Truss-based Community Search over Streaming Directed Graphs

Xuankun Liao[1], Qing Liu[2], Xin Huang[1], Jianliang Xu[1]
[1]Hong Kong Baptist University [2]Zhejiang University
{xkliao,xinhuang,xujl}@comp.hkbu.edu.hk qingliucs@zju.edu.cn

ABSTRACT

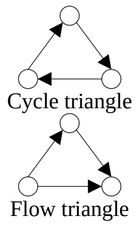
Community search aims to retrieve dense subgraphs that contain the query vertices. While many effective community models and algorithms have been proposed in the literature, none of them address the unique challenges posed by streaming graphs, where edges are continuously generated over time. In this paper, we investigate the problem of truss-based community search over streaming directed graphs. To address this problem, we first present a peeling-based algorithm that iteratively removes edges that do not meet the support constraints. To improve the efficiency of the peeling-based algorithm, we propose three optimizations that leverage the time information of the streaming graph and the structural information of trusses. As the peeling-based algorithm may suffer from inefficiency when the input peeling graph is large, we further propose a novel order-based algorithm that preserves the community by maintaining the deletion order of edges in the peeling algorithm. Extensive experimental results on real-world datasets show that our proposed algorithms outperform the baseline by up to two orders of magnitude in terms of throughput.

PVLDB Reference Format:
Xuankun Liao[1], Qing Liu[2], Xin Huang[1], Jianliang Xu[1]. Truss-based Community Search over Streaming Directed Graphs. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

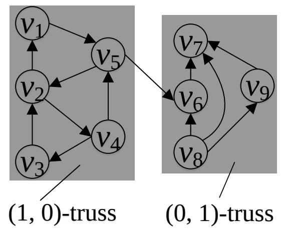
1 INTRODUCTION

In real-world applications, numerous relationships can be represented as directed graphs, such as online social networks, e-commerce networks, and financial networks. A common task in such graphs is to explore communities within them [2]. Recently, researchers have been investigating the problem of community search over directed graphs [20, 39]. The objective is to identify a dense subgraph that contains a specified set of query vertices. To achieve this, several dense subgraph models have been proposed, such as D-core [9, 19, 20] and D-truss [39, 44]. In this paper, we focus on the D-truss model, as it excels in discovering communities with distinctive structures. Specifically, the D-truss model, also denoted as (k_c, k_f)-truss, considers the cycle triangle and flow triangle (as shown in Figure 1(a)) and posits that each edge is within k_c cycle triangles and k_f flow triangles, respectively. Figure 1(b) shows two D-trusses with different structures. In the $(1, 0)$-truss, the vertices are strongly connected and all have an equal status within the community. On the other hand, the $(0, 1)$-truss exhibits

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX



(a) Triangles



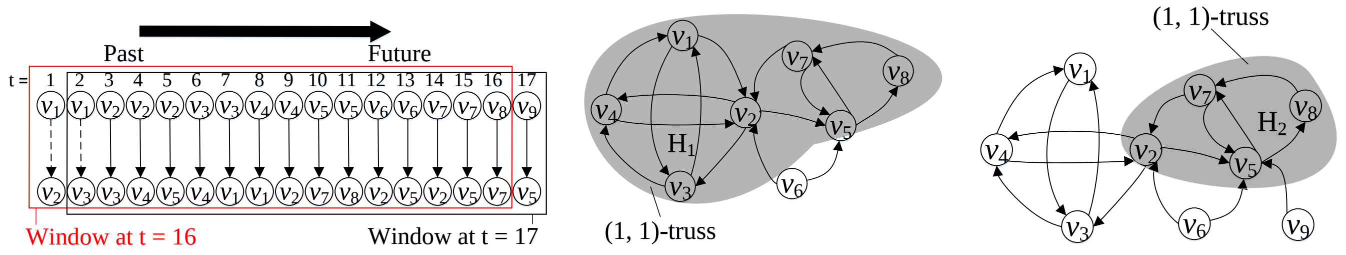
(b) D-trusses

Figure 1: Examples of Directed Graph and D-truss

weak connections among its vertices, with some vertices acting as authorities and leaders (e.g., v_7) while others functioning as hubs and followers (e.g., v_8). D-truss community search has numerous real-world applications. For instance, in social media applications like Twitter, vertices represent users, and a directed edge (u, v) indicates that user u follows user v. Suppose Figure 1(b) represents a social network from Twitter, then we can observe two communities with different properties based on the D-truss model. The left community consists of members who follow each other, demonstrating an equal relationship. In contrast, the right community exhibits a subordinate relationship, with v_7 being the leader and v_8 being the follower. The D-truss model can also be applied to analyze associative thesauri in word association networks [39]. In the Edinburgh Associative Thesaurus (EAT) network, English words are represented as vertices, and a directed edge from i to j indicates that word j comes to mind when word i is presented as a stimulus. D-truss can help us discover the words that are highly related to the query word and gain different insights [39]. For instance, when setting the query vertex as "drink" and using the (3, 0)-truss, words with equal relations to "drink" such as "glass", "wine", "water", and "bottle" would be included [39].

So far, the community search over directed graphs has focused primarily on static graphs. However, many applications involve continuously generated streaming graphs with an unbounded sequence of edges arriving at a high speed. For example, social media platforms like Twitter and Facebook can represent users as vertices and their interactions (such as retweets, comments, and votes) as edges, forming an unbounded sequence of edges over time. To address the dynamic nature of such data, this paper studies the problem of community search over streaming directed graphs, which has a wide range of applications in social networks, e-commerce networks, and financial networks, e.g.:

Example 1.1 (streaming social networks). *Community search in streaming social networks can track users' up-to-date communities for improved targeted advertising and content recommendation. By analyzing the most recent user interactions, we can identify communities with similar interests and preferences, allowing advertisers to target their ads and social networks to recommend more relevant content.*



(a) Streaming graph (b) Snapshot graph G (t = 16) (c) Snapshot graph G (t = 17)

Figure 2: A Streaming Graph

Example 1.2 (streaming financial networks). *Community search in streaming financial networks can help identify the communities of suspicious accounts by analyzing recent fund transfer transactions, enabling the timely detection of potential money laundering and fraud activities.*

However, due to the unbounded and high-velocity nature of graph streams, searching for communities over streaming directed graphs poses additional challenges. First, the sheer size of the stream makes it infeasible to retrieve a community from the entire stream. Therefore, an appropriate model for community search over streaming directed graphs should be employed. Second, as the edges arrive at a high rate, high-throughput algorithms are essential to handle incoming edges in a timely manner. Previous work has considered the problem of truss-based community search over dynamic graphs [44]. Even though both dynamic and streaming graphs consider the graph's updates over time, dynamic graphs only consider the insertion and deletion of conventional edges, while streaming graphs consist of an unbounded sequence of edges arriving at high speed. Consequently, previous index-query-based algorithms cannot satisfy the high-throughput constraint in our problem setting, and algorithms specialized for community search over streaming graphs are thus urgently needed.

To address the first issue, we resort to the sliding window model, which is widely used and forms the basis of many representative streaming graph analytics algorithms [21, 31, 41, 46]. Specifically, we use a sliding window approach to model recent edges and limit our consideration to communities within those edges to avoid high storage overhead, which provides more valuable and insightful information compared to out-of-date edges [12, 21]. For example, in Figure 2, let $k_c = k_f = 1$ and a query vertex set $Q = \{v_2\}$, we continuously monitor communities in a streaming graph using a sliding window of 16-edge duration. Figures 2(b) and 2(c) show the snapshot graphs at time points $t = 16$ and $t = 17$, respectively. These two snapshot graphs return two different communities of v_2, denoted by the shaded areas. To retrieve communities continuously from $t = 16$ to $t = 17$, a naive approach would perform D-truss community searches over two snapshot graphs at $t = 16$ and $t = 17$ from scratch, respectively. This approach is inefficient and computationally expensive. To overcome this limitation, we develop two efficient algorithms that can handle high-throughput edge updates: the peeling-based algorithm and the order-based algorithm.

Specifically, the peeling-based algorithm iteratively removes the edges without enough support to retrieve the communities based on three optimizations that significantly reduce the search space, namely *OPT1-upper bounds based pruning*, *OPT2-BFS-based incremental update*, and *OPT3-lifetime prediction*. The first optimization devises upper bounds c_m and f_m of cycle and flow trussnesses for an edge e. Based on c_m and f_m, we can directly prune all disqualified edges whose cycle and flow upper bounds are less than the input parameters k_c and k_f. The second optimization is to shrink the peeling graph to subgraphs, in which edges are *triangle connected* to the new coming edges, and their cycle and flow support are bigger than k_c and k_f, respectively. The third optimization utilizes the edges' arrival time and the window size to predict its *lifetime support*, which tracks the support of edges in a series of future windows over incoming streams. Then, we utilize the lifetime support to prune the unqualified edges directly, thereby improving efficiency.

Furthermore, we propose an order-based algorithm that resorts to an auxiliary structure called *community retrieving order*. Specifically, the community retrieving order records the sequence of edge removal during the D-truss peeling process, and is divided into different layers. Based on the community retrieving order, we can easily get the D-truss without peeling the graph from the scratch. When the window slides, we only need to maintain the community retrieving order to update the D-truss community.

We summarize the main contributions of this work as follows:

- We study a novel problem of D-truss community search over streaming graphs, which continuously finds query-dependent D-truss communities within a sliding window.
- We propose a peeling-based algorithm to answer the D-truss community search over streaming graphs, which incorporating three optimizations to reduce the graph for peeling.
- We introduce the *community retrieving order* to preserve the peeling order of edges, based on which an efficient order-based algorithm is designed to handle the D-truss community search over streaming graphs
- Both theoretical analysis and extensive empirical studies demonstrate the effectiveness and efficiency of our proposed model and algorithms. Our proposed solutions outperform the baseline by several orders of magnitude.

The rest of this paper is organized as follows. Section 2 reviews related works. Section 3 presents the formal definition of the studied problem. Section 4 and 5 propose the peeling-based algorithm and the order-based algorithm, respectively. Experimental results are reported in Section 6. Finally, Section 7 concludes the paper.

2 RELATED WORK

In this section, we review the related work from two aspects, i.e., *community detection over streaming graphs* and *community search.*

Community Detection over Streaming Graphs. Community detection over streaming graphs has been a subject of significant research interest over the years, as it facilitates the understanding of community structures in graph streams. Wang et al. [45] introduced a novel local pattern structure called local weighted-edge-based pattern summary and developed efficient algorithms to tackle the problem of dynamic community detection in locally heterogeneous weighted graph streams. Ding et al. [14] devised a pruning-based graph stream community detection algorithm, which identifies unimportant nodes based on their degree of centrality in graph streams. Hollocou et al. [24] investigated an edge streaming setting and proposed a method to construct communities by detecting local changes at each edge arrival. Liakos et al. [38] employed seed-set expansion approaches to identify communities over a graph stream, which is designed to use space sublinear to the number of edges and does not impose any restrictions on the order of the edges in the stream. Note that the community detection over streaming graphs aims to identify all communities, whereas the objective of our work is to return the community of query vertices over streaming graphs.

