Faster motif counting via succinct color coding and adaptive sampling

Marco Bressan
Università degli Studi di Milano
marco.bressan@unimi.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 [39], graph clustering [41], and biology [40]. Of particular interest are the subgraphs that are induced and connected, which are 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 [25, 34, 39, 41]. 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 [3, 8, 12, 15, 20, 25, 30, 32, 36, 37, 38, 43]. Entire frameworks have been designed to make motif mining easy, including systems based on graph databases such as Arabesque [33] and GraphSig [31], or standalone systems such as Fractal [16] and AutoMine [26].

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 $n^{\Omega(k)}$ [13] 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 special attention on guarantees. More precisely, we aim to estimate 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 enumerating and/or counting 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 [30] or [26] are reported to work only for $k \leq 5$. We note that combinatorial explosion also affects 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 [8]. 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 [9], making it comparable to brute-force enumeration. On the other hand, [9] showed that one can efficiently sample subgraphs using the color coding technique of Alon, Yuster and Zwick [6]. The idea of color coding is to assign to each node of G one of k colors independently and uniformly at random. Then, via dynamic programming, one can count the number of subtrees of G that span k distinct colors (we say they are *colorful*) in time $O(E_G)$, which gives an estimate of the actual number of subtrees in G. While [6] used color coding for counting trees, [9] 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 *color coding*), 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 [10].

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 the memory usage, which as mentioned above, grows exponentially with k. This limits the scalability; for example, even on a machine with 72GB of main memory, CC runs out of memory while 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 additive. 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. Since in many graphs nearly all graphlets have extremely low frequencies, CC would need to draw an unacceptable 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 MOTIVO, which is designed to count motifs on $k \le 16$ nodes. The second is L8MOTIF (pronounced "leitmotif", for *L*arge-graph 8-node *M*otif counter), which is optimized for motifs on $k \le 8$ nodes. In particular, L8MOTIF can count motifs 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, MOTIVO 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 $\epsilon \le 0.25$ for $\simeq 10.000$ distinct graphlets simultaneously. Many of these graphlets have frequency $\le 10^{-21}$, which means that any algorithm 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 these results.

Technical contributions. Our algorithms MOTIVO and L8MOTIF rely on two technical contributions of different nature. The first contribution is a set of efficient algorithms and data structures that improve the running time and especially the space usage of the build-up phase (one of the main bottlenecks of CC). In particular, for L8MOTIF we design data structures that use a nearly-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 how to entirely skip the heaviest round of the dynamic program via a balanced treelet decomposition trick, saving additional time and space. Thanks to these ingredients we can scale the build-up phase from millions to billions of edges and from $k = 5$ to $k = 8$, although from an asymptotic point of view we are still subject to the $O(a^k m)$ running

---

[2]All measurements are obtained on a workstation with 36 cores and 72GB of main memory — see Section 5.
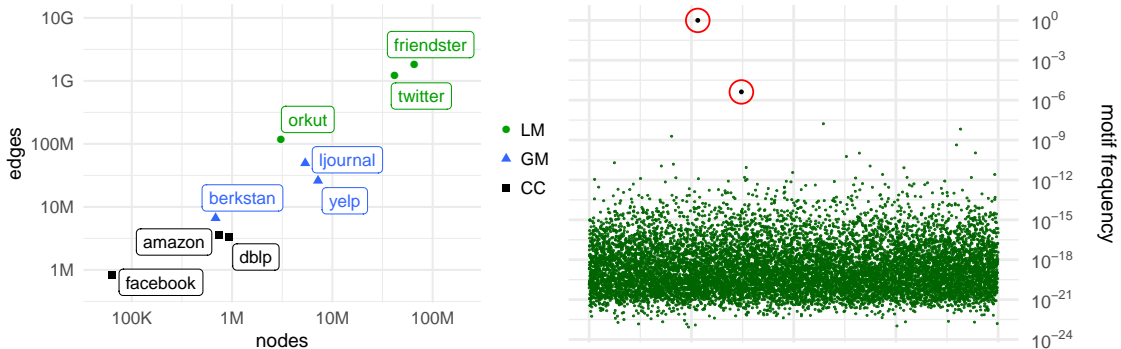
Figure 1: Our performance in a picture for motifs on $k = 8$ nodes. Left: the size of the graphs managed by the state-of-the-art CC and by our algorithms Motivo and L8Motif (abbreviated as LM). Right: the dense green band has one point for each distinct graphlet that we count accurately on the Yelp graph ($\simeq 10.000$ in total) while the two red circles show the only two graphlets detected by CC.

time and $O(c^k n)$ space usage bounds of CC. Our other algorithm, Motivo, 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 Motivo 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 deleting ~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 Motivo in [11], 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 Motivo by almost an order of magnitude, which allows us to scale $k = 7$ to $k = 8$ on the largest graphs. Recall that the time and space requirements grow exponentially with k, hence, scaling from k to $(k+1)$ is a considerable challenge. To give an idea, to count 6-node motifs on our largest graph Friendster ($\sim$ 2B nodes), Motivo needs 130 GB of space and 3 hours of time, while L8Motif needs only 25GB and 30 minutes. The source code of both Motivo and L8Motif is publicly available at `https://gitlab.com/steven3k/motivo`.

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 describes 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 several public datasets.

1.2   Related work

The traditional approach to subgraph counting is based on combinatorial counting. This approach works for $k = 4$ and 5, but becomes unusable for $k > 5$, even for approximate counting. This limitation affects many motif mining tools such as Arabesque [33], ESCAPE [30], Fractal [16], Pangolin [14], AutoMine [26], and DISC [42]. None of these works reports results for $k > 5$. The only algorithms that can scale to $k > 5$ are based on random walks or color coding.

The random walk approach works as follows. 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, and compute an unbiased estimator of its frequency [8, 36, 15, 20]. Unfortunately, the random walk may need $\Omega(n^{k-1})$ steps to reach the limit distribution, or even just to find the most frequent graphlet of G when G is fast-mixing [9, 10]. The running time of this approach is therefore close to $O(n^k)$, which is the running time of the naive brute-force enumeration. One can mitigate this mixing time explosion by walking on subgraphs on less than k nodes [15] or by sampling a spanning tree directly in G [29]. Unfortunately, this gives biased samples that needs to be reweighted, increasing the estimator variance so that $\Omega(n^{k-1})$ samples become necessary again. In addition, 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. Moreover, often they count only non-induced copies [21, 4, 43, 44, 32, 19, 12]. Here instead we want to count the *induced* copies of *all* graphlets at once. The only algorithm that can do so is the CC algorithm of [9, 10]. 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. We note that specialized algorithms can efficiently compute or estimate the number of cliques on graphs with tens or hundreds of millions of edges [26, 22, 23]. 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 consider $k \leq 16$, and we pay particular attention to $k \leq 8$. 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. With a mild abuse of notation, in

this paper we use $[k]$ to denote the set $\{0, \ldots, k-1\}$. 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 of v in G.

2   Color coding and the CC algorithm

Our algorithms, like the CC algorithm of [9, 10], are based on the color coding technique by Alon, Yuster and Zwick [6], which works as follows. First, we assign to each node $v \in G$ a color chosen uniformly and independently in $[k]$. 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 main idea 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. The running time is exponential in k, but linear in m.

