University Library

A gateway to Melbourne's research publications

Minerva Access is the Institutional Repository of The University of Melbourne

Author/s:
Lin, F;Ruan, B;Gan, J;Li, L

Title:
Efficient Bitruss Decomposition without Butterfly Enumeration

Date:
2025

Citation:
Lin, F., Ruan, B., Gan, J. & Li, L. (2025). Efficient Bitruss Decomposition without Butterfly Enumeration. KDD '25: Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2, 2, pp.1718-1728. Association for Computing Machinery. https://doi.org/10.1145/3711896.3736921.

Persistent Link:
https://hdl.handle.net/11343/361935

License:
CC-BY

Efficient Bitruss Decomposition without Butterfly Enumeration

Fengnian Lin
The University of Melbourne
Melbourne, VIC, Australia
fengnian.lin@student.unimelb.edu.au

Boyu Ruan
The Hong Kong University of Science and Technology
Hong Kong, China
boyuruan@ust.hk

Junhao Gan
The University of Melbourne
Melbourne, VIC, Australia
junhao.gan@unimelb.edu.au

Lei Li
The Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, Guangdong, China
thorli@ust.hk

Abstract

Mining cohesive subgraphs in bipartite graphs is of great importance to various real-world applications such as recommendation in e-commercial systems and fraud detections in social networks. In this paper, we study the problem of *Bitruss Decomposition* of a given bipartite graph G. The goal is to compute, for each edge e, the largest value of k such that e is still contained in a k-bitruss. Here, a k-bitruss is a maximal subgraph of G such that every edge of it is contained in at least k butterflies (i.e, $(2,2)$-cliques) in the subgraph. All the previous state-of-the-art solutions are based on the well-known *Peeling Process* framework and have to "touch" every butterfly in the graph G for at least once, causing a $O(\bowtie_G)$ cost, where $\bowtie_G$ is the number of butterflies in G. In the worst case, $\bowtie_G$ can be as large as $O(m^2)$, where m is the number of edges in G. We propose the first algorithm called *BiT-DT*, whose running time is bounded by $O(m^{1.5}\log m)$, significantly improving the state-of-the-art bound by roughly a factor of $O(\sqrt{m})$. The crucial idea of our algorithm is an efficient approach to identify the edges to be peeled next in the Peeling Process without maintaining a precise butterfly support for each edge. To further enhance the practical performance of our *BiT-DT* algorithm, we propose a heuristic to peel the edges in batch. Extensive experimental results on ten real-world datasets show that our proposed algorithm outperforms the state-of-the-art baselines by up to an order of magnitude in terms of running time.

CCS Concepts

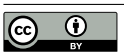
• **Theory of computation → Graph algorithms analysis**.

Keywords

Bipartite Graph, Bitruss Decomposition, Distributed Tracking

ACM Reference Format:
Fengnian Lin, Boyu Ruan, Junhao Gan, and Lei Li. 2025. Efficient Bitruss Decomposition without Butterfly Enumeration. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2 (KDD '25), August 3–7, 2025, Toronto, ON, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3711896.3736921

KDD Availability Link:
The source code of this paper has been made publicly available at https://doi.org/10.5281/zenodo.15515912.



This work is licensed under a Creative Commons Attribution 4.0 International License.
KDD '25, Toronto, ON, Canada
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1454-2/2025/08
https://doi.org/10.1145/3711896.3736921

1 Introduction

The importance of graph data has been witnessed in the past decades: a wide range of data in enormous real-world applications can be naturally modelled with graphs. Among various types of graphs, *bipartite* graphs are of particular usefulness to model the data representing "user-item"-like relations, such as item-purchase records in Amazon, authorships in DBLP, film ratings in Netflix, etc. Informally, a bipartite graph contains two *disjoint* sets of vertices: one set represents *users* and the other represents *items* (here, the notions of users and items are just two *conceptual* node types). Every edge in a bipartite graph can *only* connect and represent the relation between a user and an item. In other words, there is no edge between two users or between two items.

Butterfly. Patterns in a bipartite graph often indicate certain similarity between the users or the items. For example, both user u_1 and user u_2 had bought the same two items x and y in Amazon. In some sense, this pattern reflects the fact that u_1 and u_2 may have certain similarity in their preferences. This kind of information is crucial for recommending items to potential buyers (i.e., users). To see this, suppose that now u_2 bought another item z that u_1 has not yet to buy. By the aforementioned hypothesis that u_1 and u_2 have similar preferences, u_1 may also be interested in item z, and hence, it is worth recommending z to this user. This simple idea is actually the basis of *collaborative filtering* [6, 16], an important technique that has been widely adopted in various commercial recommendation systems. Also, this pattern is useful in detecting frauds in a social network [2, 10, 11], and identifying cohesive research groups in author-publication networks [14].

The above pattern that both u_1 and u_2 have bought items x and y is called a *butterfly* [13, 18, 19] in the context of bipartite graphs, which is the smallest non-trivial *biclique* (i.e., a $(2,2)$-clique). Analogous to *triangles*, the smallest non-trivial cliques in general (i.e., unipartite) graphs, butterflies are of fundamental importance in both theoretical study and applications (e.g., recommendation systems) for bipartite graphs. In unipartite graphs, k-truss decomposition [4, 12, 17, 22] has been studied extensively to identify k-trusses,

where each edge in a k-truss is contained in at least k triangles. Similar to k-truss, k-bitruss is defined for bipartite graphs.

k-Bitruss. While butterflies are important basic patterns, the edges in them may not always be sufficient to reflect the preference similarity between users. Nonetheless, the more butterflies sharing an edge, the more reliable information this edge provides. Motivated by this intuition, the notion of *k-bitruss* is proposed (for some integer parameter k), which is defined as a *maximal* subgraph of a given bipartite graph such that every edge in this sub-graph is shared by at least k butterflies (in the sub-graph). In some sense, users in a k-bitruss would have stronger similarity between each other. With the larger value of k, the more reliable patterns could be found, while possibly the smaller size the k-bitruss has. Thus, a root question turns out: *How do we set the value of k to achieve the "best" trade-off?* The answer to this question is unfortunately hard to tell, not only because the value of k is notoriously difficult to set, but also because in different application contexts, the most appropriate values for k can be varied even in the same bipartite graph. As a result, one may need to try computing the k-bitruss (from scratch) with multiple different values of k and pick the best one among them. However, the computational cost of such a process can be expensive. Therefore, it is of great usefulness to *pre-process* the (bipartite) graph into certain index, with which, for any given k, the k-bitruss can be returned efficiently.

Bitruss Decomposition. The *Bitruss Decomposition* [14, 24] is known to be a desired index for this purpose. Essentially, the bitruss decomposition of a bipartite graph is just a *sorted list* of the edges by their *bitruss numbers*, where the bitruss number of an edge is the *largest* value of k such that this edge can still stay in a k-bitruss. By definition, it is easy to know that the k-bitruss consists of all the edges whose bitruss numbers are at least k. Hence, the k-bitruss can be obtained simply by scanning the bitruss decomposition (i.e., the sorted edge list) and then returning all those edges with bitruss numbers at least k; thus, the running time cost is linear to the number of edges in the k-bitruss.

The Problem. In this paper, we study the problem of *Bitruss Decomposition*; the goal is to compute the bitruss number for each edge in the given bipartite graph. Once the bitruss numbers are computed, the concrete decomposition can be obtained by sorting the edges.

Existing Algorithms. *WingDec* [14], *BiT-BU* [20], and *PBNG* [8] are three state-of-the-art algorithms[1] for Bitruss Decomposition. All of these algorithms are based on the well-known *Peeling Process Framework*. As more details will be discussed in Section 3, these algorithms have to "touch" each butterfly in the given bipartite graph G at least once. As a result, their running time complexities are inevitably $\Omega(\bowtie_G)$, where $\bowtie_G$ is the total number of butterflies in G and it can be as large as $O(m^2)$ in the worst case. Unfortunately, none of the existing algorithms can overcome such a $\Omega(\bowtie_G)$-barrier.

Our Contributions. We make the following contributions.

- We propose the first algorithm, called *BiT-DT*, for Bitruss Decomposition which breaks the $\Omega(\bowtie_G)$-barrier. The worst-case running time of our algorithm is bounded by $O(\alpha(m) \cdot m \cdot \log m) = O(m^{1.5} \cdot \log m)$ improving the $O(m^2)$ bound by almost a factor of $O(\sqrt{m})$, where $\alpha(m)$ is the arboricity of the graph.
- Our *BiT-DT* algorithm also works under the *Peeling Process Framework*. A key design in *BiT-DT* is a clever approach to identify those edges to be peeled next with a somehow unrelated technique, called *Distributed Tracking* [5], which is designed for solving a tracking problem in a distributed environment. We construct a DT instance for each edge to track the moment when it should be peeled and propose a novel algorithm to organize all the DT instances with shared counters to reduce the overall running time. We believe that this novel technique is of independent interest for improving other peeling-based algorithms for solving other problems.
- In addition to the theoretical result, we further design an effective batch-peeling heuristic to improve the *practical performance* of our algorithm, while still achieving the same theoretical running time bound.
- We conduct extensive experiments on ten real-world datasets. The results show that our proposed algorithm outperforms the state-of-the-art baselines by up to an order of magnitude in terms of running time efficiency.

