# Accurate and Fast Approximate Graph Mining at Scale

by

Anna Arpaci-Dusseau

S.B., Computer Science and Engineering, Massachusetts Institute of Technology (2023)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2024



Authored by: Anna Arpaci-Dusseau
Department of Electrical Engineering and Computer Science
May 8, 2024

Certified by: Xuhao Chen
Research Scientist, MIT CSAIL, Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

# Accurate and Fast Approximate Graph Mining at Scale

by

Anna Arpaci-Dusseau

Submitted to the Department of Electrical Engineering and Computer Science
on May 8, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

## ABSTRACT

Approximate graph pattern mining (A-GPM) is an important data analysis tool for numerous graph-based applications. There exist sampling-based A-GPM systems to provide automation and generalization over a wide variety of use cases. Despite improved usability, there are two major obstacles that prevent existing A-GPM systems being adopted in practice. First, the termination mechanism that decides when to terminate sampling lacks theoretical backup on confidence, and performs significantly unstable and thus slow in practice. Second, they particularly suffer poor performance when dealing with the "needle-in-the-hay" cases, because a huge number of samples are required to converge, given the extremely low hit rate of their lazy-pruning strategy and fixed sampling schemes.

We build ScaleGPM, an accurate and fast A-GPM system that removes the two obstacles. First, we propose a novel on-the-fly convergence detection mechanism to achieve stable termination and provide theoretical guarantee on the confidence, with negligible online overhead. Second, we propose two techniques to deal with the "needle-in-the-hay" problem, *eager-verify* and *hybrid sampling*. Our eager-verify method drastically improves sampling hit rate by pruning unpromising candidates as early as possible. Hybrid sampling further improves performance by automatically choosing the better scheme between fine-grained and coarse-grained sampling schemes. Experiments show that our online convergence detection mechanism can precisely detect convergence, and results in stable and rapid termination with theoretically guaranteed confidence. We also show the effectiveness of eager-verify in improving the hit rate, and the scheme-selection mechanism in correctly choosing the better scheme for various cases. Overall, ScaleGPM achieves an `geomean` average of 565× (up to 610,169×) speedup over the state-of-the-art A-GPM system, Arya. ScaleGPM is also four orders of magnitude faster than state-of-the-art exact GPM system, GraphZero. In particular, ScaleGPM handles billion-scale graphs in seconds, where existing systems either run out of memory or fail to complete in hours.

Thesis supervisor: Xuhao Chen
Title: Research Scientist, MIT CSAIL

# Acknowledgments

First, I would like to thank to my thesis advisor, Xuhao Chen for his continual advice and support. I would also like to thank Zixiang Zhou for his insight and work on the proofs behind the ScaleGPM project. I would also like to thank all the staff and professors in the EECS department who have taught me over the past years for cultivating my interest in performance engineering and computer systems. Additionally, I am thankful to my partner who tirelessly listened to my thoughts on A-GPM problems as I completed my thesis. Finally, I want to thank my parents and sister, for all the support and encouragement they have provided over the years.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Chapter 1

# Introduction

Graph Pattern Mining (GPM) [1]–[8], which searches for instances of small patterns (e.g., triangles) in a data graph, is a key building block in graph-based data mining. Despite its wide adoption in real-world applications, GPM is hard to scale to large problem size as it is computationally quite expensive [8]. Fortunately, many real-world use cases do not require exact GPM solutions. For instance, to characterize network structures in biochemistry, ecology and engineering [9], we need only to estimate the pattern (a.k.a motif) frequency distribution, instead of exactly counting motifs. Therefore we only need Approximate GPM (A-GPM), which trades accuracy for speed, as long as some error bound is met in a given application context.

There have been A-GPM systems, e.g., ASAP [10] and Arya [11], that are built to simplify programming and improve usability. The system responsibility is two fold. First, they provide *generalized* sampling techniques for arbitrary patterns and programming APIs to support a wide variety of use cases. Second, they provide automated mechanisms to determine key sampling parameters, e.g., how many samples required to draw for a given error bound.

However, existing A-GPM systems have two major limitations that prevent them from being adopted in practice. First, the automated termination mechanism to determine when it is confident enough to terminate sampling, does not have strong theoretical backup on

confidence, and thus is significantly unstable and slow in practice (see Section 3.1 for detail). In existing systems, sampling is terminated when the predicted number of samples $N_s$ have been drawn. They predict $N_s$ based on an offline error-latency profiling (ELP) procedure before launching the sampling. However, the ELP prediction of $N_s$ is dependent on the true count, which is supposed to be estimated on the $N_s$ samples, creating a circular dependency. Therefore, the ELP cannot theoretically guarantee confidence in bounding the error. Also, the ELP returns *unstable* predictions, i.e., huge variances or even failures in predicting $N_s$, which leads to poor performance.

The second limitation is the poor performance when dealing with the extremely sparse, a.k.a, *needle-in-the-hay* cases, where only a small number of matches of the pattern exist in a big graph (see Fig. 3.2). In this case, neighbor sampling (NS), the sampling scheme used in ASAP and Arya, fails to hit a match in most of the samples, leading to a very low sampling hit rate, e.g., 0.0000017%. This results in slow convergence as a huge number of samples are required to meet the error bound. In some extreme cases, we observe that Arya can be even slower than exact GPM solutions.

To address the unstable termination problem, we propose a novel on-the-fly convergence detection method for NS. Instead of offline predicting the termination condition before execution, our *online* method dynamically collects statistics and predicts errors during the sampling execution, until convergence. The convergence is detected when the predicted error is below the user specified error bound. We prove formally in Theorem 1 that the probability of the true error being smaller than our predicted error is $1 - \delta$, for a given confidence $1 - \delta$. This provides us theoretical guarantee in confidence, which prior systems lack. Meanwhile, our detected termination points are stable across different runs while the ELP method in prior systems is highly unstable. Therefore, our online method can significantly accelerate the execution, because it needs much fewer samples than prior systems, while causing negligible online overhead as statistics can be trivially collected.

As for the low hit rate problem in NS, we find that the root cause is the delayed check on

pattern closure in prior systems, which we call lazy-verify. We then introduce *eager-verify* that applying pruning at its earliest possible point to avoid unpromising candidates and thus improve hit rates. We prove theoretically that eager-verify is unbiased, and show that it drastically improves the sampling hit rate, which in turn significantly accelerate convergence. Furthermore, for extremely sparse cases where the fine-grained NS sampling scheme can not handle, we propose a *hybrid sampling* method by adaptively selecting the better sampling scheme, between NS and a coarse-grained scheme, graph sparsification (GS), for various graphs and patterns. For scheme selection, we build a cost model for each sampling scheme to estimate the execution time. With our models, hybrid sampling manages to always select the cheaper one from NS and GS for a wide variety of test cases.

We then build ScaleGPM, an accurate and fast A-GPM system that incorporates our proposed mechanisms: online convergence detection, eager-verify and hybrid sampling. ScaleGPM efficiently leverages parallel hardware and provides flexible modes to meet various accuracy requirements with confidence. Our experiments on a multicore CPU show that, with orders-of-magnitude reduction in required samples and improvement in hit rates, ScaleGPM achieves an average of 565× (up to 610, 169×) speedup over the state-of-the-art A-GPM system, Arya. The hybrid method further improves performance by 61× on extremely hard (i.e. sparse) cases. Compared to the state-of-the-art exact GPM system, GraphZero, ScaleGPM is also four orders of magnitude faster. Particularly, ScaleGPM handles billion-scale graphs in seconds where previous frameworks either run out of memory or fail to complete in hours. This paper makes the following contributions:

- We conduct analysis and empirical study on sampling schemes, and identify the two major limitations and their root causes in existing A-GPM systems.
- We propose a novel on-the-fly convergence detection method for the NS sampling scheme, which is the first to provide theoretical guarantee on confidence.
- We introduce the eager-verify mechanism to improve hit rate of the NS scheme and

thus achieve faster convergence speed.

- We propose hybrid sampling to further improve performance, by adaptively selecting a better scheme based on cost models.

- We build ScaleGPM that incorporates the above novel mechanisms, and evaluation shows that it significantly outperforms the state-of-the-art system, and efficiently handles huge graphs.

# Chapter 2

# Background

## 2.1 Graph Pattern Mining (GPM)

Graph Pattern Mining (GPM) finds subgraphs that match given pattern(s) $\mathcal{P}$ in a given data graph $\mathcal{G}$. There exist *explicit* GPM tasks like subgraph counting (SC) and motif counting (MC), and *implicit* tasks like frequent subgraph mining (FSM) [12]. GPM has numerous applications in AI and big-data [8], including bioinformatics, chemical engineering, fraud detection, social network analysis, recommender systems, etc.

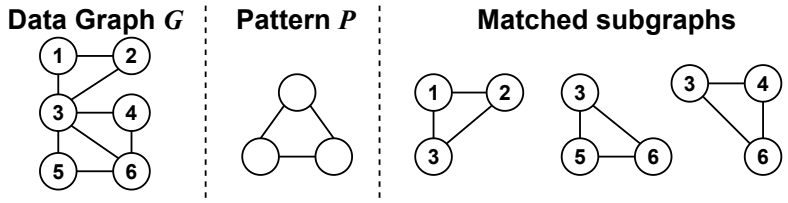



Figure 2.1: graph pattern mining example [8]. The pattern $\mathcal{P}$ is a triangle, and 3 triangles are found in the data graph $\mathcal{G}$.

We follow [8] to introduce the Graph Pattern Mining problem. Let $\mathcal{G}$ ($\mathcal{V}$, $\mathcal{E}$) be an undirected graph with $\mathcal{V}$ as the vertex and $\mathcal{E}$ as the edge set. Given a vertex $v \in \mathcal{V}$, the neighbor set of $v$ is $\mathcal{N}(v)$, the degree $d_v$ of $v$ is $|\mathcal{N}(v)|$ and $\Delta$ is the maximum degree in $\mathcal{G}$. A graph $G'(W, F)$ is said to be a subgraph of $\mathcal{G}$ if $W \subseteq \mathcal{V}$ and $F \subseteq \mathcal{E}$. $G'$ is a *vertex-induced subgraph* of $\mathcal{G}$ if $F$ contains all the edges in $E$ whose endpoints are in $W$. $G'$ is an *edge-induced*

*subgraph* of $\mathcal{G}$ if $W$ contains all the vertices in $\mathcal{V}$ which are the endpoints of edges in $F$.

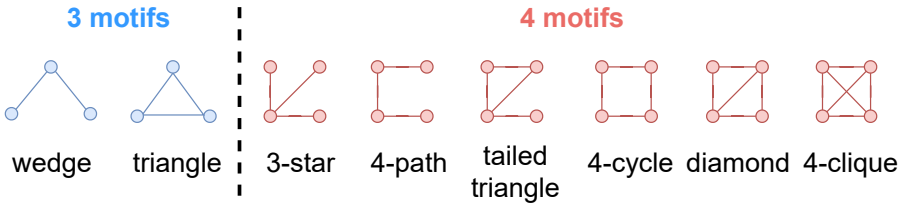



Figure 2.2: 3-vertex (left) and 4-vertex (right) motifs [12].

**Definition of GPM**. Given an undirected graph $\mathcal{G}$ and a set of patterns $S_p$={$P_1$, $P_2$, ...} by the user, GPM finds vertex-induced or edge-induced subgraphs in $\mathcal{G}$ that are isomorphic to any $\mathcal{P}$ in $S_p$. If the cardinality of $S_p$ is 1, we call it a single-pattern problem. Otherwise, it is a multi-pattern problem.

A pattern $\mathcal{P}$ is a small graph that can be defined explicitly or implicitly. An explicit definition specifies the vertices and edges of $\mathcal{P}$, whereas an implicit definition specifies the desired properties of $\mathcal{P}$. For explicit-pattern problems, the solver finds matches of $\mathcal{P}$ in $S_p$. For implicit-pattern problems, $S_p$ is not known in advance. Therefore, the solver must find the patterns as well as their matches during the search.

To avoid confusion, we call a vertex in the pattern $\mathcal{P}$ as a *pattern vertex* and denote it as $u_i$, and a vertex in the data graph $\mathcal{G}$ as a *data vertex* and denote it as $v_i$. Below are typical GPM problems from the literature [1], [5], [8], [12]:

- *$k$-clique counting* ($k$-CC): It counts the number of $k$-cliques in $\mathcal{G}$ ($k \geq 3$). A $k$-clique is a $k$-vertex graph whose every pair of vertices are connected by an edge. A triangle is a 3-clique.

- *Subgraph counting* (SC). It counts the number of edge-induced subgraphs of $\mathcal{G}$ that are isomorphic to a pattern $\mathcal{P}$.

- *$k$-motif counting* ($k$-MC): It counts the number of occurrences of all possible $k$-vertex patterns. Each pattern is called a *motif* [9], [13]. Fig. 2.2 shows all 3-motifs and 4-motifs.

This is also an example of a multi-pattern problem because we have to find all the subgraphs that are isomorphic to *any* pattern in a given set of patterns.

- *k-frequent subgraph mining* ($k$-FSM): Given $k$ and a threshold $\sigma_{min}$, this problem considers all patterns with fewer than $k$ edges and lists a pattern $\mathcal{P}$ if the support $\sigma$ of $\mathcal{P}$ is greater than $\sigma_{min}$. This is called a *frequent* pattern. If $k$ is not specified, it is set to $\infty$, meaning that it is necessary to consider all possible values of $k$. In $k$-FSM, vertices in $\mathcal{G}$ have application-specific labels.

Our work focuses on CC, SC and MC. For $k$-CC, vertex-induced and edge-induced subgraphs are the same. SC and FSM find edge-induced subgraphs, while $k$-MC looks for vertex-induced subgraphs. All problems seek to find explicit pattern(s) except FSM which finds implicit patterns. $k$-MC and FSM are multi-pattern problems, while the others are single-pattern problems.

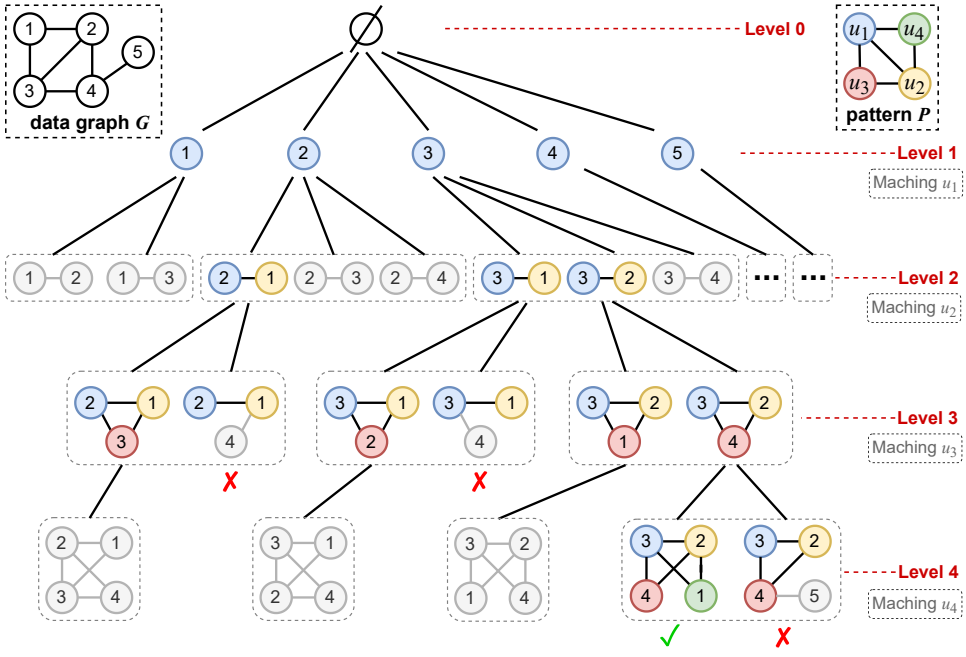



Figure 2.3: A search tree using vertex extension [8]. Vertex colors (not vertex labels) show the matching between data vertices and pattern vertices. The matching order is $\{u_1 \rightarrow u_2 \rightarrow u_3 \rightarrow u_4\}$. The symmetry order is $\{v_a > v_b, v_c > v_d\}$. Subgraphs in grey are ruled out by symmetry breaking. $\times$ shows the unnecessary extensions that are pruned by the matching order. ✓ shows the matched subgraph.

