G-Tries: an efficient data structure for discovering network motifs

Pedro Ribeiro and Fernando Silva
CRACS & INESC-Porto LA
Faculdade de Ciências, Universidade do Porto
R. Campo Alegre, 4169-007 Porto, Portugal
{pribeiro,fds}@dcc.fc.up.pt

ABSTRACT

In this paper we propose a novel specialized data structure that we call `g-trie`, designed to deal with collections of subgraphs. The main conceptual idea is akin to a prefix tree in the sense that we take advantage of common topology by constructing a multiway tree where the descendants of a node share a common substructure. We give algorithms to construct a `g-trie`, to list all stored subgraphs, and to find occurrences on another graph of the subgraphs stored in the `g-trie`. We evaluate the implementation of this structure and its associated algorithms on a set of representative benchmark biological networks in order to find network motifs. To assess the efficiency of our algorithms we compare their performance with other known network motif algorithms also implemented in the same common platform. Our results show that indeed, `g-tries` are a feasible, adequate and very efficient data structure for network motifs discovery, clearly outperforming previous algorithms and data structures.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Graphs and networks, Trees;
G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms*

General Terms

Algorithms, Performance, Experimentation

Keywords

Biological Networks, Complex Networks, Network Motifs, Graph Mining, Algorithms, Data Structures, Tries

1. INTRODUCTION

A wide variety of real-life structures can be represented as complex networks [20]. There are many different measures and concepts one can use to mine interesting and useful quantitative data from these networks [5, 6]. One of these concepts is "network motifs", originated by Milo et al. [18] in 2002. Briefly, a network motif is a recurring subnetwork conjectured to have some significance. In particular, it appears with a higher frequency than it would be expected in similar random networks. This concept has been applied in a multitude of domains, like protein-protein interactions [2], gene transcriptional regulation [7], food webs [13], brain [21] or electronic circuits [11].

Finding network motifs is a computationally hard problem, since we will be trying to match the desired motifs with subgraph patterns. This leads to graph isomorphism, which does not have any polynomial time algorithm known [17]. Therefore, the time needed to discover network motifs grows exponentially as the size of the motifs increases. Presently used methods basically revolve around two main phases: first they compute the frequency of all subgraphs of a determined size that exist in the original graph (doing what is called a "subgraph census"). Then a set of similar random graphs (with the same degree sequence as the original network) is generated and the subgraph census is calculated on each of these, from which we can calculate the statistical significance of each different class of isomorphic subgraphs, by calculating the probability of that particular pattern being over-represented. Since we typically have hundreds of random networks, the main bottleneck of the computation is doing the census on these. There are optimizations (like sampling [12]) that trade accuracy for computation time, but if exact results are needed, we are obliged to use a costly exhaustive census.

Note that finding network motifs is substantially different from the problem of discovering *frequent subgraphs* [10, 14] Although with some similarities, frequent subgraph algorithms are conceptually different (mostly based on the *a-priori* principle [1] by incrementally building subgraphs), since their goal is to find only frequent subgraphs (as opposed to a complete census) that appear at least a determined number of times in a set of graphs (as opposed to a single graph).

Regarding network motifs, current methods do the subgraph census following two very different approaches: either we enumerate all subgraphs of a determined size and then determine which ones are isomorphic [18, 24], or we know in advance which subgraphs we are looking for (for example by generating all possible subgraphs of a determined size) and then we separately compute the individual frequency of each one by running a specialized search that identifies suitable subgraph matchings [9].

These approaches are somehow in extreme corners. In the first one, we need to consider all subgraphs of the graph,

SAC'10 March 22-26, 2010, Sierre, Switzerland.

even when in the case of random graphs we are mainly interested in the frequency of the subgraphs that appeared in the original network. Thus, we may be potentially wasting a lot of time in discovering subgraphs whose frequency do not interest us. In the second approach, we are only considering one subgraph at a time, potentially wasting a lot of time by searching again on the same graph section which had another very similar type of subgraph.

Our main contribution is to provide a data structure that falls conceptually in the middle of these two approaches. Since we know in advance the set of subgraphs that we are interested in (calculated by doing the census in the original subgraph), we store this set of subgraphs on a tree-like data structure that takes advantage of common topologies. We identify and use common ancestor tree nodes to represent common substructures. We then use this tree to efficiently compute the frequency of the subgraphs in the random graphs, by doing searching and matching at the same time the entire set of graphs, while avoiding graph symmetries due to automorphisms. When we have a partial constructed subgraph, we know which subset of graphs are still viable candidates to match that particular subgraph, and when we finish that subgraph, we already know that it is isomorphic to the one we are looking for. We called this novel data structure `g-tries` (from "**G**raph re**TRIE**val"). It significantly compresses the representation of a collection of subgraphs that share common substructures and clearly outperforms previously known methods when computing the frequency of all its subgraphs within other larger graphs. Thus, in our view, it is an excellent representation structure on which one should base efficient solutions to the complete network motif discovery problem.

