Detecting Network Motifs in Knowledge Graphs using Compression

July 28, 2017

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 introduce a scalable approach for detecting such motifs in large knowledge graphs, inspired by recent work for simple graphs, and show that the motifs returned reflect the basic structure of the graph. Specifically, we show that common motifs reflect graph patterns used in common queries, basic schematic units of the graph and meaningful functional subunits, for various knowledge graphs.

The Linked, Open Data cloud contains a wealth of knowledge graphs, and is growing quickly. At the time of writing the average knowledge graph contains over 65 000 edges, with many datasets containing more than a million.[1] For such large graphs, 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 objects? What are small, recurring patterns that might represent a novel insight into the data?

In the domain of plain (unlabelled) graphs, *network motifs* [**?**] were recently introduced as a tool to provide users of large graphs insight into their data. Network motifs are small subgraphs whose frequency is unexpected with respect to a null model. That is, for a given graph dataset g and small graph m, we count the frequency $F(m, g)$: how often m occurs in g as a subgraph. We define a probability distribution over graphs $p^{\text{null}}(g)$, and estimate the probability that a graph sampled from p^{null} contains more instances of m than observed in our data: $p^{\text{null}}(F(m, G) \geq F(m, g))$, where G is a random variable representing a graph. If this probability is low (commonly, below 0.05), we consider m a motif. [2]

[1] `http://stats.lod2.eu/stats`

Unfortunately, counting all occurrences of a motif in a network is an expensive operation. One that we need to perform not only on the data, but also on many samples from the null model, in order to estimate the probability described above.

In [**?**], an alternative method is presented that uses compression as a heuristic for motif value: the better a motif compresses the data, the more likely it is to be meaningful. It is shown that this principle can be implemented in a similar sort of hypothesis test for a null model, allowing a comparable workflow to classical motif analysis. Crucially, this method scales to graphs with billions of edges, suggesting that it might be suitable for the scale of knowledge graphs we encounter in the LOD cloud.

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 multigraphs. Nodes are labeled with entity names, and links are 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. The motif *occurs* in a graph where both the graph structure and the labels match. The unlabeled nodes and links are free to contain whatever label when the motif is mapped to the data.

To maintain the connection to graph pattern queries, we do not require the motifs to be *induced* subgraphs: that is, when a motif is mapped to a graph, the graph can contain additional links that are not specified by the motif.

We perform several experiments to show that our method returns meaningful subgraphs. First, we look at several graphs for which schema information is provided. That is, we know the basic properties available for each entity. We show that the motifs found by our method correspond well to the schema, and compare performance against several basic benchmarks. Second, we run motif analysis on several datasets for which a list of real SPARQL queries entered by users is available. From these, we extract the basic graph patterns, and show that (a) these correspond to good motifs using our criterion and (b) our method can return such motifs with good precision and recall.

All code and benchmark datasets used in the paper are available, under

[2] This procedure has the structure of a hypothesis test, but it is important not to interpret it as statistical evidence for the meaningfulness of the motif. The only thing it proves (in a frequentist statistical sense) is that p^{null} is not the true source of the data. This is usually not a surprise: we are rarely able to model all aspects of a realistic data-generating process in a single distribution. The p-values used in motif analysis should be interpreted strictly as *heuristics*. See also [**?**].

open licenses.[3]

0.1 Related Work

0.2 Preliminaries

1 Method

We will start with some formal definitions of our basic ingredients. To comfortably define probabilities and codes on graphs, we will slightly deviate from conventional definitions. Let V be a countably infinite set containing all possible entities, and let R be a countably infinite set containing all possible relations between them. Let $v : V \to \mathbb{N}$ and $r : R \to \mathbb{N}$ be arbitrary mappings of these sets to the natural numbers.[4]

A *knowledge graph* G, hereafter known simply as a *graph* is a, is tuple $G = (V_G, E_G)$ with finite sets $V_G \subset V$ and $E_G \subset V \times R \times V$. We define the triple list of G as $G^T = \{(s, p, o) \in \mathbb{N}^3 : (v(s), r(p), v(o)) \in E_G\}$. In this paper we will *identify* the graph with the triple list. That is when we speak of encoding the graph, we mean encoding the triple list, and when we speak of a probability distribution over graphs, we mean a probability distribution over triple lists. Let $R_G\mathbb{N}$ be the set of relations used in the G.

