

Efficient Higher Order Truss Decomposition on Graphs

Fengnian Lin

School of Computing and Information Systems
The University of Melbourne
Melbourne, VIC, Australia
fengnian.lin@student.unimelb.edu.au

Lijun Chang

The University of Sydney
Sydney, NSW, Australia
lijun.chang@sydney.edu.au

Junhao Gan*

School of Computing and Information Systems
The University of Melbourne
Melbourne, VIC, Australia
junhao.gan@unimelb.edu.au

Seeun William Umboh

School of Computing and Information Systems
The University of Melbourne
Melbourne, VIC, Australia
Integrated Methodologies, and Applications (OPTIMA)
ARC Training Centre in Optimisation Technologies
Melbourne, VIC, Australia
william.umbroh@unimelb.edu.au

Abstract

Truss decomposition is a fundamental cohesive subgraph mining technique. In this paper, we study high-order truss decomposition, which generalizes classic truss decomposition from triangles to longer cycles and aims to capture high-order cycle structures in graphs. In particular, we focus on 4-cycle and 5-cycle truss decompositions. Naive solutions have to enumerate every 4-cycle or 5-cycle via a standard *peeling process*, leading to worst-case time complexity of $O(m^2)$ or $O(m^{2.5})$, respectively, where m is the number of edges in the graph. We first formally prove that 4-cycle truss decomposition can be reduced to the bitruss decomposition on bipartite graphs, and hence can be immediately solved by the existing state-of-the-art bitruss decomposition algorithm with running time bounded by $O(\gamma \cdot m \cdot \log m)$, where γ is the degeneracy of the graph and is upper bounded by $O(\sqrt{m})$ in the worst case. However, this reduction introduces a blow-up factor of 2 to the number of edges, which is less desirable in terms of practical performance. We further propose a unified algorithm framework for computing 4-cycle and 5-cycle truss decompositions which does not need to enumerate every 4-cycle and 5-cycle and achieves a running time complexity of $O(\gamma \cdot m \cdot \log m)$ (without the reduction overhead) and $O(\gamma^2 \cdot m \cdot \log m)$, respectively. These results significantly improve the existing known bounds by roughly a factor of $O(\sqrt{m})$. Moreover, cycle truss numbers can be stored as edge properties in graph databases to support downstream applications. We conducted two case studies to demonstrate their effectiveness in recommendation and protein analysis. Extensive experimental results on ten public real-world datasets and multiple synthetic datasets show that our proposed algorithms outperform baselines in terms of efficiency by up to an order of magnitude. The source code is available at <https://github.com/HenryLimHongni/High-Truss>.

CCS Concepts

• Theory of computation → Graph algorithms analysis.

Keywords

Higher Order Truss Decomposition, 4-Cycles, 5-Cycles

1 Introduction

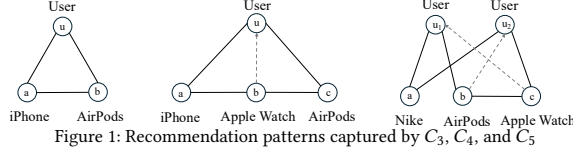
Cohesive subgraph mining has long been an important tool for understanding the structural properties of graphs [14, 16]. It has been widely applied to various domains, including community detection, recommendation, and social network analysis. To identify dense communities in large graphs, numerous cohesive subgraph models have been proposed, including cliques [30], k -cores [5, 21, 27], k -trusses [15, 31], k -plexes [4, 9], (k, p) -cores [36, 37], k -defective cliques [8, 24], and many others.

In this paper, we study the problem of *high-order Truss Decomposition* on an undirected graph G . Let C_p denote the cycle pattern of length p , i.e., a p -cycle. In particular, when $p = 3$, C_p is a 3-cycle, also known as a *triangle*; when $p = 4$, it is a 4-cycle; and so on. Given an integer $k \geq 1$, a k - C_p -truss is a subgraph $H \subseteq G$ in which every edge belongs to at least k p -cycles in H . For each edge e in G , its C_p -truss number is the largest value of k such that e belongs to a k - C_p -truss. The goal of C_p -Truss Decomposition is to compute the C_p -truss number of every edge in G for a given integer $p \geq 3$. When $p = 3$, the problem becomes the classical Truss Decomposition [15, 31], which has been extensively studied. In contrast, C_p -Truss Decomposition for higher-order cycles with $p \geq 4$ remains largely unexplored.

Cycle Patterns in Recommendation. Cycle patterns are of great usefulness in recommendation, a fundamental problem in database and data mining research. In a typical recommendation scenario, we are given a set of *user-item interactions*, such as purchase records and movie-watching histories. These interactions naturally form a bipartite *user-item graph*, with users on one side and items on the other. In practice, to capture richer information, the user-item graph is often augmented with item-item edges indicating that two items are similar¹.

¹User-user edges, however, are usually unavailable in practical applications. Unlike item similarity, which can often be measured objectively based on attributes such as categories, models, and functionalities, user similarity is considerably more difficult to quantify. Users with similar demographic attributes, such as age, salary, or occupation,

*Corresponding author.



In such an augmented user-item graph, different cycle patterns capture different recommendation semantics. Specifically, C_3 captures the pattern in which a user u purchases two similar items a and b (e.g., iPhone and AirPods in Figure 1). C_4 captures a richer pattern: user u has purchased items a and c (e.g., iPhone and AirPods in Figure 1), while an item b (e.g., Apple Watch) that u has not yet purchased is similar to both a and c (e.g., they are all Apple products). In this case, item b (e.g., Apple Watch) is a promising recommendation candidate for u . Thus, C_4 captures the semantics of a *recommendable candidate item* that C_3 cannot capture. However, C_4 still lacks the information about user similarity as only one user is involved in the pattern.

Interestingly, C_5 is the *smallest* cycle pattern that can additionally capture such user-similarity information. Consider a 5-cycle in which user u_1 purchased items a and b (e.g., Nike running shoes and AirPods in Figure 1), user u_2 purchased items a and c (e.g., Nike running shoes and Apple Watch), and items b and c are similar (e.g., both are Apple products). The common purchased item a (e.g., running shoes) suggests that u_1 and u_2 share certain similarities in their preferences (e.g., they both like running). Under this pattern, item c can be recommended to u_1 , while item b can be recommended to u_2 (e.g., indeed Apple Watch and AirPods are popular equipments for running). From the example in Figure 1, we can see that in this kind of C_5 patterns, the shared item does not need to be directly similar to the recommended items (e.g., Nike running shoes and Apple products), yet it captures certain intrinsic similarities in the users' preferences (e.g., on running). Therefore, these C_5 patterns can capture more information in recommendation which C_3 or C_4 patterns cannot capture.

For C_p with $p \geq 6$, these cycles can also capture both recommendable items and user similarity. However, the rapidly increasing number of p -cycles makes their computation prohibitively expensive on large graphs.

Two Case Studies. In Section 7, we conduct two case studies to examine the effectiveness of C_p -truss decomposition for $p \in \{3, 4, 5, 6\}$ in recommendation and protein analysis. For recommendation, we apply C_p -truss decomposition to two real-world datasets, MovieLens-100K and Foursquare-Tokyo, for movie and department store recommendation, respectively, and compare it with *itemKNN*, a classic *collaborative filtering* approach [28]. The experimental results show that C_5 -truss numbers consistently achieve the best recommendation quality in terms of NDCG@5 and Precision@5.

For protein analysis, we apply C_p -truss decomposition for $p \in \{3, 4, 5, 6\}$ to 16 protein graphs for *helix-core detection*. Again, C_5 -truss decomposition achieves the best results on most graphs in terms of ROC-AUC and PR-AUC.

Beyond these applications, cycle structures also arise in fraud detection [26] and anti-money laundering [33], where money may

may still have substantially different preferences due to factors such as cultural background and personal experience. Moreover, if reliable user-user similarity edges were available, recommendation could be performed directly based on user similarity.

pass through multiple accounts and eventually return to the starting account. These applications further suggest the potential of C_p -truss decomposition for analyzing complex relational structures.

C_p -Truss Decomposition Computation. Despite its potential usefulness, C_p -Truss Decomposition remains challenging and has not been sufficiently explored. In the existing literature, C_3 -Truss Decomposition on general graphs and C_4 -Truss Decomposition on bipartite graphs, also known as *Bitruss Decomposition*, have been extensively studied [23, 32, 39]. However, systematic studies of C_p -Truss Decomposition for $p \geq 4$ on general graphs are still lacking.

In this paper, we focus on C_4 - and C_5 -Truss Decomposition on general graphs for two main reasons. First, from a semantic perspective, C_5 is the smallest cycle pattern that captures both recommendable candidate items and user similarity in recommendation. Our case study further shows that C_5 -truss numbers often provide the best recommendation quality. Second, from a computational perspective, C_5 represents a sweet spot between richer structural information and computational efficiency.

Specifically, the number of p -cycles in a graph can be as large as $O(m^{p/2})$ in the worst case, where m is the number of edges in G . Consequently, for $p \geq 6$, the number of p -cycles is $\Omega(m^3)$, which is prohibitively large for large-scale graphs. As shown in Figure 2, the number of 6-cycles is already substantially larger than the numbers of shorter cycles on three real-world graphs and the running time of C_6 -truss decomposition is orders of magnitude higher than that of the C_5 one on the same datasets. This rapid growth in computational cost motivates us to focus on C_4 and C_5 , which offer a favorable balance between structural richness and computational feasibility.

C_4 -Truss Decomposition. For 4-cycles, the notion of trapezes was introduced in [10]. However, there is no systematic study of C_4 -Truss Decomposition on general graphs. In contrast, Bitruss Decomposition has been extensively studied on bipartite graphs [10, 19, 23, 32, 39], where each butterfly is a 4-cycle.

In this paper, we formally prove that C_4 -Truss Decomposition on general graphs can be reduced to Bitruss Decomposition on bipartite graphs. Based on this reduction and the state-of-the-art Bitruss Decomposition algorithm [23], we show that C_4 -Truss Decomposition can be solved in $O(\gamma m \log m)$ time, where γ denotes the degeneracy of the graph.

C_5 -Truss Decomposition. For 5-cycles, existing work has focused primarily on counting [17, 29] rather than decomposition. Beyond the straightforward application of a naive peeling algorithm, as discussed in Section 4, C_5 -Truss Decomposition has not been systematically studied. A naive peeling algorithm first computes the 5-cycle support of each edge, i.e., the number of 5-cycles containing the edge, and then repeatedly removes the edge with the smallest support while updating the supports of the remaining edges. This approach requires enumerating all 5-cycles in the graph. Consequently, it must either store all 5-cycles, leading to prohibitive space consumption, or generate them on the fly, incurring substantial computational overhead. Moreover, the number of 5-cycles can be as large as $O(m^{2.5})$ in the worst case [6]. Therefore, directly enumerating all 5-cycles can be prohibitively expensive.

In this paper, we develop an algorithm that beats this enumeration bound in the worst case.

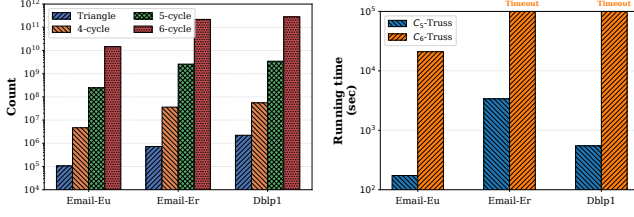


Figure 2: The left figure shows the numbers of C_3 , C_4 , C_5 and C_6 ; the right figure gives the running times of C_5 -Truss and C_6 -Truss Decompositions.

Contributions. We make the following contributions.

- We formally prove that C_4 -Truss Decomposition on general graphs can be reduced to Bitruss Decomposition on bipartite graphs with only a factor-of-two blowup in the problem size. Consequently, state-of-the-art Bitruss Decomposition algorithm [23] can be directly applied to compute C_4 -truss numbers with the same asymptotic time complexity of $O(\gamma m \log m)$.
- We develop a unified computational framework for C_p -Truss Decomposition for all $p \geq 3$. Based on this framework, we propose a new index called the $(2, 3)$ -Bloom Index for the case of $p = 5$. The $(2, 3)$ -Bloom Index orients edges according to a vertex priority (defined in Section 4) and groups priority-based 2-paths and 3-paths with the same key into two structures, called *2-blooms* and *3-blooms*. It supports efficient 5-cycle support computation and C_5 -Truss Decomposition using $O(\gamma^2 m)$ space. Based on the $(2, 3)$ -Bloom Index, we propose two algorithms for C_5 -Truss Decomposition. The first, *DET-Five*, is a peeling-based algorithm that updates edge supports using the proposed $(2, 3)$ -Bloom Index. Its running time still depends on the number of 5-cycles and can therefore be as high as $O(m^{2.5})$ in the worst case. To reduce this cost, we further develop *DDT-Five*, which employs the distributed tracking (DT) technique [11] to avoid explicit 5-cycle enumeration during the peeling process. This yields a time complexity of $O(\gamma^2 m \log m)$ and improves the worst-case bound to $O(m^2 \log m)$. Moreover, for planar graphs, although the number of 5-cycles can still be as large as $\Theta(m^2)$ [17], the degeneracy γ is bounded by a constant [22]. Therefore, *DDT-Five* runs in $O(m \log m)$ time on planar graphs, making it substantially more efficient than enumeration-based baselines.
- To reduce space consumption, we further propose *Save-DDT*, a practical space-efficient variant of *DDT-Five*. It reduces memory usage at the cost of a worst-case running time of $O(m^{2.5})$.
- Based on the same framework, we develop a new algorithm for C_4 -Truss Decomposition that runs nearly twice as fast as the straightforward reduction-based approach which first transforms the input graph into a bipartite graph and then applies Bitruss Decomposition.
- We conduct extensive experiments on 10 real-world datasets and various synthetic datasets. The experimental results show that *DET-Five*, *DDT-Five* and *Save-DDT* outperform all baselines by orders of magnitude in terms of efficiency. Moreover, we conduct two case studies on recommendation and protein analysis to demonstrate the utility of cycle truss numbers as edge properties in graph databases.

2 Preliminaries

2.1 Problem Formulation

Consider an undirected graph $G = (V, E)$, where V is the set of n vertices and E is the set of m edges. For any subgraph $H \subseteq G$, we use $V(H)$ and $E(H)$ to denote the induced vertex set and edge set of H , respectively.

p -Cycle. Consider an integer $p \in \{3, 4, 5\}$. A p -cycle, denoted by C_p , in G is formed by a sequence of distinct vertices $(v_1, v_2, \dots, v_p)$ such that $(v_i, v_{i+1}) \in E$ for $i \in \{1, \dots, p-1\}$ and $(v_p, v_1) \in E$.

C_p -Support. For a subgraph $H \subseteq G$ and an edge $e \in E(H)$, the C_p -Support of e in H , denoted by $s_p^H(e)$, is the number of C_p in H that contain e .

k - C_p -Truss. For an integer $k \geq 0$, a subgraph $H \subseteq G$ is called a k - C_p -truss if every edge in H has C_p -Support at least k , i.e., $s_p^H(e) \geq k$ for every $e \in E(H)$. A *maximal k - C_p -truss* is a k - C_p -truss that is maximal with respect to edge inclusion.

C_p -Truss Number. The C_p -truss number of an edge $e \in E$, denoted by $\tau_p(e)$, is the largest integer k such that e belongs to a maximal k - C_p -truss of G .

Definition 2.1 (Problem Definition). Given an undirected graph $G = (V, E)$, the goal of the C_p -Truss Decomposition problem is to compute $\tau_p(e)$ for all $e \in E$.

2.2 The Peeling Framework

As in many graph decomposition problems, C_p -Truss Decomposition can be solved under a peeling framework.

Let $H \subseteq G$ denote the current remaining subgraph during the peeling process, with $H = G$ initially. For each edge $e \in E(H)$, let $s_p^H(e)$ denote its current C_p -Support in H . The algorithm first computes the initial support $s_p^G(e)$ for every edge $e \in E(G)$, and set a current peeling level c_{cur} initialized to 0. The variable c_{cur} increases over time. At level c_{cur} , an edge $e \in E(H)$ is called *peelable* if $s_p^H(e) \leq c_{cur}$. The peeling algorithm repeatedly removes all peelable edges from G at the current level, and assigns each such edge e its C_p -truss number $\tau_p(e) = c_{cur}$. Peeling an edge may destroy some p -cycles in H , which decreases the supports of other remaining edges and may make them peelable at the current level. If there are no peelable edges remained in G , the current peeling level is increased by one, i.e., $c_{cur} \leftarrow c_{cur} + 1$. The process terminates when all edges have been peeled.

Algorithm 1 gives the generic peeling framework for C_p -Truss Decomposition. The key difference among concrete algorithms lies in how the initial supports are computed and how the supports are updated after peeling an edge.

2.3 Degeneracy, Core Number and Priority

Degeneracy. Consider a subgraph $H \subseteq G$; let $\deg_H(u)$ denote the degree of a vertex u in H , and $\delta(H) = \min_{u \in V(H)} \deg_H(u)$ be the minimum degree of H . The degeneracy of G is defined as $\gamma = \max_{H \subseteq G} \delta(H)$, that is, the maximum minimum degree among all possible subgraphs of G . Moreover, it is known that, in the worst case, $\gamma = O(\sqrt{m})$.

