

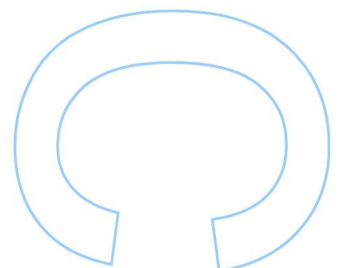
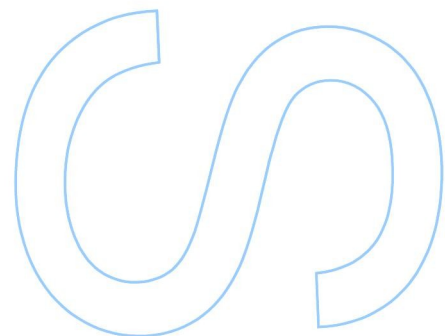
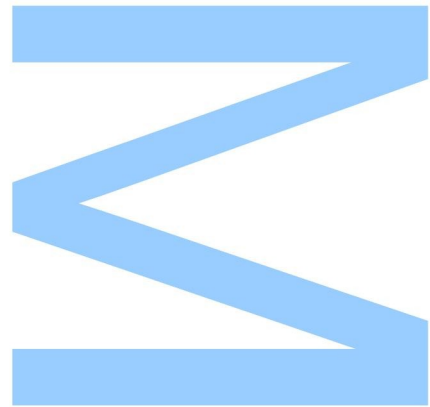
# Adaptive Parallel Subgraph Sampling in Large Complex Networks

André Eduardo Neves Cascais

Mestrado em Ciência de Computadores  
Departamento de Ciência de Computadores  
2018

**Orientador**

Pedro Manuel Pinto Ribeiro, Professor Auxiliar,  
Faculdade de Ciências da Universidade do Porto







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

O Presidente do Júri,

Porto, / /



# Abstract

Networks can be found in various domains such as biology, sociology and economics [5]. When analyzing a particular network, it is important to count some of its sub sections, this primitive is called subgraph count and its relevance has been growing lately due to its ability to characterize and describe a network. Motif finding [20] and computing graphlet degree distributions [24] are only a few applications of this primitive.

Counting subgraphs is a particularly difficult task since it is related to the subgraph isomorphism problem which is known to be in the NP-Complete complexity class [31]. The inherent difficulty of this task makes it hard to scale the problem to both large networks and high order subgraph sizes.

Two approaches used to smooth these types of problems are to sample the problem, trading a lower result accuracy for faster execution times, and take advantage of parallelism, lowering the elapsed time as more processing units are included.

In this work, we will apply both those ideas at the same time and propose an adaptive strategy which is able to mold itself to the given network along with the subgraphs to be count. Since we can't know the time that a particular task will take beforehand, we start with conservative, small sampled tasks and tune its characteristics to increase its duration in a controlled environment until an ideal task, that runs in the desired time frame, is crafted. We name that ideal time interval the *sweet spot* and both defining and reaching it in a rapid fashion are core goals of this work.

The results attained by using these ideas are very promising as we found very accurate counts with limited time frames to be computed.



# Resumo

Redes podem ser encontradas em variados domínios tais como biologia, sociologia e economia [5]. Ao analisarmos uma rede em particular, pode ser importante contar algumas das suas subsecções, esta primitiva é denominada contagem de subgrafos e a sua relevância tem vindo a crescer recentemente devido à sua habilidade de caracterizar e descrever a rede em questão. Contagem de Motifs [20] bem cálculo de distribuições de graphlet degree [24] são apenas algumas das aplicações que usam como base esta primitiva.

A contagem de subgrafos é uma tarefa particularmente difícil que está relacionada com o problema do isomorfismo de subgrafos que está na classe de complexidade NP-Completa [31]. A dificuldade inerente desta tarefa faz com que seja complicado escalar o problema para redes de dimensões superiores, bem como subgrafos de tamanhos maiores.

Existem duas estratégias utilizadas para atacar estes tipos de problemas: trocar uma menor precisão de resultados por tempos de execução e tirar partido de paralelismo.

Neste trabalho, vamos tentar aplicar ambas estas ideias em simultâneo e propusemos uma estratégia adaptativa que se consegue moldar quer à rede, quer ao conjunto de subgrafos a serem contados. Visto que não é possível saber o tempo que uma determinada tarefa vai demorar, começamos com tarefas pequenas e conservativas e vamos afinando as suas características com o objetivo de aumentar o seu tempo de execução num ambiente controlado, até chegarmos ao intervalo de tempo desejado. Esse intervalo é denominado *sweet spot* e temos como objetivo fulcral deste trabalho defini-lo e atingi-lo rapidamente.

Os resultados obtidos ao usar as estas ideias foram bastante promissores visto que conseguimos contagens muito precisas com tempos de execução limitados.





# Agradecimentos

Quero agradecer ao meu orientador, professor Pedro Ribeiro, por toda a ajuda e disponibilidade mantida ao longo do ano. Das diversas reuniões que tivemos, saíram muito bons momentos que me marcaram e fizeram crescer não só como programador mas principalmente como pessoa.

Quero agradecer também aos meus pais por me terem dado liberdade académica para escolher o meu percurso e por terem feito tudo para o facilitarem ao máximo.

Gostava de agradecer à FCUP pelos espaços disponibilizados para realizar este trabalho e em particular ao DCC por me ter fornecido a máquina na qual realizei as experiências apresentadas nesta tese.

Quero deixar também uma mensagem a toda a minha família e amigos que, quer por me terem ajudado a fazer este trabalho, quer por terem proporcionado bons momentos, tornaram este ano especial.

**Tese dedicada ao meu avô Didó.**

# Contents

<b>Abstract</b>	<b>i</b>
<b>Resumo</b>	<b>iii</b>
<b>Agradecimentos</b>	<b>v</b>
<b>Contents</b>	<b>ix</b>
<b>List of Tables</b>	<b>xi</b>
<b>List of Figures</b>	<b>xiii</b>
<b>Acronyms</b>	<b>xvii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Thesis organization . . . . .	3
<b>2 Background and Related Work</b>	<b>5</b>
2.1 Concepts and Terminology . . . . .	5
2.1.1 Subgraph Count . . . . .	6
2.1.2 Network Motifs . . . . .	6
2.2 Subgraph Count Algorithms . . . . .	7
2.2.1 MFinder . . . . .	8
2.2.2 ESU . . . . .	8
2.2.3 Symmetry breaking conditions . . . . .	9

2.2.4	G-Tries	10
2.2.5	Other sequential algorithms	11
2.3	Parallel Approaches	12
2.3.1	Parallel G-Trie for Distributed Architectures	12
2.3.2	Parallel G-Trie for Shared Memory	13
2.3.3	Parallel ESU for discovery of motifs	13
2.3.4	More parallel approaches	14
2.4	Subgraph sampling	15
2.4.1	Kashtan	15
2.4.2	Rand-ESU	15
2.5	Alternative graph representations	15
<b>3</b>	<b>Methodology</b>	<b>17</b>
3.1	Overview	17
3.2	Parallel methodology	19
3.2.1	Parallel opportunities in the subgraph counting problem	19
3.2.2	Revamped G-Trie	19
3.2.3	Master-Slave Architecture	20
3.2.4	Adapting Master-Slave to the subgraph counting problem	20
3.2.5	Finishing criteria	22
3.2.6	Master-Slave work flow	22
3.3	Adaptive Methodology	24
3.3.1	Sweet Spot	24
3.3.2	Reaching our ideal task	26
3.4	Approximating the counting	26
3.4.1	0 probability run and new ideas	28
3.4.2	Automating parameters	28
<b>4</b>	<b>Experimental Results</b>	<b>31</b>

4.1	Network Data . . . . .	31
4.2	Scalability of parallel methodology . . . . .	33
4.3	Reaching the Sweet Spot . . . . .	34
4.4	Quality of the approximation . . . . .	38
<b>5</b>	<b>Conclusion</b>	<b>45</b>
5.1	Contributions . . . . .	45
5.2	Future work . . . . .	45
	<b>Bibliography</b>	<b>47</b>



# List of Tables

- 2.1 Description of subgraph count algorithms along with its release date, the type of strategy used and the possibility of a sample option . . . . . 8
  
- 4.1 Different networks used . . . . . 31
- 4.2 Comparison of the number of tasks computed with a different number of cores among different networks . . . . . 34
- 4.3 Comparison of the time required to reach the `sweet spot` computed with a different number of cores among different parameters for the `Brightkite` network and `subgraph 4 count` . . . . . 36
- 4.4 Time elapsed in order to reach the `sweet spot` of [10 – 15] seconds in the most time consuming network `flickr` with size 3 across different parameters . . . . . 37
- 4.5 Time elapsed in order to reach the `sweet spot` of [2 – 4] seconds in the most time consuming network `flickr` with size 3 across different parameters . . . . . 37
- 4.6 Time elapsed in order to reach the `sweet spot` of [2 – 4] seconds in the most time consuming network `flickr` with size 4 across different parameters . . . . . 38





# List of Figures

- 1.1 The Dolphins Network along with its 3 5-clique subgraphs. . . . . 2
- 2.1 Motif example in a directed network. Picture taken from [25] . . . . . 7
- 2.2 ESU search tree to count all occurrences of 3 sized subgraphs in a small network. 9
- 2.3 G-Trie for all 4 size subgraphs . . . . . 10
- 3.1 Plot showcasing a typical fashion in which a sweet spot is reached . . . . . 25
- 4.1 Tasks completed by slaves across different singular task execution times. The results are taken from computing the census in the Brightkite network for the size 4 subgraphs with 15 minutes to run. . . . . 33
- 4.2 Sweet spot ping pong . . . . . 35
- 4.3 Labels of the different  $k$  sized subgraphs. This label is achieved by performing a DFS on the G-Trie 2.3 . . . . . 38
- 4.4 Progression of both individual and aggregated results in the Brightkite network sized 4 across different subgraph types. Every task run was done with  $1e^{-7}$  probability. . . . . 39
- 4.5 Precision across different sweet spots subgraph types for the Brightkite network size 4. This results were obtained after 100 seconds of run time . . . . . 40
- 4.6 Precision across different sweet spots subgraph types for the Brightkite network size 4. This results were obtained after 300 seconds of run time. . . . . 41
- 4.7 Precision across different sweet spots subgraph types for the Brightkite network size 4. This results were obtained after 15 minutes of run time. . . . . 42
- 4.8 Average precision across different fractions of exact time. Brightkite network with 4-size subgraphs . . . . . 43



# List of Algorithms

1	High-level methodology overview . . . . .	17
2	High-level Sweet spot find method . . . . .	18
3	Master Workflow . . . . .	23
4	Slave Workflow . . . . .	23
5	High-level sweet spot reaching algorithm . . . . .	26



# Acronyms

<b>SS</b>	Sweet Spot	<b>CPU</b>	Central Processing Unit
<b>MS</b>	Master-Slave	<b>GPU</b>	Graphics Processing Unit
<b>IP</b>	Initial Probability	<b>RAM</b>	Random-access Memory
<b>IM</b>	Initial Multiplicative Factor	<b>DFS</b>	Depth-First Search
<b>MF</b>	Multiplicative Factor		
<b>MPI</b>	Message Passing Interface		



# Chapter 1

## Introduction

Complex networks have become more popular over the years due to their inherent ability to solve multidisciplinary problems once they are modeled accordingly. Applications of network science methods can be found in fields such as mathematics, physics, biology or sociology [5]. With the surge of the internet, the volume of data available on different fields is increasing exponentially making the efficient study of these complex systems the more important. The social problem of finding the minimum distance connecting two different persons [2] and route planning in road/transport networks [5] are only a few examples of real world applications taken from the study of network science.

Since the applications of this area are rapidly expanding to every scientific field, different and more complex methods of analyzing the respective networks are required depending on the desired goals. Computing the average vertex degree, finding sets of closely connected components to define them as communities [4] or discovering patterns in networks that are over represented when comparing to an arbitrary network with similar characteristics (motifs [20]), are only a few ways to synthesize and extract valuable information from a given network. Counting a particular set of subgraphs in a large network can thus be seen as a primitive task used by more advanced metrics in order to extract valuable information.

Even though the previous mentioned task of counting subgraphs can be seen as a primitive task, its computational complexity is not "primitive" at all. This count is bounded by the *subgraph isomorphism problem*, that is the problem of, given two graphs  $G$  and  $H$ , finding whether  $G$  contains a subgraph isomorphic to  $H$ , is known to be in the NP-Complete class [15]. This restriction makes it so that counting high order subgraphs or counting them in a large network cost too much computational time with small increases to those dimensions. To showcase how fast this problem can scale, consider the Brightkite network presented at 4.1, counting all subgraph types with 3 nodes took only a second to be computed and 12 million occurrences were found. If we change the subgraph size to 4, the task takes 100 seconds to complete and near 200 million subgraphs are found. Note that this experience was completed using the G-Trie exact count method [27], which is an efficient state of the art data structure which we describe in more detail in Section 2.2.4.

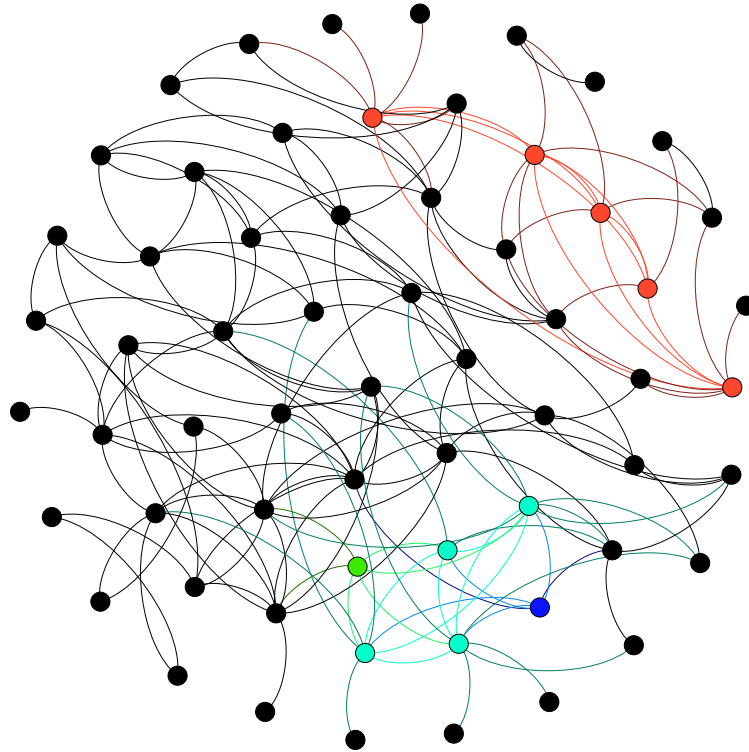


Figure 1.1: The Dolphins Network along with its 3 5-clique subgraphs.

