Fast and Scalable Triangle Counting in Graph Streams: The Hybrid Approach

by

Paramvir Singh
B.Tech. (Computer Science), Punjab Technical University, 2016

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Fast and Scalable Triangle Counting in Graph Streams: The Hybrid Approach

by

Paramvir Singh
B.Tech. (Computer Science), Punjab Technical University, 2016

Supervisory Committee

Dr. Alex Thomo, Co-Supervisor
(Department of Computer Science)

Dr. Venkatesh Srinivasan, Co-Supervisor
(Department of Computer Science)

Supervisory Committee

Dr. Alex Thomo, Co-Supervisor
(Department of Computer Science)

Dr. Venkatesh Srinivasan, Co-Supervisor
(Department of Computer Science)

ABSTRACT

Triangle counting is a major graph problem with several applications in social network analysis, anomaly detection, etc. A considerable amount of work has contributed to approximately computing the global triangle counts using several computational models. One of the most popular streaming models considered is Edge Streaming in which the edges arrive in the form of a graph stream. We categorize the existing literature into two categories: Fixed Memory (FM) approach, and Fixed Probability (FP) approach. As the size of the graphs grows, several challenges arise such as memory space limitations, and prohibitively long running time. Therefore, both FM and FP categories exhibit some limitations. FP algorithms fail to scale for massive graphs. We identified a limitation of FM category *i.e.* FM algorithms have higher computational time than their FP variants.

In this work, we present a new category called the Hybrid approach that overcomes the limitations of both FM and FP approaches. We present two new algorithms that belong to the hybrid category: Neighbourhood Hybrid Multisampling (NHMS) and Triest/ThinkD Hybrid Sampling (THS) for estimating the number of global triangles in graphs. These algorithms are highly scalable and have better running time than FM and FP variants. We experimentally show that both NHMS and THS outperform state-of-the-art algorithms in space-efficient environments.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank:

my family, for supporting me to pursue this extraordinary journey.

Dr. Thomo, for providing me this opportunity, and continuously motivating me to push my limits.

Dr. Srinivasan, for mentoring and sharing his knowledge.

Jasbir Singh for being a good friend, providing support throughout, and sharing his knowledge.

DEDICATION

This work is dedicated to my parents and my loving sister!
Thanks for believing in me.

Chapter 1

Introduction

A network is a group of two or more objects that has an association. Mathematically, a network can be viewed as a graph where the objects are the nodes and the association between them is represented by the edges. Graphs are everywhere, for illustration, the internet; a set of computers connected to each other form a computer network; a community of people is a social network; the connectivity of cities via roads make a road network, etc. The amount of data has grown exponentially in recent years with the rapid growth of the internet. This growth has resulted in graphs being a powerful tool to analyze and visualize the relationship between the data. The mining of these graphs provides us an enormous amount of crucial information.

As the web based graphs have grown to massive sizes, it is becoming difficult to calculate the characteristic of graphs and analyze them to extract useful information. The querying, storage, and mining of such data sets are highly computationally challenging tasks [14]. The problem of counting subgraphs has attracted a lot of attention in graph mining. The triangle is the simplest such subgraph, with three nodes that are connected pair-wise by edges.

The exact counting of triangles is quite expensive on massive graph data sets. There has been a lot of emphasis on counting triangles with parallel computation, which still needs substantial resources to provide the exact count. Besides this, approximation algorithms for triangle counting have become quite popular and there are tens of literature available that estimate the number of triangles of a graph with guarantees on the quality of estimation. In this thesis, we study the triangle approximation problem and present a new hybrid approach and two new algorithms for getting a triangle count estimation.

1.1 Motivation

Counting triangles forms a basis for many network analysis, such as social network analysis, anomaly detection, recommendation system, and even in understanding the evolution of web graphs [8, 11, 13, 16, 37, 41]. A lot of interest has been shown by sociologists to identify the count of triangles in graphs [8, 41].

Triangle count is critical for frequently used triangle connectivity [3], transitivity coefficient [25], and clustering coefficient [40], in the analysis. This task is especially challenging when the network is massive with millions of nodes and edges. Several methods had been proposed that are classified into two categories: exact counting and approximate counting. The exact counting, for triangles, is done through enumeration/listing which touches the triangles one by one [29]. The approximate counting is done by sampling the graph and using probabilistic formulae to estimate the total number of triangles in the whole graph based on the number of triangles found in the sample.

Edge streaming is a model where a series of edges arrives in order, one at a time. The edge streaming model is widely accepted, and a considerable amount of effort has been made in designing algorithms to estimate the number of triangles in the graph using this model. We will study about streaming in detail in Section 3.3. There are two scenarios where streaming is desired as explained below.

The first scenario is related to real-time applications where edges are streamed as they are created. The graph size is indefinite (i.e., the size of a graph is unknown beforehand, and it might grow forever) and therefore typically the graph is not stored. Furthermore, graph streams can be dynamic. Dynamic graphs are graphs where the edges can either be added or deleted. Social networks are popular for their dynamic behaviour. For this scenario, we would like to have an estimate at any time in the middle of the streaming, or on any edge arrival.

The second scenario is when the graph might be static and stored in a storage device. Static graphs are graphs that do not change with time. The streaming model is used in conjunction with the sampling method in this scenario, as for sampling, the goal is to get an estimate for the triangle count quickly using relatively small memory space. There is a trade-off between running time and accuracy for the Streaming model and Enumeration. For streaming on the static graphs to make sense, it must significantly surpass enumeration in terms of time and memory performance.

Many previous works on triangle estimation in streams has been published. We

classify them into two different categories. The first category includes algorithms that have a *Fixed Memory (FM)* Budget. These algorithms sample the edges within the fixed memory budget and require the user to input the available amount of memory. Once the available memory is full with edges, the next sampled edge randomly replaces an edge in the memory. This is how the sub-graph is maintained within the fixed memory budget.

The second category includes the algorithms that require the user to specify an edge sampling probability p that is fixed for the entire stream, we call them *Fixed Probability (FP)* approach. These algorithms maintain a memory reservoir that doesn't have any size limit. Therefore, if the graph stream arriving is massive, the algorithm often runs out of memory. But, FP algorithms have some benefits over FM algorithms. We perform some experiments and compare the time complexities of both categories, and show that the FP algorithms are faster than FM algorithms, later in Section 4.

However, both categories have their own limitations. The FP approach exhibits several notable impediments as follows: If the probability p is fixed for sampling an edge, the sample size grows with the size of the stream. Also, identifying an appropriate p is a hard-to-choose input parameter. If p is large, the algorithm sometimes even does not execute completely or may run out of memory, on the contrary, choosing smaller p might result in sub-optimal estimations. On the other hand, FM algorithms have a higher running time while edges are being added to a reservoir that has space.

The idea of using both fixed memory and fixed probability together remains unexplored. We explore this idea and the key intention behind this idea is to utilize the benefit of both the previous detailed approaches. We form a new category called the Hybrid category. As the name suggests, the Hybrid category utilizes the power of FM algorithms, that make it more scalable, and the power of FP algorithms, that make it the fastest among all available algorithms. The next chapter details our contributions made as part of this thesis.

1.2 Contribution

The contributions of this thesis are:

- We prove that FP algorithms are faster than FM algorithms. We design a new algorithm called *Neighbourhood Fixed Memory Multisampling (NFMS)* to

validate this claim. NFMS is an FM algorithm and we compare it with NMS proposed by Kavassery et al. [19] which is an FP algorithm. We provide the running time analysis to prove this claim in Section 4.

- We propose an algorithm called *Neighborhood Hybrid Multisampling (NHMS)* that proves to be highly scalable. NHMS leverages the power of both FP and FM approaches and gives better runtime and great accuracy in a space-efficient environment compared to the state-of-the-art algorithms. Not only is our algorithm highly scalable compared to FP algorithms but also is significantly faster than all other approximation algorithms. The experimental results validate our claims when we compare our approach with the several approximation algorithms on different real-life datasets.

- We propose an algorithm called *Trièst - ThinkD Hybrid Sampling (THS)*. THS is a part of the Hybrid category and is extremely efficient. We prove that THS is the fastest algorithm available for Triangle Counting in Graph Streams. Our experimental results show that THS is at least 5 times faster than its FM variant. Also, our detailed experiment results show that THS is the fastest of all other approximation algorithms including NHMS.

- We conduct extensive experimentation and prove that our algorithm could execute graphs with billions of edges on a commodity machine with 16 GB RAM and is thus scalable to large graph datasets, whereas many other approximation algorithms fail to execute on the same machine.

1.3 Organization

This thesis is organized as follows:

Chapter 1 includes a brief introduction about the relevance of graph mining and triangle counting, the motivation behind taking this project and the contributions.

Chapter 2 provides the literature review for the triangle approximation problem. It provides a brief description of the major contributions made in the field of triangle computation. It also talks about the limitations and advantages of each approach.

Chapter 3 talks about the background knowledge for graph theory, streaming model, and some sampling methods. It contains basic terminologies and concepts that are important to understand our contributions.

Chapter 4 describes the differences between fixed probability and fixed memory algorithms, and their characteristics.

Chapter 5 presents the new Hybrid approach for Triangle approximations. It also contains the details of the two new algorithms proposed for triangle approximation.

Chapter 6 contains the experimental results which show the triangle approximation done on some large real-life datasets. It also provides the comparison of our algorithms with their existing counterparts.

Chapter 7 concludes the problem statement and the contributions made. It also contains a discussion about possible future work in this area.

Chapter 2

Related Work

Extensive literature is available for triangle counting in graphs using several computational models [1, 4, 21, 23, 24, 26, 30, 31, 36, 38], . Our study focuses only on the approximation algorithm using streaming model, hence in this section, we will discuss about the algorithms in that model.

Using streaming as a method to sample triangles in a graph can be tracked back to Bar-Yossef et al. [2] in 2002. They presented an algorithm that reduces to the problem of computing the certain frequency moments of graph stream derived from the edge stream. Jowhari and Ghodsi [18], and Buriol et al. [9] made some improvements to the algorithm. In particular, they proposed one-pass streaming algorithms that rely on the idea of reservoir sampling [39]. Since then, there have been many ideas introduced to improve the time and/or space complexity of streaming algorithms.

Since in this study we propose that the literature presented until now falls in either the *Fixed Memory (FM)* or *Fixed Probability (FP)* category, we will classify the previous work into these categories in this section. Section 4 will provide detailed insights on FP and FM algorithms.

Pavan et al. [27] and Jha et al. [17] introduce algorithms for estimating the global number of triangles from edge-insertion-only streams. Pavan et al. [27] presented a Neighbourhood Sampling algorithm that needs to execute multiple copies called estimators. Each estimator uses reservoir sampling to sample a random edge e_1, a random neighbouring edge of e_1, let's say e_2, and then waits for an edge e_3 and checks if e_1, e_2, e_3 form a triangle. Its accuracy depends on the number of estimators N. The larger N is, the closer is the expected value of T to the exact number of triangles in the graph. However, the running time is also proportional to the number of estimators N. As each estimator runs in $O(m)$ time, the total running time is $O(Nm)$, which

can be very large for large graphs. Neighborhood Sampling is an FM algorithm.

