



Efficient Computation of Hyper-triangles on Hypergraphs

Haozhe Yin
The University of New South Wales
unswyhz@gmail.com

Kai Wang
Antai College of Economics and
Management, Shanghai Jiao Tong
University
w.kai@sjtu.edu.cn

Wenjie Zhang
The University of New South Wales
wenjie.zhang@unsw.edu.au

Ying Zhang
Zhejiang Gongshang University
ying.zhang@zjgsu.edu.cn

Ruijia Wu
Antai College of Economics and
Management, Shanghai Jiao Tong
University
rjwu@sjtu.edu.cn

Xuemin Lin
Antai College of Economics and
Management, Shanghai Jiao Tong
University
xuemin.lin@sjtu.edu.cn

ABSTRACT

Hypergraphs, which use hyperedges to capture groupwise interactions among different entities, have gained increasing attention recently for their versatility in effectively modeling real-world networks. In this paper, we study the problem of computing hyper-triangles (formed by three fully-connected hyperedges), which is a basic structural unit in hypergraphs. Although existing approaches can be adopted to compute hyper-triangles by exhaustively examining hyperedge combinations, they overlook the structural characteristics distinguishing different hyper-triangle patterns. Consequently, these approaches lack specificity in computing particular hyper-triangle patterns and exhibit low efficiency. In this paper, we unveil a new formation pathway for hyper-triangles, transitioning from hyperedges to hyperwedges before assembling into hyper-triangles, and classify hyper-triangle patterns based on hyperwedges. Leveraging this insight, we introduce a two-step framework to reduce the redundant checking of hyperedge combinations. Under this framework, we propose efficient algorithms for computing a specific pattern of hyper-triangles. Approximate algorithms are also devised to support estimated counting scenarios. Furthermore, we introduce a fine-grained hypergraph clustering coefficient measurement that can reflect diverse properties of hypergraphs based on different hyper-triangle patterns. Extensive experimental evaluations conducted on 11 real-world datasets validate the effectiveness and efficiency of our proposed techniques.

PVLDB Reference Format:

Haozhe Yin, Kai Wang, Wenjie Zhang, Ying Zhang, Ruijia Wu, and Xuemin Lin. Efficient Computation of Hyper-triangles on Hypergraphs. PVLDB, 18(3): 729-742, 2024.
doi:10.14778/3712221.3712238

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://github.com/dypoli/computing-hyper-triangle>.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 18, No. 3 ISSN 2150-8097.
doi:10.14778/3712221.3712238

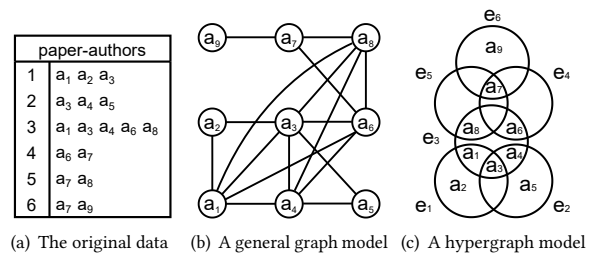


Figure 1: Graph model comparison.

1 INTRODUCTION

Hypergraphs are naturally used to model group-based relationships among entities in many real-world applications, such as co-authorship networks [23, 45, 53], biological networks [15, 35], and disease networks [37, 55]. In technical terms, a hypergraph $G = (V, E)$ is composed of a set of vertices V and a set of hyperedges E , where each hyperedge $e \in E$ represents relationships among multiple vertices. Compared with general graph models, hypergraphs can better preserve the integrity of data when describing groupwise intersections. Figure 1 shows an example to describe the co-authorship relationships using a general graph and a hypergraph. We can intuitively see that the hypergraph depicted in Figure 1(c) offers a clear advantage over general graphs by not only displaying the number of papers but also by showcasing the relationships between authors, with each paper serving as the unit of this relational mapping.

As building blocks of networks/graphs, motifs (i.e., repeated sub-graphs) are essential for graph analysis [9, 17, 34, 36, 39, 52]. Triangles (the smallest non-trivial clique), as a fundamental type of motifs, are typically used to extract information in general graphs [10, 14, 51, 56]. In hypergraphs, the concept of hyper-triangles (i.e., three pairwise connected hyperedges) has been proposed and proven useful in many applications [20, 41, 61]. For instance, in Figure 1(c), three hyper-triangles can be identified: $\{e_1, e_2, e_3\}$, $\{e_3, e_4, e_5\}$, $\{e_4, e_5, e_6\}$. These hyper-triangles illustrate complex interaction patterns and higher-order relationships within the network, while traditional triangles merely represent simple pairwise connections among three entities. By considering the relationships among the hyperedges within a hyper-triangle, we can categorize them into various patterns that exhibit distinct internal structures,

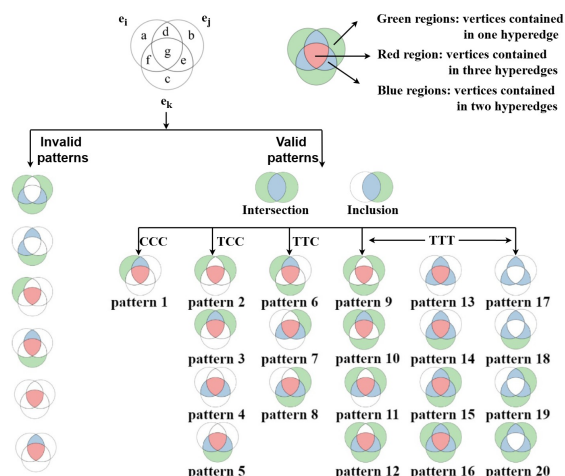


Figure 2: All the patterns for hyper-triangles.

as depicted in Figure 2. In real graphs, the semantic meaning associated with each pattern often varies. For instance, in social networks, different patterns signify distinct social behaviors or organizational structures. In this paper, we focus on the problem of computing hyper-triangles. Specifically, given a hypergraph $G = (V, E)$ and a pattern of hyper-triangles (as shown in Figure 2), we aim to find all the hyper-triangles belonging to such a pattern within G . Note that by simply combining the algorithms for computing each pattern, we can also identify all the hyper-triangles in G . The significance of hyper-triangle computation has been underscored in the literature. Below are some typical examples.

Network measurement. In hypergraphs, datasets from different domains exhibit distinct local structures due to the properties of data. By computing the frequency of hyper-triangle patterns across different datasets, some patterns within datasets of particular domains can reveal their structural design principles [28]. For example, through case studies, we find that in the co-authorship domain, publications from the same research group are more likely to form hyper-triangles of pattern 12 due to hierarchical relationships among authors, while publications from different research groups tend to form hyper-triangles of pattern 10. By analyzing the frequency of these patterns, we can estimate the number of research groups involved in a research field and their collaboration dynamics. Besides, in the email correspondence domain, email accounts within the same organization often exhibit administrative relationships, where a central "organizer" or admin account frequently sends and receives emails to and from all its member accounts. These interactions are more likely to form hyper-triangles of pattern 5, which signify centralized communication. However, when considering external or third-party accounts, due to the randomness of email sending, it is hard to form pattern 5 with these member accounts. By detecting pattern 5 hyper-triangles, we can estimate the number of members within a community. Such an approach can also offer insights for malicious email filtering and anomaly detection.

Hypergraph clustering coefficient. Triangle counting is a critical step in computing the clustering coefficient of a graph [19, 48]. Through the clustering coefficient, we can measure the localized

closeness and redundancy of a graph. In hypergraphs, the clustering coefficient [3, 13] is defined as $3 \times |\Delta| / |\sqcup|$, which requires computing the number of hyper-triangles. Here Δ denotes the set of hyper-triangles and \sqcup is the set of open hyper-triangles (i.e., a hyper-triangle in which two hyperedges are disconnected). Note that in hypergraphs, hyper-triangles encompass a variety of patterns as listed in Figure 1(b), which represents different interactions in real scenarios [28]. For example, in biological networks [18, 22], a specific pattern reflects a particular reaction chain. Based on this motivation, we further propose a fine-grained clustering coefficient metric (in Section 5) to offer users flexibility in adjusting the proportion of various hyper-triangle patterns and reflect different community structures of hypergraphs.

Motivations and Challenges. Existing studies [28, 29] search and categorize all motifs formed by three hyperedges by iteratively enumerating each hyperedge. As a result, when the search target is hyper-triangles of a specific pattern, this traverse-based algorithm still needs to enumerate all hyper-triangles. This process leads to numerous redundant checks of hyperedge combinations, resulting in inefficiencies. In this work, we aim to propose efficient algorithms that can perform targeted computing for specific hyper-triangle patterns, which faces the following challenges.

- (1) Since each hyperedge can be included in hyper-triangles of various patterns, it is challenging to identify the hyper-triangles of a specific pattern without enumerating hyper-triangles of other patterns.
- (2) Existing studies determine the pattern of hyper-triangle by enumerating all the vertices inside, which is very time-consuming. Therefore, it is also a challenge to efficiently identify the pattern of the hyper-triangle.

Our approaches. In this paper, we observe that a hyper-triangle can be viewed as a motif composed of three hyperwedges (unordered pairs of connected hyperedges). Therefore, we introduce a new pathway for forming the hyper-triangle, transitioning from two connected hyperedges to a hyperwedge, and then assembling three hyperwedges into a hyper-triangle. Based on the relationships between hyperedges, hyperwedges can be categorized into two types: intersection and inclusion. Building on this classification, we further divide hyper-triangle patterns into four classes depending on the types of three hyperwedges involved. Leveraging this insight, we propose a two-step framework to minimize the redundant computation. For a specific hyper-triangle pattern, the first step is to search and classify all the hyperwedges. Then, based on the types of hyperwedges involved, we search for hyper-triangles within the corresponding hyperwedge set, hence avoiding the enumeration of hyper-triangles of other patterns.

To address Challenge 2, by pre-saving the common vertices of the two hyperedges contained in each hyperwedge, we can avoid traversing all the vertices in hyper-triangles during the classification stage. Furthermore, by leveraging the structural characteristics of inclusion-type hyperwedges, the patterns involving inclusion-type hyperwedges can be identified in $O(1)$ time. Besides, since most patterns consist of three intersection-type hyperwedges, we further categorize patterns within this class into two subclasses and propose an advanced algorithm to accelerate the search speed for

these specific subclasses. Additionally, for the problem of counting hyper-triangles, we also provide an approximation algorithm.

Contributions. In general, our principal contributions are summarized as follows.

- We study the problem of computing hyper-triangles with different patterns to capture the intricate structural characteristics within hypergraphs.
- To avoid redundant checking during the searching process, we propose an efficient two-step framework based on a new hyper-triangle formation pathway. We also propose approximate counting algorithms for efficiently estimating the number of hyper-triangles in large hypergraphs.
- We propose a fine-grained clustering coefficient model that can reflect diverse properties based on different hyper-triangle patterns.
- We conduct extensive experiments on 11 real hypergraphs. The results demonstrate that the proposed exact algorithms outperform the state-of-the-art algorithm, and the proposed approximation algorithm can achieve more accurate results than the existing algorithm.

Organization. The remainder of this paper is structured as follows. Section 2 introduces the preliminary and the baseline algorithm. Section 3 introduces our proposed exact algorithms as well as the parallel version. All the approximation algorithms are presented in Section 4. In Section 5, we propose a fine-grained model of hypergraph clustering coefficient. Extensive experiments are conducted in Section 6. In Section 7, we introduce the related works of this paper. Finally, Section 8 concludes this paper.

