Truss Decomposition in Hypergraphs

Hongchao Qin
Beijing Institute of Technology, China
hcqin@bit.edu.cn

Guang Zeng
Ant Group
senhua.zg@antgroup.com

Rong-Hua Li
Beijing Institute of Technology, China
rhli@bit.edu.cn

Longlong Lin
Southwest University, China
longlonglin@swu.edu.cn

Ye Yuan
Beijing Institute of Technology, China
yuan-ye@bit.edu.cn

Guoren Wang
Beijing Institute of Technology, China
wanggr@bit.edu.cn

ABSTRACT

Truss decomposition is a fundamental approach in graph theory that focuses on uncovering cohesive subgraphs within networks. However, many networks involve groupwise rather than pairwise relationships and are often represented as hypergraphs. Modeling and capturing *k-truss* in hypergraphs is essential for uncovering tight-knit relationships in such multi-relational networks. In this paper, we tackle the problem of truss decomposition in hypergraph. A *hyper k-truss* is a subgraph in which each node is part of at least k hyper-triangles. We first introduce a framework for hyper-truss decomposition and determine that the most time-consuming component is counting hyper-triangles. To count all hyper-triangles efficiently, we propose an edge-iterator algorithm. To further reduce redundant computations, we present an improved algorithm that combines edge-iterator and node-iterator techniques to prune non-promising nodes. Next, to handle common nodes in hypergraphs, we develop a novel prefix forest technique to encode all hyperedges and count triangles within this prefix forest. We also propose several optimization strategies that reorder nodes and hyperedges to improve work balancing. Finally, we conduct extensive experiments on real-world hypergraph datasets, demonstrating the efficiency and effectiveness of our algorithms.

PVLDB Reference Format:
Hongchao Qin, Guang Zeng, Rong-Hua Li, Longlong Lin, Ye Yuan, and Guoren Wang. Truss Decomposition in Hypergraphs. PVLDB, 18(7): 2185-2197, 2025.
doi:10.14778/3734839.3734854

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/VeryLargeGraph/HTRUSS.

1 INTRODUCTION

In network analysis, identifying cohesive substructures within large and complex networks is essential. Traditional graph-based methods, such as *k-core* [12] and *k-truss* [48] decompositions, have provided valuable insights by focusing on vertex connectivity and cohesive subgraphs. Among these, *k-truss* decomposition is particularly noteworthy for its ability to filter out less significant information from the *k-core*, thereby preserving the most cohesive parts of the network. However, despite having polynomial-time algorithms, existing methods for computing *k-truss* often struggle to scale effectively with the size of modern networks.

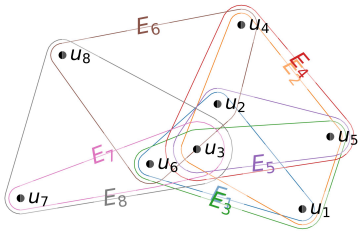


Figure 1: Example for a hypergraph

Expanding this concept to hypergraphs—where an edge can connect more than two vertices—introduces a new set of challenges and opportunities, as illustrated in Fig. 1. Hypergraphs generalize traditional graphs and are particularly useful for modeling complex relationships in domains such as bioinformatics, social networks, and information retrieval. The concept of truss decomposition in hypergraphs extends *k-truss* to these more complex structures, offering a richer and more nuanced understanding of network cohesion.

Truss decomposition in hypergraphs is an emerging field that seeks to develop efficient algorithms capable of handling the vast and intricate data associated with hypergraphs. In this extended model, a *hyper k-truss* is a subgraph in which each node is part of at least k hyper-triangles. This model more accurately captures the cohesiveness relations in a hypergraph. For example, in a co-purchases network (such as *Pinduoduo* or *Groupon*), a hyperedge can represent a group purchase, and a *hyper k-truss* can effectively capture the cohesiveness of various group purchase sizes.

However, according to the definition of *hyper k-truss*, counting triangles in hypergraphs more efficiently is of great importance for mining *hyper k-trusses* in hypergraphs. Inspired by the definition of traditional triangles in graphs, we define the triangle in hypergraphs as follows: a triangle is a triple of distinct hyperedges $\{e_1, e_2, e_3\}$ in which $e_1 \cap e_2, e_2 \cap e_3, e_1 \cap e_3 \neq \emptyset$, but $e_1 \cap e_2 \cap e_3 = \emptyset$. Counting the triangles in traditional graphs is a hot topic in recent years, and the theoretical time and space complexity of the iteration methods for triangle counting is $O(|E|^{1.5})$ and $O(|E|)$, where $|E|$ is the number of edges in the graph [18, 40]. However, most optimization methods are hard to apply to count the triangles in hypergraphs, since each hyperedge in a hypergraph has a large number of neighbor edges and nodes. Even worse, we observe that many hyperedges connect to most of the hyperedges in the hypergraph in many

Proceedings of the VLDB Endowment, Vol. 18, No. 7 ISSN 2150-8097.
doi:10.14778/3734839.3734854

datasets. Therefore, it is often very costly to count the triangles in hypergraphs, since we need to compute the interactions of each pairwise neighbor of hyperedges.

Note that, there are several implementations of the truss model. The general baseline Truss used in our experiments is based on the bipartite representation of a graph, where each hyperedge is transformed into a complete graph. For bipartite graph models, Kai Wang et al.[49] introduced *Bitruss*, where the key definition is that the butterfly support of each edge in a k-bitruss is greater than or equal to k. Additionally, regarding hypergraph trusses, Wang et al.[50] have previously studied truss models in hypergraphs. However, it has certain limitations in defining hypergraph triangles: given parameters α and β, it requires $E_x \cap E_y \cap E_z \geq \alpha$ and $E_x \cup E_y \cup E_z \geq \beta$, which now appears restrictive. Compared to the above models, our model is more general and better suited for modeling truss in native hypergraphs.

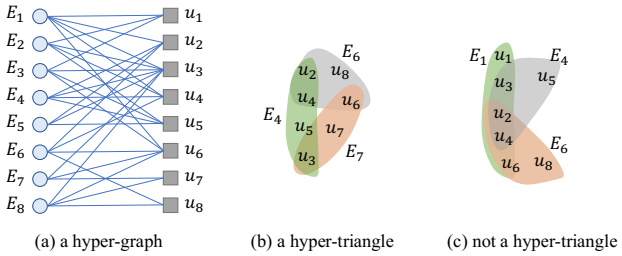
Contributions. In this paper, we formulate and provide efficient solutions for truss decomposition in a hypergraph. The main contributions of our work are summarized as follows.

(i) Novel Model. We introduce a model for k-trusses and triangles in hypergraphs and validate the definition. A *hyper k-truss* is a subgraph in which each node is part of at least k hyper-triangles. Through our framework for truss decomposition in hypergraphs, we identify that a crucial step in this process is counting all the hyper-triangles.

(ii) Efficient Algorithms. After introducing the framework for hyper-truss decomposition, we explore methods to efficiently count all hyper-triangles. Initially, we propose a baseline algorithm, HTC-B, which counts hyper-triangles using an edge-iterator approach. However, this method proves inefficient as it requires enumerating all neighbors of the hyperedges, and often, a hyperedge is connected to a large number of other hyperedges. To mitigate redundant computations, we integrate edge-iterator and node-iterator techniques, leading to an improved algorithm, HTC-P. We note that many hyperedges share common nodes but are computed independently during enumeration. To address this, we design a novel prefix forest to encode all hyperedges and introduce HTC-PF, which enumerates triangles within this prefix forest. Furthermore, we have developed several optimization techniques, incorporating node or edge ordering heuristics, to further accelerate our counting algorithm.

(iii) Extensive experiments. We conduct extensive experiments on 10 real-world datasets to evaluate our algorithms. The experimental results show that *i)* Our algorithm of truss decomposition in hyper graphs is only slightly slower than truss decomposition in traditional graph, and is faster than other comparative methods; *ii)* The hyper-triangles counting algorithm HTC-PF is much faster than the baseline method HTC-B. Particularly, HTC-PF is about 10-100 times faster than the HTC-B, and HTC-PF only consumes 0.31 seconds to count the triangles in a co-authorship hypergraph DBLP with 3.7 million hyper-edges; *iii)* Through a case study on DBLP, *hyper k-truss* demonstrates a clearer and more precise depiction of tight relationships compared to *k-truss*.

Organization. Section 2 introduces the model and formulates our problem. The framework of the hyper truss decomposition and the algorithms to efficiently counting hyper-triangles are proposed in Section 3 and 4. Experimental studies are presented in Section 5, and the related work is discussed in Section 6. Section 7 draws the conclusion of this paper.



(a) a hyper-graph (b) a hyper-triangle (c) not a hyper-triangle

Figure 2: The bipartite form of the hypergraph in Fig.1

2 PRELIMINARIES

Let $\mathbb{G} = (V_{\mathbb{G}}, E_{\mathbb{G}})$ be a hypergraph, in which $V_{\mathbb{G}}$ is the set of nodes, $E_{\mathbb{G}}$ is the set of hyperedges $(E_1, E_2 ...)$. Obviously, $|E_{\mathbb{G}}| \leq 2^{|V|}$. We consider a **simple** and **undirected** hypergraph in this manuscript, so one hyperedge $E_x \in E_{\mathbb{G}}$ is structured as an unordered set of nodes $(v_{x_1}, v_{x_2} ...)$ and it represents a set of $|E_x|$ nodes that take interactions. For convenient, we record x to be the ID of the hyperedge E_x, and mark the nodes of each hyperedge $(v_{x_1}, v_{x_2}, v_{x_3}, ...)$ in ascending order such that $(x_1 < x_2 < x_3 < ...)$. Fig. 2(a) illustrates a hypergraph with 8 nodes and 8 hyperedges. We can see that a hypergraph can be easily presented by a bipartite graph, and the two separated parts in Fig. 2(a) are nodes and hyperedges. The hyper-triangle can be defined as follows.

Definition 1 (hyper-triangle). *Given a hypergraph $\mathbb{G} = (V_{\mathbb{G}}, E_{\mathbb{G}})$, a hyper-triangle $\triangle$ in $\mathbb{G}$ is a triple of hyperedges $\{E_x, E_y, E_z\} \subseteq E_{\mathbb{G}}$ in which $E_x \cap E_y \neq \emptyset, E_x \cap E_z \neq \emptyset, E_y \cap E_z \neq \emptyset$ but $E_x \cap E_y \cap E_z = \emptyset$.*

Note that in Definition 1, we can observe that the hyper-triangle has two properties: (i) each pair of hyperedges has nodes in common; (ii) the triple of hyperedges has no node in common. Those properties have become a consensus in hypergraph theory [9, 26, 37]. The property (i) is intuitive, such that each pair of edges are connected inside each triangle $\triangle$. Consider a triple of hyperedges which satisfies property i but dissatisfies property ii, the triple becomes a shape of triple star ⊦(as Fig. 2(c) shown). In topology, the *Betti Number* [36] (the maximum number of cuts that can be made without dividing a surface into two separate pieces) of $\triangle$ and ⊦are 1 and 0, respectively. We can find that $\triangle$ is much more stable than ⊦, such that the *Betti Number* of $\triangle$ is higher than ⊦. Therefore, the hyper-triangle in a hypergraph must have the two properties described in Definition 1.

Here, we introduce some useful concepts for a hypergraph $\mathbb{G}$. Let $V_{\mathbb{G}}(E_x)$ be the set of nodes in hyperedge E_x, $E_{\mathbb{G}}(u)$ be the set of hyperedges which contains u, $n = |V_{\mathbb{G}}|$ and $m = |E_{\mathbb{G}}|$ be the number of nodes and hyperedges, $\mathsf{N}^V_{\mathbb{G}}(v) = v \cup \{u | \exists E_x \in E_{\mathbb{G}}, \{u, v\} \subseteq E_x\}$ be the set of neighbor nodes of node v, $\mathsf{N}^E_{\mathbb{G}}(E_x) = \{E_c | E_c \in E_{\mathbb{G}}, E_c \cap E_x \neq \emptyset\}$ be the set of neighbor edges of edge E_x, and $\mathsf{D}^V_{\mathbb{G}}(v) = |\mathsf{N}^V_{\mathbb{G}}(v)|$, $\mathsf{D}^E_{\mathbb{G}}(E_x) = |\mathsf{N}^E_{\mathbb{G}}(E_x)|$ be the degree of node v and hyperedge E_x in $\mathbb{G}$. The **capacity** of one hyperedge $\mathsf{C}(E_x)$ is the

number of nodes inside E_x, which equals $|V_{\mathbb{G}}(E_x)|$. The example below shows the explanations of the notations.