Jha et al. [17] applied Birthday Paradox to get the estimate of triangles in graph stream. They presented an algorithm that requires $O(\sqrt{n})$ space to store the edges for accurate results when the transitivity is constant. Similar to Neighborhood Sampling, Birthday Paradox is also FM algorithm.

Lim and Kang [24] proposed an FP algorithm called MASCOT that samples an edge with a fixed probability p and provides the local and global count of triangles. Every time the edge is sampled, it is checked if it forms the triangle within the sampled sub-graph.

Stefani et al. [35] presented a suite of algorithms called $Triest$. $Triest$ uses reservoir sampling to sample multiple edges in a fixed memory reservoir. Shin et al. [32] proposed two different algorithms named $ThinkD$ (Think before you discard). The first algorithm is $ThinkD_{Fast}$, which falls in an FP category. The other algorithm is $ThinkD_{Acc}$ which handles a dynamic stream with edges insertion and deletion. As we are considering insertion-only streams, $ThinkD_{Acc}$ is no different from $Triest$. Hence, both $Triest$ and $ThinkD_{Acc}$ belong to the FM category.

Han and Sethu [15] proposed ESD (Edge Sampling and Discard). ESD is an FP algorithm that maintains an estimate of the global count of triangles. ESD also requires the whole graph to be stored in the memory which limits its scalability. Every time the edge is sampled by ESD with probability p, it queries the whole graph to check if the triangle exists. Hence, it can be argued if it is effectively uses sampling, as it needs the whole graph to be present in memory.

Kavassery et al. [19] proposed two algorithms, Edge-Vertex Multisampling (EVMS), and Neighbourhood Multisampling (NMS). EVMS is an extension to an algorithm by Buriol et al. [9]. Kavassery et al. [19] modified the approach in [9] by sampling multiple vertices and multiple edges with fixed probability p and q, and storing the cross edges that connect the sampled vertices to get global triangles estimates. EVMS needs to be provided with the vertex set beforehand. Hence, EVMS requires additional memory to store the vertex set of the graph. Also, it might not be possible for EVMS to process the dynamic streaming of edges without tweaking it, as the sampled edge needs a vertex set to check if the triangle exists. Therefore, dynamic streaming in EVMS would lead to increased running time and more memory. EVMS is categorized as FP algorithm.

Kavassery et al. [19] presented another algorithm named NMS by modifying the Neighbourhood Sampling [27] using the multisampling approach, similar to EVMS.

NMS focuses on sampling multiple edges in two different reservoirs with probability p and q. This proves to obtain higher accuracy in the triangle count in comparison to the previous work. But it has limitations on scalability that we will discuss further in Section 4. NMS is also an FP algorithm, it instead uses sampling of edges twice.

The FM and FP categories have their impediments and hence, all these algorithms have some limitations. FP algorithms are not scalable on large graphs, whereas FM algorithms are highly scalable. On the other hand, FM algorithms are significantly slower than FP algorithms. None of these works have explored the hybrid approach. We present a new hybrid approach that is both scalable and faster than FM and FP algorithms. In addition, we present two new hybrid category algorithms ($NHMS$ and THS, more details in Chapter 5), that perform better than all the works mentioned.

In Section 6, we provide detailed experimental comparison with the algorithms that are the best performing in the literature available. $Triest$ is one of the best and most popular algorithm in graph streams that belongs to the FM category. On the other hand, NMS proves to be the best performing in the FP category. Also, our proposed algorithms are hybrid variants of $Triest$ and NMS, therefore, it makes more sense to compare our hybrid algorithms with $Triest$ and NMS. The key metrics compared are Running time, Accuracy and Scalability.

Chapter 3

Preliminaries

The purpose of this study has been discussed in detail in the previous two chapters. This chapter provides the background information and the terminologies that are important in understanding the future chapters. Section 3.1 covers the basics of graph theory and notations that are widely used throughout the course of this study. Sections 3.2, 3.3 and 3.4 explain the general concepts required to understand triangle counting on graph streams. Sections 3.5 and 3.6 detail the two triangle counting algorithms named NMS and $Triest$, which are important to understand before our own approach.

3.1 Graphs and Notations

In this thesis, we consider simple undirected graphs. A graph $G(V, E)$ is a composite object of a set of vertices V and a set of edges E connecting vertices in V. An edge stream Σ is an ordered sequence of edges. For a static graph, the set E is known, and the size of Σ is the same as the size of E. In a typical scenario, an algorithm receives edges from the stream and process them one by one. We might label the stream with the graph, such as $\Sigma(G)$.

The notations that we use here are summarized in Table 3.1. Some notations are specific to a particular algorithm, and we will define them in the respective sections.

Table 3.1: Notation used in this paper.

Symbol	Definition
$G(V, E)$	An undirected graph with set of nodes V and set of edges E.
$n = \|V\|$	The number of nodes in G.
$m = \|E\|$	The number of edges in G.
Σ	A stream of edges (may be dynamic).
$\Sigma(G)$	A stream of edges of graph G.
$e = (u, v)$	An undirected edge between u and v, equals to (v, u).
$N(u)$	The set of neighbours of node u.
$d(u)$	The degree of node u, $d(u) = \|N(u)\|$.
$\mathcal{N}(e)$	The set edges adjacent to edge e and comes after e in the stream.
$c(e)$	The size of $\mathcal{N}(e)$, i.e. $c(e) = \|\mathcal{N}(e)\|$
$(u, v, w)_{\triangle}$	A triangle with nodes u, v, and w
$(u, v, w)_{\angle}$	A wedge with nodes u, v, and w, centered at v
Δ	The exact total number of triangles.
T	The estimate total number of triangles.
Δ_u	The number of local triangles with one vertex is u.
τ_u	The estimate of Δ_u.

3.2 Probability

Many algorithms in the available literature on counting triangles in graph streams use the concept of probability to make a decision on choosing the value for a random variable following Bernoulli distribution. The FP algorithms expect the sampling probability p as an input parameter.

This probability p helps to make a decision that either the edge should be stored or discarded. This can be related directly to coin flipping which always has two outcomes, heads or tails. In some algorithms or pseudo-codes, it is considered that if flipping a coin with probability p results in heads, then the edge is stored in the memory and otherwise discarded.

3.3 Streaming Model

Data Stream is a series of data items arriving in an order, one at a time. It is well accepted model of computation for data analytics on massive data, and the goal is to compute a function over the entire stream as the items arrive. The computation

to process the whole stream often requires a lot of memory resources as it stores data items, in order to access them multiple times later. This is required for offline computation.

On the contrary, the streaming model focus on real-time processing of the data items as they arrive, and make a decision if each item will be processed or discarded. That means, not all items are kept in the memory which leads to less consumption of resources. Therefore, the result we get out of here is an approximation as exact answers are not possible because of the loss of data items. This study focus on the graph streams where each item is an edge, arriving at a time t in an arbitrary order.

3.4 Reservoir Sampling

Sampling techniques are used in streaming models to select a small portion of items to estimate characteristics for the whole population. It selects a data item to be processed or not based on the probability. In this study, the sampling of a graph stream refers to the selection of a sub-graph that resembles the characteristics of the whole graph, and the streaming model will process only the selected sub-graph. Reservoir Sampling [39] is a widely accepted sampling technique and forms the basis for many studies on triangle counting in graph streams. The key point of reservoir sampling is that at each step each item that has come through the stream has the same probability to be kept as the others.

Suppose we have a stream Σ of n items with items coming one at a time. We would like to choose k items out of the stream. For example, if we have space only for k items in the memory. The total number of items, n, might be unknown beforehand, and it can be very large. We want that each item has the same chance to be chosen.

Reservoir sampling provides a solution to this problem. It works as follows:

1. For the first k items, keep every one of them.

2. For $i > k$, when the i-th item arrives, with probability k/i keep it and simultaneously discard one of the old ones chosen at random - that is each has probability of $1/k$ to be discarded. Otherwise (with probability $1 - k/i$), discard it.

3. Continue with step 2 until the streaming is done.

3.5 Neighborhood Multisampling (NMS)

Algorithm 1 NMS

Input: A graph edge stream Σ

1: $L_1 \leftarrow \emptyset, L_2 \leftarrow \emptyset, Y \leftarrow \emptyset$
2: **for each** $e_i = (u, v) \in \Sigma$ **do**
3: Sample e_i with probability p, add to L_1
4: **if** $e_i \in N(L_1)$ **then**
5: With probability q, Add e_i to L_2
6: **for** every (e_j, e_k) where $e_j \in L_1$, $e_k \in L_2$ such that $time(e_j) < time(e_k)$ and (e_i, e_j, e_k) form a triangle **do**
7: Add the triangle to (e_i, e_j, e_k) to Y
8: Return $|Y|/pq$

Kavassery et al. [19] presented the algorithm by modifying Neighbourhood Sampling [27] using the multisampling approach. Neighbourhood Multisampling (NMS) randomly samples multiple edges and their neighbours and collect the edges that complete the triangles with the sampled edges.

To understand the intuition behind this algorithm, let's consider an example where we have two triangles, $t_1 = \{a, b, c\}$ and $t_2 = \{a, b, d\}$ that share an edge ab. Assume the order of the edges arriving in the stream is bc, ab, ca, ad, bd. NMS will sample t_1 if the first edge bc is sampled in L_1, and the edge ab is a neighbour of a sampled edge from L_1 and is sampled in L_2. When the edge ca arrives, the algorithm checks if there's an edge from L_1 (*i.e.* bc) and L_2 (*i.e.* ab) respectively, that forms a triangle. Similarly, t_2 will be counted on arrival of bd, if ad is sampled in L_1, and we already have ab sampled earlier in L_2, therefore, $< ab, ad, bd >$ forms a triangle. We discuss the workflow details of NMS below.

NMS works as follows: Maintain the edges set L_1, L_2 and fix the two probabilities p and q. For each edge e_i arriving in graph stream Σ, sample e_i with probability p and place it in L_1. If e_i is a neighbour of an edge in L_1, place it in L_2 with probability q. Moreover, if e_i forms a triangle with an edge say e_j from L_1 and e_k from L_2, such that e_j arrived before e_k, increment the counter of Y.

To get the estimate for total number of triangles in Σ, scale the count by dividing Y by pq. The pseudo-code can be found in Algorithm 1.

3.6 ThinkD / Trièst

Shin et al. [32] proposed $ThinkD_{Acc}$ that is an accurate version of *ThinkD* algorithms and falls in FM category. Since we are ony concerned with insertion-only graph streams for now, $ThinkD_{Acc}$ is exactly similar to the algorithm provided by Stefani et al. [35] called *Trièst-IMPR*. For brevity, we call this algorithm TS. It is described in Algorithm 2.