With this exponential increase of complexity when increasing the size of the subgraphs wanted and/or the size of the queried network, different strategies have been used in order to reach those dimensions. Using parallelism is a way of taking full advantage of the available computational resources and parallel strategies are able to, in most cases, reach almost linear speedups meaning that we can divide the total time that a task would take to complete by the work units at our disposal. Another way in which this goal can be reached is by using sampling instead of having the exact count. It is possible to only search for a fraction of the subgraphs and if we were to extrapolate those results, we would get an approximate count rather than the exact one. This option is only viable if we can tolerate some error but it is really useful to get a general idea of the network's properties in a much smaller amount of time.

The main purpose of this work is to tackle the mentioned problems and to be able to search for higher end subgraphs in larger networks. We chose to adapt a state of the art data structure



and algorithm called G-Tries [27] so that we would use both parallelism and sampling to lower computational times and make it possible to run more difficult tasks. Another problem we tried to tackle is the difficulty of knowing beforehand how much time a certain task in a given network would take since a seemingly lower order subgraph count in a network can be run in seconds and can take hours or days if the network were to change. In order to respond to that we present an adaptive scheme that changes and adapts its strategy to accommodate to the given network and subgraphs as good as possible with no previous information.

In this work, the count of subgraphs of a given size is done in an adaptive fashion, that means that we want to start with small and controlled tasks and successively tweak some parameters in order to adapt those so that we can create jobs that run in a desired time frame. We use a Master-Slave parallel scheme so that the Slaves can focus on doing pure computation while the Master makes sure to process each result as it arrives, crafting the next tasks using every previous result and in the end it will deliver its best guess to the subgraph counting problem. The user of our work is able to select a time limit in which he wants the results to be delivered and it is the Master's job to make good use of that time frame. The notion of a sweet spot is present in this work and it is related to a time interval in which our ideal task should run. If we have 1 minute to run our method and set the sweet spot to 1 second, the desired goal is to start by finding a single job that takes 1 second to run, and then run it as close to 60 times as possible. A revamped G-Trie data structure 3.2.2 is used so that multiple slaves can work on it at the same time with possibly different tasks.

## 1.1 Thesis organization

This work is divided into 5 chapters.

Chapter 1 starts of by introducing subgraph counting and gives a high level overview of its limitations along with some ways in which we propose to tackle those problems.

Chapter 2 introduces terminology and related problems while describing some existing approaches for subgraph counting. These approaches are divided into three categories: classical, parallel and sampling. In this chapter, we also dive into ways of representing a network when the adjacency matrix is not small enough to fit into memory.

Chapter 3 showcases our solution to the problems presented earlier while diving into its methodologies and difficulties. We also try to justify some of the choices made along the way and present details on the different components that are part of our final project.

Chapter 4 is reserved for experimental results. Here we try to show results that can back up our previous choices in the different degrees of freedom associated with this work.

Chapter 5 rounds up this thesis by summing up the work done and highlighting our contributions and then future work is reflected upon.



## Chapter 2

# Background and Related Work

### 2.1 Concepts and Terminology

- A graph  $G$  is a set of vertices  $V(G)$  along with their connections  $E(G)$ . The size of a graph is usually determined by its number of vertices  $|V(G)|$ . An edge  $(u, v) \in E(G)$  if there is a connection from  $u$  to  $v$ .
- In a directed network, the order of the vertex on an edge is relevant since  $(u, v) \in E(G) \not\Rightarrow (v, u) \in E(G)$ . In non directed graphs however, this is not true since all of the edges in this type of network are bidirectional.
- A graph is typically represented by a adjacency matrix  $Adj$  which is a  $|V(G)| \times |V(G)|$  boolean matrix.  $Adj[u][v] = 1$  if  $(u, v) \in E(G)$ .
- A graph  $H$  is said to be a subgraph of  $G$  if  $E(H) \subseteq E(G)$  and  $V(H) \subseteq V(G)$ . That subgraph is said to be induced if  $\forall_{u,v} \in V(H), (u, v) \in E(H) \iff (u, v) \in E(G)$ . A occurrence of  $H$  in  $G$  happens if there is a set of vertex that induce  $H$ . We define multiple occurrences as different set of vertex that induce a particular subgraph  $H$ .
- A vertex  $u$  neighborhood  $N(u)$  is the set of vertex  $v$  so that  $(u, v) \in E(G)$ . A vertex  $u$  exclusive neighborhood respective to a vertex set  $S$  is composed by  $v$  so that  $v \in V_{exc}(u, S) \iff v \in N(u), v \notin N(w) \forall w \in S$ .
- A  $k$ -sized graph/subgraph is a network with size  $(|V(G)|) k$ . A  $k$ -clique is a graph of size  $k$  in which every vertex is connected to every other  $\forall_{u,v} (u, v) \in E(G)$ .
- The degree of a vertex  $v$  is the number of different vertex  $u$  to which it is connected. In a directed network we can differentiate between indegree and outdegree depending if we are considering  $u, (u, v) \in E(G)$  or  $u, (v, u) \in E(G)$ . In non directed networks, there is no need to distinguish these measurements since they are always the same.
- In the remaining sections, we will be considering simple graphs, that is unweighted graphs with no self loops  $(u, u) \notin E(G)$  and no multi edges, meaning that there is only, at most,

one edge connecting two vertex. We also assume a label of the vertex so that a size- $k$  graph will have nodes labeled  $1..k$ .

- A permutation is a unique ordered way in which we can arrange a set of elements. Two permutations are distinct if at least one of the elements is in a different position in the set. For a set of  $k$  elements, there are  $k!$  unique permutations.
- We define two graphs  $G_1$  and  $G_2$  as isomorphic if they have the same size and there is a permutation  $p$  of  $G_2$ ,  $G_2^p$  so that  $\forall_{u,v}, (u,v) \in E(G_1) \implies (u,v) \in E(G_2^p)$
- An automorphism is a particular type of isomorphism that maps vertices from  $G$  so that the resulting graph is isomorphic to  $G$ .

### 2.1.1 Subgraph Count

Given a network  $G$  and a size  $k$ , the subgraph counting problem lies on counting all occurrences of each different size  $k$  subgraph on the original network. A occurrence of a particular subgraph is defined as a distinct induced group of vertex.

### 2.1.2 Network Motifs

Even though this work is not intended to be focused solely on motif discovery, this task is still one of the main problems we are trying to tackle.

Motifs were introduced by Milo et al. [20] as patterns whose frequency is much higher than the one found in similar random networks. This task is not to be confused with Frequent Graph mining since a subgraph with high frequency in the original network is not necessarily a motif.

Most of the approaches to the motif discovery problem can be divided in four main tasks:

- Finding and counting all of the  $k$ -sized subgraphs occurrences in the original network
- Generate a set of similar random networks.
- Count the frequency of each of the original subgraphs in the random networks
- Compute the significance for each subgraph type

When considering a network  $G$ , we define a similar random network as a graph with the same size as  $k$  with the same number of edges in which the node degrees are preserved. The significance that is calculated at the end of the motif find procedure is usually the  $Z$ -score which is a measure that takes into consideration the number of standard deviations from a score to the mean of all the results.

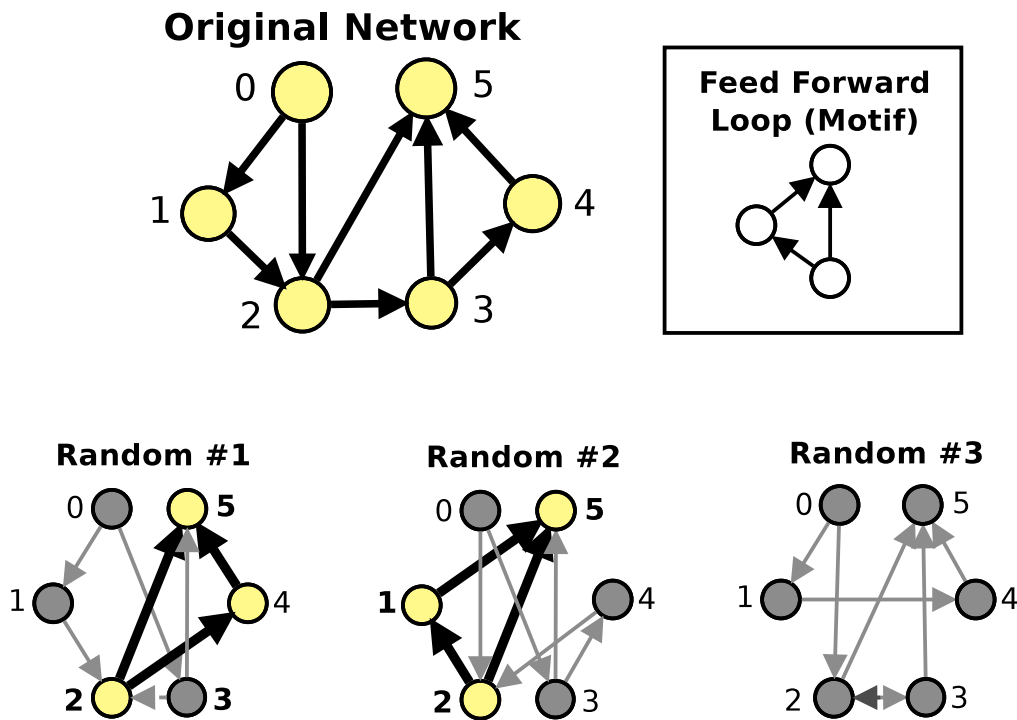


Figure 2.1: Motif example in a directed network. Picture taken from [25]

In picture 2.1 we can see an example of a motif in a subgraph along with the respective similar random networks. The Feed Forward Loop presented only has one or zero occurrences on the generated networks while it can be found multiple times in the original network. We can see that the random networks preserve both indegree and outdegree for every vertex.

## 2.2 Subgraph Count Algorithms

We can classify the algorithms for subgraph counting in two major groups. The *network-centric* approach that fully extracts the count of all subgraphs with the desired size and then matches each of those with the matching isomorphic type. *Subgraph-centric* options start with a specific isomorphism class and count its occurrences on the network. A *set-centric* approach can also be considered as an intermediate approach between the two major ones if we are neither searching all of the subgraphs, neither only one of them but rather a specific set. One can, for instance, only be interested in 4-cliques and "squares" while doing a subgraph count of size 4 on a certain Network.

We will now showcase the classic subgraph count algorithms in a chronological order and focus on strategies which are pillars to this work. Afterwards sampling and parallel options will be analyzed.

Algorithm	Year	Strategy	Sample
MFinder	2002	Network	No
ESA	2004	Network	Yes
ESU	2005	Network	Yes
Grochow	2007	Subgraph	No
G-Trie	2010	Set	Yes

Table 2.1: Description of subgraph count algorithms along with its release date, the type of strategy used and the possibility of a sample option

### 2.2.1 MFinder

Milo’s algorithm for counting the number of occurrences of all the  $k$ -sized subgraphs [20] was the first attempt at solving the problem and had some flaws. It exhaustively found all the subgraphs with the desired size and stored them in a table. After each occurrence find, the algorithm would check for an isomorphism match of the respective vertices in the table and would only store it if that isomorphism was not found, meaning that we are exploring a new occurrence. For each subgraph to be found, every permutation of its nodes would be reached, that sums to  $k!$  occurrences instead of only one along with the need to traverse the search tree that many times.

### 2.2.2 ESU

Wernicke [33] tackled a large portion of the problems laying in the available motif finding strategies. He not only proposed a faster algorithm for the enumeration of all the size- $k$  subgraphs on the original network but also presented an alternative for the random network generation.

At any stage of this algorithm we have a set of vertex and choosing one by one we try to expand them. Choosing a vertex, we expand them to a set with vertex with higher label and who are only neighbours with the chosen vector across the partial subgraph created so far. If we have two vertex  $a$  and  $b$  in our partial subgraph and we are expanding  $a$ , the expanding set is composed by vertices with label higher than  $a$  which are not neighbours with  $b$ . The labeling restriction helps with symmetry breaking since we only extend a vertex from another and not the other way around and it is this property that makes sure that each subgraph is not only found but is found exactly once. After this step the algorithm will try to recursively extend a vertex at a time and stop if either the desired subgraph size is reached or no vertex can be extended under the two previous stated conditions. This method provides a computing tree and below the root are the sets of each vertex coupled with its extension set. Figure 2.2 showcases the algorithm flow and the fashion in which it makes sure to only count a occurrence once.

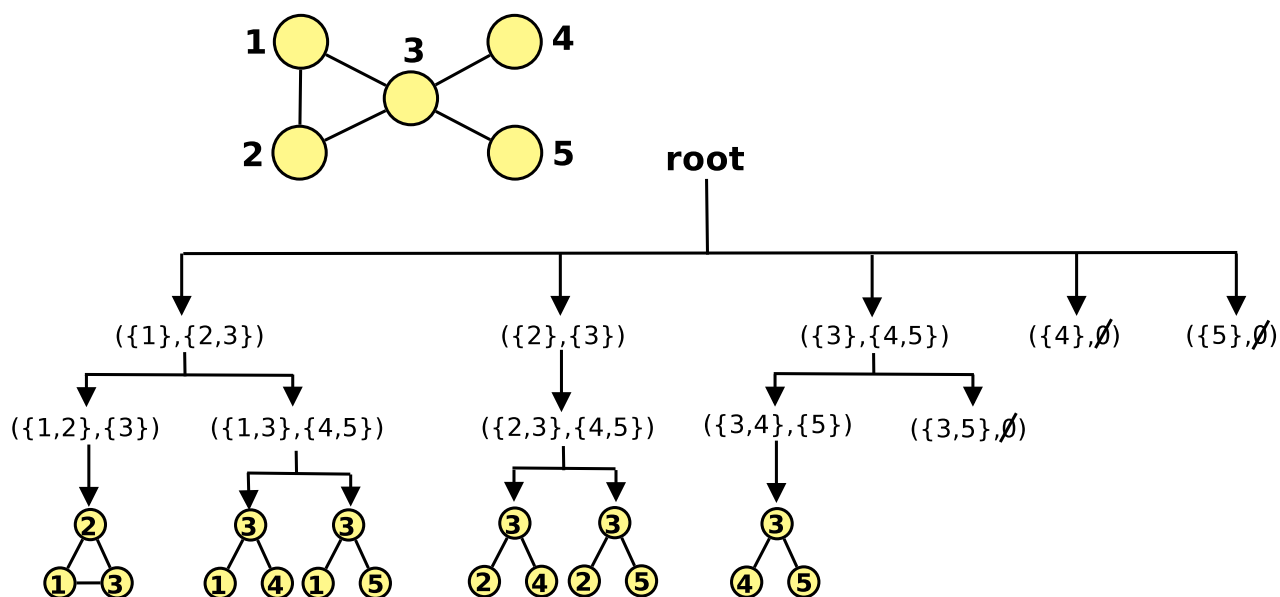


Figure 2.2: ESU search tree to count all occurrences of 3 sized subgraphs in a small network.

