Efficient Temporal Butterfly Counting and Enumeration on Temporal Bipartite Graphs

Xinwei Cai
Zhejiang University
xwcai98@zju.edu.cn

Xiangyu Ke*
Zhejiang University
xiangyu.ke@zju.edu.cn

Kai Wang
ACEM, Shanghai Jiao Tong University
w.kai@sjtu.edu.cn

Lu Chen
Zhejiang University
luchen@zju.edu.cn

Tianming Zhang
Zhejiang University of Technology
tmzhang@zjut.edu.cn

Qing Liu
Zhejiang University
qingliucs@zju.edu.cn

Yunjun Gao
Zhejiang University
gaoyj@zju.edu.cn

ABSTRACT

Bipartite graphs characterize relationships between two different sets of entities, like actor-movie, user-item, and author-paper. The butterfly, a 4-vertices 4-edges (2,2)-biclique, is the simplest cohesive motif in a bipartite graph and is the fundamental component of higher-order substructures. Counting and enumerating the butterflies offer significant benefits across various applications, including fraud detection, graph embedding, and community search. While the corresponding motif, the triangle, in the unipartite graphs has been widely studied in both static and temporal settings, the extension of butterfly to temporal bipartite graphs remains unexplored. In this paper, we investigate the *temporal butterfly counting and enumeration* problem: count and enumerate the butterflies whose edges establish following a certain order within a given duration. Towards efficient computation, we devise a non-trivial baseline rooted in the state-of-the-art butterfly counting algorithm on static graphs, further, explore the intrinsic property of the temporal butterfly, and develop a new optimization framework with a compact data structure and effective priority strategy. The time complexity is proved to be significantly reduced without compromising on space efficiency. In addition, we generalize our algorithms to practical streaming settings and multi-core computing architectures. Our extensive experiments on 11 large-scale real-world datasets demonstrate the efficiency and scalability of our solutions.

PVLDB Reference Format:
Xinwei Cai, Xiangyu Ke, Kai Wang, Lu Chen, Tianming Zhang, Qing Liu, and Yunjun Gao. Efficient Temporal Butterfly Counting and Enumeration on Temporal Bipartite Graphs. PVLDB, 17(4): 657 - 670, 2023.
doi:10.14778/3636218.3636223

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ZJU-DAILY/TBFC.

*Xiangyu Ke is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 17, No. 4 ISSN 2150-8097.
doi:10.14778/3636218.3636223

1 INTRODUCTION

Bipartite graphs, which separate vertices into two disjoint sets and allow edges only between different sets of vertices, serve as natural data models for capturing relationships between two distinct types of entities [66], such as actor-movie, user-item, and author-paper. As motifs (i.e., small frequent subgraph patterns) are fundamental building blocks of complex graphs [45], discovering and counting motifs can reveal hidden relationships among participating entities [1, 4, 45], contributing to the characterization of complex networks [33], such as social network analysis [60], traffic speed forecasting [63], and research on spiking activity in neural networks [14]. In the context of bipartite graphs, butterfly (i.e. a (2,2)-biclique) - the simplest cohesive higher-order substructure, is the most fundamental motif, analogous to the triangle in unipartite graphs [2]. A bipartite graph cannot exhibit any community structure without butterflies as analyzed in [2]. Counting and enumerating butterflies have become essential components in various downstream network analytic tasks, e.g., bipartite clustering coefficient computation [2], k-bitruss construction [43], graph embedding [15], etc.

In real-world scenarios, networks exhibit a temporal nature, where interactions between entities can arise and cease over time. To capture such dynamics, temporal graphs are employed, where edges are associated with timestamps [19, 26]. By incorporating the temporal ordering and duration constraint (i.e., all edges have to occur within a fixed duration) [10, 35], temporal motifs offer enhanced information and greater expressiveness compared to standard motifs. Temporal ordering represents the sequence of events, while temporal duration denotes the validity period of these events. Whilst *temporal motif counting and enumeration* have been extensively studied on temporal unipartite graphs [10, 19, 27, 29, 35, 36], the temporal bipartite graphs are yet to be explored, except for fundamental reachability query [6]. Motivated by this research gap, we investigate the *temporal butterfly counting and enumeration* problem, which is to count and enumerate butterflies in different temporal ordering (i.e., 6 non-isomorphic temporal butterfly types as shown in Figure 1) while adhering to duration constraint.

Applications. We present two representative real-world applications of temporal butterflies as below.

[Recommendation]: When providing recommendations on a user-item network, the edges of the network are naturally timestamped

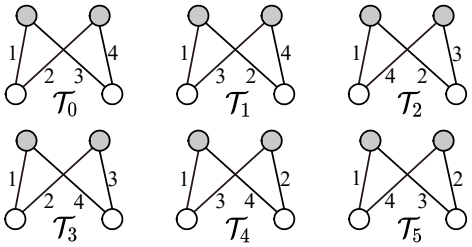


Figure 1: 6 types of non-isomorphic temporal butterfly. The vertices from the same layer have the same color (grey in U and white in L), and the number denotes the temporal order.

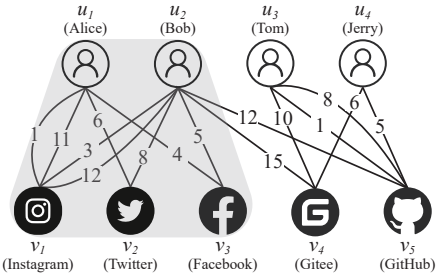


Figure 2: A user-website network.

to indicate the timing of interactions. For example, in the context of a movie recommendation system, the timestamped edges can represent instances when a customer watches a movie. Similar scenarios may include paper reading or user website exploration [12, 51, 55, 57]. While static butterflies (encompass all six types in Figure 1) can capture user/item pairs with similar behavior, butterflies with specific temporal ordering can contribute to role recognition [35]. For instance, types $\mathcal{T}_0$, $\mathcal{T}_1$, and $\mathcal{T}_2$ (as shown in Figure 1) can indicate that a user consistently follows another user in their behavior when considering the grey vertices as users. In particular, type $\mathcal{T}_0$ captures the immediate co-doing behavior, suggesting a stronger follower effect. This temporal information is valuable for modeling social influence [6]. In Figure 2, the notably higher count of type $\mathcal{T}_0$ instances between Alice and Bob suggests that Bob often accesses the same website in close succession after Alice does. This pattern strongly implies the possibility that Bob might be Alice's follower. Additionally, the duration constraint strengthens the intrinsic correlation between entities, enhancing the accuracy and effectiveness of recommendations [22, 27, 29].

[Data Monitor]: In decentralized finance, users' transaction information on each platform is publicly accessible, enabling the construction of a user-platform network [16]. While transactions are secure and transparent, user anonymity poses challenges to effective monitoring. By incorporating temporal ordering into static butterflies, we can extract more accurate relationships between anonymous users. For instance, type $\mathcal{T}_3$ represents minimal asset circulation, while types $\mathcal{T}_4$ and $\mathcal{T}_5$ indicate asset exchanges between two users (assuming grey vertices represent users and white vertices represent platforms). Introducing a stricter duration constraint can enhance the monitoring of high-frequency transactions.

Other applications include disease control in people-location network [9], fraud detection on user-page networks [24], threat hunting in process-IP network [23], etc.

Challenges. Although one can sacrifice the bipartite property and apply existing temporal motif counting and enumeration techniques to determine that of the 4-edge rectangle (then filter based on vertex type), existing techniques either fail since they only support up to 3-vertex, 3-edge temporal motifs [10, 19, 35, 36], or become extremely inefficient[1] [22, 29, 35]. Moreover, existing butterfly counting and enumeration techniques cannot be easily adapted to the temporal environment, as they struggle with handling the more complicated situations on the temporal bipartite graph (i.e., there may be multiple edges between two vertices), and require additional overhead to check whether the identified butterflies meet the specified constraints and determine the types correctly.

Our Contributions. We are the first to study the *temporal butterfly counting and enumeration* problem, which aims to quantify the number of butterflies or enumerate instances of butterflies whose edges follow a specific temporal ordering within a specified time duration (for precise details, please refer to § 2 for the formal definition).

Our baseline solutions build upon the state-of-the-art *butterfly counting* algorithm [54] through refining the vertex-priority assignment, verifying the duration constraint, and casting all possible temporal ordering (§ 3). We carefully observe and summarize the rules governing the relationships between two wedges, which are the core components of a butterfly. Based on these observations, optimization techniques are designed accordingly (§ 4). In particular, we devise the compact data structure (i.e., wedge set) and extend the priority concept from vertex to wedge level, which captures the temporal ordering from both direction and coverage perspectives (§ 4.1). The searching space is largely pruned and the redundant permutation is avoided. These optimization techniques enable efficient counting and enumeration algorithms with minor modifications to the technical framework, resulting in substantial gains in efficiency (§ 4.2, 4.3). In addition, we incorporate advanced engineering efforts to handle extreme cases and further improve counting efficiency. Theoretically, the time complexity is significantly reduced while the space complexity remains unchanged (§ 4.4). Empirical evaluation over real-world datasets validates that our optimized algorithm is up to 3773.1 times faster than the baseline.

To support the practical graph streams [31, 37, 48, 53], we extend our counting algorithm to facilitate dynamic updates over streams and further propose a non-trivial parallel algorithm to handle batch updates (§ 5). The parallel algorithm focuses on resolving the counting conflicts and providing problem-specific simplifications.

Finally, we demonstrate the efficiency and scalability of our proposed algorithms via extensive experimental evaluations on 11 large-scale temporal bipartite networks.

Our principal contributions are summarised as follows.

- We are the first to define the concept of temporal butterflies and conduct a comprehensive study on the problem of *temporal butterfly counting and enumeration* (§ 2).
- We design a non-trivial baseline based on the state-of-the-art *butterfly counting* algorithm (§ 3) and develop an optimization framework with compact data structure and effective priority strategy. Theoretically, the time complexity is significantly reduced without sacrificing the space (§ 4).

[1]They cannot avoid permuting all possible combinations of four orderly edges within a duration constraint and check if it is a butterfly, which takes $O(|E|^4)$ time.

- We adapt our algorithm to temporal bipartite graph stream setting and further propose a parallel algorithm to improve the throughput of streaming data by batch counting (§ 5).
- We conduct extensive experiments on various real-world temporal bipartite graphs to demonstrate the efficiency and scalability of our proposed algorithms (§ 6).

2 PRELIMINARIES

An undirected[2] *temporal bipartite graph* $G = (V = (U, L), E, T)$ is defined over two disjoint sets of vertices U and L, i.e., $U \cap L = \emptyset, U \cup L = V$, which represent two classes of real-world entities, known as upper and lower layer vertex sets, respectively. The connections only exist across different classes, i.e., edge set $E \subseteq U \times L \times T$, where T is a collection of timestamps. Each *temporal edge* $e = (u, v, t) \in E$ represents an interaction between u and v at the time t. Notice that multiple temporal edges may exist between the same pair of vertices with different timestamps. $E(u)$ denotes the set of temporal edges adjacent to vertex u. We extend the concept of *butterfly* and its basic component, *wedge*, from simple static graphs as follows:

Definition 1 (Temporal Wedge). *In a temporal bipartite graph G, a temporal wedge $\angle(u, v, w, t_s, t_a)$ is a 2-hop path consisting of (u, v, t_s) and (v, w, t_a).*

The inherent nature of a bipartite graph ensures that u and w belong to one layer while v is from the other layer. We designate u the start-vertex, v the middle-vertex, and w the end-vertex. In the following discussions, a wedge is *forward* if $t_s < t_a$, and is *backward* if $t_s > t_a$.

