An Evolutionary Algorithm to Learn SPARQL Queries for Source-Target-Pairs

Finding Patterns for Human Associations in DBpedia

Jörn Hees, Rouven Bauer, Joachim Folz, Damian Borth, and Andreas Dengel

[1] Computer Science Department, University of Kaiserslautern, Germany
[2] Knowledge Management Department, DFKI GmbH, Kaiserslautern, Germany
{firstname.lastname}@dfki.de

Abstract. Efficient usage of the knowledge provided by the Linked Data community is often hindered by the need for domain experts to formulate the right SPARQL queries to answer questions. For new questions they have to decide which datasets are suitable and in which terminology and modelling style to phrase the SPARQL query.
In this work we present an evolutionary algorithm to help with this challenging task. Given a training list of source-target node-pair examples our algorithm can learn patterns (SPARQL queries) from a SPARQL endpoint. The learned patterns can be visualised to form the basis for further investigation, or they can be used to predict target nodes for new source nodes.
Amongst others, we apply our algorithm to a dataset of several hundred human associations (such as "circle - square") to find patterns for them in DBpedia. We show the scalability of the algorithm by running it against a SPARQL endpoint loaded with > 7.9 billion triples. Further, we use the resulting SPARQL queries to mimic human associations with a Mean Average Precision (MAP) of 39.9% and a Recall@10 of 63.9%.

1 Introduction

The Semantic Web [1] and its Linked Data [2] movement have brought us many great, interlinked and freely available machine readable RDF [13] datasets, often summarized in the Linking Open Data Cloud[3]. Being extracted from Wikipedia and spanning many different domains, DBpedia [3] forms one of the most central and best interlinked of these datasets.

Nevertheless, even with all this easily available data, using it is still very challenging: For a new question, one needs to know about the available datasets, which ones are best suited to answer the question, know about the way knowledge is modelled inside them and which vocabularies are used, before even attempting to formulate a suitable SPARQL[4] query to return the desired information. The noise of real world datasets adds even more complexity to this.

[3] http://lod-cloud.net/
[4] https://www.w3.org/TR/rdf-sparql-query/

In this paper we present a graph pattern learning algorithm that can help to identify SPARQL queries for a relation $\mathcal{R}$ between node pairs $(s,t) \in \mathcal{R}$ in a given knowledge graph G[5], where s is a source node and t a target node. $\mathcal{R}$ can for example be a simple relation such as "given a capital s return its country t" $\mathcal{R}_{cc}$ or a complex one such as "given a stimulus s return a response t that a human would associate" $\mathcal{R}_{ha}$.

To learn queries for $\mathcal{R}$ from G, without any prior knowledge about the modelling of $\mathcal{R}$ in G, we allow users to compile a ground truth set of example source-target-pairs $\mathcal{GT} \subseteq \mathcal{R}$ as input for our algorithm. For example, for relation $\mathcal{R}_{cc}$ between capital cities and their countries, the user could generate a ground truth list $\mathcal{GT} = \{$(`dbr:Berlin`, `dbr:Germany`), (`dbr:Paris`, `dbr:France`), (`dbr:Oslo`, `dbr:Norway`)$\}$. Given $\mathcal{GT}$ and the DBpedia SPARQL endpoint[6], our graph pattern learner then learns a set of graph patterns $gpl(\mathcal{GT}, G)$ such as:

$$gp_1\text{: } \{\texttt{?source dbo:country ?target}\}$$

$$gp_2\text{: } \{\texttt{?target dbo:capital ?source. ?target a dbo:Country}\}$$

In this paper, a graph pattern $gp \in gpl(\mathcal{GT}, G) \subset GP$ is an instance of the infinite set of SPARQL basic graph patterns[7] GP. Each gp has a corresponding SPARQL ASK and SELECT query. We denote their execution against G as ASK(gp) and SELECT(gp). The graph patterns can contain SPARQL variables, out of which we reserve `?source` and `?target` as special ones. A mapping Φ can be used to bind variables in gp before execution.

The resulting learned patterns can either be inspected or be used to predict targets by selecting all bindings for `?target` given a source node s_i:

$$\text{prediction}_{gp}(s_i) = \underset{?target}{\text{SELECT}}(\phi_{?source:=s_i}(gp))$$

For example, given the source node `dbr:London` the pattern gp_1 can be used to predict `dbr:United_Kingdom` $\in \text{prediction}_{gp_1}($`dbr:London`$)$.

The remainder of this paper is structured as follows: We present related work in Section 2, before describing our graph pattern learner in detail in Section 3. In Sections 4 and 5 we will then briefly describe visualisation and prediction techniques before evaluating our approach in Section 6.

2 Related Work

To the best of our knowledge, our algorithm is the first of its kind. It is unique in that it can learn a set of SPARQL graph patterns for a given input list of source-target-pairs directly from a given SPARQL endpoint. Additionally, it can cope with scenarios in which there is not a single pattern that covers all source-target-pairs.

[5] For our purpose G is a set of RDF triples, typically accessible via a given SPARQL endpoint.

