Large Scale Graph Representations for Subgraph Census

Pedro Paredes and Pedro Ribeiro

CRACS & INESC-TEC
DCC-FCUP, Universidade do Porto, Portugal
pparedes@dcc.fc.up.pt, pribeiro@dcc.fc.up.pt

Abstract. A Subgraph Census (determining the frequency of smaller subgraphs in a network) is an important computational task at the heart of several graph mining algorithms. Recently, several efficient algorithms have been described. We focus on the g-tries, a data structure that encapsulates the topology of the smaller subgraphs in order to speed up the overall computation. Its algorithm makes extensive use of the graph primitive that checks if a certain edge exists. The original implementation used adjacency matrices in order to make this operation as fast as possible, as is the case with most past approaches. This representation is however very expensive in memory usage, obviously limiting the scale of the networks being analyzed. In this paper we study a number of possible approaches that scale linearly with the number of edges. We make an extensive empirical study of these alternatives in order to find an efficient hybrid approach that combines the best representations. We achieve a performance that is less than 50% slower than the adjacency matrix on average (almost 3 times more efficient than a naive binary search implementation), while being memory efficient and tunable for different memory restrictions.

Keywords: Complex Networks, Large Scale Graphs, Graph Mining, Subgraphs, G-Tries

1 Introduction

The use of complex networks to model real-life systems and problems has been more than established in the past few years. To characterize and compare these networks, numerous metrics have been proposed and studied. One important example are network motifs [16]. These are over-represented substructures of a network, that is, subgraphs that appear in a higher number than expected in random networks with similar topological traits. Network motif analysis has been successfully applied in several domains and problems, to name a few, on biological systems, such as brain networks [29], protein-protein interactions [1] or even gene regulation [7], on social networks [11] and more.

To perform a network motif analysis, one needs to compute one or potentially more subgraph census. A subgraph census is an operation that finds the frequencies of all or a subset of subgraphs of a network. This is a computationally hard task since it is related to the subgraph isomorphism problem, a known NP-Complete problem [6]. Furthermore, this is the main bottleneck in the calculation of network motifs and thus is the problem we address in this paper.

Previously proposed approaches range in the way they tackle the problem, in either a *network-centric* fashion [20,13], meaning they enumerate all existing subgraphs and classify them, a *subgraph-centric* fashion [10], where a single subgraph frequency is computed and finally a *set-centric* [24], where the frequency of a set of subgraphs is computed.

In this paper we address a more specific question, namely large scale representations of networks that can be quickly queried for information by subgraph census algorithms. By large scale representations we mean data structures that have a memory complexity less than the square of the number of nodes (forbidding the use of adjacency matrices), potentially proportional to the number of edges (since most real data sets are sparse networks). The current works either only briefly discuss this issue or gloss over it. Also, the current established implementations either use adjacency matrices as the base representation or simple adjacency lists.

We will expand on a previous work of ours, specifically the g-tries [24], a state-of-art set-centric data structure. We start by discussing the problem and its details, as well as the g-tries approach. We then enumerate the graph representation primitives that the algorithm requires and then pinpoint the bottleneck one. Based on that, we discuss several approaches to the problem as well as possible optimizations. Finally, we test and compare the different approaches and discuss our discoveries with the goal of obtaining the best representation.

The remainder of the paper is organized as follows. In Section 2 we start by introducing some terminology, discuss the problem the paper addresses and also briefly go through some of the existing techniques that tackle the problem. Section 3 describes the g-trie data structure and its subgraph counting algorithm. Section 4 starts by describing and justifying our definition of large scale in this context, we then pinpoint the bottleneck primitives of the representation, followed by presenting several possible alternatives and concludes with a discussion of different optimizations. We follow this with Section 5 by presenting the detailed experimental analysis. Finally, we close with Section 6, where we present some concluding remarks.

2 Preliminaries

2.1 Graph terminology

To be consistent in the used terminology, we will go over the notation used. A *graph* G is composed by the set of vertices $V(G)$ (abbreviated, when there is no ambiguity, to simply V) and the set of edges $E(G)$ (abbreviated to E), represented by pairs $(a, b) : a, b \in V(G)$. The *size* of a graph, denoted by $|V(G)|$, is the number of vertices in the graph. A k-graph has size k. All vertices are assigned consecutive integers starting from 0.

A *subgraph* G_k of a graph G is a k-graph where $V(G_k) \subseteq V(G)$ and $E(G_k) \subseteq E(G)$. This subgraph is said to be *induced* when $\forall u, v \in V(G_k) : (v, u) \in E(G) \leftrightarrow (v, u) \in E(G_k)$ and is said to be *connected* when all pairs of vertices have a sequence of edges connecting them.

