Querying Cohesive Subgraph regarding Span-Constrained Triangles on Temporal Graphs

Chuhan Hu*, Ming Zhong*†✉, Yuanyuan Zhu*, Tieyun Qian*, Ting Yu†, Hongyang Chen†, Mengchi Liu‡, and Jeffrey X. Yu§
*School of Cyber Science and Engineering & School of Computer Science, Wuhan University, Wuhan, China
†Zhejiang Lab, Hangzhou, China
‡South China Normal University, Guangzhou, China
§The Chinese University of Hong Kong, Hong Kong, China
{huchuhan, clock, yyzhu, qty}@whu.edu.cn, {yuting, hongyang}@zhejianglab.com,
liumengchi@scnu.edu.cn, yu@se.cuhk.edu.hk

2024 IEEE 40th International Conference on Data Engineering (ICDE) | 979-8-3503-1715-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICDE60146.2024.00258

***Abstract*—The recent prosperity of temporal graph research redefines many traditional concepts on static graphs, such as triangle, motif, k-core, etc. Inspired by that, we propose a novel (k, δ)-truss on temporal graphs, which requires its triangles to exist in short enough time windows ever. The (k, δ)-truss satisfies both static and temporal cohesion, while the original k-truss is its special case when $\delta = \infty$. In order to address the (k, δ)-truss query, we propose both index-free and index-based approaches. By leveraging the dual containment relation on (k, δ)-trusses, our indexes can compress all (k, δ)-trusses losslessly into map or tree structures with dramatically less space, so that a specific (k, δ)-truss can be retrieved from indexes in the optimal time. To enable our index to scale to large temporal graphs, we develop two index construction algorithms that can reduce redundant computation significantly, based on truss decomposition and truss maintenance respectively. The experimental results demonstrate that index-based approaches process queries in interactive time and outperform the index-free approach by 2∼4 orders of magnitude, while indexes achieve compression ratios up to 10^{-4}.**

***Index Terms*—temporal graph, cohesive subgraph, truss, triangle, time span, index, query processing**

I. Introduction

Recently, temporal graphs in which each edge is associated with a set of timestamps have drawn intensive research interests, as introduced by [1], [2]. The typical examples of temporal graph are such as social networks [3], transaction networks [4], transportation networks [5], communication networks [6], power networks [7], disease transmission networks [8], etc. In those graphs, the temporality enables a variety of time-relevant constraints in analytics, such as time order, time window, time span, etc.

For temporal graphs, the traditional definitions of cohesive subgraphs such as k-clique, k-truss, and k-core also need to be extended with time-relevant constraints, so that both temporal and topological features can be exploited for comprehensive analysis. For example, there have been a bunch of temporal k-core studies, which mainly fall into two categories. The first is to find primitive k-cores that exist in specific time windows, such as span-core [9], temporal k-core [10], [11], historical k-core [12], temporal $(k, \mathcal{X})$-core [13], etc. The second is to

✉ Ming Zhong is the corresponding author.

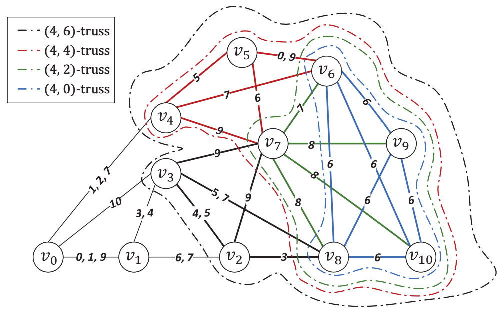
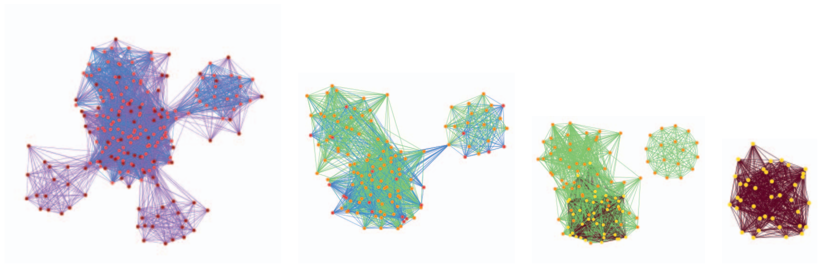


Fig. 1: A running example temporal graph and several (k, δ)-trusses, which forms a "δ-shell" with a fixed k.

study various new temporal k-cores models, such as frequent core [14], persistent core [15], bursting core [16], periodic core [17], continual core [18], reliable core [19], etc.

However, the models and approaches designed for k-core query are not sufficient for k-truss query on temporal graphs. In contrast to migrating them to k-truss directly, like [20] for the first category and [21] for the second category, it is more important to investigate the models and approaches dedicated to temporal k-truss query. As we know, k-truss in which each edge is contained by at least $k - 2$ triangles is defined on top of another more fundamental concept, namely, triangle. Thus, a reasonable definition of temporal k-truss should take the temporal constraint on triangles into consideration.

Actually, many meaningful temporal triangle or motif (note that, triangles can be seen as a kind of motifs sometimes) models [22]–[27] have been proposed recently. These models mainly consider two kinds of temporal constraints. The first is the total or partial time order among edges, which is usually defined on directed temporal graphs, so that a triangle or motif can be seen as a sequence of edges in order of timestamps. Such constraints are too specific and complicated for a basic component of k-truss, and would result in over-tailoring of k-truss. The second is the duration, which is generally measured by the maximum time lag between timestamps of edges.



(a) $(16, \infty)$ (b) $(16, 200)$ (c) $(16, 150)$ (d) $(16, 100)$

(k, δ)-truss	(a)	(b)	(c)	(d)
vertex#	213	130	108	38
edge#	4402	2355	1735	564
triangle#	42683	21738	14978	4670
coefficient	0.72	0.77	0.81	0.85
duration	803	803	803	802

Fig. 2: A case study on (k, δ)-trusses of Email dataset, which demonstrates the effectiveness of δ on improving cohesion. In each truss, we remark the vertices and edges of successive truss with different colors, so that the changes can be observed.

In this paper, we propose a novel definition of (k, δ)-truss for undirected temporal graphs, on top of a kind of "span"-constrained δ-triangles. The rationale is two-fold. Since k-truss considers a triangle as a strong evidence of its vertices are bonded tightly in a community, it is better to consistently guarantee the tightness of triangle but not truss from the perspective of time. Moreover, different from the related works, (k, δ)-truss requires the *minimum time span* but not normal duration of triangles to be no greater than δ, which regulates temporal cohesion beyond static cohesion k. Because a triangle that ever occurs in a short enough period should be more cohesive in the sense of time than another triangle that has the same duration but no interactions close in time between each pair of its vertices, as illustrated in Fig 3.

Let us consider the following empirical (k, δ)-truss queries.

Example 1 (Case Study)**.** *To demonstrate the effectiveness of (k, δ)-truss query, we conduct a case study on Email [28], a communication network between members of a European research institution. Fig 2 illustrates four trusses with $k = 16$ and $\delta = \infty$, 200, 150, and 100 respectively. The (k, δ)-truss query can help us to further tailor the static k-truss (a) in time dimension. We can see that, the stricter temporal cohesion indeed makes the truss structure compacter. The k-truss (a) is composed of several smaller communities. With the decrease of δ, the successive (k, δ)-trusses (b), (c), and (d) become more and more clustered. As an evidence, the clustering coefficient increases from 0.72 to 0.85 gradually. In contrast, the duration of whole trusses does almost not change, which means we will not find the compacter trusses like (b), (c), and (d) by using the duration of truss as temporal constraint.*

In order to address the (k, δ)-truss query, we propose both index-free and index-based approaches. Compared with the index-free approach that performs a straightforward truss decomposition under the constraint of δ, the index-based approach only needs to scan each edge in the result once, thereby being theoretically time-optimal. Since there could be a great number of (k, δ)-trusses in a temporal graph, we use Temporal Containment Index (TC-Index) or Dual Containment Index (DC-Index) to preserve trusses incrementally. Moreover, for large-scale temporal graphs, we develop two scalable index construction algorithms based on two classical paradigms, truss decomposition [29]–[33] and truss maintenance [34]–[38], respectively. Decomposition Based Algorithm (DBA) can compute the incremental edge sets between (k, δ)-truss and $(k, \delta + 1)$-truss, and Maintenance Based Algorithm (MBA) can also compute the incremental edge sets between (k, δ)-truss and $(k + 1, \delta)$-truss.

In summary, our contributions are as follows.

- Inspired by the latest studies on temporal triangle and motif, we formalize a novel (k, δ)-truss query problem on temporal graphs, with respect to a meaningful temporal triangle metric called minimum time span. The (k, δ)-truss considers both static and temporal cohesion, while the static k-truss is its special case when $\delta = \infty$.
- We leverage the dual containment relation on (k, δ)-trusses to design compact indexes that store and retrieve all possible (k, δ)-trusses efficiently. Firstly, we present a map-structured index called TC-Index that preserves the incremental edges in δ dimension. Then, we present a tree-structured index called DC-Index that preserves the globally minimum incremental edges in both k and δ dimensions. Both indexes provide the optimal query efficiency, and DC-Index is space-optimal when query efficiency cannot be degraded.
- To enable proposed indexes to scale to large temporal graphs, we follow the line of truss decomposition and truss maintenance respectively to improve the scalability of index construction. DBA decomposes k-truss gradually by removing triangles in descending order of minimum time span for each k. MBA maintains all edge trussness simultaneously when invalidating triangles in descending order of minimum time span. Both of them can reduce redundant computation in index construction significantly.
- We conduct comprehensive experimental evaluation on eight real-world temporal graphs, on which we observed that triangle counts distribute widely on minimum time span. Our index-based TC-Query and DC-Query process queries in interactive time, and outperform the index-free Online-Query by 2∼4 orders of magnitude. Meanwhile, our indexes can achieve the compression ratio up to 10^{-4}.

We present the preliminaries, index-free approach, index-based approaches, index construction, experimental evaluation,

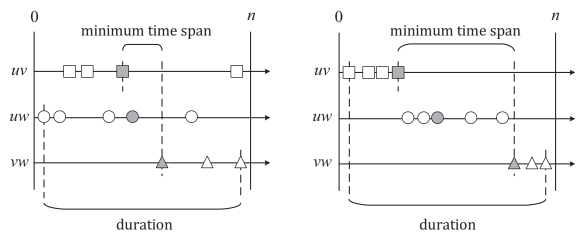


Fig. 3: Abstract comparison of different minimum time spans of a triangle $\Delta = \{u, v, w\}$. For each edge like (u, v) of Δ, its timestamps are marked by a kind of symbols like rectangles in a timeline from 0 until n. The combination of dark symbols of each kind determines the minimum time span of Δ.

related work, and conclusion in the rest sections respectively.

