Triangle Counting in Large Historical Graphs*

Regular paper†

Konstantinos Christopoulos
Computer Engineering and Informatics
University of Patras
Rion, Achaia, GREECE
kchristopou@upnet.gr

Evaggelos Daskalakis
Computer Engineering and Informatics
University of Patras
Rion, Achaia, GREECE
e_daskalakis@upnet.gr

Agorakis Bompotas
Computer Engineering and Informatics
University of Patras
Rion, Achaia, GREECE
mpompotas@ceid.upatras.gr

Konstantinos Tsichlas
Computer Engineering and Informatics
University of Patras
Rion, Achaia, GREECE
ktsichlas@ceid.upatras.gr

Abstract

Counting local topological structures, such as triangles, is crucial to analyse large-scale networks and to understand the evolution of graphs. Triangles are fundamental for computing transitivity and for applications such as community detection and link prediction. Despite the importance of triangle counting, traditional algorithms struggle with scalability in networks with millions or billions of vertices, prompting the development of approximation methods and distributed solutions.

In this paper, we present a distributed algorithm that can handle historical graphs in order to count triangles communities in a query time interval. We introduce a pioneering approach to triangle counting in historical graphs, a novel concept that incorporates temporal dimensions into traditional graph models. Our method, which has not been previously explored, uniquely counts triangles within user-defined time intervals, offering new insights into the evolution of network interactions. Experiments with real-world historical datasets validate the effectiveness of our approach in capturing temporal patterns, marking a significant advancement in the field, and setting the stage for future research.

CCS Concepts

• **Computer systems organization → Embedded systems**.

Keywords

Triangle Counting, Historical Graphs, Distributed Computing

*Produces the permission block, and copyright information
†The full version of the author's guide is available as `acmart.pdf` document

SAC'25, March 31 –April 4, 2025, Sicily, Italy

ACM ISBN 979-8-4007-0629-5/25/03
https://doi.org/https://doi.org/10.1145/3672608.3707982

ACM Reference Format:
Konstantinos Christopoulos, Evaggelos Daskalakis, Agorakis Bompotas, and Konstantinos Tsichlas. 2025. Triangle Counting in Large Historical Graphs: Regular paper. In *Proceedings of ACM SAC Conference (SAC'25).* ACM, New York, NY, USA, Article 4, 7 pages. https://doi.org/https://doi.org/10.1145/3672608.3707982

1 Introduction

Networks serve as fundamental models for representing entities and their relationships across diverse domains such as biology, communication, social systems, and transportation. Despite the varying structural characteristics of networks in these fields, certain topological patterns, particularly triangles, are frequently observed. The prevalence of triangles in real-world networks has spurred the development of metrics like the clustering coefficient and transitivity ratio, which are used to characterize and analyze network structures. These triangles are not only important for understanding network topology but also have significant implications in social science theories, such as homophily and transitivity, which explore the dynamics of social relationships.

A static graph is defined as a graph where the relationships, or edges, between nodes, or vertices, remain constant over time. In static graphs, triangle counting involves identifying all sets of three nodes that form a complete subgraph, or triangle, where each pair of nodes is interconnected. This process is vital because triangles are fundamental components that reveal insights into the network's local structure and connectivity. For example, in social networks, the presence of triangles can indicate tightly-knit groups or communities, shedding light on social cohesion and group dynamics.

Historical graphs extend the static graph model by integrating a temporal dimension, capturing how edges between nodes change over time. Formally, in a historical graph, an edge is characterized by a time interval during which it exists. This temporal aspect introduces additional complexity to triangle counting. In historical graphs, the objective is to determine the number of triangles that exist within specific time intervals. This involves not only tracking the presence of edges but also understanding how triangles form and dissolve as the network evolves. For instance, in a historical social network, triangles might emerge during periods of high

activity and fade as relationships evolve, reflecting the dynamic nature of social interactions.

The challenge of triangle counting is significantly amplified when transitioning from static to historical graphs due to the added temporal dimension. While static graphs provide a snapshot of network structure, historical graphs require continuous analysis to account for temporal changes in edge existence. This necessitates the development of advanced methods capable of handling the dynamic relationships and providing insights into the temporal evolution of network structures.

