Vertex Priority Based Butterfly Counting for Large-scale Bipartite Networks

Kai Wang[†‡], Xuemin Lin[†‡], Lu Qin[*], Wenjie Zhang[†], Ying Zhang[*]

[†]University of New South Wales, Australia   [‡]Zhejiang Lab, China   [*]University of Technology Sydney, Australia
kai.wang@unsw.edu.au, {lxue,zhangw}@cse.unsw.edu.au, {lu.qin, ying.zhang}@uts.edu.au

ABSTRACT

Bipartite networks are of great importance in many real-world applications. In bipartite networks, butterfly (i.e., a complete 2×2 biclique) is the smallest non-trivial cohesive structure and plays a key role. In this paper, we study the problem of efficient counting the number of butterflies in bipartite networks. 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 networks. 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. Moreover, we also show that our proposed techniques can work efficiently in external and parallel contexts. Our extensive empirical studies demonstrate that the proposed techniques can speed up the state-of-the-art techniques by up to two orders of magnitude for the real datasets.

1. INTRODUCTION

When modelling relationships between two different types of entities, the bipartite network arises naturally as a data model in many real-world applications [23, 44]. For example, in online shopping services (e.g., Amazon and Alibaba), the purchase relationships between users and products can be modelled as a bipartite network, where users form one layer, products form the other layer, and the links between users and productions represent purchase records as shown in Figure 1. Other examples include author-paper relationships, actor-movie networks, etc.
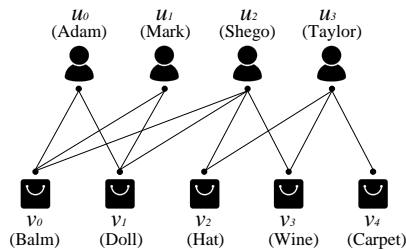
Figure 1: A bipartite network

Since network motifs (i.e., repeated sub-graphs) are regarded as basic building blocks of complex networks [48], finding and counting motifs of networks is a key to network analysis. In unipartite networks, there are extensive studies on counting and listing triangles (the smallest non-trivial clique) in the literature [16, 24, 26, 36, 42, 43, 61–65]. In bipartite networks, *butterfly* (i.e., a complete 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 [15, 58–60, 68, 76]; for instance, Figure 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 [15].

In this paper, we study the problem of *butterfly counting*, that is to compute the number of butterflies in a bipartite graph G, 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.
Network measurement. The *bipartite clustering coefficient* [15, 46, 51, 58] is a cohesiveness measurement of bipartite networks. Given a bipartite graph G, its bipartite clustering coefficient equals $4 \times \bowtie_G / \Join_G$, where $\Join_G$ is the number of caterpillars in G — the number of three-paths. For example, (u_0, v_0, u_1, v_1) in Figure 1 is a three-path. High bipartite clustering coefficient indicates localized closeness and redundancy in bipartite networks [15, 58]; for instance, in user-product networks, bipartite clustering coefficients can be used frequently to analyse the sale status for products in different categories. These statistics can also be used in `Twitter` network [10] for internet advertising where the `Twitter` network is the bipartite network consisting of Twitter users and the URLs they mentioned in their postings. Since $\Join_G$ can be easily computed in $O(m)$ time where m is the number of edges in G [15], computing $\bowtie_G$ 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 mod-

elled as a butterfly. These butterflies can reflect inter-corporate relations [52–54]. 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 [58].

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 [60] (or k-bitruss [76]) for a given k where k-wing is the maximum 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 [30, 56, 73, 75], word-document clustering [27], and viral marketing [29, 47, 69, 74]. Given a bipartite graph G, the proposed algorithms [60, 76] for k-wing computation is 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, the demand of butterfly counting in large networks can be very high. However, the state-of-the-art algorithms cannot efficiently handle large-scale bipartite networks. As shown in [59], on the `Tracker` network with 10^8 edges, their algorithm needs about 9,000 seconds to compute $\bowtie_G$. 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 [60] needs to iteratively remove the edges with the lowest number of butterflies; the number has to be exact.

State-of-the-art. Consider that there can be $O(m^2)$ butterflies in the worst case. Wang et al. in [68] 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 Figure 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 $C_3^2 (= 3)$ 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 will first use 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 [68] is $O(\sum_{u \in U(G)} deg_G(u)^2))$ if the lower layer $L(G)$ of G is chosen to have start-vertices, where $U(G)$ is the upper layer. Sanei et al. in [59] 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 [59, 68], the dominant cost is at Step 1 that enumerates wedges to compute 2-hop reachable vertices and their hits. For example, regarding Figure 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 Figure 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 $2,002$ vertices and $3,000$ edges in Figure 2(a). In G, u_0 is connected with $1,000$ vertices (v_0 to v_{999}), v_{1000} is also connected with $1,000$ vertices (u_1 to u_{1000}), and for $0 \leq i \leq 999$, 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_{1000}^2 ($= 499,500$) plus $1,000$ different wedges by the existing algorithms [59, 68].
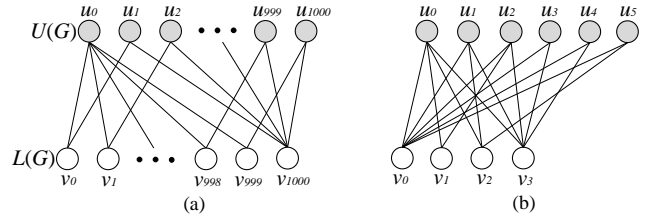

Figure 2: Some observations

Challenges. The main challenges of efficient butterfly counting are twofold.

1. Using high-degree vertices as middle-vertices of wedges may generate numerous wedges to be scanned. The existing techniques [59, 68], including the layer-priority-based techniques [59], 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.

Our approaches. To address Challenge 1, instead of the existing layer-priority-based algorithm, we propose a vertex-priority-based butterfly counting 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. The higher degree, the higher priority; and the ties are broken by vertex ID. For example, regarding Figure 2(a), the BFC-VP algorithm will choose u_0 and v_{1000} as start-vertices; consequently, only $2,000$ 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 [59], $O(\min\{\sum_{v \in U(G)} deg_G(v)^2, \sum_{v \in L(G)} deg_G(v)^2\})$, considering $deg_G(v)^2 = \sum_{(u,v) \in E(G)} deg_G(v)$ where v is fixed.

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 [14], 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 $O(\sum_{(u,v) \in E(G)} min\{deg_G(u), deg_G(v)\})$ 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 Figure 2(b), 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 projection strategy to improve the cache performance by storing high-priority (more frequently accessed) end-vertices together to reduce the cache-miss [71]. Considering the example in Figure 2(b), BFC-VP^{++} will store v_0 and v_3 together after projection.

Contribution. Our principal contributions are summarized as follows.

- We propose a novel algorithm BFC-VP to count the butterflies that significantly reduce the time complexities of the existing algorithms in both theory and practice.
- We propose a novel cache-aware butterfly counting algorithm BFC-VP^{++} by adopting cache-aware strategies to BFC-VP. The BFC-VP^{++} algorithm achieves better CPU cache performance than BFC-VP.
- We can replace the exact counting algorithm in the approximate algorithm [59] by our exact counting algorithm for a speedup.
- By extending our framework, we present an external-memory algorithm and a parallel algorithm for butterfly counting.
- We conduct extensive experiments on real bipartite networks. The result shows that our proposed algorithms BFC-VP and BFC-VP^{++} outperform the state-of-the-art algorithms by up to 2 orders of magnitude. For instance, the BFC-VP^{++} algorithm can count 10^{12} butterflies in 50 seconds on `Tracker` dataset with 10^8 edges, while the state-of-the-art butterfly counting algorithm [59] runs about $9,000$ seconds.

