Fast Counting and Utilizing Induced 6-Cycles in Bipartite Networks

Jason Niu, *Graduate Student Member, IEEE*, Jaroslaw Zola, and Ahmet Erdem Sarıyüce

***Abstract*—Bipartite graphs are a powerful tool for modeling the interactions between two distinct groups. These bipartite relationships often feature small, recurring structural patterns called motifs which are building blocks for community structure. One promising structure is the induced 6-cycle which consists of three nodes on each node set forming a cycle where each node has exactly two edges. In this paper, we study the problem of counting and utilizing induced 6-cycles in large bipartite networks. We first consider two adaptations inspired by previous works for cycle counting in bipartite networks. Then, we introduce a new approach for node triplets which offer a systematic way to count the induced 6-cycles, used in BATCHTRIPLETJOIN. Our experimental evaluation shows that BATCHTRIPLETJOIN is significantly faster than the other algorithms while being scalable to large graph sizes and number of cores. On a network with $112M$ edges, BATCHTRIPLETJOIN is able to finish the computation in 78 mins by using 52 threads. In addition, we provide a new way to identify anomalous node triplets by comparing and contrasting the butterfly and induced 6-cycle counts of the nodes. We showcase several case studies on real-world networks from Amazon Kindle ratings, Steam game reviews, and Yelp ratings.**

***Index Terms*—Bipartite, cycle, induced, motif.**

I. INTRODUCTION

THE growing interest in bipartite graphs derives from the applications which model the relationships between two distinct groups [1], [2], [3], [4], [5]. In a bipartite graph, the node set is divided into two disjoint and independent sets U and V such that every edge connects a node in U to one in V. For example, recommendation networks are often represented as a bipartite graph with users as one node set and items as the other [6]. Bipartite graphs are also used to model hypergraphs where entities take part in group relations, such as actor-movie [7], author-paper [8], and company-board member [9] connections. Despite their representation power, bipartite graphs are understudied because most graph algorithms, including motif analysis, are focused on the traditional unipartite graphs. One solution is to project bipartite graphs to obtain the unipartite representation but this comes at a cost of significant information loss and inflated graph size [10], [11]. Hence, it is essential to design algorithms that directly work on the bipartite graphs.

Received 30 July 2023; revised 12 October 2024; accepted 15 February 2025. Date of publication 27 February 2025; date of current version 1 May 2025. This work was supported by NSF under Grant OAC-1845840 and Grant OAC-2107089. Recommended for acceptance by L. Chen. *(Corresponding author: Jason Niu.)*

The authors are with the University at Buffalo, Buffalo, NY 14068 USA (e-mail: jasonniu@buffalo.edu; jzola@buffalo.edu; erdem@buffalo.edu).

Digital Object Identifier 10.1109/TKDE.2025.3546516

Motif-based analysis is shown to have significant benefits for various graph mining tasks [10], [12], [13]. The smallest non-trivial motif in a bipartite graph is a butterfly ((2,2)-biclique), also known as a four-cycle and a rectangle [14], [15]. Butterflies are shown to be an effective building block for community structure and used in various graph mining tasks in bipartite graphs [16], [17], [18]. Sequential and parallel algorithms are designed for butterfly counting in offline and online scenarios [14], [15], [19], [20], [21]. However, butterflies capture the higher-order relations between only two nodes from the same node set. Alternative measures also have limited success, for example a (3,3)-biclique suffers from the prohibitive cost of computation [1] and a 3-path (i.e., butterfly minus an edge) is unable to model cohesion [9]. There is a need to go for larger bipartite motifs which can model higher-order relations while being computationally affordable.

One promising structure in this context is the *6-cycle*, proposed by Opsahl [22] to model the triadic closure in bipartite networks. A 6-cycle consists of three nodes on each node set forming a cycle. Recently, Yang et al. introduced algorithms for counting *non-induced* 6-cycles [23]. While their algorithms are efficient, they ignore the inducedness constraint which is a key to get more informative results by avoiding combinatorial explosion. Pržulj's seminal works has shown that induced motifs (named as graphlets) [24], [25] better model the local structural properties of complex networks than non-induced motifs [26]. The main reason is that the presence and the lack of each edge is fixed in a graphlet (e.g., no diagonal edges exist in a four-cycle), and hence the frequency of a graphlet is never inflated or shadowed by another graphlet with same number of nodes (e.g., number of four-clique graphlets do not inflate the number of four-cycle graphlets). In applications where the presence or absence of every edge matters, such as in biology and anomaly detection, graphlet-based analysis is preferred over motif-based approaches [25], [27], [28]. In an *induced* 6-cycle, each node has exactly two edges. There is no butterfly (or biclique) since each pair of nodes (from the same set) shares only one neighbor. An induced 6-cycle relates three nodes in the same node set to each other by forming a triangle in the projections with the minimal number of edges (see Fig. 1). In that respect, induced 6-cycles offer a more distilled perspective than butterflies or bicliques. However, counting induced 6-cycles is more challenging than non-induced 6-cycles since one has to account for the lack of certain edges to ensure inducedness.

In this work, we present parallel algorithms to count induced 6-cycles in bipartite graphs. To the best of our knowledge,

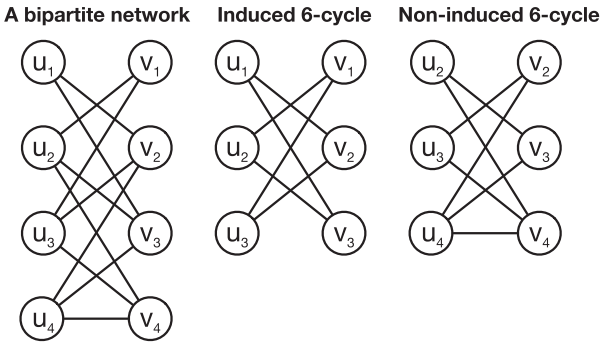

Fig. 1. A toy bipartite network G, an induced 6-cycle in G, and a non-induced 6-cycle in G. The induced 6-cycle $(u_1, u_2, u_3, v_1, v_2, v_3)$ consists of exactly six edges and the degree of each node is exactly 2. The non-induced 6-cycle $(u_2, u_3, u_4, v_2, v_3, v_4)$ has an additional edge, making the degree of two nodes three. The induced 6-cycle does not include a butterfly whereas the non-induced 6-cycle contains two butterflies: u_2, u_4, v_3, v_4 and u_3, u_4, v_2, v_4. Both the left and right projections of an induced or non-induced 6-cycle result in triangles; each node pair is related since they share a neighbor in G. However, induced 6-cycles are the smallest bipartite structure that enables such projections.

