Computing Motifs in Hypergraphs

Duarte Nóbrega[1] and Pedro Ribeiro[1,2]

[1] DCC-FCUP, Universidade do Porto, Portugal
[2] CRACS & INESC-TEC, Portugal
up202005727@edu.fc.up.pt, pribeiro@dcc.fc.up.pt

Abstract. Motifs are overrepresented and statistically significant sub-patterns in a network, whose identification is relevant to uncover its underlying functional units. Recently, its extraction has been performed on higher-order networks, but due to the complexity arising from polyadic interactions, and the similarity with known computationally hard problems, its practical application is limited. Our main contribution is a novel approach for hyper-subgraph census and higher-order motif discovery, allowing for motifs with sizes 3 or 4 to be found efficiently, in real-world scenarios. It is consistently an order of magnitude faster than a baseline state-of-art method, while using less memory and supporting a wider range of base algorithms.

Keywords: Hypergraphs, Hyper-Subgraphs, Motifs, Subgraph Census

1 Introduction

Graphs are a valuable modelling tool of real-world systems, having applications in a broad set of scientific fields. The generated complex networks exhibit non-trivial topological features and many insightful metrics have been proposed to mine information and to characterise its properties [15,21].

In particular, a motif is defined as an overrepresented and statistically significant sub-pattern, whose number of occurrences deviates from its expected value, when compared to other similar networks [12]. Motif discovery has multiple practical applications such as creating early cancer diagnosis systems [2], extracting fingerprints from social networks [6] or characterising the reliability of critical infrastructures [5]. However, it is a difficult computational task, closely related to the subgraph isomorphism problem, which is known to be $\mathcal{NP}$-complete [4].

Most research done is focused on dyadic relationships, a relation between two entities, but many real-world interactions are intrinsically polyadic, involving more than two entities [3]. In order to directly encode these relations, the usual graph definition was modified to allow edges to connect any subset of vertices. This mathematical structure is formally known as a hypergraph.

Motif discovery has also been characterised in this model and different algorithms for it begin to emerge [7,9]. These algorithms are typically limited to sub-structures with small size, due to the exponential growth of the search space. However, even with this restriction, it is still possible to extract meaningful information [10].

In this paper, we present a novel approach for extracting motifs of sizes 3 and 4 in hypergraphs. Our method works by successively counting the structures having a relation with a given size, and then by removing them, until only dyadic relations are left, after which a subgraph counting algorithm is used, similarly to the work in [9]. However, it improves upon the $\mathcal{O}(E^2)$ auxiliary memory it requires, and removes the need for an enumeration based counting algorithm. Our approach uses $\mathcal{O}(E)$ additional memory, and is independent of the method used. We also show how different methods may be used to attain considerable speedups, when compared to generic ones, like `ESU`, by taking explicit advantage of the imposed size restriction.

2 Preliminaries

2.1 Terminology

A simple hypergraph $\mathcal{H}$ is defined as $\mathcal{H} = (V, E)$ where V is a finite set of hypervertices (or vertices, if there is no ambiguity) and $E = \{e_i \,|\, e_i \subseteq V,\ |e_i| \geq 2\}$ is a finite set of hyperedges. Let $n = |V|$ be the hypergraph's size and $m = |E|$. A hyperedge e_i has size k if $|e_i| = k$. If a hyperedge has size greater than 2, it is a higher-order interaction. The k-th order degree of a hypervertex $v \in V$ is defined as the number of hyperedges e_i such that $v \in e_i$ and $|e_i| = k$. The degree of a hypervertex is a multiset having all of its order degrees.

The simple hypergraphs $\mathcal{H} = (V, E)$ and $\mathcal{H}' = (V', E')$ are *isomorphic* if there is a function $f\colon V \to V'$ such that $\forall v \in V\colon v \in V \Leftrightarrow f(v) \in V'$ and $\forall e_i = \{v_1, ..., v_k\} \in E\colon e_i \in E \Leftrightarrow \{f(v_1), ..., f(v_k)\} \in E'$. A hyper-subgraph (or sub-hypergraph) $\mathcal{S}$ of $\mathcal{H} = (V, E)$ is a hypergraph $\mathcal{S} = (V', E')$ where $V' \subseteq V$ and $E' = \{e_i \,|\, e_i \subseteq V',\ |e_i| \geq 2\} \subseteq E$. $\mathcal{S}$ is an *induced hyper-subgraph* if $\forall e_i \in E\colon e_i \subseteq V' \Rightarrow e_i \in E'$. A sub-hypergraph is *two-connected* if, after removing all hyperedges with size greater than 2, there is a path between any pair of remaining vertices. A path is a sequence of vertices $(u_1, u_2, \ldots, u_k)$ such that $\{u_{k-1}, u_k\} \in E$, $\forall k \geq 2$.

