Efficient and Accurate Triangle Count Estimation in Large Networks

Sophia Hubscher
Robert and Donna Manning College of Information and Computer Sciences
University of Massachusetts Amherst
shubscher@umass.edu

Committee Chair: Professor Cameron Musco cmusco@cs.umass.edu
Second Committee Member: Professor Ghazaleh Parvini gparvini@cs.umass.edu
Research Type: Thesis

Abstract

ABSTRACT HERE

Contents

1 Introduction

Counting triangles is a fundamental problem in graph theory with widespread applications in social networks, bioinformatics, and more [11]. These triangles, are formed by three mutually connected nodes, as shown in Figure 1, which contains three triangles. While these triangles appear simple, they are a powerful structural motif that can reveal important insights the networks they are found in.

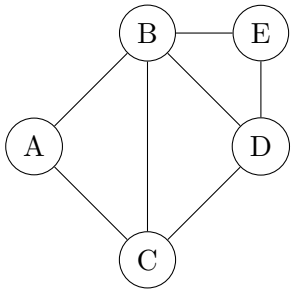

Figure 1: Graph with triangles formed between vertices (A, B, C), (B, C, D) and (B, D, E).

These triangles are more than just theoretical constructs. In social network graphs, for example, they can represent closed friendships or tightly-knit groups, signaling levels of local connectivity in a network. This, in turn, can reflect greater patterns and structures within a network. For example, in social media platforms, triangles are used to model relationships between users, where closed triangles indicate strong communities or mutual interests. A study analyzing the effect of recommender systems on X (formerly Twitter) demonstrated how an increase in closed triangles following the introduction of a "Who to Follow" friend-to-friend recommendation algorithm served as evidence of the algorithm's efficacy [17].

Additionally, triangles can be used to understand relationships within biological networks. For example, a study on yeast protein interaction networks used analysis of triangles to find transitive relationships between genes and proteins [22]. The researchers constructed graphs called "genetic congruence networks," connecting genes that shared similar interaction partners. These networks showed a higher-than-expected occurrence of triangles, indicating a strong correlation between genetic congruence and protein interactions. This suggests that triangles can capture important structural patterns, such as proteins that function within the same biological pathway or complex. Like in the case of social network analysis, this example illustrates how triangle metrics are not just useful for theoretical analysis but also for practical applications.

While the utility of triangle-based metrics is well-documented, counting triangles efficiently in large graphs remains computationally challenging. Direct enumeration methods involve inspecting all possible triples of nodes in the graph, a process with a worst-case time complexity of $O(n^3)$ where n is the number of nodes [1]. On smaller networks, this runtime may not pose issues, but unfortunately, for large graphs, especially sparse ones, where the number of edges is much smaller compared to the number of possible edges (as illustrated in Figures 2 and 3), efficiently counting these triangles poses significant computational challenges.

As graphs grow larger and more complex, direct methods for counting triangles become increasingly time-consuming, making it difficult to handle graphs of practical size in real-world applications. This issue is particularly relevant in the era of big data, where networks of millions or even billions of nodes and edges are common, and computational efficiency is critical.

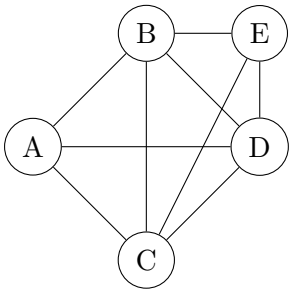

Figure 2: Dense Graph with many edges relative to the number of nodes.

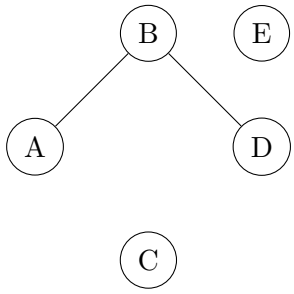

Figure 3: Sparse Graph with few edges relative to the number of nodes.

To address these challenges, researchers have developed a variety of approaches to count triangles efficiently. Some deterministic methods outlined in more detail in the background section of this proposal decrease the time it takes to compute global triangle counts [16]. However, these methods still face scalability issues. As a result, randomized algorithms [15, 18, 19] have emerged as a promising alternative. By leveraging probabilistic techniques, these algorithms provide approximate triangle counts with significant reductions in runtime while maintaining a high degree of accuracy.

Thus, this thesis aims to use randomized algorithms to find new, fast, accurate ways to estimate triangle counts that can be used in real-world applications.

2 Notation

Table 1: List of notation used.

Symbol	**Description**
$G(V, E)$	Graph with V vertices and E edges.
$n = \|V\|$	Number of vertices in graph G.
$m = \|E\|$	Number of edges in graph G.
A	The adjacency matrix for the graph G.
Δ_i	Number of triangles node i participates in.
Δ	Total number of triangles in G.
d_i	Degree of node i.

3 Background

3.1 Types of Graphs

In graph theory, graphs are classified as either directed or undirected. An *undirected graph* is one in which edges have no specific direction, so the relationship between connected nodes is mutual: If u connects to v, then v connects to u. In contrast, a *directed graph*, or digraph, has edges with a defined direction—u may point to v without v pointing to u. This directional property is particularly relevant when calculating triangle counts, as a triangle in a directed graph can follow a specific directional sequence. In this discussion, we generally refer to undirected graphs unless otherwise specified, although the methods described can be extended to directed graphs as well.

3.2 Graph Motifs

Graph motifs are subgraphs that occur frequently within a larger graph and carry significant structural information. One such motif is the *clique*, a subset of vertices such that every pair of vertices is connected by an edge. Triangles, for example, are a clique, as they consist of 3 mutually-connected nodes. Another common example of a clique is the *4-clique*, denoted as K_4, which consists of four vertices with edges connecting each pair of vertices.

Formally, we define a 4-clique, K_4, as the complete graph on four vertices, meaning every pair of vertices is connected by an edge. The set of vertices V and edges E for K_4 are given by:

$$K_4 = (V, E)$$

where

$$V = \{v_1, v_2, v_3, v_4\}, \quad E = \{(v_i, v_j) \mid 1 \leq i < j \leq 4\}.$$

Figure 4 illustrates the structure of K_4, where each vertex is connected to every other vertex, representing the maximal level of connectivity between four vertices.

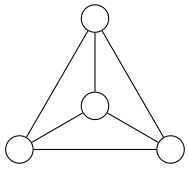

Figure 4: The complete graph K_4 on four vertices.

3.3 Methods for Triangle Counting

Triangle counting can be approached in a variety of ways, each with its own advantages and disadvantages. One of the simplest methods is the brute force technique, where all distinct sets of three vertices u, v, w are enumerated and checked for the existence of a triangle. This involves examining every possible combination of vertices in the graph and testing whether all three edges (u, v), (v, w), and (w, u) exist.

Assuming optimal conditions with edges stored in a hash table, where edge retrieval takes $O(1)$ time, the time complexity of this brute force approach is $\Theta(n^3)$. This complexity stems from the fact that $\binom{n}{k} = \Theta(n^k)$, and thus, $\binom{n}{3} = \Theta(n^3)$ [1].

While this method is straightforward, it is inefficient for large graphs due to its high computational cost. Additionally, this method is no more efficient on sparse graphs (those with relatively few edges compared to the maximum number of edges possible) than on dense ones, which is another area for improvement. Thus, researchers have turned to alternative triangle counting and estimation methods.

