Efficiently Counting Triangles in Large Temporal Graphs

Anonymous Author(s)

ABSTRACT

In many real-world applications (e.g., email networks, social networks, and phone call networks), the relationships between entities can be modeled as a temporal graph, in which each edge is associated with a timestamp representing the interaction time. As a fundamental task in temporal graph analysis, triangle counting has received much attention, and several triangle models have been developed, including δ-temporal triangle, sliding-window triangle, and $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-temporal triangle. In particular, the δ-temporal triangle, requiring the gap of timestamps of any two edges within it to be bounded by a threshold δ, has been demonstrated effective in many real applications, such as cohesiveness analysis, transitivity, clustering coefficient, and graph classification. In this paper, we study fast algorithms for counting δ-temporal triangles in a given query time window. We first propose an online algorithm, which enumerates all edges in the graph and for each edge, calculates how many δ-temporal triangles end with the edge. We further develop an efficient index-based solution, which maps δ-temporal triangles into points of the 2-dimensional space and further compactly organizes these points using hierarchical structures. Besides, we study the problem of binary δ-temporal triangle counting by considering the existence of δ-temporal triangle among three vertices. Experiments on large temporal graphs show that our online algorithm is up to 70× faster than the state-of-the-art algorithm, and our index-based algorithm is up to 10^8× faster than online algorithm.

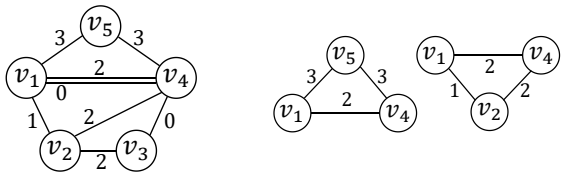
ACM Reference Format:

Anonymous Author(s). 2024. Efficiently Counting Triangles in Large Temporal Graphs. In *Proceedings of International Conference on Management of Data (SIGMOD '25).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In many real-world applications (e.g., email networks, social networks, and phone call networks), the relationships between entities can be modeled as a temporal graph [19, 26, 35, 54], in which each edge is associated with a timestamp representing the interaction time. Figure 1(a) depicts an example temporal graph, where the numbers on the edges represent their occurring timestamps. For example, the two temporal edges between vertices v_1 and v_4 indicate that they have two interactions at timestamps 0 and 2.

As a fundamental task in temporal graph analysis, triangle counting has received a great deal of attention [37, 48, 50]. Many analytical parameters contain the result of triangle counting, for instance, transitivity [13] and clustering coefficient [49]. Besides, triangle counting plays an important role in many applications, such as spam detection [3], social network analysis [44], community detection [23, 33], and so on [28, 32]. Different from triangles in static graphs, the triangles in temporal graphs often take the timestamps into consideration and several temporal triangle models have been developed, including δ-temporal triangle [42], sliding-window triangle [24], and $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-temporal triangle [43]. Among these models, the δ-temporal triangle, proposed by Paranjape, Benson, and Leskovec [42], has been demonstrated effective in many real applications. Conceptually, a δ-temporal triangle is a triangle formed by three temporal edges, such that the gap between the timestamps of any two temporal edges is bounded by a threshold δ. Consider the temporal graph in Figure 1(a) and let δ=1. Then there are two 1-temporal triangles as shown in Figure 1(b).



(a) An example temporal graph G (b) Two 1-temporal triangles

Figure 1: An example of counting δ-temporal triangles.

In this paper, we study the problem of efficiently counting δ-temporal triangles in large temporal graphs. Given a temporal graph G, a duration δ, and a time window $[t_s, t_e]$, the goal is to count all the δ-temporal triangles within $[t_s, t_e]$. For example, in Figure 1(a), assuming that δ=1 and $[t_s, t_e]$=[0,2], then there are two 1-temporal triangles within the time window $[0, 2]$ as depicted in Figure 1(b). As shown in the literature [7, 42, 43], counting δ-temporal triangles in temporal graphs has been demonstrated effective in many real applications, to name a few:

- **Graph cohesiveness analysis.** A triangle represents the strong and stable relationship among three vertices [29]. As a result, δ-temporal triangle counting can be used to measure the cohesiveness of temporal graphs. For example, the triangle density of a graph is defined as the total number of triangles over the number of vertices [45, 51], and it can be used to identify graph communities [21].
- **Graph transitivity.** Given a graph, its transitivity [13] evaluates how strong the vertices are aggregated, and is defined as three times the number of triangles over the number of wedges, where a wedge is a path of two connected edges. The transitivity can be easily extended for temporal graphs by using δ-temporal triangles.
- **Higher-order network analysis.** As a typical higher-order structure, triangle is a building block of many networks [4], and δ-temporal triangle counting is useful in analysis of users' behaviors (e.g., blocking communication and bitcoin transfer) in temporal networks [42].

SIGMOD '25, June 9 - June 15, 2025, Berlin, Germany

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Despite its wide usefulness, counting δ-temporal triangles in large temporal graphs is a very challenging task, due to the following two reasons: 1) Different from counting triangles in static graphs, counting δ-temporal triangles is more complicated since it needs to consider the timestamp of each edge; and 2) the number of δ-temporal triangles is huge, especially when δ is large and the query time window covers a long duration.

Prior works. To tackle the above challenges, Paranjape et al. [42] developed a δ-temporal triangle counting algorithm with $O(m\sqrt{\Gamma})$ time, where m is the number of edges in the graph G and $\Gamma \leq O(m^{1.5})$ is the number of static triangles in G, so it achieves a time complexity of $O(m^{1.75})$. Noujan and Seshadhri proposed the state-of-the-art (SOTA) algorithm for δ-temporal triangle counting, DOTTT, with $O(m\kappa \log(m))$ time cost [43], where κ is the degeneracy of the graph and is up to $O(m^{0.5})$. Nevertheless, these algorithms are still inefficient for processing large temporal graphs.

Online solutions. To efficiently count δ-temporal triangles, we first propose an efficient online algorithm OTTC. Specifically, given a duration δ and a query time window $[t_s, t_e]$, we first derive a projected graph $G_{[t_s,t_e]}$ by extracting edges appearing in $[t_s, t_e]$. Then, we enumerate the temporal edges chronologically to count δ-temporal triangles. OTTC completes in $O(m\kappa)$ time and it is faster than the SOTA algorithm [43] with $O(m\kappa \log(m))$ time cost, as shown in Table 1.

Index-based solutions. Although the above online algorithm outperforms existing algorithms, it may not be efficient in some scenarios because in practice, to accomplish a specific task, users often have to frequently issue δ-temporal triangle counting queries by varying the time window $[t_s, t_e]$ and duration δ multiple times. In light of this, we propose an elegant index-base solution, which converts the triangle counting problem into a point counting problem. Specifically, for each δ-temporal triangle, we first convert it into a 2-dimensional point (t_s, t_e), where (t_s, t_e) is the minimal time interval that $G_{[t_s,t_e]}$ contains this δ-temporal triangle. Afterward, we employ the wavelet tree index to solve this 2-dimensional point counting problem. However, updating the wavelet trees is costly, so we also design a fast index maintenance algorithm.

Besides, in many real-world scenarios, we may only need to consider the existence of δ-temporal triangles, so we introduce the binary δ-temporal triangle counting problem. We also develop efficient online and index-based solution to solve this variant.

In addition, we have performed extensive experimental evaluations on four real and one synthetic temporal graphs, and the results show that our algorithms are highly efficient and scalable. In particular, our online algorithm is up to 70× faster than the SOTA online algorithms, and our index-based algorithm runs up to 10^8× faster than the online algorithms. Besides, our algorithms for binary δ-temporal triangle counting are also highly efficient.

In summary, our principal contributions are as follows.

- We have developd an efficient online algorithm for counting δ-temporal triangles, whose time complexity is lower than that of the SOTA algorithm.
- We have designed a novel index-based solution by converting the δ-temporal triangles into 2-dimensional points.
- We have introduced the binary δ-temporal triangle counting problem and also developed efficient algorithms.
- We have performed comprehensive experiments on both real and synthetic large temporal graphs, and the results show that our algorithms are highly effective and efficient.

Table 1: Overview of the complexities of solutions.

Algorithm	Indexing complexity		Counting complexity	
	time	space	time	space
DOTTT [43]	–	–	$O(m\kappa \log(m))$	$O(m)$
OTTC	–	–	$O(m\kappa)$	$O(m)$
WT-Index	$O(m\kappa + \Delta \log(\Delta))$	$O(m + \Delta \log(\bar{c}))$	$O(\log^2(\Delta))$	$O(\Delta\ log(\bar{c}))$

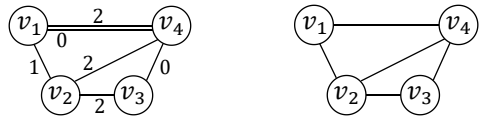
Note: given a temporal graph G, n and m are the numbers of vertices and edges in G respectively, and t_{max} is the maximum timestamp of G. Δ is the number of C-points in G, and $\bar{c}$ is the average number of δ-temporal triangles sharing the same vertices.

Outline. We present the δ-temporal triangle counting problem in Section 2. We introduce our online and index-based solutions in Sections 3 and 4, respectively. In Section 5, we focus on binary δ-temporal triangles counting. Section 6 shows experimental results. We review related works in Section 7 and conclude in Section 8.

2 PRELIMINARIES

In this section, we formally introduce the concepts of temporal graph and δ-temporal triangles, and our studied counting problems. The frequently used notations are provided in Table 2.

Definition 1 (Temporal graph). *A temporal graph is a graph $G = (V, E)$ with a set V of vertices and a set E of edges, such that each edge $e \in E$ is a triplet (u, v, t), where t is a timestamp indicating the interaction time between two vertices u and v.*



(a) The projected graph $G_{[0,2]}$ (b) The static graph $G^*_{[0,2]}$

Figure 2: Illustrating the projected graph and static graph.

Given a temporal graph $G = (V, E)$, we use $n = |V|$ and $m = |E|$ to denote the numbers of vertices and edges in G respectively. For a vertex $u \in G$, the set of its neighbors in the G is denoted as $N(u)$. We use $E(u, v)$ to denote the list of edges sharing end vertices u and v, sorted in ascending order of their timestamps. Note that for simplicity, we assume there is no more than one edge with the same timestamp between any two vertices in the graph. For instance, Figure 2 gives a temporal graph comprising four vertices and eight edges, where each edge is labeled with an integer representing the timestamp. W.l.o.g., assume that all the edge timestamps fall within the range of consecutive integers in $[0, t_{max}]$, where $t_{max} \leq m$.

Definition 2 (δ-temporal triangle [42]). *Given a temporal graph G and a duration δ ($\delta \geq 0$), a δ-temporal triangle is a subgraph of three temporal edges, i.e., (v_1, v_2, t_1), (v_2, v_3, t_2), and (v_3, v_1, t_3), such that the gap between the timestamps of any two temporal edges is at most δ, i.e., $\forall i, j \in [1, 3]$, $|t_i - t_j| \leq \delta$.*

Table 2: Notations and meanings.

