The VLDB Journal (2023) 32:257–281
https://doi.org/10.1007/s00778-022-00746-0

REGULAR PAPER

Accelerated butterfly counting with vertex priority on bipartite graphs

Kai Wang[1] · **Xuemin Lin**[1] · **Lu Qin**[2] · **Wenjie Zhang**[3] · **Ying Zhang**[2]

Received: 2 June 2021 / Revised: 22 March 2022 / Accepted: 1 April 2022 / Published online: 16 May 2022

Abstract
Bipartite graphs are of great importance in many real-world applications. Butterfly, which is a complete 2×2 biclique, plays a key role in bipartite graphs. In this paper, we investigate the problem of efficient counting the number of butterflies. The most advanced techniques are based on enumerating wedges which is the dominant cost of counting butterflies. Nevertheless, the existing algorithms cannot efficiently handle large-scale bipartite graphs. This becomes a bottleneck in large-scale applications. In this paper, instead of the existing layer-priority-based techniques, we propose a vertex-priority-based paradigm BFC-VP to enumerate much fewer wedges; this leads to a significant improvement of the time complexity of the state-of-the-art algorithms. In addition, we present cache-aware strategies to further improve the time efficiency while theoretically retaining the time complexity of BFC-VP. We also show that our proposed techniques can work efficiently in external and parallel contexts. Moreover, we study the butterfly counting problem on batch-dynamic graphs. Specifically, given a bipartite graph G and a batch-update of edges B, we aim to maintain the number of butterflies in G. To tackle this problem, fast vertex-priority-based algorithms are proposed with optimizations for reducing the computation of existing wedges in G. Our extensive empirical studies demonstrate that the proposed techniques significantly outperform the baseline solutions on real datasets.

Keywords Bipartite graph · Butterfly counting · Dynamic graph

1 Introduction

When modeling relationships between two different types of entities, bipartite graph arises naturally as a data model in many real-world applications [14,39]. For example, in online shopping services (e.g., Amazon and Alibaba), the purchase relationships between users and products can be modeled as a bipartite graph, where users form one layer, products form the other layer, and the links between users and productions represent purchase records as shown in Fig. 1. Other examples include author-paper relationships, actor-movie networks, etc.

✉ Wenjie Zhang
zhangw@cse.unsw.edu.au

Kai Wang
cskaelwang@gmail.com

Xuemin Lin
xuemin.lin@sjtu.edu.cn

Lu Qin
lu.qin@uts.edu.au

Ying Zhang
ying.zhang@uts.edu.au

1 Antai College of Economics and Management, Shanghai Jiao Tong University, Shanghai, China

2 University of Technology Sydney, Ultimo, NSW, Australia

3 University of New South Wales, Sydney, NSW, Australia

Since network motifs (i.e., repeated sub-graphs) are regarded as basic building blocks of complex networks [45], finding and counting motifs of networks/graphs is a key to network analysis. In unipartite graphs, there are extensive studies on counting and listing triangles in the literature [5,16,19,31,38,57,58,60–62]. In bipartite graphs, *butterfly* (i.e., 2×2 biclique) is the simplest bi-clique configuration with equal numbers of vertices of each layer (apart from the trivial single edge configuration) that has drawn reasonable attention recently [4,30,53,54,56,64,73]; for instance, Fig. 1 shows the record that Adam and Mark both purchased Balm and Doll forms a butterfly. In this sense, the butterfly can be viewed as an analogue of the triangle in a unipartite graph. Moreover, without butterflies, a bipartite graph will not have any community structure [4].

In this paper, we study the *butterfly counting* problem. Specifically, we aim to compute the total number of butterflies in a bipartite graph G, which is denoted by $\bowtie_G$. The importance of *butterfly counting* has been demonstrated in the literature of network analysis and graph theory. Below are some examples.

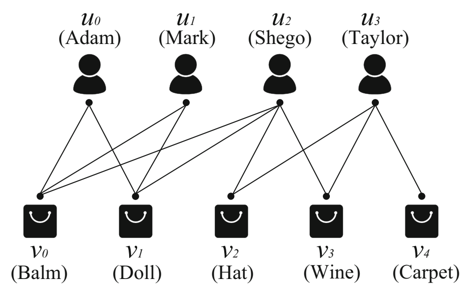


Fig. 1 A bipartite graph

Network measurement. The *bipartite clustering coefficient* [4,41,47,53] is a cohesiveness measurement of bipartite graphs. Given a bipartite graph G, its bipartite clustering coefficient equals $4 \times ⧖_G/\overline{\times}_G$, where $\overline{\times}_G$ is the total number of caterpillars (i.e., three-paths) in G. For example, (u_0, v_0, u_1, v_1) in Fig. 1 is a three-path. High bipartite clustering coefficient indicates localized closeness and redundancy in bipartite graphs [4,53]; for instance, in user-product networks, bipartite clustering coefficients can be used frequently to analyze the sale status for products in different categories. These statistics can also be used in `Twitter` network for internet advertising where the `Twitter` network is a bipartite graph consisting of Twitter users and the URLs they mentioned in their postings. Since $\overline{\times}_G$ can be easily computed in $O(m)$ time where m denotes the total number of edges in G [4], computing $⧖_G$ becomes a bottleneck in computing the clustering coefficient.

Summarizing inter-corporate relations. In a director-board network, two directors on the same two boards can be modeled as a butterfly. These butterflies can reflect inter-corporate relations [48–50]. The number of butterflies indicates the extent to which directors re-meet one another on two or more boards. A large butterfly counting number indicates a large number of inter-corporate relations and formal alliances between companies [53].
Computing k-wing in bipartite graphs. Counting the number of butterflies for each edge also has applications. For example, it is the first step to compute a k-wing [56] (or k-bitruss [67,68,73]) for a given k where k-wing is the maximal subgraph of a bipartite graph with each edge in at least k butterflies. Discovering such dense subgraphs is proved useful in many applications, e.g., community detection [25,26,69,72], word-document clustering [21], and viral marketing [24,42,65,71]. Given a bipartite graph G, the proposed algorithms [56,67,68,73] for k-wing computation are to first count the number of butterflies on each edge in G. After that, the edge with the lowest number of butterflies is iteratively removed from G until all the remaining edges appear in at least k butterflies.

Note that in real applications, butterfly counting may happen not only once in a graph. We may need to conduct such a computation against an arbitrarily specified subgraph. Indeed, there can exist a high demand for butterfly counting in large networks. However, the existing solutions cannot efficiently handle large-scale bipartite graphs. As shown in [54], on the `Tracker` network with 10^8 edges, their algorithm needs about 9000 s to compute $⧖_G$. Therefore, the study of efficient butterfly counting is imperative to support online large-scale data analysis. Moreover, some applications demand exact butterfly counting in bipartite graphs. For example, in k-wing computation, approximate counting does not make sense since the k-wing decomposition algorithm in [56] needs to iteratively remove the edges with the lowest number of butterflies; the number has to be exact.

Notably, dynamic graphs have attracted significant interest in recent research studies [11,15,28,44,46,61] since there can exist a large number of constant updates on graphs in real applications. To enable computational sharing and increase system throughout (i.e., average processing time per update) in practice, many existing studies consider processing the updates in a batch-dynamic way which handles updates as a set of batches [1,8,23,27,43]. However, there is no systematic study over the butterfly counting problem on batch-dynamic graphs in the literature. To fill this research gap, we investigate how to design efficient parallel butterfly counting algorithms for batch-dynamic settings in this paper. Specifically, given a bipartite graph G and a batch-update of edges B, we aim to maintain the number of butterflies in G.

State-of-the-art. Consider that there can be $O(m^2)$ butterflies in the worst case. Wang et al. in [64] propose an algorithm to avoid enumerating all the butterflies. It has two steps. At the first step, a layer is randomly selected. Then, the algorithm iteratively starts from every vertex u in the selected layer, computes the 2-hop reachable vertices from u, and for each 2-hop reachable vertex w, counts the number n_{uw} of times reached from u. At the second step, for each 2-hop reachable vertex w from u, we count the number of butterflies containing both u and w as $n_{uw}(n_{uw} - 1)/2$. For example, regarding Fig. 1, if the lower layer is selected, starting from the vertex v_0, vertices v_1, v_2, and v_3 are 2-hop reached 3 times, 1 time, and 1 time, respectively. Thus, there are C_3^2 [1] ($= 3$) butterflies containing v_0 and v_1 and no butterfly containing v_0 and v_2 (or v_0 and v_3). Iteratively, the algorithm first uses v_0 as the start-vertex, then v_1, and so on. Then, we add all the counts together; the added counts divided by two is the total number of butterflies.

Observe that the time complexity of the algorithm in [64] is $O(\sum_{u \in U(G)} deg_G(u)^2))$ if the lower layer $L(G)$ is chosen to have start-vertices, where $U(G)$ is the upper layer. Sanei

[1] C_n^k represents choosing k out of n.

et al. in [54] chooses a layer S such that $O(\sum_{v\in S} deg_G(v)^2)$ is minimized among the two layers.

Observation. In the existing algorithms [54,64], the dominant cost is at Step 1 that enumerates wedges to compute 2-hop reachable vertices and their hits. For example, regarding Fig. 1, we have to traverse 3 wedges, (v_0, u_0, v_1), (v_0, u_1, v_1), and (v_0, u_2, v_1) to get all the hits from v_0 to v_1. Here, in the wedge (v_0, u_0, v_1), we refer v_0 as the start-vertex, u_0 as the middle-vertex, and v_1 as the end-vertex. Continue with the example in Fig. 1, using u_2 as the middle-vertex, starting from v_0, v_1, and v_2, respectively, we need to traverse totally 6 wedges.

We observe that the choice of middle-vertices of wedges (i.e., the choice of start-vertices) is a key to improving the efficiency of counting butterflies. For example, consider the graph G with 2002 vertices and 3000 edges in Fig. 2a. In G, u_0 is connected with 1000 vertices (v_0 to v_{999}), v_{1000} is also connected with 1000 vertices (u_1 to u_{1000}), and for $0 \le i \le 999$, v_i is connected with u_{i+1}. The existing algorithms need to go through u_0 (or v_{1000}) as the middle-vertex if choosing $L(G)$ (or $U(G)$) to start. Therefore, regardless of whether the upper or the lower layer is selected to start, we have to traverse totally C^2_{1000} (= 499,500) plus 1000 different wedges by the existing algorithms [54,64].

Challenges. The main challenges of efficient butterfly counting are as follows.

1. Using high-degree vertices as middle-vertices may generate numerous wedges to be scanned. The existing techniques [54,64], including the layer-priority-based techniques [54], cannot avoid using unnecessary high-degree vertices as middle-vertices as illustrated earlier. Therefore, it is a challenge to effectively handle high-degree vertices.
2. Effectively utilizing CPU cache can often reduce the computation dramatically. Therefore, it is also a challenge to utilize CPU cache to speed up the counting of butterflies.
3. On batch-dynamic graphs, we need to handle a batch of updates on the original graph, which can be very large. Thus, it is a challenge to explore possible sharing opportunities and reduce the computation that does not lead to any new butterflies.

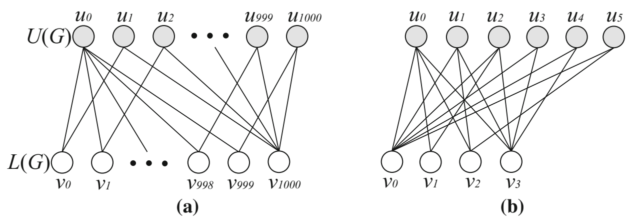


Fig. 2 Two example bipartite graphs for illustrating the butterfly counting algorithms

Our approaches. To address Challenge 1, instead of the existing layer-priority-based algorithm, we propose a vertex-priority-based algorithm BFC-VP that can effectively handle hub vertices (i.e., high-degree vertices). To avoid over-counting or miss-counting, we propose that for each edge (u, v), the new algorithm BFC-VP uses the vertex with a higher degree as the start-vertex so that the vertex with a lower degree will be used as the middle-vertex. Specifically, the BFC-VP algorithm will choose one end vertex of an edge (u, v) as the start-vertex, say u, according to its priority. Note that the vertex priority is a total ordering of vertices, and we use degree-based priority in this paper. A higher degree indicates a higher priority, and the tie is broken by the ID of vertices. For example, regarding Fig. 2a, the BFC-VP algorithm will choose u_0 and v_{1000} as start-vertices; consequently, only 2000 wedges in total will be scanned by our algorithm compared with 500, 500 different wedges generated by the existing algorithms as illustrated earlier. Once all edges from the starting vertex u are exhausted, BFC-VP moves to another edge. This is the main idea of our BFC-VP algorithm.

As a result, the time complexity of our BFC-VP algorithm is $O(\sum_{(u,v)\in E(G)} min\{deg_G(u), deg_G(v)\})$, which is in general significantly lower than the time complexity of the state-of-the-art algorithm in [54] (i.e., $O(\min\{\sum_{v\in U(G)} deg_G(v)^2, \sum_{v\in L(G)} deg_G(v)^2\})$). Note that the time complexity of BFC-VP is also bounded by $O(\alpha\cdot m)$, where α is the arboricity of G [17].

In the BFC-VP algorithm, there are $O(n)$ accesses of start-vertices because we need to explore every vertex as a start-vertex only once, $O(m)$ accesses of middle-vertices and $O(\sum_{(u,v)\in E(G)} min\{deg_G(u), deg_G(v)\})$ accesses of end-vertices in the processed wedges. Thus, the number of accesses to end-vertices is dominant. Given that the cache miss latency takes a big part of the memory access time [3], improving the CPU cache performance when accessing the end-vertices becomes a key issue. Our second algorithm, the cache-aware algorithm BFC-VP^{++}, aims to improve the CPU cache performance of BFC-VP by having high-degree vertices as end-vertices to enhance the locality while retaining the total number of accesses of end-vertices (thus, retain the time complexity of the BFC-VP algorithm). Consequently, BFC-VP^{++} proposes to request the end-vertices to be prioritized in the same way as the start-vertices in the BFC-VP algorithm.

For example, considering the graph in Fig. 2b, we have $p(v_0) > p(v_3) > p(u_0) > p(v_2) > p(v_1)$ according to their degrees where $p(v)$ denotes the priority of a vertex v. In this example, starting from v_0 to v_3, going through u_0, BFC-VP needs to process 5 wedges using u_0 as the middle-vertex (i.e., (v_0, u_0, v_1), (v_0, u_0, v_2), (v_0, u_0, v_3), (v_3, u_0, v_1) and (v_3, u_0, v_2)), and there are 3 vertices, v_1, v_2 and v_3 that need to be performed as end-vertices. Note that these are the only 5 wedges using u_0 as the middle-vertex since $p(u_0) > p(v_2) > p(v_1)$. Regarding the same example, BFC-VP^{++} also needs to process exactly 5 wedges with u_0 as the middle-vertex, (v_1, u_0, v_0), (v_1, u_0, v_3), (v_2, u_0, v_0), (v_2, u_0, v_3) and (v_3, u_0, v_0); however, only 2 vertices, v_0 and v_3, are performed as end-vertices.

We also propose the cache-aware reordering strategy to improve the cache performance by storing high-priority (more frequently accessed) end-vertices together to reduce the cache-miss [70]. Considering the example in Fig. 2b, BFC-VP^{++} will store v_0 and v_3 together after reordering.

To handle batch-dynamic graphs, we propose efficient algorithms by identifying the affected scope (subgraph) of the batch-update. Then, we can also utilize our vertex-priority-based techniques to accelerate the computation process. In addition, we categorize the new butterflies into different cases and propose effective pruning techniques to further reduce the computation (of old wedges on the original graph) that does not lead to any new butterflies.

Contribution. We summarize the principal contributions of this paper as follows.

- We propose a novel vertex-priority-based algorithm BFC-VP to count the butterflies that reduce the time complexities of the state-of-the-art algorithms significantly in both theory and practice.
- We propose a novel cache-aware butterfly counting algorithm BFC-VP^{++} by adopting cache-aware strategies to BFC-VP. Compared with BFC-VP, the BFC-VP^{++} algorithm achieves a better CPU cache performance.
- We can replace the exact counting algorithm in the approximate algorithm [54] by our exact counting algorithm for a speedup.
- We also present an external-memory algorithm and a parallel algorithm for butterfly counting.
- This is also the first work to study butterfly counting on batch-dynamic graphs. We propose a work-efficient parallel algorithm to solve the problem.
- We conduct comprehensive experimental evaluations on real datasets. It shows that our proposed algorithms BFC-VP and BFC-VP^{++} outperform the existing algorithms by up to two orders of magnitude. For instance, on the `Tracker` dataset, the BFC-VP^{++} algorithm can count 10^{12} butterflies in 50 s, and the state-of-the-art butterfly counting algorithm [54] runs about 9000 s. In addition, our advanced algorithm BFCB-IG^{+} for batch-dynamic butterfly counting is up to 2 orders of magnitude faster than the baseline algorithm.

Organization We organize the rest of the paper as follows. In Sect. 2, we show the preliminaries. Section 3 discusses the existing algorithms BFC-BS and BFC-IBS. We introduce the BFC-VP algorithm in Sect. 4. Section 5 explores cache-awareness. Section 6 extends our algorithms to count butterflies against each edge, the parallel execution of our proposed algorithms, and the external memory solution. Section 7 presents our algorithms for batch-dynamic butterfly counting. Section 8 reports experimental results. Section 9 reviews the related works. Section 10 presents the conclusion.