3.3.1 Sampling Methods

One of the most effective ways to estimate triangle counts in large, sparse graphs is through sampling methods. These methods rely on randomly selecting edges or vertices and then inspecting their local neighborhoods for the presence of triangles. Sampling-based techniques are particularly useful in scenarios where calculating the exact triangle count is computationally expensive or unnecessary.

Additionally, sampling algorithms often provide tunable accuracy, allowing for a trade-off between precision and performance, making them ideal for processing large-scale networks.

Edge Sampling

In edge sampling, we randomly sample a subset of edges from the graph, count the number of triangles in the subgraph, and scale up to reach our estimate.

One key edge sampling algorithm is Doulion [19], in which each edge in our graph G is sampled with probability p. As all triangles consist of three edges, this means that all triangles in G have probability p^3 of being counted. Thus, the number of triangles counted is scaled by $\frac{1}{p^3}$ to achieve a final estimate.

Other algorithms extend this even further. For example, a parallel implementation of Doulion [3], where each processor independently sparsifies its assigned partition of the graph, can improve accuracy.

In all of these algorithms though, the key piece of their efficiency and efficacy is the sampling of edges to get a good picture of the graph's structure without counting every triangle individually.

Wedge Sampling

Wedge sampling [15] focuses on estimating wedges—triplets of nodes that form two edges but not necessarily a triangle. A wedge is defined by three vertices (u, v, w) where u is adjacent to both v and w, but v and w may or may not be adjacent (see Figure 5).

First, the algorithm counts the total number of wedges in the graph. To count these wedges, only one pass over all nodes is required, as at each node, every unique pair of outgoing edges from the node is counted as a single wedge. Thus, this operation takes $O(m)$ time where m is the number

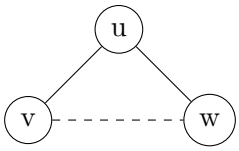

Figure 5: Wedge formed by vertices u, v, and w. Nodes v and w may or may not be connected.

of edges in G.

Once wedges are sampled, the algorithm checks how many of them are closed (i.e., form triangles). The number of triangles can then be estimated by multiplying the number of total wedges by the fraction of all wedges that were closed in the sample. Wedge sampling tends to work well in graphs with a large number of high-degree vertices, where it becomes easier to sample many wedges at once, but unlike edge sampling, it cannot be efficiently done using data structures like adjacency matrices or adjacency lists. Thus, wedge sampling comes with an additional preprocessing step that adds to runtime.

3.3.2 Linear Algebraic Methods

Along with sampling, we can employ linear algebraic techniques to increase the speed of our triangle counting.

Graphs can be conveniently represented using adjacency matrices, which, in social network analysis, are typically referred to as *sociomatrices* [5]. In these matrices, each row and column represents a node, and edges between nodes are represented as 1s in the corresponding matrix entry.

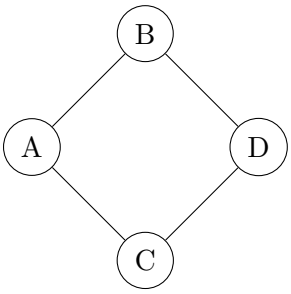

Figure 6: Graph representation of vertices A, B, C, and D.

$$A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

Figure 7: Adjacency matrix corresponding to the graph.

By using these adjacency matrices and leveraging linear algebra techniques, we can calculate triangle counts more efficiently. One simple method using the adjacency matrix is to use the following formula where A is the adjacency matrix corresponding to the graph G and Δ is the global triangle count in G:

$$\Delta = \frac{1}{6}\text{trace}(A^3)$$

This formula is derived from the fact that the diagonal elements of A^3 count the number of length-three paths (i.e. triangles) that each vertex participates in. Each triangle can be formed from six of these length-three paths, as each triangle can be drawn starting at any of its three nodes and moving either clockwise or counter-clockwise, as illustrated in Figure 8. Thus, the trace of A^3 is divided by six to scale down to the global triangle count.

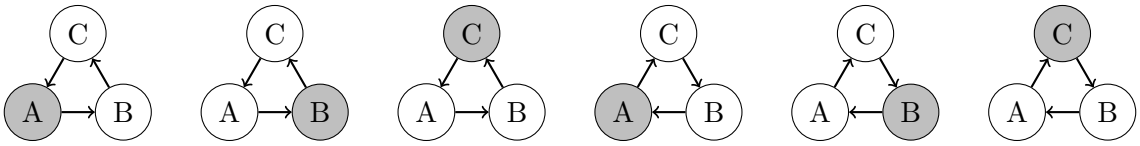

Figure 8: Six different ways to arrive at a length-three path in a triangle.

To compute A^3, we first need to calculate A^2 (which takes $O(n^3)$ for an $n \times n$ matrix, n thus also being the number of nodes in our graph G) and then multiply A^2 by A (also $O(n^3)$). Thus, the total complexity for computing A^3 is $O(n^3)$. After computing A^3, calculating the trace takes $O(n)$, as we need to iterate over the n diagonal elements. Thus, the overall runtime complexity for the operation is $O(n^3)$. While this is not a direct improvement over the runtime of the naive algorithm, this strategy forms the basis of many faster methods.

Strassen's Algorithm

This runtime analysis above assumes that matrix multiplication is performed using the standard algorithm. However, more sophisticated techniques, such as Strassen's algorithm [16], can reduce matrix multiplication time. In this algorithm, that is used on large, square matrices, such as undirected sociomatrices, each matrix is divided into smaller submatrices on which a series additions and multiplications are performed.

Specifically, Strassen's algorithm reduces the complexity of multiplying two $n \times n$ matrices to approximately $O(n^{\log_2 7})$, which is about $O(n^{2.81})$. Computing A^2 using Strassen's algorithm will take $O(n^{\log_2 7})$. Then, multiplying A^2 by A again takes $O(n^{\log_2 7})$. Therefore, the total complexity for computing A^3 with Strassen's algorithm is $O(n^{\log_2 7}) + O(n^{\log_2 7}) = O(n^{\log_2 7})$, or roughly $O(n^{2.81})$.

To contextualize this, on a 2×2 matrix, the n^3 multiplications required for the naive method would mean we would complete $2^3 = 8$ multiplications. With Strassen's method and its $n^{\log_2 7}$ multiplications, there would instead only be $2^{\log_2 7} = 7$ multiplications computed. On larger matrices, this leads to a significant speedup.

There are matrix multiplication algorithms that are even faster, such as one with a $O(n^{2.371552})$ runtime, but this algorithm relies on the use of extremely large constants and is thus rarely used in real-world applications [21].

EigenTriangle Algorithm

Another significant approach in triangle counting is the use of spectral methods. One such method is the EigenTriangle algorithm [18], which estimates the triangle count Δ by considering the spectral

decomposition of the adjacency matrix A.

The EigenTriangle algorithm is based on the observation that the number of triangles in a graph is closely related to the spectrum of its adjacency matrix. In particular, the adjacency matrix A is decomposed as:

$$A = U\Lambda U^T,$$

where U is a matrix whose columns are the eigenvectors of A, and Λ is a diagonal matrix containing the corresponding eigenvalues.

