Efficient Hyper-truss Decomposition over Hypergraphs

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

Kai Wang
Antai College of Economics and Management, Shanghai Jiao Tong University; Data-Driven Management Decision Making Lab, Shanghai Jiao Tong University
Shanghai, China
w.kai@sjtu.edu.cn

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

Xuemin Lin
The Chinese University of Hong Kong, Shenzhen
Shenzhen, China
xuemin.lin@gmail.com

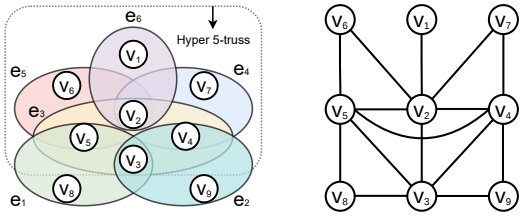
ABSTRACT

Cohesive subgraph mining in hypergraphs has recently attracted increasing research attention due to its broad applicability in domains such as social networks, co-authorship networks, and recommendation systems. An important model, the hyper k-truss, is defined as a maximal cohesive subgraph in which each hyperedge is contained in at least $(k-2)$ hyper-triangles (i.e., structures formed by three pairwise connected hyperedges). In this paper, we study the problem of hyper-truss decomposition, which aims to identify all hyper k-trusses for $k \geq 0$. Due to the complex structure of hyper-triangles, the existing hyperedge-aware framework for hyper-truss decomposition incurs extra computational cost by traversing open hyper-triangles (i.e., hyper-triangles in which two hyperedges are not connected). Moreover, existing strategies enumerate all supporting hyper-triangles for each peeled hyperedge, which substantially limits overall efficiency. To address these issues, we propose a vertex-aware framework that leverages vertex-level connectivity among hyperedges. Under this framework, we design a vertex-oriented counting strategy to completely eliminate the traversal of open hyper-triangles during the counting phase and a vertex-based state propagation method to minimize the number of hyper-triangles enumerated in the peeling phase. Extensive experiments on eleven real-world datasets demonstrate the effectiveness and efficiency of our approach.

PVLDB Reference Format:
Haozhe Yin, Kai Wang, Wenjie Zhang, and Xuemin Lin. Efficient Hyper-truss Decomposition over Hypergraphs. PVLDB, 19(10): 2549 - 2562, 2026.
doi:10.14778/3828612.3828614

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

Proceedings of the VLDB Endowment, Vol. 19, No. 10 ISSN 2150-8097.
doi:10.14778/3828612.3828614



(a) A hypergraph (b) A traditional graph

Figure 1: Comparison of hypergraph and traditional graph.

1 INTRODUCTION

In many real-world applications such as neuroscience [13, 36], social networks [57, 58], and co-authorship collaboration networks [4, 12, 18, 32, 47], hypergraphs are widely used to model systems that exhibit group-based or higher-order relationships among entities. A hypergraph is a generalization of a traditional graph in which a hyperedge can connect an arbitrary number of vertices. This property allows hypergraphs to capture complex networks with group-level interactions. For example, Figure 1(a) illustrates a hypergraph where each hyperedge represents a developer, and the vertices denote the projects to which the developer contributes. In contrast, the traditional graph shown in Figure 1(b) can only depict pairwise co-developer relationships between projects. Consequently, mining cohesive subgraphs in hypergraphs has recently become an active research topic. In the literature, several cohesive models are proposed, such as the degree-based k-core [35], k-hypercore [29], Nbr-k-core [3], $CoCore$ [30], and hyper k-truss [34], among which the hyper k-truss model has received considerable attention. By leveraging the properties of hyper-triangles (i.e., three pairwise connected hyperedges), the hyper k-truss effectively captures higher-order structural cohesiveness in hypergraphs. Specifically, given a positive integer k, a hyper k-truss is defined as a cohesive subgraph in which each hyperedge participates in at least $(k-2)$ hyper-triangles. The hyper-truss number of a hyperedge e is the largest k such that e belongs to a hyper k-truss. For instance, in Figure 1(a), the subgraph composed of hyperedges e_3, e_4, e_5, and e_6 forms a hyper 5-truss, as each of these hyperedges is contained in three hyper-triangles. Their hyper-truss number is 5, since no hyper 6-truss includes them.

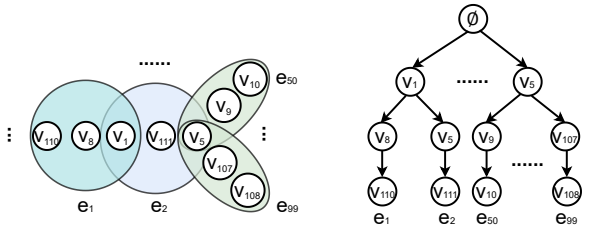
In this paper, we study the hyper-truss decomposition problem, which computes the hyper-truss number for each hyperedge in a hypergraph. In traditional graphs, extensive studies focus on truss decomposition [2, 17], which constructs a hierarchy of k-trusses where each edge participates in at least $(k-2)$ triangles. However, truss decomposition cannot be applied to hypergraphs, as the traditional triangle structure does not exist in hypergraphs. Consequently, it is essential to develop efficient algorithms for hyper-truss decomposition. The importance of hyper-truss decomposition has been demonstrated across various real-world applications, and several representative examples are presented below.

• *Co-authorship networks.* In co-authorship networks, researchers from the same institution or those who collaborate closely often form cohesive groups. Such strong and repeated collaborations tend to induce dense hyper-triangles (i.e., hyper-triangles in which all three hyperedges share common vertices). In contrast, researchers who collaborate less frequently are more likely to be connected through sparse hyper-triangles (i.e., hyper-triangles in which common vertices exist only between pairs of hyperedges). By applying hyper-truss decomposition, loosely connected structures are progressively filtered out, while densely connected collaboration groups are extracted and further organized into hierarchically nested substructures, enabling the identification of core research teams with intensive collaboration. This hierarchical decomposition provides valuable insights into collaboration patterns and helps identify influential authors within co-authorship networks.

• *Filtering spam and malicious accounts.* In communication networks, users within the same organization often exhibit an administrative relationship, where a central organizer or administrator maintains consistent interactions with all member accounts. Such structured and repeated interactions make these accounts more likely to form dense hyper-triangles. In contrast, fake accounts or coordinated malicious actors typically lack persistent and densely connected interaction patterns, and are therefore less likely to form dense hyper-triangles. By applying hyper-truss decomposition, accounts involved in stable and coordinated communications exhibit high hyper-truss values, whereas fake or loosely coordinated accounts are removed early during the decomposition process.

Existing techniques. The existing truss decomposition framework [48, 56, 59] consists of two phases: a counting phase, which counts the number of triangles each edge is involved in, and a peeling phase, which iteratively removes edges e with the minimum support and assigns the truss number to e. In [34], Qin et al. propose an efficient counting algorithm *HTC-PF+* that constructs a prefix forest to order vertices within each hyperedge and merges hyperedges sharing common prefixes into a compact hierarchical structure to search for sparse hyper-triangles. This approach effectively reduces redundant vertex intersection checks among overlapping hyperedges, thereby accelerating the search process. By employing *HTC-PF+* as the counting phase, they further develop a hyper-truss decomposition algorithm to compute the hyper-truss number for each hyperedge.

Motivations and Challenges. Existing hyper k-truss models [34] are restricted to sparse hyper-triangles and therefore overlook higher-order interactions captured by dense hyper-triangles. As a result, hyperedges with different levels of such interactions may be assigned similar hyper-truss values, reducing the discriminative power of hyper-truss decomposition. Moreover, hyper-triangles exhibit more complex structures than traditional triangles due to the varying degrees of overlap among their vertices, which makes identifying all hyper-triangles in a hypergraph difficult. In the counting phase, existing approaches [25, 34, 52] typically spend considerable time filtering open hyper-triangles (i.e., hyper-triangles in which two hyperedges are disconnected), resulting in high computational costs. For example, consider the hypergraph shown in Figure 2(a). *HTC-PF+* [34] constructs a prefix forest to search for sparse hyper-triangles. However, although hyperedges e_1 and e_2 cannot form any hyper-triangle with e_{50} to e_{99}, the algorithm may still traverse the corresponding nodes in the prefix forest during the search process. Furthermore, during the peeling phase, the existing method employs a straightforward enumeration strategy, in which all supporting hyper-triangles for each removed hyperedge must be revisited to update the support values of its neighboring hyperedges. This process renders the peeling phase time-consuming and substantially limits the overall efficiency of hyper-truss decomposition. In this work, we generalize the hyper k-truss model by incorporating all types of hyper-triangles and aim to develop efficient hyper-truss decomposition algorithms, which face the following challenges.



(a) The original graph (b) The prefix forest

Figure 2: Limitation of the existing algorithm.

- During the counting phase, it is challenging to efficiently identify all hyper-triangles while avoiding unnecessary traversal of open hyper-triangles.
- When performing hyperedge removal operations during the peeling stage, it is challenging to avoid enumerating hyper-triangles while updating the support values of the affected hyperedges.

Our approaches. In this paper, we propose a vertex-aware framework for efficient hyper-truss decomposition, which leverages vertex-level connectivity between hyperedges to accelerate hyper-triangle counting and optimize the peeling process. To address Challenge 1, we adopt a vertex-oriented counting strategy that organizes the search for hyper-triangles around shared vertices. Specifically, we first construct hyperwedges by pairing connected hyperedges (e_i, e_j) and then divide the vertices of each hyperwedge into three disjoint subsets: $(e_i \setminus e_j)$, $(e_j \setminus e_i)$, and $(e_i \cap e_j)$. By utilizing the vertices within these subsets, we further identify additional hyperwedges that share common vertices with the current one, ensuring that all enumerated combinations form valid hyper-triangles while effectively avoiding the traversal of open hyper-triangles. Concretely, for the hyperwedge (e_1, e_2) in Figure 2(a), vertex v_5 belongs to the subset $(e_2 \setminus e_1)$. In contrast, for other hyperwedges such as (e_2, e_{50}), the same vertex v_5 appears in a different partition, i.e., $(e_2 \cap e_{50})$. Therefore, when (e_1, e_2) attempts to identify another hyperedge via v_5 to construct a hyper-triangle, such hyperwedges are filtered out, thus preventing traversal of open hyper-triangles.

To address Challenge 2, we design an optimized peeling strategy with vertex-based state propagation, which records the removal information of hyperedges on their incident vertices to reduce redundant revisits to hyper-triangles. In particular, when a hyperedge e_i is removed during the peeling phase, its incident vertices store this removal state. For each neighboring hyperedge e_j, if e_i and e_j co-occur only in hyper-triangles $T(\{e_i, e_j, e_k\})$ where $|e_i \cap e_j \cap e_k| = 1$, the proposed approach infers the current number of hyper-triangles containing both e_i and e_j from the removal information stored on their shared vertices ($v \in e_i \cap e_j$). This inferred count is then used to directly update the support of e_j, thereby reducing explicit traversals of hyper-triangles and significantly improving the efficiency of the peeling process.

