UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

DEPARTMENT OF INFORMATION ENGINEERING

MASTER'S DEGREE
IN COMPUTER ENGINEERING

An Experimental Evaluation of Triangle Counting Algorithms with Random Graphs in Hierarchical Memory Architectures

Supervisor:
Prof. Francesco Silvestri

Candidate:
Claudio Peron

Academic Year 2023/2024
July 3, 2024

Abstract

This thesis examines the performance of triangle counting algorithms in memory hierarchy systems when dealing with random graphs. The objective is to assess the effectiveness of cache-optimized algorithms in processing large-scale graph data compared to traditional algorithms designed for RAM.

The thesis initially presents and investigates the MinBucket algorithm both theoretically and experimentally, contrasting its performance with that of the trivial algorithm in power-law random graphs generated using the Erased Configuration Model. The experiments measure both execution time and the number of I/O operations on the CPU cache memory.

Subsequently, experimental comparisons are made between the MinBucket algorithm and a randomized cache-aware algorithm, utilizing power-law random ECM graphs, uniformly distributed random graphs, and real-world graphs, adjusting hyperparameters and comparing execution times and cache misses.

The experimental results demonstrate the superior performance of cache-aware algorithms over MinBucket when applied to power-law distributed graphs. This highlights the potential of memory hierarchy-optimized algorithms to significantly enhance computation speed compared to traditional algorithms for this category of graphs, suggesting a promising direction for future algorithm development.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and Motivation

In recent years, the explosion of data generated by various applications such as social networks, biological networks, and communication networks has sparked a growing interest in graph analytics. Graphs provide a powerful abstraction for modeling complex relationships between entities, enabling the representation and analysis of intricate networks. Among the various tasks in graph analysis, triangle counting holds a significant place due to its implications in various domains, including social network analysis, recommendation systems, and community detection.

A triangle in a graph consists of three vertices connected by three edges, forming a closed loop. The count of triangles in a graph provides crucial insights into its structure and connectivity patterns. For instance, in social networks, triangles represent clusters of interconnected individuals, indicating tight-knit communities or cliques. Understanding these triangular relationships can help uncover hidden patterns, identify influential nodes, and predict network behavior.

However, exact triangle counting in large-scale graphs poses significant computational challenges. Therefore, it's crucial to fully leverage both the inherent characteristics of computer architectures and the structures of the graphs themselves to achieve reasonable computing times. In this regard, one of the most crucial features of modern computer hardware architectures to exploit is the memory hierarchy.

The memory hierarchy consists of multiple levels of storage devices with varying access speeds and capacities, ranging from fast but small caches to slower but larger main memory and disk storage. The performance of algorithms can heavily depend on their ability to exploit the hierarchical nature of memory and minimize data movement between different levels. Traditional algorithms designed without considering memory hierarchy often suffer from poor cache utilization and excessive data transfers, leading to

suboptimal performances.

Motivated by these challenges, researchers have focused on developing triangle counting algorithms optimized for memory hierarchies. These algorithms aim to minimize memory access patterns, exploit spatial and temporal locality, and efficiently utilize cache resources to achieve better performance. In this thesis, we will compare a very simple algorithm for counting triangles in a graph with an algorithm designed to efficiently exploit the memory hierarchy, seeking the objective to assess whether the increased complexity of the latter algorithm implementation is justified. We will delve into the details of each algorithm and test their behavior on various types of graphs. Finally, we will compare the performance of the two algorithms, aiming to understand if efficient cache utilization can enhance triangle counting computation times.

1.2 Scope and Organization of the Thesis

The thesis will start by introducing fundamental concepts in graph theory and memory hierarchy models, to provide the necessary background for understanding subsequent discussions on triangle counting algorithms and cache aware optimizations.

Following the introduction, we will delve into the analysis of random graph generation algorithms, exploring various models such as Erdős-Rényi, Configuration Model, and Inhomogeneous Random Graphs (IRGs). We will pay special attention to power-law graphs and their generation algorithms, focusing on understanding their structural characteristics and their impact on the performance of triangle counting algorithms.

The core of the thesis will be dedicated to studying the behavior of the two triangle counting algorithms, especially when running on random graphs. We will experimentally evaluate their computation times and cache efficiency, comparing them to determine if an algorithm optimized for the memory hierarchy can outperform a common, simple algorithm designed for the RAM model.

Finally, we will conclude with a summary of key findings and future research directions. Our aim is to provide a comprehensive resource for researchers and practitioners interested in graph analytics, memory hierarchy optimization, and algorithm design for large-scale graph data.

Chapter 2

Preliminaries

2.1 Graph Theory Fundamentals

2.1.1 Definition of Graph

In graph theory, a graph G is a mathematical structure consisting of two sets: a set of vertices V and a set of edges E.

- The **vertices set** V is a collection of distinct objects called vertices (or nodes). These vertices represent the entities or points in the graph. Formally, V can be defined as $V = \{v_1, v_2, ..., v_n\}$, where n is the number of vertices in the graph, and each v_i represents a unique vertex.

- The **edge set** E is a collection of pairs of vertices, representing the connections or relationships between the vertices in the graph. An edge can be either directed or undirected, depending on whether the relationship between the vertices has a direction associated with it. Formally, E can be defined as $E = \{e_1, e_2, ..., e_m\}$, where m is the number of edges in the graph, and each e_i is a pair (u, v) representing a connection between vertices u and v.

Thus, a graph G can be formally represented as $G = (V, E)$, where V is the set of vertices and E is the set of edges.

2.1.2 Basic Definitions

- **Edge Incident to a Vertex**: An edge is said to be incident to a vertex if that vertex is one of the endpoints of the edge. In other words, if the edge connects two vertices, then it is incident to both of those vertices.

- **Self-Loop Edge**: An edge is said to be a self-loop edge if it connects a vertex to itself. More formally, a loop edge is an edge $(v, v) \in E$ with $v \in V$.
- **Parallel Edges**: Two edges in a graph are said to be parallel edges if they connect the same pair of vertices. Formally, Two edges $e_1, e_2 \in E$ are said to be parallel edges if, with $u, v \in V$, $e_1 = (u, v)$ and $e_2 = (u, v)$.
- **Adjacent Vertices**: Two vertices are said to be adjacent if they are connected by an edge. Formally, two vertices $u, v \in V$ are adjacent if and only if $\exists (u, v) \in E$.
- **Neighbors of a Vertex**: The neighbors of a vertex are the vertices that are directly connected to it by an edge. In an undirected graph, the neighbors of a vertex are the other vertices that share an edge with it. In a directed graph, the neighbors of a vertex depend on the direction of the edges: the outgoing neighbors are the vertices that can be reached from the given vertex, and the incoming neighbors are the vertices from which the given vertex can be reached. Mathematically, we can define the neighbors set of a vertex v as: $N(v) = \{u \mid (u, v) \in E\}$
- **Degree of a vertex:** The degree of a vertex in a graph represents the count of edges connected to that vertex. Formally, let $G = (V, E)$ be a graph, and let v be a vertex in V, the degree of v, denoted by $\deg(v)$, is defined as the number of edges incident to v. For a directed graph, the degree of a vertex includes both the in-degree (number of incoming edges) and the out-degree (number of outgoing edges) of the vertex.
- **Path:** A path (or walk) in a graph is a sequence of vertices in which each consecutive pair is connected by an edge. A path can not contain the same vertex more than once (unless it is at the same time the starting and ending vertex of the path), and edges can't repeat either. The **length of a path** is the number of edges it contains.
- **Cycle:** A cycle is a path that starts and ends at the same vertex.
- **Multigraph:** A multigraph is a graph where E contains self-loop edges or parallel edges.
- **Simple Graph:** A graph is simple if it does not contain self-loop edges nor parallel edges.
- **Connected Graph:** A graph is said to be connected if there is a path between every pair of vertices in the graph.
- **Disconnected Graph:** A graph is disconnected if there are at least two vertices in the graph that are not connected by any path.

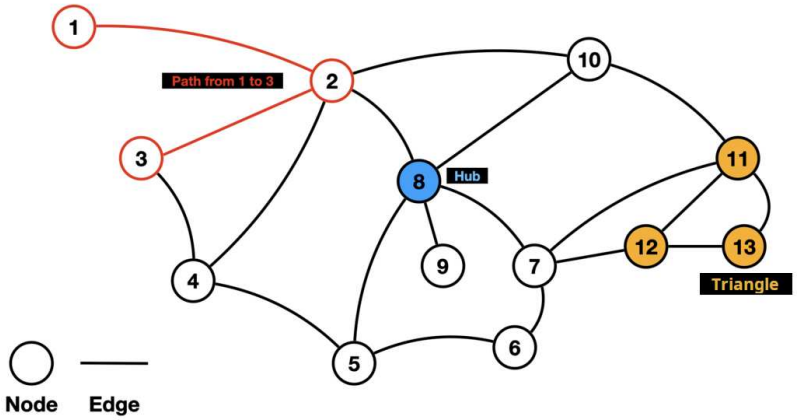

Figure 2.1: A very simple graph

- **Triangle:** In a graph, a triangle refers to a set of three vertices (u, v, w) such that there exists an edge between every pair of vertices in the set. We can also define a triangle as a cycle of length 3.

2.1.3 Graph Representation in Memory

In computer memory, vertices and their connections in a graph can be represented in various ways. Each method employs a distinct data structure, each with its own set of characteristics in terms of spatial requirements and time complexity for graph operations. In this section, we explore the most common methods, explaining their main advantages and disadvantages.

Adjacency Matrix

An adjacency matrix, denoted by $\mathbf{M}$, is a square two-dimensional array of size $\mathbf{n} \times \mathbf{n}$, where n represents the number of vertices (nodes) in the graph. Each element $\mathbf{M}[i][j]$ within the matrix corresponds to the relationship between vertex $\mathbf{i}$ and vertex $\mathbf{j}$. Here's how the value determines the connection:

- $\mathbf{M}[i][j] \neq 0$: This signifies the existence of an edge (connection) between vertex $\mathbf{i}$ and vertex $\mathbf{j}$. The specific value (often 1) depends from the application, for example in weighted graphs it can represent the weight of the edge.
- $\mathbf{M}[i][j] = 0$: Conversely, a value of 0 (or any other pre-defined value signifying absence) denotes the absence of an edge between vertex $\mathbf{i}$ and vertex $\mathbf{j}$.

The **advantages** of this kind of representation are easily understandable:

- **Constant-Time Edge Lookups**: Determining if there's a direct connection between two vertices, **i** and **j**, simply involves examining the corresponding cell $\mathbf{M}[i][j]$). This operation has constant time complexity ($\mathcal{O}(1)$), making it independent from the graph's size. This efficiency becomes particularly valuable when frequent queries about edge existence are required.

- **Suitability for Dense Graphs**: In scenarios where graphs exhibit high density, meaning that vertices possess connections with a large number of other vertices, adjacency matrices are the best choice for graph representation. In dense graphs, where the number of edges approaches the total potential connections, the spatial demand of the matrix does not result in significant space wastage. Additionally, the constant-time edge checks become even more advantageous for frequent queries on this kind of graphs.

On the other hand, these advantages may turn on **disadvantages** in other situations:

- **Space Inefficiency for Sparse Graphs**: The primary drawback of adjacency matrices arises from their space complexity. The size of the matrix grows quadratically ($\mathcal{O}(n^2)$) with the number of vertices (n) in the graph. This implies that even for moderately sized sparse graphs (where most vertices have few connections), the matrix can consume a considerable amount of memory. The inefficiency derives from the storage of numerous zero values in the matrix, representing non-existent edges.

- **Inefficient Neighbor Finding**: While checking for a specific edge is efficient, retrieving all the neighboring vertices (connected nodes) of a particular vertex can be time-consuming with an adjacency matrix. To achieve this, one would need to iterate through the entire row corresponding to that vertex, leading to linear time complexity ($\mathcal{O}(V)$). This can become a bottleneck for algorithms requiring frequent neighbor discovery.

Adjacency List

An alternative approach to graph representation is through the use of adjacency lists. The adjacency list data structure can be implemented using an array of pointers to adjacency lists: the size of the array is equal to the number of vertices (n) in the graph and each element in the array, denoted by $adjList[v]$, points to a list that stores the vertices adjacent to vertex v.
The main **advantages** of using an adjacency list representation are:

- **Space Efficiency for Sparse Graph**: A significant benefit of adjacency lists lies in their space complexity. Unlike adjacency matrices that grow quadratically with the number of vertices, adjacency lists only store actual connections. In sparse graphs, where most vertices have few connections, this translates to significant memory savings as the data structure avoids storing unnecessary zero values.
- **Efficient Neighbor Finding**: Retrieving all the neighboring vertices of a specific vertex is a strength of adjacency lists. By simply iterating through the adjacency list corresponding to that vertex, we can efficiently discover all its connected neighbors. This operation exhibits linear time complexity ($\mathcal{O}(deg[v])$), where $deg[v]$ represents the number of connections for vertex v.

However, these advantages come with the drawback that the **efficiency of edge existence checks** is compromised. Unlike adjacency matrices, which offer constant-time edge existence checks, adjacency lists require traversing the associated linked list to determine whether an edge exists between two vertices. In dense graphs, where most vertices are connected, this traversal can result in a much higher time complexity for edge existence checks.
Despite these limitations, adjacency lists remain a popular choice for graph representation, particularly in scenarios where memory efficiency is crucial, or when the graph is expected to be sparse.

Edge List

The edge list representation of a graph consists into a single list that contains all the edges present within it. This approach differs from adjacency matrices and adjacency lists, which focus on storing information around vertices.
An edge list is typically implemented as a two-dimensional array or as array of tuples, where each tuple represents a single edge. A single tuple typically contains two pieces of information:

- **Source Vertex**: This identifies the starting point of the edge, often denoted by a vertex identifier.
- **Target Vertex**: This identifies the ending point of the edge, also denoted by a vertex identifier.

Usually, in undirected graphs, the source vertex u and target vertex v are determined in the way that $u < v$. Furthermore, the edge list is often sorted lexicographically to enhance the efficiency of edge retrieval. Infact, if the edge list is ordered in such way, we can find edges much more quickly by using the binary search algorithm. On the other

hand, this representation suffers of very poor performances for **neighborhood retrieving**, which requires to iterate through the entire list.

