Ain't Nothin But a G-Trie

A Thesis
Presented to
The Division of Mathematics and Natural Sciences
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Arthur James Lawson III

May 2022

Approved for the Division
(Computer Science)

James D. Fix

Acknowledgements

Many thanks go to my thesis advisor, Jim Fix. Your expertise, and relentless energy gave me the push I needed to tackle this problem day in and day out.

Thank you to my friends at Lucas House. You always kept my spirits high and helped me center myself.

To my therapist and my family, thank you for helping me stay focused on my goals even when things got difficult.

Thank you to Torrey Payne for providing me with so much guidance in my four years at Reed. It is my honor to call you a mentor.

Last, but certainly not least, thank you to Jiarong Li. Your previous research with Jim (and Anna Ritz) made my introduction into graphlets a lot smoother than it would have been otherwise. Great work!

Contents

List of Algorithms

List of Tables

List of Figures

Abstract

This thesis considers graphlet census as a method for analyzing large network structures. Used in social and other network analysis, graphlet census counts occurrences of a curated collection of small subgraphs that are common motifs in certain networks. The collection of these counts interpreted as frequencies, can serve as a "fingerprint" for the network and provides a means for comparison to networks of a similar size.

We look at a core algorithm that relies on a g-trie to perform graphlet census. We present this algorithm in three forms. The first solution is sequential while the other two perform the work in parallel. The end goal was to implement a parallel graphlet census with efficient work sharing amongst processors. We describe the different implementations of these approaches and their results. These algorithms follow the work done by Pedro Ribeiro and Fernando Silva ([5] and [6]).

Dedication

I would like to dedicate this to my younger siblings and my niece. Alexandra, Kameron, Kai, and Alondra, there is not a single day that goes by that you don't motivate me to be the best version of myself. Thank you for being you.

Chapter 1

Introduction

Millions of people use social media every day. The amount of data available from this casual use is mind-numbingly large and can be used to serve a variety of important purposes. This includes training artificial intelligence, making more personalized advertisements, and various forms of analysis regarding human social behavior.

As social media becomes more integrated into our daily lives, concerns around privacy are growing. In an ideal world, we can use this data in a way that can help us learn, grow, and improve. Many social network studies claim their data is safe and ethical because it has been anonymized, but what does anonymous mean in this context and how do we measure it? Is data safe because it replaces names with serial numbers? Is it possible to work backwards from an "anonymous" network and connect it to the original data set? These questions have served as a guide in my research. Without a doubt, social media has provided us a platform to analyze social interactions.

One study of social networks is an examination of their structure. The existing interconnectivity amongst individuals can sometimes be used to identify the nature of their interactions. For example, Figures 1.1 and 1.2 ask you to think about what it means for one item in a network to have a relatively large number of connections. In a classroom, this may signal who the teacher is. For a group of collaborating scientists, it may signal who is the most senior (or friendliest). There are many conclusions one can draw from looking at the connections in a network. Beyond simple observations, there is a lot to learn from analyzing network structure. This thesis investigates a particular method for network structural analysis, known as *graphlet census*. In short, graphlet census counts the unique occurrences of different shapes within a larger network and uses these counts for various forms of analysis discussed later in this chapter. For now, let's take a look at what we can learn from network structure in the real world.

1.1 Social Networks

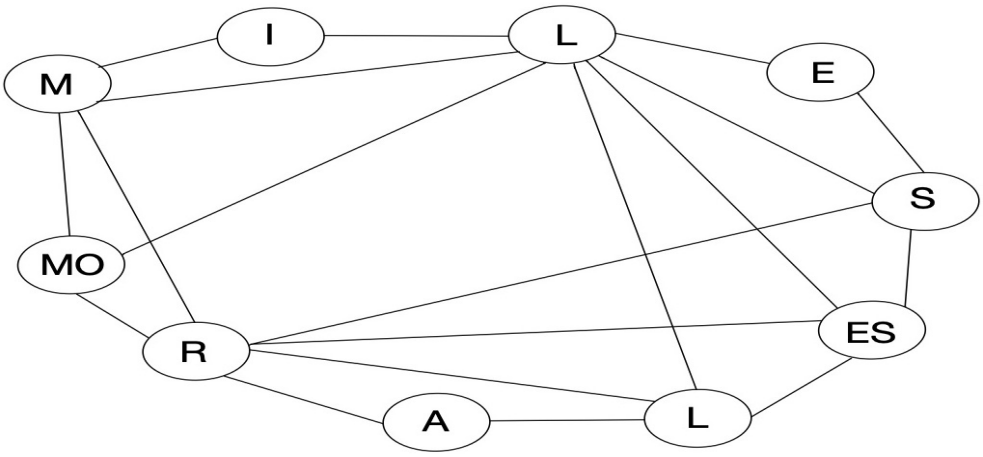
Let's take a look at Facebook. The core purpose is to help people connect with other people. The concept works because a user knows that when they create a profile,

they can easily find and connect with their friends, family, and colleagues. A lot of information can come from analysis of even small friend networks.

For example, a friend network of three people could tell you a lot. In the case where there are zero connections, it is pretty safe to assume that this social network won't be very successful. Each person that posts can only see their own posts. Without friend connections, the social site quickly loses value.

If two out of three of these people are friends, the two of them now have more reason to use this application. Instead of talking to themselves, they are interacting with another person (being social!). Now suppose that the three people form a triangle. This means that they are all friends and become a lot more likely to post and interact because they know that they have two different people that could interact with their posts. They no longer log on to see only what person A had for breakfast; instead, they walk into a virtual world where person B has posted an unbelievable anecdote about getting morning coffee at the same cafe as The Weeknd.

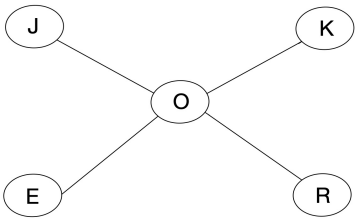
Figure 1.1: Can you pick the professor?



As networks increase in size, the possible shapes, and their implications, grow in complexity and potential value. Figure 1.1 shows an example. In a social network of ten Reed computer science community members, there are a lot of questions to be asked. If I told you the student to professor ratio was 8:2, could you pick out who the two professors are without peeking at their long list of academic degrees? Instead, you can make observations based on their relation to the rest of the crowd.

After a look at the network, one would notice that two of the people have a lot more connections than the rest. It's a safe assumption that the two professors are the people with the higher number of connections because students are interested in their wisdom, humor, and pictures of Eitan flying his plane! Similarly, what if I asked you to identify the tutor in a group of five Reed students (Figure 1.2). It can be observed that there is only one person connected to every other. It would be a safe bet to say that the person in the middle is the tutor.

Figure 1.2: Spot the tutor?

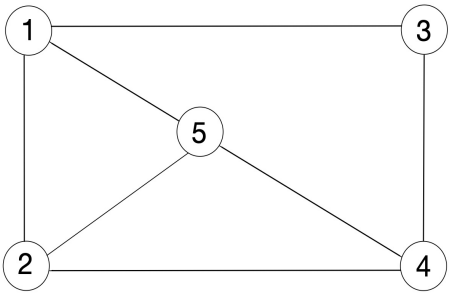


1.2 Subgraph Counting

Analysis of connections within a large network can be simplified with a bottom-up approach. Tens of millions of people in the US alone use Facebook on a daily basis. How can you analyze millions of people and the billions of connections between them? You break the larger problem into a smaller problem.

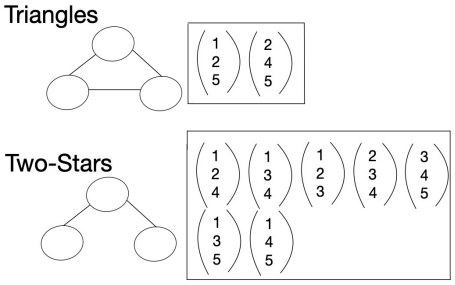
In a network of five people, you can focus on a smaller network of three people. The network of three people is a *subgraph* of the original network. We will provide a more formal definition for subgraph in the next chapter. For now it is sufficient to think of a subgraph as a portion of a larger network. The focus of this thesis is tackling the *subgraph counting problem* which can be defined as: given a specific collection of small graphs (triangles, squares, stars, etc...) and a network, how many times does each small graph occur as a subgraph within the larger network? In any network, there are certain patterns that are bound to become recurring characters. *Network motifs* are reoccurring subgraphs whose regular occurrence or absence are of statistical significance. The motifs of interest vary depending on the application of this problem. So far we've seen an example of a star. Let's take a look at some other shapes that can occur in a network.

Figure 1.3: Subgraph Counting



Figures 1.3 and 1.4 give an example of subgraph counting within a larger network. Figure 1.3 shows a network with five participants and a total of seven connections between them. For this example, let's focus on two shapes of size three: triangles and two-stars. A triangle involves three participants with each one being connected to the other two. A two-star is the same with one less connection. Figure 1.4 shows the unique occurrences of these shapes in the larger graph. Something you may have noticed is that a two-star occurs within a triangle, but the participants within triangle

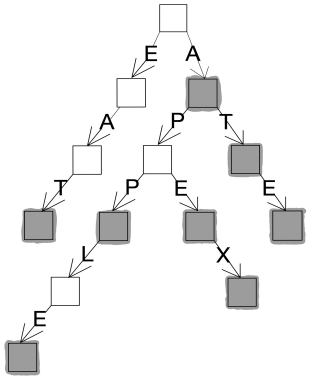
Figure 1.4: Subgraph Counts



are not counted as a two-star. This is because a motif occurrence must respect all connections (or lack thereof) of that specific shape. This is very relevant in the proper counting of shapes as we discuss in subsequent few chapters. It should also be noted that a single vertex can participate in different subgraphs as long as at least one vertex or edge is different. For example, 2 appears in two different triangles, $\{1, 2, 5\}$ and $\{2, 4, 5\}$.

1.3 Trie Search

Figure 1.5: Prefix Trie Example



The primary structure we will be working with in this thesis is known as a graphlet trie (g-trie). Before defining a g-trie, let's use a similar structure known as a *prefix trie* to build some intuition for how g-tries work.

Prefix tries are used to count how many times specific words occur in a text. A prefix trie organizes the text strings as a hierarchy organized as a parent/child relationship in a tree. The siblings within that tree all share the same prefix with their parent and ancestor strings. This structure gives us a compressed guide to the strings we are searching for in a long string of text. The empty string lives at the root of the tree, at each subsequent level, another letter is added to the current word path. A popular tradition at Reed is for seniors to insert random words in their thesis. Part of the fun is to "sneak" it past your adviser. If the adviser sees the sentence "Ape

Ate: The gorilla game app whose apex includes guiding a gorilla to eat an apple," in the middle of a computer science thesis, they may grow suspicious. Conveniently, they can construct a trie similar to Figure 1.5 to see exactly how many times these words occur in the whole thesis.

