On the Summarization of Complex Networks

Isac Daniel de Figueiredo Novo

Mestrado em Engenharia de Redes e Sistemas Informáticos
Departamento de Ciência de Computadores
2022

Supervisor
Pedro Ribeiro, Professor Auxiliar
Faculdade de Ciências da Universidade do Porto

MSC

Sworn Statement

I, Isac Daniel de Figueiredo Novo, enrolled in the Master’s Degree in Network and Information Systems Engineering at the Faculty of Sciences of the University of Porto hereby declare, in accordance with the provisions of paragraph a) of Article 14 of the Code of Ethical Conduct of the University of Porto, that the content of this dissertation reflects perspectives, research work and my own interpretations at the time of its submission.

By submitting this dissertation, I also declare that it contains the results of my own research work and contributions that have not been previously submitted to this or any other institution.

I further declare that all references to other authors fully comply with the rules of attribution and are referenced in the text by citation and identified in the bibliographic references section. This dissertation does not include any content whose reproduction is protected by copyright laws.

I am aware that the practice of plagiarism and self-plagiarism constitute a form of academic offense.

Isac Daniel de Figueiredo Novo

30/9/2022

Abstract

Complex Networks have ubiquitous applications to a variety of different scenarios, ranging from the ever growing inter-connectivity of the modern world, to the representation of complex molecular structures. As the name entails, its complexity is an innate characteristic, often detrimental to the retrieval of useful information. Graph mining algorithms, that toil to provide a suitable means of doing so, achieve that with an adequate measure, commonly in a non-deterministic way and always with a high computational cost.

Since complex real world networks, tend to exhibit statistically relevant patterns, whose properties are inherited by the large network. As such, their detection and classification tend to be the subject of much attention by part of the network science and data mining communities. A graph's relational nature, is the basis for much of its prevalence as abstract data type. Frequently being adopted as part of the implementation of another structure, or being abstracted into a completely different structure for a particular use. Nevertheless, since it also exists primarily as a pure mathematical concept, its properties remain the same no matter from which angle you choose to look at them. And, therefore, discoveries made in one field of study can usually be transferred to another, when they pertain to graph theory at its core. In the process, yielding novel ways to look at old problems.

A complete summarization of a graph, if ever possible, would allow to bypass many of the setbacks that arise when dealing with such complex structures. The goal of moving a step forward towards such possibility, if ever so slightly, is what is being proposed in this work. The summarization by compression of a complex network, via its motifs, can potentially be used to derive meaning from it. At the same time, providing a simpler and quintessential representation.

Keywords: Complex Networks, Graph Compression, Graph Mining, Graph Summarization, Isomorphism Class, Minimum Description Length, Network Motifs, Vocabulary of Graphs, Subgraph

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

Graphs are a powerful abstraction capable capturing the entities involved in a system and the relationships between them. Recently, *Network Science* has been emerging as promising multidisciplinary field, integrating areas such as graph theory, statistical mechanics, data mining and information visualization, aiming precisely to study as a whole graphs that emerge from real life systems [3, 34].

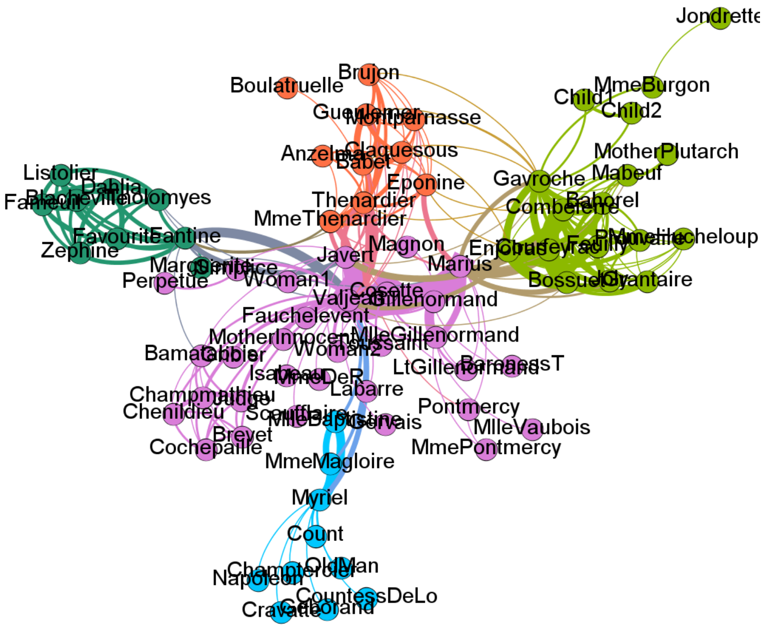


Figure 1.1: An example graph modelling the coappearance weighted network of characters in the novel *Les Miserables*. Made with Gephi [6] with data from Stanford GraphBase [19].

Network Science has a vast applicability [13] and has allowed researchers to study networks from a multitude of areas, such as biology [5] (e.g. protein interaction [26] or brain networks [11]), economics [32] (e.g. trade [14] or stock networks [37]), sociology [10] (e.g. friendship [17] or co-authorsiph networks [27]) or urban science [28] (e.g. road [33] or crime networks [36]).

Many of the studied networks have non trivial topological characteristics and a scale that does not make it easy to task of analyzing them. Because of that, since early there have has been research on how a graph could be compressed, while keeping its essential characteristics, trying for instance to speedup computations [16] or improving its visualization [15]. While summarization of other data types is older, *graph summarization* [23] is a much younger field, aiming precisely to provide methods to condense and simplify network based data. Methods for compressing graphs have thus emerged, exploiting characteristics such as the existence of "hub-like" nodes with many connections [22] or pre-defined subgraph structures [20].

1.1 Goals and Motivation

This work aims precisely to tackle graph summarization and compression, providing an initial proof of concept approach aiming towards a general compression scheme able to automatically use subgraph patterns to provide a "vocabulary" adapted to the specificity of any given networks.

Our goals are twofold. Firstly, we want to provide an alternative, straightforward, and compressed definition of a graph, other than a list of edges, which, in accordance to the minimum description length principle, provides a better description of it. Secondly, while doing so, we want to be able to use the detection of network motifs, used in the creation of that definition, as a means to infer important characteristics concerning the network.

The main motivation was exploring the premise that, at a conjectural maximum level of compression by motif detection, a putative vocabulary of its subgraphs would be able fully convey a semantic description of the graph. However, while the first goal was fully achieved, to a degree, the second remains the subject of future work.

The graph definition proposed by this work, is not only comprised of an unordered edge list, as it also includes elements which define a compressed network motif, and a dictionary of isomorphism classes for such motifs.

Compression of network motifs is done by initially detecting and capturing information regarding subgraphs on the large network, and then collapsing the nodes of non-overlapping subgraphs into a single one. In a manner analogous to consecutive edge contractions for all the edges on a subgraph. Moreover, by compressing the exact isomorphism class of subgraphs, and not induced subgraphs derived from the nodes in them, the method accounts for networks which are multigraphs or contain loops.

Since only edges between compressed motifs need to be explicit, the final definition saves memory. And, since network motifs become explicit, deriving meaning from them becomes easier. The compression is lossless, so restoring the graph to its exact original form is always possible.

An algorithm for quick discovery of network motifs by means of a novel data structure - the g-trie - as proposed by Pedro Ribeiro and Fernando Silva, was used for detecting subgraphs of a given size, which are compression candidates in the large networks. The adjacency matrix categorizing the isomorphism class of each subgraph, is given as a canonical lexicographically larger adjacency string, which prevents dictionary repetitions of symmetrical subgraphs, on the final compressed definition.

Before compression, an heuristic sorts the list of candidate subgraphs, in an attempt to maximize the compression rate. The algorithm then iterates over each element on the sorted compression candidate list, collapsing the first occurrence of each non-overlapping subgraph into a single node. In the process, removing all edges whose permutation of nodes, is in conformity with that subgraph's canonical definition.

Of the heuristics used for sorting the compression candidate's list, there are two distinct approaches, both applied in ascending and descending order. One approach sorts the list by the frequency of each occurring subgraph type, while the other sorts by the frequency of edges on the subgraph.

The application of this method was able to produce a compressed graph for every one of the large networks tested. With the information regarding compressed motifs being readily available for frequency count and classification, directly from the raw description of the compressed graph, without any sort of preprocessing being required.

1.2 Thesis Organization

The rest of this document is organized in the following manner:

- Chapter 2 provides some important background regarding graph theory. It describes not only fundamental notions regarding graphs, as well as a more in-depth description of the concepts which are most relevant to this work. It clearly defines isomorphism, subgraphs, and network motifs. As well as provides an overview of the problems and proposed solutions concerning the detection of motifs.
- Chapter 3 describes related work, which served as both basis as well as inspiration for the method implement in this work. It also describes the algorithm used for detecting network motifs, a preliminary step to the compression done.
- Chapter 4 provides an in-depth description of the implementation of programs Compress and Restore which, as the name so eloquently imply, compress and restore a network, respectively. It also clarifies the full structure and principles, behind the proposed graph description.
- Chapter 5 shows the results of testing the proposed compression algorithm, over real world networks. It provides an analysis of the results. Evaluating the level of compression of the method, its shortcomings, and potential improvements, as suggested by the resulting data.

- Lastly, chapter 6 offers conclusions regarding the proposed compression method. Going into further analysis regarding its strengths and weaknesses. Putting forth ways in which it can be improved in future work.

Chapter 2

Notions on Graph Theory

Simply put, graph theory is the study of the mathematical structures known as graphs.

On the most rudimentary of definitions, a graph is a collection of vertices paired by a given number of edges. It is often a visual representation, that can be thought of as a diagram, depicted by a series of dots or circles connected by lines that can be either straight or curved. And given that we humans are predominantly visual creatures, in the sense that we innately assimilate data in that manner, graphs can be meaningful representations of it - up to a certain size scale.

As data structures, they are versatile, being able to be applied to any type of relational data, even when the existence of that relation is not self-evident. In fact, several other forms of either representing or grouping data can be summed up as being a graph - *e.g.*, a genealogical tree, or any tree (data structure) for that matter, or a simple linked list, that is nothing more than a path graph.

For that flexibility, graphs can be used to represent any network of people, objects, tasks, etc. And, the relation between those elements it represents, can be as varied as being social, filial, conceptual, spatial, temporal, and so on and so forth. Their usage ranges from representing social groups to finite-state machines, and everything in between.

The myriad of purposes to which a graph can be applied to, unfurls its manifold applications. Several of its metrics can be applied with varied intentions and results, such as the shortest path between two vertices being used to determine the degree of separation between two people (*e.g.*, Bacon level), generate a routing table, or simply the actual shortest euclidean distance between two points.

Ideally, a graph would thus be analogous to a pictogram. A semiotic image from which a simple, undeniable meaning could be promptly inferred. Unfortunately, while that interpretation may hold as true for a select few simpler and smaller networks, the same cannot be stated - and is far from being the case - for the large and complex networks for which graphs are are frequently applied to. Furthermore, the relational nature of a graph, which still makes it a paramount data structure for several computations on networks of all sizes, further draws it away from that quasi-idyllic notion of a condensed yet substantial and straightforward visual depiction of data.

This last point, serves as the basis for what is the major ambition of this work. The notion that a lossless compression of a complex data structure, is not purely intended as a means of saving memory space, but also as a mechanism through which meaningful patterns can emerge and more easily be detected. Hopefully, thus providing insights regarding the data and the relationships therein.

2.1 Graph Definition and Terminology

Formally, on the field of discrete mathematics, a graph is a structure containing an object set and its relational data. It models pairwise relations between those objects, often called edges or links. The objects themselves, thus related, in turn being designated as vertices or nodes - used interchangeably throughout this report. It is a pair $G = (V, E)$, where V is the set of vertices and E the set of edges pairing them.

Two vertices v and u are said to be endpoints of an edge that pairs them, which is said to be incident to each of those two vertices. The given vertex v is said to be adjacent to u, and vice-versa. This reciprocity holds true for undirected graphs only[1], since the adjacency matrix of an undirected graph is not necessarily symmetric, as further detailed below.

Due to its discrete nature, the set V, an by extension E, are assumed to be finite. And, while infinite graphs depicting some sort of special binary relation can be sometimes considered, they are not the subject of this work.

2.1.1 Fundamental Notions

For a mathematical structure which, at first glance, deceptively appears to be a very basic construct, graphs can actually become quite complex. And not just as a consequence of their pervasiveness and multitude of applications.

Graph theory encompasses diverse nomenclature, not all of which is relevant for this work.

Relevant concepts include graph isomorphism which, as the name suggests, refers to having the same form or shape; or subgraphs, which are smaller graphs within larger graphs. These more pertinent topics will be delved into on section 2.3. Suffice to say, that on the matter of isomorphism, the ground principle behind the concept is that graphs which apparently are different, can in fact be equivalent. They share all the same properties.

Thus, one way to look at what constitutes a graph's property is that it is something that remains invariable throughout every possible isomorphism - hence, why it is also called an invariant. Accordingly, a graph's properties are the quintessential form of categorizing it.

[1]more information on graph characterization on section 2.1.2

Given all of the above, the following are some of the basic relevant concepts regarding graph theory, starting with its properties.

- **Order -** The total number of vertices $|V|$ for graph G.
- **Size -** Commonly, the total number of edges $|E|$ for graph G. In some particular situations, however, it may instead correspond to $|V| + |E|$.
- **Degree -** The number of incident edges to a node, also called *valency*. Nodes with self-loops are counted twice for undirected graphs, once for directed ones.

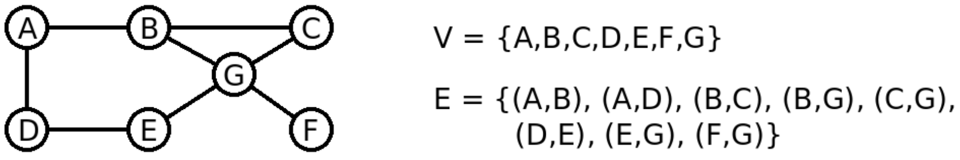


Figure 2.1: An example undirected graph with order 7 and size 8.
For instance, node A as degree 2 and node G has degree 4.

- **Volume -** The sum of the degrees on a set of vertices.
- **Self-Loop -** Also, simply loop. An edge connecting a vertex to itself. On an undirected graph, a loop adds two the vertex degree. On a directed graph it adds one to both the in degree, and the out degree of its incident vertex.
- **Path -** Sequence of edges joining a given sequence of vertices, finite or infinite. In the most universal of senses, every path is also a *walk*. Where, for $G = (V, E, \phi)$, with ϕ being a group of functions with range $P_2(V)$ and domain E, there is a composition P_i with v_n vertices and e_{n-1} edges, given the sequences $(v_1, v_2, ..., v_n)$ and $(e_1, e_2, ..., e_{n-1})$, for which there exists a $\phi(e_i) = \{v_j, v_{j+1}\}$ with $i = 1, 2, ..., n-1$. If $v_1 = v_n$, then the walk is considered closed, and open otherwise.

  A *trail* is a walk where all edges are distinct while, strictly speaking, a *path* is a trail in which all vertices are also distinct. For the latter case, some authors prefer the clear distinction of a *simple path*, when not considering every path to have all vertices be distinct from one another.

  Walks, trails, and paths, can be directed for directed graphs.
- **Cycle -** A non-empty closed trail where only the first and last nodes are the same. Can be directed or undirected, for directed or undirected graphs, respectively. Any non-empty closed trail is a *circuit*, therefore cycles are circuits where all vertices are distinct - with the above mentioned exception for the node closing the trail.

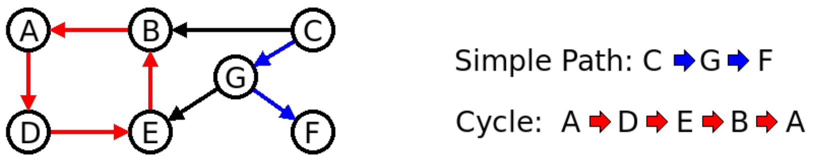


Figure 2.2: An example of a simple path and a cycle on a directed graph.

- **Hamiltonian Path -** A path, in either an undirected or directed graph, that visits each vertex exactly once. By extension, a *Hamiltonian cycle* is a cycle that does the same as the above. As such, removing a single edge from a Hamiltonian cycle, produces a Hamiltonian path.
- **Distance -** The number of edges on the shortest path between two vertices.
- **Diameter -** The longest distance between two vertices on a graph or, in other words, the maximum eccentricity of any vertex $v \in V$. Given diameter d, in which $d = max_{v \in V} \epsilon(v) = max_{v \in V} max_{u \in V} d(v,u)$, with eccentricity $\epsilon(v)$ defined as $\epsilon(v) = max_{u \in v} d(v,u)$. The radius r can, therefore, be defined as $r = min_{v \in V} \epsilon(v) = min_{v \in V} max_{u \in V} d(v,u)$. A central vertex is one which eccentricity equals the radius. While a peripheral vertex has at least one distance to another vertex that is equal to the graph's diameter.
- **Betweenness/Closeness Centrality -** Distinct approaches to measure the centrality in a graph, both centered on measuring the length of the shortest paths between all nodes. Centrality, in this sense, being an indication of how well connected a node is to the rest of the network, and thus a means of ascertaining its importance.
- **Subgraph -** Of a given graph, is another graph formed from a subset of its vertices and edges. The vertex subset must include all the endpoints for edges present on the edge subset. But, the edge subset does not necessarily need to include all the edges from the original graph, that are incidents to the vertices on the vertex subset. If it does, then the subgraph is called an *induced subgraph*.

  More on subgraphs in section 2.3.

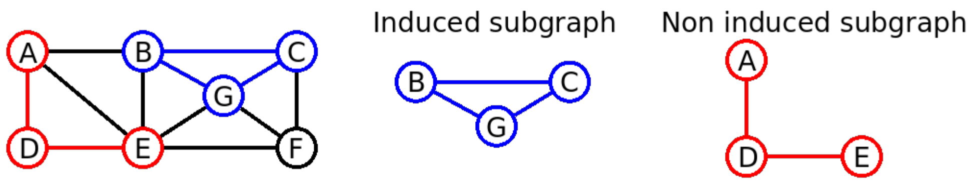


Figure 2.3: An example of an induced and a non-induced subgraph.
For the second subgraph to be induced, it would have to include the (A, E) edge.

- **Adjacency -** Two vertices are said to be adjacent, if there exists an edge that is incident to them both - *i.e.*, that connects them.

- **Neighbourhood -** Pertaining to the adjacency of a vertex v, its neighbourhood is the subgraph comprised of v and all the nodes that are adjacent to it, with the edges connecting them - induced subgraph of v.

  The other nodes are said to be *neighbors* of v.

- **Edge Contraction -** The operation of removing an edge from the graph, resulting in merging the two vertices that it was incident to, while keeping the neighbors of those vertices as neighbors of the merged one.

- **(Giant) Connected Component -** Of an undirected graph, by definition a subgraph that is not part of a larger subgraph. *Giant connected components*, are connected components that take up a significant fraction of the entire vertices on the graph, and are of a given size much larger than that of the vast majority of other components on the graph.

- **Clique -** A complete subgraph, *i.e.*, one where all distinct vertices are adjacent to each other.

- **Maximum/Maximal Clique -** A maximum clique is any clique with the largest number of vertices, on the graph. A maximal clique, is one that cannot possibly be further extended by including another vertex.

- **Clique Number -** The largest order of a complete subgraph. Specifically defined as the number of vertices present in the maximum clique.

- **Caveman Communities -** Said of two or more tightly knit subgraphs - *e.g.*, cliques - that are sparsely connected to one another, usually by a small number of edges.

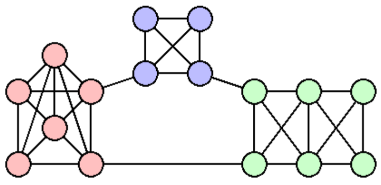


Figure 2.4: An example of a network exhibiting three different caveman communities.