A graph $\mathcal{G} = (V', E')$ is a *vertex projection graph* of a hypergraph $\mathcal{H} = (V, E)$ if $V' = V$ and $E' = \{(u, v) \,|\, (u, v) \subseteq e_k$, for some $e_k \in E\}$. A graph $\mathcal{G} = (V', E')$ is an *edge projection graph* of a hypergraph $\mathcal{H} = (V, E)$ if $V' = \{1, 2, ..., |E|\}$ and $E' = \{(i, j) \,|\, e_i \cap e_j \neq\ \emptyset,\ e_i \in E,\ e_j \in E\}$.

An example hypergraph $\mathcal{H} = (\{1, 2, 3, 4, 5, 6, 7\}, \{\{3, 5\}, \{1, 2, 3\}, \{1, 4, 6, 7\}\})$ is shown in Figure 1.

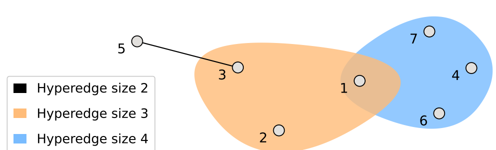

Fig. 1: Hypergraph with 7 hypervertices with 3 hyperedges.[a]

[a] All the hypergraph images were created using [8].

2.2 Problem Definition

We propose a method for the following problem:

Definition 1. ***Hyper-subgraph Census*** *Given an integer k and a hypergraph $\mathcal{H}$, count the number of distinct occurrences of each connected hypergraph of size k as an induced hyper-subgraph of $\mathcal{H}$. Two occurrences are distinct if they do not have the exact same vertex set.*

For the purposes of this paper we restrict our analysis to $k = 3$ and $k = 4$. The procedure used to solve this problem can then be used as the base counting algorithm to find motifs.

3 Literature Overview

Motif discovery is typically performed in three steps. Initially, a collection of similar networks is created, then a hyper-subgraph counting technique is utilised, and finally some metric is used to detect over (or under) occurring structures.

It is necessary to detect if two hypergraphs are isomorphic to update their counts correctly. In practice, there are methods that can solve instances with thousands of vertices and edges - one of them is `nauty` [11].

The subgraph census (SC) problem is similarly defined to its higher-order counterpart. Initially, `mfinder` appeared as the result of the work by Milo et at. [12]. It is an enumeration based recursive algorithm and a proof of concept of motif discovery. In order to guarantee the same subgraph is not counted twice, an identifier of each occurrence is kept in memory.

A more efficient algorithm called `ESU` was later proposed [22]. It greatly improved upon `mfinder` by reducing the memory consumption and search space, by only iterating through each subgraph once.

However, more sophisticated techniques were created since: a `g-trie`, a modified trie specifically designed to work with graphs [20], and `FaSE`, a state-of-art network-centric approach, that uses the g-trie's topology aware structure to avoid redundant isomorphism tests, along with additional low-level optimisations [17].

If the subgraph size is small, specific methods exist. For example, in order to find subgraphs of size 3, the triangle enumeration algorithm proposed in [16] has a $\mathcal{O}(m^{1.5})$ time and $\mathcal{O}(m)$ space complexity, and works particularly well in real-world instances. It improves upon the brute-force approach by several orders of magnitude. We refer the reader to [19] for a comprehensive overview about the existing state-of-art methods for SC.

The authors of [9] described how these algorithms could be adapted to solve our main problem: either the input hypergraph is converted to a projection graph, where then a SC algorithm is used, or multiple intermediate representations are created, by successively removing hyperedges with a given size, and then a single SC method is used at the end, when only dyadic relations are left.

The first approach, although simple, will find induced subgraphs that will not have a corresponding sub-hypergraph.

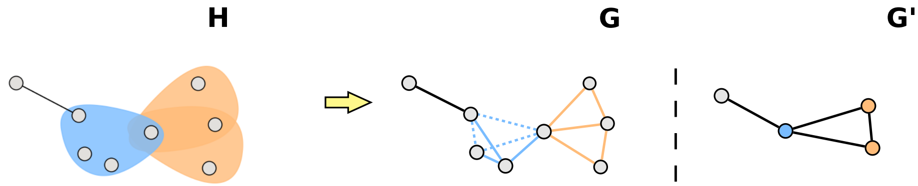

Fig. 2: Hypergraph conversion to projection graphs.

In Figure 2, when the blue-dotted triangle in **G** is found, it cannot be mapped to an existing sub-hypergraph, since those 3 nodes are a strict subset of the hyperedge of size 4. However, this problem is not inherent to the algorithm chosen, but to the transformation used.

