Finding Motifs in Knowledge Graphs using Compression

Peter Bloem
[0000−0002−0189−5817]

Knowledge Representation and Reasoning group,
Vrije Universiteit Amsterdam
`vu@peterbloem.nl`
`peterbloem.nl`

Abstract. We introduce a method to find *network motifs* in knowledge graphs. Network motifs are useful patterns or meaningful subunits of the graph that recur frequently. We extend the common definition of a network motif to coincide with a *basic graph pattern.* We introduce an approach, inspired by recent work for simple graphs, to induce these from a given knowledge knowledge graph, and show that the motifs found reflect the basic structure of the graph. Specifically, we show that in random graphs, no motifs are found, and that when we insert a motif artificially, it can be detected. Finally, we show the results of motif induction on three real-world knowledge graphs.

1 Introduction

Knowledge graphs are an extremely versatile and flexible data model. They allow knowledge to be encoded without a predefined format and they are extremely robust in the face of missing data. This versatility comes at a price. For a given knowledge graph, it can be difficult to see the forest for the trees: how is the graph structured at the lowest level? What kind of things can I ask of what types of entities? What are small, recurring patterns that might represent a novel insight into the data? Answering these questions could benefit problem domains like graph simplification, graph navigation and schema induction.

In the domain of unlabeled simple graphs, *network motifs* [9] were introduced as a tool to provide insight into local graph structure. Network motifs are small subgraphs whose frequency in the graph is unexpected with respect to a *null model.*

Unfortunately, estimating this probability usually requires repeating the subgraph count on many samples from the null model. To avoid this costly operation, [2] introduces an alternative method, using *compression* as a heuristic for motif relevance: the better a motif compresses the data, the more likely it is to be meaningful.

In this paper, we extend this compression-based motif analysis to *knowledge graphs.* For the purposes of this research we define knowledge graphs as labeled, directed multigraphs. Nodes are uniquely labeled with entity names, and links

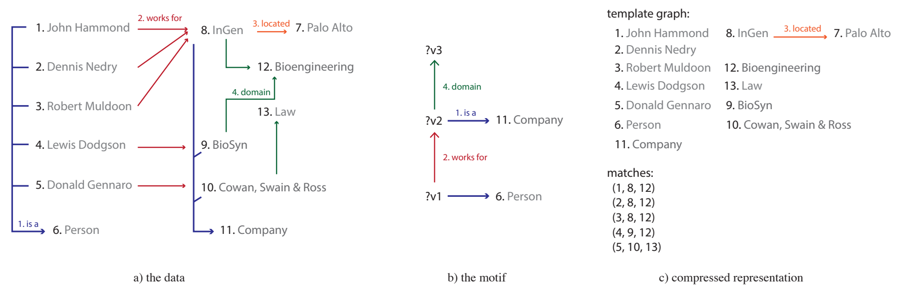

Fig. 1. An example of the principle behind our motif code. a) A basic knowledge graph. We consider only the indices of the nodes and relations, not their labels. b) A motif that occurs frequently. c) A compressed representation; we remove all edges that are part of an occurrence of the motif and store separately which nodes match the motif.

are non-uniquely labeled with relations. We extend the definition of a motif to that of a *basic graph pattern*: a small graph labeled with both variables and explicit entitites and relations. A pattern matches if the variables can be replaced with specific values from the graph so that the pattern becomes a subgraph as a result.

The intuition behind our method is that we can use graph patterns to *compress the graph*: we store the pattern, its instances, and the remainder of the graph. The better this representation compresses the graph, the more relevant the pattern. Figure 1 illustrates the principle. In Section 1.1, we justify this intuition more formally.

We perform several experiments to show that our method returns meaningful subgraphs. First we test the intuition that a random graph should contain no motifs. We also show that when we artificially insert motifs into a random graph, we can then detect these as motifs. Finally, we show the results of motif analysis on three real-world knowledge graphs, compared to the baseline of selecting the most frequent graph patterns.

All code and datasets used in this paper are available online.[1]

Related Work Network motifs for unlabeled simple graphs were introduced in [9]. A more comprehensive overview of the related literature can be found in [2]. In [2], the principle of Minimum Description Length (MDL) was first connected to motif analysis. However, the idea had earlier been exploited for detecting meaningful subgraphs in the SUBDUE algorithm [4].

A few other methods have been proposed for inducing the structure of a given knowledge graph in terms of subgraphs. In [13], the authors use the principle of characteristic sets to characterize a knowledge graph in terms of the star patterns it contains. In [12], they show that the majority of the LOD cloud can be efficiently described using such principles, showing the highly tabular

[1] `https://github.com/MaestroGraph/motive-rdf`

structure of many knowledge graphs. In [17], association rule mining is used to induce basic patterns in the graph.

To the best of our knowledge, ours is the first method presented that can potentially induce any basic graph pattern.

1.1 Preliminaries

Minimum Description Length Our method is based on the MDL principle: we should favour models that compress the data. We will show briefly how this intuition can be made mathematically precise. For more details, we refer the reader to [6] for MDL in general, and to [3], for a more extensive discussion these principles in the domain of graph analysis.