Community Search. The concept of community search was first introduced in [43]. Subsequently, numerous efforts have been devoted to exploring community search based on various models [17, 29], including k-core, k-truss, k-clique, k-edge connected component (k-ECC), and so on. To be specific, the k-core model requires every vertex in a community to have at least k neighbors [3, 4, 11, 43]. The k-truss model requires every edge in a community to be contained within at least $k-2$ triangles [1, 27, 30]. The clique model ensures that any two vertices in a community are connected to each other [10, 48]. On the other hand, the k-ECC-based community search defines a community as a steiner maximum-connected subgraph [6, 25, 26]. Besides simple graphs, community search has also been widely studied for complex graphs, such as directed graphs [9, 19, 39], temporal graphs [37], weighted graphs [8, 22, 34–36], attributed graphs [15, 28, 40], and spatial graphs [7, 16, 18]. More recently, an indexing method has been proposed for maximal D-truss searches over dynamic directed graphs [44]. However, [44] requires maintaining the skyline trussness for fully dynamic D-truss queries, which is not suitable for the streaming scenario. Despite these extensive studies, no previous work has explored the problem of community search in a streaming scenario, which motivates us to explore community search over streaming directed graphs.

3 PROBLEM FORMULATION

We consider a directed, unweighted simple graph, denoted as $G = (V_G, E_G)$. For brevity, we refer to a directed graph as a `digraph`. Each directed edge $e = (u, v) \in E_G$ represents a connection from vertex u to vertex v. If the edge (u, v) exists, u is an in-neighbor of v and v is an out-neighbor of u. For a vertex v, we denote all of its in-neighbors and out-neighbors in G by $N^+_G(v) = \{u : (u, v) \in E_G\}$ and $N^-_G(v) = \{u : (v, u) \in E_G\}$, respectively. The neighbors of vertex v is defined as $N_G(v) = N^+_G(v) \cup N^-_G(v)$. Based on the above concepts, we introduce the streaming digraph model used in this paper as follows.

Definition 3.1. ***Streaming Digraph.*** *A streaming digraph is a continually growing sequence of items denoted as $S = \langle e_1, e_2, e_3, \ldots \rangle$, where each item $e_i = \langle (u, v), t_i \rangle$ signifies that a directed edge from vertex u to vertex v arrives at time point t_i, where $t_i < t_j$ for $i < j$.*

Due to the ever-increasing volume of the streaming graph, we focus on the most recent edges using the *time-based sliding window* model [42].

Definition 3.2. ***Time-based Window.*** *A time-based window W with a length of τ contains edges with timestamps within the interval $(t - \tau, t]$, where t is the current clock time of the system. The time-based window is denoted by W^t.*

Definition 3.3. ***Time-based Sliding Window.*** *A time-based sliding window W with a slide interval of β is a time-based window that slides every β time units. The slide interval β is also referred to as the* stride *[32, 41, 47].*

Definition 3.4. ***Snapshot Digraph.*** *A snapshot digraph G^t at time point t is a digraph induced by all the edges in the time-based window W^t.*

In this paper, we focus on an *append-only* steaming digraph, where edge expiration occurs only when the time-based window slides [41]. Figure 2(a) illustrates an example of a streaming digraph, where $\tau = 16$ and $\beta = 1$. Figure 2(b) and Figure 2(c) depict the snapshot digraph at time point 16 ant time point 17, respectively. When the context is clear, we refer to the time-based window simply as "window" and omit t for G^t.

Next, we introduce the D-truss model for community search.

Definition 3.5. ***Cycle Support, Flow Support [39].*** *Given a directed graph G and an edge e, the* cycle support *of e in G , denoted by $csup_G(e)$, represents the number of vertices that can form cycle triangles with e in G. The* flow support *of e in G, denoted by $fsup_G(e)$, denotes the number of vertices that can form flow triangles with e in G.*

We call the vertices that form cycle triangles with e the cycle neighbors of e, and the vertices that form flow triangles as the flow neighbors. We also denote cycle triangle as $(\triangle^C_{v_1 v_2 v_3})$ and flow trianle as $(\triangle^F_{v_1 v_2 v_3})$. Given the definitions of cycle support and flow support, we now introduce the definition of D-truss.

Definition 3.6. ***D-truss [39].*** *Given a directed graph G and two integers k_c and k_f, a subgraph $H \in G$ is a D-truss, also denoted as (k_c, k_f)-truss, if $\forall e \in E_H$, $csup_G(e) \geq k_c$ and $fsup_G(e) \geq k_f$.*

A D-truss H is considered a maximal D-truss if there does not exist any other D-truss $H' \subseteq G$ such that $H' \supset H$.

Problem 1. *Given a streaming graph G, a sliding window with length τ, a stride β, two integers k_c and k_f, and a set of query vertices Q, the D-truss community search over the streaming digraph is to continuously return the maximal D-truss that contains Q in the snapshot digraph G^t at time point t, where $t = i \cdot \beta$ and $i \in \mathbb{N}$.*

Example 3.1. *Consider the streaming digraph in Figure 2(a). Let $k_c = k_f = 1, Q = v_2, \tau = 16,$ and $\beta = 1$. At time point 16, the snapshot digraph is Figure 2(b). The D-truss community is H_1. At time point 17, the window slides, and the corresponding snapshot digraph is Figure 2(c). Then, the D-truss community is updated to H_2.*

4 PEELING-BASED ALGORITHM

In this section, we propose a peeling-based algorithm to handle D-truss community search over streaming-directed graphs. We first present a basic algorithm and then propose several optimizations to significantly improve the algorithm performance by reducing the search space.

4.1 Basic Peeling-based Algorithm

Given a directed graph, a straightforward way to retrieve the D-truss community is to iteratively delete the edges whose cycle support and flow support are smaller than k_c and k_f, respectively. The pseudo-code of the above D-truss community search method is outlined in Algorithm 1. Specifically, it first computes the cycle and flow supports for each edge and collects the edges whose cycle support and flow support are smaller than k_c and k_f, respectively (lines 2-5). These unqualified edges are then deleted from the graph (lines 7-8), and the supports of their neighboring edges are updated correspondingly. These adjacent edges with support smaller than k_c and k_f will be removed in the subsequent edge deletion (lines 10-14). The deletion continues until all edges satisfy the support constraints, and the remaining graph is returned as the D-truss community. Based on Algorithm 1, a basic method, termed as Peeling-based Algorithm, for D-truss community search over streaming directed graphs is as follows: Whenever the sliding window moves, we employ Algorithm 1 to peel the updated snapshot graph to obtain the community.

Theorem 4.1. ***(Time and Space Complexities)*** *Let G^t be the directed graph in the window W^t. The time and space complexities of the peeling-based algorithm for each window W^t are $O(m_t^{1.5})$ and $O(m_t)$, respectively, where m_t denotes the number of edges in G^t.*

Proof. The support for each edge can be computed in $O(m_t^{1.5})$ by utilizing the triangle listing algorithm in [33]. The edge deletion and support maintenance take $O(m_t^{1.5})$ time. So the total time complexity is $O(m_t^{1.5})$. The space complexity is $O(m_t)$. □

The peeling algorithm works when handling static graphs. However, it is not well-suited to the high-speed update volume of streaming graphs and may not be able to update the community in real-time. Therefore, we propose several optimizations to improve its performance by reducing the search space of the peeling-based algorithm.

4.2 OPT-1: *Upper-bounds-based Pruning*

In the first optimization termed as upper-bounds-based pruning, we propose novel notations $c_m(e)$ and $f_m(e)$ for an edge e, based on which we shrink the directed graph for peeling. We start by giving the formal definition of $c_m(e)$ and $f_m(e)$.

Definition 4.1. *Given a directed edge $e = (u, v)$ in digraph G, $c_m(e)$ is the maximal integer k_c^m such that there is a $(k_c^m, 0)$-truss $\subseteq G$ containing e. Similarly, $f_m(e)$ is the maximal integer k_f^m such that there is a $(0, k_f^m)$-truss $\subseteq G$ containing e.*

In other words, $c_m(e)/f_m(e)$ of an edge e only considers the cycle/flow triangles that contain e. If e is in a (k_c, k_f)-truss, it holds

Algorithm 1: D-truss Peeling Algorithm

```
Input: a digraph $G = (V_G, E_G)$, two non-negative integers
       $k_c$ and $k_f$, query vertices $Q$
Output: $(k_c, k_f)$-truss containing $Q$
1  Let $L_e$ be an empty queue;
2  for $e_i \in E_G$ do
3      compute $csup_G(e_i)$ and $fsup_G(e_i)$;
4      if $csup_G(e_i) < k_c$ or $fsup_G(e_i) < k_f$ then
5          $L_e \leftarrow L_e \cup \{e_i\}$;
6  while $L_e \neq \emptyset$ do
7      Pop out an edge $e_i = \langle u, v \rangle$ from $L_e$;
8      Delete $e_i$ from $G$;
9      $N(u) \leftarrow N_G^+(u) \cup N_G^-(u)$; $N(v) \leftarrow N_G^+(v) \cup N_G^-(v)$;
10     for each vertex $w \in N(u) \cap N(v)$ do
11         for $e' \in \{\langle u, w \rangle, \langle v, w \rangle, \langle w, u \rangle, \langle w, v \rangle\} \cap E_G$ do
12             Update $csup_G(e')$ and $fsup_G(e')$ accordingly;
13             if $csup_G(e') < k_c$ or $fsup_G(e') < k_f$ then
14                 $L_e \leftarrow L_e \cup \{e'\}$;
15 if $Q \subseteq V_G$ then
16     Return $G$;
17 Return $\emptyset$;
```

that $k_c \leq c_m(e)$ and $k_f \leq f_m(e)$. Next, we introduce how to employ $c_m(e)/f_m(e)$ to shrink the input graph for peeling.

Theorem 4.2. *Given two integers k_c and k_f, and an edge e, if $k_c > c_m(e)$ or $k_f > f_m(e)$, e cannot be in a (k_c, k_f)-truss.*

Based on Theorem 4.2, if we prune the edges whose c_m and f_m are less than the parameters k_c and k_f respectively, we can reduce the peeling input graph significantly while not affecting the correctness of query results. However, computing exact c_m and f_m for an edge is time-consuming. To tackle this issue, we resort to the H-index method to estimate $c_m(e)$ and $f_m(e)$ in an efficient way. To be specific, the H-index of an integer set S is the maximum integer h such that there are at least h integer elements in S whose values are no less than h [23]. For example, if $S = \{2, 3, 4, 4, 5, 6\}$, the H-index of S is 4 since there are four integers in S whose values are no less than 4. Based on H-index, we introduce the *cycle-h-index* and *flow-h-index* for an edge.

Definition 4.2. ***Cycle-H-Index, Flow-H-Index***. *The cycle-h-index of an edge $e = (u, v)$ in a digraph G is the H-index of an integer set S, where $S = \{\min(csup_G(e'), csup_G(e'')) \colon \forall e', e''$ that can form cycle triangles with $e\}$. The flow-h-index of an edge $e = (u, v)$ in a digraph G is the H-index of an integer set S', where $S' = \{\min(fsup_G(e'), fsup_G(e'')) \colon \forall e', e''$ that can form flow triangles with $e\}$.*

It is worth noting that given a vertex w and an edge $e = (u, v)$, w, u, and v can form multiple flow triangles. We only choose the smallest $min(fsup_G(e'), fsup_G(e''))$ among all flow triangles formed by w, u, and v. The pair of cycle-h-index and $c_m(e)$ and the pair of flow-h-index and $f_m(e)$ have the following relationships.

Theorem 4.3. *Given an edge e, the cycle-h-index and flow-h-index of e are no smaller than $c_m(e)$ and $f_m(e)$, respectively.*

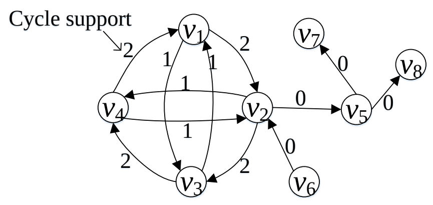


Figure 3: An illustrative example of OPT-1 on a directed graph, where edges are associated with cycle supports.