Algorithm 1 C_p -TRUSS DECOMPOSE(G)

Input: graph G ;
Output: truss numbers $\{\tau_p(e)\}_{e \in E}$;

- 1: compute the initial support $s_p^G(e)$ for each edge $e \in E$;
- 2: $H \leftarrow G$; $c_{cur} \leftarrow 0$;
- 3: **for all** $e \in E$ **do**
- 4: $s_p^H(e) \leftarrow s_p^G(e)$;
- 5: initialize auxiliary data structures restricted to H ;
- 6: **while** $E(H) \neq \emptyset$ **do**
- 7: $Q \leftarrow \{e \in E(H) : s_p^H(e) \leq c_{cur}\}$;
- 8: **while** Q is non-empty **do**
- 9: extract an edge e^* from Q ;
- 10: $\tau_p(e^*) \leftarrow c_{cur}$;
- 11: remove e^* from H and update the supports of affected edges;
- 12: **for all** affected edge $e \in E(H)$ by e^* **do**
- 13: **if** $s_p^H(e) \leq c_{cur}$ **then**
- 14: $Q \leftarrow Q \cup \{e\}$;
- 15: $c_{cur} \leftarrow c_{cur} + 1$;
- 16: **return** $\{\tau_p(e)\}_{e \in E}$;

k -core. Given an integer $k \geq 0$, a k -core of G is a maximal subgraph $H \subseteq G$ such that every vertex in H has degree at least k in H , namely $\deg_H(u) \geq k$ for all $u \in V(H)$.

Core Number. Given a graph G and a vertex $v \in V(G)$, the *core number* of v , denoted by $\kappa(v)$, is the largest integer k such that v belongs to a k -core of G .

FACT 1 ([34]). *The degeneracy γ of any undirected graph G is equal to the maximum core number $\kappa_{\max}$ of its vertices.*

Core Decomposition. It is known that the core number of each vertex can be computed by a peeling algorithm which repeatedly removes the vertex with degree in the current graph no more than the current peeling level and assigns this level as the core number of the vertex. If no vertex can be removed, then increment the peeling level by one. This process repeats until all the vertices are peeled.

Vertex Priority. In this paper, the priority of the vertices in G is defined as their peeling order in the above Core Decomposition process, where the first peeled vertex has priority 1 and the last has priority n . For any two vertices $a, b \in V$, we write $a \succ b$ if a has higher priority than b , namely, a is peeled after b .

Edge Orientation. According to the priority, each edge $(u, v) \in E$ is oriented from the lower-priority endpoint to the higher-priority endpoint. That is, $u \rightarrow v$ if $v \succ u$. For each vertex v , define $\text{Out}(v)$ to be the set of out-neighbors of v in the oriented graph, and $\text{Adj}(v)$ to be its neighbor set in the original graph.

FACT 2. $|\text{Out}(v)| \leq \kappa(v) \leq \kappa_{\max} = \gamma$ holds for all $v \in V$, where $\kappa_{\max}$ is the maximum core number and γ is the degeneracy of G .

2.4 Distributed Tracking (DT)

We give a brief overview of the distributed tracking (DT) problem and the Power-of-Two-Slack distributed tracking algorithm (P2S-DT) [11, 38], which is a crucial building block of our algorithm.

In the DT problem, there is one coordinator q and h participants $s_1, \dots, s_h$, where each participant can communicate with the coordinator q only. Each participant s_i maintains a local integer counter c_i , which is initially 0 and increases over time. At most one counter can be increased at a time and each increment is an integer at least 1. The coordinator is given a threshold $\tau > 0$ and needs to detect

the first moment when the counter sum reaches or exceeds the threshold τ , namely, $\sum_{i=1}^h c_i \geq \tau$. This moment is called the *maturity moment*. The goal is to minimize the number of messages exchanged between the coordinator and the participants.

A standard solution is the *Power-of-Two-Slack* distributed tracking algorithm (P2S-DT) [38], shown in Algorithm 2. For each participant s_i , we use l_i to denote the last value of c_i that has been reported to the coordinator. The main idea is to divide the process into rounds. In each round, when $\tau_{\text{cur}} > 2h$, the coordinator sets a slack value λ , which is chosen as the largest power of 2 no larger than $\tau_{\text{cur}}/(2h)$, where τ_{cur} is the remaining threshold of the current round, i.e., $\lambda = 2^{\lfloor \log_2(\frac{\tau_{\text{cur}}}{2h}) \rfloor}$. Each participant only sends a message when its local counter increases by at least the slack λ since its last report. We use Δ_q to denote the total increment received by the coordinator in the current round. Once $\Delta_q \geq \tau_{\text{cur}}/2$, the coordinator collects the exact current counter values from all participants, updates the remaining threshold by subtracting the total counter increment in this round, and starts a new round if the threshold has not yet been reached. When $\tau_{\text{cur}} \leq 2h$, the algorithm enters a base case: every counter increment is reported to q immediately, until the counter sum reaches or exceeds the threshold τ_{cur} . The algorithm ensures that the coordinator receives at most $O(h)$ messages in each round. Since the number of rounds is $O(\log(\tau/h))$. Therefore, its total communication complexity is $O(h \log \frac{\tau}{h})$.

Algorithm 2 Power-of-Two-Slack Distributed Tracking (P2S-DT)

- 1: **input:** threshold τ , participants $s_1, \dots, s_h$;
- 2: **output:** the maturity moment;
- 3: initialize $\tau_{\text{cur}} \leftarrow \tau$, $l_i \leftarrow 0$ for all participants s_i ;
- 4: **while** $\tau_{\text{cur}} > 0$ **do**
- 5: **if** $\tau_{\text{cur}} > 2h$ **then**
- 6: coordinator sets $\lambda \leftarrow 2^{\lfloor \log_2(\tau_{\text{cur}}/(2h)) \rfloor}$, $\Delta_q \leftarrow 0$;
- 7: send λ to all participants;
- 8: **for all** participants s_i **do**
- 9: whenever $c_i - l_i \geq \lambda$, update $\Delta_q \leftarrow \Delta_q + (c_i - l_i)$;
- 10: update $l_i \leftarrow c_i$;
- 11: **if** $\Delta_q \geq \tau_{\text{cur}}/2$ **then**
- 12: update $\tau_{\text{cur}} \leftarrow \tau_{\text{cur}} - (\Delta_q + \sum_{i=1}^h (c_i - l_i))$;
- 13: reset $l_i \leftarrow c_i$ for all participants;
- 14: **else**
- 15: every counter reports each increment to q until the counter sum reaches or exceeds the threshold τ_{cur} , and terminate;
- 16: report the maturity moment;

3 C_4 -Truss Decomposition

Next, we show that C_4 -Truss Decomposition on an undirected graph can be reduced to bitruss decomposition on a bipartite graph.

Given an undirected graph $G = (V, E)$, we construct a bipartite graph $\tilde{G} = (V_L, V_R, \tilde{E})$ as follows. Let $V_L = \{x_L \mid x \in V\}$ and $V_R = \{x_R \mid x \in V\}$. For each edge $(x, y) \in E$, we add two bipartite edges (x_L, y_R) and (y_L, x_R) into $\tilde{E}$. Thus, each edge $e = (x, y) \in E$ corresponds to exactly two edges (x_L, y_R) and (y_L, x_R) in $\tilde{G}$. For a bipartite subgraph $\tilde{H} \subseteq \tilde{G}$ and an edge $\tilde{e} \in E(\tilde{H})$, let $s_{\tilde{H}}^{\tilde{G}}(\tilde{e})$ denote the number of butterflies in $\tilde{H}$ containing $\tilde{e}$. A butterfly is a $(2, 2)$ -biclique with two vertices on each side and four edges between them. We have the following theorem whose proof is in Appendix A.

THEOREM 3.1. *Given an undirected graph G , let $\tilde{G}$ be the bipartite graph constructed by the above manner. For any edge $e = (x, y) \in E(G)$, let $\tilde{e}_1 = (x_L, y_R)$ and $\tilde{e}_2 = (y_L, x_R)$ be its two corresponding*

edges in $\tilde{G}$. The C_4 -Truss number of e in G , $\tau_4(e)$, is equal to the bitruss number of $\tilde{e}_1$ and $\tilde{e}_2$ in $\tilde{G}$.

By this reduction, C_4 -Truss Decomposition can be solved by first transforming G into $\tilde{G}$ and then applying existing bitruss decomposition algorithms on $\tilde{G}$. By the state-of-the-art bitruss decomposition algorithm [23], C_4 -Truss Decomposition problem can be solved in time bounded by $O(\gamma m \log m)$ with $O(\gamma m)$ space.

However, this reduction has two limitations. First, each edge in G is transformed into two edges in $\tilde{G}$, which introduces a factor of 2 in the graph size. Second, it requires an explicit transformation from G to the bipartite graph $\tilde{G}$, which also incurs additional running time. As we will see shortly, our algorithm framework for C_5 -Truss Decomposition can be simplified to solve C_4 -Truss Decomposition directly on the original graph G , removing this unnecessary reduction overhead. The details are discussed in Section 5.7.

4 C_5 -Truss Decomposition

4.1 Two Naive Peeling Algorithms

We first introduce two naive peeling-based algorithms, *Naive-Store* and *Naive-NoStore*, for C_5 -Truss Decomposition. Both algorithms follow the peeling framework in Section 2.2. They first enumerate all 5-cycles in G to compute the initial C_5 -Support of each edge. To achieve this, these algorithms perform a DFS traversal for each source vertex s : for each length-4 path $s \rightarrow a \rightarrow b \rightarrow c \rightarrow d$, if the closing edge (d, s) exists, then a 5-cycle is found.

The two algorithms differ in how they update the C_5 -Supports during the peeling process. In particular, *Naive-Store* stores all the 5-cycles and, for each edge e , maintains a list of all the 5-cycles containing e ; when e is peeled, all affected 5-cycles can be retrieved by the list. In contrast, *Naive-NoStore* does not store any 5-cycle. When an edge e is peeled, it computes all the 5-cycles that contain e in the current graph H on the fly by DFS. Therefore, *Naive-Store* consumes more space, whereas *Naive-NoStore* is slower in terms of running time. In the worst case, both algorithms need to enumerate all 5-cycles, whose number can be as large as $O(m^{2.5})$. Next, we show how to accelerate the peeling process with index and avoid 5-cycle enumeration.

4.2 (2, 3)-Bloom Index

All the missing proofs in this section can be found in Appendix B.

Priority 2-Paths. A *priority 2-path* is an ordered vertex triple (a, b, c) satisfies: (i) $(a, b), (b, c) \in E$, and (ii) $a \succ b$ and $a \succ c$. All priority 2-paths can be enumerated by scanning each middle vertex $b \in V$: for every $a \in \text{Out}(b)$ and every $c \in \text{Adj}(b) \setminus \{a\}$, if $a \succ c$, output (a, b, c) as a priority 2-path.

LEMMA 4.1. *The number of priority 2-paths is bounded by $O(\gamma m)$, and they can be enumerated in $O(\gamma m)$ time.*

Priority 3-Paths. A *priority 3-path* is an ordered vertex quadruple (a, b, c, d) satisfies: (i) $(a, b), (b, c), (c, d) \in E$ are distinct edges, and (ii) $a \succ b, a \succ c, a \succ d$, and $d \succ c$. The priority 3-paths can be generated by extending priority 2-paths as follows. For each priority 2-path (a, b, c) , and for each $d \in \text{Out}(c) \setminus \{a, b\}$, if $a \succ d$, output (a, b, c, d) as a priority 3-path.



Figure 3: A 5-cycle with vertex a having the largest priority 10 and d having higher priority 9 than c 's can be uniquely decomposed into a priority 2-path (a, x, d) and a priority 3-path (a, b, c, d) .

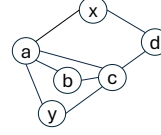


Figure 4: A 2-bloom and 3-bloom combination with key (a, d) .

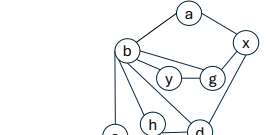


Figure 5: A graph containing 12 edges and six 5-cycles.

LEMMA 4.2. *The number of priority 3-paths is bounded by $O(\gamma^2 m)$, and they can be enumerated in $O(\gamma^2 m)$ time.*

LEMMA 4.3. *Every 5-cycle in G can be uniquely decomposed into one priority 2-path and one priority 3-path.*

PROOF. Consider any 5-cycle on vertex set $\{a, b, c, d, x\}$. Let a be the highest-priority vertex on this cycle. Among the two vertices at distance two from a , let d be the one with higher priority. Then the cycle is uniquely decomposed into the priority 3-path (a, b, c, d) and the priority 2-path (a, x, d) . See Figure 3 for an example. $\square$

Blooms. We group the priority paths by their endpoints. For a pair (a, d) with $a \succ d$, a *2-bloom* $B_{(a,d)}$ is the set of all priority 2-paths of the form (a, x, d) , and the *3-bloom* $S_{(a,d)}$ is the set of all priority 3-paths of the form (a, b, c, d) . We call (a, d) the *key* of these blooms. We use $|B_{(a,d)}|$ and $|S_{(a,d)}|$ to denote the numbers of priority 2-paths in $B_{(a,d)}$ and priority 3-paths in $S_{(a,d)}$, respectively. We also use $E(B_{(a,d)})$ and $E(S_{(a,d)})$ to denote the sets of edges in $B_{(a,d)}$ and $S_{(a,d)}$, respectively. For a 2-bloom $B_{(a,d)}$, $\text{twin}_{(a,d)}(e)$ denotes the other edge in the unique priority 2-path of $B_{(a,d)}$ that contains e . Each priority 2-path and each priority 3-path has a unique key, so each of them belongs to one and exactly one bloom of its type.

LEMMA 4.4. *Consider a fixed key (a, d) , a priority 2-path $(a, x, d) \in B_{(a,d)}$ and a priority 3-path $(a, b, c, d) \in S_{(a,d)}$ can form a 5-cycle if and only if $x \notin \{b, c\}$.*

The collection of all 2-blooms and 3-blooms is called the (2, 3)-Bloom Index. An edge may belong to multiple 2-blooms and multiple 3-blooms. Therefore, for each edge e , we maintain the sets $\mathcal{B}_G(e)$ and $\mathcal{S}_G(e)$, which denote the sets of keys of all 2-blooms and 3-blooms in G that contain e , respectively. Moreover, an edge may also appear in multiple priority 3-paths within the same 3-bloom. Hence for each edge e in a 3-bloom $S_{(a,d)}$, we use $S_{(a,d)}(e)$ to denote the set of priority 3-paths in $S_{(a,d)}$ that contain e , and use $|S_{(a,d)}(e)|$ to denote the number of such priority 3-paths.

Construction. We first enumerate all priority 2-paths and group them by their keys to build all 2-blooms. At the same time, for each edge, we record which 2-blooms contain it. Then we extend these priority 2-paths to enumerate priority 3-paths. Whenever a priority 3-path (a, b, c, d) is generated, we insert it into the 3-bloom $S_{(a,d)}$, update the keys stored for edges (a, b) , (b, c) , and (c, d) , and increase $|S_{(a,d)}(e_{ab})|$, $|S_{(a,d)}(e_{bc})|$ and $|S_{(a,d)}(e_{cd})|$ by one.

Time Complexity. The (2, 3)-Bloom Index can be constructed in $O(\gamma^2 m)$ time, since there are $O(\gamma m)$ priority 2-paths and $O(\gamma^2 m)$ priority 3-paths, and each path requires constant processing time.

Space Complexity. The (2, 3)-Bloom Index uses $O(\gamma^2 m)$ space, because it stores all priority 2-paths and priority 3-paths and the relative information.

Algorithm 3 *Count-Invalid*((a, x, d), (a, b, c, d))

Input: an invalid path-pair of a priority 2-path (a, x, d) and a priority 3-path (a, b, c, d) with $x \in \{b, c\}$

Output: a set Σ of edge-value pairs

```

1: if  $x = b$  then
2:    $\Sigma \leftarrow \{((a, b), 2), ((b, c), 1), ((c, d), 1), ((b, d), 1)\}$ 
3: else
4:    $\Sigma \leftarrow \{((c, d), 2), ((a, b), 1), ((b, c), 1), ((a, c), 1)\}$ 
5: return  $\Sigma$ 

```

4.3 C_5 -Support Computation

With the (2, 3)-Bloom Index, the C_5 -support of each edge e can be computed as follows. First, compute an upper bound, denoted by $\widehat{s}_5(e)$, on e 's C_5 -support ignoring *invalid* 5-cycles. Specifically, for each 2-bloom $B_{(a,d)}$ containing e , there is a unique priority 2-path in $B_{(a,d)}$ containing e ; and this 2-path can pair up with every priority 3-path in the 3-bloom $S_{(a,d)}$, contributing $|S_{(a,d)}|$ to $\widehat{s}_5(e)$. Moreover, for each 3-bloom $S_{(a,d)}$ containing e , each priority 3-path in $S_{(a,d)}$ can pair up with every priority 2-path in $B_{(a,d)}$, contributing $|S_{(a,d)}(e)| \cdot |B_{(a,d)}|$. Therefore, we compute:

$$\widehat{s}_5(e) = \sum_{(a,d) \in \mathcal{B}_G(e)} |S_{(a,d)}| + \sum_{(a,d) \in \mathcal{S}_G(e)} |S_{(a,d)}(e)| \cdot |B_{(a,d)}|. \quad (1)$$

However, some invalid 5-cycles may be counted in $\widehat{s}_5(e)$. To see this, for a fixed key (a, d), consider a *path-pair* of a priority 2-path (a, x, d) $\in B_{(a,d)}$ and a priority 3-path (a, b, c, d) $\in S_{(a,d)}$. By Lemma 4.4, this path-pair forms a valid 5-cycle if and only if $x \notin \{b, c\}$. A path-pair is *valid* if it forms a *valid* 5-cycle; and *invalid*, otherwise.