Contributions. In general, our main contributions of this paper are listed as follows.

- We propose a generalized hyper k-truss model that incorporates all hyper-triangles, and design an efficient vertex-aware framework for hyper-truss decomposition in hypergraphs.
- We propose a vertex-oriented counting strategy that leverages vertex-level connectivity between hyperedges to efficiently enumerate valid hyper-triangles and eliminate redundant traversal over open hyper-triangles during the counting phase.
- We design an optimized peeling strategy with vertex-based state propagation, which records the removal states of hyperedges on their incident vertices to reduce unnecessary revisits to hyper-triangles during the peeling phase.
- Experimental results on 11 real-world hypergraphs show that the proposed hyper-truss decomposition algorithms outperform the baseline method, while the proposed hyper-triangle counting techniques achieve higher efficiency than the state-of-the-art approaches [25, 34].

Organization. The rest of this paper is organized as follows. Section 2 provides the problem definition and summarizes existing solutions. Section 3 describes the proposed hyper-truss decomposition framework, incorporating an advanced hyper-triangle counting strategy and an optimized peeling process. Section 4 presents an analysis of the proposed framework. Experimental evaluations are reported in Section 5. Section 6 discusses related literature, and Section 7 concludes the paper.

2 PROBLEM DEFINITION

Our problem is defined on a simple and undirected hypergraph $G = (V, E)$, in which V denotes the set of vertices and $E = \{e_1, \ldots, e_{|E|}\}$ represents the set of hyperedges. Each hyperedge $e_i \in E$ is defined as an unordered subset of vertices. The capacity of a hyperedge e_i, denoted by $|e_i|$, refers to the number of vertices contained in e_i. The set of all hyperedges connected to e_i is denoted by $N_{e_i} = \{e_j \in E : e_i \cap e_j \neq \emptyset\}$. Accordingly, the degree of a hyperedge e_i is defined as $|N_{e_i}|$. For each vertex $v \in V$, we denote by E_v the set of hyperedges that contain v. Throughout this paper, we assume that no two distinct hyperedges share exactly the same set of vertices.

Definition 2.1 (Hyperwedge). Given a hypergraph $G = (V, E)$, a hyperwedge $\wedge_{ij}$ is a path formed by two hyperedges e_i and e_j such that $e_i \cap e_j \neq \emptyset$.

We denote the set of all hyperwedges as $\wedge$. To avoid duplication, we assign each hyperwedge $\wedge_{ij}$ in G an order denoted by $O(\wedge_{ij})$.

Notations	Description
$G = (V, E)$	A hypergraph with vertices V and hyperedges E
$E = \{e_1, ..., e_{\|E\|}\}$	The set of hyperedges
E_v	The set of all hyperedges containing the vertex v
$\mathcal{D}/\mathcal{C}$	The average degree/capacity of hyperedges
$\wedge$	The set of hyperwedges
$\wedge_{ij}$	A hyperwedge composed of e_i and e_j
Ω_{ij}	The set of common vertices between e_i and e_j
$\Omega^+_{ij}/\Omega^-_{ij}$	The set of vertices in e_i/e_j but not in e_j/e_i
$B_{ij}/B^+_{ij}/B^-_{ij}$	The bitmap for the set $\Omega_{ij}/\Omega^+_{ij}/\Omega^-_{ij}$
$\phi(e_i)$	The hyper-truss number of a hyperedge e_i
$sup(e_i)$	The support value of a hyperedge e_i
N_{e_i}	The set of neighbors of a hyperedge e_i
H_k	The hyper k-truss of G
$\mathcal{T}_d[e_i]/\mathcal{T}_d[\wedge_{ij}]$	The number of dense hyper-triangles containing hyperedge e_i/hyperwedge $\wedge_{ij}$
$\mathcal{T}_s[e_i]/\mathcal{T}_s[\wedge_{ij}]$	The number of sparse hyper-triangles containing hyperedge e_i/hyperwedge $\wedge_{ij}$
$T(\{e_i, e_j, e_k\})$	A hyper-triangle composed of e_i, e_j and e_k

Table 1: Notations

Given two hyperwedges $\wedge_{ij}$ and $\wedge_{kl}$, we define $O(\wedge_{ij}) > O(\wedge_{kl})$ when $i > k$, or when $i = k$ and $j > l$. For two hyperedges e_i and e_j, we denote the set of common vertices between them as Ω_{ij}, while Ω^+_{ij} (resp. Ω^-_{ij}) denotes the set of vertices in e_i (resp. e_j) but not in e_j (resp. e_i).

Definition 2.2 (Projected graph). Given a hypergraph $G = (V, E)$, the projected graph of G is defined as $\overline{G} = (E, \wedge)$, where each hyperedge of G acts as a vertex in $\overline{G}$, and two vertices e_i and e_j in $\overline{G}$ are connected if and only if $e_i \cap e_j \neq \emptyset$ in G.

Definition 2.3 (Hyper-triangle). Given a hypergraph $G = (V, E)$, a hyper-triangle $T(\{e_i, e_j, e_k\})$ is a subgraph in G consisting of three hyperedges $e_i, e_j, e_k \in E$ such that $e_i \cap e_j \neq \emptyset$, $e_i \cap e_k \neq \emptyset$, and $e_j \cap e_k \neq \emptyset$. Moreover, $T(\{e_i, e_j, e_k\})$ is a dense hyper-triangle if and only if $e_i \cap e_j \cap e_k \neq \emptyset$; otherwise, it is a sparse hyper-triangle, for which $e_i \cap e_j \cap e_k = \emptyset$.

Given three hyperedges e_i, e_j, and e_k, the intersections and differences among them partition the vertex set into seven disjoint subsets. Depending on whether each subset is empty or not, and after excluding equivalent configurations, we obtain 26 distinct patterns, as shown in Figure 3. Among them, patterns 1–20 are hyper-triangles in which all three hyperedges are pairwise connected, while patterns 21–26 are open hyper-triangles, where two of the hyperedges are not connected. In addition, for a hyper-triangle $T(\{e_i, e_j, e_k\})$, we define its overlap degree as $\varphi(T) = |e_i \cap e_j \cap e_k|$, which measures the number of vertices shared by all three hyperedges. According to the definition, patterns 1–16 correspond to dense hyper-triangles (i.e., $\varphi(T) \geq 1$), while patterns 17–20 correspond to sparse hyper-triangles (i.e., $\varphi(T) = 0$).

Definition 2.4 (Support). Given a hypergraph $G = (V, E)$, the support of a hyperedge $e \in E$ in G, denoted by $sup(e)$, is the number of hyper-triangles which contain hyperedge e.

Definition 2.5 (Hyper k-truss). Given a hypergraph $G = (V, E)$, a hyper k-truss in G, denoted by H_k, is a maximal subgraph such that $\forall e \in H_k$, $sup(e) \geq k - 2$.

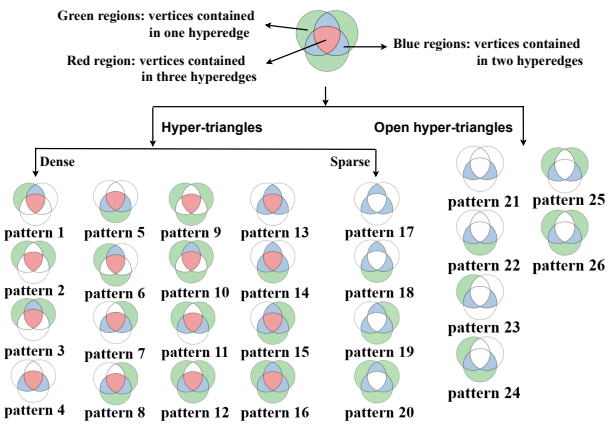


Figure 3: Illustration of all hyper-triangle patterns.

Definition 2.6 (Hyper-truss number). Given a hypergraph $G = (V, E)$, the hyper-truss number of a hyperedge $e \in E$, denoted by $\phi(e)$, is defined as the positive integer k such that there exists a hyper k-truss containing e but no hyper $(k+1)$-truss containing e.

Problem statement. Given a hypergraph $G = (V, E)$, the hyper-truss decomposition problem aims to compute the hyper-truss number $\phi(e)$ for each hyperedge $e \in E$.

Existing solutions. In [34], Qin et al. propose a counting algorithm, *HTC-PF+*, for sparse hyper-triangles. This algorithm represents all hyperedges in a prefix forest to compactly capture shared vertex prefixes and performs a depth-first search to efficiently identify sparse hyper-triangles. By employing *HTC-PF+* as the hyper-triangle counting phase and combining it with a simple peeling procedure, they develop a hyper-truss decomposition algorithm. However, in this paper, we generalize the hyper k-truss model by considering all types of hyper-triangles, including both dense and sparse ones. Since *HTC-PF+* focuses only on sparse hyper-triangles, we replace their counting phase with an enumeration-based approach [25, 52] capable of identifying all types of hyper-triangle. In this way, we obtain a baseline algorithm.

The details of the baseline algorithm are presented in Algorithm 1. In the preprocessing step (Lines 4-7), the algorithm iterates over the vertices of each hyperedge to construct the projected graph, through which the neighbors of every hyperedge are identified. In the counting phase (Lines 8–12), for each hyperedge e_i, the algorithm examines all neighboring hyperedge pairs $\{e_j, e_k\}$ and computes the supports of e_i, e_j, and e_k. Finally, in the peeling phase (Lines 13-19), for each k starting from $k = 2$, the algorithm removes every hyperedge e_i whose support is at most $k-2$, since such a hyperedge can only exist in the current hyper k-truss. However, removing e_i also invalidates all hyper-triangles involving e_i, so the supports of all affected hyperedges are updated accordingly (Lines 14–15). This process is repeated iteratively until no hyperedges remain in G (Lines 17–19).

Although Algorithm 1 performs hyper-truss decomposition on hypergraphs, it still suffers from notable computational inefficiencies. In the hyper-triangle counting phase, it exhaustively enumerates all neighbor pairs of each hyperedge, resulting in a large number of redundant checks, as most pairs only form open hyper-triangles that do not contribute to support values. Moreover, during the peeling phase, when a hyperedge e_i is removed, the algorithm must traverse all hyper-triangles involving e_i to update supports. This process repeatedly filters out open hyper-triangles and may update the support of the same hyperedge multiple times, further increasing the computational overhead.