Definition 2 (Butterfly [52]). *Given vertices $u, w \in U$ and $v, x \in L$, the subgraph induced by these four vertices in G is a butterfly if it is a 2×2 bi-clique; that is, u and w are all connected to v and x, respectively, by edges.*

A butterfly can be decomposed into a pair of wedges with the same start-vertex, the same end-vertex, and different middle-vertices. Therefore, most existing *butterfly counting* algorithms [41, 52, 54] focus on efficient enumeration and permutation of wedges.

Definition 3 (Temporal Butterfly). *Given a duration threshold δ, a temporal butterfly is a sequence of 4 temporal edges in chronological order $\langle e_1, e_2, e_3, e_4 \rangle$, s.t., (1) $e_1.t < e_2.t < e_3.t < e_4.t$, (2) $e_4.t - e_1.t \leq \delta$, and (3) the induced static graph is a butterfly.*

The duration constraint enforces that all edges must occur within a fixed duration δ to ensure the existence of a butterfly, i.e., the earliest edge has not yet expired when the latest edge appears. Regarding the graph depicted in Figure 2, upon setting δ to 15, we can identify two instances of type $\mathcal{T}_2$ involving the vertices u_2, u_3, v_4, v_5. However, if we impose a more stringent duration constraint of 10, only one butterfly will remain. The static butterfly fails to capture such kind of valuable property. The possible temporal permutations[3] (i.e., 6 non-isomorphic temporal butterflies as shown in Figure 1) of these four edges make the induced butterfly even more expressive in real-world applications, as illustrated in § 1.

[2]Bipartite graphs are generally undirected in existing studies [2, 40, 43, 54, 66] and real-world datasets (§ 6).

[3]Following [19, 35, 36], we assume that the four timestamps on a temporal butterfly are distinct, which can be implemented by applying simple tie-breaking rule, e.g., based on the unique indexes of the starting/ending vertices as in [54].

Algorithm 1: TBC

Input: the temporal bipartite graph $G = (V = (U, L), E, T)$; the threshold δ
Output: the counts $\{C[i]\}_{i=0}^{5}$

```
1  {C[i]}_{i=0}^{5} := 0
2  foreach E(u) : u ∈ V do
3      sort all e ∈ E(u) according to P_V(e.v)
4  foreach u ∈ V do
5      initialize hashmap H to store wedges
6      foreach (u, v, t) ∈ E(u) : P_V(u) > P_V(v) do
7          foreach e'(v, w, t) ∈ E(v) : P_V(u) > P_V(w) do
8              H[w].append((u, v, w, t, t'))
9      foreach vertex w ∈ H : |H[w]| > 1 do
10         foreach pair ∠_i, ∠_j ∈ H[w] : j < i do
11             if IsTB(∠_i, ∠_j) then
12                 C[Type(∠_i, ∠_j)] += 1
13 return {C[i]}_{i=0}^{5}
```

Problem Statement. Given a temporal bipartite graph G and a threshold δ, our *temporal butterfly counting* problem is to count the number $\{C[i]\}_{i=0}^{5}$ of each of six types of temporal butterflies $\mathcal{T}_0, \mathcal{T}_1, \cdots, \mathcal{T}_5$, and our *temporal butterfly enumeration* problem is to find all these temporal butterfly instances $\{B[i]\}_{i=0}^{5}$.

Solution Overview. We first devise our baseline solutions by extending the leading static butterfly counting algorithm [54] (TBC/TBE in § 3). Then we optimize the algorithms by designing a compact data structure and generalizing priority to wedge level for smart pruning and processing acceleration (TBC$^+$/TBE$^+$/TBC^{++} in § 4). In addition, we generalize our counting solution to meet the practical steaming demand (STBC/STBC$^+$ in § 5).

3 BASELINE SOLUTION

In this section, we devise our baseline solution, based on the state-of-the-art *butterfly counting* algorithm BFC-VP [54]. Hereafter, we omit temporal in the temporal edge/wedge/butterfly when the context is clear.

BFC-VP sorts the vertices based on their proposed vertex priority and efficiently enumerates all but less redundant wedges that can form a butterfly. Following the same intuition, we assign a unique vertex priority for any vertex u based on $|E(u)|$. Note that the priority is no longer neighbor-based as there may exist multiple temporal edges between two vertices. The correctness and efficiency proofs follow [54], while details can be found in full version [5].

Definition 4 (Vertex Priority). *For any vertex u in a temporal bipartite graph G, the priority $P_V(u)$ is an integer in $[1, |V|]$. For any two vertices u and w in G, $P_V(u) > P_V(w)$ if:*

- $|E(u)| > |E(w)|$, *or*
- $|E(u)| = |E(w)|$ *and* $id(u) > id(w)$

where $id(u)$ is the unique vertex ID of u.

TBC follows a sequential process of "enumerate-filter-match", as outlined in Algorithm 1. After the initialization and priority assignment (line 1-3), TBC constructs the wedges from each start-vertex u to all lower-priority vertices (line 4-8). Subsequently, for each possible wedge combination (line 9-10), TBC filters out invalid

instances (line 11) and determines their type according to Figure 1 (line 12). In particular, Type() returns the butterfly type and IsTB() (Is Temporal Butterfly) checks for the following constraints: **(1)** Middle-vertices of the two wedges should be different. **(2)** There exists a temporal ordering for the four timestamps of the two wedges, i.e., no two timestamps are equal. **(3)** All timestamps must fall within a δ duration, i.e., the difference between the maximum and minimum timestamps cannot exceed δ. Additionally, we can obtain TBE by simply modifying Algorithm 1 from counting to storing/outputting all instances (line 12).

Complexity Analysis.

• The time complexity of TBC is $O(\sum_{u\in V}|W(u)|^2)$, where $|W(u)| = \sum_{(u,v,t)\in E(u):|E(v)|\leq|E(u)|}|E(v)|$.

Proof. The first phase of TBC enumerates all possible wedge instances in $O(\sum_{(u,v)\in E}\min\{|E(u)|,|E(v)|\})$, as reported by [54] for BFC-VP. In the second phase of butterfly construction, TBC takes quadratic time overhead to check the various temporal conditions. We denote all the processed wedges with the start-vertex u by $W(u)$, and $|W(u)| = \sum_{(u,v,t)\in E(u):|E(v)|\leq|E(u)|}|E(v)|$. Therefore, the overall time complexity of TBC is $O(\sum_{u\in V}|W(u)|^2)$. □

• The space complexity of TBC is $O(|E|+\max_{u\in V}\{|W(u)|\})$, where $|W(u)| = \sum_{(u,v,t)\in E(u):|E(v)|\leq|E(u)|}|E(v)|$.

Proof. As demonstrated in the above proof for time complexity analysis, in addition to the input graph, TBC only stores the wedges of one particular start-vertex (and discards them after the current iteration). The space complexity simply follows. □

• The time/space complexity of TBE is the same as TBC.

Proof. Assuming that all butterfly instances are directly output to disk without occupying memory, there is no essential difference between TBC and TBE. □

4 A NEW FRAMEWORK WITH WEDGE SET

In this section, we first discuss the intuitions behind our optimizations (specifically for counting), inspired by the observations of butterfly types. Then, we present our detailed counting algorithm designs and smart implementations. With minor modifications, the algorithm can also be adapted for enumeration purposes. Finally, we apply advanced data structures to enhance our counting efficiency and effectively handle extreme cases.

4.1 Optimization Overview

Figure 3 illustrates all 24 potential temporal orderings between two arbitrary temporal wedges, denoted as $\angle_i$ and $\angle_j$. These wedges share the same start-vertex and end-vertex in U but differ in their middle-vertex in L. Upon analyzing the temporal relations between the time arcs induced by $\angle_i$ and $\angle_j$, we make the following observations: **(1)** From the *temporal coverage* perspective, there are three possible categories: *non-overlap*, *intersecting*, and *covering*. **(2)** From the *temporal direction* perspective, they can either follow the same (*forward* or *backward*) direction or deviate from each other. The columns in Figure 3 correspond to the coverage relationship {*non-overlap*, *intersecting*, and *covering*} (from left to right), while the rows indicate whether the two wedges follow the same temporal direction or not. Each temporal ordering (c_{xy} for that in the x^{th} row and the y^{th} column of Figure 3) in every distinct box can be transformed into one another by exchanging the wedge indices

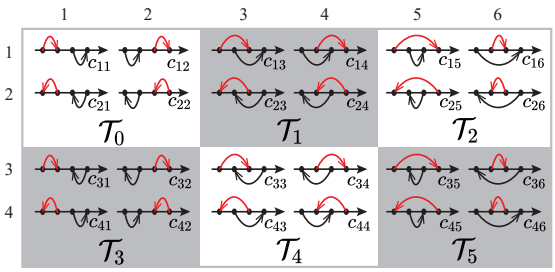


Figure 3: All possible temporal orderings of two temporal wedges $\angle_i$ and $\angle_j$, represented by red and black arcs above/under a time axis. The arcs point from t_s to t_a.

and/or reversing the start- and end-vertex, corresponding to one temporal butterfly type. In summary, the temporal butterfly types are built on 3 coverage patterns and 2 direction patterns. Therefore, we devise a compact data structure (i.e., wedge set) to distinguish the temporal directions and generalize the priority concept to the wedge level for smart pruning. In addition, we introduce the type conversion rule for counting the butterflies from either vertex layer.

Wedge Set. In each iteration, we enumerate wedges starting from a specific start-vertex. In the presence of multiple edges, it is possible to have multiple wedges with the same middle-vertex. To optimize the process, we propose organizing the wedges into different sets based on their middle-vertices, as defined in Definition 5. By doing so, during butterfly construction, we only consider wedges from different sets, effectively reducing time overhead. Furthermore, this approach reduces space requirements during counting, as we only need to store two timestamps for each wedge.

Additionally, we partition the wedge set into two disjoint subsets, namely A and D, for forward and backward wedges respectively. To accommodate backward wedges, we swap the t_s and t_a values before inserting them into subset D. This simple operation significantly reduces the 24 possible orderings depicted in Figure 3 to just 6 cases, as shown in the first row where each column becomes a merged case. However, despite the merging, determining the butterfly type remains straightforward: $A\times A$ or $D\times D$ indicates the same temporal direction ($\mathcal{T}_0$, $\mathcal{T}_1$, $\mathcal{T}_2$), while $A\times D$ or $D\times A$ represents different temporal directions ($\mathcal{T}_3$, $\mathcal{T}_4$, $\mathcal{T}_5$).

