Counting subgraphs: from static to dynamic networks

Pedro Miguel Reis Bento Paredes

Master's degree in Computer Science

Computer Science Department

2017

Supervisor

Pedro Manuel Pinto Ribeiro, Assistant Professor, Faculty of Science, University of Porto

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, / /

Abstract

Networks are ubiquitous representations of many real-world systems. As such, there is a need to develop tools and methods to analyze them, so as to gather information of their underlying systems. In the large set of techniques that were developed on the last few decades, methods that rely on looking into the frequency of certain small subgraphs, like frequent subgraph discovery, network motif discovery and graphlet based metrics, were extensively studied and improved. At the core of these techniques is one common problem: counting subgraphs, which is formally known as the subgraph census problem. In this problem we are given a graph and are asked to compute the frequency of a set of subgraphs. This is a hard problem, closely related to the subgraph isomorphism problem, a known NP-Complete problem.

We looked into this problem from two different complementary perspectives: a static one and a dynamic one. From the static point of view, we developed two algorithms that directly tackle this problem by finding all subgraphs of a given size k. The first one computes this exactly, whereas the second one computes an approximation of the real value, by obtaining an unbiased sample of the full set of subgraphs. From the dynamic point of view, we first built an algorithm that updates the frequency of subgraphs when the input graph has an edge added or removed. Furthermore, we defined the dynamic isomorphism problem as the problem that consists of finding the isomorphism information (or a canonization) of a graph in a stream, that is, has edges being added or removed.

Finally, we thoroughly analyzed each technique, comparing it with the appropriate past approaches and relevant baselines, in order to establish their meaningfulness and to show they live up to what was theorized about them and also that they outperform their corresponding past approaches.

Keywords: Graph Algorithms, Graph Mining, Subgraph Counting, Graph Isomorphism

Resumo

Redes são representações ubíquas de vários sistemas do mundo real. Assim, existe uma necessidade para desenvolver ferramentas e métodos para as analisar, com o objetivo de adquirir informação sobre os sistemas subjacentes. Do leque de técnicas que foram desenvolvidas durante as últimas décadas, métodos que olham para a frequência de certos subgrafos pequenos, como *frequent subgraph mining* (análise de subgrafos frequentes), *network motif discovery* (descoberta de motivos de redes) e métodos baseados em *graphlets* (subgrafos pequenos), foram estudados e melhorados extensivamente. No centro destas técnicas surge um problema comum: contar subgrafos, que é formalmente conhecido como o problema do censo de subgrafos. Neste problema é nos dado um grafo e pedem-nos para calcular a frequência de um conjunto de subgrafos. Este é um problema computacionalmente difícil, relacionado com o problema de isomorfismo de subgrafos, um problema conhecido por ser NP-Complete.

Olhámos para este problema de duas perspetivas complementares: uma estática e uma dinâmica. Da perspetiva estática, desenvolvemos dois algoritmos que abordam este problema diretamente, ao encontrarem todos os subgrafos de um dado tamanho k. A primeira calcula o anterior de uma forma exata, enquanto que o segundo calcula uma aproximação do valor real, através de uma amostra sem enviesamento do conjunto total de subgrafos. Do ponto de vista dinâmico, primeiro construímos um algoritmo que atualiza a frequência de subgrafos quando removemos ou adicionamos uma aresta ao grafo objetivo. Para além disso, definimos o problema do isomorfismo de grafos dinâmico como o problema que consiste em encontrar informação de isomorfismo (ou a canonização) de um grafo em *stream*, ou seja, quando lhe são adicionadas ou removidas arestas.

Finalmente, analisámos cada técnica, comparando a com o conjunto apropriado de métodos pré existentes e com patamares base relevantes, com o objetivo de estabelecer a sua significância e mostrar que confirmam o que foi teorizado sobre elas e que representam melhorias em relação ao corpo de trabalho que as precede.

Palavras-chave: Algoritmos em grafos, Análise de grafos, Contagem de subgrafos, Isomorfismo de grafos

Contents

List of Tables

List of Figures

Chapter 1

Introduction

As technology progresses, we tend to get ever more interconnected. As such, we often turn to networks or graphs as models of a lot of different complex physical systems in a multitude of fields, like in biology, sociology or medicine [11]. Why graphs? Graphs are very simple structures, yet they can contain a lot of information on their complex and convoluted inner topology. Hence, when one develops a new method or tool to analyze and extract meaningful information from a graph it translates into a new way of shedding light on our world and potentially having a big impact on society.

It was with this motivation that the area of graph mining developed over the past decades. One way of studying networks is to search for interesting groups of nodes. These groups may have a relatively large size, as is the case with community detection [14], however, they can also be of smaller sizes, like it is the case on frequent subgraph discovery [31], network motif discovery [40] or graphlet based metrics [46]. We focus on the latter.

These methodologies have been applied with success to a wide range of real systems, such as in the social networks domain, where motifs have been used, for instance, to characterize and classify co-authorship networks [9] or wikipedia edition networks [59]. Likewise, graphlets have been used to provide a complete characterization of social networks, allowing the selection of an adequate graph model [23]. These methodologies have also been successfully applied to other domains, such as biological networks [2, 53], engineering systems like electronic circuits [22] and also on software architecture [54].

In Figure 1.1 we show a representation of a real complex networks, that models collaborations between a set of Jazz musicians. Its analysis can tell us, for example, how these collaborations emerged and how similar they are to other human relationship networks, like a network of friends in a certain location. However, even though this is a relatively small network, it is almost impossible to extract any relevant information just by looking into its visual representation, which suggests computational method are a must. This particular network will be used as a benchmark in a later analysis in Chapter 5.

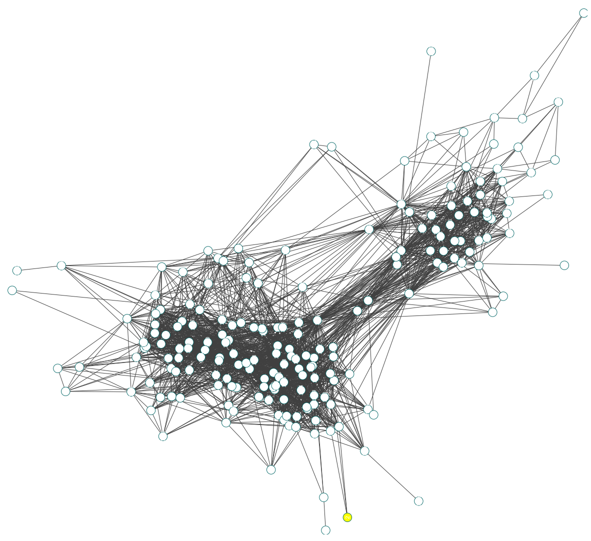


Figure 1.1: A network of Jazz musicians collaborations.

Underneath many of these techniques we can pinpoint a common problem: the subgraph census problem. In this problem, we are given a graph and are asked to compute the frequency of a set of different (in terms of isomorphism) subgraphs. This set can be all the subgraphs with a certain number of nodes k, a single subgraph or any random set. In this thesis we focus specifically on the first situation.

The subgraph census problem has been thoroughly studied over the past two decades [27, 40, 45, 48, 58]. This is a hard problem, since at its core it is related to the subgraph isomorphism problem and, by consequence, the graph isomorphism problem. The former is a well-known NP-Complete problem [10]. Consequently, the only approaches we have are experimental ones, that try to reduce as much as possible the amount of performed computation. Additionally, due to the combinatorial explosion of occurring subgraphs, all known algorithms can only support finding frequencies of very small subgraphs, with a number of nodes not exceeding 9 or 10 (if we consider simple, undirected graphs).

The previous discussion focuses on a problem which is (in the way we defined it) static, meaning our input graph or structure is fixed. However, in the real world almost everything is in motion, constantly changing. Thus, there is an interest in studying graph problems on a dynamic or streaming environment, that is, an environment where our input is changing, evolving.

There are multiple models of streaming graphs [36], that consider different types of updates, usually either edge additions, deletions or both. Particularly in the graph mining realm, there has been an increasing interest in studying dynamic graphs problems, namely, by introducing or altering known metrics to suit temporal graphs (graphs where edges have timestamps that represent intervals of time where they are active) [20, 29, 49, 55].

With this is mind, in this thesis we will take somewhat of a journey around the subgraph census problem. We start by concentrating on the static version of the problem and we will walk towards a dynamic version. In each stop, we will also discuss different approaches and variants of the problem. The ultimate goal is to provide novel algorithms and methods for the discussed problems so as to facilitate the analysis of networks using techniques that rely on these problems.

1.1 Goals and contributions

As mentioned, our goals roam around the subgraph census problem. To ease their presentation, we divided our work into four parts, tied into two main groups: one concerning static approaches and one concerning dynamic approaches.

For the static approaches, we propose two algorithms: the first is an algorithm that finds the exact frequency of all subgraphs of a given size k; the second is an approximation algorithm for the same problem, which obtains an unbiased sample of subgraphs in order to estimate the correct count.

For the dynamic approaches, again we propose two algorithms: the first is a variant of the presented static algorithm that updates the frequency computations when a single edge is either added or removed; the second is an algorithm that solves a related problem called the dynamic graph isomorphism problem.

We summarize our contributions on the following list:

- We propose an efficient algorithm that determines the exact frequency of all subgraphs of a given size k and show that it outperforms its known main past approaches. This was originally published in [42];
- We propose a version of the previously mentioned exact algorithm that approximates the frequency of subgraphs, trading accuracy for speed, by obtaining an unbiased sample of the total subgraphs. Also, we analyze its behavior, from runtime to convergence of the approximation. This was originally published in [43];
- We propose a version of the previously mentioned exact algorithm that efficiently updates the frequency of subgraphs when an edge of the original graph is either added or removed. This was originally published in [51];
- We describe a new problem, which we called the dynamic graph isomorphism problem, that amounts to finding a canonization of a graph and then updating it when it suffers edge additions or removals. We also show its usefulness in the context of the subgraph census problem. This is unpublished work;
- We propose an efficient algorithm that solves the dynamic graph isomorphism problem in a streaming-aware fashion for small graphs. We also thoroughly analyze its properties in terms of memory usage and runtime. This is unpublished work;

1.2 Thesis outline

This thesis is structured as follows:

- **Chapter 2 - Preliminaries** sets a consistent notation and terminology and briefly discusses some background concepts and past work on the topic of this thesis;

- **Chapter 3 - A Static Approach** focuses on the two proposed algorithms that target the static version of the subgraph census problem;
- **Chapter 4 - A Dynamic Approach** focuses on the two proposed algorithms that target dynamic variants of the subgraph census problem;
- **Chapter 5 - Experimental Analysis** shows the efficiency and the properties of the proposed methods;
- **Chapter 6 - Conclusion** summarizes the presented material and also wraps up this thesis;

So, without further ado, we shall delve into what matters.

Chapter 2

Preliminaries

This chapter will cover all the background theory that the thesis will use. First, we define some notation and terminology we will consistently use in all of the following chapters. We then discuss a problem closely related to the main focus of this thesis, namely the graph isomorphism problem. More than a related problem, it is a central primitive in most algorithms that tackle the subgraph census problem, our focus in this thesis. We follow with the latter problem, by formally defining it, as well as discussing some of the more important past approaches to the problem. We close with a note on network motifs, which are one of the main reasons why there is an interest in solving the subgraph census problem.

2.1 Notation and terminology

A *network* or *graph* G is a pair $(V(G), E(G))$, where $V(G)$ is a set of *vertices* and $E(G)$ a set of *edges*, represented by pairs (a, b) where $a, b \in V(G)$. We call each vertex in an edge an *endpoint*. We define the *size* of G, denoted by $|V(G)|$, as the number of vertices. A graph with size k is denoted as a k-graph. A graph G is called undirected if $\forall u, v \in V(G), (u, v) \in E(G) \leftrightarrow (v, u) \in E(G)$ and directed otherwise. The *degree* of a vertex v is the number of unique edges that have v as one of its endpoints, which we denote by $\delta(v)$. In the case of directed graphs, we distinguish between *indegree* and *outdegree* of v, respectfully $\delta(v)_i$ and $\delta(v)_o$, as the number of unique edges that have v as the second endpoint and as the first endpoint.

In this thesis we only consider *simple graphs*, with no multiple edges between two vertices and no loop, and directed graphs with no multiple edges between the same endpoints and no loops. We assume every graph is *labeled* so that every vertex of a graph G is assigned a distinct integer from 1 to $|V(G)|$. We denote the label of a vertex v by $L(v)$. *Graph equality* between two graphs G and G' with the same number of vertices, denoted by $G = G'$, is observed if and only if for all $u, v \in V(G)$ and $u', v' \in V(G')$ such that $L(u) = L(u')$, $L(v) = L(v')$, then $(u, v) \in E(G) \Leftrightarrow (u', v') \in E(G')$.

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 *induced* if and only if $\forall u, v \in V(G_k) : (u, v) \in E(G) \leftrightarrow (u, v) \in E(G_k)$ and is called *connected* if all vertex pairs are connected by a sequence of edges. The *neighborhood* of a vertex $v \in V(G)$ is defined as $N(v) = \{u : (u, v) \in E(G) \lor (v, u) \in E(G)\}$ and similarly we define the neighborhood of a set of vertices $S \subseteq V(G)$ of G, denoted as $N(S)$, as the set of all of the neighbors of vertices in S not included in S. The *exclusive neighborhood* of a vertex v in a graph G relative to a set of vertices $S \subseteq V(G)$ is defined as: $N_{exc}(v, S) = \{u : u \in N(v) \land u \notin N(S) \land u \notin S\}$.

A *permutation* π is an element of the symmetric group S_n, with its usual composition operation $\circ$. We denote the *image* of an integer x under the permutation π by π^x. For a permutation π, we denote by $\overline{\pi}$ the *inverse* of π, that is the permutation such that $\pi \circ \overline{\pi} = \mathbf{1}$, where $\mathbf{1}$ is the identity permutation. A *transposition* is a permutation that only swaps two elements and fixes all the others. Given a graph G with vertex set $V(G) = \{v_1, v_2, \ldots\}$ with $L(v_i) = i$, and a permutation π, we denote by G^π the graph with the same vertex set but with $L(v_i) = \pi^i$, meaning we permute the labels of the vertices. To simplify notation, for a given vertex v of a graph G with label i and a permutation π, we write π^v to denote the vertex in G^π with label π^i.

Two graphs G_1 and G_2 are said *isomorphic* if there is a permutation π such that $G_1^\pi = G_2$, we denote this by $G_1 \cong G_2$. The *isomorphism graph class* of a graph G is the equivalence class of G in the relation of isomorphism of graphs. For a particular k-subgraph, G_k, of a graph G, we denote the set of all subgraphs of G that belong to the same isomorphism graph class of G_k by $S(G_k, G)$ and we call *frequency* to the number of subgraphs of G that belong to that class and denote it as: $F(G_k, G) = |S(G_k, G)|$.

An *automorphism* of a graph G is a permutation π such that $G^\pi = G$. We define $\text{Aut}(G)$ as the set of automorphisms of G. The *orbits* of a graph G are the equivalence classes of vertices of G under the action of automorphisms, this means two vertices u, v have the same orbit if there is $\pi \in \mathrm{Aut}(G)$ such that $\pi^u = v$ or $\pi^v = u$. A *canonical function* is a function C that, given a graph G, $C(G) \cong G$ and for any $\pi \in \mathrm{Aut}(G)$ we have $C(G^\pi) = C(G)$.

For a subgraph G_k of a graph G, an estimator of the value of $F(G_k, G)$ is denoted as $\widehat{F}(G_k, G)$. It is called *unbiased* if its expected value, denoted as $\mathbb{E}(\widehat{F}(G_k, G))$, is equal to $F(G_k, G)$ and *biased* if not.

A *graph changing operation* of cardinality n is a pair (x_1, x_2), where x_1 and x_2 are distinct integers between 1 and n. The application of a graph changing operation $\Delta = (x_1, x_2)$ of cardinality n over a graph G with $|V(G)| = n$ is the graph $G' = G\Delta$ with the same vertex set of G, where, if $v, u \in V(G)$ are such that $L(v) = x_1$ and $L(u) = x_2$: if $(v, u) \notin E(G)$ then $E(G') = E(G) \cup \{(v, u)\}$; if $(v, u) \in E(G)$ then $E(G') = E(G) \setminus \{(v, u)\}$. Thus, the application of a graph changing operation (x_1, x_2) is equivalent to toggling on or off the edge between the two vertices with labels x_1 and x_2. A *graph stream* S of cardinality n is a sequence of graph changing operations with the same cardinality. We call the *size* of a stream $|S|$ to the number of elements in S. The application of a graph stream $S = [\Delta_1, \Delta_2, \ldots]$ with cardinality n over a graph G with $|V(G)| = n$ is a sequence of graphs $[G, G\Delta_1, G\Delta_1\Delta_2, \ldots]$, denoted by $S(G)$. For a given stream S over a graph G, if we are only interested in every other k graph, meaning $S(G)_1$, $S(G)_{1+k}$, $S(G)_{1+2k}$, $\ldots$, we say the stream S has step k.

2.2 The graph isomorphism problem

The *Graph Isomorphism* problem (GI) consists in finding a bijection between the vertex sets of two graphs that preserves the vertex adjacency or state that one does not exist. It is a widely studied problem in several domains. Its theoretical interest arises from the fact that GI is trivially in NP but is still unknown whether it is NP-Complete or in P, even though it is considered unlikely that GI is NP-Complete [16]. Recently, the upper bound on the complexity was improved to quasipolynomial time [5].

From a practical point of view, it is used as a primitive for several methods that tackle different problems, like frequent subgraph discovery [31], graph matching [17] and our friend, the subgraph census problem [58]. As such, efficient practical methods that compute isomorphism information were developed [24, 38] based on several heuristics. One of the most well-known algorithms is called `nauty` [38], an exponential algorithm that performs exceptionally well in most inputs.

The *Graph Canonization* problem (GC) is a variant of graph isomorphism that consists of finding a canonical labeling (also called a *canon*) for a graph in a way that ensures that two graphs have the same canonical label if and only if they are isomorphic. Solving GC implies solving GI, since after knowing the canonical labels of two graphs determining if they are isomorphic is simply checking if the two labels are equal. However, in general, GI is not known to be equivalent to GC [4]. The most common practical approach to GI is by solving GC [38], since it is better suited for most applications where a collection of graphs needs to be sorted for isomorphism class.

We now provide a formal definition of the canonization problem.

Definition 1. *In the canonization problem we are given a graph G and are asked to provide a canonical labeling of G, such that for any $\pi \in \mathrm{Aut}(G)$, G^π has the same labeling.*

This problem is a known problem and will be used as a primitive in this paper. We use `nauty` [38] throughout this thesis whenever we need a method that solves this problem. However, note that any method that returns the canon of a graph could be used instead of `nauty`.