Example 1. *Fig. 2(a) shows a hypergraph $\mathbb{G}$ in which $V_{\mathbb{G}} = \{u_1, u_2, ..., u_8\}$ and $E_{\mathbb{G}} = \{E_1, E_2, ..., E_8\}$. Consider hyperedge E_1, we can find that $V_{\mathbb{G}}(E_1) = \{u_1, u_2, u_3, u_4, u_6\}$, $V_{\mathbb{G}}(E_2) = \{u_1, u_3, u_4, u_5\}$ and so on. Note that node u_1 is in hyperedges E_1, E_2 and E_3, we can see that $E_{\mathbb{G}}(u_1) = \{E_1, E_2, E_3\}$ and $\mathsf{N}^V_{\mathbb{G}}(u_1) = \{u_1, u_2, u_3, u_4, u_6\} \cup \{u_1, u_3, u_4, u_5\} \cup \{u_1, u_3, u_5, u_6\} = \{u_1, u_2, ..., u_6\}$. In addition, $\mathsf{N}^E_{\mathbb{G}}(E_1) = \{E_{\mathbb{G}}(u_1) \cup E_{\mathbb{G}}(u_2) \cup E_{\mathbb{G}}(u_3) \cup E_{\mathbb{G}}(u_4) \cup E_{\mathbb{G}}(u_6)\} = \{E_1, E_2, ..., E_8\}$. Fig. 2(b) is the illustration for hyperedges E_4, E_6 and E_7. We can observe that $E_4 \cap E_6 = \{u_2, u_4\}$, $E_4 \cap E_7 = \{u_3\}$, $E_6 \cap E_7 = \{u_6\}$ and $E_4 \cap E_6 \cap E_8 = \emptyset$. So, the triple of hyperedges can form a hyper-triangle. In Fig. 2(c), although $E_1 \cap E_4, E_1 \cap E_6, E_4 \cap E_6 \neq \emptyset$, but $E_1 \cap E_4 \cap E_6 = \{u_2, u_4\}$. Therefore, the union of E_1, E_4, E_6 can not be a hyper-triangle.*

On the basis of hyper-triangles, we define the support of a hyperedge as follows.

Definition 2 (support). *Given a hypergraph $\mathbb{G} = (V_{\mathbb{G}}, E_{\mathbb{G}})$, the support of a hyperedge $E \in E_{\mathbb{G}}$ in $\mathbb{G}$, denoted by $sup(E, \mathbb{G})$, is the number of hyper-triangles which contain hyperedge E, i.e. $sup(E, \mathbb{G}) = |\{\triangle | E \in \triangle, \triangle \subseteq E_{\mathbb{G}}\}|$.*

In a traditional graph, a k-truss is the maximal subgraph where every node each edge is contained in at least k-2 triangles. In the following, we define the hyper k-truss below.

Definition 3 (hyper k-truss). *Given a hypergraph $\mathbb{G} = (V_{\mathbb{G}}, E_{\mathbb{G}})$, a hyper k-truss in $\mathbb{G}$, denoted by HT_k, is a maximal subgraph $\mathbb{H}$, such that $\forall E \in E_{\mathbb{G}}, sup(E, \mathbb{H}) \geq k - 2$.*

Besides, the hyper k-truss number of the edge E, denoted by $htn(E)$, is the maximal number of k such that there is a hyper k-truss containing E but no hyper $(k+1)$-truss containing it.

Problem Statement. The goal of truss decomposition (denoted by HTRUSS) in a hypergraph $\mathbb{G}$ is to compute the hyper k-truss number for each edge in $\mathbb{G}$.

Challenges. The problem of truss decomposition in hypergraphs is similar to truss decomposition in traditional graphs. However, the algorithm for hypergraph truss decomposition, faces several computational challenges, particularly related to the efficiency of the initial support calculation, i.e. the hyper-triangle counting in the whole hypergraph. Counting hyper-triangles in hypergraphs is analogous to counting triangles in traditional graphs, but direct application of traditional methods is inadequate. In conventional graphs, triangle counting algorithms typically list all adjacent edge pairs (wedges) and check which pairs form triangles. Given that the average degree of nodes in these graphs is not excessively high, traditional algorithms have a time complexity of $O(|E|^{1.5})$.

However, the situation is more complex in hypergraphs. Each hyperedge may have a large number of neighboring edges, and we need to verify whether the intersection $E_x \cap E_y \cap E_z$ is empty, which can result in a time complexity of up to $O(|E|^3)$. Thus, the primary challenges are *(i)* efficiently pruning impossible candidates among the neighboring edges and *(ii)* determining whether the intersection $E_x \cap E_y \cap E_z$ is empty using a more efficient method.

Addressing these challenges is crucial for the HTRUSS algorithm. The efficiency of the algorithm heavily relies on overcoming these difficulties in hyper-triangle counting. In the following sections, we will introduce several novel filtering techniques and an efficient enumeration algorithm based on a prefix forest to tackle these issues, thereby improving the overall performance and scalability of the HTRUSS algorithm for large-scale hypergraph analysis.

Algorithm 1: HTRUSS($\mathbb{G}, k$)

Input: Hyper-graph $\mathbb{G} = (V_{\mathbb{G}}, E_{\mathbb{G}})$
Output: The hyper k-truss number for $E \in E_{\mathbb{G}}$, i.e. $htn(E)$

1 $htn \leftarrow \emptyset$; $\mathbb{E} \leftarrow E_{\mathbb{G}}$;
2 **[Computing all the support for each edge]**
3 **for** *each* $E \in \mathbb{E}$ **do**
4 $htn[E] \leftarrow$ support for each hyperedge E;
// Equivalent to counting all the hyper-triangles, we can invoke the HTC algorithm in the next section
5 **for** $k = 3 : C^2_{|V_{\mathbb{G}}|}$ **do**
6 **while** $\exists E \in \mathbb{E}$ *satisfying* $htn[E] < k - 2$ **do**
7 $\mathbb{E}.remove(E)$;
8 **for** $E' \in \mathsf{N}^E_{\mathbb{G}}(E)$ *in an ascending order of* htn **do**
9 **if** $\exists E^* \in \mathsf{N}^E_{\mathbb{G}}(E) \cap \mathsf{N}^E_{\mathbb{G}}(E')$ *s.t.* $E^* \cap E' \cap E = \emptyset$ **then**
10 $htn(E') \leftarrow htn(E') - 1$;
11 **if** $\mathbb{E} = \emptyset$ **then** break for;
12 **return** htn;

3 HYPER TRUSS DECOMPOSITION FRAMEWORK

Based by a peeling method, we present a framework for truss decomposition in a hypergraph, as shown in Algorithm 1. The HTRUSS algorithm begins by initializing an empty map htn to store the truss number for each hyperedge and a working set $\mathbb{E}$ containing all hyperedges from the input hypergraph $\mathbb{G}$. For each hyperedge $E \in \mathbb{E}$, it computes the support, which counts the number of hyper-triangles that include E, and stores this support value in the htn map (lines 3-4). The algorithm iterates through possible values of k starting from 3 up to $C^2_{|V_{\mathbb{G}}|}$. For each k, it repeatedly removes hyperedges E from $\mathbb{E}$ that have a support less than $k-2$ (lines 5-6). After removing a hyperedge E, the support values of its neighboring hyperedges E' (those that share vertices with E) are decremented by 1, processing the neighbors in ascending order of their support values to ensure correct updates (lines 7-10). Note that, after removing an edge, the support reduces only when its neighbors can form a hyper-triangle (lines 9). If $\mathbb{E}$ becomes empty, the algorithm terminates early for the current k (line 11). Finally, the algorithm returns the htn map, which contains the hyper k-truss number for each hyperedge in the hypergraph. This framework efficiently decomposes a hypergraph into its k-truss components, providing a scalable method for analyzing large and complex hypergraphs.

Next, we analyze the complexity of the HTRUSS algorithm. In the initialization step, which involves initializing the htn map and the set of hyperedges $\mathbb{E}$, takes $O(|E_{\mathbb{G}}|)$, where $|E_{\mathbb{G}}|$ is the number of hyperedges in the hypergraph $\mathbb{G}$. For each hyperedge $E \in \mathbb{E}$, the support is calculated by counting the number of hyper-triangles it participates in. So we record the hyper-triangles counting time

to be $T(\#\triangle)$. Then, the peeling process involves an outer loop running from $k = 3$ to $C^2_{|V_{\mathbb{G}}|}$, but in practice, the peeling process terminates early, so the upper bound is not always reached. For each k, the while-loop iterates over the hyperedges and removes those with support less than $k-2$, considering each hyperedge once in the worst case. Removing a hyperedge and updating the support values of its neighboring hyperedges takes $O(d)$ time per hyperedge, where d is the maximum degree of the hypergraph. The inner for-loop updates the support values for affected hyperedges, involving processing neighbors in ascending order of support, which can be efficiently managed using appropriate data structures (e.g., priority queues), resulting in a time complexity of $O(d \log d)$ per hyperedge removal. The condition $\mathbb{E} = \emptyset$ is checked in each iteration of the outer for-loop, taking constant time $O(1)$.

Combining these steps, we get the overall worst-case time complexity of the HTRUSS algorithm as $O(T(\#\triangle) + |E_{\mathbb{G}}| \cdot d \log d)$, where $|E_{\mathbb{G}}|$ is the number of hyperedges and d is the maximum degree of the hypergraph. However, in the next section, we will show that our proposed best algorithm needs about $O\left(\frac{|E_{\mathbb{G}}|^3}{|V_{\mathbb{G}}|}\right)$ to count all the hyper-triangles in the hypergraph. Thus, the overall time complexity of the HTRUSS algorithm largely depends on the efficiency of hyper-triangles counting.

Efficiently counting hyper-triangles is crucial because it directly impacts the initialization phase where the support for each hyperedge is calculated. If the hyper-triangle counting is optimized, it can significantly reduce the time complexity of the HTRUSS algorithm. This efficiency gain can lead to better performance in real-world applications, especially for large-scale hypergraphs where the number of hyperedges and vertices is substantial. Therefore, improving hyper-triangle counting methods can make the HTRUSS algorithm more practical and scalable for massive hypergraph datasets. In the next sections, we will delve into the details of our optimized hyper-triangle counting techniques and demonstrate their impact on the overall performance of the HTRUSS algorithm.

4 HYPER-TRIANGLE COUNTING ALGORITHMS

In this section, we first introduce a basic framework to count the hyper-triangles. Then, we propose an advanced framework which can reduce the triangles computation by recording the interactions of the former hyperedges. Next, we develop an improved algorithm which can first store the predecessor neighbors in a prefix forest, then filter and count the hyper-triangles by trees in the constructed forest. The details of the proposed algorithms are shown below.

4.1 The HTC-Basic Algorithm

To solve the problem of counting hyper-triangles, there are two intuitive methods: edge-iterator and node-iterator based counting framework. The first one is to iterator all the edges and find which triple of edges can be a hyper-triangle. Recall Definition. 1, we can enumerate the edge in $E_{\mathbb{G}}$ to be E_x, and then search the neighbor of E_x to find E_y and E_z. The second one is to iterator all the nodes and find which triple of nodes can be contained in different hyperedges. We can enumerate the node u in $V_{\mathbb{G}}$, search $E_{\mathbb{G}}(u)$ to get a neighbor

Algorithm 2: HTC-B($\mathbb{G}$)

Input: Hyper-graph $\mathbb{G} = (V_{\mathbb{G}}, E_{\mathbb{G}})$
Output: The numbers of all the hyper-triangles in $\mathbb{G}$

```
1 count ← 0;
2 for each E_x ∈ E_G in parallel do
3     N^E_G(E_x) ← {E_c | E_c ∈ E_G, E_c ∩ E_x ≠ ∅};
4     for each E_y ∈ N^E_G(E_x) s.t. y < x do
5         for each E_z ∈ N^E_G(E_x) s.t. z < y do
6             if E_y ∩ E_z ≠ ∅ and E_x ∩ E_y ∩ E_z = ∅ then
7                 count = count + 1;
8 return count;
```

v, and search the common neighbor w of u and v to search hyper-triangles. However, this method may result in complex judgments for whether the hyper-triangle shared by u and v is same as v and w; whether the hyper-triangle formed by (u, v, w) is same as that by (u', v', w'), and so on.