- **Clustering Coefficient -** A measure of the clustering tendency of nodes. Given the *local clustering coefficient* of a vertex as the measure of how close its neighbourhood is to form a clique, the *average clustering coefficient* is simply that, the average of local clustering coefficients for every vertex on the graph. There exists also a global clustering coefficient measured in triplets of nodes - connected components of three nodes either as a closed triplet (cycle) or open triplet (open trail) - and the ratio between them.

2.1.2 Graph Characterisation

Different types of networks have been compressed, studied, and analyzed. From the aforementioned invariants, graphs can be classified in several ways, as described below.

These classifications provide immediate knowledge regarding certain characteristics of those graphs, that distinguishes them from graphs of a different classification and, like with the previously mentioned properties from which they derive, need to be accounted for during compression.

- **Undirected or Simple -** Usually simply called graph, can be seen as the standard or basic form of everything that was stated above. In it, the set of E edges is an unordered pair of vertices from V. Which is to say, a link between vertices v and u can either be edge (v, u) or (u, v). Or, in other words, if u is adjacent to v, then v is adjacent to u.
- **Directed, Mixed, and Oriented -** Unlike simple graphs, directed graphs - or *digraphs* - have orientated edges, and therefore the set of edges contains ordered pairs of vertices. For that reason, it is necessary to distinguish whether an incident edge is incoming or outgoing. Visually, this is accomplished by representing the edge as an arrow, with its tail on the starting node and the head on the destination node.

  In simple terms, for there to be a symmetrical direct path of length one between v and u, there needs to be both an edge (v, u) as well as an edge (u, v).

  This causes the vertices of directed graphs to have two types of degrees that needed to be taken into consideration. *In degree*, from incoming edges; and, *out degree*, from outgoing edges. A vertex with only outgoing edges, is also called a *source*, while a vertex with only incoming edges is called a *sink* - occurring in *flow networks*, for example.

  Graphs can be *mixed*. With both directed and undirected edges.

  An *oriented graph* is a directed graph where for each possible pair of vertices, only one ordering of that pair can exist as an edge, at most. An undirected graph, where each of its edges receives an orientation - *i.e.*, a specific order for that pair of vertices - produces an oriented graph.
- **Weighted -** A graph is said to be weighted if it has a numeric value associated to its edge.
- **Bipartite -** Is a classification given to a graph whose vertices can be divided into two disjoint and independent sets, with each edge only occurring between a vertex from a different set.
- **Connected -** A graph is said to be a connected one if there is at least one edge incident to every possible pair of vertices. The opposite is a **disconnected** graph, where at least one pair of vertices isn't joined by an edge.

  By definition, a graph is said to be connected if it contains a path between v and u, for any $v \in V$ and $u \in V$.

  On the case of a directed graph G, direct paths are considered and, if replacing all directed edges with undirected ones would result in a connected graph, then G is said to be *weakly connected.* Otherwise, if it conforms with the general definition - there exists a direct path $\phi(v, u)$, for any $v \in V$ and $u \in V$ - then it is said to be *strongly connected.* On the other hand, if for every missing direct path $\phi(v, u)$ there exists a path $\phi(u, v)$, it is said to be *unilaterally connected.*

- **Multigraph -** A multigraph is a graph which is specifically permitted to contained multiple edges joining the same pair or vertices - which are called parallel edges. Not to be confused with a *hypergraph*, in which an edge can connect more than one pair of vertices.

  For graph $G = (V, E)$, E is a multiset, rather than a set, of either unordered or ordered vertices, for undirected or directed graphs, respectively. Some multigraphs can contain edges with their own identity, for which $G = (V, E, r)$, with $r : E \to \{\{v, u\} : v, u \in V\}$ assigning a specific identifier to each edge.

- **(Search/Prefix) Tree -** A tree is generally an undirected graph for which any pair of vertices is connected by precisely one path. Some tree data structures can, however, be directed.

  Tree data structures are hierarchical. This hierarchy defining a topmost root node, parent nodes, children nodes, and terminal leaf nodes. All nodes on the path from a given node to the root node of a tree, are its ancestors, while all nodes on every path towards all leaf nodes are its descendants. The concept of sub-trees is also well-defined as being subgraphs of the original tree, in which a child node serves as root to that sub-tree.

  A *search tree* is a tree data structure used for locating specific keys within a set and retrieving a value tied to it. Nodes of a search tree are connected and ordered according to a specific design, such as all left children nodes of a *binary search tree* having smaller values than right children nodes.

  A *prefix tree* or *trie*, is a *k-ary search tree* where nodes do not store the associated keys, but instead it is the position of that node on the tree that defines it. With all the children nodes of a given node, sharing a common prefix string to the parent node, all the way to the empty string of the root.

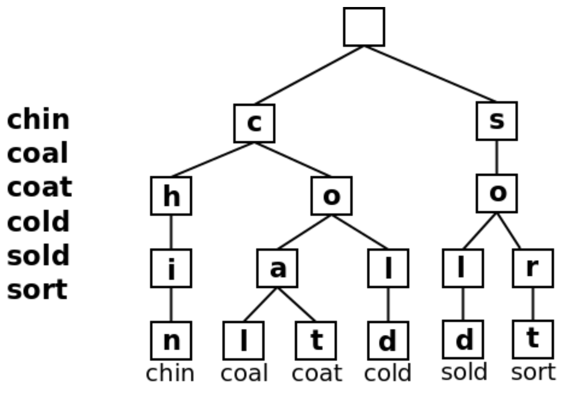


Figure 2.5: An example of a prefix tree containing six words.

2.2 Graphs as Abstract Data Types

As abstract data types, graphs are a finite set of vertices linked by a finite set of edges. Each edge can have an associated weight value, and either be directed or not. Nodes are identified by an assigned label, often simply numbered or identifying a particular trait. Edges are perfectly characterized by the nodes that they pair up, but may be labeled as well.

2.2.1 Implementation and Operations

A graph is usually implemented as either an adjacency list or an adjacency matrix, depending on the purpose for which the data structure is intended. Both implementations having their unique advantages and drawbacks, which will be further discussed on section 2.2.2.

Regardless of its implementation, a graph must provide common basic operations, which usually include, but are not restricted to:

- **Add Vertex-** Adding a new node to the graph;
- **Add Edge -** Adding a new edge, connecting two nodes, to the graph;
- **Remove Vertex -** Removing a vertex from the graph, if it exists, also removing any incident edge;
- **Remove Edge -** Removing an edge pairing two nodes, if present;
- **Get/Set Vertex Value -** Retrieving or setting the value of an existing node;
- **Get/Set Edge Value -** Retrieving or setting the value of an existing edge, for graphs which support them;
- **Adjacent -** Determine whether two nodes are adjacent, *i.e.*, there exists at least one edge connecting them;
- **Neighbors -** Return the list of all neighbours of a node, that is, of all nodes which are adjacent to it - corresponding to returning the *adjacency list* of a node.

Evidently, aside from the aforementioned operations, any number of other operations may be implemented depending on the algorithm for which the graph is being used. The most common ones being searches for a specific node or edge, upon which the majority of more complex algorithms rely upon - such as computing distance metrics or sorting algorithms.

The implementation of all these operations may - and in fact, should - take into account the nature of the mathematical graph which the abstract data type represents.

For example, whether it is directed or undirected, contains loops or cycles, is a multigraph or not, and so on and so forth. Not accounting for the attributes made explicit by these categorizations, can affect the correctness of an implementation. On the other hand, by taking advantage of them, the implementation of many of these operations can be vastly simplified and made more efficient - *e.g.*, by knowing that a graph is a tree, we can account for it being acyclical, and therefore not have to worry about causing the occurrence of an endless loop while traversing it.

Lastly, it is important to mention that these data types often take as input a list of edges and, by adding it they also add the nodes it connects, if they aren't already present.

And, while an edge list is most likely the lightest way of storing the complete definition of a graph in memory, it lacks any sort of useful applicability. It must first be processed into a data type that allows the easy manipulation of information it contains, such as for determining neighbors of a node, and so forth.

As stated, a major goal of this work was to transform this compact definition - a straightforward list of edges - into an even more compact one, while still adding pertinent information to it - namely, the subgraphs that were detected and compressed, as well as their isomorphic classes.

2.2.2 Adjacency Lists Versus Adjacency Matrices

As mentioned on the previous section, graph data structures are commonly stored in one of two ways: as an adjacency list, or an adjacency matrix.

An adjacency list is a collection of unordered lists of nodes, connected to another node. Whereas an adjacency matrix, is a square matrix whose elements indicate the existence of an edge between a vertex pair as given by the row and column of the matrix.

Adjacency matrices of undirected graphs are therefore always symmetric, while those of directed graphs can be asymmetric. On a simple graph, with no loops, the diagonal elements of its adjacency matrix do not denote the existence of an edge. And, if the adjacency matrix only contains binary elements, it cannot be used to represent multigraphs. This can be remedied by employing a count for the number of edges between two nodes, as elements of the matrix.

The trade-offs between an adjacency list and matrix, lie between both space and time complexity. The main purpose of these representations is the retrieval of adjacencies between nodes, an operation for which a matrix offers much faster computations given the direct relation between a position on a 2-dimension array and its indices.

In opposition, adjacency lists are usually much more efficient memory-wise, especially for sparse networks. This is true, despite matrices being able to store information in a more compact way, while promoting locality of reference - *e.g.*, a single bit for each entry in a continuous allocated memory space, as opposed to a pointer to another variable or object in memory.

This is due to the fact that the creation of an adjacency matrix usually implies that the entire structure be placed in memory, which can be costly for graphs with several nodes but few edges. In summation, the space occupied by an adjacency list in memory, is proportional to the number of edges and vertices in the graph, while the one occupied by an adjacency matrix is proportional to the square number of vertices.

Furthermore, despite matrices being faster at retrieving individual edges - taking constant time, an operation that for an adjacency list, in the worst case, is equal to a node's degree - retrieving all neighbours of a node has an asymptotic complexity proportional to the number of vertices in the graph, since all possible pairings must be checked. Which is much more costly.

For that reason, and due to most algorithms employed on graphs requiring the retrieval of all nodes adjacent to a given node, an adjacency lists is usually the preferred implementation for such cases. With the time complexity of such operation, being proportional only to the degree of the given node.

Other aspects can be taken into account, when choosing whether to employ and adjacency list or matrix, such as the amount of additions and removals of vertices, which is much more demanding for the latter implementation than the former. Seeing as it can imply having to recreate an entirely new matrix and copy every single element from the old matrix to the new one, in the event of the old matrix not having enough allocated memory to support the new node(s) - while also wasting memory space by keeping memory reserved for nodes that have been removed from the graph.

All these aspects were taken into consideration, when coding the compression program for this work. Which ultimately led to an implementation via adjacency lists - expressly, due to the concomitant removal and addition of nodes, associated with subgraph compression.

2.3 Subgraphs and Network Motifs

Once again, the first and foremost goal of this project, was to compress networks by reducing subgraphs to a single node, while still retaining all the information regarding the nodes and edges of the original network. Done in such a way, as to be possible to later recreate it. Furthermore, by means of this compression, it aimed to make it easier to identify not simply central - *i.e.*, important or relevant - nodes on the network, but even entire connected components or clusters.

Therefore, considering that these two subjects are the main focus of the work done, this subsection is dedicated to deepening the notions of subgraphs and motifs, after the brief mentions to them on previous sections.

In a nutshell, a network motif is a statistically significant subgraph pattern of a larger graph - often the whole network. And, in a cursory description that is made self-evident by the name itself, a subgraph is a smaller graph contained within a larger one.

However, before delving into those two definitions, it is important to first define what constitutes graph isomorphism.

2.3.1 Graph Isomorphism

Briefly put, a graph isomorphism is said to occur between two graphs G and G', if a bijection $f : V(G) \to V'(G')$ is present. Strictly, v and u are adjacent, for $v, u \in V$, if and only if $f(v)$ and $f(u)$ are adjacent with $f(v), f(u) \in V'$.

In layman's terms, what this translates to, is that graphs that appear to be distinct either because, for example, their nodes are dissimilarly labeled or their drawings (visual representation) are distinct, can actually contain the same exact structure. These graphs are said to belong to the same isomorphism class, and the bijection notation given as $G \sim G'$.

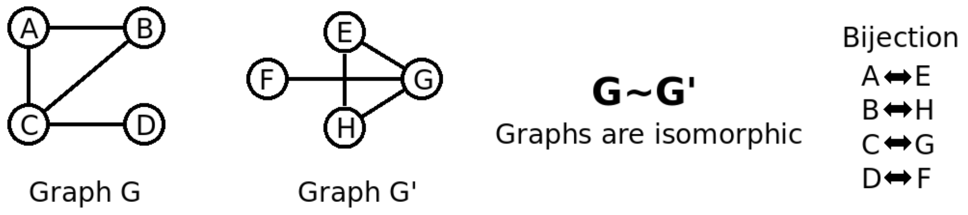


Figure 2.6: An example of two isomorphic graphs.

A reminder that for the graph $G = (V, E, \phi)$[2], V is the set of vertices, E the set of edges, and ϕ a set of P_2 functions. The functions $\phi(e_i) = \{v_i, v_{i+1}\}$ given $i = 1, 2, ..., n-1$, defines a sequence $e_1, e_2, ..., e_{n-1}$ of distinct $e \in E$ edges, called a path, for which there is a sequence $v_1, v_2, ..., v_n$ of distinct vertices $v \in V$, called the vertex sequence of the path. Then, essentially, two graphs $G = (V, E, \phi)$ and $G' = (V', E', \phi')$ belong to the same isomorphism class if the pairs V and V', E and E', and ϕ and ϕ', are all equivalent.

Properly and succinctly, given the above definitions of G and G', everything that was stated can be summed up in the following manner:

$G \sim G'$ is true, if there is a bijection $\nu : V \to V'$ such that $\{v, u\} \in E$ if and only if $\{\nu(v), \nu(u)\} \in E'$. For $\nu : V \to V'$ and $\epsilon : E \to E'$, with $\phi'(\epsilon(e)) = \nu(\phi(e))$ for all $e \in E$, and $\nu(\{x, y\})$ defined as $\{\nu(x), \nu(y)\}$.

One final and brief note, without delving too much into the subject, and regarding the detection of isomorphism between two graphs, is that the complexity of such task still remains an unsolved problem in computer science to this day.

It is uncertain whether the proposed task belongs to the P, NP-complete, or NP-intermediate complexity classes. However, its generalization, the *subgraph isomorphism problem*, is known to be NP-complete.

[2]See section 2.1.1

2.3.2 Subgraphs

As already stated, subgraphs G' of a larger graph G, are formed from a subset of vertices and edges of G. The vertex subset must include all vertices for which the edges on the subset subset are incident to, while being allowed to contain additional vertices.

Formally, a graph $G' = (V', E', \phi')$ is a subgraph of $G = (V, E, \phi)$, if $V' \subseteq V$, with ϕ' being the restriction of ϕ to E' in range $P_2(V')$, so that $\phi(e')P_2(V')$ for all $e' \in E'$. And, by restriction ϕ' of ϕ to E', it is meant a function ϕ' with domain E' such that $E' \subseteq E$, which satisfies $\phi'(x) = \phi(x)$ for all $x \in E'$.

Subgraph isomorphism, previously discussed, is of importance to the matter of locating specific subgraphs on a network, since what is basically being done is finding a subgraph that belong to the same isomorphism class as another - the goal of the search. This, accordingly, extends to finding motifs on large networks.

Several problems related to finding specific maximal subgraphs of a certain isomorphism class, are well-known to be NP-complete problems. Such as the *clique problem*, which deals with finding cliques in a graph, and of which common formulations involve finding maximum or maximal cliques of any given size. Another example, is that of finding induced subgraphs belonging to a specific isomorphism class.

This *subgraph isomorphism problem*, is a generalization of both the *maximum clique problem* - finding a maximum clique - and the *Hamiltonian cycle problem* - that of determining whether a Hamiltonian cycles occurs on a graph. Both NP-Complete.

2.3.3 Network Motif

Reinstating the already mentioned definition, a network motif is simply a relevant repeating pattern on a large network. In other words, it is a statistically significant subgraph, belonging to a given isomorphism class, that occur at much higher rates in real world networks than in similar random ones. The term *network motif* as here stated was first introduced by Milo et al. [25], and Figure 2.7 explains it visually. Being subgraphs of a specific isomorphism class important to a given network, detecting them on complex networks is a computationally demanding task.

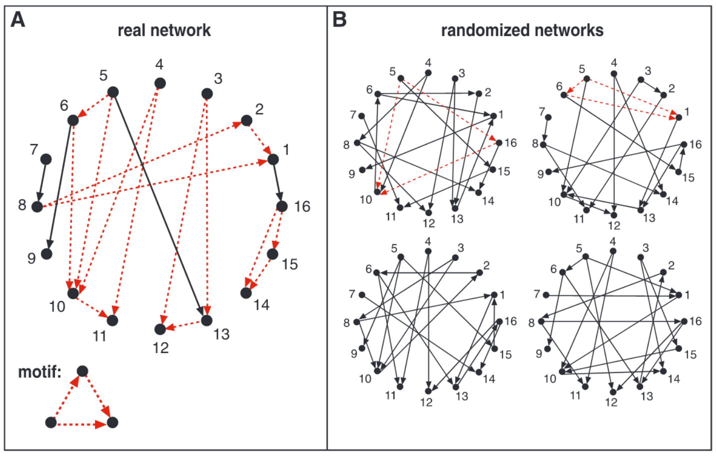


Figure 2.7: The concept of a network motif. The small three node motif (a feed forward loop) appears five times in the original network but only once or never on the similar randomized networks that preserve the same degree sequence. Taken from [25].

The concept behind the importance of detecting network motifs, is that large networks often inherent properties from those motifs and, as such, its detection can provide important insights that are often otherwise obscured by the size and complexity of the large network.

Since this work revolves around compressing such motifs, resorting to an efficient detection method was of utmost importance.

2.3.3.1 Motif Detection Algorithms

Several algorithms have been proposed as solutions for the subgraph isomorphism problem, roughly in the last twenty years - since 2004. But, prior to that, the only available method was the rigorous brute-force count proposed by Milo *et al* [25].

The following is a brief breakdown of some of those algorithms in chronological order, up until the method that was used in this work. It serves as an account on the evolution of approaches and occasional paradigm shifts, with the prospect of providing greater insight into the problem.

- **mfinder -** Starting from the standpoint that network motifs predominantly occur in biological and engineered networks, as opposed to randomized ones; *mfinder*, published in 2004 by Kashtan *et al.* [18], was the first implementation of a sampling method for the estimation of subgraph concentrations and detection of motifs, asymptotically independent of network size.

  The algorithm was based on a random sampling of connected edges, until a subgraph of a designated size n was found. At which point, that subgraph would be expanded to all the edges between those n nodes. At each iteration, the selection bias for non-uniform sampling of a new connected edge among the available candidates, was affected by a given weight attached to any specific subgraph isomorphism class.

After each sample, a weight of score $W = 1/P$ was added to update the current score of the relevant subgraph type, with P being the calculated probability of sampling a particular subgraph on the network. After repeating the sampling procedure for a given number of steps, the concentration of all subgraph types was calculated in accordance with their final scores.

However, in order to calculate the probability P for a subgraph of size n, all possible ordered sets of $n-1$ edges that could lead to sampling that subgraph, needed to be checked. This further taxed the computational work being done.

- **FPF -** Also based on the axiom that biological processes form large complex networks, in which motifs potentially underlie relevant properties, Schreiber and Schwöbbermeyer proposed the *flexible pattern finder* (FPF) algorithm in 2005 [31].

  The algorithm constructs a pattern tree consisting of nodes representing different subgraph types, which expanded in accordance to the tree's depth analogously to a prefix tree. That is to say, in which the parent of each node constitutes a subgraph of its children nodes. The patterns in nodes are the ones supported by the target subgraph being searched, and the search algorithm transverses the tree.

