Compression-based inference on graph data

Peter Bloem P@PETERBLOEM.NL
System and Network Engineering Group, University of Amsterdam, Science Park 904, 1098 XH Amsterdam, The Netherlands

Keywords: compression, machine learning, graphs, Kolmogorov complexity, minimum description length, normalized compression distance, subdue

Abstract

We investigate the use of compression-based learning on graph data. General purpose compressors operate on bitstrings or other sequential representations. A single graph can be represented sequentially in many ways, which may influence the performance of sequential compressors. Using Normalized Compression Distance (NCD), we test a sequential compressor versus a native graph compressor. We use both synthetic, randomly generated graphs and real-life datasets. We conclude that, even under adverse circumstances, sequential representations contain enough structure for shallow algorithms to perform inference successfully. Algorithms that operate directly on the graph representation usually require a considerable increase in resources, but do allow for an increase in performance also.

The mantra of compression-based inference is that *compression equals learning.* If we can compress data, we must have found some structure, we must have learned something. Conversely, if we have learned something about data, we should be able to use that knowledge to represent our data more succinctly. [1]

The idea of compression-based learning is expressed in two related subjects: *Minimum Description Length* and *Kolmogorov Complexity.* Minimum Description Length (Grünwald, 2007) builds a statistical framework on the principle that a good model is one that can be used to describe the data in as few bits as possible. Kolmogorov Complexity (Li & Vitanyi, 1997) is concerned with the mathematical expression of the *information content* of data. It states that if we can find some short description of a dataset (ie. compress it) then the total information content must be below the length of that description.

In this paper we investigate the use of these compression-based methods on graph data. Examples of such data are social graphs, transportation graphs, trade networks or semantic graphs. The graph is a powerful and versatile representation. Most applications of compression-based learning use sequential models, such as deterministic finite automata or block-sorting compressors, which operate on bitstrings. If we have data in the form of a graph, we can translate it to a bitstring, of course, but in this transformation we complicate our problem. For different orderings of nodes, the graph is the same, but the bitstring changes radically. To a simple compressor, the two bitstrings may not share very much structure, even though they represent the same graph. We will call this the problem of isomorphism.

If this issue really 'blinds' the sequential compressors to the structures in the graph, one option is to investigate compressors that operate on the level of the graph representation, for instance by finding frequent subgraphs or clustering the graph. The methods do not suffer this problem of isomorphism, but as a result they are more expensive than their sequential cousins. In this paper we hope to provide a first indication of how far the sequential approach will go, and whether the native approach will let us continue on from there.

There is a wealth of research available on machine learning and data mining both within single graphs

[1]Other features commonly associated with learning, such as generalization will, so the adherents of compression-based learning argue, be optimized along with the optimization of compression. See, for instance Grünwald, 2005; Grünwald, 2007.

Appearing in *Proceedings of BENELEARN 2013.* Copyright 2013 by the author(s)/owner(s).

and on sets of graphs (see for instance Cook & Holder, 2006). In this paper we do not share the data mining goal of extracting interesting features and models from the data, but only the goal of performing inference, using the compressors and their learning techniques as black boxes, and evaluating the the results of a chosen inference task.

Normalized Compression Distance

From the many machine learning methods based on principles of compression, we choose *Normalized Compression Distance* (NCD) as a representative method of compression-based learning. The choice is a practical one: NCD is a simple method which requires nothing more than a general purpose compressor. Any domain-specific knowledge we wish to use about our data (eg. it represents a graph) can be added to the compressor. We proceed under the assumption that conclusions reached about the performance of NCD on graph data will translate to MDL and other frameworks.

We will provide a brief, intuitive explanation of the principles involved, sufficient to understand the ideas presented in this paper. For a more in-depth and rigorous treatment we refer the reader to (Li et al., 2004) and (Cilibrasi & Vitányi, 2005). For a general introduction to Kolmogorov complexity, see (Li & Vitanyi, 1997).

