# 1 TODO

**Code**

- Create fast KGraph implements DTGraph<Integer, Integer>.
  - ~~memory-back implementation~~
  - ~~Load RDF data (from HDT)~~
  - ~~Copy all tests from DTGraph, test~~
  - Create Disk-backed version.
- Implement null model.
- Implement sampling code.
  - → Implement Nauty for knowledge graphs
    - Test *really* thoroughly.
  - Implement sampling.
  - Test throughly.
- Implement **motif code**
  - Tests:
    - Implement random sampler. Check that random samples do not have significant motifs.
    - Sum probabilities for large sample of small graphs. Check that they do not sum to more than one.
    - ???

**Paper**

- Write preliminaries
  - Code and MDL
  - DM code, $\mathbb{N}$ code
- Write hypothesis test method
- Experiments
- Conclusion

# Detecting Motifs in Knowledge Graphs using Compression

August 14, 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 entities? 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 $p^{\text{null}}$ 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 relevance: 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 $p^{\text{null}}$ 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]

**Related Work**

**Preliminaries**

**Common codes** Extended DM code: test if necessary

## 2 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. Specifically, we will analyze the *structure* of knowledge graphs only, ignoring any meaning they have outside the graph. For instance, we will distinguish between links with different labels, but we will ignore any internal structure of the label itself (such as shared IRI prefixes). Under this assumption we can assume that the node and link labels of our knowledge graphs are simply natural numbers. This leads to the following definition.

A *knowledge graph* $G$, hereafter known simply as a *graph* is a, is tuple $G = (v_G, r_G, E_G)$. $v_G \in \mathbb{N}$ is the number of nodes in the graph, and $r_G$ is the number of relations. We will define the nodeset as $V_G = \{0, \ldots, v_G - 1\}$ and the relation-set as $R_G = \{0, \ldots, r_G\}$. The *tripleset* $E_G \subset V_G \times R \times V_G$ defines the links of the graph and their labels. [4]

As shown in Figure **??**, the patterns that we aim to find are partially labeled: some nodes and edges are labeled with negative integers. In other words, the motif is a *basic graph pattern*, and the edges and nodes marked with negative integers are variables. We will define a pattern in the same way we define a graph, but we will extend its nodeset en relationset. A pattern $M$ for graph $G$ is a tuple $(v'_M, r'_M, G, E_M)$. $v'_M$ and $r'_M$ indicate the number of variable nodes and links respectively. $M$'s nodeset is defined as $V_M = \{-v'_M, \ldots, v_G - 1\}$ and the relationset as $R_M = \{-r'_M, \ldots, r_G - 1\}$, with $E_M \subset V_M \times R_M \times V_M$, as for graphs. In short, nodes can be labeled

[3] ...

[4] Commonly, the elements of the nodeset of a graph are defined to be simple atoms from some larger unspecified universe. However, since we are discussing sampling graphs, we need to specify where these atoms come from. This definition specifies that graphs are essentially unlabeled, but nodes do have a definite ordering (i.e. two graphs can be isomorphic without being the same graph).

with nonnegative integers referring to $G$'s nodes or with negative integers representing a variable node, and similar for relations.

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 link $(s, p, o) \in E_M$ 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 $I^r_{-p}$. Put simply: for a pattern to match, variable links 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 $M$.

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. In most settings we do not actually have a given $M$ and $\mathcal{I}$ and we will need to search for them. We describe a simple and fast search algorithm in Section 2.3.

### 2.1 Relevance test

We will first describe our method under the assumption that a good null model $p^{\text{null}}$ is available, and that $-\log p^{\text{null}}(G)$ can be efficiently computed. In Section 3, 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 2.1 *more conservative*.

### 2.2 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 $L^{\text{base}}$ 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 $L^{\mathbb{N}}(\cdot)$. We also store the size of $\mathcal{I}$.

**the pattern** Let $M'$ be the unlabeled graph representation of $M$: that is, $M'$ is a plain graph (not a pattern), $V_{M'}$ contains only the first $V_{M'}$ natural numbers and similarly for $R_{M'}$. Otherwise, $M$ is isomorphic to $M'$: the node and edge labels of $M$ can be mapped to those of $M'$ so that $M$ only contains an edge $E$ if and only if $M'$ contains the edge resulting from mapping $E$'s symbols.