[6] `http://dbpedia.org/sparql`

[7] `https://www.w3.org/TR/sparql11-query/#BasicGraphPatterns`

Many other algorithms exist, which learn vector space representations from knowledge graphs. An excellent overview of such algorithms can be found in [17]. We are however not aware that any of these algorithms have the ability of returning a list of SPARQL graph patterns that cover an input list of source-target-pairs.

There are other approaches that help formulating SPARQL queries, mostly in an interactive fashion such as RelFinder [10,11] or AutoSPARQL [15]. Their focus however lies on finding relationships between a short list of entities (not source-target-pairs) or interactively formulating SPARQL queries for a list of entities of a single kind. They cannot deal with entities of different kinds.

Wrt. SPARQL pattern learning, there is an approach for pattern based feature construction [14] that focuses on learning SPARQL patterns to use them as features for binary classification of entities. It can answer questions such as: does an entity belong to a predefined class? In contrast to that, our approach focuses on learning patterns between a list of source-target-pairs for entity prediction: given a source entity predict target entities. To simulate target entity prediction for a single given source with binary classification, one would need to train n classifiers, one for n potential target entities.

In the context of mining patterns for human associations and Linked Data, we previously focused on collecting datasets of semantic associations directly from humans [9,6], ranking existing facts according to association strengths [7,8] and mapping the Edinburgh Associative Thesaurus [12] to DBpedia [5]. None of these previous works directly focused on identifying existing patterns for human associations in existing datasets.

3 Evolutionary Graph Pattern Learner

The outline of our graph pattern learner is similar to the generic outline of evolutionary algorithms: It consists of individuals (in our case SPARQL graph patterns $gp_i \in GP$), which are evaluated to calculate their fitness. The fitter an individual is, the higher its chance to survive and reach the next generation. The individuals of a generation are also referred to as population. In each generation there is a chance to mate and mutate for each of the individuals. A population can contain the same individual (graph pattern) several times, causing fitter individuals to have a higher chance to mate and mutate over several generations.

As mentioned in the introduction, the training input of our algorithm is a list of ground truth source-target-pairs $gtp_i = (s_i, t_i) \in \mathcal{GT}$.

Due to size limitations, we will focus on the most important aspects of our algorithm in the following. For further detail please see our website[8] where you can find the source-code, visualisation and other complementary material.

[8] `https://w3id.org/associations`

3.1 Coverage

Before describing the realisation of the components of our evolutionary learner, we want to introduce our concept of coverage.

We say that a graph pattern gp_i covers, models or fulfils a source-target-pair (s_j, t_j) if the evaluation of its SPARQL ASK query returns true:

$$\text{ASK}(\phi_{\texttt{?source}:=s_i,\texttt{?target}:=t_i}(gp))$$

Our algorithm is not limited to learning a single best pattern for a list of ground truth pairs, but it can learn multiple patterns which together cover the list.

We realise this by invoking our evolutionary algorithm in several *runs*. In each run a full evolutionary algorithm is executed (with all its generations). After each run the resulting patterns are added to a global list of results. In the following runs, all ground truth pairs which are already covered by the patterns from previous runs become less rewarding for a newly learnt pattern to cover. Over its runs our algorithm will thereby re-focus on the left-overs, which allows us to maximise the coverage of all ground truth pairs with good graph patterns.

3.2 Fitness

In order to evaluate the fitness of a pattern, we define the following dimensions to capture what makes a pattern "good".

- High *recall*:
  A good pattern fulfils as many of the given ground truth pairs $\mathcal{GT}$ as possible:

$$\text{gt\_matches}_{gp} = |\{(s_i, t_i) \in \mathcal{GT} | \text{ASK}(\phi_{?source:=s_i,?target:=t_i}(gp))\}|$$
$$\text{recall}_{gp} = \frac{\text{gt\_matches}_{gp}}{|\mathcal{GT}|}$$

- High *precision*:
  A good pattern should also be precise. For each individual ground truth pair $(s_i, t_i) \in \mathcal{GT}$ we can define the precision as:

$$\text{precision}_{gp}((s_i, t_i)) = \frac{|\{t_i | t_i \in \text{prediction}_{gp}(s_i)\}|}{|\text{prediction}_{gp}(s_i)|}$$

  The target t_i should be in the returned result list and if possible nothing else. In other words, we are not searching for patterns that return thousands of potentially wrong target for a given source. Over all ground truth pairs, we can define the average precision for gp via the inverse of the average result lengths:

$$\text{avg result length}_{gp} = \underset{(s_i,t_i)}{\text{avg}} \; |\text{prediction}_{gp}(s_i)|$$
$$\text{precision}_{gp} = (\text{avg result length}_{gp})^{-1}$$

- High *gain*:
  A pattern discovered in run r is better if it covers those ground truth pairs $gtp \in \mathcal{GT}$ that aren't covered with high precisions in previous runs ($gp' \in run_q$) already:

  $$\text{gain}_{run_r,gp} = \sum_{gtp} \max\{0, \text{precision}_{gp}(gtp) - \max_{\forall q<r: gp' \in run_q} \text{precision}_{gp'}(gtp))\}$$

