Graphlet counting in massive networks

by

Seyed Vahid Sanei Mehri

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering (Software Systems)

Program of Study Committee:
Goce Trajcevski, Co-major Professor
Srikanta Tirthapura, Co-major Professor
Chinmay Hegde
Ryan Martin
Alexander Stoytchev

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this dissertation. The Graduate College will ensure this dissertation is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2021

Copyright © Seyed Vahid Sanei Mehri, 2021. All rights reserved.

DEDICATION

This dissertation is wholeheartedly dedicated to my beloved parents and sister who provide their emotional support and continuous care.

I would like to express the deepest appreciation to Prof. Trajcevski and Tirthapura who shared their words of encouragement with their invaluable guidance and support throughout my doctoral study.

ACKNOWLEDGEMENTS

I would like to thank the Department of Electrical and Computer Engineering of Iowa State University and my colleagues in our research group for providing a great research opportunity. I am grateful to the support of the National Science Foundation under Grant IIS-1527541 and CCF-1725702.

TABLE OF CONTENTS

DEDICATION . . . ii

ACKNOWLEDGEMENTS . . . iii

LIST OF TABLES . . . vi

LIST OF FIGURES . . . vii

ABSTRACT . . . ix

CHAPTER 1. OVERVIEW . . . 1
- 1.1 Introduction . . . 1
- 1.2 Preliminaries . . . 5
- 1.3 Contribution . . . 8
  - 1.3.1 Triangle Counting in Static Networks . . . 8
  - 1.3.2 Butterfly Counting in Static and Streaming Networks . . . 9

CHAPTER 2. EXISTING WORK AND RESEARCH PROBLEM . . . 13
- 2.1 Triangles . . . 13
  - 2.1.1 Exact methods on single-core machines . . . 13
  - 2.1.2 Exact Parallel Approaches . . . 14
  - 2.1.3 Approximation Approaches . . . 18
- 2.2 Bipartite Graphlets . . . 20
- 2.3 Butterflies . . . 21
- 2.4 Other Graphlets in Graphs . . . 21
- 2.5 Research Problem . . . 22

CHAPTER 3. TRIANGLE COUNTING IN STATIC NETWORKS . . . 23
- 3.1 One-shot Sparsification Algorithms . . . 23
  - 3.1.1 Doulion Sparsification . . . 23
  - 3.1.2 Colorful Sparsification . . . 25
  - 3.1.3 Improved Edge Sparsification . . . 26
  - 3.1.4 Approximation by Local Sampling . . . 27
  - 3.1.5 Vanilla Wedge Sampling . . . 28
  - 3.1.6 Improved Wedge Sampling . . . 30
- 3.2 Experiments on Static Algorithms . . . 31

CHAPTER 4. BUTTERFLY COUNTING IN STATIC BIPARTITE NETWORKS 37
4.1 Approximation by Local Sampling 37
4.1.1 Vertex Sampling (Algorithm VSAMP) 38
4.1.2 Edge Sampling (Algorithm ESAMP) 41
4.1.3 Accuracy and Runtime of Sampling 45
4.1.4 Faster Edge Sampling using FAST-EBFC 47
4.1.5 Comparing Sampling-based Approaches 49
4.2 Approximation by One-shot Sparsification 49
4.2.1 Edge Sparsification (Algorithm ESPAR) 50
4.2.2 Colorful Sparsification (Algorithm CLRSPAR) 52
4.2.3 Comparison of Sparsification Algorithms 54
4.2.4 Sampling or Sparsification? 55

CHAPTER 5. BUTTERFLY COUNTING IN BIPARTITE NETWORK STREAMS 57
5.1 Algorithms for Insertion-only Streams 57
5.1.1 Adaptive Sampling: FLEET1 57
5.1.2 Concentration analysis of Y^t_{FLEET1} 59
5.1.3 Improved Adaptive: FLEET2 and FLEET3 61
5.2 Sliding Window Streaming 63
5.2.1 Sequence-based Window (Algorithm FLEETSSW) 63
5.2.2 Time-based Sliding Window (Algorithm FLEETTSW) 65
5.3 Algorithms for Fully-dynamic streams 66
5.3.1 Algorithm THINKD-ACC 67
5.3.2 Algorithm FLEET-FD 70
5.4 Experimental Evaluation 71
5.4.1 Accuracy 72
5.4.2 Runtime and Throughput 74
5.4.3 Impact of γ on Runtime and Accuracy 75
5.4.4 Sliding Window 75

CHAPTER 6. CONCLUSION AND FUTURE WORK 76

BIBLIOGRAPHY 78

LIST OF TABLES

Table 1.1 Notations 7

Table 3.1 Properties of static graphs. n, m, W, W^*, κ, and Δ denote the number of vertices, edges, wedges, ordered wedges, number of pairs of triangles with a common edge, and number of triangles respectively.[1] 30

Table 3.2 Variances of DoulionSpar, ClrSpar, Impr-ESpar, for $p = 0.1$. 32

Table 3.3 Standard deviations of estimators on several graphs. For each method, the standard deviation is shown in the left column, and the Err, the ratio between the (theoretical) standard deviation and the number of triangles (i.e., Δ), is shown in the right column. 36

Table 4.1 Standard deviations of estimators on large graphs. For each method, the theoretical upper bound on the standard deviation is shown in the left column, and the “error“, the ratio between the (theoretical) standard deviation and the number of butterflies, is shown in the second column. 45

Table 4.2 Upper bounds on ESpar & ClrSpar variances, p=0.1. 54

Table 4.3 Time (in seconds) to obtain 1% relative percent error for the best sampling and sparsification algorithms. 55

Table 5.1 Properties of the bipartite graphs. $|E|$ is the number of edges, ξ the total number of butterflies, and the butterfly density is the ratio $\xi/|E|^4$. $|L|$ and $|R|$ are the number of vertices in the left and right partions, respectively. . . 73

LIST OF FIGURES

Figure 1.1 A triangle in graph $\mathcal{G}$. 5

Figure 1.2 Four butterflies in graph G. The number of per-vertex and per-edge butterflies are shown for some vertices/edges. 8

Figure 3.1 Sampling procedure in ClrSpar for C = 2. Edges with monochromatic endpoints are sampled and construct $\mathcal{G}'$. 25

Figure 3.3 Accuracy of one-shot algorithms for different sampling probabilities. Impr-ESpar performs better than both DoulionSpar and ClrSpar, yielding $< 1\%$ relative error with sampling probability 0.008 for all graphs. 33

Figure 3.5 Time performance of one-shot algorithms for different sampling probabilities. Impr-ESpar is faster than DoulionSpar and ClrSpar with a speedup factor of 17 in graph `Twitter`. 34

Figure 3.7 Accuracy of local sampling algorithms, running in 3.000×10^3 seconds. WSamp performs better than Impr-WSamp, due to the high preprocessing step of Impr-WSamp for vertex ranking, shown by the blue box. WSamp yields $< 1\%$ relative error within at most 9 seconds for all graphs. Impr-WSamp requires 8, 82, 252, 594, 2516 seconds, including the preprocessing step, to achieve $< 1\%$ relative error in graphs `DBLP`, `Journal`, `Orkut`, `Wiki`, and `Twitter` respectively. 35

Figure 4.2 A pair of butterflies in $\mathcal{G}$ can be of one of the above five types. 38

Figure 4.4 Relative error as a function of runtime, for sampling algorithms. ESamp with Fast-eBFC yields $< 1\%$ relative error within 5 seconds for all networks. 46

Figure 4.5 Average time per iteration of sampling. For ESamp with Fast-eBFC, the time is shown for 1000 iterations of Fast-eBFC (Algorithm 9). 47

Figure 4.7 Accuracy (on left y-axis) and runtime performance (on right y-axis) of sparsification algorithms for different probabilities (on x-axis). ESpar performs better than ClrSpar, and yields $< 1\%$ relative error within 4 seconds for all networks. 50

Figure 5.2 Number of butterflies as a function of stream size. 71

Figure 5.4 Accuracy of FLEET1, FLEET2, and FLEET3 for $\gamma = 0.5$ versus reservoir size. Bottom x-axis shows the reservoir size and top x-axis shows the sample rate, defined as the ratio of the reservoir size to the stream size. 72

Figure 5.6 Accuracy of FLEET1, FLEET2, and FLEET3 at different points in the stream, reservoir size is $3.00 \times 10^2 K$ and $\gamma = 0.9$. 72

Figure 5.8 Throughput of FLEET1, FLEET2, and FLEET3 algorithms as a function of reservoir size where $\gamma = 0.6$. 73

Figure 5.10 Accuracy and runtime of FLEET1, FLEET2, and FLEET3 as a function of γ where $\mathrm{M} = 600K$. 73

Figure 5.12 Relative error vs. number of edges received for FLEETSSW. Window size $= 5 \times 10^6$ edges, $\gamma = 0.9$. 74

ABSTRACT

Graphs are a standard tool for deriving a flexible abstraction of interactions among entities and are extensively employed in a myriad variety of domains from various disciplines, including bioinformatics, biochemistry, social sciences, and neurobiology. These, and many more ubiquitous domains, can be modeled as graphs (also called networks), which capture interactions (i.e., edges) and discrete entities (i.e., vertices). Understanding the underlying structure of complex networks is a typical data-mining task. Widespread research works have shown that the small subgraphs (also known as motifs or graphlets) frequency distribution is an effective tool to analyze complex networks. Graph motifs such as triangle (a cycle of size three), diamond (two triangles with one vertex in common), butterfly (a (2,2)-biclique), and k-clique (for $k \leq 6$) are indeed regarded as building blocks to understand the true structure of an underlying network. Indeed, graph motifs resemble cohesion and furnish analytic insights into the heart of real-world networks.

Due to the surge of data, further fueled by inevitable combinatorial explosion, counting motifs turns out to be a challenging task in large-scale networks. A possible approach to cope with the ever-increasing cost of counting in large graphs is to approximate the number of motifs through a sampling mechanism. The mainstream existing approximation approaches provide an estimation of motif count with a considerable reduction in runtime, tailoring these methods to accomplish counting in planetary-scale networks. The demand for approximate approaches is further nurtured by its numerous real-world applications, which do not require an exact count of motifs while an accurate but faster approximation will suffice. Consequently, several approximation approaches have been designed to estimate triangle and butterfly count, many in static graphs, and far fewer in streaming network, which the network is dynamically changed upon receiving edge insertions/deletions. The main focus of this dissertation is approximate approaches in static and streaming networks for graph motif counting, more specifically triangle and butterfly

counting. We discussed several randomized algorithms and presented the empirical evaluations for several algorithms. We compared the runtime and accuracy of existing work for triangle counting in static and streaming networks in great detail. Further, we described our approximation algorithms for butterfly counting. The experiments for butterfly counting shows that our algorithms outperform the existing methods in terms of runtime and accuracy.

CHAPTER 1. OVERVIEW

1.1 Introduction

Graphs are a standard tool for deriving a flexible abstraction of interactions among entities and are extensively employed in a myriad variety of domains from various disciplines, including bioinformatics, biochemistry, social sciences, and neurobiology. These, and many more ubiquitous domains, can be modeled as graphs (also called networks) which capture interactions (i.e., edges) and discrete entities (i.e., vertices). Understanding the underlying structure of complex networks is a typical data-mining task. Widespread research works such as [16, 21, 73] have shown that the small subgraphs (also known as motifs or graphlets) frequency distribution is an effective tool to analyze complex networks. Graphlets such as triangle (a cycle of size three) [7], diamond [13], butterfly [76, 102], (α, β)-core biclique [57], 4-vertex subgraphs [43], and k-clique [42] are indeed regarded as building blocks which resemble cohesion and furnish analytic insights into the heart of real-world networks. For this reason, finding and counting graphlets are among the most important and widely-used network analysis procedures for providing significant statistics to characterize the local properties of underlying networks. In the following, we list two real-world applications of graphlet counting.

- **Graphlet counting for aerial image category recognition:** Aerial image category recognition is an essential aspect of image recognition, with numerous applications, such as scene annotation, urban studies, landscape studies for drought monitoring, video surveillance, and robotics path planning [107], to identify scene components in aerial images. For example, an accurate scene annotation system should be able to distinguish a parking lot in an aerial image from other, similar areas, such as the landing area of an airport. The high number of components and complex spatial interactions among the components make the category detection a prohibitively time-consuming task. An aerial

image can be viewed as a set of components linked by geospatial interactions. An aerial image comprises millions of pixels that are interpreted as local features. The high number of pixels makes the computational complexity for category recognition intractable. The components of an aerial image, along with their interactions, can be interpreted as a large network. Therefore, one can apply graph mining algorithms to discover the categories of components. As shown in [107], graphlets are extracted from the underlying network after converting an aerial image to a network. The graphlets reflect the geometric property and color/texture distribution of components of an aerial image. The similarity between two components (e.g., roads, parking lots, houses) can be measured by comparing the graphlets count, thereby leading to the entity recognition in an aerial image with millions of pixels.

- **Graphlet counting in online advertising and recommendation systems:** Graphlet counting in bipartite networks is highly applied in advertising and online recommendations. Online shopping networks such as Amazon resemble bipartite networks, where customers form one partition, and products form the other. In these networks, a customer and an item are two types of vertices and interact with each other (i.e., they are connected by an edge) if the customer purchases the item. The ranking functions that are fundamentally based on the number of graphlets, such as butterflies, could be used to enhance recommendation systems for purposes of suggesting the right goods to customers. Indeed, the ranking functions use graphlet counts to show the similarity between customers' interests. Therefore, their purchases can be recommended to their peers. A reliable recommendation system enhances the quality of the recommendation and expands economic activities and e-commerce transactions in an online shopping platform. This promotes the economic growth of businesses that sell their products online through the host merchandising networks.

Due to the surge of data, further fueled by inevitable combinatorial explosion, counting graphlets turns out to be a challenging task in large-scale networks. A practical exact algorithm [80] for counting triangles, as the smallest nontrivial graphlet in a unipartite network, can handle only

medium-size graphs with millions of edges and take time complexity $\mathcal{O}(m^{3/2})$, where m is the number of edges. It stands to reason that sequential exact methods cannot truly cope with web-scale networks with billions of edges. In recent years, researchers have designed an assortment of parallel approaches with varying capabilities, which generally process multiple computational tasks concurrently, to improve the performance of the counting process. Believe it or not, the deployment of a massive parallelism in exact methods fails to handle real-life networks, growing exponentially in scale. In addition, parallel computing on graphs suffers from a range of inherent challenges, including irregular graph structures, unbalanced computational sub-tasks, poor data locality, high latency connections, and memory access overhead; thereby, highly challenging for large-scale graph processing. [58] discusses the foregoing pitfalls with more details. One possibility to cope with the ever-increasing cost of counting in large graphs is to approximate the number of graphlets through a sampling mechanism. The mainstream existing approximation approaches provide an estimation of graphlet count with a considerable reduction in runtime, tailoring these methods to accomplish counting in planetary-scale networks. The demand for approximate approaches is further nurtured by its numerous real-world applications, which do not require an exact count of graphlets while an accurate estimation using a faster estimator will suffice. Consequently, several approximation approaches have been designed to estimate triangle and butterfly count, many in static graphs, and far fewer in streaming networks, i.e., the network that are dynamically changed upon receiving edge insertions/deletions. In this vein, we mainly focused on counting two basic, yet non-trivial graphlets: triangles and butterflies.
Triangle (a cycle of size 3 or a 3-clique) is formed by three vertices which are fully connect. Triangle is considered the most basic yet non-trivial graphlet in unipartite networks which is formed by . The number of triangles in a network is used to measure important triadic metrics such as transitivity degree and clustering coefficient. The quantity is also targeted for discovering thematic structures in the Web [29] and has application in spam detection, graph classification [100], and community detection. We described existing work using parallel exact and approximate

methods, randomized algorithms, and algorithms for streaming networks. We conducted extensive empirical evaluation to compare the most efficient triangle estimators in static network.

For butterfly counting, we consider *bipartite (affiliation) networks*, an important type of network for many applications [19, 54]. For example, relationships between authors and papers can be modeled as a bipartite graph, where authors form one vertex partition, papers form the other vertex partition, and an author has an edge to each paper that she published. Other examples include user-product relations, word-document affiliations, and actor-movie networks. Bipartite graphs can represent hypergraphs that capture many-to-many relations among entities. A hypergraph $H = (\mathcal{V}_H, \mathcal{E}_H)$ with vertex set $\mathcal{V}_H$ and edge set $\mathcal{E}_H$, where each hyper-edge $h \in \mathcal{E}_H$ is a set of vertices, can be represented as a bipartite graph with vertex set $\mathcal{V}_H \cup \mathcal{E}_H$ with one partition for $\mathcal{V}_H$ and another for $\mathcal{E}_H$, and an edge from a vertex $v \in \mathcal{V}_H$ to an edge $h \in \mathcal{E}_H$ if v is a part of h in H. For example, a hypergraph corresponding to an author-paper relation where each paper is associated with a set of authors can be represented using a bipartite graph with one partition for authors and one for papers.

A common approach to handle a bipartite network is to reduce it to a unipartite co-occurrence network by a projection [62, 63]. A projection selects a vertex partition as the set of entities and creates a unipartite network whose vertex set is the set of all entities, and two entities are connected if they share an affiliation in the bipartite network. In the author-paper network, a projection on the authors creates a unipartite co-authorship network. However, such a projection causes the number of edges in the graph to explode, artificially boosts the number of triangles and clustering coefficients and results in information loss. For instance, we observed up to 4 orders of magnitude increase in size when the bipartite network between wikipedia articles and their editors in French is projected onto a unipartite network of articles – the number of edges goes from 22M to more than 200B. As a result, it is preferable to analyze bipartite networks directly.

While there is extensive work on graphlet counting in unipartite networks, these do not apply to bipartite networks. Graphlets in bipartite networks are very different from graphlets in a unipartite network. The most commonly studied graphlets in a unipartite network are cliques of

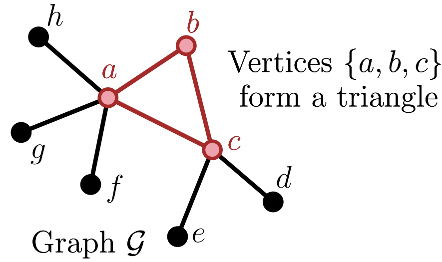


Figure 1.1: A triangle in graph $\mathcal{G}$.

small sizes, but a bipartite graph does not have any cliques with more than two vertices, not even a triangle! Instead, natural graphlets in a bipartite network are bicliques of small size. The most basic graphlet that models cohesion in a bipartite network is a complete 2×2 biclique, also known as a butterfly [6, 78] or a rectangle [101]. Although there have been attempts at defining other cohesive graphlets in bipartite networks, such as the complete 3×3 biclique [19] and 4-path [64], the butterfly remains the smallest unit of cohesion and has been used in defining basic metrics such as the clustering coefficient in a bipartite graph [56, 75]. In particular, it is the smallest subgraph that has multiple vertices on each side with multiple common neighbors. It can be considered to play the same role in bipartite networks as the triangle did in unipartite networks – a building block for community structure [6]. Notice that the rectangle counting algorithms for the unipartite graphs are not suitable for the butterfly counting in the bipartite graphs because algorithms for unipartite graphs assume edges between any two vertices in unipartite graphs which is not true in the case of bipartite graph. We view butterfly counting as a first, but important step towards general methods for graphlet counting and analysis of bipartite affiliation networks. We described and extensively evaluated static and streaming butterfly counting methods which can accurately estimate the number of butterflies in a given large bipartite network.

1.2 Preliminaries

First, we introduce the notations for both static and streaming methods for triangle counting. In static networks, we consider simple, unweighted, and undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, with $n = |\mathcal{V}|$

vertices labeled $1, 2, \cdots, n$ and $m = |\mathcal{E}|$ edges. For every vertex $v \in \mathcal{V}$, $\Gamma_v = \{u | (v, u) \in \mathcal{E}\}$ denotes the set of vertices adjacent to v (i.e., neighbors of v), and $d_v = |\Gamma_v|$ denotes the degree of v. $d_{\max}$ denotes the maximum degree of a vertex in the graph. In addition, when we refer to a wedge, we mean a path of length two, and W shows the number of wedges. A triangle is a ternary relationship among three distinct vertices such that each possible edge among them exists in the graph (see Figure 1.1). Triangle is indeed a 3-clique or a 3-cycle (a cycle of size three). Figure 1.1 shows an example of a triangle in a graph. We define $\mathcal{T}(\mathcal{G}) = \{\delta^{(1)}, \delta^{(2)}, \cdots, \delta^{(\Delta)}\}$ as the set of all triangles in $\mathcal{G}$, where $\delta^{(i)}$ is the ith triangle (the order of triangles can be arbitrary) and $\Delta(\mathcal{G})$ as the number of triangles in $\mathcal{G}$, i.e., $\Delta(\mathcal{G}) = |\mathcal{T}(\mathcal{G})|$. For simplicity, when it is clear from context, we suppress notation $\mathcal{G}$. In addition, given an edge $e \in \mathcal{E}$, $\Delta(e)$ denotes the number of triangles in $\mathcal{G}$ which edge e is a part of. Similarly, $\forall v \in \mathcal{V}, \Delta(v)$ indicates the number of triangles which vertex v is a part of. We define κ as the number of *unordered* pairs of distinct triangles that share an edge. The existing static algorithms aim to compute $\Delta(\mathcal{G})$.

As a static bipartite graph, we consider simple, unweighted, bipartite graphs, where there are no self-loops or multiple edges between vertices. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a simple bipartite graph with $n = |\mathcal{V}|$ vertices and $m = |\mathcal{E}|$ edges. Vertex set $\mathcal{V}$ is partitioned into two sets L and R such that $\mathcal{V} = L \cup R$ and $L \cap R = \emptyset$. The edge set $\mathcal{E} \subseteq L \times R$. In addition, $\Gamma_v^2 = \{w | (w, u) \in \mathcal{E} \wedge w \neq v, \ \forall\ u \in \Gamma_v\}$ is the set of vertices that are *exactly* in distance 2 from v, i.e., neighbors of the neighbors of v (excluding v itself). A biclique is a complete bipartite subgraph, and is parameterized by the number of vertices in each partition; for integers α, β, an $\alpha \times \beta$ biclique in a bipartite graph is a complete subgraph with α vertices in L and β vertices in R. A butterfly in $\mathcal{G}$ is a 2×2 biclique and consists of four vertices $\{a, b, x, y\} \subset \mathcal{V}$ where $a, b \in L$ and $x, y \in R$ such that edges (a, x), (a, y), (b, x), and (b, y) all exist in $\mathcal{E}$. Let $\bowtie(\mathcal{G})$ denote the number of butterflies in $\mathcal{G}$ (we use notation $\bowtie$ when $\mathcal{G}$ is clear from the context). For vertex $v \in \mathcal{V}$, let $\bowtie_v$ denote the number of butterflies that contain v. Similarly, for edge $e \in \mathcal{E}$, let $\bowtie_e$ denote the number of butterflies containing e (See Figure 1.2). We aim to compute $\bowtie(\mathcal{G})$ for a static graph $\mathcal{G}$.

Table 1.1: Notations

Notation	Description
$\mathcal{G} = (\mathcal{V}, \mathcal{E})$	Simple graph with vertices $\mathcal{V}$ and edges $\mathcal{E}$.
$\mathcal{V} = L \cup R$	In bipartite networks, $\mathcal{V}$ is partitioned into L and R s.t. $L \cap R = \emptyset$.
Γ_v	Set of vertices adjacent to v, i.e., $\{u \mid (v,u) \in E\}$.
d_v	Degree of vertex v, i.e., $\lvert\Gamma_v\rvert$.
n, m, W	Number of vertices ($\lvert\mathcal{V}\rvert$), edges ($\lvert\mathcal{E}\rvert$), and wedges ($\sum_{v \in \mathcal{V}} \binom{v}{2}$).
$d_{\max}$	Maximum degree of a vertex in the graph.
Γ_v^2	Distance-2 neighbors of v (excluding itself).
$\bowtie(\mathcal{G})$ (or $\bowtie$)	Number of butterflies in graph $\mathcal{G}$.
$\bowtie_{v(e)}$	Number of butterflies that contain vertex v (edge e).
$\Delta(\mathcal{G})$	Number of triangles in $\mathcal{G}$.
$\Delta(v)$	Local number of triangles which vertex v is a part of.
$\Delta(e)$	Local number of triangles which edge e is a part of.
κ	Number of (unordered) pairs of triangles in $\mathcal{G}$ that share an edge.
$\mathcal{S}$	A sequence or a stream of edges.
$\mathcal{R}$	Reservoir to keep a summary of the stream.
M	Size of reservoir, i.e., $\lvert\mathcal{R}\rvert$.

A **bipartite graph stream** is a sequence of edges $\mathcal{S} = \langle e_1, \bullet \rangle, \langle e_2, \bullet \rangle, \cdots$ where e_i is the ith edge and $\bullet \in \{+, -\}$, which indicates whether an edge is inserted or deleted from the stream. For $t > 0$, let $\mathcal{S}^t$ denote the first t edges of the stream and let $\mathcal{G}^t = (\mathcal{V}^t, \mathcal{E}^t)$ denote the graph formed by the first t edges. Let $\bowtie^t(\mathcal{G})$ denote the number of all butterflies in $\mathcal{G}^t$. In insertion-only streams, we suppress $\bullet$ because edges are always inserted to the stream. Note that, we are given a reservoir $\mathcal{R}$ to keep the summary of the graph which is used to estimate the number of butterflies at any time t.

Infinite Window: For any $t > 0$ and a stream $\mathcal{S}$, the goal is to continuously maintain (an estimate of) $\bowtie^t$, the number of butterflies in the graph $\mathcal{G}^t$, as t increases.

Sliding Window: For a window size parameter W, a sequence based sliding window is defined as the set of W most recent edges. For $t \geq W$, when edge e_t is observed, the sliding window consists of edges $e_t - W + 1, e_t - W + 2, \cdots, e_t$. For $t < W$, the window consists of the entire stream so far. The goal is to continuously maintain an estimate of the number of butterflies in the graph defined by the sliding window. We also consider time-based sliding window, a

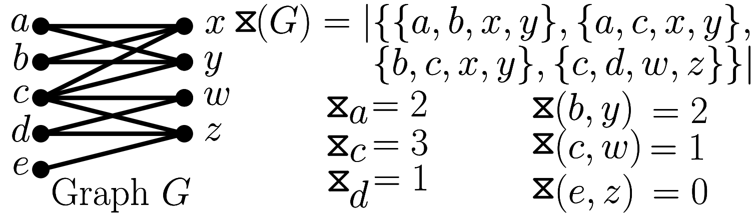


Figure 1.2: Four butterflies in graph G. The number of per-vertex and per-edge butterflies are shown for some vertices/edges.

generalization of sequence-based window, defined as the set of edges whose timestamps are the most recent. In a time-based window, the window size does not correspond to a specific number of edges, but instead to a range of timestamps.

We summarize the notations in Table 1.1. When estimating the number of triangles and butterflies, we look for provable guarantees on the estimates computed using the following notion of randomized approximation. For parameters $\epsilon, \delta \in [0, 1]$, an (ϵ, δ) − approximation of a number Z is a random variable $\hat{Z}$ such that $\Pr[|\hat{Z} - Z| > \epsilon Z] \leq \delta$.

