Data Mining and Knowledge Discovery manuscript No.
(will be inserted by the editor)

G-Tries: a data structure for storing and finding subgraphs

Pedro Ribeiro · Fernando Silva

the date of receipt and acceptance should be inserted later

Abstract The ability to find and count subgraphs of a given network is an important non trivial task with multidisciplinary applicability. Discovering networks motifs or computing graphlet signatures are two examples of methodologies that at their core rely precisely on the subgraph counting problem. Here we present the g-trie, a data-structure specifically designed for discovering subgraph frequencies. We produce a tree that encapsulates the structure of the entire graph set, taking advantage of common topologies in the same way a prefix tree takes advantage of common prefixes in strings. We thus avoid redundancy in the representation of the graphs, allowing for both memory and computation time savings. We introduce a specialized canonical labeling designed to highlight common substructure and we annotate the tree with a set of conditional rules that break symmetries, avoiding repetitions in the computation. In the end we are able to produce a novel efficient algorithm that takes as input a set of small graphs and is able to efficiently find and count them as induced subgraphs of a larger network. We perform an extensive empirical evaluation of our algorithms, focusing on efficiency and scalability, on a set of diversified complex networks. We show that g-tries are able to clearly outperform previously existing algorithms by at least one order of magnitude.

Keywords Complex Networks · Subgraphs · Data Structures · Trees · Network Motifs · Graphlets

Pedro Ribeiro · Fernando Silva
CRACS & INESC-TEC, Faculdade de Ciências, Universidade do Porto
R. Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
Tel.: +351 220402925
Fax: +351 220402950
E-mail: pribeiro@dcc.fc.up.pt, fds@dcc.fc.up.pt

1 Introduction

A wide variety of natural and artificial systems can be represented by complex networks [3,33]. Mining interesting features from these networks is a crucial task with an inherent multidisciplinary impact [13]. One way to analyze a complex network is to use a bottom-up approach, trying first to understand small topological substructures, and how they fit in the global overall behavior. We want to discover patterns of interconnections and understand why they exist and what is their meaning. These patterns can be thought of as subgraphs.

One foundational problem in subgraph mining, is the *subgraph counting problem*, that is, the ability to compute subgraph frequencies. Figure 1 illustrates the concept, by showing the frequency and occurrences of all induced subgraphs with 3 nodes in another network. This is the core problem that we are trying to solve in this article, and we give a more complete and formal description of it in Section 2.2.

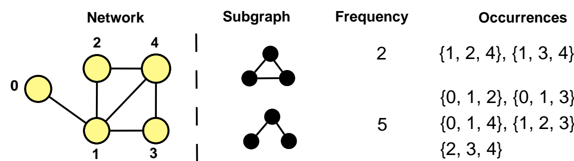


Fig. 1: An example of induced subgraphs frequency. The 5-node network has 7 induced subgraphs of size 3.

Subgraph frequencies have a great potential for network characterization and have wide applicability. One example of this are *network motifs* [30], a fundamental concept that has been used as a very useful tool to uncover structural design principles in networks from many different fields [48,2,49, 51]. They are basically subnetworks that appear with a higher frequency than it would be expected in similar random networks. Another example is the comparison of networks using fingerprints based on the *graphlet degree distribution* [39], which basically imply storing and counting the frequencies of a pre-defined set of small subgraphs.

Computing subgraph frequencies is however computationally *hard*, being closely related to the *subgraph isomorphism* problem, which is known to be NP-complete [11]. As the size of the subgraphs or networks increases, the time needed to compute the frequencies grows exponentially. This limits the applicability to very small sizes in order to obtain results in a reasonable amount of time. Being able to compute more efficiently would allow us to enlarge these limits and give practitioners of several scientific fields new angles in which to look at the networks in their application areas. Augmenting the size of the subgraphs we are able to count by even one node can bring new powerful insights.

Past approaches to subgraph frequency discovery are based on two extreme approaches: either search for all subgraphs of a certain size, or search just

for one subgraph at a time. By acknowledging that we are in fact trying to compute the frequency of a certain set of subgraphs, we can look at the problem from a new intermediate angle and take advantage of that, only looking for the subgraphs that interest us.

In this article we present the g-trie, a novel general data-structure able to efficiently store a collection of graphs. Its main property is that it identifies common smaller substructures and organizes them hierarchically in a tree. The first consequence is that we are able to compress the topological information given by the collection. This property enables us to create an efficient method capable of finding the frequency of these graphs as subgraphs of another graph. By traversing the tree, we can identify that a set of nodes is already a structural partial match to several possible descendant graphs. This avoids the need to postpone the isomorphism comparison to the end of the computation and also avoids having to start over when searching for another graph.

We extend preliminary work [42] and describe a fast algorithm for creating g-tries that iteratively inserts sugbraphs using a specialized canonical labeling procedure, based on a polynomial transformation over the canonization produced by `nauty` [27], one of the fastest isomorphism tools available. We also detail an efficient g-trie based frequency algorithm. By adding symmetry breaking conditions to the g-trie nodes and by exploiting the common substructure, we avoid redundant work.

We do an extensive and thorough empirical evaluation of the developed algorithms on a large set of representative networks from various fields. We study the details on efficiency and scalability of g-tries creation and its usage for frequency discovery. We also compare its results with the previously existing best algorithms, on a common platform, showing that g-tries consistently outperforms all competing methodologies by at least an order of magnitude. In doing this, we effectively push the limits on the applicability of network motifs, allowing the identification of larger motifs in larger networks.

The remainder of the article is organized as follows. Section 2 establishes a common graph terminology, gives a formal definition of the problem, and reviews previous approaches and related problems. Section 3 describes the g-tries data-structure, including algorithmic details on how to create and use it for counting subgraphs. Section 4 provides results about the experimental evaluation of the data-structures and associated algorithms on a large set of representative networks. Finally, Section 5 gives the final comments on the obtained results and concludes the article.

2 Preliminaries

2.1 Graph terminology

In order to establish a coherent graph terminology for this article, this section reviews the main concepts used. A *network* is modeled with the mathematical object *graph*, and we will use these two terms interchangeably.

A *graph* G is composed of a set $V(G)$ of *vertices* or *nodes* and a set $E(G)$ of *edges* or *connections*. The *size* of a graph is the number of vertices and is written as $|V(G)|$. A k-graph is a graph of size k. Every edge is composed of a pair (u, v) of two *endpoints* in the set of vertices. If the graph is *directed*, the order of the pair expresses direction, while in *undirected* graphs there is no direction in edges.

The *degree* of a node is the number of connections it has to other nodes. In directed nodes, we can also define the *indegree* and *outdegree* as, respectively, the number of ingoing and outgoing connections. A graph is classified as *simple* if it does not contain multiple edges (two or more edges connecting the same pair of nodes) and it does not contain self-loops (an edge connecting a node to itself). In the context of this work, we will be working with simple graphs.

The *neighbourhood* of a vertex $u \in V(G)$, denoted as $N(u)$, is composed by the set of vertices $v \in V(G)$ such that v and u share an edge. In the context of this article, all vertices are assigned consecutive integer numbers starting from 0. The comparison $v < u$ means that the index of v is lower than that of u. The adjacency matrix of a graph G is denoted as G_{Adj}, and $G_{Adj}[u][v]$ is 1 when $(u, v) \in E(G)$ and is 0 otherwise.

A *subgraph* G_k of a graph G is a graph of size k in which $V(G_k) \subseteq V(G)$ and $E(G_k) \subseteq E(G)$. This subgraph is said to be *induced* if $\forall u, v \in V(G_K)$, $(u, v) \in E(G_k)$ if and only if $(u, v) \in E(G)$. The neighborhood of a subgraph G_k, denoted by $N(G_k)$ is the union of $N(u)$, $\forall u \in V(G_k)$.

Two graphs G and H are said to be *isomorphic*, denoted as $G \sim H$, if there is a one-to-one mapping between the vertices of both graphs and there is an edge between two vertices of G if and only if their corresponding vertices in H also form an edge.

A *match* of a graph H in a larger graph G is a set of nodes that induce the respective subgraph H. In other words, it is a subgraph G_k of G that is isomorphic to H. The set of isomorphisms of a graph into itself is called the group of *automorphisms* and is denoted as $Aut(G)$. Two vertices are said to be *equivalent* when there exists some automorphism that maps one vertex into the other. This equivalence relation partitions the vertices of a graph G into equivalence classes denoted as G_E.

A *path* is a sequence of vertices such that there is an edge connecting any adjacent pair of vertices in the sequence. Two vertices are said to be *connected* if there is a path between them. A *connected graph* is a graph in which every pair of nodes is connected. An *articulation point* is a node from a connected graph that, when removed, disconnects the graph and creates two or more separated subgraphs.

2.2 The subgraph counting problem

The main initial motivation for the creation of g-tries was the discovery of *network motifs*. This terminology was introduced by Milo et al. [30] in 2002, and motifs were informally defined as patterns of inter-connections occurring

in complex networks in numbers that are significantly higher than those in similar randomized networks. Note that for the sake of simplicity, the term *motifs* in the context of this article will refer to network motifs.

This definition means that a motif is a subnetwork which is statistically over-represented. The key aspect to ensure statistical meaning is to be able to generate a large set of random networks as similar as possible to the original one, so that the intrinsic global and local properties of the network do not determine the motif appearance and that the motif is indeed specific to a particular network. The original proposal was therefore to maintain all single-node properties, namely the *in* and *out* degrees. Figure 2 exemplifies the concept.

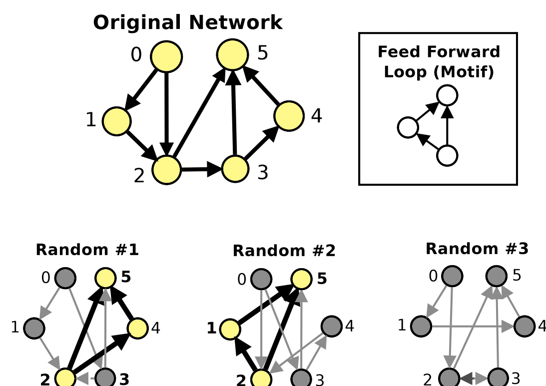


Fig. 2: An example network motif of size 3. The random networks preserve the ingoing and outgoing degrees of each node in the original network. The example motif appears at most once in each random network, but has three occurrences on the original one.

This definition is very general and can be applied equally to *directed* or *undirected* networks, or to *colored* networks, giving origin to *colored motifs* [23]. Several variations on the standard definition exist, such as looking for under-represented subgraphs (*anti-motifs* [29]), or using different constraints for the similar random networks [49].

The basic premise for finding motifs lies on the ability to compute the frequency of the subgraphs both on the original network and the randomized networks. In order to count the number of occurrences of a subgraph, the standard is to allow overlapping between two occurrences (matches), that is, they can share nodes and edges. This has an associated functional meaning as in, for example, biological applications. It is possible for several different overlapping subgraphs to be active and functioning at the same time with the same motif assuming different functions on each occurrence, as is the case of proteins in PPI networks [10]. Different frequency concepts exist [47] associated with the introduction of further constraints, for example not allowing sharing of nodes and edges. This leads to new problems with their own specifities in

choosing the "correct matches". The standard definition, which is the most widely adopted, will be used in this article.

In order to discover motifs, one has to compute a *subgraph census* on the original network, that is, calculate the frequency of all the subgraphs of a certain type. The typical application is to find all subgraphs of a given size k. Afterwards, one needs to compute the frequency of this set of subgraphs on the randomized similar networks, which is generally constituted by several dozens or even hundreds of networks. The bottleneck of the entire motif discovery process is therefore to compute subgraph frequencies, and this is the core computational problem that we are trying to solve, as stated in definition 1.

Definition 1 (General Subgraph Counting Problem)
Given a set of subgraphs S_G and a graph G, determine the exact count of all induced occurrences of subgraphs of S_G in G. Two occurrences are considered different if they have at least one node or edge that they do not share. Other nodes and edges can overlap.

2.3 Related work

We will now give a brief historical overview of the subgraph frequency computation algorithms, in the context of network motifs discovery. The first practical implementation of a sequential backtracking algorithm coincided with the origin of the term, with `mfinder` [30], in 2002. The first improvements appeared two years later, with the possibility of trading accuracy for better execution times by sampling subgraphs (`Kashtan` [21]). In the same year different frequency concepts were introduced and the algorithms were adapted accordingly (`FPF` [47]). In 2005 a breakthrough was reached, with the appearance of the first specialized algorithm that could avoid symmetries (`ESU` [54, 55]), thus avoiding redundancy in computation. In 2006, the first algorithm able to reach subgraph sizes of two digits appeared, although it succeeded in doing so by twisting a little bit the definition of motifs and only looking for a subset of all possible candidates (`NeMoFinder` [9]). In 2007, the capability of searching and counting individual subgraphs was introduced with the intent to avoid doing a complete subgraph census (`Grochow` [15]). In 2009, two new algorithms appeared, similar in concept and asymptotical behavior to `ESU` (`Kavosh` [20]) and `Grochow` (`MODA` [37]).

Finally, the concept of `g-tries` was introduced in 2010 [42]. This article differs from that previous work, given that we have expanded and refined the data structure and associated algorithms. For instance we now present a custom canonical labeling for the subgraphs and we introduce a filtering mechanism that reduces the number of symmetry breaking conditions. With this we are more efficient, as demonstrated with a much larger and thorough experimental evaluation, using both directed and undirected networks, and by comparing to three past state-of-art algorithmic competitors.