there is no prior work on counting induced 6-cycles. Due to the high computational cost, we use the affordances of shared-memory parallelization for practical runtime performance. We first consider two previous studies on cycle counting in bipartite networks and adapt them for parallel induced 6-cycle counting. In particular, we use the breadth-first search idea from [29] and wedge join technique from [20]. We show that those approaches have prohibitive time and space costs, and are therefore not scalable for large bipartite networks. As a solution, we propose counting induced 6-cycles over node triplets (three nodes on the same node set). Node triplets offer a systematic way to count induced 6-cycles in batches, thus avoiding duplicate work and enabling time-space tradeoffs for faster computation. We further consider space improvements by minimizing global storage and reduction of set intersection/difference operations when designing the BATCHTRIPLETJOIN algorithm. In all our algorithms, we expose embarrassingly parallel computations in the coarse level and also make use of a preprocessing routine to assign better workloads for the threads. Preprocessing filters out the redundant parts of the graph while keeping the induced 6-cycle count the same and performs graph reordering to increase efficiency. We perform an extensive experimental evaluation on real-world networks and investigate the runtime and memory usage performance of our algorithms along with strong and weak scalability studies. We also perform case studies to illustrate the differences between induced 6-cycles and butterflies as well as possible applications.

Our contributions can be summarized as follows:

- *Preprocessing:* We consider several techniques to filter and reorder the graph to speed up the induced 6-cycle counting. These techniques are applied to all our algorithms. Section IV introduces several graph filtering and ordering techniques to speed up the induced 6-cycle counting. We analyze the effectiveness of these techniques in Section IX-D.
- *Cycle counting algorithms:* Due to the lack of prior work on induced 6-cycle counting, we first propose two algorithms inspired by other cycle counting models in Section V. Then, we give two new approaches based on counting induced 6-cycles for node triplets which improve both the runtime and memory usage (Section VI). We provide worst-case time complexities using the work-span model for parallel computing.
- *Evaluation on real-world networks:* We compare the runtime of our algorithms on differing number of cores to demonstrate high scalability and practical runtimes for various real-world bipartite networks (Section IX). On a network with more than half a billion edges, BATCHTRIPLETJOIN our best algorithm, finishes the computation in 13.2 hours by using 52 threads.
- *Detecting anomalous node triplets:* We introduce a new algorithm, FINDTRIPLETS, which finds node triplets with high induced 6-cycle and low butterfly counts (Section VII). FINDTRIPLETS is the model behind our case studies on the `Amazon-Kindle`, `Steam-Games`, and `Yelp-Business` real-world datasets, showing examples of possible analysis and applications regarding induced 6-cycles (Section X).

Note that an earlier version of this paper has appeared in ICPP'22 [30].

Outline: We present preliminary definitions and notation in Section II and summarize the prior work on motif counting in bipartite networks in Section III. Then, we give a series of preprocessing techniques to speed up induced 6-cycle counting in Section IV and adaptations of two cycle counting algorithms for induced 6-cycle counting in Section V. Next, we present our two main algorithms based on the use of node triples in Section VI. Afterward, we show a process for finding induced 6-cycle dominant and butterfly deficient node triples in Section VII. We give our experimental evaluation in Section IX, case studies in Section X, and conclusion in Section XI.

II. PRELIMINARIES

We work on a simple and undirected bipartite graph $G = (U, V, E)$ where U is the set of nodes in the left set, V is the set of nodes in the right set, and E is the set of edges. The neighbors of a node v is denoted by $N(v)$ and its degree ($|N(v)|$) by $d(v)$. We denote $|U| + |V|$ as n (number of nodes) and $|E|$ as m (number of edges). The summation of all elements in a list X is denoted as $sum(X)$. We use the work-span model for theoretical analysis [31]. The work of an algorithm is the total number of operations and the span of an algorithm is the longest dependency path. We use hash tables that perform n operations of insertion, deletion, and membership queries in $O(n)$ work and $O(log\ n)$ span with high probability. For space complexities, we represent the number of processing units as p.

An *induced 6-cycle* is a set of six nodes $u_1, u_2, u_3 \in U$ and $v_1, v_2, v_3 \in V$ and six edges as follows (w.l.o.g):

(i) $(u_1, v_2), (u_1, v_3), (u_2, v_1), (u_2, v_3), (u_3, v_1), (u_3, v_2)$ edges exist;

(ii) $(u_1, v_1), (u_2, v_2), (u_3, v_3)$ edges do *not* exist.

If only (i) holds, it is a *non-induced 6-cycle*. In a given bipartite network G, we find the total number of instances of

induced 6-cycles. In an induced 6-cycle instance, two vertices are connected if and only if they are also connected in G. The degree of a node is exactly two in an induced 6-cycle and at least two in a non-induced 6-cycle. Fig. 1 gives an example for both.

We define a *wedge* as a 2-path composed of two *endpoint* vertices $u_1, u_2 \in U$ and a *center* vertex $v \in V$ with edges $(u_1, v), (u_2, v) \in E$ (we always consider the endpoints in the left set and the center vertex in the right set). An induced 6-cycle is made up of three wedges connected to each other in a cyclic way. We use $W(x)$ to denote the set of wedges where x is the smaller of the two endpoints (in U) and $W(x, y)$ to denote the set of wedges whose endpoints are x and y. The total number of wedges centered on V is equal to $\sum_{v \in V} \binom{d(v)}{2}$ and denoted by $|W_V|$. Likewise, $|W_U| = \sum_{u \in U} \binom{d(u)}{2}$. We denote the wedge count of a graph as $|W| = max(|W_U|, |W_V|)$.

III. Related Work

In this section, we review various related works on finding motifs in bipartite networks.

Counting Short Cycles in Bipartite Networks: A cycle in a bipartite network is considered to be short if its length k follows $g \leq k \leq 2g - 2$ where g is the length of the smallest cycle in the graph. The objective here is to count all non-induced short cycles in bipartite networks. A message-passing algorithm was proposed by Karimi and Banihashemi [32] which iteratively passes messages across a node's neighbors to count all short cycles within a bipartite graph. Dehghan and Banihashemi [29] proposed an algorithm to count short cycles by applying breadth-first search to all nodes in either the left or right set of a bipartite network.