2 Preliminaries

We define our problem over a bipartite graph $G(V = (U, L), E)$. Here $U(G)$ contains all the upper layer vertices, $L(G)$ contains all the lower layer vertices, and $E(G)$ is the edge set. Note that $U(G) \cap L(G) = \emptyset$. An edge connecting two vertices u and v is represented as (u, v) (or (v, u)). $N_G(u)$ is the neighbor set of a vertex u in G, and u's degree is represented as $deg_G(u) = |N_G(u)|$. The 2-hop neighbor set of u (i.e., the set of vertices which are exactly two edges away from u) is denoted as $2hop_G(u)$. Note that we assume each vertex in G has an id, and the IDs of the vertices in $U(G)$ are always higher than that of the vertices in $L(G)$. m and n are used to represent the number of edges and vertices in G. We use $|G|$ to denote the size of G, where $|G| = m+n$.

Definition 1 (*Wedge*) Consider a bipartite graph G, and three vertices $u, v, w \in V(G)$. A wedge (u, v, w) is a path starting from u, going through v and ending at w. u, v, and w are called the start-, the middle-, and the end-vertex in the wedge (u, v, w), respectively.

Definition 2 (*Butterfly*) Consider a bipartite graph G and four vertices $u, w \in U(G)$ and $v, x \in L(G)$. A butterfly $[u, v, w, x]$ is a complete bipartite subgraph (i.e., 2×2-biclique) induced by u, v, w, x.

The total number of butterflies containing a vertex u and an edge e are denoted as $\bowtie_u$ and $\bowtie_e$, respectively. In addition, the count of butterflies in G is denoted as $\bowtie_G$.

Problem statement Given a bipartite graph G, the *butterfly counting* problem is to compute $\bowtie_G$.

3 Existing algorithms

Here, we discuss the two existing algorithms, the baseline butterfly counting algorithm BFC-BS [64] and the improved baseline butterfly counting algorithm BFC-IBS [54]. As discussed earlier, both algorithms are based on enumerating wedges. Lemma 1 [64] is a key to the two algorithms.

Lemma 1 *In a bipartite graph G, the following equations hold:*

$$\bowtie_u = \sum_{w \in 2hop_G(u)} \binom{|N_G(u) \cap N_G(w)|}{2} \quad (1)$$

$$\bowtie_G = \frac{1}{2} \sum_{u \in U(G)} \bowtie_u = \frac{1}{2} \sum_{v \in L(G)} \bowtie_v \quad (2)$$

In fact, BFC-IBS has the same framework as BFC-BS and improves BFC-BS in two aspects: (1) pre-choosing the layer of start-vertices to achieve a lower time complexity; (2) using a hash map to speed up the implementation. We show the details of BFC-IBS in Algorithm 1.

```
Algorithm 1: BFC- IBS
   Input: $G$
   Output: $\bowtie_G$
1  $\bowtie_G \leftarrow 0$
2  $S \leftarrow U(G)$
3  if $\sum_{u \in U(G)} deg_G(u)^2 < \sum_{v \in L(G)} deg_G(v)^2$ then
4      $S \leftarrow L(G)$
5  foreach $u \in S$ do
6      $wedge\_cnt \leftarrow$ a hashmap initialized with zero
7      foreach $v \in N_G(u)$ do
8          foreach $w \in N_G(v) : w.id > u.id$ do
9              $wedge\_cnt(w) + +$
10     foreach $w \in wedge\_cnt$ do
11         if $wedge\_cnt(w) > 1$ then
12             $\bowtie_G \leftarrow \bowtie_G + \binom{wedge\_cnt(w)}{2}$
13 return $\bowtie_G$
```

Note that to avoid counting a butterfly twice, for each middle-vertex $v \in N_G(u)$ and the corresponding end-vertex $w \in N_G(v)$, BFC-IBS processes the wedge (u, v, w) only if $w.id > u.id$; consequently, in Algorithm 1 we do not need to use the factor $\frac{1}{2}$ in Equation 2 of Lemma 1.

Note that the BFC-BS algorithm has the time complexity of $O(\sum_{v \in L(G)} deg_G(v)^2)$ if starting from the layer $U(G)$, while the time complexity of BFC-IBS is $O(min\{\sum_{u \in U(G)} deg_G(u)^2, \sum_{v \in L(G)} deg_G(v)^2\})$.

4 Algorithm by vertex priority

In BFC-BS and BFC-IBS, the time complexity is related to the total number of 2-hop neighbors visited (i.e., the total number of wedges processed). When starting from one vertex layer, the number of processed wedges is decided by the sum of degree squares of middle-vertices of the other layer. If all the vertices with low degrees are distributed in one vertex layer as middle-vertices, BFC-IBS can just start from the vertices in the other layer and obtain a much lower computation cost.

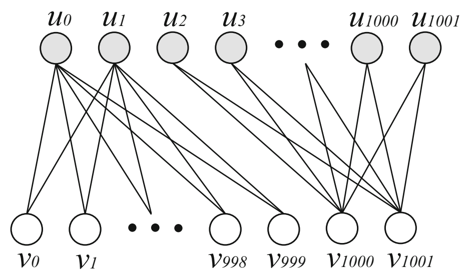


Fig. 3 A bipartite graph containing hub vertices u_0, u_1, v_{1000} and v_{1001}

However, when there are vertices with high degrees (i.e., *hub vertices*) exist in both layers, which is not uncommon in real datasets (e.g., `Tracker` dataset), choosing which layer to start cannot achieve a better performance. For example, consider the graph G with 2002 vertices and 4000 edges in Fig. 3, where u_0 and u_1 are connected with 1, 000 vertices (v_0 to v_{999}), v_{1000} and v_{1001} are also connected with 1000 vertices (u_2 to u_{1001}). In this example, choosing either of the two layers still needs to go through hub vertices, $u_0, u_1 \in U(G)$ or $v_{1000}, v_{1001} \in L(G)$.

Optimization strategy. Clearly, $[u_0, v_0, u_1, v_1]$ in Fig. 3 can be constructed in the following two ways: (1) by the wedges (u_0, v_0, u_1) and (u_0, v_1, u_1), or (2) by the wedges (v_0, u_0, v_1) and (v_0, u_1, v_1). Consequently, a hub vertex (e.g., u_0 in Fig. 3) may not always necessary to become a middle-vertex in a wedge for the construction of a butterfly. Thus, it is possible to design an algorithm which can avoid using hub vertices unnecessarily as middle-vertices. To achieve this objective, we introduce the vertex-priority-based butterfly counting algorithm BFC-VP which runs in a vertex level (i.e., choosing which vertex to be processed as the start-vertex) rather than a layer level (i.e., choosing which vertex-layer to be processed as the start-layer).

Given a graph G, BFC-VP first assigns a *priority* to each vertex $u \in V(G)$.

Definition 3 (*Priority*) Consider a bipartite graph G. The vertex priority is a total ordering of the vertices in G. Specifically, the priority $p(u)$ of a vertex $u \in V(G)$ is an integer where $p(u) \in [1, n]$. Given $u, v \in V(G)$, $p(u) \neq p(v)$ if $u \neq v$.

According to the concept of priority, we can always construct a butterfly from two wedges (u, v, w) and (u, x, w) where the start-vertex u has a higher priority than the middle-vertices v and x. This is because we can always find a vertex that has the highest priority and connects to two vertices with lower priorities in a butterfly.

Based on the above observation, the BFC-VP algorithm can get all the butterflies by only processing the wedges where the priorities of start-vertices are higher than the priorities of middle-vertices. In this way, the algorithm BFC-VP will avoid processing the wedges where middle-vertices have higher priorities than start-vertices (e.g., (v_0, u_0, v_1) in Fig. 3 if we consider that a higher vertex degree indicates a higher priority). In addition, in order to avoid duplicate counting, another constraint should also be satisfied in BFC-VP: BFC-VP only processes the wedges where start-vertices have higher priorities than end-vertices. To avoid processing unnecessary wedges in the implementation, we sort the neighbors of vertices in ascending order of their priorities. Then we can early terminate the processing once we meet an end-vertex that has higher priority than the start-vertex (or meet a middle-vertex that has higher priority than the start-vertex). We show the details of the BFC-VP algorithm in Algorithm 2.

Algorithm 2: BFC- VP

```
   Input: $G$
   Output: $\bowtie_G$
1  calculate the priority for each vertex in $V(G)$
2  for each vertex $u$, sort $N(u)$ according to their priorities
3  $\bowtie_G \leftarrow 0$
4  foreach vertex $u \in V(G)$ do
5      $wedge\_cnt \leftarrow$ a hashmap initialized with zero
6      foreach vertex $v \in N_G(u) : p(v) < p(u)$ do
7          foreach vertex $w \in N_G(v) : p(w) < p(u)$ do
8              $wedge\_cnt(w)$++
9      foreach vertex $w \in wedge\_cnt$: $wedge\_cnt(w) > 1$ do
10         $\bowtie_G \leftarrow \bowtie_G + \binom{wedge\_cnt(w)}{2}$
11 return $\bowtie_G$
```

Firstly, BFC-VP assigns a priority to each vertex $u \in V(G)$ according to Definition 3 and sort the neighbors of u. After that, BFC-VP processes wedges from each start-vertex u (Line 4). For each middle-vertex $v \in N_G(u)$, it processes v if it satisfies $p(v) < p(u)$. Then, it only processes $w \in N_G(v)$ which satisfies $p(w) < p(u)$ to avoid duplicate counting. After that, the value $|N_G(u) \cap N_G(w)|$ can be obtained for u and w which is equal to $wedge\_cnt(w)$. Then, according to Lemma 1, BFC-VP computes $\bowtie_G$. Finally, we return $\bowtie_G$.

Correctness and complexity analysis of the BFC-VP algorithm. Below we show theoretical analysis of the BFC-VP algorithms.

Theorem 1 *The BFC-VP algorithm correctly solves the butterfly counting problem.*

Proof We prove that BFC-VP correctly computes $\bowtie_G$ for a bipartite graph G. A butterfly can always be constructed from two different wedges with the same start-vertex and the same end-vertex. Thus, we only need to prove that each butterfly in G will be counted exactly once by BFC-VP. Given a butterfly $[x, u, v, w]$, we assume x has the highest priority. The vertex priority distribution must be one of the three situations

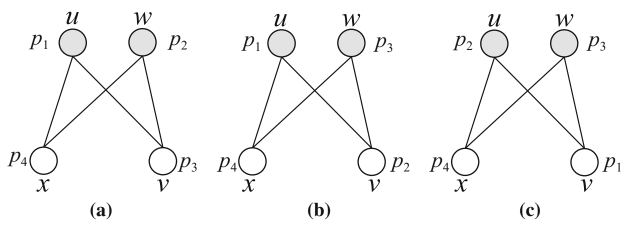


Fig. 4 Assume $p_4 > p_3 > p_2 > p_1$

as shown in Fig. 4 (the other situations can be transformed into the above by symmetry), where p_i is the corresponding vertex priority. Regarding the case in Fig. 4a, b, or c, BFC-VP only counts the butterfly $[x, u, v, w]$ once from the wedges (x, u, v) and (x, w, v). Thus, we can prove that BFC-VP correctly solves the *butterfly counting* problem. □

Theorem 2 *The time complexity of* BFC-VP *is* $O(\sum_{(u,v)\in E(G), p(u)>p(v)} deg_G(v))$.

Proof Algorithm 2 has two phases: initializing in the first phase and computing $\bowtie_G$ in the second phase. The time complexity of the first phase is $O(n + m)$. Firstly, we need $O(m)$ to get the degrees of vertices and $O(n)$ time to get the priorities by sorting the vertices using bin sort [37]. Secondly, we need $O(m)$ time to sort the neighbors of vertices in ascending order of their priorities. To achieve this, we generate a new empty neighbor list $T(u)$ for each vertex u. Then, we process the vertex with lower priority first and for each vertex u and its neighbor v, we put u into $T(v)$. Finally, the neighbors of vertices are ordered in T.

The time cost of the second phase is bounded by the number of wedge counting operations executed in Algorithm 2 Line 8 (i.e., the number of wedges traversed in BFC-VP). According to the processing rule of BFC-VP, the wedge counting operations consume $O(deg(v))$ time for each edge $(u, v) \in E(G)$ with $p(u) > p(v)$. This is because only the wedges where the priorities of middle-vertices are lower than the priorities of start-vertices are processed in BFC-VP. Hence, BFC-VP needs $O(\sum_{(u,v)\in E(G), p(u)>p(v)} deg_G(v))$ time in total, and this theorem holds. □

According to the above analysis, how to order the vertices can affect the complexity of the algorithm. In this paper, we propose using the degree-based priority to achieve both good practical and theoretical results.

Definition 4 (*Degree priority*) Consider a bipartite graph G. The degree priority $p_d(u)$ of a vertex $u \in V(G)$ is an integer where $p_d(u) \in [1, n]$. Given $u, v \in V(G)$, $p_d(u) > p_d(v)$ if

– $deg_G(u) > deg_G(v)$, or
– $deg_G(u) = deg_G(v)$, $u.id > v.id$.

Based on Definition 4, we can have the following theorem.

Theorem 3 *The time complexity of* BFC-VP *is* $O(\sum_{(u,v)\in E(G)} min\{deg_G(u), deg_G(v)\}) = O(\alpha \cdot m)$, *if the degree priority is applied.*

Proof Since the degree priority is applied, the time complexity of BFC-VP is $O(\sum_{(u,v)\in E(G), p_d(u)>p_d(v)} deg_G(v)) = O(\sum_{(u,v)\in E(G)} min\{deg_G(u), deg_G(v)\})$. According to [17], the time complexity of BFC-VP can be simplified to $O(\alpha \cdot m)$, where α is the arboricity of G. □

In the following parts, we simply call the degree priority as the priority and use $p(u)$ to denote the degree priority when the context is clear.

Theorem 4 *The space complexity of* BFC-VP *is* $O(m)$.

Proof In Algorithm 2, we need $O(m)$ space to store the graph structure and $O(n)$ space to store the arrays for the priority of vertices and counting the number of wedges. Thus, the space cost of the BFC-VP algorithm is bounded by $O(m)$. □

Lemma 2 *In a bipartite graph G, the following equation holds:*

$$\sum_{(u,v)\in E(G)} min\{deg_G(u), deg_G(v)\} \leq min\{\sum_{u\in U(G)} deg_G(u)^2, \sum_{v\in L(G)} deg_G(v)^2\} \tag{3}$$

The equality happens if and only if one of the following two conditions is satisfied: (1) for every edge $(u, v) \in E(G)$ *and* $u \in U(G)$, $deg_G(u) \leq deg_G(v)$; *(2) for every edge* $(u, v) \in E(G)$ *and* $u \in U(G)$, $deg_G(v) \leq deg_G(u)$.

Proof Given a bipartite graph G, since there are $deg_G(u)$ edges attached to a vertex u, we can get that:

$$\sum_{u\in U(G)} deg_G(u)^2 = \sum_{(u,v)\in E(G), u\in U(G)} deg_G(u) \geq \sum_{(u,v)\in E(G)} min\{deg_G(u), deg_G(v)\} \tag{4}$$

Similarly,

$$\sum_{v\in L(G)} deg_G(v)^2 = \sum_{(u,v)\in E(G), u\in U(G)} deg_G(v) \geq \sum_{(u,v)\in E(G)} min\{deg_G(u), deg_G(v)\} \tag{5}$$

Thus, we can prove that Eq. (3) holds. The condition of equality can be easily proved by contradiction which is omitted here. □

From Lemma 2, we can get that BFC-VP improves the time complexity of BFC-IBS. Now we illustrate how BFC-VP efficiently handles the hub-vertices compared with BFC-IBS using the following example.

Example 1 Consider the example in Fig. 3.

BFC-VP first assigns a priority to each vertex in G where $p(u_1) > p(u_0) > p(v_{1001}) > p(v_{1000}) > p(u_{1001}) > p(u_{1000}) > \cdots > p(v_1) > p(v_0)$. Starting from u_1, BFC-VP needs to process 1000 wedges ending at u_0. Similarly, starting from v_{1001}, BFC-VP needs to process 1000 wedges ending at v_{1000}. No other wedges need to be processed by BFC-VP. In total, BFC-VP needs to process 2000 wedges.

BFC-IBS processes each vertex $u \in U(G)$ as start-vertex. Starting from u_0, BFC-IBS needs to process 1000 wedges ending at u_1. Starting from u_1, no wedges need to be processed. In addition, starting from the vertices in $\{u_2, u_3, \ldots, u_{1001}\}$, BFC-IBS needs to process 999,000 wedges. In total, BFC-IBS needs to process 1,000,000 wedges.

5 Cache-aware techniques

As discussed in Sect. 1, below is the breakdown of memory accesses to vertices required when processing the wedges in the BFC-VP algorithm: $O(n)$ accesses of start-vertices, $O(m)$ accesses of middle-vertices, and $O(\sum_{(u,v)\in E(G)} min\{deg_G(u), deg_G(v)\})$ accesses of end-vertices. Thus, the total access of end-vertices is dominant. For example, by running the BFC-VP algorithm on `Tracker` dataset, there are about 6×10^9 accesses of end-vertices, while the accesses of start-vertices and middle-vertices are only 4×10^7 and 2×10^8, respectively. Since the cache miss latency takes a big part of the memory access time [3], we try to improve the CPU cache performance when accessing the end-vertices.