Consider a 2-bloom and 3-bloom combination ($B_{(a,d)}$ and $S_{(a,d)}$). If $E(B_{(a,d)}) \cap E(S_{(a,d)}) = \emptyset$, every path-pair between $B_{(a,d)}$ and $S_{(a,d)}$ is valid. Otherwise, each edge $e \in E(B_{(a,d)}) \cap E(S_{(a,d)})$ appears in both the two paths in some path-pairs forming invalid 5-cycles.

To correct the *overcount* due to invalid path-pairs, for each edge e , we maintain a *correction value* $\xi(e)$ which records the number of *occurrences* of e in invalid 5-cycles. Initially, $\xi(e) = 0$ for all edges $e \in E$. Then all $\xi(e)$'s can be computed as follows.

For each invalid path-pair of a 2-path (a, x, d) $\in B_{(a,d)}$ and a priority 3-path (a, b, c, d) $\in S_{(a,d)}$ with $x \in \{b, c\}$, we invoke the operation *Count-Invalid* to compute, for each edge e in these two paths, the occurrence of e in the formed invalid 5-cycle. Specifically, the occurrence count of the unique edge shared by the two paths is 2: one is counted from the 2-path and one counted from the 3-path; and for each other edge in the paths, its occurrence is 1. The pseudocode of *Count-Invalid* is shown in Algorithm 3. The output is a set of *edge-value* pairs, denoted by $\Sigma = \{(e, \delta)\}$. For each (e, δ) $\in \Sigma$, we update $\xi(e) \leftarrow \xi(e) + \delta$.

After processing all invalid path-pairs, the correction value $\xi(e)$ for each edge e is computed. And the C_5 -support of edge e in G is thus computed as: $s_5^G(e) = \widehat{s}_5(e) - \xi(e)$.

Example. Consider Figure 4; $B_{(a,d)}$ contains two priority 2-paths, (a, x, d) and (a, c, d), while $S_{(a,d)}$ contains two priority 3-paths,

(a, b, c, d) and (a, y, c, d). Thus, $|B_{(a,d)}| = 2$ and $|S_{(a,d)}| = 2$. Moreover, edge e_{cd} appears in the priority 2-path (a, c, d) and also appears in both priority 3-paths in $S_{(a,d)}$. Hence, $|S_{(a,d)}(e_{cd})| = 2$.

Suppose that $\mathcal{B}_G(e_{cd}) = \{(a, d)\}$ and $\mathcal{S}_G(e_{cd}) = \{(a, d)\}$. Then, by Eq. (1), the raw support of e_{cd} is $\widehat{s}_5(e_{cd}) = |S_{(a,d)}| + |S_{(a,d)}(e_{cd})| \cdot |B_{(a,d)}| = 2 + 2 \cdot 2 = 6$.

The two invalid pairs are ((a, c, d), (a, b, c, d)) and ((a, c, d), (a, y, c, d)). For each of them, invoking *Count-Invalid* assigns value 2 to edge (c, d). Therefore, the total correction on e_{cd} is $\xi(e_{cd}) = 4$. Hence, the corrected support $s_5^G(e_{cd}) = \widehat{s}_5(e_{cd}) - \xi(e_{cd}) = 6 - 4 = 2$.

Time complexity. Computing $\widehat{s}_5(e)$ only scans the blooms containing each edge. Moreover, handling invalid pairs also takes linear time in the total number of stored priority 3-paths: for each 2-bloom and 3-bloom combination $B_{(a,d)}$ and $S_{(a,d)}$, we enumerate the priority 3-paths in $S_{(a,d)}$, and for each such path there exists at most two invalid priority 2-paths in $B_{(a,d)}$. Thus, the total running time is linear in the total numbers of stored priority 2-paths and priority 3-paths, which gives an overall complexity of $O(\gamma^2 m)$.

Space complexity. The support computation only needs one additional array $\widehat{s}_5(\cdot)$ of size m . Hence, its extra space consumption beyond the (2, 3)-Bloom Index is bounded by $O(m)$.

4.4 Our DET-Five Algorithm

In this section, we introduce the *DET-Five* algorithm, which follows the peeling framework described in Section 2.2. The initial C_5 -Support of each edge has already been discussed above. Thus, the remaining task is to efficiently update the supports of affected edges after each edge removal during peeling. We must remove only the contribution of valid pairs and deal with the invalid 5-cycle carefully. The peeling procedure has two parts: processing the 2-blooms side, and processing the 3-blooms side. The detailed pseudocode of the peeling process is provided in Appendix C.

Peeling an edge from 2-blooms. We first process the 2-bloom side for the current peeled edge e^* . For each key (a, d) $\in \mathcal{B}_H(e^*)$, every edge in $E(B_{(a,d)})$ belongs to exactly one alive priority 2-path in the current 2-bloom $B_{(a,d)}$. Hence, there is a unique alive priority 2-path $p_2^* = (a, x, d) \in B_{(a,d)}$ such that $e^* \in E(p_2^*)$. Let $f^* = \text{twin}_{(a,d)}(e^*)$ be the other edge on p_2^* .

(i) **Remove the 2-path and update the bloom state.** We delete p_2^* from $B_{(a,d)}$ and decrease $|B_{(a,d)}|$ by 1.

(ii) **Bulk support decrement on the matched 3-bloom.** Ignoring invalid pairs for the moment, deleting one 2-path from $B_{(a,d)}$ removes exactly one pairing with each alive priority 3-path in $S_{(a,d)}$. Hence, for every edge $g \in E(S_{(a,d)})$, its support contribution under key (a, d) decreases by $|S_{(a,d)}(g)|$, and we update

$$s_5^H(g) \leftarrow s_5^H(g) - |S_{(a,d)}(g)|, \quad \forall g \in E(S_{(a,d)}). \quad (2)$$

(iii) **Support decrement on the surviving twin edge.** After deleting p_2^* , the surviving twin edge f^* no longer belongs to $B_{(a,d)}$, so it loses all pairs with the alive priority 3-paths in $S_{(a,d)}$. Ignoring invalid pairs, we decrease its support by $|S_{(a,d)}|$, namely, $s_5^H(f^*) \leftarrow s_5^H(f^*) - |S_{(a,d)}|$.

(iv) **Correction for invalid pairs.** The bulk updates need to be corrected because some pairs between the removed 2-path $p_2^* =$

Table 1: The 2-blooms and 3-blooms of the example in Figure 5

	(b, x)	(d, g)	(d, b)
2-bloom	$(b, a, x), (b, g, x)$	$(d, b, g), (d, x, g)$	$(d, c, b), (d, h, b)$
3-bloom	(b, y, g, x)	(d, b, y, g)	$(d, x, a, b), (d, x, g, b)$

(a, x, d) and alive priority 3-paths in $S_{(a,d)}$ do not form valid 5-cycles. We correct these over-decrements by directly invoking the operation *Count-Invalid* defined in Algorithm 3.

For each edge $h \in E(p_2^*) \cap E(S_{(a,d)})$, we examine all alive priority 3-paths in $S_{(a,d)}(h)$. For each such path p_3 , the pair (p_2^*, p_3) is invalid. We compute $\Sigma = \text{Count-Invalid}(p_2^*, p_3)$. Then, for every edge-value pair $(g, \delta) \in \Sigma$ with $g \neq e^*$, we compensate the previous support decrements by updating $s_5^H(g) \leftarrow s_5^H(g) + \delta$. Here we ignore e^* because it has already been peeled.

By processing all such invalid pairs under key (a, d) , all invalid pairs are corrected exactly.

Peeling an edge from 3-blooms. We next process the 3-bloom side for the current peeled edge e^* .

(i) Remove the affected 3-paths and update the bloom state. We delete all alive priority 3-paths in $S_{(a,d)}(e^*)$ from $S_{(a,d)}$, so $|S_{(a,d)}|$ decreases by $|S_{(a,d)}(e^*)|$.

(ii) Bulk support decrement on the 3-path side. For each removed path $p_3 = (a, b, c, d) \in S_{(a,d)}(e^*)$, let $E(p_3) = \{e^*, g_1, g_2\}$, where g_1 and g_2 are the other two edges on the path. Deleting p_3 removes one alive priority 3-path in $S_{(a,d)}$ that contains each of g_1 and g_2 . Hence, the values $|S_{(a,d)}(g_1)|$ and $|S_{(a,d)}(g_2)|$ each decrease by 1. Since each such occurrence contributes $|B_{(a,d)}|$ pairs under key (a, d) , we first decrease the supports of g_1 and g_2 by $|B_{(a,d)}|$. After the decrement, if $|S_{(a,d)}(g)| = 0$ for some edge $g \in \{g_1, g_2\}$, then g no longer appears in the current 3-bloom $S_{(a,d)}$.

(iii) Correction for invalid pairs. The decrements in steps (i) and (ii) need to be corrected because some pairs between a removed priority 3-path in $S_{(a,d)}(e^*)$ and alive priority 2-paths in $B_{(a,d)}$ do not form valid 5-cycles. We correct this by directly invoking the operation *Count-Invalid* defined in Algorithm 3.

For each removed path $p_3 \in S_{(a,d)}(e^*)$, and for each edge $g \in E(p_3) \setminus \{e^*\}$ such that $g \in E(B_{(a,d)})$, let $p_2(g) \in B_{(a,d)}$ be the unique alive priority 2-path containing g . Then $(p_2(g), p_3)$ is an invalid pair. We compute $\Sigma = \text{Count-Invalid}(p_2(g), p_3)$. Then, for every edge-value pair $(h, \delta) \in \Sigma$ with $h \neq e^*$, we compensate the previous support decrements by updating $s_5^H(h) \leftarrow s_5^H(h) + \delta$. Here we ignore e^* because it has already been peeled.

(iv) Propagate the decrease of $|S_{(a,d)}|$ to the 2-bloom side. After deleting the c^* paths in $S_{(a,d)}(e^*)$, every edge in $E(B_{(a,d)})$ loses c^* pairs in its first-term contribution in Equation (1) under key (a, d) . Therefore, for every edge $h \in E(B_{(a,d)})$, we decrease its support by c^* . Together with the compensation in step (iii), this ensures that only valid 5-cycles are removed from the supports.

Example. For the example in Figure 5, there are six 5-cycles: (a, b, c, d, x, a) , (a, b, h, d, x, a) , (a, b, y, g, x, a) , (b, d, x, g, y, b) , (b, h, d, x, g, b) , and (b, c, d, x, g, b) . The initial C_5 -supports are $s_5(e_{ab}) = 3$, $s_5(e_{bc}) = 2$, $s_5(e_{cd}) = 2$, $s_5(e_{dx}) = 5$, $s_5(e_{xa}) = 3$, $s_5(e_{by}) = 2$, $s_5(e_{yg}) = 2$, $s_5(e_{gx}) = 4$, $s_5(e_{bg}) = 2$, $s_5(e_{bh}) = 2$, $s_5(e_{hd}) = 2$, and $s_5(e_{bd}) = 1$. Table 1 shows the constructed 2-blooms and 3-blooms.

When $c_{cur} = 1$, edge e_{bd} is peeled first. Since $\mathcal{B}_H(e_{bd}) = S_H(e_{bd}) = \{(d, g)\}$, only the 2-bloom and 3-bloom combination with key (d, g) is affected.

We first process the 2-bloom side. Edge e_{bd} lies in the unique priority 2-path $(d, b, g) \in B_{(d,g)}$, whose twin edge is e_{bg} . After deleting (d, b, g) , the bulk decrement on $S_{(d,g)}$ decreases the supports of e_{by} and e_{yg} by 1, since $S_{(d,g)}$ contains the unique alive priority 3-path (d, b, y, g) . The twin-edge update also decreases e_{bg} by 1. However, the pair $((d, b, g), (d, b, y, g))$ is invalid. Invoking *Count-Invalid* adds back 1 to e_{bg} , e_{by} and e_{yg} , while the value on e_{bd} is ignored because it has already been peeled.

We then process the 3-bloom side. Since $S_{(d,g)}(e_{bd}) = \{(d, b, y, g)\}$, we delete the unique alive priority 3-path (d, b, y, g) from $S_{(d,g)}$. At this moment, the only alive priority 2-path in $B_{(d,g)}$ is (d, x, g) . Thus, step (ii) decreases the supports of e_{by} and e_{yg} by $|B_{(d,g)}| = 1$, and step (iv) propagates the decrease of $|S_{(d,g)}(e_{bd})|$ to the 2-bloom side, decreasing the supports of e_{dx} and e_{gx} by 1. No invalid correction is needed in this step.

Therefore, after peeling e_{bd} , we have $s_5(e_{dx}) = 4$, $s_5(e_{gx}) = 3$, and $s_5(e_{by}) = s_5(e_{yg}) = 1$. Hence, e_{by} and e_{yg} immediately become peelable at level $c_{cur} = 1$, so their C_5 -truss numbers are also 1. This process continues until all edges are removed.

4.5 Complexity Analysis of DET-Five

When an edge is peeled, we need to process all 2-blooms and 3-blooms containing this edge, and update the supports of the edges in the matched 2-blooms and 3-blooms with the same keys. This is equivalent to enumerating all 5-cycles containing the peeled edge. Therefore, over the whole decomposition process, the total work is bounded by the total cost of enumerating all 5-cycles in G . Since the number of 5-cycles can be as large as $O(m^{2.5})$ in the worst case, the total running time of DET-Five is also bounded by $O(m^{2.5})$.

5 Our Improved Algorithm

The algorithm above still requires to enumerate every 5-cycle. In this section, we propose the *DDT-Five* algorithm which adopts distributed tracking techniques to avoid touching every 5-cycle and the need of maintaining exact C_5 -Support for every edge at any time during the running of the algorithm, therefore reducing the overall running time bound.

5.1 DT Instance for Edges

For each edge e , we construct a distributed tracking instance $DT(e)$ as follows. The participants of $DT(e)$ consist of all 2-blooms containing e , all 3-blooms containing e , and one global participant W shared by all edges. For each participant corresponding to either a 2-bloom or 3-bloom containing e , its counter records the number of valid 5-cycles containing e that are *destroyed* and associated with this 2-bloom or 3-bloom. The counter of the global participant records the current peeling level c_{cur} in the peeling process. The threshold of $DT(e)$ is set $T(e) = s_5^G(e)$. We claim that e needs to be peeled if and only if $DT(e)$ reports maturity, i.e., the total counter sum $\geq T(e)$. The correctness of this claim follows from this theorem:

THEOREM 5.1. *For the current remaining graph H , edge e is peelable if and only if $\Delta_H^B(e) + \Delta_H^S(e) + c_{cur} \geq s_5^G(e)$, where $\Delta_H^B(e)$ and $\Delta_H^S(e)$ denote the total valid support decrements of e reported by all 2-bloom and 3-bloom participants of $DT(e)$, respectively.*

PROOF. By the peeling process, edge e is peelable exactly the first time when $s_5^H(e) \leq c_{cur}$. Meanwhile, the current support of e is always equal to its initial support $s_5^G(e)$ minus the number

of valid 5-cycles containing e that have been destroyed. If we use $\Delta_H(e)$ to denote the total decrement in the support of e caused by all previous peeling operations, then $s_5^H(e) = s_5^G(e) - \Delta_H(e)$. Therefore, the condition $s_5^H(e) \leq c_{cur}$ is equivalent to $\Delta_H(e) + c_{cur} \geq s_5^G(e)$. By creating the DT instance, every destroyed valid 5-cycle containing e is captured by one participant of $DT(e)$ corresponding to either a 2-bloom or a 3-bloom. Thus, $\Delta_H(e) = \Delta_H^B(e) + \Delta_H^S(e)$, and the peeling condition becomes $\Delta_H^B(e) + \Delta_H^S(e) + c_{cur} \geq s_5^G(e)$. $\square$

With the DT technique, there is no need to maintain the *precise* C_5 -support for every edge at any time during the peeling process.

5.2 Shared Counters

A direct way to maintain the dynamic C_5 -support is to keep a separate counter for each participant for every edge. However, this cannot avoid enumerating every 5-cycle. To overcome this bottleneck, for each key (a, d) we maintain two shared counters. The counter $c_{(a,d)}^B$ records the number of removed priority 3-paths in the 3-bloom $S_{(a,d)}$, while $c_{(a,d)}^S$ records the number of removed priority 2-paths in the 2-bloom $B_{(a,d)}$.