  The algorithm is based on applying different concepts for determining the frequency of a pattern in a target graph, defined as the maximum number of different matches for that pandattern. Concepts ranged from counting every possible match, to restricting the reuse of graph elements shared by different matches of a pattern.

  The search space was reduced for these latter frequency concepts, by pruning the tree given the application of the downward closure property for the frequency of descending patterns. Which states that the frequency of a subgraph decreases monotonically to the increase in its size.

- **ESU -** Following the previous trends of finding meaningful motifs in large biological networks, Wernicke proposed *ESU* in 2006 [35], a faster algorithm for subgraph sampling than the previously proposed *mfinder*.

  ESU, yielded an unbiased sampling algorithm by efficiently enumerating all subgraphs of size k, and then randomly skipping over a portion of these subgraphs during execution.

- **G-Tries -** Proposed by Ribeiro and Silva in 2010 [29], the method presents a novel data structure for storing a collection of subgraphs, for which it is named after. A *G-trie* is a prefix tree of graphs, with each node representing a directed graph, and the graph of a parent node sharing common substructures with the graph of its children nodes.

  The search tree can be further augmented by giving each node symmetry breaking conditions. It is still the fastest subgraph detection method available to this date.

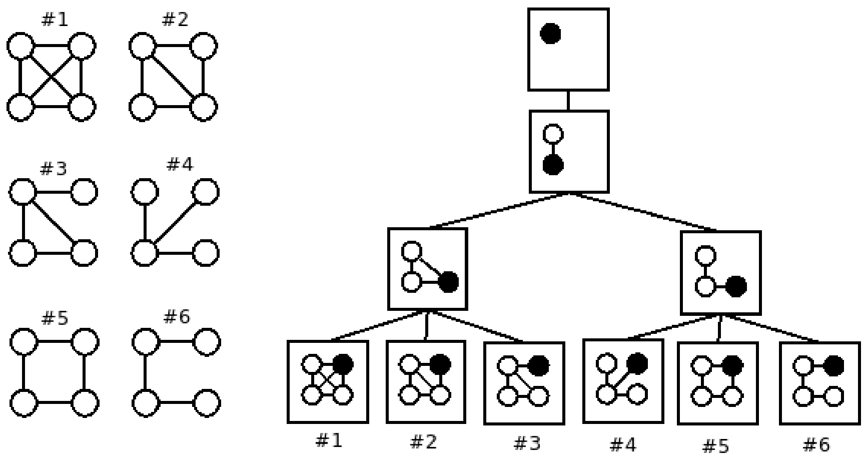


Figure 2.8: An example g-trie storing six subgraphs. Adapted from [30].

The next chapter focus on related work, in which this one is based and built upon. Particularly, the above mentioned g-trie algorithm as well as the VoG algorithm, which served as partial inspiration for the work that was done.

Chapter 3

Related Work

As stated numerous times throughout the previous chapters, a foremost objective of this work was to find a more compact description for a graph, that could easily be imported into an abstract data type in memory. While, at the same time, already displaying pertinent information as is.

This was achieved by detecting network motifs, compressing them, and outputting a new graph file which contained three distinct types of entries:

- An isomorphism class dictionary;
- An unordered list of subgraphs belonging to one of the isomorphism classes on the above dictionary;
- An unordered list of edges, not belonging to any of the subgraph entries.

The aim of this representation is not only to be able to save memory space by reducing the number of lines and characters used; as it is also to offer the ability to directly read, identify, and count, any and all subgraphs detected and compressed while generating the definition. Even before or without processing any of the other nodes or edges on the network.

The second, and more audacious goal, is that this compression might bring to light recurring motifs and patterns on a large network, that might be helpful into ascertaining relevant properties regarding the large network, that would otherwise be too demanding or maybe even impossible to directly determine.

This notion, having its inception on the VoG algorithm proposed by Koutra *et al.* [20], has the ultimate ambition of being a stepping stone into the complete formulation of a graph vocabulary. Allegorically stated, the creation of a formulation in which nodes can be seen as letters, motifs as words, and the complex network a text from which substance can be directly derived.

VoG's approach, served as the inception for the approach taken in this work for achieving a novel graph description. Nevertheless, it evolved into something entirely different.

As shown by Koutra *et al.* on VoG, Lim *et al.* on SlashBurn [22], and several other authors [9], the goal of compressing a large network oftentimes goes hand-in-hand with that of mining and summarizing its characteristics.

Ever so slightly going on a tangent, the principle of *Occam's razor* - which pop culture has probably transformed into the second most pervasive scientific term, coming in second place only to *Schrödinger's cat* - can be summed up as the principle that 'entities should not be multiplied beyond necessity'.

And, like *Schrödinger's cat*, it is also often inaccurately portrayed. Being simplified as 'the simplest answer is usually the right one', it actually attempts to convey the notion that when presented with rivaling theories, the simpler one is usually preferred. That is to say, for example, the one whose models have the fewer parameters or lemmas. It is not, in any way, shape, or form, intended as a means of selection between models offering different predictions. But, rather, that when selecting amongst models concerning the same prediction, the simpler one should be the favored one.

It is also not, in the least, an irrefutable principle of logic. Nevertheless, it hold some appeal, particularly when juxtaposed to the falsifiability criterion of the scientific method.

Going back to the subject at hand, the *minimum description length* (MDL) principle, can be succinctly described as the mathematical application of Occam's razor. Applied to model selection, it states that the best model constitutes the shortest description of the data. Its perspective, is therefore that compressing data is tantamount to learning from it [8]. And supports the aforementioned notions of mining data through compressing it.

As mentioned, several approaches to graph compression have been made, not all pertaining to identifying and compressing subgraphs, and for that reason those will not be covered on this chapter.

3.1 Graph Compression

Assuming an unweighted graph $G = (V, E)$, there exists a representation $G' = (S, C)$, consisting of a summary $S = (V_S, E_S,)$ and edge correction C [7]. Formally, for every $v \in V$ of G, it holds that $v \in V \in V_S$ of G'. And, in turn, every edge $E_{ij} = (V_i, V_j) \in E_S$. In other words, regular nodes on the original subgraph, are included on supernodes of the compressed graph, and its superedge contains the set of all edges previously connecting those original nodes that now connect the supernodes.

This notation is trivial for a bipartite graph which is compressed into two nodes V_i and V_j, with C containing information for recreating the original edges between nodes (see Figure 3.1 for a visual depiction). The cost of such representation can be defined as $cost(R) = |E_S| + |C|$, ignoring the cost regarding the mapping of nodes, which is negligible when compared to that of mapping edges.

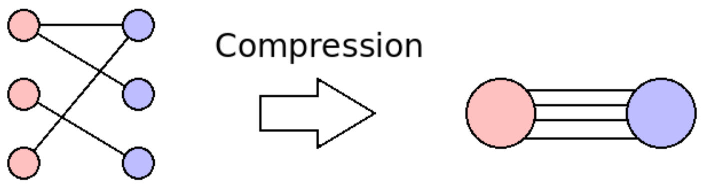


Figure 3.1: Compression of a bipartite graph into two nodes.

For any V_i and V_j, Π expresses all possible edges between all pair of nodes $(v, u), v \in V_i, u \in V_j$, and $|\Pi_{ij}| = |V_i| * |V_j|$. Then, for $A_{ij} = \Pi_{ij} \cap E$ being the set of actual edges between V_i and V_j, there exists two possible representations. Either all existing edges are added to C, resulting in an encoding cost of $|A_{ij}|$. Or, one edge is added between the supernodes while C stores all non-existing ones, which yields a cost of $1 + |\Pi_{ij}| - |A_{ij}|$. The former is better for loosely connected supernodes, while the latter for for those with a dense connection.

As per the MDL principle, the best representation is the one with the lowest cost. Therefore, determining this cost implies choosing, for each possible pair of V_i and V_j, the smaller cost $c_{ij} = min\{|A_{ij}|, 1+|\Pi_{ij}|-|A_{ij}|\}$. Altogether, finding the best set of supernodes is computationally expensive as it depends on the chosen set. As an alternative, a certain amount of error could be allowed, that sees a portion of the data being lost, in an attempt to minimize the work being performed.

Two problems immediately arise from the previous general definition, in regards to this work's proposed goal.

Firstly, by compacting a graph to such an extent, all network motifs are lost, and there is no meaningful information that can be immediately inferred.

Secondly, due to the need to calculate all possible combinations of the V_i and V_j sets, the approach is clearly unfeasible for large networks, and the alternative of a lossy compression is not the desired outcome.

There is obviously merit to the demonstration, and the notation serves as a general basis for what is intended and performed by this work. Concretely, the fact that analogous supernodes, containing the nodes of the compressed subgraph, are created. However, there is no need for a correction or error sets to be considered, as all the edges of the original graph are either present or easily inferred by the network motif's description.

In other words, the proposed method of compression, involves performing edge contractions to all edges of our (non-induced) subgraph. Furthermore, by clearly identifying the isomorphism class of all compressed subgraphs, it becomes trivial to fully restore those edges and vertices when and if needed.

Compared to the absolute compression described on this section, the proposed method of compressing non-overlapping network motifs is far from being a minimum description of a graph. Nevertheless, it is a shortest description of one that, at the same time, manages to express relevant information in a concise manner.

To tackle the problem of describing a graph with recourse to compressing network motifs, three algorithms from related work were focused on, as detailed on the following sections of this chapter. The first two, also consisting of compression algorithms: SlashBurn and VoG.

The third, already outlined on section 2.3.3.1, and being regarded as the preferred method for performing the computationally expensive task that is detecting network motifs, was the g-trie algorithm.

3.1.1 SlashBurn

Lin *et al.* proposed SlashBurn [22], a compression algorithm that approaches graphs as a collection of structures named hubs connecting spokes. It recursively splits the graph into those hubs and spokes, connected only by the hubs.

The proposed algorithm also put forth selection methods for the hubs, that work for real world graphs, gives better compression than other methods at the time, and led to faster execution times for matrix-vector operations.

Since real world graphs tend to contain few nodes with high degree, while the remaining ones have very low degree, attempting a cut tends to extract homogeneous regions. SlashBurn bypasses that problem by locating novel communities, different to those traditionally considered. It concentrates the edges on the top, left, and diagonal lines of the matrix, leaving the remainder of it empty. Compression can then be achieved, for example, by increasing the size of those zero elements to cover a larger portion of the matrix.

The SlashBurn compression produces adjacency matrices with the distinctive appearance of an arrow pointing to the top-left. With the larger empty spaces on the resulting matrix allowing for better compression.

For the preliminary sorting step, the algorithm starts by defining a permutation of the graph so that nonzero elements of the adjacency matrix are grouped together. It achieves that by removing the nodes with the highest centrality and their incident edges and, as a result, decomposes all nodes on the graph into three groups:

- k-hubset, of top k highest centrality scoring nodes;
- GCC, or nodes belonging to the giant connected component of the decomposed graph;
- spokes to the k-hubset, which are basically nodes not belonging to any of the other two groups.

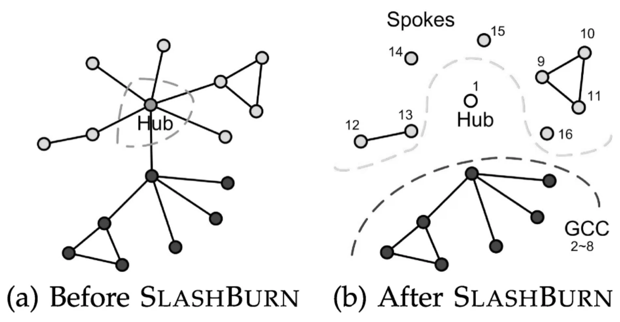


Figure 3.2: SlashBurn decomposition into k-hubset, GCC, and spokes. Adapted from [22]

After the initial iteration, the hub is labeled with the lowest id, the spokes with the highest id in decreasing order of the size of their connected components, while the GCC nodes get assigned the remaining middle-range of ids. The procedure is then recursively applied to the nodes in the GCC group. On the event of there existing more than one giant connected component with the same size, one is chosen at random.

The demanding computational steps can as such be defined as being: selecting the k-hubset, and ordering the remaining connected components. Still, the algorithm has shown to be near-linear to the number of edges on real world graphs. Which is the reason for it being used as part of the algorithm described on the next section.

3.1.2 Vocabulary of Graphs

The article published in 2014 by Koutra *et al.*, *VoG: Summarizing and Understanding Large Graphs*, is not coy regarding its intent of advancing VoG as a means of constructing a 'vocabulary' of commonly occurring subgraph motifs.

The *vocabulary of graphs* algorithm views a set of network motifs as vocabulary terms, which in turn it uses to find the most succinct description of said graph. It measures its degree of succinctness with recourse to the MDL principle, where subgraphs belonging to the established vocabulary that would shorten the graph's summary are added to it.

It presents a three-fold contribution:

- A formulation of the encoding scheme used to choose vocabulary subgraphs;
- The VoG algorithm itself, which efficiently minimizes the description cost, near-linear on the number of edges;
- And, experimental results of this algorithm, performed on multi-million-edge real world graphs, the results of which support its applicability by not only compressing as well as detecting the emergence of interesting patterns.

As elements to the subgraph vocabulary set, the authors selected six of the most commonly occurring network motifs in real graphs. Namely, full cliques and near-cliques (ensuring that the method works for 'caveman' graphs), stars, chains, and full and near bi-partite cores.

3.1.2.1 Encoding Graph Summarization

Given the vocabulary Ω, each structure $s \in \mu$ describes an adjacency matrix A, with possibly overlapping nodes and non-overlapping edges, with μ being the description model of ordered s structures. The structures s describe an area of edges $(i, j) \in A$, $area(s, \mu, A)$ or, simply $area(s)$, for brevity.

Let $C = \cup_x C_x$ be the union of all set C_x containing all possible $x \in \Omega$ structures, model $\mu \in M$ an ordered list of structures, then the model family M is composed of all possible permutations of all subsets of C. Relevant to the MDL principle, the goal is to find the permutation $\mu \in M$ which best balances the complexity of encoding A and μ.

Since μ is an approximation of the matrix A, and the desired summarization is intended to be lossless; error E, taken as the exclusive OR between μ and A, needs to be taken into account. Therefore, the score for a model μ of graph G is given as $L(G, \mu) = L(\mu) + L(E)$, in number of bits that describe the structure.

Depending on the structure being encoded, its description is an optimal prefix code defining it, followed by the description of its elements in accordance to a specific criterion. Error matrix E contains the errors on μ's description of A, seeing as it simpler and more cost-effective to encode approximate structures as belonging to a vocabulary element plus some minor changes, rather than either ignore that near-structure or add it to the vocabulary.

There are clear parallels between this and the general approach at compressing a graph, described at the beginning of this chapter on section 3.1. Explicitly, not also the need to encode corrections to the general vocabulary but, most importantly, that finding a minimum description length entails searching for a proper permutation.

Therefore, after encoding all s structures, a problem then arises due to the sheer size of the search space M, for which an exact solution would need to consider all possible combinations of all candidate structures. And, for that reason, heuristics are employed.

3.1.2.2 VoG Algorithm

Aside from a method for encoding $x \in \Omega$ structures, the list of candidate structures C needs to be instantiated, and the most informative model μ needs to be mined.

Initially, any combination of clustering and community detection algorithms can be applied in the decomposition of the graph into subgraphs. For their experimental results, the authors used SlashBurn.

Following that step, comes the subgraph labelling one. Where the encoding method described in the last section comes into play, for succinctly yet accurately characterize them. First, a subgraph is checked for whether it is a perfect vocabulary match, or not. On the latter case, a new search takes place, for the vocabulary structure that best approximates it, in regards to the MDL and error encoding.

Lastly, heuristics are used for inducing a desired Model μ from the set of candidates in C. The heuristics sorts each candidate by an assigned quality measure, given by the number of bits gained by encoding it rather than noise.

Three heuristics are employed, that evidenced varied results for different networks. So, VoG applies all of them, and afterwards picks the best result out of the three.

- **Plain -** Is a baseline approach, for which $\mu = C$.
- **Top-k -** Selects the best k candidates, in order of decreasing quality.
- **Greedy'nForget -** Sequentially iterates over each candidate in C, including it in μ if doing so would not increase the graph's total encoding cost, otherwise, it is removed. It is the most computational demanding of the three heuristics.

3.1.2.3 Experimental Results and Scalability

Experiments had the goal of performing both quantitative as well as qualitative analysis. Strictly speaking, the amount of compression achieved and structural information gained. Moreover, the algorithm's scalability was also analyzed.

The aim of VoG is primarily that of identifying network motifs that could lead to a better understanding of the network and, as such, compression is more of a means to an end, rather than an important metric.

Depending on network and heuristic chosen, in terms of the number of bits of the compressed graph in relation to that of the original one - *i.e.*, lower numbers equals better compression - it achieved compression levels ranging from as low as 99% to as high as 71%. Some heuristics - namely the *top-k* one, for k of size 10 and 100 - rarely achieved compressions better than 95%. While others, produced more varied results depending on the graph that was being compressed.

In terms of graph summarization with VoG, it is simple to output the number of compressed structures of each type, for each graph and heuristic pairing. This allows to easily quantify the most predominant detected subgraphs, which by itself gives little information as to the mindfulness of those network motifs.

Notwithstanding, after pinpointing those structures, the task of locating the most important among them, on the large network, becomes trivial. And, consequently, so does deriving meaningful interactions or patterns on the graph[1]

As for the matter of scalability, runtime tests were performed that evidenced a near-linear complexity for the number of edges on the graph.

3.2 Quick Discovery of Network Motifs

For this work, many of the given principles throughout up until this point on the document, were applied to the formulation of a shorter description of a large network. One able to impart self-evident knowledge regarding its relevant network motif composition.

As a basis for such composition, an efficient network motif discovery algorithm was employed based on a novel data structure - the g-trie.

The algorithm and abstract data type it is inherently based upon, which was briefly described on section 2.3.3.1, will be further detailed on this section.

3.2.1 G-Trie Definition

As previously stated, the g-trie method for network motif detection is able to outperform existing algorithms, and it does so by at least an order of magnitude. The data-structure corresponds to a tree that encapsulates that of the entire graph, while exploiting common topologies akin to how a prefix tree does for common prefixes. Its name reflects its purpose, **G**raph re**TRIE**val.

The search method is thus able to avoid redundancies by breaking symmetries, and saving both computation time and memory.

Subgraphs on a g-trie are therefore hierarchically organized by the size of common structures, with structures growing in complexity the deeper down the tree we are. This feature saves memory by compressing the topological information of the subgraph collection, and computational time by allowing isomorphism comparison to be performed concurrently while the tree is being built, while also eliminating the need of having to start from the ground up when searching for another subgraph.

Each node on the g-trie represent a single vertex, characterized by its connections to the respective ancestral nodes. In this way, all graphs with common ancestor tree nodes share common substructures, children of a node provide the different topologies that can emerge from a given node, and a path through the tree corresponds to a different simple subgraph.

[1]It might be useful to provide a reminder that VoG fully characterizes each structure, identifying the nodes and edges that constitute it, as well as (potentially, depending on which heuristic is used) its quality.

3.2.2 G-Trie Creation

In order to construct a g-trie, starting from an empty tree, subgraphs are repeatedly added to it, one at a time. Comparable to adding a node to a prefix tree, the g-trie must be traversed until all children nodes are tested for the existence of connections to previous ones, that are similar to the ones on the subgraph being inserted.

In order to prevent redundancies arising from the several possible unique adjacency matrices that can belong to the same isomorphism class, a canonical labeling is enforced. The one employed on g-tries, manages to both serve as an efficient labeling as well as be one that highlights its main properties. That is, by sorting the adjacency, matrix by lexicographically larger order - in simple terms, prioritising ’ones’ before ’zeroes’ while sorting - it not only achieves the goal of being a canonical labeling system, as has been shown to produce more compressed g-tries.

3.2.3 Counting Subgraph Frequencies

The frequency counting algorithm operates by inducing subgraphs of a larger one, on the g-trie. It backtracks though all possible subgraphs while, at the same time, performing isomorphism tests, in order to construct candidate subgraphs.