Two graphs G and H are *isomorphic*, denoted as $G \sim H$, if there is a bijection between $V(G)$ and $V(H)$ such that two vertices are adjacent in G if and only if their corresponding vertices in H are adjacent.

2.2 Problem definition

The base problem we are addressing in this paper is the Subgraph Census Problem (also known as Subgraph Counting Problem). Here we define it precisely.

Definition 1 *Given an integer k and a graph G, determine the frequency of a set S of connected induced k-subgraphs of G. Two occurrences of a subgraph are considered different if they have at least one node that they do not share.*

It should be noted that we are focusing on this particular frequency concept because it is a widely studied one and the one our base algorithm uses. There are, however, other studied frequency concepts, but this particular one is predominant in the literature since it is related to the standard definition for the network motif discovery problem [25].

Furthermore, our goal is to find an efficient scalable graph representation that is applicable to large scale networks, in order to increase the applicability of Subgraph Census algorithms to larger networks. We note that it is important that the representation is efficient in the context of Subgraph Census algorithms, which means that we are not concerned with the complexity or efficiency of any one operation in a particular graph, but the full weight it induces on the subgraph census algorithm execution.

Finally, another important aspect to note is that our representation is static, besides some pre computing, it is not necessary to account for insertion or removal of edges or vertices.

2.3 Current work on Subgraph Census

As far as we know, there are no current works on large scale representations for subgraph census algorithms. Some papers describing established approaches briefly mention this issue, but none goes into detail or performs any studies on different representations. Thus in this subsection we present the previous works on the area and justify our choice of base algorithm.

For network-centric approaches, the two more efficient algorithms for exact computations are `FaSE` [20] and `QuateXelero` [13]. These are two similar contemporaneous algorithms that use as a base enumeration previously established algorithms like `ESU` [30] and `Kavosh` [12], but try to avoid performing an isomorphism test per found subgraph, like the latter two did. Instead, `FaSE` uses a modified g-trie to store intermediate classes of subgraphs as it performs the enumeration, while QuateXelero uses a kind of quadtree to achieve the same.

The most prominent subgraph-centric algorithm is the work by Grochow and Kellis [10], that uses symmetry-breaking conditions to prune the enumeration tree and only find the relevant subgraphs. Building on this work is the set-centric approach of g-tries [24], the work how focus on.

There have been other research directions like approximation algorithms [22,21] and parallel approaches [26,27,18] that increase the applicability of subgraph census algorithms (and increase the need of efficient large scale representations), albeit we focus on sequential exact algorithms.

3 G-trie Based Subgraph Census

3.1 The g-trie data structure

The g-trie data structure is an application of the concept of prefix-trees to graphs. By identifying common topologies and substructures, a g-trie represents a set of graphs, much like a classic string prefix-tree does with string prefixes. It is a multiway tree where each node represents a single subgraph and each descendant node represents a node that shares a common topology with its parent. Thus each node stores information about a single vertex and its connections to the vertices stored in ancestor nodes, resulting in a path from root to each node defining a single subgraph. In Figure 1 we show an example of a g-trie with all undirected 4-subgraphs. Note that this concept can be generalized to directed subgraphs or even colored subgraphs [23].

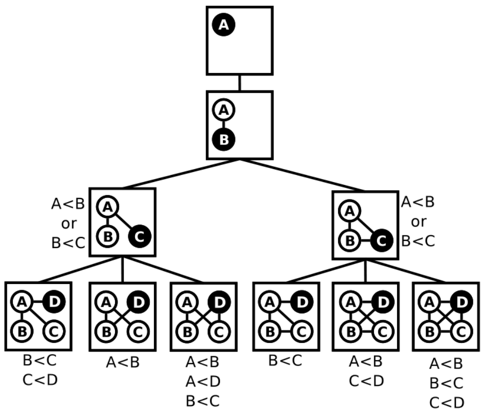

Fig. 1. A g-trie representing all undirected 4-subgraphs

The goal of the g-trie data structure is to efficiently compress a set of subgraphs in order to guide the enumeration to only consider the given subgraphs. To do so, a customized canonical form is employed, with the aim of reducing the number of nodes in the g-trie by using the most common topologies possible.

This is important since the least number of nodes the least number of paths to explore and wasted computation time.