Kolmogorov complexity

Kolmogorov complexity is a notion of information content based on two principles: (a) all data can be represented as a bitstring (b) the shorter this string can be described, the less information is contained in it.

The second principle is formalized in two steps. First, by 'description' we mean a description in a formal language that is maximally expressive. To formalize expressiveness, we require that the method of description is Turing-complete (of equivalent strength to a Turing machine). By the strong Church-Turing thesis, this suggests that there is no reasonable way of defining a more expressive method of description.

We choose the Turing machine as a canonical method of description, and we fix some enumeration of all Turing machines $\{T_i\}$. There exists a Universal Turing machine U that is defined as follows:

$$U(\langle i, p\rangle) \simeq T_i(p)$$

That is, if U's input consists of two bitstring arguments i and p, combined with some computable pairing function $\langle\cdot,\cdot\rangle$, then U computes the same function as T_i on input p if $T_i(p)$ halts, or fails to halt if T_i fails to halt. U provides our true formalization of 'description'. If our data is x, and for some $y = \langle i, p\rangle$, $U(y) = x$, we say that y is a description of x on our reference universal Turing machine.

We can now say that with respect to U, there must be a minimal description for any given data:

$$K_U(x) = \min\{|y| : U(y) = x\}$$

The result that makes Kolmogorov complexity a useful measure of information content is that $K_U(\cdot)$ is only marginally dependent on the choice of U. If we suppose that there is another universal description method, V, we might ask what the expected difference is between $K_U(x)$ and $K_V(x)$. Let $k_V(x)$ is the shortest program for x on V. Since, U is universal, we know that it can compute k_V by simulating V. Somewhere in our enumeration of Turing machines $\{T_i\}$, there is a T_ν which computes the same function as V. This simulation is a program for x on U which is a bound for the shortest program on U:

$$\begin{aligned} K_U(x) &\leqslant |\langle T_\nu, k_V(x)\rangle| \\ &= |k_V(x)| + |\nu| + p(\nu) \\ &= K_V(x) + O(1) \end{aligned}$$

Where $p(\nu)$ is the penalty (ie. the additional bits) that the pairing function requires to store its two arguments in a separable way. We require that this is only dependent on ν. The final line shows that $K_U(\cdot)$ and $K_V(\cdot)$ differ only by a constant term, which is independent of x. To summarize: we may differ in opinion on how much information our data contains, but only by a constant amount.

In some contexts, it is desirable to distinguish between the classical Kolmogorov complexity and the prefix-free Kolmogorov complexity. In the context of Normalized Compression Distance the distinction does not matter.

A complete treatment of Kolmogorov Complexity is outside the scope of this paper, but the following properties are important to understand.

$K(\cdot)$ **is uncomputable** There can be no algorithm which computes the Kolmogorov complexity of x for all x. It can, however, be bounded from above, and for every algorithm which bounds it, there is another algorithm which provides a better bound.

All computable compressors approximate $K(\cdot)$ If we have some compressor for our data x (say

GZIP) we can find the decompression algorithm somewhere in $\{T_i\}$, say as T_g. We can have a description on U as $U(\langle g, z\rangle) = x$ so that $K_U(x) \leqslant |z| + O(1)$. In this way, any computational structure in x is taken into account in $K(x)$, and $K(\cdot)$ can be approximated by any computable compressor.

This gives us the basic philosophy behind all translations of Kolmogorov complexity to the realm of practical applications: we approximate Kolmogorov complexity by some learning algorithm or compressor.

Finally, we define conditional Kolmogorov Complexity $K(x \mid y)$. Where regular Kolmogorov Complexity is defined as the shortest program which produces x, the conditional variant is defined as the shortest program which produces x when provided with y. A complete treatment is available in (Li & Vitanyi, 1997).