Because the CPU cache is hard to control in algorithms, a general approach to improve the CPU cache performance is storing frequently accessed vertices together. Suppose there is a buffer *BF* that is partitioned into a *low-frequency area LFA* and a *high-frequency area HFA* as shown in Fig. 5. The vertices are stored in *BF* and only a limited number of vertices are stored in *HFA*. For an access of the end-vertex w, we compute $miss(w)$ by the following equation:

$$\text{miss}(w) = \begin{cases} 1, & \text{iff. } w \in LFA, \\ 0, & \text{iff. } w \in HFA. \end{cases} \tag{6}$$

We want to minimize F which is computed by:

$$F = \sum_{(u,v,w)\in W} miss(w) \tag{7}$$

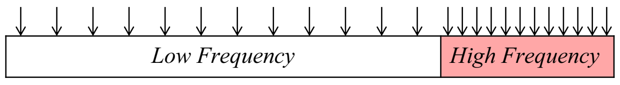


Fig. 5 The buffer *BF*

Here, W is the set of processed wedges of an algorithm.

Since F can only be derived after finishing the algorithm, the minimum value of F cannot be pre-computed. We present two strategies that aim to decrease F:

- Cache-aware wedge processing which performs more high-priority vertices as end-vertices, while retaining the total number of accesses of end-vertices (thus, the same time complexity of BFC-VP). Doing this will enhance the access locality.
- Cache-aware graph reordering which stores vertices with high-priority together in *HFA*.

5.1 Cache-aware wedge processing

Issues in wedge processing of BFC-VP. In BFC-VP, the processing rule restricts the priorities of end-vertices should be lower than the priorities of start-vertices in the processed wedges. Because of that, the accesses of end-vertices exhibit bad locality (i.e., not clustered in memory). For example, by counting the accesses of end-vertices over `Tracker` dataset, as shown in Fig. 6a, 79% of total accesses are accesses of low-degree vertices (i.e., degree < 500) while the percentage of high-degree vertices (i.e., degree > 2000) accesses is only 9% in BFC-VP. Since the locality of accesses is a key aspect of improving the CPU cache performance, we explore whether the locality of end-vertex-accesses can be improved. With the total access of end-vertices remaining unchanged, we hope the algorithm can access more high-degree vertices as end-vertices. In that manner, the algorithm will have more chance to request the same memory location repeatedly and the accesses of *HFA* are more possible to increase (i.e., F is more possible to decrease).

New wedge processing strategy. Based on the above observation, we present a new wedge processing strategy: processing the wedges where the priorities of end-vertices are higher than the priorities of middle-vertices and start-vertices. We name the algorithm using this new strategy as BFC-VP$^+$. BFC-VP$^+$ will perform more high-priority vertices as the end-vertices than BFC-VP because of the restriction of priorities of end-vertices. For example, considering the graph in Fig. 2b, we have $p(v_0) > p(v_3) > p(u_0) > p(v_2) > p(v_1)$ according to their degrees. We analyze the processed wedges starting from v_0 to v_3, going through u_0. BFC-VP needs to process 5 wedges (i.e., (v_0, u_0, v_1), (v_0, u_0, v_2), (v_0, u_0, v_3), (v_3, u_0, v_1) and (v_3, u_0, v_2)) and 3 vertices (i.e., v_1, v_2 and v_3) are performed as end-vertices. Utilizing the new wedge processing strategy, as shown in Fig. 2b, the number of processed wedges of BFC-VP$^+$ is still 5 (i.e., (v_1, u_0, v_0), (v_1, u_0, v_3), (v_2, u_0, v_0), (v_2, u_0, v_3) and (v_3, u_0, v_0)) but only 2 vertices with high-priorities (i.e., v_0 and v_3) are performed as end-vertices. Thus, the number of accessing different end-vertices is decreased from 3 to 2 (i.e., the accesses exhibit better locality). Also as shown in Fig. 6b, after applying the new wedge processing strategy, the percentage of accesses of high-degree vertices (i.e., degree > 2000) increases from 9 to 81% on `Tracker` dataset.

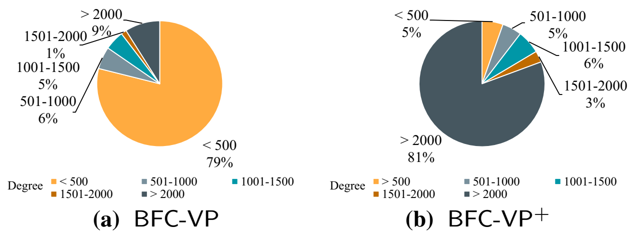


Fig. 6 The degree distribution of the end-vertex-accesses on `Tracker`

Analyzing the new wedge processing strategy. Although the new wedge processing strategy can improve the CPU cache performance of BFC-VP, there are two questions that arise naturally: (1) whether the number of processed wedges is still the same as BFC-VP; (2) whether the time complexity is still the same as BFC-VP after utilizing the new wedge processing strategy. We denote the set of processed wedges of BFC-VP as W_{vp} and the set of processed wedges of BFC-VP$^+$ as W_{vp^+}, and the following lemma holds.

Lemma 3 $|W_{vp}| = |W_{vp^+}|$.

Proof For a wedge $(u, v, w) \in W_{vp}$, it always satisfies $p(u) > p(v)$ and $p(u) > p(w)$ according to Algorithm 2. For a wedge $(u, v, w) \in W_{vp^+}$, it always satisfies $p(w) > p(v)$ and $p(w) > p(u)$ according to the new wedge processing strategy. In addition, $p(u)$ is unique for each vertex u and the new wedge processing strategy does not change $p(u)$ of u. Thus, for each wedge $(u, v, w) \in W_{vp}$, we can always find a wedge $(w, v, u) \in W_{vp^+}$. Similarly, for each wedge $(u, v, w) \in W_{vp^+}$, we can always find a wedge $(w, v, u) \in W_{vp}$. Therefore, we prove that $|W_{vp}| = |W_{vp^+}|$. □

Since no duplicate wedges are processed, based on the above lemma, BFC-VP$^+$ will process the same number of wedges with BFC-VP. However, if only applying this strategy, when going through a middle-vertex, we need to check all its neighbors to find the end-vertices which have higher priorities than the middle vertex and the start-vertex. The time complexity will increase to $O(\sum_{u \in V(G), v \in N_G(u)} deg_G(u)\, deg_G(v))$ because each middle-vertex v has $deg_G(v)$ neighbors. In order to reduce the time complexity, for each vertex, we need to sort the neighbors in descending order of their priorities. After that, when dealing with a middle-vertex, we can early terminate the priority checking once we meet a neighbor which has a lower priority than the middle-vertex or the start-vertex.

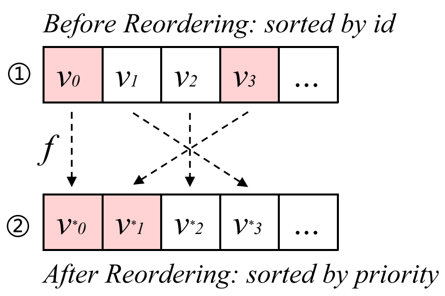

Before Reordering: sorted by id
①
v_0
v_1
v_2
v_3
...
f
②
v^*_0
v^*_1
v^*_2
v^*_3
...
After Reordering: sorted by priority

Fig. 7 Illustrating the cache-aware graph reordering

5.2 Cache-aware graph reordering

Motivation. After utilizing the cache-aware wedge processing strategy, end-vertices are mainly high-priority vertices. Generally, vertices are sorted by their ids when storing in the buffer. Figure 7 shows accesses of the buffer when processing end-vertices (i.e., v_0 and v_3) starting from v_0 to v_3 and going through u_0 in Fig. 2b by BFC-VP. We can see that although end-vertices are mostly high-priority vertices, the distance between two end-vertices (e.g., v_0 and v_3) can be very long. This is because many low-priority vertices are stored in the middle of high-priority vertices. In addition, real graphs usually follow power-law distributions which do not contain too many vertices with high priorities (degrees). For example, in the `Tracker` dataset with about 40,000,000 vertices, there are only 10,338 vertices with degree $\geq$ 1000, and only 1% vertices (400,000) with degree $\geq$ 37. Motivated by the above observations, we propose the graph reordering strategy which can further improve the cache performance.

Graph reordering strategy. The main idea of the graph reordering strategy is reordering the given bipartite graph G into a reordering graph G^* using a 1 to 1 bijective function f. The reordering graph G^* is defined as follows.

Definition 5 (*Reordering graph*) For a bipartite graph G, the reordering graph G^* is defined as: $G^* \leftarrow reordering(G, f)$, where f is a bijection from $E(G)$ to $E(G^*)$. For each $e = (u, v) \in E(G)$, $e^* = (u^*, v^*) = f(e)$ where $u^* \in U(G^*)$, $v^* \in L(G^*)$, and $u^*.id = rankU(u)+l$, $v^*.id = rankL(v)$. $rankU(u) \in [0, r-1]$ ($rankL(v) \in [0, l-1]$) denotes the rank of the priority of $u \in U(G)$ (the rank of the priority of $v \in L(G)$).

Note that our linear graph reordering uses a 1 to 1 bijective function to relabel the vertex-IDs which does not change the graph structure. Thus, the number of vertices and edges are both unchanged after reordering. After reordering the original graph G into the reordering graph G^*, the vertices with high priorities will be stored together. In this manner, we can store more high-priority vertices consecutively in *HFA*. Figure 7 illustrates the idea of graph reordering using the example in Fig. 2b. After obtaining the reordering graph G^*, we can see that the distance between two high-priority end-vertices becomes much shorter, e.g., the distance between v_1^* and v_2^* is 1 while the distance between v_0 and v_3 before reordering is 3. In the experiments, we prove that the algorithms applying with the graph reordering strategy achieve a much lower cache miss ratio than BFC-VP.

5.3 Putting cache-aware strategies together

The BFC-VP^{++} algorithm. Putting the above strategies together, we show the details of the algorithm BFC-VP^{++} in Algorithm 3. BFC-VP^{++} first generates a reordering graph G^* according to Definition 5 and for each vertex $u^* \in V(G^*)$, we sort its neighbors. Then, BFC-VP^{++} finds $N_{G^*}(u^*)$ for each vertex $u^* \in V(G^*)$. For each vertex $v^* \in N_{G^*}(u^*)$, we find $w^* \in N_{G^*}(v^*)$ with $p(w^*) > p(u^*)$ and $p(w^*) > p(v^*)$ (Lines 5 - 12). After Lines 6 - 12, we have $|N_G(u^*) \cap N_G(w^*)|$ for the start-vertex u^* and the end-vertex $w^* \in 2hop_G(u^*)$. Finally, we compute $\bowtie_G$ (Lines 13–14).

Algorithm 3: BFC- VP^{++}
```
   Input: $G$
   Output: $\bowtie_G$
1  $\bowtie_G \leftarrow 0$
2  $G^* \leftarrow reordering(G, f)$ // Definition 5
3  calculate the priority for each vertex in $V(G^*)$
4  for each $u^* \in V(G^*)$, sort $N(u^*)$ according to their priorities
5  foreach vertex $u^* \in V(G^*)$ do
6      $wedge\_cnt \leftarrow$ a hashmap initialized with zero
7      foreach $v^* \in N_{G^*}(u^*)$ do
8          foreach $w^* \in N_{G^*}(v^*) : p(w^*) > p(u^*)$ do
9              if $p(w^*) > p(v^*)$ then
10                 $wedge\_cnt(w^*)$++
11             else
12                 break
13     foreach vertex $w^* \in wedge\_cnt$: $wedge\_cnt(w^*) > 1$ do
14         $\bowtie_G \leftarrow \bowtie_G + \binom{wedge\_cnt(w^*)}{2}$
15 return $\bowtie_G$
```

Theorem 5 BFC-VP^{++} *solves the butterfly counting problem correctly.*

Proof We prove that BFC-VP^{++} correctly computes $\bowtie_G$ for a bipartite graph G. Since the graph reordering strategy just renumbers the vertices, it does not affect the structure of G. Given a butterfly $[x, u, v, w]$, we assume x has the highest priority. We only need to prove that BFC-VP^{++} will count exactly once for each butterfly in Fig. 4. Regarding the case in Fig. 4a, b, or c, BFC-VP^{++} only counts the butterfly $[x, u, v, w]$ once from the wedges (v, u, x) and (v, w, x). Thus, we can get that the BFC-VP^{++} algorithm correctly solves the *butterfly counting* problem. □

Theorem 6 *The time complexity of* BFC-VP^{++} *is* $O(\alpha \cdot m)$.

Proof Algorithm 3 has two phases including the initialization phase and $\bowtie_G$ computation phase. In the first phase, similar to BFC-VP, the algorithm needs $O(n+m)$ time to compute the priority number, sort the neighbors of vertices and compute the reordering graph. Secondly, we analyze how many wedge counting operations executed by BFC-VP^{++} in Algorithm 3 Line 10 (i.e., the number of wedges processed) as follows. In BFC-VP^{++}, we only need to process the wedges where the degree of the end-vertex is higher than or equal to the middle-vertex. Thus, the wedge counting operations consume $O(deg_G(v))$ time to process each edge $(u, v) \in E(G)$ connecting an end-vertex u and a middle-vertex v with $deg(u) \geq deg(v)$. Hence, BFC-VP^{++} needs $O(\sum_{(u,v)\in E(G)} min\{deg_G(u), deg_G(v)\}) = O(\alpha \cdot m)$ time in total, and this theorem holds. □

Theorem 7 *The space cost of* BFC-VP^{++} *is* $O(m)$.

Proof In Algorithm 3, we need $O(m)$ space to store the graph structure and $O(n)$ space to store the arrays for the priority of vertices and counting the number of wedges. The graph reordering process also needs $O(m)$ space for the reordering graph. Thus, the space cost of the BFC-VP algorithm is bounded by $O(m)$. □

Remark Note that our algorithms (i.e., BFC-VP and BFC-VP^{++}) are able to output all the butterflies in a compact format in $O(\alpha \cdot m)$ time. For instance, we can use $[u_i, u_j, \{v_x, v_y, v_z\}]$ to represent three butterflies that contain u_i and u_j with the other two vertices chosen from $\{v_x, v_y, v_z\}$. If we want to enumerate all the butterflies one by one, it needs an additional $O(\bowtie_G)$ time based on this compact data structure. Here, $O(\bowtie_G)$ denotes the total number of butterflies in G, which can reach $O(m^2)$ in the worst-case.

6 Handling other cases

In this section, firstly, we extend our algorithms to compute $\bowtie_e$ for each e in G. Secondly, we extend our algorithms to parallel algorithms. Thirdly, we introduce the external memory butterfly counting algorithm to handle large graphs with limited memory size.

6.1 Counting the butterflies for each edge

For an edge e in G, the following equation holds [64]:

$$\bowtie_{e=(u,v)} = \sum_{w\in 2hop_G(u), w\in N_G(v)} (|N_G(u)\cap N_G(w)|-1) = \sum_{x\in 2hop_G(v), x\in N_G(u)} (|N_G(v)\cap N_G(x)|-1) \quad (8)$$

Based on the above equation, our BFC-VP^{++} algorithm can be extended to compute the butterfly count for each edge. In Algorithm 3, for a start-vertex u^* and a valid end-vertex $w^* \in 2hop_G(u)$, the value $|N_G(u^*)\cap N_G(w^*)|$ is already computed which can be used directly to compute $\bowtie_e$.

Here, we present the BFC-EVP^{++} algorithm to compute $\bowtie_e$. The details of BFC-EVP^{++} are shown in Algorithm 4. In the initialization process, we initialize $\bowtie_e$ for each edge e in G. After that, for each start-vertex u^*, we run Algorithm 3 Lines 6–12 to compute $|N_G(u^*)\cap N_G(w^*)|$. Then, we run another round of wedge processing and update $\bowtie_{e(u,v)}$, $\bowtie_{e(v,w)}$ according to Eq. (8) (Lines 5–14). Finally, we return the result.

In Algorithm 4, we only need an extra array to store $\bowtie_e$ for each edge e. In addition, because it just runs the wedge processing procedure twice, we can get that the time complexity of the BFC-EVP^{++} algorithm is the same as BFC-VP^{++}.

Algorithm 4: BFC- EVP^{++}

```
   Input: $G$
   Output: $\bowtie_e$ for each $e$ in $G$
1  run Algorithm 3 Line 2 - Line 4
2  $\bowtie_e \leftarrow 0$ for each edge $e \in E(G)$
3  foreach vertex $u^* \in V(G^*)$ do
4      run Algorithm 3 Line 6 - Line 12
5      foreach $v^* \in N_{G^*}(u^*)$ do
6          foreach $w^* \in N_{G^*}(v^*) : p(w^*) > p(u^*)$ do
7              if $p(w^*) > p(v^*)$ then
8                  $\delta \leftarrow wedge\_cnt(w) - 1$
9                  $(v, w) \leftarrow f^{-1}(v^*, w^*)$
10                 $(u, v) \leftarrow f^{-1}(u^*, v^*)$
11                 $\bowtie_{(u,v)} \leftarrow \bowtie_{(u,v)} + \delta$
12                 $\bowtie_{(v,w)} \leftarrow \bowtie_{(v,w)} + \delta$
13             else
14                 break
15 return $\bowtie_e$ for each edge e in G
```

6.2 Parallelization

Shared-memory parallelization. In Algorithm 3, only read operations occur on the graph structure. This motivates us to consider the shared-memory parallelization. Assume we have multiple threads and these threads can handle different start-vertices simultaneously. No conflict occurs when these threads read the graph structure simultaneously. However, conflicts may occur when they update $wedge\_cnt$ and $\bowtie_G$ simultaneously in Algorithm 3. Thus, we can divide the data space into the global data space and the local data space. In the global data space, the threads can access the graph structure simultaneously. In the local data space, we use $local\_wedge\_cnt$ and $local\_\bowtie_G$ for each thread to avoid conflicts. Thus, we can use $O(n * t + m)$ space to extend BFC-VP^{++} into a parallel version, where t denotes the number of threads.