Organization. The rest of the paper is organized as follows. The related work follows immediately. Section 2 presents the problem definition. Section 3 introduces the existing algorithms BFC-BS and BFC-IBS. The BFC-VP algorithm is presented in Section 4. Section 5 explores cache-awareness. Section 6 extends our algorithm to count butterflies against each edge, the parallel execution of our proposed

algorithms and the external memory solution. Section 7 reports experimental results. Section 8 concludes the paper.

Related Work.

Motif counting in unipartite networks. Triangle is the smallest non-trivial cohesive structure and there are extensive studies on counting triangles in the literature [16, 19, 24, 26, 36, 37, 42, 43, 61, 61–65]. 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 ($O(m^2)$ vs $O(m^{1.5})$ in the worst case), and 2) the structures are different (4-hops' circle vs 3-hops' circle). Thus, the existing triangle counting techniques are not applicable to efficient butterfly counting because the existing techniques for counting triangles (e.g., [26, 63]) 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 in the worst case.

There are also some studies [38, 39, 57] focusing on the other cohesive structures such as 4-vertices and 5-vertices, these techniques also cannot be used to solve our problem. In [17], the authors propose generic matrix-multiplication based algorithm for counting the cycles of length k ($3 \le k \le 7$) in $O(n^{2.376})$ time and $O(n^2)$ space. While the algorithm in [17] can be used to solve our problem, it cannot process large graphs due to its space and time complexity. As shown in [68], the algorithm in [68] has a significant improvement over [17], while our algorithm significantly improves [68].

Bipartite Networks. Some studies are conducted towards motifs such as 3×3 biclique [23] and 4-path [51]. These structures are different from the butterfly thus these works also cannot be used to solve the butterfly counting problem. As mentioned earlier, the study in this paper aims to improve the recent works in [59, 68].

Graph ordering. There are some studies on specific graph algorithms using graph ordering. Then et al. [66] optimize BFS algorithms. Park et al. [55] improve the CPU cache performance of many classic graph algorithms such as Bellman-Fold and Prim. The authors in [34] 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 recent works studying general graph ordering methods to speed up graph algorithms [18, 20–22, 25, 28, 40, 71, 72]. In the experiments, we show that our cache-aware techniques outperform the state-of-the-art technique [71]; that is, our cache-aware strategy is more suitable for counting butterflies.

2. PROBLEM DEFINITION

In this section, we formally introduce the notations and definitions. Mathematical notations used throughout this paper are summarized in Table 1.

Our problem is defined over an undirected bipartite graph $G(V = (U, L), E)$, where $U(G)$ denotes the set of vertices in the upper layer, $L(G)$ denotes the set of vertices in the lower layer, $U(G) \cap L(G) = \emptyset$, $V(G) = U(G) \cup L(G)$ denotes the vertex set, and $E(G) \subseteq U(G) \times L(G)$ denotes the edge set. We use n and m to denote the number of vertices and edges in G, respectively, and we assume $m > n$. In addition, we use r and l to denote the number of vertices in $U(G)$ and $L(G)$, respectively. An edge between two vertices u and v in G is denoted as (u, v) or (v, u). The set of neighbors of a vertex u in G is denoted as $N_G(u) = \{v \in V(G) \mid (u, v) \in$

Table 1: The summary of notations

Notation	Definition
G	a bipartite graph
$V(G)$	the vertex set of G
$E(G)$	the edge set of G
$U(G), L(G)$	a vertex layer of G
u, v, w, x	a vertex in the bipartite graph
$e, (u, v)$	an edge in the bipartite graph
(u, v, w)	a wedge formed by u, v, w
$[u, v, w, x]$	a butterfly formed by u, v, w, x
$deg_G(u)$	the degree of vertex u
$p(u)$	the priority of vertex u
$N_G(u)$	the set of neighbors of vertex u
$2hop_G(u)$	the set of two-hop neighbors of vertex u
$\boxtimes_e$	the number of butterflies containing an edge e
$\boxtimes_G$	the number of butterflies in G
n, m	the number of vertices and edges in G ($m > n$)

$E(G)$}, and the degree of u is denoted as $deg_G(u) = |N_G(u)|$. The set of 2-hop neighbors of u (i.e., the set of vertices which are exactly two edges away from u) is denoted as $2hop_G(u)$. Each vertex u has a unique id and we assume for every pair of vertices $u \in U(G)$ and $v \in L(G)$, $u.id > v.id$.

DEFINITION 1 (WEDGE). *Given a bipartite graph $G(V, E)$ and vertices $u, v, w \in V(G)$. A path starting from u, going through v and ending at w is called a wedge which is denoted as (u, v, w). For a wedge (u, v, w), we call u the start-vertex, v the middle-vertex and w the end-vertex.*

DEFINITION 2 (BUTTERFLY). *Given a bipartite graph G and the four vertices $u, v, w, x \in V(G)$ where $u, w \in U(G)$ and $v, x \in L(G)$, a butterfly induced by the vertices u, v, w, x is a (2,2)-biclique of G; that is, u and w are all connected to v and x, respectively, by edges.*

A butterfly induced by vertices u, v, w, x is denoted as $[u, v, w, x]$. We denote the number of butterflies containing a vertex u as $\boxtimes_u$, the number of butterflies containing an edge e as $\boxtimes_e$ and the number of butterflies in G as $\boxtimes_G$.
Problem Statement. Given a bipartite graph $G(V, E)$, our *butterfly counting* problem is to compute $\boxtimes_G$.

3. EXISTING SOLUTIONS

In this section, we briefly discuss the two existing algorithms, the baseline butterfly counting algorithm BFC-BS [68] and the improved baseline butterfly counting algorithm BFC-IBS [59]. As discussed earlier, both algorithms are based on enumerating wedges. The following Lemma 1 [68] is a key to the two algorithms.

LEMMA 1. *Given a bipartite graph $G(V, E)$ and a vertex $u \in G$, we have the following equations:*

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

$$\boxtimes_G = \frac{1}{2} \sum_{u \in U(G)} \boxtimes_u = \frac{1}{2} \sum_{v \in L(G)} \boxtimes_v \quad (2)$$

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. The details of the BFC-IBS algorithm are shown in Algorithm 1.

Note that to avoid counting a butterfly twice, for each middle-vertex $v \in N_G(u)$ and the corresponding end-vertex

---

Algorithm 1: BFC-IBS

Input: $G(V = (U, L), E)$: the input bipartite graph
Output: $\boxtimes_G$

1 $\boxtimes_G \leftarrow 0$
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   initialize hashmap *count_wedge* with zero
7   **foreach** $v \in N_G(u)$ **do**
8   **foreach** $w \in N_G(v) : w.id > u.id$ **do**
9           $count\_wedge(w) \leftarrow count\_wedge(w) + 1$
10   **foreach** $w \in count\_wedge$ **do**
11   **if** $count\_wedge(w) > 1$ **then**
12           $\boxtimes_G \leftarrow \boxtimes_G + \binom{count\_wedge(w)}{2}$
13 **return** $\boxtimes_G$

$w \in N_G(v)$, 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.