### 2.2.3 Symmetry breaking conditions

Since the first algorithms for subgraph count and consequent motif find had size limitations, Grochow et al. [7] presented a new idea based on a singular query subgraph, a subgraph-centric approach. A complete census would be computed in this fashion by using a graph enumeration tool such as McKay's `geng` and `directg` [18].

In order to address the problem of finding the same subgraph more than once due to isomorphisms, Grochow et al. [7] purposed a novel algorithm to limit those tests using symmetry breaking conditions. The isomorphism tests to see what type of subgraph was found were also improved in this method.

Successive calls to find isomorphisms for the wanted subgraph onto the original network are made in a backtracking search. This phase tries to select nodes with many neighbors which are already mapped and whose degree is higher. A single call to this method extends the partial match by a single node. That new node has to be connected to the remaining nodes according to the wanted subgraph type isomorphism we are trying to find. These tests at early stages made sure that we are able abort computations in earlier stages, which was not possible in previous methods. A single subgraph type can be mapped in a network in different ways. For instance, a triangle  $a, b, c$  can be mapped into the 3-clique subgraph type in 6 different ways. In order to avoid this repetition due to symmetries, some restrictions are computed and those symmetry breaking conditions ensure that there is a unique match of a group of vertex to a subgraph. Simply ignoring automorphisms and count the same occurrence more than once (in a different order) and then dividing the final count by the respective permutations is a naive approach that would take excessive time. Grochow's method only detects each subgraph exactly once thus saving time.

Using a labelling for the nodes, conditions of the form  $L(n) < L(m)$  for two nodes, we make sure that any ambiguity due to them being in the same equivalence class is settled. This conditions are enforced until all the nodes are fixed.

### 2.2.4 G-Tries

The G-Tries are a novel data structure proposed by Ribeiro and Silva [27] with the purpose of improving the available algorithms and speeding up the process of subgraph counting. This data structure is the main pillar of this work.

It takes advantage of common substructure between graphs, like a prefix tree would, being able to compress space usage. G-Tries also uses symmetry breaking conditions in order to make sure we find each subgraph only once and not waste redundant computation time. Besides being fast, G-Tries are also generalist, being able to be used in most graph types such as directed, weighted or colored. It also allows for a query on a particular set of subgraphs, from a single one to the all of the  $k$ -sized subgraphs.

In a string trie, every node represents a single letter, in a G-Trie the node represents a vertex in the graph. Having two nodes with the same parent in the G-Trie means that you will have the same graph if you remove that node.

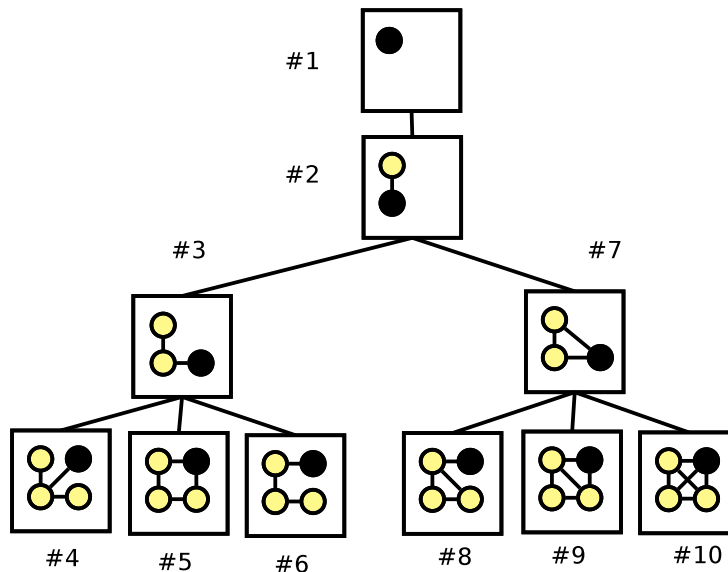


Figure 2.3: G-Trie for all 4 size subgraphs

The process of creating a G-Trie is also similar to creating a trie. We start with an empty tree and insert the desired subgraphs in an iterative fashion. We traverse the tree until we find a node in which none of the children matches our subgraph and in this case all of the remaining nodes will have to be created. The problem with this method is knowing in which order should we consider the subgraph vertex since a different order could lead to a different G-Trie. A



canonical form has to be used in order to shape the `G-Trie` in a deterministic fashion and there is a custom canonical form designed in order to increase the memory compression by identifying and taking advantage of common substructure. That canonical form is illustrated at the figure 2.3 and other less efficient canonical forms are shown to be less compressed [25].

`G-Tries` are also advantageous to use for space complexity reasons. This tree organization scheme occupies less memory and scales better for larger sizes. Increasing the amount of common substructures leads to a lower space usage that what we would have if we were to simply use the adjacency matrix for each of the queried subgraph [28].

After having the `G-Trie` ready, we can start counting the respective subgraphs on a given network. Here we will be partially matching the graph through the `G-Trie` until we either get to a final node or to a dead end. At a given node, we compute the set of remaining nodes that can still be added to the partial graph and choose one of them, check if it matches the current node, and if so, then recursively try to match the rest of the `G-Trie` and in the end, change the chosen vertex. We check if a vertex matches the a `G-Trie` node if it has the same connections to the previous vertices and if so we either continue the algorithm or stop if we reach a leaf. The former means that we found a specific subgraph from the desired set and in that case we can increment its count.

Some previous methods of counting did not include any automorphism tests and we would have to count the same subgraph multiple times ( $n!$ ) and divide the final count by that number similarly to what was done in [20]. `G-Tries` are also equipped with symmetry breaking conditions that remove the need to compute the same subgraph over and over. This idea was introduced by [7] and was adapted to the `G-Tries`. In each `G-Trie` node we have a list of conditions regarding the label of the nodes. For instance  $l(a) < l(b)$ ,  $l(d) < l(c)$  means that both the label of  $a$  has to be lower than the label of  $b$  and the same with  $d$  and  $c$ . The structure of the conditions is also ready to remove redundant conditions in order to decrease the time required to test them.

### 2.2.5 Other sequential algorithms

We chose to focus our attention on some classical algorithms which introduced fundamental steps that allowed for the former ones to be conceived. There are however many more ways available to tackle the subgraph count problem. There are some methods which are better for some cases and worse for others but there are also some fundamentally different strategies of tackling this problem.

`FaSE` [21] uses a customized `G-Trie` that is build during the algorithm execution on the fly, instead of building it before even starting any kind of computation. `FaSE` creates intermediate topological classes postponing temporally the isomorphism tests. This algorithm is divided into two steps, the enumeration, which is similar to other explored algorithms, and encapsulation, where the intermediate topological features are stored.

`QuateXelero` [10] is an exact method for subgraph count that tries to decrease isomorphism

tests and presents a quaternary tree data structure at its core. That quaternary tree is similar to the `G-Trie` and computes partial classifications for enumerated subgraphs.

There are some analytic algorithms that attempt to bypass the step of counting all occurrences by taking advantage of combinatorial characteristics of the networks. Due to the usage of those properties, this methods are often limited to particular network types and subgraph sizes.

`acc-Motif` [19] is an analytical algorithm to count subgraphs of size  $k + 2$  in a rapid fashion based on the count of induced subgraphs with size  $k$ . The main drawback of this method its inability to easily be adapted to larger sizes. This algorithm is conceived using combinatorial techniques along with a frequency histogram.

`ESCAPE` [23] provides an algorithmic framework used to compute the exact count for the 5-sized subgraphs. This analytical method uses pattern division and exploits degree orientations related to the graph in order to achieve high performance counts. In this method, only four subgraphs are required to be enumerated so that the final 5-size counts are outputted and this result is proven as part of the work. This method avoids the enumeration along with its inherent combinatorial complexity.

## 2.3 Parallel Approaches

In order to fully take advantage of a computer or a cluster of them, parallelism can be explored and in ideal scenarios, close to linear speedup can be achieved. This means that a task that took  $x$  units of time to compute would ideally take  $\frac{x}{n}$  units of time when  $n$  refers to the number of available cores. There is work in both distributed and shared memory related to subgraph counting. In distributed schemes, the programmer can count on multiple independent work units, each with their own memory. In shared programming, all the different work units share the same memory. Generally speaking, when using shared memory, the cost of communication is lowered but its harder to account for race conditions between different work units. There can also be hybrid approaches where there are multiple units, each with its own shared memory.

### 2.3.1 Parallel `G-Trie` for Distributed Architectures

The `G-Trie` data structure and census computation algorithm produces a search tree in which every call to a procedure is independent from one another. Noticing that and with the goal of accelerating the subgraph count problem, Ribeiro et al [26] adapted the former work to a distributed parallel scheme using the message passing interface (MPI) implementation OpenMPI [6].

In order to start a parallel algorithm it is useful to be able to divide the wanted work in logical pieces usually called work-units. In this particular algorithm every recursive call to the matching function, which tries to match a group of vertex with a new one in order to later associate it

with a size- $k$  subgraph, is considered as a work-unit and the main problem is to divide those work-units among all processors. Even though the algorithm is traversing a computation tree, the work load is not balanced among the nodes making it hard to statically divide the work without ending up with some workers doing substantially less work than others. The dynamic fashion in which this algorithm divides its work is a receiver-initiated scheme. Each processor chooses part of the available work and computes it while periodically checking for messages from the other processes. That check phase has the goal of seeing if any other processing is requesting more work and if that is so, the recursive calls are stopped and the computation state stored, the remaining work is divided and half of it is send to the message sender. Due to how imbalanced the associated tree can be, the work is divided in a round robin fashion and the processor who is giving part of its computation array keeps the currently explored nodes. This algorithms is tailored so that there is a threshold related to remaining work so that a unit with little computation to do doesn't spend time splitting it. This case would end up in the original processor simply asking a different thread for work division. In a final phase, each subgraph type count has to be aggregated in order to get the final count and this is accomplished by using the native reducing procedures of the used interface.

### 2.3.2 Parallel G-Trie for Shared Memory

When performing a census on a G-Trie, we are essentially traversing different tree branches that are independent from one another. Noticing that, Aparício et al. [1] adapted the previous G-Trie census algorithm to a Multicore Architecture in order to pallelize a single census. Simply dividing the branches in a static fashion was not good enough due to how imbalanced different  $k$ -sized subgraph counts can be for a single network. In this work each core is assigned a thread and the vertex are divided among them. Each thread will now compute the census as it would in a sequential fashion until it has no more work to do. When a thread finishes its work, it asks for another random thread to split some of the remaining work. Since different threads may find the same subgraph type with different sets of vertex, each thread would have a private count of each subgraph type and in the end, the global frequencies would be attained by summing every thread counts.

### 2.3.3 Parallel ESU for discovery of motifs

In order to speed up motif discovery, Ribeiro et al [26] provided a method of performing all of the required steps for motif discovery in parallel, namely the census and both the random network creation and subgraph count. Since in the ESU algorithm [33] the calls to extend a partial subgraph are independent for each other, they can be computed in parallel with no synchronization required. After being able to divide the bigger picture into smaller tasks, two methods are presented for managing the work distribution. A Master-Slave strategy and a Distributed strategy. The first one has a dedicated worker "master" that divides work among the remaining workers while those remaining "slaves" are doing the computations. The distributed

strategy has all of the workers responsible for both computing and communicating to divide work. Before starting the main computations, a pre-processing phase takes place. The required jobs are divided among workers either by sending all of it to a single unit, expecting that other workers will steal some of the work, or by statically dividing the jobs among work units in a round-robin manner. In the Master-Slave strategy, the master is continuously receiving messages from its slaves. If it gets a message requesting work it can either send unprocessed work or add that slave to a queue of workers waiting for jobs. If it receives a message with more work, it can send it directly to an idle worker or add it to the work queue. This process ends if all of the workers are idle. In the Distributed option, the work division is made by all of the cores which are all equal in this strategy. Each worker keeps computing its work until it has no more computation to do. In that case it asks another random worker for part of its remaining computations and continues. The choice of another core to steal work from is random due to how imbalanced and unpredictable the tree branches can be. If a request for more work is made to another core waiting for work, another random choice is made and if that process keeps repeating among all the workers, we can conclude that there is no more computation to be made and the count process is finished. The final step of this process is to gather the results from each worker to a chosen one, responsible for the significance calculation. This is done by fixing an order in a vector. A position in that vector denotes both the network and the count of a particular subgraph type. The workers are then organized in a binary tree fashion with the selected worker at the top. After aggregating all of the results, that root worker sequentially computes the significances and later return the found motifs.

### 2.3.4 More parallel approaches

Even though we dived into the parallel options which were the most related to our work, there is more to parallelism in graph analysis.

Ahmad and Ribeiro [29] tackled the subgraph count problem in a parallel fashion using MapReduce. MapReduce is a programming model aimed to process large amount of data and it is inspired by the map and reduce operations available in the functional programming paradigm. The presented method works on top of `G-Tries`, it is able to dynamically share work among the available work units and it is directed to cluster computing in the cloud.

Lin et al. [14] takes advantage of parallelism in a different manner. Their work focus is on Graphical Processing Units (GPU) and its inherent parallel aspects due to its high number of processing units in order to simultaneously perform different subtasks related to motif discovery. The CUDA framework is used in this work in a very detailed fashion and its respective results are not only competitive to the existing CPU approaches timewise but also cost effective when the prices of these different processing units are considered.