II. Preliminaries

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be an undirected static graph, where $\mathcal{V}$ is a set of vertices and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is a set of edges. For each edge $e = (u, v) \in \mathcal{E}$, the pair of vertices u and v may have multiple interactions at different times. We denote the set of timestamps of interaction between u and v by $\tau_{(u,v)}$. Thus, the temporal edge between u and v can be represented by $(u, v, \tau_{(u,v)})$, and the temporal graph can be represented by $\mathcal{G}^t = (\mathcal{V}, \mathcal{E}, \Gamma)$, where $\Gamma = \{\tau_{(u,v)} | (u, v) \in \mathcal{E}\}$ is the set of nonempty timestamp sets for each edge in $\mathcal{E}$. Fig 1 illustrates a temporal graph as our running example. Without loss of generality, we use consecutive natural numbers from 0 until n to denote timestamps in a temporal graph.

Then, let us consider the reasonable definition of k-truss for temporal graphs. Given a static graph $\mathcal{G}$, the k-truss, denoted by T_k, is defined as the maximal subgraph of $\mathcal{G}$ in which each edge is contained by at least $k - 2$ triangles, where $k \geq 2$ is a user-specified integer that represents topological cohesion of the subgraph. Obviously, the definition of k-truss is closely associated with that of triangle, a more fundamental concept for graph. Inspired by [22]–[27], we propose a novel and meaningful definition of temporal triangle in the context of truss study as follows.

Definition 1 (Minimum Time Span). *Given a temporal graph $\mathcal{G}^t$, for a triangle $\Delta = \{u, v, w\}$ of $\mathcal{G}^t$ with $u, v, w \in \mathcal{V}$ and $(u, v), (v, w), (w, u) \in \mathcal{E}$, its minimum time span* $\mathrm{mts}(\Delta)$ *represents the shortest duration of time window in which each two vertices have interaction, namely,* $\mathrm{mts}(\Delta) = \min\{\max\{|t_1 - t_2|, |t_2 - t_3|, |t_3 - t_1|\} \mid t_1 \in \tau_{(u,v)}, t_2 \in \tau_{(v,w)}, t_3 \in \tau_{(w,u)}\}$.

Definition 2 (δ-Triangle). *Given a threshold δ of minimum time span, a triangle Δ is called a δ-triangle if* $\mathrm{mts}(\Delta) \leq \delta$.

Intuitively, δ-triangles represent tight bonds in the sense of both topology and time, which require the involved vertices to interact with each other during at least one same short period. As illustrated in Fig 3, the left triangle is considered as a tighter bond than the right one from the perspective of time, though them have the same duration. Because, in the shorter minimum time span, all three vertices may participate in a same event. In contrast, the right triangle represents a typical counterexample. When two of the vertices have contacts, neither of them interacts with the other vertex. Like in a social network, I know both of you but do not know you know each other, which means the three of us are not that close.

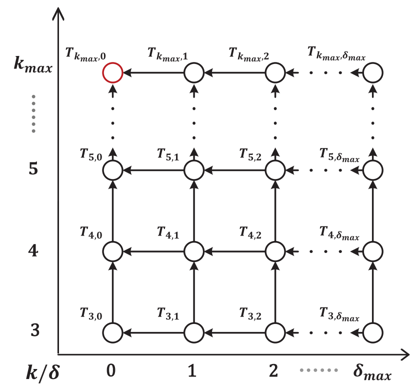


Fig. 4: An illustration of (k, δ)-truss graph of a temporal graph, where each arrow $T_{k,\delta} \rightarrow T_{k',\delta'}$ denotes that $T_{k',\delta'} \subseteq T_{k,\delta}$.

On top of δ-triangle, we give the definitions of δ-support of temporal edge and (k, δ)-truss of temporal graph as follows.

Definition 3 (δ-Support). *Given a temporal graph $\mathcal{G}^t$ and an integer $\delta \geq 0$, the δ-support of an edge $e = (u, v) \in \mathcal{E}$, denoted by* δ-$\mathrm{sup}(e)$*, is the number of δ-triangles that contain e, namely,* $|\{\Delta | u, v \in \Delta, \mathrm{mts}(\Delta) \leq \delta\}|$.

Example 2. *Consider an edge $e = (v_2, v_8)$ in Fig 1. The triangles containing e are $\Delta_1 = \{v_2, v_3, v_8\}$ and $\Delta_2 = \{v_2, v_7, v_8\}$. We have* $\mathrm{mts}(\Delta_1) = 2$ *and* $\mathrm{mts}(\Delta_2) = 6$*, so that* δ-$\mathrm{sup}(e)$ *is 2 if $\delta \geq 6$, 1 if $2 \leq \delta < 6$, or 0 otherwise.*

Definition 4 ((k, δ)-Truss). *Given a temporal graph $\mathcal{G}^t$, an integer $k \geq 2$, and an integer $\delta \geq 0$, the (k, δ)-truss of $\mathcal{G}^t$, denoted by $T_{k,\delta}$, is defined as the maximal subgraph of $\mathcal{G}^t$ in which the δ-support of each edge e is no less than $k - 2$, namely,* δ-$\mathrm{sup}(e) \geq k - 2$ *in the temporal subgraph $T_{k,\delta}$.*

Example 3. *In Fig 1, given $k = 4$, we use colored dashed lines to remark the (k, δ)-trusses with different δ. The edges surrounded by black dashed line comprise the largest $(4, 6)$-truss. When δ is decreased to 4, the edges such as (v_2, v_8) are excluded due to inadequate δ-support, and the edges surrounded by red dashed line comprise the smaller $(4, 4)$-truss. Similarly, with the decrease of δ, the (k, δ)-truss shrinks gradually, like the $(4, 2)$-truss surrounded by green dashed line and $(4, 0)$-truss surrounded by blue dashed line.*

More importantly, similar to the static cohesive subgraphs like k-truss and k-core, (k, δ)-truss also has the containment property, which is however dual with respect to both k and δ. As illustrated in Fig 4, each (k, δ)-truss is contained

```
Algorithm 1: Online-Query
  Input: a temporal graph $\mathcal{G}^t$, a support threshold $k$, a
         minimum time span threshold $\delta$
  Output: the $(k,\delta)$-truss $T_{k,\delta}$ of $\mathcal{G}^t$
1 compute $\delta$-sup$(e)$ for each edge $e \in \mathcal{E}$;
2 push all the edges to $Q$;
3 $T_{k,\delta} \leftarrow \mathcal{G}^t$;
4 while $\exists e \in Q$ such that $\delta$-sup$(e) < k-2$ do
5     $e \leftarrow Q$.pop();
6     for each $\Delta$ containing $e$ in $\mathcal{G}^t$ do
7         if $\mathrm{mts}(\Delta) \le \delta$ then
8             let $e'$ and $e''$ be the other two edges of $\Delta$;
9             $\delta$-sup$(e') \leftarrow \delta$-sup$(e') - 1$;
10            $\delta$-sup$(e'') \leftarrow \delta$-sup$(e'') - 1$;
11    $T_{k,\delta} \leftarrow T_{k,\delta}/e$;
12 return $T_{k,\delta}$;
```

by both $(k-1,\delta)$-truss and $(k,\delta+1)$-truss directly. Such containment properties are the keys to developing efficient algorithms for retrieving cohesive subgraphs, such as truss decomposition [29] and core decomposition [39]. The dual containment property is formally defined as follows.

Property 4.1 (Dual Containment). *For any two (k,δ)-trusses $T_{k,\delta}$ and $T_{k',\delta'}$ of a temporal graph $\mathcal{G}^t$, $T_{k,\delta}$ is contained by $T_{k',\delta'}$ as a subgraph, denoted by $T_{k,\delta} \subseteq T_{k',\delta'}$, if $k' \le k$ and $\delta' \ge \delta$.*

PROOF. The result can be proved by separately showing that (i) $k' \le k \Longrightarrow T_{k,\delta} \subseteq T_{k',\delta}$, and (ii) $\delta' \ge \delta \Longrightarrow T_{k,\delta} \subseteq T_{k,\delta'}$. The first argument holds as each edge $e \in T_{k,\delta}$ is contained by at least $k-2$ δ-triangles, so that e is surely contained by at least $k'-2$ δ-triangles if $k' \le k$, which means $T_{k,\delta} \subseteq T_{k',\delta}$. Similarly, the second argument can also be proved. □

In this paper, we aim to address the following problem.

PROBLEM. *Given a temporal graph $\mathcal{G}^t$, an integer $k \ge 2$, and an integer $\delta \ge 0$, find the (k,δ)-truss $T_{k,\delta}$ of $\mathcal{G}^t$.*

III. INDEX-FREE APPROACH

In this section, we firstly propose a straightforward online solution derived from the classic truss decomposition [29] as a baseline. Algorithm 1 gives the pseudo code of online solution. It firstly computes the δ-support of each edge in $\mathcal{E}$, and pushes all edges into a priority queue Q in ascending order of δ-support (Lines 1-2). Then, it performs an edge peeling process that iteratively removes the edge with the minimum δ-support from queue until the δ-supports of all rest edges are no less than $k-2$ (Lines 3-11). Upon the removal of an edge e, for the other edges in each same triangle Δ with e, we will keep their δ-support unchanged if $\mathrm{mts}(\Delta)$ is greater than δ because they are never counted in the first place, or decrease their δ-support otherwise (Lines 8-10). After the peeling process, the remaining edges in the queue comprise the target (k,δ)-truss.

Correctness. The correctness of Algorithm 1 is obvious as long as the correctness of truss decomposition holds.

Complexity. The original truss decomposition algorithm takes $O(\sum_{(u,v)\in\mathcal{E}} \min\{\deg(u), \deg(v)\})$ time, where $\deg(u)$ is the degree of u in $\mathcal{G}$. Compared with that, the main extra time cost of Algorithm 1 is to compute $\mathrm{mts}(\Delta)$ for each triangle in $\mathcal{G}^t$. Let $\overline{|\tau|}$ denote the average number of timestamps associated with an edge (u,v) and $|\Delta|$ denote the total number of triangles. The time cost of computing $\mathrm{mts}(\Delta)$ for a single triangle is $O(\overline{|\tau|})$ if $\tau_{(u,v)}$ is ordered. Thus, the time complexity of Algorithm 1 is $O(\sum_{(u,v)\in\mathcal{E}} \min\{\deg(u), \deg(v)\} + \overline{|\tau|} \cdot |\Delta|)$.