The idea behind this algorithm is to fully utilize the memory resources by sampling the edges directly without any probability until the reservoir S is full. Once S is full, the sampled edge will replace a random edge from S. Consider an example with triangles $t_1 = \{a, b, c\}$ and $t_2 = \{a, b, d\}$, and the edge stream arriving in the order ab, bc, ca, ad, bd. Consider the reservoir Size $|S| = 3$, TS will count t_1 for sure as the edges ab and bc will be stored in S considering it's not full, and ca has neighbouring edges in $S < ab, bc >$, that forms a triangle. Now let's consider $S = \{ab, bc, ca\}$, when the new edge arrives, it has to replace an existing edge from S randomly. Therefore, the triangle t_2 will be counted on arrival of bd, if both ad and ab still exist in S. We further discuss the workflow details of TS below.

TS work as follows: TS uses Reservoir Sampling for sampling edges for insertion only streams as explained in Section 3.4. Maintain a fixed memory reservoir S of size K, the estimation counter of global triangles, τ. For each edge $e_t = \{u, v\}$ arriving where $t \geq 0$, if $t \leq K$, then edge e_t is directly inserted into S. If $t > K$ and e_t is sampled with probability K/t, e_t replaces a random edge from S.

In Line 15, N_u^S denotes the Neighbours of vertex u in S. This gives us the neighbour edges in S for each vertex of e_t, and the intersection of it will result in the number of triangles incident on that edge. For every triangle found, the count of triangles is updated with the weighted increase of $\eta^{(t)} = Max\left\{1, \frac{(t-1)(t-2)}{K(K-1)}\right\}$, as shown in lines 16 - 18 of the algorithm. The triangle counter update step is called unconditionally on each edge arrival.

To get an estimate for total number of triangles in Σ, the algorithm stores it in τ which is already scaled up. Whenever the global triangle estimation is queried, the algorithm returns τ.

Algorithm 2 TRIÈST / THINKD (INSERTION ONLY)

Input: Insertion-only edge stream Σ, integer $K \geq 6$

$S \leftarrow \emptyset$, $t \leftarrow 0$, $\tau \leftarrow 0$
for each $(u, v) \in \Sigma$ **do**
    $t \leftarrow t + 1$
    UpdateCounters(u, v)
    if SampleEdge$((u, v), t)$ **then**
        $S \leftarrow S \cup \{(u, v)\}$
Function SAMPLEEDGE$((u, v), t)$
    if $t \leq K$ **then return** True
    else if FlipCoin(K/t) = "head" **then**
        $(u', v') \leftarrow$ random edge from S
        $S \leftarrow S \backslash \{(u', v')\}$
        return True
    return False
Function UPDATECOUNTERS$((u, v))$
    $N^S_{u,v} \leftarrow N^S_u \cap N^S_v$
    $\eta^{(t)} = Max\{1, \frac{(t-1)(t-2)}{K(K-1)}\}$
    for all $c \in N^S_{u,v}$ **do**
        $\tau \leftarrow \tau + \eta^{(t)}$

Chapter 4

Fixed Probability vs Fixed Memory Algorithms

This section discusses Fixed Probability and Fixed Memory algorithms, how they are different from one another and concludes that Fixed Probability algorithms are faster than their Fixed memory counterparts. The characteristics of both FP and FM algorithms are shown in Table 4.1.

4.1 Fixed Probability Algorithms

FP algorithms work as follows: Given a fixed probability p, the algorithm samples the edge from the stream with the probability p and add it to a reservoir, after which the triangle count step is executed for each of the edge in the stream. If a triangle is found, a variable Y is incremented that stores the triangle count in the sample. This continues until the edges in the stream arrive. Whenever the triangle estimates are

	Fixed Memory	Fixed Probability
Input Parameter	Memory Budget K	Probability p
Probability to sample edge	K/t	p
Use Reservoir Sampling?	✓	✗
Have memory growth limits?	✓	✗

Table 4.1: Characteristics of FM and FP algorithms

required, Y is scaled up by some scaling factor and is returned as the final estimate.

Algorithm 3 FIXED PROBABILITY PSEUDO-CODE

Input: A graph edge stream Σ, probability p

1: $R \leftarrow \emptyset, Y \leftarrow \emptyset$
2: **for each** $e_i = (u, v) \in \Sigma$ **do**
3: Sample e_i with probability p, add to R
4: **for** every e_i, perform a triangle counting step. If a triangle is found **do**
5: Add the triangle to Y
6: Return $|Y|/p^2$

We present a pseudo-code for FP category that can be found in Algorithm 3. The pseudo-code shows the basic idea for all of existing FP algorithms. The only thing we should consider here is the sampling step in Line 3 of the pseudo-code as the FP or FM categorization is independent of triangle counting step. The edge is sampled with the probability p and is added to the reservoir R, which is an available memory to the algorithm. The triangle count step in Line 4-5 may vary across the FP algorithms.

Stefani et al. [35] discussed the drawbacks of the algorithms employing FP approach. They presented an FM algorithm called *Trièst* as discussed in Section 3.6. They highlighted the drawbacks to show that FM algorithms have better scalability than FP algorithms.

The limitations of FP algorithms are as follows.

- The input parameter p that is required to be specified by the user in advance is not certain. In-depth analysis is required to get the desired approximation quality.
- The reservoir size $|R|$ always grows as there is no limit or restriction to store the number of sampled edges. So, as the stream grows, the number of sampled edges in the reservoir also grows.
- If p is chosen to be large, the algorithm may run out of memory.
- If $p$is chosen to be small, the algorithms will provide us suboptimal estimates.

These limitations definitely show that FM algorithms have an edge over FP algorithms. But, we analyzed that there is one specific characteristic of FP algorithms that makes them better than FM algorithms and we will discuss this in the next section.

4.2 Fixed Memory Algorithms

FM algorithms work as follows: Given a fixed memory budget K, the algorithm samples an edge with probability K/t where t is the time of arrival of an edge, after which the triangle count step is executed for each edge in the stream. Unlike FP algorithms, FM algorithms scale up the count of triangles found for each edge by η and then add it to Y which is the triangle estimation. This continues until the edges in the stream arrive. Whenever the triangle estimates are required, Y is returned directly as the final estimate.

Algorithm 4 Fixed Memory Pseudo-code

Input: A graph edge stream Σ, memory budget K.

1: $R \leftarrow \emptyset, Y \leftarrow 0, t \leftarrow 0$
2: **for each** $e_i = (u, v) \in \Sigma$ **do**
3: $\quad t \leftarrow t + 1$
4: $\quad$ **if** $|R| < K$ **then**
5: $\quad\quad$ Add e_i to R
6: $\quad$ **else if** coin(K/t) = "head" **then**
7: $\quad\quad$ Replace a random edge in R with e_i
8: $\quad$ **for** every e_i, perform a triangle counting step. If a triangle is found **do**
9: $\quad\quad$ $\eta^{(t)} = Max\left\{1, \frac{t^2}{K^2}\right\}$
10: $\quad\quad$ $Y \leftarrow Y + \eta^{(t)}$
11: Return Y

We present a pseudo-code for FM category that can be found in Algorithm 4. It shows the general idea for all of the existing FM algorithms. Also as described in the previous section, the FP or FM categorization is independent of the triangle counting step, so we should focus only on the way edges are sampled. As seen in Lines 4-5, the edges will be directly added to the reservoir R is it is empty, there is no sampling happening until this point in the algorithms. The sampling of the edges starts in Line 6 of the pseudo-code, where the edge is sampled by the fraction of the fixed memory budget size K and the time of arrival t of that edge. If the edge is sampled with probability K/t, then it replaces a random edge from R. The triangle count step in Lines 8-10 may vary across the FM algorithms.

We analyzed and found that there are also some drawbacks for FM algorithms as listed below.

- The selection of input parameter, memory budget K is not certain and this is similar to choosing input parameter p for FP algorithms. It needs some in-depth analysis to identify the correct value of K to obtain the desired approximation quality.
- If K is small, the triangle estimates we get have low accuracy.
- if K is large, the algorithm takes longer to run.
- We further observed that FP algorithms are faster than FM algorithms. There is a significant difference between the running time of both. To prove this, we provide experimental analysis and time complexity comparison of both FM and FP algorithms in the next section.

4.3 Analysis

This section provides the detailed analysis of how and why FP algorithms are faster than FM algorithms. We also present a new algorithm that is a FM version of the original NMS (discussed in section 3.5), which is an FP algorithm. We also provide the scalability comparison of FM and FP algorithms and conclude that FP algorithms are more time efficient than FM algorithms.

FM algorithms first directly fill the reservoir with the edges streamed. Once the reservoir is full, the edge is sampled by probability K/t and is replaced with a random edge in the reservoir. Hence, the reservoir remains almost full always.

In FP algorithms, there is no such reservoir with fixed size. The fixed probability p specified by the user plays an important role here to store the sampled edges in the memory. Let's say there are $|E|$ edges in graph stream Σ, the number of edges sampled by the algorithm will be $p|E|$.

Considering that the size of K in FM algorithms is chosen to be 1% of the Σ, it is equivalent to assigning p to be 0.01 for FP algorithms. Similarly, $K = 10\%$ of Σ is equivalent to $p = 0.1$.

4.3.1 Experimental Comparison

The triangle counting step in both kinds of algorithms is the most time consuming. To check if the triangle exists for an edge (u, v), the neighbours of u and v are filtered out of the reservoir, and then the common neighbours filtered out to identify the triangles incident on the edge (u, v). In order to compare the running time of FM algorithms with the FP algorithms, we analyzed the computation time for each edge in these algorithms. We ran an experiment measuring the computation time taken per edge and plotted the results in Figure 4.1.

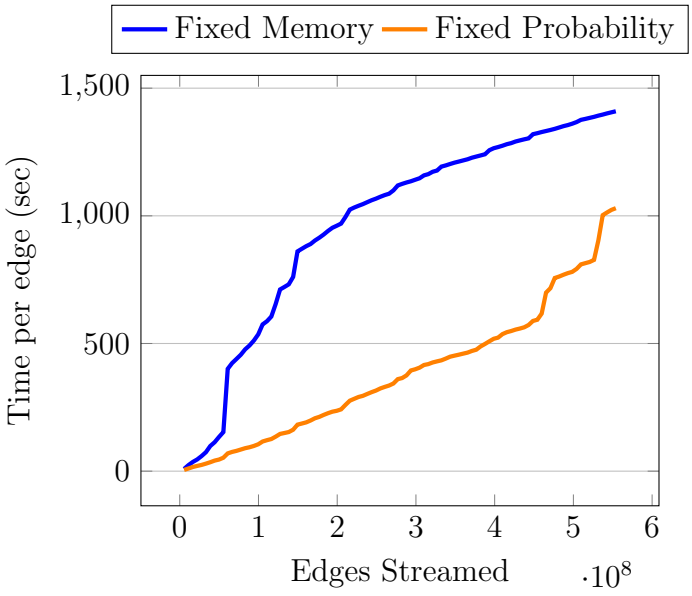


Figure 4.1: Comparison for each edge processed by Fixed Memory and Fixed Probability algorithms

It is observed that FM algorithms at first require more time to compute and gradually it becomes constant, whereas, FP algorithms computation time increases at a steady rate until the graph stream ends. FM algorithm initially sees a steep rise in the computation time per edge.