Butterfly Counting: In this problem, the objective is to count the number of butterflies in bipartite networks. A butterfly is the smallest cycle in bipartite networks and has a variety of applications such as document clustering [4] and link spam detection [3]. The first work for butterfly counting is by Wang et al. who introduced a counting scheme which uses the number of wedges containing each node in the left set to calculate the total butterfly count [14]. Sanei-Mehri et al. improved upon Wang et al.'s algorithm by computing the number of wedges for each node in the set with lower runtime cost [15]. The set with the higher sum of squares of the degrees for each node is selected. Along with the exact counting algorithms, they also proposed randomized algorithms which can approximate the number of butterflies in bipartite networks. In another work, Sanei-Mehri et al. introduced streaming algorithms to count butterflies in graph streams [21]. Shi and Shun [20] recently designed a parallel butterfly counting algorithm which modified Chiba and Nishizeki's wedge retrieval process [19] to enable parallelization.

6-Cycle Counting: The problem of counting 6-cycles in bipartite networks has only recently been studied for large bipartite networks. Yang et al. introduced algorithms to count the number of non-induced 6-cycles, which they denote as bi-triangles [23]. Their algorithms are based on combining wedges and super-wedges, with the former being 2-paths and the latter being 3-paths. They also introduce local 6-cycle counting algorithms which count the number of 6-cycles containing a specified node or edge. In our work, we consider *induced 6-cycle counting*, which is more challenging and promising for real-world applications.

Algorithm 1: PREPROCESSING (G).

Input: $G = (U, V, E)$: graph;
Output: $G' = (U', V', E')$: processed graph ;
1: $G \leftarrow$ 2-core of G ;
2: **if** $|U| > |V|$ **then** $Swap(U, V)$ // Ensure $|U| < |V|$
// Sort the nodes in U by inc. count of wedges
3: $X \leftarrow SortbyWedgeCounts(U)$;
4: Let x's rank $R[x]$ be its index in X ;
5: **parallel foreach** $u \in U$ **do** add $R[u]$ to U'
6: $V' \leftarrow V$;
// $N'(x)$ is the neighbors of node x in G'
// In both loops, neighbors sorted in descending order
7: **parallel foreach** $u \in U$ **do** $N'(R[u]) \leftarrow Sort(\{v | (u, v) \in E\})$
8: **parallel foreach** $v \in V$ **do** $N'(v) \leftarrow Sort(\{R[u] | (u, v) \in E\})$
9: **return** G'

IV. Preprocessing

We make use of a generic preprocessing step in all our algorithms which formats the graph to speed up computations. To speed up the computation for large bipartite graphs, we can shrink and reformat the graph such that the induced 6-cycle count stays the same. In PREPROCESSING, outlined in Algorithm 1, we give a computation that takes as input a bipartite graph and outputs another bipartite graph that filters out some parts of the input and reorders the nodes and neighbor lists. We first update the input graph to only consider the nodes and edges that are in a 2-core, which is a maximal connected subgraph in which all nodes have a degree of at least 2 (line 1). Since all the nodes in an induced 6-cycle have a degree of at least 2, we can simply ignore the nodes outside the 2-core, thus reducing the size of the graph. Afterwards, if necessary, we swap the left (U) and right sets (V) to ensure that the left set (U) has the smaller number of nodes (line 2). We always parallelize based on U in our counting algorithms, hence making it the smaller set increases the number of induced 6-cycles that are processed in batches for each thread. Next, we reorder each node $u \in U$ in increasing order of wedges from u (lines 3 - 5). The wedge count for a node u is $\sum_{v \in N(u)} d(v) - 1$ (we consider the wedges where u is an end-point, as defined in Section II). Note that Shi and Shun [20] showed that reordering the graph using approximate degree ordering or degeneracy ordering yields efficient results and here we consider wedge count based ordering in a similar spirit. Finally, we sort each neighbor list in descending order of node ids (lines 7 - 8). This enables linear time set intersection and

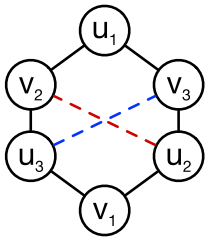

Fig. 2. NODEJOIN's BFS tree. The dotted lines represent the edges which do not exist. The BFS tree goes from top to bottom with u_1 being the root node and node v_1 at depth level three. We check for the lack of the blue edge in line 7 and of the red edge in line 9 in Algorithm 2.

difference operations. We evaluate the impact of our techniques in PREPROCESSING as well as various node reordering schemes in Section IX-D.

Work, span, and space complexity: In PREPROCESSING, core decomposition (line 1) takes $O(m)$ time [33]. The swapping of left and right sets (line 2) is $O(1)$ if pointers are swapped instead of the contents themselves. Finally, reordering the nodes and sorting neighbor lists in descending order (lines 3-8) takes $O(|X| \log |X| + m \log m)$ time where $|X| = min(|U|, |V|)$. Overall, PREPROCESSING takes $O(m\ log\ m)$ work. Since core decomposition is performed sequentially and sorting neighbor lists is done in parallel, the span is $O(m + n\ log\ n)$. Asymptotically, it never becomes a bottleneck in any of our counting algorithms. The space complexity for storing the processed graph and temporary variables is $O(m)$.

V. ADAPTING CYCLE COUNTING

Since induced 6-cycle counting has not been studied before, we start by proposing two adaptations inspired by the previous works for cycle counting in bipartite networks. The first is a modified version of breadth-first search to count induced 6-cycles, which Dehghan and Banihashemi also used to count short cycles [29] (Section V-A). The second is based on the parallel wedge retrieval algorithm proposed by Shi and Shun, which was used for butterfly counting [20] (Section V-B).

A. Counting by Breadth-First Search

One of the more common methods for finding cycles in a graph is through breadth-first search (BFS) [29]. The idea is to simply perform a traversal for a few levels and determine the number of cycles that the root node takes part in. To count induced 6-cycles, we introduce the NODEJOIN algorithm, outlined in Algorithm 2. Given a bipartite graph $G = (U, V, E)$, NODEJOIN counts the induced 6-cycles by performing a limited BFS from each vertex $u \in U$ up until a depth level of three. Fig. 2 illustrates the BFS tree where $u_1 \in U$ is the root node. $v_2, v_3 \in V$ are two of u_1's neighbors, hence put at level one. In level two, we find a neighbor of v_2 which is not connected to v_3, denoted by u_3 (and vice versa, denoted by u_2). In the last level, we find a common neighbor of u_2 and u_3, denoted by v_1, which is not connected to u_1.

