Motivo: fast motif counting via succinct color coding and adaptive sampling

Marco Bressan[*]
Dipartimento di Informatica,
Sapienza Università di Roma
bressan@di.uniroma1.it

Stefano Leucci
Department of Algorithms & Complexity,
Max Planck Institute for Informatics
stefano.leucci@mpi-inf.mpg.de

Alessandro Panconesi[*]
Dipartimento di Informatica,
Sapienza Università di Roma
ale@di.uniroma1.it

ABSTRACT

The randomized technique of color coding is behind state-of-the-art algorithms for estimating graph motif counts. Those algorithms, however, are not yet capable of scaling well to very large graphs with billions of edges. In this paper we develop novel tools for the "motif counting via color coding" framework. As a result, our new algorithm, MOTIVO, scales to much larger graphs while at the same time providing more accurate motif counts than ever before. This is achieved thanks to two types of improvements. First, we design new succinct data structures for fast color coding operations, and a biased coloring trick that trades accuracy versus resource usage. These optimizations drastically reduce the resource requirements of color coding. Second, we develop an adaptive motif sampling strategy, based on a fractional set cover problem, that breaks the additive approximation barrier of standard sampling. This gives multiplicative approximations for all motifs at once, allowing us to count not only the most frequent motifs but also extremely rare ones.

To give an idea of the improvements, in 40 minutes MOTIVO counts 7-nodes motifs on a graph with 65M nodes and 1.8B edges; this is 30 and 500 times larger than the state of the art, respectively in terms of nodes and edges. On the accuracy side, in one hour MOTIVO produces accurate counts of ≈ 10.000 distinct 8-node motifs on graphs where state-of-the-art algorithms fail even to find the second most frequent motif. Our method requires just a high-end desktop machine. These results show how color coding can bring motif mining to the realm of truly massive graphs using only ordinary hardware.

PVLDB Reference Format:
M. Bressan, S. Leucci, A. Panconesi. Motivo: fast motif counting via succinct color coding and adaptive sampling. *PVLDB*, 12(11): 1651-1663, 2019.
DOI: https://doi.org/10.14778/3342263.3342640

1. INTRODUCTION

Graphlets, also called motifs or patterns, are small induced subgraphs of a graph. Graphlets are often considered the "building blocks" of networks [17, 23, 28, 29], and their analysis has helped understanding network evolution [1], designing better graph classification algorithms [28], and developing cutting-edge clustering techniques [29].

A fundamental problem in graphlet mining and analysis is graphlet counting: estimating as accurately as possible the number of copies of a given graphlet (e.g., a tree, a clique, etc.) in a graph. Graphlet counting has a long and rich history, which began with triangle counting and received intense interest in recent years [2, 6, 7, 10, 12, 15, 17, 20, 21, 25, 26, 27, 30]. Since exact graphlet counting is notoriously hard, one must resort to approximate probabilistic counting to obtain algorithms with an acceptable practical performance. Approximate counting is indeed often sufficient, for example when performing hypothesis testing (deciding if a graph comes from a certain distribution or not) or estimating the clustering coefficient of a graph (the fraction of triangles among 3-node graphlets).

The simplest formulation of approximate graphlet counting, which we adopt in this work, is the following. We are given a simple graph G on n nodes, an integer $k > 2$, and two approximation parameters $\epsilon, \delta \in (0, 1)$. For each graphlet H on k nodes (the clique, the path, the star etc.), we want a very reliable and accurate estimate of the number of induced copies of H in G: with probability at least $1 - \delta$, all estimates should be within a factor $(1 \pm \epsilon)$ of the actual values. Note that we are talking about induced copies; non-induced copies are easier to count and can be derived from the induced ones. Our goal is to develop practical algorithms that solve this problem for sizes of G and H that were out of reach before, i.e., graphs with hundreds of millions of edges and graphlets on more than 5 or 6 nodes. We remark that scaling k is harder than it may seem at a first sight: just the number of distinct graphlets grows from 21 for $k = 5$ to over 10.000 for $k = 8$. Thus, mining larger graphlets is not simply a matter of fixing k – instead, it requires new ideas.

To appreciate the main obstacles to large-scale graphlet counting, let us review the existing techniques and their limitations. All known efficient algorithms for approximating graphlet counts are based on sampling graphlets from G. One popular way of sampling graphlets is to define a random walk over the set of graphlets of G, simulate it until it reaches stationarity, and take a sample [6, 12, 15, 25]. This technique is simple, elegant, and has a small memory footprint. Unfortunately, it can be extremely inefficient: it is

[*]Supported in part by the ERC Starting Grant DMAP 680153, a Google Focused Research Award, and by the MIUR grant "Dipartimenti di eccellenza 2018-2022" awarded to the Department of Computer Science of the Sapienza University of Rome.

Proceedings of the VLDB Endowment, Vol. 12, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3342263.3342640

known that, even if G is fast-mixing, the random walk may need $\Omega(n^{k-1})$ steps to reach stationarity and/or to hit on the most frequent graphlet of G [8, 9].

A completely different sampling approach is the CC algorithm of [8, 9], an extension of the well-known color coding technique [4] based on two key observations. The first observation is that color coding can be used to build an abstract "urn" which contains a sub-population of all the k-trees of G that is very close to the true one. The second observation is that the task of sampling k-graphlets can be reduced, with minimal overhead, to sampling k-trees from the urn. One can thus estimate graphlet counts in two steps: the *build-up phase*, where one builds the urn from G, and the *sampling phase*, where one samples k-trees from the urn. Building the urn requires time $O(a^k m)$ and space $O(a^k n)$ for some $a > 0$, where n and m are the number of nodes and edges of G, while sampling takes a variable but typically small amount of time per sample. The resulting algorithm, CC, outperforms random walk-based approaches and is the current state of the art [8, 9].

Although CC has extended the outreach of graphlet counting techniques, it cannot effectively cope with graphs with billions of edges and values of k beyond six. This is due to two main bottlenecks. First, the time and space taken by the build-up phase are significant and prevent CC from scaling to the values of G and k that we are interested in this paper. For example, on a machine with 64GB of main memory, the largest graph for which CC runs successfully has 5.4M nodes for $k = 5, 6$ and just 2M nodes for $k = 7$. Second, taking s samples from the abstract urn gives the usual additive $1/s$-approximation, which means we can accurately count only those graphlets whose occurrences are a fraction at least $1/s$ of the total. Unfortunately, in many graphs, most graphlets have a very low relative frequency, and CC is basically useless to count them.

In this work we overcome the limitations of CC by making two main contributions to the "motif counting via color coding" framework. The first contribution is reducing the running time and space usage of the build-up phase. We do so in three ways. First, we introduce succinct color coding data structures that can represent colored rooted trees on up to 16 nodes with just one machine word, and support frequent operations (e.g. merging trees) in just a few elementary CPU instructions. This is key, as colored trees are the main objects manipulated in the build-up phase. Second, for large graphs we present a simple "biased coloring" trick that we use to trade space and time against the accuracy of the urn (the distance of the urn's distribution from the actual tree distribution of G), whose loss we quantify via concentration bounds. Third, we describe a set of architectural and implementation optimizations. These ingredients make the build-up phase significantly faster and bring us from millions to billions of edges and from $k = 5$ to $k = 8$.

Our second contribution is for the sampling phase and is of a fundamentally different nature. To convey the idea, imagine having an urn with 1000 balls of which 990 red, 9 green, and 1 blue. Sampling from the urn, we will quickly get a good estimate of the fraction of red balls, but we will need many samples to witness even one green or blue ball. Now imagine that, after having seen those red balls, we could remove from the urn 99% of all red balls. We would be left with 10 red balls, 9 green balls, and 1 blue ball. At this point we could quickly get a good estimate of the fraction of green balls. We could then ask the urn to delete almost 99% of the red and green balls, and we could quickly estimate the fraction of blue balls. What we show here is that the urn built in the build-up phase can be used to perform essentially this "deletion" trick, where the object to be removed are treelets. In this way, roughly speaking, we can first estimate the most frequent graphlet, then delete it from the urn and proceed to the second most frequent graphlet, delete it from the urn and so on. This means we can in principle obtain a small *relative* error for all graphlets, independently of their relative abundance in G, thus breaking the $\Theta(1/\epsilon)$ barrier of standard sampling. We name this algorithm AGS (adaptive graphlet sampling). To obtain AGS we actually develop an online greedy algorithm for a fractional set cover problem. We provide formal guarantees on the accuracy and sampling efficiency of AGS via set cover analysis and martingale concentration bounds.

In order to properly assess the impact of the various optimizations, in this paper we have added them incrementally to CC, which acts as a baseline. In this way, it is possible to assess in a quantitative way the improvements due to the various components.

Our final result is an algorithm, MOTIVO[1], that scales well beyond the state of the art in terms of input size and simultaneously ensures tighter guarantees. To give an idea, for $k = 7$ MOTIVO manages graphs with tens of millions of nodes and billions of edges, the largest having 65M nodes and 1.8B edges. This is 30 times and 500 times (respectively in terms of n and m) what CC can manage. For $k = 8$, our largest graph has 5.4M nodes and 50M edges (resp. 18 and 55 times CC). All this is done in 40 minutes on just a high-end commodity machine. For accuracy, the most extreme example is the YELP graph, where for $k = 8$ all but two graphlets have relative frequency below 10^{-7}. With a budget of 1M samples, CC finds only the *first* graphlet and misses all the others. MOTIVO instead outputs accurate counts ($\epsilon \leq 0.5$) of more than 90% of all graphlets, or 10.000 in absolute terms. The least frequent ones of those graphlets have frequency below 10^{-20}, and CC would need $\sim 3 \cdot 10^3$ years to find them even if it took one billion samples per second.

1.1 Related work

Counting induced subgraphs is a classic problem in computer science. The exact version is notoriously hard, and even detecting a k-clique in an n-node graph requires time $n^{\Omega(k)}$ under the Exponential Time Hypothesis [11]. Practical exact counting algorithms exist only for $k \leq 5$; currently, the fastest one is ESCAPE [20], which still takes a week on graphs with a few million nodes. We use ESCAPE for computing some of our ground-truth counts.

For approximate graphlet counting many techniques have been proposed. For $k \leq 5$, one can sample graphlets via path sampling (do a walk on k nodes in G and check the subgraph induced by those nodes) [17, 26, 27]. This technique, however, does not scale to $k > 5$. A popular approach is to sample graphlets via random walks [6, 25, 12, 15]. The idea is to define two graphlets as adjacent in G if they share $k-1$ nodes. This implicitly defines a reversible Markov chain over the graphlets of G which can be simulated efficiently. Once at stationarity, one can take the sample and easily compute an unbiased estimator of the graphlet frequencies.

[1] The C++ source code of MOTIVO is publicly available at `https://bitbucket.org/steven_/motivo`.