Another important issue is the symmetries exhibited by the subgraphs, caused by automorphisms, which could lead to redundant paths and repeated occurrence finding. To solve this problem, symmetry breaking conditions of the form $X < Y$ (where X and Y are labels of two vertices) are inserted in order to only consider each symmetry once. Due to space constraints, we direct the reader to works like [24] that further explain these topics.

3.2 Subgraph counting with g-tries

In order to avoid ambiguities in the description, from now on we will use the term *node* to refer to the g-trie tree nodes, and *vertex* to refer to node in the graphs. Algorithm 1 details the how to use an already built g-trie to count subgraphs

Algorithm 1 The g-trie subgraph counting algorithm

Input: A graph G and a set of subgraphs S (described by a g-trie T)
Result: Frequencies of all elements of S

```
 1: procedure CountAll($T, G$)
 2:     for all vertex $v$ in $G$ do
 3:         for all children $c$ in $T.root$ do
 4:             Count($c$, $\{v\}$)
 5: procedure Count($T, V_{used}$)
 6:     $V \leftarrow$ MatchVertices($T, V_{used}$)
 7:     for all vertex $v$ in $V$ do
 8:         if $T.isLeaf$ then
 9:             $T.frequency$ += 1
10:         else
11:             for all children $c$ in $T$ do
12:                 Count($c$, $V_{used} \cup \{v\}$)
13: procedure MatchVertices($T, V_{used}$)
14:     $V_{conn} \leftarrow$ vertices in $V_{used}$ connected to the vertex being added
15:     $m \leftarrow$ vertex of $V_{conn}$ with smallest neighborhood
16:     $V_{cand} \leftarrow$ neighbors of $m$ that respect connections to ancestors
          and symmetry breaking conditions
17:     return $V_{cand}$
```

The algorithm uses the information stored in the g-trie to guide the search by constraining it. Initially all vertices are considered potential occurrences, since they all match the g-trie root (lines 2 to 4). Afterwards, all valid following vertices are found and for each of them, if we are at a g-trie leaf (which means we have just found an occurrence of a desired subgraph) then we increment its frequency, otherwise continue recursively to match the next g-trie nodes (lines 6 to 12). A valid following vertex is a vertex that matches the current g-trie node. To find these, we start from the current partial match and find a vertex with a connection to the vertex being added and has a minimal number of neighbors (lines 14 and 15). Then we look for the neighbors of this node that have the same set of connections to the partially matched vertices and respect each symmetry condition given by the current g-trie node (line 16). Again, we refer to [24] for more information.

4 Graph Representations

4.1 Large scale representations

In the context of this paper and, in general, the context of Subgraph Census and related metrics (like network motif analysis) we define a large scale representation as a representation which has a memory usage that scales with the number of edges or the number of nodes of the network. This forbids classic representations like adjacency matrices that, as we shall see in the next subsection, allow for more efficient primitives required by the base algorithm.

The reason for this restriction is based on the applicability of Subgraph Census algorithms. All of the state of the art algorithms have a complexity that is super exponential (excluding specific applications like triangle counting), which is based on the natural combinatoric explosion of the number of subgraphs, even of the smallest sizes, as the networks grow larger. Even though there is a lot of space for improvement, this trend will most likely go on. Thus, for most networks with a number of nodes in the order of 100 thousand, the calculations start to last several hours or days, even for a $k = 3$ computation. For larger values of k this is even worse, as expected. These numbers are based on the results obtained by several established algorithms like [24,20,13], which are omitted for briefness.

With the development of different techniques, like efficient parallel algorithms [27,18] or algorithms that approximate the results, for example, by obtaining a sample of the total subgraphs [22], it is possible to increase the applicability and run calculation on networks with up to a million or 10 million nodes in feasible time.

Thus, even though it is not feasible to apply the state of the art algorithms to the largest networks available, it is possible to apply it to networks of around a million nodes. If one were to use an adjacency matrix, or any representation with memory complexity proportional to the square of the number of nodes, this would take about one terabyte of memory, which is too much for most machines.

4.2 Role of edge verification

Having described the base algorithm, it is easy to observe that there are two primitives that the graph representation needs to handle: generating the list of neighbors of a given node in order to generate the node for the candidate list, which is easily achievable with an adjacency list representation; determining if two given nodes are connected, which is needed in order to do match the partially enumerated subgraph with the current g-trie node.

The latter operation is obvious if we have an adjacency matrix, but since the goal of this paper is to be able to scale to larger networks, that is not feasible.

So a representation like an adjacency list is required. However, the question then arises: how much weight does this operation have on the full computation? To answer it, we performed a series of tests described in the following paragraphs.

