Counting HyperGraphlets via Color Coding: a Quadratic Barrier and How to Break It

Marco Bressan
University of Milan
Milan, Italy
marco.bressan@unimi.it

Stefano Clemente
University of Milan
Milan, Italy
stefano.clemente@unimi.it

Giacomo Fumagalli
University of Milan
Milan, Italy
giacomo.fumagalli@unimi.it

ABSTRACT

We study the problem of counting k-*hyper*graphlets, an interesting but surprisingly ignored primitive, with the aim of understanding if efficient algorithms exist. To this end we consider *color coding*, a well-known technique for approximately counting k-graphlets in graphs. Our first result is that, on hypergraphs, color coding encounters a *quadratic barrier*: under the Orthogonal Vector Conjecture, no implementation of it can run in time sub-quadratic in the size of the input. We then introduce a simple property, (α, β)-niceness, that hypergraphs from real-world datasets appear to satisfy for small values of α and β. Intuitively, an (α, β)-nice hypergraph can be split into two sub-hypergraphs having respectively rank at most α and degree at most β. By applying different techniques to each sub-hypergraph and carefully combining the outputs, we show how to run color coding in time $2^{O(k)} \cdot \left(2^{\beta}|V| + \alpha^k|E| + \alpha^2\beta\|H\|\right)$, where $H = (V, E)$ is the input hypergraph. Afterwards, we can sample colorful k-hypergraphlets uniformly in expected $k^{O(k)} \cdot (\beta^2 + \ln|V|)$ time per sample. Experiments on real-world hypergraphs show that our algorithm neatly outperforms the naive quadratic algorithm, sometimes by more than an order of magnitude.

PVLDB Reference Format:
Marco Bressan, Stefano Clemente, and Giacomo Fumagalli. Counting HyperGraphlets via Color Coding: a Quadratic Barrier and How to Break It. PVLDB, 19(9): 2073 - 2085, 2026.
doi:10.14778/3819518.3819535

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/gfumagalli9/hyper-motivo/tree/hyper-motivo.

1 INTRODUCTION

Hypergraphs, the higher-order generalization of graphs, are emerging as expressive and powerful models to represent complex interactions. Recent work suggests that many real-world phenomena that have been traditionally analysed using graphs should rather be analysed using hypergraphs, in particular by looking at the frequency of small induced substructures (called *hypergraphlets* or *hypermotifs*). For example, in collaboration networks, the classic phenomenon of triadic closure actually extends to its higher-order cousin, *simplicial closure*, the tendency of a group to spawn sub-groups as well [4, 19]. Similarly, [16] shows that the frequency vector of hypergraph motifs is correlated with the domain of the hypergraph, something that was previously known of graph motifs. Other examples where hypergraph motif statistics carry fundamental information come from biology [11], chemistry [2], social networks [12], and machine learning [3]. Clearly, in order to make all of this actionable, we need good *algorithms* to compute hypergraph motif statistics. Unfortunately, this is still an unsolved challenge. According to [11], already a decade ago we lacked "sophisticated algorithms for mining" hypergraph motifs, and the picture seems to not have changed since then. The goal of our work is to make a first step towards the development of such algorithms.

To discuss the matter, let us define the problem more formally. A hypergraph is a pair $H = (V, E)$ where V is a finite set and $E \subseteq 2^V \setminus \{\emptyset\}$. The sub-hypergraph of $H = (V, E)$ induced by $U \subseteq V$, denoted by $H[U]$, is the hypergraph with vertex set U and edge set $\{e \cap U : e \in E\} \setminus \{\emptyset\}$.[1] A k-hypergraphlet of H is an induced connected sub-hypergraph $H[U]$ of H with $|U| = k$. The *Hypergraphlet Counting* problem (HC for short) asks, given a hypergraph H and an integer $k \geq 2$, to compute[2] the number t_i of k-hypergraphlets of H that are isomorphic to H_i, where H_i for $i = 1, 2, \ldots$ ranges over all isomorphism-distinct k-vertex hypergraphs.

In the special case of *graphs*, HC boils down to Graphlet Counting (GC), the problem of counting connected k-vertex subgraphs.

For GC, the state-of-the-art algorithms are built on the celebrated color-coding technique of Alon, Yuster and Zwick [1]. This technique yields sampling and approximate graphlet counting algorithms that for every fixed k run in time linear in the input. More precisely, there is a preprocessing phase that takes time $O(\|H\|)$ where $\|H\| = |V| + \sum_{e \in E} |e|$, followed by random sampling in $O(\lg|V|)$-time per sample. Besides being elegant and clean, this approach yields formal statistical guarantees through concentration bounds. An optimized implementation, Motivo, can easily manage graphs with billions of edges [8]. Quite surprisingly, the picture for HC is completely different: we have basically no efficient algorithm, except for the brute-force ones that take time $|V|^{\Theta(k)}$ [15]. Our goal is to explore this gap, and answer the following question:

Can we devise algorithms for HC as good as those for GC?
If yes, how? If not, why?

Given the success of color coding for counting graphlets, it seems compelling to try adapting it to hypergraphlets. The *Gaifman* graph or *primal* graph $G = \mathrm{Gaif}(H)$ of H is the graph where u, v are adjacent if they share a hyperedge of H. It is easy to see that $H[U]$ is a

[1] There are other notions of induced sub-hypergraphs, but the corresponding problem is computationally even harder, see below.
[2] We actually allow for approximate counts (exact counts are computationally hard in a strong sense), but we avoid being too precise for now.

Proceedings of the VLDB Endowment, Vol. 19, No. 9 ISSN 2150-8097.
doi:10.14778/3819518.3819535

k-hypergraphlet if and only if $G[U]$ is a k-graphlet. This observation can be exploited as follows. First, compute $G = \text{Gaif}(H)$; we call this step *projection*.[3] Then, use an $O(\|G\|)$-time color-coding algorithm on G to sample random k-graphlets $G[U]$; each graphlet can then be converted into its corresponding k-hypergraphlet $H[U]$ in H. The same sample weighting techniques of Motivo then yield unbiased hypergraphlet count estimates with concentration guarantees. This looks great, but there is a big catch: the size of G can be *quadratic* in $\|H\|$. Indeed, every edge $e \in E(H)$ implies at least $\binom{|e|}{2}$ edges of G. Thus, while for graphs this approach yields an $O(\|H\|)$-time algorithm, for hypergraphs it only yields a much worse $O(\|H\|^2)$-time algorithm. Needless to say, this is prohibitive for large hypergraphs. Is this unavoidable, or can we do better?

Our contributions

1) A quadratic barrier. We show that, when the input is a hypergraph, the $\Omega(\|H\|^2)$ barrier above is unavoidable for the color-coding approach. More precisely we prove that, under the well-known Orthogonal Vectors Conjecture,[4] the output of color-coding's dynamic program cannot be computed in time $O(n^{2-\epsilon})$ for any $\epsilon > 0$. Thus, without further assumptions, color coding cannot yield linear-time algorithms for HC, as it does for GC. We also show that, under another classic definition of induced sub-hypergraph, even deciding if H contains *at least one* k-hypergraphlet is W[1]-hard and unlikely to be solvable time $|V|^{o(\sqrt[3]{k})}$. Thus, our definition is arguably the only one that possibly allows for efficient algorithms.

2) A new algorithm. We introduce a new hypergraph property, (α, β)-*niceness*. A hypergraph $H = (V, E)$ is (α, β)-nice if E admits a partition $E_{\leq\alpha}, E_{>\alpha}$ such that $(V, E_{\leq\alpha})$ has rank at most α and $(V, E_{>\alpha})$ has degree at most β.[5] All the hypergraphs used in our experiments appear to be (α, β)-nice for small values of α, β. We then give a two-phase algorithm, HyperMotivo, that preprocesses H in time

$$2^{O(k)} \cdot \left(2^{\beta} |V| + \alpha^k |E| + \alpha^2 \beta \|H\|\right)$$

where, again, $\|H\| = |V| + \sum_{e \in E} |e|$. Afterwards, HyperMotivo can draw samples in expected time $k^{O(k)} \cdot (\beta^2 + \log |V|)$ per sample. All the statistical guarantees provided by Motivo's color-coding approach are retained (e.g., the concentration of estimates). Each phase of HyperMotivo is based on a careful combination of several ingredients, each of which exploits the definition of (α, β)-niceness in a specific way in order to achieve the bounds above.

3) Experiments. We provide an efficient, multi-threaded C++ implementation of HyperMotivo, and we test it on six hypergraphs obtained from real-world data (plus a synthetic one). In particular, we compare HyperMotivo to the naive approach above that uses Motivo on $\text{Gaif}(H)$. We observe an improvement of several times in terms of time and memory usage, achieving a 30× on some hypergraphs. Thanks to it, we are able to provide estimates of the frequency distribution of the most frequent k-hypergraphlets, for $k = 3, 4, 5$, on all our hypergraphs.

[3] In the literature, this is also called *Clique Expansion (CE)*.

[4] To be precise we use the Moderate-Dimension Orthogonal Vector Conjecture, which says that there is no $O(n^{2-\epsilon} \text{ poly}(d))$-time algorithm for the following problem: given n vectors from $\{0,1\}^d$, decides if there are two of those vectors that are orthogonal.

[5] The degree is the maximum number of edges incident with a vertex, and the rank is the maximum size of an edge. See Section 2.

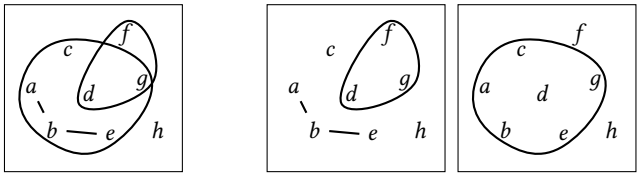


Figure 1: A toy hypergraph H (left) and its α-split into $H_{\leq\alpha}$ (right, first) and $H_{>\alpha}$ (right, second) for $\alpha = 4$. Our algorithm employs different techniques on the two sub-hypergraphs and combines the results carefully, both in the preprocessing and sampling phase.

2 PRELIMINARIES

A hypergraph is a pair $H = (V, E)$ where V is a finite set and $E \subseteq 2^V \setminus \{\emptyset\}$. The elements of V are called vertices and the elements of E are called edges. The *rank* of H is $r(H) = \max_{e \in E} |e|$. The *type* of a vertex $v \in V$ is the set of edges incident with it, $E(v) = \{e \in E : v \in e\}$, and its *degree* is $d(v) = |E(v)|$. We let $\Delta(H) = \max_{v \in V} d(v)$. Two vertices $u, v \in V$ are *adjacent* if $E(u) \cap E(v) \neq \emptyset$, that is, if they share an edge; we write $u \sim v$. The *neighborhood* of a vertex v is $N(v) = \{u \in V : u \sim v\}$. A *path* in H is a sequence $x_1 e_1 x_2 e_2 \dots x_s e_s x_{s+1}$ such that $x_1, \dots, x_{s+1} \in V$ are distinct, $e_1, \dots, e_s \in E$, and $x_i, x_{i+1} \in e_i$ for all $i = 1, \dots, s$. The *length* of the path is s, and we say that the path is a *u-v path*, or *a path from u to v*, where $u = x_1$ and $v = x_{s+1}$; we also say that u, v are *connected*. We say H is connected if all $u, v \in V$ are connected. The *Gaifman graph* (or *primal graph*) of a hypergraph H, denoted by $\text{Gaif}(H)$, is the simple undirected graph with $V(G) = V(H)$ and $E(G) = \{\{u, v\} \subseteq V \mid \exists e \in E, \{u, v\} \subseteq e\}$. In words, $u, v \in V(H)$ are adjacent in $\text{Gaif}(H)$ if and only if they share some edge in H.