Definition 5 (Wedge Set). *For all the wedges having the same start-vertex and the same end-vertex, those wedges with the same middle-vertex v are stored in the wedge set $S_v = (A, D)$. For any such wedge, if its $t_s < t_a$, (t_s, t_a) is stored in subset A, otherwise (t_a, t_s) is stored in subset D. $A \cap D = \emptyset$.*

Wedge Priority. To avoid redundant permutations and facilitate further optimization, we prioritize the wedges when considering two arbitrary temporal wedges $\angle_i$ and $\angle_j$ that share the same start-vertex and end-vertex but differ in their middle-vertex. By constructing the butterflies in a wedge-priority-increasing manner, we effectively eliminate the need to handle cases c_{12}, c_{14}, and c_{16}. Instead, we can focus on determining the type of a temporal butterfly by examining the way the wedge sets join and evaluating three coverage patterns. In cases where two wedges have the same wedge priority, their order can be arbitrarily chosen without affecting the correctness of subsequent algorithms.

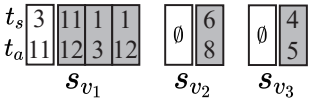


Figure 4: The wedge sets construct from Figure 2 while u_2 is the start-vertex and u_1 is the end-vertex.

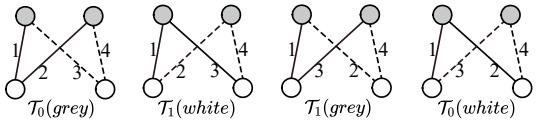


Figure 5: Example for wedge type conversion. Two wedges are marked by solid and dashed lines, respectively.

Definition 6 (Wedge Priority). *The wedge priority $P_W(\cdot)$ is a total order among all wedges. For any two wedges $\angle_i$ and $\angle_j$, $P_W(\angle_i) < P_W(\angle_j)$ if:*

- *$\angle_i.t_s > \angle_j.t_s$, or*
- *$\angle_i.t_s = \angle_j.t_s$ and $\angle_i.t_a < \angle_j.t_a$.*

Example 1. *Figure 4 shows the 3 wedge sets of Figure 2, where u_2 is the start-vertex and u_1 is the end-vertex (since $P_V(u_2) > P_V(u_1)$). Each small rectangle with two numbers represents a wedge with two timestamps t_s, t_a, where the white ones are in A and the gray ones are in D, and each subset is already sorted according to wedge priority.*

Type Conversion. In Figure 3, we only discuss the cases while the start-vertex is in U, but both layers can serve as the starting side in practice[4]. Depending on the perspective of different layers, the same butterfly can be decomposed into different sets of wedges, resulting in distinct coverage and direction patterns. For example, if the grey vertices are in U, then the butterfly belongs to type $\mathcal{T}_0$; otherwise, it falls into type $\mathcal{T}_1$. This relationship holds vice versa for the two butterflies on the right-hand side of Figure 5. Similar patterns can be observed between $\mathcal{T}_2$ and $\mathcal{T}_3$, as well as $\mathcal{T}_4$ and $\mathcal{T}_5$. Therefore, we can handle wedge combinations in a unified framework, and finally decide the butterfly type based on the conversion rule.

4.2 Algorithm Design

The introduction of our optimization algorithm, TBC$^+$, will be structured in a gradual manner, progressing from an overarching perspective (specifically, Algorithm 2) down to the finer intricacies. TBC$^+$ shares the same initialization and vertex priority assignment as TBC (lines 1-3). For each start-vertex u, TBC$^+$ initializes a nested hashmap H to store the wedges induced by each pair of end-vertex w and middle-vertex v (lines 4-8). This hashmap contains only two timestamps for each wedge and is organized based on the temporal directions (lines 9-12). Furthermore, non-empty sets with different middle-vertices are re-indexed, and the Combine() function is used to compute the number of temporal butterflies (lines 13-16), which will be explained in detail in the following paragraphs. To optimize the algorithm, we employ simple pruning by excluding wedges with $|t_s - t_a| > \delta \vee t_s = t_a$. This pruning step performs a partial check of the temporal duration constraint in advance, filtering out illegal wedges. It ensures that each wedge $\angle$ satisfies $\angle.t_s < \angle.t_a \leq \angle.t_s + \delta$.

[4]In real-world scenarios, our interest goes beyond examining the co-behavior of users. We also seek to understand the relationships between different items.

Algorithm 2: TBC$^+$

Input: the temporal bipartite graph $G = (V = (U, L), E, T)$; the threshold δ
Output: the counts $\{C[i]\}_{i=0}^{5}$

1 $\{C[i]\}_{i=0}^{5} := 0$
2 **foreach** $E(u) : u \in V$ **do**
3 sort all $e \in E(u)$ according to $P_V(e.v)$
4 **foreach** $u \in V$ **do**
5 initialize hashmap H to store sets
6 **foreach** $(u, v, t) \in E(u) : P_V(u) > P_V(v)$ **do**
7 **foreach** $(v, w, t') \in E(v) : P_V(u) > P_V(w)$ **do**
8 **if** $t \neq t' \wedge |t' - t| \leq \delta$ **then**
9 **if** $t < t'$ **then**
10 $H[w][v].A$.append$(\angle(t, t'))$
11 **else if** $t > t'$ **then**
  // t, t' is swaped when append
12 $H[w][v].D$.append$(\angle(t', t))$
13 **foreach** *vertex* $w \in H : |H[w]| > 1$ **do**
14 reindex sets in $H[w]$ to $S_0, S_1, \cdots, S_{|H(w)|}$
15 sort each subset in $H[w]$ according to $P_W(\angle)$
16 Combine$(u, \delta, H[w], \{C[i]\}_{i=0}^{5})$
17 **return** $\{C[i]\}_{i=0}^{5}$

Lemma 1. *Given a temporal threshold δ, a temporal wedge $\angle(t_s, t_a)$ with $|t_s - t_a| > \delta \vee t_s = t_a$ cannot be a part of any temporal butterfly.*

This proof, along with other straightforward proofs, is immediate and is relocated to the full version [5].

We present the implementation details of the Combine() function in Algorithm 3 as below.

Combine Wedge Sets. Taking inspiration from the renowned Mergesort method [17], we adopt a recursive merging approach to combine the wedge sets in the wedge priority-increasing order. This method ensures that wedge permutations are efficient and balanced, and that only wedge pairs with different middle-vertices are checked. As illustrated in Algorithm 3, given the hashmap $H[w]$ and the threshold δ, Recur() recursively merges two sets in a bottom-up manner until one set is left (line 1-7). Smart simultaneous implementations about set merging (line 8-28) will be elaborated on in later paragraphs. Merge() applies wedge priority as the merge rules and follows the original Mergesort method (line 29).

Order of Counting. To ensure that the processing order of wedges in the merge process always follows the wedge priority, for each subset, a pointer ptr tracks the next wedge to process and a hashmap HP maintains the visited wedges (line 9-11). Note that HP only builds an array to store t_a for each t_s. Subsequently, TBC$^+$ identifies the maximum t_s among all the unprocessed wedges, denoted as $maxn$ (lines 13-15). After filtering out illegal wedges, TBC$^+$ query each unprocessed wedge with the maximum t_s and the previous wedges in the HP (line 19-25). Function Insert() (line 26-28) updates the newly visited wedges in HP. Note that in actual implementation, Merge() (line 29) can conduct in sync with Insert() (line 26-28), as the Insert() does not change A and D. Specifically, TBC$^+$ always processes wedges with the same t_s together to avoid redundancy.

Counting Procedure. To expedite the combination process between a wedge $\angle_i$ and multiple wedges $\angle_j$ satisfying $\angle_j.t_s > \angle_i.t_s$,

Algorithm 3: Combine() for Algorithm 2

Input: the start-vertex u; the threshold δ; the hashmap $H[w]$ that including wedge sets $S_0, S_1, ..., S_{|H[w]|}$; the counts $\{C[i]\}_{i=0}^{5}$

```
1  Recur(u, δ, 0, |H[w]|, H[w], C[·])          // C[·] is {C[i]}_{i=0}^5
2  Function Recur(u, δ, p, q, H[w], C[·])
3      if p + 1 ≥ q then return
4      mid = (p + q)/2
5      Recur(u, δ, p, mid, H[w], C[·])
6      Recur(u, δ, mid, q, H[w], C[·])
7      SetCross(u, δ, S_p, S_mid, C[·])
8  Function SetCross(u, δ, S_i(A_i, D_i), S_j(A_j, D_j), C[·])
9      foreach b ∈ {A_i, D_i, A_j, D_j} do
10         ptr_b := 0
11         initialize HP_b
12     while ∃ptr_b < |b| : b ∈ {A_i, D_i, A_j, D_j} do
13         maxn := −inf
14         foreach b ∈ {A_i, D_i, A_j, D_j} : ptr_b < |b| do
15             maxn := max(maxn, s[ptr_b].t_s)
16         foreach b ∈ {A_i, D_i, A_j, D_j} do
17             Delete(maxn + δ, HP_b)
18             pre_ptr_b = ptr_b
19         foreach b ∈ {A_i, D_i, A_j, D_j} do
20             while ptr_b < |b| ∧ b[ptr_b].t_s = maxn do
21                 if A_i do Query(u, b[ptr_b], HP_{A_j}, HP_{D_j}, C[·])
22                 if D_i do Query(u, b[ptr_b], HP_{D_j}, HP_{A_j}, C[·])
23                 if A_j do Query(u, b[ptr_b], HP_{A_i}, HP_{D_i}, C[·])
24                 if D_j do Query(u, b[ptr_b], HP_{D_i}, HP_{A_i}, C[·])
25                 ptr_b += 1
26         foreach b ∈ {A_i, D_i, A_j, D_j} do
27             for k := pre_ptr_b to ptr_b do
28                 Insert(s[k], HP_b)
29     S_i := (Merge(A_i, A_j), Merge(D_i, D_j))
```

we utilize the direct derivation $\angle.t_s < \angle.t_a \leq \angle.t_s + \delta$ from Lemma 1. The set of wedges $\angle_j$ is maintained using a hashmap HP, which can be as simple as an array to store t_a values for each t_s encountered. Wedges that violate the duration constraint, as stated in Lemma 2, i.e., $\angle_j$ with $\angle_j.t_a > \angle_i.t_s + \delta$, are eliminated. Lemma 3 guarantees that a deleted wedge from HP will not be re-inserted. By appending wedges into HP according to wedge priority, all the t_a values in the same array in HP are in ascending order. This enables us to utilize binary search to expedite the counting process. In particular, $\angle_i.t_a < \angle_j.t_s < \angle_j.t_a$, $\angle_j.t_s < \angle_i.t_a < \angle_j.t_a$ and $\angle_j.t_s < \angle_j.t_a < \angle_i.t_a$ corresponding to case c_{11}, c_{13}, c_{15} in Figure 3, respectively. Moreover, according to Lemma 4, all the t_a in $HP[t_s] : t_s > \angle_i.t_a$ has satisfied the case c_{11} without binary search.