1.3 Contribution

1.3.1 Triangle Counting in Static Networks

One possibility to cope with the ever-increasing cost of counting in large graphs is to approximate the number of triangles through a sampling mechanism. The mainstream existing approximation approaches provide an estimation of triangle count with a considerable reduction in runtime, tailoring these methods to accomplish counting in planetary-scale networks. The demand for approximate approaches is additionally nurtured by numerous real-world applications, which do not require an exact count of triangles while an accurate estimation using a faster estimator will suffice. In return, several approximation approaches have been designed to estimate triangle count in static networks. Here, we list our contributions for triangle counting in static networks.

- **Theoretical comparison:** We discussed the accuracy, runtime, and space utilization of the most efficient estimator in detail. We computed variance and showed the unbiased expectations of estimators. Static estimators are divided into two subsections based on their

sampling approaches: 1) One-shot 2) Local sampling approximation. In Table 3.2, we provided the upper bounds on the variance of static estimators using one-shot approaches.

- **Framework:** We implemented several existing methods in a framework, which includes the most efficient triangle counting methods.[1] We provided a user-friendly interface that shows the progress of executions and will output the runtime and estimates of triangle count along with other useful statistics. Researchers will be able to run their experiments on different networks, and the size of output files (i.e., the number of checkpoints for recording estimates, runtime, etc.) can be determined beforehand. Also, it is possible to add a new method to the existing library with minimal effort. The library is publicly available, which makes it easy for further contribution from the research community.

- **Extensive experiments:** We presented experimental results of existing triangle counting methods in static networks. The runtime, the accuracy, and the throughput of estimators are depicted in experimental sections. Based on the results, we carried out a thorough comparison between the performance of estimators. We used real-world networks with billions of edges and hundreds of trillions of triangles. In addition, we discussed and proposed the state-of-the-art in various scenarios based on our experimental evaluations backed up with theoretical guarantees.

1.3.2 Butterfly Counting in Static and Streaming Networks

We present efficient algorithms to accurately estimate the number of butterflies in a bipartite static and streaming networks. Our algorithms are simple to implement, backed up by theoretical guarantees, and have good practical performance. We introduce randomized algorithms to find the approximate number of butterflies in a static network by sampling. Our static algorithms are able to derive accurate estimates with error as low as 1% within a few seconds, are much faster than the exact algorithms, and have an insignificant memory print. Further ahead, we present highly accurate, memory efficient algorithms for approximating butterflies in an insertion-only

[1] https://github.com/beginner1010/Triangle-Counting

graph stream. The streaming algorithms use a fixed size memory which is much smaller than the size of entire stream and continuously maintains an estimation of butterfly count upon arriving edges. In this section, we included the randomized algorithms in static networks from our prior work [76] and also our streaming algorithms of [77] for insertion-only streams. The implementation of algorithms for butterfly counting in static and streaming bipartite networks are publicly available.[2]

- **Algorithms for static bipartite networks:** In static networks, we assume that the entire network can be stored in memory. The goal here is to approximate the global butterfly count of the input graph. We present two types of randomized algorithms:
  - **One-shot sparsification:** This approach assumes that the entire graph is available for processing. it thins down the graph to a much smaller sparsified graph through choosing each edge of the original graph with a certain randomized procedure. Exact butterfly counting is then applied on the sparsified graph to estimate the number of butterflies in the original graph. We present two such algorithms – ESPAR and CLRSPAR.
  - **Local sampling:** These methods, on the other hand, can work under limited access to the input graph. They randomly sample small subgraphs *local* to an element of the graph and use them to compute an estimate. This is in contrast to sparsification, which needs a global view of the graph. We investigate sampling of subgraphs localized around a vertex (VSAMP), edge (ESAMP), and a wedge (WSAMP). Sampling algorithms are especially useful when there is a rate-limited API that provides random samples, such as the GNIP framework for Twitter [1] and the Graph API of Facebook [2]. *In the rest of this report, when we say "***sampling algorithms***", we mean local sampling algorithms.*
  - **Experiments on real-world static networks:** We present results of an evaluation of our algorithms on large real-world networks. These results show that the algorithms

[2] https://github.com/beginner1010/ButterflyCounting

can handle massive graphs with hundreds of millions of edges and trillions of butterflies. Our most efficient sampling algorithm, which we call ESAMP +FAST-EBFC, gives estimates with a relative error less than 1 percent within 5 seconds, even for large graphs with trillions of butterflies. We observed a similar behavior with our best one-shot sparsification algorithm, ESPAR, which typically yields estimates with error less than 1 percent, within 4 secs.

- **Algorithms for graph streams:** In a streaming setting, edges arrive over time as a sequence of updates. Mainly, there are two scenarios: 1) insertion-only streams where edges are only inserted to the graph 2) fully dynamic setting where the stream is in the presence of both edge deletion and insertion. Here, we present accurate, memory efficient algorithms for insertion-only streams. In addition, we present sliding window algorithms in network streams to estimate the number of butterflies in time- and sequence-based window of the stream.

  - **Infinite window insertion-only:** We present small-memory algorithms for estimating the number of butterflies within an infinite window consisting of all edges seen so far in the stream. The memory used by our algorithms is no more than a given parameter M. We first present an algorithm FLEET1 that is based on adaptive random sampling from the edge stream so that as more edges are seen, the sampling probability decreases so as to fit within available memory. We prove that the estimator returned by FLEET1 is unbiased and derive concentration bounds showing that the estimator is close to the actual value with a high probability if the memory used is large enough. We present two enhancements to FLEET1, leading to algorithms FLEET2 and FLEET3, which provide better memory-accuracy trade offs in practice.
  - **Infinite window fully-dynamic:** In a more general and challenging scenario, edges can be deleted from the input stream. While the mainstream algorithms are designed for graphlet counting in insertion-only streams, there are only few algorithms to handle

fully-dynamic streams. We present memory-efficient algorithms to count butterflies in fully-dynamic streams. We also showed the expectation return values of the algorithms are unbiased.

– **Sliding window insertion-only streaming:** In stream data mining, the scope of aggregation often needs to be restricted to include edges that have arrived within a recent window. To handle such cases, we present extensions of FLEET to the sliding window model [12, 20, 32]. We consider two types of sliding windows: 1) For a sequence-based window, defined as the set of W most recent edges in the stream for a window size parameter w, we present an algorithm FLEETSSW. 2) For a time-based window, defined as stream elements whose timestamps are greater than $(c - w)$, where c is the current time and w is the window size, we present an algorithm FLEETTSW. Both algorithms use a bounded memory that does not increase with the number of edges in the window. Our algorithm for a time-based window is flexible to receive the window size as a parameter during the query and does not need to know the window size in advance.

– **Experiments on real-world graph streams:** We experimentally evaluate FLEET on real-world graph streams. Results show that our algorithms are effectively able to handle large graph streams. For instance, on network `Bag-pubmed` (see Table 5.1) with approximately 500M edges and 40T butterflies, our algorithms achieve estimates with an error of less than 1% using a memory of 600K edges. Our methods present different trade-offs between memory, estimation accuracy, and runtime, making them applicable in real-world applications with different requirements and significantly outperform prior works on subgraph counting from graph streams.

CHAPTER 2. EXISTING WORK AND RESEARCH PROBLEM

In this section, we introduce and motivate the graphlet counting problem in massive networks. The literature is rich in triangle counting methods in various settings, such as exact methods, parallel methods, randomized methods in static networks, and streaming algorithms. However, bipartite networks are triangle free. We developed several methods for butterfly counting. Note that butterflies are the smallest non-trivial motif in bipartite networks. In the following, we introduce the existing work for graphlet counting including triangles and butterflies.

2.1 Triangles

The literature is rich for computing triangle count, including matrix-based, parallel, and randomized approaches. Several works for *enumerating* the triangles of a network have been designed. The I/O-efficient algorithms include the work of Chiba and Nishizeki [23], which enumerate triangles with time complexity $\mathcal{O}\left(m\alpha\left(\mathcal{G}\right)\right)$, where m is the number of edges and $\alpha\left(.\right)$ is the arboricity of the graph.[1] Shank and Wagner [80] developed an exact $\mathcal{O}\big(nc_{\max}^2\big)$ algorithm, where n is the number of vertices and $c_{\max}$ is maximum core number of vertices in the graph. The other enumeration methods include in-memory methods [53, 79, 84], in the external-memory model [26, 61], and in parallel/distributed settings [33, 68, 69, 91]. In the following we list prior exact triangle counting methods using single-core machines, parallel methods, and approximation methods, including main-memory, streaming, and sub-linear algorithms.

2.1.1 Exact methods on single-core machines

Coppersmith and Winograd in 1990 [24] designed an exact algorithm using matrix multiplication with $\mathcal{O}\left(n^{1.41}\right)$ time complexity. This method is still recognized as the fastest exact algorithm.

[1] Arboricity number of a graph is defined as the minimum number of spanning forests to cover all edges of the graph.

However, it requires $\mathcal{O}\left(n^2\right)$ space; hence, poorly scales in practice when a network contains millions of vertices. Prior to this work, Itai and Rodeh [41] presented an exact algorithm with time complexity $\mathcal{O}\left(m^{1.5}\right)$. While this method has a higher asymptotic time bound than of [24], it is efficient in space utilization and better scales in real-world networks, given that a high degree of sparsity is a commonplace property of mainstream real-world networks.

2.1.2 Exact Parallel Approaches

One practical solution for computing triangle counting faster than exact methods is leveraging parallel/distributed platforms. These platforms enable fast computation in massive networks with hundreds of trillions of triangles. Parallel shared-memory methods with multi core machines, shared-nothing platforms such as MapReduce, and Graphical Processing Units (GPUs)-based methods are arguably the three most prominent approaches to tackling triangle computation in very large networks.

Parallel methods using shared-memory models: In a multi-core shared memory model, a shared memory is utilized to store the input graph such that the memory is globally available to multiple threads, and threads can work on triangle computation in parallel. Shared memory approaches are of high interest in industry and research community because, today, a single multi-core machine can support hundreds of cores and several terabytes of fast memory.

Green et al. [35] developed two parallel vertex- and edge-centric triangle counting algorithms with a reasonable balanced workload among threads. The edge-centric algorithm theoretically has a better load balancing while it requires more memory space than the vertex-centric algorithm requires. Green et al. showed that due a higher overhead, computational complexity, and synchronization cost, edge-centric approach has lower performance in practice. Shun and Tangwongsan [91] designed a parallel shared-memory algorithm, which consists of two main steps: 1) Vertices are ordered in parallel based on their degrees to improve the performance of memory access and to reduce the number of cache misses. 2) Set intersection is performed on the ordered adjacency lists to count the number of triangles in parallel. Tom et al. [96] introduced

OpenMP-based shared-memory triangle counting methods, using lists and hash-maps to enhance cache utilization and to reduce the overall computation cost. While the list- and map-based algorithms considerably achieve a better runtime than of [91], they still suffer from a high message communication overhead, affecting the overall runtime drastically.

In general, the discussed methods above suffer from some drawbacks, which are mainly caused by the shared-memory nature of the methods. Specifically, imbalanced workload, inefficient memory utilization, the high overhead of communications are the main challenges of shared-memory approaches. Kanewala et al. [47] designed a generic distributed, shared-memory parallel framework to tackle the mentioned issues. In their work, vertices are blocked and partitioned based on their degrees to reduce remote communication messages. In addition, blocking strategies, such as block aggregation, were deployed to reduce the number of small messages, which can cause nontrivial overheads. Donato et al. [28] introduced an OpenMP-based algorithm that is a variation of the sequential forward algorithm of [80]. They designed data preparation strategies equipped with a cache-efficient data structure, which is composed of two Compressed Sparse Row (CSR) matrices for maintaining upper and lower triangles of the adjacency matrix.

Parallel methods using shared-nothing models: Data distribution and parallelization arise as practical solutions for triangle computation in massive graphs. In a shared-nothing model, the parallel algorithm is reposnsible to split and dispers the input graph among commodity, independent processors, where processors do not share memory nor they interact with each other. Generally, triangle counting for a graph partition, driven to a processor, is performed locally. Ultimately, the output of processors are combined and aggregated to compute the final answer. Here, we discuss counting methods based on shared-nothing models such as MapReduce frameworks and Message Passing Interfaces (MPIs). MapReduce, as a de-facto standard tool in the shared-nothing environment, enables the distribution and processing of massive input data through three main phases: Map, Shuffle, and Reduce. Map converts data into pairs of entries, and Shuffle directs the generated pairs to corresponding reducer machines[2], based on a

[2] Machines in phase Reduce

user-defined hash function. In both phases Map and Reduce, machines of the MapReduce cluster work independently in parallel on counting triangles. Suri and Vassilvitskii [93] designed a vertex-centric method for counting the number of triangles using MapReduce. The algorithm has two phases. First, wedges (path of length 2) are generated by pivoting on vertices in parallel. Second, it counts wedges, generated in the first phase, which form a triangle (i.e., closed with an additional edge) such that wedges are distributed among different machines. Park and Chung [67] designed Triangle Type Partition (TTP), which is a MapReduce algorithm and is based on a graph partitioning technique. Each triangle of a graph is classified into one of three types, based on the number of partitions which its vertices are part of. This classification sensibly reduces duplication in triangle counting and leads to a higher throughput than its prior works.
There are also parallel triangle counting methods using MPI. Arifuzzaman et al. [9] introduced a parallel MPI-based algorithm, based on a vertex-centric algorithm for counting triangles. It employs an overlapping partitioning scheme for workload balancing among individual threads. This algorithm can efficiently process sparse networks. However, since the partitions might have overlaps, they can grow as large as the size of the entire network when the network has vertices with high degrees. Arifuzzaman et al. [10] made a further improvement in runtime through designing a space-efficient MPI-based algorithm, which employs a *non*-overlapping partitioning and achieves $25x$ speedup over their prior work. Tom and Karypis [95] introduced a distributed memory MPI-based algorithm, following their prior shared-memory algorithm [96]. Vertices are ordered based on their degrees and the input graph is stored using CSR format as a preprocessing step in this method. Then, a 2D decomposition of adjacency matrix is performed to cyclically distribute blocks of the input matrix among processors. The 2D decomposition extends the degree of parallelism and reduces the overall communication overhead. In addition, there are parallel hybrid triangle counting methods such as [3, 47], which are based on both shared-memory and distributed schemes to leverage the benefits of both platforms.

Parallel methods with GPU implementation: Modern GPUs, consisting of tens of multiprocessor cores, provide massive parallelism in order to deliver high throughput. Here, we

explore GPU-based triangle counting methods and indicate that GPU implementations are extremely fast and well suited for counting triangles in large graphs which can contain tens of billions of edges.

Green et al. [36] introduced a scalable GPU implementation for triangle counting with a multi-level parallelism. In the work of Green et al. [36], first, vertices of the graph are distributed among multiprocessors on GPU. Then, a path method based on merge path [34], which is a highly efficient GPU merging algorithm for set intersection, is performed on sub-tasks in individual threads. Further ahead, Polak [72] designed a triangle counting method using Compute Unified Device Architecture (CUDA[3]) programming model on multiple GPUs, which is basically the GPU version of the sequential forward algorithm, due to Schank and Wagner [80]. In this work, edges which are originated by the endpoint with a higher degree (in tiebreak with a higher ID) are filtered out. Then, the common neighbors of each remaining edge is computed in parallel for counting triangles. Another parallel vertex-centric algorithm for CUDA is developed by Bisson and Fatica [17]. Bitmaps are used to represent adjacency lists of a graph, and the lists are stored in a shared memory to accelerate the process of triangle counting in this method.

Further, Wang et al. [103] introduced GPU triangle counting methods to mitigate a high memory usage, required by prior work. Their method is based on three techniques: 1) Triangle counting using subgraph matching which uses a filtering-and-joining procedure using Gunrock programming model [105]. 2) Triangle counting using set intersection which adapts the forward algorithm of Schank and Wagner [80] for GPU implementation. 3) Triangle counting using matrix multiplication, inspired by the work of Azad et al. [11]. Their empirical evaluation shows that, in most cases, set intersection technique obtains a better performance than other methods. Recently, Hu et al. [40] designed TriCore, a GPU-based approach, which uses a binary search-based intersection method for counting the number triangles that an edge is part of. TriCore partition the input graph among GPUs and exploits a workload management for obtaining a better load balancing. In [39], Hu et al. further improved the runtime of their prior work, i.e., TriCore,

[3]CUDA, developed by NVIDIA, is a programming interface for parallel computing on GPUs.

through a better balanced workload of the input graph partitions and higher cache efficiency of data, stored in CPU memory, for a faster processing.
However, the aforementioned approaches such as merge-path and binary-search based methods suffer from unbalanced workload distribution, high time complexities, excessive pre-processing overheads, high memory usage, and costly memory accesses. Recently, H-Index, developed by Pandey et al. [66], resolves the drawbacks and inefficiencies through a hash-indexing mechanism and a friendly memory access design. H-Index employs hash-indexing for bucketing adjacency lists into buckets. Each entry of a longer adjacency list (i.e., the adjacency list of the endpoint with a higher degree) is hashed to the corresponding buckets. This process helps accelerate searching through the common neighbors of the two endpoints of an edge. Also, it provides a memory friendly storage format for GPUs which results in a high memory throughput. Remarkably, H-Index stands out from its prior works by achieving 141 billion Traversed Edge Per Second (TEPS) rate on graph Protein K-mer V2a with 64 GPUs.

2.1.3 Approximation Approaches

On an abstract level, approximation methods through sampling mechanisms have been proposed to cope with the increasing size of input graphs. In many real-world scenarios that an exact count is not necessarily required, a faster yet accurate estimator is highly preferable. It is worth noting that while there are numerous approximation algorithms for static and streaming graphs, there are far fewer sub-linear methods. Sub-linear triangle counting, including [30, 81, 106], approximate the number of triangles while require reading a small fraction of the entire input. Note that streaming and static randomized algorithms require to read the entire graph or stream, which may be time-consuming when the size of input graph is large. Here, we briefly discuss existing methods for static and streaming algorithms. We extensively evaluate and analyze state-of-the-arts of these two types of approximation methods in the next sections.
Approximation methods for static graphs: State-of-the-art sampling method, when the entire graph can be store in the main memory, is known as wedge sampling [83, 99]. [83] estimates

the triangle count by uniformly sampling wedges (a path of length two). Either the sampled wedge does not form a triangle or it does, the estimate is updated proportionately to keep the estimation of triangle count unbiased. The accuracy of this method is further improved by [98] due to Turk and Turkoglu. They narrowed down the sample search of candidate wedges through a global degeneracy ordering in vertices. Their estimator has a lower variance as the size of the sample search is smaller, hence a lower variance of the estimator.

In addition, the research community developed parallel approximation algorithms for static graphs. Rahman and Al Hasan [74] presented an approximation counting, based on edge-centric approach, for multicore machines. Kolda et al. [48] presented a wedge sampling counting method based on a MapReduce implementation. The MapReduce framework yields fast and accurate estimators and are enabled to deal with massive graphs, containing billions of edges.

Approximation methods for graph streams: In streaming model, edges of graph arrive from a stream and a limited memory budget (also called reservoir) is available for processing. This model is advantageous when the size of stream is huge and the entire graph cannot fit in the main memory. Some edges might be evicted from the memory to keep memory usage below the given limited budget. [14, 25, 45, 50, 60] presented one- and multi-pass streaming algorithms where the size of reservoir is not fixed and depends on the structure of graph. Pavan et al. [70] and Jha et al. [44] developed one-pass algorithms, which use a fixed-size reservoir for sampling edges from an insertion-only stream. Lim and Kang [55] presented one-pass algorithms using a fixed probability but not a fixed reservoir size. Their work is further improved by Stefani et al. [92]. They presented a suite of one-pass streaming algorithms using fixed-size reservoirs. In addition, they introduced an algorithm to handle fully dynamic streams, i.e., when both insertion and deletion of edges can occur over the stream. The estimators update the triangle count when an edge is inserted or deleted from the reservoir to maintain the estimations unbiased. Shin [86] improved the accuracy of estimates through a 2-layered reservoir where recent edges have a greater likelihood to get sampled and kept in the reservoir. They empirically verified that at any time stamp of a stream recent edges of real-world networks are more likely to form triangles than older

ones. In addition, Wang et al. [104] designed a one-pass streaming algorithm to handle edge duplicates in a graph stream.

There are also works on parallel and distributed streaming algorithms. The main aim in these algorithms is to enhance both runtime and accuracy through leveraging the benefits of parallelism and distributed memory. These algorithms were first studied by Pavan et al. [94], where they designed a parallel, shared-memory algorithm for estimating the triangle count in the entire input stream. Moreover, Shin et al. [90] designed a fast and accurate distributed algorithm in a shared-nothing scheme, which significantly improves their prior parallel and distributed method [87].

2.2 Bipartite Graphlets

Modeling the smallest unit of cohesion enables a principled way to analyze networks. While the literature is quite rich with the studies on counting triangles and small cliques in unipartite graphs [5, 21, 42, 43, 70, 71, 83, 94], these works are not applicable to bipartite networks. To the best of our knowledge, Borgatti and Everett [19] are the first to consider cohesive structures in bipartite networks to analyze social networks. They proposed to use the 3×3 biclique as the smallest cohesive structure, motivated by the fact that a triangle in a unipartite graph has three vertices, and the same should be considered for both vertex sets of the bipartite graph. Opsahl also proposed a similar approach to define the clustering coefficient in affiliation networks [64]. Robins and Alexander argued that the smallest structure with multiple vertices on both vertex sets is a better model for measuring cohesion in bipartite networks [75], as also discussed in a later work [56]. They used the ratio of the number of 2×2 bicliques (butterflies) to the number of 3-paths (a path of three edges) to define the clustering coefficient in bipartite graphs. The butterfly is also adopted in a recent work by Aksoy et al. [6] to generate bipartite graphs with community structure. Butterfly counting is also applicable to the study of graphical codes (Halford and Chugg [38]). The numbers of cycles of length g, $g + 2$, and $g + 4$ in bipartite graphs,

where g is the girth[4], characterize the decoding complexity – note that the butterfly is the simplest non-trivial cycle in a bipartite graph.

2.3 Butterflies

Recently, several algorithms are designed for exact butterfly counting in large graphs. Wang et al. [101], presented exact algorithms for butterfly counting in static graphs that outperform generic matrix multiplication based methods [8]. Wang et al. [102] introduced a vertex priority based which is considered as exact state-of-the-art and is built upon the exact algorithm of [76]. In vertex priority approach, a global ranking is applied to all vertices based on their degrees in the graph. Similarly to the exact algorithm of [76], wedges are processed for counting butterflies such that among three vertices of each wedge, the middle vertex of a vertex has the lowest rank. Therefore, the time complexity of the method is reduced to
$\min\{\sum_{u \in U(\mathcal{G})} \deg(u)^2, \sum_{v \in L(\mathcal{G})} \deg(v)^2\}$. Further ahead, Shi and Shun [85] presented several parallel algorithms for global and local butterfly counting. The core of the proposed framework, ParButterfly, is mainly relied on wedge counting. More specifically, wedges are built on the fly through their middle vertex, which is considered as building block for counting butterflies. Wedge aggregation step is performed in parallel to speedup the overall process. Moreover, a global ranking is applied to vertices to avoid any duplication in counting. The approximation parallel version of ParButterfly is built upon the randomized algorithms of [76]. We [76] presented exact and suite of randomized algorithms for butterfly counting on static bipartite graphs and further developed methods for butterfly counting in stream settings [77], where edge insertion is allowed.

2.4 Other Graphlets in Graphs

Recent works on counting 4-vertex [4] and 5-vertex subgraphs [71] have focused on exact counting and are not designed for streaming graphs. Because a butterfly is a 4-cycle, prior works on

[4]length of the shortest cycle in a graph

counting 4-cycles in any graph stream [15, 18, 22, 46, 59] can be also applied to a bipartite graph stream, and we compare with such prior work. Note that these algorithms do not use the additional structure in a bipartite graph (absence of edges between vertices in the same partition) and are thus naturally disadvantageous for bipartite streams. Bordino et al. [18] present three-pass algorithms for counting 4-cycles, while we focus on single-pass streaming algorithms. Buriol et al. [22] consider 3×3 biclique counting from a stream. They assume the incidence stream model, where edges in the graph stream are presented in a specific order such that all edges incident to a given vertex arrive together, whereas we assume the more general model where edges can arrive in an arbitrary order. Bera and Chakrabarti [15] present algorithms for counting 4-cycles in a graph using two passes through the stream. The work of Ahmed et al. Other works include Manjunath et al. [59] and Kane et al. [46] for subgraph counting based on graph sketches.

2.5 Research Problem

The main focus of this dissertation is to count the number of the most basic yet non-trivial graphlets such as triangles in unipartite networks and butterflies, i.e., 2×2 bicliques, in bipartite networks. For both triangles and butterflies we consider two types of networks: 1) Static networks, where the entire network is available at once, and the goal is to count the total number of a specific graphlet. 2) Network streams, when a sequence of edges appear at a time, and the goal is to maintain graphlet count at any time upon addition/deletion of edges.

CHAPTER 3. TRIANGLE COUNTING IN STATIC NETWORKS

In this section, we theoretically and empirically discuss two types of randomized algorithms for finding the approximate number of triangles through various sampling mechanisms. We provide in-depth analyses of the variances along with experimental comparisons on their accuracy and runtime. We also demonstrate that these efficient approximation algorithms are notably faster than exact computation. Moreover, we show that state-of-the-art methods yield accurate estimates as low as 1% error in only a few seconds.

3.1 One-shot Sparsification Algorithms

In one-shot sparsification algorithms, the input graph is downsized through a sampling step. Then, an exact triangle counting algorithm is applied on the sparsified graph, and the exact count is scaled up to estimate the actual number of triangles in the original graph. Here, we present two of such algorithms: DoulionSpar due to Tsourakakis et al. [97] and ClrSpar due to Pagh and Tsourakakis [65].

3.1.1 Doulion Sparsification

In DoulionSpar, the input graph $\mathcal{G}$ is first narrowed down into a smaller graph $\mathcal{G}'$ through the *independently* sampling of each edge using a fixed probability p. Then, the number of triangles in $\mathcal{G}'$ is computed and scaled up to derive an estimate of $\Delta(\mathcal{G})$. This is faster than working with the original graph $\mathcal{G}$ since the number of edges in $\mathcal{G}'$ can be much smaller. DoulionSpar is shown in Algorithm 1.

Lemma 1. *Let Y_{DSpar} denote the output of* DoulionSpar*, given input graph $\mathcal{G}$ and probability p. Then, $\mathbb{E}\left[Y_{DSpar}\right] = \Delta$. Then, the variance of* DoulionSpar *is*
$\mathbb{V}\text{ar}\left[Y_{DSpar}\right] = \Delta\left(p^{-3} - 1\right) + \kappa\left(p^{-1} - 1\right)$.

Algorithm 1: DOULIONSPAR

Input: A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, parameter p, $0 < p < 1$.

1 Construct sparsified graph $\mathcal{G}' = (\mathcal{V}, \mathcal{E}')$ by retaining each edge $e \in \mathcal{E}$ independently with probability p.