Let $H = (V, E)$ and $H' = (V', E')$ be hypergraphs. We say H' is a *subhypergraph* of H if $V' \subseteq V$ and $e' \in E$ for every $e' \in E'$. We shall now present the notion of induced sub-hypergraph considered in this work.

Definition 2.1 (Induced sub-hypergraph)**.** *Let $H = (V, E)$ be a hypergraph and $U \subseteq V$. The sub-hypergraph induced by U in H, denoted $H[U]$, is the hypergraph with vertex set $V(H[U]) = U$ and edge set $E(H[U]) = \{e \cap U : e \in E, e \cap U \neq \emptyset\}$.*

A *k-hypergraphlet* of H is any induced connected sub-hypergraph of H on k vertices. Let:

$$\mathcal{V}_k(H) = \{U \subseteq V \,:\, H[U] \text{ is connected}\}. \tag{1}$$

The set of k-hypergraphlets of H is:

$$\mathcal{G}_k(H) = \{H[U] \,:\, U \in \mathcal{V}_k(H)\}. \tag{2}$$

For every connected hypergraph $H_1, H_2, \dots$ on k-vertices let:

$$\#\text{ind}(H, H_i) \triangleq |\{U \in \mathcal{V}_k(H) \,:\, H[U] \simeq H_i\}|\,. \tag{3}$$

where $\simeq$ denotes isomorphism. The main problem we consider in this work is:

Definition 2.2. *The* Hypergraphlet Counting *problem (HC) asks, given a hypergraph H and an integer k, to compute $\#\text{ind}(H, H_i)$ for every connected hypergraph $H_1, H_2, \dots$ on k vertices.*

It is well-known that the exact version of HC is computationally hard. For instance, it subsumes the problem of counting k-cliques, which under standard complexity-theoretical assumptions is not solvable in polynomial time and, in fact, not even in time $|V|^{o(k)}$ [9]. Thus, we consider a relaxed, approximate version of Definition 2.2, where one aims at estimates of the counts $\#\text{ind}(H, H_i)$ accurate within an additive $\pm\epsilon \cdot (\sum_i \#\text{ind}(H, H_i))$. This means that hypergraphlets with a high relative frequency are well approximated. This is the kind of guarantees given by state-of-the art algorithms for graphlet counting, such as Motivo [8].

The description of our algorithm borrows notation and concepts from [8]. A *k-treelet* is a tree on k vertices (we often talk about treelet of a graph G to mean a subgraph of G that is a tree on k vertices). We let $[k] = \{1, \ldots, k\}$. A *k-coloring* of a (hyper)graph with vertex set V is a mapping $c : V \to [k]$. A subset $U \subseteq V$ is *colored* with/by $S \subseteq [k]$ if $\{c(u) : u \in U\} = S$, and is *colorful* if it additionally satisfies $|U| = |S|$.

We assume the RAM model with words of size $\Theta(\log n + k)$. To ease our analysis, we also assume that indexing arrays/tables whose keys are tuples of $h \leq k$ vertices of H, or treelets on $h \leq k$ vertices, or subsets of $[k]$, takes constant time (in particular the counters $C(\cdot)$, see below). Note that this does not mean we treat k as a constant; in particular, our bounds state explicitly any exponential dependence on k.

Due to space limitations, proofs omitted from the main text are provided in the supplementary material.

2.1 On the definition of hypergraphlet

One may argue that the right definition of induced sub-hypergraph is the one where each edge is either taken integrally or not taken at all, and not truncated as per Definition 2.1. Such sub-hypergraphs are often referred to as *section hypergraphs*:

Definition 2.3. *Let $H = (V, E)$ be a hypergraph and $U \subseteq V$. The* section hypergraph *induced by U in H, denoted by $H\langle U\rangle$, is the hypergraph with $V(H\langle U\rangle) = U$ and $E(H\langle U\rangle) = \{e \in E : e \subseteq U\}$.*

Counting section sub-hypergraphs is a natural problem, and has been studied by [15]. Unfortunately, we show that counting section sub-hypergraphs is hard in a very strong sense. Let κ-SH denote the following problem: given a hypergraph $H = (V, E)$ and an integer $k \geq 2$, decide if H contains at least one k-vertex section sub-hypergraph that is connected—that is, $U \subseteq V$ with $|U| = k$ such that $H\langle U\rangle$ is connected. For graphs, this is equivalent to deciding whether there is a connected component on at least k vertices, and is thus in linear time. Somewhat surprisingly, we show:

Theorem 1. *Under the Exponential Time Hypothesis, κ-SH cannot be solved in time $n^{o(\sqrt[3]{k})}$.*

Thus, if one defines k-hypergraphlets via section sub-hypergraphs, then there is no hope for efficient counting algorithms, even approximate. Our definition of hypergraphlet is therefore, arguably, the most natural one that is algorithmically useful. The proof of Theorem 1 can be found in the supplementary material.

3 RELATED WORK

As said, we build on the color-coding technique [1] and the related algorithm of [7]. However, our techniques are substantially novel, although simple. Only a handful of other works seem related to our problem. [13] studies the problem of counting, for each possible Venn diagram on 3 sets, the number of hyperedges triplets in H that yield that Venn diagram (where every region of the diagram tells whether the corresponding intersection is empty or not). Thus, they do not count induced sub-hypergraphs, and not on a given number k of vertices. Moreover, their algorithm incurs an $\Omega(|E|^2)$ lower bound, as it precomputes a graph where two hyperedges of H are adjacent when they intersect. [15] gives algorithms for counting induced *section* sub-hypergraphs on k vertices, but only for $k = 3$ (which they do exactly) or $k = 4$ (approximately, via sampling). The basic idea is to first enumerate all hyperedges of size 3, computing the sub-hypergraph they induce. One then deletes all hyperedges of size at least 3, remaining with a graph, and counts exactly all its 3-graphlets through the ESU exhaustive enumeration algorithm [22]. Clearly, this requires time $\Omega(|V|^3)$ in the worst case. A few other results are known about the complexity of counting sub-hypergraphs exactly, but again under the notion of section sub-hypergraph, and with the goal of having a polynomial dependence on H, rather than a linear one [5, 17].

4 COLOR CODING AND ITS QUADRATIC BARRIER

v This section describes the color-coding approach and its inherent quadratic-time barrier. Roughly speaking, we show that the color-coding technique requires time $\Omega(|V| + \sum_{e \in E} |e|^2)$ on hypergraphs, unless a popular conjecture from computational complexity fails. The description of the color-coding technique is taken from [8], and we often use "Motivo" and "color-coding" interchangeably.

Let $G = (V, E)$ be a graph, and suppose we want to sample connected k-vertex subgraphs of G uniformly at random. The basic idea of color coding is to make the problem easier by coloring the vertices randomly. The technique has a *build-up phase* and a *sampling phase*, as follows.

The build-up phase. First, every vertex $v \in V$ is independently assigned a random uniform color $c_v \in \{1, \ldots, k\}$. Next, for every $h = 1, \ldots, k$, every rooted treelet T on h vertices, and every subset $S \subseteq \{1, \ldots, k\}$ with $|S| = h$, for every $v \in V$ we compute the number $C(T, S, v)$ of h-treelets of G that are isomorphic to T, colored by S, and rooted in v. This computation is performed using a dynamic programming approach. First, T is virtually split into two subtrees T_1, T_2 such that T_1 has the same root of T and T_2 is rooted at a child of that root. Then, the counter of T is computed according to the following equation:

$$C(T, S, v) = \frac{1}{d} \sum_{\substack{S_1, S_2 \subset S \\ S_1 \cap S_2 = \emptyset}} C(T_1, S_1, v) \sum_{u \sim v} C(T_2, S_2, u). \tag{4}$$

where d is a normalizing factor that depends only on T. It is not hard to prove that the whole build-up phase takes time $2^{O(k)} \cdot O(\|G\|)$.

The sampling phase. Using the counters C defined above, one can sample uniformly at random a k-treelet of G that is *colorful*—that is, such that every color $c \in \{1, \ldots, k\}$ appears in some vertex of the treelet. This is done through multi-stage sampling, as follows. First, pick a tree T on k vertex, as well as a vertex $v \in V$, with probability proportional to $C(T, S, v)$, the total number of colorful k-treelets of

G rooted in v that are isomorphic to T. Second, split T into T_1 and T_2 as above, and choose a partition S_1, S_2 of $S = [k]$ with probability proportional to $C(T_1, S_1, v) \cdot \sum_{u \sim v} C(T_2, S_2, u)$. Third, choose a neighbor u of v with probability proportional to $C(T_2, S_2, u)$. Finally, repeat the procedure recursively to sample a random copy of T_1 rooted at v with colors S_1, and a random copy of T_2 rooted at u with colors S_2. The union of the two copies yields the random colorful copy of T rooted at v. Once a colorful copy of T is sampled, one takes its vertex set U and stores the corresponding k-graphlet $G[U]$. Finally, as the probability of sampling $G[U]$ is proportional to the number of its spanning trees, one accepts $G[U]$ with probability *inversely* proportional to that number, and repeats the process otherwise. This makes the distribution of accepted graphlets uniform over all colorful k-graphlets of G. One can implement the sampling phase so that it runs in expected $2^{O(k)} \cdot O(\log n)$ time per sample.

The technique above can be extended to hypergraphs. The key observation is that, if $G = \mathrm{Gaif}(H)$, then $H[U]$ is connected *if and only if* $G[U]$ is connected. Thus, the k-hypergraphlets of H are precisely the k-graphlets of G (in terms of the vertex subsets inducing them). Hence, we may count the k-hypergraphlets of H by first computing G and then running color-coding on it (with the only catch of checking the k-hypergraphlet $H[U]$ rather than the k-graphlet $G[U]$).

Algorithm 1: The Naive Baseline

Input: hypergraph $H = (V, E)$, integer $k \geq 2$

1 compute $G = \mathrm{Gaif}(H)$;

2 run Motivo on G, k;

As noted in the introduction, Algorithm 1 in the worst case runs in time $\Omega(\|H\|^2)$, because in the worst case $\|\mathrm{Gaif}(H)\| = \Omega(\|H\|^2)$. One could try to bypass this obstacle by applying Equation (4) to H, rather than to $\mathrm{Gaif}(H)$. However, applying Equation (4) by listing all neighborhoods in H still takes time $\Omega(|V| + \sum_{e \in E} |e|^2)$, and it is not clear how this could be avoided. The next section shows that this is not an accident.