  Similarly, the potentially remaining gain can be computed as:

  $$\text{remains}_{run_r} = \sum_{gtp \in \mathcal{GT}} (1 - \max_{\forall q<r: gp' \in run_q} \text{precision}_{gp'}(gtp))$$

- No *over-fitting*:
  While precision is to be maximised, a good pattern should not *over-fit* to a single source or target from the training input.
- Short *pattern length* and low *variable count*:
  If all other considerations are similar, then a shorter pattern or one with less variables is preferable.
  Note, that this is a low priority dimension. A good pattern is not restricted to a shortest path between `?source` and `?target`. Good patterns can be longer and can have edges off the connecting path (e.g., see gp_2 in Section 1).
- Low execution *time* & *timeout*:
  Last but not least, to have any practical relevance, good patterns should be executable in a short *time*. Especially during the training phase, in which many queries are performed that take too long, we need to make sure to early terminate such queries on both, the graph pattern learner and the endpoint (cf. Section 3.7). In case the query was aborted due to a *timeout* and only a partial result obtained, it should not be trusted.

Based on these considerations, we define the *fitness* of an individual graph pattern as a tuple of real numbers with the following optimization directions. When comparing the fitness of two patterns, the fitness tuples for now are compared lexicographically.

1. **Remains** (max): Remaining precision sum remains_{run_r} in the current run r (see Section 3.1). Patterns found in earlier runs are considered better.
2. **Score** (max): A derived attribute combining gain with a configurable multiplicative punishment for over-fitting patterns.
3. **Gain** (max): The summed gained precision over the remains of the current run r: $\text{gain}_{run_r,gp}$. In case of timeouts or incomplete patterns the gain is set to 0.
4. F_1**-measure** (max): F_1-measure for precision and recall of this pattern.
5. **Average Result Lengths** (min): avg result length_{gp}.
6. **Recall (Ground Truth Matches)** (max): gt_matches_{gp}.
7. **Pattern Length** (min): The number of triples this pattern contains.
8. **Pattern Variables** (min): The number of variables this pattern contains.
9. **Timeout** (min): Punishment term for timeouts (0.5 for a soft and 1.0 for a hard timeout) (see Section 3.7 and gain).
10. **Query Time** (min): The evaluation time in seconds. This is particularly relevant since it hints at the real complexity of the pattern. I.e., a pattern may objectively have a small number of triples and variables, but its evaluation could involve a large portion of the dataset.

3.3 Initial Population

In order to start any evolutionary algorithm an initial population needs to be generated. The main objective of the first population is to form a starting point from which the whole search space is reachable via mutations and mating over the generations. While the initial population is not meant to immediately solve the whole problem, a poorly chosen initial population results in a lot of wasted computation time.

The starting point of our algorithm are single triple SPARQL BGP queries, consisting only of variables with at least a `?source` and `?target` variable, e.g.:

$$\{\texttt{?source ?p1 ?v1.}\}$$

While having a small chance of survival (direct evaluation would typically yield bad fitness), such patterns can re-combine (see mating in Section 3.4) with other patterns to form good and complete patterns in later generations.

For prediction capabilities, we are searching graph patterns which connect `?source` and `?target`, our algorithm mostly fills the initial population with path patterns of varying lengths l between `?source` and `?target`. Initially such a path pattern purely consists of variables and is directed from source to target:

$$\{\texttt{?source } ?p_1\ ?n_1.\ \ldots\ ?n_i\ ?p_{i+1}\ ?n_{i+1}\ .\ \ldots\ ?n_{l-1}\ ?p_l\ \texttt{?target.}\}$$

For example a pattern of desired length of $l = 3$ looks like this:

$$\{\texttt{?source ?p1 ?n1. ?n1 ?p2 ?n2. ?n2 ?p3 ?target.}\}$$

As longer patterns are less desirable, they are generated with a lower probability. Furthermore, we randomly flip each edge of the generated patterns, in order to explore edges in any direction.

In order to reduce the high complexity and noise introduced by patterns only consisting of variables, we built in a high chance to immediately subject them to the fix variable mutation (see Section 3.5).

3.4 Mating

In each generation there is a configurable chance for two patterns to mate in order to exchange information. In our algorithm this is implemented in a way that mating always creates two children, having the benefit of keeping the amount of individuals the same. Each child has a dominant and a recessive parent. The child will contain all triples that occur in both parents. Additionally, there is a high chance to select each of the remaining triples from the dominant parent and a low chance to select each of the remaining triples from the recessive parent. By this the children have the same expected length as their parents.

Furthermore, as variables from the recessive parent could accidentally match variables already being in the child, and this can be beneficial or not, we add a 50 % chance to rename such variables before adding the triples.

3.5 Mutation

Besides mating, which exchanges information between two individuals, information can also be gained by mutation. Each individual in a population has a configurable chance to mutate by the following (non exclusive) mutation strategies. Currently, all but one of the mutation operations can be performed on the pattern itself (local) without issuing any SPARQL queries. The mutation operations also have different effects on the pattern itself (grow, shrink) and on its result size (harden, loosen).

