Efficient Historical Butterfly Counting in Large Temporal Bipartite Networks via Graph Structure-aware Index

Qiuyang Mang[*]
CUHK-Shenzhen
qiuyangmang@link.cuhk.edu.cn

Jingbang Chen[*]
University of Waterloo
j293chen@uwaterloo.ca

Hangrui Zhou[*]
Tsinghua University
zhouhr23@mails.tsinghua.edu.cn

Yu Gao
Independent
ygao2606@gmail.com

Yingli Zhou
CUHK-Shenzhen
yinglizhou@link.cuhk.edu.cn

Qingyu Shi
Independent
qingyuqwq@gmail.com

Richard Peng
Carnegie Mellon University
yangp@cs.cmu.edu

Yixiang Fang
CUHK-Shenzhen
fangyixiang@cuhk.edu.cn

Chenhao Ma[†]
CUHK-Shenzhen
machenhao@cuhk.edu.cn

ABSTRACT

Bipartite graphs are ubiquitous in many domains, e.g., e-commerce platforms, social networks, and academia, by modeling interactions between distinct entity sets. Within these graphs, the butterfly motif, a complete 2×2 biclique, represents the simplest yet significant subgraph structure, crucial for analyzing complex network patterns. Counting the butterflies offers significant benefits across various applications, including community analysis and recommender systems. Additionally, the temporal dimension of bipartite graphs, where edges activate within specific time frames, introduces the concept of historical butterfly counting, i.e., counting butterflies within a given time interval. This temporal analysis sheds light on the dynamics and evolution of network interactions, offering new insights into their mechanisms. Despite its importance, no existing algorithm can efficiently solve the historical butterfly counting task. To address this, we design two novel indices whose memory footprints are dependent on #butterflies and #wedges, respectively. Combining these indices, we propose a graph structure-aware indexing approach that significantly reduces memory usage while preserving exceptional query speed. To further reduce the index size and boost the query efficiency, we design an index compression strategy, enabling the fast, high-quality, and unbiased approximation of historical butterfly counts. We theoretically prove that our approach is particularly advantageous on power-law graphs, a common characteristic of real-world bipartite graphs, by surpassing traditional complexity barriers for general graphs. Extensive experiments reveal that our query algorithms outperform existing methods by up to five magnitudes, effectively balancing speed with manageable memory requirements.

PVLDB Reference Format:
Qiuyang Mang, Jingbang Chen, Hangrui Zhou, Yu Gao, Yingli Zhou, Qingyu Shi, Richard Peng, Yixiang Fang, and Chenhao Ma. Efficient Historical Butterfly Counting in Large Temporal Bipartite Networks via Graph Structure-aware Index. PVLDB, 18(6): 1607 - 1620, 2025.
doi:10.14778/3725688.3725693

[*]The first three authors contributed equally to this research.
[†]Chenhao Ma is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 18, No. 6 ISSN 2150-8097.
doi:10.14778/3725688.3725693

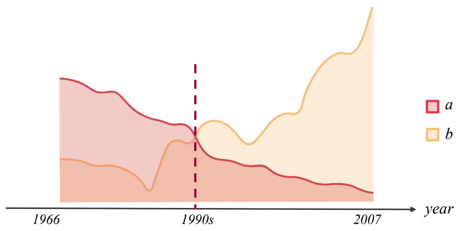


Figure 1: Jim Gray's activeness in database community (a) and astronomy (b) community.

PVLDB Artifact Availability:
The source codes and a detailed technical report have been made available at https://github.com/joyemang33/HBC.

1 INTRODUCTION

Due to its ability to model relationships between two distinct sets of entities, the bipartite graph, or network, holds significant importance across various fields, including disease control on people-location networks [11, 35], fraud detection on user-page networks [27, 45] and recommendation on customer-product networks [18, 19, 43, 46, 50]. To analyze the structure and dynamics of the network, counting motifs is one of the most popular methods [12, 13, 20, 31, 46] since motifs are considered the basic construction block of the network. The *butterfly* motif (2×2 biclique) represents fundamental interaction patterns within the graph. Counting it has wide applications ranging from biological ecosystems [37, 44, 45], where it helps in identifying mutual relationships between species, to social networks [46, 57], where it uncovers patterns of collaborations and affiliations.

Temporal bipartite graphs, in which edges typically carry timestamps, are often considered [5, 8, 12, 35] since real-world interactions (modeled as edges) usually occur at specific timestamps. Recently, Cai et al. [5] first considered counting butterflies on temporal bipartite graphs, which extends the analytical depth of traditional bipartite graph analyses by incorporating the dimension of time, making it a powerful tool for uncovering dynamic patterns in complex systems.

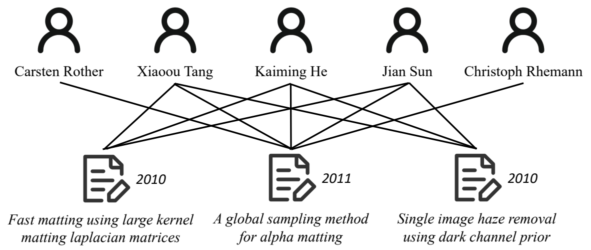

Carsten Rother
Xiaoou Tang
Kaiming He
Jian Sun
Christoph Rhemann
2010
2011
2010
Fast matting using large kernel matting laplacian matrices
A global sampling method for alpha matting
Single image haze removal using dark channel prior

Figure 2: Finding the time-window of the closest collaboration.

However, merely counting butterflies across the entire timeline, as suggested by [5], may not accurately reflect the dynamic nature of relationships, failing to capture evolving trends. To address this, it is essential to consider the temporal dimension of interactions. By focusing on the occurrence of motifs within specified time frames, we introduce the concept of *historical butterfly counting*. This approach, which involves analyzing butterflies within discrete time windows on a temporal bipartite graph, offers enhanced insights into the timing and progression of interactions. It provides an in-depth understanding of network dynamics, uncovering the mechanisms behind network evolution and revealing opportunities for precise interventions across various domains.

Applications We now discuss some interesting applications of historical butterfly counting.

• **Bipartite Clustering Coefficient (BCC) Computation.** The *bipartite clustering coefficient* [2, 26, 32] is a traditional cohesiveness measure for bipartite graphs whose computing bottleneck is counting butterflies. Specifically, considering the scientific collaboration network (modeled as a temporal graph) in the given time window, a higher BCC suggests a stronger trend of cohesive collaboration within the research community. This coefficient positively correlates with the number of author pairs publishing multiple publications within a given time-window and inversely correlates with the number of author pairs collaborating only once. For example, in academia, a scientist may change their frequent collaborators or research interests over time. For instance, examining two series of two-year BCCs within the research domains of Jim Gray in Figure 1, specifically the database and astronomy communities, reveals a gradual shift in his research interests from databases to astronomy around the 1990s. We also provide two case studies on the global research collaboration trend and close collaboration time windows in section 6.4 with BCC computation.

• **Identifying Close Communication Time-windows.** Directly counting butterflies can identify time windows during close communication within a specific community. For instance, we are interested in close collaboration. Among Kaiming He's 2-hop ego networks w.r.t. all possible time-windows, we find that the two-year time-window with the highest butterfly count is 2010 - 2011, with the corresponding research records detailed in Figure 2. During this period, Kaiming He, Jian Sun, Xiaoou Tang, Carsten Rother, and Christoph Rhemann established a series of cohesive research collaborations, resulting in the publication of three papers. In other words, Figure 2 depicts Kaiming's 2-hop ego network during 2010 - 2011, the two-year time interval with the highest butterfly count. Notably, 2010 and 2011, the last two years of Kaiming's Ph.D. under supervisor Xiaoou, marked his most cohesive collaborations.

Challenges and Contributions Counting butterflies in such a historical setting is challenging. One main reason is that the algorithm should be able to answer historical queries multiple times to analyze the changing trend. When the graph is large, it is inefficient to run existing butterfly counting algorithms from scratch for each query. Thus, we need to design algorithms that answer each query more efficiently after preprocessing. No algorithm can effectively solve this problem in existing works. This paper fills this hole by proposing a new index algorithm with consistently high performance in large-scale graphs. The proposed algorithm `GSI` (graph structure-aware index) can take advantage of graph structures and balance between query time and memory usage, enabling it to outperform previous butterfly counting algorithms on both real-world data, and synthetic data with certain distributions. `GSI` can also be parallelized, providing faster query efficiency. It can also be compressed, providing smaller memory usage. To summarize, we have made the following contributions.

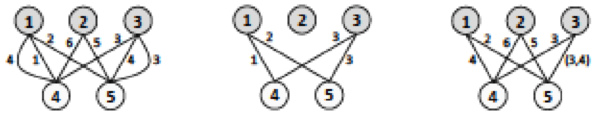
- We introduce the historical butterfly counting problem and provide its hardness. This enables in-depth trend and dynamics analysis over temporal graphs, and helps understand how temporal variations influence network structures over time.
- The graph structure-aware index (`GSI`) is designed to support efficient counting query with a controllable balance between query time and memory usage. Theoretically, we prove that the `GSI` approach transcends conventional computational complexity barriers associated with general graphs when applied to power-law graphs, a common characteristic of many real-world graphs.
- When exact counting is not required, we propose an index compression strategy to provide fast, high-quality, and unbiased approximations of the counts based on the compressed index.
- Extensive experiments demonstrate that our query algorithm achieves up to five orders of magnitude speedup over the state-of-the-art solutions with manageable memory footprints.

2 RELATED WORK

In this section, we review the related works, including the butterfly counting on static bipartite graphs, motif counting on temporal graphs, and other historical queries on temporal graphs.

• **Butterfly Counting on Static Bipartite Graphs.** Butterfly is the most fundamental sub-structure in bipartite graphs. Significant research efforts have been dedicated to the study of counting and enumerating butterflies on static bipartite graphs [37, 44, 45]. Wang et al. [44] first proposed the butterfly counting problem and designed an algorithm by enumerating wedges from a randomly selected layer. Sanei-Mehri et al. [37] further developed a strategy for choosing the layer to obtain better performance. Recently, Wang et al. [45] utilized the vertex priority and cache optimization to achieve state-of-the-art efficiency. Additionally, the parallel algorithms [37, 41], I/O efficient algorithm [48], sampling-based algorithms [23, 37, 40], GPU-based algorithm [40, 51], and batch update algorithm [47] have also been developed for the butterfly counting problem. In addition, the butterfly counting problems in steam, uncertain, and temporal bipartite graphs have also been studied [5, 38, 40, 56].

• **Motif Counting on Temporal Graphs.** The problem of temporal motif counting has been extensively studied recently



(a) a temporal bipartite graph G with duplicated edges. (b) a projected graph $G_{[1,3]}$, whose butterfly count is 1. (c) a projected graph $G_{[2,6]}$, whose butterfly count is 3.

Figure 3: A temporal bipartite graph and its projected graphs in two time-windows, associated with their butterfly counts.

[3, 17, 21, 24, 29]. Kovanen et al. [21] introduced the concept of Δ-adjacency, which pertains to two temporal edges sharing a vertex and having a timestamp difference of at most Δ, and consider the temporal ordering aspect. The Δ-temporal motif counting with [34, 36] and without temporal ordering [33] are also been studied. Furthermore, there are numerous approximation algorithms available for solving counting problems [28, 39]. When it comes to enumeration problems, isomorphism-based algorithms are the most commonly used [25, 30].