Once the decomposition is performed, the number of triangles can be computed exactly using $\Delta = \frac{1}{6}\sum_{i=1}^{n}\lambda_i^3$, and can be estimated using:

$$\Delta \approx \frac{1}{6}\sum_{i=1}^{k}\lambda_i^3,$$

where λ_i are the k eigenvalues of largest magnitude of the adjacency matrix A. The runtime of EigenTriangle is dominated by the cost of approximating the top k eigenvalues and eigenvectors of A, which, using the Lanszos method [7], can be done in roughly $O(km)$ time, where m is the number of edges and k is typically much smaller than the number of nodes n. This is a significant improvement over the runtimes of direct methods.

Specifically, for the direct method in which was compute the trace of A^3, we know $trace(A^3) = \sum_{i=1}^{n}\lambda_i(A^3) = \sum_{i=1}^{n}\lambda_i^3$. Thus, we see that EigenTriangle approximates $trace(A^3)$. Given this, it makes sense that this runtime is a substantial improvement over the complexity of computing trace(A^3) directly.

TraceTriangle Algorithm

The TraceTriangle algorithm [4] is a randomized algorithm designed for efficient triangle estimation in large graphs. It leverages trace-based techniques, which compute the trace of matrix powers to approximate the number of triangles. Specifically, the algorithm relies on the previously mentioned property: $\Delta = \frac{1}{6}\text{trace}(A^3)$, where A is the adjacency matrix of the graph and Δ is the number of triangles. However, rather than computing the full matrix multiplication A^3, which is computationally expensive for large graphs, the TraceTriangle algorithm uses a randomized approach to approximate this trace, significantly reducing computation time.

This randomized approach is based on Hutchinson's method [10], which is a technique for estimating the trace of a matrix by randomly sampling vectors. In this case, this significantly reduces computation time by approximating trace(A^3) through randomized sampling rather than explicit computation.

The TraceTriangle algorithm is a sampling algorithm, and thus, it's runtime depends on the desired accuracy of output, as more or fewer samples can be taken depending on the application. Generally though, experiments demonstrate that typically $O(log^2|n|)$, where n is the number of vertices in G,

samples are required to get good approximations on real-world graphs, and regardless of application, the runtime for taking each sample is $O(|m|)$, where m is the number of edges in G.

Comparing TraceTriangle to the EigenTriangle algorithm, TraceTriangle achieves higher accuracy across multiple types of graphs [4]. Despite this accuracy advantage, EigenTriangle tends to run more quickly than TraceTriangle on large graphs. That said, one advantage of TraceTriangle is its potential for parallelization. This allows TraceTriangle to scale effectively with the size of the graph, ultimately reducing the speed advantage of EigenTriangle in larger computations.

3.4 General Algorithmic Strategies

Beyond specific algorithms for triangle counting, various general techniques from theoretical computer science have been adapted for this problem, particularly in designing faster algorithms.

3.4.1 Variance Reduction

Variance reduction [14] is another general technique that can be applied to triangle estimation, improving accuracy without increasing the number of samples needed.

Variance reduction methods aim to reduce the spread (or variance) of estimations, leading to more reliable results even with fewer samples. This is particularly important in large-scale graphs, where taking a high number of samples may be computationally infeasible.

In terms of triangle counting, this method can be applied by finding a fast way of estimating the global triangle count, and then using sampling to estimate the error on that count. Specifically, we can begin by finding a relationship between the degree of nodes and the number of triangles they are involved in. This can be done by plotting nodes' degrees (d_i) versus triangle counts (Δ_i) on a log-log plot, finding a line of best fit, and then exponentiating as follows:

$$\log(\Delta_i) \approx \alpha \cdot \log(d_i) + \beta$$
$$\Delta_i \approx d_i^{\alpha} \cdot e^{\beta} = m_i.$$

Now, using this equation, we can estimate the overall triangle count M by applying this line of best fit relationship to all nodes in the graph:

$$M = \sum_{i=1}^{n} m_i.$$

Next, we sample our graph to get s nodes, with s being our sample size. For each of these s sampled nodes, we count the number of triangles they are involved in (written Δ_i) and find the difference between those actual triangle counts and their estimated triangle counts using the line of best fit relationship. We then take the sum of these errors and scale them up to estimate the error on our global triangle count (written E). Mathematically, this can be expressed as follows:

$$E = \left(\sum_{i=1}^{s} \Delta_i - m_i \right) \cdot \frac{n}{s}.$$

Lastly, we take the sum of our estimate and our error, and divide this sum by three to avoid triple-counting triangles, as each triangle has three nodes it is involved in:

$$\Delta \approx \frac{M + E}{3}.$$

Thus, by applying this variance reduction technique, we arrive at an estimate for the triangle count Δ.

3.4.2 Importance Sampling

One example of a variance reduction method is importance sampling. When estimating a metric relating to a large population using uniform sampling, where all edges/nodes/wedges/etc. are sampled with the same probability, often a very large number of samples is required to ensure a good relative approximation [11]. This is because uniform sampling does not prioritize areas of the graph that may have a disproportionately large impact on the estimate. Consequently, the computational cost can be high for achieving a desired accuracy level in many cases.

When using importance sampling [13], the process is improved by sampling higher-interest nodes with higher probability, focusing computational effort on the most "important" parts of a graph. The key idea behind importance sampling is to bias the sampling distribution towards more informative areas of the graph. For instance, in a graph where certain nodes are highly connected or play a critical role in the overall structure, importance sampling would prioritize these nodes to reduce the variance of the estimates.

Importance sampling can also be applied to triangle counts. For example, we can prioritize high-degree nodes as the most "important." The weight of this importance is decided by some power α greater than 1 (which is equivalent to uniform sampling). This α can be tuned to indicate different strengths of relationships between the degree and triangle counts of nodes.

Once α has been selected, we use it to ascribe each node a probability p_i to each node based in its degree d_i:

$$D = \sum_{i=1}^{n} d_i^{\alpha}$$

$$p_i = \frac{d_i^{\alpha}}{D}.$$

Next, we sample s nodes based on their probabilities p_i. For example if $p_1 = 0.01$ and $p_2 = 0.1$, we are 10 times more likely to sample node 2 than node 1.

Next, for each sampled node we count the number of triangles it is a part of, and then scale that count by $\frac{1}{s \cdot p_i}$. The sum of all these counts, scaled down by three (as to avoid triple-counting triangles), is our estimate for the global triangle count Δ.

3.4.3 Learning-Augmented Algorithms

A learning-augmented algorithm [12] is an algorithm that uses a prediction to boost its performance. Whereas most algorithms take only the problem their input, learning-augmented algorithms also accept an extra piece of information—usually a prediction about some part of the solution. The algorithm then uses this prediction to run faster or produce better results.

An example of a learning-augmented algorithm is its use in the maximum weight matching problem. The maximum weight matching problem [9] is the problem of finding a matching in which the sum of weights is maximized in a weighted graph. The typical solution for this problem, the Hungarian algorithm, runs in $O(m\sqrt{n})$ time.

When a learning-augmented approach [8] is applied however, where machine-learned predictions are used to "warm-start" the algorithm, that runtime is significantly reduced when the predictions are accurate. When the predictions are inaccurate, the runtime is simply the same as in the Hungarian algorithm.