2.4 Related problems

There exists a vast amount of work on graph mining. Particularly, the field of *frequent subgraph mining* has been very prolific. Although related, this problem, which derives from the frequent itemset problem [38], is substantially different because its goal is to find the most frequent subgraphs that appear in a set of graphs, while in the network motifs domain we aim to find the frequency of all subgraphs on a single larger graph.

Frequent subgraph mining has produced algorithms like MOFA [6], gSpan [57], FFSM [18] or Gaston [35]. Because of their goal, these algorithms differ substantially in concept and applicability. For instance, they can prune possible unfrequent subgraphs based on their frequency, while we need to find the frequency of all possible subgraphs, even if they are infrequent. We should note that some authors use the term motif in the context of this related problem for subgraphs which simply occur frequently, but are not necessarily overrepresented [52, 17].

Subgraph search is another related problem, which given a single individual query graph G retrieves all graphs of a database that contain G as their subgraph. This is again different from our problem because we are looking for a set of queried subgraphs (as opposed to just one) on a single graph (as opposed to a set of graphs on a database) and we need to find the frequency of the subgraph (and not just if it appears at all). Given this, the algorithmic techniques used on subgraph search are different and not directly applicable. Index-like structures such as gIndex [58] or Lindex [59] are built around the database of graphs and are optimized for keeping the database between different queried subgraphs, while g-tries and are built around the subgraphs and are optimized for making the same query to different individual graphs.

3 G-Tries

3.1 Motivation and Prefix Trees

Previous methods for computing a k-subgraph census follow one of two conceptually extreme approaches [44]. On one side we have the *network-centric* methods (like mfinder, ESU and Kavosh), meaning that the algorithm used relies on first enumerating all k-subgraphs, and then finding which ones are isomorphic. The enumeration itself can be very time consuming and afterwards we basically have several instances of the graph isomorphism problem, which is computationally hard [27, 22]. These methods must compute a complete census of the respective network, regardless of it being the original or a similar random network. However, the random networks can contain more types of subgraphs than the original one, and we are only interested in knowing the frequency of the subgraphs that do appear in the original network (more than that, the motifs definition implies a certain minimum frequency, and that may imply that even more subgraphs do not need to be considered for the random

networks). This means we will be doing a significant proportion of unnecessary work by computing the frequency of subgraphs that are not interesting from the point of view of motifs discovery.

At the other side, we have *subgraph-centric* methods (like Grochow and MODA) which consist in computing the frequency of a single subgraph each time, by finding isomorphic matchings of that individual subgraph. For discovering motifs we would need to first generate the list of subgraphs we are interested in (for example, all subgraphs of a determined size), and then apply the individual counting, in turn, to all list elements. Each of these instances is at least as hard as the NP-complete subgraph isomorphism problem [11], and therefore also computationally intractable. Moreover, no information is reused from one subgraph to another. For example, two subgraphs that are exactly the same with the exception of a single node, will be computed extensively and separately without taking advantage of that similarity.

The main idea for a new methodology for discovering motifs is therefore to explore the described drawbacks of previous algorithms in order to gain computational efficiency. We wanted an algorithm specialized in finding a set of subgraphs: not necessarily all possible subgraphs, but also not just one single subgraph. This is the core of what motif algorithms are really doing on the random network census computation, which constitutes the bottleneck of the whole motif discovery process.

When dealing with sequences, if we want a data structure that can store a set of a words, that is, a dictionary, we could use the trie data structure, also known as a prefix tree [14]. Tries make use of common prefixes and they are basically trees, where all descendants of a node have the same common prefix. Algorithmically, tries can be considered an efficient structure, providing linear execution time for verifying if a word of size n is in the set. Basically we can just descend the trie, one letter at a time. In memory terms they also provide big saves when comparing to actually storing all the words, because common prefixes are only stored once, avoiding redundancy.

One of the possible usages of a trie is to discover instances of all the words contained in the trie. When we have a partial match of a word, we know exactly which words can be formed from that particular subsequence. In that sense we do not need to do the redundant work of searching again for the same prefix. More than that, at a certain point, we could know that there is no possible word of a determined size starting with a certain prefix, since there are no descendant nodes, and we could stop the computation on that search branch.

The core idea for the g-trie data structure is to take advantage of these conceptual advantages and apply them in the graphs realm.

3.2 G-Tries Definition

A trie takes advantage of common prefixes. By analogy, g-tries take advantage of common substructures in a collection of graphs. In the same way two or more strings can share the same prefix, two or more graphs can share a

common smaller subgraph. Like tries, g-tries are trees. Each trie node has a single letter and each g-trie node will represent a single graph vertex. Each vertex is characterized by its connections to the respective ancestor nodes. We represent these connections as an array of boolean values (0-1), where 1 means a connection and 0 its absence. The first value of the array represents the connection to the first node, the second value represents the connection to he second node and so on. All of this can be visualized in Figure 3.

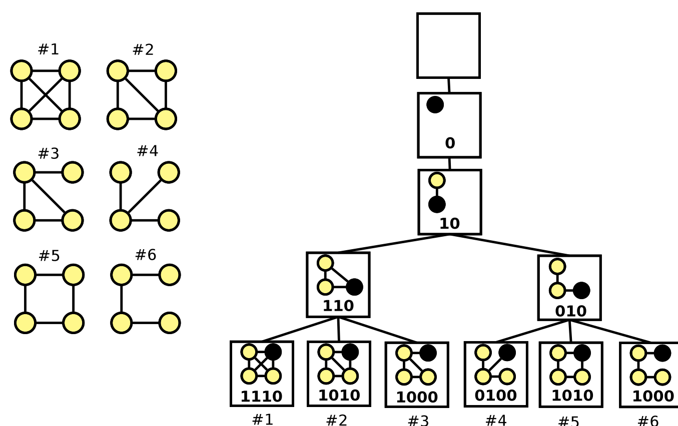


Fig. 3: A g-trie representing a set of 6 undirected graphs. Each tree node adds a new vertex (in black) to the already existing ones in the ancestor nodes (light vertices). The 0-1 boolean arrays represent the connections to already existing nodes.

Note that all graphs with common ancestor tree nodes share common substructures that are characterized precisely by those ancestor nodes. A single path through the tree corresponds to a different single graph. Children of a node correspond to the different graph topologies that can emerge from the same subgraph. Graphs of different sizes can be stored in the same tree if each tree node also signals if it corresponds to the "end" of a graph. All of this is easily applicable to both undirected and directed subgraphs.

We call these kind of trees *g-tries*, from the etymology "**G**raph re**TRIE**val". We now give an informal definition of this abstract data structure. Note that a multiway tree has a variable number of children per node.

Definition 2 (G-Trie) *A* `g-trie` *is a multiway tree that can store a collection of graphs. Each tree node contains information about a single graph vertex, its corresponding edges to ancestor nodes and a boolean flag indicating if that node is the last vertex of a graph. A path from the root to any* `g-trie` *node corresponds to one single distinct graph. Descendants of a* `g-trie` *node share a common subgraph.*

In order to avoid further ambiguities, throughout this article we will use the term *nodes* for the g-trie tree nodes, and *vertices* for the graph network nodes.

3.3 Creating a G-Trie

The first task one must be able to do in order to use g-tries is of course to be able to create one. The following sections will show how we do this.

3.3.1 Iterative Insertion

In order to construct a g-trie, we just repeatedly insert one subgraph at a time, starting with an empty tree (just a root node). In each insertion, we traverse the tree and verify if any of the children has the same connections to previous nodes as the graph we are inserting. With each increase in depth we also increase the index of the vertex we are considering. Figure 4 exemplifies this process.

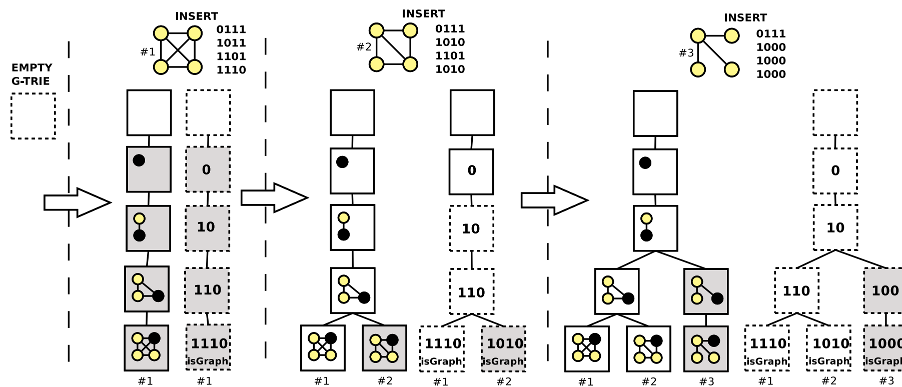


Fig. 4: Sequential insertion of 3 graphs on an initially empty g-trie. Gray squares are new g-trie tree nodes after each insertion. White squares are the already existing nodes. Dashed squares correspond to an adjacency matrix implementation, with `'1'` representing a connection to a previous vertex, and `'0'` its absence, while the other boxes give a visual representation.

Note that the g-trie root node must be empty since there are two possible direct child nodes: a vertex with or without a connection to itself. In this way, g-tries are also able to accommodate self-loops.

3.3.2 The Need for a Canonical Representation of Graphs

Following the described insertion procedure, the insertion is completely defined by the adjacency matrix of the inserted graph. However, there are many dif-

ferent possible adjacency matrices representing the same class of isomorphic graphs. The problem with this is that different matrices will give origin to different g-tries. We could even have two isomorphic graphs having different g-trie representations, leading to different branches of the tree representing the same graph, which would contradict the purpose of the g-trie. In order to avoid that we must use a canonical labeling, to guarantee that isomorphic graphs will always produce the same unique adjacency matrix, and therefore the same set of subgraphs is guaranteed to produce the same g-trie.

There are many possible canonical representations, and the representation used directly, and significantly, impacts the g-trie structure. In order to illustrate this, consider the string formed by the concatenation of all adjacency matrix rows, and call it *adjacency string*. Any choice of canonical representation will give origin to different adjacency strings. Two possible options of forming a canonical adjacency string would be to consider the *lexicographically larger* or the *lexicographically smaller* one for each graph.

Figure 5 illustrates the g-tries generated for each of these possible choices for the same set of six 4-graphs. Note the contrast between these two choices, with completely different structures of the g-trie formed. One can clearly observe a variation on the number of g-tries nodes needed and a different balance on the nodes of each size of the g-trie.

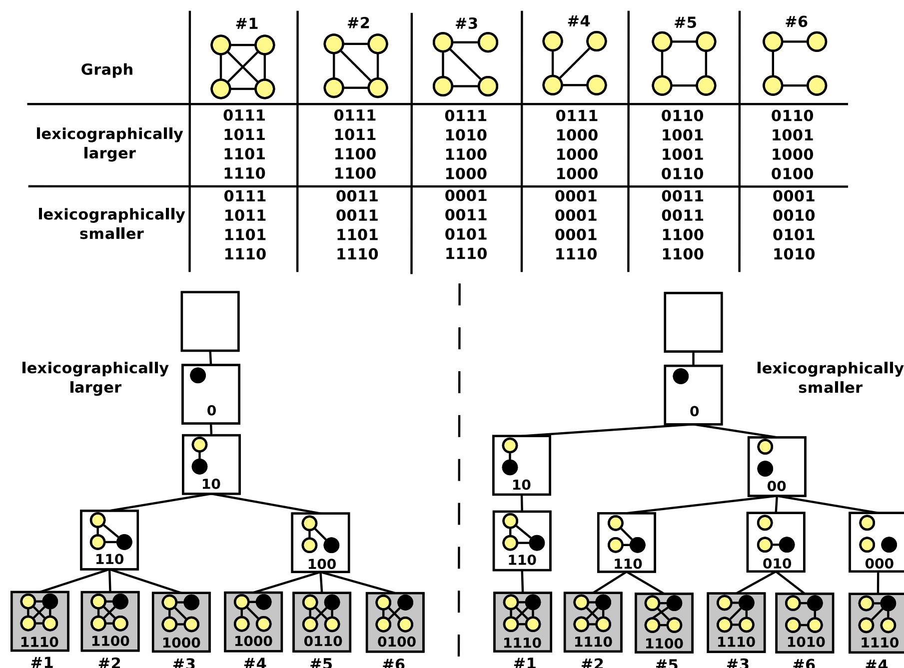


Fig. 5: Two different g-tries using lexicographically larger and smaller adjacency strings. `1's` and `'0'` represent respectively the existence or absence of a connection.

If we increase the amount of common ancestor topologies, we decrease the size of the tree and effectively we compress the representation, needing less memory to represent it than when we had the original set of subgraphs (represented by their adjacency matrices). We can measure the amount of compression if we take into account the number of nodes in the tree and the number of vertices in the subgraphs (equation (1)). By using a tree we do have to spend some auxiliary memory to represent the tree edges, but the total memory needed for the tree structure is very small compared to the actual information stored in the nodes (the graph adjacency matrix) and loses relative weight as we increase the amount of subgraphs and their size. Hence, the real memory bottleneck is in the storage of node values, and equation (1) is a good indicator of how much space we save and how much common substructure we identified.

$$\text{compression ratio} = 1 - \frac{\text{nodes in tree}}{\sum \text{nodes of stored graphs}} \qquad (1)$$

As an example, the two g-tries constructed in figure 5 have a compression ratio of respectively $58.34\% = 1 - 10/24$ (lexicographically larger) and $45.84\% = 1 - 13/24$ (lexicographically smaller). We can ignore the root, since it uses constant memory space and only exists as a placeholder for the initial children representing the first vertex. A tree with no common topologies would need a node for each graph vertex and would have a 0% compression ratio.