- **introduce var** select a component (node or edge) and convert it into a variable (loosen) (local)
- **split var** select a variable and randomly split it into 2 vars (grow, loosen) (local)
- **merge var** select 2 variables and merge them (shrink, harden) (local)
- **del triple** delete a triple statement (shrink, loosen) (local)
- **expand node** select a node, and add a triple from its expansion (grow, harden) (local for now)
- **add edge** select 2 nodes, add an edge in between if available (grow, harden) (local for now)
- **increase dist** increase distance between source and target by moving one a hop away (grow) (local)
- **simplify pattern** simplify the pattern, deleting unnecessary triples (shrink) (local) (cf. Section 3.7)
- **fix var** select a variable and instantiate it with an IRI, BNode or Literal that can take its place (harden) (SPARQL) (see below)

In a single generation sequential mutation (by different strategies in the order as above) is possible.

We can generally say that introducing a variable loosens a pattern and fixing a variable hardens it. Patterns which are too loose will generate a lot of candidates and take a long time to evaluate. Patterns which are too hard will generate too few solutions, if any at all. Very big patterns, even though very specific can also exceed reasonable query and evaluation times.

Fix Var Mutation Unlike the other mutations, the fix var mutation is the only one which makes use of the underlying dataset via the SPARQL endpoint G, in order to instantiate variables with an IRI, BNode or Literal. As it is one of the most important mutations and also because performing SPARQL queries is expensive, it can immediately return several mutated children.

For a given pattern gp we randomly select one of its variables `?v` (excluding `?source` and `?target`). Additionally, we sample up to a defined number of source-target-pairs from the ground truth which are not well covered yet (high potential gain). For each of these sampled pairs (s_s, t_s) we issue a SPARQL Select query of the form:

$$\{\ \texttt{SELECT distinct ?v \{ VALUES (?source ?target) \{ (}s_s\texttt{, }t_s\texttt{) \} }gp\texttt{ \} }\ \}$$

We collect the possible instantiations for `?v`, count them over all queries and randomly select (with probabilities according to their frequencies) up to a defined number of them. Each of the selected instantiations forms a separate child by replacing `?v` in the current pattern.

3.6 Selection and Keeping the Population Healthy

After each generation the next generation is formed by the surviving (fittest) individuals from n tournaments of k randomly sampled individuals from the previous generation.

We also employ two techniques, to counter population degeneration in local maxima and make our algorithm robust (even against non-optimal parameters):

- In each generation we re-introduce a small number of newly generated initial population patterns (see Section 3.3).
- Each generation updates a hall of fame, which will preserve the best patterns ever encountered over the generations. In each generation a small number of the best of these all-time best patterns is re-introduced.

3.7 Real World Considerations

In the following, we will briefly discuss practical problems that we encountered and necessary optimizations we used to overcome them. We implemented our graph pattern learner with the help of the DEAP (Distributed Evolutionary Algorithms in Python) framework [4].

Batch Queries The single most important optimization of our algorithm lies in the reduction of the amount of issued queries by using batch queries. This mostly applies to the queries for fitness evaluation (Section 3.2). It is a lot more efficient to run several sub-queries in one big query and to only transport the ground truth pairs to the endpoint once (via VALUES), than to ask for each result separately.

Timeouts & Limits Another mandatory optimization involves the use of timeouts and limits for all queries, even if they usually only return very few results in a short time. We found that a few run-away queries can quickly lead to congestion of the whole endpoint and block much simpler queries.

Timeouts are also especially useful as a reliable proxy to exclude too complicated graph patterns. Even seemingly simple patterns can take a very long time to evaluate based on the underlying dataset and its distribution.

Fit To Live Filter Apart from timeouts we use a filter which checks if mutants and children are actually desirable (e.g., length and variable count in boundaries, pattern is complete and connected), meaning fit to live, before evaluating them. If not, the respective parent takes their place in the new population, allowing for a much larger part of the population to be viable.

Parallelization, Caching, Query Canonicalization and Noise Two other crucial optimizations to reduce the overall run-time of the algorithm are parallelization and client side caching. Evolutionary algorithms are easy to parallelize via parallel evaluation of all individuals, but in our case the SPARQL endpoint quickly becomes the bottleneck. Ignoring the limits of the queried endpoint will resemble a denial of service attack. For most of our experiments we hence use an internal LOD cache with exclusive access for our learning algorithm. In case the

algorithm is run against public endpoints we suggest to only use a single thread in order not to disturb their service (fair use).

Client side caching further helps to reduce the time spent on evaluating graph patterns, by only evaluating them once, should the same pattern be generated by different sequences of mutation and mating operations. To identify equivalent patterns despite different syntactic surface forms, we had to solve SPARQL BGP canonicalization (NP-complete). We were able to to reduce the problem to RDF graph canonicalization and achieve good practical run-times with RGDA1 [16].

In the context of caching, one other important finding is that many SPARQL endpoints (especially the widely used OpenLink Virtuoso) often return incomplete and thereby non-deterministic results by default. Unlike many other search algorithms, an evolutionary algorithm has the benefit that it can cope well with such non-determinism. Hence, when caching is used, it is helpful to reduce, but not completely remove redundant queries.