Color coding was introduced to detect and count noninduced trees. Then, the authors of [9, 10] showed how to extend it for sampling graphlets. 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 [9, 10]. We now detail the two phases of CC, the *build-up phase* and the *sampling phase*, which will also be the same phases used in our algorithms.

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 $h \le k$. 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 [10]):

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

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 ([10], 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 needs 50GB of memory [10].

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 (u, C''), where $u \sim v$ and $C'' \subset C$ contains $|T''|$ colors, with probability proportional to $\beta_T^{-1} c((T', C \backslash C''), v) \cdot c((T'', C''), u)$. 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 u. 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 [10].

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 achieve this, we sample a treelet copy as described above. This treelet copy 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 quickly 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

The crucial point of the algorithm just described is the accuracy of the graphlet estimates, $\hat{g}_i$. There are two sources of error: the coloring, which distorts the true graphlet distribution into the colorful one, and the sampling.

Regarding the coloring error, one can prove that the colorful graphlet distribution is statistically close to the actual graphlet distribution of G. First, we report a bound from [10], slightly rephrased. Let $g = \sum_i g_i$ be the total number of induced k-graphlet copies in G. Then:

Theorem 2 ([10] Thm 5.3). *For all $\epsilon > 0$, a random coloring of G with k colors gives:*

$$\Pr\left[\left|\frac{c_i}{p_k} - g_i\right| > \frac{2\epsilon g}{1 - \epsilon}\right] = \exp\left(-\Omega\left(\epsilon^2 g^{1/k}\right)\right). \tag{3}$$

Note that the bound above is additive, that is, it establishes a deviation proportional to g, the total number of graphlets. In this work we complement it with a multiplicative bound that holds when the maximum degree Δ of G is small.

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

$$\Pr\left[\left|\frac{c_i}{p_k} - g_i\right| > \epsilon g_i\right] < 2\exp\left(-\frac{2\epsilon^2 p_k^2 g_i}{(k-1)! \Delta^{k-2}}\right). \tag{4}$$

Proof. We use a concentration bound for dependent random variables from [17]. 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 $h_1, h_2 \in \mathcal{V}_i$, the random variables X_{h_1}, X_{h_2} are independent if and only if $|V(h_1) \cap V(h_2)| \leq 1$, which means 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 $\max_{u,v \in G} g(u, v) \leq (k-1)! \Delta^{k-2} - 1$ and thus $\chi_k \leq (k-1)! \Delta^{k-2}$. The bound then follows immediately from Theorem 3.2 of [17] 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 [6]; 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 the sampling error, standard concentration bounds apply to the error of uniform sampling (see above). For AGS the analysis is more complex, and is shown in Section 4.1.

3 Fast construction of a compact treelet database

This section details the internals of the build-up phase of MOTIVO and L8MOTIF. Recall that the goal of this phase is to compute a compact database of colorful treelet counts (Section 2). This database will support 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

Using these two operations, we can implement graphlet sampling, by first selecting a node v with probability proportional to the number of treelets rooted in it, and then sampling one such treelet uniformly at random. For our adaptive sampling scheme AGS, we will also support the following operations:

- `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 MOTIVO and L8MOTIF, we describe them incrementally. We start in Section 3.1 by discussing the implementation of CC's build-up phase. Then, in Section 3.2 (MOTIVO) and Section 3.3 (L8MOTIF) we progressively replace the algorithms and data structures of CC with ours, measuring the cumulative performance impact. To measure the performance impact for MOTIVO, we use as baseline a C++ version of CC (which is originally in Java), that we wrote by carefully porting all algorithms and data structures. The baseline for L8MOTIF is MOTIVO itself. Finally, in Subsection 3.4 and Subsection 3.5 we describe additional optimizations for both MOTIVO and L8MOTIF.

3.1 Treelets and counts

In this subsection we discuss the crucial aspects of the build-up phase of CC. This phase spends nearly all its time in manipulating rooted (un)colored treelets and reading/writing their counts, as described in Section 2. Therefore, to speed up the phase, we need to make the manipulation of these objects as efficient as possible.

The first computationally intensive task is merging the treelet counts. Consider a single treelet count $c(T_C, v)$. To compute $c(T_C, v)$, both CC and our algorithms process all the neighbors u of v as follows. First, suppose we have defined 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 $T''_{C''}$ is not smaller than the smallest subtree rooted in a child of the root of $T'_{C'}$. Here, "smaller" and "smallest" are determined by the total order. If these conditions hold, then $T'_{C'}$ and $T''_{C''}$ can be *merged* into a treelet T_C. It is easy to see that for each occurrence of T_C rooted in v there are exactly β_T pairs of: (i) an occurrence of a treelet $T'_{C'}$ rooted in v, and (ii) an occurrence of a treelet $T''_{C''}$ rooted in a neighbor u of v, for which the above procedure on $T'_{C'}$ and $T''_{C''}$ returns T_C. Consequently, the value of $c(T_C, v)$ is incremented by $\beta_T^{-1} c(T'_{C'}, v) \cdot c(T''_{C''}, u)$. This procedure is essentially a direct computation of the sum in Equation 2, once a particular decomposition of T_C has been chosen by our total order. Since Equation 2 holds regardless of the chosen decomposition, this implies the correctness of the build-up phase as well. 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 quite expensive. This is because CC encodes each treelet T_C as a classic, pointer-based tree data structure. Here, we show a different implementation that makes the check-and-merge operation 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 $c(T_C, v)$. In the hash table, the key used for $c(T_C, v)$ is a 64-bit pointer to an instance of T_C. Thus, 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 space used by 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 $17\times$ slower than CC, and with the latter one it is up to $7\times$ slower. Nonetheless, after all optimizations are in place, we are always faster than CC.

3.2 Motivo: a general-purpose motif counter for k ≤ 16 nodes

We describe our first toolbox, MOTIVO, introduced in our preliminary work [11], for graphlets on up to 16 nodes. Starting from the C++ porting of CC, we introduce succinct treelets and the succinct count table, obtaining the performance improvement shown in Figure 2.

3.2.1 Succinct treelets and count table

Treelet representation. We drop CC's pointer-based structures, and we represent treelets as bitstrings. This accelerates `merge`(T, T') by up to $150\times$ for $k = 5$ and up to $1000\times$ for $k = 7$.
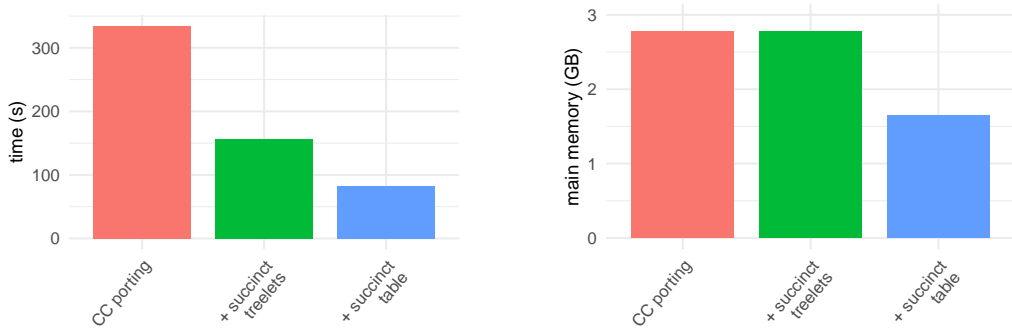

Figure 2: Cumulative impact of our optimizations on the build-up phase, for Amazon with $k = 6$. The baseline is the C++ porting of CC.