2.3 The subgraph census problem

This problem is the focus of this thesis. We will look into this problem and some of its variants (which we will present and discuss throughout this thesis), but in its essence its formal definition is the following:

Definition 2 (Subgraph Census Problem)**.** *Given an integer k and a graph G, determine the frequency of all different 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 is important to notice that we are only concerned with subgraphs that are both connected and induced. However, both are computationally equivalent, since it is possible to apply a simple linear transformation to obtain non-induced frequencies from induced frequencies, and vice-versa [45]. Note, also, how we distinguish occurrences. Other frequency concepts do exist and have been tested [50], but here we use the standard definition. This has direct implications on the number of existing subgraphs, with no downward closure on the frequencies, since a subgraph may appear more times than a subgraph contained in it. Figure 2.1 exemplifies a subgraph census for $k = 3$ a graph G with five nodes.

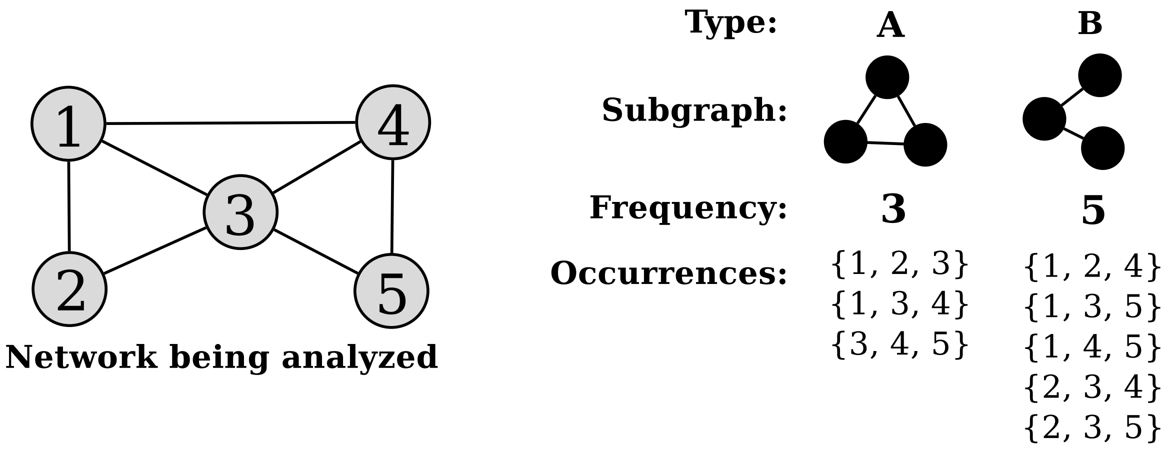


Figure 2.1: An example 3-subgraph census.

Also relevant is the distinction between a subgraph census and a subgraph enumeration, since both will be mentioned multiple times throughout this thesis. Even though some subgraph census algorithms perform an enumeration, this is not the case for the full range of existing techniques. An enumeration implies exhaustively generating or listing each subgraph occurrence. There are works such as [13] that focus on efficiently doing this explicitly, but that is not our focus.

We can divide the main algorithms that tackle this problem into three main groups related to their conceptual approach. *Network-Centric* algorithms, such as `ESU` [58] or `Kavosh` [25], compute the frequency of all possible k-sized subgraphs in the original network. By contrast, *Subgraph-Centric* algorithms, such as the one by [18], search for one single specific subgraph. Finally, the *Set-Centric* approach of g-tries [48] is conceptually in the middle, allowing for computing the frequency of a customized set of subgraphs.

In this thesis we focus mainly on network-centric approaches. As such, whenever we want to mention the subgraph size of a census, we will always use the variable

k. Note that for this problem, subgraph-centric methods would still be able to do the full enumeration, albeit they would need to search individually for all possible k-sized subgraphs. Likewise, set-centric methods would need to receive as input the same set of all possible k-sized subgraphs, regardless of having no guarantees that all possible subgraph types will appear on the network being analyzed. One possible network-centric approach can be summarized through two major steps: enumeration of connected sets of k nodes and isomorphism tests to determine to which subgraph type each enumerated set belongs to. Classical approaches do this independently: the enumeration part gives origin to sets of k nodes and afterward each one of this sets is inputted into an isomorphism computation (typically by calculating a canonical labeling) so that the correspondent subgraph type frequency can be incremented. This means that the number of performed isomorphism tests is equal to the number of occurrences of subgraphs, even though the actual number of existent subgraph types is generally much smaller.

2.4 Subgraph census algorithms

In the network-centric realm, the two main enumeration algorithms that existed before our work were `ESU` [58] and `Kavosh` [25]. Even though they are conceptually similar, since they both work by iterating through all k-subgraph occurrences incrementally and in the end perform isomorphism tests, they use two different underlying approaches. Although their execution times are usually pretty close, past tests have shown that `Kavosh` performs slightly better on average [25]. An improvement over these approaches is the `QuateXelero` algorithm [27]. It avoids having to do one isomorphism test per occurrence by storing the underlying topology of the subgraphs being enumerated in a quaternary tree. Our own work (which we will present in Chapter 3), `FaSE` [42] displays a similar strategy, and was originally published at the same time as `QuateXelero`. `FaSE` differs from `QuateXelero` because it uses a different underlying topological structure, the g-trie, which is more general in its applicability as we will see.

A different improvement approach is followed by `NetMODE` [34], that considers only very small subgraph sizes and either caches the results of isomorphism tests or builds a customized isomorphism test of a particular subgraph size. In this thesis, we are aiming at a more complete generality, with no rigid restrictions on the subgraphs size.

Regarding subgraph-centric approaches the work of [18] stands out. It works by taking a single subgraph type and computing its frequency on the input network by breaking symmetries. We would like to point out that this approach is conceptually different from the one taken in this work, since a full subgraph census would require a separate computation per subgraph and pre-generated set of subgraphs.

As for the set-centric approach, the state-of-the-art is the usage of g-tries [48]. Like in the subgraph-centric approach, this algorithm makes use of symmetry breaking conditions to enumerate not one, but a set of subgraphs. Note that a data-structure we will present later in this thesis is similar to these g-tries (and that is why we used the same name), however, ours does not use symmetry conditions and is network-centric in its nature, since it does not require a pre-generated set of subgraphs to search for.

Another possible assumption is to only consider certain types of graphs and thus explore specific combinatorial features of that graph type, as was done in [35]. More recently, a series of algorithms have been proposed that combinatorially explore the shared structures of subgraphs and are thus applicable to all graphs. Works like [1, 19, 45], create a series of linear equations based on different properties of the graph (like the degree of each node, the number of triangles that contain a certain node and the frequency of smaller subgraphs) that relate the frequency of each subgraph with all the others. Then, by finding the frequency of a single subgraph (usually the clique), solving the linear system yields the subgraph frequencies for all subgraphs. These methods hitherto are limited by the size of the subgraphs being counted, since the equations are determined manually and these are directly proportional to the number of different isomorphism-wise subgraphs. Our work differs from these types of approaches since it aims at generalness and applicability in all types of graphs.

We will now discuss in some detail the some of the previously mentioned approaches that are more relevant to this work, namely the `ESU` [58], `Kavosh` [25], `QuateXelero` [27] and `ESCAPE` [45] algorithms, as the main representatives of their algorithmic approach.

2.4.1 ESU

The `ESU` algorithm works by enumerating all k-subgraphs of a network and in the end performing an isomorphism test per enumerated occurrence. The enumeration step is thus the most important one and the breakthrough it brought, in relation to the existing approaches prior to its development, was the ability to enumerate all occurrences once and only once.

It keeps two vertex sets, which we will call V_S and V_E. The former represents the subgraph being currently enumerated and since we are enumerating induced subgraphs, we only require a vertex list. The latter is a list of vertices that neighbor any vertex in the current subgraph and can be added to the subgraph being enumerated, that is V_S.

The method repeats the same procedure for each vertex v in the input network G. Initially, it sets $V_S = \{v\}$ and $V_E = N(v)$. Then, for each vertex u in V_E, it removes it from V_E and makes $V_S = V_S \cup \{u\}$, effectively adding it to the subgraph being enumerated and $V_E = V_E \cup \{u \in N_{exc}(u, V_S) : L(u) > L(v)\}$ (where v is the original vertex to be added to V_S as stated in the beginning of the paragraph). The N_{exc} here makes sure we only grow the list of possibilities with vertices not already in V_S and the condition $L(u) > L(v)$ is used to break symmetries, consequently preventing any subgraph from being found twice. This process is done several times until V_S has k elements, which means V_S contains a single occurrence of a k-subgraph.

Since `ESU` works recursively in a set incrementation fashion, it creates an implicit recursion search tree. In each node we consider a certain V_S and V_E representing the partially (or fully if it is a k-subgraph) enumerated subgraph. Figure 2.2 exemplifies this implicit enumeration tree for a 3-subgraph census, where $\boldsymbol{A}$ and $\boldsymbol{B}$ represent the two distinct simple graphs with 3 vertices.

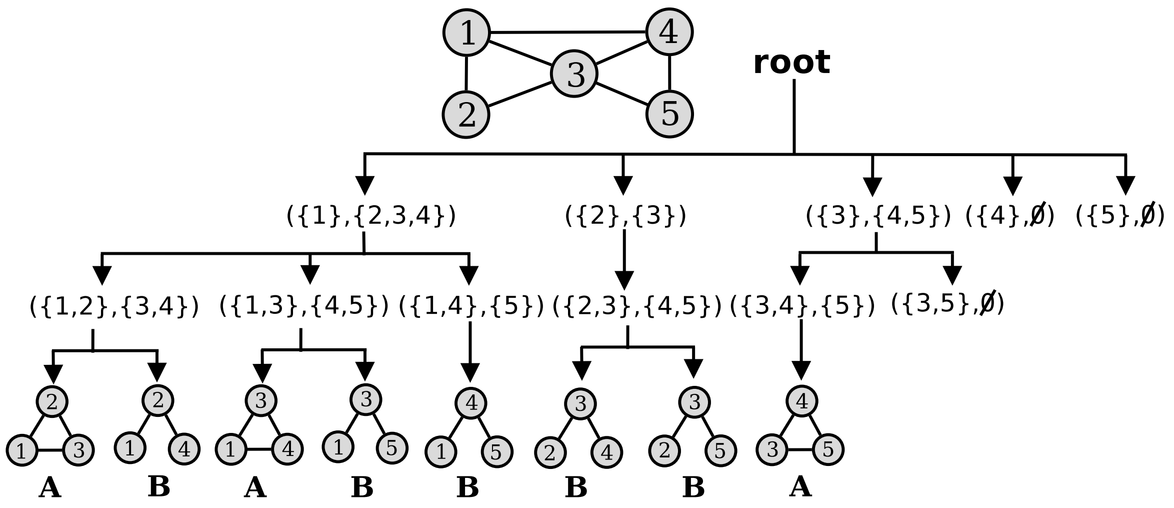


Figure 2.2: An example induced `ESU` search tree leading to eight different 3-subgraphs occurrences.

After the enumeration, the `nauty` [37] algorithm is used for isomorphism testing, so that each occurrence is attributed to the correct isomorphism class and the respective frequency is incremented.

2.4.2 Kavosh

Like `ESU`, the core idea of the `Kavosh` is to find all subgraphs that include a particular vertex, then remove that vertex and continue from there iteratively. It differs, however, because it builds an implicit tree rooted at the chosen vertex (with tree children being network neighbor vertices), and then generates all combinations with the desired number of nodes. For instance, if we are searching for 3-subgraphs, and considering that at the tree root level we can only have one vertex, we could have the combinations with pattern 1-2 (one vertex at root level 0, two vertices at level 1) or with pattern 1-1-1 (one vertex at root level 0, one at level 1 and one at level 2). In an analogous way, 4-subgraphs would lead to patterns 1-1-1-1, 1-1-2, 1-2-1 and 1-3. Figure 2.3 exemplifies this combinatorial search, by showing all patterns emerging from a single root node (the first number of the pattern is omitted, 1-1 represents 1-1-1 and 2 represents 1-2).

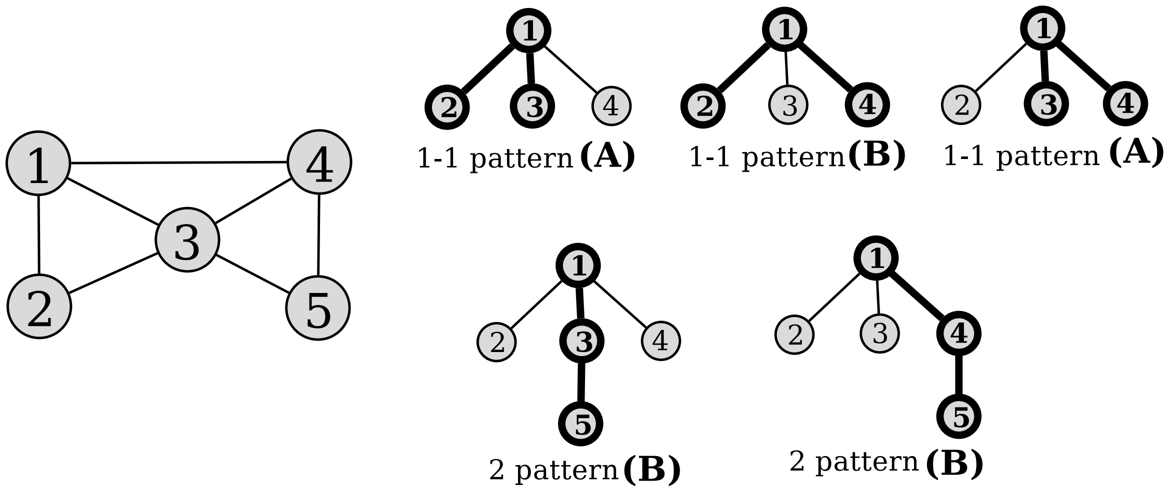


Figure 2.3: `Kavosh` combinatorial search tree starting on node 1 leads to five different 3-subgraphs occurrences.

The combinations are done using a *revolving door* algorithm [30] and, as in `ESU`, the isomorphism detection is done using `nauty` [37]

2.4.3 QuateXelero

The `QuateXelero` algorithm goes a step further in relation to the previous two algorithms. Instead of delaying the isomorphism testing, which is the heaviest bottleneck of the previous two methods, to the end of the enumeration process, it uses the information gathered during the enumeration to reduce the amount of computation needed to determine the isomorphism classes. `QuateXelero` can be built upon any set incremental enumeration algorithm, but to simplify this discussion we will focus on `ESU`, as did the authors of `QuateXelero` in [27].

To assist in this process, `QuateXelero` implements a data structure similar to a quaternary tree. Each node in the tree represents a graph, that can be built by looking into the nodes from the path from it to the root of the tree. Additionally, all graphs represented by a single node belong to the same isomorphism class. Thus, by storing the frequency information in this data structure, `QuateXelero` only requires using `nauty` to compute isomorphism classes once per leaf of the quaternary tree, instead of one per occurrence, like in `ESU`.

To fill the tree, we initially set a pointer to the root of the tree. Whenever a new vertex is added to V_S during the `ESU` procedure, `QuateXelero` looks into the existing edges between the newly added node and the previously existing nodes in V_S and stores its information in the quaternary tree. For each node in V_S, depending on whether there is no edge, an inedge, an outedge or a biedge between it and the newly added vertex, the pointer is assigned to one of its four children, creating it if it was nonexistent.

In Figure 2.4 we show an example of an enumeration of 3-subgraphs using the quaternary tree used by `Quatexerelo`. The figure shows three enumeration steps and the corresponding state of the quaternary tree. In the first quaternary tree we show the meaning of each of the 4 children of each node. Also, each edge has a curved line pointing to its corresponding edge (the dotted line in the final subgraph represents a nonedge).

The technique this method makes use of is very similar to the technique employed by the method we will describe in Section 3.1 of Chapter 3. Both methods were original published at the same time, but the method proposed in Section 3.1 can be considered a generalization of `QuateXelero`, since instead of using a quaternary tree, it uses a more flexible data structure.

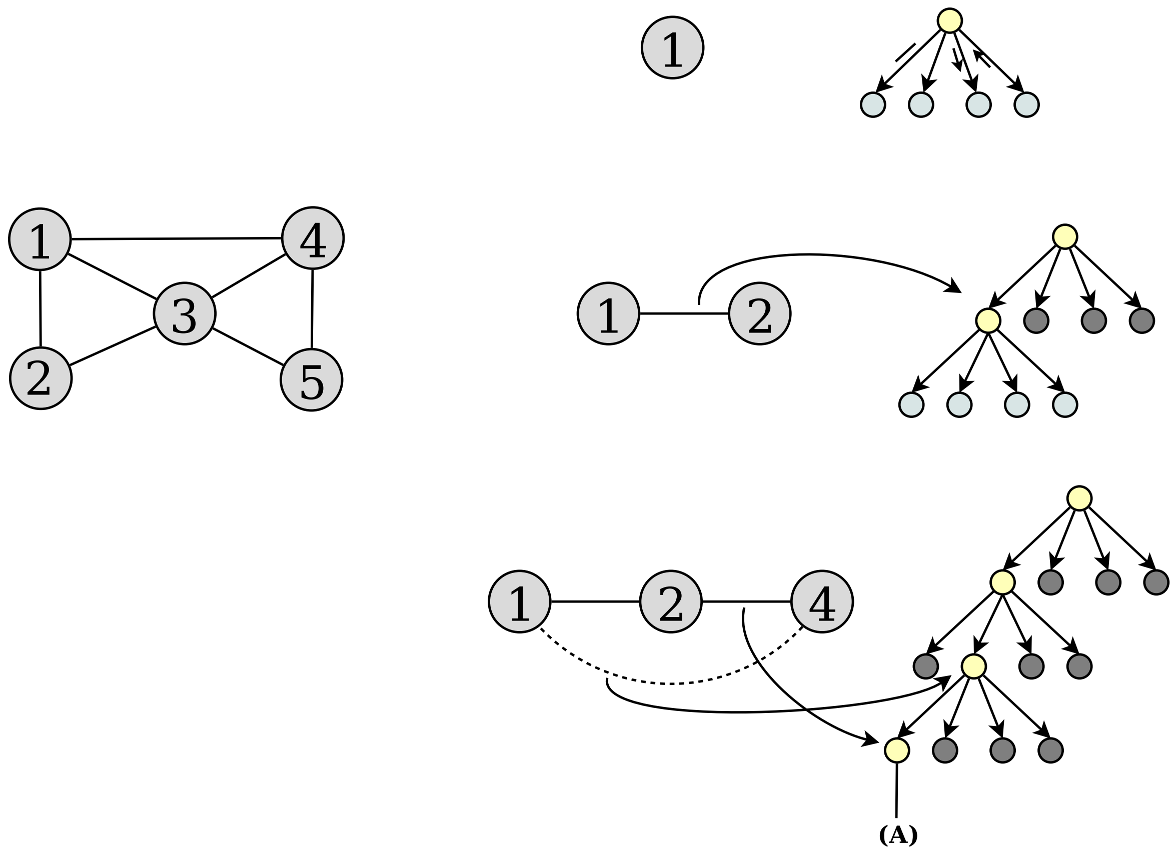


Figure 2.4: Example quaternary tree along with enumeration

2.4.4 ESCAPE

`ESCAPE` is the most recent combinatorially based algorithm to be published and currently stands as the fastest subgraph census algorithm for undirected graphs with k up to 5. Contrary to the previously described algorithms, it directly counts non-induced subgraphs and in the end uses them to compute the induced frequencies.