Organization. Section 2 presents the problem definition. Section 3 introduces the preliminaries, including the *Peeling Process Framework* and *Distributed Tracking* Problem. The *BiT-DT* algorithm is presented in Section 4. Section 5 describes the optimization technique of *Batch Edge Peeling*. Section 6 reports the experimental results. Section 7 provides the conclusion of the paper.

2 Problem Formulation

Basic Definitions. Denote by $G = \langle A \cup B, E \rangle$ an undirected *bipartite* graph, where: (i) A and B are two *disjoint* node sets on the different sides of G; (ii) $V = A \cup B$ is the set of all nodes; and (iii) E is the edge set satisfying: every edge $(u, w) \in E$ must be between A and B, namely, either $u \in A$ and $w \in B$, or $u \in B$ and $w \in A$. Furthermore, we use n and m respectively to denote the numbers of nodes and edges in G, namely, $n = |V|$ and $m = |E|$.

For each node $u \in V$, define $N(u) = \{v \in V | (u, v) \in E\}$ as u's *neighborhood*, and $d(u) = |N(u)|$ as the *degree* of u. Every node $u \in V$ is associated with a unique ID which is an integer in the range of $[1, n]$. Without loss of generality, we assume that the ID's of the nodes in A are all *smaller* than *any* ID's of the nodes in B.

Butterfly and Butterfly Support. A *butterfly* is a $(2, 2)$-clique. Consider the *induced* sub-graph $G' \subseteq G$ of an edge set $E' \subseteq E$. For each edge $e \in E'$, the *butterfly support* of e in G', denoted by $\bowtie_{G'}(e)$, is the number of (distinct) butterflies in G' containing e.

k-Bitruss. Given an integer k such that $0 \leq k \leq m-1$, the k-Bitruss is the induced sub-graph G_k of an edge set $E_k \subseteq E$ satisfying the following two conditions:

(1) Butterfly Support Requirement: for every edge $e \in E_k$, the butterfly support in G_k is at least k, i.e., $\bowtie_{G_k}(e) \geq k$;

[1] *ParButterfly* [15] and *SC-HBD* [21] are two algorithms specifically tailored for parallel environments; however, they do not deliver noticeable speedups with a single thread, and are therefore omitted from our comparisons in this paper.

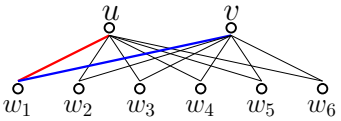


Figure 1: A bloom B of size $L(B) = 6$, where (u, w_i) and (w_i, v) (for $i = 1, 2, \ldots, 6$) are twin edges. There are $\binom{L(B)}{2}$ butterflies in B. Each edge is contained in $(L(B) - 1) = 5$ butterflies in B.

(2) Maximality: there does *not* exist any edge set E_k' such that: (i) $E_k \subset E_k' \subseteq E$ and (ii) the induced sub-graph of E_k' satisfies the butterfly support requirement with respect to k.

By the above definition, *for any given k, the k-bitruss in G is unique.*

Bitruss Number. The *bitruss number* of an edge $e \in E$, denoted by $b(e)$, is the *largest* value of k such that e is still in a k-bitruss.

Definition 2.1 (**Bitruss Decomposition**). Given an undirected bipartite graph $G = \langle A \cup B, E \rangle$, the goal of the *Bitruss Decomposition* problem is to compute the bitruss number $b(e)$ for each edge $e \in E$.

A concrete bitruss decomposition of G is a non-increasing sorted list of the edges by their bitruss numbers $b(e)$, with which the k-bitruss G_k, for any k, can be returned in $O(|E_k|)$ time by reporting all the edges with $b(e) \geq k$, where E_k is the edge set of G_k.

3 Preliminaries

3.1 Background Knowledge

We introduce some crucial concepts and useful background knowledge about the existing results.

Node Priority. For any two distinct nodes $u, v \in V$, we say u has *higher* priority than v, denoted by $u \succ v$ (or equivalently, $v \prec u$), *if and only if* either (i) $d(u) > d(v)$, or (ii) $d(u) = d(v)$ and $u.id > v.id$.

Wedge and Prioritized Wedge. A *wedge*, denoted by $\langle u, w, v \rangle$, is defined as the two-edge path from u to v through w, where $(u, w), (w, v) \in E$. Moreover, u is called *the starting node*, w *the middle node* and v *the ending node* of the wedge $\langle u, w, v \rangle$. A wedge $\langle u, w, v \rangle$ is *prioritized* if $u \succ w$ and $u \succ v$.

Bloom. A bloom B of two nodes u and v from the *same side* is the induced sub-graph of *all* the prioritized wedges starting from u and ending at v in G. Define the number of these prioritized wedges, denoted by $L(B)$, as the *size* of B. Readers should distinguish the concepts of the size of B and the number of edges in B. Note that a bloom B of size $L(B)$ has $2 \cdot L(B)$ edges. If $L(B) \geq 2$, B is a $(2, L(B))$-clique and called a *non-trivial* bloom. In this paper, we consider non-trivial blooms only, and thus, when the context is clear, we simply use "bloom" to mean "non-trivial bloom". See Figure 1 for an example.

Twin Edge in Bloom. Consider a bloom B of two nodes u and v in the two-node side of B; for each node w in the $L(B)$-node side, the edges (u, w) and (w, v) are *twin edge* of each other. It can be verified that a pair of twin edges, e.g., (u, w) and (w, v), share $(L(B) - 1)$ butterflies in B as they constitute a butterfly with any other node w' in the $L(B)$-node side. See an example in Figure 1, where the red edge (u, w_1) and the blue edge (w_1, v) are twin edges to each other, and they form a butterfly with node w_i for all $i = 2, \ldots, 6$. Moreover, the following two facts are useful:

Fact 1. *Consider a bloom B; there are $\binom{L(B)}{2}$ butterflies; and each edge in B is contained in $(L(B) - 1)$ of these butterflies.*

Fact 2 ([19]). *Each butterfly in G is in* one and exactly one *bloom.*

Butterfly Support Computation. As we will see shortly, one important step in the *Peeling Process* framework for Bitruss Decomposition is to compute the butterfly support of each edge in G. This task is called *Butterfly Support Computation*. A state-of-the-art algorithm for butterfly support computation, the *BFC-VP* algorithm [19], is designed based on Facts 1 and 2. According to these facts, for an edge e, denoted the set of all the blooms containing e by $\mathcal{B}(e)$, the butterfly support of e can be calculated as:

$$\bowtie_G(e) = \sum_{B \in \mathcal{B}(e)} (L(B) - 1). \tag{1}$$

The *BFC-VP* algorithm *conceptually* constructs all the (non-trivial) blooms by enumerating all the prioritized wedges in G. Then it computes the butterfly support of each edge $e \in E$ by Equation (1). The overall running time complexity of the *BFC-VP* algorithm is bounded by the total size of all the blooms, which is, in turn, bounded by the total number of prioritized wedges:

$$O\left(\sum_{(u,w) \in E \wedge u \succ w} d(w)\right) = O\left(\sum_{(u,w) \in E} \min\{d(u), d(w)\}\right) = O(m^{1.5}).$$

More discussions on the comparisons among the above bounds can be found in Appendix A.

3.2 The Peeling Process Framework

The *WingDec* [14], *BiT-BU* [20] and *PBNG* [8] are three state-of-the-art algorithms for Bitruss Decomposition. While *WingDec* and *BiT-BU* are under the same *Peeling Process Framework* which is introduced next, *PBNG* adopts this framework with further heuristics. The basic idea of the Peeling Process Framework is to repeatedly *peel* the edge from the *current* graph with the "smallest" butterfly support and assign a bitruss number to this edge. This peeling process continues until the graph becomes empty. Below are the main steps of the Peeling Process Framework:

- **Step 1. Butterfly Support Computation:** Compute the butterfly support with respect to G, $\bowtie_G(e)$, for each edge $e \in E$;
- **Step 2.** Initialize the *current* graph $G_{\text{cur}} \leftarrow G$, and the *current* global bitruss number $b_{\text{cur}} \leftarrow 0$ which is the bitruss number of those edges peeled from the current graph G_{cur};
- **Step 3. The Peeling Process:** While there exists an edge e in G_{cur} such that $\bowtie_{G_{\text{cur}}}(e) \leq b_{\text{cur}}$, repeat the following operations:
  - Assign b_{cur} as the bitruss number of e, i.e., $b(e) \leftarrow b_{\text{cur}}$;
  - Remove e from G_{cur};
- **Step 4.** If G_{cur} is *non-empty*, that means, there still are edges in G_{cur} but none of them satisfies $\bowtie_{G_{\text{cur}}}(e) \leq b_{\text{cur}}$,
  - $b_{\text{cur}} \leftarrow b_{\text{cur}} + 1$;
  - Repeat from Step 3;
- **Step 5.** If G_{cur} is empty, return;

The computational bottleneck of the above framework is in the Peeling Process (Step 3). The implementations of the Peeling Process are the key differences among the state-of-the-art algorithms, and hence, result in different running time complexities.

