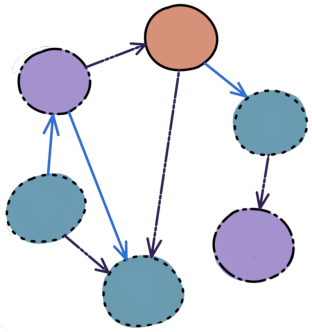

SUBGRAPH PATTERNS IN COLORED NETWORKS

Beatriz Maria Franco Pinto
Master's degree in Computer Science
Computer Science Department
2021

Supervisor
Pedro Manuel Pinto Ribeiro, Assistant Professor,
Faculty of Sciences, University of Porto

Abstract

The Universe evolved as a giant set of complex systems with large numbers of smaller interconnected units interacting in non-trivial patterns, which happens to fit the definition of a complex network. These systems can therefore be represented as graphs. The study of network science has proven of immense value for expanding our knowledge of the phenomena that shape our world and our own existence, by representing information in simple mathematical structures.

In this thesis focus our attention on a particularly hard problem, that has seen a lot of real world applications, specially in biological networks, the *Subgraph Census Problem*. Implementations to perform subgraph census face computational challenges due to its unavoidable relation to the Graph Isomorphism Problem, which is proven NP-complete for general graphs.

Although the SCP as been the focus of many studies and algorithms in the last few decades, there is not so much work done regarding this problem for the case of colored networks. The representation of colors in a network's node and edges can help identify important functional interactions, that we would not be able to see otherwise. Our work aims to expand our understanding on the expression of patterns in large colored networks with the development of a new subgraph enumeration algorithm that supports color labeled nodes and edges.

We discuss the relevance of finding patterns with color information, and the challenges faced in this type of approach, concerning for instance graph representations, the GIP and motif definition the for colored case. We provide an implementation called *ColorFaSE*(from COLORed FAst Subgraph Enumeration), an efficient way of solving the CSCP.

Our approach was based in the extension of the work done in *FaSE*, one of faster the state-of-the-art algorithms to the SCP, in order to support color information for both the nodes and edges. This method in yet unpublished work, and will soon be publicly available.

Lastly, we tested the performance of our method in a series of experiments, putting our code to test in both real and synthetic networks and compared it against the algorithms that are publicly available and we considered to be possible competitors for *ColorFaSE*. Our approach shows to be competitive, not only in terms of running time, but also because of the wider set of colors and subgraphs sizes that it is able to support.

Resumo

O Universo está em constante mudança e comporta-se como um conjunto gigantesco de sistemas complexos, cada um com um grande número de unidades mais pequenas interligadas, que interagem em padrões não triviais. Esta descrição vai de encontro à definição que conhecemos de uma rede complexa. Estes sistemas podem, portanto, ser representados como redes em grafos. O estudo da ciência de redes tem demonstrando imenso potencial para expandir o nosso conhecimento sobre os fenómenos que moldam o mundo e a nossa própria existência, ao representar informação em estruturas matemáticas simples (grafos).

Nesta dissertação, concentramos a nossa atenção num problema particularmente difícil, que tem visto muitas aplicações no mundo real, especialmente em redes biológicas, o *Subgraph Census Problem*. As implementações para realizar o census a uma rede apresentam desafios computacionais, devido à sua inevitável relação com o Problema do Isomorfismo em Grafos, que é conhecido como NP-completo.

Embora o SCP tenha sido o foco de muitos estudos e algoritmos nas últimas décadas, não tem havido tanta pesquisa em relação a este problema para o caso das redes coloridas. A representação de cores no vértices e arestas de uma rede pode ajudar a identificar interacções funcionais importantes, que de outra forma passariam despercebidas. O nosso trabalho visa expandir a compreensão que temos da expressão de padrões em grandes redes coloridas, com o desenvolvimento de um novo algoritmo de enumeração de subgrafos que suporta vértices e arestas coloridas.

Neste trabalho, discutimos a relevância de encontrar padrões com informação de cor, e os desafios enfrentados neste tipo de abordagem no que diz respeito, por exemplo às representações dos grafos, o GIP e a definição do motif para o caso colorido. A nossa contribuição prede-se também com a implementação do método *ColorFaSE*(de COLORED FAst Subgraph Enumeration), uma forma eficiente de resolver o (Colored)SCP.

A nossa metodologia baseou-se na extensão do trabalho realizado em *FaSE*, um dos mais rápidos algoritmos do estado-da-arte para o SCP, a fim desta ser capaz de suportar informação relativa à cor tanto dos nós como das arestas dos grafos. Este método, é um trabalho ainda por publicar e em breve deverá estar disponível.

Por fim fizemos uma série de experiências para testar a nossa implementação, e comparar a

sua performance com algoritmos que consideramos ser possíveis concorrentes. O nosso método mostrou ser competitivo, não só por se mostrar mais rápido nos testes em redes reais e sintéticas, como também por permititir a contagem de subgrafos de tamanhos maiores, para um maior número de cores.

Acknowledgments

To my advisors Prof. Pedro Ribeiro and Luciano Grácio, for all the discussion and ideas, but mostly for always being supportive, helpful, and always showing up with a smile. I specially want to thank Pedro Ribeiro for giving me the opportunity to work in a theme with such great interest and potential, and to make it work in a way that takes me out of my comfort zone. And to Luciano for all the patience, ideas and meetings that have been crucial for me to go through with the implementation.

To my parents, for always being understanding throughout a tough year and for being both my biggest inspirations and my greatest supporters. And to my grandma, for being the most loving person and one of my best friends, and for always caring and celebrating my accomplishments.

To Rafael, for being my best friend and partner in everything, for always being there in my best and worst moments, for understanding my fears and listening to me every single time without judgment, for believing in me and always making me laugh.

To my closest friends, with whom I know I can always count on, for understanding by absence sometimes, and for still making great memories when we are together. Hector, João, Filipe, Francisca, Mafalda, Pedro, Mário e Gonçalo, thank you for being such good friends, each one in a different way.

Contents

List of Tables

List of Figures

List of Algorithms

Acronyms

CGC Colored Graph Canonization

CGI Colored Graph Isomorphism

CSCP Colored Subgraph Census Problem

DCC Departamento de Ciência de Computadores

DFS Depth-First Search

FCUP Faculdade de Ciências da Universidade do Porto

GC Graph Canonization

GI Graph Isomorphism

ISI Induced Subgraph Isomorphism

SC Subgraph Census

SCP Subgraph Census Problem

Chapter 1

Introduction

The mystery of life begins with the intricate web of interactions, integrating the millions of molecules within each organism. The enigma of the society starts with the convoluted structure of the social network... Networks are the prerequisite for describing any complex system.

Albert-László Barabási, Linked: How Everything Is Connected to Everything Else and What It Means for Business, Science, and Everyday Life

It has always been part of our nature as humans to try to understand the world around us. Much like a small child that starts making questions which are progressively harder to answer, Humanity has also been thriving and learning more about the physical systems that describe the natural world, and they keep revealing to be more complex.

The Universe itself evolved as a giant set of complex systems with large numbers of smaller interconnected units interacting in non-trivial **patterns**, which happens to fit the definition of a complex network. These systems can therefore be represented as graphs. The study of network science is hence a crucial tool for expanding our knowledge of the phenomena that shape our world and our own existence. This thesis aims to expand our understanding on the expression of patterns in large colored networks with the development of a new subgraph enumeration algorithm that supports labeled nodes and edges.

1.1 Context

In the last few decades, much has changed in the way humans interact with each other and how we access information. From the birth of the World Wide Web, to the wide spread of telecommunications and the progress made in the ways people can travel and move, we are living in a progressively more globalized world. However, this readiness of large amounts of data and globalization eventually came to underline the importance of understanding the complex systems that govern things around us and the difficulties in extracting this knowledge. The field

of Network Science emerged from this need to find ways to describe and learn how (complex) collective groups of interacting elements behave as a whole.

Complex Networks

By representing these interactions of entities in a mathematical structure, a graph, we are able to focus on its topology and find patterns that surface only when we have seen the system as a whole without stressing too much about the individual properties of its elements. With larger and larger fluxes of data available on almost any subject we can imagine, the study of Network Science and Complex Networks has proved extremely relevant in the extraction of knowledge for many traditional fields. Nowadays, networks have been present in several applications on topics such as biological research[6], climate change[16], quantum information physics[8], economics[16], social studies[32], epidemiology[7] among others.

For instance, in the past 2 years, due to the new coronavirus pandemic (again highly fuelled by our very interconnected global economy), society has become almost familiar with social networks and graph concepts. Since the outbreak and spread of a virus in society can be mapped into a graph, as it has been done with SARS-CoV-2[7], there are many interesting properties of these structures that can give information to aid decision-making in disease control. Graph Theory can help us understand how the infection is spreading and how effective are the mitigation measures using infection models[50], tracing cases using social and communication networks[53], as well as predicting the propagation and growth of the epidemic[7].

Although graphs are very simple structures, they can encapsulate loads of information in their topology and be a subject of problems not so easy to solve. Generally speaking these network science problems are usually tackled along the following types of tasks:

- computing statistical and structural properties of the networks in study like the degree distributions, average path length, node centrality, etc;
- building models that can help explain and categorize the nature of the complex systems as in infection network models, community detection, etc;
- forecasting the graph's evolution and its properties as in temporal networks and link prediction tasks.

Motif Discovery

Another insightful structural property we can study in networks, is the presence of interesting groups of nodes and their connections. These groups, which can vary in size, are called subgraphs and can be used to characterize the network and its functionality. The expression of subgraphs in the original graph in terms of their frequency can help to create a network fingerprint and classify different networks according to it. When a particular subgraph appears in the graph more than

expected (comparing to similar randomly generated networks), it is called a motif[29]. Since the concept was created, there have been a wide range of applications of it, namely in social and information networks for finding string links in ranking tasks for heavily populated platforms like twitter[42], or to learn about user behavior by analyzing the most influencing posts[32], and also in biology with co-regulatory networks for tasks like looking for molecules and genes with similar functions[26]. In this case the presence of certain patterns (or motifs) can be translated into a specific functionality: for example a graph loop can be represented as a logical circuit managing the network's dynamic[45]. But the applications go beyond the direct use of these subgraphs to describe a specific network, they have also been used for more general modeling tasks like classification[48], community discovery[51], outlier detection[2], link prediction[41] and even in computer vision[57].

Although there has been a few different suggested ways of identifying a motif within the high frequency subgraphs, it is always necessary to find and enumerate them in the network first. This naturally concerns the Subgraph Census Problem(SCP), closely related to the subgraph isomorphism problem, a well known NP-complete problem[12, 21]. For this reason and due to the combinatorial explosion of different subgraphs with the increase of its size (for a simple directed subgraph of size $k = 3$ we have only 15 possible configurations, for $k = 5$ we get 9578 and 1540421 for $k = 7$[35]), most subgraph enumeration algorithms are bounded to smaller sized subgraphs of topological (directed or undirected) networks, without labeled nodes or edges.

Colored Motifs

Considering this, there is a lack of algorithms capable of expressing more features of the data that can be represented in the network as labeled/colored nodes and edges. Representing the elements of the network with different colors will necessarily increase the combinatorial arrangements for possible different subgraphs that need to be found, hence making the problem even more difficult. So one could ask: why should we bother, and what can we learn from it?

When we examine only topological motifs, which disregard the labeling of the nodes and or edges according to their nature and different roles, we may be identifying structures with different functionality and meaning as the same. This is very much the case for any metabolic network, where nodes can either correspond to chemical compounds or reactions that need to be modeled as input/output, and we lose data by not representing these systems as bipartite networks[45].

In fact, research has shown that in networks with different types of functioning entities, colored motifs have more information content than purely structural motifs and can be used as well to define the information content of the network itself[1], having the potential of giving a more specific and complete network fingerprint. Most of the work done on these labeled and structural motifs, which we will call from now on as colored motifs, has had applications in Biology to protein[9, 30] (Figure 1.1[39]), metabolic[21, 45], neural and gene[1] networks, but it can really be applied to any type of heterogeneous networks. Different color nodes can represent

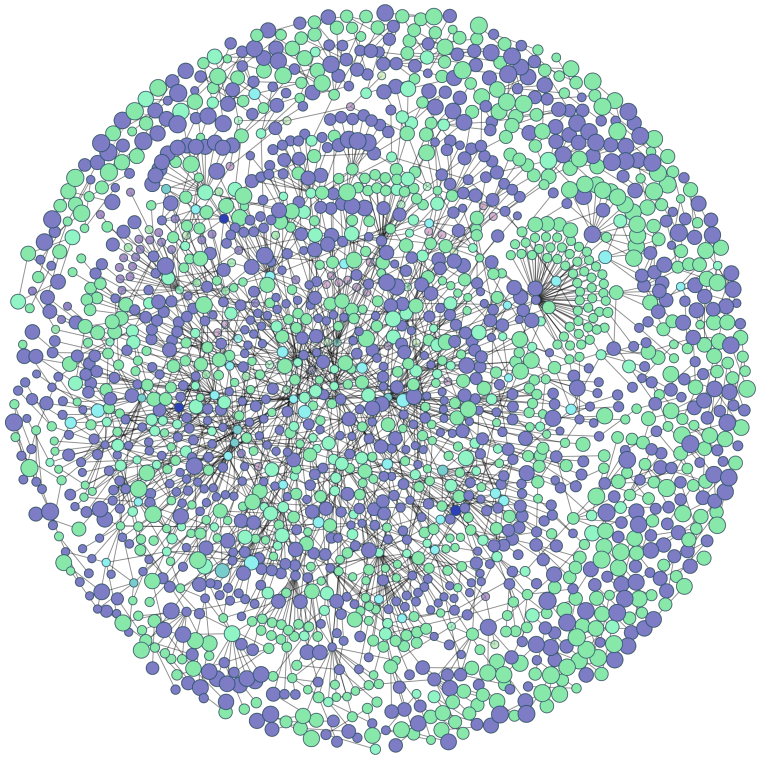

Figure 1.1: A biological yeast protein-interaction network[39].

the female and male gender in social networks and the edges can be colored according to the type of family relationships (father, spouse, brother), or for example in e-commerce we can have nodes labeled as products and clients.

With this in mind, this work aims to bring attention to the potential of colored motifs in real-world problems and go further with the development of a new tool for efficiently finding these structures in different networks.

1.2 Goals and Contributions

Our work revolves around the SCP and our main contribution is the development of an algorithm that solves this problem for the case of colored networks, through the enumeration of colored subgraph patterns. As we will see, over the last few decades there have been some efficient subgraph enumeration algorithms for the SCP, but very few have the capacity to include information on the types of nodes or edges in networks, a feature we called color. To our best knowledge, there is no available tool or algorithm that performs a full exact (or sampled) enumeration of subgraph frequencies in networks with both colored nodes and edges, besides from *FANMOD(ESU)*[55].

We discuss the relevance of finding patterns with color information, and the challenges faced in this type of approach, concerning for instance graph representations, the Graph Isomorphism

Problem and motif definition the for colored case. The implementation resulted in the algorithm *ColorFaSE* (from COLORed FAst Subgraph Enumeration), an efficient way of solving the Colored Subgraph Census Problem(CSCP). This was done by extending the work done in *FaSE*[33], one of the fastest state-of-the-art approaches to the SCP, to support color information for both the nodes and edges. This method is yet unpublished work, and it is our intention to make our code available to the general public, so that the network science community can benefit from it.

Lastly, we aim to perform a series of experiments, putting our code to test in real networks and also compare its performance regarding time and memory, with the algorithms that are publicly available, and we considered to be possible competitors for *ColorFaSE*.

1.3 Thesis Outline

Chapter 1: Introduction: In this first chapter, we first present the field of study where our work resides, the applications and the motivation behind the problem we are trying to solve and the main goals and contributions of this thesis. Finally, we also describe the thesis outline, throughout its chapters.

Chapter 2: Preliminaries: Here we introduce all the graph concepts and problem definitions, which we consider to be necessary to understanding the problem we are tackling and to the later description of how we approach it. These explanations are paired with a few illustrative examples to help understand the differences between classical and colored graphs. We end the chapter by addressing the state-of-the-art on subgraph enumeration and subgraph patterns in colored networks, describing some of the most well known subgraph census algorithms and discussing the placement of our method amongst them.

Chapter 3: Approach: This chapter consists in the discussion of the ideas and problems taken in consideration in order to implement the algorithm, as well as the description of the algorithm itself with the aid of pseudocode and illustrations of some steps of the approach.

Chapter 4: Experimental Analysis- In chapter 4 we perform a series of experiments to see how well our method behaves for different types of real and synthetic networks, and compare its performance with a possible competitor. We also develop some studies on models for colored networks generation and how color impacts the performance.

Chapter 5: Conclusions: Lastly, we summarize the work done in this thesis and turn to the possible adjustments that can be made to improve and expand our work, as well as some ideas that can be further explored in future work.

Chapter 2

Preliminaries

This chapter has the purpose of setting the terminology and background theory that we are going to work with throughout this thesis. We start by defining the basic terminology and notation around fundamental graph concepts in section 2.1, which are essential to the understanding of the following sections. In section 2.2 we move on to the definition of some important problems such as the graph isomorphism problem, subgraph census and motif discovery for both purely topological and colored networks. Finally, in section 2.3 we go through relevant work and the developments done related to the subgraph census problem and motif discovery in colored networks.