IV. INDEX-BASED APPROACH

The complexity of index-free Algorithm 1 is at least subquadratic to the number of edges for a temporal graph, and thus is infeasible for real-time processing when the graph is large. Therefore, we propose index-based approaches to efficiently answer (k,δ)-truss queries in this section.

A. TC-Index

1) Index Structure: A basic idea of indexing is to preserve all possible (k,δ)-trusses for a temporal graph, which can answer any query in the optimal time due to precomputation. However, directly preserving all possible (k,δ)-trusses takes $O(k_{max} \cdot \delta_{max} \cdot |\mathcal{E}|)$ space in the worst case, where k_{max} and δ_{max} are the maximum values of k and δ respectively for a given temporal graph. It means the index could be thousands of times or even larger than the graph itself in practice. Consequently, we propose a Temporal Containment Index (TC-Index) that adopts an incremental storage scheme to reduce the index size. With only a little compromise of query efficiency compared with the uncompressed index, the size of TC-Index is reduced to $O(k_{max} \cdot (|\mathcal{E}| + \delta_{max}))$.

Before introducing TC-Index, let us consider the following pilot concept firstly.

Definition 5 (k-Span). *Given a temporal graph $\mathcal{G}^t$ and a support threshold k, the k-span of an edge $e \in \mathcal{E}$ is an integer δ = k-spn(e), such that (i) the (k,δ)-truss contains e and (ii) the (k,δ')-truss does not contain e for any $\delta' < \delta$.*

Property 5.1. *The k-span of edges in the (k,δ)-truss is no greater than δ.*

PROOF. The argument is obvious regarding Definition 5. □

With Property 4.1 and 5.1 , we only need to consider edges in the k-truss of $\mathcal{G}$ whose k-span is no greater than δ in order to find the (k,δ)-truss of $\mathcal{G}^t$, because the (k,δ)-truss of $\mathcal{G}^t$ is certainly a subgraph of the k-truss of $\mathcal{G}$.

Based on the above observation, for a temporal graph $\mathcal{G}^t$, TC-Index maintains a map structure $\mathcal{I}_k = (E_k, D_k)$ for each possible k value. E_k is a sequence that preserves edges of the k-truss in descending order of k-span, and D_k is an index that records the unique k-spans and the offsets of the first edges with the corresponding k-span in E_k. Then, TC-Index $\mathcal{I} = (\mathcal{I}_3, \mathcal{I}_4, \cdots, \mathcal{I}_{k_{max}})$ is comprised of the map structures for all possible k. Note that TC-Index does not store the map

```
Algorithm 2: TC-Query
  Input: TC-Index $\mathcal{I}$, a support threshold $k$, a minimum
         time span threshold $\delta$
  Output: the $(k,\delta)$-truss $T_{k,\delta}$ of $\mathcal{G}^t$
1 $(E_k, D_k) \leftarrow$ access $\mathcal{I}_k$;
2 $\delta' \leftarrow$ binary search of the maximum $k$-span in $D_k$ no
    greater than $\delta$;
3 $o \leftarrow$ the offset associated with $\delta'$ in $D_k$;
4 return the edges in $E_k$ from $o$ to the end of file;
```

structures for $k \leq 2$ because the $(2,\delta)$-truss is actually the entire temporal graph, regardless of δ.

EXAMPLE 4. *The TC-Index for the temporal graph in Fig 1 is given in Fig 5. Three are three map structures $\mathcal{I}_3$, $\mathcal{I}_4$, and $\mathcal{I}_5$. Within $\mathcal{I}_4$, there are four unique k-spans 6, 4, 2, and 0 in D_4, which have pointers to the first edges with these k-spans in E_4. We can see the edges (v_2, v_3), (v_2, v_7), (v_2, v_8), (v_3, v_7), and (v_3, v_8) have the k-span 6 when $k = 4$.*

THEOREM 1. *For a temporal graph $\mathcal{G}^t$, the size of TC-Index is bounded by $O(k_{max} \cdot (|\mathcal{E}| + \delta_{max}))$.*

PROOF. For each $\mathcal{I}_k = (E_k, D_k)$, the size of E_k is linear to the number of edges in the k-truss of $\mathcal{G}$, which is less than $|\mathcal{E}|$. Meanwhile, the size of D_k is at most δ_{max}. Thus, the size of $\mathcal{I}_k$ is bounded by $O(|\mathcal{E}| + \delta_{max})$. As a result, the total size of TC-Index is bounded by $O(k_{max} \cdot (|\mathcal{E}| + \delta_{max}))$. □

2) Query Processing: The TC-Index based query algorithm is named TC-Query, whose pseudo code is presented in Algorithm 2. For the given k and δ, Algorithm 2 firstly retrieves $\mathcal{I}_k = (E_k, D_k)$ from TC-Index (Line 1). Then, it finds the maximum δ' in D_k with $\delta' \leq \delta$, and locates the position in E_k with respect to the offset associated with δ' in D_k (Lines 2-3). Lastly, it scans E_k from the position until the end, and all scanned edges comprise $T_{k,\delta}$ (Line 4).

EXAMPLE 5. *Consider the (k,δ)-truss query with $k = 4$ and $\delta = 1$. In D_4, the first k-span no greater than δ is 0. Thus, we locate the edge (v_6, v_8) in E_4 and start to scan the following edges. Lastly, we get the edges of $(4,1)$-truss, namely, (v_6, v_8), (v_6, v_9), (v_6, v_{10}), (v_8, v_9), (v_8, v_{10}), and (v_9, v_{10}), which can be verified in Fig 1 (see the blue edges).*

THEOREM 2. *TC-Query computes the edge set of $T_{k,\delta}$ in at most $O(\log \delta_{max} + |T_{k,\delta}|)$ time, where $|T_{k,\delta}|$ denotes the number of edges in $T_{k,\delta}$.*

PROOF. The argument is obvious regarding Algorithm 2. □

Compared with the optimal time of processing (k,δ)-truss query that is certainly $O(|T_{k,\delta}|)$, TC-Query is almost optimal since $\log \delta_{max}$ is usually much less than $|T_{k,\delta}|$.

B. DC-Index

1) Index Structure: Although TC-Index exploits temporal containment to achieve incremental storage, it is still not space-efficient enough as it only takes into account one aspect of Property 4.1. To fully exploit the property, we propose an advanced tree-structured index called Dual Containment Index (DC-Index). DC-Index is space-optimal while guaranteeing the same order of query efficiency as TC-Index. Specifically, DC-Index is derived in the following steps.

Definition 6 ((k,δ)-Truss Graph)**.** *Given a temporal graph $\mathcal{G}^t$, we define a (k,δ)-truss graph as a directed weighted graph $G = (V, E, w)$, where $V = \{T_{k,\delta} | 3 \leq k \leq k_{max}, 0 \leq \delta \leq \delta_{max}\}$ is the set of all possible (k,δ)-trusses of $\mathcal{G}^t$, $E = E_v \cup E_h = \{(T_{k,\delta}, T_{k+1,\delta}) | 3 \leq k \leq k_{max} - 1, 0 \leq \delta \leq \delta_{max}\} \cup \{(T_{k,\delta}, T_{k,\delta-1}) | 3 \leq k \leq k_{max}, 1 \leq \delta \leq \delta_{max}\}$ is a subset of dual containment relation on V, and $w : E \mapsto \mathbb{N}$ indicates the number of incremental edges between two connected trusses.*

Fig 6(a) illustrates the (k,δ)-truss graph of our example temporal graph. Intuitively, we call the edges in E_v as vertical edges and the edges in E_h as horizontal edges. For each edge in this graph, if the sink truss has been stored, the cost of incrementally preserving the source truss is the weight of edge.

Definition 7 ((k,δ)-Truss Arborescence)**.** *Given a (k,δ)-truss graph G, we derive a (k,δ)-truss arborescence A from it by removing the outgoing edge $(T_{k,\delta}, T_{k+1,\delta})$ or $(T_{k,\delta}, T_{k,\delta-1})$ with greater weight for each truss $T_{k,\delta}$. Note that, if $T_{k,\delta}$ has one or none outgoing edge, no edge will be removed.*

Fig 6(b) illustrates the (k,δ)-truss arborescence, which is actually a minimum-weight directed spanning tree. For each truss, there is a single directed path that leads to the root, namely, $(k_{max}, 0)$-truss.

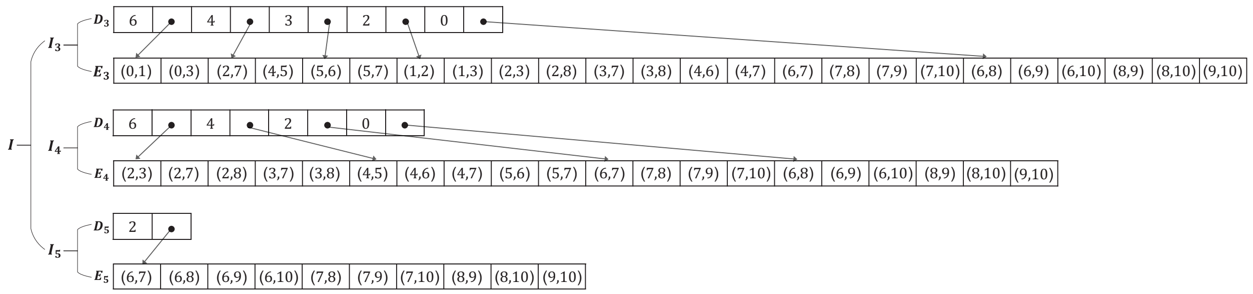
Definition 8 (Reduced (k,δ)-Truss Arborescence)**.** *Given a (k,δ)-truss arborescence A, we reduce it to another arborescence A^- by (i) removing each truss and its outgoing edge if the edge weight is zero and (ii) reconnecting each truss to the next truss on its original path in A if its sink truss is removed.*

Fig 6(c) illustrates the reduced (k,δ)-truss arborescence. Obviously, each omitted truss can still be retrieved because there is certainly another truss identical to it remained.

Then, we use an incremental storage scheme to preserve all possible (k,δ)-trusses according to the reduced (k,δ)-truss arborescence. As illustrated in Fig 6(d), the **incremental edge set tree** of DC-Index is logically equivalent to the reduced (k,δ)-truss arborescence, and preserves the Incremental Edge Sets (IESes) between trusses in each node of the tree. Compared with TC-Index, DC-Index surely has fewer redundant edges. For example, the total number of edges in Fig 6(d) is 40, and in contrast, the total number of edges in Fig 5 is 54. Actually, the space cost of DC-Index is the minimum under a precondition.