Unfortunately, these algorithms cannot estimate counts, but only frequencies. Even then, they may give essentially no guarantee unless one runs the walk for $\Omega(n^{k-1})$ steps, and in practice they are outperformed by CC [8, 9]. Another recent approach is that of edge-streaming algorithms based on reservoir sampling [22], which however are tailored to $k \leq 5$. As of today, the state of the art in terms of G and k is the color-coding based CC algorithm of [8, 9]. CC can manage graphs on $\sim$5M nodes for $k = 5, 6$, on $\sim$2M nodes for $k = 7$, and on less than 0.5M nodes for $k = 8$, in a matter of minutes or hours. As said above, CC does not scale to massive graphs and suffers from the "naive sampling barrier" that allows only for additive approximations. Finally, we shall mention the algorithm of [16] that in a few minutes can estimate clique counts with high accuracy on graphs with tens of millions of edges. We remark that that algorithm works *only* for cliques, while MOTIVO is general purpose and provides counts for *all* graphlets at once.

Preliminaries and notation. We denote the host graph by $G = (V, E)$, and we let $n = |V|$ and $m = |E|$. A *graphlet* is a connected graph $H = (V_H, E_H)$. A *treelet* T is a graphlet that is a tree. We let $k = |V_H|$. We denote by $\mathcal{H}$ the set of all k-node graphlets, i.e., all non-isomorphic connected graphs on k nodes. When needed we denote by H_i the i-th graphlet of $\mathcal{H}$. A colored graphlet has a color $c_u \in [k]$ associated to each one of its nodes u. A graphlet is *colorful* if its nodes have pairwise distinct colors. We denote by $C \subseteq [k]$ a subset of colors. We denote by (T, C) or T_C a colored treelet whose nodes span the set of colors C; we only consider colorful treelets, i.e., the case $|T|{=}|C|$. Often treelets and colored treelets are rooted at a node $r \in T$. Finally, d_v and $u \sim v$ will denote the degree of a node v in G and a neighbor u of v, respectively.

Paper organization. Section 2 reviews color coding and the CC algorithm. Section 3 introduces our data structures and techniques for accelerating color coding. Section 4 describes our adaptive sampling strategy. Section 5 concludes with our experiments.

2. COLOR CODING AND CC

The color coding technique was introduced in [4] to probabilistically detect paths and trees in a graph. The CC algorithm of [8, 9] is an extension of color coding that enables sampling colorful graphlet occurrences from G. It consists of a *build-up phase* and a *sampling phase.*

2.1 The build-up phase

The goal of this phase is to build a *treelet count table* that is the abstract "urn" used for sampling. First, we do a *coloring* of G: for each $v \in G$ independently, we draw uniformly at random a color $c_v \in [k]$. We then look at the treelets of G that are colorful. For each v and every rooted colored treelet T_C on up to k nodes, we want a count $c(T_C, v)$ of the number of copies of T_C in G that are rooted in v (note that we mean *non-induced* copies here). To this end, for each v we initialize $c(T_C, v) = 1$, where T is the trivial treelet on 1 node and $C = \{c_v\}$. For a T_C on $h > 1$ nodes, the count $c(T_C, v)$ is then computed via dynamic programming, as follows. First, T has a unique decomposition into two subtrees T' and T'' rooted respectively at the root r of T and at a child of r. The uniqueness is given by a total order over treelets (see next section). Now, since T' and T'' are smaller than T, their counts have already been computed for all possible colorings and all possible rootings in G. Then $c(T_C, v)$ is given by (see [9]):

$$c(T_C, v) = \frac{1}{\beta_T} \sum_{u \sim v} \sum_{\substack{C' \subset C \\ |C'| = |T'|}} c(T'_{C'}, v) \cdot c(T''_{C''}, u) \qquad (1)$$

where β_T is the number of subtrees of T isomorphic to T'' rooted at a child of r. CC employs (1) in the opposite way: it iterates over all pairs of counts $c(T'_{C'}, v)$ and $c(T''_{C''}, u)$ for all $u \sim v$, and if $T'_{C'}, T''_{C''}$ can be merged in a colorful treelet T_C, then it adds $c(T'_{C'}, v) \cdot c(T''_{C''}, u)$ to the count $c(T_C, v)$. This requires to perform a check-and-merge operation for each count pair, which is quite expensive (see below).

A simple analysis gives the following complexity bounds:

THEOREM 1. *([9], Theorem 5.1) The build-up takes time $O(a^k m)$ and space $O(a^k n)$, for some constant $a > 0$.*

The size of the dynamic programming table is a major bottleneck for CC: already for $k = 6$ and $n = 5$M, it takes 45GB of main memory [9].

2.2 The sampling phase

The goal of this phase is to sample *colorful* graphlet copies u.a.r. from G, using the treelet count table from the build-up phase. The key observation in [8, 9] is that we only need to sample colorful non-induced *treelet* copies; by taking the corresponding induced subgraph in G, we then obtain our induced graphlet copies. Colorful treelets are sampled via a multi-stage sampling, as follows. First, draw a node $v \in G$ with probability proportional to $\eta_v = \sum_{T_C} c(T_C, v)$. Second, draw a colored treelet T_C with probability proportional to $c(T_C, v)/\eta_v$. We want to sample a copy of T_C rooted at v. To this end we decompose T_C into $T'_{C'}$ and $T''_{C''}$, with $T'_{C'}$ rooted at the root r of T and $T''_{C''}$ at a child of r (see above). We then recursively sample a copy of $T'_{C'}$ rooted at v, and a copy of $T''_{C''}$ rooted at node $u \sim v$, where u is chosen with probability $c(T''_{C''}, u)/\sum_{z \sim v} c(T''_{C''}, z)$. Note that computing this probability requires listing all neighbors z of v, which takes time proportional to d_v. Finally, we combine $T'_{C'}$ and $T''_{C''}$ into a copy of T_C. One can see that this gives a colorful copy of T drawn uniformly at random from G.

Consider then a given k-graphlet H_i (e.g., the clique), and let c_i be the number of colorful copies of H_i in G. We can estimate c_i as follows. Let χ_i be the indicator random variable of the event that a graphlet sample x is an occurrence of H_i. It is easy to see that $\mathbb{E}[\chi_i] = c_i\, \sigma_i / t$, where σ_i is the number of spanning trees in H_i and t is the total number of colorful k-treelets of G. Both t and σ_i can be computed quickly, by summing over the treelet count table and via Kirchhoff's theorem (see below). We thus let $\hat{c}_i = t\, \sigma_i^{-1} \chi_i$, and $\mathbb{E}[\hat{c}_i] = c_i$. By standard concentration bounds we can then estimate c_i by repeated sampling. Note that the expected number of samples to find a copy of H_i grows as $1/c_i$. This is the additive error barrier of CC's sampling.

Estimators and errors. Finally, let us see how to estimate the number of total (i.e., *uncolored*) copies g_i of H_i in G, which is our final goal. First, note that the probability that a fixed subset of k nodes in G becomes colorful is $p_k = k!/k^k$. Therefore, if G contains g_i copies of H_i, and c_i is the number those copies that become colorful, then by linearity of expectation $\mathbb{E}[c_i] = p_k g_i$ (seeing c_i as a random variable). Hence, $\hat{g}_i = c_i/p_k$ is an unbiased estimator for g_i.

This is, indeed, the count estimate returned by CC and by MOTIVO.

For what concerns accuracy, the error given by $\hat{g}_i$ can be formally bounded via concentration bounds. An additive error bound is given by Theorem 5.3 of [9], which we slightly rephrase. Let $g = \sum_i g_i$ be the total number of induced k-graphlet copies in G. Then:

THEOREM 2 ([9], THEOREM 5.3). *For all* $\epsilon > 0$,

$$\Pr\Big[\big|\hat{g}_i - g_i\big| > \frac{2\epsilon g}{1-\epsilon}\Big] = \exp(-\Omega(\epsilon^2 g^{1/k})).$$

Since we aim at multiplicative errors, we prove a multiplicative bound, which is also tighter than Theorem 2 if the maximum degree Δ of G is small. We prove (see Appendix A):

THEOREM 3. *For all* $\epsilon > 0$,

$$\Pr\Big[\big|\hat{g}_i - g_i\big| > \epsilon\, g_i\Big] < 2\exp\Big(-\frac{2\epsilon^2}{(k-1)!}\frac{p_k\, g_i}{\Delta^{k-2}}\Big). \qquad (2)$$

In practice, $\hat{g}_i$ appears always concentrated. In other words, the coloring does not introduce a significant distortion. Note that this holds for a single coloring, i.e., for a single execution of CC. If one averages over γ executions with independent colorings, the probabilities decrease exponentially with γ.

3. SPEEDING UP COLOR CODING

This section details step-by-step the data structures and optimizations that are at the heart of MOTIVO's efficiency. First, we implemented a C++ porting of CC (which is in Java), translating all algorithms and data structures carefully to their closest C++ equivalent. This porting is our baseline benchmark. We then incrementally plugged in our data structures and optimizations, ultimately obtaining MOTIVO. Figures 1 and 2 show how the performance improves during the process. Note that MOTIVO uses 128-bit counts, while CC uses 64-bit counts which causes overflows (just the number of 6-stars centered in a node of degree 2^{16} is $\approx 2^{80}$).

Before moving on, we note that a perfectly fair porting of CC is not possible. This is because CC makes heavy use of fast specialized integer hash tables provided by the `fastutil`[2] library, which exists only in Java and seems to be crucial to its performance. Indeed, for the porting we tested three popular libraries – `google::sparse_hash_map` and `google::dense_hash_map` of the sparsehash library[3], and `std::unordered_map` from the C++ containers library. With the first two, the porting is up to 17× slower than CC, and with the latter one it is up to 7× slower. Nonetheless, after all optimizations are in place, MOTIVO is faster than CC and relatively insensitive to the hash table choice, with running times changing by at most 30%. We thus use `std::unordered_map` in the porting for the sake of measuring the impact of optimizations, and use the memory-parsimonious `google::sparse_hash_map` in MOTIVO.

3.1 Succinct data structures

The main objects manipulated by CC and MOTIVO are rooted colored treelets and their associated counts, which are stored in the treelet count table. We first describe their implementation in CC, then introduce the one of MOTIVO.

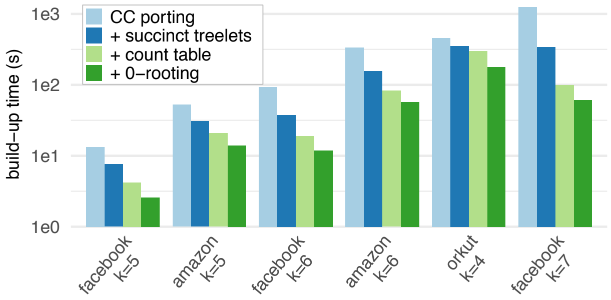

Figure 1: cumulative impact of our optimizations on the running time of the build-up phase.

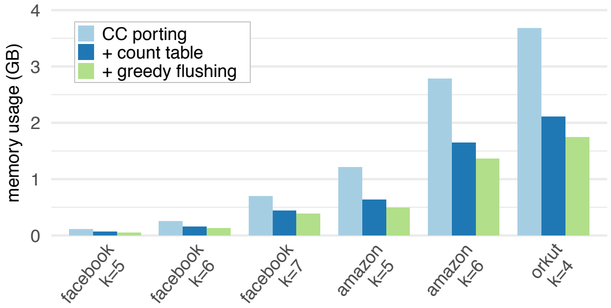

Figure 2: cumulative impact of our optimizations on the memory usage of the build-up phase.