Lemma 2. *Given the threshold δ and two wedges $\angle_i$, $\angle_j$ both satisfy $\angle.t_s < \angle.t_a \leq \angle.t_s + \delta$ and $\angle_i.t_s < \angle_j.t_s$, $\angle_i$, $\angle_j$ can form a temporal butterfly only if the condition $\angle_j.t_a \leq \angle_i.t_s + \delta$ is satified.*

Lemma 3. *Given the threshold δ and three wedges $\angle_i, \angle_j, \angle_k$ all satisfy $\angle.t_s < \angle.t_a \leq \angle.t_s + \delta$ and $\angle_i.t_s < \angle_j.t_s < \angle_k.t_s$, if $\angle_k$ and $\angle_j$ can't form a temporal butterfly, then neither can $\angle_k$ and $\angle_i$.*

Lemma 4. *If two forward wedges $\angle_i, \angle_j$, $\angle_i.t_s < \angle_j.t_s$ satisfy $\angle_i.t_a < \angle_j.t_s$, we have $\angle_i.t_s < \angle_i.t_a < \angle_j.t_s < \angle_j.t_a$.*

Table 1: Hashmap HP's operations.

API	Description
HP.erase(t)	erase the $HP[t]$
$\|HP[t]\|$	return the size of $HP[t]$
$HP[t]$.empty()	return *true* if $HP[t]$ is empty, return *false* otherwise
$HP[t]$.append(x)	push x into the back of $HP[t]$
$HP[t]$.pop($> x$)	pop all elements $> x$
$HP[t]$.count($\odot x$)	return the number of elements $\odot x$, $\odot$ can be $<, >, \leq, \geq$

Algorithm 4: Core functions for Algorithm 3

Input: the number *bound*; the hashmap HP, HP_i, HP_j; the start-vertex u; the wedge $\angle$; the counts $\{C[i]\}_{i=0}^{5}$

```
1  Function Delete(bound, HP)
2      foreach t ∈ HP do
3          HP[t].pop(> bound)
4          if HP[t].empty() then
5              HP.erase(t)
6  Function Query(u, ∠, HP_i, HP_j, {C[i]}_{i=0}^5)
7      if u ∈ U then l := 0 else l := 1
8      foreach t ∈ HP_i do
9          if t > ∠.t_a then
10             C[0 ⊕ l] += |HP_i[t]|     // ⊕ is xor operation
11         else if t < ∠.t_a then
12             C[1 ⊕ l] += HP_i[t].count(> ∠.t_a)
13             C[2 ⊕ l] += HP_i[t].count(< ∠.t_a)
14     foreach t ∈ HP_j do
15         if t > ∠.t_a then
16             C[3 ⊕ l] += |HP_j[t]|
17         else if t_s < ∠.t_a then
18             C[4 ⊕ l] += HP_j[t].count(> ∠.t_a)
19             C[5 ⊕ l] += HP_j[t].count(< ∠.t_a)
20 Function Insert(∠, HP)
21     HP[∠.t_s].append(∠.t_a)
```

Table 1 presents the operations supported by a hashmap HP that maintains an ordered array for each key. Note that $HP[t]$.pop($> x$) run in the $O(n)$ time where n denotes the number of elements to be popped out, and $HP[t]$.count($\odot x$) runs in $O(log|HP[t]|)$ time. All other operations consume $O(1)$ time.

Algorithm 4 illustrates the core functions in SetCross(). Given the *bound* (i.e., *maxn*+δ as above), Delete() deletes all elements greater than *bound* in the target hashmap and erases empty arrays (line 1-5). Query() conducts a binary search to count all types of butterflies induced by a start-vertex and one of its wedges. The variable l denotes the layer of u, and the type of butterflies can be converted through a simple xor operation $\oplus$ following our conversion rule (line 6-19). Insert() simply append $\angle.t_a$ into the back of $HP[\angle.t_s]$, and this operation won't break the order in HP (line 20-21).

Example 2. *We present a small example to illustrate HP in Figure 6. Suppose $\delta = 10$, a wedge $\angle_i(1, 7)$ and multiple wedges $\{\angle_j\}$:*

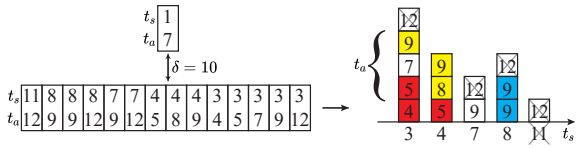


Figure 6: Example for HP.

$\angle_j.t_s > \angle_i.t_s$ are sorted according to wedge priority. After inserting all wedges $\angle_j$ into HP, numbers under the axis denote t_s, and multiple squares on the axis denote the corresponding t_a. Then, squares with grey X represent wedges that need to be deleted, blue, yellow, and red these three kinds of squares respectively represent case c_{11}, c_{13}, c_{15} after pairing with $\angle_i$.

Complexity Analysis.

• The time complexity of TBC$^+$ is $O(\sum_{u\in V}|W(u)|(log(|E(u)|)+\alpha\, log(\frac{|W(u)|}{\alpha})))$, where α is a coefficient between 1 and $|W(u)|$.

Proof. The main differences between TBC and TBC$^+$ are in the counting phase. Recur() involves each wedge asymptotic participate $log(|E(u)|)$ times in the SetCross(), where $E(u)$ is the number of wedge sets. Within SetCross(), wedges are traversed once, questioned once, inserted into HP once, and deleted from HP at most once. The time complexity of Insert() and Delete() are $O(1)$ but the Query() is $O(\alpha log(\frac{n}{\alpha}))$, where n denotes the number of wedges in HP, and α is a coefficient between 1 and n depends on how many binary searches we run in Query(). Due to the size of HP under consideration, it's essential to evaluate the complexity of Query() from a global perspective. In other words, for each wedge, distinct wedges with differing middle-vertices are considered precisely once in a Query(). Thus, the total time complexity is proved. □

• The space complexity of TBC$^+$ is $O(|E| + \max_{u\in V}\{|W(u)|\})$.

Proof. The space complexity of the wedge enumerating process in TBC$^+$ is essentially unchanged. While in the counting process, additional temporary auxiliary space is required for tracking merging sets and maintaining the HP. However, this never exceeds the original number of wedges. Thus, the space complexity remains $O(|E| + \max_{u\in V}\{|W(u)|\})$. □

4.3 Supporting Enumeration Algorithm

In this section, we discuss the modifications made to our framework to enable butterfly enumeration. We introduce the inclusion of middle-vertex information in the wedge sets and the HP hashmap, which was previously omitted during the counting process. This addition allows us to combine two wedges and obtain butterfly instances by determining the start- and end-vertices in advance. The overall procedure of TBE$^+$ closely resembles that of TBC$^+$, with a variation in the Query() function, as depicted in Algorithm 5. In HP, wedges with the same t_s are ordered based on their t_a values. This ordering enables us to utilize range traversal to easily find specific types of temporal butterfly instances, similar to the binary search employed during the counting process. Specifically, in TBE$^+$, we iterate through $B[1\oplus l]$ from the beginning to the end, stopping as soon as the constraint is no longer satisfied (line 8-9). Similarly, we iterate through $B[2\oplus l]$ from the end to the beginning, breaking the loop once the constraint is violated (line 10-11). To enumerate all instances, the combination process is still required in TBE$^+$, which means the time and space complexity remains the same as TBE. However, due to efficient pruning strategies, we can eliminate the need for additional checks during the wedge combination, as discussed in § 3. This allows us to directly determine the butterfly type, resulting in a significant improvement in efficiency, as demonstrated in § 6.

Algorithm 5: Query() for TBE$^+$

Input: the hashmap HP_i, HP_j; the start-vertex u; the wedge $\angle$; the butterfly instances $\{B[i]\}_{i=0}^{5}$

```
1  Function Query(u, ∠, HP_i, HP_j, {B[i]}_{i=0}^5)
2      if u ∈ U then l := 0 else l := 1
3      foreach t ∈ HP_i do
4          if t > ∠.t_a then
5              foreach ∠_j ∈ HP_i[t] do
6                  B[0 ⊕ l].append((∠, ∠_j))
7          else if t < ∠.t_a then
8              foreach ∠_j ∈ HP_i[t] : ∠_j.t_a > ∠.t_a do
9                  B[1 ⊕ l].append((∠, ∠_j))
10             foreach ∠_j ∈ HP_i[t] : ∠_j.t_a < ∠.t_a do
11                 B[2 ⊕ l].append((∠, ∠_j))
12     foreach t ∈ HP_j do
           // handle B[3/4/5 ⊕ l] similar to line 4-11
```

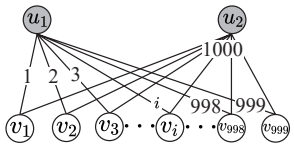


Figure 7: A temporal bipartite graph containing two high-degrees vertices u_1 and u_2.

4.4 Handling Extreme Cases

Notably, the core bottleneck of TBC$^+$ lies in the unstable efficiency of Query(). Figure 7 presents an extreme case while u_1 is the start-vertex and all the wedges have different t_s, leading to a quadratic time in wedge combinations (i.e., $\alpha \approx |W(u)|$ in the time complexity of TBC$^+$). This is a common situation in real-world datasets: a small number of vertices come with a very high degree (and subsequently many wedges with different t_s) [7, 34].

We further equip our counting solution with two red-black trees [8, 13] to resolve the issue. Specifically, TA is a red-black tree to maintain wedges with the key t_a, and TS is a twin red-black tree of TA that only maintains t_s with the key t_s. These two trees are synchronized and contain the same elements, but they are organized based on different keys. This design allows us to perform efficient two-way operations compared to storing all the elements in a single hashmap. All operations of TA, TS (detailed in [5]) run in $O(log(n))$ time, where n is the number of elements in a tree.

The core differences lie in speeding up the Query() and Delete() functions with two red-black trees TA, TS (in TBC^{++}) instead of using hashmap HP. We omit minor modifications on SetCross() due to space limitations. Algorithm 6 demonstrates the improvements made to Algorithm 4. When deleting wedges, following Lemma 2, TBC^{++} checks the last elements in TA and erases it from both TA

Algorithm 6: Core functions for TBC^{++}

Input: the number *bound*; the red-black tree TS, TS_i, TS_j with key t_s; the red-black tree TA, TA_i, TA_j with key t_a; the start-vertex u; the wedge $\angle$; the counts $\{C[i]\}_{i=0}^{5}$

1 **Function** Delete(*bound*, (TS, TA))
2 **while** $\angle := TA.back() : \angle.t_a > bound$ **do**
3 TA.erase($\angle$)
4 TS.erase($\angle.t_s$)
5 **Function** Query($u, \angle, (TS_i, TA_i), (TS_j, TA_j), \{C[i]\}_{i=0}^{5}$)
6 **if** $u \in U$ **then** $l := 0$ **else** $l := 1$
7 $C[0 \oplus l]$ += TS_i.count($> \angle.t_a$)
8 $C[1 \oplus l]$ += TA_i.count($> \angle.t_a$) − TS_i.count($\geq \angle.t_a$)
9 $C[2 \oplus l]$ += TA_i.count($< \angle.t_a$)
10 $C[4 \oplus l]$ += TS_j.count($> \angle.t_a$)
11 $C[5 \oplus l]$ += TA_j.count($> \angle.t_a$) − TS_j.count($\geq \angle.t_a$)
12 $C[6 \oplus l]$ += TA_j.count($< \angle.t_a$)
13 **Function** Insert($\angle$, (TS, TA))
14 TS.insert($\angle$)
15 TA.insert($\angle.t_s$)

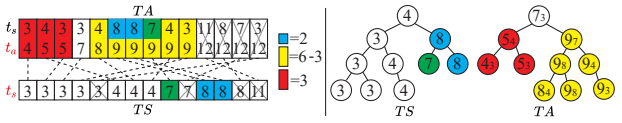


Figure 8: Example for TA, TS.

and TS if the condition is violated (line 1-4). Based on Lemma 4, TBC^{++} conducts count() on TS which corresponds to the case c_{11} (line 7, 10). Similarly, according to Lemma 6 and Lemma 7, TBC^{++} conduct count() on both TA, TS and subtract which corresponds the case c_{13} (line 8, 11). According to Lemma 5, TBC^{++} does count() on TA which corresponds the case c_{15} (line 9, 12).