This technique can be applied to triangle counting too. For example, Tonic [6], a learning-augmented algorithm for counting triangles in graph streams, leverages predictions about edge "heaviness" (i.e., the number of triangles they are involved in) to improve the accuracy and speed of triangle counting. Tonic combines these predictions with sampling methods to keep track of the most relevant edges. This allows the algorithm to focus on the edges that are most likely to contribute to the triangle count.

Notably, Tonic provides unbiased estimates of triangle counts regardless of the accuracy of the predictor. However, when the predictor provides useful information on heavy edges, the algorithm produces estimates with reduced variance compared to state-of-the-art alternatives.

In general, this method can be highly effective, as accurate predictions can significantly enhance algorithms' efficiency or result quality.

4 Methods

To evaluate different triangle count estimation methods, I implemented them in Python and compared their accuracies, runtimes, and sample sizes on a diverse set of networks. The primary methods implemented were uniform sampling, importance sampling, a variance reduction method, and a hybrid method that combines elements of these approaches.

The implemented methods were applied to both synthetic and real-world networks sourced from the Stanford Network Analysis Platform (SNAP) library. These networks span a range of sizes, densities, and structural properties to ensure robust comparisons.

4.1 Implementation

Algorithms were implemented in Python using NetworkX and SNAP, and experiments were run on consistent hardware, with multiple trials to ensure reliable comparisons.

4.2 Datasets

The datasets used in this study include both synthetic networks and real-world graphs from the SNAP library, covering various domains such as social networks, collaboration networks, and web graphs. Specifically, methods were evaluated on these three networks:

- Social Network: Social circles (or 'friends lists') are represented in this graph, where each user is a node, and each edge is a friendship between users.
- Collaboration Network: This graph represents a network of co-authorships from the General Relativity and Quantum Cosmology collaboration network, where nodes represent authors and edges indicate co-authored papers.
- Wikipedia Article Network: This graph represents a network of Wikipedia articles relating to crocodiles, where nodes represent articles and edges represent links between them.

Synthetic networks were generated using the NetworkX library to create Barabási–Albert [2] and Watts-Strogatz [20] graphs. Barabási–Albert graphs model preferential attachment, exhibiting power-law degree distributions, while Watts–Strogatz graphs represent small-world properties with adjustable clustering and path lengths. These synthetic datasets served as controlled environments to evaluate the scalability and accuracy of the methods under varying structural parameters.

To assess accuracy, ground-truth triangle counts were computed using exact algorithms and compared against the approximate counts produced by each estimation method. The evaluation metrics included accuracy, runtime efficiency, and the sample size required to achieve a specified error margin. By examining performance across different graph characteristics, the study identifies the strengths and weaknesses of each method in diverse scenarios.

4.3 Sampling Methods Evaluated

Uniform Sampling:

This baseline method involves randomly sampling edges or nodes and scaling up the observed triangle counts proportionally. While simple, uniform sampling often struggle in graphs with skewed degree distributions.

Code for this method is given below, where A is the adjacency matrix of our graph G and s is the number of nodes of G being sampled.

```
import numpy as np
import random

def gen_s_ints(s, n):
    choice_arr = sorted(random.choices(range(n), k=s))
    return choice_arr

def estimate_uniformly_per_node_method(A, s):
    n = len(A)
    sampled_nodes = gen_s_ints(s, n)

    triangle_count = 0
    for i in sampled_nodes:
        triangle_count += count_node_triangles(A, i)

    return triangle_count * (n / s) // 3
```

Importance Sampling:

In this method, we attempt to prioritize the sampling of nodes that are likely to form more triangles. This is done by sampling proportionally to the degree of nodes. This method aims for higher accuracy with fewer samples.

As written in section 3.4.2, in importance sampling, after we select a value α to indicate the strength of the relationship between the degree and triangle count of nodes, we can use this value α to ascribe each node a probability p_i to each node based in its degree d_i:

$$D = \sum_{i=1}^{n} d_i^{\alpha}$$

$$p_i = \frac{d_i^{\alpha}}{D}.$$

We can then sample each node n_i with probability p_i. The code for this sampling and estimation process is given below where, like before A is the adjacency matrix of G and s is our sample size. We also introduce the parameter α, which in the code snippet, is represented by the variable "power".

```
def sample_by_degree(A, n, s, power):
    degrees = np.sum(A, axis=1)
    degrees_to_power = np.power(degrees, power)
```

```
    sum_of_degrees_to_power = np.sum(degrees_to_power)
    probabilities = degrees_to_power / sum_of_degrees_to_power

    sampled_nodes = random.choices(range(n), weights=probabilities, k=s)

    return probabilities, sampled_nodes

def importance_estimate_per_node_method(A, s, power):
    n = len(A)
    probabilities, sampled_nodes = sample_by_degree(A, n, s, power)

    estimate = 0
    for i in sampled_nodes:
    triangle_count = count_node_triangles(A, i)
    estimate += triangle_count * (1 / (s * probabilities[i]))

    return estimate // 3
```

Variance Reduction:

Our next method is variance reduction, which aims minimize the variability of our estimates. The high variance in uniform or importance sampling methods can lead to inaccurate triangle count estimations, particularly in graphs with skewed degree distributions. To address this, we leverage additional information about the relationship between node degrees and triangle counts.

In many real-world graphs, nodes with higher degrees tend to form more triangles, suggesting a potential correlation between the log of the degree and the log of the triangle count. Thus, by performing a linear regression in log-log space, we can obtain a line of best fit that estimates triangle counts as a function of node degree, and use that to arrive an an estimate of our triangle count.

However, the number arrived at using our linear regression may either over- or under-count the number of triangles depending on the relationship between degree and triangle count. Thus, we sample s additional nodes and calculate the difference between their actual triangle counts and those predicted by our line of best fit. Then, scaling this up, we can combine our predicted triangle count and predicted error to arrive at a final estimate of the triangle count Δ.

The specific math for this method is given in section 3.4.1 and the code for it is given below.

```
def get_line_of_best_fit(A):
    n = len(A)
    degrees = np.sum(A, axis=1)
    triangles = np.zeros(n)

    for i in range(n):
        triangles[i] = count_node_triangles(A, i)

    valid_indices = (degrees > 0) & (triangles > 0)
    filtered_degrees = degrees[valid_indices]
    filtered_triangles = triangles[valid_indices]

    log_degrees = np.log(filtered_degrees)
    log_triangles = np.log(filtered_triangles)

```

```
    slope, intercept = np.polyfit(log_degrees, log_triangles, 1)

    return slope, intercept

def estimate_variance_reduction_method(A, s, power):
    n = len(A)

    slope, intercept = get_line_of_best_fit(A)

    # obtain M from the line of best fit
    degree_array = np.sum(A, axis=1)
    approx_triangles = np.power(degree_array, slope) * np.exp(intercept)
    M = np.sum(approx_triangles)

    if s == 0:
        return M // 3

    # sample s nodes, and use them to adjust our estimate M
    sampled_nodes = gen_s_ints(s, n)
    sampled_node_triangles = np.array([count_node_triangles(A, i) for i in
    sampled_nodes])

    sampled_m_i_vals = np.array([approx_triangles[i] for i in sampled_nodes])

    D = np.sum(sampled_node_triangles - sampled_m_i_vals) * (n/s)

    return (M + D) // 3
```

Hybrid Approach:

The hybrid approach combines elements from both importance sampling and variance reduction techniques. It functions by running the variance reduction method as before, but using importance sampling instead of uniform sampling to generate the s nodes used in correcting our triangle count estimate.