Algorithm 1: HTD-BS

```
1  Input: Hypergraph G(V, E)
2  Output: The hyper-truss number for e ∈ E, i.e.φ(e)
3  ∀e ∈ E initialize N_e ← ∅, sup(e) ← 0, k ←2, φ ← ∅
4  for each hyperedge e_i ∈ E do
5      for each vertex v ∈ e_i do
6          for each hyperedge e_j ∈ E_v do
7              N_{e_i} ← N_{e_i} ∪ e_j
8  for each hyperedge e_i ∈ E do
9      for each hyperedge e_j ∈ N_{e_i} and i < j do
10         for each hyperedge e_k ∈ N_{e_j} and j < k do
11             if e_i ∩ e_k ≠ ∅ then
12                 sup(e_i)+ = 1, sup(e_j)+ = 1, sup(e_k)+ = 1
13 while ∃ hyperedge e_i such that sup(e_i) ≤ (k − 2) do
14     for each hyperedge e_j ∈ N_{e_i} do
15         sup(e_j)− = |N_{e_i} ∩ N_{e_j}|
16     φ(e_i) ← k, remove e_i from G
17 if not all hyperedges in G are removed then
18     k ← k + 1
19     goto Line 13
20 return {φ(e)|e ∈ E};
```

3 THE HYPER-TRUSS DECOMPOSITION ALGORITHM

To address the limitations of existing algorithms, we propose a vertex-aware framework that leverages vertex-level connectivity among hyperedges to accelerate the hyper-truss decomposition process. The framework consists of two phases: a hyper-triangle counting phase and a peeling phase. In the hyper-triangle counting phase, we employ a vertex-oriented counting strategy that organizes the search around shared vertices among hyperedges, eliminating redundant traversal of open hyper-triangles. In the peeling phase, we design an optimized peeling strategy with vertex-based state propagation, which propagates the removal information of each hyperedge through its incident vertices and allows the support values of affected hyperedges to be directly updated in specific cases without explicitly enumerating hyper-triangles.

3.1 Pre-Processing

Before counting hyper-triangles, we need to identify all hyperwedges $\wedge_{ij}$ and store the vertices contained in Ω_{ij}, Ω^{+}_{ij}, and Ω^{-}_{ij}. However, since each hyperwedge may contain a variable number of vertices, storing these subsets incurs substantial memory overhead. To address this issue, we introduce a bitmap-based encoding scheme that uses bit positions to indicate vertex membership. We first sort the vertices within each hyperedge in ascending order.

Then, for each hyperwedge $\wedge_{ij}$ formed by hyperedges e_i and e_j (with $i < j$), the hyperedge with the smaller ID is designated as the reference hyperedge. Based on the vertex order in e_i, we construct bitmaps B_{ij} and B^+_{ij} to indicate vertex membership for the subsets Ω_{ij} and Ω^+_{ij}. In B_{ij}, the n^{th} bit is set to 1 if the n^{th} vertex of e_i also appears in e_j, and 0 otherwise. In B^+_{ij}, the n^{th} bit is set to 1 if the corresponding vertex appears only in e_i. Likewise, we construct a bitmap B^-_{ij} by taking e_j as the reference hyperedge to represent Ω^-_{ij}. Note that we use $B_{ij}(n)$, $B^+_{ij}(n)$, and $B^-_{ij}(n)$ to denote the n^{th} bit of the corresponding bitmap later in this paper. For example, consider two hyperedges $e_1 = \{1, 2, 3, 4, 7, 8\}$ and $e_2 = \{2, 5, 6, 8, 9, 10\}$, which share the common vertices $\{2, 8\}$. Based on their positions in e_1, these shared vertices are encoded as the bitmap *010001*. Likewise, the exclusive vertices of e_1 and e_2 are represented by the bitmaps *101110* and *011011*, respectively.

Algorithm 2: Pre-Processing

```
1  Input: Hypergraph G(V, E)
2  Output: The hyperwedge list ∧ and corresponding bitmaps
3  ∀e ∈ E sort vertices of e in ascending order
4  ∀v ∈ V Initialize a list E_v and sort the hyperedges of E_v in
     ascending order
5  ∧ ← ∅, α ← max_{e∈E} |e|
6  for each hyperedge e_i ∈ E by an ascending order do
7      for each n^th vertex v ∈ e_i do
8          for each hyperedge e_j ∈ E_v do
9              if i < j then
10                 if ∧_ij ∉ ∧ then
11                     ∧ ← ∧ ∪ ∧_ij
12                     B_ij ← 0, B^+_ij ← 2^α − 1, B^−_ij ← 2^α − 1
13                 B_ij(n) ←set the n^th bit as 1
14                 B^+_ij(n) ←set the n^th bit as 0
15             else if i > j then
16                 B^−_ji(n) ←set the n^th bit as 0
17 return ∧, B, B^+, B^−
```

Algorithm 2 details the pre-processing step. For each hyperedge, we first sort its vertices in ascending order (Line 3). Then, for each vertex v, we construct a list E_v to store all hyperedges that include v, sorted in ascending order (Line 4). We initialize an empty list $\wedge$ to store the hyperwedges and let α denote the maximum hyperedge capacity (Line 5). Next, we iterate through each hyperedge $e_i \in E$ in ascending order (Line 6). For every vertex v at the n^{th} position in e_i, we traverse its associated hyperedge list E_v (Lines 7–8). For each $e_j \in E_v$, when $i < j$, if the hyperwedge $\wedge_{ij}$ has not been found, we insert it into the list $\wedge$ (Line 11) and initialize three bitmaps: B_{ij} is initialized to 0, while B^+_{ij} and B^-_{ij} are initialized to $2^\alpha - 1$ (Line 12). Since the vertex v appears in both e_i and e_j, the n^{th} bit in B_{ij} is set to 1. Meanwhile, the n^{th} bit in B^+_{ij} is set to 0, reflecting that v no longer belongs exclusively to e_i (Lines 13-14). Similarly, when $i > j$, the n^{th} bit in B^-_{ji} is set to 0 (Line 16).

3.2 Hyper-Triangle Counting Phase

The support value of each hyperedge is determined by the number of hyper-triangles it participates in. Due to the structural complexity of hyper-triangles, existing techniques [25, 34, 52] incur redundant traversal of open hyper-triangles during the search process. To address this issue, we propose tailored counting algorithms for dense and sparse hyper-triangles, leveraging vertices from different regions to guide enumeration and ensure that only valid hyperedge triples are generated.

The algorithm for dense hyper-triangles. Dense hyper-triangles consist of three hyperedges that have overlapping vertices. This property enables the expansion of hyperwedges through these shared vertices to form dense hyper-triangles while avoiding the enumeration of open hyper-triangles. To achieve this, given a hypergraph $G = (V, E)$, we start from each hyperwedge $\wedge_{ij}$ and traverse its shared vertices v (i.e., $v \in \Omega_{ij}$) to search for additional hyperedges e_k such that $k > j$ and $v \in e_k$. Since each such hyperedge shares at least one of the common vertices with both e_i and e_j, these hyperedges necessarily form a dense hyper-triangle.

Algorithm 3: Count-Dense

```
1  Input: (1) Hypergraph G(V, E)
2         (2) ∧, B, B^+ B^− ←Pre-Processing(G)
3  Output: The dense hyper-triangle count of each e ∈ E
4  T_d ←Initialize a map to keep the counts of dense hyper-triangles
5  for each hyperwedge ∧_ij ∈ ∧ by an ascending order do
6      for each n^th vertex v ∈ e_i with B_ij(n) = 1 do
7          for each hyperedge e_k ∈ E_v do
8              if k > j and e_k has not been visited then
9                  T_d[e_i]+ = 1, T_d[e_j]+ = 1, T_d[e_k]+ = 1
10                 mark e_k as visited
11 return T_d
```

Based on the above strategy, we present Algorithm 3. After the *Pre-Processing*, we obtain the list of all hyperwedges along with their associated bitmaps. We then initialize a map $\mathcal{T}_d$ to record the number of dense hyper-triangles for each hyperedge (Line 4). The algorithm iterates through each hyperwedge $\wedge_{ij}$ in ascending order. We iterate through each vertex v in e_i and use the bitmap B_{ij} to determine whether v is also contained in e_j (Lines 6). If so, for each shared vertex v, we examine its incident hyperedge list E_v to search for a third hyperedge e_k such that $k > j$ and e_k has not been visited in the current iteration (Lines 7-8). If such a hyperedge is found, e_i, e_j, and e_k form a dense hyper-triangle. Finally, we increment the hyper-triangle count for each involved hyperedge and mark e_k as visited (Lines 9-10).

The algorithm for sparse hyper-triangles. Unlike dense hyper-triangles, which can be efficiently enumerated via vertices shared by all three hyperedges, sparse hyper-triangles require a different strategy since vertex overlaps exist only between pairs of hyperedges. Although Qin et al. [34] accelerate sparse hyper-triangle enumeration using a prefix forest, their method still incurs redundant traversal of open hyper-triangles. To address this issue, we identify a structural property of sparse hyper-triangles that enables the exclusion of open hyper-triangles during enumeration.

Proposition 1. Given a sparse hyper-triangle formed by hyperedges e_i, e_j, and e_k, for any two hyperwedges within the hyper-triangle (e.g., $\wedge_{ij}$ and $\wedge_{ik}$), it holds that $\Omega^-_{ij} \cap \Omega^-_{ik} \neq \emptyset$.

Proof. Since e_i, e_j, and e_k form a sparse hyper-triangle, we have $e_i \cap e_j \cap e_k = \emptyset$. Moreover, by definition of a hyper-triangle, $e_j \cap e_k \neq$

Algorithm 4: Count-Sparse

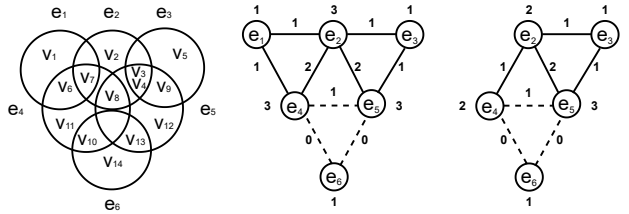
```
1  Input: (1) Hypergraph G(V, E)
2         (2) ∧, B, B+ B− ←Pre-Processing(G)
3  Output: The sparse hyper-triangle count of each e ∈ E
4  T_s ←Initialize a map to keep the counts of sparse hyper-triangles
5  ∀v ∈ V Initialize a map τ_v to store the hyperwedges
6  for each hyperwedge ∧_ij ∈ ∧ do
7      for each n^th vertex v ∈ e_j with B^-_ij(n) = 1 do
8          τ_v ← τ_v ∪ ∧_ij
9  for each hyperwedge ∧_ij ∈ ∧ by an ascending order do
10     for each n^th vertex v ∈ e_j with B^-_ij(n) = 1 do
11         for each hyperwedge ∧_ik ∈ τ_v do
12             if O(∧_ik) > O(∧_ij) and ∧_ik has not been visited then
13                 if B_ij&B_ik = 0 then
14                     T_s[e_i]+ = 1, T_s[e_j]+ = 1, T_s[e_k]+ = 1
15                     mark ∧_ik as visited
16 return T_s
```

$\emptyset$. Therefore, there exists a vertex in $(e_j \cap e_k) \setminus e_i$, which implies $\Omega^-_{ij} \cap \Omega^-_{ik} = (e_j \cap e_k) \setminus e_i \neq \emptyset$. Therefore, Proposition 1 holds.