Lemma 5. *If two forward wedges $\angle_i, \angle_j$, $\angle_i.t_s < \angle_j.t_s$ satisfy $\angle_j.t_a < \angle_i.t_a$, we have $\angle_i.t_s < \angle_j.t_s < \angle_j.t_a < \angle_i.t_a$.*

Lemma 6. *If two forward wedges $\angle_i, \angle_j$, $\angle_i.t_s < \angle_j.t_s$ satisfy $\angle_i.t_a < \angle_j.t_a$, we have $\angle_i.t_s < \angle_i.t_a \leq \angle_j.t_s < \angle_j.t_a$ or $\angle_i.t_s < \angle_j.t_s < \angle_i.t_a < \angle_j.t_a$.*

Lemma 7. *If two forward wedges $\angle_i, \angle_j$, $\angle_i.t_s < \angle_j.t_s$ satisfy $\angle_i.t_a \leq \angle_j.t_s$, we have $\angle_i.t_s < \angle_i.t_a \leq \angle_j.t_s < \angle_j.t_a$.*

Example 3. *In Figure 8, we show how to use TA, TS to handle the query in Figure 6. Suppose two ordered lists represent the red-black trees, keeping the same key order as in the corresponding tree. The elements in the two trees align with each other via dotted lines. Then rectangles with grey X represent wedges that need to be deleted, and corresponding squares with grey X will be deleted as well. Blue squares, pure yellow rectangles, and red rectangles are these three kinds of polygons representing case c_{11}, c_{13}, and c_{15}, respectively.*

Complexity Analysis.

• The time complexity of TBC^{++} is $O(\sum_{u \in V} |W(u)|(log(|E(u)|) + log(|W(u)|)))$.

Proof. Note that, Delete(), Insert() and Query() are all run in $O(log(n))$, where n is the number of wedges in TA/TS. Thus, the time complexity of TBC^{++} is proven. □

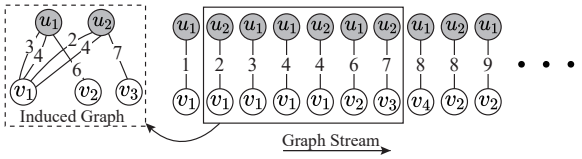


Figure 9: A temporal bipartite graph stream.

• The space complexity of TBC^{++} is $O(|E| + \max_{u \in V}\{|W(u)|\})$.

Proof. Although TS and TA require constant times the memory cost of HP, the space complexity remains unchanged. □

5 ALGORITHMS ON GRAPH STREAMS

Temporal bipartite graphs often encounter frequent rapid updates, involving the insertion or deletion of vertices and edges. These updates are commonly represented as graph streams, where edges are presented in their timestamp order [30]. To efficiently count temporal butterflies, it is preferable to update the counts incrementally instead of recomputing them from scratch. Figure 9 illustrates a temporal bipartite graph stream, with the corresponding induced graph of selected edges displayed on the left-hand side. While it is possible to extend many of the mentioned algorithms to graph streams, we will focus our discussion on the most effective counting algorithm (i.e., TBC^{++}), for the sake of brevity.

The intuitive idea is to count the number of temporal butterflies that contain the edge waiting update and modify the counts accordingly. In this context, we propose the STBC algorithm (detailed in [5]), a non-trivial extension from TBC^{++}. The major changes are as follows: **(1)** While enumerating wedges, vertex priority becomes irrelevant as all butterflies induced by the current edge must be considered, but Lemma 1 still applies as a timestamp t is known. If we store $E(u)$ in a queue and process it in chronological order, we can use binary search to compress the traversal range into $[t\text{-}\delta, t\text{+}\delta]$. **(2)** Since a middle-vertex v is determined, to avoid unnecessary merging operations, we only need to maintain two wedge sets within each $H[w]$ depending on whether the middle-vertex is v or not.

We also propose a parallel version of the algorithm, called STBC^{+}, which is presented in Algorithm 7. Regarding edge deletion, in accordance with Lemma 8, STBC^{+} redefines the traversal range to $(t, t\text{+}\delta]$, thereby preventing count conflicts. Noted that t represents the minimum timestamp in a temporal butterfly, and the temporal duration can be easily satisfied while enumerating wedges. Consequently, STBC^{+} no longer needs red-black trees TS, TA to maintain wedges dynamically, but only two simple arrays VS, VA. During combination, STBC^{+} only sorts these VS, VA in need and performs Query() in a similar manner as in TBC^{++}. To prevent read-write conflicts between threads, the deletion of edges is executed after the counting process is completed. Similarly, when inserting edges, the traversal range is redefined as $[t\text{-}\delta, t)$, and all edges should be inserted into the graph beforehand.

Lemma 8. *Counting each temporal butterfly on the edge with the minimum/maximum timestamp can prevent count conflicts when tallying within a batch of edges with the minimum/maximum time.*

Proof. A temporal butterfly only has one minimum/maximum timestamp, and thus can avoid being counted multiple times. □

Algorithm 7: STBC$^+$ (delete multiple edges)

Input: the temporal bipartite graph $G = (V = (U, L), E, T)$; the threshold δ; the edges waiting for delete $\{e_1, e_2, \cdots, e_i\}$, the counts $\{C[i]\}_{i=0}^{5}$

```
foreach e(u,v,t) ∈ {e_1, e_2, ..., e_i} in parallel do
    initialize hashmap H for each w to store sets
    foreach (u,x,t') ∈ E(u) : t < t' ≤ t + δ do
        if x ≠ v ∧ t' ≠ t then
            foreach (x,w,t'') ∈ E(x) : t < t'' ≤ t + δ do
                if w ≠ u then
                    if t' < t'' then
                        H[w].A.VS.append(t')
                        H[w].A.VA.append(t'')
                    else if t' > t'' then
                        H[w].D.VS.append(t'')
                        H[w].D.VA.append(t')
    foreach (v,w,t') ∈ E(v) : t < t' ≤ t + δ do
        if w ≠ u then
            if H[w] is unsorted then
                sort A.VS, A.VA, D.VS, D.VA in H[w]
            {C[i]}_{i=0}^{5} -= Query(u, (t,t'), H[w].A, H[w].D)
foreach e ∈ {e_1, e_2, ..., e_i} do
    delete e from G
```

Complexity Analysis.

• Given the edge $e(u, v, t)$, the time complexity of updating a single edge in STBC and STBC$^+$ are both $O(|E^2(u)|log(|E^2(u)|))$, where $|E^2(u)| = \sum_{(u,v,t)\in E(u)} |E(v)|$.

Proof. The edges enumerated by STBC and STBC$^+$ is denoted by $E^2(u)$, where $|E^2(u)| = \sum_{(u,v,t)\in E(u)} |E(v)|$, thus the time complexity are both $O(|E^2(u)|log(|E^2(u)|))$ similar to the complexity analysis of TBC^{++}. Despite STBC and STBC$^+$ having the same time complexity, STBC$^+$ has a smaller constant than STBC since STBC$^+$ uses two simple arrays to replace the red-black tree. □

• Given the edge $e(u, v, t)$, the time complexity of updating a single edge in STBC and STBC$^+$ are both $O(|E| + |E^2(u)|$, where $|E^2(u)| = \sum_{(u,v,t)\in E(u)} |E(v)|$.

Proof. As noted in the proof for time complexity, the edges enumerated by STBC and STBC$^+$ are denoted by $E^2(u)$, where $|E^2(u)| = \sum_{(u,v,t)\in E(u)} |E(v)|$. □

6 EXPERIMENTAL EVALUATION

In this section, we present the empirical evaluation of our solutions using 11 large-scale real-world datasets.

Experiment Settings. All our algorithms[5] were implemented in C++ and executed on a Ubuntu machine with an Intel(R) Core(TM) i9-10900K CPU @ 3.70GHz and 128G memory. We set a maximum running time limit of 100,000 seconds and terminate the execution if the limit is exceeded. Notably, the reported time costs do not include preprocessing time, such as the graph loading time. The space cost is measured by monitoring the maximum VmRSS (Virtual Memory Resident Set Size) of the process.

[5]Available at https://github.com/ZJU-DAILY/TBFC

Table 2: The summary of datasets.

Dataset	$\|E\|$	$\|V\|$		Time Span (days)
		$\|U\|$	$\|L\|$	
Wikiquote (WQ)	776,458	961	640,482	4625.66
Wikinews (WN)	907,499	2,200	35,979	4857.34
StackOverflow (SO)	1,301,942	545,196	96,680	1153.00
CiteULike (CU)	2,411,819	153,277	731,769	1203.10
Bibsonomy (BS)	2,555,080	204,673	767,447	7665.43
Twitter (TW)	4,664,605	175,214	530,418	1155.34
Amazon (AM)	5,838,041	2,146,057	1,230,915	3650.00
Edit-ru (ER)	8,349,235	7,816	1,266,349	4976.35
Epinions (EP)	13,668,320	120,492	755,760	504.96
Last.fm (LF)	19,150,868	992	174,077	3149.77
Wiktionary (WT)	44,788,448	66,140	5,826,113	5941.22

Algorithms. The competitors include: **(1)** Temporal Butterfly Counting algorithms: TBC, TBC$^+$, and TBC^{++}. **(2)** Temporal Butterfly Enumeration algorithms: TBE and TBE$^+$. **(3)** Streaming Temporal Butterfly Counting algorithms: STBC and STBC$^+$. **(4)** two temporal motif isomorphism algorithms [22, 29] and a temporal motif counting algorithm [35]. In the evaluation of enumeration algorithms, we do not perform any additional actions, such as outputting butterfly instances to external memory, when they are found. This is because directly storing instances in external memory would introduce additional time costs due to I/O operations while storing them in RAM would result in extra space costs. To ensure fair comparison experiments, we focus solely on the enumeration process. In addition to our proposed algorithms for temporal butterflies, we also attempted some state-of-the-art general temporal motif algorithms [22, 29, 35] in accordance with the same experimental setup. However, it is important to note that no algorithm specifically designed for temporal butterflies exists. Even on datasets considered "easy to handle", such as WQ, SO, and CU, our algorithm completed within 10 seconds (Figure 10), whereas the general algorithms failed to meet the time limit due to the need to permute all possible combinations of four edges in the worst case. Consequently, we have excluded this comparison from our evaluation.

Datasets. The dataset statistics are presented in Table 2, where "Time Span" indicates the time difference between the maximum and minimum timestamps. In our study of the temporal bipartite graph stream, we assume that edges arrive in chronological order. For the purpose of evaluation, we adopt the widely used Sliding Window Model [18] for *streaming temporal butterfly counting*. This involves counting butterflies within a window of size $|window|$ while sliding with a stride of size $|stride|$ at each step. Both $|window|$ and $|stride|$ are measured in terms of the number of edges. Additional dataset sources and more detailed statistics can be found at KONECT[6].