Proof. The proof to Theorem 4.3 can be proved by contradiction. Given an edge e, we first assume its cycle-h-index, say c, is smaller than $c_m(e)$. However, by the definition of $c_m(e)$, we can find at least c_m vertices that can form cycle triangles with e, and each edge in these triangles also has a cycle support no less than c_m. Based on these c_m vertices, we can get the cycle-h-index of e is at least c_m, which contradicts our assumption before. Hence, Theorem 4.3 holds. □

Theorem 4.4. *Given two integers k_c and k_f, and an edge e, if the cycle-h-index of e is smaller than k_c, or the flow-h-index of e is smaller than k_f, e cannot be in a (k_c, k_f)-truss.*

Proof. Based on Theorem 4.3, the cycle-h-index of an edge e is a upper-bound of $c_m(e)$. Hence, Theorem 4.4 holds based on Theorem 4.2. □

Theorem 4.4 indicates that if the cycle-h-index of e is smaller than k_c, or the flow-h-index of e is smaller than k_f, e cannot be in the final results and can be safely pruned. Based on Theorem 4.4, the basic idea of our first optimization is as follows. When the window slides to get a new snapshot graph, we first compute the cycle-h-index and flow-h-index for each edge. Then, we delete the edges using Theorem 4.4. Finally, the remaining graph is taken as the input graph for peeling in Algorithm 1.

Example 4.1. *Suppose we want to retrieve the $(2, 0)$-truss from the digraph in Figure 3, along with the corresponding cycle support for each edge. To achieve this, we calculate the* cycle-h-index *value for each edge. Let's consider the example of (v_4, v_1). We start by initializing a 2-sized set, as this edge has two cycle neighbors: v_2 and v_3. For the cycle triangle $\triangle^C_{v_1 v_2 v_4}$, we select the smaller value between (v_1, v_2) and (v_2, v_4), which is 1, and store it in the set. Similarly, we obtain a value of 1 for the other cycle triangle. The estimated cycle-h-index value of (v_4, v_1) is then the H-index value of the set $\{1, 1\}$, which is 1. We can calculate the cycle-h-index values for all edges using the same approach. Since all of them are smaller than 2, we can directly output an empty set as the retrieved community without deleting any edges, which improves efficiency.*

Theorem 4.5. ***(Time and Space Complexities)*** *Let G^t be the directed graph in the window W^t. If we denote the edge set of G^t as E, the time and space complexities of calculating the cycle-h-index and flow-h-index for each window W^t are $O(\sum_{(u,v)\in E}(|N(u)| + |N(v)|))$ and $O(m_t)$, respectively.*

Proof. For an edge $e = (u, v)$, the calculation of its cycle-h-index or flow-h-index takes $O(csup(e))$ / $O(fsup(e))$ time the the collection of the corresponding set takes $O(|N(u)| + |N(v)|)$ time. Hence, the calculation for all the edges in G^T is bounded by $O(\sum_{(u,v)\in E}(|N(u)| + |N(v)|))$, The space complexity is bounded by $O(m_t)$. □

4.3 OPT-2: *BFS-based Incremental Update*

While the estimated values of $c_m(e)$ and $f_m(e)$ can prune some disqualified edges, the resulting directed graph may still be very large. Consequently, the pruning efficiency may be limited. To address this issue, we introduce OPT-2, which employs a BFS-based shrink-and-expand search technique. The main idea is to collect the peeling input using BFS. We discuss OPT-2 for two cases, i.e., edge deletion and edge insertion.

For the edge deletion case, we use an eviction strategy to remove edges from the current community. If a deleted edge is part of the community, we evict it and check all its adjacent edges. If any of these edges fail to satisfy the support constraints of the community, we remove them and repeat the eviction process until all edges in the community satisfy the support constraints. Algorithm 2 shows the pseudo-code for the edge deletion case. The snapshot graph is first updated (lines 1-2). If the deleted edges are in the original community, they will be collected as the eviction seeds (lines 6-8) and be evicted in the while loop (line 11). The neighboring edges that can form triangles with the evicted edge are then evaluated to determine if they have enough cycle support and flow support in the updated community (lines 12-22). The edges that do not meet these criteria are added to a queue for subsequent eviction (lines 23-24).

For the edge insertion case, instead of taking all edges into the peeling process, we take the inserted edges as seeds and collect all edges that have enough support and are reachable to the new edges with a set of triangles. Then, we peel the collected edges to get the community containing the newly inserted edges. Finally, the newly peeled community is combined with the old community to get the final community. Algorithm 3 outlines the complete algorithm for the edge insertion case. It first updates the snapshot graph with newly inserted edges (lines 1-5). Then, Algorithm 3 identifies new edges that have sufficient support and treats them as seeds (lines 7-9). Next, Algorithm 3 gets the graph S induced by the edges that (1) satisfy $csup_G(e) \geq k_c$ and $fsup_G(e) \geq k_f$ conditions, and (2) are reachable from the newly inserted edge via BFS (lines 10-16). Iteratively, the algorithm continues to delete the edges whose $csup_G$ or $fsup_G$ values are lower than k_c or k_f, respectively (line 17).

Example 4.2. *Consider Figures 2(b) and 2(c), where we have $k_c = 1$, $k_f = 1$, $\tau = 16$, and $\beta = 1$. In the case of edge insertions, Algorithm 3 does not collect any edges when (v_9, v_5) is inserted at $t = 17$, since it fails to meet the support constraints based on the existing edges. However, for the expiring edge (v_1, v_2), Algorithm 2 collects its neighboring edges (v_1, v_3), (v_3, v_1), (v_4, v_2), and (v_2, v_4) as they no longer satisfy the support requirements after the deletion. This, in turn, triggers the eviction of edges (v_3, v_4), (v_4, v_1), and (v_2, v_3). Ultimately, the community that satisfies the support constraints is updated as H_2.*

Theorem 4.6. ***(Time and Space Complexities)*** *Let G^t be the directed graph in the window W^t, m_t be the number of edges in G^t, δ be the maximal cycle-support/flow-support of an edge in G^t, C be the candidate area where each edge in it satisfies $csup_{G^t}(e) > k_c$ and $fsup_{G^t}(e) > k_f$, then for each window W^t, Algorithms 3 and 2 take $O(C^{1.5} + \sum_{(u,v)\in E_{G^t}}(|N(u)| + |N(v)|)$ time and $O(\delta \cdot$*

Algorithm 2: Optimized Peeling Algorithm for Edge Deletion with OPT-2

```
Input: a snapshot graph $G = (V_G, E_G)$, a batch of deleted
       edge $\mathcal{E}_d$, original D-truss $DT$, parameters $k_c$ and $k_f$,
       query vertices $Q$
Output: updated $(k_c, k_f)$-truss containing $Q$
1  for $e_i \in \mathcal{E}_d$ do
2      $G \leftarrow G \setminus \{e_i\}$;
3  if all edges in $\mathcal{E}_d$ are not in $DT$ then
4      return
5  let queue $Q \leftarrow \emptyset$;
6  for $e_i \in \mathcal{E}_d$ do
7      if $DT[e_i] = true$ then
8          $Q \leftarrow Q \cup \{e_i\}$;
9  while $Q \neq \emptyset$ do
10     Pop an edge $e'$ from $Q$;
11     $DT[e'] \leftarrow false$;
12     for $(e_0, e_1)$ that can form a triangle with $e'$ do
13         if $DT[e_0] = false$ or $DT[e_1] = false$ then
14             continue;
15         if $e_0 \notin Q$ then
16             $csup(e_0) \leftarrow 0, fsup(e_0) \leftarrow 0$;
17             for $(e_x, e_y)$ form a cycle triangle with $e_0$ do
18                 if $DT[e_x] = true$ and $DT[e_y] = true$ then
19                     $csup(e_0) \leftarrow csup(e_0) + 1$;
20             for $(e_u, e_v)$ form a flow triangle with $e_0$ do
21                 if $DT[e_u] = true$ and $DT[e_v] = true$ then
22                     $fsup(e_0) \leftarrow fsup(e_0) + 1$;
23             if $csup(e_0) < k_c$ or $fsup(e_0) < k_f$ then
24                 $Q \leftarrow Q \cup e_0$
25         Perform the same operations for $e_1$ as lines 14-25;
26 if $Q \subset DT$ then
27     Return $DT$;
28 Return $\emptyset$;
```

$\sum_{(u,v)\in E_{Gt}}(|N(u)| + |N(v)|)$ *time, respectively. In addition, both of them take* $O(m_t)$ *space.*

Proof. The collection of the peeling input, i.e., S, takes $O(\sum_{(u,v)\in E_{Gt}}(|N(u)|+|N(v)|))$ time, and the peeling process takes $O(C^{1.5})$ time. So, the total time complexity of Algorithm 3 is $O(C^{1.5} + \sum_{(u,v)\in E_{Gt}}(|N(u)| + |N(v)|))$. The running time overhead of Algorithm 2 is dominated by the while loop, i.e., line 10, which is bounded by $O(\delta \cdot \sum_{(u,v)\in E_{GT}}(|N(u)| + |N(v)|)$ and takes $O(m_t)$ space. □

4.4 OPT-3: *Life-time Prediction*

The initial two optimizations overlook the temporal aspect of edges in streaming-directed graphs. Yet, our problem permits us to infer edge expiration times within the window based on parameters like window size and arrival times. This enables us to forecast community structure in future windows by only considering edges in the current window. However, there is a dilemma in designing a prediction-based algorithm, i.e., we mainly aim to pre-compute the community structure to eliminate the impact of edge deletions while cannot afford to store too many temporary results. To tackle this issue, we present our third optimization, which anticipates the involvement of current edges in upcoming windows to avoid processing for edge deletion. For instance, in Figure 4, given the

Algorithm 3: Peeling Algorithm for Edge Insertion with OPT-2

```
Input: a snapshot graph $G = (V_G, E_G)$, inserted edges $\mathcal{E}_i$,
       querying parameters $k_c$ and $k_f$, original community
       $DT$, query vertices $Q$
Output: updated $(k_c, k_f)$-Truss containing $Q$
1  for $e_i \in \mathcal{E}_i$ do
2      if $\exists\, e_j \in E_G = e_i$ then
3          $e_j.time \leftarrow e_i.time$;
4      else
5          $G \leftarrow G \cup \{e_i\}$;
6  Let queue $Q \leftarrow \emptyset, S \leftarrow \emptyset$;
7  for $e_i \in \mathcal{E}_i$ do
8      if $csup_G(e_i) \geq k_c$ and $fsup_G(e_i) \geq k_f$ then
9          $Q \leftarrow Q \cup \{e_i\}$;
10 while $Q \neq \emptyset$ do
11     Pop an edge $e'$ from $Q$;
12     if $csup_G(e') \geq k_c$ and $fsup_G(e') \geq k_f$ then
13         $S \leftarrow S \cup \{e'\}$;
14     for $e_j$ that can form a triangle with $e'$ do
15         if $e_j \notin Q$ and $e_j \notin S$ then
16             $Q \leftarrow Q \cup \{e'\}$;
17 $DT' \leftarrow$ Peeling$(S, k_c, k_f, \emptyset)$;
18 $DT \leftarrow DT \cup DT'$;
19 if $Q \subset DT$ then
20     Return $DT$;
21 Return $\emptyset$;
```

snapshot graph G at $t = 16$ with two incoming edges (v_3, v_2) and (v_2, v_1), which are marked in red in Figure 4(b). Three different communities can be predicted in Figures 4(c), 4(d), and 4(e), for different time points, respectively. By leveraging this prediction, we can eliminate the effect of edge expiration and focus solely on edge insertion. For instance, to update the community in Figure 4(b), we only need to insert edges (v_3, v_2) and (v_2, v_1) to predicted view 1 in Figure 4(c) and perform the incremental maintenance. Next, we present the details of optimization 3. We begin by introducing the concept of the *lifetime* of an edge.