The differences between the following code snippet and that in variance reduction begins on line 13.

```
def estimate_importance_variance_reduction_method(A, s, power):
    n = len(A)

    slope, intercept = get_line_of_best_fit(A)

    degree_array = np.sum(A, axis=1)
    approx_triangles = np.power(degree_array, slope) * np.exp(intercept)
    M = np.sum(approx_triangles)

    if s == 0:
        return M // 3

    probabilities, sampled_nodes = sample_by_degree(A, n, s, power)
    sampled_node_probabilities = np.array([probabilities[i] for i in sampled_nodes
    ])
    sampled_node_triangles = np.array([count_node_triangles(A, i) for i in
    sampled_nodes])

    sampled_m_i_vals = np.array([approx_triangles[i] for i in sampled_nodes])
```

```
18
19      D = np.sum((sampled_node_triangles - sampled_m_i_vals) * (1 / (s *
        sampled_node_probabilities)))
20
21      return (M + D) // 3
```

4.4 Evaluation Metrics

To assess the performance of the triangle count estimation methods, the following evaluation metrics were measured on real-world and synthetic networks:

- Accuracy: The difference between the estimated and true triangle counts, calculated as the relative error.
- Runtime: The computational time required to generate triangle count estimates.
- Sample size: The number of nodes or edges sampled to produce an estimate.
- Variance: The variability in triangle count estimates across multiple independent runs of the same method.

4.5 Additional Explorations

4.5.1 4-Clique Variant

To evaluate how well these methods generalize, the code for each algorithm was adapted to count 4-cliques. To adapt the previous methods to count 4-cliques, we simply replace the `count_node_triangles()` method with a `count_node_4_cliques()` method, and scale final results by 4 instead of by 3.

4.5.2 Simulated Method Tests

In order to learn more about when importance sampling and variance reduction perform best, I ran these algorithms on manually defined triangle-degree relationships.

The methods were tested on two different degree-triangle relationships. First, I set a variable `degrees` to be the degree distribution of a real-world dataset. I also defined `noise` as `noise = np.random.normal(0, noise_scale, triangles.shape)`. This noise is normally distributed with a mean of zero and a standard deviation of `noise_scale`, allowing for random fluctuations around the expected triangle and maintaining a symmetric distribution.

The two relationships defined using these degrees and noise values are the **Gaussian noise** and **multiplicative noise** relationships. For the first, artificial triangle counts for the degree distributions were set to `np.power(degrees, slope) * np.exp(intercept) + noise` where the slope and intercept are the outputs from the `get_line_of_best()` function defined in section 4.3. Thus, in this relationship, noise does not grow with degree.

For the second relationship, the **multiplicative noise** relationship, the artificial triangle counts were set to `np.power(degrees, slope) * np.exp(intercept) * (1 + noise)` where the slope, intercept, and noise are defined as before. In this relationship, noise grows as the degree does too, distinguishing this degree-triangle relationship from the former.

These noise models allow us to analyze how different types of randomness impact the performance of variance reduction and importance sampling techniques.

5 Results

5.1

5.2 4-Clique Counting

5.3 Simulated Method Tests

To assess the performance of importance sampling and variance reduction, we tested these algorithms under two different triangle-degree relationships: Gaussian noise and multiplicative noise. In this section, we present the results of these tests, focusing on the relationship between sample size and average error for both types of noise across various scales.

5.3.1 Gaussian Noise Relationship

In this subsection, we examine the performance of the algorithms with Gaussian noise. The degree distribution was derived from the Facebook dataset, and the triangle counts were generated using the formula:

$$\text{triangle counts} = \text{np.power(degrees, slope)} \times \text{np.exp(intercept)} + \text{noise},$$

where noise is normally distributed with mean zero and standard deviation defined by three scales: 10, 100, and 500. The following figures display the relationship between sample size and average error for each of these noise scales.

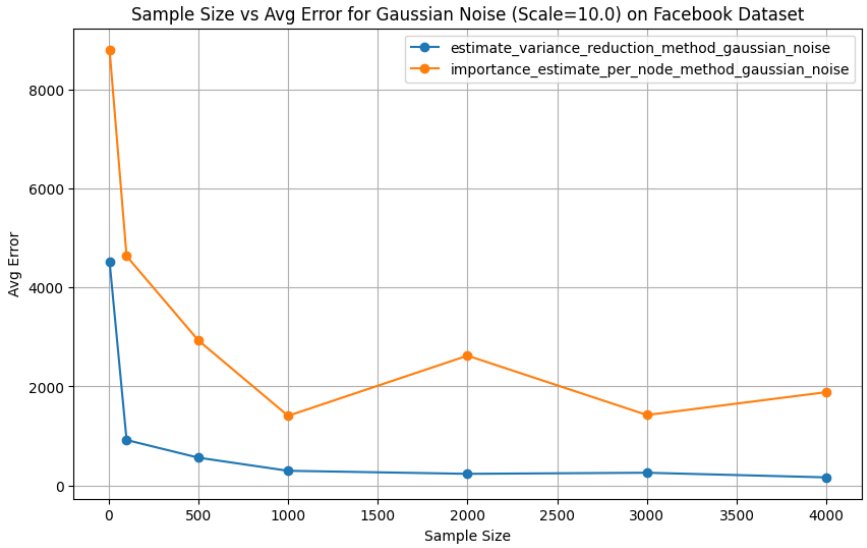

Figure 9: Sample size vs. average error with Gaussian noise (scale = 10).

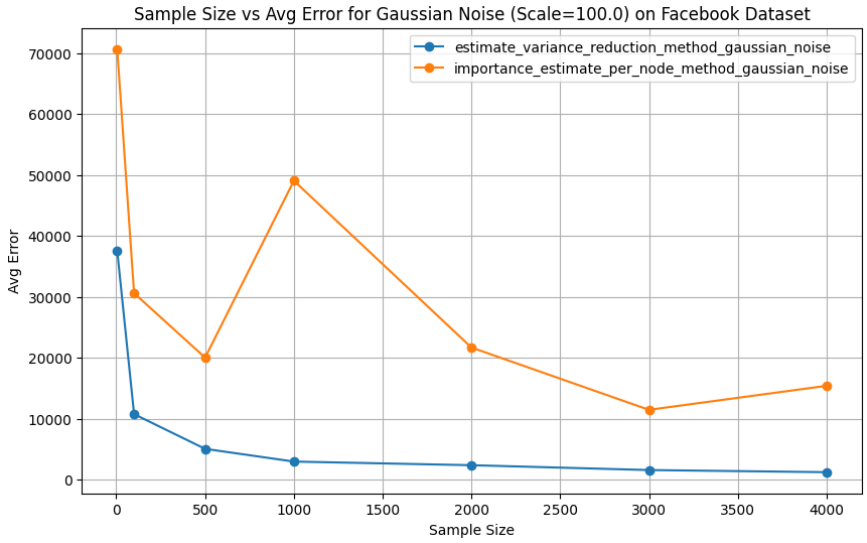

Figure 10: Sample size vs. average error with Gaussian noise (scale = 100).