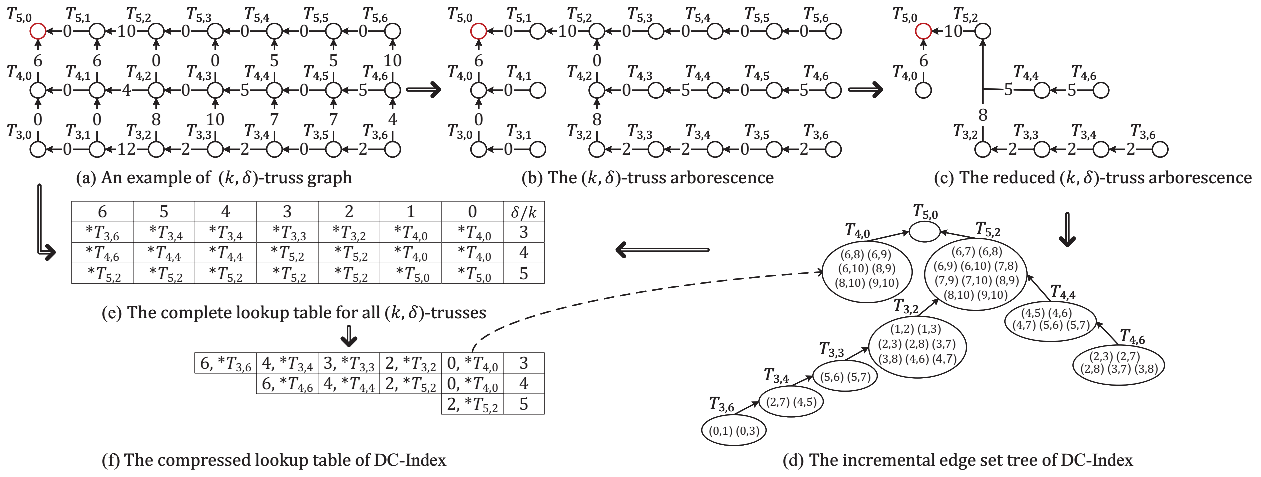
THEOREM 3. *Given a temporal graph $\mathcal{G}^t$, the incremental edge set tree of DC-Index is space-optimal for preserving all possible (k,δ)-trusses of $\mathcal{G}^t$, when the efficiency of retrieving a specific (k,δ)-truss cannot be degraded.*

PROOF. Logically, retrieving a specific (k,δ)-truss from DC-



I
I₃
D₃ 6 4 3 2 0
E₃ (0,1) (0,3) (2,7) (4,5) (5,6) (5,7) (1,2) (1,3) (2,3) (2,8) (3,7) (3,8) (4,6) (4,7) (6,7) (7,8) (7,9) (7,10) (6,8) (6,9) (6,10) (8,9) (8,10) (9,10)
I₄
D₄ 6 4 2 0
E₄ (2,3) (2,7) (2,8) (3,7) (3,8) (4,5) (4,6) (4,7) (5,6) (5,7) (6,7) (7,8) (7,9) (7,10) (6,8) (6,9) (6,10) (8,9) (8,10) (9,10)
I₅
D₅ 2
E₅ (6,7) (6,8) (6,9) (6,10) (7,8) (7,9) (7,10) (8,9) (8,10) (9,10)

Fig. 5: An example of TC-Index, in which (v_i, v_j) is represented by (i, j).



(a) An example of (k, δ)-truss graph
(b) The (k, δ)-truss arborescence
(c) The reduced (k, δ)-truss arborescence
(d) The incremental edge set tree of DC-Index
(e) The complete lookup table for all (k, δ)-trusses
(f) The compressed lookup table of DC-Index

6	5	4	3	2	1	0	δ/k
$*T_{3,6}$	$*T_{3,4}$	$*T_{3,4}$	$*T_{3,3}$	$*T_{3,2}$	$*T_{4,0}$	$*T_{4,0}$	3
$*T_{4,6}$	$*T_{4,4}$	$*T_{4,4}$	$*T_{5,2}$	$*T_{5,2}$	$*T_{4,0}$	$*T_{4,0}$	4
$*T_{5,2}$	$*T_{5,2}$	$*T_{5,2}$	$*T_{5,2}$	$*T_{5,2}$	$*T_{5,0}$	$*T_{5,0}$	5

6, $*T_{3,6}$	4, $*T_{3,4}$	3, $*T_{3,3}$	2, $*T_{3,2}$	0, $*T_{4,0}$	3
	6, $*T_{4,6}$	4, $*T_{4,4}$	2, $*T_{5,2}$	0, $*T_{4,0}$	4
				2, $*T_{5,2}$	5

Fig. 6: An example of DC-Index, in which (v_i, v_j) is represented by (i, j).

Index is simply a scan of edges in the (k,δ)-truss, the time complexity of which is $O(|T_{k,\delta}|)$. To guarantee this complexity, for each (k,δ)-truss, the index needs to store it as a single sequence of IESes, thereby avoiding to search multiple sequences with duplicated edges. Under this precondition, we prove that DC-Index is space-optimal using induction. Let S_c be a space-optimal store of all possible (k,δ)-trusses. The initial S_c only contains $\mathcal{T}_{k_{max},0}$, because $\mathcal{T}_{k_{max},0}$ is the minimal (k,δ)-truss and cannot be compressed incrementally. Then, the neighbors of $\mathcal{T}_{k_{max},0}$, namely, $\mathcal{T}_{k_{max},1}$ and $\mathcal{T}_{k_{max-1},0}$ are added into S_c, and only their incremental edges regarding $\mathcal{T}_{k_{max},0}$ are preserved. It is obvious that the updated S_c is still space-optimal. Furthermore, we keep on adding the neighbors of current trusses in S_c into S_c, namely., $\mathcal{T}_{k_{max-2},0}$, $\mathcal{T}_{k_{max},2}$, and $\mathcal{T}_{k_{max-1},1}$. It is certain that, for each newly added truss, the minimum incremental storage cost is one of the weights of its outgoing edges to the trusses of S_c. For example, the truss $\mathcal{T}_{k_{max-1},1}$ has two outgoing edges to $\mathcal{T}_{k_{max-1},0}$ and $\mathcal{T}_{k_{max},1}$ respectively, and thus the IES corresponding to the edge with less weight is stored. As a result, each truss in S_c is still stored with the minimum cost. Applying this recursively, S_c is actually the incremental edge set tree of DC-Index. □

Moreover, to retrieve a specific (k,δ)-truss in the tree, DC-Index uses a lookup table to record the pointers to tree nodes. As illustrated in Fig 6(e), the cell in k row and δ column contains the pointer to the tree node that represents an identical truss of (k,δ)-truss. For example, to lookup $T_{3,0}$, the pointer to $T_{4,0}$ in the tree is returned. We further compress the lookup table by skipping the consecutively repeating pointers for each row. As illustrated in Fig 6(f), when $k = 5$, there are only two unique pointers to $T_{5,0}$ and $T_{5,2}$ respectively, and we only record them and their smallest column ids (namely, 0 and 2) in the row. For $0 < \delta < 2$ or $2 < \delta \leq 6$, it is easy to know the corresponding $T_{5,\delta}$ is identical to $T_{5,0}$ or $T_{5,2}$.

2) Query Processing: The DC-Index based query algorithm is named DC-Query, which is similar to TC-Query. For the given k and δ, it firstly finds the maximum $\delta' \leq \delta$ in row k and gets the pointer to a node of incremental edge set tree. Then, we traverse the path from this node to the root, and the union of all traversed edge sets is the edge set of $T_{k,\delta}$. The pseudo code is omitted.

Theorem 4. *DC-Query is as efficient as TC-Query.*

Proof. Logically, they have the same computational complexity (see Theorem 2), though tree traversal is slower than sequential scan in physical implementation. □

V. Index construction

In this section, we address the scalable construction of TC/DC-Index on large-scale temporal graphs. For that, we propose two algorithms based on two classic paradigms, namely, truss decomposition [29]–[33] and truss maintenance [34]–[38], respectively.

A. TC-Index Construction based on Decomposition

Intrinsically, the construction of TC-Index can be addressed by computing the Incremental Edge Sets (IESes) between each pair of (k, δ)-truss and $(k, \delta+1)$-truss for a temporal graph. Since these IESes correspond to the horizontal edges (E_h) of the (k, δ)-truss graph illustrated in Fig 4, we call them Horizontal IES (H-IES), The index construction algorithm needs to reduce the redundant part in the computation of different H-IESes to scale to large temporal graphs. A basic strategy is to exploit the dual containment property to decrementally decompose (k, δ)-trusses in a particular order with respect to k and δ, like the existing temporal k-core decomposition [11].

However, temporal k-core decomposition does not involve the extra metric like the minimum time span of triangle, which may be needed repeatedly for evaluating δ-supports of edges. Thus, to eliminate the repeating computation of minimum time span, we design the following data structure to store $\mathrm{mts}(\Delta)$ for each Δ in a temporal graph.

Definition 9 (δ-Triangle List)**.** *Given a temporal graph $\mathcal{G}^t$, a δ-triangle list is a list of triangle sets $(S_0^\Delta, S_1^\Delta, \cdots, S_{\delta_{max}}^\Delta)$, where S_δ^Δ is the set of all triangles in $\mathcal{G}^t$ whose minimum time spans are exactly δ with $0 \le \delta \le \delta_{max}$.*

Then, for each k, we decrementally induce each (k, δ)-truss from $(k, \delta+1)$-truss, so that the H-IES between them can be obtained. Algorithm 3 presents the pseudo code of our Decomposition Based Algorithm (DBA). Initially, we enumerate all triangles and evaluate their minimum time span, for building the δ-triangle list (Line 1). Let X_k^Δ denote the set of triangles of k-truss. Obviously, the 2-truss is $\mathcal{G}^t$ itself, and thereby X_2^Δ is the union of all sets in k-triangle list (Line 2). Then, for each k with $3 \le k \le k_{max}$, we decompose $T_{k,\delta_{max}} = T_k$ gradually until $T_{k,0}$ is obtained, and collect the H-IESes (Lines 3-8). Specifically, each iteration starts with obtaining T_k of $\mathcal{G}$ by a traditional truss decomposition function `decompv()` (Line 4), since the (k, δ)-truss is always a subgraph of k-truss. In particular, we can also obtain the set of deleted triangles X^Δ by `decompv()`. Thus, the triangle set of k-truss X_k^Δ can be computed by removing X^Δ from X_{k-1}^Δ (Line 5). Then, for each δ from $\delta_{max}-1$ until 0, the (k, δ)-truss is induced by a new decomposition function `decomph()`, which invalidates the triangles in X_k^Δ from the $(k, \delta+1)$-truss (Lines 6-7). In this function, the H-IES can also be obtained (Line 8). The details of functions are as follows.