Let $\mathbb{B}$ be the set of all finite-length binary strings. We use $|b|$ to represent the length of $b \in \mathbb{B}$. Let $\log(x) = \log_2(x)$. A *code* for a set of objects $\mathcal{X}$ is an injective function $f : \mathcal{X} \to \mathbb{B}$. All codes in this paper are *prefix-free*: no code word is the prefix of another. We will denote a *codelength function* with the letter L, ie. $L(x) = |f(x)|$. We commonly compute $L(x)$ directly, without first computing $f(x)$.

There is a strong relation between codes and probability distributions: for each probability distribution p on $\mathcal{X}$, there exists a prefix-free code L such that for all $x \in \mathcal{X}$: $-\log p(x) \leq L(x) < -\log p(x) + 1$. Inversely, for every prefix-free code L for $\mathcal{X}$, there exists a probability distribution p such that for all $x \in \mathcal{X}$: $p(x) = 2^{-L(x)}$. For proofs, see [6, Section 3.2.1] or [5, Theorem 5.2.1].

Relevance testing We will use the MDL principle to perform a hypothesis test. Assume we have some data $x \in \mathbb{B}$ and a null hypothesis stating that it was sampled from distribution p^{null} (with corresponding code L^{null}). A simple but crucial result, known as the *no-hypercompression inequality* [6, p103] tells us that the probability of sampling any data x from p^{null} that can be described in less than $L^{\text{null}}(x) - k$ or more bits, *using any code* is less than 2^{-k}. Thus, we can reject the hypothesis that the data was sampled from p^{null} by designing a code which compresses the data better than L^{null} by, say, 10 bits and rejecting the null hypothesis with confidence 2^{-10}. For a longer, more intuitive explanation of this principle in pattern induction, we refer the reader to [3].

Note that when we use this procedure to find motifs, we are not providing statistical evidence for the hypothesis that the motif is "correct" [3, Section 4.1]. We are simply using the principle of hypothesis testing as a *heuristic* for pattern mining. The only assertion we are proving (in a statistical sense) is that the data did not come from the null model.

Common codes In the construction of our graph codes, we require some simpler codes as building blocks. First, when we store any positive integer n, we do so with the code corresponding to the distribution $p^{\mathbb{N}}(n) = 1/(n(n+1))$, and denote it $L^{\mathbb{N}}(n)$. For nonnegative numbers we add 1 to the argument. For the full range of integers ($L^{\mathbb{Z}}$), we add an extra bit for the sign, and then use the first code for negative integers and the second for positive ones.

We will often need to encode *sequences* of integers as well. These will be highly skewed, with only a subset of integers occurring frequently, and others occurring infrequently or not at all. As noted in [15] a code based on the Pitman-Yor model [14] is very effective in such situations. Let $S = \langle S_1, ..., S_n \rangle$ be a sequence of integers of length n. We first store its members $m(S)$ in the order in which they occur: we first store n and the first member using $L^{\mathbb{N}}$ and then store each subsequent member by encoding the distance to the previous member using $L^{\mathbb{Z}}$. Having encoded the members of S we can store the sequence itself using the Pitman-Yor model as follows.

Let $f(A, B)$ be the frequency of symbol A in sequence B. We then store the complete sequence using the code corresponding to the following distribution:

$$p(S) = \prod_{i \in [1,k]} p(S_i \mid S_{1:i-1}) \;\text{ with }\; p(S_i \mid S') = \begin{cases} \frac{\alpha - d|m(S')|}{|m(S')|+\alpha} & \text{if} f(S_i, S') = 0 \\ \frac{f(S_i,S')-d}{|m(S')|+\alpha} & \text{otherwise} \end{cases}$$

See [15] for a more intuitive explanation. In all experiments we use $\alpha = 0.5$, $d = 0.1$. We will refer to the total resulting codelength as $L^{PY}(S)$.

2 Method

We will first give a precise definition of a knowledge graph as used in this paper. We will then describe the null model which is used both as a point of comparison in our hypothesis test, and within the motif code to compress the remainder of the graph. Next, we describe how to compress a graph using a given motif, and a set of instances. Finally, we will describe how to search for likely motifs using simulated annealing. Specifically, we model a knowledge graph as a multigraph with nodes and edges labeled with integers that map to entities and relations. This mapping is stored, but only the integer-labeled graph is modelled.[2]

A *knowledge graph* G, is a tuple $G = (v_G, r_G, E_G)$. $v_G \in \mathbb{N}$ is the number of nodes in the graph, and $r_G \in \mathbb{N}$ is the number of relations. We define the nodeset of G as $V_G = \{0, \ldots, v_G - 1\}$ and the relation-set as $R_G = \{0, \ldots, r_G\}$. The *tripleset* $E_G \subset V_G \times R \times V_G$ determines the edges of the graph and their labels: each triple $(s, r, o) \in E_G$ encodes the originated node s, the target node o and the relation r of an edge in the graph.

This definition is compatible with RDF data. We interpret literals as nodes, considered the same node if they are expressed by the same string.