6.1 Evaluation on Temporal Bipartite Graphs

Overall Performance. The efficiency of our baseline algorithms TBC and TBE, as well as our three optimization versions TBC$^+$, TBE$^+$, and TBC^{++}, is compared on various datasets as shown in Figure 10, with a default δ value of 40 days.

[6]http://konect.cc/

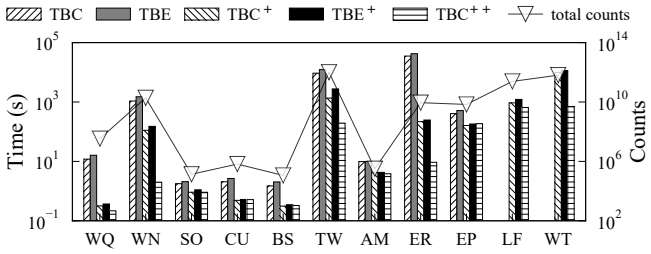


Figure 10: Time and total counts on varying datasets.

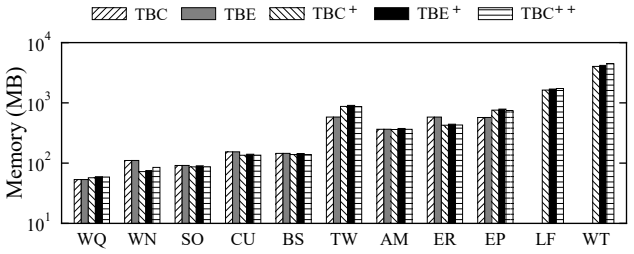


Figure 11: Memory on varying datasets.

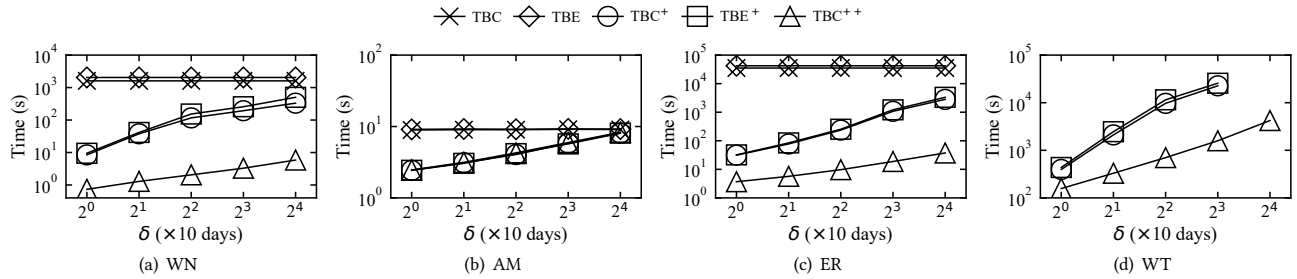


Figure 12: Time on varying δ.

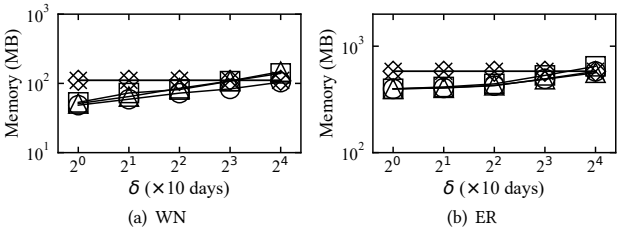


Figure 13: Memory on varying δ.

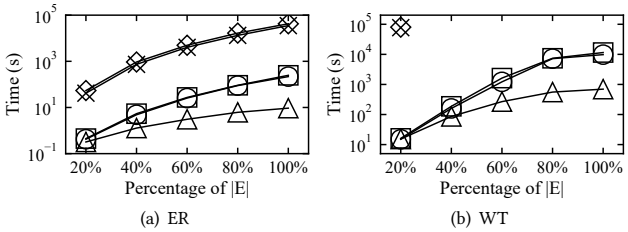


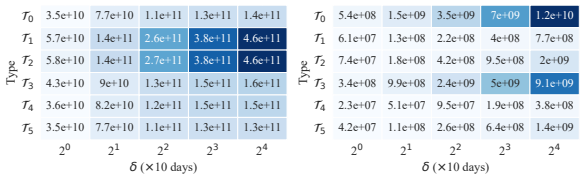
Figure 14: Time on varying scale of $|E|$.

As expected, TBC is the slowest counting algorithm and even exceeds the time limit on the LF and WT datasets. TBC$^+$ demonstrates a speedup ranging from 1.9× to 161.9× compared to TBC. The performance of TBC^{++} is the most favorable, being comparable to TBC$^+$ on certain datasets (e.g., SO, EP) and significantly outperforming it on others (e.g., WN, WT). For instance, on the WN dataset, TBC^{++} achieves a speedup of up to 61.3×. Notably, TBC$^+$ and TBC^{++} perform similarly mostly on datasets that are "easy to handle". This is mainly due to the small size of the candidate wedge set in these datasets. Although TBE follows the same flow as TBC, it is slightly slower due to the additional time required for instance construction. Similarly, TBE$^+$ performs similarly to TBC$^+$ despite utilizing range traversal instead of binary search. Despite having the same theoretical complexity as TBE, the notable improvement of TBE$^+$ over TBE provides strong evidence for the effectiveness of our proposed framework. The experimental results are consistent with our time complexity analysis, further validating our approach. Figure 10 provides an illustration of the total counts of all six types of butterflies on different datasets, with the efficiency of the algorithms showing a positive correlation with the counts.

The memory cost is presented in Figure 11. Clearly, memory consumption increases with the graph size. The memory consumption of the 5 algorithms is nearly identical, which aligns with our theoretical analysis. In some datasets (e.g., WN, ER), the optimization algorithms even exhibit lower memory overhead than the baseline algorithm, indicating the effectiveness of the pruning strategy during wedge enumeration. However, in certain datasets (e.g., TW, EP), the optimization algorithms incur slightly higher memory overhead due to the presence of auxiliary data structures. Nevertheless, the additional memory overhead of our optimization algorithms remains small compared to the significant efficiency improvements they provide. Furthermore, even on the largest ten-million-scale dataset WT, our algorithms require only 4GB.

Effects of the Duration Constraint. The duration constraint δ is the only parameter in our problem. Larger δ allows more temporal butterflies, resulting in a greater number of permutations.

As shown in Figure 12, among the algorithms tested, TBC and TBE perform the worst, while TBC^{++} shows the best performance. TBC$^+$ and TBE$^+$ fall in between. As δ increases, the time cost of TBC and TBE remains nearly constant since they don't consider δ during the wedge enumeration process and explore every possible combination. On these "easy to handle" datasets (e.g., AM), the performance of optimization algorithms is identical. However, on other datasets, the time cost of TBC$^+$ grows faster than TBC^{++}, which is reasonable because the wedge set grows as δ gets bigger (and subsequently more arrays in HP), and TBC$^+$ will be affected greatly - it even runs out of time on WT dataset when $\delta = 160$ days. The time cost gap between TBE$^+$ and TBC$^+$ widens as δ increases since the efficiency disparity between range traversal and the binary search becomes more noticeable with larger wedge sets.



(a) TW (b) EP

Figure 15: Counts on varying δ.

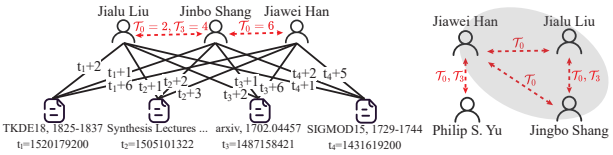


Figure 16: Case study on DBLP.

The memory cost over varying δ is presented in Figure 13. The baseline algorithms have a constant memory cost since they lack specific strategies for δ. For small δ, the optimization algorithms employ a pruning strategy during the wedge enumeration phase, resulting in a significant reduction in storage cost. However, as δ increases, the impact of the pruning strategy diminishes as fewer wedges can be filtered, and the auxiliary data structures' cost grows with the expanding wedge set. Consequently, the memory cost of the optimization algorithms increases continuously, possibly surpassing that of the baseline algorithms (see Figure 13(a)). Nonetheless, the growth rate is much lower than that of δ, and the overall memory overhead remains similar to that of the baseline algorithms.

Figure 15 displays the counts of each type of temporal butterflies with varying δ, where darker grids represent higher counts. As δ increases, the counts also tend to rise. The rate of increase varies across different datasets and types of temporal butterflies, making it unpredictable. Furthermore, in line with previous studies [32, 35, 50, 62], the distribution of temporal butterflies' counts shows minimal variation as the threshold changes. For instance, on the EP dataset, type $\mathcal{T}_0$ consistently accounts for approximately half the total, while type $\mathcal{T}_3$ always constitutes 30% of the total counts.

Distribution of Different Temporal Butterfly Types. Table 3 presents the count distribution, highlighting prevalent types. Clear distinctions are observed across various datasets, but commonalities also exist (datasets with the same cell color). In datasets like WQ, ER, and WT, where edges represent user-page edits, butterfly types $\mathcal{T}_1$ and $\mathcal{T}_2$ appear frequently, accounting for at least 39% of the total count. This indicates the moderate follower effect (less significant $\mathcal{T}_0$) as page editing usually requires time. CU and BS datasets, where edges denote tag assignments in CiteULike and BibSonomy, show higher percentages of types $\mathcal{T}_0$, $\mathcal{T}_2$, and $\mathcal{T}_3$. In the EP dataset (Epinions' user-product rating network), types $\mathcal{T}_0$ and $\mathcal{T}_3$ constitute almost 85% of the total count, with $\mathcal{T}_0$ alone making up over half. SO and AM datasets, representing marks between users and items (e.g., posts and products), exhibit a relatively balanced and prominent distribution of types $\mathcal{T}_0$, $\mathcal{T}_1$, $\mathcal{T}_2$, and $\mathcal{T}_3$. Notably, types $\mathcal{T}_4$ and $\mathcal{T}_5$ usually appear less frequently. In a nutshell, these observations suggest that focusing on some specific butterfly types, such as $\mathcal{T}_0$ for the strong follower effect, could lead to enhanced results while minimizing unnecessary effort.

Table 3: The distribution of counts while $\delta = 40$ days.