As shown, the time complexity of BFC-BS is $O(\sum_{v \in L(G)} deg_G(v)^2)$ if starting from the layer $U(G)$, while the time complexity of BFC-IBS is $O(min\{\sum_{u \in U(G)} deg_G(u)^2, \sum_{v \in L(G)} deg_G(v)^2\})$.
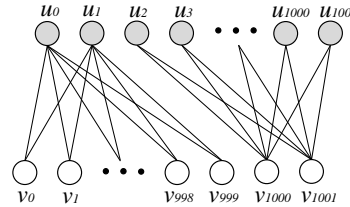
4. ALGORITHM BY VERTEX PRIORITY

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

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 (e.g., $U(G)$), the number of processed wedges is decided by the sum of degree squares of middle-vertices in the other layer (e.g., $\sum_{v \in L(G)} deg_G(v)^2$). If all the vertices with lower-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. 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 2,002 vertices and 4,000 edges in Figure 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 1,000 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 Figure 3 can be constructed in 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 Figure 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). The time complexity of BFC-VP is $O(\sum_{(u,v)\in E(G)} min\{deg_G(u), deg_G(v)\})$.

Given a graph G, the BFC-VP algorithm first assigns a *priority* to each vertex $u \in V(G)$ which is defined as follows.

DEFINITION 3 (PRIORITY). *Given a bipartite graph $G(V, E)$, for a vertex $u \in V(G)$, the priority $p(u)$ is an integer where $p(u) \in [1, |V(G)|]$. For two vertices $u, v \in V(G)$, $p(u) > p(v)$ if*

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

Given the priority, a butterfly can always be constructed 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 which 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 Figure 3). 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 unecessary 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 which has higher priority than the start-vertex (or meet a middle-vertex which has higher priority than the start-vertex). The details of the BFC-VP algorithm are shown in Algorithm 2.

Algorithm 2: BFC-VP

Input: $G(V = (U, L), E)$: the input bipartite graph
Output: $\boxtimes_G$

1 Compute $p(u)$ for each $u \in V(G)$ // Definition 3
2 Sort $N(u)$ for each $u \in V(G)$ according to their priorities
3 $\boxtimes_G \leftarrow 0$
4 **foreach** $u \in V(G)$ **do**
5   initialize hashmap *count_wedge* with zero
6   **foreach** $v \in N_G(u) : p(v) < p(u)$ **do**
7   **foreach** $w \in N_G(v) : p(w) < p(u)$ **do**
8              $count\_wedge(w) \leftarrow count\_wedge(w) + 1$
9   **foreach** $w \in count\_wedge$ **do**
10   **if** $count\_wedge(w) > 1$ **then**
11              $\boxtimes_G \leftarrow \boxtimes_G + \binom{count\_wedge(w)}{2}$
12 **return** $\boxtimes_G$

Given a bipartite graph G, BFC-VP first 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 the wedges from each start-vertex $u \in V(G)$ and initializes the hashmap *count_wedge* with zero. For each middle-vertex $v \in N_G(u)$, we process v if $p(v) < p(u)$ according to the processing rule. Then, to avoid duplicate counting, we only process $w \in N_G(v)$ with $p(w) < p(u)$. After running lines 4 - 8, we get $|N_G(u) \cap N_G(w)|$ (i.e., *count_wedge(w)*) for the start-vertex u and the end-vertex w. Then, according to Lemma 1, BFC-VP computes $\boxtimes_G$. Finally, we return $\boxtimes_G$.

Analysis of the BFC-VP algorithm. Below we show the correctness and the time complexity of BFC-VP.
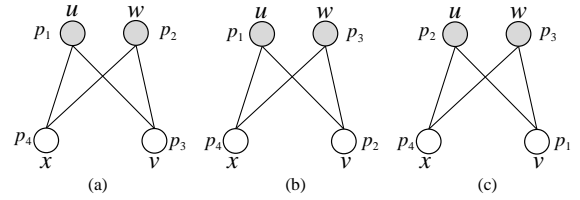

Figure 4: Assume $p_4 > p_3 > p_2 > p_1$

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

PROOF. We prove that BFC-VP correctly computes $\boxtimes_G$ 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 as shown in Figure 4 (the other situations can be transformed into the above by a symmetric conversion), where p_i is the corresponding vertex priority. Regarding the case in Figure 4(a), 4(b), or 4(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. $\square$

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

PROOF. The Algorithm 2 has two phases: initializing in the first phase and computing $\boxtimes_G$ 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 [41]. 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 related to the number of processed wedges and each wedge needs $O(1)$ time to process. In BFC-VP, we only need to process the wedges where the degrees of middle-vertices are lower or equal than the degrees of start-vertices based on the processing rule of BFC-VP and Definition 3. Considering an edge $(u, v) \in E(G)$ connecting a start-vertex u and a middle-vertex v, BFC-VP needs to process $O(deg_G(v))$ end-vertices from (u, v). That is, for each edge $(u, v) \in E(G)$, BFC-VP needs to process $O(min\{deg_G(u), deg_G(v)\})$ wedges since the middle-vertex has a lower or equal degree than the start-vertex in a processed wedge. In total, BFC-VP needs to process $O(\sum_{(u,v)\in E(G)} min\{deg_G(u), deg_G(v)\})$ wedges. Therefore, the time complexity of BFC-VP is $O(\sum_{(u,v)\in E(G)} min\{deg_G(u), deg_G(v)\})$. $\square$

THEOREM 3. *The space complexity of BFC-VP is $O(m)$.*

PROOF. This theorem is immediate. $\square$

LEMMA 2. *Given a bipartite graph G, we have the following equation:*

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

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:

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

Similarly,

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

Thus, we can prove that Equation 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 bipartite graph G in Figure 3.* BFC-VP *first assigns a priority to each vertex in G where $p(u_1) > p(u_0) > p(v_{1001}) > p(v_{1000}) > p(u_{1001}) > p(u_{1000}) > ... > p(v_1) > p(v_0)$. Starting from u_1,* BFC-VP *needs to process $1,000$ wedges ending at u_0. Similarly, starting from v_{1001},* BFC-VP *needs to process $1,000$ wedges ending at v_{1000}. No other wedges need to be processed by* BFC-VP. *In total,* BFC-VP *needs to process 2,000 wedges.*

BFC-IBS *processes each vertex $u \in U(G)$ as start-vertex. Starting from u_0,* BFC-IBS *needs to process $1,000$ wedges ending at u_1. Starting from u_1, no wedges need to be processed. In addition, starting from the vertices in $\{u_2, u_3, ..., u_{1001}\}$,* 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 Section 1, below is the breakdown of memory accesses to vertices required when processing the wedges: $O(n)$ 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 [14], we try to improve the CPU cache performance when accessing the end-vertices.
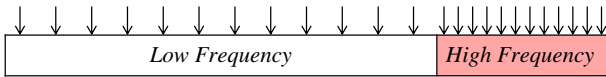

Figure 5: The buffer BF

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 and BF is partitioned into a *low-frequency area*

LFA and a *high-frequency area HFA* as shown in Figure 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:

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

We want to minimize F which is computed by:

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

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 which 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 projection which stores high-priority vertices together in *HFA*.

5.1 Cache-aware Wedge Processing
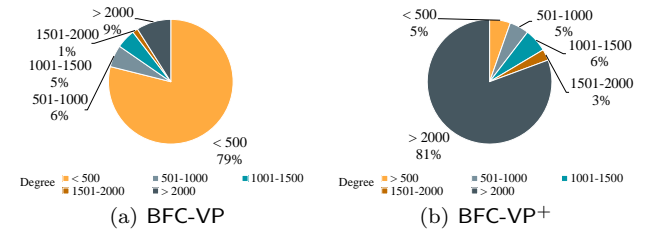

(a) BFC-VP (b) BFC-VP$^+$

Figure 6: The degree distribution of the end-vertex-accesses on Tracker

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 Figure 6(a), 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* is 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 Figure 2(b), we have $p(v_0) > p(v_3) > p(u_0) > p(v_2) > p(v_1)$ according to their degrees. We analyse 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, in Figure 2(b), 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 Figure 6(b), 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.

