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

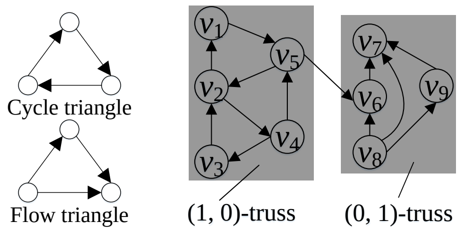
Community search aims to retrieve dense subgraphs that contain the query vertices. While many effective community models and algorithms have been proposed in the literature, none of them address the unique challenges posed by streaming graphs, where edges are continuously generated over time. In this 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

PVLDB Artifact Availability:
The source code, data, and other artifacts have been made available at https://github.com/hkbudb/streaming-dtruss.

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, 42]. Recently, researchers have been investigating the problem of community search over directed graphs [21, 43]. 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 [10, 20, 21] and D-truss [43, 50]. 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 equal status within the community. On the other hand, the $(0, 1)$-truss exhibits 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).



(a) Triangles (b) D-trusses

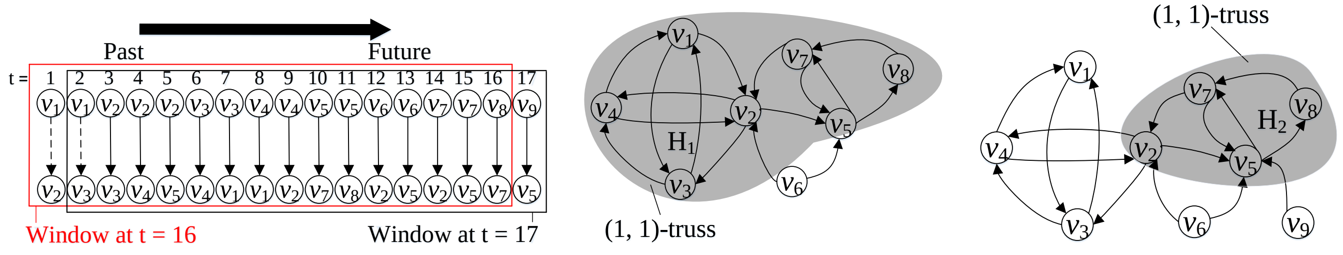
Figure 1: Examples of directed graph and D-truss

The D-truss model has numerous real-world applications [3, 32, 33, 43, 50]. For example, in social networks like Twitter and Facebook, D-truss community search can be used to identify friendship communities with key opinion leaders. The D-truss model can also be applied to fraud detection in financial networks, where vertices represent financial entities (e.g., banks, firms) and directed edges represent fund transfers between entities. By leveraging the D-truss model, we can detect fraud rings composed of kingpins and minions.

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 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. Community search in streaming social networks can track users' up-to-date communities, allowing more effective friendship recommendations. 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. Community search in streaming e-commerce networks can help find the latest customer groups with the same interests for real-time advertisement.

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.

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



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

Figure 2: A streaming graph

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 [50]. 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 [44, 49], while streaming graphs consist of an unbounded sequence of edges arriving at a high speed. Consequently, previous index-query-based algorithms cannot satisfy the high-throughput requirement in the streaming setting, and specialized algorithms for community search over streaming graphs are thus 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 [22, 34, 46, 52]. 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 [13, 22]. For example, in Figure 2, let $k_c = k_f = 1$, and consider 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, highlighted 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. This process is facilitated by three optimizations that significantly reduce the search space, namely *OPT-1-upper bounds based pruning*, *OPT-2-BFS-based update*, and *OPT-3-lifetime support prediction*. The first optimization devises upper bounds c_m and f_m of cycle and flow trussnesses for every edge. 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, where edges are *triangle connected* to the new coming edges, and their cycle and flow supports are greater than k_c and k_f, respectively. The third optimization utilizes the edge's arrival time and the window size to predict its *lifetime supports*, which track the supports of the edge in a series of future windows over incoming streams. Then, we use the lifetime supports to prune unqualified edges directly, thereby improving efficiency.

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

We summarize the main contributions of this paper as follows:

- We study a novel problem of D-truss community search over streaming graphs, which continuously identifies query-dependent D-truss communities within a sliding window.
- We propose a peeling-based algorithm to address the D-truss community search over streaming graphs, which incorporates three optimizations to reduce the graph for peeling.
- We introduce the *D-truss peeling order* to maintain 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 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. [51] 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. [15] 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. [25] investigated an edge streaming setting and proposed a method to construct communities by detecting local changes at each edge arrival. Liakos et al. [41] 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 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 [48]. Subsequently, numerous efforts have been devoted to exploring community search based on various models [18, 30], 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 [4, 5, 12, 48]. The k-truss model requires every edge in a community to be contained within at least $k-2$ triangles [1, 28, 31]. The clique model ensures that any two vertices in a community are connected to each other [11, 54]. On the other hand, the k-ECC-based community search defines a community as a steiner maximum-connected subgraph [7, 26, 27]. Besides simple graphs, community search has also been widely studied for complex graphs, such as directed graphs [10, 20, 43], temporal graphs [40], weighted graphs [9, 23, 37–39], attributed graphs [16, 29, 45], and spatial graphs [8, 17, 19]. More recently, an indexing method has been proposed for maximal D-truss searches over dynamic directed graphs [50]. However, [50] 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 inspires 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 notions, 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 [47].

