Detecting Motifs in Knowledge Graphs using Compression

Author names withheld

Abstract

We introduce a method to detect *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*, commonly used in the knowledge graph literature. We introduce an approach, inspired by recent work for simple graphs, for inducing these for a given knowledge knowledge graph, and show that the motifs returned, 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, even if as few as 10 instances are inserted. Finally, we show the results of motif induction on three real-world knowledge graphs, and show that our method provides good suggestions for motifs that reflect the structure of the graph in an intuitive way.

Knowledge graphs are an extremely versatile way of storing knowledge. They allow knowledge to be encoded without a predefined schema, they allow different amounts of data to be encoded for different entities and they allow different modalities to be integrated seamlessly (Wilcke, Bloem, and De Boer 2017). 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* (Milo et al. 2002) 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, (Bloem and de Rooij 2017) 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 the 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 are non-uniquely labeled with relations. We extend the definition of a motif to that of a *basic graph pattern*: that is, a motif is a small graph with *some* of its nodes and links labeled with entities and relations as they occur in the graph. Other nodes and links are labeled as *variables*. A pattern matches at a particular location in the graph, if the variables can be replaced with values from the graph, and the motif becomes a subgraph as a result. The intuition behind our method is that the more compactly a pattern compresses the graph, the better the motif. Figure 1 shows an illustration of the basic principle. In the preliminaries section below, 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, sometimes with as few as 10 instances inserted into the graph. Finally, we show the results of motif analysis on three real-world knowledge graphs.

All code and datasets used in this paper are available, under open licenses.[1]

Related Work Network motifs for unlabeled simple graphs were introduced in (Milo et al. 2002). A more comprehensive overview of the related literature can be found in (Bloem and de Rooij 2017, Section 1.1). In (Bloem and de Rooij 2017), 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 (Cook and Holder 1994).

A few other methods have been proposed for inducing the structure of a given knowledge graph. In (Pham et al. 2015), the authors use the principle of characteristic sets to characterize a knowledge graph in terms of the star patterns it contains. In (Pham and Boncz 2016), they show that the majority of the LOD cloud can be efficiently described using such principles, showing the highly tabular structure of many knowledge graphs. In (Völker and Niepert 2011), association rules mining is used to induce basic patterns in the

[1] URL withheld for purposes of blind reviewing.