Time complexity unchanged. Although the new wedge processing strategy can improve the CPU cache performance of BFC-VP, there are two questions 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+}, we have the following lemma.

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, every vertex $u \in G$ has a unique $p(u)$ 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.

5.2 Cache-aware Graph Projection

Before Projection: sorted by id
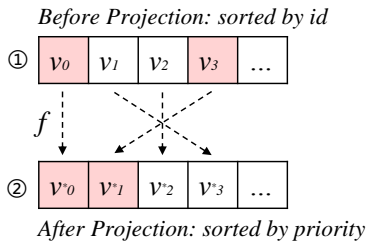
After Projection: sorted by priority

Figure 7: Illustrating the cache-aware graph projection

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 Figure 2(b) 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 ≥ 1000, and only 1% vertices ($400,000$) with degree ≥ 37. Motivated by the above observations, we propose the graph projection strategy which can further improve the cache performance.

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

DEFINITION 4 (PROJECTION GRAPH). *Given a bipartite graph $G(V, E)$, a projection graph $G^*(V, E)$ is defined as: $G^* \leftarrow projection(G, f)$, 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 $u^*.id = rankU(u) + l$, $v^*.id = rankL(v)$. Here, $rankU(u) \in [0, r-1]$ ($rankL(v) \in [0, l-1]$) denotes the rank of the priority of $u \in U(G)$ (the rank of the priority of $v \in L(G)$).*

Unlike the conventional graph projection method in [49, 50] which projects a bipartite graph to a unipartite graph, our linear graph projection 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 projecting. After projecting the original graph G into the projection 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 projection using the example in Figure 2(b). After obtaining the projection 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 projection is 3. In the experiments, we prove that the algorithms applying with the graph projection strategy achieves 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, details of the algorithm BFC-VP^{++} are shown in Algorithm 3. Given a bipartite graph G, BFC-VP^{++} first generates a projection graph G^* according to Definition 4 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 running lines 6 - 12, we get $|N_G(u^*) \cap N_G(w^*)|$ (i.e., $count\_wedge(w^*)$) for the start-vertex u^* and the end-vertex $w^* \in 2hop_G(u^*)$. Finally, we compute $\boxtimes_G$ (lines 13 - 15).

THEOREM 4. *The BFC-VP^{++} algorithm correctly solves the butterfly counting problem.*

PROOF. We prove that BFC-VP^{++} correctly computes $\boxtimes_G$ for a bipartite graph G. Since the graph projection strategy just renumbers the vertices, it does not affect the structure

---

Algorithm 3: BFC-VP^{++}

Input: $G(V = (U, L), E)$: the input bipartite graph
Output: $\mathbb{X}_G$

1   $\mathbb{X}_G \leftarrow 0$
2   $G^* \leftarrow projection(G, f)$ // Definition 4
3   compute $p(u^*)$ for each $u^* \in V(G^*)$ // Definition 3
4   sort $N(u^*)$ for each $u^* \in V(G^*)$ according to their priorities
5   **foreach** $u^* \in V(G^*)$ **do**
6   initialize hashmap $count\_wedge$ 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           $count\_wedge(w^*) \leftarrow count\_wedge(w^*) + 1$
11   **else**
12   break
13   **foreach** $w^* \in count\_wedge$ **do**
14   **if** $count\_wedge(w^*) > 1$ **then**
15         $\mathbb{X}_G \leftarrow \mathbb{X}_G + \binom{count\_wedge(w^*)}{2}$
16   **return** $\mathbb{X}_G$

---

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 Figure 4. Regarding the case in Figure 4(a), 4(b), or 4(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 5. *The time complexity of* BFC-VP^{++} *is* $O(\sum_{(u,v) \in E(G)} min\{deg_G(u), deg_G(v)\})$.

PROOF. The Algorithm 3 has two phases including the initialization phase and $\mathbb{X}_G$ computation phase. In the first phase, similar with BFC-VP, the algorithm needs $O(n + m)$ time to compute the priority number, sort the neighbors of vertices and compute the projection graph. Secondly, since we can use $O(1)$ time to process one wedge, we analyse the number of processed wedges by BFC-VP^{++} as follows. In BFC-VP^{++}, we only need to process the wedges where the degree of end-vertex is higher or equal than the middle-vertex. Considering an edge $(u, v) \in E(G)$ connecting an end-vertex u and a middle-vertex v, we need to process $O(deg_G(v))$ wedges containing (u, v). Thus, we need to process $O(\sum_{(u,v) \in E(G)} min\{deg_G(u), deg_G(v)\})$ wedges in total. Therefore, the time complexity of BFC-VP^{++} is $O(\sum_{(u,v) \in E(G)} min\{deg_G(u), deg_G(v)\})$. □

THEOREM 6. *The space complexity of* BFC-VP^{++} *is* $O(m)$.

PROOF. This theorem is immediate. □

Remark. The cache-aware strategies proposed in this section are not applicable for the algorithms BFC-BS and BFC-IBS. This is because these strategies are priority-based strategies while the algorithms BFC-BS and BFC-IBS are not priority-based algorithms.

6. EXTENSIONS

In this section, firstly, we extend our algorithms to compute $\mathbb{X}_e$ for each edge 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

Given an edge e in G, we have the following equation [68]:

$$\mathbb{X}_{e=(u,v)} = \sum_{w \in 2hop_G(u), w \in N_G(v)} (|N_G(u) \cap N_G(w)| - 1)$$

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

Based on the above equation, our BFC-VP^{++} algorithm can be extended to compute $\mathbb{X}_e$ for each edge e in G. 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 $\mathbb{X}_e$.

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

In Algorithm 4, we only need an extra array to store $\mathbb{X}_e$ for each edge e. In addition, because it just runs the wedge processing procedure twice, the time complexity of BFC-EVP^{++} is also $O(\sum_{(u,v) \in E(G)} min\{deg_G(u), deg_G(v)\})$.

Algorithm 4: BFC-EVP^{++}

Input: $G(V = (U, L), E)$: the input bipartite graph
Output: $\mathbb{X}_e$ for each $e \in E(G)$

1   run Algorithm 3 Line 2 - Line 4
2   $\mathbb{X}_e \leftarrow 0$ for each $e \in E(G)$
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           $\delta = count\_wedge(w) - 1$
9           $(v, w) \leftarrow f^{-1}(v^*, w^*)$
10           $(u, v) \leftarrow f^{-1}(u^*, v^*)$
11           $\mathbb{X}_{(u,v)} \leftarrow \mathbb{X}_{(u,v)} + \delta$
12           $\mathbb{X}_{(v,w)} \leftarrow \mathbb{X}_{(v,w)} + \delta$
13   **else**
14   break
15   **return** $\mathbb{X}_e$ *for each* $e \in E(G)$

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 $count\_wedge$ and $\mathbb{X}_G$ 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 generate $local\_count\_wedge$ and $local\_\mathbb{X}_G$ 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 is the number of threads.