Notation(s)	Meaning
$G=(V,E)$	A temporal graph with vertex set V and edge set E
(u,v,t)	An edge between u and v with timestamp t
n, m	The number of vertices and edges of G respectively
t_{max}	The maximum timestamp in G
κ	The degeneracy of G
$N(u)$	The set of neighbors of $u \in V$
$[t_s, t_e]$	A time window with $t_s \le t_e$
$E(u,v)$	A list of edges with ending vertices u and v
$\langle (x,y), c \rangle$	A C-point at (x,y) with count c
Δ	The total number of C-points of G
$\Delta_{u,v,w}$	A static triangle with its vertices set $\{u,v,w\}$

Definition 3 (Projected graph). *Given a temporal graph G and a time window $[t_s, t_e]$, the projected graph of G over $[t_s, t_e]$ is a temporal graph, denoted by $G_{[t_s,t_e]}$, that encompasses all the edges $(u,v,t) \in G$ with timestamps $t \in [t_s, t_e]$.*

Problem 1 (δ-temporal triangle counting [42]). *Given a temporal graph G, a time window $[t_s, t_e]$, and a duration δ ($\delta \ge 0$), return the number of δ-temporal triangles in $G_{[t_s,t_e]}$.*

For example, in the temporal graph of Figure 1(a), let $[t_s, t_e]$=[0,2] and δ=1. We can first obtain the projected graph $G_{[0,2]}$ as depicted in Figure 2(a), and then find that there is one 1-temporal triangle in $G_{[0,2]}$ as shown in the right part of Figure 1(b).

3 ONLINE ALGORITHMS

We begin with the concept of static graph: the static graph of a temporal graph G=(V,E) is a graph, denoted by G^*=(V,E^*), that only stores the static edges of G, where $E^* = \{(u,v)|(u,v,t) \in E\}$. To count triangles in static graphs, researchers often use the concepts of degeneracy and degeneracy order [12, 43]:

Definition 4 (Degeneracy and degeneracy order [43]). *Given a static graph G^*=(V,E^*), its degeneracy is the smallest integer κ, such that there exists an order P of all the vertices, $v_1 \prec v_2 \prec \cdots \prec v_n$, a.k.a. degeneracy order, satisfying $d^+(v) \le \kappa$ for each $v \in V$, where $d^+(v)$ is the size of set $N^+(v)$ of v's neighbors with larger orders in P, i.e., $N^+(v) = \{u | v \prec u\}$.*

Multiple degeneracy orders may exist in a static graph, and one degeneracy order can be efficiently computed by using an algorithm whose time complexity is linear to the number of edges in the graph [41]. For instance, Figure 2(b) depicts the static graph $G^*_{[0,2]}$ of the projected graph $G_{[0,2]}$. In $G^*_{[0,2]}$, a possible degeneracy order P is $v_4 \prec v_3 \prec v_1 \prec v_2$, and its degeneracy is $\kappa = 2$.

3.1 The SOTA algorithm DOTTT

To solve Problem 1, a natural idea is to first extract the projected graph $G_{[t_s,t_e]}$, and then employ the algorithm of counting δ-temporal triangles in $G_{[t_s,t_e]}$. To the best of our knowledge, the SOTA algorithm is DOTTT [43], which has three steps:

(1) Derive the degeneracy order P of all vertices in $G^*_{[t_s,t_e]}$;
(2) Enumerate all the (static) triangles $\Delta_{u,v,w}$ in $G^*_{[t_s,t_e]}$;
(3) For each triangle in $G^*_{[t_s,t_e]}$, derive the number R of δ-temporal triangles in $G_{[t_s,t_e]}$ which share its three vertices.

In step (2), we can use a classic triangle counting algorithm [12] which costs $O(m\kappa)$ time by using the degeneracy order. Specifically, for each edge (u,v) in $G^*_{[t_s,t_e]}$, we first obtain $N^+(u)$ and $N^+(v)$, and then get their intersection to enumerate the triangles.

In step (3), for each static triangle $\Delta_{u,v,w}$, DOTTT counts all the δ-temporal triangles containing vertices $\{u,v,w\}$, within $O((|E(u,v)|+|E(u,w)|)\log(|E(v,w)|))$ time. Since the details of step (3) are very complicated, we omit the introduction here.

As a result, by considering all the static triangles, DOTTT achieves a time complexity of $O(\sum_{\{u,v,w\}}(|E(u,v)|+|E(u,w)|)\log(|E(v,w)|))$. It is proved in [43] that $O(\sum_{\{u,v,w\}}(|E(u,v)|+|E(u,w)|)) = O(m\kappa)$, where u has the smallest degeneracy order among $\{u,v,w\}$. Hence, DOTTT costs $O(m\kappa\log(m))$ time in total.

3.2 Our online algorithm OTTC

In this section, we present a faster online algorithm OTTC with $O(m\kappa)$ time, which counts the number of δ-temporal triangles by enumerating the temporal edges chronologically. Similar to DOTTT, it first derives the degeneracy order P for $G^*_{[t_s,t_e]}$. Then, it sequentially enumerates the timestamps $t_i \in [t_s, t_e]$, during which it dynamically maintains the numbers of temporal edges and temporal wedges respectively containing each vertex pair in a sliding window of duration δ, and derives the number of δ-temporal triangles by using the numbers maintained above. Here, a *temporal wedge* is formed by two temporal edges (v_1,v_2,t) and (v_1,v_3,t') satisfying $v_1 \prec v_2$, $v_1 \prec v_3$, and $|t-t'| \le \delta$. In Figure 2(a), for instance, assuming δ=1, then $(v_2,v_3,1)$ and $(v_3,v_4,1)$ form a temporal wedge where the center is v_3.

Next, we discuss how to dynamically maintain $I(u,v)$ and $\Lambda(u,v)$. Specifically, assume $I(u,v)$ has stored the number of temporal edges between u and v, and $\Lambda(u,v)$ has stored the number of temporal wedges that take u and v as end vertices, in a time window $[t_i-\delta-1, t_i-1]$. Then, when sliding to the next timestamp t_i, it updates $I(u,v)$ and $\Lambda(u,v)$ by throwing historical edges and taking new edges (always assume $u \prec v$):

- **Throw historical edges:** For each historical edge $(u,v,t_i-\delta-1)$, decrease $I(u,v)$ by 1 since this edge is not in the new time window $[t_i-\delta, t_i]$. Meanwhile, decrease $\Lambda(v,w)$ by $I(u,w)$ for each vertex $w \in N^+(u)$, since removing this edge will eliminate temporal wedges that use u as the centers.
- **Take new edges:** For each new edge (u,v,t_i), increase $I(u,v)$ by 1 since this edge is in the new time window $[t_i-\delta, t_i]$. Meanwhile, increase $\Lambda(v,w)$ by $I(u,w)$ for each vertex $w \in N^+(u)$, since adding this edge will generate temporal wedges that use u as the centers.

After updating $I(u,v)$ and $\Lambda(u,v)$, we can count the number of temporal triangles with vertices $\{u,v,w\}$ that include the temporal edge $e=(u,v,t_i)$, by considering two different cases:

(1) w **has the smallest order** ($w \prec u \wedge w \prec v$): The number of δ-temporal triangles containing e is $\Lambda(u,v)$, since each of them includes a temporal wedge between u and v.
(2) u **has the smallest order** ($u \prec w \wedge u \prec v$) : The number of δ-temporal triangles containing e is $\sum_{w \in N^+(u)} I(u,w) \times I(v,w)$, since any temporal edge between u and w forms a triangle with any temporal edge between v and w, and e.

Note that since we assume $u \prec v$ and v cannot have the smallest order, there are only two cases above. Algorithm 1 shows OTTC. We first obtain $G^*_{[t_s,t_e]}$ and P, and initialize $I(u,v)$ and $\Lambda(u,v)$ (lines 1-2). Then, we sequentially enumerate the timestamps (lines 4-15), during which we dynamically maintain $I(u,v)$ and $\Lambda(u,v)$ in a sliding window of duration δ. For each timestamp t_i, we first throw historical edges (lines 5-8), and then consider each new edge and also count the number of temporal triangles containing it (lines 9-15). Finally, we obtain the result R (line 16).

Algorithm 1: OTTC

Input: A temporal graph G, a time window $[t_s, t_e]$, a threshold δ.
Output: The number of δ-temporal triangles in $G_{[t_s,t_e]}$.

1 Extract the static graph $G^*_{[t_s,t_e]}$ and derive its degeneracy order P;
2 **foreach** $(u,v) \in G^*_{[t_s,t_e]}$ **do** $I(u,v) \leftarrow 0, \Lambda(u,v) \leftarrow 0$;
3 $R \leftarrow 0$;
4 **foreach** $t_i \in [t_s, t_e]$ **do**
5 **foreach** $(u,v,t) \in G_{[t_s,t_e]}$ *with* $t = t_i - \delta - 1$ **do**
6 W.l.o.g., assume $u \prec v$ under P;
7 **foreach** $w \in N^+(u)$ **do** $\Lambda(v,w) \leftarrow \Lambda(v,w) - I(u,w)$;
8 $I(u,v) \leftarrow I(u,v) - 1$;
9 **foreach** $(u,v,t_i) \in G_{[t_s,t_e]}$ **do**
10 W.l.o.g., assume $u \prec v$ under P;
11 **foreach** $w \in N^+(u)$ **do**
12 $\Lambda(v,w) \leftarrow \Lambda(v,w) + I(u,w)$;
13 $R \leftarrow R + I(u,w) \times I(v,w)$;
14 $I(u,v) \leftarrow I(u,v) + 1$;
15 $R \leftarrow R + \Lambda(u,v)$;
16 **return** R;

Example 1 further illustrates our algorithm OTTC.

Example 1. *Consider the temporal graph in Figure 1(a) and let $[t_s, t_e]$ = [0,2] and δ = 1. We first obtain the static graph $G^*_{[0,2]}$ shown in Figure 2(b) and derive its degeneracy order $P : v_1 \prec v_3 \prec v_4 \prec v_2$. Then, we sequentially enumerate the timestamps in [0,2], during which we dynamically maintain $I(u,v)$ and $\Lambda(u,v)$, with values in Table 3, where each cell is presented in the form $(I(u,v), \Lambda(u,v))$.*

Let's take the vertex pair (v_2, v_4) as an example to show how to update $I(v_2,v_4)$ and $\Lambda(v_2,v_4)$. We sequentially enumerate all the edges: (1) $(v_1,v_2,1)$: as there is a temporal wedge between v_2 and v_4, we increase $\Lambda(v_2,v_4)$ to 1. (2) $(v_1,v_4,2)$: the temporal wedge of $(v_1,v_4,0)$ and $(v_1,v_2,1)$ expires since $(v_1,v_4,0)$ is a historical edge, but there is another temporal wedge between v_2 and v_4, so $\Lambda(v_1,v_2)$ is still 1. (3) $(v_2,v_4,2)$: we increase $I(v_2,v_4)$ to 1.

Table 3: Dynamically maintain $I(u,v)$ and $\Lambda(u,v)$.