2.1 Graph Terminology

A **graph** G is a structure made of a pair (V, E), where V is the set of **vertices** in G and E is the set of **edges** connecting pairs of vertices in G, such that the vertices are called the **endpoints** of edges. The **size** of the graph $|V(G)|$ is determined by the number of vertices, and a graph of size k is also denoted as a k-graph.

A graph G is **undirected** if $\forall v_1, v_2 \in V, (v_1, v_2) \in E \Leftrightarrow (v_2, v_1) \in E$, meaning that there is no difference between the endpoints of the edges, otherwise the graph is **directed** and the edges have endpoints identified as source and destination, and can also be called a **digraph**. The **degree** of a vertex v is the number of edges in the graph which have v as one of its endpoints (Fig.2.1). For digraphs, we can differentiate between **outdegree**, the number of out-edges, and **indegree**, the number of in-edges. The **degree sequence** of a graph, is the non-increasing sequence of its vertex degrees.

A **path** $P(v_1, v_2)$ from node v_1 to node v_2 is a finite sequence of nodes and edges in G that traverse the graph beginning in v_1 and ending in v_2 (without visiting the same node more than once), and the length of the path $l(P)$ is given by the number of edges in this sequence. A graph G is connected if $\forall v_1, v_2 \in V(G)$, $\exists\, P(v_1, v_2)$, and the same applies for directed graphs by considering the all the edges as if they were undirected (Fig.2.1).

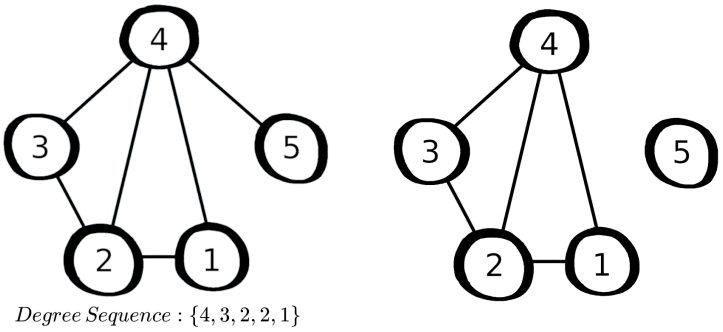

Figure 2.1: A connected 5-graph and its degree sequence on the left and a disconnected 5-graph on the right

Edges can have a numerical feature called **weight**, and in this case we have a weighted graph. A **self-loop** is an edge that connects one vertex to itself. A **multi-edge** is a set of identical edges connecting the same vertices. A simple graph is one without any self-loops or multi-edges. In this thesis, we will only consider connected simple graphs with no weighted edges, but our implementation can also support self-loops (Fig.2.2).

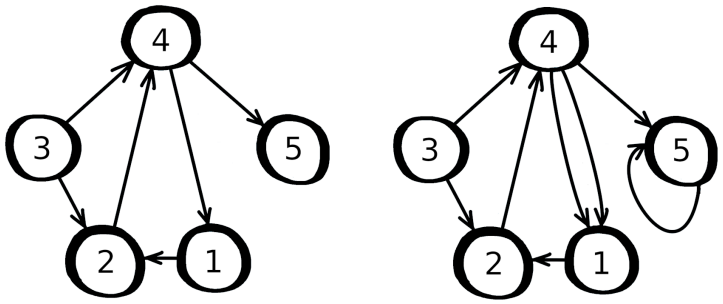

Figure 2.2: A simple directed graph on the left and a general graph with one self-loop and a multi-edge on the right.

In this work, the graphs are considered to be labeled so that every vertex is identifiable by an integer in the interval $[1, |V(G)|]$. This **label** of a vertex, $L(v)$, should not be confused with the notion of **color**, $C(v)$, which can be applicable to vertices and also edges. In our own definition, the color of a graph component (vertex or edge) can represent any type of feature we wish to incorporate to better model the nature of the system at study into the network. For instance, different color vertices can represent different genders and different color edges can identify different types of relationships between people in a social network. However, we will not focus on the meaning of this feature for now, but rather extend previously known definitions and problems to include the concept of color, which is the main focus of this thesis. A graph is considered to be colored if we associate a set of colors (or some type of label that can be mapped into a color) to its vertices and/or edges, and a non-colored graph can be understood as the simpler case of having only one color nodes and edges. Although it is possible for a node to be assigned with more than one color, in this thesis we only work with graphs that have a single color associated with each vertex and edge.

Note: The terms **node** and **vertex** are used interchangeably, as well as **link** and **edge**. The term **label** will be used from now on to denominate the identification or tag of the vertices, and not to mention the color of a vertex or edge. The terms *network* and *graph* are also used interchangeably throughout the text.

The **color degree sequence** of a colored graph is defined as the set of degree sequences for each different node color, partitioning them between each different edge color. This can be done in more than one way, and therefore we can have different sequences of the same set of numbers for the same graph, depending on the order of colors for the vertices and edges we choose. In Figure 2.3 we have an example of this color degree sequence being computed in a colored graph with three different vertex colors and two different colors for edges (in this case we considered the "total" degree and disregarded direction, not discriminating between the indegrees and outdegrees of the vertices). The color degree sequence on the bottom right was obtained by computing the number of connections of each edge color for each vertex, so its length is given as $k \times c_e = 12$, where k is the number of nodes and c_e is the number of edge colors. There are two purple(vertex color 1) vertices (vertex 1 and 4), one has two dark purple dashed edges and the other has none, putting this in non-increasing order we get $\{2, 0\}$, and they both have one blue link each $\{1, 1\}$, concatenating we have $\{2, 0, 1, 1\}$. We do the same thing for the single orange vertex we get $\{1, 2\}$ and for the blue ones $\{1, 1, 1, 2, 1, 1\}$, and joining this sequences together we get the full color degree sequence represented in the image ($\{2, 0, 1, 1, 1, 2, 1, 1, 1, 2, 1, 1\}$). In each node partition, the edge colors are always represented in the same order, in this case: the blue links first and then the dark purple dashed links.

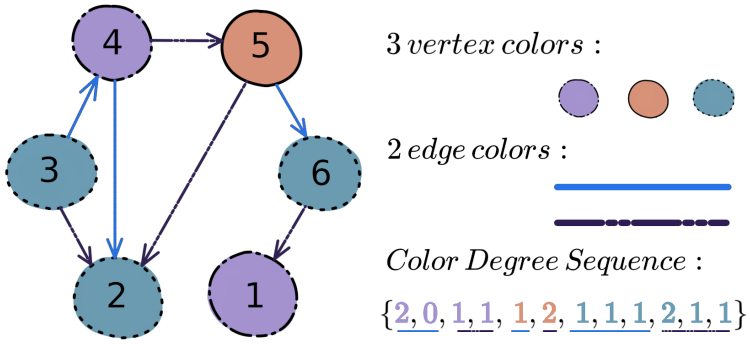

Figure 2.3: A colored digraph of size $k = 6$, with three different vertex colors ($c_v = 3$) and two different colors for the edges($c_e = 2$), and its color degree sequence labeled with different colors to illustrate how it was computed.

A **subgraph** G_k of a graph G is a graph of size k where $V(G_k) \subseteq V(G)$ and $E(G_k) \subseteq E(G)$, meaning that its vertices and edges are all in G. In this case, we can also say that G is a **supergraph** of G_k. Moreover, we say G_k is an **induced subgraph** of G, if and only if $\forall v_1, v_2 \in V(G_k) : (v_1, v_2) \in E(G) \Leftrightarrow (v_1, v_2) \in E(G_k)$, such that G_k must necessarily contain all edges in G that link any pair of nodes present in G_k. (Figure 2.4). In the context of this thesis, we will only consider induced subgraphs, therefore for now on the term *subgraph* will be used to

refer to induced subgraphs.

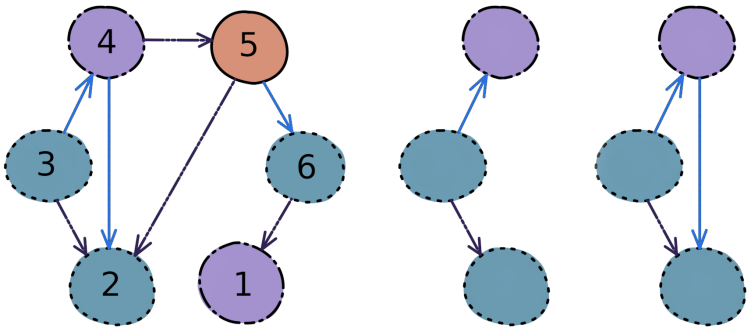

Figure 2.4: From left to right: A colored digraph G of size $k = 6$, with three different vertex colors ($c_v = 3$) and two different colors for the edges ($c_e = 2$), a non-induced subgraph of size 3 of G and an induced subgraph of size 3 of G.

The **neighborhood** of a vertex $v \in V(G)$ is given as, $N(v_1) = \{v_2 : (v_2, v_1) \in E(G) \vee (v_1, v_2) \in E(G)\}$ and the neighborhood of a set of vertices $S \subseteq V(G)$ is the set of vertices that results from the union of the neighbors of each vertex in S, which are not already a vertex of this set ($N(S)$). The **k-neighborhood** of a vertex v is the subgraph induced by all vertices that are at a **distance** smaller than k from v, where this distance is given by the number of edges in the minimal path that connects the two vertices. The k-neighbourhood of an edge is the union of the k-neighbourhoods of both nodes in the edge, excluding them. The **exclusive neighborhood** of a vertex v relative to a set S is defined as: $N_{exc}(v, S) = \{v_1 : v_1 \in N(v) \wedge v_1 \notin N(S) \wedge v_1 \notin S\}$.

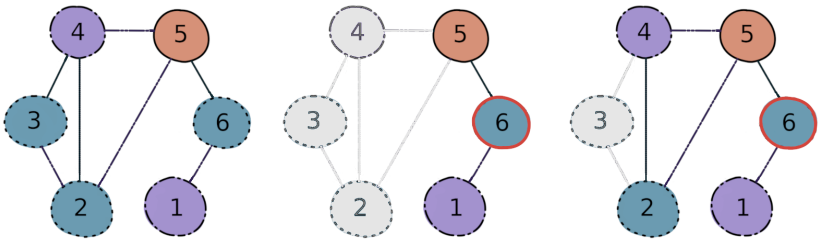

Figure 2.5: From left to right: A colored 6-graph G of size $k = 6$, with three different vertex colors, the 1-neighborhood of the vertex 6 of G, the 2-neighborhood of the vertex 6 of G (other vertices and edges outside the neighborhoods in grey).

Two graphs G_1 and G_2 are **isomorphic** ($G_1 \cong G_2$) if there is a **structure-preserving bijection** between them such that the number of edges between every pair of different vertices in G_1 equals the number of edges between their corresponding vertices in G_2. If two subgraphs H_1 and H_2 of size k of G are isomorphic, they are occurrences of the same pattern S_k and are said to belong to the same **isomorphism graph class**, $S(H_k, G)$. The **frequency** of a given subgraph H_k in G, is the number of subgraphs of G that belong to that class and is represented as: $F(H_k, G) = |S(H_k, G)|$. In Figure 2.6 we have an example of the isomorphism classes, subgraphs and their occurrences in two different graphs, a simple undirected graph, and a simple undirected graph topologically equivalent but colored. This image goes to show the combinatorial explosion

of different isomorphism classes of subgraphs having a much larger expression when we had color to a graph. An **automorphism** is an isomorphism from a graph into itself. **Graphlets** are induced non-isomorphic subgraphs.

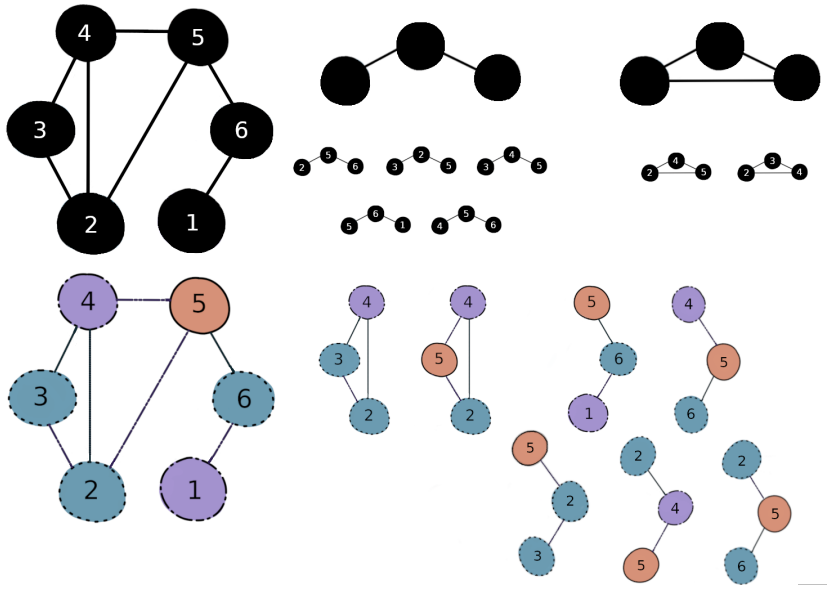

Figure 2.6: At the top part of the image: a simple undirected graph of size 6 and its two subgraph types of size 3 with their occurrences: a chain with five occurrences and a triangle with two occurrences, both with their respective labeled isomorphic subgraphs occurrences bellow. At the bottom: a colored graph of size 6 with the same topology as the previous one. For this one, we have seven different isomorphism graph classes of size 3, instead of two.

A subgraph H_k is called a **motif** of a graph G if it appears with a significantly higher frequency in G that it is expected with reference to a null model. This null model usually consists in counting the frequency of the subgraph in a large number of randomly generated networks which are similar to the original graph G, maintaining some of its properties like the degree sequence. This method and concept will be discussed in more detail in the next section.

2.2 Problem Definition

In this section, we present more formally some relevant problems necessary for achieving the goal of this thesis, finding motifs in colored networks. We start by addressing the Graph Isomorphism Problem(GIP) and its related variants. Then we introduce the subgraph census problem, and finally we talk about the task of Motif Discovery.

2.2.1 Graph Isomorphism

The Graph Isomorphism Problem is very relevant in any task that involves enumerating subgraphs and computing their frequency, since in these we need to ask if the occurrences we found in the

graph correspond to the subgraph being enumerated.

Problem 1 (Graph Isomorphism): Given two graphs G_1 and G_2, determine if there exists a bijection between $V(G_1)$ and $V(G_2)$ such that two vertices in G_1 are adjacent if and only if their corresponding vertices are connected in G_2.

An isomorphism is an equivalence relation, such that it is reflexive, symmetric and transitive. Therefore, given any three graphs G_1 , G_2 and G_3, we have that if $G_1 \cong G_2$ and $G_1 \cong G_3$, then $G_2 \cong G_1, G_1 \cong G_3 \Rightarrow G_2 \cong G_3$. Figure 2.7 shows four isomorphic graphs, where G_1 and G_2 are represented in different ways but are essentially the same graph, having the same edges between the same pairs of nodes. Similarly, G_3 and G_4 are also equivalent. On the other hand, although G_1 and G_4 look similar, their nodes are labeled differently, and in order to know they belong to the same isomorphism class we need to check each pair of nodes in each and see if they are equivalent, which in this case they are. In fact, in these graphs the vertices as well as the edges are all of the same type, and each vertex has 2 links, so any permutation of vertices that preserves this feature will result in an isomorphic graph.

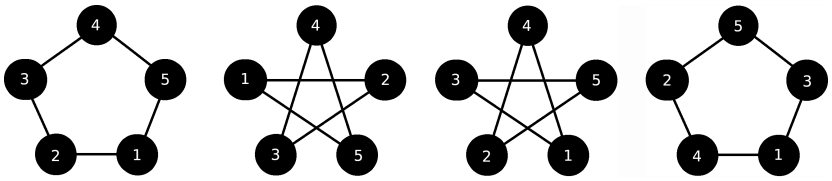

Figure 2.7: From left to right: three simple undirected 6-graphs G_1, G_2, G_3 and G_4.

When we deal with colored graphs, we also have to pay attention to the colors of the nodes and links we are comparing when checking if there is a bijection. As an example, in Figure 2.8, we have 4 graphs topologically equivalent to the ones in Figure 2.7, but with colored nodes. In this case, each graph has an equal number of nodes of each node color, and apparently analogously to the previous example they could all be isomorphic, but that is not the case. We have again that the pairs G_1, G_2 and G_3, G_4 are equivalent, but there is no bijection between, for instance G_2 and G_3 that preserves vertex adjacency according to their colors. This is easily shown by looking for the blue node labeled with number four in G_3 which has one edge with another blue node, and a one edge with the orange node, while neither of the blue nodes (labeled with three and four) in G_3 have these connections (node four is linked to two purple nodes, and node three is linked to a purple and a orange node).

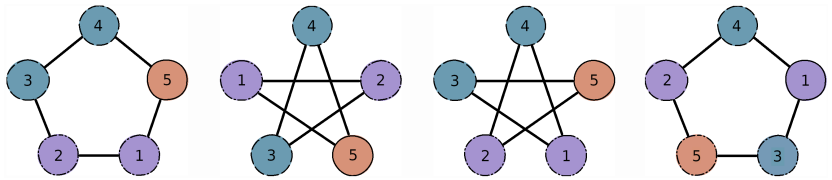

Figure 2.8: From left to right: three simple undirected colored 6-graphs G_1, G_2, G_3 and G_4.