**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 $L^{\text{base}}(\cdot)$.

**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 node in $M'$, we create a list of length $|\mathcal{I}|$ of the nodes in $G$ to which it maps for each instance. For the non-variable nodes of the pattern, this will be a list simply repeating one node node of $G$. For the variable nodes, it will be a list containing some subset of the nodes in $G$. We store each list using the extended DM code. It may seem wasteful to store a list repeating a single element (instead of storing a labeled version of the pattern), but under the extended DM code such lists will be stored efficiently, using only a small, constant number of bits. The more varied the list, the more bits will be required to store it.

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

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

We note a few aspects of this code. Firstly, it achieves its compression by reducing the number of links it has to store. For each additional instance, it pays the cost of storing the instance, and any variable nodes and links, but it saves $|E_M|$ links in storing the template. 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, for some codes).

Nevertheless, we can make a few broad observations: patterns with many variable nodes and links are more costly to store than fully labeled patterns. On the other hand, a pattern with many variable nodes and links is more likely to have many instances, so that *if* it results in a small amount of compression per instance, the large number of instances may cause it to compress a lot. For the variable 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, however, all nodes in the graph are equally likely to appear in the position of the variable, 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. Thus, to our code, non-variable nodes are just a highly regular type of variable node. The distinction is only meaningful in the search algorithm.

### 2.3 Motif search

Usually, we do not start with a given pattern $M$ and a set of instances $\mathcal{I}$. We have only the graph $G$, and we wish to find *any* pattern 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. If 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 few as 500 samples can be sufficient to find the most frequent subgraph.

Theoretically, this allows us to generate promising candidates 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 *induced subgraph* for these nodes. Second, it con-

verts this induced 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 [**?**] (so that we record isomorphic samples as the same pattern).

For an induced subgraph with $n$ nodes and $e$ links, we can generate up to $n^2e^3$ different patterns: each node can be left labeled or made a variable node, each link can be removed, left unlabeled or be made variable. 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 $e^*$ $U([0..e'])$, with $n'$ the number of nodes to leave labeled, $e'$ the number of links to leave, and $e^*$ the number of links to leave labeled, respectively. If the resulting pattern is disconnected, we reject the sample.

We sample $K$ such patterns, recording a dictionary which maps canonical patterns $M$ to a list of observed instances $\mathcal{I}'_M$ of $M$ in $G$. Note that we must store the *links* 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.[5][6]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 without pruning it.

[5]

Our sampling algorithm is described more precisely in Algorithm 2.

# 3 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 length-$n$ integer sequences: $(D^{\text{in}}, D^{\text{rel}}, D^{\text{out}})$. If $D$ is the degree sequence of a graph, then node $i$ has $D_i^{\text{in}}$ incoming links, $D_i^{\text{out}}$ outgoing links and for each relation $r$, there are $D_r^{\text{rel}}$ 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 a fast approximation to the configuration model, which works well in practice for motif detection.

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 $j$ $D_j^{\text{out}}$ times, $P$ should contain relation $r$ $D_r^{\text{rel}}$ times and $O$ should contain node $j$ $D_j^{\text{in}}$ 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, which means they are not in $\mathcal{G}_D$. 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 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. Thus, we have

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

We can thus use

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

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.[7] Experiments in the classical motif setting have shown that the EL model is an acceptable proxy for the DS model [**?**], especially considering the extra scalability it affords.

**Encoding D** In order to encode a graph with $L_D^{\mathrm{EL}}$, we must first encode $D$. [8] 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 $p^{\mathrm{EL}}$ 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 $p^{\mathrm{EL}}$ as part of the motif code, we must use a fair encoding, so we use a Dirichlet-Multinomial model to store each sequence in $D$.

# 4 Experiments

## 4.1 Schema reconstruction

## 4.2 Query induction

## 4.3 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 [].

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

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

### 4.4 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 succesful motif detection.

#### 4.4.1 Analogical reasoning

#### 4.4.2 Visualization

#### 4.4.3 Feature extraction

#### 4.4.4 Node embedding

## A Appendix

### A.1 The canonical isomorphism algorithm for knowledge graphs

During sampling, we want to re-order the nodes in a sampled pattern to some canonical ordering. This way, when we sample isomorphic patterns, we recognize it, and group their instances together. The *nauty* algorithm is the standard solution for findings such *canonical isomorphs*. As we are sampling partially labeled, directed graphs, we have a lot of information that can speed up canonicalization. Since this steop is usually the bottleneck in sampling, it's important to perform it as efficiently as possible. Below, we provide a very succinct description of how nauty works, when applied to our patterns.