Static edges	Temporal edges					
–	$(v_1,v_4,0)$	$(v_3,v_4,0)$	$(v_1,v_2,1)$	$(v_1,v_4,2)$	$(v_2,v_3,2)$	$(v_2,v_4,2)$
(v_1,v_4)	(1 , 0)	(1 , 0)	(1 , 0)	(1 , 0)	(1 , 0)	(1 , 0)
(v_1,v_2)	(0 , 0)	(0 , 0)	(1 , 0)	(1 , 0)	(1 , 0)	(1 , 0)
(v_2,v_4)	(0 , 0)	(0 , 0)	(0 , 1)	(0 , 1)	(0 , 1)	(1 , 1)
(v_2,v_3)	(0 , 0)	(0 , 0)	(0 , 0)	(0 , 0)	(1 , 0)	(1 , 0)
(v_3,v_4)	(0 , 0)	(1 , 0)	(1 , 0)	(0 , 0)	(0 , 0)	(0 , 0)

Lemma 1. *Given a temporal graph G, a query time window $[t_s, t_e]$, and a threshold δ, OTTC completes in $O(m\kappa)$ time.*

Proof. In OTTC, for each temporal edge (u,v,t), we need to access each vertex in $N^+(u)$ constant times. Since $|N^+(u)| = d^+(u) \leq \kappa$, the total time complexity of OTTC is $O(m\kappa)$. □

4 AN INDEX-BASED SOLUTION

To enable frequent δ-temporal triangle counting queries, in this section we propose an index-based solution that offer accelerated response times while keeping acceptable preprocessing time and space costs. One direct index-based solution is to run the online algorithm for all possible time windows $[t_s, t_e]$ and durations δ, and then store the counting results. This, however, incurs an index space cost of $O(t^3_{max})$ and requires $O(m\kappa \times t^3_{max})$ time to build the index, making it impractical for large t_{max}. To alleviate this issue, we propose a novel index-based solution using the wavelet tree, which significantly reduces the indexing time and space costs. Moreover, the index-based counting algorithm is much faster than OTTC. In the following, we first present the main idea of our index, and then show the detailed index-based solution.

4.1 Main idea

Our main idea is to convert the δ-temporal triangles to some points, called *counting-points* or *C-points*, in the 2-dimensional space, and then show that the δ-temporal triangle counting problem can be solved by counting C-points. We define C-point as follows.

Definition 5 (C-point). *Given a temporal graph G, a C-point, denoted by $\langle (x,y), c \rangle$, means that the number of δ-temporal triangles in G satisfying: 1) δ=∞ and 2) the minimum and maximum timestamps of each triangle are exactly x and y respectively, is c.*

In the above definition, we set δ=∞, which basically means that any three temporal edges that form a triangle structure in the time window $[x,y]$ will be considered as a δ-temporal triangle. Assuming that G has t_{max} timestamps, there are $O(t^2_{max})$ possible different time windows, so the total number of C-points, denoted as Δ, is bounded by $O(t^2_{max})$. Meanwhile, the number of C-points is also bounded by the number of all the possible δ-temporal triangles in G. All these C-points can be placed on a 2-dimensional space of x and y, or more precisely in the area above the curve $y = x$ since $x \leq y$. Example 2 further illustrates the concept of C-point.

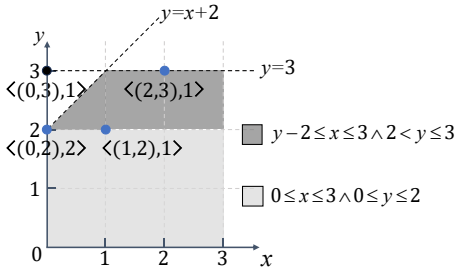


Figure 3: C-points of the temporal graph in Figure 1.

Example 2. *In Figure 1, there are 5 δ-temporal triangles in total (δ=∞), but there are 4 C-points, as depicted in the 2-dimensional space of Figure 3. For instance, the C-point $\langle (1,2), 1 \rangle$ indicates that in the time window $[1,2]$, there is one δ-temporal triangle whose minimum and maximum timestamps are exactly 1 and 2 respectively, since its three edges are $(v_1,v_2,1)$, $(v_1,v_4,2)$, and $(v_2,v_4,2)$.*

After converting all the δ-temporal triangles in the temporal graph to C-points, we can directly count the number of δ-temporal triangles for any arbitrary query parameters δ and $[t_s, t_e]$, by only using these C-points without accessing the original graph. Specifically, we can first collect all the C-points $\langle (x, y), c\rangle$ satisfying $x \geq t_s$, $y \leq t_e$, and $y - x \leq \delta$. Afterward, we just need to add their c values together to get the final counting result.

With careful observation, we find that all these C-points are actually covered by a trapezoid area and a rectangle area :

- **Trapezoid area:** $y - \delta \leq x \leq t_e$ and $t_s + \delta < y \leq t_e$.
- **Rectangle area:** $t_s \leq x \leq t_e$ and $t_s \leq y \leq t_s + \delta$.

Example 3. *Reconsider Example 2 and let $[t_s, t_e] = [0, 3]$ and $\delta = 2$. To answer this counting, we need to collect all the C-points (marked in blue) covered by a square area ($0 \leq x \leq 3$ and $0 \leq y \leq 2$) and a trapezoid area ($y - 2 \leq x \leq 3$ and $2 < y \leq 3$) which are marked in two different colored areas. The counting result is the sum of all the c values of these C-points, i.e., 2 + 1 + 1 = 4.*

Next, we introduce efficient algorithms for generating C-points and indexing C-points to support δ-temporal triangle counting.

4.1.1 Converting δ-temporal triangles to C-points: To generate C-points, we sequentially enumerate the edges in G, and for each edge (u, v, t), we count the δ-temporal triangles ($\delta = t_{max}$) that include u and v and take t as the maximum timestamp, with three steps:

(1) Find the vertex set $W = N(u) \cap N(v)$;
(2) For each vertex $w \in W$, we enumerate each edge (w, v, t') in $E(w, v)$, then count the number c of δ-temporal triangles that take t' and t as the minimum and maximum timestamps respectively which can be completed efficiently by using binary search, and finally obtain a C-point $\langle (t', t), c\rangle$.
(3) For each vertex $w \in W$, we also enumerate each edge in $E(u, w)$ and obtain a list of C-points in a similar manner.

Algorithm 2: Convert δ-temporal triangles to C-points

Input: A temporal graph $G = (V, E)$
Output: A list of C-points L

```
1  L ← ∅;
2  foreach (u, v, t) ∈ G do E(u, v) ← ∅;
3  foreach (u, v, t) ∈ G with timestamps in ascending order do
4      append (u, v, t) to E(u, v);
5      foreach w ∈ N(u) ∩ N(v) do
6          foreach (w, v, t') ∈ E(w, v) in ascending order of t' do
7              c ← |{(u, w, t'') | (u, w, t'') ∈ G ∧ t'' ∈ [t', t]}|;
8              append ⟨(t', t), c⟩ to L;
9          foreach (u, w, t') ∈ E(u, w) in ascending order of t' do
10             c ← |{(w, v, t'') | (w, v, t'') ∈ G ∧ t'' ∈ (t', t]}|;
11             append ⟨(t', t), c⟩ to L;
12 return L;
```

Algorithm 2 shows the details. We first initialize a list L and some edge lists (lines 1-2). Then, we generate C-points by enumerating the edges chronologically (lines 3-11). Specifically, we first update $E(u, v)$ (line 4), then find a vertex set $W = N(u) \cap N(v)$ (line 5), and finally count the δ-temporal triangles ($\delta = t_{max}$) that include vertices $\{u, v, w\}$ and take t as the maximum timestamp to obtain the C-points (lines 6-11). Finally, we get all the C-points (line 12).

Lemma 2. *Given a temporal graph G, Algorithm 2 completes in $O(m\kappa + \Delta \log(m))$ time, where Δ denotes the total number of C-points.*

Proof. For lines 3-5, the time cost of finding $N(u) \cap N(v)$ is the same as that of finding static triangles in G^*, which is $O(m\kappa)$ [2]. For lines 6-11, we can finish counting in $O(|E(w, v)| \log(|E(u, w)|) + |E(u, w)| \log(|E(w, v)|))$ time. Note that $O(|E(w, v)| + |E(u, w)|)$ equals to the time complexity of the for-loops at lines 6-9, where each iteration produces a C-point, so the time cost of lines 6-9 is $O(\Delta)$. The total time cost of lines 6-11 is bounded by $O(\Delta \log(m))$. Hence, Algorithm 2 costs $O(m\kappa + \Delta \log(m))$ time. □

4.1.2 Indexing C-points for counting δ-temporal triangles: After obtaining all the C-points, a natural method to count the δ-temporal triangles is to collect all the C-points $\langle (x, y), c\rangle$ satisfying $x \geq t_s$, $y \leq t_e$, and $y - x \leq \delta$, and summarize their c values together. This method, however, is very costly since it takes $O(\Delta)$ time, and may be even slower than our online algorithm OTTC in some cases.

We notice that in the literature, many effective index structures, such as wavelet tree [25], kd-tree [5], Fenwick tree [22], segment tree [15], etc., have been developed for supporting range search, which aim to efficiently count the number of points in an arbitrary rectangle area in the 2-dimensional space. As aforementioned, the C-points for counting the δ-temporal triangles in $[t_s, t_e]$ are actually covered by a rectangle area and a trapezoid area. For the C-points in the rectangle area, we can efficiently count them by building these index structures for all the C-points of the whole graph.

However, for the C-points in the trapezoid area, they cannot be counted directly by employing these index structures since they focus on rectangle areas. To resolve this issue, we propose to change the 2-dimensional location of each C-point, so that the set of C-points in any original trapezoid area can be covered by a rectangle area, making the above existing index structures applicable. Recall that for any C-point $\langle (x, y), c\rangle$ in the trapezoid area, we have

$$\begin{cases} y - \delta \leq x \leq t_e \\ x \leq y \end{cases} \implies 0 \leq y - x \leq \delta \tag{1}$$

Since $x \leq y$ and $y - x \leq \delta$, by setting $z = y - x$, we further obtain

$$\begin{cases} 0 \leq y - x \leq \delta \\ t_s + \delta < y \leq t_e \end{cases} \overset{z=y-x}{\implies} \begin{cases} 0 \leq z \leq \delta \\ t_s + \delta < y \leq t_e \end{cases} \tag{2}$$

Hence, for any C-point $\langle (x, y), c\rangle$ in the 2-dimensional space of x and y, by changing x to $z = y - x$, we can obtain another kind of C-point $\langle (z, y), c\rangle$, which is in the 2-dimensional space of z and y. To distinguish it with the original C-point, we call it a $\widehat{\text{C}}$-point.

A nice feature of all the $\widehat{\text{C}}$-points is that they must be in a rectangle area. Thus, by making such changes, all the C-points in the original trapezoid area can be covered by a rectangle area, so the existing index structures for range search can be used as well.

Example 4. *Consider all the C-points in the 2-dimensional space of x and y in Figure 3. By changing x to $z = y - x$ for each C-point, we get all the $\widehat{\text{C}}$-points in another 2-dimensional space of z and y in Figure 4. For instance, a C-point $\langle (1, 3), 1\rangle$ is changed to $\langle (2, 3), 1\rangle$. Besides, for all the C-points in the trapezoid area of Figure 3, their corresponding $\widehat{\text{C}}$-points are in a rectangle area marked in grey.*

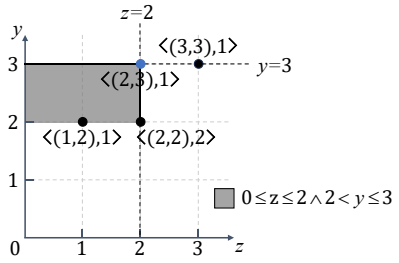


Figure 4: Changing C-points to $\widehat{\text{C}}$-points.

As shown in the literature, the Chazelle's structure [10], a.k.a. wavelet tree [25], achieves a better balance between time and space than others for counting points in the 2-dimensional space [17]. Thus, in this paper we employ it to index all the C-points and $\widehat{\text{C}}$-points respectively, and call the wavelet tree-based index `WT-Index`.

4.2 Our `WT-Index`-based solution

We first introduce the `WT-Index` for all the C-points, then discuss the index-based counting algorithm, and finally show the index maintenance algorithm for dynamic temporal graphs. The `WT-Index` for $\widehat{\text{C}}$-points can be built similarly, so we omit the details.