3.3 State-of-the-Art Algorithms

To identify the edges to be peeled next, the state-of-the-art algorithms *WingDec*, *BiT-BU* and *PBNG* maintain the *precise* butterfly support $\bowtie_{G_{\text{cur}}}(e)$ for each edge e with respect to the current graph G_{cur}. They differ only in how to maintain the butterfly supports.

The *WingDec* Algorithm. When an edge e is peeled, *WingDec* enumerates all the butterflies in the current graph G_{cur} containing e *on the fly*. For each of these butterflies, it decreases the butterfly support by one for every *affected* edge in the butterfly. The overall running time of the *WingDec* algorithm is bounded by $O(\sum_{(u,w)\in E}\sum_{v\in N(u)\setminus\{w\}}\min\{d(v),d(w)\})$ in expectation. Recall that this bound is at least $\Omega(\bowtie_G)$, as the *WingDec* algorithm touches each butterfly at least once in the peeling process.

The *BiT-BU* Algorithm. In contrast, the *Bit-BU* algorithm *materializes* all the blooms, and enumerates butterflies with the blooms. Specifically, when an edge e is peeled from the current graph G_{cur}, *BiT-BU* works as follows. For each bloom $B \in \mathcal{B}(e)$,

- for e's twin edge e_{twin} in B, decrease $\bowtie_{G_{\text{cur}}}(e_{\text{twin}})$ by $L(B)-1$;
- for each of the other edges $e' \in B$, decrease $\bowtie_{G_{\text{cur}}}(e')$ by 1;
- remove both e and e_{twin} from B (and hence, the size $L(B)$ is decreased by 1).

For example, as shown in Figure 1, when the red edge (u,w_1) is peeled from B, the butterfly support of its twin edge (w_2,v), the blue edge, is decreased by $(L(B)-1=5)$ because *all* the butterflies containing (w_2,v) in the bloom B have been destroyed. However, for any other edges in B, its butterfly support is decreased only by one (as only the butterfly formed with (u,w_1) is destroyed).

Overall Running Time. While *BiT-BU* can enumerate the butterflies more efficiently than *WingDec* does, it still needs to touch each butterfly at least once in the worst case. The overall running time of *BiT-BU* is bounded by $O(\sum_{(u,w)\in E}\min\{d(u),d(w)\}+\bowtie_G)$, still suffering from the $\Omega(\bowtie_G)$-barrier.

Space Consumption. As *BiT-BU* needs to materialize all the blooms, its space consumption is $O(\sum_{(u,w)\in E}\min\{d(u),d(w)\})$, which is worse than *WingDec*'s $O(n+m)$.

The *PBNG* Algorithm. *PBNG* was originally designed as a parallel algorithm for Bitrust Decomposition, yet it also works well with a single thread. The crucial idea of *PBNG* is that the strict peeling order on the edges in the peeling process is not necessary. To see this, consider an edge e with bitruss number $b(e)$; the peeling order of those edges with bitruss numbers smaller than $b(e)$ do not affect the peeling of e in the peeling process. Motivated by this, *PBNG* partitions the bitruss number spectrum into ranges: $R_1, R_2, \ldots, R_\ell$, and assigns each edge to exactly one range. Then *PBNG* performs the peeling process on each bitruss number value range one by one. Specifically, when the peeling process is done on a range R_i, it performs a *batch* maintenance on the butterfly supports for the edges of R_{i+1} and then starts the peeling process on R_{i+1}. Thanks to this heuristic, *PBNG* usually runs faster than *BiT-BU* due to the batch updates on the butterfly supports and consumes slightly less peak memory usage by performing the peeling process only on the edges of a range R_i at a time.

Overall Running Time. Despite of the effective heuristic, *PBNG* still has to touch every butterfly at least once. The overall running time of *PBNG* is still bounded by $O(\sum_{(u,w)\in E}\min\{d(u),d(w)\}+\bowtie_G)$.

Space Consumption. As *PBNG* also needs to construct blooms in the peeling process, its space consumption complexity is the same as that of *BiT-BU*, bounded by $O\left(\sum_{(u,w)\in E}\min\{d(u),d(w)\}\right)$.

An Open Question. All the state-of-the-art algorithms suffer from the bound $\Omega(\bowtie_G)$ of "touching" each butterfly at least once. The question that whether there exists an algorithm for Bitruss Decomposition that can overcome this $\Omega(\bowtie_G)$-barrier on the running time remains open. In this paper, we give an *affirmative answer* to this open question and propose the first algorithm with running time bounded by $O(\log m\cdot\sum_{(u,w)\in E}\min\{d(u),d(w)\})$, which is just within a log factor of the cost of butterfly support computation. Moreover, as discussed earlier, the running time bound of our algorithm improves the those state-of-the-art ones by almost up to a factor of $O(\sqrt{m})$.

3.4 Distributed Tracking

The last piece of preliminaries is about the *Distributed Tracking* (DT) problem [5, 23] and its solution techniques. The DT problem is defined in a *distributed* environment, where there are h participants, $s_1, s_2, \ldots, s_h$, and a *coordinator*, q. Each participant s_i (for $i=1,2,\ldots,h$) can only communicate with the coordinator q by sending *messages* through a 2-way communication channel; however, direct communications between participants are prohibited. In the DT problem, each participant has an integer *counter* c_i. Initially, each counter c_i is set to 0, and they can be incremented over time. At each timestamp, *at most one* (that means, there can be none) of the h counters is incremented by at least 1. On the other hand, the coordinator q has an integer threshold $\tau>0$. The task of the coordinator q in the DT problem is to report *immediately* the "*maturity moment*" when the sum of all these h counters at least τ, namely, $\sum_{i=1}^{h}c_i\geq\tau$. The goal of the DT problem is to minimize the *total number of messages* (each of $O(1)$ words) sent and received by q, with which q can *correctly* capture the maturity moment. It is known that the DT problem can be solved with $O(h\log\frac{\tau}{h})$ messages [5, 23]. In this paper, we are particularly interested in a special version of the DT algorithms, called *Power-of-Two-Slack DT* (P2S-DT) [23], whose pseudo code is shown in Algorithm 1.

Analysis. It is shown in [23] that, the communication cost in each round is bounded by $O(h)$. Furthermore, since at the end of each round, τ_{cur} decreases by a factor of 2 (see Line 11in Algorihtm 1). there are at most $O(\log(\tau/h))$ rounds. The overall communication cost is, therefore, bounded by $O(h\log(\tau/h))$ messages.

4 Bitruss Decomposition with DT

In this section, we reveal the details of our algorithm, *Bitruss Decomposition with Distributed Tracking* (for short, *BiT-DT*) algorithm. Specifically, our *BiT-DT* algorithm:

- Works under the Peeling Process Framework;
- Materializes all the blooms (as what *BiT-BU* does) in the implementation of the peeling process (Step 3).

The crucial difference between our *BiT-DT* and the *BiT-BU* algorithm lies in the way how to *identify the next edge to peel*. This difference

Algorithm 1: The Power-of-Two-Slack DT Algorithm [23]

Data: A coordinator q with a threshold τ and h participants $s_1, \ldots, s_h$: each is with a counter c_i for $i = 1, \ldots, h$
Result: Capture the maturity moment for q

1 $\tau_{\text{cur}} \leftarrow \tau$, $c_i \leftarrow 0$, for $i = 1, \ldots, h$;
2 **while** *true* **do**
3 **if** $\tau_{cur} \le 2h$ **then**
4 Report to q for every counter increment until the maturity moment is captured and then return;
5 q sends a slack $\lambda = 2^{\lfloor \log_2 \frac{\tau_{\text{cur}}}{2h} \rfloor}$ to each participant;
6 $c_i^{\text{last}} \leftarrow c_i$, for $i = 1, \ldots, h$; ▷ Initialize the last communicated counter values
7 $\Delta q \leftarrow 0$; ▷ Initialize the total observed counter increment
8 **if** c_i *is increased and* $\lfloor \frac{c_i}{\lambda} \rfloor - \lfloor \frac{c_i^{last}}{\lambda} \rfloor \ge 1$ **then**
9 $\Delta q \leftarrow \Delta q + c_i - c_i^{\text{last}}$; ▷ s_i sends the counter increment $c_i - c_i^{\text{last}}$ to q
10 Update the last communicated counter: $c_i^{\text{last}} \leftarrow c_i$;
11 **if** $\Delta q \ge \tau_{cur}/2$ **then**
12 Collect counters c_i and c_i^{last} from each s_i;
13 $\tau_{\text{cur}} \leftarrow \tau_{\text{cur}} - \left(\Delta q + \sum_{i=1}^{h}(c_i - c_i^{\text{last}})\right)$;
14 **if** $\tau_{cur} \le 0$ **then**
15 Report maturity and return;
16 **else**
17 Continue, i.e., start a new round from Line 3;

is the key to eliminating the need of enumerating all the butterflies in the worst case, and hence, to overcome the $\Omega(\bowtie_G)$-barrier in the running time complexity.