Given an uncolored treelet T rooted at r, its bitstring representation s_T is defined as follows. Perform a DFS traversal of T starting from r. The i-th bit of s_T is 1 (resp. 0) if the i-th edge is traversed moving away from (resp. towards) r.

This encoding exhibits several nice properties. First, for all $k \leq 16$, it requires at most 30 bits, which fit in a 4-byte integer type. Second, the total order over the treelets T is just the lexicographic order over their encodings (and we can show that this coincides with the total order used by CC as well). Third, the order serves also as 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. Fourth, merging T' and T'' into T boils down to concatenating the bitstrings $1, s_{T''}, s_{T'}$, in this order, which is typically faster than manipulating a pointer-based structure.

To encode a *colored* rooted treelet, $T_C = (T, C)$, we simply concatenate s_T and the characteristic vector s_C of C.[5] For all $k \leq 16$, the resulting bitstring s_{T_C} fits in 46 bits. Set-theoretical operations on C become bitwise operations over s_C (or for union, and for intersection). And again, the lexicographical order of the encodings gives the total order over the treelets. An example of a colored rooted treelet and its encoding is given in Figure 3.
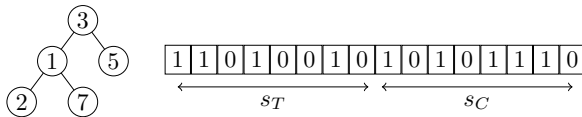

Figure 3: A colored rooted treelet T_C and its encoding, shown for simplicity on just $8 + 8 = 16$ bits. The label ℓ of each node represents its color and the corresponding ℓ-th least significant bit of s_C is set to 1. s_T is the bitstring representing the tree T.

Count tables. Instead of using hash tables, Motivo maintains the key-value pairs $(T_C, c(T_C, v))$ in a set of arrays, one for each $v \in G$ and each treelet size $h \in \{1, \ldots, k\}$, storing only the entries with a non-zero count $c(T_C, v) > 0$ (note that each array is built only once and never modified thereafter). Clearly, using arrays makes iteration extremely fast. Moreover, given the key of an entry, we do not need to dereference any pointer to access the corresponding colored treelet

---

[5]Given a universe $U = 0, 1, \ldots, eta$, the characteristic vector $\langle x_0, x_1, \ldots, x_{\eta-1} \rangle$ of a subset $S \subseteq U$ contains one bit x_i for each element in U, where x_i is 1 if $i \in S$ and 0 otherwise.

—the key is the treelet itself. The price to pay is that searching for a given key is potentially more expensive. However, by sorting every array according to the total order over the keys (see above), we can find a key via binary search in time $O(k)$, since every array has length $O(6^k)$.[6]

For what concerns the treelet counts themselves, while CC uses 64 bits per count, MOTIVO uses 128 bits per count. This adds a small overhead,[7] but avoids dangerous overflows for large values of k.[8] Moreover, since each key fits into 48 bits, each array entry actually uses 48+128=176 bits. We conclude with one final optimization. In place of $c(T_C, v)$, MOTIVO stores the *cumulative* count $\eta(T_C, v) = \sum_{T'_{C'} \leq T_C} c(T'_{C'}, v)$, see Figure 4 for a toy example. This has several benefits: (i) the count $c(T_C, v)$ can be computed with negligible overhead as the difference between two consecutive entries, (ii) the total treelet count η_v for v is at the end of the record, and (iii) we can sample a uniform random treelet rooted in v in time $O(k)$, by drawing a uniform random value X in $\{1, \ldots, \eta_v\}$ and then doing a binary search for X over the array of v.

3.3 L8Motif: counting 8-graphlets in large graphs

In this subsection we describe L8MOTIF, a version of MOTIVO specialized for $k \leq 8$. To this end, we start with MOTIVO and plug in three new ingredients which improve its performance —see Figure 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 an 11-bit integer. This reduces the space by more than $4\times$ compared to the bitstring encoding of MOTIVO. From an asymptotic 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. To ensure treelets are memory-aligned, we pad the 11-bit representation into a 16-bit integer. This leaves 5 spare bits, that we use to encode the length of our variable-length count, see described in Section 3.3.3. The impact of ITE on the overall space usage of MOTIVO is around $\simeq 20\%$ on all instances (see Figure 5).

Adopting ITE makes treelet manipulations harder. Recall for example that $\mathtt{merge}(T'_{C'}, T''_{C''})$ checks if $T'_{C'}$ and $T''_{C''}$ can be merged into a treelet T_C, and that T can be decomposed into T'
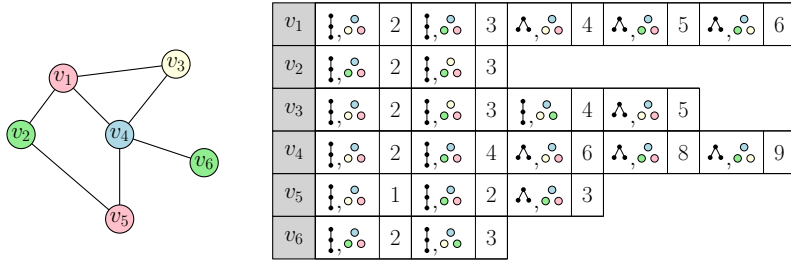

Figure 4: Left: a graph G whose vertices have been colored using $k = 4$ colors. Right: a graphical representation of the count table (implicitly) storing the number $c(T_C, v)$ of all colored treelets T_C of size 3 in G. Notice how we actually store $\eta(T_C, v)$ instead of $c(T_C, v)$.

---

[6]By Cayley's formula: there are $O(3^k k^{-3/2})$ rooted treelets on k vertices [28], 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.