Therefore, we propose an edge-iterator based framework as a baseline to count the hyperedges, as shown in Algorithm 2. It first enumerates the edge $E_{\mathbb{G}}$ to be E_x in parallel (line 2). Then, the algorithm initializes a hyperedge set, $\mathsf{N}^E_{\mathbb{G}}(E_x)$, which contains all the hyperedges whose intersection with E_x is not empty (line 3). Next, it enumerates the edges E_y and E_z in $\mathsf{N}^E_{\mathbb{G}}(E_x)$ to be another two edges in the hyper-triangle (lines 4-5), and the value of *count* will increase 1 if $E_y \cap E_z \neq \emptyset$ and $E_x \cap E_y \cap E_z = \emptyset$. Note that, the three hyperedges $\{E_x, E_y, E_y\}$ in a hyper-triangle need to satisfy $z < y < x$ to avoid redundant computations.

Theorem 4.1 (**Complexity of** HTC-B). *The time and space complexity of Algorithm 2 are $O(\overline{C}m^3)$ and $O(\overline{C}m)$, respectively, in which m and $\overline{C}$ are the number and average capacity of hyperedges.*

Proof. The algorithm first enumerates the hyperedges in $E_{\mathbb{G}}$, such that it will have m iterations. Then, it needs to calculate the neighour edges of hyer-edges E_x in line 3, of which the time complexity is certainly bounded by $O(m)$. Next, the algorithm enumerates $\mathsf{N}^E_{\mathbb{G}}(E_x)$ twice to search the proper E_y and E_z in lines 4-5, such that it will have at most m^2 iterations. In line 6, it needs $O(\overline{C})$ to compute $E_y \cap E_z$ and $E_x \cap E_y \cap E_z$, in which $\overline{C}$ is the average capacity of all the hyperedges. To sum up, Algorithm 2 needs the time complexity of $O(m \times (m + \overline{C}m^2)) = O(\overline{C}m^3)$ to count all the hyperedges. Consider the occupation of space, Algorithm 2 needs not to store any other structures expect the hypergraph, such it only needs space of $O(\overline{C}m)$. □

4.2 The HTC-Plus Algorithm

To reduce the redundant computations, the current advanced triangle counting algorithm consists of two major steps: orientation and computation. Orientation steps focus on reducing the redundant counting since $\{E_1, E_2, E_3\}$ and $\{E_3, E_2, E_1\}$ are same triangles. Computation steps focus on reducing the cost of calculating the interaction of hyperedges. Here we introduce HTC-P, a Plus version of the hyper-triangle counting algorithm.

There are three optimization features in the algorithm HTC-P: (i) We design a partial ordering relation for all the hyperedges,

Algorithm 3: HTC-P($\mathbb{G}$)

Input: Hyper-graph $\mathbb{G} = (V_{\mathbb{G}}, E_{\mathbb{G}})$
Output: The numbers of all the hyper-triangles in $\mathbb{G}$

1 $count \leftarrow 0$;
2 **for** *each* $E_x \in E_{\mathbb{G}}$ *in parallel by a decreasing order* **do**
3 $Prec \leftarrow \emptyset$;
4 $\mathsf{N}^E_{\mathbb{G}}(E_x) \leftarrow \{E_c | E_c \in E_{\mathbb{G}}, E_c \cap E_x \neq \emptyset\}$;
5 **for** $E_c \in \mathsf{N}^E_{\mathbb{G}}(E_x)$ *s.t.* $E_c \prec E_x$ **do**
6 $Prec[E_c] \leftarrow E_c \cap E_x$;
7 **for** $E_y \in \mathsf{N}^E_{\mathbb{G}}(E_x)$ *s.t.* $E_y \prec E_x$ *and* $Prec[E_y] \neq \emptyset$ **do**
8 $HSet \leftarrow \emptyset$;
9 **for** *each* $v \in (E_y \setminus Prec[E_y])$ **do**
10 **for** $E_z \in E_{\mathbb{G}}(v)$ *s.t.* $E_z \prec E_y$ **do**
11 **if** $Prec[E_z] = \emptyset$ *or* $z \in HSet$ **then**
12 **contine**;
13 $HSet \leftarrow HSet \cup z$;
14 **if** $Prec[E_y] \cap Prec[E_z] = \emptyset$ **then**
15 $count = count + 1$;
16 **return** *count*;

and search the hyper-triangles in which the hyperedges satisfy $E_z \prec E_y \prec E_x$. Suppose all the nodes in the hyperedges are ordered in increasing order of their IDs. One hyperedge $E_y \prec E_z$ if there exists a k such that the indices of the nodes from 0 to k are the same, and the index of the $(k+1)$-th node in E_z is greater than that in E_y. For example, the hyperedge $u_1, u_3, u_4, u_5 \prec u_1, u_4, u_5$ because the index of the first node is the same ($1 = 1$), and the index of the fourth node is $3 < 4$. (*ii*) Recall in HTC-B, we enumerate all the neighbor edges of E_x to seek E_y and E_z. However, the enumeration will result in redundant computations for computing the interactions. So, we compute and store the common nodes for $E_x \cap E_y$ and $E_x \cap E_z$ to reduce the considerate nodes. (*iii*) While searching the last hyperedge E_z in the triangle, we enumerate the nodes in E_y to filter more impossible edges. As it is hard to avoid orientation while enumerating nodes, we set some flag values which help us ensure that there are no repeat countings.

Algorithm 3 shows the details of the improved counting algorithm. Same as HTC-B, it first enumerates the edge $E_{\mathbb{G}}$ to be E_x in parallel (line 2). Then, HTC-P initializes a set *Prec* to record the interaction of E_x and its neighbors, a set *Flag* to store the loop number of the edges, and a set $\mathsf{N}^E_{\mathbb{G}}(E_x)$ to record hyperedges which are connected with E_x (lines 3-6). Next, as shown in the third optimization above, the algorithm enumerates E_y in $\mathsf{N}^E_{\mathbb{G}}(E_x)$ (line 7), and searches node v in E_y to find E_z (line 9). Note that, if E_z has been searched through one node v, z will be in *HSet* (line 11), such that if $|E_y \cap E_z| \geq 2$, the triangle will not be counted more than once (lines 11-13). In the last step of the loop, if $Prec[E_y] \cap Prec[E_z] = \emptyset$, then $E_x \cap E_y \cap E_z = \emptyset$ and the value of *count* will increase 1 (line 15). The following example shows the process of Algorithm 3 while search hyper-triangles in Fig. 2(a).

Example 2. *Fig. 3 shows the process of counting hyper-triangles in Fig. 2(a). Consider hyperedge* $E_x = E_8$, $\mathsf{N}^E_{\mathbb{G}}(E_8) = \{E_1, E_2, E_3, E_4, E_5, E_7\}$. *Suppose* $E_y = E_7$, *we can see* $Prec[E_7] = \{u_3, u_6, u_7\}$ *and it is not* $\emptyset$. *However, each item in* E_7

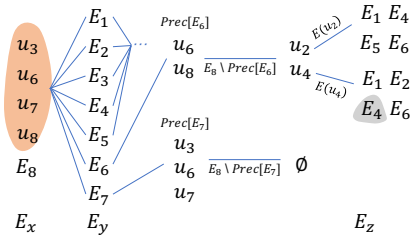


Figure 3: Running example for HTC-P

are all in E_8 *such that the loop continues to consider* $E_y = E_6$. *We can get* $Prec[E_6] = \{u_6, u_8\}$, *and there are two nodes* $\{u_2, u_4\}$ *which are in* E_6 *but not in* E_8. *Consider* $E_{\mathbb{G}}(u_4) = \{E_1, E_2, E_4, E_6\}$, *we can find that only* E_4 *satisfies that* $Prec[E_4] \cap Prec[E_6] = \emptyset$, *such that there is a hyper-triangle while* $E_z = E_4$. *Consider* $E_{\mathbb{G}}(u_2) = \{E_1, E_4, E_5, E_6\}$, *we can find that* E_4 *satisfies that* $Prec[E_4] \cap Prec[E_6] = \emptyset$, *but* $Flag[E_4]$ *has been recorded as* $flag$. *So, this hyper-triangle with* $E_z = E_4$ *has been counted before and it will continue to loop as lines 11-12 show. Combining the edge-iterator and node-iterator, Algorithm 3 reduces several redundant computations to count the hyper-triangles.* □

Theorem 4.2 (**Complexity of** HTC-P). *The time and space complexity of Algorithm 3 are* $O(\frac{\overline{C}^3}{n}m^3)$ *and* $O(\overline{C}m)$, *respectively, in which* n *is the number of nodes,* m *and* $\overline{C}$ *are the number and average capacity of hyperedges.*

Proof. The algorithm first enumerates the hyperedges in $E_{\mathbb{G}}$ which requires m iterations (line 2). Then, it calculates all the *Prec* set of E_x, which needs $O(\overline{C}m)$ time (lines 4-6). Next, it searches each E_y in $\mathsf{N}^E_{\mathbb{G}}(E_x)$ and each v in E_y (lines 7-9), which has loops of $O(\overline{C}m)$. In lines 10-15, the algorithm enumerates the hyperedges which v is in, and the average number of iterations is $O(\frac{\overline{C}m}{n})$. Note that, all $Prec[E_z]$ has been calculated before, and computing $Prec[E_y] \cap Prec[E_z]$ needs $O(\overline{C})$ time. To sum up, Algorithm 3 needs the time complexity of $O(m \times (\overline{C}m + \overline{C}m \times \frac{\overline{C}m}{n} \times \overline{C})) = O(\frac{\overline{C}^3}{n}m^3)$. Consider the occupation of space, Algorithm 3 stores all the temporary *Prec* set of E_x, which needs at most $O(\overline{C}m)$, such that the space complexity of Algorithm 2 is also $O(\overline{C}m)$. □

Note that, in a real-life hypergraph the average capacity of hyperedges is much less than the node of the hypergraphs, such that $O(\frac{\overline{C}^3}{n}m^3)$ will be much less than $O(\overline{C}m^3)$ in real applications. The experiments will show that Algorithm 3 is much efficient than Algorithm 2 in practice.

4.3 The HTC-Prefix Forest Algorithm

Although Algorithm 3 is efficient in practice, it still has two limitations: (*i*) Algorithm 3 still needs to enumerate the neighbors of E_x in line 7 first, and then uses the sets of *Prec* and *Flag* to reduce the interactions of hyperedges. As the $\mathsf{N}^E_{\mathbb{G}}(E_x)$ may contain large amounts of hyperedges, we need to reduce it before the enumerations. (*ii*) In hypergraphs, most hyperedges may share several common nodes, but they are all computed separately during the enumerations. For example, consider hyperedges $E' = \{v_1, v_2, v_3\}$ and $E'' = \{v_1, v_2, v_3, v_4\}$, clearly they have three common

Algorithm 4: BuildPrefixForest($\mathbb{G}$)

Input: Hyper-graph $\mathbb{G} = (V_{\mathbb{G}}, E_{\mathbb{G}})$
Output: A prefix forest which encodes all the edges.