Algorithm 5: BFC- VP^{++} IN PARALLEL

```
  Input: $G$ and $t$
  Output: $\bowtie_G$
1 run Algorithm 3 Line 1 - Line 4
2 initialize $local\_wedge\_cnt[i]$ and $local\_ \bowtie_G [i]$ for each thread
  $i \leftarrow 1..t$
3 sort $u^* \in V(G^*)$ in non-ascending order by their priorities
4 foreach vertex $u^* \in V(G^*)$ do
5     allocate $u^*$ to an idle thread $i$
6     run Algorithm 3 Line 6 - Line 15, replace $wedge\_cnt$, $\bowtie_G$
      with $local\_wedge\_cnt[i]$, $local\_ \bowtie_G [i]$
7 /* on master thread */
8 $\bowtie_G \leftarrow \bowtie_G + local\_ \bowtie_G [i]$ for each thread $i \leftarrow 1..t$
9 return $\bowtie_G$
```

The algorithm BFC-VP^{++} in parallel. We show the details of the algorithm BFC-VP^{++} in parallel in Algorithm 5. Note that we use the priority-based dynamic scheduling strategy by considering the workload balance [66]. Similar as BFC-VP^{++}, we first generate a reordering graph G^*. Then, the algorithm sequentially processes the start-vertices in non-ascending order by their priorities. For a vertex $u^* \in V(G^*)$, it will be dynamically allocated to an idle thread i. Note that, for each thread i, we generate an independent space for $local\_wedge\_cnt[i]$ and $local\_ \bowtie_G [i]$. After all the threads finishing their computation, we compute $\bowtie_G$ on the master thread.

Note that the work-span model is popularly used to analyze the parallel algorithms. The work of an algorithm measures the total number of operations, and the span of an algorithm is the longest dependency path [20]. Based on this model, Algorithm 5 takes $O(\alpha \cdot m)$ work and $O(log(m))$ span with high probability, which can be analyzed similarly as done in [59].

6.3 External memory butterfly counting

In order to handle large graphs with limited memory size, we introduce the external memory algorithm BFC-EM in Algorithm 6 which is also based on the vertex priority. We first run an external sorting on the edges to group the edges with the same vertex-IDs together. Then, we compute the priorities of vertices by sequentially scanning these edges once. Then, for each vertex $v \in V(G)$, we sequentially scan its neighbors from the disk and generate the wedges (u, v, w) with $p(w) > p(v)$ and $p(w) > p(u)$ where $w \in N_G(v)$ and $u \in N_G(v)$ (Lines 4–6). For each wedge (u, v, w), we only store the vertex-pair (u, w) on disk. After that, we maintain the vertex-pairs on disk such that all (u, w) pairs with the same u and w values are stored continuously (Line 7). This can be simply achieved by running an external sorting on these (u, w) pairs. Then, we sequentially scan these vertex-pairs, and for the vertex-pair (u, w), we count the occurrence of it and compute $\bowtie_G$ based on Lemma 1 (Lines 8–10).

Algorithm 6: BFC- EM

```
   Input: $G$
   Output: $\bowtie_G$
 1 sort all the edges $e \in G$ on disk
 2 calculate the priority for each vertex in $V(G)$ on disk
 3 $\bowtie_G \leftarrow 0$
 4 foreach vertex $v \in G$ do
 5     forall the $u, w \in N_G(v) : p(w) > p(v), p(w) > p(u)$ by
       sequentially scanning $N_G(v)$ from disk do
 6         store vertex-pair $(u, w)$ on disk
 7 sort all the vertex-pairs on disk
 8 foreach vertex-pair $(u, w)$ do
 9     $count\_pair(u, w) \leftarrow$ count the occurrence of $(u, w)$ on disk
       sequentially
10     $\bowtie_G \leftarrow \bowtie_G + \binom{count\_pair(u,w)}{2}$
11 return $\bowtie_G$
```

I/O complexity analysis. We use the standard notations in [2] to analyze the I/O complexity of BFC-EM: M is the main memory size and B is the disk block size. The I/O complexity to scan N elements is $scan(N) = \Theta(\frac{N}{B})$, and the I/O complexity to sort N elements is $sort(N) = O(\frac{N}{B} log_{\frac{M}{B}} \frac{N}{B})$. In BFC-EM, the dominate cost is to scan and sort the vertex-pairs. Since there are $O(\alpha \cdot m)$ vertex-pairs generated by the BFC-EM algorithm, we can get that the I/O complexity of BFC-EM is $O(scan(\alpha \cdot m) + sort(\alpha \cdot m))$.

7 Batch-dynamic butterfly counting

In this section, we discuss the problem of batch-dynamic butterfly counting. Given a bipartite graph G and a batch-update B, we aim to compute the number of butterflies resulting from the batch-update B (denoted as $\bowtie_B$). Here, a batch-update B is a batch of edge insertion and edge deletion operations. In other words, suppose we already know how many butterflies in G, we want to obtain the number of butterflies after updating B on G. Since B contains edge insertion and deletion operations, we use B^+ to represent the set of inserting edges and use B^- to represent the set of deleting edges. We suppose B^+ and B^- are disjoint (i.e., $B^+ \cap B^- = \emptyset$) since we can safely remove all the common edges in these two sets without affecting the final butterfly counts. Then, we can first count the number of affected butterflies of B^- and then count the number of affected butterflies of B^+. Note that the counting procedures of the deletion batch and the insertion batch are inherently the same since we can consider deleting B^- from G as inserting B^- into $G \backslash B^-$. All we need to know is the number of affected butterflies resulting from a batch of updates. It is worth noticing that to transform a deletion case into an insertion case, we need to adjust the initial data structures. Specifically, we remove B^- from G and consider B^- as B^+. As evaluated in our experiments, such overhead is small. For the ease of presentation, we suppose all the updates are edge insertions in the following parts (i.e., $B = B^+$).

7.1 Computing $\bowtie_B$ from each new edge

It is apparent that each new butterfly contains at least one new edge from B. As a result, a straightforward algorithm BFCB-BS can be designed by sequentially processing each edge $(u, v) \in B$. For a new edge $(u, v) \in B$, we insert it into G and compute the number of new butterflies resulting from it according to Eq. (8). The details of BFCB-BS are shown in Algorithm 7.

The BFCB-BS Algorithm. For each new edge (u, v) in B, we first insert it into G and get the neighbor set $N_G(v)$ of v. Then, for each $w \in N_G(v)$, we compute the number of common neighbors between u and w (i.e., $|N_G(u) \cap N_G(w)|$) and add $|N_G(u) \cap N_G(w)| - 1$ to $\bowtie_B$ according to Eq. (8). When computing the common neighbors of u and w, we can construct a hash map H using the neighbor set of u (Line 4). Then, we can just look up each vertex $x \in N_G(w)$ to check whether it is in H (Lines 4–8). The time cost of the above procedure for processing each edge (u, v) is $O(\sum_{w \in N(v)} deg_G(w))$. To reduce the time cost, when processing the edge (u, v), we first compute the values of $\sum_{x \in N_G(u)} deg_G(x)$ and $\sum_{w \in N_G(v)} deg_G(w)$. If it satisfies $\sum_{x \in N_G(u)} deg_G(x) < \sum_{w \in N_G(v)} deg_G(w)$, we exchange u and v (i.e., process the neighbor set of u instead of that of v). In this way, we can reduce the number of wedges processed in the algorithm. One may also consider pre-computing the hash map of neighbor set for each vertex in G. Then, when computing $|N_G(u) \cap N_G(w)|$, we can choose to scan the smaller set between $N_G(u)$ and $N_G(w)$ with $O(min\{deg_G(u), deg_G(w)\})$ time. However, it incurs additional time cost to compute the hash map for each vertex in G which can be a very large overhead under the batch-dynamic context.

Example 2 Consider the original graph G and the batch-update $B = \{(u_3, v_1), (u_3, v_2)\}$ in Fig. 8a. We first insert (u_3, v_1) into G. Since we have $\sum_{x \in N_G(u_3)} deg_G(x) < \sum_{w \in N_G(v_1)} deg_G(w)$, we process the neighbor set of u_3. For each $x \in N_G(u_3) \backslash v_1$ (i.e., v_0), we compute $|N_G(v_1) \cap N_G(v_0)| = 3$ (i.e., u_0, u_1, and u_3). Then, we can get the number of new butterflies containing (u_3, v_1) is $3 - 1 = 2$.

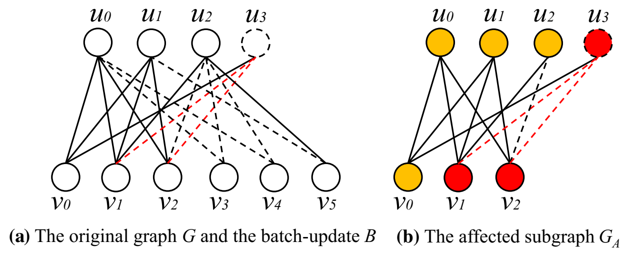


Fig. 8 An example of batch-updating. The edges in red color are the new edges

After that, we insert (u_3, v_2) into G, and get the number of new butterflies containing (u_3, v_2) is 5. In total, we can get the number of new butterflies resulting from B is 7.

Theorem 8 *The time complexity of* BFCB-BS *is* $O(\sum_{(u,v) \in B} min\{\sum_{x \in N(u)} deg_G(x), \sum_{w \in N(v)} deg_G(w)\})$ *in the worst-case.*

Proof Algorithm 7 processes the edges in B one by one. For each inserted edge $e = (u, v)$, the new butterflies can be enumerated by exploring the two-hop neighbors of u and check if they are connected to v, which takes $O(\sum_{x \in N(u)} deg_G(x))$ time. An alternative approach is to explore the two-hop neighbors of v that are connected to u, which takes $O(\sum_{w \in N(v)} deg_G(w))$ time. With the time costs of these two approaches pre-computed (Line 4), the time complexity of enumerating new butterflies for each edge (u, v) is the smaller of the two. Summing this time complexity over all the inserted edges completes the proof. □

In addition, BFCB-BS has the space complexity of $O(|G| + |B|)$ since we need $O(|G| + |B|)$ space to store the graph structure, and the hashmap H used in Algorithm 7 needs additional $O(|V(G \cup B)|)$ space.

Algorithm 7: BFCB- BS

```
Input: $G$ and $B$
Output: $\bowtie_B$: the number of new butterflies
1  $\bowtie_B \leftarrow 0$
2  foreach edge $(u, v) \in B$ do
3      insert $(u, v)$ into $G$;
4      exchange $u$ and $v$ if $\sum_{x \in N(u)} deg_G(x) > \sum_{w \in N(v)} deg_G(w)$
5      initialize a hashmap $H$ using $N_G(u)$
6      foreach vertex $w \in N_G(v) \backslash u$ do
7          foreach vertex $x \in N_G(w) \backslash v$ do
8              if vertex $x \in H$ then
9                  $\bowtie_B \leftarrow \bowtie_B + 1$
10 return $\bowtie_B$
```

Analysis of BFCB-BS Reviewing Algorithm 7, the drawbacks of BFCB-BS are apparent. (1) BFCB-BS needs to enumerate each butterfly containing a new edge in B which is inefficient.

The in-depth reason is that if we want to count the number of butterflies containing exactly one new edge, we cannot use vertex-priority-based techniques to amortize the time cost like Algorithm 4. (2) It is not easy to parallelize BFCB-BS since it needs to process each edge $e \in B$ sequentially and insert the edge e into G after processing it. Specifically, if we directly parallelize Line 2 in Algorithm 7, the parallel algorithm will likely overlook the new butterflies with more than one new edge which are processed by different threads simultaneously. There is no simple parallel implementation of BFCB-BS which can address this issue efficiently.

7.2 Computing $\bowtie_B$ from the affected subgraph

To address the above-mentioned issues, we propose an affected-subgraph-based algorithm BFCB-IG. The main intuition behind BFCB-IG is that before computing the new butterflies resulting from B, we first update B to G and identify the affected scope (or subgraph) of B. In this way, we not only can use the vertex-priority-based algorithms (in a subgraph with limited size) but also can easily derive a parallel implementation. We define the affected subgraph as follows.

Definition 6 (*Affected subgraph*) Given a bipartite graph G and a batch-update B, the affected subgraph G_A due to B is the induced subgraph of all the vertices in $V_B^1 \cup V_B^2$. Here, V_B^1 is the set of vertices that are incident to at least one new edge in B. V_B^2 is the set of vertices that are the neighbors of at least one vertex in V_B^1. For instance, the affected subgraph G_A of G is shown in Fig. 8.

Lemma 4 *Consider a bipartite graph G and a batch-update B. All the new butterflies resulting from B are contained in the affected subgraph G_A.*

Proof For each new butterfly, it must contain at least one new edge $e = (u, v)$. Since V_B^1 includes all the vertices incident to the new edges, u and v must be in V_B^1. Due to the butterfly structure, the other two vertices of the new butterfly must be connected to u and v, which must be contained in V_B^2 (Definition 6). Therefore, all of the new butterflies resulting from B are contained in the affected subgraph G_A. □

Based on the above lemma, we can guarantee that all the new butterflies are contained in G_A. Since G_A may also contain many butterflies which originally exist in G (i.e., old butterflies), we need to subtract the count of these old butterflies when applying the butterfly counting algorithm (e.g., BFC-VP^{++}) on G_A. Reviewing the BFC-VP^{++} algorithm (i.e., Algorithm 3), we can see that it starts from each vertex and processes valid wedges to count the number of butterflies. In the batch-dynamic context, we call a wedge is a new wedge if it contains at least one new edge from B. Otherwise, we call it is an old wedge. For example, (v_0, u_3, v_1) is a new wedge. It is easy to observe that any new butterfly in G_A is composed of (1) one new wedge and one old wedge; or (2) two new wedges. Thus, we can have the following lemma.

Lemma 5 *Given an affected subgraph G_A and a vertex $u \in G_A$, the number of new butterflies containing u denoted as $\bowtie_u^+$ can be computed by the following equation:*

$$\bowtie_u^+ = \sum_{w \in 2hop_{G_A}(u)} \binom{|N_{G_A}(u) \cap N_{G_A}(w)|}{2} - \binom{C_{\text{old}}(w)}{2} \tag{9}$$

Here $C_{\text{old}}(w)$ is the number of old wedges containing u and w in G_A.

Proof For any two vertices x_1, x_2 in $N_{G_A}(u) \cap N_{G_A}(w)$, $[u, x_1, w, x_2]$ is a butterfly in G_A, so the first term in the equation counts the number of butterflies containing u and w. To compute the number of new butterflies in G_A, we need to subtract the number of old butterflies. Note that an old butterfly can only be formed by two old wedges. Thus, the number of old butterflies can be computed as $\binom{C_{\text{old}}(w)}{2}$. $\bowtie_u^+$ is computed by taking the difference of these numbers and summing over all 2-hop neighbors of u, which completes the proof. □

The BFCB-IG Algorithm. Based on the above lemma, we design the algorithm BFCB-IG to count the number of new butterflies in G_A as shown in Algorithm 8. We first insert each edge in B into G. Then, we construct the affected subgraph G_A according to Definition 6. After that, we re-organized G_A using the cache-aware graph reordering technique and compute the vertex priority on G_A. Then, starting from each vertex in $V(G_A)$, we use the same strategy as BFC-VP^{++} to process the wedges. Unlike BFC-VP^{++}, BFCB-IG needs two hash maps C_{total} and C_{old} to record the number of wedges and the number of old wedges containing an end-vertex w, respectively. This is because we need to obtain the number of new butterflies which can be computed according to Lemma 5 (Line 17).

Example 3 Consider the example in Fig. 8a and the batch-update $B = \{(u_3, v_1), (u_3, v_2)\}$. We first construct G_A as shown in Fig. 8b. In G_A, we have $p(v_2) > p(v_1) > p(u_3) > p(u_1) > p(u_0) > p(v_0) > p(u_2) > p(v_3)$. Then, we start from each vertex in G_A to count the number of new butterflies by traversing valid wedges. From v_0, we get wedges (v_0, u_0, v_1), (v_0, u_1, v_1), (v_0, u_3, v_1), (v_0, u_0, v_2), (v_0, u_1, v_2), and (v_0, u_3, v_2). Since only (v_0, u_3, v_1) and (v_0, u_3, v_2) are new wedges, we have $C_{\text{total}}(v_1) = 3$, $C_{\text{old}}(v_1) = 2$, $C_{\text{total}}(v_2) = 3$, $C_{\text{old}}(v_2) = 2$. Thus, the number of new butterflies can be obtained is $\binom{3}{2} - \binom{2}{2} + \binom{3}{2} - \binom{2}{2} = 4$. Then, when starting from v_2, we will get 3 new butterflies similarly. In total, we have 7 new butterflies.