When G_1 and G_2 are two labeled graphs with the same number of nodes and edges (and

colors), it is feasible to check if there is a structure and adjacency preserving bijection between them. We only have to compare the ordered node pairs that define their edges. This is however not the case with unlabeled graphs, because we then have $N!$ different possible ways to label the N nodes of the graphs, and this task would become too difficult for slightly larger graphs, and there are still no algorithms that solve this problem in polynomial time. In our work, when looking for subgraphs we are not interested in their labels for the enumeration, and so we are dealing with this exact problem. Thankfully, there is one well known algorithm which solves this task in exponential time and has a special feature that is very relevant for this thesis, which is supporting colors in the vertices (which we will further discuss later). This algorithm, called nauty[27], works with several heuristics and not only answers to the question if G_1 and G_2 are isomorphic, but is also able to retrieve a **canonical label** for an isomorphism class.

This refers to Graph Canonization(GC), which is another problem very important in our work and associated with the GI problem. Generally speaking, it focuses on finding a single labeling (or canon) for all the possible graphs of the same isomorphism class. In other words, this canon must be such that for any permutation $\pi \in Aut(G_1)$, the graph resulting of that permutation has the same canonical label of G_1. This problem, can even be seen as a variation of the GI problem, since in order to know if to graphs are isomorphic we just have to compute their canonical label and then compare them, if the canon is the same, then they have to be in the same isomorphism graph class.

Problem 2 (Graph Canonization): Given a graph G_1, find a graph labeling of G_1, called a canonical labeling (or canon), such that any graph G_2 can have the same canonical labeling as G_1, if and only if G_1 and G_2 are isomorphic.

Finally, the Induced Subgraph Isomorphism problem is another variant of the GI problem, and one that is also related to the task of subgraph counting, since it aims to find if a given graph H is represented as an induced subgraph in another graph G. When counting induced subgraph frequencies, our main original network is G, and each size k induced subgraph is corresponds to H, which occurrences we wish to find. This is a computationally NP-Complete problem[12] for general graphs.

Problem 3 (Induced Subgraph Isomorphism): Given a graph G and a graph H, determine if there is an induced subgraph in G that is isomorphic to H.

All the problems and concepts described before are almost trivially extendable to a graph with colored nodes and/or edges.

2.2.2 Subgraph Census Problem

The subgraph census problem is where we will focus much of our attention in this thesis and consists in computing the frequency of all different connected induced k-subgraphs in a given

network.

Problem 4 (Subgraph Census): Given a graph G and an integer k, determine the frequencies of all its connected induced k-subgraphs. Two occurrences are considered different if they do not share at least one node.

In this context, we established that two occurrences of a subgraph are counted as different if there is at least one node represented in one but not the other. There are other possible variants of this problem where the search is around a particular set of relevant subgraphs or subgraphs with different sizes, but our goal is to enumerate all the k-subgraphs represented in the network, in particular that can have colored edges and/or nodes. Therefore, the size of the subgraphs we are looking for as well as the presence of color in the network will largely contribute for the combinatorial explosion of the different possible types of subgraphs, and also for the number of necessary isomorphism tests and canonizations.

2.2.3 Motif Discovery

Motif discovery wraps up the set of problems relevant to this work, and is also a major possible application task to the results of any subgraph census algorithm.

Problem 5 (Motif Discovery): Given a graph G and a set of size k induced subgraphs S_k of G, find which of those subgraphs are overrepresented in G.

After having determined the frequencies of each isomorphism subgraph class of size k in our network, finding which of the subgraphs appear more than expected in an interesting problem and can help us describe the network. The subgraphs that are more frequent than it was expected according to some null model, are called motifs. This null model usually consists in generating a large set of random networks which preserve some properties of the original graph, in our case it would be the color degree sequence. Then we run the subgraph census is these networks, and compare the mean of the subgraphs' occurrences against the occurrences in the original study network, usually through a Z-score.

2.3 Related Work

In this section we go through the state of the art of subgraph census algorithms in general, and more in specific we discuss the work done for colored networks. We first look for the available subgraph counting algorithms, with or without color, and separate them according to their different searching approaches: Network, Subgraph or Set-Centric. Then we focus on the available methods for colored subgraphs or motif discovery and discuss their definitions of colored graphs and their range of application.

Note: In the following paragraphs mentioning previous related work, we will refer to the author's own terminology, so terms like subgraph, motif and graphlet can be used with the same meaning, as well as heterogeneous and colored networks.

2.3.1 Subgraph Census Algorithms

As previously mentioned, there are different ways of approaching the subgraph census problem in terms of the type of subgraphs we are counting. We can divide them in three main categories:

- **Network-Centric:** We count all subgraphs of size k (after enumeration we perform isomorphism tests to determine the subgraph class of each found occurrence);
- **Subgraph-Centric:** We focus on counting the occurrences of one specific subgraph in the network;
- **Set-Centric:** We search for a specific set of subgraphs.

We also have both **analytical**[28] (with statistical inference), and **enumeration** based algorithms. In this last category, we can have both **exact**[17, 18, 33, 54] or **approximate**[34, 44, 49] (with sampling) methods. In the last few years, we have seen a lot of procedures that use distributed systems and parallelization[24, 43, 46, 47, 49] to speed-up subgraph enumeration. We are interested in enumeration algorithms and within these we will be looking into both exact and sampling approaches, assessing the trade-off between accuracy and computation times. Our own approach can be called as network-centric, as we are reusing information when traversing the graph for different subgraph occurrences, but we only dedicate memory to for the subgraphs that are represented in the network. Due to the combinatorial explosion when we add color to nodes and edges, it is likely that we will not have every possible isomorphism graph class of a size k, and so it would be a waste of time and memory to partially consider every possible subgraph. We also have a method that is able to switch from a fully exact enumeration to a sampled one, where the percentage of the network's nodes visited can be adjusted as we wish, as in Rand-FaSE[34].

Regarding the state-of-the-art on the SCP, in specific the most important known to date algorithms, we will now briefly describe some of them: their methodology, limitations and how they are related with our work. Considering the previously discussed types of approaches, although any of three can solve the SCP, we will mostly focus our overview in the ones that are network-centric, as it is the case of our own approach. Nevertheless and to be clear, we should mention that the SC can be performed on a network via a subgraph-centric approach like Grochow [13], by simply running the algorithm each time for every possible isomorphism k- class (similarly we can apply this to set-centric approaches, if we considered the algorithm's input). A more complete review on the state-of-the-art algorithms for (non-colored) subgraph counting can be found in this survey[35].

ESU

One of the most widely used algorithm for subgraph enumeration is still *ESU*[54], a network-centric approach that visits each k-subgraph occurrence of the graph only once, introduced by Wernicke. In order to go through each vertex only once, *ESU* keeps two vertex lists while performing DFS and backtracking operations to traverse the graph.

One list, V_S contains the vertices of the current (induced) subgraph (always in increasing order), and V_E is the list of neighbours of the vertices in V_S that are still in conditions to be visited next. To run the census, *ESU* starts by setting $V_S = \{v_1\}$ and $V_E = N(v_1)$, for a given vertex v_1. Then, it recursively removes a neighbour vertex v_2 from V_E (from left to right) and sets the values for the two lists in the following depth by adding v_2 to the subgraph list, $V_S = V_S \cup v_2$, and by adding the vertices in the exclusive neighbourhood of v_2 relative to V_S that have higher label than the first node of V_S that was enumerated in V_S, this is: $V_E = V_E \cup \{v_2 \in N_{exc}(v_2, V_S) : L(u) > L(v)\}$. This process is repeated for each node in the original graph until we have all possible different k-subgraphs occurrences, and by removing the visited vertices from the list, and using their exclusive neighbourhood, we guarantee that no subgraph or even node is being visited more than once.

To serve as an example of this implementation, in Figure 2.9, we have the full content of the these two lists being updated as we go deeper, or jump back up in the graph BFS. We have a size $k = 6$ uncolored undirected graph (the same depicted at the top of Figure 2.6), and *ESU* performs the subgraph census to find subgraph occurrences of size 3. The first node to be visited is node 1, and there is only one possible subgraph occurrence involving this node ($\{1, 6, 5\}$).

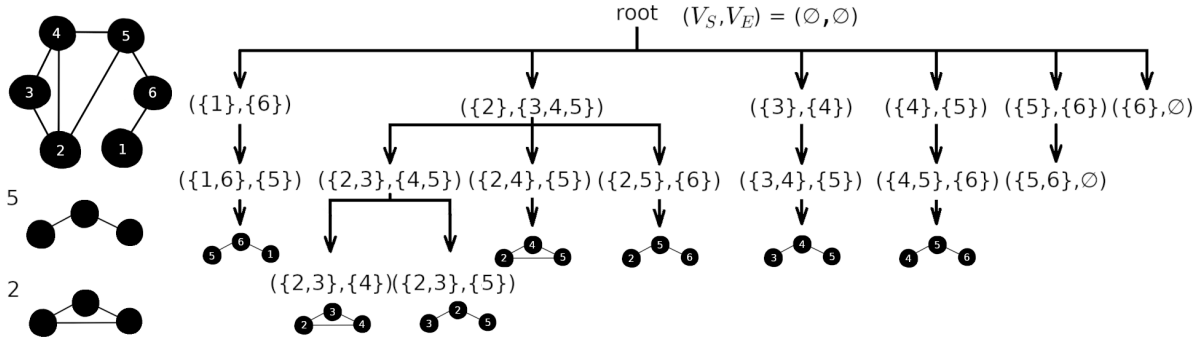

Figure 2.9: The search tree and the two main lists V_S and V_E for the *ESU* enumeration process applied to a 6-graph and different occurrences and frequencies of the found isomorphism classes for subgraphs of size 3.

After finding this subgraph we jump back to the second node to be visited, node 2 ($V_S = 2$) with neighbours $V_E = \{3, 4, 5\}$. In this branch of the tree, we can notice that when the subgraph $\{2, 4\}$ is being enumerated, despite the fact of the node 3 having a connection with the node being added to the subgraph (4), it does not feature V_E, and this is because this node has already been removed from V_E when we enumerated the subgraph $\{2, 3\}$ and 3 also has a connection with the first node added to the current subgraph (2), and therefore is not part of the exclusive

neighbourhood of node 4 relative to $\{2\}$. Another relevant example can be noticed in the enumeration of both the subgraphs $\{5, 6\}$ and $\{6\}$, where the neighbours list is an empty set, $V_E =$, since there are no more nodes with higher label than node 6. Finally, we can observe the seven different size 3 subgraph occurrences found by the algorithm, corresponding to 2 isomorphism subgraph classes: a triangle with frequency equal to two, and a chain with frequency equal to 5. To compute these frequencies, *ESU* needs to perform isomorphism tests for each and every found occurrence, calling nauty[27] to find the isomorphism classes.

Kavosh

Kavosh[17] is another network-centric subgraph census algorithm, that is similar to *ESU* in the sense that it also visits each node only once, and it uses nauty[27] as well to perform isomorphism tests. In this implementation, the vertices are also chosen in increasing order, according to their label, but this time we have a tree that is rooted in each vertex and this vertex is only removed from the list when we have all the subgraphs that include it represented in its tree. The size k subgraph enumeration is done with this tree of $depth = k$ and is based on the neighbourhood of each vertex. To descend in the tree, a child neighbour is chosen, provided that this node is not represented in upper levels of this tree and also has higher label than the root node, and when we reach the full depth, the tree is ascended and the process is repeated and this time, nodes that were visited in other paths that have already descended the tree and been enumerated, are considered unvisited for this tree branch.

FaSE

The latest and most relevant algorithm for our own work, is the approach developed by Paredes and Ribeiro, called *FaSE*[33] (Fast Subgraph Enumeration). *FaSE* expands and improves the work done on subgraph enumeration with the *G-Trie*[37] (Graph reTRIEval) data-structure. This data-structure, which we will cover in further detail later, is a multiway tree that is used to encapsulate information on subgraph sets. Each node of the **g-trie** stores the edges of the vertex that is being inserted in the tree to the ancestor vertices already in the upper level. This way, we can use the fact that all descendants of a node share a common subtopology, to reuse information when traversing the graph.

> **Note:** In the following sections, to avoid ambiguity when addressing a g-trie structure, the term **node** will be used to refer to the **g-trie** tree nodes, and the term **vertex** for mentioning actual graph vertices that are being stored in the g-trie.

FaSE also uses a g-trie, but instead of already having the g-trie built from the with all the possible paths to every possible k-subgraph before the enumeration process, the g-trie is built as we traverse the graph and throughout the enumeration process. This means that we only insert nodes in the g-trie if there is a path to a corresponding occurrence in the network we

are working with, saving time and memory, especially in cases where there are many possible subgraph classes and configurations and only few of them are present.

The algorithm works by first enumerating all the subgraph occurrences, and then identifying isomorphic classes to compute the final frequencies. For this enumeration process, any algorithm that works incrementally, this is, by adding one vertex at the time to the current subgraph being enumerated, can be used as it is compatible with the way we store new information in the g-trie data-structure. Therefore, both *ESU* and *Kavosh* can and are considered in this approach. We will focus on the implementation using *ESU*, since it is the algorithm that we are also going to use as part of our method.

As we have seen, *ESU* creates a recursion tree where each node of the tree has two lists of vertices, one with the current subgraph and another with the vertices that can still be visited in the next level of the tree. *FaSE* is able to use this recursion to build a canonical label (LS-labeling) and add it to the g-trie structure during the enumeration. The process of adding a new vertex v to the subgraph being enumerated is similar as in the g-trie original method, we just add a new g-trie node with labeling that represents the connections to ancestor vertices already in upper levels of that g-trie path. If this node, with the same label, already exists, we add 1 to its frequency count. The leaf nodes of the g-trie will be at depth k and represent a subgraph occurrence.

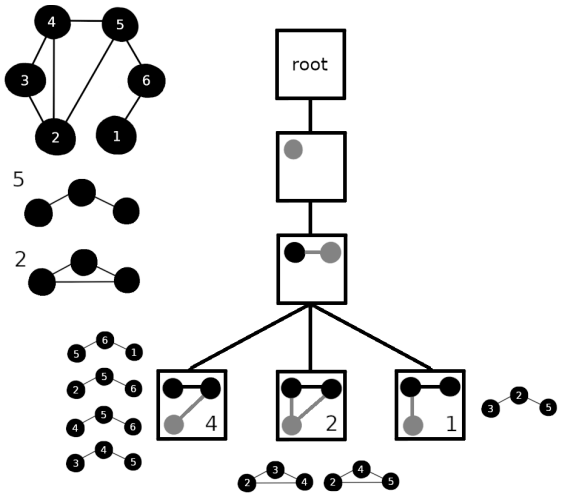

Figure 2.10: The g-trie that results of the enumeration of the 6-graph on the left with the *FaSE* algorithm. The vertex that is being added and its connections to previous vertices are represented in the tree node with grey color, while the previous elements of the subgraph are in black.

One of the main advantages of this algorithm compared to the previous ones is the much smaller number of isomorphism tests that it needs to perform. Since *FaSE* is able to reuse information and store more subgraphs with common topology, and we only need to call nauty to perform GC on the leaf nodes (there is no need for isomorphism tests during the actual enumeration process). Having a canonical label for each leaf node, computing the isomorphism of

the occurrences is much simpler, and to compute the frequencies we just need to add the counts of subgraphs with the same label.

In Figure 2.10 we have the subgraph enumeration of a graph (the same one as in Figure 2.9) being stored in the g-trie according to the *FaSE* algorithm. We can notice that the isomorphism subgraph class frequencies are the same as the ones found in the application of the *ESU* algorithm (Figure 2.9), but here we just need to perform three isomorphism tests, instead of seven, since we only have three leaf nodes. The enumeration recursion tree would be exactly the same, but here we store the information in the g-trie instead and perform the isomorphism tests only on the leaves of the g-trie. We build this g-trie as we traverse our graph according to *ESU*, so we need to know where we are in terms of depth in the subgraph enumeration, to navigate the right path in the g-trie as well.

2.3.2 Colored Motif Discovery Algorithms

Although the SCP as been the focus of many studies and algorithms in the last few decades, there is not so much work done regarding this problem for the case of colored networks. Here, when we mention "colored" graphs, we are not limiting our scope exclusively to the approaches that use this term. In this field, have seen works referring to these networks as "labeled", "heterogeneous", "typed", amongst others. The available approaches also differ from their target pattern search and its own definition. Some of them focus only on the types (or colors) present in a subgraph, disregarding the structure[43], and others focus only on a particular set of colored subgraphs and focus on developing a faster algorithm for their search[15]. There are even a few approaches that search non-induced subgraphs, which is also not our target with this work[49].

It is important to mention that there has been some previous work done on the use of g-tries for subgraph counting. In [36], a solution to this problem as been proposed, however the approach is set-centric as uses a g-trie that needs to be pre-computed before enumeration, unlike ours that, as in *FaSE* grows as needed.

Fanmod (ESU)

FANMOD is an improvement of the previously discussed *ESU* algorithm, that allows the enumeration of subgraphs in a colored network. It also used randomized sampling or a fully exact enumeration process, and the method for determining the subgraph frequencies is, again, nauty through canonical graph-labeling. This algorithm can detect subgraphs up to size 7 (or less, depending on the number of node colors and edges).

Finally, this tool also provides the results for each subgraph significance, expressed with P-Values and Z-Scores that are computed with reference to a series of randomly generated networks. These networks are transformations of the original input network, obtained by switching edges according to some graph property, which the user can pick within a list.

Considering this, we will be using this approach to test our own approach against both in terms of performance and in correctness.