Hash Table

Hash tables offer an alternative approach to graph representation, leveraging their ability to store key-value pairs for efficient retrieval. In the context of graph representation, a possible implementation can use an adjacency lists approach implementing an hash table instead of using a linked list or an array. Choosing a good hash function, this implementation can bring to multiple **advantages**:

- **Fast Average-Case Edge Lookups**: Retrieving the adjacency information for a specific vertex can be achieved in average constant time ($\mathcal{O}(1)$) due to the direct indexing capabilities of hash tables. This can be advantageous for operations that frequently check for edge existence.
- **Maintains Space Efficiency For Sparse Graphs**: As adjacency lists, it only store entries for vertices that actually have connections.

However, despite its theoretical superiority, the representation of graphs using hash tables hides practical difficulties. The efficiency of hash tables heavily relies on a good hash function and managing collisions effectively. In the worst case, with a poor hash function or excessive collisions (which are frequent in large graphs), lookup times can deteriorate in practise, making hash table slower than other well-implemented representation methods.

2.2 Power-law Distributed Graphs

If we analyze complex real world networks, we can often observe that the degrees of their vertices are not completely random, but they follow a **power-law distribution**. This distribution signifies a remarkable imbalance in the connectivity of nodes. In this section we will try to understand why real-world data often follows a power-law distribution, the nature of the distribution itself, and the key characteristics of graphs exhibiting this phenomenon.

2.2.1 Introduction to Power-law Graphs

The prevalence of power-law distributions in real-world networks suggests the existence of underlying mechanisms that favor the creation of highly connected nodes (often called

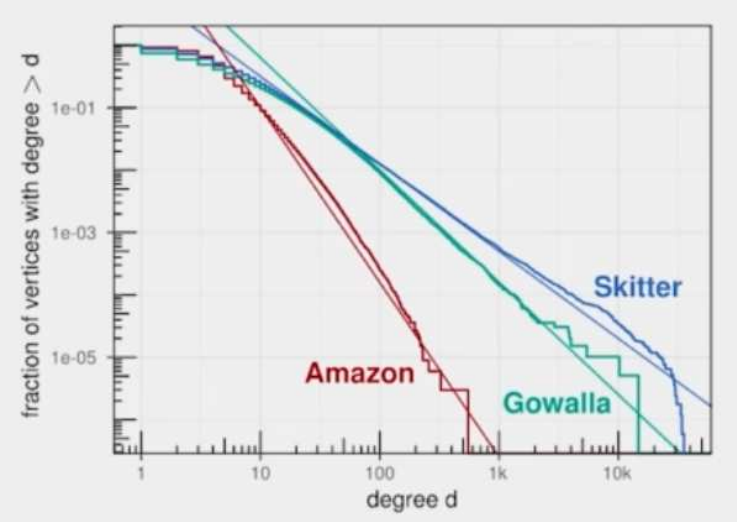

Figure 2.2: Degree distribution of real-word graphs in a log-log plot: the closeness to a straight line indicates a power-law distribution - Credits: Thomas Bläsius - Introductory Presentation to Geometric Inhomogeneous Random Graphs

”hubs”) compared to others. Several real-world phenomena can explain this typical behaviour:

- **Preferential Attachment:** This principle, also known as the ”rich-get-richer” phenomenon, implies that nodes with a high number of connections are more likely to attract new connections. Imagine a social network where popular individuals (hubs) tend to gain more followers than those with fewer connections. This self-reinforcing mechanism leads to a skewed distribution where a few nodes accumulate a disproportionate number of connections.
- **Growth and Evolution:** Many real-world networks exhibit continuous growth and evolution. New nodes are constantly added, and connections are formed based on existing connections. In such scenarios, preferential attachment naturally emerges. As the network grows, new nodes are more likely to connect to existing hubs, further reinforcing the power-law distribution.
- **Copying Mechanisms:** In some networks, new nodes may form connections by mimicking the connection patterns of existing nodes. This ”copying” behavior can amplify existing imbalances, leading to a power-law distribution.

These mechanisms, operating alone or in combination, can lead to the emergence of power-law distributions in real-world networks. Understanding these mechanisms is crucial aspect for designing efficient algorithms for network analysis, where we can exploit them to achieve better performances in practical usage.

2.2.2 Characteristics and Properties

Definition 2.1. The **probability density function** (PDF) of a power-law distribution is given by:

$$f_D(x) = c \cdot x^{-\tau}$$

where c is a normalization constant and τ is the **power law parameter**.

Definition 2.2. The **complementary cumulative distribution function** (CCDF) of a power-law distribution is computed from the PDF through integration:

$$F_D(x) = \int_x^{\infty} f_D(t)\, dt$$

Resulting in:

$$F_D(x) = e^{-b} \cdot x^{-\alpha}$$

Where $\alpha = \tau - 1$ and b is a constant.

Definition 2.2 tells us how likely it is for a number from a power-law distribution to be equal to or bigger than x. In simple terms, it shows the chance of seeing something as big as x or even bigger. For example, if $F(x) = 0.1$, it means there's a 10% chance of finding something with a size of x or larger.
In the context of power-law distributed graphs, we consider the degree of a vertex as a random variable following a power-law distribution. The power-law CCDF then quantifies the probability that this random variable exceeds or equals a given value x. In simpler terms, it indicates the probability of encountering a vertex with a degree equal to or greater than x.

Theorem 2.1. *Let X be the random variable denoting the degree of a vertex in a graph G, let τ be the parameter of the power-law degree distribution of G, let n be the number of vertices in the graph, then:*

$$E[X] \in \Theta(1) \text{ if } \tau > 2$$

$$E[X] \in \Theta(\log(n)) \text{ if } \tau = 2$$

$$E[X] \in \Theta(n^{2-\tau}) \text{ if } \tau < 2$$

Proof: *Let $x_m in$ the minimum value allowed from the power-law distribution. We will compute the expected value $E[X]$ using the PDF and integrate over the entire range.*

$$E[X] = \int_{x_{min}}^{\infty} x \cdot f(x)\, dx$$

$$E[X] = \int_{x_{min}}^{\infty} x^{1-\tau}\, dx$$

Performing the integration, we get what is stated by the theorem.

That means that the average degree of the vertices in the graph is finite only when $\tau > 2$. This is a typical value in real-world power-law graphs, thus the other cases are usually not of interest.

Theorem 2.2. *Let X be a random variable following a power-law distribution of parameter τ, let x_{min} the minimum value that X can assume, the variance Var$[X]$ is:*

$$Var[X] \in \Theta(1) \text{ if } \tau > 3$$

$$Var[X] \in \Theta(\log(n)) \text{ if } \tau = 3$$

$$Var[X] \in \Theta(n^{2-\tau}) \text{ if } \tau < 3$$

Proof: *To compute the variance of a power-law distribution, we first find the second moment, then subtract the square of the first moment (expected value). We notice that:*

$$Var[X] = E[X^2] - (E[X])^2 = \Theta(E[X^2])$$

The second moment is found computing the integral:

$$E[X^2] = \int_{x_{min}}^{\infty} x^2 \cdot f(x)\, dx = \int_{x_{min}}^{\infty} x^{2-\tau}\, dx$$

Performing the integration, we get what is stated by the theorem

Theorems 2.1 and 2.2 explain how a power-law graph behaves differently as τ changes. When τ is greater than 2, the average degree of the graph doesn't depend on the size of the graph (n). However, the variance of degrees in the graph becomes constant only when τ is greater than 3.

2.3 Random Graph Generation Algorithms

2.3.1 Introduction to Random Graphs

In general, many real-world networks, such as social networks (e.g., Facebook, Instagram, X), the internet and citation networks in scientific literature, exhibit characteristics that are neither purely deterministic nor entirely random. These networks often display properties like:

- **Small-world phenomenon**: In many real-world networks, even in large ones, most nodes can be reached from any other node through a relatively short path of connections.
- **Clustering**: Nodes tend to connect with others that share similar properties, leading to the formation of densely connected communities within the network.
- **Power-law distribution**: The degree distribution (number of connections per node) follows a power law, meaning a few nodes have a very high number of connections (hubs), while most nodes have relatively few.

Random graph models provide a way to capture these essential features in a mathematically tractable manner. By introducing randomness into the process of creating a graph, we can analyze the resulting network structure and its properties statistically. This allows us to compare real-world networks to these models, identifying deviations that might reveal underlying mechanisms or functionalities of the network. We can also use these models to test algorithms on graphs that mimic real-world data, also observing their behaviour varying the characteristics of the input networks. It this section we will explore some random graph generation models, delving into their functioning and their main characteristics.

2.3.2 The G(n,m) Model

The G(n,m) model is a classic approach to generating random graphs. In this model, both the number of vertices, denoted as n, and the number of edges, denoted as m, are fixed. The generation process involves incrementally adding m edges to n vertices, chosen uniformly at random from the set of all possible edges on n vertices. Algorithm 2.1 shows the typical implementation of the G(n,m) model: the graph generation process begins with a set of n isolated vertices. Edges are then added one by one until the desired number m is reached. To add an edge, two distinct vertices are selected uniformly at random from the set of n vertices, and an edge is added between them (if it does not already exist). This process continues until m edges have been added. Despite its simplicity, the G(n,m) model is a random graph generator that provides very important guarantees:

- **Fixed Number of Edges:** By fixing the number of edges m, researchers can control the graph's density and connectivity properties.
- **Simple Implementation:** The algorithm for generating G(n,m) graphs is straightforward.

Algorithm 2.1 G(n,m) Random Graph Generation

```
Initialize an empty graph G with n vertices
edges ← 0
while edges < m do
    Select two distinct vertices u and v uniformly at random from G
    if (u, v) is not an edge in G then
        Add edge (u, v) to G
        edges ← edges + 1
    end if
end while
return G
```

However, precisely because of its simplicity, fails into generate graphs that reflect the properties of the graphs that we can find in the real world, for several key reasons:

- **Uniform Edge Distribution**: As we discussed above, real-world networks often exhibit heterogeneity in the number of connections nodes have. Some nodes might be highly connected, while others have very few connections. The G(n,m) model, with its uniform edge probability, assigns an equal chance of connection to all node pairs.

- **Absence of Underlying Mechanisms**: Real-world networks often form due to specific processes or mechanisms. For example, social networks might see connections based on shared interests or geographical proximity. The G(n,m) model simply throws random edges, ignoring these underlying mechanisms that govern network formation.

- **No Control Over Degree Distribution**: The G(n,m) model doesn't offer direct control over the degree distribution (number of connections per node) of the generated graph. For this reason, we can not directly generate graphs with degree distributions that mimic real-world graphs.

2.3.3 The Erdős-Rényi Model

The Erdős-Rényi model is an other classic random graph generation approach. In this model, a graph is generated by independently adding each possible edge between n vertices with a fixed probability p.
Algorithm 2.2 shows the steps to generate an Erdős-Rényi graph. The graph generation process begins with a set of n isolated vertices. Then, each possible edge is added with a fixed probability p. This means that for each pair of vertices, an edge between them is included with probability p, independent of the other edges. The trivial implementation

is obviously $\Theta(n^2)$, but there exists implementations that lower the time complexity to $\Theta(n + m)$ [6].

Algorithm 2.2 Erdős-Rényi Model Random Graph Generation

```
Initialize an empty graph G with n vertices
for each pair of vertices u and v do
    Generate a random number r between 0 and 1
    if r ≤ p then
        Add edge (u, v) to G
    end if
end for
return G
```

While Erdős-Rényi can appear similar to the G(n, m) model, it is generally more used because of its additional advantages:

- **More flexibility**: In this model, each pair of vertices is connected with probability p, independently of the other pairs. This means you can specify the expected number of edges in the graph, rather than the exact number as in the $G(n, m)$ model.
- **Easier Analysis**: Differently from G(n,m) model, in this model each edge is added independently from previously added edges. This makes the resulting graph mathematically easier to analyze using the probability theory. This model is thus a more tractable model for theoretical purposes.

On the other hand, it still has the same criticalities of the G(n,m) model. As G(n,m) model, it can not capture real-world underlying mechanisms, even the simplest ones such as preferential attachment or community structure. Basically, it is like trying to understand how people make friends by randomly assigning friendships rather than considering factors like common interests or mutual friends. Consequently, while the G(n,p) model provides a good basic framework for understanding certain aspects of network behavior, it doesn't fully capture the complexities we often see in real-world networks. In the next sections we will see more complex algorithms that can mimic certain properties of real-world networks, such as the degree distribution and the tendency to form clusters between similar nodes.

2.3.4 The Configuration Model

While G(n,m) and G(n,p) models do not provide any control over the generated graph structure, the Configuration Model (CM) is a random graph generation approach that

allows to specify the graph's degree sequence in input. In other words, given a specific degree sequence, it generates a random graph having exactly that degree sequence. Algorithm 2.3 shows the functioning of the configuration model. Given a desired degree sequence, the algorithm creates for each vertex as many "stubs" as specified on its degree. The algorithm then randomly matches pairs of stubs to form edges. This process ensures that the resulting graph has the specified degree sequence.

Algorithm 2.3 Configuration Model Random Graph Generation

```
Specify the desired degree sequence d_1, d_2, ..., d_n
Initialize an empty graph G
for each vertex v with degree d_v do
    Create d_v stubs for vertex v
end for
while there are unmatched stubs do
    Randomly select two unmatched stubs and add an edge between them
end while
return G
```

The Configuration Model possesses important advantages over the models seen above:

- **Control over degree distribution**: The model allows for the specification of a desired degree sequence, which can be sampled from any probability distribution model.
- **Realism**: We can generate graphs that mimic real-world networks characteristics by giving in input a realistic degree sequence. For example, we can sample it from a power-law probability distribution (as we seen in sections above) or just copy it from a real-world graph.
- **Easy to implement**: The configuration model can be easily implemented by generating an array of stubs and randomly shuffle them, forming edges by coupling stubs in the order they appear. This algorithm is straightforward and its complexity is $\Theta(m)$.