Algorithm 8: BFCB- IG

```
Input: $G$ and $B$
Output: $\bowtie_B$: the number of new butterflies
1  $\bowtie_B \leftarrow 0$;
2  update $B$ to $G$;
3  construct the affected subgraph $G_A$ according to Definition 6
4  $G_A \leftarrow$ run Algorithm 3 Lines 2–4 on $G_A$
5  foreach $u \in V(G_A)$ do
6      $C_{\text{total}} \leftarrow$ a hashmap initialized with zero
7      $C_{\text{old}} \leftarrow$ a hashmap initialized with zero
8      foreach $v \in N_{G_A}(u)$ do
9          foreach $w \in N_{G_A}(v) : p(w) > p(u)$ do
10             if $p(w) > p(v)$ then
11                 $C_{\text{total}}(w) \leftarrow C_{\text{total}}(w) + 1$
12                 if $(u, v) \notin B \wedge (v, w) \notin B$ then
13                     $C_{\text{old}}(w) \leftarrow C_{\text{old}}(w) + 1$
14             else
15                 break
16     foreach $w : C_{\text{total}}(w) > 1$ do
17         $\bowtie_B \leftarrow \bowtie_B + \binom{C_{\text{total}}(w)}{2} - \binom{C_{\text{old}}(w)}{2}$
18 return $\bowtie_B$
```

Theorem 9 *The* BFCB-IG *algorithm correctly solves the batch-dynamic butterfly counting problem.*

Proof By Lemma 5, the new butterflies due to the batch-update are all contained in the affected subgraph G_A. Algorithm 8 visits the vertices in the affected subgraph G_A and counts the number of new butterflies. Specifically, for each visited vertex $u \in V(G_A)$, the algorithm explores the two-hop neighbors of u and counts how many new wedges and old wedges are formed according to the vertex priority. For each two-hop neighbor w of u, $C_{\text{total}}(w)$ counts the total number of wedges containing u and w and $C_{\text{old}}(w)$ counts the old wedges (Lines 6, 7, 11, and 13). Note that only the wedges in which w has the highest priority are visited (Lines 9–10), which can avoid counting duplicate butterflies. By Lemma 5, the new butterflies containing u and w is computed by subtracting the number of the old butterflies containing u and w from the total count (Line 17). Since $\bowtie_B$ keeps an accumulating sum of the number of new butterflies containing the visited vertices, $\bowtie_B$ becomes the number of new butterflies when the for-loop terminates in line 18, which completes the proof. □

Theorem 10 *The worst-case time complexity of* BFCB-IG *is* $O(TC_{G_A} + \sum_{(u,v)\in E(G_A)} min\{deg_{G_A}(u), deg_{G_A}(v)\}$. *Here,* TC_{G_A} *is the worst-case time complexity of constructing* G_A, *bounded by* $O(|B| + \sum_{u\in V(G_A)} deg_G(u))$.

Proof Algorithm 8 constructs the affected subgraph G_A from B first and then counts the new butterflies on G_A. When constructing G_A, the algorithm inserts the edges in B into G, which takes $O(|B|)$ time. Then, it needs to visit the vertices in $V(G_A)$ and scan through each vertex's neighbors in G to construct the affected subgraph. Therefore, the affected subgraph construction takes $TC_{G_A} = O(|B| + \sum_{u\in V(G_A)} deg_G(u))$ time. Also, Algorithm 8 needs to enumerate all the valid wedges similar to Algorithm 3. By Theorem 3, this process takes $O(\sum_{(u,v)\in E(G_A)} min\{deg_{G_A}(u), deg_{G_A}(v)\})$ time. Adding it to the time complexity of G_A construction completes the proof. □

In addition, BFCB-IG has the space complexity of $O(|G| + |B|)$ since apart from the graph structure, the data structures used in BFCB-IG$^+$ (e.g., C_{total} and C_{old}) are all bounded by $O(|G| + |B|)$.

7.3 Reducing the computation of old wedges

Motivation. Although the algorithm BFCB-IG can effectively utilize the vertex-priority-based techniques, it can be further sped up by reducing the computation of old wedges. In BFCB-IG, the number of new butterflies is obtained based on Lemma 5 which needs to compute the total number of butterflies and the number of old butterflies on G_A. In this manner, we need to traverse many old wedges, and some of them do not lead to a new butterfly. For instance, consider the affected subgraph G_A in Fig. 10, we need to traverse many wedges which do not form any new butterfly such as (u_1, v_1, u_2) and (u_1, v_2, u_2). To explore how to reduce the computation of old wedges (and old butterflies), we first present all the cases of new butterflies as shown in Fig. 9. These cases are based on the fact that each new butterfly must contain at least one new edge.

- **The new butterfly contains one new edge.** Case 1 and Case 2. The number of new edges that w is incident to: Case 1: 0; Case 2: 1.
- **The new butterfly contains two new edges.** Case 3–Case 6. The number of new edges that w is incident to: Case 3: 1 (two new edges are incident edges); Cases 4: 0; Case 5: 1 (two new edges are not incident edges); Case 6: 2.
- **The new butterfly contains three new edges.** Case 7 and Case 8. The number of old edges that w is incident to: Case 7: 0; Case 8: 1.
- **The new butterfly contains four new edges.** Case 9.

Handling new butterflies of different cases. Reviewing the vertex-priority-based algorithm BFCB-IG, a butterfly is counted by composing two wedges which both use a vertex w with the highest priority as the end-vertex. We show all the cases of new butterflies in Fig. 22. We can see that the butterflies in Cases 4–9 are all composed of two new wedges (i.e., a wedge composed by at least one new edge). Thus, to avoid touching old wedges when identifying these butterflies, we can split N_{G_A} into $N_{\text{old}}(u)$ and $N_{new}(u)$ for each $u \in V(G_A)$ which contain the old neighbors and new inserted neighbors of u, separately. Then, when starting from a vertex u, we can

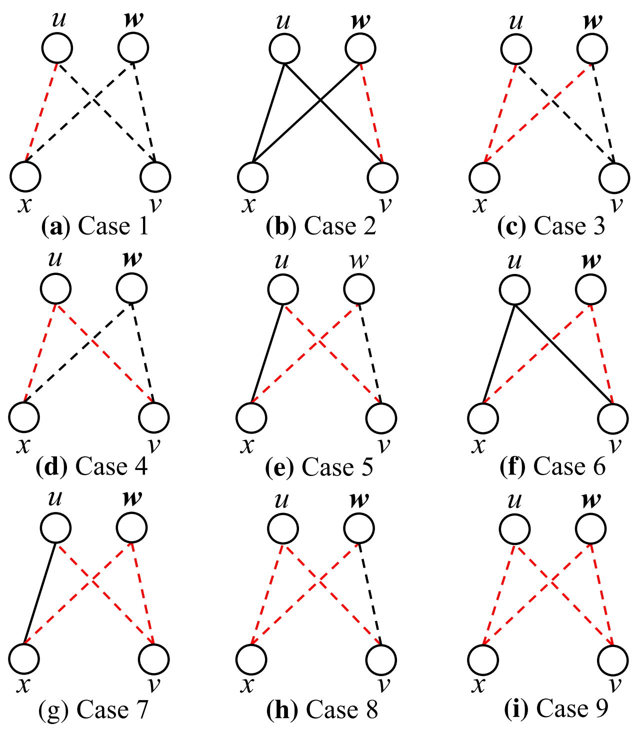


Fig. 9 All the cases of new butterflies. We suppose w is the vertex with the highest priority in a butterfly, and the new edges are denoted in dotted line

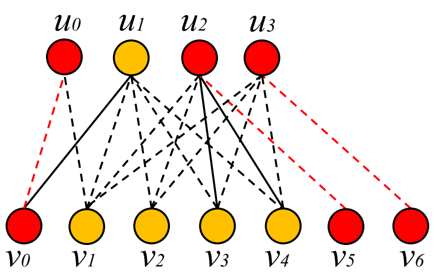


Fig. 10 Illustrating the BFCB-IG$^+$ algorithm

choose a vertex v from $N_{\text{old}}(u)$ (or $N_{new}(u)$) and choose a vertex w from $N_{new}(v)$ (or $N_{G_A}(v)$) to only traverse new wedges.

When handling Cases 1–3, we cannot avoid the traversal of old wedges since these butterflies are composed of one new wedge and one old wedge. For instance, considering Case 1 in Fig. 10, the butterfly is composed of the new wedge (u, x, w) and the old wedge (u, v, w). To reduce the computation of old wedges which do not lead to new butterflies, we adopt the following strategies. (1) We do not process a start-vertex u if there is no new wedge generated from it. To achieve this, we can record the set of end-vertices of the new wedges in an array *arr_2hop* when handling Cases 4–9 and terminate the finding process if *arr_2hop* is empty. For instance, consider the example in Fig. 10. When starting from u_1, there is no new wedge exists from it and we skip the process of finding butterflies in Cases 4–9 from u_1. (2) When enumerating old wedges from a start-vertex u, we need to scan the neighbor list $N_{\text{old}}(v)$ for each $v \in N_{\text{old}}(u)$ to find whether $w \in arr\_2hop$ exists. In many cases, the number of end-vertices of new wedges can be much smaller compared with the size of the neighbor set of v (i.e., $|N_{\text{old}}(v)|$). In such cases, we can perform a binary search to check whether each $w \in arr\_2hop$ exists in $N_{\text{old}}(v)$. For instance, when starting u_0, the only end-vertex of a new wedge is u_1. Thus, when finding old wedges starting from u_0, we can use binary search to check whether u_1 exists in $N_{\text{old}}(v_1)$ rather than scan $N_{\text{old}}(v_1)$.

The BFCB-IG$^+$ Algorithm. With the above optimization strategies, we propose the BFCB-IG$^+$ algorithm in Algorithm 9. We construct the affected subgraph G_A similarly as BFCB-IG. After that, we split $N_{G_A}(u)$ into $N_{\text{old}}(u)$ and $N_{new}(u)$ for each $u \in V(G_A)$ (Line 5). Then, starting from each vertex in $V(G_A)$, we use the same strategy as BFC-VP^{++} to process the wedges. Unlike BFCB-IG, we first count the number of butterflies in Cases 4–9 (i.e., the butterflies composed by two new wedges) as shown in Fig. 22. To achieve this, we need to process all the new wedges. We can choose a vertex v from $N_{\text{old}}(u)$ (or $N_{new}(u)$) and choose a vertex w from $N_{new}(v)$ (or $N_{G_A}(v)$) (Lines 10–25). Note that all the end-vertices of the new wedges containing u are recorded in an array *arr_2hop*. Then, we count the number of butterflies for Cases 1–3 (Lines 31–41). We skip this stage if there is no new wedges in G_A to reduce the time cost (Line 28). Inherently, in this stage, we aim to find the number of common neighbors for each u and $w \in arr\_2hop$ on the original graph. We can scan the neighbor list for each $v \in N_{\text{old}}(u)$ to find whether $w \in arr\_2hop$ exists. Alternatively, we can use binary search approach to check whether each $w \in arr\_2hop$ in $N_{\text{old}}(v)$. For each $v \in N_{\text{old}}(u)$, we can pre-compute $|N_{\text{old}}(v)|$ and $|arr\_2hop| \cdot log(|N_{\text{old}}(v)|)$ to choose which method is more efficient (Line 32).

Example 4 Consider the affected subgraph G_A in Fig. 10. The only new butterfly in G_A is $[u_0, v_0, u_1, v_1]$. To count this new butterfly using BFCB-IG$^+$, we first start from u_0 and get the new wedge (u_0, v_0, u_1). Then, we use binary search to get that u_1 is the neighbor of v_1 (i.e., there exists a old wedge (u_0, v_1, u_1)). Compared with BFCB-IG, BFCB-IG$^+$ does not need to traverse any other old wedges. For instance, the old wedge (u_1, v_1, u_2) is not touched since there is no new wedge generated starting from u_1.

Theorem 11 *The* BFCB-IG$^+$ *algorithm correctly solves the batch-dynamic butterfly counting problem.*

Proof By Lemma 5, Algorithm 9 only counts the new butterflies in G_A. For each vertex u, it first counts the new butterflies with two new wedges (Lines 9–27) and then counts the ones with only one new wedge (Lines 31–41). Specifically, Lines 10–17 process the new wedges in which (u, v) is old and (v, w) is new, and Lines 18–25 process those in which (u, v) is new. Since the new butterflies in Cases 4–9 are formed by two new wedges, Lines 26 and 27 correctly compute the number of such butterflies containing u. In Lines 31–41, the algorithm finds an old wedge (u, v, w) and checks how many new wedges containing u and w exist. Since the butterflies of Cases 1–3 need one old wedge and one new wedge, there are $C_{\text{total}}(w)$ many such butterflies corresponding to each old wedge (u, v, w). Thus, this algorithm handles all possible cases of new butterflies and correctly returns the number of new butterflies. □

Algorithm 9: BFCB- IG$^+$

```
Input: $G$ and $B$
Output: $\bowtie_B$: the number of new butterflies
$\bowtie_B \leftarrow 0$;
update $B$ to $G$;
construct the affected subgraph $G_A$ according to Definition 6
$G_A \leftarrow$ run Algorithm 3 Lines 2 - 4 on $G_A$
split $N_{G_A}(u)$ into $N_{\text{old}}(u)$ and $N_{new}(u)$ for each $u \in V(G_A)$.
foreach $u \in V(G_A)$ do
    $C_{\text{total}} \leftarrow$ a hashmap initialized with zero
    $arr\_2hop \leftarrow$ an array initialized with empty
    /*Counting butterflies for Cases 4 - 9*/
    foreach $v \in N_{\text{old}}(u)$ do
        foreach $w \in N_{new}(v) : p(w) > p(u)$ do
            if $p(w) > p(v)$ then
                $C_{\text{total}}(w) \leftarrow C_{\text{total}}(w) + 1$
                if $w \notin arr\_2hop$ then
                    $arr\_2hop.add(w)$
            else
                break
    foreach $v \in N_{new}(u)$ do
        foreach $w \in N_{G_A}(v) : p(w) > p(u)$ do
            if $p(w) > p(v)$ then
                $C_{\text{total}}(w) \leftarrow C_{\text{total}}(w) + 1$
                if $w \notin arr\_2hop$ then
                    $arr\_2hop.add(w)$
            else
                break
    foreach $w : C_{\text{total}}(w) > 1$ do
        $\bowtie_B \leftarrow \bowtie_B + \binom{C_{\text{total}}(w)}{2}$
    if $arr\_2hop$ is empty then
        continue;
    /*Counting butterflies for Cases 1–3*/
    foreach $v \in N_{\text{old}}(u)$ do
        if $|N_{\text{old}}(v)| \le |arr\_2hop| \cdot log(|N_{\text{old}}(v)|)$ then
            foreach $w \in N_{\text{old}}(v) : p(w) > p(u)$ do
                if $p(w) > p(v)$ then
                    $\bowtie_B \leftarrow \bowtie_B + C_{\text{total}}(w)$
                else
                    break
        else
            foreach $w \in arr\_2hop : p(w) > p(v)$ do
                if $binary\_search(N_{\text{old}}(v), w)$ = True then
                    $\bowtie_B \leftarrow \bowtie_B + C_{\text{total}}(w)$
return $\bowtie_B$
```

Complexity analysis. BFCB-IG$^+$ has the same worst-case time complexity as BFCB-IG. However, it reduces much computation of old wedges and can be more efficient as studied in our experimental evaluations. In addition, BFCB-IG$^+$ has the same space complexity as BFCB-IG (i.e., $O(|G| + |B|)$) since the additional data structures used in BFCB-IG$^+$ (i.e., arr_2hop, N_{old}, and N_{new}) are all bounded by $O(|G|+|B|)$.

Remark The algorithms BFCB-IG and BFCB-IG$^+$ can all be parallelized similarly as Algorithm 5. Note that it is straightforward to devise a work-efficient[2] parallel algorithm based on BFCB-IG$^+$. In the affected graph constructing phase, we can scan each new edge in B in parallel to get the vertex set V_B^1. Then, we scan the vertices in V_B^1 in parallel to get the vertex set V_B^2. In the butterfly counting phase, we can follow the same paradigm as Algorithm 5 to process the start-vertices in parallel.

8 Experiments

In this section, we present the results of empirical studies. In particular, our empirical studies are conducted against the following designs in Sect. 8.2: (1) the state-of-the-art BFC-IBS in [54] as the baseline algorithm (we thank the authors for providing the code), (2) BFC-VP in Sect. 4, (3) BFC-VP$^+$ in Sect. 5.1, (4) BFC-VP^{++} in Sect. 5.3, (5) BFC-EIBS, BFC-EVP, BFC-EVP^{++} by extending BFC-IBS, BFC-VP and BFC-VP^{++}, respectively, to compute $\bowtie_e$, (6) the parallel version of BFC-IBS, BFC-VP and BFC-VP^{++}, (7) the most advanced approximate butterfly counting algorithm BFC-ESap in [54], (8) BFC-ESap$_{\text{vp}^{++}}$ by combining BFC-VP^{++} with BFC-ESap since BFC-ESap relies on the exact butterfly counting techniques on samples, and (9) the external memory algorithm BFC-EM.