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 *[35, 46, 53].*

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 [46]. Figure 2(a) illustrates an example of a streaming digraph, where $\tau = 16$ and $\beta = 1$. Figures 2(b) and 2(c) show 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 [43].*** *Given a digraph 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 [43].*** *Given a digraph 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 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 from 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 digraphs. We first present a

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 $\emptyset$;
```

basic algorithm, and then propose several optimizations to enhance the algorithm's performance.

4.1 Basic Peeling-based Algorithm

Given a digraph, a common approach (known as the peeling algorithm) to retrieve the D-truss community is through iterative deletion of edges whose cycle and flow supports are smaller than k_c and k_f, respectively. The pseudocode of the peeling algorithm is provided in Algorithm 1. Specifically, it first computes the cycle and flow supports for each edge and identifies the edges whose cycle and flow supports are smaller than k_c and k_f, respectively (lines 2-5). These edges are considered unqualified and are subsequently deleted from the graph (lines 7-8). The supports of their neighboring edges are updated correspondingly, and the neighboring edges with cycle and flow supports smaller than k_c and k_f will be removed in the subsequent round of edge deletions (lines 9-14). The process of deletions continues until all edges in the graph satisfy the support constraints. Finally, the remaining graph is returned as the D-truss community. Based on Algorithm 1, a basic method, called the peeling-based algorithm, for D-truss community search over streaming digraphs is as follows: whenever the window slides, we employ Algorithm 1 to peel the updated snapshot digraph and retrieve the D-truss community.

Theorem 4.1. ***(Time and Space Complexities)*** *Let G^t be the dirgraph 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 [36]. 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 $O(m_t)$. □

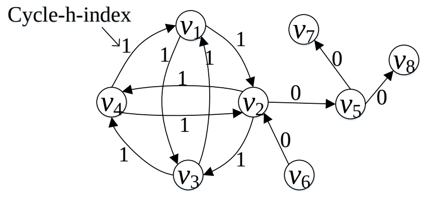


Figure 3: An illustrative example of OPT-1

The peeling-based algorithm is primarily designed for static graphs and may not be well-suited to high-speed streaming graphs where real-time community updates are required. Therefore, in the following subsections, we propose three optimizations to improve the efficiency of the peeling-based algorithm.

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

In the first optimization, referred to as upper-bounds-based pruning, we propose novel notations $c_m(e)$ and $f_m(e)$ for each edge e, based on which we shrink the digraph for the peeling process. To begin, we provide the formal definitions 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 maximum integer k_c^m such that there is a $(k_c^m, 0)$-truss $\subseteq G$ containing e. Similarly, $f_m(e)$ is the maximum integer k_f^m such that there is a $(0, k_f^m)$-truss $\subseteq G$ containing e.*

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

Proof. We prove this theorem by contradiction. Assume that the edge e is contained in a (k_c, k_f)-truss, where $k_c > c_m(e)$ or $k_f > f_m(e)$. However, according to the definitions of $c_m(e)$ and $f_m(e)$, they are the maximal integers such that there is a $(c_m(e), 0)$-truss or $(0, f_m(e))$-truss containing e, which leads to a contradiction. □

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 digraph 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 [24]. 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 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 in a digraph G is the*

H-index of an integer set S', where $S' = \{\min(fsup_G(e'), fsup_G(e'')): \forall e', e''$ 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 may 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.*

Proof. We prove this theorem by contradiction. Assume that the cycle-h-index of the edge e, denoted as c, is less than $c_m(e)$. However, according to the definition of $c_m(e)$, we can identify at least $c_m(e)$ vertices that are capable of forming cycle triangles with e. Moreover, each edge within these triangles also possesses a cycle support that is not less than $c_m(e)$. With these $c_m(e)$ vertices, we can deduce that the cycle-h-index of e is at least $c_m(e)$, which leads to a contradiction. The case for the flow-h-index can be proven in a similar manner. □

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. Building upon Theorem 4.3, we can derive that the cycle-h-index and flow-h-index of an edge e serve as an upper bound for $c_m(e)$ and $f_m(e)$, respectively. Hence, Theorem 4.4 holds by following from 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 digraph, 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 digraph is taken as the input digraph for peeling in Algorithm 1.

Example 4.1. *Suppose we want to retrieve a $(2, 0)$-truss from the digraph in Figure 3. The number along an edge is the cycle-h-index of the corresponding edge. As shown in the figure, since the cycle-h-indexes of all edges are smaller than 2, we can safely prune all edges.*

Theorem 4.5. ***(Time and Space Complexities)** Let G^t be the digraph in the window W^t. If we denote the number of edges in G^t as m_t, the time and space complexities of calculating the cycle-h-index and flow-h-index for each window W^t are $O(\sum_{u \in V_{G^t}}(|N(u)|^2))$ 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 and the collection of the corresponding set takes $O(|N(u)| + |N(v)|)$ time. Hence, the calculation on the h-index values for all the edges in G^t is bounded by $O(\sum_{u \in V_{G^t}}(|N(u)|^2))$. The space complexity is bounded by $O(m_t)$. □

Algorithm 2: Peeling Algorithm with OPT-2

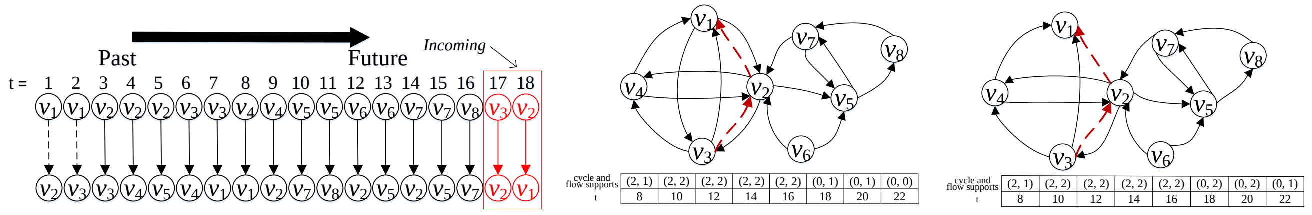
```
Input: a snapshot graph $G = (V_G, E_G)$, a batch of deleted
       edge $\mathcal{E}_d$, a batch of inserted edge $\mathcal{E}_i$, original
       D-truss $DT$, parameters $k_c$ and $k_f$, query vertices $Q$