4.2.1 Overview of `WT-Index`. Given a C-point list L, a `WT-Index` is a binary tree, where each node[1] has a time interval and stores the information of C-points whose x value falls in this interval. To ease the illustration, we declare that a tree node *includes* a C-point if its time interval covers the C-point's x value. Specifically, each node of `WT-Index` has three key components:

(1) **A time interval $[l, r]$:** It indicates that the node stores the information of all the C-points $\langle (x, y), c\rangle$ with $x \in [l, r]$. The root node has the largest interval $[0, T]$ with $T < 2t_{max}$. For each node, if $l \neq r$, then it has two children whose time intervals are $\left[l, \lfloor \frac{l+r}{2} \rfloor\right]$ and $\left[\lfloor \frac{l+r}{2} \rfloor + 1, r\right]$ respectively; otherwise, it is a leaf node with $l = r$.
(2) **An integer array C[ ]:** It stores the c values of all the C-points included by the node, where all the C-points are assumed to be sorted in ascending order of their y values.
(3) **A boolean array Pos[ ]:** It records which child nodes include the C-points that the current node includes. Given that all the C-points included in the current node are sorted in ascending order of their y values, if the i-th C-point is also included by the left node, then $Pos[i]$=0; otherwise, $Pos[i]$=1. For leaf nodes, their $Pos[\,] = \emptyset$.

In addition, to enable efficient retrieval, we also add a prefix sum array for C[ ] and Pos[ ] in each node, respectively.

We further illustrate the `WT-Index` via Example 5.

Example 5. *Figure 5 depicts the* `WT-Index` *built for all the C-points in Figure 3. For instance, the internal node with time interval* $[0, 1]$, *it includes three C-points, i.e.,* $\langle (0, 2), 2\rangle$, $\langle (0, 3), 1\rangle$ *and* $\langle (1, 2), 1\rangle$, *so* $C[1] = 2$, $C[2] = 1$, *and* $C[3] = 1$. *Since these three C-points are also included by its left and right child nodes respectively, we have* $Pos[1] = 0$, $Pos[2] = 1$, *and* $Pos[3] = 0$.

[1]To avoid ambiguity, we use "node" to mean a node of the tree index, and use "vertex" to denote a graph vertex in this paper.

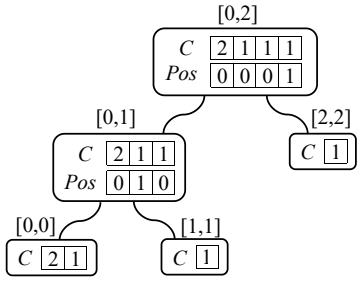


Figure 5: `WT-Index` for the C-points in Figure 3.

As shown in [10], given a list of 2-dimensional points, we can build the `WT-Index` efficiently in a bottom-up manner: we first initialize all the leaf nodes, and then iteratively build their parent nodes by merging child nodes' information until the root is built, which is similar to the process of MergeSort. The indexing time and space costs [10] of building the `WT-Index` are $O(\Delta \log(\Delta))$ and $O(m+\Delta \log(\bar{c}))$, where $\bar{c}$ is the average of the c values of all C-points, and the space cost of `WT-Index` is $O(\Delta \log(\bar{c}))$.

4.2.2 `WT-Index`*-based counting algorithm.* Recall that the number of δ-temporal triangles in $[t_s, t_e]$ equals to the summarized value of the c values of all the C-points in a rectangle area and a triangle area. Since the triangle area can be transferred to a rectangle area by changing C-points to $\widehat{\text{C}}$-points. Therefore, the key question is that given a rectangle area: $t_s \leq x \leq t_e$ and $t_s \leq y \leq t_s+\delta$, how to obtain the summarized value of the c values of all the C-points it covers. Since this rectangle is the difference of the other two rectangle areas: (1) $t_s \leq x \leq t_e$ and $0 \leq y \leq t_s + \delta$ and (2) $t_s \leq x \leq t_e$ and $0 \leq y \leq t_s - 1$, we can derive the summarized values from them respectively and subtract them. As these two rectangle areas can be processed by the same procedure, we design an efficient algorithm to obtain the summarized value of the c values of all the C-points in a rectangle area with $t_s \leq x \leq t_e - \delta$ and $0 \leq y \leq \theta$ where $\theta \in \{t_s + \delta, t_s - 1\}$.

Specifically, we start from the root of the `WT-Index` and recursively find all the nodes whose time intervals overlap with $[t_s, t_e]$ level by level in a top-down manner, during which we add the c values of all the C-points in these nodes. Algorithm 3 shows the details. We first get the root and obtain the number of C-points it includes (lines 1-2). Then, we use a function `sum` to sum the c values recursively (lines 3-10). For each node, if its time interval is covered by $[t_s, t_e - \delta]$, we sum up the c values of the C-points it includes (lines 5-6); if its time interval is not intersected with $[t_s, t_e - \delta]$, the count is 0; otherwise, we need to traverse its left and right child nodes and get the summarized result recursively (lines 7-10).

According to [10], by storing $\sum_{i=1}^{p} node.Pos[i]$ and $\sum_{i=1}^{p} node.C[i]$ for $p \mod \log(\Delta) = 0$ to accelerate the calculation, Algorithm 3 can complete in $O(\log^2(\Delta))$ time. Thus, the overall time cost of the index-based counting algorithm is also $O(\log^2(\Delta))$.

4.2.3 Index maintenance. In real-world applications, the temporal graphs are often continually updating, due to the generation of new temporal edges, which requires the `WT-Index` to be updated as well. A simple method of updating the index is to rebuild the `WT-Index` from scratch whenever a new temporal edge is inserted, but it is extremely costly when the new edges are frequently inserted.

Algorithm 3: Sum the c values of C-points in a rectangle

Input: WT-Index, a rectangle area $t_s \le x \le t_e$ and $0 \le y \le \theta$
Output: The summarized c values of C-points in the rectangle

```
1  node ←root of the WT-Index;
2  p ← # of C-points whose y ≥ θ;
3  Function sum(node, p):
4      [l, r] ←the time interval of node;
5      if [l, r] ⊆ [t_s, t_e] then
6          return Σ_{i=1}^{p} node.C[i];
7      else
8          o ← Σ_{i=1}^{p} node.Pos[i];
9          if [l, r] ∩ [t_s, t_e] = ∅ then return 0;
10         else return sum(node.lChild, p−o)+sum(node.rChild, o);
```

A few works have studied the maintenance of wavelet trees, but they are often limited to very specific scenarios. For instance, [14] presents an algorithm for maintaining the wavelet tree, but it aims to count the frequency of some texts, which actually counts points in a 1-dimensional space, so it cannot process our C-points.

In this paper, we develop a novel efficient maintenance algorithm to update the WT-Index for the C-points after inserting a new temporal edge $(u, v, t_{max}+1)$ into the graph. The WT-Index for the $\widehat{\text{C}}$-points can be maintained similarly, so we omit its details. Our key idea is that after an edge insertion, we first identify all the newly formed δ-temporal triangles (δ=∞) and convert them into some C-points, which can be easily implemented by following Algorithm 2 since it generates C-points by sequentially processing the temporal edges one by one. Afterward, we update the WT-Index to incorporate the information of the new C-points. Specifically, for each new C-point $\langle (x, y), c\rangle$, we consider two cases:

1) $\langle (x, y), c\rangle$ **is included by the root:** we can start from the root of the WT-Index and recursively update the information of all the nodes including the new C-point level by level in a top-down manner.
2) $\langle (x, y), c\rangle$ **is not included by the root:** Let the time interval of the root be $[0, T]$. We first create a new root with a time interval $[0, 2T]$ and initialize its arrays using the information in the original root, then continue the generating process until the new root includes $\langle (x, y), c\rangle$, and finally use the idea of case 1) to update the remaining nodes.

Algorithm 4 shows our maintenance algorithm. We first generate new C-points by Algorithm 2 (line 1). Then, for each new C-point $\langle (x, y), c\rangle$, we check whether it is included by the root of WT-Index (lines 2-4). If not, we generate new roots until the new root includes it (lines 5-6). Next, we use a recursive function to update the nodes' information from the root node (lines 7-16). In update, we append the c value to $node.C[\ ]$ of the current node (line 9), then if the current node is a leaf node, we terminate the recursion (line 10); otherwise, we check which child node includes the new C-point to update $node.Pos[\ ]$ and continue the recursion (lines 11-16).

Example 6. *Figure 6(a) shows the* WT-Index *for C-points of the projected graph* $G_{[0,2]}$ *in Figure 2(a). After inserting edges with timestamp 3, we get a new C-point* $\langle (2, 3), 1\rangle$. *By Algorithm 4, since* $2 \notin [0, 1]$, *we first generate a new root node with time interval* $[0, 2]$. *We then use* update *function to update* WT-Index. *Specifically, we first append 1 to* $C[\ ]$ *and* $Pos[\ ]$ *of the root node, then go to the right child of the root since it includes the new C-point, and finally append 1 to* $C[\ ]$ *of the leaf node with time interval* $[2, 2]$.

Algorithm 4: WT-Index maintenance algorithm

Input: G, WT-Index for G, and a new edge $(u, v, t_{max}+1)$
Output: The updated WT-Index

```
1  run lines 3-11 of Algorithm 2 to generate a new list L′ of C-points;
2  foreach C-point ⟨(x, y), c⟩ ∈ L′ do
3      node ← root of the WT-Index with time interval [0, T];
4      while node does not include ⟨(x, y), c⟩ do
5          generate a new root with time interval [0, 2T] storing the
             information of the original C-points;
6          node ← new root of the WT-Index;
7      update(node, ⟨(x, y), c⟩);
8  Function update(node, ⟨(x, y), c⟩):
9      append c to the end of node.C[ ];
10     if node is a leaf node then return;
11     if node.lChild includes ⟨(x, y), c⟩ then
12         append 0 to the end of node.Pos[ ];
13         update(node.lChild,⟨(x, y), c⟩);
14     else
15         append 1 to the end of node.Pos[ ];
16         update(node.rChild,⟨(x, y), c⟩);
```

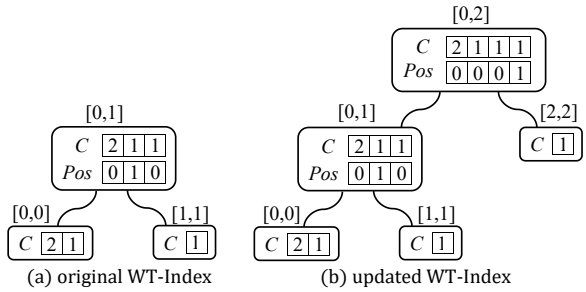


Figure 6: An example of updating WT-Index.