2 PROBLEM DEFINITION

Notations	Description
$G = (V, E)$	A hypergraph with vertices V and hyperedges E
$E = \{e_1, \dots, e_{ E }\}$	The set of hyperedges
E_v	The set of all hyperedges containing the vertex v
$\wedge/\Delta/\sqcup$	The set of hyperwedges/hyper-triangles/open hyper-triangles
Δ_p	The set of hyper-triangles of pattern p
\wedge^t/\wedge^c	The set of all hyperwedges of the intersection/inclusion type
\wedge_{ij}	A hyperwedge composed of e_i and e_j
Ω_{ij}	The set of common vertices between e_i and e_j
ω_{ij}	The number of common vertices between e_i and e_j
N_{e_i}	The set of neighbors of hyperedge e_i
$T(\{e_i, e_j, e_k\})$	A hyper-triangle composed of e_i, e_j and e_k
$H[p]$	The count of the number of hyper-triangles of pattern p

Table 1: Notations

Definition 2.1 (Hypergraph). A hypergraph is a graph $G = (V, E)$ where V is a set of vertices and $E = \{e_1, \dots, e_{|E|}\}$ is a set of hyperedges. Each hyperedge $e_i \in E$ is a non-empty set of vertices.

In this paper, we use $|e_i|$ to denote the number of vertices in the hyperedge e_i . Additionally, for any two hyperedges, if they contain the same vertices, we consider them to be connected. We use $N_{e_i} = \{e_j \in E : e_i \cap e_j \neq \emptyset\}$ to represent all the hyperedges connected to e_i . For vertex $v \in V$, we use E_v to denote all the hyperedges containing the vertex v and we use E_v^r to represent r^{th} hyperedge in E_v . Note that, we assume that two different hyperedges do not contain the same set of vertices in this paper.

Definition 2.2 (Hyperwedge). Given a hypergraph $G = (V, E)$ and two hyperedges $e_i, e_j \in E$ with $e_i \cap e_j \neq \emptyset$, a hyperwedge \wedge_{ij}

is a path composed of e_i and e_j in G . We use Ω_{ij} to denote the set of common vertices between e_i and e_j and use ω_{ij} to denote the number of these vertices.

For example, in Figure 1(c), hyperedges e_1 and e_2 can form a hyperwedge \wedge_{12} since these two hyperedges share a common vertex a_3 . We denote the set of all hyperwedges as \wedge . For the hyperedge e_i , we use \wedge_{i-} to denote the set for all the hyperwedges containing e_i . Besides, we define the order of each hyperwedge \wedge_{ij} in G as $O(\wedge_{ij})$ to avoid duplication. For two hyperwedges \wedge_{ij} and \wedge_{kl} , $O(\wedge_{ij}) > O(\wedge_{kl})$ if $i > k$ or $i = k, j > l$.

Definition 2.3 (Intersection/Inclusion hyperwedge). Given a hypergraph $G = (V, E)$, for a hyperwedge \wedge_{ij} composed of e_i and e_j in G , if $e_i \subset e_j$ or $e_j \subset e_i$, then the hyperwedge \wedge_{ij} is of the inclusion type; otherwise, it is of the intersection type.

Similarly, we denote the set of all hyperwedges of the intersection and inclusion types as \wedge^t and \wedge^c respectively.

Definition 2.4 (Hyper-triangle). Given a hypergraph $G = (V, E)$ and three hyperedges $e_i, e_j, e_k \in E$ with $e_i \cap e_j \neq \emptyset$, $e_i \cap e_k \neq \emptyset$, and $e_j \cap e_k \neq \emptyset$, a hyper-triangle $T(\{e_i, e_j, e_k\})$ is a subgraph composed of e_i, e_j and e_k in G .

For a hyper-triangle $T(\{e_i, e_j, e_k\})$, it encompasses a total of seven regions, which are (a) $e_i \setminus e_j \setminus e_k$ (b) $e_j \setminus e_k \setminus e_i$ (c) $e_k \setminus e_i \setminus e_j$ (d) $e_i \cap e_j \setminus e_k$ (e) $e_j \cap e_k \setminus e_i$ (f) $e_k \cap e_i \setminus e_j$ (g) $e_i \cap e_j \cap e_k$ as shown in Figure 2. Based on the emptiness of each region, after excluding symmetric types, we can divide hyper-triangles into 20 different patterns. Based on the types of the three hyperwedges within each hyper-triangle, we further classify these 20 patterns into four major classes: *CCC*, *TCC*, *TTC* and *TTT* where *T* represents intersection, and *C* represents inclusion. For example, pattern 1 belongs to the *CCC* class because the three hyperwedges forming it are all of the inclusion types. Similarly, patterns 2 to 5 belong to the *TCC* class, patterns 6 to 8 belong to the *TTC* class, and patterns 9 to 20 belong to the *TTT* class. In real datasets, different hyper-triangle patterns can reflect different structural characteristics. Therefore, it is meaningful to search for hyper-triangles of specific patterns.

Problem statement. Given a hypergraph $G = (V, E)$ and a given pattern p (i.e., one of the patterns as shown in Figure 2), we aim to compute the hyper-triangles of the given pattern p in G .

In this paper, we propose exact and approximate algorithms to compute the hyper-triangles. Since the steps of count and enumerate are the same in the exact algorithm, hence the compute means enumerate/count in the exact algorithm. In the approximate algorithm, we estimate the number of hyper-triangles. Note that any hyper-triangle can be captured by one pattern in Figure 2, hence we can easily obtain all the hyper-triangles by simply combining algorithms for each pattern.

Existing solutions. [28] introduces an enumeration-based algorithm to find motifs in hypergraphs consisting of three hyperedges, which can also be used to find hyper-triangles.

The details of the algorithm are illustrated in Algorithm 1. The algorithm is based on each hyperedge e_i (Line 3) and checks all its neighboring pairs $\{e_j, e_k\}$ that connect to e_i in a higher order (Line 4), ensuring that each triple $\{e_i, e_j, e_k\}$ is enumerated only once. Note that the algorithm determines whether two hyperedges

are connected by pre-constructing a projection graph. If e_j, e_k are connected (i.e., $e_j \cap e_k \neq \emptyset$), then it proceeds to identify the pattern of the hyper-triangle $T(\{e_i, e_j, e_k\})$ (Lines 5-7).

Algorithm 1: Exact-bs

```

1  $G(V, E) \leftarrow$  Input hypergraph
2  $H \leftarrow$  Initialize a map to store the number of hyper-triangles
3 for each hyperedge  $e_i \in E$  do
4   for each unordered hyperedge pair  $\{e_j, e_k\} \in \binom{N_{e_i}}{2}$  do
5     if  $e_j \cap e_k \neq \emptyset$  and  $i < \min\{j, k\}$  then
6        $p \leftarrow$  Pattern of hyper-triangle  $T(\{e_i, e_j, e_k\})$ 
7        $H[p] + 1$ 
8 return  $H$ ;
```

However, Algorithm 1 performs redundant checking when searching for hyper-triangles. Specifically, to ensure the correctness of the results, the algorithm must navigate each hyperedge. This process not only consumes considerable time in filtering instances where the three hyperedges only form an open hyper-triangle but also results in the algorithm lacking specificity in computing particular hyper-triangle patterns. Furthermore, constructing the projection graph and identifying the pattern of the hyper-triangle necessitates the repeated enumeration of vertices within each hyperedge, resulting in significant time consumption. For example, consider a sparse hypergraph G with 102 hyperedges and 99 vertices, which contains only three hyper-triangles, as shown in Figure 3. If the required pattern belongs to the *CCC* class, Algorithm 1 needs to go through 104 hyperwedges and filter out 97 open-triangles formed by these hyperwedges to find the hyper-triangle $T(\{e_1, e_2, e_3\})$ of *CCC* class.

3 EXACT ALGORITHMS

To address the issues in the existing algorithm, we introduce a new pathway for constructing hyper-triangles. Rather than forming hyper-triangles directly from hyperedges, this pathway involves transitioning from pairwise connected hyperedges to hyperwedges, and subsequently assembling three hyperwedges into a hyper-triangle. For example, for the hypergraph in Figure 3, we first identify and classify all the hyperwedges, resulting in 6 inclusion-type hyperwedges and 100 intersection-type hyperwedges. In this case, for hyper-triangles of the *CCC* class, we only need to traverse these 6 inclusion-type hyperwedges to find the hyper-triangle $T(\{e_1, e_2, e_3\})$, which improves the search efficiency compared to Algorithm 1. Building upon this concept, we propose a two-step framework to reduce the redundant checking of hyperedge combinations. Under this framework, we propose efficient exact algorithms for computing hyper-triangle of specific patterns. Additionally, we provide parallel versions of exact algorithms to handle large datasets.

3.1 The Two-step Framework

The two-step framework contains the following steps. In the first step, we identify and categorize all hyperwedges into two groups w.r.t. their types (i.e., intersection and inclusion). In the second step, we resort to different algorithms to search for hyper-triangles of specific patterns based on these hyperwedges.

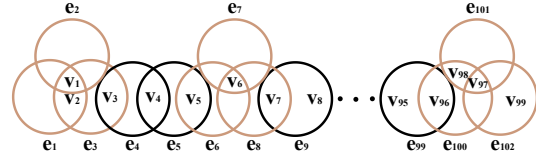


Figure 3: A sparse hypergraph

Algorithm 2 shows details of the two-step framework. Firstly, for each vertex v , we construct a list E_v that saves all hyperedges containing v in ascending order based on the IDs of hyperedges (Line 8). Then, we traverse each hyperedge e_i (Line 9). By using the lists E_v corresponding to all vertices contained in e_i , we search for all hyperedges e_j that can form hyperwedges with e_i and record the set of common vertices between e_i and e_j into Ψ (Lines 12-18). Based on the number of vertices contained in Ψ , we can determine the type of the hyperwedge \wedge_{ij} (Lines 19-22). After enumerating all the hyperwedges, all hyperwedges are categorized into two lists based on their types. Finally, based on the given pattern p , we employ the corresponding algorithm to find the hyper-triangles. Note that to avoid redundant computations for the same hyperedge in later steps, we store the hyperwedges as follows: for intersection-type hyperwedges \wedge_{ij}^t , we default to arranging hyperedges with smaller IDs first ($i < j$). For inclusion-type hyperwedges \wedge_{ij}^c , we default to place the hyperedge with a larger size first ($|e_i| > |e_j|$).

Algorithm 2: The Two-step Framework

```

1  $G(V, E) \leftarrow$  Hypergraph,  $p \leftarrow$  pattern of the hyper-triangles
2  $H \leftarrow$  Initialize a map to store the number of hyper-triangles
3  $\wedge^t, \wedge^c \leftarrow$  Preprocess( $G$ ) // step 1
4 Run the corresponding algorithms for  $p$  on the respective
  hyperwedge lists // step 2
5 return  $H$ 
6 Function Preprocess( $G$ ):
7    $\wedge^c \leftarrow \emptyset, \wedge^t \leftarrow \emptyset$ 
8    $\forall v \in V$  build a list  $E_v$  that saves all hyperedges containing  $v$ 
9   for each hyperedge  $e_i \in E$  do
10    while exist hyperedges that can form hyperwedge with  $e_i$  do
11       $e_j \leftarrow \emptyset, n \leftarrow 0, \Psi \leftarrow \emptyset$ 
12      for each vertex  $v \in e_i$  and  $|E_v| > 1$  do
13        Remove  $e_i$  from  $E_v$ 
14        if  $e_j = \emptyset$  or  $e_j > E_v^1$  then
15           $e_j \leftarrow E_v^1, n \leftarrow 1, \Psi \leftarrow \{v\}$ 
16        else if  $e_j = E_v^1$  then
17           $n \leftarrow n + 1, \Psi \leftarrow \Psi \cup v$ 
18       $\Omega_{ij} \leftarrow \Psi$  // set of common vertices between  $e_i$  and  $e_j$ 
19      if  $n < \min\{|e_i|, |e_j|\}$  then
20         $\wedge^t \leftarrow \wedge^t \cup \wedge_{ij}$ 
21      else if  $n = \min\{|e_i|, |e_j|\}$  then
22         $\wedge^c \leftarrow \wedge^c \cup \wedge_{ij}$ 
23 return  $\wedge^c, \wedge^t$ 
```

3.2 Algorithms for TTT class

In this subsection, we present the exact algorithms for searching patterns of the *TTT* class (patterns 9-20). Given that a significant portion of hyper-triangles falls within this class, the efficient computation and classification of these patterns are important. We first

propose a basic algorithm for this class. Then, we categorize the *TTT* class into two subclasses and propose more efficient algorithms.