This means that we are purposely *not* modelling certain aspects of the data. There might, for instance be a similarity between two node labels $v(a)$ and $v(b)$ (such as a shared IRI prefix). We are not interested in exploiting such structure, so we eliminate it from our notation. Additionally, we introduce some bias with our choice of e and r. If we were to use a simple code to store the integers in the triple-list we will end up using fewer bits for those nodes and triples with labels who were assigned short integers. Equivalently, when sampling from a simple distribution over the integers, small numbers will be more likely to be generated. In practice, however, we will always *learn* a distribution over the integers from the data, so that e and r are purely a means to simplify the notation. Which functions we choose will not affect our model.

As shown in Figure **??**, the motifs that we aim to find are partially labeled: some nodes and edges are labeled with the marker "?" indicating that when we fit the motif to the graph, those edges can contain any label. In other words, the motif is a *basic graph pattern*, and the edges and nodes

[3] ...

[4] For instance, let V and E be the set of finite strings, and let v and e represent length-lexicographical order.

marked with "?" are variables. To represent this, we add a single variable symbol ?. Thus, a motif M is defined by the triple set $M \subset (? \cup \mathbb{N})^3$, with integer values in triples interpreted as they are in complete knowledge graphs.

We will first assume that a target pattern M is given for the data G and that we have a set of *instances* $\mathcal{I}'$ where $I \in \mathcal{I}'$ is a subset of E_G which matches the pattern. Moverover, we require that all instances in $\mathcal{I}$ are mutually disjoint. Thus, we cannot simply return all matches of the pattern, we require some selection to ensure that none of them overlap. In most settings we do not have a given M and I and we will need to search for them. We describe a simple and fast search algorithm in Section 1.3.

1.1 Relevance test

We will first describe our method under the assumption that a good null model p^{null} is available, and that $-\log p^{\text{null}}(G)$ can be efficiently computed. In Section 2, we explain which null model we use for our experiments, and how it is implemented.

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 described in Section 1.1 *more conservative.*

1.2 Motif code

As described above, we can perform our relevance test with any compression method which exploits the pattern M, and its instances 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. 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 sum the codelengths we get for each aspect to get the codelength for the complete graph.

We also assume that we are given a code L^{base} for generic knowledge graphs.

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 pattern M using L^{base}. We increment each of the pattern's existing link labels by 1, and use the label 0 to represent unlabeled links. Since L^{base} does not store node labels (V_M is always the set of the first $|V_M|$ natural numbers) we need to store separately to which nodes in G the labeled nodes of M refer. We store a sequence of integers g_i where $g_i = 0$ indicates that the i-th node of M is unlabeled, and $g_i \neq 0$ indicates that g_i should map to node $g_i - 1$ in G.

the template This is the graph, minus all links occurring in instances. 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^{base}.

the instances In order to reconstruct the graph from the template, we need to know which nodes correspond to instances of the motif, and in what order. For each unlabeled node in M, we store one list of length $\mathcal{I}$ of the nodes in G to which it maps for each instance. We store these lists using the DM code, since it is likely that a particular subset of the nodes of G is frequent for a particular unlabeled node of a pattern.

the edge labels We now have enough information to reconstruct the graph structure. All that remains to fully reconstruct G is to label the unlabeled link of the motif instances. As for the nodes, we create one list of length $\mathcal{I}$ for each unlabeled link in the pattern which we store using the DM code.

We can note a few aspects of this code. Firstly, it achieves its compression from not having to store the edges of the motif instances. For each additional instance, it pays the cost of storing the instance, and any unlabeled nodes and links, but it saves E_M edges. The precise trade-off depends on the choice of base code, and on the structure of the rest of the graph (some edges are cheaper to store than others).