• **Other Historical Queries on Temporal Graphs.** The historical queries on the temporal graphs aim to compute the specific structure in the snapshot of an arbitrary time window. The historical reachability [49], k-core [54], structural diversity [7], and connected components [52] of temporal graphs have been defined, and index-based solutions have also been proposed. Note that our work is significantly different with [5], as our work is focused on more generalized scenarios of temporal graph mining, i.e., analyzing the relationship between vertices in a time interval and without any temporal ordering limitations.

3 PRELIMINARIES

Throughout the paper, we use the big-O notation to express the upper bounds of the *absolute values* of functions. For example, $O(\frac{1}{n})$ includes functions between $-\frac{c}{n}$ and $\frac{c}{n}$ for some constant c and sufficiently large n. We say $f(n) \geq g(n) + O(h(n))$ if $f(n) \geq g(n) - c|h(n)|$ for some constant c and sufficiently large n. We use $\tilde{O}(f(n))$ to denote all functions bounded by $O(f(n)\log^c n)$ for some constant c. We use $[n]$ to denote the set $\{1, \ldots, n\}$. For an undirected graph, we denote every vertex u's degree as deg_u.

3.1 Problem Definitions

Two common motifs are being widely studied on bipartite graphs: wedges and butterflies. We give their formal definition as follows:

Definition 3.1 (Wedge [45]). *Given a bipartite graph $G = (V = (U, L), E)$, a wedge $\langle x \rightsquigarrow y \rightsquigarrow z\rangle$ is a 2-hop path consisting of edges (x, y) and (y, z).*

Definition 3.2 (Butterfly [45]). *Given a bipartite graph $G = (V = (U, L), E)$, and the four vertices $x, y, z, w \in V$ where $x, z \in U$ and $y, w \in L$, $b : \langle x, y, z, w\rangle$ is a butterfly iff the subgraph induced by x, y, z, w is a $(2, 2)$-biclique of G; that is, x and z are all connected to y and w, respectively.*

Throughout the paper, we study the temporal bipartite graphs. A temporal bipartite graph is an undirected graph $G = (V = (U, L), E)$, where each edge $e \in E$ is a triple (u, v, t) with two vertices $u \in U, v \in L$ and a timestamp t. Our major focus is the historical type query on temporal bipartite graphs; that is, we query on an extracted graph from G with respect to a certain time window. We denote it as the *projected graph* and its formal definition is as follows:

Definition 3.3 (Projected graph [13]). *Given a temporal bipartite graph $G = (V, E)$ and a time-window $[t_s, t_e] (t_s \leq t_e)$, the projected graph $G_{[t_s,t_e]}$ of G is an undirected bipartite graph without timestamps, where its vertex set $V_{[t_s,t_e]}$ is V and the edge set $E_{[t_s,t_e]}$ is $\{(u, v) \mid \exists (u, v, t) \in E \ \wedge \ t \in [t_s, t_e]\}$.*

Now, we are ready to state the major problem formally:

Problem 1 (Historical butterfly counting). *Given a temporal bipartite graph G and a time-window $[t_s, t_e]$, find the number of butterflies in the projected graph $G_{[t_s,t_e]}$.*

In Figure 3, we are given a temporal bipartite graph G in (a), in which the number represents the timestamp of each edge. We consider two time-windows $[1, 3]$ and $[2, 6]$. The corresponding projected graphs $G_{[1,3]}$ and $G_{[2,6]}$ are shown in (b) and (c), respectively. The historical butterfly counting query associated with these two-time windows is, in fact, counting on these two projected graphs. Therefore, the answer is 1 ($\langle 1, 3, 4, 5\rangle$) for $[1, 3]$ and 3 ($\langle 1, 2, 4, 5\rangle$, $\langle 1, 3, 4, 5\rangle$, $\langle 2, 3, 4, 5\rangle$) for $[2, 6]$.

3.2 Some Key Techniques in Motif Counting

In the previous works on motif counting [45, 48], the value of $\frac{1}{|E|}\sum_{(u,v)\in E}\min(\deg_u, \deg_v)$ is widely used for complexity analysis for the given input graph $G = (V, E)$. By [9], this can be simplified as $O(\delta)$, where δ is defined as the arboricity of the given graph. We will adapt the δ notion in our paper for brevity.

Our proposed algorithms widely use two important techniques: the *Vertex Priority* method and the *Chazelle's structure*.

Vertex Priority. The vertex priority method reduces the number of wedges we need to consider. To begin with, we define the vertex priority as follows:

Definition 3.4 (Vertex priority [45]). *For any pair vertices x, y in a temporal bipartite graph G, we define x is prior to y ($pr(x) \prec pr(y)$) if and only if: $\overline{deg_x} > \overline{deg_y}$, or $\overline{deg_x} = \overline{deg_y}$ and $id(x) < id(y)$, where $\overline{deg_u}$ denotes the degree of u when only considering unique edges in G, and $id(u)$ denotes the unique ID of u.*

By [45], we only need to consider the wedges $\langle x \rightsquigarrow y \rightsquigarrow z\rangle$ that satisfy $pr(x) \prec pr(y) \wedge pr(x) \prec pr(z)$ in order to count the butterflies without repetition or missing. Each butterfly $\langle x, y, z, w\rangle$ is constructed from two such wedges $\langle x \rightsquigarrow y \rightsquigarrow z\rangle$ and $\langle x \rightsquigarrow w \rightsquigarrow z\rangle$.

2D-range Counting. Our algorithm will transform counting butterflies into counting points on a 2-dimensional plane, known as the *2D-range counting* problem. Specifically, let P be a set of n points in 2-d space $\mathbb{R}^2$. The *2D-range counting* problem is: Given an orthogonal rectangle Q of the form $[x_1, x_2] \times [y_1, y_2]$, find the size of $|Q \cap P|$. For an instance of the 2D-range counting problem, we use a classic data structure known as Chazelle's structure [6] to handle all 2D-range counting queries after preprocessing:

Theorem 3.1 (Chazelle's structure [6]). *A Chazelle's structure CS is a data structure that can answer each 2D-range counting in $O(\log n)$ time and $O(\frac{n \log n}{\omega})$ memory usage, where ω is the word size. The preprocessing time is $O(n \log n)$.*

In practice, 2^{ω} significantly exceeds the number of points involved in our method's 2D-range counting task. Therefore, we assume that $\log n$ is $O(\omega)$ for Chazelle's structures used and simplify the memory usage into $O(n)$ for brevity. In later sections, we use $\mathcal{CS}$ to denote such a data structure.

3.3 Baselines

Our baseline solutions are built upon the *states-of-the-arts* methods for exact butterfly counting (i.e., BFC-VP++ [45]) and approximate butterfly counting (i.e., Weighted Pair Sampling (WPS) [55]) on the static graphs. We need to extract the static graph from the temporal bipartite graph for a given time-windows first, and then run the following solutions:

- BFC-VP++: The BFC-VP++ algorithm sorts vertices by their proposed vertex priority (Definition 3.4) and efficiently identifies almost all minimally redundant wedges that can form a butterfly, offering a practical, efficient, and cache-friendly solution. In addition, BFC-VP++ can be highly parallelized. We will also compare our methods with its parallel version in the following.
- WPS: The basic idea behind WPS is to estimate the total butterfly count with the number of butterflies containing two randomly sampled vertices from the same side of the two vertex sets. The method has been proven as unbiased and theoretically efficient in power-law graph models.

4 INDEX-BASED ALGORITHMS

In this section, we consider solving Problem 1 exactly by index-based solutions. To begin with, in Section 4.1, we give a hardness result showing that $\bar{O}(m^2/\lambda^2)$ space is needed to answer queries exactly in $\bar{O}(\lambda)$ time, where m denotes the number of edges in the given temporal bipartite graph G. Correspondingly, we provide an algorithm that meets such bound in Section 4.2, named as *Enumeration-based Index (EBI)*. EBI achieves $\bar{O}(1)$ query time but it needs expensive memory usage for large graph data. In Section 4.3, we introduce a different algorithm named *Combination-based Index (CBI)* that can be constructed under practical memory constraints.

To bridge the gap between theory and practice, we propose a new algorithm in Section 4.4 that effectively combines EBI and CBI, demonstrating strong performance on real-world graph data. This algorithm, named *Graph Structure-aware Index (GSI)*, intelligently allocates graph data to the two indices which allows it to take advantage of the underlying graph structure. In addition, without compromising the performance, GSI can also handle duplicate edges with proper modification, which is discussed in Section 4.6.

4.1 Problem Hardness

Our result is motivated by the hardness result in [10] where they reduce the Set Disjointness problem (Definition 4.1) into Problem 1. While [10] considers timestamps on vertices, we prove for the setting where timestamps are on edges separately.

Definition 4.1 (Set Disjointness). *Given a collection of $s \geq 2$ sets $S_1, S_2, \ldots, S_s$, a query $(a, b) \in [s]^2$ asks for whether $S_a \cap S_b$ is empty.*

The strong set disjointness conjecture [14, 15] is as follows:

Theorem 4.1. *For the set disjointness problem, any data structure with query time λ must use $\bar{\Omega}(n^2/\lambda^2)$ space where n is the sum of the sizes of $S_1, \ldots, S_s$.*

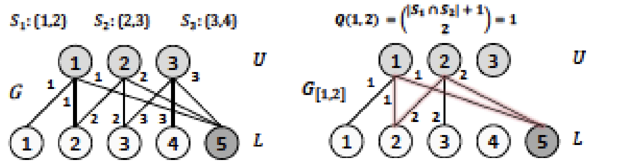


Figure 4: An example for reducing the set disjointness problem into exact historical butterfly counting in temporal bipartite graphs.

Specifically, we prove the following result for Problem 1:

Theorem 4.2. *Consider Problem 1. Let m denote the number of edges in G. Fix any $\lambda \in [1, m]$ and $\delta > 0$. Suppose that we have a data structure for Problem 1 using $\bar{O}(m^{2-\delta}/\lambda^2)$ space and exactly answers each query in $\bar{O}(\lambda)$ time. Then, for any set disjointness problem with $\sum_{i=1}^{s} |S_s| = N$, we have a data structure that uses $\bar{O}(N^{2-\delta}/\lambda^2)$ time and that +answers each query in $\bar{O}(\lambda)$ time.*

Proof. Suppose that we have a data structure for Problem 1 using $\bar{O}(m^{2-\delta}/\lambda^2)$ space and exactly answers each query in $\bar{O}(\lambda)$ time. We can utilize the data structure for a set disjointness instance as follows:

Let there be s sets $S_1, \ldots, S_s$. Without loss of generality, we let $1, \ldots, t$ denote all distinct elements in $S_1, \ldots, S_s$, *i.e.*, $\bigcup_{i=1}^{s} S_i = [t]$. We construct a graph $G = (V = (U, L), E)$ such that $U = [s]$ and $L = [t+1]$. An edge (i, j) exists if $j \in S_i$ or $j = t+1$. We assign timestamp i to the edge (i, j).

We build the data structure for the graph G defined above. For two integers $a \leq b \in [s]$, we can query the data structure for the number of butterflies in the time window $[a, b]$. Since the timestamp on edge (i, j) is equal to i, such butterflies are exactly those with the form $\langle u, v, w, x \rangle$ such that $u, w \in [a, b]$. The number of such butterflies is

$$Q(a, b) \overset{\text{def}}{=} \sum_{i=a}^{b} \sum_{j=i+1}^{b} \binom{|S_i \cap S_j| + 1}{2}.$$

Notice that $Q(a, b)$ is the two-dimensional prefix sum of $\binom{|S_a \cap S_b|+1}{2}$. In other words, we can calculate $\binom{|S_a \cap S_b|+1}{2}$ by $Q(a, b) - Q(a+1, b) - Q(a, b-1) + Q(a+1, b-1)$, where we define $Q(x, y) = 0$ for $x \geq y$. We can answer whether $|S_a \cap S_b| = 0$ by checking whether this value is equal to zero.