3.3.3 An Efficient Custom Built Canonical Form

The lexicographically largest adjacency string seems like a good candidate for the canonical representation and in fact it was the first we experimented. However, it is not the only possible choice, and we will now give the rationale behind our chosen canonization.

In general, a canonical label suitable for our purposes should try to highlight the main characteristic of g-tries, which is the identification of common substructure. Given that, we are aiming to create the smallest possible g-trie, that is, the one with the highest compression, or the one which identifies more common topology. This has the immediate effect of reducing the memory needed but at the same time has implications on the actual computation of frequencies. As you will be able to see in detail in the following sections, we will try to produce partial matches of graph nodes along g-trie traversals. This means that at a given point in our search, we know that a certain set of network vertices corresponds to a certain g-trie node. By having less g-trie nodes, we actually decrease the structural matching that this process needs.

If we really look into this search, we will be trying to augment our partial match by adding another network vertex, descending one level in our g-trie. In order to improve the efficiency of our search, we need to constrain as much as possible the vertices which are suitable candidates for this augmentation of our partial match. For this we take into account that real networks are usually

sparse: they have much less connections than the maximum possible number of edges.

Given this, we want to produce the labeling which maximizes the number of connections to ancestor vertices. When we are adding a new vertex, it will have to match these connections, and since the network is sparse, the more connections we have, the less candidates will precisely match those needed edges. As an example, consider that the extreme case of having a label that produces disconnected subgraphs. This means we will have a node inside the subgraph which is not connected to any other node. The candidates for this position would be all the vertices not connected to any of the vertices already matched, which would be a really large set given the sparsity of the network. By contrast, if we require connected subgraphs, we guarantee that we can only use the nodes that are neighbours of the partial match. With more connections to ancestor nodes, we potentially reduce even more the suitable vertex candidates.

The lexicographically largest string is not geared specifically towards these desired properties. It is also very time consuming to compute. Given this, we opted to create our own more efficient canonical representation, geared to being more efficient to compute and to create as much constraints as possible for later use when matching the g-trie graphs as subgraphs of another larger network. Algorithm 1 describes our method for computing a canonical form, and we call it `GTCanon`.

Algorithm 1 Converting a graph to a canonical form

Require: Graph G
Ensure: Canonical Form of G

1: **function** GTCanon(G)
2: G := nautyLabeling(G)
3: **for all** $i \in V(G)$ **do**
4: current_degree[i] := nr ingoing+outgoing connections of i
5: global_degree[i] := last_degree[i] := current_degree[i]
6: **for** $pos : |V(G)|$ down to 1 **do**
7: Choose u_{min} subject to:
8: • u_{min} is still not labeled and is not an articulation point
9: • u_{min} has minimum current_degree
10: • In case of equal min. current_degree, choose u_{min} with min. last_degree
11: • In case of equal min. last_degree, choose u_{min} with min. global_degree
12: label$_{CanonG}[u_{min}]$:= pos
13: last_degree[] := current_degree[]
14: update current_degree[] removing u_{min} connections
15: **return** CanonG

The first step of `GTCanon` is to apply any other canonical representation. In our case we use `nauty` [27], a proven and very efficient third-part algorithm (line 2). Then, several lookup tables are initialized with the degrees of every node of the graph. The core of the algorithm is iterative and in each step

we select a new node for being labeled at the last available labeling position. The idea is to choose a node that has the minimum amount of connections as possible (lines 9 to 11), guaranteeing that it does not divide the graph in two (line 8) and then label it (line 12) and remove it from the graph. Before the next iteration, the lookup tables with degree information are updated (lines 13 and 14).

By removing a node not densely connected to the rest of the graph, we increase the number of connections in lower labeling positions, and therefore we increase constraining. By not choosing articulation points, we guarantee connectivity in the subgraph. Finally, each time we remove a node, we get a smaller instance of the same problem, having to compute a canonical form of the graph with one less vertex. By using the same criteria on each phase for all graphs, we increase the compressibility. When our criteria does not suffice to chose an unique candidate, the fact that we first used another canonical form guarantees that `GTCanon` will also be canonical and always return the same labeling for isomorphic graphs.

Note that computing a canonical form is always a computational hard problem because solving it is at least as hard as the isomorphism problem (two graphs are isomorphic if they have the same canonical form). However, `GTCanon` takes advantage of an efficient third party algorithm, `nauty`, which is state-of-art, and uses an efficient polynomial algorithm after that computation. Computing the articulation points can be done in linear time $O(|V(g)| + |E(G)|)$ with a simple depth-first search [50]. Computing and updating the three degree arrays can also be done in linear time.

Figure 6 illustrates `GTCanon` in action and was automatically created using our own code. The results section gives more empirical verification that `GTCanon` is indeed a good choice for the labeling (see Section 4.3).

3.3.4 Insertion Algorithm

With the considerations made in the previous sections we are now ready to detail our algorithm for creating a g-trie. Algorithm 2 details a method to insert a single graph in a g-trie. As said, constructing a complete g-trie from a set of subgraphs can be done by inserting the graphs, one by one, into an initially empty tree. In each g-trie node, *in* represents the incoming connections, *out* represents the outgoing connections and isGraph is a boolean value indicating if a stored subgraph ends at that node.

Explaining in more detail, we start by computing the canonical adjacency matrix of the graph being inserted (line 2). Then we recursively traverse the tree, inserting new nodes when necessary, with the procedure `insertRecursive()`. This is done by going through all possible children of the current node (line 8) and checking if their stored value is equal to the correspondent part of the adjacency matrix (lines 9 and 10). If it is, we just continue recursively with the next vertex (line 11). If not, we create a new child (lines 13 to 16) and continue as before (line 18). When there are no more vertices to process, we

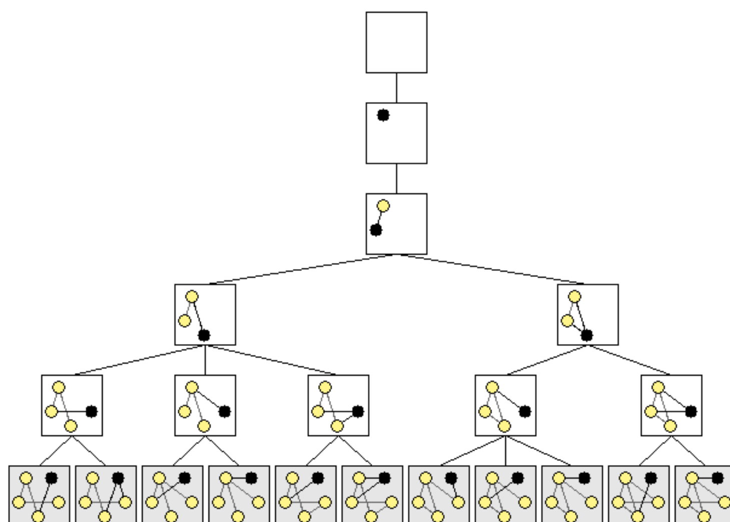


Fig. 6: A g-trie containing the 11 undirected 5−subgraphs found in an electronic circuit network.

stop the recursion (line 5) and set the `isGraph` variable, indicating the end of the graph (line 6).

Regarding the complexity of the algorithm, `insertRecursive()` takes $O(|V(G)|^2)$, the size of the adjacency matrix. We could reduce this by using for instance an hash table, but as we will see later, the insertion will not be at all the bottleneck of the entire subgraph search algorithm and this procedure, as it is, keeps simplicity and memory usage as low as possible. The whole insertion also needs to calculate the canonical labeling of the graph.

After constructing the g-trie, if we want to retrieve the initial set of graphs a simple depth-first search of the tree will suffice. A path from the root to any given g-trie node at depth k with `isGraph` set to true, represents a k-graph.

3.4 Computing subgraph frequencies

Once the g-trie is built, the next logical step is to create a method for finding instances of the g-trie graphs as subgraphs of another larger network.

3.4.1 An Initial Approach

Algorithm 3 details a method for finding and counting all occurrences of the g-trie graphs as induced subgraphs of another larger graph. The main idea is to backtrack through all possible subgraphs, and at the same time do the isomorphism tests as we are constructing the candidate subgraphs. We take

Algorithm 2 Inserting a graph G in a g-trie T

Require: Graph G and G-Trie T
Ensure: Inserts graph G in G-Trie T

```
 1: procedure gtrieInsert(G, T)
 2:     M := adjacency matrix of GTCanon(G)
 3:     insertRecursive(M, T, 0)
 4: procedure insertRecursive(M, T, k)
 5:     if k = numberRows(M) then
 6:         c.isGraph = true
 7:     else
 8:         for all children c of T do
 9:             if (c.out = first k + 1 values of k-row of M) AND
10:                     (c.in = first k + 1 values of k-column of M) then
11:                 insertRecursive(M, c, k + 1)
12:                 return
13:         nc := new g-trie node
14:         nc.in := first k + 1 values of k-row of M
15:         nc.out := first k + 1 values of k-column of M
16:         nc.isGraph := false
17:         T.insertChild(nc)
18:         insertRecursive(M, nc, k + 1)
```

advantage of common substructures in the sense that at a given time we have a partial isomorphic match for several different candidate subgraphs (all the descendants).

At any stage, V_{used} represents the currently partial match of graph vertices to a g-trie path. We start with the g-trie root children nodes and call the recursive procedure `match()` with an initial empty matched set (line 2). The later procedure starts by creating a set of vertices that completely match the current g-trie node (line 4). We then traverse that set (line 5) and recursively try to expand it through all possible tree paths (lines 7 and 8). If the node corresponds to a full subgraph, then we have found an occurrence of that subgraph (line 6). Note that at this time no isomorphic test is needed, since this was implicitly done as we were matching the vertices.

Generating the set of matching vertices is done in the `matchingVertices()` procedure. The efficiency of the algorithm heavily depends on the above mentioned constraints as they help in reducing the search space. To generate the matching set, we start by creating a set of candidates (V_{cand}). If we are at a root child, then all graph vertices are viable candidates (line 10). If not, we select from the already matched vertices that are connected to the new vertex (line 12), the one with the smallest neighborhood (line 13), thus reducing the possible candidates to the unused neighbors (line 14). Then, we traverse this set of candidates (line 16), and if one respects all connections to ancestors (lines 17 and 18) we add it to the set of matching vertices (line 20). Since we are using the lexicographically larger representation, the initial nodes will have the maximum possible number of connections. This also helps in constraining

Algorithm 3 Census of subgraphs of T in graph G

Require: Graph G and G-Trie T
Ensure: All occurrences of the graphs of T in G

1: **procedure** GTRIEMATCH(T, G)
2: **for all** children c of $T.root$ **do** MATCH(c, $\emptyset$)
3: **procedure** MATCH(T, V_{used})
4: $V :=$ MATCHINGVERTICES(T, V_{used})
5: **for all** node v of V **do**
6: **if** $T.isGraph$ **then** output $V_{used} \cup \{v\}$
7: **for all** children c of T **do**
8: MATCH($c, V_{used} \cup \{v\}$)
9: **function** MATCHINGVERTICES(T, V_{used})
10: **if** $V_{used} = \emptyset$ **then** $V_{cand} := V(G)$
11: **else**
12: $V_{conn} := \{v : v \in N(V_{used})\}$
13: $m := m \in V_{conn} : \forall v \in V_{conn}, |N(m)| \leq |N(v)|$
14: $V_{cand} := \{v \in N(m) : v \notin V_{used}\}$
15: $Vertices := \emptyset$
16: **for all** $v \in V_{cand}$ **do**
17: **if** $\forall i \in [1..|V_{used}|]$:
18: $T.in[i] := G_{Adj}[V_{used}[i]][v] \quad \wedge T.out[i] = G_{Adj}[v][V_{used}[i]]$ **then**
19: $Vertices := Vertices \cup \{v\}$
20: **return** $Vertices$

the search and reducing the possible matches. Figure 7 exemplifies the flow of the previously described procedure. Note how the subgraph $\{0, 1, 4\}$ is found twice.

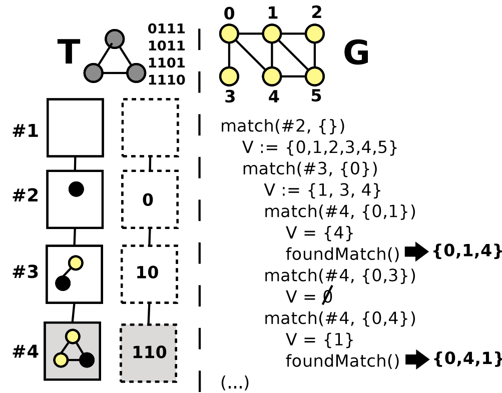


Fig. 7: An example of a partial program flow of the recursive g-trie `match()` procedure, when searching for a 3-subgraph on a graph of 6 vertices.

3.4.2 Symmetry Breaking Conditions

One problem with the `gtrieMatch()` method described is that we do not avoid subgraph symmetries. If there are automorphisms on a subgraph, then that specific subgraph will be found multiple times. In the example of figure 7, we would not only find $\{0,1,4\}$ but also $\{0,4,1\}$, $\{1,0,4\}$, $\{1,4,0\}$, $\{4,0,1\}$ and $\{4,1,0\}$. At the end we can divide by the number of automorphisms to obtain the real frequency, but a lot of valuable computation time is wasted.

G-tries need to avoid this kind of redundant computations and find each subgraph only once. In order to achieve that we generate a set of symmetry breaking conditions for each subgraph, similarly to what was done by Grochow and Kellis [15]. The main idea is to generate a set of conditions of the form $a < b$, indicating that the vertex in position a should have an index smaller than the vertex in position b. This is illustrated in Figure 8.