2 $\alpha_{\text{DSpar}} \leftarrow \Delta(\mathcal{G}')$ `// exact counting in` $\mathcal{G}'$

3 **return** $\alpha_{\text{DSpar}} \cdot p^{-3}$ `// scale up the count by` p^{-3}

Proof. For $i = 1, \cdots, \Delta$, let X_i denote a Bernoulli random variable such that X_i is 1 if i^{th} triangle exists in the sampled graph $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$, and 0 otherwise. Let $\alpha_{\text{DSpar}} = \sum_{i=1}^{\Delta} X_i$ as defined in Line 2 of Algorithm 1. With linearity of expectation, we know $\mathbb{E}[\alpha_{\text{DSpar}}] = \sum \mathbb{E}[X_i] = \sum \Pr[X_i = 1]$. Because each single edge is sampled independently with fixed probability p, a triangle appears in sampled graph $\mathcal{G}'$ with probability p^3, i.e., $\Pr[X_i = 1] = p^3$. Hence, $\mathbb{E}[Y_{\text{DSpar}}] = p^{-3} \cdot \mathbb{E}[\alpha_{\text{DSpar}}] = p^{-3} \cdot \sum \Pr[X_i = 1] = \Delta$. Here, we compute the variance of DOULIONSPAR. Two distinct triangles i and j share at most one edge which accounts for the covariances of random variables X_i and X_j, i.e., $\mathbb{C}\text{ov}[X_i X_j]$. Hence, the computation of $\mathbb{V}\text{ar}[Y_{\text{DSpar}}]$ leads to:

$$\begin{aligned}
\mathbb{V}\text{ar}[Y_{\text{DSpar}}] &= \mathbb{V}\text{ar}\left[p^{-3} \cdot \sum_{i=1}^{\Delta} X_i\right] \\
&= p^{-6}\left[\sum_{i=1}^{\Delta}\left(\mathbb{E}[X_i] - \mathbb{E}[X_i]^2\right) + \sum_{i \neq j} \mathbb{C}\text{ov}[X_i X_j]\right] \\
&= p^{-6}\left[\Delta \cdot (p^3 - p^6) + \sum_{i \neq j} \mathbb{C}\text{ov}[X_i X_j]\right]
\end{aligned}$$

To compute $\mathbb{C}\text{ov}[X_i X_j]$ of two distinct triangles i and j, we consider two cases: 1) Triangles i and j share no edge. The probability of both i and j are sampled in $\mathcal{G}'$ is p^6, i.e., $\mathbb{E}[X_i X_j] = p^6$. Thus, $\mathbb{C}\text{ov}[X_i X_j] = \mathbb{E}[X_i X_j] - \mathbb{E}[X_i]\mathbb{E}[X_j] = p^6 - p^6 = 0$. 2) Triangles i and j have an edge in common. The probability that both i and j appear in $\mathcal{G}'$ is p^5, i.e., $\mathbb{E}[X_i X_j] = p^5$. Hence, $\mathbb{C}\text{ov}[X_i X_j] = \mathbb{E}[X_i X_j] - \mathbb{E}[X_i]\mathbb{E}[X_j] = p^5 - p^6$, and $\mathbb{V}\text{ar}[Y_{\text{DSpar}}] = \Delta\left(p^{-3} - 1\right) + \kappa\left(p^{-2} - 1\right)$. □

The proof of Theorem 1 reveals that pairs of triangles with edges in common impact the variance. If such pairs of triangles are more in a particular graph, the variance of DOULIONSPAR will be higher.

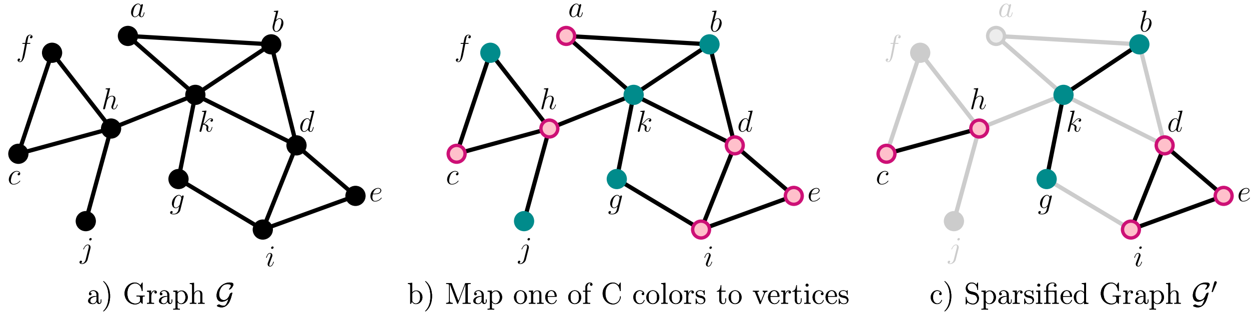


Figure 3.1: Sampling procedure in ClrSpar for C = 2. Edges with monochromatic endpoints are sampled and construct $\mathcal{G}'$.

Algorithm 2: ClrSpar

Input: A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, number of colors C.

1 Let $f : \mathcal{V} \rightarrow \{1, \cdots, \mathrm{C}\}$ `// map to random colors`
2 Construct $\mathcal{G}' = (\mathcal{V}, \mathcal{E}')$ such that $\mathcal{E}' \leftarrow \{(u, v) \in \mathcal{E} | f(v) = f(u)\}$
3 $\alpha_{\mathrm{CSpar}} \leftarrow \Delta(\mathcal{G}')$
4 **return** $\alpha_{\mathrm{CSpar}} \cdot p^{-2}$ where $p = 1/\mathrm{C}$

3.1.2 Colorful Sparsification

ClrSpar samples edges and constructs a smaller sparsified graph, as in DoulionSpar, while unlike DoulionSpar sampled edges have dependencies such that there is a greater likelihood to capture triangles in ClrSpar. The algorithm assigns one of C colors to every vertex in $\mathcal{G}$ uniformly at random and then downsizes graph $\mathcal{G}$ by keeping the edges whose endpoints have the same color (see Figure 3.1 for the sampling procedure in ClrSpar when C = 2). Ultimately, an exact counting is performed on the sparsified graph, and a derandomized factor is used to derive an unbiased estimate in the original graph. Algorithm 2 presents a description of ClrSpar. Note that the expected number of sampled edges is mp, the same as in DoulionSpar, while the scaling-up factor of ClrSpar is p^{-2}, which is smaller than of DoulionSpar $\left(p^{-3}\right)$. This fact implies that for the same sampling probability p and roughly the same size of sampled edges, ClrSpar can capture more triangles from the original graph.

Algorithm 3: IMPR-ESPAR

Input: A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, parameter p, $0 < p < 1$.

1 Construct sparsified graph $\mathcal{G}' = (\mathcal{V}, \mathcal{E}')$ by sampling each edge $e \in \mathcal{E}$ independently with probability p.
2 $\alpha_{\text{IMSpar}} \leftarrow 0$
3 **for each** *sampled edge* $(v, u) \in \mathcal{E}'$ **do**
4   **if** $d_v > d_u$ **then** Swap$(v,\ u)$
5   Randomly select vertex $x \in (\Gamma_v \setminus \{u\})$.
6   **if** $\{v, u, x\} \in \mathcal{T}(\mathcal{G})$ **then** `// forms a triangle`
7         $\alpha_{\text{IMSpar}} \leftarrow \alpha_{\text{IMSpar}} + d_v - 1$
8   **end**
9 **end**
10 **return** $\alpha_{\text{IMSpar}}/(3 \cdot p)$

Lemma 2. *Let Y_{CSpar} denote the output of* CLRSPAR, *given input graph $\mathcal{G}$ and C colors (here $p = 1/C$). Then, $\mathbb{E}[Y_{CSpar}] = \Delta$. Then, the variance of* CLRSPAR *is* $\mathbb{V}\text{ar}[Y_{CSpar}] = \Delta\left(p^{-2} - 1\right) + \kappa\left(p^{-1} - 1\right)$.

Proof. $\mathbb{E}[Y_{\text{CSpar}}]$ and $\mathbb{V}\text{ar}[Y_{\text{CSpar}}]$ can be derived through a pretty much the same approach as the analyses of Theorem 1. □

3.1.3 Improved Edge Sparsification

IMPR-ESPAR due to Turk and Turkoglu [99] comprises two separate sampling steps. First, IMPR-ESPAR sparsifies the input graph similar to algorithm DOULIONSPAR through sampling each edge independently with a fixed probability p. Then, it randomly samples wedges (i.e., two adjacent edges) whose one of edges is already selected with probability p in the previous sampling step. Finally, the estimator is updated if the sampled wedge is closed (i.e., it erects a triangle). Algorithm 3 describes IMPR-ESPAR in detail.

Lemma 3. *Let Y_{IMSpar} denote the output of* Impr-ESpar*, given input graph $\mathcal{G}$ and fixed probability p. Then, $\mathbb{E}\left[Y_{IMSpar}\right] = \Delta$. The variance of* Impr-ESpar *is:*

$$\mathbb{V}\mathrm{ar}\left[Y_{IMSpar}\right] = p\left(\sum_{\substack{e=(v,u)\in\mathcal{E}\\ d_v\le d_u}} d_v \cdot \Delta\left(e\right) - 3\right) - p^2\left(3\Delta + \kappa\right)$$

Proof. Note that here we only present a proof of the expectation. For a thorough analysis on the variance of Impr-ESpar, see Lemma 2 of [99].

For every three vertices $v, u, w \in \mathcal{V}$ such that $v \le u$ and $\{v, u, w\} \in \mathcal{T}(\mathcal{G})$ (i.e., v, u and w form a triangle in $\mathcal{G}$), let S_{vuw} be a random variable where S_{vuw} is $v-1$ if edge $e = (v, u)$ is sampled with probability p and vertex w is randomly selected from $\Gamma_v \setminus \{u\}$ (Line 5 of Algorithm 3), otherwise the random variable is 0. Hence,

$\mathbb{E}\left[S_{vuw} = v-1\right] = \Pr\left[(v,u) \in \mathcal{E}'\right] \cdot \Pr\left[\{v,u,w\} \in \mathcal{T}(\mathcal{G}) | (v,u) \in \mathcal{E}'\right] = p$. We compute α_{IMSpar} (defined in Algorithm 3) as follows:

$$\alpha_{\text{IMSpar}} = \sum_{\substack{e=(v,u)\in\mathcal{E}\\ v\le u}} \sum_{w\in\Gamma_v} S_{vuw} = 3 \cdot \Delta$$

Note that factor 3 appears in the result of α_{IMSpar} as every single triangle is counted by 3 distinct edges from the first summation. With linearity of expectation, we get

$\mathbb{E}\left[\alpha_{\text{IMSpar}}\right] = \sum_{(v,u)} \sum_{w} \mathbb{E}\left[S_{vuw}\right] = 3 \cdot p \cdot \Delta$. Thus,

$$\mathbb{E}\left[Y_{\text{IMSpar}}\right] = \frac{\mathbb{E}\left[\alpha_{\text{IMSpar}}\right]}{3p} = \Delta$$

which completes the proof. □

3.1.4 Approximation by Local Sampling

In this section, we present two methods that estimate $\Delta(\mathcal{G})$ through sampling small substructures from the input graph. The intuition in such approaches is that a particular subgraph is sampled, and then an estimation is derived. Since the subgraph is typically smaller than the original graph,

Algorithm 4: WSAMP (single iteration)

Input: A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$

1 $W \leftarrow \sum_{v \in \mathcal{V}} \binom{d_v}{2}$
2 Select a vertex $v \in \mathcal{V}$ with probability $\binom{d_v}{2}/W$.
3 Randomly select two distinct vertices $x, y \in \Gamma_v$.
4 **if** $\{v, x, y\} \in \mathcal{T}(\mathcal{G})$ **then** $\alpha_{\text{WSamp}} \leftarrow 1$
5 **else** $\alpha_{\text{WSamp}} \leftarrow 0$
6 **return** $\alpha_{\text{WSamp}} \cdot W/3$

it is less expensive to perform an exact computation. *The sampling process is repeated multiple times, and estimates are averaged to improve the accuracy* Here, we present two method based on wedge (path of length two) sampling: WSAMP due to Seshadhri [82] and IMPR-WSAMP due to Turk and Turkoglu [98].

3.1.5 Vanilla Wedge Sampling

Here, we present WSAMP due to Seshadhri et al. [82] which is a wedge sampling approach and is known as an efficient sampling scheme for triangle counting. WSAMP samples a wedge uniformly at random from $\mathcal{G}$ and then examines if the sampled wedge is closed (i.e., forms a triangle). To be more specific, WSAMP samples a vertex v with probability W_v/W, where $W_v = \binom{d_v}{2}$ and $W = \sum_{u \in \mathcal{V}} \binom{d_u}{2}$. Then, a pair of neighbors of v is selected uniformly at random which means vertex v coupled with the two sampled neighbors form a wedge. If the sampled wedge is closed (i.e., it is a part of a triangle), the estimator is updated accordingly. Remind that the sampling process is repeated multiple times, and averaged, to enhance the accuracy of estimates. Algorithm 4 presents a single iteration of WSAMP.

Lemma 4. *Let Y_{WSamp} denote the output of* WSAMP*, given graph $\mathcal{G}$. Then, $\mathbb{E}[Y_{WSamp}] = \Delta$ and $\mathbb{V}\text{ar}[Y_{WSamp}] = \Delta\left(\frac{W}{3} - 1\right) - 2\binom{\Delta}{2}$.*

Proof. Consider that the triangles in $\mathcal{G}$ are labeled from 1 to $\Delta(\mathcal{G})$. Let X_i denote a Bernoulli random variable such that X_i is 1 if i^{th} triangle includes the sampled wedge (sampled in Line 3 of Algorithm 4), and X_i is 0 otherwise. First, we show that each wedge is sampled uniformly at random, and then verifies that the expectation is unbiased and equals $\Delta(\mathcal{G})$.

Algorithm 5: IMPR-WSAMP (single iteration)

Input: A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$

1. Compute $d_v^*, \forall v \in \mathcal{V}$. `// degenercy number`
2. $\Gamma_v^* = \{u | (v,u) \in \mathcal{E} \wedge \mathsf{rank}(v) < \mathsf{rank}(u)\}$ and $d_v^* = |\Gamma_v^*|, \forall v \in \mathcal{V}$.
3. $W^* \leftarrow \sum_{v \in \mathcal{V}} \binom{d_v^*}{2}$
4. Select a vertex $v \in \mathcal{V}$ with probability $\binom{d_v^*}{2}/W^*$.
5. Randomly select two distinct vertices $x, y \in \Gamma_v^*$.
6. **if** $\{v, x, y\} \in \mathcal{T}(\mathcal{G})$ **then** $\alpha_{\text{WSamp}} \leftarrow 1$
7. **else** $\alpha_{\text{WSamp}} \leftarrow 0$
8. **return** $\alpha_{\text{WSamp}} \cdot W^*$

Consider a wedge $w = \{v, x, y\}$ such that $x, y \in \Gamma_v$. The probability that wedge w is sampled in Line 3 equals $\left(\binom{d_v}{2}/W\right) \cdot \left(1/\binom{d_v}{2}\right) = 1/W$, where $W = \sum_{v \in \mathcal{V}} \binom{d_v}{2}$, i.e., each wedge is sampled uniformly at random. Hence, we get $\Pr[X_i] = 3/W$ since each triangle is made up by three wedges.

Let $\alpha_{\text{WSamp}} = \sum_{i=1}^{\Delta} X_i$, as defined in Algorithm 4. Then, $\mathbb{E}[\alpha_{\text{WSamp}}] = \sum \mathbb{E}[X_i] = (3\Delta)/W$. Thus, $\mathbb{E}[Y_{\text{WSamp}}] = \mathbb{E}[\alpha_{\text{WSamp}}] \cdot (W/3) = \Delta$.

Here, we show the analysis on the variance of WSAMP.

$$\begin{aligned}\mathbb{V}\text{ar}[Y_{\text{WSamp}}] &= \mathbb{V}\text{ar}\left[\frac{W}{3}\sum_{i=1}^{\Delta} X_i\right] = \frac{W^2}{9}\mathbb{V}\text{ar}\left[\sum_{i=1}^{\Delta} X_i\right] \\ &= \frac{W^2}{9}\left[\Delta\left(\frac{3}{W} - \frac{9}{W^2}\right) + \sum_{i \neq j} \mathbb{C}\text{ov}[X_i X_j]\right]\end{aligned}$$

Given two distinct triangles i and j, we show that $\mathbb{C}\text{ov}[X_i X_j]$ is always negative. It is obvious that the probability that two distinct triangles i and j are discovered in each iteration of WSAMP is 0, which leads to $\mathbb{E}[X_i X_j] = 0$. It is because a sampled wedge can be part of at most one triangle, Thus, $\mathbb{C}\text{ov}[X_i X_j] = \mathbb{E}[X_i X_j] - \mathbb{E}[X_i]\mathbb{E}[X_j] = 0 - 9/W^2$. By further simplification, we derive $\mathbb{V}\text{ar}[Y_{\text{WSamp}}] = \Delta\left(\frac{W}{3} - 1\right) - 2\binom{\Delta}{2}$ which completes the proof. □

WSAMP achieves a high accuracy while it is much faster than exact computation as it requires to sample only a small fraction of wedges in graphs with trillions of triangles.

Table 3.1: Properties of static graphs. n, m, W, W^*, κ, and Δ denote the number of vertices, edges, wedges, ordered wedges, number of pairs of triangles with a common edge, and number of triangles respectively.[1]

Graphs	n	m	W	W^*	κ	Δ
DBLP	1.2M	5.1M	203M	96M	421M	11M
Journal	5.2M	48M	7.5B	2.4B	87B	310M
Orkut	3M	117M	45B	17B	67B	627M
Wiki	12M	288M	2.1T	386B	8.9T	11B
Twitter	31M	675M	107T	15T	119T	21B

3.1.6 Improved Wedge Sampling

Turk and Turkoglu in [98] improved the accuracy of WSamp through sampling wedges from a smaller sample space. Indeed, a global degeneracy ordering (called greedy ordering in [98]) is applied on vertices such that every single triangle can be captured through sampling only one of its three wedges, whereas in WSamp (that there is no ordering) a triangle can be found through sampling any of the three wedges. Indeed, Impr-WSamp narrows down the sample space, compared to WSamp, which leads to a lower variance and a better accuracy.

Here, we discuss the details of Impr-WSamp and the differences with WSamp. Let $\vartheta(v)$ denote the degeneracy number for vertex v, then for two distinct vertices v and u, we have $\mathsf{rank}(v) < \mathsf{rank}(u)$ if $\vartheta(v) < \vartheta(u)$ or $\vartheta(v) = \vartheta(u)$ and $v < u$. Given this ranking on vertices, Impr-WSamp converts input graph $\mathcal{G}$ into a directed graph $\mathcal{G}'$ such that there is an edge from vertex v to u in $\mathcal{G}'$ if edge (v, u) exists in the original graph and $\mathsf{rank}(v) < \mathsf{rank}(u)$. It is easy to verify that the directed graph is also acyclic (i.e., triangle free), which indicates that every three vertices of a triangle in the original graph is indeed a directed wedge (i.e., a directed path of length two) in $\mathcal{G}'$.

Impr-WSamp has two main differences: 1) Vertices are ordered based on their degeneracy number. 2) The probability that is assigned to every vertex (i.e., in Line 2 of Algorithm 4) is

[1] W^* is defined in the description of algorithm Impr-WSamp.

changed and equals to $\binom{d_v^*}{2}/W^*$, where $\forall v \in \mathcal{V}, d_v^*$ denote the degeneracy number of vertex v and $W^* = \sum_{v\in\mathcal{V}} \binom{d_v^*}{2}$.

Lemma 5. *Let Y_{IWSamp} denote the output of* Impr-WSamp*. Then,* $\mathbb{E}[Y_{IWSamp}] = \Delta$ *and* $\mathbb{V}\text{ar}[Y_{IWSamp}] = \Delta\left(\frac{W^*}{3} - 1\right) - 2\binom{\Delta}{2}$.

Proof. The proof of expectation and variance of Impr-WSamp is similar to the proof of Theorem 4, and the only difference is that we need to replace W^* with W in our computation. □

3.2 Experiments on Static Algorithms

We experimentally evaluate the runtime and accuracy performance of both one-shot and local sampling algorithms on various real-world networks from several domains such as social and web networks.

Networks, metrics, and experimental setup: Here, we describe the details of real-world networks, selected from publically available network repository [52]. Table 3.1 summarizes the characteristics of graphs. We converted all of the static graphs to simple, undirected by removing multiple edges and edge directions (if graph was directed). We shrunk graph `Twitter` by considering its first $675M$ edges.[2] We implemented the approximation algorithms for both static and streaming graphs in `C++`. The source codes are compiled with `g++` using `-Ofast` as the optimization level. Our source codes are publically available on Github repository.[3] We run the experiments on a machined equipped with a 2.0 GHZ 8-Core Intel E5 4620 and 1TB memory. Let the true value of triangle count be $x > 0$. Then, the relative error percent is defined as $100 \times |x - \hat{x}|/x$. When we use Error(%) to represent the results, we indeed refer to relative error percent. Moreover, we report the variance of one-shot algorithms, which is computed as mentioned in the description of algorithms.

Comparing one-shot approaches: Here, we compare one-shot algorithms DoulionSpar, ClrSpar, and Impr-ESpar. A common property of one-shot algorithms is the fixed probability

[2] We used larger graphs in the next section for streaming algorithms.

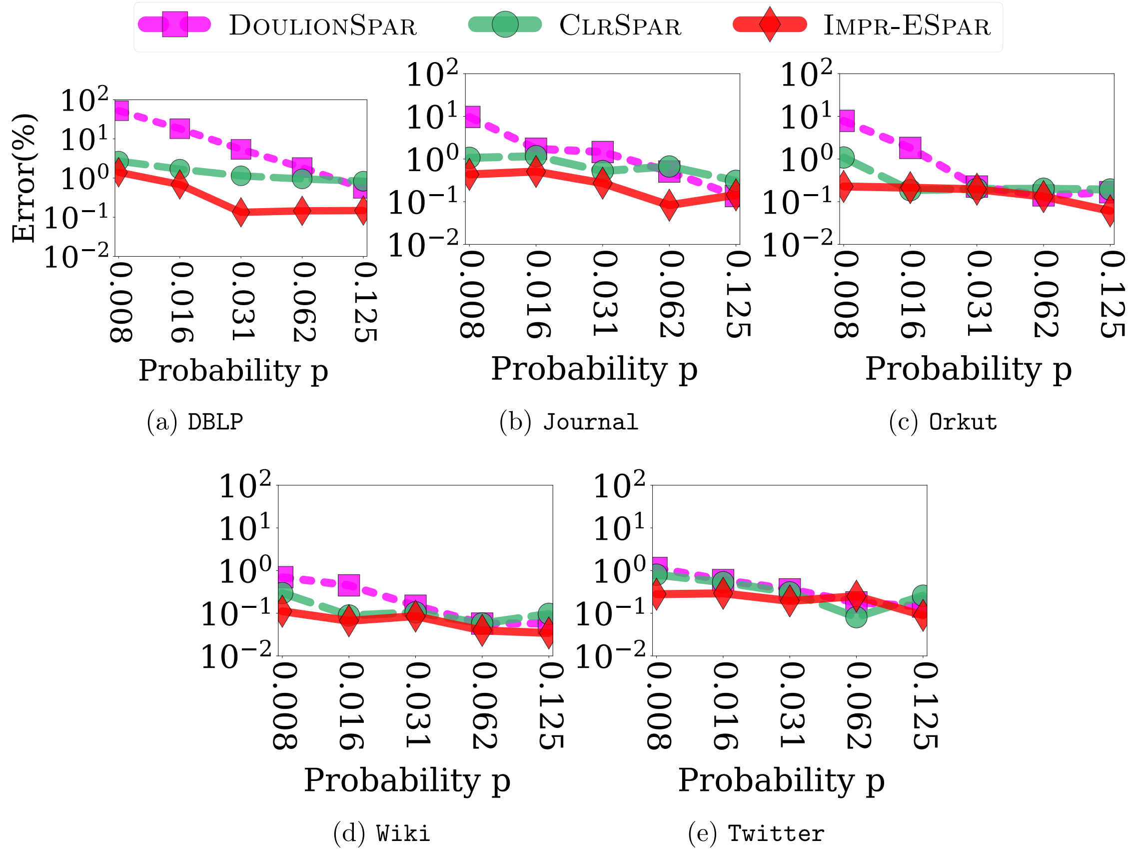
[3] https://github.com/beginner1010/Triangle-Counting

Table 3.2: Variances of DoulionSpar, ClrSpar, Impr-ESpar, for $p = 0.1$.

Graphs	DoulionSpar $\mathbb{V}\mathrm{ar}[Y_{\mathrm{DSpar}}]$	ClrSpar $\mathbb{V}\mathrm{ar}[Y_{\mathrm{CSpar}}]$	Impr-ESpar $\mathbb{V}\mathrm{ar}[Y_{\mathrm{IMSpar}}]$
DBLP	53.3×10^9	4.9×10^9	$\mathbf{130 \times 10^6}$
Journal	89.6×10^{12}	817×10^9	$\mathbf{22.8 \times 10^9}$
Orkut	7.2×10^{12}	666×10^9	$\mathbf{69.4 \times 10^9}$
Wiki	900×10^{12}	81.9×10^{12}	$\mathbf{2.2 \times 10^{12}}$
Twitter	11×10^{15}	1.07×10^{15}	$\mathbf{83 \times 10^{12}}$

p, which controls the likelihood of an edge being included from the original in the sparsified graph. It is rational to choose a small value, say less than 0.1, as the fixed probability p, in order to sample fewer edges in the sparsified graph. Therefore, counting triangles in a sparsified graph is expected to be computationally less expensive. We report both runtime and accuracy for various values of probability p in Figures 3.3 and 3.5. In addition, for a better comparison between algorithms DoulionSpar, ClrSpar, and Impr-ESpar, we report the variance of each algorithm for different graphs in Table 3.2.