For each root node $u_1 \in U$, the same u_2, u_3 pair may appear in multiple induced 6-cycles containing u_1. To avoid duplicate processing, we use the container S (line 4 in Algorithm 2) to store the number of nodes in $N(u_2) \cap N(u_3) \backslash N(u_1)$, which corresponds to v_1s in the last level of the BFS tree (line 12). We also ensure an ordering such that $u_3 > u_1$ and $u_2 > u_1$ (lines 7 and 9) to break the symmetry and thus prevent the duplicate processing of node pairs from U. Note that the *counts* list contains the number of induced 6-cycles counted *through* each vertex; it is not the actual count for each vertex. The sum of *counts* gives the total induced 6-cycle count (line 14). NODEJOIN has a coarse-grained parallelism where the root nodes in U are shared among the threads (line 3).

Algorithm 2: NODEJOIN (G).

```
   Input: G (U, V, E): graph;
   Output: count: number of induced 6-cycles;
1: G ← PREPROCESSING (G);
2: counts ← []                                  // |U| values
3: parallel foreach u1 ∈ U do
4:   S ← ∅ // Hashmap of node pairs (from
       U) to values
5:   foreach v2, v3 ∈ N(u1) s.t. v2 > v3 do
6:     H ← ∅                                    // Set of nodes
7:     foreach u3 ∈ N(v2)\N(v3) and u3 > u1 do
8:       add u3 to H
9:     foreach u2 ∈ N(v3)\N(v2) and u2 > u1 do
10:      foreach u3 ∈ H do
           // S stores the number of v1s
             (see Fig. 2)
11:        if (u2, u3) ∉ S then
12:          S[(u2, u3)] ← |N(u2) ∩ N(u3)\N(u1)|
13:        counts[u1] ← counts[u1] + S[(u2, u3)]
14: count ← sum(counts)                         // Parallel
    reduction
15: return count;
```

A significant drawback of NODEJOIN is the recomputation of set intersections across BFS trees. Multiple root nodes u_1 may participate in an induced 6-cycle with the same u_2 and u_3 node pair, resulting in the recomputation of $N(u_2) \cap N(u_3)$ (line 12). One solution would be to store each set intersection for all pairs of root nodes, but it will have prohibitive space usage for large networks.

Work and span: For each node $u \in U$, lines 3 and 5 iterate over all wedges where u is the center, corresponding to $O(|W_U|)$ iterations. The cost of lines 7 and 8 is $O(|U|)$. Lines 9 and 10 take $O(|U|)$ iterations each, for a total of $O(|U|^2)$ iterations. Computing the set operations in line 12 takes $O(|V|)$ time because $N(u_2) \cap N(u_3) \backslash N(u_1)$ can be computed in linear-time by simultaneously going over the neighbor lists of u_2, u_3, and u_1 (neighbor lists are kept sorted in descending order, see Section IV). Overall, NODEJOIN can be performed in $O(|W_U| \cdot (|U| + |U|^2 \cdot |V|))$ work, which is equal to $\boldsymbol{O(|W| \cdot |U|^2 \cdot |V|)}$. Regarding the span, only the outer loop on line 3 is parallelized, resulting in a span of $O(|W| \cdot |U| \cdot |V|)$.

Space complexity: In addition to the $O(m)$ space taken by the graph, NODEJOIN uses one global container *counts* (line 2) and two local containers S (line 4) and H (line 6) per thread

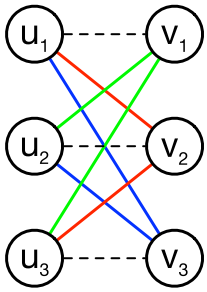

Fig. 3. A cycle of three wedges (red, blue, and green) and the lack of any other edge are needed to form an induced 6-cycle. The dashed edges must be nonexistent; including any of those would make the 6-cycle non-induced.

to store various auxiliary information. $counts$ stores $|U|$ values. S stores $O(|U|^2)$ values. H stores up to $|U|$ nodes. Therefore, the space complexity of NODEJOIN is $O(m + |U| + p \cdot (|U|^2 + |U|) = O(p \cdot |U|^2)$. Note that in practice we observe that this is a loose bound and the actual memory footprint is much smaller (see Section IX-C).

B. Counting by Wedges

An alternative way to count induced 6-cycles is by aggregating wedges. Since induced 6-cycles are composed of three overlapping wedges (see Fig. 3), we can reduce the cost of computation by operating on wedges rather than nodes. Shi and Shun proposed to use (and store) wedges for counting butterflies [20]. We can count induced 6-cycles using a similar wedge retrieval technique while taking advantage of the patterns associated with inducedness.

We describe our wedge based counting algorithm WEDGEJOIN in Algorithm 3. WEDGEJOIN simply goes over triples of wedges and counts the ones that form an induced 6-cycle. Wedge retrieval (lines 2-6) is based off of Shi and Shun's [20] algorithm and enables the parallel processing of wedges. The parallel container W allows for fast access of wedges based on endpoints. Unlike their algorithm, we only find wedges with endpoints in U instead of the entire node set. In our implementation, W is a list of all the wedges in the graph such that each wedge consists of two nodes from U (endpoints) and one node from V (center). We partition W based on the smaller endpoint (u_1) and sort each partition with respect to the larger endpoint (u_2). For all nodes $u_1 \in U$, W enables the retrieval of all wedges with endpoints u_1, u_2 and center v_3 such that $u_1 < u_2$.

WEDGEJOIN finds cycles of wedges (u_1, v_3, u_2), (u_2, v_1, u_3), and (u_1, v_2, u_3) such that $u_1 < u_2 < u_3$. Line 8 (in Algorithm 3) iterates over all blue wedges (u_1, v_3, u_2) and line 10 iterates over all green wedges (u_2, v_1, u_3) (Fig. 3). The lack of edges (u_1, v_1), (u_2, v_2), and (u_3, v_3) is needed to satisfy the inducedness (the dashed black edges in Fig. 3). To ensure that the blue and green wedge pair satisfy the inducedness constraint, we first check for the nonexistence of (u_1, v_1) and (u_3, v_3) in line 13. Afterwards, we traverse all red wedges (u_1, v_2, u_3) (Fig. 3) in line 15. Finally, we check for the last unwanted edge (u_2, v_2) in line 17 and increment the induced 6-cycle count of the blue wedge.