`decompv()`. This function could be any existing truss decomposition algorithm like [29]. So the details are omitted.

Algorithm 3: Decomposition Based Algorithm (DBA)

```
   Input: a temporal graph $\mathcal{G}^t$
   Output: H-IESes between $(k,\delta)$-trusses of $\mathcal{G}^t$
1  build the $\delta$-triangle list $(S_0^\Delta, S_1^\Delta, \cdots, S_{\delta_{max}}^\Delta)$;
2  $T_2 \leftarrow \mathcal{G}^t$, $X_2^\Delta \leftarrow \cup_{\delta=0}^{\delta_{max}} S_\delta^\Delta$;
3  for $k \leftarrow 3$ until $k_{max}$ do
4      $T_k, X^\Delta \leftarrow$ decompv($T_{k-1}$) ;  // $T_{k,\delta_{max}} = T_k$
5      $X_k^\Delta \leftarrow X_{k-1}^\Delta / X^\Delta$;
6      for $\delta \leftarrow \delta_{max} - 1$ until 0 do
7          $T_{k,\delta}, R \leftarrow$ decomph($T_{k,\delta+1}, X_k^\Delta$);
8          collect $R$ as the H-IES between $(k, \delta+1)$-truss
             and $(k,\delta)$-truss;

9  Function decompv($T_k$):
10     decompose $k$-truss and return both $T_{k+1}$ and the
         set of deleted triangles $X^\Delta$;

11 Function decomph($T_{k,\delta}, X_k^\Delta$):
12     $Q \leftarrow \varnothing$, $R \leftarrow \varnothing$;
13     for each triangle $\Delta \in X_k^\Delta \cap S_\delta^\Delta$ do
14         delete $\Delta$ from $X_k^\Delta$;
15         for each $e$ of $\Delta$ do
16             $\delta$-sup($e$) $\leftarrow$ $\delta$-sup($e$) $- 1$;
17             if $\delta$-sup($e$) $< k-2$ and $e \notin Q$ then
18                 $Q$.push($e$);

19     while $Q$ is not empty do
20         $e \leftarrow Q$.pop();
21         $T_{k,\delta} \leftarrow T_{k,\delta}/e$;
22         add $e$ to $R$;
23         for each $\Delta \in X_k^\Delta$ containing $e$ do
24             delete $\Delta$ from $X_k^\Delta$;
25             for each other $e'$ of $\Delta$ do
26                 $\delta$-sup($e'$) $\leftarrow$ $\delta$-sup($e'$) $- 1$;
27                 if $\delta$-sup($e'$) $< k-2$ and $e' \notin Q$ then
28                     $Q$.push($e'$);

29     return $T_{k,\delta}$, $R$;
```

`decomph()`. This function mainly consists of two phases. In the first phase (Lines 13-18), for each triangle whose minimum time span is δ in X_k^Δ, we decrease the δ-support of its edges by 1 and push the edges whose new δ-support is less than $k-2$ into a candidate container Q, which collects edges waiting to be deleted. In the second phase (Lines 19-28), for each edge e in Q, we remove it from $T_{k,\delta}$, and meanwhile add it to the H-IES R. Upon the removal of e, if there is any other triangle $\Delta \in X_k^\Delta$ that contains e, we process the other edges of Δ than e as in the first phase.

Example 6. *Consider the temporal graph in Fig 1. Since the greatest minimum time span of its triangles $\delta_{max} = 6$, the 4-truss is actually the $(4, 6)$-truss surrounded by black dashed line. For $k = 4$, we can get the 4-truss from previous 3-truss by a traditional truss decomposition, and start to decompose*

$(4,6)$-truss from $\delta = 5$. The triangle $\Delta = \{v_2, v_7, v_8\} \in X_4^{\Delta}$ with $\mathrm{mts}(\Delta) = 6$ *will be invalidated because we are trying to obtain $(4,5)$-truss currently. Thus, the δ-supports of its edges are decreased by* 1. *Then, we have both* $5\text{-sup}(v_2, v_7)$ *and* $5\text{-sup}(v_2, v_8) = 1 < 4 - 2$, *so (v_2, v_7) and (v_2, v_8) are pushed into Q and waiting to be deleted. On the removal of edge (v_2, v_7), the triangle $\{v_2, v_3, v_7\}$ containing it will be broken, so that the other two edges (v_2, v_3) and (v_3, v_7) also need to be checked like (v_2, v_7). Iteratively, the edges not belong to $(4,5)$-truss are all deleted and pushed into the H-IES between $(4,6)$-truss and $(4,5)$-truss, until Q is empty. Similarly, when δ decreases gradually, the edges remarked by red color and green color will be deleted respectively, until only the blue edges that comprise the $(4,0)$-truss remain.*

Correctness. We only discuss the correctness of `decomph()` here, since the other parts of Algorithm 3 are straightforward. Due to Property 4.1, we can induce the (k,δ)-truss from the $(k,\delta+1)$-truss. It is easy to know, we can finish that by invalidating all triangles in X_k^{Δ} whose minimum time span is greater than δ. However, for avoiding unnecessary computation, we have a trick that is to only invalidate the triangle whose minimum time span is exactly $\delta + 1$, because the other triangles have been invalidated during the previous calls of `decomph()`. The rest decomposition procedure is correct obviously.

Complexity. The time complexity of building the δ-triangle list is $O(\sum_{(u,v)\in\mathcal{E}} \min\{\deg(u), \deg(v)\} + |\overline{\tau}| \cdot |\Delta|)$, which is the same as Algorithm 1. The total time cost of calling `decompv()` is $O(\sum_{(u,v)\in\mathcal{E}} \min\{\deg(u), \deg(v)\}$. For the inner loop with a specific k, the time complexity is $O(|X_k^{\Delta}| + \sum_{(u,v)\in T_k} \min\{\deg(u), \deg(v)\})$, since each triangle in T_k is invalidated at most once and each edge in T_k is visited at most once. Thus, the total time cost of calling `decomph()` is $O(\sum_{k=3}^{k_{max}} (|X_k^{\Delta}| + \sum_{(u,v)\in T_k} \min\{\deg(u), \deg(v)\}))$, which dominates the total time cost of DBA (Algorithm 3).

B. TC/DC-Index Construction based on Maintenance

DBA (Algorithm 3) can only produce H-IES but not IES between each pair of (k,δ)-truss and $(k+1,\delta)$-truss, which are called Vertical IES (V-IES) for corresponding to the vertical edges (E_v) of the (k,δ)-truss graph in Fig 6, and thereby is not efficient for constructing DC-Index. Thus, we propose another Maintenance Based Algorithm (MBA). MBA can construct both TC-Index and DC-Index, and is more efficient than DBA.

The traditional truss maintenance problem is to update the trussness of edges when edges are inserted or deleted, which has been widely studied [34]–[37]. Different from that, we mainly focus on truss maintenance when triangles are validated or invalidated with respect to minimum time span. Moreover, similar to edge-oriented maintenance, updating trussness for triangle invalidation is much more efficient than triangle validation. Therefore, we only maintain edge trussness when a triangle becomes invalid due to the decease of δ, with respect to the following observations.

Lemma 1. *Given a graph $\mathcal{G}$, the trussness of any edge in $\mathcal{E}$ can be decreased by at most 1 if a triangle Δ of $\mathcal{G}$ gets invalid.*

PROOF. The previous work [34] has proved the conclusion for deleting an edge, which will cause the invalidation of at least one triangle. Thus, the conclusion still holds for invalidating only one triangle. □

With Lemma 1, we only need to identify the edges whose trussness will be affected by triangle invalidation and decrease their trussness by 1 for maintenance.

Definition 10 (k-Triangle)**.** *Given a graph $\mathcal{G}$, a triangle Δ of $\mathcal{G}$ is a k-triangle if it is contained by the k-truss but not the $k+1$-truss of $\mathcal{G}$. We say the level of Δ is k, denoted by $L(\Delta)$.*

Lemma 2. *For any k-triangle Δ of $\mathcal{G}$, the trussness* $\mathrm{trn}(e)$ *of an edge $e \in \mathcal{E}$ will not be updated when Δ gets invalid if* $\mathrm{trn}(e) \neq k$.

PROOF. For $\mathrm{trn}(e) \neq k$, we consider two cases (i) $\mathrm{trn}(e) > k$ and (ii) $\mathrm{trn}(e) < k$ respectively. For the case (i), we have the edge e that belongs to the $\mathrm{trn}(e)$-truss, and the k-triangle Δ does not belong to the $\mathrm{trn}(e)$-truss because $\mathrm{trn}(e) > k$, so that the invalidation of Δ will not affect the $\mathrm{trn}(e)$-truss and also the support of e in the $\mathrm{trn}(e)$-truss. Thus, $\mathrm{trn}(e)$ will not be updated. For the case (ii), we prove it by a contradiction. Assume that an edge e with $\mathrm{trn}(e) < k$ has its trussness decreased, it can be concluded that at least one support triangle of e in $\mathrm{trn}(e)$-truss has its level decreased, which implicates that at least one edge e^* with $\mathrm{trn}(e^*) = \mathrm{trn}(e)$ of this support triangle has its trussness decreased. Applying this recursively through a series of triangles sharing common edges, we must reach an edge e' of Δ with $\mathrm{trn}(e') = \mathrm{trn}(e)$, and its trussness is decreased. However, there is no edge e' of Δ with $\mathrm{trn}(e') = \mathrm{trn}(e)$ as the level of Δ is k, which causes a contradiction. □

Lemma 3. *For any k-triangle Δ of $\mathcal{G}$, the trussness* $\mathrm{trn}(e)$ *of an edge $e \in \mathcal{E}$ may be updated when Δ gets invalid, if* $\mathrm{trn}(e) = k$ *and one of following conditions is satisfied: (i) $e \in \Delta$ or (ii) $e \notin \Delta$ and $\exists e' \in \Delta$ such that* $\mathrm{trn}(e') = k$ *and e is connected with e' though a series of k-triangles sharing common edges.*

PROOF. For the case (i), we have Δ that belongs to the k-truss because it is a k-triangle, and thus the edge e with $\mathrm{trn}(e) = k$ loses a support triangle in the k-truss when Δ gets invalid. As a result, the support of e may no longer satisfy the requirement of k-truss, which means $\mathrm{trn}(e)$ may be decreased. The case (ii) can be proved like the case (ii) of Lemma 2. □

With Lemma 2 and 3, we develop an algorithm to maintain the edge trussness for a single triangle invalidation, which draws inspiration from the removal algorithm proposed by [35]. The most important trick is that, for each edge $e \in \mathcal{E}$, we maintains a stricter k-support $ks(e) = |\{\Delta | e \in \Delta, L(\Delta) = \mathrm{trn}(e)\}|$, which is the number of $\mathrm{trn}(e)$-triangles containing e. In the $\mathrm{trn}(e)$-truss, $ks(e)$ actually becomes the number of triangles that contain e, so that $ks(e)$ is no less than $\mathrm{trn}(e) - 2$.