1 $Entrance \leftarrow \emptyset$;
2 **for** *each* $E_x \in E_{\mathbb{G}}$ *in an increasing order of edges* **do**
3 **if** $|E_x| = 1$ **then contine**;
4 $Node \leftarrow Entrance$;
5 **for** $u \in E_x$ **do**
6 **if** $u \in Node.next.keys()$ **then**
7 $Node \leftarrow Node.next[u]$; **contine**;
8 **else**
9 $Node.next \leftarrow Node.next \cap \{u : \emptyset\}$;
10 $Node \leftarrow Node.next[u]$;
11 **if** $u = E_x.end()$ **then**
12 $Node.elabel \leftarrow E_x$;
13 **return** $Entrance$;

nodes. Suppose $E^* = \{v_1, v_2, v_5\}$ and $E^{**} = \{v_3, v_5\}$, we can find that there are two hyper-triangles $\{E', E^*, E^{**}\}$ and $\{E'', E^*, E^{**}\}$. It can be observed that node v_4 does not participate in the construction of triangles, so E' and E'' are same while constructing the triangles. As many hyperedges will have overlaps of nodes, we need to merge some calculations during the loops.

To overcome those limitations, we propose a novel counting algorithm, HTC-PF, which relies on *Prefix Forests* to record the hyperedges and finds the triangles in the forests. The striking features of HTC-PF are twofold: (i) On one hand, it needs not to enumerate the neighbor of each hyperedge to search the triangles. Instead, it enumerates some root of trees to find the next hyperedge of the triangle. (ii) On the other hand, HTC-PF can reduce the redundant computations while the hyperedges have overlap nodes since those two similar hyperedges come from one branch in the prefix tree. Below, we first introduce how to build the prefix forest in Algorithm 4, and then describe the HTC-PF in Algorithm 5.

Building the Prefix Forest. Suppose that the nodes in hyperedges are sorted in an increasing order of their IDs. Algorithm 4 first initializes a pointer *Entrance* to record the start of the forest (line 1). Then, it encodes each hyperedges into the forest (lines 2-12). In line 3, we eliminate hyperedges with a single node in order to reduce the number of single-node trees in the prefix forest, as these do not contribute to the formation of triangles. In each loop, the algorithm literately searches the node u in E_x. If the current node u is in $Node.next.keys()$, then u is already in the branch of the tree and the loop continues (lines 6-7). Otherwise, there is a new branch starting at u in the tree (line 9). If u is the last item of E_x, then the *elabel* of the current node is marked as E_x (lines 11-12). After all the hyperedges are encoded into the trees, the algorithm returns the pointer *Entrance* to be the start of the prefix forest. The following example shows the process of building the prefix forest.

Example 3. *Fig. 4 shows the prefix forest which encodes the hypergraph in Fig. 2(a). To construct the prefix forest, algorithm* BuildPrefixForest *initializes an entrance of the forest, and literately adds the hyperedges into it. The algorithm first adds E_1 into the forest, such that we can get the left branch. Next, it will encode*

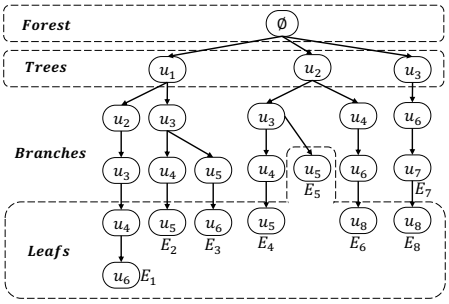


Figure 4: The prefix forest which encodes the hyperedges in Fig. 2(a)

$E_2 = \{u_1, u_3, u_4, u_5\}$ into the forest. We can see that the current $Node = \emptyset$ (line 4) and $\emptyset \rightarrow = \{u_1\}$, so $Node = \emptyset \rightarrow u_1$. Then, consider u_3 in E_2, u_3 is not in $\emptyset \rightarrow u_1 \rightarrow$, so we create a new node u_3 and add it into $\emptyset \rightarrow u_1 \rightarrow$. If it comes to the end of E_2, the node will be marked by an elabel, such that all the leaf in the forest will have an elabel. Note that, not all the elabels are on leafs. Consider E_7 and E_8, E_7 is totally a prefix of E_8, such that E_7 is the branch of $\emptyset \rightarrow u_3 \rightarrow u_6 \rightarrow u_7$, but E_8 is $\emptyset \rightarrow u_3 \rightarrow u_6 \rightarrow u_7 \rightarrow u_8$. We can also observe that the path from $\emptyset$ to the node which has elabel represents one hyperedge in the hypergraph. □

Theorem 4.3 (**Complexity of** BuildPrefixForest). *The time and space complexity of Algorithm 4 are $O(\overline{C}m)$ and $O(\overline{C}m)$, respectively. m and $\overline{C}$ are the number and average capacity of the hyperedges.*

Proof. In Algorithm 4, each node in the hypergraph will be considered to be encoded into the prefix forest once, such that the time complexity of Algorithm 4 is $O(\overline{C}m)$. In addition, the prefix forest will not store any other structure, so the space complexity of Algorithm 4 is $O(\overline{C}m)$. □

Counting the hyper-triangles in the Prefix Forest. After encoding all the hyperedges by building the prefix forest, we propose two-stage depth-first searches for constructing the hyper-triangles. The depth-first searches are divided into two kinds. The first kind, DFS-Same, is to search the other hyperedges in the same tree as a given hyperedge E_x. The second kind, DFS-Different, is to search the other hyperedges in tree T' for a given hyperedge E_x in T ($T \neq T'$). In the following, we store and record all the temporary interaction of two hyperedges E_x, E_y by $Prec[E_x, E_y]$, which is significant to record the results of the depth-first searches.

For a given hyperedge E_x, there are three situations for the position of other two hyperedges E_y and E_z: (i) E_y is the prior node in the same tree of E_x, and E_z is in another tree. (ii) E_y and E_z are in the same tree, but this tree is different from the tree of E_x. (iii) E_x, E_y and E_z are all in different trees.

For situation (i), since E_y and E_x are in the same tree, so we can invoke the DFS-S once in the tree which has E_x to get all the $Prec[E_x, E_y]$. Under this situation, E_z will certainly on the other tree, otherwise E_z will share at least one node $E_x.first()$ with E_x and E_y. To search E_z, we can invoke the DFS-D in the trees whose root is less than $E_x.first()$. We can also use $Prec[E_x, E_y]$ to early terminate the DFS-D if the visiting node is already in $Prec[E_x, E_y]$.

Algorithm 5: HTC-PF($\mathbb{G}$)

Input: Hyper-graph $\mathbb{G} = (V_{\mathbb{G}}, E_{\mathbb{G}})$
Output: The numbers of all the hyper-triangles in $\mathbb{G}$

```
1  Forest ← BuildPrefixForest(𝔾);
2  Trees ← Forest.next(); count ← 0; Prec ← ∅;
3  for tree_1 ∈ Trees in parallel do
4      for E_x ∈ tree_1.leafs() do
5          [Situation (i)]
6          DFS([tree_1, ∅, ∅, ∅], Prec, E_x, ∅, ∅);
7          for E_y ∈ tree_1.leafs() s.t. E_y ≺ E_x do
8              for tree_2 ∈ Trees s.t. tree_2 ≺ tree do
9                  DFS([tree_2, ∅, ∅, ∅], Prec, E_x, E_y, count);
10         [Situation (ii)]
11         for tree_2 ∈ Trees s.t. tree_2 ≺ tree do
12             if tree_2.first() ∉ E_x then contine;
13             DFS([tree_2, ∅, ∅, ∅], Prec, E_x, ∅, ∅);
14             for E_y ∈ tree_2.leafs() do
15                 DFS([tree_2, ∅, ∅, ∅], Prec, E_x, E_y, count);
16                 [Situation (iii)]
17                 for tree_3 ∈ Trees s.t. tree_3 ≺ tree_2 do
18                     DFS([tree_3, ∅, ∅, ∅], Prec, E_x, E_y, count);
19 return count;

20 Procedure DFS([node, Visited, P, P_2], Prec, E_x, E_y, count)
21 if node.elabel ≠ ∅ then
22     Prec(E_x, E_{node.elabel}) ← P;
23     if E_y ≠ ∅ then Prec[E_y, E_{node.elabel}] ← P_2;
24     if count ≠ ∅ and P ∪ P_2 ≠ ∅ then count ← count + 1;
25 for u ∈ node.next() s.t. !Visited[u] do
26     Visited[u] ← True;
27     if u ∈ E_x then P ← P ∪ u;
28     if E_y ≠ ∅ then
29         if u ∈ Prec[E_x, E_y] then continue;
30         if u ∈ E_y then P_2 ← P_2 ∪ u;
31     DFS([u, Visited, P, P_2], Prec, E_x, E_y, count);
32     Visited[u] ← False;
```

For situation (ii), for a given hyperedge E_x, we can first invoke DFS-D to calculate and record the $Prec[E_x, E_y]$ on each tree. Then, for each E_y in one given tree, we can call the DFS-S once to count all the hyper-triangles in the tree.

For situation (iii), since E_x, E_y and E_z are in the different trees, we need to invoke DFS-D first to calculate and record the *Prec* of all hyperedges E' which satisfy $E' \prec E_x$. Then, we must invoke DFS-D for a second time to check whether hyperedges $\{E''|E'' \prec E'\}$ satisfy $Prec[E'', E'], Prec[E'', E_x] \neq \emptyset$ and $E'' \notin Prec[E', E_x]$.

Since it is hard to performed powerful optimization strategies in situation (iii), the following example shows how to conduct the two-stage *DFS* under the first and second situations above.

Example 4. *Fig. 5 (i) shows the first situation of the triple hyperedges. Suppose $E_x = E_6$, E_y are all in the tree started at u_2. We can invoke the* DFS-S *to get $Prec[E_5] = \{u_2\}$ and $Prec[E_4] = \{u_2, u_4\}$. Then, we find E_z in the tree started at u_1. The branch $u_1 \rightarrow u_2 \rightarrow$ is pruned, since all the possible hyperedge E_y will have u_2 in Prec. If $E_y = E_5$, we call* DFS-D *in tree started at u_1, and we can find that both E_2 and E_3 can be E_z, such that there are two hyper-triangles $\{E_6, E_5, E_2\}$ and $\{E_6, E_5, E_3\}$. If $E_y = E_4$, only E_3 can be E_z. So, we get a hyper-triangle $\{E_6, E_4, E_3\}$. Next, if $E_x = E_5$, we can see that $Prec[E_4] = \{u_2, u_3\}$, and then all the branches are pruned in the tree started at u_1. Therefore, there is no triangles when $E_x = E_5, E_y = E_4$.*

Fig. 5 (ii) shows the second situation of the triple hyperedges. Suppose $E_x = E_8$, we can find that the branch of $u_2 \rightarrow u_3 \rightarrow$ will only support one hyperedge in the triangle since u_3 is in E_8. If $E_y = E_6$, we can invoke DFS-S *once to find all the interaction of E_6 and hyperedges in the tree started at u_2. We can find that $E_6 \cap E_5 = \{u_2\}$, $E_6 \cap E_4 = \{u_2, u_4\}$. Recall in Fig. 5 (i), we have the similar results. So, we can invoke* DFS-S *in each tree once to get all the interaction of edges in the same tree. Next, we can find that $E_6 \cap E_5 \notin E_8$ and $E_6 \cap E_4 \notin E_8$, such that there are two triangles $\{E_8, E_6, E_4\}$ and $\{E_8, E_6, E_5\}$ in Fig. 5 (ii).* □

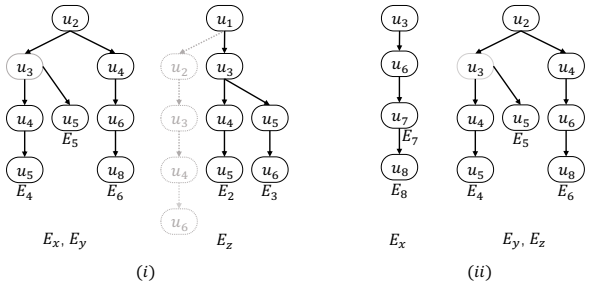


Figure 5: Different situations of hyperedges in the prefix forest

Algorithm 5 shows the details of counting hyper-triangles in the prefix forest. It first initializes the prefix forest by invoking Algorithm 4 (line 1), sets the final results by *count*, and records the temporary interaction set by *Prec* (line 2). Next, the algorithm starts to carry out the two-stage depth-first searches for each $tree_1$ in the forests. The calculations in lines 6-9 correspond to the situation (i) when E_y and E_x are in the same tree, but E_z is in the different tree. It calls DFS first to calculate all the *Prec* in the first stage in the same tree as $tree_1$ (line 6), and then uses it to count the triangles in another tree $tree_2$ (line 9). The calculations in lines 10-18 correspond to situations (ii) and (iii) as described above. There are two parts in this process: one part invokes DFS on $tree_2$ twice since E_y and E_z are in the same tree (lines 13,15), the other part invokes DFS on $tree_2$ and $tree_3$ since E_x, E_y and E_z are in different trees (lines 13,18).