Based on Proposition 1, we can leverage the vertices v in Ω^-_{ij} to combine each hyperwedge $\wedge_{ij}$ with another hyperwedge $\wedge_{ik}$ such that $v \in \Omega^-_{ik}$. This ensures that each enumerated pair of hyperwedges corresponds to a valid hyper-triangle, while open hyper-triangles are naturally excluded. However, for dense hyper-triangles $T(\{e_i, e_j, e_k\})$, except for patterns 2 and 9, the remaining patterns also contain at least one pair of hyperwedges (e.g., $\wedge_{ij}$ and $\wedge_{ik}$) satisfying $\Omega^-_{ij} \cap \Omega^-_{ik} \neq \emptyset$. Consequently, the above search procedure may also enumerate these dense hyper-triangles, resulting in additional computational overhead. This observation leads to two issues. The first is that it remains unclear whether performing extra traversals over dense hyper-triangles to avoid enumerating open ones can enhance the overall performance of the search process. The second is that, since both sparse and dense hyper-triangles are generated during enumeration, it is non-trivial to efficiently determine the type of each identified hyper-triangle.

For the first issue, we observe that dense hyper-triangles are significantly less frequent than open hyper-triangles in real-world datasets. As indicated in [25, 52], this is because dense hyper-triangles require all three hyperedges to share common vertices, resulting in a more compact structure that is inherently harder to form. Consequently, the number of open hyper-triangles can be up to 100× greater than that of dense ones in practice. This implies that the computational overhead of traversing dense hyper-triangles is negligible compared with that of traversing open hyper-triangles, and thus the proposed enumeration strategy remains efficient.

To address the second issue, the bitmap-based intersection test provides a mechanism for reasoning about vertex-sharing relations among multiple hyperedges. Specifically, we determine the type of each identified hyper-triangle $T(\{e_i, e_j, e_k\})$ by performing a bitwise operation between B_{ij} and B_{ik}. The result of $B_{ij}\&B_{ik}$ equals zero if and only if no vertex is shared among all three hyperedges, indicating a sparse hyper-triangle; otherwise, the hyper-triangle is of the dense type. Since the length of each bitmap is typically small



(a) A hypergraph G (b) The projected graph (c) After removing e_1

Figure 4: An example of the hyper-truss peeling process

in real-world hypergraphs, this bitwise operation can be performed in constant time, enabling efficient hyper-triangle classification.

Based on the above optimization, we propose Algorithm 4. After the *Pre-Processing*, the algorithm initializes a map $\mathcal{T}_s$ to record the number of sparse hyper-triangles associated with each hyperedge (Line 4). For each vertex v, a map τ_v is also initialized to store the corresponding hyperwedges (Line 5). Next, the algorithm iterates over all hyperwedges $\wedge_{ij}$. For each n^{th} vertex $v \in e_j$, we use the bitmap B^-_{ij} to determine whether $v \in \Omega^-_{ij}$. If so, the hyperwedge $\wedge_{ij}$ is inserted into τ_v (Lines 6–8). Then, all hyperwedges are revisited in ascending order (Line 9). For every vertex $v \in \Omega^-_{ij}$, the algorithm searches for another hyperwedge $\wedge_{ik}$ with a larger order that has not yet been visited in the current iteration (Lines 10–12). For each candidate hyper-triangle formed by $\wedge_{ij}$ and $\wedge_{ik}$, the algorithm determines its type by computing $B_{ij}\&B_{ik}$ (Line 13). If the result equals zero, the hyper-triangle is classified as a sparse hyper-triangle; in this case, we increment the hyper-triangle count for each involved hyperedge and mark $\wedge_{ik}$ as visited (Lines 14–15).

3.3 Peeling Phase

In the peeling phase, existing algorithms employ a simple peeling strategy that traverses all neighboring hyperedge pairs of each removed hyperedge to update the support values of the affected hyperedges. However, this approach not only incurs additional time in filtering open hyper-triangles but also causes redundant support updates for hyperedges that co-occur with removed ones in multiple hyper-triangles. To overcome these limitations, we classify the affected hyperedges into two groups: those forming dense hyper-triangles and those forming sparse hyper-triangles with the removed hyperedge. Based on this classification, we design tailored optimization strategies to efficiently update the support values.

Optimization for hyperedges in dense hyper-triangles. The affected hyperedges are limited to the neighbors of the removed hyperedge. Based on this observation, we record the number of dense hyper-triangles involving each hyperwedge $\wedge_{ij}$ before the peeling phase. After that, when a hyperedge e_i is removed during peeling, the support value of each neighboring hyperedge e_j can be directly updated using this pre-recorded count. For example, Figure 4(b) illustrates the projected graph derived from the hypergraph in Figure 4(a), showing both the support values of hyperedges and the dense hyper-triangle counts associated with each hyperwedge. During the peeling process, when e_1 is removed, the supports of e_2 and e_4 can be updated directly by decreasing the counts of dense hyper-triangles corresponding to $\wedge_{12}$ and $\wedge_{14}$, without enumerating them. However, as hyperedges are progressively removed, these counts must also be dynamically adjusted to reflect the remaining

dense hyper-triangles. As shown in Figure 4(c), after removing e_1, $\wedge_{24}$ is involved in only one dense hyper-triangle.

Lemma 3.1. *During the peeling process, the number of dense hyper-triangles involving a hyperwedge $\wedge_{ij}$ can be computed as: $\mathcal{T}_d[\wedge_{ij}] - \sum_{v\in(e_i\cap e_j)} |\overline{E}_v| + \sum_{e_k\in(\overline{N}_{e_i}\cap\overline{N}_{e_j})}(|e_i\cap e_j\cap e_k|-1)$, where $\overline{E}_v$ and $\overline{N}_{e_i}$ denote the sets of removed hyperedges e_k such that $v\in e_k$ and $e_i\cap e_k\neq\emptyset$, respectively.*

Proof. Initially, the hyperwedge $\wedge_{ij}$ is involved in $\mathcal{T}_d[\wedge_{ij}]$ dense hyper-triangles. For each vertex $v\in e_i\cap e_j$, every removed hyperedge e_k with $v\in e_k$ forms a dense hyper-triangle with $\wedge_{ij}$, contributing at most $\sum_{v\in(e_i\cap e_j)}|\overline{E}_v|$ removed dense hyper-triangles. However, for any removed dense hyper-triangle $T(\{e_i,e_j,e_k\})$ with $\varphi(T)>1$ (i.e., $\varphi(T)=|e_i\cap e_j\cap e_k|$), the three hyperedges share multiple common vertices, causing each such hyper-triangle to be counted redundantly $(\varphi(T)-1)$ times. After correcting this overcounting, the number of dense hyper-triangles removed for $\wedge_{ij}$ is: $\sum_{v\in(e_i\cap e_j)}|\overline{E}_v| - \sum_{e_k\in(\overline{N}_{e_i}\cap\overline{N}_{e_j})}(|e_i\cap e_j\cap e_k|-1)$. Therefore, subtracting this value from $\mathcal{T}_d[\wedge_{ij}]$ completes the proof.

Based on Lemma 3.1, when a hyperedge e_i is removed during the peeling process, we traverse its neighboring hyperedges e_j. If the hyperwedge $\wedge_{ij}$ is currently involved in dense hyper-triangles T with $\varphi(T)>1$, we adopt the same procedure as in *Count-Dense* to enumerate all dense hyper-triangles containing $\wedge_{ij}$ and update the support value of e_j accordingly. During this process, for each identified dense hyper-triangle $T(\{e_i,e_j,e_k\})$, we maintain a counter $\mathcal{R}[\wedge_{jk}]$ for the hyperwedge $\wedge_{jk}$ and increase it by $\varphi(T)-1$, which is later used to correct potential overcounting. If $\wedge_{ij}$ is not involved in any dense hyper-triangles T with $\varphi(T)>1$, the support value of e_j can be directly decreased by $\mathcal{T}_d[\wedge_{ij}] - \sum_{v\in(e_i\cap e_j)}|\overline{E}_v| + \mathcal{R}[\wedge_{ij}]$ without traversing any dense hyper-triangles.

Optimization for hyperedges in sparse hyper-triangles Since the three hyperedges of a sparse hyper-triangle do not share any common vertices, the removal information of hyperedges cannot be propagated through shared vertices. Consequently, it is impractical to dynamically update the counts of sparse hyper-triangles for each hyperwedge. However, we observe that the number of sparse hyper-triangles in most datasets is relatively small. Therefore, for each removed hyperedge e_i, we directly identify the sparse hyper-triangles involving e_i using *Count-Sparse* to update the support values. In addition, when processing the affected hyperedges that form dense hyper-triangles with e_i, we mark these dense hyper-triangles in advance to avoid revisiting them during the search for sparse hyper-triangles.

By combining the above optimizations with the *Pre-Processing* and hyper-triangle counting steps, we obtain the advanced hyper-truss decomposition algorithm, as presented in Algorithm 5. We first apply *Count-Dense* and *Count-Sparse* to compute the number of hyper-triangles containing each hyperedge, which yields the initial support value of every hyperedge (Lines 4-5). Then, for each vertex v, we initialize a list $\overline{E}_v$ to store the removed hyperedges that contain v. In addition, for each hyperwedge $\wedge_{ij}$, we initialize a counter $\mathcal{R}[\wedge_{ij}]$, which is used to correct overcounting, as discussed earlier (Line 6). Next, for each k starting from 2, we iteratively remove any hyperedge e_i whose support value satisfies $sup(e_i)\le(k-2)$. For

Algorithm 5: HTD-ADV+

```
1  Input: Hypergraph G(V, E)
2  Output: The hyper-truss number for e ∈ E, i.e.φ(e)
3  ∧ ←Pre-Processing(G), k ←2, φ ← ∅
4  T_d ←Count-Dense(G), T_s ←Count-Sparse(G)
5  ∀e ∈ E initialize sup(e) ← T_d[e] + T_s[e]
6  ∀v ∈ V initialize Ē_v ← ∅, ∀∧_ij ∈ ∧ initialize R[∧_ij] ← 0
7  while ∃ hyperedge e_i such that sup(e_i) ≤ (k − 2) do
8      for each hyperedge e_j ∈ N_{e_i} do
9          if ∄T({e_i, e_j, e_k}) s.t. φ(T) > 1 then
10             if i < j then
11                 sup(e_j)− = T_d[∧_ij] − Σ_{v∈(e_i∩e_j)} |Ē_v| + R[∧_ij]
12             else
13                 sup(e_j)− = T_d[∧_ji] − Σ_{v∈(e_i∩e_j)} |Ē_v| + R[∧_ji]
14         else
15             for each dense hyper-triangle T({e_i, e_j, e_k}) do
16                 if j < k then
17                     mark ∧_jk as visited
18                     R[∧_jk]+ = φ(T) − 1
19                 sup(e_j)− = 1
       // skip the hyperwedge ∧_jk that has been visited
20     for each sparse hyper-triangle T({e_i, e_j, e_k}) do
21         sup(e_j)− = 1, sup(e_k)− = 1
22     for each vertex v ∈ e_i do
23         Ē_v ← Ē_v ∪ e_i
24     φ(e_i) ← k, remove e_i from G
25 if not all hyperedges in G are removed then
26     k ← k + 1
27     goto Line 7
28 return {φ(e)|e ∈ E};
```

each neighboring hyperedge e_j, if e_i and e_j are only involved in hyper-triangles T with $\varphi(T)\le 1$, the support value of e_j can be updated directly without enumerating the corresponding hyper-triangles (Lines 9-13). Otherwise, we apply the same procedure as in *Count-Dense* to identify all dense hyper-triangles containing both e_i and e_j and update the support value of e_j accordingly (Lines 14-19). For each identified dense hyper-triangle $T(\{e_i,e_j,e_k\})$, we mark the hyperwedge $\wedge_{jk}$ as visited and increase $\mathcal{R}[\wedge_{jk}]$ by $\varphi(T)-1$ to correct potential overcounting. Afterwards, we employ the procedure in *Count-Sparse* to identify all sparse hyper-triangles involving e_i and update the support values of the affected hyperedges (Lines 20–21). Note that during this process, the algorithm skips any visited hyperwedge $\wedge_{jk}$, since it corresponds to a dense hyper-triangle with e_i and has already been processed. After processing all neighboring hyperedges of e_i, we iterate over each vertex $v\in e_i$ and insert e_i into $\overline{E}_v$ to indicate the removal of e_i (Lines 22–23). Finally, we assign the hyper-truss number k to e_i and remove e_i from the hypergraph. This process is repeated until no hyperedges remain in the hypergraph.