The basic algorithm. The details of the basic algorithm are illustrated in Algorithm 3. Initially, we enumerate each intersection-type hyperwedge $\wedge_{ij}^t \in \wedge^t$, then search for another hyperwedge $\wedge_{ik}^t \in \wedge^t$ with $O(\wedge_{ij}^t) < O(\wedge_{ik}^t)$ (Line 3). If they can form a hyper-triangle, we further classify the hyper-triangles (Lines 5-7). Note that in order to determine the pattern of the hyper-triangle $T(\{e_i, e_j, e_k\})$, we need to find the common vertices between Ω_{ij} and Ω_{ik} .

Algorithm 3: Count-TTT

```

1  $G(V, E) \leftarrow$ Hypergraph,  $\wedge^t, \wedge^c \leftarrow$ Preprocess( $G$ )
2  $H \leftarrow$ Initialize a map to store the number of hyper-triangles
3 for each hyperwedge  $\wedge_{ij}^t \in \wedge^t$  do
4   for each hyperwedge  $\wedge_{ik}^t \in \wedge^t$  and  $O(\wedge_{ij}^t) < O(\wedge_{ik}^t)$  do
5     if  $e_j \cap e_k \neq \emptyset$  and  $\omega_{jk} < \min\{|e_j|, |e_k|\}$  then
6        $p \leftarrow$ Pattern of the hyper-triangle  $T(\{e_i, e_j, e_k\})$ 
7        $H[p] += 1$ 
8 return  $H$ 

```

While Algorithm 3 can search hyper-triangles of the *TTT* class, it still faces the following issues. (1) Since the majority of hyper-triangles belong to the *TTT* class, which includes 12 patterns, the runtime of the algorithm is still relatively long if we wish to find hyper-triangles of a specific pattern in this class. (2) The algorithm spends a significant amount of time filtering out the hyperedges that can only form an open hyper-triangle. To further enhance the efficiency of the algorithm, it is important to avoid traversing the open hyper-triangles. (3) The above process computes hyper-triangles by iteratively enumerating each hyperwedge. As the dataset grows, this method becomes increasingly time-consuming. Therefore, avoiding iteratively checking each hyperwedge would further enhance the efficiency of the algorithm.

To tackle the issue (1), the patterns within the *TTT* class are further divided into two subclasses. If there exists a set of vertices contained by all three hyperedges in a hyper-triangle pattern, then such a pattern is categorized into the *DenseTTT* subclass (patterns 9-16). Otherwise, they are classified into the *SparseTTT* subclass (patterns 17-20).

The algorithm for DenseTTT subclass. To address the issue (2), we observe that the three hyperedges contained in an open hyper-triangle do not share any common vertices, which is opposite to the structure of the hyper-triangles of the *DenseTTT* subclass. This implies that for an intersection-type hyperwedge \wedge_{ij}^t , if we can find other intersection-type hyperwedges \wedge_{ik}^t such that $\Omega_{ij} \cap \Omega_{ik} \neq \emptyset$, then they can definitely form hyper-triangles, hence we can avoid traversing open hyper-triangles. To achieve this method, for a hypergraph $G = (E, V)$, we initially construct a list τ_v for each vertex $v \in V$ which maps v to all the intersection-type hyperwedges \wedge_{ij}^t where $v \in \Omega_{ij}$. Then, for each intersection-type hyperwedge $\wedge_{ij}^t \in \wedge^t$, based on the list τ_v of each vertex $v \in \Omega_{ij}$, we search for other intersection-type hyperwedges \wedge_{ik}^t where $O(\wedge_{ij}^t) < O(\wedge_{ik}^t)$ (we denote the set of these hyperwedges for \wedge_{ij}^t as S_{ij}). Since the

hyperwedges in S_{ij} contain the same hyperedge e_i with \wedge_{ij}^t , all of them can form a hyper-triangle with \wedge_{ij}^t .

Lemma 3.1. For two hyperwedges $\wedge_{ij}^t, \wedge_{ik}^t \in \wedge^t$ where $O(\wedge_{ij}^t) < O(\wedge_{ik}^t)$. if the set of common vertices between e_i and e_j equals to the set of common vertices between e_i and e_k (i.e., $\Omega_{ij} = \Omega_{ik}$), then $S_{ij} \supset S_{ik}$.

Proof: For each hyperwedge $\wedge_{il}^t \in S_{ik}$, we have $O(\wedge_{ij}^t) < O(\wedge_{ik}^t) < O(\wedge_{il}^t)$. Since \wedge_{il}^t is found through the vertices $v \in \Omega_{ik}$, and $\Omega_{ij} = \Omega_{ik}$, hence \wedge_{il}^t will also be found when searching for hyperwedges through the vertex $v \in \Omega_{ij}$. Therefore, $S_{ij} \supset S_{ik}$.

Algorithm 4: Count-DenseTTT

```

1  $G(V, E) \leftarrow$ Hypergraph,  $\wedge^t, \wedge^c \leftarrow$ Preprocess( $G$ )
2  $H \leftarrow$ Initialize a map to store the number of hyper-triangles
3  $\forall v \in V$  Initialize a list  $\tau_v$ 
4 for each hyperwedge  $\wedge_{ij}^t \in \wedge^t$  do
5   for each vertex  $v \in \Omega_{ij}$  do
6      $\tau_v \leftarrow \tau_v \cup \wedge_{ij}^t$ 
7 for each hyperwedge  $\wedge_{ij}^t \in \wedge^t$  and  $\wedge_{ij}^t$  has not been visited do
8    $\Phi \leftarrow \emptyset, \Phi \leftarrow \Phi \cup \wedge_{ij}^t$ 
9   while exist hyperwedges that form hyper-triangles with  $\wedge_{ij}^t$  do
10     $w_1 \leftarrow \emptyset, \alpha \leftarrow 0$ 
11    for each vertex  $v \in \Omega_{ij}$  and  $|\tau_v| > 1$  do
12      remove  $\wedge_{ij}^t$  from  $\tau_v$ 
13       $\wedge_{kl}^t \leftarrow$ a hyperwedge in  $\tau_v$  with smallest order and
        contain same hyperedge with  $\wedge_{ij}^t$  (i.e.,  $k = i$ )
14      if  $w_1 = \emptyset$  or  $O(w_1) > O(\wedge_{kl}^t)$  then
15         $w_1 \leftarrow \wedge_{kl}^t, \alpha \leftarrow 1$ 
16      else if  $O(w_1) = O(\wedge_{kl}^t)$  then
17         $\alpha += 1$ 
18    for each hyperwedge  $w_2 \in \Phi$  do
19       $p \leftarrow$ pattern of hyper-triangle formed by  $w_1$  and  $w_2$ 
20       $H[p] += 1$ 
21    if  $\alpha = \omega_{ij}$  then
22       $\Phi \leftarrow \Phi \cup w_1$  and mark  $w_1$  as visited
23 return  $H$ 

```

We can address the issue (3) based on Lemma 3.1. Specifically, during the aforementioned traversal process, we incorporate the following step: for the current hyperwedge \wedge_{ij}^t , whenever another hyperwedge \wedge_{ik}^t is found so that $\Omega_{ij} = \Omega_{ik}$, we save \wedge_{ik}^t . After that, every hyperwedge found that can form a hyper-triangle with \wedge_{ij}^t will also be able to form a hyper-triangle with \wedge_{ik}^t .

Based on the above optimization, we obtain Algorithm 4. Firstly, we build the list τ_v for each vertex v (Lines 4-6). Then, we iteratively enumerate each hyperwedge. For the current hyperwedge \wedge_{ij}^t , if it has been visited, we skip it (Line 7). Otherwise, based on the lists τ_v for each vertex $v \in \Omega_{ij}$, we search for another hyperwedges w_1 (Lines 11-17). If w_1 can form a hyper-triangle with \wedge_{ij}^t , then it can also form hyper-triangles with the hyperwedges stored in Φ (lines 18-20). Finally, if the set of common vertices between the two hyperedges in hyperwedge w_1 equal to the set Ω_{ij} , then we add w_1 to Φ and mark it as visited (Lines 21-22).

The algorithm for SparseTTT subclass. The *SparseTTT* subclass includes four patterns (patterns 17-20). In this subclass, the three hyperedges within each pattern do not share any common vertices, which means we cannot use vertices to identify hyper-wedges. Therefore, the only way to search for hyper-triangles of this subclass is through a similar approach in *Count-TTT*. By checking whether three hyperedges contain the same vertices, we can rapidly filter out hyper-triangles of the *Dense-TTT* subclass, thereby speeding up the search process. However, the time savings from this approach are minimal, making the search for hyper-triangles of the *SparseTTT* subclass the most time-consuming task.

3.3 Algorithms for other classes

In this subsection, we introduce algorithms for the remaining three classes: *CCC* (pattern 1), *TCC* (patterns 2-5), and *TTC* (patterns 6-8). Unlike the *TTT* class, these three classes each contain at least one inclusion-type hyperwedge. Therefore, by leveraging the structural characteristics of inclusion-type hyperwedge, we can determine the pattern of the hyper-triangle in $O(1)$ time. This approach avoids the process of determining the pattern of the hyper-triangle by finding common vertices among the three hyperedges in it.

The algorithm for CCC class. We begin with the *CCC* class (pattern 1), which consists of one pattern formed by three inclusion-type hyperwedges. The details are presented in lines 4-8 of Algorithm 5. First, we traverse the hyperwedges in Λ^c . For the current hyperwedge Λ_{ij}^c , we search for another hyperwedge Λ_{ik}^c . If $e_j \supset e_k$, then there is a hyper-triangle $T(\{e_i, e_j, e_k\})$ of pattern 1 (denoted as p_1 in Algorithm 5).

The algorithm for TCC class. The *TCC* class includes four patterns (patterns 2-5). By analyzing the structure of the patterns within the *TCC* class, we can efficiently find and classify the hyper-triangles that belong to this class. Specifically, patterns 2 and 3 are constructed from two intersecting hyperedges, accompanied by a third hyperedge that is inclusively contained by the first two. Therefore, to find hyper-triangles of patterns 2 and 3, we first enumerate all intersection-type hyperwedges Λ_{ij}^t (Line 10). Then, for each hyperwedge Λ_{ij}^t , we search for hyperedges e_k that are contained by both e_i and e_j (Lines 11-12). For the found hyper-triangle $T(\{e_i, e_j, e_k\})$, if the number of common vertices between e_i and e_j is greater than the number of vertices in e_k (i.e., $\omega_{ij} > |e_k|$), then it belongs to pattern 3 (Lines 13-14). Otherwise, it belongs to pattern 2 (Lines 15-16). For patterns 4 and 5, the structure consists of two intersecting hyperedges, complemented by an additional hyperedge that includes both intersecting ones within it. Similarly, we enumerate all intersection-type hyperwedges Λ_{ij}^t (Line 10). Then, for each hyperwedge Λ_{ij}^t , we search for hyperedges e_k that contain both e_i and e_j (Lines 17-18). For the found hyper-triangle $T(\{e_i, e_j, e_k\})$, if $\omega_{ij} + \omega_{ik} - \omega_{jk} = |e_i|$, then it belongs to pattern 4 (Lines 19-20). Otherwise, it belongs to pattern 5 (Lines 21-22).