It is possible to replace **G** by **G'**, an edge projection graph, and although fixing the previous issue, this representation is harder to manipulate, as it is more resource demanding to know how many vertices are already included in a corresponding sub-hypergraph during execution.

The second approach successively modifies the original hypergraph in order to simplify its structure and to speedup the usage of a SC algorithm. In Figure 3, an example is shown where this approach is executed in H_1, with the goal of finding all hyper-subgraphs of size 3.

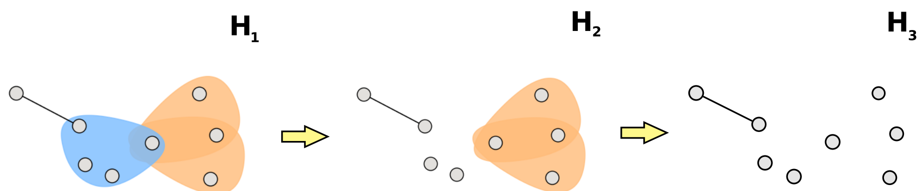

Fig. 3: Hypergraph conversion to intermediate forms. H_1 and H_2 are intermediate forms.

Firstly, any induced hyper-subgraph of size k will never have a hyperedge of size greater than k. In fact, it may only have one hyperedge with said size. In this case, the algorithm begins by removing all hyperedges of H_1 with excessive size, and H_2 is obtained. The search space can be further divided into two: the hyper-subgraph has a hyperedge of size 3, or it does not.

If it does, it can be induced by the vertex set of one of the two hyperedges of size 3, so each of them can be checked independently. If it does not, it implies it only contains hyperedges of size 2, so we can remove the others, and we obtain H_3, a graph with only dyadic relations where any SC algorithm can be used. All the subgraphs found have a valid corresponding sub-hypergraph. There are only two of size 3 in H_1, and they would have been found in the first step.

In both approaches, the same hyper-subgraph could be found multiple times. However, the total number of duplicates is bounded by $\mathcal{O}(E)$, if the size is 3, and $\mathcal{O}(E^2)$, if it is 4. The authors of the method in [9], from now on referred as `baseline`, use an auxiliary data structure to keep track of these duplicates.

In [9] an efficient technique to induce a sub-hypergraph with a vertex set $\mathcal{V}$ is described. The authors hash every hyperedge of size smaller or equal to 4, and then iterate over the subsets of $\mathcal{V}$ in order to verify if a given polyadic relation exists. The lookup takes $\mathcal{O}(1)$ amortised time and since $|\mathcal{V}| \leq 4$, in the worst-case, only $2^4 = 16$ subsets are checked, which can be regarded as a constant factor.

Finally, in order to identify an unexpected number of occurrences, a sufficiently large set of random similar networks is obtained by a configuration model [3,14], the z_{score} is calculated for each desired subgraph, as shown in Equation 1. Alternatively, Equation 2 can be used, with $\epsilon = 4$, following [13].

$$z_{score}(S_i) = \frac{F_{orig}(S_i) - \overline{F}_{random}(S_i)}{\sigma_{random}(S_i)} \tag{1}$$

$$\Delta(S_i) = \frac{F_{orig}(S_i) - \overline{F}_{random}(S_i)}{F_{orig}(S_i) + \overline{F}_{random}(S_i) + \epsilon} \tag{2}$$

$F_{orig}(S_i)$ denotes the frequency of the hyper-subgraph S_i in the original network. $\overline{F}_{random}(S_i)$ its average number of occurrences, and $\sigma_{random}(S_i)$ the standard deviation of its frequency, both calculated in the sample set.

4 Contribution

The methodology proposed in [9] requires an additional data structure to store all hyper-subgraphs found at every intermediate form, and uses `ESU`, when more efficient alternatives exist. Moreover, counting algorithms are incompatible with that approach, and enumeration ones require changes, as duplicate occurrences must be disregarded.

Our contribution builds upon this work, by entirely removing the dependence on this auxiliary data structure. With our method, any algorithm, counting based or not, may be used without any modification. This implies every existing SC algorithm may be swiftly integrated. Additionally, the memory overhead and a vast number of lookup calls are avoided, resulting in a performance improvement.

Similarly to `baseline`, we independently optimise our tool for the two sizes. Our tool also makes use of the same intermediate forms of the work in [9], since the authors concluded the usage of SC algorithms on projection graphs, depicted in Figure 2, is significantly slower. However, we avoid duplicates differently.