Its main idea is to breakdown subgraphs into smaller subgraphs and determine a mathematical expression that relates their frequencies. In [45], this is done by establishing a cutting framework that given a cut set of a graph H computes the frequency of H in a larger graph G using only the frequency of the connected components of H produced by removing the vertices of the cut set. Since every graph, except the complete graph, has a cut set, this can be done for all graphs except the complete graph.

Now we only need to select the appropriate cut set for each graph and apply the expression given by the cutting framework. If we do this for all graphs and then count the frequency of cliques separately, using an efficient clique counting algorithm, we have the frequency of all k-subgraphs. Note that applying the expression to all graphs of size k involves manually manipulating the expression, which is too time consuming for $k > 5$ or for directed graphs.

As an example, if we want to count the frequency of the 5-star, the 5 vertex graph star (a graph with a single vertex connected to all other 4 and no other connection), if we choose the center vertex as the only element of the cut set needed to apply the cutting framework, we obtain the following expression for its frequency:

$$\sum_{v \in V(G)} \binom{\delta(v)}{4}$$

2.5 Approximated subgraph census algorithms

We now turn our attention to approximated or sampling based approaches to the subgraph census problem. For these approaches, one of the first to appear was [26], an algorithm that provided a biased estimator by doing a random walk on the network. To correct the bias it calculated the probability to sample each subgraph and used it to weight each sampled subgraph. As an extension of the `ESU` algorithm there exists `Rand-ESU` [58], which works by placing probabilities in each level of the enumeration, thus giving an unbiased estimator for the number of subgraphs of each isomorphism class. We drew upon this idea on the method we discuss on Section 3.2. Another extension of an exact method are `Rand-gtries` [47], which work in a similar fashion to `Rand-Esu`.

`GUISE` [8] works by using a Markov Chain Monte Carlo sampling method. However, it is also more specialized on a specific census, namely undirected subgraphs of sizes 3 to 5. More recently, works like `Moss` [56] focus on applying combinatorial features of the graph to accelerate the search. As in the exact algorithms, there are approximate approaches that are geared only towards certain subgraph types and try to exploit specific properties of those types. For instance, `Fascia` [52] provides an approximate count of non-induced tree-like subgraphs. Our work differs from the last three because right from the start we aim towards total generalization and we support both directed and undirected networks of any size for which we have enough memory to store the subgraph classes.

We will now discuss in some detail some of the previously mentioned approaches that are more relevant to this work, namely the `Rand-ESU` and the `Moss` algorithms [56, 58].

2.5.1 Rand-ESU

The approximate version of `ESU` is very similar to the exact one. The idea behind it is very similar to the one we will present on Section 3.2 of Chapter 3, since the underlying structure of both algorithms is very similar.

For each level of the enumeration tree, the algorithm places a probability of descending, meaning it will only go on exploring that branch with that particular probability. This results in only a fraction of all subgraphs occurrences being enumerated, where each occurrence is sampled with the same probability (we will address and prove this later). Thus, it is possible to have an unbiased estimator for the number of occurrences in each isomorphism class.

In Figure 2.5 we show the same situation as in Figure 2.2, but with probabilities on each depth and some prunings, meaning branches that we discarded randomly, which are marked with a red cross.

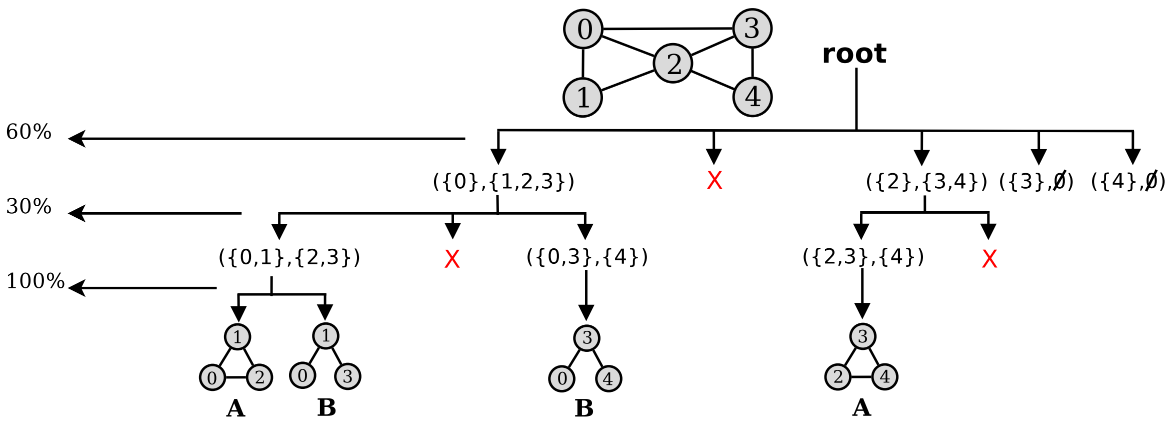


Figure 2.5: Example of `Rand-ESU` induced search tree with prunings

2.5.2 Moss

The main idea of `Moss` is very similar to the core idea of [26], which is to perform some kind of random walk and then correct any surging bias by considering the probability of sampling that subgraph in relation to others with the same size. However, since `Moss` focuses individually on 4 and 5 subgraph census, it is able to obtain more specific and efficient methods.

The idea of sampling 4 or 5 subgraphs is similar: start by obtaining a 4 or 5 size path, where the probability of each step is given by specific probability functions described in [56]. One then manually calculates how each frequency value needs to be corrected, based on the chosen probability function used to perform the random walk. The final result is an efficient $\mathcal{O}(|E(G)| + l \log |V(G)|)$ algorithm to sample l subgraphs of size 4 or 5 (there are some other parameters on the 5 subgraph method).

The main advantage of this method is that it allows us to devise a method that is biased towards certain subgraphs and thus reduce the variance of sampling them. If we want to target low occurring subgraphs, like the clique, this method can reduce the variance of the sampling for normally critical subgraphs.

2.6 Dynamic approaches to subgraph census

The transition of the subgraph census problem to the dynamic realm has only recently started to boom.

One of the earliest works is [29], which suggests a notion of temporal motifs, over-represented temporal patterns in a graph, in order to classify and study certain networks.

Another early notion that surged related to subgraphs in dynamic networks was the investigation of triadic closures, which is the theory of how triangles form in a temporal graph. A known property of many complex networks (especially social networks) is that they have high clustering coefficients, meaning that if a is connected to b and c, there is a high probability that b and c are also connected. In [20], these were investigated in temporal social networks, with the ultimate goal of predicting which links were going to be formed in the future.

In [49], an algorithm was described that counts 4-subgraphs in a network stream, were nodes and edges are changing. This was used to classify streams of networks and predict future behavior. The authors show that this method has a very large speedup when compared to snapshot-based methods.

To investigate the evolution of networks, [41] proposes an edge based metric of temporal motifs in temporal graphs and proposes an algorithm to efficiently count them.

In our work, we focused on two main problems. The first one is very similar to the problem targeted by [49], however, we only consider edge modifications and provide a general method, useful in directed and undirected graphs for multiple subgraph sizes. The second one is a dynamic isomorphism problem, which is a problem we propose as a stream based extension of the canonization problem described in Definition 1.

2.7 Network motif analysis

One of the main applications of a subgraph census is computing network motifs. In a nutshell, these are over-represented patterns in a network, which can be used as a fingerprint or feature vector of the network. This fingerprint can then be used as the representation of a graph in many graph mining tasks, like classification, prediction, etc. As we have seen on the previous sections, many works describe multiple frameworks of these graph mining tasks that have as the core concept the calculation of over-represented subgraphs [9, 22, 59].

This concept of building blocks of networks was first described by Milo et al. in [40] as patterns of inter-connections occurring in numbers that are significantly higher than what one would expect. To simplify notation, we will refer to network motifs simply as motifs.

A determined subgraph is considered significant if its frequency in the original graph is exceptionally high in comparison with its frequency on random networks under a certain null model. To assess exceptionality, one computes the probability that the number of times the subgraph appears on a randomized network is lower than on the original network and then compares it with a certain threshold P. This probability can be estimated using Z-scores on a standard normal distribution, by computing the standardized difference between the observed and expected frequency.

To be classified as a motif, according to the original definition [40], it is also required to fulfill two other properties. For a given subgraph, let f_o be the frequency of the subgraph on the original network and f_r the average frequency of the same subgraph on random networks with an unspecified null model. The first constraint is minimal frequency, that is, f_o has to have a minimum value of U, to ensure a quantitative minimum. The second constraint is minimal deviation, that is, f_o needs to be significantly larger than f_r, to prevent the detection of motifs that have a small difference between these two values but have a narrow distribution in the random networks. This can be stated has $f_o - f_r > D \cdot f_r$, where D is a proportionality threshold.

With this information, we can give a formal definition of motif. Given a set of parameters $\{P, U, D\}$, a subgraph of a given graph is considered a motif if:

- $P(f_r > f_o) \leq P$ (**over-representation**)
- $f_o \geq U$ (**minimal frequency**)
- $f_o - f_r > D \cdot f_r$ (**minimum deviation**)

We can see in Figure 2.6 an example of a plot of motif fingerprints for several networks. Networks with similar sources, for example social, semantic or biological, are grouped together to show that similar source networks have very similar fingerprints, but distant from one another. Its this separation property that allows this method to be used as a preprocessing step for graph mining tasks.

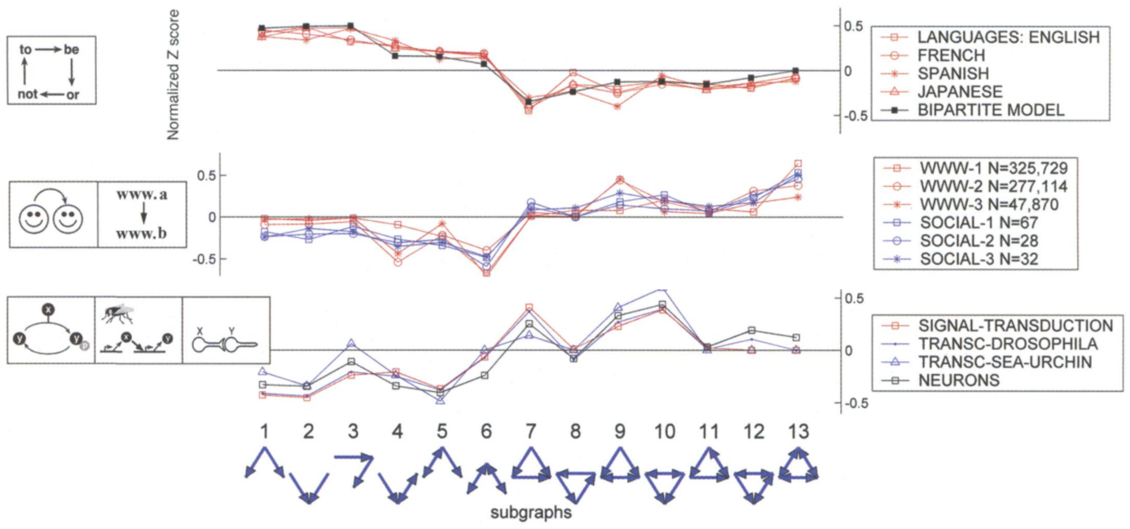


Figure 2.6: Example of motif fingerprints, adapted from [39]

Chapter 3

A Static Approach

In this chapter we will focus on discussing two algorithms with different approaches to the static subgraph census problem. We start by discussing a network-centric exact algorithm that we first presented in [42]. We follow with an approximated version of the former algorithm, that samples the search space in order to estimate the frequency of subgraphs, which we first presented in [43].

3.1 Exact fast subgraph census

To tackle the base subgraph census problem, we try to explore the underlying structure of networks to decrease the amount of computation needed to classify each occurrence in its isomorphism class. The goal is to separate all occurrences in intermediate classes that have two important properties: they can be calculated quickly; each occurrence in the same intermediate class is in the same isomorphism class. Following the complete enumeration of subgraphs in the network it is only necessary to compare a single representative subgraph per intermediate class, hence decreasing the number of isomorphism tests required.

To accomplish that, our algorithm, which we called `FaSE` (from FAst Subgraph Enumeration), is composed of two processes closely integrated with each other: enumeration and encapsulation. The former pertains to the fundamental process of actually finding each individual occurrence of a subgraph in the original network.

This is required to be done by an incremental growth of a connected set of vertices. The encapsulation process is where the isomorphism classes are obtained by storing the topological features of the subgraph. Whenever a vertex is added to the current set of enumerated vertices, we generate a label that describes the relation of the newly added vertex to the already added ones. This corresponds to the partitioning in intermediate classes mentioned above. To actually accomplish this, we use a generic process we called LS-Labeling that categorizes each subgraph's intermediate class. The actual storage of the labels and subgraphs is done using a tree data structure that acts as a customized g-trie in which the LS-Labeling works as the divider, that is, it is responsible by the tree's edges. The following sections describe these techniques thoroughly.

3.1.1 Subgraph enumeration

The enumeration process is not constrained, allowing different approaches. As long as it counts every occurrence of each subgraph once and only once and provided that it does so in an incremental fashion (meaning node by node) any enumeration algorithm can serve as the enumeration step of `FaSE`. The goal here is to enforce that the process transitions from state to state adding a single new node at a time. This permits that each enumerated subgraph is labeled according to the transitions it took to reach the final state.

Consequently, it is possible to use any modern enumeration algorithm. As described in Chapter 2, two of best that accomplish this task are `ESU` [58] and `Kavosh` [25] and they can both be integrated in `FaSE` since they follow the required behavior.

3.1.2 Encapsulating isomorphism information in a tree

As the enumeration process is running, we need to use the information provided by this procedure. The reason to do so is to take advantage of the topology of subgraphs, which in practice is separating the subgraphs into said intermediate classes. Thus, we require a data structure that is adapted to the behavior of the enumeration step, but also compact and benefiting from the common topology given by the labels. A good candidate must follow these parameters and also fit the idea of hierarchical construction of the enumeration. The actual data structure we use is based on an existing data structure called a g-trie [48], which can be thought of as “prefix trees of graphs”, although `FaSE`’s setup is somewhat altered. To avoid ambiguity, throughout the rest of this paper, we will use *nodes* to refer to tree nodes (in our g-trie) and *vertices* to network and subgraph vertices.

3.1.2.1 G-Tries

Our custom g-trie works as a tree whose nodes represent graphs. This is done in an order that respects the topology of the subgraphs, meaning if a certain node is parent of another node, then the graph represented by the former is a subgraph of the latter (in this particular case, with only one additional vertex). Each node stores two pieces of information: a frequency, which is the number of subgraphs of the original network that are of that particular type; a label information regarding its topological structure. The idea is to start off with an empty graph and sequentially add new vertices. For each vertex added, a label that portrays its relation with the previous added vertices is calculated and used to determine its node on the g-trie. Each vertex addition follows a new node on the g-trie. Note that this is a deterministic process, meaning that if the same subgraph is added twice the resulting label is the same. In terms of the g-trie correspondence, the calculated label establishes the node to follow (and the due edge). If this node is nonexistent, both the node and the edge are created. As a result, if two different subgraphs are processed and end up on the same g-trie node, it is assured that they are isomorphic, thanks to the label requirements. An example g-trie can be visualized in Figure 3.1.

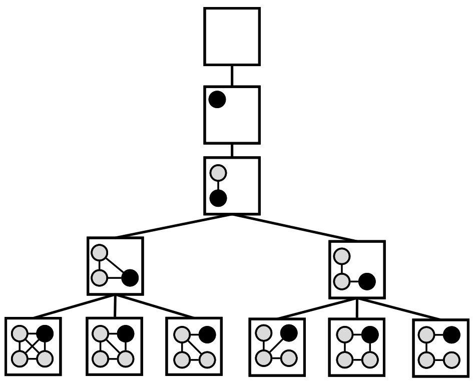


Figure 3.1: An example g-trie with some graphs up to 4 vertices. The dark vertices represent newly added vertices.

Regarding how the g-trie actually accomplishes this, it works by keeping a current node that represents the partial subgraph being enumerated, which is initially the root node (corresponding to the empty graph). It uses two procedures to progress: `Deepen` and `Jump`. The first one inserts a new vertex into the current graph by moving along the g-trie to the corresponding node, a process which lowers the current node ("*deepening*"). Additionally, it creates the new node and edge if they were previously nonexistent and augments the frequency count of that particular node by one. It uses the label generated for the added vertex, which is assigned to a determined edge, to decide where to go in the tree. This is implemented using a prefix tree (or "trie") to ensure linear time search of the new node on the length of the label. Contrary to this, the `Jump` procedure sets the current vertex to its parent, thus going up in the g-trie.

To actually insert graphs into this g-trie, it is possible to take advantage of the common topologies inherent to the enumeration of the subgraphs. Whenever a new vertex is selected by the enumeration process, a labeling algorithm assigns a new label to this vertex in relation to the already selected ones and uses this information to perform a `Deepen` operation on the g-trie. After the recursive call made to enumerate all the subgraphs that exist from the current subgraph, a `Jump` call is performed to go back to the previous node in the g-trie. The reason this works (and why it is done) is because all subgraphs achieved from a particular state (corresponding to a node in the g-trie) will share a common topology related to the partial enumerated set (the state) and therefore share the same label information to that point.

Summarizing the previous paragraphs, it is possible to conclude that this setup is the one of a simple tree regulated by the labels assigned in each step. The consequence of this is that it ends up representing graphs simply because the label is designed in that way. Hence, this is a very general data structure adaptable to different labeling algorithms.

3.1.2.2 LS-Labeling

The generic labeling algorithm is called LS-Labeling. As already mentioned, it is the core of the g-trie and it is also directly related with the branching factor of the tree since it governs the different edges, thus it is associated with both the algorithm running time and the used memory.

It needs to fulfill two conditions: first, that it deterministically partitions the different subgraphs occurrences in classes where no two non-isomorphic subgraphs share the same class; second, that it does so incrementally (emulating the behavior of the enumeration step) using only information regarding the newly added vertex and its relationship with the already added ones. From these conditions one could idealize that this labeling algorithm could simply be a procedure that actually calculated isomorphisms, thus rendering the point of the tree useless. However, as was said in Chapter 2, this is a computationally hard problem and so its use is exactly what we are trying to avoid. Thus it makes sense to ensure another condition: that the algorithm runs in polynomial time. This behavior sets up a trade off regarding the time spent labeling the various subgraphs and the time spent on the actual g-trie (which includes the final isomorphism test time).

We tried two intuitive labeling algorithms which are called the "adjacency list" label and the "adjacency matrix" label, coming from the corresponding graph data structures. When a new vertex is added, the algorithms act on the current subgraph and the vertex to be added. For simplicity we will consider the undirected case first when adding the k-th vertex and then distinguish the directed one. In the case of the adjacency list, the label corresponds to a ordered list of at most $k-1$ integers where the value i ($1 \leq i \leq k$) is present if there is a connection from the newly vertex to the i-th added vertex. Similarly, in the adjacency matrix case a list of $k-1$ boolean values is kept, each one indicating if there is a connection between the newly added vertex and each vertex added before in order of addition, which corresponds to a segment of the actual adjacency matrix of those vertices. We show an example of both labels in Figure 3.2.