We have two speedups for faster computation, mentioned in lines 12 and 14. In the first speedup, we aim to skip the processing

Algorithm 3: WEDGEJOIN (G).

Input: $G\ (U, V, E)$: graph;
Output: $count$: number of induced 6-cycles;
1: $G \leftarrow$ PREPROCESSING (G);
2: $W \leftarrow \emptyset$ `// Parallel container of wedges`
3: **parallel foreach** $u_1 \in U$ **do**
4: **foreach** $v_3 \in N(u_1)$ **do**
5: **foreach** $u_2 \in N(v_3)$ s.t. $u_2 > u_1$ **do**
6: add (u_1, v_3, u_2) to W (sorted by endpoints)
7: $counts \leftarrow []$ `// |W| values`
8: **parallel foreach** $w_1 \in W$ **do**
9: $(u_1, v_3, u_2) \leftarrow w_1$ `// Blue wedge in Fig. 3; u1 < u2`
10: **foreach** $w_2 \in W(u_2)$ **do**
11: $(u_2, v_1, u_3) \leftarrow w_2$ `// Green wedge in Fig. 3; u2 < u3`
12: `// Speedup #1: If v3 ∈ N(u3), skip all the successive wedges with the same endpoints; also no need to check v3 ∉ N(u3) for such wedges`
13: **if** $v_1 \notin N(u_1)$ *and* $v_3 \notin N(u_3)$ **then**
14: `// Speedup #2: Reuse the count below for all the successive green wedges with the same pair of endpoints`
15: **foreach** $w_3 \in W(u_1, u_3)$ **do**
16: $(u_1, v_2, u_3) \leftarrow w_3$ `// Red wedge in Fig. 3`
17: **if** $v_2 \notin N(u_2)$ **then** $counts[w_1] + +$
18: $count \leftarrow sum(counts)$ `// Parallel reduction`
19: **return** $count$

of some green wedges with particular endpoints. Note that in W, the wedges with the same smaller-endpoint (u_1) are sorted with respect to their larger-endpoint (u_2). This means that while going over the green wedges (w_2) in line 10, where u_2 is the smaller-endpoint, we may encounter successive green wedges with the same pair of endpoints, u_2 and u_3 (where the center point (v_1) from V is different). In that case, if $v_3 \in N(u_3)$ happens to be true, we can skip processing all such successive green wedges with endpoints u_2, u_3 because the (u_3, v_3) edge violates the inducedness condition. We can do this by simply keeping a flag and temporary variable to remember the larger-endpoint (u_3) from the last processed green wedge. This way we do not check whether $v_3 \notin N(u_3)$ again and again. More importantly, if $v_3 \in N(u_3)$, we skip processing all the green wedges with the same pair of endpoints u_2, u_3. In the second speedup, we again take advantage of the successive green wedges with the same pair of endpoints. We perform the computation in lines 15 to 17 once for a w_1, w_2 pair and reuse the induced 6-cycle count for all successive w_1, w_2' pairs where w_2 and w_2' share the same endpoints. We use both speedups in our implementation.

WEDGEJOIN computes the true count of induced 6-cycles by finding three wedges where: (1) *Nodes on the left are unique:*

Lines 9 and 11 enforce uniqueness by establishing an ordering $u_1 < u_2 < u_3$; (2) *Nodes on the right are unique:* Each node on right is the center of a traversed wedge (lines 9, 11, and 16) which contain two endpoints—through the inducedness checks in lines 13 and 17, we prove uniqueness by checking if a pair of endpoints from all three traversed wedges connects to multiple nodes on right; (3) *The six induced edges exist* (the blue, green, and red edges in Fig. 3): Since all nodes are unique, each traversed wedge (lines 8, 10, and 15) contains two of the induced edges; (4) *The three non-induced edges do not exist* (the dashed black edges in Fig. 3): We have explicit conditions on lines 13 and 17 corresponding to the three inducedness checks. Finally, since we go over all triples of wedges, all the induced 6-cycles are counted.

Work and span: Wedge retrieval (lines 3-6) iterates over all wedges where a $v \in V$ is a center, corresponding to $O(|W_V|)$ iterations. Starting in line 8, we iterate over all the wedges, taking $|W_V|$ iterations. The loop on line 10 finds the wedges where a node $u \in U$ is the smaller endpoint, which is $O(|W_V|)$, as it is upper bounded by the number all wedges with a center point in V. We find the third wedge in line 15, which takes $O(|V|)$ iterations. Line 17 simply takes $O(1)$ time. Overall, WEDGEJOIN can be performed in $O(|W_V| + |W_V|^2 \cdot |V|)$ work, which is equal to $\boldsymbol{O(|W|^2 \cdot |V|)}$. Compared with NODEJOIN, which has $O(|W| \cdot |U|^2 \cdot |V|)$ work, WEDGEJOIN is expected to be faster as $|W|$ is often way smaller than $|U|^2$ in real-world networks. Regarding the span, as in NODEJOIN, only the outer loops are parallelized, resulting in a span of $O(|W| \cdot |V|)$.

Space complexity: The global containers *counts* (line 7) and W (line 2) each takes $O(|W_V|)$ space. Overall space complexity is $O(m + |W|)$.

VI. Node Triplets for Faster Counting

In this section, we propose a new technique that considers node triplets to count the induced 6-cycles. We define a node triplet to be a grouping of three unique nodes such that all nodes are in the same set (U or V) and there exists a 4-path connecting the three nodes. Inspired by Yang et al.'s approach for non-induced 6-cycles [23], we derive a formula to find the number of induced 6-cycles for a given node triplet and compute the total count by going over all node triplets. The formula lets us systematically avoid the duplicate work and engage in time-space tradeoffs for faster computation. We first introduce the TRIPLETJOIN algorithm in Section VI-A which simply applies the formula for all node triplets and also stores the set of common neighbors for fast computation. Then, we present our final algorithm, BATCHTRIPLETJOIN, in Section VI-B which improves TRIPLETJOIN by storing common neighbors more efficiently and reducing set operations.

A. Counting by Node Triplets

Node triplets offer a systematic way to count the induced 6-cycles. Given that there are exactly six edges in an induced 6-cycle and no two nodes share more than one neighbor, we can derive a formula to find the number of induced 6-cycles for a given node triplet:

Algorithm 4: TRIPLETJOIN (G).