The remainder of the paper is as follows. We start by introducing some common network terminology (section 2), then we define `g-tries`, and present the main algorithms for its construction and usage (section 3). Next, we study the performance of the `g-tries` implementation using a set of representative real complex networks and compare its efficiency to the previously used methods also implemented in the same common platform (section 4). We finally conclude by summarizing contributions and suggesting some future work (section 5).

2. GRAPH TERMINOLOGY

In order to have a well defined and coherent graph terminology throughout the paper, this section reviews the main concepts used.

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. The *neighborhood* 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. 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 $G_{Adj}[a][b]$ represents a possible edge between vertices with index a and b.

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 for any pair of vertices $(u, v) \in E(G_k)$ 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 $N(u)$ for all $u \in V(G_k)$.

A *mapping* of a graph is a bijection where each vertex is assigned a value. 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 where two vertices of G share an edge if and only if their corresponding vertices in H also share an edge. 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.

3. G-TRIES

Given the space constraints and for the sake of clarity of the explanation, we will exemplify the use of `g-tries` on simple undirected graphs. The same basic ideas being detailed, are easily expandable to more complex cases, like directed, weighted or colored graphs.

3.1 Core idea and definition

When working with sequences, prefix trees (or tries) [8] are a well known concept. This data structure is a tree where all descendants of a node have the same common prefix, as illustrated in Figure 1.

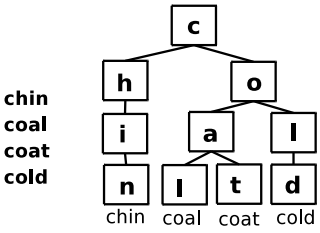

Figure 1: An example trie representing a set of four words. Note how common prefixes are aggregated in the same nodes.

What we propose is to apply a similar concept in graphs, in order to take advantage of possible common substructures of 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. Figure 2 exemplifies this.

Figure 2: These five 6-vertices subgraphs all share a common substructure indicated by the black vertices (a clique of size 3).

In the same way each trie node has a single letter, we will create a tree where each node represents a single graph vertice. Each vertice is characterized by its connections to the respective ancestor nodes. Figure 4 exemplifies this kind of trees.

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.

We will call these kind of trees as `g-tries`, following the etymology "**G**raph re**TRIE**val". We now give an informal

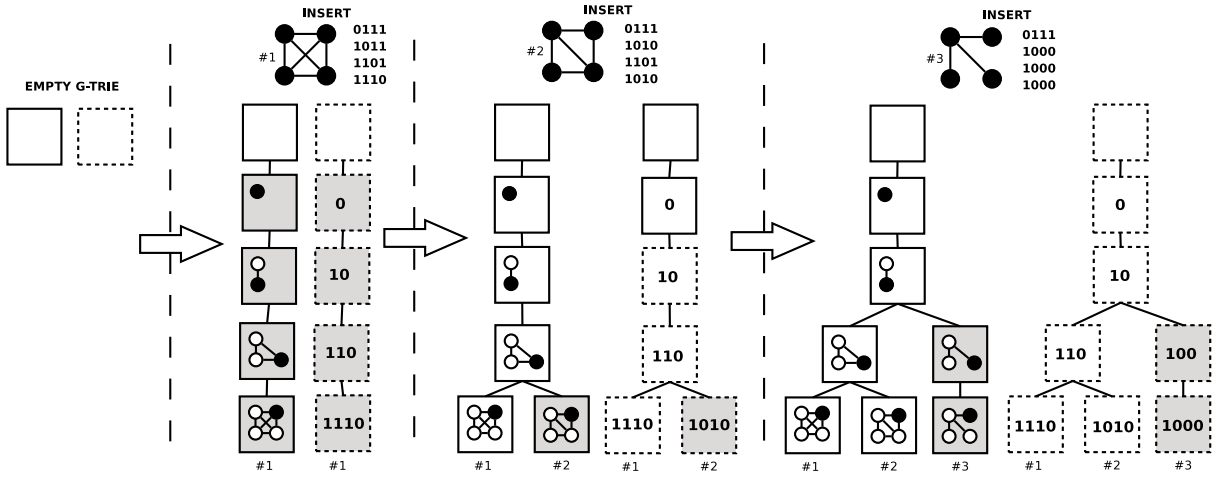

Figure 3: Sequential insertion of 3 graphs on an initially empty `g-trie`. Tree nodes in gray are the new ones after each insertion and nodes in white are the ones that remain from before. Squares with dashed lines represent the actual `g-trie` implementation with adjacency matrices, and squares with normal lines give the correspondent visual representation. Black vertices indicate new vertices, while the old ones are white.