Since this tool makes use of the *ESU* algorithm for subgraph enumeration, we expect it to be slower than our own approach at identifying the subgraph types present in the network and their frequencies. This is because, as we have seen, using a g-trie structure allows us to reuse information and that leads to fewer leaves in our tree, and therefore fewer graph isomorphism tests to be done, comparing to an enumeration done with *ESU*.

Chapter 3

Approach

In this chapter we explain the ideas and procedures behind our algorithmic implementation of *ColorFaSE*. In the first section, we make some first considerations on the nature of the CSCP and how we decided to implement our solution. We also give a brief description on the overall structure of the code, before we explain some steps of the implementation in detail in the following sections. In section 3.2 we address the fundamental first step for any practical program of this kind: how to represent a graph with color, where we discuss the transformations needed to the original $FaSE$ algorithm. In section 3.3 we explain how we encapsulate and recover the subgraph topology information in the g-trie. This includes the description of the new type of labeling we created to support color information. In section 3.4 we discuss the CGI problem and how we obtain a canonical form for our isomorphism classes using nauty. Finally, in ColorFaSE we give an overview on our implementation and its features.

3.1 Subgraph Census in Colored Networks

The CSCP is very similar to the original SCP apart from the extra information we have to take into account on the node and edge colors. If with simple, not labeled, networks we have an exponential growth of our number of subgraph occurrences with the subgraph size and network size, in the case of a colored network we will also face this problem, but with even larger dimensions. As the number of colors increases, we get larger and larger numbers of possible combinations that give origin to different subgraph. This means that we need to perform much more isomorphism tests, which as we know is a very time-consuming task. At the same time, in most cases with smaller to medium networks, it is not likely that every possible subgraph type appears. With this in mind, it was important to have an approach that tries to minimize the number of isomorphism tests done, by reusing information in a tree, as it is done in a g-trie. But also, we want to save memory and only create the necessary nodes and paths in our structure, since the initial g-trie for all possible subgraph types would be too big and probably unnecessary. Given this, rewriting and adapting FaSE to support colored networks was our option from the beginning, since it behaves well in usual networks and has is able to save time by reusing

information in the g-trie, and memory, by using a dynamic g-trie that grows as needed. FaSE is also a great starting point, given that we are familiar with its method and have more possibilities of improving and inventing new features on it.

We decided to reuse the structure behind *FaSE*, in particular a version featuring the possibility of performing approximate enumeration through uniform sampling, *Rand-FaSE*[34] (RANDomized FAst Subgraph Enumeration). This means that, as in $FaSE$, our algorithm can be broken down in the following steps:

- **Enumeration:** We visit the vertices one at the time, keeping two lists, one with the vertices that have been enumerated (induced subgraph), and another with the vertices that can still be visited. These lists are updated every time we go a level down in the search tree, and the procedure for these list creation and update methods are the same as the ones described in *ESU*. This way, while still doing a DFS along the graph, using the approach of *ESU* we make sure not to visit the same node twice or enumerate the exact same subgraph occurrence more than once.
- **Encapsulating subgraph information:** This is a crucial step in our approach, since the way we store and reuse information while traversing the graph is directly associated with not only the memory we are going to have to allocate to keep track of all the found occurrences, but also on the running time of the procedure. This is because the following step (isomorphism tests) is computationally very heavy and having more paths in our search enumeration tree, means that more GI operations are needed. In this approach, we only add a new node to the tree if there is no path to a node with the same labeling as the current one. This labeling, has the information about the vertex that is being visited, in specific its color and the color of the edges it has with the already enumerated vertex in the subgraph (as well as the absence of any vertex). We will explain this process later in this chapter.
- **Isomorphism Testing:** The final step to complete the SCP is to compute the frequencies of each isomorphism class subgraph of size k. To do this, we have to perform isomorphism tests on the leaf nodes of our g-trie, and add the counts of occurrences of isomorphic subgraphs. In our approach, we do this by calling the *nauty* GC procedure for node-colored subgraphs to return a canonical labeling of each leaf node of the g-trie. And this way computing the frequency is trivial, as isomorphic subgraphs should have exactly the same canonical label. Since nauty only supports colors on the vertices, we need to transform our vertex and edge colored graph into a vertex colored graph that is readable for nauty, and process the canonization it returns into an isomorphic graph to the original one with colors on both edges and vertices.

In the following sections, we will discuss the fundamental changes we had to make to the original implementations, and discuss the challenges of representing and handling colors in our algorithm in an efficient way.

3.2 Colored Graph Representation

Knowing how to represent our graph and how our machine is going to read and change it is an essential part of any subgraph enumeration implementation. As previously mentioned, the basis for our approach is the *FaSE* algorithm, and in this method, the graph is given as an edge list and is most of the time represented in an adjacency matrix. This matrix M is square has the size of the graph $G(N)$, $N * N$, and each entry represents a connection between two vertices. The matrix is filled with boolean values such that, $M[v_1, v_2] = 1$ if $\exists(v_1, v_2) \in E(G)$, if there is an edge in G from v_1 to v_2, otherwise $M[v_1, v_2] = 0$.

All the methods that perform any kind of operation on a graph or subgraph in $FaSE$ are built on top of this kind of representation, using a class type *Graph*.It is quite easy to understand from the beginning that this type of representation, and a single matrix structure is not enough to express all the information we have in a colored graph (with both colored edges and nodes), so it is necessary to set our own edge and vertex color definitions and update the algorithm to be able to use a new graph representation structure.

The first decision we have to make, is how to represent color in the implementation, both for a vertex and an edge, it might sound trivial, but this representation will have a big impact in the running times and memory in the later stages of the algorithm. We decided that both the types(colors) of nodes and edges would be identified by an integer number, starting from 1 and ending in the total number of node colors, c_v, and node edges, c_e, respectively. This means that a node can have the same integer for its color as an edge, and we do this to save memory in future procedures without loss of any information, since there is no original correspondence between the colors of the nodes and the edges. Notice also that we start coloring with number 1 and not 0, this is because 0 is going to be used to identify the absence of a connection between two nodes.

Matrices are structures that provide fast access to information, provided that we know where this information is located through its indices, which is the case of these graph representations, as we do have an integer label that identifies that node and has a direct correspondence with its positions in the matrix. This label also starts at 1 and ends in $|V(G)|$, and should not be confused with the integer that represents the node's color. Given this, it only makes sense to represent our graph in a matrix as well, and one that stores the color information of the edges. This is done by simply replacing the boolean values by the integers representing node colors, and again we have that the matrix is filled with $M[v_1, v_2] = c(v_1, v_2)$ if $\exists(v_1, v_2) \in E(G)$, and $M[v_1, v_2] = 0$ otherwise, where $c(v_1, v_2)$ is the integer that represents the color of the edge between nodes v_1 and v_2.

After this, we are still missing the node color information in our graph matrix structure. The way we decided to solve this was by creating a list of node colors, with the integers representing each node color sorted in increasing order of the node labels. This list of node colors, paired with our new (colored) graph matrix, constitute all the necessary data structures to unequivocally represent any colored graph. The edge list for a colored graph, which is the main input of our

algorithm, is also easy to be obtained, we just add the integer of the edge color in a third column in front of each vertex pair, and again, paired with the list of node colors is able to represent all the information in a colored graph. We could also have store the colors of the nodes in the diagonal of our matrix, as it is done in *fanmod*, but this would mean we could not support self edges in our network. By having a list of node colors, we can represent self edges with colors as well if it is needed in those entries of our matrix.

In Figure 3.1 we have an example of a colored directed graph, with three node colors and two edge colors, and its representations both in an edge list and in a matrix, paired with a list of node colors, according to our method.

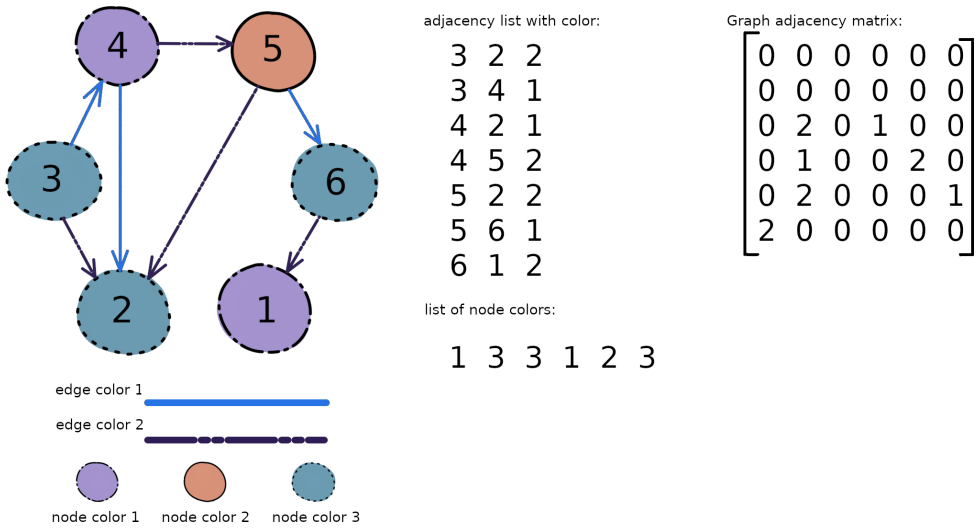

Figure 3.1: Example of two proposed ways of representing a directed colored graph: an edge list and an adjacency matrix, both paired with a list of node colors.

Given this, the first challenge in our implementation was to rewrite every procedure of the class Graph to support this representation. This class is fundamental throughout the enumeration process, since it is constantly used to get things like the number of neighbors of a node, a list of neighbors of a node, the color of an edge, the color of a node, etc. So it is important to have procedures that we can call at any time to perform this tasks in an efficient way.

3.3 Colored Subgraph Labeling

Our approach follows the method in *FaSE* which uses a g-trie to encapsulate information on the vertices being added and their connections to ancestor ones. This information is stored in a label for each tree node, which *FaSE* originally does with a binary representation of these connections. This label should have the ability of, when merged with the labels from ancestor nodes of a particular path in the g-trie, be enough to reconstruct the entire subgraph pattern that was being enumerated.

The label is also used as input to the nauty procedure that performs GC and returns a new

canonical labeling that represents the subgraph occurrence. Therefore, it is crucial that the labels we are encapsulating, have a clear color representation, and we need to make sure that there is a bijection between the subgraph matrix of the occurrence found, and the labeling used to store it in the g-trie.

3.3.1 Encapsulating Colored Subgraphs in the G-Trie

The presence of color in a graph is responsible for increasing the number of subgraph patterns. Different colored graphs will give different colored subgraph patterns, even if they have the same topological structure, as we have seen in section 2.1. Therefore, in our implementation we will end up having much more different tree (g-trie) nodes, with different labels, at the same tree level. This labeling is what determines in the enumeration procedure, if we need to add a new g-trie node or not, and we should be able to represent the subgraph that is being enumerated through the labels in its tree path only.

3.3.2 Colored Labeling

In order to try to keep the algorithm as efficient as possible, we decided to keep the same type of binary structure that was being used in $FaSE$, and represent this information with a fixed number of bits for each tree level. The label must represent both the color of the edges of the current vertex with the previous ones (in both directions, when the graph is directed), the color of the self-edge or its absence, and also the color of the vertex itself.

Given this, our label has a constant number of allocated bits for each tree depth, and can be broke down in the following steps:

- **Edge color information:** We first insert the color of the edges between the current vertex, v and the vertices, v_a already in tree path upper levels, as well as the absence of connection. The edges are inserted according to the order that the previous vertices were visited in the g-trie, and for the case of directed graphs we first insert the edge (v_a, v), and then (v, v_a). So we need to allocate $\lceil log_2(c_e + 1) \rceil$ bits for writing each edge color or the absence of an edge, hence the $+1$. And we have as many edge colors to be added as the number of nodes, or the depth we are currently in, times two if the graph is directed. Therefore, for representing the edges we need a total of $depth * \lceil log_2(c_e + 1) \rceil$ bits for an undirected graph, and $2 * depth * \lceil log_2(c_e + 1) \rceil$ for a directed one.
- **Self-edge color information:** We want to support the possibility of having self-loops, and these can have color as well. So next, we insert the color of the self edge if it exists, or leave the necessary $\lceil log_2(c_e + 1) \rceil$ bits as zero.
- **Vertex color information:** After this, we insert the color of the vertex that is currently being added, and for that we only need $\lceil log_2(c_v) \rceil$ bits.

In Figure 3.2 we have an example of the construction of this label for the third node (5) of the subgraph $\{2, 4, 5\}$, of the same graph represent in Figure 3.1, with three vertex colors and two edge colors. From right to left, the first two bits ($\lceil log_2(c_e + 1) \rceil = \lceil log_2(2 + 1) \rceil = 2$) are 0, since there is no edge from first vertex that was visited in the enumeration (2), and the vertex currently being added (5). Then, we have two bits that correspond to the color of the link $(5, 2)$, which is of color 2, and therefore is represented in binary by 10. The process is repeated for the edge pairs (4,5) and (5,4). And then we have 2 bits which are zero because the new node 5 has no self edge, and the last two bits ($_2(c_v)\rceil =_2 (3)\rceil = 2$) represent the color of node 5, in this case it was the second color, so we have 01.

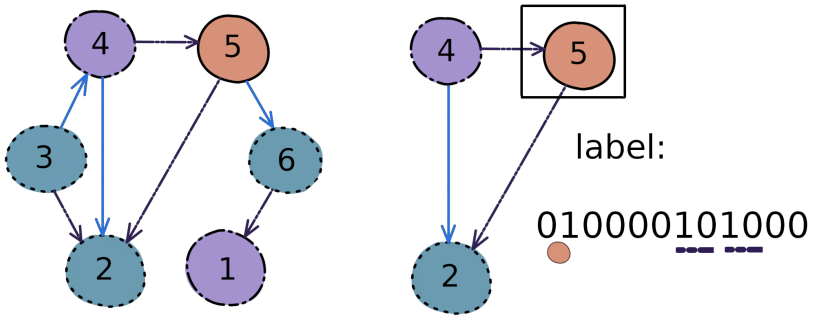

Figure 3.2: Example of the color labeling to be inserted in the g-trie for a subgraph of size 3, from a graph with three vertex colors and two edge colors.

3.3.3 G-trie Functionality

It is important to understand how the g-trie structure is being used to know how to store this label correctly. We previously had the case without color, where each new node label only needed as many bits as the *depth* of our method (times 2 for a directed network), because we only needed to represent the presence of an edge or its absence, so we would only have to write either a 1 or a 0, respectively in that position. This is not the case with our label, which with binary representation needs much more space to be able to store all the possible colors. To understand the reasoning behind our label construction and how relevant it is to our methods' performance, we need to dive into the g-trie structure implementation a bit further.

We have previously described how the g-trie used in *FaSE* works, and when a new g-trie node needs to be added, we can see an example of this in Figure 2.10. What we have not touched yet is how the g-trie really works in terms of its data structures and how in manages the allocation of memory. In the original method each g-trie node, is actually recursively represented in a series of nodes with constant size labels. Namely, for each node in the conceptual g-trie, the one we mentioned until here and have the example represented in Figure 2.10, we can have several nodes in the implementation tree, as much as needed to store the label. This makes it easier for the g-trie to perform the insert, update and recover operations, as it saves memory from being unnecessarily allocated. We do not need to know the exact size of each label, we just insert each

set of n bits in consecutive g-trie nodes until our label is fully represented, and we store the last identifier of the last node of the g-trie that represents the label and how many bits remained (to properly recover it later). This is done recursively by adding as many new nodes in the g-trie as necessary, and then returning the last node to the main method to continue the enumeration. To illustrate how this works, we show the pseudocode for the function that inserts new nodes in the g-trie.

Algorithm 1 Insert Label in G-trie Node

Input: a node identifier *labelNode*, the label to be inserted *label* and the number of bits that constitute our label *digits*
Output: the next g-trie node identifier to return to the main enumeration method
procedure: *INSERT_LABEL(labelNode, label, digits)*

if there is no path in the g-trie from the last n bits of the label, with the node labelNode as ancestor **then**
    if $numLabels = maxLabels$ **then**
        expand the gtrie (allocate more memory)
    end if
    $newNode \leftarrow numLabels$
    $numLabels \leftarrow numLabels + 1$
    if $digits < n$ **then**
        $labelLeaf \leftarrow true$
    else
        $labelLeaf \leftarrow false$
    end if
end if
$nextNode \leftarrow$ the node identifier of the child of the node labelNode for the label path
if *not* $labelLeaf$ **then**
    INSERT_LABEL(nextNode, label » n , digits - n)
end if
return $nextNode$ (back to the main enumeration method)

In our implementation, the input *digits* is calculated in the main method and given as: $digits(depth) = (d * depth + 1) * \lceil log_2(c_e + 1) \rceil + \lceil log_2(c_v) \rceil$, where *depth*, is the depth of the enumeration tree, the number of vertex being enumerated, and not the depth of the g-trie. And d is equal to 1 for undirected networks, and 2 for directed ones.

As we partitioned the label in several chunks of n bits, we can get two vertices that came from the same path and have different label representations, can be stored in the same g-trie nodes if they share common topology in the first connections, and therefore in the first few chunks of label. This can be very useful for saving information and makes it easier to navigate the g-trie up and down, since it becomes smaller. For this reason, we chose to have a bigger label that is

independent of any external information, written in binary representation, so it can be processed the same way in the g-trie methods, and that has fixed size with the depth of the main method.