Definition 4.3. ***Lifetime****. Given a sliding window size τ and a stride β, the lifetime of an edge $e = \langle (u, v), t_i \rangle$ is a vector that has a length of $\lceil \frac{\tau}{\beta} \rceil$. Each item in the vector corresponds to the cycle-support and flow-support of a specific time. The leftmost item has the time label equaling the time label of this edge as t_i, the time label of the second item is $t_i + 1 \cdot \beta$, the next one is $t_i + 2 \cdot \beta$, and so on.*

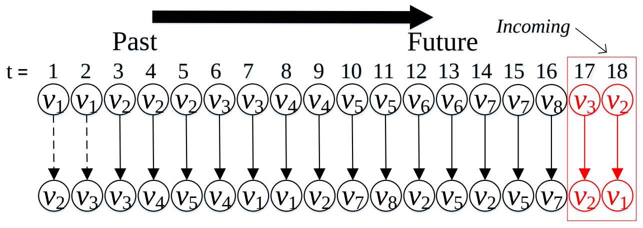
As time elapses, the items in the lifetime vector of an edge expire one by one. Figure 4(f) shows the lifetime support update of edge $\langle (v_4, v_1), 8 \rangle$ in Figure 4(b) with the insertion of (v_2, v_1) and (v_3, v_2). The first and second rows show the lifetime support before and after the update with each item in the form of *(cycle-support, flow-support)*, respectively. The third row displays the time labels for the predicted window. We can see that at time point 16, the items in the lifetime vector of (v_4, v_1), whose time label is smaller than 16, are expired.

Algorithm 4: Peeling Algorithm with OPT-3

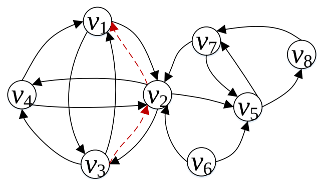
```
Input: a snapshot graph $G$, current window $W^i$, inserted
       edges $\mathcal{E}$, parameters $k_c$ and $k_f$, query vertices $Q$
Output: updated $(k_c, k_f)$-truss containing $Q$
1  Let $S \leftarrow \emptyset$;
2  for $e_i \in \mathcal{E}$ do
3      initialize $e_i.lifetime$;
4  $G \leftarrow G \cup \mathcal{E}$;
5  for $e_i \in \mathcal{E}$ do
6      for $(e_0, e_1)$ forms cycle triangles with $e_i$ do
7          $csup(e_0) \leftarrow csup(e_0) + 1$, $csup(e_1) \leftarrow csup(e_1) + 1$;
8          update the items in $lifetime$ for $e_0$, $e_1$ and $e_i$;
9      for $(e_0, e_1)$ forms flow triangles with $e_i$ do
10         $fsup(e_0) \leftarrow fsup(e_0) + 1$, $fsup(e_1) \leftarrow fsup(e_1) + 1$;
11         update the items in $lifetime$ for $e_0$, $e_1$ and $e_i$;
12 for $e_i \in E_G$ do
13     $j \leftarrow \lceil \frac{\tau}{\beta} \rceil - \lceil \frac{t(e_i) - W_s^i}{\beta} \rceil + 1$ ;
14     if $e_i.lifetime[j].first \geq k_c$ and
         $e_i.lifetime[j].second \geq k_f$ then
15         $S \leftarrow S \cup \{e_i\}$;
16 $DT \leftarrow$ Peeling($S, k_c, k_f, Q$);
17 Return $DT$;
```

By tracking the lifetime for each edge, we can predict the cycle support and flow support for a series of future windows. The candidate area G_a, which consists of the edges with $csup_{G_a} \geq k_c$ and $fsup_{G_a} \geq k_f$, can be easily derived based on edge' lifetime by incorporating edges with corresponding items in their lifetime larger than k_c and k_f. Besides, since no updates are required for handling expired edges, the efficiency can be improved. The pseud-code of OPT-3 is outlined in Algorithm 4. We assume that the rightmost item in $e_0.lifetime$ corresponds to the latest time window that e_0 can survive in and the leftmost is the earliest one. Given a set of newly inserted edges $\mathcal{E}$, we first initialize the lifetime array for each edge e_i with length $\lceil \frac{\tau}{\beta} \rceil$ and all values being two zeros (lines 2-3), representing their cycle support and flow support in the predicted window respectively. Then, for all newly inserted edges and the edges that form tangles with them, we update their supports and lifetime (lines 5-11). The peeling input is decided based on edges' lifetime (lines 12 - 15). The principle for maintaining the items in the edges' lifetime is that the future windows' predicted view only includes edges present in the current window and still exist at the future time point. Specifically, considering a window size τ_i, when examining a future window with time point t_i, only edges with a timestamp $t(e) + \tau_i > t_i$ are taken into account. For instance, let's consider the predicted view of the edge (v_4, v_1) with time point 18, as illustrated in Figure 4(c). In this case, edges (v_1, v_2) and (v_1, v_3) are excluded from the view. This is because their timestamps, $t(v_1, v_2)$ = 1+16 = 17 and $t(v_1, v_3)$ = 2+16 = 18, respectively, are both less than or equal to 18. Therefore, they are not considered in the predicted view at time 18. Whenever the window slides a stride, the edges' lifetime is updated by incorporating the new edges.

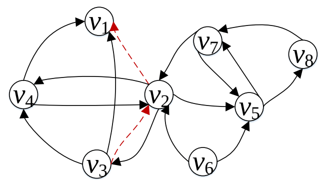
Example 4.3. *Figure 4 illustrates the lifetime support-based algorithm. In Figure 4(a), we have a streaming graph, and Figure 4(b) shows the snapshot graph taken at time 16 with a window length of 16.*


Past
Future
Incoming
t = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
v1 v1 v2 v2 v2 v3 v3 v4 v4 v5 v5 v6 v6 v7 v7 v8 v3 v2
v2 v3 v3 v4 v5 v4 v1 v1 v2 v7 v8 v2 v5 v2 v5 v7 v2 v1

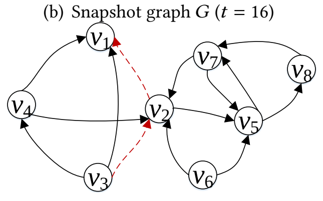
(a) Streaming graph



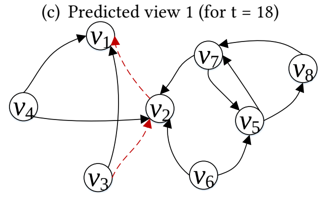
(b) Snapshot graph G (t = 16)



(c) Predicted view 1 (for t = 18)



(d) Predicted view 2 (for t = 20)



(e) Predicted view 3 (for t = 22)

▨	▨	▨	▨	(2, 2)	(0, 1)	(0, 1)	(0, 0)	Old
▨	▨	▨	▨	(2, 2)	(0, 2)	(0, 2)	(0, 1)	New
8	10	12	14	16	18	20	22	t

(f) Life support update of (v_4, v_1)

Figure 4: An illustrative example of *Lifetime Support-Based* Algorithm. Here, the window size τ = 16 and the stride β = 2.

The incoming edges (v_2, v_1) and (v_3, v_2) are marked in red. The stride length is set to 2. Figures 4(c), 4(d), and 4(e) display the predictions for the snapshots at time 18, 20, and 22, respectively. The predicted view 1 does not consider edges (v_1, v_2) and (v_1, v_3) at time 18, as they have expired at that time. In Figure 4(f), the lifetime support update for (v_4, v_1) is shown. The first and second rows depict the lifetime support before and after the update, respectively. The third row displays the time labels for the predicted window. The tuples in the first row are calculated based on the predicted view without the red edges. After considering the incoming red edges, the tuples are updated accordingly, as shown in the second row. For example, at t = 20, the tuple changes from (0, 1) to (0, 2) as it is now contained in another triangle $\triangle^F_{v_2 v_1 v_4}$. The support in the current snapshot graph at time 16 is taken as the corresponding item in the second row.

Theorem 4.7. ***(Time and Space Complexities)*** *Let G^t be the directed graph in the window W^t, δ be the maximal cycle support/flow support among edges in $\mathcal{E}$, C be the candidate area, where all edges have $csup_{G^t} \geq k_c$ and $fsup_{G^t} \geq k_f$, the time and space complexities of Algorithm 4 for each window W^t are $O(|\mathcal{E}| \cdot \delta \cdot \lceil \frac{\tau}{\beta} \rceil + C^{1.5})$ and $O(\lceil \frac{\tau}{\beta} \rceil \cdot m_t)$, respectively, where m_t is the number of edges in G^t.*

Proof. The update of lifetime items takes $O(|\mathcal{E}| \cdot \delta \cdot \lceil \frac{\tau}{\beta} \rceil)$ time, and the peeling process takes $O(C^{1.5})$ time. So the total time complexity is $O(|\mathcal{E}| \cdot \delta \cdot \lceil \frac{\tau}{\beta} \rceil + C^{1.5})$. Since each edge takes $O(\lceil \frac{\tau}{\beta} \rceil)$ space to store their lifetime, the total space complexity is $O(\lceil \frac{\tau}{\beta} \rceil \cdot m_t)$. □

5 THE ORDER-BASED ALGORITHM

The peeling algorithm needs to iteratively delete the edges to obtain the community. While we have devised three optimizations to shrink the directed graph for peeling, the resulting graph may still be very large, which can impact the speed of obtaining results. In this section, we propose an order-based D-truss community search algorithm. The rationale behind this is that a streaming graph can be viewed as a series of ordered edges based on the time information. If we can employ the property of D-truss to devise a rule to re-order the streaming graph, we can find the D-truss directly from the order rather than peeling the directed graph from scratch, which can improve efficiency.

5.1 D-truss Peeling Order and Layers

We first introduce the concept of *community retrieving order*.

Definition 5.1. ***Community Retrieving Order***. *The community retrieving order of G, denoted by $E_{\preceq} = (e_1, e_2, \ldots, e_{|E|})$, is identical to the order of edges' deletions in the D-truss peeling algorithm. That is, for any two edges e and e' of G, if e precedes e' in $E_{\preceq}$, denoted by $e \preceq e'$, e is deleted before e' in the peeling process.*

In essence, the community retrieving order $E_{\preceq}$ records the process of (k_c, k_f)-truss computation. Note that there may be multiple orders of edges' deletions in the peeling process, and any one of them can be used as the community retrieving order. For convenience, we use $E_{e_{\preceq}}$ to denote the set of edges appearing after e in $E_{\preceq}$, i.e., $E_{e_{\preceq}} = \{e' | e \preceq e'\}$.