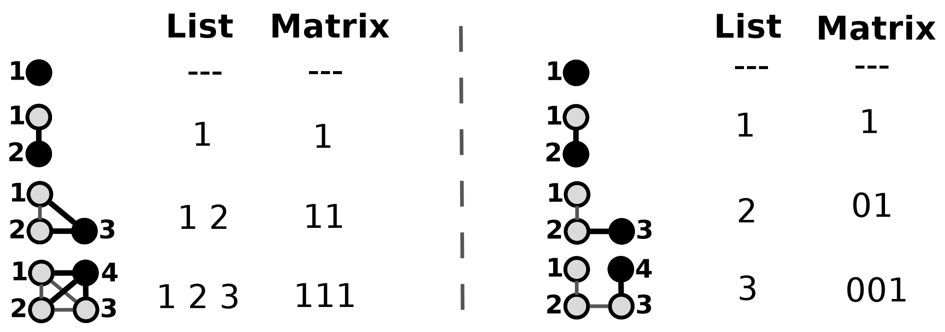


Figure 3.2: Two different valid LS-Labeling schemes on two example graphs. Dark vertices are the ones being added.

To prove the correctness of these two labels options, first notice that they are methodically equivalent and only change the way they represent the information. Thus, to prove its correctness it suffices to show that two subgraphs labeled equally belong to the same isomorphism class. To show that, we need to find a bijection between the two subgraphs. This is achieved simply by following the order in which each vertex was enumerated, which is implicitly represented on the actual label, and map the vertex in each position of the order to one another. Hence, any two subgraphs labeled equally belong to the same isomorphism class and we have our correctness proof.

This method scales pretty easily to the directed case, where instead of just keeping one list, in both cases we keep two, one pertaining to the ingoing connections and the other two the outgoing (in practice a separator value is also used on the adjacency list case to separate the ingoing from the outgoing list).

Having described all components of a g-trie, in Figure 3.3 we show a visual representation of one with the labels associated with each edge using the "adjacency list" label.

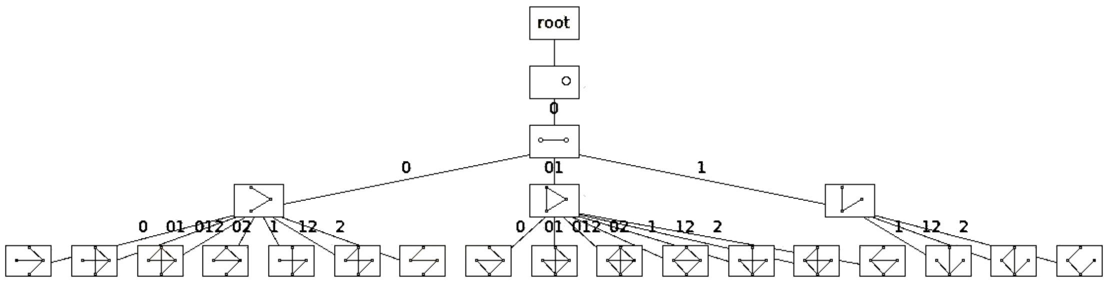


Figure 3.3: An example g-trie with list LS-Labeling after searching for 4-subgraphs.

We are aware that there are more possibilities for this operation that we did not previously address. One of them is what we called the "nth-neighbor" label, which instead of simply considering the connections between the already added vertices and the newly added, also considers the connections from the nodes at distance of a maximum of n from the corresponding vertices. Obviously, these connections augment exponentially and if they are all considered it corresponds to a full isomorphic label. However, a simple "2nd-neighbor" label could, in some cases where the subgraph fingerprint is more heavily populated with certain subgraphs, decrease the run time and memory used. Since this method is not so simple as the previous ones it would probably have greater costs on the general case and thus we did not experiment with it.

Note also that since the LS-Labeling is being used as an intermediate classifier, the g-trie will end up having more leaves than there are different isomorphism classes. This affects the overall run time , since we need to perform an isomorphism test per intermediate class. Although, in the case of both the adjacency list and matrix label, the number of leaves is directly correlated to the different automorphisms of a same graph. Thus it ends up being just a very small fraction of the total number of occurrences in any practical example, as can be seen in detail later on Chapter 5, and so there is a significant gain of computation time. Also, since we only create the nodes we visit, there is a trivial upper bound in the number of leaves that is the total number of occurrences, but it is almost never the case where this upper bound is strict.

We conclude this section by highlighting the flexibility the LS-Labeling generic algorithm displays. Since it only enforces a small number of conditions, it allows for the trade off referred earlier to be adjusted by changing the type of LS-Labeling. Perhaps more importantly, it is adaptable to different formulations of the problem, as was possible to observe with the case of directed graphs. The algorithm is still the same, but the labeling is tuned to suit this particular instance. So it could be easily extended to other problem formulations such as colored graphs, weighted graphs or even multigraphs.

3.1.3 The FaSE algorithm

We present an overview of the whole `FaSE` in Algorithm 1. This incorporates the enumeration step, the g-trie and the LS-Labeling. We use the expression `+=` to denote “increment by a value”.

This algorithm puts together all the discussed parts of `FaSE`. The procedure `EnumerateAll()` iterates through all subgraphs of all sizes up to k, incrementing the counter when the size is k. The frequencies are stored internally by the g-trie. However, since the LS-Labeling does not give the final classes, it is necessary to accumulate the results from each g-trie node and perform an isomorphism test to a representing graph. In the original implementation we do so resorting to `nauty` [37], a third-party efficient

Algorithm 1 The `FaSE` Algorithm

Input: A graph G and a subgraph size k

Result: Frequencies of all k-subgraphs of G

```
1: procedure FaSE($G, k$)
2:     $EnumerateAll(G, k, \emptyset, 0)$
3:     for all $n$ in $GTrie.leaves()$ do
4:         $frequency[CanonicalLabel(n.Graph)]$ += $n.count$
5: procedure EnumerateAll($G, k, S, d$)                ▷ $S$:subgraph; $d$:depth
6:     if $d = k$ then
7:         $GTrie.current.count$ += 1
8:     else
9:         while $nS \leftarrow EnumerateNext(S)$ do
10:            $w \leftarrow nS.NextNode()$
11:            $nL \leftarrow LSLabel(S, w)$
12:            $GTrie.Deepen(nL)$
13:            $nS.Subgraph \leftarrow nS.Subgraph \cup w$
14:            $EnumerateAll(G, k, nS, d + 1)$
15:            $GTrie.Jump()$
```

isomorphism toolkit we mention in Chapter 2, although any algorithms that create a canonical label (that is, a label that represents isomorphism classes) will work.

Note also that in our original implementation (and in any practical implementation) we hard coded the enumeration step into the `EnumerateAll()` function to increase efficiency and explore low level features of the algorithm.

3.2 Approximate fast subgraph census

We now turn to the second algorithm of this chapter and consequently to a different approach to the subgraph census problem. An interesting feature of the `FaSE` algorithm is that it can be adapted to an approximation algorithm to estimate the frequency of each subgraph type in a network by obtaining a sample of subgraphs. It is possible to tune the algorithm to trade accuracy for time, which allows it to be run in a wider range of real networks, which are usually too large for a complete exact enumeration for higher subgraph sizes. The actual method we use is very similar to the one presented in [58] but we will provide our analysis and discussion.

Since each subgraph is enumerated once and only once in the exact version, we can use that to only find a sample of the total number. To do so, we will introduce a probability p_d at each depth d (d varies from 0 to $k-1$, where k is the desired size of the enumerated subgraphs) of the enumeration (which can conceptually be easier to imagine in the g-trie). To clarify the previous sentence, the depth here is the order of the vertex being currently added to the partial set, which is equivalent to the size of the partially enumerated subgraph. The idea is to instead of always processing each newly enumerated vertex (which corresponds to lines `10 - 15` in Algorithm 1), do it with probability p_d at each level d.

We can easily observe that the probability of a particular subgraph on the network being sampled is the probability of the first vertex being chosen (at level 0) which is p_0, times the probability of the second vertex being chosen and so on, which equals $\prod_{0 \leq d < k} p_d$. We will call this value the *sampling percentage*, and denote it as: $\eta = \prod_{0 \leq d < k} p_d$.

We denote the total number of subgraphs of size k (the leaves in the induced `ESU` search tree) in graph G by $T(G)$. It is possible to show that the average number of sampled subgraphs is $\eta \cdot T(G)$. To do so, first note that each k-subgraph has the same probability of η of being sampled. Since there are $T(G)$ subgraphs and each one has the same probability of η of being sampled, the average number of sampled subgraphs is $\eta \cdot T(G)$.

We will call $F_{sample}(G_k, G)$ to the frequency of subgraphs of G sampled by the algorithm that are from the same isomorphism class as G_k. This definition allows us to define an estimator for the value of $F(G_k, G)$ as follows:

$$\widehat{F}(G_k, G) = \frac{F_{sample}(G_k, G)}{\eta}$$

Note that, since all the isomorphism classes are disjoint, to obtain an estimator for the total number of subgraphs it suffices to sum all the $\widehat{F}(G_k, G)$, one per different isomorphism class.

3.2.1 Uniform sampling

To start the theoretical discussion of the approximation, we will first prove the estimator is an unbiased estimator. To do so, observe that since the probability of sampling each subgraph is the same, η, the expected value of $F_{sample}(G_k, G)$ is simply $\eta \cdot F(G_k, G)$.

To calculate the expected value of $\widehat{F}(G_k, G)$, we observe that since the expected value is a linear operator, this corresponds to the previously calculated value divided by η. Plugging this into the formula of the estimator gives:

$$\mathbb{E}(\widehat{F}(G_k, G)) = \frac{\mathbb{E}(F_{sample}(G_k, G))}{\eta} = F(G_k, G)$$

Thus we conclude that $\widehat{F}(G_k, G)$ is an unbiased estimator for $F(G_k, G)$.

Using this information, Algorithm 2 shows the adapted algorithm, which from now on we will call `Rand-FaSE` to distinguish from the exact version of `FaSE`.

Note that in all practical implementations the actual probability call should be hard coded, since it can prevent some unneeded work done in the `EnumerateNext()` function.

3.2.2 Performance analysis

Before we conclude, we will reason about the variance of the estimator and how the choice of each individual value of p_d affects it and thus the quality of the estimation.

First of all, notice that the number of subgraphs sampled of a certain type depends on the structure of the enumeration tree. If it were perfectly balanced and each subgraph type were evenly distributed along the tree, then the individual values would not matter but only their product (what we called the sampling percentage). However, this is not the case in any of the presented enumeration algorithms. Even though for instance the ESU enumeration tree is naturally skewed, since it enforces an order on the enumeration, it is highly unlikely that any algorithm generates a balanced enumeration tree since this is very input dependent.

Since the enumeration tree is not balanced, the choice of parameters influences the quality of the sample and run time. If lower values for p_d are chosen for levels of

Algorithm 2 The Rand-FaSE Algorithm

Input: A graph G and a subgraph size k

Result: Frequencies of all k-subgraphs of G

```
1: procedure FaSE($G, k$)
2:     $EnumerateAll(G, k, \emptyset, 0)$
3:     for all $n$ in $GTrie.leaves()$ do
4:         $frequency[CanonicalLabel(n.Graph)]$ += $n.count$
5: procedure EnumerateAll($G, k, S, d$)                ▷ $S$:subgraph; $d$:depth
6:     if $d = k$ then
7:         $GTrie.current.count$ += 1
8:     else
9:         while $nS \leftarrow EnumerateNext(S)$ do
10:            with probability $p_d$ do
11:                $w \leftarrow nS.NextNode()$
12:                $nL \leftarrow LSLabel(S, w)$
13:                $GTrie.Deepen(nL)$
14:                $nS.Subgraph \leftarrow nS.Subgraph \cup w$
15:                $EnumerateAll(G, k, nS, d + 1)$
16:                $GTrie.Jump()$
17:
```

the tree nearer to the root, this will increase the variance of the results, since it is possible to branch out a sub-tree with more occurrences of a certain type. However, the run time of the algorithm is decreased in exchange for the increase of variance. This decrease is two-fold: on one hand, the amount of subgraphs sampled has a higher variance, which results in fluctuations in run time; on the other hand, since a subgraph that is not going to be sampled is pruned earlier in the tree, we can avoid most work on its partial enumeration, which is costly since it involves traversing the g-trie, generating its label through the LS-Labeling and doing the actual enumeration.

A consequence of the unbalance of the enumeration tree is that even if given the values for p_d, calculating the variance is hard since it is highly dependent on the input network. It is possible to draw some conclusions though, the most important one being that the variance is higher in relative value for lower $F(G_k, G)$ values. To explain this recall that the average number of sampled subgraphs in $S(G_k, G)$ is: $\eta \cdot F(G_k, G)$. When this value is small (specially when it approaches 1 or is smaller than 1) since the number of sampled subgraphs is a discrete quantity, the actual value of $F_{sample}(G_k, G)$ is going to be rounded down or up. This means the variance will be higher in relative value, since for high values of $F(G_k, G)$ the continuous approach is a good approximation.

There are ways of decreasing the variance while keeping the estimator unbiased. In [58], the author suggests that, instead of simply continuing with a certain probability, from a node (of the enumeration tree) with x children at depth d randomly choose $x' = \lceil x \cdot p_d \rceil$ with probability $x \cdot p_d - \lfloor x \cdot p_d \rfloor$ or choose $\lfloor x \cdot p_d \rfloor$ with probability $(1-(x \cdot p_d - \lfloor x \cdot p_d \rfloor))$. The idea is to choose a fixed number of children instead of taking each one with a certain probability, ensuring that there is always a collection of nodes that will be followed. The author also showed that this leads to a lower variance. In our implementation, which we will discuss on Chapter 5, we did not include this because even though this improves the quality of the sample on average, for lower values of $F(G_k, G)$ it usually results in a decrease, particularly when $\lceil x \cdot p_d \rceil$ rounds to 0, where depending on the input network, the algorithm would not sample any subgraphs of a certain isomorphism class.

3.2.3 Further discussion

To conclude the discussion about sampling, we will mention two important aspects regarding the sampling's application and how to improve it.

Naturally, the main purpose of doing a sample in place of a full enumeration is to use it in inputs that would take too much time to calculate using the exact approach. On these cases are included networks with a high number of vertices and edges. Therefore, the data structure used to represent the network can not be a simple adjacency matrix, since it would draw too much memory and thus would be unfeasible. The obvious substitute is an adjacency list, but due to the fact that `FaSE` requires a way of knowing if two certain vertices are connected (in the LS-Labeling and in the isomorphism test), the adjacency list will hurt time performance compared with the simple matrix (which implements this operation in constant time). There are multiple ways of targeting this issue and although it is a rather well known problem in general, we provided an in depth study of the problem applied to subgraph census in [44], but it is outside the scope of this thesis.

Another aspect that could improve the quality of the sample would be to automatize the choice of the individual probabilities p_d. This could be achieved through an adaptive sample, that would start out with very low parameters and over multiple runs only explore the enumeration tree where needed. This works since for high values of $F(G_k, G)$ the estimator result converges rather quickly whereas for lower values it does not. So exploring this could significantly improve the sample.

3.3 Other approaches

We have now discussed the two approaches of interest in this chapter, however, we will complement them with a brief note.

As was mentioned in Chapter 2, there is a possibility of extending a lot of the know methods to different environments. A well-known one that can be applied to the `FaSE` algorithm is parallelism. With that in mind, in [3] we provided a shared memory parallel version of `FaSE` and tested its performance and efficiency.

Another variation we worked on will be extensively studied on the next chapter, namely focusing on temporal or dynamic graphs.

Chapter 4

A Dynamic Approach

We now turn to a different focus: subgraph census on temporal or dynamic networks. Unlike in the static case, where the problem is very well defined, in the dynamic case there are multiple definitions available depending on the goal of the census. In this chapter we will discuss two different methods that perform two very different tasks.

Firstly, we look into updating the frequency results of a census. Given a network and a series of edge modifications, what is the frequency of subgraphs after each modification. Afterwards, we look into a more general problem: the dynamic graph isomorphism problem. Given a network and a series of edge modifications, what is the resulting graph's isomorphism class. We will also argue why this problem is relevant to the subgraph census problem by providing a brief case study.

4.1 Updating a census

Before we turn to the algorithm, we give a formal definition of the problem we aim to solve:

Definition 3 (Updating Subgraph Census Problem)**.** *Given an integer k, a graph G and a graph stream S with step t, determine the frequency of all connected induced k-subgraphs of $S(G)$.*

Given this definition, note that a trivial method would simply compute the frequency of k-subgraphs for each element of $S(G)$ independently. However, this is inefficient and can be improved by only updating what is actually changed by each graph changing operation. Thus, we will alter the `FaSE` algorithm, described in the previous chapter, to only count subgraphs that touch the edge given by a graph changing operation.

Our method to efficiently update frequency counts works by altering the enumeration algorithm to count frequencies starting on edges. When adding an edge, the algorithm first counts all subgraphs that use the edge's two ends and decrements their frequency. Afterwards, it adds the new edge and counts all subgraphs that touch that edge. To remove an edge we do an analogous process. Our method is based on the `ESU` enumeration algorithm, altering it to start on a given edge.

For a given edge to add (a, b), the algorithm first considers as initial sets $V_S = \{a, b\}$ and $V_E = (N(a) \cup N(b)) \setminus \{a, b\}$ and only uses these as initial sets (meaning it does not recurse on other initial V_S and V_E). The rest of the procedure is similar to the original `ESU` algorithm, but the symmetry breaking is removed, that is, when adding a node u' to V_E, there is no comparison with a: if u' belongs to $N_{exc}(u, V_s)$ it will be added to V_E.

In Figure 4.1 we show which subgraphs touch two distinct edges (showed in dotted red) of the same graph, using the method based on our updated enumeration method. The sets for each graph represent the initial V_S and V_E as in the previous discussion.

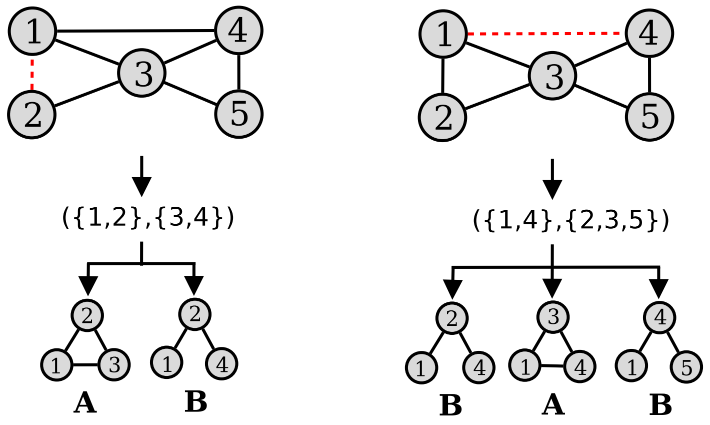


Figure 4.1: Subgraphs that touch two edges

To prove that this method is correct we use the original correction proof of the `ESU` algorithm. If a is the minimal node of the graph (that is, for every node v, $L(a) \leq L(v)$), all subgraphs that include a will be enumerated on the first iteration of the algorithm. For that iteration, if b is the first element of N_{ext}, then it will be removed and the next iteration has $V_S = \{a, b\}$ and $V_E = (N(a) \setminus \{b\}) \cup N_{exc}(b, \{a\}) = (N(a) \cup N(b)) \setminus \{a, b\}$. Since this is the only recursion path that will include a and b (since b was the first node to be removed from the initial N_{ext}), all subgraphs that contain a and b will be counted on this recursive subtree. Since this is analogous to our method, its correctness implies the correctness of our method.