The results of accuracy are consistent with the variance of estimators. Algorithms DoulionSpar and ClrSpar have a higher variance in different algorithms, which leads to a higher error percent. In all graphs, Impr-ESpar can achieve less than 1% error by a sampling probability as small as 0.008. However, DoulionSpar and ClrSpar end up with a higher error percent when such a low sampling probability is employed (for example DoulionSpar yielded 52% error in DBLP when the sampling probability is 0.008). In all algorithms, the accuracy and sampling probability are directly related to each other, i.e., when the sampling probability is higher, the accuracy of estimators is improved. The reason is that more edges are sampled with a higher sampling rate, and the scaling-up factor is smaller, which leads to a lower variance and a higher accuracy. Figure 3.5 reports the runtime of algorithms. Impr-ESpar is significantly faster, up to a factor of 177, than the two other one-shot algorithms, i.e., DoulionSpar and ClrSpar, which take similar runtimes on different graphs. In addition, when the sampling rate increases, the runtimes of DoulionSpar and ClrSpar increase substantially while the changes in the



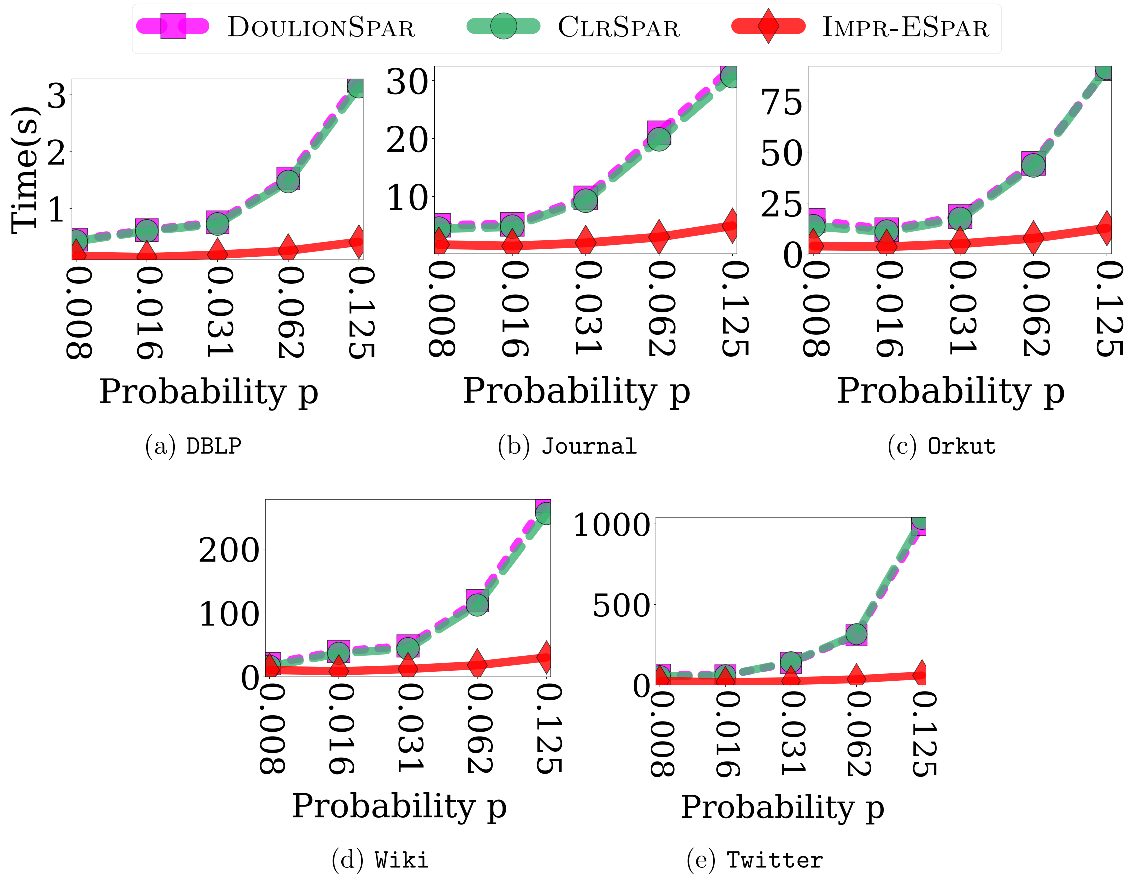
(a) DBLP (b) Journal (c) Orkut

(d) Wiki (e) Twitter

Figure 3.3: Accuracy of one-shot algorithms for different sampling probabilities. Impr-ESpar performs better than both DoulionSpar and ClrSpar, yielding $< 1\%$ relative error with sampling probability 0.008 for all graphs.

runtime of Impr-ESpar is negligible. Impr-ESpar is a faster estimator as it applies two sampling steps. More specifically, Impr-ESpar runs a sampling procedure to estimate the local triangle count of each sampled edge, unlike DoulionSpar and ClrSpar which apply an exact algorithm on the sparsified graph. To sum up, we can unquestionably conclude that Impr-ESpar is a better one-shot algorithm than both DoulionSpar and ClrSpar, as it is a faster and highly accurate estimator.

Comparing local sampling approaches: We compare the performance of WSamp and Impr-WSamp. WSamp immediately starts producing estimates of triangles based on sampled wedges while Impr-WSamp requires to rank vertices based on their degrees. Theoretically speaking, Theorem 5 and Table 3.3 show that Impr-WSamp is expected to yield better accuracy

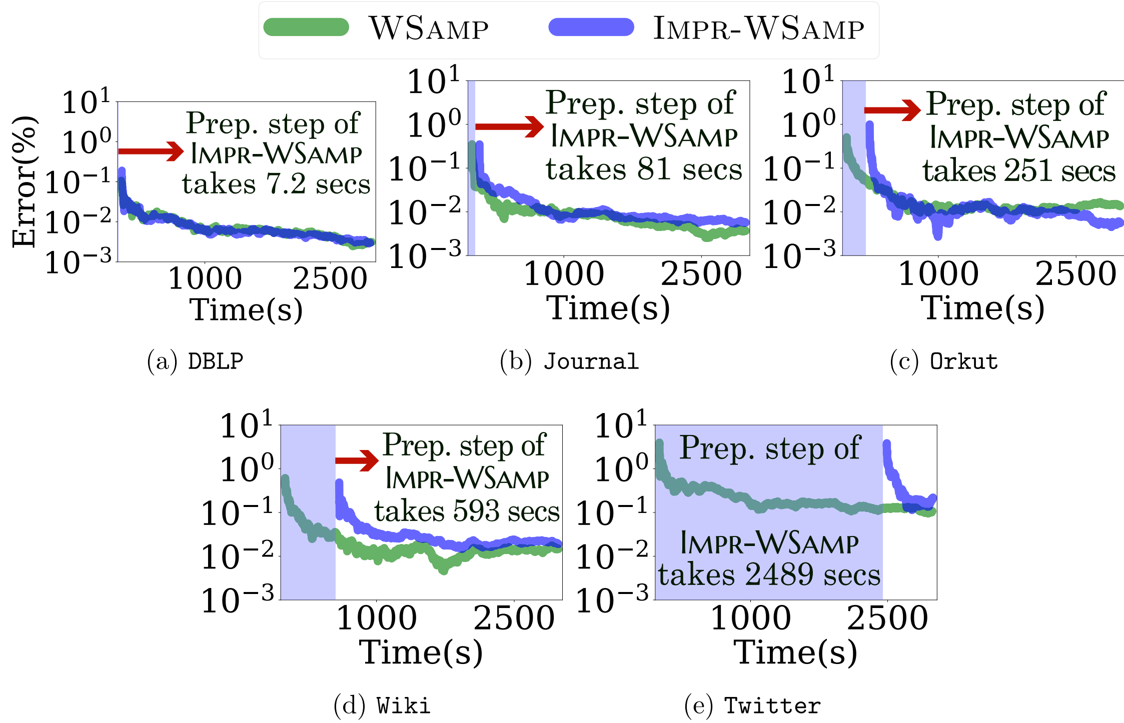


(a) DBLP (b) Journal (c) Orkut

(d) Wiki (e) Twitter

Figure 3.5: Time performance of one-shot algorithms for different sampling probabilities. Impr-ESpar is faster than DoulionSpar and ClrSpar with a speedup factor of 17 in graph Twitter.

than WSamp since it lowers the variance by narrowing down the sample space (More specifically, Impr-WSamp narrows down the number of candidate wedges through ranking vertices in a pre-processing step). However, Figure 3.7 shows that the pre-processing step could be computationally expensive which wipes out the benefits of obtaining a low-variance estimator. For example, Figure 3.6e shows that Impr-WSamp requires 2489 seconds for ranking vertices of Twitter before starting the sampling step while WSamp achieves less than 1% error in 9 seconds in the same graph. Note that the required time for the pre-processing step is directly related to the size of input graph. Therefore, the pre-processing step in Impr-WSamp could be quite time-consuming for large graphs with millions of edges while WSamp quickly obtains $< 1\%$ error, as shown in Figure 3.7.



(a) DBLP (b) Journal (c) Orkut

(d) Wiki (e) Twitter

Figure 3.7: Accuracy of local sampling algorithms, running in 3.000×10^3 seconds. WSAMP performs better than IMPR-WSAMP, due to the high preprocessing step of IMPR-WSAMP for vertex ranking, shown by the blue box. WSAMP yields $< 1\%$ relative error within at most 9 seconds for all graphs. IMPR-WSAMP requires 8, 82, 252, 594, 2516 seconds, including the preprocessing step, to achieve $< 1\%$ relative error in graphs DBLP, Journal, Orkut, Wiki, and Twitter respectively.

One-shot or local sampling approaches? Here, we compare the accuracy and runtime of best one-shot estimator, i.e., IMPR-ESPAR, and the best local sampling estimator, i.e., WSAMP. One-shot algorithms have three main drawbacks compared to local sampling algorithms: 1) The memory consumption of algorithm IMPR-ESPAR is $\mathcal{O}(\text{mp})$ to store the sparsified graph, where p is the fixed sampling probability and is larger than of WSAMP, whose memory consumption is $\mathcal{O}(d_{\max})$, where $d_{\max}$ denotes the maximum degree of the input graph. With such a low memory complexity, WSAMP is enabled to scale to larger graphs while IMPR-ESPAR requires memory linearly in the size of the input graph. 2) IMPR-ESPAR need to determine parameter p, prior to the sampling step, whereas WSAMP immediately starts the sampling procedure without any prior knowledge about the graph. If p is too large, more edges will be sampled, and counting triangles in the sparsified graph could be time-consuming. On the other hand, if p is too small, the

Table 3.3: Standard deviations of estimators on several graphs. For each method, the standard deviation is shown in the left column, and the Err, the ratio between the (theoretical) standard deviation and the number of triangles (i.e., Δ), is shown in the right column.

Graphs	WSAMP		IMPR-WSAMP	
	Err(%)	$\sqrt{\mathbb{V}\text{ar}\left[Y_{\text{WSamp}}\right]}$	Err(%)	$\sqrt{\mathbb{V}\text{ar}\left[Y_{\text{IWSamp}}\right]}$
`DBLP`	25.6×10^6	119.5	15.4×10^6	**32**
`Journal`	826×10^6	165.7	390×10^6	**25.7**
`Orkut`	3×10^9	382	1.8×10^9	**188.7**
`Wiki`	90×10^9	678	36×10^9	**216.4**
`Twitter`	880×10^9	3966.3	329×10^9	**1421.5**

accuracy is low. 3) IMPR-ESPAR assume that the entire graph is stored and accessible while WSAMP needs access to a small portion of the input graph. Therefore, WSAMP is more favorable when the entire graph is not available in memory. Overall, local sampling methods consume less memory and handle more restrictive settings which leads to attaining a broad application in real-world scenarios than one-shot sampling methods.

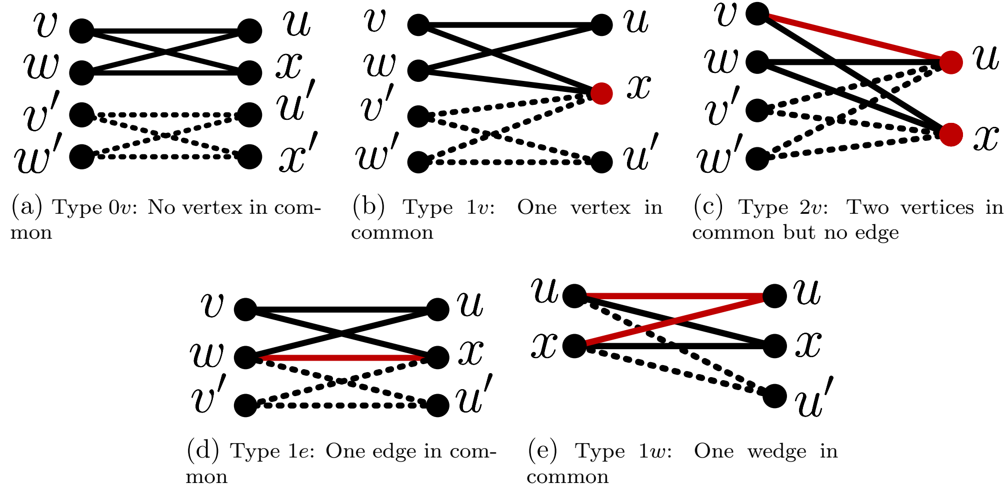
CHAPTER 4. BUTTERFLY COUNTING IN STATIC BIPARTITE NETWORKS

We present randomized algorithms for counting butterflies in a static bipartite network. We assume that the entire graph resides in memory. Our randomized algorithms are highly accurate and fast. More specifically, the algorithms obtain butterfly count in 5 seconds with relative error 1% in graphs with hundreds of millions of edges. Here, we discuss randomized algorithms [76] in detail which estimate the number of triangles based on two main approaches: 1) Local sampling 2) One-shot sparsification.

4.1 Approximation by Local Sampling

In this section, we present approaches to approximating $\bowtie(\mathcal{G})$ using random sampling. The intuition behind sampling is to examine a randomly sampled subgraph of $\mathcal{G}$ and compute the number of butterflies in the subgraph to derive an estimate of $\bowtie(\mathcal{G})$. Since the subgraph is typically much smaller than $\mathcal{G}$, it is less expensive to perform an exact computation. The size of the chosen subgraph, the cost of computing on it, and the accuracy of the estimate vary according to the method by which we choose the random subgraph. *This sampling process can be repeated multiple times, and averaged, in order to get a better accuracy.*

We consider three natural sampling methods: vertex sampling (VSamp), edge sampling (ESamp), and wedge sampling (WSamp). In VSamp, the subgraph is chosen by first choosing a vertex uniformly at random, followed by the induced subgraph on the distance-2 neighborhood of the vertex. In ESamp, a random edge is chosen, followed by the induced subgraph on the union of the immediate neighborhoods of the two endpoints of the edge. In WSamp, a random wedge (path of length two) is chosen, followed by the induced subgraph on the intersection of the immediate neighborhoods of the two endpoints of the wedge. While the methods themselves are



(a) Type $0v$: No vertex in common

(b) Type $1v$: One vertex in common

(c) Type $2v$: Two vertices in common but no edge

(d) Type $1e$: One edge in common

(e) Type $1w$: One wedge in common

Figure 4.2: A pair of butterflies in $\mathcal{G}$ can be of one of the above five types.

simple, the analysis of their accuracy involves having to deal carefully with the interactions of different butterflies being sampled together.

4.1.1 Vertex Sampling (Algorithm VSamp)

The idea in VSamp is to sample a random vertex v and count the number of butterflies that contain v – this is accomplished by counting the number of butterflies in the induced subgraph consisting of the distance-2 neighborhood of v in the graph. We show that the algorithm, described in Algorithm 6, yields an unbiased estimate of $\bowtie(\mathcal{G})$, and also analyze the variance of the estimate. The variance is reduced by taking the mean of multiple independent runs of the estimator.

Let Y_V denote the return value of Algorithm 6. Let p_V denote the number of pairs of butterflies in $\mathcal{G}$ that share a single vertex.

Algorithm 6: VSamp (single iteration)

Input: A bipartite graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$

Output: An estimate of $\bowtie(\mathcal{G})$

1. Choose a vertex v from V uniformly at random.
2. $\bowtie_v \leftarrow$ vBFC $(v, \mathcal{G})$ `// local counting [76]`
3. **return** $\bowtie_v \cdot n/4$

Lemma 1. $\mathbb{E}[Y_V] = \bowtie$, *and* $\mathbb{V}\mathrm{ar}[Y_V] \leq \frac{n}{4}(\bowtie + p_V)$

Proof. Consider that the butterflies in $\mathcal{G}$ are numbered from 1 to $\bowtie$. Let $X = \bowtie_v$, the number of butterflies that contain the vertex v, which is sampled uniformly. For $i = 1, \ldots, \bowtie$, let X_i be an indicator random variable equal to 1 if the ith butterfly includes the vertex v. We have $X = \sum_{i=1}^{\bowtie} X_i$. Since each butterfly has four vertices, $\mathbb{E}[X_i] = \Pr[X_i = 1] = 4/n$. Thus $\mathbb{E}[X] = \sum_{i=1}^{\bowtie} \mathbb{E}[X_i] = \sum_{i=1}^{\bowtie} \Pr[X_i = 1] = \frac{4\bowtie}{n}$. Since $Y_V = X \cdot \frac{n}{4}$, we have $\mathbb{E}[Y_V] = \bowtie$.
For the variance of Y_V, we consider the joint probabilities of different butterflies being sampled together. The set of all pairs of butterflies are partitioned into different types as follows. This partitioning will help not only with analyzing this algorithm, but also in subsequent sampling algorithms. A pair of butterflies is said to be of type:

- $0v$ if they share zero vertices (Figure 4.1a)
- $1v$ if they share one vertex (Figure 4.1b)
- $2v$ if they share two vertices but no edge (Figure 4.1c)
- $1e$ if they share two vertices and exactly one edge (Figure 4.1d)
- $1w$ if they share three vertices and two edges i.e. share a wedge (Figure 4.1e)

It can be verified that every pair of distinct butterflies must be one of the above types $\{0v, 1v, 2v, 1e, 1w\}$, and other combinations such as three vertices and more than two edges are not possible. For each type $t \in \{0v, 1v, 2v, 1e, 1w\}$, let p_t denote the number of pairs of butterflies of that type. Thus $p_{0v} + p_{1v} + p_{2v} + p_{1e} + p_{1w} = \binom{\bowtie}{2}$. Let $p_V = p_{1v} + p_{2v} + p_{1e} + p_{1w}$ be the number of pairs of butterflies that share at least one vertex.

$$\begin{aligned}
\mathbb{V}\mathrm{ar}[Y_V] = \mathbb{V}\mathrm{ar}\left[\frac{n}{4}\sum_{i=1}^{\bowtie} X_i\right] &= \frac{n^2}{16}\mathbb{V}\mathrm{ar}\left[\sum_{i=1}^{\bowtie} X_i\right] \\
&= \frac{n^2}{16}\left[\sum_{i=1}^{\bowtie}\mathbb{V}\mathrm{ar}[X_i] + \sum_{i \neq j}\mathbb{C}\mathrm{ov}[X_i, X_j]\right] \\
&= \frac{n^2}{16}\left[\bowtie\left(\frac{4}{n} - \frac{16}{n^2}\right) + \sum_{i \neq j}\left(\mathbb{E}[X_i X_j] - \mathbb{E}[X_i]\,\mathbb{E}[X_j]\right)\right]
\end{aligned}$$

Consider the different types of butterfly pairs (i, j):

- Type $0v$, there is zero probability of i and j being counted within $\bowtie_v$, hence $\mathbb{E}\left[X_i X_j\right] = 0$, $\mathbb{C}\mathrm{ov}\left[X_i, X_j\right] = -16/n^2$
- Type $1v$, $\mathbb{E}[X_i X_j] = \Pr[X_i = 1]\Pr[X_j = 1|X_i = 1] = (4/n)(1/4) = 1/n$. $\mathbb{C}\mathrm{ov}\left[X_i, X_j\right] = 1/n - 16/n^2$.
- Type $2v$, $\mathbb{E}[X_i X_j] = (4/n)(1/2) = 2/n$. $\mathbb{C}\mathrm{ov}\left[X_i, X_j\right] = 2/n - 16/n^2$
- Type $1e$, $\mathbb{E}[X_i X_j] = (4/n)(1/2) = 2/n$. $\mathbb{C}\mathrm{ov}\left[X_i, X_j\right] = 2/n - 16/n^2$
- Type $1w$, $\mathbb{E}[X_i X_j] = (4/n)(3/4) = 3/n$. $\mathbb{C}\mathrm{ov}\left[X_i, X_j\right] = 3/n - 16/n^2$.

By adding up the different contributions, we arrive: □

Let Z be the average of $\alpha = \frac{8n}{\epsilon^2 \bowtie}\left(1 + \frac{p_V}{\bowtie}\right)$ independent instances of Y_V. Using $\mathbb{V}\mathrm{ar}[Z] = \mathbb{V}\mathrm{ar}[Y_V]/\alpha$ and Chebyshev's inequality:

$$\Pr\left[|Z - \bowtie| \geq \epsilon \bowtie\right] \leq \frac{\mathbb{V}\mathrm{ar}[Z]}{\epsilon^2 \bowtie^2} = \frac{\mathbb{V}\mathrm{ar}[Y_V]}{\alpha\epsilon^2 \bowtie^2} \leq \frac{n(\bowtie + p_V)}{4\alpha\epsilon^2 \bowtie^2} = \frac{1}{32}$$

We can turn the above estimator into an (ϵ, δ) estimator by taking the median of $\mathcal{O}(\log(1/\delta))$ estimators, using standard methods.

Lemma 2. *There is an algorithm that uses $\tilde{\mathcal{O}}\left(\frac{n}{\bowtie}\left(1 + \frac{p_V}{\bowtie}\right)\right)$ iterations* [1] *of* VSAMP *(Algorithm 6) and yields an (ϵ, δ)-estimator of $\bowtie(\mathcal{G})$ using expected time $\tilde{\mathcal{O}}\left(\frac{m\Delta}{\bowtie}\left(1 + \frac{p_V}{\bowtie}\right)\right)$. Expected additional space is $\mathcal{O}\left(\frac{m\Delta}{n}\right)$.*

Proof. An iteration of VSAMP samples a vertex v and calls vBFC of [76] for local butterfly counting once, which takes $\mathcal{O}(|\Gamma_v^2|)$ time. Hence, the expected runtime of an iteration is $\mathcal{O}\left(\mathbb{E}\left[|\Gamma_v^2|\right]\right)$, where the expectation is taken over a uniform random choice of a vertex. We note that $|\Gamma_v^2| \leq d_v\ \Delta$ where d_v is v's degree and Δ is the maximum degree in the graph. Thus $\mathbb{E}\left[|\Gamma_v^2|\right] \leq \sum_{v \in V} \frac{1}{n} d_v \Delta = \frac{\Delta}{n}\sum_{v \in V} d_v = \frac{m\Delta}{n}$. The space of VSAMP is same with vBFC of [76]; $\mathcal{O}(|\Gamma_v^2|)$ for handling vertex v. The expected value is $\mathcal{O}(\frac{m\Delta}{n})$. □

[1] We use the notation $\tilde{\mathcal{O}}(f)$ to suppress the factor $\frac{\log(1/\delta)}{\epsilon^2}$, i.e. mean $\mathcal{O}\left(f \cdot \frac{\log(1/\delta)}{\epsilon^2}\right)$

Algorithm 7: ESAMP (single iteration)

Input: A bipartite graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
Output: An estimate of $\bowtie(\mathcal{G})$

1 Choose an edge e from E uniformly at random.
2 $\bowtie_e \leftarrow$ EBFC $(e, \mathcal{G})$ `// local counting`
3 using EBFC of [76]
4 **return** $\bowtie_e \cdot m/4$

4.1.2 Edge Sampling (Algorithm ESamp)

In this algorithm, the idea is to sample a random edge and count the number of butterflies that contain this edge, using EBFC of [76]. We present ESAMP in Algorithm 7, and state its properties.

Lemma 3. *Let Y denote the return value of Algorithm 7. Then, $\mathbb{E}[Y] = \bowtie$.*

Proof. Consider that the butterflies in $\mathcal{G}$ are numbered from 1 to $\bowtie$. Let X denote $\bowtie_e$, when edge e is chosen randomly from E. For $i = 1, \ldots, \bowtie$, let X_i be an indicator random variable equal to 1 if edge e is contained in butterfly i, and 0 otherwise. We have $X = \sum_{i=1}^{\bowtie} X_i$. Since each butterfly has exactly four edges in E, we have $\Pr[X_i = 1] = 4/m$.

$$\mathbb{E}[X] = \sum_{i=1}^{\bowtie} \mathbb{E}[X_i] = \sum_{i=1}^{\bowtie} \Pr[X_i = 1] = \frac{4 \bowtie}{m}$$

Since $Y = X \cdot \frac{m}{4}$, it follows that $\mathbb{E}[Y] = \bowtie$. □

Let p_E be the number of pairs of butterflies that share at least one edge. Then, $p_E = p_{1e} + p_{1w}$.

Lemma 4. $\mathbb{V}\mathrm{ar}[Y] \leq \frac{m}{4}(\bowtie + p_E)$

Proof. Proceeding similar to proof of Lemma 1:

$$\begin{aligned}
\mathbb{V}\mathrm{ar}[Y] = \mathbb{V}\mathrm{ar}\left[\frac{m}{4}\sum_{i=1}^{\bowtie} X_i\right] &= \frac{m^2}{16}\mathbb{V}\mathrm{ar}\left[\sum_{i=1}^{\bowtie} X_i\right] \\
&= \frac{m^2}{16}\left[\sum_{i=1}^{\bowtie}\mathbb{V}\mathrm{ar}[X_i] + \sum_{i \neq j}\mathbb{C}\mathrm{ov}[X_i, X_j]\right] \\
&= \frac{m^2}{16}\left[\bowtie\left(\frac{4}{m} - \frac{16}{m^2}\right) + \sum_{i \neq j}\left(\mathbb{E}[X_i X_j] - \mathbb{E}[X_i]\,\mathbb{E}[X_j]\right)\right]
\end{aligned}$$

For a pair of butterflies (i, j) of

- Type $0v$, $1v$, or $2v$: there is zero probability of i and j being counted within $\bowtie_e$, and hence $\mathbb{E}[X_i X_j] = 0$. $\mathbb{C}\text{ov}[X_i, X_j] = -16/m^2$

- Type $1e$: $\mathbb{E}[X_i X_j] = \Pr[X_i = 1]\Pr[X_j = 1 | X_i = 1] = (4/m)(1/4) = 1/m$. $\mathbb{C}\text{ov}[X_i, X_j] = 1/m - 16/m^2$

- Type $1w$, $\mathbb{E}[X_i X_j] = \Pr[X_i = 1]\Pr[X_j = 1 | X_i = 1] = (4/m)(2/4) = 2/m$. $\mathbb{C}\text{ov}[X_i, X_j] = 2/m - 16/m^2$

$$\begin{aligned}\mathbb{V}\text{ar}[Y] &= \frac{m^2}{16}\Bigg[\bowtie\left(\frac{4}{m} - \frac{16}{m^2}\right) - p_{0v}\frac{16}{m^2} - p_{1v}\frac{16}{m^2} \\ &\quad - p_{2v}\frac{16}{m^2} + p_{1e}\left(\frac{1}{m} - \frac{16}{m^2}\right) + p_{1w}\left(\frac{2}{m} - \frac{16}{m^2}\right)\Bigg] \\ &\leq \frac{m^2}{16}\left[\frac{4}{m}(\bowtie + p_{1e} + p_{1w}) - \binom{\bowtie}{2}\frac{16}{m^2}\right] \\ &= \frac{m}{4}(\bowtie + p_{1e} + p_{1w}) - \binom{\bowtie}{2} \leq \frac{m}{4}(\bowtie + p_E)\end{aligned}$$

□

Let Z be the average of $\alpha = (8m(1 + p_E/\bowtie))/(\epsilon^2 \bowtie)$ independent instances of Y. Using Chebyshev's inequality:

$$\Pr[|Z - \bowtie| \geq \epsilon \bowtie] \leq \frac{\mathbb{V}\text{ar}[Z]}{\epsilon^2 \bowtie^2} = \frac{\mathbb{V}\text{ar}[Y]}{\alpha\epsilon^2 \bowtie^2} \leq \frac{m(\bowtie + p_E)}{4\alpha\epsilon^2 \bowtie^2} = \frac{1}{32}$$

Lemma 5. *There is an algorithm that uses $\tilde{\mathcal{O}}\left(\frac{m}{\bowtie}\left(1 + \frac{p_E}{\bowtie}\right)\right)$ iterations of Algorithm 8 and yields an (ϵ, δ)-estimator of $\bowtie(\mathcal{G})$ using time $\tilde{\mathcal{O}}\left(\frac{m^2\Delta}{n\bowtie}\left(1 + \frac{p_E}{\bowtie}\right)\right)$. The additional space complexity is $\mathcal{O}(m\Delta/n)$.*

Proof. Each iteration of the ESAMP algorithm counts $\bowtie_e$ for a randomly chosen e. Similar to the analysis of VSAMP, the expected time of an iteration is $\mathcal{O}(m\Delta/n)$. Hence, the runtime follows. The space complexity is the space taken by Algorithm EBFC of [76], which is $\mathcal{O}(m\Delta/n)$. □

4.1.2.1 Wedge Sampling (Algorithm WSamp)

In WSAMP, we first choose a random "wedge", a path of length two in the graph. This already yields three vertices that can belong to a potential butterfly. Then we count the number of

Algorithm 8: WSAMP (single iteration)

Input: A bipartite graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$

Output: An estimate of $\bowtie(\mathcal{G})$

1 $\wedge \leftarrow \sum_{u \in V} \binom{d_u}{2}$ `// number of wedges`

2 Choose a vertex $u \in V$ with probability $\binom{d_u}{2} / \wedge$

3 Choose two distinct vertices $v, w \in \Gamma_u$ uniformly at random

4 $\beta \leftarrow |\Gamma_v \cap \Gamma_w| - 1$ `// # butterflies that has` (u, v, w)

5 **return** $\beta \cdot \wedge / 4$

butterflies that contain this wedge by finding the intersection of the neighborhoods of the two endpoints of the wedge. Algorithm 8 describes the WSAMP algorithm. Let Y_W denote the return value of Algorithm 8.

