduction and Motivation

The Web of Data has been growing continuously over the past years and now contains more than 100 public SPARQL endpoints with dozens of billion of triples.[1] The data available via those endpoints spans several domains reaching from music, art and science to spatial and encyclopaedic information as can be observed at `http://lod-cloud.net`. Providing this knowledge and improving its quality are important steps towards realising the Semantic Web vision. However, for this vision to become reality, knowledge also needs to be easy to query and use.

Typically, querying an RDF knowledge base via SPARQL queries is not considered an end user task as it requires familiarity with its syntax and the structure of the underlying knowledge base. For this reason, query interfaces are often tight to a specific knowledge base. More flexible techniques include facet based browsing and graphical query builders. We briefly analyse advantages and disadvantages of those approaches and explain how they relate to AutoSPARQL.

Knowledge Base Specific Interfaces: Special purpose interfaces are often convenient to use since they usually shield the user from the complexity and heterogeneity of the underlying knowledge bases. The vast majority of web search forms fall into this category. Such interfaces are often designed to capture the most relevant queries users may ask. However, those queries have to be known in

[1] See `http://ckan.net` for data set statistics.

advance. They usually do not allow to explore the underlying RDF graph structure. Other disadvantages are the development effort required for developing specific interfaces and their inflexibility in case of schema changes or extensions. *Facet-Based Browsing* is a successful technique for exploring knowledge bases, where users are offered useful restrictions (facets) to the resources s/he is viewing. The technique is not knowledge base specific and, thus, requires no or only small adaptations to be used on top of existing SPARQL endpoints. Two examples are the Neofonie Browser `http://dbpedia.neofonie.de`, the Virtuoso facet service `http://dbpedia.org/fct/` and OntoWiki[2] facets. The first is tailored towards DBpedia, whereas the latter two examples can be run on top of arbitrary knowledge bases. A disadvantage of facet-based browsers is that they allow only a limited set of queries. For instance, it is easy to query for objects belonging to a class "Person". However, facets do not work well for more complex queries like "Persons who went to school in Germany", because the restriction "in Germany" refers to the school and not directly a person. Another type of difficult queries is "Persons who live in x", where x is a small city. In this case, the difficulty is that the facet "live in x" may not be offered to the user, because there are many other more frequently occurring patterns offered as facets.

Visual SPARQL Query Builders lower the difficulty of creating SPARQL queries. However, their target user groups are still mostly knowledge engineers and developers. Examples of visual query builders are SPARQL Views[3] [4] and Virtuoso Interactive Query Builder[4]. Even though the queries are visualised, users still need some understanding of how SPARQL queries actually work and which constructs should be used to formulate a query. To visually build a query, users also need a rough understanding of the underlying schema.

Question Answering (QA) Systems allow the user to directly enter his question, e.g. in Ginseng[5], NLP-Reduce[6] or PowerAqua[7]. Usually, they need to be adapted to a particular domain, e.g. via patterns or models. Cross domain question answering without user feedback can be brittle.

AutoSPARQL: In this paper, we propose the QTL algorithm and the AutoSPARQL user interface. It provides an alternative to the above interfaces with a different set of strengths and restrictions. AutoSPARQL uses active supervised machine learning to generate a SPARQL query based on positive examples, i.e. resources which should be in the result set of the SPARQL query, and negative examples, i.e. resources which should not be in the result set of the query. The user can either start with a question as in other QA systems or by directly searching for a relevant resource, e.g. "Berlin". He then selects an appropriate result, which becomes the first positive example. After that, he is asked a series of questions on whether a resource, e.g. "Paris", should also be contained in the result set. These questions are answered by "yes" or "no". This feedback allows the supervised learning method to gradually learn which query

[2] `http://ontowiki.net`

[3] `http://drupal.org/project/sparql_views`

[4] `http://wikis.openlinksw.com/dataspace/owiki/wiki/OATWikiWeb/InteractiveSparqlQueryBuilder`

[5] `http://www.ifi.uzh.ch/ddis/research/talking-to-the-semantic-web/ginseng/`

[6] `http://www.ifi.uzh.ch/ddis/research/talking-to-the-semantic-web/nlpreduce/`

[7] `http://technologies.kmi.open.ac.uk/poweraqua/`

the user is likely interested in. The user can always observe the result of the currently learned query and stop answering questions if the algorithm has correctly learned it. The system can also inform the user if there is no learnable query, which does not contradict with the selection of positive and negative examples. AutoSPARQL can generate more complex queries than facet based browsers and most knowledge base specific applications, while there are some restrictions – explained in detail later – compared to manually or visually creating queries. We argue that it is easier to use than manual or visual SPARQL query builders and not much more difficult to use than facet-based browsers or standard QA systems. Due to this different sets of strengths and weaknesses, it provides a viable alternative to the methods described above. Our claim is that AutoSPARQL is the first user interface, which allows end users to create and refine non-trivial SPARQL queries over arbitrary knowledge bases.

Overall, we make the following contributions:

– introduction of a new active learning method for creating SPARQL queries
– the Query Tree Learner (QTL) algorithm
– the AutoSPARQL interface at `http://autosparql.dl-learner.org`
– an evaluation on a benchmark data set for question answering over Linked Data

Sections 2 to 4 are the formal part of the paper. In Section 2, we introduce the concept of query trees as underlying structure. After that, we show basic operations on those trees and proof their properties in Section 3. The following section explains the QTL algorithm, which combines results from the previous sections. In Section 5, the AutoSPARQL workflow and user interface are presented. In Section 6, we measure how well AutoSPARQL works on a benchmark data set for question answering over Linked Data. Related work is described in Section 7. Finally, some high level key aspects of our approach are discussed in Section 8.