The internals of CC. In CC, each T_C has a unique *representative instance*, that is a classic pointer-based tree data structure equipped with a structure storing the colors. The pointer to this instance acts as unique identifier for T_C. The treelet count table of CC is then implemented as follows: for each $v \in G$, a hash table maps the pointer of each T_C to the count $c(T_C, v)$, provided that $c(T_C, v) > 0$. Thus, each entry uses 128 bits – 64 for the pointer and 64 for the count – plus the overhead of the hash table. For computing $c(T_C, v)$, CC processes every neighbor $u \sim v$ as follows (see also Section 2.1). For every pair of counts $c(T'_{C'}, v)$ and $c(T''_{C''}, u)$ in the hash tables of v and u, check that $C' \cap C'' = \emptyset$, and that $T''_{C''}$ comes before the smallest subtree of $T'_{C'}$ in the total order of the treelets (see below). If these conditions hold, then $T'_{C'}$ and $T''_{C''}$ can be merged into a treelet T_C whose unique decomposition yields precisely $T'_{C'}$ and $T''_{C''}$. Then, the value of $c(T_C, v)$ in the hash table of v is incremented by $c(T'_{C'}, v) \cdot c(T''_{C''}, u)$. The expensive part is the check-and-merge operation, which CC does with a recursive algorithm on the treelet representative instances. This has a huge impact, since on a graph with a billion edges the check-and-merge is easily performed trillions of times. In fact, in our porting, check-and-merge operations consume from 30% to 70% of the build-up time.

Motivo's treelets. Let us now describe MOTIVO's data structures, starting with an uncolored treelet T rooted at $r \in T$. We encode T with the binary string s_T defined as follows. Perform a DFS traversal of T starting from r. Then the i-th bit of s_T is 1 (resp. 0) if the i-th edge is traversed moving away from (resp. towards) r. For all $k \leq 16$, this encoding takes at most 30 bits, which fits nicely in a 4-byte integer type (padded with 0s). The lexicographic ordering over the

[2] `http://fastutil.di.unimi.it/`
[3] `https://github.com/sparsehash/sparsehash`

s_T's gives a total ordering over the T's that is exactly the one used by CC. This ordering is also a tie-breaking rule for the DFS traversal: the children of a node are visited in the order given by their rooted subtrees. This implies that every T has a well-defined unique encoding s_T. Moreover, merging T' and T'' into T requires just concatenating $1, s_{T''}, s_{T'}$ in this order. This makes check-and-merge operations extremely fast: we measure a speedup ranging from 150× for $k = 5$ to 1000× for $k = 7$.

This succinct encoding supports the following operations:

- `getsize()`: return the number of vertices in T. This is one plus the Hamming weight of s_T, which can be computed in a single machine instruction (e.g., `POPCNT` from the SSE4 instruction set).
- `merge(`T'`, `T''`)`: merge two treelets T', T'' by appending T'' as a child of the root of T'. This requires just to concatenate $1, s_{T''}, s_{T'}$ in this order.
- `decomp(`T`)`: decompose T into T' and T''. This is the inverse of `merge` and is done by suitably splitting s_T.
- `sub(`T`)`: compute the value β_T of (1), i.e., the number of subtrees of T that (i) are isomorphic to the treelet T'' of the decomposition of T, and (ii) are rooted at some child of the root. This is done via bitwise `shift` and `and` operations on s_T.

A colored rooted treelet T_C is encoded as the concatenation s_{T_C} of s_T and of the characteristic vector s_C of C.[4] For all $k \leq 16$, s_{T_C} fits in 46 bits. Set-theoretical operations on C become bitwise operations over s_C (`or` for union, `and` for intersection). Finally, the lexicographical order of the s_{T_C}'s induce a total order over the T_C's, which we use in the count table (see below). An example of a colored rooted treelet and its encoding is given in Figure 3 (each node labelled with its color).

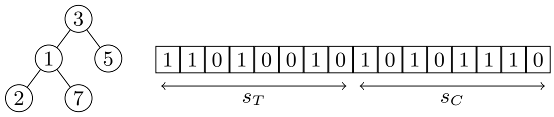

Figure 3: a colored rooted treelet and its encoding, shown for simplicity on just $8 + 8 = 16$ bits.

Motivo's count tables. In CC, treelet counts are stored in n hash tables, one for each node $v \in G$. In each table, the pair $(T_C, c(T_C, v))$ is stored using the pointer to the representative instance of T_C as key. This imposes the overhead of dereferencing a pointer before each check-and-merge operation to retrieve the actual structure of T_C. Instead of using a hash table, MOTIVO maintains the key-value pairs $(T_C, c(T_C, v))$ such that $c(T_C, v) > 0$ in a set of arrays, one for each $v \in G$ and for each treelet size $h \in [k]$. These arrays are sorted lexicographically w.r.t. the order of the keys described above. This makes iterating over the counts extremely fast and, since each key T_C is explicitly stored using its representation s_{T_C}, eliminates the need for dereferencing. The price to pay is that searching for a given T_C in the count table requires a binary search. However, this still takes only $O(k)$ time, since the whole record has length $O(6^k)$.[5] Recall that MOTIVO uses 128-bit counts[6], whereas CC uses 64-bit integers. This increases by 64 bits the space per pair compared to CC; however, MOTIVO saves 16 bits per pair by packing s_{T_C} into 48 bits, using a total of 176 bits per pair. Finally, in place of $c(T_C, v)$, MOTIVO actually stores the cumulative count $\eta(T_C, v) = \sum_{T'_{C'} \leq T_C} c(T'_{C'}, v)$. In this way each $c(T_C, v)$ can be recovered with negligible overhead, and the total count for a single node v (needed for sampling) is just at the end of the record.

MOTIVO's count table supports the following operations:

- `occ(`v`)`: returns the total number of colorful treelet copies rooted at v. Running time: $O(1)$.
- `occ(`T_C, v`)`: returns the total number of copies of T_C rooted at v. Running time: $O(k)$ via binary search.
- `iter(`T, v`)`: returns an iterator to the first pair $(T_C, c(T_C, v))$ stored in the table such that $T_C = (T, C)$. Running time: $O(k)$, plus $O(1)$ per accessed treelet.
- `iter(`T_C, v`)`: If $c(T_C, v) > 0$, returns an iterator to the pair $(T_C, c(T_C, v))$. When $c(T_C, v) = 0$, the returned iterator refers to the pair $T'_{C'}$, where $T'_{C'}$ if the first colored treelet that follows T_C for which $c(T'_{C'}, v) > 0$. Running time: $O(k)$, plus $O(1)$ per accessed treelet.
- `sample(`v`)`: returns a random colored treelet T_C with probability proportional to $c(T_C, v)/\eta_v$. This is used in the sampling phase. Running time $O(k)$: first we get η_v in $O(1)$ time (see above); then in $O(k)$ time we draw R u.a.r. from $\{1, \ldots, \eta_v\}$; finally, we search for the first pair (T_C, η) with $\eta \geq R$, and we return T_C.

0-rooting. Consider a colorful treelet copy in G that is formed by the nodes $v_1, \ldots, v_h$. In the count table, this treelet is counted in each one of the h records of $v_1, \ldots, v_h$, since it is effectively a colorful treelet rooted in each one of those nodes. Therefore, the treelet is counted h times. This is inevitable for $h < k$, since excluding some rooting would invalidate the dynamic programming (Equation 1). However, for $h = k$ we can store only one rooting and the sampling works just as fine. Thus, for $h = k$ we count only the k-treelets rooted at their node of color 0. This cuts the running time by $30\% - 40\%$, while reducing by a factor of k the size of the k-treelets records, and by $\approx 10\%$ the total space usage of MOTIVO.

3.2 Other optimizations

Greedy flushing. To further reduce memory usage, we use a greedy flushing strategy. Suppose we are currently building the table for treelets of size h. While being built, the record of v is actually stored in a hash table, which allows for efficient insertions. However, immediately after completion it is stored on disk in the compact form described above, but still unsorted w.r.t. the other records. The hash table is then emptied and memory released. When all records have been stored on disk, a second I/O pass sorts them w.r.t. their

[4] Given an universe U, the characteristic vector $\langle x_1, x_2, \ldots \rangle$ of a subset $S \subseteq U$ contains one bit x_i for each element $i \in U$, which is 1 if $i \in S$ and 0 otherwise.

[5] By Cayley's formula: there are $O(3^k k^{-3/2})$ rooted treelets on k vertices [19], and 2^k subsets of k colors.

[6] Tests on our machine show that summing 500k unsigned integers is 1.5× slower with 128-bit than with 64-bit integers.

corresponding node. At the end, the treelet count table is stored on disk without having entirely resided in memory. The price to pay is the time for sorting, which was at most 10% of the total time in all our runs.

Neighbor buffering. Our final optimization concerns sampling. In most graphs, MOTIVO natively achieves sampling rates of 10k samples per second or higher. But on some graphs, such as BerkStan or Orkut, we get only 100 or 1000 samples per second. The reason is the following. Those graphs contain a node v with a degree Δ much larger than any other node. Inevitably then, a large fraction of the treelets of G are rooted in v. This has two combined effects on the sampling phase (see Subsection 2.2). First, v will be frequently chosen as root. Second, upon choosing v will spend time $\Theta(\Delta)$ to sweep over its neighbors. The net effect is that the time to take one sample grows *superlinearly* with Δ, slowing down the sampling dramatically. To overcome this, we perform buffered sampling. If $d_v \geq 10^4$, then with a single sweep over v's neighbors, MOTIVO samples 100 neighbors at random instead of just one. It then returns the first, and caches the remaining 99 for the future. In this way, on large-degree nodes MOTIVO sweeps only once every 100 samples. As Figure 4 shows, this increases the sampling speed by ≈ 20× on Orkut and ≈ 40× on BerkStan.

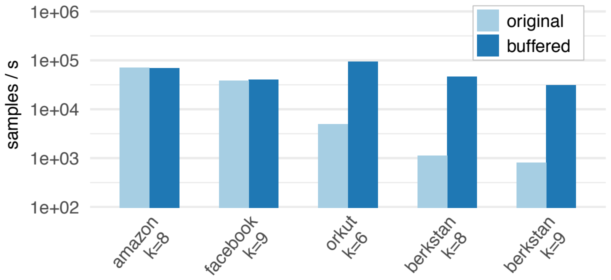

Figure 4: impact of neighbor buffering on sampling.

3.3 Implementation details

We describe some other implementation details of MOTIVO that, although not necessarily being "optimizations", are necessary for completeness and reproducibility.

Input graph. The graph G is represented by adjacency lists. Each list is a sorted static array of the vertex's neighbors; arrays of consecutive vertices are contiguous in memory. This allows for fast iterations over the set of outgoing edges of a vertex, and for $O(\log n)$-time edge-membership queries[7], that we need in the sampling phase to obtain the induced graphlet from the sampled treelet.