The obvious first thing to do is to profile the code using any profiler. We did so on the original g-tries code from [24] (with small modifications to remove hard coded uses of an adjacency matrix, which were initially used to increase efficiency), which uses an adjacency matrix as the base representation, using `kcachegrind` [17] a tool from `valgrind` (we chose it because it is reliable and simple to use). We ran it on some of the data sets described in Section 5 with small parameters, since the profiler slows down the computation in order to examine memory calls and the program state. Table 1 shows the obtained results.

Table 1. Percentage of time spent in edge verification primitive

Network	Jazz	Foldoc	Metabolic
(k)	(4)	(3)	(4)
% `of total time`	35.76	28.43	38.97

Table 1 clearly shows that this graph primitive has a relevant weight in the run time of the whole algorithm, ranging from 30 to 40% of the computation. However, even though the previous results seem to indicate it is a heavy operation, it would not mean that there was a lot of space for optimizations, that is, it is possible that a naive representation would only be a small fraction slower than the adjacency matrix representation. Thus we ran the original code against a modified code that used an adjacency list with sorted lists in order to perform simple binary searches in connectivity lookups. Again, we did so on the same data sets as in the previous test. The results obtained are summarized in Table 2.

Table 2. Slow down factor of binary search in relation to the adjacency matrix

Network	Jazz	Foldoc	Metabolic
(k)	(5)	(4)	(5)
`Slow down`	4.41	2.99	4.31

This shows that a naive representation can be much slower than the base adjacency matrix one and, indeed, this tendency was followed in most of the other datasets, as can be observed in Section 5.

To end this subsection, we will discuss yet another important detail of this operation that will be important to the rest of the paper. As Table 1 showed, the operation of determining if two nodes are connected in the graph has a considerable weight in relation to the total computation time, however the operation itself is a very light one (constant for adjacency matrices). This indicates the reason why it has a considerable weight is because the operation is performed a huge number of times. This is confirmed by counting the number of operations done in different datasets, which yielded huge values in relation to the number of times most other operations are performed. We omit these tests for brevity.

4.3 Proposed representations

Now that we have established the need for efficient representations and the requirements and target of those, we will describe a set of possible representations that we will then study and compare on different data sets. The goal of each method is to perform an operation of checking if two nodes are connected, since this is the bottleneck primitive. The following list details all the studied representations and labels them with simple three letter names (like `BNS`). These labels will be used in further discussions and on the results section. All of these methods are built on top of an augmented simple adjacency list representation.

Linear search [LIN] A simple linear search through the elements of each list ($\mathcal{O}(|V|)$). It is a trivial and very inefficient method (unless the list size is very tiny), but it is included for completeness and comparison purposes.

Binary search [BNS] The classic divide and conquer approach to finding elements in sets ($\mathcal{O}(\log |V|)$). It requires the neighbor list of each node to be sorted in the pre computation step.

Interpolation search [IPS] The well-known adaptation of binary search that assumes the data is uniformly distributed or close to that [2] ($\mathcal{O}(\log \log |V|)$). It also requires a sorted neighbor list for each node.

Hash table node based [HSN] Each node has a simple hash table with size $\frac{|E|}{|V|}$, where the hash table is simply the *mod* of its size ($\mathcal{O}(1)$). To sort out collisions it uses a simple linked list. Requires a pre computation step of creating and filling the hash tables.

Hash table edge based [HSE] A different hash table setup where each node has a hash table of a constant number times its original neighbor list size ($\mathcal{O}(1)$). The constant used in the implementation was 2.5, where this value was fined tuned after several manual experiments in order to balance time and memory efficiency.

Trie [TRI] A prefix-tree of digits of the individual elements of the original adjacency list ($\mathcal{O}(\log |V|)$). Requires a pre computation step of creating the prefix-tree.

Hybrid [HBR] A hybrid approach that combines three of the previously mentioned approaches to apply them in the best possible way. For an adjacency list of size less than 2, a simple linear search is used; for the $\frac{|E|}{|V|}$ nodes with highest degree, a line from the adjacency matrix is stored; finally for the rest, the edge based hash table method is used. It requires the pre computation of the hash table.