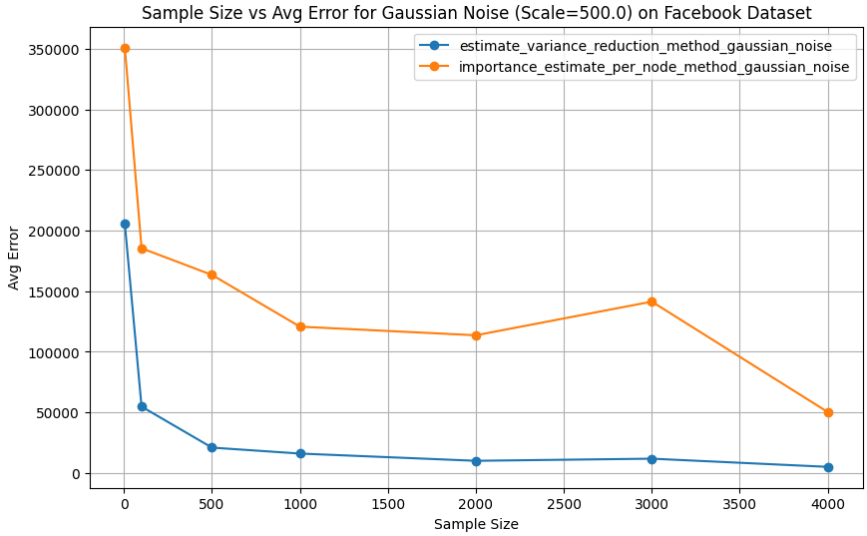

Figure 11: Sample size vs. average error with Gaussian noise (scale = 500).

5.3.2 Multiplicative Noise Relationship

Next, we consider the performance of the algorithms with multiplicative noise. The triangle counts were generated using the following formula:

$$\text{triangle counts} = \text{np.power(degrees, slope)} \times \text{np.exp(intercept)} \times (1 + \text{noise}),$$

where noise is again normally distributed and grows with the degree, and the noise scale was tested at three values: 0.01, 0.1, and 0.5. Below are the figures for sample size vs. average error for each noise scale in the multiplicative noise case.

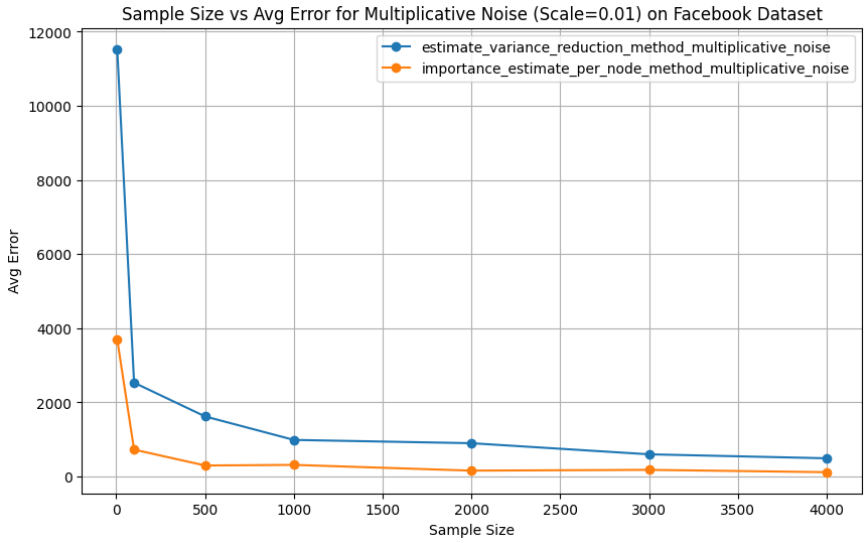

Figure 12: Sample size vs. average error with multiplicative noise (scale = 0.01).

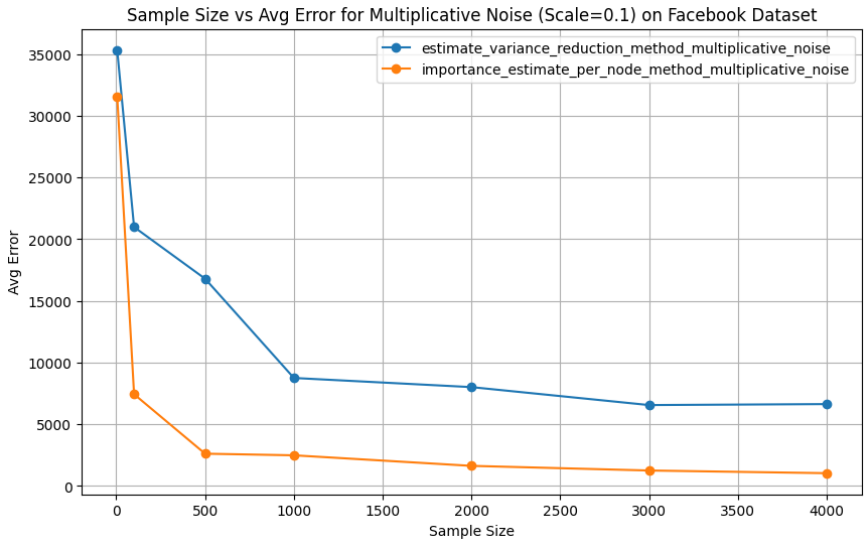

Figure 13: Sample size vs. average error with multiplicative noise (scale = 0.1).

6 Discussion

6.1 Theoretical Analysis of Variances

Understanding the variance of the algorithms tested is crucial for evaluating their reliability. Thus, in this section, we derive expressions for the variance of the estimated triangle count across methods.

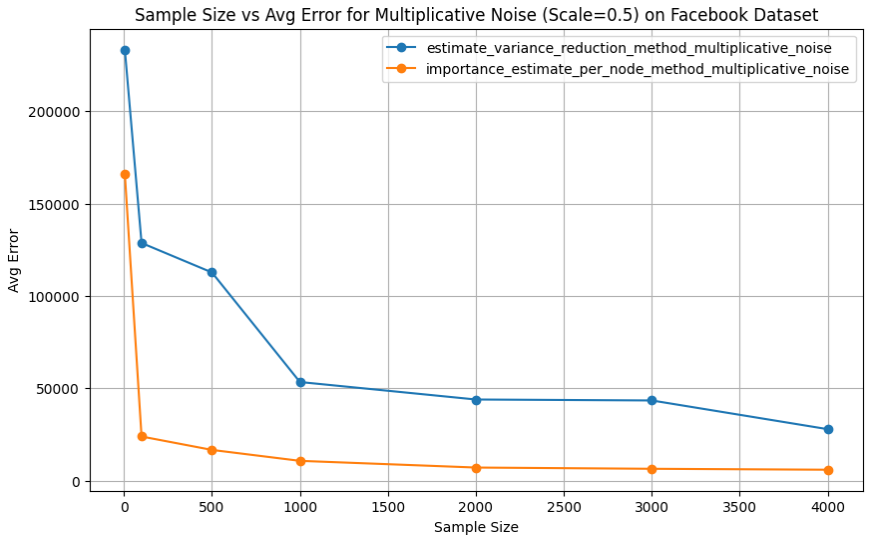

Figure 14: Sample size vs. average error with multiplicative noise (scale = 0.5).