Consider the procedure DFS, it has several parameters in which *node* is the current searched nodes in the tree, *Visited* records the set of the visited nodes in the depth-first search, P,P_2 are the temporary paths of $Prec[E_x, node]$ and $Prec[E_y, node]$, *Prec* is the set which store $Prec[E_x, E_y]$ for the pair of hyperedges and *count* is the number of the triangle countings. The procedure first checks whether the *elabel* of the current node is $\emptyset$ (line 21). If so, the depth-first search has find a hyperedge and record the iteration path to be the *Prec* of the hyperedge E_x. If the DFS is in the second stage (lines 9,15,18), E_y and *count* will not be $\emptyset$ and the *count* will increase (lines 23-24). Then, DFS searches the node u to be the next

node of the current *Node* (line 25), and searches all the branches by depth-first (lines 26-32). For a given hyperedge E_x, we can get all the *Prec* of hyperedges in the same tree after invoking the DFS which is accumulated in line 27. Note that, in procedure DFS, the branch of searches will be pruned once the node is already in the interaction $Prec[E_x, E_y]$ (lines 29).

In conclude, HTC-PF separates the triple of hyperedges in the prefix forest into three situations, and proposes several two-stage depth-first searches for counting the hyper-triangles.

Theorem 4.4 (**Complexity of** HTC-PF). *The time and space complexity of Algorithm 5 are $O(\frac{\overline{C}^3}{nx}m^3)$ and $O(\overline{C}m^2)$, respectively, in which x is the number of the trees in the prefix forest, m and $\overline{C}$ are the number and average capacity of hyperedges.*

Proof. In Algorithm 5, it first invokes Algorithm 4 to build the prefix forest, which requires $O(\overline{C}m)$ time. Then, it enumerates the leafs on $tree_1$ to be E_x which has $\overline{C}m$ iterations in total (lines 3-4). In line 6, it needs $O(\overline{C}m)$ time to construct the $Prec[E_x, E_y]$ for all E_y in $tree_1$. Next, the algorithm needs to perform DFS on all $tree_2$, so the total time of lines 7-9 in all the loops is $O(\frac{\overline{C}^2m^2}{x^2n})$ in which x is the number of trees. In lines 11-15, the average number of E_y in line 9 is $\frac{\overline{C}m}{nx}$, and the time of invoking the DFS will be $\frac{\overline{C}m}{x}$(line 15). In lines 16-18, searching the nodes by DFS in all trees will need $O(\overline{C}m)$. To sum up, the time complexity of Algorithm 5 is $O(\frac{\overline{C}^3}{nx^2}m^3)$ *(Situation (i),(ii))* + $O(\frac{\overline{C}^3}{nx}m^3)$ *(Situation (iii))* = $O(\frac{\overline{C}^3}{nx}m^3)$. Consider the occupation of space, Algorithm 5 will store the prefix forest and the *Prec* sets for all pairs of the hyperedges, which in total consume the space complexity of $O(\overline{C}m^2)$. □

4.4 Optimizations

In this section, firstly, we discuss the effect of the nodes' orders on the efficiency for the hyper-triangle counting algorithms. Secondly, we examine and analyze a finer-grain for the parallel computations.

4.4.1 Node Ordering Heuristic. To reduce the computation costs of HTC-PF, we need to avoid counting the hyper-triangles in situation (iii) in section 4.3 while the triple of hyperedges are in different trees. To address this problem, we can re-order the nodes to get different prefix forests, which are significant to counting the hyper-triangles in algorithm HTC-PF. Below, we discuss the non-increasing degree ordering strategy.

The non-increasing degree order. Given a hypergraph $\mathbb{G}$, we can order the nodes in $\mathbb{G}$ by sorting the nodes in a non-increasing of degree. If the nodes are in the non-increasing degree order, consider two nodes v_x, v_y in $V_{\mathbb{G}}$ $(x < y)$, we can get $D^V_{\mathbb{G}}(v_x) \geq D^V_{\mathbb{G}}(v_y)$.

We observe that the ordering of nodes has great impact on the structure of the prefix forest, which can be seen in the modified prefix forest of Fig. 6(b).

Example 5. *Consider the hypergraph in Fig. 2(a), we can observe that $D^V_{\mathbb{G}}(u_3) = 7$, which has highest degree among all the nodes. Thus, we reorder the nodes in the hypergraph into $\{v_1, v_2, ...v_8\}$ in Fig. 6(a), which corresponds to $\{u_3, u_6, u_2, u_4, u_5, u_1, u_7, u_8\}$ in Fig. 2(a). Recall the partial ordering relation of hyperedges in Section 4.2, we can insert all the hyperedges into the prefix forest by an increasing of the partial ordering. Also, from Fig. 6(b) we can see $\{E_1 \prec E_3 \prec E_7 \prec E_8 \prec E_4 \prec E_5 \prec E_2 \prec E_6\}$. Surprisingly, we find that there are only two trees in the Fig. 6(b), such that we need not to consider the situation (iii) in section 4.3 any longer. So, the hyper-triangles must have E_6 since the triple of hyperedges can not be in one same tree. Therefore, we can easily find that there are 10 hyper-triangles, which are $\{E_6, E_2, E_5\}$, $\{E_6, E_2, E_7\}$, $\{E_6, E_2, E_8\}$, $\{E_6, E_2, E_3\}$, $\{E_6, E_5, E_8\}$, $\{E_6, E_5, E_7\}$, $\{E_6, E_5, E_3\}$, $\{E_6, E_4, E_8\}$, $\{E_6, E_4, E_7\}$ and $\{E_6, E_4, E_3\}$.*

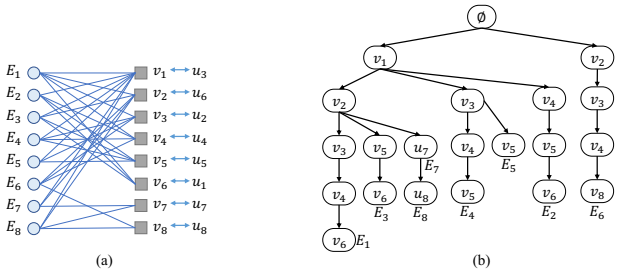


Figure 6: The prefix forest with the nodes in a non-increasing degree order

Remark. Consider the complexity of Algorithm. HTC-PF, if two node all in one same tree, the time complexity will be $O(\frac{\overline{C}^3}{nx^2}m^3)$. Otherwise if three nodes are in different trees (situation *(iii)*), the time complexity will be $O(\frac{\overline{C}^3}{nx}m^3)$. So we can merge more trees to have better performance. The non-increasing degree order of nodes can reduce the running time of HTC-PF in practice, which will be shown in the experiments in Section 5.

4.4.2 Parallelization of Finer-grained. In this paper, all the proposed hyper-triangle counting algorithms are implemented in parallel, as shown in line 2 of Algorithm 2, 3 and line 3 of 5. For example, if the number of cores in the CPU is $\mathbb{T}$, then theoretically HTC-PF will have the running time of $O(\frac{1}{\mathbb{T}}\frac{\overline{C}^3}{nx}m^3)$. However, since the the workloads in the trees can continue to be divided and set *Prec* is maintained in HTC-PF, which gives rise to unbalanced workloads and access delay on different cores. We consider the situations below to improve the shortcomings.

(i) Both the number of leafs on $tree_1$, $tree_2$ are small numbers. In general, if the number leafs on $tree_1$ and $tree_2$ are smaller than a constant τ, the stride factor of different cores are also small. In this case, the overhead of partitioning the forests from merge-based algorithm takes the majority of the runtime. Consider the set *Prec* in the shared memory, it can be cached by the shared memory with the average cost on caching reducing to a small number, such that all cores can read it by a small number of coalesced memory reads.

(ii) The number of leafs on $tree_1$ is small number, while the number of leafs on $tree_2$ is large number. When $|tree_1| << |tree_2|$, one of the two vertices has large degree while the other has small degree. Obviously, we can first compute the *Prec* on $tree_1$ and store it in the shared memory. Then, we can read *Prec* in the cache, and partition the nodes on $tree_2$ for better parallelization.

(iii) Both the number leafs on $tree_1$ and $tree_2$ are large numbers. In this case, the workloads are much heavy, and more value must be stored in *Prec*. However, we can still partition the trees quickly since we can separate it from the deep layer of the tree. This process can be done by a breadth-first search on the trees. To overcome the

Table 1: Statistics of datasets

Dataset	$\|\mathbb{E}\| = m$	$\|V\| = n$	$\frac{m}{n}$	$\hat{C}$	$\overline{C}$
NS	112,919	5,557	20.32	187	9.98
CHS	172,035	328	524.49	5	2.32
TAU1	192,947	200,975	0.96	14	2.3
TAU2	271,233	3,030	89.51	5	3.42
TMS1	719,792	201,864	3.56	5	3.49
TMS2	822,059	1,630	504.33	21	2.6
CMG	1,591,166	1,261,130	1.261	284	3.76
CMH	1,813,147	1,034,877	1.75	925	3.09
DBLP	3,700,681	1,930,379	1.91	280	3.33
TSO	11,305,343	3,455,074	3.27	25	2.64

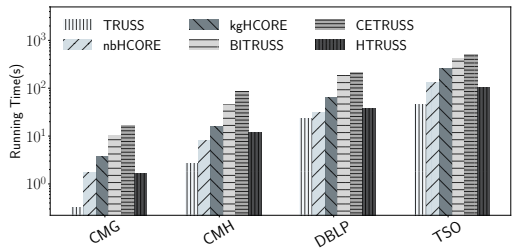


Figure 7: Runtime of different algorithms (in single thread)

problems that *Prec* is much larger, we can just store the temporary *Prec* in the separate trees, just like the steps in HTC-P.

Remark. Consider Algorithm HTC-PF, we only do the parallel computations in line 3. Furthermore, we can still separate the computations of the *for* loops in line 4,7,11,14 and 17. The discussion above details how to split the computations when the considered two trees are in different sizes. The experiments in Section 5 show the impacts of this improvement.

5 EXPERIMENTS

In this section, we conduct extensive experiments to evaluate the proposed algorithms.

For hyper truss decomposition, we compare six different algorithms: 1) TRUSS [48]: the fastest truss decomposition algorithm (transform each hyperedge into a clique and then apply TRUSS). 2) nbHCORE [3]: the latest neighborhood-based hypergraph core decomposition method. 3) kgHCORE [25]: the (k, g)-core decomposition in the hypergraph. 4) BITRUSS [49] : the k-bitruss in a bipartite graph. CETRUSS [50] : the (k, α, β)-truss in a hypergraph. 5) HTRUSS: the fastest algorithm proposed in this paper (a combination of Alg. 1 and Alg. 5).

For hyper-triangles counting (HTC),we implement six different algorithms for comparison: *(i)* HTC-B is a baseline that counts the hyper-triangles by a naive edge-iterator based framework, which is proposed in Algorithm. 2. *(ii)* HTC-P is an improved algorithm that improves HTC-B by considering several orientation and computation steps, which is proposed in Algorithm. 3. *(iii)* HTC-PF is a tricky algorithm that relies on a prefix forests to record the hyper-edges and finds the triangles in the forests, which is proposed in Algorithm. 5. *(iv)* HTC-PFO_1 is a variation of Algorithm. 5 which reorders the nodes in the hyper-edges in section 4.4.1. *(v)* HTC-PFO_2 is a variation of Algorithm. 5 which optimizes the hyper-edges with different workloads in section 4.4.2. *(vi)* HTC-PF+ is a variation of Algorithm. 5 which combines the benefits of HTC-PFO_1 and HTC-PFO_2.