For example, let's look at the string "Ape Ate: The gorilla game app whose apex includes guiding a gorilla to eat an apple." Scanning through, we start with the word "ape." The letter "a" is scanned first, so we take one step down the right side of the tree. Next is "p" so we step down and left, followed by "e" with a step right. Now that we are at the end of the word, we notice that the box after the final letter is highlighted, indicating that it is a word of interest. Because of this, we increase the count of occurrences for the word "ape." If we were just scanning the word "ap", we would not count it because it is not a word of interest (or an english word at all). Given the context, it is unlikely that the adviser would see many occurrences of most of these words. They can use this low count to make an informed guess that this sentence has ulterior motives.

The benefits of this approach start with having the ability to search for common or related strings in a larger text. Once you find them, the trie offers a convenient and space-efficient way to store them. Let's say someone took this same trie and applied it to a biology textbook. One would reasonably expect this to result in words appearing more frequently. With that increased frequency, we get the benefit of compression. Using a trie relieves the need to store each occurrence of a word. Instead we can follow the path to that work and only store relevant information at the node. This can include location of the word or just an incremented counter. It can also be helpful to have a means for comparing the occurrences of words with a similar ancestry. An example of this is using tries to check the usage of words with various spellings (grey vs gray, enrol vs enroll, lambaste vs lambast, etc...).

That being said, this thesis considers graph search instead of text search. In doing so, we work to find occurrences of *graphlets*, small subgraphs within a larger network. To differentiate between motifs and graphlets, think back to figure 1.3. A triangle is considered a motif, while the specific occurrence of $\{1, 2, 5\}$ is a graphlet. Graphlet search is the network analogue to text search. We explore use of a g-trie data structure. This is the network analogue to a prefix trie. It is used to guide the algorithms of our network search. Figure 1.6 characterizes the g-trie we use in chapters 3 and 4.

Like the prefix trie, it is a tree where nodes corresponds to items we seek in our search. Rather than being short text strings like "Eat", these are instead small graphs like the triangle, star, or two-star. Figure 1.6 shows ten nodes that represent the possible combinations of one, two, three, and four vertices within a graph. Similar to the prefix trie, there is a parent-child relationship encoded by the tree. Since a four-clique of four vertices, contains a triangle of three vertices, it sits below the triangle in the g-trie.

G-tries are very similar to prefix tries. Each step you take adds a vertex to a graph path, and has a similar mechanism that highlights the trie nodes of interest. This is depicted in Figure 1.7, showing the behind the scenes representation of Figure 1.6. For the purposes of my research, all motifs of size three and size four will be

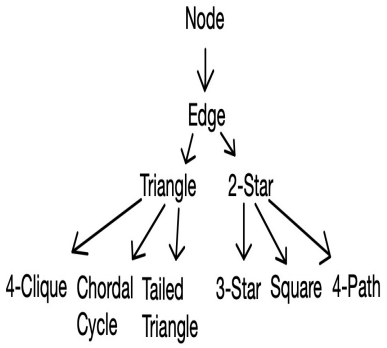
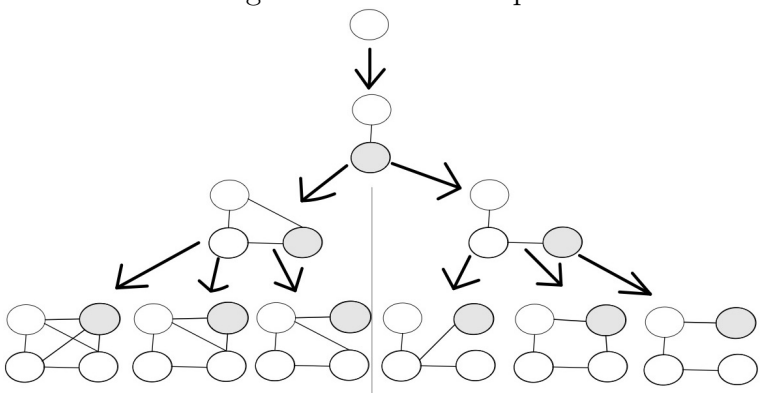


Figure 1.6: Network Motifs

deemed nodes of interest. Large scale subgraph counting becomes much slower when computing subgraphs of size four or greater. Because of this, focusing on "easy" subgraphs of size three and harder subgraphs of size four has shown to be a balanced approach that is scalable to subgraphs of size n. For $n > 4$ the tree would continue to grow to n levels.

Figure 1.7: Motif Shapes



1.4 Graphlet Census in Computational Biology

In the field of computational biology, there exist many use cases for analyzing the topology of a network. Different types of cellular networks are frequently represented as a graph (see Aparício et al. [1] for an example). Vertices represent biological components such as proteins or genes. The physical and/or chemical interactions between them are represented as edges. This analysis extends to local and global topologies within a network. Graphlet census is often helpful for focusing on local structures, but has applications on both levels. Protein-protein interaction (PPI) networks, cell signaling networks, and metabolism networks are all examples of networks that have a history of meaningful research using graphlets. This history includes the comparison of diseased and healthy cell networks to inform treatment plans; as well as cross-species comparisons that allow for insights into evolutionary progress. Brain

networks [3], disease genes [7], and PPI networks [1] have all benefited from research using graphlets. Most cases use motifs up to size five. Graphlet research in this field is limited by graphlets being better suited for undirected graphs. For certain biological use cases, the directionality of an edge holds a lot of valuable information that does not translate well to an undirected search. That being said, it is possible to alter the common approach to fit the needs of a directed graph.

As we grow older, humans become more susceptible to disease. Graphlet census has been useful in research focusing on how the aging of cells contributes to this. GDD analysis allowed Faisal and Milenkovic [2] to draw conclusions about global and local topologies of different PPI networks. They compared networks known to be related to aging with those they predicted to also be aging related. This relation worked in conjunction with analysis into the overlap between the two network sets in terms of their involvement in aging-related, and non-aging related diseases. That being said, graphlets were able to serve their work beyond the use cases in this thesis. In the process of performing a graphlet census, they also stored how many graphlets a particular vertex participated in. This allowed them to measure a specific vertex's centrality. Vertices with a higher level of connectedness serve as an excellent opportunity for treatment. Drugs targeting well connected proteins allow for a treatment with higher impact (as opposed to targeting proteins in more sparse areas of the network).

Sun et al. [7] used various methods of comparing disease networks. Graphlets were the only method based on network topology. Specifically, they conducted a graphlet census to compare genes in human PPI networks. In addition to that, graphlets were used to compare diseases and gain a further understanding of the reason behind and likelihood of co-occurrence for a given pair of diseases. This type of research allows for an increased understanding in how diseases spread and the conditions in which they thrive. It also helps differentiate between diseases with overlapping symptoms. Similar work has also shown this to be useful in disease treatment. In cases where one is trying to treat a lesser-known disease, it helps to classify it accurately among better understood diseases.

Graphlets can also be used for brain research. Brain networks have been identified as "small world" networks [1]. This means that any neuron (vertex) is only a few connections away from any other neuron in the network. Interestingly enough, there have been studies showing neural path lengths have an inverse relationship with a person's IQ. This shows that the "small world" structure of neural networks is directly tied to its efficiency. Kuchaiev et al. looked at a subsection of these networks known as brain functional networks (BFN). While a lot of brain function can be recorded non-invasively using electroencephalography (EEG), BFN requires the higher level of precision granted by invasive procedures using electroencephalography (ECoG). BFN are in charge of working together to respond to changing stimuli and performing cognitive functions. Kuchaiev et al. [3] used Graphlet Degree Distribution to compare the structure of the brain as it performed various tasks. This collection of motif counts serves as the fingerprint for a network, allowing it to be compared to other networks. This study focused on comparing neural activity for one person across different tasks, and across different people on similar tasks.

Chapter 2

Preliminaries

2.1 Graphs

In mathematics, the abstract object most often used to represent networks is called a **graph**. This models the relationship between a collection of entities represented by vertices, and the connections among them, represented by the graph's edges.

Definition 1 (Graph)**.** *A graph $G = (V, E)$ consists of a set of **vertices** V and a set of **edges** E $\{(u, v) \mid u, v \in V, u \neq v\}$*

Definition 2 (Graph Types)**.** *A graph G is directed when all $e \in E(G)$ are interpreted as ordered pairs. G is undirected when they are interpreted as unordered pairs, that is, subsets of size 2.*

Given a formal definition, we can represent the graph in Figure 1.2 as $G = (\{E, J, K, O, R\}, \{(E, O), (J, O), (K, O), (R, O)\})$. Thus far, we have only been depicting undirected graphs. J knows O so there exists an edge (O, J). This is the same as the edge (J, O). If instead, we were depicting the "X consulted Y" relationship of tutoring, then we would have a directed graph where all edges point to the tutor O. This would give us the edge set $\{(J, O), (K, O), (E, O), (R, O)\}$. The return edges (such as (O, J) would not exist unless the tutor consulted one of her tutees.

Weighted graphs are common in travel networks. The weight of an edge is often associated with the time or cost of a given route. Weights are also used behind the scenes to represent the strength of connections in social networks. People who interact more frequently or have more in common, have a higher weight than those with less interaction. Figure 2.1 shows a small example of a weighted social graph.

In Figure 2.1, Art, Jim, and Ian are all friends with varying weights between them. Art and Jim meet weekly and keep a consistent rapport, so they have built up a weight of 18. Art and Ian live together so they have a higher weight of 31. Jim and Ian have not spoke much since Ian's graduation, so their weight is only a 3. Figure 2.2 depicts an example of directed versus undirected graphs. The directionality of edges is similar to social media structures. The way Twitter handles interpersonal connections is similar to a directed graph. Just because I follow LeBron James, does not mean he follows me back. Undirected graphs are more akin to Facebook friends: I can only "friend" with Jim if and only if he is willing to "friend" me.

Figure 2.1: Weighted Social Triangle

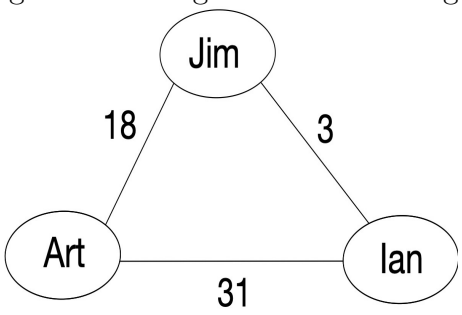
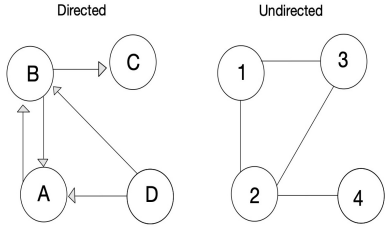


Figure 2.2: Directed vs Undirected Edges



Definition 3 (Neighbor and Degree)**.** *For vertices $v, u \in V(G), v$ is a neighbor of u if and only if $(u, v) \in E(G)$. The set of neighbors of vertex v is denoted $N(v)$. The degree of v is $|N(v)|$.*

In social networks, friends are equivalent to neighbors. The number of friends someone has is their degree. For directed graphs, there are *indegree* and *outdegree*. Indegree counts the number of incoming edges (number of followers), while outdegree counts the number of outgoing edges (number of people you follow).

