Faster motif counting via succinct color coding and adaptive sampling

Marco Bressan
Sapienza Università di Roma
bressan@di.uniroma1.it

Stefano Leucci
Università dell'Aquila
stefano.leucci@univaq.it

Alessandro Panconesi
Sapienza Università di Roma
ale@di.uniroma1.it

Abstract

We address the problem of computing the distribution of induced connected subgraphs, aka *graphlets* or *motifs*, in large graphs. The current state-of-the-art algorithms estimate the motif counts via uniform sampling, by leveraging the color coding technique by Alon, Yuster and Zwick. In this work we extend the applicability of this approach, by introducing a set of algorithmic optimizations and techniques that reduce the running time and space usage of color coding and improve the accuracy of the counts. To this end, we first show how to optimize color coding to efficiently build a compact table of a representative subsample of all graphlets in the input graph. For 8-node motifs, we can build such a table in one hour for a graph with 65M nodes and 1.8B edges, which is 2000 times larger than the state of the art. We then introduce a novel adaptive sampling scheme that breaks the "additive error barrier" of uniform sampling, guaranteeing multiplicative approximations instead of just additive ones. This allows us to count not only the most frequent motifs, but also extremely rare ones. For instance, on one graph we accurately count nearly 10.000 distinct 8-node motifs whose relative frequency is so small that uniform sampling would literally take centuries to find them. Our results show that color coding is still the most promising approach to scalable motif counting.

1 Introduction

Counting the number of copies of a given pattern graph in a host graph is one of the basic graph mining primitives, with applications in network analysis [1], graph classification [36], graph clustering [38], and biology [37]. Of particular interest are the subgraphs that are induced and connected, commonly known as *graphlets* or *motifs*. Indeed, motifs are often seen as "high-order edges" that are the true building blocks of real-world networks and give fundamental insights into the nature of a graph [22, 31, 36, 38]. The problem of counting the number of copies of a given motif in a graph has a long and rich history, which started with triangle counting and evolved towards larger and more complex motifs [2, 7, 11, 14, 18, 22, 27, 29, 33, 34, 35, 39]. Entire frameworks have been designed to make motif mining easy, including systems based on graph databases such as Arabesque [30] and GraphSig [28], or standalone systems such as Fractal [15] and AutoMine [23].

Unfortunately, motif counting becomes quickly intractable with the size k of the motif. For this reason exact counting is practically feasible only for $k \leq 5$, save for special cases such as counting cliques in sparse graphs. This hardness is not surprising, since the problem is widely believed to require time $n^{\Omega(k)}$ [12] where n is the number of nodes in the input graph. The natural approach to overcome this barrier is to abandon exact counting in favor of approximate counting. Approximate counting can replace exact counting in many cases, such as in hypothesis testing (deciding if a graph comes from a certain distribution or not) or in estimating the clustering coefficient of a graph (the fraction of triangles among 3-node motifs). In this work we focus on approximate motif counting, with a special attention on guarantees. More precisely, we aim at estimating, as accurately as possible, the number of occurrences of *every* possible distinct motif on k nodes (the star, the clique, the path, etc.) in a graph. Formally, suppose we are given a simple graph G (e.g., a social network), an integer $k > 2$, and two approximation parameters $\epsilon, \delta \in (0,1)$. For each motif H on k nodes (the star, the clique, the path, etc.), we want an estimate of the number of induced copies of H in G, so that with probability at least $1-\delta$ all estimates are within a factor $(1 \pm \epsilon)$ of the actual values. Our goal is to develop practical algorithms that solve this problem for G and k significantly larger than the state of the art. This means we aim at graphs G with billions of edges and motifs on more than 5 nodes. Note that we are looking at *induced* copies; counting non-induced copies can be significantly easier (think of the stars). We also remark that, following all previous literature, graphlets are defined as connected.

The most natural approach to motif counting is combinatorial counting. Unfortunately, this approach requires to enumerate and/or count a number of subgraphs that can grow as $n^{\Omega(k)}$, and therefore does not scale to large G and k. Indeed, even state-of-the-art exact counting algorithms such as [27] or [23] are reported to work only for $k \leq 5$. We note that combinatorial explosion affects also approximation algorithms as long as one wants to estimate the counts of *all* motifs at once, as we do. Consider indeed N_k, the number of distinct graphlets on k nodes (that is, the number of non-isomorphic connected simple k-node graphs). Obviously, estimating all k-graphlet counts takes time N_k since we might need to output one count for each graphlet. One can show that N_k grows extremely fast, as $N_k = \exp(\Omega(k^2))$; for example, $N_{20} > 10^{30}$, so already for $k = 20$ the task is hopeless in practice.[1] However, $N_8 \simeq 11\,000$ and $N_{10} \simeq 12\,000\,000$, so for these values of k the task could still be feasible even on large graphs such as real-world social networks.

With combinatorial algorithms ruled out, the most appealing approach left is sampling. The idea is just to sample graphlet copies uniformly at random from G and estimate their frequencies and counts consequently. The difficulty lies in implementing the graphlet sampling primitive in an efficient way, which is trickier than it may appear. The first general graphlet sampling technique was based on Markov chains and was introduced in [7]. The approach is elegant and works in principle for every G and k, but in practice it is efficient only for $k = 4, 5$ on medium-sized graphs. It was later proved that this approach needs $n^{\Omega(k)}$ steps in the worst case to produce just a single unbiased graphlet sample [8], making it comparable to brute-force enumeration. On the other hand, [8] showed one can efficiently sample subgraphs using the color coding technique of Alon, Yuster and Zwick [5]. The idea of color coding is to assign to each node of G one of k colors independently and uniformly at random. Afterwards, via dynamic programming, in time $O(E_G)$ one can count the number of subtrees of G that span k distinct colors (we say they are *colorful*), which gives an estimate of the actual number of subtrees in G. While [5] used color coding for counting trees, [8] showed how to use it to implement graphlet sampling. This sampling framework has two components. The first component is an enriched version of the aforementioned dynamic programming, which builds an abstract "urn" containing a representative sub-population of all the trees of G on up to k nodes (henceforth "k-treelets"). The second component is a recursive algorithm that uses the urn to sample a random k-treelet

[1]See sequence A001349 in the OEIS, `https://oeis.org/A001349`.

from G and, thus, obtain a random connected subset of k nodes (that is, a motif). 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-treelets from the urn. The build-up phase takes 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. This algorithm, named CC by the authors (after *c*olor *c*oding), can reliably and accurately count motifs on $k > 5$ nodes on medium-large graphs and is the current state of the art in motif counting [9].

While CC was the first algorithm consistently able to count motifs on more than 5 nodes on large graphs, the hardness of the problem still imposed some limitations on it. First, the build-up phase of CC is resource-demanding, especially concerning memory (recall that the build-up phase takes time and space growing exponentially in k). This limits its scalability; for example, even using a machine with 72GB of main memory, CC runs out of resources when estimating 7-graphlet counts on graphs with more than 2M nodes, as shown in our experiments. Second, since CC samples graphlets uniformly at random, its approximation guarantees are only of additive type. That is, using s samples, CC can only detect graphlets whose relative frequency is at least $1/s$ — all other graphlets will be undetected or heavily misestimated. Unfortunately, in many graphs nearly all graphlets have extremely low frequencies, so CC would need an unbearable number of samples (this is true not only for CC but for any algorithm based on uniform sampling). In this work we overcome these two bottlenecks, pushing the color-coding approach in the realm of massive graphs.

1.1 Our results

We present two motif counting algorithms. The first is GM (for *G*eneral-purpose *M*otif counter), which is designed to count motifs on $k \leq 16$ nodes. The second is L8Motif (pronounced "leitmotif", for *L*arge-graph 8-node *M*otif counter), which is optimized for motifs on $k \leq 8$ nodes. These optimizations allow us to scale to graphs larger than GM and significantly improve on the state of the art. Thanks to these algorithms we can count motifs on up to 8 nodes on graphs with billions of edges with excellent accuracy, using just ordinary hardware. To convey the idea, in one hour we can accurately count 8-node motifs in a graph with 65M nodes and 1.8B edges (Friendster); this is 200 and 2000 times larger, in terms of n and m, than the prior art.[2] Unlike all previous algorithms, GM and L8Motif compute accurate counts for nearly *all* graphlets at once, even extremely rare ones. Consider for instance the Yelp graph (one of our datasets). On this graph, the state-of-the-art algorithm CC finds only the top two most frequent 8-graphlets and literally misses all the others. In the same amount of time, L8Motif produces counts within a multiplicative error of $\epsilon \leq 0.25$ for $\simeq 10.000$ distinct graphlets simultaneously. Many of these graphlets have frequency $\leq 10^{-21}$, and many previous algorithms based on uniform sampling, like CC or random walks, would need 10^3 years to find them even by sampling 10^6 graphlets/second. Figure 1 gives a pictorial summary of our results.

Technical contributions. Our algorithms GM and L8Motif rely on two technical contributions of different nature. The first contribution is a set of efficient algorithms and data structures for the dynamic programming of the build-up phase, which reduce the running time and especially the space usage (one of the main bottlenecks of CC). In particular, for L8Motif we design data structures and algorithms that use an optimal amount of bits in an information-theoretical sense, by adopting a compact integer representation of rooted colored treelets, and a variable-length encoding of the treelet counts. In addition, we show one can entirely skip the heaviest round of the dynamic program via a balanced treelet decomposition trick, saving additional time and space. Thanks to these ingredients, although theoretically we inherit the $O(a^k m)$

[2] All measurements are obtained on a workstation with 36 cores and 72GB of main memory — see Section 5.