Scheduling. In the parallel algorithm, we also need to consider the schedule strategies which may affect the load balance. We denote the workload for each start-vertex u as $u.l$. We want to minimize the makespan L:

$$L = \max_{1 \le i \le t} \left(\sum_{u \in V_i} u.l \right) \qquad (9)$$

Here, V_i is the set of start-vertices on thread i.

The minimization of L is a well-known NP-hard optimization problem (i.e., the multiprocessor scheduling problem) [31] assuming we know the exact $u.l$ for each start-vertex u. Nevertheless, the workload is unknown prior, this makes the problem more challenging.
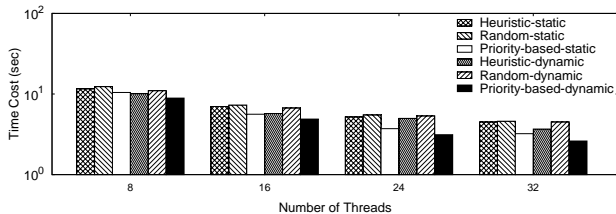
Below we discuss two types of schedule strategies: the dynamic scheduling and the static scheduling.

Dynamic scheduling. For the dynamic scheduling, we queue all start-vertices in a specified order. Once a thread is idle, we dequeue a start-vertex and allocate it to the idle thread. The dynamic scheduling always delivers a schedule that has makespan at most $(2 - \frac{1}{t})L_{opt}$ where L_{opt} is the optimal makespan even if the workload is unknown in advance [32]. The bound is further reduced to $(\frac{4}{3} - \frac{1}{3t})L_{opt}$ by scheduling the job (i.e., start-vertex) with longer processing time (i.e., larger workload) first [33]. Since the vertex order may affect the performance, we compare three ordering strategies:
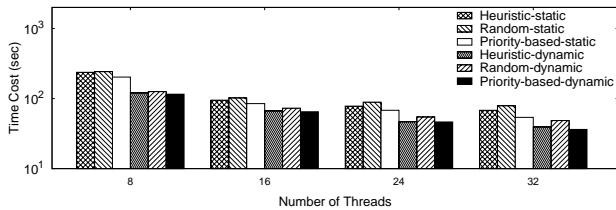
1) Heuristic strategy. Sort start-vertices in non-ascending order by their estimated workloads (i.e., $\widetilde{u.l}$ for u);

2) Random strategy. Sort start-vertices in random order.

3) Priority-based strategy. Sort start-vertices in non-ascending order by their priorities.

To make our investigation more complete, we also consider the static scheduling. For the static scheduling, we need to pre-compute the allocations of start-vertices on the threads.

Static scheduling. Here we discuss and compare three allocation strategies:

1) Heuristic strategy. We first estimate the workload for each start-vertex u as: $\widetilde{u.l} = |S|$, where $S = \{w|w \in N_{G^*}(v), v \in N_{G^*}(u), p(w) > p(v)\}$. After that, we sort start-vertices by their estimated workloads in non-ascending order. Then we sequentially allocate these vertices and for each start-vertex, we always allocate it to the thread which has the minimal workload so far.

2) Random strategy. For each start-vertex u, we randomly allocate u to a thread under a uniform distribution.

3) Priority-based strategy. First, we sort start-vertices according to their priorities in non-ascending order. Then, for each start-vertex u with priority $p(u)$, we allocate u to the thread i if $p(u) \bmod t = i - 1$.

(a) `Tracker`, varying the number of threads

(b) `Bi-twitter`, varying the number of threads

Figure 8: Effect of scheduling

In Figure 8, we compare the performance of the scheduling strategies by applying them into the parallel BFC-VP^{++} algorithm on `Tracker` and `Bi-twitter` datasets. As shown in Figure 8, since the priority-based strategies are light-weight greedy strategies, they perform better than the other strategies in the static scheduling and the dynamic scheduling, respectively. Note that the heuristic strategies need additional pre-computation and the random strategies cannot achieve good performance. In addition, the priority-based dynamic scheduling strategy performs the best on these datasets.

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

Input: $G(V = (U, L), E)$: the input bipartite graph, t: number of threads
Output: $\boxtimes_G$ for graph G
1 run Algorithm 3 Line 1 - Line 4
2 initialize $local\_count\_wedge[i]$ and $local\_\boxtimes_G[i]$ 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 $count\_wedge$, $\boxtimes_G$ with $local\_count\_wedge[i]$, $local\_\boxtimes_G[i]$
7 /* on master thread */
8 $\boxtimes_G \leftarrow \boxtimes_G + local\_\boxtimes_G[i]$ for each thread $i \leftarrow 1..t$
9 **return** $\boxtimes_G$

The algorithm BFC-VP^{++} in parallel. Since the priority-based dynamic scheduling strategy performs the best in the evaluation, we use it in our parallel algorithms. The details of the algorithm BFC-VP^{++} in parallel are shown in Algorithm 5. Given a bipartite graph G, we first generate a projection 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 $local\_count\_wedge[i]$ and $local\_\boxtimes_G[i]$. After all the threads finishing their computation, we compute $\boxtimes_G$ on the master thread.

Remark. The strategies presented here can be easily applied to the BFC-BS, BFC-IBS, and BFC-VP algorithms.

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 $\boxtimes_G$ based on Lemma 1 (lines 8 - 10).

I/O complexity analysis. We use the standard notations in [13] to analyse 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 dom-

Algorithm 6: BFC-EM

Input: $G(V = (U, L), E)$: the input bipartite graph
Output: $\boxtimes_G$

1. sort all the edges $e \in G$ on disk
2. compute $p(u)$ for each $u \in V(G)$ on disk // Definition 3
3. $\boxtimes_G \leftarrow 0$
4. **foreach** *vertex* $v \in G$ **do**
5.   **forall** $u, w \in N_G(v) : p(w) > p(v), p(w) > p(u)$ *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. $\boxtimes_G \leftarrow \boxtimes_G + \binom{count\_pair(u,w)}{2}$
11. **return** $\boxtimes_G$

---

inate cost is to scan and sort the vertex-pairs. Since there are $O(\sum_{(u,v) \in E(G)} min\{deg_G(u), deg_G(v)\})$ vertex-pairs generated by BFC-EM, the I/O complexity of BFC-EM is $O(scan(\sum_{(u,v) \in E(G)} min\{deg_G(u), deg_G(v)\}) + sort(\sum_{(u,v) \in E(G)} min\{deg_G(u), deg_G(v)\}))$.

7. EXPERIMENTS

In this section, we present the results of empirical studies. In particular, our empirical studies have been conducted against the following algorithms: 1) the state-of-the art BFC-IBS in [59] as the baseline algorithm (we thank the authors for providing the code), 2) BFC-VP in Section 4, 3) BFC-VP$^+$ in Section 5.1, 4) BFC-VP^{++} in Section 5.3, 5) BFC-EIBS, BFC-EVP, BFC-EVP^{++} by extending BFC-IBS, BFC-VP and BFC-VP^{++}, respectively, to compute $\boxtimes_e$ for each edge e in G, 6) the parallel version of BFC-IBS, BFC-VP and BFC-VP^{++}, 7) the most advanced approximate butterfly counting algorithm BFC-ESap in [59], 8) BFC-ESap$_{vp++}$ 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.

The algorithms are implemented in C++ and the experiments are run on a Linux server with $2 \times$ Intel Xeon E5-2698 processors and 512GB main memory. Although most empirical studies have been against single core, we want our empirical studies to be conducted on the same computer as the evaluation of parallel performance. *We terminate an algorithm if the running time is more than 10 hours.*

7.1 Datasets

We use 12 datasets in our experiments including all the 9 real datasets in [59] to ensure the fairness. We add 3 more datasets to evaluate the scalability of our techniques.