Nevertheless, we can make a few broad observations: patterns with many unlabeled nodes and links are more costly to store than fully labeled patterns. On the other hand, a pattern with many unlabeled nodes and links is more likely to have many instances, so that *if* it results in a small amount compression per instance, the large number of instances may cause it to compress a lot. For the unlabeled nodes and links, we use a DM model for each specific node and link. This means that the more structure there is to the label given to a particular part of the pattern, the better we compress. For instance, suppose a particular node of the pattern is always labeled with either a node corresponding to YES or NO, in equal proportion, we will require only a bit per instance to record those labels. If all nodes in the graph are equally likely to appear in that position, we will pay approximately $\log |V_G|$ bits per instance. If only a single node appears in the position, we pay only a a constant number of bits to record the entire sequence of instances. [5]

[6]

1.3 Motif search

In most use cases, we do not start with a given motif and a set of instances. We have only the graph G, and we wish to find any motifs that might be interesting. In order to do this efficiently, we generate candidate patterns by *sampling* them from the data. This method, inspired by [**?**] is based on the simple insight that when we sample a pattern from the data, we are more likely to sample a frequently occurring pattern than a infrequent one. In we sample K times, and rank the results by relative frequency within the sample, we are likely to end up with a good indication of the most frequent patterns in the data. For unlabeled graphs, as little as 500 samples can be sufficient to find the most frequent subgraph.

Theoretically, this allows us to generate promising motifs with time complexity independent of the size of the graph. Our algorithm samples a pattern in three steps. First, it samples a weakly connected subset of n nodes, and extracts the subgraph induced by these nodes. Second, it converts this subgraph to a non-induced pattern by randomly removing labels from some of the nodes and links and randomly removing some links entirely (as detailed below). Finally, we sort the nodes of this pattern into a canonical ordering, using the Nauty algorithm [**?**].

For a subgraph with n nodes and e links, we can generate n^2e^3 patterns. A uniform random choice among these would make patterns with many labels very unlikely. Instead, we sample n' $U([0..n])$, e' $U([0..e])$ and e^* $U([0..e'])$: the number of nodes to leave labeled, the number of nodes links to leave, and the number of links to leave labeled respectively. If the resulting pattern is disconnected, we reject the sample.

[6] In fact, we could simplify the model by keeping the pattern entirely unlabeled, and trusting that the DM code will store everything efficiently. However, for our search algorithm, and for the interpretation of the motif as a graph pattern, we prefer to distinguish between labeled and unlabeled elements explicitly.

We sample K such patterns, recording a dictionary mapping canonical patterns M to a list of observed instances $\mathcal{I}'_M$ of M in G. Note that we must store the *link* of the occurrence in $\mathcal{I}'_M$, since the same pattern may emerge from a single induced subgraph in multiple ways.

Before we can can pass each pattern to the relevance test to see if it is a motif, we must first ensure that $\mathcal{I}'_M$ contains only link-disjoint instances. Assuming that all instances contribute equally to the compression, we would like to select the largest subset of $\mathcal{I}'_M$ consisting entirely of mutually disjoint sets. This is an instance of the Maximum Independent Set Problem, which is NP-Hard to solve optimally. Since we are aiming for scalability, we use a simple greedy algorithm to create our set $\mathcal{I}_M$ of disjoint instances: we include the first instance I in $\mathcal{I}'_M$ in $\mathcal{I}_M$ and exclude any other instances that overlap with I. We then move to the next instance that has not already been included or excluded, and repeat the procedure. We continue until all instances have been included or excluded.

We then pass each M and $\mathcal{I}_M$ to the relevance test to see if they are motifs. For experiments reported here, we check only the 100 patterns with the largest number of instances.[7][8]In [**?**], the number of edges between nodes inside the instance and outside it (the exdegree of the instance), was a strong predictor of whether the instances would benefit compression. Thus, the algorithm was strongly helped by pruning the list of instances using a ternary search. For the current code, the exdegree does not affect compression (since the instance nodes are not removed from the template graph). Therefore, we pass the entire instance list $\mathcal{I}_M$ to the relevance test wihtout pruning it.