Exact GPM is solved by enumerating subgraphs in the data graph and searching for matches. The search space can be defined as a *subgraph tree*: each node in the tree is a subgraph of the data graph $\mathcal{G}$. A subgraph $H$ in level $l$ of the tree have $l$ vertices. The root (level 0) is an empty subgraph. A parent node $H$ at level $i$ can be *extended* to a child node $H'$, by adding a vertex/edge in $H$'s neighborhood in $\mathcal{G}$, i.e., $H' = H + \{v\}$, $v \in \mathcal{N}(H)$. Each leaf of the tree is a candidate of match, which is then compared with the pattern $\mathcal{P}$, to test if it is a match. Pruning schemes, e.g., matching order [3], [4], [8], symmetry breaking [7], [12], [14] and decomposition [15], [16], are applied to reduce the search space (i.e., prune the subgraph tree). Nevertheless, this search is extremely expensive, as the computational complexity increases exponentially in the size of the pattern.

The efficiency of a GPM algorithm depends heavily on how much we can prue the search tree. State-of-the-art GPM frameworks [3], [4] use *pattern-aware* search plans that leverage the properties of the pattern to prune the tree. A pattern-aware search plan consists of a *matching order* and *symmetry order*. We use the definitions from [8] below to introduce them.

**Matching order** is a total order that defines how the data vertices are matched to pattern vertices. This order is used to eliminate irrelevant subgraphs on-the-fly. As shown in Fig. 2.3, to find the `diamond` pattern, we use a matching order among pattern vertices: $\{u_1 \rightarrow u_2 \rightarrow u_3 \rightarrow u_4\}$, meaning that each vertex $v_1$ added at level 1 is matched to $u_1$; each vertex $v_2$ added at level 2 are matched to $u_2$, and so on. To search for matching candidates, there are connectivity constraints for the data vertices. For example, in `diamond`, since $u_3$ is connected to both $u_1$ and $u_2$, candidate vertices of $v_3$ must be found in the intersection of $v_1$ and $v_2$'s neighborhoods, i.e., $v_3 \in \mathcal{N}(v_1) \cap \mathcal{N}(v_2)$. The same constraint should also be applied to $v_4$. For a given pattern $\mathcal{P}$, there exist multiple valid matching orders. To choose the best performing matching order, prior works [3], [4], [6], [14], [17]–[20] have proposed various cost models to predict the performance of matching orders, and choose the one with the highest expected performance.

**Symmetry order** is a partial order enforced among data vertices for *symmetry breaking*,

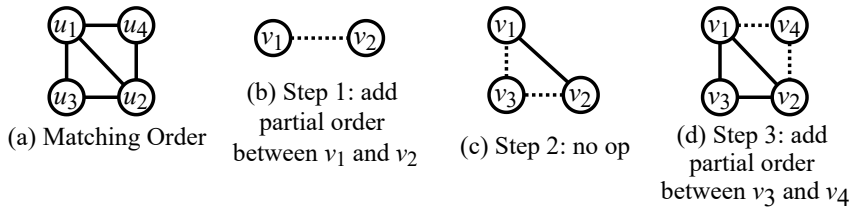



Figure 2.4: Generating symmetry order for `diamond` [21].

which removes redundant subgraph enumerations (a.k.a *automorphism* [12]), and thus guarantees that any match of $\mathcal{P}$ in $\mathcal{G}$ is found only once. For example, for `diamond`, we enforce that vertices added at level 1 must have larger ids than vertices added at level 2, i.e., $v_1 > v_2$. Thus, in level 2 of the tree in Fig. 2.3, the subgraph $\{2, 1\}$ is selected to be extended further, but subgraph $\{1, 2\}$ is pruned. Similarly we add a constraint that $v_3 > v_4$. So the symmetry order for `diamond` is $\{v_1 > v_2, v_3 > v_4\}$.

Algorithm 1 shows the pseudo code for exact `4-cycle` counting, which uses matching order and symmetry order.

**Algorithm 1** Exact `4-cycle` counting in GraphZero [14]

```
1: for each vertex v1 ∈ V do                                  ▷ match v1 to u1
2:     for each vertex v2 ∈ N (v1) do                         ▷ match v2 to u2
3:         if v2 ≥ v1 then break;                             ▷ symmetry breaking
4:         for each vertex v3 ∈ N (v1) do                     ▷ match v3 to u3
5:             if v3 ≥ v2 then break;                         ▷ symmetry breaking
6:             for each vertex v4 ∈ N (v1) ∩ N (v2) do         ▷ match v4 to u4
7:                 if v4 ≥ v0 then break;                     ▷ symmetry breaking
8:                 else count ++;                             ▷ do the counting
```

## 2.2 Approximate Graph Pattern Mining

Many real-world use cases do not require exact GPM solutions. For example, when we use motif (a.k.a graphlet) distribution as a "signature" (e.g., graph similarity) for social network analysis or fraud detection, it is quite sufficient to just provide approximate counts of the motifs. Also in FSM the users only want to find those *frequent* patterns whose occurrences are above some user specified threshold, or simply top-$K$ most frequent patterns, where estimated counts would be sufficient to give high quality solutions. Therefore, for all these

use cases, we can perform Approximate GPM (A-GPM) instead to substantially reduce the total amount of computation.

In this paper we focus on sampling based A-GPM approaches. Generally, such an approach first samples a portion of the graph, searches for match in the sample, and makes the estimation by scaling the sampled result. This process can be repeated multiple times to improve the confidence of estimation. Formally, given $\mathcal{G}$ and $\mathcal{P}$, an A-GPM solver aims to use a randomized ($\epsilon$, $\delta$)-approximation scheme, which estimates the number of (non-induced or induced) occurrences of $\mathcal{P}$ in $\mathcal{G}$ within a factor of $(1 \pm \epsilon)$, with probability at least $1 - \delta$, where $\epsilon$ and $\delta$ are user defined parameters. There are a large volume of studies on A-GPM applications, such as triangle counting [22]–[30], clique/cycle counting [29], [31], [32], motif counting [33]–[43], butterfly counting [44], frequent subgraph mining [45]–[49]. They all use sampling to reduce computation, though their sampling schemes are *customized* for the specific problems. These custom implementations do not offer system support, like automated termination, generic APIs, or choices in speed and error trade-off.

## 2.3 Sampling Schemes for GPM Problems

### 2.3.1 Neighbor Sampling (`NS`)

Algorithm 2 shows how neighbor sampling works. For each sample, it starts with sampling one edge from $\mathcal{G}$ uniformly at random (Line 3), and then repetitively samples one more edge from the neighborhood of the currently sampled edges (Line 5), until the size (number of sampled vertices) is the same as the pattern. It then does *closure check* (Line 7), i.e., check in $\mathcal{G}$ the existence of the *closing edges*, which form a match of the pattern together with existing edges. We can draw multiple samples (Line 1) in parallel, and average them for improved accuracy (Line 8).

For a given match $\hat{M}$ where $\hat{e_1}$ is the first edge, $Pr[\hat{M}] = \frac{1}{m \cdot \prod_2^{k-1} c_j}$, where $c_j = |\mathcal{N}(H_{j-1})|$, $H_{j-1}$ is the $E_{j-1}$ induced subgraph, and $\mathcal{N}(H)$ is the neighbor set of $H$. So the count is scaled

**Algorithm 2** Neighbor Sampling

1: **for each** sampler $i \in [1, N_s]$ **do**
2: $C_i \leftarrow 0, \alpha \leftarrow m$
3: Sample an edge $e_1$ from $\mathcal{E}$, let $E_1 \leftarrow \{e_1\}$
4: **for** $j$ in $[2, k-1]$ **do**
5: Sample a neighboring edge $e_j$ from $E_{j-1}$'s neighborhood
6: $E_j \leftarrow E_{j-1} \bigcup \{e_j\}$, $\alpha \leftarrow \alpha \times c_j$ ▷ $c_j$ is the neighborhood size
7: **if** closing edges for $(E_{k-1}, \mathcal{P})$ exist in $\mathcal{G}$ **then** $C_i \leftarrow \alpha$ ▷ closure check
8: $C' = \sum C_i / N_s$ is the estimated count

by $\alpha = m \cdot \prod_2^{k-1} c_j$ (Line 6). Apparently $\alpha$ varies in different samples, meaning matches are not equally likely to be sampled. Thus $\alpha$ is maintained for each sample and used for normalization.

NS has been widely used in A-GPM applications [26], [27], [40], [50] with customized optimizations. In NS, each time an arbitrary neighbor is sampled from $H_i$'s neighborhood, which may lead to a high failure rate in the closure check. Therefore, restrictions are added to make the sampled subgraph more likely be a match. For example, if $\mathcal{P}$ is a cycle, we can sample a path [34], i.e., restrict the next neighbor to be from the two endpoints' neighborhood. For an arbitrary pattern $\mathcal{P}$, we can do neighbor sampling following $\mathcal{P}$'s spanning tree [37]. Furthermore, automorphism check can be added to avoid redundant subgraphs [51]. More generally, we can sample multiple neighbors each time, instead of a single neighbor [51].

**Algorithm 3** Edge Sparsification

1: Randomly select a subset $E'$ of $p \times m$ edges from $E$, $m = |E|$, $0 < p \leq 1$
2: Generate an induced subgraph $G' = (V, E')$
3: $C'$ += ExactCounting($\mathcal{G}'$, $\mathcal{P}$) ▷ Exact counting on $\mathcal{G}'$
4: $C = C' \times p^{-l}$ is the estimated count in $\mathcal{G}$ ▷ $l$ is # of edges in $\mathcal{P}$

### 2.3.2 Subgraph Sampling

The idea of subgraph sampling is to sample a subset of vertices or edges from $\mathcal{G}$ with some probability $p$, to form a subgraph $G'$, and then do exact search in $G'$, and finally scale the count based on $p$. One popular way to sample a subgraph is to use the *graph sparsification* (GS) technique [52]–[56], which sparsifies $\mathcal{G}$ by randomly removing some edges. Bernoulli

Edge Sparsification (BES) [25] is such an example, as shown in Algorithm 3. For each edge in $\mathcal{G}$, include it with probability $p$ (Line 1) to get graph $G'$ (Line 2). For each match $M$ of $\mathcal{P}$ in $\mathcal{G}$, the probability that $M$ exists in $G'$ is $p^l$, hence the expected count in $G'$ is $C' = p^l \times C$. BES is simple and easy to implement, and can be trivially parallelized.

Color Sparsification [23] (CS), another GS approach, sparsifies $\mathcal{G}$ by first randomly assigning a color from $\{1, 2, \cdots, c\}$ to each vertex and then only preserving edges whose two endpoints are in the same color. The probability of choosing an edge is $p = 1/c$, and the probability of choosing a match is $p^{l-1}$. Hence the expected count in $G'$ is $C' = p^{l-1} \times C$. Apparently, the probability of preserving a match of $\mathcal{P}$ is higher than that in BES. In other words, it could meet the same error bound with more edges removed, thus less computation. The downside is that different edges are not independent of each other any more, which means potentially higher variance. Usually a single sampler is used for sparsification, though more samplers can be used to improve confidence.

Another way to sample a subgraph is Egonet Sampling [44], [57], in which an element (vertex or edge) of $\mathcal{G}$ is sampled and the *egonet*, i.e., the local neighborhood, of this element is extracted as a subgraph. It often requires many samplers.

One advantage of subgraph sampling is that a state-of-the-art exact counting algorithm can be applied on the sampled subgraph. But the cost is extra time and space to extract the subgraph(s), and the time complexity is still exponential in the size of pattern $k$. Subgraph sampling has only been used in A-GPM applications for specific patterns [23]–[25], [32], [44], [57], [58].

### 2.3.3 Other Sampling Schemes

There exist many sampling methods other than neighbor sampling and graph sparsification. We discuss some of the typical methods in the following. We do not compare with them in Chapter 6 because they do not provide either generalization or automated termination (with confidence). However, they can potentially be used to replace the GS engine in ScaleGPM.

We leave it as a future work.

**Color Coding**. It first colors each vertex in $\mathcal{G}$ using a color randomly chosen from $\{1,2,\ldots,c\}$ ($c \geq k$), and then counts *colorful* matches, i.e., every vertex in the matched subgraph has a unique color. The requirement of distinct colors allows for heavy pruning: the number of colorful matches $Z$ can be naturally determined by a dynamic programming based counting routine [35]. Color-coding is originally for finding paths or cycles [59], [60], and is then adapted for motif counting [42], [43], [61]. There also exist many parallel and distributed implementations [35], [62]–[64]. In addition, the colorful matches can be further sampled [37]–[39], instead of exactly counted, to reduce computation. Like `GS`, color-coding is also a coarse-grain scheme, which can be included in ScaleGPM.

**Loop Perforation**. SampleMine [51] proposed to perforate the nested `for` loops in GPM programs, with a certain probability $p_i$ for the $i$-th loop. The count is then scaled based on the $p_i$. SampleMine can be thought of as a generalization of the vanilla `NS` scheme, as it collectively samples multiple candidates, instead of a single one, at a time. Although it has larger sampling granularity than the vanilla `NS`, its granularity is still limited by the egonet of a vertex or edge, similar to Egonet Sampling [57]. More importantly, SampleMine does not provide a systematic way for sampling termination, which is instead hand-tuned by executing the sampling procedure multiple times and manually observing convergence (i.e. small variance).

**Other Schemes and Approaches**. Monte Carlo Markov Chain (MCMC) [28], [36], [41], [65] defines a random walk over the set of subgraphs until it reaches stationarity. MCMC has been used for motif counting. However, it has been shown that MCMC can be extremely inefficient as the random walk may take a huge amount of steps to reach stationarity [37]–[39].

## 2.4 Approximate GPM Systems

A-GPM systems, such as ASAP [10] and Arya [11], have been proposed to simplify A-GPM programming. These systems provide APIs to the users for them to easily compose various A-GPM applications. As opposed to those case-by-case customized implementations, an A-GPM system provides a generalized, sampling-based approximation method, for arbitrary patterns. Moreover, instead of hand tuning the key sampling parameters, e.g., the number of samples, in hand-implemented applications, these systems provide the Error-Latency Profile (ELP) method to automatically choose the values of the sampling parameters for each specific case, e.g., input data graph and pattern, to meet the user specified error bound.

Nevertheless, all the prior systems use fixed sampling schemes to make approximation. For example, ASAP uses the `NS` scheme, and implements `NS` in the *edge streaming* fashion [66]–[70] where edges are streamed in as a sequence, instead of loaded all at once, to save memory space. Arya is also based on `NS`, but adds pattern decomposition on top of ASAP, to reduce the amount of work in each sample for large, easy-to-decompose patterns. Since both ASAP and Arya are based on `NS`, they both suffer the shortcomings of `NS`, which are discussed in detail next.

# Chapter 3

# Understanding Sampling Tradeoffs

## 3.1 Termination Condition and Confidence

A major responsibility of an A-GPM system is to decide when the sampling can be terminated with enough confidence to meet the error bound. Existing A-GPM systems use error-latency profile (ELP), before the execution of the sampling procedure, to pre-determine the number of samples $N_s$ required. The execution is then terminated simply when $N_s$ samples have been drawn. However, we observe that the value of $N_s$ that ELP predicts vary dramatically across different runs of ELP. Fig. 3.1 shows the results of three runs of ELP. Each curve represents the prediction of one ELP run. We get predictions of $N_s$ as 5,260, 26,510 and

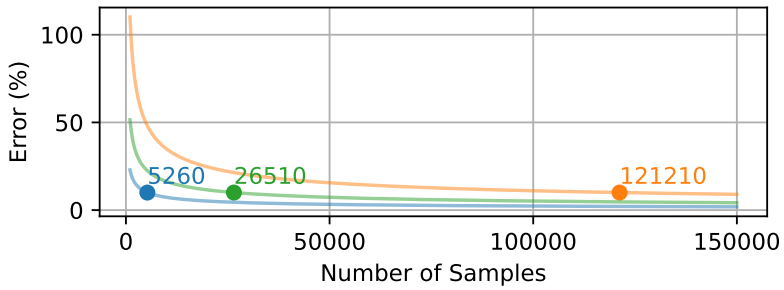



Figure 3.1: Three different runs (three curves) of Arya's ELP prediction given the `LiveJ` graph and `triangle` pattern. With an error bound of 10%, the curves give dramatically different prediction on the number of samples $N_s$: 5,260, 26,510 and 121,210. This leads to a 25× performance difference in the sampling execution phase.