4.1 The quadratic barrier

We show that, unless the widely accepted Orthogonal Vector Conjecture fails, computing the counters given by Equation (4) actually *requires* time essentially $\Omega(|V| + \sum_{e \in E} |e|^2)$. The Orthogonal Vector problem (OV) asks, given a set A of n Boolean vectors of dimension d, whether there exist $u, v \in A$ orthogonal, i.e., such that $u[i] \cdot v[i] = 0$ for all $i \in \{1, ..., d\}$. The Strong Exponential Time Hypothesis (SETH) implies the following lower bound for OV, named *Moderate-dimension OVC*; see [10].

Conjecture 1 (*Moderate-dimension OVC*)**.** *There is no algorithm for OV running in time $O(n^{2-\varepsilon} poly(d))$ for any $\varepsilon > 0$.*

Our result is:

Theorem 2 (Quadratic barrier of color coding on hypergraphs)**.** *Assume the moderate-dimension OVC (Conjecture 1). Then, for every $k \geq 2$ and $\epsilon > 0$, no algorithm can compute the counters of Equation (4) on a hypergraph $H = (V, E)$ in time $O(|V| + \sum_{e \in E} |e|^{2-\epsilon})$.*

As computing the counters of Equation (4) is the very heart of the color-coding technique, Theorem 2 says that color coding alone is unlikely to give
a sub-quadratic algorithm for Hypergraphlet Counting. Note also that Theorem 2 still agrees with the fact that, *on graphs*, color-coding yields linear-time algorithms—indeed, for a graph we have $\sum_{e \in E} |e|^2 = 4|E|$.

The rest of this section gives the proof of Theorem 2; the uninterested reader may skip it.

4.1.1 Proof of Theorem 2. The proof goes through an intermediate problem, the *Neighbor Count* problem (NC), which, given in input a hypergraph $H = (V, E)$, asks to compute the cardinality $|N(v)| = |\{u \in V : u \sim v\}|$ of each vertex $v \in V$. We show that OVC reduces efficiently to NC, then we show that NC reduces efficiently to the problem of computing the counters of Equation (4).

Lemma 1. *OV can be solved in time $O(dn) + T(n, d)$, where $T(n, d)$ is the complexity of NC on hypergraphs with n vertices and d edges.*

Proof. Let $\mathbf{v}_1, \ldots, \mathbf{v}_n \in \mathbb{R}^d$ be the input of OV. Consider the hypergraph $H = (V, E)$ with $V = \{\mathbf{v}_1, \ldots, \mathbf{v}_n\}$ and $E = \{e_1, \ldots, e_d\}$, where $e_\ell = \{\mathbf{v}_i : \mathbf{v}_i[\ell] = 1\}$ for $\ell = 1, \ldots, d$; note that H can be constructed in time $O(nd)$. Now, observe that $\mathbf{v}_i \cdot \mathbf{v}_j = 0$ if and only if $E_{\mathbf{v}_i} \cap E_{\mathbf{v}_j} = \emptyset$, that is, if $\mathbf{v}_i \nsim \mathbf{v}_j$. Thus, the answer to OV is YES if and only if H contains two vertices that are not adjacent, that is, if and only if $|N(v)| < n - 1$ for some $v \in V$. This can be checked in time $O(|V|) = O(dn)$ from the solution to NC. We conclude that OV can be solved in time $O(dn) + T(n, d)$, as claimed. □

Lemma 2. *For every fixed $k \geq 2$, NC can be solved in time $O(\|H\|) + R(k|V|, |E|)$, where $R(kn, d)$ is the complexity of computing the counters of Equation (4) on a hypergraph with kn vertices and d edges.*

Proof. Let $H = (V, E)$ be the input of NC. We construct the hypergraph $H' = (V', E')$ where $V' = V \times [k]$ and E' contains, for every $e \in E$, the hyperedge $e' = \{(u, c) : u \in e,\ c \in [k]\}$ to $E(H')$. We also construct the coloring $c : V' \to [k]$ given by the indices c of the vertices $(v, c) \in V'$. As k is fixed, both H and the coloring c can be constructed in time $O(\|H\|)$. Note also that $|V'| = k|V|$ and $|E'| = |E|$.

Suppose now we have computed the counters of H' in time $R(k|V|, |E|)$. Let T be the k-star (i.e., the star on $k - 1$ leaves, $K_{1,k-1}$) and let $S = [k]$. For every $v' = (v, c) \in V'$, by definition of the counters, and by a simple counting argument, we have:

$$C(K_{1,k-1}, [k], v') = (|N(v)|)^{k-1},$$

where $N(v)$ is the neighborhood of v in H. Therefore $|N(v)| = \left(C(K_{1,k-1}, [k], v')\right)^{\frac{1}{k-1}}$ for every $v \in V$ and every $v' = (v, c) \in V'$. Hence, given the counters of H', one can compute the solution to NC on H in time $O(|V|) = O(\|H\|)$. This completes the proof. □

Wrap-up. We conclude the proof of Theorem 2. Suppose that, for some $k \geq 2$ and $\epsilon > 0$, we can compute the counters of Equation (4) in time $O(|V| + \sum_{e \in E} |e|^{2-\varepsilon})$. This implies:

$$R(kn, d) = O\left(n + \sum_{e \in E} |e|^{2-\epsilon}\right) = O\left(dn^{2-\epsilon}\right)$$

Moreover, by combining the two lemmas above, OV has complexity:

$$O(dn) + T(n,d) = O(dn) + (O(nd) + R(kn,d)) = O(dn) + R(kn,d)$$

We conclude that OV has complexity $O(dn^{2-\epsilon})$, contradicting Conjecture 1.

5 THE (α,β)-NICENESS OF A HYPERGRAPH

Section 4 shows that, without further assumptions, there is no hope to obtain $O(\|H\|^2)$-time algorithms for HC via color coding. Thus, we shall look for additional properties of H that can make color coding easier. One such property is having small rank, say $\text{rank}(H) \leq 10$. This obviously implies $\sum_{e\in E} |e|^2 = O(|E|)$, making the naive algorithm (Algorithm 1) run in time linear in $\|H\|$. However, this assumption is quite strong, and real-world hypergraphs seem far from it—see Table 1. Another such property could be having small degree, say $\Delta(H) \leq 10$. It is not immediately clear how this could accelerate color coding; but, again, Table 1 suggests that in practice this property often does not hold.

We bypass these limitations by introducing (α,β)-*niceness*, an "interpolation" between small rank and small degree. On the one hand, hypergraphs from real-world datasets are (α,β)-nice for small values of α,β; on the other hand, we show how to exploit (α,β)-niceness to bring color coding into near-linear time.

Definition 5.1. *Let $H = (V,E)$ be a hypergraph and let $\alpha \geq 0$. The α-*split *of H is the pair of hypergraphs $(H_{\leq\alpha}, H_{>\alpha})$ where $H_{\leq\alpha} = (V, E_{\leq\alpha})$ and $H_{>\alpha} = (V, E_{>\alpha})$ satisfy:*

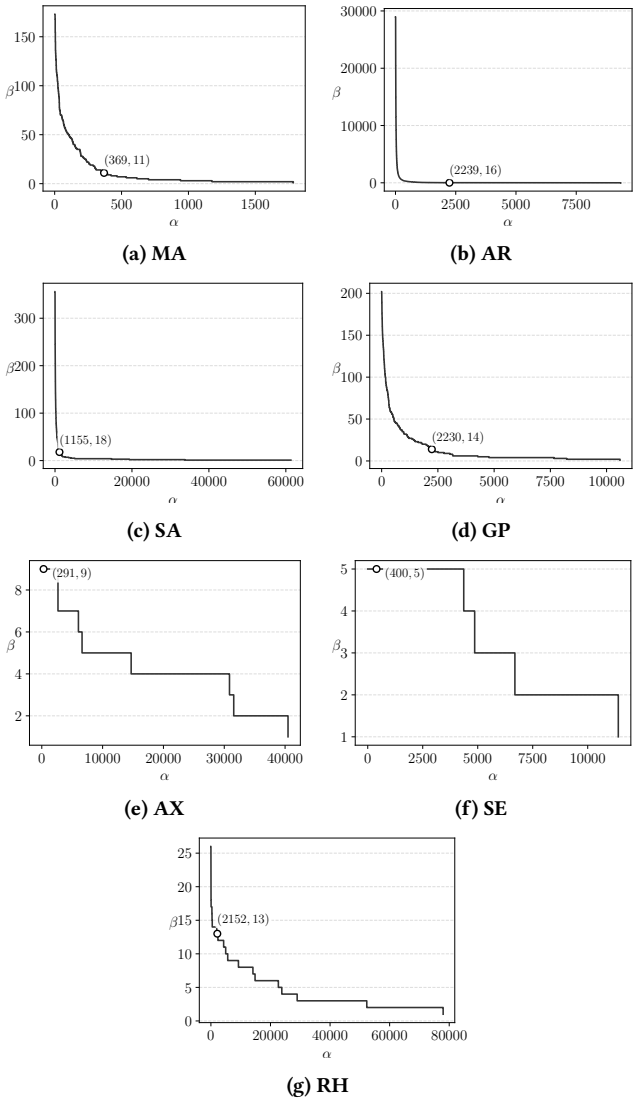
$$\begin{aligned} E_{\leq\alpha} &= \{e \in H : |e| \leq \alpha\} \\ E_{>\alpha} &= \{e \in H : |e| > \alpha\} \end{aligned} \tag{5}$$

A hypergraph $H = (V,E)$ is (α,β)-nice if $\Delta(H_{>\alpha}) \leq \beta$.

In other words, H is (α,β)-nice if it can be edge-partitioned into a hypergraph $H_{\leq\alpha}$ with rank at most α and a hypergraph $H_{>\alpha}$ of degree at most β. We call $H_{\leq\alpha}$ the *lower part* and $H_{>\alpha}$ the *upper part* of the split. Furthermore, we denote by $N_{\leq\alpha}(v)$ and $N_{>\alpha}(v)$ the set of neighbors of v in their respective parts. The same convention is used for the type $E(v)$ as well.