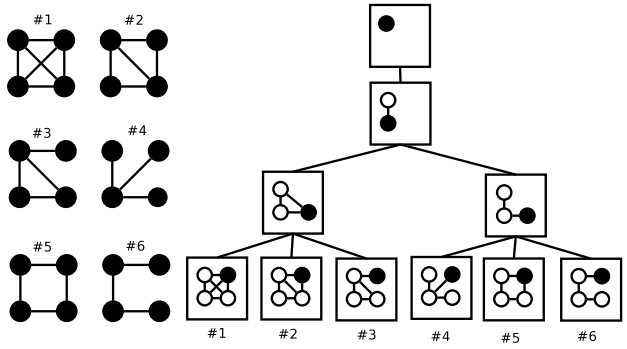

Figure 4: A tree representing a set of 6 graphs. Each tree node adds a new vertex (in black) to the already existing ones in the ancestor nodes (white vertices)

definition of this abstract data structure. Note that a multiway tree is one with any number of children for each node.

Definition 1 (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 and its corresponding edges to ancestor nodes. A path from the root to a leaf corresponds to one single graph. Descendants of a* `g-trie` *node share a common subgraph.*

3.2 Constructing G-Tries

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

As said before, each node needs to store information about a new vertex and its connections to ancestor vertices. Given its simplicity and ease of use, we will represent the graph by an adjacency matrix with `'1'` representing a connection and `'0'` its nonexistence. A vertex and its edges is therefore represented by the respective row of the matrix. Since every tree node represents a single new vertex, we can therefore just store in it the corresponding row. More than that, since we only need to know the connections to ancestor vertices, we can just store the values of the row up to the value defining if there is a connection to itself. Note that the graph to be stored is undirected and therefore the matrix is symmetrical, and so in fact we only need to store half of the matrix.

To construct a `g-trie`, we just repeatedly insert one subgraph at a time, starting with an empty tree (just a root node). Each time, we traverse the tree and verify if any of the children has the same partial adjacency matrix row as the graph we are inserting. With each increase in depth we also increase the index of the vertex we are considering. Figure 3 exemplifies this process.

There are however many different possible adjacency matrices representing the same class of isomorphic graphs, as exemplified in figure 5.

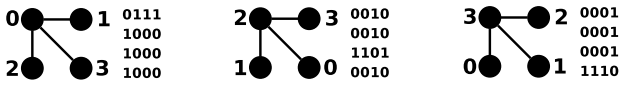

Figure 5: Three different adjacency matrices representing the same graph. Note how the labelling of the vertices affects the correspondent matrix.

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 cope with that we use a canonical labelling, which guarantees that isomorphic graphs will always produce the same adjacency matrix, and therefore the same set of subgraphs is guaranteed to produce the same `g-trie`.

There are many possible canonical representations. We choose to use the labelling that produces the lexicographically larger matrix, ensuring that the vertices with the highest number of connections will appear first in the matrix. This favours the occurrence of possible common substructures, specially on the lower index vertices. For example, with canonical labelling, all graphs without self-loops will produce the same partial row on the first vertex (just a `'0'` indicating no connection to itself), the same partial row on

the second vertex (just '10' - the first vertex needs to be connected to some other vertex and this labelling will ensure that there is always a connection to the second vertex) and so on. Figure 3 illustrates this effect in practice.

Having many shared substructures on the lower index vertices is highly desirable, since this will produce `g-tries` with less total number of nodes and small number of nodes in the lower depths. As 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 info stored on 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 the actual node values, and equation (1) is a good indicator of how much space we save and how much common substructure we identified.

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

As an example, the `g-trie` constructed in figure 3 has a compression ratio of $41.67\% = 1-7/(4+4+4)$ (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.

Algorithm 1 details a method to insert a graph in a `g-trie`. As said, constructing a complete `g-trie` from a set of subgraphs can be done by inserting them one by one into an initially empty tree.

Algorithm 1 Inserting a graph G in a `g-trie` T

```
1: procedure INSERT(G, T)
2:     M := canonicalAdjMatrix(G)
3:     insertRecursive(M, T, 0)
4: procedure INSERTRECURSIVE(M, T, k)
5:     if k < numberRows(M) then
6:         for all children c of T do
7:             if c.value = first k+1 values of M[k] then
8:                 insertRecursive(M, c, k+1)
9:                 return
10:        nc := new g-trie node
11:        nc.value := first k+1 values of M[k];
12:        T.insertChild(nc)
13:        insertRecursive(M, nc, k+1);
```

Explaining in more detail, we start by obtaining a canonical adjacency matrix of the graph being inserted (line 2). In our case, as explained before, we opted for the lexicographically bigger matrix. Then we recursively traverse the tree, inserting new nodes when necessary, with the procedure `insertRecursive()`. This is done by going trough all possible children of the current node (line 6) and checking if their stored value is equal to the correspondent part of the adjacency matrix (line 7). If it is, we just continue recursively with the next vertex (line 8). If not, we create a new child (lines 10 to 12) and continue as before (line 13). When there are no more vertices to process, we stop the recursion (line 5).