Example 2. *For the hypergraph G shown in Figure 4(a), Figure 4(b) illustrates the initial support values of all hyperedges and the number of dense hyper-triangles $\mathcal{T}_d[\wedge_{ij}]$ associated with each hyperwedge $\wedge_{ij}$. During the peeling phase, we first remove e_1. Since e_1 is contained only in the hyper-triangle $T(\{e_1,e_2,e_4\})$ where $\varphi(T)=1$, the support values of e_2 and e_4 can be directly decreased by one. For each vertex*

$v \in e_1$, *hyperedge* e_1 *is then inserted into* $\overline{E}_v$ *to record its removal. Next, when* e_3 *is removed, since* $\varphi(T(\{e_2, e_3, e_5\})) = 2$*, we need to traverse all dense hyper-triangles containing* e_3 *to update the support values of* e_2 *and* e_5*. During this process, we set* $\mathcal{R}[\wedge_{25}] = 1$*, and insert* e_3 *into* $\overline{E}_v$ *for each* $v \in e_3$*. Then, when* e_2 *is removed, it is contained only in the hyper-triangle* $T(\{e_2, e_4, e_5\})$ *with* $\varphi(T) = 1$*. Accordingly, we directly decrease the support value of* e_4 *by* $\mathcal{T}_d[\wedge_{24}] - |\overline{E}_{v_7}| = 1$*, and that of* e_5 *by* $\mathcal{T}_d[\wedge_{25}] - |\overline{E}_{v_3}| - |\overline{E}_{v_4}| + \mathcal{R}[\wedge_{25}] = 1$*. Finally, when* e_6 *is removed, since it is not contained in any dense hyper-triangles, we directly traverse the sparse hyper-triangle* $T(\{e_4, e_5, e_6\})$ *to update the support values of* e_4 *and* e_5*.*

4 COMPLEXITY ANALYSIS AND DISCUSSIONS

In this section, provides a comprehensive analysis of the proposed framework, including its theoretical time and space complexity, parallelization potential, and generality.

4.1 Complexity Analysis

Table 2 presents the time and space complexities of three hyper-truss decomposition algorithms: *HTRUSS*, *HTD-BS*, and *HTD-ADV+*. *HTRUSS* is an existing algorithm that combines the sparse hyper-triangle counting algorithm *HTC-PF+* with a simple peeling strategy [34]. *HTD-BS* serves as a baseline method that integrates an enumeration-based counting approach in [25, 52] with the same peeling strategy, while *HTD-ADV+* is the advanced algorithm proposed in this paper. In Table 2, C and $\mathcal{D}$ denote the average capacity and average degree of hyperedges, respectively, and x denotes the number of trees in the prefix forest.

In the preprocessing stage, *HTRUSS* only traverses each hyperedge to construct the prefix forest, resulting in a time complexity of $O(|E|C)$. By contrast, *HTD-BS* and *HTD-ADV+* traverse the vertices within each hyperedge to enumerate all hyperwedges and identify their common vertices, which leads to a time complexity of $O(|E|\mathcal{D}C)$. Therefore, as the number of hyperedges or the hyperedge capacity increases, the preprocessing time of all three algorithms grows accordingly. However, this impact is more pronounced for *HTD-BS* and *HTD-ADV+* in dense hypergraphs with large hyperedge degrees.

During the counting phase, *HTD-BS* examines every pair of neighboring hyperedges for each hyperedge, resulting in a time complexity of $O(|E|\mathcal{D}^2)$. For *HTRUSS*, sparse hyper-triangles are identified through a prefix forest. In the worst case, where the three participating hyperedges belong to different prefix trees, the algorithm still needs to scan the vertices of all three hyperedges, leading to a time complexity of $O((C|E|)^3/(|V| \cdot x))$. For *HTD-ADV+*, during the counting phase, the algorithm only traverses hyper-triangles for each hyperedge. Although it may revisit some dense hyper-triangles, it avoids enumerating open hyper-triangles. Consequently, the time complexity of this phase is bounded by $O(\sum_{e_i \in E}(\mathcal{T}_s[e_i] + \mathcal{T}_d[e_i]))$. From the above complexity analysis, we observe that the running time of *HTRUSS* is primarily influenced by hyperedge capacity, whereas the performance of *HTD-BS* is dominated by hyperedge degree. Moreover, both methods incur substantial overhead due to redundant traversal of open hyper-triangles. In contrast, the running time of *HTD-ADV+* depends on the number of hyper-triangles. Since its additional cost arises only from revisiting dense hyper-triangles, which are fewer than open hyper-triangles in practice, *HTD-ADV+* remains more efficient in the counting phase, with the advantage being more pronounced on sparse hypergraphs, where the total number of hyper-triangles is smaller.

In the peeling phase, both *HTD-BS* and *HTRUSS* update support values by iterating over all neighboring hyperedges of each hyperedge e_i, which results in a time complexity of $O(|E|\mathcal{D}^2)$. For *HTD-ADV+*, in the worst case where no hyper-triangle T with $\varphi(T) = 1$ exists, the algorithm still needs to traverse all hyper-triangles. In this scenario, the time complexity of the peeling phase is bounded by that of the counting phase. Nevertheless, even in this worst-case scenario, *HTD-ADV+* remains more efficient than *HTD-BS* and *HTRUSS*.

For the space complexity, *HTD-BS* only needs to store each hyperedge and its neighboring hyperedges, resulting in a space complexity of $O(|E|C + |E|\mathcal{D})$. In contrast, both *HTRUSS* and *HTD-ADV+* additionally store the common vertices associated with each hyperwedge, leading to a space complexity of $O(|E|\mathcal{D}C)$. Note that the space usage is dominated by the preprocessing stage; therefore, the hyper-triangle counting algorithm used in each decomposition algorithm incurs the same asymptotic space complexity.

4.2 Parallelization

Our hyper-truss decomposition algorithm can be parallelized to better handle large-scale hypergraphs. By using multiple threads to concurrently process different hyperwedges (Line 5 of Algorithm 3 and Line 9 of Algorithm 4), we can compute the support values in parallel, thereby accelerating the hyper-triangle counting phase. Since this phase dominates the overall decomposition cost in our framework, parallelizing the counting procedure can accelerate the entire decomposition.

4.3 Generalization of Optimizations

The proposed bitmap-based encoding can be extended to algorithms that require computing about vertex overlaps among hyperedges. By encoding vertex membership using bit positions, the bitmap representation provides a compact way to store overlap information between hyperedges. For example, when identifying higher-order motifs such as hyper-cliques, the same bitmap-based representation can be reused to efficiently verify overlap conditions and classify motif patterns. In addition, the optimization strategy adopted in the peeling phase of our framework can also be applied to higher-order motif–based decomposition frameworks.

Moreover, the proposed framework can potentially be implemented on top of existing hypergraph processing systems, such as MESH [14] and HyperX [16], through lightweight algorithmic customization. In detail, the framework relies on common primitives such as hyperedge-level iteration and per-hyperedge support maintenance, while requiring only additional bitmap-based operations to efficiently capture vertex-sharing relations among hyperedges. By using standard data structures, the proposed approach can be incorporated into current hypergraph processing systems without modifying their underlying execution models.

Algorithms	*HTRUSS [34]*	*HTD-BS*	*HTD-ADV+*
Preprocessing (T_1)	$O(\|E\|C)$	$O(\|E\|\mathcal{D}C)$	$O(\|E\|\mathcal{D}C)$
Counting dense hyper-triangle (T_2)	N/A	$O(\|E\|\mathcal{D}^2)$	$O\left(\sum_{e_i \in E} \mathcal{T}_s[e_i]\right)$
Counting sparse hyper-triangle (T_3)	$O\left((C\|E\|)^3 / (\|V\| \cdot x)\right)$		$O\left(\sum_{e_i \in E} (\mathcal{T}_s[e_i] + \mathcal{T}_d[e_i])\right)$
Peeling (T_4)	$O(\|E\|\mathcal{D}^2)$	$O(\|E\|\mathcal{D}^2)$	$O\left(\sum_{e_i \in E} (\mathcal{T}_s[e_i] + \mathcal{T}_d[e_i])\right)$
Space complexity	$O(\|E\|\mathcal{D}C)$	$O(\|E\|C + \|E\|\mathcal{D})$	$O(\|E\|\mathcal{D}C)$

* C and $\mathcal{D}$ are the average capacity and degree number of hyperedges. x is the number of trees in the prefix forest.

Table 2: Time and space complexity

Dataset	|V|	|E|	$\|e\|_{max}$	$\|e\|_{avg}$	$\|\wedge\|$
HPS	327	7,818	5	2.3	593K
CMH	1,014,734	895,439	25	1.5	1.7M
CPS	242	12,704	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	1,512	18	3.0	87.8M
CD	1,924,991	2,466,792	25	2.9	125M
TAU	3,029	147,222	5	3.4	564M
ThM	176,445	595,749	21	2.4	647M
TMS	1,629	170,476	5	3.5	913M

Table 3: Summary of datasets

5 EXPERIMENTS

In this section, we report the evaluation of the proposed algorithms on 11 real-world datasets.

5.1 Experimental Settings

In the experiments, we evaluate three hyper-truss decomposition algorithms: *HTD-BS*, *HTD-ADV*, and *HTD-ADV+*. Among them, *HTD-BS* serves as the baseline, combining the enumeration-based approach in [25] with the simple peeling procedure in [34]. *HTD-ADV* is an optimized variant that incorporates the proposed preprocessing and hyper-triangle counting optimizations. *HTD-ADV+* is the most advanced variant, integrating all optimizations introduced in this paper, including both counting and peeling enhancements.