Lemma 3. *Given a temporal graph* G *and its* WT-Index, *Algorithm 4 costs* $O((|L'| + \Delta)\log(t_{max}))$ *time to process an edge insertion.*

Proof. Assume the root node of the WT-Index has a time interval $[0, T]$ where $T < 2t_{max}$. Then, the tree height of WT-Index is $O(\log(t_{max}))$. To generate a new root node, we have to copy all the information from the original root, which takes $O(\Delta)$ time, and the number of new root nodes is bounded by $O(\log(t_{max}))$, so generating the root nodes costs at most $O(\Delta \log(t_{max}))$ time. Besides, the time complexity of running update is linear to the height of the tree, so it costs $O(\log(t_{max}))$ time. Since there are $|L'|$ C-points, the overall cost of running update is $O(|L'|\log(t_{max}))$ time. Hence, the total time cost of Algorithm 4 is $O((|L'| + \Delta)\log(t_{max}))$. □

5 BINARY δ-TEMPORAL TRIANGLE COUNTING

As shown in Definition 2, multiple δ-temporal triangles may share the same three vertices, due to the existence of multiple temporal edges between two vertices. In many real-world scenarios, we may

only need to consider the existence of δ-temporal triangle [24, 31], so we introduce the binary δ-temporal triangle counting problem:

Problem 2 (Binary δ-temporal triangle counting). *Given a temporal graph G, a time window $[t_s, t_e]$, and a duration δ ($\delta \geq 0$), return the number of sets of vertices, each of which contains three vertices and forms at least one δ-temporal triangle within $G_{[t_s,t_e]}$, where each such set is called a binary δ-temporal triangle.*

We illustrate Problem 2 via Example 7.

Example 7. *In the temporal graph of Figure 1(a), let $[t_s, t_e]$=[0,2] and δ=2. The projected graph $G_{[0,2]}$ is depicted in Figure 2(a). Clearly, there are two binary 2-temporal triangles in $G_{[0,2]}$, i.e., $\{v_1, v_2, v_4\}$ and $\{v_2, v_3, v_4\}$. Note that there are three 2-temporal triangles since two 2-temporal triangles share a vertex set $\{v_1, v_2, v_4\}$.*

To solve Problem 2, we propose an efficient online algorithm and an index-based solution in Sections 5.1 and 5.2, respectively.

5.1 Our online algorithm BTTC

In Problem 2, when multiple δ-temporal triangles share the same set of three vertices, they will be counted only once, thereby making both OTTC and DOTTT inapplicable. Nevertheless, we can adapt DOTTT to solve Problem 2 by slightly changing step (3) of DOTTT; that is, after deriving the value R, if $R > 1$, we lower it to 1. We denote the adapted algorithm by B-DOTTT.

However, B-DOTTT is inefficient due to the complex process of computing R. We further develop a novel faster algorithm, denoted as BTTC, which follows the three steps of DOTTT, but changing its step (3) to check the existence of a δ-temporal triangle. Specifically, given a static triangle $\Delta_{u,v,w}$ of $G^*_{[t_s,t_e]}$, we enumerate each edge (u, v, t) in $E(u, v)$, and find edges $e_1 \in E(u, w), e_2 \in E(v, w)$ such that both of them appear in a time interval either $[t, t + \delta]$ or $[t - \delta, t]$. If such two edges exist, we identify a δ-temporal triangle with (u, v, t) as the first edge or last edge. The edge (u, v, t) may also serve as the middle edge in the δ-temporal triangle, and to find such triangles, we find δ-temporal triangles with (u, w, t') as the first edge or last edge through a similar process.

Algorithm 5 presents BTTC. We first extract $G^*_{[t_s,t_e]}$ and P, and initialize R (lines 1-2). Then, we enumerate static triangles in $G^*_{[t_s,t_e]}$, and check the existence of δ-temporal triangle sharing the vertices in each static triangle (lines 3-18). We initialize the flag of existence and find u (line 5). For each edge (u, v, t), check the existence of δ-temporal triangle with (u, v, t) as the first edge or the last edge (lines 6-11). We also enumerate edges $\in E(u, w)$ to check the existence of δ-temporal triangle with (u, w, t') as the first edge or the last edge (lines 12-17). Finally, we get the result (lines 18-19).

Example 8. *Consider the temporal graph in Figure 1(a) and let $[t_s, t_e] = [0, 2]$ and $\delta = 2$. We first get the static graph $G^*_{[0,2]}$ with 2 static triangles Δ_{v_1,v_2,v_4} and Δ_{v_2,v_3,v_4}, shown in Figure 2 and derive its degeneracy order $P : v_1 \prec v_3 \prec v_4 \prec v_2$. Then, for Δ_{v_1,v_2,v_4}, when enumerating edge $(v_1, v_2, 1)$, we find $(v_1, v_4, 2)$ and $(v_2, v_4, 2)$, so there is a binary 2-temporal triangle $\{v_1, v_2, v_4\}$. For Δ_{v_2,v_3,v_4}, we can also find a binary 2-temporal triangle. Hence, the counting result is 2.*

Lemma 4. *Given a temporal graph G, a query time window $[t_s, t_e]$, and a duration δ, BTTC completes in $O(m\kappa \log(m))$ time.*

Algorithm 5: BTTC

Input: A temporal graph G, a time window $[t_s, t_e]$, a threshold δ
Output: The number of binary δ-temporal triangles in $G_{[t_s,t_e]}$

1 Extract the static graph $G^*_{[t_s,t_e]}$ and derive its degeneracy order P;
2 $R \leftarrow 0$;
3 Enumerate the static triangles in $G^*_{[t_s,t_e]}$;
4 **foreach** *static triangle* $\Delta_{u,v,w} \in G^*_{[t_s,t_e]}$ **do**
5 $f \leftarrow 0$, $u \leftarrow$ vertex with the minimum degeneracy order;
6 **foreach** $(u, v, t) \in E(u, v)$ **do**
7 $S_1 \leftarrow \{(u, w, t') | (u, w, t') \in E(u, w), t' \in [t, t + \delta]\}$;
8 $S_2 \leftarrow \{(v, w, t'') | (v, w, t'') \in E(v, w), t'' \in [t, t + \delta]\}$;
9 $S_3 \leftarrow \{(u, w, t') | (u, w, t') \in E(u, w), t' \in [t - \delta, t]\}$;
10 $S_4 \leftarrow \{(v, w, t'') | (v, w, t'') \in E(v, w), t'' \in [t - \delta, t]\}$;
11 **if** ($|S_1|$ and $|S_2| > 0$) or ($|S_3|$ and $|S_4| > 0$) **then** $f \leftarrow 1$;
12 **foreach** $(u, w, t') \in E(u, w)$ **do**
13 $S_1 \leftarrow \{(u, v, t) | (u, v, t) \in E(u, v), t \in [t', t' + \delta]\}$;
14 $S_2 \leftarrow \{(v, w, t'') | (v, w, t'') \in E(v, w), t'' \in [t', t' + \delta]\}$;
15 $S_3 \leftarrow \{(u, v, t) | (u, v, t) \in E(u, v), t \in [t' - \delta, t']\}$;
16 $S_4 \leftarrow \{(v, w, t'') | (v, w, t'') \in E(v, w), t'' \in [t' - \delta, t']\}$;
17 **if** ($|S_1|$ and $|S_2| > 0$) or ($|S_3|$ and $|S_4| > 0$) **then** $f \leftarrow 1$;
18 $R \leftarrow R + f$;
19 **return** R;

Proof. As analyzed in Section 3.1, the time cost of steps (1) and (2) is $O(m\kappa)$. Step (3) (lines 4-18 in Algorithm 5) costs $O(\sum_{\Delta_{u,v,w}} (|E(u, v)| + |E(u, w)|) \log(m)) = O(m\kappa \log(m))$ time. Hence, the lemma holds. □

5.2 An index-based solution

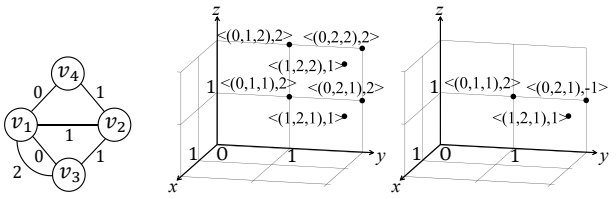
5.2.1 Main idea. Similar to WT-Index, we propose to convert the binary δ-temporal triangle problem into a point counting problem. However, the idea of converting δ-temporal triangle to C-points cannot be applied to binary δ-temporal triangle as multiple δ-temporal triangles may be regarded as a single binary δ-temporal triangle.

To resolve the above issue, we proposed to store the binary δ-temporal triangle counting results, rather than the binary δ-temporal triangles. A naive idea is to run BTTC for every possible time window $[x, y]$ and duration z, and then keep a 3-dimensional point $\langle (x, y, z), c_{(x,y,z)} \rangle$ where $c_{(x,y,z)}$ denotes the number of binary z-temporal triangles in $[x, y]$. However, it costs $O(t^3_{max})$ space and $O(m\kappa \log(m) \times t^3_{max})$ time to build the index, thereby being impractical for large t_{max}. To alleviate this issue, we propose to compress these 3-dimensional points into some BC-points, defined as follows, by only storing the differences between counting results.

Definition 6 (BC-point). *Given a temporal graph G, a time window $[x, y]$, and a duration z, a BC-point, denoted by $\langle (x, y, z), d \rangle$, is a 3-dimensional point with*

$$d = c_{(x,y,z)} - c_{(x,y,z-1)} - c_{(x,y-1,z)} - c_{(x+1,y,z)} + c_{(x+1,y-1,z)} + c_{(x+1,y,z-1)} + c_{(x,y-1,z-1)} - c_{(x+1,y-1,z-1)}.$$

Note that when $d = 0$, we omit the BC-point due to the zero difference. Clearly, given all the BC-points, the number of binary δ-temporal triangles in the time window $[t_s, t_e]$ equals the summarized d values of BC-points in the cube $[t_s, t_e] \times [t_s, t_e] \times [0, \delta]$.



(a) A temporal graph G_2 (b) 3-dimensional points (c) All the BC-points

Figure 7: A temporal graph and its 3-dimensional points before and after compression (we omit points with $c = d = 0$).

Example 9. *In Figure 7, we present a temporal graph G_2, and its 3-dimensional points before and after compression, where points with c and d being 0 are omitted. For instance, there is a BC-point $\langle(0, 2, 1), -1\rangle$ because $c_{(0,2,1)} - 0 - c_{(0,1,1)} - c_{(1,2,1)} + 0 + 0 + 0 - 0 = -1$.*

To obtain BC-points, a naive method is to enumerate all the possible time windows and durations and compress the 3-dimensional points, requiring $O(m\kappa \log(m) \times t_{max}^3)$ time, which is very costly. To speed up this process, we generate BC-points from each static triangle. Specifically, for each static triangle $\Delta_{u,v,w}$, we first fetch all the timestamps of edges in $E(u, v)$, $E(u, w)$, and $E(w, v)$, denoted by a set Φ. Then, we only need to obtain the $c_{(x,y,z)}$ for $\Delta_{u,v,w}$ with every time window $[x, y]$ and duration z where $x, y, z \in \Phi$. Finally, we get all the BC-points.

As aforementioned, to answer a counting query, we need to summarize the d values of BC-points in the cube $[t_s, t_e] \times [t_s, t_e] \times [0, \delta]$, which actually is a 3-dimensional point counting problem. Many index structures have been developed for solving this problem, such as KD-tree [5], wavelet tree [25], and segment tree [15]. Given Δ BC-points, since KD-tree costs $O(\Delta)$, while the other two indexes cost $O(\Delta \log(\Delta))$ space, we employ the KD-tree in this paper and denote the KD-tree-based index by `KDT-Index`.

5.2.2 `KDT-Index`. Given a BC-point list L, a `KDT-Index` is a binary tree, where each node stores a BC-point and the information of BC-points in its subtrees, with four components in total:

(1) **A BC-point** $\langle(x, y, z), d\rangle$.
(2) **A selected dimension** D: It indicates the construct rule of the KD-tree. For any BC-point, if its coordinate in D-th dimension is smaller than or equal to the D-th dimension of $\langle(x, y, z), d\rangle$, it will be in the left child of the node; otherwise, it will be in the right child of the node.
(3) **A cube** A: The smallest cube containing all BC-points in the subtree of the node, denoted by $[x_1, x_2] \times [y_1, y_2] \times [z_1, z_2]$.
(4) **An integer** S: It records the summarized d value of all BC-points in the subtree of the node.