The core design of our algorithm stems from a useful observation: ***It is not necessary to maintain the precise butterfly support for each edge.*** According to the peeling process framework (Step 3), to ensure the correctness, it suffices to identify every edge e whose butterfly support in G_{cur} is no more than the current bitruss number b_{cur}, namely, $\bowtie_{G_{\text{cur}}}(e) \le b_{\text{cur}}$. Based on this observation, our algorithm aims to track, for edge edge e, the *first moment* when $\bowtie_{G_{\text{cur}}}(e) \le b_{\text{cur}}$ holds. This first moment is called the *target moment* of the edge e, at which e should be peeled from G_{cur}. We adopt the DT technique to capture such target moment for each edge e. The correctness of our *BiT-DT* follows immediately from the correctness of both the DT technique and the Peeling Process Framework.

4.1 Creating a DT Instance Per Edge

To capture the target moment for an edge e, we construct a DT instance, denoted by $DT(e)$, for the edge e as follows:

- The coordinator q is the edge e itself and $\tau = \bowtie_G(e)$;
- There are $h = |\mathcal{B}(e)| + 1$ participants, where:
  - $\mathcal{B}(e)$ is the set of all the blooms containing e;
  - Each of the first $(h-1)$ participants corresponds to a bloom $B_i \in \mathcal{B}(e)$ and is associated with a counter c_i recording the number of butterflies in B_i which: (i) contain e and (ii) have been *destroyed* (due to edge peeling) so far;
  - The h^{th} participant is associated with the global bitruss number b_{cur} as its counter c_h.

The goal of $DT(e)$ is to report the maturity moment when $\sum_{i=1}^{h} c_i \ge \tau$ holds. Specifically, $DT(e)$ *conceptually* works as follows:

- When an edge e' is peeled from a bloom $B_i \in \mathcal{B}(e)$:
  - If e is a twin edge of e', $c_i \leftarrow c_i + L(B_i) - 1$, and freeze the counter c_i, that is, c_i is no longer incremented any further;
  - Otherwise, $c_i \leftarrow c_i + 1$.
- When the global bitruss number b_{cur} increases by 1, $c_h \leftarrow c_h + 1$.

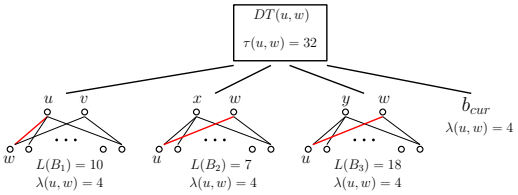


Figure 2: A DT instance of edge (u, w)

- When $DT(e)$ matures, report that e is an edge to be peeled next.

Example. Figure 2 shows an example of the DT instance of edge (u, w), which is contained in three blooms of sizes $L(B_1) = 10$, $L(B_2) = 7$ and $L(B_3) = 18$, respectively. Thus, the butterfly support of (u, w) is $\bowtie_G(u, w) = \sum_{i=1}^{3}(L(B_i) - 1) = 32$, and the DT threshold $\tau(u, w) = 32$. There are four participants in $DT(u, w)$, where three for the blooms, each of which has a counter c_i recording the number of butterflies containing (u, w) in B_i that have been destroyed, and one for the global bitruss number b_{cur} with a counter $c_4 = b_{\text{cur}}$. According to Algorithm 1, slack $\lambda(u, w) = 2^{\lfloor \log_2 \frac{32}{2\times 4} \rfloor} = 2^2 = 4$. When $DT(u, w)$ matures, that is $\sum_{i=1}^{4} c_i \ge \tau(u, w) = 32$, then the edge (u, w) needs to be peeled.

The correctness of the above process follows from this theorem.

Theorem 4.1 (Correctness). *The DT instance $DT(e)$ reports the maturity* if and only if $\bowtie_{G_{cur}}(e) \le b_{cur}$ *holds.*

Proof. The proof can be found in Appendix B. □

By Theorem 4.1, simulating the DT instance $DT(e)$ for each edge e gives a correct peeling algorithm for the Bitruss Decomposition problem. However, the issue is that simulating each $DT(e)$ *separately* essentially has no difference from maintaining a precise butterfly support counter for each edge. To see this, for each edge e' peeled, it has to increase the counter in $DT(e)$ for each edge e in a same bloom of e'. And hence, it still has to touch each butterfly once and results a cost of $\Omega(\bowtie_G)$.

4.2 Sharing Counters Across the DT Instances

To overcome the aforementioned issue, we introduce a novel approach to share counters across the DT instances. Before we show the details, we first give some key observations which reveal the rationale in the design of our algorithm:

(1) The counters for the global bitruss number for b_{cur} are the same in all the DT instances. It thus suffices to just maintain one counter for b_{cur} which is shared across all the DT instances.
(2) Each bloom B_i serves as a participant in the $DT(e)$ for each $e \in B_i$. When an edge e' is peeled from a bloom B_i, for each edge $e \in B_i$ other than e'_{twin}, the twin edge of e', the counter with respect to B_i in $DT(e)$ is incremented by 1 simultaneously. For e'_{twin}, the counter with respect to B_i in its DT instance is incremented by the current size of $B_i \setminus \{e', e'_{\text{twin}}\}$, i.e., $L(B_i \setminus \{e', e'_{\text{twin}}\})$, after which no more butterfly containing e'_{twin} is in B_i. Hence, e'_{twin} can be safely removed from B_i and the participant with respect to B_i can be also removed from $DT(e'_{\text{twin}})$.
(3) In P2S-DT (Algorithm 1), the slack λ is always a power-of-two value 2^i for some i; every time a participant's counter reach to the *next integer multiple* of 2^i (Line 10 in Algorithm 1), it reports

the counter increment to q. Thus, we can store all the edges in a bloom B into a *power-of-two* bucket array by their slack value λ. Whenever the counter of B reaches an integer multiple of 2^i, B instructs all the edges in the i^{th} bucket to send the counter increment to their coordinators according to Algorithm 1.

We organize the DT instances with shared counters according to the above observations.

Data Structures for Each Bloom. For each bloom B, we maintain the following data structures:

- A Counter c_B: it records the number of edges *peeled* from B. Note that c_B does not count for those edges removed from B as a twin edge of some edge peeled from B. In other words, for an edge e peeled from B, while its twin edge $\text{twin}(e)$ is also removed from B, c_B only increments by 1. Thus, it can be verified that the value of c_B is always at most $L(B)$, the original size of B.
- A Bucket Array $\mathcal{A}_B$ of length $\lfloor \log_2 L(B) \rfloor + 1$, where:
  - For $i = 0, 1, \ldots, \lfloor \log_2 L(B) \rfloor$, $\mathcal{A}_B[i]$ is a bucket storing a pair $(e, c_B^{\text{last}}(e))$ for each of those edges $e \in B$ whose participant of B in $DT(e)$ has a slack value $\lambda(e) = 2^i$;
  - For the last bucket with $i = \lfloor \log_2 L(B) \rfloor + 1$, it stores the pairs $(e, c_B^{\text{last}}(e))$ for all the edges with $\lambda(e) \geq L(B)$.

  Furthermore, $c_B^{\text{last}}(e)$ is the value of counter c_B when the participant of B last communicated with the coordinator $q(e)$ in $DT(e)$. Initially, $c_B^{\text{last}}(e) \leftarrow c_B$ when e was inserted to a bucket.

Data Structures for the Global Bitruss Number. Our *BiT-DT* maintains a *global participant* s_{bn} with respect to the global bitruss number b_{cur}, which serves as a participant for all the DT instances. The data structures of s_{bn} are analogous to the blooms'.

- A Counter $c_{s_{\text{bn}}}$: it records the current global bitruss number b_{cur} in the peeling process, such that $c_{s_{\text{bn}}} = b_{\text{cur}}$ always holds.
- A Bucket Array $\mathcal{A}_{s_{\text{bn}}}$ of length $\lfloor \log_2 m \rfloor$. Each bucket $\mathcal{A}_{s_{\text{bn}}}[i]$ stores a pair $(e, c_{s_{\text{bn}}}^{\text{last}}(e))$ for each of those edges $e \in E$ with a slack $\lambda(e) = 2^i$, where $c_{s_{\text{bn}}}^{\text{last}}(e)$ is the value of $c_{s_{\text{bn}}}$ when s_{bn} last communicated with $q(e)$ in $DT(e)$. Initially, $c_{s_{\text{bn}}}^{\text{last}}(e) \leftarrow c_{s_{\text{bn}}}$ when e was inserted to some bucket in $\mathcal{A}_{s_{\text{bn}}}$.

4.3 Our *BiT-DT* Algorithm

Algorithm 2 gives the pseudocode of our *BiT-DT* algorithm. Its core idea is to maintain the DT instances to identify all the edges which should be peeled next. Below, we explain the insight of each main component in *BiT-DT* to achieve this purpose.

Construct All the DT Instances. Lines 4-9 construct DT instance for each edge. Basically, it just computes the slack $\lambda(e)$ for each edge e (according to Algorithm 1), and then insert the pair $(e, c_p^{\text{last}}(e))$ to the corresponding bucket in each participant p in $DT(e)$.