Lemma 6. *The wedge (x, y, z) in lines 2 and 3 of Algorithm 8 is chosen uniformly at random from the set of all wedges in $\mathcal{G}$.*

Proof. In order to choose the wedge (x, y, z), vertex y must be chosen in step (2), followed by vertices x and z in step (3). The probability is $\frac{\binom{d_y}{2}}{h} \cdot \frac{1}{\binom{d_y}{2}} = 1/h$, showing that a wedge is sampled uniformly at random. □

Lemma 7. *Let Y denote the return value of Algorithm 8. Then, $\mathbb{E}[Y] = \bowtie$*

Proof. Consider that the butterflies in $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ are numbered from 1 to $\bowtie$. Suppose that $\wedge$ is the number of wedges in $\mathcal{G}$. For $i = 1, \ldots, \bowtie$, let X_i be an indicator random variable equal to 1 if the ith butterfly includes wedge w. Let X denote β, the number of butterflies that contain the sampled wedge (x, y, z). We have $X = \sum_{i=1}^{\bowtie} X_i$. Since each butterfly has four wedges in $\mathcal{G}$, we have $\Pr[X_i = 1] = 4/\wedge$. The rest of the proof is similar to that of Lemma 1. □

Lemma 8. $\mathbb{V}\text{ar}[Y] \leq \frac{\wedge}{4}(\bowtie + p_{1w})$

Proof.

$$\mathbb{V}\text{ar}[Y] = \mathbb{V}\text{ar}\left[\frac{\wedge}{4} \sum_{i=1}^{\bowtie} X_i\right] = \frac{\wedge^2}{16} \mathbb{V}\text{ar}\left[\sum_{i=1}^{\bowtie} X_i\right]$$
$$= \frac{\wedge^2}{16}\left[\bowtie\left(\frac{4}{\wedge} - \frac{16}{\wedge^2}\right) + \sum_{i \neq j} (\mathbb{E}[X_i X_j] - \mathbb{E}[X_i]\,\mathbb{E}[X_j])\right]$$

For a pair of butterflies (i, j) of

- Type $0v$, $1v$, $2v$, or $1e$, there is zero probability of i and j being counted together, and hence $\mathbb{E}[X_i X_j] = 0$. We have $\mathbb{C}\text{ov}[X_i, X_j] = -16/\wedge^2$

- Type $1w$, $\mathbb{E}[X_i X_j] = \Pr[X_i = 1] \Pr[X_j = 1 | X_i = 1] = (4/\wedge)(1/4) = 1/\wedge$. Therefore, $\mathbb{C}\text{ov}[X_i, X_j] = 1/\wedge - 16/\wedge^2$.

$$\begin{aligned}
\mathbb{V}\text{ar}[Y] &= \frac{\wedge^2}{16}\Bigg[\bowtie\left(\frac{4}{\wedge} - \frac{16}{\wedge^2}\right) - p_{0v}\frac{16}{\wedge^2} - p_{1v}\frac{16}{\wedge^2} \\
&\quad - p_{2v}\frac{16}{\wedge^2} - p_{1e}\frac{16}{\wedge^2} + p_{1w}\left(\frac{1}{\wedge} - \frac{16}{\wedge^2}\right)\Bigg] \\
&\le \frac{\wedge^2}{16}\left[\frac{4}{\wedge}(\bowtie + p_{1w}) - \binom{\bowtie}{2}\frac{16}{\wedge^2}\right] \\
&= \frac{\wedge}{4}(\bowtie + p_{1w}) - \binom{\bowtie}{2} \le \frac{\wedge}{4}(\bowtie + p_{1w})
\end{aligned}$$

□

Let Z be the average of $\alpha = (8\wedge(1 + p_{1w}/\bowtie))/(\epsilon^2 \bowtie)$ independent instances of Y. Using Chebyshev's inequality, we arrive that Z is an $(\epsilon, 1/32)$-estimator of $\bowtie(\mathcal{G})$.

Lemma 9. *There is an algorithm that uses $\tilde{\mathcal{O}}\left(\frac{\wedge}{\bowtie}\left(1 + \frac{p_{1w}}{\bowtie}\right)\right)$ iterations of Algorithm 8 and yields an (ϵ, δ)-estimator of $\bowtie(\mathcal{G})$ using time $\tilde{\mathcal{O}}\left(\frac{(\Delta + \log n)\wedge}{\bowtie}\left(1 + \frac{p_{1w}}{\bowtie}\right)\right)$ and space $\mathcal{O}(n)$.*

Proof. We first consider the runtime of Algorithm 8. At first glance, it looks like Steps (1) and (2) take $\mathcal{O}(n)$ time. This can be reduced to $\mathcal{O}(\log n)$ using a pre-computation step of time $\mathcal{O}(n)$, which precomputes the value of h, and also stores the values of $\binom{d_v}{2}$ for different vertices v in an array $A[1 \ldots n]$ of length n. We consider another array $B[1 \ldots n]$, where each element $B[j] = \sum_{i=1}^{j} A[i]$. When WSAMP is called, a random number r in the range $\{1, 2, \ldots, h\}$ is first generated, and Step (2) is accomplished through finding the smallest j such that $B[j] \ge r$. Since array B is sorted in increasing order, this can be done in time $\mathcal{O}(\log n)$ using a binary search. Steps (3) and (4) of Algorithm 8 can be performed in $\mathcal{O}(\Delta)$ time – Step (4), for example, through storing all elements of Γ_x in a hash set, and repeatedly querying for elements of Γ_y in this hash set, for a total of $\mathcal{O}(d_y) = \mathcal{O}(\Delta)$ time. The additional space required by this algorithm, in addition to the stored graph itself, is $\mathcal{O}(n)$, for random sampling. □

Table 4.1: Standard deviations of estimators on large graphs. For each method, the theoretical upper bound on the standard deviation is shown in the left column, and the "error", the ratio between the (theoretical) standard deviation and the number of butterflies, is shown in the second column.

	VSAMP		ESAMP		WSAMP	
	$\sqrt{\frac{(\bowtie+p_V)n}{4}}$	error %	$\sqrt{\frac{(\bowtie+p_E)m}{4}}$	error %	$\sqrt{\frac{(\bowtie+p_{1w})h}{4}}$	error %
`Deli`	2.8×10^{13}	494.9	2.1×10^{12}	38.3	7.7×10^{11}	13.6
`Journal`	2.3×10^{15}	708.56	2.1×10^{13}	6.4	1.4×10^{14}	99.3
`Orkut`	5×10^{15}	223.6	2×10^{14}	9.2	2.9×10^{14}	13.3
`Web`	6.8×10^{16}	3431.5	8.2×10^{13}	4.1	4.3×10^{16}	2194.8
`Wiki`	1.3×10^{16}	6823.6	3.8×10^{13}	19	1.4×10^{15}	703.4

4.1.3 Accuracy and Runtime of Sampling

Accuracy of a single iteration: In order to understand the relation between the three sampling algorithms, we compare the variance of the estimates retuned by these algorithms. Note that each of them returns an unbiased estimate of the number of butterflies. The standard deviations (square root of variances) of VSAMP, ESAMP, and WSAMP for different graphs are estimated and summarized in Table 4.1. The results show that the variances of VSAMP and WSAMP are much higher than the variance of ESAMP. Note that these are estimates of an upper bound on the variances, and the actual variances could be (much) smaller.

The variance of VSAMP is proportional to np_V where p_V is the number of *pairs of butterflies that share a vertex* and n is the number of vertices. The variance of ESAMP is proportional to mp_E where p_E is the number of *pairs of butterflies that share an edge* and m is the number of edges. Note that typically $\bowtie \ll p_V, p_E$ and hence $\bowtie + p_V \approx p_V$ and $\bowtie + p_E \approx p_E$. If two butterflies share an edge, they certainly share a vertex, hence $p_E \leq p_V$. Since it is possible that two butterflies share a vertex but do not share an edge, p_E could be much smaller than p_V. It turns out that in most of these graphs, p_E was much smaller than p_V. On the other hand, the number of vertices in a graph (n) is comparable to the number of edges (m). Typically $m < 10n$, and only in one case (`Orkut`), we have $m \approx 30n$. Thus, $np_V \ll mp_E$ for the graphs we consider. As a

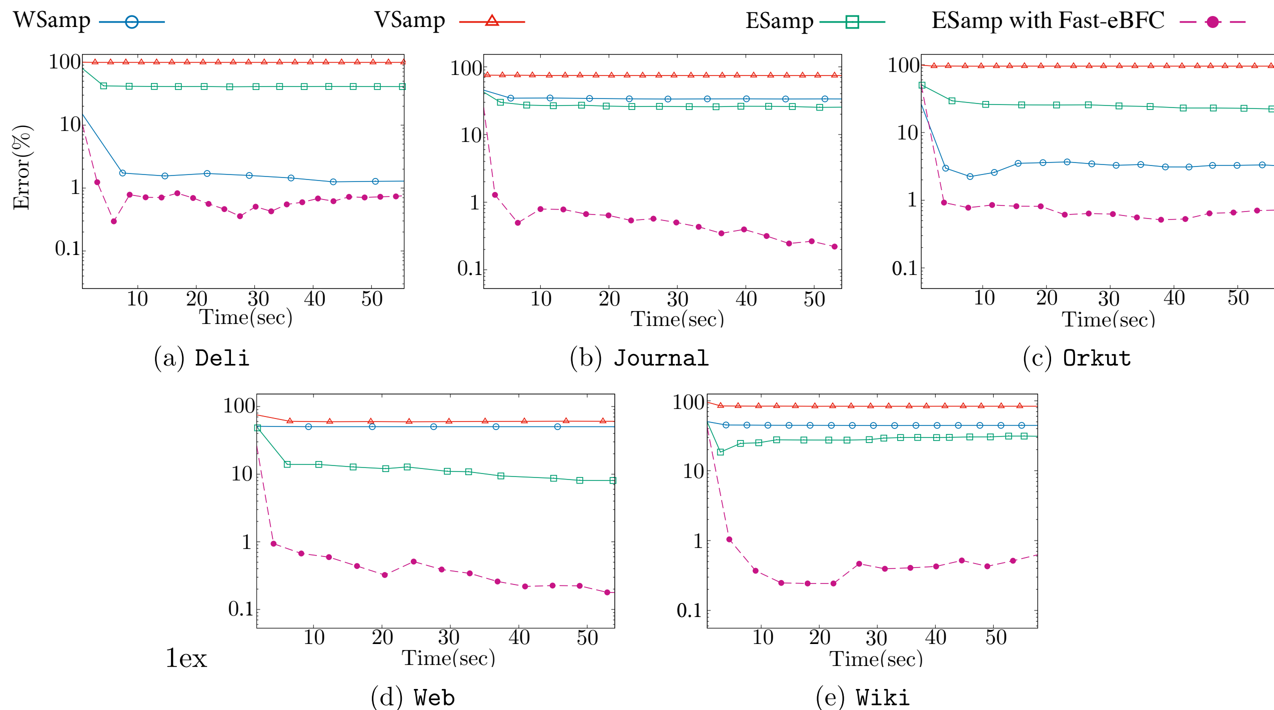


Figure 4.4: Relative error as a function of runtime, for sampling algorithms. ESAMP with FAST-EBFC yields < 1% relative error within 5 seconds for all networks.

result, the variance of VSAMP is much larger than the variance of ESAMP, which is reflected clearly in Table 4.1.

Comparing ESAMP with WSAMP, we note that the variance of WSAMP is proportional to $\wedge \cdot p_{1w}$, where $\wedge$ is the number of wedges in the graph ($\mathcal{O}(\sum_v d_v^2)$) and p_{1w} is the number of pairs of butterflies that share a wedge. The variance of ESAMP is proportional to mp_E. $p_{1w} \leq p_E$ since each pair of butterflies that shares a wedge also shares an edge. At the same time, we see that $\wedge$ is substantially greater than m. Overall, there is no clear winner among WSAMP and ESAMP in theory, but ESAMP seems to have the smaller variance on real-world networks, typically, sometimes much smaller, as in graph Wiki.

Runtime per iteration: The performance of a sampling algorithm depends not only on the variance of an estimator, but also on how quickly an estimator can be computed. Figure 4.5 shows the time taken to compute a single estimator using different sampling algorithms for the five largest networks. We note that ESAMP (which calls EBFC) requires the largest amount of

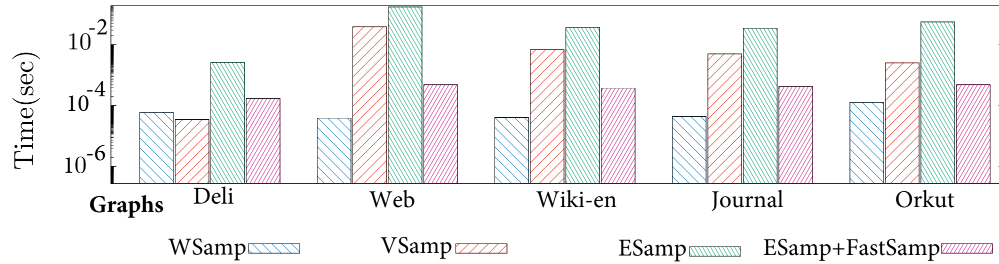


Figure 4.5: Average time per iteration of sampling. For ESAMP with FAST-EBFC, the time is shown for 1000 iterations of FAST-EBFC (Algorithm 9).

time per sampling step, among all estimators. This decreases the overall accuracy of ESAMP, despite its smaller variance.

4.1.4 Faster Edge Sampling using Fast-eBFC

Since ESAMP had a low variance, but a high runtime per iteration, we tried to achieve different tradeoffs with respect to runtimes and accuracy, which led to our next algorithm FAST-EBFC, a faster variant of butterfly counting per edge (EBFC). Like ESAMP, we first sample a random edge from the graph, but instead of exactly counting the number of butterflies that contain the sampled edge, which leads to (relatively) expensive iterations, FAST-EBFC only estimates the number of butterflies per edge, through a further sampling step (replacing EBFC in Line 2 of ESAMP (Algorithm 7)). For an edge (u, v) this estimation is performed by randomly choosing one neighbor each of u and of v, and checking if the four vertices form a butterfly. Algorithm 9 presents a single iteration of FAST-EBFC. This procedure is repeated a few times for a given edge, to improve the accuracy of the estimate. In our implementation we repeated it 1000 times for each sampled edge, and it was still significantly faster than EBFC (Figure 4.5). We use it instead of the EBFC algorithm in ESAMP. While the estimate from each iteration is less accurate than in ESAMP, more iterations are possible within the same time. FAST-EBFC (with 1000 repetitions) is faster than EBFC by 15x-357x, when used in ESAMP. Overall, in large graphs, this leads to an improvement in accuracy over EBFC in ESAMP. Let Y_{FE} denote the return value of Algorithm 9.

Lemma 10. $\mathbb{E}[Y_{FE}] = \bowtie_e$ *and* $\mathbb{V}\mathrm{ar}[Y_{FE}] \leq \bowtie_e (d_u \cdot d_v)$.

Algorithm 9: FAST-EBFC (replaces EBFC in ESAMP)

Input: An edge $e = (v, u) \in E$ in $\mathcal{G} = (\mathcal{V}, \mathcal{E})$

Output: An estimate of $\bowtie_e$

1 Choose a vertex w $(\neq v)$ from Γ_u uniformly at random
2 Choose a vertex x $(\neq u)$ from Γ_v uniformly at random
3 **if** (u, v, w, x) *forms a butterfly* **then** $\beta \leftarrow 1$
4 **else** $\beta \leftarrow 0$
5 **return** $\beta \cdot d_u \cdot d_v$

Proof. Consider that the butterflies include the edge $e = (u, v)$ are numbered from 1 to $\bowtie_e$. For $i = 1, \ldots, \bowtie_e$, let X_i be an indicator random variable. X_i is equal 1 if the ith butterfly contains both sampled neighbors v' and u', otherwise 0. Let $X = \sum_{i=1}^{\bowtie_e} X_i$. Since vertex v' and u' are chosen with probability $1/d_v$ and $1/d_u$ respectively, $\Pr[X_i = 1] = 1/(d_v \cdot d_u)$. Then,

$$\mathbb{E}[X] = \sum_{i=1}^{\bowtie_e} \mathbb{E}[X_i] = \sum_{i=1}^{\bowtie_e} \Pr[X_i = 1] = \frac{\bowtie_e}{d_v \cdot d_u}$$

$Y_{FE} = X \cdot d_v \cdot d_u$ follows that $\mathbb{E}[Y_{FE}] = \bowtie_e$. □

Lemma 11. $\mathbb{V}\text{ar}[Y_{FE}] \leq \bowtie_e (d_v \cdot d_u)$

Proof.

$$\begin{aligned}\mathbb{V}\text{ar}[Y_{FE}] &= \mathbb{V}\text{ar}\left[(d_v \cdot d_u)\sum_{i=1}^{\bowtie_e} X_i\right] = (d_v \cdot d_u)^2\ \mathbb{V}\text{ar}\left[\sum_{i=1}^{\bowtie_e} X_i\right] \\ &= (d_v \cdot d_u)^2 \left[\sum_{i=1}^{\bowtie_e}\left[\frac{1}{d_v \cdot d_u} - \frac{1}{(d_v \cdot d_u)^2}\right] - \binom{\bowtie_e}{2}\right] \\ &\leq \ \bowtie_e \cdot d_v \cdot d_u\end{aligned}$$

□

Let Z be the average of $\alpha = 32(d_u \cdot d_v)/(\epsilon^2 \bowtie_e)$ independent instances of Y_{FE}. Using Chebyshev's inequality, we arrive that Z is an $(\epsilon, 1/32)$-estimator of $\bowtie_e$.

Fast wedge sampling: We also experimented with a faster version of WSAMP, where the number of butterflies containing a wedge was estimated using sampling. But this did not give good results, partly because the time for each iteration of WSAMP was already relatively small. For example, in `Deli`, after 5 secs, WSAMP have less than 2% error while Fast Wedge Sampling

ended up with 29% error after 10 secs. In `Web` also the error percentage of Fast Wedge Sampling is 60% higher than of WSamp.

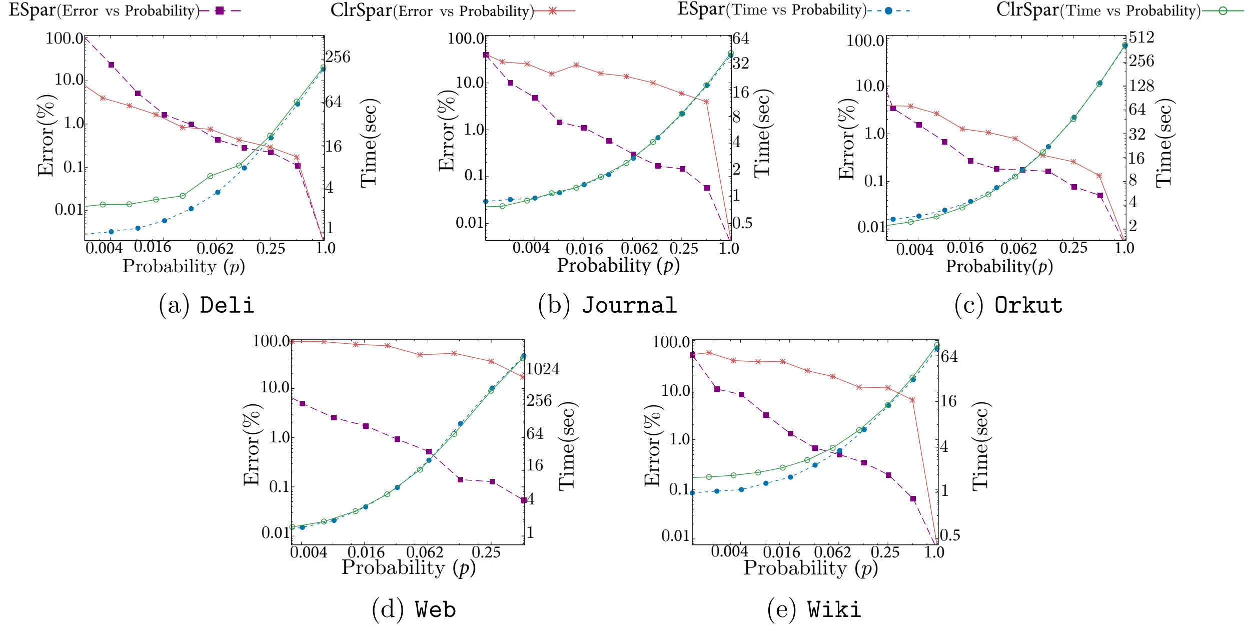
4.1.5 Comparing Sampling-based Approaches

We compare the sampling algorithms VSamp, ESamp + eBFC, ESamp + Fast-eBFC, and WSamp. A sampling algorithm immediately starts producing estimates that get better as more iterations are executed. We record the number of iterations and relative percent error of sampling algorithms up to 60 seconds. Figure 4.4 shows the relative percent error with respect to the runtime for five large bipartite graphs. We report the median error of the 30 trials of sampling methods for each data point.

Our experiments show that VSamp performs poorly when compared with ESamp and WSamp under the same time budget – this is along expected lines, based on our analysis of the variance. The accuracy of ESamp and WSamp are comparable, with ESamp being slightly better. Note that ESamp has a better variance, while WSamp has faster iterations. For instance, on the `Wiki` network, 1000 iters of WSamp yields 51% error in 0.13 secs, whereas 1000 iters of ESamp yields 29% error in 33 seconds. Across all the graphs, ESamp using Fast-eBFC , which combines the benefits of faster iterations with a good variance, yields the best results, and is superior to all other sampling methods; **it leads to less than 1% relative error within** 5 **seconds**.

4.2 Approximation by One-shot Sparsification

We present methods for estimating $\bowtie(\mathcal{G})$ using *one-shot sparsification* – where we thin down the input graph into a smaller graph through a global sampling step. The number of butterflies in the sparsified graph is used to estimate $\bowtie(\mathcal{G})$. Unlike algorithms such as ESamp and WSamp, which work on a small subgraph constructed around a single randomly sampled edge or a wedge, sparsification methods put each edge in the graph into the sample with a certain probability. We consider two approaches to sparsification (1) ESpar in Section 4.2.1, where edges are chosen



(a) `Deli` (b) `Journal` (c) `Orkut` (d) `Web` (e) `Wiki`

Figure 4.7: Accuracy (on left y-axis) and runtime performance (on right y-axis) of sparsification algorithms for different probabilities (on x-axis). ESPAR performs better than CLRSPAR, and yields $< 1\%$ relative error within 4 seconds for all networks.

independently of each other and (2) CLRSPAR in Section 4.2.2, based on a sampling method where different edges are not independently chosen, but dense regions appear with higher probability in the sample – based on a similar idea in the context of triangle counting due to Pagh and Tsourakakis [65].

4.2.1 Edge Sparsification (Algorithm ESpar)

In ESPAR, the input graph $\mathcal{G}$ is sparsified by *independently* retaining each edge in $\mathcal{G}$ in the sampled graph $\mathcal{G}'$, with a probability p. The number of butterflies in $\mathcal{G}'$, obtained using EXACTBFC, is used to construct an estimate of $\bowtie(\mathcal{G})$, after applying a scaling factor. This is much faster than working with the original graph $\mathcal{G}$ since the number of edges in $\mathcal{G}'$ can be much smaller. The ESPAR algorithm is described in Algorithm 10.

Lemma 12. *Let Y_{ES} denote the output of* ESPAR *on input graph $\mathcal{G}$. Then $\mathbb{E}[Y_{ES}] = \bowtie(\mathcal{G})$. If $p > \max\left(\sqrt[4]{\frac{24}{\bowtie}}, \sqrt{\frac{24\Delta}{\bowtie}}, \frac{24\Delta^2}{\bowtie}\right)$, then $\mathbb{V}\text{ar}[Y_{ES}] \leq \bowtie^2/8$.*

Proof. For $i = 1, \cdots, \bowtie$, let X_i be a random variable indicating the ith butterfly, s.t. X_i is 1 if the ith butterfly exists in the sampled graph $(\mathcal{V}_\mathcal{G}, \mathcal{E}')$ and 0 otherwise. Let β be as defined in the algorithm. Clearly, $\beta = \sum_{i=1}^{\bowtie} X_i$. We have $\mathbb{E}[\beta] = \sum \mathbb{E}[X_i] = \sum \Pr[X_i = 1]$. Since different edges are sampled independently, we have $\Pr[X_i = 1] = p^4$. Hence,
$\mathbb{E}[Y_{ES}] = p^{-4}\mathbb{E}[\beta] = p^{-4}\sum_{i=1}^{\bowtie} p^4 = \bowtie$.
The random variables X_is corresponding to different butterflies are not independent of each other, since butterflies may share edges. Accounting for the covariances of X_i, X_j pairs leads to:

$$\begin{aligned}\mathbb{V}\text{ar}[Y_{ES}] &= \mathbb{V}\text{ar}\left[p^{-4}\sum_{i=1}^{\bowtie} X_i\right] \\ &= p^{-8}\left[\sum_{i=1}^{\bowtie}(\mathbb{E}[X_i] - \mathbb{E}^2[X_i]) + \sum_{i\neq j}\mathbb{C}\text{ov}\left[X_i \wedge X_j\right]\right] \\ &= p^{-8}\left[\bowtie(p^4 - p^8) + \sum_{i\neq j}(\mathbb{E}[X_iX_j] - \mathbb{E}[X_i]\mathbb{E}[X_j])\right]\end{aligned}$$

Let p_{0e}, p_{1e}, and p_{1w} respectively denote the number of pairs of butterflies that share zero edges, one edge, and one wedge (two edges). Note that these three cases account for all pairs of distinct butterflies, since it is not possible for two distinct butterflies to share three or more edges. Proceeding with a calculation similar to ones in Section 4.1, we get:

$$\begin{aligned}\mathbb{V}\text{ar}[Y_{ES}] &= p^{-8}\left[\bowtie(p^4 - p^8) + p_{1e}(p^7 - p^8) + p_{1w}(p^6 - p^8)\right] \\ &\leq \bowtie p^{-4} + p_{1w}p^{-2} + p_{1e}p^{-1}\end{aligned}$$

By Chebyshev's inequality we have $\Pr[|Y_{ES} - \bowtie| \geq \epsilon \bowtie] \leq \frac{\mathbb{V}\text{ar}[Y_{ES}]}{\epsilon^2\mathbb{E}^2[Y_{ES}]}$. We need to find the probability p such that $\mathbb{V}\text{ar}[Y_{ES}] = o(\mathbb{E}^2[Y_{ES}])$ which results in $\mathbb{E}[Y_{ES}] = \bowtie$ with probability $1 - o(1)$. Thus,

Algorithm 10: ESPAR: Edge Sparsification

Input: A bipartite graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, parameter $p, 0 < p < 1$

1 Construct $\mathcal{E}'$ by including each edge $e \in \mathcal{E}$ independently with probability p
2 $\beta \leftarrow$ EXACTBFC$(\mathcal{V}, \mathcal{E}')$ `// exact global counting`
3 **return** $\beta \times p^{-4}$

Algorithm 11: ClrSpar

Input: Bipartite graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, number of colors N

1 Let $f : \mathcal{V} \to \{1, \ldots, N\}$`// map to random colors`
2 $\mathcal{E}' \leftarrow \{(u,v) \in \mathcal{E}_{\mathcal{G}} | f(u) = f(v)\}$
3 $\beta \leftarrow$ ExactBFC $(\mathcal{V}, \mathcal{E}')$ `// exact global counting`
4 **return** $\beta \cdot p^{-3}$ where $p = 1/N$

$$\mathbb{E}^2[Y_{ES}] \gg \mathbb{V}\text{ar}[Y_{ES}]$$

$$\bowtie^2 \gg \bowtie p^{-4} + p_{1w}p^{-2} + p_{1e}p^{-1} - (\bowtie + p_{1w} + p_{1e})$$

$$\bowtie^2 \gg \bowtie p^{-4} + p_{1w}p^{-2} + p_{1e}p^{-1}$$

We know that $p > \max\left(\sqrt[4]{\frac{24}{\bowtie}}, \sqrt{\frac{24\Delta}{\bowtie}}, \frac{24\Delta^2}{\bowtie}\right)$. Using Observation 1, this implies that: $p > \max\left(\sqrt[4]{\frac{24}{\bowtie}}, \frac{\sqrt{24p_{1w}}}{\bowtie}, \frac{24p_{1e}}{\bowtie^2}\right)$, we get:

$$\mathbb{V}\text{ar}\left[Y_{ES}\right] \leq \bowtie \cdot \frac{\bowtie}{24} + p_{1w} \cdot \frac{\bowtie^2}{24p_{1w}} + p_{1e} \cdot \frac{\bowtie^2}{24p_{1e}} = \frac{\bowtie^2}{8}$$

□

Observation 1. *$p_{2v} \leq \bowtie d_{\max}{}^2$, $p_{1e} \leq \bowtie d_{\max}{}^2$, and $p_{1w} \leq \bowtie d_{\max}$.*

Using Chebyshev's inequality and standard methods, the estimator Y_{ES} can be repeated $\mathcal{O}(\log(1/\delta)/\epsilon^2)$ times to get an (ϵ, δ) estimator of $\bowtie(\mathcal{G})$.