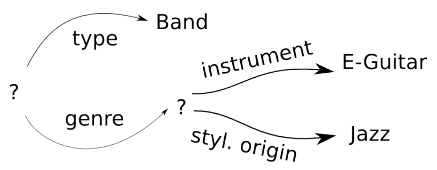
2 Query Trees

Before explaining query trees, we fix some preliminaries. We will often use standard notions from the RDF and SPARQL specifications[8], e.g. triple, RDF graph, triple pattern, basic graph pattern. We denote the set of RDF resources with R, the set of RDF literals with L, the set of SPARQL queries with SQ, and the set of strings with S. We use $f_{|D}$ to denote the restriction of a function to a domain D.

We call the structure, which is used internally by the QTL algorithm, a *query tree.* A query tree roughly corresponds to a SPARQL query, but not all SPARQL queries can be expressed as query trees.

Definition 1 (Query Tree). *A* query tree *is a rooted, directed, labelled tree* $T = (V, E, \ell)$*, where* V *is a finite set of* nodes*,* $E \subset V \times R \times V$ *is a finite set of* edges*,* $NL = L \cup R \cup \{?\}$ *is a set of* node labels *and* $\ell : V \to NL$ *is the* labelling function*. The* root *of* T *is denoted as* $root(T)$*. If* $\ell(root(T)) = ?$*, we call the query tree* complete*. The set of all query trees is denoted by* $\mathcal{T}$ *and* $\mathcal{T}_C$ *for complete query trees. We use the notions* $V(T) := V$*,* $E(T) := E$*,* $\ell(T) := \ell$ *to refer to nodes, edges and label function of a tree* T*. We say* $v_1 \xrightarrow{e_1} \cdots \xrightarrow{e_n} v_{n+1}$ *is a* path *of length* n *from* v_1 *to* v_{n+1} *in* T *iff* $(v_i, e_i, v_{i+1}) \in E$ *for* $1 \leq i \leq n$*. The depth of a tree is length of its longest path.*

[8] `http://www.w3.org/TR/rdf-concepts`, `http://www.w3.org/TR/rdf-sparql-query`



```
SELECT ?x0 WHERE {
?x0 rdf:type dbo:Band.
?x0 dbo:genre ?x1.
?x1 dbo:instrument dbp:Electric_guitar.
?x1 dbo:stylisticOrigin dbp:Jazz.
}
```

Fig. 1. Query tree (left) and corresponding SPARQL query (right).

Definition 2 (Subtrees as Query Trees). *If $T = (V, E, \ell)$ is a query tree and $v \in V$, then we denote by $T(v)$ the tree $T' = (V', E', \ell')$ with root$(T') = v$, $V' = \{v' \mid$ there is a path from v to $v'\}$, $E' = E \cap V' \times R \times V'$ and $\ell' = \ell_{|V'}$.*

Definition 3 (Node Label Replacement). *Given a query tree T, a node $v \in V(T)$ and $n \in NL$, we define $\ell[v \mapsto n]$ as $\ell[v \mapsto n](v) := n$ and $l[v \mapsto L](w) := \ell(w)$ for all $w \neq v$. We define $T[v \mapsto n] := (V(T), E(T), \ell(T)[v \mapsto n])$ for $v \in V(T)$. We say that the* label of v is replaced by n.

2.1 Mapping Query Trees to SPARQL Queries

Each query tree can be transformed to a SPARQL query. The result of the query always has a single column. This column is created via the label of the root node, which we will also refer to as the *projection variable* in this article. Please note that while QTL itself learns queries with a single column, i.e. lists of resources, those can be extended via the AutoSPARQL user interface. Each edge in the query tree corresponds to a SPARQL triple pattern. Starting from the root node, the tree is traversed as long as we encounter variable symbols. Each variable symbol is represented by a new variable in the SPARQL query. An example is shown in Figure 1.

Formally, the mapping is defined as follows: Each node with label ? is assigned a unique number via the function id : $V \mapsto \mathbb{N}$. A root node is assigned value 0. The function mapnode : $V \mapsto S$ is defined as: $mapnode(n) =$ "$?x$" $+$ id(n) if $\ell(n) =?$ and $mapnode(n) = n$ otherwise. Note that "+" denotes string concatenation. Based on this, the function mapedge : $E \mapsto S$ is defined as $map(v) +$ " " $+ e +$ " " $+ map(v') +$ ".". Finally, the function sparql : $T_C \mapsto SQ$ is defined as shown in Function 1 by tree traversal starting from the root of T and stopping when a non-variable node has been reached.

2.2 Mapping Resources to Query Trees

Each resource in an RDF graph can be mapped to a query tree. Intuitively, the tree corresponds to the neighbourhood of the resource in the graph. In order to map a resource to a tree, we have to limit ourselves to a recursion depth for reasons of efficiency. This recursion depth corresponds to the maximum nesting of triple patterns, which can be learned by the QTL algorithm, which we will detail in Section 4. Another way to view a query tree for a resource is that it defines a very specific query, which contains the resource itself as result (if it

```
query = “SELECT ?x0 { ?x0 ?y ?z . ”;
nodequeue = [root(T)];
while nodequeue not empty do
    v = poll(nodequeue) // pick and remove first node ;
    foreach edge (v, e, v′) in E(T) do
        query += mapedge((v, e, v′)) ;
        if ℓ(v′) =? then add v′ at the end of nodequeue
query += “}”;
return query
```

Function `sparql(`T`)`

occurs at least once in the subject position of a triple in the knowledge base). The formal definition of the query tree mapping is as follows:

Definition 4 (Resource to Query Tree Mapping). *A resource r in an RDF graph $G = (V, E, \ell)$ is mapped to a tree $T' = (V', E', \ell')$ with respect to a recursion depth $d \in \mathbb{N}$ as follows: $V' = \{v_p \mid v_p \in V,$ there is a path p of length $l \leq d$ from r to v in $G\}$, $E' = V' \times R \times V' \cap E$, $\ell' = \ell_{|V'}$. The result of the function $map : R \times G \times \mathbb{N} \rightarrow \mathcal{T_C}$ is then defined as $T := T'[root(T') \mapsto ?]$.*

Query trees act as a bridge between the description of a resource in an RDF graph and SPARQL queries containing the resource in their result set. Using them enables us to define a very efficient learning algorithm for SPARQL queries. Note that a query tree T does not contain cycles, whereas an RDF graph G can, of course, contain cycles. Also note that query trees intentionally only support a limited subset of SPARQL.