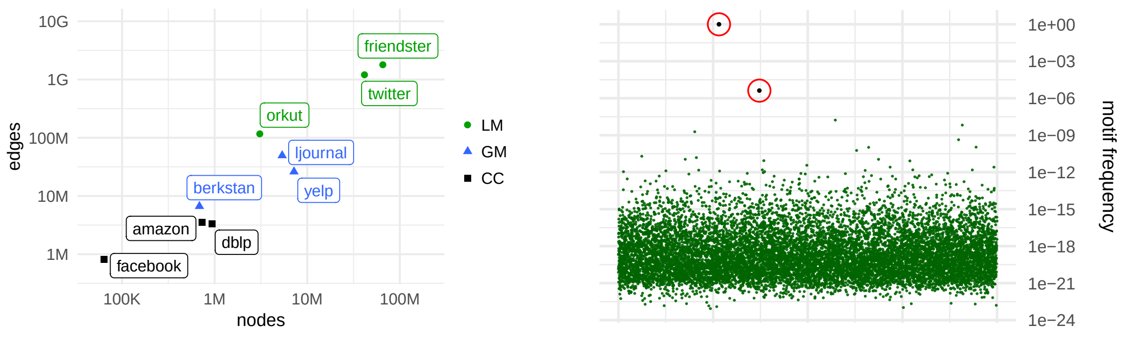


Figure 1: Our performance in a picture for motifs on $k = 8$ nodes. Left: the size of the graphs managed by the state-of-the-art CC and by our algorithms GM and L8Motif (abbreviated as LM). Right: the dense green band has one point for each distinct graphlet that we count accurately on the Yelp graph ($\simeq$ 10.000 in total) while the two red circles show the only two graphlets detected by CC.

running time and $O(c^k n)$ space usage of CC, we can scale the build-up phase from millions to billions of edges and from $k = 5$ to $k = 8$. Our other algorithm, GM, can be used to handle graphlets on up to $k = 16$ nodes (with some loss of efficiency compared to L8Motif).

The second technical contribution, common to both GM and L8Motif, is for the sampling phase and is of a fundamentally different nature. To convey the idea, imagine having an urn with 1000 balls — 990 red, 9 green, and 1 blue. With uniform sampling, we will quickly obtain a good estimate of the fraction of red balls, but we will need many samples to observe green or blue balls. This is the uniform sampling barrier mentioned above. Now imagine to remove ∼99% of all red balls from the urn. We would be left with 10 red, 9 green, and 1 blue ball, and we would then quickly get a good estimate of the fraction of green balls. Then imagine to delete ∼99%of the red and green balls; after this, we could quickly estimate the fraction of blue balls. We show that our treelet database supports a similar "deletion trick": we can ignore some treelets (say, stars) and focus on other ones (say, paths). In this way we can estimate the most frequent graphlet, delete it and proceed to the second most frequent one, and so on. We name this algorithm Adaptive Graphlet Sampling, or AGS. Technically speaking, AGS is based on an online greedy algorithm for a fractional set cover problem (we want to "cover" all graphlets with their spanning treelets). We provide theoretical guarantees on the accuracy and efficiency of AGS via competitive analysis and martingale concentration bounds. We note that AGS is the first algorithm to ensure accuracy for all graphlets at once.

Remark. We first described GM in [10], a preliminary version of this work. The main contribution of this extended version is L8Motif and all the associated technical ideas (integer treelet encoding, round skipping, and variable-length counts). Thanks to these ideas we can reduce the running time and space usage of GM by almost an order of magnitude, which allows us to scale $k = 7$ to $k = 8$ on the largest graphs (we recall that the time and space requirements grow exponentially with k, which makes moving from k to $(k + 1)$ a considerable challenge). To give an idea, to count 6-node motifs on our largest graph Friendster (∼ 2B nodes), GM needs 130 GB of space and 3 hours of time, while L8Motif needs only 25GB and 30 minutes.

Manuscript organization. Subsections 1.2 and 1.3 respectively discuss related work and introduce notation and conventions. Section 2 reviews color coding and CC, the starting points of our work. Section 3 describe the build-up phase of our algorithms, comparing them to CC.

Section 4 describes our adaptive graphlet sampling technique and other graphlet sampling optimizations. Finally, Section 5 provides experimental results on a set of publicly available graphs.

1.2 Related work

The traditional approach to subgraph counting is based on combinatorial counting. These techniques work fine for $k = 4$ and 5, but become unusable for $k > 5$, even for approximate counting. This limitation affects many subgraph mining tools such as Arabesque [30], ESCAPE [27], Fractal [15], Pangolin [13] and AutoMine [23]. Indeed, none of them is reported to scale beyond $k = 5$. The only algorithms that potentially scale to $k > 5$ are, as anticipated, based on random walks or on color coding.

The algorithms based on random walks are based on the following idea. We define a graph $\mathcal{G}_k$ whose nodes are the graphlets of G and where two graphlets are adjacent if they share $k-1$ nodes. Then, the lazy random walk on $\mathcal{G}_k$ can be shown to be ergodic and converge towards a unique limit distribution on the set of all graphlets of G. Moreover, each step of the walk can be efficiently simulated using only local information from G. Thus one just simulates enough steps, draw a graphlet from the limit distribution, compute an unbiased estimator its frequency [7, 33, 14, 18]. The drawback of this technique is its inefficiency: even if G is fast-mixing, the random walk on $\mathcal{G}_k$ may need $\Omega(n^{k-1})$ steps to reach the limit distribution, or even just to find the most frequent graphlet of G [8, 9]. This is close to $O(n^k)$, the running time of the naive exhaustive enumeration algorithm. One can mitigate this mixing time explosion by walking on subgraphs on less than k nodes [14] or by sampling a spanning tree directly in G [26]. Unfortunately, this gives biased samples that needs to be reweighted, increasing the estimator variance so that $\Omega(n^{k-1})$ samples might again be necessary. Note also that most of these algorithms can estimate only the relative *frequencies* of graphlets, but not their counts.

There are several algorithms based on color coding. Most of them can only count special motifs, such as trees and subgraphs of small treewidth, often only in their non-induced copies [19, 3, 39, 40, 29, 17, 11], and therefore cannot be applied to our setting. The only algorithm that can count all induced motifs on k nodes without any further assumption, which is the goal of our work, is the CC algorithm of [8, 9]. The only two limitations of CC are that (1) for $k = 8$, it can only manage graphs on less than 0.5M nodes, and (2) it gives accurate counts only for the few most frequent graphlets. In contrast, thanks to our optimization of the build-up phase and the introduction of adaptive sampling, we manage graphs with tens of millions of nodes and billions of edges while giving guarantees for nearly all k-graphlets at once. Finally, we note that specialized algorithms such as [23] or [20] can efficiently compute or estimate the number of cliques on 8 or 10 nodes, on graphs with tens of millions of edges. Again, these algorithms work only for a particular motif (the clique) while we can count all motifs at once.

1.3 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 $H = (V_H, E_H)$. An *occurrence* or *copy* of H in G is a subgraph of G isomorphic to H. Unless otherwise specified, a copy of H in G is meant as induced. We let $k = |V_H|$; in this work we are interested in $k \leq 16$ and especially in $k \leq 8$. A *treelet* T is a graphlet that is a tree. When using treelets as spanning trees, their copies in G are meant as not necessarily induced. We denote by $\mathcal{H} = \mathcal{H}_k$ 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 $|T|{=}|C|$. We often consider treelets and colored treelets rooted at a node $r \in T$ (different rootings can give different treelets). Finally, by d_v we denote the degree of v in G, and by $u \sim v$ we indicate that u is a neighbor u of v.

2 Color coding and the CC algorithm

Our algorithms, like the CC algorithm of [8, 9], are based on the color coding technique by Alon, Yuster and Zwick [5], which works as follows. First, we assign to each node $v \in G$ a color chosen uniformly and independently in $\{0, \dots, k-1\}$. Now consider any k-node treelet in G: the random coloring makes it colorful with probability $p_k = \frac{k!}{k^k} \approx e^{-k}$. Therefore a constant fraction of all treelets of G will become colorful. The crucial observation of color coding is that the number of colorful k-treelets can be counted in time $O(m \cdot a^k)$, with a bottom-up dynamic program. Note that the running time is exponential in k, but linear in m.

While color coding was introduced to detect and count noninduced trees, the authors of [8, 9] showed how to extend it to sampling graphlets from G. The idea is to run a modified dynamic program that collects information about the "colorful structure" of G. Once this is done, it is easy to sample colorful k-treelets from G and, so, to sample colorful graphlets (just take the graphlet spanned by the treelet). This is the essence of the CC algorithm [8, 9]. We now detail the two phases of CC, the *build-up phase* and the *sampling phase*, which form our algorithm as well.