The algorithm for TTC class. The *TTC* class includes three patterns (patterns 6-8), and the search process is similar to the *TCC* class. As illustrated in lines 23-33 of Algorithm 5. First, we enumerate each inclusion-type hyperwedge $\Lambda_{ij}^c \in \Lambda^c$ and search for hyperedges e_k that intersect with both e_i and e_j (Lines 24-26). For

Algorithm 5: Count-TTC/TCC/CCC

```

1  $G(V, E) \leftarrow$  Hypergraph,  $\Lambda^t, \Lambda^c \leftarrow$  Preprocess( $G$ )
2  $p \leftarrow$  pattern of the hyper-triangles
3  $H \leftarrow$  Initialize a map to store the number of hyper-triangles
4 if  $p \in$  CCC class then
5   for each hyperwedge  $\Lambda_{ij}^c \in \Lambda^c$  do
6     for each hyperwedge  $\Lambda_{ik}^c \in \Lambda^c$  do
7       if  $e_j \supset e_k$  then
8          $H[p_1] + = 1$ 
9 if  $p \in$  TCC class then
10  for each hyperwedge  $\Lambda_{ij}^t \in \Lambda^t$  do
11    for each hyperwedge  $\Lambda_{ik}^c \in \Lambda^c$  do
12      if  $e_j \supset e_k$  then
13        if  $\omega_{ij} > |e_k|$  then
14           $H[p_3] + = 1$ 
15        else
16           $H[p_2] + = 1$ 
17      for each hyperwedge  $\Lambda_{ki}^c \in \Lambda^c$  do
18        if  $e_k \supset e_j$  then
19          if  $\omega_{ki} + \omega_{kj} - \omega_{ij} = |e_k|$  then
20             $H[p_4] + = 1$ 
21          else
22             $H[p_5] + = 1$ 
23 if  $p \in$  TTC class then
24  for each hyperwedge  $\Lambda_{ij}^c \in \Lambda^c$  do
25    for each hyperwedge  $\Lambda_{ik}^t, \Lambda_{ki}^t \in \Lambda^t$  do
26      if  $e_j \cap e_k \neq \emptyset$  and  $\omega_{jk} < \min\{|e_j|, |e_k|\}$  then
27        if  $\omega_{ik} = \omega_{jk}$  then
28           $H[p_6] + = 1$ 
29        else
30          if  $\omega_{ij} + \omega_{ik} - \omega_{jk} = |e_i|$  then
31             $H[p_7] + = 1$ 
32          else
33             $H[p_8] + = 1$ 
34 return  $H$ 

```

the found hyper-triangle $T(e_i, e_j, e_k)$, if the number of common vertices between e_i and e_k equals the number of common vertices between e_j and e_k (i.e., $\omega_{ik} = \omega_{jk}$), then this hyper-triangle belongs to pattern 6 (Lines 27-28). If not, we compute the value of $\omega_{ij} + \omega_{ik} - \omega_{jk}$. If it equals the number of vertices in e_i , then the hyper-triangle belongs to pattern 7 (Lines 30-31); otherwise, it belongs to pattern 8 (Lines 32-33).

3.4 Complexity Analysis

Algorithms	Preprocessing (T_1)	Searching and Classifying (T_2)
<i>Exact-bs</i>	$O(\sum_{e_i \in E} (e_i \cdot N_{e_i}))$	$O(\sum_{e_i \in E} (N_{e_i} ^2 \cdot e_i))$
<i>Count-TTC</i>	$O(\sum_{\Lambda_{ij} \in \Lambda} (e_i + e_j))$	$O(\sum_{\Lambda_{ij}^c \in \Lambda^c} (\Lambda_{ij}^c + \Lambda_{ij}^t))$
<i>Count-TCC</i>	$O(\sum_{\Lambda_{ij} \in \Lambda} (e_i + e_j))$	$O(\sum_{\Lambda_{ij}^c \in \Lambda^c} (\Lambda_{ij}^c + \Lambda_{ij}^t))$
<i>Count-CCC</i>	$O(\sum_{\Lambda_{ij} \in \Lambda} (e_i + e_j))$	$O(\sum_{\Lambda_{ij}^c \in \Lambda^c} (\Lambda_{ij}^c))$
<i>Count-TTT</i>	$O(\sum_{\Lambda_{ij} \in \Lambda} (e_i + e_j))$	$O(\sum_{\Lambda_{ij}^c \in \Lambda^c} (\Lambda_{ij}^c \cdot \omega_{ij}))$
<i>Count-DenseTTT</i>	$O(\sum_{\Lambda_{ij} \in \Lambda} (e_i + e_j))$	$O(\sum_{\Lambda_{ij}^c \in \Lambda^c} (S_{ij} \cdot \omega_{ij}))$
<i>Count-SparseTTT</i>	$O(\sum_{\Lambda_{ij} \in \Lambda} (e_i + e_j))$	$O(\sum_{\Lambda_{ij}^c \in \Lambda^c} (\Lambda_{ij}^c \cdot \omega_{ij}))$

* The total time complexity = $T_1 + T_2$

Table 2: Time complexity

Table 2 shows the time complexity of all exact algorithms in the two stages. First, in the preprocessing stage, *Exact-bs* examines each hyperedge e_i and enumerates the vertices within e_i to identify all its neighbors N_{e_i} . As a result, the time complexity of this process is $O(\sum_{e_i \in E} (|e_i| \cdot |N_{e_i}|))$. However, the proposed algorithms not only identify all hyperwedges \wedge_{ij} but also determine the common vertices between the two hyperedges in \wedge_{ij} . Therefore, the time complexity of the proposed approach is $O(\sum_{\wedge_{ij} \in \Lambda} (|e_i| + |e_j|))$.

After preprocessing, *Exact-bs* examines every pair of neighbors for each hyperedge e_i . As a result, it traverses $O(\sum_{e_i \in E} |N_{e_i}|^2)$ triples. For each hyper-triangle, *Exact-bs* requires $O(|e_i|)$ time to determine its pattern. Therefore, the time complexity at this stage is $O(\sum_{e_i \in E} (|N_{e_i}|^2 \cdot |e_i|))$. For the proposed algorithms, they perform the search based on hyperwedges. For example, *Count-TTC* first traverses each inclusion-type hyperwedge \wedge_{ij}^c , and for each \wedge_{ij}^c , it searches for all other intersection-type hyperwedges that contain the hyperedge e_i . Therefore, it traverses $O(\sum_{\wedge_{ij}^c \in \Lambda^c} (|\wedge_{i-}^t| + |\wedge_{-i}^t|))$ hyperwedge pairs (note that \wedge_{i-}^t is the set of all hyperwedges \wedge_{ik} where $i < k$). For each hyper-triangle, *Count-TTC* only requires $O(1)$ time to determine its pattern. hence the time complexity is $O(\sum_{\wedge_{ij}^c \in \Lambda^c} (|\wedge_{i-}^t| + |\wedge_{-i}^t|))$. For both *Count-TCC* and *Count-CCC*, they can also determine the hyper-triangle pattern in $O(1)$ time, therefore their time complexity is similar to that of *Count-TTC*. For *Count-TTT*, *Count-DenseTTT*, and *Count-SparseTTT*, for each hyperwedge pair $\{\wedge_{ij}, \wedge_{ik}\}$, they need to enumerate the vertices in Ω_{ij} to determine the pattern of the hyper-triangle, hence their time complexities are $O(\sum_{\wedge_{ij}^t \in \Lambda^t} (|\wedge_{i-}^t| \cdot \omega_{ij}))$, $O(\sum_{\wedge_{ij}^t \in \Lambda^t} (|S_{ij}| \cdot \omega_{ij}))$, $O(\sum_{\wedge_{ij}^t \in \Lambda^t} (|\wedge_{i-}^t| \cdot \omega_{ij}))$ respectively.

For the space complexity of *Exact-bs*, it requires $O(\sum_{e_i \in E} (|e_i| + |\wedge|))$ space to store all the hyperedges and the projection graph. As for the proposed algorithms, in addition to storing all the hyperedges, they also need to store the hyperwedges and the common vertices between the two hyperedges in each hyperwedge. Therefore, the space complexity is $O(\sum_{e_i \in E} (|e_i|) + \sum_{\wedge_{ij} \in \Lambda} (\omega_{ij}))$.

3.5 Parallelization

Note that our algorithms are also easily to be parallelized. Existing studies [28] convert the exact algorithm into a parallel form by parallelizing hypergraph projection. This allows multiple threads to independently process different hyperedges concurrently (denoted as *Par-bs*). In our algorithm, we initially use multiple threads to concurrently process different hyperedges and extract all hyperwedges along with their types and the common vertices between the two hyperedges involved. Once all threads stop, we aggregate the information for hyperwedges obtained from each thread together. Then, for each exact algorithm, we use multiple threads to process the respective hyperwedges in parallel (denoted as *Par-adv*).

4 APPROXIMATE COUNTING

In this section, we propose approximate algorithms that can efficiently estimate the number of hyper-triangles of various patterns in hypergraphs. Currently, commonly used estimation methods resort to random sampling approaches for motifs [24, 33, 42, 43, 47, 60], and there are three types of sampled elements including vertex sampling, edge sampling, and wedge sampling. Similarly, in

hypergraphs, we can employ vertex sampling, hyperedge sampling, and hyperwedge sampling for approximation. In general, the workflow of hypergraph sampling algorithm is as follows: (1) Extract a subgraph from the entire graph. (2) Sample some hyperedges, hyperwedges, and vertices from the subgraph, and for each hyperedge or hyperwedge, count the number of hyper-triangles where its ID/order is the smallest; for each vertex, identify all hyperedges containing it, and count the number of hyper-triangles where the IDs of those hyperedges are the smallest. (3) Based on the number of sample elements and the size of the subgraph, compute the number of hyper-triangles in the original graph. However, although these three sampling methods can be used to estimate the number of hyper-triangles, their results still exhibit differences. In fact, since each hyperedge contains a different number of vertices, the results obtained from vertex sampling are biased. The results from hyperedge and hyperwedge sampling, while unbiased, also have different variances. Hence, we present the corresponding proof and propose an approximation algorithm based on the sampling method with the smallest variance.

Lemma 4.1. *Using vertex sampling to estimate the number of hyper-triangles is not unbiased, while using hyperedge sampling or hyperwedge sampling to estimate the number of hyper-triangles is unbiased.*

Proof: We use the element to refer to a vertex, hyperedge, or hyperwedge, and we use δ to refer to the total number of the elements ($|V|$ for vertex, $|E|$ for hyperedge, $|\wedge|$ for hyperwedge). We denote the hyperedge with the lowest ID in the j^{th} hyper-triangle as Γ_j . Besides, we use $X_{ij}^p = 1$ to represent that j^{th} hyper-triangle of pattern p is being counted for the i^{th} element, otherwise $X_{ij}^p = 0$. For the i^{th} vertex, the probability of it being included in the j^{th} hyper-triangle is $\frac{|\Gamma_j|}{|V|}$. So, if we sample α vertices, by the linearity of expectation, its expected value is: $E[X] = \sum_{i=1}^{\alpha} \sum_{j=1}^{H[p]} E[X_{ij}^p] = \frac{\alpha}{|V|} \sum_{j=1}^{H[p]} |\Gamma_j|$. Finally, based on the proportion of α to $|V|$, the result is obtained as: $\frac{\alpha}{|V|} \sum_{j=1}^{H[p]} |\Gamma_j| \times \frac{|V|}{\alpha} = \sum_{j=1}^{H[p]} |\Gamma_j| \neq H[p]$. Therefore, the results obtained through this method are not unbiased. However, since the probability of i -th hyperedge or hyperwedge being included in the j^{th} hyper-triangle is $\frac{1}{\delta}$, if we sample α hyperedges or hyperwedges, its expected value is $E[X] = \sum_{i=1}^{\alpha} \sum_{j=1}^{H[p]} \frac{1}{\delta} = \frac{\alpha H[p]}{\delta}$. Based on the proportion of α to δ , we can obtain an unbiased result: $\frac{\alpha H[p]}{\delta} \times \frac{\delta}{\alpha} = H[p]$. Therefore, the results obtained through these two methods are unbiased.

Due to the property of hypergraph, the results obtained through vertex sampling are not unbiased. Therefore, we proceed to compare the variance between hyperedge sampling and hyperwedge sampling.