The reason behind this significantly higher running time in FM algorithms is that an edge on arrival has more neighbours readily available to be traversed in the reservoir as the first K edges are stored directly in the reservoir in the case of FM algorithms. Also, once the reservoir is full, the edge replacements add to the running time in the case of FM algorithms. The initial linear growth in computation time per edge in FP algorithms is because the reservoir initially is empty and the edges are

sampled uniformly with a fixed probability and added to it. Hence, the neighbours to be traversed in the triangle count step are not many at the beginning, like in FM algorithms. Once, the reservoir has grown bigger, FP algorithms have more neighbor edges to process due to which we see an irregular pattern at the end.

4.3.2 Neighbourhood Fixed Memory Multisampling

In the previous section, we have analyzed the experimental comparison of FM and FP algorithms. We observed that the way edges are sampled in FM and FP algorithms make the whole difference in the computation time. To analyze this further, we plan to compare the FP and FM variant of the same algorithm, where the triangle counting step would be similar, but the sampling method would vary based on FM or FP category.

Therefore, we designed and implemented an FM variant of the NMS algorithm (discussed in Section 3.5) that uses a Fixed Probability approach originally. Our variant uses the Reservoir Sampling approach with Fixed Memory Budget to sample the edges, which is exactly similar to the other FM algorithms. In this approach, the algorithm would be certain of the sample size to be kept in the memory as we would have a fixed dedicated memory assigned to it. The pseudo-code can be found in Algorithm 5. For brevity, we will call this algorithm, NFMS.

We compare NFMS and NMS to analyze their computational time growth trend. This comparison makes more sense to validate our claim that FP algorithms are faster than FM algorithms, because, the triangle counting step is similar for both algorithms and the only difference is the way edges are sampled. Let us understand the technical details of NFMS first and then discuss the results.

NFMS works as follows: Maintain the reservoirs L_1 and L_2. For each edge e_i arrives in graph stream Σ at time i, sample e_i if $|L_1| < K$ else we sample e_i with probability K/t and replace it with a random edge in L_1. If e_i is a neighbour of an edge in L_1, sample e_i if $|L_2| < K$ else we sample e_i with probability K/c and replace it with a random edge in L_1, where c is the arrival time of neighbour edge.

Moreover, if e_i forms a triangle with an edge from L_1 and L_2, place e in L_3. Compute all the triangles, say Y, with edges e_1, e_2, e_3 such that $e_1 \in L_1, e_2 \in L_2, e_3 \in L_3$ and e_1 arrived before e_2, and e_2 arrived before e_3. Whenever the triangle is found, the count Y is incremented by tc/K^2. To get the estimate for total number of triangles in Σ, return Y.

Algorithm 5 Neighbourhood Fixed Memory Multisampling

Input: A graph edge stream Σ, memory budget K.

$L_1 \leftarrow \emptyset, L_2 \leftarrow \emptyset, Y \leftarrow 0$
for each $e_i = (u, v) \in \Sigma$ **do**
    $t \leftarrow t + 1$
    if $|L_1| < K$ **then**
        Add e_i to L_1
    else if coin(K/t) = "head" **then**
        Replace a random edge in L_1 with e_i
    if $e_i \in N(L_1)$ **then**
        $c \leftarrow c + 1$
        if $|L_2| < K$ **then**
            Add e_i to L_2
        else if coin(K/c) = "head" **then**
            Replace a random edge in L_2 with e_i
    for every $(e_j, e_k), e_j \in L_1$, $e_k \in L_2$ such that $time(e_j) < time(e_k)$ and (e_i, e_j, e_k) form a triangle **do**
        $\eta^{(t)} = Max\left\{1, \frac{tc}{K^2}\right\}$
        $Y \leftarrow Y + \eta^{(t)}$
Return Y

We provide the formal description of the NFMS algorithm but do not discuss theoretical or in-depth experimental analysis, except for the computational time experiment. This is because, we anticipate that NFMS (FM algorithm) will have poor performance in comparison to NMS (FP algorithm), hence we do not proceed with detailed analysis. Our purpose is to just verify the claim that FP algorithms are faster than FM algorithms and we will do so by comparing the running time of NFMS with NMS. The results are discussed below.

We ran experiments on Dewiki graph containing more than 33 million edges, for both NFMS and NMS. We observed that NFMS took 478.24 seconds to execute if we store 20% of the edges in the reservoir, whereas, NMS just takes 196.79 seconds for the same experiment. Therefore, NFMS computational time is higher than NMS even though the triangle counting step is the same. We observed that in the case of NFMS, there are more number of comparisons during the triangle counting step as it has more neighbour edges ready in the reservoir to be traversed since the beginning, however, that is not the case for NMS. Also, NFMS followed the same computation trend as in FM algorithms shown earlier in Figure 4.1.

4.3.3 Time Complexity Comparison

Now that we have seen experiments confirming FP algorithms are faster than FM algorithms, we further wanted to validate it by comparing the time complexities of the algorithms from both categories. We choose NMS algorithm (detailed in Section 3.5) from FP category and TS algorithm (detailed in Section 3.6) from FM category for comparison. The time complexities of both algorithms are discussed below.

Time Complexity for TS

As detailed earlier, TS algorithm stands for ThinkD/Triest Sampling. Shin et al. [32] proposed ThinkD whose insertion-only version is exactly similar to an algorithm proposed by Stefani et al. [35] called Triest. Shin et al. [32] reported the time complexity of their algorithm as $O(Kt)$ to process first t elements in Σ and K is the reservoir size, which is a loose bound. Stefani et al. [35] provides a time bound for each edge. For each edge to compute triangles, algorithm requires $O(d(u) + d(v))$ steps.

Time Complexity for NMS

Kavassery et al. [19] does not provide the theoretical time bound proofs for NMS algorithm. We analyzed the NMS algorithm to measure the time complexity and below is our contribution.

Theorem 1 (**Time Complexity for NMS**). *Let R be the number of edges that arrived in the stream, p and q be the fixed probabilities to sample the edges. The time complexity of NMS algorithm is*

$$(p+q)\sum_{u,v\in R}(d(u)+d(v)+c)$$

where d(u) is the degree of vertex u in the reservoir, and c is the constant.

Proof. Suppose that R is the number of edges processed by NMS Algorithm 1, p is the probability to sample L_1 edges, q is the probability to sample L_2 edges, $d(u)$ is the degree of the vertex u in the reservoir, and c is a constant.

The running time of NMS algorithm is dependent on the degree of the vertices of the edges sampled. The triangle count is calculated by traversing the edges from both L_1 and L_2 set, which are stored in the memory. When an edge (u, v) arrives, the common neighbours are searched for each vertex of that edge in L_1 and L_2 set which

takes $pd(u) + qd(v)$ and $pd(v) + qd(u)$ steps. For computing the time complexity of this algorithm, we take the sum for all the edges processed which is shown below.

Time complexity of NMS algorithm is:

$$\sum_{u,v \in R}(pd(u) + qd(v) + pd(v) + qd(u) + c)$$

which can be simplified to

$$(p + q) \sum_{u,v \in R}(d(u) + d(v) + c)$$

■

Comparison

While comparing the time complexity of NMS and TS, it is clear that they both depend upon the sum of degree of the vertices of an edge arriving in the graph stream *i.e.* $O(d(u) + d(v))$. We know that the FM algorithms store more number of edges in the reservoir all the time, whereas, the FP algorithms gradually increases the stored edges in the memory. Therefore, the computation of the shared neighbourhood will be more in FM algorithms. Hence, it concludes the higher computational times in case of FM algorithms.

In conclusion, we have shown that the FM algorithms are slow by their nature. We validated this claim by providing experimental analysis and the time complexities comparisons in the previous sections. Therefore, this claim adds up to be another limitation of FM algorithms. We present the Hybrid approach in the next Chapter that helps us overcoming these limitations and proves to be better than both FM and FP approaches.

Chapter 5

The Hybrid Approach and Algorithms

The key idea behind the hybrid approach is to utilize the benefits of both FM and FP approaches and eliminate their limitations. The hybrid approach algorithms have the combination of the fixed memory budget K and the fixed sampling probability p. We present two new algorithms in this category. We also provide a detailed analysis of these algorithms and their comparison with FM and FP algorithms.

Before moving further directly to the new algorithms, let us consider why the hybrid approach is better than FP and FM algorithms. As we have already discussed the limitations of FP and FM approaches in the previous chapter, the questions below would answer how hybrid algorithms overcome them.

Will hybrid algorithms ever run out of memory like FP algorithms?

The hybrid approach limits the size of an edge reservoir by K, similar to FM algorithms, whereas in FP algorithms, there is no such limit and the edge reservoir size always grows. Hence, we overpower FP algorithm's biggest limitation.

Are hybrid algorithms faster than both FM and FP algorithms?

We have already detailed how FP algorithms are faster than FM algorithms in Chapter 4. Hybrid algorithms follow the same trend as FP algorithms until the first K edges in the memory are stored. After the first K edges in FP algorithms, the number of edges keep on growing in the memory and will have more number of neighbours stored, whereas in the hybrid approach, the limit on memory reservoir is set, due to

which the traversal time becomes almost constant and does not grow much. Hence, hybrid algorithms are faster than both FP and FM algorithms.

How should we select the value of the input p and K in hybrid algorithms together?

The selection of the input parameter values is one of the limitations for both FM and FP algorithms. To reason this question better, we first need to understand how do we decide on the input parameters selection for both FM and FP algorithms.

FM algorithms need the edge reservoir size K as an input. The user has to decide how many number of edges are to be stored in the memory. Considering the user is willing to store 10% of the edges in memory, the user should provide a number that is 10% of the total edges of a graph. If the graph properties are unknown beforehand, the user might provide any number and in case the number is too small with respect to the graph's size, the output of the estimation will be inaccurate.

On the other hand, FP algorithms expect a fixed probability as an input parameter. Considering that the size of K in FM algorithms is chosen to be 1% of the Σ, it is equivalent to assigning p to be 0.01 for FP algorithms. Similarly, $K = 10\%$ of Σ is equivalent to $p = 0.1$. Now, if a user is willing to store 10% of the edges in the graph, the input provided should be 0.1. But this will always grow the number of edges stored in the memory and crash the algorithm once the system memory is exhausted.

Hybrid algorithms expect the input parameters similarly. Considering a user wants to store 10% of the edges in memory, the value of K should be 10% of the total number of edges and p should be 0.1. In case some other values are provided, hybrid algorithms will have two possibilities as detailed below:

- If K is small or p is large, the algorithm will not go beyond the K number of the edges in the memory, similar to FM algorithms.
- If K is large or p is small, the algorithm has more than enough memory to finish its work.

These characteristics of input parameters for Hybrid algorithms prove to be better than FM and FP approaches. Considering the advantages of the hybrid approach over FM and FP approaches, We developed the two new algorithms that belong to the hybrid category, and are detailed in the subsequent sections.