Note that our algorithm HyperMotivo has a build-up time of $f(\alpha,\beta,k)\cdot O(\|H\|)$, see Theorem 3, and is thus efficient if α,β,k are small. Figure 2 shows that this is indeed the case in practice. For every $\alpha = 0,1,\ldots$, the figure shows the smallest value of β for which a hypergraph is (α,β)-nice, for all hypergraphs used in our experiments. Note that, in each curve, the rightmost point corresponds to $\alpha = \text{rank}(H)$ and $\beta = 0$, while the topmost point to $\beta = \Delta(H)$ and $\alpha = 0$. These are trivial values, because obviously any H is both $(\text{rank}(H),0)$-nice and $(0,\Delta(H))$-nice. The interesting point of the curves is that real-world hypergraphs are often (α,β)-nice for α,β *orders of magnitude smaller* than $\text{rank}(H),\Delta(H)$. This seems to be consistent with the general idea that, in social networks, individuals typically participate in a large number of small communities, but only in a few (if any) large ones [14]. Thus, (α,β)-niceness is a good candidate for a parameterization of an algorithm's running time. The dot across each curve denotes the pair (α,β) chosen by HyperMotivo, which is typically with $\alpha \simeq 10^3$ and $\beta \simeq 10$.

The next section shows how to leverage (α,β)-niceness to break through the quadratic barrier of color coding on hypergraphs.



(a) MA **(b) AR** **(c) SA** **(d) GP** **(e) AX** **(f) SE** **(g) RH**

Figure 2: The (α,β)-curve of our hypergraphs. For each $\alpha = 0,\ldots,\text{rank}(H)$, the curve shows the smallest of β such that the hypergraph is (α,β)-nice. The dot marks the point (α,β) chosen by HyperMotivo. See Section 7.1 for more details.

6 HYPERMOTIVO

We describe HyperMotivo, our algorithm for sampling and approximately counting k-hypergraphlets through color coding. The main idea behind HyperMotivo is to exploit the (α,β)-niceness of a hypergraph (Section 5) to bypass the quadratic barrier of the naive color-coding algorithm (Section 4). Our main result is:

Theorem 3. *There exists a two-phase algorithm, HyperMotivo, with the following guarantees. Given in input an integer $k \geq 2$, a value $\alpha \geq 0$, and an (α,β)-nice hypergraph $H = (V,E)$,*

- *the* build-up phase *takes time*

$$2^{O(k)} \cdot O\left(2^{\beta}|V| + \alpha^k|E| + \alpha^2\beta\|H\|\right);$$

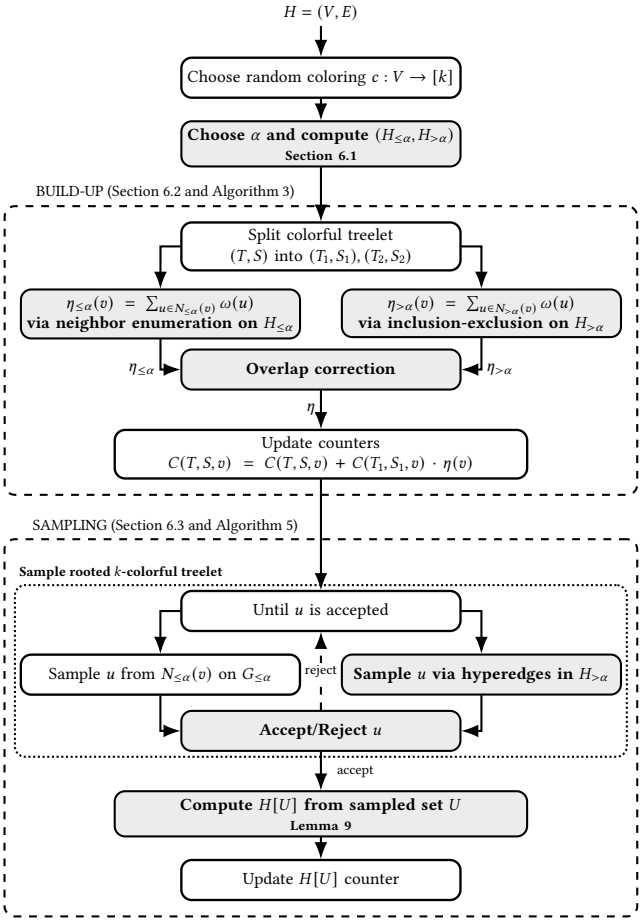


Figure 3: Structure of HyperMotivo. White blocks are in common with Motivo; gray ones are introduced in this work.

- *the* sampling phase *takes, on each invocation, expected time*

$$k^{O(k)} \cdot (\beta^2 + \ln |V|)$$

and yields a colorful k-hypergraphlet $H[U]$ uniformly at random.

A high-level overview of the algorithm is shown in Figure 3. Note that, instead of sampling $H[U]$ uniformly at random, one can sample it with probability proportional to the number of its spanning trees $\sigma(H[U])$, while also computing $\sigma(H[U])$ itself. The $k^{O(k)}$ factor in the sampling time decreases to $2^{O(k)}$, and one can estimate the hypergraphlets counts in the same way as Motivo, by adding $\frac{1}{\sigma(H[U])}$ to the count of hypergraphlets isomorphic to $H[U]$. The same statistical guarantees of Motivo apply, including concentration bounds that depend on the number of k-hypergraphlets of H; see [6]. Note that, in our implementation, we avoid the $\alpha^k|E|$ part of the build-up phase. In theory, this makes the sampling phase slower; in practice, it is still almost as fast as Motivo's. See Section 7. Finally, let us observe that Theorem 3 is obtained by a very careful combination of several parts—from computing counters, to sampling neighbors, to computing $H[U]$—each of which exploits (α, β)-niceness in a very specific way. We find this quite interesting.

The remainder of the section is organized as follows. Section 6.1 describes how HyperMotivo computes an α-split of the input hypergraph. Section 6.2 describes the build-up phase of HyperMotivo, and Section 6.3 describes its sampling phase.

6.1 Splitting the hypergraph

This section discusses the first step of HyperMotivo: given the hypergraph H, compute an α-split $(H_{\leq\alpha}, H_{>\alpha})$. In fact, we give a $O(\|H\|)$-time algorithm that finds a "good" value of α, computes the corresponding split $(H_{\leq\alpha}, H_{>\alpha})$, and moreover provides an asymptotic estimate of the number of operations performed by our algorithm when given in input $(H_{\leq\alpha}, H_{>\alpha})$. In section 7 we also show that this choice of α is rather robust to variations in α itself (and, therefore, our method to choose α does not "overfit").

It is straightforward that, given α, one can compute $(H_{\leq\alpha}, H_{>\alpha})$ in time $O(\|H\|)$ by simply placing each edge e of H in $H_{\leq\alpha}$ if $|e| \leq \alpha$ and in $H_{>\alpha}$ otherwise. We shall thus focus on the choice of α. Roughly speaking, we seek to choose α so as to minimize the running time of HyperMotivo's build-up phase. That running time is dominated by the computation of the counters $C(\cdot, \cdot, \cdot)$, which is carried out independently, and by two distinct algorithms, on $H_{\leq\alpha}$ and on $H_{>\alpha}$. As explained later in Section 6.2, the times $T_{\leq\alpha}$ and $T_{>\alpha}$ spent by the two algorithms can be bound as follows (we use $x \lesssim y$ to denote $x = O(y)$):

$$T_{\leq\alpha} \lesssim \sum_{e \in E_{\leq\alpha}} |e|^2 \tag{6}$$

$$T_{>\alpha} \lesssim \sum_{v \in V} 2^{d_{>\alpha}(v)} \tag{7}$$

If H is (α, β)-nice, it follows immediately that:

$$T_{\leq\alpha} + T_{>\alpha} \lesssim \alpha^2|E| + 2^\beta|V| \tag{8}$$

Therefore, one sensible way to choose α is to (1) compute all pairs (α, β) such that H is (α, β)-nice (i.e., the curve shown in Section 5), and (2) choose the pair that minimizes the right-hand side of Equation (8). To this end we proceed as follows. First, we sort the edges of H in nondecreasing order of size, $e_1, \ldots, e_{|E|}$. We also let $\alpha_0 = 0$ and $\beta_0 = \Delta$, where Δ is the maximum degree of H. Clearly, H is (α_0, β_0)-nice. Then, for $i = 1, \ldots, |E|$, we let $\alpha_i = |e_i|$, and we let β_i be the maximum degree of $H_{>\alpha_i}$. Note that, to compute β_i, it is sufficient to keep track of the degrees in $H_{>\alpha_i}$, and take their maximum through a max-heap. To update those degrees from $H_{>\alpha_{i-1}}$ to $H_{>\alpha_i}$, one simply decreases by 1 the degree of every vertex in e_i. This yields an update time of $O(|e_i| \log |V|)$ for computing α_i, β_i from $\alpha_{i-1}, \beta_{i-1}$. Once we have the complete curve $(\alpha_i, \beta_i)_{i=0,\ldots,|E|}$, it is straightforward to choose i so that (α_i, β_i) minimizes the right-hand side of Equation (8). Overall, we have proved:

Lemma 3. *Given H, in time $O(\|H\| \log |V|)$ one can compute a pair (α, β), and the corresponding α-split $(H_{\leq\alpha}, H_{>\alpha})$, such that H is (α, β)-nice and the right-hand side of Equation* (8) *is minimized.*

The simple algorithm behind Lemma 3 can actually be refined in order to compute a pair (α, β) that minimizes the more accurate estimate of the running time:

$$T_{\leq\alpha} + T_{>\alpha} \lesssim \sum_{e \in E_{\leq\alpha}} |e|^2 + \sum_{v \in V} 2^{d_{>\alpha}(v)} \tag{9}$$

Indeed, we can prove:

Lemma 4. *Given H, in time $O(\|H\|)$ one can compute a pair (α, β), and the corresponding α-split $(H_{\leq\alpha}, H_{>\alpha})$, such that H is (α, β)-nice and the right-hand side of Equation (9) is minimized.*

The algorithm of Lemma 4 is the one actually used by HyperMotivo. The rest of this section describes this algorithm, proving Lemma 4. At a high level, the procedure evaluates all candidate thresholds s_i by sweeping them from larger to smaller values and maintaining on the fly the quantities appearing in the right-hand side of Equation (9). After a linear-time preprocessing phase, each new threshold is handled by a constant amount of work, so that the best (α, β)-split can be found in overall $O(\|H\|)$ time.

We begin by constructing, for every vertex $v \in V$, the list I_v of hyperedges incident with v, sorted in non-increasing order of hyperedge size. Thus $I_v[0]$ is the largest hyperedge incident with v, $I_v[1]$ is the second largest, and so on. For each vertex–hyperedge incidence (v, e), we let $s = |e|$ be the size of e and let p be the position of e in I_v (with $p = 0$ for $I_v[0]$). We then create a triple (v, s, p) for every incidence and collect all such triples into a single list S. Next, we sort S by non-increasing s and, for equal s, by non-increasing p, and process them in this order. Let $s_1 < s_2 < \cdots < s_m$ be the distinct hyperedge sizes appearing in H. For each $i \in [m]$, we denote by S_i the (maximal) contiguous sublist of S consisting of all triples (v, s, p) with $s = s_i$. Since S is sorted by non-increasing s, these blocks appear in S in the order $S_m, \ldots, S_1$. We now show how, by sweeping through all the blocks S_i in this order, we can compute the right-hand side of Equation (9) for every candidate threshold s_i and its corresponding s_i-split $(H_{\leq s_i}, H_{>s_i})$.