Multi-threading. Similarly to CC, MOTIVO makes heavy use of thread-level parallelism in both the build-up and sampling phases. For the build-up phase, for any given v the counts $c(\cdot, v)$ can be computed independently from each other, which we do using a pool of threads. As long as the number of remaining vertices is sufficiently large, each thread is assigned a (yet unprocessed) vertex v and will compute all the counts $c(T_C, v)$ for all pairs T_C. While this requires minimal synchronization, when the number of unprocessed vertices decreases below the amount of available threads, the above strategy is no longer advantageous as it would cause some of the threads to become idle. This can increase the time needed by the build-up phase if G exhibits skewed degree and/or treelet distributions. To overcome this problem, the last remaining vertices are handled differently: we allow multiple threads to concurrently compute different summands of the outermost sum of (1) for the same vertex v, i.e., those corresponding to the edges $(v, u) \in E$. Once all the incident edges of v have been processed, the partial sums are then combined together to obtain all the counts $c(\cdot, v)$. This reduces the running time by a few percentage points. For the sampling phase, samples are by definition independent and are taken by different threads.

Memory-mapped reads. In the build-up phase, to compute the count table for treelets of size h we must access the count tables of size j, for all $j < h$. For large instances, loading all those tables simultaneously in memory is infeasible. One option would be to carefully orchestrate I/O and computation, hoping to guarantee a small number of load/store operations on disk. We adopt a simpler solution: memory-mapped I/O. This delegates the I/O to the operating system in a manner that is transparent to MOTIVO, which sees all tables as if they resided in main memory. When enough memory is available this solution gives ideally no overhead. Otherwise, the operating system will reclaim memory by unloading part of the tables, and future requests to those parts will incur a page fault and prompt a reload from the disk. The overhead of this approach can be indeed measured via the number of page faults. This reveals that the total I/O volume due to page faults is less than 100MB, except for $k = 8$ on LiveJournal (34GB) and Yelp (8GB) and for $k = 6$ on Friendster (15GB). However, in those cases additional I/O is inevitable, as the total size of the tables (respectively 99GB, 90GB, and 61GB) is close to or even larger than the total memory available (about 60GB).

Alias method sampling. Recall that, to sample a colorful graphlet from G, we first sample a node v with probability proportional to the number of colorful k-treelets rooted at v (Subsection 2.2). We do this in time $O(1)$ by using the alias method [24], which requires building an auxiliary lookup table in time and space linear in the support of the distribution. In our case this means time and space $O(n)$; the table is built during the second stage of the build-up process. In practice, building the table takes negligible amounts of time (a fraction of a second out of several minutes).

Graphlets. In MOTIVO, each graphlet H is encoded as an adjacency matrix packed in a 128-bit integer. Since a graphlet is a simple graph, the $k \times k$ adjacency matrix is symmetric with diagonal 0 and can be packed in a $(k-1) \times \frac{k}{2}$ matrix if k is even and in a $k \times \frac{k-1}{2}$ matrix if k is odd (see e.g. [5]). The resulting triangular matrix can then be reshaped into a $1 \times \frac{k^2-k}{2}$ vector, which fits into 120 bits for all $k \leq 16$. In fact, one can easily compute a bijection between the pair of vertices of the graphlet and the indices $\{1, \ldots, 120\}$. Before encoding a graphlet, MOTIVO replaces it with a canonical representative from its isomorphism class, computed using the Nauty library [18].

Spanning trees. By default, MOTIVO computes the number of spanning trees σ_i of H_i in time $O(k^3)$ via Kirchhoff's matrix-tree theorem which relates σ_i to the determinant of a $(k-1) \times (k-1)$ submatrix of the Laplacian matrix of H_i. To compute the number σ_{ij} of occurrences of T_i in H_j (needed

[7] This is actually $O(\log d_u)$ where (u, v) is the edge being tested. In practice, it is often the case that $d_u \ll n$.

for our sampling algorithm AGS, see Section 4), we use an in-memory implementation of the build-up phase. The time taken is negligible for $k < 7$, but is significant for $k \geq 7$. For this reason, MOTIVO caches the σ_{ij} and stores them to disk for later reuse. This accelerates sampling by up to 10×.

3.4 Biased coloring

Finally, we describe an optimization, that we call "biased coloring", that can be used to manage graphs that would otherwise be too large. Suppose for simplicity that, for each treelet T on j nodes, each $v \in G$ appears in a relatively small number of copies of T, say $k^j/j!$. Then, given a set C of j colors, a copy of T is colored with C with probability $j!/k^j$. This implies that we will have an expected $\Theta(1)$ copies of T colored with C containing v, in which case the total table size (and the total running time) will approach the worst-case space bounds.

Suppose now we bias the distribution of colors. In particular, we give probability $\lambda \ll \frac{1}{k}$ to each color in $\{1, \ldots, k-1\}$. The probability that a given j-treelet copy is colored with C is then:

$$p_{k,j}(C) = \begin{cases} j!\lambda^j & \text{if } k \notin C \\ \sim j!\lambda^{j-1} & \text{if } k \in C \end{cases} \tag{3}$$

If λ is sufficiently small, then, for most T we will have a zero count at v; and most nonzero counts will be for a restricted set of colorings – those containing k. This reduces the number of pairs stored in the treelet count table, and consequently the running time of the algorithm. The price to pay is a loss in accuracy, since a lower p_k increases the variance of the number c_i of colorful copies of H_i. However, if n is large enough and most nodes v belong to even a small number of copies of H_i, then the *total* number of copies g_i of H_i is large enough to ensure concentration. In particular, by Theorem 3 the accuracy loss remains negligible as long as $\lambda^{k-1}n/\Delta^{k-2}$ is large (ignoring factors depending only on k). We can thus trade a $\Theta(1)$ factor in the exponent of the bound for a $\Theta(1)$ factor in both time and space, especially on large graphs where saving resources is precious. To find a good value for λ, one can start with $\lambda \ll 1/kn$ and grow it until a small but non-negligible fraction of counts are positive. At this point by Theorem 3 we have achieved concentration, and we can safely proceed to the sampling phase.

Impact. With $\lambda = 0.001$, the build-up time on FRIENDSTER (65M nodes, 1.8B edges) shrinks from 17 to 10 minutes (1.7×) for $k = 5$, and from 1.5 hours to 13 minutes (7×) for $k = 6$. In both cases, the main memory usage and the disk space usage decrease by at least 2×. The relative graphlet count error increases correspondingly, as shown in Figure 5 (see Section 5 for the error definition). For $k = 7$, the build takes 20 minutes – in this case we have no comparison term, as without biased coloring MOTIVO did not terminate a run within 2 hours. Note that FRIENDSTER has 30 (500) times the nodes (edges) of the largest graph managed by CC for the same values of k [9]. In our experiments (Section 5) biased coloring is disabled since mostly unnecessary.

4. ADAPTIVE GRAPHLET SAMPLING

This section describes AGS, our adaptive graphlet sampling algorithm for color coding. Recall that the main idea of CC is to build a sort of "urn" supporting a primitive sample() that returns a colorful k-treelet occurrence u.a.r. from G. The first step of AGS is to "refine" this interface with one urn for each possible k-treelet shape T. More precisely, for every k-treelet shape T our urn should support the following primitive:

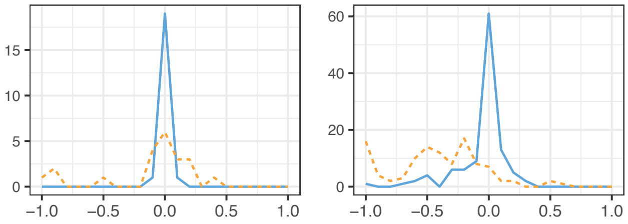

Figure 5: graphlet count error distribution of uniform and biased coloring (dashed), for k=5 and k=6.

- sample(T): return a colorful copy of T u.a.r. from G

With sample(T) one can selectively sample treelets of different shapes, and this can be used to virtually "delete" undesired graphlets from the urn. Let us try to convey the idea with a simple example. Imagine G contains just two types of colorful graphlets, H_1 and H_2, of which H_2 represents a tiny fraction p (say 0.01%). Using our original primitive, sample(), we will need $\Theta(1/p)$ calls before finding H_2. Suppose however H_1 and H_2 are spanned by treelets of different shape, say T_1 and T_2. We could then start by using sample(T_1), until we estimate accurately H_1. At this point we switch to sample(T_2), which completely ignores H_1 (since it is not spanned T_2), until we estimate accurately H_2 as well. In this way we can estimate accurately both graphlets with essentially $O(1)$ samples. Clearly, in general we have more than just two graphlets, and distinct graphlets may have the same spanning trees. Still, this adaptive sampling strategy strikingly outperforms naive sampling in the presence of rare graphlets. Moreover, AGS yields multiplicative guarantees on all graphlets, while taking only $O(k^2)$ times the minimum number of samples any algorithm must take (see below).

We now describe AGS in more detail. Recall from Section 2 that for every $v \in G$ we know $\texttt{occ}(T, v)$, the number of colorful copies of T rooted at v. Using this fact, one can easily restrict the sampling (Subsection 2.2) to a specific treelet T, thus drawing u.a.r. a colorful copy of T in G. This gives our primitive sample(T). We then start invoking sample(T), using the k-treelet T with the largest number of colorful occurrences. Eventually, some graphlet H_i spanned by T will appear enough times, say $\Theta(\frac{1}{\epsilon^2}\ln(\frac{1}{\delta}))$. We then say H_i is *covered*. Now, since we do not need more samples of H_i, we would like to continue with sample(T') for some T' that does *not* span H_i, as if we were asking to "delete" H_i from the urn. More precisely, we seek T' that minimizes the probability that by calling sample(T') we observe H_i.

The crux of AGS is that we can find T' as follows. First, we estimate the number g_i of colorful copies of H_i in G, which we can do since we have enough samples of H_i. Then, for each k-treelet T_j we estimate the number of copies of T_j that span a copy of H_i in G as $g_i\sigma_{ij}$, where σ_{ij} is the number of spanning trees of H_i isomorphic to T_j. We then divide this estimate by the number t_j of colorful copies of T_j in G, obtained summing $\texttt{occ}(T_j, v)$ over all $v \in G$. The result is an estimate of the probability that sample(T_j) spans a copy of H_i, and we choose the treelet T_{j^*} that minimizes this probability. More in general, we need the probability that sample(T_j) spans a copy of *some* graphlet among the ones covered so far, and to estimate g_i we must take into account that we have used different treelets along the sampling.