Subgraphs automorphisms - an isomorphism with itself - would cause the same subgraph to be counted more than once - for each possible automorphism. To prevent this, and the waste of computational time, a set of symmetry breaking conditions are generated for each subgraph and used to prevent it from being counted more than once, rather than simply dividing a final count by the number of automorphism classes at the end. These conditions are in the form of $a > b$, for any pair of vertices with a and b indexes.

These symmetry conditions are stored on the g-trie nodes themselves, alongside the respective subgraphs, and remain true throughout a path. Every time a node expansion is attempted for a new series of nodes with a new vertex, the algorithm tests whether or not there is at least one possible subgraph amongst those that still satisfies the symmetry conditions along that path, before doing so.

However, storing the set of all symmetry conditions for a g-trie node can be costly, memory-wise. As such, the number of conditions stored on a node are reduced as much as possible, by applying filters. Filters which, briefly put, are: the application of transitive properties, relevancy to a particular node, redundancy, or conditions already assured by another one.

On this work, the implementation of this motif detection algorithm, the program *gtrieScanner*, was used to detect all subgraphs of size 3, 4, and 5, of a given graph. An information which was then fed to the designed compression algorithm for contracting all edges of non-overlapping subgraph nodes, that precisely represent the detected subgraphs.

The complete algorithm and its implementation, is described on the following chapter.

Chapter 4

Design & Development

This chapter will address the design and development of the compression program, and underlying algorithms, that were written in an attempt to summarize large networks in terms of their compressed motifs.

The first step in its development, was defining which approach to take for the description of a large network, in an attempt to solve both the problems of compression and summarization. A relation clearly exposed by the VoG algorithm.

The implementation involved writing three distinct programs, and a Bash script that pipelines it with gtrieScanner as well, for ease of use.

The main program, that handles compressing the network by reading the output provided by gtrieScanner, is called Summarize. The program works by implementing a unique graph data structure streamlined for the desired compression. The inputs of this program is the plaintext file containing the graph's original description - as a series of edges connecting node pairs - and the subgraph information for that network given by gtrieScanner. The output is the novel, more succint, graph description.

A second program, called Rebuild, is able to do the opposite. Read the proposed shorter description given by Summarize, and fully restore the original uncompressed network. It is a simple and straightforward program, that serves both as support that the compression is lossless and that reverting it constitutes a trivial matter.

The motivation for developing separate programs to compress and decompress the novel description, was therefore to ensure that restoring the original graph is an agnostic process to compressing it, and that anyone can develop a similar program, without any knowledge of the underlying working of the Summarize algorithm, by simply studying its produced output.

A third separate program called Sorter, simply handles a preprocessing step required by some networks, for ease of analysing the rebuilding effort. It sorts the original network into a canonical form - without affecting the information it contains - that matches that of the output of the Rebuild program. It also eliminates any non-encoding graph characters, such as trailing whitespaces or weight information for weighted networks. That way, verifying that the restored network exactly matches the useful information on the original network, is as simple as running any program that searches and displays differences between two files, such as the *diff* Bash command.

This sorting is not fundamental to neither summarizing nor rebuilding, and was therefore written as a separate small program, so it could be executed at any point during the execution pipeline, in order to just compare and analyze the results of running both aforementioned programs.

Nevertheless, since on the pipeline given by the written Bash script, Sorter is ran before Summarize and Rebuild, an option for also removing repeated edges was included in it, to test for the overhead of compressing networks with self-loops or multiple edges.

The following sections go over each major aspect of the algorithm and the reasonings behind some of its design choices. Section 4.1 details the structure of the chosen graph description, section 4.2 describes the Summarize program, and section 4.3 the Rebuild program.

Relevant listings 4.1, 4.2, 4.3, 4.4, and 4.5, can be found at the end of this chapter.

4.1 Graph description by Network Motif Compression

The proposed novel graph description is written to a plaintext file. This is to balance the playing field when comparing it with the original networks, which were also in plaintext. If compression was the only desired outcome, then writing the output in the form of a compressed object would be the desired approach. But that would both require that any program attempting to read the graph be aware of its data structure, and would inviolate any direct measure in terms of MDL - which is to say, literally how fewer characters were used to write the novel description in comparison to the original.

Starting from a graph described by its set of edges and a list of subgraphs described by a lexicographical ordering of their matrices and node indices, the Summarize program outputs a new graph description with less edges, while being one that also includes a network motif identification and dictionary.

Therefore, the final description, rather than containing a list of homogeneous elements - *e.g.*, vertex pairs - contains a heterogeneity of three distinct elements, depicted on figure 4.1, and defined as such:

- **Dictionary Entry -** Which maps a given key to an adjacency string.
- **Network Motif -** Which precisely characterizes a subgraph by a dictionary key and its constituting nodes.

- **Edge -** That indicates the existence of an edge between two nodes, that is not already expressed in any of the network motifs.

At the head of the network's description is the list of all dictionary entries mapping an adjacency string - the canonical definition for a subgraph as given by the g-trie algorithm - to an arbitrary key. The trade-off of adding the dictionary as a header to the graph's description is immensely advantageous, since the adjacency strings that canonically define subgraph isomorphism classes, are considerably long. Adding them once at the top of the graph's description incurs a penalty to the overall description length that is, however, vastly surpassed by what is gained (literally, reduced) in not having to fully express them on every single occurrence of a network motif.

Otherwise, there would be no gain in terms of MDL of expressing motifs, if every time we do it we have to explicitly state the adjacency matrix for that motif. An even shorter description could be achieved by omitting the dictionary from the graph's description and either print it to a different key file or embed it in code. Still, the goal was to study how much could we gain while still exposing the entire relevant information, and also allowing for anyone to reconstruct the original graph from a single source, without any knowledge of it or how the summarization works.

```
1 2          A 0110100110010110
1 4          A 1 4 2 3
1 5          A 1 7 5 6
1 7          4 7
2 3
3 4
4 7
5 6
6 7
```

Figure 4.1: Regular graph description on the left, versus novel graph description on the right. The regular description contains 27 encoding characters, while the novel description 21, not counting those of the dictionary entry.

This dictionary is in the form of lines containing a $< key, value >$ pair each. The key is used to identify the motifs, and its value is a canonical form of an isomorphism class, as given by gtrieScanner, that constitutes an adjacency matrix on a single line.

Following the dictionary, there are lines representing all the compressed motifs. They start with a dictionary key - a single character - tied to the description of its isomorphism class, followed by the numbering of nodes that makes up that description.

Finally, the remaining lines represent edges connecting a pair of nodes, same as on the original network.

Dictionary entries are lexicographically sorted by key, which is simply given by the compression bias. Motifs are sorted top to bottom by key first, then by natural numerical order, starting from the leftmost node. Edges are also sorted in natural numerical order, left to right and top to bottom.

The same sorting of edges, is what both the Sorter and Rebuild program perform over the original network or when restoring a compressed network, respectively.

Figure 4.1 illustrates the regular versus the novel graph description, which uses less encoding characters. The dictionary entry obviously incurs a hefty overhead for small networks, and the amount of clustering on the network directly impacts compression as well.

One important note, is that because motifs are in a canonical form that does not always match the natural ordering of nodes, the nodes expressed on the motif entries are ordered as to match the explicit adjacency matrix given by the dictionary entry, instead of being ordered from lowest to highest.

This entails the computation of a permutation for those subgraph nodes on the network, that matches the given matrix string. Nevertheless, given that automorphisms can occur in subgraphs, allowing for an implementation that will stop the matching process as soon as the first suitable match is found, and that the largest detected subgraph has at most size five, this computation is never too demanding.

4.2 Building a Graph for Compression

The gtrieScanner program is first run over the network, for subgraphs of size three, four, and five. Generating dump files containing all detected occurrences of those subgraphs in the already described manner of adjacency string followed by a series of nodes that, while forming that subgraph, are not necessarily ordered according to the adjacency string given.

The written program only works for undirected graphs, reading and building the graph one edge at a time from its description, and then reading the entire output of discovered subgraphs given by gtrieScanner. For the purposes of the summarization algorithm, the list of subgraphs is called the workload. Each candidate subgraph is an instance of compression work that could potentially be done.

The graph data structure is implemented as a hash table based implementation of a map of nodes, tied to the node key, for constant time retrieval. A node's label is used to retrieve it from memory. The graph also contains the workload list of compression candidates, and a subgraph dictionary map with iteratively incremented keys, as new subgraphs are added - but only during the compression stage.

Also stored in memory, are a set of selection bias attributes measured from the network, that are used when prioritizing work, and another hash table of already contracted edges. Also present is a list of repeated edges and self-loops, that are not included in the graph data structure so as to not affect or be affected by the compression algorithm.

Despite internally, all nodes being the same, they can be split into two groups, lower-level and higher-level nodes. Higher-level nodes or supernodes, represent higher levels of abstraction, in which a node is actually a subgraph of other nodes. The lowest-level nodes being those of the original graph, which haven't suffered any compression. These nodes are called downstream and upstream nodes, respectively. With upstream nodes being closer to the source, *i.e.*, the original uncompressed graph, and downstream nodes being closer to the final compressed graph.

Figure 4.2 illustrates this concept. Of note that while the graph accepts multiple layers of compression, this is not something that is employed by the compression algorithm on this current version.

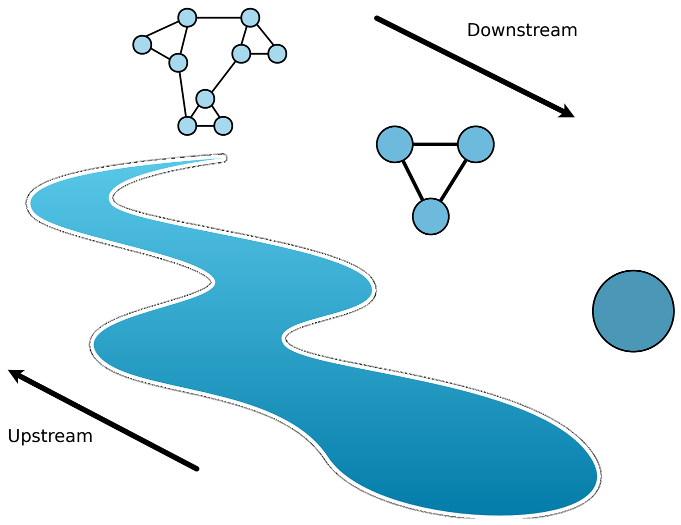


Figure 4.2: Visual interpretation of compression pointers and the reasoning behind calling them upstream and downstream pointers. Downstream pointers point to a supernode, while upstream pointers points to its sub-nodes. The further upstream, the closer we are to the source - *i.e.*, the original graph. Downstream nodes are higher-level nodes representing compressed motifs, obtained via what we call node contraction. A code implementation of an edge contraction.

Each node on the graph therefore possesses a string label, which not only identifies it, but also whether it represents an uncompressed or a compressed node. With the labels of compressed entries matching the final intended description: a dictionary key, followed by its correct ordering of sub-nodes. A node can contains a single downstream pointer and an array of upstream pointers. In that way, the easiest manner for the program to determine if it is looking at a simple or a compressed node, is to look at its downstream or upstream pointers.

Compressing a subgraph into a node, simply involves adding a new downstream node to the graph which points to the upstream nodes that it is replacing. And, in turn, setting its upstream nodes' downstream pointers to it.

4.2.1 Handling Edge Deletion and Update

At a higher level, compression of a motif is achieved by contracting all the edges on that motif, as detailed on figures 4.3 and 4.4.

Edge contraction entails merging the nodes to which it was originally incident to. Seeing that the summarization algorithm performs several edge contractions over entire subgraphs, thus merging several nodes into one, and that our graph is a hash table of nodes, the focus of our approach will be on merging nodes and not removing edges.

For this reason, the term node contraction will be used synonymously with motif or node compression, as well as a counterpart to, and interchangeably with, edge contraction.

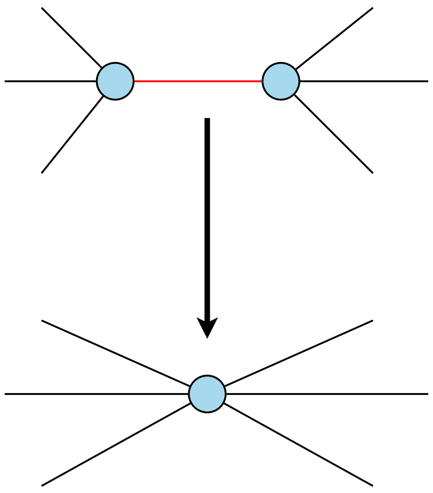


Figure 4.3: Illustration of the process of edge contraction, in which an edge is removed from the network, resulting in the two nodes it previously connected being merged into one. All other incident edges to those two nodes are now incident to the merged node.

At a lower level, what is happening is that a new node is being added to the graph, and the appropriate upstream and downstream nodes between it and the contracted nodes it now represents, are set accordingly. Figure 4.5 illustrates the actual procedure taking place.

Contracting a network motif into a single node implies the removal and resetting of several edges. Contracted edges are removed from the graph, and all incident edges to a contracted node are now incident to the newly formed supernode.

Actually implementing these modifications at a lower level, is computationally wasteful. The adjacency lists of every node to be contracted has to be iterated and compared against the edges to be deleted, so it takes $E \times e$ time for E number of total edges in all adjacency lists of candidate nodes and e edges in the network motif to be compressed. Then, after aggregating what's left of those adjacency lists as the adjacencies of the new contracted node, and since the graph is undirected, we need to visit all of its neighbours and update the edges on their adjacency lists to now point to the supernode. Which takes $E \times n$, for E being the entire set of edges on all neighbouring adjacency lists, and n the number of neighbouring nodes.

Furthermore, the above stated is aggravated by the fact that, the original edge incidence information needs to be stored, if it is ever intended to be restored after compression. For example, if node w is contracted, amongst others, into supernode W; saying that node v is now connected to supernode W instead of w, constitutes a loss of information.

Fortunately, this all becomes superfluous, and therefore trivialized, by taking advantage of the unique downstream and upstream pointers.

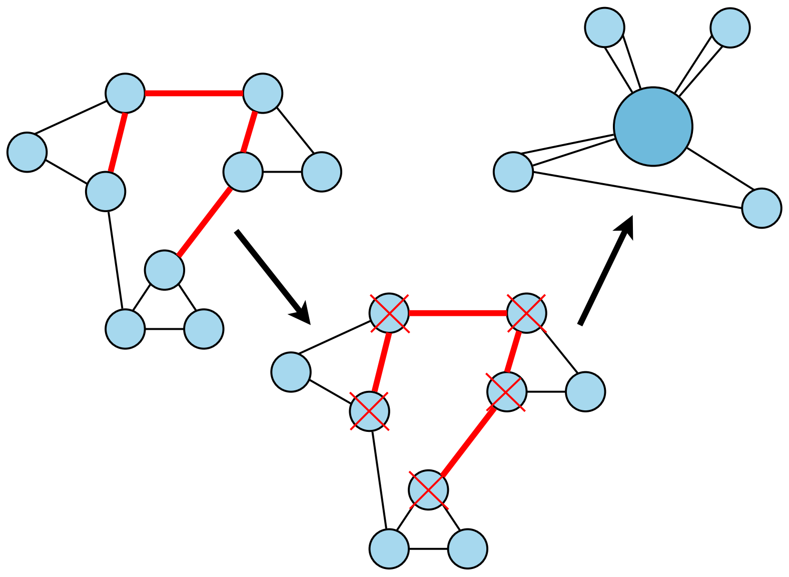


Figure 4.4: Illustration of edge contraction over the entire set of edges of a network motif of size five. Crossed nodes are merged into the supernode shown on the last step.

Whenever an edge is contracted it is marked as such. Then, fetching the adjacency list of a supernode is the same as fetching the adjacency lists of all it's sub-nodes - *i.e.*, upstream nodes. And, reaching a supernode is the same as reaching one of these upstream nodes.

By simply taking into account whether we are looking at a simple or a supernode, before retrieving adjacency lists or determining on which node we are on the higher level network, there is no need whatsoever to update the adjacency lists of any node. Instead, as stated, we simply add a new node, set pointers to and from it, and mark the edges as being contracted. All done in constant time.

Since nodes retain adjacency information regarding the simple nodes, and not super ones, it is possible to both retrieve the incidence to the upstream simple node - the information that is stored in memory - and the downstream supernode - by following the downstream pointer.

This fact, discards the need to use any sort of error set, to later compute original edge information when needed.

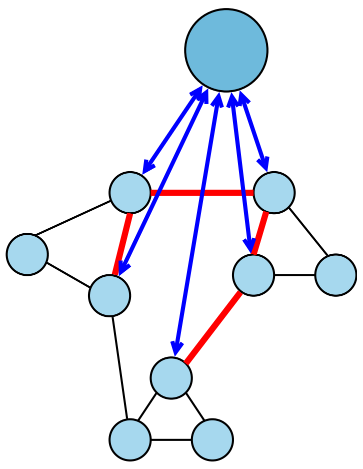


Figure 4.5: Actual implementation of the edge/node contraction of figure 4.4. No edge or node is actually removed from the network, and a new supernode is added. Upstream and downstream pointers, represented as blue bi-directional arrows, are set accordingly.

4.2.2 Workload Sorting and Contraction Validation

As mentioned, the workload is the list of network motifs detected by gtrieScanner, that is then used by the Summarize to compress the network. But because gtrieScanner performs a comprehensive subgraph search, a single node may be indicated as belonging to several different subgraphs, several times, in several different structures.

As such, for any given node indicated as being part of multiple subgraphs, a selection of which of those subgraphs to compress must be made. And, as was already extensively demonstrated by the related work described on the previous chapter, finding an exact minimum cost for such compression is no trivial matter.

Explicitly, one would need to generate multiple compressed graphs, one for each possible combination of non-overlapping subgraphs, and then measure and output the one that produces the shortest description. This implies an absurd amount of permutations of an immensely huge number of combinatorial compressions. Since that is clearly not a feasible computation, a (deterministic) heuristic approach was deemed to be the only valid route to take.

On a naive compression, without any kind of selection process, the program simply looks at each subgraph as a compression candidate, in the order in which they are read. Every time a compression candidate is not validated for contraction, it is simply skipped.

The employed heuristic techniques do exactly the same, but attempt to induce a selection bias on the order in which compression candidates are validated.

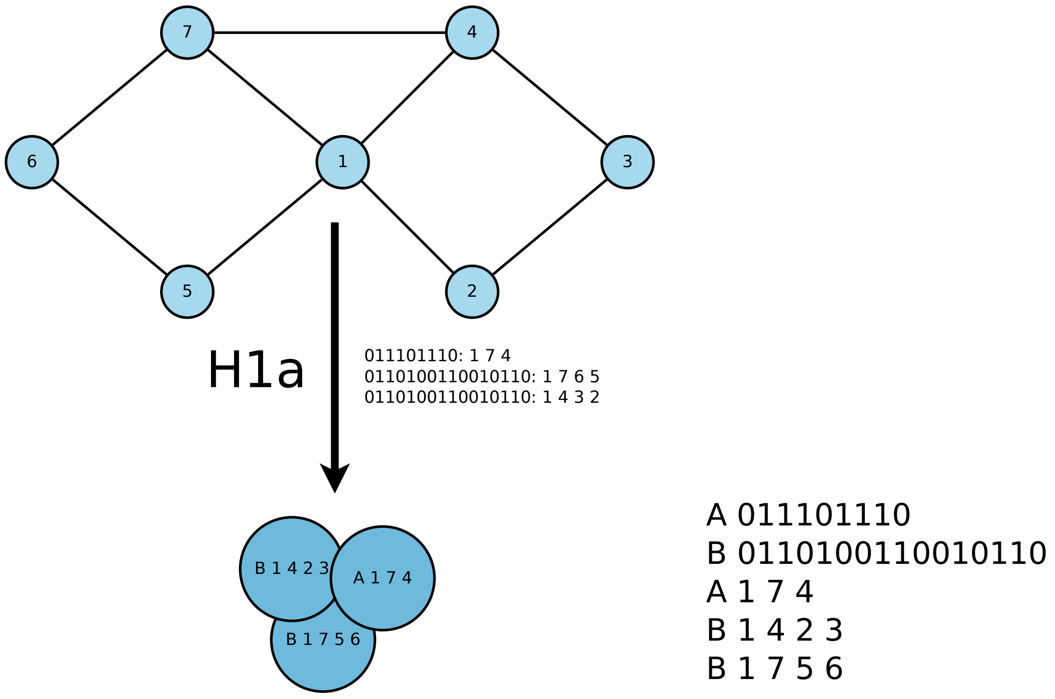
An edge cannot be contracted more than once and, as such, the order in which network motifs are compressed is inducing of the compression effectiveness. Motifs are only compressed if doing so would reduce the MDL of the final graph, on a case-by-case basis. Therefore, there is no way of knowing whether performing a good compression now, will block a more lucrative compression down the line.