It is important to highlight that the configuration model produces in output a **multigraph**. This means that it is not guaranteed that the graph will not include self-loops or redundant arcs. This may be seen as an advantage, since many real-world graphs are actually multigraphs, but it can be also useless or even a complication in certain contexts. For example, when counting triangles in a graph, redundant arcs may produce multiple counts of the same triangle, while self-loops are simply irrelevant. The configuration model can be modified by adding a post-processing phase that removes all

redundant arcs and self-loops from the graph. In that case, the model takes the name of **Erased Configuration Model**. Note that the probability of generating a self-loop or a redundant arc using the configuration model decreases when m increases, thus an erased configuration model graph may be very similar or even equal to the not-erased graph when m is large.
In this thesis we will extensively use erased configuration model to generate large random power-law graphs, to mimic real-world networks and test the triangle counting algorithms.

2.3.5 The Chung-Lu Model

In the Chung-Lu random graph model, proposed by F.Chung and L.Lu [3], the degree sequence is not a strict constraint for the output graph, unlike the configuration model. Instead, the degree distribution $d_1, d_2, \ldots, d_n$ serves as a sequence of weights that determine the probability for each edge to appear in the graph. For each pair of vertices i and j, an edge is added between them with probability p_{ij}, where p_{ij} is calculated as:

$$p_{ij} = \frac{d_i \cdot d_j}{2m}$$

Here, d_i and d_j are the degrees of vertices i and j respectively, and m is the total number of edges in the graph.
One can easily show that the degree assigned to each vertex in the degree sequence corresponds to the expectation of the actual degree in the output graph:

Theorem 2.3. *Let $G = (V, E)$ be a graph and let $v \in V$ be a vertex in the graph. Let d_v be its degree specified in the degree sequence given as input to the Chung-Lu model. Let X_v be the random variable denoting the actual degree of v in the graph generated by the Chung-Lu model. Then, it holds that:*

$$E[X_v] = d_v$$

Algorithm 2.4 shows a straightforward implementation of the Chung-Lu generation process. The proposed algorithm is clearly $\Theta(n^2)$, but there exists more sophisticated implementations that lower the complexity to $\Theta(n + m)$ in expectation [6].
Graphs generated by the Chung-Lu model has several differences between graphs generated by the erased configuration model, the most relevant are:

- **Degree Expectation**: The expectation of the degree of a vertex is equal to its degree given in input to the model. This is not true for erased configuration model

Algorithm 2.4 Chung-Lu Random Graphs Generation

Specify the desired degree distribution $d_1, d_2, ..., d_n$
Initialize an empty graph G
for each pair of vertices i and j **do**
    Calculate the edge probability $p_{ij} = \frac{d_i \cdot d_j}{2m}$ using the degree sequence
    Generate a random number r between 0 and 1
    if $r \leq p_{ij}$ **then**
        Add edge (i, j) to G
    end if
end for
return G

where the erasing process can modify the graph's structure, especially for small values of m.

- **Easier Analysis**: The probability of an edge to appear in the graph is independent from the presence of the other edges, which is not true for the configuration model. This makes the theoretical analysis of the model much easier.

Despite his theoretical superiority, the model hides some practical problems:

- **Soft Constraints**: The output graph will have the same degree sequence specified in input only in expectation. This means that to make a good experimental analysis we need to generate more graphs to obtain reliable results.
- **Higher Complexity**: Compared to the configuration model, the Chung-Lu model has an higher complexity. This may be a problem when we want to generate very large graphs.

2.3.6 Geometric Inhomogeneous Random Graphs

The models we have seen so far only focus on the number of connections each node has, ignoring everything else that shapes real networks. Imagine we want to make a random graph that looks like a social network. Using those models would be like randomly assigning friends to each person, just based on some probability. But in real social networks, connections are influenced by other factors like where people live, what they are interested in, and so on. Consequently, it is more likely that connections happen between nodes that are close in some way, like geographically or based on shared interests. Geometric Inhomogeneous Random Graphs (GIRGs) resolve this problem by incorporating spatial information into the generation process. In GIRGs, vertices are randomly placed in a metric space, such as Euclidean space, according to a spatial distribution. The spatial distribution may reflect underlying characteristics of the system

being modeled, such as geographical locations in social networks or physical distances in sensor networks.
During the edges generation process, vertices that are close in the metric space have a higher probability of being connected, which is also conditioned by the specified degree for the vertices. In particular, given n vertices placed randomly in a d-dimensional space, given a degree sequence $w_1, w_2, ..., w_n$ and a constant a (that regulates the average degree), an edge (u, v) is formed iff:

$$\text{dist}(u, v)^d \leq \frac{a \cdot w_u \cdot w_v}{n}$$

This condition is expressed as:

$$\Pr[(u, v) \in E | w_u, w_v] = \Pr\left[\text{dist}(u, v)^d \leq \frac{a \cdot w_u \cdot w_v}{n}\right]$$

However, this formulation is just a threshold case. Infact, it is useful to introduce an additional parameter T (where $T \in (0, 1)$), called "temperature", to interpolate between high and low locality effect. The probability $\Pr_{u,v}$ of forming an edge between vertices u and v is then actually defined as:

$$\Pr_{u,v} = \min\left\{1, \left(\frac{1}{\text{dist}(u, v)^d} \cdot \frac{a \cdot w_u \cdot w_v}{n}\right)^{1/T}\right\}$$

Note that for $T = 0$ we obtain the threshold case seen above.

Using this definition, a straightforward implementation of the algorithm is very similar to the Chung-Lu algorithm (refer to Algorithm 2.4), but also in this case, it is available a more efficient implementation that generates a GIRG in expected linear time [2].

From the practical point of view, using Geometric Inhomogeneous Random Graphs rather simpler degree-distribution-based models has many advantages on the resulting network realism, which can be summarized as follows:

- **Incorporation of Spatial Information:** GIRGs incorporate spatial information into the graph generation process, allowing for the modeling of networks where proximity plays a significant role in edge formation.

- **Heterogeneous Connectivity:** By combining spatial proximity with specified degree distributions, GIRGs generate graphs with heterogeneous connectivity patterns. Some regions of the graph may exhibit dense connectivity, while others may be sparser, reflecting the underlying spatial structure.

- **Realism in Complex Systems:** GIRGs are particularly suitable for modeling complex systems where spatial proximity influences connectivity, such as social networks, sensor networks, and biological networks.

On the other hand, GIRGs introduce dependencies on various factors beyond mere degree distribution. Parameters such as the metric space and temperature play crucial roles in shaping the resulting graph structure and, while this complexity allows for more realistic modeling, it also complicates the analysis process, both theoretically and experimentally.

2.4 Memory Hierarchy Model

The memory hierarchy is a crucial aspect of modern computing systems, designed to optimize data access and storage by organizing memory into multiple levels with varying speeds, capacities, and costs. This hierarchical structure aims to minimize the latency of memory accesses while maximizing throughput, improving overall system performance. In this section, we will provide an overview of the memory hierarchy model, discuss its implementation in modern computing systems, and explore the categories of algorithms designed to efficiently manage data within this hierarchy.

2.4.1 Overview of Memory Hierarchy

The memory hierarchy model is a memory architecture that consists of multiple levels of memory, each with different characteristics and access times. The primary goal of the hierarchy is to exploit the principle of locality, which states that programs tend to access a small portion of their data and instructions repeatedly, either spatially (within a small region of memory) or temporally (over a short period of time). By organizing memory into hierarchical levels, the system aims to optimize memory access for both spatial and temporal locality.
At the top of the memory hierarchy are registers, which are small, fast storage locations directly accessible by the CPU. Registers typically hold data and instructions currently being processed by the CPU, providing the fastest access times but limited capacity. Below registers are various levels of cache memory, usually Level-1 (L1), Level-2 (L2) and Level-3 (L3) caches, which are small, high-speed memory units located closer to the CPU than main memory (RAM). Caches store copies of frequently accessed data and instructions, exploiting temporal locality to reduce the latency of memory accesses. Further down the hierarchy is the main memory, which serves as the primary storage for program data and instructions. While main memory offers larger capacities compared to

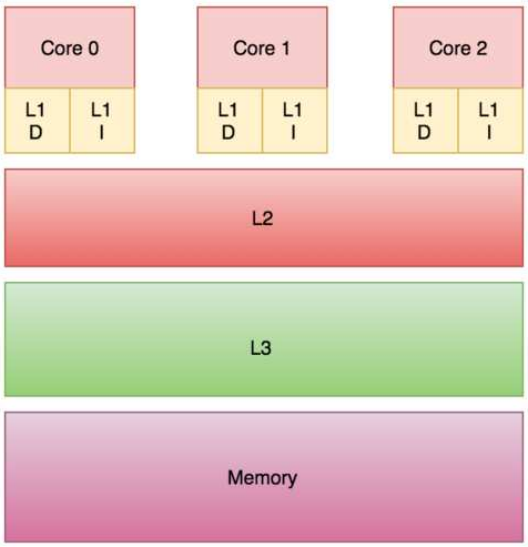

Figure 2.3: Typical memory hierarchy architecture in a multi-core CPU system

caches, it has much higher access times, making it slower to access data.
Finally, at the bottom of the hierarchy are secondary storage devices such as hard disk drives (HDDs) and solid-state drives (SSDs). These devices provide large storage capacities but have much slower access times compared to caches and main memory.
Modern computing systems employ sophisticated memory hierarchies tailored to meet the demands of increasingly complex applications and workloads. These systems often feature multiple levels of cache memory (usually 3 levels), with each level designed to balance between access times, capacities, and costs.
Multicore CPUs often feature distinct L1 caches for instructions and data per core, where L1 data caches typically prioritize speed over capacity, ensuring faster access times compared to L1 instruction caches. Moreover, these CPUs commonly integrate larger shared L2 and L3 caches across cores, to further improve cache hit rates and reduce memory latency.
Beyond caches, modern systems may also incorporate techniques such as prefetching, where the CPU predicts and fetches data from main memory or secondary storage before it is actually needed, to reduce memory access latency.

2.4.2 Algorithms For Memory Hierarchy

Efficient management of data within the memory hierarchy requires specialized algorithms designed to optimize memory access patterns and minimize cache misses. The two main categories of algorithms for memory hierarchy management are cache-aware and cache-oblivious algorithms, each with distinct characteristics and

approaches.

Cache-Aware Algorithms

Cache-aware algorithms are designed with explicit knowledge of the cache architecture and parameters, such as cache size and line size. These algorithms aim to maximize cache utilization and minimize cache misses by taking advantage of the cache's properties and optimizing data access patterns accordingly.
One common strategy employed by cache-aware algorithms is to organize data structures and access patterns to exploit spatial and temporal locality, thereby improving cache hit rates. For example, algorithms may reorder data accesses to improve temporal locality, ensuring that frequently accessed data remains in the cache for faster access.
Additionally, cache-aware algorithms may utilize techniques such as cache blocking or loop tiling to enhance cache efficiency. By partitioning data into blocks that fit within the cache and optimizing loop iterations to operate on these blocks, these algorithms can reduce cache thrashing and improve data reuse within the cache.
While cache-aware algorithms can achieve high performance on specific cache configurations, they may be less portable and may require manual tuning to optimize for different cache architectures.

Cache-Oblivious Algorithms

Cache-oblivious algorithms, in contrast, are designed to perform efficiently across a wide range of cache configurations without explicit knowledge of the cache parameters. These algorithms aim to achieve optimal performance by adaptively adjusting their behavior based on the available memory hierarchy without relying on cache-specific optimizations.
One key characteristic of cache-oblivious algorithms is their recursive or hierarchical structure, which allows them to exploit locality of reference at multiple levels of the memory hierarchy. By recursively dividing the problem into smaller subproblems, cache-oblivious algorithms can naturally exploit spatial and temporal locality, resulting in efficient cache utilization.
A common example of a cache-oblivious algorithm is the cache-oblivious matrix multiplication algorithm. This algorithm recursively partitions matrices into smaller submatrices, performing matrix multiplication on each submatrix and combining the results to obtain the final result. By carefully managing data accesses and recursive subdivisions, the algorithm optimizes cache performance without relying on cache-specific optimizations.

Cache-oblivious algorithms offer the advantage of portability and ease of implementation, as they can adapt to different cache architectures without requiring manual tuning. However, they may not always achieve the same level of performance as cache-aware algorithms optimized for specific cache configurations.

Chapter 3

Triangle Counting Algorithms

3.1 Introduction to Triangle Counting

The triangle enumeration problem is a fundamental task in graph theory, involving the identification and listing of all triangles within a given graph. A triangle, in graph theory, is defined as a set of three vertices interconnected by edges, forming a cycle of length 3. This problem is of particular interest because triangles represent tight relationships between vertices, indicating strong connectivity patterns within the graph.

The triangle counting problem shares similarities with enumeration but with a distinct focus: instead of listing every triangle, the objective is simply to determine the total count of triangles present within the graph. This task involves determining the number of triangles without necessarily listing them explicitly.

Triangle counting algorithms play a crucial role in graph analysis due to their wide range of applications. One common application is community detection, where the goal is to identify groups of vertices that exhibit dense internal connections and, since triangles represent strong local connectivity, communities within a graph are expected to contain a significant number of triangles. By leveraging triangle counting, analysts can effectively detect and delineate these communities.

Moreover, the number of triangles in a graph is particularly used to measure the clustering tendency of the vertices. A common metric is the clustering coefficient, which measures the degree to which nodes tend to cluster together. High clustering coefficients indicate dense local neighborhoods, often characterized by the presence of numerous triangles. Thus, by computing the total number of triangles in a graph, analysts can better understand the network's overall clustering behavior and identify regions of high connectivity.

Given the significance of triangle counting in graph analysis, there is substantial interest in finding fast methods to compute it. Faster triangle counting not only speeds up the

process itself but also accelerates other tasks that rely on triangle counting, such as clustering coefficient calculation.
In this chapter, we will explore the triangle counting algorithms utilized in my research training activity, delving into their functioning and characteristics. Subsequently, we will investigate how the worst-case performances of these algorithms can be mitigated when applied to power-law graphs. Finally, we will see how an algorithm designed for execution in a memory hierarchy architecture can enhance the speed of the triangle counting process in practice.