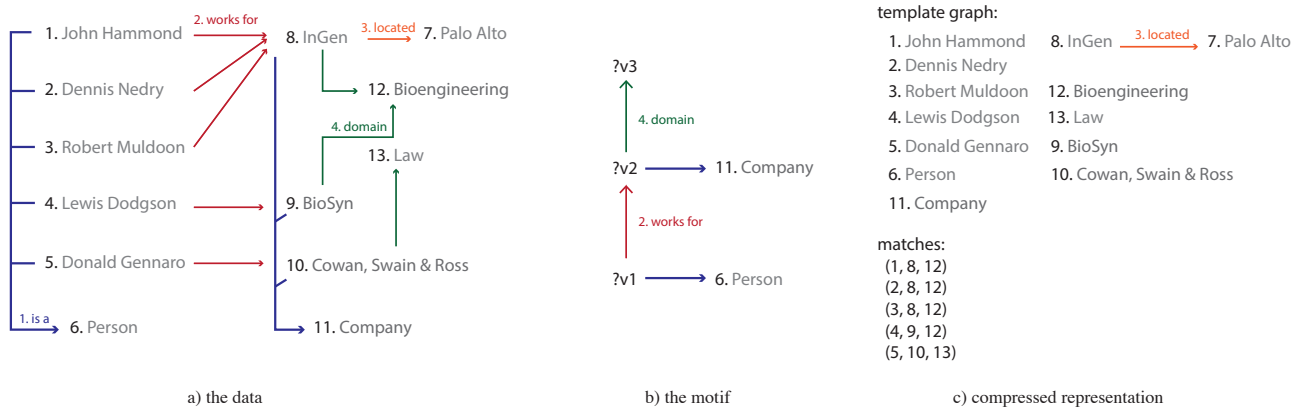

Figure 1: An example of the principle behind our motif code. a) A basic knowledge graph. Note that in our analysis, we consider only the indices of the nodes and relations, not their labels. b) A motif that occurs frequently. c) A compressed representation; we remove all edges that are part of an occurrence of the motif. We then store separately which nodes match the motif. The motif, together with the template graph and the matches can be used to reconstruct the graph. We take this as a heuristic: the better the motif compresses, the more characteristic it is likely to be for the graph.

graph.

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

Preliminaries

Minimum Description Length Our method is based on the MDL principle: that 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 (Grünwald 2007) for MDL in general, and to (Bloem and de Rooij 2017), 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 $\mathcal{X}$ (such as graphs) is an injective function $f : \mathcal{X} \to \mathbb{B}$, mapping objects to binary code words. 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 $f(x)$ and we refer to $L(x)$ as a code.

A well known result in information theory is the association between codes and probability distributions, implied by the *Kraft inequality*: 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 (Grünwald 2007, Section 3.2.1) or (Cover and Thomas 2006, Theorem 5.2.1). To explain the intuition, note that we can easily transform a code L into a sampling algorithm for p by feeding the decoding function random bits until it produces an output. To transform a probability distribution to a code, techniques like arithmetic coding (Rissanen and Langdon 1979) can be used.[2]

Relevance testing We will use the MDL principle to perform a hypothesis test. Assume we have some data x and a null hypothesis that it was sampled from distribution p^{null}. Let L^{null} be the corresponding codelength. A simple but crucial result, known as the *no-hypercompression inequality* (Grünwald 2007, p103) tells us that the probability of sampling data x from p^{null} that can be described shorter than $L^{\text{null}}(x)$ by k or more bits, *using any code* is less than 2^{-k}. Thus, an MDL hypothesis test would consist of stating the null hypothesis in terms of a code L^{null}, designing a code (before seeing the data) 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 (Bloem and de Rooij 2017, Section 2).

Note that when we use this procedure to find motifs, we are not providing evidence for the hypothesis that the motif is "correct" (Bloem and de Rooij 2017, Section 2).

Common codes In the construction of our codes for graphs, 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, 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. These will be highly skewed, with only a subset of integers occur-

[2] As explained in (Grünwald 2007, page 96), the fact that $-\log p^*(x)$ is real-valued and $L^*(x)$ is integer-valued can be safely ignored and we may *identify* codes with probability distributions, allowing codes to take non-integer values.

ring frequently, and others occurring infrequently or not at all. As noted in (de Rooij et al. 2016) a code based on the Pitman-Yor model (Pitman and Yor 1997) 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 first occur: we first store n and the first member using the codes described above. The rest of the members are stored sequentially using the distance from the current to the next member.

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:

$$p^{\text{PY}}_{\alpha,d}(S) = \prod_{i \in [1,k]} \text{PY}_{\alpha,d}(S_i \mid S_{1:i-1})$$

$$\text{PY}_{\alpha,d}(S_i \mid S') = \begin{cases} \frac{\alpha - d|m(S)|}{|m(S)| + \alpha} & \text{if} f(S_i, S') = 0 \\ \frac{f(S_i,S') - d}{|m(S)| + \alpha} & \text{otherwise} \end{cases}$$

See (de Rooij et al. 2016) for a more intuitive explanation. In all experiments we use $\alpha = 0.5$, $d = 0.1$. We will refer to the total codelength for a sequence under this procedure as $L^{PY}(S)$

Method

To comfortably define probabilities and codes on graphs, we will slightly deviate from conventional definitions. Specifically, we will analyse the *structure* of knowledge graphs only, ignoring any meaning they have outside the graph. Under this assumption we can assume that the node and link labels of our knowledge graphs are simply natural numbers.[3] This leads to the following definition.