Our sampling algorithm is described more precisely in Figure 1.

The sampling algorithm produces full labeled, induced subgraphs.

For instance, we will use a very fast search for candidate motifs, which returns only a limited number of instances. A more expensive search might yield more instances, but never fewer. That is, for those motifs that we find, we can be sure that they would also be found by a more expensive method. Since pattern mining of this kind often results in too many potential patterns rather than too few, it seems efficient to start with the motifs that can be found with fast, simple methods, and only extend the search once these have been exhausted.

[7]

```
Given: a graph $G$,
    a minimum motif size $s_{\text{min}}$ and a maximum motif size $s_{\text{max}}$
    a target number of graphs sampled

Initialize $d$, a dictionary from motif $M$ to a list of instances $d[M] \in 2^{2^N}$
Choose $s$ uniformly from $[s_{\text{min}}..s_{\text{max}}]$
Choose a starting node $n$ uniformly from $G$
$N \leftarrow \{n\}$
while $|N| < s$:
    Choose $n$ uniformly from $N$
    Choose $n'$ uniformly from the neighbors of $n$
    $N \leftarrow N \cup n'$
    After 500 cycles, sample a different starting node
$M \leftarrow G[N]$, the induced subgraph of $N$
$M \leftarrow$ canonical($M$) using the Nauty canonization algorithm, ignoring label
```

Figure 1: The motif sampling algorithm.

2 Null model

The most common null model for classical motif analysis is the degree-sequence model (also known a the configuration model []): a uniform distribution over all graphs that share the same in and out degrees of the data for every node. We extend this to knowledge graphs by also including the frequency with which each relation occurs. 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 links.

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

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

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 [**?**], we define an approximation. We can define a knowledge graph by three length-m integer sequences: S, P, O, with $(S_j, P_j, O_j)_j$ 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 $\mathcal{S}_D$ be the set of all such triples of integer sequences satisfying D. We have

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

While every member of $\mathcal{S}_D$ contains a valid graph satisfying D, many graphs are represented multiple times. Firstly, many elements of S_D contain the same link multiple times, which means they are not is $\mathcal{G}_D$. 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 an encoding 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. Thus, we have

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

We can thus use

$$p_D^{\text{EL}}(G) = \frac{m!}{\binom{m}{D_1^{\text{out}}, \ldots, D_n^{\text{out}}} \binom{m}{D_1^{\text{rel}}, \ldots, D_n^{\text{rel}}} \binom{m}{D_1^{\text{in}}, \ldots, D_n^{\text{in}}}}$$

as an approximation for the DS model. We call this the edge-list (EL) model. It always lower-bounds the true degree sequence model, since it affords some probability mass to graphs that cannot exist. [9]Experiments in the classical motif setting have shown that the ES model is an acceptable proxy [**?**], especially considering the extra scalability it affords.

Encoding D In order to encode a graph with L_D^{EL}, we must first encode D. [10] For each of the three sequences in D we use the following model:

$$p(D) = \prod_i p^{\mathbb{N}}(D_i) L(D) = -\sum_i \log p^{\mathbb{N}}(D_i)$$

where $p^{\mathbb{N}}$ is a distribution on the natural numbers. This is an optimal encoding for D assuming that its members are independently drawn from $p^{\mathbb{N}}$. When we use p^{DS} as the null model, we use the data distribution for $p^{\mathbb{N}}$ to ensure that we have a lower bound to the optimal code-length. When we use p^{DS} as part of the motif code, we must use a fair encoding, so we use a Dirichlet-Multinomial model to store D.

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

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

3 Experiments

3.1 Schema reconstruction

3.2 Query induction

3.3 Scale

To show the scale of our method, we show its performance on some very large graphs, culminating with a dump of the complete LOD cloud, containing 38 billion triples, from [].

3.4 Various

In this section, we show some experiments that are not intended to substantiate any claims about our method, but rather to illustrate the variety of uses for succesfull motif detection.

3.4.1 Analogical reasoning

3.4.2 Visualization

3.4.3 Feature extraction

3.4.4 Node embedding