Most of these techniques have an extra constant factor of memory required, with exception to the binary search and interpolation search methods. Even though these methods in practice require some more memory to work, they usually pay off in terms of running time (as we will see on Section 5). Also, most networks where this method is applicable have a number of edges where a constant factor is feasible (this stems from the discussion of the beginning of the section). However, a benefit of some of the representation is that the extra usage of memory is tunable. For example, if there is some tighter memory restriction, the `HSE` method can use a lower constant (we used 2.5, as mentioned) in order to save memory. Likewise, if there is a loosened memory restriction, this constant can be increased in order to obtain better results (we justified our choice in the method definition above).

Before heading for the next section, a quick note about the hybrid method. We tried different possible hybrid methods, for example, combining the different hash table methods with the trie method, using an heuristic method that estimated the average required number of accesses in the linked list of the hash table and the trie nodes. However, the overhead of choosing which method to apply (the actual choosing step is pre computed, but the access to the pre computed result) did not pay off against the fastest methods. The specified hybrid method had better results and thus was chosen as the preferred one. We also tested different set ups to use more lines of the adjacency matrix instead of only $\frac{|E|}{|V|}$, but this particular value, besides ensuring a memory complexity of $\mathcal{O}(|E|)$, balanced the overhead of choosing the method to use with its benefits, which was obtained by running several alternatives with the datasets of Section 5.

4.4 Optimizations

To complement the methods described in the previous subsection, several optimizations where tried, some with success and others without. We will list them here in the same fashion.

Optimal operators Based on the tests performed in the beginning of this section and on some of the results of Section 5, it is noticeable that small changes in one method lead to a large impact on the overall run time. For example, if one method only requires doing a couple of sums (like `BNS`), but another has to perform one or two division operations (like `IPS`), the latter is usually a lot slower. This is due to the large number of times the primitive of edge verification is called.
Thus, in all methods where an operation of $a \mod b$ (where a, b are arbitrary integers) was required, instead the closest power of two of b was determined, that is $\min(p|2^p \geq b)$, and the modular operation was performed as a more efficient `bitwise and` operation (& in `C++`) with $2^p - 1$ (which is equivalent to a $a \mod (2^p - 1)$ operation). For example, this was done to the hash table sizes of methods `HSN` and `HSE`. It was also done in method `TRI`, by considering the numeric representation of each element in base 16 (where a $\mod$ operation is a `bitwise and` with 15).

Simple Node Cache For each node, a simple cache was built that keeps the last found node on the cache. Each cache is divided into $\log(|V|)$ levels, meaning there are $\log(|V|)$ small caches per node. When a node u is found to be connected to v in a primitive call, u is stored on the $u \mod \log(|V|)$ small cache of v and replaces the previous value. In practice, the same optimization as in the last item is performed and the $\mod$ operation is avoided.

The goal of this type of cache is to make use of some inherent locality and redundancies that the base algorithm has in its pattern of calls to the primitive. This was found by analyzing the spectrum of calls in different data sets, but we omit the results for briefness.

Bloom Filter Another common trick for the kind of primitive we are addressing is a bloom filter [5]. This data structure has a bit array of m bits and uses h hash functions to map each integer to h of the m bits. There are different possibilities for an implementation of a bloom filter, we opted to try the simplest ones, because the described heavy number of calls would turn a more complex implementation too inefficient. Hence, we used simple hash functions of the kind $a_i \times x \mod m$, where we fix h different integers a_i. As in the previous sections, we used m as a power of two minus one two improve the $\mod$.

We tried to hand-tune the values of m and h, unfortunately, we could not find a good balance. If h is too big, the actual calculation of all the hash functions is too inefficient, if it is too small the effects of using a bloom filter are unnoticeable. Thus, we cold not reach an efficient implementation of a bloom filter. We also considered similar data structures, like a quotient filter [4], but did not implement them since they might have similar problems. For this reason, we omit our results using a bloom filter from Section 5.

5 Experimental Results

We now turn to the experimental evaluation. We implemented [1] these approaches in `C++` on top of the already existing code of the g-tries [24]. We ran all tests on a Linux machine with an AMD Opteron 6376 (2.3GHz) and 4GB of memory.

In order to compare with the adjacency matrix approach (which we will denote as `AMT` in a similar fashion to what was done in the previous section), we ran our implementations on data sets feasible for that approach. We list the data sets used in Table 3. Note that we included a wide range of networks, directed and undirected, ranging from social to biological to geographic networks in source, with different orders of magnitude. This is important to establish the generalness of the results.

We started by testing all the methods on all the data sets. To simplify the implementation, we implemented them all and choose dynamically (in the input) the method used, using a series of `if` instructions. To ensure the order of the `if` instructions does not interfere with the accuracy of the results, we permuted them regularly on different runs and averaged the results. Table 4 lists these results. The highlighted cells indicate the fastest time for each dataset.