3 Operations on Query Trees

In this section, we define operations on query trees, which are the basis of the QTL algorithm. We define a subsumption ordering over trees, which allows to apply techniques from the area of Inductive Logic Programming [13]. Specifically, we adapt least general generalisation and negative based reduction [2].

3.1 Query Tree Subsumption

In the following, we define query tree subsumption. Intuitively, if a query tree T_1 is subsumed by T_2, then the SPARQL query corresponding to T_1 returns fewer results than the SPARQL query corresponding to T_2. The definition of query tree subsumption will be done in terms of the SPARQL algebra. Similar as in [1, 14], we use the notion $[[q]]_G$ as the evaluation of a SPARQL query q in an RDF graph G.

Definition 5 (Query Tree Subsumption). *Let T_1 and T_2 be complete query trees. T_1 is subsumed by T_2, denoted as $T_1 \preceq T_2$, if we have $[[sparql(T_1)]]_G(?x0) \subseteq [[sparql(T_2)]]_G(?x0)$ for any RDF graph G.*

Definition 6 ($\leq$ relation). *For query trees T_1 and T_2, we have $T_1 \leq T_2$ iff the following holds:*

1. *if $\ell(root(T_2)) \neq ?$, then $\ell(root(T_1)) = \ell(root(T_2))$*
2. *for each edge $(root(T_2), p, v_2)$ in T_2 there exists an edge $(root(T_1), p, v_1)$ in T_1 such that:*
   (a) if $\ell(v_2) \neq ?$, then $\ell(v_1) = \ell(v_2)$
   (b) if $\ell(v_2) = ?$, then $T(v_1) \leq T(v_2)$ (see Definition 2)

We define $T_1 \simeq T_2$ as $T_1 \leq T_2$ and $T_2 \leq T_1$. $T_1 < T_2$ is defined as $T_1 \leq T_2$ and $T_1 \not\simeq T_2$.

The following is a consequence of the definition of $\leq$. It connects the structure of query trees with the semantics of SPARQL.

Proposition 1. *Let T_1 and T_2 be complete query trees. $T_1 \leq T_2$ implies $T_1 \preceq T_2$.*

Proof. We prove the proposition by induction over the depth of T_2. Let G be an RDF graph.

Induction Base ($depth(T_2) = 0$): In this case, $[[sparql(T_2)]]_G(?x0)$ is the set of all resources occurring in subjects of triples in G and, therefore, $T_1 \preceq T_2$.

Induction Step ($depth(T_2) > 0$): $sparql(T_1)$ and $sparql(T_2)$ have a basic graph pattern in their WHERE clause, i.e. a s set of triple patterns. Due to the definition of *sparql*, the triple patterns with subject $?x0$ have one of the following forms: 1.) $?x0\ ?y\ ?z$ 2.) $?x0\ p\ m$ with $m \in R \cup L$ 3.) $?x0\ p\ ?xi$. Each such pattern in a SPARQL query is a restriction on $?x0$. To prove the proposition, we show that for each such triple pattern in $sparql(T_2)$, there is a triple pattern in $sparql(T_1)$, which is a stronger restriction of $?x0$, i.e. leads to fewer results for $?x0$. We do this by case distinction:

1. $?x0\ ?y\ ?z$: The same pattern exists in $sparql(T_1)$.
2. $?x0\ p\ m$ with $m \in R \cup L$: Thus, there is an edge $(root(T_2), p, v_2)$ with $\ell(v_2) = m$ in T_2. Because of the definition of $\leq$, there is an edge $(root(T_1), p, v_1)$ with $\ell(v_1) = m$ in T_1, which leads to the same triple pattern in $sparql(T_1)$.
3. $?x0\ p\ ?xi$: This means that there is an edge $(root(T_2), p, v_2)$ with $\ell(v_2) =?$ in T_2. Let $(root(T_1), p, v_1)$ with $\ell(v_1) = s$ be a corresponding edge in T_1 according to the definition of $\leq$. We distinguish two cases: a) $s \neq ?$. In this case, $sparql(T_1)$ contains the pattern $?x0\ p\ s$, which is a stronger restriction on $?x0$ than $?x0\ p\ ?xi$. b) $s = ?$. In this case, $sparql(T_1)$ contains the pattern $?x0\ p\ ?xj$. By induction, we know $T(v_1) \leq T(v_2)$ and, consequently, $[[sparql(T(v_1))]]_G(?x0) \subseteq [[sparql(T(v_2))]]_G(?x0)$, i.e. the pattern is a stronger restriction on $?x0$, because there are fewer or equally many matches for $?xj$ than for $?xi$. □

The proposition means that whenever $T_1 \leq T_2$, the result of the SPARQL query corresponding to T_1 does not return additional results compared to the SPARQL query corresponding to T_2. Note that the inverse of the proposition does not hold: If a query q_1 returns fewer results than a query q_2, this does not mean that $T_1 \leq T_2$ for the corresponding query trees, because q_1 and q_2 can be structurally completely different queries.