[8]Already for $k = 6$, the number of k-stars centered in a node of degree 2^{16} is $\approx 10^{22}$.

and T''. Since in ITE a treelet is just a number, `merge` would need to go back and forth between ITE and bitstring encodings. To resolve, we *precompute* the results of $\mathtt{merge}(T'_{C'}, T''_{C''})$ on all treelets on up to 8 nodes, and store all the results, in ITE format, in a bidimensional array. In this array, entry $[i][j]$ is the ITE index of the treelet resulting from merging i and j (or -1 for FAIL). Another array tells, for every ITE index of a treelet T, the ITE indices i, j of the treelets of the canonical decomposition of T. Other arrays tell us the ITE index of the treelet T and the set of colors C associated to a colored treelet $T_C = (T, C)$, and so on. Thus, each treelet operation is just a sequence of black-blox lookups in these arrays, using only the ITE encoding. The total size of all arrays is less than 3 megabytes, which fits in the cache of a modern CPU. With this technique, the time taken by a single treelet operation is similar to the one required by the bitstring encoding of Section 3.2.1.
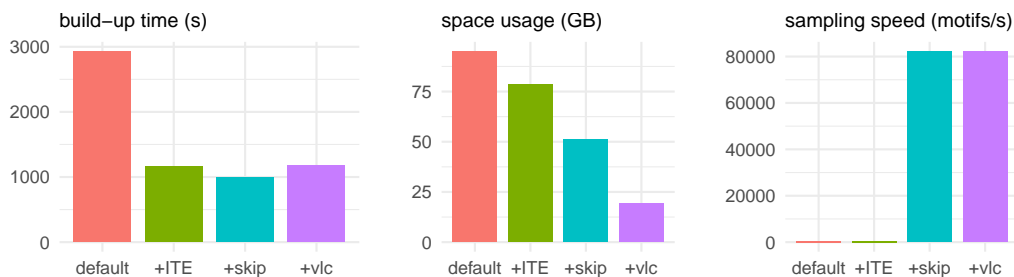

Figure 5: Twitter graph, $k = 6$. Cumulative impact of using ITE, skipping the heaviest round of the dynamic program, and using variable-length counts. The "default" baseline is Motivo, described in the previous section.

3.3.2 Round skipping via balanced treelet decomposition

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 $k \le 8$, 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. We would like to reduce the time and space usage 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; for all other treelets, we can find a decomposition where the largest subtree contains at most $k-2$ nodes. And for stars, we can avoid counting them at all: at sampling time, they 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 basic fact:

Lemma 1. *Any k-treelet T that is not a star has an edge that, if cut, yields two trees each with at most $k-2$ nodes.*

Proof. If T is not a star then it contains a path on 3 edges, say (x, u, v, y). Cutting (u, v) yields two trees, each one with at least 2 nodes or, equivalently, at most $(k-2)$ nodes. $\qquad\square$

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 into T' and T'', that is, one where both T' and T'' have at most $\leq k-2$ nodes, 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 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).
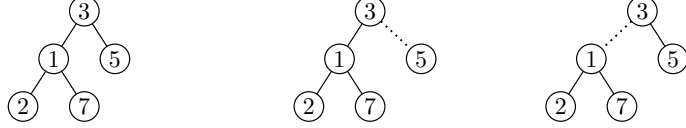

Figure 6: A rooted colored 5-treelet (left) with its original decomposition in two subtrees on 4 and 1 nodes (middle) and its balanced decomposition in two subtrees on 3 and 2 nodes (right).
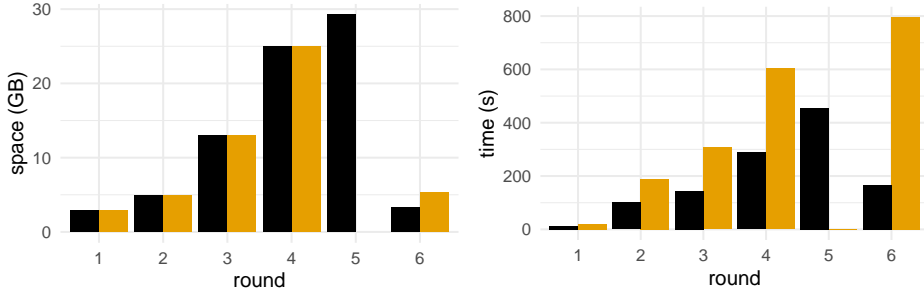

Figure 7: Build-up phase on TWITTER for $k = 6$: space and time usage of single rounds, before and after adopting balanced treelet decompositions and round skipping.

As said, since we do not have the counts of k-stars anymore, to sample a star we now 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, which is even better since we avoid the noise introduced by coloring. Sampling in this fashion is much faster than using the count table, and since most treelets of G are often stars, the sampling rate can increase by several orders of magnitude (see Figure 5).

3.3.3 Variable-length counts

Our third and final ingredient is aimed again at saving space. To this end, we encode each treelet count $c(T_C, v)$ as a *variable-length count*. The rationale is that, in practice, most of the treelet counts can be stored in very few bits as shown, e.g., by Figure 8. Our variable-length counts can be thought of as a variant of Elias delta coding [18], which can represent any integer x using only $(1 + o(1))\lceil \log_2 x \rceil$ bits. In practice, by using the 5 spare bits left by the encoding of T_C (see Section 3.3.1), we encode each key-value pair $(T_C, c(T_C, v))$ in a byte-aligned memory region, as shown in Figure 9, that is we use:

- 11 bits for the integer representation of T_C, see Section 3.3.1.

- 5 bits for the length ℓ of $c(T_C, v)$, expressed in bytes. Thus, we can support counts $c(T_C, v)$ on up to 256 bits (twice the maximum count length supported by MOTIVO), and with values as large as $2^{256} - 1$.

13

- ℓ bytes for the binary encoding of $c(T_C, v)$

Variable-length counts yield an additional space reduction of $\geq 60\%$ on all graphs for all $k \geq 6$. The downside is that 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. This has a significant impact on the build-up time; in the worst case, we witness an increase of about 50%. However, the gains outweigh the losses: 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.
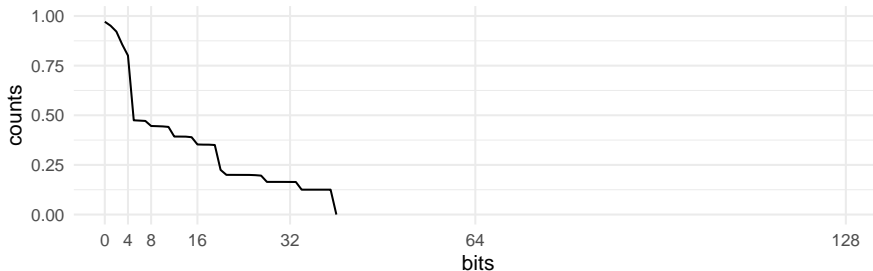

Figure 8: The fraction of treelet counts that require at least b bits, as a function of b, for the treelet table of TWITTER with $k = 6$. The resulting average bit length is just 14, almost 90% less than 128 bits.
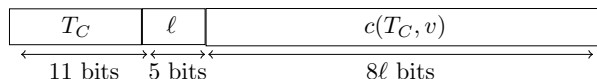

Figure 9: Variable-length encoding of a treelet count.

3.4 Lower-level optimizations and architectural details

For completeness and reproducibility, we describe some additional optimizations and features of MOTIVO and L8MOTIF, some with a significant impact.

3.4.1 Zero-rooting

This is only for MOTIVO. 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 L8MOTIF already counts k-treelets only once, thanks to the balanced treelet decomposition of Section 3.3.2.

3.4.2 Greedy flushing

To reduce the memory footprint, we use a greedy flushing strategy. Suppose we are building the count table of the h-treelets. We temporarily store the record/array of v in a hash table, which

allows for efficient insertions and lookups; when done, we immediately flush it on disk and delete the hash table. In this way we produce the count tables for all nodes of G, in some order. Thus, a second I/O pass is needed to sort the tables by their corresponding node of G, so that they can be retrieved efficiently in the next round. This technique increased the total runtime by at most 10% in all our runs.