4.2.2 Colorful Sparsification (Algorithm ClrSpar)

The idea in ClrSpar is to sample edges at a rate of p, as in ESpar, but add dependencies between the sampling of different edges, such that there is a greater likelihood that a dense structure, such as the butterfly, is preserved in the sampled graph. The algorithm randomly assigns one of N colors to vertices, and sample only those edges whose endpoints have the same color. The ClrSpar algorithm for approximate butterfly counting is presented in Algorithm 11. We developed this method due to the following reason. Suppose $p = 1/N$. Though the expected number of edges in the sampled graph is mp, the same as in ESpar, it can be seen that the expected number of butterflies in the sampled graph is equal to $p^3 \bowtie$, which is higher than in the

case of ESPAR ($p^4 \bowtie$). Thus, for a sampled graph of roughly the same size, we expect to find more butterflies in the sampled graph. Note however that this does not directly imply a lower variance of the estimator due to CLRSPAR.

Lemma 13. *Let Y be the output of* CLRSPAR *on input $\mathcal{G}$, and $p = 1/N$. $\mathbb{E}[Y] = \bowtie(\mathcal{G})$. If $p > \max\left(\sqrt[3]{\frac{32}{\bowtie}}, \sqrt{\frac{32 d_{\max}}{\bowtie}}, \frac{32 {d_{\max}}^2}{\bowtie}\right)$, then $\mathbb{V}\mathrm{ar}[Y] \leq \bowtie^2 /8$.*

Proof. For each butterfly $i = 1, 2, \ldots, \bowtie(\mathcal{G})$ in $\mathcal{G}$, let X_i be a random variable such that $X_i = 1$ if the ith butterfly is monochromatic, i.e. all its vertices have been assigned the same color by the function f, and $X_i = 0$ otherwise. Clearly, the set of butterflies that are preserved in $\mathcal{E}'$ are the monochromatic butterflies. Let β be as defined in the algorithm. It follows that $\beta = \sum_{i=1}^{\bowtie} X_i$, and $Y = p^{-3} \cdot \sum_{i=1}^{\bowtie} X_i$. 4 Note that $\mathbb{E}[X_i] = \Pr[X_i = 1] = p^3$. To see this, consider the vertices of butterfly i in any order. It is only required that the colors of the second, third, and fourth vertices in the butterfly match that of the first vertex. Since vertices are assigned colors uniformly and independently, this event has probability p^3. Using linearity of expectation,
$\mathbb{E}[Y] = p^3 \sum_{i=1}^{\bowtie} \mathbb{E}[X_i] = \bowtie$.
For its variance, using the same argument in Algorithm 10, we have:

$$\mathbb{V}\mathrm{ar}[Y] = p^{-6} \left[\bowtie (p^3 - p^6) + \sum_{i \neq j} (\mathbb{E}[X_i X_j] - \mathbb{E}[X_i]\mathbb{E}[X_j]) \right]$$

For a pair of butterflies (i, j) of.

- Type $0v$ or $1v$: $\mathbb{E}[X_i X_j] - \mathbb{E}[X_i]\mathbb{E}[X_j] = p^6 - p^6 = 0$
- Type $2v$: $\mathbb{E}[X_i X_j] - \mathbb{E}[X_i]\mathbb{E}[X_j] = p^5 - p^6$
- Type $1e$: $\mathbb{E}[X_i X_j] - \mathbb{E}[X_i]\mathbb{E}[X_j] = p^5 - p^6$
- Type $1w$: $\mathbb{E}[X_i X_j] - \mathbb{E}[X_i]\mathbb{E}[X_j] = p^4 - p^6$

Therefore, we have

$$\begin{aligned}\mathbb{V}\text{ar}[Y] &= p^{-6}\left[\bowtie(p^3-p^6)+p_{2v}(p^5-p^6)\right.\\ &\left.+\,p_{1e}(p^5-p^6)+p_{1w}(p^4-p^6)\right]\\ &= \bowtie p^{-3}+p_{1w}p^{-2}+p_{1e}p^{-1}+p_{2v}p^{-1}\\ &-(\bowtie+p_{1w}+p_{1e}+p_{2v})\\ &\leq \bowtie p^{-3}+p_{1w}p^{-2}+p_{1e}p^{-1}+p_{2v}p^{-1}\end{aligned}$$

Using Observation 1, $p > \max\left(\sqrt[3]{\frac{32}{\bowtie}}, \sqrt{\frac{32d_{\max}}{\bowtie}}, \frac{32{d_{\max}}^2}{\bowtie}\right)$ implies $p > \max\left(\sqrt[3]{\frac{32}{\bowtie}}, \frac{\sqrt{32p_{1w}}}{\bowtie}, \frac{32p_{1e}}{\bowtie^2}, \frac{32p_{2v}}{\bowtie^2}\right)$. Hence,

$$\begin{aligned}\mathbb{V}\text{ar}\left[Y\right] &\leq \bowtie p^{-3}+p_{1w}p^{-2}+p_{1e}p^{-1}+p_{2v}p^{-1}\\ &\leq \bowtie\cdot\frac{\bowtie}{32}+p_{1w}\cdot\frac{\bowtie^2}{32p_{1w}}+p_{1e}\cdot\frac{\bowtie^2}{32p_{1e}}+p_{2v}\cdot\frac{\bowtie^2}{32p_{2v}}=\frac{\bowtie^2}{8}\end{aligned}$$

□

4.2.3 Comparison of Sparsification Algorithms

The parameter p controls the probability of an edge being included in the sparsified graph. As p increases, we expect the accuracy as well as the runtime to increase. The relative accuracies of the two methods depends on the variances of the estimators. We estimated the variances using our analysis, and Table 4.2 presents the results. We observe that the variance of ESpar is predicted to be much lower than that of ClrSpar. To understand this, we begin with Lemmas 12 and 13 which show expressions bounding the variances in terms of p_{1w}, p_{1e}, and p_{2v}, also summarized in Table 4.2. The difference in the variance between ClrSpar and

Table 4.2: Upper bounds on ESpar & ClrSpar variances, p=0.1.

Graph \ Algorithm	ESpar $\bowtie p^{-4}+p_{1w}p^{-2}+p_{1e}p^{-1}$	ClrSpar $\bowtie p^{-3}+p_{1w}p^{-2}+p_{1e}p^{-1}+p_{2v}p^{-1}$
Deli	4.047×10^{15}	9.163×10^{20}
Journal	1.413×10^{19}	2.042×10^{25}
Orkut	5.576×10^{19}	8.514×10^{25}
Web	7.760×10^{20}	4.692×10^{27}
Wiki	1.303×10^{20}	3.062×10^{26}

Table 4.3: Time (in seconds) to obtain 1% relative percent error for the best sampling and sparsification algorithms.

	ESAMP (with FAST-EBFC)	ESPAR
Deli	3.4	2.1
Journal	5.0	1.7
Orkut	3.4	3.4
Web	4.1	3.9
Wiki	4.8	2.3

ESPAR boils down to $(\bowtie p^{-3} + p_{2v}p^{-1} - \bowtie p^{-4})$. In our experiments, we found that p_{2v}, the number of pairs of butterflies that share two vertices without sharing an edge, was very high, much larger than p_{1e}, p_{1w}, and $\bowtie$. This explains why the variance of ESPAR was higher.
This observation about variance is consistent with our experimental results. In Figure 4.7, we report the relative percent error as well as the runtime as the sampling probability p increases. Results show that ESPAR obtains less than one percent error when 5 percent of edges are sampled. However, as shown in Figures 4.6b, 4.6d and 4.6e, CLRSPAR requires a larger sampling probability to achieve a reasonable accuracy.

4.2.4 Sampling or Sparsification?

The accuracy of the best sampling algorithm, ESAMP with FAST-EBFC, is compared with the best sparsification algorithm, ESPAR, in Table 4.3. Overall, two algorithms take similar times to reach a 1% error on all graphs we considered, in the range of 1.7-5 sec, with ESPAR achieving this accuracy faster than ESAMP with FAST-EBFC.
However, ESPAR has other downsides when compared with ESAMP. First, the memory consumption of ESPAR is $O(mp)$ where p is a parameter, and is larger than ESAMP with FAST-EBFC, whose memory consumption is $\mathcal{O}(\Delta)$. As a result, we expect the memory of ESPAR to be linearly in the size of the graph, showing that it may be easier for ESAMP to scale to graphs of even larger sizes. Next, ESPAR needs to decide on a sampling parameter p to balance between accuracy and runtime. If p is too large, then the runtime is high, and if p is too small,

then the accuracy is low. Finally, one-shot sparsification algorithms assume that the entire graph is available whereas the ESamp needs access to only a subgraph of the graph. Thus if one has to pay for data about the graph, or if the data about edges is hard to obtain, then sampling may be the better alternative.

Overall, local sampling algorithms can find a larger application space in real-world. *If the entire graph is available, and memory is not a bottleneck, it may be better to go with* ESpar*. For more restrictive scenarios,* ESamp *combined with* Fast-eBFC *is the better option.*

CHAPTER 5. BUTTERFLY COUNTING IN BIPARTITE NETWORK STREAMS

We present streaming algorithms for estimating the number of butterflies over an infinite window, i.e., all edges seen so far, for insertion-only and fully-dynamic graph streams. Our randomized algorithms [77] maintain an unbiased estimate of the number of butterflies using a bounded memory of M, and provide a trade-off between memory used and the accuracy of the estimate. Besides these, we present streaming algorithms in sliding sequence-based and time-based windows to estimate the number of butterflies in a given window.

5.1 Algorithms for Insertion-only Streams

Here, we consider space-efficient single-pass estimation of the number of butterflies from insertion-only bipartite stream such that each edge represents a connection between entities in two different partitions. In insertion-only models, edges of a stream are *only* added to the stream at a time. The estimators hold an unbiased estimation of butterfly count at any moment of the stream while they require a limited memory budget, which is considerably smaller than the size of the entire stream.

5.1.1 Adaptive Sampling: Fleet1

We use random sampling from the stream of edges. An initial attempt uses Bernoulli Sampling (Bern) with parameter $p, 0 < p \leq 1$. Each arriving edge is sampled into a reservoir with probability p. The number of butterflies among the sampled edges is incrementally maintained, and is multiplied by the appropriate normalization factor, to estimate the number of butterflies in the stream. The disadvantage of Bern is that it requires setting parameter p to the "right

value", which depends on the input itself. If p is too small, the error in estimation can be large, and if p is too large, then the reservoir size can be very large.

Our first algorithm, FLEET1, solves this problem by adaptively setting p throughout the computation, so as to keep the memory bounded by M. Initially, p is set to 1, and all edges are sampled into the reservoir. When the size of the reservoir exceeds M, FLEET1 sub-samples by retaining each edge in the current reservoir with a probability γ. Further edges are sampled with probability γ. When the size of the reservoir again exceeds M, the same process is repeated, and further edges are sampled with probability γ^2, and so on. The size of the reservoir never exceeds M edges, and is at least γM, with high probability, except during the initial stages of the stream. FLEET1 also continuously maintains the number of butterflies among sampled edges. When an estimate is desired, the number of butterflies among the sampled edges is returned, after multiplying by the appropriate normalization factor.

Details are presented in Algorithm 12. Each time the reservoir is sub-sampled, FLEET1 uses an (exact) algorithm to compute $\xi(\mathcal{R})$, the number of butterflies in the reservoir using prior methods designed for a static graph, such as[76, 102]. FLEET1 also uses an algorithm $\mathrm{EBFC}(e, E)$ to count the number of butterflies that contain edge e in the graph induced by edge set E. This can be achieved using prior work such as [76]. For the purposes of the current discussion, the reader can assume that parameter γ is set to $1/2$ – the main advantages of our algorithm, including bounded sample size and provable accuracy still hold. A modest tradeoff between accuracy and runtime can be obtained by setting γ to other values between $1/2$ and 1, as we discuss further in Section 5.4.3. Lemma 14 shows that FLEET1 maintains an unbiased estimate of the butterfly count after observing each edge. Let Y^t_{FLEET1} denote the estimate of ξ^t returned by Algorithm 12 (Line 12) at time t.

Lemma 14. $E[Y^t_{\text{FLEET1}}] = \xi^t$

Proof. Suppose the butterflies in $\bowtie^t$ are numbered from 1 to ξ^t. Let $X^t_i (1 \leq i \leq \xi^t)$ be a random variable equal to 1 if all edges of the ith butterfly appear in $\mathcal{R}$, and 0 otherwise. Let $X^t = \sum_{i=1}^{\xi^t} X^t_i$. From Line 11 of Algorithm 12, we have $Y^t_{\text{FLEET1}} = X^t \left(p^t\right)^{-4}$. When $t \leq M$, all

Algorithm 12: FLEET1 $(\mathcal{S}, \mathrm{M})$: Adaptive sampling

Input: Edge stream $\mathcal{S}$, max. reservoir size M, resampling parameter γ (default value of $\gamma = 0.5$)
Output: Estimate of ξ^t, the number of butterflies at t

```
1  p ← 1, R ← ∅, t ← 0, β ← 0
2  for each edge e in S do
3      t ← t + 1
4      while |R| ≥ M do
5          p ← γp
6          for each edge e ∈ R do
7              Keep e in R with prob. γ and discard with prob. 1 − γ
8          β ← p^{-4} × ξ(R) // # of butterflies in R
9      if coin (p) is Head then
10         R ← R ∪ {e}
11         β ← β + p^{-4} × eBFC(e, R)
12     Y^t_FLEET1 ← β
```

edges of the stream are sampled, and $X^t = \xi^t$. When $t > M$, each edge appears in $\mathcal{R}$ with probability p^t. Note that p^t itself is a random variable equal to the sampling probability of an incoming edge at time t. To compute $\mathbb{E}[Y^t_{\text{FLEET1}}]$, we first condition on p^t and then remove the conditioning.

Since there are four edges in a butterfly, $\mathbb{E}\left[X_i^t \mid p^t\right] = \left(p^t\right)^4$. With linearity of expectation, $\mathbb{E}\left[X^t \mid p^t\right] = \sum_{i=1}^{\xi^t} \mathbb{E}\left[X_i^t \mid p^t\right] = \xi^t \left(p^t\right)^4$. Then, $\mathbb{E}\left[Y^t_{\text{FLEET1}} \mid p^t\right] = \mathbb{E}\left[X^t \left(p^t\right)^{-4} \mid p^t\right] = \xi^t$. By conditional expectation, $\mathbb{E}\left[Y^t_{\text{FLEET1}}\right] = \mathbb{E}\left[\mathbb{E}\left[Y^t_{\text{FLEET1}} \mid p^t\right]\right] = \xi^t$. □

5.1.2 Concentration analysis of $Y^t_{\textbf{Fleet1}}$

We next show that if the upper bound on the reservoir size (M) is large enough, then the estimate Y^t_{FLEET1} will be concentrated around its expectation, i.e., have a small relative error, with a high probability. There are a few complexities to deal with here. First, the sampling probability p^t is itself a random variable, since it is the result a random process. We first analyze a simpler algorithm BERN(p) that is based on fixed sampling probability p, which we have explained earlier. After t edges, let Z_p^t denote the estimate computed by BERN(p), of $\xi(\mathcal{G}^t)$, the number of butterflies in $\mathcal{G}^t$. Lemma 15 shows a concentration bound for Z_p^t. The difficulty with analyzing Z_p^t is in handling the dependency between random variables corresponding to different butterflies

being sampled into the reservoir. Though different edges are sampled independently by Bern(p), different butterflies are not necessarily independent since butterflies may share edges. We address this with the help of the Hajnal-Szemerédi theorem, building on ideas from prior works [27, 49, 65] all of who applied this idea in the context of triangle counting.

Lemma 15. *After t edges, let h^t denote the maximum number of butterflies in $\mathcal{G}^t$ that an edge can be part of. For a fixed p, and for any $\epsilon \in (0,1)$, we have* $\Pr\left[\left|Z_p^t - \xi^t\right| \geq \epsilon\xi^t\right] \leq 8h^t \cdot exp\left(-\frac{\epsilon^2\xi^t p^4}{12h^t}\right)$

Proof. Consider a new graph $H^t = (\mathcal{V}_{H^t}, \mathcal{E}_{H^t})$ constructed from $\mathcal{G}^t$: each vertex $v \in \mathcal{V}_{H^t}$ corresponds to a butterfly in $\mathcal{G}^t$. For each pair of butterflies $u, v \in \mathcal{V}_{H^t}$ that share at least one edge, there is an edge in $\mathcal{E}_{H^t}$. Since each edge in $\mathcal{G}^t$ can be shared by at most h^t butterflies, the maximum degree of vertex $u \in \mathcal{V}_{H^t}$ is $(4h^t - 4)$. Using the Hajnal-Szemerédi theorem [37], there exists an equitable coloring of H^t with at most $(4h^t - 3)$ colors. Let the colors be numbered as $\{1, 2, \ldots, 4h^t - 3\}$. For each butterfly $i \in \mathcal{V}_{H^t}$, let random variable X_i be defined as: $X_i = 1$ if all edges of butterfly i are sampled by Bern at time t, and 0 otherwise. Let the set of butterflies assigned color j be denoted by C_j. For two butterflies $a, b \in C_j$, note that X_a and X_b are independent, since a and b do not share edges, and all their edges are sampled independently. Define $Y_j = \sum_{i \in C_j} X_i$. Note that Y_j is a Binomial random variable since it is the sum of independent 0-1 random variables. We have $\mathbb{E}[Y_j] = |C_j| \cdot p^4$ and $|C_j| > \xi^t/4h^t$, since the coloring of vertices in $\mathcal{V}_{H^t}$ is an equitable coloring. Using the Chernoff bound,
$\Pr\left[\left|Y_j - |C_j| \cdot p^4\right| \geq \epsilon \cdot |C_j| \cdot p^4\right] \leq 2\exp\left(-\frac{\epsilon^2}{3} \cdot \frac{\xi^t}{4h^t} \cdot p^4\right)$
Using the union bound,
$\Pr\left[|Z_p^t - \xi^t| \geq \epsilon\xi^t\right] = \Pr\left[\left|\sum_{j=1}^{4h^t-3} Y_j \cdot p^{-4} - \sum_{j=1}^{4h^t-3} |C_j|\right| \geq \epsilon \sum_{j=1}^{4h^t-3} |C_j|\right]$
$\leq \sum_{j=1}^{4h^t-3} \Pr\left[|Y_j \cdot p^{-4} - |C_j|| \geq \epsilon|C_j|)\right] \leq 8h^t \cdot exp\left(-\frac{\epsilon^2\xi^t p^4}{12h^t}\right)$ □

We next derive a concentration result for Fleet1. Fleet1 essentially returns the estimate due to Bern(p^t) where p^t is itself a random variable. While it is possible to compute the expected value of p^t, this cannot be directly plugged into the Lemma 15. Lemma 16 shows that for a large enough reservoir size, Fleet1 returns an estimate that is concentrated around its expectation.

The proof considers multiple $\textsc{Bern}(p)$ instances, one for each sampling probability and combines with the event of $\textsc{Fleet1}$ stopping at one of these levels.

Lemma 16. *Assume $\gamma \le 0.9$. For any $\epsilon, \delta \in (0,1)$, when M satisfies $M \ge 6t \cdot \sqrt[4]{\frac{12h^t}{\epsilon^2\xi^t} \cdot \ln\frac{8h^t(4+\delta)}{\delta}}$ then $\Pr\left[\left|Y^t_{\textsc{Fleet1}} - \xi^t\right| \ge \epsilon\xi^t\right] \le \delta$.*

Proof. Note that the sampling rate of $\textsc{Fleet1}$ is γ^i for $i \ge 0$. We say that $\textsc{Fleet1}$ is at level i when the sampling rate is γ^i. For $i \in [0, +\infty)$, define events S_i and B_i as follows. Event B_i: Suppose that we execute algorithm $\textsc{Bern}(\gamma^i)$, and we have $\left|Z^t_{\gamma^i} - \xi^t\right| \ge \epsilon\xi^t$. S_i is the event that $\textsc{Fleet1}$ stops at level i.

Define event B: $\left|Y^t_{\textsc{Fleet1}} - \xi^t\right| \ge \epsilon\xi^t$. We decompose the probability of event B in terms of B_i and S_i.

$$\begin{aligned}\Pr[B] &= \sum_{i=0}^{\infty}\Pr[B_i \wedge S_i] = \sum_{i=0}^{\ell}\Pr[B_i \wedge S_i] + \sum_{i=\ell+1}^{\infty}\Pr[B_i \wedge S_i] \\ &\le \sum_{i=0}^{\ell}\Pr[B_i] + \sum_{i=\ell+1}^{\infty}\Pr[S_i]\end{aligned}$$

where $\ell = \frac{\log(M/6t)}{\log\gamma}$. The sampling probability at level i is γ^i, by Lemma 15 we have $\sum_{i=0}^{\ell}\Pr[B_i] = \sum_{i=0}^{\ell} 8h^t \cdot exp\left(-\frac{\epsilon^2\xi^t\gamma^{4i}}{12h^t}\right) = \sum_{i=0}^{\ell}\alpha^{\left(\gamma^{-4i}\right)}$ where $\alpha = 8h^t \cdot exp\left(-\frac{\epsilon^2\xi^t\gamma^{4\ell}}{12h^t}\right)$ and $\alpha < 1$. When $\gamma \le 0.9$, $\gamma^{-4} > e^{1/e}$, we have $\gamma^{-4i} \ge i$ for any $i \ge 1$. Applying this fact, and using the bound on M we get: $\sum_{i=0}^{\ell}\alpha^{\left(\gamma^{-4i}\right)} \le \alpha + \sum_{i=1}^{\ell}\alpha^i \le \frac{2\alpha}{1-\alpha} \le \frac{\delta}{2}$

Let X_ℓ denote the number of edges in $\mathcal{R}$ when $\textsc{Fleet1}$ is at level ℓ. Note that the event $\textsc{Fleet1}$ stops at level higher than ℓ is equivalent to the event that X_ℓ is greater than the reservoir size M. By $\mathbb{E}[X_\ell] = t \cdot \gamma^\ell$ and Chernoff bound, we have

$\sum_{i=\ell+1}^{\infty}\Pr[S_i] = \Pr[X_\ell > M] \le \Pr[X_\ell > 6 \cdot \mathbb{E}[X_\ell]] \le \frac{\delta}{2}$. Combining the above bounds, we arrive at $\Pr[B] \le \delta$. □

5.1.3 Improved Adaptive: Fleet2 and Fleet3

We present two algorithms $\textsc{Fleet2}$ and $\textsc{Fleet3}$ which improve upon $\textsc{Fleet1}$, providing a better memory-accuracy tradeoff. $\textsc{Fleet2}$ is similar to $\textsc{Fleet1}$, but handles sub-sampling differently. Say $\textsc{Fleet1}$ is at "level i" when its sampling probability is γ^i. In $\textsc{Fleet1}$, each time the level

changes from i to $(i+1)$, edges are discarded according to a random process, and the number of butterflies is recomputed on the new reservoir from scratch (Line 8 of Algorithm 12, shown in pink color). Due to this re-computation, butterflies that were already detected at the higher sampling rate (level i) may no longer have all their edges present at the lower sampling rate (level $(i+1)$). In contrast, FLEET2 does not recompute when the reservoir is sub-sampled. Instead, the current butterfly count at level i is maintained, and as more butterflies are detected at level $i+1$, they are accumulated into this estimate. It can be expected that FLEET2 obtains a better accuracy than FLEET1, since it "catches" more butterflies than FLEET1. It is easy to see that the estimation of FLEET2 is unbiased. In addition to better accuracy, FLEET2 is also slightly faster than FLEET1, since it avoids recomputation of butterflies at the sub-sampling step.

FLEET3 (described in Algorithm 13) improves accuracy over FLEET1 and FLEET2 by handling new edges differently. This idea is inspired by Algorithm MASCOT of [55], which used the idea in the context of counting triangles from a graph stream (the same idea is also used in [92]). Upon receiving a new edge, the estimate is updated by accounting for butterflies that are created by the new edge (and existing sampled edges), even before deciding whether or not to sample the new edge (see Line 1). In other words, FLEET3 first updates the estimate and then samples. If the current edge sampling probability is p, then the probability of detecting a butterfly involving the new edge increases from p^4 (in FLEET1) to p^3 (in FLEET3). This helps increase the accuracy of butterfly counting while using the same memory. Algorithm 13 has further details. We omit further details and proofs, due to space constraints.

Comments: Instead of Bernoulli sampling, we could also use reservoir sampling of edges as the basis. We took the route of Bernoulli sampling to simplify the analysis, since it leads to edges being sampled independent of each other (given a sampling probability), unlike reservoir sampling, where the sampling of edges are not independent events. Our initial implementations of algorithms based on reservoir sampling without replacement, showed that the accuracy-memory tradeoffs were similar to that of our current algorithms. We can also achieve estimates of per-vertex butterfly counts (the number of butterflies that each vertex is a part of) using a similar

Algorithm 13: FLEET3 ($\mathcal{S}$, M): Adaptive sampling

Input: Edge stream $\mathcal{S}$, max. reservoir size M, resampling parameter γ (default value is $\gamma = 0.5$)
Output: Estimate of ξ^t, the number of butterflies at t

```
1  p ← 1, R ← ∅, t ← 0, β ← 0
2  for each edge e in S do
3      t ← t + 1
4      β ← β + p^{-3} × EBFC(e, R)
5      while |R| ≥ M do
6          p ← γp
7          for each edge e ∈ R do
8              Keep e in R with prob. γ and discard with prob. 1 − γ
9      if coin (p) is Head then R ← R ∪ {e}
10     Y^t_FLEET3 ← β
```

sampling approach of estimating per-vertex subgraph counts from the reservoir, and maintaining additional state for each vertex. A detailed study of local butterfly counting is a goal for future work.