Algorithm 4: Triangle Invalidation

```
Input: a static graph $\mathcal{G}$ of $\mathcal{G}^t$, a triangle $\Delta$
Output: the set $R$ of edges affected by invalidating $\Delta$
1  $Q \leftarrow \varnothing$; $R \leftarrow \varnothing$;
2  $k \leftarrow L(\Delta)$;
3  for each $e \in \Delta$ with $\mathrm{trn}(e) = k$ do
4      $ks(e) \leftarrow ks(e) - 1$;
5      if $ks(e) < \mathrm{trn}(e) - 2$ then
6          $Q$.push($e$);
7  while $Q$ is not empty do
8      $e \leftarrow Q$.pop();
9      $R \leftarrow R \cup e$;
10     $\mathrm{trn}(e) \leftarrow \mathrm{trn}(e) - 1$ (Lemma 1);
11     for each $\Delta$ containing $e$ in $\mathcal{G}$ do
12         if $\Delta$ is valid then
13             invalidate $\Delta$;
14             let $e'$ and $e''$ be the other two edges of $\Delta$;
15             $k^* \leftarrow \min\{\mathrm{trn}(e'), \mathrm{trn}(e'')\}$;
16             if $k^* = k$ then
17                 if $\mathrm{trn}(e') = k^*$ and $e' \in Q$ then
18                     $ks(e') \leftarrow ks(e') - 1$;
19                     if $ks(e') < \mathrm{trn}(e') - 2$ then
20                         $Q$.push($e'$);
21                 repeat Lines 17-20 for $e''$;
22     compute $ks(e)$ according to the new $\mathrm{trn}(e)$;
23 return $R$;
```

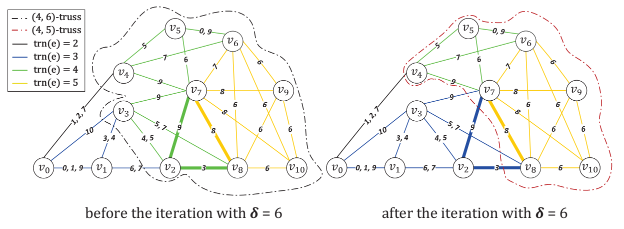

(4, 6)-truss
(4, 5)-truss
trn(e) = 2
trn(e) = 3
trn(e) = 4
trn(e) = 5
before the iteration with $\boldsymbol{\delta}$ = 6
after the iteration with $\boldsymbol{\delta}$ = 6

Fig. 7: An example of executing MBA before and after $\delta = 6$.

Algorithm 5: Maintenance Based Algorithm (MBA)

```
Input: a temporal graph $\mathcal{G}^t$
Output: both H-IES and V-IES between $(k, \delta)$-trusses
1  build the $\delta$-triangle list $(S_0^\Delta, S_1^\Delta, \cdots, S_{\delta_{max}}^\Delta)$;
2  initialize edge trussness by decompv();
3  for $\delta \leftarrow \delta_{max}$ until 1 do
4      for $k \leftarrow 3$ until $k_{max}$ do
5          add edges whose trussness is $k$ to the V-IES
             between $(k, \delta)$-truss and $(k+1, \delta)$-truss;
6      for each triangle $\Delta \in S_\delta^\Delta$ do
7          mark $\Delta$ as globally invalidated;
8          $k \leftarrow L(\Delta)$;
9          $R \leftarrow$ Algorithm 4 with $\Delta$ as input;
10         for each edge $e \in R$ do
11             add $e$ to the H-IES between $(k, \delta)$-truss and
                 $(k, \delta - 1)$-truss;
```

The pseudo code is present in Algorithm 4. After the invalidation of input k-triangle Δ, we firstly decrease the k-support of each edge e of Δ with $\mathrm{trn}(e) = k$ according to the case (i) of Lemma 3, and pushes e to a queue Q if $ks(e)$ is no longer greater than $\mathrm{trn}(e) - 1$, implying that its trussness will decrease (Lines 3-6). Then, for each edge $e \in Q$, since the decrease of $\mathrm{trn}(e)$ can further result in the decrease of trussness of other edges in a same triangle with e, we perform a breadth-first search to find such edges according to the case (ii) of Lemma 3 (Lines 7-22). Lastly, the algorithm returns the set of edges whose trussness is decreased by triangle invalidation.

Then, let us consider how to construct DC-Index through edge trussness maintenance. Since the construction of DC-Index requires to compute both V-IES and H-IES, we present the methods respectively.

V-IES. The V-IES between (k, δ)-truss and $(k+1, \delta)$-truss is simply the set of edges whose trussness is $k+1$ when the triangles whose minimum time span is greater than δ have been invalidated. Thus, we can obtain all V-IESes by invalidating triangles in descending order of minimum time span gradually, and maintain the edge trussness simultaneously.

H-IES. The H-IES between (k, δ)-truss and $(k, \delta-1)$-truss can be further obtained with respect to the following observation.

Lemma 4. *When all triangles whose minimum time span is greater than δ have been invalidated, for an edge e with $\mathrm{trn}(e) = k$, it belongs to the H-IES between (k, δ)-truss and $(k, \delta - 1)$-truss if $\mathrm{trn}(e)$ decreases after any triangle Δ with $\mathrm{mts}(\Delta) = \delta$ has been further invalidated.*

PROOF. For a specific δ, since $\mathrm{trn}(e)$ = k before the invalidation of a triangle Δ with $\mathrm{mts}(\Delta) = \delta$, the edge e is contained by $T_{k,\delta}$. Moreover, since $\mathrm{trn}(e)$ = $k' < k$ after the invalidation of Δ, the edge e is not contained by $T_{k,\delta-1}$ (and is contained by $T_{k'',\delta-1}$ with $k'' \leq k'$). Thus, e belongs to the H-IES between $T_{k,\delta}$ and $T_{k,\delta-1}$. □

With Lemma 4, we can obtain both V-IES and H-IES in the procedure of truss maintenance. Specifically, after invalidating each triangle whose minimum time span is δ, we only need to check the edges whose trussness is exactly the level of this triangle according to Lemma 2, and add edges whose trussness has decreased into the corresponding H-IES.

The pseudo code of MBA is given in Algorithm 5. We firstly build the δ-triangle list (Line 1) and compute the initial edge trussness of $\mathcal{G}$ (Line 2). Then, we enumerate δ gradually in descending order and compute the H-IES and V-IES respectively (Lines 3-11). For a specific δ, obtaining the V-IES is straightforward since the edge trussness is already known (Lines 4-5). Moreover, we invalidate each triangle Δ with $\mathrm{mts}(\Delta) = \delta$ by calling Algorithm 4. For each edge returned, we add e to the H-IES between (k,δ)-truss and $(k,\delta-1)$-truss, where k is the level of Δ.

Example 7. *Consider running MBA on the temporal graph in Fig 1. As illustrated in Fig 7, we remark the trussness of edges by different colors. The initial trussness before iterations is shown on the left. We highlight the $(4,6)$-truss comprised of green and yellow edges because $\delta_{max}=6$. Then, in the iteration with $\delta=6$, the triangle $\Delta=\{v_2,v_7,v_8\}$ is invalidated because $\mathrm{mts}(\Delta)=6$. According to Lemma 2, (v_7,v_8) will be not affected since $L(\Delta)=4\neq \mathrm{trn}(v_7,v_8)=5$. According to the case (i) of Lemma 3, (v_2,v_7) and (v_2,v_8) will be affected. According to the case (ii) of Lemma 3, (v_2,v_3), (v_3,v_7), and (v_3,v_8) will be affected. Lastly, according to Lemma 1, the trussness of affected edges is decreased by 1. The updated trussness is shown on the right, and the current green and yellow edges comprise the highlighted $(4,5)$-truss.*

Correctness. The correctness of Algorithm 5 is established on the corretness of Lemma 1, 2, 3, and 4.

Complexity. For brevity, we only compare the difference of dominant time cost between DBA and MBA here. For the quantity of visited edges (pushed into Q), two algorithms are equivalent, though DBA enumerates k firstly and MBA enumerates δ firstly. For the quantity of invalidated triangles, MBA only needs to invalidate each triangle in $\mathcal{G}^t$ once, while DBA does that for each k. Thus, the dominant time cost of MBA is $O(\sum_{k=3}^{k_{max}}\sum_{(u,v)\in T_k}\min\{\deg(u),\deg(v)\}+|\Delta|)$, which means MBA is more efficient than DBA.

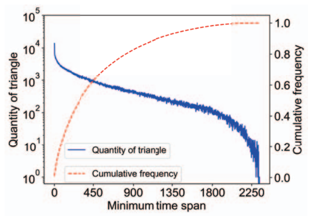
VI. Experiments

In this section, we conduct extensive experiments to evaluate the proposed approaches on a Linux machine with Intel Xeon 3.5GHz CPU and 128GB RAM. All algorithms are implemented in C++ 11 and compiled by g++.

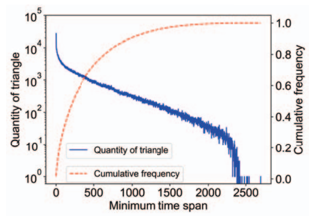
A. Dataset and Empirical Study

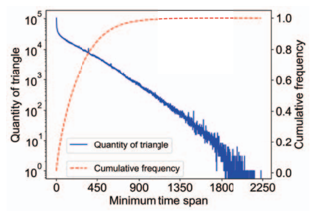
We evaluate proposed approaches on eight real-world temporal graphs, which are available publicly in [40] and [41]. The detailed statistics of these graphs are presented in Table I, where $|\mathcal{V}|$ is the number of vertices, $|\mathcal{E}|$ is the number of (static) edges, n is the number of distinct timestamps, $\overline{|\tau|}$ is the average number of timestamps associated with each edge, $|\Delta|$ is the number of triangles, k_{max} is the maximum trussness of edges, and δ_{max} is the maximum minimum time span of all triangles in the graph. In terms of $|\mathcal{E}|$, the scales of test datasets are in a wide range from 16K to 36M. While, no matter how large is the graph, k_{max} fluctuates only within a small range. This is because a k-truss is a subgraph with strict static cohesion.