In this paper, we present a novel approach tailored for triangle counting in historical graphs. Our method addresses the complexities introduced by the temporal nature of these graphs, offering a framework for accurately counting triangles within user-defined time intervals. This work advances research in historical graph analysis and paves the way for future exploration in the field of temporal network analysis.

2 Related Work

The existing literature covers triangle counting in both static and dynamic graphs. However, there is no research specifically focused on historical graphs, considering edge time intervals and the periods during which an edge remains active. In the following section, we review recent work in this area, spanning approaches from sequential to parallel, single-machine to distributed, exact to approximate, and offline to streaming methodologies.

In this paper [4], the authors presented an algorithm named BA to anonymise triangle counts on large graphs using node-differential privacy. They developed two methods, Tr-Histogram and TC-Histogram, to publish these triangle counts. After conducting a theoretical analysis, they decided to evaluate the two approaches. Furthermore, they showed that the cumulative distribution TC-Histogram delivers superior performance compared to the triangle count distribution Tr-Histogram. They also extended their research to the local clustering coefficient, which they published by categorising the coefficients into k distinct groups.

In [18], the authors introduce DOULION, an algorithm that employs random sampling to create a smaller, weighted graph, providing a triangle count that closely approximates the actual number. They emphasize that DOULION complements existing triangle counting methods by serving as an effective preprocessing step for both streaming and non-streaming algorithms. The algorithm is noted for its high parallelizability, achieving optimal performance in Hadoop environments. Experiments with real-world graphs reveal that DOULION delivers nearly perfect accuracy across various probability settings and can accelerate triangle counting by approximately 130 times compared to a basic exact counting algorithm when used as an initial step.

In this work [8], the authors present a highly efficient triangle-counting approximation algorithm that is designed to be adapted to the semistreaming model as described in [6]. The core idea behind their approach is the innovative integration of the sampling algorithm from [16, 18] with a method that partitions the set of vertices into high- and low-degree subsets, as proposed by [2]. They apply their methods to a range of networks with million edges and achieve exceptional results, exceeding the performance of current triangle-counting techniques. Additionally, they introduce a method for triangle counting that leverages random projection and establish a sufficient condition to ensure that the estimates have low variance.

Authours in [7] introduced TriX, a highly scalable and distributed algorithm for triangle counting designed for external memory use. TriX utilizes an innovative 2-D managed partitioning scheme and a parallel binary search method for intersections optimized for GPUs. The algorithm effectively scales across multiple GPUs, enabling the counting of triangles in extremely large graphs with billions of nodes and edges in a relatively short time. It demonstrates a high performance rate, processing millions of traverse edges per second.

Authors in [17] present two algorithms: Eigen-Triangle for counting the total number of triangles in a graph, and EigenTriangleLocal for counting triangles that include a specific node. They demonstrate that both algorithms achieve high accuracy and are up to approximately 1000 times faster on real-world graphs. The key innovation is the Eigen-Triangle algorithm, which links triangle counts to the eigenvalues of the adjacency matrix, focusing on the significant contribution of the top eigenvalues. This approach leverages advanced, parallelizable eigenvalue algorithms, making it well-suited for extremely large graphs. The method can be efficiently applied to petabyte-scale graphs using existing eigenvalue implementations like Lanczos. Additionally, the authors discover two new power laws, Degree-Triangle and TriangleParticipation, which reveal unexpected properties.

Two advanced MPI-based distributed memory parallel algorithms for efficiently counting triangles in large graphs are introduced in [3]. The first algorithm utilizes overlapping partitioning and load balancing to provide rapid parallel processing, accurately computing the number of triangles in a network with 10 billion edges in just 16 minutes. The second algorithm emphasizes space efficiency by partitioning the network into non-overlapping segments, which significantly reduces memory usage. Both algorithms feature an innovative approach to minimizing communication costs, enhancing both space and runtime efficiency. Additionally, these methods are applicable for listing all triangles in a graph, calculating node clustering coefficients, and can be adapted for parallel approximation using edge sparsification.