6.1.1 Uniform Sampling

The global triangle count Δ can be estimated as $\tilde{\Delta}$ using the following formula:

$$\tilde{\Delta} = \frac{n}{3s} \sum_{i=1}^{s} \Delta_i,$$

where n is the number of nodes in our graph G, s is our sample size, and $\sum_{i=1}^{s} \Delta_i$ is the sum of all sampled triangle counts. Using this, we can find the variance of $\tilde{\Delta}$ in terms of n, s, and $\text{Var}(\Delta_i)$.

$$\begin{aligned}
\text{Var}(\tilde{\Delta}) &= \text{Var}\left(\frac{n}{3s} \sum_{i=1}^{s} \Delta_i\right) \\
&= \frac{n^2}{9s^2} \text{Var}\left(\sum_{i=1}^{s} \Delta_i\right) \\
&= \frac{n^2}{9s^2} \sum_{i=1}^{s} \text{Var}(\Delta_i)
\end{aligned}$$

Next, we find $\text{Var}(\Delta_i)$.

$$\text{Var}(\Delta_i) = \mathbb{E}[(\Delta_i - \mathbb{E}[\Delta_i])^2]$$

$$\mathbb{E}[\Delta_i] = 3(\frac{1}{n}\Delta_1 + \frac{1}{n}\Delta_2 + \ldots + \frac{1}{n}\Delta_n) = \frac{3\Delta}{n}$$

$$
\begin{aligned}
\mathbb{E}[(\Delta_i - \mathbb{E}[\Delta_i])^2] &= \mathbb{E}[(\Delta_i - \frac{3\Delta}{n})^2] \\
&= \sum_{i=1}^{n} \frac{1}{n}(\Delta_i - \frac{3\Delta}{n})^2 \\
&= \sum_{i=1}^{n} \frac{1}{n}(\Delta_i^2 - 6\Delta_i \frac{\Delta}{n} + \frac{9\Delta^2}{n^2}) \\
&= \frac{1}{n}\sum_{i=1}^{n} \Delta_i^2 - \frac{6}{n}\sum_{i=1}^{n} \Delta_i \frac{\Delta}{n} + \frac{1}{n}\sum_{i=1}^{n} \frac{9\Delta^2}{n^2} \\
&= \frac{1}{n}\sum_{i=1}^{n} \Delta_i^2 - \frac{6}{n}\sum_{i=1}^{n} \Delta_i \frac{\Delta}{n} + \frac{9\Delta^2}{n^2} \\
&= \frac{1}{n}\sum_{i=1}^{n} \Delta_i^2 - \frac{18\Delta^2}{n^2} + \frac{9\Delta^2}{n^2} \\
&= \frac{1}{n}\sum_{i=1}^{n} \Delta_i^2 - \frac{9\Delta^2}{n^2}
\end{aligned}
$$

Now, we can plug our value of $\text{Var}(\Delta_i)$ into $\text{Var}(\tilde{\Delta}) = \frac{n^2}{9s^2}\sum_{i=1}^{s} \text{Var}(\Delta_i)$.

$$
\begin{aligned}
\text{Var}(\tilde{\Delta}) &= \frac{n^2}{9s^2}\sum_{i=1}^{s}[\frac{1}{n}\sum_{i=1}^{n} \Delta_i^2 - \frac{9\Delta^2}{n^2}] \\
&= \frac{n^2}{9s^2} s[\frac{1}{n}\sum_{i=1}^{n} \Delta_i^2 - \frac{9\Delta^2}{n^2}] \\
&= \frac{n^2}{9s}[\frac{1}{n}\sum_{i=1}^{n} \Delta_i^2 - \frac{9\Delta^2}{n^2}] \\
&= \frac{n}{s}[\frac{1}{9}\sum_{i=1}^{n} \Delta_i^2 - \frac{\Delta^2}{n}] \\
&= \frac{n}{9s}\sum_{i=1}^{n} \Delta_i^2 - \frac{\Delta^2}{s}
\end{aligned}
$$

6.1.2 Variance Reduction

Across this section, we will use the fact that $\text{Var}[X+Y] \leq 2(\text{Var}[X] + \text{Var}[Y])$. We derive that as follows:

$$\begin{aligned}
\text{Var}[X+Y] &= \mathbb{E}[(X+Y)^2] \\
&= \mathbb{E}[X^2] + \mathbb{E}[Y^2] + 2\mathbb{E}[XY] \\
&\leq \mathbb{E}[X^2] + \mathbb{E}[Y^2] + (\mathbb{E}[X^2] + \mathbb{E}[Y^2]) \\
&= 2(\mathbb{E}[X^2] + \mathbb{E}[Y^2]) \\
&= 2(\text{Var}[X] + \text{Var}[Y])
\end{aligned}$$

Variance Reduction with Uniform Noise

Take the estimator of Δ_i to be denoted by M where $M = \sum_{i=1}^{n} m_i$ and $m_i = \Delta_i + N(0, \sigma^2)$.

$$\tilde{\Delta} = \sum_{i=1}^{n} m_i + \frac{n}{s} \sum_{i=1}^{s} (m_i - \Delta_i)$$

$$\begin{aligned}
\text{Var}(\tilde{\Delta}) &= \text{Var}(\sum_{i=1}^{n} m_i + \frac{n}{s} \sum_{i=1}^{s} (m_i - \Delta_i)) \\
&\leq 2 \sum_{i=1}^{n} \text{Var}(m_i) + 2\text{Var}(\frac{n}{s} \sum_{i=1}^{s} (m_i - \Delta_i)) \\
&= 2 \sum_{i=1}^{n} \text{Var}(m_i) + \frac{2n^2}{s^2} \text{Var}(\sum_{i=1}^{s} (m_i - \Delta_i)) \\
&= 2 \sum_{i=1}^{n} \text{Var}(m_i) + \frac{2n^2}{s^2} \sum_{i=1}^{s} \text{Var}(m_i - \Delta_i) \\
&= 2n\sigma^2 + \frac{2n^2}{s^2} \sum_{i=1}^{s} \text{Var}(m_i - \Delta_i) \\
&= 2n\sigma^2 + \frac{2n^2}{s^2} s\sigma^2 \\
&= 2n\sigma^2 + \frac{2n^2}{s} \sigma^2
\end{aligned}$$

To allow for better comparison with other methods, assume σ^2 is the average squared error when using the trivial predictor of $m_i = 0$. In this case, $\sigma^2 = \frac{\sum_{i=1}^{n} \Delta_i^2}{n}$.