121,210, respectively. This huge prediction difference leads to a 25× performance difference in the sampling execution phase.

More importantly, the ELP method for termination condition adds an indirection between the error estimation and the confidence. Originally, the required number of samples $N_s$ is derived from the Chernoff bound in ASAP [10] or Chebyshev's inequality in Arya [11]. For example, given $\epsilon$ and $\delta$, the lower bound in Arya is $N_s \geq \frac{K \cdot m \cdot \rho}{C \cdot \epsilon^2 \delta}$, where $K$ is a constant, $m = |E|$, $\rho$ is a pattern specific constant, and $C$ is the true count. The problem is, this bound contains the true count $C$, which is the output that is supposed to be estimated. Therefore, what ELP does is to essentially first estimate $C$ without theoretical confidence-error ($\delta$-$\epsilon$) guarantees, then feed it into the lower bound to get $N_s$. $N_s$ is then used to do sampling and estimate a more accurate $C$. Note that ELP estimates $C$ by sampling in a sparsified graph and iteratively updating the parameters and the estimates until convergence. Although the bound inequation used to calculate $N_s$ contains $\delta$, estimating $C$ by ELP *has no involvement of* $\delta$. It means the estimation of $N_s$ loses connection to the confidence. The root cause is the *circular dependency* between $C$ and $N_s$, i.e., $C$ is used to estimate $N_s$, and $N_s$ is used to estimate $C$, which is fundamentally unavoidable in the ELP based approach.

Due to the *circular dependency*, ELP provides only a heuristic rather than a strong theoretical bound. Another limitation of ELP is its own convergence speed. We observe that in some cases, ELP fails to converge within 10 hours (see details in Table 6.2).

## 3.2 Characterizing Neighbor Sampling

For the sampling based approximation approach, the estimation difficulty depends on the density and distribution of the matches of $\mathcal{P}$ in the graph $\mathcal{G}$. When there are plenty of matches, defined as the *dense case*, it is easier to make estimation than the *sparse case*, where there exist only a few matches. This is because it is more likely to draw a *successful* sample (i.e., find a match) if there are more matches, and the confidence to meet an error bound

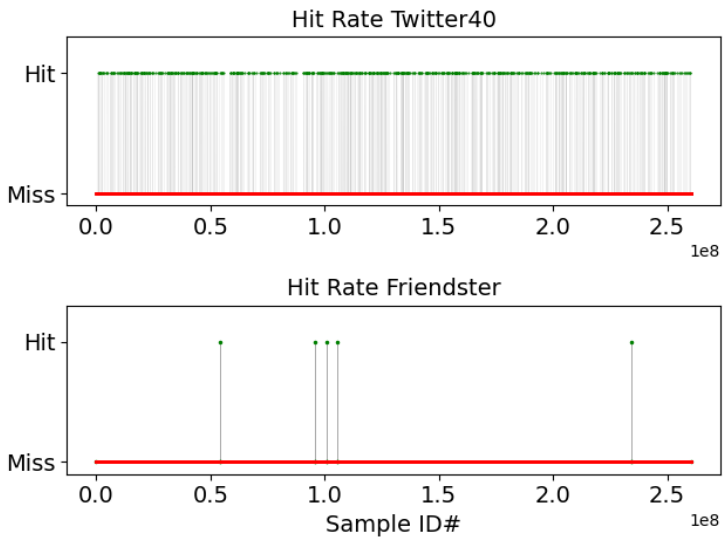
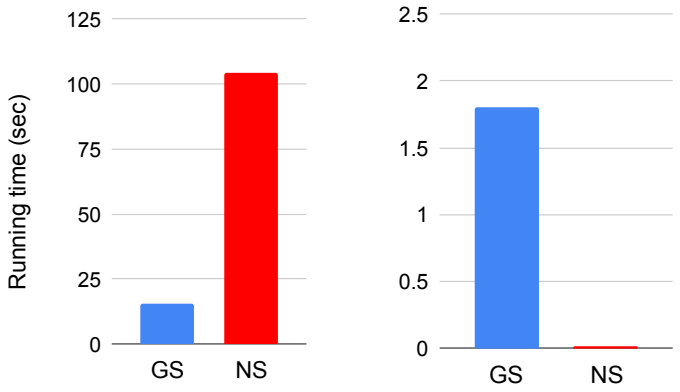



Figure 3.2: Sample hits and misses in Arya, on `Twitter40` (top) and `Friendster` (bottom) graphs. The pattern is `4-clique` for both. In total $10^8$ samples are drawn in both cases. Each green point is a hit sample, while each red point is a miss sample. For `Twitter40`, there are 7,033 hits with a hit rate of $7 \times 10^{-5}$. For `Friendster`, there are only 5 hits with a $5 \times 10^{-8}$ hit rate (i.e. needle in the hay).

depends on seeing enough number of successful samples. Note that the extremely sparse case is known as finding the *needle in the hay*.

We call a sample that finds a match a *hit*, otherwise a *miss*. Fig. 3.2 shows how the hits and misses are distributed in $1 \times 10^8$ samples drawn by Arya [11], for the `4-clique` pattern on graphs `Twitter` (top) and `Friendster` (bottom). The two graphs have similar sizes but quite different degree distribution (see maximum degree in Table 6.1), and thus different hit rates. There are 7,033 hits in `Twitter` ($7 \times 10^{-5}$ hit rate), while only 5 hits appeared in `Friendster` ($5 \times 10^{-8}$ hit rate) which is a typical needle-in-the-hay case. Since the execution time is roughly linear in $N_s$, this difference in hit rates would result in a $\sim$700$\times$ execution time difference in practice, assuming the same average time per sample.

It is known that `NS` works poorly in the sparse case, for example, when $\mathcal{P}$ is dense and $\mathcal{G}$ is sparse, and particularly in the case of needle in the hay [11]. One can expect that in this case, the closure check fails frequently, which means a large number of samples $N_s$ is required to meet a certain error bound. Moreover, dense patterns are particularly problematic for




(a) `house` on `fr` with 0.1% error (b) `5path` on `livej` with 2% error

Figure 3.3: Execution time variance of Neighbor Sampling (`NS`) and Graph Sparsification (`GS`), under the same error bounds.

Arya [11] which decomposes the pattern into sub-patterns, as a dense pattern would be split with many edgecuts, and checking closure is quite expensive. Arya thus suffers significant slowdown in those cases, and in some extreme cases it is even slower than the exact solution.

## 3.3 Coarse-grain vs. Fine-grain Sampling

A key difference between `NS` and `GS` is the granularity of each sample. Since each sample contains at most one match, we classify `NS` to be a *fine-grain* sampling scheme, as opposite to *coarse-grain* sampling schemes, each of whose samples can contain multiple matches. For `NS`, each sample is in the size of the pattern size $k$, and each sample task is lightweight. But we need a lot of samples, i.e., $N_s \gg 1$, to get a meaningful estimation, since each sample contains at most one match. In contrast, subgraph sampling schemes, including `GS`, are coarse grained. In `GS`, a sample is a sparsified graph, which could potentially contain many matches in it. As the sample granularity in `GS` is much larger, it performs better than `NS` when given a sparse case, as it is more likely to hit matches in a big region of the graph. However, this advantage comes at the cost of several drawbacks. First, each sample in `GS` is a much larger computational task, and the complexity is exponential in the pattern size. Second, as multiple matches appear in the same sample, `GS` may yield worse variance than

NS in the worst case.

To summarize, given the distinct characteristics in the input data (graph and pattern), none of these sampling schemes can always be the best solution. Fig. 3.3 compares the running time of NS with GS, when given the same error bound. On the left we mine the house pattern on the Friendster graph with an error bound of 0.1%, where GS is 6.7× **faster** than NS. On the right we mine the 5-path pattern in the Livej graph with an error bound of 2%, where GS is 10× **slower** than NS.

# Chapter 4

# Proposed Mechanisms and Optimizations

To achieve stable termination with confidence, we propose a novel on-the-fly convergence detection mechanism in Section 4.1 that is fundamentally different from the existing ELP approach. To improve hit rate in NS, we introduce eager-verify and prove it unbiased in Section 4.2. To further improve performance in handling needle-in-the-hay cases, we propose a hybrid method that adaptively selects the best-performing sampling scheme. For scheme selection, we establish cost models (a.k.a. performance models) for the NS (Section 4.3) and GS (Section 4.4) scheme to estimate their execution time, and select the faster one. We only focus on the two schemes, but this hybrid method can be extended to include other schemes in the future.

## 4.1 Online Convergence Detection

Due to the circular dependency issue (Section 3.1), the ELP method breaks the theoretical guarantee on confidence. Also, it is fundamentally hard for ELP to establish confidence because it is done before execution, and there is little information we can leverage. Therefore, we propose an on-the-fly approach to establish confidence. This is based on our observation in NS sampling procedures, that the estimation errors tend to converge over time (see Fig. 6.2b). Instead of predetermining the required number of samples offline (i.e. before execution), we

detect online if the estimates have converged, and then terminate the execution subsequently. However, convergence detection is not trivial. A straightforward method is to check if the difference of the error curve is small enough, i.e., within a fixed threshold. But this does not work because the termination point depends on the user defined confidence and error bound. The key challenge is then how to decide termination to meet error bound with confidence.

To address it, we propose to predict the error online periodically with confidence, and terminate when the predicted error is below the error bound. To make predictions with confidence, we collect online statistics during execution, which allows us to formally derive predicted errors based on the probability theory. Our key insight is that confidence can be established by the normal distribution of sampled means (formally proved in Theorem 1), as shown in Fig. 4.1. Specifically, we keep track of the *mean* $\mu$ of the estimated counts and the *standard deviation* $\sigma$ of the means, and then use $\mu$, $\sigma$ and $\delta$ to compute a relative error $\hat{\epsilon}$ using Eq. (4.1), where $\Phi^{-1}$ is inverse of the cumulative distribution function of the standard normal. Note that $\mu$ is also used as the estimate of the true count $C$.

$$\hat{\epsilon} = \frac{\Phi^{-1}(1 - \frac{\delta}{2})\sigma}{\mu} \tag{4.1}$$

When we detect that $\hat{\epsilon}$ is below the user specified error bound $\epsilon$, according to Theorem 1, we can safely terminate the sampling, and conclude that $\mu$ is an estimate of $C$, under an error bound $\epsilon$ with a confidence of $1 - \delta$. We refer our approach as NS-online.

**Theorem 1.** *Given $\delta$, $n$ samples $X_1, \ldots, X_n$ drawn by using the NS sampling scheme, and the mean of sampled counts $\mu = \frac{1}{n}\sum_{i=1}^{n} X_i$, let $C$ be the true count and $\hat{\epsilon}$ be the estimated error computed by Eq. (4.1). As $n \to \infty$, the probability of the true relative error being smaller than the estimated relative error is $\mathbb{P}\left(\frac{|\mu - C|}{C} < \hat{\epsilon}\right) = 1 - \delta$.*

The proof of the above theorem is provided by Zixiang Zhou in our paper [71] (see Appendix A.1).

Algorithm 4 shows the NS-online algorithm. Each time we draw a sample (Line 4), the

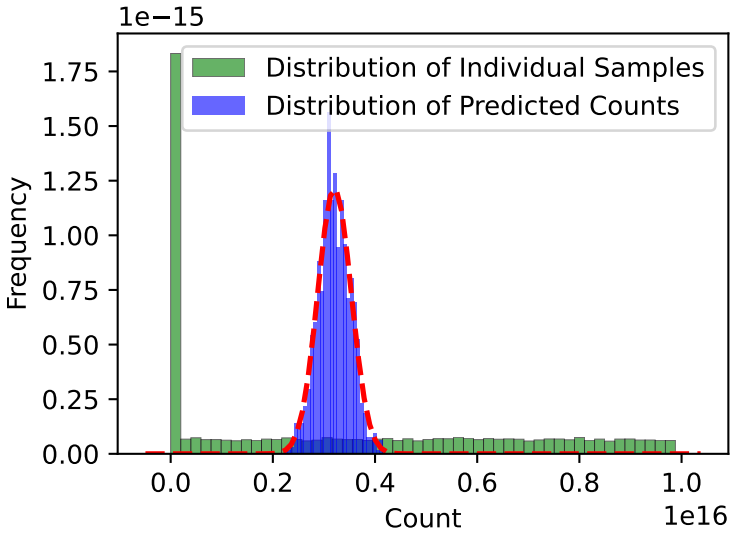



Figure 4.1: The normal distribution of the means of sampled counts (i.e. our predicted counts) using neighbor sampling (NS). We ran NS to collect $10^6$ samples on LiveJ, 4-clique. We obtained a predicted count by taking the mean of a random subset of 100 of these underlying samples. We simulated 1000 of these predicted counts. Although the underlying distribution of the sampled counts (green bars) is not a normal distribution, their means (purple bars), which are our predicted counts, do follow a normal distribution (dashed red line).

only information we need to keep track of is the accumulated sum $\sum_{i=1}^{n} X_i$ (Line 5) and the accumulated squared sum $\sum_{i=1}^{n} X_i^2$ (Line 6). At the end of each interval (i.e. $W$ samples in Line 3), we compute the standard deviation $\sigma$ (Line 10) and predict the error $\hat{\epsilon}$ (Line 11). If $\hat{\epsilon}$ is below the user's error bound (Line 2), sampling is then terminated and it reports the estimated count $\mu$.

## 4.2 Eager Verify for Neighbor Sampling

A major drawback of NS in prior systems is the low hit rate in dealing with sparse cases. By looking at individual samples, we find that most of these samples fail to pass the closure check (Line 7 in Algorithm 2). Therefore we looked deeper into the failures (i.e. missed samples). Our key observation is that many of the failures have been unpromising candidates even at the early stage of the sampling. For example, if we search for 6-cliques, and the

**Algorithm 4** NS-online convergence detection

sum $\leftarrow 0$, squaredSum $\leftarrow 0$, $n = 0$, $W \leftarrow N_{min}$
**while** $\hat{\epsilon} > \epsilon$ **do**
  **for each** sampler $i \in [1, W]$ **do** ▷ $W$ is the window size
    $X_i \leftarrow$ DrawASample() ▷ $X_i$ is the $i$-th sampled count
    sum $\leftarrow$ sum $+X_i$ ▷ $\sum_{i=1}^{n} X_i$
    squaredSum $\leftarrow$ squaredSum $+X_i * X_i$ ▷ $\sum_{i=1}^{n} X_i^2$
    $n \leftarrow n + 1$
  $\mu \leftarrow$ sum / $n$ ▷ mean of sampled counts
  var $\leftarrow$ squaredSum / $n - \mu * \mu$ ▷ variance of sampled counts
  $\sigma \leftarrow$ sqrt(var/$n$) ▷ standard deviation
  $\hat{\epsilon} \leftarrow \Phi^{-1}(1 - \frac{\delta}{2}) * \sigma/\mu$ ▷ predicted error

first four vertices in the sample does not form a 4-clique, this sample is impossible to form a 6-clique. Therefore, in general, the low hit rate in ASAP and Arya is because the closure check is delayed to the very last step, which we refer to as *lazy-verify*. Based on this understanding, we propose a *eager-verify* approach to improve NS performance. The key idea is to sample from promising candidates by verifying the pattern's connectivity constraints as early as possible. This strategy has two advantages. First, since unpromising candidates are pruned at early stage, and we only sample from promising candidates, each sample is more likely to succeed, i.e., hit a match. Second, if a sample starts from an edge whose neighborhood contains no or very few matches (unlikely to hit), eager-verify can minimize the work as this sample would fail at its early stage, while lazy-verify will have to proceed until the end and fail.