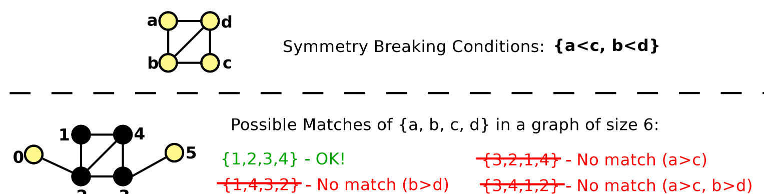


Fig. 8: Symmetry breaking conditions for an example graph. The conditions only allow one automorphism, disallowing all others.

Although inspiration was taken from Grochow and Kellis [15], a slightly different method for generating the conditions is used, which we detail in algorithm 4. Our algorithm differs from Grochow and Kellis [15] essentially because of the method by which we choose the candidates for the conditions. Instead of choosing arbitrary nodes inside the same equivalence class, we always choose the smallest possible index, so that any condition of the form $a < b$ implies that a is smaller than b, as you can see in Figure 9. This simplifies later the actual application of the conditions, as we will detail in the next section.

We start by emptying the set of conditions (line 2). We then calculate the set Aut of automorphisms of the graph (line 3), and start adding conditions that when respected will reduce the above mentioned set to the identity map. Note that although calculating automorphisms is thought to be computationally expensive, in practice it was found to be almost instantaneous for the subgraph sizes used, and with `nauty` [27] we were able to test much bigger subgraphs (with hundreds of nodes) and obtain their respective automorphisms very quickly, in less than 1 second. Thus, this calculation is very far from being a bottleneck in the whole process of generating and using g-tries.

In each iteration, to add a new condition, the algorithm finds the minimum index m corresponding to a vertex that still has at least another equivalent node (line 5). It then adds conditions stating that the vertex in position m should have an index lower than every other equivalent position (lines 6 and

Algorithm 4 Symmetry breaking conditions for graph G

Require: Graph G
Ensure: Symmetry breaking conditions of G
1: **function** GTRIECONDITIONS(G)
2: $\quad Conditions := \emptyset$
3: $\quad Aut :=$ SETAUTOMORPHISMS(G)
4: $\quad$ **while** $|Aut| > 1$ **do**
5: $\quad\quad m :=$ minimum $v : \exists\, map \in Aut, map[v] \neq v$
6: $\quad\quad$ **for all** $v \neq m : \exists\, map \in Aut, map[m] = v$ **do**
7: $\quad\quad\quad$ add $m < v$ to $Conditions$
8: $\quad\quad Aut := \{map \in Aut : map[m] = m\}$
9: $\quad$ **return** $Conditions$

7), which in fact fixes m in its position. We choose the minimum index vertex in order that, when searching, we can know as soon as possible that a certain partial match is not a suitable candidate. Note that lower indexes mean lower depths in the g-trie.

After this, the algorithm reduces Aut by removing the mappings that do not respect the newly added connections, that is, the ones that do not fix m. It repeats this process until there is only the identity left Aut' (line 4), and finally returns all the generated conditions (line 9). In the case of the graph of figure 8, this algorithm would create the exact same set of conditions as depicted there. Figure 9 illustrates the symmetry conditions found.

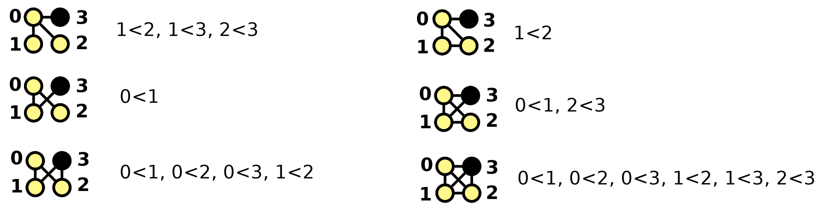


Fig. 9: Symmetry conditions computed for all possible undirected 4-subgraphs. All the graphs are in the `GTCanon` canonical form.

3.4.3 Using the Conditions to Constraint Search

In order to use the symmetry breaking conditions in g-tries, we store the graph symmetry conditions in all the nodes corresponding to its g-trie path. The matching algorithm can then determine if the partial subgraph constructed respects the conditions of at least one possible descendant g-trie node, that is, there is at least one possible subgraph that can still be constructed by expanding the current partial match and still obeys to the symmetry conditions.

Algorithms 5 details how the insertion of the conditions is done. For the sake of understanding, we repeat several lines that are the same as the previ-

ous insertion algorithm, and indicate which lines are new. Basically, the only difference is that we now compute the symmetry breaking conditions (line 3) and then we store those conditions along the g-trie path that leads from the root to the final graph node (line 18).

Algorithm 5 Inserting a graph G in a g-trie T [with symmetry breaking]

```
Require: Graph G and G-Trie T
Ensure: Inserts graph G in G-Trie T

 1: procedure gtrieInsert(G, T)
 2:     M := adjacency matrix of GTCanon(G)
 3:     C := symmetryConditions(G)                        ▷ new code
 4:     insertCondRecursive(M, T, 0, C)                   ▷ new function header
 5: procedure insertCondRecursive(M, T, k, C)            ▷ new function header
 6:     if k = numberRows(M) then
 7:         c.isGraph = True
 8:     else
 9:         for all children c of T do
10:             if (c.out = first k + 1 values of k-row of M) AND
11:                 (c.in = first k + 1 values of k-column of M) then
12:                 insertCondRecursive(M, c, k + 1, C)   ▷ new f. header
13:                 return
14:         nc := new g-trie node
15:         nc.in := first k + 1 values of k-row of M
16:         nc.out := first k + 1 values of k-column of M
17:         nc.isGraph = False
18:         nc.addConditions(C)                           ▷ new code
19:         T.insertChild(nc)
20:         insertCondRecursive(M, nc, k + 1, C)          ▷ new function header
```

With the symmetry breaking conditions placed in the g-trie nodes, we are now able to search more efficiently for subgraphs. Algorithm 6 details how a census with symmetry breaking is done. The same conventions of the insertion algorithm are followed, meaning that we repeat lines that were already on the previous census algorithm and we indicate the new lines of code.

The basic difference is that we now only accept matchings that respect at least one of the sets of conditions stored, that is, can still correspond to a descendant graph (line 11). Moreover, we detect the minimum possible index for the current node being matched (line 12) and use it to further constraint the generation of candidates (lines 13 and 15). This is easy to find since in all stored conditions $a < b$ the node being matched is always on the b position and therefore the minimum index is one plus the largest index of nodes that must have lower indexes than this new node. If the neighbours of each network node are sorted (which can be done only once before starting the census), we

Algorithm 6 Census of subgraphs of T in graph G [with symmetry breaking]

Require: Graph G and G-Trie T
Ensure: All occurrences of the graphs of T in G

```
1:  procedure gtrieMatchCond(T, G)
2:      for all children c of T.root do matchCond(c, ∅) new function header
3:  procedure matchCond(T, V_used)                                  ▷ new function header
4:      V := matchingVerticesCond(T, V_used)                        ▷ new function header
5:      for all node v of V do
6:          if T.isGraph ∧ T.ConditionsRespected() then             ▷ updated code
7:              output V_used ∪ {v}
8:          for all children c of T do
9:              matchCond(c, V_used ∪ {v})                          ▷ new function header
10: function matchingVerticesCond(T, V_used)                        ▷ new function header
11:     if NOT ∃C ∈ T.cond : V_used respects C then return ∅       ▷ new code
12:     label_min := minimum possible index for current position    ▷ new code
13:     if V_used = ∅ then V_cand := {v ∈ V(G) : v ≥ label_min}     ▷ updated code
14:     else
15:         V_conn := {v : v ∈ N(V_used) ∧ v ≥ label_min}          ▷ updated code
16:         m := m ∈ V_conn : ∀v∈ V_conn, |N(m)| ≤ |N(v)|
17:         V_cand := {v ∈ N(m) : v ∉ V_used}
18:     Vertices := ∅
19:     for all v ∈ V_cand do
20:         if ∀i∈[1..|V_used|]:
21:             T.in[i] := G_Adj[V_used[i]][v]   ∧ T.out[i] = G_Adj[v][V_used[i]] then
22:             Vertices := Vertices ∪ {v}
23:     return Vertices
```

can use this minimum to discover that further smaller neighbours will never be suitable candidates. In the end we must verify that a particular matching respects all symmetry breaking conditions for that subgraph. If the end of the graph is in a g-trie leaf, this step can be skipped, since for sure the conditions are respected. However, if the g-trie node is not a leaf, the search might have arrived there because of the conditions of another descendant subgraph, and therefore the algorithm must assure that the conditions for that particular subgraph are respected.

The method for choosing the minimum possible index for the current node that still respects the symmetry conditions ($label_{min}$ on line 12) consists in computing, for each set of conditions, the maximum already mapped node that must be smaller than the current node, and then we pick the minimum of these. Illustrating with an example, imagine that we are trying to match a node to position 2, the symmetry conditions are $\{\{0{<}1, 1{<}2\}, \{0{<}2, 1{<}2\}\}$, and the current matching is $V_{used} = \{34, 12\}$, which means that network node

number 34 is matched to position 0, and node 12 is matched to position 1. Since we are matching position 2, only the conditions with 2 matter. For the first set this is 1<2, which means that the current node must be larger than 12. For the second set, we must take into consideration 0<2 and 1<2, which means that the node must be simultaneously larger than 34 and 12. We take the maximum, which is 34. Afterwards, we know that the node must be larger than 12 (first set) or larger than 34 (second set), and therefore we take the minimum, and we would have $label_{min} := 12$. If any set of conditions is empty, than $label_{min} := 0$, that is, there is no minimum for the index of the node.

With these two symmetry aware algorithms (insertion and census), a subgraph will only be found once. All other possible matchings of the same set of vertices will be broken somewhere in the recursive backtracking. Moreover, since the conditions generation algorithm always create conditions of the minimal indexes still not fixed (line 5 of algorithm 4), the census algorithm can discover early in the recursion that a condition is being broken, therefore cutting branches of the possible search tree as soon as possible.

3.4.4 Reducing the Number of Conditions

By using the last two algorithms, we may end up having a large number of symmetry conditions on a single g-trie node, since it can have a very large number of descendants. This can have a severe impact on memory costs and influence the performance, and we should reduce as much as possible this cost. With that in mind, we use four steps to filter and reduce the symmetry conditions.

Step #1 reduces the set of conditions by using the transitive property of the "less" relationship, and in the cases where $a < b$, $a < c$ and $b < c$ are in the set of conditions, we remove the condition $a < c$.

Step #2 reduces the conditions to the ones that matter to that particular node. This means that if we are at a certain g-trie depth, conditions in which one of the elements is bigger than the depth are discarded, that is, the conditions that are referring to descendant nodes that are still not matched.

Step #3 discards sets of conditions that are redundant. Since each g-trie node has a group of sets of conditions (one for each descendant graph) and it must assure that at least one of those sets is respected (meaning that that at least one descendant graph is achievable), we search the group for sets that are redundant, in the sense that they include another one, and we remove those sets. One example can be given using the sets of conditions $\{\{0<1\}, \{0<1, 1<2\}\}$. In this case we can discard the second set since it includes the first set, that is, if a partial graph respects the second set, it would also respect the first set, and therefore the second set is redundant if the algorithm is trying to assure that at least one of the condition sets is respected.

Step #4 is the final one and removes conditions that are already assured. It is applied after having all graphs already inserted in the g-trie, and all other filtering steps are already made. If at any g-trie node there is a specific condition $a<b$ that is included in all of the sets, we can be assured that this

condition is certainly respected and all descendant nodes do not need to verify it again. As an example, consider the set $\{\{0<1\}, \{0<1, 1<2\}\}$. The condition 0<1 is in every set and therefore we remove it from all descendant g-trie nodes.

By following the described filtering steps, the resulting g-trie has a much reduced number of stored conditions, which will not only save memory, but will also be more efficient for census computation, as there are less conditions to be verified. Figure 10 shows an example filtered g-trie.

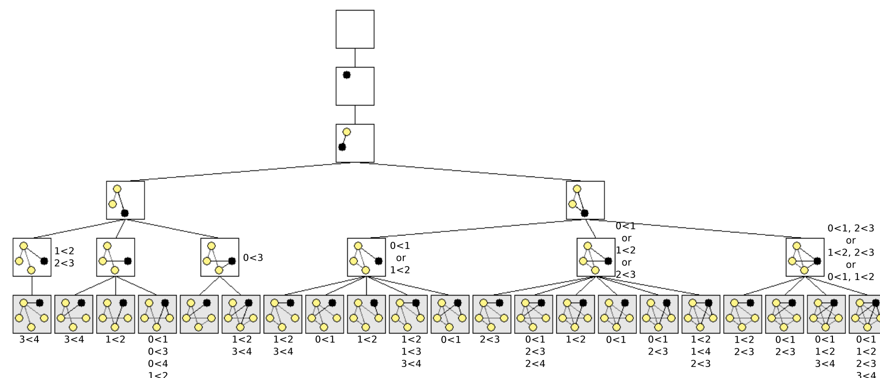


Fig. 10: A g-trie containing all 21 undirected 5-subgraphs, with filtered conditions in place.

3.5 Motif Discovery with G-Tries

With the g-tries algorithms described in this section it is now possible to discover motifs. The main flow of all exact network motifs algorithms is to calculate a census of subgraphs of a determined size k in the original network, then generate a set of similar random networks, followed by the calculation of the census on all of those, in order to assess the significance of the subgraphs present in the original network.