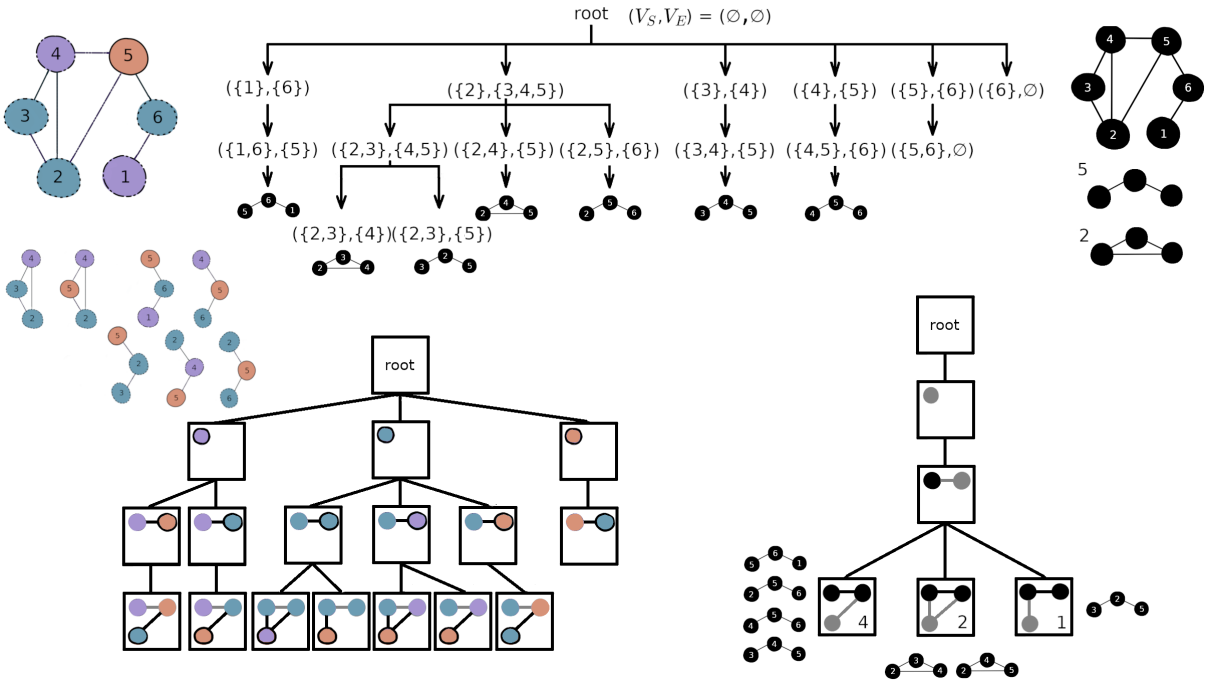

Figure 3.3: Example of two graphs with the same structure and therefore with the same enumeration tree, on the top. The colored graph and the resulting g-trie and subgraph types on the left, and the same for the graph without color on the right. For the colored tree, the vertex that we are adding in each node of the g-trie is identified with a black line as well as its new connections.

Another important detail to address, is the initialization of the tree. In the original implementation, the first vertex being enumerated in the first g-trie node, always had the same label. This is because *FaSE* does not support self-edges, and so there is no other necessary information to store about any first vertex, and there is no label for it in the original method. This is obviously not the case with our structure, even when disregarding any self-loops, we will have as many different labels as the number of different node colors. Figure 3.3 illustrates exactly this, where we have the same graphs as in Figure 2.6, both with exactly the same topology, but one is colored. The search tree, according to *ESU* is represented in the image, and since the graphs have the same structure besides coloring, the enumeration will go in exactly the same way. The g-trie however is very different, and since we have seven different subgraph types for the colored version (and seven occurrences only), we are going to have seven g-trie leaves, instead of three. In the first level, we have three different nodes, one for each color, instead of the typical one.

In our method, we call the function that generates the label starting from the first level of the main method (enumeration) and insert it in the g-trie. When recovering the total label, we

also need to take this into account.

3.3.4 Dynamic_Bitset version

The original label is implemented with a *long long int* datatype, so it can be processed very quickly with operations such has bit shifts, bitwise OR or any bitwise operation. We build our own label with the same datatype, for maximum efficiency, but since the presence of colors makes the label larger than the original one, for some cases the 64 bits of the *long long int* structure could be insufficient. For example, if we have a directed subgraph of size $k = 6$, in a network with $c_e > 3$ and $c_v > 2$, we need at least $((2 + 1) * 3 + 2) * 6 = 66$bits.

A possible alternative would be to use the *bitset* datatype available in the c++ *BoostLibrary*, which allows us to choose the number of bits we want for our variable, and supports the same, or similar operations to *longlongint*. But there is a major problem for our implementation with this approach, which is the declaration of variables at the time of compilation. In order to use the *bitset* datatype, we would need to know the size of the larger label before running our code, even when just compiling. Unless we would give a sufficiently large number of bits for every case, this method would not work. And since the operations with this datatype are known to be slower than with a *int* structure, we do not want to lose more efficiency by allocating unnecessary bits throughout the all process of enumeration, since even the first smaller labels would have this size.

The solution was to use another publically available datatype from the c++ *BoostLibrary*, that lets as declare the variables with parameters that are dependent on the input graph and initialize them while running the code. The *dynamic_bitset*, is a datatype that allows us to change the size of the label to just as many bits as we need. The operations are equivalent to the ones we use with the *longlongint*, but are costly, and have some limitations since you can not perform a bitwise operation between a *int* and a *dynamic_bitset*. Nevertheless, this adaptation is optional, and only used automatically by our method, for subgraph sizes and input graphs that need a label larger than 64 bits. This way, we are no longer restricted, in theory, by any number of subgraph sizes and colors.

3.4 Colored Subgraph Canonization

The last and fundamental part of the algorithm is to recover all the labels stored in the g-trie and their frequencies, and then, for each leaf, get a canonical label for the subgraph type it represents. This way, we just need to add the frequencies of the occurrences with the same canonical label, instead of performing isomorphism tests for every pair.

3.4.1 Recovering Subgraphs from Labels

After having the set of reconstructed labels and their frequencies, it is useful to process them to a subgraph form, since we have implemented many procedures for the *Graph* class type. This is easy to do, given that we always know the necessary information, like the number of edge and vertex colors and if the graph is directed, so we only need to iterate through the label. The following algorithm describes the function which does this task.

After this, we can start the canonization of our subgraph, which as we previously mention is done with a third party tool, *nauty*, so we first need to understand how it receives and returns graph information.

3.4.2 Calling Nauty for Graph Canonization

Nauty was the chosen algorithm to perform isomorphism tests, more specifically in this case to perform graph canonization, not only because it is still one of the most efficient algorithms to solve this computationally hard problem, but also because it has very relevant feature for our work. This is the fact that the available nauty tool, is able to perform on graphs with colored nodes, and to our knowledge there is no other algorithm that returns canonical labels on node-colored graphs.

Nevertheless, nauty still does not feature the possibility of representing colors on the graph edges, hence a part of our work is also to create a bijection between our node and edge-colored subgraph labeling, and the node colored string we give to nauty as input.

Nauty needs a graph adjacency matrix string with only 0s and 1s, and for describing colors it can also have two input arrays. The array *ptn* (partition) and *lab*(label), represent the coloring partition of the vertices of the graph. The *lab* array is the list of graph vertices labeled, in a particular order, according to the color partitions represented in *ptn*. This last array is written in 0s and 1s, where the 0 value means that a particular color class ends in that node, and when it does not, we have the value 1.

This node labels however, nothing have to do with our original subgraph vertices labels in the network. We do not care at this point which node is which, we are only concerned in the relationships between them, their **topology**. In fact, for an uncolored graph these two arrays are even a default setting that we do not have to give as input. In Figure 3.3 we have an example of these input lists for a graph with three different node colors and 6 vertices.

After having performed graph canonization, *nauty* returns a new string, called *canon*, with the canonical labeling of the subgraph, and a new array *lab* representing the permutations done to the original node order.

We need to process our graph in order to have a bijection not only between our graph and the one we give nauty, but also between the canonical returned by nauty and the new canonical

Algorithm 2 Recover Subgraph From Label

Input: a long long int label that was enumerated *label*, the subgraph size*k*, a boolean as the direction of the network *dir* and a class Graph object pointer to write in *subG*
procedure: *LABEL_TO_SUBGRAPH(label, k, dir, subG)*

$bit\_shift \leftarrow 0$

$n\_bits\_edges \leftarrow 2^{c_e} - 1$

$n\_bits\_nodes \leftarrow 2^{c_v} - 1$

$space\_edge\_colors \leftarrow \lceil log_2(c_e + 1) \rceil$

if *dir* **then**

$d \leftarrow 2$

else

$d \leftarrow 1$

end if

for $i = K - 1\ ;\ i > -1; i--$ **do**

$node\_bits \leftarrow LABEL\_SIZE(i)$

$bits\_label \leftarrow 2^{node\_bits} - 1$

$vertex\_label \leftarrow bits\_label(label >> bit\_shift)$

$bit\_shift \leftarrow bit\_shift + node\_bits$

for $j = 0\ ;\ j < i + 1; j++$ **do**

$edge1 \leftarrow n\_bits\_edges(vertex\_label >> (d * j * space\_edge\_colors))$

$subG \leftarrow addEdge(j + 1, i + 1, edge1)$

if dir j != i **then**

$edge2 \leftarrow n\_bits\_edges(vertex\_label >> (d * j + 1) * space\_edge\_colors))$

$subG \leftarrow addEdge(i + 1, j + 1, edge2)$

else

$subG \leftarrow addEdge(i + 1, j + 1, edge1)$

end if

end for

$vertex \leftarrow n\_bits\_nodes(vertex\_label >> (d * i + 1) * space\_edge\_colors))$

$subG \leftarrow addNodeColor(i + 1, vertex)$

end for

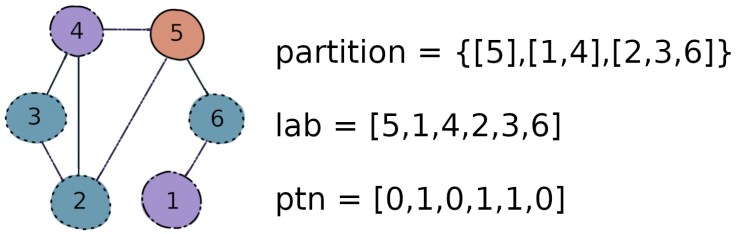

Figure 3.4: Example of the arrays *lab* and *lab* of nauty, that represent the three node color partitions of our graph.

subgraph type.

The first step is to write a graph procedure that turns our edge and node colored graph into a readable, and only node colored graph matrix and *ptn* and *lab* arrays. We decided to perform a transformation, that transfers the information about the edge colors, to the graph nodes by having c_e different vertices for each original node, each one representing and edge color. So if we have a subgraph of size n with c_e edge colors, our new subgraph will have size $n * c_e$. There are several ways of doing such transformation, but the most important thing is having one where we can reconstruct and edge and colored graph from the output given by nauty (and our previous information on the permutated nodes). We need bijections f_1, f_2 such that $f_1 : G(n, c_e, c_v) \implies G(n * c_e, c_v)$, $\quad nauty(G(n * c_e, c_v)) \implies G(n * c_e, c_v)'$ and $f_2 : G(n * c_e, c_v)' \implies G(n, c_e, c_v)'$, where $G(n, c_e, c_v)$ and $G(n, c_e, c_v)'$ are isomorphic, and $G(n, c_e, c_v)'$ is the canonical representation of $G(n, c_e, c_v)$.

Our approach was to create, for each node, a new node for each edge color in the original graph, the new nodes have the same colors as the original ones and are linked in a chain according to the increasing number of edge color they represent. We then make the connections between different original nodes, which in the new graph are only represented as links between nodes that represent the same edge colors. At the end, we get a new graph $G(n * c_e, c_v)$, with $n * c_e$ vertices, c_v node colors and $n * (c_e - 1) + e$, e is the number of edges of the original graph. Figure 3.5 illustrated this transformation for a graph with two edge colors, six nodes and three node colors.

In Figure 3.6 we have three other possible ways of transforming the same edge and node colored graph as in the previous example. Although we can clearly see a bijection that would get as back from each of the new graphs to the original ones, this is not as trivial after having the data that was processed by nauty. We did a series of experiments, to try to find and new node order to represent our original subgraph in a canonical form, based on the permutation and string given by nauty. For isomorphic graphs we would always get the same graph string, so we knew the canonization made by nauty was correct, but we were unable to reconstruct the original graph in an order, based on the permutation array, that would give the same representations for all isomorphic graphs. This happened when we were using graph transformations such as the one we can see in the image (a) of Figure 3.6, and what we discovered was that nauty would permutate the nodes representing different edge colors with each other, and for different graphs we could have node order permutations or edge color order permutations that would give different

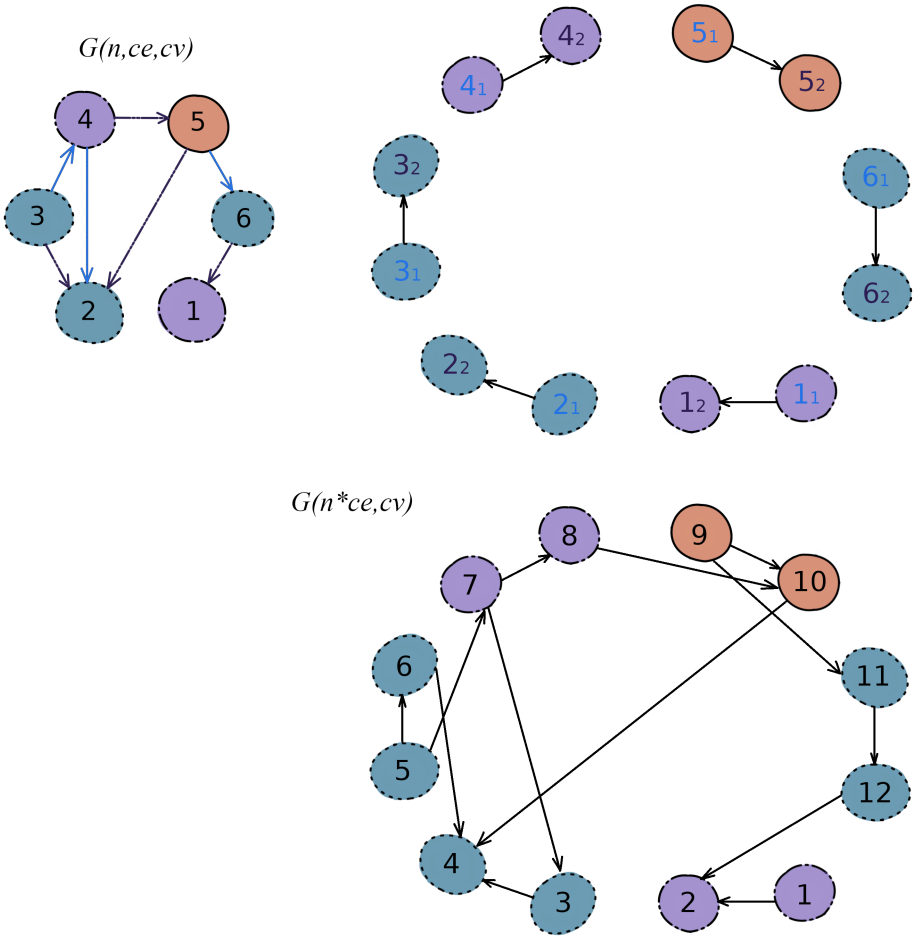

Figure 3.5: Example of the transformation method for getting a only node colored graph, from an edge and node colored original graph.

final canonical types of graphs of size n.