Pattern Simplification Last but not least, as our algorithm can create patterns that are unnecessarily complex, it is useful to simplify them. We developed a pattern simplification algorithm, which given a complicated graph pattern gp_c finds a minimal equivalent pattern gp_s with the same result set wrt. the `?source` and `?target` variables. The simplification algorithm removes unnecessary edges, such as redundant parallel variable edges, edges between and behind fixed nodes and unrestricting leaf branches.

4 Visualisation

After presenting the main components of our evolutionary algorithm in the previous section, we will now briefly present an interactive visualisation[9]. As the learning of our evolutionary algorithm can produce many graph patterns, the visualisation allows to quickly get an overview of the resulting patterns in different stages of the algorithm.

Figure 1 (left) shows a screen shot of the visualisation of a single learned graph pattern. In the sidebar the user can select between individual generations, the results of a whole run or the overall results (default) to inspect the outcomes at various stages of the algorithm. Afterwards, the individual result graph patterns can be selected. Below these selection options the user can inspect statistics about the selected graph pattern including its fitness, a list of matching training ground truth pairs and the corresponding SPARQL SELECT query for the pattern. Links are provided to perform live queries on the SPARQL endpoint.

At each of the stages, the user can also get an overview of the precision coverage of a single pattern (as can be seen in Figure 1 (right)) or the accumulated coverage over all patterns.

[9] Also available at `https://w3id.org/associations`.

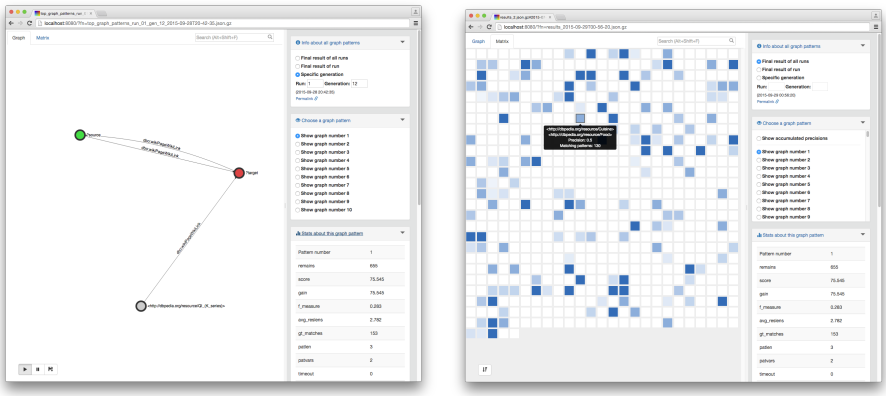

Fig. 1. Visualisation of graph pattern 1 from run 1, generation 12 (left) and the precision vector over all training ground truth pairs of graph pattern 1 (right). Each block represents a $(?source, ?target)$ pair from the ground truth training set. The darker its colour the higher the precision for the ground truth pair.

5 Prediction

As already mentioned in the introduction, the learned patterns can be used to predict targets for a given source. The basic idea is to insert a given source s_i in place of the `?source` variable in each of the learned patterns $gp \in gpl(\mathcal{GT}, G)$ and execute a SPARQL Select query over the `?target` variable (c.f., $\text{prediction}_{gp}(s_i)$ in Section 1).

5.1 Query Reduction Technique

While interesting for manual exploration, for practical prediction purposes the amount of learned graph patterns can easily become too large by discovering many very similar patterns that are only differing in minor aspects.

One realisation from visualising the resulting patterns gp, is that we can use their precision vectors wrt. the ground truth pairs to cluster graph patterns. The i-th component of $\vec{pv}_{gp}$ is defined by the precision value corresponding to the i-th ground truth source-target-pair $stp_i \in \mathcal{GT}$:

$$pv_{gp,i} = \text{precision}_{gp}(stp_i)$$

We employ several standard clustering algorithms on $\vec{pv}_{gp}$ and select the best patterns $\text{cluster}(gpl(\mathcal{GT}, G))$ in each cluster as representatives to reduce the amount of queries. By default our algorithm applies all of these clustering techniques and then selects the one which minimises the precision loss at the desired number of queries to be performed during prediction.

In our tests we could observe, that clustering (e.g., with hierarchical scaled euclidean ward clustering) allows us to reduce the number of performed SPARQL queries to 100 for all practical purposes with a precision loss of less than 1%.

5.2 Fusion Variants

When used for prediction, each graph pattern gp creates an unordered list of possible target nodes $t_j \in \text{prediction}_{gp}(s_i)$ for an inserted source node s_i. We evaluated the following fusion strategies to combine and rank the returned target candidates t_j (higher fusion value means lower rank):

- **target occurrences**: a simple occurrence count of each of the targets over all graph patterns.
- **scores**: sum of all graph pattern scores (from the graph pattern's fitness) for each returned target.
- **f-measures**: sum of all graph pattern F_1-measures (from the graph pattern's fitness) for each returned target.
- **gp precisions**: sum of all graph pattern precisions (from the graph pattern's fitness) for each returned target.
- **precisions**: sum of the actual precisions per graph pattern in this prediction.

By default our algorithm will calculate them all, allowing the user to pick the best performing fusion strategy for their use-case.