With rows and columns indexed by $V(G)$, the **adjacency matrix** of G is another representation of its connections. If A is an adjacency matrix, then $A[u, v]$ has the entry 1 whenever $(u, v) \in E(G)$ and is 0 otherwise. For undirected graphs, this is a diagonally symmetrical matrix. The following matrix is the adjacency matrix for the undirected graph in Figure 2.2.

$$\begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \end{array} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix}$$

Definition 4 (Subgraph)**.** *A graph G' is a subgraph of graph G whenever $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$. We denote the set of subgraphs of G as $S(G)$.*

Definition 5 (Induced Subgraph)**.** *Let $G = (V, E)$ and $V' \subseteq V$. The subgraph induced by V' is the graph $G' = (V', E')$ where $E' = \{(u, v) \in E \mid u, v \in V'\}$*

An induced subgraph is a subset of vertices of graph G along with any edges whose endpoints are both in the subset. Figure 2.3 depicts a graph G_1, and two

subgraphs, G_2 and G_3. Both subgraphs are over the vertex set $\{1, 4, 5\}$. G_2 is an induced subgraph, while G_3 is not because it is missing the edge $(4, 5)$.

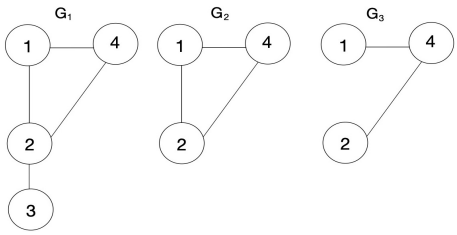


Figure 2.3: G_2 and G_3 are subgraphs of G_1. G_2 is induced by $\{1, 2, 3\}$ and G_3 is not.

Definition 6 (Graph Restriction)**.** *Let $X \subseteq V(G)$. We denote the subgraph induced by X as G/X.*

2.2 Graph Isomorphism

Definition 7 (Isomorphism)**.** *An isomorphism of graphs G and H is a bijection $f : V(G) \to V(H)$ such that $(u, v) \in E(G)$ if and only if $(f(u), f(v)) \in E(H)$. When G is isomorphic to H, we write $G \sim H$.*

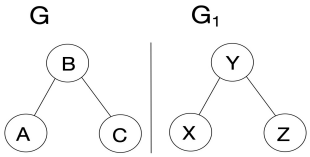


Figure 2.4: Isomorphism Ex. 1

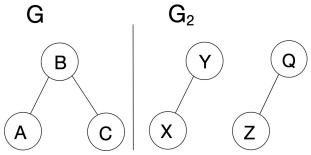


Figure 2.5: Isomorphism Ex. 2

Two graphs G and H are isomorphic when there is a correspondence between their vertices that preserves connectivity. If no correspondence exists where the connections of H respect the connections of G, then they are not isomorphic, i.e. $G \nsim H$. Let's look at some small examples to develop some intuition. Let:

- $G = (\{A, B, C\}, \{(A, B), (B, C)\})$
- $G_1 = (\{X, Y, Z\}, \{(X, Z), (Y, X)\})$

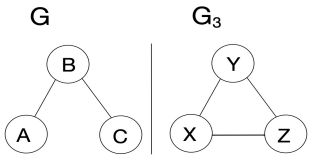


Figure 2.6: Isomorphism Ex. 3

- $G_2 = (\{X, Y, Z, Q\}, \{(X, Y), (Q, Z)\})$
- $G_3 = (\{X, Y, Z\}, \{(X, Z), (Y, X), (Y, Z)\})$

Intuitively, $G \sim G_1$ because $|E(G)| = |E(G_1)|, |V(G)| = |V(G_1)|$, and the edge structure of G can be preserved by G_1 (Figure 2.4). $G \nsim G_2$ because $|V(G)| < |V(G_2)|$ (Figure 2.5). It follows that $G \nsim G_3$ because $|E(G)| < |E(G_3)|$ (Figure 2.6). Now that we have a sense for induced subgraphs, let's take a closer look at what a graphlet is. In the literature, a *graphlet* is any of the collections of undirected graphs of some size k, where k is typically small. For example, Ribeiro and Silva [5] considered graphlets with up to 4 vertices, so $k = 4$. In Figure 1.6, we see subgraphs corresponding to graphlets of size one, two, three, and four. A graphlet g occurs in a graph G when there is a set of vertices $X \subseteq V(G)$ s.t. $G/X \sim g$. For example, in Figure 2.7, there are two triangle occurrences. The larger graph restricted to the vertex set $\{S, B, C\}$, is isomorphic to the shape of a triangle.

2.2.1 Complexity of Finding Isomorphisms

The *subgraph isomorphism problem* can be stated as: given graphs $\{G, G', H \mid G' \in S(G)\}$, determine if $H \sim G'$. This problem is well known to be *NP-complete*. NP is a class of decision problems[1] for which there exists a polynomial-time[2] verifier for possible solutions. NP-complete problems are ones that, by a very technical definition that we omit here, are the hardest problems in the class NP. These hard problems are not believed to have any efficient algorithm for solving them. A problem related to our problems of study is the *clique* problem. A k-clique is a subgraph of size k where all vertices in that subgraph are connected by edges. For example, a triangle is a 3-clique. Determining whether a graph has a clique of size k is NP-complete. The graph isomorphism problem—the algorithmic question as to whether two graphs are isomorphic—is also believed to be a computationally hard problem. It is not known to be NP-complete, yet it is believed to be computationally hard to solve. Some algorithmic complexity researchers believe it to be neither in P nor NP-complete. Of course, for this to be resolved we would need to know whether P $\neq$ NP.

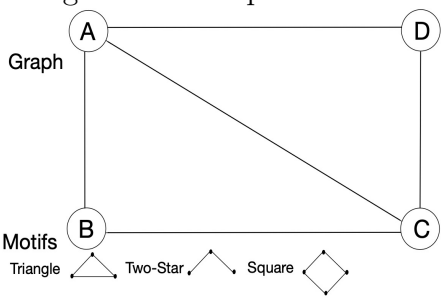
[1] A decision problem has a yes or no answer

[2] Polynomial time is defined as: $O(n^c)$ or better where c is a constant value

2.3 Graphlet Census

The focus of this thesis is to find sequential and parallel solutions to the *subgraph counting problem*. These subgraph occurrences within a larger graph are known as graphlets. The *Graphlet Degree Distribution* (or GDD) is the table used to keep track of these motifs. Each time a new occurrence of motif m is found, GDD[m] is incremented by one. When completed, the counts of all motifs serves as the network fingerprint.

Figure 2.7: Graphlet Census



Graphlet Degree Distribution		
Motif	Count	Graphlets
Triangle	2	{ABC}, {ACD}
Two Star	2	{BCD}, {BAD}
Square	1	{ABCD}

Table 2.1: Simple Census Table

2.3.1 Example

Table 2.1 shows the GDD for Figure 2.7. This census counts triangles, two stars, and squares. It is important to note, that one motif may occur within another of the same degree. Every triangle includes a two star, but it isn't counted as a two star because that collection of vertices has the additional edge a two star is missing. In our algorithm, the motif's adjacency matrix is what helps separate this. The final vertex in a triangle would have a $\{1, 1\}$ relationship to the previous two. This means a connection does exist between the last vertex and each of the first two.This contrasts with the final vertex of a two star having a $\{0, 1\}$ relationship, meaning the latest vertex does not share an edge with the first one.

2.3.2 Complexity of Subgraph Counting

The subgraph counting problem is computationally hard. If it is hard to compute whether or not a subgraph occurs within a network, it follows that counting all

occurrences of multiple subgraphs is even harder. As the motifs of interest increase in size, the overall runtime also grows. Parallelization and g-tries are tools that allow us to work faster and limit the exhaustive search for graphlets. The next section provides additional information to help establish the usefulness of these techniques.

2.3.3 Naive Approach

A naive algorithm for the subgraph counting problem would do an individual search for each possible shape. Initial trials started with motifs of degree three. The following pseudocode shows a brute force solution to searching for all *triangles* and *two-stars* in a graph.

Algorithm 1 Brute force subgraph counting of size 3 in graph G

```
1:  function SubgraphCounting(G)
2:      triangles = []
3:      twoStars = []
4:      forall v1 ∈ V(G) do
5:          forall v2 ∈ N(v1) do
6:              forall v3 ∈ N(v2) do
7:                  if v3 ∈ N(v1) then
8:                      triangles = triangles ∪ {v1, v2, v3}
9:                  else
10:                     twoStars = twoStars ∪ {v1, v2, v3}
11:     return triangles, twoStars
```

Algorithm 1 is constructed of a triply nested loop. The first of which scans through all graph vertices. The second loops through the neighbors of the current v_1 and the second loops through the neighbors of v_2. If there exists an edge (v_1, v_3) then the shape is counted as a triangle (line 7), otherwise the shape is counted as a two star. This algorithm runs in $O(|V|^3)$ time.

This approach has a number of issues. The primary one being that every graphlet is found three times. For every triangle $\{1, 2, 3\}$, time would be wasted also finding $\{2, 1, 3\}, \{2, 3, 1\}$ and so on. As currently constructed, we would overcount graphlet occurrences.

Another issue with this approach is that there must be a completely new search for each shape. As we saw in Figure 1.6, the motifs of degree three build on motifs of size two. Motifs of degree four build on motifs of degree three and so on. Every motif of degree four benefits from being calculated from its parent graphlet. In algorithm 1 we are only searching for motifs of size three. We can do two in one search because there is only one edge that differentiates a *triangle* from a *two-star*. For motifs of size four and beyond, this is not as simple as singular if statement. Early in this process, I tinkered with the algorithm such that I combine multiple motifs of size four in one function. This idea still resulted in multiple scans over all $V \in G$ just for motifs of size four. Then you factor in having to do so again for size three, (and larger depending on the application) and things get messy and inefficient.

For motifs of size three, the runtime is $O(|V|^3)$. It follows intuitively that for motifs of size four, it would grow to $O(|V|^4)$ and that is just for one motif! There are ways to optimize this search with techniques such as symmetry breaking conditions, but fundamentally, it does not scale well. The next chapter introduces *graphlet-tries*, which will be the next step in our quest for sequential success.