Input: G (U, V, E): graph;
Output: *count*: number of induced 6-cycles;
1: $G \leftarrow$ PREPROCESSING (G);
2: $counts \leftarrow []$ `// |U| values`
`// For each node pair in U, common neighbors stored in S`
3: $S \leftarrow \emptyset * |U|$ `// |U| hashmaps of nodes to sets`
4: **parallel foreach** $u_1 \in U$ **do**
5: **foreach** $v_j \in N(u_1)$ **do**
6: **foreach** $u_i \in N(v_j)$ s.t. $u_i > u_1$ **do**
7: add v_j to $S[u_1][u_i]$
8: **parallel foreach** $u_1 \in U$ **do**
9: $H \leftarrow \emptyset$ `// Distance-2 neighbors of` u_1 `with greater id`
10: **foreach** $v_j \in N(u_1)$ **do**
11: **foreach** $u_i \in N(v_j)$ s.t. $u_i > u_1$ **do**
12: add u_i to H
13: **foreach** $u_2, u_3 \in H$ s.t. $u_3 > u_2$ **do**
14: **if** $u_3 \in S[u_2].keySet()$ **then**
15: $counts[u_1] \leftarrow counts[u_1] + (|S[u_1][u_2] \backslash N(u_3)|) \cdot (|S[u_1][u_3] \backslash N(u_2)|) \cdot (|S[u_2][u_3] \backslash N(u_1)|)$
16: $count \leftarrow sum(counts)$ `// Parallel reduction`
17: **return** *count*

Theorem 1: Given a bipartite network $G = (U, V, E)$ and three unique nodes u_1, u_2, and $u_3 \in U$, the number of induced 6-cycles containing the node triplet (u_1, u_2, u_3) is:

$$|N(u_1) \cap N(u_2) \backslash N(u_3)| \cdot |N(u_1) \cap N(u_3) \backslash N(u_2)| \cdot |N(u_2) \cap N(u_3) \backslash N(u_1)| \quad (1)$$

Proof: Let unique nodes v_1, v_2, and $v_3 \in V$ be in an induced 6-cycle with u_1, u_2, and u_3 as depicted in the induced 6-cycle of Fig. 3. The difference between a 6-cycle and an induced 6-cycle is that, in an induced 6-cycle, neither of v_1, v_2, and v_3 can be a common neighbor of all three nodes u_1, u_2, and u_3. Therefore, the number of induced 6-cycles containing u_1, u_2, u_3, v_1, and v_3 is the number of possible v_2s which are neighbors of u_1 and u_3 but not u_2. This can be represented as $|N(u_1) \cap N(u_3) \backslash N(u_2)|$. Likewise, the number of possible v_1s and v_3s are $|N(u_2) \cap N(u_3) \backslash N(u_1)|$ and $|N(u_1) \cap N(u_2) \backslash N(u_3)|$, respectively. The sets of $N(u_1) \cap N(u_2) \backslash N(u_3)$, $N(u_1) \cap N(u_3) \backslash N(u_2)$, and $N(u_2) \cap N(u_3) \backslash N(u_1)$ are mutually exclusive. Therefore, multiplying the size of these three sets gives the number of induced 6-cycles for the node triplet u_1, u_2, u_3.□

Algorithm 4 outlines the TRIPLETJOIN algorithm. Given a bipartite graph $G = (U, V, E)$, TRIPLETJOIN computes the number of participating induced 6-cycles for all node triplets of U and returns the total sum. TRIPLETJOIN processes at most $\binom{|U|}{3}$ node triplets, of which many may share multiple nodes, such as the same pair of nodes $u_1, u_2 \in U$. This may cause serious recomputation of $N(u_1) \cap N(u_2)$, which corresponds

to a significant runtime cost. Therefore, we store the common neighbors of node pairs in U, i.e., $N(u_1) \cap N(u_2) \, \forall \, u_1, u_2 \in U$, in container S (lines 3-7). Then, for each u_1, we use a container H (line 9) to store all its distance-2 neighbors which are greater than itself. Afterwards, we obtain node triplets by iterating over unique node pairs in H and compute the induced 6-cycle count of each by Theorem 1 (line 15). This process of finding node triplets avoids going over all $\binom{|U|}{3}$ triplets by only processing the three nodes which form a 4-path (lines 10- 12: u_1-v_x-u_2 and u_1-v_y-u_3 for arbitrary v_x and v_y). Such node triplets are more likely to be a part of an induced 6-cycle when compared to an arbitrary node triplet in U.

Work and span: Lines 4-7 iterates over all wedges where a $v \in V$ is a center, corresponding to $O(|W_V|)$ iterations. Then, for each node in U (line 8), we find its distance-2 neighbors (lines 9-12). In total, lines 8 and 9-12 takes at most $O(|W_V|)$ work due to each wedge with center $v \in V$ contributing to a unique distance-2 neighbor. Line 13 traverses through pairs of distance-2 neighbors in H, which takes $O(\binom{|U|}{2})$ work. Line 15 does a computation based on Theorem 1, which takes $O(|V|)$ time. In total, lines 8 and 13-15 is performed in $O(|U|^3 \cdot |V|)$ work. Therefore, TRIPLETJOIN can be performed in $O(|W_V| + |W_V| + |U|^3 \cdot |V|)$, which is equal to $\boldsymbol{O(|U|^3 \cdot |V|)}$ work. Note that this is a loose bound, especially due to the analysis of lines 13-15. Regardless, this is asymptotically better than NODEJOIN, which takes $O(|W| \cdot |U|^2 \cdot |V|)$ work. As in NODEJOIN and WEDGEJOIN, only the outer-most loops are parallelized, resulting to a span of $O(|U|^2 \cdot |V|)$.

Space complexity: In addition to the $O(m)$ space required for the graph and the $O(|U|)$ space required for the container $counts$ (line 2), TRIPLETJOIN involves storing wedges (line 3) in the global scope, which takes $O(|W_V|)$ space. Local storage of distance-2 neighbors of a node $u \in U$ (line 9) only takes $O(p \cdot |U|)$ space. Overall, the space complexity is $O(m + |U| + |W| + p \cdot |U|) = O(|W|)$.

B. Faster Triplet Counting With Less Space

Here we consider three orthogonal improvements on top of TRIPLETJOIN for a more time and space efficient algorithm.