An hybrid option between CPU and GPU usage is presented by RA Rossi and R Zhou [30]. By simultaneously taking advantage of both these types of processing units along with its different strengths, a fast algorithm for extracting information from a network regarding induced

$k$  subgraphs is presented. The results obtained by this method fully use the available processing resources and are not only fast but also cost and energy efficient.

## 2.4 Subgraph sampling

With the goal of being able to aim for higher order subgraphs or to be able to find them in larger networks, there has been work dedicated to trade accuracy for speed in the form of sampling. Most of the sequential algorithms presented in the respective section can be adapted to search only a given fraction of the subgraphs rather than the whole set. This is typically done by moving in the computing tree only with a certain probability when there is a tree like structure. The way of distributing the fraction across levels is not obvious and can be achieved using different strategies.

### 2.4.1 Kashtan

Kashtan et al. [9] introduced the concept of sampling in motif discovery. After the initial census on the original network, this method would sample each of the random networks instead of exhaustively counting each subgraph occurrence by picking a random edge from the graph and then iteratively choosing a random neighbour edge until the desired size is reached. The sampled subgraph is then composed by all of the nodes found along with all of the edges between them (not only the expanded ones in the sampling process).

### 2.4.2 Rand-ESU

The ESU algorithm was presented along with a sampling version [33], making it possible to effectively trade speed for result accuracy. In order to sample the ESU-tree a given probability is divided across all the levels except the last one so that the product of each partial probability on the levels was the original probability. If there are  $k$  levels, assigning a probability of  $p^{\frac{1}{k}}$  on each level would assure the previous condition. Wernicke noticed that a cut on a branch closer to the tree would highly influence the rest of the run compared to a cut closer to the leaves.

## 2.5 Alternative graph representations

We are trying to explore the subgraph problem on large and complex network in which the size of the graph has to be addressed. Since a typical representation of a graph as an adjacency matrix takes  $O(V^2)$  size, for large graphs, this may not fit in memory and alternative representations have to be explored.

Paredes and Ribeiro. [22] studied this issue and proposed several different options to represent

a graph so that it would be small enough to fit in memory. Of course that we are giving up the  $O(1)$  cost of the operation to see if there is a connection between two nodes and so the execution times are expected to be higher when using this options. Graph representations such as a sorted list so that binary search can be used and hash table based on both nodes and edge are explored in depth. In the end Paredes uses a hybrid approach, combining several methods, and reaches performances only two times slower than the adjacency matrix.

# Chapter 3

## Methodology

In order to solve the subgraph counting problem, we propose a high-level adaptive strategy whose goal is to be able to deliver the best guess possible relative to the frequency of each subgraph type in a given time frame. Afterwards we concretize this idea using the `G-Trie` data structure with its sample version as the base and modify in order to make parallelism possible. Sampling is used since we are trying to search complex networks in which the exact count would take too much time and the parallel component will make sure that we can do more computation in that selected time frame, compared to a sequential approach.

This work is directed so that it is possible to shape the algorithm according to the queried network's characteristics, the subgraph size and the available time. We want to find a task that is able to run in a desired time frame and once task is crafted, we can complete it over and over until the available time is over. The ideal `sweet spot` is a time frame in which we want our computation to fit. The goal is to tune some parameters, namely the fraction associated with sampling, so that we can rapidly craft a task that would lead to that `sweet spot`.

### 3.1 Overview

---

**Algorithm 1** High-level methodology overview

---

```
1: idealTask ← findSweetSpotTask()
2: while not timeIsOver() do
3:   idealTask
4: end while
5: aggregateResults()
6: outputSubgraphCount()
```

---

**Algorithm 2** High-level Sweet spot find method

---

```

1:  $task \leftarrow initialTask$ 
2: while  $elapsedTaskTime \notin idealInterval$  do
3:    $task \leftarrow refineTask()$ 
4: end while
5: return  $task$ 

```

---

A simplified version of our methodology is described at algorithm 3.1. We start by finding our ideal task (line 1) and then repeat it until the time period given by the user runs out (lines 2 - 3). After the time runs out, we gather the results and output them (lines 5 - 6). Note that the definition of the ideal task can be provided by the user or by some automatic parameters suggested at sections 3.4.1 and 3.4.2 and besides that, the aggregation phase can be updated as new results arrive.

The general process of finding the ideal task, given a time frame, is described in algorithm 2. We start by computing some initial task (line 1) and proceed by refining it until it fits the desired time frame (lines 2 - 4). After the "perfect" task is crafted, the process stops, and its characteristics are returned (line 5).

We want to make sure that we can run this method in networks in which the time required to do a complete census would be overwhelming. With that in mind we chose to start with a small, conservative sampling fraction of the desired count such as  $10^{-10}$ , measure the time associated with that fraction and change it until it fits our time goal for a task. Along with that initial probability  $initialProb$ , we also provide a multiplicative factor  $initialMultFactor$  to determine the next probability to be tried. A single task in this algorithm can still take too much time to run and in this case, we decided to lower the probability to the last "feasible" task, change the multiplicative factor and continue from that point.

The Sampled G-Trie presented by Ribeiro and Silva [26] is the algorithm used to do the successive sampled census required. More information about what fraction is used and its distribution among the different G-Trie nodes will be detailed at section 3.2.4.1. Parallelism is included in this methodology so that we can achieve the desired sweet spot faster, by sending different workers to compute different tasks. Also, parallelism will make sure that we can have more repetitions of the final, ideal job as more cores are added. In order to enable parallelism, the original G-Trie was modified so that it would support multiple units performing computations on the structure simultaneously. This is described at section 3.2.2.



## 3.2 Parallel methodology

### 3.2.1 Parallel opportunities in the subgraph counting problem

Parallelism was considered and applied to this work with the main goal of making it possible to run potential different jobs independently from one another at the same time. Each job can be assigned to a single work unit with its own sampling probability distribution. Parallelism could also be applied to make a single run faster [1] but it is not the main focus of this work.

### 3.2.2 Revamped G-Trie

In order to be able to run our parallel algorithm, the G-Trie structure had to go through some changes. It was possible to have a copy of the G-Trie in each work unit and simply run the wanted tasks in that particular copy but that would possibly take too much space. With the goal of lowering that space usage, and given the fact that we would be mostly doing memory reads until the count increment stage after a match between the original network and a particular subgraph type is achieved, we adapted the base structure so that it was possible to run several census at the same time in a parallel fashion.

The base data structure was changed in a way that each work unit would have its own copy of each non shared variables. We coupled those variables in a structure and chose to use an array of those structures rather than single arrays for each variable due to cache efficiency problems. When a variable was requested, instead of loading into memory every work unit's variable of the same type, we would be loading all of the non shared variables related to that thread. This arrangement of the variables only proved to be the right choice after also making sure that the structure combining all of those variables would take exactly a cache line (in a hypothetical case, any number of fully filled cache lines would work) because if that was not the case, we could have a thread's respective structure being divided among different cache lines and that would result in constant cache misses. This was achieved by adding a padding dummy variable to fill the remaining of the structure.

This padding adjustment decreased the number of cache misses and was proven instrumental in order to achieve linear scaling on the parallel G-Trie.

We also added the possibility of searching a custom set of subgraphs instead of the whole set. This was done by adding a boolean label to the G-Trie nodes that would tell us if that node was enabled or disabled. This feature would propagate the disabled nodes to ancestor nodes (if all of the edges of particular node lead to a disabled node, that node could be disabled as well) and the labeling was done in a DFS fashion as can be seen in picture 2.3.

### 3.2.3 Master-Slave Architecture

A master/slave architecture is a programming model where one or more components, the masters, have control over the remaining components, the slaves. The master, in this strategy, is often the one who decides what jobs each slave is doing and is the entity responsible for scheduling and dividing work among the slaves as well as aggregating their respective results. The slaves are mostly responsible for requesting a task, doing its associated computation and delivering the results back to the master.

Usually we can separate some different stages in this strategies, the work division phase, the computation phase and the result aggregation stage. The work division phase can be either static or dynamic. In static approaches, the work division can be made at the beginning, with the associated disadvantage that a static partition may provide unbalanced work among slaves, potentially forcing the slaves to be idle after their respective share of the work is completed, wasting computational resources. Dynamic approaches tend to have higher overheads due to more frequent communication across work units but will likely provide a more balanced work division. Dynamic work divisions can be made either by the master, who will create new tasks as soon as slaves computations are ending, or by the slaves who can steal part of the work from other workers when they finish their initial share.

The Master-Slave paradigm was the one chosen to make use the of multiple cores available. This strategy was used since we found it useful to have a central entity managing all of the jobs with the possibility to interrupt some and change parameters of the future ones with no need of synchronization among the remaining working cores. This strategy was chosen over others specially due to the adaptive component of this work. Since we start with little information, it is useful to have a central master trying to queue different magnitude tasks and adjust them to the desired time frame.

### 3.2.4 Adapting Master-Slave to the subgraph counting problem

The main idea of this architecture is to have a queue of jobs generated by the master and a queue of results to hold the calculations performed by the slaves. The job queue will be updated as results of previous tasks come to the master in an adaptive fashion. For instance if we notice that a job with a set of parameters took too much time, the master can simply stop all of the workers who are doing similar things and they would poll the queue for new tasks to perform. In this method, the slaves simply grab tasks from the queue, compute the census associated with the task and send the results to a results queue. The master will be keeping track of both completed tasks and queued ones and will manage the slaves work while processing completed computations and crafting better tasks for future stages.

### 3.2.4.1 Work units and Job Queue

We define a work unit as the essential information that a certain worker should have in order to complete a particular task crafted by the master. A work unit, in this architecture, is defined by a scheme, a fraction and the labels regarding the subgraphs that we want to count. The fraction in this work unit is the part of the graph that we want to sample. There are different ways in which we can divide the same fraction across the `G-Trie` levels and the available schemes are the ones proposed by Ribeiro and Silva [26] and can be divided in three when fixing the subgraph size  $k$  and the fraction  $f$ .

- low:  $P_0 = {}^k\sqrt{f}, \dots, P_{k-3} = {}^k\sqrt{f}, P_{k-2} = {}^k\sqrt{f}, P_{k-1} = 1$
- medium:  $P_0 = 1, \dots, P_{k-4} = 1, P_{k-3} = \sqrt{f}, P_{k-2} = \sqrt{f}, P_{k-1} = 1$
- high:  $P_0 = 1, \dots, P_{k-3} = 1, P_{k-2} = f, P_{k-1} = 1$

The initial motivation for having these different ways to divide the probabilities across different levels was to have more flexibility regarding the distribution of that particular fraction among the `G-Tries` nodes. For a fixed fraction, we could chose the low scheme if we wanted faster and less accurate results, high scheme if we had enough time and wanted a count as precise as possible or medium as an in between option. At later stages of development, the option to switch schemes was particularly important since we found it impossible to reach some desired time frames with the high scheme since it was bounded by the exact computations associated with the previous size problem ( $k - 1$ ). This happens due to the fact that fraction of each level but the last is 1. In those cases we can switch the scheme to low and craft tasks with the desired time frame while running the same adaptive ideas.

The job queue holds this sort of structures and it is filled by the master who tries to keep it with  $2 \times n_{cores}$  at every computation stage. The same queue is polled by the slaves at any stage in which they aren't doing any computation. More details on the this queue implementations can be read at the section 3.2.6.1.

### 3.2.4.2 Aggregating partial results

After doing a specific task, each worker will communicate the results found to the master. We keep track of the sampled count of each subgraph enabled, the time elapsed in the respective calculation along with the fraction and scheme using during the census. There is a result queue to which the slaves send this gathered information and the master will periodically poll it in order to process its results and take actions for future tasks.

### 3.2.4.3 Acting according to ongoing computation state

Even though both previous mentioned queues would be enough for this strategy to work, we would not be able to check neither the running jobs for each slaves neither would we be able to check if a particular task was taking too much time. In order to address this problem, third structure was used in this strategy, a vector of running jobs with a entry for each slave. The purpose of this structure was to make it possible to the master to scan what every slave was doing giving it the ability to interrupt some of the tasks. This third queue would hold, for each running task, the id of the thread that was processing it, the time associated with the start of the task along with the respective work unit with the information described above at section 3.2.4.1.

### 3.2.5 Finishing criteria

With the purpose of allowing for multiple goals, we set a few ways to stop the whole sampled subgraph count process. We can provide running time and/or a convergence metric and convergence value. If the master notices that it has been running for longer than the time set by the user, it will stop the slaves and output the results. The same will happen if the results converge to a given value given a certain metric. If we finish a job with probability 1 over all of the labels, the program will also end since we just did a complete census and the results will be the ones got at that final iteration.

### 3.2.6 Master-Slave work flow

Our whole process is always controlled by the master and it is the master that is called at the beginning of this strategy. Algorithm 3 describes this component's work flow.

After being initially called, the master starts by doing some setup operations (line 1). This setup includes launching the slaves threads, filling the job queue with some initial tasks, loading the network and the `G-Trie` among other beginning tasks. It will then start its main body consisting of generating tasks and aggregating upcoming finished computations (lines 2-9) until the given time is over. The slaves are checked periodically to see if they are doing meaningless computation such as tasks running for longer the desired ideal frame and tasks whose characteristics are already proven to result in higher executions than desired (line 3). If the job queue is getting empty, the master fills it with new tasks based on the previous results attained (line 4). The master will check for new results and process them if so. This result process stage will partially aggregate the latest result and update some variables which are instrumental in determining future tasks (lines 5-7). At the end of the available time, the master does some cleanup operations such as shutting down the slaves and then it can simply output our results as the guess for the subgraph count problem since the master is updating results as they come by (lines 10-11).

---

**Algorithm 3** Master Workflow

---

```

1: setup()
2: while not finished() do
3:   checkSlaves()
4:   addTasks()
5:   if newResults() then
6:     res ← resQueue.pop()
7:     processResults(res)
8:   end if
9: end while
10: cleanup()
11: outputResults()

```

---

The slaves workflow is simpler than the masters but very important nonetheless. It can be described at algorithm 4.

The slaves are always on a loop computing whatever results required (lines 1-7). If they have work to do they start by getting a work unit from its queue, run the associated task and output the results to the result queue (lines 2-5).