We further illustrate the `KDT-Index` via Example 10.

Example 10. *Figure 8 depicts the* `KDT-Index` *built for all BC-points in Figure 7. Let us take the root node as example: we pick a BC-point $\langle(0, 2, 1), -1\rangle$ and select the x dimension ($D = x$). The smallest cube A to contain all the BC-points is $[0, 1] \times [1, 2] \times [1, 1]$. Since the summarized d value of all BC-points is 2, $S = 2$.*

The result of a counting query equals the summarized S values of nodes in the `KDT-Index` whose cubes are contained by the cube $[t_s, t_e] \times [t_s, t_e] \times [0, \delta]$. We apply a recursive method starting from the root to find the counting result. For lack of space, we show the counting algorithm, `KDT-Index` construction and maintenance algorithms [5, 16, 38] in the appendix of our technical report [1].

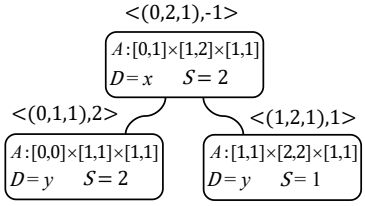


Figure 8: The `KDT-Index` for the BC-points in Figure 7

6 EXPERIMENTS

We now present the experimental results. We begin with the setup in Section 6.1, then show the efficiency results in Sections 6.2 and 6.3, and finally present case study results in Section 6.4.

6.1 Setup

Table 4: Datasets used in our experiments.

Datasets	Abbr.	n	m	t_{max}	Δ	κ
contact	CT	274	28,244	15,661	15M	39
email-eu	EM	986	332,334	207,879	211M	34
wiki-talk	WK	1,140,149	7,833,140	5,799,205	651M	119
stackoverflow	ST	2,601,977	63,497,050	41,484,768	2.7B	198
graph500-23	GR	4,610,222	129,333,677	6,807,835	3.2B	1,222

Datasets. We use four real-world temporal graphs sourced from SNAP [39] and KONECT [36], along with a synthetic temporal graph generated from LDBC [20]. Table 4 summarizes the statistics of each graph, including the numbers of vertices (n) and edges (m), the maximum timestamp (t_{max}), the number of C-points (Δ), and the degeneracy (κ). The last graph does not have timestamps, so we randomly generate timestamps for it.

Algorithms. We mainly evaluate the following algorithms.
- `DOTTT` [43]: the SOTA δ-temporal triangle counting algorithm;
- `OTTC`: our online δ-temporal triangle counting algorithm;
- `WT-Index-query`: the `WT-Index` based δ-temporal triangle counting algorithm;
- `B-DOTTT`: adapted `DOTTT` for binary δ-temporal triangle counting;
- `BTTC`: our online binary δ-temporal triangle counting algorithm;
- `KDT-Index-query`: the `KDT-Index`-based binary δ-triangle counting algorithm.

To evaluate the efficiency, for each dataset, we consider 5 different interval lengths $|t_e - t_s| = t_{max} \cdot x$ with $x \in \{20\%, 40\%, 60\%, 80\%, 100\%\}$, and then for each interval length, we randomly generate 1,000 queries by varying $\delta = |t_e - t_s| \cdot y$ with $y \in \{10\%, 30\%, 50\%, 70\%, 90\%\}$, where the default interval length and δ are set to $100\% \times t_{max}$ and $10\%|t_e - t_s|$ respectively. All algorithms are implemented in C++ and compiled with the g++ compiler at the -O3 optimization level. The experiments are conducted on a Linux machine equipped with an Intel Xeon 2.90GHz CPU and 512GB RAM.

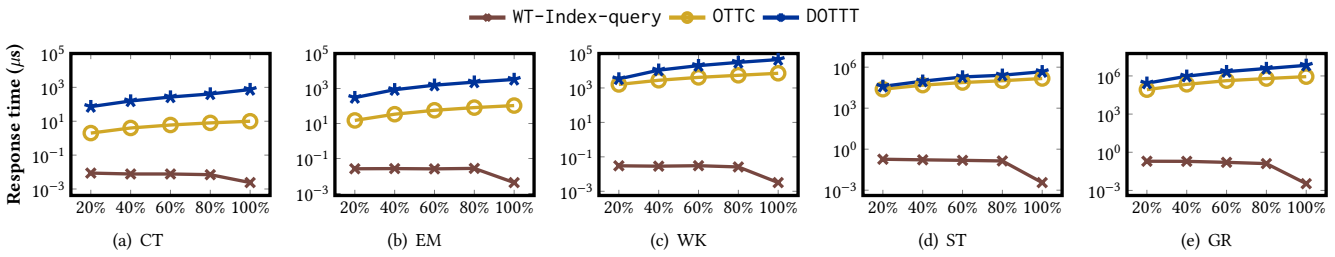


Figure 9: Effect of $(t_e - t_s)$ (which varies from 20%, 40%, 60%, 80%, and 100% of t_{max}).

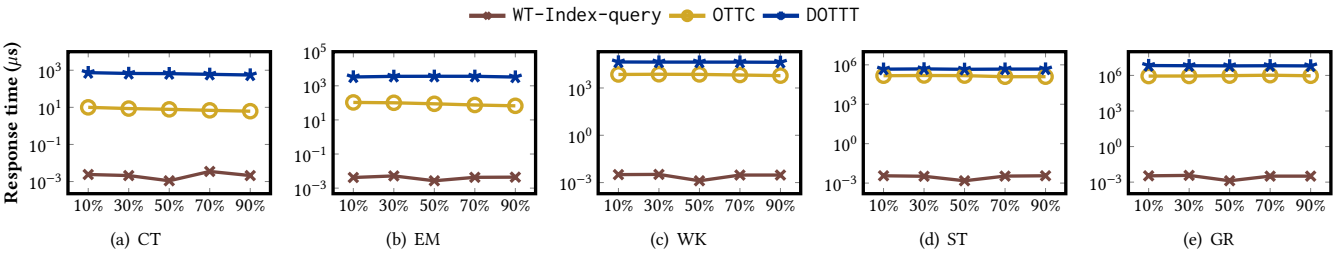


Figure 10: Effect of the value of δ (which varies from 10%, 30%, 50%, 70%, and 90% of t_{max}).

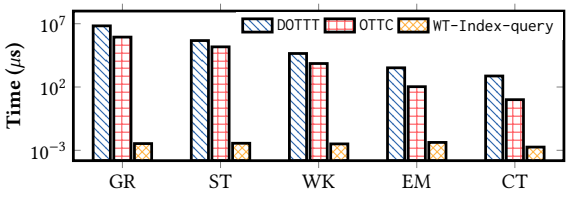


Figure 11: Efficiency of δ-temporal triangle counting.

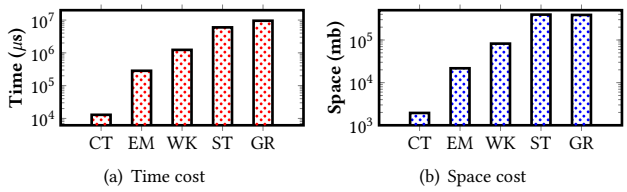


Figure 12: Time and space cost of `WT-Index` construction.

6.2 Efficiency of δ-temporal triangle counting

• **Overall results.** For each dataset, we report the average response time of each algorithm in Figure 11. Clearly, our online algorithm `OTTC` consistently outperforms `DOTTT` across all datasets. For example, on the CT dataset, `OTTC` is 70 × faster than `DOTTT`. Moreover, `WT-Index-query` demonstrates superior performance since it is up to eight orders of magnitude faster than both `OTTC` and `DOTTT`.

• **Effect of $(t_e - t_s)$.** In this experiment, we consider five different interval lengths: 20%, 40%, 60%, 80%, and 100% of t_{max}. For each interval length, we conduct 1,000 counting queries with t_s randomly selected, where δ is consistently set to 10% of each interval length, and depict the efficiency results in Figure 9. As the interval length increases, both `OTTC` and `DOTTT` experience longer response times, since the number of δ-temporal triangles increases. In contrast, the response time of `WT-Index` remains relatively stable, exhibiting a slight decrease. This is because when the query interval spans the entire duration $[0, t_{max}]$, the `WT-Index` operates with $O(1)$ time complexity for responses.

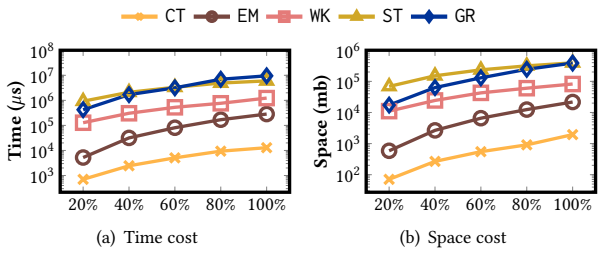


Figure 13: Scalability test of indexing time and space.

• **Effect of δ.** In this experiment, we set $[t_s, t_e] = [0, t_{max}]$, and vary $\delta = t_{max} \cdot y$ with $y \in \{10\%, 30\%, 50\%, 70\%, 90\%\}$. For each δ, we conduct 1,000 queries and record the average response time. The findings are summarized in Figure 10. Clearly, the choice of δ demonstrates little impact on counting time cost. Because the time complexity of all three algorithms has nothing to do with δ. Again, the index-based algorithm is much faster than all the algorithms.

• **Time and space costs of index construction.** In this experiment, we report the time and space costs of index construction for all graphs in Figure 12. Clearly, the time and space costs of the `WT-Index` scale with graph sizes. Note that ST takes more space cost than GR, primarily because the t_{max} for ST is larger than that for GR, resulting in a greater tree height for the `WT-Index` of ST.

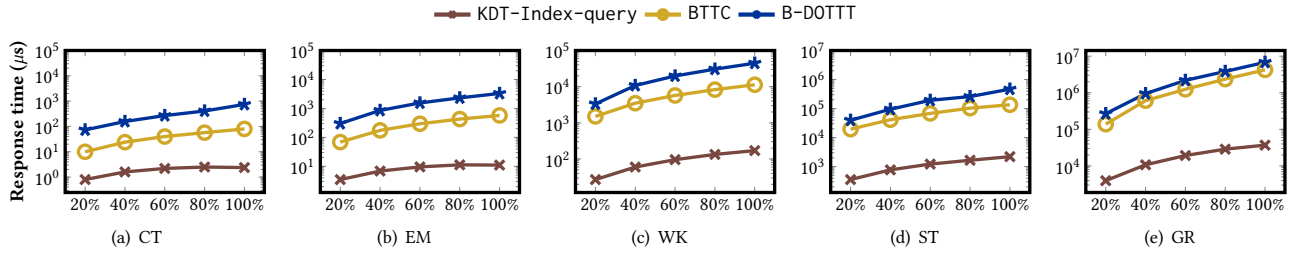


Figure 14: Effect of $(t_e - t_s)$ (which varies from 20%, 40%, 60%, 80%, and 100% of t_{max}).