For any edge $e \in E(B_{(a,d)})$, each removed priority 3-path in $S_{(a,d)}$ destroys one pairing with the unique priority 2-path in $B_{(a,d)}$ that contains e . Therefore, for edges in $E(B_{(a,d)})$, the increment of $c_{(a,d)}^B$ is exactly the number of destroyed pairs contributed from the 3-bloom $S_{(a,d)}$ side, before invalid 5-cycle correction. Similarly, for any edge $e \in E(S_{(a,d)})$, each removed priority 2-path in $B_{(a,d)}$ destroys $|S_{(a,d)}(e)|$ pairs containing e before invalid 5-cycle correction.

To support reporting the increment, we store the last reported counter of the corresponding shared counter, denoted by $\ell_{(a,d)}^B(e)$ for $e \in E(B_{(a,d)})$ and $\ell_{(a,d)}^S(e)$ for $e \in E(S_{(a,d)})$. We also maintain $\ell_{cur}(e)$ as the last reported value of the global level counter c_{cur} .

Since DT does not immediately materialize the change of C_5 -support, when an edge is peeled, we record the temporal corrections caused by all related invalid 5-cycles, denoted by $\delta_{(a,d)}^B(e)$ for $e \in E(B_{(a,d)})$ and $\delta_{(a,d)}^S(e)$ for $e \in E(S_{(a,d)})$. These values are then used when a participant reports its accumulated increment to the DT coordinator to ensure the correct number of destroyed valid 5-cycles for correctness.

5.3 (2,3)-Bloom Insertion and Removal Rules

We maintain one power-of-two bucket array for each participant, R_{cur} for the global participant, $R_{(a,d)}^B$ for each 2-bloom $B_{(a,d)}$, and $R_{(a,d)}^S$ for each 3-bloom $S_{(a,d)}$. Each array stores the edges assigned to this participant together with their last reported counter values.

The bucket of an edge is determined by its slack value. Let $h(e)$ be the number of participants currently contained in $DT(e)$, and $\lambda(e) = 2^{\lfloor \log_2 \left(\frac{T(e)}{2h(e)} \right) \rfloor}$. For an edge stored in the global bucket array or in a 2-bloom bucket array, the slack is $\lambda(e)$. For an edge stored in the 3-bloom bucket array $R_{(a,d)}^S[\cdot]$, we use the normalized slack $\tilde{\lambda}_{(a,d)}(e) = 2^{\lfloor \log_2 \left(\max \left\{ 1, \frac{T(e)}{2|S_{(a,d)}(e)|h(e)} \right\} \right) \rfloor}$. Accordingly, edge e is placed into buckets $R_{cur}[\log_2 \lambda(e)]$ for the global participant, $R_{(a,d)}^B[\log_2 \lambda(e)]$ for each key $(a, d) \in \mathcal{B}(e)$, and $R_{(a,d)}^S[\log_2 \tilde{\lambda}_{(a,d)}(e)]$ for each key $(a, d) \in \mathcal{S}(e)$.

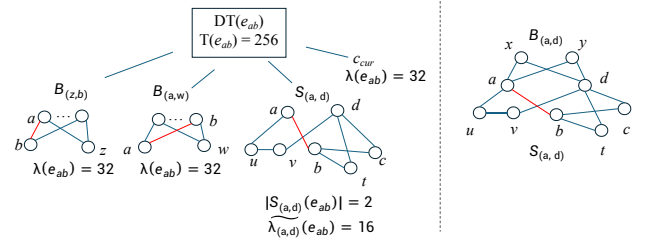


Figure 6: The left side shows the example of $DT(e_{ab})$. The right side shows the pairing of $S_{(a,d)}$ and $B_{(a,d)}$.

When a shared counter increases, we scan only the buckets up to the highest bit position that changes. More precisely, if a counter increases from c_{old} to c_{new} , we compute $j = \lfloor \log_2(c_{old} \oplus c_{new}) \rfloor$ and scan the buckets at indexes $0, 1, \dots, j$ in the bucket array.

While scanning an edge e , it reports the newly accumulation to the coordinator of $DT(e)$. We maintain an accumulator $A(e)$ for each edge e to store the total counter increment reported so far.

For an edge $e \in E(B_{(a,d)})$ in $R_{(a,d)}^B[\cdot]$, the reported increment is

$$A(e) \leftarrow A(e) + (c_{(a,d)}^B - \ell_{(a,d)}^B(e)) - \delta_{(a,d)}^B(e). \quad (3)$$

Here, $c_{(a,d)}^B - \ell_{(a,d)}^B(e)$ is the counter increment accumulated since the last report, and $\delta_{(a,d)}^B(e)$ is the invalid 5-cycle correction. Then we update $\ell_{(a,d)}^B(e)$ to $c_{(a,d)}^B$ and reset $\delta_{(a,d)}^B(e)$ to 0.

For an edge $e \in E(S_{(a,d)})$ in $R_{(a,d)}^S[\cdot]$, the reported increment is

$$A(e) \leftarrow A(e) + |S_{(a,d)}(e)| \cdot (c_{(a,d)}^S - \ell_{(a,d)}^S(e)) - \delta_{(a,d)}^S(e). \quad (4)$$

The factor $|S_{(a,d)}(e)|$ appears because each removed priority 2-path in $B_{(a,d)}$ destroys $|S_{(a,d)}(e)|$ pairs containing e . After reporting, we update $\ell_{(a,d)}^S(e)$ to $c_{(a,d)}^S$ and reset $\delta_{(a,d)}^S(e)$ to 0.

For the global participant, no invalid 5-cycle correction is needed, so the reported increment is

$$A(e) \leftarrow A(e) + (c_{cur} - \ell_{cur}(e)). \quad (5)$$

After that, we also update $\ell_{cur}(e)$ to c_{cur} .

When $A(e) \geq T(e)/2$, we collect the unreported increment of e from all its participants:

$$\begin{aligned} & A(e) + (c_{cur} - \ell_{cur}(e)) \\ & + \sum_{(a,d) \in \mathcal{B}(e)} \left((c_{(a,d)}^B - \ell_{(a,d)}^B(e)) - \delta_{(a,d)}^B(e) \right) \\ & + \sum_{(a,d) \in \mathcal{S}(e)} \left(|S_{(a,d)}(e)| \cdot (c_{(a,d)}^S - \ell_{(a,d)}^S(e)) - \delta_{(a,d)}^S(e) \right). \end{aligned} \quad (6)$$

After that, we update all stored last reported counter values of e to the current counters and reset all stored invalid-5-cycle corrections of e to 0.

If the total increment reaches $T(e)$, then e is inserted into the peel queue Q . Otherwise, we start a new DT round: subtract the total increment from $T(e)$, reset $A(e)$ to zero, recompute its slack values, and reinsert e into all corresponding buckets.

Example. We give an example shown in Figure 6. Consider an edge e_{ab} with initial support $T(e_{ab}) = 256$. Suppose it has four participants: $DT(e_{ab}) = \{W, S_{(a,d)}, B_{(z,b)}, B_{(a,w)}\}$, so $h(e_{ab}) = 4$. At initialization, the coordinator of $DT(e_{ab})$ computes the slack $\lambda(e_{ab}) = 2^{\lfloor \log_2 \left(\frac{T(e_{ab})}{2h(e_{ab})} \right) \rfloor} = 2^{\lfloor \log_2(256/8) \rfloor} = 32$, and sends this slack

value to all participants of e_{ab} . Therefore, the global participant stores e_{ab} in bucket $R_{cur}[\log_2 32] = R_{cur}[5]$.

Now consider the 2-bloom $B_{(a,d)} = \{(a, x, d), (a, y, d)\}$ and the 3-bloom $S_{(a,d)} = \{(a, b, c, d), (a, b, t, d), (a, u, v, d)\}$ under key (a, d) . Then $|B_{(a,d)}| = 2$, $|S_{(a,d)}| = 3$, and edge e_{ab} appears in the first two priority 3-paths, so $|S_{(a,d)}(e_{ab})| = 2$. Since e_{ab} also lies in the unique priority 2-path of both $B_{(z,b)}$ and $B_{(a,w)}$, the participants $B_{(z,b)}$ and $B_{(a,w)}$ store e_{ab} in buckets $R_{(z,b)}^B[\log_2 32] = R_{(z,b)}^B[5]$ and $R_{(a,w)}^B[\log_2 32] = R_{(a,w)}^B[5]$, respectively. On the 3-bloom side, the normalized slack is $\bar{\lambda}_{(a,d)}(e_{ab}) = 2^{\lfloor \log_2 (256/(2 \cdot 2 \cdot 4)) \rfloor} = 2^{\lfloor \log_2 16 \rfloor} = 16$. Hence, e_{ab} is inserted into $R_{(a,d)}^S[\log_2 16] = R_{(a,d)}^S[4]$.

5.4 DDT-Five Algorithm

Algorithm 4 presents the overall procedure of *DDT-Five*. The detail of peeling process is shown in Algorithm 7 in Appendix.

The peeling process has two parts: processing on the 2-bloom side and processing on the 3-bloom side. When an edge e^* is peeled from a 2-bloom $B_{(a,d)}$, it belongs to exactly one priority 2-path in $B_{(a,d)}$. Let $f^* = \text{twin}_{(a,d)}(e^*)$ be the other edge on the unique priority 2-path $p_2^* = (a, x, d)$ containing e^* . Before removing this path, we first deal with the unreported contribution accumulated from the shared counter $c_{(a,d)}^B$ since its last report, namely $(c_{(a,d)}^B - \ell_{(a,d)}^B(f^*)) - \delta_{(a,d)}^B(f^*)$, and add it to $A(f^*)$. After deleting p_2^* , edge f^* no longer belongs to $B_{(a,d)}$, so it loses all pairs between p_2^* and the currently alive priority 3-paths in $S_{(a,d)}$. Hence, we further add $|S_{(a,d)}|$ to $A(f^*)$. For each alive priority 3-path $p_3 \in S_{(a,d)}$ with $E(p_3) \cap E(p_2^*) \neq \emptyset$, the pair (p_2^*, p_3) is invalid. We compute $\Sigma = \text{Count-Invalid}(p_2^*, p_3)$. Since $A(f^*)$ was just updated, we subtract 1 from $A(f^*)$. Moreover, for each edge $g \in E(p_3) \setminus \{e^*\}$, this invalid pair would also be counted later by the shared-counter report on the 3-bloom side, so we increase $\delta_{(a,d)}^S(g)$ by 1. In particular, if $g = f^*$, then one unit is corrected immediately in $A(f^*)$ and one more unit is stored in $\delta_{(a,d)}^S(f^*)$, matching the value returned by *Count-Invalid*. After that, instead of immediately updating every edge in $S_{(a,d)}$, we simply increase the shared counter $c_{(a,d)}^S$ by 1, scan only the affected buckets and process the corresponding edges according to bucket scanning rule introduced in Section 5.3.

When an edge e^* is peeled from $S_{(a,d)}$, let $r = |S_{(a,d)}(e^*)|$ be the number of alive priority 3-paths in $S_{(a,d)}$ that contain e^* . The algorithm enumerates exactly these destroyed priority 3-paths. For each such path p with $E(p) = \{e^*, e_1, e_2\}$, it processes the other two edges e_1 and e_2 one by one. For each edge $e_i \in \{e_1, e_2\}$, we first remove it from the bucket array of participant $S_{(a,d)}$ and materialize its still-unreported contribution from the shared counter $c_{(a,d)}^S$, namely $|S_{(a,d)}(e_i)|(c_{(a,d)}^S - \ell_{(a,d)}^S(e_i)) - \delta_{(a,d)}^S(e_i)$, into $A(e_i)$, and then update $\ell_{(a,d)}^S(e_i)$ to $c_{(a,d)}^S$ and reset $\delta_{(a,d)}^S(e_i)$ to 0. After that, since the removed priority 3-path p destroys one pairing with each alive priority 2-path in $B_{(a,d)}$, we add $|B_{(a,d)}|$ to $A(e_i)$.

If $e_i \in E(B_{(a,d)})$, then the pair formed by p and the unique alive priority 2-path $p_2(e_i) \in B_{(a,d)}$ containing e_i is invalid. In this case, we compute $\Sigma = \text{Count-Invalid}(p_2(e_i), p)$. For each edge $h \in E(p) \setminus \{e^*\}$ returned by Σ , one invalid count has already been included in the direct update on the destroyed priority 3-path side, so we subtract 1 from $A(h)$. For each edge $h \in E(p_2(e_i))$, the same

Algorithm 4 DDT-Five(G)

Input: graph G with (2,3)-blooms built; initial supports $\{s_5^G(e)\}_{e \in E}$;
Output: C_5 -truss numbers $\{\tau_5(e)\}_{e \in E}$;
1: initialize the global level counter $c_{cur} \leftarrow 0$;
2: **for all** keys (a, d) **do**
3: $c_{(a,d)}^B \leftarrow 0$; $c_{(a,d)}^S \leftarrow 0$;
4: $Q \leftarrow \emptyset$;
5: **for all** $e \in E$ **do**
6: $T(e) \leftarrow s_5^G(e)$; $A(e) \leftarrow 0$; $\ell_{cur}(e) \leftarrow 0$;
7: $h(e) \leftarrow |\mathcal{B}(e)| + |\mathcal{S}(e)| + 1$;
8: **for all** $(a, d) \in \mathcal{B}(e)$ **do**
9: $\ell_{(a,d)}^B(e) \leftarrow 0$; $\delta_{(a,d)}^B(e) \leftarrow 0$;
10: **for all** $(a, d) \in \mathcal{S}(e)$ **do**
11: $\ell_{(a,d)}^S(e) \leftarrow 0$; $\delta_{(a,d)}^S(e) \leftarrow 0$;
12: **if** $T(e) = 0$ **then**
13: $\tau_5(e) \leftarrow 0$, and marked as peeled;
14: **else**
15: INSERTEDGEINTOBUCKETS(e);
16: **while** there exists an unpeeled edge **do**
17: **if** Q is empty **then**
18: increase c_{cur} by 1 and scan the affected buckets of $R_{cur}[\cdot]$;
19: collect all edges e with $A(e) \geq T(e)/2$;
20: **for all** such edges e **do**
21: collect the current increment of e from all its participants;
22: **if** the total increment reaches $T(e)$ **then**
23: insert e into Q ;
24: **else**
25: start a new DT round;
26: **else**
27: $e^* \leftarrow \text{POP}(Q)$;
28: **if** e^* is already peeled **then**
29: **continue**;
30: mark e^* peeled; $\tau_5(e^*) \leftarrow c_{cur}$;
31: PEELEDGE(e^*, Q);

invalid pair would also be counted later by the shared-counter report on the 2-bloom side, so we increase $\delta_{(a,d)}^B(h)$ by 1.

We then decrease $|S_{(a,d)}(e_i)|$ by 1. If it becomes zero, then e_i no longer belongs to $S_{(a,d)}$ and is removed from this participant; otherwise, its slack is recomputed and it is reinserted into the new bucket of $R_{(a,d)}^S[\cdot]$. After all r destroyed priority 3-paths containing e^* have been processed, we update the 2-bloom side in one batch by setting $c_{(a,d)}^B \leftarrow c_{(a,d)}^B + r$ and scanning the affected buckets of participant $B_{(a,d)}$. Finally, we decrease $|S_{(a,d)}|$ by r .

This process repeats until all edges are peeled from the graph.

5.5 Complexity Analysis

Let P_2 and P_3 be the numbers of priority 2-paths and priority 3-paths in the (2, 3)-Bloom Index, respectively. According to Lemmas 4.1 and 4.2, we have $P_2 = O(\gamma m)$ and $P_3 = O(\gamma^2 m)$. Let $M_2 = \sum_{e \in E} |\mathcal{B}(e)|$ and $M_3 = \sum_{e \in E} |\mathcal{S}(e)|$ be the total numbers of edge-to-bloom memberships on the 2-bloom side and the 3-bloom side. We also define the total number of participants across all edge-level DT instances as

$$J = \sum_{e \in E} (|\mathcal{B}(e)| + |\mathcal{S}(e)| + 1) = M_2 + M_3 + m. \quad (7)$$

Since each priority 2-path belongs to one 2-bloom and each priority 3-path belongs to one 3-bloom, and each priority path contains $O(1)$ edges, we have $M_2 = \Theta(P_2)$ and $M_3 = \Theta(P_3)$, hence $J = O(\gamma^2 m)$.

We first analyze the time cost of the peeling process, excluding the costs of scanning the bucket arrays and moving the edge to the new bucket if a DT instance starts a new round.

When an edge e^* is peeled, we process all 2-blooms $B \in \mathcal{B}(e^*)$. Deleting the unique 2-path in B , updating the local counters, and processing the twin-edge accumulator all take $O(1)$ time per processed 2-bloom. Summed over the whole algorithm, this part costs $O(\sum_{e \in E} |\mathcal{B}(e)|) = O(M_2) = O(P_2)$. For the invalid 5-cycle correction during the processing of these 2-blooms, let the removed 2-path under key (a, d) be (a, x, d) , and let e_{twin} denote the other edge on this 2-path besides e^* . We then iterate over the 3-paths in $S_{(a,d)}$ that contain either e^* or e_{twin} , and update the temporal corrections of the other two edges on each such 3-path. Indeed, for any same-key 3-path (a, b, c, d) , an invalid pair can arise only with the 2-path (a, b, d) or the 2-path (a, c, d) , corresponding to the two endpoint edges (a, b) and (c, d) of the 3-path, respectively. Hence, viewed from the 3-path side, each stored 3-path can be scanned by this procedure at most twice over the whole algorithm. Therefore, the total time for correcting invalid 5-cycles is bounded by $O(P_3)$.