The generation of the random networks themselves (normally done by a Markov chain process [30]) is just a very small fraction of the time that the census takes. Computing the census on all random networks is the main bottleneck of the whole process (there can be hundreds of random networks) and `g-tries` can help precisely in this phase. In order to use g-tries we give two possible approaches:

- **G-Trie use only** - generate all possible graphs of a determined size (for example using McKay's `gtools` package [28]), insert them all in a g-trie and then match to the original network. Create a new g-trie only with the subgraphs found, and then match it to the random networks.
- **Hybrid approach** - use another network-centric method to enumerate the subgraphs in the original network, like the efficient `ESU` algorithm. Create a g-trie only with the subgraphs found, and then apply g-trie matching to the random networks.

In both cases we will only be trying to discover in the random networks the subgraphs that appear in the original network, and not spending execution time trying to find subgraphs that are not interesting from the motifs problem point of view. Also note that g-tries could easily be extended to really find and store all occurrences, instead of simply counting them. More than that, g-tries are general enough to be used for any situation in which we have a set of initial graphs that we need to consider as subgraphs of a larger network.

4 Experimental Evaluation

We did an extensive experimental evaluation of our proposed approach, showing that for all the data sets used our algorithm outperforms all competing algorithms, being from two to three orders of magnitude faster. For all the tests, we used Xeon 5335 processors with 6 GB of RAM. All code[1] was developed in C++ and compiled with gcc 4.1.2. Execution times measure wall clock time, meaning the real time elapsed from the start to the end of the respective computation. The time unit used is the second.

We use a large set of real representative networks from several domains, focusing on getting diversity in topological features and scientific fields. We now describe each of these networks, indicating the source and providing a name for future reference.

- **Biological networks:**
  - `ppi`: an undirected budding yeast (*S. cerevisiae*) protein-protein interaction (PPI) network [7]. Source: [5].
  - `neural:` a directed neural network of the small nematode roundworm *C. elegans*, describing its nervous system [56,53]. Source: [32].
  - `metabolic:` a directed metabolic network of the small nematode roundworm *C. elegans* [12]. Source: [4].
- **Social networks:**
  - `coauthors:` an undirected network describing coauthorship between scientists working on network theory [34]. Source: [32].
  - `dolphins:` an undirected social network of frequent associations between 62 dolphins in a community living near New Zealand [26]. Source: [32].
  - `links`: a directed network of hyperlinks between weblogs on US politics [1]. Source: [32].
  - `company`: a directed ownership network of companies in the telecommunications and media industries [36]. Source: [5].
- **Physical networks:**
  - `circuit:` an undirected network representing an electronic circuit. Source: auxiliary material of [30].
  - `power:` an undirected network representing the topology of the Western States Power Grid of the United States of America [53]. Source: [32].

[1] A preliminary version of the g-tries source code is available at: `http://www.dcc.fc.up.pt/gtries/`

  - `internet:` a symmetrized snapshot of the structure of the Internet at the level of autonomous systems, reconstructed from BGP tables posted by the University of Oregon Route Views Project. This data was compiled by Mark Newman in July, 2006. Source: [32].
- **Semantic networks:**
  - `foldoc`: Foldoc is an online dictionary of computing terms [16]. This is a directed network where an edge (X, Y) means that term Y is used to describe the meaning of term X. Source: [5].
  - `oldis`: Oldis is the Online Dictionary of Library and Information Science [40]. It is a directed network built in the same way as `foldoc`. Source: [5].

Table 1 summarizes the topological features of all these networks. They are shown ordered by increasing number of nodes. All networks are simple unweighted graphs, in accordance with the definition. Some simplifications on the original networks were done as necessary. Self-loops were discarded, multiple edges were transformed in a simple unique edge between the two nodes, and all weights were ignored.

Network	Group	Directed	$\|V(G)\|$	$\|E(G)\|$	Nr. of Neighbours	
					Average	Max
`dolphins`	social	no	62	159	5.1	12
`circuit`	physical	no	252	399	3.2	14
`neural`	biological	yes	297	2,345	14.5	134
`metabolic`	biological	yes	453	2,025	8.9	237
`links`	social	yes	1,490	19,022	22.4	351
`coauthors`	social	no	1,589	2,742	3.5	34
`ppi`	biological	no	2,361	6,646	5.6	64
`odlis`	semantic	yes	2,909	18,241	11.3	592
`power`	physical	no	4,941	6,594	2.7	19
`company`	social	yes	8,497	6,724	1.6	552
`foldoc`	semantic	yes	13,356	120,238	13.7	728
`internet`	physical	no	22,963	48,436	4.2	2,390

Table 1: Topological features of the networks used to test the algorithms.

For each network we also provide the average and the maximum number of neighbours per node, since this can provide an insight on how the number of subgraph occurrences grows as its size increases. Generally speaking, the more neighbours a single node has, the more subgraph occurrences it will participate in. Therefore, a larger number of neighbours implies a larger growth ratio on the number of subgraph occurrences.

In order to perform some systematic tests on our algorithms, we also use synthetic networks, artificially generated to present some topological constraints. In particular we will use a benchmark social network proposed by Lancichinetti et al. [24], that was originally created with the purpose of evaluating community detection algorithms. It provides undirected or directed networks with features close to a real social network and is fully customizable.

Since we are using a very large set of possible networks, we will not use all of them for every aspect tested. For the sake of simplicity and ease of reading, some sections will only present some representative cases.

4.1 Competing Algorithms

We will compare the performance of our algorithm against the main state-of-art algorithms that were designed to compute the same task. `mfinder` is not used since it is significantly slower, by several orders of magnitude, than all other major algorithms, as it is shown for example in [55]. For instance, we empirically verified that it was more than 1000x slower than g-tries when computing the 5-subgraph census of the undirected network `ppi` (an undirected network) and that it was more than 2000x slower than g-tries when computing the 4-subgraph census of the directed network `company`.

In order to compare with the network-centric approach, we will use both the `ESU` and `Kavosh` algorithms. We use our own implementation of the algorithms, so that the whole code infra-structure is the same with the exception of the algorithm itself, allowing for better and fairer comparison. For example, both `ESU` and `Kavosh` use the `nauty` third-part algorithm for isomorphism calculation, but the author's implementation use different versions of it. In our implementation, we can make sure that this is not the case and the same `nauty` version is used in both algorithms.

Great care was taken in the implementation of these algorithms, with special emphasis on guaranteeing the best performance possible. We also downloaded, compiled and used as a guide the original source code provided by the authors of `ESU` (fanmod tool[2]) and Kavosh[3], and compared the execution times with our own implementation, to certify that we were not slowing down the algorithms. Both original implementations use C++.

Table 2 shows this comparison. The Fanmod tool has a mandatory graphical user interface that is not available at the dedicated cluster we used. Therefore, for this experiment, we used a personal computer with an Intel Core i5-450M processor running at 2.4GHz with 4GB of DDR3 memory. We show the execution times (measured in seconds) for a complete k-subgraph census on four networks, two undirected and two directed, with increasing k.

We can observe that our `Kavosh` implementation is very similar to the original implementation, with a slight gain. With respect to `ESU`, our implementation is faster than the original, with an average speedup close to 2x. This gain seems to be even more pronounced in the undirected networks (`coauthors` and `power`). Moreover, our own implementation of `ESU` does not impose strict limits on the motif size, while the original source code is limited to 8 nodes. We also implemented `RAND-ESU`, the sampling version of this algorithm.

Regarding the subgraph-centric approach we will use `Grochow`. However, given that `Grochow` is implemented in Java, we do not show a comparison with

[2] Fanmod is available at `http://theinf1.informatik.uni-jena.de/~wernicke/motifs/`

[3] Kavosh source code is available in `http://lbb.ut.ac.ir/Download/LBBsoft/Kavosh/`

network	k	ESU		Kavosh	
		Original	Our Implementation	Original	Our Implementation
coauthors	3	0.02	0.01	0.06	0.05
	4	0.17	0.05	0.06	0.05
	5	1.16	0.41	0.46	0.41
power	3	0.04	0.02	0.01	0.01
	4	0.18	0.07	0.07	0.07
	5	0.81	0.31	0.35	0.31
neural	3	0.08	0.03	0.03	0.08
	4	2.09	1.29	1.29	1.26
	5	76.64	69.27	70.84	63.12
metabolic	3	0.12	0.04	0.05	0.05
	4	5.66	3.15	3.22	3.35
	5	386.52	270.40	283.55	272.67

Table 2: Execution time of our ESU and Kavosh implementations.

our implementation. Generally speaking, Java is considered to be slower than C++ [19], although each case may have its particular behaviour. Nevertheless, our own implementation provides a much fairer comparison to the others, in the sense that it is written in the same language and with the same code-infrastructure.

MODA is not used for comparison since it has a very high memory cost (it needs to store the actual occurrences of all k-subgraphs). This hinders its applicability, specially when the number of subgraphs found becomes large. Instead, we compare with algorithms that base their computation only on storing subgraph counts, as is the case with g-tries, ESU, Kavosh and Grochow.

4.2 G-Tries Creation

We will start by evaluating the time it takes to construct a g-trie and its ability to compress the topological information of the original set of graphs.

We first generated all possible simple undirected k-graphs, increasing k, using the nauty tools. We then experimented reading all those graphs from a file and inserting them in an initially empty g-trie. No step was bypassed, such as computing the symmetry breaking conditions, and filtering these conditions in the g-trie. Note that on a real usage case, this could be avoided by pre-computing these conditions, but the intention of this experiment is to show how much time it takes to create a g-trie on the fly.

Table 3 shows the results obtained. We took note on the number of graphs, the compression ratio (see Equation 1), the time it takes to create the g-tries and the average number of subgraphs stored per second. We stop at the first k where the number of different graphs is greater than one million. The smallest k used throughout this section is 3, since those are the simplest subgraphs and considering less would correspond to look for single vertices ($k = 1$) or for edges ($k = 2$).

k	Nr. Graphs	Compression	Time (s)	Graphs/sec
3	2	33.33%	< 0.001	37,736
4	6	58.33%	< 0.001	35,088
5	21	70.48%	< 0.001	29,494
6	112	77.98%	0.004	25,524
7	853	82.21%	0.040	21,514
8	11,117	85.01%	0.588	18,917
9	261,080	87.12%	15.766	16,559
10	11,716,571	88.78%	917.379	12,772

Table 3: Execution time for inserting all undirected k-graphs in a g-trie.

The table shows that the number of k-graphs grows exponentially and, at the same time, the compression also increases, meaning that more common substructure is identified. This is mainly because there are more graphs and therefore more potential for overlapping of structure. For example, all subgraphs will naturally share the same root g-trie ancestor, containing a single node without a connection to itself. Regarding the execution time, we can see that it is initially almost instantaneous, but it grows exponentially with the number of subgraphs. The results show that this approach could become prohibitive as k grows, but it should be noted that the number of graphs would become so large that even the g-trie itself containing all k-graphs would be too big to fit in main memory (for $k = 11$, the number of different subgraph types would be 1,006,700,565). Note however that g-tries can provide support for larger graphs, provided that we do not need to store all possible k-graphs, but instead we are storing a specific (smaller) set [43].

We now show the results for creating a g-trie with all directed k-graphs in the same way, stopping again as soon as the number of graphs exceeds one million. The results can be seen in Table 4.

k	Nr. Graphs	Compression	Time (s)	Graphs/sec
3	13	56.41%	< 0.001	156,627
4	199	71.98%	0.002	83,789
5	9,364	79.07%	0.150	62,535
6	1,530,843	83.03%	50.254	30,462

Table 4: Execution time for inserting all directed k-graphs in a g-trie.

The results are very similar to what happens with undirected networks, with growing compression and again an exponential growth in the number of graphs and execution time, exposing the same virtues and limitations.

These complete g-tries containing all possible k-graphs can be reused and they could be stored in the file system ready to be uploaded to main memory without really computing them. Tables 5 and 6 show execution times when reading files of g-tries containing all possible k-graphs. Again we show the number of graphs and the ratio of graphs per second. In this experiment we

also take note of the size of the file (in bytes) containing the respective g-trie, and the number of bytes per graph.

We can observe that reading the previously computed g-trie from a file is obviously much faster than creating the g-trie on the fly, and the main bottleneck is the file size itself. The number of necessary bytes per graph decreases as k increases since the compression ratio also increases.

k	Nr. Graphs	Time (s)	Graphs/sec	File Size (bytes)	Bytes/graph
3	2	< 0.001	83,333	42	21.0
4	6	< 0.001	193,548	97	16.2
5	21	< 0.001	437,500	281	13.4
6	112	< 0.001	783,217	1,287	11.5
7	853	< 0.001	1,055,693	8,935	10.5
8	11,117	0.010	1,059,670	100,182	9.0
9	261,080	0.209	1,249,288	2,020,172	7.7
10	11,716,571	8.277	1,415,603	79,571,853	6.8

Table 5: Execution time for reading a g-trie with all undirected k-graphs.

k	Nr. Graphs	Time (s)	Graphs/sec	File Size (bytes)	Bytes/graph
3	13	< 0.001	351,351	126	9.7
4	199	< 0.001	1,463,235	1,398	7.0
5	9,364	0.004	2,084,131	54,354	5.8
6	1,530,843	1.384	1,106,481	8,113,436	5.3

Table 6: Execution time for reading a g-trie with all directed k-graphs.

For motif computation, as discussed before, we will typically find all k-subgraphs of the original network and then populate a g-trie only with those that both appear in the network and are meaningful for motif computation, that is, that appear at least a given minimum of times. This means that in practice we will be inserting a dynamic set of graphs, which is generally just a small percentage of the whole set of possible subgraphs of a determined size, which would mean much smaller creation times. Note that some of the computational costs could be further reduced, for instance by pre-computing the symmetry conditions, avoiding the necessity of computing the automorphisms.