Output: updated $(k_c, k_f)$-truss containing $Q$
1  for $e_i \in \mathcal{E}_d$ do
2      $G \leftarrow G \setminus \{e_i\}$;
3  if all edges in $\mathcal{E}_d$ are not in $DT$ then
4      $D_1 \leftarrow DT$;
5  else
6      $D_1 \leftarrow$ Peeling($DT, k_c, k_f, \emptyset$);
7  for $e_i \in \mathcal{E}_i$ do
8      $G \leftarrow G \cup \{e_i\}$;
9  Let queue $P \leftarrow \emptyset$, $S \leftarrow \emptyset$;
10 for $e_i \in \mathcal{E}_i$ do
11     if $csup_G(e_i) \geq k_c$ and $fsup_G(e_i) \geq k_f$ then
12         $P \leftarrow P \cup \{e_i\}$;
13 while $P \neq \emptyset$ do
14     Pop an edge $e'$ from $P$;
15     if $csup_G(e') \geq k_c$ and $fsup_G(e') \geq k_f$ then
16         $S \leftarrow S \cup \{e'\}$;
17     for $e_j$ that can form a triangle with $e'$ do
18         if $e_j \notin P$ and $e_j \notin S$ then
19             $P \leftarrow P \cup \{e'\}$;
20 $D_2 \leftarrow$ Peeling($S, k_c, k_f, \emptyset$);
21 $DT \leftarrow D_1 \cup D_2$;
22 if $Q \subset DT$ then
23     Return $DT$;
24 Return $\emptyset$;
```

4.3 OPT-2: *BFS-based Update*

The first optimization employs the cycle-h-index and flow-h-index to prune unqualified edges. In the second optimization, we focus on smaller subgraphs to retrieve the D-truss community based on the following two observations. (1) When edges are deleted from the digraph, the new D-truss must be contained in the original D-truss. Thus, we can peel the original D-truss to obtain the updated D-truss. (2) When new edges are inserted into the digraph, the new D-truss (if exists) must be contained within a subgraph S where edges are triangle-connected to the inserted edges. Thus, we only need to peel S instead of the entire digraph. Based on the above observations, the basic idea of the second optimization is as follows. When the window slides, we first delete the expired edges and maintain D-truss D_1. Then, we insert new edges and form a subgraph S where edges are triangle-connected to the inserted edges, and their cycle and flow supports are greater than k_c and k_f, respectively. Afterwards, we peel S to get D-truss D_2. Finally, $D_1 \cup D_2$ is the final D-truss community. Algorithm 2 describes the peeling algorithm with the second optimization.

For edge deletions, there are two cases. If the deleted edge is within the original D-truss community, we remove the edge from the D-truss and use Algorithm 1 to peel the remaining subgraph to obtain an updated D-truss (lines 5-6). Otherwise, no updates are made to the original D-truss (lines 3-4).

For edge insertions, we first update the snapshot graph with newly inserted edges (lines 7-8). Then, we identify the new edges that have sufficient supports and treat them as seeds (lines 10-12). Next, we perform a BFS search to get the graph S induced by the


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
v1 v2 v3 v4 v5 v6 v7 v8
cycle and flow supports | (2, 1) | (2, 2) | (2, 2) | (2, 2) | (2, 2) | (0, 1) | (0, 1) | (0, 0)
t | 8 | 10 | 12 | 14 | 16 | 18 | 20 | 22
cycle and flow supports | (2, 1) | (2, 2) | (2, 2) | (2, 2) | (2, 2) | (0, 2) | (0, 2) | (0, 1)
t | 8 | 10 | 12 | 14 | 16 | 18 | 20 | 22

(a) Streaming graph

(b) Snapshot graph G and lifetime supports of (v_4, v_1) $(t = 16)$

(c) Snapshot graph G and lifetime supports of (v_4, v_1) $(t = 18)$

Figure 4: An illustrative example of lifetime support-based algorithm ($\tau = 16$ and $\beta = 2$)

edges that (1) have supports of $csup_G(e) \geq k_c$ and $fsup_G(e) \geq k_f$, and (2) are triangle-connected to the new edges, i.e., reachable through a set of triangles starting from the newly inserted edges (lines 13-19). Finally, the algorithm continues to invoke Algorithm 1 to peel the subgraph S (line 20). The final community answer is updated as $D_1 \cup D_2$ (line 21).

EXAMPLE 4.2. *Consider the example in Figure 2. Let $k_c = k_f = 1$, $\tau = 16$, and $\beta = 1$. We assume that the window slides from $t = 16$ to $t = 17$. Then, the edge (v_1, v_2) is deleted from the digraph and the edge (v_9, v_5) is inserted into the digraph. First, we handle (v_1, v_2), which is within the original (1, 1)-truss H_1 shown in Figure 2(b). We directly peel H_1 to get the updated (1, 1)-truss H_2, as shown in Figure 2(c). Next, we handle (v_9, v_5). Since none of the edges are triangle connected with (v_9, v_5), we do not need to peel the digraph. Thus, H_2 is the final D-truss community.*

THEOREM 4.6. ***(Time and Space Complexities)*** *Let G^t be the digraph in the window W^t, m_t be the number of edges in G^t, Algorithm 2 takes $O(m_t^{1.5})$ time and $O(m_t)$ space.*

PROOF. The time cost for edge deletions is determined by the peeling process in line 6, which has a time complexity bounded by $O(m_t^{1.5})$. The collection of the peeling input, i.e., S, requires $O(m_t)$ time, and the peeling process in line 20 takes $O(|S|^{1.5})$ time. Thus, the overall time complexity for edge insertions is $O(|S|^{1.5} + m_t)$. Consequently, the running time of Algorithm 2 is $O(m_t^{1.5})$. The space complexity is $O(m_t)$. □

4.4 OPT-3: *Lifetime Support Prediction*

OPT-1 and OPT-2 mainly reduce the size of the digraph for peeling to improve efficiency. However, both of them overlook the temporal information of edges in the streaming digraph, which provides insight into future changes. Specifically, when a new edge e arrives, we can record its arrival time. Additionally, by utilizing the sliding window size, we can easily determine e's expired time, which allows us to predict its cycle and flow supports in future windows by considering only the edges in the current window. Thus, we propose the third optimization, namely lifetime support prediction.

We start by introducing the concept of *lifetime support* of an edge before presenting the details of OPT-3.

DEFINITION 4.3. ***Lifetime Support****. Given a sliding window size τ and a stride β, assume that an edge $e = (u, v)$ arrives at the time point t_i. The lifetime supports of e consist of the cycle and flow supports of e for all time points $t_i + j \cdot \beta$, where $j \in [0, \lceil \frac{\tau}{\beta} \rceil - 1]$.*

Algorithm 3: Peeling Algorithm with OPT-3