5.1 Neighborhood Hybrid Multisampling (NHMS)

Neighborhood Hybrid Multisampling (NHMS) is the hybrid variant of NMS algorithm discussed in Section 3.5. The intuition behind this algorithm is exactly same as NMS algorithm, the only difference is the way edges are sampled using a hybrid approach. To understand this better, let's consider an example where we have two triangles, $t_1 = \{a, b, c\}$ and $t_2 = \{a, b, d\}$ that share an edge ab. Assume the order of the edges arriving in the stream is bc, ab, ca, ad, bd. NHMS will sample t_1 if the first edge bc is sampled in L_1, and the edge ab is a neighbour of a sampled edge from L_1 and is sampled in L_2. When the edge ca arrives, the algorithm checks if there's an edge from L_1 (*i.e.* bc) and L_2 (*i.e.* ab) respectively, that forms a triangle. Similarly, t_2 will be counted on arrival of bd, if ad is sampled in L_1, and we already have ab sampled earlier in L_2, therefore, $\langle ab, ad, bd \rangle$ forms a triangle. We discuss the technical details of NHMS below.

The algorithm requires probabilities p and q just like NMS, along with a memory budget K. We maintain two edge reservoirs L_1 and L_2. In our hybrid approach, we limit the size of both the reservoirs to $K/2$.

When an edge e_i arrives from graph stream Σ and sampled with probability p, it is placed in L_1 if there's a vacant room in L_1 reservoir, otherwise, it replaces a random edge from L_1. Then, if e_i is a neighbour of an edge in L_1 and sampled with probability q, it is placed in L_2 if there's a vacant room in L_2 reservoir, otherwise, it replaces a random edge from L_2. Moreover, if a triangle is formed by an edge e_i with an edge from L_1 and L_2, the Y counter is incremented. The triangle count step is similar to the NMS algorithm. Whenever a triangle estimate is needed, the variable Y is scaled up by dividing it with pq and is returned as the final triangle estimation. The pseudo-code can be found in Algorithm 6.

Lines 3-7 explain the sampling on an edge in L_1 reservoir, whereas lines 8-13 explains the sampling of edges into L_2 reservoir, only if the edge arrived is the neighbour of any edge already sampled in L_1. Lines 14-15 details the triangle counting step, which is exactly similar to the NMS algorithm.

Algorithm 6 NHMS

Input: A graph edge stream Σ, memory budget K, probabilities p and q.

$L_1 \leftarrow \emptyset, L_2 \leftarrow \emptyset, Y \leftarrow \emptyset$
for each $e_i = (u, v) \in \Sigma$ **do**
    if coin(p) = "head" **then**
        if $|L_1| < K/2$ **then**
            Add e_i to L_1
        else
            Replace a random edge in L_1 with e_i
    if $e_i \in N(L_1)$ **then**
        if coin(q) = "head" **then**
            if $|L_2| < K/2$ **then**
                Add e_i to L_2
            else
                Replace a random edge in L_2 with e_i
    for every (e_j, e_k) where $e_j \in L_1$, $e_k \in L_2$ such that $time(e_j) < time(e_k)$ and (e_i, e_j, e_k) form a triangle **do**
        Add the triangle (e_i, e_j, e_k) to Y
Return $|Y|/pq$

It can be formally verified that NHMS produces unbiased estimates when the reservoirs used are not full. When the reservoirs are full, we still do not observe any bias in the estimations produced by our algorithms, and in practice, our estimations are equal or better than those produced by FM and FP algorithms. While we are not able to show the unbiasedness of our algorithms formally, based on our extensive experiments with large datasets and reservoir sizes of only 1% of the datasets, we conjecture that our algorithms are unbiased or exhibit negligible bias.

Theorem 2 (**Time Complexity for NHMS**). *Let p and q be the fixed probabilities to sample the edges. The time complexity to process an edge $e = (u, v)$ arriving in the stream by Algorithm 6 is*

$$(p + q) \cdot O(d(u) + d(v))$$

where d(u) is the degree of vertex u in the reservoir.

Proof. Suppose that R is the number of edges processed by Algorithm 6, p is the probability to sample L_1 edges, q is the probability to sample L_2 edges, K is the memory budget, and $d(u)$ is the degree of the vertex u in the reservoir.

The running time of NHMS algorithm is dependent on the degree of the vertices of the edges sampled in L_1 and L_2. The triangle count is calculated by traversing the edges from both L_1 and L_2 set, which are stored in the memory. When an edge (u, v) arrives, the common neighbours are searched for each vertex of that edge in L_1 and L_2 set which takes $pd(u) + qd(v)$ and $pd(v) + qd(u)$ steps. This can be further reduced to $(p + q) \cdot O(d(u) + d(v))$.

■

Theorem 3 (**Space Complexity for NHMS**). *Let K be the number of edges to be stored on the memory. The space complexity of Algorithm 6 is $O(K)$.*

Proof. Let K be the number of edges to be stored in the memory. The algorithm maintains two edge reservoirs L_1 and L_2 with the maximum size limit of $K/2$. Therefore, the total memory consumed by algorithm will be $O(K)$.

■

5.2 Trièst / ThinkD Hybrid Sampling (THS)

Trièst/ThinkD Hybrid Sampling (THS) is the hybrid variant of the TS algorithm discussed in Section 3.6. The algorithm requires a memory budget K just like TS, along with fixed probability p. We maintain an edge reservoir S which would have a maximum size limit K, the triangle counter variable Y.

The intuition behind this algorithm is somewhat similar to NHMS algorithm, the only difference is that we just have one reservoir in this case. Consider an example with triangles $t_1 = \{a, b, c\}$ and $t_2 = \{a, b, d\}$, and the edge stream arrive in an order ab, bc, ca, ad, bd. TS will count t_1 if the edges ab and bc will be stored in S considering it's not full, and ca has neighbouring edges in S $\langle ab, bc\rangle$, that forms a triangle. Now let's consider $S = \{ab, bc, ca\}$ and the reservoir is full, when the new edge arrives, it has to replace an existing edge from S randomly. Therefore, the triangle t_2 will be counted on arrival of bd, if both ad and ab still exists in S. We further discuss the technical details of TS below.

THS algorithm is very easy to understand. When an edge $e = (u, v)$ arrives from graph stream Σ, the Update function is called which checks if there's any triangle

Algorithm 7 THS

Input: A graph stream Σ, sampling probability p

```
1:  S ← ∅, Y ← 0
2:  for each pair e = (u, v) in Σ do
3:      Update(u, v)
4:      Insert(u, v)
5:  Estimate(Y)
6:  Function UPDATE(u, v)
7:      for each w ∈ N(u) ∩ N(v) do
8:          Y ← Y + 1
9:  Function INSERT(u, v)
10:     if coin(p) = "head" then
11:         if |S| < K then
12:             S ← S ∪ {(u, v)}
13:         else
14:             Replace a random edge in S with (u, v)
15: Function ESTIMATE(Y)
16:     return Y/p^2
```

formed by the edge. The triangles are counted if there is a common neighbour of the vertices u and v available in S. If the triangle is found, Y is incremented by one. Then, the Insert function is called to sample the e. If e is sampled with probability p, it is placed in S if there's a vacant room in S reservoir, otherwise, it replaces a random edge from S. Whenever a triangle estimate is needed, the variable Y is scaled up by dividing it with p^2 and is returned as the final triangle estimation. The pseudo-code can be found in Algorithm 7.

Line 6-8 shows from Algorithm 7 explains the way triangles are counted by THS. For each edge that arrives, the intersection of the neighbours of each vertex of an edge is filtered out from the reservoir that stores the sampled edges, and then the triangle counter is incremented by the size of an intersection set. Line 9-14 explains the sampling of an edge into the reservoir.

It can be formally verified that THS produces unbiased estimates when the reservoirs used are not full. When the reservoirs are full, we still do not observe any bias in the estimations produced by our algorithms, and in practice, our estimations are equal or better than those produced by FM and FP algorithms. While we are not able to show the unbiasedness of our algorithms formally, based on our extensive experiments with large datasets and reservoir sizes of only 1% of the datasets, we

conjecture that our algorithms are unbiased or exhibit negligible bias.

Theorem 4 (**Time Complexity for THS**). *Let p be the fixed probability to sample the edges. The time complexity to process an edge $e = (u, v)$ arriving in the stream by Algorithm 7 is*

$$p \cdot O(d(u) + d(v))$$

where $d(u)$ is the degree of vertex u in the reservoir.

Proof. Suppose that R is the number of edges processed by Algorithm 7, p is the probability to sample S edges, K is memory budget, and $d(u)$ is the degree of the vertex u in the reservoir.

The running time of this algorithm is dependent on the degree of the vertices of the edges sampled in the memory budget K. The triangle count is calculated by traversing the common neighbours of the vertices of an edge. When an edge (u, v) arrives, the common neighbours are searched for each vertex of that edge in S set which takes $p \cdot O(d(u) + d(v))$ steps.

If $|R| > K$, the algorithm for the intersection of common neighbours requires $O(d(u) + d(v))$ time in the worst case, where the degrees are w.r.t. the graph formed by a stream so far. ■

Theorem 5 (**Space Complexity for THS**). *Let K be the number of edges to be stored on the memory. The space complexity of Algorithm 7 is $O(K)$.*

Proof. Let K be the number of edges to be stored in the memory. The algorithm maintains an edge reservoir S with the maximum size limit of K. Hence, the total memory consumed by the algorithm will be $O(K)$.

■

Chapter 6

Evaluation, Analysis, and Comparisons

In this chapter, we discuss evaluation experiments and results for the two new algorithms proposed in the previous chapter. The experimental study has been performed on undirected graphs and the comparison is provided between our algorithms and NMS (discussed in section 3.5) and TS (discussed in section 3.6). Section 6.1 provides the details of various datasets used for experimental analysis. In Section 6.2, we have discussed the details of resources used for conducting these experiments. Finally, SectionS 6.3 and 6.4 give the experimental results and analysis based on comparison of the algorithms mentioned.

6.1 Datasets

We perform the evaluation of these algorithms on six real word datasets of varied sizes. All these datasets have been downloaded from the Laboratory of Web Algorithms which provides the compressed form of large datasets using WebGraph [6, 7]. These graphs are then symmetrized and any self-loop is deleted to get the corresponding simple undirected graphs.

The graphs used for experimentation have varying densities ranging from small graphs (Enron with 254,449 edges) to very large graphs (Arabic with 550 million edges). We also performed scalability tests on all these algorithms using a Twitter graph having 1.2 billion edges. Below is a brief overview of each of the datasets:

1. **enron**: made available by Federal Energy Regulatory, that contains email communications among Enron employees.
2. **cnr-2000**: a small crawl from the Italian CNR (Consiglio Nazionale delle Ricerche).
3. **dblp-2011**: Digital Bibliography and Library Project is a popular service which provides the bibliography for published papers.
4. **dewiki-2013**: This graph represents the Wikipedia in German (https://de.wikipedia.org/wiki/Wikipedia:Hauptseite).
5. **ljournal-2008**: abbreviated as **ljournal**, a snapshot from LiveJournal (https://www.livejournal.com/) in 2008.
6. **arabic-2005**: a crawl of websites that contain Arabic, performed by UbiCrawler [5] in 2005.
7. **twitter-2010**: a snapshot of Twitter follow connections [22] in 2010.