The 9 real datasets we used are `DBPedia` [5], `Twitter` [10], `Amazon` [1], `Wiki-fr` [12], `Wiki-en` [11], `Live-journal` [7], `Delicious` [6], `Tracker` [9] and `Orkut` [8].

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` [3], `sk-2005` [2] and `uk-2006-05` [4]). In order 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.

The summary of datasets is shown in Table 2. U and L are vertex layers, $|E|$ is the number of edges. $\boxtimes_G$ is the number of butterflies. $\sum_{u \in L} d(u)^2$ and $\sum_{v \in R} d(v)^2$ represent the sum of degree squares for L and R, respectively. TC_{ibs}

is computed by $min\{\sum_{u \in L} d(u)^2, \sum_{v \in R} d(v)^2\}$ which is the time complexity bound of BFC-IBS. TC_{new} is computed by $\sum_{(u,v) \in E(G)} min\{deg_G(u), deg_G(v)\}$ which is the time complexity bound of BFC-VP and BFC-VP^{++}.

7.2 Performance Evaluation

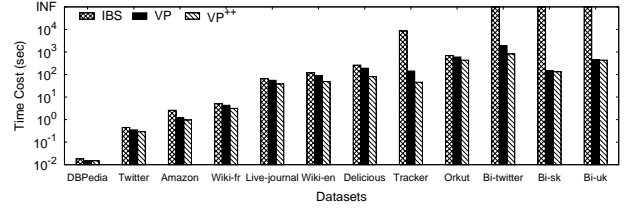In this section, we evaluate the performance of the proposed algorithms.
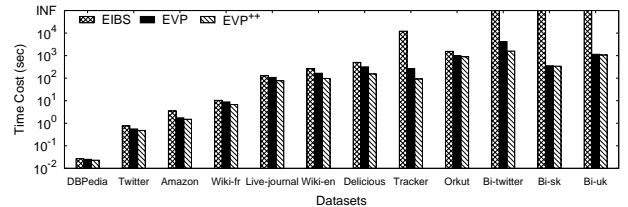

Figure 9: Performance on different datasets

Figure 10: Performance on different datasets (Counting the number of butterflies containing each edge e in G)

Evaluating the performance on all the datasets. In Figure 9, we show the performance of the BFC-IBS, BFC-VP and BFC-VP^{++} algorithms on different datasets. We can observe that BFC-VP^{++} is the most efficient algorithm, while BFC-VP also outperforms BFC-IBS. This is because the BFC-VP^{++} algorithm utilizes both the vertex-priority based optimization and the cache-aware strategies which significantly reduce the computation cost. On `Tracker`, the BFC-VP and BFC-VP^{++} algorithms are at least two orders of magnitude faster than the BFC-IBS algorithm. On `Bi-twitter`, `Bi-sk` and `Bi-uk`, the BFC-IBS algorithm cannot finish within 10 hours. This is because the degree distribution of these datasets are skewed and high-degree vertices exist in both layers. For instance, TC_{ibs} is more than $100\times$ 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 this situation efficiently.

In Figure 10, we show the performance of the algorithms which compute $\boxtimes_e$ for each edge e in G. The performance differences of these algorithms follow similar trends to those in Figure 9. We can also observe that the BFC-EVP^{++} algorithm is the most efficient algorithm.
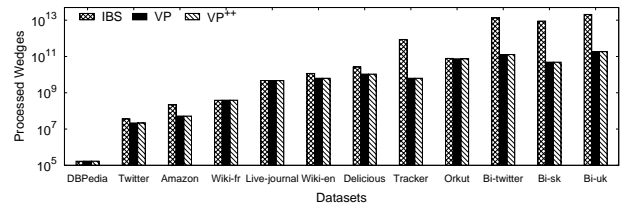

Figure 11: The number of processed wedges

Evaluating the number of processed wedges. In Figure 11, we show the number of processed wedges of the algorithms on all the datasets. We can observe that on `Tracker`, `Bi-twitter`, `Bi-sk` and `Bi-uk` datasets, BFC-IBS needs to process $100\times$ more wedges than BFC-VP and BFC-VP^{++}.

Table 2: Summary of Datasets

Dataset	$|E|$	$|U|$	$|L|$	$\boxtimes_G$	$\sum_{u\in L} d(u)^2$	$\sum_{v\in R} d(v)^2$	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}

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.

(a) Wiki-en, varying n (b) Delicious, varying n

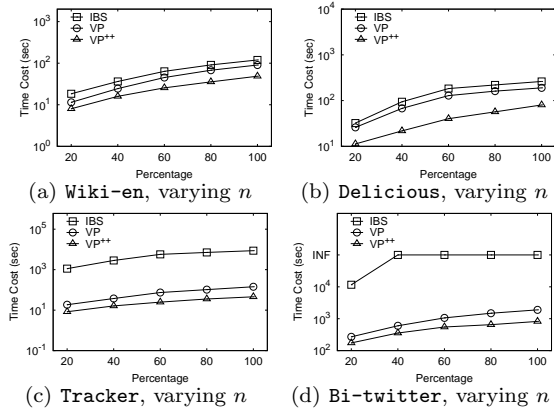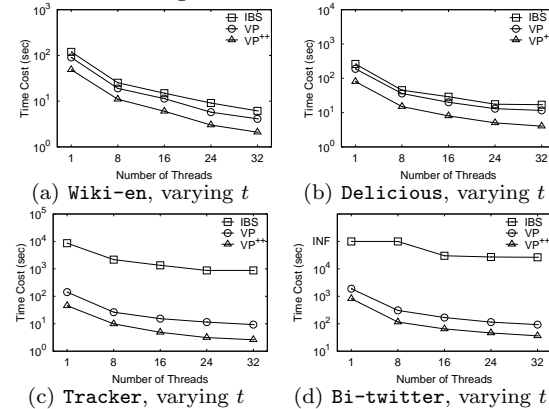(c) Tracker, varying n (d) Bi-twitter, varying n

Figure 12: Effect of graph size

Scalability. *Evaluating the effect of graph size.* Figure 13 studies the scalability of the algorithms by varying the graph size n on four datasets. When varying n, we randomly sample 20% to 100% vertices of the original graphs, and construct the induced subgraphs using these vertices. We can observe that, BFC-VP and BFC-VP^{++} are scalable and the computation cost of them all 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.

(a) Wiki-en, varying t (b) Delicious, varying t

(c) Tracker, varying t (d) Bi-twitter, varying t

Figure 13: Effect of t

Speedup. *Evaluating the parallelization.* Figure 13 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. 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 computation costs of the BFC-VP and BFC-VP^{++} algorithms in parallel decrease when the number of threads increases and the algorithm BFC-VP^{++} in parallel is more efficient than the other algorithms.