```
Input: a snapshot digraph $G$, 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 $Q$
Let $S \leftarrow \emptyset$;
for $e_i \in \mathcal{E}$ do
    initialize $e_i.lifetime\_supports$;
$G \leftarrow G \cup \mathcal{E}$;
for $e_i \in \mathcal{E}$ do
    for $(e_0, e_1)$ forms cycle triangles with $e_i$ do
        $csup(e_0) \leftarrow csup(e_0) + 1$, $csup(e_1) \leftarrow csup(e_1) + 1$;
        update $lifetime\_supports$ for $e_0$, $e_1$ and $e_i$;
    for $(e_0, e_1)$ forms flow triangles with $e_i$ do
        $fsup(e_0) \leftarrow fsup(e_0) + 1$, $fsup(e_1) \leftarrow fsup(e_1) + 1$;
        update $lifetime\_supports$ for $e_0$, $e_1$ and $e_i$;
for $e_i \in E_G$ do
    $j \leftarrow \lceil \frac{\tau}{\beta} \rceil$ - $\lceil \frac{t(e_i) - W_s^i}{\beta} \rceil + 1$ ;
    if $e_i.lifetime\_supports[j].first \geq k_c$ and
      $e_i.lifetime\_supports[j].second \geq k_f$ then
        $S \leftarrow S \cup \{e_i\}$;
$DT \leftarrow$ Peeling($S, k_c, k_f, Q$);
Return $DT$;
```

By tracking the lifetime supports of each edge, we can predict the cycle and flow supports for a series of future windows. The input digraph for the peeling algorithm consists of the edges whose cycle and flow supports are greater than k_c and k_f in future windows. Besides, since no updates are needed to handle expired edges, the efficiency of the peeling-based algorithm can be improved.

Based on the concept of lifetime support, the third optimization OPT-3 works as follows. We maintain the lifetime supports for each edge in the window. As the window slides, we update the lifetime supports based on the inserted new edges. Specifically, for each new edge and its neighboring edges, we increment their cycle and flow supports for all future time points where those edges can still exist. The principle for the support update is that when calculating the supports for a future time point, we consider only the edges that are present in the current window and still exist at that future time point. Based on the lifetime supports, we identify the edges with supports greater than k_c and k_f. These identified edges serve as the input for Algorithm 1.

Based on the above discussion, we outline the peeling-based algorithm with OPT-3 in Algorithm 3. Given a set of newly inserted edges $\mathcal{E}$, we first initialize the lifetime supports for each edge e_i (lines 2-3). Then, for all newly inserted edges and the edges that form triangles with them, we update their lifetime supports (lines 5-11). Based on edges' lifetime supports, we find the subgraph S whose cycle and flow supports are greater than k_c and k_f, respectively, and take S as the input digraph for peeling (lines 12-15).

Example 4.3. *Figure 4 illustrates the lifetime support-based algorithm. Suppose we want to update the lifetime supports of edge (v_4, v_1) and update the community from time point 16 to 18 with the streaming graph given in Figure 4(a). Figures 4(b) and 4(c) show the snapshot graph taken at time points 16 and 18, respectively. The lifetime supports for (v_4, v_1) are shown under each snapshot graph, and the tuples in the first row represent (cycle support, flow support) of (v_4, v_1). With newly inserted edges (v_2, v_1) and (v_3, v_2), the lifetime supports of (v_4, v_1) are updated. The cycle and flow supports of (v_4, v_1) in the snapshot graphs at time points 16 and 18 can be obtained directly from the corresponding lifetime supports. All the edges whose cycle and flow supports are greater than k_c and k_f form the subgraph for peeling in Algorithm 1.*

Theorem 4.7. ***(Time and Space Complexities)*** *Let G^t be the digraph in the window W^t, δ be the maximal cycle support/flow support of an edge in $\mathcal{E}$, S be the subgraph where each edge satisfies $csup_{G^t}(e) > k_c$ and $fsup_{G^t}(e) > k_f$, the time and space complexities of Algorithm 3 for each window W^t are $O(|\mathcal{E}| \cdot \delta \cdot \lceil \frac{\tau}{\beta} \rceil + |S|^{1.5})$ 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 supports takes $O(|\mathcal{E}| \cdot \delta \cdot \lceil \frac{\tau}{\beta} \rceil)$ time, and the peeling process takes $O(|S|^{1.5})$ time. So the total time complexity is $O(|\mathcal{E}| \cdot \delta \cdot \lceil \frac{\tau}{\beta} \rceil + |S|^{1.5})$. Since each edge takes $O(\lceil \frac{\tau}{\beta} \rceil)$ space to store their lifetime supports, the total space complexity is $O(\lceil \frac{\tau}{\beta} \rceil \cdot m_t)$. □

5 ORDER-BASED ALGORITHM

The peeling-based algorithm needs to iteratively delete edges to obtain the community. Although we have proposed three optimizations to reduce the size of the digraph for the peeling algorithm, the resulting graph may still be quite large. This can potentially impact the performance of the peeling algorithm. In this section, we propose an order-based D-truss community search algorithm. The rationale behind this approach is that a streaming graph can be viewed as a sequence of ordered edges based on their time information. By leveraging the properties of D-truss, we can devise a rule to re-order the streaming graph. This allows us to directly identify the D-truss from the order itself, eliminating the need to peel the digraph from scratch, which can improve efficiency.

5.1 D-truss Peeling Order and Layers

We first introduce the concept of *D-truss peeling order*.

Definition 5.1. ***D-truss Peeling Order****. The D-truss peeling 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 D-truss peeling order $E_{\preceq}$ records the process of (k_c, k_f)-truss computation. Note that there may be multiple orders of edge deletions in the peeling process, and any one of them can be used as the D-truss peeling order. For convenience, we use $E_{e_{\preceq}}$ to denote the set of edges appearing after e in $E_{\preceq}$, i.e., $E_{e_{\preceq}} = \{e' \mid e \preceq e'\}$.

It is worth mentioning that the D-truss peeling order can be generated by invoking the peeling algorithm. Our order is different from the *cycle decomposition order* proposed in the previous work [50]. In [50], the order-based D-index uses the cycle decomposition order (i.e., CD order), which is the sequence of deleting edges based on the cycle truss number for a $(0, k_{f_i})$-truss. The D-index in [50] consists of multiple CD orders for all possible k_{f_i} values. Each order considers the two types of trussness separately during maintenance. In contrast, our proposed order is a one-dimensional index that considers both cycle support and flow support simultaneously. Specifically, an edge is removed if it fails to meet either the cycle support or flow support criterion, and its position in the order is identified. 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 two concepts of *layers in D-truss peeling order* and *layer number.*

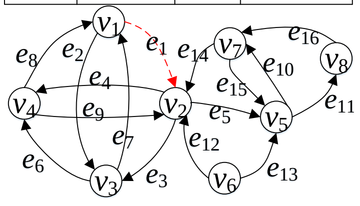