A *pattern* M for graph G is a tuple (V_M, R_M, G, E_M). Let v_M and r_M indicate the number of variable nodes and variable links in M respectively, then $V_M \subseteq \{-v'_M, \ldots, v_G - 1\}$ and $R_M \subseteq \{-(r'_M + v'M), \ldots, -v'M, 0, \ldots, r_G - 1\}$, with $E_M \subset V_M \times R_M \times V_M$ representing the edges as before. That is; nodes can be labeled either with nonnegative integers referring to G's nodes or with negative integers representing a variable node, and similar for relations. The

[2] For practitioners this restriction is not noticeable, as the indices can simply be mapped back to the original strings when the found motifs are presented.

negative integers are always contiguous within a single pattern, with the highest representing the node labels and the lowest representing the edge labels

An *instance* for M in G is a pair of sequences of integers: $I = (I^n, I^r)$. I^n is a sequence of distinct integers of length v'_M. I^r is a sequence of non-distinct integers of length r'_M. For each edge $(s, p, o) \in E_M$ with any or all of s, p and o negative, there is a corresponding link in E_G with a negative s replaced by I^n_{-s}, a negative o replaced by I^n_{-o}, and a negative p replaced by $I^r_{-p-v'_M}$. Put simply: for a pattern to match, variable edges marked with the same negative integer, must map to the same relation in order for the pattern to match, but variable links labeled with different negative integers *may* map to the same relation. Variable nodes are always labeled distinctly and may never map to the same node in G. An instance describes a subgraph of G that *matches* the pattern M. Each edge in the motif may only match one edge in the graph. In other words, the occurrence of the motif in the graph must have as many edges as the motif itself.[3]

2.1 Null model

For a proper hypothesis test, we must compare the compression achieved by our motif code to the compression under a general model for knowledge graphs: a null model.

The most common null model in classical motif analysis is the degree-sequence model (also known as the configuration model [11]): a uniform distribution over all graphs with a particular degree sequence. We extend this to knowledge graphs by also including the degree of each relation: that is, degree of a relation is the frequency with which it occurs in the tripleset. Let a *degree sequence* D of length n be a triple of three integer sequences: $(D^{\text{in}}, D^{\text{rel}}, D^{\text{out}})$. If D is the degree sequence of a graph, then node i has D^{in}_i incoming links, D^{out}_i outgoing links and for each relation r, there are D^{rel}_r triples.

Let $\mathcal{G}_D$ be the set of all graphs with degree sequence D. Then the degree-sequence model can be expressed simply as

$$p^{\text{DS}}(G) = \frac{1}{|\mathcal{G}_D|}$$

for any G that satisfies D and $p(G) = 0$ otherwise. Unfortunately, there is no efficient way to compute $|\mathcal{G}_D|$ and even approximations tend to be costly for large graphs. Following the approach in [2], we define a fast approximation to the configuration model, which works well in practice for motif detection.

We can describe a knowledge graph by three length-m integer sequences: S, P, O, such that $\{(S_j, P_j, O_j)\}_j$ is the graph's tripleset. If the graph satisfies degree sequence D, then we know that S should contain node j D^{out}_j times, P should contain relation r D^{rel}_r times and O should contain node j D^{in}_j times. Let

[3] In this aspect our definition differs from the SPARQL Basic Graph Pattern. Patterns for which this distinction is relevant are rare, and patterns returned by our method are still compatible with SPARQL.

$\mathcal{S}_D$ be the set of all such triples of integer sequences satisfying D. We have

$$|\mathcal{S}_D| = \binom{m}{D^{\text{out}}_1, \ldots, D^{\text{out}}_n}\binom{m}{D^{\text{rel}}_1, \ldots, D^{\text{rel}}_{|R_G|}}\binom{m}{D^{\text{in}}_1, \ldots, D^{\text{in}}_n} .$$

While every member of $\mathcal{S}_D$ represents a valid graph satisfying D, many graphs are represented multiple times. Firstly, many elements of S_D contain the same link multiple times. We call the set without these elements $\mathcal{S}'{}_D \subset \mathcal{S}_D$. Secondly the links of the graph are listed in arbitrary order; if we apply the same permutation to all three lists S, P and O, we get a new representation of the same graph. Since we know that any element in $\mathcal{S}'_D$ contains only unique triples, we know that each graph is present exactly $m!$ times. This gives us

$$|\mathcal{G}_D| = |\mathcal{S}'_D|\frac{1}{m!} \leq |\mathcal{S}_D|\frac{1}{m!} .$$

We can thus use

$$p^{\text{EL}}_D(G) = \frac{m!}{\binom{m}{D^{\text{out}}_1, \ldots, D^{\text{out}}_n}\binom{m}{D^{\text{rel}}_1, \ldots, D^{\text{rel}}_{|R_G|}}\binom{m}{D^{\text{in}}_1, \ldots, D^{\text{in}}_n}} \leq p^{\text{DS}}(G).$$

Filling in the definition of the multinomial coefficient, and rewriting, we get a codelength of:

$$-\log p^{\text{EL}}_D(G) = 2\log(m!) - \sum_i \log(D^{\text{in}}_i!) - \sum_i \log(D^{\text{rel}}_i!) - \sum_i \log(D^{\text{out}}_i!)$$

as an approximation for the DS model. We call this the edgelist (EL) model. It gives a probability that always lower-bounds the configuration model, since it affords some probability mass to graphs that cannot exist. Experiments in the classical motif setting have shown that the EL model is an acceptable proxy for the DS model [2], especially considering the extra scalability it affords.