It is worth mentioning that the community retrieving order can be generated by invoking the peeling algorithm. Our order is different from the *cycle decomposition order* proposed in previous work [44]. Specifically, their cycle decomposition order (i.e., CD order) is essentially the sequence of deleting edges along the cycle truss number of edges for a $(0, k_{f_i})$-truss. Their D-index consists of multiple CD orders for all possible k_{f_i}s and considers the two types of trussness separately during maintenance. However, in our problem setting, any edge failing to meet the support constraints in either the cycle support or flow support dimension will be removed, and a single-dimensional order that accounts for both cycle support and flow support dimensions simultaneously is considered. Nevertheless, it is non-trivial to retrieve the community structure with the order directly. Inspired by the fact that edges are usually peeled in batches during the peeling process, we present the concept of *layers in community retrieving order* and *layers number.*

Definition 5.2. ***Layers in community retrieving order.*** *Given a community retrieving order $E_{\preceq}$ for an digraph G, and the community query parameters k_c and k_f. Edges in E_G can be accommodated in different layers $\{L_1, L_2, \ldots, L_\mu\}$. The j-th layer is the set of edges satisfying $L_j = \{e \in E_G | \{csup_{H_j}(e) < k_c \textbf{or} fsup_{H_j}(e) < k_f\} \textbf{and} \{csup_{H_{j-1}}(e) \geq k_c \textbf{and} fsup_{H_{j-1}}(e) \geq k_f\}\}$, where $H_j = E_G \backslash U_{i=1}^{j-1} L_i$. The initial layer is $L_1 = \{e \in E_G : csup_G < k_c \textbf{or} sup_G < k_f\}$.*

Definition 5.3. ***Layer number $\mathcal{L}$.*** *Given an edge e, its layer number is defined as $\mathcal{L}(e) = \{l \in \mathbb{N}^+ : e \in L_l\}$. The layer number indicates the number of rounds in which e is peeled from the original graph during the community finding. For two edges e_i and e_j with $\mathcal{L}(e_i) < \mathcal{L}(e_j)$, we have $e_i \preceq e_j$.*

Based on the community retrieving order, the concept of layers provides a more coarse-grained way to record the sequence of the edge removal in the peeling algorithm, while still ensuring the correctness of the retrieved community. With the layers introduced, an order $E_{\preceq}$ can be represented as $\{L_1, L_2, \ldots L_\mu\}$. The edges in the same layer can have an arbitrary order, i.e., if e and e' are both in L_a for $\forall a : 1 \leq a \leq \mu$, then both $e \preceq e'$ and $e' \preceq e$ are possible. We use an example for illustration.

Example 5.1. *We use Figure 5(a) to illustrate the order. Assuming $k_c = k_f = 1$, Figure 5(a) represents a digraph G with edge e_1 to be inserted. The red line indicates the edge to be inserted. The layers of G without considering edge e_1 are shown in the upper part of Figure 5(a). Initially, we have $e_2, e_4, e_7, e_9, e_{12}, e_{13}$ with support values smaller than the given parameters. These edges are peeled first and placed in L_1, which is marked in the first row of the table in Figure 5(a). The deletion of these edges subsequently leads to e_3, e_6, e_8 violating the support constraints, and they are placed in L_2. Finally, the remaining subgraph satisfies the support constraints and is accommodated in L_3. Therefore, the edges $E_{\preceq} = \{L_1, L_2, L_3\}$, and the maximum layer number is $\mu = 3$.*

5.2 Property Analysis of Layers on Streaming

We present the properties of layers on streaming updates of edge insertions/deletions, which are the basis of designing efficient D-truss community search algorithms.

Property 5.1. *Given a $E_{\preceq}$ for digraph G that is divided into μ layers, if we denote $H_j = E_G \backslash U_{i=1}^{j-1} L_i$ and $H_1 = E_G$, then the following properties hold:*

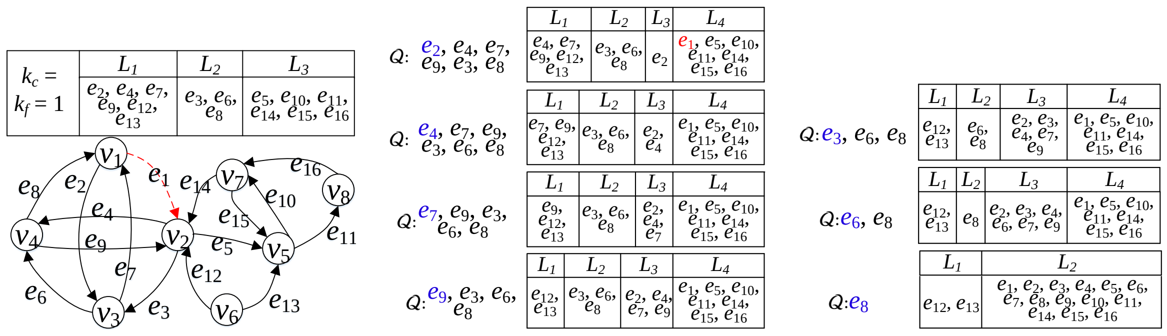
(1) Given an edge e with layer number $\mathcal{L}(e) = j$, we have $\{csup_{H_j}(e) < k_c \textbf{or} fsup_{H_j}(e) < k_f\} \textbf{and} \{csup_{H_{j-1}}(e) \geq k_c \textbf{and} fsup_{H_{j-1}}(e) \geq k_f\}$

(2) L_μ is the retrieved community. Specifically, for edges e in L_μ, we have $csup_{L_\mu}(e) \geq k_c$ and $fsup_{L_\mu}(e) \geq k_f$, and we cannot find another L_a with $a \leq \mu$.

Proof. First, Property 5.1(1) can be directly inferred from Definition 5.2. Next, we prove Property 5.1(2). For a graph G, if there is D-truss in G, then according to the peeling algorithm, all the edges with enough support will be returned as the community. If we denote the returned community as C, then for the edges in C, we have $csup_C \geq k_c$ and $fsup_C \geq k_f$. Also, according to the peeling algorithm, as long as we cannot find edges violating the support constraints, the whole algorithm will terminate and the remaining graph will be returned. Hence, only one layer satisfying Property 5.1(2) will be generated.

□

Property 5.1(1) is the underlying principle for us to accommodate the edges during the order maintenance. Note that Property 5.1(2) is based on the assumption that there are some edges in graph G satisfying the support constraints. Since examining whether L_μ is the community entails a very light cost, we focus on how to maintain the layers and retrieve L_μ. The main idea of our order maintenance algorithm is to heuristically put edges into the layer such that Property 5.1(1) is fulfilled until all edges are updated. The last layer of the updated order is ultimately retrieved as the


$k_c = k_f = 1$
L_1: $e_2, e_4, e_7, e_9, e_{12}, e_{13}$ | L_2: e_3, e_6, e_8 | L_3: $e_5, e_{10}, e_{11}, e_{14}, e_{15}, e_{16}$
v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8
e_1 e_2 e_3 e_4 e_5 e_6 e_7 e_8 e_9 e_{10} e_{11} e_{12} e_{13} e_{14} e_{15} e_{16}
Q: $e_2, e_4, e_7, e_9, e_3, e_8$ — L_1: $e_4, e_7, e_9, e_{12}, e_{13}$ | L_2: e_3, e_6, e_8 | L_3: e_2 | L_4: $e_1, e_5, e_{10}, e_{11}, e_{14}, e_{15}, e_{16}$
Q: $e_4, e_7, e_9, e_3, e_6, e_8$ — L_1: e_7, e_9, e_{12}, e_{13} | L_2: e_3, e_6, e_8 | L_3: e_2, e_4 | L_4: $e_1, e_5, e_{10}, e_{11}, e_{14}, e_{15}, e_{16}$
Q: e_7, e_9, e_3, e_6, e_8 — L_1: e_9, e_{12}, e_{13} | L_2: e_3, e_6, e_8 | L_3: e_2, e_4, e_7 | L_4: $e_1, e_5, e_{10}, e_{11}, e_{14}, e_{15}, e_{16}$
Q: e_9, e_3, e_6, e_8 — L_1: e_{12}, e_{13} | L_2: e_3, e_6, e_8 | L_3: e_2, e_4, e_7, e_9 | L_4: $e_1, e_5, e_{10}, e_{11}, e_{14}, e_{15}, e_{16}$
Q: e_3, e_6, e_8 — L_1: e_{12}, e_{13} | L_2: e_6, e_8 | L_3: e_2, e_3, e_4, e_7, e_9 | L_4: $e_1, e_5, e_{10}, e_{11}, e_{14}, e_{15}, e_{16}$
Q: e_6, e_8 — L_1: e_{12}, e_{13} | L_2: e_8 | L_3: $e_2, e_3, e_4, e_6, e_7, e_9$ | L_4: $e_1, e_5, e_{10}, e_{11}, e_{14}, e_{15}, e_{16}$
Q: e_8 — L_1: e_{12}, e_{13} | L_2: $e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}, e_{11}, e_{14}, e_{15}, e_{16}$

(a) Graph G (b) Maintaining Order $E_{\preceq}$: part 1 (c) Maintaining Order $E_{\preceq}$: part 2

Figure 5: An illustrative example of the order-based algorithm, with e_1 being the inserted edge. The edge being processed is marked as blue in (b) and (c).

community answer based on Property 5.1(2). Consider the edge insertion case as an example. When a new edge is inserted, we put the edges into some layers initially and adjust the layer for all of its neighboring edges. Specifically, at the beginning of processing, we should put it into the layers where Property 5.1(1) is fulfilled. Here, a special case may happen, that is, the new edge has support in L_μ larger than the query parameters. In this situation, we directly put it at the beginning of L_μ. Note that in both cases, we should still consider all the neighboring edges of the newly inserted edge in the subsequent processing. There is a key challenge, i.e., finding neighboring edges that may have their layers adjusted. We have the following theorems to help us to find the affected edges.

THEOREM 5.1. *All the neighboring edges of the newly inserted/deleted edge should be considered and processed in Algorithm 5*

PROOF. The proof of Theorem 5.1 is straightforward. Since all the neighbors of the newly inserted edge have their support incremented by 1, they may be accommodated to different layers. Hence, all of them should be taken into consideration when we maintain the layers. □

THEOREM 5.2. *Given an edge e_i, if its layer is recently incremented from L_o to L_n ($L_o \preceq L_n$), then only the edges in L_{o+1} to L_n may have their layers incremented.*

PROOF. We prove Theorem 5.2 then. The key to determining whether the adjacent edges of E_i, whose layers have recently been incremented, need to be updated is to find another layer that satisfies Property 5.1(1). We observe that the neighboring edges can only have their layers changed if they belong to the layers that are preceded by the original layer of E_i. Based on this observation, we derive Theorem 5.2. □

5.3 Order-based Maintenance Algorithms for D-truss Community Search

In this subsection, we first propose the order maintenance algorithms for streaming updates and then develop the corresponding techniques for D-truss community search. For order maintenance, we handle the cases of *edge insertions* and *edge deletions* like-wise. Therefore, we first present the order maintenance algorithm for an edge insertion.