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, the count of each node $v \in G$ is computed by an independent thread from a thread pool of a fixed size. 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 all threads in parallel.

3.4.4 Memory-mapped reads

Recall that our treelet count database is stored in external memory. This entails I/O access, since computing the count table for treelets of size h requires 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 the count 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 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). In these cases the overhead is inevitable, as the aggregate size of the tables is close to or even larger than the memory size.

3.5 Biased coloring

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

Consider the following color distribution. We choose each color $i \in \{1, \ldots, k-1\}$ with probability $\lambda \ll \frac{1}{k}$, and color 0 with probability $1 - \lambda(k-1)$. Then, for any set of j colors C, the probability that a given j-treelet is colored with C is:

$$p_{k,j}(C) = \begin{cases} j! \lambda^j & \text{if } 0 \notin C \\ \simeq j! \lambda^{j-1} & \text{if } 0 \in C \end{cases} \tag{5}$$

Therefore, if λ is sufficiently small, for most treelets we will have a zero count at v. Moreover, most nonzero counts will be for a restricted set of colorings – those containing color 0. This reduces the size of the treelet count table, and thus the running time of the algorithm. As a downside we suffer a loss of accuracy, since a lower colorful probability increases the variance of the number c_i of colorful copies of H_i. However, if n is large enough, and a substantial fraction of nodes of G is part of some occurrence 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 is negligible as long as $\lambda^{k-1} n / \Delta^{k-2}$ is large.

This technique allows us to manage our largest instances, TWITTER and FRIENDSTER for $k = 8$, with an acceptable loss of accuracy, see Section 5. In those experiments, we use an educated guess of $\lambda = 0.001$. We note that, otherwise, one could find a good value for λ by setting $\lambda \ll 1/kn$ and then growing λ until a good fraction of counts are positive, at which point Theorem 3 ensures concentration.

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 strategies, and, in many cases, increment the sampling rate substantially.

4.1 Adaptive Graphlet Sampling (AGS)

Recall that the main idea of CC is to build a compact database for sampling k-treelets from G. Interestingly, 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 operation:

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

We can use this primitive to virtually "delete" certain graphlets from the database, and focus on other ones.

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, that 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), which will return only copies of H_1, until we estimate accurately H_1. At this point we call sample(T_2), which will return only copies of H_2, until we estimate accurately H_2, too. Thus, using sample(T) we can estimate both graphlets with just $O(1)$ samples. Clearly, the general situation is more complex, as we have thousands of graphlets with common spanning treelets. Still, the idea can be adapted and it 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 with a multiplicative approximation of $1 + \epsilon$, with a probability of at least $1 - \delta$. 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 switch to another treelet 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. Now the crucial point is that we can find T' as follows. First, 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,

$$\Pr(\text{sample}(T_j) \text{ yields a copy of } H_i) = \frac{\text{\# of colorful copies of } T_j \text{ in G spanning } H_i}{\text{\# of colorful copies of } T_j \text{ in } G} = \frac{\hat{c}_i \sigma_{ij}}{t_j} \quad (6)$$

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

$$\Pr(\text{sample}(T_j) \text{ yields a covered graphlet}) = \frac{1}{t_j} \sum_{H_i \text{ covered}} \hat{c}_i \sigma_{ij} \qquad (7)$$

We switch to the treelet T_{j^*} minimizing this probability, and continue 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. To have a probability of at least $1 - \delta$ of obtaining a multiplicative $(1 + \epsilon)$-approximation 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 $H_1, \ldots, H_s$ the distinct k-node graphlets and by $T_1, \ldots, T_\varsigma$ the distinct k-node treelets.

Algorithm AGS(ϵ, δ)

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

4.2   Approximation guarantees of AGS

This section is dedicated to showing that AGS provides strong statistical guarantees. Our main result is that, if AGS chooses the "right" treelet T_{j^*}, then we obtain multiplicative error guarantees for all graphlets at once. 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 $i = 1, \ldots, s$.*

The proof requires a martingale analysis, since the distribution from which we draw the graphlets changes over time. To this end, from now on we fix a graphlet H_i and analyse the concentration of its estimate. We drop the index i from the notation unless necessary. We start by recalling the following martingale tail inequality from [5]:

Theorem 5 ([5], Theorem 2.2)**.** *Let $(Z_0, Z_1, \ldots)$ be a martingale with respect to the filter $(\mathcal{F}_\tau)_{t \geq 0}$. Suppose that $Z_{\tau+1} - Z_\tau \leq M$ for all τ, and write $V_t = \sum_{\tau=1}^{t} \text{Var}[Z_\tau | \mathcal{F}_{\tau-1}]$. Then for any $z, v > 0$*

we have:

$$\Pr\left[\exists\, t : Z_t \geq Z_0 + z,\, V_t \leq v\right] \leq \exp\left(-\frac{z^2}{2(v + Mz)}\right) \tag{8}$$

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

A) For $t \geq 1$, let X_t be the indicator random variable of the event "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 $Y_j^\tau : j \in [\varsigma],\, \tau = 0, \ldots, t$

D) For any random variable Z, then, $\mathbb{E}[Z \mid \mathcal{F}_t] = \mathbb{E}[Z \mid Y_j^\tau : j \in [\varsigma],\, \tau = 0, \ldots, t]$, and $\mathrm{Var}[Z \mid \mathcal{F}_t]$ is defined analogously

E) For $t \geq 1$ let $P_t = \mathbb{E}[X_t \mid \mathcal{F}_{t-1}]$ 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 $P_t = \sum_{j=1}^\varsigma Y_j^{t-1} a_{ji}$

F) Let $Z_0 = 0$, and for $t \geq 1$ let $Z_t = \sum_{\tau=1}^t (X_\tau - P_\tau)$. 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. $\mathcal{F}_{t-1}$

G) 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 constant over $\mathcal{F}_{t-1}$, 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 $V_t = \sum_{\tau=1}^t \mathrm{Var}[Z_\tau \mid \mathcal{F}_{\tau-1}] \leq \sum_{\tau=1}^t P_\tau$. By applying Theorem 5 above, we obtain:

$$\Pr\left[\exists\, t : Z_t \geq z,\, \sum_{\tau=1}^t P_\tau \leq v\right] \leq \exp\left(-\frac{z^2}{2(v+z)}\right) \qquad \forall\, z, v > 0 \tag{9}$$

Now consider 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, $\sum_{\tau=1}^t P_\tau$ 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 $c_i - g_i w_i$.