There are two observations that support our motivation of studying (k,δ)-truss on the datasets. Firstly, by comparing n and δ_{max}, we can see there are indeed triangles with minimum time span as long as the duration of whole graph, which are not cohesive in terms of time. Moreover, we conduct an empirical study on these datasets. Fig 8 illustrates the distribution of triangle counts on minimum time span for four of them. We



(a) Mathoverflow.


(b) Superuser.


(c) Wikitalk.

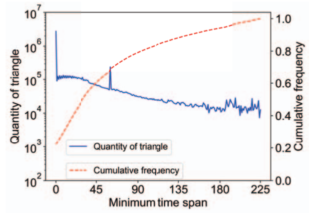

(d) Youtube.

Fig. 8: Distribution of triangle counts on minimum time span.

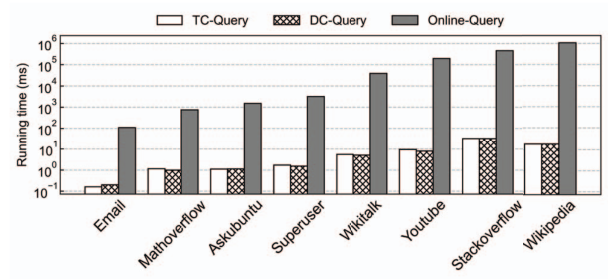


Fig. 9: Response time of query processing on different datasets, in which $k=30\%k_{max}$ and $\delta=60\%\delta_{max}$.

can see that, although the triangles with longer minimum time span are less, the distribution does not have a typical long tail. In contrast, the counts are not dropping that fast. Thus, δ is effective to constrain the structure of (k,δ)-truss.

B. Query Processing

To the best of our knowledge, no existing work investigates the (k,δ)-truss query. Thus, we use the proposed query processing algorithms, Online-Query, TC-Query, and DC-Query to evaluate efficiency. Given k and δ, we record the average running time of 100 times of repeated execution.

Since each dataset has different value ranges of k and δ, we adopt relative parameters and set $k=30\%k_{max}$ and $\delta=60\%\delta_{max}$ respectively in default for a given dataset. The running time of three algorithms under the default parameters is reported in Fig 9. TC-Query and DC-Query have similar query efficiency, and are 2∼4 orders of magnitude faster than Online-Query. Even for the two largest graphs with tens of millions of edges, index-based approaches can finish within less than 100 ms.

Then, we test the query efficiency with respect to varying parameters. Firstly, the running time of three algorithms with varying values of k on four datasets is reported in Fig 10.

TABLE I: Statistics of datasets.

Dataset	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	n	$\overline{\lvert\tau\rvert}$	$\lvert\Delta\rvert$	k_{max}	δ_{max}
`Email`	0.9K	16K	803	11.5	105K	23	800
`MathOverflow`	24K	187K	2350	1.6	1.4M	42	2336
`AskUbuntu`	159K	455K	2613	1.2	680K	26	2040
`Superuser`	194K	714K	2773	1.2	1.5M	35	2692
`WikiTalk`	1.1M	2.7M	2320	1.4	8.1M	49	2231
`YouTube`	322K	9.3M	225	1.0	12M	33	225
`Stackoverflow`	2.6M	28.1M	2774	1.2	114.2M	79	2768
`Wikipedia`	1.8M	36.5M	2235	1.1	126.6M	59	2231

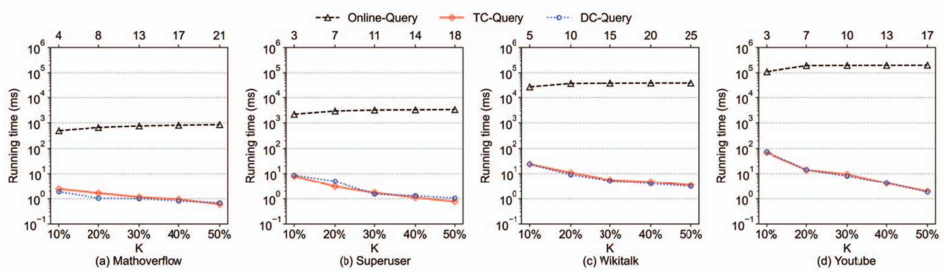


Fig. 10: Response time of query processing with varying k and $\delta = 60\%\delta_{max}$.

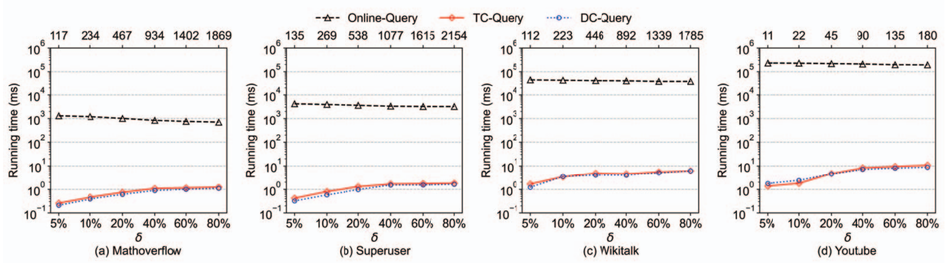


Fig. 11: Response time of query processing with varying δ and $k = 30\%k_{max}$.

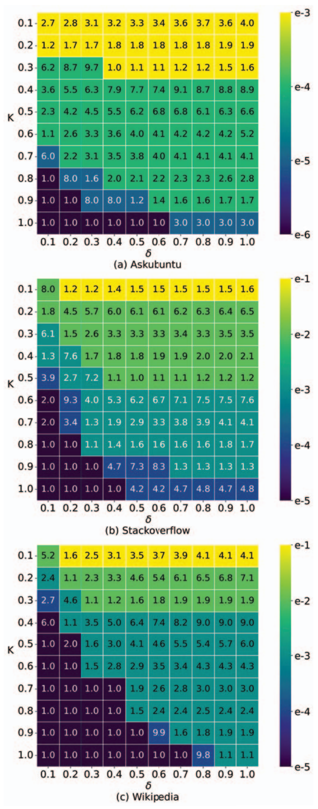


Fig. 12: Response time of DC-Query with varying k and δ.

As expected, index-based approaches spend less time on processing queries with greater k, because fewer edges need to be scanned. In contrast, Online-Query spends more time because truss decomposition needs to peel more edges. Note that, even when $k = 10\%k_{max}$ (which is usually the minimum value 3), the running time of TC-Query and DC-Query is never greater than 100 ms. Moreover, the running time of three algorithms with varying values of δ on those datasets is reported in Fig 11. Different from k, the running time of index-based approaches increases gradually with increasing δ, because the greater δ relaxes the constraint and more edges need to be scanned. Lastly, we use heat map to visualize the running time of DC-Query under more combinations of query parameters (k = 10%, 20%, $\cdots$, 100% of k_{max} and δ = 10%, 20%, $\cdots$, 100% of δ_{max}) in Fig 12. The time cost is at most 0.41 sec on the largest dataset Wikipedia, and generally decreases as k increases or δ decreases.

C. Index Construction

Firstly, we report the construction time of proposed TC-Index and DC-Index. For each dataset, we use DBA (Algorithm 3) to construct TC-Index and MBA (Algorithm 5) to construct both TC-Index and DC-Index respectively. As illustrated in Fig 13, index construction costs less than 1 sec for the smallest dataset Email with 16K edges and nearly 10,000 sec for the largest dataset Wikipedia with 36M edges. The construction time increases evenly with the increasing scale of graph, which implies that DBA and MBA reduce the redundant computation effectively. Moreover, as expected, MBA is more efficient than DBA on all datasets.

Moreover, the statistics of constructed indexes are shown in Table II. The total edge number of DC-Index is about 1.5X-10.4X large as the corresponding graph. Compared with storing all possible (k, δ)-trusses directly, DC-Index achieves the compression ratio up to 10^{-4} on most datasets. The only exceptional dataset is Youtube. According to Theorem 1, δ_{max} is the key factor affecting the compression ratio. Since YouTube has a very small δ_{max}, the compression ratio of its DC-Index is worse than other datasets. Due to the effective compression, the space cost of DC-Index is at most 903MB in our experiments. However, we observed that DC-Index is only a little smaller than TC-Index. The rationale is that, since δ_{max} is much greater than k_{max} for these datasets, the weights of horizontal edges in E_h are generally less than vertical edges

TABLE II: Statistics of indexes.

dataset	avg. entry (k-span) #	TC-Index	DC-Index				
		total edge #	total edge #	total edge # / $\|\mathcal{E}\|$	space (MB)	compression ratio (total edge # / $\sum_{k,\delta} \|T_{k,\delta}\|$)
`Email`	290	162K	154K	9.57	0.76	17.5×10^{-4}
`MathOverflow`	1365	1959K	1957K	10.40	9.35	6.754×10^{-4}
`AskUbuntu`	1086	959K	958K	2.10	7.33	11.43×10^{-4}
`Superuser`	1365	2108K	2106K	2.95	13.83	7.4×10^{-4}
`WikiTalk`	1089	10.60M	10.58M	3.79	62.01	7.67×10^{-4}
`YouTube`	170	16.74M	14.25M	1.52	125.93	1.11×10^{-2}
`Stackoverflow`	2028	139.07M	138.92M	4.93	746.15	6.00×10^{-4}
`Wikipedia`	1304	164.24M	163.40M	4.47	902.63	8.60×10^{-4}

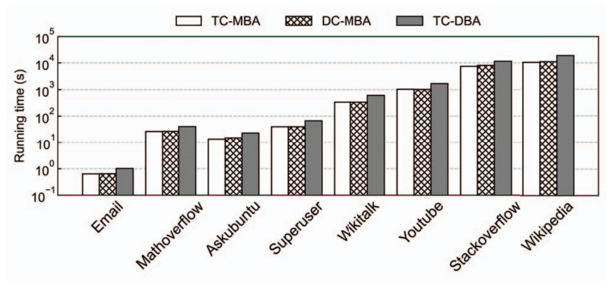


Fig. 13: Construction time of TC-Index and DC-Index using DBA and MBA for different datasets.

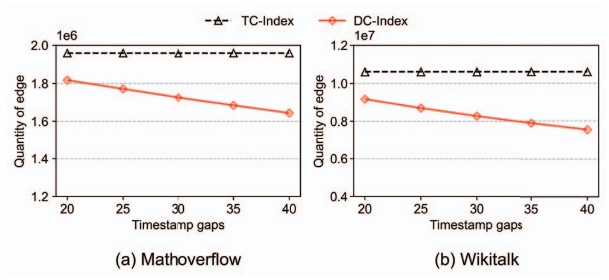


Fig. 14: Comparison of total edge number between DC-Index and TC-Index with different time granularity.

in E_v, so that the (k, δ)-truss arborescence has only a few vertical edges. As a result, the space of DC-Index is almost as large as TC-Index.