Algorithm 5: Order Insertion Maintenance Algorithm

```
Input: a snapshot graph $G$, inserted edge $e_i = (u, v)$,
       parameters $k_c$ and $k_f$, original order
       $E_{\preceq} = L_1 L_2 \cdots L_\mu$
Output: updated order $E'_{\preceq} = L'_1 L'_2 \cdots$
1  Let a priority queue $Q \leftarrow \emptyset$ and graph $G \leftarrow G \cup \{e_i\}$;
2  if $csup_{H_{\mu-1}}(e_i) \geq k_c$ and $fsup_{H_{\mu-1}}(e_i) \geq k_f$ and
     $csup_{H_\mu}(e_i) < k_c$ or $fsup_{H_\mu}(e_i) < k_f$ then
3      insert a new layer $L'$ between $L_{\mu-1}$ and $L_\mu$;
4      insert $e_i$ to $L'$;
5  else
6      insert edge $e_i$ into $L_i$ such that $csup_{H_{i-1}}(e_i) \geq k_c$ and
         $fsup_{H_{i-1}}(e_i) \geq k_f$ and $csup_{H_i}(e_i) < k_c$ or
         $fsup_{H_i}(e_i) < k_f$ ;
7  $N(u) \leftarrow N^+_G(u) \cup N^-_G(u)$; $N(v) \leftarrow N^+_G(v) \cup N^-_G(v)$;
8  for each vertex $w \in N(u) \cap N(v)$ do
9      for $e' \in \{\langle u, w\rangle, \langle v, w\rangle, \langle w, u\rangle, \langle w, v\rangle\} \cap E_G$ do
10         $Q$.enqueue($e'$);
11 if all edges in $Q$ are in $L_\mu$ then
12     return;
13 while $Q \neq \emptyset$ do
       // $e'$ is the leftmost edge among egdes in $Q$
14     $e' \leftarrow Q$.dequeue();
15     if $csup_{H_{\mu-1}}(e') \geq k_c$ and $fsup_{H_{\mu-1}}(e') \geq k_f$ and
         $csup_{H_\mu}(e_i) < k_c$ or $fsup_{H_\mu}(e_i) < k_f$ then
16         insert a new layer $L'$ between $L_{\mu-1}$ and $L_\mu$;
17         insert $e'$ to $L'$;
18         if all edges in $L' \cup L_\mu$ satisfying support constraint
             then
19             append $L'$ to $L_\mu$;
20     else
21         move $e'$ right from original layer $L_i$ to $L_{i'}$ such that
             $csup_{H_{i'-1}}(e') \geq k_c$ and $fsup_{H_{i'-1}}(e') \geq k_f$ and
             $csup_{H_{i'}}(e') < k_c$ or $fsup_{H_{i'}}(e') < k_f$;
22     if $L_i \neq L_{i'}$ then
23         for $e_1$ and $e_2$ that form triangles with $e'$ and both in
             layers preceded by $L_i$ and precedes $L'_i$ and not in $L_\mu$
             do
24             $Q$.enqueue($e_1$), $Q$.enqueue($e_2$);
25 Return Updated $E_{\preceq}$ as $E'_{\preceq}$;
```

The pseudo-code for maintaining the order after a single edge insertion is presented is Algorithm 5. We introduce the notation

Algorithm 6: Order-based D-truss Community Search

```
Input: snapshot graph $G$, inserted edges $\mathcal{E}_i$, deleted edges
       $\mathcal{E}_d$ parameters $k_c$ and $k_f$, original order
       $E_{\preceq} = L_1 L_2 \cdots L_\mu$, query vertices $Q$
Output: updated $(k_c, k_f)$-truss for all the edges in updated
        $G'$
1 for $e_d \in \mathcal{E}_d$ do
2     $E_{\preceq'} \leftarrow$ Apply the order deletion maintenance algorithm
       similar to the order insertion maintenance in
       Algorithm 5$(G, e_d, k_c, k_f, E_{\preceq})$;
3 for $e_i \in \mathcal{E}_i$ do
4     $E_{\preceq'} \leftarrow$ Algorithm 5$(G, e_i, k_c, k_f, E_{\preceq'})$;
5 if edges in $L'_\mu$ satisfy the support constraint then
6     $DT \leftarrow$ the last layer $L'_\mu$ of $E_{\preceq'}$;
7 if $Q \subset DT$ then
8     Return $DT$;
9 Return $\emptyset$;
```

$H_j = E_G \setminus U_{i=1}^{j-1} L_i$, where H_1 is E_G, for the ease of presentation. It is worth noting that the operations of removing an edge from an order, inserting an edge into an order, and determining the precedence of edges can be performed in constant time [5, 13]. The algorithm proceeds as follows: First, the newly inserted edge e is placed in layer L_i based on Property 5.1(1) (lines 5-6) or a new layer is created (lines 1-4). Second, all neighboring edges of e are added to a list (lines 7-10). Third, for each edge in the list, it is either placed in a layer to satisfy Property 5.1(1) (lines 20-21), or a new layer is generated to accommodate it (lines 15-17). After placing an edge in a new layer, it is removed from the list, and all its neighboring edges fulfilling Theorem 5.2 are added to the list (lines 22-24). This process is repeated until the list becomes empty. It is important to note that the edges in the list are processed from left to right. Some corner cases may arise during the execution of the algorithm. Firstly, certain layers may be merged with L'_μ and subsequently removed from $E_{\preceq}$, resulting in a decrease in the total number of layers in the new order $E_{\preceq}$ (lines 18-19). Specifically, if all edges in a layer L_o satisfy $csup_{H_o} \geq k_c$ and $fsup_{H_o} \geq k_f$, all layers $L_o \cdots L_{\mu-1}$ can be combined with L_μ. Secondly, new layers may be generated (lines 2-3, 15-16). This occurrence is due to the adjustment of the edges' layer numbers, during which some edges have sufficient support in $L_{\mu-1} \cup L_\mu$ but lacks the necessary support in L_μ. Consequently, a new layer is created "between" the original $L_{\mu-1}$ and L_μ to ensure the fulfillment of Property 5.1(1).

The overall algorithm for order maintenance with edge deletions is similar to that of the insertion case, with the following difference:

(1) Given an edge e_i, if its layer is recently decremented from L_o to L_n ($L_n \preceq L_o$). Then only edges in L_n to L_{o+1} may have their layers decremented. Note that if L_{o+1} happens to be L_μ, the range is L_n to L_o.

With both the edge insertion and deletion algorithms, we now present the order-based community search algorithm, as outlined in Algorithm 6. We will maintain the layers with edge deletions and insertions, respectively (lines 2 and 4), and then directly output L'_μ. Example 5.2 illustrates the case of an edge insertion. The case for edge deletions is similar and thus omitted. Note that the peeling-based algorithm and the order-based algorithm cannot be combined. Specifically, the peeling-based algorithm iteratively removes the edges that violate the flow and cycle support constraints. This removal process can generate an order to denote the sequence of the edges' removal. Correspondingly, the order-based algorithm is to maintain the above order to simulate the peeling process. In other words, these two algorithms inherently share the same principle with different implementations and cannot be combined.

Example 5.2. *We use a graph in Figure 5 to illustrate Algorithm 5 for $k_c = k_f = 1$. How the layers in G when edge e_1 (marked as red) is inserted are shown in Figure 5(b) and 5(c), where each row represents the order with one edge being processed. Note that the edges in the current Q are displayed on the left of each row, with the one being processed marked as blue. The original layer is displayed on the upper side of Figure 5(a). Note that when we have inserted edge 8 into the second layer from the right, the condition in line 18 of Algorithm 5 is fulfilled and this layer is appended to L'_μ. The updated layers are shown in the third row of Figure 5(c).*

Theorem 5.3. ***(Time and Space Complexities)*** *Let G^t be the directed graph in the window W^t, C be the number of edges that have their layers changed, m_t be the number of edges in G^t, δ be the maximal cycle-support/flow-support in G^t, and μ be the total number of layers. Algorithm 5 takes $O(C * \mu * \delta)$ time and $O(m_t)$ space for each window W^t.*

Proof. An edge can have its layer changed at most $\mathcal{L}$ times. As each time costs δ time, the total time complexity is $O(C * \mathcal{L} * \delta)$. The space complexity is $O(m_t)$. □

6 PERFORMANCE EVALUATION

In this section, we evaluate the efficiency of our proposed algorithms through extensive experiments on real-world datasets. All experiments are conducted on a Linux server with an Intel Xeon Gold 6230R 2.1GHz CPU and 128 GB of memory, running Oracle Linux 8.6. Our algorithms were implemented in C++.

Datasets. We use six real-world directed graphs in our experiments. Table 1 summarizes the statistics of these graphs. Note that k_c^{max} and k_f^{max} are the maximal parameters for which a non-empty $(k_c^{max}, 0)$-truss or $(0, k_f^{max})$-truss can be obtained with the default window size for each dataset. Specifically, `College-Msg`[1] is a private message graph; `Amazon`[1] is a product co-purchasing network; `DBLP`[1] is a co-authorship network ; `Flickr`[2] is a social graph; `Sx-Mathoverflow`[1], `Ask-Ubuntu`[1], and `Stack-Overflow`[1] are internet interaction graphs; `UK-2002`[3] is a web graph. Note that `UK-2002`, `Amazon`, and `DBLP` do not come with timestamps. Thus, we generated random timestamps for each edge in them.

Algorithms. We compare several algorithms in our experiments.

- **DYNAMIC**: [44] proposes batch insertion and deletion algorithms for D-truss retrieval in dynamic graphs. We adapted these algorithms to solve our problem.
- **REPEEL**: The basic peeling-based algorithm that peels the directed graph whenever the window slides.

[1]http://snap.stanford.edu/data/index.html
[2]http://konect.cc
[3]http://law.di.unimi.it/datasets.php

Table 1: Statistics of the datasets (K = 10^3 and M = 10^6)

Dataset	Abbr.	$\lvert V_G\rvert$	$\lvert E_G\rvert$	deg_{avg}	k_c^{max}	k_f^{max}
College-Msg	MSG	1.9K	59.8K	63.0	6	8
Sx-Mathoverflow	SX	24.8K	506.5K	40.8	7	6
Ask-Ubuntu	UT	159.3K	964.4K	12.1	2	3
Amazon	AM	334.8K	1.9M	11.1	2	2
DBLP	DP	317.1K	2.1M	13.2	10	10
Flickr	FC	2.3M	33.1M	28.8	3	2
Stack-Overflow	SO	2.6M	63.5M	48.8	1	4
UK-2002	UK2	18.5M	298.1M	32.2	7	5

Table 2: Default parameter settings

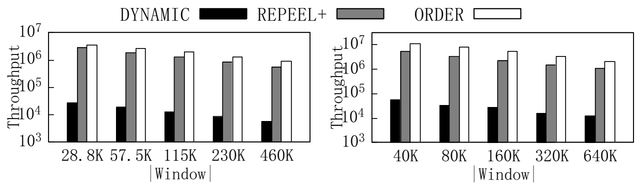
Dataset	$\lvert Window\rvert$	$\lvert Stride\rvert$	k_c	k_f
College-Msg	9K	900	1	1
Sx-Mathoverflow	6K	600	0	1
Ask-Ubuntu	11K	1.1K	0	1
Amazon	19K	1.9K	1	1
DBLP	21K	2.1K	8	8
Flickr	115K	11.5K	2	2
Stack-Overflow	160K	16K	0	1
UK-2002	700K	70K	4	4

- **REPEEL + OPT1, REPEEL + OPT2, and REPEEL + OPT3**: The peeling-based algorithm with optimizations OPT-1, OPT-2, and OPT-3, respectively.
- **REPEEL+**: The peeling-based algorithm with all three optimizations.
- **ORDER**: The order-based algorithm.

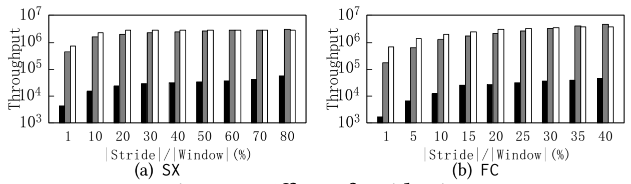
Parameters and Metrics. The parameters tested in the experiments include k_c and k_f, the window size τ, and the stride size β. The settings for k_c and k_f are determined based on the intrinsic characteristics of the graph and are different for different datasets. In our problem, k_c and k_f are parameters specified by the users, which can not be adjusted adaptively. Usually, if the users want to find dense and small communities, they can set large values of k_c and k_f. Otherwise, the users can input small values of k_c and k_f. Our objective is to choose these parameters to be as large as possible, as larger values allow us to retrieve denser communities. However, it is essential to strike a balance to ensure that the resulting community remains meaningful, containing a sufficient number of densely connected edges. Another factor that affects the resulting community is the window size. A larger window size encompasses more edges in the snapshot graph, thereby reducing the likelihood of obtaining an empty final community. Table 2 summarizes the default parameter settings. For each experiment, we report the throughput, which represents the number of processed edges per second. In each experiment, we run 100 queries and report the average throughput. If an algorithm cannot complete within 10 days, it is denoted by INF.