All algorithms are implemented with C++ and compiled using gcc version 11.1.0 with optimization level set to O3. All the experiments are conducted on a Linux kernel 4.4 server with an AMD Threadripper 3990X of 64 cores and 128 GB memory. When quantity measures are evaluated, the test was repeated over 5 times and the average is reported here.

Datasets. We use 10 different real-world hypergraphs in the experiments. The detailed statistics of our datasets are summarized in Table 1, where $|\mathbb{E}| = m$ denotes the number of the hyperedges, $|V| = n$ is the number of the nodes, $\hat{C}$ denotes the maximum number of the hyperedge's capacity, and $\overline{C}$ represents the average number of the hyperedge's capacity. All the datasets are downloaded from https://www.cs.cornell.edu/~arb/data/, and they are listed in an increasing order of number of hyperedges. NS (*NDC-substances*) is the datasets of substances making up drugs. CHS (*contact-high-school*) describes groups of people in contact at a high school. TAU1 (*threads-ask-ubuntu*) is the datasets of users asking and answering questions on threads on askubuntu.com. TAU2 (*tags-ask-ubuntu*) is a set of tags applied to questions on askubuntu.com. TMS1 (*tags-math-sx*) records the sets of tags applied to questions on math.stackexchange.com. TMS2 (*threads-math-sx*) is the sets of users asking and answering questions on threads on math.stackexchange.com. CMG (*coauth-MAG-Geology*) and CMH (*coauth-MAG-History*) are co-authorships on Geology and History papers. DBLP (*coauth-DBLP*) records the co-authorship on DBLP papers. TSO (*threads-stack-overflow*) is a set of users asking and answering questions on threads on stackoverflow.com.

5.1 Efficiency Testings of HRTUSS

Exp-1. Runtime of different algorithms (in single thread). Fig. 7 shows the Runtime of our proposed best HTRUSS algorithm compared to TRUSS, nbHCORE, kgHCORE, BITRUSS and CETRUSS on four hypergraphs. Similar results can also be observed on the other datasets. Note that, the above algorithms are all considered *SOTA* methods for dense sub-hypergraph decomposition, but they may not produce identical results. The goal of this experiment is to evaluate the efficiency of our algorithm in comparison to these established methods. As illustrated in the figure, our algorithm even outperforms the core decomposition algorithms kgHCORE for hypergraphs on all four datasets, while kgHCORE incorporates g-distance influence constraints. Compared to the bipartite truss models (BITRUSS) and hyper-truss models (CETRUSS), our approach demonstrates significantly better performance. This is because these models rely on motif counting—BITRUSS requires counting butterflies, while CETRUSS counts (α, β)–triangles. In contrast, our algorithm leverages an efficient prefix forest to minimize redundant computations, giving it a clear advantage in speed. Additionally, our algorithm's runtime is slower than TRUSS by 0.5 orders of magnitude. This is because the number of triangles in traditional graphs is much larger than in hypergraphs. Since the peeling time does not significantly impact overall performance, it indicates that our pruning technique for hyper-triangle counting is very effective.

Exp-2. Runtime of peeling time vs. counting time in HTRUSS. Table. 2 analyzes the time spent on different components of the best HTRUSS algorithm, including $t_{reading}$: the time to read the graph; $t_{counting}$: the time to count hyper-triangles, i.e., compute

Table 2: Runtime (s) of different components in HTRUSS ($t_{counting}$ part: 64x in parallel , other parts: in single thread)

Dataset	$t_{reading}$	$t_{counting}$	$t_{peeling}$	t_{total}	$\frac{t_{counting}}{t_{total}}$
NS	0.01	0.029	0.083	0.122	23.77%
CHS	0.011	0.033	0.099	0.143	23.08%
TAU1	0.18	0.0042	0.0138	0.198	2.12%
TAU2	0.2	20.78	3.23	24.21	85.83%
TMS1	0.45	2.12	0.67	3.24	65.43%
TMS2	0.43	87.49	3.2	96.45	90.71%
CMG	1.08	0.17	0.13	1.38	12.32%
CMH	1.21	0.012	0.1	1.322	0.91%
DBLP	2.18	0.31	0.15	2.64	11.74%
TSO	4.56	50.25	17.13	71.94	69.85%

Table 3: Runtime (s) of different algorithms (64x in parallel)

Dataset	HTC-B	HTC-P	HTC-PF	#hyper-triangles
NS	0.34	0.14	**0.029**	572,714,845
CHS	0.73	0.20	**0.033**	660,585,969
TAU1	0.27	0.052	**0.0042**	4,757,191
TAU2	474.58	21.24	**20.78**	1,440,582,225
TMS1	59.42	4.71	**2.12**	1,400,810,790
TMS2	899.85	88.69	**87.49**	38,323,775,462
CMG	5.08	0.47	**0.17**	39,213,491
CMH	1.21	0.055	**0.012**	836,375
DBLP	11.58	1.71	**0.31**	70,022,805
TSO	INF	389.74	**50.25**	4,636,096,640

the initial support of all edges; $t_{peeling}$: the time for the peeling process as described in Alg. 1; and t_{total}: the total time. The table also shows the percentage of $t_{counting}$ relative to t_{total}. The results indicate that $t_{peeling}$ is consistently shorter than $t_{counting}$ across all datasets, confirming the theoretical analysis from previous sections. Additionally, in the TAU2, TMS1, TMS2, and TSO datasets, $t_{counting}$ accounts for over 60% of t_{total} due to the higher degree of nodes in these datasets. Furthermore, the following experimental table (Table. 3) shows that the number of hyper-triangles in the TAU2, TMS1, TMS2, and TSO datasets is significantly higher compared to other datasets.

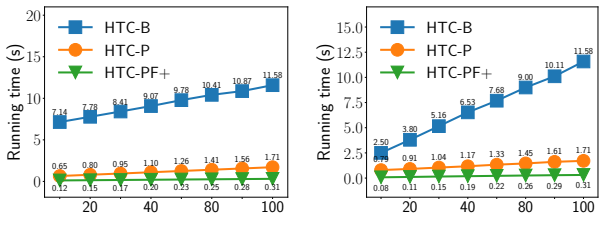
5.2 Efficiency Testings of HTC

Exp-3. Runtime of HTC-B, HTC-P and HTC-PF. Table. 3 evaluates the Runtime with 64 threads in parallel of HTC-B, HTC-P and HTC-PF in the 10 real-world hypergraphs, in which INF represents that the algorithm can not be done in 1 day. We invoke each algorithm for 10 times in different datasets, and then record the average time of computing once. From Table. 3, we can see that HTC-PF is much faster than HTC-P and HTC-B. This is because that we use a prefix forest encoding to speed up counting the hyper-triangles in HTC-PF. Note that, in TAU2 and TMS2, the counting time is slower than the other datasets and HTC-PF is slightly faster than HTC-P, since the average degree $\frac{m}{n}$ in TAU2 and TMS2 is very high such that HTC-PF needs to compute most interactions of the hyperedges. We can also see that HTC-P is slightly faster than HTC-B, since it records the *Prec* to reduce some redundant computations for the interactions. As can be seen, on DBLP with 3.7 million hyperedges, HTC-B takes only 11.58 seconds and our proposed algorithm HTC-PF only consumes 0.31 seconds. These results confirm that our proposed algorithms are indeed very efficient on large real-life hyper-graphs.

Exp-4. The impact of the optimization techniques. Table. 4 evaluates the Runtime with 64 threads in parallel of HTC-PF, HTC-PFO$_1$, HTC-PFO$_2$ v.s. HTC-PF+. From Table. 4, we can see

Table 4: The impact of the optimizations (64x in parallel)

Dataset	HTC-PF	HTC-PFO$_1$	HTC-PFO$_2$	HTC-PF+
NS	0.029	0.023	0.025	**0.021**
CHS	0.033	0.027	0.031	**0.023**
TAU1	0.0042	0.0032	0.0037	**0.0030**
TAU2	20.78	17.13	18.13	**17.03**
TMS1	2.12	1.83	2.02	**1.81**
TMS2	87.49	62.69	83.12	**58.69**
CMG	0.17	0.15	0.17	**0.15**
CMH	0.012	0.011	0.012	**0.010**
DBLP	0.31	0.26	0.29	**0.24**
TSO	50.25	37.32	48.97	**36.97**



(a) Percents of nodes (b) Percents of hyperedges

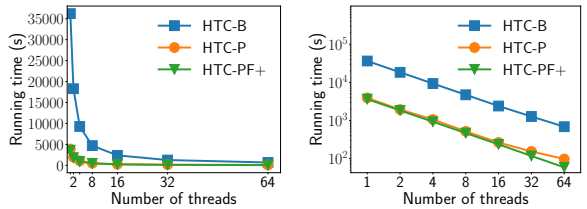
Figure 8: Scalability of the proposed algorithms on DBLP

that HTC-PFO$_1$, HTC-PFO$_2$ and HTC-PF+ are much faster than HTC-PF. This is because that we use the re-ordering and balanced strategies to speed up HTC-PF. We observe that HTC-PFO$_1$ is much faster than HTC-PFO$_2$ in all the datasets, such that the strategy of re-ordering nodes can benefit more than the work balance. Note that, the $\frac{time(\text{HTC-PFO}_1)}{time(\text{HTC-PF})}$ is much lower on TAU2 and TMS2, such that the impact of re-ordering on TAU2 and TMS2 is much higher. It may be the reason that there are nodes with high degree on TAU2 and TMS2, such that we can re-order the nodes to largely reduce the number of trees in the prefix forest. These results suggest that the proposed optimizations can improve the speed of HTC-PF.

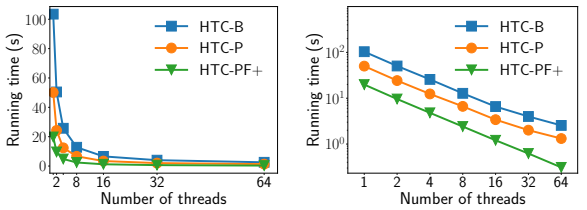
Exp-5. Scalability and parallelizability. Fig. 8 shows the scalability testings with 64 threads in parallel of HTC-B, HTC-P and HTC-PF+ on the DBLP dataset. Similar results can also be observed on the other datasets. We first generate ten hypergraphs by randomly picking 10%-100% of the nodes and the hyperedges, and evaluate the Runtimes of the proposed algorithms on those subgraphs. As shown in Fig. 8(a)-(b), the Runtime increases smoothly with increasing numbers of nodes or hyperedges. Also, we can see that HTC-PF+ is significantly faster than HTC-B and HTC-P with all datasets of different scale, which is consistent with our previous findings. These results suggest that our proposed algorithms are scalable when handling large hypergraphs.

Fig. 9 shows the parallel performance of HTC-B, HTC-P and HTC-PF+ on datasets TMS2 and DBLP. Similar results can also be observed on the other datasets. We change the threads number of the CPU by 1, 2, 4, 8, 16, 32 and 64 to test the parallelizability of those algorithms. As expected in Fig. 9(b),(d), when the Runtimes are listed in log-scale, the Runtime of all the algorithms decreases linearly as the number of threads increases. This is because that all the calculations can be carried out asynchronously. These results confirm that the proposed algorithms are good in parallelizability.