**Algorithm 5** NS-base for 4-cycle

**for each** sampler $i \in [1, N_s]$ **do**
  $e(v_0, v_1) \leftarrow$ SAMPLE($\mathcal{E}$) , $\alpha \leftarrow 0$ ▷ sample an edge $(v_0, v_1)$
  $A \leftarrow N(v_0) \cup N(v_1)$ - $\{v_0, v_1\}$ ▷ set union and difference
  **if** |A| = 0 **then** break
  $v_2 \leftarrow$ SAMPLE($A$) ▷ sample node $v_2$ from set $A$
  $B \leftarrow N(v_0) \cup N(v_2)$ - $\{v_0, v_1, v_2\}$ ▷ set union and difference
  **if** |B| = 0 **then** break
  $v_3 \leftarrow$ SAMPLE($B$) ▷ sample node $v_3$ from set $B$
  **if** edge $(v_0, v_3)$ exist in $\mathcal{G}$ **then** ▷ check closure
    $\alpha \leftarrow m * |A| * |B|/16$ ▷ scaling factor

The challenge in implementing eager-verify is how to avoid unpromising candidates but still retain unbiasedness in sampling. Based on the subgraph tree abstraction (Section 2.2),

**Algorithm 6** NS-prune for `4-cycle`

1: **for each** sampler $i \in [1, N_s]$ **do**
2: $e(v_0, v_1) \leftarrow$ SAMPLE($\mathcal{E}$) , $\alpha \leftarrow 0$ ▷ sample an edge $(v_0, v_1)$
3: $A \leftarrow N(v_1)$ - $\{v_0, v_1\}$, bound by $v_0$ ▷ set difference
4: **if** |A| $= 0$ **then** break
5: $v_2 \leftarrow$ SAMPLE($A$) ▷ sample node $v_2$ from set $A$
6: $B \leftarrow N(v_0)$ & $N(v_2)$, bound by $v_1$ ▷ set intersection
7: **if** |B| $= 0$ **then** break
8: $v_3 \leftarrow$ SAMPLE($B$) ▷ sample node $v_3$ from set $B$
9: $X_i \leftarrow m * |A| * |B|$ ▷ sampled count

our key finding is that each leaf in the tree corresponds to a unique path. As long as the pruning does not change this one-to-one mapping, we can prove that the sampler is unbiased. We find that two typical pruning techniques, symmetry breaking [14] and matching order [8], meet this requirement. There exist other pruning schemes in the literature [16], [72], which could be applied as well, but we leave this as a future work. We refer NS used in ASAP as *NS-base*, and NS with the two pruning techniques as *NS-prune*. We first give an example to show how pruning avoids unpromising candidates, and then prove NS-prune an unbiased estimator in Appendix A.3. Algorithm 5 and Algorithm 6 show the pseudo code for finding `4-cycle` using NS =base and NS-prune respectively. In Line 6 we compute a set intersection $N(v_0)$ & $N(v_2)$ which is the candidate set of the third vertex $v_3$, because $v_3$ is a common neighbor of $v_0$ and $v_2$ in the `4-cycle` pattern. In contrast, in NS-base, because both $v_0$'s and $v_2$'s neighbors are possible candidates, $v_3$ is sampled from set union $N(v_0) \cup N(v_2)$, and in the final step it checks closure between $v_3$ and $v_2$ if $v_3$ is from $v_0$'s neighborhood, otherwise it checks closure between $v_3$ with $v_0$. If the closure check fails, it is a miss. However, in NS-prune the closure check is unnecessary because $v_3$ is guaranteed to be connected to $v_0$ and $v_2$, and thus much more likely to hit a match.

The proof that NS-prune is an unbiased estimator is provided by Zixiang Zhou in our paper [71] (see Appendix A.3).

Note that the scaling factor in NS-prune tends to be much smaller than that in NS-base, because it involves the sizes of intersection instead of union sets. This results in lower variances and more stable and faster convergence, as we will show in Chapter 6.

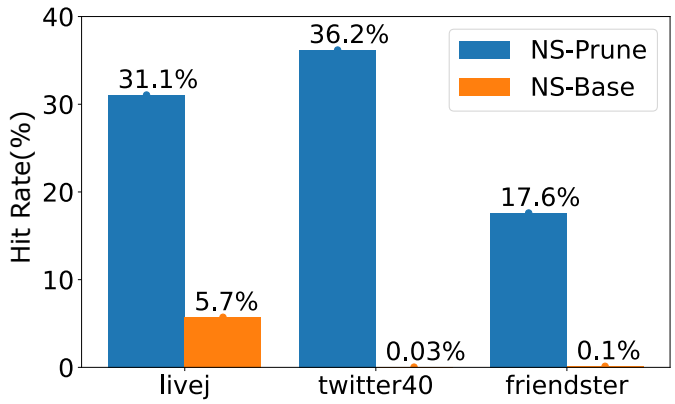



Figure 4.2: Comparing the hit rate of NS-prune with NS-base, with the 4-clique pattern on various graphs. [71]

## 4.3 Cost Model for Neighbor Sampling

In NS, the total work is $\sum_{i=1}^{N_s} W_i$, where $W_i$ is the work of the $i$-th sample $s_i$. Given that $N_s \gg 1$ in the NS scheme, it is reasonable to assume that the NS execution time is linear in the number of samples $N_s$. So we can predict the execution time as $t_{tot} = t_{avg} * N_s = c * W_{avg} * N_s$, where $W_{avg}$ is the average work per sample, and $c$ is a hardware specific constant factor to translate work to time. We estimate $N_s$ by profiling Section 5.3. As this is only used for performance prediction, it does not affect error and confidence. To estimate $c$, a simple profiler can be run on each hardware machine to determine this constant scale factor. This profiling overhead is negligible, as it can be determined by running only once per machine or once per graph on only a small number of points.

The challenge, however, is to estimate $W_{avg}$ for the given $\mathcal{G}$ and $\mathcal{P}$, as $W_i$ varies for different samples. In fact $W_i$ depends on the local neighborhood structure of $s_i$. More specifically, the task of each sample is a sequence of set operations and sample operations, for example, see Algorithm 6. Given a specific pattern $\mathcal{P}$, this sequence of operations is fixed, but each set operation in the sequence may take different (worse-case) time depending on the cardinality of the input sets, which overall depends on the structure of $\mathcal{G}$. So it is difficult to get an accurate estimation of $W_{avg}$, i.e., the slope the linear relationship, without really

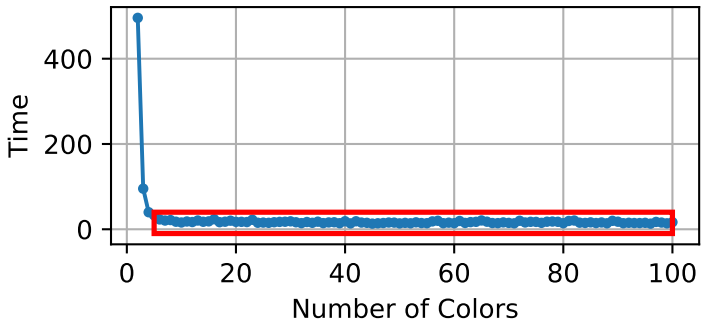



Figure 4.3: The execution time for 6clique-Friendster using Color Sparsification under different numbers of colors. Each point is one run. Red region is the stabilization window.

running NS.

At each break point (e.g., Line 4 in Algorithm 5), we must consider the possibility of an early-exit from the sampling procedure. Each break point is triggered based on the probability of being empty based on the candidate set size. However, it can be hard to predict this probability in practice, as certain graphs have many dense clustered areas, when sampled in few breakpoints occur, whereas others frequently terminate early. To address this, we model the performance as a *performance cone* (see example in Fig. 6.6) consisting of two slopes instead of one, where the performance is upper bounded by none of the breakpoints triggering (completing the full work of the sampling procedure) and lower bounded by the first break point.

## 4.4 Cost Model for Graph Sparsification

GS running time contains two parts, one is the **preprocessing** time to generate the sparsified graph $G'$, the other is the time spent on **exact search** in the sparsified graph $G'$.

Preprocessing involves looping over every edge in order to remove edges. Therefore, the total work is $|E| \cdot (w_1 + p \times w_2)$, where $p$ is the probability that an edge is kept, $w_1$ is the work on every edge, $w_2$ is the work on kept edges. In our implementation $w_1$ is one read operation, $w_2$ is one write operation.

For exact search in $G'$, our estimation is again based on set operations. But instead of a sequence of operations in NS, the work in GS consists of nested loops, each of which corresponds to one vertex in the pattern and iterates over the candidates of that vertex. The candidate vertex set is computed by set operations. If we describe a GPM algorithm as a sequence of nested for loops $M = (X_1, \ldots, X_n)$, which refine the candidate set to a possible match. Each nested loop can be described as $X_j = (o_j, i_j)$. Where $o_i$ is the number of operations performed in the inner body of that loop to refine the candidate set. $i_j$ is the number of iterations for the $j$th loop. $i_j = |Z_i|$, the size of the candidate set at the $j$th level. $o_i$ is the work of the operations to generate the candidate set $|Z_{i+1}|$. Then the total work can be described as $\prod_j^{i_j} o_j = \prod_j^{|Z_j|} o_j$. To estimate $i_j$ and $o_j$, we need to estimate the cardinality of each candidate set $Z$.

If $Z = A - B$, i.e., set difference, $|Z|$ is bounded by $|A|$, which can be simply estimated as the average degree, $|V|/|E|$. If $Z = A \cap B$, i.e., set intersection, $|Z|$ is bounded by $\min(|A|, |B|)$. To get a better bound, we can use the method in GraphPi [6] and estimate $|Z|$ as $|V| \cdot p_1 \cdot p_2^{n-1}$, where $p_1 = \frac{2 \cdot |E|}{|V|^2}$, $p_2 = \frac{T \cdot |V|}{2 \cdot |E|^2}$ and $T$ is the triangle count in $\mathcal{G}$. Intuitively, $p_1$ is the probability of any pair of vertices being neighbors. $p_2$ is the probability of any pair of vertices in a neighborhood being directly connected to each other. To use this estimation for GS, we need to make the following adjustments. Note that in a sparsified graph $G'$, $|V'| = |V|$ and $|E'| = |E| \times p$, and we have $T' = T \times p^2$ as discussed in Section 2.3.2. Then if $Z = A - B$, $|Z|$ is estimated as $\frac{|V|}{|E| \cdot p}$. If $Z = A \cap B$, $|Z|$ is estimated as $|V| \cdot p_1 \cdot p_2^{n-1}$, where $p_1 = \frac{2 \cdot |E| \cdot p}{|V|^2}, p_2 = \frac{T \cdot |V|}{2 \cdot |E|^2 \cdot p^4}$.

# Chapter 5

# System Design and Implementation

We give an overview of the ScaleGPM system in Section 5.1, describe details of the GS engine in Section 5.2 and our proposed profiling mechanism in Section 5.3, and other implementation details in Section 5.4.

## 5.1 System Overview and Interface

Fig. 5.1 illustrates the major components in our system. ScaleGPM is composed of a fast profiler, two cost models and two execution engines for NS and and GS respectively. The cost models have been described in Section 4.3 and Section 4.4 respectively. The NS engine is enhanced with our novel convergence detection mechanism (Section 4.1) and is significantly improved by our proven unbiased optimizations (Section 4.2). Our GS engine is the first generalized color sparsification for arbitrary patterns, as all prior GS-based work are customized for specific patterns, e.g., triangle. To generalize the GS approach for an arbitrary pattern $\mathcal{P}$, we need to (1) generate a pattern specific exact search program, (2) determine the scaling factor for $\mathcal{P}$ and (3) determine the values of its key parameter, i.e., the sparsify probability $p$ or the number of colors $c$ ($c = \frac{1}{p}$). For (1) we can leverage the state-of-art compiler based approach [14]. For (2) we explain in Section 5.2. For (3) our fast profiler in Section 5.3 determines it.

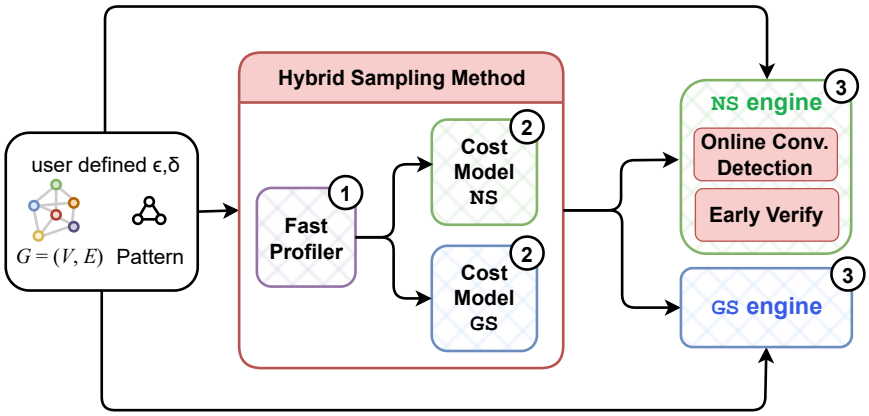



Figure 5.1: ScaleGPM system overview. The three red boxes are our proposed novel techniques: online convergence detection, early pruning and hybrid sampling. `NS`: neighbor sampling. `GS`: graph sparsification. The system execution flow is ① fast profiler estimates input parameters (e.g., `#colors`, `#samples`), ② cost models predict performance and select from `NS` and `GS` sampling schemes, and ③ the selected (`NS` or `GS`) engine is invoked to conduct sampling.

To meet various accuracy requirements in applications, ScaleGPM provides two modes for the user to choose, *strict mode* (default) and *loose mode*. The strict mode uses only the `NS` engine with online convergence detection to guarantee high confidence. In this mode the fast profiling and cost models are bypassed. The loose mode, however, employs our proposed hybrid approach. It uses the fast profiler and cost models to determine if the `GS` or `NS` engine is used. When comparing the predicted performance of `NS` and `GS`, ScaleGPM uses a thresholding mechanism to check if the predicted performance of `GS` overlaps with the performance cone of `NS`, i.e., it is either entirely above or entirely below the cone. If not, ScaleGPM chooses the faster one and activates the corresponding engine. Otherwise, the two schemes should perform similarly well, hence ScaleGPM chooses `NS` to guarantee high confidence.

## 5.2 Tradeoff in the `GS` Engine

For simplicity, we discuss edge sparsification, while color sparsification is similar. The estimated count is $\hat{C} = Y \cdot p^{-l}$, where the random variable $Y$ denotes the number of matches in the sampled graph $G'$, which is sparsified from $\mathcal{G}$ with probability $p$. For each match $M$

in $\mathcal{G}$, the probability that $M$ exists in $G'$ is $p^l$, hence the expected number of matches in $G'$ is $\mathbb{E}[Y] = p^l C$, i.e., $\mathbb{E}[Y \cdot p^{-l}] = C$, so we have $\mathbb{E}[\hat{C}] = C$. Although the edges are sampled independently, the matches are not. Consider an edge $e$ shared between two matches $M_i$ and $M_j$ of $\mathcal{P}$. When $e$ gets removed during sparsification, necessarily both $M_i$ and $M_j$ will not be counted. So the Chernoff bounds do not apply directly for GS. We can use Chebyshev's bounds instead. $\mathbb{V}ar[\hat{C}] = C \cdot (p^{-l} - 1) + \mathbb{C}ov$, where $\mathbb{C}ov = \sum_{z=2}^{k-1} t_z \cdot (p^{1-z} - 1)$ and $t_z$ is the number of pairs of matches that share $z$ vertices. The variance depends on both the number of matches $C$ in $\mathcal{G}$ and the number of pairs of matches that share one or more edges.

Apparently, the key knob to tune accuracy is $p$. As we increase $p$, the variance decreases and accuracy increases. Since GS speed is insensitive to the number of colors $c = \frac{1}{p}$ within a wide stabilization window (Fig. 4.3), we can pick a large $p$ to achieve better accuracy. Note that the variance increases exponentially in the pattern size $k$. This means larger patterns are more difficult to estimate accurately, and so we may need a larger $p$, to guarantee the same error bound.

## 5.3 Fast Profiling for Cost Models

To predict performance in our cost models, we need the number of samples $N_s$ for NS and the number of colors $c$ for GS. Our fast profiler is used to quickly determine the values of these parameters. Note that $N_s$ here is only used in the NS cost model, not used for NS sampling (as our NS-online does not need to predict $N_s$). The profiler first generates a sparsified graph $G'$, e.g., 10% of the original graph. Then, it runs our NS-online engine with an internal error bound (50%) and confidence (99%). By detecting convergence, we can determine $N_s = \frac{N_o \cdot \hat{\epsilon} \cdot \mu \cdot \rho(P,G)}{\overline{S} \cdot \epsilon^2 \cdot \rho(P,G')}$, where $N_o$ is the number of samples NS-online converged with, $\mu$ is the count returned by NS-online, $\overline{S}$ is the scaled count from $G'$ to $G$ (see 2.3.2), $\hat{\epsilon}$ is the final predicted error by NS-online, $\rho(P, G)$ is the probability of sampling pattern $P$ in $G$, which is used in prior systems [10], [11], and determined by properties of $G$, e.g., $\Delta$ and $|E|$.