We initialize the procedure at the largest candidate value, s_m. In this configuration, all hyperedges belong to the lower part, so $T_{\leq s_m} = \sum_{e \in E} |e|^2$, while $T_{>s_m} = 0$. Moreover, we precompute, for each $i \in [m]$, the number $c_i = |\{e \in E : |e| = s_i\}|$. Now, by iterating on $S_m, \ldots, S_1$ in order, it is easy to see that after processing block S_m we are at threshold s_m, and for every $i = m-1, \ldots, 1$ we can compute $T_{\leq s_i}$ from $T_{\leq s_{i+1}}$ as $T_{\leq s_i} = T_{\leq s_{i+1}} - c_i \cdot s_i^2$. For the upper part, fix $i \in [m]$ and $v \in V$. Among all incidences (v, e) with $|e| = s_i$, let $p_i(v)$ be the maximum position of such an edge in I_v. The key observation is that, since I_v is sorted by non-increasing edge size, the entries $I_v[0], \ldots, I_v[p_i(v)]$ are exactly the hyperedges incident to v that belong to the upper part at threshold s_i, and therefore $d_{>s_i}(v) = p_i(v) + 1$. Within block S_i, the triples (v, s_i, p) corresponding to a fixed vertex v appear in non-increasing order of p, and hence the first such triple we encounter has $p = p_i(v)$. Thus, to update $T_{>s_i}$ we just need to maintain for each vertex v its current degree. When we process the first triple (v, s_i, p) for vertex v in S_i, we set its degree to $p + 1$ and update its contribution in the sum accordingly.

At the end of this process, we have computed the right-hand side of Equation (9) for every candidate threshold s_i. It is trivial to see that we can also keep track of the s_i that minimizes the equation. At the same time, for each s_i we maintain the maximum degree $\beta(s_i) = \max_{v \in V} d_{>s_i}(v) = \max_{v \in V} p_i(v) + 1$. Let $\alpha^\star$ be the threshold chosen by this procedure, and let $\beta^\star = \beta(\alpha^\star)$. By construction, the corresponding split $(H_{\leq\alpha^\star}, H_{>\alpha^\star})$ satisfies $\Delta(H_{>\alpha^\star}) \leq \beta^\star$, i.e., H is $(\alpha^\star, \beta^\star)$-nice, and the quantity in Equation (9) is minimized at $(\alpha^\star, \beta^\star)$. Since sorting S takes $O(\|H\|)$ time via counting sort, and the sweep as well, this proves Lemma 4.

6.2 The build-up phase

The role of the build-up phase is, given H and its coloring, to compute the counters $C(T, S, v)$ of Equation (4). By Theorem 2, there is little hope of doing so in sub-quadratic time *as a function solely of* $|V|$ and $|e|$ for $e \in E$. The goal of this section is to show how, if H is (α, β)-nice, then we can perform the build-up phase in time near-linear in $\|H\|$, times a function of α, β. Let $(H_{\leq\alpha}, H_{>\alpha})$ be the α-split of H; see Section 6.1. The idea is to compute counters for $H_{\leq\alpha}$ and $H_{>\alpha}$ separately, and postprocess them in order to get the counters of H. To this end, observe how, by Equation (4), $C(T, S, v)$ can be written as a sum $S_1, S_2, \ldots$ of terms in the form:

$$C(T_1, S_1, v) \cdot C(T_2, S_2, N(v)) \tag{10}$$

where $N(v)$ is the set of vertices adjacent to v in H, and

$$C(T_2, S_2, N(v)) = \sum_{u \in N(v)} C(T_2, S_2, u)\,. \tag{11}$$

It is immediate to see that, if one can compute $C(T_2, S_2, N(v))$ in time t, then one can compute all counters $C(T, S, v)$ in time $2^{O(k)} \cdot t$.

By abstracting details away, one reduces to the following problem. A *weighted hypergraph* is a hypergraph $H = (V, E)$ with a vertex weighting $w : V \to \mathbb{N}_0$. Extend w to subsets of V in the natural way. The *neighbor weight* of a vertex v is $\eta(v) = w(N(v))$. The *Neighbor Weight Problem* (NW) asks, given (H, w), to compute $(\eta(v))_{v \in V}$. Clearly, computing $C(T_2, S_2, N(v))$ for all $v \in V$ reduces, in time $O(\|H\|)$, to NW on (H, w) with weights $w(v) = C(T_2, S_2, v)$. To prove the first item of Theorem 3, we shall thus prove how one can solve NW in the time claimed by that item.

6.2.1 Solving NW on (α, β)-nice hypergraphs. Let (H, w) be a weighted hypergraph, let $(H_{\leq\alpha}, H_{>\alpha})$ be the α-split of H, and suppose H is (α, β)-nice. We shall adopt the following strategy. First, we solve NW on the weighted hypergraph $(H_{\leq\alpha}, w)$ and obtain the neighbor weights $\eta_{\leq\alpha}(v)$. Second, we solve NW on the weighted hypergraph $(H_{>\alpha}, w)$ and obtain the neighbor weights $\eta_{>\alpha}(v)$. Third, for each vertex we sum the counters and obtain

$$\tilde{\eta}(v) = \eta_{\leq\alpha}(v) + \eta_{>\alpha}(v) = \sum_{u \in N_{\leq\alpha}(v)} w(u) + \sum_{u \in N_{>\alpha}(v)} w(u) \tag{12}$$

Clearly we may have $\eta(v) < \tilde{\eta}(v)$, since a vertex $u \in N(v)$ can appear in both $N_{\leq\alpha}(v)$ and $N_{>\alpha}(v)$. The third step thus entails subtracting $w(u)$ from $\tilde{\eta}(v)$ for all those u to retrieve $\eta(v)$. The rest of the subsection describes the three steps above.

6.2.2 Solving NW on $H_{\leq\alpha}$. For $H_{\leq\alpha}$, we simply compute $G_{\leq\alpha} = \mathrm{Gaif}(H_{\leq\alpha})$, and then compute $\eta_{\leq\alpha}(v)$ by explicitly enumerating the neighborhood $N_{\leq\alpha}(v)$ of v in $G_{\leq\alpha}$. As $\mathrm{rank}(H_{\leq\alpha}) \leq \alpha$, computing $G_{\leq\alpha}$ takes time $O(|V| + \alpha^2|E_{\leq\alpha}|) \leq O(|V| + \alpha^2|E|)$. Enumerating the neighborhoods obviously requires linear time. This gives a total time of $O(|V| + \alpha^2|E|)$.

6.2.3 Solving NW on $H_{>\alpha}$. To ease the notation we describe the algorithm on a generic weighted hypergraph (H, w). The first observation is that, obviously, every $v \in V$ satisfies:

$$\eta(v) = w\Big(\bigcup E(v)\Big) - w(v)\,. \tag{13}$$

Hence, solving NW reduces to computing $w(\bigcup E(v))$ for each v. As $|E(v)| \leq \beta$, we resort to compute $w(\bigcup E(v))$ via *inclusion-exclusion*, as follows. Let $E(v) = \{e_1, \ldots, e_\ell\}$. Then, the inclusion-exclusion principle says that:

$$w\Big(\bigcup E(v)\Big) = w\Big(\bigcup_{i\in[\ell]} e_i\Big) = \sum_{\emptyset\neq I\subseteq[\ell]} (-1)^{|I|+1}\, w\Big(\bigcap_{i\in I} e_i\Big). \quad (14)$$

Thus, it suffices to (1) compute $w\left(\bigcap_{i\in I} e_i\right)$ for all $I \subseteq [\ell]$ for all $E(v)$, and (2) apply Equation (14). This is what Algorithm 2 does.

Algorithm 2: NW-IE

Input: A weighted hypergraph $(H = (V, E), w)$
Output: The vector η of neighbor weights

```
t ← empty dictionary with default value 0;
for each v ∈ V do
    for each X ⊆ E(v) do
        t[X] ← t[X] + w(v) ;
η ← empty dictionary with default value 0 ;
for each v ∈ V do
    for each ∅ ≠ X ⊆ E(v) do
        η(v) ← η(v) + (−1)^{|X|+1} t[X];
    η(v) ← η(v) − w(v);
return η;
```

We prove:

Lemma 5. *NW-IE solves NW in time* $O(2^{\Delta(H)} \cdot |V|)$.

To conclude, since $\Delta(H_{>\alpha}) \leq \beta$, applying NW-IE to $(H_{>\alpha}, w)$ yields the counters $\eta_{>\alpha}(v)$ in time $O(2^\beta \cdot |V|)$.

6.2.4 Correcting $\tilde{\eta}(v)$. The third and final step in the build-up phase is retrieving the correct neighbor weights η from $\eta_{\leq\alpha}$ and $\eta_{>\alpha}$. To this end, for each $v \in V$ we iterate over $u \in N_{\leq\alpha}(v)$ and check if $u \in N_{>\alpha}(v)$ too. If this is the case, then we subtract $w(u)$ from $\tilde{\eta}(v)$. Iterating over v and u takes time $O(\alpha^2|E|)$ by using $\mathrm{Gaif}(H_{\leq\alpha})$. For each such iteration, checking $u \in N_{>\alpha}(v)$ takes time $O(\beta)$ by intersecting the list of edges (i.e., the types) of u and v in $H_{>\alpha}$. This requires those types to be sorted, which one can do beforehand in time:

$$O\left(\sum_{v\in V} d_{>\alpha}(v) \ln d_{>\alpha}(v)\right) = O\left(\|H_{>\alpha}\| \ln \beta\right) = O\left(\|H\|\beta\right). \quad (15)$$

Hence, computing η from $\eta_{\leq\alpha}$ and $\eta_{>\alpha}$ takes total time $O(\alpha^2\beta\|H\|)$.

6.2.5 Wrapping up. Algorithm 3 gives the pseudocode for computing the counters $C(\cdot)$. Note that this is not *all* the build-up phase: in order to have an efficient sampling phase, we need some additional preprocessing, detailed in Section 6.3. In particular, the $O(\alpha^k|E|)$ term appearing in the bounds of Theorem 3 is due to Lemma 9. The other two terms are instead due to Algorithm 3:

Lemma 6. *If H is (α, β)−nice, Algorithm 3 computes the counters of Equation* (4) *in time* $2^{O(k)} \cdot \left(2^\beta|V| + \alpha^2\beta\|H\|\right)$.