We introduce the notion of two-connectivity and modify the algorithms in [9] to attain the benefits mentioned, while maintaining correctness. Our proposed methods proceed similarly, following the procedure illustrated in Figure 3. The method for $k = 4$ implements an additional step, when compared to the version for $k = 3$. The same technique could theoretically be used for higher values of k, however implementing and maintaining a low memory usage is more difficult.

4.1 Hyper-subgraphs of size 3

The procedure for this size is shown in Algorithm 1.

Algorithm 1 Counting hyper-subgraphs of size 3

Require: A hypergraph $\mathcal{H} = (V, E)$.
Ensure: Frequency distribution of hyper-subgraphs with size 3.
1: Let $\mathcal{M}$ be a frequency hash map
2: Let $\mathcal{H}_2 \leftarrow$ remove all hyperedges with size greater than 2 from $\mathcal{H}$
3: **for each** hyperedge e of size 3 in E **do**
4:     $motif \leftarrow$ hyper-subgraph induced by e on $\mathcal{H}$
5:     **if** twoConnected($motif$) **then**
6:         $motif_2 \leftarrow$ hyper-subgraph induced by e on $\mathcal{H}_2$
7:         $\mathcal{M}$[isomorphicClass($motif_2$)] -= 1 $\triangleright$ Will be found again later
8:     **end if**
9:     $\mathcal{M}$[isomorphicClass($motif$)] += 1
10: **end for**
11: $\mathcal{S} \leftarrow$ ClassicalAlgorithm($\mathcal{H}_2$, 3)
12: **return** $\mathcal{M} + \mathcal{S}$ $\triangleright$ Add new occurrences to hash map and return

The correctness lies on the fact the only case a duplicate may be found is when a vertex set in $\mathcal{S}$ induces on $\mathcal{H}$ a structure having a hyperedge of size 3. However, if a sub-hypergraph of $\mathcal{H}$ has a hyperedge of size 3 and is two-connected, its vertex set will be in $\mathcal{S}$, but its induced occurrence on $\mathcal{H}_2$ is invalid, because it lacks the hyperedge of that size.

This suggests two different approaches: We either discard from $\mathcal{S}$ an occurrence that was already seen, by keeping track of each of them using an auxiliary data structure, or we preemptively subtract 1 to the counter of each invalid pattern, a two-connected hyper-subgraph that will be in $\mathcal{S}$. We chose the latter, contrary to the `baseline` method, as it allows us not to keep track of occurrences, while keeping the SC algorithm independent of the rest of the code.

This independence allows the counting method in line 11 to be enumeration based or not, exact or approximate, as long as it tackles the SC problem. Minimal changes must be made to accommodate different methods. The number of duplicate occurrences is bounded by the number of hyperedges of size 3, and in the worst-case, we enumerate them twice, which does not affect the time complexity. In line 9 we count sub-hypergraphs with a hyperedge of size 3, and without in line 11. The only particular case was handled, as described, in line 7. Any induced hyper-subgraph may be efficiently extracted from $\mathcal{H}$, using a hash table and applying the subset approach described previously.

Comparing with the method in [9], we do not maintain any data structure that keeps track of each individual occurrence, as we allow the algorithm to iterate through them at most two times. With this change, we save memory and achieve independence from the specific counting method used. The auxiliary space used is bounded by the number of isomorphic classes, instead of the number of occurrences, and an extra copy of $\mathcal{H}$.

4.2 Hyper-subgraphs of size 4

Our approach for this size extends the previous and is shown in Algorithm 2.

Algorithm 2 Counting hyper-subgraphs of size 4

Require: A hypergraph $\mathcal{H} = (V, E)$.
Ensure: Frequency distribution of hyper-subgraphs with size 4.

1: Let $\mathcal{M}$ be a frequency hash map
2: Let $\mathcal{H}_2 \leftarrow$ remove all hyperedges with size greater than 2 from $\mathcal{H}$
3: **for each** hyperedge e of size 4 in E **do**
4:     $motif \leftarrow$ hyper-subgraph induced by e on $\mathcal{H}$
5:     **if** twoConnected($motif$) **then**
6:         $motif_2 \leftarrow$ hyper-subgraph induced by e on $\mathcal{H}_2$
7:         $\mathcal{M}$[isomorphicClass($motif_2$)] -= 1 ▷ Will be found again later
8:     **end if**
9:     $\mathcal{M}$[isomorphicClass($motif$)] += 1
10: **end for**
11: **for each** hyperedge e of size 3 in E **do**
12:     Let $\mathcal{V}$ be a hash table
13:     **for each** hyperedge e_i adjacent to e **do**
14:         $motif \leftarrow$ hyper-subgraph induced by e on $\mathcal{H}$
15:         Let $size_e$ be the size of the biggest hyperedge in $motif$
16:         Let e_{lex} be the lexicographical greater hyperedge of size 3 in $motif$
17:         Let $node = (e \cup e_i) \setminus e$
18:         **if** $|e \cup e_i| = 4$ and $e = e_{lex}$ and $size_e = 3$ and $node$ not in $\mathcal{V}$ **then**
19:             **if** twoConnected($motif$) **then**
20:                 $motif_2 \leftarrow$ hyper-subgraph induced by e on $\mathcal{H}_2$
21:                 $\mathcal{M}$[isomorphicClass($motif_2$)] -= 1 ▷ Will be found again later
22:             **end if**
23:             $\mathcal{M}$[isomorphicClass($motif$)] += 1
24:             $\mathcal{V}$.insert($node$)
25:         **end if**
26:     **end for**
27: **end for**
28: $\mathcal{S} \leftarrow$ ClassicalAlgorithm($\mathcal{H}_2$, 4)
29: **return** $\mathcal{M} + \mathcal{S}$ ▷ Add new occurrences to hash map and return