We now derive our concentration 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 that $g_i w_i \leq c_i(1 - \frac{\epsilon}{1+\epsilon})$, 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 (9) 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:

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

but $\frac{\epsilon^2 \bar{c}}{2(1+\epsilon)^2} \geq \frac{\epsilon^2}{2(1+\epsilon)^2} \frac{4}{\epsilon^2} \ln\left(\frac{2s}{\delta}\right) \geq \ln\left(\frac{2s}{\delta}\right)$ 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 (9) holds if we replace Z_t with $-Z_t$. Let $t_0 \leq t$ 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 (9) with $z = \frac{\epsilon}{1-\epsilon} \bar{c}$ and $v = \frac{1}{1-\epsilon} \bar{c} + 1$, we obtain:

$$\Pr\left[\frac{c_i}{w_i} \leq g_i(1-\epsilon)\right] \leq \exp\left(-\frac{(\bar{c}\frac{\epsilon}{1-\epsilon})^2}{2(\frac{1+\epsilon}{1-\epsilon}\bar{c}+1)}\right) \leq \exp\left(-\frac{\epsilon^2 \bar{c}^2}{2(1+\bar{c})}\right) \tag{11}$$

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 $\frac{2\epsilon^2 \bar{c}}{5} \geq \frac{2\epsilon^2}{5} \frac{4}{\epsilon^2} \ln\left(\frac{2s}{\delta}\right) > \ln\left(\frac{2s}{\delta}\right)$ and thus once again the probability of deviation is bounded by $\frac{\delta}{2s}$.

By using the union bound on the two cases, the probability that $\frac{c_i}{w_i}$ is not within a factor $(1 \pm \epsilon)$ of g_i is at most $\frac{\delta}{s}$. Using the union bound once again on all $i \in [s]$, we obtain theorem 4.

4.3 Sampling efficiency of AGS

4.3.1 Near-optimality of AGS

Imagine a "clairvoyant" algorithm that knows, for every treelet T_j, the number of invocations of sample(T_j) necessary in order to get the desired accuracy bounds while minimizing the total number of taken samples. We show that the number of samples used by AGS is close to the number of samples used by this clairvoyant algorithm. 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 $\text{sample}()$ 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 proving Theorem 6. We will write the problem of minimizing the total number of samples as a covering problem, and show that AGS is a greedy algorithm for that problem. This will allow us to prove the guarantees of AGS by adapting a standard proof for greedy algorithms for covering problems.

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 $a_{ji} = g_i \sigma_{ij}/t_j$, which is 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 $\mathbf{A} = (a_{ji})^T$, 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:

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

We now describe the natural greedy algorithm for this problem; it turns out that this is precisely AGS. The algorithm works in 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

19

the vector of residuals at time t as $\mathbf{c}^t = \max(\mathbf{0}, \mathbf{c} - \mathbf{A}\mathbf{x}^t)$, and we write $c^t = \langle \mathbf{1}, \mathbf{c}^t \rangle$. 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. Clearly, $U^0 = [s]$.

At step t, the algorithm chooses T_{j^*} such that sample(T_{j^*}) spans an uncovered graphlet with the highest probability. To this end, it computes:

$$j^* := \arg \max_{j=1,\ldots,\varsigma} \sum_{i \in U_t} a_{ji} \tag{12}$$

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. *The greedy algorithm returns a solution of cost $O(z \ln(s))$, where z is the cost of the optimal solution.*

Proof. Let $w_j^t = \sum_{i \in U_t} a_{ji}$. For any $j \in [\varsigma]$ let $\Delta_j^t = c^t - c^{t+1}$. This is the decrease in the overall residual weight that we would obtain if $j^* = j$. Note that $\Delta_j^t \le w_j^t$. We consider two cases.
Case 1: $\Delta_{j^*}^t < w_{j^*}^t$. This means that 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 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 $\Delta_j^t \ge \frac{1}{z}c^t$; otherwise, any solution for the residual problem would have cost strictly larger than z. But, by the choice of j^*, we have $\Delta_{j^*} = w_{j^*}^t \ge w_j^t \ge \Delta_j^t$ for any j, hence $\Delta_{j^*}^t \ge \frac{1}{z}c^t$. Thus by choosing j^* we get $c^{t+1} \le (1 - \frac{1}{z})c^t$. Therefore, after running into this case ℓ times, the residual cost is at most $c^0(1 - \frac{1}{z})^\ell$.

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

Now, note that the treelet index j^* given by (12) 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 includes many natural graphlet sampling algorithms such as [25, 37, 38].

Theorem 7. *For any constant $k \ge 2$ there are graphs on an arbitrarily large number of nodes n such that (i) some graphlet H represents a fraction $p_H = 1/\text{poly}(n) = \Omega(n^{1-k})$ of all graphlet copies, and (ii) 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

20

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 all the graphlets. 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 therefore 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'. $\qquad\square$

4.4 Lower-level optimizations and architectural details

For completeness and reproducibility, as we did for the build-up phase, we describe some lower-level details of 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 [35]. This method requires us to build an *alias table*, which requires $O(n)$ time and space. For uniform sampling, the alias table is built in the second stage of the build-up phase. For AGS, the alias table is rebuilt every time a new treelet is selected. In any case we observe that, in practice, building the alias table consumes a negligible fraction of the overall running time.

4.4.2 Neighbor buffering

We have observed that, if G has a node v with degree $d_v = \Delta$ much higher than all other nodes, then the sampling rate is very low. We argue that the reason is the following. First, if Δ is large then $c(T_C, v)$ is large, so the sampling routine will often choose v as root node. Second, drawing a neighbor of u will take time, as we have to potentially scan $\Theta(\Delta)$ nodes. The combination of these two effects imply that, if Δ is large, the sampling phase will spend most of the time scanning the neighbors of v. To mitigate this problem, the first time we sample a neighbor of v, we actually sample *many* neighbors of v, say B, and keep them for later. To this end we use reservoir sampling, which allows us to sample B neighbors by sweeping over them only once, and thus at the same cost of sampling just one neighbor. We apply this trick to every node v with $d_v \geq \Delta_0$ for some tunable parameter Δ_0. As a result, we scan the neighbors of large-degree nodes only once in a while. As Figure 10 shows, this increases the sampling speed of MOTIVO significantly (we note that L8MOTIF already achieves those sampling rates and buffering does not increase it further).
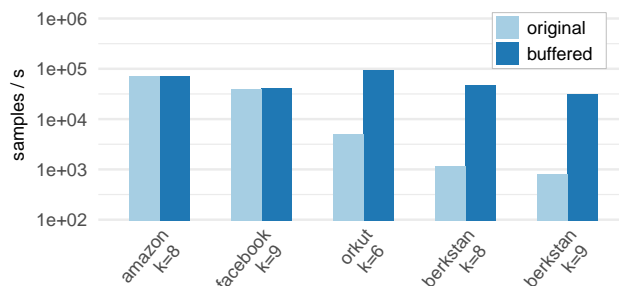

Figure 10: impact of neighbor buffering on sampling.

4.4.3 Graphlet manipulations

Recall that, after sampling a graphlet occurrence, we have to perform isomorphism tests (to identify its class H) and compute its spanning trees (in order to weigh the sample). To perform the isomorphism test, we first replace the graphlet with a canonical representative of its isomorphism class, computed using the Nauty library [27]. Then, as the $k \times k$ (boolean) adjacency matrix of the graphlet is symmetric with diagonal 0, we pack it as 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. [7]). Finally, we further reshape this matrix into a $1 \times \frac{k^2-k}{2}$ vector, which fits into 128 bits for all $k \leq 16$. Therefore, every graphlet is mapped into a 128-bit string identifying its isomorphism class, and the isomorphism test between two graphlets boils down to comparing these 128-bit encodings.