3.2 Least General Generalisation

The least general generalisation (lgg) operation takes two query trees as input and returns the most specific query tree, which subsumes both input trees. We first define the operation *lgg* algorithmically and then proof its properties.

```
1  init $T = (V, E, \ell)$ with $V = \{v\}$, $E = \emptyset$, $\ell(v) = ?$;
2  if $\ell(v_1) = \ell(v_2)$ then $\ell(v) = \ell(v_1)$;
3  foreach $p$ in $\{p' \mid \exists v_1'.(v_1, p', v_1') \in E(T_1)$ and $\exists v_2'.(v_2, p', v_2') \in E(T_2)\}$ do
4      foreach $v_1'$ with $(v_1, p, v_1') \in E(T_1)$ do
5          foreach $v_2'$ with $(v_2, p, v_2') \in E(T_2)$ do
6              $v' = root(\text{lgg}(T(v_1'), T(v_2')))$; add = true;
7              foreach $v_{prev}$ with $(v, p, v_{prev}) \in E(T)$ do
8                  if $add = true$ then
9                      if $T(v_{prev}) \leq T(v')$ then add = false;
10                     if $T(v') < T(v_{prev})$ then remove edge $(v, p, v_{prev})$ from $T$;
11             if $add = true$ then add edge $(v, p, v')$ to $T$;
12 return $T$
```

Function `lgg(`T_1`,` T_2`)`

Function 2 defines the algorithm to compute least general generalisations of query trees. It takes two query trees T_1 and T_2 as input and returns T as their lgg. T is initialised as empty tree in Line 1. The next line compares the labels of the root nodes of T_1 and T_2. If they are equal, then this label is preserved in the generalisation, otherwise ? is used as label. Line 3 groups outgoing edges in the root nodes of T_1 and T_2 by their property label – only if a property is used in both trees, it will be part of the lgg. Line 4 and 5 are used for comparing pairs of edges in T_1 and T_2. For each combination, the lgg is recursively computed. However, in order to keep the resulting tree small, only edges which do not subsume another edge are preserved (Lines 7 to 10). Finally, Line 11 adds the computed lgg to the tree T, which is returned. *lgg* is commutative. We use $lgg(\{T_1, \dots, T_n\})$ as shortcut notation for $lgg(T_1, lgg(T_2, \dots, T_n) \dots)$.

Proposition 2. *Let lgg be defined as in Function 2, T_1 and T_2 be trees and $T = lgg(T_1, T_2)$. Then the following results hold:*

1. *$T_1 \leq T$ and $T_2 \leq T$ (i.e. lgg generalises)*
2. *for any tree T', we have $T_1 \leq T'$, $T_2 \leq T'$ implies $T \leq T'$ (i.e. lgg is least)*

Proof. The proofs are as follows:

1.) We prove by induction over the depth of T. Without loss of generality, we show $T_1 \leq T$.

Induction Base ($depth(T) = 0$): If $\ell(root(T)) \neq ?$, then by Function 2 $\ell(root(T_1)) = \ell(root(T))$ (see also table below).

Induction Step ($depth(T) > 0$): We have to show that for an edge $e = (root(T), p, v) \in E(T)$ the conditions in Definition 6 hold. Due to the definition of Function 2, $e = (root(T), p, root(\text{lgg}(T(v_1), T(v_2))))$ was created from two edges $(\mathit{root}(T_1), p, v_1) \in E(T_1)$ and $(\mathit{root}(T_2), p, v_2) \in E(T_2)$. If $\ell(v) \neq ?$ (Definition 6, condition 1), then $\ell(v_1) = \ell(v)$ by Line 2 of Function 2. If $\ell(v) = ?$ (condition 2), then $T(v_1) \leq T(v)$ follows by induction.

2.) We use induction over the depth of T.

Induction Base ($\mathit{depth}(T) = 0$): We first show $\mathit{depth}(T') = 0$. By contradiction, assume that T' has at least one edge. Let p be the label of an outgoing edge from the root of T'. By Definition 6, both T_1 and T_2 must therefore also have outgoing edges from their respective root nodes with label p. Consequently, $T = \mathit{lgg}(T_1, T_2)$ has an outgoing edge from its root by Function 2 (Lines 4-11 create at least one such edge). This contradicts $\mathit{depth}(T) = 0$.

We make a complete case distinction on root node labels of T_1 and T_2 (note that $m \neq ?$, $n \neq ?$):

$\ell(\mathit{root}(T_1))$	$\ell(\mathit{root}(T_2))$	$\ell(\mathit{root}(T))$ according to Function 2
m	m	m
m	$n(\neq m)$	?
m	?	?
?	m	?
?	?	?

In Row 1, $\ell(\mathit{root}(T'))$ is either m or ?, but in any case $T \leq T'$. For Rows 2-5, $\ell(\mathit{root}(T')) = ?$, because otherwise $T_1 \not\leq T'$ or $T_2 \not\leq T'$. Again, we have $T \leq T'$.

Induction Step ($\mathit{depth}(T) > 0$): Again, we show $T \leq T'$ using Definition 6:

Condition 1: $? \neq \ell(\mathit{root}(T')) = \ell(\mathit{root}(T_1)) = \ell(\mathit{root}(T_2))$ (from $T_1 \leq T'$ and $T_2 \leq T'$) $= \ell(\mathit{root}(T))$ (from Function 2)

Condition 2: Let $(\mathit{root}(T'), p, v')$ be an edge in T'. Due to $T_1 \leq T'$ and $T_2 \leq T'$, there is an edge $(\mathit{root}(T_1), p, v_1)$ in T_1 and an edge $(\mathit{root}(T_2), p, v_2)$ in T_2.