For the 3-bloom side, when peeling e^* , we enumerate all destroyed priority 3-paths in each 3-bloom containing e . Observe that each priority 3-path in S can be destroyed once during the entire peeling process. Thus, the total enumeration cost is $O(P_3)$. For each destroyed 3-path, the processing for the two remaining edges, including increasing the accumulator A , detecting maturity, checking whether an edge also belongs to the same-key 2-bloom, and updating temporal corrections when needed, can be performed in $O(1)$ time. Therefore, the cost of these steps is bounded by $O(P_3)$.

Putting these parts together, the total cost of the peeling process, excluding DT bucket scans and reinsertion overhead, is thus bounded by $O(P_2 + P_3) = O(\gamma m + \gamma^2 m) = O(\gamma^2 m)$.

Next, we analyze the running time of bucket scans and edge reinsertion caused by new DT rounds. Recall that the total number of participants over DT instances of all edges is $J = \sum_{e \in E} (|\mathcal{B}(e)| + |S(e)| + 1) = M_2 + M_3 + m = O(\gamma^2 m)$.

For each participant, we maintain a power-of-two bucket array. Whenever its shared counter increases, it scans only the buckets whose indices may be affected by the changed highest bit. For each scanned record, the participant performs only $O(1)$ local work, including updating the accumulator, reset the last reported counter and checking maturity.

An edge is reinserted into a new bucket only when its DT instance starts a new round and its slack changes. By the standard analysis of the P2S-DT framework, for each edge e , the total running time cost of bucket scans and reinsertion events over the whole algorithm is bounded by $O(J(e) \cdot \log s_5^{\max})$, where $s_5^{\max} = \max_{e \in E} s_5^G(e)$ is the largest 5-cycle support of the edges and $J(e) = |\mathcal{B}(e)| + |S(e)| + 1$. Therefore, the total overhead over all participants for all edges is $O(J \cdot \log s_5^{\max})$. Since an edge can be contained in at most $O(n^3)$ distinct 5-cycles, $\log s_5^{\max} = O(\log n) = O(\log m)$. And thus $O(J \cdot \log s_5^{\max}) = O(J \cdot \log m) = O(\gamma^2 m \cdot \log m)$.

Next, we discuss the effect of invalid 5-cycle corrections. In an ideal DT round, once the total increment reported by the participants of an edge e reaches or exceeds $T(e)/2$, the coordinator completes the round. However, in our problem, some pairs correspond to invalid 5-cycles. As a result, when a participant sends an increment, the actual increase received by the coordinator may be smaller than the raw reported amount after subtracting the temporal invalid 5-cycle correction. Therefore, for every DT instance, the number

of messages between participants and the coordinator may exceed the number of its participants in the current round of $DT(e)$.

This does not change the bound. Such extra messages can occur only because of invalid 5-cycle corrections. For a fixed edge e , each additional message can be charged to at least one invalid 5-cycle containing e . Hence, the total number of extra messages associated with e is bounded by the number of invalid 5-cycles containing e . Summed over all edges, this extra cost is no more than the total number of invalid 5-cycles in G . For any fixed priority 3-path in a 3-bloom, at most two priority 2-paths in the same-key 2-bloom can form invalid pairs with it, corresponding to the two possible shared edges (except for the middle one) of the 3-path. Therefore, the total number of invalid 5-cycles in G is bounded by $O(P_3) = O(\gamma^2 m)$.

Moreover, while deleting a 3-path from $S_{(a,d)}$, the participant of an edge needs to send the increment and may need to reinsert to the bucket array if the slack changes. This also does not change the bound because every 3-path only runs this operation at most twice.

Combining all these parts, the overall running time of *DDT-Five* is bounded by $O(\gamma^2 m \cdot \log m)$.

5.6 A Space-Saving Implementation

The space consumption bottleneck of *DDT-Five* is that it needs to store all priority 3-paths in each 3-bloom. To reduce the space consumption in practice, in this section, we give a space-saving implementation of *DDT-Five*, called *Save-DDT*. During index construction, it still enumerates all priority 3-paths to compute $|S_{(a,d)}|$, $|S_{(a,d)}(e)|$, the initial C_5 -supports, and the invalid pair corrections. However, it does not materialize these priority 3-paths in the index.

When the peeling process needs to retrieve the priority 3-paths containing an edge e^* in $S_{(a,d)}$, *Save-DDT* generates these paths on the fly as follows. There are only three possible cases with respect to the possible positions of e^* in a path $p = (a, b, c, d)$. Recall that $\text{Out}(v)$ is the set of out-neighbors of v in the oriented graph, and $\text{Adj}(v)$ the neighbor set of v in the original graph.

- Case 1: $e^* = e_{ab}$. For each $c \in \text{Adj}(b) \setminus \{a, d\}$, retrieve the 3-path if $a \succ c$ and $d \in \text{Out}(c)$.
- Case 2: $e^* = e_{bc}$. Retrieve (a, b, c, d) if $a \in \text{Out}(b)$ and $d \in \text{Out}(c)$, and (a, c, b, d) if $a \in \text{Out}(c)$ and $d \in \text{Out}(b)$.
- Case 3: $e^* = e_{cd}$. For each $b \in \text{Adj}(c) \setminus \{a, d\}$, retrieve the 3-path if $a \in \text{Out}(b)$.

We maintain a set D to record the edges which have been peeled. A generated priority 3-path that contains the current edge e^* is alive if and only if its other two edges do not belong to D . All the other steps of the algorithm are the same as *DDT-Five*.

In our experiments, *Save-DDT* can substantially reduce the memory usage to about 40% as before. Generating 3-paths on the fly during the peeling process increases the running time and degenerates the worst-case overall time complexity to $O(m^{2.5})$. Nonetheless, our experimental results in Section 6 show that such overhead of *Save-DDT* is acceptable in practice. *Save-DDT* provides a practical trade-off between memory consumption and the running time.

5.7 Simplified C_4 -Truss Decomposition

Our framework for C_5 -Truss Decomposition can be simplified to solve C_4 -Truss Decomposition directly.

For 4-cycles, 3-paths and 3-blooms are not needed. Every 4-cycle is uniquely contained in one 2-bloom. Let a be its highest-priority vertex and d be the vertex opposite to a . Then the cycle is uniquely represented by two distinct priority 2-paths with key (a, d) . Therefore, C_4 -Truss Decomposition can be indexed using only the collection of all 2-blooms. For an edge e , its C_4 -Support is $s_4^G(e) = \sum_{(a,d) \in \mathcal{B}_G(e)} (|B_{(a,d)}| - 1)$.

The peeling process is also a simplification of our solution to C_5 -Truss Decomposition. When an edge e^* is peeled, for each $(a, d) \in \mathcal{B}(e^*)$, the unique priority 2-path in $B_{(a,d)}$ containing e^* is removed, and thus $|B_{(a,d)}|$ decreases by one. Consequently, every other alive edge except the twin edge $\text{twin}_{(a,d)}(e^*)$ in $E(B_{(a,d)})$ loses one 4-cycle support from this 2-bloom while $\text{twin}_{(a,d)}(e^*)$ loses exactly $|B_{(a,d)}|$ supports from this 2-bloom. This is the same as the 2-bloom-side update in our 5-cycle Truss Decomposition, except for the 3-blooms and invalid pairs.

The distributed tracking instances are constructed and organized in the same way as that in our *DDT-Five* algorithm. For each edge e , the participants of $DT(e)$ are all 2-blooms containing e and global participant. The shared counter of a 2-bloom records the number of removed 2-paths. Equivalently, for any edge that is still alive in this 2-bloom, it equals the number of destroyed 4-cycles in this 2-bloom that contain this edge. Bucket scanning, and maturity detection follow the same logic as the 2-bloom side of *DDT-Five*.

With our direct algorithm, the blowup factor of 2 in the number of edges introduced by the reduction can be eliminated. This superiority is verified in our experimental study, our algorithm is roughly 2× faster than the reduction-based solution.

6 Experiments

6.1 Experimental Setup

Competitors. We conduct experiments using seven algorithms. For C_5 -Truss Decomposition, we include *Naive-Store*, which explicitly stores all 5-cycles, *Naive-NoStore*, which does not store 5-cycles, and our proposed algorithms *DET-Five*, *DDT-Five* and *Save-DDT*. For C_4 -Truss Decomposition, we denote the algorithm that reduces the problem to bitruss decomposition on a bipartite graph by *Red-BiT*, and the simplified C_4 -Truss Decomposition algorithm introduced in Section 5.7 by *DDT-Four*. The source code is available at <https://github.com/HenryLimHongni/High-Truss>.

Environment. We conducted our experiments on a machine with 1,024 GB of memory and an Intel Xeon Gold 6448H CPU. All these seven algorithms are implemented in C++ and compiled with the -O3 optimization on Linux operating system. All programs were executed in a single thread.

Time and Space Limits. In our experiments, an execution of an algorithm is terminated manually if its running time exceeds the limit of 150,000 seconds, in which case the result is recorded as INF. Moreover, if the memory space usage of an algorithm exceeds the 1,024 GB memory of our machine, its result is recorded as OOM (indicating out of memory).

Datasets. We evaluated our performance on 10 public datasets collected from SNAP [20], KONECT [18], and DIMACS [2]. Table 2 summarizes their statistics. Beyond the numbers of vertices and edges, we report the numbers of 4-cycles ($\#C_4$), 5-cycles ($\#C_5$), and the degeneracy (i.e., γ).

6.2 Performance of C_5 -Truss Decomposition

Overall Running Time. Figure 7 reports the running time of the five C_5 -Truss Decomposition algorithms on each dataset. The y-axis is shown in base-10 logarithmic scale, and each group of bars corresponds to one dataset; within each group, the five bars represent the total running time of the five competitors.

Across all the ten datasets, *DDT-Five*, *DET-Five* and *Save-DDT* consistently outperform *Naive-Store* and *Naive-NoStore* by orders of magnitude, and *DDT-Five* is the fastest among all the competitors. On Web-Bs, Dblp2, Dbpedia, Zhishi, and DIMACS10, *Naive-Store* runs out of memory, while *Naive-NoStore* exceeds the time limit. In contrast, *DDT-Five* and *Save-DDT* successfully finish on all datasets, and *DET-Five* finishes on all except Web-Bs within the time limit.

On most datasets, *DDT-Five* and *DET-Five* achieve a speedup of 10× (one order of magnitude) compared with *Naive-Store* and *Naive-NoStore*. *Save-DDT* also clearly outperforms the two naive methods, although its speedup is smaller on some small datasets such as Email-Eu and Email-Er due to the overhead of generating paths on the fly. While on smaller datasets such as Email-Eu and Email-Er, *DDT-Five* and *DET-Five* show similar performance, on larger datasets, *DDT-Five* provides a big improvement over *DET-Five*. In particular, on Web-Bs and Dbpedia, *DDT-Five* is nearly an order of magnitude faster than *DET-Five*. The relative performance of *DET-Five* and *DDT-Five* is affected by the number of 5-cycles and the local graph density. These two graph properties do not always change proportionally. For example, consider datasets Dblp2 and Web-Bs; as shown in Table 2, Dblp2 has a higher degeneracy than Web-Bs, with 286 and 201 respectively, yet Web-Bs contains more than 20 times of 5-cycles. The speedup of *DDT-Five* over *DET-Five* is over 13× on Web-Bs, while it is about 5× on Dblp2. This observation is as expected because *DET-Five* needs to enumerate all 5-cycles while *DDT-Five* does not. On datasets with more 5-cycles, the performance gap between *DDT-Five* and *DET-Five* becomes larger. We can also observe that although Web-Bs has only about one tenth of the edges of the largest dataset DIMACS10, its degeneracy is nearly 10× higher and its 5-cycle number is roughly 500× larger. Both *DDT-Five* and *DET-Five* take substantially longer running time on Web-Bs than on DIMACS10 (especially *DET-Five*, which is more than 10× slower). Moreover, the performance advantage of *DDT-Five* over *DET-Five* is more pronounced on Web-Bs than on DIMACS10. These results show that 5-cycles number and the local graph density both affect the relative performance between *DDT-Five* and *DET-Five*.

Compared with *DDT-Five*, *Save-DDT* is slower as it needs to generate priority 3-paths during the peeling process. Nonetheless, it is still faster than *DET-Five* on large datasets including Web-Bs, Dblp2, Zhishi, and DIMACS10. For example, *Save-DDT* is over 4× faster than *DET-Five* on Web-Bs. Combining the space consumption as discussed next, these results show the tradeoff of *Save-DDT* between memory consumption and running time.

Space Consumption. Table 3 reports the peak memory consumption of the five C_5 -Truss Decomposition algorithms. As shown in Table 3, *Naive-Store* consumes the most memory because it explicitly stores all 5-cycles in the graph, and it exceeds the 1024 GB limit on the five largest datasets. *Naive-NoStore* consumes little memory because it only stores the basic graph information; however, its

Table 2: Summary of datasets and statistics

Dataset	#V	#E	#C ₄ (10 ⁶)	#C ₅ (10 ⁶)	γ
Email-Eu	986	16,064	4.6	245.6	34
Email-Er	36,692	183,831	36.3	2,587.8	43
Gowalla	196,591	950,327	146.7	12,932.2	51
Dblp1	317,080	1,049,866	55.1	3,449.3	113
Web-Google	875,713	4,322,051	539.6	11,072.6	44
Web-Bs	685,230	6,649,470	127,118.3	8,016,054.0	201
Dblp2	1,824,701	8,344,615	1,911.4	336,692.1	286
Dbpedia	3,966,895	12,610,982	31,569.2	311,443.9	20
Zhishi	2,452,673	18,690,759	3,135.3	38,838.4	16
DIMACS10	8,388,607	63,501,393	1,673.4	16,465.5	20

Table 3: Space consumption (GB) among five algorithms

Dataset	DET-Five	DDT-Five	Save-DDT	Naive-Store	Naive-NoStore
Email-Eu	0.78	0.83	0.30	15.56	0.007
Email-Er	8.395	8.92	3.55	162.49	0.063
Gowalla	33.18	35.394	13.31	810.84	0.32
Dblp1	7.85	8.474	3.07	214.63	0.32
Web-Google	31.43	33.56	12.87	699.06	1.25
Web-Bs	285.82	291.41	89.75	OOM	INF
Dblp2	233.19	251.85	92.57	OOM	INF
Dbpedia	48.43	50.94	25.01	OOM	INF
Zhishi	103.59	112.94	48.11	OOM	INF
DIMACS10	438.08	520.67	208.93	OOM	INF

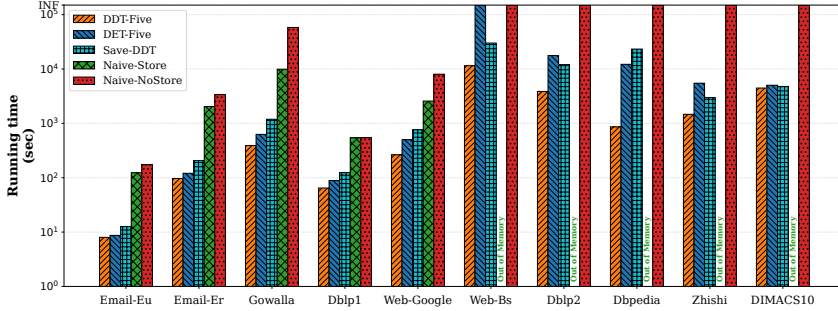
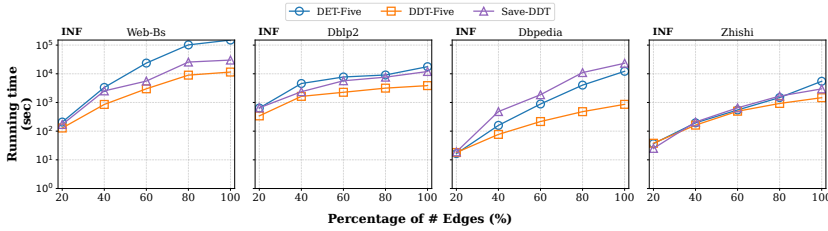


Figure 7: Performance on ten public datasets

Figure 8: Performance on different datasets for C₅-Truss Decomposition

running time cost is prohibitively expensive and exceeds the time limit on the five largest datasets.

DDT-Five consumes slightly more memory than *DET-Five* because it needs to maintain the DT information for each edge. Nevertheless, the main memory cost of both algorithms comes from storing the priority 2-paths and priority 3-paths. In contrast, *Save-DDT* does not store the priority 3-paths and therefore uses less memory than both *DET-Five* and *DDT-Five* on every dataset. On average, *Save-DDT* uses only around 40% of the memory required by *DDT-Five*. On some large datasets like Web-Bs, its memory consumption is even close to one third of that of *DDT-Five*.