We also evaluate the algorithms for batch-dynamic butterflies counting in Sect. 8.3 including the baseline algorithm BFCB-BS, the affected-graph-based butterfly counting algorithm BFCB-IG, and the advanced affected-graph-based algorithm BFCB-IG$^+$.

The algorithms are implemented with C++. We run the empirical studies on a computer with 512 GB main memory and 2 × Intel Xeon E5-2698 processors (2.20GHz, 640 KB L1I Cache, 640 KB L1D Cache, 5MB L2 Cache, 50MB L3 Cache, and 40 cores in total). Although most empirical studies have been against single-core, we want our empirical studies to be conducted on the same computer as the evalu-

[2] A parallel algorithm is work-efficient if the total work of it matches that of the best sequential algorithm.

Table 1 Statistics of datasets

Dataset	$\|E\|$	$\|U\|$	$\|L\|$	$⧖_G$	$\sum_{u\in L} d(u)^2$	$\sum_{v\in R} d(v)^2$	TC_{ibs}	TC_{new}
DBPedia	293,697	172,091	53,407	3.76×10^6	6.30×10^5	2.46×10^8	6.30×10^5	5.95×10^5
Twitter	1,890,661	175,214	530,418	2.07×10^8	7.42×10^7	1.94×10^9	7.42×10^7	3.02×10^7
Amazon	5,743,258	2,146,057	1,230,915	3.58×10^7	8.29×10^8	4.37×10^8	4.37×10^8	6.90×10^7
Wiki-fr	22,090,703	288,275	4,022,276	6.01×10^{11}	2.19×10^{12}	7.96×10^8	7.96×10^8	7.08×10^7
Live-journal	112,307,385	3,201,203	7,489,073	3.30×10^{12}	9.57×10^9	5.40×10^{12}	9.57×10^9	8.01×10^9
Wiki-en	122,075,170	3,819,691	21,504,191	2.04×10^{12}	1.26×10^{13}	2.33×10^{10}	2.33×10^{10}	9.32×10^9
Delicious	101,798,957	833,081	33,778,221	5.69×10^{10}	8.59×10^{10}	5.28×10^{10}	5.28×10^{10}	1.31×10^{10}
Tracker	140,613,762	27,665,730	12,756,244	2.01×10^{13}	1.73×10^{12}	2.11×10^{14}	1.73×10^{12}	7.83×10^9
Orkut	327,037,487	2,783,196	8,730,857	2.21×10^{13}	1.57×10^{11}	4.90×10^{12}	1.57×10^{11}	1.12×10^{11}
Bi-twitter	601,734,937	20,826,115	20,826,110	6.30×10^{13}	2.69×10^{13}	3.48×10^{13}	2.69×10^{13}	1.66×10^{11}
Bi-sk	910,924,634	25,318,075	25,318,075	1.22×10^{14}	3.42×10^{13}	1.80×10^{13}	1.80×10^{13}	7.83×10^{10}
Bi-uk	1,327,632,357	38,870,511	38,870,511	4.89×10^{14}	4.22×10^{13}	4.16×10^{13}	4.16×10^{13}	2.92×10^{11}

ation of parallel performance. *An algorithm is terminated if it cannot finish within 10 h.*

8.1 Datasets

Twelve datasets are used in our experiments including all the 9 real datasets in [54]. We add 3 more datasets to evaluate the scalability of our techniques. All these datasets can be downloaded from KONECT (http://konect.cc/).

The 9 real-world datasets that we used are `DBPedia`, `Twitter`, `Amazon`, `Wiki-fr`, `Wiki-en`, `Live-journal`, `Delicious`, `Tracker`, and `Orkut`.

To further test the scalability, we also evaluate three bipartite networks (i.e., `Bi-twitter`, `Bi-sk` and `Bi-uk`) which are sub-networks obtained from billion-scale real datasets (i.e., `twitter`, `sk-2005` and `uk-2006-05`). To obtain bipartite-subgraphs from these two datasets, we put the vertices with odd ids in one group while the vertices with even ids in the other group and remove the edges which formed by two vertices with both odd ids or even ids.

In Table 1, we show the statistics of datasets. Note that $\sum_{u\in L} d(u)^2$ and $\sum_{v\in R} d(v)^2$ represent the sum of degree squares for L and R, respectively. $TC_{\text{ibs}} = \min\{\sum_{u\in L} d(u)^2, \sum_{v\in R} d(v)^2\}$ which is the time complexity bound of BFC-IBS. TC_{new} is the time complexity bound of BFC-VP and BFC-VP^{++}.

8.2 Evaluation of butterfly counting algorithms

In this section, the performance of the proposed algorithms for butterfly counting is evaluated.

Evaluating the performance on all datasets. Figure 11 shows the performance of BFC-IBS, BFC-VP, and BFC-VP^{++} on different datasets. We can observe that BFC-VP^{++} is the most efficient algorithm, while BFC-VP also outperforms BFC-IBS. This is because BFC-VP^{++} utilizes both the vertex-priority-based optimization and the cache-aware strategies which significantly reduce the computation cost. On `Tracker`, BFC-VP and BFC-VP^{++} are faster than BFC-IBS by at least two orders of magnitude. On `Bi-twitter`, `Bi-sk` and `Bi-uk`, BFC-IBS cannot finish within 10 h. This is because the degree distribution of these datasets is skewed and high-degree vertices exist in both layers. For instance, TC_{ibs} is more than 100× larger than TC_{new} in `Tracker`. This property leads to a large number of wedge processing for BFC-IBS while our BFC-VP and BFC-VP^{++} algorithms can handle such situation efficiently. In Fig. 12, we also show the performance for computing $⧖_e$ for each e. The performance differences of these algorithms follow similar trends to those in Fig. 11.

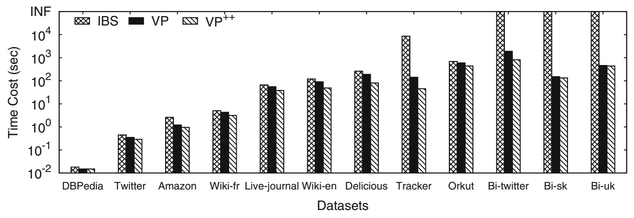


Fig. 11 Evaluation on all datasets

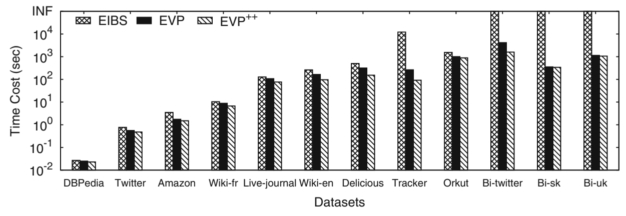


Fig. 12 Performance on all datasets (computing $⧖_e$)

Evaluating the number of processed wedges. Figure 13 shows the number of processed wedges of the proposed algorithms. We can observe that on `Tracker`, `Bi-twitter`, `Bi-sk` and `Bi-uk` datasets, BFC-IBS needs to process 100× more wedges than BFC-VP and BFC-VP^{++}. This is because TC_{ibs} is much larger than TC_{new} and there exist many hub-vertices in both L and R in these datasets. Thus, BFC-VP and BFC-VP^{++} only need to process a limited number of wedges while BFC-IBS should process numerous wedges no matter choosing which layer to start. We also observe that BFC-VP and BFC-VP^{++} process the same number of wedges since BFC-VP^{++} improves BFC-VP on cache performance, which does not change the number of processed wedges.

Scalability. Figure 14 studies the scalability of the algorithms by varying the number of vertices n. Specifically, we randomly sample 20–100% of the dataset's vertices and retrieve the induced subgraphs of the selected vertices. It is observed that BFC-VP and BFC-VP^{++} are scalable and the computational cost of all the algorithms increases when the percentage of vertices increases. On `Bi-twitter`, BFC-IBS can only complete when $n = 20\%$. As discussed before, BFC-VP^{++} is the most efficient algorithm.

Evaluating the effect of graph density. Figure 15 studies the effect of graph density (or sparsity) by varying the number of edges m. Specifically, we fix all the vertices and randomly sample 20–100% of the edges in the datasets. We can observe that when the graph is sparse, the performance gap between the vertex-priority-based algorithms and the baseline algorithm is relatively small. This is because the number of high degree vertices and the total number of butterflies are relatively low in a sparse graph. When the graph becomes denser, the advantage of the vertex-priority-based algorithms becomes more apparent.

Evaluating the parallelization techniques. Figure 16 studies the performance of the BFC-IBS, BFC-VP, and BFC-VP^{++} algorithms in parallel by varying the thread number t from 1 to 32 on four datasets. Note that we are actually increasing the number of physical cores when increasing the number of threads. The BFC-IBS algorithm in parallel is not parallel-friendly. For example, on `Tracker`, the BFC-IBS algorithm in parallel performs worse when t increases from 16 to 32. On `Bi-twitter`, the algorithm BFC-IBS in parallel cannot get a result within the timeout threshold when $t = 1$ and $t = 8$. We can also observe that, on all these datasets, the

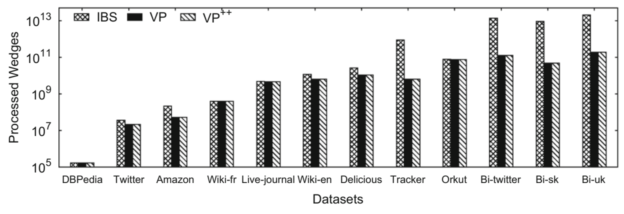


Fig. 13 The number of processed wedges

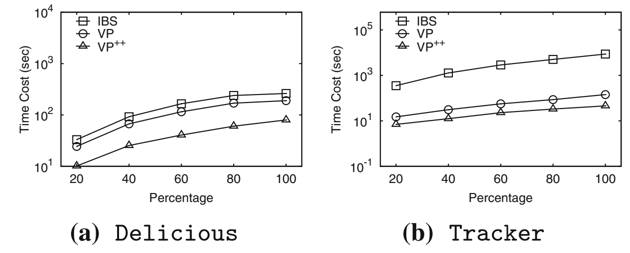


Fig. 15 Varying the number of edges

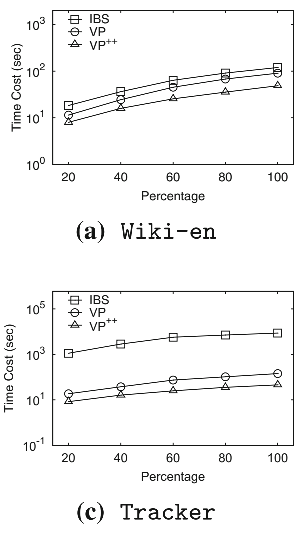
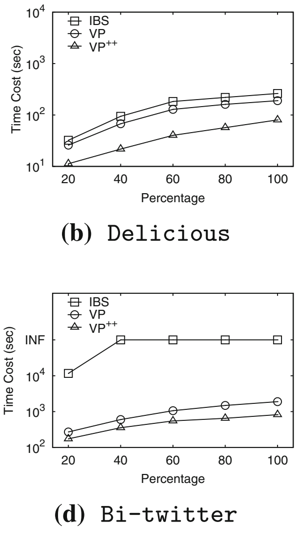



Fig. 14 Varying the number of vertices

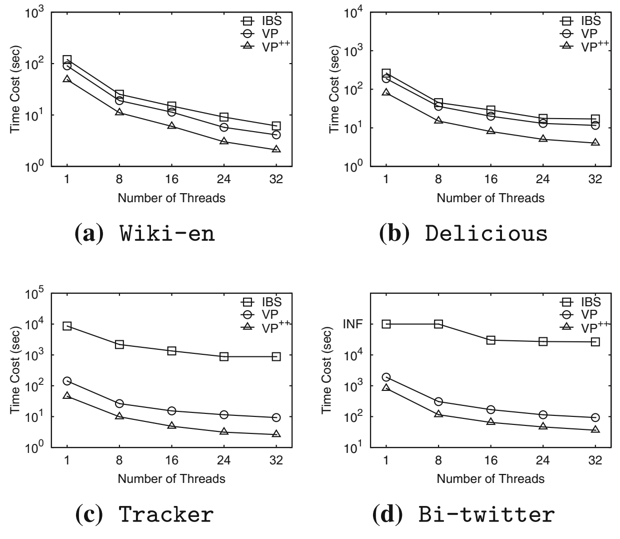


Fig. 16 Varying the number of threads

computation costs of the BFC-VP and BFC-VP^{++} algorithms in parallel decrease when the number of threads increases, and BFC-VP^{++} in parallel is more efficient than the other algorithms.

Evaluating the cache-aware strategies. In Tables 2, 3, 4 and 5, we evaluate the cache-aware strategies on `Wiki-en`, `Delicious`, `Tracker` and `Bi-twitter`, respectively. Here, *Cache-ref* denotes the total cache access number. *Cache-m* denotes the total cache miss number which means the number of cache references missed. *Cache-mr* denotes the percentage of cache references missed over the total cache access number. *Time* denotes the computation time of different algorithms. Here, BFC-VP^{+} is the BFC-VP algorithm deploying with only the cache-aware wedge processing strategy. BFC-VPC is the BFC-VP algorithm deploying with only the graph reordering strategy. BFC-VP has the largest number of cache-miss on all the datasets. By utilizing the cache-aware wedge processing, compared with BFC-VP, BFC-VP^{+} reduces the number of cache miss over 50% on all the datasets. By utilizing the cache-aware reordering, compared with BFC-VP, BFC-VPC also reduces over 50% cache-miss on all the datasets. BFC-VP^{++} achieves the smallest cache-miss-numbers and reduces the cache-miss-ratio significantly on all these datasets since BFC-VP^{++} combines the two cache-aware strategies together. Compared with BFC-VP, BFC-VP^{++} reduces over more than 70% cache miss on all the testing datasets.

Table 2 Cache statistics over `Wiki-en`

Algorithm	Cache-ref	Cache-m	Cache-mr	Time (s)
BFC-VP	2.78×10^{11}	3.13×10^{9}	1.12%	90.41
BFC-VPC	2.39×10^{11}	1.46×10^{9}	0.61%	63.45
BFC-VP^{+}	2.68×10^{11}	1.55×10^{9}	0.58%	65.26
BFC-VP^{++}	2.36×10^{11}	8.30×10^{8}	0.35%	48.60

Table 3 Cache statistics over `Delicious`

Algorithm	Cache-ref	Cache-m	Cache-mr	Time (s)
BFC-VP	4.53×10^{11}	8.36×10^{9}	1.85%	189.71
BFC-VPC	4.19×10^{11}	4.08×10^{9}	0.97%	133.48
BFC-VP^{+}	4.40×10^{11}	3.87×10^{9}	0.88%	102.82
BFC-VP^{++}	4.13×10^{11}	1.01×10^{9}	0.24%	80.26

Table 4 Cache statistics over `Tracker`

Algorithm	Cache-ref	Cache-m	Cache-mr	Time (s)
BFC-VP	2.74×10^{11}	5.27×10^{9}	1.93%	142.66
BFC-VPC	2.40×10^{11}	1.88×10^{9}	0.84%	87.61
BFC-VP^{+}	2.52×10^{11}	1.75×10^{9}	0.78%	82.16
BFC-VP^{++}	2.39×10^{11}	6.20×10^{8}	0.26%	45.48

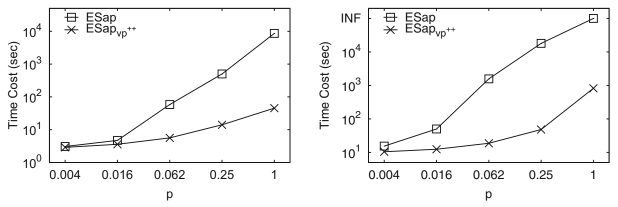
Table 5 Cache statistics over `Bi-twitter`

Algorithm	Cache-ref	Cache-m	Cache-mr	Time (s)
BFC-VP	4.87×10^{12}	4.96×10^{10}	1.02%	1897.15
BFC-VPC	4.55×10^{11}	2.47×10^{10}	0.54%	1261.11
BFC-VP^{+}	4.58×10^{12}	2.39×10^{10}	0.52%	1096.86
BFC-VP^{++}	4.54×10^{12}	1.35×10^{10}	0.30%	822.31

Speeding up the approximate butterfly counting algorithm. In the approximate algorithm BFC-ESap [54], the exact butterfly counting algorithm BFC-IBS is served as a basic block to count the butterfly exactly in a sampled graph. Since BFC-VP^{++} and BFC-IBS both count the number of butterflies exactly, the approximate algorithm BFC-ESap$_{vp^{++}}$ can be obtained by applying BFC-VP^{++} in BFC-ESap without changing the theoretical guarantee.

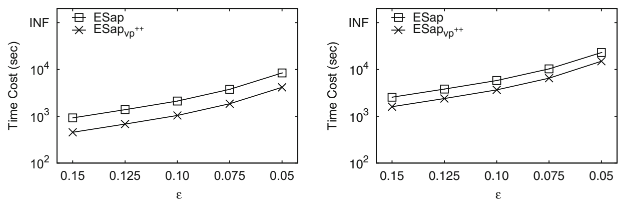
In Fig. 17, we first evaluate the average running time of BFC-ESap and BFC-ESap$_{vp^{++}}$ for each iteration by varying the probability p. Comparing two approximate algorithms, BFC-ESap$_{vp^{++}}$ outperforms BFC-ESap under all the setting of p on `Tracker` and `Bi-twitter` datasets. Especially, on these two datasets, BFC-ESap$_{vp^{++}}$ is more than one order of magnitude faster than BFC-ESap when $p \geq 0.062$.