Note first that there is a lot of fluctuation in the relative results between the various methods, for example, the `BNS` outperforms `IPS` in some datasets but is outperformed in others, the same thing happens with `TRI` and `HSN`. This indicates that different types of graphs prefer different representations, which

[1] The initial version of the implemented code can be found at `https://github.com/ComplexNetworks-DCC-FCUP/gtrieScanner/tree/DynamicGraph`

Table 3. Datasets used in the experiments

Network	Directed	Nodes	Edges	Avg. Degree	Type	Source
Jazz	No	198	2,742	13.85	Social	Arenas [9]
Facebook	No	4,039	88,234	21.85	Social	SNAP [15]
Wordnet	No	146,005	656,999	9.00	Semantic	KONECT [8]
Enron	No	36,692	367,662	10.02	Social	SNAP [14]
Foldoc	Yes	13,356	120,700	9.04	Semantic	Pajek [3]
Metabolic	Yes	453	2,025	4.47	Biological	Arenas [9]
Flights	Yes	2,939	30,501	20.76	Geometric	KONECT [19]
Epinions	Yes	75,879	508,837	13.41	Social	KONECT [28]

Table 4. Detailed experimental results for the 8 datasets used (times in seconds)

Method	Jazz	Facebook	Wordnet	Enron	Foldoc	Metabolic	Flights	Epinions
(k)	(6)	(4)	(4)	(4)	(4)	(5)	(4)	(3)
`AMT`	198.03	68.39	-	-	28.12	21.20	20.74	-
`BNS`	1,129.39	320.08	767.09	1,034.71	88.40	107.02	109.25	36.51
`IPS`	1,104.48	404.25	847.18	1,549.16	81.99	86.55	161.27	69.59
`HSN`	488.22	159.46	522.81	877.59	65.67	80.77	63.88	50.09
`TRI`	691.49	223.43	658.89	749.46	65.19	60.91	54.45	28.19
`LIN`	1,234.37	513.35	917.26	1,380.28	171.06	151.81	181.16	118.38
`HSE`	461.08	135.14	**438.78**	**557.50**	51.39	49.98	43.91	**18.58**
`HBR`	**289.31**	**125.83**	480.83	586.49	**50.67**	**45.80**	**39.34**	20.77

(-) For these networks the adjacency matrix method requires too much memory

means there is space for hybrid methods to use the best methods for different graph sources. It is also important to note how the results show that this is a very heavy operation, meaning it is called a massive number of times. For example, even though the `IPS` method has a better time complexity ($\mathcal{O}(\log\log|V|)$) than `BNS` ($\mathcal{O}(\log|V|)$) it is outperformed in about half the networks. Moreover, it is outperformed in the networks with highest average degree, where it should have a bigger theoretical advantage, however, since the operation is so massively called, the requirement of performing several division and multiplication operations drags down the overall performance. There could be different explanations, like the neighbor distribution is not close to uniform, but further testing like including dummy division operations in the `BNS` implementation seem to confirm the previous intuition.

The most important conclusion to take from Table 4 is that `HSE` and `HBR` (that uses the former) consistently outperform the rest. Intuitively, this is related to the fact that `HSE` only requires very light operations in most cases, except when there are collisions (where it is necessary to follow pointers to iterate the linked list), but the number of collisions is generally small since the neighbor distribution is well behaved and the size of the hash table is, in the case the chosen parameters of our implementation, over double the size of the element set. The `HBR` seems to capture the best of `HSE` since it has similar results for most networks, outperforming it on most. The most relevant example is the one of the Jazz dataset, where `HBR` takes almost 50% less time than `HSE`, indicating that there are some graph types where bypassing the hash map pays off. However, `HBR` seems to be mostly on par with `HSE`. One of the reasons of such is that `HSE` has a high constant in the implementation, thus if the memory restrictions are tighter, the benefits of using an hybrid method are greater.

Overall, we seem to achieve a method that ranges from almost one to two times slower than the base `AMT` method. Further tests indicate that reducing the constant for `HSE` (saving more memory) can yield similar results (including hybrid approaches) in most networks (due to cache and similar effects, using more memory does not yield a linear time benefit, it has more of a threshold effect).