Normalized Information Distance (NID)

The length of the shortest program to get from y to x intuitively suggests that $K(\cdot \mid \cdot)$ can be seen as a similarity measure. Clearly, very little is required to transform a string into itself, or into a very similar string, whereas for two random strings, only a program that stores the second in its entirety can make the transformation.

This intuition prompted Li and Vitányi (Li et al., 2004) to investigate the use of Kolmogorov Complexity as a metric of computational similarity. To acquire a true metric, some problems have to be solved. The first is that $K(\cdot \mid \cdot)$ is not symmetric: it takes a small program to blank out the collected works of Shakespeare, but the reverse is a more complex operation. The first step, then, is to define the (symmetric) Information Distance:

$$\mathrm{ID}(x, y) = \max\left[K(x \mid y), K(y \mid x)\right]$$

The second issue is one of scale. If two strings of a million bits differ by 1000 bits, we might consider them quite similar, whereas two strings of 1000 bits that differ by that amount could not be more different.[2] Therefore, we would like to take the length of the strings into account. This gives us the Normalized information Distance (NID)

$$\mathrm{NID}(x, y) = \frac{\max\left[K(x \mid y), K(y \mid x)\right]}{\max\left[K(x), K(y)\right]}$$

[2]Note that this is only an intuitive example. If two strings differ in exactly every bit, a very short program transforms one into the other, so by NID, they are very similar.

We would like to approximate this by replacing each occurrence of the Kolmogorov complexity with an approximation by a compressor, which we will call C. As most compressors do not work on a conditional basis (expressing data given some existing data), we want to rewrite the conditional K's as nonconditional ones. To achieve this, we accept beyond the constant term uncertainty that is innate to Kolmogorov Complexity, a further logarithmic inaccuracy. This allows us to rewrite as

$$\begin{aligned}\mathrm{NID}(x, y) &= \frac{\max\left[K(x, y) - K(x), K(y, x) - K(y)\right]}{\max\left[K(x), K(y)\right]} \\ &= \frac{\max\left[K(xy) - K(x), K(yx) - K(y)\right]}{\max\left[K(x), K(y)\right]}\end{aligned}$$

If we replace the Kolmogorov complexity with a compressor C, we get the normalized compression distance

$$\mathrm{NCD}(x, y) = \frac{C(xy) - \min\left[C(x), C(y)\right]}{\max\left[C(x), C(y)\right]}$$

This step also includes the assumption that our compressor is roughly symmetric ($C(xy) = C(yx)$).

When our data is represented as a graph, rather than a string, we replace the notion of concatenation of strings by concatenation of graphs. That is, we combine the graphs x and y into a single (disconnected) graph.

Methods

Our aim is to test a sequential and a graph-based compressor on an inference task for a variety of graph data. To ascertain the performance of the compressors, we generate graphs from different sources, calculate their NCD distances and see whether a clustering algorithm can reconstruct the original sources as clusters. Datasets and source code for these experiments are available. [3]

Node ordering

An important and subtle concern is the ordering of nodes in the sequential representation of our graphs. This issue is detailed very well by Kang & Faloutsos, 2011. As shown, there are various algorithms to determine node orderings that bring out a lot of the graph's inherent structure in the adjacency matrix, allowing a sequential compressor to exploit it. We could use substantial resources to find a good ordering of nodes to

[3]`http://www.peterbloem.nl/benelearn2013`

improve the performance of the sequential compressor. If we did, however, the extra computation might mean that the compressor is no longer a shallow model. To maintain the sequential compressor as a representative example of shallow models, the node ordering should be cheap to establish from a random ordering, preferably in linear time.

Since we are testing the capacity of the general purpose compressor to perform inference despite the problem of isomorphism, we will actually present it with a worst case scenario. We use a random ordering of nodes for all graphs. If the general purpose compressor still outperforms the random baseline under these conditions, it will tell us that it is, at least in part, resistant to the problem of isomorphism.

Experiment 1: Synthetic data

We generate graphs from four models.