Exp-6. Memory overhead. Fig. 10 shows the memory usage of HTC-B, HTC-P and HTC-PF+ on different datasets. The y-axis datas are in log scale for better visualization. The algorithm HTC-B



(a) TMS2 (b) TMS2 (y-axis in log scale)



(c) DBLP (d) DBLP (y-axis in log scale)

Figure 9: Parallel performance of the proposed algorithms

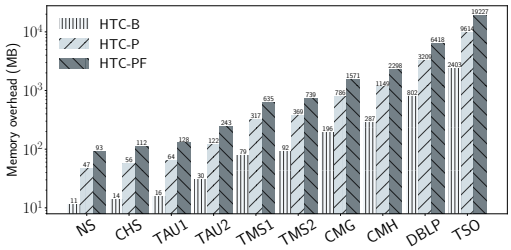


Figure 10: Memory overhead of HTC-B, HTC-P and HTC-PF+

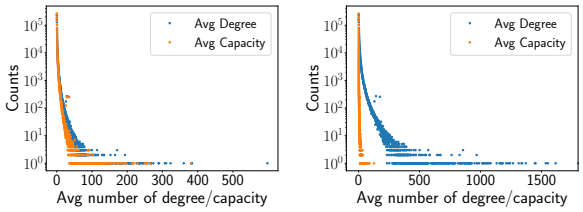
only needs to store the hypergraph, such that its memory overhead is close to the size of the hypergraph. We can see that the memory usages of HTC-P and HTC-PF+ are higher than the memory usage of HTC-B, because HTC-P needs to store the $Prec$ of E_x and HTC-PF+ needs to store the prefix forest and the $Prec[E_x, E_y]$ for pairs of (E_x, E_y) during the counting process. In practice, in HTC-P and HTC-PF+, we can free the memory of $Prec$ once the hyper-triangles start at E_x has been counted. Therefore, on large datasets, the memory usages of HTC-P and HTC-PF+ are typically lower than ten times of the size of the hypergraph. For instance, on DBLP, HTC-PF+ consumes 6,419MB memory while HTC-B needs 802MB. These results indicate that all the proposed algorithms can achieve near linear space complexity, which confirms our theoretical analysis in Section 3.

5.3 Effectiveness Testings

Exp-7. Density of different dense sub-hypergraph models. Table 5 presents a comparative analysis of the density values across different models, including TRUSS, nbHCORE, kgHCORE, BITRUSS, CETRUSS, and our proposed HTRUSS, across various hypergraph datasets. The results demonstrate that HTRUSS consistently achieves the highest density in all cases, highlighting its ability to capture stronger structural connectivity. Notably, nbHCORE generally exhibits higher values than kgHCORE, suggesting it may better reflect the dynamics of core-periphery

Table 5: Density of different dense sub-hypergraph models

Dataset	TRUSS	nbHCORE	kgHCORE	BITRUSS	CETRUSS	HTRUSS
NS	91.26	123.45	115.75	109.60	110.25	**153.10**
CHS	100.87	146.80	139.50	124.00	125.25	**170.50**
TAU1	0.86	1.40	1.30	1.10	1.13	**1.70**
TAU2	162.45	216.20	204.80	192.00	197.25	**247.60**
TMS1	4.90	7.10	6.60	6.20	6.30	**9.00**
TMS2	123.57	217.20	204.50	172.32	183.25	**246.50**
CMG	1.90	3.20	2.80	2.53	2.59	**3.50**
CMH	2.80	4.00	3.80	3.54	3.49	**4.60**
DBLP	3.15	4.70	4.40	4.22	4.01	**5.20**
TSO	3.98	6.10	5.70	5.12	5.20	**6.50**



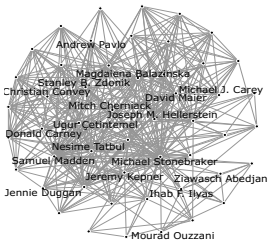
(a) DBLP (b) TMS2

Figure 11: Distributions of average degree and capacity of hyper-triangles

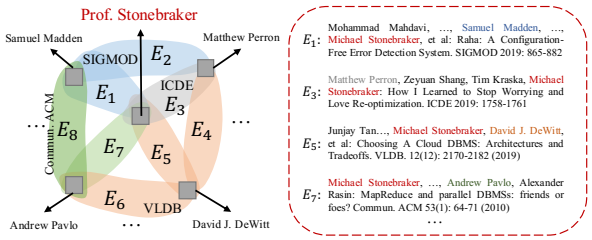
structures. This trend is particularly evident in datasets like CHS and TMS2, where nbHCORE significantly outperforms kgHCORE. Overall, these findings emphasize the effectiveness of HTRUSS in providing a more robust and comprehensive measure of hypergraph cohesiveness compared to other baseline models.

Exp-8. Distributions of average degree and capacity of hyper-triangles. In real-world, the degree in a graph always follows the power-law distribution. In Fig. 11, we collect all the hyper-triangles on DBLP and TMS2 and show the distributions of average degree and capacity of hyper-triangles. We can observe that both the average degree or capacity follow power-law distributions. Similar results can also be observed on the other datasets. Note that, the means of the average degree in Fig. 11 (b) is higher than that in Fig. 11 (a), but the means of the average capacity in Fig. 11 (b) is lower than that in Fig. 11 (a). It is because that dataset TMS2 has larger average degree and lower capacity than DBLP, which can be observed in Table. 1. In Fig. 11 (a), we can find that the average degree and capacity of most hyper-triangles are no more than 200, and in Fig. 11 (a) the average degree and capacity of most hyper-triangles are no more than 200 and 500. The results indicate that the average degree and capacity will be bounded by a constant, which guarantees the efficiency of our proposed algorithm.

Exp-9. Case study of hyper k-truss on DBLP. Fig. 12 shows the effectiveness the *hyper k-truss* model. In Fig. 12(a), we first construct a traditional graph on DBLP and connect two authers once they have co-operated in one article. Then, we invoke the traditional *k-truss* method to get the cohesive subgraph. In Fig. 12(b), the *hyper k-truss* is a subgraph in which each node is in at least k hyper-triangles. Fig. 12 shows cohesive subgraphs which contains Prof. Michael Stonebraker obtained by *k-truss* and *hyper k-truss*, respectively. As shown in Fig.12(a), the *k-truss* ($k = 5$) subgraph includes a large number of collaborators, making it difficult to identify meaningful real-life communities. Additionally, multiple edges connect to Prof. Stonebraker in Fig.12(a), further complicating the extraction of well-defined dense subgraphs. In *hyper k-truss*



(a) Traditional k-truss of Prof. Stonebraker



(b) Hyper k-truss of Prof. Stonebraker

Figure 12: Case study of hyper k-truss on DBLP

($k = 5$), displaying all hyperedges would reduce visibility, so we provide a summary of frequent relationships in Fig. 12(b). Since each hyperedge represents a publication—for instance, E_1 corresponds to a SIGMOD paper co-authored by Samuel Madden and Prof. Stonebraker, while E_3 represents an ICDE paper co-authored by Matthew Perron and Prof. Stonebraker—we can infer Prof. Stonebraker's frequent participation in conferences such as SIGMOD, VLDB, and ICDE, allowing for a clearer interpretation of these relationships. The key difference between these two models lies in how they capture collaborations. The *k-truss* model merges all co-authorships into a single structure: while we know that Samuel Madden frequently collaborates with Prof. Stonebraker, the model does not distinguish between their work in *Communications of the ACM* and *SIGMOD*. In contrast, the *hyper k-truss* model preserves the influence of individual publications, ensuring that different collaborations contribute separately to the results. Thus, *hyper k-truss* offers a more structured representation of groupwise relationships, making it a superior approach to modeling cohesive subgraphs in hypergraphs compared to *k-truss*.

6 RELATED WORK

This work primarily relates to cohesive subgraph models, specifically k-truss decomposition, and triangle counting in hypergraphs. Below are the latest developments.

Cohesive Subgraph Models. Numerous cohesive subgraph models have been proposed based on different measures of cohesiveness [11, 12, 43]. Notable examples include k-core [41], k-truss [48], maximal k-edge connected subgraphs [59], and maximal cliques [54–56]. The k-core is a maximal subgraph in which the degree of each node is at least k [30]. Recently, this concept has been extended to uncertain graphs [7, 52], attributed graphs [28, 29], distance generalized cores [8, 14, 33], and more [4, 31, 32, 44, 45]. A k-truss is a maximal subgraph where each edge participates in at least $k-2$ triangles [13, 22, 23, 34, 46–48]. All the k-core, k-truss, and maximal k-edge connected subgraphs can be computed in polynomial time using a peeling-style algorithm. Additionally, several studies have explored core decomposition in hypergraphs [3, 10, 15, 25], dense subgraph mining in weighted hypergraphs [5], and maintaining core structures in dynamic hypergraphs [21]. Unlike these works, this paper focuses on truss decomposition algorithms in hypergraphs. The primary challenge of these algorithms lies in efficiently counting hyper-triangles, which will be discussed in the next section.

Triangle counting. Traditional triangle counting algorithm iterates through each edge of the graph and intersects the neighbor lists of both source and destination nodes. Once a common neighboring vertex is found, a triangle is enumerated. As such, the theoretical time and space complexity of optimal methods for triangle counting is $O(|E|^{1.5})$ and $O(|E|)$, where $|E|$ is the number of edges in the graph [18, 40]. In addition, Latapy [27] shows that the running time of triangle counting can be bounded by $\Theta(|E||V|^{\frac{1}{\alpha}})$ in which the degree distribution of the graph is governed by a power law with exponent α. Futhermore, Berry, et al. [6] show that the running time can be $\Theta(|V|)$ in realistic circumstances where the 4/3 moment of a graph is bounded by a constant. For a normal traditional graph, the most efficient algorithm to count the triangles relies on matrix multiplication and runs in $O(|V|^w)$ or $O(|E|^{w/(1+ws)})$ where $w < 2.376$ is the matrix multiplication exponent [2, 51, 53], but these algorithms require $\Theta(|V|^2)$ space to construct the matrix. Zhang, et al. [58] propose a sampling-based method for estimating triangle counts in hypergraph streams. In recent years, there are also many works focus on counting triangles in distributed architectures [1, 16, 19], on GPUs [20, 38, 39], and in graph streams [17, 24, 35, 42, 57] and so on. Although various studies have explored triangle counting, existing methods cannot enumerate triangles in hypergraphs for truss decomposition, and current optimization techniques are not directly applicable for efficient triangle counting in hypergraphs.

7 CONCLUSION

In this work, we address the challenge of truss decomposition in hypergraphs. We begin by presenting a framework for hyper truss decomposition, focusing on the key challenge of computing the initial support for all hyperedges. Our analysis reveals that this task is essentially equivalent to performing a global count of hyper-triangles. We then introduce a baseline algorithm for hyper-triangle counting, which provides a foundation for further enhancements. To improve this approach, we incorporate edge-iterator and node-iterator strategies, significantly reducing redundant computations. To further improve the efficiency, we propose a novel prefix forest structure to encode hyperedges and enumerate hyper-triangles within this forest. This method is further optimized through techniques such as node reordering by degree and work balancing, all contributing to enhanced performance. Extensive experimental evaluations on real-world hypergraph datasets confirm the effectiveness and efficiency of our algorithms.

ACKNOWLEDGMENTS

This work was partially supported by NSFC under Grants 62202053, U2241211, 62372342, and 62402399. Guoren Wang is the corresponding author.

REFERENCES

[1] Seher Acer, Abdurrahman Yasar, Sivasankaran Rajamanickam, Michael M. Wolf, and Ümit V. Çatalyürek. 2019. Scalable Triangle Counting on Distributed-Memory Systems. In *HPEC*. 1–5.

[2] Noga Alon, Raphael Yuster, and Uri Zwick. 1997. Finding and Counting Given Length Cycles. *Algorithmica* 17, 3 (1997), 209–223.

[3] Naheed Anjum Arafat, Arijit Khan, Arpit Kumar Rai, and Bishwamittra Ghosh. 2023. Neighborhood-based Hypergraph Core Decomposition. *Proc. VLDB Endow.* 16, 9 (2023), 2061–2074.

[4] Wen Bai, Yuncheng Jiang, Yong Tang, and Yayang Li. 2023. Parallel Core Maintenance of Dynamic Graphs. *IEEE Trans. Knowl. Data Eng.* 35, 9 (2023), 8919–8933.

[5] Oana Balalau, Francesco Bonchi, T.-H. Hubert Chan, Francesco Gullo, Mauro Sozio, and Hao Xie. 2024. Finding Subgraphs with Maximum Total Density and Limited Overlap in Weighted Hypergraphs. *ACM Trans. Knowl. Discov. Data* 18, 4 (2024), 95:1–95:21.

[6] Jonathan W. Berry, Luke A. Fostvedt, Daniel J. Nordman, Cynthia A. Phillips, C. Seshadhri, and Alyson G. Wilson. 2015. Why Do Simple Algorithms for Triangle Enumeration Work in the Real World? *Internet Math.* 11, 6 (2015), 555–571.