Process Counter Increments for s_{bn}. When the global bitruss number b_{cur} increments by one, we need to increment the counter of the global participant s_{bn}. To achieve this, Algorithm 3 is invoked (Line 15 in Algorithm 2) with s_{bn} as the input participant.

To see how Algorithm 3 works, recall that s_{bn} stores all the edges in the buckets in $\mathcal{A}_{s_{\text{bn}}}$ with respect to their current slack values in the corresponding DT instances. Every time the counter $c_{s_{\text{bn}}}$ reaches to an *integer multiple* of 2^i, it scans the i^{th} bucket $\mathcal{A}_{s_{\text{bn}}}[i]$

Algorithm 2: Our *BiT-DT* Algorithm

```
Input: A bipartipte graph G = ⟨A ∪ B, E⟩
Output: For each edge e ∈ E, return the bitruss number of e
1  Compute the butterfly support ⧖_G(e) for each e ∈ E;
2  Compute all the blooms in G;
3  Initialize the data structures for each bloom and for s_bn;
4  foreach e ∈ E do
5      Let P ← B(e) ∪ {s_bn} be the participant set in DT(e);
6      τ(e) ← ⧖_G(e), h ← |P|, i ← ⌊log_2 τ(e)/2h⌋;
7      foreach Participant p ∈ P do
8          Initialize c_p^last(e) ← c_p;
9          Insert a pair (e, c_p^last(e)) to bucket A_p[i];
10 G_cur ← G, b_cur ← 0;
11 Initialize the set of edges to be peeled F ← ∅;
12 while G_cur is non-empty do
13     while F is empty do
14         b_cur ← b_cur + 1;
15         F ← IncCounter(s_bn);                    ▷ Algorithm 3
16     while ∃e ∈ F do
17         F ← F \ {e};
18         Remove e from all the participants in DT(e);
19         foreach B ∈ B(e) do
20             F' ← PeelEdge(B, e);                 ▷ Algorithm 4
21             F ← F ∪ F';
22         Bitruss number b(e) ← b_cur;
23         G_cur ← G_cur \ {e};
24 return the bitruss number b(e) for each e ∈ E;
```

Algorithm 3: IncCounter(p)

```
Input: A participant p which can be either the global participant for the bitruss number
       or a bloom
Output: The set F of all the edges managed by this participant to be peeled next
1  c_p ← c_p + 1;
2  j ← log_2((c_p − 1) ⊕ c_p + 1);          ▷ Stopping index for bucket scanning
3  i ← 0, F ← ∅;
4  while i < j do
5      foreach (e, c_p^last(e)) ∈ A_p[i] do
6          Send the counter increment c_p − c_p^last(e) to coordinator q(e) in DT(e);
7          c_p^last(e) ← c_p;
8          if DT(e) starts a new round then
9              Move e to the bucket of the new slack in each participant p' of DT(e)
                 and update c_p'^last(e);
10         if DT(e) matures then
11             F ← F ∪ {e};
12     i ← i + 1;
13 return F;
```

and instructs each edge $e \in \mathcal{A}_{s_{\text{bn}}}[i]$ to send the counter increment, i.e., $c_{s_{\text{bn}}} - c_{s_{\text{bn}}}^{\text{last}}(e)$, to the coordinator $q(e)$ in $DT(e)$, according to P2S-DT (Algorithm 1).

This bucket scanning rule can be implemented by maintaining the counter $c_{s_{\text{bn}}}$ in binary representation and scanning the buckets corresponding to all those bits flipped when $c_{s_{\text{bn}}}$ is incremented by one. For example, when $c_{s_{\text{bn}}}$ is incremented from 0 to 1, only $\mathcal{A}_{s_{\text{bn}}}[0]$ is scanned because only the lowest bit in the binary representation of $c_{s_{\text{bn}}}$ is flipped. When $c_{s_{\text{bn}}}$ is further incremented to 2, as the binary representation changes from $01|_2$ to $10|_2$, the two lowest bits flipped, and hence, the first two buckets should be scanned. When $c_{s_{\text{bn}}}$ is incremented by one, all the bits flipped for this increment can be computed by the bit-wise XOR between $(c_{s_{\text{bn}}} - 1)$ and $c_{s_{\text{bn}}}$, namely, $(c_{s_{\text{bn}}} - 1) \oplus c_{s_{\text{bn}}}$. As a result, the index $j = \log_2\left((c_{s_{\text{bn}}} - 1) \oplus c_{s_{\text{bn}}} + 1\right)$ computed in Line 3 in Algorithm 3

Algorithm 4: PeelEdge(B, e)

Input: A bloom B and an edge $e \in B$
Output: The set F of all the edges in B to be peeled next

1 // Handle the twin edge of e first;
2 $e_{\text{twin}} \leftarrow$ the twin edge of e in bloom B;
3 $B \leftarrow B \setminus \{e, e_{\text{twin}}\}$;
4 $L(B) \leftarrow$ the current size of B;
5 Send the counter increment $c_B - c_B^{\text{last}}(e_{\text{twin}}) + L(B)$ to coordinator $q(e_{\text{twin}})$ in $DT(e_{\text{twin}})$;
6 Remove participant B from $DT(e_{\text{twin}})$;
7 **if** $DT(e_{twin})$ *starts a new round* **then**
8 Move e_{twin} to the bucket of the new slack in each participant p of $DT(e_{\text{twin}})$ and update $c_p^{\text{last}}(e_{\text{twin}})$;
9 **if** $DT(e_{twin})$ *matures* **then**
10 $F \leftarrow F \cup \{e_{\text{twin}}\}$;
11 // Handle the remaining edges in B;
12 $F' \leftarrow$ IncCounter(B); ▷ Algorithm 3
13 $F \leftarrow F \cup F'$;
14 **return** F;

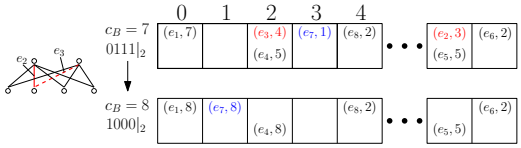


Figure 3: Example of peeling an edge e_3 from a bloom B

is the index of the first bucket which should not be scanned, and such an index j is called the *stopping index*.

Peel an Edge from a Bloom. Algorithm 4 shows how to peel an edge e from a bloom B. For the twin edge e_{twin} of e in B, it sends the counter increment plus $L(B \setminus \{e, e_{\text{twin}}\})$, the size of B after removing e and e_{twin}, to the coordinator $q(e_{\text{twin}})$. Afterward, B is removed from $DT(e_{\text{twin}})$ because all the butterflies containing e_{twin} in B have been destroyed as e is peeled. For the remaining edges in B, it just invokes Algorithm 3 to handle the counter increment, which is the same process for the global participant s_{bn}.

Example. Figure 3 shows a running example of peeling an edge e_3 from a bloom B. The upper row shows the status on the bucket array $\mathcal{A}_B$ and $c_B = 7$ before the red dashed edge e_3 being peeled. For instance, $(e_4, 5)$ in the bucket $\mathcal{A}_B[2]$ means that the current slack $\lambda(e_4) = 2^2 = 4$ in $DT(e_4)$ and the last communication with coordinator $q(e_4)$ was happened when $c_B = 5$, namely, $c_B^{\text{last}} = 5$.

The lower row shows the status after e_3 is peeled from B. Firstly, when e_3 is peeled, both e_3 and e_2 (as it is the twin edge of e_3) are removed from B. Algorithm 4 then instructs e_2 to send the counter increment computed as $c_B - c_B^{\text{last}}(e_2) + L(B \setminus \{e_3, e_2\}) = 7 - 3 + 3 = 7$ to the coordinator $q(e_2)$ in $DT(e_2)$. To handle the counter increment for the remaining edges, Algorithm 3 is invoked. Specifically, it first increments c_B by one, and then computes the stopping index j for the bucket scanning in $\mathcal{A}_B$. To achieve this, it frist computes $(c_B - 1) \oplus c_B = 0111|_2 \oplus 1000|_2 = 1111|_2$, and then $1111|_2 + 1 = 10000|_2 = 16$; taking the logarithmic with base 2, we have the stopping index $j = \log_2 16 = 4$. Therefore, all the buckets with index $i < 4$ in $\mathcal{A}_B$ are needed to be scanned. For each edge e in those buckets, Algorithm 3 instructs e to send the counter increment $c_B - c_B^{\text{last}}(e)$ to their coordinator $q(e)$ and then updates $c_B^{\text{last}}(e)$ to the current value of c_B. For example, after scanning buckets $\mathcal{A}_B[0]$ and $\mathcal{A}_B[2]$, $c_B^{\text{last}}(e_1)$ and $c_B^{\text{last}}(e_5)$ are updated to 8.