2a): $? \neq \ell(v') = \ell(v_1)) = \ell(v_2)$ (from $T_1 \leq T'$ and $T_2 \leq T'$) $= \ell(v)$ (from Function 2)

2b): Due to $\ell(v') = ?$, we get $T(v_1) \leq T(v')$ and $T(v_2) \leq T(v')$. Hence, we can deduce $T(v) = \mathit{lgg}(T(v_1), T(v_2)) \leq T(v')$ by induction. □

3.3 Negative Based Reduction

Negative based reduction is used to generalise a given tree T using trees $T_1, \ldots, T_n$ as input. For each tree, we assume $T_i \not\leq T$ ($1 \leq i \leq n$). The idea is to generalise T by removing edges or changing node labels without *overgeneralising*. Overgeneralising means that $T_i \leq T$ for some i ($1 \leq i \leq n$). Negative based reduction has already been used in ILP and is a relatively simple procedure in most cases. In QTL, it is more involved, which is why we only sketch it here due to lack of space. The basic idea is that upward refinement operations are used on the given query tree T until a negative example is covered. When this happens, QTL queries for results of the SPARQL query corresponding to the refined tree. If there are no new resources compared to the *lgg*, then a different upward refinement path is used. If there are new resources, then a binary search procedure is used to find the most specific tree on the upward refinement path, which still delivers new resources. This tree is then returned by QTL.

4 QTL Algorithm

The Query Tree Learner (QTL) integrates the formal foundations from Sections 2 and 3 into a light-weight learning algorithm. QTL is a supervised algorithm, i.e. it uses positive and negative examples as input. In this case, an example is an RDF resource. In a first step, all examples are mapped to query trees as shown in Algorithm 3. The mapping, specified in Definition 4, requires a recursion depth as input. The recursion depth has influence on the size of the generated tree. It is the maximum depth of the generated query tree and, therefore, also the maximum depth of the learned SPARQL query. The mapping method also requires a method to obtain information about a resource from G. In our implementation, this is done via SPARQL queries with the option to use a cache in order to minimise the load on the endpoint. A properly initialised cache also ensures roughly constant response times for the user.

```
input : RDF graph $G$, recursion depth $d$, pos. examples
        $E^+ = \{r_1, \dots, r_m\} \subset R$, $E^+ \neq \emptyset$, neg. examples $E^- = \{s_1, \dots, s_n\} \subset R$
output: SPARQL Query $q$
1 $T^+ = \{T_i^+ \mid \exists i.r_i \in E^+, T_i^+ = \text{map}(G, d, r_i)\}$; $T^-$ analogously;
2 $T = T_1^+$; for $i \leftarrow 2$ to $m$ do $T = \text{lgg}(T, T_i^+)$;
3 if there exists a $T_i^-$ with $T_i^- \leq T$ then print "no learnable query exists" ;
4 if $T^- = \emptyset$ then $T' = \text{pg}(T)$ else $T' = \text{nbr}(T, T^-)$;
5 $q = \text{sparql}(T')$
```

Algorithm 3: QTL Algorithm.

The next step in QTL is to compute the lgg of all positive examples (Line 2). If this results in a tree, which subsumes negative examples, then no query fitting the positive and negative examples can be learned. This is a consequence of Proposition 2. Usually, this happens when the RDF graph does not contain necessary features to construct a query. For instance, a user may want to get all cities in France, but the endpoint does not contain properties or classes to infer that some city is located in a particular country.

In Line 4 of the algorithm, negative based reduction (nbr) is used to generalise the lgg. A potentially large tree containing everything the positive examples have in common, is generalised by using negative examples as explained in Section 3. In case there are no negative examples available yet, a different operation, *positive generalisation* (pg) is used. Positive generalisations uses the nbr function, but calls it with a seed of resources which is disjoint with the positive examples. This allows to use QTL as positive only algorithm. Finally, in Line 5 of Algorithm 3, the query tree is converted into a SPARQL query. Some characteristics of QTL in combination with the AutoSPARQL interface are discussed in Section 8.

5 AutoSPARQL User Interface

AutoSPARQL is available at `http://autosparql.dl-learner.org`. It is a rich internet application based on the Google Web Toolkit. AutoSPARQL and the QTL algorithm are part of DL-Learner [9]. Their source code is available in the DL-Learner SVN repository.

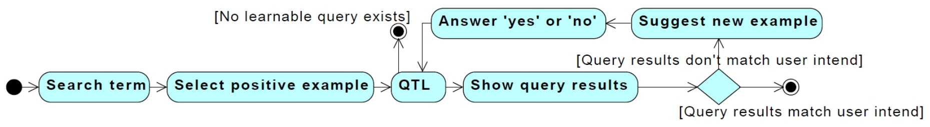

[No learnable query exists]
Answer 'yes' or 'no'
Suggest new example
[Query results don't match user intend]
Search term
Select positive example
QTL
Show query results
[Query results match user intend]