---

**Algorithm 4** Slave Workflow

---

```

1: while true do
2:   if workToProcess() then
3:     workUnit ← workQueue.pop()
4:     res ← runCensus(scheme, fraction, labels)
5:     resQueue.push(res)
6:   end if
7: end while

```

---

### 3.2.6.1 Parallel synchronization, cancel methods and implementation details

One of the goals of using this architecture was to remove part of the inherent synchronization that would need to happen if there was no master managing all of the slaves.

There are, however, a few sections that are protected with locks. Namely the working and the results queue. The master must be sure that no slave is popping the tasks queue when adding new tasks and when the master is getting results from the respective queue, no other unit can be accessing it. We decided to implement our version of a parallel queue, coupling each queue with its respective lock mechanism, so that it would have to be acquired in order to do the regular queue operations such as pushing, popping and emptiness test.

One important feature of this working structure is the ability to remove/cancel jobs at any time required. For this to happen, we used two cancel types on the slaves: the asynchronous and

the deferred types. When at the deferred state, a thread can only be canceled at a cancellation point. This is useful if we have a series of operations that need to be considered atomic. The asynchronous type allows a thread to be canceled at will.

We chose to set cancellation points both at the beginning and the end of each iteration and chose to set the census as asynchronous so that it can be interrupted at any time. If, for instance, a cancel request is made right after a finished census, the cancellation type is already deferred and the slave will only cancel after sending its respective results. It was this deferred option that made sure that a thread wouldn't be cancelled, for instance, after acquiring a particular lock, which could cause a deadlock after being cancelled.

The slaves are periodically checked by the master and can be set to be canceled if they are taking too much time (this threshold is set to be  $2 \times \text{sweetSpotFinish}$  by default) or if they are doing a task that is known to run outside the `sweet spot`. This final situation can happen if the master notices that a certain task is taking too much time and all of the tasks with a fraction above that one are to be cancelled. The slaves are also canceled if a finish state is reached.

### 3.3 Adaptive Methodology

The adaptive fashion in which we reach our ideal tasks is at the core of this work. In one hand, we want to reach that sweet spot quickly but on the other, we don't want to start many tasks that will have to be canceled due to their excessive duration. Assuming that we have no idea of the time that a certain sample with a given fraction will take to compute, we have to be conservative with our initial probability so that we have a baseline which can be increased afterwards. We don't want to discard valuable computational time just because a task takes a bit more time than our ideal task duration. We added a threshold over the desired time limit so that we can both use those results and we can use that elapsed time as an upper bound for our computations.

#### 3.3.1 Sweet Spot

The idea of a `sweet spot` is a time frame in which a single task should compute for the best approximation of the subgraph count problem. Ideally, when a `sweet spot` is set, the goal is to run the corresponding task  $\text{totalTime}/\text{sweetTime}$  times during the available time period. Even though this concept is a parameter of our program, we present a few methods of logically defining a `sweet spot`.

One may think that the ideal task should be associated with a low time frame task so that it can be repeated over and over until the time given by the user for the subgraph count problem is over. This approach has the apparent advantage of being able to fit several tasks in the given elapsed time with the motivation that more tasks should provide a better guess for this problem. The problem is that, despite the high number of runs, we are always running them with a small probability and that factor can have a negative impact in our final approximation.

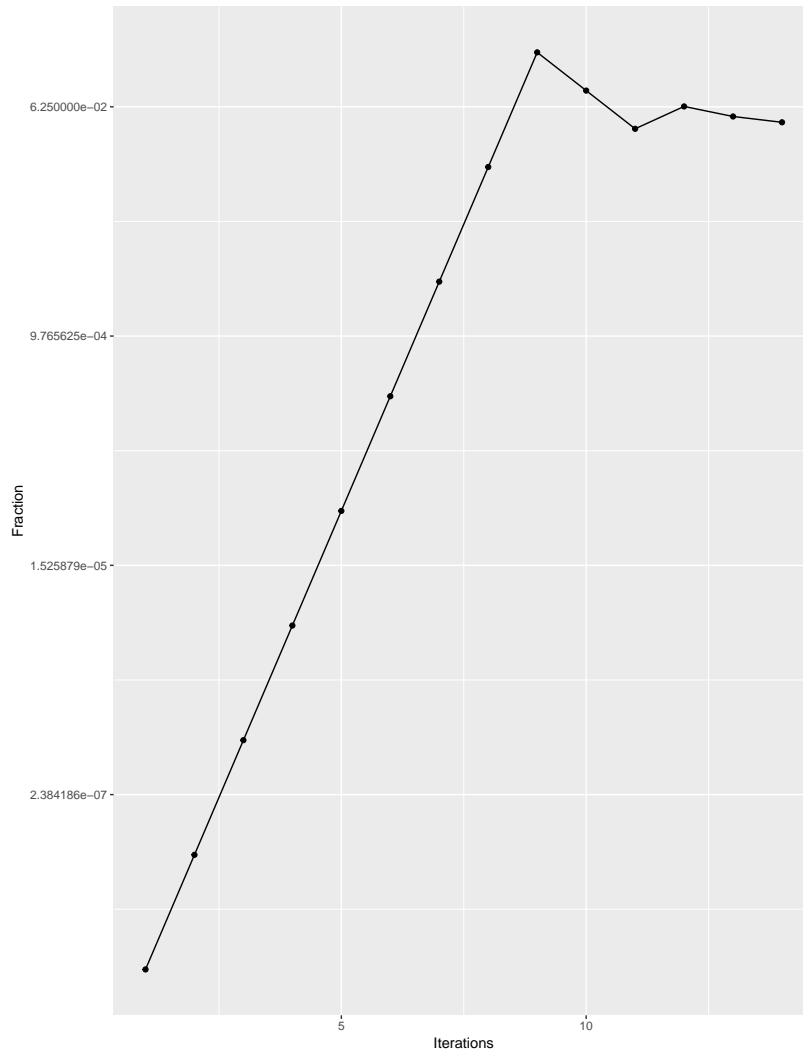


Figure 3.1: Plot showcasing a typical fashion in which a sweet spot is reached. We can observe that the fraction increases at the same multiplicative rate until its associated task takes more time than expected and from that point it lowers the multiplicative factor and continues this method until the sweet spot is reached.

We can think that the ideal sweet spot should be related to a task which maxes its elapsed time. The motivation to this option is that running a census with a higher probability should give us a better estimate but its drawbacks are the lower sample size along with the tighter time restrictions since we can't be sure of the elapsed time of a task before hand, making it hard to guess the associated probability and thus making it possible that our "ideal task" will not even end. An in between strategy can also be considered, in which we want an average timed task running an average number of times so that we can get the advantages of both approaches with less impactful disadvantages.

### 3.3.2 Reaching our ideal task

Regardless of the strategy used among the ones referred above, we need to have an algorithm that is able to craft a task that will fit in the given `sweet spot` restrictions. We present algorithm 5 that proves to reach sweet spots in a rapid fashion. This algorithm has two main parameters: The initial probability and the initial multiplicative factor (lines 1 - 2). The initial probability should be conservative so that we have a lower bound task as soon as possible. Afterwards, we want to tweak the tasks characteristics so that we reach our desired job (lines 4 - 13). If we notice that a task's elapsed time is below our time interval, we can simply create a new task with a new increased probability and reach that probability by multiplying the previous one by the respective multiplicative factor (lines 5 -6). It is also possible that after multiplying a completed task's fraction by the associated multiplicative factor, we face a new tasks whose time frame is above our `sweet spot`. In that case we should proceed by returning back to the previous task's fraction, lower the multiplicative factor, get a new fraction and continue searching for the ideal task (lines 7 - 10). If we find that a certain task fits the given time frame, that task will be the ideal one and the `sweet spot` finding phase can be considered over (12 -14).

---

#### Algorithm 5 High-level sweet spot reaching algorithm

---

```

1:  $p \leftarrow initialProb$ 
2:  $f \leftarrow initialMultFactor$ 
3:  $sweetReached \leftarrow \mathbf{false}$ 
4: while not  $sweetReached$  do
5:   if  $taskTime < sweetSpot$  then
6:      $p \leftarrow p \times f$ 
7:   else if  $taskTime > sweetSpot$  then
8:      $p \leftarrow \frac{p}{f}$ 
9:      $f \leftarrow \frac{f}{2}$ 
10:     $p \leftarrow p \times f$ 
11:  else
12:     $sweetReached \leftarrow \mathbf{true}$ 
13:  end if
14: end while

```

---

## 3.4 Approximating the counting

Due to our method's property of receiving several results for potential different tasks, we decided that a simple average of each count ,with its respective extrapolation ,would not suffice.

A weighted mean was the metric used since it can emphasize experiences with higher sampled fractions, which should lead to more precise counts. With a set of  $n$  extrapolate results  $res$  and associated sample fractions  $frac$ , if we were to fix a subgraph, its estimate is given by the formulae below.



$$wAvg = \frac{\sum_{i=0}^n res_i \times frac_i}{\sum_{i=0}^n frac_i}$$

We are interested on following the distribution of the results over the time so that we can potentially stop computations when we notice that recent results are not changing our global estimate by much. Calculating the result dispersion can be achieved by using variance and its standardized version, coefficient of variance. Since we always want to give more weight to tasks made with higher sampling portions, we will present weighted versions of these measures.

$$wVar = \frac{\sum_{i=0}^n frac_i \times (res_i - wAvg)^2}{\sum_{i=0}^n frac_i}$$

$$wCoefVar = \frac{wStdDev}{wAvg} = \frac{\sqrt{wVar}}{wAvg}$$

Since our method is able to run tasks with execution times as little as seconds for several hours, it can be dangerous to store all the results at all times and the calculation of this metrics would take more time as more results got processed. In order to solve this problem, we adapted these formulae so that only the last run, along with a few fixed numbered variables were needed. Those new variables that are updated as a new result comes by are the number of tasks completed  $n$ , the sum of the fractions used  $\sum_{i=0}^n frac_i$ , the sum of squared weighted results  $\sum_{i=0}^n frac_i \times res_i^2$  and the sum of the weighted results  $\sum_{i=0}^n (frac_i \times res_i)$ .

The new weighted average can be simply computed with the following.

$$wAvg_{new} = \frac{(wAvg_{old} \times \sum_{n=0}^{n-1} frac_i) + res_n \times frac_n}{\sum_{n=0}^n frac_i}$$

After updating the remaining variables, we can get the new weighted variance along with weighted coefficient of variance.

$$wVar = \frac{\sum_{i=0}^n frac_i \times (res_i - wAvg)^2}{\sum_{i=0}^n frac_i} =$$

$$\frac{\sum frac_i \times (res_i^2 - 2 \times res_i \times wAvg + wAvg^2)}{\sum_{n=0}^n frac_i} =$$

$$\frac{\sum frac_i \times res_i^2 + 2 \times \sum frac_i * wAvg + \sum wAvg^2}{\sum_{n=0}^n frac_i}$$

The coefficient of variance can be computed as before.

### 3.4.1 0 probability run and new ideas

The high scheme mentioned earlier at section 3.2.4.1 is the starting scheme used in our runs. There is a chance that , for a set of network and subgraph size  $k$ , the defined sweet spot couldn't be achieved despite the fraction sampled. This weird phenomena is explained by the definition of the high scheme. The probabilities in this idea are scattered among the G-Trie nodes so that only the  $k - 1$  level represents that fraction. This means that the count for that  $k$  is bounded by the exact count of  $k - 1$  in this scheme.

This realization made us adapt our algorithm so that it would always start of by computing the lower bounded 0 probability in the high scheme (with the only goal of seeking a lower bound for the default scheme) run and if the respective elapsed time was over the defined by the user, we could switch the method to low, which can be easily adapted to lower time frame, and start the algorithm from there. Since this run with 0 probability provided a lower bound for the scheme, its respective execution time can be seen as a good time frame to run a task if we want it to be repeated many times. We propose a choice of the sweet spot of  $2 \times t_{zeroProb}$  for these situations.

### 3.4.2 Automating parameters

Both defining and finding the sweet spot are among the core of this work. If we knew exactly the perfect task to fit a given sweet spot, then we would not have the need of searching for that sweet spot and could start to replicate those task's characteristics between the available cores. Even if this is not the case, we could luckily end up finding the sweet spot after only a few attempts with the right initial parameters or we could have a set of parameters that would make it so that adding more cores would not accelerate the process of finding that time frame. For instance if we were starting our algorithm with a starting probability of  $1E - 5$  and the starting multiplicative factor of 2, the tasks generated would be  $1E - 5, 2E - 5, 4E - 5, 8E - 5, 1.6E - 4, 3.2E - 4, \dots$  for a sufficiently large sweet spot. If we were to discover that the ideal task would be achieved with the fraction  $2.56E - 3$ , that would be 8 tasks after the initial one and if we were to split those across the available cores, after having more than those 8 slaves, the time to reach the sweet spot would be the same since, despite the number of available workers, there would always be one that would start that particular task (this only happens if the infrastructure is scalable). With this problems in mind, we believe that the best way is to take the number of cores into account when choosing the initial factors. If we fix an initial probability we can, for instance, chose a multiplicative factor so that, at the beginning, we would have one worker computing the census associated with the initial probability  $ip$  and other computing the exact task (probability 1). We could accomplish this by choosing the multiplicative factor  $mf$  depending on the available cores  $nc$  so that.

$$ip \times mf^{nc} = 1 \implies mf = \sqrt[nc]{\frac{1}{ip}}$$

---

This strategy can be adapted to cover any desired initial interval (by replacing 1 with the desired higher bound).



## Chapter 4

# Experimental Results

In order to showcase the results with, as many processors as possible and with as little noise as possible, we used a parallel machine with 64 logical cores provided by DCC. This Machine has 4 CPU sockets, each with 8 physical cores with each core being able to run 2 threads. This adds up to 64 processing units along the four AMD Opteron 6376 processors.

The code was an extension to the existing `G-Trie` base code and was thus developed using C++11 using `gcc 4.8.3` as the compiler. Due to portability options, we chose to use POSIX Threads as our parallel model.

Since there are many degrees of freedom among the variables of this algorithm, we want to showcase the choices made in some variables by isolating the remaining ones and then combining the choices into more advanced versions. Plots for those experiences were attained using `R` and `Rstudio`.