The work in [10] presents both deterministic and random sampling techniques for quickly discovering the most significant 3-cliques (triangles) in weighted graphs. For instance, one of their algorithms can identify the top 1,000 triangles with the highest weights in a graph containing billions of edges in just thirty seconds on a standard server, making it vastly quicker than current rapid enumeration methods. These innovations facilitate scalable pattern extraction in weighted graphs. Similar paper for top k weighted triangles we can find in [15].

The authors in [13] introduce a new MapReduce algorithm, known as the *TTP* (Triangle Type Partition) algorithm, which is based on graph partitioning to efficiently count triangles in large-scale graphs. This algorithm significantly enhances efficiency by classifying triangles into three distinct types and processing them accordingly, which greatly reduces data redundancy. Experimental evaluations, including tests on both synthetic and real-world datasets with millions of nodes and billions of edges, show that

the TTP algorithm outperforms previous MapReduce algorithms in most cases. Notably, it demonstrates more than double the speed on a Twitter dataset, with its performance advantage increasing as graphs become larger and denser.

In [5] is presented an adaptive triangle-counting algorithm utilizing SQL queries, tailored for large-scale graphs. This method integrates smoothly into data analysis workflows and operates efficiently within distributed DBMS, big data platforms, and analytical tools. A central feature is the advanced vertex partitioning strategy, which assigns graph data to cluster machines without duplication or data exchange, facilitating local triangle detection within each partition. The algorithm excels with skewed graphs, offering balanced load distribution and significant performance improvements over conventional approaches like Spark GraphX and G-thinker. They also introduce a Python-based alternative using Pandas and MPI, which shows strong performance on small to medium datasets. Experimental results confirm the algorithm's competitiveness with specialized graph analysis engines.

3 Triangle Counting in Historical Graphs

3.1 Preliminaries

Let $G = (V_T, E_T)$ be a historical network. The set of historical nodes V_T consists of a set of nodes along with their time intervals, that is $V_T \subset V \times \mathbb{N}^2$. The set of historical edges E_T is a set of edges along with their time intervals, that is $E_T \subset E \times \mathbb{N}^2$, where E contains all possible $\binom{|V|}{2}$ undirected edges. Note that we consider nodes and edges that have a single valid time interval, but it is easy to generalize to a set of valid time intervals. The preceding definitions mean that each node $v \in V$ (and edge $e \in E$) has a time interval attached $[t_v^{(s)}, t_v^{(f)}]$ (similarly $[t_e^{(s)}, t_e^{(f)}]$) (where (s) and (f) stand for start and finish respectively) that dictates the time instances where node v (edge e) is existent. Thus, if $t \notin [t_v^{(s)}, t_v^{(f)}]$, then v is not existent at time t. $V_{ij} \subseteq V$ contains all nodes that have a time interval that spans the query interval $[t_i, t_j]$. The time interval of each edge is by definition a subset of the time interval of the respective vertices. In case of multiple time intervals, we have to define the borders of each interval accordingly, to avoid overlaps. For example, each interval should be open at the left and closed at the right. The convention we make is that a time point t is represented by $(t, t]$.

Assume that by $\mathcal{N}_{ij}(v)$ we represent the neighborhood of node v in the query time interval $[t_i, t_j]$. Note that $\mathcal{N}_{ij}(v)$ may even be the empty set for specific query time intervals. The routing table $r(v)$ of node v, contains all historical edges to other nodes in G. This means that $r(v)$ contains all edges and nodes along with their time interval. By $r_{ij}(v)$, we represent the part of this table that contains historical edges with time intervals that intersect the query time interval $[t_i, t_j]$, that is all edges that point to nodes that belong to $\mathcal{N}_{ij}(v)$.

Our algorithm is developed within the framework of the Local distributed model [11]. In this model, graph nodes are uniquely identified by distinct IDs and communicate with their neighboring nodes via messages of unlimited size. Both communication and computation occur synchronously, and the distributed system is assumed to be fault-free, with nodes considered to possess infinite computational power. In this work, we focus exclusively on historical edges, not historical nodes. This means that while nodes remain valid at all times, edges are only valid during specific time intervals.