6.1 Efficiency Evaluation

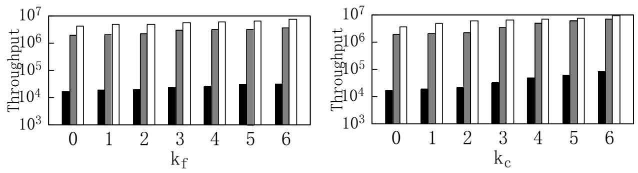
Exp-1: Effect of window size. In the first experiment, we evaluate the performance of our proposed algorithms by varying the window size. The results are shown in Figure 6. As expected, the throughput of all algorithms decreases when the window size increases. This is because a larger window size induces a larger directed graph, which takes more time to process. In addition, we observe that both REPEEL+ and ORDER are two orders of magnitude faster than DYNAMIC. This is because REPEEL+ integrates all three optimizations we proposed, which reduce the size of the peeling input, resulting in higher efficiency. Moreover, ORDER is more efficient than REPEEL+ because it does not require peeling the graph from scratch.



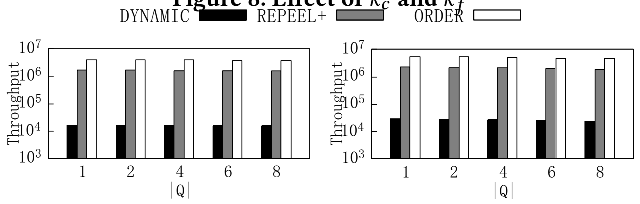
(a) FC (b) SO
Figure 6: Effect of window size



(a) SX (b) FC
Figure 7: Effect of stride size



(a) MSG (k_c = 1) (b) MSG (k_f = 1)
Figure 8: Effect of k_c and k_f



(a) UT Figure 9: Effect of $|Q|$ (b) SO

Exp-2: Effect of stride size. We next evaluate the impact of stride size on our proposed algorithms by varying the stride size on SX and FC, respectively. As shown in Figure 7, larger strides result in higher throughput. This is because larger strides mean fewer community searches need to be performed, leading to higher throughput. In all cases, both REPEEL+ and ORDER outperform DYNAMIC. Additionally, it is worth noting that although both REPEEL+ and ORDER exhibit an improvement in throughput as the stride size increases, the magnitude of this improvement is larger for REPEEL+ compared to ORDER. Consequently, when dealing with large stride sizes (specifically, 80% and 35% of the window size for SX and FC), the peeling-based method (REPEEL+) outperforms the order-based algorithm (ORDER) in terms of efficiency. This observation can be attributed to the fact that although both algorithms require searching through fewer communities, it takes more time for ORDER to maintain a community compared to REPEEL+ when there are a larger number of edge insertions or deletions. The discrepancy arises from the higher time cost required by ORDER to adjust the layers of edges. In contrast, the time cost for REPEEL+ to process a single community remains relatively stable, regardless of the number of edge modifications.

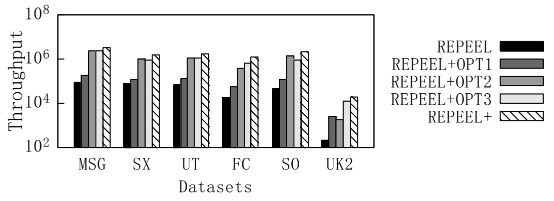


Figure 10: Effect of optimizations

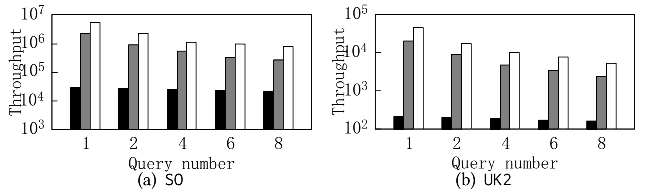


Figure 11: Effect of query number

Exp-3: Effect of query parameters k_c and k_f. We also evaluate the impact of parameters k_c and k_f on the performance of our proposed algorithms using the MSG datasets. To do so, we fix one parameter and vary the other. The results are presented in Figure 8. We can see that the throughput of the algorithms gradually increases as k_c and k_f grow. As the query parameters increase, the search space becomes smaller, resulting in higher throughput.

Exp-4: Effect of the query vertex number $|Q|$. To examine the impact of the number of query vertices on our algorithms, we vary $|Q|$ from 1 to 8. The results in Figure 9 show that when $|Q|$ increases, the throughput of all algorithms decreases slightly. The reason behind this is that the running time overhead is dominated by the maintenance of the community and checking whether the query vertices are contained in the community entails very slight overhead. Hence, the variance of $|Q|$ does not cause too much impact on throughput. However, it is worth noting that ORDER consistently outperforms the other algorithms.

Exp-5: Effect of the optimizations. Next, we evaluate the effectiveness of our proposed optimizations for REPEEL, and the results are presented in Figure 10. It is clear that the algorithm with all optimizations has the best performance, followed by the algorithms with only one optimization. The basic peeling-based algorithm without any optimization has the worst performance, demonstrating the effectiveness of our proposed optimizations. Furthermore, REPEEL+ consistently outperforms REPEEL by one to two orders of magnitude over all datasets. For example, REPEEL+ is 96 times faster than REPEEL on UK2. On average, the optimizations boost the algorithm by 44 times. Additionally, the efficiency of the optimizations varies across different datasets. For instance, OPT-1 outperforms OPT-2 on MSG, SX, UT, FC, and S0, while OPT-2 has a better performance on UK2 compared to OPT-1.

Exp-6: Effect of query numbers. In practical scenarios with adjustable parameters, it is often advantageous to perform simultaneous queries for multiple parameter combinations. Therefore, we conducted an assessment of the impact of processing multiple queries concurrently. Our algorithm can maintain multiple orders simultaneously by maintaining an order or peeling a graph for each query parameter setting. As for DYNAMIC, which retains the entire trussness sets for each edge, we maintained the proposed D-index and executed all the queries based on the updated index [44]. The results are presented in Figure 11. As the number

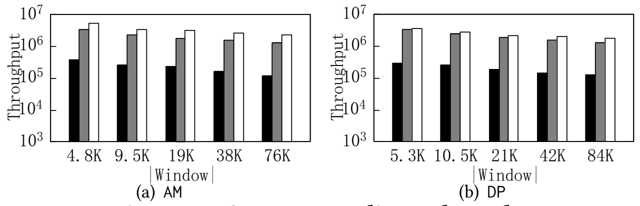


Figure 12: Query on undirected graphs

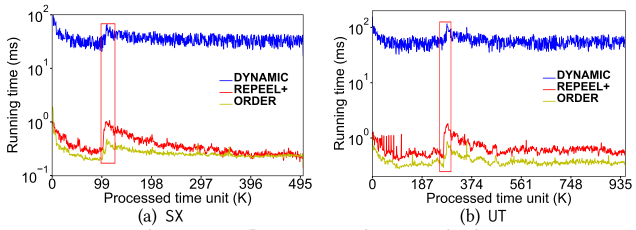
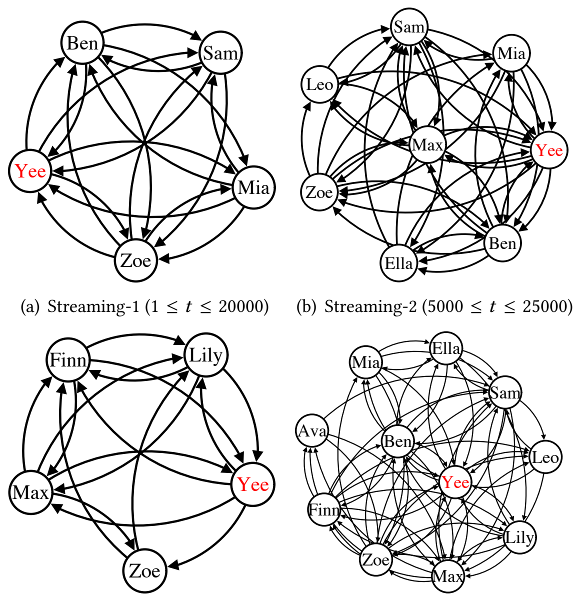


Figure 13: Response time variation

of queries increases, the resource utilization of DYNAMIC remains relatively stable. This stability arises because the primary overhead of DYNAMIC is associated with index maintenance, a one-time task, regardless of the query count. Most of the growth in running time can be attributed to the execution of queries themselves. In contrast, for ORDER and REPEEL+, which involve the maintenance of order or graph peeling for each query, the computational cost increases linearly with the rising number of queries. However, it's worth noting that our methods consistently outperform DYNAMIC by orders of magnitude.

Exp-7: Query the undirected graph. This experiment validates the efficiency of our algorithms when querying undirected graphs. To handle k-truss community search on undirected graphs, we first transform the undirected graph into a directed one by replacing each undirected edge with two bidirectional edges. During the query, we set $k_c = k_f = k$ and return the D-truss community. To convert it back to the k-truss community, we replace the bidirectional edges with an undirected edge. We compare the performance of our algorithms against the baseline across various window sizes. The results, illustrated in Figure 12, consistently align with our findings in Exp-1. Our algorithms consistently outperform the baseline by several orders of magnitude, demonstrating the strong generalization capability of our algorithms to undirected graphs.

Exp-8: Examination on the response time variation . As the sliding window moves, it is natural for the response time to exhibit fluctuations. To evaluate these variations in our algorithms' response time, we measured the processing time for a single sliding window during the processing. The experimental results are presented in Figure 13. The response time varies as the window slides. However, the magnitude of change in the DYNAMIC algorithm is larger compared to the other two algorithms. This can be attributed to the fact that the dynamic algorithm itself has a longer execution time, which leads to larger variations in its performance. Overall, these three algorithms demonstrate stability. However, there are specific time points, such as 104K in Figure 13(a) and 270K in Figure 13(b) (marked with red rectangles), where these algorithms exhibit relatively large fluctuations. This is because, at those particular moments, the updates to the edges result in significant changes to the underlying graphs and communities, thereby causing larger fluctuations in the response time.



(a) Streaming-1 ($1 \le t \le 20000$) (b) Streaming-2 ($5000 \le t \le 25000$)

(c) Streaming-3 ($10000 \le t \le 30000$) (d) Non-streaming ($1 \le t \le 30000$)

Figure 14: Case study over Flickr

6.2 Case Study

In this subsection, we present a case study on `Flickr` to monitor the evolution of communities among users in real time. Specifically, `Flickr` contains 30,000 consecutive edges. In our case study, we take the user *Yee* as the query vertex and set the parameters as follows: k_c=2, k_f=1, |Window| = 20,000, |Stride| = 5,000. Figures 14(a), 14(b), and 14(c) show *Yee*'s communities at timestamps 20,000, 25,000, and 30,000, respectively. For comparison, we also take `Flickr` as a static directed graph and find the D-truss community of *Yee*, which is shown in Figure 14(d). We can observe that *Yee*'s community becomes larger at timestamp 25,000 with the addition of new friends *Max*, *Leo*, and *Ella*. Subsequently, at timestamp 30,000, *Yee*'s community shrinks as old friends *Mia*, *Ben*, *Sam*, *Leo*, and *Ella* are replaced by *Lily* and *Finn*. This community evolution clearly shows the latest friendships of *Yee* at different periods. For example, in the time period [1, 20,000], *Yee* interacts more frequently with *Ben*, *Sam*, *Mia*, and *Zoe*. However, in the time period [25,000, 30,000], *Yee* communicates better with *Finn*, *Lily*, *Max*, and *Zoe*. In contrast, the static community in Figure 14(d) only shows a large community over the whole time period, which cannot reflect such time-dependent information.