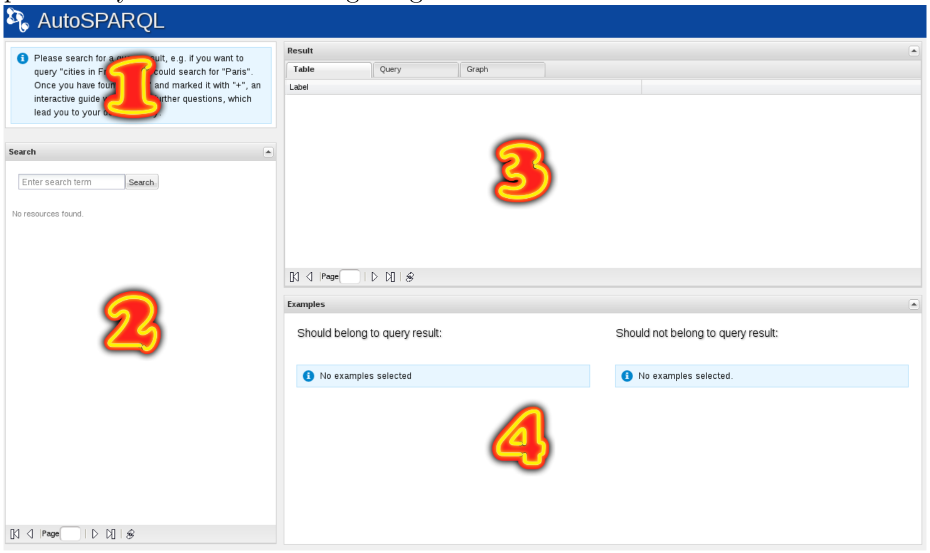
Fig. 2. AutoSPARQL Workflow

The tool implements an active learning method as shown in Figure 2. In a first step, the user performs a query and selects at least one of the search results as positive example, i.e. it should be returned as result of the query he constructs. From this, an initial query is suggested by QTL and the user is asked the question whether a certain resource should be included in the result set.

After each question is answered, QTL is invoked again. This process is the active learning part of AutoSPARQL. It is iterated until the desired query is found or no learnable query, matching the examples, exists. The user interface allows several other options such as changing previous decisions, deletion of examples or the selection of several positive examples in one through a tabular interface. The following result shows that AutoSPARQL always returns a correct query or replies that no learnable query exists after a finite number of iterations. The proof, which mainly uses the properties of the lgg function, is omitted, because of lack of space.

Proposition 3. *Let $A = \{r_1, \ldots, r_n\}$ be a target set of resources in an RDF graph G, $d \in \mathbb{N}$ and assume that a user/oracle answers questions by AutoSPARQL correctly. If there exists a query tree T with depth $\leq d$ such that $[[sparql(T)]]_G(?x0) = A$, then AutoSPARQL learns a tree T' with $[[sparql(T')]]_G(?x0) = A$, else it reports that no such tree exists.*

In order to improve the efficiency, AutoSPARQL can optionally use the natural language query of the user to filter the query trees. If neither the property nor the label of the target node of an edge in a query tree has a sufficiently high string similarity to a phrase or a WordNet-synonym of a phrase in the natural language query, then it is discarded. Four different string metrics are combined to reduce the probability of filtering relevant edges. If this filter in AutoSPARQL is enabled, the completeness result above no longer holds, because there is non-zero probability that a relevant edge might be filtered.



Please search for a [illegible]ult, e.g. if you want to query "cities in Fr[illegible] could search for "Paris". Once you have foun[illegible] and marked it with "+", an interactive guide [illegible]urther questions, which lead you to your [illegible].

No resources found.

Should belong to query result:

No examples selected

Should not belong to query result:

No examples selected.

Fig. 3. Screenshot of initial AutoSPARQL user interface: It consists of four areas (1) question panel (2) search panel (3) query result panel and (4) example overview panel.

After QTL has been invoked through a question-answer session, AutoSPARQL allows to further fine-tune the query. For instance, users can select which properties to display, ordering by a property and language settings. As an example, a typical AutoSPARQL session for learning the following query could look as follows. Note that the resources are displayed via a knowledge base specific template.

```
PREFIX dbpedia:  <http://dbpedia.org/resource/>
PREFIX dbo:  <http://dbpedia.org/ontology/>
SELECT ?band ?label ?homepage ?genre WHERE {
?band a dbo:Band .
?band rdfs:label ?label .
OPTIONAL { ?band foaf:homepage ?homepage } .
?band dbo:genre ?genre .
?genre dbo:instrument dbpedia:Electric_guitar .
?genre dbo:stylisticOrigin dbpedia:Jazz .
}
ORDER BY ?label LIMIT 100
```

– search for “bands with a genre which mixes electric guitars and Jazz”
– resource: `dbpedia:Foals` answer: YES
– resource: `dbpedia:Hot_Chip` answer: NO
– resource: `dbpedia:Metalwood` answer: YES
– resource: `dbpedia:Polvo` answer: YES
– resource: `dbpedia:Ozric_Tentacles` answer: YES
– resource: `dbpedia:New_Young_Pony_Club` answer: NO
– select “genre” as property to return and “homepage” as additional property
– click on “label” column head to order by it and adjust limit

After that, the query can be saved and a URL is provided for the user to call it. Results will be cached with a configurable timeout on the AutoSPARQL server, such that users can efficiently embed it in websites. In particular, in combination with DBpedia Live, this allows users to include up-to-date information in homepages, blogs or forums.

6 Evaluation

We used the benchmark data set of the 1st Workshop on Question Answering over Linked Data (QALD)[9], which defines 50 questions to DBpedia and their answers. From those queries, we filtered those, which return at least 3 resources. This excludes questions asking directly for facts instead of lists. Furthermore, we filtered queries, which are not in the target language of AutoSPARQL, e.g. contain UNION constructs. The resulting evaluation set contains 15 natural language queries. Since answers for all questions were given, we used them as oracle, which answers ”YES“ when a resource is in the result set and ”NO“ otherwise. We seeded the positive examples by using the search function of Wikipedia. At most 3 positive examples from the top 20 search results were selected. If less than 3 positive examples are in the top 20, then we picked positive examples from the answer set. In any case, the active learning approach starts with 3 positive and 1 negative examples. The NLP filter, described in the previous section, was switched on since DBpedia has a very large and diverse schema.