Algorithm 3: HyperMotivo build-up

Input: A hypergraph $H = (V, E)$ and $k \geq 2$
Output: The counters $C(T, S, v)$ of Equation (4)

```
C ← empty dictionary with default value 0;
for each v ∈ V do
    c(v) ← random color from [k];
    C(T, {c(v)}, v) ← 1 where T is the trivial tree on one vertex;
compute the α-split (H_{≤α}, H_{>α}) of H as per Lemma 4;
compute G_{≤α} = Gaif(H_{≤α});
sort the types of each v ∈ V in H_{>α};
for every h = 2, ..., k, every h-treelet T, and every S ∈ binom([k], h) do
    let T_1, T_2 be the canonical decomposition of T;
    for each partition S_1, S_2 of S with |S_1| = |T_1| do
        let w : V → ℕ be given by w(v) = C(T_2, S_2, v);
        η_{≤α} ← NW-Naive(H_{≤α}, w);
        η_{>α} ← NW-IE(H_{>α}, w);
        for each v ∈ V do
            η(v) ← η_{≤α}(v) + η_{>α}(v);
            for each u ∈ N_{≤α}(v) do
                if u ∈ N_{>α}(v) then
                    η(v) ← η(v) − w(u);
            C(T, S, v) ← C(T, S, v) + d^{-1} C(T_1, S_1, v) · η(v);
return C
```

6.3 The sampling phase

The goal of the sampling phase is to sample colorful, uniform k-hypergraphlets of H. To this end, we follow the same high-level strategy of Motivo. To begin with, we need an efficient *neighbor-sampling* routine. Given a vertex v, a set of colors $S_2 \subseteq [k]$, and a treelet T_2 with $|T_2| = |S_2|$, we need to draw $u \in N(v)$ with probability proportional to $C(T_2, S_2, u)$ (see Section 4), while avoiding the explicit visit of $N(v)$. Once we have this routine, we can then implement an efficient procedure for sampling k-colorful rooted treelets uniformly at random from H. At this point, we can use the sampled k-colorful rooted treelets to sample k-hypergraphlets, by following standard rejection sampling techniques. While at a high level this may look easy, the nontrivial challenge is to make all these routines run in time nearly independent of $\|H\|$. Somewhat surprisingly, we will show how to do so by repeatedly exploiting the (α, β)-niceness of H.

For the remainder of the section, we assume that the projection $\mathrm{Gaif}(H_{\leq\alpha})$ has already been computed, and, for each $v \in H$, $N_{\leq\alpha}(v)$ and $E_{>\alpha}(v)$ have been sorted. Such computations can be carried out during the build-up phase without increasing its running time.

6.3.1 Sampling a neighbor. The subroutine for sampling a neighbor is presented in Algorithm 4. For the purpose of readability we simplify the notation as follows. Recall that, given a k-treelet T with $k \geq 2$, there is a canonical decomposition of T into smaller trees T_1 and T_2; see Section 2. For any subset S_1 of colors, and any vertex v, we then write $C(S_1, v)$ for the counter $C(T_1, S_1, v)$, and $C(S_2, v)$ for the counter $C(T_2, S_2, v)$. Similarly, for an edge $e \in E$, we write $C(S_1, e)$ for $\sum_{u\in e} C(T_1, S_1, e)$, and so on. Finally, let $W_{\leq\alpha}(v) = \sum_{u\in N_{\leq\alpha}(v)} C(S_2, u)$ and $W_{>\alpha}(v) = \sum_{u\in N_{>\alpha}(v)} C(S_2, u)$.

Algorithm 4: SampleNeigh

Input: (T, S, v)
Output: (T_2, S_2, u)

let T_1, T_2 be the canonical decomposition of T;
choose S_1, S_2 w.p. $\propto C(S_1, v) \cdot \sum_{u \in N(v)} C(S_2, u)$;
let $p(N_{\leq\alpha}(v)) := \frac{W_{\leq\alpha}(v)}{W_{\leq\alpha}(v)+W_{>\alpha}(v)}$;
while u is not accepted **do**
  with probability $p(N_{\leq\alpha}(v))$ **do**
    choose $u \in N_{\leq\alpha}(v)$ with probability $\frac{C(S_2,u)}{W_{\leq\alpha}(v)}$;
  else
    while u is not accepted **do**
      choose $e \in E(v)$ with probability $\frac{C(S_2,e)}{\sum_{e' \in E(v)} C(S_2,e')}$;
      choose $u \in e$ with probability $\frac{C(S_2,u)}{C(S_2,e)}$;
      accept u with probability $\frac{1}{|E(v) \cap E(u)|}$
  accept u with probability $\frac{1}{1\{u \in N_{\leq\alpha}(v)\}+1\{u \in N_{>\alpha}(v)\}}$;
return (T_2, S_2, u);

Lemma 7. *Given as input (T, S, v), SampleNeigh returns (T_2, S_2, u) such that $u \in N(v)$ with probability proportional to $C(T_2, S_2, u)$.*

6.3.2 Sampling colorful treelets. We now use SampleNeigh to sample a colorful rooted treelet of H. To reduce the complexity of the task, we assume that the build-up phase computes a set of random generators that produce samples from certain distributions. Using the Alias method [21], these generators can be constructed in time linear in the support of the distribution, and yield independent random samples in time $O(1)$. In particular, we will assume the following generators:

- A generator that returns a pair (T, v) with probability proportional to $C(T, [k], v)$. The size of the support is $2^{O(k)} \cdot |V|$.
- For every $S_2 \subset [k]$, every treelet T_2 on $|S_2|$ vertices, and every $v \in V$, a generator that returns a vertex $u \in N_{\leq\alpha}(v)$ with probability proportional to $C(T_2, S_2, u)$. This can be done by iterating over $C(T_2, S_2, u)$ for all $u \in N_{\leq\alpha}(v)$, which can be done in time $O(\alpha^2|E|)$. The total time is therefore $2^{O(k)} \cdot \alpha^2|E|$.
- For every $S_2 \subset [k]$, every treelet T_2 on $|S_2|$ vertices, and every $e \in E_{>\alpha}$, we compute $C(T_2, S_2, e)$ in time $O(|e|)$. At this point we compute a generator that returns $u \in e$ with probability proportional to $C(T_2, S_2, u)$, which takes again time $O(|e|)$. The total time is therefore $2^{O(k)} \cdot |E_{>\alpha}|$.
- For every $S_2 \subset [k]$, every treelet T_2 on $|S_2|$ vertices, and every vertex type $E(v)$, a generator returning $e \in E(v)$ with probability proportional to $C(T_2, S_2, e)$, in time $O(\beta)$, for a total time of $2^{O(k)} \cdot \beta|V|$.
- For every $v \in V$, a generator that returns (S_1, S_2) with probability proportional to $C(T_1, S_1, v) \cdot \sum_{u \in N(v)} C(T_2, S_2, u)$. Recall that the neighbor sums $\sum_{u \in N(v)} C(T_2, S_2, u)$ are computed by the build-up phase. This takes total time $2^{O(k)} \cdot |V|$.

We also pre-compute the canonical decomposition of every treelet T on at most k vertices (this can be done in time $2^{O(k)} \cdot O(k)$). One can check that the total time spent by the calculations above is bounded by the running time of the build-up phase.

Lemma 8. *With the random number generators precomputed as above, each invocation of SampleNeigh takes expected time $O(\beta^2)$.*

The procedure for sampling a treelet is presented in Algorithm 5.

Algorithm 5: Sample

Input: (T, S, v) or NULL
Output: random uniform colorful treelet of H isomorphic to T colored with S rooted in v

if input is NULL **then**
  $S = [k]$;
  draw (T, v) with probability proportional to $C(T, S, v)$;
if $|T| = 1$ **then**
  return $(\{v\}, \emptyset)$;
$(T_2, S_2, u) \leftarrow$ SampleNeigh (T, S, v);
let $T_1 = T \setminus T_2$, $S_1 = S \setminus S_2$;
return Sample (T_1, S_1, v) + uv + Sample (T_2, S_2, u);

Corollary 1. *Algorithm 5 takes, on each invocation, expected time $O(k\beta^2)$. Moreover, the vertex set of the output treelet is $U \in \mathcal{V}_k(H)$ with probability proportional to the number of spanning trees $\sigma(H[U])$ of $\mathrm{Gaif}(H[U])$.*

6.3.3 Sampling hypergraphlets. As a final step, we show how to obtain an unbiased estimate of the frequency of the k-hypergraphlets. To this end, consider an invocation of Sample and let U the set of vertices of the returned treelet. We then need to compute:

(1) the k-hypergraphlet $H[U]$,
(2) the number of spanning trees $\sigma(H[U])$ of $\mathrm{Gaif}(H[U])$.

Thereafter, we just increase the sample counter of $H[U]$ by $\frac{1}{\sigma(H[U])}$. By virtue of Corollary 1, this yields an unbiased estimator of the number of colorful k-hypergraphlets of H isomorphic to $H[U]$. Note that, at this point, one can sample k-hypergraphlets uniformly by accepting $H[U]$ with probability $\frac{1}{\sigma(H[U])}$. Since $\frac{1}{\sigma(H[U])} \geq k^{-O(k)}$, see [7], this makes the sampling time's dependence on k grow to $k^{O(k)}$, as per Theorem 3.

For item 1, one could list the vertex types of the vertices of U to find the hyperedges of $H[U]$. This, however, would have a running time that scaled with the maximum degree $\Delta(H)$. Interestingly, we can do better by taking advantage of the (α, β)-niceness of H.

Lemma 9. *By spending an additional $O(\alpha^k \cdot |E_{\leq\alpha}|)$ preprocessing time, one can compute the k-hypergraphlet $H[U]$ induced by any given k-vertex $U \subseteq V$ in time $2^{O(k)} \cdot \beta$.*

For item 2, we first compute $G[U] = \mathrm{Gaif}(H[U])$ and then $\sigma(G[U])$. It is well known that $\sigma(G[U])$ can be computed via Kirchhoff's matrix tree theorem in time $O(k^\omega)$, where $\omega < 2.38$ is the matrix multiplication exponent. Let us then focus on computing $G[U]$. The obvious approach is to consider every pair $\{u, v\} \in U$ and check whether they are adjacent in $H[U]$, that is, in H. This could be done by intersecting their types, that is, by checking whether $E(u) \cap E(v) = \emptyset$. This however would take time $\Omega(E(u) + E(v))$, which could be linear in V. Once again, we leverage the (α, β)-niceness of H, together with the precomputations done in the build-up phase, to obtain a faster algorithm:

Lemma 10. *Given a subset $U \subseteq V$ of k vertices, one can compute the adjacency matrix A_U of $\mathrm{Gaif}(H[U])$ in time $O(k^2(\beta + \ln |V|))$.*

7 EXPERIMENTS

We conducted experiments to evaluate the practical performance of HyperMotivo in terms of sensitivity to the parameters, running time, memory usage, and estimation accuracy. We present the results for a limited choice of parameters; for further ones see our repository. We implemented HyperMotivo in C++ as an extension of Motivo[6] and made the source publicly available[7]. Like Motivo, we implemented HyperMotivo with native support for multi-threading.