The pseudocode of AGS is listed below. A graphlet is marked as covered when it has appeared in at least $\bar{c}$ samples. For a union bound over all k-graphlets one would set $\bar{c} = O(\frac{1}{\epsilon^2}\ln(\frac{s}{\delta}))$ where $s = s_k$ is the number of distinct k-graphlets. In our experiments we set $\bar{c} = 1000$, which gives good accuracy on most graphlets. We denote by $H_1, \ldots, H_s$ the distinct k-node graphlets and by $T_1, \ldots, T_\varsigma$ the distinct k-node treelets.

```
Algorithm  AGS($\epsilon, \delta$)
 1: $(c_1, \ldots, c_s) \leftarrow (0, \ldots, 0)$                ▷ graphlet counts
 2: $(w_1, \ldots, w_s) \leftarrow (0, \ldots, 0)$                ▷ graphlet weights
 3: $\bar{c} \leftarrow \lceil \frac{4}{\epsilon^2} \ln(\frac{2s}{\delta}) \rceil$            ▷ covering threshold
 4: $C \leftarrow \emptyset$                                      ▷ graphlets covered
 5: $T_j \leftarrow$ an arbitrary treelet type
 6: while $|C| < s$ do
 7:     for each $i'$ in $1, \ldots, s$ do
 8:         $w_{i'} \leftarrow w_{i'} + \sigma_{i'j}/t_j$
 9:     $T_G \leftarrow$ an occurrence of $T_j$ drawn u.a.r. in $G$
10:     $H_i \leftarrow$ the graphlet type spanned by $T_G$
11:     $c_i \leftarrow c_i + 1$
12:     if $c_i \geq \bar{c}$ then                   ▷ switch to a new treelet $T_j$
13:         $C \leftarrow C \cup \{i\}$
14:         $j^* \leftarrow \arg\min_{j'=1,\ldots,\varsigma} \frac{1}{t_{j'}} \sum_{i' \in C} \sigma_{i'j'} c_{i'}/w_{i'}$
15:         $T_j \leftarrow T_{j^*}$
16: return $(\frac{c_1}{w_1}, \ldots, \frac{c_s}{w_s})$
```

4.1 Approximation guarantees

We prove that, if AGS chooses the "right" treelet T_{j^*}, then we obtain multiplicative error guarantees. Formally:

THEOREM 4. *If the tree T_{j^*} chosen by AGS at line 14 minimizes* Pr[sample(T_j) *spans a copy of some $H_i \in C$*] *then, with probability $(1-\delta)$, when AGS stops c_i/w_i is a multiplicative $(1 \pm \epsilon)$-approximation of g_i for all $i = 1, \ldots, s$.*

The proof requires a martingale analysis and is deferred to Appendix B. We stress that the guarantees hold for all graphlets, irrespective of their relative frequency. In practice, AGS gives accurate counts for many or almost all graphlets at once, depending on the graph (see Section 5).

4.2 Sampling efficiency

Let us turn to the sampling efficiency of AGS. We first observe that, in some cases, AGS does no better than naive sampling. However, this is not a limitation of AGS itself: it holds for *any* algorithm based solely on sampling treelets via the primitive sample(T). This class of algorithms is quite natural, and indeed includes the graphlet sampling algorithms of [17, 26, 27]. Formally, we prove:

THEOREM 5. *For any constant $k \geq 2$, there are graphs G in which some graphlet H represents a fraction $p_H = 1/\operatorname{poly}(n) = \Omega(n^{1-k})$ of all graphlet copies, and any algorithm needs $\Omega(1/p_H)$ calls to* sample(T) *in expectation to just find one copy of H.*

PROOF. Let T and H be the path on k nodes. Let G be the $(n-k+2, k-2)$ lollipop graph; so G is formed by a clique on $n-k+2$ nodes and a dangling path on $k-2$ nodes, connected by an arc. G contains $\Theta(n^k)$ non-induced occurrences of T in G, but only $\Theta(n)$ induced occurrences of H (all those formed by the $k-2$ nodes of the dangling path, the adjacent node of the clique, and any other node in the clique). Since there are at most $\Theta(n^k)$ graphlets in G, then H forms a fraction $p_H = \Theta(n^{1-k})$ of these. Obviously T is the only spanning tree of H; however, an invocation of sample(G, T) returns H with probability $\Theta(n^{1-k})$ and thus we need $\Theta(n^{k-1}) = \Theta(1/p_H)$ samples in expectation before obtaining H. One can make p_H larger by considering the $(n', n-n')$ lollipop graph for larger values of n'. □

Theorem 5 rules out good *absolute* bounds on the number of samples used by AGS. However, we can show that AGS is close to the best possible, clairvoyant algorithm based on sample(T). By clairvoyant we mean that the algorithm knows in advance how many sample(T_j) calls to make for every treelet T_j in order to get the desired bounds with the minimum total number of calls. Formally, we prove:

THEOREM 6. *If the tree T_{j^*} chosen by AGS at line 14 minimizes* Pr[sample(T_j) *spans a copy of some $H_i \in C$*], *then AGS makes a number of calls to* sample() *that is at most $O(\ln(s)) = O(k^2)$ times the minimum needed to ensure that every graphlet H_i appears in $\bar{c}$ samples in expectation.*

The proof of the theorem relies on a fractional set cover and can be found in Appendix C.

5. EXPERIMENTAL RESULTS

In this section we compare the performance of MOTIVO to CC [9] which, as said, is the current state of the art. For readability, we give plots for a subset of datasets that are representative of the entire set of results.

Set-up. We ran all our experiments on a commodity machine equipped with 64GB of main memory and 48 Intel Xeon E5-2650v4 cores at 2.5GHz with 30MB of L3 cache. We allocated 880GB of secondary storage on a Samsung SSD850 solid-state drive, dedicated to the treelet count tables of MOTIVO. Table 1 shows the graphs on which we tested MOTIVO, and the largest tested value of k. All graphs were made undirected and converted to the MOTIVO binary format. For each graph we ran MOTIVO for all $k = 5, 6, 7, 8, 9$, or until the build time did not exceed 1.5 hours; except for TWITTER and LIVEJOURNAL, where we did $k = 5, 6$ regardless of time.

Table 1: our graphs (* = with biased coloring)

graph	M nodes	M edges	source	k
FACEBOOK	0.1	0.8	MPI-SWS	9
BERKSTAN	0.7	6.6	SNAP	9
AMAZON	0.7	3.5	SNAP	9
DBLP	0.9	3.4	SNAP	9
ORKUT	3.1	117.2	MPI-SWS	7
LIVEJOURNAL	5.4	49.5	LAW	8
YELP	7.2	26.1	YLP	8
TWITTER	41.7	1202.5	LAW	6 (7*)
FRIENDSTER	65.6	1806.1	SNAP	6 (7*)

Ground truth. We computed exact 5-graphlet counts for FACEBOOK, DBLP, AMAZON, LIVEJOURNAL and ORKUT by running the ESCAPE algorithm [20]. On the remaining graphs ESCAPE died by memory exhaustion or did not return within 24 hours. For $k > 5$ and/or larger graphs, we averaged the counts given by MOTIVO over 20 runs, 10 using naive sampling and 10 using AGS.

5.1 Computational performance

Build-up time. The table below shows the speedup of MOTIVO's build-up phase over the original Java implementation of CC; dashes mean that CC failed by memory exhaustion or 64-bit integer overflow. We do not report TWITTER and FRIENDSTER, as CC failed even for $k = 5$. MOTIVO is 2×-5× faster than CC on 5 out of 7 graphs, and never slower on the other ones.

Table 2: MOTIVO vs. CC build-up phase speedup

graph	k=5	k=6	k=7	k=8	k=9
FACEBOOK	4.2	3.4	3.9	5.5	10.0
BERKSTAN	2.2	-	-	-	-
AMAZON	2.1	1.7	1.5	1.6	2.2
DBLP	2.1	1.4	1.7	2.2	3.9
ORKUT	5.6	-	-		
LIVEJOURNAL	3.1	3.1	-	-	
YELP	2.4	-	-	-	

Count tables size. The table below shows the ratio between the main memory footprint of CC and the total external memory usage of MOTIVO; both are indicators of the total count table size. The footprint of CC is computed as the smallest JVM heap size that allowed it to run. In almost all cases MOTIVO saves a factor of 2, in half of the cases a factor of 5, and on YELP, the largest graph managed by CC, a factor of 8. For $k = 7$, CC failed on 6 over 9 graphs, while MOTIVO processed all of them with a space footprint of less than 12GB (see below).

Table 3: MOTIVO vs. CC table size shrinkage factor

graph	k=5	k=6	k=7	k=8	k=9
FACEBOOK	108.7	87.5	54.5	17.3	7.1
BERKSTAN	36.5	-	-	-	-
AMAZON	6.6	3.3	2.1	1.1	1.0
DBLP	6.2	4.0	2.4	1.1	1.0
ORKUT	8.5	-	-		
LIVEJOURNAL	11.4	3.8	-	-	
YELP	8.0	-	-	-	

Sampling speed. The table below shows the sampling speedup of MOTIVO's naive sampling with respect to CC. MOTIVO is always 10× faster, and even 160× faster on YELP. This means MOTIVO gives more accurate estimates for a fixed time budget, even though in the sampling phase it must access the count tables from disk.

Table 4: MOTIVO vs. CC sampling rate speedup

graph	k=5	k=6	k=7	k=8	k=9
FACEBOOK	14.9	13.2	9.4	50.0	115.9
BERKSTAN	29.3	-	-	-	-
AMAZON	60.7	12.6	13.2	13.2	16.5
DBLP	17.7	11.4	10.1	44.8	88.6
ORKUT	29.2	-	-		
LIVEJOURNAL	31.8	28.5	-	-	
YELP	159.6	-	-	-	

Additional remarks. MOTIVO runs in minutes on graphs that CC could not even process, and in less than one hour for all but the largest instance. The contrast with ESCAPE [20] and [6, 12, 15, 25] is even starker, as those algorithms take entire days even for graphs 10–100 times smaller. We also point out that, unlike all these algorithms, MOTIVO's behavior is very predictable, as shown in Figure 6.

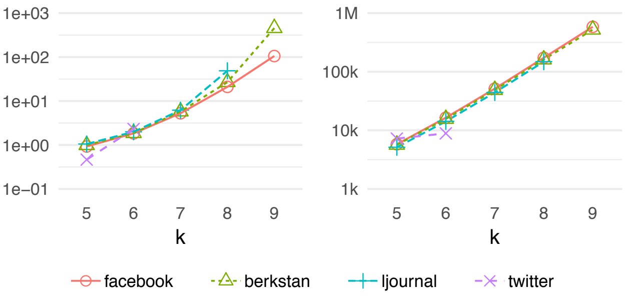

Figure 6: MOTIVO's build-up time (seconds per million edge) and space usage (bits per input node).

5.2 Accuracy

We assess the accuracy of MOTIVO's count estimates, and in particular of AGS against the naive sampling strategy. All plots below report the average over 10 runs, with whiskers for the 10% and 90% percentiles. Naive sampling is shown by the left bars, and AGS by the right bars. Note that MOTIVO's naive sampling strategy and CC's sampling algorithm are exactly the same algorithm, so for a given number of samples they give the same output. However, MOTIVO is much faster than CC and so takes many more samples per unit of time. Hence, the accuracy of CC is dominated by the accuracy of MOTIVO.