Dataset	Entities	Percentage of Total Counts					
		$\mathcal{T}_0$	$\mathcal{T}_1$	$\mathcal{T}_2$	$\mathcal{T}_3$	$\mathcal{T}_4$	$\mathcal{T}_5$
WQ	user-page	18.4%	**22.6%**	**29.5%**	15.2%	6.9%	7.5%
ER	user-page	17.1%	**34.1%**	**24.0%**	12.2%	7.2%	5.4%
WT	user-page	15.8%	**19.8%**	**19.7%**	16.6%	14.3%	13.8%
TW	user-tag	11.1%	**26.2%**	**26.3%**	13.1%	12.2%	11.0%
LF	user-band	15.1%	**21.6%**	**21.8%**	16.9%	13.1%	11.6%
CU	tag-publication	**20.6%**	15.1%	**19.7%**	**20.6%**	11.3%	12.7%
BS	tag-publication	**21.0%**	13.0%	**19.4%**	**22.1%**	10.9%	13.6%
SO	user-post	**19.3%**	**20.5%**	**19.2%**	**21.8%**	10.0%	9.2%
AM	user-product	**23.1%**	**19.6%**	**19.2%**	**20.7%**	9.1%	8.4%
WN	user-page	**30.1%**	12.2%	12.6%	**19.8%**	**20.2%**	5.1%
EP	user-product	**51.1%**	3.2%	6.1%	**34.4%**	1.4%	3.8%

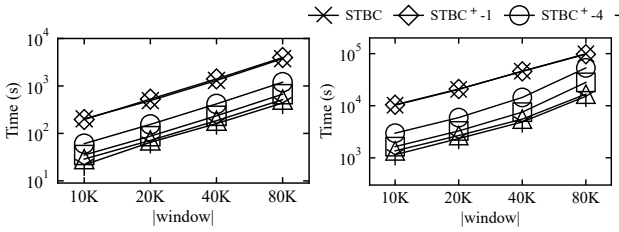
Scalability.The scalability of the algorithms is depicted in Figure 14, illustrating the running time of all competitors across different graph sizes. To evaluate scalability, we randomly select a portion of edges (i.e., {20%, 40%, 60%, 80%}) from the initial datasets, apply our method to the induced graph, and average the running time over 10 iterations. As anticipated, the time overhead for all algorithms increases with the percentage of edges, albeit at varying rates. Notably, TBC$^+$, TBE$^+$, and TBC^{++} exhibit excellent scalability, with computation costs increasing linearly relative to the percentage of edges. Among them, TBC^{++} demonstrates the best performance. On the WT dataset, our baseline algorithms only achieve completeness when the percentage is set to 20%.

Case Study. We collect an author-paper dataset from DBLP[7] using the search key "mining", comprising 50,536 papers and 78,459 distinct authors. To timestamp the edges, we use the paper publication time with an additional offset determined by author order as a tie-breaking rule. Upon analysis, we uncovered an interesting trend: the closer the collaboration between authors, the more tense butterflies we observed. Most notably, the majority of butterflies in our dataset fell into $\mathcal{T}_0$ and $\mathcal{T}_3$, mainly due to the substantial time gap between publications and the assigned offset. We extract a representative part of the data, as shown in Figure 16. Here, we find that the number of butterflies between Jialu Liu and Jinbo Shang is evenly split between $\mathcal{T}_0$ and $\mathcal{T}_3$, suggesting a balanced collaboration between them. In contrast, when examining Jiawei Han in relation to the first two authors, all butterflies were $\mathcal{T}_0$. This pattern may indicate a guiding relationship within these collaborations. The left part of Figure 16 presents details about their collaborative papers. Further investigation reveals that Jialu Liu and Jinbo Shang are graduate students closely collaborating, while Jiawei Han serves as their primary supervisor. Similarly, we can find that Jiawei Han and Philip S. Yu are scholars who work closely together, as indicated by the even distribution between $\mathcal{T}_0$ and $\mathcal{T}_3$.

6.2 Evaluation on Graph Streams

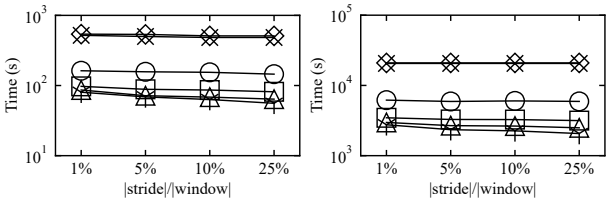
Varying Window Sizes. Figure 17 showcases the efficiency of STBC and STBC$^+$ for varying $|window|$ sizes, with a fixed $|stride|$ set to 5% of $|window|$. Additionally, we evaluate the performance

[7] https://dblp.org/



(a) LF (b) WT

Figure 17: Time on varying $|window|$.



(a) LF (b) WT

Figure 18: Time on varying $|stride|/|window|$.

of STBC$^+$ with different thread sizes, denoted as STBC$^+$-4 for a 4-thread parallel algorithm, for example. As the $|window|$ size increases, the time cost of all algorithms also increases due to the larger candidate wedge set in the induced graph.

Comparing STBC$^+$-1 and STBC, they exhibit similar efficiency since they have the same time complexity. However, STBC$^+$ with multi-threading shows significantly improved speed, with STBC$^+$-32 being up to 12.7 times faster than STBC on the WT dataset. It's worth noting that on the LF dataset, STBC$^+$-1 is slower than STBC when $|window|$ is greater than or equal to 20K. This is because STBC$^+$ needs to insert all edges before counting, resulting in an actual graph size of $|window|$ + $|stride|$ during querying, which slows it down compared to STBC. However, this drawback is tolerable because $|stride|$ is always much smaller than $|window|$, and the benefits of multi-threading far outweigh the cost.

Varying Stride Sizes. The evaluations in Figure 18 analyze the impact of varying $|stride|$ on algorithm performance, with a fixed window size of 20K. For STBC, stability against different stride values is observed due to its sequential edge updates and consistent graph size during querying. Initially, the time cost of STBC$^+$ decreases as larger strides facilitate load balancing among threads. However, this improvement diminishes as load balancing approaches its limit. Notably, on the WT dataset, STBC$^+$ experiences a slowdown when the stride percentage is $\geq$ 5%, indicating a slight impact on algorithm efficiency, consistent with findings from evaluations on varying window sizes. In summary, when ample computational resources are available, STBC$^+$ with multi-threading is the preferred choice.

7 RELATED WORK

Butterfly on Bipartite Graphs. Significant research efforts have been dedicated to the study of *butterfly counting and enumeration*, which is the most fundamental sub-structure in bipartite graphs. Wang *et al.* [52] first propose the *butterfly counting* problem and counting through enumerating wedges from a randomly selected layer. Sanei-Mehri *et al.* [41] further develop a strategy for choosing the layer while Wang *et al.* [54] achieve state-of-the-art efficiency by employing vertex priority. Additionally, various techniques have been explored in *butterfly counting*, including parallel processing [41, 47], external memory optimization [54], sampling [20, 41, 46], GPU [59], and batch update [56]. Recent advancements have extended the *butterfly counting* problem to bipartite graph streams [42, 46] and uncertain bipartite graphs [64, 65]. Yang *et al.* [61] propose a competitive search-based method for counting and enumerating the (p, q)-bicliques, with the butterfly serving as a special case where p, q=2. It is noteworthy that although the majority of research on butterflies has predominantly concentrated on counting, most of these methods can be readily expanded to facilitate enumeration with minimal modifications ([41, 54, 64]).

Temporal Motif on Temporal Unipartite Graphs. The problem of *temporal motif counting and enumeration* holds particular significance as the butterfly motif represents a distinct type of motif within this context. Building upon the concept of *motif counting* [39, 58], *temporal motif counting* has been extensively studied recently [3, 11, 21, 26, 49, 55], but their definitions vary. Kovanen *et al.* [19] introduce the concept of $\triangle T$-adjacency, which pertains to two temporal edges sharing a vertex and having a timestamp difference of at most $\triangle T$. Additionally, they take into account the temporal ordering aspect. Redmond *et al.* [38] study the δ-temporal motif counting without temporal ordering. The most relevant work to ours, Paranjape *et al.* [35] define δ-temporal motif where edges in the motif are within δ duration and the temporal ordering is considered as well. Pashanasangi *et al.* [36] introduce different thresholds for the time difference between each pair of adjacent edges in a temporal triangle. While numerous studies have focused on the 3-vertex temporal motif counting [10, 19, 35, 36], Boekhout *et al.* [3] delve into the 4-vertex temporal motifs, but specifically omitted the discussion of temporal rectangle. There are 3 non-isomorphic temporal rectangles and require a specially designed solution. Fortunately, our research encompasses this problem as a subset, and our techniques readily handle it. Furthermore, there are numerous approximation algorithms available for solving counting problems [25, 44]. When it comes to enumeration problems, isomorphism-based algorithms are the most commonly used [22, 28, 29], but they lack optimizations tailored to specific motifs, resulting in low efficiency.

8 CONCLUSION

In this paper, we investigate the *temporal butterfly counting and enumeration* problem. We formally define the problem and propose a solution based on the state-of-the-art *butterfly counting* algorithm. We further devise three optimization algorithms, two for counting and one for enumeration. Within a unified framework, these algorithms harness a combination of techniques, resulting in a significant reduction in overall time complexity without compromising on space efficiency. Additionally, we extend our algorithms to address the practical scenario of graph streams and further propose a parallel algorithm. Finally, extensive experiments validate the efficiency and scalability of our proposed algorithms.

ACKNOWLEDGMENTS

This work was supported in part by the NSFC under Grants No. (62025206, U23A20296, 62302444, 62302294, and 62302451) and NSF of Zhejiang under Grant No. LQ22F020018.

REFERENCES

[1] Nesreen K Ahmed, Jennifer Neville, Ryan A Rossi, Nick G Duffield, and Theodore L Willke. 2017. Graphlet decomposition: Framework, algorithms, and applications. *KAIS* 50, 3 (2017), 689–722.

[2] Sinan G Aksoy, Tamara G Kolda, and Ali Pinar. 2017. Measuring and modeling bipartite graphs with community structure. *Journal of Complex Networks* 5, 4 (2017), 581–603.

[3] 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), 1–34.

[4] Marco Bressan, Flavio Chierichetti, Ravi Kumar, Stefano Leucci, and Alessandro Panconesi. 2017. Counting graphlets: Space vs time. In *WSDM*. 557–566.

[5] 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).

[6] Xiaoshuang Chen, Kai Wang, Xuemin Lin, Wenjie Zhang, Lu Qin, and Ying Zhang. 2021. Efficiently answering reachability and path queries on temporal bipartite graphs. *Proceedings of the VLDB Endowment* 14, 10 (2021), 1845–1858.

[7] Aaron Clauset, Cosma Rohilla Shalizi, and Mark EJ Newman. 2009. Power-law distributions in empirical data. *SIAM review* 51, 4 (2009), 661–703.

[8] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms*. MIT press.

[9] Stephen Eubank, Hasan Guclu, VS Anil Kumar, Madhav V Marathe, Aravind Srinivasan, Zoltan Toroczkai, and Nan Wang. 2004. Modelling disease outbreaks in realistic urban social networks. *Nature* 429, 6988 (2004), 180–184.

[10] Zhongqiang Gao, Chuanqi Cheng, Yanwei Yu, Lei Cao, Chao Huang, and Junyu Dong. 2022. Scalable Motif Counting for Large-scale Temporal Graphs. In *ICDE*. 2656–2668.

[11] Saket Gurukar, Sayan Ranu, and Balaraman Ravindran. 2015. Commit: A scalable approach to mining communication motifs from dynamic networks. In *SIGMOD*. 475–489.

[12] Xiangnan He, Ming Gao, Min-Yen Kan, and Dingxian Wang. 2016. Birank: Towards ranking on bipartite graphs. *TKDE* 29, 1 (2016), 57–71.

[13] Ralf Hinze et al. 1999. Constructing red-black trees. In *WAAAPL*, Vol. 99. 89–99.