Definition 5.2. ***Layers in D-truss peeling order.*** *Given a D-truss peeling order $E_{\preceq}$ for a digraph G and the community query parameters k_c and k_f, the edges in E_G can be uniquely accommodated in different layers $\{L_1, L_2, \ldots, L_\mu\}$. The j-th layer is the set of edges satisfying $L_j = \{e \in E_G \mid \{csup_{H_j}(e) < k_c \ \vee \ fsup_{H_j}(e) < k_f\} \ \wedge \ \{csup_{H_{j-1}}(e) \geq k_c \ \wedge \ fsup_{H_{j-1}}(e) \geq k_f\}\}$, where $H_j = E_G \setminus \bigcup_{i=1}^{j-1} L_i$. The initial layer is $L_1 = \{e \in E_G \mid csup_G(e) < k_c \ \vee \ fsup_G(e) < k_f\}$.*

Definition 5.3. ***Layer number $\mathcal{L}$.*** *Given an edge e, the layer number of e is defined as $\mathcal{L}(e) = l$, where $e \in L_l$.*

The layer number $\mathcal{L}(e)$ represents the number of the round in which e is peeled from the original digraph during the community finding process. 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 D-truss peeling order, the concept of layers provides a more coarse-grained way to record the sequence of the edge removals 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 $\{L_1, L_2, \cdots, L_\mu\}$. The edges in the same layer can have an arbitrary order, i.e., if e and e' both belong to L_i for $\forall i (1 \leq i \leq \mu)$, then both $e \preceq e'$ and $e' \preceq e$ hold.

Example 5.1. *We use Figure 5(a) to illustrate the D-truss peeling order. Assume that $k_c = k_f = 1$. The red dashed line indicates the edge to be inserted. The layers of G without considering edge e_1 are shown at the top of Figure 5(a). Initially, we have $e_2, e_4, e_7, e_9, e_{12}, e_{13}$ with cycle/flow support values smaller than parameters $k_c = k_f = 1$. 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 kept in L_3. Therefore, the D-truss peeling order is $E_{\preceq} = \{L_1, L_2, L_3\}$, and the maximum layer number is $\mu = 3$.*

$k_c =$ $k_f = 1$	L_1	L_2	L_3
	$e_2, e_4, e_7, e_9, e_{12}, e_{13}$	e_3, e_6, e_8	$e_5, e_{10}, e_{11}, e_{14}, e_{15}, e_{16}$



(a) Graph G

Q: $\mathbf{e_2}$, e_4, e_7, e_9, e_3, e_8

L_1	L_2	L_3	L_4
$e_4, e_7, e_9, e_{12}, e_{13}$	e_3, e_6, e_8	e_2	$e_1, e_5, e_{10}, e_{11}, e_{14}, e_{15}, e_{16}$

Q: $\mathbf{e_4}$, e_7, e_9, e_3, e_6, e_8

L_1	L_2	L_3	L_4
e_7, e_9, e_{12}, e_{13}	e_3, e_6, e_8	e_2, e_4	$e_1, e_5, e_{10}, e_{11}, e_{14}, e_{15}, e_{16}$

Q: $\mathbf{e_7}$, e_9, e_3, e_6, e_8

L_1	L_2	L_3	L_4
e_9, e_{12}, e_{13}	e_3, e_6, e_8	e_2, e_4, e_7	$e_1, e_5, e_{10}, e_{11}, e_{14}, e_{15}, e_{16}$

Q: $\mathbf{e_9}$, e_3, e_6, e_8

L_1	L_2	L_3	L_4
e_{12}, e_{13}	e_3, e_6, e_8	e_2, e_4, e_7, e_9	$e_1, e_5, e_{10}, e_{11}, e_{14}, e_{15}, e_{16}$

(b) Maintaining Order $E_{\preceq}$: part 1

Q: $\mathbf{e_3}$, e_6, e_8

L_1	L_2	L_3	L_4
e_{12}, e_{13}	e_6, e_8	e_2, e_3, e_4, e_7, e_9	$e_1, e_5, e_{10}, e_{11}, e_{14}, e_{15}, e_{16}$

Q: $\mathbf{e_6}$, e_8

L_1	L_2	L_3	L_4
e_{12}, e_{13}	e_8	$e_2, e_3, e_4, e_6, e_7, e_9$	$e_1, e_5, e_{10}, e_{11}, e_{14}, e_{15}, e_{16}$

Q: $\mathbf{e_8}$

L_1	L_2
e_{12}, e_{13}	$e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}, e_{11}, e_{14}, e_{15}, e_{16}$

(c) Maintaining Order $E_{\preceq}$: part 2

Figure 5: An illustrative example of the order-based algorithm (the edges being processed are highlighted in blue)

5.2 Property Analysis of Layers in Streaming Graphs

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

Property 5.1. *Given an order $E_{\preceq}$ for digraph G with μ layers, if we denote $H_j = E_G \setminus \bigcup_{i=1}^{j-1} L_i$ and $H_1 = E_G$, then the following properties hold:*

(1) For an edge $e \in L_j$, it holds that: $\{csup_{H_j}(e) < k_c \vee fsup_{H_j}(e) < k_f\} \wedge \{csup_{H_{j-1}}(e) \geq k_c \wedge fsup_{H_{j-1}}(e) \geq k_f\}$.

(2) L_μ is the D-truss 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. Property 5.1(1) is obvious based on Definition 5.2. We prove Property 5.1(2) in the following. For a digraph G, if there is a D-truss in G, then according to the peeling algorithm, all the edges with enough cycle and flow supports will be returned as the community. If we denote the returned community as $\mathcal{G}$, then for the edges in $\mathcal{G}$, we have $csup_{\mathcal{G}} \geq k_c$ and $fsup_{\mathcal{G}} \geq k_f$. 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) can help us update 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 incurs a very light cost, we focus on how to maintain the layers and retrieve L_μ. The main idea of the order maintenance algorithm is to put edges into the layer that satisfies Property 5.1(1) until all edges are updated. The last layer of the updated order is ultimately retrieved as the D-truss community based on Property 5.1(2). Take the edge insertion as an example. When a new edge is inserted, we place the edge into a layer initially and adjust the layers for all of its neighboring edges. Specifically, at the beginning of processing, we place the new edge into the layer such that Property 5.1(1) is satisfied. Here, a special case may happen, that is, the new edge has supports in L_μ greater than the query parameters. In this case, we directly place it at the beginning of L_μ. Note that we should always consider all the neighboring edges of the newly inserted edge in the subsequent processing, regardless of whether this special case occurs or not. There is a key challenge, i.e., finding neighboring edges that may have their layers adjusted. We have the following theorems to assist in identifying the affected edges.

Theorem 5.1. *All the neighboring edges of the newly inserted/deleted edge should be considered and processed in Algorithm 4.*

Proof. The proof of Theorem 5.1 is straightforward. Since all the neighbors of the newly inserted edge have their supports 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 consider two types of edges: those in the layers preceding L_{o+1} and those preceded by L_n. Let L_f be a layer preceding L_{o+1}. Then, for edges contained in L_f, their supports in H_f are not affected by the increment of the layer of e_i, as e_i is always a part of H_f. Therefore, the layers of those edges cannot be incremented. Similarly, let L_b be a layer preceded by L_n. For those edges in L_b, their supports in H_b remain unaffected, and their layers remain the same. Hence, Theorem 5.2 holds. □

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

In this subsection, we first propose the order maintenance algorithms for streaming digraphs and then develop the corresponding techniques for D-truss community search. We handle both edge insertions and edge deletions in the order maintenance process. Therefore, we first present the order maintenance algorithm for an edge insertion.

The procedure for order maintenance with an edge insertion is presented in Algorithm 4. We introduce the notation $H_j = E_G \setminus \bigcup_{i=1}^{j-1} L_i$, where H_1 is E_G, for the ease of presentation. It is

Algorithm 4: 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 $\{csup_{H_{i-1}}(e_i) \geq k_c \wedge$
       $fsup_{H_{i-1}}(e_i) \geq k_f\} \wedge \{csup_{H_i}(e_i) < k_c \vee$
       $fsup_{H_i}(e_i) < k_f\}$;
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
           $\{csup_{H_{i'-1}}(e') \geq k_c \wedge fsup_{H_{i'-1}}(e') \geq k_f\} \wedge$
           $\{csup_{H_{i'}}(e') < k_c \vee fsup_{H_{i'}}(e') < k_f\}$;
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 $E'_{\preceq}$;
```