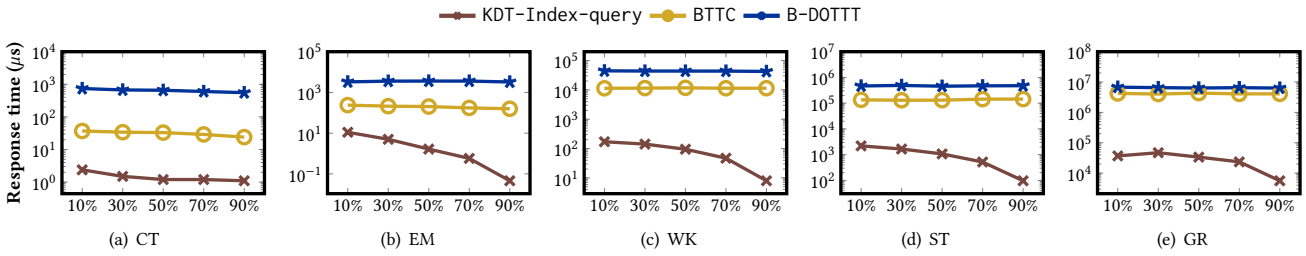


Figure 15: Effect of the value of δ (which varies from 10%, 30%, 50%, 70%, to 90% of t_{max}).

In the above experiments, the efficiency of the index-based counting algorithms is measured without considering index construction time, so it may be unfair when the indexing time should be considered. To make a fair comparison, we amortize the index construction time across the index-based queries and compare it with the online query algorithm OTTC. Table 5 reports the number of counting queries required for our index method to surpass the online algorithm for each dataset, where the queries span the full-length time interval with randomly selected δ. These values are remarkably low compared to the total number of timestamps, especially for larger graphs. Hence, even with a modest number of queries, the index-based algorithm consistently outperforms the online algorithm.

Table 5: Number of counting queries to offset indexing time.

Dataset	CT	EM	WK	ST	GR
Number of counting queries	1.6K	2.8K	172	41	11

• **Scalability test.** To evaluate the scalability of our index construction algorithm, we randomly select 20%, 40%, 60%, 80%, and 100% of the edges from each graph, thereby obtaining five induced subgraphs from these edges. We then building indices on these subgraphs of all datasets. As shown in Figure 13, the indexing time and space costs of our index construction algorithm increase linearly with the graph size, thereby demonstrating good scalability.

• **Index maintenance.** In this experiment, we first build the WT-Index by using edges in the range $[0, 0.8t_{max}]$. Afterward, we update the WT-Index by sequentially considering the remaining edges from $(0.8t_{max} + 1, t_{max}]$. The experiments show that the average time cost of updating WT-Index for each new edge on the five datasets are 2.03, 6.39, 2.13, 0.62 and 0.43 μs, respectively. Clearly, our proposed index maintenance algorithm is significantly faster than rebuilding WT-Index from scratch.

6.3 Efficiency of binary δ-temporal triangle counting

• **Overall results.** For each dataset, we report the average response time of each algorithm in Figure 16. Our online algorithm BTTC consistently outperforms B-DOTTT across all datasets. For example, on the CT dataset, BTTC is 23× faster than B-DOTTT. Moreover, KDT-Index-query achieves the best performance, as it is up to four orders of magnitude faster than BTTC.

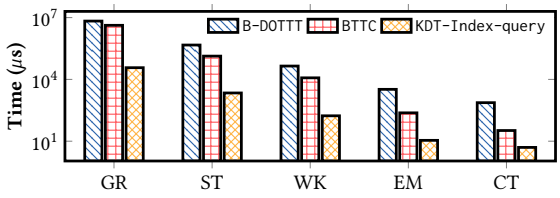


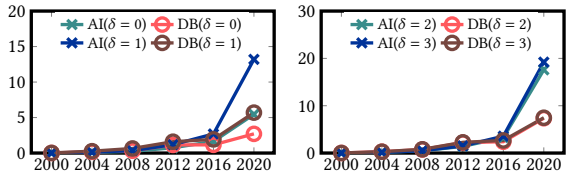
Figure 16: Efficiency of binary δ-temporal triangle counting.

• **Effect of $(t_e - t_s)$.** In this experiment, we test five different lengths: 20%, 40%, 60%, 80%, and 100% of t_{max}, with δ set to the default value. For each length, we execute 1,000 counting queries with randomly selected t_s and report the average response time in Figure 14. Again, the response time increases as the interval length increases since more binary δ-temporal triangles are involved.

• **Effect of δ.** In this experiment, we set $[t_s, t_e] = [0, t_{max}]$, and vary $\delta = t_{max} \cdot y$ with $y \in \{10\%, 30\%, 50\%, 70\%, 90\%\}$. For each δ, we conduct 1,000 queries and record the average response time. The findings are summarized in Figure 15. We observe that δ has little effect on the efficiency on BTTC and B-DOTTT. But it affects the efficiency of KDT-Index-query a lot because when δ goes larger, the response time complexity of KDT-Index approaches $O(1)$.

6.4 Case study

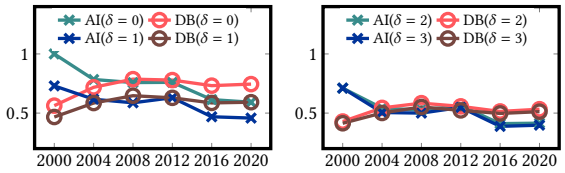
We conduct a case study on the temporal co-authorship networks of papers published in database (DB) and artificial intelligence (AI) areas during 2000 to 2023. Specifically, we first identify the top-50 most frequent keywords in titles of papers in SIGMOD, VLDB, and ICDE as representative DB keywords, and the top-50 most frequent keywords in titles of papers in NIPS, ICML, and ICLR as representative AI keywords (stopwords are omitted). Then, we classify each paper into DB, or AI, or none of them, if the corresponding area has more representative keywords in its title. Afterward, we build two temporal graphs, $G_{AI} = (V, E_{AI})$ and $G_{DB} = (V, E_{DB})$, where V consists of authors who have published at least three papers at KDD, an edge $(u, v, t) \in E_{AI}$ indicates that authors u and v collaborate on an AI paper published in year t, and an edge in E_{DB} indicates similar collaborations on DB papers. Finally, we divide the whole time interval $[2000, 2023]$ into six disjoint intervals, each having 4 years length, and analyze the AI and DB communities by counting δ-temporal triangles.



(a) $\delta \in \{0, 1\}$ (b) $\delta \in \{2, 3\}$

Figure 17: δ-temporal triangle density.

• **Collaboration density trends of DB and AI communities.** As a well-known metric of measuring the subgraph cohesiveness [45, 47, 51], the triangle density of a graph is defined as the number of δ-temporal triangles over the number of vertices. Figure 17 shows the δ-temporal triangle densities for the DB and AI communities across all time intervals with varying δ values. We observe that after 2016, the δ-temporal triangle density of the AI community surpasses that of the DB community, indicating AI's rising prominence post-2016. Besides, the number of 1-temporal triangles is significantly higher than that of 0-temporal triangles, while the difference between the number of 3-temporal and 2-temporal triangles is minimal. This suggests that authors are more likely to continue collaborating over short periods of time.



(a) $\delta \in \{0, 1\}$ (b) $\delta \in \{2, 3\}$

Figure 18: δ-transtivity.

• **Transitivity trends of DB and AI communities.** Transitivity is a widely used metric for measuring graph sparsity [13]. We extend the δ-transitivity as three times the number of binary δ-temporal triangles divided by the number of binary δ-temporal wedges, where a binary δ-temporal wedge is a path of three vertices u-v-w with the timestamp gap of the two edges not exceeding δ. Figure 18 shows the δ-transitivity of the DB and AI communities in all the six time intervals with varying δ values. We observe a continuous decline in the δ-transitivity of the AI community, indicating that it has become sparser as more researchers join it.

7 RELATED WORK

In this section, we review the related works of triangle counting in static graphs and temporal graphs.

• **Triangle counting in static graphs.** As an essential problem in network analysis, triangle counting has garnered significant research attention. Most existing works have concentrated on static undirected graphs. The exact counting solution, relying on enumeration, is bounded by $O(m\kappa)$ [12]. Alternatively, counting triangles through matrix multiplication improves time complexity [2]. Recent advancements have pushed the time complexity bottleneck of matrix multiplication to $O(n^{2.371552})$ [53].

Additionally, some works focus on approximating the number of triangles either through local or global counting methods. For example, Kolountzakis and Miller [34] presented a method for approximating the local number of triangles, while several studies [11, 18, 52] focused on estimating global triangle counting. Despite this positive research progress, these methods cannot be directly applied to triangle counting in temporal graphs since they do not consider temporal information.

• **Triangle counting in temporal graphs.** Several recent works have studied triangle counting in temporal graphs. [42] defined the δ-temporal triangle and provided an algorithm of counting δ-temporal triangles. [43] further defined $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-temporal triangles and provided an algorithm based on degeneracy order to count the $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-temporal triangles. There are several papers [24, 30] that counted the triangles within a sliding time window. [8, 37] proposed efficient sampling methods for updating the number of triangles in continuously updating temporal graphs. To our best knowledge, our work is the first to focus on counting δ-triangles within arbitrary time windows in temporal graphs.

In addition, many works study the counting of motifs, which are more general than triangles [6, 40, 42]. For instance, [42] also defined δ-temporal motifs and provided a general method for counting them. Besides, some works focus on counting some special motifs, such as clique [27] and butterfly [9, 46].

8 CONCLUSION

In this paper, we study the problem of efficiently counting δ-temporal triangles in large temporal graphs. We first propose an online algorithm OTTC by counting the δ-temporal triangles sharing the vertices of each triangle in the static graph. Afterward, we develop an efficient index-based solution that maps δ-temporal triangles into 2-dimensional points and compactly organizes them in a tree structure. Besides, we study efficient algorithms for binary δ-temporal triangle counting. Experiments on both real and synthetic large temporal graphs show that OTTC is up to 70× faster than the SOTA algorithm, and our index-based algorithm is up to 10^8× faster than online algorithms. In the future, we will study how to apply our algorithms to directed graphs, and try to develop fast approximation algorithms through the sampling method.

REFERENCES

[1] [n. d.]. Efficiently Counting Triangles in Large Temporal Graphs (Technical Report). https://anonymous.4open.science/r/counting-triangles-2154.

[2] Mohammad Al Hasan and Vachik S Dave. 2018. Triangle counting in large networks: a review. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 8, 2 (2018), e1226.

[3] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. 2008. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. 16–24.

[4] Austin R Benson, David F Gleich, and Jure Leskovec. 2016. Higher-order organization of complex networks. *Science* 353, 6295 (2016), 163–166.

[5] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (sep 1975), 509–517. https://doi.org/10.1145/361002.361007

[6] Hanjo D Boekhout, Walter A Kosters, and Frank W Takes. 2019. Efficiently counting complex multilayer temporal motifs in large-scale networks. *Computational Social Networks* 6, 1 (2019), 8.

[7] Giorgos Bouritsas, Fabrizio Frasca, Stefanos Zafeiriou, and Michael M Bronstein. 2022. Improving graph neural network expressivity via subgraph isomorphism counting. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45, 1 (2022), 657–668.

[8] Luciana S Buriol, Gereon Frahling, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Christian Sohler. 2006. Counting triangles in data streams. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 253–262.

[9] Xinwei Cai, Xiangyu Ke, Kai Wang, Lu Chen, Tianming Zhang, Qing Liu, and Yunjun Gao. 2023. Efficient Temporal Butterfly Counting and Enumeration on Temporal Bipartite Graphs. *arXiv preprint arXiv:2306.00893* (2023).