## 5.4 Parallel Implementation Details

Our NS and GS engines are both parallelized. In GS, because sparsification creates non-overlapping subgraph partitions, each partition can be searched independently. Within each partition, we further parallelize it over each vertex in the subgraph, which provides enough parallelism. In NS, as each sample is an independent task, it can be embarrassingly parallelized across samples. Note that our NS-online requires a barrier synchronization at the end of each interval $W$ (Line 3 in Algorithm 4). If $W$ is too small, we have limited parallelism and too much synchronization overhead. But if $W$ is too large, we may end up with drawing more samples than necessary. Thus, we want $W$ to be large enough to maximize parallelism and minimize synchronization overhead, and small enough to minimize redundant work. We set $W$ to be a fixed percent of the estimate of $N_s$ returned from the fast profiler (e.g. 10% of $N_s$).

# Chapter 6

# Evaluation

We implement ScaleGPM in C++ and OpenMP for parallelization. We use ScaleGPM-NS, ScaleGPM-GS, ScaleGPM-HY to represent the NS, GS, and hybrid mode of our system respectively. In this evaluation we focus on comparing with prior A-GPM systems that provide both generalization and automation (see Section 2.3.3 for discussion on non-systematic solutions). We compare ScaleGPM with the state-of-the-art A-GPM system, Arya [11], and exact GPM system, GraphZero [14]. We do not include ASAP since Arya always outperforms ASAP. We test on a 3.0 GHz, 48-core (2-sockets, 24 cores per socket) Intel CPU without hyperthreading, with up to 1TB of memory. Table 6.1 shows the graphs used in our experiments, which are representative real-world graphs with varying sizes and topology characteristics. In ScaleGPM, graphs are represented in the Compressed Sparse Row (CSR) format. We evaluate two types of GPM tasks: subgraph counting (edge-induced) and motif counting (vertex-induced). For subgraph counting, we test on patterns including $k$-cliques, 5-path, house, and dumbbell. We do not include even larger non-clique patterns, because we can not verify the errors as their exact counts are unknown for most of the graphs. In all the experiments, we time out at 10 hours. We then conservatively use 10 hours for the timed-out cases when calculating speedups.

We first compare the overall performance of ScaleGPM with state-of-the-art systems in

| Graph | Source | \|V\| | \|E\| | Avg deg. | Max deg. |
|---|---|---|---|---|---|
| lj | liveJournal [73] | 4.8M | 43M | 17.7 | 20,333 |
| tw | twitter40 [74] | 42M | 2.4B | 57.7 | 2,997,487 |
| fr | friendster [75] | 66M | 3.6B | 55.1 | 5,214 |
| uk | uk2007 [76] | 106M | 6.6B | 62.4 | 975,419 |
| gsh | gsh-2015 [77] | 988M | 51B | 52.0 | 58,860,305 |
| cw | clueweb12 [77] | 978M | 75B | 76.4 | 75,611,696 |

Table 6.1: Data graphs (symmetric, no loops or duplicate edges). Maximum degrees are smaller when orientation is applied for cliques.

| **Pattern** | | 3-**clique** (triangle) | | | **4-clique** | | | **6-clique** | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Graph** | | Lj | Tw | Fr | Lj | Tw | Fr | Lj | Tw | Fr |
| ScaleGPM-NS | time (sec) | **0.001** | **0.046** | 0.026 | **0.003** | **0.068** | **0.090** | **0.059** | **0.707** | **1.132** |
| | hit rate | 18% | 55% | 53% | 7.3% | 46% | 31% | 6.4% | 46% | 16% |
| | # samples | 1e4 | 2e4 | 1e4 | 8e4 | 2e5 | 5e4 | 2.1e6 | 4.1e6 | 7.7e7 |
| Arya | time (sec) | 0.014 | 1.193 | **0.017** | 94.2 | 9656.9 | 2057.7 | TO | TO | TO |
| | hit rate | 10.6% | 3.9% | 2.9% | 3.2e-3% | 2.7e-3% | 1.7e-6% | – | – | – |
| | # samples | 4.3e4 | 6.3e4 | 4.6e4 | 3.3e8 | 2.6e8 | 5.1e9 | × | × | × |
| GraphZero | time (sec) | 0.434 | 58.0 | 83.0 | 2.1 | 4004.5 | 73.1 | 2502.5 | TO | 160.4 |

Table 6.2: $k$-clique performance (10% error). TO: timed out. Arya uses ELP. ×: ELP does not converge. Fastest time in bold.

Section 6.1. We show how our convergence detection performs in Section 6.2, and the accuracy the NS and GS cost models in Section 6.3. We discuss system efficiency in Section 6.4.

## 6.1 Sampling Performance vs. State-of-the-Art

We compare sampling performance with Arya and GraphZero. For the hybrid mode we discuss its profiling time in Section 6.4. For ScaleGPM-GS, we do include the preprocessing time spent on sparsifying the data graph. We use an error bound of 10% and confidence of 99%, which is the common practice in ASAP and Arya.

Table 6.2 compares the $k$-clique ($k$=3,4,6) running time of ScaleGPM with Arya and GraphZero. Overall cliques, ScaleGPM achieves a geomean average speedup of 2747× (up to 610,169×) against Arya, and 4,045× over (up to 65,525×) GraphZero respectively. In general, the speedup of ScaleGPM-NS over Arya comes from two parts. First, our online convergence detection gives stable and precise termination condition (demonstrated later in Fig. 6.1 and Fig. 6.2a), while ELP in Arya could suggest very conservative termination

| Pattern | | 5-path | | | 5-house | | | 6-dumbbell | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Graph | | Lj | Tw | Fr | Lj | Tw | Fr | Lj | Tw | Fr |
| ScaleGPM-NS | time (sec) | **0.004** | **0.91** | **0.10** | **0.008** | **944.6*** | **0.159** | **0.017** | **159.3** | **0.331** |
| | hit rate | 29% | 99% | 90% | 43% | 43% | 15% | 16% | 43% | 32% |
| | # samples | 3e4 | 1e4 | 2e4 | 5e4 | 1.9e7 | 3.3e5 | 1.5e5 | 1.6e7 | 6.9e5 |
| Arya | time (sec) | 4.422 | 13.78 | 77.93 | 395.6 | 1297.4 | TO | 346.2 | 4644.4 | TO |
| | hit rate | 0.04% | 5.8e-03% | 1.7% | 2.6e-3% | 0.02% | – | 0.04% | 2.2e-3% | – |
| | # samples | 1.1e7 | 1.4e7 | 3.4e5 | 1.5e9 | 6.8e7 | × | 2.1e8 | 1.1e9 | × |
| GraphZero | time (sec) | 262.1 | TO | TO | TO | TO | TO | TO | TO | TO |

Table 6.3: Non-clique pattern performance (10% error). TO: timed out. ×: ELP does not converge. * GS is selected by our hybrid method.

conditions that do way more samples than necessary (see Fig. 6.3). Second, our NS-prune approach dramatically improves the hit rate over NS-base and thus the total number of samples is reduced. This is evidenced in Fig. 4.2 and further confirmed in Fig. 6.2b. The significant speedups over Arya are expected because (1) Arya is based on pattern decomposition and cliques are hard to decompose, (2) cliques are more likely to fall in the needle-in-the-hay cases which Arya handles poorly. The speedups are further evidenced by the hit rate and number of samples $N_s$ used. In particular, for 4-clique, Arya requires 3 to 5 orders of magnitude more samples, while its hit rates are extremely low, e.g., $1.7\times10^{-6}$% for Fr. Notably, Arya is 28× slower than GraphZero for the sparse case 4-clique on Fr. Moreover, Arya's ELP can not converge within 10 hours for the even more sparse case, 6-clique on Fr, emphasizing the limitation of ELP. For triangle on Fr, ScaleGPM-NS is slightly slower than Arya due to synchronization overhead.

Table 6.3 compares performance on large, non-clique patterns, including 5-path, 5-house and 6-dumbbell. Note that it is well known that for exact GPM solvers (e.g. GraphZero) it is much more expensive to search for these patterns than cliques, as they are sparser and their sizes are beyond 4. We observe that GraphZero is timed out for most of the cases, which shows that it is critical to use approximation for large (sparse) patterns. Arya enjoys fairly good speedups over GraphZero. This is because, compared to cliques, these patterns are easier to decompose, which is the case that favors Arya. However, for Fr, Arya still suffers extremely high $N_s$ and low hit rate, and hence runs more than 10 hours for 5-house and

| Pattern | 3-motif | | | 4-motif | | |
|---|---|---|---|---|---|---|
| **Graph** | Lj | Tw | Fr | Lj | Tw | Fr |
| ScaleGPM-NS | **0.004** | **0.5** | 0.14 | **0.06** | **82.7** | **0.2** |
| Arya | 0.020 | 1.9 | **0.08** | 231.63 | 13180.2 | 6157.9 |
| GraphZero | 1.283 | 16316.2 | 242.62 | 1927.27 | TO | TO |

Table 6.4: Motif counting running time (sec) with 10% error. TO: timed out.

| Pattern | triangle | | | 4-clique | | | 5-path* | | |
|---|---|---|---|---|---|---|---|---|---|
| **Graph** | uk | gsh | cw | uk | gsh | cw | uk | gsh | cw |
| ScaleGPM-NS | **0.09** | **0.28** | **0.19** | **0.11** | **0.88** | **1.09** | **0.1** | **11.2** | **156.0** |
| Arya | 0.2 | 0.7 | OoM | 175.5 | × | OoM | 4.2 | × | OoM |
| GraphZero | 73.3 | 153.6 | 198.2 | TO | TO | TO | TO | TO | TO |

Table 6.5: Running time (sec) on huge graphs. TO: timed out. ×: ELP does not converge. * the true count is unknown, so accuracy is not verified.

6-dumbbell. In all these cases, ScaleGPM significantly improves hit rates and reduces $N_s$, which lead to fast convergence speed. Across these patterns, ScaleGPM achieves an geomean average of 599× and 27,641× speedup against Arya and GraphZero respectively. Note that for Tw on house, as shown in Table 6.6, ScaleGPM-HY selects the GS engine in this case, since GS is predicted to be faster, based on our prediction on the number of samples and time per sample.

Table 6.4 reports the motif counting performance. Note that in motif counting, we look for vertex-induced subgraphs, unlike cases in Table 6.3 which search for edge-induced subgraphs. The key difference in computation is that we need both set intersection and set difference to find vertex-induced subgraphs, but only set intersection is needed for edge-induced subgraphs. Despite more computation needed, ScaleGPM is still significantly faster than Arya. Across motifs, ScaleGPM achieves a geomean speedup of 125× speedup against Arya, and 10563× speedup over GraphZero.

Table 6.5 compares performance on huge graphs, i.e., uk2007 (uk) gsh-2015 (gsh) and clueweb12 (cw). Note that Arya mostly runs out of memory for cw because (1) it has to maintain the set union results (for each of the parallel threads) in memory, and (2) its internal representation of the graph stream is implemented in COO-like format, which is less compact than CSR. ScaleGPM achieves a geomean average 130× compared to Arya and a 7,245×

| Pattern | 8-clique | | 9-clique | | 5-house |
|---|---|---|---|---|---|
| Graph | lj | fr | lj | fr | tw |
| ScaleGPM- NS | 3.0 | 1663.6 | 17.4 | 8358.0 | 944.6 |
| ScaleGPM- GS | **0.7** | **43.4** | **2.1** | **134.9** | **200.1** |

Table 6.6: Comparing running time of ScaleGPM-NS and ScaleGPM-GS when ScaleGPM-HY selects GS (using the cost models).

| Graph & Pattern | Livej – house | | | | Twitter – 4-clique | | | |
|---|---|---|---|---|---|---|---|---|
| Error bound | 10% | 5% | 2% | 1% | 10% | 5% | 2% | 1% |
| ScaleGPM-NS | **0.008** | **0.05** | **0.31** | **1.14** | **0.07** | **0.13** | **0.69** | **2.68** |
| Arya | 395.56 | 1639.59 | 9873.81 | TO | 9656.85 | TO | TO | TO |

Table 6.7: Running time (sec) when adjusting Error Bounds from 10%-1%. ScaleGPM uses NS-online mode.

speedup compared to GraphZero.

Table 6.6 shows the cases where ScaleGPM-HY selects the GS mode. Notably, for Fr and $k = 9$ (sparse graph and big dense pattern) which is a needle-in-the-hay case for NS, ScaleGPM successfully choose to use GS instead of NS. This switch from NS to GS brings us a 61× performance improvement. Over all the cases where the GS engine is selected, the GS engine leads to a geomean average 13× speedup over the NS engine.

Table 6.7 shows Arya and ScaleGPM running time with varied error bounds. We observe that the performance gap between Arya and ScaleGPM remains huge (ranging from 31 to 49 thousand times) as we decrease the error bound.

Overall, we achieve 565× speedup over Arya, and four orders of magnitude speedup over GraphZero.

## 6.2 Effectiveness of Convergence Detection

Fig. 6.1 compares the error predicted by ScaleGPM-NS with the actual error throughout the sampling procedure, to show the effectiveness of ScaleGPM-NS's online convergence detection. We illustrate two cases here, but we have verified the same trend for all our test cases in the evaluation. In Fig. 6.1 we observe that for both cases, the predicted error curves

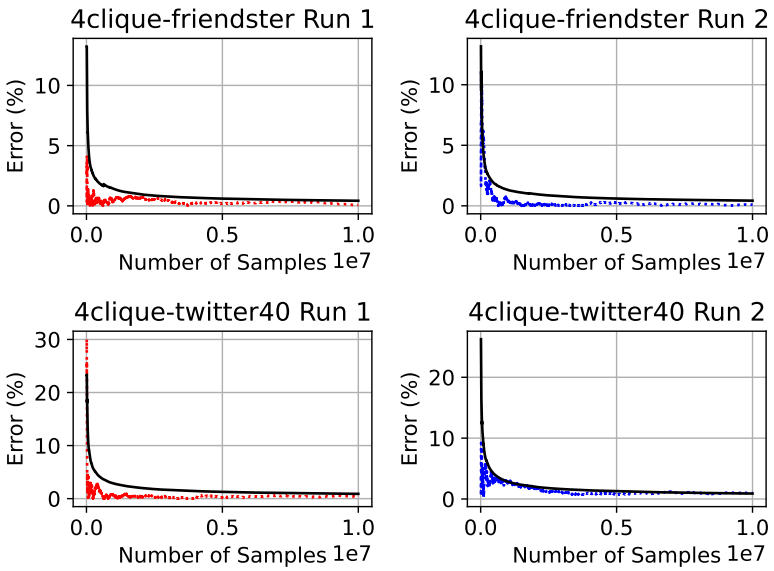
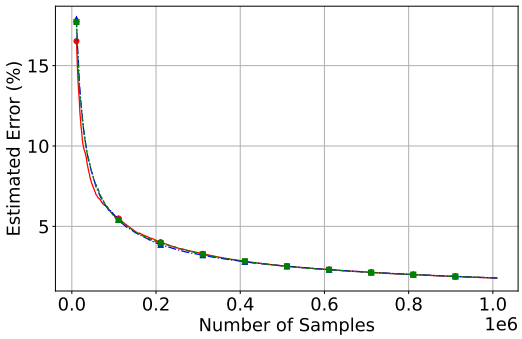



Figure 6.1: Comparing ScaleGPM-NS's estimated errors (black curves) with actual errors (red and blue curves). We do two runs of 4-clique counting on two graphs Fr and Tw. [71]

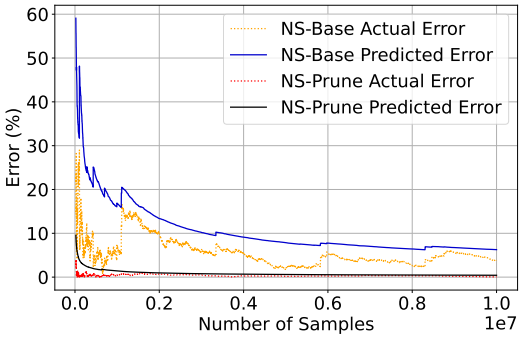
strictly bound the actual error, which verifies the high confidence that our method achieves. In addition to our theoretical proof in Section 4.1, this empirical study further demonstrates that our method provides strong guarantee on confidence, which is critical in applications where users want to have strict accuracy requirement.