The first is the classic Erdős-Rényi (ER) model, where a uniform random choice is made from all graphs with n nodes and m links. The second is the Barabási-Albert (BA) model (Albert & Barabási, 2002), which grows a graph from a set of n_0 unconnected nodes, one node at a time, connecting each new node to k distinct existing nodes where the probability that a given existing node is chosen for a connection is its degree, divided by the sum of the degrees of all nodes. Thus, under the BA model the more links a node has, the higher the probability that it will accrue even more. This effect causes the degree distribution of a BA network to become scale-free (ie. it follows a power law).

Since we want there to be some challenge in separating the two classes of network, we ensure that they have the same number of nodes and links. To achieve this, we first generate the BA networks, count their nodes and links and use these as parameters for the ER model.

We also include graphs from the fractal graph generation algorithm from (Song et al., 2006). We set the hub-parameter which determines the level of fractality (as a trade-off with the level of small-worldness) to 0.0 (for a small world network) and to 1.0 (for a fractal network).

Once we have this dataset, consisting of four gold clusters, we calculate the NCD with a given compressor for every pair of graphs in the dataset, giving us a symmetric matrix. We use the k-medoids algorithm to cluster this set into four clusters.

To assess the performance of the clustering we label the clusters so that the accuracy is maximized (essentially assigning the optimal labels). Clearly, this would be cheating when testing a classifier, but since we are only interested in the clustering aspect, it gives us a straightforward performance measure.

As a random baseline, we generate a random distance matrix with every distance a uniform random value in $[0, 1)$, and run the clustering algorithm on that.

Experiment 2: Real-life data

In this experiment we sample subgraphs from large, existing graphs. We sample by choosing a random node uniformly from all nodes and performing a random walk of length n. We then extract a subgraph containing the nodes encountered and any links connecting two encountered nodes. We replace all node and link labels with a single canonical symbol.

With this dataset of subgraphs, we proceed as before, calculating the distances between the subgraphs and clustering them into as many clusters as we have sources, to see whether the resulting clusters match the sources.

We use the following datasets:

cellular The cellular network of the E. Coli bacterium. (Jeong et al., 2000) Acquired from `http://www.nd.edu/~networks/resources/metabolic/index.html`

neural The neural network of the C. Elegans nematode worm (ignoring link directions). (Achacoso & Yamamoto, 1991; Watts & Strogatz, 1998) Acquired from `http://toreopsahl.com/datasets/#celegans`

co-purchase A graph of items commonly purchased together on internet retailer Amazon.com. (Leskovec et al., 2007) Acquired from `http://snap.stanford.edu/data/amazon0302.html`

Compressors

GZIP

We use GZIP as our general purpose compressor. Specifically, in our experiments, we use the implementation of GZIP that is part of the standard Java SDK. To store a graph with GZIP, we flatten the lower half of its adjacency matrix into a bitstring and store this together with a list of the node and link labels. We use Java object serialization to take care of delimiting the label data and translating it to bits. (Since all graphs in our experiments have a single label, this is unlikely to affect the outcome).

Subdue

Subdue (Jonyer et al., 2004; Ketkar et al., 2005) is an algorithm for finding frequent subgraphs in graph data. The algorithm searches for the subgraph that maximally compresses the data. The body of the algorithm is essentially a beam search through the space of subgraphs. It consists of three main routines:

Subgraph matching This is an algorithm for finding the occurrences of a given subgraph in a graph. The method used is detailed by Bunke & Allermann, 1983. Since this is a semi-exhaustive search for the solution to the NP-complete problem of subgraph isomorphism, the matching can only be solved for very small subgraphs. Unfortunately, even with subgraphs of four or five nodes, the matching is too slow in combination with the number of times it is executed to calculate a full distance matrix. To combat this issue, we remove all but the first $\mathsf{b}_{\text{inner}}$ elements from the search queue after it is sorted at each iteration, effectively turning the algorithm into a beam search.