With this in mind, three heuristics were tested, each in ascending and descending order, that sort the entire workload via an established criterion related to a chosen network property. The chosen properties were purposed to have some relation to the motif prevalence and their distribution and impact on the network.

The first heuristic approach, sorts the workload by the frequency of motif occurrences, aiming to determine whether motif prevalence has any impact on compression. The second approach, sorts them by number of edges inside a subgraph, therefore either prioritizing the largest or smaller number of contracted edges. The third and final heuristic, sorts the workload by the prevalence in which every node on a subgraph, also appears on other subgraphs. Therefore, either prioritizing more or less central nodes.

An initial experiment to quantify the value of compressing only non-overlapping subgraphs over that of compressing every single one detected by gtrieScanner was performed, and is fully detailed on section 4.2.3. Suffice to say at this point that - as it pertains to whether or not to apply a heuristic for non-overlapping edge contraction - the results of that experiment were graph descriptions that, not only did not possess a shorter description length, as had lost all of the meaning regarding network motifs.

The conclusion from that previous experiment was that contracting every single node multiple times, in multiple layers, alongside multiple contractions of the same edges, was not a feasible approach towards approaching an MDL. Still, we were able to refine a different method that allows for multiple contractions of the same node, on the same layer, without multiple contractions of the same edge. This method further reduced the final description length.



```
A 011101110
B 0110100110010110
A 1 7 4
B 1 4 2 3
B 1 7 5 6
```

Figure 4.6: Example of compression for the given toy network, heuristic (h1a), and motif file. Since the heuristic prioritizes validating compression on less frequent subgraphs, the motif *'A 1 7 4'* will be compressed first, marking its edges as contracted. Next, compressing motif *'B 1 4 2 3'* produces an equal length description than describing the not contracted edges *'1 5'*, *'5 6'*, and *'6 7'*. Lastly, motif *'B 1 7 5 6'* is also compressed for the same reason. Notice how there is no edge between supernodes. Instead, they merely overlap in our representation. On the output, there is no need to specify a connection between those motifs since the edges are implicit in their description.

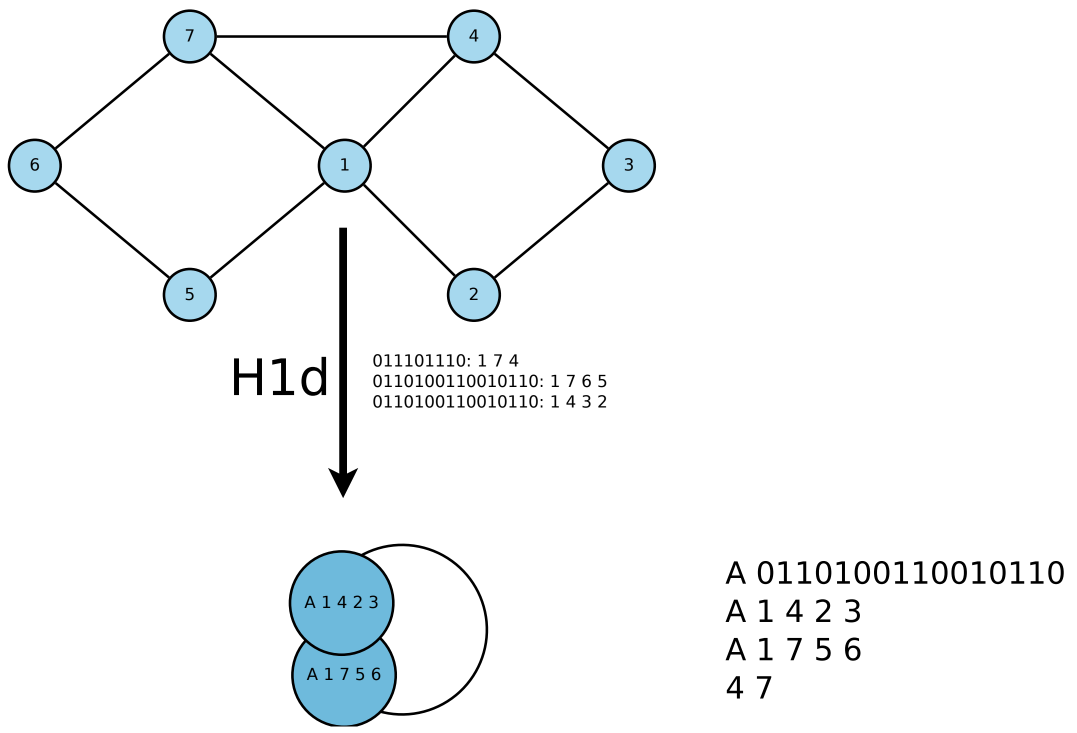


Figure 4.7: Example of compression for the given toy network, heuristic (h1d), and motif file. Only difference in input from figure 4.6 is the chosen heuristic, which prioritizes contraction validation for the most frequent motifs. Motif *'A 1 4 2 3'* and *'A 1 7 5 6'* are both contracted, in turn. This is because describing both motifs is less costly than describing the individual four edges for each motif. Finally, motif *'B 1 7 4'* is not validated for compression - and the key and canonical matrix not added to the output dictionary - since it is less costly to just output the remaining uncompressed edge *'4 7'* than the entire motif. The resulting network can be represented as two overlapping supernodes with an edge between them. This heuristic produced a better compression than that of figure 4.6 for these inputs.

Independently of which sorting method is being applied, contraction is only performed if by doing so, and given the current state of the network, describing the compressed motif is less or equally costly than describing the individual uncompressed edges - compression takes priority for the same cost.

Notice that this method takes into account which edges were already compressed, and excludes them from the computation. But allows for the repetition of nodes, if doing so leads to a cost reduction.

Contraction validation, therefore, simply entails summing the cost of describing all yet not contracted edges and comparing it to the cost of describing the compressed motif. If the latter has a smaller or equal cost, which is to say produces a shorter or equal description length, then the motif is compressed and the (not yet contracted) edges marked as having been contracted.

Figures 4.6 and 4.7 depicts the process of contraction validation, as well as how the heuristic sampling bias works and also how it still affects it. Those figures illustrate an actual execution and the output produced when running the program with those inputs and chosen heuristics.

4.2.3 The Folding Problem

As mentioned, the nature of the downstream and upstream pointers allows nodes to be able to continuously point to other nodes further down or up the compression layer. That is to say, a node that is downstream to another, can be upstream to a third one.

Due to this, a naive brute-force full compression was attempted, in order to verify and analyze the results. By full compression, it is meant that no entry on the graph's workload was skipped, and that a subgraph S can be folded into another subgraph S', in lieu of one or more of its nodes and edges that are a part of S'.

Due to the fact that gtrieScanner is extensive on its exploration of subgraph motifs, and without a bias criterion for discarding some motifs in favor of another, the above-mentioned compression approach, as was initially suspected, produced graphs with few but heavily compressed nodes. These nodes, in turn, themselves containing multiple levels of other compressed nodes in multiple combinations. This resulted in an excessively long subgraph description for every single subgraph, albeit at a short number of motif and edge entries for the full graph. Number of dictionary entries remained comparatively the same, surprisingly.

The trade-off between having less lines on the graph's description, but those lines having a longer length (by several orders of magnitude), weighted heavily on the side of the longer lengths. The descriptions produced in this way, not only took up more characters, as they lost any meaningful interpretation. They consisted of few lines with dozens or hundreds of labels muddled by several layers of compression, where it was difficult to even understand where one ended and the other began.

In a sense, reinforcing the relevance of the MDL principle to this work, by opposing it.

This short experiment also served to reinforce the notion that a selection of which structures to compress, and how to compress them, was more valuable than simply trying to arbitrarily compress every motif. And regarding future work, if there is any value to be found in allowing for multiple layers of compression, those most likely need be done iteratively, while still applying some specific selection principle at each compression layer.

4.2.4 Compressing the Graph

The pseudocode 1 details the most important steps of the Summarize algorithm. Some of the implementation can be reviewed on listings 4.1, 4.2, 4.3, 4.4, and 4.5.

After reading the inputs, the graph is built while storing repeated edges and self-loops on a separate list, and the workload is built while extracting the network properties that will be used for the heuristics.

For very large networks, some motif files generated by gtrieScanner are of dozens or even hundreds of Gigabytes in size and, as such, cannot be loaded into the memory of an average machine. The Summarize program can be effectively - but never efficiently, due to the bottleneck of disk read/write operations - run with an external sorter method, that allows for the heuristic selection bias to be applied to the workload.

On the compression function, the workload is sorted with a given comparator specific for the chosen heuristic. As stated, each of the three heuristics can sort the workload in either ascending or descending order. A second different heuristic can also be chosen as tiebreaker during comparisons. This leads to a total of eighteen possible comparators for sorting.

As described on previous sections, the workload is iterated, at each iteration compression is validated, and the nodes contracted or not, in accordance with the validation result.

While gtrieScanner outputs the matrix in canonical form, the nodes belonging to a network motif are not necessarily on the order that matches that canonical string, but in the order that they were read. Therefore, an additional step of obtaining a correct permutation of nodes that matches that canonical adjacency matrix description is required.

After iterating through the possible permutations for the given compression candidate nodes, and returning a valid one as soon as the first match is found, the validation proper can be performed.

The method for validating compression proceeds as previously described. Edges belonging to the canonical matrix and a valid permutation of nodes for that motif, are tested whether they were already contracted or not.

Each not contracted edge adds its characters, including whitespaces, to the description cost. Then that value is simply compared to the cost, in characters as well, of describing the given motif. If describing the latter has an equal or lesser cost than describing the former, the nodes are contracted.

Compressing a motif is simply adding its new node to the graph, with appropriate pointers, as previously described. Edges are marked as having been contracted as well, for further description cost calculations involving them.

After the whole workload has been visited, the output is generated. Which is done by visiting every node, checking whether it represents a motif or a simple node, and adding the motifs to a list for printing. Every node's adjacency list - or those of upstream nodes to a supernode - is checked for edges not yet added to the printing list as well.

Finally, the stored repeated edges and self-loops are added to the entire list and it is then sorted according to the desired output of the novel graph description, already described in section 4.1, and outputted.

With this algorithm and its implementation, it was possible to create shorter and (hopefully) more meaningful descriptions of large networks, in relation to their motifs. All the while, ensuring that the compression was lossless, as demonstrated by the Sorter and Rebuild programs, described on the following section.

4.3 Restoring the Uncompressed Graph

The Sorter program simply applies the same sorting principle that the output of the Summarize and Rebuild programs establish. Edges are sorted in ascending numerical order, from left to right and top to bottom.

It does this by reading the original network file, reading only the first two numerical entries on each line - thus bypassing weights for weighted graphs - and adding them to a list for printing.

As described, it can also detect and bypass repeated edges, for testing purposes.

The list is then sorted, and outputted.

Neither sorting the list, removing repeated edges and self-loops, nor bypassing weights, is required for the successful execution of the Summarize program. The Sorter program only exists to ensure that the original networks is represented on the same fashion as the restored one will, to accommodate for a simple validation on whether or not the restored network exactly matches the original one.

The Rebuild program is also very simple and straightforward. It reads the compressed graph's description one line at a time. First, storing the dictionary entries to a hash table. Afterwards, once motifs start appearing, it adds the edges as described by the adjacency matrices on the stored aforementioned dictionary. Finally, it adds the remaining edges, not described in motifs, to the list as well.

The list is sorted in the same manner as with the Sorter program, and the whole data it contains, is outputted.

Algorithm 1 Summarize

```
    Input $G = (V, E)$, $W$, $H$.                                   ▷ Graph, Workload, Heuristic
    Output $G' = (D, M, E')$                                         ▷ Compressed Graph
1: procedure ValidateEdgeContraction($w_{Matrix}, w_{Subnodes}$)
2:     $E_{NotContracted} \leftarrow \emptyset$
3:     $C \leftarrow 0$                                              ▷ Description Cost
4:     for $e \in w_{Matrix}$ do
5:         if $e \notin E_{Contracted}$ then
6:             $E_{NotContracted} \leftarrow e$
7:             $C \leftarrow e$
8:         end if
9:     end for
10:    if $C < C_{M \leftarrow w_{Subnodes}}$ then
11:        return false
12:    end if
13:    for $e \in E_{NotContracted}$ do
14:        $E_{Contracted} \leftarrow e$
15:    end for
16:    return true
17: end procedure
18: procedure CompressOne($V'_{Subnodes}$)
19:    $M \leftarrow V'_{Subnodes}$
20:    $G' \leftarrow M$
21: end procedure
22: procedure CompressAll($W, H$)
23:    $D \leftarrow \emptyset$                                      ▷ Dictionary
24:    $W \leftarrow$ Sort($W, H$)
25:    for $w \in W$ do
26:        $V'_{Subnodes} \leftarrow$ Permutation($w_{matrix}, w_{Subnodes}$)
27:        if ValidateEdgeContraction($V'_{Subnodes}, w_{Matrix}$) then
28:            $D \leftarrow w_{Matrix}$
29:            CompressOne($V'_{Subnodes}$)
30:        end if
31:    end for
32: end procedure
33: procedure Main
34:    $E_{Contracted} \leftarrow \emptyset$
35:    CompressAll($W, H$)
36:    Print($G' \leftarrow (D, e \notin E_{Contracted})$)
37: end procedure
```

4.3.1 The Execution Pipeline

The written bash script, simply automates running one program after the other, and directing outputs from one as inputs to another. It also runs a series of terminal commands to output various execution results, such as the sizes of the generated files, execution times, compression metrics, compression validation, etc.

The main execution pipeline, in terms of the described programs, consists of running gtrieScanner (or not, if a motif file is already provided) first to generate motifs for compression; running the Sorter program second, just to preemptively eliminate any noise in the data; directing sorted network and motifs to the Summarize program; and finally, redirecting its output to the Rebuild program, and checking if the output of this program matches that of the sorted network.

The next chapter, describes and analyzes the results of running this execution pipeline, with a variety of different parameters, on a set of test case networks.

```java
//. . .

   // method that iterates through sorted workload and summarizes the network
   void compressAll(String motifFile, int[] heuristic, boolean useExternalSort){

   Comparator<String> workloadSorter = getHeuristicSorter(motifFile, heuristic);

   if(!useExternalSort){
       workload.sort(workloadSorter);

       for(String work : workload){
      String[] line = work.split(": ");
      String key = line[0];
      String[] list = line[1].split(" ");

      int size = list.length;
      String[] aux = key.split("");
      String[][] matrix = new String[size][size];
      int pos=0;

      for(int i=0; i<size; i++){
          String[] row = new String[size];
          for(int j=0; j<size; j++)
         row[j] = aux[pos++];
          matrix[i] = row;
      }

      Node[] subNodes = getCorrectPermutation(list, matrix);

      // easy debug for null pointer caused by permutation not found
      // print occurs right before exception is thrown
      // should never happen with gtrieScanner
      // only on manually created motif file with incorrect motif matrix
      if(subNodes == null)
          pl("This shouldn't have happened\nprobable cause, motif file is
              incorrect");

      // compress if edge contraction results in lesser cost
      if(validateEdgeContraction(matrix, subNodes)){
          // create new dictionary entries as needed
          if(!subgraphDict.containsKey(key))
         subgraphDict.put(key, index++);

          compressOne(subgraphDict.get(key), subNodes);
      }
       }
   }else{

      //. . .

        }
    }

//. . .
```

Listing 4.1: Java method for graph compression

```java
//. . .

  Node[] getCorrectPermutation(String[] list, String[][] matrix){

  ArrayList<Node[]> perms = new ArrayList<>();
  int size = list.length;

  findAllPermutations(list, matrix, 0, new boolean[size], new int[size], perms);

  for(Node[] perm : perms)
      if(isCorrectPermutation(perm, matrix))
     return perm;
  return null;
   }

   void findAllPermutations(String[] list, String[][] matrix,
             int cur, boolean[] used, int[] perm, ArrayList<Node[]> perms){
  int size = list.length;
  if(cur == size){
      Node[] base = new Node[size];
      for(int i=0; i<size; i++)
     base[i] = nodes.get(list[perm[i]]);
      perms.add(base);
  }

  for(int i=0; i<size; i++)
      if(!used[i]){
     used[i] = true;
     perm[cur] = i;
     findAllPermutations(list, matrix, cur+1, used, perm, perms);
     used[i] = false;
      }
   }

   boolean isCorrectPermutation(Node[] perm, String[][] matrix){

  for(int i=0; i<perm.length; i++)
      for(int j=0; j<perm.length; j++){
     if(i!=j)
         if((matrix[i][j].equals("1") && !perm[i].adj.contains(perm[j]))){
       //||            (matrix[i][j].equals("0") &&
           perm[i].adj.contains(perm[j]))){
       return false;
         }
      }

  return true;
   }

//. . .
```

Listing 4.2: Java method for subgraph node permutation

```java
//. . .

  boolean validateEdgeContraction(String[][] matrix, Node[] subNodes){

  String snodeForm = "S";
  for(int i=0; i<subNodes.length; i++)
      snodeForm += " "+subNodes[i].key;
  int snodeCost = snodeForm.length();

  String edgeForm = "";
  LinkedList<String> relevantEdges = new LinkedList<>();

  for(int i=0; i<matrix.length; i++){
      for(int j=i; j<matrix.length; j++){
     if(matrix[i][j].equals("1")){
         Node v = nodes.get(""+subNodes[i].key);
         for(Node w : v.adj)
        if(w.key.equals(""+subNodes[j].key)){
             Node l,r;
             if(subNodes[i].key.compareTo(subNodes[j].key) < 0){
            l = subNodes[i];
            r = subNodes[j];
             }else{
            l = subNodes[j];
            r = subNodes[i];
             }
             if(!contractedEdges.getOrDefault(l.key+" "+r.key, false)){
            String relevantEdge = l.key+" "+r.key;
            relevantEdges.add(relevantEdge);
            edgeForm += relevantEdge;
             }
        }
     }
      }
  }

  int edgeCost = edgeForm.length();

  //no need to compress
  if(edgeCost < snodeCost)
      return false;

  for(String relevant : relevantEdges)
      contractedEdges.put(relevant, true);

  return true;
   }

//. . .
```

Listing 4.3: Java method for validating edge contraction

```java
//. . .

  void printUpstreamEdges(Node v){

  if(v.upstream.size() > 0)
      for(Node w : v.upstream)
     printUpstreamEdges(w);
  else
      for(Node w : v.adj)
     printEdge(v.key, w.key);
   }

  void printEdge(String a, String b){

  String e = a+" "+b;
  if(a.compareTo(b) > 0)
      e = b+" "+a;
  if(!usedEdges.getOrDefault(e, false)){
      usedEdges.put(e, true);
      if(!contractedEdges.getOrDefault(e, false)){
     sortedNodes.add(e);
      }
  }
   }

  void createOutput(){

  usedEdges.clear();

  // print the motif dictionary
  Map<String, String> keys = new HashMap<>();
  List<String> orderedKeys = new ArrayList<>(subgraphDict.size());
  for(String k : subgraphDict.keySet()){
      String v = ""+subgraphDict.get(k);
      orderedKeys.add(v);
      keys.put(v, k);
  }

  Collections.sort(orderedKeys);

  for(String k : orderedKeys)
      pl(k+" "+keys.get(k));

  // add nodes and edges to print list for sorting
  for(String k : nodes.keySet()){
      Node v = nodes.get(k);
      //compressed node, print its key and every outgoing edge
      if(v.upstream.size() > 0){
     sortedNodes.add(v.key);
     printUpstreamEdges(v);
      }else{
     //simple node, print edges
     for(Node w : v.adj)
         printEdge(v.key, w.key);

     }
      }

//. . .
```

Listing 4.4: Java method for generating output - first half

```java
//. . .

   // add multiedges to print list
   sortedNodes.addAll(multiEdges);

   // print list sort
   sortedNodes.sort((k1, k2) ->
          (Character.isLetter(k1.charAt(0)) ?
           (Character.isLetter(k2.charAt(0)) ?
            (Character.compare(k1.charAt(0), k2.charAt(0)) != 0 ?
             Character.compare(k1.charAt(0), k2.charAt(0)) :
             Arrays.compare(Arrays.asList(k1.substring(2, k1.length()).split(" "))
                  .stream().mapToInt(Integer::parseInt).toArray(),
                  Arrays.asList(k2.substring(2, k2.length()).split(" "))
                  .stream().mapToInt(Integer::parseInt).toArray())) :
            -1) :
           (Character.isLetter(k2.charAt(0)) ?
            1 :
            Arrays.compare(Arrays.asList(k1.split(" "))
                 .stream().mapToInt(Integer::parseInt).toArray(),
                 Arrays.asList(k2.split(" "))
                 .stream().mapToInt(Integer::parseInt).toArray())));


   // print list print
   for(String n : sortedNodes)
       pl(n);
    }

//. . .
```

Listing 4.5: Java method for generating output - second half

Chapter 5

Testing and Analyzing Graph Compression

With the intention of eliminating bias that could skew the observations towards a specific result, the Summarize program was tested on six real world networks, semi-picked at random for their distinct topologies, attributes and classifications. The only information used on their selection was their size, number of edges, and category. Although the program is not ready to handle directed graphs, any directed graph can be assumed to be an undirected one with possible multiple edges, for the purpose of testing compression.