Using the canonical string given by nauty and the set of node and edge colors of each graph was also not a viable solution, since we would get, in some cases, the same canonical string for different subgraph types. Given this, we needed to find a new way of getting enough information for nauty not these permutations, and we found a few other ways of doing that. The ones we implemented were, first the transformation in Figure 2.5, where besides what we have in the transformation (a) of Figure 2.6 we add directed edges in a chain between nodes that represent different edge links of the same original nodes. This way we are giving an order to the graph representation of edge colors, and nauty is not able to make permutations that go against this order. In this case, the new order of the original graph, can be given by the order in which a representation of each original node first appears in the new label given by *nauty*. In order to do this, we have to process undirected graphs as directed ones, so we create bidirectional edges for each original graph edge. The other approach is the one in the image (b) of Figure 2.6, where we link the nodes that represent the same original node (with bidirectional edges for the directed case) all between each other, and create as many new colors as necessary to represent

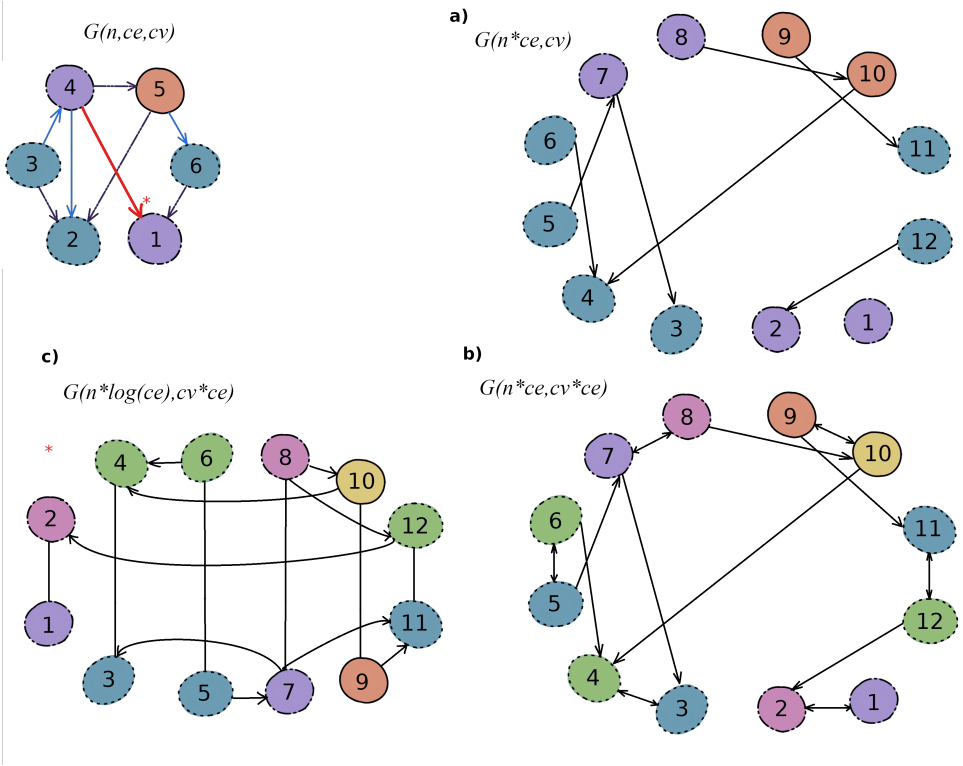

Figure 3.6: For the same original graph as in the previous examples, we show three different ways, (a, b and c), of getting a new only node colored graph. The edge $(4, 1)$ in red, is only represented for the third method, c.

the different nodes that have different node colors and represent different original edge colors. For this case, we get a new graph with $n * c_e$ nodes, $n * (\frac{c_e*(c_e-1)}{2}) + e$ edges and $c_v * c_e$ node colors. And the order for the canonical form of the original graph can be given as the order in which the first original edge color appears represented in a node in the label.

Finally we have the case in image (c) of Figure 2.6, where we create as many nodes or layers as $log_2(c_e)$, with different colors for each layer, and represent the edges of different colors according to their order, in each layer. In the example graph we have three edge colors, the first is represented in the first layers, the second is represented in the second (upper) layer, and the third is represented in both. For this case, the canonical labeling of the original graph can be given by the order of the canonical label of the nodes in the first layer.

The only other detail to guarantee we match the isomorphic graphs correctly, is make sure we give this node colors and the representations of the edge colors, always according to the same order. So we always traverse the graph first to get the set of ordered node and edge colors.

3.5 ColorFaSE

ColorFase is our main contribution in this thesis, it performs subgraph census on a given colored graph (with a colored edge list as input), returning the frequencies of each colored isomorphism class of size k. Our program works for both directed and undirected networks, with colors on the nodes, edges or both, and also supports colored self-edges. It is able to perform full exact subgraph enumeration, or approximate enumeration through a list of given input probability for each level (subgraph size). Our method is not restricted in terms of the number of colors (since we have the possibility of writing dynamic labels), or subgraph sizes, unless there is not enough memory to store the rapidly large numbers of subgraph occurrences. The output file reports the enumeration and canonization running times, as well as the number of subgraph occurrences, subgraph canonical types, and can also print the subgraph types matrix alongside with their frequencies if asked when calling the method.

The code is all implemented in c++, and was based on the *FaSE* implementation structure, using *ESU* algorithm to perform an efficient search on the graph, a dynamic g-trie to store and reuse subgraph information, and the third party tool *nauty* to perform colored subgraph canonization.

The algorithm is summarized in pseudocode, to help understand the backbone of the enumeration. Again, in concept this structure is very similar to $FaSE$, but all the procedures in the different steps, apart from the enumeration itself, had to be rewritten to be able to support colored networks, without compromising the algorithm efficiency too much.

Algorithm 3 ColorFaSE Algorithm

Input: a colored graph G, a boolean value for identifying direction *dir*, and the subgraph size k
Output: The frequencies for all k-subgraphs in G
procedure: *COLORFASE(G, dir, k)*

$run\_census(G, k, , 0)$

for each l *in* $gtrie.leaves()$ **do**

$label \leftarrow recover\_label\_path(l.label)$

$subG \leftarrow label\_To\_Subgraph(label, k, dir)$

$canon\_graph \leftarrow Colored\_Canonization(subG)$

$frequency[canons] += l.occurrences$

end for
procedure: *RUN_CENSUS(G,k,SubG,depth)*

if $d = k$ **then**

$gtrie[current\_node].occurrences += 1$

else

while *there is a path in enumeration tree from* $SubG$ **do**

$newSubG \leftarrow Enumerate\_From(SubG)$

$newNodePath \leftarrow newSubG.nextNode()$

$label \leftarrow Colored\_Labelling(SubG, newNodePath$

$nextGtrieNode \leftarrow gtrie.insert\_label(label)$

$newSubG.SubgraphnewNodePath$

$run\_census(G, k, newSubG, d + 1$

Chapter 4

Experimental Analysis

In this chapter we show that our method is competitive and can have a meaningful contribution to the CSCP. To this end, we performed a series of tests and experiments with our method. We then also use it to further explore the behavior of colored networks in terms of their subgraph patterns.

In the first section, we show our results for different subgraph sizes in different real colored networks and compare them with the available tool *fanmod*[55]. We briefly describe the datasets used and our experimental setup.

Following this, in section 4.2 we turn our attention to the experiments on several synthetic networks. We first explain the base network models we are working with and then propose the changes we have made to the algorithms to generate colored networks. We then study the color distributions and number of subgraphs that in the different networks, generated by different models. Finally, we assess the time efficiency of our method again with some of these networks and compare them with *fanmod*.

In the last section we look at how our method behaves with different combinations of numbers of colors in the edges and nodes on a series of different size synthetic networks generated with the models described in section 4.2. We also compare the performance of our two implementations: the one with the label written with a *long long int* datatype and the one suitable for a larger number of colors, with a *dynamic bitset* datatype.

All tests on our method were run in a Linux machine with an Intel Core i7-7500U (2.7GHz x 4) and 8GB of memory. The tests with *fanmod* were done in a Windows machine with the same characteristics, since we used their executable interface.

4.1 Experiments in Real Networks

To study the performance and range of our code, we tried to look for real world networks with different characteristics. We mainly look for labeled or heterogeneous network datasets, but also

took some other examples and done the necessary changes to represent a colored network. The networks used in this experiment have different sizes, average degree and numbers of colors in the edges and nodes. The first network, PTC-FM[39, 52], is a dataset on the toxicology of several chemical compounds in female mice. The graph we used represents a compound where the nodes are its different atoms and the edges are their type of chemical bond (single, double, aromatic). AIDS [38, 39] is also a dataset of molecular compounds, but it was created from an antiviral screen database. Similarly, Mutagenicity[39, 58]is a database of chemical compounds on drugs which can be mutagenic, and our network represents one of those compounds. cora[4, 39] is a citation network, where the nodes represent scientific publications and the edges are citations. DD6[31, 39] is a network which represents a macromolecule, in this case the nodes are different amino acids and the edges their spatial proximity. Finally, bitcoin[19, 20] is a weighted temporal network of bitcoin users and their ratings between each other. We disregarded the temporal tag and transformed the weights of the rating (integer numbers) into different types of edges.

The details on the direction, number of nodes, number of edges, average degree, number of node colors, number of edge colors and the type of each network, are represented in the table 4.1.

Network	Directed	Nodes	Edges	Avg. Degree	Node Colors	Edge Colors	Type
PTC-FM	No	51	56	2.2	6	3	biochemical[39, 52]
AIDS	No	77	83	2.2	5	2	biochemical[38, 39]
Mutagenicity	No	105	106	2.0	5	2	biochemical[39, 58]
cora	Yes	2709	5429	2.0	7	1	social[4, 39]
DD6	No	4152	10320	5	20	1	biological[31, 39]
bitcoin	Yes	3783	24186	6.4	1	7	social[19, 20]

Table 4.1: Real networks used in the experiment and their characteristics.

For each network we run our application for increasing subgraph sizes (from 3 to 10 depending on the network size), and the same was done with *fanmod* in order to compare the performances. Since a brute force approach to the subgraph census would not be workable in terms of execution time, we also used the results given by *fanmod* to validate our own. For the different types of networks, we always got the exact same number of occurrences and isomorphism classes. This is important, since it again shows that our method for CGC works well, and we are using *nauty* and its output correctly.

As for the results, we can confirm that our approach is suitable for a considerably larger number of vertex and edge colors, as well as motif sizes. In fact, by implementing the *dynamic_bitset* labeling, we are no longer conditioned to any number of colors, except for the maximum number of nodes accepted by *nauty* which as we have seen will increase with the number of edge colors. Of course more colors, means more different subgraphs and a larger g-trie, and we can easily be limited in terms of memory when working with bigger subgraph sizes, but theoretically our implementation would be able to accept as input as many colors as needed. This is very noticeable for the case of smaller networks, where our algorithm is very fast to compute all the subgraph patterns and their frequencies even for larger subgraph sizes, while *fanmod* does not

even accept most numbers of colors for medium subgraph sizes. For instance, in table 4.2) we can see that for the first three networks it takes only 1 second for *ColorFaSE* to compute the census for subgraphs of size 10, while *fanmod* does not accept subgraph sizes larger than 8, and the number of colors accepted also restricts the subgraph sizes, in these cases it can go only up to 5. All the results are summarized in table 4.2).

Network	K	Subgraph Enumeration		ColorFaSE		fanmod			
		Isomorphism Classes	Occurrences	Enumeration Time (ms)	Total Time (s)	Enumeration Time (ms)	Speedup	Total Time (s)	Speedup
PTC-FM	3	13	83	<1	<0.01	<1	1	<1	1
	4	25	134	<1	<0.01	<1	1	<1	1
	5	41	227	<1	<0.01	10	10	<1	1
	6	81	368	1	0.03	(*)	-	(*)	-
	7	142	573	1	0.02	(*)	-	(*)	-
	8	232	860	2	0.05	(*)	-	(*)	-
	9	348	1261	13	0.18	(*)	-	(*)	-
	10	493	1906	5	1.21	(*)	-	(*)	-
AIDS	3	14	129	<1	<0.01	<1	1	<1	1
	4	30	207	<1	<0.01	<1	1	<1	1
	5	62	345	<1	<0.01	<1	1	<1	1
	6	114	565	1	0.011	(*)	-	(*)	-
	7	295	930	2	0.03	(*)	-	(*)	-
	8	581	1518	2	0.06	(*)	-	(*)	-
	9	1052	2431	5	0.25	(*)	-	(*)	-
	10	1805	3843	10	0.29	(*)	-	(*)	-
Mutagenicity	3	20	166	<1	<0.01	<1	1	<1	1
	4	46	296	<1	<0.01	<1	1	<1	1
	5	106	563	<1	<0.01	10	10	<1	1
	6	274	1078	2	0.02	(*)	-	(*)	-
	7	659	2021	3	0.06	(*)	-	(*)	-
	8	1502	3708	26	0.16	(*)	-	(*)	-
	9	3044	6700	12	0.39	(*)	-	(*)	-
	10	5795	12000	56	0.78	(*)	-	(*)	-
cora	3	719	49041	4	0.04	55	13.8	<1	1
	4	6990	1295733	59	0.46	1191	20.2	8	17.4
	5	62903	45174275	1862	6.87	(**)	-	(**)	-
DD6	3	3008	32042	4	0.17	(*)	-	(*)	-
	4	27210	110631	29	1.40	(*)	-	(*)	-
bitcoin	3	4626	807652	56	0.35	802	14.3	5	14.3
	4	430269	73832383	5226	(***)	113248	21.7	135	(***)

(*) - *fanmod* does not support this number of colors.
(**) - *fanmod* interface crashed without writing the output file.
(***) - *ColorFaSE* completed the enumeration, but the process was killed during canonization due to lack of memory.

Table 4.2: Detailed experimental results for different subgraph sizes in the real networks and comparison with *fanmod* method.

Looking at the execution times, we can also see that our implementation largely outperforms *fanmod*, with speedups up to 22 times faster. We discriminate between the time it takes to count all the subgraph occurrences, and the total time, which ideally would correspond to the enumeration time plus the canonization/isomorphism tests time, but since *fanmod* only reports the enumeration time in the output file, we went with the total *elapsed time* which is shown on

the interface for this comparison. Furthermore, the numerical precision of this total time only goes up to one second, so in most cases for these networks we do not have enough resolution to get real speedup values.

Given the restrictive range values of our real networks' characteristics, it is recommended to perform more tests with networks with different features. To do so, we have generated a series of synthetic networks, which we present in the following section.

4.2 Networks Models for Synthetic Colored Networks

The generation of synthetic networks is very useful for performing experiments on the behavior of the implementations in a wider range of parameters. Considering this, and the few and low quality colored networks datasets available, we decided to generate our own networks to perform further testing.

There are several well studied network models which can generate different types of random networks to mimic the profile of real world networks. In this experiment, we focus our attention in two of them: the Erdős Rényi Model[10], and the Barabási-Albert Model[3] (or Preferential Attachment).

4.2.1 Erdős Rényi Model for Colored Networks

The Erdős Rényi Model generates a random graph $G_{ER}(n, p)$, according to a two parameter input: the number of nodes, n, and the probability for a pair of nodes having an edge, p. For each pair of nodes (i, j), there is the same probability p of occurring and edge.

To generalize this concept to colored networks, we can just assume a fixed probability distribution $p(c_v)$ for the nodes, and another for the edges, $p(c_e)$. The colors of the nodes would be initially set according to this distribution $p(c_v)$, and every time we get an edge between a pair of nodes, we would sample its color based exclusively on $p(c_e)$. This would mean there is no relationship between the colors of the nodes we are connecting, or between the colors of the nodes and the color of the new edge.

In order to better illustrate this idea, we represent this algorithm, which we have implemented to generate random networks, in pseudocode.

Having generalized the model for colored networks, we can generate them for a series of different sizes $(n, p)values$, numbers of colors and color distributions. We want to use those networks to test our approach, but due to the nature of the original model, which directly translates to the colored version, we can not guarantee that the output random graph G is a fully connected network. For large numbers of p this is a likely scenario, but for smaller ones we get many graph components which are not suitable for performing subgraph census unless we would generate a new file for each component.

Algorithm 4 Erdős Rényi Colored Network Generator

Input: number of nodes n, probability p, number of node colors c_v, node color probability distribution $p(c_v)$, number of edge colors c_e and edge color probability distribution $p(c_e)$
Output: a colored graph G
procedure: *ERDOS_RENYI_COLORED_MODEL(n, p, c_v, $p(c_v)$, c_e, $p(c_e)$)*

$G(node\_colors\_list) \leftarrow sample(n, c_v, p(c_v))$
for $i \leftarrow 0$ *to* n **do**
  for $j \leftarrow 0$ *to* n **do**
    $random\_number \leftarrow sample\_uniform(0, 1)$
    if $random\_number < p$ **then**
      $edge\_color \leftarrow sample(1, c_e, p(c_e))$
      add edge (i, j) with color $edge\_color$ to G
    end if
  end for
end for
return G

For this reason, we turn to another very well known graph model.

4.2.2 Barabási-Albert Model for Colored Networks

The Barabási-Albert model generates a scale-free network, giving two input parameters: the number of nodes, n, and the number of connections each new node will make to pre-existing ones, m. This model simulates the behavior of many real world networks, which exhibit a power-law degree distribution based on a preferential attachment behavior, where nodes with a higher degree have higher probability of getting new links from new nodes.

In real life, this is many times the case when for instance, someone who knows a lot of people is more likely to get to know new people coming into their social circle (as a friend of a friend for example), than someone who has few connections into that same circle.

To study our approach with this model, we also have to implement a new version with colored nodes and edges, and there is no standard way of doing so. We propose two different ways of attributing color to this model: one with constant color probability distributions (much like we did in the Erdős Rényi Model), and one that translates this preferential behavior to the colors of the new nodes and edges. We show both algorithms in pseudocode, but it is easy to understand that they are equivalent in terms of the number of connections they generate and degree distributions. The only thing that changes is the number of nodes of each color and the types of connections between them.

We named the model with fixed probability distribution, *Barabási* 1, and the one with

preferential coloring, *Barabási* 2. The probability distribution with which we choose the color of the new node in the second model, and the color of its connections, is similar to the probability distribution for choosing the nodes to connect. The probability of a new node connecting with a node i in a network with n nodes is given as: $p(i) = \frac{degree[i]}{\sum_{j=0}^{n} degree[j]}$). We apply a similar probability distribution for the edge and node colors, but we replace the degree of each node, with the frequency of each color. The other main difference, is that for each iteration we sample m nodes from that distribution without replacement, in order not to get the same node more than once, and in the case of the colors we can sample with replacement and the m edge colors for the m new connections can be the same. With this model, we are replicating a behavior where a node with a specific color is more likely to enter a network where its color is largely represented than one with very few nodes of the same type, and the same goes for the connections it is going to make.