2.1 The build-up phase

The goal of this phase is to build a *count table* holding the counts of colorful treelets of G. The phase starts by coloring G: for each node v, we draw a color c_v uniformly at random in $[k]$. Now, for every $v \in G$ and every rooted colored treelet T_C on up to k nodes, we want to compute the following quantity:

$$c(T_C, v) = \text{the number of copies of } T_C \text{ in } G \text{ that are rooted in } v \tag{1}$$

We compute $c(T_C, v)$ by dynamic programming. For each v we initialize $c(T_C, v) = 1$, where T is the trivial treelet on 1 node and $C = \{c_v\}$; all other counts are implicitly 0. Now suppose we have computed the counts of all treelets on $h-1$ nodes for some $h \leq k$. To compute $c(T_C, v)$ for some $T_C = (T, C)$ on h nodes, we decompose T in two smaller subtrees T' and T'', rooted respectively at the root r of T and at a child of r, and combine their counts. It is easy to see that $c(T_C, v)$ is given by (see [9]):

$$c(T_C, v) = \frac{1}{\beta_T} \sum_{u \sim v} \sum_{\substack{C' \subset C \\ |C'|=|T'|}} c(T'_{C'}, v) \cdot c(T''_{C''}, u) \tag{2}$$

where β_T is the number of subtrees of T isomorphic to T'' rooted in a child of r. In practice, one uses a *canonical decomposition* which defines the pair (T', T'') uniquely as a function of T. A simple analysis of the entire dynamic program shows that:

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

In practice, the table size grows quickly. For $k = 6$ on a graph with 5M nodes, CC already needs 50GB of main memory [9].

2.2 The sampling phase

The goal of this phase is to estimate the graphlet counts by sampling graphlets from G. We do this by sampling colorful treelet copies from the treelet count table, as follows. First, draw a pair (T_C, v) with probability proportional to $c(T_C, v)$. This is possible since we know all the counts $c(T_C, v)$. Now we want to sample a copy of $T_C = (T, C)$ rooted at v. To this end we take again the canonical decomposition of T into T' and T''. We then sample a pair (v, C''), where $v \sim u$ and $C'' \subset C$ contains $|T''|$ colors, with probability proportional to $\beta_T^{-1} c((T', C \setminus C'')) \cdot c((T'', C''), v)$. We then recursively sample a copy of $T'_{C'} = (T', C \backslash C'')$ rooted at v, and a copy of $T''_{C''} = (T'', C'')$ rooted at v. Once we have the copies of $T'_{C'}$ and $T''_{C''}$, we just combine them into a copy of T_C. One can verify that the resulting copy is drawn uniformly at random from the set of all colorful treelets of G [9].

Using this k-treelet sampling primitive, one can estimate the copies of any given k-graphlet H_i (e.g., the clique). First, we estimate the number c_i of *colorful* copies of H_i. To this end we sample a treelet copy as described above; being connected, the treelet necessarily spans some induced k-node subgraph x of G. Let χ_i be the indicator random variable of the event that x is a copy 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. Now, t is known from the treelet count table, and σ_i can be computed e.g. via Kirchhoff's theorem (see below). Therefore we can compute $\hat{c}_i = t\,\sigma_i^{-1} \chi_i$, which is an unbiased estimator of c_i. By standard concentration bounds, $\hat{c}_i$ concentrates around c_i if enough samples are taken.

Finally, to estimate the total number g_i of copies of H_i, we simply divide $\hat{c}_i$ by the probability $p_k = k!/k^k$ that a fixed set of k nodes in G becomes colorful. Indeed, if G contains g_i copies of H_i, then by linearity of expectation the expected number of copies of H_i that become colorful is $\mathbb{E}[c_i] = p_k g_i$. Therefore $\hat{g}_i = \hat{c}_i / p_k$ is an unbiased estimator for g_i.

2.3 Statistical guarantees of the estimates

With the algorithm pinned down, the next crucial point is the accuracy of the graphlet estimates $\hat{g}_i$. Note that there are two distinct sources of error: the coloring, which distorts the true graphlet distribution into the colorful one, and the sampling.

For what concerns the coloring, one can prove that the colorful graphlet distribution is very close to the original one in a statistical sense. First, we report a bound from [9], slightly rephrased. Let $g = \sum_i g_i$ be the total number of induced k-graphlet copies in G. Then:

Theorem 2 ([9] Thm 5.3). *For all $\epsilon > 0$, a random coloring of G with k colors gives:*

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

Since this bound is additive, we complement it by proving a multiplicative one, which is tighter if the maximum degree Δ of G is small. Formally:

Theorem 3. *For all $\epsilon > 0$, a random coloring of G with k colors gives:*

$$\Pr\Big[\Big|\frac{c_i}{p_k} - g_i\Big| > \epsilon\, g_i\Big] < 2\exp\Big(-\frac{2\epsilon^2 p_k^2\, g_i}{(k-1)!\Delta^{k-2}}\Big). \tag{3}$$

Proof. We use a concentration bound for dependent random variables from [16]. 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 $\mathbb{E}[c_i] = p_k g_i$. 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 a 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 [16] by setting $t = \epsilon\mathbb{E}[c_i] = \epsilon p_k g_i$, $(b_\alpha - a_\alpha) = 1$ for all $\alpha = h \in \mathcal{V}_i$, and $\chi^*(\Gamma) \leq \chi_k \leq (k-1)!\Delta^{k-2}$. □

These two bounds suggest that the random coloring does not introduce a significant distortion. This is confirmed by our experiments, where the $\hat{g}_i$ appear concentrated around the mean. Hence, one may avoid averaging over $\Theta(\exp(k))$ independent colorings, as suggested in the original color coding paper [5]; one run is enough. Thus, in a sense, the treelet count table is w.h.p. a database that holds (implicitly) a representative sample of all k-graphlets in G.

For what concerns sampling, standard concentration bounds apply to the error of uniform sampling (see above). For AGS, we show concentration bounds for all graphlets in Section 4.1.

3 Fast construction of a compact treelet database

In this section we describe the internals of the build-up phase of our algorithms GM and L8Motif. Recall that, at a high level, the goal of the build-up phase is to compute a compact database of colorful treelet counts (Section 2). This database to provide the following operations:

- `occ(`v`)`: get the total number of colorful treelet copies rooted at v
- `sample(`v`)`: get a uniform random colored treelet rooted at v

Note indeed that using these two operations we can realize our sampling, by first selecting a node v proportionally to its number of treelets and then sampling one such treelet uniformly at random. For our adapting sampling scheme, we must also be able to specify the treelet shape and coloring:

- `occ(`T_C, v`)`: get the total number of copies of T_C rooted at v
- `sample(`T, v`)`: get a uniform random colored treelet rooted at v with shape T

To give a detailed account of the performance of GM and L8Motif, we describe the algorithms in an incremental way: we start from the state of the art algorithm CC, and we then introduce step by step our algorithmic ingredients, measuring their (positive) impact on the overall performance. Thus, the baseline for GM is a C++ version of CC (which is originally in Java), that we wrote by porting all algorithms and data structures carefully. The baseline for L8Motif is instead GM itself. Before starting, in Subsection 3.1 we recall the main bottlenecks of CC. Finally, in Subsection 3.4 and Subsection 3.5 we describe additional optimizations that apply to both GM and L8Motif.

3.1 Treelets and counts

In this subsection we discuss the crucial aspects of the build-up phase of CC. In a nutshell, the build-up phase spends virtually all its time manipulating rooted (un)colored treelets and reading/writing their counts – see Section 2. Therefore, it is essential to make these objects as efficient as possible.

The first computationally intensive task is merging the counts. Consider a single count $c(T_C, v)$. To compute $c(T_C, v)$, we process all the neighbors u of v as follows. First, define a total order over all the T_C (this order is described below). For every pair of non-zero counts $c(T'_{C'}, v)$

and $c(T''_{C''}, u)$, check that $C' \cap C'' = \emptyset$, and that $T''_{C''}$ comes not later (w.r.t. the total order) than the smallest subtree rooted in a child of the root of $T'_{C'}$. 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 $c(T_C, v)$ is incremented by $\beta_T^{-1} c(T'_{C'}, v) \cdot c(T''_{C''}, u)$; see Equation 2. In this process, the computationally intensive part is the check-and-merge operation, which can be formalized as the primitive:

- `merge(`$T'_{C'}$`,`$T''_{C''}$`)`: if possible, merge $T'_{C'}$, $T''_{C''}$ by appending $T''_{C''}$ to the root of $T'_{C'}$, else FAIL

In CC, this primitive is implemented as a recursive algorithm, which can be rather expensive. Here we use a different implementation that makes it much faster.

The second computationally expensive task is storing and accessing the counts. CC does it as follows: for each node $v \in G$, it keeps a dedicated hash table in main memory which maps each colored treelet $T_C > 0$ to its count $c(T_C, v)$. The hash table key is a pointer to a unique *representative instance* of T_C: a classic pointer-based tree data structure equipped with a satellite set data structure for storing the colors. Each entry of CC's table thus uses 128 bits: 64 for the key, and 64 for the integer count. Already for $k = 6$ on a graph with a few million nodes, storing the hash table can require a dozen GB of main memory, so this approach becomes quickly impractical. Here, we show how to reduce the overall space requirements of the count table by almost an order of magnitude.

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`[3] 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[4], 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, we are always faster than CC.

3.2 GM: a general-purpose motif counter for k $\leq$ 16 nodes

We describe our first toolbox, GM, introduced in our preliminary work [10], for graphlets on up to 16 nodes. Starting from the C++ porting of CC (see above), we improve the performance by introducing succinct treelets and the succinct count table — see the comparison in Figure 2.

3.2.1 Succinct treelets and count table

Treelet representation For the first optimization, we drop CC's recursive pointer-based structures, and switch to a representation of treelets as bitstrings. This representation makes the `merge(`T`,`T'`)` operation much faster, up to 150× for $k = 5$ and up to 1000× for $k = 7$. Given an uncolored treelet T rooted at r, its bitstring s_T is 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. The lexicographic ordering over the 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.