MDL Scoring This routine takes a subgraph, finds its occurrences in the data by the previous routine and deletes these from the data. The subgraph is then stored once, together with the remainder of the data and a list of where the subgraph should be attached to reconstruct the original data. See the appendix for an exact description.

Subgraph search This is the 'outer loop' of the algorithm. Starting with a graph of a single link between two nodes, it searches through the space of all connected graphs by extending each current candidate by one link at a time (possibly by adding a new node as well). The buffer of current candidates is sorted by MDL score, and the candidate with the highest score is extended to create new candidates. At each iteration all but the top $\mathsf{b}_{\text{outer}}$ candidates are removed, turning the search into a beam search.

Algorithm 1 shows a broad description of the whole procedure.

The graph matching search (the first line of the **score** function) allows for inexact matches of the subgraph. In these cases, we use a rough upper bound of the number of bits required to transform the stored subgraph into the subgraph that is actually present in the data.

Parameter settings and specifics

All graphs generated contain 100 nodes. In the BA-model, we attach one node each step, giving 100 links also. Note that this makes the BA graphs UAGs. For more nodes attached per step the clustering problem would become more difficult. The random graphs are generated with the exact same number of nodes and links. The fractal graphs we generate to depth 2 by adding 4 ancestors at each side of each link and 1 extra link between the groups of ancestors. This results in networks of 90 nodes and 100 links.

For each source, we generate 3 graphs.

The Subdue algorithm has a lot of parameters. During the search we return only one best subgraph. Our beam width at the top level ($\mathsf{b}_{\text{outer}}$) is set to 5. The beam width in the graph matching routine ($\mathsf{b}_{\text{inner}}$) is set to 10. We run the search for 10 iterations, limiting the size of the subgraph used to 5.

We let the k-medoids algorithm run for 20 iterations. This is more than enough for convergence in all experiments.

Algorithm 1 Pseudocode for the Subdue algorithm

```
G: the data graph
$\mathsf{b}_{\text{outer}}$: the beam size

$S \leftarrow [K_1]$ # initialize the list of substructures with a graph of a single node

loop
    $s \leftarrow$ head element of $S$
    add all extensions of $s$ to $S$
    sort $S$ by score($s'$, $G$) for $s' \in S$
    remove all but the first $\mathsf{b}$ elements of $S$

function score($s'$, $G$)
    replace occurrences of $s'$ in $G$ with node $N$
    annotate links to $N$ with the nodes in $s'$
    return nr of bits to store the edited $G$ and $s'$
```

Results

Table 1 shows the results on randomly generated graphs. Table 2 shows the results for subgraphs sampled from real-life datasets.

Conclusions and future work

Our experiments show that sequential, general purpose compressors are better at performing graph inference than expected. Despite the random ordering of the nodes, the bitstring contains enough shallow patterns that a compressor like GZIP can tell the two types of fractal graphs apart, and only struggles with the dif-

Table 1: Confusion matrices for various compressors. Columns represent the clusters found by the k-medoids algorithm. To calculate the error, we label the resulting clusters so that the error is minimized (ie. reorder the columns of the confusion matrix to maximize the sum of the diagonal). We report the mean error (1 - the sum of the diagonal) over 10 experiments (and the standard deviation in brackets) below each confusion matrix. The confusion matrix shown is always for the first experiment in the run.