Dataset	Nodes	Edges	Triangles
enron	69,244	254,449	1,067,993
cnr	325,557	2,738,969	20,977,629
dblp	986,324	3,353,618	7,005,235
dewiki	1,532,354	33,093,029	88,611,129
ljournal	5,363,260	49,514,271	411,155,444
arabic	22,744,080	553,903,073	36,895,360,842
twitter	41,652,230	1,202,513,046	34,824,916,864

Table 6.1: Summary of real-world graphs

Table 6.1 shows the summary of real world graphs used for the experiments. These graphs downloaded are the compressed files that need to be decompressed in the memory using Webgraph framework, and also we normally need more memory space than just for the dataset. Nonetheless, the WebGraph framework enables us to load the data into the memory part by part. Using this tool, and about 10 GB

memory allocated to the algorithms, we were able to process twitter on our proposed algorithms as well.

6.2 Experimental Settings

The experiments are conducted on Intel Xeon Server with the following configuration:

- Processor: Intel$^{(R)}$ Xeon$^{(R)}$ CPU E5620 @ 2.40 GHz
- Memory: 64 GB RAM
- OS: Ubuntu 14.04.1 LTS

We implemented and executed all algorithms in Java 8. We also used Webgraph framework [7] because of its great compression ratio in saving or loading graphs. Even though the Xeon server had 64GB RAM, we did not change the default memory settings of JVM. Server JVM heap configuration is 1/4 of the total System memory available. Hence, the allocated memory that can be used by the whole implementation of algorithm will be the maximum of 16GB. We had to tweak the JVM heap size for running the original NMS implementation provided by authors on Large graphs like Arabic and Twitter, as it creates a lot of objects and does not scale up in 16GB RAM.

The comparison of all the algorithms is done considering the total number of edges sampled by the respective algorithms. We followed this criteria to get a fair comparison of all the algorithms (irrespective of FP, FM, or Hybrid approach) as to figure out how the performance of the algorithms looks like when they store the same number of edges. Based on this, our results are discussed in the next section.

6.3 Results

The experiments performed measure the key performance metrics, *i.e.* Accuracy and Running time of our algorithms detailed in Chapter 5. The section 6.3.1 presents the results for NHMS and section 6.3.2 presents THS results.

6.3.1 NHMS

We ran our implementation of NHMS on the graphs as described in Section 6.1 in 5 different settings. These settings vary by the sample size of the graph, to keep it

Graphs	Edges Sampled	Runtime(sec)	Error rate (%)
enron	50889	0.99	0.71
cnr	547793	15.15	0.82
dblp	670723	3.65	0.16
dewiki	6618605	178.21	0.09
ljournal	9902854	178.99	0.04
arabic	110774765	8352.94	0.03

Table 6.2: Performance results of various graphs for NHMS while storing 20% of edges in memory.

generic for the comparison between different algorithms, we choose 1%, 5%, 10%, 15% and 20% of the graph stream size. The five iterations of each experiments are run and the results are averaged. The results for sampling 20% of edges are shown in Table 6.2. The detailed results for all the test cases executed can be found in Appendix A.

6.3.2 THS

We kept the settings for THS same as for NHMS. We choose five different settings for experiments: 1%, 5%, 10%, 15% and 20% of the graph stream size. The five iterations of each experiments are run and the results are averaged. The results for sampling 20% of edges are shown in Table 6.3. The detailed results for all the test cases executed can be found in Appendix B.

Graphs	Edges Sampled	Runtime(sec)	Error rate (%)
enron	50889	0.29	0.91
cnr	547793	1.71	0.70
dblp	670723	2.99	0.38
dewiki	6618605	31.37	0.11
ljournal	9902854	45.24	0.04
arabic	110774765	743.21	0.04

Table 6.3: Performance results of various graphs for THS while storing 20% of edges in memory.

6.4 Comparison

This section lists out various experiments performed to evaluate the performance of the algorithms proposed and compare them with the best available algorithms published in the literature. For comparison, we choose NMS as it belongs to FP category, TS as it belongs to FM category, and our two proposed algorithms: NHMS and THS, both belong to the Hybrid category. Section 6.4.1 compares the triangle estimations accuracy between the algorithms for all of the graph datasets. The goal of the second part of the experiment (section 6.4.2) is to compare the computation time for all of the algorithms. We also provide the Scalability test in section 6.4.3 on Twitter graph that has more than 1.2 billion edges.

6.4.1 Accuracy

We ran an experiment by sampling 1%, 5%, 10% 15% and 20% of the graph size in respectively on THS, NHMS, TS and NMS algorithms and compared their accuracy. The results are plotted in Figure 6.1 for all the graph datasets we used. Clearly, both THS and NHMS have better accuracy than their counterparts. It is important to note that the difference in accuracy is not bigger in large graphs as the error rate is below 0.5% for all the test cases. Even if TS and NMS algorithms are highly accurate, NHMS and THS still have even better accuracy. The error rate results can be found in Table 6.4. It is clear from Figure 6.1 that NHMS has the most accurate results, even better than THS, in most of the test cases.

Graphs	NHMS	THS	NMS	TS
enron	3.93	4.41	5.80	7.58
cnr	4.77	4.46	4.55	8.87
dblp	2.07	2.64	2.32	1.88
dewiki	0.89	1.09	1.02	0.68
ljournal	0.27	0.16	0.11	0.32
arabic	0.27	0.15	0.28	0.22

Table 6.4: Error Rate (%) results of various graphs for NHMS, THS, NMS and TS while storing 1% of edges in memory.

6.4.2 Running time

As we detailed in Accuracy section, we ran an experiment in similar setting *i.e.* by sampling 1%, 5%, 10% 15% and 20% of the graph size in respectively on THS, NHMS, TS and NMS algorithms and compared their running time. We ignored the edge arrival time from the input graph stream to accurately measure the run-time performance of the algorithms.

The results are shown in Table 6.5 and plotted in Figure 6.2 for all the graph datasets that we used. Both THS and NHMS are faster than their counterparts. The results clearly validate our claim we made in Section 5 that Hybrid algorithms are faster than both FM and FP algorithms.

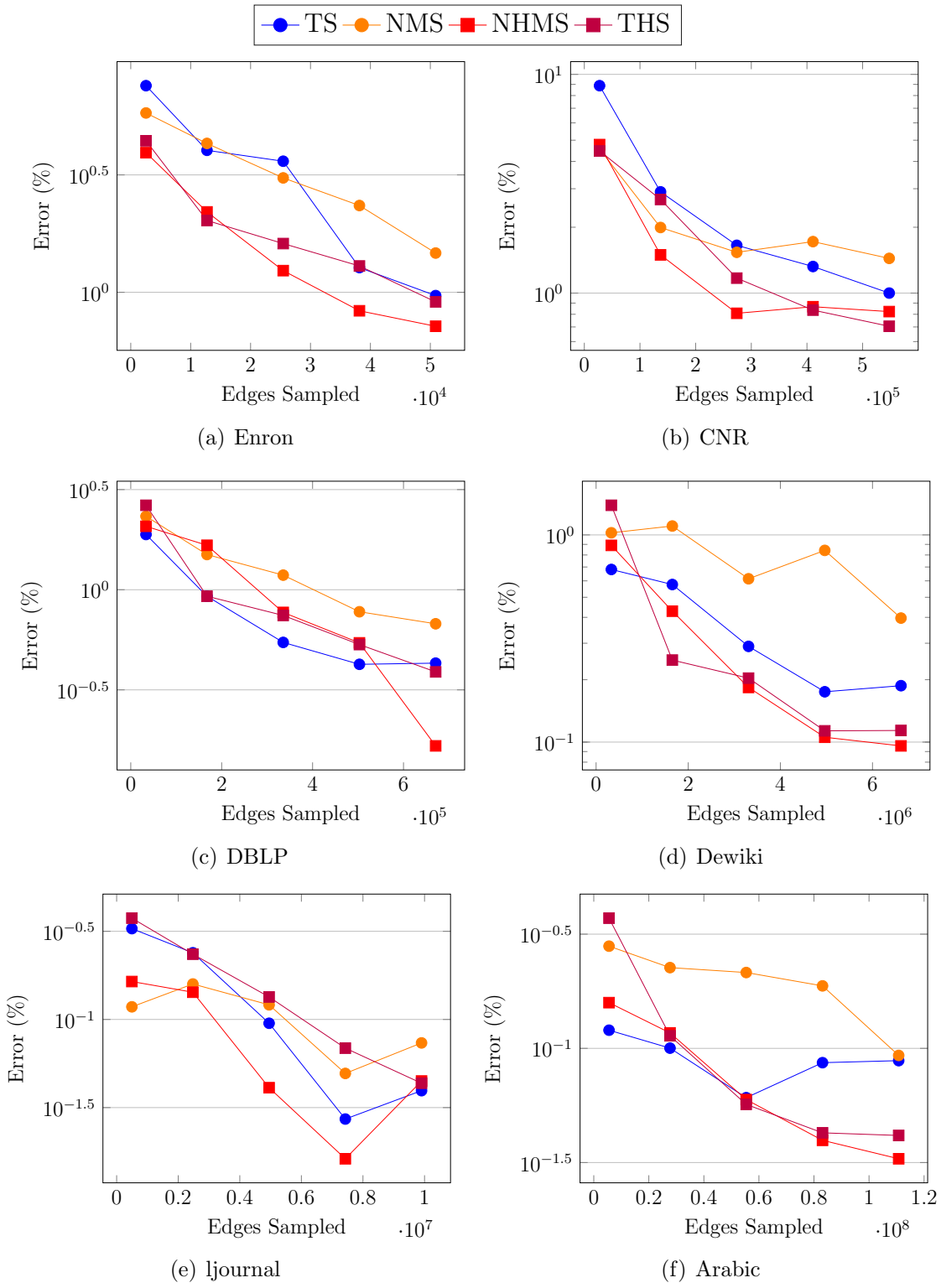
Comparison of FM vs FP approach: As TS is FM algorithm and NMS is FP algorithm, we can observe from Figure 6.2 that NMS is faster than TS. This validate our claim from our analysis provided in Section 4.

Comparison of THS and TS (Trièst / ThinkD): THS proves to be **5 - 10X faster** than its counterpart TS. Hence, the results validates our claim that Hybrid algorithms are faster than FM algorithms.

Comparison of NHMS and NMS: There is only a slight difference between NHMS and NMS. The reason behind this is that NMS is already an FP algorithm and Hybrid algorithms follow the same execution trend as FP algorithms until the first k edges are processed, after which the increase in time becomes constant. This would be more clear in the next section in scalability. Hence, NHMS is faster than NMS algorithms after the first k edges are processed.

Comparison of THS and NHMS: Among our algorithms, THS outperformed NHMS considering the running time, but there's consistently a trade-off among both between run-time and accuracy. NHMS is more accurate whereas THS is much faster than NHMS.

Overall, both the hybrid algorithms proposed by us have better running time in comparison to both FM and FP algorithms. THS outperformed all algorithms including NHMS.