Overall, *DDT-Five* remains the preferred method when running time is the principal consideration. When memory is limited, *Save-DDT* provides a more practical choice.

Scalability Study. To evaluate the effect of graph size for our proposed algorithms *DET-Five* and *DDT-Five* and *Save-DDT*, we conduct the scalability study on Web-Bs, Dblp2, Dbpedia and Zhishi. We randomly sample 20%, 40%, 60%, and 80% of edges from each of the four datasets, remove isolated vertices, and obtain new graphs accordingly. The degeneracy of the obtained graphs is shown in Appendix E. We then collect the running time of *DET-Five* and *DDT-Five* as well as *Save-DDT* on these sampled graphs. As shown in Figure 8, *DDT-Five* consistently outperforms *DET-Five* on Web-Bs and Dblp2. When the sampled graphs are relatively small, the

advantage of *DDT-Five* is less pronounced; however, as the sampling rate increases, the performance gap becomes increasingly evident. In particular, once the sampling rate exceeds 60%, *DDT-Five* is consistently faster than *DET-Five* by over ten times on Web-Bs. *Save-DDT* is faster than *DET-Five* on Web-Bs in all sampling rates, and becomes faster than *DET-Five* on Dblp2 when the sampling rate reaches 40% or higher.

For Dbpedia and Zhishi, at 20% sampling rate, *DET-Five* can be faster than *DDT-Five* on these two datasets. On Dbpedia, *DDT-Five* becomes clearly faster than *DET-Five* when the sampling rate reaches 40% or higher, and achieves nearly a 10× speedup at the 80% sampling rate. On Zhishi, the two methods remain close at the 40% and 60% sampling rates, while at the 80% sampling rate *DDT-Five* becomes noticeably faster than *DET-Five*. This phenomenon is expected. Compared with *DET-Five*, *DDT-Five* introduces extra running time overhead due to additional index construction and DT operations. When the graph is small under low sampling rates, the benefit of *DDT-Five* in the peeling process is relatively limited and cannot fully cover this overhead. This is similar to *Save-DDT*, because it requires additional time to generate priority 3-paths during the peeling process.

Impact of Graph Density. We study how graph density affects the performance of *DET-Five* and *DDT-Five* on synthetic graphs. We fix the degeneracy to 60, set the number of edges to $m = |E| = 4 \times 10^6$,

Table 4: Runtime and space of Red-Bit and DDT-Four on 10 datasets

Dataset	Running time (s)		Space consumption (GB)	
	Red-Bit	DDT-Four	Red-Bit	DDT-Four
Email-Eu	0.80	0.47	0.11	0.06
Email-Er	9.90	5.03	1.01	0.51
Gowalla	42.33	21.34	3.80	1.93
Dblp1	11.65	5.93	1.60	0.85
Web-Google	56.94	30.08	8.25	4.33
Web-Bs	3295.29	1395.37	28.35	14.68
Dblp2	274.62	135.15	22.25	11.45
Dbpedia	504.82	173.34	12.33	6.54
Zhishi	519.23	244.17	26.15	13.90
DIMACS10	1220.93	688.21	161.59	83.77

Table 5: Running time on different synthetic graphs

Setting	Value	DET-Five (s)	DDT-Five (s)	Save-DDT (s)
Fixed degeneracy $\gamma = 60$, fixed size $ E = 4 \times 10^6$				
$ V $	2×10^5	3506.80	2149.24	4903.57
$ V $	4×10^5	3526.35	1573.01	2131.03
$ V $	6×10^5	3236.87	1410.89	2026.82
$ V $	8×10^5	3304.94	1555.93	2181.12
Fixed size $n = 4 \times 10^5$, $ E = 4 \times 10^6$				
γ	25	313.55	315.05	331.38
γ	50	1576.98	1052.46	1248.37
γ	75	6298.41	3052.02	4255.09
γ	100	18009.92	11514.86	24577.79

and vary the number of vertices as $n = |V| \in \{2 \times 10^5, 4 \times 10^5, 6 \times 10^5, 8 \times 10^5\}$. The details of the generation of these synthetic graphs can be found in Appendix F. Table 5 reports the running time results. Overall, the running times of both *DET-Five* and *DDT-Five* remain relatively stable across all settings, with slight increasing trend. This suggests that the performance of both algorithms is relatively insensitive to the change in $|V|$ in this experiment. Meanwhile, *DDT-Five* consistently outperforms *DET-Five* on these four graphs.

Next, we study the impact of the degeneracy. We fix the number of vertices and edges to 4×10^5 and 4×10^6 , respectively, and vary the degeneracy $\gamma \in \{25, 50, 75, 100\}$. We observe that the running time of both *DET-Five* and *DDT-Five* increases significantly as γ grows. This shows that higher local density makes the running time of the algorithms longer. When $\gamma = 25$, the two algorithms have very similar running time. As γ becomes larger, *DDT-Five* becomes consistently faster than *DET-Five*, and the performance gap gradually increases. This result shows that *DDT-Five* becomes more efficient on graphs with denser local structure. Compared with the overall density of the graph, the degeneracy has a greater impact on the efficiency of *DET-Five* and *DDT-Five*.

Choosing among *DET-Five*, *DDT-Five*, and *Save-DDT*. The three algorithms provide different tradeoffs between running time and memory consumption. When the memory space is a main concern, we suggest using *Save-DDT*. Otherwise, according to our experimental observations on the synthetic graphs and 10 real-world datasets, we recommend the following empirical guideline to choose between *DET-Five* and *DDT-Five*. If $m \leq 4 \times 10^6$ and $\gamma \leq 25$, the graph is considered having few 5-cycles and thus, *DET-Five* is recommended. Otherwise, *DDT-Five* would be the one to choose.

6.3 Performance of C_4 -Truss Decomposition

Overall Running Time. Table 4 reports the running time of *Red-BiT* and *DDT-Four* on ten datasets. *DDT-Four* consistently outperforms *Red-BiT* on all datasets, and its running time is usually about half of that of *Red-BiT*. The main reason is that *Red-BiT* first transforms the original graph into a bipartite graph, which doubles the number of edges, and then performs bitruss decomposition on the enlarged graph. Moreover, the graph transformation itself also introduces additional preprocessing overhead. In contrast, *DDT-Four* works directly on the original graph with the simplified bloom-based framework, and thus avoids both the factor-of-two blowup and the extra transformation cost.

Space Consumption. Table 4 also reports the memory usage of *Red-BiT* and *DDT-Four* on ten datasets. We can see that the space consumption of *DDT-Four* is roughly half of that of *Red-BiT*. This shows that after transforming the original graph into a bipartite graph, *Red-BiT* needs almost twice as much memory for storing the edges and the corresponding index structures. In contrast, *DDT-Four* works directly on the original graph, and therefore avoids this two times memory overhead.

7 Two Case Studies

7.1 Case Study on Recommendation

Datasets. We conduct a case study in the recommendation on two real-world datasets: Movielens-100K [12] and Foursquare-Tokyo [35].

Table 6: Recommendation performance on two datasets

Method	MovieLens-100K		Foursquare-Tokyo	
	NDCG@5	Precision@5	NDCG@5	Precision@5
ItemKNN	0.1473	0.1402	0.1342	0.0633
C_3 -truss	0.1514	0.1421	0.2171	0.0918
C_4 -truss	0.1213	0.1084	0.2192	0.0878
C_5 -truss	0.1703	0.1570	0.2243	0.0918
C_6 -truss	INF	INF	0.2067	0.0878

In MovieLens-100K, each item is a movie and each user-item edge represents a rating interaction. For two movies i and j , their similarity is computed as $s(i, j) = 0.8 \cos(\mathbf{g}_i, \mathbf{g}_j) + 0.2 \frac{1}{1 + |y_i - y_j|/10}$, where $\mathbf{g}_i$ and y_i denote the genre vector and release year of movie i , respectively. For each movie i , an item-item edge is added between i and j if $s(i, j) > 0.4$ and j is among the top-20 most similar movies to i . We use the first 80% of interactions in chronological order to construct the user-item graph and select test edges from the remaining 20%, retaining only unseen user-item pairs whose endpoints both appear in the constructed graph.

In Foursquare-Tokyo, items are department stores. Each user-item edge represents a user's visit to a department store. Two department stores are connected by an item-item edge if their geographical distance is at most 2 km. We use the visit records before December 1, 2012 to construct the user-item graph and the visit records (i.e., user-item edges) which have both endpoints in the user-item graph after this date for testing.

For both datasets, only the unseen user-item edges would be tested. The statistics of the two graphs can be found in Appendix G. **Competitors.** We use *ItemKNN* [28], a widely used collaborative filtering algorithm as the baseline method, and use C_3 -truss, C_4 -truss, C_5 -truss and C_6 -truss decompositions for comparisons.

For C_3 -truss, for an unseen user-item pair (u, i) , we consider each item b that user u has interacted with and that is connected to i by an item-item edge $e_{b,i}$. Let $E_3(u, i)$ denote the set of such item-item edges. The recommendation score is $S_3(u, i) = \max_{e \in E_3(u, i)} \tau_3(e)$. If $E_3(u, i)$ is empty, the score is zero.

For C_4 -truss, consider a 4-cycle (u, a, b, c, u) , where u is a user and a, b, c are distinct items. User u has interacted with a and c , while b is connected to both items by the item-item edges $e_{a,b}$ and $e_{b,c}$. This cycle supports recommending the unseen item b to u . For each such 4-cycle, we use the larger truss number of its two item-item edges: $\max\{\tau_4(e_{a,b}), \tau_4(e_{b,c})\}$. If item b and u are in multiple 4-cycles, we take the largest score among these cycles.

For C_5 -truss, consider a 5-cycle (u_1, a, u_2, b, c, u_1) , where u_1, u_2 are users and a, b, c are items. This cycle supports recommending b to u_1 and c to u_2 . We use the C_5 -truss number of the item-item edge $e_{b,c}$ as the recommendation score. Since the same item may be recommended to the same user through multiple 5-cycles, let $E_5(u, i)$ denote the set of item-item edges in all 5-cycles that recommend item i to user u . The recommendation score is $S_5(u, i) = \max_{e \in E_5(u, i)} \tau_5(e)$. If $E_5(u, i)$ is empty, the score is zero.

For C_6 -truss, consider a 6-cycle $(u_1, a, b, u_2, c, d, u_1)$, where u_1, u_2 are users and a, b, c, d are items. This cycle recommends b, c to u_1 and a, d to u_2 . The recommendations of a and b use item-item edge $e_{a,b}$ to compute the recommendation score, while those of c and d use $e_{c,d}$. Let $E_6(u, i)$ be the set of item-item edges in all 6-cycles that recommend item i to user u . The recommendation score is $S_6(u, i) = \max_{e \in E_6(u, i)} \tau_6(e)$.

In the above comparisons, ties are broken by ranking items by a non-increasing order of their degrees and then by ID.

Evaluation Metrics. We report NDCG@5 and Precision@5 to evaluate the performance of all comparisons on both datasets.

Results. The results are shown in Table 6. On MovieLens-100K, C_5 -truss achieves the best performance on both metrics, outperforming ItemKNN, C_3 -truss, C_4 -truss. The C_6 -truss runs for more than 24 hours, so it terminates automatically. On Foursquare-Tokyo, C_5 -truss achieves the highest NDCG@5 of 0.2243 and Precision@5 of 0.0918, the same as C_3 -truss for the best Precision@5. Overall, C_5 -truss consistently provides the strongest recommendation performance across the two datasets.

7.2 Case Study on Protein Helix-Core Detection

We also conduct a case study on Protein Helix-Core Detection to demonstrate the application of C_4 -truss and C_5 -truss decompositions in the protein analysis scenario. Due to the page limit, we defer the detailed setup of this case study in Appendix H. Overall, the experimental results show that C_4 -truss and C_5 -truss decompositions capture useful structural signals that are not fully reflected by C_3 - or C_6 -truss scores in Protein Helix-Core Detection.

8 Conclusion

In this paper, we studied high-order truss decomposition with a focus on 4-cycle and 5-cycle trusses. We first proved that 4-cycle truss decomposition can be reduced to bitruss decomposition on bipartite graphs, and thus can be solved with the state-of-the-art algorithm in $O(\gamma \cdot m \cdot \log m)$ running time, where γ is the degeneracy and m is the number of edges. Then we proposed a unified framework for both 4-cycle and 5-cycle truss decompositions that avoids enumerating all cycles. Our algorithms achieve time complexities of $O(\gamma \cdot m \cdot \log m)$ for 4-cycle trusses without the reduction overhead and $O(\gamma^2 \cdot m \cdot \log m)$ for 5-cycle trusses, improving the existing upper bounds by roughly a factor of $O(\sqrt{m})$. Experiments on 10 real-world and over multiple synthetic datasets demonstrated the efficiency of our methods.

Acknowledgments

Junhao Gan is supported by ARC DP230102908. Lijun Chang is supported by ARC DP220103731. Seeun William Umboh is supported by ARC DP240101353.

References

- [1] 1971. Crystallography: Protein Data Bank. *Nature New Biology* 233, 42 (Oct 1971), 223–223. doi:10.1038/newbio233223b0
- [2] David A. Bader, Henning Meyerhenke, Peter Sanders, Christian Schulz, Andrea Kappes, and Dorothea Wagner. 2014. *Benchmarking for Graph Clustering and Partitioning*. Springer New York, New York, NY, 73–82. doi:10.1007/978-1-4614-6170-8_23
- [3] Wen Bai, Xiaodong Zhu, and Yuncheng Jiang. 2025. Dense subgraph mining in dynamic bipartite graphs. *Information Sciences* (2025), 122758.
- [4] Balabhaskar Balasundaram, Sergiy Butenko, and Ilya V Hicks. 2011. Clique relaxations in social network analysis: The maximum k-plex problem. *Operations Research* 59, 1 (2011), 133–142.
- [5] Vladimir Batagelj and Matjaz Zaversnik. 2003. An $o(m)$ algorithm for cores decomposition of networks. *arXiv preprint cs/0310049* (2003).
- [6] Suman K Bera, Noujan Pashanasangi, and C Seshadhri. 2019. Linear time subgraph counting, graph degeneracy, and the chasm at size six. *arXiv preprint arXiv:1911.05896* (2019).
- [7] Broto Chakraborty and Nita Parekh. 2016. NAPS: network analysis of protein structures. *Nucleic acids research* 44, W1 (2016), W375–W382.
- [8] Lijun Chang. 2023. Efficient maximum k-defective clique computation with improved time complexity. *Proceedings of the ACM on Management of Data* 1, 3 (2023), 1–26.
- [9] Lijun Chang, Mouyi Xu, and Darren Strash. 2022. Efficient maximum k-plex computation over large sparse graphs. *Proceedings of the VLDB Endowment* 16, 2 (2022), 127–139.
- [10] Jonathan D Cohen. 2019. Trusses and trapezes: Easily-interpreted communities in social networks. *arXiv preprint arXiv:1907.09417* (2019).
- [11] Graham Cormode, Shanmugavelayutham Muthukrishnan, and Ke Yi. 2011. Algorithms for distributed functional monitoring. *ACM Transactions on Algorithms (TALG)* 7, 2 (2011), 1–20.
- [12] F Maxwell Harper and Joseph A Konstan. 2015. The movielens datasets: History and context. *Acem transactions on interactive intelligent systems (tiis)* 5, 4 (2015), 1–19.
- [13] Jun Huan, Deepak Bandyopadhyay, Wei Wang, Jack Snoeyink, Jan Prins, and Alexander Tropsha. 2005. Comparing graph representations of protein structure for mining family-specific residue-based packing motifs. *Journal of Computational Biology* 12, 6 (2005), 657–671.
- [14] Xin Huang, Laks VS Lakshmanan, and Jianliang Xu. 2022. Cohesive Community Search. In *Community Search over Big Graphs*. Springer, 27–81.
- [15] Xin Huang, Wei Lu, and Laks VS Lakshmanan. 2016. Truss decomposition of probabilistic graphs: Semantics and algorithms. In *Proceedings of the 2016 International Conference on Management of Data*. 77–90.
- [16] Dahee Kim, Song Kim, Jeongseon Kim, Junghoon Kim, Kaiyu Feng, Sungsu Lim, and Jungeun Kim. 2024. Experimental analysis and evaluation of cohesive subgraph discovery. *Information Sciences* 672 (2024), 120664.
- [17] Lukasz Kowalik. 2003. Short cycles in planar graphs. In *International Workshop on Graph-Theoretic Concepts in Computer Science*. Springer, 284–296.
- [18] Jérôme Kunegis. 2013. Konect: the koblenz network collection. In *Proceedings of the 22nd international conference on world wide web*. 1343–1350.
- [19] Kartik Lakhota, Rajgopal Kannan, and Viktor Prasanna. 2023. Parallel peeling of bipartite networks for hierarchical dense subgraph discovery. *ACM Transactions on Parallel Computing* 10, 2 (2023), 1–35.
- [20] Jure Leskovec and Rok Sosič. 2016. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)* 8, 1 (2016), 1–20.
- [21] Rong-Hua Li, Jeffrey Xu Yu, and Rui Mao. 2013. Efficient core maintenance in large dynamic graphs. *IEEE transactions on knowledge and data engineering* 26, 10 (2013), 2453–2465.
- [22] Don R Lick and Arthur T White. 1970. k-Degenerate graphs. *Canadian Journal of Mathematics* 22, 5 (1970), 1082–1096.
- [23] Fengnian Lin, Boyu Ruan, Junhao Gan, and Lei Li. 2025. Efficient Bitruss Decomposition without Butterfly Enumeration. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*. 1718–1728.
- [24] Chunyu Luo, Yi Zhou, Zhengren Wang, and Mingyu Xiao. 2024. A Faster Branching Algorithm for the Maximum k-Defective Clique Problem. *arXiv preprint arXiv:2407.16588* (2024).
- [25] SM Patra and Saraswathi Vishveshwara. 2000. Backbone cluster identification in proteins by a graph theoretical method. *Biophysical chemistry* 84, 1 (2000), 13–25.
- [26] Tahereh Pourhabibi, Kok-Leong Ong, Booi H Kam, and Yee Ling Boo. 2020. Fraud detection: A systematic literature review of graph-based anomaly detection approaches. *Decision Support Systems* 133 (2020), 113303.
- [27] Ahmet Erdem Sariyuce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V Çatalyürek. 2013. Streaming algorithms for k-core decomposition. *Proceedings of the VLDB Endowment* 6, 6 (2013), 433–444.
- [28] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. 2001. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th international conference on World Wide Web*. 285–295.
- [29] Jessica Shi, Louisa Ruixue Huang, and Julian Shun. 2022. Parallel five-cycle counting algorithms. *ACM Journal of Experimental Algorithmics* 27 (2022), 1–23.
- [30] Jayme L Szwarcfiter. 2003. A survey on clique graphs. In *Recent advances in algorithms and combinatorics*. Springer, 109–136.
- [31] Jia Wang and James Cheng. 2012. Truss decomposition in massive networks. *arXiv preprint arXiv:1205.6693* (2012).
- [32] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2020. Efficient bitruss decomposition for large-scale bipartite graphs. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 661–672.
- [33] Mark Weber, Jie Chen, Toyotaro Suzumura, Aldo Pareja, Tengfei Ma, Hiroki Kanezashi, Tim Kaler, Charles E Leiserson, and Tao B Schardl. 2018. Scalable graph learning for anti-money laundering: A first look. *arXiv preprint arXiv:1812.00076* (2018).
- [34] Dong Wen, Lu Qin, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2018. I/o efficient core graph decomposition: application to degeneracy ordering. *IEEE Transactions on Knowledge and Data Engineering* 31, 1 (2018), 75–90.
- [35] Dingqi Yang, Daqing Zhang, Vincent W Zheng, and Zhiyong Yu. 2014. Modeling user activity preference by leveraging user spatial temporal characteristics in LBSNs. *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 45, 1 (2014),