[7] Francesco Bonchi, Francesco Gullo, Andreas Kaltenbrunner, and Yana Volkovich. 2014. Core decomposition of uncertain graphs. In *KDD*. 1316–1325.

[8] Francesco Bonchi, Arijit Khan, and Lorenzo Severini. 2019. Distance-generalized Core Decomposition. In *SIGMOD*. 1006–1023.

[9] Alain Bretto. 2013. *Hypergraph Theory: An Introduction*. Springer.

[10] Fanchen Bu, Geon Lee, and Kijung Shin. 2023. Hypercore decomposition for non-fragile hyperedges: concepts, algorithms, observations, and applications. *Data Min. Knowl. Discov.* 37, 6 (2023), 2389–2437.

[11] Hongbo Cai, Hong Chen, and Shen Liu. 2017. A Method for Constructing Open-Domain Chinese Entity Hypernym Hierarchical Structure. *ZTE Communications* 15, 1 (2017), 49–54.

[12] Lijun Chang and Lu Qin. 2019. Cohesive Subgraph Computation Over Large Sparse Graphs. In *ICDE*. 2068–2071.

[13] Zi Chen, Long Yuan, Li Han, and Zhengping Qian. 2023. Higher-Order Truss Decomposition in Graphs. *IEEE Trans. Knowl. Data Eng.* 35, 4 (2023), 3966–3978.

[14] Qiangqiang Dai, Rong-Hua Li, Lu Qin, Guoren Wang, Weihua Yang, Zhiwei Zhang, and Ye Yuan. 2021. Scaling Up Distance-generalized Core Decomposition. In *CIKM*. 312–321.

[15] Kasimir Gabert, Ali Pinar, and Ümit V. Çatalyürek. 2021. A Unifying Framework to Identify Dense Subgraphs on Streams: Graph Nuclei to Hypergraph Cores. In *WSDM*. 689–697.

[16] Sayan Ghosh and Mahantesh Halappanavar. 2020. TriC: Distributed-memory Triangle Counting by Exploiting the Graph Structure. In *HPEC 2020*. 1–6.

[17] Xiangyang Gou and Lei Zou. 2021. Sliding Window-based Approximate Triangle Counting over Streaming Graphs with Duplicate Edges. In *SIGMOD '21*. 645–657.

[18] Mohammad Al Hasan and Vachik S. Dave. 2018. Triangle counting in large networks: a review. *Wiley Interdiscip. Rev. Data Min. Knowl. Discov.* 8, 2 (2018).

[19] Loc Hoang, Vishwesh Jatala, Xuhao Chen, Udit Agarwal, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2019. DistTC: High Performance Distributed Triangle Counting. In *HPEC*. 1–7.

[20] Lin Hu, Lei Zou, and Yu Liu. 2021. Accelerating Triangle Counting on GPU. In *SIGMOD '21*. 736–748.

[21] Qiang-Sheng Hua, Xiaohui Zhang, Hai Jin, and Hong Huang. 2023. Revisiting Core Maintenance for Dynamic Hypergraphs. *IEEE Trans. Parallel Distributed Syst.* 34, 3 (2023), 981–994.

[22] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-truss community in large and dynamic graphs. In *SIGMOD*. 1311–1322.

[23] Xin Huang, Laks V. S. Lakshmanan, Jeffrey Xu Yu, and Hong Cheng. 2015. Approximate Closest Community Search in Networks. *Proc. VLDB Endow.* 9, 4 (2015), 276–287.

[24] Rajesh Jayaram and John Kallaugher. 2021. An Optimal Algorithm for Triangle Counting in the Stream. In *APPROX/RANDOM 2021*, Vol. 207. 11:1–11:11.

[25] Dahee Kim, Junghoon Kim, Sungsu Lim, and Hyun Ji Jeong. 2023. Exploring Cohesive Subgraphs in Hypergraphs: The (k, g)-core Approach. In *CIKM*. 4013–4017.

[26] Alexandr V. Kostochka, Dhruv Mubayi, and Jacques Verstraëte. 2013. Hypergraph Ramsey numbers: Triangles versus cliques. *J. Comb. Theory, Ser. A* 120, 7 (2013), 1491–1507.

[27] Matthieu Latapy. 2008. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theor. Comput. Sci.* 407, 1-3 (2008), 458–473.

[28] Rong-Hua Li, Lu Qin, Fanghua Ye, Jeffrey Xu Yu, Xiaokui Xiao, Nong Xiao, and Zibin Zheng. 2018. Skyline Community Search in Multi-valued Networks. In *SIGMOD*. 457–472.

[29] Rong-Hua Li, Lu Qin, Jeffrey Xu Yu, and Rui Mao. 2015. Influential Community Search in Large Networks. *Proc. VLDB Endow.* 8, 5 (2015), 509–520.

[30] Rong-Hua Li, Jeffrey Yu, and Rui Mao. 2014. Efficient Core Maintenance in Large Dynamic Graphs. *IEEE Trans. Knowl. Data Eng.* 26, 10 (2014), 2453–2465.

[31] Xuankun Liao, Qing Liu, Jiaxin Jiang, Xin Huang, Jianliang Xu, and Byron Choi. 2022. Distributed D-core Decomposition over Large Directed Graphs. *Proc. VLDB Endow.* 15, 8 (2022), 1546–1558.

[32] Jingxin Liu, Chang Xu, Chang Yin, Weiqiang Wu, and You Song. 2022. K-Core Based Temporal Graph Convolutional Network for Dynamic Graphs. *IEEE Trans. Knowl. Data Eng.* 34, 8 (2022), 3841–3853.

[33] Qing Liu, Xuliang Zhu, Xin Huang, and Jianliang Xu. 2021. Local Algorithms for Distance-generalized Core Decomposition over Large Dynamic Graphs. *Proc. VLDB Endow.* 14, 9 (2021), 1531–1543.

[34] Qi Luo, Dongxiao Yu, Xiuzhen Cheng, Hao Sheng, and Weifeng Lyu. 2023. Exploring Truss Maintenance in Fully Dynamic Graphs: A Mixed Structure-Based Approach. *IEEE Trans. Computers* 72, 3 (2023), 707–718.

[35] Andrew McGregor and Sofya Vorotnikova. 2020. Triangle and Four Cycle Counting in the Data Stream Model. In *PODS 2020*. 445–456.

[36] James R Munkres. 2018. *Elements of algebraic topology*. CRC press.

[37] Jiaxi Nie, Sam Spiro, and Jacques Verstraëte. 2021. Triangle-Free Subgraphs of Hypergraphs. *Graphs and Combinatorics* (2021), 1–16.

[38] Zhiyou Ouyang, Shanni Wu, Tongtong Zhao, Dong Yue, and Tengfei Zhang. 2019. Memory-Efficient GPU-Based Exact and Parallel Triangle Counting in Large Graphs. In *HPCC/SmartCity/DSS 2019*. 2195–2199.

[39] Santosh Pandey, Zhibin Wang, Sheng Zhong, Chen Tian, Bolong Zheng, Xiaoye Li, Lingda Li, Adolfy Hoisie, Caiwen Ding, Dong Li, and Hang Liu. 2021. Trust: Triangle Counting Reloaded on GPUs. *IEEE Trans. Parallel Distributed Syst.* 32, 11 (2021), 2646–2660.

[40] Thomas Schank and Dorothea Wagner. 2005. Finding, Counting and Listing All Triangles in Large Graphs, an Experimental Study. In *Experimental and Efficient Algorithms*. 606–609.

[41] Stephen B. Seidman. 1983. Network structure and minimum degree. *Social Networks* 5, 3 (1983), 269–287.

[42] Kijung Shin, Euiwoong Lee, Jinoh Oh, Mohammad Hammoud, and Christos Faloutsos. 2021. CoCoS: Fast and Accurate Distributed Triangle Counting in Graph Streams. *ACM Trans. Knowl. Discov. Data* 15, 3 (2021), 38:1–38:30.

[43] Guojie Song, Ping Lu, Wei Wei, and Danmeng Liu. 2015. Community Discovery with Location-Interaction Disparity in Mobile Social Networks. *ZTE Communications* 13, 2 (2015), 53–61.

[44] Longxu Sun, Xin Huang, Rong-Hua Li, Byron Choi, and Jianliang Xu. 2022. Index-Based Intimate-Core Community Search in Large Weighted Graphs. *IEEE Trans. Knowl. Data Eng.* 34, 9 (2022), 4313–4327.

[45] Xin Sun, Xin Huang, and Di Jin. 2022. Fast Algorithms for Core Maximization on Large Graphs. *Proc. VLDB Endow.* 15, 7 (2022), 1350–1362.

[46] Zitan Sun, Xin Huang, Qing Liu, and Jianliang Xu. 2023. Efficient Star-based Truss Maintenance on Dynamic Graphs. *Proc. ACM Manag. Data* 1, 2 (2023), 133:1–133:26.

[47] Anxin Tian, Alexander Zhou, Yue Wang, and Lei Chen. 2023. Maximal D-truss Search in Dynamic Directed Graphs. *Proc. VLDB Endow.* 16, 9 (2023), 2199–2211.

[48] Jia Wang and James Cheng. 2012. Truss Decomposition in Massive Networks. *Proc. VLDB Endow.* 5, 9 (2012), 812–823.

[49] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2020. Efficient Bitruss Decomposition for Large-scale Bipartite Graphs. In *ICDE*. IEEE, 661–672.

[50] Xinzhou Wang, Yinjia Chen, Zhiwei Zhang, Pengpeng Qiao, and Guoren Wang. 2022. Efficient Truss Computation for Large Hypergraphs. In *WISE*. 290–305.

[51] Michael M. Wolf, Mehmet Deveci, Jonathan W. Berry, Simon D. Hammond, and Sivasankaran Rajamanickam. 2017. Fast linear algebra-based triangle counting with KokkosKernels. In *HPEC*. 1–7.

[52] Bohua Yang, Dong Wen, Lu Qin, Ying Zhang, Lijun Chang, and Rong-Hua Li. 2019. Index-Based Optimal Algorithm for Computing K-Cores in Large Uncertain Graphs. In *ICDE*. 64–75.

[53] Abdurrahman Yasar, Sivasankaran Rajamanickam, Jonathan W. Berry, and Ümit V. Çatalyürek. 2022. A Block-Based Triangle Counting Algorithm on Heterogeneous Environments. *IEEE Trans. Parallel Distributed Syst.* 33, 2 (2022), 444–458.

[54] Dongxiao Yu, Lifang Zhang, Qi Luo, Xiuzhen Cheng, and Zhipeng Cai. 2023. Maximal Clique Search in Weighted Graphs. *IEEE Trans. Knowl. Data Eng.* 35, 9 (2023), 9421–9432.

[55] Kaiqiang Yu and Cheng Long. 2023. Fast Maximal Quasi-clique Enumeration: A Pruning and Branching Co-Design Approach. *Proc. ACM Manag. Data* 1, 3 (2023), 211:1–211:26.

[56] Ting Yu, Ting Jiang, Mohamed Jaward Bah, Chen Zhao, Hao Huang, Mengchi Liu, Shuigeng Zhou, Zhao Li, and Ji Zhang. 2024. Incremental Maximal Clique Enumeration for Hybrid Edge Changes in Large Dynamic Graphs. *IEEE Trans. Knowl. Data Eng.* 36, 4 (2024), 1650–1666.

[57] Lingling Zhang, Hong Jiang, Fang Wang, Dan Feng, and Yanwen Xie. 2020. Reservoir-based sampling over large graph streams to estimate triangle counts and node degrees. *Future Gener. Comput. Syst.* 108 (2020), 244–255.

[58] Lingling Zhang, Zhiwei Zhang, Guoren Wang, Ye Yuan, and Zhao Kang. 2023. Efficiently Counting Triangles for Hypergraph Streams by Reservoir-Based Sampling. *IEEE Trans. Knowl. Data Eng.* 35, 11 (2023), 11328–11341.

[59] Rui Zhou, Chengfei Liu, Jeffrey Xu Yu, Weifa Liang, Baichen Chen, and Jianxin Li. 2012. Finding maximal k-edge-connected subgraphs from a large graph. In *EDBT*. 480–491.