A *knowledge graph* G, hereafter simply a *graph*, 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 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.

As shown in Figure 1, the patterns that we aim to find are partially labeled: some nodes and edges are labeled with as variables. We will represent these with negative integers. 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 $E_M \subset V_M \times R_M \times V_M$, as for regular graphs. 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 negative integers are always contiguous within a single graph, with the highest representing the node labels and the lowest representing the edge labels.

[3]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.

log-factor	frequency	
Dogfood , top 5 by log-factor (> 100 positive)		
83119.1	9344	?n1 dc:creator ?n2. ?n2 rdfs:label ?n3. ?n2 foaf:name ?n3.
81075.1	10322	?n1 foaf:maker ?n2. ?n2 rdfs:label ?n3. ?n2 foaf:name ?n3.
77692.6	5234	?n2 rdf:type ?n3. ?n2 rdfs:label ?n1. ?n2 foaf:name ?n1.
75632.3	3536	?n1 swc:isPartOf ?n2. ?n2 swc:hasPart ?n1.
75592.4	3536	?n1 swc:isPartOf ?n2. ?n2 ?p3 ?n1.
Dogfood, top 3 by frequency		
-3032.5	137008	?n1 rdf:_1 ?n2. ?n1 rdf:_2 ?n4. ?n1 rdf:_3 ?n3.
-3251.4	109315	?n1 swc:isPartOf ?n3. ?n2 swc:isPartOf ?n3.
2089.0	92165	?n2 swc:isPartOf ?n3. ?n3 swc:hasPart ?n1.
AIFB, top 5 by log-factor (> 100 positive)		
79234.0	7728	?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.
x		?n1 swrs:publication ?n3.
42110.1	17512	?n3 swrs:author ?n2. ?n3 swrs:author ?n1. ?n2 swrs:publication ?n3.
AIFB, top 3 by frequency		
2287.7	198668	?n1 swrs:publication ?n3. ?n1 swrs:publication ?n2.
7822.1	191083	?n1 swrs:author ?n2. ?n2 swrs:publication ?n3.
1264.0	190636	?n1 swrs:isAbout ?n3. ?n2 swrs:isAbout ?n3.
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.
90447.5	18634	?n1 mtg:_hasBond ?n2. ?n2 mtg:_inBond ?n4. ?n2 mtg:_inBond ?n3.
79052.0	9317	?n1 mtg:_hasBond ?n2. ?n2 rdf:type ?n3.
58973.9	7892	?n1 mtg:_hasAtom ?n2. ?n2 rdf:type ?n3.
Mutag, top 3 by frequency		
1025.7	341143	?n2 mtg:_hasAtom ?n3. ?n2 mtg:_hasBond ?n1.
-374.5	178837	?n1 mtg:_charge ?n3. ?n2 mtg:_charge ?n3.
-2088.0	52985	?n1 rdf:type owl:Class. ?n3 rdfs:subClassOf ?n2. ?n4 ?p5 owl:Class. ?n4 rdfs:subClassOf ?n2.

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.

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 $(s, p, o) \in E_M$ containing a negative s, p or o, 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.

We will first assume that a target pattern M is given for the data G and that we have a set of instances $\mathcal{I}$. Moreover, we require that all instances in $\mathcal{I}$ are mutually disjoint: no two subgraphs defined by member of $\mathcal{I}$ may share an edge, but nodes may be shared.

Null model The most common null model for classical motif analysis is the degree-sequence model (also known a the configuration model (Newman 2010)): 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, 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_i^{in} incoming links, D_i^{out} outgoing links and for each relation r, there are D_r^{rel} triples.