Algorithm 5 Barabási-Albert Colored Network Generator 1

Input: number of nodes n, number of new connections for each new node m, number of node colors c_v, node color probability distribution $p(c_v)$, number of edge colors c_e and edge color probability distribution $p(c_e)$

Output: a colored graph G

procedure: *BARABASI_ALBERT_COLORED_MODEL_1(n, p, c_v, p(c_v), c_e, p(c_e))*

$G(node\_colors\_list) \leftarrow sample(n, c_v, p(c_v))$

for $i \leftarrow 0$ *to* m **do**

  $edge\_color \leftarrow sample(1, c_e, p(c_e))$

  add edge (i, m) with color $edge\_color$ to G

  $degree[i] \leftarrow 1$

end for

$degre[m] \leftarrow m$

for $i \leftarrow m + 1$ *to* n **do**

  $p(d_x) = \frac{degree[x]}{\sum_{j=0}^{i} degree[j]}$

  $edges \leftarrow sample(m, i, p(d_x)$

  for $j \leftarrow 0$ *to* m **do**

    $edge\_color \leftarrow sample(1, c_e, p(c_e))$ $(without\ replacement)$

    add edge $(i, edges[j])$ with color $edge\_color$ to G

    $degree[edge[j]] \leftarrow degree[edge[j]] + 1$

  end for

  $degree[i] \leftarrow m$

end for

return G

To understand the implications of these two forms of applying colors to a scale-free network, we generated a series of these networks with both models for varying sizes of n and m. We then

Algorithm 6 Barabási-Albert Colored Network Generator 2

Input:number of nodes n, number of new connections for each new node m, number of node colors c_v, node color initial probability distribution $p(c_v)$, number of edge colors c_e and edge color initial probability distribution $p(c_e)$

Output: a colored graph G

procedure: *BARABASI_ALBERT_COLORED_MODEL_2(n, p, c_v, $p(c_v)$, c_e, $p(c_e)$)*

$G(node\_colors\_list) \leftarrow sample(m+1, c_v, p(c_v))$

for $i \leftarrow 0$ *to* c_v **do**

$node\_colors[i] \leftarrow 1$

end for

for $i \leftarrow 0$ *to* c_e **do**

$edge\_colors[i] \leftarrow 1$

end for

for $i \leftarrow 0$ *to* m **do**

$p(c'_e) \leftarrow p(x) = \frac{edge\_colors[x]}{\sum_{j=0}^{c_e} edge\_colors[j]}$

$edge\_color \leftarrow sample(1, c_e, p(c'_e))$

add edge (i, m) with color $edge\_color$ to G

$degree[i] \leftarrow 1$

$edge\_colors[edge\_color] \leftarrow edge\_colors[edge_color] + 1$

$node\_colors[node\_colors\_list[i]] \leftarrow node\_colors[node\_colors\_list[i]] + 1$

end for

$node\_colors[node\_colors\_list[m]] \leftarrow node\_colors[node\_colors\_list[m]] + 1$

$degre[m] \leftarrow m$

for $i \leftarrow m+1$ *to* n **do**

$p(d_x) \leftarrow \frac{degree[x]}{\sum_{j=0}^{i} degree[j]}$

$p(c'_e) \leftarrow p(x) = \frac{edge\_colors[x]}{\sum_{j=0}^{c_e} edge\_colors[j]}$

$p(c'_v) \leftarrow p(x) = \frac{node\_colors[x]}{\sum_{j=0}^{c_v} node\_colors[j]}$

$edges \leftarrow sample(m, i, p(d_x) \quad (without\ replacement)$

$ecolors \leftarrow sample(m, c_e, p(c'_e) \quad (with\ replacement)$

$ncolor \leftarrow sample(1, c_v, p(c'_v)$

$node\_colors[ncolor] \leftarrow node\_colors[ncolor] + 1$

$node\_colors\_list[i] \leftarrow ncolor$

for $j \leftarrow 0$ *to* m **do**

add edge $(i, edges[j])$ with color $ecolors[j]$ to G

$degree[edge[j]] \leftarrow degree[edge[j]] + 1$

$edge\_colors[ecolors[j]] \leftarrow edge\_colors[ecolors[j]] + 1$

end for

$degree[i] \leftarrow m$

end for

return G

run our implementation to get the number of subgraph occurrences and canonical subgraph types for both models and compare them.

Figure 4.1 and 4.2 show the results for the number of subgraphs found and number of different canonical subgraph types, respectively. For each model we generated five networks with $m = [1, 2, 3, 4, 5]$ for each number of nodes, $n = [100, 250, 500, 1000, 2500]$ and with colors $c_v = 2$ and $c_e = 3$. The probability distribution for the first model was $p(c_v) = [0.4, 0.6]$ for the nodes, and $p(c_e) = [0.2, 0.3, 0.5]$, and these probabilities were also given to sample the colors of the first m nodes and edges of the second model. The frequencies correspond to the census of these networks for subgraph size $k = 4$.

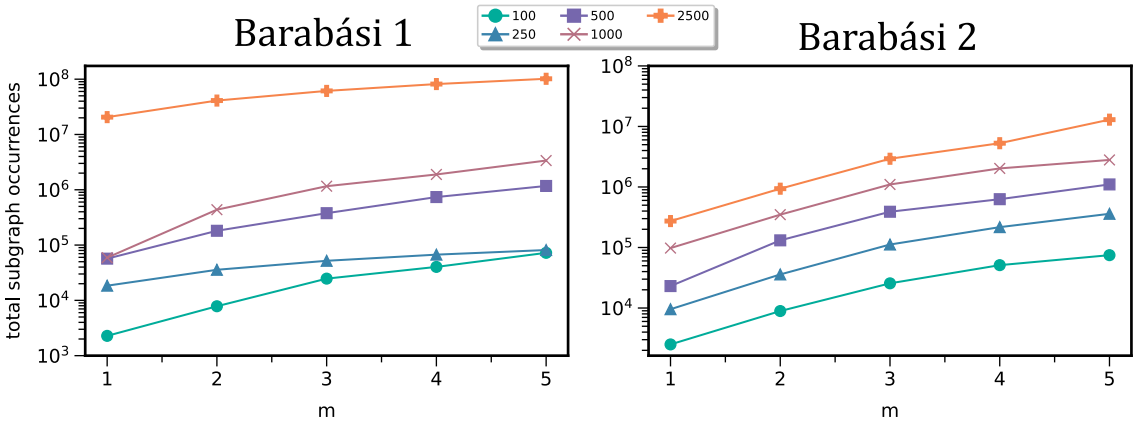

Figure 4.1: Total Subgraph Occurrences found on the networks generated with the Barabási 1 model on the left, and with the Barabási 2 model on the right

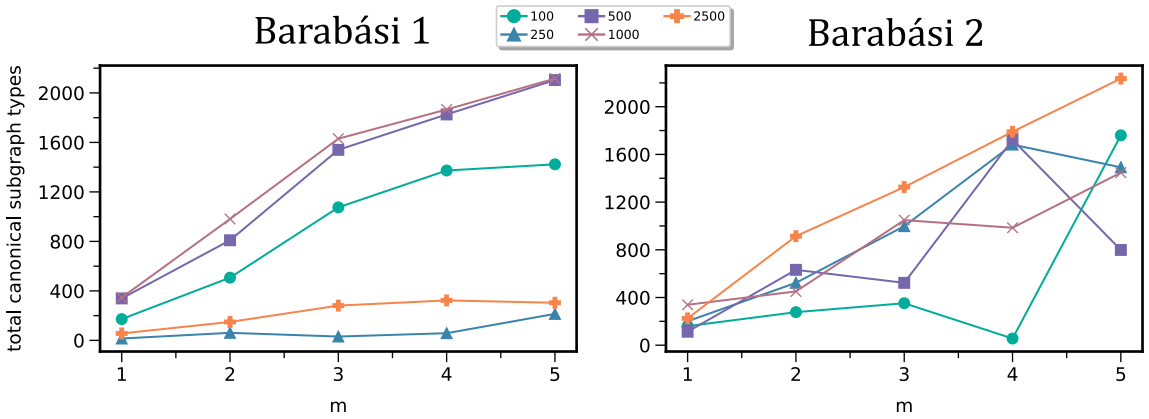

Figure 4.2: Total Canonical Subgraph Types found on the networks generated with the Barabási 1 model on the left, and with the Barabási 2 model on the right.

Both models exhibit similar numbers of subgraph occurrences, and subgraph types. We could expect a similar number of occurrences, since the two models follow the same method to create new edges. Nevertheless, model 2 reaches higher numbers of isomorphism subgraph classes, for smaller networks. This is expected since in the second model we are going to have a power-law

distribution in frequencies of the nodes and edge colors, which means that some types of patterns are much more likely to occur than almost all others.

4.2.3 Performance Tests

Lastly, we did a series of experiments to better understand the performance of our method, using the Barabási 2 models to generate different sets of networks. Given the limitations of the parameters in real networks, we want to show the efficiency of our program for different subgraph sizes, numbers of colors, average degrees and numbers of nodes using these networks. So we generated 2 files for each network, one according to our input format, and the other to be read by *fanmod*.

Here we fixed the number of colors as $c_v = 2$ and $c_e = 3$, and the number of connections $m = 3$. We then run tests for increasing subgraph sizes for networks of size $n = [100, 250, 500, 1000, 2500, 4000]$. The results are summarized in Table 4.3, for the different subgraph sizes K, and number of nodes.

N	K	Subgraph Enumeration		ColorFaSE		fanmod			
		Isomorphism Classes	Occurrences	Enumeration Time (ms)	Total Time (s)	Enumeration Time (ms)	Speedup	Total Time (s)	Speedup
100	3	37	2430	1	<0.01	1	**10**	<1	**1**
	4	352	25588	5	0.06	28	**5.60**	< 1	**1**
	5	4240	276025	46	0.49	515	**11.20**	4	**8.16**
	6	53977	2870737	236	7.21	(*)	-	(*)	-
250	3	78	7433	1	0.02	3	**3**	<1	1
	4	998	111987	17	0.11	112	**6.59**	1	**9.1**
	5	14590	1853010	106	1.52	2804	**26.45**	16	**10.47**
500	3	55	18609	4	0.02	17	**4.25**	1	**25**
	4	523	389850	16	0.079	670	**41.88**	4	**50.63**
	5	5911	9187559	379	1.18	25986	**68.56**	86	**72.88**
1000	3	74	41328	10	0.08	29	**2.90**	<1	1
	4	1049	1100020	62	0.10	1772	**28.58**	8	**80.0**
	5	17999	35624063	1531	3.19	(**)	-	(**)	-
2500	3	86	105116	8	0.01	140	**17.50**	<1	> **30.0**
	4	1326	2926940	144	0.27	4400	**30.55**	22	**81.48**
	5	25062	102881170	6082	9.94	(**)	-	(**)	-
4000	3	77	174044	17	0.02	182	**10.70**	1	**50.0**
	4	1091	5180007	295	0.41	3777	**12.80**	35	**85.37**
	5	19031	199495543	11883	15.13	(**)	-	(**)	-

(*) - *fanmod* does not support this number of colors.
(**) - *fanmod* interface crashed before completing the enumeration or writing any output file.

Table 4.3: Detailed experimental results for different subgraph sizes in the generated Barabási networks, and comparison with *fanmod* method.

As we can see, our approach not only works for larger numbers of subgraph sizes and colors, but also provides a considerable speedup in the execution times, compared to *fanmod*. It is also

interesting to see that this speedup has a tendency to increase with the size of the subgraphs enumerated and the size of the networks. This means that our approach scales better than *fanmod*, hence being a not only faster tool, but also one that provides a wider range of applications and features.

4.3 Further Studies

In this chapter, we resorted to the previously generated networks to run our code in different settings, and explore more about its behavior. We first study how the number of edge and node colors impacts the performance. And lastly, we did an experiment using the two different labels (*long long int* and *dynamic bitset*) for the same networks and subgraph settings, to see the impact of using a different datatype.

4.3.1 The Impact of Number of Node and Edge Colors in the Performance

In this experiment, we wanted to study how the number of colors on the edges played a role in the performance of our method. To do so, we used a series of Barabási Networks and for each combination of network size $n = [200, 500, 100]$, and $m = [2, 3]$, we first fixed the number of edge colors to 1 and generated eight networks with $c_v = [1, 2, 3, 4, 5, 6, 7, 8]$, and then we did the opposite, setting the number of node colors to 1 and varying the number of edge colors. The Barabási networks were generated with the model 2, and the input probabilities for the first m nodes and edges were uniform. For each case, we used our algorithm to count the number of canonical subgraph types, and registered the execution times.

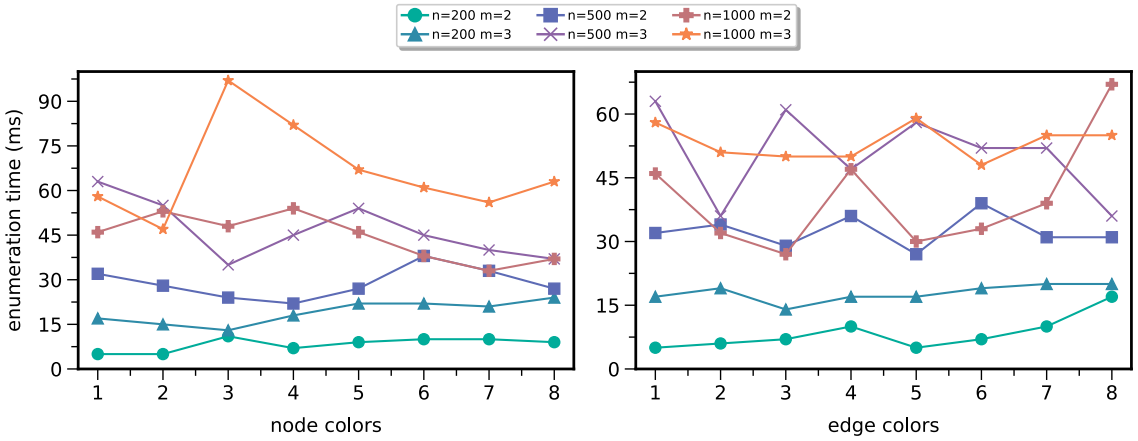

Figure 4.3: Total Subgraph Occurrences found on the networks generated with the Barabási 1 model on the left, and with the Barabási 2 model on the right

The results are presented for the enumeration times in Figure 4.3, for the canonization time in Figure 4.4, and lastly Figure 4.5 shows the total number of subgraph types. This canonization type includes every step, from translating the label to a subgraph matrix, to performing the

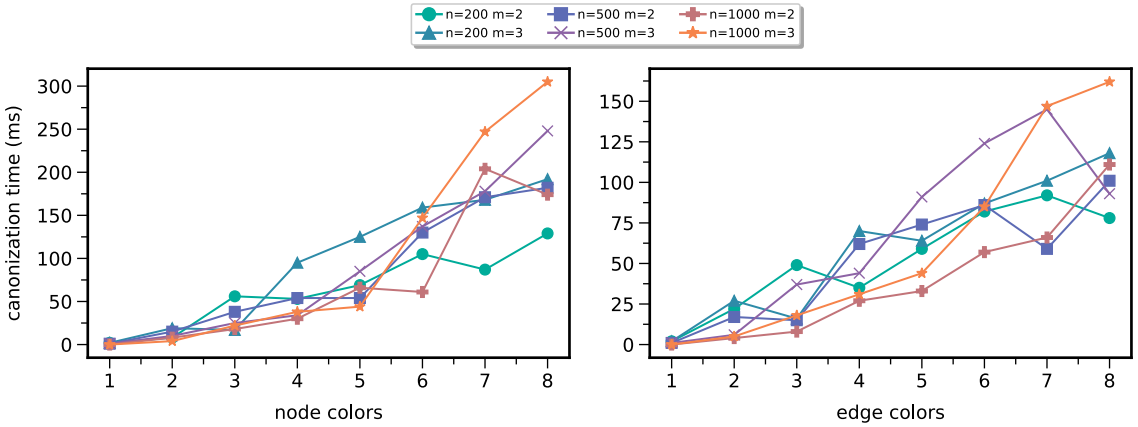

Figure 4.4: Total enumeration time in ms for a series of six different networks as a function of the number of node colors, on the left, and as a function of the number of edge colors, on the right.

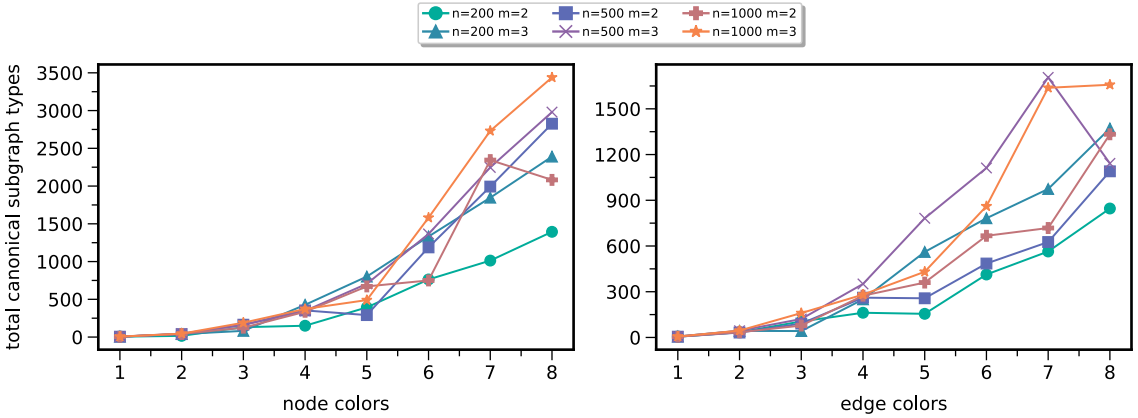

Figure 4.5: Total canonization time in ms for a series of six different networks as a function of the number of node colors, on the left, and as a function of the number of edge colors, on the right.

necessary transformations for the *nauty* input, and finally processing the *nauty* output into a canonical colored form and counting the final frequencies.