Adding to these results, we will also briefly show the benefits of adding a cache as described in the previous section. Not all methods benefit from the cache, for example, the `TRI` has a slow down in most networks due to the overhead of filling the cache (it is most likely a implementation issue, but it is hard to avoid). Thus, we chose to only show the results of the cache on 3 of the methods with better results: `HSN`, `HSE`, `HBR`. We also tested our cache with the other methods, but the results are not so relevant and hence were omitted. We will denote a method using the small cache by adding a `+CH` in its abbreviation (for example `HSN+CH`). Table 5 depicts these results, we included the results of the `AMT` method obtained previously to serve as a baseline.

Table 5. Experimental results for the small cache behavior (times in seconds)

Method	Jazz	Facebook	Wordnet	Enron	Foldoc	Metabolic	Flights	Epinions
(k)	(6)	(4)	(4)	(4)	(4)	(5)	(4)	(3)
`AMT`	198.03	68.39	-	-	28.12	21.20	20.74	-
`HSN`	488.22	159.46	522.81	877.59	65.67	80.77	63.88	50.09
`HSN+CH`	406.57	110.02	481.06	695.62	75.97	78.06	56.50	44.44
`HSE`	461.08	135.14	**438.78**	557.50	51.39	49.98	43.91	18.58
`HSE+CH`	392.50	**100.22**	439.77	**534.67**	**45.51**	46.42	40.27	**18.93**
`HBR`	**289.31**	125.83	480.83	586.49	50.67	**45.80**	39.34	20.77
`HBR+CH`	334.32	114.45	453.24	569.73	45.76	53.33	**33.12**	19.05

(-) For these networks the adjacency matrix method requires too much memory

We can observe minor benefits in a lot of the networks. This showcases how it is possible to take advantage of the regularity of the distribution of neighbors with little extra memory spent. Unfortunately, the small cache seems to clash with the hybrid part of `HBR`, meaning the small cache had higher benefits in certain nodes with higher degree and thus more queries to the edge verification primitive, and part of those nodes do not use the small cache since they have the full adjacency matrix on `HBR`.

We conclude this section by drawing a small conclusion on our findings in order to select the best approach possible. Our results and analysis clearly show that the `HBR` and the `HSE+CH` worked the best on our datasets. Intuitively, this makes sense since both methods only use very light operations and can mimic fairly well the behavior of an adjacency matrix. We stripped the previous implementation[2] of all other methods and kept only HBR and ran it with the same datasets. These results are described in Table 6.

Table 6. Experimental results for the final implementation (times in seconds)

Method	Jazz	Facebook	Wordnet	Enron	Foldoc	Metabolic	Flights	Epinions
(k)	(6)	(4)	(4)	(4)	(4)	(5)	(4)	(3)
AMT	198.03	68.39	-	-	28.12	21.20	20.74	-
Final run time	235.02	102.33	397.68	495.56	35.02	30.19	27.78	15.76
Slow down	1.19	1.50	-	-	1.25	1.42	1.34	-

(-) For these networks the adjacency matrix method requires too much memory

These show a slow down factor improvement from about 4 of the initial naive binary search, to a factor of less than 1.5 on average.

6 Conclusion

In this paper we studied a number of alternative graph representations that scale with the number of edges or the number of nodes of the network memory-wise, in order to extend the applicability of current algorithms to larger networks. The goal was to find an efficient representation to be used by subgraph census algorithms, more specifically, our study was tailored to a previous work of ours, a state-of-the-art data structure called g-tries.

We studied different methods with several optimizations and additional improvement strategies. In the end, they converged in a hybrid method that tries to apply some of the best methods in their preferred situations. The described method is easily tuned to be used with different memory restricted environments. In the end, it improved the slow down factor of the naive binary search method in relation to the adjacency matrix from 4 to around 1.5.

This work did not have any parallel considerations, but a possible further work would be to do this type of analysis in parallel versions of the subgraph census algorithms, considering different effects that can harm the computation (like cache hierarchies) and even distributing the graph by different machines. Another different progression would be applying the methods to different datasets and obtaining relevant results on those.

References

1. Albert, I., Albert, R.: Conserved network motifs allow protein–protein interaction prediction. Bioinformatics 20(18), 3346–3352 (2004)
2. Andersson, A., Mattsson, C.: Dynamic interpolation search in o (log log n) time. In: Automata, Languages and Programming, pp. 15–27. Springer (1993)
3. Batagelj, V., Mrvar, A.: Pajek datasets. `http://vlado.fmf.uni-lj.si/pub/networks/data/` (2006)
4. Bender, M.A., Farach-Colton, M., Johnson, R., Kraner, R., Kuszmaul, B.C., Medjedovic, D., Montes, P., Shetty, P., Spillane, R.P., Zadok, E.: Don't thrash: how to cache your hash on flash. Proceedings of the VLDB Endowment 5(11), 1627–1637 (2012)