3.2 Problem Formulation

In a vertex-centric system, as described by Kosmatopoulos et al.[9], the historical graph is organized to efficiently store the entire evolution of a node and its adjacent edges over time. This system is designed for space efficiency and optimized query performance, particularly for temporal queries that involve both structural and temporal dimensions of the graph. In this model, the entire history of a node, including its adjacent edges and their states across different time periods, is stored within the node itself. This compact representation makes it possible to efficiently perform both updates and queries on the graph, particularly in scenarios involving dynamic or evolving graphs.

Triangle Counting Over a Time Interval

The main task at hand is to efficiently compute triangle counts in a temporal graph for a specific time interval, $[t_i, t_j]$. In a graph, a triangle is formed by three nodes that are all mutually connected by edges. However, in the temporal context, edges and nodes may have varying lifespans, meaning they are only valid or active during specific time periods. This adds a layer of complexity to triangle counting as we need to account for both structural relationships and temporal validity.

When the query asks for triangle counting at a specific time instance t_i, this problem reduces to the traditional triangle counting algorithm applied to a single snapshot of the graph at time t_i. In this case, the algorithm identifies all triangles that exist at that moment, excluding nodes or edges that are not valid at t_i. This is a static graph triangle counting problem, and existing algorithms such as those discussed in the work of Pagh et al. [12] can be applied efficiently. When the query spans a time interval $[t_i, t_j]$, we consider two different approaches to measure the number of triangles:

1. Aggregate Number of Triangles

This approach introduces a more nuanced method of triangle counting by considering how long a triangle exists within the time interval $[t_i, t_j]$. Instead of simply counting whether a triangle exists, it weights each triangle based on the overlap between the lifespans of its edges and the query interval.

For example, suppose an edge e_{ab} is active between nodes a and b from t_1 to t_5, and the query interval is $[2, 6]$. In this case, e_{ab} is valid for 4 out of the 5 time instances in the interval $[2, 6]$, so it would be assigned a weight of $\frac{4}{5}$. This weight reflects the proportion of the time interval during which the edge contributes to forming a triangle. For each edge that could potentially form a triangle, calculate how long it is active within the time interval $[t_i, t_j]$. If an edge is entirely outside the interval, it contributes nothing. Otherwise, compute its weight based on the proportion of the interval during which it is valid. For every potential triangle, check whether all three constituent edges are valid during some overlapping portion of the interval. Then, sum up the weights of the

edges that participate in the triangle. The total aggregate triangle count is the sum of the weighted contributions of all valid triangles.

This method provides a more detailed understanding of the triangle structure in a dynamic graph by accounting for the temporal persistence of triangles, rather than just their presence or absence. Triangles that persist longer within the time interval contribute more heavily to the final count, reflecting their significance during the specified time window.

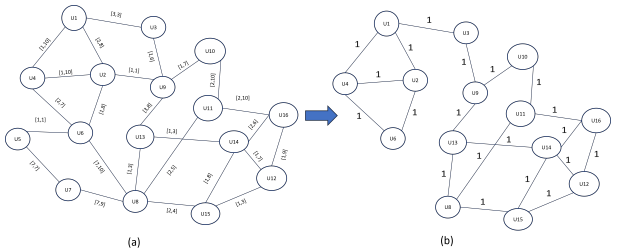

Figure 1: Illustration of graph transformation over a specific time interval to estimate the total number of triangles. a) The initial network G, where each edge is associated with a time interval. b) The transformed graph G for the query time interval $[2, 6]$, with a weight of 1 assigned to each active edge.

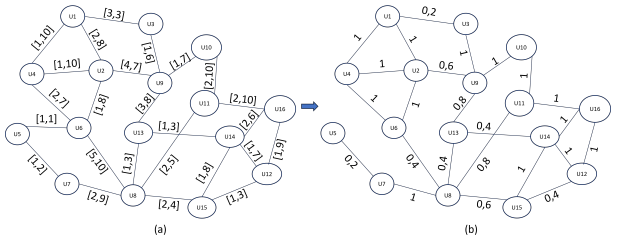

Figure 2: Illustration of graph transformation over a specific time interval to estimate the aggregate number of triangles. a) The initial network G, where each edge is associated with a time interval. b) The transformed graph G for the query time interval $[2, 6]$, showing the corresponding weights on each active edge.