From Figure 4.3 we can understand that the number of both edge and node colors, does not seem to affect the execution time of the enumeration. This values only change with the network size and average degree, apart from some fluctuations. This is a good indication that our colored label and every method that involves processing this label, is efficiently written and does not compromise the performance, just as we intended.

However, the story is different with the execution times for the canonization, as one can see in Figure 4.4. Both the number of node and edge colors seem to increase the canonization times exponentially. But this can be explained with the same exponential increase of the number of subgraph types, represented in Figure 4.5, which was already expected given the combinatorial explosion when attributing colors to subgraphs.

Another interesting behavior that surfaces with this experiment, is the fact that for these

networks, the number of canonical subgraph types for each number of node colors is almost the double of the number of types, for the same number of edge colors. The most likely explanation for this is in the way we generated the initial probability distributions in our model.

4.3.2 long long int vs. dynamic_bitset label

Lastly, we wanted to see how much slower the *dynamic_bitset* label is, in comparison with the default *long long int implementation*. To do so, we used the Barabási networks with parameters: $c_v = 2$, $c_e = 3$, $m = 3$ and $n = [50, 100, 250, 1000, 2500, 4000]$, and run our code for a subgraph size $k = 4$. In Figure 4.6 we have execution times for the enumeration procedure, and in Figure 4.7 we have the execution times for the canonization procedure.

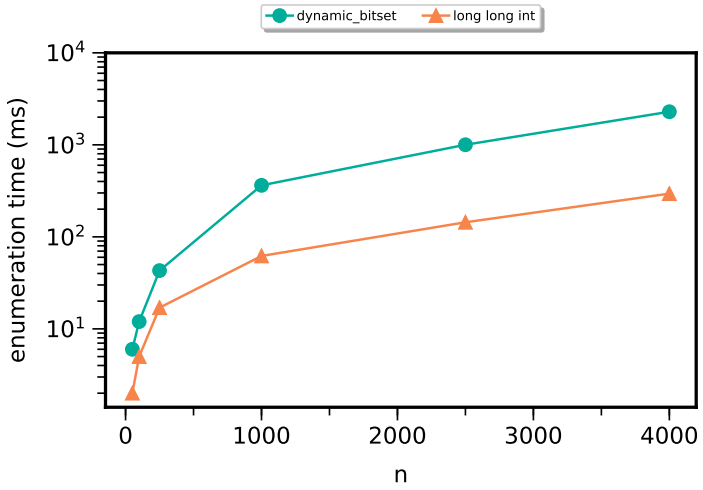

Figure 4.6: Enumeration times for Barabási networks of increasing sizes for the *long long int* label and the *dynamic_bitset* label.

For smaller networks the difference between the execution times of the two labels is not much, but as we increase the number of nodes, this difference grows up to almost two orders of magnitude. This is not ideal, but for larger numbers of node and edges colors is the best option we have, and still runs fairly fast.

As for the canonization procedure, would be expected, the type of label does not impact the running times. This is because this label is only used to get the subgraph matrix for each g-trie leaf, and the rest of the most costly tasks are independent of this label.

Since the canonization time, will be for many more complex networks the bottleneck of our approach, both in terms of time and memory, these results are not that worrisome, since our version of the implementation with the *dynamic_bitset* datatype only slows down the execution time of the enumeration procedure of the approach.

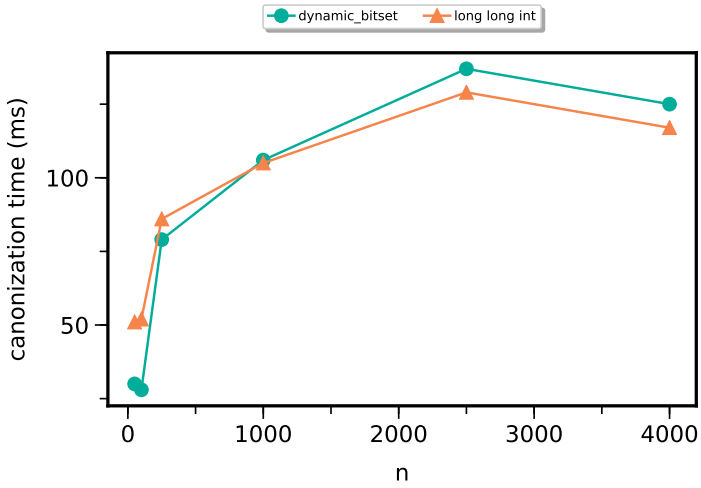

Figure 4.7: Canonization times for Barabási networks of increasing sizes for the *long long int* label and the *dynamic_bitset* label.

Chapter 5

Conclusions

Our literature review around the Subgraph Census Problem showed that, although there has been quite a few developments over the last few decades, with the emergence of new efficient subgraph enumeration algorithms, there are few approaches that have the capacity to include information on the types of nodes or edges in networks, a feature we call color.

We discuss the relevance of finding patterns with color information, and the challenges faced in this type of approach, concerning for instance graph representations, the GIP and colored graph canonization. We also present *ColorFase*, our main contribution in this thesis, which is an algorithm that performs subgraph census on a given graph network (with a colored edge list as input), returning the frequencies of each colored isomorphism class of size k. Our implementation, is an efficient way of solving the Colored SCP, and was done by extending the work done in FaSE[33], one of faster the state-of-the-art approaches to the SCP. The processed included rewriting or creating procedures, and changing some datatypes of the implementation, in order for it to support color information for both the nodes and edges. We also managed to support colored self-edges in our version. This method in yet unpublished work, and it is our intention to make our code available to the general public, so that the network science community can benefit from it.

The experiments on the performance of our implementation showed that it is competitive in terms of running times, offering as much as +80x speedup in comparison with *FANMOD*, for both real and synthetic networks. But most importantly, this speedup grows with the size of the network and the subgraph, and our implementation features a much wider range of numbers of colors in larger subgraph sizes.

5.1 Future Work

Finally, we would like to mention that the work here developed, can easily be improved and extended to give rise to a faster implementation via parallelization and some fine-tuning changes to our code. Another possible approach that could give a better performance to this algorithm,

would be to study ways of optimizing the sampling to avoid uninteresting clusters or parts of the graph, and explore this probability conditions. Or even to write a version that is able to focus on the enumeration of a specific type of interesting patterns, or avoids searching another set of uninteresting ones and their super-graphs, saving time and memory.

Since the bottleneck of our approach in terms of memory is in the canonization step, it would be interesting to study an implementation which could reduce the number of g-trie leaves to perform canonization, by doing isomorphism tests while enumerating the subgraph and recursively shrink the tree.

The further study on colored network motifs definition and subgraph significance is also of our interest, and even though it would likely not change the way we approach the CSCP, could give rise to potentially interesting applications of it. So in the future we want to feature the generation of random colored networks to calculate the subgraph representation, and return a possible list of subgraph motifs.

Bibliography

[1] Adami, C., J. Qian, M. Rupp, and A. Hintze (2011). Information content of colored motifs in complex networks. *Artificial Life 17*(4), 375–390.

[2] Akoglu, L., H. Tong, and D. Koutra (2015). Graph based anomaly detection and description: a survey. *Data mining and knowledge discovery 29*(3), 626–688.

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

[4] Andrew Kachites McCallum, Kamal Nigam, J. R. K. S. cora dataset. In *Automating the Construction of Internet Portals with Machine Learning*.

[5] Barabási, A.-L. (2013). Network science. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 371*(1987), 20120375. Available online at http://networksciencebook.com/.

[6] Betzler, N., R. Van Bevern, M. R. Fellows, C. Komusiewicz, and R. Niedermeier (2011). Parameterized algorithmics for finding connected motifs in biological networks. *IEEE/ACM Transactions on Computational Biology and Bioinformatics 8*(5), 1296–1308.

[7] Bhapkar, H., P. N. Mahalle, and P. S. Dhotre (2020). Virus graph and covid-19 pandemic: a graph theory approach. In *Big Data Analytics and Artificial Intelligence against COVID-19: Innovation Vision and Approach*, pp. 15–34. Springer.

[8] Biamonte, J., M. Faccin, and M. De Domenico (2019). Complex networks from classical to quantum. *Communications Physics 2*(1), 1–10.

[9] Chen, J., W. Hsu, M. L. Lee, and S.-K. Ng (2007). Labeling network motifs in protein interactomes for protein function prediction. In *2007 IEEE 23rd International Conference on Data Engineering*, pp. 546–555. IEEE.

[10] Erdös, P. and A. Rényi (1959). On random graphs publ. *Math. debrecen 6*, 290–297.

[11] Fellows, M. R., G. Fertin, D. Hermelin, and S. Vialette (2007). Sharp tractability borderlines for finding connected motifs in vertex-colored graphs. In *International Colloquium on Automata, Languages, and Programming*, pp. 340–351. Springer.

[12] Goldreich, O. (2010). *P, NP, and NP-Completeness: The basics of computational complexity.* Cambridge University Press.

[13] Grochow, J. A. and M. Kellis (2007). Network motif discovery using subgraph enumeration and symmetry-breaking. In *Annual International Conference on Research in Computational Molecular Biology*, pp. 92–106. Springer.

[14] Gross, J. L. and J. Yellen (2005). *Graph theory and its applications.* CRC press.

[15] Gu, S., J. Johnson, F. E. Faisal, and T. Milenković (2018). From homogeneous to heterogeneous network alignment via colored graphlets. *Scientific reports 8*(1), 1–16.

[16] Havlin, S., D. Y. Kenett, E. Ben-Jacob, A. Bunde, R. Cohen, H. Hermann, J. Kantelhardt, J. Kertész, S. Kirkpatrick, J. Kurths, et al. (2012). Challenges in network science: Applications to infrastructures, climate, social systems and economics. *The European Physical Journal Special Topics 214*(1), 273–293.

[17] Kashani, Z. R. M., H. Ahrabian, E. Elahi, A. Nowzari-Dalini, E. S. Ansari, S. Asadi, S. Mohammadi, F. Schreiber, and A. Masoudi-Nejad (2009). Kavosh: a new algorithm for finding network motifs. *BMC bioinformatics 10*(1), 1–12.

[18] Khakabimamaghani, S., I. Sharafuddin, N. Dichter, I. Koch, and A. Masoudi-Nejad (2013). Quatexelero: an accelerated exact network motif detection algorithm. *PloS one 8*(7), e68073.

[19] Kumar, S., B. Hooi, D. Makhija, M. Kumar, C. Faloutsos, and V. Subrahmanian (2018). Rev2: Fraudulent user prediction in rating platforms. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, pp. 333–341. ACM.

[20] Kumar, S., F. Spezzano, V. Subrahmanian, and C. Faloutsos (2016). Edge weight prediction in weighted signed networks. In *Data Mining (ICDM), 2016 IEEE 16th International Conference on*, pp. 221–230. IEEE.

[21] Lacroix, V., C. G. Fernandes, and M.-F. Sagot (2006). Motif search in graphs: application to metabolic networks. *IEEE/ACM transactions on computational biology and bioinformatics 3*(4), 360–368.

[22] Latora, V., V. Nicosia, and G. Russo (2017). *Complex networks: principles, methods and applications.* Cambridge University Press.

[23] Li, G., J. Luo, Z. Xiao, and C. Liang (2018). Mtmo: an efficient network-centric algorithm for subtree counting and enumeration. *Quantitative Biology 6*(2), 142–154.

[24] Lin, W., X. Xiao, X. Xie, and X.-L. Li (2016). Network motif discovery: A gpu approach. *IEEE transactions on knowledge and data engineering 29*(3), 513–528.

[25] Liu, B., J. Li, C. Chen, W. Tan, Q. Chen, and M. Zhou (2015). Efficient motif discovery for large-scale time series in healthcare. *IEEE Transactions on Industrial Informatics 11*(3), 583–590.

[26] Luo, J., L. Ding, C. Liang, and N. H. Tu (2018). An efficient network motif discovery approach for co-regulatory networks. *IEEE Access 6*, 14151–14158.

[27] McKay, B. D. and A. Piperno (2012). nautytraces, software distribution web page.

[28] Micale, G., R. Giugno, A. Ferro, M. Mongiovì, D. Shasha, and A. Pulvirenti (2018). Fast analytical methods for finding significant labeled graph motifs. *Data Mining and Knowledge Discovery 32*(2), 504–531.

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

[30] Moon, H. S., J. Bhak, K. H. Lee, and D. Lee (2005). Architecture of basic building blocks in protein and domain structural interaction networks. *Bioinformatics 21*(8), 1479–1486.

[31] Morris, C., N. M. Kriege, F. Bause, K. Kersting, P. Mutzel, and M. Neumann (2020). Tudataset: A collection of benchmark datasets for learning with graphs. *arXiv preprint arXiv:2007.08663*.

[32] Müngen, A. A. and M. Kaya (2017). Influence analysis of posts in social networks by using quad-motifs. In *2017 International Artificial Intelligence and Data Processing Symposium (IDAP)*, pp. 1–5. IEEE.

[33] Paredes, P. and P. Ribeiro (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*, pp. 264–271.

[34] Paredes, P. and P. Ribeiro (2015). Rand-fase: fast approximate subgraph census. *Social Network Analysis and Mining 5*(1), 17.

[35] Ribeiro, P., P. Paredes, M. E. Silva, D. Aparicio, and F. Silva (2019). A survey on subgraph counting: Concepts, algorithms and applications to network motifs and graphlets. *arXiv preprint arXiv:1910.13011*.

[36] Ribeiro, P. and F. Silva (2014a). Discovering colored network motifs. In *Complex Networks V*, pp. 107–118. Springer.

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

[38] Riesen, K. Aids dataset. In *IAM Graph Database Repository for Graph Based Pattern Recognition and Machine Learning*.

[39] Rossi, R. A. and N. K. Ahmed (2015). The network data repository with interactive graph analytics and visualization. In *AAAI*.

[40] Rossi, R. A., N. K. Ahmed, A. Carranza, D. Arbour, A. Rao, S. Kim, and E. Koh (2019). Heterogeneous network motifs. *arXiv preprint arXiv:1901.10026*.

[41] Rossi, R. A., N. K. Ahmed, and E. Koh (2018). Higher-order network representation learning. In *Companion Proceedings of the The Web Conference 2018*, pp. 3–4.

[42] Rotabi, R., K. Kamath, J. Kleinberg, and A. Sharma (2017). Detecting strong ties using network motifs. In *Proceedings of the 26th International Conference on World Wide Web Companion*, pp. 983–992.

[43] Rudi, A. G., S. Shahrivari, S. Jalili, and Z. R. M. Kashani (2012). Rangi: a fast list-colored graph motif finding algorithm. *IEEE/ACM transactions on computational biology and bioinformatics 10*(2), 504–513.

[44] Saha, T. K. and M. Al Hasan (2015). Finding network motifs using mcmc sampling. In *Complex Networks VI*, pp. 13–24. Springer.

[45] Schbath, S., V. Lacroix, and M.-F. Sagot (2008). Assessing the exceptionality of coloured motifs in networks. *EURASIP Journal on Bioinformatics and Systems Biology 2009*, 1–9.

[46] Shahrivari, S. and S. Jalili (2015a). Distributed discovery of frequent subgraphs of a network using mapreduce. *Computing 97*(11), 1101–1120.

[47] Shahrivari, S. and S. Jalili (2015b). Fast parallel all-subgraph enumeration using multicore machines. *Scientific Programming 2015*.

[48] Shervashidze, N., S. Vishwanathan, T. Petri, K. Mehlhorn, and K. Borgwardt (2009). Efficient graphlet kernels for large graph comparison. In *Artificial intelligence and statistics*, pp. 488–495. PMLR.

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

[50] Small, M. and D. Cavanagh (2020). Modelling strong control measures for epidemic propagation with networks—a covid-19 case study. *IEEE access 8*, 109719–109731.

[51] Solava, R. W., R. P. Michaels, and T. Milenković (2012). Graphlet-based edge clustering reveals pathogen-interacting proteins. *Bioinformatics 28*(18), i480–i486.

[52] Toivonen, H., A. Srinivasan, R. D. King, S. Kramer, and C. Helma (2003, 07). Statistical evaluation of the Predictive Toxicology Challenge 2000–2001. *Bioinformatics 19*(10), 1183–1193.

[53] Wang, B., Y. Sun, T. Q. Duong, L. D. Nguyen, and L. Hanzo (2020). Risk-aware identification of highly suspected covid-19 cases in social iot: A joint graph theory and reinforcement learning approach. *Ieee Access 8*, 115655–115661.

[54] Wernicke, S. (2005). A faster algorithm for detecting network motifs. In *International Workshop on Algorithms in Bioinformatics*, pp. 165–177. Springer.

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

[56] Yang, S. (2013). Networks: An introduction by mej newman: Oxford, uk: Oxford university press. 720 pp., $85.00.

[57] Zhang, L., R. Hong, Y. Gao, R. Ji, Q. Dai, and X. Li (2015). Image categorization by learning a propagated graphlet path. *IEEE Transactions on Neural Networks and Learning Systems 27*(3), 674–685.

[58] Zhen Zhang, Jiajun Bu, M. E. Mutagenicity dataset. In *Hierarchical Graph Pooling with Structure Learning*.