□

For example, in the Figure 4, we reduce the set disjointness problem for three sets $\{1, 2\}$, $\{2, 3\}$ and $\{3, 4\}$ to a historical butterfly counting problem on the graph we constructed. For determining whether $S_1 \cap S_2$ is empty, we count the number of butterflies of the form $\langle 1, v, 2, x \rangle$, $Q(1, 2)$, which is equal to $\binom{|S_1 \cap S_2|+1}{2}$. For determining whether $S_1 \cap S_3$ is empty, we calculate $\binom{|S_1 \cap S_3|+1}{2} = Q(1, 3) - Q(2.3) - Q(1, 2) + Q(2, 2) = 2 - 1 - 1 + 0 = 0$.

This hardness result shows that to achieve efficient query runtime (e.g., $\bar{O}(\lambda)$), large memory space is necessary (e.g., $\bar{O}(m^2/\lambda^2)$). Even though we can design theoretically optimal algorithms that reach the lower bound, they might not be practical when the input graph is large. One possible way to overcome such a challenge from a practical perspective is to develop algorithms that take advantage of real-world graphs' properties.

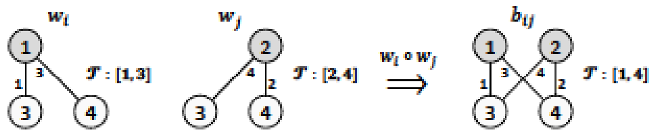


Figure 5: An illustrative example for active timestamps of wedges and the butterfly constructed from them.

4.2 Enumeration-based Index

Since the structure of a temporal graph changes over time, each subgraph has its own life cycle, i.e., the only interval of time in which it exists. We denote such an interval as the active timestamp of this subgraph. We give a formal definition as follows:

Definition 4.2 (Active Timestamp). *For any subgraph P of a bipartite temporal graph G, we define the active timestamp $\mathcal{T}(P)$ as a pair of ordered timestamp $[l,r](l \le r)$, such that P exist in the projected graph $G_{[t_s,t_e]}$ if and only if $t_s \le l$ and $r \le t_e$.*

In Figure 5, there are two wedges w_i and w_j. For w_i, since it contains two edges $(1,3,1)$ and $(1,4,3)$, it only exists in a projected graph $G_{[t_s,t_e]}$ satisfying $t_s \le 1$ and $t_e \ge 3$. The active timestamp $\mathcal{T}(w_i)$ is $[1,3]$. Similarly, for w_j, its active timestamp is $[2,4]$.

The proposed *Enumeration-based Index (*EBI*)* algorithm is shown in Algorithm 1 and Algorithm 2. For brevity, we assume that no duplicate edge exists. We discuss how to handle duplicate edges without compromising the performance in Section 4.6.

• **Construction** The goal is to enumerate every butterfly and compute its active timestamp. After initializing a 2D-range counting data structure T_E by all butterflies' timestamps, we can answer every historical butterfly counting query through a single query on T_E. We initialize it to be empty in the beginning (Line 1).

To find all the butterflies in G, we enumerate all the wedges. We consider two wedges w_i, w_j of G with shared endpoints but different middle vertices, denoted as $w_i : \langle x \rightsquigarrow y_i \rightsquigarrow z\rangle$ and $w_j : \langle x \rightsquigarrow y_j \rightsquigarrow z\rangle$. Let $\mathcal{T}(w_i) = [l_i, r_i]$ and $\mathcal{T}(w_j) = [l_j, r_j]$. The temporal butterfly constructed from them is denoted as $b_{ij} = w_i \circ w_j$, where $\mathcal{T}(b_{ij}) = [\min(l_i, l_j), \max(r_i, r_j)]$.

In Figure 5, we demonstrate how to construct a butterfly b_{ij} with two wedges w_i and w_j. Since b_{ij} contains edges $(1,3,1)$, $(1,4,3)$, $(2,3,4)$, and $(2,4,2)$, its active timestamp can be computed by $l = \min\{t \mid \exists e = (u,v,t) \in edges(b_{ij})\} = 1$, and $r = \max\{t \mid \exists e = (u,v,t) \in b_{ij}\} = 4$. By the definition of the $\circ$ operation, it can also be computed from $\mathcal{T}(w_i)$ and $\mathcal{T}(w_j)$. This also indicates that a butterfly $b_{ij} = w_j \circ w_j$ exists in a given time-window, if and only if the active timestamp of w_i and w_j are both included in the window. Such observation helps us develop an alternative index approach in the later section.

We follow the method in [45] to enumerate every wedge $w_i : \langle x \rightsquigarrow y \rightsquigarrow z\rangle$ $[l_i, r_i]$ in G without repetition or missing (Line 3). The complexity is bounded by the total number of wedges in G. Let $W[(x,z)]$ be a set of wedges whose endpoints are x, z. W is the family of sets $W[(x,z)]$. W is initialized to be empty (Line 2). For the current wedge w_i, we first check if its endpoints (x,z) exists in W (Line 4) (by a hash table, for example). If not, we initialize the corresponding set $W[(x,z)]$ to be empty (Line 4).

Since any two different wedges with the same pair of endpoints (x,z) construct a unique butterfly, we enumerate every w_j in $W[(x,z)]$ (Line 5) and construct a butterfly $b_{ij} = w_i \circ w_j$ with $\mathcal{T}(b_{ij}) = (t_l, t_r)$ (Line 6). Correspondingly, we insert a single 2D point (t_l, t_r) into T_E. After enumeration, we need to update $W[(x,z)]$ by inserting w_i in it (Line 8). After finding all the butterflies, we run the preprocessing for T_E and return it (Line 10).

In Figure 6, we provide an example of our construction process. There are three wedges $w_1 : \langle 1 \rightsquigarrow 4 \rightsquigarrow 2\rangle$, $w_2 : \langle 1 \rightsquigarrow 5 \rightsquigarrow 2\rangle$, $w_3 : \langle 1 \rightsquigarrow 4 \rightsquigarrow 3\rangle$, $w_4 : \langle 1 \rightsquigarrow 5 \rightsquigarrow 3\rangle$, $w_5 : \langle 2 \rightsquigarrow 4 \rightsquigarrow 3\rangle$, $w_6 : \langle 2 \rightsquigarrow 5 \rightsquigarrow 3\rangle$ where $\mathcal{T}(w_1) = [4,6]$, $\mathcal{T}(w_2) = [2,5]$, $\mathcal{T}(w_3) = [4,4]$, $\mathcal{T}(w_4) = [2,4]$, $\mathcal{T}(w_5) = [4,6]$, $\mathcal{T}(w_6) = [4,5]$. We group them by (x,z). There will be three groups (sets) of wedges: $W[(1,2)] = \{w_1, w_2\}$, $W[(1,3)] = \{w_3, w_4\}$, $W[(2,3)] = \{w_5, w_6\}$. For each group, we construct a butterfly b_{ij} from each pair of different wedges w_i, w_j from the group. We have $b_1 = w_1 \circ w_2$, $b_2 = w_3 \circ w_4$, $b_3 = w_5 \circ w_6$. We compute their active timestamp through simple calculation: $\mathcal{T}(b_1) = [2,4]$, $\mathcal{T}(b_2) = [2,6]$, $\mathcal{T}(b_3) = [4,6]$. In the end, we insert $(2,4)$, $(2,6)$, $(4,6)$ into T_E. Note that in this showcase, for demonstrating the group dividing, we do not follow the vertex priority in Definition 3.4. Otherwise, we only consider three wedges and they are all in the group $W[(4,5)]$.

• **Answering Query** Since each butterfly with active timestamp $[l,r]$ is represented by a point (l,r) in T_E, a historical butterfly counting of time-window $[t_s, t_e]$ can be interpreted as a 2D-range query counting the number of points in the rectangle $[t_s, \infty] \times [-\infty, t_e]$. We query T_E for it and return the answer directly (Line 1).

In Figure 6, after the preprocessing of T_E, we can answer any 2D-range counting query concerning those three points, which represent the three butterflies. When we want to query the number of butterflies in time-window $[1,5]$, we are asking how many butterflies' active timestamps are in the range of $[1,5]$. We interpret this as querying the number of points in the rectangle $[1,\infty] \times [-\infty, 5]$. There is only one point $(2,4)$ in the rectangle. Therefore, there is only one butterfly in the projected graph of the queried time-window. The calculation is done by a single query on T_E. It is a typical 2D-range counting query.

• **Time and Space Complexity** The time and space complexity of EBI is summarized as follows:

Theorem 4.3. *Denote B as the number of butterflies of the input graph G and δ as the arboricity, EBI's construction (Algorithm 1) runs in $O(m\delta + B\log m)$ and takes $O(B)$ space. After the construction, each query (Algorithm 2) takes $O(\log m)$ to answer.*

Proof. The construction process requires enumerating all wedges. By [45], this takes $O(m\delta)$. For each wedge $w_i : \langle x \rightsquigarrow y \rightsquigarrow z\rangle$, it is inserted in $W[(x,z)]$. By enumerating every existing w_j in $W[(x,z)]$, every butterfly $b_{ij} = w_i \circ w_j$ is enumerated exactly once. Since the $\circ$ operation takes $O(1)$ and the insertion to T_E takes $O(\log m)$ by Theorem 3.1, the total preprocess time for all butterflies takes $O(B \log m)$. Therefore, the total runtime for Algorithm 1 is $O(m\delta + B\log m)$ as desired. Every wedge will be stored in $W[(x,z)]$ where x, z are its endpoints. All the wedges take $O(m\delta)$ space in total. T_E contains $O(B)$ points in total. By Theorem 3.1, it takes $O(B)$ space. The total memory usage sum up to $O(B + m\delta)$. Each query on EBI is a range query on T_E. By Theorem 3.1, this takes $O(\log m)$ time and no additional space. □

The EBI algorithm reaches the bound in Section 4.1: Since δ is bounded by $O(\sqrt{m})$ and B is bounded by $O(m^2)$, the memory usage is bounded by $\tilde{O}(m^2)$. Therefore, EBI answers each query in $\tilde{O}(1)$

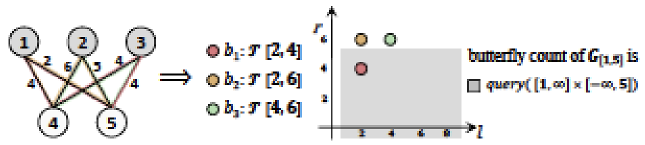

1
2
3
4
5
2
6
5
4
4
4
⟹
b_1: $\mathcal{T}$ [2, 4]
b_2: $\mathcal{T}$ [2, 6]
b_3: $\mathcal{T}$ [4, 6]
r
6
4
2
2
4
6
8
l
butterfly count of $G_{[1,5]}$ is
query([1, ∞] × [−∞, 5])

Figure 6: An example for transforming historical butterfly counting into 2D-range counting.

```
Algorithm 1: EBI-Construction
  Input: a bipartite temporal graph: $G = (V, E)$;
  Output: a $\mathcal{CS}$ (Theorem 3.1) for indexing (EB-index): $T_E$;
1 Initialize $T_E$ with an empty $\mathcal{CS}$;
2 $W$ ← an empty hashmap mapping pairs of vertices to sets;
3 for each wedge $w_i : \langle x \rightsquigarrow y \rightsquigarrow z\rangle\ [l_i, r_i] \in G$ do
4     if $(x, z) \notin W.keys()$ then $W[(x, z)] \leftarrow \emptyset$ ;
5     for $w_j \in W[(x, z)]$ do
6         $b_{ij} \leftarrow w_i \circ w_j$;
7         $T_E.insert(t_l, t_r)$;                // $(t_l, t_r) = \mathcal{T}(b_{ij})$
8     $W[(x, z)] \leftarrow W[(x, z)] \cup w_i$;
9 Preprocess $T_E$ (Theorem 3.1);
10 return $T_E$;
```

```
Algorithm 2: EBI-Querying
  Input: a time-window: $[t_s, t_e]$; a EB-index: $T_E$;
  Output: temporal butterfly counting: $num_{[t_s,t_e]}$;
1 return $num_{[t_s,t_e]} \leftarrow T_E.query([t_s, \infty] \times [-\infty, t_e])$;
```

time and take $\tilde{O}(m^2)$ space. This indicates that `EBI`'s asymptotic memory usage cannot be further improved without affecting the $\tilde{O}(1)$ query time. On the other hand, though reaching theoretical optimality, `EBI`'s performance on large-scale data is not ideal because enumerating and storing all butterflies is challenging for these graphs. For example, in the `Wiktionary` dataset[1], the butterfly count exceeds 10^{16} where the graph contains about 4×10^7 edges[2].