### 4.1 Network Data

Name	Vertex	Edges	Representation
Dolphins	62	159	Adjacency Matrix
Virgili	1133	5451	Adjacency Matrix
Power	4941	6594	Adjacency Matrix
Brightkite	58228	214078	Hybrid
Flickr	105938	2316948	Hybrid
Pennsylvania	1088092	1541898	Hybrid

Table 4.1: Different networks used

In order to test the quality of our work for different types of networks, we chose a few different ones with increasing dimensions and some different edge density. We also chose networks related different fields such as social networks, road networks and animal networks. The networks presented are all undirected, unweighted and uncolored for legibility and result comparison

reasons but our strategy is ready to be used in other networks with the above features. All of the following networks can be found in the Koblenz Network Collection [11] and The Snap Library [12].

`Dolphins` [16] is the smallest network presented and was used mostly in the early development stages due to its small size, making it easy to reach high order motifs even in the exact version of the algorithm. It is an animal network that describes frequent associations between bottlenose dolphins and has 62 vertex with 159 edges.

`Power` [32] is a network that represents information about the Western United States of America power grid. Edges in this network refer to power supply lines between the vertex, that can be either generators, transformers or a substation. This network has 4941 vertices coupled with 6594 edges.

`Virgili` [8] showcases e-mails sent between members of the University Rovira i Virgili (Tarragona, Spain). The nodes and edges in this network refer to students and e-mails sent respectively. The Virgili network gathers information about 1133 users over 5451 communications. An interesting observation about this network is that, despite having a lower vertex and edge count than the `Power` network, it takes longer to perform computations on. For instance, if we count all of the 6-sized subgraphs in this network, we take more than a minute, compared to less than a second for the same task performed in the `Power` network.

`Brightkite` [3] is a social network that describes relations between users from different locations. A connection between two nodes lets us know that they are friends with each other. This network has 58228 users and 214078 edges representing friendship. This amount of users/vertex would lead to a adjacency matrix with over  $2 * 10^9$  entries, that is approximately  $2GB$  which is a huge lower bound for our program since it only covers a portion of the memory, that high lower bound is particularly problematic since the graph is sparse and only has an average of about 7 connections per vertex compared to the  $58228^2$  potential connections saved by an adjacency matrix. The representations of both this network and others with higher vertex count is now the hybrid scheme proposed by [22] which allows to have an efficient memory representation while only being less than 50% slower.

`Flickr` [17] is an image network created by linking images with similar metadata attributes. The images are vertex in this network and a link between two images exists if they share the same location, tags, etc. This data set has 105938 images along with 2316948 links between them. The representation used for this network is the hybrid scheme rather than the adjacency matrix option.

`Pennsylvania` [13] is a road network of Pennsylvania, United States of America. Nodes in this network refer to either endpoints or intersections while the edges describe roads between those entities. This is the biggest network considered and has 1088092 vertex coupled with 1541898 road connections. Note that even though this network has the most vertex, it has less edges than the `Flickr` network.

## 4.2 Scalability of parallel methodology

Regardless of the accuracy or any metrics provided by this algorithm, we first need to make sure that the parallel tasks running on the new G-Trie (section 3.2.2) can scale to a larger amount of workers allowing for multiple tasks to run simultaneously with no significant performance loss.

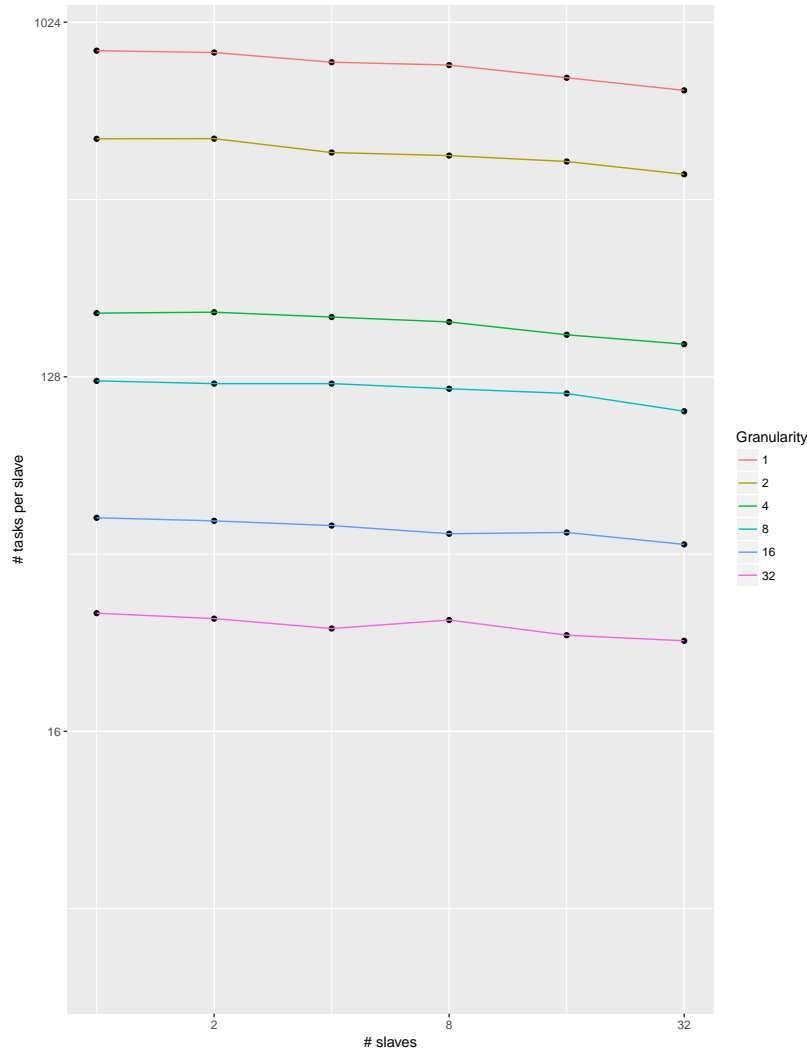


Figure 4.1: Tasks completed by slaves across different singular task execution times. The results are taken from computing the census in the Brightkite network for the size 4 subgraphs with 15 minutes to run.

Figure 4.1 shows that the ratio of tasks done per slave close to constant as more cores are added to computations. This is showcased over tasks with different granularity and have the inherent overhead included in the results observed. Note that the number of slaves used in this work is not equal to the total number of cores since we are using a master dedicated to control and distribute work units. This results that, despite the sweet spot used, we can use more cores to obtain more runs in the end.

To the data provided by the picture 4.1 it is added the table 4.2 where we can see details

Network	K-Size	Fraction	Baseline	#slaves: #tasks				
				2	4	8	16	32
Brightkite	4	0.004	867	1.978	3.738	7.353	13.651	25.354
Brightkite	4	0.007	517	2.002	3.691	7.25	14.008	25.988
Brightkite	4	0.02	186	2.011	3.909	7.597	14.091	26.672
Brightkite	4	0.03	125	1.968	3.936	7.64	14.864	26.784
Brightkite	4	0.07	56	1.964	3.821	7.286	14.679	27.375
Brightkite	4	0.12	32	1.938	3.656	7.688	14.063	27.219
Flickr	3	0.004	1071	1.923	3.128	5.625	10.963	19.894
Flickr	3	0.012	391	1.726	3.061	5.757	11.156	19.77
Flickr	3	0.02	242	1.876	2.971	6.041	11.306	19.814
Flickr	3	0.04	128	1.609	3.516	6.023	10.781	19.617
Flickr	3	0.09	60	1.867	2.933	5.6	11.083	19.7
Flickr	3	0.19	29	1.862	3.207	5.862	11.414	20.448
Pennsylvania	7	0.0004	1229	1.956	3.57	6.782	13.193	25.101
Pennsylvania	7	0.005	345	1.988	3.736	7.133	13.881	27.203
Pennsylvania	7	0.01	226	1.85	3.708	7.367	14.19	27.624
Pennsylvania	7	0.02	144	1.965	3.889	7.368	14.257	28.139
Pennsylvania	7	0.03	110	1.991	3.8	7.327	14.1	27.955
Pennsylvania	7	0.08	56	2	3.732	7.339	14.196	27.982

Table 4.2: Comparison of the number of tasks computed with a different number of cores among different networks

on the runs that lead to the plot along with more results over different networks with different subgraph sizes. Here we can observe the speedups relative to the number of tasks attained with only one slave which he named baseline. We can see that for a wide range of values, doubling the number of slaves results in next to double the speedup with some diminishing returns as we get closer to the machine limits. Part of these diminishing returns can be justified by the overhead of the master slave architecture who has to both initialize, finish and monitor more processing units.

Both the results presented at figure 4.1 and table 4.2 were attained by running our algorithm with an initial probability which would lead to a tasks duration according to the wanted granularity (2, 4, 8, 16, 32) along with an initial multiplicative factor of 1 so that we would compute the same sized tasks over and over. The probability that would lead to a task fit to a particular time frame was retrieved by running the same algorithm with the `sweet spot` set to the desired time frame.

### 4.3 Reaching the Sweet Spot

Reaching the `sweet spot` is an important task regarding our work and we want it to be as fast as possible so that we save time for more repetitions of the ideal task relative to that time frame. Figure 4.2 presents a particular case of this stages showing how the time elapsed by the



successive performed tasks is updated until we overshoot the desired result and from that point we can do a process similar to the binary search method in order to divide our lower and upper bounds, tightening them until the ideal task is conceived. The initial similar and unpredictable times are caused by conservative initial tasks that can have very low precision and execution times.

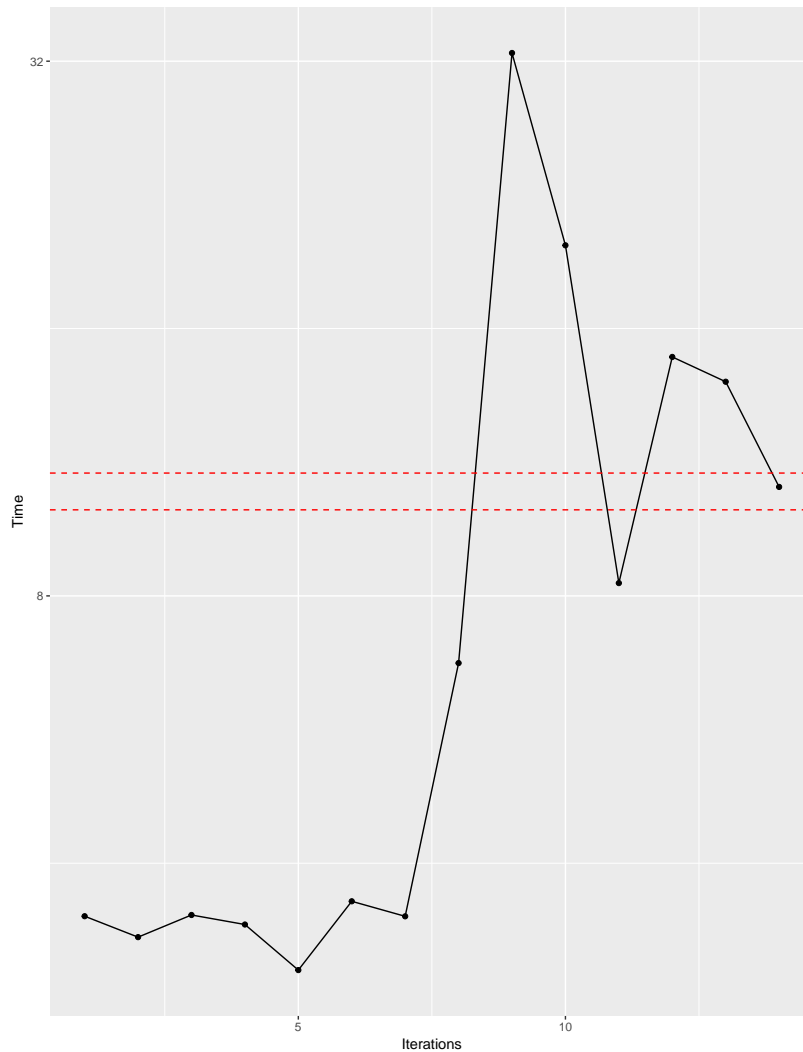


Figure 4.2: A plot showing the time variance of the tasks until the sweet spot is reached. The first iterations are not conclusive since their associated probability is really low, but we can see the way in which the fraction changes influence the time taken to compute the respective tasks. The dotted lines represent the wanted sweet spot.

We want to motivate the choice of some parameters by showcasing its ability to fasten the sweet spot reaching steps.

To further motivate the use of parallelism in this strategy, we showcase at table 4.3 that for different sweet spots and other variables, having more work units leads to a faster process of finding the desired task. Here we have to take into account that the minimum time possible is bound by the sweet spot lower interval and it can be reached if the initial distribution of

probabilities across slaves makes it so that one of those initial tasks is completed in the wanted time interval.

From this point on, we will assume that our parallel strategy is useful in both stages of finding the `sweet spot` and ideal task repetition. The remaining of the information will be delivered by experiences made with only 1 slave.

ip	im	ss	#cores: elapsed time				
			2	4	8	16	32
0.0001	2	[6-8]	44.0497	16.1577	12.2288	6.43386	6.01137
0.00001	1.1	[10-14]	459.438	160.655	72.9032	37.7238	21.0014

Table 4.3: Comparison of the time required to reach the `sweet spot` computed with a different number of cores among different parameters for the `Brightkite` network and subgraph 4 count

We mentioned the virtue of setting conservative initial probabilities in our algorithm. To show what little time loss could be achieved by under estimating the ideal task by many magnitude orders, we tested our method for a wide range of initial probabilities and initial multiplicative factors. Our initial guess was that having low, conservative initial probabilities would not be bad when coupled with high multiplicative factors since we could rapidly discard jobs whose runtime was much lower than the desired one.

Table 4.4 collected the time to reach the `sweet spot` [10 – 15] relative to a task attained by running sampling with fractions close to 0.0625. We can see that generally, if we are close to the ideal probability, the time to reach that is lower, as expected. If we move far away from that fraction, we can verify that the time doesn't change drastically and can even be similar for most cases. This algorithm was idealized so that the probability would be increasing until the ideal one and over estimating in the beginning could lead to very slow runtimes in this stage since we could be performing several tasks which would have to be canceled due to its elapsed time being above the wanted one. Parallelism would make this conservative choices even better since more slaves would be able to reach from the conservative fraction to the desired one faster while choosing an over estimate first probability would lead to multiple slaves having to abort ongoing computations.