In the second experiment, we run the algorithms to yield the theoretical guarantee $Pr[|\hat{\Join}_G - \Join_G| > \epsilon \Join_G] \leq \delta$ as shown in [54]. We vary ϵ and fix $\delta = 0.1$ and p as the optimal p as suggested in [54]. We can see that in Fig. 18, for these two approximate algorithms, the time costs are increased on these two datasets in order to get better accuracy and



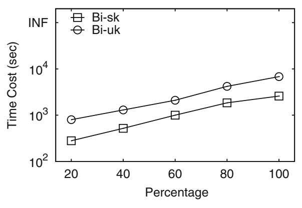
(a) `Tracker`, varying p **(b)** `Bi-twitter`, varying p

Fig. 17 Effect of p

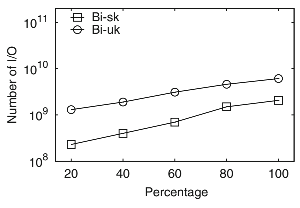


(a) `Tracker`, varying ϵ **(b)** `Bi-twitter`, varying ϵ

Fig. 18 Effect of ϵ



(a) Time Cost, varying n



(b) I/O, varying n

Fig. 19 Evaluating the external memory algorithm

BFC-ESap$_{vp^{++}}$ significantly outperforms BFC-ESap as mentioned before.

Evaluating the external memory algorithm. In Fig. 19, we evaluate the scalability of the external memory algorithm BFC-EM on two large datasets `Bi-sk` and `Bi-uk` by varying the graph size n. We limit the memory size to 1GB in our evaluation. On `Bi-sk` and `Bi-uk`, we can see that the time cost and I/O both increase with the percentage of vertices increases.

Cache-aware graph reordering versus Gorder. In [70], the authors propose *Gorder* to reduce the cache miss in graph algorithms. Here, we replace the graph reordering strategy with Gorder in BFC-VP^{++} and evaluate the performance.

Table 6 shows the time cost. We can observe that the renumbering time cost of the graph reordering is much less than Gorder on all datasets. This is because graph reordering can be simply obtained according to the priority number of vertices while Gorder needs complex renumbering computation. Regarding the computation time, the performance of the algorithm with graph reordering is better than the algorithm with Gorder on 9 datasets while the algorithm with Gorder is better on 3 datasets. Finally, the total time cost of graph reordering is better than Gorder.

Table 7 shows the cache statistics. Firstly, they have a similar number of cache references since the renumbering process does not change the algorithm itself. Secondly, graph reordering achieves a better CPU performance than Gorder on almost all datasets (i.e., fewer cache-misses and fewer cache-miss-ratios on 9 datasets) when handing the butterfly counting problem with BFC-VP^{++}. In summary, our graph reordering strategy is more suitable when handling butterfly counting.

8.3 Evaluation of batch-dynamic butterfly counting

In this subsection, we evaluate the performance of BFCB-BS, BFCB-IG, and BFCB-IG$^+$ to solve the batch-dynamic butterfly counting problem. For each test, we randomly extract a proportion b of edges from the graph as the newly inserted edges for batch-update, and the graph consisting of the remaining edges is used as the original graph. By default, we set the batch size $b = 0.5\%$ and the number of threads $t = 1$.

Evaluating the performance on all datasets. Figure 20 shows the performance of the BFCB-BS, BFCB-IG, and BFCB-IG$^+$ algorithms on all datasets with b and t set to default values. We can observe that BFCB-IG and BFCB-IG$^+$ outperform BFCB-BS on almost all the large datasets. The BFCB-IG and BFCB-IG$^+$ algorithms are faster than BFCB-BS by at least one order of magnitude on `Wiki-fr`, `Live-journal`, `Wiki-en`, `Orkut Bi-twitter`, `Bi-sk`, and `Bi-uk`. Especially, on `Trackers`, the BFCB-IG$^+$ algorithm is faster than the BFCB-BS algorithm by at least two orders of magnitude. In addition, BFCB-IG$^+$ is faster than BFCB-IG on all the datasets since it is designed to avoid much computation of existing wedges than BFCB-IG. Note that both BFCB-IG and

Table 6 Time cost compared with Gorder

Dataset	Renumbering time		Computation time		Total time	
	Reordering	Gorder	Reordering	Gorder	Reordering	Gorder
DBPedia	**0.01**	0.04	**0.02**	0.03	**0.03**	0.07
Twitter	**0.06**	4.26	0.29	**0.25**	**0.35**	4.51
Amazon	**0.30**	3.56	**0.96**	1.46	**1.26**	5.02
Wiki-fr	**0.49**	28.51	**3.16**	5.28	**3.65**	33.79
Live-journal	**1.32**	125.96	**37.86**	52.76	**39.18**	178.72
Wiki-en	**3.02**	856.07	**48.60**	75.78	**51.62**	931.85
Delicious	**3.82**	2225.44	**80.26**	134.86	**84.08**	2360.30
Tracker	**4.89**	315.01	**45.48**	56.13	**50.37**	371.13
Orkut	**2.17**	1615.01	**435.12**	553.03	**437.29**	2168.04
Bi-twitter	**6.64**	3211.63	**822.31**	1276.63	**828.95**	4488.26
Bi-sk	**8.32**	605.87	133.34	**107.07**	**141.66**	692.94
Bi-uk	**9.91**	1231.93	435.29	**401.64**	**445.20**	1633.57

Table 7 Cache statistics compared with Gorder

Dataset	Cache reference		Cache miss		Cache miss ratio	
	Reordering	Gorder	Reordering	Gorder	Reordering	Gorder
DBPedia	**4.02** $\times 10^7$	5.61 $\times 10^7$	**4.54** $\times 10^4$	1.20 $\times 10^5$	**0.11%**	0.21%
Twitter	**8.89** $\times 10^8$	9.56 $\times 10^8$	5.09 $\times 10^5$	**4.68** $\times 10^5$	0.06%	**0.05%**
Amazon	**2.51** $\times 10^9$	2.52 $\times 10^9$	**8.93** $\times 10^6$	1.02 $\times 10^7$	**0.36%**	0.40%
Wiki-fr	**1.34** $\times 10^{10}$	1.38 $\times 10^{10}$	**7.33** $\times 10^7$	8.40 $\times 10^7$	**0.55%**	0.61%
Live-journal	1.72 $\times 10^{11}$	**1.68** $\times 10^{11}$	**6.68** $\times 10^8$	8.02 $\times 10^8$	**0.39%**	0.48%
Wiki-en	2.36 $\times 10^{11}$	**2.30** $\times 10^{11}$	**8.30** $\times 10^8$	1.29 $\times 10^9$	**0.35%**	0.56%
Delicious	4.13 $\times 10^{11}$	**4.03** $\times 10^{11}$	**1.01** $\times 10^9$	1.63 $\times 10^9$	**0.24%**	0.40%
Tracker	2.39 $\times 10^{11}$	**2.34** $\times 10^{11}$	**6.20** $\times 10^8$	7.29 $\times 10^9$	**0.26%**	0.31%
Orkut	2.69 $\times 10^{12}$	**2.58** $\times 10^{12}$	**7.21** $\times 10^9$	8.38 $\times 10^9$	**0.27%**	0.33%
Bi-twitter	4.54 $\times 10^{12}$	**4.49** $\times 10^{12}$	**1.35** $\times 10^{10}$	3.04 $\times 10^{10}$	**0.30%**	0.68%
Bi-sk	1.64 $\times 10^{12}$	**1.58** $\times 10^{12}$	2.29 $\times 10^9$	**2.01** $\times 10^9$	0.14%	**0.13%**
Bi-uk	6.15 $\times 10^{12}$	**6.00** $\times 10^{12}$	3.67 $\times 10^9$	**3.21** $\times 10^9$	0.06%	**0.05%**

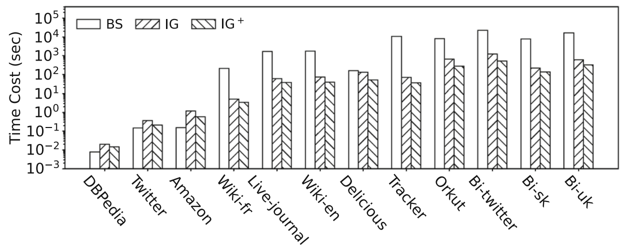


Fig. 20 Performance on different datasets

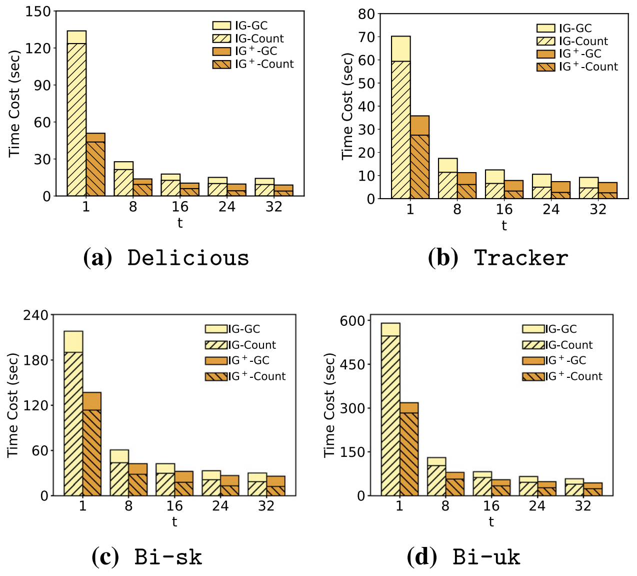


Fig. 21 Varying *t*

BFCB-IG$^+$ are slightly slower than BFCB-BS on small datasets (i.e., `DBPedia`, `Twitter`, and `Amazon`). This is because when the datasets are small, the number of updates (i.e., b) is also very small. Thus, the updating process of BFCB-BS can be finished efficiently without counting a large number of butterflies while BFCB-IG and BFCB-IG$^+$ need to construct the affected graph at first. In the following experiments, we omit the BFCB-BS algorithm since the algorithms BFCB-IG and BFCB-IG$^+$ significantly outperform BFCB-BS as evaluated here.

Evaluating the effect of t. Figure 21 shows the result of the BFCB-IG and BFCB-IG$^+$ algorithms in parallel by varying the thread number t from 1 to 32 on 4 large datasets (with $b = 0.5\%$). We also evaluate the breakdown of execution time (i.e., the execution time of different stages) including the affected subgraph construction stage (`-GC`) and the butterfly counting stage (`-Count`). Note that these two stages both run in parallel. We can see that the parallelization techniques improve the efficiency of both BFCB-IG and BFCB-IG$^+$ significantly. In addition, the speedup of the butterfly counting stage is higher than the speedup of the affected subgraph construction stage for both of the algorithms. This is because there do not exist many writing conflicts in the butterfly counting stage, and the updating/writing operations are need to be performed to construct the affected subgraph which may lead to extra latency when the same memory location is updated simultaneously by multiple threads.

Evaluating the effect of b. Here, we evaluate the effect of the batch size b by setting b from 0.01 to 2% of the original number of edges (m) in the dataset. We first show the number of affected butterflies in Fig. 22. It is observed that the number of affected butterflies can be very large even the batch-size is small. For instance, on `Bi-uk`, the number of affected butterflies reaches 3.80×10^{13} with $b = 2\%$ (the original butterfly counts on `Bi-uk` is 4.89×10^{14} as shown in Table 1).

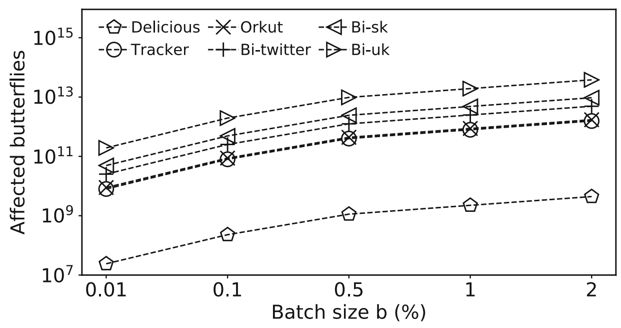


Fig. 22 The number of affected butterflies w.r.t. the batch size b

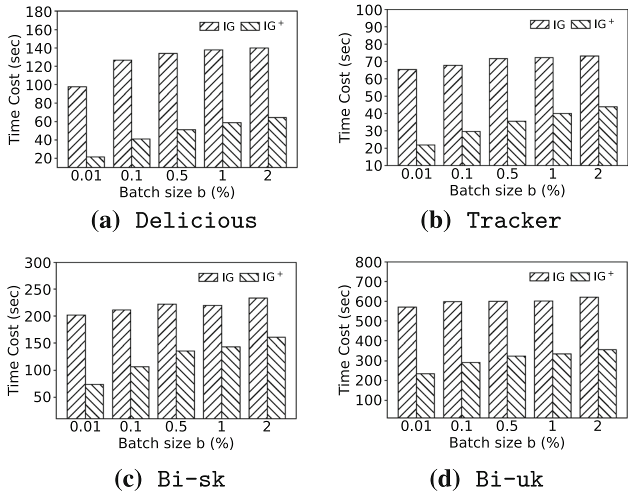


Fig. 23 Time cost w.r.t. the batch size b

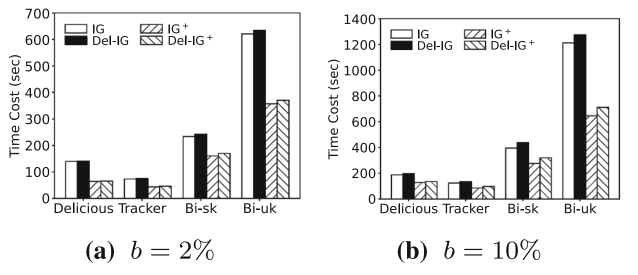


Fig. 24 Evaluating the deletion cases, varying b

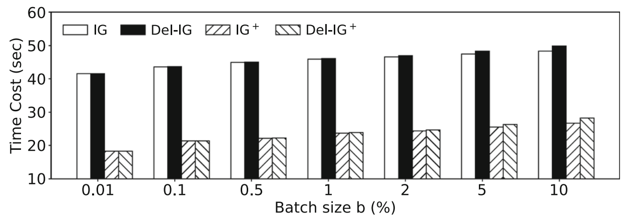


Fig. 25 Performance on `MoiveLense`

In Fig. 23, we show the processing time of the BFCB-IG and BFCB-IG$^+$ algorithms when varying b. We can see that BFCB-IG$^+$ consistently outperforms BFCB-IG for all values of b. Especially, when b becomes smaller, the margin between BFCB-IG$^+$ and BFCB-IG becomes larger. This is because BFCB-IG$^+$ can reduce the traversal of wedges in the original graph and take advantage of the small number of updated edges.

Evaluating the deletion cases. In this part, we evaluate the performance of the algorithms for both insertion and deletion cases.

(1) Firstly, we evaluate the performance of the algorithms on `Delicious`, `Tracker`, `Bi-sk`, and `Bi-uk` by setting $b = 2\%$ and $b = 10\%$ in Fig. 24. In deletion cases, we randomly extract a proportion b of edges from the input graph and consider these edges as the deletion edges. When using BFCB-IG and BFCB-IG$^+$ for handling deletion cases, we denote these two algorithms as Del-IG and Del-IG$^+$, respectively. We can observe that the overhead for deletion cases is relatively small when b=2% and becomes large when b=10%, especially on datasets `Bi-sk` and `Bi-uk`.

(2) Secondly, we run the algorithms on a dynamic graph `MovieLens`, which contains 162,000 users (U), 62,000 movies (L), 25,000,095 ratings (E) and the timestamp of the ratings. In real scenarios, the edge insertion/deletion operations usually run in a sliding-window manner. Thus, in insertion cases, we consider a proportion b of edges with the latest timestamp as the insertion edges. In deletion cases, we consider a proportion b of edges with the oldest timestamp as the deletion edges. The performance of the algorithms is shown in Fig. 25. We can see that the time cost of the algorithms increases with the increase of b in both insertion and deletion cases. In addition, the overhead for deletion cases also increases when b becomes large. Note that compared to insertion cases, the total time cost for handling deletion cases is only slightly higher since the dominant cost in the algorithms is to maintain the number of butterflies.

9 Related work

Motif counting in unipartite graphs. Triangle is the smallest non-trivial cohesive structure and there are extensive studies on counting triangles in the literature [5,9,16,19,31,32,38,57, 57,58,60–62]. However, the butterfly counting is inherently different from the triangle counting for two reasons, (1) the number of butterflies may be significantly larger than that of triangles ($O(m^2)$ versus $O(m^{1.5})$ in the worst case), and (2) the structures are inherently different. Thus, the existing triangle counting techniques are not applicable to efficient butterfly counting because the existing techniques for counting triangles (e.g., [19,60]) are based on enumerating all triangles and the enumeration is not affordable in counting butterflies due to the quadratic number $O(m^2)$ of butterflies. There are also some studies [33,34,52] focusing on the other cohesive structures such as 4-vertices and 5-vertices. In [6], the authors propose generic matrix-multiplication based algorithm for counting the cycles of length k $(3 \leq k \leq 7)$ in $O(n^{2.376})$ time and $O(n^2)$ space. While the algorithm in [6] can be used to solve our problem, it cannot process large graphs due to its space and time complexity. As shown in [64], the algorithm in [64] has a significant improvement over [6], while our algorithm significantly improves [64].