Definition 3.1. The **observed interval** of an edge is the intersection between the query time interval and the valid time interval of this edge.

When an edge is not valid at any time instance of the query time interval, then it is non-existent during this time interval. Consequently, the unweighted historical graph is transformed into a weighted static graph for a particular time interval, where the weight of each node/edge represents the ratio of the size of its observed interval to the size of the total query interval.

Definition 3.2. The **observation ratio** of an object (edge/triangle) is the ratio between the size of the observed interval of the object and the size of the query time interval defined by the user.

In Figure 2, we illustrate a temporal network along with the corresponding observation ratios. The maximum weight of an edge is 1, which occurs when the edge's time interval fully overlaps with the query interval. Conversely, the minimum observation ratio is zero when the time intervals of the edge and the query are disjoint, resulting in the absence of an edge.

2. Total Number of Triangles

In this approach, the goal is to count all triangles that exist at any point during the time interval $[t_i, t_j]$. Here, a triangle is considered valid if all three edges that form the triangle are active simultaneously at some time within the interval. This approach does not take into account how long the triangle persists in the graph during the interval. It simply checks whether a triangle is present at any moment within the interval.

To compute this, we can iterate over each node in the graph, retrieve its historical edges, and check for periods during which all three edges of a potential triangle are active together. If there is any overlap in the lifespans of the three edges, the triangle is counted.

For example, if an edge e_{ab} between nodes a and b is active from t_2 to t_4 and the other two edges e_{bc} and e_{ac} also overlap during this interval, then the triangle formed by nodes a, b, and c will be counted. Figure 1 presents two possible scenarios: either the edge is active at least once within the query interval, in which case its weight is always 1, or it is inactive throughout the interval, meaning no edge is present.

Although the algorithms developed in this paper focus on the total number of triangles, the aggregate method offers useful definitions that enhance the understanding of the approaches described below.

3.3 Methods of Triangle Counting in Historical Graphs

In this section, we present three algorithms for counting triangles in historical graphs, where the temporal dimension plays a crucial role in determining the validity of edges and their contribution to triangle formation.

(1) **Time-Interval-Based Triangle Counting in historical Graphs**
This method operates without relying on queries and follows a streamlined process as outlined below:
   (a) The algorithm begins by creating neighbor sets for each vertex in the graph. Each set includes information about neighboring vertices and the time intervals during which the edges between them are active.
   (b) Next, it joins these neighbor sets with the original graph, resulting in a new graph where each vertex's attributes contain its neighbor set along with their respective time intervals.
   (c) A function is defined to calculate the intersection of time intervals between edges.
   (d) For each edge in the graph, the algorithm checks for the presence of triangles by examining the time intervals of the edges. It calculates the number of triangles where all three vertices are interconnected by edges with overlapping time intervals.

(e) Finally, the triangle counts from each vertex are aggregated to obtain the total number of triangles present in the graph.

This approach provides a method for detecting triangles by evaluating every possible combination of edges.

The following two methods rely on queries and follow a streamlined process, detailed as follows

(2) **Filtering-based Triangle Counting Algorithm in Historical Graphs**

This algorithm introduces a crucial difference from the Time-Interval-Based Triangle Counting method by beginning with a preliminary filtering of edges based on a specified query time interval. Only edges that are active within this time window are retained, creating a subgraph that meets the temporal criteria. After this initial filtering, the remaining steps - such as the creation of the neighbour set, the detection of triangles, and the aggregation - are identical to those of the time interval-based triangle count method.

By focusing on time-based filtering at the outset,this approach simplifies the triangle detection process by significantly reducing the graph size that needs to be processed.

(3) **Thresholding and Filtering-Based Triangle counting in Historical Graphs**

The Thresholding and Filtering-Based Temporal Triangle Algorithm introduces an additional layer of optimization by incorporating both filtering based on time intervals and a thresholding mechanism that evaluates edge relevance.