6 Evaluation

In order to evaluate our graph pattern learner, we performed several experiments which we will describe in the following.

We ran our experiments against a local Virtuoso 7.2 SPARQL endpoint containing over 7.9 G triples, from many central datasets[10] of the LOD cloud, denoted as G in the following.

6.1 Single Pattern Re-Identification

One of our claims is that our algorithm can learn good SPARQL queries for a relation $\mathcal{R}$ represented by a set of ground truth source-target-pairs $\mathcal{GT}$. In order to evaluate this, we started with simple relations such as "given a capital s return its country t" (see $\mathcal{R}_{cc}$ in Section 1). For each $\mathcal{R}$, we used a generating SPARQL query gp_g (such as gp_2 from Section 1) to generate $\mathcal{GT} \subset \text{SELECT}(gp_g)$, then executed our graph pattern learner $gpl(\mathcal{GT}, G)$ and checked if gp_g was in the resulting patterns:

$$gp_g \stackrel{?}{\in} gpl(\mathcal{GT}, G)$$

The result of these experiments is that our algorithm is able to re-identify such simple, readily modelled relations $\mathcal{R}$ in 100% of our test cases (typically within the first run, so the first 3 minutes). While this might sound astonishing, it is merely a test that our algorithm can perform the simplest of its tasks: If there is a single SPARQL BGP pattern gp that models the whole training list $\mathcal{GT}$ in G, then our algorithm is quickly able to find it via the fix var mutation in Section 3.5. Due to the page limit, we omit further details and instead turn to a more complex relation in the next section.

[10] Most notably: DBpedia 2015-04 (en, de), Freebase, Yago, Wikidata, GeoNames, DBLP, Wordnet and BabelNet.

6.2 Learning Patterns for Human Associations from DBpedia

Two additional claims are that our algorithm can learn a set of patterns, which cover a complex relation $\mathcal{R}$ that is not readily modelled in G, and that we can use the resulting patterns for prediction. Hence, in the following we focus on one such complex relation $\mathcal{R}_{ha}$: human associations. We will present some of the identified patterns and then evaluate the prediction quality.

Dataset Human associations are an important part of our thinking process. An *association* is the mental connection between two ideas: a *stimulus* (e.g., "pupil") and a *response* (e.g., "eye"). We call such associations *strong associations* if more than 20 % of people agree on the response.

In the following, we focus on a dataset of 727 strong human associations (corresponding to $\sim$ 25.5 K raw associations) from the Edinburgh Associative Thesaurus [12] that we previously already mapped to DBpedia Entities [5]. The dataset contains stimulus-response-pairs such as (`dbr:Pupil`, `dbr:Eye`), (`dbr:Stanza`, `dbr:Poetry`) and (`dbr:Paris`, `dbr:France`).[11]

We randomly split our 727 ground truth pairs into a training set $\mathcal{GT}_{\text{train}}$ of 655 and a test set $\mathcal{GT}_{\text{test}}$ of 72 pairs (10 % random split). All training, visualising and development has been performed on the training set in order to reduce the chance of over-fitting our algorithm to our ground truth.

Basic Statistics We ran the algorithm ($gpl(\mathcal{GT}_{\text{train}}, G)$) on G with a population size of 200, a maximum of 20 generations each in a maximum of 64 runs. The first 5 runs of our algorithm are typically completed within 3, 6, 9, 13 and 15 minutes. In the first couple of minutes all of the very simple patterns that model a considerable fraction of the training set's pairs are found. With the mentioned settings the algorithm will terminate after around 3 hours. It finds roughly 530 graph patterns with a score > 2 (cf. Section 3.2).

Notable Learned Graph Patterns Due to the page limit, we will briefly mention only 3 notable patterns from the resulting learned patterns in this paper. We invite the reader to explore the full results online[12] with the interactive visualisation presented in Section 4. The three notable patterns we want to present here are:

$$\{\texttt{?source gold:hypernym ?target}\}$$

$$\{\texttt{?source dbo:wikiPageWikiLink ?target. ?target dbo:wikiPageWikiLink ?source}\}$$

$$\{\texttt{?source dbo:wikiPageWikiLink ?target. ?v0 skos:exactMatch ?v1. ?v1 dbprop:industry ?target}\}$$

The first two are intuitively understandable patterns which typically are amongst the top patterns. The first one shows that human associations often seem to be represented via `gold:hypernym` in DBpedia (the response is often a hypernym (broader term) for the stimulus). The second one shows that associations often correspond to bidirectionally linked Wikipedia articles. The third pattern

[11] The full dataset is available at `https://w3id.org/associations`.
[12] `https://w3id.org/associations`

represents a whole class of intra-dataset learning by making use of a connection of the `?target` to BabelNet's `skos:exactMatch`.

Prediction & Fusion Strategies Evaluation As human associations are not readily modelled in DBpedia, it is difficult to assess the quality of the learned patterns gp directly. Hence, we evaluate the quality indirectly via their prediction quality on the test-set $\mathcal{GT}_{\text{test}}$.