7 CONCLUSION

In this paper, we have studied the problem of D-truss community search over streaming directed graphs. To address the problem, we have proposed two algorithms, i.e., the peeling-based algorithm and the order-based algorithm. The peeling-based algorithm peels the directed graph to obtain the community whenever the window slides. We have also devised three optimizations, including upper-bounds-based pruning, BFS search, and life-time prediction, to improve the performance of the peeling-based algorithm. Moreover, we have introduced the community retrieving order and layers in order to devise the order-based algorithm. Our theoretical analysis and empirical evaluations confirm the efficiency of our proposed algorithms. In the future, we plan to develop efficient distributed algorithms with parallel strategies for community search over multi-source streaming graphs.

REFERENCES

[1] Esra Akbas and Peixiang Zhao. 2017. Truss-based community search: a truss-equivalence based indexing approach. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1298–1309.

[2] Soumya Banerjee, Sumit Singh, and Eiman Tamah Al-Shammari. 2018. Community Detection in Social Network: An Experience with Directed Graphs. In *Encyclopedia of Social Network Analysis and Mining, 2nd Edition*. Springer.

[3] Nicola Barbieri, Francesco Bonchi, Edoardo Galimberti, and Francesco Gullo. 2015. Efficient and effective community search. *Data mining and knowledge discovery* 29, 5 (2015), 1406–1433.

[4] Vladimir Batagelj and Matjaz Zaversnik. 2003. An O (m) algorithm for cores decomposition of networks. *arXiv preprint cs/0310049* (2003).

[5] Michael A Bender, Richard Cole, Erik D Demaine, Martin Farach-Colton, and Jack Zito. 2002. Two simplified algorithms for maintaining order in a list. In *Algorithms—ESA 2002: 10th Annual European Symposium Rome, Italy, September 17–21, 2002 Proceedings*. Springer, 152–164.

[6] Lijun Chang, Xuemin Lin, Lu Qin, Jeffrey Xu Yu, and Wenjie Zhang. 2015. Index-based optimal algorithms for computing steiner components with maximum connectivity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 459–474.

[7] Lu Chen, Chengfei Liu, Rui Zhou, Jianxin Li, Xiaochun Yang, and Bin Wang. 2018. Maximum co-located community search in large scale social networks. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1233–1246.

[8] Shu Chen, Ran Wei, Diana Popova, and Alex Thomo. 2016. Efficient computation of importance based communities in web-scale networks using a single machine. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. 1553–1562.

[9] Yankai Chen, Jie Zhang, Yixiang Fang, Xin Cao, and Irwin King. 2021. Efficient community search over large directed graphs: An augmented index-based approach. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*. 3544–3550.

[10] Wanyun Cui, Yanghua Xiao, Haixun Wang, Yiqi Lu, and Wei Wang. 2013. Online search of overlapping communities. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*. 277–288.

[11] Wanyun Cui, Yanghua Xiao, Haixun Wang, and Wei Wang. 2014. Local search of communities in large graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 991–1002.

[12] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 2002. Maintaining stream statistics over sliding windows. *SIAM journal on computing* 31, 6 (2002), 1794–1813.

[13] Paul Dietz and Daniel Sleator. 1987. Two algorithms for maintaining order in a list. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. 365–372.

[14] Yue Ding, Ling Huang, Chang-Dong Wang, and Dong Huang. 2017. Community detection in graph streams by pruning zombie nodes. In *Advances in Knowledge Discovery and Data Mining: 21st Pacific-Asia Conference, PAKDD 2017, Jeju, South Korea, May 23-26, 2017, Proceedings, Part I 21*. Springer, 574–585.

[15] Yixiang Fang, Reynold Cheng, Yankai Chen, Siqiang Luo, and Jiafeng Hu. 2017. Effective and efficient attributed community search. *The VLDB Journal* 26, 6 (2017), 803–828.

[16] Yixiang Fang, Reynold Cheng, Xiaodong Li, Siqiang Luo, and Jiafeng Hu. 2017. Effective community search over large spatial graphs. *Proceedings of the VLDB Endowment* 10, 6 (2017), 709–720.

[17] Yixiang Fang, Xin Huang, Lu Qin, Ying Zhang, Wenjie Zhang, Reynold Cheng, and Xuemin Lin. 2020. A survey of community search over big graphs. *The VLDB Journal* 29, 1 (2020), 353–392.

[18] Yixiang Fang, Zheng Wang, Reynold Cheng, Xiaodong Li, Siqiang Luo, Jiafeng Hu, and Xiaojun Chen. 2018. On spatial-aware community search. *IEEE Transactions on Knowledge and Data Engineering* 31, 4 (2018), 783–798.

[19] Yixiang Fang, Zhongran Wang, Reynold Cheng, Hongzhi Wang, and Jiafeng Hu. 2018. Effective and efficient community search over large directed graphs. *IEEE Transactions on Knowledge and Data Engineering* 31, 11 (2018), 2093–2107.

[20] Christos Giatsidis, Dimitrios M Thilikos, and Michalis Vazirgiannis. 2013. D-cores: measuring collaboration of directed graphs based on degeneracy. *Knowledge and information systems* 35, 2 (2013), 311–343.

[21] Xiangyang Gou and Lei Zou. 2021. Sliding window-based approximate triangle counting over streaming graphs with duplicate edges. In *Proceedings of the 2021 International Conference on Management of Data*. 645–657.

[22] Wafaa MA Habib, Hoda MO Mokhtar, and Mohamed E El-Sharkawi. 2020. Weight-based k-truss community search via edge attachment. *IEEE Access* 8 (2020), 148841–148852.

[23] Jorge E. Hirsch. 2005. *H-index*. https://en.wikipedia.org/wiki/H-index

[24] Alexandre Hollocou, Julien Maudet, Thomas Bonald, and Marc Lelarge. 2017. A linear streaming algorithm for community detection in very large networks. *arXiv preprint arXiv:1703.02955* (2017).

[25] Jiafeng Hu, Xiaowei Wu, Reynold Cheng, Siqiang Luo, and Yixiang Fang. 2016. Querying minimal steiner maximum-connected subgraphs in large graphs. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management.* 1241–1250.

[26] Jiafeng Hu, Xiaowei Wu, Reynold Cheng, Siqiang Luo, and Yixiang Fang. 2017. On minimal steiner maximum-connected subgraph queries. *IEEE Transactions on Knowledge and Data Engineering* 29, 11 (2017), 2455–2469.

[27] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-truss community in large and dynamic graphs. In *Proceedings of the 2014 International Conference on Management of Data.* 1311–1322.

[28] Xin Huang and Laks VS Lakshmanan. 2017. Attribute-driven community search. *Proceedings of the VLDB Endowment* 10, 9 (2017), 949–960.

[29] Xin Huang, Laks VS Lakshmanan, and Jianliang Xu. 2019. Community search over big graphs. *Synthesis Lectures on Data Management* 14, 6 (2019), 1–206.

[30] Xin Huang, Laks VS Lakshmanan, Jeffrey Xu Yu, and Hong Cheng. 2015. Approximate closest community search in networks. *Proceedings of the VLDB Endowment* 9, 4 (2015), 276–287.

[31] Bogyeong Kim, Kyoseung Koo, Undraa Enkhbat, and Bongki Moon. 2022. DenForest: Enabling Fast Deletion in Incremental Density-Based Clustering over Sliding Windows. In *Proceedings of the 2022 International Conference on Management of Data.* 296–309.

[32] Bogyeong Kim, Kyoseung Koo, Undraa Enkhbat, and Bongki Moon. 2022. DenForest: Enabling Fast Deletion in Incremental Density-Based Clustering over Sliding Windows. In *Proceedings of the 2022 International Conference on Management of Data.* 296–309.

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

[34] Jianxin Li, Xinjue Wang, Ke Deng, Xiaochun Yang, Timos Sellis, and Jeffrey Xu Yu. 2017. Most influential community search over large social networks. In *2017 IEEE 33rd international conference on data engineering (ICDE).* IEEE, 871–882.

[35] Rong-Hua Li, Lu Qin, Jeffrey Xu Yu, and Rui Mao. 2015. Influential community search in large networks. *Proceedings of the VLDB Endowment* 8, 5 (2015), 509–520.

[36] Rong-Hua Li, Lu Qin, Jeffrey Xu Yu, and Rui Mao. 2017. Finding influential communities in massive networks. *The VLDB Journal* 26, 6 (2017), 751–776.

[37] Rong-Hua Li, Jiao Su, Lu Qin, Jeffrey Xu Yu, and Qiangqiang Dai. 2018. Persistent community search in temporal networks. In *2018 IEEE 34th International Conference on Data Engineering (ICDE).* IEEE, 797–808.

[38] Panagiotis Liakos, Katia Papakonstantinopoulou, Alexandros Ntoulas, and Alex Delis. 2020. Rapid detection of local communities in graph streams. *IEEE Transactions on Knowledge and Data Engineering* 34, 5 (2020), 2375–2386.

[39] Qing Liu, Minjun Zhao, Xin Huang, Jianliang Xu, and Yunjun Gao. 2020. Truss-based community search over large directed graphs. In *Proceedings of the 2020 ACM International Conference on Management of Data.* 2183–2197.

[40] Qing Liu, Yifan Zhu, Minjun Zhao, Xin Huang, Jianliang Xu, and Yunjun Gao. 2020. VAC: vertex-centric attributed community search. In *2020 IEEE 36th International Conference on Data Engineering (ICDE).* IEEE, 937–948.

[41] Anil Pacaci, Angela Bonifati, and M. Tamer Özsu. 2020. Regular Path Query Evaluation on Streaming Graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data.* 1415–1430.

[42] Aida Sheshbolouki and M Tamer Özsu. 2022. sGrapp: Butterfly approximation in streaming graphs. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 16, 4 (2022), 1–43.

[43] Mauro Sozio and Aristides Gionis. 2010. The community-search problem and how to plan a successful cocktail party. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining.* 939–948.

[44] Anxin Tian, Alexander Zhou, Yue Wang, and Lei Chen. 2022. Maximal D-truss Search in Dynamic Directed Graphs. *Proceedings of the VLDB Endowment* 16, 9 (2022), 2199–2211.

[45] Chang-Dong Wang, Jian-Huang Lai, and Philip S Yu. 2013. Dynamic community detection in weighted graph streams. In *Proceedings of the 2013 SIAM international conference on data mining.* SIAM, 151–161.

[46] Di Yang, Elke A Rundensteiner, and Matthew O Ward. 2009. Neighbor-based pattern detection for windows over streaming data. In *Proceedings of the 12th international conference on extending database technology: advances in database technology.* 529–540.

[47] Di Yang, Elke A Rundensteiner, and Matthew O Ward. 2009. Neighbor-based pattern detection for windows over streaming data. In *Proceedings of the 12th international conference on extending database technology: advances in database technology.* 529–540.

[48] Long Yuan, Lu Qin, Wenjie Zhang, Lijun Chang, and Jianye Yang. 2017. Index-based densest clique percolation community search in networks. *IEEE Transactions on Knowledge and Data Engineering* 30, 5 (2017), 922–935.