Several other networks were experimented on while writing and debugging the program, none of them were used on the actual testing stage, and no compression metric was taken during development. Only algorithm validation was tested during the development stage.

The networks used for testing and analysis, are available online on *The KONECT Project* [21], and are described on section 5.1. They are the following: Air Traffic Control [2], American Revolution [12], A Song of Ice and Fire [24], Euroroads, Network Science [1], and Pretty Good Privacy [4].

The compression results, are analyzed and discussed on section 5.2. Section 5.3 endeavours to relate the results to meaningful network metrics.

5.1 Test Case Networks

These real world networks are ordered by size. Following their description and until the end of this chapter, they will be referred by an assigned label, for brevity, as stated on their descriptions.

5.1.1 American Revolution

- **label -** *revolution.*

- **Size -** $n = 141$.
- **Edges -** $e = 160$.
- **Diameter -** $\delta = 6$.
- **Category -** Affiliation Network.
- **Node Meaning -** Person, organization.
- **Edge Meaning -** Membership.
- **Network Format -** Bipartite, undirected.
- **Edge Type -** Unweighted, no multiple edges.

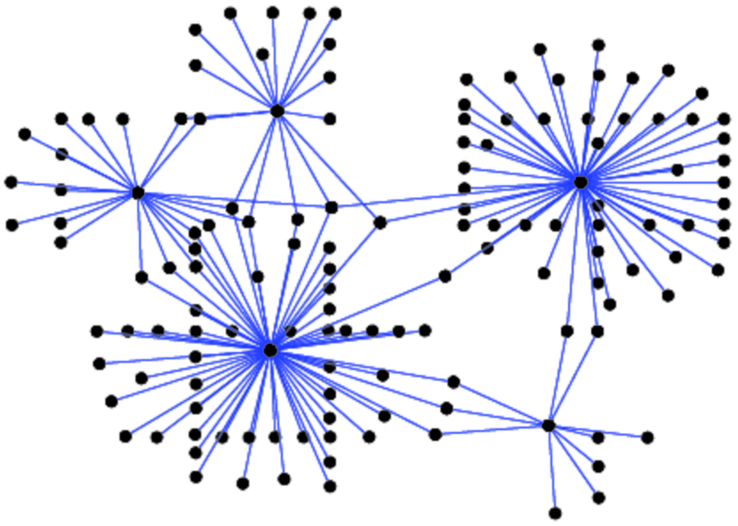


Figure 5.1: The American Revolution network.

A small network was picked at random to determine the effectiveness (or rather, ineffectiveness) of the compression algorithm over them, given that the motif dictionary entails a considerable overhead for smaller networks.

Despite being described as undirected with no multiple edges, repeated edges were found during compression. Meaning that one or the other of its described attributes is incorrect.

5.1.2 A Song of Ice and Fire

- **label -** *asoiaf*.
- **Size -** $n = 796$.
- **Edges -** $e = 32{,}629$.

- **Diameter -** $\delta = 9$.
- **Category -** Miscellaneous.
- **Node Meaning -** Character.
- **Edge Meaning -** Co-appearance.
- **Network Format -** Unipartite, undirected.
- **Edge Type -** Unweighted, multiple edges, does not contain loops.

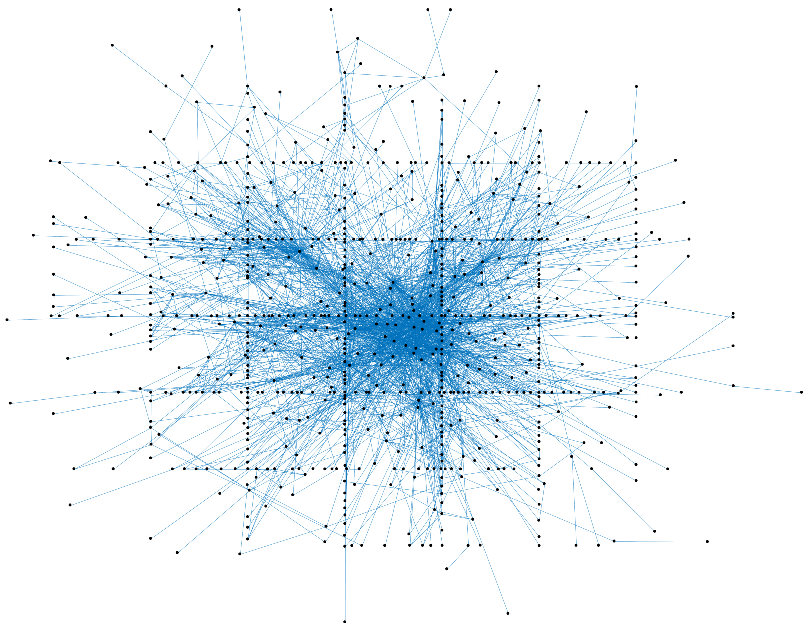


Figure 5.2: The A Song of Ice and Fire network.

This network was chosen both due to its size and volume ratio, and the fact that it serves as a social network of sorts. It was deemed relevant to have a test case where the number of edges were considerably larger than that of its nodes.

Even though this is the second smallest network in size, it is by far the largest in number of edges.

5.1.3 Euroroads

- **label -** *euroroads.*

- **Size -** n = 1,174.
- **Edges -** e = 1,417
- **Diameter -** δ = 62.
- **Category -** Infrastructure network.
- **Node Meaning -** City.
- **Edge Meaning -** Road.
- **Network Format -** Unipartite, undirected.
- **Edge Type -** Unweighted, no multiple edges, does not contain loops.

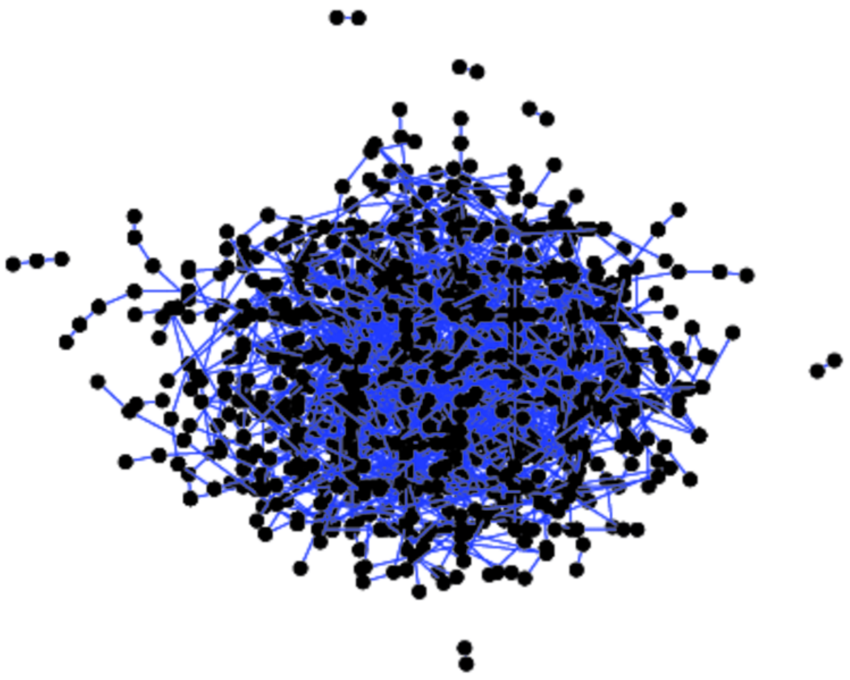


Figure 5.3: The Euroroads network.

The network was chosen for its near-similar number of nodes and edges and with the objective to test compression on a real-world topography network.

5.1.4 Air Traffic Control

- **label -** *air control.*
- **Size -** n = 1,226.
- **Edges -** e = 2,615.

- **Diameter -** δ = 17.
- **Category -** Infrastructure network.
- **Node Meaning -** Airport/service center.
- **Edge Meaning -** Preferred route.
- **Network Format -** Unipartite, directed.
- **Edge Type -** Unweighted, no multiple edges, contains reciprocal edges, self-loops, directed cycles.

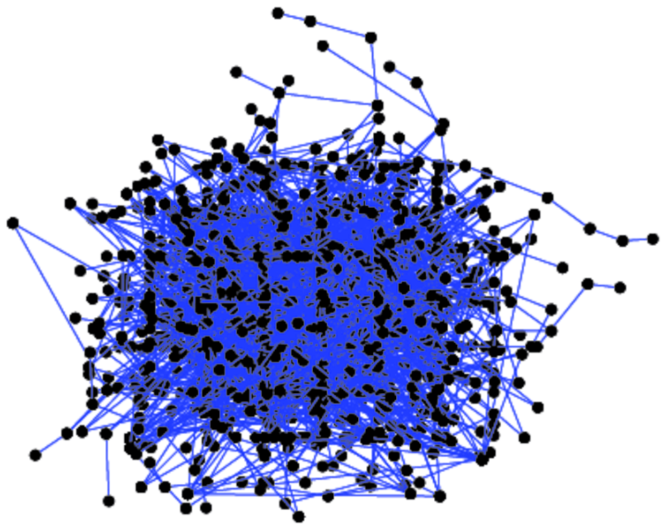


Figure 5.4: The Air Traffic Control Network.

A network with a moderate number of nodes and edges, with several traits that distinguishes it from the others, namely the prevalence of loops and cycles. As previously stated and contrary to its description, despite originally being a directed graph, for the purposes of this study it was treated as an undirected graph with multiple edges.

5.1.5 Network Science

- **label -** *netsci*.
- **Size -** n = 1,461.
- **Edges -** e = 2,742.
- **Diameter -** δ = 17.
- **Category -** Co-authorship network.
- **Node Meaning -** Author.

- **Edge Meaning -** Co-authorship.
- **Network Format -** Unipartite, undirected.
- **Edge Type -** Unweighted, no multiple edges, does not contain loops.

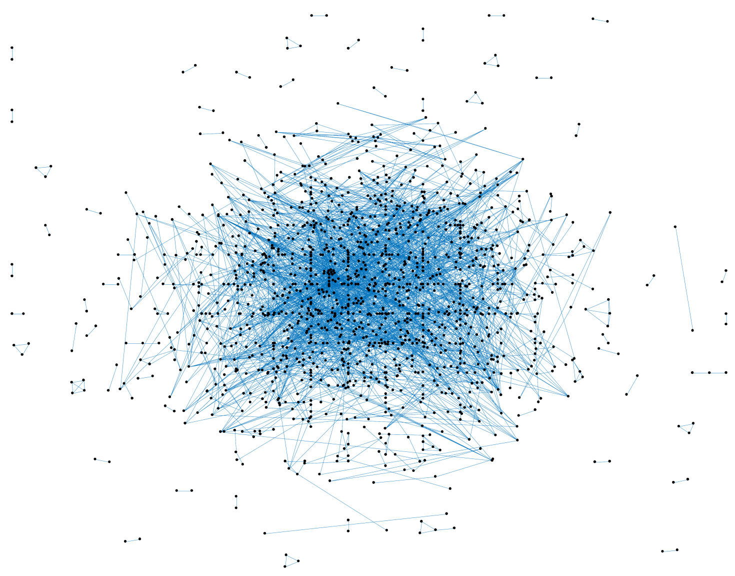


Figure 5.5: The Network Science network.

The criteria for selecting this network as a case study were its size, and the near double number of edges. Similar in size and number of edges to the previous network, but little else, the goal was to analyze how the differences between these two networks would translate in terms of summarization.

5.1.6 Pretty Good Privacy

- **label -** *pgp*.
- **Size -** n = 10,680.
- **Edges -** e = 24,316.
- **Diameter -** δ = 24.
- **Category -** Online contact network.
- **Node Meaning -** User.

- **Edge Meaning -** Interaction.
- **Network Format -** Unipartite, undirected.
- **Edge Type -** Unweighted, no multiple edges, does not contain loops, is a snapshot of only the largest connected component of the original data.

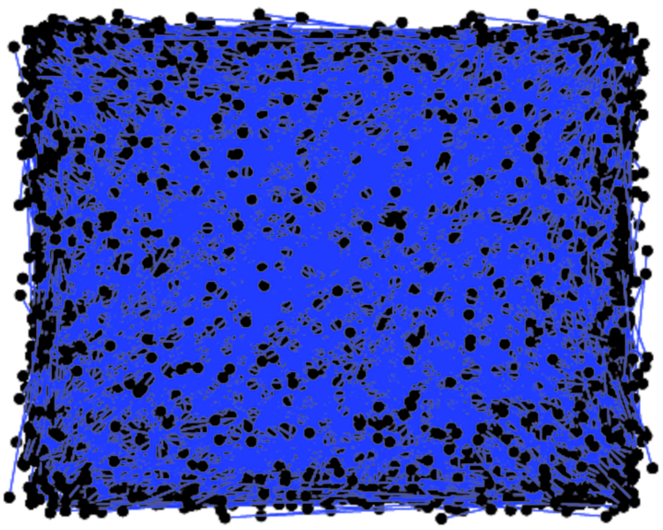


Figure 5.6: The Pretty Good Privacy network.

The pgp network is by far the largest in size, out of all the test network - despite still being the second largest in terms of edges - and that was the reason for selecting it for the study.

5.2 Obtaining and Analyzing Results

As previously mentioned, with the goal of testing whether it was possible to improve compression by motif sampling, three heuristic methods were employed for summarization. Each method scans the network for the frequency of a different given attribute, hypothesized to be strictly related to its summarization, prioritizing some motifs when attempting MDL-focused compression.

The intention was to determine if a different compression priority would impact the final compression ratio in a significant way, and whether or not a method would be significantly superior to the others. Also, if the compression was only related to the heuristic used, independently of the network it was being applied to, or rather affected by some other network metrics.

Throughout the remainder of this chapter, the heuristics applied for sorting the workload set of compression candidates are simply abbreviated as *h1a*, *h1d*, *h2a*, *h2d*, *h3a*, and *h3d*. With *h1a* and *h1b* concerning a sorting by ascending or descending frequency of network motifs on the whole network, respectively. Sorting by ascending or descending order of edges within each motif, being *h2a* and *h2b*, respectively. And, lastly, *h3a* and *h3d* comprising a sorting by the frequency

	revolution	**euroroad**	**air control**	**netsci**	**pgp**	**asoiaf**
–	90.61% (75.98%)	88.70%	89.61% (79.44%)	81.44%	85.27%	89.26% (67.55%)
h1a	106.01% (88.90%)	**81.23%**	**78.35% (69.45%)**	72.11%	77.77%	78.18% (59.17%)
h1d	**87.12% (73.05%)**	87.73%	89.90% (79.70%)	72.59%	76.33%	79.00% (59.79%)
h2a	93.62% (78.50%)	89.15%	89.53% (79.36%)	82.21%	85.81%	89.19% (67.51%)
h2d	108.30% (90.81%)	84.07%	81.78% (72.50%)	**57.65%**	**68.34%**	**68.03% (51.49%)**
h3a	96.63% (81.02%)	88.30%	89.13% (79.01%)	78.44%	84.17%	89.21% (67.52%)
h3d	117.68% (98.68%)	91.26%	92.48% (81.98%)	73.25%	75.37%	75.84% (57.40%)

Table 5.1: Compression ratios obtained for each network-heuristic pair, given in percentage, with the best obtained ratios highlighted in bold. Values are given as a ratio between compressed network over sorted network . Values in parenthesis are the compressions ratios over the original networks instead. The *revolution* and *air control* source networks had noise in the form of non-encoding trailing whitespace characters, while the *asoiaf* network was a weighted one with additional characters before sorting.

that a given node appears on different detected motifs, also by ascending and descending order, respectively.

An initial pass with no heuristic was also performed, in order to assess whether the sampling preference provided by sorting offered any discernible advantage. The motifs were purposefully feed in an arbitrary order - first were the motifs of size five, followed by those of size three, and finally motifs of size four - in an attempt to stray away from the heuristic's sorting criteria.

5.2.1 Overall Compression Ratios

Table 5.1 summarizes the compression ratios for each network-heuristic pair, taken from the ratios between the character count of the summarized descriptions and that of the original ones.

Analysing the table, and as initially suspected, the *revolution* network presents itself as an outlier. With some heuristics actually increasing the number of characters present on the network summary, due to the motif dictionary overhead.

It is clearly evident that a certain degree of compression was always achieved for each of the other networks, and not proportional to the network's size. And while some ratios are considerable, other networks do not exhibit a significant compression rate. Reiterating what was stated on section 3.1, this is far from being a superlative network compression algorithm, but it remains, nevertheless, a compression algorithm - particularly if the motif dictionary is to be hard-coded in some other file or as part of the graph-reading program.

It was also tested if removing repeated edges and self-loops would significantly improve compression. In the event that it would, then implementing a formula for accounting for such edges in future related work, could prove to be an important consideration.

Only two of the tested networks possessed repeated edges, as shown on table 5.2. Although

	revolution		**air control**	
	without repeated edges	full network	without repeated edges	full network
–	90.56 (75.58%)	90.61% (75.98%)	88.79% (72.93%)	89.61% (79.44%)
h1a	**106.04% (88.49%)**	106.01% (88.90%)	76.63% (62.94%)	78.35% (69.45%)
h1d	87.06% (72.65%)	87.12% (73.05%)	89.10% (73.19%)	89.90% (79.70%)
h2a	93.59% (78.10%)	93.62% (78.50%)	88.70% (72.85%)	89.53% (79.36%)
h2d	**108.34% (90.41%)**	108.30% (90.81%)	80.34% (65.99%)	81.78% (72.50%)
h3a	96.61% (80.62%)	96.63% (81.02%)	88.27% (72.50%)	89.13% (79.01%)
h3d	**117.77% (98.28%)**	117.68% (98.68%)	91.88% (75.47%)	92.48% (81.98%)

Table 5.2: Compression results after removing all duplicate edges and self-loops, given in percentage ratios as in table 5.1. Values in bold highlight instances were compression actually worsen, due to a heightened impact of the dictionary's overhead cost.

hardly a representative study of the prevalence and importance of edge multiplicity over large networks, at this time, the results do not support that taking into account the compression of repeated edges produce considerable improvements for the compression rate. Furthermore, it might be more adequate and efficient to simply transform multi-edges into weighted ones, and perform normal compression over the thus transformed graph.