To evaluate the effectiveness of the optimization strategies proposed for hyper-triangle counting, we compare four algorithms. These include *MoCHy-E* [25], an existing hyper-triangle counting algorithm that explicitly identifies and categorizes all hyper-triangles. *Count-adv*, our advanced counting algorithm that integrates *Count-Dense* and *Count-Sparse*; *HTC-PF+* [34], which employs prefix forests to detect sparse hyper-triangles; and *Count-Sparse*, the sparse hyper-triangle counting algorithm proposed in this paper.

In our experiments, the reported runtime of all hyper-truss decomposition algorithms and hyper-triangle counting algorithms includes the preprocessing time. For all experiments, we set a maximum runtime limit of 24 hours. All algorithms are implemented in C++, and the experiments are conducted on a Linux server equipped with an Intel(R) Xeon(R) Platinum 8260L CPU (2.30 GHz) and 256 GB of main memory.

Datasets. In our experiments, we evaluate performance using 11 real-world datasets from four distinct domains. The co-authorship domain includes four datasets: coauthor-DBLP (CD), coauthor-geology (CMG), and coauthor-history (CMH), derived from Microsoft Academic Graph [38]. In these datasets, nodes correspond to authors, and each hyperedge denotes a publication involving multiple co-authors. The contact domain consists of two datasets: contact-primary-school (CPS) [39] and contact-high-school (HPS) [31]. Here, vertices represent individuals, while hyperedges capture group interactions or contacts among them. The email domain includes Email-EU (EEU) [27, 50] and Email-Enron (EEN) [23], where vertices are email accounts and hyperedges represent communication events, such as messages sent or received among these entities. We also consider datasets from the tags and threads domains. The tags domain includes tags-ubuntu (TAU) and tags-math (TMS), where vertices represent tags and hyperedges correspond to forum posts containing them. The threads domain comprises threads-ubuntu (ThAU) and threads-math (ThM), in which nodes represent users and hyperedges model discussions involving these users. All datasets in these two domains are obtained from [5]. To ensure the hypergraph property, specifically that no two hyperedges contain the identical set of vertices, we apply de-duplication pre-processing to eliminate redundant hyperedges. Detailed statistics of all datasets are summarized in Table 3.

5.2 Evaluation of Hyper K-truss Models

Distribution of hyper-truss numbers. Figure 6 presents the distribution of hyper-truss numbers assigned to hyperedges under the existing hyper k-truss model [34], denoted as *STruss*, and our proposed hyper k-truss model, denoted as *HTruss*. As shown in the figure, across all four datasets, *HTruss* produces a substantially wider range of hyper-truss numbers. In contrast, the hyper-truss numbers produced by *STruss* are highly concentrated within a relatively narrow range. This difference arises from the fact that *HTruss* considers all types of hyper-triangles, whereas the existing model *STruss* is restricted to sparse hyper-triangles only. Since sparse hyper-triangles typically constitute only a small fraction of all hyper-triangle patterns, ignoring dense hyper-triangles further prevents *STruss* from capturing the distinct semantic and structural information encoded by different hyper-triangle patterns. As a result, *STruss* provides limited structural evidence for effectively differentiating hyperedges. By incorporating all hyper-triangle patterns, *HTruss* achieves a finer-grained and more expressive assignment of hyper-truss numbers, thereby enhancing its discriminative power and enabling more precise extraction of cohesive structures from hypergraphs.

Case study. In this part, we present a case study based on the DBLP dataset to illustrate how incorporating diverse hyper-triangle patterns enables our model to capture fine-grained collaboration structures. As shown in Figure 5, we select six papers together with their associated authors and report their corresponding hyper-truss numbers under two models, where *STruss* denotes the hyper-truss number obtained by the existing model and *HTruss* represents the hyper-truss number derived from our model. The table reveals a clear difference between the hyper-truss numbers produced by the two models. In particular, for papers e_1 and e_6, the hyper-truss number is zero under the existing model. In co-authorship networks, research papers are typically produced through collaborations among researchers with different levels of seniority. Senior

Author (ID)	
Guoliang Li (1)	Wenyan Li (2)
Yongkui Man (3)	H. V. Jagadish (4)
He Wen (5)	Dong Deng (6)
Jianhua Feng (7)	Xiaohang Zhang (8)
Cong Yu (9)	Jian Li (10)
Yu Jiang (11)	Li Qian (12)
Jinyang Gao (13)	Lawrence O'Gorman (14)

Paper	Title	Author ID	STruss	HTruss
e_1	Optimal parameter design of input...	1, 2, 3	0	3083
e_2	META: an efficient matching-based...	1, 4, 5, 6, 7	34	7262
e_3	Crowdsourced top-k algorithms...	1, 7, 8	10	3083
e_4	Scalable column concept determination...	1, 6, 9, 10, 11	30	62130
e_5	Learning user preferences by adaptive...	4, 12, 13	42	7262
e_6	An object model for image recognition	4, 14	0	7262

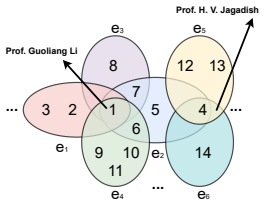
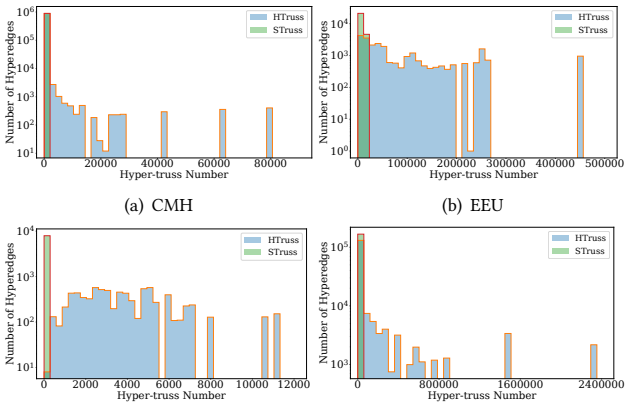


Figure 5: Case study of different hyper k-truss models



(a) CMH (b) EEU

(c) HPS (d) ThAU

Figure 6: Distribution of hyper-truss numbers.

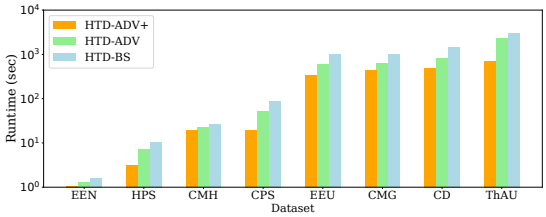
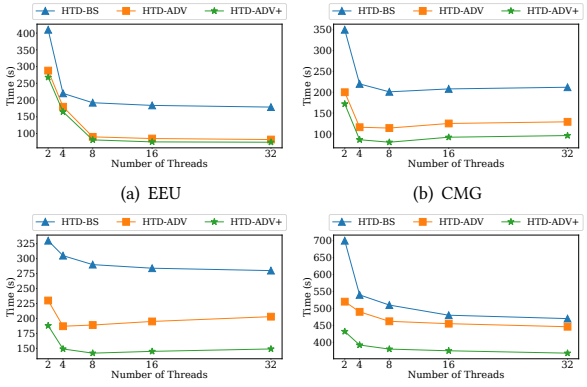


Figure 7: Runtime of decomposition algorithms.



(a) EEU (b) CMG

(c) CD (d) ThAU

Figure 8: Parallelizability of decomposition algorithms.

academics (e.g., Author 1 and Author 4) often initiate and supervise projects, while junior collaborators and student members contribute to specific parts of the study. This hierarchical collaboration pattern ensures that papers originating from the same research group are closely connected through the senior researcher. In our case study, papers e_1, e_2, e_3, and e_4 are centered around Author 1, forming a tightly connected collaboration group. The hyper-triangles formed in such scenarios mainly correspond to dense types, which capture strong intra-group collaborations. If only sparse hyper-triangles are considered, such intra-group structural information is largely overlooked. Under the existing model, which relies solely on sparse hyper-triangles, papers e_2 and e_4 are assigned very similar hyper-truss numbers. In contrast, by incorporating dense hyper-triangles that naturally arise from intra-group collaborations, our model assigns a higher hyper-truss number to e_4 than to e_2, reflecting stronger internal cohesion. This case study demonstrates that integrating all hyper-triangle patterns into the hyper k-truss model better captures diverse collaboration semantics and preserves richer higher-order structural information in real-world hypergraphs.

5.3 Evaluation of Decomposition Algorithms

Runtime for the decomposition algorithms. Figure 7 shows the runtime of three algorithms, *HTD-BS*, *HTD-ADV*, and *HTD-ADV+*, on eight hypergraphs. We can observe that *HTD-BS* requires the longest runtime, as it does not incorporate any optimization. In contrast, *HTD-ADV* achieves up to a 2× speedup over *HTD-BS* since the proposed optimization completely avoids traversing open hyper-triangles, thereby improving efficiency. As for *HTD-ADV+*, it further incorporates optimized peeling strategies on top of *HTD-ADV* and achieves up to a 4× speedup compared to *HTD-BS*. This is because, in most real-world hypergraphs, hyperedges participating in dense hyper-triangles typically share a single vertex. As a result, the proposed optimization effectively avoids revisiting a large number of hyper-triangles during peeling, thereby significantly accelerating the overall decomposition process.

Parallelizability for the decomposition algorithms. Figure 8 presents the scalability evaluation of the parallel versions of *HTD-BS*, *HTD-ADV*, and *HTD-ADV+* on four datasets. As the number of threads increases, the runtime of all three algorithms decreases substantially. In particular, *HTD-ADV* and *HTD-ADV+* achieve speedups of up to 6×, whereas *HTD-BS* stabilizes at around a 2× speedup. These results demonstrate both the effectiveness of the proposed parallel optimizations and the strong scalability of the proposed framework.

Redundant traversals of the decomposition algorithms. In Figure 9, we report the number of redundant traversals for *HTD-BS*, *HTD-ADV*, and *HTD-ADV+* when counting hyper-triangles. For *HTD-BS*, the value represents the number of open hyper-triangles traversed, whereas for *HTD-ADV* and *HTD-ADV+*, it corresponds to the number of dense hyper-triangles that are revisited. Since

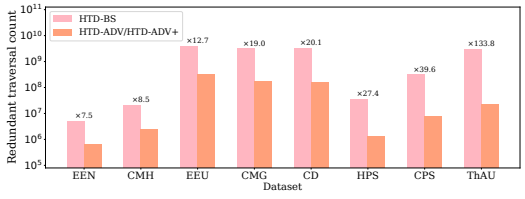


Figure 9: Redundant traversals of decomposition algorithms.