Storing size of intersections: By globally storing wedges in WEDGEJOIN and set intersections in TRIPLETJOIN, we are able to solve the recomputation issue of wedges and set intersections, respectively. However, for large graphs, the space required for this storage is prohibitive and may exceed the amount of available memory. To reduce the memory usage, we can make a more efficient use of global storage across loop iterations and local storage within loop iterations. Since local storage is temporary, its memory is freed (and thus can be reallocated) after each iteration, unlike global storage. Compared to TRIPLETJOIN, which stores the set intersections in global storage, we can only store the *sizes* of set intersections globally (not the sets) and use local storage only for the set intersections which directly relate to the associated loop iteration.

Reduced set operations: Another improvement is about the computation of induced 6-cycle counts for a node triple. In an induced 6-cycle, each node $v \in V$ has exactly two edges. We can use this to improve upon Theorem 1 by eliminating the need for the set difference operation, reducing the number of computations. As shown in Theorem 2, we can instead subtract the number of nodes which are connected to all three nodes in a node triplet. Therefore, instead of computing three $O(|V|)$ set difference computations, we can simply compute one $O(|V|)$ set intersection computation.

Algorithm 5: BATCHTRIPLETJOIN (G).

Input: G (U, V, E): graph;
Output: $count$: number of induced 6-cycles;
1: $G \leftarrow$ PREPROCESSING (G);
2: $counts \leftarrow []$ // $|U|$ values
// For each node pair in U, # of common neighbors stored in S
3: $S \leftarrow \emptyset * |U|$ // $|U|$ hashmaps of nodes to values
4: **parallel foreach** $u_1 \in U$ **do**
5: **foreach** $v_j \in N(u_1)$ **do**
6: **foreach** $u_i \in N(v_j)$ s.t. $u_i > u_1$ **do**
7: $S[u_1][u_i] \leftarrow S[u_1][u_i] + 1$
8: **parallel foreach** $u_1 \in U$ **do**
9: $H \leftarrow \emptyset$ // Hashmap of nodes to node sets
10: **foreach** $v_j \in N(u_1)$ **do**
11: **foreach** $u_i \in N(v)$ s.t. $u_i > u_1$ **do**
12: add v_j to $H[u_i]$
13: **foreach** $u_2, u_3 \in H.keySet()$ s.t. $u_3 > u_2$ **do**
14: **if** $u_3 \in S[u_2].keySet()$ **then**
15: $x \leftarrow |H[u_2] \cap H[u_3]|$;
16: $counts[u_1] \leftarrow counts[u_1] +$
$(|H[u_2]| - x) \cdot (|H[u_3]| - x) \cdot (S[u_2][u_3] - x)$
17: $count \leftarrow sum(counts)$ // Parallel reduction
18: **return** $count$

Theorem 2: Given a bipartite network $G = (U, V, E)$, three unique nodes $u_1, u_2, u_3 \in U$, and $x = |N(u_1) \cap N(u_2) \cap N(u_3)|$, the number of induced 6-cycles containing the node triplet is:

$$(|N(u_1) \cap N(u_2)| - x) \cdot (|N(u_1) \cap N(u_3)| - x) \cdot (|N(u_2) \cap N(u_3)| - x) \quad (2)$$

Proof: Given a set X, $|X| - |X \cap A| = |X \backslash A|$. Therefore, $|N(u_1) \cap N(u_2)| - x$ is equivalent to $|N(u_1) \cap N(u_2) \backslash N(u_3)|$ in Theorem 1. As such, Theorem 2 is correct by the same reasoning as Theorem 1. □

We consider the improvements above in BATCHTRIPLETJOIN, outlined in Algorithm 5, which reduces the number of computations and is more efficient than TRIPLETJOIN in terms of memory usage. In lines 4-7, we compute and globally store the sizes of the set intersections between the neighbor lists for all u_1, u_i node pairs. Afterwards, we iterate through all u_1s (line 8) and store the non-empty set intersections between the neighbor lists of u_1 and all $u_i \in U$ s.t. $u_i > u_1$ (lines 9-12). Finally, we count the number of induced 6-cycles associated with each node

triplet $\{u_1, u_2, u_3\}$ by Theorem 2 (line 16) and return the sum of all the counts. In our implementation, we force the compiler to vectorize the inner loops and it gave a slight improvement on the largest networks.

Work and span: BATCHTRIPLETJOIN features three orthogonal improvements over TRIPLETJOIN but the time complexity does not change. Efficient usage of memory and reduction of set operations (line 16) only offer constant time speedup and thus does not affect the overall time complexity. Overall, the work and span of BATCHTRIPLETJOIN is equal to TRIPLETJOIN, which is $\boldsymbol{O(|U|^3 \cdot |V|)}$ work and $O(|U|^2 \cdot |V|)$ span.

Space complexity: BATCHTRIPLETJOIN utilizes three global storage containers - one for storing the graph, one for the container $counts$ (line 2), and one for the container S (line 3) - and one local container H (line 9). Storing the graph requires $O(m)$ space and $counts$ uses $O(|U|)$ space. S stores the size of the non-empty intersections of the neighbor lists of node pairs in U, which is equal to the number of wedges centered in V, namely $O(|W_V|)$. The local container H stores edges and thus takes $O(p \cdot m)$ space. In total, the space complexity of BATCHTRIPLETJOIN is $O(m + |U| + |W_V| + p \cdot m)$=$O(|W|)$.

VII. FINDING ANOMALOUS TRIPLES

Here, we propose a way to do anomalous detection by using butterfly and induced 6-cycle counts, which we use in our case studies in Section X.

Triples of nodes (in the same set) which are in many butterflies form a clique-like structure. These node triplets may also be in many induced 6-cycles due to their high degrees. However, it would be interesting to find node triplets which are in many induced 6-cycles but few butterflies. This causes a positive relationship between pairs of nodes and a negative relationship between triplets of nodes from the same set.

Algorithm 6, FINDTRIPLETS, shows the pseudocode for finding node triplets whose butterfly count is upper bounded and induced 6-cycle is lower bounded by parameters ub and lb, respectively. Lines 2-9 performs butterfly counting for node pairs and stores the number of common neighbors in a hashmap S if its butterfly count is upper bounded by ub. The butterfly count for each node pair $\{u_1, u_2\}$ on line 8 is obtained by $\binom{(N(u_1) \cap N(u_2))}{2}$. Then, lines 11-15 iterate through node triplets $\{u_1, u_2, u_3\}$ where $\{u_1, u_2\}$, $\{u_1, u_3\}$, and $\{u_2, u_3\}$ are all in S and computes its induced 6-cycle count. We use Theorem 2 to compute the induced 6-cycle count for each node triplet and store each triplet in a hashmap T if its induced 6-cycle count is lower bounded by lb. Finally, lines 17-19 formats hashmap T into a list of node triplets which the algorithm returns.