Encoding D In order to encode a graph with L^{EL}_D, we must first encode D.[4] For each of the three sequences D' in D we use the following model:

$$p(D') = \prod_i q^{\mathbb{N}}(D'_i) \qquad L(D') = -\sum_i \log q^{\mathbb{N}}(D'_i)$$

where $1^{\mathbb{N}}$ is any distribution on the natural numbers. This is an optimal encoding for D assuming that its members are independently drawn from $q^{\mathbb{N}}$. When we use p^{EL} as the null model, we use the data distribution for $q^{\mathbb{N}}$ to ensure that we have a lower bound to the optimal code-lengthn (in essence, we cheat, giving the null model a slightly lower than optimal codelength). When we use p^{EL} as part of the motif code, we must use a fair encoding, so we use the Pitman-Yor code to store each sequence in D.

[4] Or, equivalently, to make p^{EL} a complete distribution on all graphs, we must provide it with a prior on D.

2.2 Motif code

Having defined our representation of a knowledge graph, and a general null model for compressing such knowledge graphs, we can now define how we use a given motif (together with its instances) to compress a dataset.

We will assume that a target pattern M is given for the data G and that we have a set of instances $\mathcal{I}$ of M in G. Moreover, we require that all instances in $\mathcal{I}$ are mutually disjoint: no two subgraphs defined by a member of $\mathcal{I}$ may share an edge, but nodes may be shared. Given this information, we will define a motif code that will help us determine whether or not M is a likely motif for G. In section Section 2.3, we detail a method to search for pairs $(M, \mathcal{I})$ to pass to the motif code.

As described above, we can perform our relevance test with any compression method which exploits the pattern M and its instances $\mathcal{I}$ to store the graph efficiently. The better our method, the more motifs we will find. Note that there is no need for our code to be optimal in any sense. We know that we will not find all motifs that exist, and we will not use them optimally to represent the graph, but the test is still valid. This also means that we are free to trade off compression performance against efficiency of computation.

We store the graph by encoding various aspects, one after the other. The information in all of these together is sufficient to reconstruct the graph. Note that everything is stored using prefix-free codes, so that we can simply concatenate the codewords we get for each aspect, to get a codeword for the whole graph.

We also assume that we are given a code L^{base} for generic knowledge graphs (in practice, this will be the null model, although the motif code is valid for any base code).

We store, in order:

the graph dimensions We first store $|V_G|$, $|E_G|$ and $|R_G|$ using the generic code $L^{\mathbb{N}}(\cdot)$.

the pattern We store the structure of the pattern using the base code, and its labels as a sequence using the Pitman-Yor code.

the template This is the graph, minus all links occurring in instances of M. Let E'_G be E_G minus any link occurring in any member of $\mathcal{I}$. We then store (V_G, E'_G) using $L^{\text{base}}(\cdot)$.

the instances To store the instances, we view the connections between the nodes made by motifs as a hypergraph, and we extend the EL code to store it. The details are given below.

The precise computation of the codelength is given in Algorithm 1.

Encoding motif instances To encode a list of instances $\mathcal{I}$ of a given pattern M, we generalize the idea of the edgelist model described above.

To generalize this notion to arbitrary patterns, to be defined for a given template graph, we define the *degree constraint* $D^{\mathcal{I}}$ of a list of instances for a given pattern as follows: for each variable node i in the pattern, the degree constraint provides an integer sequence D^i of length v_G, indicating how often

Algorithm 1 The motif code $L^{\text{motif}}(G; M, \mathcal{I}, L^{\text{base}})$. Note that the nodes and relations of the graph are integers.

```
function codelength($G; M, \mathcal{I}, L^{\text{base}}$):
        a graph $G$, a pattern $M$
        instances $\mathcal{I}$ of $M$ in $G$, a code $L^{\text{base}}$.

$b_{\text{dim}} \leftarrow L^{\mathbb{N}}(|V_G|) + L^{\mathbb{N}}(|R_G|) + L^{\mathbb{N}}(|E_G|)$

// Turn the pattern into a normal knowledge graph
$E_{M'} \leftarrow$ the edges of $M$ with contiguous integer labels
$M' \leftarrow (|V_M|, |R_M|, E_{M'})$
$S_M \leftarrow$ the labels of M in canonical order
$b_{\text{pattern}} \leftarrow L^{\text{base}}(M') + L^{PY}(S_M)$

// Store the template graph
$E'_G \leftarrow E_G - \cup_{\mathcal{I} \in \mathcal{I}} \text{triples}(I)$
$b_{\text{template}} \leftarrow L_{\text{base}}((V_G, R_G, E'_G))$

$b_{\text{instances}} \leftarrow -\log p_M(\mathcal{I}) + \sum_{D \in D^{\mathcal{I}}} L^{PY}(D)$

return $b_{\text{dim}} + b_{\text{pattern}} + b_{\text{template}} + b_{\text{instances}}$
```

each node in the completed knowledge graph takes that position in the pattern. Similarly, for each variable link j in the pattern, the degree sequence provides an integer sequence C^j of length r_G indicating for each relation how often it takes that position in the pattern.