For each of the $(s_t, t_t) \in \mathcal{GT}_{\text{test}}$ we generate a ranked target list $rtpl_{s_t} = [tp_1, \ldots, tp_n]$ of target predictions tp_i. The list is the result of one of the fusion variants (cf. Section 5.2) after clustering (cf. Section 5.1). For evaluation, we can then check the rank r_t of t_t in $rtpl_{s_t}$ (lower ranks are better). If $t_t \notin rtpl_{s_t}$, we set $r_t = \infty$.

An example of a ranked target prediction list (for the fusion method *precisions*) for source $s_t =$`dbr:Sled` is the ranked list: $rtpl_{\texttt{dbr:Sled}} = [$`dbr:Snow`, `dbr:Christmas`, `dbr:Deer`, `dbr:Kite`, `dbr:Transport`, `dbr:Donkey`, `dbr:Ice`, `dbr:Ox`, `dbr:Obelisk`, `dbr:Santa_Claus`$]$. In this case the ground truth target $t_t =$`dbr:Snow` is at rank $r_t = 1$. As we can see most of the results are relevant as associations to humans. Nevertheless, for the purpose of our evaluation, we will only consider the single t_t corresponding to a s_t as relevant and all other tp_i as irrelevant.

Based on the ranked result lists, we can calculate the Recall@k[13], Mean Average Precision (MAP) and Normalised Discounted Cumulative Gain of the various fusion variants over the whole test set $\mathcal{GT}_{\text{test}}$, as can be seen in Table 1 and Figure 2.

We also calculate these metrics for several baselines, which try to predict the target nodes from the 1-neighbourhood (bidirectionally, incoming or outgoing) by selecting the neighbour with the highest PageRank, HITS score, in- and out-degree [18,19]. As can be seen, all our fusion strategies significantly outperform the baselines.

7 Conclusion & Outlook

In this paper we presented an evolutionary graph pattern learner. The algorithm can successfully learn a set of patterns for a given list of source-target-pairs from a SPARQL endpoint. The learned patterns can be used to predict targets for a given source.

We use our algorithm to identify patterns in DBpedia for a dataset of human associations. The prediction quality of the learned patterns after fusion reaches a Recall@10 of 63.9 % and MAP of 39.9 %, and significantly outperforms PageRank, HITS and degree based baselines.

The algorithm, the used datasets and the interactive visualisation of the results are available online[14].

In the future, we plan to enhance our algorithm to support Literals in the input source-target-pairs, which will allow us to learn patterns directly from

[13] We don't provide Precision@k, as it degenerates to Recall@k/k due to the fact that we only have 1 relevant target per result of any (s_t, t_t).

[14] `https://w3id.org/associations`

Table 1. Recall@k, MAP and NDCG for our fusion variants and against baselines.

	Recall@1	Recall@2	Recall@3	Recall@4	Recall@5	Recall@10	MAP	NDCG
outdeg in	0.000	0.000	0.000	0.000	0.042	0.097	0.029	0.105
outdeg out	**0.069**	**0.125**	**0.153**	0.153	0.167	0.181	0.126	0.209
outdeg bidi	0.014	0.014	0.014	0.014	0.056	0.125	0.045	0.131
indeg in	0.056	0.111	**0.153**	0.167	0.181	**0.306**	0.129	0.207
indeg out	0.056	**0.125**	**0.153**	0.153	0.153	0.194	0.121	0.200
indeg bidi	0.042	0.069	0.111	0.139	0.139	0.194	0.104	0.205
pagerank in	**0.069**	**0.125**	**0.153**	**0.194**	**0.194**	0.292	**0.140**	**0.219**
pagerank out	0.056	0.097	**0.153**	0.153	0.167	0.208	0.117	0.195
pagerank bidi	0.056	0.069	0.111	0.139	0.153	0.236	0.113	**0.219**
hits in	0.014	0.028	0.042	0.069	0.083	0.111	0.046	0.095
hits out	0.056	0.056	0.111	0.125	0.153	0.181	0.102	0.181
hits bidi	0.014	0.042	0.042	0.056	0.069	0.125	0.050	0.110
scores	0.236	0.278	0.375	0.389	0.389	0.556	0.323	0.413
gp precisions	0.250	0.319	0.417	0.500	0.528	**0.639**	0.365	0.457
precisions	0.250	**0.361**	0.444	0.486	0.528	0.625	0.371	0.460
target occs	0.278	0.319	0.458	**0.528**	0.528	0.611	0.381	0.466
f measures	**0.306**	0.347	**0.472**	0.500	**0.542**	0.611	**0.399**	**0.479**

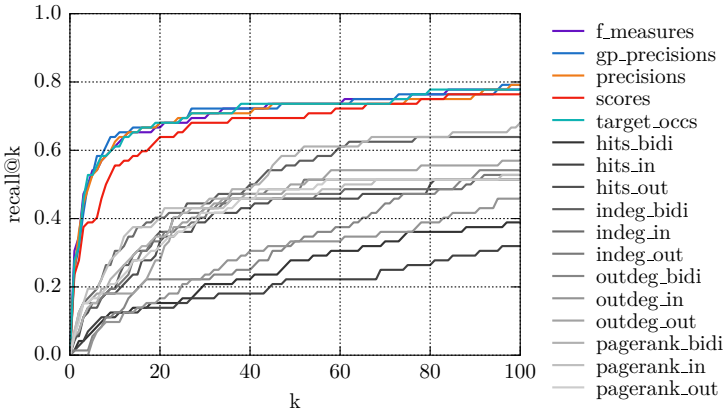

Fig. 2. Recall@k over the different fusion variants and against baselines.

lists of textual inputs. Further, we are investigating mutations, for example to introduce FILTER constraints. We also plan to investigate the effects of including negative samples (currently we only use positive samples and treat everything else as negative).