4.3 Combination-based Index

If the graph is large, it is not realistic to pre-compute and store all butterflies to handle historical queries, as this requires too much space. On the other hand, we are only interested in the number of butterflies, so it is not necessary to construct them explicitly.

Recall that a butterfly exists in a given time window if and only if the active timestamp of the two wedges are both included in the window. Consider a *group of wedges* $S = \{\langle x \rightsquigarrow y \rightsquigarrow z\rangle\}$ where x, z are fixed. Any two wedges from S form a butterfly. Therefore, the total number of butterflies that comes from S is $\binom{|S|}{2}$. This idea leads us to a different index algorithm, named *Combination-based Index* (`CBI`). The implementation details are provided in Algorithm 3 and Algorithm 4. Similarly, we assume no duplicate edge for brevity. Resolving them is deferred to Section 4.6.

• **Construction** In `EBI`, we use one 2D-counting data structure to store butterflies. Instead, in `CBI`, we map every type of wedge to a separate data structure. We initialize a hashmap T_C in the beginning (Line 1). We still need to enumerate every wedge $w_i : \langle x \rightsquigarrow y \rightsquigarrow z\rangle\ [l_i, r_i]$ in G (Line 2). A wedge with endpoints x, z is maintained by the 2D-counting data structure $T_C[(x, z)]$. If $T_C[(x, z)]$ does not exist(Line 3), we initialize an empty $\mathcal{CS}$ (Theorem 3.1) for $T_C[(x, z)]$ (Line 4). Unlike in `EBI`, where we compute every butterfly that is constructed from the new wedges w_i and a previous wedge $w_j \in W[(x, z)]$, we directly insert a point $[l_i, r_i]$ (w_i's active timestamp) into the corresponding $T_C[(x, z)]$ (Line 5). After the enumeration, we preprocess all data structures $T_C[(x, z)]$ and return T_C (Line 7).

• **Answering Query** To answer a historical butterfly counting of time-window $[t_s, t_e]$, for every type of wedges, we first need to compute the number of wedges that exist in the time-window, then compute the number of butterflies by a simple binomial coefficient. We set a counter $num_{[t_s,t_e]}$ to be 0 initially (Line 1). We enumerate every existing data structure in T_C (Line 2). Similar to what we do in `EBI`, we query on the corresponding 2D-counting data structure $T_C[(x, z)]$ with a query $[t_s, \infty] \times [-\infty, t_e]$ to get the number of wedges that are active in the time-window (Line 3). Denoting the query answer as *count*, we add $\binom{count}{2}$ to $num_{[t_s,t_e]}$, which is the number of butterflies that are constructed from these wedges. After enumerating all the data structures, we return $num_{[t_s,t_e]}$ as the total number of butterflies (Line 7).

• **Time and Space Complexity** The time and space complexity of `CBI` is summarized as follows:

Theorem 4.4. *Denote $\bar{w}$ as the number of wedge groups and δ as the arboricity. `CBI`'s construction (Algorithm 3) runs in $O(m\delta \log n)$ and takes $O(m\delta)$ space. After the construction, `CBI` answers each query (Algorithm 4) in $O(\bar{w} \log n)$ time.*

Proof. The construction process (Algorithm 3) also requires enumerating all wedges, which takes $O(m\delta)$ by [45]. For each group of wedges with the same endpoints, a $\mathcal{CS}$ needs to be initialized with all wedges from this group. Since the number of wedges is bounded by $O(m\delta)$ and each group contains at most $O(n)$ wedges, the total time for preprocessing all the data structures is $O(m\delta \log n)$ by Theorem 3.1. Similar to Theorem 4.3, it takes $O(m\delta)$ memory in total. When answering a query (Algorithm 4), all groups of wedges need to be enumerated, and their corresponding $\mathcal{CS}$ will be queried exactly once. The total query time will be $O(\bar{w} \log n) = \bar{w} \times O(\log n)$ by Theorem 3.1. Memory usage is dominated by 2D-range counting data structures for each group of wedges. □

As a result, `CBI` manages to resolve the memory issue of `EBI` when there are too many butterflies. Although $\bar{w}$ is bounded by the number of wedges (*i.e.*, $m\delta$), in practice, it is significantly smaller (*i.e.*, $\bar{w} \lll m\delta$). For example, in the `Wiktionary` dataset, there is about $m\delta \approx 1.3 \times 10^9$ wedges but only $\bar{w} \approx 5 \times 10^7$ wedge groups with different (x, z).

By further observing the distribution of wedges on real-world data, we see a concentration phenomenon that our algorithm has not addressed: Most wedges are grouped in a few groups, while the rest contain only a very small number of wedges. Therefore, building a separate data structure for every wedge group might be too expensive and unnecessary, especially for groups that only contribute a small amount of butterflies. For example, in the `Wiktionary (WT)` dataset, more than 81.3% wedge groups contain less than 10 wedges.

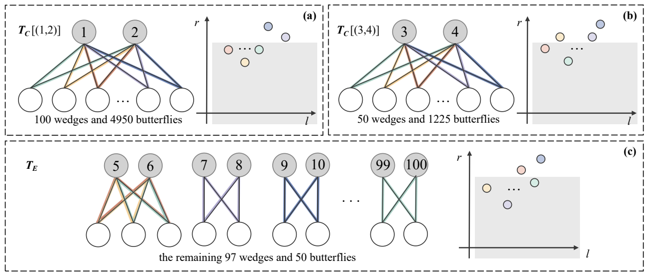
[1]http://konect.cc/
[2]Here, #butterflies is larger than the square of #edges due to duplicate edges, which will be handled in Section 4.6.

Algorithm 3: CBI-Construction

```
  Input: a bipartite temporal graph: $G = (V, E)$;
  Output: a hashmap mapping pairs of vertices to multiple $\mathcal{CS}$s: $T_C$;
1 $T_C \leftarrow$ an empty hashmap mapping pairs of vertices to $\mathcal{CS}$s;
2 for each wedge $w_i : \langle x \rightsquigarrow y \rightsquigarrow z \rangle\ [l_i, r_i] \in G$ do
3     if $(x, z) \notin T_C.keys()$ then
4         Initialize $T_C[(x, z)]$ with an empty $\mathcal{CS}$;
5     $T_C[(x, z)].insert(l_i, r_i)$;
6 Preprocess each data structure in $T_C$ (Theorem 3.1);
7 return $T_C$;
```

Algorithm 4: CBI-Querying

```
  Input: a time-window: $[t_s, t_e]$; a CB-index: $T_C$;
  Output: temporal butterfly counting: $num_{[t_s,t_e]}$;
1 $num_{[t_s,t_e]} \leftarrow 0$;
2 for $(x, z) \in T_C.keys()$ do
3     $count \leftarrow T_C[(x, z)].query([t_s, \infty] \times [-\infty, t_e])$;
4     $num_{[t_s,t_e]} \leftarrow num_{[t_s,t_e]} + \binom{count}{2}$
5 return $num_{[t_s,t_e]}$;
```


(a)
$T_C[(1,2)]$
1
2
r
l
100 wedges and 4950 butterflies
(b)
$T_C[(3,4)]$
3
4
50 wedges and 1225 butterflies
(c)
T_E
5
6
7
8
9
10
99
100
the remaining 97 wedges and 50 butterflies
Figure 7: An illustrative example for demonstrating GSI.

4.4 Graph Structure-aware Index

Recall that the number of butterflies dominates the memory usage of EBI, and the number of wedge groups greatly affects the query time of CBI. They fail to take into account the actual structure of the input graph. If we assume that most wedge groups only contain a very small amount of wedges, we might be able to take advantage of both EBI and CBI by distributing each wedge group into one of EBI and CBI by its size. As the main result of our paper, we propose a new index algorithm named *Graph Structure-aware Index (GSI)* with such an idea. Besides its promising performance on real-world data, it can also be parallelized to reach an even better performance. The implementation details are provided in Algorithm 5 and Algorithm 6.

• **Construction** The important assumption we make here is that a small number of groups contain most of the wedges. Our key idea is to store them in the same way as CBI, building a 2D-range counting data structure for each index, maintained by a hashmap T_C. For the rest of the wedges, we precompute all butterflies constructed from them and store them in one data structure T_E like EBI. In the beginning, in addition to T_E (Line 1) and T_C (Line 2), we also initialize a hashmap W to group wedges directly (Line 3). We enumerate all $w_i : \langle x \rightsquigarrow y \rightsquigarrow z \rangle\ [l_i, r_i]$ in G (Line 4) and store them into the corresponding set mapped by $W[(x, z)]$ (Line 6) . If the key does not exist in W, we need to initialize the set to be empty (Line 5). After grouping wedges, we sort the sets in W with decreasing order of sizes. We also need to compute the total number of butterflies and store it in *num* and *numB* (Line 8 to Line 10).

We introduce a parameter α to control the cutoff between the two methods. More specifically, we build a data structure for some sets in W such that the total number of butterflies constructed from them does not exceed α fraction of all the butterflies in G. The butterflies constructed from the other sets in W are maintained by one data structure per set.

We enumerate the sets in W by decreasing the order of sizes (Line 11). *num* represents the number of butterflies we processed. If we have not processed the majority of butterflies, i.e., when *num* is no less than $\alpha \cdot numB$ (Line 12), we build a $\mathcal{CS}$ for every wedge in $W[(x, z)]$, inserting points representing their active timestamps (Line 13 to Line 15). *num* is subtracted by $\binom{|W[(x,z)]|}{2}$, indicating that these of butterflies are recorded by $T_C[(x, z)]$ (Line 21).

When *num* is smaller than $\alpha \cdot numB$, we have already handled the majority of butterflies. Now, we need to process butterflies built from the rest of the wedges groups. Though each of these groups only contains a relatively small amount of wedges, the number of groups might be large. Here we apply the idea of EBI. We enumerate every possible wedge pair in the current enumerated $W[(x, z)]$ (Line 17 to Line 18), compute the butterfly $b_{ij}[t_l, t_r] \leftarrow w_i \circ w_j$ (Line 19) and insert a point (t_l, t_r) into the global data structure T_E (Line 20). In the end, after preprocessing every $\mathcal{CS}$ in T_C and T_E, we return them and finish the construction (Line 23).