3.2 Trivial Triangle Counting

The first, trivial algorithm that we can build for counting triangles in a graph is a straightforward approach that uses directly the definition of triangle in graph theory. The essence of the algorithm lies in simply examining all possible paths of length 2 within the graph and checking whether they are closed by a third edge, forming a triangle. Given $G = (E, V)$, Pseudocode 3.1 shows the trivial algorithm functioning.

Algorithm 3.1 Trivial Triangle Counting

```
1: for each vertex v ∈ V do
2:     for each couple of neighbors u, w ∈ N(v) do
3:         if (u, w) ∈ E then
4:             Count triangle (v, u, w)
5:         end if
6:     end for
7: end for
```

Theorem 3.1. *Let $G = (V, E)$, $n = |V|$, $m = |E|$. Let $d_1, d_2, ..., d_n$ be the degree sequence of the vertices in V. The trivial algorithm has a running time:*

$$\Theta(\sum_{v=1}^{n} d_v^2)$$

Proof: *For each vertex $v \in V$, the algorithm involves taking pairs of neighbors of v to check if there exists an edge between them. Since the degree of v is d_v, this requires examining $\binom{d_v}{2}$ pairs of vertices, which is $\Theta(d_v^2)$. Since we have to repeat this process for each vertex v, the total time complexity is $\Theta(\sum_{v=1}^{n} d_v^2)$.*

The complexity provided by Theorem 3.1 can degrade to $O(n^3)$ in a worst-case scenario, particularly when the graph is complete or close to be complete.

The main inefficiency of this algorithm arises from the fact that, during execution, it enumerates each triangle present in the graph three times. Indeed, it is evident that we enumerate all the P_2s (paths of length 2) starting from each vertex in the graph, and since each cycle of length 3 has three possible starting vertices, each triangle is enumerated three times.
However, despite the trivial algorithm not being the most efficient in terms of theoretical time complexity, its simplicity makes it an excellent starting point for developing better algorithms. In fact, through the analysis of this simple algorithm, we can discover some important characteristics:

- **Dependency from the degree sequence**: Despite the bad performances in a worst-case scenario, the actual time complexity of the trivial algorithm is dependent from the degree sequence of the graph. This means that the actual complexity can be much lower than the worst case scenario in some cases. This aspect will be studied in depth in the next sections.

- **Highly parallelizable**: It is evident that this algorithm can be readily parallelized by subdividing each P_2 enumeration task based on the starting vertex. The ability to parallelize the process easily can significantly enhance computation times, despite the algorithm's poor theoretical performance. This can potentially make it faster than superior algorithms on paper, although non-parallelizable.

Knowing these considerable advantages, we can try to improve the trivial algorithm, trying to mitigate its criticalities while maintaining its positive aspects. A notable improvement of this algorithm in this sense is represented by the MinBucket algorithm.

3.3 MinBucket Algorithm

The MinBucket algorithm can be viewed as an enhancement of the trivial algorithm. The name originates from the paper "Why do simple algorithms for triangle enumeration work in the real world?" by Berry et al. [1], which forms the foundation for the theoretical analysis of both the trivial and MinBucket algorithms, but it is known also with other names (e.g `NodeIterator++`). These analyses are presented in the following sections of this thesis, and they served as the basis for my research training work. As we have seen in the section above, the trivial algorithm suffers of the fact that it produces multiple enumerations of the same triangle. This is clearly an inefficient behaviour that can not be avoided without computing additional structures to trace whether a triangle has already been enumerated or not. A simple example is given by Example 3.2.

Example 3.2. *let $G = (V, E)$ be a graph consisting only in a triangle of three vertices, means that $V = \{v, u, w\}$ and $E = \{(v, u), (v, w), (u, w)\}$.*
Looking at Algorithm 3.1 it is easy to see that the algorithm counts 3 triangles:

$$(v, u, w), (u, w, v), (w, v, u)$$

Note that, in order to count the triangle only once, it would be enough to choose arbitrarily one vertex as start for the trivial algorithm, and ignore the other ones. Thus, multiple enumerations can be avoided if, using some heuristic, we choose only one of these vertices to enumerate all the P_2s using the trivial algorithm.
Since the P_2s enumeration task has complexity $\Theta(d_v^2)$, where d_v is the starting vertex degree, and since we can choose any of the three vertices of the triangle, it is reasonable to choose the vertex having the lowest degree. The MinBucket algorithm consists exactly into applying this heuristic to the trivial algorithm.
From the practical point of view, the functioning of the MinBucket algorithm is described by Algorithm 3.2.

Algorithm 3.2 MinBucket Algorithm

```
1:  Create n empty buckets B_1, B_2, ..., B_n
2:  for each edge (u, v) ∈ E do
3:      if d_u ≤ d_v then
4:          Place v in B_u
5:      else
6:          Place u in B_v
7:      end if
8:  end for
9:  for each vertex v do
10:     Enumerate all P_2 formed by vertices in B_v, and output the closed ones
11: end for
```

To better understand what Algorithm 3.2 algorithm does, we can refer to Figure 3.1. The algorithm formally involves re-computing the adjacency lists of vertices in the graph, denoted as $B_1, B_2, \ldots, B_n$, where each list B_v includes only the neighbors of vertex v that have a higher degree compared to vertex v. The result is a transformation of the graph into a directed graph, where every edge starts from the vertex with the lowest degree and ends at a vertex with a higher degree.
It is important to note that in this transformed graph, within each triangle, we always have one vertex that has no outgoing edges, and it always corresponds to the triangle's vertex with the highest degree. Note also that, in such triangle, the trivial algorithm can detect the triangle only by starting from the vertex having the lowest degree, which is exactly the improvement that we aimed to obtain.

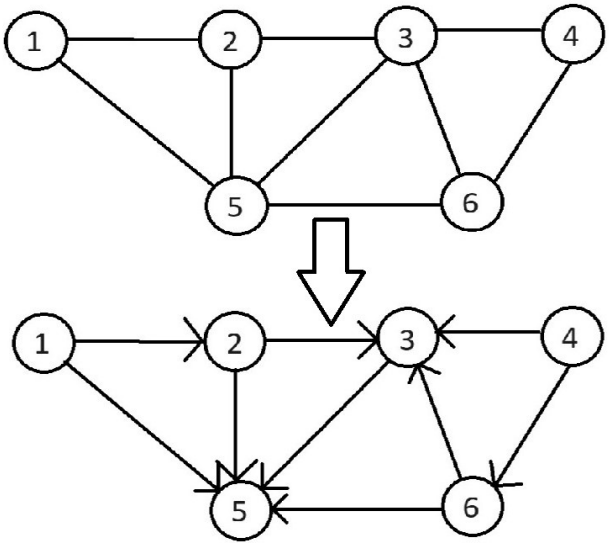

Figure 3.1: Graphical Representation of MinBucket

The algorithm requires an additional $\Theta(m)$ time to compute buckets and $\Theta(m)$ additional space to store them. We can easily observe that, in a worst-case scenario (e.g. large dense bipartite graphs), the computational time of this algorithm is still impracticable. However, it has been shown experimentally that this algorithm beats trivial algorithm by a constant factor of more than 100 in common datasets [8]. The explanation of this paradox lies in the dependency of the MinBucket algorithm performances on the degree sequence of the graph. As we have seen in Chapter 2, real-world networks differ significantly from the worst-case scenario, as they are typically very sparse, with a degree sequence that often follows a power-law distribution. In the next section, we will provide a theoretical explanation of this phenomenon, demonstrating why this simple algorithm can perform very well when operating on real-world networks.

3.4 Triangle Counting In Power-law Graphs

We have seen in the previous section that MinBucket algorithm can count triangles in a graph much faster than trivial algorithm in common datasets, despite its very poor theoretical upper bound. The reader might therefore ask if we can, given certain conditions, obtain a theoretical explanation of this results, and prove mathematically the superiority of MinBucket algorithm in real-world networks. The answer comes from the work of Berry et. al [1], that studied theoretically the behaviour of trivial and MinBucket algorithms under the assumption of using random graphs that reflects the characteristics of real-world networks. My research training has been centered on studying the bounds

established by these theorems and experimentally verifying their results. The outcomes of these experiments are detailed in the Results chapter.
From the theoretical point of view, the time performance of a triangle counting algorithm can be summarized by the number of P_2s (also called "wedges") enumerated by it. Let's first focus in detail on the number of wedges enumerated by the MinBucket algorithm. By looking Algorithm 3.2, we know that, once we have computed all the buckets, we have to enumerate all the wedges within each bucket. This in practical terms consists into enumerating all the possible couples of the vertices that are in that bucket.

Definition 3.1. Let X_v be the size of the bucket B_v, then the **Bucket Work** of B_v is defined as:

$$BW(v) = \binom{X_v}{2} = \frac{X_v(X_v - 1)}{2}$$

Definition 3.2. Let $G = (V, E)$ be a graph, let $n = |V|$, let X_v be the size of the bucket B_v, then the **Total Work** of MinBucket algorithm for counting triangles in graph G is defined as:

$$TW(G) = \sum_{v=1}^{n} BW(v)$$

The total work represents the number of wedges enumerated by the algorithm. These definitions can be readily adapted for the trivial algorithm, wherein the bucket sizes coincide with the degrees of the vertices.
It's important to note that in MinBucket each bucket size, denoted as X_v, is upper-bounded by the degree of the corresponding vertex, d_v. Consequently, it's evident that the total work is in some way dependent on the degrees of the vertices of the graph. Note also that $TW(G)$ can quickly escalate if there are "too many" large buckets. Fortunately, as discussed in the Preliminaries, real-world data often follows a power-law distribution, where only a few nodes have high degrees while the vast majority of vertices have low degrees.
To study the total work more formally, we require a random graph model capable of mathematically describing a graph that mirrors a specific degree sequence. Berry et al. [1] selected the **Erased Configuration Model** for their research. The main **results** of their work are summarized by the following theorems:

Theorem 3.3. *Consider a degree sequence $d = (d_1, d_2, \ldots, d_n)$ for its vertices, with maximum degree $d_m < \sqrt{m}/2$, then the total work of MinBucket algorithm over the random graph ECM(d) is:*

$$TW(G) \in O(n + m^{-2}(E[d_v^{4/3}]^3))$$

Theorem 3.3 shows how the total work upper bound of MinBucket algorithm heavily depends on the 4/3-moment of the degrees probability distribution. When this moment is high, the total work tends to escalate cubically. Note that, in these terms, the trivial algorithm has an $O(nE[d_v^2])$ upper bound. However, in the context of triangle counting, we are mostly interested in power-law distributed degree sequences. So how these algorithms perform over power-law distributed graphs? The answer is provided by Theorem 3.4

Theorem 3.4. *Consider a power law degree graph with exponent τ, and maximum degree d_m. Then the expected number of P_2s enumerated over this graph are:*

- $O(n + nd_m^{7-3\tau})$ *for MinBucket algorithm*
- $O(n + nd_m^{3-\tau})$ *for trivial algorithm*

Theorem 3.4 finally shows us that the MinBucket algorithm is superior to the trivial algorithm for a class of power-law distributed graphs. The MinBucket algorithm outperforms the trivial algorithm only when $\tau > 2$, and it becomes linear when $\tau > 7/3$ (approximately 2.33). Note that the trivial algorithm becomes linear when $\tau > 3$. In other words, this result tells us that we have the maximum advantage using the MinBucket algorithm when $2 < \tau < 3$. For $\tau < 2$ the MinBucket algorithm performs worse, while for $\tau \gg 3$ the two algorithms can be considered mostly equivalent. Fortunately, the majority of real-world networks have $2 < \tau < 3$, which explains why the MinBucket algorithm beats the trivial algorithm in practice.
Berry et al. [1] also provided another theorem that further lowers the upper bound when the expected value and the 4/3-moment of the degree distribution are finite. The statement is reported in Theorem 3.5.

Theorem 3.5. *(Constant factors in MinBucket total work): Let X_v be the random variable indicating the size of the bucket B_v in a graph of n nodes. Fix any n and a degree distribution D such that $E[d]$ and $E[d^{4/3}]$ are finite. Then the total work of MinBucket algorithm is $O(nC)$, where:*

$$C \equiv \lim_{n\to\infty} \frac{1}{n} E\left[\sum_{v=1}^{n} \binom{X_v}{2}\right] = \frac{1}{2(E[d])^2} \sum_{t_1=1}^{\infty} \sum_{t_2=t_1}^{\infty} \sum_{t_3=t_1}^{\infty} t_1(t_1-1)t_2t_3 f(t_1)f(t_2)f(t_3) \in (0,\infty).$$

Lemma 3.6. *Let C be the constant defined by Theorem 3.5, then it holds that:*

$$C \in O((E[d_v^{4/3}]^3))$$

In other words, Theorem 3.5 provides a tighter bound for the total work of the MinBucket algorithm when $\tau > 7/3$, essentially bounding the constant factors present in

the total work. Lemma 3.6 proves that this bound is upper-bounded by the bound provided by Theorem 3.3.
It is worth noting that Berry et al. also proved that all the theorems presented above hold true when MinBucket is run on a random graph generated using the Chung-Lu model. This demonstrates the equivalence of the two models in terms of MinBucket performance.

3.5 Triangle Counting in Memory Hierarchy

In the sections above we have seen the theoretical and practical superiority of the MinBucket algorithm for counting triangles in a power-law distributed graph. For this reason, the MinBucket algorithm is one of the most commonly used algorithms when an exact count of triangles is needed in real-world networks. We have seen the work provided by Berry et al. [1] that gives a theoretical bound for the number of wedges enumerated by the algorithm and, consequentially, the theoretical bound in terms of computation time. Now, what direction should we take to further improve the algorithm performances in practise, and lower down the execution time? We have seen in the Preliminaries how a wise cache usage can strongly improve in practise an algorithm performances. Two algorithms sharing the same theoretical bound in terms of computation time may have completely different execution times in practise due the different usage of the cache memory. Thus, it is important to try to minimize not only the time complexity, but also the **I/O complexity** of the cache memory.
The MinBucket algorithm is not designed to exploit the cache memory, we can notice it directly from the functioning of the algorithm itself (Algorithm 3.2). For each bucket B_v, the algorithm iterates within it by taking all the possible couples of vertices in the bucket and it check if there exists a closing edge between them. If all the edges do not fit in the cache memory, the algorithm causes frequent cache-misses during the iterations, significantly lengthening data access times.
Thus, can we design a triangle counting algorithm that takes advantage of cache memory? The answer comes from Pagh and Silvestri [7], that proposed a randomized cache-aware algorithm that exactly counts the triangles in a graph, exploiting the cache memory.
Before delving into the characteristics of this algorithm, we need to add some specific **preliminaries**:

Theorem 3.7. *[9] Let $G = (V, E)$. The I/O complexity of sorting the edge set $sort(E)$ is*

$$sort(E) = O\left(\frac{E \log(E/B)}{B \log M} + \frac{E}{B}\right)$$

Definition 3.3. Let $G = (V, E)$ and $v_1, v_2, v_3 \in V$. For a triangle $\{v_1, v_2, v_3\}$, with $v_1 < v_2 < v_3$, we call the edge $\{v_2, v_3\}$ its *pivot edge*, and the vertex v_1 its *cone vertex*.