For the above anomaly observed, we conduct an extra experiment to verify the effectiveness of DC-Index. Specifically, we merge every 20, 25, 30, 35, or 40 consecutive timestamps into a single new timestamp for coarsening the time granularity (like from day to month), which will decrease δ_{max} but not change k_{max}. Thus, the weights of horizontal edges in new E_h will become greater. Fig 14 illustrates the comparison of total edge number between DC-Index and TC-Index with the new settings, on two selected datasets. Clearly, DC-Index regains the advantage in the more balanced temporal graphs.

VII. Related Work

Recently, there emerge studies on k-truss for temporal graphs. The (k, Δ, θ)-truss [21] inherits the definition of persistent core [15], and requires its edges to have supports no less than $k-2$ in a number of time windows no shorter than Δ, the total duration of which is no less than θ. Such a truss model is not general enough for many application scenarios. The (k, Δ)-truss (also known as span truss) [20] defines the temporal support of edges, which is the support in a projection of temporal graph during a period, and aims to find the k-truss that is cohesive enough in the specific time window Δ. This truss model is more general. However, it always needs the user to give a specific time window in which triangles occur. In contrast, our (k, δ)-truss allows triangles to occur in different time windows that are still short enough. Thus, our (k, δ)-truss is more relaxed than span truss, and meanwhile, can be equivalent to span truss when an extra time window is specified and $\delta = \infty$ if necessary.

Since our (k, δ)-truss is designed on top of temporal triangles, we also investigate the related researches [22]–[27]. Different from their typical definitions of triangle/motif duration, we propose another kind of duration, namely, minimum time span that is more meaningful in the context of k-truss.

Moreover, the design of our indexes is inspired by (k, l)-core [42] and (k, p)-core [43], which also consider the similar property with respect to dual parameters. The various elegant algorithms [29], [34]–[37] to solve k-truss problems provide useful guides to develop our index construction algorithms.

VIII. Conclusion

In this paper, we study a novel (k, δ)-truss on temporal graphs, which constrains the minimum time span of triangles to guarantee temporal cohesion. Such a constraint tailors the static truss in time dimension effectively. To address the query problem of (k, δ)-truss, we propose both index-free and index-based approaches. The indexes that exploits the dual containment property of (k, δ)-truss to compress the space can deliver efficient query processing. Moreover, we develop scalable index construction algorithms. The theoretical proof and experimental evaluation of our approach are provided.

Acknowledgment

This work was supported by the grants of the National Natural Science Foundation of China (No. 61202036, 61672389, 62272353, and 62276193), the Guangzhou Key Laboratory of Big Data and Intelligent Education (No. 201905010009), and the Research Grants Council of Hong Kong, China (No. 14205520).

References

[1] N. Masuda, R. Lambiotte, "A Guide to Temporal Networks," World Scientific Publishing Europe Ltd, 2016.

[2] P. Holme, J. SaramakiJari, "Temporal network," Springer, 2012.

[3] G. Kossinets, D. Watts, "Empirical analysis of an evolving social network," science, 311(5757): pp. 88-90, 2006.

[4] X. Huang, Y. Yang, Y. Wang, et al, "Dgraph: A large-scale financial dataset for graph anomaly detection," Advances in Neural Information Processing Systems, 35: pp. 22765-22777, 2022.

[5] M. Filipovska, H. Mahmassani, "Spatio-Temporal Characterization of Stochastic Dynamic Transportation Networks," IEEE Transactions on Intelligent Transportation Systems, 24(9): pp. 9929-9939, 2023.

[6] C. Hidalgo, C. Rodríguez-Sickert, "The dynamics of a mobile phone network," Physica A: Statistical Mechanics and its Applications, 387(12): pp. 3017-3024, 2008.

[7] J. Simeunović, B. Schubnel, et al, "Spatio-temporal graph neural networks for multi-site PV power forecasting," IEEE Transactions on Sustainable Energy, 13(2): pp. 1210-1220, 2021.

[8] N. Masuda, P. Holme, "Introduction to temporal network epidemiology," Springer Singapore, 2017.

[9] E. Galimberti, A. Barrat, F. Bonchi, et al, "Mining (maximal) span-cores from temporal networks," Proceedings of the 27th ACM international Conference on Information and Knowledge Management, pp. 107-116, 2018.

[10] H. Wu, J. Cheng, et al, "Core decomposition in large temporal graphs," 2015 IEEE International Conference on Big Data (Big Data), pp. 649–658, 2015.

[11] J. Yang, M. Zhong, Y. Zhu, T. Qian, M. Liu, and J. Yu, "Scalable time-range k-core query on temporal graphs," PVLDB, 16(5): pp. 1168–1180, 2023.

[12] M. Yu, D. Wen, L. Qin, et al, "On querying historical k-cores," PVLDB, 14(11): pp. 2033–2045, 2021.

[13] M. Zhong, J. Yang, Y. Zhu, T. Qian, M. Liu, J. Yu, "A Unified and Scalable Algorithm Framework of User-Defined Temporal $(k, \mathcal{X})$-Core Query," IEEE Transactions on Knowledge and Data Engineering, 2024.

[14] W. Bai, Y. Chen, and D. Wu, "Efficient temporal core maintenance of massive graphs," Information Sciences, 513: pp. 324–340, 2020.

[15] R. Li, J. Su, L. Qin, J. Yu, and Q. Dai, "Persistent community search in temporal networks," ICDE, pp. 797–808, 2018.

[16] L. Chu, Y. Zhang, Y. Yang, L. Wang, and J. Pei, "Online density bursting subgraph detection from temporal graphs," PVLDB, 12(13): pp. 2353–2365, 2019.

[17] H. Qin, R. Li, Y. Yuan, G. Wang, W. Yang, and L. Qin, "Periodic communities mining in temporal networks: Concepts and algorithms," IEEE Transactions on Knowledge and Data Engineering, 34(8): pp. 3927-3945, 2020.

[18] Y. Li, J. Liu, H. Zhao, J. Sun, Y. Zhao, and G. Wang, "Efficient continual cohesive subgraph search in large temporal graphs," World Wide Web, 24(5): pp. 1483–1509, 2021.

[19] Y. Tang, J. Li, N. Haldar, Z. Guan, J. Xu, and C. Liu, "Reliable community search in dynamic networks," PVLDB, 15(11): pp. 2826–2838, 2022.

[20] Q. Lotito, A. Montresor, "Efficient Algorithms to Mine Maximal Span-Trusses From Temporal Graphs," unpublished.

[21] L. Xu, R. Li, G. Wang, et al, "Research on K-truss Community Search Algorithm for Temporal Networks," Journal of Frontiers of Computer Science and Technology, 14(9): pp. 1482-1489, 2020.

[22] N. Pashanasangi, and C. Seshadhri, "Faster and Generalized Temporal Triangle Counting, via Degeneracy Ordering," KDD, pp. 1319–1328, 2021.

[23] J. Wang, Y. Wang, W. Jiang, Y. Li, and K. Tan, "Efficient Sampling Algorithms for Approximate Temporal Motif Counting," CIKM, pp. 1505–1514, 2020.

[24] A. Paranjape, A. Benson, and J. Leskovec, "Motifs in Temporal Networks," WSDM, pp. 601–610, 2017.

[25] P. Liu, A. Benson, and M. Charikar, "Sampling Methods for Counting Temporal Motifs," WSDM, pp. 294–302, 2019.

[26] P. Liu, V. Guarrasi, and A. Sarıyuce, "Temporal Network Motifs: Models, Limitations, Evaluation," IEEE Transactions on Knowledge and Data Engineering, 35: pp. 945–957, 2023.

[27] L. Kovanen, M. Karsai, K. Kaski, J. Kertész, and J. Saramäki, "Temporal motifs in time-dependent networks," Journal of Statistical Mechanics: Theory and Experiment, P11005, 2011.

[28] J. Leskovec, J. Kleinberg, C. Faloutsos, "Graphs evolution: Densification and shrinking diameters" ACM transactions on Knowledge Discovery from Data (TKDD), 1(1): pp. 2–42, 2007.

[29] J. Wang, J. Cheng, "Truss decomposition in massive networks," PVLDB, 5(9): pp. 812–823, 2012.

[30] P. Chen, C. Chou, and M. Chen, "Distributed algorithms for k-truss decomposition," IEEE International Conference on Big Data (Big Data), pp. 471–480, 2014.

[31] Y. Che, Z. Lai, S. Sun, Y. Wang, and Q. Luo, "Accelerating truss decomposition on heterogeneous processors," Proceedings of the VLDB Endowment, 13(10): pp. 1751-1764, 2020.

[32] H. Kabir, K. Madduri, "Parallel k-truss decomposition on multicore systems," IEEE High Performance Extreme Computing Conference (HPEC), pp. 1-7, 2017.

[33] H. Kabir, K. Madduri, "Shared-memory graph truss decomposition," IEEE 24th International Conference on High Performance Computing (HiPC), pp. 13–22, 2017.

[34] X. Huang, H. Cheng, L. Qin, et al, "Querying k-truss community in large and dynamic graphs," Proceedings of the 2014 ACM SIGMOD international conference on Management of data, pp. 1311-1322, 2014.

[35] Y. Zhang, J. Yu, "Unboundedness and efficiency of truss maintenance in evolving graphs," Proceedings of the 2019 International Conference on Management of Data, pp. 1024-1041, 2019.

[36] R. Zhou, C. Liu, J. Yu, et al, "Efficient truss maintenance in evolving networks," arXiv preprint arXiv:1402.2807, 2014.

[37] Q. Luo, D. Yu, X. Cheng, et al, "Batch processing for truss maintenance in large dynamic graphs," IEEE Transactions on Computational Social Systems, pp. 1435-1446, 2020.

[38] Z. Sun, X. Huang, Q. Liu, et al, "Efficient Star-based Truss Maintenance on Dynamic Graphs," Proceedings of the ACM on Management of Data, 1(2): pp. 1-26, 2023.

[39] V. Batagelj and M. Zaversnik, "An o (m) algorithm for cores decomposition of networks," arXiv preprint cs/0310049, 2003.

[40] J. Leskovec, A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[41] J. Kunegis, "Konect: the koblenz network collection," in Proceedings of the 22nd international conference on world wide web, pp. 1343–1350, 2013.

[42] Y. Chen, J. Zhang, Y. Fang, et al, "Efficient community search over large directed graphs: An augmented index-based approach," Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence, pp. 3544-3550, 2021.

[43] C. Zhang, F. Zhang, W. Zhang, et al, "Exploring finer granularity within the cores: Efficient (k, p)-core computation," ICDE, pp. 181-192, 2020.