In practice, this means that we can achieve a very fast g-trie creation, with the g-trie creation step not being the bottleneck in the motif computation. In fact, this time will only start to be more meaningful as the number of graphs reaches a number so high that their g-trie representation cannot be stored in main memory. This is the main limitation of the g-trie creation process as it is right now, since all algorithms assume that the g-trie in itself fits in memory.

Given this, for the last experiment in this section, we took note of the total memory spent by a g-trie residing in memory during motif computation, using the `valgrind` tool [31]. Tables 8 and 7 show the amount of memory needed for a g-trie containing all possible k-graphs, as well as the number of stored graphs and the average memory per graph.

k	Nr. Graphs	Memory (bytes)	Bytes/graph
3	2	814	407.0
4	6	2,294	382.3
5	21	7,172	341.5
6	112	34,852	311.2
7	853	241,294	282.9
8	11,117	2,899,538	260.8
9	261,080	63,504,120	243.2
10	11,716,571	2,680,803,240	228.8

Table 7: Memory needed for a g-trie containing all undirected k-graphs.

k	Nr. Graphs	Memory (bytes)	Bytes/graph
3	13	3,004	231.1
4	199	38,210	192.0
5	9,364	1,635,190	174.6
6	1,530,843	260,796,274	170.4

Table 8: Memory needed for a g-trie containing all directed k-graphs.

We can observe that, as expected, the needed memory grows exponentially with the number of stored graphs. The average memory per graph decreases as k grows because of larger compression ratios. When comparing this with the number of bytes per graph needed to store the g-trie in a file, we can see that the actual memory needed for the computation itself is much larger. Several factors contribute for this, such as the overhead introduced by using C++ objects or the added extra information, like the actual frequencies found.

4.3 Effect of Canonical Labeling

As explained in detail in Section 3.3.2, the canonical representation will influence the topology of the g-trie and therefore will have a great impact in the efficiency of the census algorithm. In order to empirically justify our option for the canonical labeling, we show the execution time of a complete k-subgraph census using a g-trie created with four different methods:

- **GTCanon:** our custom built `GTCanon` labeling algorithm, as described, that favors in lower levels nodes with more connections to ancestor nodes in the g-trie.
- **Larger:** the lexicographically largest possible adjacency matrix, that will induce high compression in the g-trie.
- **Random:** a deterministic pseudo-random labeling (obtained by fixing the seed) applied after the `nauty` canonization. We create it by a sequential random choice of a node that is connected to an ancestor node (when possible). This is to ensure minimum efficiency, because a purely random labeling would create many unconnected nodes, for which any unused network node would be a suitable candidate. This would exponentially in-

crease the execution time needed for computing a census, as was explained in Section 3.3.2.

- **Opposite:** a labeling algorithm expressing the opposite of `GTCanon`, that chooses the nodes with less connections for the lower levels of the g-trie. As in the `Random` labeling case, we ensure connectivity in order to guarantee minimum efficiency.

Table 9 shows the results obtained for two networks, one directed and one undirected, for a representative set of sizes k so that the computation is not instantaneous but at the same time small enough so that even the slowest labeling method takes only a few minutes. We computed the census by using a g-trie with all possible k-subgraphs.

	network	`circuit`			`metabolic`	
	k	7	8	9	4	5
Execution time (s)	GTCanon	0.037	0.202	1.230	0.178	10.611
	Larger	0.075	0.554	4.202	0.224	19.108
	Random	0.251	2.846	45.814	0.292	40.319
	Opposite	0.628	8.723	145.116	0.351	55.927
Compression Ratio	GTCanon	82.21%	86.33%	87.12%	71.98%	79.07%
	Larger	83.44%	86.33%	88.32%	72.86%	79.48%
	Random	67.43%	68.01%	69.00%	68.84%	74.15%
	Opposite	78.80%	81.87%	84.28%	72.61%	79.18%

Table 9: Effect of canonical labelings on k-census computation.

We can observe that our chosen strategy has the best behaviour (smaller execution times) for every pair of network and subgraph size, both in undirected and directed cases.

The lexicographically largest adjacency matrix (`larger`), which is more costly to compute, produces higher compression in the g-tries, but this does not have a proportional impact in lowering the execution time. Indeed, a greater compression ratio is desirable, but in itself it does not guarantee better performance and the `GTCanon` labeling takes advantage of the fact that we know the census algorithm and therefore can optimize it for discovering, as soon as possible, that a set of nodes will not match to a subgraph. The `random` labeling shows that the compression ratio and census efficiency cannot be assumed, and thus efficiency must be obtained by choosing an appropriate labeling. Finally, `opposite`, the reverse of our chosen strategy, is even worst than the random case, further substantiating the claim that our labeling choice has a positive effect on the efficiency of the census.

4.4 Effect of Symmetry Breaking Conditions

If we did not use the symmetry breaking conditions detailed in Section 3.4.2, all automorphisms of each subgraph isomorphic class would be found and the

census would be substantially slower. Our filtering of symmetry conditions, as previously explained, reduces the memory needed for storing the g-trie, and improves the execution times.

Table 10 shows execution times for a k-census on the same two networks used in the previous section, with a g-trie with all k-subgraphs, varying the symmetry breaking conditions used:

- **Normal:** our standard use of symmetry breaking conditions.
- **No filtering:** symmetry breaking conditions are used, but we do not minimize their number by following the 4 filtering steps described on section 3.4.4.
- **No conditions:** No symmetry breaking conditions are used at all.

	network	`circuit`			`metabolic`	
	k	7	8	9	4	5
Execution time (s)	Normal	0.037	0.202	1.230	0.178	10.611
	No filtering	0.062	0.590	38.562	0.238	15.638
	No conditions	0.205	1.592	13.155	0.460	56.414

Table 10: Effect of symmetry breaking conditions on k-census computation.

As expected, not using any conditions slows the algorithm. Not filtering also slows down, albeit by a smaller margin. However, for bigger g-tries, the amount of unfiltered symmetry conditions starts to be so large that the algorithm becomes even slower than the g-trie without any conditions (see $k = 9$ for `circuit`). This showcases the need for reducing the number of conditions and the validity of our filtering process.

4.5 Asymptotic Behavior

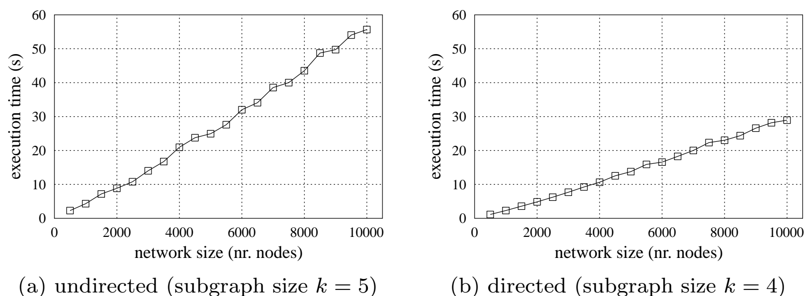
We will now have a look at the empirical asymptotic behavior of the g-trie census algorithm as the size of both the networks and subgraphs grows.

For testing network growth, we will use the described synthetic social benchmark network, as it allows us to slowly grow network size n while preserving the same topological characteristics. As before, we will be computing k-census with g-tries containing all possible subgraphs. The network is customized with the following parameters: average degree 20; minimum community size 20, maximum community size 50, mixing parameter 0.1 (for more information on these see [25] and the networks generation source code `Readme` file[4]).

Figure 11 shows the execution time for computing a full k-subgraph census by using a g-trie with all possible k-subgraphs. We vary the size from 500 to 10,000 nodes, with increments of 500 nodes and we use both undirected

[4] The source code is available at `http://sites.google.com/site/andrealancichinetti/files`

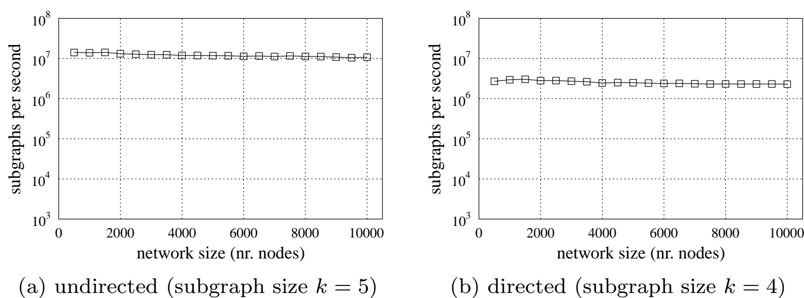
and directed versions of the network. We opted for choosing k, the subgraph size, so that the execution times are within one minute. For better legibility, both graphs use the same scale, but one should note that in the case of the undirected network we are computing subgraphs of size 5, and in the directed network subgraphs of size 4.



(a) undirected (subgraph size $k = 5$) (b) directed (subgraph size $k = 4$)

Fig. 11: Execution time for a census on a social network as the number of nodes increases.

We can see that the execution time appears to have a linear relation to the network size, for this particular network. Remember that g-tries (and for that matter all other current major motif detection algorithms) must explicitly pass in every subgraph occurrence in order to have an exhaustive perfect count. Taking this into account, a more telling statistic is to check the subgraph discovery ratio, that is, the number of occurrences found per second. A constant value would be the desirable situation, since the algorithm (such as the others) is by now limited by design to those occurrences. Figure 12 illustrates the subgraph discovery ratio for the same experiment.



(a) undirected (subgraph size $k = 5$) (b) directed (subgraph size $k = 4$)

Fig. 12: Subgraph discovery ratio for a social network as the number of nodes increases.

We can see that our algorithm is able to maintain a steady flow on the number of subgraphs found per second, without losing much performance as

the network size grows. Note however that the ratio is different in the two cases, since it is quite different to look for directed or undirected subgraphs, and their size is different.

In general, two different networks will induce different discovery ratios, as this depends heavily on the topology of the network itself. For example, a network with dense regions would be more prone to a bigger ratio, with many subgraphs packed together, increasing the amount of common topologies that the g-trie can take advantage of. However, at the same time, it would have more subgraph occurrences, increasing the total execution time. Table 11 and Table 12 give a more practical view of this effect, by showing the subgraph ratio for different networks on subgraphs of the same size, when again computing a census with a complete g-trie. Note the differences, but also note the magnitude of the ratio, that stays between 10^6 and 10^7. This does not mean however that, for every network, g-tries subgraph discovery ratio will stay within this margin.

Network	Execution Time	Subgraph Occurrences	Subgraphs/sec
`dolphins`	3.027	12,495,833	4,128,240
`circuit`	1.230	13,512,688	10,985,192
`coauthors`	448.071	886,423,840	1,978,312
`power`	18.444	183,453,978	9,946,388

Table 11: Subgraph discovery ratio for 9-census on undirected networks.

Network	Execution Time	Subgraph Occurrences	Subgraphs/sec
`neural`	3.778	43,256,069	11,448,911
`metabolic`	10.611	195,573,511	18,431,064
`links`	971.269	7,347,672,714	7,562,022
`odlis`	713.630	8,655,784,561	12,129,235

Table 12: Subgraph discovery ratio for 5-census on directed networks.

In what concerns the execution time behaviour when we increase the size of the motifs for the same network, what generally happens is that we have a subgraph discovery ratio that very slowly starts degrading as the size increases, but keeping the same magnitude.

4.6 G-Tries Comparison with Other Algorithms

We now provide a thorough comparison of g-tries, with other competing algorithms, namely `ESU`, `Grochow` and `Kavosh`, as explained in Section 4.1.

For the first set of tests we will fully enumerate all k-subgraphs present in the original network, doing the equivalent to the first step of any motif discovery process. In the case of network-centric algorithms (`ESU` and `Kavosh`),

this is simply running them. In the case of the subgraph-centric algorithms (`Grochow`), we queried all possible k-subgraphs, in the same way we used g-tries with all those subgraphs inserted.

In order to give a broad overview of how g-tries consistently outperform every other major motif algorithm, we used for comparison the entire set of 12 real complex networks. For each one of them we did a k-census, increasing k one by one starting with 3. In order to produce readable tables, we stopped when the slowest algorithm would take more than 5 hours to run and took note of the execution times and of the relative speedup of g-tries versus the other algorithms.

Tables 13 and 14 present the results obtained for the undirected and directed networks, respectively. All methods are identified by the first three letters of their name (`GTR` for g-tries, `ESU`, `KAV` for `Kavosh` and `GRO` for `Grochow`). Every experiment was executed at least three times and average execution times were recorded.