[10] Bernard Chazelle. 1988. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.* 17, 3 (1988), 427–462.

[11] Xiaowei Chen and John CS Lui. 2018. Mining graphlet counts in online social networks. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 12, 4 (2018), 1–38.

[12] Norishige Chiba and Takao Nishizeki. 1985. Arboricity and subgraph listing algorithms. *SIAM Journal on computing* 14, 1 (1985), 210–223.

[13] Shumo Chu and James Cheng. 2011. Triangle listing in massive networks and its applications. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. 672–680.

[14] Paulo GS da Fonseca and Israel BF da Silva. 2017. Online construction of wavelet trees. In *16th International Symposium on Experimental Algorithms (SEA 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[15] Mark De Berg. 2000. *Computational geometry: algorithms and applications*. Springer Science & Business Media.

[16] Mark De Berg, Otfried Cheong, Marc Van Kreveld, and Mark Overmars. 2008. Orthogonal range searching: Querying a database. *Computational Geometry: Algorithms and Applications* (2008), 95–120.

[17] Shiyuan Deng, Shangqi Lu, and Yufei Tao. 2023. Space-Query Tradeoffs in Range Subgraph Counting and Listing. *arXiv preprint arXiv:2301.03390* (2023).

[18] Talya Eden, Amit Levi, Dana Ron, and C Seshadhri. 2017. Approximately counting triangles in sublinear time. *SIAM J. Comput.* 46, 5 (2017), 1603–1646.

[19] Thomas Erlebach, Michael Hoffmann, and Frank Kammer. 2021. On temporal graph exploration. *J. Comput. System Sci.* 119 (2021), 1–18.

[20] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 619–630.

[21] Yixiang Fang, Kaiqiang Yu, Reynold Cheng, Laks VS Lakshmanan, and Xuemin Lin. 2019. Efficient algorithms for densest subgraph discovery. *arXiv preprint arXiv:1906.00341* (2019).

[22] Peter M Fenwick. 1994. A new data structure for cumulative frequency tables. *Software: Practice and experience* 24, 3 (1994), 327–336.

[23] David F Gleich and C Seshadhri. 2012. Vertex neighborhoods, low conductance cuts, and good seeds for local community methods. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. 597–605.

[24] 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.

[25] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. 2003. High-order entropy-compressed text indexes. (2003).

[26] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. 2014. Chronos: a graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems*. 1–14.

[27] Anne-Sophie Himmel, Hendrik Molter, Rolf Niedermeier, and Manuel Sorge. 2016. Enumerating maximal cliques in temporal graphs. In *2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*. IEEE, 337–344.

[28] Sitao Huang, Mohamed El-Hadedy, Cong Hao, Qin Li, Vikram S Mailthody, Ketan Date, Jinjun Xiong, Deming Chen, Rakesh Nagi, and Wen-mei Hwu. 2018. Triangle counting and truss decomposition using FPGA. In *2018 IEEE high performance extreme computing conference (HPEC)*. IEEE, 1–7.

[29] 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 ACM SIGMOD international conference on Management of data*. 1311–1322.

[30] Madhav Jha, Ali Pinar, and C Seshadhri. 2015. Counting triangles in real-world graph streams: Dealing with repeated edges and time windows. In *2015 49th Asilomar Conference on Signals, Systems and Computers*. IEEE, 1507–1514.

[31] Minsoo Jung, Yongsub Lim, Sunmin Lee, and U Kang. 2019. FURL: Fixed-memory and uncertainty reducing local triangle counting for multigraph streams. *Data Mining and Knowledge Discovery* 33 (2019), 1225–1253.

[32] Arijit Khan, Nan Li, Xifeng Yan, Ziyu Guan, Supriyo Chakraborty, and Shu Tao. 2011. Neighborhood based fast graph search in large networks. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 901–912.

[33] Christine Klymko, David Gleich, and Tamara G Kolda. 2014. Using triangles to improve community detection in directed networks. *arXiv preprint arXiv:1404.5874* (2014).

[34] Mihail N Kolountzakis, Gary L Miller, Richard Peng, and Charalampos E Tsourakakis. 2012. Efficient triangle counting in large graphs via degree-based vertex partitioning. *Internet Mathematics* 8, 1-2 (2012), 161–185.

[35] Lauri Kovanen, Márton Karsai, Kimmo Kaski, János Kertész, and Jari Saramäki. 2013. Temporal motifs. *Temporal networks* (2013), 119–133.

[36] Jérôme Kunegis. 2013. KONECT – The Koblenz Network Collection. In *Proc. Int. Conf. on World Wide Web Companion*. 1343–1350.

[37] Dongjin Lee, Kijung Shin, and Christos Faloutsos. 2020. Temporal locality-aware sampling for accurate triangle counting in real graph streams. *The VLDB Journal* 29 (2020), 1501–1525.

[38] D. T. Lee and C. K. Wong. 1977. Worst-Case Analysis for Region and Partial Region Searches in Multidimensional Binary Search Trees and Balanced Quad Trees. *Acta Inf.* 9, 1 (mar 1977), 23–29. https://doi.org/10.1007/BF00263763

[39] Jure Leskovec and Rok Sosič. 2016. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. *ACM Transactions on Intelligent Systems and Technology (TIST)* 8, 1 (2016), 1.

[40] Paul Liu, Austin Benson, and Moses Charikar. 2018. A sampling framework for counting temporal motifs. *arXiv preprint arXiv:1810.00980* (2018).

[41] David W Matula and Leland L Beck. 1983. Smallest-last ordering and clustering and graph coloring algorithms. *Journal of the ACM (JACM)* 30, 3 (1983), 417–427.

[42] Ashwin Paranjape, Austin R Benson, and Jure Leskovec. 2017. Motifs in temporal networks. In *Proceedings of the tenth ACM international conference on web search and data mining*. 601–610.

[43] Noujan Pashanasangi and C Seshadhri. 2021. Faster and generalized temporal triangle counting, via degeneracy ordering. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 1319–1328.

[44] Joseph J Pfeiffer, Timothy La Fond, Sebastian Moreno, and Jennifer Neville. 2012. Fast generation of large scale social networks while incorporating transitive closures. In *2012 International Conference on Privacy, Security, Risk and Trust and 2012 International Confernece on Social Computing*. IEEE, 154–165.

[45] Raman Samusevich, Maximilien Danisch, and Mauro Sozio. 2016. Local triangle-densest subgraphs. In *2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*. IEEE, 33–40.

[46] Seyed-Vahid Sanei-Mehri, Ahmet Erdem Sariyuce, and Srikanta Tirthapura. 2018. Butterfly counting in bipartite networks. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2150–2159.

[47] Comandur Seshadhri, Aneesh Sharma, Andrew Stolman, and Ashish Goel. 2020. The impossibility of low-rank representations for triangle-rich complex networks. *Proceedings of the National Academy of Sciences* 117, 11 (2020), 5631–5637.

[48] Kijung Shin. 2017. Wrs: Waiting room sampling for accurate triangle counting in real graph streams. In *2017 IEEE International Conference on Data Mining (ICDM)*. IEEE, 1087–1092.

[49] Sara Nadiv Soffer and Alexei Vazquez. 2005. Network clustering coefficient without degree-correlation biases. *Physical Review E* 71, 5 (2005), 057101.

[50] Kanat Tangwongsan, Aduri Pavan, and Srikanta Tirthapura. 2013. Parallel triangle counting in massive streaming graphs. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*. 781–786.

[51] Charalampos Tsourakakis. 2015. The k-clique densest subgraph problem. In *Proceedings of the 24th international conference on world wide web*. 1122–1132.

[52] Charalampos E Tsourakakis, Mihail N Kolountzakis, and Gary L Miller. 2009. Approximate triangle counting. *arXiv preprint arXiv:0904.3761* (2009).

[53] Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. 2024. New bounds for matrix multiplication: from alpha to omega. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 3792–3835.

[54] Ling Zhao, Yujiao Song, Chao Zhang, Yu Liu, Pu Wang, Tao Lin, Min Deng, and Haifeng Li. 2019. T-gcn: A temporal graph convolutional network for traffic prediction. *IEEE transactions on intelligent transportation systems* 21, 9 (2019),

3848–3858.

A MORE DETAILS OF OUR KDT-INDEX-BASED INDEX SOLUTION

Before introducing the construction of KDT-Index, we preset a lemma, which states that the number of BC-points is at most twice the number of C-points of in the temporal graph.

Lemma 5. *Given a temporal graph G, the number of BC-points of G is at most twice the number of C-points of G.*

Proof. For each static triangle $\Delta_{u,v,w}$, there are only two types of BC-points: d=1 and d=−1. For the BC-point $\langle (x, y, z), d\rangle$ with d=1, it corresponds to a C-point $\langle (x, y), c\rangle$, since it comes from a δ-temporal triangle whose minimum and maximum timestamps are x and y respectively. Besides, it is impossible to have two BC-points $\langle (x_1, y_1, z_1), 1\rangle$ and $\langle (x_2, y_2, z_2), 1\rangle$ with $x_1 = x_2, y_1 = y_2$, and $z_1 \neq z_2$, since we can always eliminate the one with larger z. Thus, the number of BC-points with d=1 is bounded by the number of C-points. Meanwhile, the number of BC-points with d=−1 is less than the number of BC-points with d=1. Hence, Lemma 5 holds. □

• KDT-Index **construction.** Given a list of BC-points L, we can construct the KDT-Index level by level in a top-down manner: we first choose the dimension D and select the BC-point with medium coordinate in the dimension for the root, and then go through L to get the A, S for the root. Afterward, the unselected BC-points in L will be sent to the children of the root, and we continue to construct the KDT-index in the children nodes.

• KDT-Index**-based binary δ-temporal triangle counting.** Given a time window $[t_s, t_e]$ and a duration δ, the counting result equals the summarized S values of nodes in the KDT-Index whose A is contained by the cube $[t_s, t_e]\times[t_s, t_e]\times[0, \delta]$. We apply a recursive method starting from the root to find the counting result, as shown in Algorithm 6. Initially, we let *node* be the root of KDT-Index (line 1). Then, we use a function KDsum to sum the R values of nodes whose A is contained by the cube (line 2). If the cube A of the current node is contained by the query cube, we return the R of the node (line 3). If the intersection of the cube and A is empty, we return 0 (line 5); otherwise, we continue the recursion in the children nodes (line 6).

Algorithm 6: Sum the R values of nodes in a cube

Input: KDT-Index, a query cube $[t_s, t_e] \times [t_s, t_e] \times [0, \delta]$
Output: Total R value of nodes whose A is contained by the cube

```
1 node ← root of KDT-Index;
2 Function KDsum(node):
3     if node.A ⊆ [t_s, t_e] × [t_s, t_e] × [0, δ] then return node.S;
4     else
5         if node.A ∩ [t_s, t_e] × [t_s, t_e] × [0, δ] = ∅ then return 0;
6         else return KDsum(node.lChild) + KDsum(node.rChild);
```

Based on analysis of [38], we can conclude that Algorithm 6 completes in $O(\Delta^{\frac{2}{3}})$ time.

• KDT-Index **maintenance.** For the dynamic temporal graph, when a new BC-point is generated, we update the KDT-Index starting from the root node in a top-down manner. Specifically, we first update the cube A and sum R in the current node, and then compare the coordinates of the new BC-point and the BC-point of the current node in the selected dimension. If the new BC-point has a larger value in the selected dimension, it will be in the right child to continue the update; otherwise, it will be in the left child.