Duplicates are avoided by generalising the previous idea, although the implementation is slightly more convoluted, as additional cases exist. We utilise the same concept of two-connectivity to deal with invalid structures in $\mathcal{S}$.

The algorithm begins by considering hyper-subgraphs having a hyperedge of size 4, similarly to how Algorithm 1 dealt with those of size 3.

The additional step, between lines 11 and 27, counts those having a hyperedge of size at most 3. This is done by fixing one with said size, and then pairing it with an adjacent one, e_i, in order to obtain a 4-node structure. The condition in line 18 guarantees each of them is only counted once, by skipping those having hyperedges with size greater than 3, and only counting when its lexicographically greater hyperedge is selected as value of e (line 11).

However, even when the greatest hyperedge is selected, different values of e_i may yield the same structure, so these duplicates must also be avoided. For this purpose, we use the data structure $\mathcal{V}$, which requires, in the worst case, $\mathcal{O}(|V|)$ additional memory, and keeps track of which nodes have been added. In total, we use $\mathcal{O}(|E|)$ extra space, since a copy of $\mathcal{H}$ is required. This improves over the `baseline` method, which needs $\mathcal{O}(|E|^2)$, because it would keep in memory every structure found until line 27.

5 Experimental Results

We implemented the methodology described and made the code publicly available[b]. We tested our algorithm with 3 different SC methods: `triangle` [16], `FaSE` [17] and `ESU'` [22]. `ESU'` is similar to `ESU`, but replaces the costly isomorphism tests during execution by a query to a hash table, which contains the pre-calculated labels by `nauty`. We compared their performance against the exact `baseline` approach [9]. All the methods were implemented in C++ and ran in a common framework. A summary of each dataset used is shown in Table 1.

Dataset	Source	Domain	Nodes	$\sum_H$	H_2	H_3	H_4
`ps`	[9]	proximity	242	12695	7748	4600	347
`hs`	[9]	proximity	327	7811	5498	2091	222
`EU`	[9]	e-mail	956	19985	12753	4938	2294
`history`	[9]	co-auth	371883	227428	160885	47423	19120
`geology`	[9]	co-auth	754196	663195	275736	227950	159509
`dblp`	[9]	co-auth	1433153	1780083	693364	667301	419418
`random`	own	synthetic	9999997	49999996	16671458	16659802	16668736
`clique`	own	synthetic	500	124750	124750	0	0

Table 1: Dataset description, after an initial pre-processing step, in order to satisfy our input restrictions. H_i denotes the number of hyperedges with size i.

A complete description of the first 6 datasets can be found in [9]. The `clique` dataset is a complete graph with 500 nodes. The `random` dataset was produced by the generator provided[b], using the number 5 as a seed, and with 5×10^7 random hyperedges. All the tests where performed in a similar environment, using an i9-13900KF Intel CPU, 32GB of RAM, and running Ubuntu 20.04.1 LTS. Our program used the number 10001 as a seed, to ensure reproducibility.

The results in Table 2 and Table 3 were obtained by executing each corresponding algorithm four times. The first result was ignored, and the average value of the remaining 3 was used. All the values were rounded to the nearest integer. The values shown only regard the time taken by the SC method, using the STEADY CLOCK implementation from the C++ STL, between the appropriate portions of code. The exact code used during testing is provided[b]. The time is in seconds, and the speedup measures how many times a given method was relatively faster than `baseline`.