Network	k	k-census execution time (s)				Speedup of g-tries vs		
		GTR	ESU	KAV	GRO	ESU	KAV	GRO
`dolphins`	3	< 0.001	< 0.001	< 0.001	< 0.001	14.9x	14.3x	9.8x
	4	< 0.001	0.006	0.006	0.003	15.5x	15.6x	8.3x
	5	0.002	0.036	0.036	0.032	14.7x	14.5x	12.8x
	6	0.014	0.251	0.241	0.324	17.7x	17.0x	22.8x
	7	0.085	1.499	1.405	3.727	17.7x	16.6x	43.9x
	8	0.483	9.182	8.468	55.093	19.0x	17.5x	114.0x
	9	3.027	52.431	46.033	1323.984	17.3x	15.2x	437.4x
`circuit`	3	< 0.001	0.001	0.001	0.002	12.6x	12.4x	18.8x
	4	< 0.001	0.007	0.007	0.006	18.0x	18.1x	17.1x
	5	0.002	0.034	0.034	0.043	21.9x	21.8x	27.7x
	6	0.007	0.221	0.218	0.317	32.4x	32.1x	46.5x
	7	0.037	1.365	1.311	2.778	36.7x	35.3x	74.8x
	8	0.202	9.267	8.670	32.630	45.9x	42.9x	161.5x
	9	1.230	59.804	53.152	632.213	48.6x	43.2x	514.0x
`coauthors`	3	< 0.001	0.010	0.010	0.030	14.1x	14.4x	42.0x
	4	0.006	0.077	0.077	0.172	12.0x	11.9x	26.7x
	5	0.053	0.633	0.617	1.635	12.0x	11.7x	31.0x
	6	0.442	5.711	5.568	18.465	12.9x	12.6x	41.8x
	7	4.088	50.748	49.579	284.614	12.4x	12.1x	69.6x
	8	40.560	481.432	464.540	6669.196	11.9x	11.5x	164.4x
`ppi`	3	0.004	0.086	0.092	0.092	21.8x	23.4x	23.4x
	4	0.068	3.106	3.053	1.450	45.9x	45.1x	21.4x
	5	1.507	84.959	84.852	39.182	56.4x	56.3x	26.0x
	6	40.275	2922.555	2934.426	1092.221	72.6x	72.9x	27.1x
`power`	3	0.002	0.017	0.018	0.215	7.9x	8.3x	98.3x
	4	0.008	0.101	0.102	0.845	12.6x	12.7x	105.3x
	5	0.029	0.481	0.486	5.164	16.6x	16.8x	178.1x
	6	0.119	2.947	2.913	35.729	24.7x	24.4x	299.7x
	7	0.600	17.891	17.285	313.052	29.8x	28.8x	521.5x
	8	3.247	120.695	112.065	3840.250	37.2x	34.5x	1182.7x
`internet`	3	0.423	11.571	12.206	6.865	27.3x	28.8x	16.2x
	4	204.442	11044.788	10827.744	3390.107	54.0x	53.0x	16.6x

Table 13: Comparison of g-tries with other algorithms when doing a full k-census on original undirected networks.

Network	k	k-census execution time (s)				Speedup of g-tries vs		
		GTR	ESU	KAV	GRO	ESU	KAV	GRO
neural	3	0.004	0.043	0.047	0.031	12.3x	13.2x	8.9x
	4	0.112	1.989	1.981	1.676	17.7x	17.6x	14.9x
	5	3.778	107.646	111.350	124.329	28.5x	29.5x	32.9x
	6	128.364	5382.221	5837.072	17214.101	41.9x	45.5x	134.1x
metabolic	3	0.003	0.062	0.067	0.035	22.3x	24.0x	12.6x
	4	0.178	5.037	5.215	3.214	28.3x	29.3x	18.1x
	5	10.611	428.613	424.673	238.696	40.4x	40.0x	22.5x
links	3	0.080	1.086	1.110	0.719	13.6x	13.9x	9.0x
	4	9.427	124.496	124.852	110.434	13.2x	13.2x	11.7x
	5	971.269	16984.341	16895.159	16766.414	17.5x	17.4x	17.3x
odlis	3	0.062	1.027	1.105	1.015	16.5x	17.8x	16.3x
	4	5.394	237.232	237.193	145.371	44.0x	44.0x	26.9x
company	3	0.016	0.296	0.308	0.656	18.7x	19.5x	41.5x
	4	1.751	67.476	70.457	35.434	38.5x	40.2x	20.2x
	5	241.378	13460.332	13459.479	2127.840	55.8x	55.8x	8.8x
foldoc	3	0.352	2.720	2.799	20.343	7.7x	8.0x	57.8x
	4	18.338	379.087	393.359	1124.439	20.7x	21.5x	61.3x

Table 14: Comparison of g-tries with other algorithms when doing a full k-census on original directed networks.

The main fact to notice is that there is not a single case where g-tries performs worst than other method, which showcases the efficiency of our algorithm. G-Tries clearly outperform the other algorithms in all the complex networks used. Specific speedups obtained can vary from method to method and from network to network, but g-tries consistently perform better. The exact speedup obtained is related to the amount of common substructure in different occurrences of subgraphs in the networks, which depends on the topology of the original network.

For all the networks, the speedup is at least very close to an order of magnitude faster. If we consider the last computed subgraph size, there is only one network (`company`) where our algorithm is less than 10 times faster than all the other algorithms, and even so it is 8.8 times faster than the closest competitor. In the other cases it is much faster, like in the case of `circuit` or `neural`, where it is more than 40 times faster than all other approaches. Note also that the speedups have the general tendency of growing as the subgraph size increases, meaning that g-tries appear to scale a little better than the other algorithms. The only case were this is clearly not the case is `company`. Analyzing its topology, we can see that it is specially sparse (among all networks used, it is the one with the least average number) and, more than that, it has an almost tree-like structure, that decreases the amount of common substructure between different subgraph occurrences.

Regarding this experiment, one can observe that the behaviour of the network-centric methods (`ESU` and `Kavosh`) is very similar to each other, with only a marginal difference. `Grochow`, on the other hand, performs differently, having difficulties when the number of subgraph classes increases. This is also a problem with our approach, since increasing k will likely make the g-trie too big to fit in memory. However, remember that this last experiment pertained to the computation of the full k-census in the original network. As shown in Sec-

tion 3.5, we can opt for a hybrid approach and use a network-centric method on the original network, and only after that populate the g-trie with the found classes of subgraphs, which typically will be a much smaller percentage of the whole set of possible subgraphs.

Moreover, in the current motif discovery flow, the real bottleneck comes after the first census, that is, when the frequencies must be computed for the ensemble of similar random networks. Since the size of this set is normally large (one hundred is a typical minimum use case) and since the random networks have the same number of nodes and edges of the original network, the time spent on this step of the computation takes precedence over everything else. It is therefore important to test the algorithms on randomized networks generated in this way.

In order to do this test, we applied a random Markov-Chain process like it was done in [30], with 3 swaps per edge and computed the census of all subgraphs that appeared on the original network. Note that the motif definition allows one to specify the minimum frequency that a subgraph must have in order to be considered a motif. In a real use case we can discard some subgraphs that are below this threshold and not count them on the randomized networks. However, for the sake of a more complete experiment, here we consider that this threshold is zero, meaning that every subgraph class that appears at least once in the original network is considered.

We use the same randomized networks in all algorithms, obtained by using the same pseudo-random number generation seed. For network-centric methods, the only option is to do an exhaustive census. For subgraph-centric methods capable of querying a single subgraph, namely `Grochow`, we only queried the pertinent subgraphs. For the g-tries, we created a g-trie precisely with the subgraphs for which we want to know the frequency in the random networks. We produced the same type of data as we had for the original network, but this time we ran the algorithms on 25 different random networks.

Considering that in a real use case, normally a minimum number of 100 random networks is used, we stopped when the average time per random network meant that computing the census in a set of 100 randomized networks with the slowest algorithm for that case would take more than 10 hours. In other words, the average execution time per randomized network must be smaller than 360 seconds. In the cases where this means that there would only be results for 3-subgraphs, we also report results on 4-subgraphs.

Tables 15 and 16 show the obtained average execution time for each similar randomized network, and the average speedup of g-tries against the other approaches. The standard deviation for the time spent in each randomized network, not shown in the table for the sake of legibility, is always smaller than 25% of the average, meaning that this average is a good indicator and that the time needed per randomized network does not suffer large variations.

First, observe that the computation time for a single randomized network tends to be larger than the time to compute the census in the original network. This is caused by the fact that randomization creates a network with more subgraph occurrences to be found, due to a more chaotic structure. Nevertheless,

Network	k	Avg. execution time per network (s)				Avg G-Tries Speedup		
		GTR	ESU	KAV	GRO	ESU	KAV	GRO
dolphins	3	< 0.001	< 0.001	< 0.001	< 0.001	17.7x	18.3x	10.7x
	4	< 0.001	0.008	0.008	0.003	24.8x	24.5x	9.4x
	5	0.003	0.068	0.065	0.036	25.7x	24.6x	13.6x
	6	0.021	0.576	0.572	0.393	27.8x	27.6x	19.0x
	7	0.155	4.403	3.988	4.211	28.4x	25.8x	27.2x
	8	1.121	32.399	30.140	44.293	28.9x	26.9x	39.5x
circuit	3	< 0.001	0.001	0.001	0.002	11.6x	12.0x	16.9x
	4	< 0.001	0.007	0.007	0.006	18.8x	19.3x	17.0x
	5	0.002	0.036	0.036	0.037	23.1x	23.2x	23.8x
	6	0.008	0.239	0.236	0.231	31.7x	31.3x	30.6x
	7	0.040	1.514	1.509	1.412	38.0x	37.9x	35.5x
	8	0.228	10.799	10.656	8.641	47.4x	46.8x	37.9x
	9	1.403	74.652	72.969	55.307	53.2x	52.0x	39.4x
coauthors	3	< 0.001	0.015	0.016	0.029	19.0x	20.1x	37.1x
	4	0.006	0.231	0.229	0.198	38.3x	38.0x	32.9x
	5	0.060	2.873	2.900	2.281	48.1x	48.5x	38.2x
	6	0.747	48.128	49.532	29.687	64.4x	66.3x	39.7x
ppi	3	0.004	0.097	0.096	0.089	25.9x	25.6x	23.6x
	4	0.072	3.657	3.659	1.516	50.6x	50.7x	21.0x
	5	1.862	115.058	115.597	47.668	61.8x	62.1x	25.6x
power	3	0.002	0.020	0.020	0.198	8.9x	8.8x	85.9x
	4	0.014	0.132	0.134	0.848	9.7x	9.9x	62.4x
	5	0.050	0.758	0.746	5.433	15.2x	15.0x	109.2x
	6	0.211	5.528	5.499	39.124	26.2x	26.1x	185.6x
	7	1.064	40.585	40.399	304.141	38.2x	38.0x	285.9x
internet	3	0.491	11.608	12.378	6.472	23.7x	25.2x	13.2x
	4	245.065	11495.764	11148.303	3614.261	46.9x	45.5x	14.7x

Table 15: Comparison of g-tries with other algorithms when computing a k-census in the similar undirected randomized networks.

Network	k	k-census execution time (s)				Speedup of g-tries vs		
		GTR	ESU	KAV	GRO	ESU	KAV	GRO
neural	3	0.004	0.049	0.055	0.033	13.5x	15.1x	9.2x
	4	0.143	2.496	2.431	2.330	17.4x	17.0x	16.3x
	5	5.506	139.271	140.120	158.658	25.3x	25.5x	28.8x
metabolic	3	0.003	0.059	0.061	0.019	22.7x	23.6x	7.3x
	4	0.109	3.955	4.086	1.271	36.3x	37.5x	11.7x
	5	4.418	308.769	304.222	68.257	69.9x	68.9x	15.4x
links	3	0.089	1.369	1.469	0.791	15.5x	16.6x	8.9x
	4	13.031	193.734	197.661	171.364	14.9x	15.2x	13.2x
odlis	3	0.075	1.173	1.222	1.165	15.6x	16.2x	15.4x
	4	8.848	259.565	262.449	199.647	29.3x	29.7x	22.6x
company	3	0.025	0.308	0.338	0.595	12.1x	13.3x	23.4x
	4	1.396	68.241	69.906	35.375	48.9x	50.1x	25.3x
foldoc	3	0.608	4.200	4.252	23.804	6.9x	7.0x	39.1x
	4	32.907	520.484	526.030	1661.945	15.8x	16.0x	50.5x

Table 16: Comparison of g-tries with other algorithms when computing a k-census in the similar directed randomized networks.

g-tries again consistently outperform all other algorithms, being substantially faster for every network and for every subgraph size. Moreover, the speedups versus the network centric methods are generally even better, partially due to the fact that we are now using smaller g-tries, populated with only the relevant subgraphs, instead of all possible subgraphs of that size. Observing the speedup against all other algorithms on the last subgraph sizes computed in each network, we can see that it always larger than 10x for all networks, and even larger than 20x for 8 out of the 12 networks. Also note that, again, the speedups show a tendency to increase as the subgraph size is incremented, even for the `company` case, because we are keeping the degree sequence, but not enforcing a tree-like structure. All of this points to even further gains for larger sizes.

Combining the observations on the original network with the observations on the randomized networks, we can estimate the time needed for discovering motifs, that will be essentially dominated by the time spent in the large set of similar networks. The result is a new much faster algorithm for motif detection.

Table 17 shows the average speedup obtained with g-tries when compared against the three other competing algorithms. We used as a basis the results of the last two tables (execution time for a census in the randomized networks) and computed the average speedup per network, giving the same weight to each network in the calculation of the global average. We show the g-tries speedup against each method category (network-centric: `ESU` and `Kavosh`; and subgraph-centric: `Grochow`) and subgraph type (undirected and directed). We also show this average speedup when considering all the subgraph sizes computed and also when only considering the last subgraph size computed for each network, which corresponds to larger execution times.

Subgraph Size	Subgraph Type	Average Speedup vs method type		
		All methods	Network-centric	Subgraph-centric
All Measured Sizes	All	30.1x	28.6x	34.4x
	Undirected	37.2x	33.5x	47.3x
	Directed	23.0x	23.8x	21.5x
Last Measured Size	All	44.3x	40.1x	49.9x
	Undirected	57.2x	46.0x	73.8x
	Directed	31.4x	34.1x	26.0x

Table 17: Average speedup of g-tries against competing algorithms.