Table 3: Cache Statistics over Wiki-en

Algorithm	$Cache\text{-}ref$	$Cache\text{-}m$	$Cache\text{-}mr$	$Time(s)$
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 4: Cache Statistics over Delicious

Algorithm	$Cache\text{-}ref$	$Cache\text{-}m$	$Cache\text{-}mr$	$Time(s)$
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 5: Cache Statistics over Tracker

Algorithm	$Cache\text{-}ref$	$Cache\text{-}m$	$Cache\text{-}mr$	$Time(s)$
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 6: Cache Statistics over Bi-twitter

Algorithm	$Cache\text{-}ref$	$Cache\text{-}m$	$Cache\text{-}mr$	$Time(s)$
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

Evaluating the cache-aware strategies. In Table 3, Table 4, Table 5 and Table 6, 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 projection 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 projection, 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.
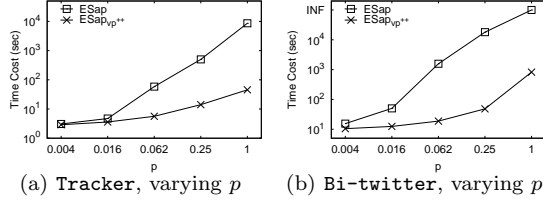

(a) `Tracker`, varying p (b) `Bi-twitter`, varying p

Figure 14: Effect of p

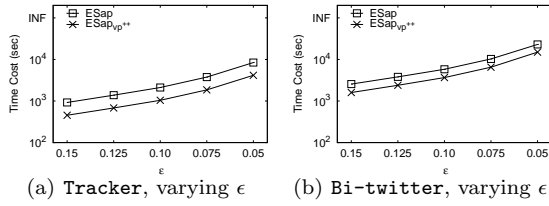(a) `Tracker`, varying ϵ (b) `Bi-twitter`, varying ϵ

Figure 15: Effect of ϵ

Speeding up the approximate butterfly counting algorithm. In the approximate algorithm BFC-ESap [59], 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 butterfies exactly, the approximate algorithm BFC-ESap$_{vp++}$ can be obtained by applying BFC-VP^{++} in BFC-ESap without changing the theoretical guarantee.

In Figure 14, 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 $Pr[|\hat{\boxtimes}_G - \boxtimes_G| > \epsilon\boxtimes_G] \leq \delta$ as shown in [59]. We vary ϵ and fix $\delta = 0.1$ and p as the optimal p as suggested in [59]. We can see that for these two approximate algorithms, the time costs are increased on these two datasets in order to get a better accuracy and BFC-ESap$_{vp++}$ significantly outperforms BFC-ESap as mentioned before.
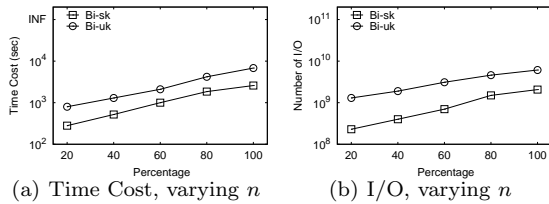

(a) Time Cost, varying n (b) I/O, varying n

Figure 16: Evaluating the external memory algorithm

Evaluating the external memory algorithm. In Figure 16, 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.

Table 7: Time cost compared with Gorder

Dataset	Renumbering time		Computation time		Total time	
	Projection	Gorder	Projection	Gorder	Projection	Gorder
DBPedia	**0.01**	0.04	**0.02**	0.03	**0.03**	0.07
Twitter	**0.06**	4.26	**0.25**	0.29	**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 8: Cache statistics compared with Gorder

Dataset	Cache reference		Cache miss		Cache miss ratio	
	Projection	Gorder	Projection	Gorder	Projection	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%**

Graph projection vs Gorder. In [71], the authors proposed the *Gorder* model to reduce the cache miss in graph algorithms. Here, we replace the graph projection with Gorder in BFC-VP^{++} and evaluate the difference of performances.

Table 7 shows the time cost. We can observe that the renumbering time cost of the graph projection is much less than Gorder on all datasets. This is because graph projection 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 projection 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 projection is better than Gorder.

Table 8 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 projection achieves a better CPU performance than Gorder on almost all the datasets (i.e., less cache misses and less cache miss ratios on 9 datasets) when dealing with the butterfly counting problem with the BFC-VP^{++} algorithm.

In summary, our graph projection strategy is more suitable when dealing with the butterfly counting problem.

8. CONCLUSION

In this paper, we study the *butterfly counting* problem. 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. We conduct extensive experiments on real datasets and the result shows that our BFC-VP^{++} algorithm significantly outperforms the state-of-the-art algorithms. In the future, we plan to study the butterfly counting problem in a distributed environment [45,67] or a data stream [35,70].

9. ACKNOWLEDGMENT

10. REFERENCES

[1] Amazon. http://liu.cs.uic.edu/download/data/.
[2] Bi-sk. http://law.di.unimi.it/webdata/sk/.
[3] Bi-twitter.
http://an.kaist.ac.kr/traces/WWW2010.html.
[4] Bi-uk.
http://law.di.unimi.it/webdata/uk-2006-05/.
[5] Dbpedia. http://wiki.dbpedia.org/Downloads.
[6] Delicious. http://dai-labor.de/IRML/datasets.
[7] Live-journal. http://socialnetworks.mpi-sws.org.
[8] Orkut. http://socialnetworks.mpi-sws.org.
[9] Tracker. https://ssc.io/trackingthetrackers/.
[10] Twitter.
http://public.asu.edu/~mdechoud/datasets.html.
[11] Wiki-en. http://dumps.wikimedia.org/.
[12] Wiki-fr. http://dumps.wikimedia.org/.
[13] A. Aggarwal, J. Vitter, et al. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
[14] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. Dbmss on a modern processor: Where does time go? In *PVLDB*, number DIAS-CONF-1999-001, pages 266–277, 1999.
[15] S. G. Aksoy, T. G. Kolda, and A. Pinar. Measuring and modeling bipartite graphs with community structure. *Journal of Complex Networks*, 5(4):581–603, 2017.
[16] M. Al Hasan and V. S. Dave. Triangle counting in large networks: a review. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 8(2):e1226, 2018.
[17] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.
[18] L. Auroux, M. Burelle, and R. Erra. Reordering very large graphs for fun & profit. In *International Symposium on Web AlGorithms*, 2015.
[19] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *KDD*, pages 16–24. ACM, 2008.
[20] D. K. Blandford, G. E. Blelloch, and I. A. Kash. Compact representations of separable graphs. In *ACM-SIAM symposium on Discrete algorithms*, pages 679–688. Society for Industrial and Applied Mathematics, 2003.
[21] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *WWW*, pages 587–596. ACM, 2011.
[22] P. Boldi, M. Santini, and S. Vigna. Permuting web graphs. In *International Workshop on Algorithms and Models for the Web-Graph*, pages 116–126. Springer, 2009.
[23] S. P. Borgatti and M. G. Everett. Network analysis of 2-mode data. *Social networks*, 19(3):243–269, 1997.
[24] L. Chang, C. Zhang, X. Lin, and L. Qin. Scalable top-k structural diversity search. In *ICDE*, pages 95–98. IEEE, 2017.
[25] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *KDD*, pages 219–228. ACM, 2009.