Moreover, this algorithm makes some **assumptions** in order to simplify the theoretical analysis:

- Vertices in V are ordered according to their degree. Each edge $\{v_1, v_2\}$ is represented by the tuple (v_1, v_2) such that $v_1 < v_2$, and these tuples are ordered lexicographically. If the graph does not meet these requirements, it can be converted to this form in $sort(E)$ I/Os.
- Each edge requires one memory word.
- Cache rows are replaced using an optimal replacement policy.

The algorithm is based on two **subroutines**:

- **Subroutine 1**: (Pagh et al. [7, Lemma 1]) This subroutine counts, given an edge set E and a vertex v, the number of triangles in E that contain v. This algorithm has an $O(sort(E))$ I/O complexity and it is used to count triangles for "too high" degree vertices, a process that would not be efficient using Subroutine 2. The pseudocode of this subroutine is shown in Algorithm 3.3.
- **Subroutine 2**: (Hu et al. [5, Algorithm 1, step 2], Pagh et al. [7, Lemma 2]) This subroutine counts all the triangles in an edge set E with a pivot edge in $E' \subseteq E$. The algorithm has an $O\left(\left\lceil \frac{E'}{M} \right\rceil \sum_{v \in V} \frac{\deg(v)}{B}\right) \in O\left(\frac{E}{B} + \frac{E'E}{MB}\right)$ I/O complexity. The pseudocode of this subroutine is shown in Algorithm 3.4.

The **main algorithm** (Algorithm 3.5) utilizes the two subroutines to count all the triangles in the graph. The algorithm is divided into steps:

1. V_h computation: The algorithm determines the set of high-degree vertices V_h, which contains all the vertices $v \in V$ having $deg(v) > \sqrt{EM}$. The algorithm also determines the set E_h of all edges incident to at least one vertex in V_h. The sets $V_l = V \setminus V_h$ and $E_l = E \setminus E_h$ are determined consequentially.
2. Triangle counting on high-degree vertices: The algorithm counts all the triangles in the graph that include at least one vertex in V_h, using Subroutine 1.
3. Vertices coloring and edge partitioning: The algorithm partitions the vertices into $c = \sqrt{E/M}$ partitions, inducing $c^2 = E/M$ partitions in E_l, where each partition is characterized by the color of the starting vertex τ_1 and the ending vertex τ_2. More formally, each partition of E_l is defined as

$E_{\tau_1,\tau_2} = \{\{v_1, v_2\} \in E_l \mid v_1 < v_2,\ \xi(v_1) = \tau_1,\ \xi(v_2) = \tau_2\}$. Once we apply vertex coloring, all the sets E_{τ_1,τ_2} can be constructed using a sorting algorithm over E_l.

4. Triangle counting on low-degree vertices: For every triple $(\tau_1, \tau_2, \tau_3) \in \{1, \ldots, c\}^3$, enumerate all triangles with a cone vertex of color τ_1 and a pivot edge in E_{τ_2,τ_3}. We use the algorithm in Subroutine 2 by setting the pivot edge to E_{τ_2,τ_3}, the edge set to $E_{\tau_1,\tau_2} \cup E_{\tau_1,\tau_3} \cup E_{\tau_2,\tau_3}$, and ignoring triangles where the cone vertex does not have color τ_1.

Algorithm 3.3 Cache-Aware Triangle Enumeration: Subroutine 1

Require: $v \in V$

$\Gamma_v \leftarrow N(v)$
Sort Γ_v by degree
Sort E by smallest vertex
Find the set $E_v \subseteq E$ of edges with the smallest vertex in Γ_v
Sort E_v by largest vertex
Compute the set of edges $E'_v \subseteq E_v$ with both both vertices in Γ_v
for $e = \{u, w\} \in E'_v$ **do**
    Count triangle $\{v, u, w\}$
end for

Algorithm 3.4 Cache-Aware Triangle Enumeration: Subroutine 2

Require: Pivot edge set E', Edge set E, color τ_1

$\Gamma_{\text{mem}} \leftarrow$ vertices appearing in edges of E' in internal memory
for each vertex $v \in V$ **do**
    $\Gamma_v \leftarrow \{u \mid (v, u) \in E, u > v, u \in \Gamma_{\text{mem}}\}$
end for
Count all triangles where $\{u, w\} \in E'$ and $u, w \in \Gamma_v$

Let's break down this algorithm for a clearer understanding. The algorithm begins by focusing on triangles involving vertices with high degrees. Since these vertices are incident to many edges, it's more efficient in terms of I/O operations of the cache memory to count triangles using set intersections. Once all triangles involving at least one high-degree vertex are counted, they can be safely removed from the graph, leaving a subgraph comprising only low-degree vertices, $G_l = (V_l, E_l)$.
Next, the aim is to partition V_l into $c = \sqrt{\frac{E}{M}}$ partitions, where E represents the number of edges in the graph and M is the cache memory size. Note that, since this is a cache-aware algorithm, it requires a knowledge of the cache memory present in the system, particularly of its size. Since the vertices set is divided into $c = \sqrt{E/M}$, and considering each edge connects two vertices, the edge set is consequentially divided into

Algorithm 3.5 Cache-Aware Triangle Enumeration

1: $c \leftarrow \sqrt{E/M}$
2: $V_h \leftarrow$ set of high-degree vertices such that $\deg(v) > \sqrt{EM}$
3: $V_l \leftarrow V \setminus V_h$
4: $E_l \leftarrow \{(v,u) \in E \mid v,u \in V_l\}$
5: **for** each vertex $v \in V_h$ **do**
6: Subroutine1(v)
7: **end for**
8: Choose ξ uniformly at random from a 4-wise independent family of functions
9: Construct sets E_{τ_1,τ_2} using a sorting algorithm on E_l
10: **for** every triple $(\tau_1,\tau_2,\tau_3) \in \{1,\ldots,c\}^3$ **do**
11: Subroutine2(E_{τ_2,τ_3}, $E_{\tau_1,\tau_2} \cup E_{\tau_1,\tau_3} \cup E_{\tau_2,\tau_3}$, τ_1)
12: **end for**

$c^2 = E/M$ partitions, means that each partition has a size of M, which exactly fits into the cache memory. However, due to the random nature of the coloring function, this holds true only in expectation.

If all the partitions fit in the cache memory, operations within it do not cause cache memory I/Os. Subroutine 2 can then utilize a partition as a pivot set E', checking how many adjacent vertices form triangles with each edge in E'. Notably, if a partition E_{τ_2,τ_3} is used as the pivot set, the remaining two edges of the triangle must be in $E_{\tau_1,\tau_2} \cup E_{\tau_1,\tau_3}$, reducing the edge set E for Subroutine 2 and limiting its I/O complexity. Note that, in general, if $E' \leq M$, as expected, the I/O complexity of Subroutine 2 is $O(\sum_{v \in V_l} \deg(v)/B)$.

Chapter 4

Experimental Evaluation

In this chapter, we present the experimental evaluation of MinBucket and cache aware algorithms. The primary focus of this analysis is to understand the performance implications of these algorithms when used on modern computational architectures that feature multiple levels of memory hierarchies. In my experiments, I extensively used random graphs varying in size, density, and degree distribution, to assess the robustness and scalability of the algorithms across different graph configurations. The following sections detail the experimental setup, including the hardware and software environments, the specific implementations of the algorithms, and the methodologies employed for data collection and analysis. Subsequently, we present a comprehensive set of results, followed by a discussion on the observed trends and their implications for both theoretical and practical aspects of triangle counting in hierarchical memory systems.

4.1 Benchmark Setup and Performance Metrics

The experiments were conducted on the `AlgoDei` cluster, provided by the University of Padua. The `AlgoDei` cluster features 125GB of RAM and two Intel Xeon(R) W-2245 CPU 3.90GHz processors, each equipped with a cache memory having the following characteristics:

- L1 Data Cache: 8 x 32 KiB
- L1 Instruction Cache: 8 x 256 KiB
- L2 cache: 8 x 1024 MiB
- L3 cache: 16.5 MiB

The experimental evaluations of the two triangle counting algorithm are primarily focused on two aspects:

1. **Total Computation Time**: This metric represents the overall time taken by the algorithm to execute and compute the total number of triangles within a given graph instance. This is the metric of most interest to practitioners focused on fast computation.

2. **Number of Cache Misses**: Cache misses occur when the required data is not present at the current cache level and must be retrieved from a higher level, requiring additional time. The count of cache misses serves as a crucial indicator of cache utilization efficiency within the hierarchical memory architecture.

All algorithm implementations were developed from scratch in the C language, aiming to minimize overheads and maximize memory locality in the utilized data structures. Power-law distributed weights for the Erased Configuration Model are generated using the `PLSeqGen` library by Del Genio et al. [4].
Cache misses were measured using the Intel Performance Monitoring Unit (PMU), an integrated hardware component within modern Intel CPUs that provides access to hardware counters, including the cache misses counter. Access to the Intel PMU was achieved using the `perf_event` subsystem provided by the Linux kernel, which can be utilized in a C program through system calls. The decision to directly utilize hardware counters offers several advantages:

- **Realism**: Unlike software simulation tools such as `Valgrind`, using hardware events allows us to observe the actual behavior of algorithms in a real environment.

- **Accuracy**: The Intel PMU counters enable us to obtain more precise information about cache misses compared to external tools like `perf`, especially when we are interested in measuring specific parts of the code.

4.2 Results and Analysis

Here, we present all the results obtained from the conducted experiments, analyzing the outcomes and trying to provide explanations for them. Firstly, we present the results of my research training activity, which primarily focused on understanding the MinBucket algorithm and testing it using random ECM graphs, following the approach of Berry et al. [1]. Secondly, we present the results of the cache aware algorithm, using them to compare the two algorithms, assessing the advantage of utilizing the memory hierarchy.

4.2.1 MinBucket Algorithm

Total Work Analysis

These analyses follow the work of Berry et al. [1], aiming to experimentally validate the theorems presented in their study.

The experiment involved measuring the total number of P_2s enumerated by the MinBucket algorithm (referred to as **Total Work** in Definition 3.2) to verify the bound provided by Theorem 3.4.

For this experiment, ECM graphs of various sizes were generated based on a power-law degree distribution with a power-law exponent of τ, where $\tau < 7/3$ and $\tau > 7/3$, with the maximum degree truncated to $\sqrt{N}$.

In this tests, sizes ranging from 10^3 to 10^7 nodes were used, with τ values chosen from $\{2.2, 2.4, 2.6\}$. Each total work value represents the mean over 10 experiments. Table 4.1 presents the obtained results, also including the total work of the trivial algorithm for comparison. Figure 4.1 illustrates the results varying with N for each τ value, considering experimental errors. Additionally, Figure 4.2 plots the results obtained for the trivial algorithm.

Table 4.1: MinBucket Total Work

τ	N	Avg. MinBucket Total Work	Avg. Trivial Total Work
2.2	10^4	$(8.08 \pm 2.26) \times 10^3$	2.35×10^6
2.4	10^4	$(2.73 \pm 0.72) \times 10^3$	2.93×10^5
2.6	10^4	$(1.00 \pm 0.38) \times 10^3$	1.02×10^5
2.2	10^5	$(1.57 \pm 0.18) \times 10^5$	9.48×10^7
2.4	10^5	$(3.76 \pm 0.38) \times 10^4$	1.04×10^7
2.6	10^5	$(1.13 \pm 0.14) \times 10^4$	2.13×10^6
2.2	10^6	$(2.65 \pm 0.28) \times 10^6$	1.68×10^9
2.4	10^6	$(4.57 \pm 0.32) \times 10^5$	2.56×10^8
2.6	10^6	$(1.20 \pm 0.12) \times 10^5$	3.53×10^7
2.2	10^7	$(4.35 \pm 0.14) \times 10^7$	1.81×10^{10}
2.4	10^7	$(5.37 \pm 0.22) \times 10^6$	3.06×10^9
2.6	10^7	$(1.25 \pm 0.02) \times 10^6$	6.60×10^8

Table 4.2: MinBucket Constant Factors

τ	N	Constant Factors	C
2.4	10^4	0.27	0.63
2.6	10^4	0.10	0.30
2.4	10^5	0.38	0.63
2.6	10^5	0.11	0.30
2.4	10^6	0.46	0.63
2.6	10^6	0.12	0.30
2.4	10^7	0.54	0.63
2.6	10^7	0.13	0.30