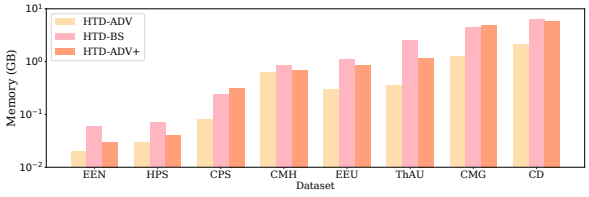


Figure 10: Space of decomposition algorithms.

Dataset	t_{pre}	t_{count}	t_{peel}	t_{total}	$\frac{t_{count}}{t_{total}}$
EEN	0.1	0.55	0.42	1.07	51.4%
HPS	0.63	1.57	1.05	3.25	48.31%
CMH	3	10	7	20	50%
CPS	3.3	11.7	5	20	58.5%
EEU	31.6	158.4	153	343	46.2%
CMG	108	192	150	450	42.67%
CD	130	255	160	545	46.79%
ThAU	110	568	32	710	80%

Table 4: Runtime (sec) of different phases in *HTD-ADV+*.

HTD-ADV and *HTD-ADV+* employ the same counting phase, they incur the same amount of redundant traversal. We also report the ratio of redundant traversals between *HTD-BS* and *HTD-ADV*/*HTD-ADV+*. As can be observed, the proposed methods incur fewer redundant traversals than the baseline across all datasets. In particular, email-domain datasets exhibit relatively smaller ratios, while threads-domain datasets demonstrate the most pronounced reduction in redundant traversal. This is because the distribution of hyper-triangle patterns varies across different application domains [25]. In email-domain datasets, patterns such as 5 and 8 occur more frequently, leading to a relatively larger number of dense hyper-triangles being revisited. In contrast, in threads-domain datasets, only pattern 12 appears with noticeable frequency, resulting in fewer dense hyper-triangles being revisited. Although the dataset type affects the optimization efficiency of *HTD-ADV* and *HTD-ADV+*, dense hyper-triangles remain inherently rarer than open hyper-triangles in real-world datasets. Consequently, the proposed algorithms consistently outperform *HTD-BS*.

Space of the decomposition algorithms. Figure 10 shows the space consumption of *HTD-BS*, *HTD-ADV*, and *HTD-ADV+* across eight datasets. We observe that on six of these datasets, *HTD-BS* consumes more memory than both *HTD-ADV* and *HTD-ADV+*. Although *HTD-BS* has a lower theoretical space complexity, it incurs higher memory usage in practice. This is because *HTD-BS* stores the neighbors of each hyperedge using sets to determine whether three hyperedges can form a hyper-triangle, which introduces significant space overhead. In contrast, *HTD-ADV* and *HTD-ADV+* avoid this overhead, since the hyperedge triples identified by *Count-Dense* and *Count-Sparse* are guaranteed to form valid hyper-triangles. As a result, neighboring hyperedges can be stored using lists instead of sets. Moreover, both methods further reduce space consumption by employing bitmaps to encode vertex positions within each hyperwedge. Note that *HTD-BS* could also store neighboring hyperedges using lists to reduce space usage; however, this would substantially increase the search time. Therefore, we do not adopt this approach.

Between the two optimized variants, *HTD-ADV+* requires additional space compared with *HTD-ADV*. This is because the optimized peeling strategy in *HTD-ADV+* needs to store extra information for each hyperwedge. Consequently, *HTD-ADV* exhibits the lowest space consumption among the three algorithms.

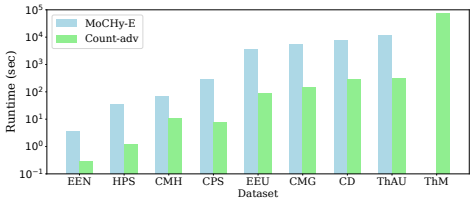
Runtime breakdown of decomposition algorithms. Table 4 presents the runtime breakdown of *HTD-ADV+* across eight datasets. Specifically, t_{pre} denotes the preprocessing time, t_{count} represents the runtime of the hyper-triangle counting phase, and t_{peel} corresponds to the peeling phase, with the total runtime denoted by t_{total}. The results show that the hyper-triangle counting phase is consistently the most time-consuming component. Except for the ThAU dataset, the time spent on hyper-triangle counting accounts for up to 58.5% of the total runtime, underscoring the importance of optimizing this phase to improve the overall efficiency of the algorithm. In contrast, for the ThAU dataset, hyper-triangle counting contributes 80% of the total runtime. This is because the hyperedges in ThAU are generally very small, causing most dense hyper-triangles T to have an overlap degree of one (i.e., $\varphi(T) = 1$). As a result, our optimization substantially accelerates the peeling phase on ThAU. This observation indicates that the proposed method is particularly effective on datasets with small hyperedge sizes.

5.4 Evaluation of Counting Algorithms

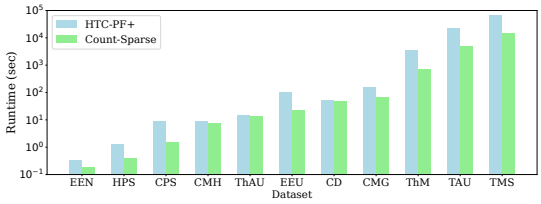
Runtime for hyper-triangle counting algorithms. To demonstrate the effectiveness of our hyper-triangle counting optimizations, we compare *Count-adv* with the existing algorithm *MoCHy-E* [25], both of which aim to identify all hyper-triangles. In addition, we compare *Count-Sparse* with *HTC-PF+* [34], which focus on counting only sparse hyper-triangles. The runtime results are shown in Figure 11(a) and Figure 11(b), respectively.

As shown in Figure 11(a), *Count-adv* achieves more than a 10× speedup over *MoCHy-E*. This is because *Count-adv* completely avoids traversing open hyper-triangles, thereby substantially accelerating the search process. In addition, bitwise operations are used to determine hyper-triangle types, further reducing the computational overhead. In contrast, *MoCHy-E* not only traverses open hyper-triangles, but also requires additional iteration over the vertices within each hyper-triangle to determine its type, which incurs extra computational cost and leads to lower efficiency. Notably, on the ThM dataset, *MoCHy-E* exceeds the time limit.

Similarly, Figure 11(b) shows that *Count-Sparse* achieves up to a 5× speedup over *HTC-PF+*. Although *HTC-PF+* employs prefix forests to reduce redundant vertex intersection checks among overlapping hyperedges, it still incurs substantial overhead in filtering open hyper-triangles. In contrast, *Count-Sparse* avoids enumerating open hyper-triangles by leveraging vertex-level connectivity among hyperedges, resulting in significantly improved runtime performance. Note that the TAU dataset contains more hyperwedges.



(a) Algorithms for counting all hyper-triangles



(b) Algorithms for counting sparse hyper-triangles

Figure 11: Runtime (s) of the counting algorithms.

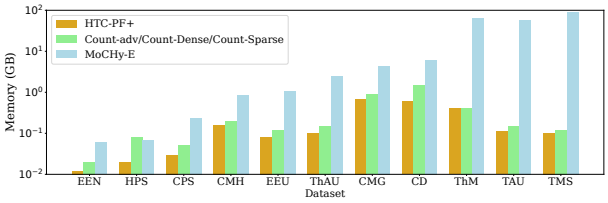


Figure 12: Space of counting algorithms.

As a result, although its size is smaller than that of CD and CMG, it requires a longer counting time.

Space of the counting algorithms. Figure 12 reports the space consumption of *HTC-PF+*, *Count-adv*, and *MoCHy-E*. Note that *Count-Sparse* and *Count-Dense* share the same preprocessing stage as *Count-adv* and therefore incur the same space overhead. As shown in the figure, *MoCHy-E* requires substantially more memory than the other two algorithms. This is because, to determine whether three hyperedges can form a hyper-triangle, *MoCHy-E* maintains hyperedge neighbors using set-based data structures, which significantly increases memory usage. In contrast, *HTC-PF+* organizes data using a prefix forest structure and thus achieves the lowest space consumption. For *Count-adv*, its space usage is slightly higher than that of *HTC-PF+*. This is because *Count-adv* integrates preprocessing and counting into a unified process, dynamically maintaining only the neighbors of the currently processed hyperedge during hyper-triangle identification rather than storing the neighbors of all hyperedges. As a result, *Count-adv* achieves efficient hyper-triangle counting with modest space overhead.

6 RELATED WORK

Cohesive subgraph models. Cohesive subgraph discovery has been extensively studied in graph analysis [7, 19, 28, 41, 45, 51, 55, 60]. In graphs, the k-core model identifies vertex-cohesive subgraphs in which each vertex has degree at least k. This idea has been generalized to hypergraphs to capture higher-order relations. Arafat et al. propose the nbr-k-core [3], which enforces a minimum neighborhood size for each vertex, and further extend it to (k, q)-core [26] and (k, d)-core [3] by jointly considering degree and neighborhood constraints. However, these models do not explicitly account for the strength of vertex co-occurrence. To address this limitation, Kim et al. introduce the (k, g)-core [22], which jointly models neighborhood size and co-occurrence frequency to better capture frequent group interactions. Beyond vertex-centric models, the k-truss decomposition measures edge-level cohesiveness in graphs by requiring each edge to participate in at least $(k-2)$ triangles [2, 8, 9, 17, 21, 44, 48, 56, 59]. Wang et al. further study truss maintenance in dynamic hypergraphs [46], defining trussness via a parameter pair (k_{in}, k_{out}) to quantify intra- and inter-hyperedge cohesiveness of vertex pairs. To directly model hyperedge-level interactions, Qin et al. propose the hyper k-truss [34], which provides a finer-grained characterization of higher-order cohesiveness among hyperedges.

Triangle Counting. Triangle counting is a fundamental problem in graph analytics and plays a central role in understanding network properties [6, 20, 33, 40, 42, 49, 50]. Early studies propose enumeration-based algorithms that process edges in degree order and compute neighborhood intersections to avoid redundant traversal [10, 24]. Subsequent works explore alternative paradigms, including adjacency testing [37], matrix-based formulations [11], and orientation-based traversal that directs edges by vertex degrees to reduce the search space [53]. To further improve scalability, approximate and sampling-based methods are proposed to estimate triangle frequencies in large graphs [1, 43], while GPU-accelerated solutions exploit parallel intersection and optimized memory access [15]. Recent studies extend triangle counting from graphs to hypergraphs to capture higher-order interactions. A hyper-triangle is defined as a triplet of hyperedges that are pairwise intersecting [25, 52, 54]. Lee et al. [25] propose a basic enumeration framework that examines all hyperedge triplets, while Yin et al. [52] develop optimized techniques to accelerate the identification of dense hyper-triangles. More recently, Qin et al. [34] design an algorithm that focuses specifically on sparse hyper-triangles. Although various studies have investigated hyper-triangle counting, existing approaches still need to traverse open hyper-triangles when searching for hyper-triangles, leading to redundant computations.