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

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

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 (Bloem and de Rooij 2017), 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_j^{out} times, P should contain relation r D_r^{rel} times and O should contain node j D_j^{in} times. Let $\mathcal{S}_D$ be the set of all such triples of integer sequences satisfying D. We have

$$|\mathcal{S}_D| = \binom{m}{D_1^{\text{out}}, \ldots, D_n^{\text{out}}} \binom{m}{D_1^{\text{rel}}, \ldots, D_{|R_G|}^{\text{rel}}} \binom{m}{D_1^{\text{in}}, \ldots, D_n^{\text{in}}}.$$

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

$$p_D^{\text{EL}}(G) = \frac{m!}{\binom{m}{D_1^{\text{out}}, \ldots, D_n^{\text{out}}} \binom{m}{D_1^{\text{rel}}, \ldots, D_{|R_G|}^{\text{rel}}} \binom{m}{D_1^{\text{in}}, \ldots, D_n^{\text{in}}}} \leq p^C(G).$$

log-factor	frequency		
220360.2	12138	`?n1 foaf:maker ?n2.` `?n2 foaf:made ?n1.`	D
3157.0	1011	`?n1 ?p2 "false".`	M
3150.2	985	`?n1 ?p2 "true".`	M

12871.8	8308	`?n1 rdf:type ?n2.`
		`?n1 swrs:year ?n3.`
		`?n4 swrs:publication ?n1.`
			A

Table 2: Selected motifs.The last column indicates the dataset.

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

$$-\log p_D^{\text{EL}}(G) = 2\log(m!) - \sum_i \log(D_i^{\text{in}}!) - \sum_i \log(D_i^{\text{rel}}!) - \sum_i \log(D_i^{\text{out}}!)$$

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.[4] Experiments in the classical motif setting have shown that the EL model is an acceptable proxy for the DS model (Bloem and de Rooij 2017), especially considering the extra scalability it affords.

Encoding D In order to encode a graph with L_D^{EL}, we must first encode D.[5] For each of the three sequences 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.

In the design of our method, we will constantly aim to find a trade-off between completeness and efficiency that allows the method to scale to very large graphs. Specifically, when we economize, we will only do so in a way that makes the hypothesis test *more conservative*.

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

[4]Note that we cannot simply think of p^{EL} as a uniform model for graphs containing multiple links, since we can only divide by $m!$ if we know that all triples are unique.

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

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^{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 $b_{\text{dim}} + b_{\text{pattern}} + b_{\text{template}} + b_{\text{instances}}$

method, the more motifs we will find. Note that there is no need for our code to be optimal in any sense. So long as we accept that we won't find all motifs that exist, 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 (or rather, sum the codelengths we get for each aspect).

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 understand the following ,it may be instructive to think of a single triple as an instance for the pattern `?n1 ?rel ?n2`. We can think of the edgelist model as storing a sequence of instances for this pattern, which together make up the entire knowledge graph.

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 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 degree constraint. We will follow the same logic as for the EdgeList 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

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

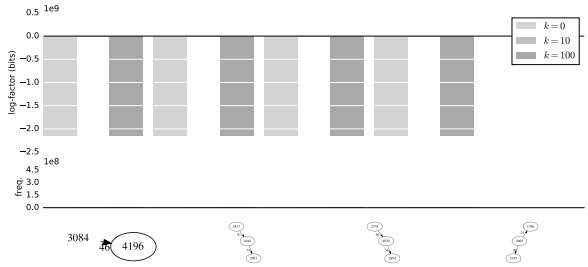

Figure 2: The result of the random graph experiment. We sort the motifs by their score in the $k = 10$ experiment and plot their frequency and log-factor.

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 (Saltz et al. 2014) 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 $p < 0.001$.

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. 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 (2) 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 (3) Choose a random constant node, and turn it in to a variable node.

Make an edge a variable (3) Choose a random constant edge label, and turn it in to a variable (always introducing a new variable).

Make a variable node constant (2) Choose a random variable node and turn it into a constant. Take the value from a random instance.

Make a variable edge constant (2) Choose a random variable edge and turn it into a constant. Take the value from a random instance.

Remove edge (3) Remove a random edge from the pattern, ensuring that it stays connected.