Let $V'_M$ and $R'_M$ represent the labels of the variable nodes and links in $M$. Pattern $M$ is an isomorphism of pattern $M'$ if there exist bijections $\sigma_v : V'_M \times V'M$ and $\sigma_r : \mathbb{R}'_M \times \mathbb{R}'_M$ such that applying these to the variable elements of $M$ yields $M'$ (under graph equivalence). The set of all patterns isomorphic to $G$ is the isomorphism class $[G]$. A function $C(P)$ is a canonicalization function if it produces for all any pattern $M$ in $[M]$ the same, canonical member of $M$.

Note that a very simple canonicalization algorithm would be to pass the unlabeled graph to the nauty algorithm, and use its canonical ordering on the labeled graph. However, the labels can make the canonicalization procedure significantly faster.

The key principle behind the nauty algorithm is the use of a *colouring*: a partition of the nodes of a graph. A *safe colouring* is defined as a partition $\pi$ of the nodes of the graph such that any permutation that maps nodes from different partitions to one another is guaranteed *not* to be an automorphism.

No guarantees are made about nodes in the same cell. For instance, the partition that puts all nodes in the same cell is always safe. If two nodes have different degrees, we can put them in different cells and be sure that we have a safe colouring. Given one safe colouring $\pi$, if node $a$ from partition $i$ has a neighbor in partition $j$ and node $b$ from partition $j$ has no such node, we can *refine* the partition $\pi$ by separating $i$ into a new partition (one contain $a$ and one containing $b$).

The nauty algorithm operates on a graph with a given colouring (with the unit colouring used by default). It first refines the colouring as much as possible, using degree information, and propagating the distinguishing features through the graph. Once the most refined colouring has been achieved, it searches through all autorphisms of the graph to produce a canonical example (using various optimizations to prune the search tree).

In our case, we use the colouring to indicate what our labeling tells us about the colouring of the nodes. Note the following:

- Labeled nodes in the pattern can be placed in a discrete cell of the colouring.
- Variable nodes should be placed into a single cell in the partition.
- We define a partition over nodes *and* relation labels. joh

**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 $M$
    a list $\mathcal{I}$ of instances of $M$ in $G$, a code $L^{\text{base}}$ on knowledge graphs.

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

*// Turn the pattern into a normal knowledge graph*
$E_{M'} = \{(\text{i}(s, V_M), \text{i}(p, R_M), \text{i}(o, V_M)) \mid (s, p, o) \in E_M\}$
$M' = ([0..|V_M| - 1], [0..|R_M| - 1], E_{M'})$
$b_{\text{pattern}} \leftarrow L^{\text{base}}(M')$

$E'_G = E_G - \cup_{\mathcal{I} \in \mathcal{I}} I$ *// Store the graph, with instances removed*
$b_{\text{template}} = L_{\text{base}}((V_G, R_G, E'_G))$

$b_{\text{labels}} \leftarrow 0$ *// Store the node labels*
**for** $n$ in $V_M$
    $\mathcal{I}_n = \langle v(n, I_1), v(n, I_2), ... \rangle$
        with $v(n, I)$ the label (from $V_G$) of node $n$ in instance $I$
    $b_{\text{labels}} \leftarrow b_{\text{labels}} + L^{DM}(\mathcal{I}_n)$

**for** $e$ in $E_M$ *// Store the edge labels*
    $\mathcal{I}_e = \langle r(e, I_1), r(n, I_2), ... \rangle$
        with $r(e, I)$ the relation (from $R_G$) of edge $e$ in instance $I$
    $b_{\text{labels}} \leftarrow b_{\text{labels}} + L^{DM}(\mathcal{I}_e)$

**return** $b_{\text{dim}} + b_{\text{pattern}} + b_{\text{template}} + b_{\text{labels}}$

**function** i$(s, S)$:
    **return** index of $s$ in sorted$(S)$

**Algorithm 2** The motif sampling algorithm.

**function** sample($G, s_{\min}, s_{\max}$):
    a graph $G$, a minimum and maximum motif size $s_{\min}$, $s_{\max}$

  Choose $s$ uniformly from $[s_{\min}..s_{\max}]$
  Choose a starting node $n$ uniformly from $V_G$
  $N \leftarrow \{n\}$
  **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 $N$
  Sample $n'$ $U([0..n])$, $e'$ $U([0..e])$ and $e^*$ $U([0..e'])$
  Randomly remove $e^*$ links, $e'$ linklabels and $n'$ nodelabels from $M$.
    (removed labels replaced by the next available negative integer)

  **return** canonical($M, N$)

**function** canonical($G, I$):
    a graph $G$, list of integers $I$
  **return** $G$ with nodes in canonical order, $I$ permuted the same way