[b] https://github.com/ComplexNetworks-DCC-FCUP/hypermotifs

Dataset	Subgraphs		`triangle`		`FaSE`		`ESU'`		`baseline`
	Types	Frequency	Time(s)	Speedup	Time(s)	Speedup	Time(s)	Speedup	Time(s)
`ps`	6	387846	< 0.005	-	0.035	8.143x	0.131	2.176x	0.285
`hs`	5	145829	< 0.005	-	0.010	9.600x	0.036	2.667x	0.096
`EU`	6	670087	< 0.005	-	0.054	9.037x	0.207	2.357x	0.488
`history`	6	531561	0.011	**32.455x**	0.101	3.535x	0.139	2.568x	0.357
`geology`	6	1159160	0.022	**46.500x**	0.268	3.817x	0.379	2.699x	1.023
`dblp`	6	4561470	0.089	**54.596x**	0.888	5.472x	1.925	2.524x	4.859
`random`	4	72234343	3.156	**31.412x**	20.724	4.784x	46.551	2.13x	99.137
`clique`	1	20708500	0.510	**39.298x**	3.576	5.605x	11.003	1.822x	20.042

Table 2: Results obtained, for each dataset, with $k = 3$.

Dataset	Subgraphs		`FaSE`		`ESU'`		`baseline`
	Types	Occurrences	Time(s)	Speedup	Time(s)	Speedup	Time(s)
`ps`	76	20409856	2.988	**9.136x**	8.188	3.334x	27.299
`hs`	68	4586917	0.498	**11.116x**	1.487	3.723x	5.536
`EU`	109	46710311	6.075	**10.766x**	18.377	3.559x	65.405
`history`	72	11382822	0.413	**33.201x**	2.929	4.681x	13.712
`geology`	130	11055083	0.671	**22.398x**	3.219	4.669x	15.029
`dblp`	167	61722014	3.832	**23.448x**	22.628	3.971x	89.853
`random`	11	430323276	53.052	-	215.055	-	-[c]
`clique`	1	2573031125	535.62[d]	**7.893x**	1615.79	2.616x	4227.71

Table 3: Results obtained, for each dataset, with $k = 4$.

Dataset	Time(s), $K = 3$		Time(s), $K = 4$	
	`modified`	`baseline`	`modified`	`baseline`
`ps`	0.006	0.005	1.270	0.959
`hs`	< 0.005	< 0.005	0.318	0.230
`EU`	0.007	0.006	1.514	1.183
`history`	0.057	0.065	0.490	0.492
`geology`	0.390	0.448	5.088	5.465
`dblp`	1.323	1.514	27.656	29.366
`random`	39.480	49.298	601.137	$-^c$
`clique`	< 0.005	< 0.005	0.034	0.032

(a) Processing time of intermediate forms.

Dataset	$K = 3$	$K = 4$
`ps`	4600	549505
`hs`	2091	134767
`EU`	4938	704704
`history`	47423	248545
`geology`	227950	2232716
`dblp`	667301	11603771
`random`	16659802	$> 138000000^c$
`clique`	0	0

(b) Hyper-subgraphs stored.

Table 4: Hash table impact on performance.

[c] Memory limit exceeded and execution killed.
[d] `FaSE` uses a 32-bit integer for subgraph occurrence counting, resulting in overflow.

In Table 2, a comparison between the execution time of the methods `triangle`, `FaSE`, `ESU’` and `baseline` is provided, for $k = 3$. In Table 3, with $k = 4$, but without the method `triangle`, as it is not compatible with this parameter.

Table 4 illustrates the hash table impact on performance, by showing: the difference in processing time of intermediate forms, when compared to our method, and the maximum number of hyper-subgraphs simultaneously stored.

As a proof of concept, we implemented a simple configuration model that randomly swaps nodes between different hyperedges, while maintaining the original degree sequence. The values given by Equations 1 and 2 are shown in Figure 4, for the 3 datasets in the co-auth domain. For comparison purposes, we included the dataset `EU`, which is from another domain.

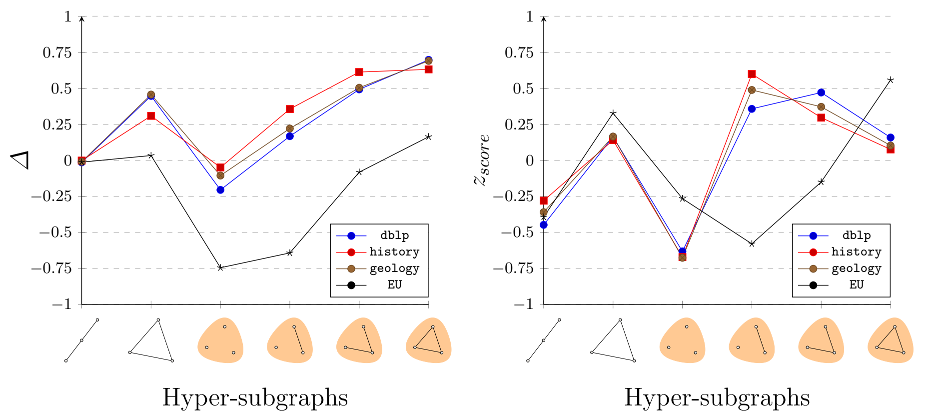

Fig. 4: On the left, $\Delta(S_i)$, on the right, $z_{score}(S_i)$, for each existing hypergraph of size 3.