Lemma 4.2. *The variance of the results obtained by hyperwedge sampling is less than that obtained by hyperedge sampling.*

Proof: Similarly, we use the element to refer to a hyperedge, or hyperwedge. The variance that hyper-triangles of pattern p are counted while processing a sampled element can be expressed as $\text{Var}[\sum_{j=1}^{H[p]} X_{ij}^p]$. Besides, since the sampling is done uniformly at random, hence $\text{Cov}[X_{ij}^p, X_{kl}^p] = 0$. Therefore, when we sample α elements, their variance can be expressed and decomposed as

line 1 of Equation 1. Next, since $(X_{ij}^p)^2 = X_{ij}^p$ and through the relationships between variance, covariance, and expectation, we can further decompose the expression as:

$$\begin{aligned} \text{Var} \left[\sum_{i=1}^{\alpha} \sum_{j=1}^{H[p]} X_{ij}^p \right] &= \sum_{i=1}^{\alpha} \left(\sum_{j=1}^{H[p]} \text{Var}[X_{ij}^p] + \sum_{j \neq k} \text{Cov}(X_{ij}^p, X_{ik}^p) \right) \\ &= \sum_{i=1}^{\alpha} \left(\sum_{j=1}^{H[p]} (E[(X_{ij}^p)^2] - E[X_{ij}^p]^2) + \sum_{j \neq k} (E[X_{ij}^p \cdot X_{ik}^p] - E[X_{ij}^p]E[X_{ik}^p]) \right) \quad (1) \\ &= \alpha \left(H[p] \left(\frac{1}{\delta} - \frac{1}{\delta^2} \right) + \sum_{j \neq k} E[X_{ij}^p \cdot X_{ik}^p] - \sum_{j \neq k} \frac{1}{\delta^2} \right) \end{aligned}$$

We can further decompose the expected value $E[X_{ij}^p \cdot X_{ik}^p]$ as follows: $E[X_{ij}^p \cdot X_{ik}^p] = P[X_{ij}^p = 1]P[X_{ik}^p = 1 | X_{ij}^p = 1]$. For the j^{th} and k^{th} hyper-triangle, if they do not contain any common element, it is impossible to sample them at the same time. In this case, $E[X_{ij}^p \cdot X_{ik}^p] = 0$. Since we only count the hyper-triangle in which the ID/order of the hyperedge/hyperwedge is the smallest. Therefore, for two hyper-triangles that contain the same elements, the probability of being sampled simultaneously is $\frac{1}{3}$. In this case, $E[X_{ij}^p \cdot X_{ik}^p] = \frac{1}{\delta} \cdot \frac{1}{3}$. The variance can be expressed as: $\text{Var} \left[\sum_{i=1}^{\alpha} \sum_{j=1}^{H[p]} X_{ij}^p \right] = \alpha \left(H[p] \left(\frac{1}{\delta} - \frac{1}{\delta^2} \right) + \frac{\gamma_p}{3\delta} - \frac{\gamma_p}{\delta^2} \right)$, where γ_p is total pairs of hyper-triangle of pattern p , and γ_p' is total pairs of hyper-triangle of pattern p that share the same hyperedge/hyperwedge.

Comparing the two sampling methods. When we sample the same proportion of hyperedges/hyperwedges (i.e., $\frac{\alpha}{\delta} = c$ where c is a constant), the variance can be simplified as $H[p] \left(c - \frac{c}{\delta} \right) + \frac{c\gamma_p'}{3} - \frac{c\gamma_p}{\delta}$.

In real datasets, the term $\frac{c\gamma_p'}{3}$ is significantly larger than the other two terms. Therefore, we only need to compare the magnitudes of this term. It is obvious that the number of hyper-triangle pairs sharing the same hyperedge is greater. Hence, the results obtained by hyperwedge sampling are more accurate.

4.1 The Basic Algorithm

Through the above comparison, we conclude that hyperwedge sampling is unbiased and has smaller errors, making it the most suitable approximation method. In [28], researchers use hyperwedge sampling to estimate the number of motifs in a hypergraph. However, since their algorithm involves estimating the number of open hyper-triangles and the estimation process includes redundant checking of vertices. Therefore, we propose an algorithm specifically designed for estimating hyper-triangles. Additionally, we incorporate Preprocess to reduce redundant checking. To accommodate datasets of various sizes, we introduce a parameter σ , where $\sigma \in (0, 1]$. Depending on the value of this parameter, we extract subgraphs from the original graph for estimation. This approach helps avoid excessive computation times in large datasets.

The details of the approximate algorithm are presented in Algorithm 6. Firstly, we uniformly at random select $\sigma|E|$ hyperedges from the original graph to obtain the subgraph G' (Line 2). Then, we sample α hyperwedges from the subgraph (Line 4). For each hyperwedge \wedge_{ij} , we search for all hyperwedges \wedge_{ik} where $O(\wedge_{ij}^t) < O(\wedge_{ik}^t)$ that can form hyper-triangles with it (Lines 7-9). Finally,

we determine the number of hyper-triangles in the subgraph based on the ratio of α to $|\wedge|$, and then scale it up to estimate the number of hyper-triangles in the original graph based on the ratio of $|E'|^3$ to $|E|^3$ (Lines 10-11).

Algorithm 6: Appro-bs

```

1  $G(V, E) \leftarrow$  Hypergraph,  $\sigma \leftarrow$  sampling proportion,  $\alpha \leftarrow$  number of
   samples,  $\hat{H} \leftarrow$  map to store the estimate count of each pattern
2  $G'(V', E') \leftarrow$  sample  $\sigma|E|$  hyperedges from  $G$  uniformly at random
3  $\wedge^t, \wedge^c \leftarrow$  Preprocess( $G'$ )
4 for  $n \leftarrow 1 \dots \alpha$  do
5    $\wedge_{ij} \leftarrow$  sample a hyperwedge from  $G'$  uniformly at random
6   for each hyperwedge  $\wedge_{ik} \in \wedge$  do
7     if  $O(\wedge_{ij}^t) < O(\wedge_{ik}^t)$  and  $e_j \cap e_k \neq \emptyset$  then
8        $p \leftarrow$  Pattern of hyper-triangle  $T(\{e_i, e_j, e_k\})$ 
9        $\hat{H}[p] += 1$ 
10 for  $p \leftarrow 1 \dots 20$  do
11    $\hat{H}[p] = \hat{H}[p] \cdot \frac{|\wedge|}{\alpha} \cdot \sigma^{-3}$ 
12 return  $\hat{H}$ 

```

4.2 The Advanced Algorithm

For the basic algorithm, although using hyperwedges as samples shows advantages over vertex and hyperedge sampling, it still has some limitations. Firstly, the basic algorithm does not incur any optimization step to estimate specific patterns, therefore, it still needs to find the common vertices among the three hyperedges when classifying hyper-triangles. Furthermore, the basic algorithm is unable to estimate hyper-triangles of specific patterns, and when the sample size is insufficient, hyper-triangles of some patterns are difficult to capture, resulting in an underestimated outcome.

Algorithm 7: Appro-adv

```

1  $G(V, E) \leftarrow$  Hypergraph,  $\sigma \leftarrow$  sampling proportion,  $\alpha \leftarrow$  number of
   samples,  $\hat{H} \leftarrow$  map to store the estimate count of each pattern
2  $G'(V', E') \leftarrow$  sample  $\sigma|E|$  hyperedges from  $G$  uniformly at random
3  $\wedge^t, \wedge^c \leftarrow$  Preprocess( $G'$ )
4  $\alpha_1 \leftarrow \frac{|\wedge^t|^2 \cdot \alpha}{|\wedge|^2}$ ,  $\alpha_2 \leftarrow \frac{|\wedge^c|^2 \cdot \alpha}{|\wedge|^2}$ ,  $\alpha_3$  and  $\alpha_4 \leftarrow \frac{|\wedge^t| |\wedge^c| \cdot \alpha}{|\wedge|^2}$ 
5 for  $n \leftarrow 1 \dots \alpha_1$  do
6    $\wedge_{ij}^t \leftarrow$  sample a hyperwedge from  $\wedge^t$ 
7    $\hat{H} \leftarrow$  use Count-TTT to count the number of TTT-class
   hyper-triangles that include  $\wedge_{ij}^t$ 
8 for  $t \leftarrow 9 \dots 20$  do
9    $\hat{H}[p] = \hat{H}[p] \cdot \frac{|\wedge^t|}{\alpha_1} \cdot \sigma^{-3}$ 
10 Sample  $\alpha_2, \alpha_3$ , and  $\alpha_4$  hyperwedges from  $\wedge^c, \wedge^t$ , and  $\wedge^c$ 
11  $\hat{H} \leftarrow$  Estimate the number of hyper-triangles of class CCC, TTC, and
   TCC following the similar procedure of lines 5-9.
12 return  $\hat{H}$ 

```

To address these issues, we observe that by separately sampling from the lists of intersection-type and inclusion-type hyperwedges, we can independently estimate the number of hyper-triangles for each class. Additionally, during the estimation process, we can incorporate optimization steps specific to each class, thereby enhancing the efficiency of the algorithm. Based on this method, we

propose an advanced approximation algorithm that allows for targeted estimation of specific patterns. The detail is presented in Algorithm 7. After extracting a random subgraph from the original graph, we divide the number of samples into four parts according to the method described in line 4. When counting hyper-triangles of the *TTT* class, we randomly sample $\frac{|\Lambda^t|^{2-\alpha}}{|\Lambda|^2}$ hyperwedges from Λ^t , and use the *Count-TTT* algorithm to calculate the number of hyper-triangles that include them (Lines 5-9). For hyper-triangles of the *CCC* class, we sample $\frac{|\Lambda^c|^{2-\alpha}}{|\Lambda|^2}$ hyperwedges from Λ^c , and then use *Count-CCC* to count the number of hyper-triangles containing these hyperwedges. Similarly, following the above procedure, we sample $\frac{|\Lambda^t||\Lambda^c| \cdot \alpha}{|\Lambda|^2}$ hyperwedges to estimate the number of the hyper-triangles of the remaining two classes. Note that if our goal is to focus on a specific class, we can sample hyperwedges only for that class, thereby avoiding unnecessary counting.

Lemma 4.3. *The time complexity of Appro-adv is: $O(\sum_{\Lambda_{ij} \in \Lambda} (|e_i| + |e_j|) + \sum_{i=1}^{\alpha_1} (|\Lambda_{-i}^t| \cdot \omega_{ij}) + \sum_{i=1}^{\alpha_2} (|\Lambda_{-i}^c|) + \sum_{i=1}^{\alpha_3} (|\Lambda_{-i}^c| + |\Lambda_{-i}^t|) + \sum_{i=1}^{\alpha_4} (|\Lambda_{-i}^t| + |\Lambda_{-i}^c|))$.*

Proof: In the preprocessing stage, the time complexity of Appro-adv is $O(\sum_{\Lambda_{ij} \in \Lambda} (|e_i| + |e_j|))$. After that, it samples α_1 hyperwedges and counts the number of hyper-triangles of *TTT* class that contain these hyperwedges through *Count-TTT*. Hence the time complexity is $O(\sum_{i=1}^{\alpha_1} (|\Lambda_{-i}^t| \cdot \omega_{ij}))$ (as mentioned in Table 2). Using the same method, we can obtain the time complexities of the remaining three parts. By combining the preprocessing time with the time of these four parts, we can obtain the total time complexity.

Lemma 4.4. *The variances of the results obtained by Appro-adv corresponding to the hyper-triangles of TTT, CCC, TCC, TTC class are: $(|\Lambda^t|-1)\mathcal{T} + \frac{|\Lambda^t|}{|\Lambda|}\mathcal{T}' - \mathcal{T}''$, $(|\Lambda^c|-1)\mathcal{T} + \frac{|\Lambda^c|}{|\Lambda|}\mathcal{T}' - \mathcal{T}''$, $\frac{(|\Lambda^t|-1)|\Lambda^c|}{|\Lambda^t|}\mathcal{T} + \frac{|\Lambda^c|}{|\Lambda^t|}\mathcal{T}' - \frac{|\Lambda^c|}{|\Lambda^t|}\mathcal{T}''$, $\frac{(|\Lambda^c|-1)|\Lambda^t|}{|\Lambda^c|}\mathcal{T} + \frac{|\Lambda^t|}{|\Lambda^c|}\mathcal{T}' - \frac{|\Lambda^t|}{|\Lambda^c|}\mathcal{T}''$ respectively, where $\mathcal{T} = \frac{\alpha H[p]}{|\Lambda|^2}$, $\mathcal{T}' = \frac{\alpha Y_p}{3|\Lambda|}$ and $\mathcal{T}'' = \frac{\alpha Y_p}{|\Lambda|^2}$.*

Proof: Based on the proof of Lemma 4.2, when sampling α hyperwedges, by setting δ as $|\Lambda|$, the resulting variance is $\frac{(|\Lambda|-1)\alpha H[p]}{|\Lambda|^2} + \frac{\alpha Y_p}{3|\Lambda|} - \frac{\alpha Y_p}{|\Lambda|^2}$. For *Appro-adv*, when estimating the hyper-triangle of the *TTT* class, we sample α_1 hyperwedges from Λ^t for estimation where $\alpha_1 = \frac{|\Lambda^t|^{2-\alpha}}{|\Lambda|^2}$. Hence, we replace Λ and α with Λ^t and α_1 , respectively. After simplification, we obtain the variance of the *TTT* class: $\frac{(|\Lambda^t|-1)\alpha H[p]}{|\Lambda|^2} + \frac{|\Lambda^t|}{|\Lambda|} \frac{\alpha Y_p}{3|\Lambda|} - \frac{\alpha Y_p}{|\Lambda|^2}$. Using the same method, we can obtain the variance of the remaining three classes.