We store these sequences in the same manner as the degree sequence of the template graph, using the Pitman-Yor code for each.

Given this information, all we need to do is describe which of the possible sequences of matches for this pattern satisfying the given degree sequence we are encoding. As with the configuration model, the ideal is a uniform code over all possible configurations, for which we will define an approximation. Given w variable nodes in a pattern, and l variable links, we can define such a collection of instances using $w + l$ integer sequences: $N^1, \ldots, N^n, L^1, \ldots, L^l$, with the t-th instance defined by the integer tuple $(N^1_t, \ldots, N^n_t, L^1_t, \ldots, L^l_t)$. If this set of sequences satisfies the degree constraint, we know that node q must occur D^i_q times in sequence N^i, and similarly for the variable links. Let $\mathcal{S}_{\mathcal{I}}$ be the set of all such integer sequences satisfying the constraint. We follow the same logic as for the EL model. Let k be the number of matches of the pattern. We have:

$$|\mathcal{S}| = \binom{k}{D^1_1, \ldots, D^1_v} \times \ldots \times \binom{k}{D^w_1, \ldots, D^w_v} \times \binom{k}{C^1_1, \ldots, C^1_r} \times \ldots \times \binom{k}{C^l_1, \ldots, C^l_r}$$

As before, this set is larger than the set we are interested in. First, each set of pattern matches is contained multiple times (once for each permutation) and second, not all elements are valid pattern matches (in some, a single triple may be represented by multiple instances). Let $\mathcal{S}'_\mathcal{I}$ be the subset representing only valid matches, and let $\mathcal{G}_\mathcal{I}$ be the set of valid instances with permutations removed. As before, we have

$$|\mathcal{G}_\mathcal{I}| = |\mathcal{S}'_\mathcal{I}|\frac{1}{k!} \leq |\mathcal{S}_\mathcal{I}|\frac{1}{k!} .$$

Which gives us the following distribution

$$p_M(G) = \frac{k!}{|\mathcal{S}_D|} < \frac{1}{\mathcal{G}_D} ,$$

with $-\log p_M(\mathcal{I})$ as a code to store the instances. Rewriting as before, gives us a codelength of

$$-\log p_M(G) = (w + l - 1)\log(k!) - \sum_{j\in[1,w],i} \log(D^j_i!) - \sum_{j\in[1,l],i} \log(C^j_i!)$$

Note that if we store a graph with the pattern `?n1 ?rel ?n2` we obtain an empty template graph, and the motif code achieves the same codelength as the edgelist model, up to a small constant amount for storing the pattern.

For a given graph and pattern, we can simply find the complete list of instances using a graph pattern search. Since we require a slightly different semantics than standard graph pattern matchers, we adapt the DualIso algorithm [16] for knowledge graph matching. Before computing the motif code, we prune the list of instances provided by this search iterating over the instances and removing and instance that produces a triple also produced by an earlier instance. To guard against rare patterns that produce long-running searches we terminate all searches after 5 seconds, returning only those matches that were found within the time limit.

We express the strength of a motif by its log-factor:

$$-\log p^{\text{null}}(G) + \log p^{\text{motif}}(G; M, \mathcal{I}, L^{\text{base}}) .$$

If this value is positive, the motif code compresses the graph better than the null model. If the log-factor is greater than 10 bits, it corresponds to a rejection of the null model at $p < 0.001$.

2.3 Motif search

Ultimately, we want to find any motifs that have a high log-factor for a given graph G. Since we can readily compute the log-factor, any black-box optimization algorithm can be used to search the space of all possible motifs. For the sake of simplicity, we will use basic simulated annealing here: We start with a given pattern, and iterate by modifying the pattern in one of seven ways, chosen

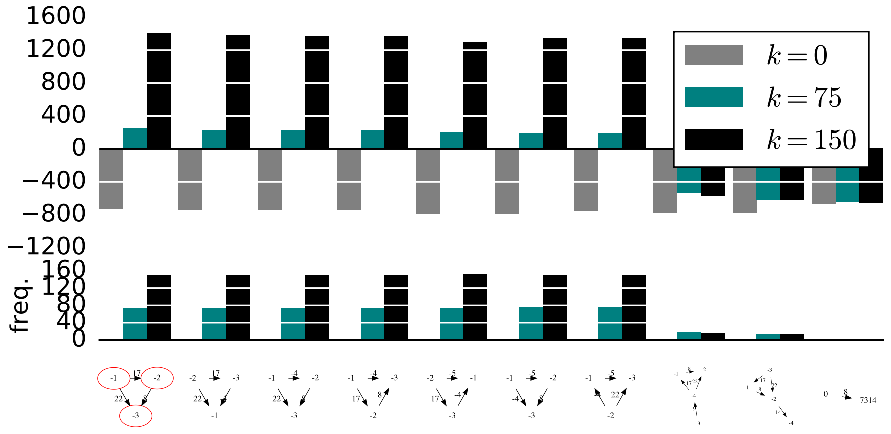

Fig. 2. The result of the random graph experiment. We sort the motifs by their score in the $k = 75$ experiment and plot their frequency and log-factor.

randomly. At each iteration, we search for instances of the pattern (limiting the time per search to 5 seconds) and compute the log-factor. If the log factor is better, we move to the new pattern, if it is worse, we move to the new pattern with probability 0.5.