This algorithm differs significantly from the Time-Filtered Triangle Counting Algorithm by first applying an overlap threshold to the edges active during the specified query time interval. In other words, if the observation ratio is equal to or greater than the overlap threshold, the edge is considered valid and can be included in the resulting subgraph produced by the filtering process. This dual approach enhances efficiency by ensuring that only edges with significant temporal coherence are included in the analysis, thus further reducing the size of the graph that needs to be processed.

4 Experiment Design

To assess the performance and scalability of our triangle counting algorithms, we conduct experiments using three real-world datasets, each representing a distinct type of network. These datasets encompass a variety of network structures and sizes, allowing us to thoroughly evaluate the algorithms under different conditions. The datasets include social networks, collaboration networks, and product networks, all characterized by their undirected edges and inherent community structures. The primary characteristics of the datasets used in this study are summarized below:

- **com-Youtube:** Representing the YouTube online social network, this dataset has 1,134,890 nodes and 2,987,624 edges, with 3,056,386 triangles. This smaller dataset, relative to others in the study, allows us to explore the algorithms' efficiency in sparse networks.
- **com-DBLP:** The DBLP collaboration network dataset consists of 317,080 nodes and 1,049,866 edges, with 2,224,385 triangles. This graph represents a collaboration network where researchers are linked through co-authorship, making it ideal for testing triangle counting in academic and professional networks with a moderate community structure.
- **com-Amazon:** This dataset captures the Amazon product network, containing 334,863 nodes, 925,872 edges, and 667,129 triangles. Nodes represent products, and edges indicate co-purchase relationships, offering a scenario with a retail-based network structure.

These datasets provide a comprehensive foundation for testing the performance of triangle counting algorithms across a range of network sizes, edge densities, and community distributions. Each dataset's specific structure allows us to explore different challenges, from large-scale networks with millions of edges to smaller, more sparse networks. The diversity of these datasets ensures that our experiments yield insights into the algorithms' behavior in real-world scenarios.

Since the datasets used in our experiments are not inherently temporal or historical, it is necessary to assign time intervals to the edges in order to simulate dynamic graph behavior. To achieve this, we associate each edge with a randomly generated time interval using a uniform distribution, where the start and end times are drawn from a predefined time range. This approach allows us to introduce a temporal dimension to the graph, enabling the evaluation of our triangle counting algorithms in a temporal context. By distributing time intervals uniformly, we ensure that all edges have equal probability of being active at any point in time, avoiding biases toward specific time periods and creating a balanced temporal distribution across the graph. This artificial temporal assignment is crucial for testing how well the algorithms perform when handling historical graphs where edges have varying periods of validity.

The range of each dataset spans from 1 to 100,000. In the first method, Time-Interval-Based, we do not utilize query time intervals, as mentioned earlier. In contrast, the other two methods employ two distinct query time intervals: the first has range 5,000, and the second 10,000. For the Thresholding and Filtering-Based method, we established an overlap threshold of 40%. All of these parameters were chosen randomly to obtain preliminary results for the aforementioned methods.Lastly, for the sake of brevity, we refer to the Time-Interval-Based method as Tr-Interval, the Filtering-Based method as Tr-Filtering, and the Thresholding and Filtering-Based method as Tr-Thresholding. The numbers 1 and 2 in the tables below correspond to the ranges of 5,000 and 10,000, respectively.

The experiments were conducted in an Apache Spark 3.5.3 environment running on a small Kubernetes cluster composed of various commodity hardware PCs. This heterogeneous setup allowed for flexibility, as nodes of differing specifications could work together, contributing to a robust yet economical test-bed for distributed processing. The final experimental configuration utilized 12 executor pods, each provisioned with 2 CPU cores and 4 GB of RAM. This configuration enabled efficient parallel processing and resource allocation across the cluster, allowing Spark jobs to leverage Kubernetes' orchestration capabilities for load balancing and fault tolerance. This setup facilitated the effective testing of Spark's distributed computing capabilities on a modest-scale cluster.