2.4 What is Parallelization?

Imagine you and a friend are working on 100 Christmas cards split 50–50. When you are done, your friend has 10 cards remaining, so you split the last 10, 5–5. This way, you are not waiting while your friend works and the overall task gets done faster. The power of good teamwork translates well into the computational world, with some tasks benefiting greatly from a parallel "split the work" approach.

Parallel algorithms are ones where different processors are performing computations simultaneously to solve a problem. One way to do this is to decompose a problem into several smaller subproblems and have them processed and solved in parallel. To illustrate the idea, consider the problem of summing the numbers in an array.

2.4.1 Example

Algorithm 2 Sequential Array Sum

```
1: function SequentialSum(A)
2:     sum = 0
3:     i = 0
4:     while i < A.size do
5:         sum = sum + A[i]
6:         i = i + 1
7:     return sum
```

Let an array A be comprised of n numbers that need to be added together with $A[i]$ denoting the i^{th} element of A. An initial attempt to solve this problem would scan the array and track a running total that is modified with each element processed (Algorithm 2). One processor can add up the values of A in $\Theta(n)$ time. This is fine for small values of n, but as n increases, it can be helpful to spread the load among multiple processors. Two processors can split the work and do it in half the time and four processors can take a quarter of the time. The only issue is the work involved in the sharing and calculation of sums amongst processors. One method that uses $n/2$ processors is known as *parallel tree sum*.

Figure 2.8 shows an example with an array of size 8 and 4 processors. The top list is the initial array with 1-8 enumerating the indices of each element. The blue numbers represent the processor handling each operation. At level one, all four processors are adding up their allotted section. At level two, processors 2 and 4 add the sums from processors 1 and 3 respectively. At level three, processor 4 gets the total sum from adding its own sum with the sum calculated by processor 2.

If we were to do this same calculation with only one processor, we would have to wait on eight calculations: $(32+67), (99+18), (117+54)$...and so on. With a parallel approach, all calculations at one level are occurring simultaneously so there are only 3 total calculations in Figure 2.8. The total work performed by this sum is still $\Theta(n)$, but it is shared by $n/2$ processors. They work collectively over $log_2 n$ rounds, and the parallel running time is $\Theta(log_2 n)$.

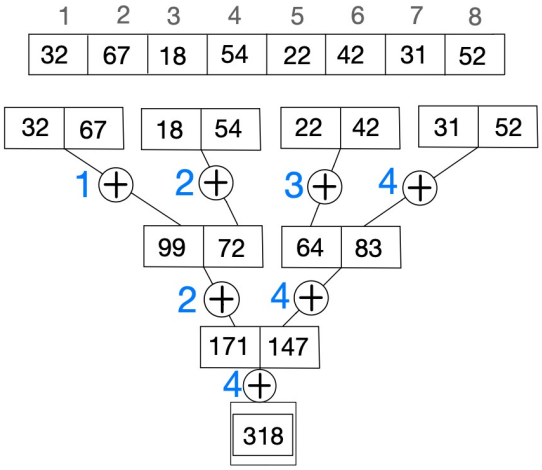


Figure 2.8: Parallel Sum

2.4.2 Parallel Practice

Parallel programming introduces a layer of complexity that takes some time to get used to. My first experience with parallel programming was working in the C++ *POSIX threads* library.

The following is an example of a function `vecSum`. It that takes a `vector, total,` and `window` into the vector as parameters. Threads 1 and 2 are each passed the function `vecSum` and two different windows. After `thread2.join()` has been executed, both threads have completed their task.

```
1  #include <iostream>
2  #include <thread>
3  #include <vector>
4
5  void vecSum(std::vector<int> &v, int &total,
6          int window_begin, int window_end) {
7
8    for (int i = window_begin; i < window_end; i++) {
9        total += v[i];
10   }
11 }
12
13 int main() {
14
15   std::vector<int> vec = {1,2,3,4,5,6,7,8};
```

```
16   int total_1, total_2 = 0;
17   int size = vec.size();
18
19   std::thread thread1 {vecSum, vec, total_1,0, size/2};
20   std::thread thread2 {vecSum, vec, total_2,size/2, size};
21   thread1.join();
22   thread2.join();
23   int total = total_1 + total_2;
24
25   std::cout << "The vector sum is " << total << std::endl;
26   return 318;
27 }
```

A slightly more complex example shows what happens when two or more threads need to alter the same data at the same time. When this happens without proper care, behavior is undefined and results are often far from what they were expected. To solve this, we need a lock to protect the necessary value(s) from being accessed at the same time. In C++, this comes from the *mutex* library. In this example, two processors are incrementing a shared counter variable. To protect it against parallel alterations, the mutex is included when creating the custom type for storing the counter. When `mutex.lock` is called by a processor, no other processor can access the values inside of that object until `mutex.unlock` is called.

```
1  #include <iostream>
2  #include <thread>
3  #include <mutex>
4
5  struct inc_arg_t {
6    int howmany;
7    int counter;
8    std::mutex lock;
9    inc_arg_t(int hm) : howmany {hm}, counter {0}, lock {} {}
10 };
11
12 void increment(inc_arg_t* arg) {
13   for (int i = 0; i < arg->howmany; i++) {
14     arg->lock.lock();
15     arg->counter++;
16     arg->lock.unlock();
17   }
18 }
19
20 int main() {
21
22   inc_arg_t arg { 1000000 };
23
24   std::thread thread1 {increment, &arg};
25   std::thread thread2 {increment, &arg};
26   thread1.join();
27   thread2.join();
28
29   std::cout << "1000000 + 1000000 = " << arg.counter << std::endl;
30   return 318;
```

```
31 }
```

These examples serve to illustrate the kind of coding we'll need in chapter 4 when we give a parallel implementation of graphlet census.

Chapter 3

Graphlet Census With G-Tries

Graphlet-tries are the tool we'll be using to implement our sequential solution. A *graphlet-trie (g-trie)* is defined as a structure used to store graphlets in the shape of a tree. Recall Figure 1.7 where we showed all possible motifs of sizes one through four. This is the visual representation of a graphlet-trie. Each node in the g-trie stores a graphlet. The top node stores the graphlet consisting of a single vertex. The node below it stores the graphlet with two vertices (an edge). This node is the parent of two nodes of size 3 and each of those is the parent of three nodes of size 4. We refer to sibling nodes as those that share a parent. The latest vertex in a sibling node has a different relationship with the vertices inherited from its parent. For example, a triangle and two star share a parent, and therefore share multiple edges and vertices. Where they differ is the latest vertex of a two star is missing the connection to the first vertex that a triangle has.

In our implementation, each node holds a certain amount of information. The most important being `IsGraphlet` and adjacency matrix `A`. `IsGraphlet` is a boolean used to determine if a graphlet is one we are interested in counting. When false, we know the current graphlet is only a middle man between us and deeper graphlets of interest. Recall the prefix trie in Figure 1.5; "ap" is not considered a valid english word, but "app" is. `A` is an adjacency matrix that serves to check if the newest vertex has the necessary connections to the vertices in the current search path.

3.1 My Implementation

To further orient ourselves to what is going on, let's take a bird's eye view of the algorithm before doing a deeper dive. The algorithm's search walks through the graph, along a path of vertices. As it does so, it works its way down the g-trie, going down one level each time it extends the path. At any given point, it has walked some path p and is within some g-trie node x. If that x corresponds to a graphlet g that we are interested in counting, we include (p, g) in the census. From here, the idea is to be careful about how we extend p with candidate matching vertices v. We take steps further to v whenever $G/p \cdot v$ corresponds to the graphlet of a child g-trie node c of x.

Looking at the code, (line 4) of `Match` takes that next step, then seeks to repeat this `Match` step recursively with each child c of x. This happens in the loop of (lines 7-8). (Line 2) of `Match` is where we find vertices v that work for the subgraph of our current node x. There, we rely on `MatchingVertices` (which calls `CandidateCreator`) to find these valid path extensions.

We start with an empty path and the trivial trie node. We then create a collection of vertices to match the current trie node we are in search of. Since our search is just beginning, we only need to construct a singular node, corresponding to one graph vertex. Because of this, all graph vertices are matches. In a way, these are just the seeds of a depth-first walk through G in search of our graphlets. The depth-first search is guided by the g-trie. We then check if this is a node of interest; if so, we output the current path. We continue this for the current trie node's children until we have reached the maximum node size. In our case, this occurs at size four. Once we have reached our maximum depth, we go back up one level and continue the search with the remaining matches at that level. Following the pattern of a depth first search, eventually we run through all paths starting with the initial vertex. We then move on to the next potential start vertex, and continue until all possible paths have been exhausted.

Algorithm 3 provides a method for finding and counting all occurrences of the g-trie nodes as induced subgraphs of an inputted graph G. Recall, the core benefit of a tree structure is the ability to find partial matches for potential graphlets and being able to continue searching without starting from scratch. We start with an initial call to `GTrieMatch` with the root node of our g-trie and a graph G. Each child of the trie node is then passed to `Match` with an empty list representing the current partial match of vertices in a g-trie path. On (line 2) of `Match`, `MatchingVertices` is called to gather all vertices that match the current g-trie node, x. If x corresponds to a motif of interest, then we output that graphlet occurrence (lines 4 and 5). Next we make a recursive call to `Match` on all g-trie nodes that are children of the current node x.

Let's take a deeper look into the functionality of `MatchingVertices`. On (line 2), we see a call to `CandidateCreator`. If the current path (`p`) is empty, all vertices of G are candidates. An empty path means that we are just starting our graphlet search at some place in the graph. In that situation, all vertices are possible matches. This is because we are working from the root g-trie node and a match is comprised of a single vertex. In general, if the current path is not empty, then we collect all neighbors of the vertices in the current path (that are not already in the path). We then repeat this by collecting all of the non-path neighbors of the vertices in the candidate pool. Once we return back to `MatchingVertices`, we use `CheckConnections` to check each candidate's relationship to the current path (MV line 5). If a candidate vertex matches the required relationship to each of the vertices in the current path, it is added to the set of matching vertices. In the code, `CheckConnections` does this check by comparing the adjacency matrix of x with the adjacency matrix of G. The adjacency matrix for G is a $|V| \times |V|$ matrix for all possible edges in G. Since the adjacency matrix of a g-trie node x is shared as a sub-matrix of all of the g-trie nodes that are ancestors of x, we only need to store a row of the adjacency matrix at each g-trie

node.

For example, if x were a triangle, the current path would have two vertices. Since a triangle is constructed of three vertices, with each being connected to the other two, the required adjacency matrix would be $\{1, 1, 0\}$. The newest vertex would need to connect to each of the prior vertices without a self-looping edge. This is implemented in the actual code, but instead denoted as $G \setminus p \sim x.graphlet$ in the pseudocode (MV line 5).