6 Results Analysis

Only the median value of the three runs was shown, but the results did not significantly fluctuate.

The difference in processing time of intermediate forms was not significant, because our method did not specifically try to improve it. However, it still performed better, even though sharing the same underling idea, because our version avoids hash table insertions, saving time and memory, which is increasingly noticeable as the number of stored hyper-subgraphs increases.

When executing any SC method, we are also able to induce a hyper-subgraph faster, because we only have to process dyadic relations. This improvement is more apparent when $k = 4$.

6.1 Hyper-subgraphs of size 3

In terms of execution time, all the 3 proposed methods were faster than `baseline`. In particular, the `ESU'` method was consistently 2x faster. This result indicates the usage of our approach made each proposed algorithm (at least) 2x faster. The other gains may be attributed to the inner workings of each algorithm. The smallest speedup was registered in the `clique` dataset, because the auxiliary hash table was empty, so no lookup overhead existed in any method.

`FaSE` outperformed both versions of `ESU`. This was expected, and is in accordance with its author's claims. However, the difference was not very expressive, in part due to the small value of k, where this approach is not able to fully show its potential. Moreover, the isomorphic classes were pre-calculated, already replacing isomorphism tests by hash table lookups.

The method `triangle` clearly stood out from the rest. It was almost instantaneous in the datasets `ps`, `hs` and `EU`, so we did not include its time. Its performance may be explained by the low overhead, and because the largest networks were sparse, which is ideal for this method.

The number of structures `baseline` keeps in memory is exactly equal to the number of hyperedges of size 3, a linear overhead our method avoids, albeit not a significant one.

6.2 Hyper-subgraphs of size 4

As before, the two proposed algorithms were faster than `baseline`. `ESU'` was roughly 4x faster, double the speedup of its $k = 3$ counterpart, indicating the overhead increased. This was expected, as many more sub-structures are stored in the hash table, making both insertions and lookups slower. Additionally, the procedure to induce hyper-subgraphs takes longer. The smallest speedup was in the `clique` dataset, for similar reasons as aforementioned. In this dataset, `FasE` overflowed, since the occurrence counters are stored as integers. This maybe be fixed by adapting the original source code.

Overall, `FaSE` proved to be an order of magnitude faster, improving upon its previous results.

The `random` dataset and all dense datasets, apart from `clique`, required `baseline` to use substantially more memory when compared to its size 3 counterpart. In particular, in `random`, `baseline` used all the system's memory, and the execution was killed. This is explained by the approximate number of sub-hypergraphs which were stored just before it being stopped, 1.4×10^8. However, our method did not encounter any memory issues, showing how the difference between $\mathcal{O}(|E|)$ and $\mathcal{O}(|E|^2)$ is relevant in practice.

6.3 Motif Discovery

The results reported in Figure 4 allows us to exemplify the kind of information motif discovery may provide. Graphs from the same domain have a similar fingerprint, and the 3 datasets from the co-authorship domain generated similar graphs. However, here we see this result also holds for higher-order interactions. The difference between those 3 and the EU dataset (from a different domain) is very clear, as the graphs are completely different.

If comparing both formulas, we see both classify the same sub-structures as over or under represented, but the z_{score} produced a wider range of values. Our results are in line with the ones from the work in [10].

7 Conclusion

In this paper, we explored motif discovery in a general graph model, known as a hypergraph, which is capable of handling higher-order interactions. We introduced a novel way to perform motif discovery, tailored for sub-structures with size 3 or 4.

Our method was consistently an order of magnitude faster, and used less memory, when compared to a state-of-art method. Additionally, it may be swiftly paired with any existing algorithm for the subgraph counting problem. In certain scenarios, it was up to 55x faster, and able to solve instances the reference method could not, as it exhausted the available resources.

We also developed a command line tool that integrates all of our improvements, allowing for the use of different pre-existing methods, and offering customisation to the user. Our recommended choice, among the tested procedures, is the method `triangle`, for size 3, and `FaSE`, for size 4.