7 CONCLUSION

In this paper, we study the problem of hyper-truss decomposition in hypergraphs and propose a generalized hyper k-truss model that supports both dense and sparse hyper-triangles, enabling the discovery of richer higher-order structural patterns. To efficiently perform decomposition, we develop a vertex-oriented counting strategy that avoids traversing open hyper-triangles during counting, as well as a vertex-based state propagation optimization that reduces redundant revisits during the peeling phase. As future work, we plan to further optimize the proposed framework on top of existing hypergraph processing systems, focusing on system-level optimizations and large-scale performance evaluation.

8 ACKNOWLEDGMENT

Kai Wang is the corresponding author. Kai Wang is supported by NSFC 62302294 and U2241211. Wenjie Zhang is supported by ARC DP230101445 and FT210100303. 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] Mohammad Almasri, Omer Anjum, Carl Pearson, Zaid Qureshi, Vikram S Mailthody, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. 2019. Update on k-truss decomposition on gpu. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[3] Naheed Anjum Arafat, Arijit Khan, Arpit Kumar Rai, and Bishwamittra Ghosh. 2023. Neighborhood-based hypergraph core decomposition. *arXiv preprint arXiv:2301.06426* (2023).

[4] Song Bai, Feihu Zhang, and Philip HS Torr. 2021. Hypergraph convolution and hypergraph attention. *Pattern Recognition* 110 (2021), 107637.

[5] Austin R Benson, Rediet Abebe, Michael T Schaub, Ali Jadbabaie, and Jon Kleinberg. 2018. Simplicial closure and higher-order link prediction. *Proceedings of the National Academy of Sciences* 115, 48 (2018), E11221–E11230.

[6] Paul Burkhardt. 2024. Triangle centrality. *ACM Transactions on Knowledge Discovery from Data* 18, 9 (2024), 1–34.

[7] Lijun Chang and Lu Qin. 2019. Cohesive subgraph computation over large sparse graphs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2068–2071.

[8] Huiping Chen, Alessio Conte, Roberto Grossi, Grigorios Loukides, Solon P Pissis, and Michelle Sweering. 2021. On breaking truss-based communities. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 117–126.

[9] Pei-Ling Chen, Chung-Kuang Chou, and Ming-Syan Chen. 2014. Distributed algorithms for k-truss decomposition. In *2014 IEEE International Conference on Big Data (Big Data)*. IEEE, 471–480.

[10] Norishige Chiba and Takao Nishizeki. 1985. Arboricity and subgraph listing algorithms. *SIAM Journal on computing* 14, 1 (1985), 210–223.

[11] 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.

[12] Yifan Feng, Haoxuan You, Zizhao Zhang, Rongrong Ji, and Yue Gao. 2019. Hypergraph neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 3558–3565.

[13] Sathyanarayanan Gopalakrishnan, Supriya Sridharan, Soumya Ranjan Nayak, Janmenjoy Nayak, and Swaminathan Venkataraman. 2022. Central hubs prediction for bio networks by directed hypergraph-ga with validation to covid-19 ppi. *Pattern Recognition Letters* 153 (2022), 246–253.

[14] Benjamin Heintz, Rankyung Hong, Shivangi Singh, Gaurav Khandelwal, Corey Tesdahl, and Abhishek Chandra. 2019. MESH: A flexible distributed hypergraph processing system. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 12–22.

[15] 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.

[16] Jin Huang, Rui Zhang, and Jeffrey Xu Yu. 2015. Scalable hypergraph learning and processing. In *2015 IEEE International Conference on Data Mining*. IEEE, 775–780.

[17] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-truss community in large and dynamic graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1311–1322.

[18] Masaaki Inoue, Thong Pham, and Hidetoshi Shimodaira. 2022. A hypergraph approach for estimating growth mechanisms of complex networks. *IEEE Access* 10 (2022), 35012–35025.

[19] Said Jabbour, Mourad Kmimech, and Badran Raddaoui. 2022. Discovering Overlapping Communities Based on Cohesive Subgraph Models over Graph Data. In *International Conference on Big Data Analytics and Knowledge Discovery*. Springer, 189–201.

[20] Said Jabbour, Nizar Mhadbhi, Badran Raddaoui, and Lakhdar Sais. 2018. Triangle-driven community detection in large graphs using propositional satisfiability. In *2018 IEEE 32nd International Conference on Advanced Information Networking and Applications (AINA)*. IEEE, 437–444.

[21] Humayun Kabir and Kamesh Madduri. 2017. Parallel k-truss decomposition on multicore systems. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[22] Dahee Kim, Junghoon Kim, Sungsu Lim, and Hyun Ji Jeong. 2023. Exploring cohesive subgraphs in hypergraphs: The (k, g)-core approach. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*. 4013–4017.

[23] 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.

[24] Matthieu Latapy. 2008. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical computer science* 407, 1-3 (2008), 458–473.

[25] Geon Lee, Jihoon Ko, and Kijung Shin. 2020. Hypergraph motifs: Concepts, algorithms, and discoveries. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2256–2269.

[26] Jongshin Lee, Kwang-Il Goh, Deok-Sun Lee, and B Kahng. 2023. (k, q)-core decomposition of hypergraphs. *Chaos, Solitons & Fractals* 173 (2023), 113645.

[27] 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.

[28] Yuan Li, Jinsheng Liu, Huiqun Zhao, Jing Sun, Yuhai Zhao, and Guoren Wang. 2021. Efficient continual cohesive subgraph search in large temporal graphs. *World Wide Web* 24, 5 (2021), 1483–1509.

[29] Qi Luo, Dongxiao Yu, Zhipeng Cai, Xuemin Lin, and Xiuzhen Cheng. 2021. Hypercore maintenance in dynamic hypergraphs. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2051–2056.

[30] Qi Luo, Dongxiao Yu, Yu Liu, Yanwei Zheng, Xiuzhen Cheng, and Xuemin Lin. 2023. Finer-grained engagement in hypergraphs. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 423–435.

[31] 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.

[32] Josh Payne. 2019. Deep hyperedges: a framework for transductive and inductive learning on hypergraphs. *arXiv preprint arXiv:1910.02633* (2019).

[33] Arnau Prat-Pérez, David Dominguez-Sal, Josep-M Brunat, and Josep-Lluis Larriba-Pey. 2016. Put three and three together: Triangle-driven community detection. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 10, 3 (2016), 1–42.

[34] Hongchao Qin, Guang Zeng, Rong-Hua Li, Longlong Lin, Ye Yuan, and Guoren Wang. 2025. Truss Decomposition in Hypergraphs. *Proceedings of the VLDB Endowment* 18, 7 (2025), 2185–2197.

[35] Emad Ramadan, Arijit Tarafdar, and Alex Pothen. 2004. A hypergraph model for the yeast protein complex network. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* IEEE, 189.

[36] Alicia Sanchez-Gorostiaga, Djordje Bajić, Melisa L Osborne, Juan F Poyatos, and Alvaro Sanchez. 2019. High-order interactions distort the functional landscape of microbial consortia. *PLoS biology* 17, 12 (2019), e3000550.

[37] 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.

[38] 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.

[39] Juliette Stehlé, Nicolas Voirin, Alain Barrat, Ciro Cattuto, Lorenzo Isella, Jean-François Pinton, Marco Quaggiotto, 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.

[40] Siddharth Suri and Sergei Vassilvitskii. 2011. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th international conference on World wide web*. 607–614.

[41] Xingyu Tan. 2024. *Cohesive Subgraph Detection and Search in Large Graphs*. Master's thesis. University of New South Wales (Australia).

[42] Eric A Treml and Patrick N Halpin. 2012. Marine population connectivity identifies ecological neighbors for conservation planning in the Coral Triangle. *Conservation Letters* 5, 6 (2012), 441–449.

[43] Charalampos E Tsourakakis, U Kang, Gary L Miller, and Christos Faloutsos. 2009. Doulion: counting triangles in massive graphs with a coin. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. 837–846.

[44] Chad Voegele, Yi-Shan Lu, Sreepathi Pai, and Keshav Pingali. 2017. Parallel triangle counting and k-truss identification using graph-centric methods. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[45] Kai Wang, Gengda Zhao, Wenjie Zhang, Xuemin Lin, Ying Zhang, Yizhang He, and Chunxiao Li. 2023. Cohesive subgraph discovery over uncertain bipartite graphs. *IEEE Transactions on Knowledge and Data Engineering* 35, 11 (2023), 11165–11179.

[46] Meng Wang, Qiang-Sheng Hua, Yefei Wang, Hai Jin, and Zhiyuan Shao. 2024. Parallel truss maintenance algorithms for dynamic hypergraphs. In *International Computing and Combinatorics Conference*. Springer, 538–550.

[47] 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.

[48] Jian Wu, Alison Goshulak, Venkatesh Srinivasan, and Alex Thomo. 2018. K-truss decomposition of large networks on a single consumer-grade machine. In *2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*. IEEE, 873–880.

[49] Xiaowei Xu, Nurcan Yuruk, Zhidan Feng, and Thomas AJ Schweiger. 2007. Scan: a structural clustering algorithm for networks. In *Proceedings of the 13th ACM*

SIGKDD international conference on Knowledge discovery and data mining. 824–833.

[50] 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.

[51] Haozhe Yin, Kai Wang, Wenjie Zhang, Dong Wen, Xiaoyang Wang, and Ying Zhang. 2023. Discovering Densest Subgraph over Heterogeneous Information Networks. In *Australasian Database Conference*. 341–355.

[52] Haozhe Yin, Kai Wang, Wenjie Zhang, Ying Zhang, Ruijia Wu, and Xuemin Lin. 2025. Efficient Computation of Hyper-triangles on Hypergraphs. *arXiv preprint arXiv:2504.02271* (2025).

[53] 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.

[54] 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).

[55] Gengda Zhao, Kai Wang, Wenjie Zhang, Xuemin Lin, Ying Zhang, and Yizhang He. 2022. Efficient computation of cohesive subgraphs in uncertain bipartite graphs. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2333–2345.

[56] Jun Zhao, Renjie Sun, Qiuyu Zhu, Xiaoyang Wang, and Chen Chen. 2020. Community identification in signed networks: a k-truss based model. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*. 2321–2324.

[57] Xiaoyao Zheng, Yonglong Luo, Liping Sun, Xintao Ding, and Ji Zhang. 2018. A novel social network hybrid recommender system based on hypergraph topologic structure. *World Wide Web* 21, 4 (2018), 985–1013.

[58] Jianming Zhu, Junlei Zhu, Smita Ghosh, Weili Wu, and Jing Yuan. 2018. Social influence maximization in hypergraph in social networks. *IEEE Transactions on Network Science and Engineering* 6, 4 (2018), 801–811.

[59] Ruicheng Zhu, Xintong Wang, Kai Wang, Fan Zhang, Zhengping Qian, and Long Yuan. 2025. Efficient k-Truss Breaking and Minimization. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE, 2628–2641.

[60] Yuanyuan Zhu, Qian Zhang, Lu Qin, Lijun Chang, and Jeffrey Xu Yu. 2020. Cohesive subgraph search using keywords in large networks. *IEEE Transactions on Knowledge and Data Engineering* 34, 1 (2020), 178–191.