Algorithm 5: BatchPeeling(F)

Input: A set F of edges to be peeled
Output: A set of F' of edges to be peeled next

1 $F' \leftarrow \emptyset$; $\mathcal{S} \leftarrow \emptyset$; ▷ A set records the blooms affected
2 **foreach** $e \in F$ **do**
3 remove e from all the participants in $DT(e)$;
4 **foreach** $B \in \mathcal{B}(e)$ **do**
5 **if** $B \notin \mathcal{S}$ **then**
6 $j_B \leftarrow 1$; ▷ Initialize the stopping index
7 $c_B \leftarrow c_B + 1$;
8 $j_B \leftarrow \max\{j_B, \log_2((c_B - 1) \oplus c_B + 1)\}$; ▷ Record the largest stopping index of B seen so far
9 run Lines 1 - 8 in Algorithm 4 to handle e's twin edge e_{twin} in B, and add e_{twin} to F' if its DT matures;
10 $\mathcal{S} \leftarrow \mathcal{S} \cup \{(B, j_B)\}$;
11 bitruss number $b(e) \leftarrow b_{\text{cur}}$;
12 $G_{\text{cur}} \leftarrow G_{\text{cur}} \setminus \{e\}$;
13 **foreach** $B \in \mathcal{S}$ **do**
14 run Lines 5-13 in Algorithm 3 on B with stopping index $j = j_B$ and collect all the edges whose DT instances mature in F';
15 **return** F';

Since the current round of their DT's are still on-going, they remain in the same bucket. In contrast, after processing e_7 in $\mathcal{A}_B[3]$, e_7 is moved to bucket $\mathcal{A}_B[1]$ because $DT(e_7)$ starts a new round with $\lambda(e_7) = 2$. This completes the process for peeling e_3 from B.

We have the following theorem:

Theorem 4.2. *Given a bipartite graph $G = \langle A \cup B, E \rangle$, there exists an implementation of our* BiT-DT *algorithm which correctly solves the Bitruss Decomposition problem in $O(\sum_{(u,v)\in E} \min\{d(u), d(v)\} \cdot \log m)$ time, where $m = |E|$, and its overall space consumption is bounded by $O(\sum_{(u,v)\in E} \min\{d(u), d(v)\})$.*

Proof. The proof can be found in Appendix C. □

Remark. As disucssed in Appendix A, $\sum_{(u,v)\in E} \min\{d(u), d(v)\}$ is upper bounded by $O(m^{1.5})$ in the worst case.

5 An Optimization: Batch Edge Peeling

In this section, we introduce a powerful heuristic, called *Batch Edge Peeling*, to further enhance the practical performance of our *BiT-DT* algorithm. The basic idea is to peel the edges in batch instead of peeling them one by one. The rationale of this heuristic comes from a crucial observation: The set F of edges to be peeled in Algorithm 2 often contains more than one edge from the same bloom.

According to this observation, peeling the edges in F in batch would have the following two advantages. First, peeling multiple edges from the same bloom in batch can avoid repeatedly scanning certain buckets. Second, it also makes each "communication" in the DT instances more effective as the counter increment sent to the coordinators is potentially larger, and hence, the DT instance can track more counter increments in a round. As shown in our experimental result in Section 6, this heuristic substantially enhances the efficiency of our *BiT-DT* algorithm.

Algorithm 5 shows the detailed implementation of our batch edge peeling heuristic. It can be plugin to our *BiT-DT* by replacing the while-loop (Line 16 - 23) in Algorithm 2 with a line of code:

$$F \leftarrow \text{BatchPeeling}(F)\ ;$$

At a high level, given a set F of edges to be peeled, Algorithm 5 increments the counters of all the "affected" blooms B, records the largest stopping index j_B seen so far and processes the twin

Table 1: Summary of Datasets

Dataset	$\lvert A\rvert$	$\lvert B\rvert$	$\lvert E\rvert$	$\lvert \bowtie_G \rvert$
Github	56,519	120,867	440,237	50,894,505
Discogs	1,754,823	270,771	5,302,276	3,261,758,502
Artist–style	1,617,943	383	5,740,842	77,383,418,076
Movie	69,878	10,677	10,000,054	1,197,019,065,804
Epinions	120,493	755,760	13,668,320	170,305,214,345
Wiki(pl)	207,781	2,664,432	21,219,204	1,843,357,963,467
Wiki(fr)	757,621	8,870,762	52,950,008	4,493,825,773,935
Delicious	833,081	33,778,221	101,798,957	56,892,252,403
Livejournal	3,201,203	7,489,073	112,307,385	3,297,158,439,527
Tracker	27,665,730	12,756,244	140,613,762	20,067,567,209,850

Table 2: Memory Consumption (GB)

Dataset	BIT-BU	BIT-PC	PBNG	Our-DT	Our-Batch
Github	0.40	0.42	0.33	0.86	0.86
Discogs	3.30	4.55	2.86	6.59	6.59
Artist–style	1.95	2.08	1.80	3.12	3.22
Movie	103.84	107.26	94.61	191.97	191.93
Epinions	19.19	20.53	16.83	36.40	36.15
Wiki(pl)	17.06	15.81	15.02	30.42	30.42
Wiki(fr)	54.42	56.01	49.49	99.10	99.10
Delicious	330.01	342.56	275.95	702.40	703.20
Livejournal	184.47	194.68	160.67	349.88	350.53
Tracker	239.86	258.92	209.37	455.06	456.93

edges for all the edges in F together. And then, for each affected bloom B, it scans its bucket array up the stopping index j_B once and processes all the edges in those scanned buckets accordingly.

6 Experiments

In this section, we examine the effectiveness of our algorithms on ten public real-world datasets.

6.1 Experiment Setting

Competitors. We evaluate two variants of our *BiT-DT* algorithm, denoted *Our-DT* and *Our-Batch*, to distinguish our methods from existing competitors. Specifically, *Our-DT* is the vanilla version of *BiT-DT* (Algorithm 2), whereas *Our-Batch* equips with the Batch Edge Peeling heuristic (Section 5). As for the state-of-the-art baselines, we adopt: (i) the vanilla *BiT-BU*, (ii) the strengthened *BiT-BU++*, (iii) its fastest version *Bit-PC*, and (iv) the recent *PBNG* algorithm, discussed in Section 3.3. As demonstrated in [20], *BiT-BU* outperforms *WingDec*; therefore we exclude *WingDec* from our experiments.

Evaluation Metrics. We evaluate the performance of these algorithms in terms of running time and peak memory consumption.

Environment. All the six algorithms are implemented in C++ compiled with -O3. The source code of the implementation of our algorithms can be found in [9]. The experiments are conducted on a Linux server equipped with an Intel Xeon Platinum 8375C CPU and 2 TB of memory. All algorithms are run with a single thread and terminated if it takes more than 100,000 seconds.

Datasets. Our experiments are run on ten real-world datasets from KONECT [7]. Table 1 gives the statistics of these datasets.

6.2 Overall Performance

Overall Running Time. Figure 4 shows the overall running time of all the competitors on each dataset. The y-axis is in log-scale (with base of 10), and each group corresponds to a dataset. In each group, there are six bars corresponding to the overall running time of the six competitors. Each bar consists of two parts, where the lower part shows the running time of initialization which includes the butterfly support computation and bloom construction, while the upper part indicates the running time of the peeling process. These two parts combined together is the overall running time of the algorithm.

From Figure 4, we make the following observations: (1) The initialization time are similar among all the competitors and negligible compared to their overall running time on large datasets. This reflects the true challenge in Bitruss Decomposition lies in the Peeling Process. (2) Across all the datasets except for Delicious, *PBNG* consistently outperforms *BiT-BU*, *BiT-BU++* and *Bit-PC*. (3) Except for Github (the smallest dataset) and Delicious (which will be further analyzed in Ablation Study), *Our-Batch* substantially outperforms *Our-DT*. This clearly shows the effectiveness of our Batch Edge Peeling optimization. (4) The performance of *Our-DT* and *PBNG* is mixed, with each algorithm outperforming the other on different datasets. However, *Our-Batch* is faster and substantially outperforms *PBNG* and *Bit-PC* on most of the datasets except for Github. Notably, on the Wiki(pl) and Wiki(fr) datasets, *Our-Batch* is roughly five times ($5\times$) faster than *PBNG* and almost ten times ($10\times$) faster than *Bit-PC*. This shows the effectiveness of our proposed algorithms.

Peak Memory Consumption. Table 2 shows the peak memory consumption the competitors, where the one of *BiT-BU++* is omitted as it is the same as *BiT-BU*. From this table, we can see that both the optimized versions *Bit-PC* and *Our-Batch* consume slightly more memory space than their corresponding vanilla versions *BiT-BU* and *Our-DT*, respectively. Moreover, while the theoretical space consumption bounds are the same, *Our-Batch* consumes roughly 1–2 times and 2-3 times more space than *Bit-PC* and *PBNG* do, respectively. The reason is that *Our-Batch* needs to maintain some extra data structures in each bloom, i.e., a counter of each edge, the bucket array and the DT instances, which introduce a small constant blow up in the space consumption.