Regarding the complexity of the algorithm, `insertRecursive()` takes $O(|V(G)|^2)$, the size of the adjacency matrix. Besides this, the whole insertion needs to calculate a canonical labelling of the graph. This can be done using whatever already existing method for canonical labelling, like the highly efficient third-party algorithm *nauty* [17]), or using our own algorithm that guarantees certain properties (like the lexicographically biggest adjacency matrix).

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 leaf at depth k represents a $k-$graph.

3.3 Census of a set of subgraphs

Algorithm 2 details a method for finding and counting all occurrences of the `g-trie` collection of graphs as induced subgraphs of another larger graph. The main idea is to backtrack through all possible subgraphs, but at the same time do the isomorphism tests as we are constructing the candidate subgraphs. Moreover, taking 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).

Algorithm 2 Census of subgraphs of T in graph G

```
1: procedure CENSUS(G, t)
2:     for all children c of T.root do
3:         match(c, G, 0, ∅)
4: procedure MATCH(T, G, k, V_used)
5:     if V_used = ∅ then
6:         V_cand := V(G)
7:     else
8:         V_conn = {V_used[i]: T.value[i]='1'}
9:         m := m∈V_conn : ∀v∈V_conn, |N(m)| ≤ |N(v)|
10:        V_cand := {v ∈ N(m) : v ∉ V_used}
11:    for all v ∈ V_cand do
12:        add v to end of V_used
13:        if ∀i∈[0..k]: T.value[i] = G_Adj[v][V_used[i]] then
14:            if T.isLeaf() then
15:                reportGraph();
16:            else
17:                for all children c of T do
18:                    match(c, G, k+1, V_used)
19:        remove v from V_used
```

We start by iterating trough all children of the root (line 2), calling the recursive backtracking procedure `match()` (line 3). This procedure starts by constructing the set V_{cand} of candidate nodes to expand the V_{used} partially constructed subgraph. If we still do not have any vertices, all nodes of the graphs are suitable candidates for being the first vertex (lines 5 and 6). If not, we first compute the V_{conn} vertices to which the current vertex should be connected (line 8). We then choose the vertex of this set which has the smaller neighborhood (line 9), in order to minimize potential candidates. We proceed by adding to the candidate list the vertices of that minimal neighborhood which are still not in the partially construct subgraph (line 10). After all of this we traverse the possible candidates (line 11) and start by adding each one to the partial graph (line 12), do some

tests (lines 13 to 18) and then remove the vertex from V_{used} in order to continue with the next candidate. To check if a candidate is a plausible match for this graph position, we start by looking at the `g-trie` node value and seeing if the candidate provides a complete match do the desired connections with the ancestor vertex of V_{used} (line 13). If that is the case, we have two options. If we are already in a leaf, we completed a graph matching and we can do whatever we want, like increasing its frequency (lines 14 and 15). If we are not in a leaf, we recursively try to continue the matching with all the children of the current tree node (line 18).

Figure 6 exemplifies the flow of the previously described procedure. Note how the list of candidates is constructed from the ancestor with the minimal neighborhood and how candidates are rejected when they do not have the needed connections to ancestor vertices.

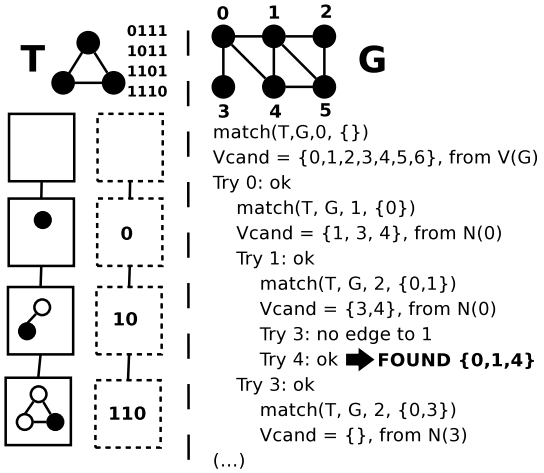

Figure 6: An example partial program flow of the census procedure, when searching for a 3-clique on graph with 6 vertices.

3.4 Breaking Symmetries

One main problem with the census 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 6, we would not only find $\{0, 1, 4\}$ but also $\{0, 4, 1\}$, $\{1, 0, 4\}$, $\{1, 4, 0\}$, $\{4, 0, 1\}$ and $\{4, 1, 0\}$. In the end we can divide by the number of automorphisms to obtain the real frequency, but a lot of valuable computation time is wasted.