[26] S. Chu and J. Cheng. Triangle listing in massive networks. *TKDD*, 6(4):17, 2012.
[27] I. S. Dhillon. Co-clustering documents and words using bipartite spectral graph partitioning. In *KDD*, pages 269–274. ACM, 2001.
[28] L. Dhulipala, I. Kabiljo, B. Karrer, G. Ottaviano, S. Pupyrev, and A. Shalita. Compressing graphs and indexes with recursive graph bisection. In *KDD*, pages 1535–1544. ACM, 2016.
[29] D. C. Fain and J. O. Pedersen. Sponsored search: A brief history. *Bulletin of the american Society for Information Science and technology*, 32(2):12–13, 2006.
[30] Y. Fang, X. Huang, L. Qin, Y. Zhang, W. Zhang, R. Cheng, and X. Lin. A survey of community search over big graphs. *VLDB Journal*, 2019.
[31] M. R. Garey and D. S. Johnson. *Computers and intractability*, volume 29. wh freeman New York, 2002.
[32] R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9):1563–1581, 1966.
[33] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 17(2):416–429, 1969.
[34] S. Han, L. Zou, and J. X. Yu. Speeding up set intersections in graph algorithms using simd instructions. In *SIGMOD*, pages 1587–1602. ACM, 2018.
[35] M. R. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. *External memory algorithms*, 50:107–118, 1998.
[36] X. Hu, Y. Tao, and C.-W. Chung. Massive graph triangulation. In *SIGMOD*, pages 325–336. ACM, 2013.
[37] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. *SIAM Journal on Computing*, 7(4):413–423, 1978.
[38] S. Jain and C. Seshadhri. A fast and provable method for estimating clique counts using turán's theorem. In *WWW*, pages 441–449. International World Wide Web Conferences Steering Committee, 2017.
[39] M. Jha, C. Seshadhri, and A. Pinar. Path sampling: A fast and provable method for estimating 4-vertex subgraph counts. In *WWW*, pages 495–505. International World Wide Web Conferences Steering Committee, 2015.
[40] U. Kang and C. Faloutsos. Beyond'caveman communities': Hubs and spokes for graph compression and mining. In *ICDM*, pages 300–309. IEEE, 2011.
[41] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo. K-core decomposition of large networks on a single pc. *PVLDB*, 9(1):13–23, 2015.
[42] M. N. Kolountzakis, G. L. Miller, R. Peng, and C. E. Tsourakakis. Efficient triangle counting in large graphs via degree-based vertex partitioning. *Internet Mathematics*, 8(1-2):161–185, 2012.
[43] M. Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical computer science*, 407(1-3):458–473, 2008.
[44] M. Latapy, C. Magnien, and N. Del Vecchio. Basic notions for the analysis of large two-mode networks. *Social networks*, 30(1):31–48, 2008.
[45] X. Lin, M. M. Orlowska, and Y. Zhang. On data

allocation with minimum overall communication costs in distributed database design. In *ICCI*, pages 539–544. IEEE, 1993.

[46] P. G. Lind, M. C. Gonzalez, and H. J. Herrmann. Cycles and clustering in bipartite networks. *Physical review E*, 72(5):056127, 2005.

[47] B. Liu, L. Yuan, X. Lin, L. Qin, W. Zhang, and J. Zhou. Efficient (α, β)-core computation: An index-based approach. In *WWW*, pages 1130–1141. ACM, 2019.

[48] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[49] M. E. Newman. Scientific collaboration networks. i. network construction and fundamental results. *Physical review E*, 64(1):016131, 2001.

[50] M. E. Newman. Scientific collaboration networks. ii. shortest paths, weighted networks, and centrality. *Physical review E*, 64(1):016132, 2001.

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

[52] M. Ornstein. Interlocking directorates in canada: Intercorporate or class alliance? *Administrative science quarterly*, pages 210–231, 1984.

[53] M. D. Ornstein. Interlocking directorates in canada: Evidence from replacement patterns. *Social Networks*, 4(1):3–25, 1982.

[54] D. Palmer. Broken ties: Interlocking directorates and intercorporate coordination. *Administrative Science Quarterly*, pages 40–55, 1983.

[55] J.-S. Park, M. Penner, and V. K. Prasanna. Optimizing graph algorithms for improved cache performance. *IEEE Transactions on Parallel and Distributed Systems*, 15(9):769–782, 2004.

[56] Y. Peng, Y. Zhang, W. Zhang, X. Lin, and L. Qin. Efficient probabilistic k-core computation on uncertain graphs. In *ICDE*, pages 1192–1203. IEEE, 2018.

[57] A. Pinar, C. Seshadhri, and V. Vishal. Escape: Efficiently counting all 5-vertex subgraphs. In *WWW*, pages 1431–1440. International World Wide Web Conferences Steering Committee, 2017.

[58] G. Robins and M. Alexander. Small worlds among interlocking directors: Network structure and distance in bipartite graphs. *Computational & Mathematical Organization Theory*, 10(1):69–94, 2004.

[59] S.-V. Sanei-Mehri, A. E. Sariyuce, and S. Tirthapura. Butterfly counting in bipartite networks. In *KDD*, pages 2150–2159. ACM, 2018.

[60] A. E. Sarıyüce and A. Pinar. Peeling bipartite networks for dense subgraph discovery. In *WSDM*, pages 504–512. ACM, 2018.

[61] T. Schank and D. Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *International workshop on experimental and efficient algorithms*, pages 606–609. Springer, 2005.

[62] C. Seshadhri, A. Pinar, and T. G. Kolda. Triadic measures on graphs: The power of wedge sampling. In *SDM*, pages 10–18. SIAM, 2013.

[63] J. Shun and K. Tangwongsan. Multicore triangle computations without tuning. In *ICDE*, pages 149–160. IEEE, 2015.

[64] L. D. Stefani, A. Epasto, M. Riondato, and E. Upfal. Triest: Counting local and global triangles in fully dynamic streams with fixed memory size. *TKDD*, 11(4):43, 2017.

[65] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *WWW*, pages 607–614. ACM, 2011.

[66] M. Then, M. Kaufmann, F. Chirigati, T.-A. Hoang-Vu, K. Pham, A. Kemper, T. Neumann, and H. T. Vo. The more the merrier: Efficient multi-source graph traversal. *PVLDB*, 8(4):449–460, 2014.

[67] C. Wang, J. Wang, X. Lin, W. Wang, H. Wang, H. Li, W. Tian, J. Xu, and R. Li. Mapdupreducer: detecting near duplicates over massive datasets. In *SIGMOD*, pages 1119–1122, 2010.

[68] J. Wang, A. W.-C. Fu, and J. Cheng. Rectangle counting in large bipartite graphs. In *BigData Congress*, pages 17–24. IEEE, 2014.

[69] K. Wang, X. Cao, X. Lin, W. Zhang, and L. Qin. Efficient computing of radius-bounded k-cores. In *ICDE*, pages 233–244. IEEE, 2018.

[70] X. Wang, Y. Zhang, W. Zhang, X. Lin, and W. Wang. Ap-tree: Efficiently support continuous spatial-keyword queries over stream. In *ICDE*, pages 1107–1118. IEEE, 2015.

[71] H. Wei, J. X. Yu, C. Lu, and X. Lin. Speedup graph processing by graph ordering. In *SIGMOD*, pages 1813–1828. ACM, 2016.

[72] W. Yu, X. Lin, W. Zhang, L. Chang, and J. Pei. More is simpler: Effectively and efficiently assessing node-pair similarities based on hyperlinks. *PVLDB*, 7(1):13–24, 2013.

[73] F. Zhang, L. Yuan, Y. Zhang, L. Qin, X. Lin, and A. Zhou. Discovering strong communities with user engagement and tie strength. In *DASFAA*, pages 425–441. Springer, 2018.

[74] F. Zhang, Y. Zhang, L. Qin, W. Zhang, and X. Lin. When engagement meets similarity: efficient (k, r)-core computation on social networks. *PVLDB*, 10(10):998–1009, 2017.

[75] F. Zhang, Y. Zhang, L. Qin, W. Zhang, and X. Lin. Efficiently reinforcing social networks over user engagement and tie strength. In *ICDE*, pages 557–568. IEEE, 2018.

[76] Z. Zou. Bitruss decomposition of bipartite graphs. In *DASFAA*, pages 218–233. Springer, 2016.