A necessary remark. The accuracy of estimates obviously depends on the number of samples taken. One option would be to fix an absolute budget, say 1M samples. Since however for $k = 5$ there are only 21 distinct graphlets and for $k = 8$ there are over 10k, we would certainly have much higher accuracy in the first case. As a compromise, we tell MOTIVO to spend in sampling the same amount of time taken by the build-up phase. This is also what an "optimal" time allocation strategy would do – statistically speaking, if we have a budget of 100 seconds and the build-up takes 5 seconds, we would perform 10 runs with 5 seconds of sampling each and average over the estimates.

Error in ℓ_1 norm. First, we evaluate how accurately MOTIVO reconstructs the global k-graphlet distribution. If $\mathbf{f} = (f_1, \ldots, f_s)$ are the ground-truth graphlet frequencies, and $\hat{\mathbf{f}} = (\hat{f}_1, \ldots, \hat{f}_s)$ their estimates, then the ℓ_1 error is $\ell_1(\mathbf{f}, \hat{\mathbf{f}}) = \sum_{i=1}^{s} |\hat{f}_i - f_i|$. In our experiments, the ℓ_1 error was below 5% in all cases, and below 2.5% for all $k \leq 7$.

Single-graphlet count error. The count error of H is:

$$\mathrm{err}_H = \frac{\hat{c}_H - c_H}{c_H} \tag{4}$$

where c_H is the ground-truth count of H and $\hat{c}_H$ its estimate. Thus $\mathrm{err}_H = 0$ means a perfect estimate, and $\mathrm{err}_H = -1$ means the graphlet is missed. Figure 7 shows the distribution of err_H for one run, for naive sampling (top) and AGS (bottom), as k increases from left to right. AGS is much more accurate, especially for larger values of k, and CC's naive sampling misses many graphlets. (We will give a very

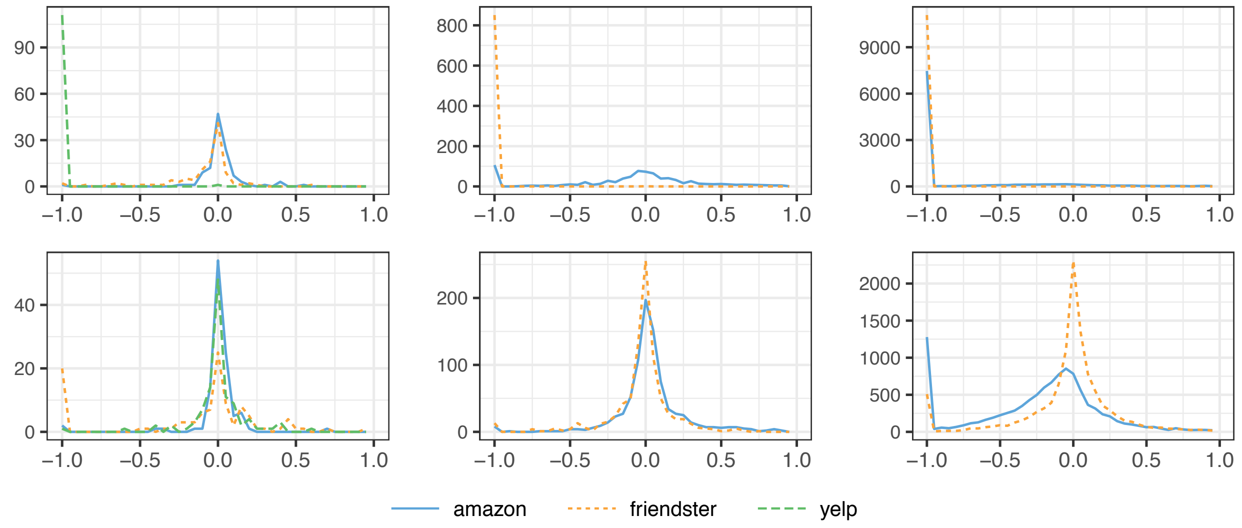

Figure 7: distribution of graphlet count error for $k = 6, 7, 8$. Top: naive sampling. Bottom: AGS.

Consider now AGS(ϵ, δ). Recall that we are looking at a *fixed* graphlet H_i (which here does *not* denote the graphlet sampled at line 10). Note that $\sum_{\tau=1}^{t} X_\tau$ is exactly the value of c_i after t executions of the main cycle (see line 11). Similarly, note that $\sum_{\tau=1}^{t} P_\tau$ is the value of $g_i \cdot w_i$ after t executions of the main cycle: indeed, if $Y_j^{t-1} = 1$, then at step τ we add to w_i the value $\frac{\sigma_{ij}}{t_j}$ (line 8), while the probability that a sample of T_j yields H_i is exactly $\frac{g_i \sigma_{ij}}{t_j}$. Therefore, after the main cycle has been executed t times, $Z_t = \sum_{\tau=1}^{t}(X_t - P_t)$ is the value of $c_i - g_i w_i$.

Now to the bounds. Suppose that, when AGS(ϵ, δ) returns, $\frac{c_i}{w_i} \geq g_i(1+\epsilon)$, i.e., $c_i(1 - \frac{\epsilon}{1+\epsilon}) \geq g_i w_i$. On the one hand this implies that $c_i - g_i w_i \geq c_i \frac{\epsilon}{1+\epsilon}$, i.e., $Z_t \geq c_i \frac{\epsilon}{1+\epsilon}$; and since upon termination $c_i = \bar{c}$, this means $Z_t \geq \bar{c}\frac{\epsilon}{1+\epsilon}$. On the other hand it implies $g_i w_i \leq c_i(1 - \frac{\epsilon}{1+\epsilon})$, i.e., $\sum_{\tau=1}^{t} P_\tau \leq c_i(1 - \frac{\epsilon}{1+\epsilon})$; again since upon termination $c_i = \bar{c}$, this means $\sum_{\tau=1}^{t} P_\tau \leq \bar{c}(1 - \frac{\epsilon}{1+\epsilon})$. We can then invoke Lemma 1 with $z = \bar{c}\frac{\epsilon}{1+\epsilon}$ and $v = \bar{c}(1 - \frac{\epsilon}{1+\epsilon})$, and since $v + z = \bar{c}$ we get:

$$\Pr\Big[\frac{c_i}{w_i} \geq g_i(1+\epsilon)\Big] \leq \exp\Big[-\frac{(\bar{c}\frac{\epsilon}{1+\epsilon})^2}{2\bar{c}}\Big] \tag{7}$$

$$= \exp\Big[-\frac{\epsilon^2 \bar{c}}{2(1+\epsilon)^2}\Big] \tag{8}$$

but $\frac{\epsilon^2 \bar{c}}{2(1+\epsilon)^2} \geq \frac{\epsilon^2}{2(1+\epsilon)^2} \frac{4}{\epsilon^2} \ln\big(\frac{2s}{\delta}\big) \geq \ln\big(\frac{2s}{\delta}\big)$ and thus the probability above is bounded by $\frac{\delta}{2s}$.

Suppose instead that, when AGS(ϵ, δ) returns, $\frac{c_i}{w_i} \leq g_i(1-\epsilon)$, i.e., $c_i(1 + \frac{\epsilon}{1-\epsilon}) \leq g_i w_i$. On the one hand this implies that $c_i - g_i w_i \geq \frac{\epsilon}{1-\epsilon} c_i$, that is, upon termination we have $-Z_t \geq \frac{\epsilon}{1-\epsilon}\bar{c}$. Obviously $(-Z_t)_{t \geq 0}$ is a martingale too with respect to the filter $(\mathcal{F}_t)_{t\geq 0}$, and therefore Lemma 1 still holds if we replace Z_t with $-Z_t$. Let then $t_0 \leq t$ be the first step where $-Z_{t_0} \geq \frac{\epsilon}{1-\epsilon}\bar{c}$; since $|Z_t - Z_{t-1}| \leq 1$, it must be $-Z_{t_0} < \frac{\epsilon}{1-\epsilon}\bar{c} + 1$. Moreover $\sum_{\tau=1}^{t} X_\tau$ is nondecreasing in t, so $\sum_{\tau=1}^{t_0} X_\tau \leq \bar{c}$. It follows that $\sum_{\tau=1}^{t_0} P_\tau = -Z_{t_0} + \sum_{\tau=1}^{t_0} X_\tau < \frac{\epsilon}{1-\epsilon}\bar{c} + 1 + \bar{c} = \frac{1}{1-\epsilon}\bar{c} + 1$. Invoking again Lemma 1 with $z = \frac{\epsilon}{1-\epsilon}\bar{c}$ and $v = \frac{1}{1-\epsilon}\bar{c} + 1$, we obtain:

$$\Pr\Big[\frac{c_i}{w_i} \leq g_i(1-\epsilon)\Big] \leq \exp\Big[-\frac{(\bar{c}\frac{\epsilon}{1-\epsilon})^2}{2(\frac{1+\epsilon}{1-\epsilon}\bar{c} + 1)}\Big] \tag{9}$$

$$\leq \exp\Big[-\frac{\epsilon^2 \bar{c}^2}{2(1+\bar{c})}\Big] \tag{10}$$

but since $\bar{c} \geq 4$ then $\frac{\bar{c}}{1+\bar{c}} \geq \frac{4}{5}$ and so $\frac{\epsilon^2 \bar{c}^2}{2(1+\bar{c})} \geq \frac{2\epsilon^2 \bar{c}}{5}$. By replacing $\bar{c}$ we get $\frac{2\epsilon^2 \bar{c}}{5} \geq \frac{2\epsilon^2}{5} \frac{4}{\epsilon^2} \ln\big(\frac{2s}{\delta}\big) > \ln\big(\frac{2s}{\delta}\big)$ and thus once again the probability of deviation is bounded by $\frac{\delta}{2s}$.

By a simple union bound, the probability that $\frac{c_i}{w_i}$ is not within a factor $(1 \pm \epsilon)$ of g_i is at most $\frac{\delta}{s}$. The theorem follows by a union bound on all $i \in [s]$.

C. PROOF OF THEOREM 6

For each $i \in [s]$ and each $j \in [\varsigma]$ let a_{ji} be the probability that sample(T_j) returns a copy of H_i. Note that $a_{ji} = g_i \sigma_{ij} / t_j$, the fraction of colorful copies of T_j that span a copy of H_i. Our goal is to allocate, for each T_j, the number x_j of calls to sample(T_j), so that (1) the total number of calls $\sum_j x_j$ is minimised and (2) each H_i appears at least $\bar{c}$ times in expectation. Formally, let $\mathbf{A} = (a_{ji})^\intercal$, so that columns correspond to treelets T_j and rows to graphlets H_i, and let $\mathbf{x} = (x_1, \ldots, x_\varsigma) \in \mathbb{N}^\varsigma$. We obtain the following integer program:

$$\begin{cases} \min \mathbf{1}^\intercal \mathbf{x} \\ \text{s.t. } \mathbf{A}\mathbf{x} \geq \bar{c}\,\mathbf{1} \\ \quad\;\; \mathbf{x} \in \mathbb{N}^\varsigma \end{cases}$$

We now describe the natural greedy algorithm for this problem; it turns out that this is precisely AGS. The algorithm proceeds in discrete time steps. Let $\mathbf{x}^0 = \mathbf{0}$, and for all $t \geq 1$ denote by $\mathbf{x}^t$ the partial solution after t steps. The vector $\mathbf{A}\mathbf{x}^t$ is an s-entry column whose i-th entry is the expected number of occurrences of H_i drawn using the sample allocation given by $\mathbf{x}^t$. We define the vector of residuals at time t as $\mathbf{c}^t = \max(\mathbf{0}, \mathbf{c} - \mathbf{A}\mathbf{x}^t)$, and for compactness we let $c^t = \mathbf{1}^\intercal \mathbf{c}^t$. Note that $\mathbf{c}^0 = \bar{c}\,\mathbf{1}$ and $c^0 = s\bar{c}$. Finally, we let vivid account of this difference below). Inevitably, the error spreads out with k; recall that the total number of distinct 8-graphlets is over 10^4.