Additionally, we plan to employ more advanced late fusion techniques, in order to learn when to trust the prediction of which pattern. As this idea is conceptually close to interpreting the learned patterns as a feature vector (with understandable and executable patterns to generate target candidates), we plan to investigate combinations of our algorithm with approaches that learn vector space representations from knowledge graphs.

This work was supported by the University of Kaiserslautern CS PhD scholarship program and the BMBF project MOM (Grant 01IW15002).

References

1. Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web Scientific American 284(5), 34–43 (2001)
2. Bizer, C., Heath, T., Berners-Lee, T.: Linked Data - The Story So Far. International Journal on Semantic Web and Information Systems 5(3), 1–22 (2009)
3. Bizer, C., Lehmann, J., Kobilarov, G., Auer, S., Becker, C., Cyganiak, R., Hellmann, S.: DBpedia - A crystallization point for the Web of Data. Web Semantics: Science, Services and Agents on the World Wide Web 7(3), 154–165 (2009)
4. Fortin, F.A., De Rainville, F.M., Gardner, M.A., Parizeau, M., Gagné, C.: DEAP: Evolutionary Algorithms Made Easy. Journal of Machine Learning Research 13, 2171–2175 (2012)
5. Hees, J., Bauer, R., Folz, J., Borth, D., Dengel, A.: Edinburgh Associative Thesaurus as RDF and DBpedia Mapping. The Semantic Web: In: ESWC SE. Springer, Heraklion, Crete, Greece (2016)
6. Hees, J., Khamis, M., Biedert, R., Abdennadher, S., Dengel, A.: Collecting Links between Entities Ranked by Human Association Strengths. In: ESWC. vol. 7882, pp. 517–531. Springer LNCS, Montpellier, France (2013),
7. Hees, J., Roth-berghofer, T., Biedert, R., Adrian, B., Dengel, A.: BetterRelations: Using a Game to Rate Linked Data Triples. In: KI 2011: Advances in Artificial Intelligence. pp. 134–138. Springer (2011)
8. Hees, J., Roth-Berghofer, T., Biedert, R., Adrian, B., Dengel, A.: BetterRelations: Collecting Association Strengths for Linked Data Triples with a Game. In: Search Computing, vol. 7538, pp. 223–239. Springer LNCS (2012)
9. Hees, J., Roth-Berghofer, T., Dengel, A.: Linked Data Games: Simulating Human Association with Linked Data. In: LWA. pp. 255–260. Kassel, Germany (2010)
10. Heim, P., Hellmann, S., Lehmann, J., Lohmann, S., Stegemann, T.: RelFinder: Revealing Relationships in RDF Knowledge Bases. In: SAMT 2009. LNCS, vol. 5887, pp. 182–187. Springer, Graz, Austria (2009)
11. Heim, P., Lohmann, S., Stegemann, T.: Interactive Relationship Discovery via the Semantic Web. In: ESWC 2010. LNCS, pp. 303–317. Springer, Heraklion, Greece
12. Kiss, G.R., Armstrong, C., Milroy, R., Piper, J.: An associative thesaurus of English and its computer analysis. In: The Computer and Literary Studies, pp. 153–165. Edinburgh University Press, Edinburgh, UK (1973)
13. Klyne, G., Carroll, J.J.: Resource Description Framework (RDF): Concepts and Abstract Syntax (2004), `http://www.w3.org/TR/rdf-concepts/`
14. Lawrynowicz, A., Potoniec, J.: Pattern based feature construction in semantic data mining. Int. J. on SemWeb and Information Systems (IJSWIS) 10(1), 27–65 (2014)
15. Lehmann, J., Bühmann, L.: AutoSPARQL: Let users query your knowledge base. In: ESWC. LNCS, vol. 6643, pp. 63–79. Springer, Heraklion, Crete, Greece (2011)
16. McCusker, J.P.: WebSig: A Digital Signature Framework for the Web. Ph.D. thesis, Rensselaer Polytechnic Institute, Troy, NY (2015)
17. Nickel, M., Murphy, K., Tresp, V., Gabrilovich, E.: A Review of Relational Machine Learning for Knowledge Graphs pp. 1–23 (2015)
18. Reddy, D., Knuth, M., Sack, H.: DBpedia GraphMeasures (2014), `http://semanticmultimedia.org/node/6`
19. Thalhammer, A., Rettinger, A.: PageRank on Wikipedia: Towards General Importance Scores for Entities. In: Know@LOD&CoDeS 2016. CEUR-WS Proceedings.