(a) Enron (b) CNR

(c) DBLP (d) Dewiki

(e) ljournal (f) Arabic

Figure 6.1: Error Rate for algorithms TS, NMS, NHMS and THS

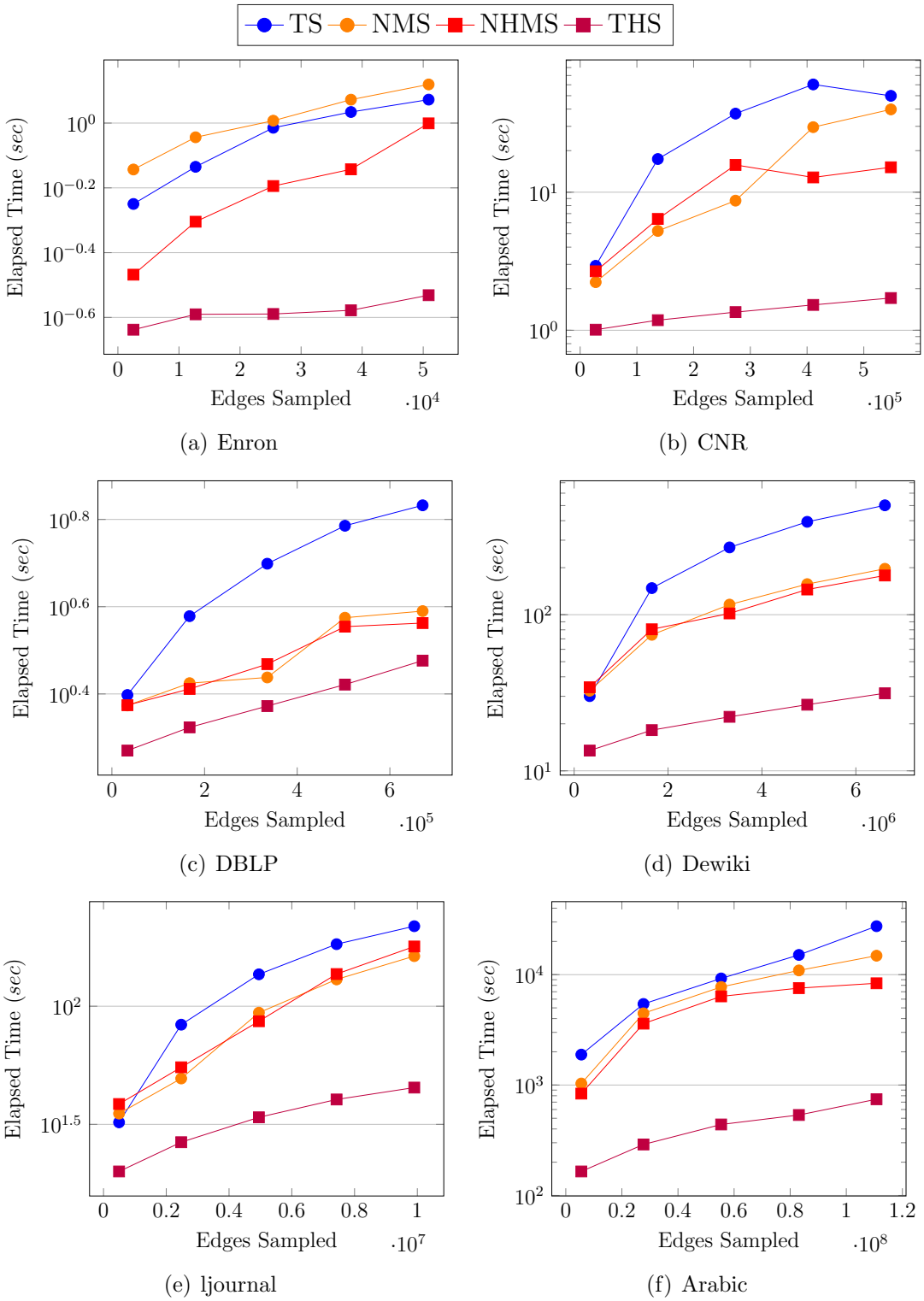


Figure 6.2: Run-time for algorithms ThinkD, NMS, EVMS, NHMS and THS

Graphs	THS	NHMS	NMS	TS
enron	0.29	0.99	1.31	1.18
cnr	1.71	15.15	39.78	49.88
dblp	2.99	3.65	3.88	6.79
dewiki	31.37	178.21	196.79	501.79
ljournal	45.24	178.99	162.97	218.14
arabic	743.21	8352.94	14845.81	27476.80

Table 6.5: Run-time results of various graphs for NHMS, THS, NMS and TS while storing 20% of edges in memory.

6.4.3 Scalability

We planned to measure the scalability of our algorithms and compare them with NMS and TS. The idea of measuring the scalability is to check if the algorithms can handle infinite graph stream and not just medium size graphs. For measuring the scalability scenario, we ran an experiment on Twitter graph that has more than 1.2 billion edges. The data points including the time and the number of edges streamed are collected after every 1 million edges in all the algorithms. The results are plotted in Figure 6.3.

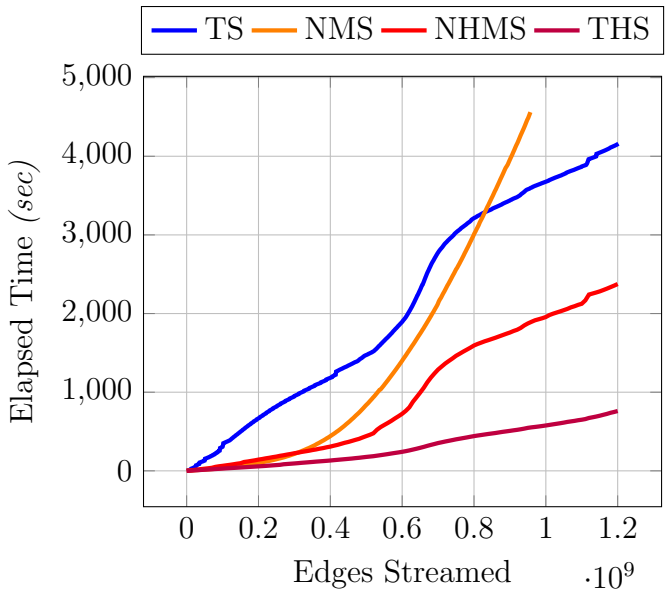


Figure 6.3: Scalability Analysis of algorithms on Twitter Graph

Scalability of FP algorithms: FP algorithms are not scalable due to their characteristics. As the edges stored always grows in FP algorithms, elapsed time grows exponentially as the number of edges increases on massive graphs. We can see the results of NMS have grown exponentially and the algorithm is unable to process the whole graph stream and halted even before a billion edges were processed. Hence, NMS is not scalable.

NHMS can scale NMS: As NHMS is the hybrid variant of NMS, it is important to compare both. NMS does not scale on the large graphs, whereas we see NHMS is able to process the whole graph stream. As we have already discussed, NHMS follows the same trend as FP algorithms until the first k edges. We observe that behavior in Figure 6.3 around 600 million edges, where the curve started to flatten, after a slight exponential rise (similar to NMS).

Scalability of THS and comparison with TS: As seen in Figure 6.3, THS scales linearly. In comparison to TS that takes around 4200 seconds to execute on the Twitter graph, THS just takes 760 seconds and is **5.5X faster**.

Overall, Both hybrid algorithms scale really well for large graphs and are faster in executing the graph stream than both FM and FP algorithms.

Chapter 7

Conclusion

In this thesis, we studied approximation algorithms for triangle estimations using the edge streaming model. We studied existing streaming algorithms and found that they can be categorized into two categories *i.e.* Fixed Memory and Fixed Probability. For brevity, we call Fixed Memory algorithms as FM algorithms and Fixed Probability algorithms as FP algorithms. We also categorized the existing algorithms into these categories. The impediments for both FM and FP categories were found after in-depth analysis. We analyzed that FM algorithms have better scalability than FP algorithms. We also prove that FP algorithms are faster than FM algorithms, due to their characteristics that we discussed in Section 4.

In addition, we explored the idea of utilizing the benefits of both FP and FM approaches, which remain unexplored until now. Therefore, we presented a new category that we call Hybrid Category. The primary focus is to resolve the impediments of both mentioned categories. Hence, the Hybrid algorithms are both scalable and faster than FM or FP algorithms. Also, they have slightly better accuracy over the existing algorithms.

We present two new algorithms for triangle estimations that belong to the Hybrid category. We picked one of the best algorithms in FP category called Neighbourhood Multisampling (NMS) and present a hybrid variant of it called Neighbourhood Hybrid Multisampling, abbreviated as $NHMS$. Considering the FM category, Triest being one of the most popular in the category which is exactly similar to ThinkD (insertion only), we present the hybrid variant of these known as Triest/ThinkD Hybrid Sampling, abbreviated as THS.

The algorithms were implemented in Java 8 and used WebGraph framework that provides a lot better graph compression, which enables us to analyze massive graphs

on a commodity machine, we kept default memory settings for JVM that allowed us to use the maximum of 16GB.[1] The real-world graphs were used for in-depth experimentation of all the algorithms. Both $NHMS$ and THS were successfully able to run on 1.2 billion edges within 16 GB memory.

In our evaluation, we observed that $NHMS$ and THS perform a lot better than their counterparts, *i.e.*NMS and TS, in both running time and accuracy. THS proves to be at least **5 times faster** than TS in all the cases with varied graph sizes and different experimental settings. It is worth noting that TS being an FM algorithm has a limitation of slow running time, whereas, its hybrid variant has boosted its computational time. Similarly, while analyzing $NHMS$, we observed that $NHMS$ successfully executed the Twitter graph with more than 1.2 billion edges, whereas NMS algorithm failed to scale within the limited memory resources. Considering NMS to be an FP algorithm that has an impediment of not scaling on massive graphs, its hybrid variant $NHMS$ could not only **scale up within 16GB memory**, but also is approximately **2 times faster** in most of the cases.

Furthermore, there is a significant difference between the computation times of both THS and $NHMS$ over NMS and TS algorithms. Comparing $NHMS$ with THS, we see that $NHMS$ has better accuracy than THS as it uses two reservoirs of half sizes, and hence, edges have a better chance of getting sampled. On the other hand, THS is faster than $NHMS$ as it just has a single reservoir, and while sampling, there are fewer number of edge replacements.

Considering the ever growing size of the graphs in today's world, we need the algorithms to be both fast and scalable, and the Hybrid algorithms prove to be the best fit to handle massive graphs. Also, this research opens up many new doors which are discussed in the next section of Future Work.

[1]Notably, some other works that have successfully used Webgraph for scaling various algorithms to big graphs are [10, 12, 20, 28, 33, 34].

Chapter 8

Future Work

This research opens up a completely new dimension as we present a new hybrid approach, along with the two new algorithms for estimating triangles in graph streams. There are many open questions for future work, and are detailed below.

We present the two new hybrid algorithms (NHMS and THS) and provide the conceptual reasoning and in-depth experimental analysis that validates them. Future contribution here can be providing theoretical proofs for the accuracy of the estimates of these algorithms. This will provide additional validation of the algorithms.