4.1 Experimental Results

The tables below present our preliminary results on four real-world datasets. Each cell in the tables shows both the number of detected triangles (tr) and the execution time in milliseconds. At first glance, we can observe that as the dataset size increases, the thresholding method yields more favorable results. This is logical, as the overlap threshold leads to a smaller subgraph. When the initial graph is large, triangles can be counted more efficiently within the reduced subgraph.

Table 1: Experiments in Amazon and DBLP

	Amazon	DBLP
Tr-Intervals	286994 tr | 8288ms	957120 tr | 8717ms
Tr-Filtering 1	105703 tr | 12312ms	352710 tr | 9118ms
Tr-Filtering 2	186242 tr | 12791ms	622119 tr | 13041ms
Threshold 1	61916 tr | 6863ms	205998 tr | 10778ms
Threshold 2	90317 tr | 9972ms	302611 tr | 7821ms

Table 2: Experiments in Youtube

	Youtube
Tr-Intervals	1325051 tr | 20705ms
Tr-Filtering 1	494973 tr | 28188ms
Tr-Filtering 2	864790 tr | 30302ms
Threshold 1	291911 tr | 18332ms
Threshold 2	420434 tr | 17449ms

The Tr-Interval method consistently demonstrates superior performance in terms of both triangle detection and execution time, particularly in the Amazon and DBLP datasets, where it detects 286,994 triangles in 8,288 ms and 957,120 triangles in 8,717 ms, respectively. This makes it the most efficient method for these datasets, particularly when processing large-scale graphs. In Youtube, it also identifies a substantial number of triangles (1,325,051) in 20,705 ms, reaffirming its effectiveness in large graphs.

Both Tr-Filtering 1 and 2 show significantly lower triangle counts compared to the Tr-Interval method. For example, in Amazon, Tr-Filtering 1 detects 105,703 triangles in 12,312 ms, while Tr-Filtering 2 detects 186,242 triangles in 12,791 ms. Similarly, in DBLP, Tr filtering 1 and 2 detect 352,710 and 622,119 triangles respectively, with execution times higher than Tr-interval. This suggests that the additional filtering step required by these methods increases computational overhead, particularly as the graph size grows. The Youtube dataset shows the same trend, where both Tr-Filtering methods have the slowest times, with Tr-Filtering 2 taking the longest (30,302 ms), while Tr-Filtering 1 detects fewer triangles (494,973) in 28,188 ms.

The threshold-based methods show mixed results across datasets. In Amazon, both Threshold 1 and Threshold 2 detect the fewest triangles (61,916 and 90,317, respectively), though Threshold 1 is the fastest, completing in 6,863 ms. In DBLP, these methods detect more triangles (205,998 and 302,611) than in Amazon, but with longer execution times. In the Youtube dataset, Threshold 1 detects 291,911 triangles in 18,332 ms, while Threshold 2 detects 420,434 triangles in 17,449 ms. These results suggest that the thresholding methods, still perform reasonably well in larger graphs like Youtube and DBLP.

The results indicate that the Tr-Interval method, in average, excels in both triangle detection and speed, especially in Amazon and DBLP graphs. The reason could be that it skips the complex filtering and thresholding steps, focusing directly on time intervals, which simplifies the computation. On the other hand, the Tr-Filtering methods, while more precise in filtering based on time intervals, struggle with time complexity. Threshold methods appear to offer a trade-off between computational efficiency and triangle detection accuracy. In larger datasets, they perform relatively well and in some cases surpass the Tr-Interval method in speed.

5 Conclusion

In this work, we focus on counting triangles in historical graphs that include a temporal dimension. Specifically, the edges are associated with valid time intervals, indicating their existence during specific time instances. This represents a novel contribution to the field of Temporal/Historical Graphs by introducing the innovative concept of edge and node activity, which facilitates triangle counting through the adaptation and extension of existing methodologies.

We aim to build upon the preliminary findings presented in this paper through a more comprehensive theoretical analysis, including an examination of time and computational complexity. In our theoretical exploration, we will also investigate the implications of nodes with multiple time intervals.

A significant extension of this research involves utilizing triangle counting in historical graphs to detect communities. Drawing from previous works such as [1] and [14], where triangle counting is applied to undirected plain graphs to discover communities, we plan to extend and adapt these methods to identify communities within historical graphs.