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 [6, 14]. The algorithm works as follows. First, a new layer is created (lines 1-4) or the newly inserted edge e is placed in layer L_i based on Property 5.1(1) (lines 5-6). Second, all neighboring edges of e are added to a queue (lines 7-10). Third, for each edge in the queue, it is either placed into a layer that satisfies Property 5.1(1) (lines 20-21), or a new layer is generated to accommodate it (lines 15-17). After placing an edge into a new layer, it is removed from the queue, and all its neighboring edges satisfying Theorem 5.2 are added to the queue (lines 22-24). This process is repeated until the queue becomes empty. It is important to note that the edges in the queue are processed from left to right. Some corner cases may arise during the execution of the algorithm. First, certain layers may be merged with L'_μ and subsequently removed from $E_{\preceq}$, resulting in a decrease in the total

Algorithm 5: 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 4($G, e_d, k_c, k_f, E_{\preceq}$);
3  for $e_i \in \mathcal{E}_i$ do
4      $E_{\preceq'} \leftarrow$ Algorithm 4($G, e_i, k_c, k_f, E_{\preceq'}$);
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 $\emptyset$;
```

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. Second, new layers may be generated (lines 2-3, 15-16). This is due to the adjustment of the edges' layer numbers, during which some edges have sufficient supports in $L_{\mu-1} \cup L_\mu$ but lack the necessary supports in L_μ. Consequently, a new layer is created "between" the original $L_{\mu-1}$ and L_μ to satisfy Property 5.1(1).

The order maintenance algorithm for edge deletions is similar to that of the insertion case, but with a few differences. In specific, given an edge e_i, if its layer is recently decremented from L_o to L_n ($L_n \leq L_o$), then only the 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 becomes L_n to L_o. The proof for the edge deletion can be done similarly to Theorem 5.2.

Equipped with the order maintenance algorithms for edge insertions/deletions, we now present the order-based community search algorithm, as outlined in Algorithm 5. We 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 an edge deletion is similar. Note that the peeling-based algorithm and the order-based algorithm cannot be combined. Specifically, the peeling-based algorithm iteratively removes edges that violate the flow and cycle support constraints. This removal process naturally generates an order that represents the sequence of edge removal. On the other hand, the order-based algorithm is designed to maintain and simulate the order generated by the peeling process. In essence, these two algorithms share the same underlying principle but have different implementations.

Example 5.2. *We use the digraph in Figure 5 to illustrate Algorithm 4. Let $k_c = k_f = 1$. Figures 5(b) and 5(c) show the order maintenance process after the edge e_1 (marked as red) is inserted. Note that the edges in the current queue Q are shown on the left of each order, and the edge to be processed is marked as blue. The original layers are shown at the top of Figure 5(a). Note that after we insert edge e_8 into L_3 from L_2, the condition in line 18 of Algorithm 4 is fulfilled and this layer is appended to L'_μ. The updated layers are shown in Figure 5(c).*

Theorem 5.3. ***(Time and Space Complexities)*** *Let G^t be the digraph 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 4 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 μ times. As each layer changes costs δ time, the total time complexity is $O(C * \mu * \delta)$. 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 networks in our experiments. Table 1 summarizes the statistics of these graphs. Note that k_c^{max} and k_f^{max} are the maximum 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**: [50] 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 networks whenever the window slides.