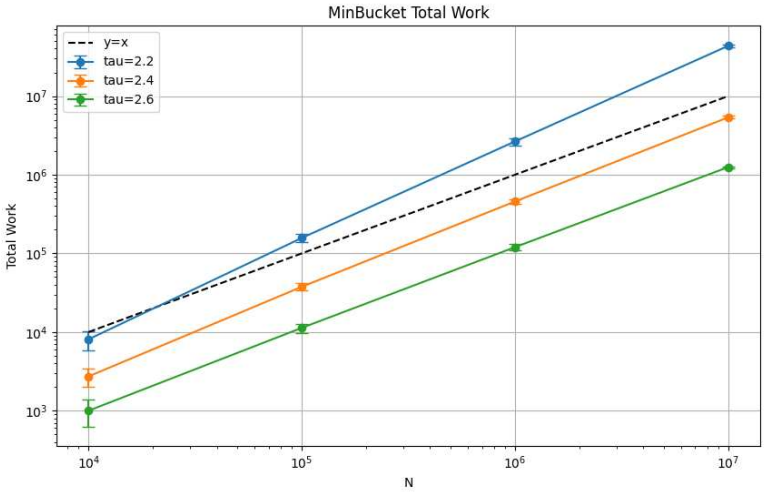

Figure 4.1: MinBucket Total work varying N and τ

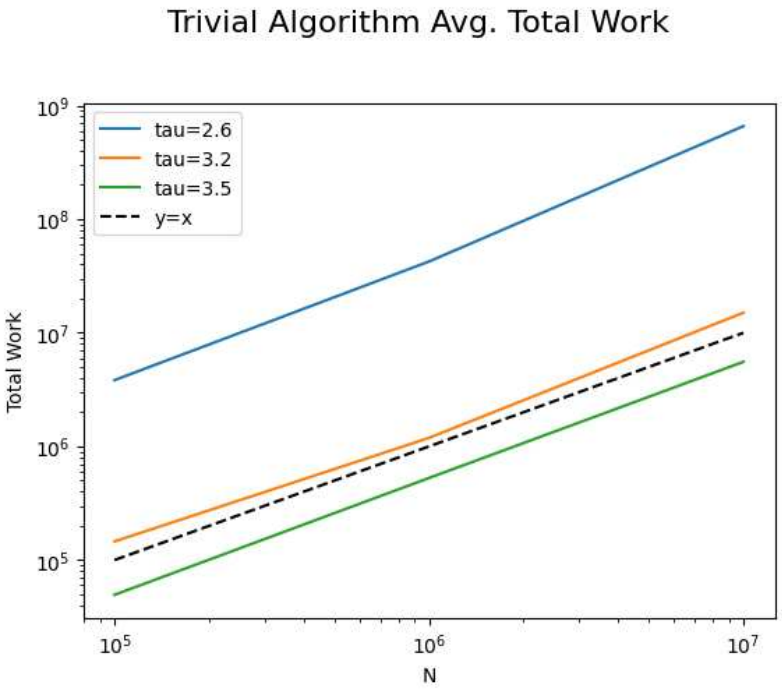

Figure 4.2: Trivial Algorithm Total work varying N and τ

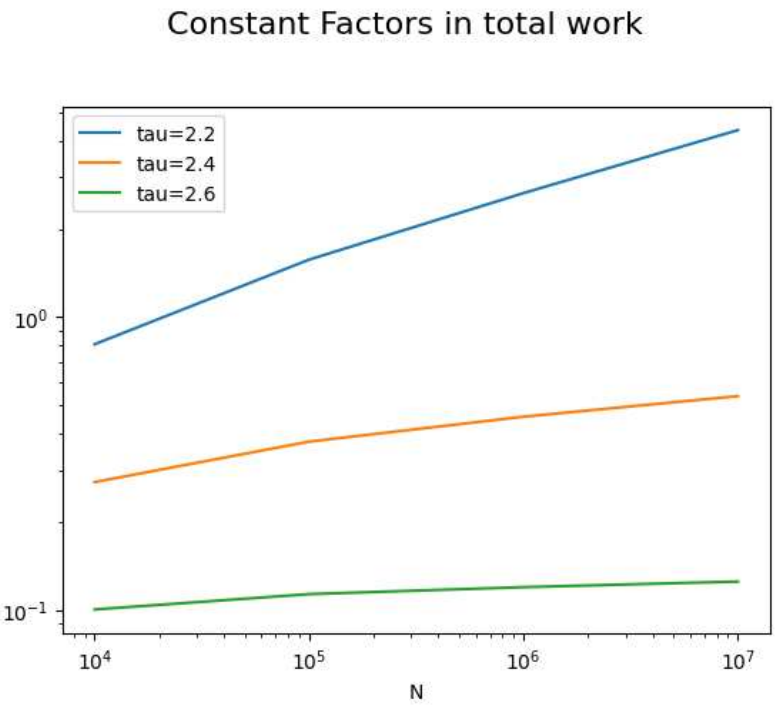

Figure 4.3: MinBucket Constant Factors Plot

From Figure 4.1, we can assess the validity of Theorem 3.4. The plot shows the line $y = x$, as a reference. It is evident from the plot that for $\tau > 2.3$, the total work tends to increase almost linearly with N, and as τ increases, the total work behavior becomes increasingly linear. It is noteworthy that, as stipulated by the theorem, the trivial algorithm begins to exhibit linear behavior only when $\tau > 3$, as illustrated in Figure 4.2. Moreover, Table 4.1 demonstrates that the trivial algorithm requires over 100 times more total work compared to the MinBucket algorithm. Consequently, in power-law graphs under identical testing conditions, we can find that the execution time of the trivial algorithm will exhibit a similar ratio relative to the MinBucket execution time in power-law graphs. This assertion is corroborated by Suri et al.'s experiments [8] on real graphs, which are known to follow a power-law distribution.

To test Theorem 3.5, I computed the C value as defined by the theorem for $\tau = 2.4$ and $\tau = 2.6$, and compared it to the constant factors derived from the total work shown in Table 4.1. These constant factors were computed as the ratio between the total work and N. It is worth recalling that the theorem states that the total work is $O(nC)$ when $\tau > 7/3$.

Table 4.2 presents the results, and Figure 4.3 illustrates the constant factors as a function of N. The figure also includes the case where $\tau = 2.2$, where the theorem bound is not applicable.

The results prove that the theorem is valid; in fact, the actual constant factors given by

the experiments are upper-bounded by the C constant given by the theorem.

Execution Time Analysis

Berry et al. [1] in their study assumed that checking the existence of an edge requires constant time. This performance can be easily achieved using an adjacency matrix, but the space requirements ($O(n^2)$) make its usage infeasible for large graph instances. Thus, in my research training activity, I evaluated two possible solutions to address this problem:

- **Hash table**: Using an array of hash tables, one for each adjacency list, should guarantee constant-time edge existence checks on average.
- **Binary Search**: If edges are stored in lexicographic order, then a simple binary search algorithm on the correct bucket can check edge existence in $O(\log(X_v))$ time, where the edge to check is (v, u) and X_v is the size of the bucket B_v.

To determine the optimal approach for checking the existence of edges (necessary for verifying if a $P2$ is closed in the MinBucket algorithm), I conducted profiling on two versions of the MinBucket algorithm: one utilizing the hash table approach and the other employing the binary search approach.
Measurements were carried out using the `gprof` tool in Linux, which allows analysis of the time spent inside a function during program execution and the total count of calls to that function. By understanding the total time spent within the function and the overall number of calls, I calculated the mean time spent per execution of the function.
The mean execution time for checking edge existence was measured for both the hash table and binary search versions. These measurements were taken across varying values of N and τ. Table 4.3 presents the mean execution time based on 10 instances for each combination of N and τ. From the results, it is evident that the binary search approach is significantly faster than the hash table approach, despite its theoretical dependency on the size of the search space. These observed execution times may be attributed to the issues of collisions and the overhead introduced by the hash table structure, as described in the Preliminaries. For this reason, all the measurements conducted in the time analysis section refer to the binary search version.
Table 4.4 shows the mean execution time, along with its experimental error. Figure 4.4 illustrates the execution time plot as N increases. The plot is clearly similar to the plot in Figure 4.1, confirming that the time performance is mainly determined by the number of P_2s enumerated. However, the results are slightly different because of the time needed to check if each P_2 is closed, which is not constant for the reasons described above.

Table 4.3: Hash Table vs. Binary Search Lookup Time

τ	N	Binary Search Avg. Lookup Time (us)	Hash Table Avg. Lookup Time (us)
2.1	10^5	0.062	0.112
2.3	10^5	0.054	0.114
2.5	10^5	0.057	0.137
2.1	10^6	0.082	0.269
2.3	10^6	0.085	0.228
2.5	10^6	0.084	0.207
2.7	10^6	0.101	0.254
2.1	10^7	0.105	0.360
2.3	10^7	0.113	0.363
2.5	10^7	0.117	0.344
2.7	10^7	0.123	0.319

Table 4.4: MinBucket Execution Time Varying N and τ

τ	N	Avg. Execution Time (ms)
2.2	10^4	1.43 ± 0.2
2.4	10^4	1.22 ± 0.35
2.6	10^4	1.03 ± 0.32
2.2	10^5	22.75 ± 0.9
2.4	10^5	12.28 ± 0.35
2.6	10^5	9.63 ± 0.21
2.2	10^6	539.9 ± 21.55
2.4	10^6	264.76 ± 8.94
2.6	10^6	201.58 ± 7.74
2.2	10^7	10009.3 ± 124.48
2.4	10^7	4097.77 ± 46.59
2.6	10^7	2890.29 ± 33.31

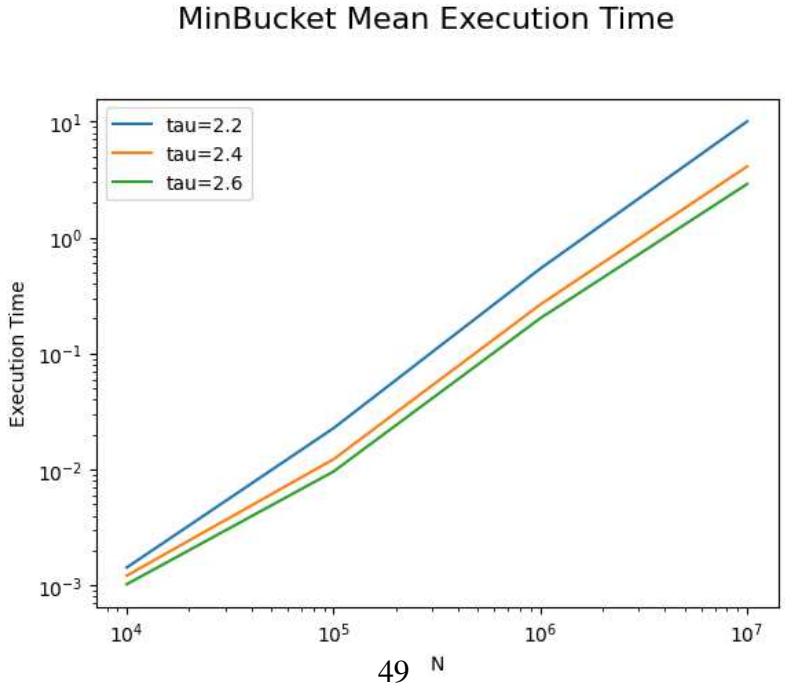

Figure 4.4: MinBucket Execution Time Plot

Cache I/O Analysis

The number of cache I/O operations can be effectively summarized by the count of cache misses occurring during the algorithm's execution. As mentioned at the beginning of this section, to profile the count of cache misses for each parameter setting, I utilized the `perf_event` subsystem in Linux to count the Last-Level Cache misses. As in the all other experiments, to ensure reliable results, the test was performed by repeating the execution 10 times for each combination of the parameters N and τ. Table 4.5 shows the experiment results.

Table 4.5: MinBucket Cache Misses Varying N and τ

N	τ	Avg. Number of Edges	Avg. Triangles Count	Avg. Cache Misses
10^4	2.2	1.45×10^4	7.07×10^3	$(1.30 \pm 1) \times 10^0$
10^4	2.4	1.08×10^4	1.26×10^3	$(4.00 \pm 2) \times 10^0$
10^4	2.6	8.85×10^3	2.58×10^2	$(3.90 \pm 2) \times 10^0$
10^5	2.2	1.53×10^5	9.25×10^4	$(1.35 \pm 0.26) \times 10^3$
10^5	2.4	1.10×10^5	1.04×10^4	$(8.73 \pm 3.55) \times 10^2$
10^5	2.6	8.67×10^4	7.00×10^2	$(5.46 \pm 1.32) \times 10^2$
10^6	2.2	1.64×10^6	1.69×10^6	$(1.88 \pm 0.1) \times 10^5$
10^6	2.4	1.11×10^6	8.61×10^4	$(1.48 \pm 0.036) \times 10^5$
10^6	2.6	8.75×10^5	4.66×10^3	$(1.21 \pm 0.056) \times 10^5$
10^7	2.2	1.71×10^7	2.98×10^7	$(2.95 \pm 0.078) \times 10^7$
10^7	2.4	1.11×10^7	7.19×10^5	$(2.06 \pm 0.021) \times 10^7$
10^7	2.6	8.76×10^6	2.78×10^4	$(1.80 \pm 0.005) \times 10^7$

4.2.2 Cache Aware Algorithm

Implementation

Before presenting the experimental results, it is appropriate to delve into the details of my implementation of the cache-aware algorithm. The implementation of Subroutine 1 and the main algorithm are simply developed as described in Pseudocodes 3.3 and 3.5. The particularity of my specific implementation resides in the implementation of Subroutine 2 (Pseudocode 3.4), particularly in step 5.

During my experiments, I was not able to run the algorithm in a feasible amount of time on large graphs instances using the straightforward implementation of step 5. This straightforward implementation consists of checking, for every $\{u, v\}$ in the pivot edge set, if u and v are both in the same Γ_v. Essentially, this requires scanning all of Γ_v for each edge in the pivot set.

My implementation draws inspiration from the MinBucket algorithm. Given E_{τ_1,τ_2}, E_{τ_1,τ_3}, and the pivot edge set E_{τ_2,τ_3}, my implementation enumerates each possible pair of

vertices u, w such that $(v_1, u) \in E_{\tau_1,\tau_2}$ and $(v_2, w) \in E_{\tau_1,\tau_3}$ with $v_1 = v_2$. For each such pair, the algorithm performs a binary search in the pivot edge set. If the edge exists, then a new triangle is counted. The pseudocode for this implementation is provided in Pseudocode 4.1.

This implementation does not incur more I/O operations in the cache memory compared to the straightforward implementation if all the sets E_{τ_1,τ_2}, E_{τ_1,τ_3}, and E_{τ_2,τ_3} together fit into the cache memory. Experimentally, this implementation has been able to compute the total number of triangles in a reasonable time, even for large graphs instances.