The starting pattern is always a single random triple from the graph, with its relation made a variable. We define seven possible transition from one pattern to another:

Extend Choose an instance of the pattern and an adjacent triple not part of the instance. Add the triple to the pattern.

Make a node a variable Choose a random constant node, and turn it in to a variable node.

Make an edge a variable Choose a random constant edge label, and turn it in to a variable (always introducing a new variable).

Make a variable node constant Choose a random variable node and turn it into a constant. Take the value from a random instance.

Make a variable edge constant Choose a random variable edge and turn it into a constant. Take the value from a random instance.

Remove edge Remove a random edge from the pattern, ensuring that it stays connected.

Couple Take two distinct edge variables, which for at least one instance hold the same value and turn them into a single variable.

All transitions are equally likely. If the transition cannot be made (for instance, there are no constant nodes to make variable) or if the resultant pattern is in some way invalid, we sample a new transition.

We store all encountered patterns and their scores. In order to exploit all available processor cores, we run several searches in parallel. We take the top

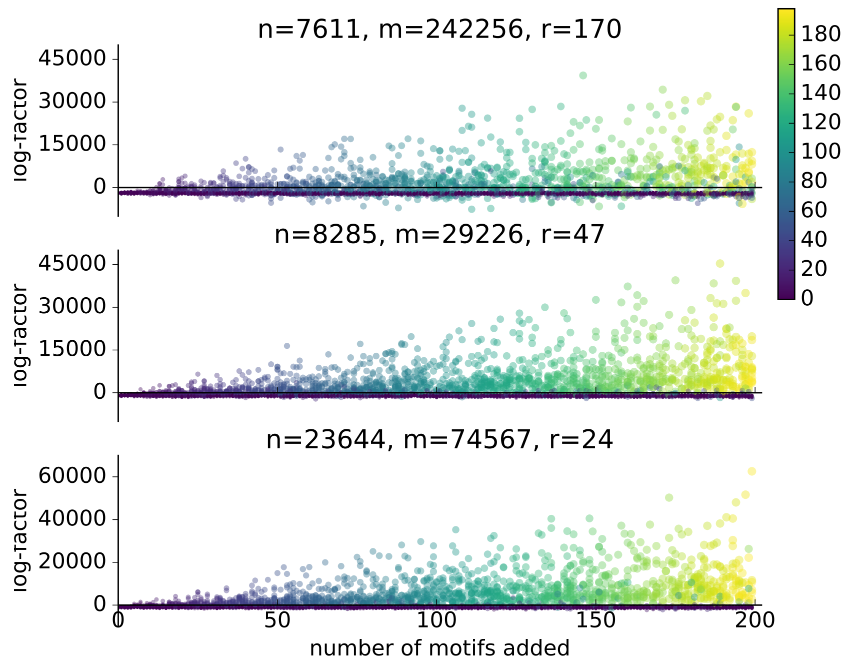

Fig. 3. The result of the repeated random graph experiment. Color and size show the number of matches of the pattern after pruning. Plot titles show the graph dimensions before adding instances.

1000 patterns from each and sort them by motif codelength. Variables are re-ordered to a canonical ordering using the Nauty algorithm [8], so that isomorphic patterns are not tested twice.

3 Experiments

Random graphs To validate the method, we first test it on random graphs. We sample a directed graph with a given number of nodes n and edges m, with no self-connections and multiple edges (that is, we sample from from the $G(n, m)$ Erdős-Renyi model). We then label the nodes uniformly at random with one of the relations in $0, \ldots, r$. To make the dimensions realistic we base them on those of the Mutag dataset used later: $n = 23644$, $m74567$, $r = 24$.

We then take one randomly chosen pattern, and insert $k = 75$ instances of the pattern into the graph. We run a search for 100000 iterations. And collect the 10 motifs with the best log-factor. We then sample two other graphs by the same method: one with $k = 0$ and one with $k = 150$. We also test each of the 10 motifs found on these two graphs.

The results are shown in Figure 2. For $k = 0$, as expected, we find no patterns with positive compression. We also ran a full search on this graph to verify that no motifs can be found unless they are explicitly added to the graph.

This experiment only tests a single pattern. To see the effect of multiple random patterns, we repeat the experiment many times, sampling both the pattern and the random graph.

To sample the pattern we first sample a random number of nodes n from $U(3, 6)$, the uniform distribution over the integer range $(3, 6)$ (including both end points). We then sample a random number of links m from $U(n, n^2 - n)$, and sample a random directed graph from $G(n, m)$. We make $U(0, n)$ nodes and $U(0, m)$ links into variables, choosing constants for the rest uniformly from the data. If the pattern is disconnected, we reject and sample again.

We sample a random graph as in the previous experiments, using the dimensions from the three real world datasets used later. We then add k instances of the motif to the graph and compute the log-factor of the sampled pattern (we do not conduct a search).

We let k range from 0 to 200, and repeat the experiment 25 times for each k, sampling a new graph and pattern each time. The results are shown in Figure 3. We observe first that under this ad-hoc sampling regime, we produce some patterns that create only very few instances in the graph, after overlapping instances are pruned. Since it is no surprise that these don't allow significant compression, we plot these as small points so that they don't obscure the other points.