[3]`http://fastutil.di.unimi.it/`

[4]`https://github.com/sparsehash/sparsehash`

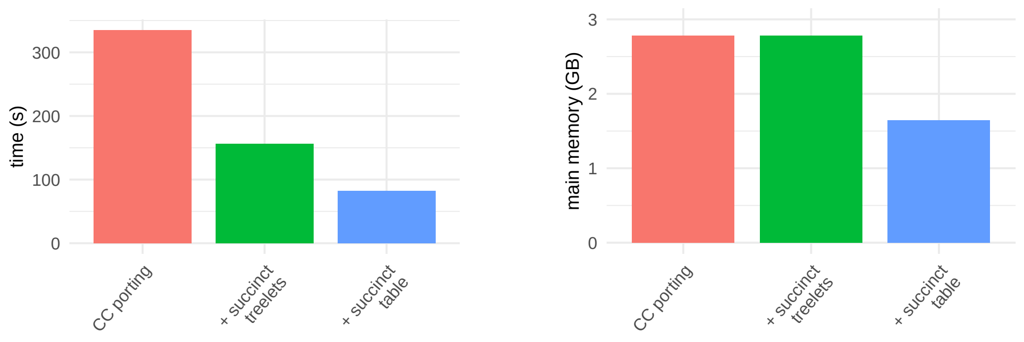


Figure 2: Cumulative impact of our optimizations on the build-up phase of CC, for Amazon with $k = 6$.

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.[5] 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).

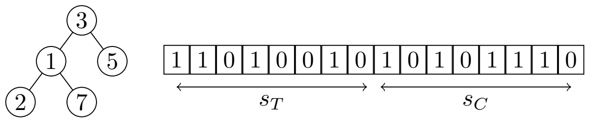


Figure 3: A colored rooted treelet and its encoding, shown for simplicity on just $8 + 8 = 16$ bits.

Count tables Recall that, in CC, treelet counts are stored in n hash tables, one for each node $v \in G$, and that retrieving the actual structure of T_C requires dereferencing a pointer before each check-and-merge operation. Instead of using a hash table, GM 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

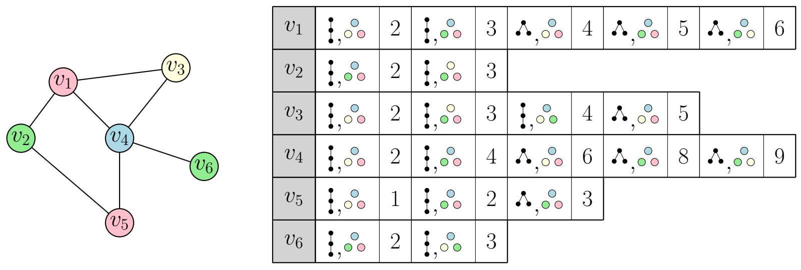


Figure 4: Left: a graph G whose vertices have been colored using $k = 4$ colors. Right: a graphical representation of the count table (implicitly) storing the number $c(T_C, v)$ of all colored treelets T_C of size 3 in G. Notice how we actually store $\eta(T_C, v)$ instead of $c(T_C, v)$.

[5] Given an universe U, the characteristic vector $\langle x_1, x_2, \dots \rangle$ 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.

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 $O(6^k)$.[6] Note that, while CC uses 64-bit counts, which causes overflows (just the number of 6-stars centered in a node of degree 2^{16} is $\approx 10^{22}$), GM uses 128-bit counts, which adds a small overhead[7]. However, we save 16 bits by packing s_{T_C} into 48 bits. In the end, we use 176 bits per each pair $(T_C, c(T_C, v))$. Finally, in place of $c(T_C, v)$, GM 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, the total count η_v for v is at the end of the record, and a random treelet rooted in v can be sampled by binary searching for a random value in $[1, \eta_v]$ over $\eta(\cdot, v)$. See Figure 4 for an example on a small graph.

3.3 L8Motif: counting 8-graphlets in large graphs

In this subsection we describe L8Motif, a version of GM specialized for $k \leq 8$. We start with GM as the default baseline and then plug in three new ingredients, each of which improves the performance – see Figure 5. We also compare against the state of the art (CC) in Section 5.

3.3.1 Integer treelet encoding (ITE)

Our first ingredient is an extremely compact representation of treelets as unsigned integers. The idea is simple: we observe that there are exactly 1991 rooted colored treelets on at most 8 nodes. Therefore, we can encode each treelet T_C as a unique 11-bit integer. This is $4\times$ less than the bistring encoding of Section 3.2.1, which uses 46 bits per treelet. From a theoretical point of view, we are using the optimal amount of bits, up to a multiplicative factor $1 + o(1)$. In this sense, one cannot use fewer bits per treelet, which is exactly what we want.

To ensure treelets are memory-aligned, we pad the 11-bit representation into a 16-bit integer. This leaves 5 spare bits to encode the *length* of the count, that is, the number of bytes used by $c(T_C, v)$ — see Section 3.3.3. In this way we can support counts on up to $2^5 = 32$ bytes i.e. 256 bits, twice the count size of our previous branch. In the end, for each count we use only 16 bits rather than 48 bits. In practice, we observe a deterministic space saving of $\simeq 20\%$ on all instances (see Figure 5 for an example). Regarding time, on large graphs ITE can already make the difference by drastically reducing the time spent in I/O operations, as a consequence of saving space.

While ITE optimizes space, it makes manipulations harder. Indeed, unlike the bitstring representation of Section 3.2.1, which encodes the treelet structure, in ITE the index of a treelet is just a number. Recall for example the `merge`$(T'_{C'}, T''_{C''})$ operation: we must check if $T'_{C'}$ and $T''_{C''}$ can be merged into a treelet T_C whose unique decomposition yields precisely $T'_{C'}$ and $T''_{C''}$. To do this, we *precompute* the results of `merge`$(T'_{C'}, T''_{C''})$ on all inputs (again on up to 8 nodes) and store all such results in a bidimensional array. In this array, entry $[i][j]$ is the index of the treelet resulting from merging i and j (or -1 for FAIL). Another array tells for every treelet the indices i, j associated to the treelets of its canonical decomposition. Other arrays tell us the index of the treelet T and the set of colors C associated to a colored treelet $T_C = (T, C)$, and so on. In this way we can perform every treelet operations quickly, with a sequence of array

[6]By Cayley's formula: there are $O(3^k k^{-3/2})$ rooted treelets on k vertices [25], and 2^k subsets of k colors.

[7]Tests on our machine show that summing 500k unsigned integers is $1.5\times$ slower with 128-bit than with 64-bit integers.

lookups. The total size of all arrays is less than 3 megabytes, which fits in the CPU cache. The time for a single operation is similar to the bitstring encoding of Section 3.2.1.

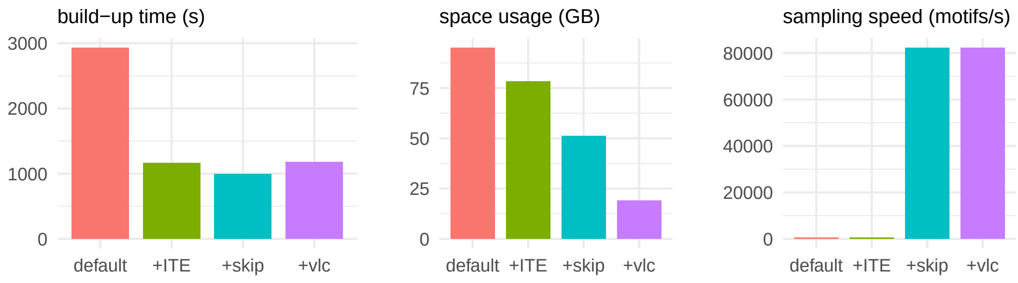


Figure 5: Twitter graph, $k = 6$. Cumulative impact of using ITE, skipping the heaviest round of the dynamic program, and using variable-length counts.

3.3.2 Round skipping via balanced treelet decomposition

Our second ingredient is somewhat trickier. Recall again that, in the build-up phase, we repeatedly merge the counts of smaller treelets. In particular, in round $k-1$ we produce the counts for all treelets on $k-1$ nodes. For $k \leq 8$, round $k-1$ is consistently the most expensive one, and consumes roughly 40% of the time and space of the entire phase — see Figure 7. It would be useful to cut down the time and space of this round.

Our observation is the following: *a $(k-1)$-treelet count is used only to compute the counts of k-treelets whose smallest subtree in the canonical decomposition is a single node.* For example, a k-star is certainly decomposed into a single node and a treelet on $k-1$ nodes and therefore to count k-stars we need the $(k-1)$-treelet counts. It turns out that stars are the only treelets whose decomposition necessarily contains a treelet on a single node. However, we can avoid counting stars, as it is superfluous — at sampling time, stars can be sampled very efficiently in the naive way.

Let us now describe our round skipping technique in more detail. We rely on the following simple fact:

Lemma 1. *Any k-treelet T that is not a star contains an edge uv that, when cut, produces two trees on at most $k-2$ nodes each.*

Proof. If T is not a star then it contains a path on 3 edges; let it be $xuvy$. Cutting uv yields two trees each one having at least 2 nodes or, equivalently, at most $(k-2)$ nodes. ☐

Therefore, for any non-star k-treelet T, we can find a root u and a child v of u that give a *balanced* decomposition of T. That is, a decomposition of T into T' and T'' such that both T' and T'' have size at most $\leq k-2$, but none of them has size $k-1$; see Figure 6. We therefore replace the canonical decomposition of Subsection 3.2.1 with this balanced decomposition, and we exclude k-stars from the set of treelets that we count in the k-th round. Then, we can completely skip round $(k-1)$ of the dynamic program. This yields a reduction in both space and time, consistently across all instances, of up to 40% (Figure 7).