Comparing the two approximate algorithms. As mentioned earlier, the second term in the variance is the largest term. The variance of *Appro-bs* is given by $\frac{(|\Lambda|-1)\alpha H[p]}{|\Lambda|^2} + \frac{\alpha Y_p}{3|\Lambda|} - \frac{\alpha Y_p}{|\Lambda|^2}$. Obviously, its second term is greater than the second term in all the variances from Lemma 4.4, leading to a larger error in the results obtained through *Appro-bs*. Additionally, apart from accuracy, since *Appro-adv* uses methods from exact algorithms to compute the number of hyper-triangles, it requires less time when the sample sizes are the same.

5 THE FINE-GRAINED HYPERGRAPH CLUSTERING COEFFICIENT

In graph theory, the clustering coefficient is a metric used to measure the tendency of nodes within a graph to form clusters or closely-knit groups. Based on the definition of clustering coefficient in general graphs [19, 48], existing studies [3, 13] propose the definition of clustering coefficient for hypergraphs, which is $3 \times |\Delta|/|\sqcup|$, where Δ is the set of hyper-triangles and \sqcup is the set of open hyper-triangles (i.e., a hyper-triangle in which two hyperedges are disconnected).

While the current definition can be used to measure the localized closeness and redundancy of the hypergraphs, it still has some limitations. Since the semantic meanings vary across different hyper-triangle patterns, there are cases where we prioritize the information conveyed by specific patterns. Therefore, it is crucial to distinguish between different patterns. Motivated by this, we propose a fine-grained hypergraph clustering coefficient, which enables us to adjust the proportion of each pattern according to the domains of the datasets. Note that by setting $\epsilon_p = 1$ for each pattern p , our model can be transformed into the existing model.

Definition 5.1. [**Fine-grained hypergraph clustering coefficient**] Given a hypergraph $G = (V, E)$, the fine-grained clustering coefficient of G is given by $\frac{3 \times \sum_{p=1}^{20} \epsilon_p |\Delta_p|}{|\sqcup|}$, where $\epsilon_p \in [0, 1]$ is a parameter for the hyper-triangle pattern p , $|\Delta_p|$ is the number of hyper-triangles of pattern p in G and $|\sqcup|$ is the number of open hyper-triangles (i.e., a hyper-triangle in which two hyperedges are disconnected) in G .

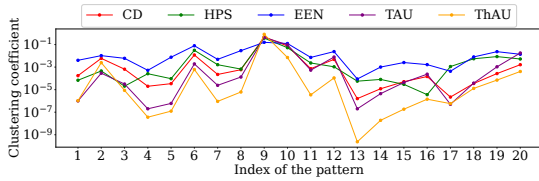
To clearly compare the distribution of different hyper-triangle patterns across datasets from various domains, we calculate the hypergraph clustering coefficient for each pattern in real-world datasets by setting the parameter of the current pattern to 1 and all others to 0. The results are displayed in Figure 4. From these figures, we can observe that the distribution frequency of patterns varies when the data comes from different domains. However, when the data originates from the same domain, the distribution frequency of patterns remains consistent. This indicates that data from specific domains, due to its inherent properties, tends to form certain specific hyper-triangle patterns. In some cases, we may prefer to use specific patterns to reflect the inherent properties of datasets. Hence, we develop the fine-grained clustering coefficient for hypergraphs, which can be flexibly adjusted to more intuitively display the information contained in different patterns.

6 EXPERIMENTS

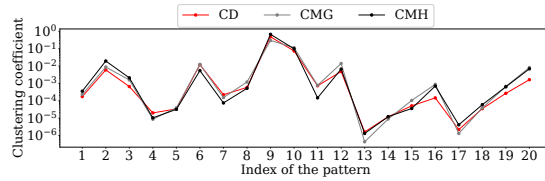
In this section, we evaluate the effectiveness and efficiency of our proposed algorithms.

6.1 Experimental Settings

Datasets. In the experiments, we use the following 11 datasets. The co-authorship [49] domain includes four datasets (coauthor-DBLP (CD), coauthor-geology (CMG), and coauthor-history (CMH)). In these datasets, vertices represent authors, and hyperedges represent publications, with the authors being the vertices included in the hyperedges. The contact domain comprises two datasets



(a) Datasets from different domains



(b) Datasets from same domain

Figure 4: Clustering coefficient comparison

Dataset	V	E	$ e _{max}$	$ e _{avg}$	Λ
HPS	327	7,818	5	2.3	593K
CMH	1,014,734	895,439	25	1.5	1.7M
CPS	242	12704	5	2.4	2.2M
EEU	998	25,027	25	3.4	8.3M
ThAU	125,602	166,999	14	1.9	21.6M
CMG	1,256,385	1,203,895	25	3.1	37.6M
EEN	143	1512	18	3.0	87.8M
CD	1,924,991	2,466,792	25	2.9	125M
TAU	3029	147,222	5	3.4	564M
ThM	176,445	595,749	21	2.4	647M
TMS	1629	170,476	5	3.5	913M

Table 3: Summary of Datasets

(contact-primary (CPS) [50] and contact-high (HPS) [40]). In these datasets, vertices represent individuals, while hyperedges represent relationships between individuals. Email-EU (EEU) [30, 57] and email-Enron (EEN) [25] are two datasets belong to the email domain, where vertices represent email entities, and hyperedges represent either senders or receivers of the emails. The tags domain and threads domain each consist of two datasets. They are tags-ubuntu (TAU), tags-math (TMS), threads-ubuntu (ThAU), and threads-math (ThM). In the datasets from the tags domain, vertices represent tags, and hyperedges represent posts containing these tags. In the threads domain, vertices represent users, and hyperedges represent threads in which users are involved. These datasets are sourced from [5]. Due to the property of hypergraphs, where any two hyperedges cannot contain the same set of vertices, we performed deduplication on the original data. Table 3 provides details about the datasets.

Algorithms. In the experiments, we use *Exact-bs* as the baseline for the exact algorithm and compare its performance with the following five algorithms: *Count-CCC* (for pattern 1), *Count-TCC* (for patterns 2-5), *Count-TTC* (for patterns 6-8), *Count-TTT* (for patterns 9-20), and *Exact-adv*. Here *Exact-adv* is to compute hyper-triangles of all the patterns by combining the search processes of *Count-TTT* and *Count-TTC*, as well as *Count-TCC* and *Count-CCC*, respectively. Subsequently, we conduct a comparison between *Count-TTT*, *Count-DenseTTT* (for patterns 9-16) and *Count-SparseTTT* (for patterns 17-20) in terms of the time required to find *TTT*-class hyper-triangles. For the approximation algorithms, we examine the performance of *Appro-bs* and *Appro-adv*. Similarly, in the parallel algorithms section, we compare the performance between *Par-bs* and *Par-adv*. All the algorithms are implemented in C++, and all experiments are conducted on a Linux machine with an Intel(R) Xeon(R) Platinum 8260L CPU at 2.30 GHz and 256GB of memory.

6.2 Case Study

Publications (Authors)		
e_1 : (1, 2, 3)	e_2 : (3, 4, 5)	e_3 : (2, 3, 4, 6)
e_4 : (2, 3, 4, 7)	e_5 : (4, 8, 9, 10)	

Hyper-triangles (Pattern)		
$\{e_1, e_2, e_3\}$: 12	$\{e_2, e_3, e_4\}$: 10	$\{e_3, e_4, e_5\}$: 10

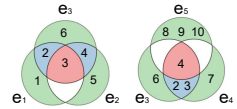


Figure 5: Real data for co-authorship relations

Email account (Senders or Receivers)			
e_1 : (1, 29, 41, 51, 62, 65, 97, 107, 133)			
e_2 : (1, 51)	e_3 : (1, 133)	e_4 : (29, 97)	
e_5 : (41, 97)	e_6 : (29, 65)		

Hyper-triangles (Pattern)		
$\{e_1, e_2, e_3\}$: 5	$\{e_1, e_4, e_5\}$: 5	$\{e_1, e_4, e_6\}$: 5

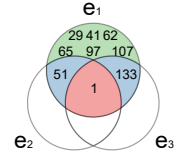


Figure 6: Real data for email correspondence relations

Different hyper-triangle patterns encapsulate distinct meanings, leading to variations in their frequency distribution across various datasets. These variations reveal underlying design principles specific to each dataset. As analyzed in [28], pattern 12 is more prevalent in the co-authorship domain, whereas pattern 10 frequently appears in email correspondence domain. To understand the reasons behind these distributions, we conduct case studies that identify and analyze typical hyper-triangles in these two domains.

In the co-authorship domain, we use the CMH dataset to analyze the hyper-triangle patterns. Figure 5 displays ten authors involved in three hyper-triangles, along with the five publications. The dataset shows that Authors 1-7 have collaborated over twenty times, while Authors 8-10 have more than ten collaborations, indicating they are likely from two different groups. Through hyper-triangle $\{e_1, e_2, e_3\}$, we can see that all three publications are centered around Author 3, and Author 3 maintains a stable co-authorship with Authors 2 and 4. Authors 1, 5, and 6, as the first authors of the three publications, do not have direct connections to each other. In fact, most hyper-triangles of pattern 12 display a similar structure to the one described above, as completing a publication usually involves multiple contributors arranged in a hierarchical structure. A senior researcher often initiates the project (Author 3), followed by contributions from young scholars (Authors 2 and 4). Finally, several student participants also contribute to the publication (Authors 1, 5 and 6). For the collaborative relationships between different groups (e.g., the hyper-triangle $\{e_3, e_4, e_5\}$), we can observe that e_3 and e_4 , which are publications from the same group, share three coauthors. In contrast, the publication e_5 , originating from another group, has only one coauthor in common with e_3 and e_4 . This

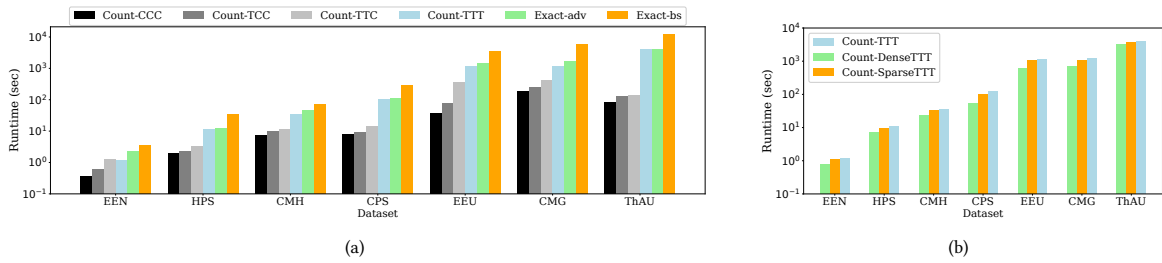


Figure 7: Runtime for exact algorithms.

suggests that inter-group collaborations often depend on specific authors (in this case, Author 4). As a result, the hyper-triangles formed are more likely to align with pattern 10. This reasoning indicates that co-authorship datasets are naturally predisposed to contain a higher number of hyper-triangles of pattern 12.