A short overview of the whole method in Algorithm 3. The parameter *increment* indicates the enumeration method if it should increase or decrease the frequency of the found subgraphs.

Algorithm 3 The Update `FaSE` Algorithm

Input: An edge e and a graph G

Result: The updated frequencies of subgraphs

```
 1: procedure AddEdge($G, e = (a, b)$)
 2:     $EnumerateEdge(G, a, b, -1)$        ▷ We assume there is no current edge $(a, b)$
 3:     $AddEdgeToGraph(G, e)$
 4:     $EnumerateEdge(G, a, b, 1)$
 5: procedure RemoveEdge($G, e = (a, b)$)
 6:     $EnumerateEdge(G, a, b, -1)$        ▷ We assume there is a current edge $(a, b)$
 7:     $RemoveEdgeFromGraph(G, e)$
 8:     $EnumerateEdge(G, a, b, 1)$
 9: procedure EnumerateEdge($G, a, b, increment$)
10:     $RunESU(G, \{a, b\}, (N(a) \cup N(b)) \setminus \{a, b\}, increment)$
```

4.1.1 Random graphs with given subgraph frequencies

The method above was first introduced in [51] as a subprocedure of a method that generated random graphs with fixed subgraph frequencies. This method is given as input a set of subgraph frequencies and outputs a graph that matches these subgraph frequencies up to a certain given percentage.

The method works by applying an optimization algorithm based on simulated annealing [28]. It sets up an optimization scheme that will randomly perturb an initially randomly generated network (using any other random graph model) and accept or reject each change according to how far way from the preferred subgraph frequencies the resulting graph is. To govern the accept/reject method, a decreasing temperature value is used to provide an acceptance probability function. The step of interest here is how the random perturbations are performed and how to update the subgraph frequencies after a perturbation.

In [51], two Markov chain edge swap methods were used, which were based on selecting edges from the graph and swapping them (connecting their opposing incident vertices). This operation is equivalent to removing two edges and adding two new ones, thus we can apply the method described in this section four times to update the subgraph frequencies. We will show the improvements this technique allowed in Chapter 5.

4.2 Dynamic graph isomorphism

We now turn to the second part of this chapter, where we look into a more general problem: the dynamic graph isomorphism problem. We will look into how this can be applied to a temporal subgraph census in the end of this section, but for now we first formally define the problem we are tackling.

Definition 4. *In the dynamic canonization problem we are given a graph G with n vertices and a graph stream S of cardinality n, and we are asked to provide a canonical representation for each graph in $S(G)$.*

Note that, with this formulation, we fix the number of vertices and only vary the edge set.

We propose a method to approach this problem that is focused on the isomorphism of small graphs (we will better define how small later in this section), which are in the usual range of subgraph sizes the subgraph census problem applies to.

Our method explores the dimension of the total number of graphs of a certain size to build a data structure that compresses the relationship between their topologies. This data structure is analogous to a deterministic finite automaton (a finite-state machine), where each node represents a different graph and transitions represent additions or deletions of edges. The result is an algorithm that solves the dynamic canonization problem in an online fashion. We will first describe how the automaton works and how to use it, then we follow up with how to build the automaton efficiently.

To avoid ambiguities, we use "node" and "transition" to refer to properties of the automaton and "vertex" and "edge" to refer to properties of the graphs represented by the automaton.

4.2.1 The automaton

As mentioned above, we use a data structure that is analogous to an automaton to support our algorithm. This will be used as we iterate through each graph in $S(G)$ to follow the isomorphism graph class.

For our purposes, an automaton is a finite-state machine, defined as a set S of states, with an initial state s_0, and a set of transitions $T : S \to S$. We will augment each transition with other information useful for our method, but fundamentally this is all we need to define our automaton. We will refer to nodes of the automaton by bold uppercase letters, like $\boldsymbol{A}$.

The node set of the automaton represents the different isomorphism graph classes of a fixed number of vertices n. For each different class, we fix one label function and associate to it a single node of the automaton. This equates to fixing a permutation per isomorphism class and using it as a canonical labelling. For each node, there is one transition coming out of it per possible pair of two vertices of the underlying graph. Each one of this transitions represents an edge toggle, meaning an addition or removal of an edge (which can be written as a pair of graph labels) to the represented graph, which depend on whether the two vertices of this transition are connected or not on the represented graph. Thus, the destination of each transition is the node whose isomorphism graph class is the one of the altered graph. We portray a pictorial representation of this object in Figure 4.2.

Since every change between two consequent graphs in $S(G)$ is described by a single pair of vertices it is natural to use the described automaton to follow the isomorphism graph class of each graph by walking through the automaton. On each step, we use the transition that is associated with the pair of vertices on the current graph changing operation. Initially, the automaton starts on the node that represents the empty

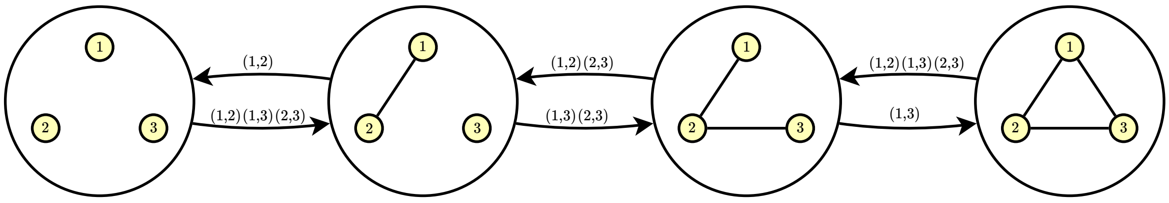


Figure 4.2: An automaton representing undirected size 3 graphs

graph with n vertices. To find the node that represents G, we build G by following all transitions that represent the pairs of vertices on each edge of the graph, in any order. Subsequently, each graph changing operation results in following one transition.

However, this is not enough to actually apply the automaton, since the order of vertices that was fixed on a certain node may not be the same as the one the current graph we are considering from $S(G)$. Thus, we keep a permutation π_p that tells us how to change the order of vertices of the current graph in order to have the same graph as the one the current node represents. If we think about labels, let G^c be the current graph and G^n be the graph represented by the current node (by definition we have $G^c \cong G^n$), π_p has the following property: $L(G^c) \circ \pi_p = L(G^n)$, since the label function works like a permutation from vertices to indices and taking $\circ$ as regular permutation composition.

To accommodate this change, we also need to update how the transitions work, since after following a transition the relation between the current graph and the graph represented by the current node may change. Thus, we associate a permutation with each transition that informs on how to update π_p. If the permutation for a certain transition is p, then the new π'_p is obtained by $\pi'_p = \pi_p \circ p$. Initially, π_p is set to the identity permutation, since the initial node represents the empty graph (where every permutation is valid). Note that in Figure 4.2 the permutations were omitted for brevity.

The resulting automaton represents all different graphs of size n and can be used to keep track of the canonical representation of a graph after each vertex pair change by following a transition and composing a permutation. If we are applying a change of vertex pair (a, b), note that we follow the transition related to (π_p^a, π_p^b), since we always work on top of the representation the automaton gives.

4.2.2 Building the automaton

Now that we have described how the automaton works and how to use it, we will specify how to build it. There are two important aspects here that not only heavily influence the complexity of the building process, but also the complexity of using the automaton. The first is how to fix the graph each node represents. The second is when to build the automaton, since we can pre-build it or build it as we process the graph stream. We will answer the first question through the following explanation and also point out why it is a relevant question. Regarding the second, we will first describe an on-the-fly method and then a method that pre-builds the automaton but leads to a more efficient representation.

4.2.2.1 On-the-fly method

In order to fix a canonical order for each node, we use the representation `nauty` provides, since running it returns an adjacency matrix, which can serve as the canonical order. Our method to dynamically build the automaton is based on following the supposed transitions as the stream is processed. Whenever we find ourselves on a non existing node, we run `nauty` to know where we should be and either create a new node or point the transition to the correct destination. Additionally, we fill the transition permutations accordingly.

The only node we pre-build is the node that represents the empty graph. Afterwards, we will process each new vertex pair (a, b). Let $a_p = \pi_p^a$ and $b_p = \pi_p^b$. On processing a new pair, we first check if the transition of (a_p, b_p) was already created. If not, we first run `nauty` on the transformed graph, that is, if G is the current graph after adding or removing the edge induced by (a, b), we do so on G' where $L(G') = L(G) \circ \pi_p$, meaning the graph from the current node altered by the pair (a_p, b_p). We do so because `nauty` not only returns the canonical adjacency matrix that we will use to represent the automaton node, but also a permutation P that transforms the graph represented by the canonical adjacency matrix into G'. We can then create a new transition by (a_p, b_p) from the current node $\boldsymbol{C}$ to the new node $\boldsymbol{N}$ (found with `nauty`) with permutation $\overline{p}$, since this permutation transforms the graph on $\boldsymbol{C}$ with added vertex pair (a_p, b_p) into the graph on $\boldsymbol{N}$, which is the same that `nauty` returns.

This was implicit in the previous paragraph, but we also need a bookkeeping mechanism to store the node representations, so as to avert having a duplicated node representing the same graph class. This can be done using a dictionary data structure that maps canonical representations, as obtained through `nauty`, to automaton nodes (if they exist). Since the graph representation is fixed by the `nauty` canonical representation, the method described in the previous paragraph is exactly the same whether the destination node ($\boldsymbol{N}$, in the previous paragraph's notation) has to be created or not. If the node is missing, we simply create a new node and feed it to the bookkeeping dictionary.

Using the previous paragraphs notation, when processing a change (a, b), let p be the permutation `nauty` returns, $\boldsymbol{C}$ be the initial automaton node and $\boldsymbol{N}$ the destination node. Since $\overline{p}$ transforms graphs in the $\boldsymbol{C}$ representation to the $\boldsymbol{N}$ representation, the converse is also true, that is, P transforms graphs in the $\boldsymbol{N}$ representation to the $\boldsymbol{C}$. Thus, we can use this information to right away fill another transition, p, from $\boldsymbol{N}$ to $\boldsymbol{C}$. However, since the representation changed, the vertex edge associated with this transition is not (a_p, b_p) but rather $(\overline{p}^{a_p}, \overline{p}^{b_p})$, since this is the corresponding edge pair in $\boldsymbol{N}$.

It is important to note that the real temporal bottleneck of using this automaton lies on the application step rather than the building step, as we will observe in Chapter 5. This means that the advantage of using a dynamic building method is only observable if the full automaton is impossible to be generated. For example, if we are applying the method in an instance graph with a high number of vertices, but where we know the total number of different graph types in the stream is low, using the dynamic building method we only build a partial automaton, according to the input.

Consequently, it is useful to optimise the automaton underlying representation and methodology if this improves the runtime of applying, even if it worsens the building procedure. With this in mind, we can compress the permutations associated with each transition in order to avoid iterating n integers. By observing the different canonical representations given by `nauty`, one can observe that they are fairly regular, meaning that often if two graphs differ by a single edge, their adjacency matrix only differ in one (or two, in the undirected case) entries. This implies that the permutation associated with the transition between the two is simple, often either the identity or a single transposition. Thus, we can compress these cases to a special representation that instead of composing a permutation with π_p, either does nothing or simply swaps two entries of π_p. We will see a detailed analysis of the effect of this in Chapter 5, but theoretically this would lower the complexity of following a transition.

4.2.2.2 Pre-building method

There are not many points to improve related to the actual on-the-fly building process, since this method does the bare minimum to know where each transition leads to. Consequently, our pre-building method works very similarly to the on-the-fly method, but it does a depth-first search on the automaton in the beginning, in order to generate all possible nodes and transitions.

However, the advantage of doing a method that pre-computes the automaton is that it is easier to fix a different representation of graphs per node, since there is no need to follow the canonical representation given by `nauty` (or to have one that works regardless of the order with which we build the automaton). This is important since changing the underlying representation changes the permutations associated with each edge and this has a direct effect on their compressibility and thusly on the runtime.

It is easy to prove that composing a permutation to the graph of each node does not change the correctness of the algorithm, as long as we update the transitions accordingly, since we are simply projecting the automaton to a different space. Hence, it is easy to change the underlying representation of each node by composing a permutation to the permutation `nauty` returns during the create new transition procedure, as long as we compose the same permutation to each transition coming into the same node. In practice, we are simply changing the representation given by `nauty` to one that better suits our goals.

All that is left is to choose which permutations to compose with. Instead of focusing on individual permutations, one can determine the underlying representation and choose the permutation that generates this representation. To choose a representation, we can choose a building order, that is, choose the order under which we initially traverse the automaton to pre-build it and use the first graph to touch each node as its representation. To implement this, the permutation we compose with each node is simply $\overline{p}$ (borrowing from the previous subsection's notation), where we fix the permutation p obtained on the first time we visit that node (which is when we actually create the node). This results in choosing the identity permutation as the permutation from $\boldsymbol{C}$ to $\boldsymbol{N}$ on the first visit to the node.

Different orders were tested, with the goal of increasing the percentage of transitions whose permutation was either the identity permutation or a single transposition. It would be possible to implement an optimisation algorithm here, like a local search algorithm, that would repeatedly perturb the traversal order. Although, this would be computationally heavy and would probably not yield much better results than a simply greedy approach. Consequently, we chose an altered edge lexicographical order, that is, we first follow all pairs that create edges before any pair that removes edges and we break ties choosing the lexicographical first transition vertex pairs. We tested different approaches, but this one yielded the better results. In Chapter 5 we will show more details regarding its performance.

Note that, for graphs with 4 or more vertices, it is impossible to build an automaton where each transition permutation is either the identity permutation or a single transposition. Note that this is equivalent to saying the adjacency matrix of graphs in two adjacent nodes (adjacent in the sense there is a transition between both) differ by only one (or two, if the graph is undirected) entries, which means they differ by at most one edge. To prove the impossibility premise, we will assume that it is possible to build an automaton for 4 vertex undirected graphs. Consider Figure 4.3, which represents a partially constructed automaton for 4 vertex graphs. We omit the multiple transitions between nodes and simply fix each node's graph representation and show the relationships between nodes (where one or more transitions would be present).

In this example, there is a mismatch between nodes $\boldsymbol{G}$ and $\boldsymbol{H}$, since their graph's adjacency matrices differs by more than an edge. We can show that this example is "canonical", meaning that all possible automata for 4 vertex graphs are equivalent to this example. Node $\boldsymbol{A}$ is always the same, by definition. For node $\boldsymbol{B}$, since all one edge graphs of 4 vertices are isomorphic, then the choice of which two vertices to connect is irrelevant. In node $\boldsymbol{C}$ we could have multiple representations. After fixing node $\boldsymbol{C}$, nodes $\boldsymbol{D}$ and $\boldsymbol{E}$ are also fixed, since we can only add one edge to get that isomorphism class. From nodes $\boldsymbol{D}$ and $\boldsymbol{E}$ we also fix node $\boldsymbol{H}$, since it needs to have all edges $\boldsymbol{D}$ and $\boldsymbol{E}$ have. Node $\boldsymbol{F}$ is also fixed by fixing node $\boldsymbol{B}$, since this is the only edge we can add to get that isomorphism class. Node $\boldsymbol{G}$ is fixed if we know $\boldsymbol{C}$ and $\boldsymbol{F}$. The only free parameter here is choosing node $\boldsymbol{C}$, but notice that regardless of which extra vertex

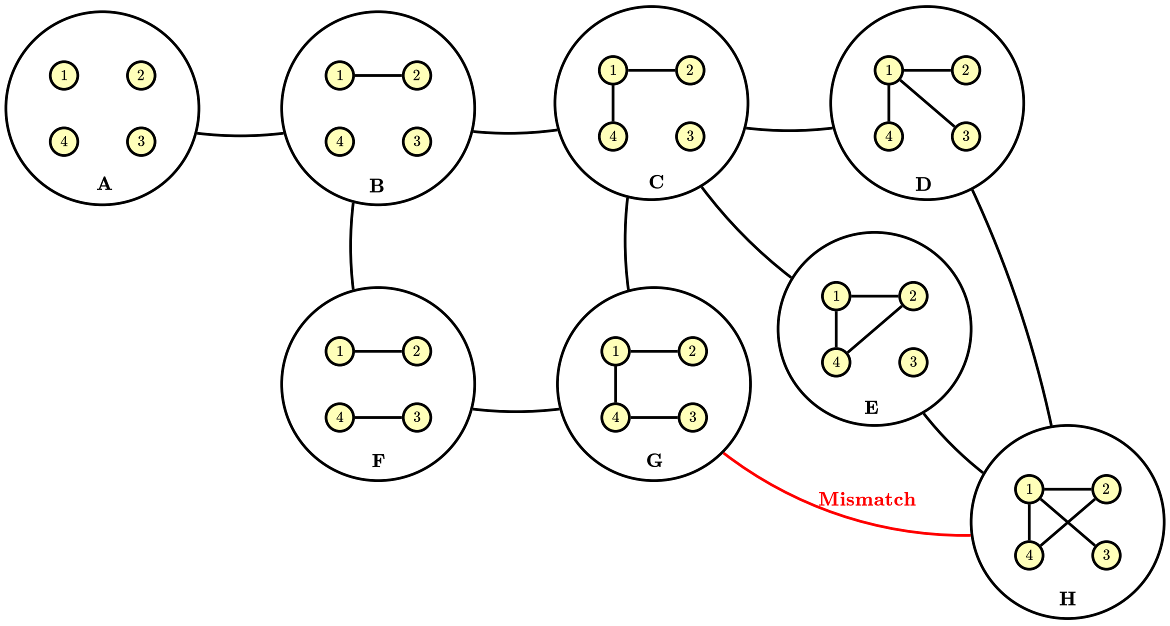


Figure 4.3: A partial automaton representing undirected size 4 graphs

pair we choose to connect, the adjacency matrices of nodes $\boldsymbol{G}$ and $\boldsymbol{H}$ always differ by more than one edge. Thus, this contradicts the assumption that it is possible to build an automaton for 4 vertex undirected graphs where all the graphs of all adjacent nodes differ by at most one edge. Since the 4 vertex automaton is a subset of the n vertex automaton for $n \geq 4$ (we can add extra vertices that are not connected to any other vertex), this proves the nonexistence of an automaton where each transition is either the identity permutation or a transposition.

4.2.3 Optimising the automaton

In this section we will discuss a few optimisations to improve the automaton. We will start by a few notes on the representation of the various elements of the automaton and finish with a method to optimise the building method and the compressibility of permutations.