Number of accurate graphlets. For a complementary view, we consider the number of graphlets whose estimate is within ±50% of the ground-truth value (Figure 8). This number easily often reaches the thousands, and for $k = 9$ even hundreds of thousands (note that the plot is in log-scale). We remind the reader that all this is carried out in minutes or, in the worst case, in two hours. Alternatively, we can look at these numbers in relative terms, that is, as a fraction of the total number of distinct graphlets in the ground truth (Figure 8). On all graphs except BerkStan, this ratio is over 90% of graphlets for $k = 6$, over 75% of graphlets for $k = 7$, and over 50% of graphlets for $k = 8$, for either naive sampling or AGS. The choice of 50% is just to deliver the picture, but we remark that such an error is achieved simultaneously for thousand of distinct graphlets whose counts differ by many orders of magnitude.

5.3 Breaking the sampling barrier with AGS

Finally, we show how AGS breaks the additive error barrier of naive sampling. The best example is the Yelp graph. Look again at Figure 8. For $k = 8$, over 99.9996% of the k-graphlets are stars. Consequently, in our experiments naive sampling finds only the star graphlet. This means that naive sampling gives accurate estimates for only 1 graphlet, that is, 0.01% of the total. Put in other terms, it misses 9999 out of 10000 graphlets. Instead, AGS returns fair estimates for 9645 graphlets, that is, 87% of the total.

The contrast is even sharper if we look at the frequency of the graphlets found by the two algorithms. These are shown in Figure 9 (to filter out noise, we consider only graphlets appearing in at least 10 samples). Naive sampling finds only graphlets with frequency at least 99.9996% (that is, again, the star). AGS finds graphlets with frequency below 10^{-21}, that is, seven orders of magnitude smaller. To convey the idea, to find those graphlets naive sampling would need $\approx 3 \cdot 10^3$ years even if running at 10^9 samples per second.

Let us make a final remark. On some graphs, AGS is slightly worse than naive sampling. This is expected: AGS is designed for skewed graphlet distributions, and loses ground on flatter ones. As a sanity check, we computed the ℓ_2 norm of the graphlet distributions. The three graphs where AGS beats naive sampling by a largest margin, BerkStan, Yelp and Twitter, have for all k the highest ℓ_2 norms (> .99). Symmetrically, Facebook, Dblp and Friendster, have for all k the three lowest ℓ_2 norms, and there AGS performs slightly worse than naive sampling.

6. CONCLUSIONS

We have shown how the color coding technique, despite its simplicity (or perhaps thanks to it), can be harnessed to scale motif counting to truly massive graphs. Thanks to color coding we can mine motifs in graphs with billions of edges, with approximation guarantees previously out of reach, using just ordinary hardware. These results can be seen as the first steps in the direction of *unexpensive large-scale motif mining*. We believe it may be possible to scale even further, by devising appropriate optimizations and algorithmic solutions; and especially by reducing the space usage, which is still a bottleneck of this approach.

APPENDIX

A. PROOF OF THEOREM 3

We use a concentration bound for dependent random variables from [13]. Let $\mathcal{V}_i$ be the set of copies of H_i in G. For any $h \in \mathcal{V}_i$ let X_h be the indicator random variable of the event that h becomes colorful. Let $c_i = \sum_{h \in \mathcal{V}_i} X_h$; clearly $\mathbb{E}[c_i] = p_k|\mathcal{V}_i| = p_k n_i$. Note that for any $h_1, h_2 \in \mathcal{V}_i$, X_{h_1}, X_{h_2} are independent if and only if $|V(h_1) \cap V(h_2)| \leq 1$, i.e., if h_1, h_2 share at most one node. For any $u, v \in G$ let then $g(u,v) = |\{h \in \mathcal{V}_i : u, v \in h\}|$, and define $\chi_k = 1 + \max_{u,v \in G} g(u,v)$. By standard counting argument one can see that $\max_{u,v \in G} g(u,v) \leq (k-1)!\Delta^{k-2} - 1$ and thus $\chi_k \leq (k-1)!\Delta^{k-2}$. The bound then follows immediately from Theorem 3.2 of [13] by setting $t = \epsilon c_i$, $(b_\alpha - a_\alpha) = 1$ for all $\alpha = h \in \mathcal{V}_i$, and $\chi^*(\Gamma) \leq \chi_k \leq (k-1)!\Delta^{k-2}$.

B. PROOF OF THEOREM 4

The proof requires a martingale analysis, since the distribution from which we draw the graphlets changes over time. We use a martingale tail inequality originally from [14] and stated (and proved) in the following form in [3, p. 1476]:

Theorem 7. ([3], Theorem 2.2). *Let $(Z_0, Z_1, \ldots)$ be a martingale with respect to the filter $(\mathcal{F}_\tau)_{t \geq 0}$. Suppose that $Z_{\tau+1} - Z_\tau \leq M$ for all τ, and write $V_t = \sum_{\tau=1}^{t} \mathrm{Var}[Z_\tau|\mathcal{F}_{\tau-1}]$. Then for any $z, v > 0$ we have:*

$$\Pr\left[\exists\, t : Z_t \geq Z_0 + z, V_t \leq v\right] \leq \exp\left[-\frac{z^2}{2(v + Mz)}\right] \quad (5)$$

We now plug into the formula of Theorem 7 the appropriate quantities. In what follows we *fix* a graphlet H_i and analyse the concentration of its estimate. Unless necessary, we drop the index i from the notation.

A. For $t \geq 1$ let X_t be the indicator random variable of the event that H_i is the graphlet sampled at step t (line 10 of AGS).

B. For $t \geq 0$ let Y_j^t be the indicator random variable of the event, at the end of step t, the treelet to be sampled at the next step is T_j.

C. For $t \geq 0$ let $\mathcal{F}_t$ be the event space generated by the random variables $Y_j^\tau : j \in [\varsigma]$, $\tau = 0, \ldots, t$. For any random variable Z, then, $\mathbb{E}[Z \,|\, \mathcal{F}_t] = \mathbb{E}[Z \,|\, Y_j^\tau : j \in [\varsigma], \tau = 0, \ldots, t]$, and $\mathrm{Var}[Z \,|\, \mathcal{F}_t]$ is defined analogously.

D. For $t \geq 1$ let $P_t = \mathbb{E}[X_t|\mathcal{F}_{t-1}]$ be the probability that the graphlet sampled at the t-th invocation of line 10 is H_i, as a function of the events up to time $t-1$. It is immediate to see that $P_t = \sum_{j=1}^{\varsigma} Y_j^{t-1} a_{ji}$.

E. Let $Z_0 = 0$, and for $t \geq 1$ let $Z_t = \sum_{\tau=1}^{t}(X_t - P_t)$. Now, $(Z_t)_{t \geq 0}$ is a martingale with respect to the filter $(\mathcal{F}_t)_{t \geq 0}$, since Z_t is obtained from Z_{t-1} by adding X_t and subtracting P_t which is precisely the expectation of X_t w.r.t. $\mathcal{F}_{t-1}$.

F. Let $M = 1$, since $|Z_{t+1} - Z_t| = |X_{t+1} - P_t| \leq 1$ for all t.

Finally, notice that $\mathrm{Var}[Z_t|\mathcal{F}_{t-1}] = \mathrm{Var}[X_t|\mathcal{F}_{t-1}]$, since again $Z_t = Z_{t-1} + X_t - P_t$, and both Z_{t-1} and P_t are a function of $\mathcal{F}_{t-1}$, so their variance w.r.t. $\mathcal{F}_{t-1}$ is 0. Now, $\mathrm{Var}[X_t|\mathcal{F}_{t-1}] = P_t(1 - P_t) \leq P_t$; and therefore we have $V_t = \sum_{\tau=1}^{t} \mathrm{Var}[Z_\tau \,|\, \mathcal{F}_{\tau-1}] \leq \sum_{\tau=1}^{t} P_\tau$. Then by Theorem 7:

Lemma 1. *For all $z, v > 0$ we have*

$$\Pr\left[\exists\, t : Z_t \geq z, \sum_{\tau=1}^{t} P_\tau \leq v\right] \leq \exp\left[-\frac{z^2}{2(v + z)}\right] \quad (6)$$

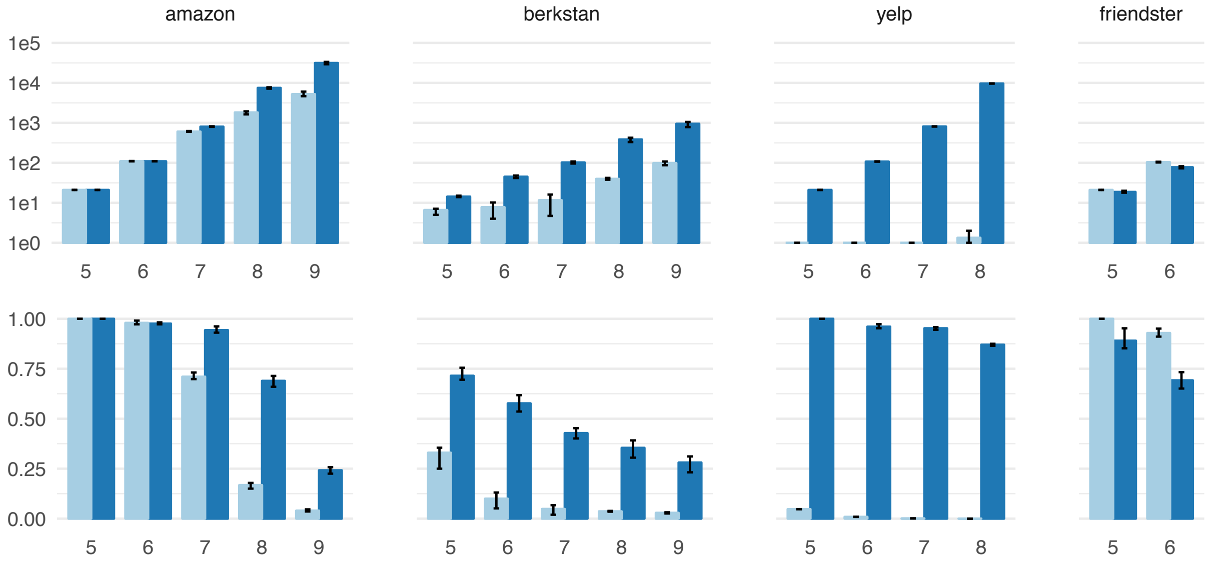

Figure 8: graphlet counts with error within ±50% (dark bars = AGS). Top: absolute number. Bottom: as a fraction.