Therefore, the most interesting prospect at this juncture, is to attempt to identify any existing relationship between the differences in compression rate, the nature and quantity of contracted motifs, and any number of network properties. Which, might provide insights into future approaches and optimizations, or even provide a better, yet entirely different method for graph compression and hopefully summarization, via motif contraction.

For that purpose, on the grounds of summarization, table 5.3 highlights some frequencies counts of compressed network motifs, that might offer some clues as to how the current method can be improved, or even what are the aforementioned network properties behind the differences in compression effectiveness.

One thing that is immediately evident by studying the provided tables is that, disregarding the *revolution* network outlier, two heuristics were clearly superior to the rest.

Heuristic *h1a*, which sorts detected network motifs by their increasing frequency of occurrence on the overall network, produced the best results for the *euroroad* and *air control* networks. On the other hand, for the larger networks, heuristic *h2d* produced the best compression results. Not only when compared to the other heuristics applied to those networks but in terms of overall compression across all studied networks, as well.

This reinforces the idea that compressing through network motif contraction, might become more viable the larger the network, due to a prevalence of a greater number of edges and connected components in those networks.

Heuristic *h2d* prioritizes contraction of motifs with the higher number of edges. Thus, it

revolution							**euroroad**							**air control**							
3	8	2	3	9	4	**12**	4	11	6	3	13	3	**14**	5	14	8	4	**21**	4	19	# of unique
37	36	38	45	26	**49**	26	646	236	326	643	226	**654**	260	1082	364	708	**1128**	163	1123	383	# of most frequent
4	4	5	4	5	4	5	3	3	5	3	5	3	5	3	3	5	3	5	3	5	size of most frequent
54	49	39	53	36	52	39	670	458	427	647	410	**675**	442	1175	799	764	1136	661	**1192**	776	TOTAL # of
–	*h1a*	*h1d*	*h2a*	*h2d*	*h3a*	*h3d*	–	*h1a*	*h1d*	*h2a*	*h2d*	*h3a*	*h3d*	–	*h1a*	*h1d*	*h2a*	*h2d*	*h3a*	*h3d*	**isomorphism class(es)**
3	13	13	3	21	3	**24**	6	4	7	4	**8**	6	**8**	7	5	7	4	**8**	7	**8**	# of unique
884	609	200	631	252	843	178	7245	4749	5657	**10484**	3681	7683	2857	876	546	794	**1283**	569	947	345	# of most frequent
3	3	5	3	5	3	5	3	3	4*	3	4*	3	4*	3	3	4*	3	4*	3	4*	size of most frequent
1177	855	705	1159	535	1137	691	11742	9691	8027	11522	7258	**11898**	7896	**1369**	1075	922	1352	824	1354	922	TOTAL # of
netsi							**pgp**							**asoiaf**							

Table 5.3: Different frequency measures of isomorphism classes, for each network-heuristic pair. Values in bold highlight the highest frequency of compressions of a given category.

will first compress larger and more denser clusters. This difference in best heuristic for different networks, hints that for networks lacking a high clustering coefficient, simply contracting the less frequent isomorphism classes first is the best approach.

It seems common sense that the proposed method of compression, focused on reducing the description of densely connected neighbouring nodes, works best when prioritizing those structures and for networks where they are prevalent. For networks that do not match this description, then starting with eccentric motifs and moving towards the most frequent ones reduces the latter's description size most likely by reducing the number of times that those motifs are fully described.

This hypotheses is corroborated by the findings expressed on table 5.3, where more structures were contracted, even though a fewer number of different isomorphism classes were compressed in total. Despite the fact that, at first glance, the table does not corroborate anything, due to lacking any strong relation between the ranking of effectiveness between overall compression and motif contractions. There is such a strong relation at the extremes.

The *netsci* network, for example, for which *h2d* reduced its size by nearly half, showed the highest number of compressed motifs when no heuristic was even employed. And, as such, in a cursory manner, nothing can be deduced by analyzing the number of compressed motifs. But that statement is not entirely true, seeing as two significant pieces of knowledge can actually be derived from its study.

While there is evidence that there is no direct relation between highest number of compressed motifs and highest graph compression, or even a direct correlation between the ranking for a given graph compression and number of compressed motifs. There is however a strict relation between one heuristic and the fewer number of contracted network motifs during summarization. Which turns out to be the one that achieves the best overall compression rates. Heuristic *h2d*, always compresses the fewer number of motifs, even for networks where it does not achieves the best compression. This is true even for the outlier *revolution* network.

Picking up on what was stated on the previous paragraph, this is certainly due to how by extensively describing overlapping structures, we are actually expending more characters than by describing fewer key structures and some surplus edges between them.

	best compression heuristic				
	best overall compression	**second best overall compression**	**lowest total # of compressed motifs**	**highest # of unique isomorphism classes**	**lowest # for the most prevalent compressed motif**
revolution	h1d	–	**h2d**	h3d	**h2d/h3d**
euroroad	h1a	**h2d**	**h2d**	h3d	**h2d**
air control	h1a	**h2d**	**h2d**	h2d	**h2d**
netsci	**h2d**	h1a	**h2d**	h3d	h3d
pgp	**h2d**	h3d	**h2d**	**h2d/h3d**	h3d
asoiaf	**h2d**	h3d	**h2d**	**h2d/h3d**	h3d

Table 5.4: Direct comparison between the best compression heuristics, those contracting the lowest number of motifs, and the highest number of isomorphism classes. Heuristic *h2d* is highlighted in bold.

	worst compression heuristic				
	worst overall compression	**second worst overall compression**	**highest total # of compressed motifs**	**lowest # of unique isomorphism classes**	**highest # for the most prevalent compressed motif**
revolution	h3d	h2d	–	h1d	h3a
euroroad	h3d	h2a	h3a	h2a/h3a	h3a
air control	h3d	h1d	h3a	h2a/h3a	h2a
netsci	h2a	–	–	–/h2a/h3a	–
pgp	h2a	–	h3a	h2a	h2a
asoiaf	–	h3a	–	h2a	h2a

Table 5.5: Direct comparison between the worst compression heuristics, those contracting the highest number of motifs, and the lowest number of isomorphism classes.

It is important to notice that, *h2d* achieved good compressions, despite the fact that it contracted a high - often the highest - number of different isomorphism classes. This both denotes the insignificance of the overhead cost of printing the motif dictionary alongside the actual network, for large networks, as well as the fact that the compression rates with this heuristic would be even better if the dictionary was not included.

A summary of the highest results for the above discussed metrics, are summarized on table 5.4 and table 5.5. In them, the relation between best and worst compressions, and most and least number of motifs compressed, are evidenced.

Worthy of note is that heuristic *h3d* produced good results for the last two largest networks, *pgp* and *asoiaf*. It, alongside *h2d* also led to the highest number of isomorphism classes contracted, and the lowest frequency for the most compressed motif. Which is in line with a more diverse compression approach; that is to say, more variety in the number of contracted network motifs and less predominance of a high number of compressions for a single or a few contracted motifs.

Seeing as heuristic *h3d* gives preference to nodes that occur in multiple detected isomorphism classes, it is also a good indicator of clustering. Furthermore, it might prove as an indicator of the distribution of node centrality and the size of large connected components.

In conclusion and if nothing else, it is safe to derive that the summarization algorithm is able to detect if a network has a high clustering coefficient or not, simply by observing what is the heuristic that best compresses it.

	tie-breaking heuristic	**Compression ratios**	**# of unique isomorphism classes**	**# of most compressed motif**	**Total # of compressed motifs**
euroroad	h1a2d	81.23%	11	236	458
		81.23%	11	236	458
air control	h1a2d	78.35% (69.45%)	14	364	799
		78.35% (69.45%)	14	364	799
netsci	h2d1a	57.80%	22	252	533
		57.65%	21	252	535
pgp	h2d3a	70.30%	8	3752	7288
		68.34%	8	3681	7258
asoiaf	h2d3d	73.33% (55.50%)	8	611	888
		68.03% (51.49%)	8	569	824

Table 5.6: Results of applying a composite of two heuristics: a second as tiebreaker, after the best one for each network. Top values for each network are the results for the composite heuristic, while bottom values the ones for the simple one.

5.2.2 Further Experimentation and Scalability

It is natural to encounter ties when sorting any list. Applying the described heuristics to sort the workload of compression candidates is no exception. As such, an experiment was performed in order to determine if applying a second heuristic on top of a main one, in order to solve ties, would improve the overall compression.

For each test network, the second best compressing heuristic was used as tiebreaker during sorting comparisons. The results, expressed on table 5.6, shows that there is no improvement in doing so.

In fact, in most cases the compression rate slightly worsened. The differences in compression, when present and either way, are relatively negligible. With indicates that there is no strong relation one side or another, whether it produces better, worst, or equal results. And, any small non-consistent variance, seems therefore fortuitous.

At most, one can deduce that by using a second heuristic to break a tie, we may be forcing a new ordering that, in the scope of the entire program execution, disrupts the efficacy of the main sorting ever so slightly.

If two or more heuristics can ever be employed concomitantly, it must be done with a different criteria, that allows giving precedence to a given sampling bias, rather than just solve ties.

Going back to tables 5.4 and 5.5, and reiterating what was said, a second major observation is realizing that the juxtaposition of *h2a* and *h2b* shows the clear inverse trend between overall compression and number of isomorphism classes. This is important, seeing that the two sort approaches only differ in their ordering.

In fact, returning even further back to table 5.3, looking at the size of the most frequently compressed motif for each heuristic, a pattern is apparent: that the size of the most frequently compressed motif is only dictated by whether the ordering is ascending or descending.

Independently of sorting by motif frequency (*h1a*, *h1d*), number of edges (*h2a*, *h2d*), or nodes occurrences in subgraphs (*h3a*, *h3d*), descending orders will always compress with more frequency a motif of size five, while ascending orders one of size three.

This is self-evident, seeing as motifs of larger size tend to have more edges, have more variations and therefore appear on more isomorphism classes, and include more nodes and therefore have more chance of overlapping. This fact, has no bearing on the compression rate or any other metric, as illustrated on table 5.3.

Two important distinctions need to be made on this point. The *revolution* network once again proves to be an outlier and that a small bipartite network is not favored by the summarization algorithm. And, that the *pgp* and *asoiaf* networks only go up to motifs of size 4.

This last point occurs, simply because no list of motifs of size five were provided to the Summarize program. These networks are so large, that they contain millions of motifs of that size. The simple plaintext file containing such motifs was approximately 20 GB and 90 GB, for the *pgp* and *asoiaf* networks, respectively. Such big files could not fit into memory, so a buffered reader and an external sorting method of splitting the large file into smaller fragments, and externally merge sort them on storage had to be used.

Even so, due to the bottleneck of reading from and writing to disk, coupled with the fast pace at which the file fragments grew in size, made the approach unpractical on the modest machine used for testing. A few attempts were successfully made for the smaller network, where one test took anything between one to three hours and used up nearly 100 GB of storage space.

For the larger network, *asoiaf*, with dozens of thousands of edges, the process had to be stopped when there was no more space left on the SSD.

A question could be made whether to attempt to optimize the external sorting in terms of read/write and merge sort, or whether to upgrade the working machine to a cluster of machines. However, comparing the nearly identical results obtained on the few successful tests for the *pgp* network, it was considered that the sane course of action was simply to entirely forego compressing motifs of size five for those two networks.

This last point leads to the last exploratory analysis performed. In which all the possible combinations of differently sized motifs were fed into the compression algorithm alongside the network to be compressed. And the variances in compression were then compared.

euroroad

	3	**4**	**5**	**3, 4**	**3, 5**	**4, 5**	**3, 4, 5**
h1a	87.40%	78.86%	84.20%	80.83%	83.22%	**79.49%**	81.23%
h1d	89.01%	78.74%	88.31%	**77.98%**	87.69%	88.41%	87.73%
h2a	89.01%	**78.26%**	90.41%	89.15%	89.23%	78.48%	89.15%
h2d	87.40%	76.88%	84.56%	**76.18%**	83.95%	84.59%	84.07%
h3a	88.28%	**77.70%**	84.62%	88.62%	88.62%	79.02%	88.30%
h3d	88.86%	79.48%	92.23%	**78.44%**	91.16%	91.98%	91.26%

air control

	3	**4**	**5**	**3, 4**	**3, 5**	**4,5**	**3, 4, 5**
h1a	85.32%	**74.90%**	80.31%	81.39%	79.75%	76.82%	78.35%
h1d	89.38%	78.73%	90.29%	**78.43%**	90.11%	90.07%	89.90%
h2a	89.38%	**78.77%**	90.06%	89.53%	89.61%	79.07%	89.53%
h2d	85.32%	73.33%	81.81%	**72.99%**	81.70%	81.90%	81.78%
h3a	89.09%	**78.37%**	88.96%	89.21%	89.12%	80.27%	89.13%
h3d	88.69%	78.82%	92.58%	**78.38%**	92.42%	92.65%	92.48%

netsci

	3	**4**	**5**	**3, 4**	**3, 5**	**4, 5**	**3, 4, 5**
h1a	75.59%	**66.83%**	75.96%	72.29%	71.72%	68.99%	72.11%
h1d	82.15%	74.55%	79.94%	**71.11%**	74.76%	76.01%	72.59%
h2a	82.15%	**74.99%**	83.99%	82.21%	82.26%	76.64%	82.21%
h2d	75.59%	64.27%	64.94%	60.91%	59.67%	60.86%	**57.65%**
h3a	78.49%	**69.40%**	74.86%	78.52%	78.51%	70.39%	78.44%
h3d	78.29%	72.41%	81.39%	**68.67%**	75.23%	76.58%	73.25%

Table 5.7: Compression results of using all possible combinations of selecting differently sized motifs. Values in bold highlight the best compression ratios for each heuristic.

For consistency of results, out of the pool of six initial test networks the aforementioned two were not included since, as previously stated, any compression that involved motifs of size five was unfeasible. The common outlier *revolution* was also excluded, for the obvious reasons of it only figuring on these experimentations due to its deviant nature, in the first place.

Table 5.7 depicts all the compression ratios for all the experiments, and a pattern clearly emerges.

There was little to no advantage in using all the motif sizes available, for the majority of heuristics. Furthermore, decreasing orders show best results when using only motifs of size three and four. While, increasing orders, showed the best results with using only motifs of size four.

There are only two instances where adding motifs of size five improve the final compression. And only one of those two instances produce the overall best result amongst the rest - on the *netsci* network.

Therefore, it seems plausible to say that the choice of heuristic is both more relevant as well as more informative. And if the need for further optimization arises, then performing a batch of compressions including motifs of increasing sizes can be performed.

Important to notice that if we reduce the maximum size of the motifs to be sampled for the *euroroad* and *air control* networks - using only motifs of size three and four - then the best summarizing heuristic becomes *h2d* as well, same as what is the case for the larger networks. This heuristic ran for this combination of motifs, obtains even better compression results that with *h1a* ran with motifs of all three possible sizes (size three, four, and five). This seems to indicate that motif size and network size are strictly correlated in terms of compression efficacy.

The fact that better execution times can be achieved with smaller sized motifs, and these generally produce the best results, strongly motivates the addition of larger sized motifs only when deemed necessary. This addition is an exploratory one, seeing as it may or may not produce better results in the end. So it should either be done because a maximum compression is strictly necessary, or for networks for which the entire workload can be loaded into memory and therefore the algorithm be run efficiently.

Overall, execution times were very fast, with *pgp* being the outlier due to being a single giant connected component. For all other networks, each heuristic performance was under a minute. The *pgp* network took at most nearly three minutes.

The plots of figures 5.7 and 5.8 illustrate this. Showing that heuristics *h3a* and *h3d* had the worst performance of them all, while *h1a*, *h1d*, *h2a*, and *h2d* where mostly on par with each other.

Results were also steady between different networks. That is to say, no one network performed better on some heuristics, and worse on others, in comparison to another network. Borrowing the expression, each network stuck to their execution time lane.

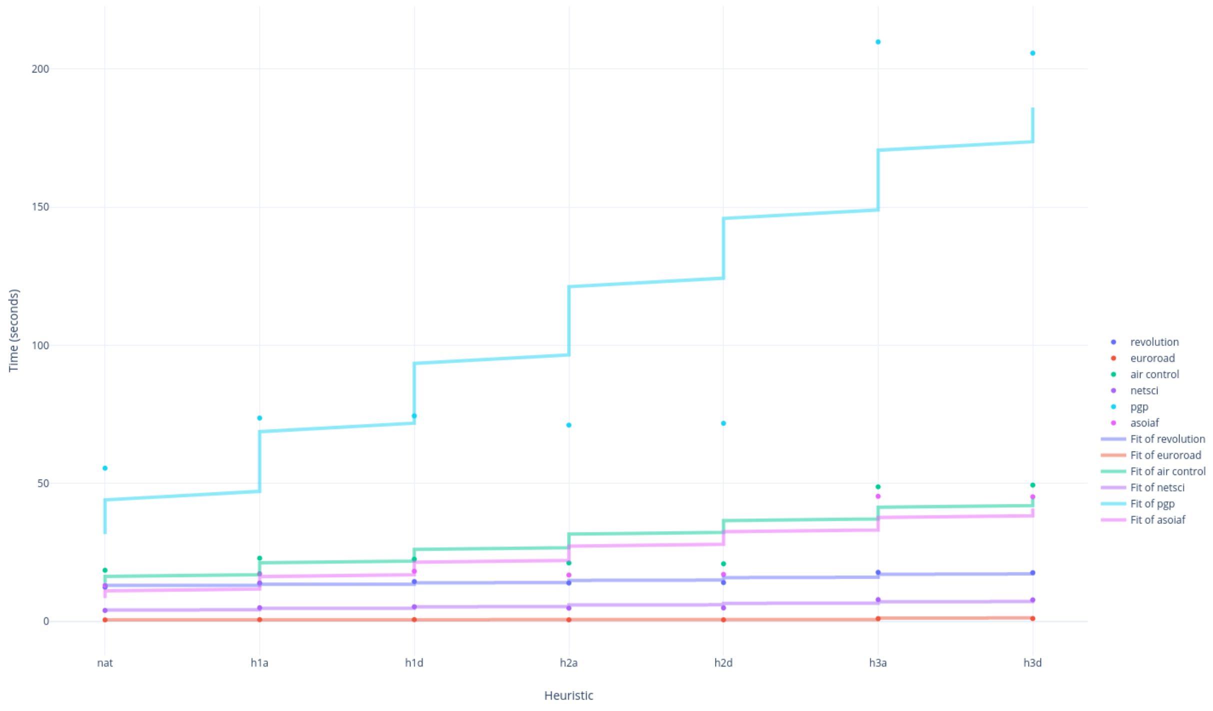


Figure 5.7: Plot of the different execution times for the different heuristics across all networks. Including curve fitting.