6.3 Scalability Study

To study the scalability of the algorithms, we take the largest five datasets: Wiki(pl), Wiki(fr), Delicious, Livejournal, and Tracker; and we randomly delete edges from them until there remain 10%, 20%, 50% and 80% of the edges, respectively. Figure 5 shows the overall running time of *Bit-PC*, *PBNG* and *Our-Batch*. We observe that: (1) Except for Delicious, *Our-Batch* is superior to *Bit-PC* and *PBNG* across the other four datasets with different sizes. In particular, *Our-Batch* is up to an order of magnitude faster than *Bit-PC* on Wiki(pl), Wiki(fr), and LiveJournal when the graphs are with $\geq 50\%$ of the edges; likewise, on Wiki(pl) and Wiki(fr), *Our-Batch* remains at least $3\times$ faster than *PBNG* when the percentage of the edges exceeds 80%. Moreover, the performance gap widens as the datasets contain more edges. (2) On Delicious, *Our-Batch* outperforms *Bit-PC* when the percentage of the edges is up to 50%, while it becomes roughly the same as *Bit-PC* on 80% and 100% of the edges. The performance of *Our-Batch* and *PBNG* are similar on Delicious, except that *Our-Batch* is faster when the percentage of the edges is 10%. Figure 5 demonstrates that *Our-Batch* is superior to *Bit-PC* and *PBNG* overall in terms of efficiency.

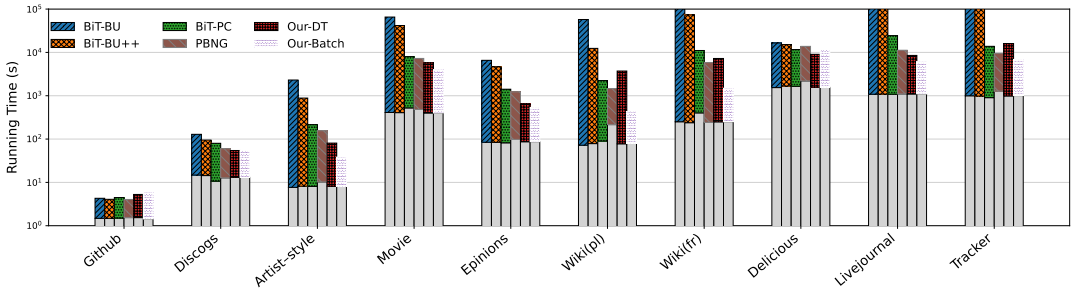


Figure 4: Overall Running Time on All Datasets

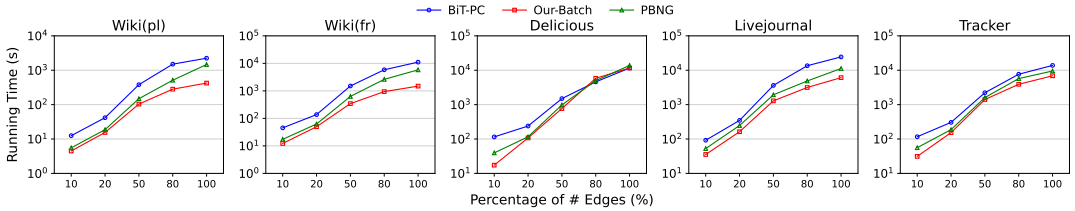


Figure 5: Impact of Graph Size

Table 3: Ablation Study

Dataset	$\bowtie_G/\lvert E\rvert$	Our-DT		Our-Batch	
		$\overline{DT}$	$\overline{BK}$	$\overline{DT}$	$\overline{BK}$
Github	115.61	0.89	10.48	0.88	8.81
Discogs	615.16	1.43	7.58	1.41	5.43
Delicious	558.87	0.81	42.136	0.81	39.55
Movie	119,701.26	5.38	222.74	4.64	103.12
Epinions	12,459.85	3.87	28.06	3.65	13.72

6.4 Ablation Study

Next, we conduct an ablation study on our algorithms *Our-DT* and *Our-Batch* to see in what cases they would not work well and when the Batch Edge Peeling would be less effective. We adopt two types of datasets in our study: (i) Github, Discogs and Delicious, on which *Our-Batch* performs worse than or comparable to its vanilla version *Our-DT* does, and (ii) Movie and Epinions, where *Our-Batch* is better than *Our-DT*. We characterize these datasets by the butterfly-edge ratio, which is computed as the number of butterflies divided by the number of edges, namely, $\bowtie_G/|E|$. The larger this ratio, the denser the graph is. Furthermore, we measure the performance of our algorithms in terms of: (i) the average number of DT rounds per edge, denoted by $\overline{DT}$, and (ii) the average number of buckets scanned per edge, denoted by $\overline{BK}$.

Table 3 shows the experimental results, where the average number of DT rounds can be less than 1. This is due to our implementation optimisation which adopts the brute-force approach when the DT slack value is less than 32, and hence, the DT round number in this case is 0. Combining the results shown in Figure 4, we can see that on those datasets (in the first group) with small $\bowtie_G/|E|$ ratio, *Our-Batch* does not win *Our-DT* much. This is because, on these datasets, each edge is contained in a small number of butterflies and hence, the size of each bloom tends to be small making it less likely that multiple edges are peeled from the same bloom simultaneously. As a result, it considerably limits the power of the Batch Edge Peeling heuristic. This can also be seen from Table 3, the values of $\overline{DT}$ and $\overline{BK}$ of the two algorithms are pretty much the same. These small values of $\overline{DT}$ also reflect that the DT approach in our algorithms is less effective on these datasets, as just in no more than two rounds on average, a DT instance matures. It means that the threshold τ is not much larger than the number of participants, and hence, DT is not much better than the straightforward brute-force butterfly support updating approach (equivalently, enumerating the butterflies). This explains why we observe less or no superiority from our algorithm comparing to *Bit-PC* on these datasets (e.g., Delicious).

On the second group of datasets with larger $\bowtie_G/|E|$ ratio in Table 3, *Our-Batch* starts outperforming *Our-DT* from the smaller average number of scanned buckets $\overline{BK}$ and smaller average DT rounds $\overline{DT}$. It demonstrates that the batch peeling idea can substantially save the cost on both DT tracking and bucket scanning in the peeling process.

7 Conclusion

In this paper, we study the Bitruss Decomposition problem. All the previous state-of-the-art algorithms have to "touch" each butterfly in the given bipartite graph G at least once, and hence, suffer from a running time cost $\Omega(\bowtie_G)$, where $\bowtie_G$ is the total number of butterflies in G and $\bowtie_G$ can be as large as $O(m^2)$, and m is the number of edges in G. We propose the first algorithm, called *BiT-DT*, for Bitruss Decomposition, which can get around the aforementioned $\Omega(\bowtie_G)$-barrier and achieves running time complexity $O(\log m \cdot \sum_{(u,v)\in E} \min\{d(u), d(v)\})$. As it is known that this summation term is upper bounded by $O(m^{1.5})$ in the worst case, thus, our *BiT-DT* significantly improves the state-of-the-art $O(m^2)$ bound by roughly a factor of $O(\sqrt{m})$. Extensive experimental results on ten real-world datasets show that our *BiT-DT* algorithm outperforms the SOTA baselines by up to an order of magnitude.

Acknowledgment

In this work, Junhao Gan is in part supported by ARC DP230102908. We also would like to thank the anonymous reviewers for their valuable feedback and suggestions.

References

[1] Noga Alon. 1988. The linear arboricity of graphs. *Israel Journal of Mathematics* 62, 3 (1988), 311–325.

[2] Alex Beutel, Wanhong Xu, Venkatesan Guruswami, Christopher Palow, and Christos Faloutsos. 2013. Copycatch: stopping group attacks by spotting lockstep behavior in social networks. In *Proceedings of the 22nd international conference on World Wide Web*. 119–130.

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

[4] Jonathan Cohen. 2008. Trusses: Cohesive subgraphs for social network analysis. *National security agency technical report* 16 (2008), 3–1.

[5] Graham Cormode, S. Muthukrishnan, and Ke Yi. 2011. Algorithms for distributed functional monitoring. *ACM Trans. Algorithms* 7, 2 (2011), 21:1–21:20. https://doi.org/10.1145/1921659.1921667

[6] Jonathan L Herlocker, Joseph A Konstan, and John Riedl. 2000. Explaining collaborative filtering recommendations. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*. 241–250.

[7] Jérôme Kunegis. [n. d.]. *The KONECT Project.* konect.cc

[8] Kartik Lakhotia, Rajgopal Kannan, and Viktor Prasanna. 2023. Parallel Peeling of Bipartite Networks for Hierarchical Dense Subgraph Discovery. *ACM Trans. Parallel Comput.* 10, 2 (2023).

[9] Fengnian Lin. 2025. Source code. https://github.com/HenryLimHongni/BIT-DT

[10] L Qin, Y Peng, Y Zhang, X Lin, W Zhang, and Jingren Zhou. 2019. Towards bridging theory and practice: hop-constrained st simple path enumeration. In *International Conference on Very Large Data Bases*. VLDB Endowment.

[11] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time constrained cycle detection in large dynamic graphs. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1876–1888.

[12] Kazumi Saito, Takeshi Yamada, and Kazuhiro Kazama. 2008. Extracting communities from complex networks by the k-dense method. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* 91, 11 (2008), 3304–3311.

[13] 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.

[14] Ahmet Erdem Sarıyüce and Ali Pinar. 2018. Peeling bipartite networks for dense subgraph discovery. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*. 504–512.

[15] Jessica Shi and Julian Shun. 2020. *Parallel Algorithms for Butterfly Computations*. Society for Industrial and Applied Mathematics (SIAM), 16–30.