To verify the correctness of the algorithm, it is imperative to ensure that a single iteration of Subroutine 2 does not result in more cache misses than the size of the input sets. In other words, experimentally, we need to ascertain that, by utilizing a suitable value of parameter c that ensures that all the sets fit together into the cache memory, the ratio between the number of cache misses produced in an iteration and the size of the input sets closely approximates 1 in expectation (taking into account the size of the cache line B). To conduct this assessment, the algorithm has been tested on L1 cache over different ECM power-law graphs. Each ECM configuration was subjected to 10 repetitions of the experiment.

The experimental results are presented in Table 4.6. Additionally, the table highlights the advantage of employing the binary search approach, which enables the algorithm to reduce cache misses by approximately 25% compared to a simple linear search in the pivot set.

Since the pivot set arrives in lexicographic order, a further optimization that may reduce the computation time, particularly when dealing with low values of c, involves using an additional data structure to store pointers to the list of neighbors of each vertex $\{v \mid (v, u) \in E_{\tau_2,\tau_3}\}$. This would enable the algorithm to "jump" directly to the relevant portion of the pivot set for the search. While this would require additional $O(E_{\tau_2,\tau_3})$ I/Os, it may reduce the search time within the pivot set, thus decreasing the overall computation time.

Algorithm 4.1 Subroutine 2: Implementation

Require: Pivot edge set E_{τ_2,τ_3}, Sets E_{τ_1,τ_2}, E_{τ_1,τ_3}

1: $\Gamma_{\text{mem}} \leftarrow$ vertices appearing in edges of E_{τ_2,τ_3}
2: **for** each couple of vertices $u, w \in \Gamma_{\text{mem}}$ such that $(v_1, u) \in E_{\tau_1,\tau_2}$, $(v_2, w) \in E_{\tau_1,\tau_3}$ with $v_1 = v_2$ **do**
3: **if** BinarySearch((u, w), E_{τ_2,τ_3}) **then**
4: Count triangle (u, w, v)
5: **end if**
6: **end for**

Table 4.6: Cache Misses / Input size Ratio

N	τ	Avg. N. of edges	Avg. Ratio (Linear Search)	Avg. Ratio (Binary Search)
8×10^5	2.2	$(1.31 \pm 0.02) \times 10^6$	1.07 ± 0.04	0.80 ± 0.01
8×10^5	2.4	$(8.80 \pm 0.14) \times 10^5$	1.03 ± 0.01	0.75 ± 0.01
6×10^5	2.2	$(9.82 \pm 0.14) \times 10^5$	1.08 ± 0.07	0.80 ± 0.01
6×10^5	2.4	$(6.80 \pm 0.13) \times 10^5$	1.03 ± 0.01	0.74 ± 0.01
4×10^5	2.2	$(6.50 \pm 0.09) \times 10^5$	1.11 ± 0.05	0.81 ± 0.01
4×10^5	2.4	$(4.43 \pm 0.07) \times 10^5$	1.01 ± 0.01	0.74 ± 0.01

Hyperparameter Tuning

From a practical point of view, the most important objective is to reduce the computation time needed to process graph instances as much as possible. From the definition of the algorithm [7], we know that vertices are partitioned using $c = \sqrt{E/M}$ different colors, inducing c^2 partitions of the edge set, each of size M in expectation. However, the parameter c can be adjusted by a suitable constant α to interpolate between a high number of small partitions and a low number of large partitions. For example, to ensure that during the execution of Subroutine 2 all three partitions involved fit into the cache memory in expectation, we need $\alpha > \sqrt{3}$.
However, this may not be the best choice in terms of overall computation time for several reasons:

- **Number of partitions**: A large number of partitions may increase the overhead of the algorithm, slowing down the computation process.
- **Optimizations**: The use of optimizations (e.g., the binary search approach described above, the use of additional data structures) may change the theoretical number of I/O operations.
- **Hidden mechanisms**: Some hidden mechanisms may interfere with the ideal cache behavior (e.g., prefetching, replacement policy, etc.).

Therefore, we need to tune α through an experimental process, evaluating the computation time and choosing the best values, which may change based on the characteristics of the graphs involved. Thus, I evaluated the computation time of the algorithm over different ECM power-law graphs, uniformly distributed graphs, and common real-world networks. Tables 4.7 to 4.9 present the experimental results, which are graphically illustrated in Figures 4.5 to 4.7. These figures depict the execution times for each graph, normalized with respect to the minimum execution time obtained within each graph.

Table 4.7: Execution Time Varying α Across Power-law ECM Graphs

N	τ	Avg. number of edges	α	Avg. Execution Time (s)
1.00×10^7	2.1	2.35×10^7	0.5	19.16 ± 1.01
1.00×10^7	2.1	2.35×10^7	1.0	19.01 ± 1.48
1.00×10^7	2.1	2.35×10^7	1.5	20.54 ± 1.56
2.00×10^7	2.4	2.23×10^7	0.5	5.02 ± 0.04
2.00×10^7	2.4	2.23×10^7	1.0	7.43 ± 0.05
2.00×10^7	2.4	2.23×10^7	1.5	10.98 ± 0.48
3.00×10^7	2.7	2.42×10^7	0.5	5.79 ± 0.01
3.00×10^7	2.7	2.42×10^7	1.0	9.20 ± 0.01
6.00×10^7	3.0	4.11×10^7	0.5	10.79 ± 0.05
6.00×10^7	3.0	4.11×10^7	1.0	27.17 ± 0.76

Table 4.8: Execution Time Varying α Across Uniformly Distributed Graphs

N	Avg. number of edges	α	Avg. Execution Time (s)
1.00×10^7	2.35×10^7	0.5	10.24
1.00×10^7	2.35×10^7	1.0	11.72
1.00×10^7	2.35×10^7	1.5	12.82
2.00×10^7	2.23×10^7	0.5	8.96
2.00×10^7	2.23×10^7	1.0	11.15
2.00×10^7	2.23×10^7	1.5	12.71
3.00×10^7	2.42×10^7	0.5	9.91
3.00×10^7	2.42×10^7	1.0	12.73
3.00×10^7	2.42×10^7	1.5	14.39
6.00×10^7	4.11×10^7	0.5	17.39
6.00×10^7	4.11×10^7	1.0	26.3
6.00×10^7	4.11×10^7	1.5	53.02

Table 4.9: Execution Time Varying α Across Real-World Graphs

Graph	α	Average Execution Time (s)
LiveJournal1	0.5	41.18 ± 0.11
LiveJournal1	1.0	42.07 ± 0.64
LiveJournal1	1.5	46.53 ± 0.04
LiveJournal1	2.0	54.12 ± 1.50
soc-Pokec	0.5	15.53 ± 0.04
soc-Pokec	1.0	15.43 ± 0.01
soc-Pokec	1.5	16.06 ± 0.03
soc-Pokec	2.0	18.10 ± 0.03
higgs-twitter	1.0	19.84 ± 0.06
higgs-twitter	1.5	18.77 ± 0.01
higgs-twitter	2.0	18.82 ± 0.26
as-Skitter	1.0	4.99 ± 0.01
as-Skitter	1.5	5.28 ± 0.01
as-Skitter	2.0	5.72 ± 0.02
wiki-Talk	1.0	2.46 ± 0.01
wiki-Talk	2.5	2.61 ± 0.01

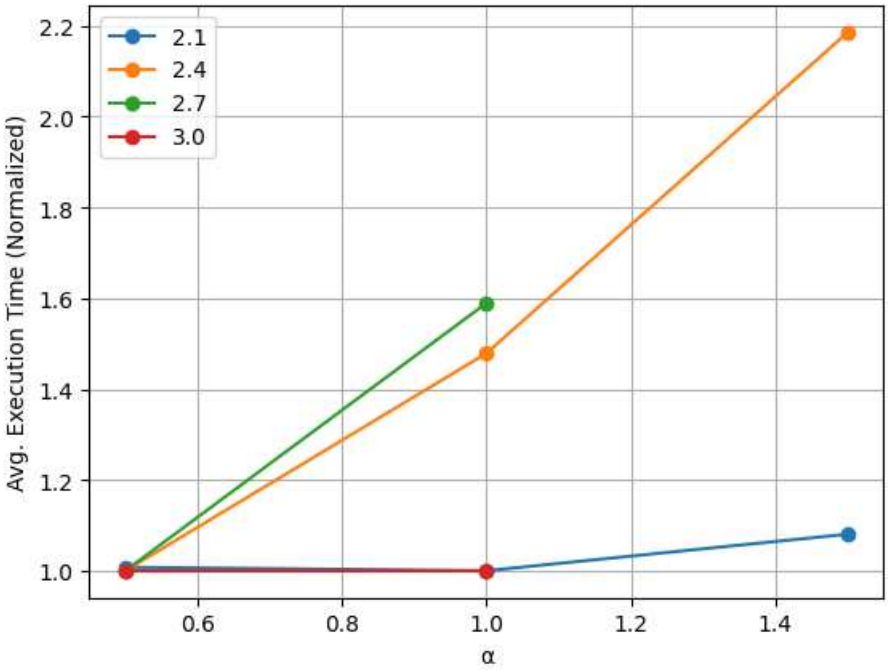

Figure 4.5: Avg. Execution Time (Normalized) Varying α in Power-law ECM Graphs

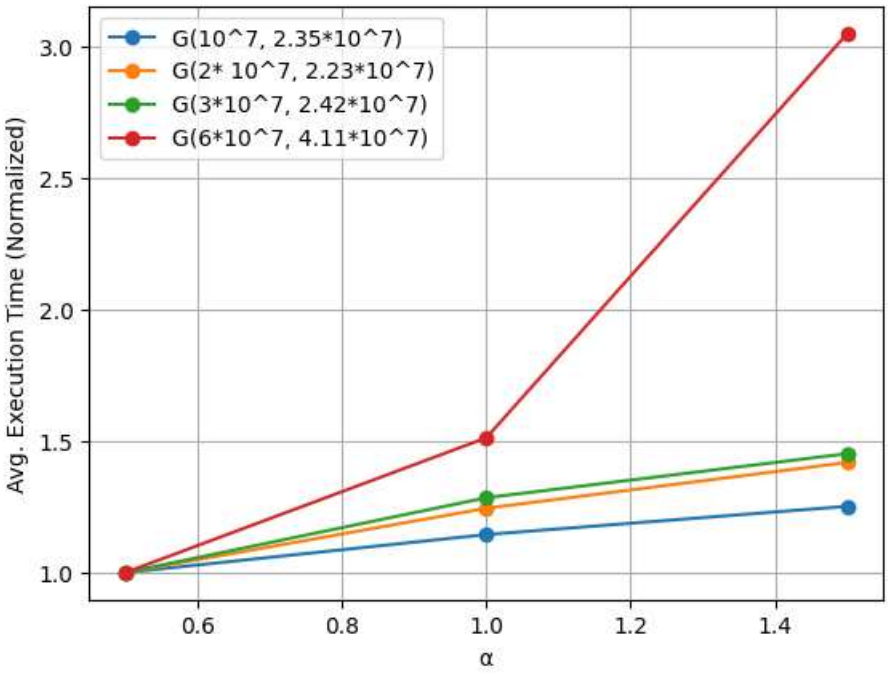

Figure 4.6: Avg. Execution Time (Normalized) Varying α in Uniformly Distributed Graphs

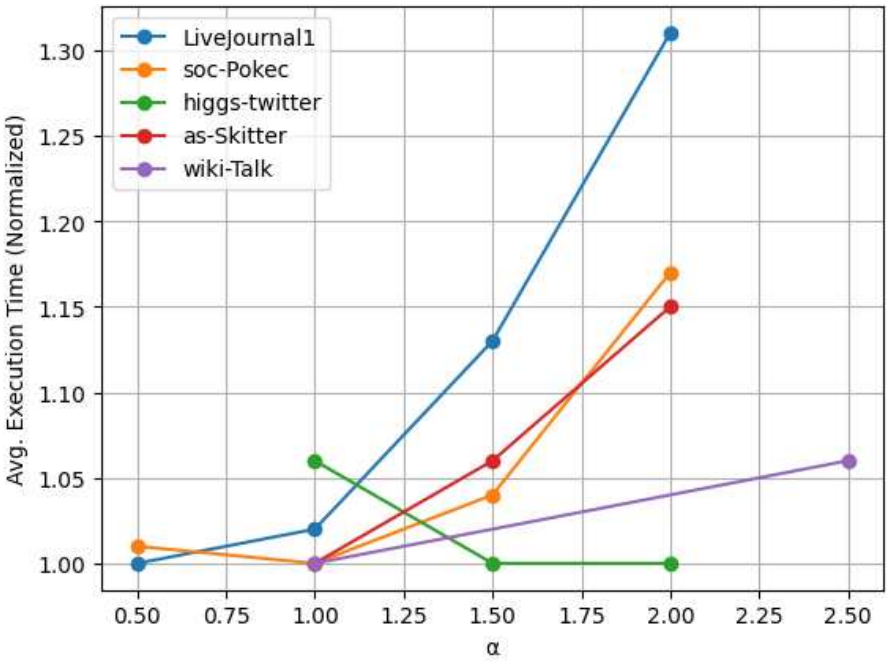

Figure 4.7: Avg. Execution Time (Normalized) Varying α in Real-World Graphs

Execution Time And Cache Misses Analysis

Here, we present the results of experiments aimed at measuring the execution time and cache misses required by the MinBucket and cache aware algorithms for computing triangles on different types of graphs. The experiments involved the usage of:

- Power-law distributed ECM random graphs
- Uniform sparse graphs, generated using the $G(n, m)$ model
- Uniform dense graphs, generated using the $G(n, p)$ model
- Real-world graphs datasets

Like all other experiments, each result represents the average of 10 experiments, with the corresponding experimental error also reported. Tables 4.10 to 4.17 present the experimental results.
The results underscore the superior performance of the cache-aware algorithm over the MinBucket algorithm in graphs following a power-law distribution, both random and real. This highlights the advantage of efficient cache utilization in such graphs, which is particularly encouraging news for practitioners, as these are the types of graphs they are most interested in.
However, for uniformly distributed graphs, this advantage almost disappears. This outcome is not unexpected: in graphs following a power-law distribution, a few nodes (hubs) have very high degrees, while most nodes have low degrees. Consequently, the cache-aware algorithm can effectively leverage this by processing these nodes in a more cache-efficient manner compared to MinBucket.
In uniformly distributed graphs instead, node degrees are relatively evenly distributed, with no significant hubs. This means that every part of the graph is accessed roughly equally, reducing the advantage of cache-aware optimizations. Moreover:

- **In uniform sparse graphs**, nodes have a small number of neighbors. This implies that, in MinBucket, each bucket will likely have a small size, potentially fitting into the cache memory. Thus, even though the MinBucket algorithm is not explicitly designed for memory hierarchy, its cache efficiency increases in uniformly sparse graphs, making it comparable to a cache-aware algorithm.
- **In uniform dense graphs**, all nodes tend to have a high number of neighbors. While it is true that the cache-aware approach will decrease the total number of I/O operations, it is also true that the time complexity of both algorithms explodes on dense graphs. Moreover, the overhead introduced by the cache-aware algorithm may nullify its advantages compared to a simpler algorithm like MinBucket.