Both NHMS and THS algorithms currently are limited to the insertion-only streams. Another contribution to this work can be handling dynamic streams using these Hybrid algorithms, where the stream also deletion edges.

Sub-graph structures, other than triangles, such as cliques have proven to have many benefits in identifying clusters. It would be interesting to extend the Hybrid approach and the algorithms proposed to estimate other sub-graph patterns.

Bibliography

[1] Nesreen K Ahmed, Nick Duffield, Theodore L Willke, and Ryan A Rossi. On sampling from massive graph streams. *Proceedings of the VLDB Endowment*, 10(11):1430–1441, 2017.

[2] Ziv Bar-Yossef, Ravi Kumar, and D Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 623–632. Society for Industrial and Applied Mathematics, 2002.

[3] Vladimir Batagelj and Matjaž Zaveršnik. Short cycle connectivity. *Discrete Mathematics*, 307(3-5):310–318, 2007.

[4] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 16–24. ACM, 2008.

[5] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice and Experience*, 34(8):711–726, 2004.

[6] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. ACM Press, 2011.

[7] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[8] Elizabeth Bott and Elizabeth Bott Spillius. *Family and social network: Roles, norms and external relationships in ordinary urban families*. Routledge, 2014.

[9] Luciana S Buriol, Gereon Frahling, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Christian Sohler. Counting triangles in data streams. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 253–262. ACM, 2006.

[10] Shu Chen, Ran Wei, Diana Popova, and Alex Thomo. Efficient computation of importance based communities in web-scale networks using a single machine. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 1553–1562. ACM, 2016.

[11] Jean-Pierre Eckmann and Elisha Moses. Curvature of co-links uncovers hidden thematic layers in the world wide web. *Proceedings of the national academy of sciences*, 99(9):5825–5829, 2002.

[12] Fatemeh Esfahani, Venkatesh Srinivasan, Alex Thomo, and Kui Wu. Efficient computation of probabilistic core decomposition at web-scale. In *Advances in Database Technology-EDBT 2019, 22nd International Conference on Extending Database Technology*, pages 325–336, 2019.

[13] David V Foster, Jacob G Foster, Peter Grassberger, and Maya Paczuski. Clustering drives assortativity and community structure in ensembles of networks. *Physical Review E*, 84(6):066117, 2011.

[14] Mohamed Medhat Gaber, Arkady Zaslavsky, and Shonali Krishnaswamy. Mining data streams: a review. *ACM Sigmod Record*, 34(2):18–26, 2005.

[15] Guyue Han and Harish Sethu. Edge sample and discard: A new algorithm for counting triangles in large dynamic graphs. In *Proceedings of the 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2017*, pages 44–49. ACM, 2017.

[16] Zan Huang. Link prediction based on graph topology: The predictive value of generalized clustering coefficient. *Available at SSRN 1634014*, 2010.

[17] Madhav Jha, Comandur Seshadhri, and Ali Pinar. A space efficient streaming algorithm for triangle counting using the birthday paradox. In *Proceedings of the*

19th ACM SIGKDD international conference on Knowledge discovery and data mining, pages 589–597. ACM, 2013.

[18] Hossein Jowhari and Mohammad Ghodsi. New streaming algorithms for counting triangles in graphs. In *International Computing and Combinatorics Conference*, pages 710–716. Springer, 2005.

[19] Neeraj Kavassery-Parakkat, Kiana Mousavi Hanjani, and A Pavan. Improved triangle counting in graph streams: Power of multi-sampling. In *2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 33–40. IEEE, 2018.

[20] Wissam Khaouid, Marina Barsky, Venkatesh Srinivasan, and Alex Thomo. K-core decomposition of large networks on a single pc. *Proceedings of the VLDB Endowment*, 9(1):13–23, 2015.

[21] Konstantin Kutzkov and Rasmus Pagh. Triangle counting in dynamic graph streams. In *Scandinavian Workshop on Algorithm Theory*, pages 306–318. Springer, 2014.

[22] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600, 2010.

[23] Matthieu Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical Computer Science*, 407(1-3):458–473, 2008.

[24] Yongsub Lim and U Kang. Mascot: Memory-efficient and accurate sampling for counting local triangles in graph streams. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 685–694. ACM, 2015.

[25] MEJ Newman and J. Park. Why social networks are different from other types of networks. *Physical Review E*, 68(3):36122, 2003.

[26] Rasmus Pagh and Charalampos E Tsourakakis. Colorful triangle counting and a mapreduce implementation. *Information Processing Letters*, 112(7):277–281, 2012.

[27] A. Pavan, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. Counting and sampling triangles from a graph stream. *Proc. VLDB Endow.*, 6(14):1870–1881, September 2013.

[28] Diana Popova, Naoto Ohsaka, Ken-ichi Kawarabayashi, and Alex Thomo. Nosingles: a space-efficient algorithm for influence maximization. In *Proceedings of the 30th International Conference on Scientific and Statistical Database Management*, page 18. ACM, 2018.

[29] Yudi Santoso, Venkatesh Srinivasan, Alex Thomo, and Sean Chester. Triad enumeration at trillion-scale using a single commodity machine. In *22nd EDBT*, 2019.

[30] Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *Experimental and Efficient Algorithms, 4th InternationalWorkshop, WEA 2005, Santorini Island, Greece, May 10-13, 2005, Proceedings*, pages 606–609, 2005.

[31] Kijung Shin. Wrs: Waiting room sampling for accurate triangle counting in real graph streams. In *2017 IEEE International Conference on Data Mining (ICDM)*, pages 1087–1092. IEEE, 2017.

[32] Kijung Shin, Jisu Kim, Bryan Hooi, and Christos Faloutsos. Think before you discard: Accurate triangle counting in graph streams with deletions. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 141–157. Springer, 2018.

[33] M. Simpson, V. Srinivasan, and A. Thomo. Clearing contamination in large networks. *IEEE Transactions on Knowledge and Data Engineering*, 28(6):1435–1448, June 2016.

[34] Michael Simpson, Venkatesh Srinivasan, and Alex Thomo. Efficient computation of feedback arc set at web-scale. *Proceedings of the VLDB Endowment*, 10(3):133–144, 2016.

[35] Lorenzo De Stefani, Alessandro Epasto, Matteo Riondato, and Eli Upfal. Triest: Counting local and global triangles in fully dynamic streams with fixed memory size. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 11(4):43, 2017.

[36] 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. IEEE, 2008.

[37] Charalampos E Tsourakakis, Petros Drineas, Eirinaios Michelakis, Ioannis Koutis, and Christos Faloutsos. Spectral counting of triangles via element-wise sparsification and triangle-based link recommendation. *Social Network Analysis and Mining*, 1(2):75–81, 2011.

[38] 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*, pages 837–846, 2009.

[39] Jeffrey S Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

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

[41] K Wolff. Selections from the sociology of georg simmel. 1950.

Appendix A

Performance Results of NHMS

In this section, We provide the full results of the experiments performed for NHMS algorithm. Below is the table that shows the Computational time and the error rate for all the graphs.

Graphs	Edges Sampled	Runtime(sec)	Error rate (%)
enron	2544	0.34	3.93
cnr	27389	2.67	4.77
dblp	33536	2.36	2.07
dewiki	330930	34.27	0.89
ljournal	495142	38.55	0.16
arabic	5540111	836.25	0.15

Table A.1: Performance results of various graphs for NHMS while storing 1% of edges in memory.

Graphs	Edges Sampled	Runtime(sec)	Error rate (%)
enron	12722	0.49	2.19
cnr	136948	6.40	1.49
dblp	167680	2.57	1.66
dewiki	1654651	80.61	0.42
ljournal	2475713	55.15	0.14
arabic	27697591	3598.858	0.11

Table A.2: Performance results of various graphs for NHMS while storing 5% of edges in memory.

Graphs	Edges Sampled	Runtime(sec)	Error rate (%)
enron	25444	0.63	1.23
cnr	273896	15.75	0.80
dblp	335361	2.94	0.77
dewiki	3309302	101.85	0.18
ljournal	4951427	86.40	0.04
arabic	55381721	6355.87	0.05

Table A.3: Performance results of various graphs for NHMS while storing 10% of edges in memory.

Graphs	Edges Sampled	Runtime(sec)	Error rate (%)
enron	38167	0.72	0.83
cnr	410845	12.79	0.86
dblp	503042	3.58	0.54
dewiki	4963954	144.96	0.10
ljournal	7427140	136.83	0.01
arabic	83103987	7552.25	0.03

Table A.4: Performance results of various graphs for NHMS while storing 15% of edges in memory.

Graphs	Edges Sampled	Runtime(sec)	Error rate (%)
enron	50889	0.99	0.71
cnr	547793	15.15	0.82
dblp	670723	3.65	0.16
dewiki	6618605	178.21	0.09
ljournal	9902854	178.99	0.04
arabic	110774765	8352.94	0.03

Table A.5: Performance results of various graphs for NHMS while storing 20% of edges in memory.

Appendix B

Performance Results of THS

In this section, We provide the full results of the experiments performed for THS algorithm. Below is the table that shows the Computational time and the error rate for all the graphs.

Graphs	Edges Sampled	Runtime(sec)	Error rate (%)
enron	2544	0.23	4.41
cnr	27389	1.01	4.46
dblp	33536	1.83	2.64
dewiki	330930	13.47	1.39
ljournal	495142	19.98	0.37
arabic	5540111	164.83	0.37

Table B.1: Performance results of various graphs for THS while storing 1% of edges in memory.

Graphs	Edges Sampled	Runtime(sec)	Error rate (%)
enron	12722	0.25	2.02
cnr	136948	1.18	2.67
dblp	167680	2.10	0.92
dewiki	1654651	18.21	0.24
ljournal	2475713	26.56	0.23
arabic	27697591	289.49	0.11

Table B.2: Performance results of various graphs for THS while storing 5% of edges in memory.

Graphs	Edges Sampled	Runtime(sec)	Error rate (%)
enron	25444	0.25	1.61
cnr	273896	1.35	1.17
dblp	335361	2.35	0.74
dewiki	3309302	22.16	0.20
ljournal	4951427	33.87	0.13
arabic	55381721	439.21	0.05

Table B.3: Performance results of various graphs for THS while storing 10% of edges in memory.

Graphs	Edges Sampled	Runtime(sec)	Error rate (%)
enron	38167	0.26	1.29
cnr	410845	1.52	0.83
dblp	503042	2.63	0.53
dewiki	4963954	26.50	0.11
ljournal	7427140	40.29	0.06
arabic	83103987	535.18	0.04

Table B.4: Performance results of various graphs for THS while storing 15% of edges in memory.

Graphs	Edges Sampled	Runtime(sec)	Error rate (%)
enron	50889	0.29	0.91
cnr	547793	1.71	0.70
dblp	670723	2.99	0.38
dewiki	6618605	31.37	0.11
ljournal	9902854	45.24	0.04
arabic	110774765	743.21	0.04

Table B.5: Performance results of various graphs for THS while storing 20% of edges in memory.