An example of construction is shown in Figure 7. In the input graph, there are 100 wedges with endpoints $(1, 2)$ and 50 wedges with endpoints $(3, 4)$, while there are only 97 wedges in other groups. We treat these two groups with the CBI method and build $T_C[(1, 2)]$ (a), $T_C[(3, 4)]$ (b) correspondingly. We compute all 50 butterflies for the remaining wedges and store them in T_E (c).

• **Answering Query** For a historical butterfly counting of time-window $[t_s, t_e]$, we need to accumulate answers from both T_C and T_E. As in EBI, we directly query the 2D range $[t_s, \infty] \times [-\infty, t_e]$ on T_E (Line 1). Then, as in CBI, we enumerate all groups of wedges from T_C (Line 2). We query $T_C[(x, z)]$ with the same 2D range (Line 3) and add up the number of butterflies constructed from each group of wedges (Line 4). We return the answer as the sum of these two parts (Line 5). In the Figure 7 showcase, we query from all three $\mathcal{CS}$s: $T_C[(1, 2)]$, $T_C[(3, 4)]$, T_E and sum up the answer.

• **Time and Space Complexity** The time and space complexity of GSI is summarized as follows:

Theorem 4.5. *Denote β as the number of $\mathcal{CS}$ building for wedge groups of the input graph G and δ as the arboricity. GSI's construction (Algorithm 5) runs in $O(m\delta \log n + \alpha B \log m)$ and takes $O(m\delta + \alpha B)$ space. After the construction, each query (Algorithm 6) takes $O(\beta \log n + \log m)$ to answer.*

Proof. The construction process (Algorithm 5) requires enumerating all wedges, which takes $O(m\delta)$ by [45]. The rest of the algorithm can be divided into two parts: a CBI instance of β wedges group containing $(1 - \alpha)B$ butterflies and an EBI instance containing αB butterflies. By Theorem 4.3 and Theorem 4.4, the total time complexity is $O(m\delta \log n + \alpha B \log m)$ and the corresponding query complexity (Algorithm 6) is $O(\beta \log n + \log m)$. The memory usage is $O(m\delta + \alpha B)$. □

Algorithm 5: GSI-Construction

```
   Input: a bipartite temporal graph: $G = (V, E)$; a parameter for
          space using: $\alpha$;
   Output: a $\mathcal{CS}$ for indexing (EB-index): $T_E$; a hashmap of $\mathcal{CS}$ for
           indexing: $T_C$;
1  Initialize $T_E$ with an empty $\mathcal{CS}$;
2  $T_C \leftarrow$ an empty hashmap mapping pairs of vertices to $\mathcal{CS}$s;
3  $W \leftarrow$ an empty hashmap mapping pairs of vertices to sets;
4  for each wedge $w_i : \langle x \rightsquigarrow y \rightsquigarrow z \rangle\ [l_i, r_i] \in G$ do
5      if $(x, z) \notin W.keys()$ then $W[(x, z)] \leftarrow \emptyset$ ;
6      $W[(x, z)] \leftarrow W[(x, z)] \cup w_i$;
7  Sort sets in $W$ with decreasing order of sizes;
8  $num \leftarrow 0$;
9  for $(x, z) \in W.keys()$ do $num \leftarrow num + \binom{|W[(x,z)]|}{2}$ ;
10 $numB \leftarrow num$;
11 for $(x, z) \in W.keys()$ do
12     if $num \geq \alpha \cdot numB$ then
13         Initialize $T_C[(x, z)]$ with an empty $\mathcal{CS}$;
14         for $w_i : [l_i, r_i] \in W[(x, z)]$ do
15             $T_C[(x, z)].insert(l_i, r_i)$;
16     else
17         for $w_i : [l_i, r_i] \in W[(x, z)]$ do
18             for $w_j : [l_j, r_j] \in W[(x, z)] \wedge (j > i)$ do
19                 $b_{ij} \leftarrow w_i \circ w_j$;
20                 $T_E.insert(t_l, t_r)$;            // $(t_l, t_r) = \mathcal{T}(b_{ij})$
21     $num \leftarrow num - \binom{|W[(x,z)]|}{2}$;
22 Preprocess $T_E$ and each data structure in $T_C$ (Theorem 3.1);
23 return $T_E, T_C$;
```

Algorithm 6: GSI-Querying

```
   Input: a time-window: $[t_s, t_e]$; GSI-indexes: $T_E, T_C$;
   Output: temporal butterfly counting: $num_{[t_s,t_e]}$;
1  $num_{[t_s,t_e]} \leftarrow T_E.query([t_s, \infty] \times [-\infty, t_e])$;
2  for $(x, z) \in T_C.keys()$ do
3      $count \leftarrow T_C[(x, z)].query([t_s, \infty] \times [-\infty, t_e])$;
4      $num_{[t_s,t_e]} \leftarrow num_{[t_s,t_e]} + \binom{count}{2}$
5  return $num_{[t_s,t_e]}$;
```

Note that β is determined by α and bounded by the number of wedge groups $\widetilde{w}$. In real-world datasets with practical memory constraints, β is significantly smaller than $\widetilde{w}$ (*i.e., $\beta \ll \widetilde{w}$*). For example, in the WT dataset and under a 500 GB memory constraint, the β of GSI is 6986 when setting the optimal α that maximize the memory utilization, while the $\widetilde{w}$ is 3, 925, 064.

As a result, GSI manages to address the concentration phenomenon of the wedges distribution on real-world graphs, taking advantage of both EBI (for scattered wedges) and CBI (for concentrated wedges). It also has flexibility regarding the different limitations of memory. In addition, under the well-known power-law model [1], GSI has a non-trivial complexity, which is analyzed in Section 5.

4.4.1 Automatically Determining α. The parameter α reflects a trade-off between query time and memory usage. Therefore, selecting a proper α based on the input graph and the memory limit is important. We introduce an easy method to automatically select an efficient α before executing GSI. In the beginning, we set $\alpha = 0$, which means that all wedge groups will be maintained separately with a $\mathcal{CS}$ data structure. We sort wedge groups by size from small to large and enumerate them one by one. For each enumerated wedge group i with w_i wedges, we increase α by $\binom{w_i}{2}/B$, where B is the number of butterflies, precomputed in Line 8 to Line 9. We can estimate the memory usage by simple calculation as Chazelle's structure's memory usage can be precomputed based on the number of inserted points in $O(1)$. We stop and determine α if the estimated memory exceeds the limit with the next wedge group. This process takes $O(m\delta)$ time and avoids actually building indexes.

4.4.2 Parallelized Querying. After the construction process, when answering a query (Algorithm 6), we are, in fact, summing up the answer from several independent $\mathcal{CS}$ data structures, which motivates us to utilize parallelization to speed up. Assume we have multiple threads, and these threads can handle different $\mathcal{CS}$s simultaneously. Since no conflict occurs when these threads read the indexes simultaneously, we consider Algorithm 6 is highly parallelizable when the number of $\mathcal{CS}$s greater than the number of threads. To demonstrate, we implement and evaluate a parallel version of GSI in Section 6.1.4.

4.5 Index Compression

EBI has already reached the memory lower-bound in Theorem 4.2. Meanwhile, GSI provides a lower and adjustable memory usage in practice, with a better theoretical guarantee in the power-law graph. Both our algorithms are designed for exact solutions. However, in some applications of butterfly counting, such as graph kernel analysis [40] and network measurement [37, 44], using approximation counts is sufficient. In this section, we show that if we allow a small error ratio (e.g., $\leq 10^{-6}\%$) of the butterfly counting, we can reach a much lower memory usage on real-world graphs. Two index compression methods for approximate historical butterfly counting with strong theoretical guarantees are introduced.

Single-sided Compression (SGSI). Intuitively, most butterflies in Algorithm 6 come from T_C (Line 2 to Line 3) while the actual memory usage of T_E is much larger in general. This inspires us only to maintain a small portion (e.g., $\frac{1}{1000}$) of all T_E's butterflies. We denote λ_1 as the single-sided compression ratio, which indicates we only preserve $\lambda_1 \cdot \alpha B$ butterflies, where αB is the total number of butterflies in T_E. Each butterfly is preserved with probability λ_1 independently. In other words, we modify the GSI algorithm so that when GSI inserts a 2D point into T_E, the insertion is actually performed with probability λ_1. We denote the compressed T_E by $\widetilde{T_E}$. To answer queries, we return $\widetilde{T_E}.query([t_s, \infty] \times [-\infty, t_e])/\lambda_1$ (Line 1 in Algorithm 6). This results in a compressed memory usage of $O(m\delta \log n + \lambda_1 \cdot \alpha B \log m)$.

Double-sided Compression (DGSI). SGSI only compresses on T_E, which does not affect the query efficiency. If we further compress T_C, we can reach a faster query time and even lower memory usage. Specifically, in DGSI, besides compressing T_E as in SGSI, we only maintain λ_2 portion of wedges for each group in T_C's $\mathcal{CS}$s. Each wedge is preserved with probability λ_2 independently. We denote the compressed T_C by $\widetilde{T_C}$. To answer queries, each $\widetilde{\mathcal{CS}}$ in $\widetilde{T_C}$ contributes $\binom{count}{2}/\lambda_2^2$ (Line 4 in Algorithm 6) to the total answer where *count* is the number of wedges in $\widetilde{\mathcal{CS}}$ in the projected graph.

This results in an $O(\beta \log(\lambda_2 n) + \log(\lambda_1 m))$ query time, which is more efficient than our exact algorithm, GSI.

In our technical report Part A, we prove that both compression methods are unbiased and bound their errors by the following theorems.

Theorem 4.6 (Compressing T_E). *For any historical butterfly counting query $[t_s, t_e]$ on a bipartite graph G, let B denote the correct number of butterflies in $G_{[t_s,t_e]}$ that is maintained by T_E, i.e., $B = T_E.query([t_s, \infty] \times [-\infty, t_e])$. Let B' denote the number of butterflies computed by the compressed T_E in SGSI with compression ratio λ_1, i.e., $B' = \widetilde{T_E}.query([t_s, \infty] \times [-\infty, t_e])/\lambda_1$. B' is unbiased, i.e., the expectation of B' is equal to B. The absolute error $|B - B'|$ is $O(B^{0.5} \log^{0.5}(n)\lambda_1^{-0.5})$ with high probability, where n is the number of vertices.*

Theorem 4.7 (Compressing T_C). *For any historical butterfly counting query $[t_s, t_e]$ on a bipartite graph G, let S denote the correct number of butterflies in $G_{[t_s,t_e]}$ that is maintained by T_C, i.e., the total increment of $num_{[t_s,t_e]}$ in Line 4 of Algorithm 6. Let S' denote the number of butterflies computed by the compressed $\widetilde{T_C}$ in DGSI with compression ratio λ_2. S' is unbiased, i.e., the expectation of S' is equal to S. The absolute error $|S - S'|$ is $O(S^{0.75} \ln(n)^{0.75}\lambda_2^{-1.75})$ with high probability, where n is the number of vertices.*

Theorem 4.6 directly bounds the error of SGSI. To bound the error of DGSI, we may add the error bounds for compressing the T_E part (Theorem 4.6) and compressing the T_C part (Theorem 4.7).