Bipartite graphs. Some studies are conducted toward motifs such as 3×3 biclique [14] and 4-path [47]. These structures are different from the butterfly thus these works also cannot be used to solve the butterfly counting problem. Algorithms for accurate estimation of the number of butterflies in graph streams is studied [55]. The authors in [40] propose an approach that applies sampling and sketching techniques for approximate butterfly counting in graph streams which outperforms the algorithms in [55]. Independent from our work, in [59], the authors study the parallel implementation of butterfly counting algorithms. [59] mainly focuses on parallel algorithms for butterfly computations including butterfly counting and wing/tip decomposition, while we propose efficient butterfly counting algorithms under different settings including in-memory algorithms, parallel algorithms, I/O efficient algorithms, and batch-dynamic algorithms. When implementing the parallel butterfly counting algorithms, [59] uses different data structures compared to us. [59] also provides a more detailed theoretical analysis about the work/span of parallel algorithms. Based on the butterfly structure, the k-bitruss (or k-wing) and k-tip decomposition problems are studied in the literature [56,59,67,68,73]. Recently, the authors in [36] propose a new coarse-to-fine framework for parallel tip decomposition.

Dynamic graphs. There are many existing studies on triangle counting [15,28,61], butterfly counting [44], and other motif counting [11,46] on dynamic graphs. Stefani et al. [61] propose a streaming algorithm for triangle counting on dynamic graphs with both edge insertions and deletions considered. Bulteau et al. [15] introduce a one-pass algorithm for triangle counting estimation. In [28], an edge-sampling-based framework is proposed to augment traditional models for triangle counting and improve computational and memory costs. In [44], the authors present a distributed implementation of butterfly counting in dynamic environments. As shown in [46], the authors propose a suite of approaches to count the number of motifs by calculating the number of embeddings of each motif. In addition, counting multi-layer temporal motifs on dynamic graphs is studied in [11].

In the literature, there are also various works that consider the batch-dynamic settings [1,8,23,27,43]. Ediger et al. [23] present an algorithm for computing clustering coefficients in batch-dynamic graphs. A batch algorithm of the single-source dynamic shortest path is proposed in [8]. In [1], the authors present a parallel algorithm for batch-dynamic connectivity problems. In order to improve the computational efficiency, [27] propose GPU-based algorithms under the batch-dynamic setting. Makkar et al. [43] study the exact triangle counting algorithm in batch-dynamic graphs.

Graph ordering. There are some studies on specific graph algorithms using graph ordering. Then et al. [63] optimize BFS algorithms. Park et al. [51] improve the CPU cache performance of many classic graph algorithms such as Bellman-Fold and Prim. The authors in [29] present a suite of approaches to accelerate set intersections in graph algorithms. Since these techniques are specific to the problems studied, they are not applicable to butterfly counting.

In the literature, there are also works studying general graph ordering methods to speed up graph algorithms [7,10,12,13,18,22,35,70]. In the experiments, we show that our cache-aware techniques outperform the state-of-the-art technique [70]; that is, our cache-aware strategy is more suitable for counting butterflies.

10 Conclusion

We study the *butterfly counting* problem in this paper. We propose a vertex-priority-based butterfly counting algorithm BFC-VP which can effectively handle high-degree vertices. We also propose the cache-aware butterfly counting algorithm BFC-VP^{++} which improves the CPU cache performance of BFC-VP with two cache-aware strategies. Efficient butterfly counting algorithms for batch-dynamic graphs are also investigated. We conduct comprehensive experimental evaluations and the result shows that our vertex-priority-based solutions outperform the baseline algorithms significantly.

Acknowledgements Wenjie Zhang is supported by ARC FT210100303 and ARC DP200101116. Ying Zhang is supported by FT170100128. Lu Qin is supported by ARC FT200100787 and DP210101347.

Funding Open Access funding enabled and organized by CAUL and its Member Institutions

References

1. Acar, U.A., Anderson, D., Blelloch, G.E., Dhulipala, L.: Parallel batch-dynamic graph connectivity. In: The 31st ACM Symposium on Parallelism in Algorithms and Architectures, pp. 381–392 (2019)
2. Aggarwal, A., Vitter, J., et al.: The input/output complexity of sorting and related problems. Commun. ACM **31**(9), 1116–1127 (1988)
3. Ailamaki, A., DeWitt, D.J., Hill, M.D., Wood, D.A.: Dbmss on a modern processor: Where does time go? In: PVLDB, number DIAS-CONF-1999-001, pp. 266–277 (1999)
4. Aksoy, S.G., Kolda, T.G., Pinar, A.: Measuring and modeling bipartite graphs with community structure. J. Complex Netw. **5**(4), 581–603 (2017)
5. Al Hasan, M., Dave, V.S.: Triangle counting in large networks: a review. Wiley Interdiscipl. Rev. Data Min. Knowl. Discov. **8**(2), e1226 (2018)
6. Alon, N., Yuster, R., Zwick, U.: Finding and counting given length cycles. Algorithmica **17**(3), 209–223 (1997)
7. Auroux, L., Burelle, M., Erra, R.: Reordering very large graphs for fun and profit. In: International Symposium on Web Algorithms (2015)
8. Bauer, R., Wagner, D.: Batch dynamic single-source shortest-path algorithms: An experimental study. In: International Symposium on Experimental Algorithms, pp. 51–62. Springer (2009)
9. Becchetti, L., Boldi, P., Castillo, P., Gionis, A.: Efficient semi-streaming algorithms for local triangle counting in massive graphs. In: KDD, pp. 16–24. ACM (2008)
10. Blandford, D.K., Blelloch, G.E., Kash, I.A.: Compact representations of separable graphs. In: ACM-SIAM Symposium on Discrete Algorithms, pp. 679–688. Society for Industrial and Applied Mathematics (2003)
11. Boekhout, H.D., Kosters, W.A., Takes, F.W.: Efficiently counting complex multilayer temporal motifs in large-scale networks. Comput. Soc. Netw. **6**(1), 1–34 (2019)
12. Boldi, P., Rosa, M., Santini, M., Vigna, S.: Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In: WWW, pp. 587–596. ACM (2011)
13. Boldi, P., Santini, M., Vigna, S.: Permuting web graphs. In: International Workshop on Algorithms and Models for the Web-Graph, pp. 116–126. Springer (2009)
14. Borgatti, S.P., Everett, M.G.: Network analysis of 2-mode data. Soc. Netw. **19**(3), 243–269 (1997)
15. Bulteau, L., Froese, V., Kutzkov, K., Pagh, R.: Triangle counting in dynamic graph streams. Algorithmica **76**(1), 259–278 (2016)
16. Chang, L., Zhang, C., Lin, X., Qin, L.: Scalable top-k structural diversity search. In: ICDE, pp. 95–98. IEEE (2017)
17. Chiba, N., Nishizeki, T.: Arboricity and subgraph listing algorithms. SIAM J. Comput. **14**(1), 210–223 (1985)
18. Chierichetti, F., Kumar, R., Lattanzi, S., Mitzenmacher, M., Panconesi, A., Raghavan, P.: On compressing social networks. In: KDD, pp. 219–228. ACM, (2009)
19. Chu, S., Cheng, J.: Triangle listing in massive networks. TKDD **6**(4), 17 (2012)
20. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT Press, London (2009)
21. Dhillon, I.S.: Co-clustering documents and words using bipartite spectral graph partitioning. In: KDD, pp. 269–274. ACM (2001)
22. Dhulipala, L., Kabiljo, I., Karrer, B., Ottaviano, G., Pupyrev, S., Shalita, A.: Compressing graphs and indexes with recursive graph bisection. In: KDD, pp. 1535–1544. ACM (2016)
23. Ediger, D., Jiang, K., Riedy, K., Bader, D.A.: Massive streaming data analytics: A case study with clustering coefficients. In: 2010 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW), pp. 1–8. IEEE (2010)
24. Fain, D.C., Pedersen, J.O.: Sponsored search: A brief history. Bull. Am. Soc. Inf. Sci. Technol. **32**(2), 12–13 (2006)
25. Fang, Y., Huang, X., Qin, L., Zhang, Y., Zhang, W., Cheng, R., Lin, X.: A survey of community search over big graphs. VLDB J. **29**(1), 353–392 (2020)
26. Fang, Y., Wang, K., Lin, X., Zhang, W.: Cohesive subgraph search over big heterogeneous information networks: Applications, challenges, and solutions. In: Proceedings of the 2021 International Conference on Management of Data, pp. 2829–2838 (2021)
27. Green, O., Bader, D.A.: custinger: Supporting dynamic graph algorithms for gpus. In: 2016 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–6. IEEE (2016)
28. Han, G., Sethu, H.: Edge sample and discard: A new algorithm for counting triangles in large dynamic graphs. In: 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM), pp. 44–49. IEEE (2017)
29. Han, S., Zou, L., Yu, J.X.: Speeding up set intersections in graph algorithms using simd instructions. In: SIGMOD, pp. 1587–1602. ACM (2018)
30. He, Y., Wang, K., Zhang, W., Lin, X., Zhang, Y.: Exploring cohesive subgraphs with vertex engagement and tie strength in bipartite graphs. Inf. Sci. **572**, 277–296 (2021)
31. Hu, X., Tao, Y., Chung, C.-W.: Massive graph triangulation. In: SIGMOD, pp. 325–336. ACM (2013)
32. Itai, A., Rodeh, M.: Finding a minimum circuit in a graph. SIAM J. Comput. **7**(4), 413–423 (1978)
33. Jain, S., Seshadhri, C.: A fast and provable method for estimating clique counts using turán's theorem. In: WWW, pp. 441–449. International World Wide Web Conferences Steering Committee (2017)
34. Jha, M., Seshadhri, C., Pinar, A.: Path sampling: A fast and provable method for estimating 4-vertex subgraph counts. In: WWW, pp. 495–505. International World Wide Web Conferences Steering Committee, (2015)
35. Kang, U., Faloutsos, C.: Beyond' caveman communities': Hubs and spokes for graph compression and mining. In: ICDM, pp. 300–309. IEEE (2011)
36. Kannan, R., Prasanna, V.K., De Rose, C.A.F. et al.: Receipt: Refine coarse-grained independent tasks for parallel tip decomposition of bipartite graphs. In: Proceedings of the VLDB Endowment, 2020, Estados Unidos. (2020)
37. Khaouid, W., Barsky, M., Srinivasan, V., Thomo, A.: K-core decomposition of large networks on a single pc. PVLDB **9**(1), 13–23 (2015)
38. Kolountzakis, M.N., Miller, G.L., Peng, R., Tsourakakis, C.E.: Efficient triangle counting in large graphs via degree-based vertex partitioning. Internet Math. **8**(1–2), 161–185 (2012)
39. Latapy, M., Magnien, C., Del Vecchio, N.: Basic notions for the analysis of large two-mode networks. Soc. Netw. **30**(1), 31–48 (2008)

40. Li, R., Wang, P., Jia, P., Zhang, P., Zhao, J., Tao, J., Yuan, Y., Guan, X.: Approximately counting butterflies in large bipartite graph streams. IEEE Trans. Knowl. Data Eng. (2021)
41. Lind, P.G., Gonzalez, M.C., Herrmann, H.J.: Cycles and clustering in bipartite networks. Phys. Rev. E **72**(5), 056127 (2005)
42. Liu, B., Yuan, L., Lin, X., Qin, L., Zhang, W., Zhou, J.: Efficient (α, β)-core computation: An index-based approach. In: WWW, pp. 1130–1141. ACM (2019)
43. Makkar, D., Bader, D. A., Green, O.: Exact and parallel triangle counting in dynamic graphs. In: 2017 IEEE 24th International Conference on High Performance Computing (HiPC), pp. 2–12. IEEE (2017)
44. Malkhi, D., Naor, M., Ratajczak, D.: Viceroy: A scalable and dynamic emulation of the butterfly. In: Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing, pp. 183–192 (2002)
45. Milo, R., Shen-Orr, S., Itzkovitz, S., Kashtan, N., Chklovskii, D., Alon, U.: Network motifs: simple building blocks of complex networks. Science **298**(5594), 824–827 (2002)
46. Mukherjee, K., Hasan, M.M., Boucher, C., Kahveci, T.: Counting motifs in dynamic networks. BMC Syst. Biol. **12**(1), 6 (2018)
47. Opsahl, T.: Triadic closure in two-mode networks: Redefining the global and local clustering coefficients. Soc. Netw. **35**(2), 159–167 (2013)
48. Ornstein, M.: Interlocking directorates in Canada: Intercorporate or class alliance? Admin. Sci. Quarterly 210–231 (1984)
49. Ornstein, M.D.: Interlocking directorates in Canada: evidence from replacement patterns. Soc. Netw. **4**(1), 3–25 (1982)
50. Palmer, D.: Broken ties: Interlocking directorates and intercorporate coordination. Adminis. Sci. Q. 40–55 (1983)
51. Park, J.-S., Penner, M., Prasanna, V.K.: Optimizing graph algorithms for improved cache performance. IEEE Trans. Parallel Distrib. Syst. **15**(9), 769–782 (2004)
52. Pinar, A., Seshadhri, C., Vishal, V.: Escape: Efficiently counting all 5-vertex subgraphs. In: WWW, pp. 1431–1440. International World Wide Web Conferences Steering Committee (2017)
53. Robins, G., Alexander, M.: Small worlds among interlocking directors: network structure and distance in bipartite graphs. Comput. Math. Organ. Theory **10**(1), 69–94 (2004)
54. Sanei-Mehri, S.-V., Sariyuce, A. E., Tirthapura, S.: Butterfly counting in bipartite networks. In: KDD, pp. 2150–2159. ACM (2018)
55. Sanei-Mehri, S.-V., Zhang, Y., Sariyüce, A. E., Tirthapura, S.: Fleet: butterfly estimation from a bipartite graph stream. In: CIKM, pp. 1201–1210 (2019)
56. Sarıyüce, A.E., Pinar, A.: Peeling bipartite networks for dense subgraph discovery. In: WSDM, pp. 504–512. ACM (2018)
57. Schank, T., Wagner, D.: Finding, counting and listing all triangles in large graphs, an experimental study. In: International Workshop on Experimental and Efficient Algorithms, pp. 606–609. Springer (2005)
58. Seshadhri, C., Pinar, A., Kolda, T.G.: Triadic measures on graphs: The power of wedge sampling. In: SDM, pp. 10–18. SIAM (2013)
59. Shi, J., Shun, J.: Parallel algorithms for butterfly computations. In: Symposium on Algorithmic Principles of Computer Systems, pp. 16–30. SIAM (2020)
60. Shun, J., Tangwongsan, K.: Multicore triangle computations without tuning. In: ICDE, pp. 149–160. IEEE (2015)
61. Stefani, L.D., Epasto, A., Riondato, M., Upfal, E.: Triest: Counting local and global triangles in fully dynamic streams with fixed memory size. TKDD **11**(4), 43 (2017)
62. Suri, S., Vassilvitskii, S.: Counting triangles and the curse of the last reducer. In: WWW, pp. 607–614. ACM (2011)
63. Then, M., Kaufmann, M., Chirigati, F., Hoang-Vu, T.-A., Pham, K., Kemper, A., Neumann, T., Vo, H.T.: The more the merrier: efficient multi-source graph traversal. PVLDB **8**(4), 449–460 (2014)
64. Wang, J., Fu, A.W.-C., Cheng, J.: Rectangle counting in large bipartite graphs. In: BigData Congress, pp. 17–24. IEEE (2014)
65. Wang, K., Cao, X., Lin, X., Zhang, X., Qin, L.: Efficient computing of radius-bounded k-cores. In: ICDE, pp. 233–244. IEEE (2018)
66. Wang, K., Lin, X., Qin, L., Zhang, W., Zhang, Y.: Vertex priority based butterfly counting for large-scale bipartite networks. In: PVLDB (2019)
67. Wang, K., Lin, X., Qin, L., Zhang, W., Zhang, Y.: Efficient bitruss decomposition for large-scale bipartite graphs. In: ICDE, pp. 661–672. IEEE (2020)
68. Wang, K., Lin, X., Qin, L., Zhang, W., Zhang, W.: Towards efficient solutions of bitruss decomposition for large-scale bipartite graphs. In: VLDB Journal, pp. 1–24 (2021)
69. Wang, K., Zhang, W., Lin, X., Zhang, Y., Qin, L., Zhang, Y.: Efficient and effective community search on large-scale bipartite graphs. In: ICDE. IEEE (2021)
70. Wei, H., Yu, J.X., Lu, C., Lin, X.: Speedup graph processing by graph ordering. In: SIGMOD, pp. 1813–1828. ACM (2016)
71. Zhang, F., Zhang, Y., Qin, L., Zhang, W., Lin, X.: When engagement meets similarity: efficient (k, r)-core computation on social networks. PVLDB **10**(10), 998–1009 (2017)
72. Zhang, F., Zhang, Y., Qin, L., Zhang, W., Lin, X.: Efficiently reinforcing social networks over user engagement and tie strength. In: ICDE, pp. 557–568. IEEE (2018)
73. Zou, Z.: Bitruss decomposition of bipartite graphs. In: DASFAA, pp. 218–233. Springer (2016)