[9] `http://www.sc.cit-ec.uni-bielefeld.de/qald-1`

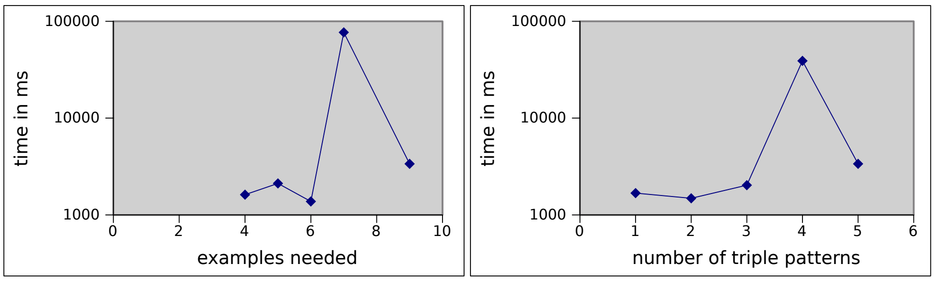


Fig. 4. Statistics showing that roughly 1 s per example are needed (left) and roughly constant time independent of the number of triple patterns (right). The peak in both images is caused by only one single question. All other questions needed less than 10 s to be learned correctly.

The hardware, we used, was a 6-core machine with 2 GB RAM allocated to the VM. We used a recursion depth of 2 for our experiments. We asked questions against a mirror of `http://dbpedia.org/sparql` containing DBpedia 3.5.1. Due to Proposition 3, AutoSPARQL always learns a correct query via this procedure. We were interested in two questions: 1. How many examples are required to learn those queries? 2. Is the performance of AutoSPARQL sufficient?

Regarding the number of examples, we found that 4 (in this case the lgg was already the solution) to 9 were needed and 5 on average. We consider this number to be very low and believe that this is due to the combination of active learning and lggs, which are both known not to require many examples. Most of the examples were positives, which indicates that the questions by AutoSPARQL are mostly close to the intuition of the user, i.e. he is not required to look at a high number of seemingly unrelated RDF resources.

Regarding performance, we discovered that AutoSPARQL requires 7 seconds on average to learn a query with a maximum of 77 seconds. From this, < 1 % of the time are required to calculate the lgg, 73 % to calculate the nbr and 26 % for SPARQL queries to a remote endpoint.

Overall, we consider the performance of AutoSPARQL to be good and a lot of engineering effort was spend to achieve this. The low computational effort required allows to keep response times for users at a minimum and learn several queries in parallel over several endpoints on average hardware.

Apart from the total numbers, we looked at some aspects in more detail. First, we analysed the relation between the number of examples needed and the total time required by AutoSPARQL. Figure 4 shows that roughly the same time per example is needed independent of the total number of examples. This means that the response time for the user after each question remains roughly constant. We also analysed whether there is a relation between the complexity of a query, which we measure in number of triple patterns here, and the time required to learn it. Figure 4 shows that there is no such correlation, i.e. more complex queries are not harder to learn than simple ones. This is common for lgg based approaches. The peak in both diagrams is based on one single question, which was the only question where 7 examples were needed and 1 of 2 questions with 4 triple patterns in the learned query. We discovered that in this case the query tree after the lgg was still very large, so the nbr needed more time than in the other questions.

7 Related Work

In Section 1, we already compared the AutoSPARQL user interface to other techniques like facet-based browsing, visual query builders and interfaces adapted to a specific knowledge base. In this section, we focus on the technical aspects of our solution. AutoSPARQL was mainly inspired by two main research areas: Inductive Logic Programming (ILP) and Active Learning.

The target of ILP [13] is to learn a hypothesis from examples and background knowledge. It was most widely applied for learning horn clauses, but also in the Semantic Web context based on OWL and description logics [6, 11, 10, 7, 5] with predecessors in the early 90s [8]. Those approaches use various techniques like inverse resolution, inverse entailment and commonly refinement operators. Least general generalisation, as we used here, is one of those techniques. It has favourable properties in the context of AutoSPARQL, because it is very suitable for learning from a low number of examples. This is mainly due to the fact, that lgg allows to make large leaps through the search space in contrast to gradual refinement. More generally, generate-and-test procedures are often less efficient then test-incorporation as pointed out in an article about the ProGolem system [12], which has influenced the design of our system. ProGolem, which is based on horn logics, also employs negative based reduction, although in a simpler form than in QTL. Drawbacks of lggs usually arise when expressive target languages are used and the input data is very noisy. The latter is usually not a problem in AutoSPARQL, because examples are manually confirmed and can be revised during the learning process. As for the expressiveness of the target language, we carefully selected a fragment of SPARQL, where lggs exist and can be efficiently computed.

Active learning (survey in [15]) aims to achieve high accuracy with few training examples by deciding which data is used for learning. As in AutoSPARQL, this is usually done by asking a human questions, e.g. to classify an example as positive or negative. Active learning has been combined with ILP in [2] to discover gene functions. In our context, an advantage of active learning is that it reduces the amount of background knowledge required for learning a SPARQL query by only considering the RDF neighbourhood of few resources. This way, the burden on SPARQL endpoints is kept as low as possible and the memory requirements for AutoSPARQL are small, which allows to serve many users in parallel. In addition, we use a cache solution, not described here for brevity, to reduce network traffic and allow predictable execution times.

Also related are natural language query interfaces like Google Squared[10], which is easy to use, but less accurate and controllable than AutoSPARQL. In [3], intensional answers have been learned by applying lggs on answers retrieved via the ORAKEL natural language interface. In contrast to our approach, this is done via clausal logic. An integration of natural language interfaces and AutoSPARQL is an interesting target for future work.

[10] `http://www.google.com/squared`

8 Discussion and Conclusions

In the final section, we discuss some key aspects of AutoSPARQL/QTL and give concluding remarks.

Efficiency: As demonstrated, one of the key benefits of our approach is its efficiency. This was possible by focusing on a subset of SPARQL and using query trees as a lightweight data structure acting as bridge between the structure of the background RDF graph and SPARQL queries.