4.6 Handling Duplicate Edges

The key challenge when the graph has duplicate edges is that the life cycle of a wedge or a butterfly is no longer an active timestamp as defined in Definition 4.2. As we will see in our technical report Part B, the life cycle can be decomposed into several redefined active timestamps (Definition 4.3) for graphs with duplicate edges. We will prove that the decomposition does not increase the time complexity or memory usage of GSI (Lemma 4.8 and Theorem 4.9).

Definition 4.3 (Active Intervals for Graphs with Duplicate Edges). *Given a bipartite temporal graph G with duplicate edges, a subgraph P, we define the active intervals $\widetilde{\mathcal{T}}(P)$ as a tuple of timestamps of the form $[l, r_1, r_2](l \le r_1 \le r_2)$ such that P is active in the query time-window $[t_s, t_e]$ if and only if for exactly one of the timestamps $[l, r_1, r_2]$, $t_s \le l$ and $r_1 \le t_e < r_2$.*

For applying GSI to a graph with duplicate edges, we first modify $\mathcal{CS}$ such that it can answer 2D-range queries on timestamps of the form $[l, r_1, r_2]$, i.e., counting the number of $[l, r_1, r_2]$ such that $t_s \le l$ and $r_1 \le t_e \le r_2$ given the query time-window $[t_s, t_e]$. This can be done by applying the inclusive-exclusive principle on 2D ranges. Then we generate the active intervals (tuple of timestamps $[l, r_1, r_2]$) for each wedge. We feed each $[l, r_1, r_2]$ to GSI (with the modified $\mathcal{CS}$). A detailed explanation can be found in our technical report Part B. Lastly, we prove that if the size (number of $[l, r_1, r_2]$ in the tuple) of the active intervals of each wedge is bounded (Lemma 4.8), both EBI and CBI's time complexity will not be compromised.

Lemma 4.8. *For any two vertices $u, v \in V(G)$, we denote $cnt_{u,v}$ as the number of edges $(u, v, t) \in E(G)$. There exists an algorithm (Algorithm 7 in the technical report Part B) that returns its active intervals (Definition 4.3) of size $O(\min(cnt_{x,y}, cnt_{y,z}))$ for any wedge $\langle x \rightsquigarrow y \rightsquigarrow z \rangle$.*

Proof of 4.8 can be found in our technical report Part B. With this lemma, we are ready to prove that the time complexity for our algorithms will not be compromised by duplicate edges, whose proof can also be found in our technical report Part B.

Theorem 4.9 (Time complexity with Duplicate Edges). ***(i)*** *There exists a modification for EBI that can run in $O(\log m)$ time and $O(m^2)$ memory usage for bipartite temporal graphs with duplicate edges;* ***(ii)*** *There exists a modification for CBI that can run in $O(\tilde{w} \log m)$ time and $O(m\delta)$ memory usage for bipartite temporal graphs with duplicate edges.*

5 ANALYSIS ON POWER-LAW GRAPHS

• **Power-law Bipartite Graphs** Previous research [16, 42] shows that many real-world bipartite graphs follow the *power-law* distribution with γ generally between 2 to 3, similar to the *scale-free* graphs model for unipartite graphs. By leveraging the properties of the power-law degree distribution, there are many studies on analyzing algorithms based on such settings [4, 22, 53, 55]. In this section, we prove that GSI runs in $\widetilde{O}(\lambda)$ time and $O(m^{2-\delta}/\lambda^2)$ memory for some $\delta > 0$ with high probability on power-law bipartite graphs [1] with $\gamma \in (2, 3)$, in contrast to the hardness result in Section 4.1 for the general case. We use the following model for G.

Definition 5.1. *Let n_1 be the number of vertices in U. Let n_2 be the number of vertices in L. Let γ_i, Δ_i $(i = 1, 2)$ be the exponents and max degrees of two power-law distributions. A power-law bipartite graph is obtained by the following process:* ***(1)*** *For each vertex x in U, sample $d_x \in [1, \Delta_1]$ such that $\mathbb{P}[d_x = k] \propto k^{-\gamma_1}$.* ***(2)*** *For each vertex y in L, sample $d_y \in [1, \Delta_2]$ such that $\mathbb{P}\left[d_y = k\right] \propto k^{-\gamma_2}$.* ***(3)*** *Let $m = \mathbb{E}\left[\sum_{x \in U} d_x\right] = \mathbb{E}\left[\sum_{y \in L} d_y\right]$ be the expected number of edges in G.* ***(4)*** *For each pair of vertices $x \in U$, $y \in L$, sample the existence of the edge (x, y) such that $\mathbb{P}[(x, y) exists] = \frac{d_x d_y}{m}$.*

For Definition 5.1 to be well-defined, we need to choose parameters such that $\mathbb{E}\left[\sum_{x \in U} d_x\right] = \mathbb{E}\left[\sum_{y \in L} d_y\right]$, i.e., the expected sums of degrees for vertices in U and L are equal. We note that the sampled d_x and d_y values are intermediate variables of the sampling process. They are not necessarily the same as the degrees $\deg_x$ and $\deg_y$ of vertex x and y. We can interpret d_x as the expected degree of x. Based on Definition 5.1, there are two following two types of power-law bipartite graphs to model the real-world networks comprehensively.

Definition 5.2 (Double-sided Power-law Bipartite Graph). *A double-sided power-law bipartite graph is a power-law bipartite graph (Definition 5.1) satisfying $\gamma_1 \in (2, 3)$, and $\gamma_2 \in (2, 3)$.*

Definition 5.3 (Single-sided Power-law Bipartite Graph). *A single-sided power-law bipartite graph is a power-law bipartite graph (Definition 5.1) satisfying $\gamma_1 \in (2, 3)$, $\gamma_2 = 0$, and $\Delta_1 > \Delta_2$.*

Twitter exemplifies a single-sided power-law bipartite graph, where the distribution of followers per user varies significantly, often displaying a vast disparity. In contrast, the number of accounts each user follows tends to be more uniform and comparatively narrow in range. In contrast, movie-actor networks exhibit a double-sided power-law distribution. Most actors appear in just a few films, and the majority of films feature only a small number of actors. However, a select group of highly connected actors mirrors the highly-followed users on social media platforms, while films with large casts resemble “super-connected” nodes.

• **Time and Space Complexity** We calculate the expected time and space of `GSI` for each type of power-law bipartite graph.

Theorem 5.1 (`GSI` on double-sided power-law bipartite graphs). *Let G be a single-sided power-law bipartite graph. We can set α in Algorithm 5, such that the expected space usage of* `GSI` *is $O(\Delta^{6-2\gamma})$, and that the query time is $\widetilde{O}(1)$. We can also set α such that the expected space usage is $O\left(n\Delta^{3-\gamma}\right)$, and that the expected query time is $\widetilde{O}(n^2 \min(\Delta_1, \Delta_2)^{2-2\gamma})$.*

We note that for both settings of Theorem 5.1, the time and space complexities surpass the lower bound in Theorem 4.2. This shows that the historical butterfly counting problem on power-law graphs are not as hard as the general case, and that `GSI` successfully exploits the features of the power-law graphs to reduce the computation cost. The first setting has $\widetilde{O}(1)$ time complexity and $o(n^2)$ space complexity since $\Delta \le n$ and $\gamma > 2$. The second setting has $\widetilde{O}(\lambda) = \widetilde{O}(n^2 \min(\Delta_1, \Delta_2)^{2-2\gamma})$ time complexity and $o(n^2/\lambda^2)$ space complexity when $k > n^{\frac{3}{4\gamma-4}} \Delta^{\frac{3-\gamma}{4\gamma-4}}$.

Theorem 5.2 (`GSI` on single-sided power-law bipartite graphs). *Let G be a single-sided power-law bipartite graph.*

We can set α in Algorithm 5, such that the expected space usage of `GSI` *is $O\left(\Delta_2 + \left(\frac{\Delta_1^{6-\gamma_1}}{\Delta_2^5}\right)\right)$ and that the expected query time is $\widetilde{O}\left(1\right)$.*

Similar to the double-sided case, Theorem 5.2 solves the historical butterfly counting problem with a better trade-off than Theorem 4.2.

Proof sketch for Theorem 5.1 and Theorem 5.2. By Theorem 4.5, we know that the query time for `GSI` is nearly linear in the number of keys (x, z) in T_C. The space usage for `GSI`is bounded by the number of butterflies not maintained by $T_C[(x, z)]$, plus the total number of wedges in each $W[(x, z)]$ for $(x, z) \in T_C.keys()$. To bound the query time and space usage, let k be a parameter between 1 and $\Delta = \max(\Delta_1, \Delta_2)$. Let $P_{\ge k}$ be the set of unordered pairs (x, z) such that x, z are on the same side of the bipartite graph G, and that $d_x, d_z \ge k$. Let $\# \bowtie_{\ge k}$ be the number of butterflies B such that $\exists \{x, z\} \in P_{\ge k}$, $\{x, z\} \subseteq B$. Here we abuse notation and use B to mean the vertices of a butterfly B. In Line 11 of Algorithm 5, we construct $T_C[(x, z)]$s for pairs (x, z) with the largest $W[(x, z)]$s until the total number of butterflies in these largest $W[(x, z)]$s exceeds $(1 - \alpha)numB$. The rest of the butterflies will be maintained by T_E. Intuitively, for proving the efficiency of `GSI`, we would like to show that a small number of $\{x, z\}$ (those in $P_{\ge k}$) covers a large fraction ($\# \bowtie_{\ge k} / numB$) of the total number of butterflies. Formally, we can prove that for some k, `GSI` has query complexity $\widetilde{O}(|P_{\ge k}|)$ and expected space complexity. To show this, let’s consider an algorithm similar to Algorithm 5. The modified algorithm changes the condition on Line 12 from “$num \ge \alpha \cdot numB$” to “$\{x, z\} \in P_{\ge k}$”. Then T_C contains $|P_{\ge k}|$ elements (one for each $(x, z) \in P_{\ge k}$), and T_E contains $\# \bowtie_{\ge 1} - \# \bowtie_{\ge k}$ butterflies. The time complexity of the modified algorithm is $\widetilde{O}(|P_{\ge k}|)$ and that the expected space usage of it is $O\left(\# \bowtie_{\ge 1} - \# \bowtie_{\ge k} + \sum_{(x,z) \in P_{\ge k}} |T_C[(x, z)]|\right)$.

In the original `GSI` (Algorithm 5), for a fixed choice of k, we can choose α properly such that the condition on Line 12 evaluates to true for the first $|P_{\ge k}|$ iterations, i.e., after the first $|P_{\ge k}|$ iterations of the loop, num is no more than $\alpha \cdot numB$. The query time of `GSI` is bounded by $\widetilde{O}(|P_{\ge k}|)$. The space usage of `GSI` is bounded above by

Table 1: The summary of datasets, including the size of vertices (*i.e.,* $|U|$ and $|L|$) and edges (*i.e.,* $|E|$), butterfly counts (*i.e.,* B), and the γ value of power-law exponent assuming the graph’s degree follows a power-law distribution (*i.e.,* $\hat{\gamma}$). For synthetic datasets, we provide the γ value for each side (*i.e.,* γ_1 and γ_2).