Plugging this value for σ^2 into our result, we get:

$$\begin{aligned}
\mathrm{Var}(\tilde{\Delta}) &\leq 2n\sigma^2 + \frac{2n^2}{s}\sigma^2 \\
&= 2n\frac{\sum_{i=1}^{n}\Delta_i^2}{n} + \frac{2n^2}{s}\frac{\sum_{i=1}^{n}\Delta_i^2}{n} \\
&= 2\sum_{i=1}^{n}\Delta_i^2 + \frac{2n}{s}\sum_{i=1}^{n}\Delta_i^2 \\
&= 2\sum_{i=1}^{n}\Delta_i^2(1+\frac{n}{s})
\end{aligned}$$

This is the variance expression reached with the trivial predictor, but a better predictor $\sigma^2 = \frac{\sum_{i=1}^{n}\Delta_i^2}{n}\epsilon$ for some small value ϵ would lead to a far smaller variance of:

$$\mathrm{Var}(\tilde{\Delta}) \leq 2\sum_{i=1}^{n}\Delta_i^2\epsilon(1+\frac{n}{s})$$

Variance Reduction with Multiplicative Noise

Here, take the estimator of Δ_i to be denoted by M where $M = \sum_{i=1}^{n} m_i$ and $m_i = \Delta_i + \sigma_i^2$ with $\sigma_i^2 = \Delta_i^2\sigma^2$. i.e., as the size of Δ_i increases, so does the noise m_i.

$$\tilde{\Delta} = \sum_{i=1}^{n} m_i + \frac{n}{s}\sum_{i=1}^{s}(m_i - \Delta_i)$$

$$
\begin{aligned}
\mathrm{Var}(\tilde{\Delta}) &= \mathrm{Var}(\sum_{i=1}^{n} m_i + \frac{n}{s}\sum_{i=1}^{s}(m_i - \Delta_i)) \\
&\leq 2\sum_{i=1}^{n}\mathrm{Var}(m_i) + 2\mathrm{Var}(\frac{n}{s}\sum_{i=1}^{s}(m_i - \Delta_i)) \\
&= 2\sum_{i=1}^{n}\mathrm{Var}(m_i) + \frac{2n^2}{s^2}\mathrm{Var}(\sum_{i=1}^{s}(m_i - \Delta_i)) \\
&= 2\sum_{i=1}^{n}\sigma_i^2 + \frac{2n^2}{s^2}\mathrm{Var}(\sum_{i=1}^{s}(m_i - \Delta_i)) \\
&= 2\sigma^2\sum_{i=1}^{n}\Delta_i^2 + \frac{2n^2}{s^2}\mathrm{Var}(\sum_{i=1}^{s}(m_i - \Delta_i)) \\
&= 2\sigma^2\sum_{i=1}^{n}\Delta_i^2 + \frac{2n^2}{s^2}\sum_{i=1}^{s}(\frac{1}{n}\sum_{i=1}^{n}\sigma^2\Delta_i^2) \\
&= 2\sigma^2\sum_{i=1}^{n}\Delta_i^2 + \frac{2n}{s}\sigma^2\sum_{i=1}^{n}\Delta_i^2 \\
&= 2\sigma^2(\sum_{i=1}^{n}\Delta_i^2 + \frac{n}{s}\sum_{i=1}^{n}\Delta_i^2) \\
&= 2\sigma^2\sum_{i=1}^{n}\Delta_i^2(1 + \frac{n}{s})
\end{aligned}
$$

As with uniform noise, we will plug in the value $\sigma^2 = \frac{\sum_{i=1}^{n}\Delta_i^2}{n}$. This yields the following:

$$
\begin{aligned}
\mathrm{Var}(\tilde{\Delta}) &\leq 2\sigma^2\sum_{i=1}^{n}\Delta_i^2(1 + \frac{n}{s}) \\
&= 2\frac{\sum_{i=1}^{n}\Delta_i^2}{n}\sum_{i=1}^{n}\Delta_i^2(1 + \frac{n}{s}) \\
&= \frac{2}{n}(\sum_{i=1}^{n}\Delta_i^2)^2(1 + \frac{n}{s}) \\
&= 2(\sum_{i=1}^{n}\Delta_i^2)^2(\frac{1}{n} + \frac{1}{s})
\end{aligned}
$$

As before, with a better predictor of $\sigma^2 = \frac{\sum_{i=1}^{n}\Delta_i^2}{n}\epsilon$ for some small value ϵ we can recompute the variance as:

$$
\mathrm{Var}(\tilde{\Delta}) \leq 2\epsilon(\sum_{i=1}^{n}\Delta_i^2)^2(\frac{1}{n} + \frac{1}{s})
$$

7 Conclusion

References

[1] Mohammad Al Hasan and Vachik S. Dave. Triangle counting in large networks: a review. *WIREs Data Mining and Knowledge Discovery*, 2018.

[2] Reka Albert and Albert-Laszlo Barabasi. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74(1):47–97, January 2002.

[3] SM Arifuzzaman, Maleq Khan, and Madhav Marathe. Parallel Algorithms for Counting Triangles and Computing Clustering Coefficients. pages 1448–1449, 2012.

[4] Haim Avron. Counting Triangles in Large Graphs using Randomized Matrix Trace Estimation. In *Proceedings of Kdd-Ldmta'10*, 2010.

[5] Corlin O. Beum and Everett G. Brundage. A Method for Analyzing the Sociomatrix. *Sociometry*, pages 141–145, 1950.

[6] Cristian Boldrin and Fabio Vandin. Fast and Accurate Triangle Counting in Graph Streams Using Predictions, 2024.

[7] Jane K. Cullum and Ralph A. Willoughby. *Lanczos Algorithms for Large Symmetric Eigenvalue Computations: Vol. I: Theory*. Society for Industrial and Applied Mathematics, 2002.

[8] Michael Dinitz, Sungjin Im, Thomas Lavastida, Benjamin Moseley, and Sergei Vassilvitskii. Faster Matchings via Learned Duals, 2021.

[9] Ran Duan and Seth Pettie. Linear-Time Approximation for Maximum Weight Matching. *J. ACM*, pages 1:1–1:23, 2014.

[10] M.F. Hutchinson. A stochastic estimator of the trace of the influence matrix for laplacian smoothing splines. *Communications in Statistics - Simulation and Computation*, pages 433–450, 1990.

[11] Laszlo Lovasz. *Large networks and graph limits*. American Mathematical Society colloquium publications. American Mathematical Society, Providence, Rhode Island, 2012.

[12] Michael Mitzenmacher and Sergei Vassilvitskii. Algorithms with Predictions. In Tim Roughgarden, editor, *Beyond the Worst-Case Analysis of Algorithms*, pages 646–662. Cambridge University Press, 2020.

[13] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, Cambridge, 1995.

[14] P. Prescott, J. M. Hammersley, and D. C. Handscomb. Monte Carlo Methods. *Applied Statistics*, 14(2/3):211, 1965.

[15] C. Seshadhri, Ali Pinar, and Tamara G. Kolda. Triadic Measures on Graphs: The Power of Wedge Sampling. In *Proceedings of the 2013 SIAM International Conference on Data Mining*, pages 10–18, 2013.

[16] V. Strassen. Gaussian Elimination is not Optimal. *Numerische Mathematik*, 13:354–356, 1969.

[17] Jessica Su, Aneesh Sharma, and Sharad Goel. The Effect of Recommendations on Network Structure. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, pages 1157–1167, Republic and Canton of Geneva, CHE, 2016. International World Wide Web Conferences Steering Committee.

[18] Charalampos E. Tsourakakis. Fast Counting of Triangles in Large Real Networks without Counting: Algorithms and Laws. In *2008 Eighth IEEE International Conference on Data Mining*, pages 608–617, 2008.

[19] Charalampos E. Tsourakakis, U. Kang, Gary L. Miller, and Christos Faloutsos. DOULION: counting triangles in massive graphs with a coin. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '09, pages 837–846. Association for Computing Machinery, 2009.

[20] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, June 1998.

[21] Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. New Bounds for Matrix Multiplication: from Alpha to Omega, 2023.

[22] Ping Ye, Brian D. Peyser, Forrest A. Spencer, and Joel S. Bader. Commensurate distances and similar motifs in genetic congruence and protein interaction networks in yeast. *BMC Bioinformatics*, 6:270, November 2005.