Acknowledgment

"This research was supported by the Hellenic Foundation for Research and Innovation (H.F.R.I.) under the "2nd Call for H.F.R.I. Research Projects to support Faculty Members & Researchers" (Project Number: 3480). "

References

[1] Tariq Abughofa, Ahmed A Harby, Haruna Isah, and Farhana Zulkernine. 2021. Incremental community detection in distributed dynamic graph. In *2021 IEEE Seventh International Conference on Big Data Computing Service and Applications (BigDataService)*. IEEE, 50–59.

[2] Noga Alon, Raphael Yuster, and Uri Zwick. 1997. Finding and counting given length cycles. *Algorithmica* 17, 3 (1997), 209–223.

[3] Shaikh Arifuzzaman, Maleq Khan, and Madhav Marathe. 2019. Fast parallel algorithms for counting and listing triangles in big graphs. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 14, 1 (2019), 1–34.

[4] Xiaofeng Ding, Shujun Sheng, Huajian Zhou, Xiaodong Zhang, Zhifeng Bao, Pan Zhou, and Hai Jin. 2021. Differentially private triangle counting in large graphs. *IEEE Transactions on Knowledge and Data Engineering* 34, 11 (2021), 5278–5292.

[5] Abir Farouzi, Xiantian Zhou, Ladjel Bellatreche, Mimoun Malki, and Carlos Ordonez. 2024. Balanced parallel triangle enumeration with an adaptive algorithm. *Distributed and Parallel Databases* 42, 1 (2024), 103–141.

[6] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. 2005. On graph problems in a semi-streaming model. *Theoretical Computer Science* 348, 2-3 (2005), 207–216.

[7] Yang Hu, Pradeep Kumar, Guy Swope, and H Howie Huang. 2017. Trix: Triangle counting at extreme scale. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[8] Mihail N Kolountzakis, Gary L Miller, Richard Peng, and Charalampos E Tsourakakis. 2012. Efficient triangle counting in large graphs via degree-based vertex partitioning. *Internet Mathematics* 8, 1-2 (2012), 161–185.

[9] Andreas Kosmatopoulos, Kostas Tsichlas, Anastasios Gounaris, Spyros Sioutas, and Evaggelia Pitoura. 2017. Hinode: an asymptotically space-optimal storage model for historical queries on graphs. *Distributed and Parallel Databases* 35 (2017), 249–285.

[10] Raunak Kumar, Paul Liu, Moses Charikar, and Austin R Benson. 2020. Retrieving top weighted triangles in graphs. In *Proceedings of the 13th International Conference on Web Search and Data Mining*. 295–303.

[11] Nathan Linial. 1992. Locality in distributed graph algorithms. *SIAM Journal on computing* 21, 1 (1992), 193–201.

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

[13] Ha-Myung Park and Chin-Wan Chung. 2013. An efficient mapreduce algorithm for counting triangles in a very large graph. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*. 539–548.

[14] Arnau Prat-Pérez, David Dominguez-Sal, and Josep-Lluis Larriba-Pey. 2014. High quality, scalable and parallel community detection for large real graphs. In *Proceedings of the 23rd international conference on World wide web*. 225–236.

[15] Ryosuke Taniguchi, Daichi Amagata, and Takahiro Hara. 2022. Efficient retrieval of top-k weighted triangles on static and dynamic spatial data. *IEEE Access* 10 (2022), 55298–55307.

[16] Charalampos Tsourakakis, Mihail Kolountzakis, and Gary Miller. 2011. Triangle sparsifiers. *Journal of Graph Algorithms and Applications* 15, 6 (2011), 703–726.

[17] Charalampos E Tsourakakis. 2008. Fast counting of triangles in large real networks without counting: Algorithms and laws. In *2008 Eighth IEEE International Conference on Data Mining*. IEEE, 608–617.

[18] Charalampos E Tsourakakis, U Kang, Gary L Miller, and Christos Faloutsos. 2009. Doulion: counting triangles in massive graphs with a coin. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. 837–846.