In the email correspondence domain, Figure 6 show three hyper-triangles from EEN, where hyperedges represent email accounts and the vertices correspond to accounts that have sent or received emails from those accounts. It can clearly be seen that these hyper-triangles contain hyperedge e_1 , and e_1 encompasses the remaining five hyperedges. This is because, in reality, some organizations have "organizer" or admin account that frequently sends and receives emails to and from its member accounts. Consequently, the member accounts tend to interact predominantly with the 'organizer' or admin accounts. Thus, compared to other domains, datasets in the email correspondence domain are more prone to forming hyper-triangles of pattern 5.

6.3 Performance Evaluations

Evaluating exact algorithms. In Figure 7(a), we present the comparison of exact algorithms in 7 datasets. Through Figure 7(a), it can be observed that *Count-CCC*, *Count-TCC*, and *Count-TTC* significantly outperforms *Exact-bs* by up to 140×, 90× and 80×, respectively. This is because these three algorithms can avoid enumerating hyper-triangles of the *TTT* class and, compared to *Exact-bs*, do not require additional time to determine the pattern of the hyper-triangle. While *Count-TTT* requires more time than the other algorithms, it is still at least 3× faster than *Exact-bs*. In addition, *Exact-adv* is capable of computing all hyper-triangles in a shorter time compared to *Exact-bs*, which achieves at least 3× faster than *Exact-bs* in the large datasets. These findings affirm the efficacy of our proposed techniques in reducing the redundant enumeration of vertices within hyperedges.

Evaluating exact algorithms for *TTT* class. For the three algorithms targeting the *TTT* class: *Count-DenseTTT*, *Count-SparseTTT* and *Count-TTT*, we also use the aforementioned 7 datasets to compare their differences. The results are displayed in Figure 7(b). It can be observed that *Count-DenseTTT* has the least runtime. For *Count-DenseTTT*, it avoids traversing open hyper-triangles, thereby achieving about 2× faster than *Count-TTT*. For *Count-SparseTTT*, since finding hyper-triangles of the *SparseTTT* class still requires time to filter open hyper-triangles, the runtime is about 1.5× faster than *Count-TTT*.

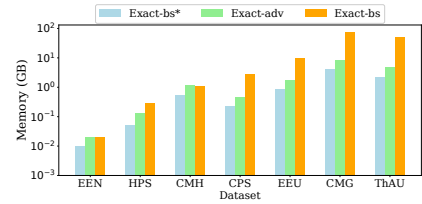


Figure 8: The memory required for exact algorithms.

Evaluating the space for the exact algorithms. Since the source code of *Exact-bs* [28] additionally stores the 2-hop neighbors of each hyperedge, leading to extra space consumption, we improve its code to store only the 1-hop neighbors of each hyperedge without affect the performance (denoted as *Exact-bs**). Figure 8 examines the memory usage of *Exact-bs*, *Exact-bs** and *Exact-adv*. From the results, we observe that *Exact-adv* consistently consumes about 2× the space of *Exact-bs**. This means that compared to the baseline algorithm, the proposed algorithms can find all hyper-triangles with at least 3× the efficiency while only using 2× the space. Besides, if the goal of users is to find hyper-triangles of a specific pattern, the proposed algorithms can even achieve up to 140× faster than the baseline algorithm without increasing space consumption.

Evaluating approximate algorithms. Here we compare the performance of *Appro-bs* and *Appro-adv*. We use $\sum_{p=1}^{20} \frac{|H[p] - \hat{H}[p]|}{20 \times H[p]}$ as the metric to measure the accuracy of the algorithms where $H[p]$ and $\hat{H}[p]$ are the exact and estimated results of the hyper-triangles of pattern p , respectively. We first randomly extract 20% of the hyperedges from the original graph and then perform a comparison on this induced subgraph. By comparing the accuracy of the results obtained by the two algorithms within the same time period, we can determine the differences between the two algorithms.

Figure 9 shows the performance of the two approximation algorithms across four datasets: TAU, ThM, TMS, and CD. Due to their large size, we are unable to get the exact number of hyper-triangles in the original graph. Therefore, we only compute the exact number of hyper-triangles in the subgraph, and based on this, we compare the relative accuracy of the two approximation algorithms. It can be observed that for the first three datasets, when the estimation time is less than 400 seconds, the accuracy of *Appro-adv* is at least 5% higher than *Appro-bs*. Only when the estimation time exceeds 800 seconds does the gap between them narrow. For the CD dataset, the gap is also very significant when the estimation time is less than 80 seconds. The experimental results show that the accuracy of *Appro-adv* is at most 3% higher than *Appro-bs*. Furthermore, the

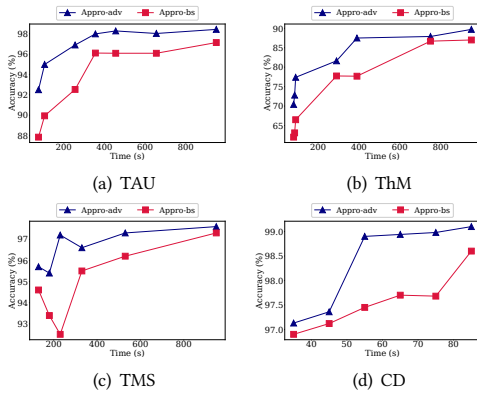


Figure 9: Approximate algorithms' result, $\sigma = 0.2$.

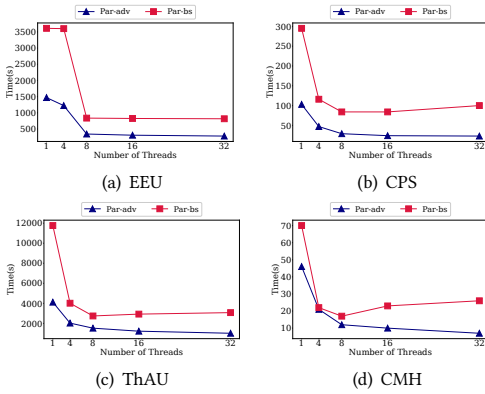


Figure 10: Parallel algorithms' result.

superiority of *Appro-adv* is especially pronounced when the estimation time is brief. These findings clearly confirm the superiority of *Appro-adv* over *Appro-bs*. Additionally, we can observe that for the TAU, ThM, TMS, and CD datasets, *Appro-adv* can estimate the number of hyper-triangles within 400 seconds with an accuracy close to 90%. In contrast, the exact algorithm requires more than 10,000 seconds to obtain accurate results. This indicates that when users prioritize efficiency over accuracy, the approximation algorithm, compared to the exact algorithm, has a clear advantage.

Evaluating parallel algorithms. In this section, we compare the runtime of *Par-bs* and *Par-adv* under different numbers of threads in 6 datasets. The experimental results are presented in Figure 10. For *Par-bs*, across all datasets, the runtime decreases gradually with an increasing number of threads. However, when the number of threads exceeds 16, the runtime begins to increase. In contrast, for *Par-adv*, as the number of threads increases, the runtime of the algorithm first decreases rapidly and then stabilizes. When we use 32 threads, *Par-adv* achieves at least 3× faster than *Par-bs*.

7 RELATED WORK

Triangle counting. In general graphs, triangle represents the smallest non-trivial cohesive structure, and there are extensive studies on counting triangles in the literature [1, 2, 6, 11, 12, 21, 27, 33, 42, 44, 46, 59]. In [11], Chiba and Nishizeki propose an enumeration-based algorithm to find triangles that orders the vertices by degree and

processes each edge only once, using its lower-degree vertex. Some existing works [1, 2] focus on edge sampling strategies for estimating triangle counts in a graph stream. Building on this, [33, 42] propose a wedge-based sampling method to estimate the number of triangles. In [59], an orientation-based algorithm is proposed to improve the efficiency of finding triangles. [21] proposes a novel lightweight graph preprocessing method for triangle counting using GPU. However, existing triangle counting techniques cannot be directly used for hyper-triangle counting since there are significant structural differences between triangle and hyper-triangle.

Hypergraph analysis. Hypergraphs naturally represent group interactions and are widely used in various fields such as social networks [31, 54], bioinformatics [22], recommendation systems [8, 32], and LLM [16, 38]. Existing research extensively explores motifs in hypergraphs [7, 62]. Hypergraph clustering coefficients are studied in [4, 58] to express the overall connectivity among hyperedges and can be used to measure the localized closeness and redundancy of hypergraphs. [61] proposes sampling-and-estimating frameworks for counting three types of triangles over hypergraph streams. In [26, 28], researchers focus on finding the connectivity patterns among three hyperedges and reveal that the distribution frequency of these patterns varies across different data domains. Although the algorithms in [26, 28] can be adopted to solve the hyper-triangle computation problem, our proposed techniques outperform existing solutions as validated in our experiments.

8 CONCLUSION

In this paper, we investigate the hyper-triangle computing problem. We present a two-step framework that efficiently searches for hyper-triangles of specific patterns, supplemented by a parallel version to handle large datasets. Additionally, we propose approximation algorithms for counting the number of hyper-triangles in hypergraphs. Furthermore, we introduce a fine-grained model of the hypergraph clustering coefficient, offering greater flexibility for application across diverse datasets. Through extensive experiments on real datasets, we demonstrate the significant superiority of our algorithms over state-of-the-art approaches.

In the future, we will explore how to extend our approach to search for larger hypergraph patterns. Firstly, since hyper-triangles are a specific case of hyper-cliques, the proposed algorithms can be adapted to identify hyper-cliques by counting the number of hyper-triangles associated with each hyperedge, enabling us to filter hyperedges and narrow the search range. Additionally, in general graphs, triangles serve as fundamental building blocks for the truss model. By counting triangles, we can derive truss structures that facilitate community detection. Similarly, we can extract pattern-based hyper-truss structures from hypergraphs by analyzing hyper-triangles of various patterns. This approach will improve community detection in hypergraphs and offer deeper insights into the intricate interconnections within these complex data structures.

9 ACKNOWLEDGMENT

Kai Wang is the corresponding author. Kai Wang is supported by NSFC 62302294 and U2241211. Wenjie Zhang is supported by ARC FT210100303. Ruijia Wu is supported by NSFC 12301382. Xuemin Lin is supported by NSFC U2241211.