This round skipping has a positive side-effect at sampling time, too. Since we do not have counts of colorful k-stars anymore, we cannot sample k-stars from the database. However, we can simply draw a root node v from G with probability proportional to $\binom{d_v}{k-1}$ and then choose $k-1$ neighbors of v u.a.r. without replacement. This gives an *uncolored* k-star u.a.r. from G,

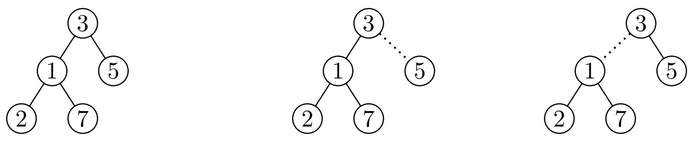


Figure 6: A rooted colored 5-treelet (left) with its original decomposition in two subtrees on 4 and 1 nodes (middle) and its balanced decomposition in two subtrees on 3 and 2 nodes (right).

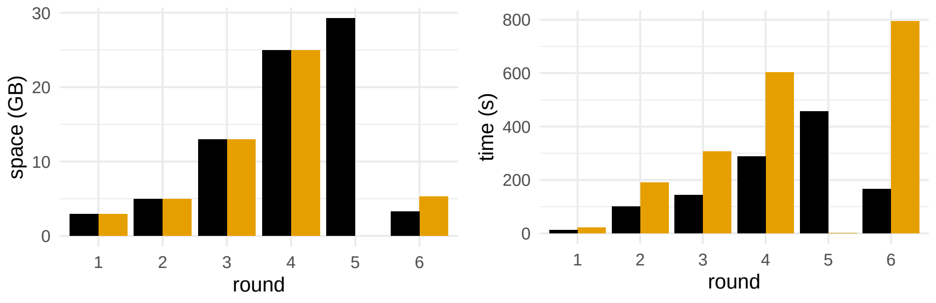


Figure 7: Build-up phase on Twitter for $k = 6$: space and time usage of single rounds, before and after adopting balanced treelet decompositions and round skipping.

which for our purposes is obviously even better since we avoid the error introduced by coloring. Sampling in this fashion is much faster than using the database; and since often most treelets of G are stars, this boosts the sampling rate of uniform sampling by orders of magnitude (Figure 5).

3.3.3 Variable-length counts

Our third and final ingredient is again aimed at saving space by encoding efficiently the *counts* stored in the database. Specifically, we encode each treelet count $c(T_C, v)$ as a variable-length count. In practice, when we need to store $c(T_C, v)$ on disk, we allocate $b = \lceil \log_2(c(T_C, v))/8 \rceil$ bytes for $c(T_C, v)$ and we encode b in the padding of the encoding of T_C (Subsection 3.3.1). The rationale is that, heuristically, most of the counts that we compute fit in much less than 128 bits, see Figure 8. Summarizing, we encode each treelet count record as shown in Figure 9, that is:

- 11 bits for T_C,
- 5 bits for the length of $c(T_C, v)$ in bytes, $\ell \in \{1, \ldots, 32\}$
- ℓ bytes for $c(T_C, v)$

We remark that we are using an amount of space that is almost optimal, that is, we are only a factor $1 + o(1)$ away from the optimum. In this sense, no color coding algorithm can outperform ours if not by a very small margin.

With variable-length counts, we observe an additional space reduction of $\geq 60\%$ on all graphs for all $k \geq 6$. Note that with variable-length counts we cannot find counts via binary search, as the counts are not aligned in memory anymore. Moreover, we pay the obvious overhead of encoding and decoding the counts. In the worst case we witness an increase in the building time by 50%, and only on small instances. On our largest graphs (Twitter and Friendster), variable-length counts are crucial to reduce the space footprint enough to manage graphlets on $k = 8$ nodes.

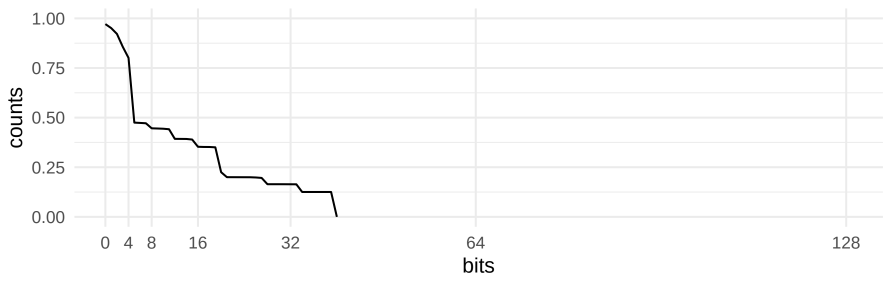


Figure 8: The fraction of treelet counts that would fit in at most b bits, as a function of b, for the treelet table of TWITTER with $k = 6$. The resulting average bit length is just 14, almost 90% less than 128 bits.

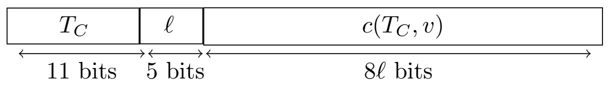


Figure 9: Variable-length encoding of a treelet count.

3.4 Lower-level optimizations and architectural details

For completeness and reproducibility, we provide some additional optimizations and features of GM, some with a significant impact.

3.4.1 Zero-rooting

This optimization applies only to GM ($k \leq 16$). Consider a colorful treelet copy in G formed by the nodes $v_1, \ldots, v_h$. This treelet appears in the records of $v_1, \ldots, v_h$, since it counts as a rooted treelet for each of them. Therefore, the treelet is counted h times. This redundancy is necessary when $h < k$, since we need all rootings for the next round of the dynamic program, see (2). However, for $h = k$ this is useless. Thus, we store k-treelet counts only at nodes of color 0. This cuts the running time by $30\% - 40\%$, while reducing the size of the k-treelets records by a factor of k, and the total space usage by $\approx 10\%$. Notice that the second branch ($k \leq 8$) already counts each k-treelet only once, due to the balanced treelet decomposition described above (but these treelets are not necessarily stored at nodes of color 0).

3.4.2 Greedy flushing

To reduce the main memory footprint, we use a greedy flushing strategy. Suppose we are building the count table of the h-treelets. We temporarily store the record of v in a hash table, which allows for efficient insertions and lookups; when the record is complete, we immediately flush it on disk and release the hash table memory. In this way we end up with the set of records for all nodes of G. When all records have been flushed, a second I/O pass sorts them w.r.t. their node in G, so that we can access them efficiently in the next round. In all our runs, the sorting took at most 10% of the total time.

3.4.3 Multi-threading

We make heavy use of thread-level parallelism in both the build-up and sampling phases. For the build-up phase, for any 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. Obviously, when the number of remaining vertices drops below the number of available threads, some threads become idle. When this happens, we partition the edges of a single vertex v across different threads and make them compute different summands of the outermost sum of Equation (2). The partial sums are then summed together into $c(\cdot, v)$. For the sampling phase, samples are by definition independent and are taken by different threads.

3.4.4 Memory-mapped reads

Recall that the treelet count database is stored in external memory. This entails I/O access, since computing the count table for treelets of size h requires accessing the count tables of each size $j < h$. We delegate the task to the operating system by using memory-mapped I/O. This means that we see all tables as if they resided in main memory, and the operating system takes care of loading and storing them to disk. With enough memory this gives virtually no overhead; otherwise, the OS 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 actual overhead in terms of additional I/O turns out to be at most 100MB, except for $k = 8$ on LiveJournal (34GB) and Yelp (8GB) and for $k = 6$ on Friendster (15GB). Tn those cases the overhead is inevitable since the total size of the tables is close to or even larger than the main memory available.

3.5 Biased coloring

Finally, we describe a simple trick that reduces space significantly (in exchange for accuracy) and can thus be useful to manage very large graphs. The idea is to skew the distribution of colors in the coloring phase, so that fewer treelets become colorful and we have to process and store less counts.

Suppose then we have probability $\lambda \ll \frac{1}{k}$ to pick color $i \in \{1, \ldots, k-1\}$, and probability $1-\lambda(k-1)$ to pick color 0. The probability that a given j-treelet copy is colored with C becomes:

$$p_{k,j}(C) = \begin{cases} j!\lambda^j & \text{if } k \notin C \\ \sim j!\lambda^{j-1} & \text{if } k \in C \end{cases} \tag{4}$$

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. We can thus trade a $\Theta(1)$ factor in the exponent of the bound for a $\Theta(1)$ factor in both time and space. On large graphs this makes a difference, and indeed it allows us to manage our largest instances (see Section 5). Note that one could find a good value for λ by setting $\lambda \ll 1/kn$ and then growing λ until a small but non-negligible fraction of counts are positive. At this point Theorem 3 ensures concentration, and we proceed to the sampling phase. In our experiments, we use biased coloring for Twitter and Friendster for $k = 8$, setting $\lambda = 0.001$.