Fig. 6.2a shows the stability of the error estimation method in ScaleGPM-NS. We do three repeated runs on the 4-clique pattern and Lj graph. Figures Fig. 6.4a and Fig. 6.4b show the predicted errors in our convergence detection mechanism on Fr and Tw respectively when the pattern is 4-clique. In contrast to the unstable predictions (Fig. 3.1) given by ELP in prior systems, our estimated errors are almost the same across three independent runs. Moreover, our online mechanism never fails, while ELP suffers convergence issue that may lead to endless preprocessing (e.g., for 6-clique in Table 6.2). We observe the same trend in stability on other graphs and patterns. This experiment further demonstrates that our method is highly reliable and can be adopted in practice.

Fig. 6.2b compares the convergence rate of NS-prune and NS-base, both under our online detection framework. We observe that our proposed early-pruning mechanism employed in




(a) Predicted errors in our convergence detection mechanism across three different runs (`4-clique` on `Lj`) are extremely stable. [71]




(b) Comparison between the convergence rate of NS-prune and NS-base. The same formula (with two standard deviations of confidence) was used to generate the error estimate curves. [71]

Figure 6.2: Trends in error and conversion with NS-Online.

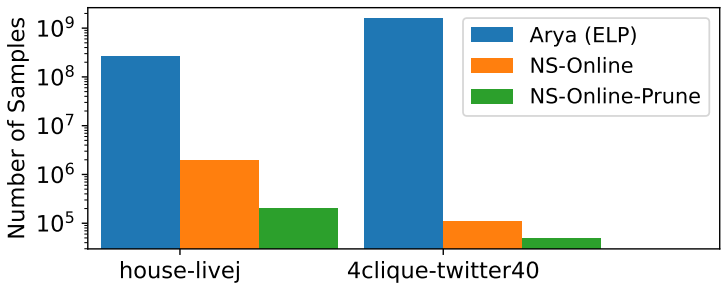
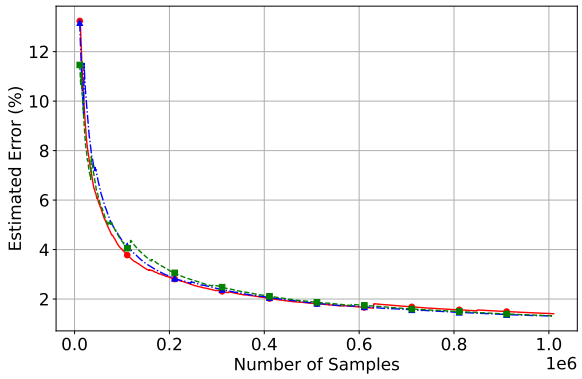


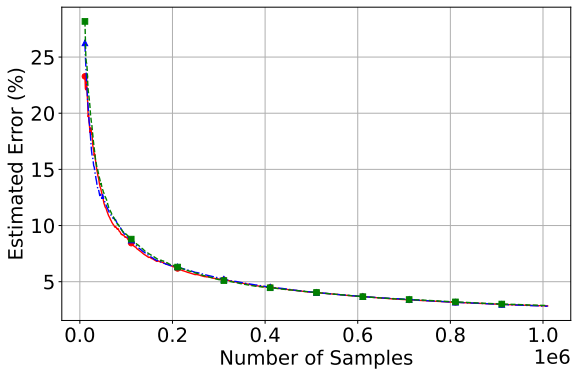

Figure 6.3: The reduction on the number of samples.

NS-prune result in roughly an order of magnitude lower error for the same number of samples. Therefore, with pruning, our system can converge much faster that ASAP and Arya, which is the other major reason why ScaleGPM achieves much better performance.

Fig. 6.3 shows the reduction on the number of samples $N_s$, by incrementally applying online convergence detection (orange) and eager-verify (green), against ELP in Arya (blue). Note that in Table 6.2 and Table 6.3 we report $N_s$ only for NS-online-prune, but here we breakdown the contributions of online detection and pruning. We observe that our online method reduces $N_s$ sharply over Arya. Applying pruning in eager-verify further reduces $N_s$ by a significant amount. Together, we can meet the same error bound with much few samples, and more importantly, with confidence.




(a) Predicted errors in our convergence detection mechanism across three different runs (`4-clique` on `Fr`) are extremely stable. [71]




(b) Predicted errors in our convergence detection mechanism across three different runs (`4-clique` on `Tw`) are extremely stable. [71]

Figure 6.4: Trends in predicted error with NS-Online.

## 6.3 Prediction Accuracy of Cost Models

Fig. 6.5 shows the effectiveness of our cost model in predicting the running time of the `GS` engine in ScaleGPM. As we increase the number of colors $c = \frac{1}{p}$ used in `GS`, the total work is exponentially decreased. Thus, the `GS` execution time rapidly becomes dominated by the preprocessing time spent on sparsifying the graph. For `4clique` and `house` on `Lj` and `Fr`, we see that the cost model precisely captures the exponential trend, as well as the stabilization window of the `GS` engine that we discussed in Fig. 4.3. Note that when $c$ is extremely small, e.g., $c < 5$, it is hard to accurately model the performance due to the steep slope in that curve. However, in this case (which is a needle-in-the-hay case) we would favor the use of `NS` or even exact counting, as `GS` with a small $c$ won't be much faster than exact counting.

Fig. 6.6 shows the effectiveness of our `NS` cost model. We see that our proposed *performance cone* correctly captures the running time of the `NS` engine at varying numbers of samples. Note that for `Fr`, a relatively sparse graph, the execution time approaches the bottom of the performance cone as expected. We find in practice, that `GS` and `NS` are often predicted to have very distinct performance, so the width of the cone is not an obstacle in making the

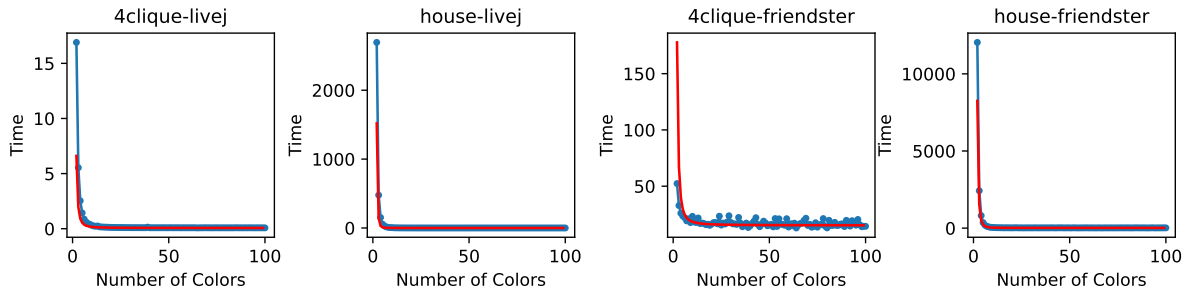



Figure 6.5: The quality of performance prediction for ScaleGPM's GS Engine. The red lines are our predictions, while blue dots are actual time.

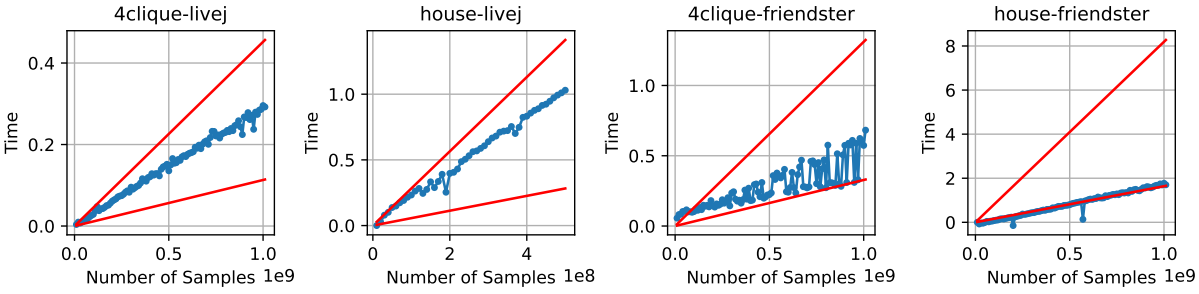



Figure 6.6: The quality of performance prediction for ScaleGPM's NS Engine. The red lines are our predictions, while blue dots are actual time.

correct choice. The fluctuation in the sparse case `4clique-Fr` is also expected, as sample hits are less frequent and thus more randomness is involved.

## 6.4 System Efficiency

**Timing Breakdown**. Fig. 6.7 shows the breakdown of the execution time spent on different components of ScaleGPM. We consider the profiling time, the GS preprocessing time,

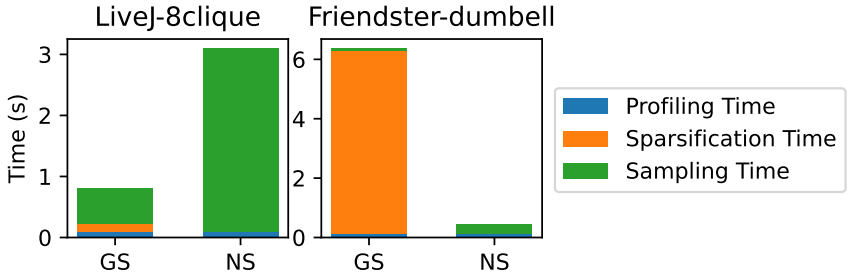



Figure 6.7: End-to-end time breakdown of ScaleGPM.

and the sampling time (either exact counting in GS or drawing samples in NS). We see that profiling remains a low percentage of the overall runtime. As for the sampling time, Lj-8clique requires less time using the GS engine, while Fr-dumbbell is processed faster using the NS engine, which aligns with our cost model prediction and thus ScaleGPM-HY makes the correct thresholding decision.

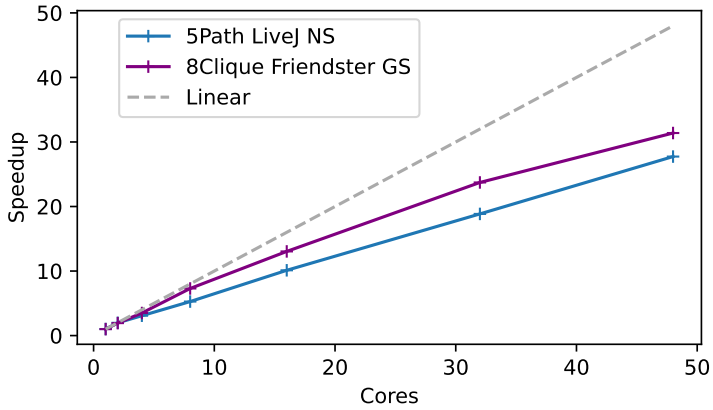



Figure 6.8: ScaleGPM speedup scaling over single-thread.

**Scalability**. Fig. 6.8 shows how the performance of GS engine and NS engine in ScaleGPM (error bound of 10%) scales in response to the increase of parallel cores (i.e., the number of threads). We evaluate the NS engine on 5path-Lj which is a case that favors the use of the NS engine (i.e., NS is faster than GS). GS is evaluated on 8clique-Fr as 8-clique is a dense pattern and is rare in Fr, preferably executed in GS. In both cases, we observe strong scaling that the execution time of both engines increases linearly as we range the number of cores from 1 to 48.

# Chapter 7

# Future Work

## 7.1 Expanded Sampling Schemes

Currently, ScaleGPM employs two complementing sampling schemes, ScaleGPM-GS and ScaleGPM-NS for it's hybrid model. However, as mentioned in Section 2.3.3, any number of different sampling schemes could be fit into the framework and be compared with a cost model.

For example, ScaleGPM-NS could be expanded to include a pattern-decomposition sampling scheme similar to Arya in addition to it's pruning optimization. ScaleGPM-NS could then estimate the performance of any decomposition or further sampling optimizations for that pattern and graph combination. Future work could adapt the cost model to calculate the trade-offs between different sampling optimizations. Then, we may find specific graphs and patterns where a particular decomposition would be helpful.

Future work could also expand ScaleGPM-GS to include loop perforation [51] and color-coding [35], [37]–[39], [42], [43], [59]–[64]. We could expand the cost model to encompass these methods, comparing the time of perforating each loop, creating a graph coloring for colorful matches, or sparsifying the graph. Then, ScaleGPM could adapt to select the best method for coarse grained sampling.

In the future, we imagine a multitude of sampling modes could abstractly be described to the cost model. This would allow for the best sampling scheme for any graph or pattern to be dynamically selected.

## 7.2 Distribution and GPU Acceleration

Currently, ScaleGPM is limited to multi-threaded single-machine mining on CPUs.

A promising area of work would be expanding ScaleGPM to a distributed setting for further performance improvements.

ScaleGPM-GS lends itself to distributed computation since each colored partition is independent and can be counted on independent workers.

However, in ScaleGPM-NS, there would need to be communication between workers about the current estimated error to determine the online stopping condition. Future work could involve designing a distributed protocol for calculating the current estimated error and disseminating the termination condition to all workers. Furthermore, if we can not hold an entire graph on one machine, we would have account for the additional error introduced by partitioning the graph with color sparsification, or include a message-passing scheme to cover broken edges.

We also imagine performing A-GPM on GPUs [12], [78]–[80] could allow for further speed-ups of ScaleGPM.

# Chapter 8

# Conclusion

Approximate graph pattern mining (A-GPM) systems are the backbone to support numerous real-world graph data analytics applications, including financial security, social network analysis, chemical engineering, bio-medicine. On very large graphs, an approximate strategy allows us to mine patterns where exact counting fails to terminate.

However, two key obstacles prevent A-GPM systems from being adopted in practice. The first limitation is a the lack of stable, confident termination mechanism. Previous systems rely on an ELP stage, which suffers a circular dependency issue in it's design, resulted in unstable performance. The second obstacle is the even worse performance and scalability when dealing with the hard "needle-in-the-hay" cases. When a pattern has a very low hit rate, existing systems require a huge number of samples to meet the error bound.

In the work, we present ScaleGPM, an accurate, high-performance and scalable A-GPM system. ScaleGPM involves two key innovations that remove the above obstacles.

First, we propose a novel online convergence detection mechanism, which can provide theoretical guarantee on prediction confidence and also yield stable termination condition. Our online method collects statistics about the samples during the sampling execution and uses the measured samples' variance to strongly bound error. Our online error detection drastically reduces the number of samples required, which leads to the 3 orders of magnitude

speedups over state-of-the-art A-GPM.

Second, we propose the eager-verify mechanism, which introduces pruning techniques into sampling to significantly improve the hit rate of our sampling procedure, and thus further reduce the number of samples needed, particularly in "needle-in-the-hay" cases. We also propose the hybrid sampling method to adaptively select the best-performing sampling scheme from two complementary schemes, based on our proposed cost models. Our hybrid sampling method uses cost models to estimate the performance of the fine-grained `NS` and coarse-grained `GS` sampling schemes, and adaptively switches to the `GS` scheme for the "needle-in-the-hay" cases that `NS` is incapable to handle.

The resulting system, ScaleGPM, achieves an average of 565× (up to 610,169×) speedup over the state-of-the-art A-GPM system, Arya, and is four orders of magnitude faster than state-of-the-art exact GPM system, GraphZero. In particular, ScaleGPM manages to rapidly mine billion-scale graphs, for which previous frameworks either run out of memory or fail to complete in hours.

# Appendix A

# Proofs

## A.1 Proof for Online Convergence

**Theorem 1.** *Given $\delta$, $n$ samples $X_1, \ldots, X_n$ drawn by using the* NS *sampling scheme, and the mean of sampled counts $\mu = \frac{1}{n}\sum_{i=1}^{n} X_i$, let $C$ be the true count and $\hat{\epsilon}$ be the estimated error computed by Eq. (4.1). As $n \to \infty$, the probability of the true relative error being smaller than the estimated relative error is $\mathbb{P}\left(\frac{|\mu - C|}{C} < \hat{\epsilon}\right) = 1 - \delta$.*

*Proof.* Let $D$ be the distribution $X_1, \ldots, X_n$ are sampled from. Since the NS estimator is unbiased, the true count $C$ is the mean of the distribution $D$. Our estimator of $C$ is $\mu := \frac{1}{n}\sum_{i=1}^{n} X_i$. This estimator satisfies $\mathbb{E}[\mu] = \mathbb{E}[\frac{1}{n}\sum_{i=1}^{n} X_i] = \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}[X_i] = \frac{1}{n}\sum_{i=1}^{n} C = C$ and $\mathbb{V}ar[\mu] = \mathbb{V}ar[\frac{1}{n}\sum_{i=1}^{n} X_i] = \frac{1}{n^2}\sum_{i=1}^{n}\mathbb{V}ar[X_i] = \frac{1}{n}\mathbb{V}ar[X_1]$. Because $\mu$ is the average of $n$ samples from a distribution $D$ (which clearly has finite mean and variance), the central limit theorem (CLT) applies, so $\mu$ follows the normal distribution. Formally, $\frac{\mu - \mathbb{E}[\mu]}{\sqrt{\mathbb{V}ar[\mu]}}$ converges in distribution to a standard normal as $n \to \infty$.