Real-world dataset	$\lvert U\rvert$	$\lvert L\rvert$	$\lvert E\rvert$	B	$\hat{\gamma}$
Wikiquote (WQ)	961	640,482	776,458	5e9	4.281
Wikinews (WN)	2,200	35,979	907,499	3e11	2.654
StackOverflow (SO)	545,196	96,680	1,301,942	1e7	2.797
CiteULike (CU)	153,277	731,769	2,411,819	6e8	2.325
Bibsonomy (BS)	204,673	767,447	2,555,080	8e8	2.209
Twitter (TW)	175,214	530,418	4,664,605	1e12	2.460
Amazon (AM)	2,146,057	1,230,915	5,838,041	3e7	2.957
Edit-ru (ER)	7,816	1,266,349	8,349,235	1e13	5.301
Edit-vi (EV)	72,931	3,512,721	25,286,492	3e14	2.017
Wiktionary (WT)	66,140	5,826,113	44,788,448	1e16	1.826
Synthetic dataset	$\lvert U\rvert$	$\lvert L\rvert$	$\lvert E\rvert$	γ_1	γ_2
DoublePower-law (DPW)	25,599,878	25,599,878	100,000,000	2.1	2.1
SinglePower-law (SPW)	21,020,985	199,802	100,000,000	2.1	0

that of the modified algorithm, because the sets in W are sorted with decreasing order of sizes. Each set $W[(x, z)]$ costs $|W[(x, z)]|$ space if it is maintained in T_C and $\binom{|W[(x,z)]|}{2}$ space if it is maintained in T_E. `GSI` costs less space because it maintains larger sets in T_C, compared to the modified algorithm. In our technical report Part C, we prove time and space complexities for the modified algorithm. These will automatically translate to the same bounds for `GSI`. □

6 EXPERIMENTS

• **Datasets** We use 10 large-scale real-world datasets to evaluate our algorithm. All real-world dataset sources and more detailed statistics are available at KONECT[3]. In addition, we randomly generate 2 power-law bipartite graphs with billion edges according to the models in Section 5, where their timestamps are also uniformly sampled from $[1, |E|]$. The dataset statistics are presented in Table 2.

• **Algorithms** Our algorithms are implemented in C++ with STL used. The implementation of `BFC-VP++` is obtained from their authors. As `WPS` is an approximation method, the reported running time for `WPS` represents the time required first to reach a relative error of 10%.

• **Hyperparameter Settings** The only hyperparameter for our methods is α. In our study of space-query trade-offs (Section 6.2), we determine the appropriate α using the algorithm described in Section 4.4.1 to constrain memory usage. We employ the same algorithm for all other experiments to optimize α for maximal memory utilization.

All experiments are conducted on a Ubuntu 22.04 LTS workstation, equipped with an Intel(R) Xeon(R) Gold 6338R 2.0GHz CPU and 512GB of memory.

6.1 Efficiency

6.1.1 Query Processing. Shown in Figure 8, the efficiency of `GSI`, as well as the two baseline algorithms `BFC-VP++`, `WPS`, is compared on various datasets, with 5000 random queries.

On real-world datasets, `GSI` demonstrates a significant speedup ranging from 100× to 10000× comparing to `BFC-VP++`, 100× to 1000× comparing to `WPS`. In most datasets, `GSI` needs less than 1 seconds to answer all 5000 queries online, even when the dataset

[3]from http://konect.cc/, where the $\hat{\gamma}$ of `ER` only considers the tail degrees (i.e., γ_t) because the global value missed in the website.

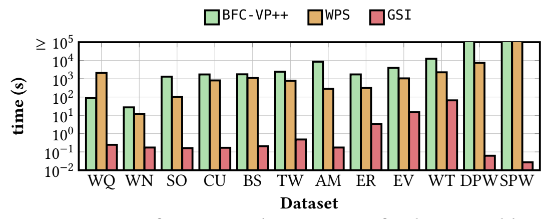


Figure 8: Time cost of 5,000 random queries for historical butterfly counting on datasets, where all algorithms use sequential versions.

Table 2: The index construction time for GSI in Figure 8.

Dataset	WQ	WN	SO	CU	BS	TW
time (s)	74	6	9	129	207	217
Dataset	**AM**	**ER**	**EV**	**WT**	**DPW**	**SPW**
time (s)	39	3,654	3,842	3,964	2,566	341

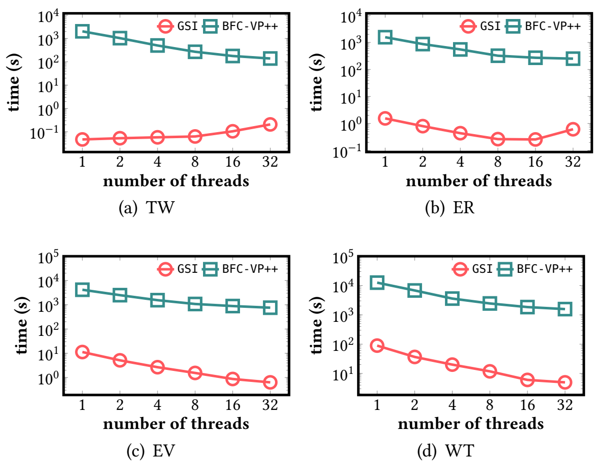


Figure 9: Evaluating the parallelization: time cost for 5,000 random queries against the number of threads.

has around 4×10^6 vertices (AM). In general, the query time is proportional to the number of edges. For the largest dataset WT, which has around 6×10^6 vertices and 4.5×10^8 edges, all queries can be answered within 10^2 seconds, which means less than 0.1 seconds for each query in average. This shows the consistency and efficiency of our algorithms under very large real-world datasets.

On the synthetic dataset, we generated, our algorithm shows an extremely large speedup greater than 100000×. Especially on SPW, while both BFC-VP++ and WPS cannot process all queries within 10^5 seconds, our algorithms produce answers within 10^{-1} seconds.

6.1.2 Index Construction. The reported time for index construction is reported in Table 2. For most datasets, even considering the time of index construction, GSI's performance outperforms the two baselines, indicating that the index construction performance of GSI is both reasonable and practical.

6.1.3 Empirical Study on Power-law Graphs. In this section, we further analyze the performance of GSI and baselines on the power-law graphs. In addition to two synthetic graphs in Table 2, we generate power-law graphs with various γ_1 and γ_2 between $(2, 3)$ and uniformly sampled timestamps. Specifically, to ensure that baselines can finish in practical time (*i.e.,* less than $\leq 10^5$ seconds), we reduce the number of edges in these graphs to 10^7 instead of 10^8. in DPW

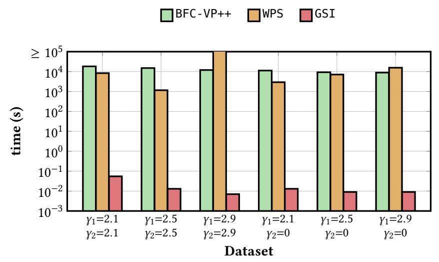


Figure 10: Time cost of 5,000 random queries for historical butterfly counting on different power-law bipartite graphs, where all algorithms use sequential versions.

and SPW. Shown in Figure 10, both BFC-VP++ and WPS require at least 10^3 seconds in all synthetic data we generate while our algorithm can process all 5,000 queries within 10^{-1} seconds. This gap is even more significant compared to results on real-world data. Combined with the theoretical analysis in section 5, GSI has outstanding performance both theoretically and practically on Power-law graphs. This further reveals the advantage that GSI admits and makes full use of additional properties on graphs.

6.1.4 Parallelization. In addition, we compare GSI's parallelized querying with BFC-VP++'s parallelized version. In Figure 9, we test the running time for 5,000 queries with various numbers of threads from 1, 2, 4, 8, 16, and 32. In two large-scale datasets EV and WT, our algorithm obtains higher parallelism compared to the baseline, which has spent efforts in optimizing for higher parallelism. On the other hand, in two other two large-scale datasets TW and ER, despite its fast runtime, our algorithm's performance does not always improve as the number of threads increases, which is not surprising and is under our expectations. This is because every historical query is essentially querying on multiple CSs and summing up the answer. In the parallel version, we allocate threads to these data structures to increase efficiency. However, given a fixed memory limitation, GSI may create less CSs if the number of motifs is relatively small. In such a circumstance, if the number of threads approaches or exceeds the number of data structures we built in construction, the runtime may increase. In general, our parallelized GSI still obtains a faster runtime and, most likely, better parallelism on very large-scale datasets.

6.2 Space-Time Trade-Offs

Since GSI is a flexible algorithm that strikes to balance the trade-off between space and time, we further study its performance under different memory limitations. Shown in Figure 11, the query time can be efficiently reduced if provided more memory space: When the applicable memory is enlarged to 10×, the query time is decreased to $\frac{1}{10}\times$ to $\frac{1}{100}\times$. Furthermore, it is shown that even when the memory space is very limited, GSI can still obtain a good query efficiency. For example, on WT, only given 40 Gigabytes memory, both the index-construction time and query time are less than 600 seconds, which still holds a significant advantage compared to our baseline in fig. 8. In all, this study further reveals GSI's flexibility on the trade-off between space and time. In practice, it can fit different circumstances while providing high-efficiency performance.

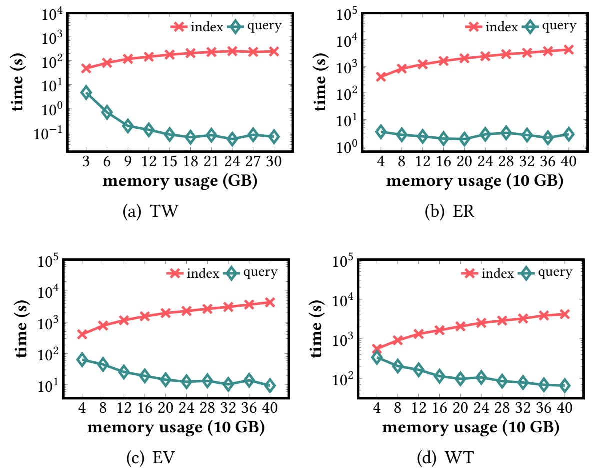


Figure 11: Space-query trade-offs study

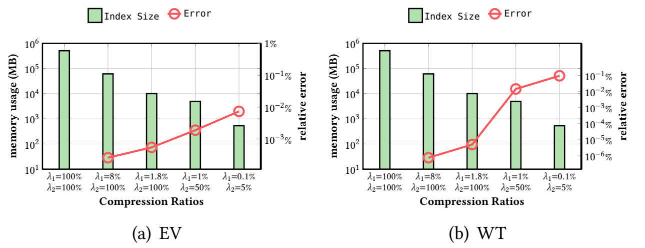


Figure 12: Evaluating memory usage and relative error in various compression ratio settings. Note that both our compression algorithms are not affect the query effciency.

6.3 Study on Compressed Indexes

In addition, shown in Figure 12, we evaluate the index compression on two large-scale datasets EV and WT. With the single-sided compression algorithm SGSI, we are able to guarantee a relative error $\leq 10^{-3}$ while compressing memory usage to $\frac{1}{50}$ (i.e., 500G $\longrightarrow$ 1G). With the double-sided compression algorithm DGSI, we achieve $\leq 10^{-1}$ relative error while compressing memory for around 1000 times (500G $\longrightarrow$ 500MB). This experiment validates the effectiveness of our GSI algorithm with index compression based on sampling. Moreover, it empirically provides a trade-off between memory and accuracy, i.e., if we need a higher accuracy guarantee, we use SGSI. To compress the memory further, we may use DGSI instead.

6.4 Case Study

To begin with, we acquired the author-publication datasets [4], which contains 10,419,221 author-publication records before 2016. Based on that, we build an author-publication bipartite temporal graph containing 14,223,972 vertices and 10,419,216 edges. Consider the projected graph $G_{[t_s,t_e]}$, its corresponding BCC should be $\frac{4\times \bowtie_{[t_s,t_e]}}{\ltimes_{[t_s,t_e]}}$, where $\bowtie_{[t_s,t_e]}$ is the butterfly count in the projected graph $G_{[t_s,t_e]}$ and $\ltimes_{[t_s,t_e]}$ is the number of 3-paths in $G_{[t_s,t_e]}$.