REFERENCES

- [1] Nesreen K Ahmed, Nick Duffield, Jennifer Neville, and Ramana Kompella. 2014. Graph sample and hold: A framework for big-graph analytics. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 1446–1455.
- [2] Nesreen K Ahmed, Nick Duffield, Theodore Willke, and Ryan A Rossi. 2017. On sampling from massive graph streams. *arXiv preprint arXiv:1703.02625* (2017).
- [3] Sinan G Aksoy, Cliff Joslyn, Carlos Ortiz Marrero, Brenda Praggastis, and Emilie Purvine. 2020. Hypernetwork science via high-order hypergraph walks. *EPJ Data Science* 9, 1 (2020), 16.
- [4] Ilya Amburg, Nate Veldt, and Austin Benson. 2020. Clustering in graphs and hypergraphs with categorical edge labels. In *Proceedings of The Web Conference 2020*. 706–717.
- [5] Austin R Benson, Rediet Abebe, Michael T Schaub, Ali Jabdabaie, and Jon Kleinberg. 2018. Simplicial closure and higher-order link prediction. *Proceedings of the National Academy of Sciences* 115, 48 (2018), E11221–E11230.
- [6] JW Berry, I Fosvedt, Daniel Nordman, Cynthia A Phillips, and Alyson G Wilson. 2011. Listing triangles in expected linear time on power law graphs with exponent at least 7.3. *Sandia National Laboratories, Tech. Rep. SAND2010-4474c* (2011).
- [7] Alain Bretto. 2013. Hypergraph theory. *An introduction. Mathematical Engineering. Cham: Springer* 1 (2013).
- [8] Jiajun Bu, Shulong Tan, Chun Chen, Can Wang, Hao Wu, Lijun Zhang, and Xiaofei He. 2010. Music recommendation by unified hypergraph: combining social media information and music content. In *Proceedings of the 18th ACM international conference on Multimedia*. 391–400.
- [9] Xinwei Cai, Xiangyu Ke, Kai Wang, Lu Chen, Tianming Zhang, Qing Liu, and Yunjun Gao. 2023. Efficient Temporal Butterfly Counting and Enumeration on Temporal Bipartite Graphs. *arXiv preprint arXiv:2306.00893* (2023).
- [10] James Cheng, Yiping Ke, Shumo Chu, and M Tamer Özsu. 2011. Efficient core decomposition in massive networks. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 51–62.
- [11] Norishige Chiba and Takao Nishizeki. 1985. Arboricity and subgraph listing algorithms. *SIAM Journal on computing* 14, 1 (1985), 210–223.
- [12] Shumo Chu and James Cheng. 2011. Triangle listing in massive networks and its applications. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. 672–680.
- [13] Ernesto Estrada and Juan A Rodríguez-Velázquez. 2006. Subgraph centrality and clustering in complex hyper-networks. *Physica A: Statistical Mechanics and its Applications* 364 (2006), 581–594.
- [14] Yixiang Fang, Yixing Yang, Wenjie Zhang, Xuemin Lin, and Xin Cao. 2020. Effective and efficient community search over large heterogeneous information networks. *Proceedings of the VLDB Endowment* 13, 6 (2020), 854–867.
- [15] Song Feng, Emily Heath, Brett Jefferson, Cliff Joslyn, Henry Kvinge, Hugh D Mitchell, Brenda Praggastis, Amie J Eisfeld, Amy C Sims, Larissa B Thackray, et al. 2021. Hypergraph models of biological networks to identify genes critical to pathogenic viral response. *BMC bioinformatics* 22, 1 (2021), 287.
- [16] Yifan Feng, Chengwu Yang, Xingliang Hou, Shaoyi Du, Shihui Ying, Zongze Wu, and Yue Gao. 2024. Beyond Graphs: Can Large Language Models Comprehend Hypergraphs? *arXiv preprint arXiv:2410.10083* (2024).
- [17] Zhongqiang Gao, Chuanqi Cheng, Yanwei Yu, Lei Cao, Chao Huang, and Junyu Dong. 2022. Scalable motif counting for large-scale temporal graphs. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2656–2668.
- [18] Sathyanarayanan Gopalakrishnan and Swaminathan Venkatraman. 2024. Prediction of influential proteins and enzymes of certain diseases using a directed unimodular hypergraph. *Mathematical Biosciences and Engineering* 21, 1 (2024), 325–345.
- [19] Oded Green and David A Bader. 2013. Faster clustering coefficient using vertex covers. In *2013 International Conference on Social Computing*. IEEE, 321–330.
- [20] Ervin Györi. 2006. Triangle-free hypergraphs. *Combinatorics, Probability and Computing* 15, 1-2 (2006), 185–191.
- [21] Lin Hu, Lei Zou, and Yu Liu. 2021. Accelerating triangle counting on GPU. In *Proceedings of the 2021 International Conference on Management of Data*. 736–748.
- [22] TaeHyun Hwang, Ze Tian, Rui Kuangy, and Jean-Pierre Kocher. 2008. Learning on weighted hypergraphs to integrate protein interactions and gene expressions for cancer outcome prediction. In *2008 Eighth IEEE International Conference on Data Mining*. IEEE, 293–302.
- [23] Masaaki Inoue, Thong Pham, and Hidetoshi Shimodaira. 2022. A hypergraph approach for estimating growth mechanisms of complex networks. *IEEE Access* 10 (2022), 35012–35025.
- [24] Madhav Jha, C Seshadhri, and Ali Pinar. 2015. Path sampling: A fast and provable method for estimating 4-vertex subgraph counts. In *Proceedings of the 24th international conference on world wide web*. 495–505.
- [25] Bryan Klimt and Yiming Yang. 2004. The enron corpus: A new dataset for email classification research. In *European conference on machine learning*. Springer, 217–226.
- [26] Yunbum Kook, Jihoon Ko, and Kijung Shin. 2020. Evolution of real-world hypergraphs: Patterns and models without oracles. In *2020 IEEE International Conference on Data Mining (ICDM)*. IEEE, 272–281.
- [27] Matthieu Latapy. 2008. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical computer science* 407, 1-3 (2008), 458–473.
- [28] Geon Lee, Jihoon Ko, and Kijung Shin. 2020. Hypergraph motifs: Concepts, algorithms, and discoveries. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2256–2269.
- [29] Geon Lee and Kijung Shin. 2023. Temporal hypergraph motifs. *Knowledge and Information Systems* 65, 4 (2023), 1549–1586.
- [30] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2005. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. 177–187.
- [31] Dong Li, Zhiming Xu, Sheng Li, and Xin Sun. 2013. Link prediction in social networks based on hypergraph. In *Proceedings of the 22nd international conference on world wide web*. 41–42.
- [32] Lei Li and Tao Li. 2013. News recommendation via hypergraph learning: encapsulation of user behavior and news content. In *Proceedings of the sixth ACM international conference on Web search and data mining*. 305–314.
- [33] Yongsub Lim and U Kang. 2015. Mascot: Memory-efficient and accurate sampling for counting local triangles in graph streams. In *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*. 685–694.
- [34] Qing Liu, Minjun Zhao, Xin Huang, Jianliang Xu, and Yunjun Gao. 2020. Truss-based community search over large directed graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2183–2197.
- [35] Jose Lugo-Martinez, Daniel Zeberg, Thomas Gaudelet, Noël Malod-Dognin, Natasa Przulj, and Predrag Radivojac. 2021. Classification in biological networks with hypergraphlet kernels. *Bioinformatics* 37, 7 (2021), 1000–1007.
- [36] Chenhao Ma, Reynold Cheng, Laks VS Lakshmanan, Tobias Grubenmann, Yixiang Fang, and Xiaodong Li. 2019. Linc: a motif counting algorithm for uncertain graphs. *Proceedings of the VLDB Endowment* 13, 2 (2019), 155–168.
- [37] Yingjun Ma and Yuanyuan Ma. 2022. Hypergraph-based logistic matrix factorization for metabolite–disease interaction prediction. *Bioinformatics* 38, 2 (2022), 435–443.
- [38] Yuren Mao, Yuhang Ge, Yijiang Fan, Wenyi Xu, Yu Mi, Zhonghao Hu, and Yunjun Gao. 2025. A survey on lora of large language models. *Frontiers of Computer Science* 19, 7 (2025), 197605.
- [39] Dror Marcus and Yuval Shavitt. 2010. Efficient counting of network motifs. In *2010 IEEE 30th International Conference on Distributed Computing Systems Workshops*. IEEE, 92–98.
- [40] Rossana Mastrandrea, Julie Fournet, and Alain Barrat. 2015. Contact patterns in a high school: a comparison between data collected using wearable sensors, contact diaries and friendship surveys. *PLoS one* 10, 9 (2015), e0136497.
- [41] Jiayi Nie, Sam Spiro, and Jacques Verstraëte. 2021. Triangle-free subgraphs of hypergraphs. *Graphs and Combinatorics* 37 (2021), 2555–2570.
- [42] Aduri Pavan, Kanat Tangwongsan, Srikanta Tirhappura, and Kun-Lung Wu. 2013. Counting and sampling triangles from a graph stream. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1870–1881.
- [43] Mahmudur Rahman, Mansurul Alam Bhuiyan, and Mohammad Al Hasan. 2014. Graft: An efficient graphlet counting method for large graph analysis. *IEEE Transactions on Knowledge and Data Engineering* 26, 10 (2014), 2466–2478.
- [44] Kaushik Ravichandran, Akshara Subramaniasivam, PS Aishwarya, and NS Kumar. 2023. Fast exact triangle counting in large graphs using SIMD acceleration. In *Advances in Computers*. Vol. 128. Elsevier, 233–250.
- [45] Sanjukta Roy and Balaraman Ravindran. 2015. Measuring network centrality using hypergraphs. In *Proceedings of the 2nd ACM IKDD Conference on Data Sciences*. 59–68.
- [46] Thomas Schank and Dorothea Wagner. 2005. Finding, counting and listing all triangles in large graphs, an experimental study. In *International workshop on experimental and efficient algorithms*. Springer, 606–609.
- [47] Comandur Seshadhri, Ali Pinar, and Tamara G Kolda. 2013. Triadic measures on graphs: The power of wedge sampling. In *Proceedings of the 2013 SIAM international conference on data mining*. SIAM, 10–18.
- [48] Comandur Seshadhri, Ali Pinar, and Tamara G Kolda. 2014. Wedge sampling for computing clustering coefficients and triangle counts on large graphs. *Statistical Analysis and Data Mining: The ASA Data Science Journal* 7, 4 (2014), 294–307.
- [49] Arnab Sinha, Zhihong Shen, Yang Song, Hao Ma, Darrin Eide, Bo-June Hsu, and Kuansan Wang. 2015. An overview of microsoft academic service (mas) and applications. In *Proceedings of the 24th international conference on world wide web*. 243–246.
- [50] Juliette Stehlé, Nicolas Voirin, Alain Barrat, Ciro Cattuto, Lorenzo Isella, Jean-François Pinton, Marco Quaghiotto, Wouter Van den Broeck, Corinne Régis, Bruno Lina, et al. 2011. High-resolution measurements of face-to-face contact patterns in a primary school. *PLoS one* 6, 8 (2011), e23176.

- [51] Jia Wang and James Cheng. 2012. Truss decomposition in massive networks. *arXiv preprint arXiv:1205.6693* (2012).
- [52] Jingjing Wang, Yanhao Wang, Wenjun Jiang, Yuchen Li, and Kian-Lee Tan. 2020. Efficient sampling algorithms for approximate temporal motif counting. In *Proceedings of the 29th ACM international conference on information & knowledge management*. 1505–1514.
- [53] Hanrui Wu, Yuguang Yan, and Michael Kwok-Po Ng. 2022. Hypergraph collaborative network on vertices and hyperedges. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45, 3 (2022), 3245–3258.
- [54] Dingqi Yang, Bingqing Qu, Jie Yang, and Philippe Cudre-Mauroux. 2019. Revisiting user mobility and social relationships in lbsns: a hypergraph embedding approach. In *The world wide web conference*. 2147–2157.
- [55] Hongpeng Yang, Yijie Ding, Jijun Tang, and Fei Guo. 2021. Identifying potential association on gene-disease network via dual hypergraph regularized least squares. *BMC genomics* 22 (2021), 1–16.
- [56] Yixing Yang, Yixiang Fang, Xuemin Lin, and Wenjie Zhang. 2020. Effective and efficient truss computation over large heterogeneous information networks. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 901–912.
- [57] Hao Yin, Austin R Benson, Jure Leskovec, and David F Gleich. 2017. Local higher-order graph clustering. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 555–564.
- [58] Se-eun Yoon, Hyungseok Song, Kijung Shin, and Yung Yi. 2020. How much and when do we need higher-order information in hypergraphs? a case study on hyperedge prediction. In *Proceedings of The Web Conference 2020*. 2627–2633.
- [59] Michael Yu, Lu Qin, Ying Zhang, Wenjie Zhang, and Xuemin Lin. 2020. Aot: Pushing the efficiency boundary of main-memory triangle listing. In *Database Systems for Advanced Applications: 25th International Conference, DASFAA 2020, Jeju, South Korea, September 24–27, 2020, Proceedings, Part II 25*. Springer, 516–533.
- [60] Fangyuan Zhang, Dechuang Chen, Sibowang, Yin Yang, and Junhao Gan. 2023. Scalable Approximate Butterfly and Bi-triangle Counting for Large Bipartite Networks. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–26.
- [61] Lingling Zhang, Zhiwei Zhang, Guoren Wang, Ye Yuan, and Kangfei Zhao. 2023. Efficiently Counting Triangles for Hypergraph Streams by Reservoir-Based Sampling. *IEEE Transactions on Knowledge and Data Engineering* (2023).
- [62] Dengyong Zhou, Jiayuan Huang, and Bernhard Schölkopf. 2006. Learning with hypergraphs: Clustering, classification, and embedding. *Advances in neural information processing systems* 19 (2006).