As for future work, support for greater hyper-subgraph sizes could be explored. We focused and tested exact approaches for subgraph counting, but approximate methods can also be used, using for instance sampling as it was done with `Rand-FaSE` [18]. We did not attempt to parallelise our application or any of the counting algorithms used, but it is possible, for instance with dynamic load balancing as it was shown with a shared memory multicore version of `FaSE` [1]. Additionally, we can further optimise our code, if the input instances are known to be small, to support $\mathcal{O}(1)$ node connectivity using an adjacency matrix. Finally, we intend to apply our approach to real data sets and to experiment with more complex and robust nulls models that are specific for random hypergraphs [3].

References

1. David Aparicio, Pedro Paredes, and Pedro Ribeiro. A scalable parallel approach for subgraph census computation. In *Euro-Par 2014: Parallel Processing Workshops, Revised Selected Papers*, pages 194–205, 2014.
2. Lina Chen, Xiaoli Qu, Mushui Cao, Yanyan Zhou, Wan Li, Binhua Liang, Weiguo Li, Weiming He, Chenchen Feng, Xu Jia, and Yuehan He. Identification of breast cancer patients based on human signaling network motifs. *Scientific reports*, 3(1):1–7, 2013.
3. Philip S Chodrow. Configuration models of random hypergraphs. *Journal of Complex Networks*, 8(3):cnaa018, 2020.

4. Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, 1971.
5. Asim K. Dey, Yulia R. Gel, and H. Vincent Poor. What network motifs tell us about resilience and reliability of complex networks. *Proceedings of the National Academy of Sciences*, 116(39):19368–19373, 2019.
6. Xu Hong-Lin, Yan Han-Bing, Gao Cui-Fang, and Zhu Ping. Social network analysis based on network motifs. *Journal of Applied Mathematics*, 2014, 2014.
7. Geon Lee, Jihoon Ko, and Kijung Shin. Hypergraph motifs: Concepts, algorithms, and discoveries. *Proceedings of the VLDB Endowment*, 13(12):2256–2269, 2020.
8. Quintino Francesco Lotito, Martina Contisciani, Caterina De Bacco, Leonardo Di Gaetano, Luca Gallo, Alberto Montresor, Federico Musciotto, Nicolò Ruggeri, and Federico Battiston. Hypergraphx: a library for higher-order network analysis. *Journal of Complex Networks*, 11(3):cnad019, 2023.
9. Quintino Francesco Lotito, Federico Musciotto, Federico Battiston, and Alberto Montresor. Exact and sampling methods for mining higher-order motifs in large hypergraphs. *Computing*, pages 1–20, 2023.
10. Quintino Francesco Lotito, Federico Musciotto, Alberto Montresor, and Federico Battiston. Higher-order motif analysis in hypergraphs. *Communications Physics*, 5(1):79, 2022.
11. Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, ii. *Journal of Symbolic Computation*, 60:94–112, 2014.
12. R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: Simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.
13. Ron Milo, Shalev Itzkovitz, Nadav Kashtan, Reuven Levitt, Shai Shen-Orr, Inbal Ayzenshtat, Michal Sheffer, and Uri Alon. Superfamilies of evolved and designed networks. *Science*, 303(5663):1538–1542, 2004.
14. Michael Molloy and Bruce Reed. A critical point for random graphs with a given degree sequence. *Random Structures & Algorithms*, 6(2-3):161–180, 1995.
15. David F Nettleton. Data mining of social networks represented as graphs. *Computer Science Review*, 7:1–34, 2013.
16. Daniel J. Nordman, Jonathan W. Berry, Cynthia A. Phillips, Luke A. Fostvedt, Alyson G. Wilson, and C. Seshadhri. Why do simple algorithms for triangle enumeration work in the real world? *Internet Mathematics*, 11(6), 11 2015.
17. Pedro Paredes and Pedro Ribeiro. Towards a faster network-centric subgraph census. In *2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2013)*, pages 264–271, 2013.
18. Pedro Paredes and Pedro Ribeiro. Rand-fase: fast approximate subgraph census. *Social Network Analysis and Mining*, 5(1):17, 2015.
19. Pedro Ribeiro, Pedro Paredes, Miguel EP Silva, David Aparicio, and Fernando Silva. A survey on subgraph counting: concepts, algorithms, and applications to network motifs and graphlets. *ACM Computing Surveys (CSUR)*, 54(2):1–36, 2021.
20. Pedro Ribeiro and Fernando Silva. G-tries: an efficient data structure for discovering network motifs. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 1559–1566, 2010.
21. Takashi Washio and Hiroshi Motoda. State of the art of graph-based data mining. *ACM SIGKDD Explorations Newsletter*, 5(1):59–68, 2003.
22. Sebastian Wernicke. Efficient detection of network motifs. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 3(4):347–359, 2006.