Couple (1) Take two distinct edge variables, which for at least one instance hold the same value and turn them into a single variable.

The values in brackets are the relative weights of each transition type (chosen by trial and error). We choose a transition at random using the normalized weights as probabilities and compute a new pattern. If the transition cannot be made (for instance, there are no constant node to make variable) or if the resultant pattern is in some way invalid, we sample a new transition.

Once a new pattern has been sampled, we compare its codelength under the motif model to that of the previous sample. If the codelength is lower, we continue with the new pattern. If the codelength is longer, we continue with the new sample with probability α or return to the previous pattern otherwise. We use $\alpha = 0.5$ in all experiments.

We store all patterns seen and their scores. In order to exploit all available processor cores, we run several searches in parallel. After each the searches are finished, we take the top 1000 patterns from each and sort them by motif codelength.

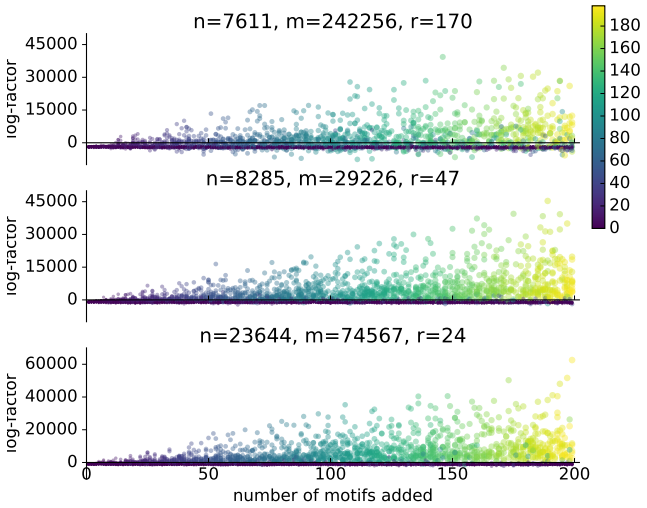

Figure 3: The result of the repeated random graph experiment. Color and size show the number of matches of the pattern after pruning. Plot titles show the graph dimensions before adding instances.

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 instances of the pattern into the graph. We run the experiment for $k = 0$, $k = 75$ and $k = 150$. The results are shown in Figure 2. For $k = 0$, as expected, we find no patterns with positive compression. For $k = 75$, we find ...such patterns and for $k = 150$ we find ... . While we only inserted one pattern we can expect to find more than one, since its subpatterns may also become motifs. However, the pattern we inserted should receive the highest log-factor.

Of course, 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. As the number of added instances increases, the proportion of motifs with separate instance that cannot be retrieved diminishes.

Real-world data Finally, we will test our method on real-world dat, to confirm that the motifs found coincide with our intuition. We test three datasets: The semantic Web dogfood dataset (Möller et al. 2007) ($n = 7611$,$m = 242256$,$r = 170$) describing researchers and publications in the Semantic Web domain, the AIFB dataset (Bloehdorn and Sure 2007) ($n = 8285$,$m = 29226$,$r = 47$) describing the structure of the AIFB institute, and the Mutag RDF dataset [6]($n = 23644$,$m = 74567$,$r = 24$), 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. For each dataset we report the top 5 motifs by log-factor, and the top 3 motifs by frequency. Note that these are the top 10 motifs for frequency in the set of 32000 high-log-factor motifs resulting from the parallel search. The supplemental materials show the top 100 motifs for both criteria.

The patterns with negative score but high frequency show the pitfalls our code avoids. For instance, in the top motif by frequency for the Dogfood dataset, we see a four-node pattern with two edges fanning in to a single node. These are caused by a single node having the same ingoing relation to many other nodes. Clearly such a pattern is very frequent in the data, but capturing just three of these incoming links does not constitute a satisfying motif, and indeed such patterns receive a negative log-factor.