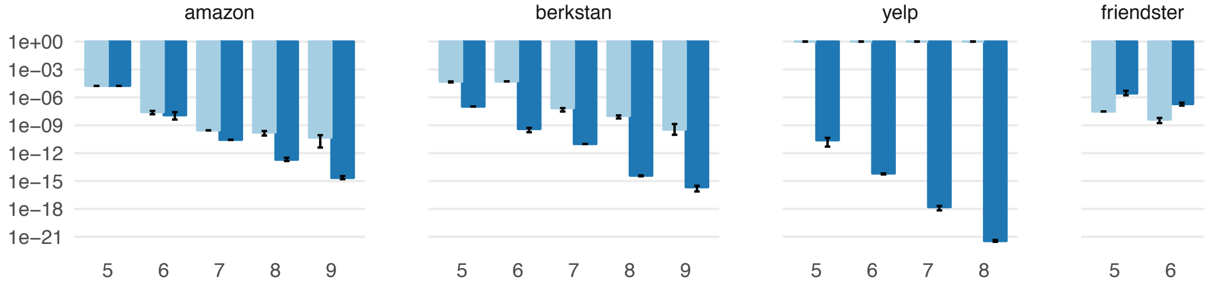

Figure 9: frequency of the rarest graphlet appearing in 10 or more samples (dark bars = AGS).

$U^t = \{i : c_i^t > 0\}$; this is the set of graphlets not yet covered at time t, and clearly $U^0 = [s]$.

At the t-th step the algorithm chooses the T_{j^*} such that sample(T_{j^*}) spans an uncovered graphlet with the highest probability, by computing:

$$j^* := \arg\max_{j=1,\ldots,\varsigma} \sum_{i \in U_t} a_{ji} \tag{11}$$

It then lets $\mathbf{x}^{t+1} = \mathbf{x} + \mathbf{e}_{j^*}$, where $\mathbf{e}_{j^*}$ is the indicator vector of j^*, and updates $\mathbf{c}^{t+1}$ accordingly. The algorithm stops when $U^t = \emptyset$, since then $\mathbf{x}^t$ is a feasible solution. We prove:

Lemma 2. *Let z be the cost of the optimal solution. Then the greedy algorithm returns a solution of cost $O(z \ln(s))$.*

Proof. Let $w_j^t = \sum_{i \in U_t} a_{ji}$ (note that this is a *treelet* weight). For any $j \in [\varsigma]$ denote by $\Delta_j^t = c^t - c^{t+1}$ the decrease in overall residual weight we would obtain if $j^* = j$. Note that $\Delta_j^t \leq w_j^t$. We consider two cases.
Case 1: $\Delta_{j^*}^t < w_{j^*}^t$. This means for some $i \in U_t$ we have $c_i^{t+1} = 0$, implying $i \notin U_{t+1}$. In other terms, H_i becomes covered at time $t+1$. Since the algorithm stops when $U_t = \emptyset$, this case occurs at most $|U^0| = s$ times.
Case 2: $\Delta_{j^*}^t = w_{j^*}^t$. Suppose then that the original problem admits a solution with cost z. Obviously, the "residual" problem where $\mathbf{c}$ is replaced by $\mathbf{c}^t$ admits a solution of cost z, too. This implies the existence of $j \in [\varsigma]$ with $\Delta_j^t \geq \frac{1}{z} c^t$, for otherwise any solution for the residual problem would have cost $> z$. But by the choice of j^* it holds $\Delta_{j^*} = w_{j^*}^t \geq w_j^t \geq \Delta_j^t$ for any j, hence $\Delta_{j^*}^t \geq \frac{1}{z} c^t$. Thus by choosing j^* we get $c^{t+1} \leq (1 - \frac{1}{z}) c^t$. After running into this case ℓ times, the residual cost is then at most $c^0 (1 - \frac{1}{z})^\ell$.

Note that $\ell + s \geq c^0 = s \cdot \bar{c}$ since at any step the overall residual weight can decrease by at most 1. Therefore the algorithm performs $\ell + s = O(\ell)$ steps overall. Furthermore, after $\ell + s$ steps we have $c^{\ell+s} \leq s\bar{c}e^{-\frac{\ell}{z}}$, and by picking $\ell = z\ln(2s)$ we obtain $c^{\ell+s} \leq \frac{\bar{c}}{s}$, and therefore each one of the s graphlets receives weight at least $\frac{\bar{c}}{2}$. Now, if we replace $\bar{c}\mathbf{1}$ with $2\bar{c}\mathbf{1}$ in the original problem, the cost of the optimal solution is at most $2z$, and in $O(z \ln(s))$ steps the algorithm finds a cover where each graphlet has weight at least $\bar{c}$. □

Now, note that the treelet index j^* given by Equation 11 remains unchanged as long as U_t remains unchanged. Therefore we need to recompute j^* only when some new graphlet exits U_t, i.e., becomes covered. In addition, we do not need each value a_{ji}, but only their sum $\sum_{i \in U_t} a_{ji}$. This is precisely the quantity that AGS estimates at line 14. Theorem 6 follows immediately as a corollary.

D. REFERENCES

[1] A. F. Abdelzaher, A. F. Al-Musawi, P. Ghosh, M. L. Mayo, and E. J. Perkins. Transcriptional network growing models using motif-based preferential attachment. *Frontiers in Bioengineering and Biotechnology*, 3:157, 2015.

[2] N. K. Ahmed, J. Neville, R. A. Rossi, and N. Duffield. Efficient graphlet counting for large networks. In *Proc. of ICDM*, pages 1–10, 2015.

[3] N. Alon, O. Gurel-Gurevich, and E. Lubetzky. Choice-memory tradeoff in allocations. *The Annals of Applied Probability*, 20(4):1470–1511, 2010.

[4] N. Alon, R. Yuster, and U. Zwick. Color-coding. *J. ACM*, 42(4):844–856, 1995.

[5] T. Baroudi, R. Seghir, and V. Loechner. Optimization of triangular and banded matrix operations using 2d-packed layouts. *ACM TACO*, 14(4):55:1–55:19, 2017.

[6] M. A. Bhuiyan, M. Rahman, M. Rahman, and M. Al Hasan. Guise: Uniform sampling of graphlets for large graph analysis. In *Proc. of ICDM*, pages 91–100, 2012.

[7] M. Bressan. Counting subgraphs via DAG tree decompositions. In *Proc. of IPEC*, 2019. (To appear).

[8] M. Bressan, F. Chierichetti, R. Kumar, S. Leucci, and A. Panconesi. Counting graphlets: Space vs time. In *Proc. of ACM WSDM*, pages 557–566, 2017.

[9] M. Bressan, F. Chierichetti, R. Kumar, S. Leucci, and A. Panconesi. Motif counting beyond five nodes. *ACM TKDD*, 12(4), 2018.

[10] V. T. Chakaravarthy, M. Kapralov, P. Murali, F. Petrini, X. Que, Y. Sabharwal, and B. Schieber. Subgraph counting: Color coding beyond trees. In *Proc. of IEEE IPDPS*, pages 2–11, 2016.

[11] J. Chen, X. Huang, I. A. Kanj, and G. Xia. Strong computational lower bounds via parameterized complexity. *Journal of Computer and System Sciences*, 72(8):1346–1367, 2006.

[12] X. Chen, Y. Li, P. Wang, and J. C. S. Lui. A general framework for estimating graphlet statistics via random walk. *PVLDB*, 10(3):253–264, 2016.

[13] D. Dubhashi and A. Panconesi. *Concentration of Measure for the Analysis of Randomized Algorithms.* Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[14] D. A. Freedman. On tail probabilities for martingales. *The Annals of Probability*, 3(1):100–118, 1975.

[15] G. Han and H. Sethu. Waddling random walk: Fast and accurate mining of motif statistics in large graphs. *Proc. of ICDM*, pages 181–190, 2016.

[16] S. Jain and C. Seshadhri. A fast and provable method for estimating clique counts using Turán's theorem. In *Proc. of WWW*, pages 441–449, 2017.

[17] M. Jha, C. Seshadhri, and A. Pinar. Path sampling: A fast and provable method for estimating 4-vertex subgraph counts. In *Proc. of WWW*, pages 495–505, 2015.

[18] B. D. McKay and A. Piperno. Practical graph isomorphism, {II}. *Journal of Symbolic Computation*, 60(0):94–112, 2014.

[19] R. Otter. The number of trees. *Annals of Mathematics*, pages 583–599, 1948.

[20] A. Pinar, C. Seshadhri, and V. Vishal. ESCAPE: Efficiently counting all 5-vertex subgraphs. In *Proc. of WWW*, pages 1431–1440, 2017.

[21] G. M. Slota and K. Madduri. Fast approximate subgraph counting and enumeration. In *Proc. of ICPP*, pages 210–219, 2013.

[22] L. D. Stefani, A. Epasto, M. Riondato, and E. Upfal. TriÈst: Counting local and global triangles in fully dynamic streams with fixed memory size. *ACM Trans. Knowl. Discov. Data*, 11(4):43:1–43:50, June 2017.

[23] N. H. Tran, K. P. Choi, and L. Zhang. Counting motifs in the human interactome. *Nature Communications*, 4(2241), 2013.

[24] M. D. Vose. A linear algorithm for generating random numbers with a given distribution. *IEEE Trans. Software Eng.*, 17(9):972–975, 1991.

[25] P. Wang, J. C. S. Lui, B. Ribeiro, D. Towsley, J. Zhao, and X. Guan. Efficiently estimating motif statistics of large networks. *ACM TKDD*, 9(2):8:1–8:27, 2014.

[26] P. Wang, J. Tao, J. Zhao, and X. Guan. Moss: A scalable tool for efficiently sampling and counting 4- and 5-node graphlets. *CoRR*, abs/1509.08089, 2015.

[27] P. Wang, X. Zhang, Z. Li, J. Cheng, J. C. S. Lui, D. Towsley, J. Zhao, J. Tao, and X. Guan. A fast sampling method of exploring graphlet degrees of large directed and undirected graphs. *CoRR*, abs/1604.08691, 2016.

[28] Ö. N. Yaveroğlu, N. Malod-Dognin, D. Davis, Z. Levnajic, V. Janjic, R. Karapandza, A. Stojmirovic, and N. Pržulj. Revealing the hidden language of complex networks. *Scientific Reports*, 4:4547 EP –, 04 2014.

[29] H. Yin, A. R. Benson, J. Leskovec, and D. F. Gleich. Local higher-order graph clustering. In *Proc. of ACM KDD*, pages 555–564, New York, NY, USA, 2017. ACM.

[30] 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 *Proc. of ICPP*, pages 594–603, 2010.