By the law of large numbers, as $n \to \infty$, the sample variance $\frac{1}{n}\sum_{i=1}^{n} X_i^2 - \mu^2$ converges in probability to $\mathbb{V}ar[X_1]$. Letting $\sigma^2 := \frac{1}{n}\left(\frac{1}{n}\sum_{i=1}^{n} X_i^2 - \mu^2\right)$, this means that $\frac{\mathbb{V}ar[\mu]}{\sigma^2} = \frac{\frac{1}{n}\mathbb{V}ar[X_1]}{\frac{1}{n}\left(\frac{1}{n}\sum_{i=1}^{n} X_i^2 - \mu^2\right)} = \frac{\mathbb{V}ar[X_1]}{\left(\frac{1}{n}\sum_{i=1}^{n} X_i^2 - \mu^2\right)}$ converges in probability to 1. Additionally, $\mu$ converges in probability to $C$, so $\frac{\mu}{C}$ converges in probability to 1. Therefore, by Slutsky's Theorem,

$\frac{\mu-\mathbb{E}[\mu]}{\sqrt{\mathbb{V}ar[\mu]}} \cdot \sqrt{\frac{\mathbb{V}ar[\mu]}{\sigma^2}} \cdot \frac{\mu}{C} = \frac{\mu-C}{C} \cdot \frac{\mu}{\sigma}$ converges in distribution to a standard normal. This means that for any fixed $x$, we have

$$\mathbb{P}\left(\frac{\mu - C}{C} \cdot \frac{\mu}{\sigma} < x\right) \to \Phi(x) \text{ as } n \to \infty.$$

This implies that

$$\mathbb{P}\left(\left|\frac{\mu - C}{C} \cdot \frac{\mu}{\sigma}\right| < x\right) \to 2\Phi(x) - 1 \text{ as } n \to \infty.$$

Plugging in $x = \Phi^{-1}(1 - \frac{\delta}{2})$, we get

$$\mathbb{P}\left(\left|\frac{\mu - C}{C} \cdot \frac{\mu}{\sigma}\right| < \Phi^{-1}(1 - \frac{\delta}{2})\right) \to 2\Phi\left(\Phi^{-1}(1 - \frac{\delta}{2})\right) - 1 \text{ as } n \to \infty.$$

Rearranging terms and simplifying yields

$$\mathbb{P}\left(\left|\frac{\mu - C}{C}\right| < \frac{\Phi^{-1}(1 - \frac{\delta}{2})\sigma}{\mu}\right) \to 2\left(1 - \frac{\delta}{2}\right) - 1 \text{ as } n \to \infty$$

$$\implies \mathbb{P}\left(\left|\frac{\mu - C}{C}\right| < \hat{\epsilon}\right) \to 1 - \delta \text{ as } n \to \infty.$$

□

## A.2 Lower Bound for Graph Sparsification

**Theorem 2.** *Given a data graph $\mathcal{G}$ and a pattern $\mathcal{P}$, $p \geq \frac{-3 \cdot \ln(\delta/2)\gamma}{\epsilon^2 \cdot C}$ obtains an $\epsilon$-$\delta$ estimation of $C$, the number of matches of $\mathcal{P}$ in $\mathcal{G}$.*

*Proof.* Formally, let $Y_1 \ldots Y_C$ be the random variables that represent the existence of matches, where $Y_i = 1$ if the match $M_i$ was preserved in the sparsified graph and $Y_i = 0$ if any of the edges within the match were removed. As $Y_i$ is not independent, this means that we can not use the standard Chernoff bound to tightly bound error. We could use Chebyshev bound but this is not a tight enough bound to be useful in practice.

We use the 'read-$k$' extension of Chernoff bounds [81]. The key idea is to map the dependent random variables $Y_i$ to a set of independent random variables $X_i$, and derive a bound from them. The 'read-$k$' extension gives the same bound as that of the standard Chernoff bound, except that the exponent is divided by $k$, where $k$ is the maximum number of $Y_i$ mapped to any $X_j$.

In our case, the edges are removed randomly. Let $X_1 \dots X_m$ be random variables for each of the $m$ edges in $\mathcal{G}$, where $X_i = 1$ if the edge was preserved and $X_i = 0$ if the edge was removed. We can see each edge has the probability $\hat{p}$ of being preserved and this probability is independent of any other edge, $X_j$, even if they may share one endpoint. Then $Y_i$ is dependent on exactly $l$ of the $X_1 \dots X_m$ edges, which are the specific edges that make up that match $M_i$. We define $\gamma$ as the maximum number of matches (of $\mathcal{P}$) incident to an edge. Thus, we can apply the 'read-$k$' extension with $k = \gamma$, which gives the following theorem. □

So in order to determine a good value for $p$, we need to estimate $C$ and $\gamma$, which is done by the fast profiling Section 5.3 on the graph $\mathcal{G}$. We assume that $\hat{p}$ is not very sensitive to $C$ and $\gamma$. So we don't need a very accurate estimation of $C$ and $\gamma$ in this profiling.

## A.3 Proof for Unbiasedness of NS-Prune

**Lemma 3.** *NS-prune is an unbiased estimator.*

*Proof.* Let $(v_0, \dots, v_{k-1})$ be the vertices of an occurrence, i.e., a *match*, of the pattern $\mathcal{P}$ in the matching order (e.g. for `4-cycle`, $(v_0, v_1), (v_1, v_2), (v_2, v_3), (v_3, v_0) \in E$). Each match corresponds to a unique $k$-tuple $(v_0, \dots, v_{k-1})$, e.g., for `4-cycle`, we enforce $v_0 = \max(v_0, v_1, v_2, v_3)$ and $v_3 < v_1$. During the execution of NS-prune, the probability that $v_0$ and $v_1$ are chosen is $1/m$ (see Line 2 in Algorithm 6), as all edges are equally likely. The probability that $v_2$ is chosen (see Line 5) is $1/|A|$ as $v_2 \in A$. In general, the probability that $v_i$ is chosen is $1/|S_i|$, where $S_i$ is the candidate set that $v_i$ is drawn from. Therefore

the probability that this particular match is sampled by NS-prune is the product of these probabilities (e.g. $1/(m \cdot |A| \cdot |B|)$ for `4-cycle`). The scaling factor $\alpha$ is the inverse of this probability, so the expected contribution of this match to the estimated count for one sampler is $\frac{1}{\alpha} \cdot \alpha = 1$. Let $C$ be the true count of $\mathcal{P}$ in $\mathcal{G}$, and let $X_{ij} = \alpha = (m \cdot |S_2| \cdots |S_{k-1}|)$ if the $i$th sample hits the $j$th match of $\mathcal{P}$, otherwise $X_{ij} = 0$, where $1 \leq i \leq N_s, 1 \leq j \leq C$. The estimated count is $\mathbb{E}\left(\frac{1}{N_s}\sum_{i,j} X_{ij}\right) = \frac{1}{N_s}\left(\sum_{i,j} \mathbb{E}(X_{ij})\right) = \frac{1}{N_s}\sum_{i,j} 1 = C$. Therefore NS-prune is an unbiased estimator. □

# Appendix B

# Artifact

## Abstract

This artifact appendix helps the readers reproduce the main evaluation results of ScaleGPM.

## Scope

The artifact can be used for evaluating and reproducing the main results of the thesis, including Table 6.2, Table 6.3, Table 6.4, Table 6.5, Table 6.5, Table 6.7, Fig. 6.7, Fig. 6.5, Fig. 6.6, Fig. 6.3 in Chapter 6.

## Contents

The details of the contained code and how to run ScaleGPM are described:

https://anonymous.4open.science/r/scale-gpm-B987/experiments.md.

# Hosting

The source code of this artifact can be found at https://anonymous.4open.science/r/scale-gpm-B987/.

## Requirements

### Hardware dependencies

This artifact depends on a 3.0 GHz, 48-core (2-sockets, 24 cores per socket) Intel CPU without hyperthreading, with up to 1TB of memory

### Software dependencies

This artifact requires OpenMP and GCC 8 or greater.

# References


[1] C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga, "Arabesque: A system for distributed graph mining," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15, Monterey, California: ACM, 2015, pp. 425–440, ISBN: 978-1-4503-3834-9. DOI: 10.1145/2815400.2815410. URL: http://doi.acm.org/10.1145/2815400.2815410.

[2] K. Wang, Z. Zuo, J. Thorpe, T. Q. Nguyen, and G. H. Xu, "Rstream: Marrying relational algebra with streaming for efficient graph mining on a single machine," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18, Carlsbad, CA, USA: USENIX Association, 2018, pp. 763–782, ISBN: 978-1-931971-47-8. URL: http://dl.acm.org/citation.cfm?id=3291168.3291225.

[3] K. Jamshidi, R. Mahadasa, and K. Vora, "Peregrine: A pattern-aware graph mining system," in *Proceedings of the Fifteenth EuroSys Conference*, ser. EuroSys '20, 2020.

[4] D. Mawhirter and B. Wu, "Automine: Harmonizing high-level abstraction and high performance for graph mining," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19, Huntsville, Ontario, Canada: ACM, 2019, pp. 509–523, ISBN: 978-1-4503-6873-5. DOI: 10.1145/3341301.3359633. URL: http://doi.acm.org/10.1145/3341301.3359633.

[5] V. Dias, C. H. C. Teixeira, D. Guedes, W. Meira, and S. Parthasarathy, "Fractal: A general-purpose graph pattern mining system," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19, Amsterdam, Netherlands: ACM, 2019, pp. 1357–1374, ISBN: 978-1-4503-5643-5. DOI: 10.1145/3299869.3319875. URL: http://doi.acm.org/10.1145/3299869.3319875.

[6] T. Shi, M. Zhai, Y. Xu, and J. Zhai, "Graphpi: High performance graph pattern matching through effective redundancy elimination," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20, Atlanta, Georgia: IEEE Press, 2020, ISBN: 9781728199986.

[7] X. Chen, R. Dathathri, G. Gill, L. Hoang, and K. Pingali, "Sandslash: A Two-Level Framework for Efficient Graph Pattern Mining," in *Proceedings of the 35th ACM International Conference on Supercomputing*, ser. ICS '21, 2021.

[8] Xuhao Chen and Arvind, "Efficient and Scalable Graph Pattern Mining on GPUs," in *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation* **(OSDI)**, 2022 [pdf].

[9] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, "Network motifs: Simple building blocks of complex networks," *Science*, vol. 298, no. 5594, pp. 824–827, 2002, ISSN: 0036-8075. DOI: 10.1126/science.298.5594.824. eprint: https://science.sciencemag.org/content/298/5594/824.full.pdf. URL: https://science.sciencemag.org/content/298/5594/824.

[10] A. P. Iyer, Z. Liu, X. Jin, S. Venkataraman, V. Braverman, and I. Stoica, "Asap: Fast, approximate graph pattern mining at scale," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18, Carlsbad, CA, USA: USENIX Association, 2018, pp. 745–761, ISBN: 978-1-931971-47-8. URL: http://dl.acm.org/citation.cfm?id=3291168.3291224.

[11] Z. Zhu, K. Wu, and Z. Liu, "Arya: Arbitrary graph pattern mining with decomposition-based sampling," in *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'23, 2023.

[12] X. Chen, R. Dathathri, G. Gill, and K. Pingali, "Pangolin: An efficient and flexible graph mining system on cpu and gpu," *Proc. VLDB Endow.*, vol. 13, no. 8, Aug. 2020, ISSN: 2150-8097. DOI: 10.14778/3389133.3389137.

[13] A. R. Benson, D. F. Gleich, and J. Leskovec, "Higher-order organization of complex networks," *Science*, vol. 353, no. 6295, pp. 163–166, 2016, ISSN: 0036-8075. DOI: 10.1126/science.aad9029. eprint: https://science.sciencemag.org/content/353/6295/163.full.pdf. URL: https://science.sciencemag.org/content/353/6295/163.

[14] D. Mawhirter, S. Reinehr, C. Holmes, T. Liu, and B. Wu, "Graphzero: A high-performance subgraph matching system," *SIGOPS Oper. Syst. Rev.*, vol. 55, no. 1, pp. 21–37, Jun. 2021, ISSN: 0163-5980. DOI: 10.1145/3469379.3469383. URL: https://doi.org/10.1145/3469379.3469383.

[15] 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*, ser. WWW '17, Perth, Australia: International World Wide Web Conferences Steering Committee, 2017, pp. 1431–1440, ISBN: 978-1-4503-4913-0. DOI: 10.1145/3038912.3052597. URL: https://doi.org/10.1145/3038912.3052597.

[16] J. Chen and X. Qian, "Decomine: A compilation-based graph pattern mining system with pattern decomposition," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS 2023, Vancouver, BC, Canada: Association for Computing Machinery, 2022, pp. 47–61, ISBN: 9781450399159. DOI: 10.1145/3567955.3567956. URL: https://doi.org/10.1145/3567955.3567956.

[17] J. Chen and X. Qian, *Kudu: An efficient and scalable distributed graph pattern mining engine*, 2021. arXiv: 2105.03789 [cs.DC].

[18] J. Chen and X. Qian, *Dwarvesgraph: A high-performance graph mining system with pattern decomposition*, 2021. arXiv: 2008.09682 [cs.DC].

[19] H. Kim, J. Lee, S. S. Bhowmick, W.-S. Han, J. Lee, S. Ko, and M. H. Jarrah, "DUALSIM: Parallel subgraph enumeration in a massive graph on a single machine," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16, San Francisco, California, USA: ACM, 2016, pp. 1231–1245, ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2915209. URL: http://doi.acm.org/10.1145/2882903.2915209.

[20] K. Ammar, F. McSherry, S. Salihoglu, and M. Joglekar, "Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows," *Proc. VLDB Endow.*, vol. 11, no. 6, pp. 691–704, Feb. 2018, ISSN: 2150-8097. DOI: 10.14778/3184470.3184473. URL: https://doi.org/10.14778/3184470.3184473.

[21] X. Chen, T. Huang, S. Xu, T. Bourgeat, C. Chung, and Arvind, "Flexminer: A pattern-aware accelerator for graph pattern mining," in *Proceedings of the International Symposium on Computer Architecture*, 2021.

[22] L. S. Buriol, G. Frahling, S. Leonardi, A. Marchetti-Spaccamela, and C. Sohler, "Counting triangles in data streams," in *Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS '06, Chicago, IL, USA: Association for Computing Machinery, 2006, pp. 253–262, ISBN: 1595933182. DOI: 10.1145/1142351.1142388. URL: https://doi.org/10.1145/1142351.1142388.

[23] R. Pagh and C. E. Tsourakakis, "Colorful triangle counting and a mapreduce implementation," *Inf. Process. Lett.*, vol. 112, no. 7, pp. 277–281, Mar. 2012, ISSN: 0020-0190. DOI: 10.1016/j.ipl.2011.12.007. URL: https://doi.org/10.1016/j.ipl.2011.12.007.

[24] C. E. Tsourakakis, P. Drineas, E. Michelakis, I. Koutis, and C. Faloutsos, "Spectral counting of triangles via element-wise sparsification and triangle-based link recommendation," *Social Network Analysis and Mining*, vol. 1, pp. 75–81, 2011.

[25] 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*, ser. KDD '09, Paris, France: Association for Computing Machinery, 2009, pp. 837–846, ISBN: 9781605584959. DOI: 10.1145/1557019.1557111. URL: https://doi.org/10.1145/1557019.1557111.

[26] A. Pavan, K. Tangwongsan, S. Tirthapura, and K.-L. Wu, "Counting and sampling triangles from a graph stream," *Proc. VLDB Endow.*, vol. 6, no. 14, pp. 1870–1881, Sep. 2013, ISSN: 2150-8097. DOI: 10.14778/2556549.2556569. URL: https://doi.org/10.14778/2556549.2556569.