- 129–142.
- [36] Chen Zhang, Fan Zhang, Wenjie Zhang, Boge Liu, Ying Zhang, Lu Qin, and Xuemin Lin. 2020. Exploring finer granularity within the cores: Efficient (k, p) -core computation. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 181–192.
 - [37] Chenghan Zhang, Yuanyuan Zhu, and Lijun Chang. 2025. A local search approach to efficient (k, p) -core maintenance. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–26.
 - [38] Zhuo Zhang, Junhao Gan, Zhifeng Bao, Seyed Mohammad Hussein Kazemi, Guangyong Chen, and Fengyuan Zhu. 2022. Approximate range thresholding. In *Proceedings of the 2022 International Conference on Management of Data*. 1108–1121.
 - [39] Zhaonian Zou. 2016. Bitruss decomposition of bipartite graphs. In *International conference on database systems for advanced applications*. Springer, 218–233.

A Proof of Theorem 3.1

PROOF. For any subgraph $H \subseteq G$, let $\tilde{H}$ be the bipartite graph obtained from H by the same reduction. Fix any edge $e = (x, y) \in E(H)$, and let its two copies in $\tilde{H}$ be $\tilde{e}_1 = (x_L, y_R)$ and $\tilde{e}_2 = (y_L, x_R)$.

We first show that the C_4 -support of e in H is exactly equal to the butterfly supports of $\tilde{e}_1$ and $\tilde{e}_2$ in $\tilde{H}$. Indeed, every 4-cycle (x, y, a, b) in H containing e corresponds to a unique butterfly (x_L, y_R, a_L, b_R) in $\tilde{H}$ containing $\tilde{e}_1$, and also to a unique butterfly (y_L, x_R, b_L, a_R) containing $\tilde{e}_2$. Conversely, every butterfly in $\tilde{H}$ containing $\tilde{e}_1$ (or $\tilde{e}_2$) uniquely corresponds to a 4-cycle in H containing e . Hence, $s_4^H(e) = s_b^{\tilde{H}}(\tilde{e}_1) = s_b^{\tilde{H}}(\tilde{e}_2)$.

More generally, for every edge $f \in E(H)$, its C_4 -support in H equals the butterfly support of each copy of f in $\tilde{H}$. Therefore, H is a k - C_4 -truss if and only if $\tilde{H}$ is a k -bitruss. It follows that an edge e belongs to a k - C_4 -truss of G if and only if its copies $\tilde{e}_1$ and $\tilde{e}_2$ belong to a k -bitruss of $\tilde{G}$. Hence the maximum such k is the same for e , $\tilde{e}_1$, and $\tilde{e}_2$, and thus $\tau_4(e) = \tau_B(\tilde{e}_1) = \tau_B(\tilde{e}_2)$. $\square$

B Missing Proofs in Section 4.2

PROOF OF LEMMA 4.1. By definition, the number of priority 2-paths is bounded by $O(\sum_{b \in V} |\text{Out}(b)| \cdot |\text{Adj}(b)|)$. By Fact 2, we have $|\text{Out}(b)| \leq \gamma$ for all $b \in V$. Therefore, $O(\sum_{b \in V} |\text{Out}(b)| \cdot |\text{Adj}(b)|) = O(\gamma m)$. The $O(\gamma m)$ enumeration time bound follows immediately from the aforementioned enumeration process. $\square$

PROOF OF LEMMA 4.2. By Lemma 4.1, there are $O(\gamma m)$ priority 2-paths. Each of them can be extended in $O(\gamma)$ time since $|\text{Out}(c)| \leq \gamma$. Thus, the enumeration time and the number of priority 3-paths are bounded by $O(\gamma^2 m)$. $\square$

PROOF OF LEMMA 4.4. Since the two paths have the same key (a, d) , together they form the closed walk $a \rightarrow x \rightarrow d \rightarrow c \rightarrow b \rightarrow a$. This is a 5-cycle if and only if its five vertices are distinct. The only possible repetition is $x = b$ or $x = c$. Hence, the two paths can form a 5-cycle if and only if $x \notin \{b, c\}$. $\square$

C Detailed Procedures in DET-Five

We present the detailed pseudocode of the peeling operations used in *DET-Five*. Algorithm 7 is the wrapper procedure for removing a peeled edge. Algorithms 5 and 6 describe how to process the 2-bloom side and the 3-bloom side, respectively.

D Detailed Procedures in DDT-Five

This section presents the detailed peeling procedures of *DDT-Five*.

E Degeneracy under Edge Sampling

Table 7 reports the degeneracy values under different edge sampling ratios.

F Synthetic Graph Generation

For all synthetic graphs, we first create a planted core and then generate the rest of the graph. After the planted core is created, the remaining vertices are divided into unequal communities. If the

Algorithm 5 PEELTWOBLOOMSIDE(H, e^*)

Input: H ; supports $s_5^H(\cdot)$; for each key (a, d) , the current states of $B_{(a,d)}$ and $S_{(a,d)}$, including $|B_{(a,d)}|$, $|S_{(a,d)}|$, $E(B_{(a,d)})$, $E(S_{(a,d)})$, and $|S_{(a,d)}(g)|$ for each edge $g \in E(S_{(a,d)})$; incidences $\mathcal{B}_H(\cdot)$; twin pointers $\text{twin}_{(a,d)}(\cdot)$;

Output: updated states for all touched keys; affected edge set A_2 ;

```

1:  $A_2 \leftarrow \emptyset$ ;
2: for all  $(a, d) \in \mathcal{B}_H(e^*)$  do
3:   find the unique alive priority 2-path  $p_2^* = (a, x, d) \in B_{(a,d)}$  such
   that  $e^* \in \{(a, x), (x, d)\}$ ;
4:    $f^* \leftarrow \text{twin}_{(a,d)}(e^*)$ ;
5:   delete  $p_2^*$  from  $B_{(a,d)}$ ;
6:    $|B_{(a,d)}| \leftarrow |B_{(a,d)}| - 1$ ;
7:    $E(B_{(a,d)}) \leftarrow E(B_{(a,d)}) \setminus \{e^*, f^*\}$ ;
8:    $\mathcal{B}_H(f^*) \leftarrow \mathcal{B}_H(f^*) \setminus \{(a, d)\}$ ;
9:   for all  $g \in E(S_{(a,d)})$  do
10:     $s_5^H(g) \leftarrow s_5^H(g) - |S_{(a,d)}(g)|$ ;
11:     $A_2 \leftarrow A_2 \cup \{g\}$ ;
12:    $s_5^H(f^*) \leftarrow s_5^H(f^*) - |S_{(a,d)}|$ ;
13:    $A_2 \leftarrow A_2 \cup \{f^*\}$ ;
14:    $h_1 \leftarrow (a, x); \quad h_2 \leftarrow (x, d)$ ;
15:   for all  $h \in \{h_1, h_2\}$  do
16:     for all alive priority 3-path  $p_3 \in S_{(a,d)}(h)$  do
17:        $\Sigma \leftarrow \text{Count-Invalid}(p_2^*, p_3)$ ;
18:       for all  $(g, \delta) \in \Sigma$  do
19:         if  $g \neq e^*$  then
20:            $s_5^H(g) \leftarrow s_5^H(g) + \delta$ ;
21:            $A_2 \leftarrow A_2 \cup \{g\}$ ;
22: return  $A_2$ ;
```

Table 7: Degeneracy vs. sampling ratios

Dataset	80%	60%	40%	20%
Web-Bs	157	129	94	49
Db1p2	279	267	230	148
Dbpedia	16	13	9	5
Zhishi	13	10	8	5

number of remaining vertices is N , the number of communities is set to $\lfloor N/400 \rfloor$. The community sizes are generated from normalized Gamma(1.2, 1) random weights, with minimum size 80, and are adjusted so that their total size is N . We add a cycle over these remaining vertices to avoid isolated vertices. The other edges are generated by biased sampling: 92% of candidate edges are sampled within one randomly selected community, and 8% are sampled as bridge edges between different communities.

For the overall graph density experiment, we fix $m = 4 \times 10^6$, set the target maximum core number to 60, and use $n \in \{2 \times 10^5, 4 \times 10^5, 6 \times 10^5, 8 \times 10^5\}$. For each graph, we create a circulant graph as the planted core on 6×10^4 vertices, where every vertex in the core has degree 60. The numbers of remaining vertices are 140,000, 340,000, 540,000, and 740,000, and the corresponding numbers of communities are 350, 850, 1,350, and 1,850. During edge insertion, if the degree of either endpoint exceeds 60, we ignore this insertion and regenerate an insertion.

For the local graph density experiment, we fix $n = 4 \times 10^5$ and $m = 4 \times 10^6$, and use $\gamma \in \{25, 50, 75, 100\}$. For each graph, we create

Algorithm 6 PEELTHREEBLOOMSIDE(H, e^*)

Input: H ; supports $s_5^H(\cdot)$; for each key (a, d) , the current states of $B_{(a,d)}$ and $S_{(a,d)}$, including $|B_{(a,d)}|$, $|S_{(a,d)}|$, $E(B_{(a,d)})$, $E(S_{(a,d)})$, and $|S_{(a,d)}(g)|$ for each edge $g \in E(S_{(a,d)})$; incidences $S_H(\cdot)$; twin pointers $\text{twin}_{(a,d)}(\cdot)$;

Output: updated states for all touched keys; affected edge set A_3 ;

```

1:  $A_3 \leftarrow \emptyset$ ;
2: for all  $(a, d) \in S_H(e^*)$  do
3:    $c^* \leftarrow 0$ ;
4:   for all alive priority 3-path  $p_3 = (a, b, c, d) \in S_{(a,d)}(e^*)$  do
5:      $c^* \leftarrow c^* + 1$ ;
6:     let  $\{g_1, g_2\} = \{(a, b), (b, c), (c, d)\} \setminus \{e^*\}$ ;
7:     for all  $g \in \{g_1, g_2\}$  do
8:        $|S_{(a,d)}(g)| \leftarrow |S_{(a,d)}(g)| - 1$ ;
9:        $s_5^H(g) \leftarrow s_5^H(g) - |B_{(a,d)}|$ ;
10:       $A_3 \leftarrow A_3 \cup \{g\}$ ;
11:      if  $|S_{(a,d)}(g)| = 0$  then
12:         $S_H(g) \leftarrow S_H(g) \setminus \{(a, d)\}$ ;
13:         $E(S_{(a,d)}) \leftarrow E(S_{(a,d)}) \setminus \{g\}$ ;
14:      for all  $g \in \{g_1, g_2\}$  do
15:        if  $g \in E(B_{(a,d)})$  then
16:          let  $p_2(g)$  be the unique alive priority 2-path in  $B_{(a,d)}$ 
17:          that contains  $g$ ;
18:           $\Sigma \leftarrow \text{Count-Invalid}(p_2(g), p_3)$ ;
19:          for all  $(h, \delta) \in \Sigma$  do
20:            if  $h \neq e^*$  then
21:               $s_5^H(h) \leftarrow s_5^H(h) + \delta$ ;
22:               $A_3 \leftarrow A_3 \cup \{h\}$ ;
23:          delete  $p_3$  from  $S_{(a,d)}$ ;
24:       $|S_{(a,d)}| \leftarrow |S_{(a,d)}| - c^*$ ;
25:       $|S_{(a,d)}(e^*)| \leftarrow 0$ ;
26:       $S_H(e^*) \leftarrow S_H(e^*) \setminus \{(a, d)\}$ ;
27:       $E(S_{(a,d)}) \leftarrow E(S_{(a,d)}) \setminus \{e^*\}$ ;
28:      for all  $h \in E(B_{(a,d)})$  do
29:         $s_5^H(h) \leftarrow s_5^H(h) - c^*$ ;
30:         $A_3 \leftarrow A_3 \cup \{h\}$ ;
31: return  $A_3$ ;

```

Algorithm 7 PEEL(H, e^*)

Input: current remaining graph H ; supports $\{s_5^H(e)\}_{e \in E(H)}$; current bloom states and incidences;

Output: updated $H \leftarrow H \setminus \{e^*\}$ and updated states; affected edge set A ;

```

1:  $A \leftarrow \emptyset$ ;
2: remove  $e^*$  from  $E(H)$ ;
3:  $A \leftarrow A \cup \text{PEELTWOBLOOMSIDE}(H, e^*)$ ;
4:  $A \leftarrow A \cup \text{PEELTHREEBLOOMSIDE}(H, e^*)$ ;
5:  $\mathcal{B}_H(e^*) \leftarrow \emptyset$ ;
6:  $S_H(e^*) \leftarrow \emptyset$ ;
7: return  $A$ ;

```

a circulant graph as the planted core on 6×10^4 vertices, where every vertex in the core has degree γ . The remaining 340,000 vertices are divided into 850 communities. During edge insertion, if the degree of each remaining vertex exceeds γ , the edge insertion is ignored and a new one is generated.