Expressiveness: AutoSPARQL supports a subset of SPARQL, which we deem relevant to cover relevant for typical queries by users. However, it certainly does not render SPARQL experts unnecessary, because e.g. when developing Semantic Web applications, more complex queries are needed. Some constructs in SPARQL, e.g. UNION, were avoided, because they would significantly increase the search space and render the approach less efficient. Some extensions of the current expressiveness are already planned and preliminary algorithms drafted, e.g. for learning graph patterns with the same variable occurring more than once in the object of triple patterns and support for different FILTERs.

Low number of questions: Because of Proposition 3, AutoSPARQL is guaranteed to terminate and correctly learn a query tree if it exists. The evaluation shows that a low number of questions is needed to learn typical queries. In the future, we will integrate a natural language interface in AutoSPARQL, such that the first search by the user (see workflow in Figure 2) returns more positive examples, which further simplifies query creation.

Noise: AutoSPARQL/QTL do not support handling noisy data, i.e. it is assumed that the answers to the questions posed by AutoSPARQL are correct. While it is extensible in this direction, we currently pursue the approach of notifying a user when there is a conflict in his choice of positive and negative examples. This can then be corrected by the user. Given the low number of examples in our active learning strategy, this appears to be feasible. However, we envision adding noise handling to QTL for other usage scenarios which do not have these favourable characteristics.

Reasoning: Since AutoSPARQL uses triple stores, it depends on the inferences capabilities (if any) of those stores. It is noteworthy that the SPARQL 1.1 working draft contains various entailment regimes[11]. The standardisation of inference in SPARQL is likely to increase support for it in triple stores and, therefore, allow more powerful queries in general and for AutoSPARQL in particular.

Overall: We introduced the QTL algorithm, which is the first algorithm to induce SPARQL queries to the best of our knowledge. The AutoSPARQL interface provides an active learning environment on top of QTL. As we argued in the introduction, the key impact of AutoSPARQL is to provide a new alternative for querying knowledge bases, which is complementary to existing techniques like facet based browsing or visual query builders. We believe that AutoSPARQL is

[11] `http://www.w3.org/TR/sparql11-entailment/`

one of the first interfaces to flexibly let non-experts ask and refine non-trivial queries against an RDF knowledge base.

References

1. Renzo Angles and Claudio Gutierrez. The expressive power of SPARQL. In Amit P. Sheth, Steffen Staab, Mike Dean, Massimo Paolucci, Diana Maynard, Timothy W. Finin, and Krishnaprasad Thirunarayan, editors, *International Semantic Web Conference*, volume 5318 of *LNCS*, pages 114–129. Springer, 2008.
2. Christopher H. Bryant, Stephen Muggleton, Stephen G. Oliver, Douglas B. Kell, Philip G. K. Reiser, and Ross D. King. Combining inductive logic programming, active learning and robotics to discover the function of genes. *Electron. Trans. Artif. Intell.*, 5(B):1–36, 2001.
3. Philipp Cimiano, Sebastian Rudolph, and Helena Hartfiel. Computing intensional answers to questions - an inductive logic programming approach. *Data Knowl. Eng.*, 69(3):261–278, 2010.
4. Lin Clark. Sparql views: A visual sparql query builder for drupal. In *9th International Semantic Web Conference (ISWC2010)*, November 2010.
5. C. M. Cumby and D. Roth. Learning with feature description logics. In S. Matwin and C. Sammut, editors, *Proceedings of the 12th International Conference on Inductive Logic Programming*, volume 2583 of *LNAI*, pages 32–47. Springer-Verlag, 2003.
6. Nicola Fanizzi, Claudia d'Amato, and Floriana Esposito. DL-FOIL concept learning in description logics. In *Proceedings of the 18th International Conference on Inductive Logic Programming*, volume 5194 of *LNCS*, pages 107–121. Springer, 2008.
7. Luigi Iannone, Ignazio Palmisano, and Nicola Fanizzi. An algorithm based on counterfactuals for concept learning in the semantic web. *Applied Intelligence*, 26(2):139–159, 2007.
8. Jörg-Uwe Kietz and Katharina Morik. A polynomial approach to the constructive induction of structural knowledge. *Machine Learning*, 14:193–217, 1994.
9. Jens Lehmann. DL-Learner: learning concepts in description logics. *Journal of Machine Learning Research (JMLR)*, 10:2639–2642, 2009.
10. Jens Lehmann and Christoph Haase. Ideal downward refinement in the EL description logic. In *Inductive Logic Programming, 19th International Conference, ILP 2009, Leuven, Belgium*, 2009.
11. Jens Lehmann and Pascal Hitzler. Concept learning in description logics using refinement operators. *Machine Learning journal*, 78(1-2):203–250, 2010.
12. Stephen Muggleton, José Carlos Almeida Santos, and Alireza Tamaddoni-Nezhad. Progolem: A system based on relative minimal generalisation. In Luc De Raedt, editor, *ILP*, volume 5989 of *LNCS*, pages 131–148. Springer, 2009.
13. Shan-Hwei Nienhuys-Cheng and Ronald de Wolf, editors. *Foundations of Inductive Logic Programming*, volume 1228 of *LNCS*. Springer, 1997.
14. Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. In Isabel F. Cruz, Stefan Decker, Dean Allemang, Chris Preist, Daniel Schwabe, Peter Mika, Michael Uschold, and Lora Aroyo, editors, *International Semantic Web Conference*, volume 4273 of *LNCS*, pages 30–43. Springer, 2006.
15. Burr Settles. Active learning literature survey. Computer Sciences Technical Report 1648, University of Wisconsin–Madison, 2009.