Let's take another look at (line 5) of `Match`. In the previous chapter, we pointed out that the naive solution is incorrect because it does not account for duplicates. Without symmetry-breaking conditions, there are still a few ways to protect against this. In my implementation, I stored a sorted version of the current path in a C++ *set* object to avoid duplicates being counted.

3.2 Different Faces of a G-trie

For this sequential solution, I followed the pseudocode from Ribeiro and Silva [5] pretty closely. Like any pseudocode, there was room for choices to be made in my implementation. These choices are typically made for style, efficiency, readability, and/or ease of debugging. In this section, we will discuss some of the implementation decisions I made along the way. One of the first decisions I had to make was choosing not to use *canonical form*. Canonical form is the simplest way to represent an object in which it can be uniquely identified[1]. Ribeiro and Silva used it to reduce the size and complexity of an inputted graph. Because of this, not only does the size of the graph decrease, but it also reduces the number of duplicate calculations. Ribeiro and Silva [5] go into more detail about specific forms they decided were optimal for this use case.

The sequential g-trie paper by Ribeiro and Silva [5] has more details on a custom heuristic function they used to create the graphlet trie. In summary, this can be useful for use cases where specific motifs are not as important as getting a sense for the structure of a network. In other cases, such as cell-network analysis, the motifs are crucial to the search itself and must be customized. My research includes all motifs of size three and four, resulting in a trie structured like Figure 1.7.

Another decision I made was in `MatchingVertices`. I decided for readability, to split the core functions of `MatchingVertices` into `CandidateCreator` and `CheckConnections`. In candidate creation, Ribeiro and Silva [5] decided: from the neighbors of all $v \in p$, they selected the neighbor with the smallest neighborhood. Instead my solution collects all neighbors of every $v \in p$ not already in `p`. From there I included all of their neighbors to ensure that every possible path was considered. In chapter 5, we talk more about this being a point of optimization and continuation for this line of research.

[1]Canonical form can also be used to solve the graph isomorphism problem. Two graphs are isomorphic if and only if they have the same canonical form.

Algorithm 3 Census of sub-graphs of x in graph G

```
1: procedure GTrieMatch(x, G)
2:     forall children c of x do
3:         Match(c, ∅)
```

```
1: procedure Match(x, p)
2:     M := MatchingVertices(x, p)
3:     forall v ∈ M do
4:         p' = p · v
5:         if x.isGraphlet then
6:             output p'                ▷ Indicates that we have found a match
7:         forall children c of x do
8:             Match(c, p')
```

```
1: function MatchingVertices(T, p)
2:     M = ∅
3:     C := CandidateCreator(p)
4:     forall v ∈ C do
5:         if G \ p ∼ x.graphlet then
6:             M := M ∪ {v}
7:     return M
```

```
1: function CandidateCreator(p)
2:     if p = ∅ then
3:         C := V(G)
4:     else
5:         C := {v : v ∈ N(p)}
6:         C := C ∪ {v : v ∈ N(C)}
7:         return C
```

3.3 Examples

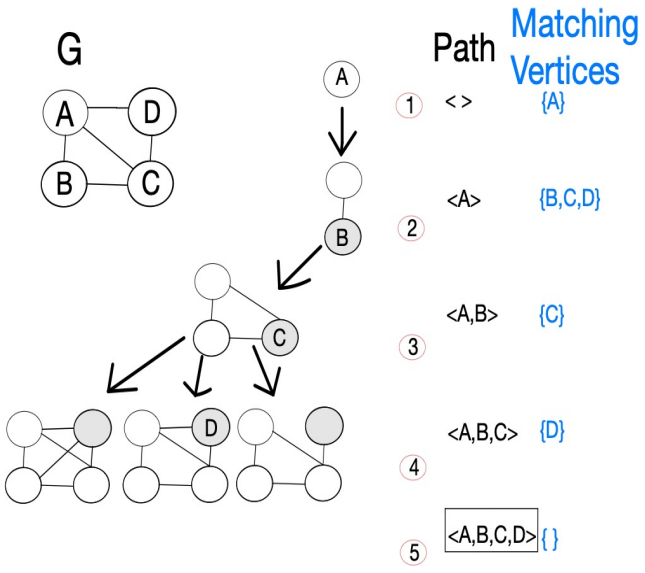


Figure 3.1: Sequential Walk

For the following examples, only graphlets of size 4 are considered motifs of interest. The only thing this changes is simplifying how many times we must output a current path.

3.3.1 An example walk through

In Figure 3.1, We start with a walk through example, followed by a larger one in figures 3.2, 3.3 and 3.4. Figure 3.1 depicts a graph G, g-trie, current path, and a set of matching vertices. The circled numbers indicate the current step in the walk through.

In the sequential solution, we only keep track of the matching vertices at the current depth. At each step, the g-trie node highlights the most recent addition to the current path. At step $\{1\}$, `p` is empty and A is our lone vertex. At step $\{2\}$, we see that $\{A\}$ has two neighbors, $\{B, C\}$ and B is chosen. Step $\{3\}$ follows similarly. At the end of step $\{4\}$, we have found an induced graphlet of size 4! Because of this, step $\{5\}$ outputs the current path of $< A, B, C, D >$.

3.3.2 A longer walk through

Figures 3.2, 3.3 and 3.4 show us a more complex example using graph Q. Steps $\{1\}$ through $\{5\}$ of Figure 3.2 follow the same as Figure 3.1. At the conclusion of step $\{5\}$, we have reached the maximum depth for graphlets of interest. Now we step back up one level and see that there exists an unexplored path. Step $\{6\}$ in Figure 3.3 shows the current status of us returning to level $\{4\}$ and moving past the previously

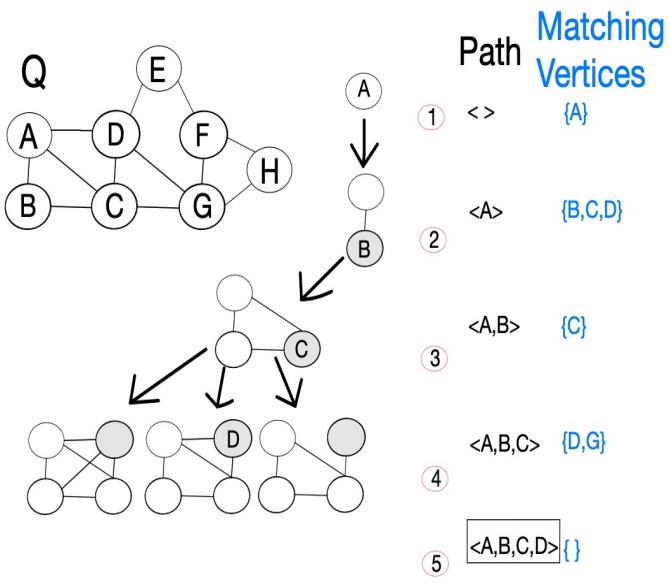


Figure 3.2: Larger Sequential Walk

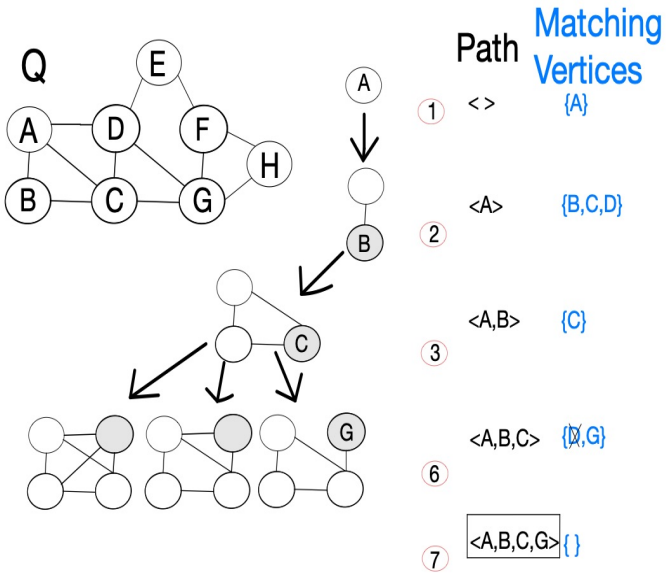


Figure 3.3: Larger Sequential Walk Snapshot

explored path at the current depth. D is crossed out, the previous path $< A, B, C >$ continues with G. This gives us a different graphlet than the one outputted in Figure 3.2.

In Figure 3.4, we continue through the rest of its neighbors that weren't selected. This gives us a new outputted graphlet of $\{A, C, D, E\}$. From there we go even further back up the stack to process through potential paths in the same fashion as Depth First Search. When all neighbors of A have been fully processed, we return to (line

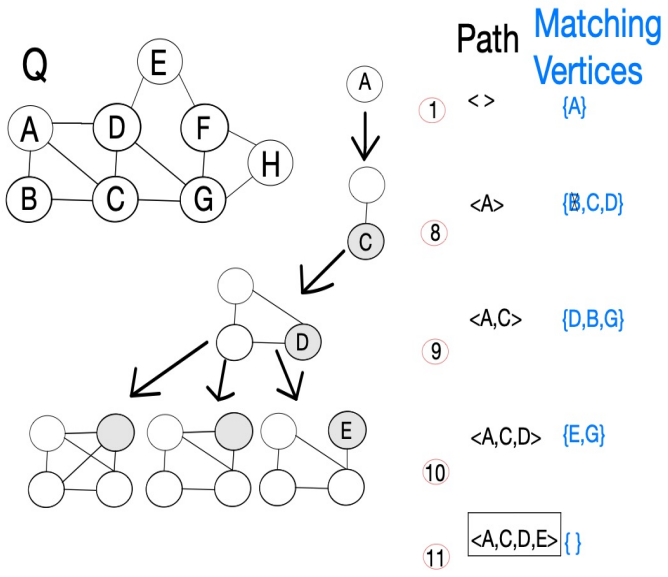


Figure 3.4: Larger Sequential Walk Continued

3) of `Match` and continue with the second vertex returned from `MatchingVertices` and so on. Recall, since the root of the g-trie is a node, the next path would start with a singular vertex.

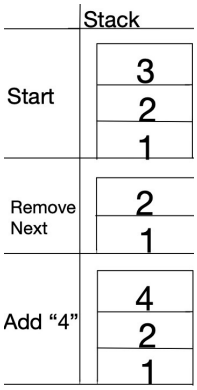


Figure 3.5: Stack Example

The concept of working back up through previous levels is known as a *stack*. A stack is a collection of objects processed in a particular order. Think of a stack of pancakes. Putting them on a plate, the first pancake goes on the bottom, second on top of the first, third on top of the second and so on. Eating one at a time means that you eat the topmost (most recent) pancake, then the second, and so on. See Figure 3.5 for an example. Objects $1, 2$, and 3 are placed on the stack in that order. When removing from the stack, 3 is removed first. When adding a new object 4 to the stack, it is added to the top and will be next in line to be removed.