ER	0.17	0.083	0	0
BA	0.083	0.17	0	0
fractal (pure)	0.083	0	0.17	0
fractal (small world)	0	0	0	0.25

(a) Random baseline: error 0.46 (0.11)

ER	0	0.25	0	0
BA	0	0.25	0	0
fractal (pure)	0	0	0.25	0
fractal (small world)	0	0	0	0.25

(b) GZIP: error 0.27 (0.12)

ER	0.25	0	0	0
BA	0	0.25	0	0
fractal (pure)	0	0	0.25	0
fractal (small world)	0	0	0	025

(c) Subdue: error 0.14 (0.14)

Table 2: Results for the experiment on natural datasets.

cellular	0.11	0.11	0.11
neural	0.11	0.11	0.11
co-purchase	0.11	0	0.22

(a) Random baseline: error 0.43 (0.11)

cellular	0.33	0	0
neural	0.22	0.11	0
co-purchase	0	0.22	0.11

(b) GZIP: error 0.28 (0.17)

cellular	0.33	0	0
neural	0	0.33	0
co-purchase	0.33	0	0

(c) Subdue: error 0.34 (0.17)

ference between the random and BA graphs. This is particularly interesting considering the high resource requirements of most algorithms for graph inference, and the low resource use of general purpose compressors.

This result suggests that at least some inference on graphs can be performed by sequential algorithms on a sequential representation in linear time, with decent results.

As for the graph-compressors, we see a small improvement relative to the sequential compressors for a strong increase in computational resources. Subdue as used in this paper is a relatively simple compressor, which isolates only a single subgraph for compression and we tested it only at modest parameters. The publications surrounding Subdue offer much more complex solutions (such as the induction of graph grammars (Jonyer et al., 2004)). To investigate the promise of these models as compressors further, it will be necessary to investigate both parallelized versions of these algorithms and a more elegant relaxation of the exhaustive nature of their components. Subgraph sampling methods like the ones detailed by Kashtan et al., 2004, may be able to provide a significant increase in performance.

The notion of compression-based learning is a good framework within which to combine many approaches to inference from the most general to the most domain specific. The Minimum Description Length principle and its associated techniques, which have not been investigated yet for reasons of scope, offer the promise of an even broader field of approaches to the analysis of graph data.

Acknowledgements

This publication was supported by the Dutch national program COMMIT. The author would like to thank Leen Torenvliet, Eugenio Bargiacchi and Eugenio di Leo for comments and preliminary research.

We would like to thank the reviewers for many insightful comments.

Appendix: Graph coding

The method of coding data is always a sensitive point in compression-based learning. The precise choices made in translating the data to a string of bits can greatly affect which patterns are picked up, or ignored by the subsequent inference procedure. Here we detail the procedure used to encode the graphs in both compressors.

GZIP

We take half of the adjacency matrix ($(n^2+n)/2$ bits), and store it in flattened form as an array of java byte primitives, together with the size of the string (since the stored representation is padded to a multiple of eight). We then serialize this representation into a java GZIPOutputStream. To make our implementation generic, we assume that the nodes and links are labeled, and serialize the labels in a fixed order after the adjacency matrix. In the graphs mentioned in this paper, there is a single label assigned to all elements, so inference shouldn't be affected by the labels.

Subdue

In the subdue case we do our coding in a more precise way, without relying on platform-specific functions. More importantly, we only count the bits required to store our graph, rather than actually constructing the representation itself.

Plain graph

To store a plain graph, we follow roughly the coding strategy outlined in Holder et al., 1994.

We first store the number of nodes n in prefix free coding, and the maximum number of neighbours for a node in the graph $n_{\max}$ (in $\log n$ bits). We then store the lower half of the adjacency matrix (including the diagonal) row by row.

For node i, we only need to store the connections to the i nodes below it including itself. We use $\log n_{\max}$ bits to store the number of such neighbours n_i, and $\log \binom{i}{n_i}$ bits to store the configuration of those neighbours.

After this, all that remains is to encode the labels of the nodes and links. We assume that the sender and receiver in our coding scheme possess a codebook that is efficient for the data given, so that if a label l occurs with frequency $\#l$, we can encode it in $-\log \frac{\#l}{\sum_k \#k}$ bits.

We assume that there is some canonical ordering among the nodes and links, and store them as a stream. Since the number of labels is known from the adjacency matrix, the code for the entire graph is self-delimiting.