We need to avoid this kind of redundant computations and find each subgraph only once. In order to do that we will generate a set of symmetry breaking conditions for each subgraph, similarly to what was done by Grochow and Kellis [9]. 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. Figure 7 shows an example of a graph and conditions of the type we described that break the symmetry.

Although our inspiration was Grochow and Kellis [9], we use a slightly different method for generating the conditions, detailed in algorithm 3.

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 mapping.

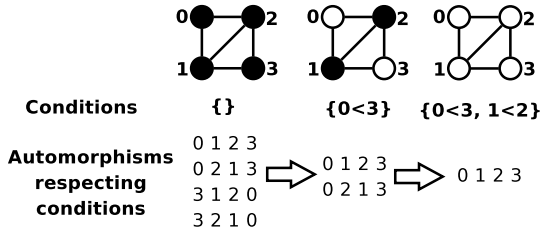

Figure 7: Symmetry breaking conditions for an example graph with 4 vertices. The conditions fix the position of the vertices indicated in white. Vertices in black have at least another equivalent vertex.

Algorithm 3 Symmetry breaking conditions for graph G

```
1: function findConditions(G)
2:     Conditions := ∅
3:     Aut := setAutomorphisms(G)
4:     while |Aut| > 1 do
5:         m := minimum v : ∃ map ∈ Aut, map[v] ≠ v
6:         for all v ≠ m : ∃ map ∈ Aut, map[m] = v do
7:             add m < v to Conditions
8:         Aut := {map ∈ Aut : map[m] = m}
9:     return Conditions
```

Note that although calculating automorphisms is thought to be computationally expensive, in practice we found it to be almost instantaneous for the subgraph sizes used and with *nauty* [17] we were able to test much bigger subgraphs (with hundreds of nodes) and obtain their respective automorphisms very quickly, in less that 1 second. What remains is that this calculation is very far from being a bottleneck in the whole process of generating and using g-tries. Each time we start by finding the minimum index m where there is still a mapping in *Aut* that allows for a vertex with index different than m in position m, that is, we calculate the minimum vertex that still has at least another equivalent node (line 5). We then add conditions stating that the vertex in position m should have an index lower than every other equivalent position (lines 6 and 7), which in fact fixes m in its position. We then reduce *Aut* removing the mappings that do not respect the newly added connections, that is, the ones that do not fix m. We continue repeating this process until there is only the identity left Aut' (line 4). We finally return all the generated conditions (line 9). Note that in the case of the graph of figure 7, this algorithm would create the exact same set of conditions as depicted there.

We will now show how we can expand the insertion and census algorithms in order to use symmetry breaking conditions. The basic idea is that each `g-trie` node should know which set of graphs (and respective conditions) are in their descendant leafs. With this, the matching algorithm of the census can see if the partial subgraph constructed is respecting the conditions of at least one possible end leaf, that is, there is at least one possible subgraph that can still be constructed by expanding the partial subgraph. Algorithms 4 and 5 detail how this is done. Note how almost everything is the same as in the previously described algorithms.

With these two algorithms, 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 we always create conditions of the minimal

Table 1: Number of possible different undirected and connected subgraphs with k vertices.

k	3	4	5	6	7	8	9	10	11	12	13
Subgraphs	2	6	21	112	853	11117	261080	11716571	1006700565	164059830476	50335907869219

Algorithm 4 Inserting a graph G in a `g-trie` T [symmetry breaking condition version]

1: **procedure** INSERTCOND(G, T)
2: $M :=$ canonicalAdjMatrix(G)
3: insertCondRecursive($M, T, 0, C$)
4: **procedure** INSERTCONDRECURSIVE(M, T, k, C)
5: lines 5 to 11 of algorithm 1
6: nc.addSetConditions(C)
7: lines 12 to 13 of algorithm 1

Algorithm 5 Census of subgraphs of T in graph G [symmetry breaking condition version of `match()`]

1: **procedure** MATCH(T, G, k, V_{used})
2: lines 5 to 13 of algorithm 2
3: **if** $\exists C \in T$.conditions: V_{used} respects C **then**
4: lines 14 to 18 of algorithm 2
5: line 19 of algorithm 2

indexes still not fixed (line 5 of algorithm 3), we tend to discover early in the recursion that a condition is being broken, therefore cutting branches of the possible search tree as soon as possible.

3.5 Application in network motifs discovery

The main flow of all exact network motifs algorithms it 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 calculating 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 [18]) is just a very small fraction of the time their census takes. Computing the census on all the random networks is therefore the main bottleneck of the whole process (there can be hundreds of random graphs) and `g-tries` can help precisely in this phase.