7.1 Datasets

We use six hypergraphs obtained from publicly available data, and one synthetic hypergraph generated by a simple random model. See Table 1. We describe here below how each hypergraph was obtained. For reproducibility, all scripts used to construct the hypergraphs are available at our repository. Note that some hypergraphs are derived from inhomogeneous data through significant data wrangling (e.g., downloading several XML files from an interface, and then parsing and combining them).

stackoverflow-answers (SA) *& mathoverflow-answers* (MA). The SA and MA dataset, introduced in [20], encode answering activity respectively on StackOverflow and MathOverflow. Vertices correspond to questions, and each hyperedge contains all questions answered by a given user.

amazon-reviews (AR). The AR hypergraph has been introduced in [18]. Vertices represent individual reviewers, while each hyperedge corresponds to a product and contains the set of its reviewers.

cpc-group (CP). A patent–classification hypergraph built from USPTO PatentsView bulk data. Vertices are granted U.S. utility patents; hyperedges are Cooperative Patent Classification (CPC) group codes, each containing all patents assigned to that group.

datascience-se-tags (SE). Vertices represent questions, and hyperedges correspond to tags. By design, each question in stackexchange can have at most 5 tags; in other words, this means that this hypergraph is $(\alpha, 5)$-nice for every possible α. The SE hypergraph is built from the public data dump of Datascience Stack-Exchange.

arxiv-categories (AX). Vertices are arXiv articles, and hyperedges are arXiv subject categories. Each hyperedge contains all articles annotated with that category during a time span of a year. Both primary and secondary categories are included. The dataset has been generated by collecting data from the arXiv base endpoint.

Random Hypergraph (RH). An artificial hypergraph generated by a simple power-law model. First, we let $V = \{1, \ldots, 100.000\}$. We then generated 55.000 independent random hyperedges, each one as follows: we drew an integer $s \in \{2, \ldots, |V|\}$ with probability proportional to s^{-2}, and we then picked a subset $e \in \binom{V}{s}$ uniformly at random.

7.2 Setup

All experiments were conducted on a dedicated Linux machine equipped with two Intel(R) Xeon(R) CPU X5660 @ 2.80GHz, 128 GB of RAM and 2 TB of storage. Unless otherwise specified, all experiments reported here are performed for $k = 5$ using 8 threads; for other values see the supplementary material.

For each dataset, and for each $3 \le k \le 8$, we ran both HyperMotivo and Algorithm 1, asking each algorithm to take $K = 100.000$ samples. Whenever a time wall budget of 12 hours was reached, the process was killed. The value $\alpha^\star$ of α chosen by HyperMotivo, see Figure 2, comes from the linear-time algorithm presented in Section 6.1, with the following modification: instead of minimizing the bound of Equation (9), we minimize a weighted version of that bound, in the form $\gamma \cdot (\sum_{e \in E_{\le \alpha}} |e|^2) + (1-\gamma) \cdot (\sum_{v \in V} 2^{d_{>\alpha}(v)})$. We set $\gamma = 0.01$; this provided good performance across all datasets.

In the sampling phase, the hypergraphlet $H[U]$ is not constructed using the algorithm of Lemma 9. We instead use a simpler routine that, for each $v \in U$, intersects the incident hyperedges $e \in E_v$ with U. This approach performs well in our experiments and is considerably easier to implement.

Finally, we remark that the Amazon Reviews dataset represents a worst-case scenario for our algorithm. In this dataset, high-degree vertices are incentivized to participate in very large hyperedges, which affects the exponential term in the complexity bound. This behavior reflects the natural dynamics of review platforms, where popular products with many reviews attract an increasing number of buyers. Despite this, the performance of our algorithm is comparable to that of the baseline.

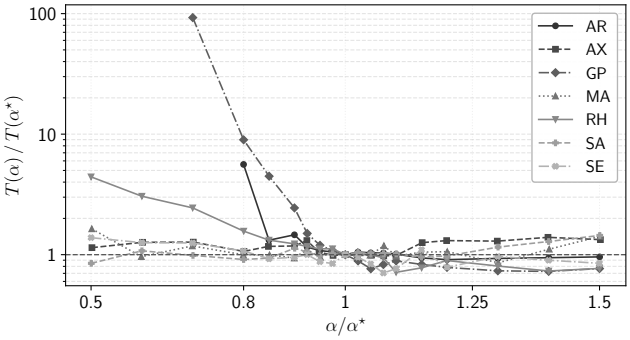


Figure 4: Sensitivity of HyperMotivo to the choice of $\alpha^\star$. For α varying around $\alpha^\star$, the plot shows the ratio $T(\alpha)/T(\alpha^\star)$ between the build-up times using α and using $\alpha^\star$. Our choice of $\alpha^\star$ yields good performance over all inputs. For AR and GP we omit points with running times exceeding 24 hours.

7.3 Computational Performance

7.3.1 Sensitivity to the choice of α. We evaluated the sensitivity of HyperMotivo's preprocessing time to perturbations of the value of $\alpha^\star$ by simply running it with values ranging from $0.5\alpha^\star$ to $1.5\alpha^\star$. Figure 4 shows that, across all datasets, the algorithm is rather stable and the chosen value $\alpha^\star$ provides good performance. (This is consistent with the fact that $\alpha^\star$ is not finely-tuned, but chosen in a way only coarsely dependent from H, see above).

7.3.2 The quadratic barrier and HyperMotivo. Figure 5 shows the build-up time of Algorithm 1 and of HyperMotivo as a function

[6] https://bitbucket.org/steven_/motivo/
[7] https://github.com/gfumagalli9/hyper-motivo/tree/hyper-motivo

Table 1: Hypergraphs used in our experiments.

H	$\lvert V(H)\rvert$	$\lvert E(H)\rvert$	rank(H)	$\Delta(H)$	avg. deg	$\lVert H\rVert$	$\lVert \text{Gaif}(H)\rVert$
SA	15,211,989	1,103,243	61,315	356	1.7	26,109,177	29,372,932,379
RH	100,000	55,000	77,965	26	9.6	960,168	12,083,305,610
AR	2,268,231	4,285,363	9,350	28,973	32.0	73,141,425	7,351,426,495
AX	259,961	164	40,571	9	1.9	487,469	5,744,552,301
GP	479,285	173,085	10,598	202	9.5	4,573,183	1,971,829,081
SE	36,775	702	11,391	5	3.1	113,074	295,309,540
MA	73,851	5,446	1,784	173	1.8	131,714	35,382,628

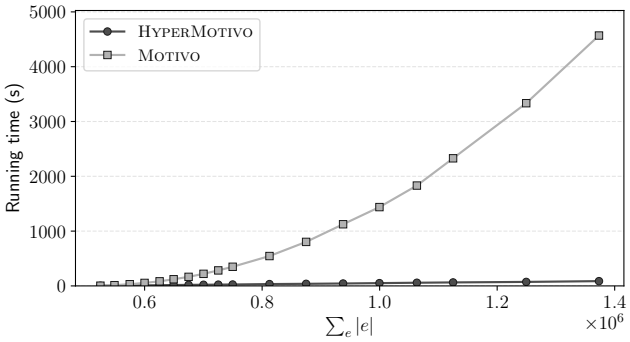


Figure 5: The linear-time behavior of HyperMotivo and the quadratic-time behavior of the naive Algorithm 1, on hypergraphs produced by a simple random model.

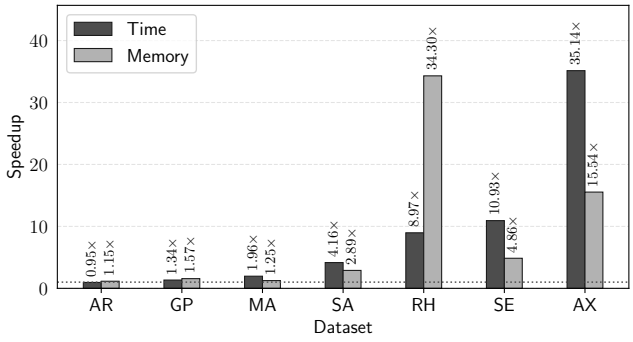


Figure 6: Reduction in wall-clock time and peak memory usage of HyperMotivo over Algorithm 1, on the build-up phase. On three inputs, HyperMotivo saves at least an order of magnitude in time or memory.

of the input size; note how Algorithm 1 exhibits a behavior coherent with $\Omega(\|H\|^2)$, and HyperMotivo a behavior consistent with $O(\|H\|)$. The inputs are generated as follows. Let $n, m, \alpha, \beta, \rho$ be fixed. We generate an n-vertex, m-hyperedge hypergraph by sampling ρm *small* hyperedges and $(1-\rho)m$ *large* hyperedges independently. Each small hyperedge is generated by first drawing its size s uniformly from $\{2, \ldots, \alpha - 1\}$ and then selecting s uniform random vertices. Each large hyperedge is generated by selecting L uniform random vertices for a fixed $L > \alpha$; any vertex whose degree in $H_{>\alpha}$ exceeds β is rejected and resampled. This yields an (α, β)-nice hypergraph. (This process is not meant to model real data, but rather to control the (α, β)-niceness of H).

7.3.3 Time and memory improvement. Figure 6 shows the ratio between the running time and memory usage of the build-up phase of Algorithm 1 and HyperMotivo (the higher the bar, the larger the improvement), for $k = 5$. For HyperMotivo, this includes the computation of the α-split (which is typically negligible) and the random number generators. For Algorithm 1, this includes the computation of Gaif(H), which often dominates, and the whole build-up phase of Motivo. As expected, HyperMotivo is substantially faster in almost all cases, even in hypergraphs of high degree. The only exception is AR, where HyperMotivo is only marginally outperformed.

7.3.4 Parallel scalability of the build-up phase. Figure 7 shows the running time of HyperMotivo's build-up phase for $k = 5$ on the MA hypergraph, as a function of the number of CPU threads. When

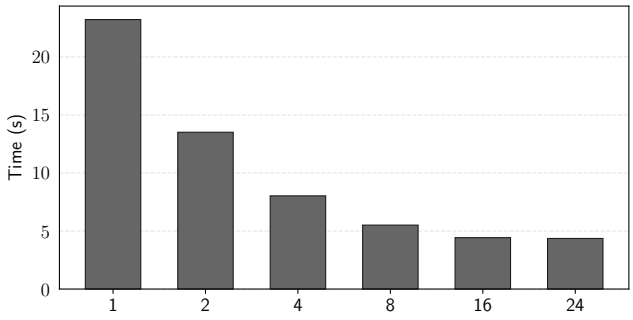


Figure 7: Running time of HyperMotivo's build-up phase on the MA dataset, for $k = 5$, as a function of the number of CPU threads used. Doubling the number of threads almost halves the running time, at least up to 4 threads.

using up to 4 threads, doubling the number of threads almost halves the running time. Other datasets, and other values of k, exhibit a similar behaviour. This shows that our techniques can be effectively employed in practice, too.