Since we are working with a low number of vertices, the representation of permutations can be trivially compressed to a 64 bit integer, by assigning 4 bits to each element of the permutation. Additionally, implementing the transposition compression can be done by simply fixing the first two bits to different values to represent either a full permutation, the identity function or a transposition. Representing automaton nodes as pointers also proved more efficient than using an array of nodes, since following a transition is simply assigning a variable representing the current node and requires no look up into memory. Finally, we use an array to represent the automaton transitions by indexing each vertex pair to an integer from 1 to the number of possible vertex pairs.

We now turn to a more interesting discussion regarding optimising the actual methods. First of all, it is easy to observe that we can use multiple possible permutations to represent the transition between some nodes. If we think about the 3 vertex undirected automaton (like the one in Figure 4.2), the transition permutation between the initial node and the node with a single edge between vertices 1 and 2 can either be the identity permutation or the transposition between 1 and 2. This works because, in this case, 1 and 2 share the same orbit in the destination graph. We can use this fact to improve the way we build the automaton and to increase the compressibility of transition permutations. Given any transition permutation π to a certain node $\boldsymbol{N}$, if a set of vertices $\{v_1, v_2, \ldots, v_k\}$ have the same orbit, then we can interchange their position in π to obtain another valid transition permutation. If we try to greedily reorder their positions to decrease the number of transpositions in π, we will obtain another valid transition permutation that may be compressible by our previous metric. This does not entail any extra work, since `nauty` also returns the orbits of the objective graph.

Another optimisation made possible by the use of orbits is obtainable if we notice that if adding (or removing) an edge between two vertices results in a graph G, then adding (or removing) an edge between any two other vertices with the same orbits (excluding self loops, in case the two vertices have the same orbit) results in a graph G' where $G \cong G'$. With this, after completing a transition permutation for vertex pair (a, b) (as described in the on-the-fly algorithm from Section 4.2.2) we can guess permutations for other vertex pairs (a', b') that have the same orbits as the original pair by considering the original permutation and swapping a with a' and b with b'. This does not generate a correct permutation all the times, since we could have permuted the rest of the topology (for example, if other orbit equivalent vertices are permuted), but we can check its correctness by comparing the adjacency matrices of the objective graph with the initial graph after applying the guessed permutation. Empiric tests showed we guess the correct permutation at least 10% of the time.

4.2.4 Theoretical analysis

Here we discuss some important theoretical considerations regarding the spacial and temporal complexity of our automaton method and its variants.

Let $\mathcal{G}_n$ denote the set of different graphs with n vertices (note this is an agnostic analysis, since it works for both undirected and directed graphs). Let E_n be the maximum number of edges for a graph with n vertices, that is, $E_n = n^2$ for directed graphs and $E_n = n(n+1)\frac{1}{2}$ for undirected graphs. Since the automaton has one node per different isomorphic graph and each node has a transition per possible pair of vertices, it has $|\mathcal{G}_n|$ nodes and $|\mathcal{G}_n|E_n$ transitions. These pose as the main bottleneck of the automaton method, since they are directly related with memory usage, where each node holds a canonical label and each transition a permutation and destination node. Since $\mathcal{G}_n$ grows rapidly with n, this method is only appropriate to small graphs, depending on the available memory.

For the base building on-the-fly method, we run `nauty` once per transition pair (since we build a transition and its reverse per `nauty` call), thus we call it $|\mathcal{G}_n|E_n\frac{1}{2}$ times. To follow a transition of the automaton, assuming it exists, it is necessary to compose a permutation, which takes at most $\mathcal{O}(n)$ time for a graph with n vertices. This is true if we have the default representation, if the permutation to compose with can be compressed, then the time needed is only $\mathcal{O}(1)$.

4.2.5 Subgraph census estimation in temporal graphs

We now turn to a possible application of our automaton method. This case study illustrates how the dynamic graph isomorphism problem is tied to the subgraph census problem.

Let a G be a temporal graph, that is, a graph where each edge is augmented with a time interval that indicates when that edge is active. We want to analyse how patterns evolve in this network and for that we will focus on how a determined induced subgraph of G in a certain timestamp evolves through time. Thus, given two graph types H_1 and H_2 (with the same number of vertices), we want to know the percentage of times that a set of nodes in a certain timestamp in G is isomorphic to H_1 and in a future timestamp isomorphic to H_2. If we do this for all possible graphs H_1 and H_2 of a certain size n, then we get a Markov chain of temporal subgraph transitions that can be used as a fingerprint of the network and be further used for multiple graph mining tasks, such as graph classification, link prediction, ... This technique is similar to what was done in [20], but here only patterns of at most 3 vertices were studied, and to what was done in [41], but here this was done in a edge oriented fashion and with a slightly different formulation.

Doing a complete search of all possible patterns and transitions is possible, but very heavy, even for a relatively small network. Because of that, we only consider connected induced subgraphs. Furthermore, we propose an approximated approach to this problem.

We will first sample a single connected induced subgraph H from G in any timestamp. Afterwards, we follow the vertex set of H through time in G. To do so, we use our automaton method to first represent H and then for each edge that we either turn off or on when its time interval expire or begin, we introduce that change in the automaton. We fix a time step λ, such that whenever λ units of time have passed, we record the current isomorphism class and add a transition on the Markov chain table from the previous isomorphism type to the current one. By doing so, we can follow the isomorphism information of that particular vertex set throughout the whole life time of G. If we repeat this procedure enough times, we have effectively sampled a portion of the temporal transition space.

We did not further expand this example application of our automaton method, but on Chapter 5 we will show its improvement over a simple non-streaming aware method.

Chapter 5

Experimental Analysis

We will now turn to the experimental analysis of our work. Our goal is to show each proposed method has a meaningful contribution to its field. We divide this analysis in four parts, similarly to what was done in the core of this thesis: we first analyze the `FaSE` algorithm, described in Section 3.1 in the context of exact subgraph census; we follow with the `Rand-FaSE` algorithm, described in Section 3.2 in the context of approximated subgraph census; then, we focus on the updating census variant of `FaSE`, described in Section 4.1 in the context of efficiently updating a subgraph census on edge modifications; finally, we close with the analysis of our automaton based method for the dynamic graph isomorphism problem, described in Section 4.2, also, we briefly explore the case study mentioned in Section 4.2.5.

In each part, we first describe the experimental setup we used, including any used datasets, and then follow with the obtained results.

All used datasets are undirected and directed networks. In all networks, weights, self-loops and multiple edges were either ignored or nonexistent.

All tests related with the first two parts (Sections 5.1 and 5.2) were performed on a Linux machine with an Intel Core 2 6600 (2.4GHz) and 2GB of memory. For the remaining two parts (Sections 5.3 and 5.4), we used a Linux machine running Fedora 20 on a AMD Opteron(tm) with 2.30 GHz processor, with 4GB of RAM.

All methods were implemented in `C++` compiled with `GCC 4.8.3`. The source codes are all publicly available:

- `FaSE` and `Rand-FaSE` are available as a joined tool in: `https://github.com/ComplexNetworks-DCC-FCUP/fase`.
- The updating version of `FaSE` is available in: `https://github.com/ComplexNetworks-DCC-FCUP/fase/tree/temporal`.
- The automaton based method for dynamic graph isomorphism is available in: `https://github.com/ComplexNetworks-DCC-FCUP/streaming-small-isomorphism`.

5.1 Exact fast subgraph census

To test the performance of the exact approach, we ran `FaSE` with different subgraph sizes and different networks and compared the execution times to `ESU`, through its publicly available tool, and `Kavosh`, through its original source code. We chose these algorithms since at the time of publication of its original paper [42] they were the main previous approaches. The networks used are summarized in Table 5.1.

Network	Directed	Nodes	Edges	Avg. Degree	Type	Source
StarWars	No	51	157	3.08	Social	Our Own [42]
Jazz	No	198	2,742	13.85	Social	Arenas [15]
Neural	Yes	297	2,359	7.94	Biological	Newman [57]
Foldoc	Yes	13,356	120,700	9.04	Semantic	Pajek [7]

Table 5.1: Networks used for the exact subgraph census experimentation

We implemented both the adjacency list and matrix LS-Labeling methods, but the two had very similar execution times, although the list method ended up having slightly better results most of the time, so we opted to only show the results obtained using it. As stated in Section 3.1, we used the third party tool `nauty` [37] to efficiently perform the isomorphism classifications.

We measure the time each algorithm took to perform a complete k-subgraph census on all networks, with k varying from 3 to 9. Due to time constraints, we only show execution times up to 5 hours. All the results as well as statistics about the number of subgraphs per network and how many leaves of `FaSE`'s g-trie used are shown in Table 5.2.

Network	K	Subgraphs found		FaSE		ESU		Kavosh	
		Types	Occurrences	Time (s)	Leaves	Time (s)	Speedup	Time (s)	Speedup
StarWars	3	2	1,449	<0.01	3	<0.01	—	<0.01	—
	4	6	12,958	<0.01	17	0.04	**23.5**	0.03	**17.6**
	5	21	98,426	0.01	171	0.39	**30.7**	0.21	**16.5**
	6	106	630,369	0.08	2,406	3.12	**38.0**	1.90	**23.1**
	7	699	3,445,808	0.58	26,692	21.95	**38.0**	13.26	**23.0**
	8	5,601	16,320,648	3.55	203,687	133.34	**37.6**	78.18	**22.0**
	9	41,790	67,883,236	19.08	1,133,749	(*)	—	395.90	**20.7**
Jazz	3	2	67,414	<0.01	3	0.14	**31.8**	0.06	**13.6**
	4	6	1,833,618	0.15	17	4.24	**28.9**	2.55	**17.4**
	5	21	49,500,654	4.65	171	143.64	**30.9**	89.3	**19.2**
	6	112	1,266,953,062	140.84	3,113	(**)3,630.00	**25.8**	2,912.43	**20.7**
	7	853	30,166,157,456	3,946.81	106,417	>5h	—	>5h	—
Neural	3	13	47,322	0.01	45	0.09	**16.7**	0.04	**7.4**
	4	197	1,394,259	0.13	1,846	2.21	**17.5**	1.71	**13.5**
	5	7,072	43,256,069	4.73	76,214	102.14	**21.6**	91.03	**19.3**
	6	286,376	1,309,307,357	170.96	2,499,645	(**) 4,420.00	**25.9**	4,636.43	**27.1**
Foldoc	3	13	2,553,830	0.35	45	3.97	**11.2**	2.17	**6.1**
	4	198	228,272,189	27.80	2,304	903.39	**32.5**	308.78	**11.1**
	5	8,345	29,621,881,964	3,735.20	141,115	>5h	—	>5h	—

(*) ESU accepts only 8 as the maximum subgraph size.

(**) Overflow problem in its own reported enumeration time and so we used elapsed time.

Table 5.2: Detailed experimental results for the 4 networks used for the exact setup.

Analyzing the results, the general trend is that `FaSE` obtains better results in all setups than both `ESU` and `Kavosh`, as was expected. Moreover, it was always an order of magnitude faster, except for a couple of outliers. Another observation in order is that there is a tendency for the speedup to increase as k increases, which means there is a larger speedup in setups where the total execution time is higher, which are the ones where a faster algorithm is more critical. This is a sensible outcome, since the speedup comes mainly from the isomorphism tests avoided, which is directly related to the ratio between the total number of subgraphs and number of g-trie leaves and this is a quantity that generally increases for smaller subgraph sizes and larger networks (as is possible to observe in the results table). The actual values are very much network dependent and there is no "external" measure (number of nodes, edges . . .) that allows a prediction of the actual execution times in any order of accuracy, since it heavily depends on combinatorial features of the network.

It is also important to notice that the major bottleneck of a network-centric approach to subgraph census that relies on enumeration is isomorphism testing, which is what the algorithm aims to improve. To check that `FaSE` addresses this and is not a somehow faster implementation of the `ESU` algorithm, we ran it without the g-trie functionality, simulating the actual `ESU` algorithm functioning. The result proved to be slightly better than the original `ESU` code, but was still roughly an order of magnitude slower than `FaSE`'s normal functioning. Furthermore, the contribution of the enumeration process was compared to the final execution time by running the algorithm without the isomorphism tests, meaning we only ran the enumeration algorithm. Obviously, this does not allow to compute the actual census. The results indicate that the actual enumeration is only a tiny fraction (less than 1%) of the whole execution time, confirming what we stated above.

The final aspect we want to highlight is that the number of leaves used by the g-trie has a heavy influence on the memory used by the algorithm. This implies that it is impossible to run it with much larger subgraph sizes than the ones tested in this work. Even though the super exponential growth of the number of subgraph types eventually makes it impossible to even store the individual frequencies of each type, this is still prohibitive and actually potentially slightly affects the execution time.

5.2 Approximated fast subgraph census

We divided the tests in this section into three parts with different aims. We first compared the approximated algorithm with its exact approach, in order to assess the accuracy of the approximation for different sampling values. Then we compared it with previous work, by testing how many subgraphs were sampled per second, so as to evaluate the time efficiency of the approximation. Finally, we tested how the approximation converges to the exact values by measuring the error and standard deviation displayed through various sampling percentages.

Network	Directed	Nodes	Edges	Avg. Degree	Type	Source
Jazz	No	198	2,742	13.85	Social	Arenas [15]
Yeast	No	2,361	6,646	2.81	Biological	Pajek [7]
AstroPh	No	18,772	198,050	10.55	Social	SNAP [32]
Metabolic	Yes	453	2,025	4.47	Biological	Arenas [15]
Foldoc	Yes	13,356	120,700	9.04	Semantic	Pajek [7]
Neural	Yes	297	2,359	7.94	Biological	Newman [57]

Table 5.3: Complex networks used in the Approximation tests

We summarize the networks we used in this analysis in Table 5.3. Note that we repeated some networks used in the previous section, but we included them in this table for completeness.

Recall that we called η to the sampling percentage, the percentage of subgraphs we want to sample. Since our algorithm requires choosing the multiple probabilities per level, p_d, for a given η, we opted for the following three setups, that explore the sampling properties differently:

High: $p_0 = 1, \ldots, p_{k-3} = 1, p_{k-2} = \eta, p_{k-1} = 1$

Medium: $p_0 = 1, \ldots, p_{k-4} = 1, p_{k-3} = \sqrt{\eta}, p_{k-2} = \sqrt{\eta}, p_{k-1} = 1$

Low: $p_0 = 1, p_1 = \sqrt[k-2]{\eta}, \ldots, p_{k-2} = \sqrt[k-2]{\eta}, p_{k-1} = 1$

Note that we always considered p_0 and p_{k-1} to be 1 since due to the way we implemented our algorithm having $p_{k-1} \neq 1$ means it will do all the work enumerating a certain subgraph and then discard it with probability $1 - p_{k-1}$ and having $p_0 \neq 1$ means discarding a whole branch of the enumeration recursive tree, which means a whole isomorphism class could be discarded.

5.2.1 Comparison with the exact approach

To compare with the exact approach, we first ran `Rand-FaSE` with two different input networks, `Yeast` and `Metabolic`, with different sampling percentages. We used these two for this particular test since they are average sized directed and undirected networks and so allow us to perform more time demanding tests that would otherwise be unfeasible on larger networks. To measure the accuracy of the approximation, we calculated the percentage of isomorphism classes correctly estimated by the algorithm over a single run and considered the frequency of an isomorphism class to be correctly estimated when the approximated value is within 15% of the real value (calculated through the exact approach) for the three sampling setups described above (*high*, *medium* and *low*). We did not consider isomorphism classes where the expected number of subgraphs sampled is smaller than 10, the reason being that in these cases the error associated would be too large to estimate the real value in any practical scenario. The obtained results are graphed in a semi-log plot displayed in Figure 5.1.

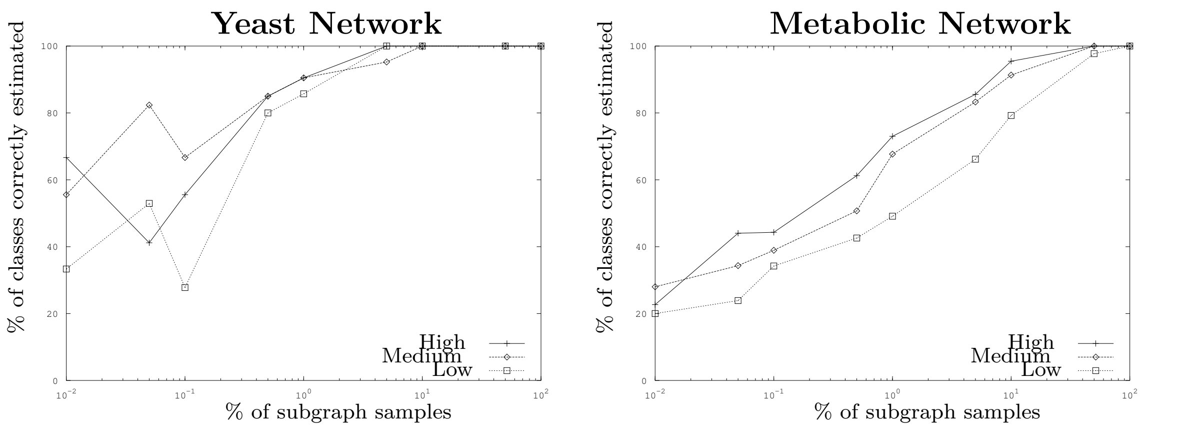


Figure 5.1: Accuracy of `Rand-FaSE` for the undirected `Yeast` Network and directed `Metabolic` Network for size 5 subgraphs.

Excluding a few outliers, both plots are approximately a line (they both eventually converges to 100% correctness). Since this is a semi-log plot, this means that it is approximately a logarithmic function, that is, multiplying by 10 the number of samples should roughly double the correctness of the approximation. Of course this result is dependent on the way we measured correctness and thus is not fit for all scenarios.

Another observation to make is that, as expected, since the *high* setup places the probabilities in lower levels it should have a lower variance, which results in overall better results. Likewise, since the *low* setup distributes the probabilities more evenly it has the highest variance and obtains the overall worse results. There were a few outliers on the lower probabilities, but it was probably due to the fact that for lower sampling percentages the variance is obviously higher and so there are a lot more fluctuations in the results.

To have a better understanding on how the sampling works for individual isomorphism class sizes, we ran the algorithm with different sampling percentages and with subgraph size equal to 3 in the `foldoc` network and measured the relative error to the real value. We used the `foldoc` network to showcase this since it is a rather dense network and thus for the particular subgraph size chosen, it has at least a subgraph of each existing type. Thus it is clearer how our algorithm behaves for lower and higher frequency subgraph classes. The results are showcased in Table 5.4.