We did the same experience as presented in table 4.5 with a lower `sweet spot` of [2 – 4] and found that the results were similar. The associated ideal probability was near 0.0125 and again, tasks with initial probabilities close to that reached the time frame faster but not if that probability was above the ideal one. We can for instance see that the lowest probability which was tested was many orders of magnitude lower than the ideal fraction still manage to out perform the highest which was only 10 times above that, showcasing the advantages of being conservative.

In order to exaggerate how bad a wrong initial fraction choice could be, we jumped to a more time consuming task by increasing the size of the subgraphs to be count. In order to fill the same `sweet spot` of [2 – 4], the associated probability was 0.00001. The results were gathered

in table 4.6 and here, even the given 15 minutes were not enough for some first probabilities above the ideal one. The lowest fractions tested still managed to be competitive with the ones closer to the ideal one.

In the same tables mentioned before, we also tweak the initial multiplicative factors. Even though the presented ones can be better depending on the initial probabilities chosen, we can see how bad the lower versions can perform if the initial fraction is far from the ideal one. This table strengthens our initial believes that the first tasks should start of low on probabilities but somehow high on the multiplicative factor. We didn't expect such good results for the seemingly high initial multiplicative factor of 8.

ip im	1.1	2	4	8
0.1	55.0008	58.3749	39.4208	<b>35.6055</b>
0.01	112.18	51.6731	73.1356	<b>43.8778</b>
0.001	169.493	36.8594	51.4888	<b>18.8083</b>
0.0001	198.198	40.1411	54.2231	21.8838
0.00001	233.7	67.7562	<b>23.5501</b>	134.216
0.000001	266.645	51.7172	<b>32.2805</b>	102.609
0.0000001	304.518	<b>52.3028</b>	64.2436	93.0482
0.00000001	337.474	<b>82.5525</b>	91.4379	87.2837
0.000000001	371.167	88.4405	<b>43.3636</b>	60.7231
0.0000000001	404.011	64.5019	67.8906	<b>53.0995</b>

Table 4.4: Time elapsed in order to reach the sweet spot of  $[10 - 15]$  seconds in the most time consuming network `flickr` with size 3 across different parameters

ip im	1.1	2	4	8
0.1	timeout	35.5633	<b>24.4945</b>	57.7288
0.01	5.95253	<b>2.25315</b>	5.92643	5.9274
0.001	26.8345	<b>5.80789</b>	6.21456	7.02522
0.0001	2.19312	2.19836	<b>2.00911</b>	7.98265
0.00001	92.9445	17.736	<b>2.00869</b>	2.19205
0.000001	123.818	<b>22.11554</b>	13.1221	10.1744
0.0000001	155.892	25.1325	<b>16.2458</b>	21.7627
0.00000001	189.027	31.1317	<b>19.9499</b>	21.1927
0.000000001	219.988	34.616	<b>19.4646</b>	21.3
0.0000000001	246.517	40.53	22.835	<b>21.487</b>

Table 4.5: Time elapsed in order to reach the sweet spot of  $[2 - 4]$  seconds in the most time consuming network `flickr` with size 3 across different parameters

ip im	1.1	2	4	8
0.01	timeout	timeout	timeout	timeout
0.001	timeout	79.8941	<b>48.1256</b>	timeout
0.0001	96.4992	<b>33.4786</b>	49.1907	37.4812
0.00001	32.6154	<b>23.6641</b>	24.8593	38.4217
0.000001	43.9132	27.4307	<b>23.229</b>	35.0774
0.0000001	58.1563	31.3275	<b>24.4292</b>	29.7246
0.00000001	45.0733	<b>24.8558</b>	28.5985	27.5082
0.000000001	46.2733	27.3502	32.5875	<b>24.018</b>
0.0000000001	46.1457	26.3342	<b>23.8007</b>	23.9522

Table 4.6: Time elapsed in order to reach the sweet spot of  $[2 - 4]$  seconds in the most time consuming network `flickr` with size 4 across different parameters

## 4.4 Quality of the approximation

At the end of the day, the most important factor for this algorithm, is to get the best estimate which lessens its associated error. We will consider relative percent error and percent accuracy in this examples.

$$error = \frac{|count_{exact} - count_{approx}|}{count_{exact}}$$

$$accuracy = 1 - error$$

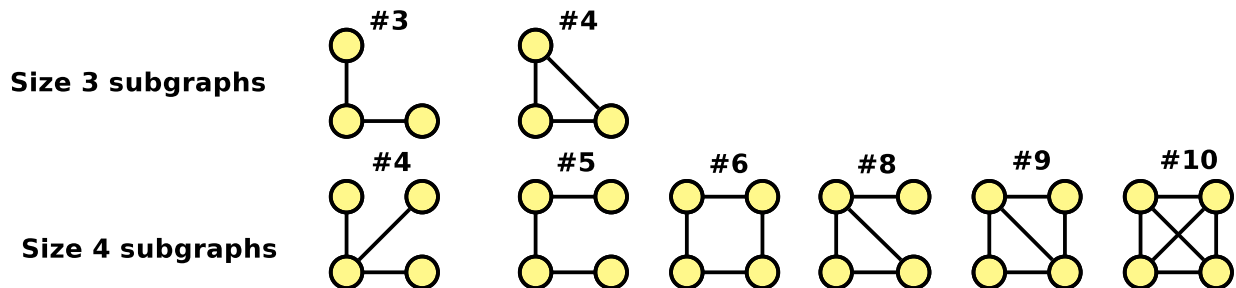


Figure 4.3: Labels of the different  $k$  sized subgraphs. This label is achieved by performing a DFS on the `G-Trie 2.3`

The labels presented in the following examples where multiple subgraph types are available can be consulted in figure 4.3. In cases where we only had a single accuracy value for a single census, the average between the different accuracy across the subgraph types was used.

Figure 4.4 shows us that a single run can be away from the exact count but that won't deviate our global guess using the weighted average. Even in smaller, beginning tasks, we can see that we can get near the exact count in a surprisingly rapid fashion.

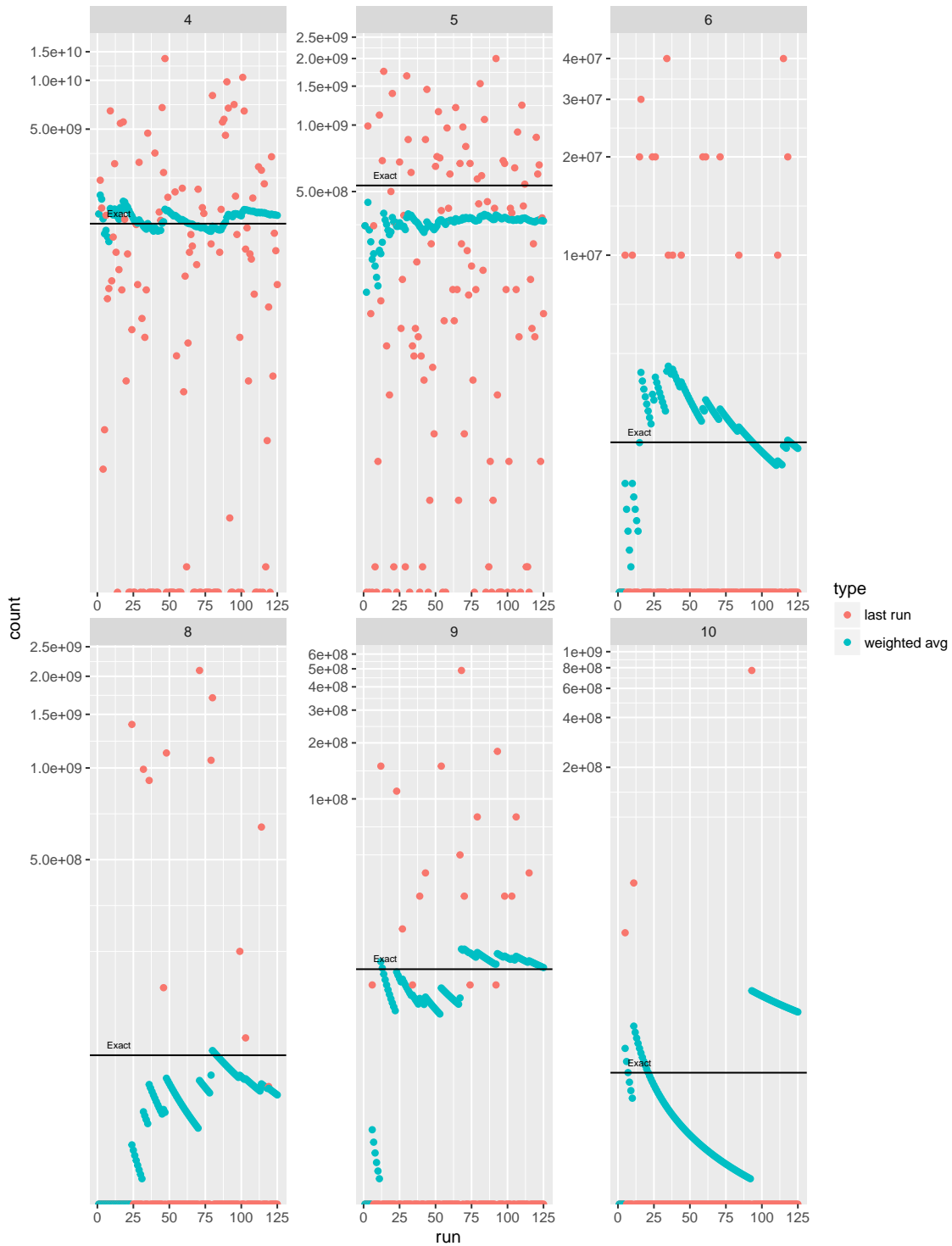


Figure 4.4: Progression of both individual and aggregated results in the Brightkite network sized 4 across different subgraph types. Every task run was done with  $1e^{-7}$  probability.

In figures 4.5, 4.6 and 4.7, we tried to study the effect of the chosen sweet spot on the final guess for the subgraph counting problem. We initially guessed that having a low sweet spot would lead to many tasks being computed, and that ideal task would be found faster in

the initial stage of our algorithm. This was not exactly the case since we found out that in some cases, the additional time to find a higher `sweet spot` and even the reduced number of sampled tasks gave us better results in the end. Figure 4.5 guides us towards thinking that choosing a small time frame for our ideal task is good if the run time is also low.

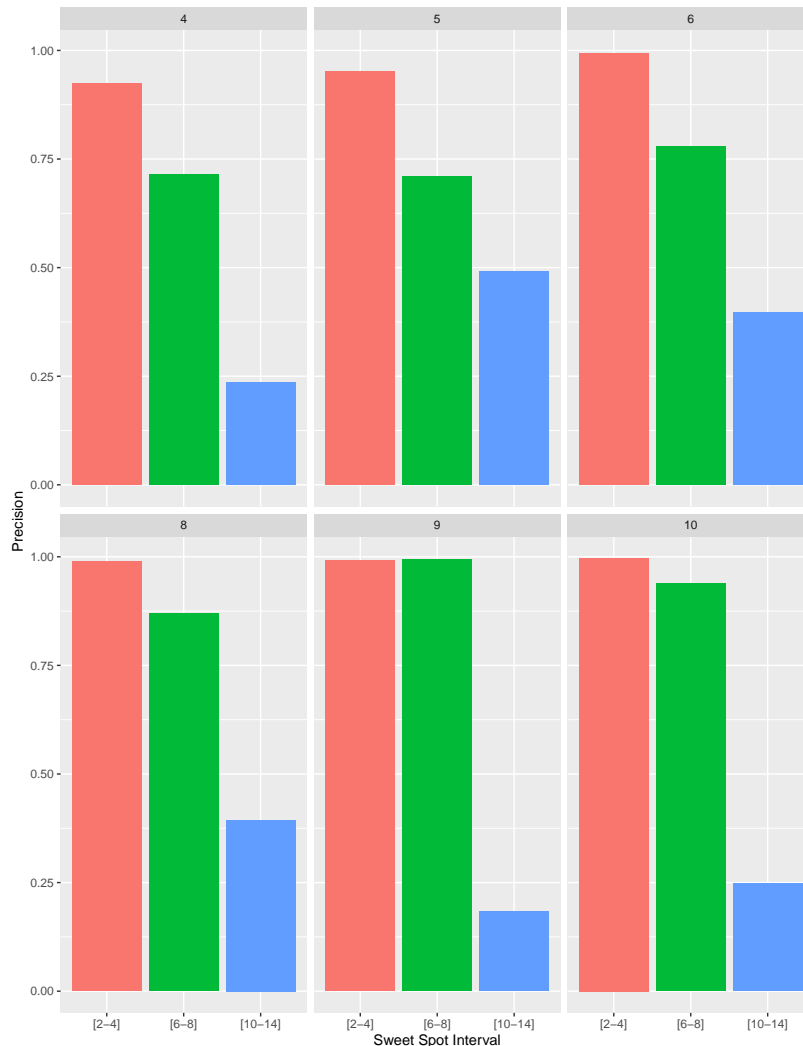


Figure 4.5: Precision across different `sweet spots` subgraph types for the Brightkite network size 4. This results were obtained after 100 seconds of run time

If we increase the available runtime, higher `sweet spots` begin to become preferable since the time spent to find its associated task becomes less significant as we let the algorithm run longer and we are also able to perform that task more often as described at figure 4.6.