- **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. For each dataset, the settings for k_c and k_f are determined based on the intrinsic characteristics of the graph. They are set to be no larger than k_c^{max} and k_f^{max}, respectively, while ensuring that the resulting community is not empty. Table 2 summarizes the default parameter settings. For each experiment, we report the throughput, which represents the number of incoming edges processed per second. In each experiment, we run 100 queries and report the average throughput. If an algorithm cannot be completed within 10 days, it is denoted by INF.

[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	1.9**K**	59.8**K**	63.0	6	8
Sx-Mathoverflow	SX	24.8**K**	506.5**K**	40.8	7	6
Ask-Ubuntu	UT	159.3**K**	964.4**K**	12.1	2	3
Amazon	AM	334.8**K**	1.9**M**	11.1	2	2
DBLP	DP	317.1**K**	2.1**M**	13.2	10	10
Flickr	FC	2.3**M**	33.1**M**	28.8	3	2
Stack-Overflow	SO	2.6**M**	63.5**M**	48.8	1	4
UK-2002	UK2	18.5**M**	298.1**M**	32.2	7	5

Table 2: Default parameter settings

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

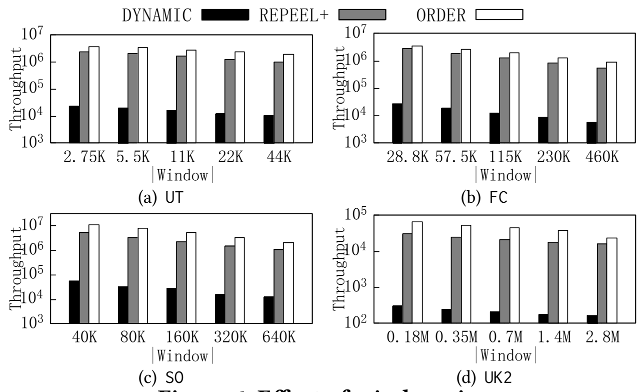


Figure 6: Effect of window size

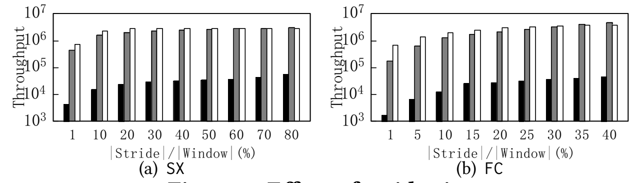
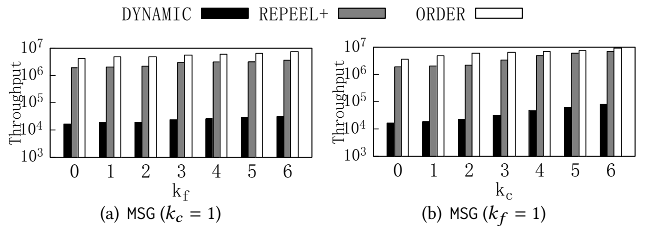


Figure 7: Effect of stride size

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) MSG ($k_c = 1$) (b) MSG ($k_f = 1$)

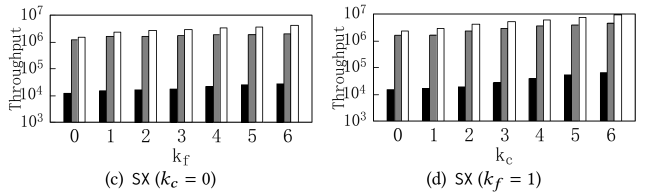
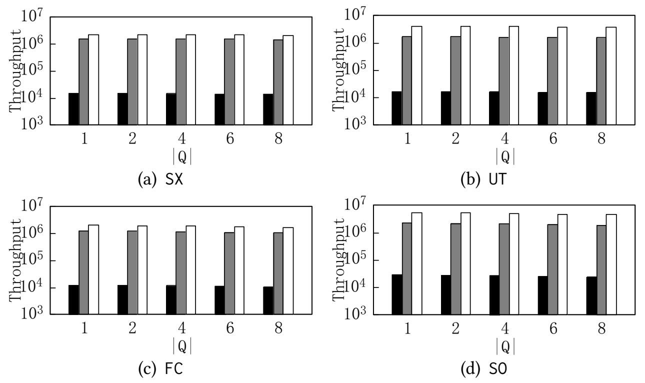

(c) SX ($k_c = 0$) (d) SX ($k_f = 1$)

Figure 8: Effect of k_c and k_f



(a) SX (b) UT (c) FC (d) SO

Figure 9: Effect of $|Q|$

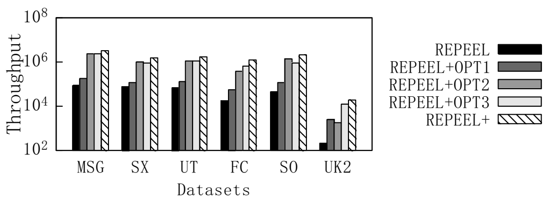
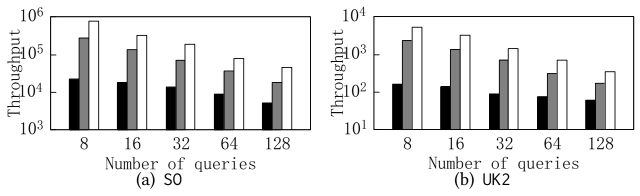


Figure 10: Effect of optimizations



(a) SO (b) UK2

Figure 11: Effect of concurrent queries

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. As shown in Figure 7, larger strides result in higher throughput. This is because larger strides mean that 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 the stride size is larger than 80% and 35% of the window size (i.e., 4.8K and 40.3K edges, respectively), REPEEL+ outperforms ORDER in terms of efficiency. This phenomenon can be attributed to the fact that 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 main overhead 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.

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 dataset. 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 $|Q|$. 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 as $|Q|$ 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 incurs only a slight overhead. As a result, the change of $|Q|$ does not significantly impact the 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 shown 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 SO, while OPT-2 has a better performance on UK2 compared to OPT-1.

Exp-6: Effect of concurrent queries. In practical applications, there are scenarios where simultaneous queries are performed for different parameter combinations. Therefore, we evaluate the performance of our proposed algorithms compared to DYNAMIC when processing multiple queries simultaneously. Our proposed algorithms can support multiple queries as follows. Specifically, for REPEEL+, we can re-peel the directed graph for each pair of k_c and k_f to obtain different communities; for ORDER, we can maintain a separate order for each pair of k_c and k_f, and as the window slides, we can update each order to return different communities. As for DYNAMIC, which retains the entire trussness sets for each edge, we maintain a D-index and execute all the queries based on the updated index [50]. The results are shown in Figure 11. As the number of queries increases from 8 to 128, the throughput of all the algorithms decreases. However, the running time of DYNAMIC remains relatively stable. This is because its primary overhead is

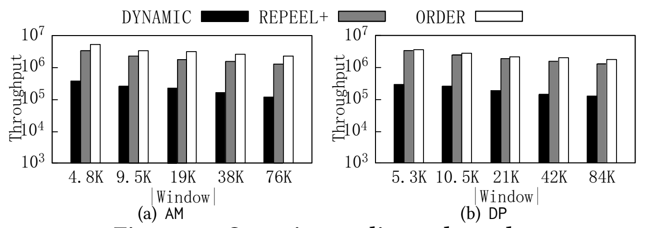


Figure 12: Querying undirected graphs

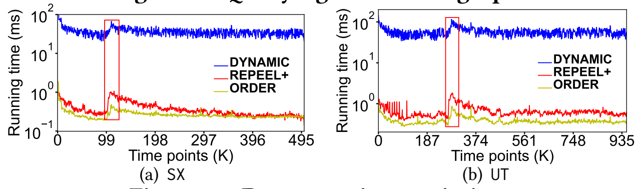


Figure 13: Response time variations

associated with index maintenance, which is a one-time task regardless of the number of queries. Most of the increase in running time is attributed to the execution of the queries themselves. In contrast, for ORDER and REPEEL+, which involve order maintaining or graph peeling for each query, the computational cost grows linearly with the number of queries. Nevertheless, our algorithms consistently outperform DYNAMIC in all cases.