[14] Yu Hu, James Trousdale, Krešimir Josi'c, and Eric Shea-Brown. 2013. Motif statistics and spike correlations in neuronal networks. *Journal of Statistical Mechanics: Theory and Experiment* 2013, 03 (2013), P03012.

[15] Junjie Huang, Huawei Shen, Qi Cao, Shuchang Tao, and Xueqi Cheng. 2021. Signed Bipartite Graph Neural Networks. In *CIKM*. 740–749.

[16] Johannes Rude Jensen, Victor von Wachter, and Omri Ross. 2021. An introduction to decentralized finance (defi). *Complex Systems Informatics and Modeling Quarterly* 26 (2021), 46–54.

[17] Jyrki Katajainen, Tomi Pasanen, and Jukka Teuhola. 1996. Practical In-Place Mergesort. *Nord. J. Comput.* 3, 1 (1996), 27–40.

[18] Bogyeong Kim, Kyoseung Koo, Undraa Enkhbat, and Bongki Moon. 2022. DenForest: Enabling Fast Deletion in Incremental Density-Based Clustering over Sliding Windows. In *SIGMOD*. 296–309.

[19] Lauri Kovanen, Márton Karsai, Kimmo Kaski, János Kertész, and Jari Saramäki. 2011. Temporal motifs in time-dependent networks. *Journal of Statistical Mechanics: Theory and Experiment* 2011, 11 (2011), P11005.

[20] Rundong Li, Pinghui Wang, Peng Jia, Xiangliang Zhang, Junzhou Zhao, Jing Tao, Ye Yuan, and Xiaohong Guan. 2021. Approximately counting butterflies in large bipartite graph streams. *TKDE* 34, 12 (2021), 5621–5635.

[21] Yuchen Li, Zhengzhi Lou, Yu Shi, and Jiawei Han. 2018. Temporal motifs in heterogeneous information networks. In *MLG Workshop@ KDD*.

[22] Youhuan Li, Lei Zou, M Tamer Özsu, and Dongyan Zhao. 2019. Time constrained continuous subgraph search over streaming graphs. In *ICDE*. 1082–1093.

[23] Zhenyuan Li, Qi Alfred Chen, Runqing Yang, Yan Chen, and Wei Ruan. 2021. Threat detection and investigation with system-level provenance graphs: a survey. *Computers & Security* 106 (2021), 102282.

[24] Boge Liu, Long Yuan, Xuemin Lin, Lu Qin, Wenjie Zhang, and Jingren Zhou. 2019. Efficient (α, β)-core computation: An index-based approach. In *WWW*. 1130–1141.

[25] Paul Liu, Austin R Benson, and Moses Charikar. 2019. Sampling methods for counting temporal motifs. In *WSDM*. 294–302.

[26] Penghang Liu, Valerio Guarrasi, and A Erdem Sariyuce. 2021. Temporal network motifs: Models, limitations, evaluation. *TKDE* 35, 1 (2021), 945–957.

[27] Giorgio Locicero, Giovanni Micale, Alfredo Pulvirenti, and Alfredo Ferro. 2020. TemporalRI: A Subgraph Isomorphism Algorithm for Temporal Networks. In *Complex Networks*, Vol. 944. 675–687.

[28] Giorgio Locicero, Giovanni Micale, Alfredo Pulvirenti, and Alfredo Ferro. 2021. TemporalRI: a subgraph isomorphism algorithm for temporal networks. In *Proceedings of the Ninth International Conference on Complex Networks and Their Applications COMPLEX NETWORKS 2020*. 675–687.

[29] Patrick Mackey, Katherine Porterfield, Erin Fitzhenry, Sutanay Choudhury, and George Chin. 2018. A chronological edge-driven approach to temporal subgraph isomorphism. In *IEEE international conference on Big Data*. 3972–3979.

[30] Andrew McGregor. 2014. Graph stream algorithms: a survey. *ACM SIGMOD Record* 43, 1 (2014), 9–20.

[31] Youshan Miao, Wentao Han, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Enhong Chen, and Wenguang Chen. 2015. Immortalgraph: A system for storage and analysis of temporal graphs. *TOS* 11, 3 (2015), 1–34.

[32] Ron Milo, Shalev Itzkovitz, Nadav Kashtan, Reuven Levitt, Shai Shen-Orr, Inbal Ayzenshtat, Michal Sheffer, and Uri Alon. 2004. Superfamilies of evolved and designed networks. *Science* 303, 5663 (2004), 1538–1542.

[33] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. 2002. Network motifs: simple building blocks of complex networks. *Science* 298, 5594 (2002), 824–827.

[34] J-P Onnela, Jari Saramäki, Jorkki Hyvönen, György Szabó, David Lazer, Kimmo Kaski, János Kertész, and A-L Barabási. 2007. Structure and tie strengths in mobile communication networks. *Proceedings of the national academy of sciences* 104, 18 (2007), 7332–7336.

[35] Ashwin Paranjape, Austin R Benson, and Jure Leskovec. 2017. Motifs in temporal networks. In *WSDM*. 601–610.

[36] Noujan Pashanasangi and C Seshadhri. 2021. Faster and generalized temporal triangle counting, via degeneracy ordering. In *SIGKDD*. 1319–1328.

[37] Fabiola SF Pereira, Sandra de Amo, and João Gama. 2016. Evolving centralities in temporal graphs: a twitter network analysis. In *MDM*, Vol. 2. 43–48.

[38] Ursula Redmond and Pádraig Cunningham. 2013. Temporal subgraph isomorphism. In *ASONAM*. 1451–1452.

[39] Pedro Ribeiro and Fernando Silva. 2014. Discovering colored network motifs. In *Complex networks V*. 107–118.

[40] Seyed-Vahid Sanei-Mehri, Ahmet Erdem Sariyüce, and Srikanta Tirthapura. 2018. Butterfly Counting in Bipartite Networks. In *SIGKDD*. 2150–2159.

[41] Seyed-Vahid Sanei-Mehri, Ahmet Erdem Sariyuce, and Srikanta Tirthapura. 2018. Butterfly counting in bipartite networks. In *SIGKDD*. 2150–2159.

[42] Seyed-Vahid Sanei-Mehri, Yu Zhang, Ahmet Erdem Sariyüce, and Srikanta Tirthapura. 2019. FLEET: butterfly estimation from a bipartite graph stream. In *CIKM*. 1201–1210.

[43] Ahmet Erdem Sarıyüce and Ali Pinar. 2018. Peeling bipartite networks for dense subgraph discovery. In *WSDM*. 504–512.

[44] Ilie Sarpe and Fabio Vandin. 2021. OdeN: simultaneous approximation of multiple motif counts in large temporal networks. In *CIKM*. 1568–1577.

[45] Comandur Seshadhri, Ali Pinar, and Tamara G Kolda. 2013. Triadic measures on graphs: The power of wedge sampling. In *SDM*. 10–18.

[46] Aida Sheshbolouki and M Tamer Özsu. 2022. sGrapp: Butterfly approximation in streaming graphs. *TKDD* 16, 4 (2022), 1–43.

[47] Jessica Shi and Julian Shun. 2022. Parallel algorithms for butterfly computations. In *Massive Graph Analytics*. 287–330.

[48] Benjamin Steer, Felix Cuadrado, and Richard Clegg. 2020. Raphtory: Streaming analysis of distributed temporal graphs. *Future Generation Computer Systems* 102 (2020), 453–464.

[49] Xiaoli Sun, Yusong Tan, Qingbo Wu, Jing Wang, and Changxiang Shen. 2019. New algorithms for counting temporal graph pattern. *Symmetry* 11, 10 (2019), 1188.

[50] A Vazquez, R Dobrin, D Sergi, J-P Eckmann, Zoltan N Oltvai, and A-L Barabási. 2004. The topological relationship between the large-scale attributes and local interaction patterns of complex networks. *Proceedings of the National Academy of Sciences* 101, 52 (2004), 17940–17945.

[51] Jun Wang, Arjen P De Vries, and Marcel JT Reinders. 2006. Unifying user-based and item-based collaborative filtering approaches by similarity fusion. In *SIGIR*. 501–508.

[52] Jia Wang, Ada Wai-Chee Fu, and James Cheng. 2014. Rectangle counting in large bipartite graphs. In *IEEE International Congress on Big Data*. 17–24.

[53] Jingjing Wang, Yanhao Wang, Wenjun Jiang, Yuchen Li, and Kian-Lee Tan. 2022. Efficient Sampling Algorithms for Approximate Motif Counting in Temporal Graph Streams. *arXiv preprint arXiv:2211.12101* (2022).

[54] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2019. Vertex Priority Based Butterfly Counting for Large-scale Bipartite Networks. *Proceedings of the VLDB Endowment* 12, 10 (2019), 1139–1152.

[55] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2020. Efficient bitruss decomposition for large-scale bipartite graphs. In *ICDE*. 661–672.

[56] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2022. Accelerated butterfly counting with vertex priority on bipartite graphs. *The VLDB Journal* (2022), 1–25.

[57] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2022. Towards efficient solutions of bitruss decomposition for large-scale bipartite graphs. *The VLDB Journal* 31, 2 (2022), 203–226.

[58] Duncan J Watts and Steven H Strogatz. 1998. Collective dynamics of 'small-world'networks. *nature* 393, 6684 (1998), 440–442.

[59] Qingyu Xu, Feng Zhang, Zhiming Yao, Lv Lu, Xiaoyong Du, Dong Deng, and Bingsheng He. 2022. Efficient load-balanced butterfly counting on GPU. *PVLDB* 15, 11 (2022), 2450–2462.

[60] Carl Yang, Mengxiong Liu, Vincent W Zheng, and Jiawei Han. 2018. Node, motif and subgraph: Leveraging network functional blocks through structural convolution. In *ASONAM*. 47–52.

[61] Jianye Yang, Yun Peng, Dian Ouyang, Wenjie Zhang, Xuemin Lin, and Xiang Zhao. 2023. (p, q)-biclique counting and enumeration for large sparse bipartite graphs. *The VLDB Journal* (2023), 1–25.
[62] Ömer Nebil Yaveroğlu, Noël Malod-Dognin, Darren Davis, Zoran Levnajic, Vuk Janjic, Rasa Karapandza, Aleksandar Stojmirovic, and Nataša Pržulj. 2014. Revealing the hidden language of complex networks. *Scientific reports* 4, 1 (2014), 1–9.
[63] Na Zhang, Xuefeng Guan, Jun Cao, Xinglei Wang, and Huayi Wu. 2019. A hybrid traffic speed forecasting approach integrating wavelet transform and motif-based graph convolutional recurrent neural network. *arXiv preprint arXiv:1904.06656* (2019).
[64] Alexander Zhou, Yue Wang, and Lei Chen. 2021. Butterfly counting on uncertain bipartite graphs. *Proceedings of the VLDB Endowment* 15, 2 (2021), 211–223.
[65] Alexander Zhou, Yue Wang, and Lei Chen. 2023. Butterfly counting and bitruss decomposition on uncertain bipartite graphs. *The VLDB Journal* (2023), 1–24.
[66] Tao Zhou, Jie Ren, Matúš Medo, and Yi-Cheng Zhang. 2007. Bipartite network projection and personal recommendation. *Physical review E* 76, 4 (2007), 046115.