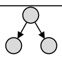
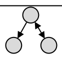
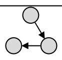
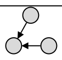
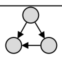
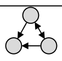
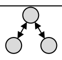
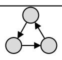
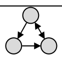
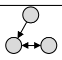
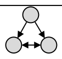
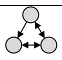
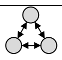
Subgraph Type	Number	Number	Error	Number	Error	Number	Error	Number	Error
	Exact	50% Sample		10% Sample		1% Sample		0.1% Sample	
	178,812	179,364	0.3%	177,400	0.8%	186,500	4.3%	184,000	2.9%
	167,053	166,736	0.2%	170,820	2.3%	159,100	4.8%	138,000	17.4%
	420,580	423,762	0.8%	437,710	4.1%	371,400	11.7%	311,000	26.1%
	1,354,914	1,353,372	0.1%	1,348,450	0.5%	1,321,900	2.4%	1,534,000	13.2%
	30,118	30,448	1.1%	29,420	2.3%	27,400	9.0%	29,000	3.7%
	13,783	13,870	0.6%	14,280	3.6%	13,300	3.5%	7,000	49.2%
	65,626	65,616	0.0%	64,570	1.6%	57,700	12.1%	62,000	5.5%
	676	698	3.4%	670	0.7%	500	25.9%	0	100.0%
	2,254	2,222	1.2%	2,180	3.1%	1,700	24.4%	0	100.0%
	262,620	263,238	0.2%	270,730	3.1%	268,500	2.2%	237,000	9.8%
	29,963	29,972	0.0%	29,130	2.8%	28,000	6.5%	42,000	40.2%
	7,401	7,484	1.2%	7,550	2.1%	5,500	25.6%	5,000	32.4%
	20,030	19,942	0.4%	19,940	0.4%	19,500	2.6%	22,000	9.8%
Total	2,553,830	2,556,724	0.1%	2,572,850	0.7%	2,461,000	3.6%	2,571,000	0.7%

Table 5.4: Results obtained for different sampling percentages for the directed `Foldoc` network for size 3 subgraphs with setup *high*.

Taking a closer look at the table confirms that the relative error for isomorphism classes with fewer subgraphs is higher, specially for the smaller sampling percentages. Furthermore, for the 0.1% there were even isomorphism classes that did not get any subgraphs sampled at all.

5.2.2 Temporal comparison with the previous approaches

Comparing our algorithm with the previous approaches was done by analyzing the runtime performance. We compared our algorithm with `Rand-ESU`, the approximated version of the `ESU` algorithm. Since the functioning of `Rand-ESU` is conceptually similar to `Rand-FaSE` and uses the idea of probabilities per level, when comparing with it we used the same probabilities in the same depths. Note however, that we did not enforce $p_{k-1} = 1$ in `Rand-ESU`'s tests since its implementation places the probability before performing the enumeration, contrary to how `Rand-FaSE` does it (as explained above). Even though in Section 5.1 we compared with `Kavosh`, it does not own an approximate version, as far as we know, so we did not consider it in this section.

We first ran our algorithm against the `Rand-ESU` in the `Jazz` network and the `Neural` network for a 10% sampling percentage and recorded the number of subgraphs sampled per second. We used this instead of the raw execution time since the actual number of sampled subgraphs oscillates and thus it does not represent the quality of the algorithm speed-wise. We chose these two networks to both vary the type of tested networks and the average degree. The results obtained are plotted in Figure 5.2.

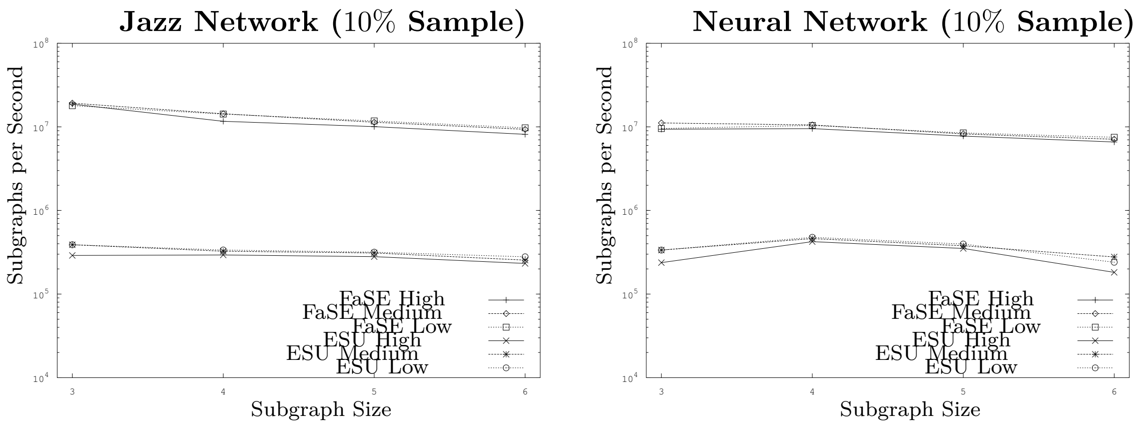


Figure 5.2: Sampling runtime comparison for a 10% sample for the undirected `Jazz` Network and directed `Neural` Network.

The principal aspect to take note is that `Rand-FaSE` always outperforms `Rand-ESU`, being roughly an order of magnitude faster. This result is consistent with the one obtained in the exact approach. However, the speedup is expected to be slightly less in the sampling version, since the speedup derives from the number of enumerated subgraphs that do not require an isomorphism test, thus by reducing the number of subgraphs that are actually enumerated, the speedup will tend to decrease. Nevertheless, the results on these networks show that it is not a noticeable decrease.

Other important observation is that our algorithm, as well as `Rand-ESU`, appear to scale well with the increased subgraph size. As it increases, a small drop is detectable, however it is a very subtle one.

To evaluate how the approximation method runtime compares to the exact method runtime, we ran `Rand-FaSE` on the networks of Subsection 5.2.1 and plotted the execution time for various sampling percentages in relation to the time the exact approach took, on a 5-subgraph census in Figure 5.3. By using the same networks here as in the previous subsection we get an idea of how runtime performance compares with accuracy for the same setups.

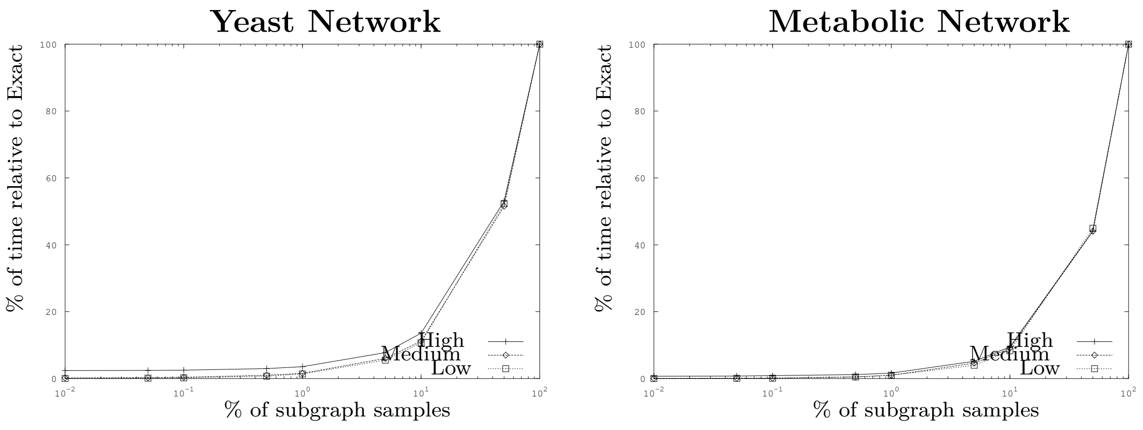


Figure 5.3: Comparing the execution time of various sampling percentages with the exact approach for the undirected `Yeast` Network and directed `Metabolic` Network for size 5 subgraphs.

The result display a roughly linear growth behavior (since the graph is in a semi-log scale, the exponential represents a linear growth). However, even though for both networks a 50% sample takes approximately 50% of the time the exact approach takes, a 1% sample takes 3% of the time the exact approach, for the *high* setup. This effect worsens as the sampling percentage drops and ends up stabilizing at about 2% of the time the exact approach takes, regardless of the sampling percentage. The reason for so is that, thanks to the way the *high* setup is designed, it ends up enumerating all subgraphs up to size $k - 1$. For the *medium* setup a similar effect is noticeable, but much subtler, since in this case we are enumerating all subgraphs up to size $k - 2$. Obviously, the *low* setup does not display this behavior, but likewise, as the sampling percentage decreases the relation between time of the approximation and time of the exact deviates more from an exponential. For example, a 0.1% sample takes approximately 0.2% of the time the exact approach does.

Since the main goal of running an approximation algorithm is to apply it to a network where the exact approach is unfeasible, we tested our algorithm using the ideas discussed in Subsection 3.2.3 in an undirected network with $1,134,890$ nodes and $2,987,624$ edges that represents the network of a Youtube community taken from SNAP [60]. We ran a 0.1% sample for 4-subgraphs with setup *high*, which took about 20 minutes to complete. Based on tests on enumerations on 3-subgraphs of the same network and the results of this section, it should take about a full day to run the exact approach. The results allow us to have an idea of the total number of subgraphs as well as the distribution of the different subgraph types having run for only a very small fraction of the time that the exact approach is expected to take.

5.2.3 Measuring convergence

To bring this section to an end, we performed some tests to assess the convergence of our algorithm. In Subsection 5.2.1 we could observe the percentage of correctly estimated values converging towards the optimal value, so we consolidate this with a more detailed view over the percentage of error and the standard deviation. We ran `Rand-FaSE` with the `astroPh` network with setup *high* for various sampling percentages and measured both the relative error and the standard deviation normalized by the real value. We used this network since it is of a larger size and thus allows us to better observe the convergence and standard deviation evolution. The results are plotted in Figure 5.4. Note that “Sub1” refers to the “L” shaped size 3 subgraph and “Sub2” refers to the triangle (size 3 complete graph). The number of subgraphs of type “Sub1” and “Sub2” in the `astroPh` network is of the same order of magnitude.

As expected, both the relative error and the standard deviation decrease towards 0. It is interesting to notice that above the 10% sampling percentage the relative error and standard deviation stabilize and decrease very slowly, with little fluctuations.

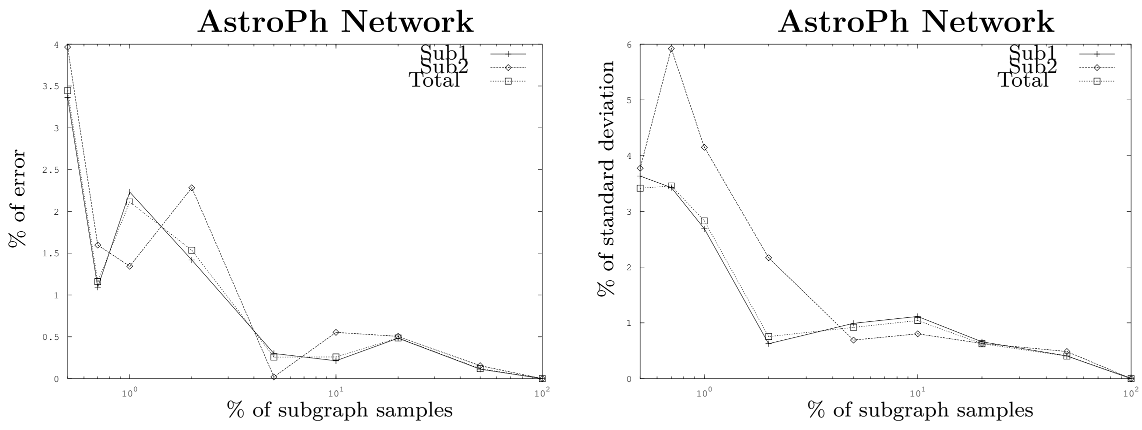


Figure 5.4: Testing convergence through % of error to the real value and standard deviation (normalized by the real value).

5.3 Updating a census

Since there are no known approaches that efficiently update a subgraph census, in order to avail the performance of our census updating variant of `FaSE`, we ran it against a baseline method that runs `FaSE` from scratch after each edge modification. To do so, we chose 4 different networks that are listed in Table 5.5 and compared the runtime of both approaches.

Network	Directed	Nodes	Edges	Avg. Degree	Type	Source
Jazz	No	198	2,742	13.85	Social	Arenas [15]
Power	No	4,941	6,594	1.33	Geo-spacial	Newman [57]
MCortex	Yes	30	311	10.37	Neurobiological	Paper [53]
MVisCortex	Yes	71	746	10.51	Neurobiological	Paper [53]

Table 5.5: Complex networks used in the updating subgraph census tests

Since these are static networks, we needed a stream of edge modifications in order to be able to perform the experiments. Instead of choosing random edges, we chose to use the random graphs with given subgraph frequencies of the case study [51] mentioned in Subsection 4.1.1. We fix a set of parameters that made sense in the context of the random graph model and ran two implementations of this model: one with the updating method and the other without. Table 5.6 shows the average execution time, in seconds, for each network, comparing the efficient update against running a full census after each edge modification. We perform a 4-subgraph census for the `MCortex` and `MVisCortex` networks and a 5-subgraph census for `Jazz` and `Power`.

	`Jazz`	`Power`	`MCortex`	`MVisCortex`
Efficient Update (s)	1,034.06	239.56	64.85	0.22
Full Census (s)	2,5102.0	4,274.47	103.58	12.35
Speedup ($\times$ faster)	24.3	17.8	1.6	56.1

Table 5.6: Average execution time, in seconds, and speedup, of the efficient update in comparison with the full census.

For the `MCortex`, on average, each network took nearly twice as much to perform the full census after each edge modification than using our efficient updating method. However, for the `jazz` and `power` networks, on average, each network was 1 order of magnitude faster using the efficient update technique and the `MVisCortex` was about 2 orders of magnitude faster.

Clearly, both directed networks are outliers of efficiency, probably because they are both small dense networks. Our updating method works best for larger sparse networks, because in this case, on average, the number of subgraphs that change after a single edge addition or removal is only a small fraction of the total number of subgraphs. In this sense, the `jazz` and `power` networks are better fits for this model, as are most social networks.

Also, note that these speedups are influenced by the graph generation method runtime. In principle, performing the subgraph censuses is the bottleneck of the graph generation method, however the rest of the method can have a relevant weight on the total runtime. This might also have been one of the reasons for such a low speedup on `MCortex`.

5.4 Dynamic graph isomorphism

Our analysis focuses on two main themes: the compressibility of the transition permutations and the runtime of using the automaton versus using a simpler base approach, namely recalculating the isomorphism class for every instance using `nauty`.

We define two notions of compressibility: C_0 is the *zero compressibility* of an automaton, meaning the percentage of transition permutations that are the identity permutation; C_1 is the *one compressibility* of an automaton, meaning the percentage of transition permutations that are either a single transposition or the identity permutation. In Table 5.7 we show the C_0 and C_1 values for some automata of different sizes, both undirected and directed, for the two building methods. We omit the results pertaining to automata that were too memory intensive to compute (the directed size 6, 7 and 8 automata).

	On-the-fly				Pre-build			
	Undirected		Directed		Undirected		Directed	
	C_0	C_1	C_0	C_1	C_0	C_1	C_0	C_1
3	25%	75%	31%	73%	33%	78%	34%	69%
4	18%	52%	25%	62%	24%	53%	29%	56%
5	14%	39%	20%	53%	20%	44%	26%	46%
6	12%	29%	-	-	15%	30%	-	-
7	9%	21%	-	-	11%	22%	-	-
8	6%	15%	-	-	11%	19%	-	-

Table 5.7: Values of C_0 and C_1 for different automata and build methods

It is clear that the pre-build method achieves better compressibilities, specially C_0 compressibilities, which are more critical in terms of runtime. If we discount the building time, which is slightly higher for the pre-build method (but constant), in general, this results in a speedup of up to 2 times, for most input graph streams. However, the increased building time means that for higher vertex numbers (from 8 up) the runtime advantage only becomes noticeable for larger stream sizes. This result was obtained empirically using the graph streams used in the analysis of the following paragraphs.

To compare the temporal behaviour of our method with the base `nauty` recomputation method we generated several synthetic networks, with different goals and variants. Here we use the version using the on-the-fly building method. We selected 13 graph stream descriptions with different properties and for each one studied the runtime of our method and of the base recomputation method for several stream sizes. We summarise them in Table 5.8 and will explain the origin of each one next.

Designation	Direction	$\|V(G)\|$	Origin	Step
`ER-6`	Undirected	6	ER Model	1
`ER-7`	Undirected	7	ER Model	1
`ER-8`	Undirected	8	ER Model	1
`PR-6`	Undirected	6	PR Model	1
`PR-7`	Undirected	7	PR Model	1
`PR-8`	Undirected	8	PR Model	1
`SW-5`	Undirected	5	SWAP Model	4
`SW-6`	Undirected	6	SWAP Model	4
`SW-7`	Undirected	7	SWAP Model	4
`dER-4`	Directed	4	D-ER Model	1
`dER-5`	Directed	5	D-ER Model	1
`dPR-4`	Directed	4	D-PR Model	1
`dPR-5`	Directed	5	D-PR Model	1

Table 5.8: Graphs used for the experimental analysis

The designation is used to identify the graph description in the following analysis, the direction is the graph type (directed or undirected), $|V(G)|$ represents the number of vertices, the origin is the model used to generate the stream, the step is the number of graph changing operations between each canonization request, that is, if the step of a graph stream description is k then we are only interested on the canonization of every other k element of $S(G)$, as defined previously.

The following list summarises each model used to generate graph streams:

- **ER Model**, is a simple model generation based on the Erdos-Rényi [12] random graph model, where each graph changing operation is chosen uniformly at random from all the possible vertex pairs. Its directed version, the **D-ER Model** is analogous.

- **PR Model**, is a model based on a preferential attachment rule for networks [6] where each vertex pair is chosen as a graph changing operation depending on the degree of each of its vertices. Its directed version, the **D-PR Model** is analogous.