To compute the number of spanning trees σ_i of H_i, we employ 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

We measure the performance of MOTIVO and L8MOTIF in terms of running time, space usage, and accuracy of the counts, with a special attention towards L8MOTIF. We recall that CC is the current state of the art; in particular, algorithms based on random walks are outperformed by CC, see [10]. Therefore, we compare only against CC. 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 used to store the count tables.

To begin, we tested MOTIVO and L8MOTIF on all our graphs for increasing values of k, stopping when witnessing a slowdown due to excessive I/O (recall that our algorithms must repeatedly read and store the count tables on disk). In the sampling phase, we took 5 million samples. As we show below, this was 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 MOTIVO we reached $k > 8$ on half of them. The table does not show the YEASTPROTEIN graph [24], a small graph on which we successfully ran MOTIVO for $k = 16$ in less than three hours (we recall that there are $6 \cdot 10^{22}$ distinct motifs on 16 nodes). For comparison, [4] on YEASTPROTEIN reached $k = 10$ and only on tree-like graphlets.

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	MOTIVO
DBLP	0.9	3.4	SNAP	9	7m	MOTIVO
AMAZON	0.7	3.5	SNAP	9	8m	MOTIVO
BERKSTAN	0.7	6.6	SNAP	9	55m	MOTIVO
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 performance of Motivo and L8Motif for $k = 8$: the running time (seconds), the total space usage of the build-up phase (GB), and the speed of uniform sampling (graphlets/second). This shows how L8Motif manages graphs significantly larger than the state of the art. Note also the reduction in space usage of L8Motif compared to Motivo. For both Twitter and Friendster, we used biased coloring to keep the build-up time below 3 hours.
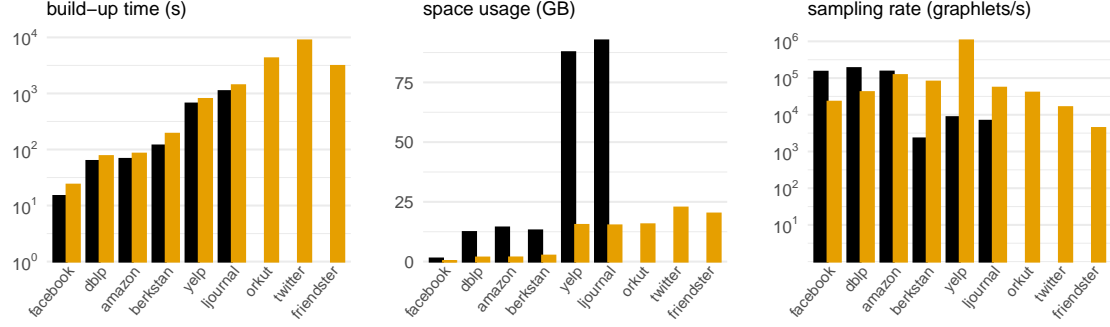

Figure 11: Computational performance of Motivo (black bars) and L8Motif (yellow bars), for $k = 8$. Missing bars represent the failure of Motivo by memory exhaustion. Note the drop in space usage of L8Motif which allows it to run even where Motivo fails.

We also measured the performance of L8Motif as a function of $n = |V(G)|$ and $m = |E(G)|$, by computing the average time *per million edges* and the average space *per node* on all our graphs, see Figure 12. We did this to show that L8Motif is predictable as a function of n and m. This sets it apart from most graphlet counting algorithms, whose running time varies chaotically. 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 [30]. The reason is that the algorithm enumerates all occurrences in G of certain "critical" subgraphs (for instance, cliques), whose number can vary wildly between graphs of comparable size. A similar chaotic behaviour is exhibited by random walks [9, 10], since their mixing time depends on the ratio between the maximum and minimum degree of G, raised to the power of $\Theta(k)$ [2].
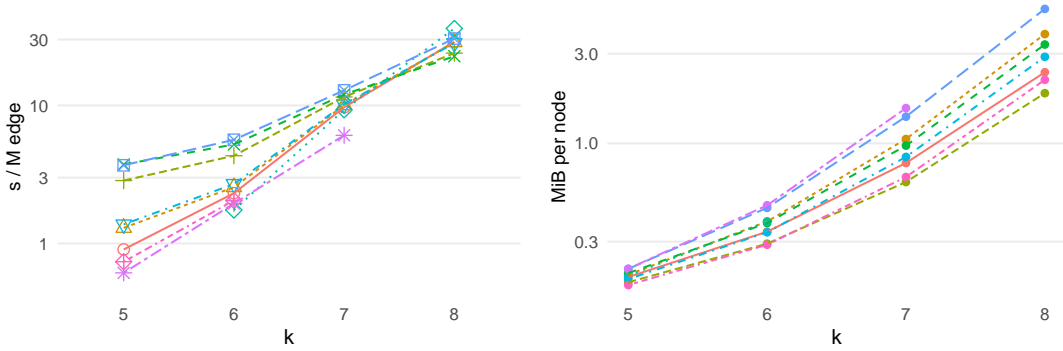

Figure 12: Build-up time in seconds per million edge, and space usage in bits per input node, on all our graphs, showing how L8Motif is predictable as a function of n, m, and k.

Comparison against CC. We compare Motivo and L8Motif against CC in Table 2 and Table 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 algorithms (recall that CC works in main memory). The sampling rate refers to uniform sampling, which is the only one supported by CC. The sampling speed of AGS is similar, and never more than 40% lower than uniform sampling. The only exception is the Yelp graph, on which AGS for $k = 8$ is $20\times$ slower than uniform sampling. But it is also much more accurate and still $10\,000\times$ faster than CC, as we show below.

graph	k	build-up time (seconds)			build-up space (GB)			sampling rate (motifs/sec)		
		CC	Motivo	speedup	CC	Motivo	reduction	CC	Motivo	speedup
Facebook	9	860	86	$10\times$	33	5	$7\times$	5	70k	$14\,000\times$
Facebook	8	95	17	$5.5\times$	24	1.4	$17\times$	419	149k	$356\times$
Dblp	9	1245	320	$3.9\times$	43	44	$1\times$	72	112k	$1\,556\times$
Dblp	8	182	82	$2.2\times$	30	27	$1\times$	679	186k	$274\times$
Amazon	9	376	84	$2.2\times$	49	51	$1\times$	226	87k	$385\times$
Amazon	8	140	87	$1.6\times$	33	31	$1\times$	1556	150k	$96\times$
BerkStan	5	14	7	$2\times$	18	0.5	$36\times$	160	6770	$42\times$
Yelp	5	167	71	$2.4\times$	36	4.5	$8\times$	20	910	$45\times$
LiveJournal	6	306	99	$3\times$	36	9.5	$4\times$	295	12k	$41\times$
Orkut	5	225	40	$5.6\times$	27	3.2	$8\times$	295	15k	$51\times$

Table 2: Computational performance of Motivo 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	$4.3\times$	24	0.2	$114\times$	420	23k	$55\times$
Dblp	8	182	77	$2.4\times$	30	1.7	$1.8\times$	680	41k	$60\times$
Amazon	8	140	84	$1.7\times$	31	1.8	$17\times$	1550	120k	$77\times$
BerkStan	5	14	7	$2\times$	18	0.2	$130\times$	160	1.2M	$7\,700\times$
Yelp	5	167	69	$2.4\times$	36	1.3	$27\times$	20	4.2M	$210\,000\times$
LiveJournal	6	306	79	$3.9\times$	36	1.8	$20\times$	295	112k	$380\times$
Orkut	5	225	38	$6\times$	27	0.7	$40\times$	295	162k	$550\times$

Table 3: Computational performance of L8Motif versus CC. Note that the maximum value of k tested is $k = 8$ as this is the largest value supported by L8Motif.