[27] A. Turk and D. Turkoglu, "Revisiting wedge sampling for triangle counting," in *The World Wide Web Conference*, ser. WWW '19, San Francisco, CA, USA: Association for Computing Machinery, 2019, pp. 1875–1885, ISBN: 9781450366748. DOI: 10.1145/3308558.3313534. URL: https://doi.org/10.1145/3308558.3313534.

[28] S. K. Bera and C. Seshadhri, "How to count triangles, without seeing the whole graph," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '20, Virtual Event, CA, USA: Association for Computing Machinery, 2020, pp. 306–316, ISBN: 9781450379984. DOI: 10.1145/3394486.3403073. URL: https://doi.org/10.1145/3394486.3403073.

[29] J. Y. Chen, T. Eden, P. Indyk, H. Lin, S. Narayanan, R. Rubinfeld, S. Silwal, T. Wagner, D. Woodruff, and M. Zhang, "Triangle and four cycle counting with predictions in graph streams," in *International Conference on Learning Representations*, 2022. URL: https://openreview.net/forum?id=8in_5gN9I0.

[30] N. K. Ahmed, N. Duffield, J. Neville, and R. Kompella, "Graph sample and hold: A framework for big-graph analytics," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '14, New York, New York, USA: Association for Computing Machinery, 2014, pp. 1446–1455, ISBN: 9781450329569. DOI: 10.1145/2623330.2623757. URL: https://doi.org/10.1145/2623330.2623757.

[31] X. Ye, R.-H. Li, Q. Dai, H. Chen, and G. Wang, "Lightning fast and space efficient k-clique counting," in *Proceedings of the ACM Web Conference 2022*, ser. WWW '22, Virtual Event, Lyon, France: Association for Computing Machinery, 2022, pp. 1191–1202, ISBN: 9781450390965. DOI: 10.1145/3485447.3512167. URL: https://doi.org/10.1145/3485447.3512167.

[32] J. Shi, L. R. Huang, and J. Shun, "Parallel five-cycle counting algorithms," *ACM J. Exp. Algorithmics*, vol. 27, Oct. 2022, ISSN: 1084-6654. DOI: 10.1145/3556541. URL: https://doi.org/10.1145/3556541.

[33] E. R. Elenberg, K. Shanmugam, M. Borokhovich, and A. G. Dimakis, "Beyond triangles: A distributed framework for estimating 3-profiles of large graphs," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '15, Sydney, NSW, Australia: ACM, 2015, pp. 229–238, ISBN: 978-1-4503-3664-2. DOI: 10.1145/2783258.2783413. URL: http://doi.acm.org/10.1145/2783258.2783413.

[34] 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*, ser. WWW '15, Florence, Italy: International World Wide Web Conferences Steering Committee, 2015, pp. 495–505, ISBN: 978-1-4503-3469-3. DOI: 10.1145/2736277.2741101. URL: https://doi.org/10.1145/2736277.2741101.

[35] G. M. Slota and K. Madduri, "Fast approximate subgraph counting and enumeration," in *2013 42nd International Conference on Parallel Processing*, 2013, pp. 210–219. DOI: 10.1109/ICPP.2013.30.

[36] 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*, 2012, pp. 91–100. DOI: 10.1109/ICDM.2012.87.

[37] M. Bressan, F. Chierichetti, R. Kumar, S. Leucci, and A. Panconesi, "Motif counting beyond five nodes," *ACM Trans. Knowl. Discov. Data*, vol. 12, no. 4, 48:1–48:25, Apr. 2018, ISSN: 1556-4681. DOI: 10.1145/3186586. URL: http://doi.acm.org/10.1145/3186586.

[38] M. Bressan, S. Leucci, and A. Panconesi, "Motivo: Fast motif counting via succinct color coding and adaptive sampling," *Proc. VLDB Endow.*, vol. 12, no. 11, pp. 1651–1663, Jul. 2019, ISSN: 2150-8097. DOI: 10.14778/3342263.3342640. URL: https://doi.org/10.14778/3342263.3342640.

[39] 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*, ser. WSDM '17, Cambridge, United Kingdom: ACM, 2017, pp. 557–566, ISBN: 978-1-4503-4675-7. DOI: 10.1145/3018661.3018732. URL: http://doi.acm.org/10.1145/3018661.3018732.

[40] K. Paramonov, D. Shemetov, and J. Sharpnack, "Estimating graphlet statistics via lifting," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '19, Anchorage, AK, USA: Association for Computing Machinery, 2019, pp. 587–595, ISBN: 9781450362016. DOI: 10.1145/3292500.3330995. URL: https://doi.org/10.1145/3292500.3330995.

[41] P. Wang, J. C. S. Lui, B. Ribeiro, D. Towsley, J. Zhao, and X. Guan, "Efficiently estimating motif statistics of large networks," *ACM Trans. Knowl. Discov. Data*, vol. 9, no. 2, Sep. 2014, ISSN: 1556-4681. DOI: 10.1145/2629564. URL: https://doi.org/10.1145/2629564.

[42] Z. Zhao, M. Khan, V. S. A. Kumar, and M. V. Marathe, "Subgraph enumeration in large social contact networks using parallel color coding and streaming," in *2010 39th International Conference on Parallel Processing*, 2010, pp. 594–603. DOI: 10.1109/ICPP.2010.67.

[43] N. Alon, P. Dao, I. Hajirasouliha, F. Hormozdiari, and S. C. Sahinalp, "Biomolecular network motif counting and discovery by color coding," *Bioinformatics*, vol. 24, no. 13, pp. i241–i249, 2008.

[44] 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*, ser. KDD '18, London, United Kingdom: Association for Computing Machinery, 2018, pp. 2150–2159, ISBN: 9781450355520. DOI: 10.1145/3219819.3220097. URL: https://doi.org/10.1145/3219819.3220097.

[45] M. Kuramochi and G. Karypis, "Grew-a scalable frequent subgraph discovery algorithm," in *Proceedings of the Fourth IEEE International Conference on Data Mining*, ser. ICDM '04, USA: IEEE Computer Society, 2004, pp. 439–442, ISBN: 0769521428.

[46] G. Preti, G. De Francisci Morales, and M. Riondato, "Maniacs: Approximate mining of frequent subgraph patterns through sampling," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, ser. KDD '21, Virtual Event, Singapore: Association for Computing Machinery, 2021, pp. 1348–1358, ISBN: 9781450383325. DOI: 10.1145/3447548.3467344. URL: https://doi.org/10.1145/3447548.3467344.

[47] E. Abdelhamid, I. Abdelaziz, P. Kalnis, Z. Khayyat, and F. Jamour, "Scalemine: Scalable parallel frequent subgraph mining in a single large graph," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16, Salt Lake City, Utah: IEEE Press, 2016, 61:1–61:12, ISBN: 978-1-4673-8815-3. URL: http://dl.acm.org/citation.cfm?id=3014904.3014986.

[48] V. Bhatia and R. Rani, "Ap-fsm: A parallel algorithm for approximate frequent subgraph mining using pregel," *Expert Systems with Applications*, vol. 106, pp. 217–232, 2018, ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2018.04.010. URL: https://www.sciencedirect.com/science/article/pii/S0957417418302409.

[49] S. Purohit, S. Choudhury, and L. B. Holder, "Application-specific graph sampling for frequent subgraph mining and community detection," in *2017 IEEE International Conference on Big Data (Big Data)*, 2017, pp. 1000–1005. DOI: 10.1109/BigData.2017.8258022.

[50] M. Bressan, "Efficient and near-optimal algorithms for sampling connected subgraphs," in *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, ser. STOC 2021, Virtual, Italy: Association for Computing Machinery, 2021, pp. 1132–1143, ISBN: 9781450380539. DOI: 10.1145/3406325.3451042. URL: https://doi.org/10.1145/3406325.3451042.

[51] P. Jiang, Y. Wei, J. Su, R. Wang, and B. Wu, "Samplemine: A framework for applying random sampling to subgraph pattern mining through loop perforation," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '22, Chicago, Illinois: Association for Computing Machinery, 2023, pp. 185–197, ISBN: 9781450398688. DOI: 10.1145/3559009.3569658. URL: https://doi.org/10.1145/3559009.3569658.

[52] A. A. Benczúr and D. R. Karger, "Approximating s-t minimum cuts in õ(n2) time," in *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, ser. STOC '96, Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pp. 47–55, ISBN: 0897917855. DOI: 10.1145/237814.237827. URL: https://doi.org/10.1145/237814.237827.

[53] W. S. Fung, R. Hariharan, N. J. Harvey, and D. Panigrahi, "A general framework for graph sparsification," in *Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing*, ser. STOC '11, San Jose, California, USA: Association for Computing Machinery, 2011, pp. 71–80, ISBN: 9781450306911. DOI: 10.1145/1993636.1993647. URL: https://doi.org/10.1145/1993636.1993647.

[54] D. A. Spielman and S.-H. Teng, "Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems," in *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, 2004, pp. 81–90.

[55] D. A. Spielman and N. Srivastava, "Graph sparsification by effective resistances," *SIAM Journal on Computing*, vol. 40, no. 6, pp. 1913–1926, 2011. DOI: 10.1137/080734029. eprint: https://doi.org/10.1137/080734029. URL: https://doi.org/10.1137/080734029.

[56] D. A. Spielman and S.-H. Teng, "Spectral sparsification of graphs," *SIAM Journal on Computing*, vol. 40, no. 4, pp. 981–1025, 2011.

[57] R. A. Rossi, R. Zhou, and N. K. Ahmed, "Estimation of graphlet counts in massive networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 30, no. 1, pp. 44–57, 2019. DOI: 10.1109/TNNLS.2018.2826529.

[58] J. Shi, L. Dhulipala, and J. Shun, "Parallel clique counting and peeling algorithms," in *Conference on Applied and Computational Discrete Algorithms*, 2020.

[59] N. Alon, R. Yuster, and U. Zwick, "Color-coding: A new method for finding simple paths, cycles and other small subgraphs within large graphs," in *Proceedings of the Twenty-sixth Annual ACM Symposium on Theory of Computing*, ser. STOC '94, Montreal, Quebec, Canada: ACM, 1994, pp. 326–335, ISBN: 0-89791-663-8. DOI: 10.1145/195058.195179. URL: http://doi.acm.org/10.1145/195058.195179.

[60] N. Alon, R. Yuster, and U. Zwick, "Color-coding," *J. ACM*, vol. 42, no. 4, pp. 844–856, Jul. 1995, ISSN: 0004-5411. DOI: 10.1145/210332.210337. URL: https://doi.org/10.1145/210332.210337.

[61] F. Hüffner, S. Wernicke, and T. Zichner, "Algorithm engineering for color-coding with applications to signaling pathway detection," *Algorithmica*, vol. 52, pp. 114–132, 2008.

[62] G. M. Slota and K. Madduri, "Parallel color-coding," *Parallel Computing*, vol. 47, pp. 51–69, 2015, Graph analysis for scientific discovery, ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2015.02.004. URL: https://www.sciencedirect.com/science/article/pii/S0167819115000423.

[63] V. T. Chakaravarthy, M. Kapralov, P. Murali, F. Petrini, X. Que, Y. Sabharwal, and B. Schieber, "Subgraph counting: Color coding beyond trees," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Los Alamitos, CA, USA: IEEE Computer Society, May 2016, pp. 2–11. DOI: 10.1109/IPDPS.2016.122. URL: https://doi.ieeecomputersociety.org/10.1109/IPDPS.2016.122.

[64] Z. Zhao, G. Wang, A. R. Butt, M. Khan, V. A. Kumar, and M. V. Marathe, "Sahad: Subgraph analysis in massive networks using hadoop," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, 2012, pp. 390–401. DOI: 10.1109/IPDPS.2012.44.

[65] X. Chen, Y. Li, P. Wang, and J. C. S. Lui, "A general framework for estimating graphlet statistics via random walk," *Proc. VLDB Endow.*, vol. 10, no. 3, pp. 253–264, Nov. 2016, ISSN: 2150-8097. DOI: 10.14778/3021924.3021940. URL: https://doi.org/10.14778/3021924.3021940.

[66] Z. Bar-Yossef, R. Kumar, and D. Sivakumar, "Reductions in streaming algorithms, with an application to counting triangles in graphs," in *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '02, San Francisco, California: Society for Industrial and Applied Mathematics, 2002, pp. 623–632, ISBN: 089871513X.

[67] J. Kallaugher, A. McGregor, E. Price, and S. Vorotnikova, "The complexity of counting cycles in the adjacency list streaming model," in *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, ser. PODS '19, Amsterdam, Netherlands: Association for Computing Machinery, 2019, pp. 119–133, ISBN: 9781450362276. DOI: 10.1145/3294052.3319706. URL: https://doi.org/10.1145/3294052.3319706.

[68] M. Aliakbarpour, A. S. Biswas, T. Gouleakis, J. Peebles, R. Rubinfeld, and A. Yodpinyanee, "Sublinear-time algorithms for counting star subgraphs via edge sampling," *Algorithmica*, vol. 80, no. 2, pp. 668–697, Feb. 2018, ISSN: 0178-4617. DOI: 10.1007/s00453-017-0287-3. URL: https://doi.org/10.1007/s00453-017-0287-3.

[69] S. Assadi, M. Kapralov, and S. Khanna, "A simple sublinear-time algorithm for counting arbitrary subgraphs via edge sampling," in *Information Technology Convergence and Services*, 2018.

[70] J. Chen, T. Eden, P. Indyk, S. Narayanan, R. Rubinfeld, S. Silwal, D. Woodruff, and M. Zhang, "Triangle and four cycle counting with predictions in graph streams," *Tenth International Conference on Learning Representations (ICLR 2022)*, URL: https://par.nsf.gov/biblio/10338743.

[71] A. Arpaci-Dusseau, Z. Zhou, and X. Chen, *Accurate and fast approximate graph pattern mining at scale*, 2024. arXiv: 2405.03488 [cs.PF].

[72] A. Mhedhbi and S. Salihoglu, "Optimizing subgraph queries by combining binary and worst-case optimal joins," *Proc. VLDB Endow.*, vol. 12, no. 11, pp. 1692–1704, Jul. 2019, ISSN: 2150-8097. DOI: 10.14778/3342263.3342643. URL: https://doi.org/10.14778/3342263.3342643.

[73] J. Leskovec, *Snap: Stanford network analysis platform*, 2013. URL: http://snap.stanford.edu/data/index.html.

[74] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" In *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10, Raleigh, North Carolina, USA: ACM, 2010, pp. 591–600, ISBN: 978-1-60558-799-8. DOI: 10.1145/1772690.1772751. URL: http://doi.acm.org/10.1145/1772690.1772751.

[75] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *CoRR*, vol. abs/1205.6233, 2012. arXiv: 1205.6233. URL: http://arxiv.org/abs/1205.6233.

[76] P. Boldi, M. Santini, and S. Vigna, "A large time-aware graph," *SIGIR Forum*, vol. 42, no. 2, pp. 33–38, 2008.

[77] P. Boldi and S. Vigna, "The WebGraph Framework I: Compression Techniques," in *Proceedings of the 13th International Conference on World Wide Web*, ser. WWW '04, New York, NY, USA: ACM, 2004, pp. 595–602, ISBN: 1-58113-844-X. DOI: 10.1145/988672.988752. URL: http://doi.acm.org/10.1145/988672.988752.

[78] W. Guo, Y. Li, M. Sha, B. He, X. Xiao, and K.-L. Tan, "Gpu-accelerated subgraph enumeration on partitioned graphs," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20, Portland, OR, USA: Association for Computing Machinery, 2020, pp. 1067–1082, ISBN: 9781450367356. DOI: 10.1145/3318464.3389699. URL: https://doi.org/10.1145/3318464.3389699.

[79] M. Almasri, I. E. Hajj, R. Nagi, J. Xiong, and W.-m. Hwu, "K-clique counting on gpus," *arXiv preprint arXiv:2104.13209*, 2021. URL: https://arxiv.org/abs/2104.13209.

[80] A. Chatterjee, S. Radhakrishnan, and J. K. Antonio, "Counting problems on graphs: Gpu storage and parallel computing techniques," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, 2012, pp. 804–812. DOI: 10.1109/IPDPSW.2012.99.

[81] D. Gavinsky, S. Lovett, M. Saks, and S. Srinivasan, *A tail bound for read-k families of functions*, 2012. arXiv: 1205.1478 [cs.DM].