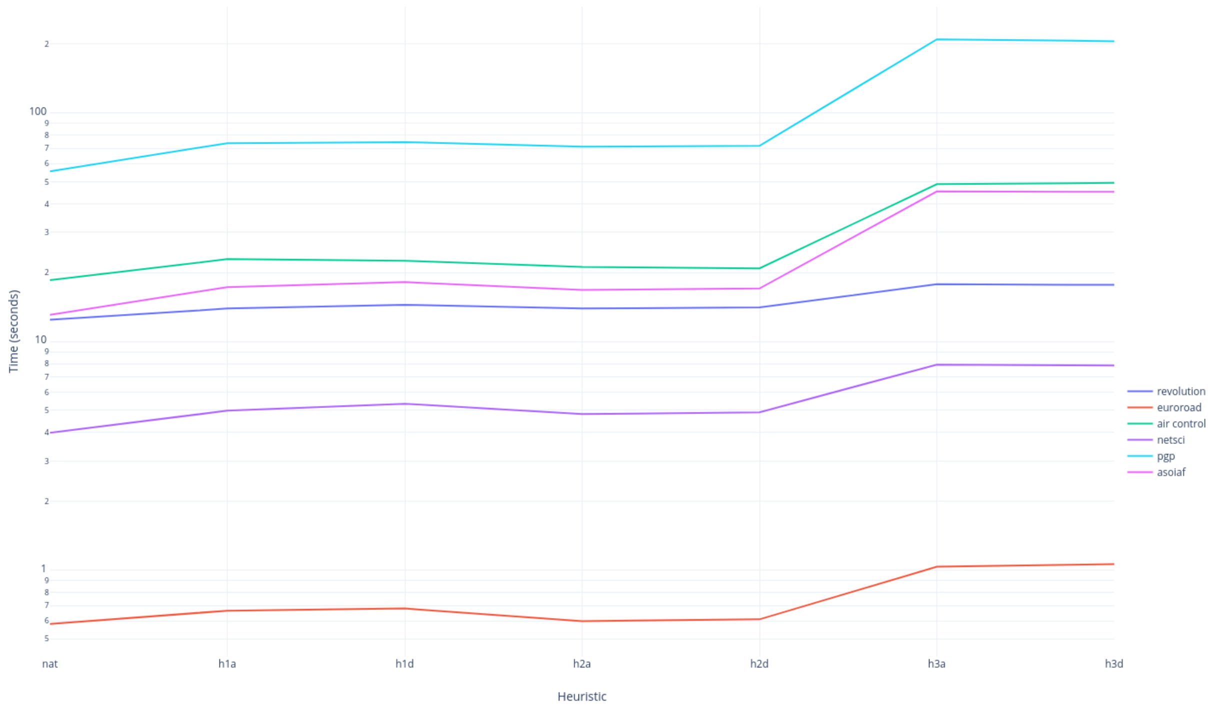


Figure 5.8: Logarithmic plot of the different execution times for the different heuristics across all networks.

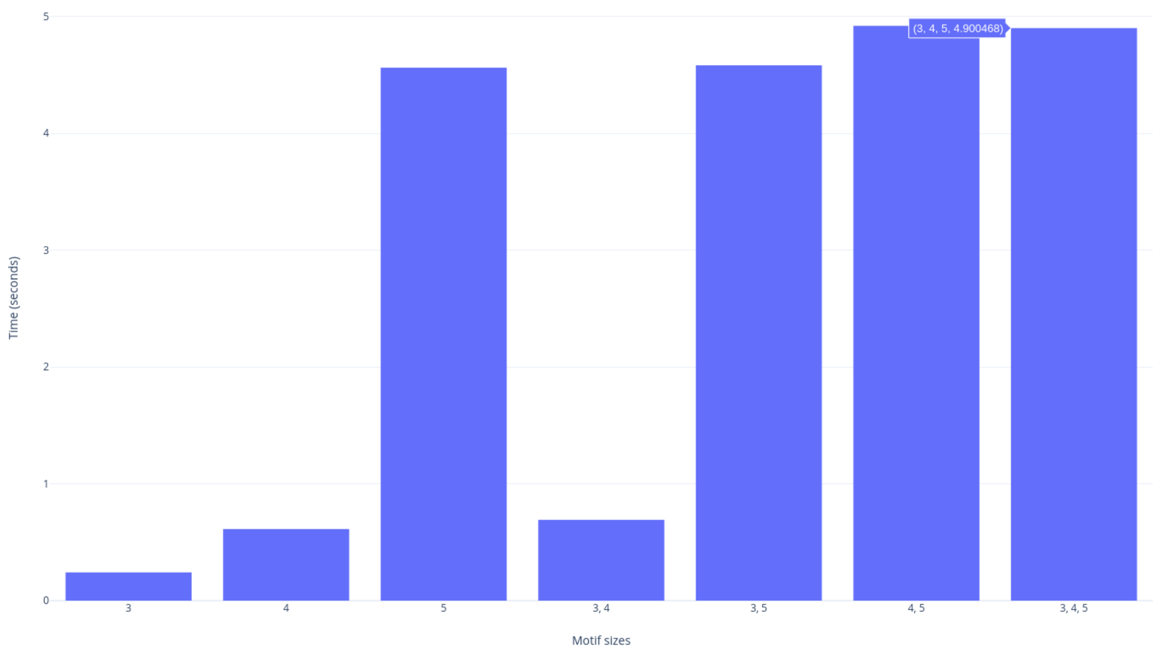


Figure 5.9: Plot of the different execution times for all possible combinations of differently sized motifs, for a sample network - *netsci*.

Finally, figure 5.9 shows execution for a single network across all possible variations of running compression with a given set of differently sized motifs. It comes with no surprise, that providing motifs of size five to the Summarize algorithm, severely impacts the execution time. This is noticeable even on a network like *netsci*, where the worst run time was still less than five seconds.

Overall the algorithm is scalable for very large networks in terms of volume - since, clearly, the number of edges is more relevant than that of nodes for time complexity, due to a larger number of detected motifs. The major limitation being the sheer size of the workload for larger networks, which can safely be bypassed if needed, by reducing the maximum size of the subgraphs that are fed into the compression algorithm, with little to no impact on the overall compression.

5.3 Relating Results to Network Metrics

The effectiveness of a sorting heuristic is dependent of the network. Sorting by ascending motif frequency produced the best compression for the *euroroad* and *air control*, these networks have a clustering coefficient of 0.063 and 0.034, respectively.

For the larger networks *netsci*, *pgp*, and *asoiaf*; the clustering coefficient is, in order, 0.693, 0.378, and 0.209. These networks' coefficient is higher than that of the previous two by an entire order of magnitude. This supports the notion that the higher the clustering coefficient the more compression is achieved by prioritizing motifs with more edges.

Which, in turn, supports the observation that contracting edges on highly dense networks leads to the overall best results, while doing so for more sparse networks entail a loss in efficacy. This is likely linked to the overlapping of multiple motifs on tightly knit clusters, that decreases the need to fully describe a motif in lieu of a few additional edges not already included in other motifs present within those clusters.

On the other hand, the size of the largest connected component (LCC) is directly related to both the amount of compressed motifs and total execution time. Although it does not translate into a clear determination for which heuristic achieves the best compression for a network, it seems to indicate that the largest the connected component is, the less effective the compression.

This relation can be observed when comparing the size of the LCC and the total number of compressed motifs with that network's best compression heuristic.

The *euroroad* network has a large connected component with size 1,039 for a network of size 1,174, with 458 compressed motifs. And the *air control* network an LCC of size 1,226 for a network of size the same size, and 799 compressed motifs. Both networks achieved best compressions with the *h1a* heuristic. Both networks had a high number of compressed motifs, in particular for their size. And, both networks exhibited a worse compression time for their size as well - *air control* even managed to exhibit the second worst time, surpassing even the *asoiaf* network.

The *netsci* network possesses an LCC of only 379 for a network of size 1,461, and 535 compressed motifs. *pgp*, being a snapshot of the largest connected component of a broader network, had an LCC of 10,680 having size 10,680, which was the entire network size, and achieved a number of 7258 compressed motifs. And, finally, the *asoiaf* network, also with a matching LCC of size 796 for a network of size 796, presented 824 compressed motifs with the best compression heuristic.

As stated, the difference in LCC size's magnitude does not correlate to the difference in magnitude for the achieved compression rates across all networks. But it does seem to indicate why the *netsci* network achieved such good compression, maybe due to opposing features. Namely, where the larger the volume of the network in relation to its size, the better the compression; and, the larger the LCC, the worse the compression.

This could account as to why the largest networks, with high clustering coefficients and a very high volume, achieved moderate compression despite there being a single giant connected component. And, why the *netsci* network, with also a high number of edges for its size and high clustering coefficient, but with lower LCC, achieved the best compression.

It could also account for the difference between the compression results of the *netsci* and *air control* networks, which have a similar number of nodes and edges, but very disparate clustering coefficient and sizes of LCC.

Compression seems to be independent of other key metrics such as degree distribution, diameter, and power law exponent. These metrics were either similar among them or, like in the case of network diameter, varied but irrelevant to compression.

Unfortunately, correlating centralities with compression would entail a dedicated deeper study, in order to compare which nodes were contracted, their original centralities, and that of the contracted supernode.

Initial centralities could be computed for the entire network, and fetched from memory as the contracted nodes are being added, so as to create a relation between compressed motifs and the centralities of their original counterparts. Possibly, calculating final centralities for new nodes as well, after the compression is finished.

That computation is not a part of the program on its current state.

Chapter 6

Conclusion

From a standpoint of learning more about the possibility of summarizing large networks, by compressing their statistically significant motifs, it was possible to formulate a unique system for describing graphs. The proposed method, is able to generate descriptions of the graph that are not only shorter, but that simultaneously are able to convey meaning regarding the network.

The ultimate goal of being able to apply this approach in such a way as to minimize a graph's description length, while maximizing the amount of readily available information regarding its properties, remains just that. A goal.

Graphs, despite appearing as a deceptively simple mathematical and abstract data structure, can quickly become immensely complex. And not simply by essence of its size, but more so by its inherent relational quality.

Being able to mine relevant substructures on a network - a computational demanding task by itself - and, by doing so, at the same time reduce the amount of noise in it, that would make further mining attempts easier, certainly presents itself as a worthwhile endeavour.

Unfortunately, the difficulty of ever achieving a complete, categorical, and feasible algorithm of doing so, seems tantamount to that of brute-forcing the solution in the first place.

Several authors proposed unique solutions addressing this problem, with various degrees of success. The more modest contribution of this work, managed to achieve what it set out to do and was certain it could do. Albeit, not to the degree to which was hoped that could have been done.

In a less roundabout way, the hypothesis that the proposed description for a graph, based not solely on its vertices or edges but rather focusing on its subgraphs, would be a shorter one, was shown to be true.

The amount of compression it achieved, although not groundbreaking, is still significant and relevant when it comes to larger and complex networks. Which were the main focus of study to begin with, and for which a description reduction to nearly half its original size was obtained, in terms of characters. A more modest degree of compression was achieved for smaller-sized networks with less clustering, not surpassing a description reduction of thirty percent.

Of a group of three heuristic employed to sample the search space of overlapping subgraphs on a network, one in particular was revealed to be promising, that prioritizes the compression validation of motifs with a larger volume first. Further refinements made to this heuristic, potentially over multi-layered compression that would allow to effectively combine another method for something more than simply breaking sampling ties, could just be the way to move forward.

6.1 Summarization Through Compression

Compression aside, despite existing a clear relation between the clustering coefficient of the network and the compression effectiveness of the aforementioned heuristic, little else was determined in regards to summarization.

The compressed networks contained information regarding compressed motifs. In that sense alone that constitutes a summarization. Whether that summarization is representative or even relevant to any network property is still to be determined.

The fact that the best compression was achieved for those instances where the least total amount of network motifs were compressed while, reciprocally, having the highest variety of different isomorphism classes, could be significant. It can signify that those better compressions are in fact displaying a wider and more relevant summarization - less structures, meaning more meaningful structures.

On the other hand, it can mean nothing at all, other than the fact that largest structures were compressed first, allowing for the remaining structures on those clusters to be adequately described by a few edges.

Grimness aside, the application of what can accurately be described as the first iteration of its method, presented some substantial relevant information, in the sense that it can be used for improving the algorithm towards a second iteration.

Only possible future work can tell whether a worthwhile summarization can be achieved, by actively examining the nature of the structures being compressed while they are being so in search of correlations between compression selection and feature importance. For this purpose, the proposed subgraph data structure implementation can help, seeing as it achieves a fast inclusion of supernodes on a network, without destroying the edge connections on the original network.

6.2 Future Work

Whatever the case, the results of the application of the proposed compression method, can and should be use in its improvement. Such as in the enhancement of its heuristics.

Immediate future work, could be expanding the sampling criterion of the most promising heuristic. Maybe even composing it alongside another. A feasible implementation of a multi-layered compression, taking advantage of the upstream and downstream quality of the nodes on the proposed graph, could be attempted.

One that does not simply brute-force compressions for all candidate subgraphs to maximize over-lapping, which leads to the folding problem described on section 4.2.3.

On the subject of a shorter description, it was ultimately shown that the trade-off of having a larger motif dictionary for a smaller number of compressed motifs, leads to better overall compression.

A possible way of exploiting this, could be expanding the definitions of detected subgraphs to include near-definitions of them, which would constitute a composition of those structures over overlapping motifs.

Still in terms of description length and achieving a more refined one, an obvious and straightforward first step is to include the motif dictionary on a separate file that could be used for multiple different networks. Another, might be to find a short way to describe the combination of multiple compressed motifs with overlapping nodes into a new motif description.

And, while counting characters as a measurement of MDL is a well-defined approach, since the original goal was to not deviate from the original plaintext description, a better rate of compression can be achieved by encoding the compressed network in another format entirely. Though this would require that any attempt to read the compressed network be aware of its encoding.

Finally, the goal of summarizing a network remains a pipe dream. The proposed algorithm achieved reasonable compressions for what was an exploratory and conceptual experiment, and the description is more informative that that of the original network. Still, there is no definitive way at this point to determine if the information summarized through compression is meaningful in any way.

Further testing is required on a multitude of networks to ascertain if a pattern exists between network metrics and the nature of the motifs summarized by the proposed compression.

Bibliography

[1] 10th dimacs implementation challenge - graph partitioning and graph clustering.

[2] Research portal: Icahn school of medicine.

[3] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Reviews of modern physics*, 74(1):47, 2002.

[4] Alex Arenas. Alex arenas website.

[5] Albert-Laszlo Barabasi and Zoltan N Oltvai. Network biology: understanding the cell's functional organization. *Nature reviews genetics*, 5(2):101–113, 2004.

[6] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. Gephi: an open source software for exploring and manipulating networks. In *Proceedings of the international AAAI conference on web and social media*, volume 3, pages 361–362, 2009.

[7] Edward A Bender and S Gill Williamson. *Foundations of combinatorics with applications*. Courier Corporation, 2013.

[8] Peter Bloem and Steven de Rooij. A tutorial on mdl hypothesis testing for graph analysis. *arXiv preprint arXiv:1810.13163*, 2018.

[9] Peter Bloem and Steven de Rooij. Large-scale network motif analysis using compression. *Data Mining and Knowledge Discovery*, 34(5):1421–1453, 2020.

[10] Stephen P Borgatti, Martin G Everett, and Jeffrey C Johnson. *Analyzing social networks*. Sage, 2018.

[11] Ed Bullmore and Olaf Sporns. Complex brain networks: graph theoretical analysis of structural and functional systems. *Nature reviews neuroscience*, 10(3):186–198, 2009.

[12] Corybrunson. Corybrunson/triadic: Replicable code for "triadic analysis of affiliation networks".

[13] Luciano da Fontoura Costa, Osvaldo N Oliveira Jr, Gonzalo Travieso, Francisco Aparecido Rodrigues, Paulino Ribeiro Villas Boas, Lucas Antiqueira, Matheus Palhares Viana, and Luis Enrique Correa Rocha. Analyzing and modeling real-world phenomena with complex networks: a survey of applications. *Advances in Physics*, 60(3):329–412, 2011.

[14] Ruijin Du, Ya Wang, Gaogao Dong, Lixin Tian, Yixiao Liu, Minggang Wang, and Guochang Fang. A complex network perspective on interrelations and evolution features of international oil trade, 2002–2013. *Applied Energy*, 196:142–151, 2017.

[15] Tim Dwyer, Nathalie Henry Riche, Kim Marriott, and Christopher Mears. Edge compression techniques for visualization of dense directed graphs. *IEEE transactions on visualization and computer graphics*, 19(12):2596–2605, 2013.

[16] Tomás Feder and Rajeev Motwani. Clique partitions, graph compression and speeding-up algorithms. In *Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 123–133, 1991.

[17] Blake Hendrickson, Devan Rosen, and R Kelly Aune. An analysis of friendship networks, social connectedness, homesickness, and satisfaction levels of international students. *International journal of intercultural relations*, 35(3):281–295, 2011.

[18] Nadav Kashtan, Shalev Itzkovitz, Ron Milo, and Uri Alon. Efficient sampling algorithm for estimating subgraph concentrations and detecting network motifs. *Bioinformatics*, 20(11): 1746–1758, 2004.

[19] Donald E Knuth. *The Stanford GraphBase: a platform for combinatorial computing*. ACM, 1993.

[20] Danai Koutra, U Kang, Jilles Vreeken, and Christos Faloutsos. Vog: Summarizing and understanding large graphs. In *Proceedings of the 2014 SIAM international conference on data mining*, pages 91–99. SIAM, 2014.

[21] Jérôme Kunegis. Konect: the koblenz network collection. In *Proceedings of the 22nd international conference on world wide web*, pages 1343–1350, 2013.

[22] Yongsub Lim, U Kang, and Christos Faloutsos. Slashburn: Graph compression and mining beyond caveman communities. *IEEE Transactions on Knowledge and Data Engineering*, 26 (12):3077–3089, 2014.

[23] Yike Liu, Tara Safavi, Abhilash Dighe, and Danai Koutra. Graph summarization methods and applications: A survey. *ACM computing surveys (CSUR)*, 51(3):1–34, 2018.

[24] Mathbeveridge. Mathbeveridge/asoiaf: Character interaction networks for george r. r. martin's "a song of ice and fire" saga.

[25] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594): 824–827, 2002.

[26] Matteo Pellegrini, David Haynor, and Jason M Johnson. Protein interaction networks. *Expert review of proteomics*, 1(2):239–249, 2004.

[27] Antonio Perianes-Rodríguez, Carlos Olmeda-Gómez, and Félix Moya-Anegón. Detecting, identifying and visualizing research groups in co-authorship networks. *Scientometrics*, 82(2): 307–319, 2010.

[28] Géraldine Pflieger and Céline Rozenblat. Introduction. urban networks and network theory: the city as the connector of multiple networks, 2010.

[29] Pedro Ribeiro and Fernando Silva. G-tries: an efficient data structure for discovering network motifs. In *Proceedings of the 2010 ACM symposium on applied computing*, pages 1559–1566, 2010.

[30] Pedro Ribeiro and Fernando Silva. G-tries: a data structure for storing and finding subgraphs. *Data Mining and Knowledge Discovery*, 28(2):337–377, 2014.

[31] Falk Schreiber and Henning Schwöbbermeyer. Frequency concepts and pattern detection for the analysis of motifs in networks. In *Transactions on computational systems biology III*, pages 89–104. Springer, 2005.

[32] Frank Schweitzer, Giorgio Fagiolo, Didier Sornette, Fernando Vega-Redondo, Alessandro Vespignani, and Douglas R White. Economic networks: The new challenges. *science*, 325 (5939):422–425, 2009.

[33] Zhao Tian, Limin Jia, Honghui Dong, Fei Su, and Zundong Zhang. Analysis of urban road traffic network based on complex network. *Procedia engineering*, 137:537–546, 2016.

[34] Alessandro Vespignani. Twenty years of network science. *Nature*, 558(7711):528–530, 2018.

[35] Sebastian Wernicke. Efficient detection of network motifs. *IEEE/ACM transactions on computational biology and bioinformatics*, 3(4):347–359, 2006.

[36] Sarah White, Tobin Yehle, Hugo Serrano, Marcos Oliveira, and Ronaldo Menezes. The spatial structure of crime in urban environments. In *International Conference on Social Informatics*, pages 102–111. Springer, 2014.

[37] M Wiliński, A Sienkiewicz, Tomasz Gubiec, R Kutner, and ZR Struzik. Structural and topological phase transitions on the german stock exchange. *Physica A: Statistical Mechanics and its Applications*, 392(23):5963–5973, 2013.