In order to do that we must start by doing the census of all the k-subgraphs in the original network by using whatever method we have available. The best general exhaustive network-centric algorithm available we are aware of is the one implemented in `FanMod` [24], which is orders of magnitude faster than the original algorithm of `mfinder` [18]. Note that although FanMod is able to use sampling, it is also capable to do an exhaustive and complete census. The alternative is to generate the list of all possible subgraphs of a determined size and then use a graph-centric algorithm to calculate the frequency of all of them, one at a time. This is what `Grochow` [9] proposes in this phase (by doing queries for single subgraphs), and `g-tries` can also be used with the same intent: we can generate all possible k-subgraphs, insert them all in a `g-trie` and then calculate their census in the original network with the algorithms given in section 3.3. This approach has the problem of potentially spending time searching for subgraphs that end up not existing in the network, but for small k is feasible. As k grows, it becomes completely unfeasible because the number of possible k-subgraphs grows exponentially, as we can see in table 1.

After having the census of all the k-subgraphs in the original network, we insert all of them in a `g-trie`. Then we can do the most computationally costly part of the network motif discovery process, generating the ensemble of random networks and using the created `g-trie` to discover their frequency on all of the random networks. When compared with the network-centric alternative of doing a census of all k-subgraphs in all the random networks (`FanMod`), we can potentially save time because we will just look into the k-subgraphs that matter. Note that in some cases one may also be interested in *anti-motifs*, which as the name suggests are patterns underrepresented. In this case the complete census performed by the network-centric approach would be useful since we may need to discover subgraphs not present in the original graph. The other alternative is to use `Grochow` and query individually all important subgraphs in all the random networks. We can potentially save time with `g-tries` when comparing with this approach, since we will be traversing each region of the graph only once, because we will be matching at the same time several different classes of isomorphic subgraphs.

4. RESULTS

In order to evaluate the performance of our proposed data structure and assess the efficiency of the described algorithms, we implemented `g-tries` using C++. In order to compare to previous algorithms, we also implemented the best known previous strategies using the network-centric approach (`FanMod` [24]) and subgraph-centric approach (`Grochow` [9]) in the same common C++ platform, enabling us to be sure that differences in execution times are due to the algorithms themselves and not due to the programming language or graph data structures used (note that these two last approaches also break symmetries, such as `g-tries` do). Isomorphisms and canonical labellings were computed during runtime, with the aid of `nauty` tool [17]. All tests were made on a computer with an Intel Core 2 6600 (2.4GHz) with 2GB of memory. Results obtained were double checked with the publicly available `FanMod` tool [25], ensuring that the frequency counts found were correct.

We used a set of complex networks from different domains, with different number of vertices and edges. Table 2 summarizes the five networks used. For simplification, self-loops (connections from a node to itself) were removed when they existed, and all weight in edges was ignored. `Circuit` was initially a directed network that we converted to an undirected one by considering that all edges were bi-directional.

Evaluation started with a census of the original network using `FanMod` (exhaustive census), `Grochow` (query all k-subgraphs) and `G-Tries` (apply algorithm 2 with a `g-trie` including all k-subgraphs), and registering the processor time spent. We then registered the number of different classes of isomorphism found, noting the compression achieved when we inserted all of them in a `g-trie`, defined by equation (1), as well as the time spent creating the respective `g-trie`. After that, we generated a random ensemble of five different

Table 2: The set of five different networks used to test the algorithms.

Network	Vertices	Edges	Edges/Vertices	Description	Source
dolphins	62	159	2.56	Frequent associations between a group of dolphins	[16, 19]
circuit	252	399	1.58	Electronic circuit	[18]
social	1000	7770	7.77	Benchmark social network with heterogeneous communities	[15]
yeast	2361	6646	2.81	Protein-protein interaction network in budding yeast	[4, 3]
power	4941	6594	1.33	U.S.A. western states power grid	[23, 19]

Table 3: Comparison of `g-trie` data structure with previous algorithms. Execution time is measured in seconds and is relative to processor time. vs indicates the speedup ratio of `g-tries` against other algorithms on the average time spent doing a census on a random network.