[16] Xiaoyuan Su and Taghi M Khoshgoftaar. 2009. A survey of collaborative filtering techniques. *Advances in artificial intelligence* 2009 (2009).

[17] Jia Wang and James Cheng. 2012. Truss Decomposition in Massive Networks. *Proceedings of the VLDB Endowment* 5, 9 (2012).

[18] 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.

[19] 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.

[20] 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.

[21] Yue Wang, Ruiqi Xu, Xun Jian, Alexander Zhou, and Lei Chen. 2022. Towards distributed bitruss decomposition on bipartite graphs. *Proc. VLDB Endow.* 15, 9 (2022), 1889–1901.

[22] Yang Zhang and Srinivasan Parthasarathy. 2012. Extracting analyzing and visualizing triangle k-core motifs within networks. In *2012 IEEE 28th international conference on data engineering*. IEEE, 1049–1060.

[23] Zhuo Zhang, Junhao Gan, Zhifeng Bao, Seyed Mohammad Hussein Kazemi, Guangyong Chen, and Fengyuan Zhu. 2022. Approximate Range Thresholding. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. 1108–1121. https://doi.org/10.1145/3514221.3526123

[24] Zhaonian Zou. 2016. Bitruss decomposition of bipartite graphs. In *International Conference on Database Systems for Advanced Applications*. Springer, 218–233.

A Comparison among Bounds

We discuss the comparisons among the bounds that we frequently see throughout this paper.

First, $\sum_{(u,w)\in E} \min\{d(u), d(w)\}$ is known to be bounded by $O(\alpha(G) \cdot m)$, where $\alpha(G)$ is the *arboricity* [1, 3] of the graph G. In general, $\alpha(G)$ often acts like a constant. However, in the worst case, $\alpha(G) = O(\sqrt{m})$. Therefore, in the worst case,

$$O\left(\sum_{(u,w)\in E} \min\{d(u), d(w)\}\right) = O(m^{1.5}).$$

Furthermore, observe that the maximum possible butterfly support of an edge e is $m-1$. Thus, the total number of butterflies $\bowtie_G$ is bounded by $O(\sum_{e\in E} \bowtie_G(e)) = O(m^2)$. As a result, the bound $O(\sum_{(u,w)\in E} \min\{d(u), d(w)\})$ improves the bound $O(\bowtie_G)$ by up to a factor of $O(\sqrt{m})$ in the worst case; this improvement is significant when m is large. Finally, since the expression

$$\sum_{(u,w)\in E} \sum_{v\in N(w)\setminus\{u\}} \min\{d(u), d(v)\} \geq \sum_{(u,w)\in E} \sum_{v\in N(w)\setminus\{u\}} |(N(u)\setminus\{w\}) \cap N(v)| = 4 \cdot \bowtie_G,$$

this bound is thus no better than $O(\bowtie_G)$; but, in the worst case, it is also bounded by $O(m^2)$.

B Proof of Theorem 4.1

According to the Peeling Process Framework, we have:

$$\bowtie_{G_{\text{cur}}}(e) = \bowtie_G(e) - |\Delta_{\text{cur}}(e)|,$$

where $\Delta_{\text{cur}}(e)$ is the set of all the butterflies that: (i) contain e and (ii) have been *destroyed* due to the edges peeled from G so far. Therefore, the target moment $\bowtie_{G_{\text{cur}}}(e) \leq b_{\text{cur}}$ is equivalent to:

$$\bowtie_G(e) - |\Delta_{\text{cur}}(e)| \leq b_{\text{cur}} \Leftrightarrow |\Delta_{\text{cur}}(e)| + b_{\text{cur}} \geq \bowtie_G(e). \quad (2)$$

Furthermore, by Fact 2 that each butterfly in $\Delta_{\text{cur}}(e)$ must be contained in one and exactly one boom which contains e, and by our contruction of $DT(e)$, $|\Delta_{\text{cur}}(e)|$ can be decomposed into:

$$|\Delta_{\text{cur}}(e)| = \sum_{B_i\in\mathcal{B}(e)} c_i. \quad (3)$$

Substituting Equation (3), $b_{\text{cur}} = c_h$ and $\tau = \bowtie_G(e)$ to Inequality (2), we have:

$$\bowtie_{G_{\text{cur}}}(e) \leq b_{\text{cur}} \Leftrightarrow \sum_{B_i\in\mathcal{B}(e)} c_i + b_{\text{cur}} \geq \bowtie_G(e) \Leftrightarrow \sum_{i=1}^{h} c_i \geq \tau.$$

C Proof of Theorem 4.2

In this section, we prove Theorem 4.2 by analyzing the overall running time and the space consumption of our *BiT-DT* algorithm.

Overall Running Time Analysis. Denote the set of all the blooms on G by $\mathcal{B}(G)$. Define $\|\mathcal{B}(G)\| = \sum_{B\in\mathcal{B}(G)} L(B)$, the total size of all the blooms in $\mathcal{B}(G)$. Recall that each bloom B is a $(2, L)$-clique and has exactly $2 \cdot L(B)$ edges. Therefore, the total sum of the number of the edges in each bloom $B \in \mathcal{B}(G)$ is bounded by $O(\|\mathcal{B}(G)\|)$. According to the discussion in Section 3, the total running time cost of the computation of butterfly supports and the construction of all the blooms in Lines 1-2 in our *BiT-DT* (Algorithm 2) is bounded by $O(\|\mathcal{B}(G)\|)$.

Next, we analyze the total running time of the peeling process in our *BiT-DT*. Recall that, to peel an edge e, our *BiT-DT* performs the following for each bloom $B \in \mathcal{B}(e)$: (i) increments the counter c_B, (ii) scans the buckets in $\mathcal{A}_B$, (iii) sends the counter increment

for each edge e' stored in the buckets scanned, and (iv) moves the edges to a new bucket if their DT instances start a new round. Furthermore, the running time cost of each increment of the global bitruss number is analogous and consists of these costs of (i) - (iv).

Clearly, the running time costs of (i) and (ii) are bounded by $O(1)$ and $O(\log L(B))$, respectively. As for the costs of (iii) and (iv), we charge them to the corresponding edges which are processed. As a result, the cost of (iii) can be charged to the total DT running time costs of the corresponding edges. As for the cost of (iv), observe that each movement of an edge among the buckets in $\mathcal{A}_B$ can be performed in $O(1)$ time, and each of such movements is triggered by a new round in the DT instance of the corresponding edge. Thus, the cost of (iv) can be also charged to the DT running costs of the corresponding edges. According to our construction of $DT(e)$, there are $|\mathcal{B}(e)| + 1$ participants and the target threshold $\tau = \bowtie_G(e) \leq m - 1$. By the analysis of the P2S-DT algorithm, the total running time of $DT(e)$ is bounded by $O((|\mathcal{B}(e)| + 1) \cdot \log m)$. Putting these together, the overall running time of *BiT-DT* is bounded by:

$$O\left(||\mathcal{B}(G)|| + \underbrace{\sum_{e\in E} |\mathcal{B}(e)| \cdot (1 + \log L(B))}_{\text{total cost of (i) and (ii)}} + \underbrace{\sum_{e\in E} (|\mathcal{B}(e)| + 1) \cdot \log m}_{\text{total cost of (iii) and (iv)}}\right)$$

$$= O\left(||\mathcal{B}(G)|| + \sum_{B\in\mathcal{B}(G)} L(B) \cdot \log m\right) = O(||\mathcal{B}(G)|| \cdot \log m),$$

where the first equality comes from the facts that $\sum_{e\in E} |\mathcal{B}(e)| = \sum_{B\in\mathcal{B}(G)} L(B) = ||\mathcal{B}(G)||$ and $L(B) \leq m$. Moreover, as discussed in Section 3, we have $||\mathcal{B}(G)|| = \sum_{(u,v)\in E} \min\{d(u), d(v)\}$. This lemma thus follows:

Lemma C.1. *Given a bipartite graph $G = \langle A \cup B, E\rangle$, our* BiT-DT *algorithm can correctly solve the Bitruss Decomposition problem in $O(\sum_{(u,v)\in E} \min\{d(u), d(v)\} \cdot \log m)$ time, where $m = |E|$.*

Space Consumption Analysis. Recall that the global participant s_{bn} stores a counter $c_{s_{\text{bn}}}$, the $(e, c^{\text{last}}_{s_{\text{bn}}}(e))$-pairs of m edges, and a bucket array of length $O(\log m)$. Thus, its space consumption is bounded by $O(m)$. Analogously, each bloom $B \in \mathcal{B}(G)$ stores a count c_B, the pairs of $2 \cdot L(B)$ edges, and a bucket array of length $O(\log L(B))$. As a result, the space consumption of each bloom $B \in \mathcal{B}(G)$ is bounded by $O(L(B))$.

Summing all these together, we have the following lemma:

Lemma C.2. *The overall space consumption of our* BiT-DT *algorithm is bounded by $O(\sum_{(u,v)\in E} \min\{d(u), d(v)\})$.*

Proof of Theorem 4.2. Combining Lemmas C.1 and C.2, Theorem 4.2 follows. □