[2] The implemented code can be found at `https://github.com/ComplexNetworks-DCC-FCUP/gtrieScanner/tree/finalGraph`

5. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM 13(7), 422–426 (1970)
6. Cook, S.A.: The complexity of theorem-proving procedures. In: ACM Symposium on Theory of computing. pp. 151–158. STOC, ACM, New York, NY, USA (1971)
7. Dobrin, R., Beg, Q.K., Barabási, A.L., Oltvai, Z.N.: Aggregation of topological motifs in the escherichia coli transcriptional regulatory network. BMC bioinformatics 5(1), 10 (2004)
8. Fellbaum, C.: WordNet. Wiley Online Library (1998)
9. Gleiser, P.M., Danon, L.: Community structure in jazz. Advances in Complex Systems 06(04), 565–573 (2003)
10. Grochow, J., Kellis, M.: Network motif discovery using subgraph enumeration and symmetry-breaking. Research in Computational Molecular Biology pp. 92–106 (2007)
11. Juszczyszyn, K., Kazienko, P., Musiał, K.: Local topology of social network based on motif analysis. In: Knowledge-based intelligent information and engineering systems. pp. 97–105. Springer (2008)
12. 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)
13. Khakabimamaghani, S., Sharafuddin, I., Dichter, N., Koch, I., Masoudi-Nejad, A.: Quatexelero: An accelerated exact network motif detection algorithm. PLoS ONE 8(7), e68073 (07 2013)
14. Klimt, B., Yang, Y.: Introducing the enron corpus. In: CEAS (2004)
15. Leskovec, J., Mcauley, J.J.: Learning to discover social circles in ego networks. In: Advances in neural information processing systems. pp. 539–547 (2012)
16. 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)
17. Nethercote, N., Walsh, R., Fitzhardinge, J.: Building workload characterization tools with valgrind. In: Workload Characterization, 2006 IEEE International Symposium on. pp. 2–2. IEEE (2006)
18. Oliveira Aparicio, D., Pinto Ribeiro, P.M., Da Silva, F.M.A.: Parallel subgraph counting for multicore architectures. In: Parallel and Distributed Processing with Applications (ISPA), 2014 IEEE International Symposium on. pp. 34–41. IEEE (2014)
19. Opsahl, T., Agneessens, F., Skvoretz, J.: Node centrality in weighted networks: Generalizing degree and shortest paths. Social Networks 32(3), 245–251 (2010)
20. Paredes, P., Ribeiro, P.: Towards a faster network-centric subgraph census. In: Advances in Social Networks Analysis and Mining (ASONAM), 2013 IEEE/ACM International Conference on. pp. 264–271. IEEE (2013)
21. Paredes, P., Ribeiro, P.: Rand-fase: fast approximate subgraph census. Social Network Analysis and Mining 5(1), 1–18 (2015)
22. Ribeiro, P., Silva, F.: Efficient subgraph frequency estimation with g-tries. In: International Workshop on Algorithms in Bioinformatics. WABI, vol. 6293, pp. 238–249. Springer (2010)
23. Ribeiro, P., Silva, F.: Discovering colored network motifs. In: Complex Networks V, pp. 107–118. Springer (2014)
24. Ribeiro, P., Silva, F.: G-tries: a data structure for storing and finding subgraphs. Data Mining and Knowledge Discovery 28(2), 337–377 (2014)
25. Ribeiro, P., Silva, F., Kaiser, M.: Strategies for network motifs discovery. In: IEEE International Conference on e-Science. pp. 80–87. e-Science (2009)
26. Ribeiro, P., Silva, F., Lopes, L.: Efficient parallel subgraph counting using g-tries. In: Cluster Computing (CLUSTER), 2010 IEEE International Conference on. pp. 217–226. IEEE (2010)
27. Ribeiro, P., Silva, F., Lopes, L.: Parallel discovery of network motifs. Journal of Parallel and Distributed Computing 72(2), 144–154 (2012)
28. Richardson, M., Agrawal, R., Domingos, P.: Trust management for the semantic web. In: The Semantic Web-ISWC 2003, pp. 351–368. Springer (2003)
29. Sporns, O., Kötter, R.: Motifs in brain networks. PLoS Biol 2(11), e369 (2004)
30. Wernicke, S.: Efficient detection of network motifs. IEEE/ACM Transactions on Computational Biology and Bioinformatics pp. 347–359 (2006)