Algorithm 8 PELEDGE(e^*, Q)

Input: peeled edge e^* ; peel queue Q ;

Output: update shared counters, accumulators, corrections, and bucket memberships;

```

1: remove  $e^*$  from  $R_{cur}[\cdot]$  and all bloom bucket arrays containing it;
2:  $\text{PEELTHROUGHTWOBLOOM}(e^*, Q)$ ;
3:  $\text{PEELTHROUGHTHREEBLOOM}(e^*, Q)$ ;
4:  $\mathcal{B}(e^*) \leftarrow \emptyset$ ;
5:  $S(e^*) \leftarrow \emptyset$ ;

```

Algorithm 9 INSERTEDGEINTOBUCKETS(e)

Input: an unpeeled edge e with current residual threshold $T(e)$;

Output: insert all current incidences of e into the bucket arrays;

```

1:  $h(e) \leftarrow |\mathcal{B}(e)| + |S(e)| + 1$ ;
2:  $\lambda(e) \leftarrow \max\left\{1, 2^{\left\lceil \log_2 \left( \frac{T(e)}{2h(e)} \right) \right\rceil}\right\}$ ;
3: insert  $e$  into  $R_{cur}[\log_2 \lambda(e)]$ ;
4: for all  $(a, d) \in \mathcal{B}(e)$  do
5:   insert  $e$  into  $R_{(a,d)}^B[\log_2 \lambda(e)]$ ;
6: for all  $(a, d) \in S(e)$  do
7:    $\tilde{\lambda}_{(a,d)}(e) \leftarrow 2^{\left\lceil \log_2 \left( \max\left\{1, \frac{T(e)}{2|S_{(a,d)}(e)|h(e)}\right\} \right) \right\rceil}$ ;
8:   insert  $e$  into  $R_{(a,d)}^S[\log_2 \tilde{\lambda}_{(a,d)}(e)]$ ;

```

Algorithm 10 GLOBALADVANCEANDSCAN()

Input: global bucket array $R_{cur}[\cdot]$;

Output: return $\mathcal{M}$: edges whose accumulated decrement reaches $T(e)/2$;

```

1:  $\mathcal{M} \leftarrow \emptyset$ ;
2:  $c_{old} \leftarrow c_{cur}$ ;  $c_{cur} \leftarrow c_{cur} + 1$ ;
3:  $j \leftarrow \lfloor \log_2(c_{old} \oplus c_{cur}) \rfloor$ ;
4: for  $b = 0$  to  $j$  do
5:   for all  $e \in R_{cur}[b]$  do
6:     apply report (5);
7:     if  $A(e) \geq T(e)/2$  then
8:        $\mathcal{M} \leftarrow \mathcal{M} \cup \{e\}$ ;
9: return  $\mathcal{M}$ ;

```

Table 8: Statistics of MovieLens-100K and Foursquare-Tokyo

Dataset	$ V_U $	$ V_I $	$ E_{UI} $	$ E_{II} $	$ E_{\text{test}} $
MovieLens-100K	751	1,616	79,999	22,945	2,864
Foursquare-Tokyo	916	172	1,679	765	160

G Dataset Statistics in Recommendation Study

Table 8 reports the statistics of the two graphs. Here, $|V_U|$ and $|V_I|$ denote the numbers of users and items, respectively. $|E_{UI}|$ denotes the number of given user-item edges, $|E_{II}|$ denotes the number of item-item edges, and $|E_{\text{test}}|$ denotes the number of new user-item edges used for testing.

H Case Study on Protein Helix-Core Detection

We also conduct a case study on Protein Helix-Core Detection to demonstrate the application of C_4 -truss and C_5 -truss decompositions in the protein analysis scenario. In this task, each protein is represented by a graph, where each of its amino acid residues is a vertex.

Algorithm 11 SCANPARTICIPANT($Z, (a, d), c_{\text{old}}, c_{\text{new}}$)

Input: $Z \in \{B, S\}$, key (a, d) , the corresponding bucket array, and a counter change $c_{\text{old}} \rightarrow c_{\text{new}}$;
Output: return $\mathcal{M}$: edges whose accumulated decrement reaches $T(e)/2$;
1: $\mathcal{M} \leftarrow \emptyset$;
2: $j \leftarrow \lfloor \log_2(c_{\text{old}} \oplus c_{\text{new}}) \rfloor$;
3: **for** $b = 0$ to j **do**
4: **if** $Z = B$ **then**
5: **for all** $e \in R_{(a,d)}^B[b]$ **do**
6: apply report (3);
7: **if** $A(e) \geq T(e)/2$ **then**
8: $\mathcal{M} \leftarrow \mathcal{M} \cup \{e\}$;
9: **if** $Z = S$ **then**
10: **for all** $e \in R_{(a,d)}^S[b]$ **do**
11: apply report (4);
12: **if** $A(e) \geq T(e)/2$ **then**
13: $\mathcal{M} \leftarrow \mathcal{M} \cup \{e\}$;
14: **return** $\mathcal{M}$;

Algorithm 12 COLLECTANDDECIDE(e, Q)

Input: an unpeeled edge e with $A(e) \geq T(e)/2$; peel queue Q ;
Output: either push e into Q or start a new DT round for e ;
1: **if** e is already peeled or $e \in Q$ **then**
2: **return** ;
3: $D(e) \leftarrow A(e) + (c_{\text{cur}} - \ell_{\text{cur}}(e))$;
4: **for all** $(a, d) \in \mathcal{B}(e)$ **do**
5: $D(e) \leftarrow D(e) + (c_{(a,d)}^B - \ell_{(a,d)}^B(e)) - \delta_{(a,d)}^B(e)$;
6: **for all** $(a, d) \in \mathcal{S}(e)$ **do**
7: $D(e) \leftarrow D(e) + |S_{(a,d)}(e)| \cdot (c_{(a,d)}^S - \ell_{(a,d)}^S(e)) - \delta_{(a,d)}^S(e)$;
8: $\ell_{\text{cur}}(e) \leftarrow c_{\text{cur}}$;
9: **for all** $(a, d) \in \mathcal{B}(e)$ **do**
10: $\ell_{(a,d)}^B(e) \leftarrow c_{(a,d)}^B$; $\delta_{(a,d)}^B(e) \leftarrow 0$;
11: **for all** $(a, d) \in \mathcal{S}(e)$ **do**
12: $\ell_{(a,d)}^S(e) \leftarrow c_{(a,d)}^S$; $\delta_{(a,d)}^S(e) \leftarrow 0$;
13: remove all current incidences of e from their bucket arrays;
14: **if** $D(e) \geq T(e)$ **then**
15: $Q \leftarrow Q \cup \{e\}$;
16: **else**
17: $T(e) \leftarrow T(e) - D(e)$;
18: INSERTEDGEINTOBUCKETS(e);
19: $A(e) \leftarrow 0$;

Two sequentially adjacent residues in the same chain are connected by an edge. In addition, two non-adjacent residues in the same chain are connected by an edge if the Euclidean distance between their C_α atoms is at most C_α -cutoff, a domain specific threshold. This is a commonly used construction for protein graphs [13], and we consider two widely used cutoff values, 6.5Å [25] and 7.0Å [7]. The goal is to classify each residue as a helix-core or not.

C_p -Truss Decomposition can be used to perform this task. Specifically, for each protein graph G , we first compute the C_p -truss number $\tau_p(e)$ for each edge in G , and assign to each vertex v a score $S_p(v)$ which is the maximum C_p -truss number of the edges incident on v , namely, $S_p(v) = \max_{e \ni v} \tau_p(e)$. This score $S_p(v)$ indicates how likely vertex v is to be a helix-core. To evaluate the

Algorithm 13 PEELTHROUGHTWOBLUM(e^*, Q)

Input: peeled edge e^* ; peel queue Q ;
1: **for all** $(a, d) \in \mathcal{B}(e^*)$ **do**
2: find the unique alive priority 2-path $p_2^* = (a, x, d) \in B_{(a,d)}$ such that $e^* \in E(p_2^*)$;
3: $f^* \leftarrow \text{twin}_{(a,d)}(e^*)$;
4: remove f^* from $R_{(a,d)}^B[\cdot]$;
5: $A(f^*) \leftarrow A(f^*) + (c_{(a,d)}^B - \ell_{(a,d)}^B(f^*)) - \delta_{(a,d)}^B(f^*)$;
6: $\ell_{(a,d)}^B(f^*) \leftarrow c_{(a,d)}^B$; $\delta_{(a,d)}^B(f^*) \leftarrow 0$;
7: delete p_2^* from $B_{(a,d)}$;
8: $|B_{(a,d)}| \leftarrow |B_{(a,d)}| - 1$;
9: $E(B_{(a,d)}) \leftarrow E(B_{(a,d)}) \setminus \{e^*, f^*\}$;
10: $A(f^*) \leftarrow A(f^*) + |S_{(a,d)}|$;
11: **for all** priority 3-paths $p_3 \in S_{(a,d)}$ with $E(p_3) \cap E(p_2^*) \neq \emptyset$ **do**
12: $\Sigma \leftarrow \text{Count-Invalid}(p_2^*, p_3)$;
13: **for all** $(g, \delta) \in \Sigma$ **do**
14: **if** $g = f^*$ **then**
15: $A(f^*) \leftarrow A(f^*) - 1$;
16: **if** $g \neq e^*$ and $g \in E(p_3)$ **then**
17: $\delta_{(a,d)}^S(g) \leftarrow \delta_{(a,d)}^S(g) + 1$;
18: delete (a, d) from $\mathcal{B}(f^*)$;
19: $c_{\text{old}} \leftarrow c_{(a,d)}^S$; $c_{(a,d)}^S \leftarrow c_{(a,d)}^S + 1$;
20: $\mathcal{M} \leftarrow \text{SCANPARTICIPANT}(S, (a, d), c_{\text{old}}, c_{(a,d)}^S)$;
21: **for all** $e \in \mathcal{M}$ **do**
22: COLLECTANDDECIDE(e, Q);
23: **if** $A(f^*) \geq T(f^*)/2$ **then**
24: COLLECTANDDECIDE(f^*, Q);

Algorithm 14 UPDATEDESTROYEDPATHEDGE($e_i, p, (a, d), Q$)

Input: an edge e_i on a destroyed priority 3-path p under key (a, d) ; peel queue Q ;
1: remove e_i from $R_{(a,d)}^S[\cdot]$;
2: $A(e_i) \leftarrow A(e_i) + |S_{(a,d)}(e_i)| \cdot (c_{(a,d)}^S - \ell_{(a,d)}^S(e_i)) - \delta_{(a,d)}^S(e_i)$;
3: $\ell_{(a,d)}^S(e_i) \leftarrow c_{(a,d)}^S$; $\delta_{(a,d)}^S(e_i) \leftarrow 0$;
4: $A(e_i) \leftarrow A(e_i) + |B_{(a,d)}|$;
5: **if** $e_i \in E(B_{(a,d)})$ **then**
6: let $p_2(e_i)$ be the unique alive priority 2-path in $B_{(a,d)}$ that contains e_i ;
7: $\Sigma \leftarrow \text{Count-Invalid}(p_2(e_i), p)$;
8: **for all** $(h, \delta) \in \Sigma$ **do**
9: **if** $h \in E(p) \setminus \{e^*\}$ **then**
10: $A(h) \leftarrow A(h) - 1$;
11: **if** $h \in E(p_2(e_i))$ **then**
12: $\delta_{(a,d)}^B(h) \leftarrow \delta_{(a,d)}^B(h) + 1$;
13: $|S_{(a,d)}(e_i)| \leftarrow |S_{(a,d)}(e_i)| - 1$;
14: **if** $|S_{(a,d)}(e_i)| = 0$ **then**
15: delete (a, d) from $\mathcal{S}(e_i)$;
16: $E(S_{(a,d)}) \leftarrow E(S_{(a,d)}) \setminus \{e_i\}$;
17: **else if** $e_i \notin Q$ **then**
18: $\tilde{\lambda}_{(a,d)}(e_i) \leftarrow 2^{\lfloor \log_2 \left(\max \left\{ 1, \frac{T(e_i)}{|S_{(a,d)}(e_i)| \cdot |h(e_i)|} \right\} \right) \rfloor}$;
19: insert e_i into $R_{(a,d)}^S[\log_2 \tilde{\lambda}_{(a,d)}(e_i)]$;

effectiveness of the scores $S_p(v)$, we measure them by ROC-AUC and PR-AUC with respect to the ground truth labels.

Algorithm 15 PEELTHROUGHTHREEBLOOM(e^*, Q)

Input: peeled edge e^* ; peel queue Q ;

```

1: for all  $(a, d) \in S(e^*)$  do
2:    $M \leftarrow \emptyset$ ;
3:    $r \leftarrow |S_{(a,d)}(e^*)|$ ;
4:   for all destroyed priority 3-paths  $p \in S_{(a,d)}$  with  $e^* \in E(p)$  do
5:     let  $E(p) = \{e^*, e_1, e_2\}$ ;
6:     UPDATEDESTROYEDPATHEDGE( $e_1, p, (a, d), Q$ );
7:     UPDATEDESTROYEDPATHEDGE( $e_2, p, (a, d), Q$ );
8:      $M \leftarrow M \cup \{e_1, e_2\}$ ;
9:     delete  $p$  from  $S_{(a,d)}$ ;
10:   $c_{old} \leftarrow c_{(a,d)}^B$ ;  $c_{(a,d)}^B \leftarrow c_{(a,d)}^B + r$ ;
11:   $M \leftarrow M \cup \text{SCANPARTICIPANT}(B, (a, d), c_{old}, c_{(a,d)}^B)$ ;
12:   $|S_{(a,d)}| \leftarrow |S_{(a,d)}| - r$ ;
13:   $E(S_{(a,d)}) \leftarrow E(S_{(a,d)}) \setminus \{e^*\}$ ;
14:  for all  $e \in M$  do
15:    if  $e$  is not peeled and  $e \notin Q$  and  $A(e) \geq T(e)/2$  then
16:      COLLECTANDDECIDE( $e, Q$ );

```

Table 9: Helix-core detection at 6.5 Å.

PDB	V	E	C_3		C_4		C_5		C_6	
			ROC	PR	ROC	PR	ROC	PR	ROC	PR
1A6M	151	556	0.6444	0.7801	0.7359	0.8104	0.8416	0.8838	0.8124	0.8512
2CI2	65	202	0.8725	0.3500	0.9706	0.7000	0.9552	0.6829	0.9552	0.6829
5PTI	58	175	0.8370	0.2105	0.9130	0.3333	0.9783	0.6667	0.9674	0.5714
1YCC	107	351	0.9206	0.6429	0.9365	0.6923	0.9603	0.8057	0.9515	0.7853
2RN2	155	535	0.8200	0.4286	0.9180	0.6203	0.8987	0.6292	0.8526	0.5485
4ICB	76	245	0.7976	0.6186	0.8730	0.7161	0.8784	0.7507	0.8748	0.7599
3GB1	56	166	0.9048	0.5556	0.9643	0.7692	0.9464	0.6971	0.9464	0.6869
4CPV	108	361	0.8085	0.6170	0.9127	0.7885	0.9167	0.7904	0.9098	0.7874

We conducted experiments on eight proteins from Protein Data Bank (PDB) [1]. For each protein, we construct two protein graphs with respect to C_α -cutoff values: 6.5Å and 7.0Å. Thus, in total 16 protein graphs are obtained. For each protein graph, we compute the score $S_p(v)$ for each vertex v with respect to $p \in \{3, 4, 5, 6\}$.

The results are shown in Tables 9 and 10. Under the 6.5Å cutoff, C_5 -truss achieves the best PR-AUC on five out of the eight protein graphs. C_4 -truss performs best on 2CI2 and 3GB1, while C_6 -truss performs best on 4ICB. For ROC-AUC, C_5 -truss performs best on five graphs and C_4 -truss on the remaining three.

Under the 7.0Å cutoff, C_5 -truss also achieves the best PR-AUC on five protein graphs. C_4 -truss performs best on 5PTI and 4CPV, while C_6 -truss performs best on 1A6M. For ROC-AUC, C_5 -truss performs best on four graphs, C_4 -truss on three, and C_6 -truss on one. C_3 -truss does not achieve the best PR-AUC or ROC-AUC under either cutoff.

Overall, these results provide empirical evidence that C_4 -truss and C_5 -truss decompositions capture useful structural signals that are not fully reflected by C_3 - or C_6 -truss scores in Protein Helix-Core Detection.

I Future Work

Dynamic Graphs. Our algorithms are designed for static graphs. We note that maintaining C_p -truss numbers under graph updates remains a challenging and open problem. To see this, an edge update can possibly change the C_p -truss numbers of a large number

Table 10: Helix-core detection at 7.0 Å.

PDB	V	E	C_3		C_4		C_5		C_6	
			ROC	PR	ROC	PR	ROC	PR	ROC	PR
1A6M	151	594	0.6001	0.7577	0.7449	0.8137	0.8142	0.8491	0.8617	0.9161
2CI2	65	233	0.7157	0.1944	0.5980	0.1795	0.8025	0.3166	0.4790	0.1463
5PTI	58	211	0.6848	0.1212	0.8587	0.2353	0.8478	0.2222	0.8478	0.2222
1YCC	107	406	0.7434	0.3681	0.8704	0.6847	0.8541	0.7118	0.8302	0.6545
2RN2	155	632	0.5900	0.2755	0.6746	0.2973	0.7009	0.3636	0.6872	0.3615
4ICB	76	272	0.7359	0.5559	0.7913	0.6151	0.8711	0.7221	0.8176	0.6646
3GB1	56	203	0.7619	0.3333	0.9405	0.6667	0.9464	0.6971	0.4607	0.2114
4CPV	108	404	0.6915	0.5000	0.7726	0.5817	0.7080	0.5606	0.6853	0.5255

of edges, and thus, requires to “touch” each of these edges to update the truss numbers. Therefore, the update cost can be as high as $O(m)$ in the worst case. Nonetheless, this difficulty is not just for C_4 - or C_5 -Truss Decompositions, but also for all peeling-based decompositions, such as core decomposition [21] and bitruss decomposition [3]. Designing efficient dynamic algorithms would be an interesting direction for future work.

Parallelization. All our algorithms and competitors in our experiments are single-threaded and follow the peeling-process framework. These algorithms are not easy to be parallelized. This is because in the peeling process, the edges are peeled following a considerably strict sequential order, where some edge must be peeled before certain edges. Violations on such order would lead to incorrect results. Developing a new parallel algorithm for C_p -Truss Decomposition may require substantial new ideas and techniques to identify which edges could be peeled in parallel while still retaining correctness. This research direction deserves a dedicated future study.