4 Sampling treelets from the database

This section describes in detail the algorithms for sampling graphlets from the treelet count table. Recall that we support two sampling algorithms: uniform sampling, which is the native sampling algorithm of CC, and our novel adaptive graphlet sampling strategy (AGS). Uniform sampling is exactly the one described in Section 2. Hence we directly move on to AGS in the next section; we then conclude by describing lower-level optimizations that apply to both sampling algorithms and in many cases increment the sampling rate substantially.

4.1 Adaptive Graphlet Sampling (AGS)

This section describes AGS, our adaptive graphlet sampling algorithm. Recall that the main idea of CC is to build a compact database for sampling k-treelets from G. In particular, we can choose the kind of treelet to sample (a star, a path, etc.). That is, for every k-treelet T our database supports the following primitive:

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

With this primitive, we can selectively sample any desired treelet shape, and virtually "delete" graphlets from the urn.

Let us explain the idea with an example. Suppose G contains just two types of colorful graphlets, H_1 and H_2, of which H_2 represents a tiny fraction, say 10^{-10}. With uniform sampling, we will need approximately 10^{10} samples 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 can then start calling sample(T_1), until we estimate accurately H_1. At this point we call sample(T_2), thus ignoring H_1 completely (since it is not spanned T_2), until we estimate accurately H_2, too. Using sample(T) we can thus estimate both graphlets with $O(1)$ samples. Clearly, things are in general more complex, since we have thousands of graphlets, many of them with common spanning trees. Still, the idea works strikingly well.

Let us describe AGS in more detail. We start by invoking sample(T) on the most frequent k-treelet T in G (which we know from the database). Eventually, some graphlet H_i spanned by T will appear enough times, say $\Theta(\epsilon^{-2}\ln(1/\delta))$, so that we can estimate its occurrences accurately. We then say H_i is *covered*. Now we do not need any additional sample of H_i, so we would like to "delete" it. That is, we want to sample using a T' that does *not* span H_i. Such a T' may not exist, but we can use the T' that *minimizes* the probability of returning a copy of H_i. The crucial observation of AGS is that we can find T' as follows. First, as said, we have a good estimate $\hat{c}_i$ of the number of colorful copies of H_i. Then, for each k-treelet T_j we can estimate the number of colorful copies of T_j that span a colorful copy of H_i in G as $\hat{c}_i\sigma_{ij}$, where σ_{ij} is the number of spanning trees of H_i isomorphic to T_j. Finally, dividing this estimate by the number t_j of colorful copies of T_j in G yields an estimate of the probability that sample(T_j) spans a copy of H_i. That is,

$$\Pr(\text{sample}(T_j) \text{ yields a copy of } H_i) = \frac{\#\text{ of colorful copies of } T_j \text{ in G spanning } H_i}{\#\text{ of colorful copies of } T_j \text{ in } G} = \frac{\hat{c}_i\sigma_{ij}}{t_j} \quad (5)$$

More in general, we need the probability that sample(T_j) spans a copy of some covered graphlet:

$$\Pr(\text{sample}(T_j) \text{ yields a covered graphlet}) = \frac{1}{t_j}\sum_{H_i \text{ covered}} \hat{c}_i\sigma_{ij} \quad (6)$$

We switch to the treelet T_{j^*} that minimizes this probability, and continue our sampling until a new graphlet becomes covered.

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 $\bar{c} = O(\epsilon^{-2} \ln(s/\delta))$ 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 $H_1, \dots, H_s$ the distinct k-node graphlets and by $T_1, \dots, T_\varsigma$ the distinct k-node treelets.

Algorithm AGS(ϵ, δ)

```
1: $(c_1, \dots, c_s) \leftarrow (0, \dots, 0)$                              ▷ graphlet counts
2: $(w_1, \dots, w_s) \leftarrow (0, \dots, 0)$                              ▷ 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 $1, \dots, s$ 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:        $j^* \leftarrow \arg\min_{j'=1,\dots,\varsigma} \frac{1}{t_{j'}} \sum_{i' \in C} \sigma_{i'j'} \, c_{i'}/w_{i'}$
15:        $T_j \leftarrow T_{j^*}$
16: return $(\frac{c_1}{w_1}, \dots, \frac{c_s}{w_s})$
```

4.2 Approximation guarantees of AGS

This section is entirely dedicated to prove formal statistical guarantees on the estimates returned by AGS. Specifically, we prove that if AGS chooses the "right" treelet T_{j^*}, then we obtain multiplicative error guarantees for all graphlets at once, as desired. Formally:

Theorem 4. *If the tree T_{j^*} chosen by AGS at line 14 minimizes* $\Pr[\text{sample}(T_j)$ *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 $i = 1, \dots, s$.*

The proof requires a martingale analysis, since the distribution from which we draw the graphlets changes over time. 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. We start by recalling the following martingale tail inequality [4, p. 1476]:

Theorem 5 ([4], Theorem 2.2)**.** *Let $(Z_0, Z_1, \dots)$ 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:*

$$\Pr\left[\exists\, t : Z_t \geq Z_0 + z, V_t \leq v\right] \leq \exp\left[-\frac{z^2}{2(v + Mz)}\right] \tag{7}$$

We now plug into Theorem 5 the appropriate quantities from our algorithm.
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 5:

$$\Pr\left[\exists\, t : Z_t \geq z, \sum_{\tau=1}^{t} P_\tau \leq v\right] \leq \exp\left[-\frac{z^2}{2(v+z)}\right] \qquad \forall\, z, v > 0 \tag{8}$$

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 apply (8) 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:

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

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 $(\mathcal{F}_t)_{t \geq 0}$, hence (8) 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 $\sum_{\tau=1}^{t_0} P_\tau = -Z_{t_0} + \sum_{\tau=1}^{t_0} X_\tau < \frac{\epsilon}{1-\epsilon}\bar{c} + 1 + \bar{c} = \frac{1}{1-\epsilon}\bar{c} + 1$. Applying again (8) with $z = \frac{\epsilon}{1-\epsilon}\bar{c}$ and $v = \frac{1}{1-\epsilon}\bar{c} + 1$, we obtain:

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

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 union bound, the probability that $\frac{c_i}{w_i}$ is not within a factor $(1 \pm \epsilon)$ of g_i is at most $\frac{\delta}{s}$. Theorem 4 follows by a union bound on all $i \in [s]$.

4.3 Sampling efficiency of AGS

4.3.1 Near-optimality of AGS

We start by showing that, under a certain assumption, AGS is essentially optimal in the sense of drawing the minimal number of samples necessary to see every graphlet sufficiently often. This can be thought of as comparing against a clairvoyant algorithm, i.e. an algorithm that 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 treelet T_{j^*} chosen by AGS at line 14 minimizes* $\Pr[\text{sample}(T_j)$ *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 rest of this section is devoted to prove 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 $U^t = \{i : c_i^t > 0\}$; 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 $\mathbf{x}^{t+1} = \mathbf{x}^t + \mathbf{e}_{j^*}$, 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 (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.

4.3.2 A general lower bound

We conclude by showing a lower bound for *all* algorithms based solely on the primitive sample(T). This is a natural class, which includes many graphlet sampling algorithms [22, 34, 35]. Formally, we prove:

Theorem 7. *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'. □

4.4 Lower-level optimizations and architectural details

As we did for for the build-up phase, for completeness and reproducibility we describe lower-level details that apply to uniform sampling and AGS.

4.4.1 Alias method sampling

Recall that sampling starts by drawing a node v with probability proportional to $c(T_C, v)$. We do this in time $O(1)$ by using the alias method [32], which requires building an auxiliary lookup table in time and space linear in n. For uniform sampling, the alias table is built in the second stage of the build-up phase. For AGS, the alias table is rebuilt upon a change of treelet. In practice, building the alias table takes negligible amounts of time.

4.4.2 Neighbor buffering

We experimentally observe that, if G has a node v with degree $d_v = \Delta$ much higher than all other degrees, then the sampling rate is very low. The reason is the following. First, if Δ is large then $c(T_C, v)$ is large; hence, v will be often the root for sampling. In addition, drawing a neighbor of u will take time $\Theta(\Delta)$. Summarizing, if Δ is large, the sampling phase will spend most of the time listing v's neighbors. To avoid this problem, we perform buffered sampling: if d_v is at least some parameter Δ_0, then we use reservoir sampling to sample B neighbors of v. This can be done at essentially the same cost of sampling one neighbor, but we can cache the remaining $B - 1$ for the future. In this way we list the neighbors of large-degree nodes only once in a while. As Figure 10 shows, this increases the sampling speed of GM significantly (we note that L8Motif already achieves those sampling rates and buffering does not increase it further).

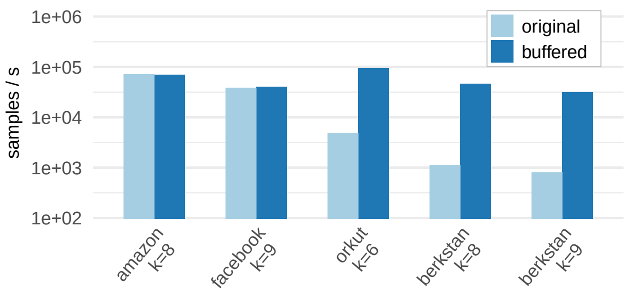


Figure 10: impact of neighbor buffering on sampling.