Network	K	G-Trie Subgraphs Original Network			Census Original Network			Average Census on Similar Random Networks				
		Subgraphs	Compression	Time	FanMod	Grochow	G-Trie	FanMod	Grochow	G-Trie	vs FanMod	vs Grochow
dolphins	5	21	71.43%	0.00	0.07	0.03	0.01	0.13	0.04	0.01	16.00x	4.75x
	6	101	78.55%	0.00	0.48	0.28	0.04	1.14	0.35	0.07	17.27x	5.24x
	7	633	82.83%	0.01	3.02	3.44	0.23	8.34	3.55	0.46	18.21x	7.74x
	8	4940	85.53%	0.04	19.44	73.16	1.69	67.94	37.31	4.03	16.87x	9.27x
	9	39963	87.28%	1.87	100.86	2984.22	6.98	493.98	366.79	24.84	19.88x	14.76x
circuit	6	33	74.24%	0.00	0.49	0.41	0.03	0.55	0.24	0.03	19.57x	8.43x
	7	89	78.01%	0.00	3.28	3.73	0.22	3.53	1.34	0.17	20.55x	7.81x
	8	293	81.87%	0.00	17.78	48.00	1.52	21.42	7.91	1.06	20.17x	7.45x
social	3	2	33.33%	0.00	0.31	0.11	0.02	0.35	0.11	0.02	14.67x	4.75x
	4	6	58.33%	0.00	7.78	1.37	0.56	13.27	1.86	0.57	23.28x	3.26x
	5	21	71.43%	0.00	208.30	31.85	14.88	531.65	62.66	22.11	24.05x	2.83x
yeast	3	2	33.33%	0.00	0.47	0.33	0.02	0.57	0.35	0.02	31.67x	19.33x
	4	6	58.33%	0.00	10.07	2.04	0.36	12.90	2.25	0.41	31.15x	5.44x
	5	21	71.43%	0.00	268.51	34.10	12.73	400.13	47.16	14.98	26.70x	3.15x
power	3	2	33.33%	0.00	0.51	1.46	0.00	0.91	1.37	0.01	113.25x	171.00x
	4	6	58.33%	0.00	1.38	4.34	0.02	3.01	4.40	0.03	107.43x	157.07x
	5	21	71.43%	0.00	4.68	16.95	0.10	12.38	17.54	0.14	91.06x	128.96x
	6	101	78.55%	0.00	20.36	95.58	0.55	67.65	92.74	0.88	76.52x	104.90x
	7	626	82.82%	0.01	101.04	765.91	3.36	408.15	630.65	5.17	78.92x	121.94x

similar random networks, by applying a random Markov-chain process like in [18] (with 3 swaps per edge), and we calculated their census applying `FanMod` (exhaustive census), `Grochow` (query the k-subgraphs of the original network) and `G-Tries` (apply algorithm 2 with the `g-trie` of the k-subgraphs in the original network), again as described in section 3.5. This last phase is the real bottleneck of a complete motifs detection algorithm (we will have to calculate it potentially for hundreds of random networks) and therefore we took note of not only the processor time spent, but the speedup obtained with `g-tries` relative to the other algorithms. We experimented with different sizes k of motifs, and in order to limit the number of possible k we only registered the experiments in which `FanMod` average time for a random network was greater than $0s$ and less than $1000s$.

Note that the time needed to generate all possible undirected k-subgraphs is quite small (`nauty` generates all undirected graphs of size $k < 10$ in less than $0.5s$). The time needed to create the respective `g-trie` with all the possible k-subgraphs (used for the census in the original network) is also very small and all of this can even be precalculated for all possible small k sizes and reused on all graphs being analyzed. If k is too big, we would still need to resort to `FanMod` for the census on the original network, as said in section 3.5.

Table 3 details the results obtained. The first and most relevant fact to notice is that in all cases `g-tries` outperform the previous algorithms. The comparison with `FanMod` is overwhelmingly positive, with `g-tries` being orders of magnitude faster. Against `Grochow` the difference is not so accentuated, but even so `g-tries` always produces a better behaviour. We can also observe that `g-tries` really excel on less dense graphs, presenting the better speedup ratios on networks with a small number of average edges per vertice. The network `power` is the most extreme example of this.

One other important aspect is that the compression rates of the trees are always significant and grow with the size of k, since there are more graphs and more common parts (for $k > 5$, the compression rate is always greater than 70%). However, this does not have a direct influence on the time needed by the `g-tries` census, and the speedup versus `Grochow` seem to decrease as k increases (with the exception of `dolphins`). This is probably caused by the fact that the `g-tries` induce a specific order on the way the nodes of a subgraph are found (since it is the same order that allows us to create common parts), while `Grochow` does not, and therefore `g-tries` spend some more time looking for the right order of the nodes of subgraphs, and this happens more frequently as k increases.

Regarding the census in the original network, since the graphs are undirected and k is always relatively small, doing the census by first generating all subgraphs and then searching all those subgraphs with a `g-trie` is not only feasible, but also much faster than doing a census with `FanMod`. However, it should be noticed that as k increases, the time needed with a graph-centric approach can grow exponentially, since we need to compute the frequency of all subgraphs of that size (see table 1), even when the frequency for the majority of those subgraphs is zero. This effect can be seen with `Grochow` when $k = 9$ (261080 subgraphs) in `dolphins`, with much more time needed than with the network-centric `FanMod`, meaning that more time is being spent searching for subgraphs that do not appear at all in the original network.

5. CONCLUSION

In this paper we introduced `g-tries`, a novel data structure specialized in efficiently representing collections of subgraphs. `G-Tries` are multiway trees that take advantage of common substructures in the subgraphs. We gave algorithms for inserting subgraphs, discovering their frequency on another larger graph and on how to break subgraph symmetries that could lead to redundant computations. We have also shown how `g-tries` could be applied in the network motifs discovery problem.