5.2 Sliding Window Streaming

In this section, we consider butterfly counting in two types of sliding window models: sequence-based and time-based. We assume the number of edges in a window is (much) greater than the available memory M – otherwise, the entire window can be stored within memory and an exact algorithm can be applied.

5.2.1 Sequence-based Window (Algorithm FleetSSW)

We first present FLEETSSW for sequence-based sliding window (Algorithm 14), which is based on maintaining a sample of edges from within the sliding window. In the initial stages of observation, all edges fit in memory, and as more edges are observed, we recursively decrease the sampling probability as in FLEET1, to ensure that the sample fits in memory. However, once the number of edges in a window reaches W, it will stay at W henceforth. As a result, when the edge sampling probability p reaches M/W, the algorithm does not decrease p any further holds it at M/W[1]. The algorithm stores only active edges in the sample, i.e., any item (e, t') such that t' is not

[1] For simplicity of exposition, we assume M/W is a power of γ.

within the current window is discarded. Let Y^t_{sw} denote an estimate returned by Algorithm 14,

Algorithm 14: FLEETSSW ($\mathcal{S}$,M,W): Seq-based SW

Input: Edge stream $\mathcal{S}$, max. reservoir size M, window size W ($\gg M$)
Output: Estimate of ξ^t_W, the number of butterflies in window at t

```
1  p ← 1, R ← ∅, t ← 0, β ← 0
2  for each edge e in S do
3      t ← t + 1
4      if p > (M/W) then Run FLEET1 (S, M) and update p, β, R
5      else
6          p ← (M/W)
7          if coin (p) is Head then
8              R ← R ∪ {e, t}, and β ← β + p^{-4} × eBFC(e, R)
       /* Delete expired edges and update estimate */
9      if (e', t') ∈ R s.t. t' ≤ (t − W) then
10         β ← β − p^{-4} × eBFC(e', R), and R ← R \ (e', t')
11     Y^t_sw ← β
```

of ξ^t_W, the number of butterflies in the window at time t.

Lemma 17. *The space of $\mathcal{R}$ in Algorithm 14 is no greater than M in expectation and $\Pr(|\mathcal{R}| \geq 2M) \leq \left(\frac{e}{4}\right)^M$. Y^t_{sw} is an unbiased estimate of ξ^t_W. For parameters $0 < \epsilon < 1$ and $0 < \delta < 1$, when $M \geq 6W \cdot \sqrt[4]{\frac{12h^t}{\epsilon^2 \xi^t_W} \cdot \ln \frac{8h^t(4+\delta)}{\delta}}$, then $\Pr\left[\left|Y^t_{\text{sw}} - \xi^t_W\right| \geq \epsilon \xi^t_W\right] \leq \delta$.*

Proof. We sketch the proof ideas and omit details, due to space constraints. For the space complexity, note that when $p > M/W$, the algorithm runs FLEET1 and its space is strictly bounded by M. Otherwise, $p = M/W$ and the number of edges in the sample is a binomial random variable with parameters W and M/W, and the space bounds follow using Chernoff bounds.

At any given time, each edge currently in the window is sampled into $\mathcal{R}$ with probability p, and $\mathbb{E}\left[Y^t_{\text{sw}}\right] = \xi^t_W \cdot p^4$.When $p > (M/W)$, we apply results from FLEET1 for expectation (Lemma 14) and concentration (Lemma 16) to show the corresponding properties of Y^t_{sw}. When $p = (M/W)$, we rely on an analysis similar to Algorithm BERN in Lemma 15 and derive the concentration result that $\Pr\left[\left|Y^t_{\text{sw}} - \xi^t_W\right| \geq \epsilon \xi^t_W\right] \leq 8h^t \cdot exp\left(-\frac{\epsilon^2 \xi^t_W}{12h^t} \cdot \left(\frac{M}{W}\right)^4\right) \leq \delta$. □

5.2.2 Time-based Sliding Window (Algorithm FleetTSW)

We next consider the case of a time-based sliding window, where each element has an associated timestamp, and the window at time t consists of all elements with timestamps greater than $(t-W)$, where W is a specified window size. Handling a time-based sliding window is more challenging than a sequence-based window since the number of elements within a time-based window can grow and shrink with time. The sequence-based window can be seen as a special case of time-based window such that at each time, exactly one edge arrives in the stream.
FleetTSW (Algorithm 15), our algorithm for time-based sliding window, can estimate the number of butterflies when the window size W is provided at query time. We assume an upper bound n_{max} number of edges within a window. Let $T = \lceil 1 + \log_\gamma \frac{M}{n_{max}} \rceil$. FleetTSW is based on maintaining not only a single sample, as in FleetSSW or Fleet1, but T reservoirs $\mathcal{R}_i, i = 0, 1, 2, \ldots$, at different sampling rates. Every edge is sampled into $\mathcal{R}_0$. For $i > 0$, each edge that was sampled into $\mathcal{R}_{i-1}$ is sampled into $\mathcal{R}_i$ with probability γ. Each reservoir has a capacity of $M' = M/T$ edges, and contains the most recent edges sampled into the reservoir. Each $\mathcal{R}_i$ is stored as a first-in-first-out queue, so that if a new edge enters when the queue is full, the edge with the earliest timestamp is deleted.

Lemma 18. *Y^t_{tw} is an unbiased estimate of ξ^t_W. If $M \in \Omega\left(\log_\gamma \frac{M}{n_{\max}} \sqrt[4]{\frac{n^4_{\max} h^t}{\epsilon^2 \xi^t_W}} \ln \frac{h^t}{\delta}\right)$, then we obtain* $\Pr\left[|Y^t_{\text{tw}} - \xi^t_W| \geq \epsilon \xi^t_W\right] \leq \delta$.

Proof. At query time t, when presented with a window size W, let $\mathcal{G}^t_W$ denote the graph consisting of all edges that have timestamps in $[t-W+1, t]$. For level $\ell \in \{0, 1, 2, \ldots\}$, let $\mathcal{G}^t_W(\ell)$ be defined inductively as follows. $\mathcal{G}^t_W(0) = \mathcal{G}^t_W$. For $i \geq 0$, $\mathcal{G}^t_W(i+1)$ is derived from $\mathcal{G}^t_W(i)$ by choosing each edge in $\mathcal{G}^t_W(i)$ with probability γ. We note that in Algorithm 15, $\mathcal{R}_i$ contains the M/T most recent elements from $\mathcal{G}^t_W(i)$. Further, when a query arrives at time t, the algorithm uses $\mathcal{R}_{\ell_*}$ such that ℓ_* is the smallest value, where $\mathcal{G}^t_W(\ell_*)$ is completely contained in $\mathcal{R}_{\ell_*}$.
Let $Z^t_W(i)$ denote the estimate of $\xi(\mathcal{G}^t_W)$ derived from $\mathcal{R}_i$. Similar to the proof of Lemma 16, we define: event S_i is the event that the algorithm chooses $\mathcal{R}_i$ to answer the sliding window query, and B_i is the event that the estimate $Z^t_W(i)$ has a relative error that is greater than ϵ. The

Algorithm 15: FLEETTSW ($\mathcal{S}$, M, n_{max}): Time-based SW

Input: Edge stream $\mathcal{S}$, reservoir size M, $n_{max}(\gg M)$ (the maximum number of elements within a window)

Output: Estimate of $\xi(\mathcal{G}_W^t)$

```
1  T ← ⌈1 + log_γ (M / n_max)⌉
   // d_ℓ is the time of most recent discarded edge in R_ℓ
2  ∀i ≤ T, R_i ← ∅, d_i ← 0
3  for each edge e in S at time t do
4      ℓ ← 0
5      do
6          R_ℓ ← R_ℓ ∪ {(e, t)}
7          if |R_ℓ| > M/T then
8              R_ℓ ← R_ℓ \ {(e*, t*)} s.t. t* = min{t'| (e', t') ∈ R_ℓ}
9              d_ℓ ← t*
10         ℓ ← ℓ + 1
11     while coin (γ) is Head
12 ℓ_* ← arg min_{ℓ∈[T]} {d_ℓ | d_ℓ ≤ (c − W)}
13 A ← {(e, t') ∈ R_{ℓ_*}} s.t. t' > (c − W) // sample of window
14 Y^t_tw ← γ^{−4ℓ_*} × ξ(A)
```

probability that the algorithm fails to return an estimate that has a relative error within ϵ is given by $\Pr[B] = \sum_{i=0}^{\infty} \Pr[S_i \wedge B_i]$. Using an argument similar to the proof of Lemma 16, we arrive at the result. □

5.3 Algorithms for Fully-dynamic streams

A more general problem is counting butterflies in a fully dynamic stream where both edge insertion and deletion may occur. While the mainstream methods are designed for graphlet counting in insertion-only streams, there are only few counting methods for handling edge deletions in a stream. It is clear that counting graphlets in a fully-dynamic steam is more challenging because it supports more operations than an insertion-only setting, and, therefore, it is of high interest in research/industry communities with numerous real-world applications. Note that the classical solutions for insertion-only setting, including reservoir sampling technique, cannot be directly applied as deletion is not supported which makes the estimations unbiased. This is because reservoir sampling will result in non-uniform sample and inaccurate estimations when deletions occur.

In this section, we introduce estimators for counting butterfly in fully-dynamic streams. The goal, here, is to maintain an unbiased estimation of butterfly count at any moment while using a fixed-size reservoirs over the entire graph stream which is in the presence of both edge insertions and deletions.

5.3.1 Algorithm ThinkD-ACC

We describe algorithm ThinkD-ACC which is a modified version of [89] for triangle counting in fully-dynamic streams. This method uses random pairing sampling (henceforth RP), due to Gemulla et al. [31]. RP is an extension of the reservoir sampling technique. It maintains two integer variables c_b and c_g to represent "bad" and "good" uncompensated edge deletions. As shown in [31], RP keeps the uniformity of sampled edges over the entire stream. We describe more details of RP in the description of algorithm ThinkD-ACC. Here, we introduce a

Algorithm 16: ThinkD-ACC

Input: Fully-dynamic edge stream $\mathcal{S}$, an integer M.
Output: Estimate of $\bowtie^t$, i.e. # of butterflies at t.

```
1  α ← 0, t ← 0, R ← ∅, c_g ← 0, c_b ← 0
2  for each item (δ, e) in S, s.t. δ ∈ {+, −} do
3      B ← eBFC(e, G)
4      p^t = ∏_{i=0}^{2} (y−i)/(|S|+c_g+c_b−i)
5      if δ is − then
6          α ← α − B/p^t
7          if edge e ∈ S then  R ← R \ {e}, c_b ← c_b + 1
8          else c_g ← c_g + 1
9      else if δ is + then
10         α ← α + B/p^t
11         if c_b + c_g = 0 then
12             if t ≤ M then R ← R ∪ {e}
13             else if coin(M/t) is Head then
14                 e′ ← a random edge from R
15                 R ← (R \ {e′}) ∪ {e}
16         else if coin(c_b/(c_g + c_b)) is Head then
17             R ← R ∪ {e}, c_b ← c_b − 1
18         else c_g ← c_g − 1
19     return α
```

butterfly counting method for fully-dynamic streans which is based on a triangle counting

algorithm and is first introduced by Shin et al. [89], and itself is inspired by TRIÈST-FD [27], the fully-dynamic work of Stefani et al. Shin et al. [89] use a technique such that each arrival edge from the stream contributes to the estimation. It is worth noting that the technique is first introduced by Lim and Kang [55]. Here, we describe THINKD-ACC, a butterfly counting version of the work of Shin et al. [89]. Recall that THINKD-ACC uses RP for sampling edges to keep a uniform sample over the stream. First, we compute the probability of each observed butterfly in reservoir $\mathcal{R}$ with size M at time t using Lemma 19.

Lemma 19. *Let the size of stream be* $|\mathcal{S}^t|$[2] *at time* t *and reservoir* $\mathcal{R}$ *with size* M *is used. Let* $y = \min\{M, |\mathcal{S}^t| + c_g + c_b\}$. *Upon receiving edge* e *at time* t *from the stream, the probability that a butterfly, which includes edge* e *in reservoir* $\mathcal{R}$*, equals*

$$p^t = \prod_{i=0}^{2} \frac{y - i}{|\mathcal{S}| + c_g + c_b - i}$$

Proof. Suppose $\{u, v, w, x\}$ be a newly formed butterfly when a new edge $\{u, v\}$ is received from the stream at time instant t . Then, the probability that this butterfly is sampled is,

$$\begin{aligned}
&\Pr\left[\{u,x\} \in \mathcal{S}^t \cap \{v,w\} \in \mathcal{S}^t \cap \{w,x\} \in \mathcal{S}^t\right] \\
&\qquad = \Pr\left[\{w,x\} \in \mathcal{S}^t | \{u,x\} \in \mathcal{S}^t \cap \{v,w\} \in \mathcal{S}^t\right] \times \\
&\qquad\qquad \Pr\left[\{u,x\} \in \mathcal{S}^t \cap \{v,w\} \in \mathcal{S}^t\right]
\end{aligned}$$

Using Lemma 7 of [88] we obtain,

$$\begin{aligned}
&\Pr\left[\{w,x\} \in \mathcal{S}^t | \{u,x\} \in \mathcal{S}^t \cap \{v,w\} \in \mathcal{S}^t\right] \\
&= \Pr\left[\{w,x\} \in \mathcal{S}^t \setminus (\{u,x\} \cup \{v,w\})\right] \\
&= \frac{y^{(t)} - 2}{\mathcal{S}^{(t)} + c_g^{(t)} + c_b^{(t)} - 2}
\end{aligned}$$

And, using Lemma 2 of [89] we obtain,

$$\Pr\left[\{u,x\} \in \mathcal{S}^t \cap \{v,w\} \in \mathcal{S}^t\right] = \frac{y^{(t)}}{\mathcal{S}^{(t)} + c_g^{(t)} + c_b^{(t)}} \times \frac{y^{(t)} - 1}{\mathcal{S}^{(t)} + c_g^{(t)} + c_b^{(t)} - 1}$$

Thus, the result follows. ∎

[2]Note that since the stream is a fully-dynamic, the size of stream is not necessarily t.

Next, we show that ThinkD-ACC gives an unbiased estimates of the global butterfly count at any given point of time. We use the following results in our proof:

Lemma 20. *The count of butterflies $|\mathcal{B}^t|$ in the bipartite graph at time t (i.e., $|\mathcal{B}^t| = \xi^t$) equals to the count of added butterflies $|\mathcal{X}^t|$ subtracted by the count of deleted butterflies $|\mathcal{Y}^t|$. Formally,*

$$|\mathcal{B}^{(t)}| = |\mathcal{X}^{(t)}| - |\mathcal{Y}^{(t)}| \quad \forall t \geq 1.$$

Now, we can state the following:

Theorem 1. ThinkD-ACC *gives an unbiased estimates at any time t. Formally,*

$$\mathbb{E}\left[\alpha^{(t)}\right] = \xi^t \quad \forall t \geq 1.$$

Proof. Assume that when an edge $e^{(t)} = \{u, v\}$ from the stream was added to the bipartite stream at time t. Also, assume that $\gamma^t_{uv} = +B/p^t$ is a random variable for the amount of change (increments) occurs, where B and p^t are computed in Line 3 and Line 4 of Algorithm 16, respectively. Similarly, if the edge $e^{(t)}$ in the stream is deleted from the graph, and assume that $\gamma^{t-}_{uv} = -B/p^t$ be a random variable for the amount of change (decrements) occurs in α. By Lemma 19, we showed that $\Pr\left[\{u, x\} \in \mathcal{S}^t \cap \{v, w\} \in \mathcal{S}^t \cap \{w, x\} \in \mathcal{S}^t\right] = p^t$. This shows that at any time t, $\mathbb{E}\left[\alpha^t\right] = \mathbb{E}\left[\gamma^t_{uvwx}\right] - \mathbb{E}\left[\gamma^{t-}_{uvwx}\right] = |\mathcal{X}^{(t)}| - |\mathcal{Y}^{(t)}| = |\mathcal{B}^{(t)}| = \xi^t$ considering that the probability of each newly formed/deleted butterfly is p^t. □

Algorithm 16 describes ThinkD-ACC. Line 3 computes the number of butterflies upon receiving edge e. This number contributes to the estimation regardless the fact that e is sampled in reservoir $\mathcal{R}$. In addition, p^t is computed and used as a scale-up factor in Lines 6 and 10 to maintain an unbiased estimation of butterfly count over the stream. The rest of the algorithm uses technique RP to ensure that reservoir $\mathcal{R}$ keeps uniformity while edges are sampled from the stream. Two variables c_b and c_g are used to record the number of "uncompensated" edges. Uncompensated edges are simply the difference between the number of cumulative insertions and deletions. As shown in [31], RP ensures that the sampled edges in the reservoir are sampled uniformly even though edges could be removed from the stream and possibly from the reservoir.

Algorithm 17: FLEET-FD ($\mathcal{S}$, M): Fleet for fully-dynamic streams

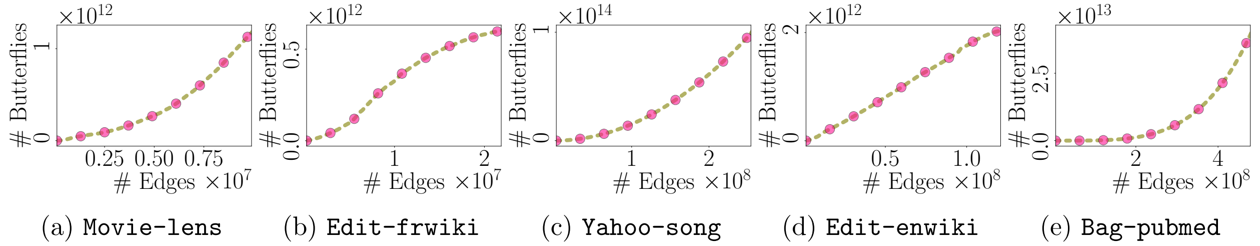
Input: Fully-dynamic edge stream $\mathcal{S}$, max. reservoir size M, resampling parameter γ (default value is $\gamma = 0.5$)

Output: Estimate of ξ^t, the number of butterflies at t

```
1  p ← 1, R ← ∅, t ← 0, β ← 0
2  for each item (δ, e) in S, s.t. δ ∈ {+, −} do
3      t ← t + 1
4      if δ is − then
5          β ← β − p^{-3} × eBFC(e, R)
6          R ← R \ {e}
7      else
8          β ← β + p^{-3} × eBFC(e, R)
9          while |R| ≥ M do
10             p ← γp
11             for each edge e ∈ R do
12                 Keep e in R with prob. γ and discard with prob. 1 − γ
13         if coin (p) is Head then  R ← R ∪ {e}
14     Y^t_FLEET3 ← β
```

5.3.2 Algorithm Fleet-FD

Here, we introduce FLEET-FD for counting butterflies in a fully dynamic graph stream. This algorithm is basically the modified version of algorithm FLEET3. Recall that FLEET3 processes edges of an insertion-only stream and guarantees the uniformity by sampling each edge from the stream with probability p^t at any time t. Because at any time the sampling probability is fixed, we can simply update the estimation even if an edge deletion occurs. Algorithm 17 represents FLEET-FD. When a deletion occurs, the estimation is updated and the edge is simply removed from reservoir $\mathcal{R}$ (Line 4). Note that this operation does not change the sampling probability of remaining edges in the reservoir. On the other hand, if an edge insertion occurs, the rest of the procedure is similar to FLEET3 – edges are sampled with the sampling probability, and whenever the reservoir is full, a downsampling is performed to keep the size of reservoir below the given threshold. As a result, at any time t, the probability of each sampled edge in the reservoir is p^t, and we obtain an unbiased estimation through the scale-up factor, i.e., $\left(p^t\right)^{-3}$.



(a) `Movie-lens` (b) `Edit-frwiki` (c) `Yahoo-song` (d) `Edit-enwiki` (e) `Bag-pubmed`

Figure 5.2: Number of butterflies as a function of stream size.

5.4 Experimental Evaluation

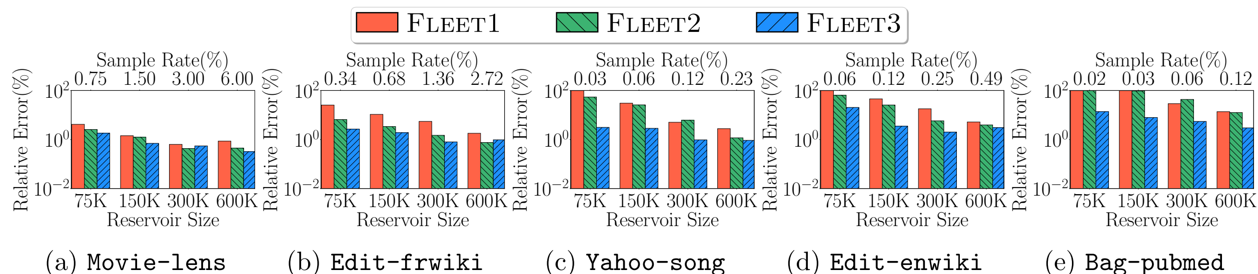
We experimentally evaluate the infinite window and sliding window algorithms on real-world temporal bipartite networks with hundreds of millions of edges from a variety of domains, such as social, web, and rating networks.

Bipartite networks and experimental setup: We used five real-world temporal bipartite networks from the publicly available KONECT repository [51][3], summarized in Table 5.1. `Movie-lens` is the ratings by users for movies. `Edit-frwiki` is a bipartite network of editors and pages of the French Wikipedia where each edge represents an edit. `Edit-enwiki` is the English version of `Edit-frwiki`. `Yahoo-song` is a ratings by users for songs. `Bag-pubmed` is a word-document bipartite network. Note that `Bag-pubmed` is not a temporal network, and we generated a stream by randomly permuting the edge set of `Bag-pubmed`. `Edit-frwiki`, `Edit-enwiki`, and `Bag-pubmed` had multiple edges between the same node pairs, and we only considered the first interaction. Edges are read from the stream in the order of timestamps. Figure 5.2 shows the number of butterflies as a function of stream size.

The streaming algorithms were implemented in `C++`[4] and compiled with `g++` compiler using `-O3` as the optimization level. We ran the experiments on a machine equipped with a 2.0 GHz 16-Core Intel E5 2650 processor and 128GB of memory.

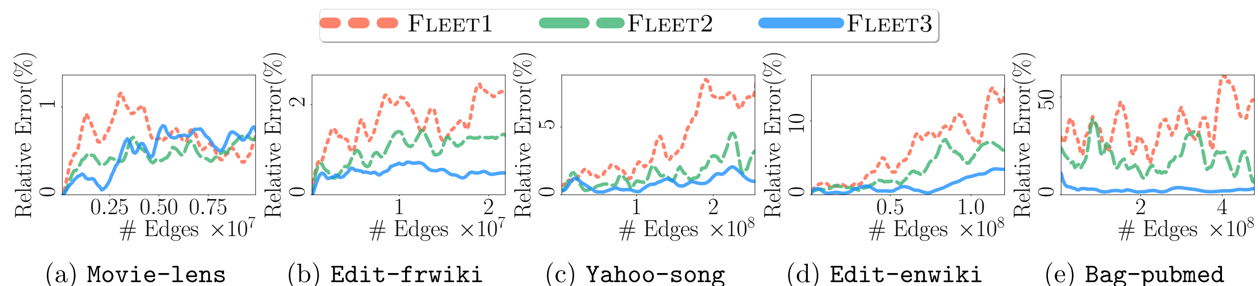
[3]http://konect.cc/

[4]https://github.com/beginner1010/ButterflyCounting



(a) `Movie-lens` (b) `Edit-frwiki` (c) `Yahoo-song` (d) `Edit-enwiki` (e) `Bag-pubmed`

Figure 5.4: Accuracy of FLEET1, FLEET2, and FLEET3 for $\gamma = 0.5$ versus reservoir size. Bottom x-axis shows the reservoir size and top x-axis shows the sample rate, defined as the ratio of the reservoir size to the stream size.

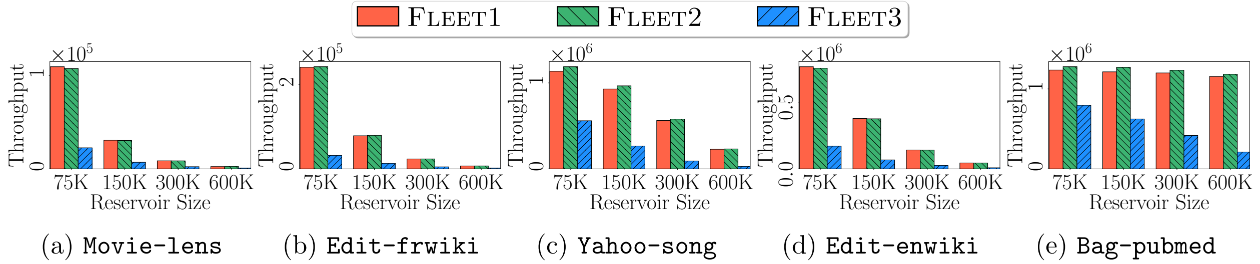


(a) `Movie-lens` (b) `Edit-frwiki` (c) `Yahoo-song` (d) `Edit-enwiki` (e) `Bag-pubmed`

Figure 5.6: Accuracy of FLEET1, FLEET2, and FLEET3 at different points in the stream, reservoir size is $3.00 \times 10^2 K$ and $\gamma = 0.9$.

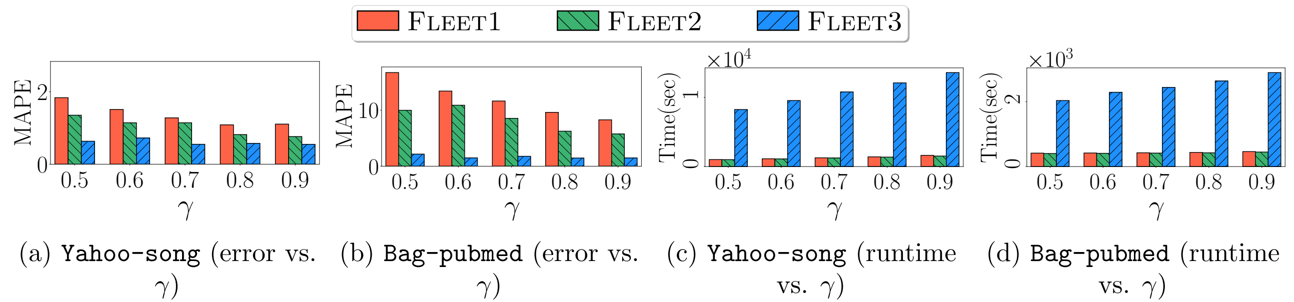
5.4.1 Accuracy

If the true value of the butterfly count is $x > 0$, then the relative error of an estimate $\hat{x}$ is defined as $|x - \hat{x}|/x$ and is usually shown as percent error. We also used MAPE (Mean Average Percentage Error) to measure the accuracy over the entire stream, defined as the average of the relative error, taken over the entire stream. Figure 5.4 shows the accuracy on the entire stream vs the reservoir size. Larger reservoirs yield better accuracies, as expected. FLEET3 can keep the estimation error around 1% for all networks by storing only 600K edges in the reservoir. This corresponds to 6%, 2.7%, 0.49%, 0.23%, and 0.12% of the total stream sizes for `Movie-lens`, `Edit-frwiki`, `Edit-enwiki`, `Yahoo-song`, and `Bag-pubmed`, respectively. When the reservoir size is 300K, FLEET3 yields 3% error for `Edit-enwiki` and `Bag-pubmed` and less than 1% for other networks. As expected FLEET2 has better accuracy than FLEET1, and FLEET3 has the best accuracy.