4.4.3 Graphlet manipulations

Once we have sampled a graphlet copy from G, we have to perform isomorphism tests (to identify its class H) and compute its spanning trees (in order to weight the sample properly). 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. [6]). The resulting matrix can then be reshaped into a $1 \times \frac{k^2-k}{2}$ vector, which fits into 128 bits for all $k \leq 16$. Before encoding a graphlet, GM replaces it with a canonical representative from its isomorphism class, computed using the Nauty library [24]. The isomorphism test then boils down to comparing the encodings. To compute the number of spanning trees σ_i of H_i, GM employs Kirchhoff's matrix-tree theorem which relates σ_i to the determinant of a submatrix of the Laplacian H_i. The running time is $O(k^3)$. To compute the number σ_{ij} of occurrences of a specific treelet T_i in H_j (needed for our sampling algorithm AGS, see Section 4.1), we use an in-memory implementation of the build-up phase where each vertex H_j is assigned a distinct color in $\{0, \ldots, k-1\}$.

5 Experimental results

In this section we measure the performance of GM and L8Motif in terms of running time, space usage, and accuracy of the counts, with a special attention towards L8Motif. We recall that, as shown in [9], CC is the current state of the art, and in particular it outperforms algorithms based on random walks. Therefore we limit our comparison against CC (whenever possible, since on many instances CC dies by memory exhaustion or integer overflow). All our experiments are performed on an Amazon EC2 `c5d.9xlarge` instance, with 36 virtual CPUs, 72GB of main memory, and a 900GB solid-state disk drive (we recall that we store the count tables in external memory).

To begin, we tested GM and L8Motif on all our graphs for increasing values of k, stopping when witnessing a slow-down due to excessive I/O (recall that our algorithms must repeatedly read and store the count tables in main memory). The sampling phase took 5 million samples (as we show, this appeared sufficient to guarantee high accuracy on most graphlets). Table 1 summarizes the results. Using L8Motif we reached $k = 8$ on all our graphs, and using GM we reached $k > 8$ on half of them. The table does not show the YeastProtein graph [21], a small graph on which we tested GM for $k = 16$ in less than three hours (we recall that there are $6 \cdot 10^{22}$ distinct motifs on 16 nodes). For comparison, [3] on YeastProtein reached $k = 10$ and only on treelets and tree-like subgraphs.

Table 1: Summary of our results. For each graph we report the maximum reached value of k and the total wall time (* = with biased coloring). The wall time includes sampling.

graph	nodes (millions)	edges (millions)	source	k	wall time	algorithm
Facebook	0.1	0.8	MPI-SWS	11	1h	GM
Dblp	0.9	3.4	SNAP	9	7m	GM
Amazon	0.7	3.5	SNAP	9	8m	GM
BerkStan	0.7	6.6	SNAP	9	55m	GM
Yelp	7.2	26.1	YLP	8	13m	L8Motif
LiveJournal	5.4	49.5	LAW	8	24m	L8Motif
Orkut	3.1	117.2	MPI-SWS	8	1h11m	L8Motif
Twitter	41.7	1202.5	LAW	8*	2h45m	L8Motif
Friendster	65.6	1806.1	SNAP	8*	1h10m	L8Motif

5.1 Computational efficiency

Figure 11 shows the running time (seconds) and total space usage of the build-up phase (GB), and the speed of the sampling phase (graphlets per second) using uniform sampling. We used biased coloring to keep the build-up time below 3 hours for both Twitter and Friendster, while on all the other graphs, the build-up phase already took less than 1 hour. Thus, in a matter of hours, L8Motif yields accurate counts for $k = 8$ on graphs of size significantly larger than the state of the art.

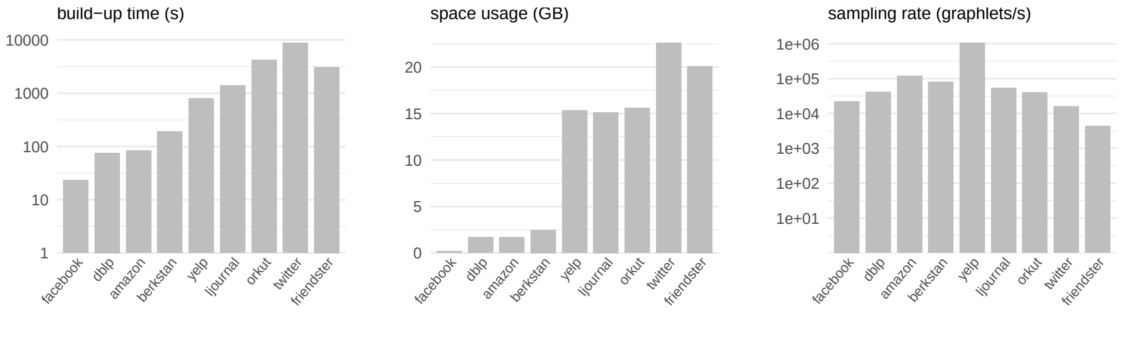
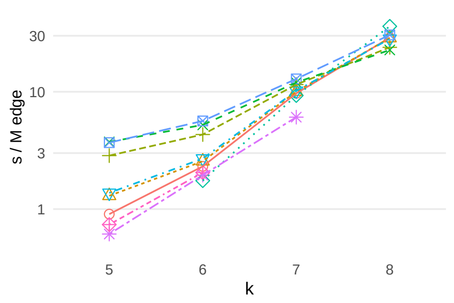


Figure 11: L8Motif's computational performance for $k = 8$.

We also measured the build-up performance of L8Motif as a function of G and k, by computing the average time spent per million edges and the average space used per node — see Figure 12. We do so because, typically, motif counting algorithms have a running time that depends chaotically on G. For instance, ESCAPE takes 5 seconds on a graph with 1.2M edges and 11 days on a graph with 3.6M edges, a blow-up of 175.000 times [27]. A very similar behaviour is exhibited by random walks [8]. In contrast, L8Motif appears to be reasonably predictable as a function of G and k.



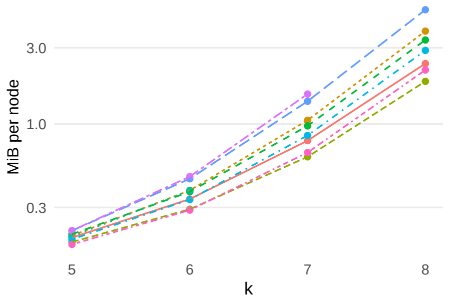


Figure 12: Build-up time in seconds per million edge, and space usage in bits per input node, on all our graphs.

Comparison against CC. We compare GM and L8Motif against CC in Table 2 and 3. For each graph we report the largest k for which CC ran, without dying by memory exhaustion or integer overflow. For the space usage we compare the main memory used by CC to the total external memory usage of our algorithm (recall that CC works in main memory and therefore it is not easily comparable). The sampling rate of our algorithms refers again to uniform sampling, which is the only one supported by CC. The sampling speed of AGS is typically similar, and never more than 40% slower than uniform sampling (recall that AGS has the overhead of repeatedly solving an online problem, computing spanning tree counts, and switching treelets). The only exception is the Yelp graph, on which AGS for $k = 8$ is 20× slower than uniform sampling; but, as we show below, it is also dramatically more accurate (and still 10 000× faster than CC).

Table 2: Computational performance of GM versus CC.

graph	k	build-up time (seconds)			build-up space (GB)			sampling rate (motifs/sec)		
		CC	GM	speedup	CC	GM	reduction	CC	GM	speedup
Facebook	9	860	86	10×	33	5	7×	5	95k	18 400×
Dblp	9	1245	320	3.9×	43	44	1×	72	140k	1 900×
Amazon	9	376	84	2.2×	51	49	1×	226	116k	512×
BerkStan	5	14	7	2×	18	0.5	36×	160	5300	33×
Yelp	5	167	71	2.4×	36	4.5	8×	20	470	24×
LiveJournal	6	306	99	3×	36	9.5	4×	295	16k	54×
Orkut	5	225	40	5.6×	27	3.2	8×	295	17k	58×

5.2 Accuracy of the estimates and performance of AGS

We evaluate the accuracy of the estimates produced by our algorithm L8Motif. The accuracy of GM is the same, as the output of the two algorithms is identical (but L8Motif is faster). To

Table 3: Computational performance of L8Motif versus CC.

graph	k	build-up time (seconds)			build-up space (GB)			sampling rate (motifs/sec)		
		CC	LM	speedup	CC	LM	reduction	CC	LM	speedup
Facebook	8	95	22	4.3×	24	0.2	114×	420	23k	55×
Dblp	8	182	77	2.4×	30	1.7	1.8×	680	41k	60×
Amazon	8	140	84	1.7×	31	1.8	17×	1550	120k	77×
BerkStan	5	14	7	2×	18	0.2	130×	160	1.2M	7 700×
Yelp	5	167	69	2.4×	36	1.3	27×	20	4.1M	200 000×
LiveJournal	6	306	79	3.9×	36	1.8	20×	295	112k	380×
Orkut	5	225	38	6×	27	0.7	40×	295	162k	550×

begin, for each graph we compute a ground-truth count of the number of copies of each possible k-graphlet. For $k = 5$ we used the exact algorithm ESCAPE [27] which works well on small graphs (Facebook, Dblp, Amazon, LiveJournal, Orkut). On all other graphs and/or for $k > 6$, we used as ground truth the average of the counts returned by 20 independent runs of L8Motif, of which 10 used uniform sampling and 10 used AGS. We then measured the average accuracy of L8Motif against the ground truth, over 10 runs. In each run we took 10M samples, or in any case stopped the sampling after 600s (ten minutes). To quantify the accuracy, we compute the per-graphlet multiplicative count error. Denote by c_H the ground-truth count of a specific graphlet H, and let $\hat{c}_H$ be the estimate returned by the algorithm. Then we define the relative error of H as:

$$\mathrm{err}_H = \frac{\hat{c}_H - c_H}{c_H}. \tag{12}$$

Therefore a value $\mathrm{err}_H \simeq 0$ means c_H has been accurately estimated; whereas $\mathrm{err}_H \gg 0$ means c_H is overestimated, and $\mathrm{err}_H \ll 0$ means c_H is underestimated (and the extreme case $\mathrm{err}_H = -1$ means no copy of H was found).