The importance of symmetry breaking conditions is shown again here. There are

different ways to come across the same graphlet. When we return up the stack, there will be different routes to the same path (such as the triangle $\{0, 1, 2\}$ also being found as $\{0, 2, 1\}, \{2, 0, 1\}$, and so on). Our algorithm handles this by using C++ *set* objects to maintain no duplicates being stored in the GDD counter. We will talk in chapter 5 about what an optimized setup would look like with symmetry breaking conditions.

Chapter 4

Parallel Census G-Tries

Following the completion of the sequential algorithm, we set our sights on a parallel solution. We worked on two models: one with work sharing and one without. *Work sharing* is defined as simply as it is named: when one processor is done with its work, it asks for more from one of the processors still grinding away. The goal is to share work as seamlessly as the Christmas card example at the beginning of section 2.4.

4.1 Without Work Sharing

Algorithm 4 Parallel Census

```
1: function ParallelCandidateCreator(p, processor_id)
2:     if p = ∅ then
3:         forall i = processor_id, i < |Vertices|, i+ = P do
4:             C := C ∪ G(V)[i]
5:     else
6:         C := {v : v ∈ N(p)^1}
7:         C := C ∪ {v : v ∈ N(C)}
8:         return C
```

Transitioning from sequential code to an initial parallel solution did not require many changes to the fundamental approach. To create a parallel algorithm, we start by looking at the functionality within `MatchingVertices`, more specifically, `CandidateCreator`. In the sequential solution, when `path` is empty, all vertices within the graph are listed as candidates. For our parallel version, this initial allocation is divided up by the number of processor round-robin style, similar to what was done by Ribeiro et al. [6]. If there are 100 vertices and 10 processors, processor 1 will get vertices 1,11,21..., processor 2 will get 2,12,22... and so on. This division means that each processor will have its own set of starting vertices as it scans for motifs of interest. At the top level of the algorithm, `GTrieMatch` is called for each processor, and they each conduct an independent search with their unique search pool of starting vertices. This shows what would happen if we modified the existing

structure. In the following section, we discuss a different style better suited for work sharing.

While the parallel approach offers significant improvement over the sequential algorithm, it still has room for improvement. For example, there is no easy way to evenly divide the work amongst processors. In my implementation, vertices are serialized as they are read in from a source file. While the original vertex set may be comprised of $\{31, 42, 75\}$, these vertices would be relabeled $\{1, 2, 3\}$. This is important because it means that all P processors start with a search pool of $|V|/P$ vertices. Although they each start with the same number of vertices to work from, this is no guarantee of an even workload. Every graph vertex has between 0 and $|V| - 1$ neighbors. This normally leads to some processors having much more work than others, resulting in some processors finishing sooner than others. Think back to the Christmas card example at the top of section 2.4. What if, instead of a 50–50 split, the cards were divided 95–5. On top of that, you and your friend are working in separate buildings, with no means of communication or ability to share unfinished cards. These are the current working conditions faced by the processors. It makes sense to work towards a solution where a processor can ask for more work once it completes its initial allocation.

4.2 Work Sharing

The model we adapted for work sharing continues a similar process to what we have done thus far. However, we have structured it in a way that is more intuitive and allows for straightforward communication amongst processors. Before diving into our work sharing implementation, let's start with a new implementation of our sequential algorithm to help build an intuition for what is going on. This work relies on a *work queue*.

The work sharing graphlet paper by Ribeiro et al. [6] introduced the work queue as a means to facilitate communication between processors. Each processor is assigned a work unit. The work unit, roughly speaking, contains all the information needed to describe a snapshot of the sequential depth first search in progress. When a processor finishes its initial search, it makes a request to the global work queue, waiting until another processor is able to stop and share work. The values returned from `MatchingVertices`, current depth, path, and a set of unexplored nodes are all essential pieces of information stored in the work unit. It also stores the current g-trie node of interest. Localizing this information to each processor allows for fluid communication amongst them. Before discussing the algorithm, let's develop an understanding of how each piece of information is stored and why.

The structure `w.matches` represents the values returned from `MatchingVertices` (line 12). It is separated by depth to keep track of what would be the recursive stack. We store a numerical value as `w.depth` to represent the length of the current path. The current g-trie node of interest is stored as `w.currnode`. This retains the same level of information as it did in the sequential algorithm (such as `isGraphlet`). Lastly, `w.gnodes` stores the next round of nodes to explore. Similar to `w.matches`,

this is also stored by depth. At depth n, our path has size n. Our current node is the corresponding g-trie node to our path. At this depth, our `gnodes` are children of the current node. For example, suppose we are at depth three with the current node being a triangle. `w.gnodes` would consist of its three child nodes.

4.2.1 Alternative Sequential Implementation

Algorithm 5 shows this sequential adaptation. Instead of recursive calls to `Match`, we've implemented a looping search dependent on our current depth. If depth equals zero, then we know the search has been completed. `Count` is passed a work unit `w`. The current node begins with the root trie node, which is equivalent to the empty string in a prefix trie. Since the node at depth 1 consists of a singular graph vertex, `w.matches` is populated with all $v \in V(G)$.

(Line 4) pulls the first matching vertex at the current depth. We then increment depth (line 6) and check if the current node is one of interest (lines 7-8). We then populate `gnodes` at our new depth with the children of our current node (line 9). This process repeats until we get to the maximum depth allowed by the current path. Once we reach that depth, `matches` and `gnodes` are both empty, therefore we remove the latest node from the end of our current path (line 14) and decrement depth (line 15). This returns us up one level in the call stack.

A snapshot into the algorithm would look as follows: suppose the maximum path depth for our first search ended on a *4-clique*. Removing the last vertex moves us back up to a triangle. The current node is triangle. Each matching vertex in `matches` forms a 4-clique with the current path, and `gnodes` is populated with the remaining children of a triangle. We repeat the condition from (lines 3-9) until all potential cliques with the current path are found. Once the matches at depth 3 have been exhausted, we jump down to (lines 10-12) where we pull in the next child node of a triangle, populate `matches` accordingly, and repeat (lines 3-9) for this new node. Once we have exhausted the children of a triangle, we step back up to its parent node (edge) and repeat.

Similar to Algorithm 3, we can use this same structure to implement a parallel search without work sharing. The initial allocation of vertices is the only substantial change. Each processor makes a different call to `Count`, performing an independent search with a unique set of initial matches.

4.2.2 Work Sharing Implementation

Before seeing how the alternative sequential algorithm is modified into one with intuitive work sharing, we must define the functions that make work sharing possible. `AskForWork` and `OfferWork`. When a processor has completed its initial workload, it makes a request for more work. This sends a message to the other processors, requesting one of them to share their remaining workload with the requester. `OfferWork` begins by checking if there are any active requests for work. If so, then it pulls in a request and splits its `matches` at each level in half to share with the requesting processor. The new processor receives a copy of the sender's `gnodes`, current node, and

Algorithm 5 Alternate Graphlet Census

```
1: procedure COUNT(w)
2:     while w.depth > 0 do
3:         if w.matches[w.depth] ≠ ∅ then
4:             v := NEXTMATCH(w.matches[w.depth])
5:             PUSHPATH(w.path, v)
6:             w.depth = w.depth + 1
7:             if w.currnode.isGraphlet then
8:                 OUTPUT(w.currnode, w.path)
9:             w.gnodes[w.depth] := CHILDREN(w.currnode)
10:        else if w.gnodes[w.depth] ≠ ∅ then
11:            w.currnode = NEXTGNODE(w.gnodes[w.depth])
12:            w.matches[w.depth] = MATCHINGVERTICES(w.path, w.gnodes)
13:        else
14:            POPPATH(w.path)
15:            w.depth = w.depth − 1
```

path. This is because they both resume working from where the sender is currently at. The split in matching vertices allows them to explore different paths branching from the same start path. We see an example of this in section 4.3.

We see the work sharing approach implemented in Algorithm 6. The only change at this level is the addition of a new conditional check based on the current depth (lines 16-20). When depth equals zero, then the processor has completed its current workload. It then makes a request for more work and updates its depth to match its new path. If a processor has not completed its work, it checks to see if there are any pending requests currently.

4.2.3 Beyond the Pseudocode

To properly share my experience, it is helpful to provide some additional context. My experience working from the work sharing pseudocode from Ribeiro et al. [6] left even more room for implementation choice when compared to the sequential algorithm. For example, Ribeiro et al. [6] constructed a work unit using one continuous array to store current depth, `unexploredVertices`, `unexploredNodes` and `path`. This requires careful indexing to pass this work array to functions `DivideWork` and `ResumeWork`. In my implementation, I used a C++ class to create a work unit object. This way, I could track the same values by name and store them with different data structure that made sense on a case-by-case basis. The change in structure shown in Algorithm 6 is another example of a stylistic choice I made along the way. While it functions similarly to to the adaptation they used, my implementation simplified how the processors share and continue work after a request. This results in more readable code that is easier to debug.

Algorithm 6 Work Sharing Graphlet Census

```
1: procedure ParallelCount(w)
2:     while w.depth > 0 do
3:         if w.matches[w.depth] ≠ ∅ then
4:             v := nextMatch(w.matches[w.depth])
5:             pushPath(w.path, v)
6:             w.depth = w.depth + 1
7:             if w.currnode.isGraphlet then
8:                 output(w.currnode, w.path)
9:             w.gnodes[w.depth] := children(w.currnode)
10:        else if w.gnodes[w.depth] ≠ ∅ then
11:            w.currnode = nextGnode(w.gnodes[w.depth])
12:            w.matches[w.depth] = MatchingVertices(w.path, w.gnodes)
13:        else
14:            popPath(w.path)
15:            w.depth = w.depth − 1
16:        if w.depth = 0 then
17:            AskForWork(w)
18:            w.depth = |w.path|
19:        else
20:            OfferWork(w)
```

4.2.4 Thread-Safety

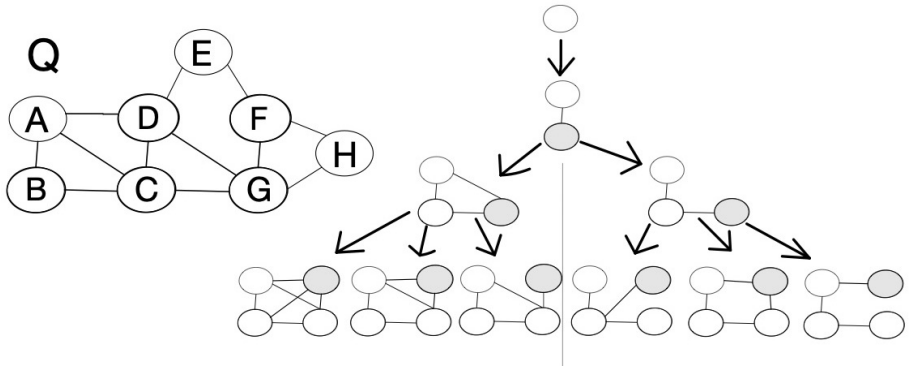
Another important aspect of the work-sharing model is proper thread safety. The original parallel algorithm had to safeguard against multiple writes to a global GDD. This means that two processors attempted to put new paths into the GDD at the same time. Competing writes lead to undefined (and very problematic) behavior without proper protection. Initially, we fixed this by creating a thread-safe GDD object, locking access whenever one processor was actively writing to it. With frequent writes from all processors, this locking, unlocking, and associated overhead, caused large slow downs. We solved this problem by diverging from the sequential code and giving each processor a local GDD to write to. Once all calculations have been performed, these are all aggregated into one global GDD. This offered great speedups that put us a lot more in-line with the hope of linear speedups as the number of processors increases.