Globally, g-tries are 30.1 times faster, on average, than its competing algorithms. The speedup is more pronounced on undirected subgraphs (37.2x) than on directed ones (23.0x). The speedup is also larger, on average, against subgraph-centric methods (34.4x) than against the network-centric methods (28.6x). This last observation is inverted in the case of directed subgraphs, where the speedup of network-centric methods (23.8x) is larger than the subgraph-centric one (21.5x).

If we just take into account the last subgraph size computed for each network (which factors bigger sizes, more subgraph occurrences, and larger execution times), the results are similar but the speedup obtained is even larger, with a global average speedup of 44.3x against the competing algorithms.

5 Conclusions

This article describes g-tries, a novel data-structure created for storing and finding subgraphs. Inspired by prefix trees, g-tries store the subgraphs in a multiway tree that encapsulates information about common substructure, with related subgraphs being stored in common branches. We detailed fast algorithms for creating g-tries, using a customized canonical labeling procedure, and for computing the frequencies of the graphs stored in the g-trie as subgraph instances of another larger network. We described how we use symmetry breaking conditions in conjunction with the common topologies of the stored subgraphs in order to devise an efficient non-redundant methodology. We also empirically evaluated our algorithm on a large set of representative networks, analyzing in detail the efficiency and scalability of our approach. We have shown that g-tries can consistently outperform the previous best methods, thus pushing the limits in the applicability of subgraph frequency discovery.

Taking advantage of the tree based nature of g-tries, we could also use a sampling methodology capable of trading accuracy for even faster execution times, further improving the potential of the developed data-structure, and we have already started working in this direction [41].

A main current limitation of our methodology is that presently it assumes that the entire network will be available in main memory. This also affects the other existing algorithms. It is our intention to further develop our methods and provide support to very large scale networks (such as the twitter interaction network collected by Cha et al. [8], with almost 55 million nodes and 2 billion connections). We have already some preliminary work done on the parallelization of g-tries algorithms [45,46].

Acknowledgements We thank the reviewers for their constructive and helpful suggestions, which helped in improving the quality of this manuscript. Pedro Ribeiro is funded by an FCT Research Grant (SFRH/BPD/81695/2011).

References

1. Adamic, L.A., Glance, N.: The political blogosphere and the 2004 U.S. election: divided they blog. In: 3rd international workshop on Link discovery (LinkKDD), pp. 36–43. ACM, New York, NY, USA (2005)
2. Albert, I., Albert, R.: Conserved network motifs allow protein-protein interaction prediction. Bioinformatics **20**(18), 3346–3352 (2004)
3. Albert, R., Barabasi, A.L.: Statistical mechanics of complex networks. Reviews of Modern Physics **74**(1) (2002)

4. Arenas, A.: Network data sets.
http://deim.urv.cat/~aarenas/data/welcome.htm (2011)
5. Batagelj, V., Mrvar, A.: Pajek datasets.
http://vlado.fmf.uni-lj.si/pub/networks/data/ (2006)
6. Borgelt, C., Berthold, M.R.: Mining molecular fragments: Finding relevant substructures of molecules. In: 2nd IEEE International Conference on Data Mining (ICDM). IEEE Computer Society Press, Washington, DC, USA (2002)
7. Bu, D., Zhao, Y., Cai, L., Xue, H., Zhu, X., Lu, H., Zhang, J., Sun, S., Ling, L., Zhang, N., Li, G., Chen, R.: Topological structure analysis of the protein-protein interaction network in budding yeast. Nucleic Acids Research **31**(9), 2443–2450 (2003)
8. Cha, M., Haddadi, H., Benevenuto, F., Gummadi, K.P.: Measuring User Influence in Twitter: The Million Follower Fallacy. In: 4th International AAAI Conference on Weblogs and Social Media (ICWSM)
9. Chen, J., Hsu, W., Lee, M.L., Ng, S.K.: Nemofinder: dissecting genome-wide protein-protein interactions with meso-scale network motifs. In: 12th ACM SIGKDD international conference on Knowledge Discovery and Data Mining (KDD), pp. 106–115. ACM, New York, NY, USA (2006)
10. Ciriello, G., Guerra, C.: A review on models and algorithms for motif discovery in protein-protein interaction networks. Briefings in Functional Genomics **7**(2), 147–156 (2008)
11. Cook, S.A.: The complexity of theorem-proving procedures. In: 3rd annual ACM symposium on Theory of computing, STOC '71, pp. 151–158. ACM, New York, NY, USA (1971)
12. Duch, J., Arenas, A.: Community detection in complex networks using extremal optimization. Physical Review E (Statistical, Nonlinear, and Soft Matter Physics) **72**, 027,104 (2005)
13. da F. Costa, L., Rodrigues, F.A., Travieso, G., Boas, P.R.V.: Characterization of complex networks: A survey of measurements. Advances In Physics **56**, 167 (2007)
14. Fredkin, E.: Trie memory. Communications of the ACM **3**(9), 490–499 (1960)
15. Grochow, J., Kellis, M.: Network motif discovery using subgraph enumeration and symmetry-breaking. Research in Computational Molecular Biology pp. 92–106 (2007)
16. Howe, D.: Foldoc, free online dictionary of computing. http://foldoc.org/ (2010)
17. Huan, J., Bandyopadhyay, D., Prins, J., Snoeyink, J., Tropsha, A., Wang, W.: Distance-based identification of structure motifs in proteins using constrained frequent subgraph mining. In: IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB), (2006)
18. Huan, J., Wang, W., Prins, J.: Efficient mining of frequent subgraphs in the presence of isomorphism. In: 3rd IEEE International Conference on Data Mining (ICDM), p. 549. IEEE Computer Society Press, Washington, DC, USA (2003)
19. Kärkkäinen, L.: Yet another java vs. c++ shootout.
http://zi.fi/shootout/ (2008)
20. Kashani, Z., Ahrabian, H., Elahi, E., Nowzari-Dalini, A., Ansari, E., Asadi, S., Mohammadi, S., Schreiber, F., Masoudi-Nejad, A.: Kavosh: a new algorithm for finding network motifs. BMC Bioinformatics **10**(1), 318 (2009)
21. Kashtan, N., Itzkovitz, S., Milo, R., Alon, U.: Efficient sampling algorithm for estimating subgraph concentrations and detecting network motifs. Bioinformatics **20**(11), 1746–1758 (2004)
22. Köbler, J., Schöning, U., Torán, J.: The graph isomorphism problem: its structural complexity (Progress in Theoretical Computer Science). Birkhauser Verlag, Basel, Switzerland (1993)
23. Lacroix, V., Fernandes, C.G., Sagot, M.F.: Motif search in graphs: Application to metabolic networks. IEEE/ACM Transactions on Computational Biology and Bioinformatics **3**(4), 360–368 (Oct.-Dec. 2006)
24. Lancichinetti, A., Fortunato, S., Radicchi, F.: Benchmark graphs for testing community detection algorithms. Physical Review E (Statistical, Nonlinear, and Soft Matter Physics) **78**(4) (2008)
25. Lancichinetti, A., Fortunato, S., Radicchi, F.: Benchmark graphs for testing community detection algorithms. Physical Review E (Statistical, Nonlinear, and Soft Matter Physics) **78**(4), 046,110 (2008)

26. Lusseau, D., Schneider, K., Boisseau, O.J., Haase, P., Slooten, E., Dawson, S.M.: The bottlenose dolphin community of doubtful sound features a large proportion of long-lasting associations. can geographic isolation explain this unique trait? Behavioral Ecology and Sociobiology **54**(4), 396–405 (2003)
27. McKay, B.: Practical graph isomorphism. Congressus Numerantium **30**, 45–87 (1981)
28. McKay, B.: Isomorph-free exhaustive generation. Journal of Algorithms **26**(2), 306–324 (1998)
29. Milo, R., Itzkovitz, S., Kashtan, N., Levitt, R., Shen-Orr, S., Ayzenshtat, I., Sheffer, M., Alon, U.: Superfamilies of evolved and designed networks. Science **303**(5663), 1538–1542 (2004)
30. Milo, R., Shen-Orr, S., Itzkovitz, S., Kashtan, N., Chklovskii, D., Alon, U.: Network motifs: simple building blocks of complex networks. Science **298**(5594), 824–827 (2002)
31. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM SIGPLAN Notices **42**, 89–100 (2007)
32. Newman, M.: Network data. `http://www-personal.umich.edu/~mejn/netdata/` (2009)
33. Newman, M.E.J.: The structure and function of complex networks. SIAM Review **45** (2003)
34. Newman, M.E.J.: Finding community structure in networks using the eigenvectors of matrices. Physical Review E (Statistical, Nonlinear, and Soft Matter Physics) **74**(3), 036,104 (2006)
35. Nijssen, S., Kok, J.N.: Frequent graph mining and its application to molecular databases. In: Systems, Man and Cybernetics, 2004 IEEE International Conference on, vol. 5 (2004). DOI 10.1109/ICSMC.2004.1401252. URL http://dx.doi.org/10.1109/ICSMC.2004.1401252
36. Norlen, K., Lucas, G., Gebbie, M., Chuang, J.: EVA: Extraction, Visualization and Analysis of the Telecommunications and Media Ownership Network. In: International Telecommunications Society 14th Biennial Conference (ITS), Seoul Korea,. International Telecommunications Society (2002)
37. Omidi, S., Schreiber, F., Masoudi-Nejad, A.: Moda: An efficient algorithm for network motif discovery in biological networks. Genes & genetic systems **84**(5), 385–395 (2009)
38. Pasquier, N., Bastide, Y., Taouil, R., Lakhal, L.: Discovering frequent closed itemsets for association rules. In: ICDT '99: 7th International Conference on Database Theory, pp. 398–416. Springer, London, UK (1999)
39. Pržulj, N.: Biological network comparison using graphlet degree distribution. Bioinformatics **23**, e177–e183 (2007)
40. Reitz, J.: Odlis: Online dictionary of library and information science. `http://vlado.fmf.uni-lj.si/pub/networks/data/dic/odlis/odlis.pdf` (2002)
41. Ribeiro, P., Silva, F.: Efficient subgraph frequency estimation with g-tries. In: International Workshop on Algorithms in Bioinformatics (WABI), *LNCS*, vol. 6293, pp. 238–249. Springer (2010)
42. Ribeiro, P., Silva, F.: G-tries: an efficient data structure for discovering network motifs. In: 25th ACM Symposium on Applied Computing (SAC), pp. 1559–1566. ACM (2010)
43. Ribeiro, P., Silva, F.: Querying subgraph sets with g-tries. In: 2nd ACM SIGMOD Workshop on Databases and Social Networks, pp. 25–30. ACM (2012). DOI 10.1145/2304536.2304541
44. Ribeiro, P., Silva, F., Kaiser, M.: Strategies for network motifs discovery. In: 5th IEEE International Conference on e-Science, pp. 80–87. IEEE Computer Society Press, Oxford, UK (2009)
45. Ribeiro, P., Silva, F., Lopes, L.: Efficient parallel subgraph counting using g-tries. In: IEEE International Conference on Cluster Computing (Cluster), pp. 1559–1566. IEEE Computer Society Press (2010)
46. Ribeiro, P., Silva, F., Lopes, L.: Parallel discovery of network motifs. Journal of Parallel and Distributed Computing **72**, 144–154 (2012)
47. Schreiber, F., Schwobbermeyer, H.: Towards motif detection in networks: Frequency concepts and flexible search. In: International Workshop on Network Tools and Applications in Biology (NETTAB), pp. 91–102 (2004)
48. Shen-Orr, S.S., Milo, R., Mangan, S., Alon, U.: Network motifs in the transcriptional regulation network of escherichia coli. Nature Genetics **31**(1), 64–68 (2002)

49. Sporns, O., Kotter, R.: Motifs in brain networks. PLoS Biology **2** (2004)
50. Tarjan, R.: Depth-first search and linear graph algorithms. In: Annual IEEE Symposium on Foundations of Computer Science, pp. 114–121. IEEE Computer Society, Los Alamitos, CA, USA, (1971)
51. Valverde, S., Solé, R.V.: Network motifs in computational graphs: A case study in software architecture. Physical Review E (Statistical, Nonlinear, and Soft Matter Physics) **72**(2) (2005)
52. Wang, C., Parthasarathy, S.: Parallel algorithms for mining frequent structural motifs in scientific data. In: ACM International Conference on Supercomputing (ICS) (2004)
53. Watts, D.J., Strogatz, S.H.: Collective dynamics of 'small-world' networks. Nature **393**(6684), 440–442 (1998)
54. Wernicke, S.: A faster algorithm for detecting network motifs. In: International Workshop on Algorithms in Bioinformatics (WABI), *LNCS*, vol. 3692, pp. 165–177. Springer (2005)
55. Wernicke, S.: Efficient detection of network motifs. IEEE/ACM Transactions on Computational Biology and Bioinformatics **3**(4), 347–359 (2006)
56. White, J.G., Southgate, E., Thomson, J.N., Brenner, S.: The Structure of the Nervous System of the Nematode Caenorhabditis elegans. Philosophical Transactions of the Royal Society of London. B, Biological Sciences **314**(1165), 1–340 (1986)
57. Yan, X., Han, J.: gspan: Graph-based substructure pattern mining. In: 2nd IEEE International Conference on Data Mining (ICDM), p. 721. IEEE Computer Society Press, Washington, DC, USA (2002)
58. Yan, X., Yu, P.S., Han, J.: Graph indexing: a frequent structure-based approach. In: Proceedings of the 2004 ACM SIGMOD international conference on Management of data, SIGMOD '04, pp. 335–346. ACM, New York, NY, USA (2004)
59. Yuan, D., Mitra, P.: A lattice-based graph index for subgraph search. In: 14th International Workshop on the Web and Databases (WebDB) (2011)