(a) `Movie-lens` (b) `Edit-frwiki` (c) `Yahoo-song` (d) `Edit-enwiki` (e) `Bag-pubmed`

Figure 5.8: Throughput of FLEET1, FLEET2, and FLEET3 algorithms as a function of reservoir size where $\gamma = 0.6$.



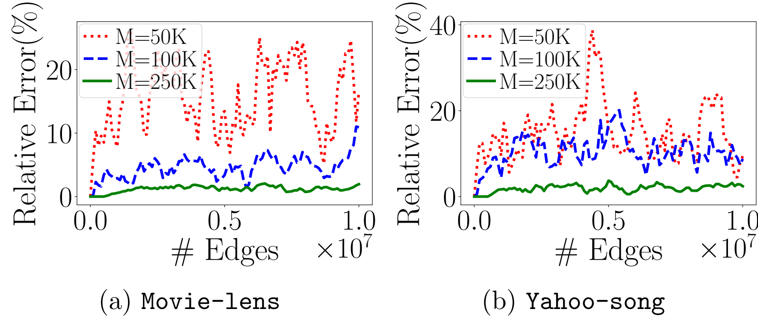
(a) `Yahoo-song` (error vs. γ) (b) `Bag-pubmed` (error vs. γ) (c) `Yahoo-song` (runtime vs. γ) (d) `Bag-pubmed` (runtime vs. γ)

Figure 5.10: Accuracy and runtime of FLEET1, FLEET2, and FLEET3 as a function of γ where $\mathrm{M} = 600K$.

Figure 5.6 shows the relative error at different points in the stream, for a fixed reservoir size. As the stream size increases, the error of FLEET1 and FLEET2 increase slightly. This can be attributed to the fact that the edge sampling probability p is proportional to $1/t$, where t is the number of edges, and from Lemma 15, the probability of a given relative error decreases with $p^4\xi^t$. Unless ξ^t increases as the fourth power of t, the probability of a given relative error increases with the stream size.

Table 5.1: Properties of the bipartite graphs. $|E|$ is the number of edges, ξ the total number of butterflies, and the butterfly density is the ratio $\xi/|E|^4$. $|L|$ and $|R|$ are the number of vertices in the left and right partions, respectively.

Graphs	$\|E\|$	$\|L\|$	$\|R\|$	ξ	Butterfly density
`Movie-lens`	10M	69K	10K	1.1T	1.1×10^{-16}
`Edit-frwiki`	22M	288K	3M	601.2B	2.5×10^{-18}
`Yahoo-song`	256M	1M	624K	101.4T	2.3×10^{-20}
`Edit-enwiki`	122M	3M	21M	2T	9.1×10^{-21}
`Bag-pubmed`	483M	8M	141K	40.8T	7.4×10^{-22}



(a) `Movie-lens` (b) `Yahoo-song`

Figure 5.12: Relative error vs. number of edges received for FLEETSSW. Window size $= 5 \times 10^6$ edges, $\gamma = 0.9$.

Butterfly Density: We note the errors of FLEET1 and FLEET2 for a given reservoir size are roughly correlated with the butterfly density (ξ^t/t^4 where t is the number of edges). One reason is as follows. Following Lemma 15, the probability of a high relative error decreases with $p^4\xi^t$. Setting $p \approx M/t$ where M is the reservoir size, this is M^4 times the butterfly density $\frac{\xi^t}{t^4}$, showing that the error probability decreases quickly as the butterfly density increases. When the networks are ordered according to increasing butterfly density, we get the order `Bag-pubmed`, `Edit-enwiki`, `Yahoo-song`, `Edit-frwiki`, and `Movie-lens`. We note that for the same reservoir size, this is exactly the increasing order of accuracy (decreasing order or error) for algorithms FLEET1 and FLEET2 (Figure 5.6). The trend is not so clear for algorithm FLEET3, since its accuracy depends heavily on the temporal order of the edges within a butterfly.

5.4.2 Runtime and Throughput

The better accuracy of FLEET3 comes at the cost of increased runtime. From Figure 5.8, we see FLEET3 has the lowest throughput (number of edges processed per second), while FLEET1 and FLEET2 have similar throughputs. The reason is there is one per-edge butterfly computation for each arriving edge in FLEET3, where as there is one such computation only for each sampled edge in FLEET1 and FLEET2. The throughput decreases as the reservoir size increases, due to the increased cost of per-edge butterfly counting on the reservoir. FLEET3 is able to achieve quite a high throughput, e.g. 6.2×10^5 edges per second on graph `Bag-pubmed` with reservoir size 150K, making it suitable for practical scenarios. FLEET2 always has a slightly higher throughput than

FLEET1. Overall, FLEET3 has the best accuracy with a good throughput, while FLEET2 trades a lower accuracy for a higher throughput.

5.4.3 Impact of γ on Runtime and Accuracy

Figures 5.9a and 5.9b show the accuracy as a function of γ. As γ increases, the average size of the reservoir increases, while the frequency of sub-sampling also increases. The accuracies of FLEET1 and FLEET2 improve slightly as γ increases from 0.5 to 0.9 e.g. for graph `Yahoo-song`. In contrast, from Figures 5.9c and 5.9d, the runtime increases for all estimators as γ increases. A value of γ of about 0.7 seems to be a good “middle ground” since it achieves nearly the best throughput as well as accuracy.

5.4.4 Sliding Window

Figure 5.12 shows the relative error of FLEETSSW for window size $W = 5$M edges, when the reservoir size is varied from 1% to 5% of W. The accuracy improves as the reservoir size increases; when M is 5% of the window size, the relative error is always under 5%. The number of butterflies within a window ranges from 5×10^{10} to 10^{11} for `Movie-lens` and 1×10^{10} to 6×10^{10} for `Yahoo-song`. We also experimented with FLEETTSW for time-based windows. We used 30 queries, and a window size is randomly generated at query time. When the reservoir size M is 10% of the stream size, the average relative error over the queries is 2.55% for movie, 5.52% for `Edit-frwiki` and 6.32% for `Edit-enwiki`. This result shows FLEETTSW can achieve good accuracy using memory much smaller than the whole stream.

CHAPTER 6. CONCLUSION AND FUTURE WORK

In this dissertation, we described a suite of randomized algorithms for counting triangles and butterflies in static and streaming networks. We detailed state-of-the-art triangle counting algorithms. We theoretically and empirically compared the run-time, memory usage performance and proposed the most efficient algorithm in various scenarios. Note that while there are numerous existing work for triangle counting, there is no comprehensive comparison between the triangle methods in the literature. The existing methods are developed for different scenarios, and, therefore, thorough empirical and theoretical analyses will be helpful to shed light on the actual performance of the prior methods.

We introduced our butterfly estimators which are considered state-of-the-art in the literature. Our butterfly counting algorithms in static networks are fast and efficient that are enabled to estimate the number of butterflies with less than 1% error in large networks with trillions of butterflies within 5 seconds. We developed streaming algorithms which use a fixed size memory to maintain an accurate butterfly estimate over the entire stream. In addition, we developed algorithms for sliding window scenarios in bipartite network stream. Our methods outperform the existing methods in terms of accuracy and runtime. We also provide C++ libraries that includes the implementation of static and streaming algorithms for triangle and butterfly counting and also support fully-dynamic streams where both insertion and deletion of edges occur in the stream. This framework is supplied with a user-friendly interface that shows the progress of executions and will output the runtime and estimates of triangle count along with other useful statistics.

One future direction is to extend our scope to a more restricted yet interesting scenario which is butterfly counting in online social network (OSNs) where the topology of the input network is not readily shared, due to either data privacy or the sheer size of the input data; however, it is still expected that crawling from a vertex of the network to its neighbors are available through an

API. Because only a few number of vertices/edges (sub-linear to the size of network) can be explored, an efficient computation of butterfly count could be highly challenging. Note that none of the streaming nor static algorithms are applicable in this restricted scenario.

BIBLIOGRAPHY

[1] Gnip, 2017. (`https://gnip.com/about/`). ↑10

[2] The graph api for the facebook social graph, 2017. (`https://developers.facebook.com/docs/graph-api`). ↑10

[3] S. Acer, A. Yaşar, S. Rajamanickam, M. Wolf, and Ü. V. Catalyürek. Scalable triangle counting on distributed-memory systems. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–5. IEEE, 2019. ↑16

[4] N. K. Ahmed, J. Neville, R. A. Rossi, and N. Duffield. Efficient graphlet counting for large networks. In *2015 IEEE International Conference on Data Mining*, pages 1–10. IEEE, 2015. ↑21

[5] N. K. Ahmed, J. Neville, R. A. Rossi, N. Duffield, and T. L. Willke. Graphlet decomposition: Framework, algorithms, and applications. *KAIS*, pages 1–32, 2016. ↑20

[6] S. Aksoy, T. G. Kolda, and A. Pinar. Measuring and modeling bipartite graphs with community structure. *Journal of Complex Networks*, 5(4):581–603, 2017. ↑5, ↑20

[7] M. Al Hasan and V. S. Dave. Triangle counting in large networks: a review. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 8(2):e1226, 2018. ↑1

[8] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. In *European Symposium on Algorithms*, pages 354–364. Springer, 1994. ↑21

[9] S. Arifuzzaman, M. Khan, and M. Marathe. Patric: A parallel algorithm for counting triangles in massive networks. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 529–538. ACM, 2013. ↑16

[10] S. Arifuzzaman, M. Khan, and M. Marathe. A space-efficient parallel algorithm for counting exact triangles in massive networks. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 527–534. IEEE, 2015. ↑16

[11] A. Azad, A. Buluç, and J. Gilbert. Parallel triangle counting and enumeration using matrix algebra. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 804–811. IEEE, 2015. ↑17

[12] B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. In *2002 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002)*. Stanford InfoLab, 2001. ↑12

[13] G. Ballard, T. G. Kolda, A. Pinar, and C. Seshadhri. Diamond sampling for approximate maximum all-pairs dot-product (mad) search. In *2015 IEEE International Conference on Data Mining*, pages 11–20. IEEE, 2015. ↑1

[14] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. Efficient algorithms for large-scale local triangle counting. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 4(3):13, 2010. ↑19

[15] S. K. Bera and A. Chakrabarti. Towards tighter space bounds for counting triangles and other substructures in graph streams. In *34th Symposium on Theoretical Aspects of Computer Science*, 2017. ↑22

[16] M. A. Bhuiyan, M. Rahman, M. Rahman, and M. Al Hasan. Guise: Uniform sampling of graphlets for large graph analysis. In *2012 IEEE 12th International Conference on Data Mining*, pages 91–100. IEEE, 2012. ↑1

[17] M. Bisson and M. Fatica. High performance exact triangle counting on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 28(12):3501–3510, 2017. ↑17

[18] I. Bordino, D. Donato, A. Gionis, and S. Leonardi. Mining large networks with subgraph counting. In *2008 Eighth IEEE International Conference on Data Mining*, pages 737–742. IEEE, 2008. ↑22

[19] S. P. Borgatti and M. G. Everett. Network analysis of 2-mode data. *Social Networks*, 19(3):243 – 269, 1997. ↑4, ↑5, ↑20

[20] V. Braverman, R. Ostrovsky, and C. Zaniolo. Optimal sampling from sliding windows. In *Proceedings of the twenty-eighth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 147–156, 2009. ↑12

[21] M. Bressan, F. Chierichetti, R. Kumar, S. Leucci, and A. Panconesi. Counting graphlets: Space vs time. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, pages 557–566, 2017. ↑1, ↑20

[22] L. S. Buriol, G. Frahling, S. Leonardi, and C. Sohler. Estimating clustering indexes in data streams. In *European Symposium on Algorithms*, pages 618–632. Springer, 2007. ↑22

[23] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on computing*, 14(1):210–223, 1985. ↑13

[24] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of symbolic computation*, 9(3):251–280, 1990. ↑13, ↑14

[25] G. Cormode and H. Jowhari. A second look at counting triangles in graph streams. *Theoretical Computer Science*, 552:44–51, 2014. ↑19

[26] Y. Cui, D. Xiao, and D. Loguinov. On efficient external-memory triangle listing. *IEEE Transactions on Knowledge and Data Engineering*, 2018. ↑13

[27] L. De Stefani, A. Epasto, M. Riondato, and E. Upfal. TRIÈST: Counting Local and Global Triangles in Fully-Dynamic Streams with Fixed Memory Size. In *KDD*, pages 825–834, 2016. ↑60, ↑68

[28] E. Donato, M. Ouyang, and C. Peguero-Isalguez. Triangle counting with a multi-core computer. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018. ↑15

[29] J.-P. Eckmann and E. 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. ↑3

[30] T. Eden, A. Levi, D. Ron, and C. Seshadhri. Approximately counting triangles in sublinear time. *SIAM Journal on Computing*, 46(5):1603–1646, 2017. ↑18

[31] R. Gemulla, W. Lehner, and P. J. Haas. A dip in the reservoir: Maintaining sample synopses of evolving datasets. In *Proceedings of the 32nd international conference on Very large data bases*, pages 595–606, 2006. ↑67, ↑69

[32] P. B. Gibbons and S. Tirthapura. Distributed streams algorithms for sliding windows. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 63–72, 2002. ↑12

[33] I. Giechaskiel, G. Panagopoulos, and E. Yoneki. Pdtl: Parallel and distributed triangle listing for massive graphs. In *2015 44th International Conference on Parallel Processing*, pages 370–379. IEEE, 2015. ↑13

[34] O. Green, R. McColl, and D. A. Bader. Gpu merge path: a gpu merging algorithm. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 331–340. ACM, 2012. ↑17

[35] O. Green, L.-M. Munguía, and D. A. Bader. Load balanced clustering coefficients. In *Proceedings of the first workshop on Parallel programming for analytics applications*, pages 3–10. ACM, 2014. ↑14

[36] O. Green, P. Yalamanchili, and L.-M. Munguía. Fast triangle counting on the gpu. In *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*, pages 1–8. IEEE Press, 2014. ↑17

[37] A. Hajnal and E. Szemerédi. Proof of a conjecture of p. erdos. *Combinatorial theory and its applications*, 2:601–623, 1970. ↑60

[38] T. R. Halford and K. M. Chugg. An algorithm for counting short cycles in bipartite graphs. *IEEE Transactions on Information Theory*, 52(1):287–292, 2006. ↑20

[39] Y. Hu, H. Liu, and H. H. Huang. High-performance triangle counting on gpus. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–5. IEEE, 2018. ↑17

[40] Y. Hu, H. Liu, and H. H. Huang. Tricore: Parallel triangle counting on gpus. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 171–182. IEEE, 2018. ↑17

[41] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. *SIAM Journal on Computing*, 7(4):413–423, 1978. ↑14

[42] S. Jain and C. Seshadhri. A fast and provable method for estimating clique counts using turán's theorem. In *Proceedings of the 26th International Conference on World Wide Web*, pages 441–449. International World Wide Web Conferences Steering Committee, 2017. ↑1, ↑20

[43] M. Jha, C. Seshadhri, and A. Pinar. Path sampling: A fast and provable method for estimating 4-vertex subgraph counts. In *Proceedings of the 24th International Conference on World Wide Web*, pages 495–505. International World Wide Web Conferences Steering Committee, 2015. ↑1, ↑20

[44] M. Jha, C. Seshadhri, and A. Pinar. A space-efficient streaming algorithm for estimating transitivity and triangle counts using the birthday paradox. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 9(3):15, 2015. ↑19

[45] H. Jowhari and M. Ghodsi. New streaming algorithms for counting triangles in graphs. In *International Computing and Combinatorics Conference*, pages 710–716. Springer, 2005. ↑19

[46] D. M. Kane, K. Mehlhorn, T. Sauerwald, and H. Sun. Counting arbitrary subgraphs in data streams. In *International Colloquium on Automata, Languages, and Programming*, pages 598–609. Springer, 2012. ↑22

[47] T. A. Kanewala, M. Zalewski, and A. Lumsdaine. Distributed, shared-memory parallel triangle counting. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, page 5. ACM, 2018. ↑15, ↑16

[48] T. G. Kolda, A. Pinar, T. Plantenga, C. Seshadhri, and C. Task. Counting triangles in massive graphs with mapreduce. *SIAM Journal on Scientific Computing*, 36(5):S48–S77, 2014. ↑19

[49] M. N. Kolountzakis, G. L. Miller, R. Peng, and C. E. Tsourakakis. Efficient triangle counting in large graphs via degree-based vertex partitioning. In *International Workshop on Algorithms and Models for the Web-Graph*, pages 15–24. Springer, 2010. ↑60

[50] M. N. Kolountzakis, G. L. Miller, R. Peng, and C. E. Tsourakakis. Efficient triangle counting in large graphs via degree-based vertex partitioning. *Internet Mathematics*, 8(1-2):161–185, 2012. ↑19

[51] J. Kunegis. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 1343–1350. ACM, 2013. ↑71

[52] J. Kunegis. Konect network dataset, 2017. (`http://konect.uni-koblenz.de`). ↑31

[53] M. Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical computer science*, 407(1-3):458–473, 2008. ↑13

[54] M. Latapy, C. Magnien, and N. D. Vecchio. Basic notions for the analysis of large two-mode networks. *Social Networks*, 30(1):31 – 48, 2008. ↑4

[55] Y. 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. ↑19, ↑62, ↑68

[56] P. G. Lind, M. C. González, and H. J. Herrmann. Cycles and clustering in bipartite networks. *Phys. Rev. E*, 72:056127, 2005. ↑5, ↑20

[57] B. Liu, L. Yuan, X. Lin, L. Qin, W. Zhang, and J. Zhou. Efficient (a, β)-core computation: an index-based approach. In *The World Wide Web Conference*, pages 1130–1141. ACM, 2019. ↑1

[58] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007. ↑3

[59] M. Manjunath, K. Mehlhorn, K. Panagiotou, and H. Sun. Approximate counting of cycles in streams. In *European Symposium on Algorithms*, pages 677–688. Springer, 2011. ↑22

[60] A. McGregor, S. Vorotnikova, and H. T. Vu. Better algorithms for counting triangles in data streams. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 401–411. ACM, 2016. ↑19

[61] B. Menegola. An external memory algorithm for listing triangles. 2010. ↑13

[62] M. Newman. Scientific collaboration networks. ii. shortest paths, weighted networks, and centrality. *Phys. Rev. E*, 64:016132, 2001. ↑4

[63] M. E. J. Newman. Scientific collaboration networks. i. network construction and fundamental results. *Phys. Rev. E*, 64:016131, 2001. ↑4

[64] T. Opsahl. Triadic closure in two-mode networks: Redefining the global and local clustering coefficients. *Social Networks*, 35(2):159 – 167, 2013. Special Issue on Advances in Two-mode Social Networks. ↑5, ↑20

[65] R. Pagh and C. E. Tsourakakis. Colorful triangle counting and a mapreduce implementation. *Information Processing Letters*, 112(7):277–281, 2012. ↑23, ↑50, ↑60

[66] S. Pandey, X. S. Li, A. Buluc, J. Xu, and H. Liu. H-index: Hash-indexing for parallel triangle counting on gpus. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2019. ↑18

[67] H.-M. Park and C.-W. Chung. An efficient mapreduce algorithm for counting triangles in a very large graph. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 539–548. ACM, 2013. ↑16

[68] H.-M. Park, S.-H. Myaeng, and U. Kang. Pte: Enumerating trillion triangles on distributed systems. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1115–1124. ACM, 2016. ↑13

[69] H.-M. Park, F. Silvestri, U. Kang, and R. Pagh. Mapreduce triangle enumeration with guarantees. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pages 1739–1748. ACM, 2014. ↑13

[70] A. Pavan, K. Tangwongsan, S. Tirthapura, and K.-L. Wu. Counting and sampling triangles from a graph stream. *Proceedings of the VLDB Endowment*, 6(14), 2013. ↑19, ↑20

[71] A. Pinar, C. Seshadhri, and V. Vishal. Escape: Efficiently counting all 5-vertex subgraphs. In *Proceedings of the 26th International Conference on World Wide Web*, pages 1431–1440, 2017. ↑20, ↑21

[72] A. Polak. Counting triangles in large graphs on gpu. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 740–746. IEEE, 2016. ↑17

[73] N. Pržulj, D. G. Corneil, and I. Jurisica. Efficient estimation of graphlet frequency distributions in protein–protein interaction networks. *Bioinformatics*, 22(8):974–980, 2006. ↑1

[74] M. Rahman and M. Al Hasan. Approximate triangle counting algorithms on multi-cores. In *2013 IEEE International Conference on Big Data*, pages 127–133. IEEE, 2013. ↑19

[75] G. Robins and M. Alexander. Small worlds among interlocking directors: Network structure and distance in bipartite graphs. *Computational & Mathematical Organization Theory*, 10(1):69–94, 2004. ↑5, ↑20

[76] S.-V. Sanei-Mehri, A. E. Sariyuce, and S. Tirthapura. Butterfly counting in bipartite networks. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2150–2159. ACM, 2018. ↑1, ↑10, ↑21, ↑37, ↑38, ↑40, ↑41, ↑42, ↑58

[77] S.-V. Sanei-Mehri, Y. Zhang, A. E. Sariyüce, and S. Tirthapura. Fleet: butterfly estimation from a bipartite graph stream. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 1201–1210, 2019. ↑10, ↑21, ↑57

[78] A. E. Sarıyüce and A. Pinar. Peeling bipartite networks for dense subgraph discovery. In *WSDM*, 2018. ↑5

[79] T. Schank. Algorithmic aspects of triangle-based network analysis. 2007. ↑13

[80] T. Schank and D. Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *International workshop on experimental and efficient algorithms*, pages 606–609. Springer, 2005. ↑2, ↑13, ↑15, ↑17

[81] C. Seshadhri. A simpler sublinear algorithm for approximating the triangle count. *arXiv preprint arXiv:1505.01927*, 2015. ↑18

[82] C. Seshadhri, A. Pinar, and T. G. Kolda. Triadic measures on graphs: The power of wedge sampling. In *Proceedings of the 2013 SIAM International Conference on Data Mining*, pages 10–18. SIAM, 2013. ↑28

[83] C. Seshadhri, A. Pinar, and T. G. Kolda. Wedge sampling for computing clustering coefficients and triangle counts on large graphs. *Statistical Analysis and Data Mining: The ASA Data Science Journal*, 7(4):294–307, 2014. ↑18, ↑20

[84] M. Sevenich, S. Hong, A. Welc, and H. Chafi. Fast in-memory triangle listing for large real-world graphs. In *Proceedings of the 8th Workshop on Social Network Mining and Analysis*, page 2. ACM, 2014. ↑13

[85] J. Shi and J. Shun. Parallel algorithms for butterfly computations. *arXiv preprint arXiv:1907.08607*, 2019. ↑21

[86] K. 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. ↑19

[87] K. Shin, M. Hammoud, E. Lee, J. Oh, and C. Faloutsos. Tri-fly: Distributed estimation of global and local triangle counts in graph streams. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 651–663. Springer, 2018. ↑20

[88] K. Shin, J. Kim, B. Hooi, and C. Faloutsos. Think before you discard: Accurate triangle counting in graph streams with deletions (supplementary document). ↑68

[89] K. Shin, J. Kim, B. Hooi, and C. 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. ↑67, ↑68

[90] K. Shin, E. Lee, J. Oh, M. Hammoud, and C. Faloutsos. Cocos: Fast and accurate distributed triangle counting in graph streams. *Manuscript submitted for publication*, 1:1–1, 2019. ↑20

[91] J. Shun and K. Tangwongsan. Multicore triangle computations without tuning. In *2015 IEEE 31st International Conference on Data Engineering*, pages 149–160. IEEE, 2015. ↑13, ↑14, ↑15

[92] L. D. Stefani, A. Epasto, M. Riondato, and E. 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. ↑19, ↑62

[93] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th international conference on World wide web*, pages 607–614. ACM, 2011. ↑16

[94] K. Tangwongsan, A. Pavan, and S. Tirthapura. Parallel triangle counting in massive streaming graphs. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 781–786. ACM, 2013. ↑20

[95] A. S. Tom and G. Karypis. A 2d parallel triangle counting algorithm for distributed-memory architectures. In *Proceedings of the 48th International Conference on Parallel Processing*, pages 1–10, 2019. ↑16

[96] A. S. Tom, N. Sundaram, N. K. Ahmed, S. Smith, S. Eyerman, M. Kodiyath, I. Hur, F. Petrini, and G. Karypis. Exploring optimizations on shared-memory platforms for parallel triangle counting algorithms. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2017. ↑14, ↑16

[97] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. 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. ACM, 2009. ↑23

[98] A. Turk and D. Turkoglu. Revisiting wedge sampling for triangle counting. In *The World Wide Web Conference*, pages 1875–1885. ACM, 2019. ↑19, ↑28, ↑30

[99] D. Türkoglu and A. Turk. Edge-based wedge sampling to estimate triangle counts in very large graphs. In *2017 IEEE International Conference on Data Mining (ICDM)*, pages 455–464. IEEE, 2017. ↑18, ↑26, ↑27

[100] S. V. N. Vishwanathan, N. N. Schraudolph, R. Kondor, and K. M. Borgwardt. Graph kernels. *Journal of Machine Learning Research*, 11(Apr):1201–1242, 2010. ↑3

[101] J. Wang, A. W. C. Fu, and J. Cheng. Rectangle counting in large bipartite graphs. In *2014 IEEE International Congress on Big Data*, pages 17–24, 2014. ↑5, ↑21

[102] K. Wang, X. Lin, L. Qin, W. Zhang, and Y. Zhang. Vertex priority based butterfly counting for large-scale bipartite networks. *Proceedings of the VLDB Endowment*, 12(10):1139–1152, 2019. ↑1, ↑21, ↑58

[103] L. Wang, Y. Wang, C. Yang, and J. D. Owens. A comparative study on exact triangle counting algorithms on the gpu. In *Proceedings of the ACM Workshop on High Performance Graph Processing*, pages 1–8. ACM, 2016. ↑17

[104] P. Wang, Y. Qi, Y. Sun, X. Zhang, J. Tao, and X. Guan. Approximately counting triangles in large graph streams including edge duplicates with a fixed memory usage. *Proceedings of the VLDB Endowment*, 11(2):162–175, 2017. ↑20

[105] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. Gunrock: A high-performance graph processing library on the gpu. In *ACM SIGPLAN Notices*, volume 51, page 11. ACM, 2016. ↑17

[106] B. Wu, K. Yi, and Z. Li. Counting triangles in large graphs by random sampling. *IEEE Transactions on Knowledge and Data Engineering*, 28(8):2013–2026, 2016. ↑18

[107] L. Zhang, Y. Han, Y. Yang, M. Song, S. Yan, and Q. Tian. Discovering discriminative graphlets for aerial image categories recognition. *IEEE Transactions on Image Processing*, 22(12):5071–5084, 2013. ↑1, ↑2