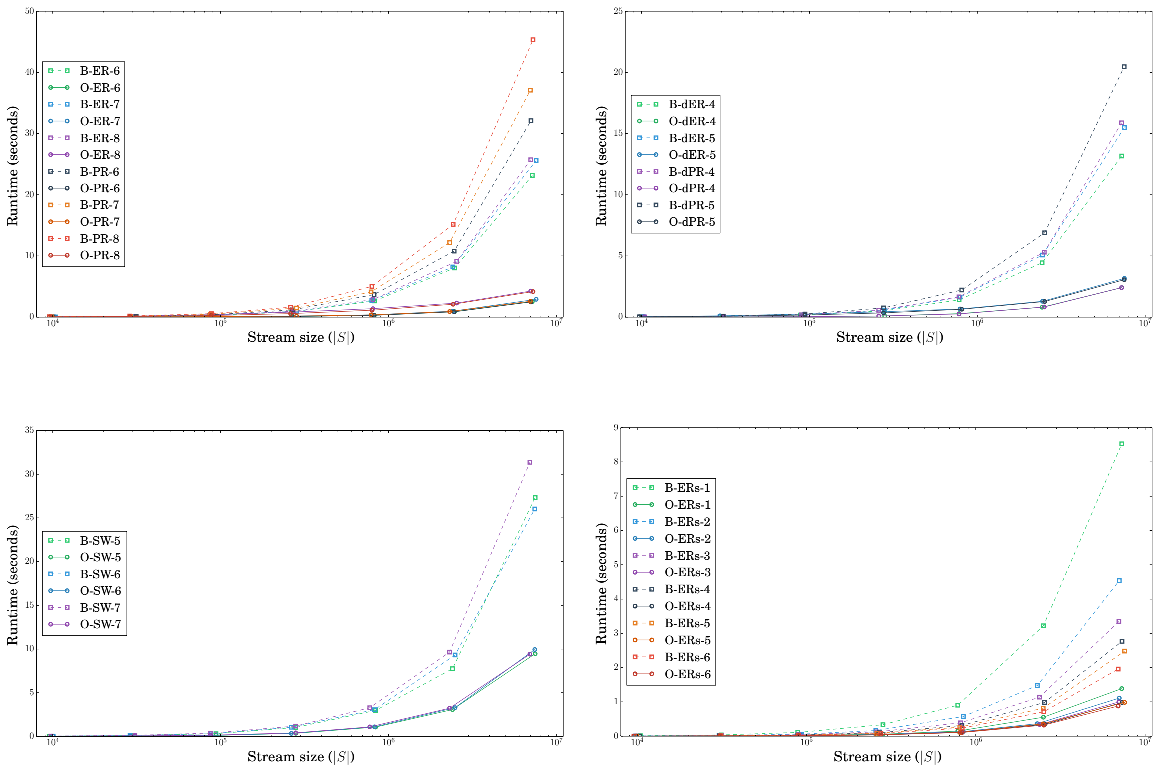


Figure 5.5: Comparison of our method versus the base method for multiple streams. A prefix of `B-` indicated this result is related to the base recomputation method and a prefix of `O-` indicated this result is related to our method

- **SWAP Model**, is a model that simulates edge swapping operations, each 4 contiguous graph changing operation represent swapping two edges (chosen uniformly at random). It has a step of 4 because we are only interested in the graphs after each swap.

To study each one we generated multiple streams with increasing sizes, from 10^4 to 10^7 and observed the runtime of both methods. We plot the results of that analysis in Figure 5.5 (note that the X axis is in logarithmic scale). The top left figure pertains to the undirected models, the top right figure directed models, the bottom left figure contains all streams based on the `SWAP` model and the bottom right figure represents a growing step experiment that will be further explained bellow. The dashed lines with square points always represent times obtained using the base method, and the solid lines with circular points were obtained using our method.

It is noticeable that our method greatly outperforms the base method on all streams. Furthermore, the asymptotic behaviour of our method suggests that for even greater stream sizes the benefit is only going to increase. The same applies to the speedups obtained by the method. For the unit step streams, the speedup grew approximately linearly from about one up to 15 times. For the `SWAP` model the speedup was more stable, varying between 2.7 and 3.1. It is also interesting to note that our method had very similar results for different stream models with the same number of vertices, whereas the base method was much more input dependant, which shows that our method is agnostic to the input source.

In the bottom left figure, regarding the `SWAP` model, it is interesting to note that, even though there is a step of 4, our method still maintains a good speedup when comparing to the base method. Note that the higher the step the worse is the benefit of our method, since the base method only performs computation when it is required to return a canonical label whereas our method has to update the automaton after each change operation.

There is a clear tipping point observable in the data, which represents the minimum stream size for which it is more beneficial to use our method instead of the base method. For the top left figure, it appears to be around 10^5. This value is related to the automaton size and with the number of times that the method needs to run `nauty` in the building time. We can extrapolate from here and estimate for different streams sizes and different inputs (even with a number of vertices higher than memory restrictions would allow) and estimate how good our method is going to be in relation to the base method.

Building on this tipping point argument, the bottom right figure shows a growing step experiment. We used the `ER` model to generate various networks with 6 vertices and artificially vary the step from 1 to 6 (each integer in the figure legend indicates the step of that measure, using a similar notation as in the other three figures). It is important to point out that for all different steps, our method outperformed the base method, with decreasing speedups. Additionally, as we increase the step, the mentioned tipping point of efficiency also increases. Further similar experiments indicate that there is always a tipping point when the step is of the order of $\mathcal{O}(n)$, which means our method is useful as long as the average number of edge modifications between required canonical labels is in the order of the number of vertices.

5.4.1 Case study: subgraph census estimation in temporal graphs

We implemented a basic version of the method mentioned in Subsection 4.2.5 (which is available on the public source code in the `Examples` directory) and ran it using both the base method and our method as the underlying isomorphism tool. To compare the runtime of both methods, we ran them on a small set of complex networks with 1,000,000 samples, which we list in Table 5.9.

Designation	Name	Direction	$\|V(G)\|$	$\|S(G)\|$	Origin
`email`	`email-eu-core`	Directed	986	332,334	Communication [33]
`college`	`college-msg`	Directed	1,899	20,296	Communication [33]
`infectious`	`infect`	Undirected	410	17,298	Social [21]
`arxiv`	`arxiv-hep-th`	Undirected	22,908	2,673,133	Coauthorship [33]

Table 5.9: Graphs used for the case study

The runtimes obtained for multiple subgraph size n are shown in Table 5.10. These runtimes include the time for sampling and performing other supporting computation, which lower the speedup in relation to the runtimes obtained in the beginning of this section.

	Using the base method				Using our method				Mean Speedup
	`email`	`college`	`infect`	`arxiv`	`email`	`college`	`infect`	`arxiv`	
3	10.86	8.81	8.33	32.81	6.62	5.08	3.19	31.52	1.75×
4	22.55	17.12	17.81	66.72	10.73	8.62	4.98	60.48	2.19×
5	34.45	30.01	34.36	113.95	16.24	13.34	8.56	98.74	2.39×

Table 5.10: Runtimes, in seconds, for the case study analysis

Chapter 6

Conclusion

And so we reach the end of our journey.

We first looked into `FaSE`, an algorithm that performs an exact subgraph census. By making use of the common topology of the enumerated subgraphs and encapsulating this information in a tree structure called a g-trie, `FaSE` is able to discard most of the isomorphism tests required to correctly identify each subgraph type, which is the main bottleneck of the problem it aims to solve. Hence, it achieves much better results than any of the past approaches that tackle the same problem, which is shown by the results found by comparing all approaches.

Thanks to `FaSE`'s use of LS-Labeling, the algorithm is very generic and allows for various different LS-Labeling functions. This means that the algorithm can be easily adapted for different scenarios such as colored graphs or multigraphs. In these more complex setups, network-centric approaches have a clear advantage over other approaches that require a pre-generated set of subgraphs as input since the addition of colors or multiedges vastly increments the number of possible subgraphs types. Network-centric algorithms naturally only count the existing types, which are normally a very small fraction of the total number of possibilities.

Subsequently, we turned to a sampling version of `FaSE`, appropriately named `Rand-FaSE`. It acts as a logical extension and works by sampling only a percentage of the total number of subgraphs. By placing a probability in each depth of the enumeration tree and only continuing the enumeration at each branch if a drawn random number is smaller than that depth's probability, the algorithm gives an unbiased estimate of the real frequency of each subgraph type in the original network.

We then described a technique to efficiently update the frequency of subgraphs after an addition or removal of a single edge. In summary, our updating `FaSE` algorithm works by searching all the subgraphs that touch the edge's endpoints and updates their frequency. We showed that, on average, it is at least 2 times faster and in many cases orders of magnitude faster than running the full networks census from scratch on a particular case study related to generating random networks with prescribed subgraph frequencies.

Finally, we introduced a new problem that inserts the known problem of graph isomorphism on a dynamic or streaming environment. It consists on computing isomorphism information for several graphs in a stream. We focused on fully dynamic streams, meaning between each iteration we can insert or delete edges.

We presented an efficient algorithm that tackles this problem using a data structure similar to a discrete finite automaton to represent the full space of different isomorphism classes. Compared to a simple non-streaming-aware approach of recomputing the solution for each iteration of the stream, the automaton method and its variations obtained a much better performance, with speedups increasing with the stream size. We also briefly studied the applicability of our method, studying how the stream parameters (the stream size, the stream step, . . .) vary while keeping the usefulness of our method in relation to the simpler approach.

In terms of potential future work that can be drawn from this thesis, we first note that there is not much improvement to be made on top of our exact algorithm. Currently, the focus of the field is geared towards combinatorial algorithms like [1, 19, 45], which, even though can only target small subgraph sizes, are much more efficient that general methods like `FaSE`. It is highly unlikely, if not impossible, that a general method that enumerates all occurrences is ever able to match the efficiency of combinatorial approaches. However, it is also unlikely that these combinatorial approaches can be generalized in a way that is more efficient that methods like `FaSE` for larger subgraph sizes. So both methods will most likely coexist, albeit only new combinatorial methods have been proposed since the publication of `FaSE`.

If we look into approximated approaches the coexistance scenario is similar, since there are also new sampling methods that use combinatorial properties of graphs, like [56]. Fortunately, there is still a lot of improvement to be made on both, from adaptive sampling approaches (that adapt the sampling percentages depending on the density of the input graph) to methods that analytically compute frequencies based on different null models of graphs.

The brighter future, in our opinion, is reserved to the dynamic methods. As we saw on Chapter 2, there are very few works that target any variant of the subgraph census problem on a dynamic environment, so much so that, as far as we know, we were the first to describe and tackle the dynamic isomorphism problem. There is still a lot of low hanging fruit on several levels, for example, there is no known combinatorial approach that updates frequencies of subgraphs when there are edge modifications. Furthermore, applying these techniques to real temporal networks can lead to a lot of new insights that were previously harder to obtain. We can also think of extending the currently known techniques to parallel or distributed environments.

Bibliography

[1] Ahmed, N. K., Neville, J., Rossi, R. A., and Duffield, N. (2015). Efficient graphlet counting for large networks. In *Data Mining (ICDM), 2015 IEEE International Conference on*, pages 1–10. IEEE.

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

[3] Aparicio, D., Paredes, P., and Ribeiro, P. (2014). A scalable parallel approach for subgraph census computation. In *European Conference on Parallel Processing*, pages 194–205. Springer International Publishing.

[4] Arvind, V., Das, B., and Köbler, J. (2007). The space complexity of k-tree isomorphism. In *International Symposium on Algorithms and Computation*, pages 822–833. Springer.

[5] Babai, L. (2016). Graph isomorphism in quasipolynomial time [extended abstract]. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing*, pages 684–697. ACM.

[6] Barabási, A.-L. and Albert, R. (1999). Emergence of scaling in random networks. *Science*, 286(5439):509–512.

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

[8] Bhuiyan, M., Rahman, M., Rahman, M., and Hasan, M. A. (2012). Guise: Uniform sampling of graphlets for large graph analysis. In *IEEE International Conference on Data Mining*, ICDM, pages 91–100.

[9] Choobdar, S., Ribeiro, P., Bugla, S., and Silva, F. (2012). Co-authorship network comparison across research fields using motifs. In *IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pages 147–152. IEEE.

[10] Cook, S. A. (1971). The complexity of theorem-proving procedures. In *ACM Symposium on Theory of computing*, STOC, pages 151–158, New York, NY, USA. ACM.

[11] Costa, L., Oliveira Jr, O., Travieso, G., Rodrigues, F., Boas, P., Antiqueira, L., Viana, M., and Da Rocha, L. (2011). Analyzing and modeling real-world phenomena with complex networks: a survey of applications. *Adv Phys*, 60:329–412.

[12] Erdos, P. and Rényi, A. (1960). On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(1):17–60.

[13] Ferreira, R. (2013). Efficiently listing combinatorial patterns in graphs. *arXiv preprint arXiv:1308.6635*.

[14] Fortunato, S. (2010). Community detection in graphs. *Phys Rep*, 486(3-5):75–174.

[15] Gleiser, P. M. and Danon, L. (2003). Community structure in jazz. *Advances in Complex Systems*, 06(04):565–573.

[16] Goldreich, O., Micali, S., and Wigderson, A. (1991). Proofs that yield nothing but their validity or all languages in np have zero-knowledge proof systems. *Journal of the ACM (JACM)*, 38(3):690–728.

[17] Gori, M., Maggini, M., and Sarti, L. (2005). Exact and approximate graph matching using random walks. *IEEE transactions on pattern analysis and machine intelligence*, 27(7):1100–1111.

[18] Grochow, J. and Kellis, M. (2007). Network motif discovery using subgraph enumeration and symmetry-breaking. *Research in Computational Molecular Biology*, pages 92–106.

[19] Hočevar, T. and Demšar, J. (2017). Combinatorial algorithm for counting small induced graphs and orbits. *PloS one*, 12(2):e0171428.

[20] Huang, H., Tang, J., Liu, L., Luo, J., and Fu, X. (2015). Triadic closure pattern analysis and prediction in social networks. *IEEE Transactions on Knowledge and Data Engineering*, 27(12):3374–3389.

[21] Isella, L., Stehlé, J., Barrat, A., Cattuto, C., Pinton, J.-F., and Van den Broeck, W. (2011). What's in a crowd? analysis of face-to-face behavioral networks. *Journal of theoretical biology*, 271(1):166–180.

[22] Itzkovitz, S., Levitt, R., Kashtan, N., Milo, R., Itzkovitz, M., and Alon, U. (2005). Coarse-graining and self-dissimilarity of complex networks. *s. Phys. Rev. E (Stat. Nonlin. Soft Matter Phys.)*, 71(016127).

[23] Janssen, E., Hurshman, M., and Kalyaniwalla, N. (2012). Model selection for social networks using graphlets. *Internet Mathematics*.

[24] Junttila, T. and Kaski, P. (2007). Engineering an efficient canonical labeling tool for large and sparse graphs. In *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 135–149. SIAM.

[25] Kashani, Z., Ahrabian, H., Elahi, E., Nowzari-Dalini, A., Ansari, E., Asadi, S., Mohammadi, S., Schreiber, F., and Masoudi-Nejad, A. (2009). Kavosh: a new algorithm for finding network motifs. *BMC bioinformatics*, 10(1):318.

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

[27] Khakabimamaghani, S., Sharafuddin, I., Dichter, N., Koch, I., and Masoudi-Nejad, A. (2013). Quatexelero: An accelerated exact network motif detection algorithm. *PLoS ONE*, 8(7):e68073.

[28] Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, 220(4598):671–680.

[29] Kovanen, L., Karsai, M., Kaski, K., Kertész, J., and Saramäki, J. (2011). Temporal motifs in time-dependent networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2011(11):P11005.

[30] Kreher, D. L. and Stinson, D. R. (1999). Combinatorial algorithms: generation, enumeration, and search. *SIGACT News*, 30(1):33–35.

[31] Kuramochi, M. and Karypis, G. (2004). An efficient algorithm for discovering frequent subgraphs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1038–1051.

[32] Leskovec, J., Kleinberg, J. M., and Faloutsos, C. (2007). Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery From Data*, 1(1).

[33] Leskovec, J. and Krevl, A. (2014). SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`.

[34] Li, X., Stones, D. S., Wang, H., Deng, H., Liu, X., and Wang, G. (2012). Netmode: Network motif detection without nauty. *PLoS One*, 7(12):e50093.

[35] Marcus, D. and Shavitt, Y. (2010). Efficient counting of network motifs. In *ICDCS Workshops*, pages 92–98. IEEE Computer Society.

[36] McGregor, A. (2014). Graph stream algorithms: a survey. *ACM SIGMOD Record*, 43(1):9–20.

[37] McKay, B. (2012). nauty. `http://cs.anu.edu.au/~bdm/nauty/`.

[38] McKay, B. D. and Piperno, A. (2014). Practical graph isomorphism, ii. *Journal of Symbolic Computation*, 60:94–112.

[39] Milo, R., Itzkovitz, S., Kashtan, N., Levitt, R., Shen-Orr, S., Ayzenshtat, I., Sheffer, M., and Alon, U. (2004). Superfamilies of evolved and designed networks. *Science*, 303(5663):1538–1542.

[40] Milo, R., Shen-Orr, S., Itzkovitz, S., Kashtan, N., Chklovskii, D., and Alon, U. (2002). Network Motifs: Simple Building Blocks of Complex Networks. *Science*, 298(5594):824–827.

[41] Paranjape, A., Benson, A. R., and Leskovec, J. (2016). Motifs in temporal networks. *arXiv preprint arXiv:1612.09259*.

[42] Paredes, P. and Ribeiro, P. (2013). Towards a faster network-centric subgraph census. In *Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, ASONAM '13, pages 264–271, New York, NY, USA. ACM.

[43] Paredes, P. and Ribeiro, P. (2015). Rand-fase: fast approximate subgraph census. *Social Network Analysis and Mining*, 5(1):1–18.

[44] Paredes, P. and Ribeiro, P. (2016). Large scale graph representations for subgraph census. In *International Conference and School on Network Science*, pages 186–194. Springer International Publishing.

[45] Pinar, A., Seshadhri, C., and Vishal, V. (2016). Escape: Efficiently counting all 5-vertex subgraphs. *arXiv preprint arXiv:1610.09411*.

[46] Pržulj, N. (2010). Biological network comparison using graphlet degree distribution. *Bioinformatics*, 26(6):853–854.

[47] Ribeiro, P. and Silva, F. (2010). Efficient subgraph frequency estimation with g-tries. In *International Workshop on Algorithms in Bioinformatics*, volume 6293 of *WABI*, pages 238–249. Springer.

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

[49] Schiller, B., Jager, S., Hamacher, K., and Strufe, T. (2015). Stream-a stream-based algorithm for counting motifs in dynamic graphs. In *International Conference on Algorithms for Computational Biology*, pages 53–67. Springer.

[50] Schreiber, F. and Schwobbermeyer, H. (2004). Towards motif detection in networks: Frequency concepts and flexible search. In *International Workshop on Network Tools and Applications in Biology*, NetTAB, pages 91–102.

[51] Silva, M. E., Paredes, P., and Ribeiro, P. (2017). Network motifs detection using random networks with prescribed subgraph frequencies. In *Workshop on Complex Networks CompleNet*, pages 17–29. Springer, Cham.

[52] Slota, G. M. and Madduri, K. (2013). Fast approximate subgraph counting and enumeration. In *42nd International Conference on Parallel Processing (ICPP)*, pages 210–219.

[53] Sporns, O. and Kötter, R. (2004). Motifs in brain networks. *PLoS Biol*, 2(11):e369.

[54] Valverde, S. and Solé, R. V. (2005). Network motifs in computational graphs: A case study in software architecture. *Phys. Rev. E*, 72:026107.

[55] Wang, C. and Chen, L. (2009). Continuous subgraph pattern search over graph streams. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 393–404. IEEE.

[56] Wang, P., Tao, J., Zhao, J., and Guan, X. (2015). Moss: A scalable tool for efficiently sampling and counting 4-and 5-node graphlets. *arXiv preprint arXiv:1509.08089*.

[57] Watts, D. and Strogatz, S. (1998). Collective dynamics of small-world networks,. *Nature*, 393:440–442.

[58] Wernicke, S. (2006). Efficient detection of network motifs. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 3(4).

[59] Wu, G., Harrigan, M., and Cunningham, P. (2011). Characterizing wikipedia pages using edit network motif profiles. In *3rd Int. workshop on search and mining user-generated contents (SMUC)*, pages 45–52, New York, NY, USA. ACM.

[60] Yang, J. and Leskovec, J. (2012). Defining and evaluating network communities based on ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, MDS '12, pages 3:1–3:8, New York, NY, USA. ACM.