The results obtained are very promising. `G-Tries` can significantly compress the representation of a set of subgraphs and the comparison with previously existent algorithms shows that `g-tries` clearly outperforms those other methods when computing subgraph census. The execution times can be orders of magnitude better and in particular with small motif sizes and not too dense networks the advantage in using `g-tries` is overwhelming.

This data structure has also the potential to be used in different scenarios. We could use it to just do the initial network census, useful for example on the social network analysis realm (where triads census are very common [22]). Or, we could use it to discover larger motifs, by specifying beforehand larger subgraphs and then do the search for all of them at the same time on a network.

In the near future, we plan to extend `g-tries` to more complicated networks (for example, directed and colored networks), and study its behaviour on these networks. In order to do that, we only have to change the g-trie node content to reflect the increased graph complexity. We also intend to research on how they could be used to obtain approximated motifs results, such as it happens when subgraph sampling is used in `mfinder` [18] or `FanMod` [24]. Finally, we intend to parallelize the algorithms of `g-tries`, allowing for its optimized and large scale use.

6. ACKNOWLEDGMENTS

Pedro Ribeiro is funded by an FCT Research Grant (SFRH/BD/19753/2004).

7. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pages 487–499, 1994.

[2] I. Albert and R. Albert. Conserved network motifs allow protein-protein interaction prediction. *Bioinformatics*, 20(18):3346–3352, December 2004.

[3] V. Batagelj and A. Mrvar. Pajek datasets, 2006. `http://vlado.fmf.uni-lj.si/pub/networks/data/`.

[4] D. Bu, Y. Zhao, L. Cai, H. Xue, X. Zhu, H. Lu, J. Zhang, S. Sun, L. Ling, N. Zhang, G. Li, and R. Chen. Topological structure analysis of the protein-protein interaction network in budding yeast. *Nucl. Acids Res.*, 31(9):2443–2450, May 2003.

[5] D. Chakrabarti and C. Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM Computing Surveys*, 38:1, 2006.

[6] L. da F. Costa, F. A. Rodrigues, G. Travieso, and P. R. V. Boas. Characterization of complex networks: A survey of measurements. *Advances In Physics*, 56:167, 2007.

[7] R. Dobrin, Q. K. Beg, A. Barabasi, and Z. Oltvai. Aggregation of topological motifs in the escherichia coli transcriptional regulatory network. *BMC Bioinformatics*, 5:10, 2004.

[8] E. Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, September 1960.

[9] J. Grochow and M. Kellis. Network motif discovery using subgraph enumeration and symmetry-breaking. pages 92–106. 2007.

[10] J. Han and M. Kamber. *Data Mining, Second Edition: Concepts and Techniques*. Morgan Kaufmann, September 2006.

[11] S. Itzkovitz, R. Levitt, N. Kashtan, R. Milo, M. Itzkovitz, and U. Alon. Coarse-graining and self-dissimilarity of complex networks. *Phys Rev E Stat Nonlin Soft Matter Phys*, 71(1 Pt 2), January 2005.

[12] N. Kashtan, S. Itzkovitz, R. Milo, and U. Alon. Efficient sampling algorithm for estimating subgraph concentrations and detecting network motifs. *Bioinformatics*, 20(11):1746–1758, 2004.

[13] M. Kondoh. Building trophic modules into a persistent food web. *Proceedings of the National Academy of Sciences*, 105(43):16631–16635, 2008.

[14] M. Kuramochi and G. Karypis. Frequent subgraph discovery. *Data Mining, IEEE International Conference on*, 0:313, 2001.

[15] A. Lancichinetti, S. Fortunato, and F. Radicchi. Benchmark graphs for testing community detection algorithms. *Physical Review E (Statistical, Nonlinear, and Soft Matter Ph ysics)*, 78(4), 2008.

[16] D. Lusseau, K. Schneider, O. J. Boisseau, P. Haase, E. Slooten, and S. M. Dawson. 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.

[17] B. McKay. Practical graph isomorphism. *Cong. Numerantium*, 30:45–87, 1981.

[18] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, October 2002.

[19] M. Newman. Network data, september, 2009. `http://www-personal.umich.edu/~mejn/netdata/`.

[20] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45, 2003.

[21] O. Sporns and R. Kotter. Motifs in brain networks. *PLoS Biology*, 2, 2004.

[22] S. Wasserman, K. Faust, and D. Iacobucci. *Social Network Analysis : Methods and Applications (Structural Analysis in the Social Sciences)*. Cambridge University Press, November 1994.

[23] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, June 1998.

[24] S. Wernicke. Efficient detection of network motifs. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 3(4):347–359, 2006.

[25] S. Wernicke and F. Rasche. Fanmod: a tool for fast network motif detection. *Bioinformatics*, 22(9):1152–1153, May 2006.