Exp-7: Querying undirected graphs. 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 graph 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 undirected edges. We compare the performance of our algorithms against the baseline across various window sizes. The results, shown in Figure 12, are consistent 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 for undirected graphs.

Exp-8: Examination of response time variations. As the sliding window moves, the response time may exhibit fluctuations. To evaluate such fluctuations, we measure the query processing time at each sliding window during the execution. The experimental results are shown in Figure 13. It is observed that the response time fluctuates as the window slides. However, DYNAMIC exhibits larger performance fluctuations compared to the other two algorithms. This can be attributed to the longer execution time of DYNAMIC, which leads to greater variations in its performance. Overall, the variations of all three algorithms remain relatively small, except for some time points indicated by red rectangles, which show relatively significant fluctuations. These fluctuations occur because the updates to the edges at these time points result in substantial changes to the underlying graphs and communities, thereby causing larger variations in the response time.

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

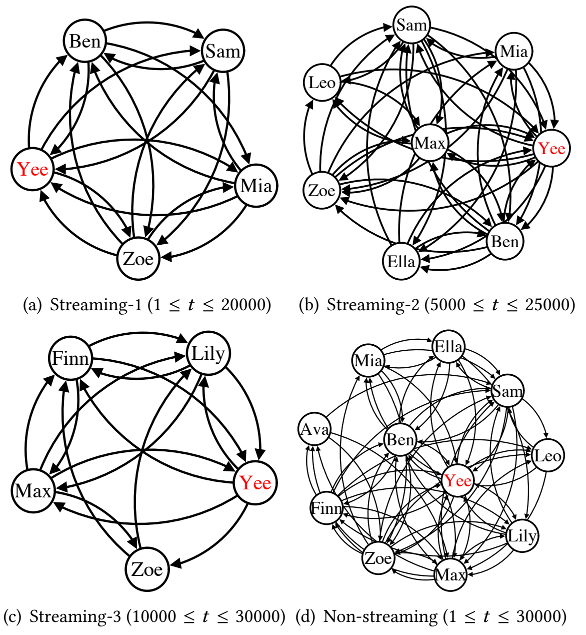

Ben
Sam
Yee
Mia
Zoe
Leo
Max
Ella
Finn
Lily
Ava
(a) Streaming-1 (1 ≤ t ≤ 20000)
(b) Streaming-2 (5000 ≤ t ≤ 25000)
(c) Streaming-3 (10000 ≤ t ≤ 30000)
(d) Non-streaming (1 ≤ t ≤ 30000)

Figure 14: Case study over `Flickr`

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 time points 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 time point 25,000 with the addition of new friends *Max*, *Leo*, and *Ella*. Subsequently, at time point 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 entire time period, which cannot capture the 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-based update, and lifetime support prediction, to improve the performance of the peeling-based algorithm. Moreover, we have introduced the D-truss peeling order and layers in D-truss peeling 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] Jørgen Bang-Jensen and Gregory Z Gutin. 2008. *Digraphs: theory, algorithms and applications.* Springer Science & Business Media.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

[25] 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).

[26] 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.

[27] 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.

[28] 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.

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

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

[31] 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.

[32] Paul Irofti, Andrei Patrascu, and Andra Baltoiu. 2019. Quick survey of graph-based fraud detection methods. *arXiv preprint arXiv:1910.11299* (2019).

[33] Meng Jiang, Peng Cui, Alex Beutel, Christos Faloutsos, and Shiqiang Yang. 2014. Catchsync: catching synchronized behavior in large directed graphs. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining.* 941–950.

[34] 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.

[35] 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.

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

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] Xuankun Liao, Qing Liu, Jiaxin Jiang, Xin Huang, Jianliang Xu, and Byron Choi. 2022. Distributed D-Core Decomposition over Large Directed Graphs. *Proceedings of the VLDB Endowment* (2022), 1546–1558.

[43] 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.

[44] Qing Liu, Xuliang Zhu, Xin Huang, and Jianliang Xu. 2021. Local algorithms for distance-generalized core decomposition over large dynamic graphs. *Proceedings of the VLDB Endowment* 14, 9 (2021), 1531–1543.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] Zitan Sun, Xin Huang, Qing Liu, and Jianliang Xu. 2023. Efficient Star-based Truss Maintenance on Dynamic Graphs. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–26.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.