Figure 13 shows the distribution of the relative error err_H for uniform sampling, for $k = 7$, on three representative graphs (all other graphs behave similarly). The x-axis shows the value of err_H, and the y-axis the number of H for which that value is achieved. Note that for Yelp and Amazon almost all graphlets have $\mathrm{err}_H = -1$, as can be seen by the straight segments leaving from the left part of the plot. This means uniform sampling misses almost all graphlets on Amazon and Yelp.

Figure 14 gives the same plot, but for AGS. One can see that the distribution of the relative error is now concentrated around 0: AGS gives an accurate estimate of nearly all graphlets, in line with our theoretical predictions.

To complete the evaluation of AGS, we computed the number of graphlets with relative error below 0.25. This number is shown in Figure 15, where the shaded area represents the maximum achievable, i.e., $N_8 = 11\,117$ (the number of non-isomorphic simple connected graphs on 8 nodes). The plot is particularly telling if we look at the Yelp graph. According to our ground truth, in this graph over 99.9996% of all 8-graphlets are stars. Thus, we can expect uniform sampling to waste essentially all of its samples by drawing stars. The figure shows this is exactly the case, and indeed uniform sampling achieves a relative error $\leq 25\%$ only for the 4 most frequent graphlets (as a fraction, 0.04% of the total). AGS instead achieves a relative error ≤ 0.25 for 9 860 graphlets (as a fraction, 89% of the total). This includes many graphlets with frequency below 10^{-21} which are well-estimated in all the 10 independent runs (and thus are not just noise). To find those graphlets, uniform sampling would need more than 10^3 years even if running at

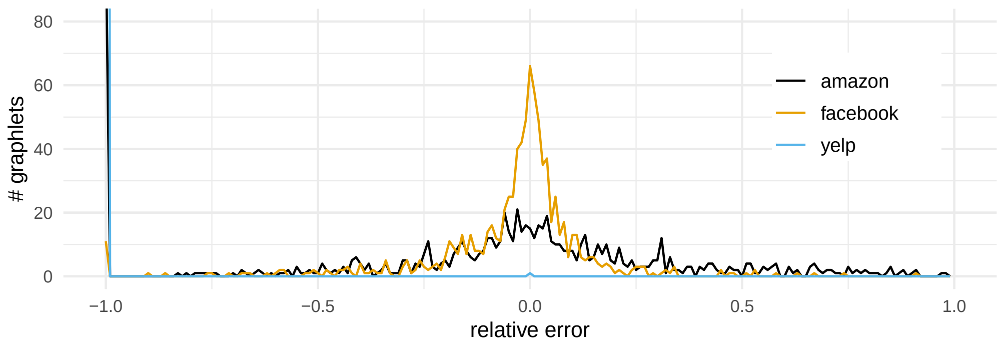


Figure 13: Relative error distribution of uniform sampling for $k = 7$. On Yelp and Amazon almost all graphlets have a relative error of -1, i.e., they are completely missed.

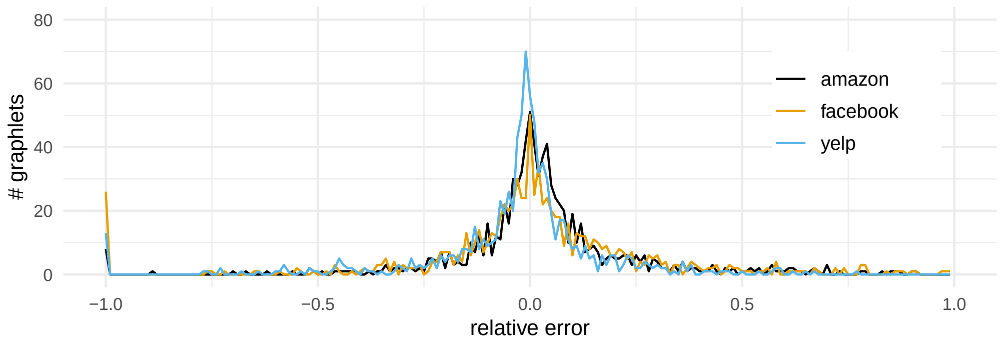


Figure 14: Relative error distribution of AGS for $k = 7$. Unlike the case of uniform sampling, here almost all graphlets are accurately estimated and thus have error $\simeq 0$.

10^9 samples per second. Thus, AGS can count extremely rare graphlets, which uniform sampling simply cannot.

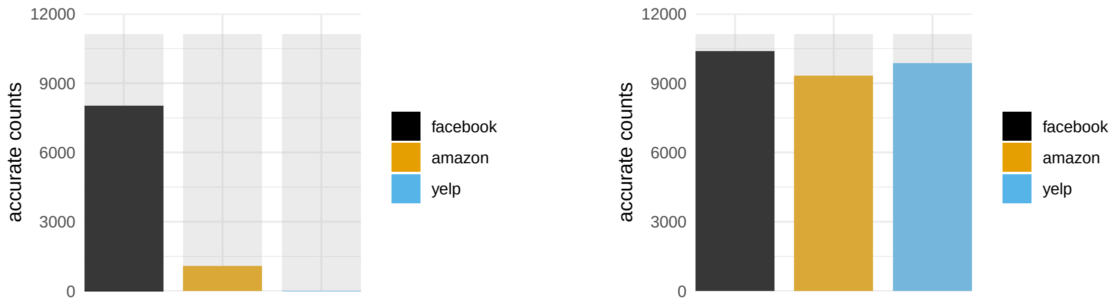


Figure 15: Number of 8-graphlets for which L8Motif achieved relative error below 25%. Left: uniform sampling. Right: AGS. The shaded area shows for reference the total number of 8-graphlets, $N_8 = 11\,117$.

6 Conclusions

In this work we confirm that color coding is an effective technique for sampling and counting motifs in large graphs. Although this was already suggested by existing work, here we refine the approach and push the color coding motif mining paradigm forward. It would be interesting to investigate how this color coding approach could be extended to richer and more challenging scenarios. Two of these scenarios that fit well with the assumption of large graphs are a distributed computing setting and graphs that evolve in time.

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, P. Dao, I. Hajirasouliha, F. Hormozdiari, and S. C. Sahinalp. Biomolecular network motif counting and discovery by color coding. *Bioinformatics*, 24(13):i241–249, Jul 2008.

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

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

[6] 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.

[7] 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.

[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] M. Bressan, S. Leucci, and A. Panconesi. Motivo: Fast motif counting via succinct color coding and adaptive sampling. *Proc. VLDB Endow.*, 12(11):1651–1663, July 2019.

[11] 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.

[12] 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.

[13] X. Chen, R. Dathathri, G. Gill, and K. Pingali. Pangolin: An efficient and flexible graph mining system on CPU and GPU. *Proc. VLDB Endow.*, 13(8):1190–1205, Apr. 2020.

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

[15] V. Dias, C. H. C. Teixeira, D. Guedes, W. Meira, and S. Parthasarathy. Fractal: A general-purpose graph pattern mining system. In *Proc. of ACM SIGMOD*, pages 1357–1374, 2019.

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

[17] I. Finocchi, M. Finocchi, and E. G. Fusco. Clique counting in MapReduce: Algorithms and experiments. *ACM J. Exp. Algorithmics*, 20, Oct. 2015.

[18] 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.

[19] F. Hüffner, S. Wernicke, and T. Zichner. Algorithm engineering for color-coding with applications to signaling pathway detection. *Algorithmica*, 52:114–132, 2007.

[20] 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.

[21] H. Jeong, S. P. Mason, A.-L. Barabási, and Z. N. Oltvai. Lethality and centrality in protein networks. *Nature*, 411(6833):41–42, May 2001.

[22] 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.

[23] D. Mawhirter and B. Wu. AutoMine: Harmonizing high-level abstraction and high performance for graph mining. In *Proc. of ACM SOSP*, pages 509–523, 2019.

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

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

[26] K. Paramonov, D. Shemetov, and J. Sharpnack. Estimating graphlet statistics via lifting. In *Proc. of ACM SIGKDD*, pages 587–595, 2019.

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

[28] S. Ranu and A. K. Singh. GraphSig: A scalable approach to mining significant subgraphs in large graph databases. In *Proc. of IEEE ICDE*, pages 844–855, 2009.

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

[30] 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 *Proc. of ACM SOSP*, pages 425–440, 2015.

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

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

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] E. Yeger-Lotem, S. Sattath, N. Kashtan, S. Itzkovitz, R. Milo, R. Y. Pinter, U. Alon, and H. Margalit. Network motifs in integrated cellular networks of transcription–regulation and protein–protein interaction. *Proceedings of the National Academy of Sciences*, 101(16):5934–5939, 2004.

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

[39] 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.

[40] Z. Zhao, G. Wang, A. R. Butt, M. Khan, V. S. A. Kumar, and M. V. Marathe. SAHAD: Subgraph analysis in massive networks using Hadoop. In *Proc. of IEEE IPDPS*, pages 390–401, 2012.