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 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 e=the assertions that something is true or false are both motifs. The example from the AIFB date shows

Discussion

We have presented a new method for inducing graph patterns from knowledge graphs. To our knowledge, this is the first method presented that can potentially find arbitrary 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 (Bloem and de Rooij 2017), the original method on which this method is based was shown to scale to graphs with billions of links. However, this scalability did not translate directly to knowledge graphs, since the sampling approach, used there is exteremely unlikely to generate the patterns that are succesful for motifs. If a better sampling method could be divised, we could likely scale the method up to web-scale knowledge graphs.

Our method currently produces a large number of motifs. We can show that worthwhile motifs are included, and that it performs better than a baseline like selecting by frequency, but it still takes some manual effort to sort through the suggestions to find the kind of motifs that fit a particular usecase. 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 of a *set* of motifs which together compress well, each motif claiming a certain part of the knowledge graph to represent. If such an approach were fruitful it would truly represent a versatile and powerful form of schema induction for knowledge graphs.

[6]Distributed as an example dataset with the DL-Learner framework. (Lehmann 2009)

References

Bloehdorn, S., and Sure, Y. 2007. Kernel methods for mining instance data in ontologies. In *The Semantic Web*. Springer. 58–71.

Bloem, P., and de Rooij, S. 2017. Large-scale network motif learning with compression. *arXiv preprint arXiv:1701.02026*.

Cook, D. J., and Holder, L. B. 1994. Substructure discovery using minimum description length and background knowledge. *CoRR* cs.AI/9402102.

Cover, T. M., and Thomas, J. A. 2006. *Elements of information theory (2. ed.)*. Wiley.

de Rooij, S.; Beek, W.; Bloem, P.; van Harmelen, F.; and Schlobach, S. 2016. Are names meaningful? quantifying social meaning on the semantic web. In *International Semantic Web Conference*, 184–199. Springer.

Grünwald, P. 2007. *The minimum description length principle*. The MIT Press.

Lehmann, J. 2009. Dl-learner: learning concepts in description logics. *Journal of Machine Learning Research* 10(Nov):2639–2642.

Milo, R.; Shen-Orr, S.; Itzkovitz, S.; Kashtan, N.; Chklovskii, D.; and Alon, U. 2002. Network motifs: simple building blocks of complex networks. *Science* 298(5594):824–827.

Möller, K.; Heath, T.; Handschuh, S.; and Domingue, J. 2007. Recipes for semantic web dog foodthe eswc and iswc metadata projects. In *The Semantic Web*. Springer. 802–815.

Newman, M. 2010. *Networks: an introduction*. Oxford University Press.

Pham, M.-D., and Boncz, P. 2016. Exploiting emergent schemas to make rdf systems more efficient. In *International Semantic Web Conference*, 463–479. Springer.

Pham, M.-D.; Passing, L.; Erling, O.; and Boncz, P. 2015. Deriving an emergent relational schema from rdf data. In *Proceedings of the 24th International Conference on World Wide Web*, 864–874. International World Wide Web Conferences Steering Committee.

Pitman, J., and Yor, M. 1997. The two-parameter poisson-dirichlet distribution derived from a stable subordinator. *The Annals of Probability* 855–900.

Rissanen, J., and Langdon, G. G. 1979. Arithmetic coding. *IBM Journal of research and development* 23(2):149–162.

Saltz, M.; Jain, A.; Kothari, A.; Fard, A.; Miller, J. A.; and Ramaswamy, L. 2014. Dualiso: An algorithm for subgraph pattern matching on very large labeled graphs. In *Big Data (BigData Congress), 2014 IEEE International Congress on*, 498–505. IEEE.

Völker, J., and Niepert, M. 2011. Statistical schema induction. In *Extended Semantic Web Conference*, 124–138. Springer.

Wilcke, X.; Bloem, P.; and De Boer, V. 2017. The knowledge graph as the default data model for learning on heterogeneous knowledge. *Data Science* (Preprint):1–19.