In this study, we investigate the trend of the BCC over various two-year time windows. To achieve that, we need to efficiently

[4]https://github.com/THUDM/citation-prediction

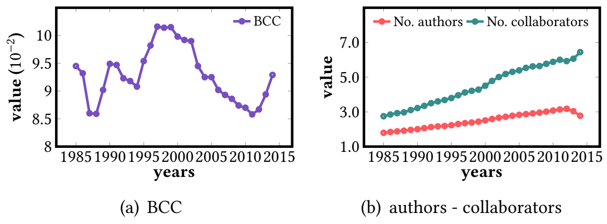


Figure 13: Case study (1): the trend of closeness in global research collaboration

query several historical butterfly counts by GSI. The result is shown in Figure 13 (a), where we set the length of the time windows as two years (e.g., 1995 - 1996). Surprisingly, the value of BCCs doesn't consistently rise alongside the research community's overall trend toward collaboration. This implies that the research community might not always achieve greater cohesiveness through collaboration. In contrast, in Figure 13 (a), BCCs exhibit an initial general increase (1985 - 2000) followed by a decrease (2000 - 2015). We delve deeper into the underlying reasons for these trends and analyze the changing pattern of **(1)** *the average number of publications per author within each time-window* and **(2)** *the average number of unique collaborators per author within each time-window*, shown in Figure 13 (b). As the average number of publications generally rises from 1985 to 2015, it suggests a higher probability of collaboration, consequently increasing the butterfly counts in the author-publication bipartite temporal graph. Nevertheless, a greater likelihood of collaboration does not necessarily equate to more cohesive collaboration. As depicted in Figure 13 (b), there is a notable increase in the average number of unique collaborators after 2000, indicating a rise in 3-paths counts within the graph, which may be an important factor causing BCCs decreases again from 2000 to 2010. Entering 2010, BCCs come across another rising trend. There are many possible reasons, one of which could be the significant increase in total publications.

7 CONCLUSIONS

This paper introduces the historical butterfly counting problem on temporal bipartite graphs. We propose a graph structure-aware indexing approach to facilitate efficient query processing with a tunable balance between query time and memory cost. Besides, we theoretically prove that our approach is especially advantageous on power-law graphs, by breaking the conventional complexity barrier associated with general graphs. To further mitigate the index overhead, we devised a high-quality approximation algorithm that leverages a compressed index structure, thereby enhancing overall efficiency. Extensive empirical tests show that our algorithm is up to five orders of magnitude faster than existing algorithms with controllable memory costs.

ACKNOWLEDGMENTS

Chenhao Ma was partially supported by NSFC under Grant 62302421, Basic and Applied Basic Research Fund in Guangdong Province under Grant 2023A1515011280, 2025A1515010439, and the Guangdong Provincial Key Laboratory of Big Data Computing, The Chinese University of Hong Kong, Shenzhen. We thank Xinwen Zhang for her valuable advice on this project.

REFERENCES

[1] William Aiello, Fan Chung, and Linyuan Lu. 2000. A random graph model for massive graphs. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*. 171–180.

[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), 8.

[4] Paweł Brach, Marek Cygan, Jakub Łącki, and Piotr Sankowski. 2016. Algorithmic complexity of power law networks. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 1306–1325.

[5] Xinwei Cai, Xiangyu Ke, Kai Wang, Lu Chen, Tianming Zhang, Qing Liu, and Yunjun Gao. 2024. Efficient Temporal Butterfly Counting and Enumeration on Temporal Bipartite Graphs. *Proc. VLDB Endow.* 17, 4 (mar 2024), 657–670. https://doi.org/10.14778/3636218.3636223

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

[7] Kaiyu Chen, Dong Wen, Wenjie Zhang, Ying Zhang, Xiaoyang Wang, and Xuemin Lin. [n. d.]. Querying Structural Diversity in Streaming Graphs. ([n. d.]).

[8] 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* (2021).

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

[10] Shiyuan Deng, Shangqi Lu, and Yufei Tao. 2023. Space-Query Tradeoffs in Range Subgraph Counting and Listing. In *26th International Conference on Database Theory (ICDT 2023)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik.

[11] 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.

[12] 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.

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

[14] Isaac Goldstein, Tsvi Kopelowitz, Moshe Lewenstein, and Ely Porat. 2017. Conditional Lower Bounds for Space/Time Tradeoffs. In *Algorithms and Data Structures*, Faith Ellen, Antonina Kolokolova, and Jörg-Rüdiger Sack (Eds.). Springer International Publishing, Cham, 421–436.

[15] Isaac Goldstein, Moshe Lewenstein, and Ely Porat. 2019. On the Hardness of Set Disjointness and Set Intersection with Bounded Universe. In *30th International Symposium on Algorithms and Computation (ISAAC 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 149)*, Pinyan Lu and Guochuan Zhang (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 7:1–7:22. https://doi.org/10.4230/LIPIcs.ISAAC.2019.7

[16] Jean-Loup Guillaume and Matthieu Latapy. 2006. Bipartite graphs as models of complex networks. *Physica A: Statistical Mechanics and its Applications* 371, 2 (2006), 795–813.

[17] Saket Gurukar, Sayan Ranu, and Balaraman Ravindran. 2015. Commit: A scalable approach to mining communication motifs from dynamic networks. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 475–489.

[18] Xiangnan He, Kuan Deng, Xiang Wang, Yan Li, Yongdong Zhang, and Meng Wang. 2020. Lightgcn: Simplifying and powering graph convolution network for recommendation. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*. 639–648.

[19] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural collaborative filtering. In *Proceedings of the 26th international conference on world wide web*. 173–182.

[20] Madhav Jha, Comandur Seshadhri, and Ali Pinar. 2013. A space efficient streaming algorithm for triangle counting using the birthday paradox. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. 589–597.

[21] 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.

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

[23] 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. *IEEE Transactions on Knowledge and Data Engineering* 34, 12 (2021), 5621–5635.

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

[25] Youhuan Li, Lei Zou, M Tamer Özsu, and Dongyan Zhao. 2019. Time constrained continuous subgraph search over streaming graphs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1082–1093.

[26] Pedro G Lind, Marta C Gonzalez, and Hans J Herrmann. 2005. Cycles and clustering in bipartite networks. *Physical review E* 72, 5 (2005), 056127.

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

[28] Paul Liu, Austin R Benson, and Moses Charikar. 2019. Sampling methods for counting temporal motifs. In *Proceedings of the twelfth ACM international conference on web search and data mining*. 294–302.

[29] Penghang Liu, Valerio Guarrasi, and Ahmet Erdem Sarıyüce. 2021. Temporal network motifs: Models, limitations, evaluation. *IEEE Transactions on Knowledge and Data Engineering* 35, 1 (2021), 945–957.

[30] Giorgio Locicero, Giovanni Micale, Alfredo Pulvirenti, and Alfredo Ferro. 2021. TemporalRI: a subgraph isomorphism algorithm for temporal networks. In *Complex Networks & Their Applications IX: Volume 2, Proceedings of the Ninth International Conference on Complex Networks and Their Applications COMPLEX NETWORKS 2020*. Springer, 675–687.

[31] 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.

[32] Tore Opsahl. 2013. Triadic closure in two-mode networks: Redefining the global and local clustering coefficients. *Social networks* 35, 2 (2013), 159–167.

[33] 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.

[34] 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.

[35] Georgios A Pavlopoulos, Panagiota I Kontou, Athanasia Pavlopoulou, Costas Bouyioukos, Evripides Markou, and Pantelis G Bagos. 2018. Bipartite graphs in systems biology and medicine: a survey of methods and applications. *GigaScience* 7, 4 (2018), giy014.

[36] Ursula Redmond and Pádraig Cunningham. 2013. Temporal subgraph isomorphism. In *Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*. 1451–1452.

[37] 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.

[38] Seyed-Vahid Sanei-Mehri, Yu Zhang, Ahmet Erdem Sariyüce, and Srikanta Tirthapura. 2019. Fleet: Butterfly estimation from a bipartite graph stream. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. 1201–1210.

[39] Ilie Sarpe and Fabio Vandin. 2021. OdeN: simultaneous approximation of multiple motif counts in large temporal networks. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 1568–1577.

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

[41] Jessica Shi and Julian Shun. 2022. Parallel algorithms for butterfly computations. In *Massive Graph Analytics*. Chapman and Hall/CRC, 287–330.

[42] Demival Vasques Filho and Dion RJ O'Neale. 2018. Degree distributions of bipartite networks and their projections. *Physical Review E* 98, 2 (2018), 022307.

[43] Jun Wang, Arjen P De Vries, and Marcel JT Reinders. 2006. Unifying user-based and item-based collaborative filtering approaches by similarity fusion. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*. 501–508.

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

[45] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2019. Vertex Priority Based Butterfly Counting for Large-scale Bipartite Networks. *PVLDB* (2019).

[46] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2020. Efficient bitruss decomposition for large-scale bipartite graphs. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 661–672.

[47] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2023. Accelerated butterfly counting with vertex priority on bipartite graphs. *The VLDB Journal* 32, 2 (2023), 257–281.

[48] Zhibin Wang, Longbin Lai, Yixue Liu, Bing Shui, Chen Tian, and Sheng Zhong. 2023. I/O-Efficient Butterfly Counting at Scale. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–27.

[49] Dong Wen, Bohua Yang, Ying Zhang, Lu Qin, Dawei Cheng, and Wenjie Zhang. 2022. Span-reachability querying in large temporal graphs. *The VLDB Journal* (2022), 1–19.

[50] Shiwen Wu, Fei Sun, Wentao Zhang, Xu Xie, and Bin Cui. 2022. Graph neural networks in recommender systems: a survey. *Comput. Surveys* 55, 5 (2022), 1–37.

[51] Yifei Xia, Feng Zhang, Qingyu Xu, Mingde Zhang, Zhiming Yao, Lv Lu, Xiaoyong Du, Dong Deng, Bingsheng He, and Siqi Ma. 2024. GPU-based butterfly counting. *The VLDB Journal* (2024), 1–25.

[52] Haoxuan Xie, Yixiang Fang, Yuyang Xia, Wensheng Luo, and Chenhao Ma. 2023. On querying connected components in large temporal graphs. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.

[53] Yixing Yang, Yixiang Fang, Maria E Orlowska, Wenjie Zhang, and Xuemin Lin. 2021. Efficient bi-triangle counting for large bipartite networks. *Proceedings of the VLDB Endowment* 14, 6 (2021), 984–996.

[54] Michael Yu, Dong Wen, Lu Qin, Ying Zhang, Wenjie Zhang, and Xuemin Lin. 2021. On querying historical k-cores. *Proceedings of the VLDB Endowment* (2021).

[55] Fangyuan Zhang, Dechuang Chen, Sibo Wang, Yin Yang, and Junhao Gan. 2023. Scalable Approximate Butterfly and Bi-triangle Counting for Large Bipartite Networks. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–26.

[56] Alexander Zhou, Yue Wang, and Lei Chen. 2021. Butterfly counting on uncertain bipartite graphs. *Proceedings of the VLDB Endowment* 15, 2 (2021), 211–223.

[57] Zhaonian Zou. 2016. Bitruss decomposition of bipartite graphs. In *International conference on database systems for advanced applications*. Springer, 218–233.