Once we got to the work-sharing model, thread safety became even more important. In addition to GDD, we need to now protect the mechanism in charge of handling work requests. My implementation used a *global queue*. This queue object stores pointers to the work units of requesting processors. Anytime a processor checks for requests to fulfill, the requesting work unit gets dequed from the queue. We must protect against multiple processors attempting to fulfill the same request. We must also protect against parallel writes to avoid undefined behavior. Similar to our first approach for GDD, I created a custom class with a mutex to protect calls to *enqueue*

and dequeue. This protects against parallel reads and writes to the global queue.

4.3 A More Complex Example

Figure 4.1: Parallel Walk Graph



Here, we show a step-by-step example of how work sharing operates. We reuse the same graph from Figure 3.2, depicted in Figure 4.1. The g-trie structure is also included for convenience.

Processor 0				Processor 1		
Path_0	MatchingV_0	Motif_0	**Step**	Path_1	MatchingV_1	Motif_1
$<>$	$\{A, C, E, G\}$	**none**	**0**	$<>$	$\{B, D, F, H\}$	**none**
$< A >$	$\{C, B, D\}$	Node	**1**	$< B >$	$\{C, A\}$	Node
$< A, C >$	$\{D, B\}$	Edge	**2**	$< B, C >$	$\{D, A, G\}$	Edge
$< A, C, D >$	$\{E\}$	Triangle	**3**	$< B, C, D >$	$\{E\}$	Two Star
$< A, C, D, E >$	$\{\}$	Tailed Triangle	**4**	$< B, C, D, E >$	$\{\}$	Four Path

Table 4.1: Parallel Census Table

Table 4.1 provides a walk through for the graph $Q = (\{A, B, C, D, E, F, G\}$, $\{(A, B), (A, C), (A, D), (B, C), (C, D), (C, G), (D, E), (D, G), (E, F), (F, G), (F, H), (G, H)\})$, when $P = 2$. At each step, processors p_0 and p_1 are given an allocation of vertices ($MatchingV$), path, and motif. Since there are two processors, p_0 receives the odd indexed vertices and processor 1 receives the even. The motif at step k matches the path at step k. For example, triangle is the motif for p_0 at step 3, and the corresponding path is $\{A, C, D\}$. It also holds true that the motif at step k is the shape obtained by adding the next vertex in the allocation to the path at step $k - 1$. This means the $k - 1$ allocation of vertices, all correspond to the motif at step k. At step 3, D has 2 neighbors not currently in the path, B and E. E is the only allocated vertex because it is the only one that can create a Tailed Triangle at step 4.

Steps 1-4 continue similar to the sequential solution. This results in $< A, C, D, E >$ and $< B, C, D, E >$ being stored at their respective motifs. Table 4.2 is the current status for each processor at the conclusion of step 4. Similar to the sequential algorithm, the subsequent steps continue working up the stack until processing through

Processor 0			
Step	Path_0	MatchingV_0	Motif_0
0	<>	$\{\cancel{A}, C, E, G\}$	**none**
1	$< A >$	$\{\cancel{C}, B, D\}$	Node
2	$< A, C >$	$\{\cancel{D}, B\}$	Edge
3	$< A, C, D >$	$\{\cancel{E}\}$	Triangle
4	$< A, C, D, E >$	{}	Tailed Triangle
Processor 1			
Step	Path_1	MatchingV_1	Motif_1
0	<>	$\{\cancel{B}, D, F, H\}$	**none**
1	$< B >$	$\{\cancel{C}, A\}$	Node
2	$< B, C >$	$\{\cancel{D}, A, G\}$	Edge
3	$< B, C, D >$	$\{\cancel{E}\}$	Two Star
4	$< B, C, D, E >$	{}	Four Path

Table 4.2: Snapshot after one round

the initial allocation of start vertices. We now fast forward to when things start to get interesting. At step 5 (Table 4.3), p_0 has completed its work (`path` and $MatchingV$ are both empty). At step 6, p_0 makes a request for more work. P_1 then stops its work, stores its current state, and prepares to make a call to `DivideWork`.

Processor 0				Processor 1		
Path_0	MatchingV_0	Motif_0	**Step**	Path_1	MatchingV_1	Motif_1
<>	$\{\cancel{A,C,E,}\ \cancel{G}\}$	**none**	**5**	<>	$\{\cancel{B,}\ \cancel{D}, F, H\}$	**none**
please	send	work	**6**	$< F >$	$\{E, H, G\}$	Node

Table 4.3: Snapshot of work being requested

Table 4.4 shows the aftermath of this transaction. At step 8, p_0 has copied the current path and motif from p_1. They split the allocated vertices at steps 7 and 8. From here, they both resume search as normal.

4.4 Performance

While chapter 5 is dedicated to more formal performance analysis, I can use this time to tell you what my experience was. Initially, I did not see great speedups transitioning from my initial parallel code to proper work sharing. Part of this was dealing with standard debugging. But the core issue I faced was handling the shared GDD between processors. Work sharing in itself does not leave much room for undefined behavior because those calls occur less frequently. Very rarely do several occur at the same time. Each processor is handling hundreds or *thousands* of matches that must all be counted. This creates a lot of opportunities for parallel writes that cause undefined behavior. This causes invalid results and program crashes. Once I transitioned to a localized GDD solution, my results got faster and more consistent. The additional

Processor 0			
Step	Path_0	MatchingV_0	Motif_0
7	$<>$	$\{H\}$	**none**
8	$< F >$	$\{G\}$	Node
9	$< F, G >$	$\{C, D\}$	Edge
10	$< F, G, C >$	$\{D\}$	Two Star
11	$< F, H, G, D >$	$\{\}$	Tailed Triangle
Processor 1			
Step	Path_1	MatchingV_1	Motif_1
7	$<>$	$\{F\}$	**none**
8	$< F >$	$\{E, H\}$	Node
9	$< F, E >$	$\{D\}$	Edge
10	$< F, E, D >$	$\{G, C, A\}$	Two Star
11	$< F, E, D, G >$	$\{\}$	Square

Table 4.4: Snapshot after work is shared

work of aggregating the data in the end added a negligible amount of time to the total runtime. Even when using 128 threads working with a large collaboration network of astrophysicists from the Stanford Network Analysis Platform [4], this took under 30 seconds. For context, the search itself took 40 hours with work sharing. All of this goes to say, localizing GDD to each processor was the better design.

Chapter 5

Results and Discussion

5.1 Results

In this chapter, I describe the performance and results of running my code on sample data. I conducted final performance testing using another SNAP network. I chose the *Gnutella peer-to-peer file sharing network*, collected on August 9th, 2002 [4]. I decided on this particular data set because of its vertex (8,114) to edge (26,013) ratio. I wanted a data set that would take approximately 90 minutes for the sequential solution to complete. This would allow me to do several runs, including runs with a variable number of processing threads. The sequential algorithm completed its census in 1 hour, 30 minutes, and 36 seconds. This time served as the benchmark for comparing our parallel implementations. Linear scaling is often used as a goal for parallelizing a sequential algorithm. With *linear scaling*, each doubling of threads results in a halving of runtime. An added benefit of choosing a SNAP data set is that it comes with triangle counts pre-calculated. This serves as an additional verification that our code is computing correct graphlet counts. All tests were run on a *AMD Ryzen Threadripper 3970X* with 32 cores and two threads per core. Both parallel solutions were run three times then averaged, and their averages are compared in Figure 5.1.

5.1.1 Analysis

Tables 5.1.1 and 5.1.2 show performance numbers for the parallel and work sharing solutions respectively. Starting with the parallel solution, we get near linear speedups until we reach 32 threads. This isn't a case of diminishing returns, but rather a drop off from 16 to 32 and another dip from 32 to 64. Load balancing plays a large part in the performance of this algorithm. As we discussed previously, an even distribution of initial vertices does not correspond to an even amount of work. This means that, without a method for work sharing, some threads will inevitably have much more work than others. Some threads will inevitably finish faster, while the rest are still busy at work. Workloads are more even when all graph vertices have a similar number of neighbors.

For work sharing, we get near linear speedups until we go from 16 to 32 threads (as seen in Figure 5.2). The overhead for my implementation of work sharing increases

Parallel Performance (H:MM:SS)				
Threads	Trial 1	Trial 2	Trial 3	**Average**
1	1:30:54	1:30:56	1:30:50	1:30:54
2	0:48:23	0:48:19	0:48:21	0:48:21
4	0:27:41	0:27:49	0:27:39	0:27:43
8	0:14:37	0:14:37	0:14:37	0:14:37
16	0:11:28	0:11:28	0:11:28	0:11:28
32	0:07:04	0:07:07	0:07:07	0:07:06
64	0:06:47	0:06:47	0:06:47	0:06:47
128	0:04:59	0:05:01	0:05:02	0:05:01

significantly at this point. Using print statements to see exactly when work was being exchanged, it became clear that the "first come, first served" implementation does not do well when there are frequent requests for work. For example, a thread that very recently asked for work, would be the first to offer work to another requesting thread. This leads to smaller workloads being shared, and more requests needing to be filled. For 64 and 128 threads, this certainly worked to its detriment. Another challenge faced by the run with 128 threads is hardware limitations. With only 64 hardware threads, attempting to run more than that requires tricks that are beyond the scope of this thesis. It is sufficient to say that attempting to use more threads than the hardware has available results in diminished performance.