We see that most of the other instances—those that generate enough non-overlapping instances—can be retrieved as motifs.

Real data Finally, we will test our method on real data, to confirm that the motifs found coincide with our intuition. We test three datasets: The Semantic Web dogfood dataset [10] ($n = 7611, m = 242256, r = 170$) describing researchers and publications in the Semantic Web domain, the AIFB dataset [1] ($n = 8285, m = 29226, r = 47$) describing the structure of the AIFB institute, and the Mutag RDF dataset[5] ($n = 23644, m = 74567, r = 24$), describing a set of carcinogenic and non-carcinogenic molecules both in structure and properties.

For all datasets, we run 32 parallel searches, with 3125 iterations per search. Table 1 reports the top 5 motifs by log-factor, and the top 3 motifs by frequency. We provide the top 100 motifs under both criteria online. [6]

The method provides many positives. To see that these are not just random noise, consider those patterns that have high frequency, but a negative log-factor. For instance, the most frequent pattern in the AIFB data describes entities having the same "year" property. Clearly, such a pattern can be matched often, and in many different ways, but it does not provide a satisfying explanation of the the structure of the graph.

Much of what the motif code picks up on is redundancy in the original data. For instance, in the AIFB data both the `swrs:publication` relation and its

[5] Originally distributed as an example dataset with the DL-Learner framework [7].
[6] `https://github.com/MaestroGraph/motive-rdf`

log-factor	frequency	
		Dogfood , top 5 by log-factor (> 100 positive)
361495.0	10475	?n1 dc:creator ?n2. ?n1 foaf:maker ?n2. ?n2 foaf:made ?n1.
244579.5	7674	?n1 dc:creator ?n2. ?n1 foaf:maker ?n2. ?n1 swrs:author ?n2.
220360.2	12138	?n1 foaf:maker ?n2. ?n2 foaf:made ?n1.
189627.3	9888	?n1 foaf:member ?n2. ?n2 swrs:affiliation ?n1.
187972.9	10475	?n1 dc:creator ?n2. ?n2 foaf:made ?n1.
		Dogfood, top 3 by frequency
-3076.2	134853	?n1 rdf:_1 ?n2. ?n1 rdf:_2 ?n4. ?n1 rdf:_3 ?n3.
-3435.0	116074	?n1 swc:heldBy ?n3. ?n1 swc:heldBy ?n2.
-2379.9	110461	?n1 rdf:type owl:Thing. ?n2 rdf:type owl:Thing.
		AIFB, top 5 by log-factor (> 100 positive)
79234.0	7549	?n1 ?p3 ?n2. ?n2 ?p4 ?n1.
61310.4	4154	?n1 swrs:publication ?n2. ?n2 ?p3 ?n1.
57641.1	3965	?n1 swrs:publication ?n2. ?n2 swrs:author ?n1.
57603.1	3965	?n1 swrs:author ?n2. ?n2 ?p3 ?n1.
33168.0	7930	?n1 swrs:publication ?n2. ?n2 rdf:type ?n3. ?n2 swrs:author ?n1.
		AIFB, top 3 by frequency
-908.2	181246	?n1 swrs:year ?n3. ?n2 swrs:year ?n3.
-1524.3	173059	?n1 swrs:publication ?n3. ?n1 swrs:publication ?n2.
-1667.9	103434	?n1 swrs:member ?n2. ?n3 ?p5 ?n1. ?n4 swrs:author ?n2.
		Mutag, top 5 by log-factor (87 positive)
178304.4	18634	?n1 mtg:_hasAtom ?n3. ?n1 mtg:_hasBond ?n2. ?n2 mtg:_inBond ?n3.
97237.8	9189	?n1 mtg:_hasAtom ?n2. ?n2 mtg:_charge ?n3.
93819.3	8924	?n2 rdf:type ?n3. ?n2 mtg:_charge ?n1.
90447.5	18634	?n1 mtg:_hasBond ?n2. ?n2 mtg:_inBond ?n4. ?n2 mtg:_inBond ?n3.
79027.5	8924	?n1 mtg:_hasAtom ?n2. ?n2 rdf:type ?n3.
		Mutag, top 3 by frequency
-2040.6	68514	?n1 rdfs:subClassOf ?n2. ?n3 rdf:type owl:Class. ?n4 rdf:type owl:Class. ?n4 rdfs:subClassOf ?n2.
-2077.8	60832	?n1 ?p5 owl:Class. ?n3 rdfs:subClassOf ?n2. ?n4 ?p5 owl:Class. ?n4 rdfs:subClassOf ?n2.
-1532.6	32009	?n1 mtg:_cytogen_sce "true". ?n1 mtg:_salmonella ?n3. ?n2 mtg:_amesTestPositive ?n3.