Graph with substructure

To store the graph with a substructure, we first store the substructure itself. Since this contains no special symbols, we can store it simply using the above method (except we use the codebook based on the whole graph rather than the substructure to encode the labels). This is a prefix code, so we can start encoding the rest of the graph right after.

In the rest of the graph, we remove the nodes matched to the substructure and all links connecting to them. We store the 'silhouette' of the substructure as a plain graph (again with the codebooks of the whole graph).

We then store the way the substructures should be connected into the silhouette to reconstruct the original graph. For each substructure we first store the transformation cost (if the substructure was an inexact match), with a prefix penalty to make the description self-delimiting, we then store the number of links connecting the substructure un prefix-coded form, and then for each link we take $\log s$ bits to encode how to connect it in the substructure and $\log d$ to connect it in the rest of the graph, where s is the number of nodes in the substructure and d is the number of nodes in the silhouette.

In later versions of our code, the occurrences of the subgraph are replaced by symbol nodes, but the version used to perform these experiments uses the silhouette method.

References

Achacoso, T. B., & Yamamoto, W. S. (1991). *Ay's neuroanatomy of c. elegans for computation.* CRC.

Albert, R., & Barabási, A.-L. (2002). Statistical mechanics of complex networks. *Reviews of modern physics*, *74*, 47.

Bunke, H., & Allermann, G. (1983). Inexact graph matching for structural pattern recognition. *Pattern Recognition Letters*, *1*, 245–253.

Cilibrasi, R., & Vitányi, P. M. (2005). Clustering by compression. *Information Theory, IEEE Transactions on*, *51*, 1523–1545.

Cook, D. J., & Holder, L. B. (2006). *Mining graph data.* Wiley-Interscience.

Grünwald, P. (2005). A tutorial introduction to the minimum description length principle.

Grünwald, P. D. (2007). *The minimum description length principle.* MIT press.

Holder, L. B., Cook, D. J., & Djoko, S. (1994). Substructure discovery in the subdue system. *Proceedings of the Workshop on Knowledge Discovery in Databases* (pp. 169–180).

Jeong, H., Tombor, B., Albert, R., Oltvai, Z. N., & Barabási, A.-L. (2000). The large-scale organization of metabolic networks. *Nature*, *407*, 651–654.

Jonyer, I., Holder, L. B., & Cook, D. J. (2004). Mdl-based context-free graph grammar induction and applications. *International Journal on Artificial Intelligence Tools*, *13*, 65–79.

Kang, U., & Faloutsos, C. (2011). Beyond'caveman communities': Hubs and spokes for graph compression and mining. *Data Mining (ICDM), 2011 IEEE 11th International Conference on* (pp. 300–309).

Kashtan, N., Itzkovitz, S., Milo, R., & Alon, U. (2004). Efficient sampling algorithm for estimating subgraph concentrations and detecting network motifs. *Bioinformatics*, *20*, 1746–1758.

Ketkar, N. S., Holder, L. B., & Cook, D. J. (2005). Subdue: compression-based frequent pattern discovery in graph data. *Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations* (pp. 71–76).

Leskovec, J., Adamic, L. A., & Huberman, B. A. (2007). The dynamics of viral marketing. *ACM Transactions on the Web (TWEB)*, *1*, 5.

Li, M., Chen, X., Li, X., Ma, B., & Vitányi, P. M. (2004). The similarity metric. *Information Theory, IEEE Transactions on*, *50*, 3250–3264.

Li, M., & Vitanyi, P. M. (1997). *An introduction to kolmogorov complexity and its applications*. Springer Verlag.

Song, C., Havlin, S., & Makse, H. A. (2006). Origins of fractality in the growth of complex networks. *Nature Physics*, *2*, 275–281.

Watts, D. J., & Strogatz, S. H. (1998). Collective dynamics of small-world networks. *nature*, *393*, 440–442.