Work Sharing Performance				
Threads	Trial 1	Trial 2	Trial 3	**Average**
1	1:24:35	01:25:13	01:24:32	1:24:47
2	0:42:12	0:48:06	0:43:18	0:44:32
4	0:25:54	0:22:00	0:24:12	0:24:02
8	0:14:18	0:10:48	0:12:30	0:12:32
16	0:06:36	0:06:00	0:06:48	0:06:28
32	0:06:00	0:06:06	0:06:12	0:06:06
64	0:07:12	0:07:06	0:07:18	0:07:12
128	0:10:36	0:10:30	0:10:33	0:10:33

Figure 5.1 provides a visual of the average performance for each algorithm. The work sharing implementation consistently performs better until we get to 64 threads. At this point, they are near equal, and the original parallel solution actually performs better with 128 threads. This is evidence of the added overhead to more frequent calls to share work. Comparing the tables, it's also clear that the work sharing implementation has less consistent results. Work requests are offered and fulfilled in a different order each time the code is run. Sometimes larger work loads are among the first to be split up, which leads to a better use of processing power. Other times, work is spread by those who have less work to offer, which causes a chain of more requests. Without work sharing, each thread has the same amount of work each time it runs because there is no change to the initial allocation of vertices. This leads to

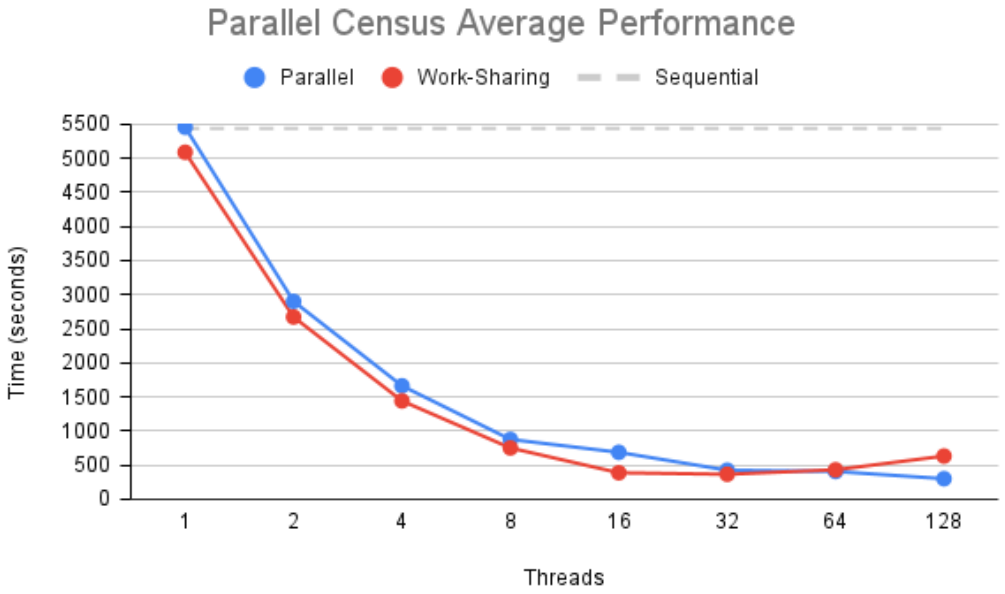


Figure 5.1: Performance Comparison

more consistent results.

Figure 5.2 shows how close each implementation is to the goal of linear speedups. Up to 8 threads, both implementations show near-linear speedups. This continues to 16 processors for the work sharing implementation. Once we go beyond 16 threads, both fall behind.

5.2 Furthering This Research

Through the course of this thesis, we have briefly discussed things that could be done to improve our algorithm. Now is the time for us to flesh out these ideas in more detail.

The first improvement mentioned is the addition of symmetry breaking conditions. For shapes with *automorphisms*—an isomorphic mapping onto itself—it helps to have something in place to prevent multiple discoveries. For example, take a look at Figure 5.3. Here we show motifs of size four with labeled vertices. When looking at a tailed triangle, we see that vertices 1 and 4 could be swapped and the same shape would still hold true. Similarly, in a four clique, there exist 4! ways to find the same clique. Ribeiro and Silva [5] implemented a solution in which, each point of swapping is given a comparison to eliminate multiple discoveries of the same path. They decided on sorting by vertex index. For example, the tailed triangle would have the condition $(2 < 3)$. This means that any search for a tailed triangle where $(2 > 3)$ is terminated early. In the case of the four clique, its conditions would be $\{(0 < 1), (0 < 2), (0 < 3), (1 < 2), (1 < 3), (2 < 3)\}$. Each node object with automorphisms would hold its own set of symmetry breaking conditions. Conditions

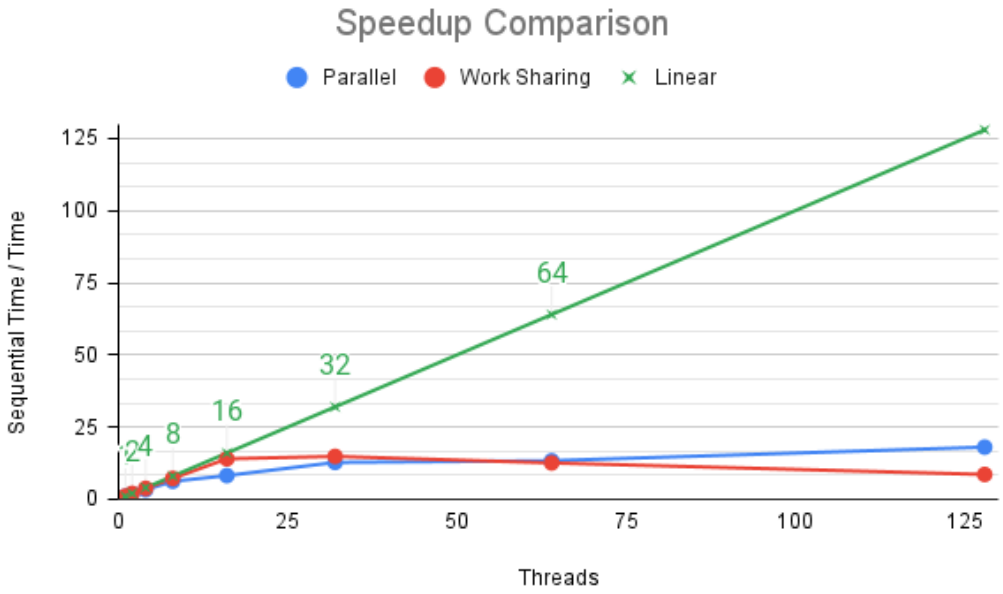


Figure 5.2: Speedup Comparison

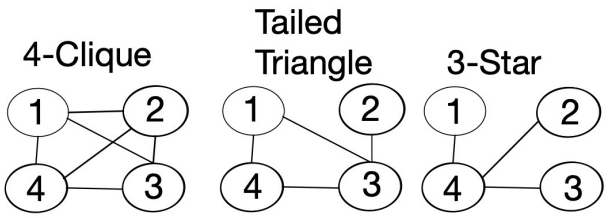


Figure 5.3: Motifs of size 4 with automorphisms

would be used to prevent the output of duplicate graphlets. They are also used to terminate further calls to `MatchingVertices`. On (line 3) of Algorithm 7, we see this in action. If the current path does not respect symmetry conditions, then it is no longer expanded, saving valuable time and computational resources.

Another area of improvement is the optimization of our work sharing implementation. A reason for the diminishing returns in our results is the frequent calls to share work. Currently, the first processor to see a pending work request is the first one to respond to it. This lead to a lot of splitting of work when there isn't much work to be split. The increased overhead of frequent search stops could be reduced by implementing a mechanism to determine if the current work unit contains enough work that it is worth the effort of sharing. If not, then it does not attempt to offer work.

The work sharing solution proposed by Ribeiro et al. [6], also implemented a time check that would reduce the number of checks for incoming work by a processor. When the program first starts, every processor is busy. As some processors begin to finish their initial workload, work requests grow in number and frequency. Ideally,

Algorithm 7 Symmetry Conditions

```
1: function MatchingVerticesConditions(x, p)
2:     M = ∅
3:     if p does not respect x.conditions then                    ▷ new
4:         return ∅                                               ▷ new
5:     C := CandidateCreator(p)
6:     forall v ∈ C do
7:         if G \ p ∼ x.graphlet then
8:             M := M ∪ {v}
9:     return M
```

this time check would be a function of the graph size and number of processors. It should also be able to dynamically adjust based on the frequency of work requests at a given time. A way to measure this is to increase the time whenever there is a check for work requests that returns false. A false return means that work was offered, but no one needed work in that moment. Similarly, we would decrease the timer when more requests come back true. Reducing the total number of checks helps decrease the overhead of sharing.

5.3 Conclusion

In conclusion, the speedups provided by parallelizing the initial algorithm are a testament to the power of parallel processing. Taking a problem that took over an hour to complete, and getting it to finish in under seven minutes is no small task. Even working on a personal computer (with less than 32 cores), parallelization allowed the runtime to decrease from ninety minutes to less than fifteen. In real world applications, this would allow for much faster turnaround times, allowing for increased productivity. Compared to individually searching for shapes of interest, g-tries proved to be a much faster solution. The combination of parallelization and graphlet census showed an ability to speed up a computation that has a wide variety of meaningful applications. Whether it be social networks, biological networks, or any application involving subgraph counting: parallel graphlet census has proven to be a valuable tool.

Bibliography

[1] David Aparício, Pedro Ribeiro, and Fernando Silva. Network comparison using directed graphlets. *arXiv preprint arXiv:1511.01964*, 2015.

[2] Fazle Elahi Faisal and Tijana Milenkovic. Dynamic networks reveal key players in aging. *arXiv e-prints*, pages arXiv–1307, 2013.

[3] Oleksii Kuchaiev, Po T Wang, Zoran Nenadic, and Natasa Przulj. Structure of brain functional networks. In *2009 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 4166–4170. IEEE, 2009.

[4] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[5] Pedro Ribeiro and Fernando Silva. G-tries: a data structure for storing and finding subgraphs. *Data Mining and Knowledge Discovery*, 28(2):337, 2014.

[6] Pedro Ribeiro, Fernando Silva, and Luis Lopes. Efficient parallel subgraph counting using g-tries. In *2010 IEEE International Conference on Cluster Computing*, pages 217–226, 2010.

[7] Kai Sun, Joana P Gonçalves, Chris Larminie, and Nataša Pržulj. Predicting disease associations via biological network analysis. *BMC bioinformatics*, 15(1): 1–13, 2014.

[8] Dongran Yu, Huaxing Zhao, Li-e Wang, Peng Liu, and Xianxian Li. A hierarchical k-anonymous technique of graphlet structural perception in social network publishing. In Éric Renault, Selma Boumerdassi, and Samia Bouzefrane, editors, *Mobile, Secure, and Programmable Networking*, pages 224–239, Cham, 2019. Springer International Publishing. ISBN 978-3-030-03101-5.