5.2 Accuracy of the estimates and performance of AGS

We evaluate the accuracy of the estimates produced by our algorithm L8Motif.

Uniform sampling. First, we consider uniform sampling. In this case, the accuracy of L8Motif and the accuracy of Motivo are identical, as they give the same output (only the sampling rate changes). We started by computing the ground-truth count of the number of copies of each possible k-graphlet in each graph. For $k = 5$ we used the exact algorithm ESCAPE [30] 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 stopped the sampling after 600s (10 minutes). To measure

the accuracy, we use the relative error. We 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:

$$\text{err}_H = \frac{\hat{c}_H - c_H}{c_H}. \tag{13}$$

Therefore a value $\text{err}_H \simeq 0$ means c_H has been accurately estimated; whereas $\text{err}_H \gg 0$ means c_H is overestimated, and $\text{err}_H \ll 0$ means c_H is underestimated (and $\text{err}_H = -1$ 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. The x-axis shows the value of err_H, and the y-axis the number of distinct graphlets H for which that value is achieved. Note that for YELP and AMAZON almost
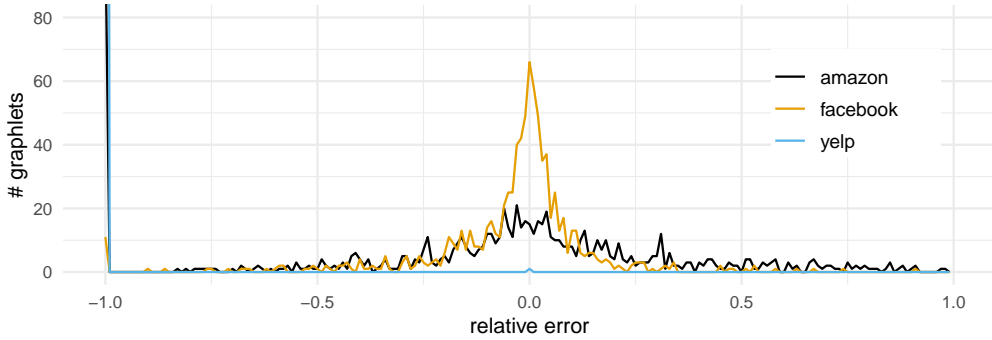

Figure 13: Relative error distribution of uniform sampling for $k = 7$. On YELP and AMAZON almost all graphlets have a relative error of -1, i.e., they are completely missed.

all graphlets have $\text{err}_H = -1$, as can be seen by the straight segments leaving in the leftmost part of the plot. This means uniform sampling misses almost all graphlets on AMAZON and YELP.

AGS. Figure 14 gives the relative error distribution for AGS. Note how the distribution is now concentrated around 0. This means that AGS gives an accurate estimate of nearly all graphlets, in line with our theoretical predictions.
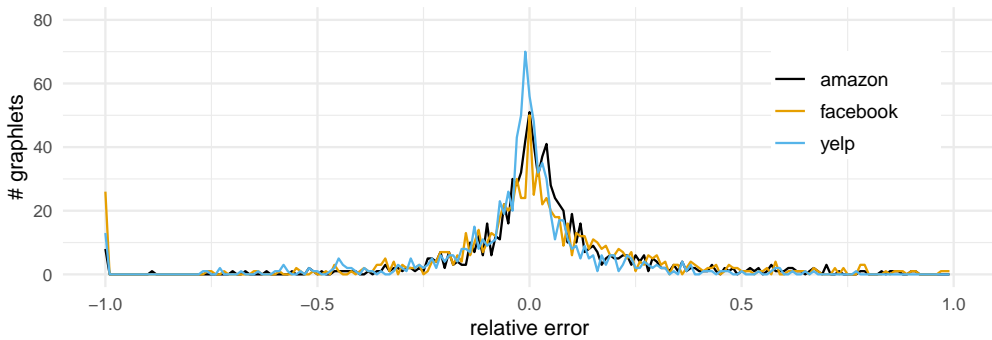

Figure 14: Relative error distribution of AGS for $k = 7$. Unlike the case of uniform sampling, here almost all graphlets are accurately estimated and have a relative error close to 0.

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. 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}. (These graphlet are well-estimated in all 10 runs, so they are not the product of chance). To find those graphlets, uniform sampling would need more than 10^3 years even if running at 10^9 samples per second. Therefore, we can say that AGS can count graphlets that uniform sampling cannot.
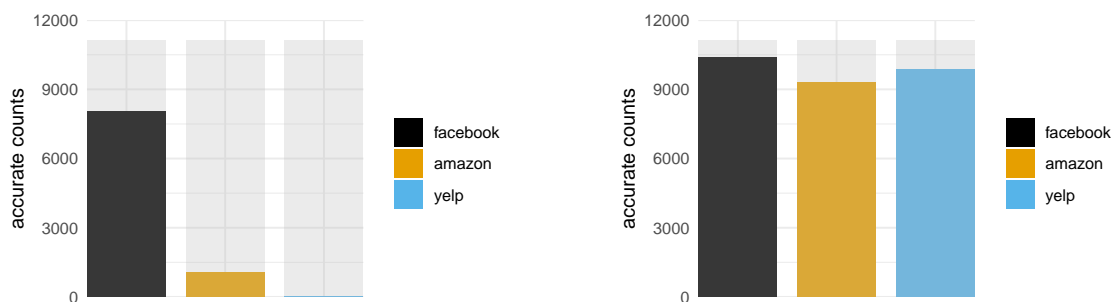

Figure 15: Number of 8-graphlets for which L8MOTIF achieved relative error below 25%. Left: uniform sampling. Right: AGS. The shaded area shows the total number of 8-graphlets, $N_8 = 11\,117$.

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] M. Agostini, M. Bressan, and S. Haddadan. Mixing time bounds for graphlet random walks. *Information Processing Letters*, 152:105851, 2019.

[3] 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.

[4] 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.

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

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

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

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

[20] 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.

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

[22] 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.

[23] S. Jain and C. Seshadhri. The power of pivoting for exact clique counting. In *Proc. of ACM WSDM*, pages 268–276, 2020.

[24] 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.

[25] 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.

[26] 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.

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

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

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

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

[31] 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.

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

[33] 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.

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

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

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] H. Zhang, J. X. Yu, Y. Zhang, K. Zhao, and H. Cheng. Distributed subgraph counting: A general approach. *Proc. VLDB Endow.*, 13(12):2493–2507, August 2020.

[43] 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.

[44] 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.