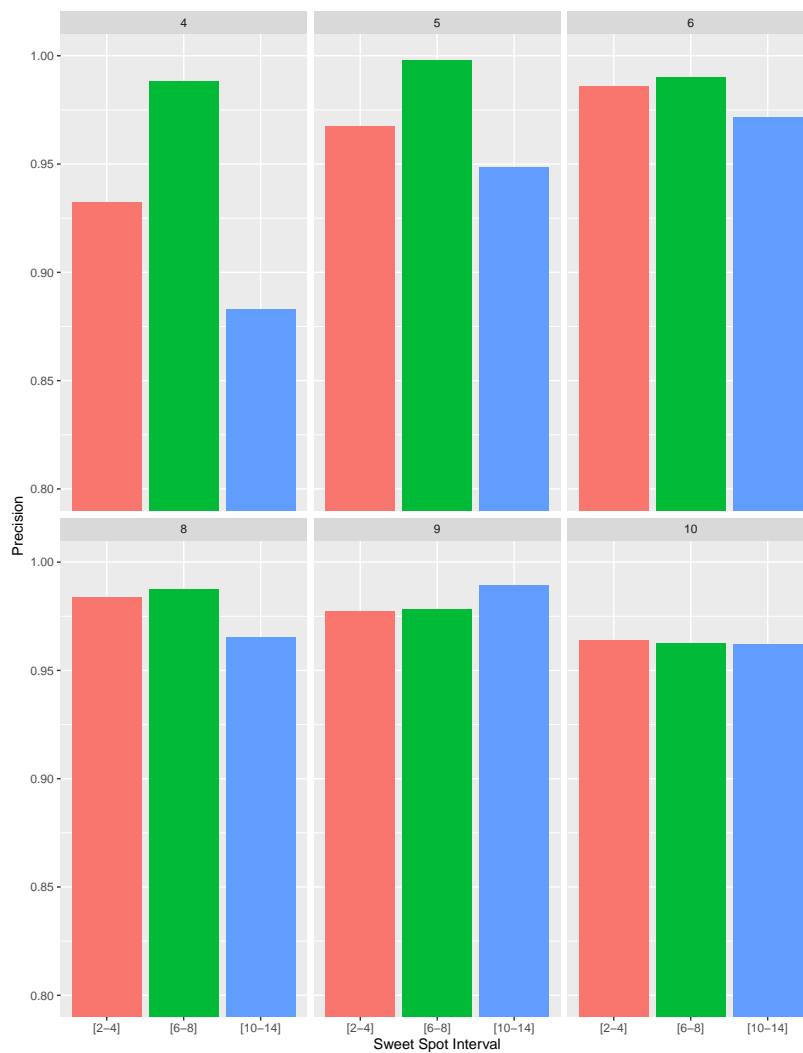


Figure 4.6: Precision across different sweet spots subgraph types for the Brightkite network size 4. This results were obtained after 300 seconds of run time.

If we let our program run for a really long time, we begin to notice that it is even more advantageous to increase both the sweet spot and its associated finding time in return of more accurate runs that, even though would be performed less times, provide better estimates to the counting problem across different subgraph types. This can be observed at figure 4.7.

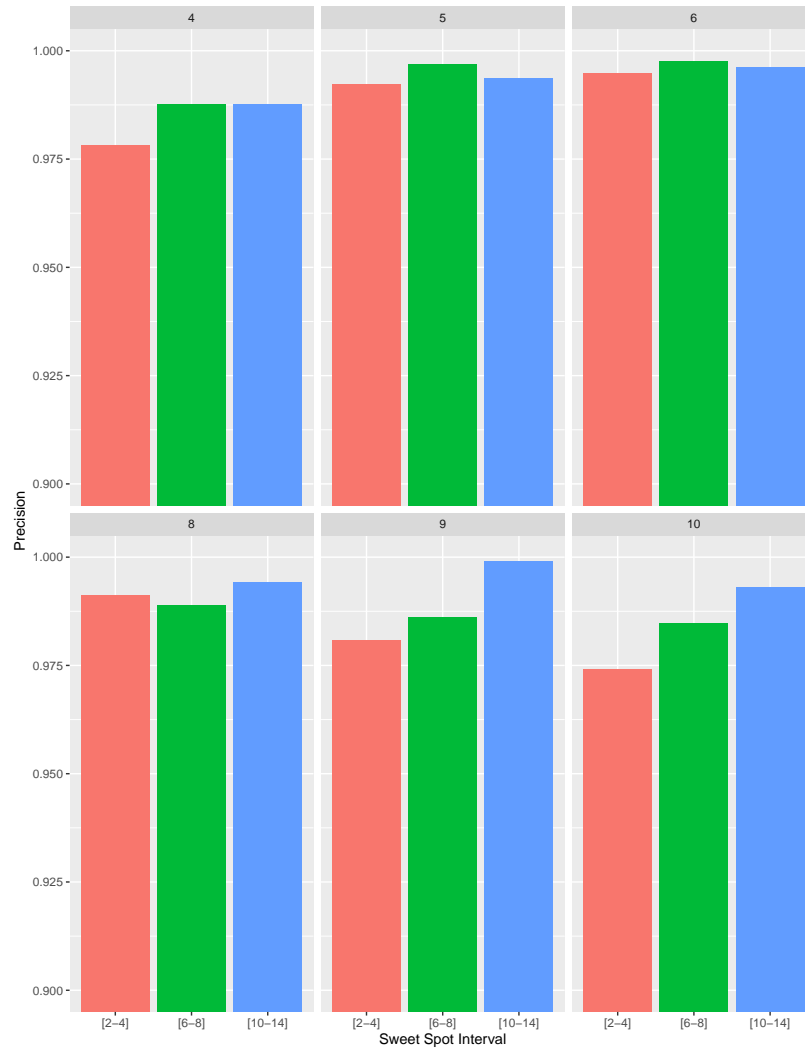


Figure 4.7: Precision across different sweet spots subgraph types for the Brightkite network size 4. This results were obtained after 15 minutes of run time.

In the end we decided to get an exact count along with its sequential running time and take percentages of that time and feed them to our algorithm, testing it with all of the mentioned components all together. The results from figure 4.8 were good especially given the fact that, even though we calculated it beforehand, the program didn't knew the time to get the exact run. The presented results use increasing sweet spot as more time is added, a low initial probability along with a high initial multiplicative factor.



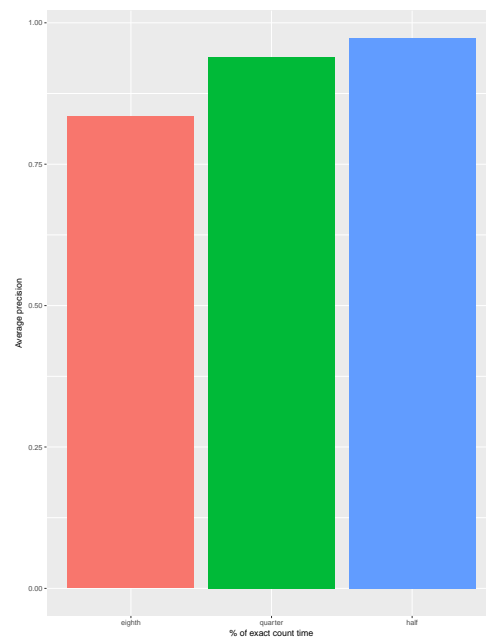


Figure 4.8: Average precision across different fractions of exact time. Brightkite network with 4-size subgraphs



# Chapter 5

## Conclusion

### 5.1 Contributions

This work tackled the subgraph count problem using both parallelism and sampling. Those two components were coupled with a third major one, the adaptive scheme. Every major component added performance to the task and provided good results with the right parameters.

With the ideal task crafted beforehand, our method was separated in two stages: The sweet spot reaching stage and the ideal task repetition stage. Both those stages used sampling and were improved as the available work units increased.

Using this ideas, we were able to achieve very accurate results for the subgraph count with limited time at our disposable and we would like to think that the time conceived to our method is used in an efficient manner.

### 5.2 Future work

Even though this work was satisfying and provided good results for the subgraph counting problem, there are a few options that could still be explored as future work.

- **Removing subgraphs** - The presented revamped `G-Trie` 3.2.2 is equipped with a non explored feature, the ability of removing subgraphs from the `G-Trie`. Along with a proper coefficient of variation, we could, after a few runs, decide that a particular subgraph count was "good enough" since new results were not changing our final approximation of its count. If that were the case we could disable that particular subgraph so that future census would take less time. After removing a subgraph, the `sweet spot` finding phase could start over and the new task would have a higher precision since the computational time is lowered with the removal of nodes. We believe that this would made it possible to find accurate counts faster and focus on harder to find, rarer subgraphs.

- **Adapting to cluster computing** - In this work, we explored the shared memory paradigm and got good results, making it obvious that parallelism usage is helpful in this particular strategy. We would like to adapt this program to be able to run in a cluster of computers. This could be done using OpenMPI [6] and if each single unit had enough sub units, a hybrid scheme combining OpenMPI between machines and Pthreads locally should be interesting to explore.
- **Dive into more complex topologies** For both practical and readability reasons, this work's focus was on non directed, non weighted and non colored networks. Since one of the advantages of G-Tries is its flexibility, it would be curious to find out how could our method help counting subgraphs with some of those topological features. There would be more subgraphs of each size and thus, it is harder to compute even low  $k$  subgraph counts on those networks, making it even more useful to be able to use the available time efficiently even if we are not able to get the exact count, which could take many hours to complete.
- **Explore convergence and its metrics** The execution time given by the user to our algorithm was the core of the stopping conditions allowed by our infra structure. Even though some metrics of convergence and coefficient of variability were presented, they were not explored in detail and could be useful in justifying early stops in computations, if those were not contributing as much to the overall count guesses and use the adaptive nature of our ideas to craft tasks molded to every degree of freedom in a more successful fashion.

# Bibliography

- [1] David Oliveira Aparício, Pedro Manuel Pinto Ribeiro, and Fernando Manuel Augusto da Silva. Parallel subgraph counting for multicore architectures. In *Parallel and Distributed Processing with Applications (ISPA), 2014 IEEE International Symposium on*, pages 34–41. IEEE, 2014.
- [2] Albert-Laszlo Barabasi. *Linked: How Everything Is Connected to Everything Else and What It Means*. Plume, 2003. ISBN: 0452284392.
- [3] Eunjoon Cho, Seth A Myers, and Jure Leskovec. Friendship and mobility: user movement in location-based social networks. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1082–1090. ACM, 2011.
- [4] L da F Costa, Francisco A Rodrigues, Gonzalo Travieso, and Paulino Ribeiro Villas Boas. Characterization of complex networks: A survey of measurements. *Advances in physics*, 56(1):167–242, 2007.
- [5] Luciano da Fontoura Costa, Osvaldo N Oliveira Jr, Gonzalo Travieso, Francisco Aparecido Rodrigues, Paulino Ribeiro Villas Boas, Lucas Antiqueira, Matheus Palhares Viana, and Luis Enrique Correa Rocha. Analyzing and modeling real-world phenomena with complex networks: a survey of applications. *Advances in Physics*, 60(3):329–412, 2011.
- [6] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.
- [7] Joshua A Grochow and Manolis Kellis. Network motif discovery using subgraph enumeration and symmetry-breaking. In *RECOMB*, volume 4453, pages 92–106. Springer, 2007.
- [8] Roger Guimera, Leon Danon, Albert Diaz-Guilera, Francesc Giralt, and Alex Arenas. Self-similar community structure in a network of human interactions. *Physical review E*, 68(6):065103, 2003.

- 
- [9] Nadav Kashtan, Shalev Itzkovitz, Ron Milo, and Uri Alon. Efficient sampling algorithm for estimating subgraph concentrations and detecting network motifs. *Bioinformatics*, 20(11): 1746–1758, 2004.
- [10] Sahand Khakabimamaghani, Iman Sharafuddin, Norbert Dichter, Ina Koch, and Ali Masoudi-Nejad. Quatexelero: an accelerated exact network motif detection algorithm. *PloS one*, 8(7):e68073, 2013.
- [11] Jérôme Kunegis. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 1343–1350. ACM, 2013.
- [12] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. <http://snap.stanford.edu/data>, June 2014.
- [13] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.
- [14] Wenqing Lin, Xiaokui Xiao, Xing Xie, and Xiao-Li Li. Network motif discovery: A gpu approach. *IEEE Transactions on Knowledge and Data Engineering*, 29(3):513–528, 2017.
- [15] Anna Lubiw. Some np-complete problems similar to graph isomorphism. *SIAM Journal on Computing*, 10(1):11–21, 1981.
- [16] David Lusseau, Karsten Schneider, Oliver J Boisseau, Patti Haase, Elisabeth Slooten, and Steve M Dawson. The bottlenose dolphin community of doubtful sound features a large proportion of long-lasting associations. *Behavioral Ecology and Sociobiology*, 54(4):396–405, 2003.
- [17] Julian McAuley and Jure Leskovec. Image labeling on a network: using social-network metadata for image classification. In *European conference on computer vision*, pages 828–841. Springer, 2012.
- [18] Brendan D McKay. Isomorph-free exhaustive generation. *Journal of Algorithms*, 26(2): 306–324, 1998.
- [19] Luis AA Meira, Vinícius R Máximo, Álvaro L Fazenda, and Arlindo F Da Conceição. Acc-motif: accelerated network motif detection. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 11(5):853–862, 2014.
- [20] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594): 824–827, 2002.
- [21] Pedro Paredes and Pedro Ribeiro. Towards a faster network-centric subgraph census. In *Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pages 264–271. ACM, 2013.

- 
- [22] Pedro Paredes and Pedro Ribeiro. Large scale graph representations for subgraph census. In *International Conference and School on Network Science*, pages 186–194. Springer, 2016.
- [23] Ali Pinar, C Seshadhri, and Vaidyanathan Vishal. Escape: Efficiently counting all 5-vertex subgraphs. In *Proceedings of the 26th International Conference on World Wide Web*, pages 1431–1440. International World Wide Web Conferences Steering Committee, 2017.
- [24] Nataša Pržulj. Biological network comparison using graphlet degree distribution. *Bioinformatics*, 23(2):e177–e183, 2007.
- [25] Pedro Ribeiro. *Efficient and scalable algorithms for network motifs discovery*. PhD thesis, Citeseer, 2011.
- [26] Pedro Ribeiro and Fernando Silva. Efficient subgraph frequency estimation with g-tries. *Algorithms in Bioinformatics*, pages 238–249, 2010.
- [27] Pedro Ribeiro and Fernando Silva. G-tries: an efficient data structure for discovering network motifs. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 1559–1566. ACM, 2010.
- [28] Pedro Ribeiro and Fernando Silva. G-tries: a data structure for storing and finding subgraphs. *Data Mining and Knowledge Discovery*, 28(2):337–377, 2014.
- [29] Pedro Ribeiro et al. Scalable subgraph counting using mapreduce. In *Proceedings of the Symposium on Applied Computing*, pages 1574–1581. ACM, 2017.
- [30] Ryan A Rossi and Rong Zhou. Leveraging multiple gpus and cpus for graphlet counting in large networks. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 1783–1792. ACM, 2016.
- [31] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proceedings of the VLDB Endowment*, 1(1):364–375, 2008.
- [32] Duncan J Watts and Steven H Strogatz. Collective dynamics of ‘small-world’ networks. *nature*, 393(6684):440, 1998.
- [33] Sebastian Wernicke. Efficient detection of network motifs. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 3(4), 2006.