These arguments are also supported by Tables 4.14 to 4.17, which show the number of cache misses incurred by both algorithms during the experiments.

Table 4.10: Execution time comparison in ECM power-law graphs

N	τ	Avg. Number of edges	Avg. Number of triangles	CacheAware Avg. Ex. time (s)	MinBucket Avg. Ex. time (s)
1.50×10^7	2.10	3.56×10^7	2.74×10^8	35.63 ± 2.03	72.83 ± 3.59
1.50×10^7	2.40	1.67×10^7	1.10×10^6	3.60 ± 0.04	6.60 ± 0.11
1.00×10^7	2.10	2.35×10^7	1.66×10^8	19.52 ± 1.31	39.41 ± 3.01
1.00×10^7	2.40	1.12×10^7	7.74×10^5	2.32 ± 0.03	4.15 ± 0.07
5.00×10^6	2.10	1.15×10^7	6.54×10^7	7.43 ± 0.43	15.19 ± 1.09
5.00×10^6	2.40	5.61×10^6	5.18×10^5	1.10 ± 0.01	1.86 ± 0.04

Table 4.11: Execution time comparison in uniformly distributed sparse random graphs

N	Number of edges	Avg. Number of triangles	CacheAware Avg. Ex. time (s)	MinBucket Avg. Ex. time (s)
1.50×10^7	3.56×10^7	19.40	15.00 ± 0.61	11.32 ± 0.06
1.50×10^7	1.67×10^7	1.30	3.75 ± 0.01	4.61 ± 0.02
1.00×10^7	2.35×10^7	17.10	7.58 ± 0.03	7.12 ± 0.03
1.00×10^7	1.12×10^7	2.20	2.41 ± 0.01	2.94 ± 0.02
5.00×10^6	1.15×10^7	17.90	2.62 ± 0.01	3.13 ± 0.01
5.00×10^6	5.61×10^6	1.50	1.13 ± 0.01	1.35 ± 0.01

Table 4.12: Execution time comparison in uniformly distributed dense random graphs

N	Number of edges	Avg. Number of triangles	CacheAware Avg. Ex. time (s)	MinBucket Avg. Ex. time (s)
4000	2.40×10^6	2.88×10^8	65.13 ± 0.12	66.86 ± 0.08
4500	3.04×10^6	4.10×10^8	94.20 ± 0.08	96.45 ± 0.14
5000	3.75×10^6	5.62×10^8	130.79 ± 0.23	133.88 ± 0.11

Table 4.13: Execution time comparison in real-world graphs

Graph Name	N	Number of edges	Cache Aware Avg. Ex. Time (s)	MinBucket Ex. time (s)
higgs-twitter	4.57×10^5	1.25×10^7	19.54 ± 0.06	39.94
LiveJournal1	4.85×10^6	4.29×10^7	41.16 ± 0.23	55.62
soc-Pokec	1.63×10^6	2.23×10^7	15.18 ± 0.05	21.76
as-Skitter	1.70×10^6	1.11×10^7	4.94 ± 0.02	7.12
wiki-Talk	2.39×10^6	4.66×10^6	2.43 ± 0.01	4.11

Table 4.14: Cache misses comparison in ECM power-law graphs

N	τ	Avg. Number of edges	Avg. Number of triangles	CacheAware Avg. Cache Misses	MinBucket Avg. Cache Misses
1.50×10^7	2.10	3.56×10^7	2.74×10^8	$(8.80 \pm 1.57) \times 10^7$	$(4.59 \pm 0.28) \times 10^8$
1.50×10^7	2.40	1.67×10^7	1.10×10^6	$(1.63 \pm 0.02) \times 10^7$	$(5.17 \pm 0.05) \times 10^7$
1.00×10^7	2.10	2.35×10^7	1.66×10^8	$(2.95 \pm 0.08) \times 10^7$	$(2.01 \pm 0.17) \times 10^8$
1.00×10^7	2.40	1.12×10^7	7.74×10^5	$(9.00 \pm 0.13) \times 10^6$	$(3.12 \pm 0.02) \times 10^7$
5.00×10^6	2.10	1.15×10^7	6.54×10^7	$(5.31 \pm 0.20) \times 10^6$	$(5.03 \pm 0.34) \times 10^7$
5.00×10^6	2.40	5.61×10^6	5.18×10^5	$(3.18 \pm 0.07) \times 10^6$	$(1.31 \pm 0.01) \times 10^7$

Table 4.15: Cache misses comparison in uniformly distributed sparse random graphs

N	Number of edges	Avg. Number of triangles	MinBucket Avg. Cache Misses	CacheAware Avg. Cache Misses
1.50×10^7	3.56×10^7	19.40	$(1.18 \pm 0.00) \times 10^8$	$(1.62 \pm 0.28) \times 10^8$
1.50×10^7	1.67×10^7	1.30	$(4.30 \pm 0.01) \times 10^7$	$(2.76 \pm 0.01) \times 10^7$
1.00×10^7	2.35×10^7	17.10	$(7.35 \pm 0.01) \times 10^7$	$(7.41 \pm 0.05) \times 10^7$
1.00×10^7	1.12×10^7	2.20	$(2.69 \pm 0.01) \times 10^7$	$(1.54 \pm 0.02) \times 10^7$
5.00×10^6	1.15×10^7	17.90	$(3.06 \pm 0.00) \times 10^7$	$(1.80 \pm 0.01) \times 10^7$
5.00×10^6	5.61×10^6	1.50	$(1.15 \pm 0.00) \times 10^7$	$(5.58 \pm 0.06) \times 10^6$

Table 4.16: Cache misses comparison in uniformly distributed dense random graphs

N	Avg. Number of edges	Avg. Number of triangles	CacheAware Avg. Cache Misses	MinBucket Avg. Cache Misses
4000	2.40×10^6	2.88×10^8	$(2.13 \pm 0.17) \times 10^5$	$(9.32 \pm 0.23) \times 10^6$
4500	3.04×10^6	4.10×10^8	$(4.15 \pm 0.37) \times 10^5$	$(2.44 \pm 0.04) \times 10^7$
5000	3.75×10^6	5.62×10^8	$(7.04 \pm 1.42) \times 10^5$	$(4.97 \pm 0.04) \times 10^7$

Table 4.17: Cache misses comparison in real-world graphs

Graph Name	N	Number of edges	Number of triangles	CacheAware Avg. Cache Misses	MinBucket Cache Misses
higgs-twitter	4.57×10^5	1.25×10^7	8.30×10^7	$(5.81 \pm 0.23) \times 10^6$	1.64×10^8
LiveJournal1	4.85×10^6	4.29×10^7	2.86×10^8	$(9.99 \pm 0.13) \times 10^7$	2.13×10^8
soc-Pokec	1.63×10^6	2.23×10^7	3.26×10^7	$(2.34 \pm 0.03) \times 10^7$	1.24×10^8
as-Skitter	1.70×10^6	1.11×10^7	2.88×10^7	$(3.93 \pm 0.02) \times 10^6$	1.10×10^7
wiki-Talk	2.39×10^6	4.66×10^6	9.20×10^6	$(6.26 \pm 0.16) \times 10^5$	8.03×10^6

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this thesis, we investigated the performance of triangle counting algorithms on random graphs in hierarchical memory architectures. Specifically, we focused on two algorithms: the MinBucket algorithm and a cache aware algorithm. Our experiments were conducted on a variety of graph types, including power-law distributed ECM graphs, uniformly distributed sparse and dense graphs, and real-world graph datasets. We evaluated the algorithms based on total work (i.e. the total number of P_2s enumerated), execution time, and cache efficiency.
The experimental results evaluated the MinBucket algorithm's performance across different types of graphs to validate its theoretical bounds. Key findings include:

- **Total Work Analysis**: The experimental results confirmed the theoretical predictions by Berry et al. [1]. For power-law graphs, the total work of the MinBucket algorithm scales almost linearly with N for $\tau > 2.3$, validating Theorem 3.4. The trivial algorithm, by comparison, required significantly more total work, particularly when τ is small.

- **Execution Time**: A binary search approach for checking edge existence proved superior to the hash table approach, reducing the mean execution time. The time performance of the MinBucket algorithm has been proved to closely follow its total work performance, affirming that the primary factor influencing execution time was the number of P_2s enumerated.

- **Cache I/O Analysis**: Profiling with `perf_event` showed that cache misses are a significant factor in performance. The MinBucket algorithm, while not specifically designed for cache efficiency, demonstrated varying cache performance based on graph structure and size.

We then evaluated the cache aware algorithm, designed to exploit the memory hierarchy, aiming to minimize cache misses and improve execution times. Key findings include:

- **Implementation**: An optimized implementation of Subroutine 2 using binary search within the pivot set significantly improved performance. The tuning of the hyperparameter α was crucial, as it influenced the number of partitions and overall computation time.
- **Performance Analysis**: The cache aware algorithm demonstrated superior performance on power-law graphs, both random and real-world, compared to the MinBucket algorithm. This advantage was primarily due to efficient cache utilization, which is critical in power-law distributed graphs with high-degree nodes.
- **Uniform Graphs Problem**: In uniformly distributed graphs, the performance advantage of the cache aware algorithm diminished. Uniform sparse graphs allowed MinBucket to benefit from smaller bucket sizes that fit into cache, while uniform dense graphs led to high time complexity for both algorithms, neutralizing the benefits of cache-aware optimizations.

The experimental results underscore the importance of considering memory hierarchy in algorithm design, particularly for large-scale graph processing. The cache aware algorithm's performance boost in power-law graphs highlights the potential of tailored memory-efficient algorithms in practical applications, such as social network analysis, web graph mining, and bioinformatics.
However, the results also indicate that for uniformly distributed graphs, simpler algorithms like MinBucket can be competitive, especially when graph density varies. This suggests that algorithm selection should be context-dependent, based on the specific characteristics of the input graph.

5.2 Future Work

This study opens several directions for future research, which can further enhance our understanding and capabilities in triangle counting algorithms and their performance in hierarchical memory architectures. Key areas for future work may include:

- **Extended Graph Types**: While this thesis focused on power-law distributed ECM graphs, uniformly distributed graphs, and real-world datasets, it would be valuable to investigate the performance of triangle counting algorithms on other types of graphs. Geometric Inhomogeneous Random Graphs, for instance, present unique

challenges and opportunities for optimization. These graphs are characterized by non-uniform distribution of nodes and edges in a geometric space, where nodes closer together are more likely to be connected. Understanding how memory hierarchy impacts these structures could lead to the development of more adaptive and resilient algorithms.

- **Deeper Cache Analysis**: A more detailed analysis of cache behavior, including the effects of different cache levels (L1, L2, L3) and architectures, would provide deeper insights into the performance bottlenecks and opportunities for optimization. For example, given the size and structure of the graph, it may be advantageous to tailor algorithms to operate with a specific memory hierarchy level in mind and, by adjusting parameters accordingly and implementing optimizations tailored to the graph's characteristics, we can maximize performance gains.
- **Parallelization**: It is possible to parallelize the operations of the cache-aware algorithm by leveraging multi-core processing. Additionally, evaluating the utilization of parallelization frameworks such as MapReduce could facilitate the efficient processing of extensive graph datasets by distributing computations across multiple machines. Moreover, exploring the potential for parallelizing the algorithm on GPUs or other specialized hardware accelerators could lead to significant performance improvements.
- **Algorithmic Optimizations and Data Structures**: Further optimization of the existing algorithms and exploration of new data structures could lead to substantial performance gains. Investigating the use of concurrent data structures in multi-threaded environments could also enhance the execution speed of triangle counting algorithms.

Bibliography

[1] Jonathan W Berry et al. “Why do simple algorithms for triangle enumeration work in the real world?” In: *Proceedings of the 5th conference on Innovations in theoretical computer science*. 2014, pp. 225–234.

[2] Thomas Bläsius et al. “Efficiently generating geometric inhomogeneous and hyperbolic random graphs”. In: *Network Science* 10.4 (2022), pp. 361–380.

[3] Fan Chung and Linyuan Lu. “The average distances in random graphs with given expected degrees”. In: *Proceedings of the National Academy of Sciences* 99.25 (2002), pp. 15879–15882.

[4] Charo I Del Genio et al. “Efficient and exact sampling of simple graphs with given arbitrary degree sequence”. In: *PloS one* 5.4 (2010), e10012.

[5] Xiaocheng Hu, Yufei Tao, and Chin-Wan Chung. “Massive graph triangulation”. In: *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*. 2013, pp. 325–336.

[6] Joel C Miller and Aric Hagberg. “Efficient generation of networks with given expected degrees”. In: *International workshop on algorithms and models for the web-graph*. Springer. 2011, pp. 115–126.

[7] Rasmus Pagh and Francesco Silvestri. “The input/output complexity of triangle enumeration”. In: *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 2014, pp. 224–233.

[8] Siddharth Suri and Sergei Vassilvitskii. “Counting triangles and the curse of the last reducer”. In: *Proceedings of the 20th international conference on World wide web*. 2011, pp. 607–614.

[9] Jeffrey Scott Vitter et al. “Algorithms and data structures for external memory”. In: *Foundations and Trends® in Theoretical Computer Science* 2.4 (2008), pp. 305–474.