Table 1. Results of the experiment on real-world data. For each experiment we also report the number of motifs found with a positive log-factor.

log-factor	frequency		
220360.2	12138	`?n1 foaf:maker ?n2.` `?n2 foaf:made ?n1.`	Dogfood
3157.0	1011	`?n1 ?p2 "false".`	Mutag
3150.2	985	`?n1 ?p2 "true".`	Mutag
12871.8	8308	`?n1 rdf:type ?n2.` `?n1 swrs:year ?n3.` `?n4 swrs:publication ?n1.`	AIFB

Table 2. Selected motifs. The frequency is the number of matches found in the set time limit. The last column indicates the dataset.

inverse `swrs:author` are always included. Extracting these into a motif is simple way of achieving compression. In fact, the AIFB data caontains so many of these relation pairs that the two-node loop with variable labels is the highest scoring motif. In the Dogfood data, we see similar patterns emerge.

Table 2 shows some interesting motifs from the top 100 for each dataset. We see, for instance that the assertions that something is true or false are both motifs. While these are single triples with only one variable, they occur often enough, that encoding them separately provides a positive compression. The example from the AIFB date shows a typical ”star” pattern likely to emerge from relational data: a single entity, surrounded by a set of attributes.

4 Discussion

We have presented a new method for mining graph patterns from knowledge graphs. To our knowledge, this is the first method presented that can potentially find arbitrary basic graph patterns to describe the innate structure of a knowledge graph.

Limitations and future work Currently, the greatest limitation of this method is scalability. In [2], the original method on which this method is based was shown to scale to graphs with billions of links. However, this scalability does not translate directly to knowledge graphs, since the sampling approach used there is exteremely unlikely to generate the patterns that are succesful for knowledge graphs (requiring the use of simulated annealing istead). Since the motif test is just as scalable as it simple-graph equivalent, the only thing that is required to make the full motif-finding method scalable, is a faster search algorithm, perhaps based on an adaptation of the sampling approach from [2].

Our method currently produces a large number of motifs. We can show that worthwhile motifs are included, and that it performs better than a frequency baseline, but it still takes some manual effort to sort through the suggestions to find the kind of motifs that fit a particular use case. This is not surprising; it is the nature of knowledge graphs that many different and overlapping substructures can be seen as natural or meaningful. One promising avenue to reduce this manual effort is to search for a *set* of motifs which together compress well, each motif claiming a certain part of the knowledge graph to represent.

Acknowledgements This publication was supported by the Amsterdam Academic Alliance Data Science (AAA-DS) Program Award to the UvA and VU Universities.

References

1. Bloehdorn, S., Sure, Y.: Kernel methods for mining instance data in ontologies. In: The Semantic Web, pp. 58–71. Springer (2007)
2. Bloem, P., de Rooij, S.: Large-scale network motif learning with compression. arXiv preprint arXiv:1701.02026 (2017)
3. Bloem, P., de Rooij, S.: A tutorial on mdl hypothesis testing for graph analysis. arXiv preprint arXiv:1810.13163 (2018)
4. Cook, D.J., Holder, L.B.: Substructure discovery using minimum description length and background knowledge. CoRR **cs.AI/9402102** (1994), `http://arxiv.org/abs/cs.AI/9402102`
5. Cover, T.M., Thomas, J.A.: Elements of information theory (2. ed.). Wiley (2006)
6. Grünwald, P.: The minimum description length principle. The MIT Press (2007)
7. Lehmann, J.: Dl-learner: learning concepts in description logics. Journal of Machine Learning Research **10**(Nov), 2639–2642 (2009)
8. McKay, B.D., et al.: Practical graph isomorphism. Department of Computer Science, Vanderbilt University Tennessee, US (1981)
9. Milo, R., Shen-Orr, S., Itzkovitz, S., Kashtan, N., Chklovskii, D., Alon, U.: Network motifs: simple building blocks of complex networks. Science **298**(5594), 824–827 (2002)
10. Möller, K., Heath, T., Handschuh, S., Domingue, J.: Recipes for semantic web dog foodthe eswc and iswc metadata projects. In: The Semantic Web, pp. 802–815. Springer (2007)
11. Newman, M.: Networks: an introduction. Oxford University Press (2010)
12. Pham, M.D., Boncz, P.: Exploiting emergent schemas to make rdf systems more efficient. In: International Semantic Web Conference. pp. 463–479. Springer (2016)
13. Pham, M.D., Passing, L., Erling, O., Boncz, P.: Deriving an emergent relational schema from rdf data. In: Proceedings of the 24th International Conference on World Wide Web. pp. 864–874. International World Wide Web Conferences Steering Committee (2015)
14. Pitman, J., Yor, M.: The two-parameter poisson-dirichlet distribution derived from a stable subordinator. The Annals of Probability pp. 855–900 (1997)
15. de Rooij, S., Beek, W., Bloem, P., van Harmelen, F., Schlobach, S.: Are names meaningful? quantifying social meaning on the semantic web. In: International Semantic Web Conference. pp. 184–199. Springer (2016)
16. Saltz, M., Jain, A., Kothari, A., Fard, A., Miller, J.A., Ramaswamy, L.: Dualiso: An algorithm for subgraph pattern matching on very large labeled graphs. In: Big Data (BigData Congress), 2014 IEEE International Congress on. pp. 498–505. IEEE (2014)
17. Völker, J., Niepert, M.: Statistical schema induction. In: Extended Semantic Web Conference. pp. 124–138. Springer (2011)