7.3.5 Sampling speed. Figure 8 shows the number of samples per second achieved by HyperMotivo and by Algorithm 1. Note that

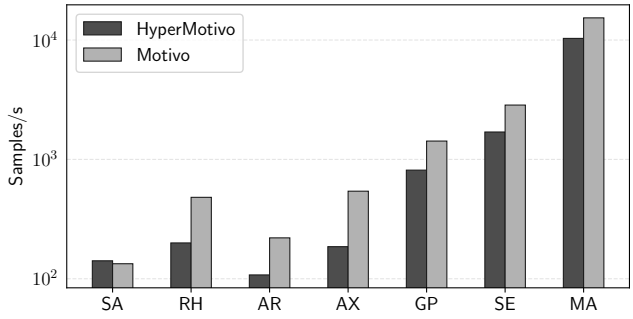


Figure 8: The sampling speed of HyperMotivo and Algorithm 1, in samples per second.

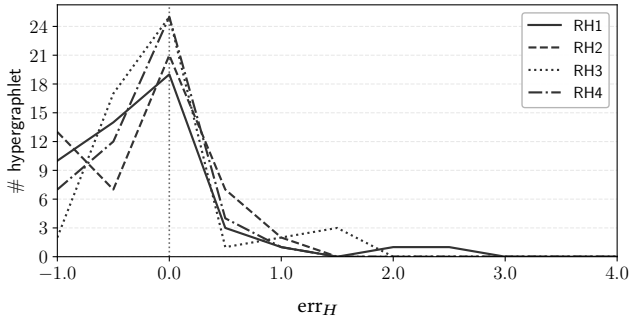


Figure 9: Distribution of per-hypergraphlet count errors on four synthetic inputs, in bins in the form $[\text{err} - 0.25, \text{err} + 0.25)$. To prevent extremely rare hypergraphlets from dominating the tails, we considered only the 50 most frequent ones.

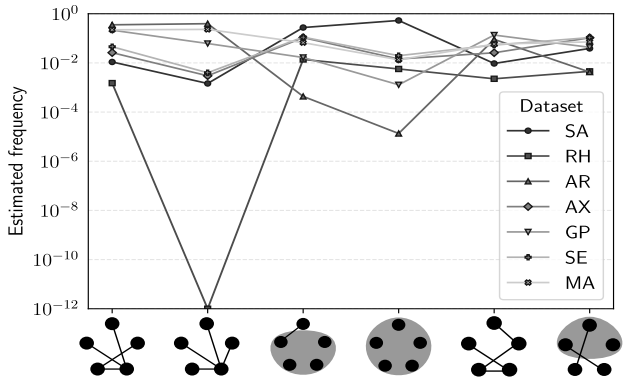


Figure 10: Frequency distribution of 5-hypergraphlets with highest aggregate frequency across all hypergraphs.

HyperMotivo's sampling algorithm is more complex than Motivo's one (which is in fact used as a subroutine). Nonetheless, HyperMotivo is always within a factor of 2 of Motivo.

7.4 Accuracy

We empirically evaluate the accuracy of HyperMotivo on small synthetic instances, for which exact counts can be computed. To this end we fix $|V| = 1000$ and $|E| = 500$. To generate an edge e, we first draw its size $s = |e|$ from a power-law distribution so that $\Pr[s] \propto s^{-3}$; then, we select s vertices uniformly at random. We generate in this way four random hypergraphs, RH1, …, RH4, so as to check that the results are consistent. For each hypergraphlet isomorphism type H, we measure the relative count error $\text{err}_H = (\hat{c}_H - c_H)/c_H$, where c_H denotes the exact count of H in the input hypergraph and $\hat{c}_H$ is the estimate returned by HyperMotivo when using 10^5 samples. Thus $\text{err}_H = 0$ corresponds to a perfect estimate, and $\text{err}_H = -1$ means H was never sampled. Figure 9 reports the distribution of err_H, clustered in bins in the form $[\text{err} - 0.25, \text{err} + 0.25)$. It can be seen that HyperMotivo yields accurate estimates for most of them, with the error distribution concentrated around 0.

A note of warning is needed here. The total number of (non-isomorphic) hypergraphlets on k vertices is *doubly exponential* in k; for $k = 5$, it is already around 10 million.[8] Thus, most of them necessarily have a very low frequency, and their count cannot be easily estimated. For this reason, Figure 9 is constructed considering only the 50 hypergraphlets with largest absolute counts; across the four inputs, the ground truth exhibits roughly 100 distinct hypergraphlet types in total; the remaining types are extremely rare and would therefore be missed by sampling with high probability (i.e., $\text{err}_H = -1$).

7.5 The k-hypergraphlet distribution

Finally, Figure 10 shows the frequency distribution of 5-hypergraphlets, as estimated by HyperMotivo. Note that the number of k-hypergraphlets explodes super-exponentially with k, and moreover for $k > 3$ even *depicting* k-hypergraphlets is challenging, save for the simplest ones. For this reason we show only the 5-hypergraphlets with highest aggregate frequency across all datasets.

[8] https://oeis.org/A000612 shows that the number of hypergraphs on 5 vertices is 18.666.624; at least half of them have the edge $e = V$ and are therefore connected.

8 CONCLUSIONS AND FUTURE WORK

We have shown that counting k-hypergraphlets is subtly harder than counting k-graphlets, but equally efficient algorithms exist under mild structural assumptions on the input hypergraphs. Our work is among the very first to investigate this problem formally, and leaves open many questions for future research. Is there an assumption *weaker* than (α, β)-niceness that is still sufficient for color coding to run in time linear in H? Under the definition of k-hypergraphlet given by so-called section hypergraphs (see Section 2.1), which properties of H allow for efficient counting algorithms? Is (α, β)-niceness helpful for other problems, too?

ACKNOWLEDGMENTS

The authors thank the Laboratory for Web Algorithmics[9] for kindly providing access to computational facilities.

[9] https://law.di.unimi.it/

REFERENCES

[1] Noga Alon, Raphael Yuster, and Uri Zwick. 1995. Color-Coding. *J. ACM* 42, 4 (1995), 844–856. https://doi.org/10.1145/210332.210337

[2] Jakob Lykke Andersen, Christoph Flamm, Daniel Merkle, and Peter F. Stadler. 2017. Chemical Transformation Motifs—Modelling Pathways as Integer Hyperflows. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 16 (2017), 510–523. https://doi.org/10.48550/arXiv.1712.02594

[3] Alessia Antelmi, Gennaro Cordasco, Daniele De Vinco, Valerio Di Pasquale, Mirko Polato, and Carmine Spagnuolo. 2025. Hypergraph Motif Representation Learning *(KDD '25)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/3690624.3709274

[4] Austin R. Benson, Rediet Abebe, Michael T. Schaub, Ali Jadbabaie, and Jon Kleinberg. 2018. Simplicial closure and higher-order link prediction. *Proceedings of the National Academy of Sciences* 115, 48 (Nov. 2018). https://doi.org/10.1073/pnas.1800683115

[5] Marco Bressan, Julian Brinkmann, Holger Dell, Marc Roth, and Philip Wellnitz. 2025. The Complexity of Counting Small Sub-Hypergraphs. arXiv:2506.14081 [cs.CC] https://arxiv.org/abs/2506.14081

[6] Marco Bressan, Flavio Chierichetti, Ravi Kumar, Stefano Leucci, and Alessandro Panconesi. 2018. Motif Counting Beyond Five Nodes. *ACM TKDD* 12, 4 (2018).

[7] Marco Bressan, Stefano Leucci, and Alessandro Panconesi. 2019. Motivo: Fast Motif Counting via Succinct Color Coding and Adaptive Sampling. *Proc. VLDB Endow.* 12, 11 (July 2019), 1651–1663. https://doi.org/10.14778/3342263.3342640

[8] Marco Bressan, Stefano Leucci, and Alessandro Panconesi. 2021. Faster Motif Counting via Succinct Color Coding and Adaptive Sampling. *ACM Trans. Knowl. Discov. Data* 15, 6, Article 96 (May 2021), 27 pages. https://doi.org/10.1145/3447397

[9] Jianer Chen, Benny Chor, Mike Fellows, Xiuzhen Huang, David W. Juedes, Iyad A. Kanj, and Ge Xia. 2005. Tight lower bounds for certain parameterized NP-hard problems. *Inf. Comput.* 201, 2 (2005), 216–231. https://doi.org/10.1016/j.ic.2005.05.001

[10] Jiawei Gao and Russell Impagliazzo. 2016. Orthogonal vectors is hard for first-order properties on sparse graphs. In *Electronic Colloquium on Computational Complexity (ECCC)*, Vol. 23. 53.

[11] Thomas Gaudelet, Noël Malod-Dognin, and Nataša Pržulj. 2018. Higher-order molecular organization as a source of biological function. *Bioinformatics* 34, 17 (09 2018), i944–i953. https://doi.org/10.1093/bioinformatics/bty570

[12] Jonas L. Juul, Austin R. Benson, and Jon Kleinberg. 2024. Hypergraph patterns and collaboration structure. *Frontiers in Physics* Volume 11 - 2023 (2024). https://doi.org/10.3389/fphy.2023.1301994

[13] Geon Lee, Jihoon Ko, and Kijung Shin. 2020. Hypergraph motifs: concepts, algorithms, and discoveries. *Proc. VLDB Endow.* 13, 12 (July 2020), 2256–2269. https://doi.org/10.14778/3407790.3407823

[14] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. 2008. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *Internet Mathematics* 6 (2008), 123 – 29. https://api.semanticscholar.org/CorpusID:3612445

[15] Quintino Francesco Lotito, Federico Musciotto, Federico Battiston, and Alberto Montresor. 2024. Exact and sampling methods for mining higher-order motifs in large hypergraphs. *Computing* 106, 2 (Feb. 2024), 475–494. https://doi.org/10.1007/s00607-023-01230-5

[16] Quintino Francesco Lotito, Federico Musciotto, Alberto Montresor, and Federico Battiston. 2022. Higher-order motif analysis in hypergraphs. *Communications Physics* 5, 1 (April 2022). https://doi.org/10.1038/s42005-022-00858-7

[17] Dániel Marx. 2013. Tractable Hypergraph Properties for Constraint Satisfaction and Conjunctive Queries. *J. ACM* 60, 6 (2013), 42:1–42:51. https://doi.org/10.1145/2535926

[18] Jianmo Ni, Jiacheng Li, and Julian McAuley. 2019. Justifying Recommendations using Distantly-Labeled Reviews and Fine-Grained Aspects. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. 188–197.

[19] Alice Patania, Giovanni Petri, and Francesco Vaccarino. 2017. The shape of collaborations. *EPJ Data Science* 6, 1 (2017), 18.

[20] Nate Veldt, Austin R. Benson, and Jon Kleinberg. 2020. Minimizing Localized Ratio Cut Objectives in Hypergraphs. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM Press.

[21] M. D. Vose. 1991. A linear algorithm for generating random numbers with a given distribution. *IEEE Transactions on Software Engineering* 17, 9 (1991), 972–975. https://doi.org/10.1109/32.92917

[22] Sebastian Wernicke. 2006. Efficient Detection of Network Motifs. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 3, 4 (Oct. 2006), 347–359. https://doi.org/10.1109/TCBB.2006.51