Work and span: Butterfly counting on lines 2-9 iterates over all wedges where a $v \in V$ is a center, taking $O(|W_V|)$ work and $O(|V| \cdot |U|)$ span. Then, similar to BATCHTRIPLETJOIN, lines 11-15 computes the induced 6-cycle count for node triplets with $O(|U|^3 \cdot |V|)$ work. Since we parallelize based on node pairs, the span for induced 6-cycle counting becomes $O(|U| \cdot |V|)$. Finally, lines 17-19 takes $O(|U|^3)$ work and $O(|U|^3)$ span to convert a hashmap into a set of node triplets. Overall, the work and span of FINDTRIPLETS is $O(|W_V| + |U|^3 \cdot |V| + |U|^3)$=$O(|U|^3 \cdot |V|)$ and $O(|V| \cdot |U| + |U| \cdot |V| + |U|^3)$=$O(|U|^3)$, respectively.

Space complexity: FINDTRIPLETS uses global containers S (line 1), T (line 10), and $triplets$ (line 16) with a local container H (line 3). S takes $O(|U|^2 \cdot |V|)$ space by storing intersections of node pairs in U, T and $triplets$ both take $O(|U|^3))$ space for storing node triplets, and H uses $O(p \cdot (|U| \cdot |V|))$ space. Therefore, the space complexity of FINDTRIPLETS is $O(|U|^2 \cdot |V| + 2 \cdot |U|^3 + p \cdot (|U| \cdot |V|))$=$O(n^3)$.

TABLE I
STATISTICS OF THE REAL-WORLD BIPARTITE NETWORKS USED IN THE EXPERIMENTS (SECTION VIII)

Networks	$\|U\|$	$\|V\|$	m	$I6C$
DBLP (DB)	4,000,150	1,425,813	10,002,631	5.10 x 10^7
Github (GI)	56,555	123,345	440,237	1.37 x 10^{11}
IMDB (IM)	1,232,031	419,661	5,596,667	2.01 x 10^{10}
Kindle (KI)	1,406,890	430,530	3,205,467	5.20 x 10^9
Twitter (TW)	175,214	530,418	1,890,661	5.58 x 10^{11}
Movielens (ML)	69,878	10,677	10,000,054	1.69 x 10^{17}
Reuters (RE)	781,265	283,911	60,569,726	9.91 x 10^{18}
LiveJournal (LJ)	3,201,203	7,489,073	112,307,385	2.10 x 10^{18}

I6C stands for number of induced 6-cycles and the rest are defined in Section II.

Algorithm 6: FINDTRIPLETS (G, ub, lb).

Input: G (U, V, E): graph, ub: upper bound (butterflies), lb: lower bound (induced 6-cycles);
Output: $triplets$: set of node triplets;

1: $S \leftarrow \emptyset * |U|$ // $|U|$ hashmaps of nodes to sets
2: **parallel foreach** $u_1 \in U$ **do**
3: $H \leftarrow \emptyset$ // Hashmap of nodes to sets
4: **foreach** $v \in N(u_1)$ **do**
5: **foreach** $u_2 \in N(v)$ s.t. $u_2 > u_1$ **do**
6: add v to $H[u_2]$
7: **foreach** $u_2 \in H.keySet()$ **do**
8: $x \leftarrow \frac{|H[u_2]| \cdot (|H[u_2]|-1)}{2}$;
9: **if** $x \leq ub$ **then** $S[u_1][u_2] = H[u_2]$
10: $T \leftarrow \emptyset$ // Hashmap of node pairs to sets
11: **parallel foreach** u_1, u_2 s.t. $u_2 \in S[u_1].keySet()$ **do**
12: **foreach** $u_3 \in S[u_1].keySet() \cap S[u_2].keySet()$ **do**
13: $x \leftarrow |S[u_1][u_2] \cap S[u_1][u_3]|$;
14: $y \leftarrow (|S[u_1][u_2]| - x) \cdot (|S[u_1][u_3]| - x) \cdot (|S[u_2][u_3]| - x)$;
15: **if** $y \geq lb$ **then** add u_3 to $T[u_1, u_2]$
16: $triplets \leftarrow \emptyset$ // Set of node triplets
17: **foreach** $u_1, u_2 \in T.keySet()$ **do**
18: **foreach** $u_3 \in T[u_1, u_2]$ **do**
19: add $\{u_1, u_2, u_3\}$ to $triplets$
20: **return** $triplets$

VIII. DATASETS

Here we introduce our real-world datasets from Konect [34] and ICON [35]. Table I gives broad statistics of the datasets. We

TABLE II
STATISTICS OF THE NETWORKS AFTER PRE

Net.	$\|U\|$	$\|V\|$	m	$\|W\|$	NJ	WJ	TJ
DB	644K	1.92M	5.89M	175M	1.39 x 10^{26}	5.88 x 10^{22}	5.13 x 10^{23}
GI	22.9K	34.3K	335K	36.3M	6.53 x 10^{20}	4.52 x 10^{19}	4.12 x 10^{17}
IM	350K	497K	4.80M	126M	7.67 x 10^{24}	7.89 x 10^{21}	2.13 x 10^{22}
KI	198K	370K	2.01M	313M	4.54 x 10^{24}	3.63 x 10^{22}	2.87 x 10^{21}
TW	129K	138K	1.46M	971M	2.23 x 10^{24}	1.3 x 10^{23}	2.96 x 10^{20}
ML	10.6K	69.9K	10.0M	37.7B	2.96 x 10^{23}	9.94 x 10^{25}	8.32 x 10^{16}
RE	169K	781K	60.5M	1.54T	3.44 x 10^{28}	1.85 x 10^{30}	3.77 x 10^{21}
LJ	2.19M	2.98M	107M	2.70T	3.86 x 10^{31}	2.17 x 10^{31}	3.13 x 10^{25}

We give the numerical work values for NJ ($O(|W| \cdot |U|^2 \cdot |V|)$), WJ ($O(|W|^2 \cdot |V|)$), and TJ ($O(|U|^3 \cdot |V|)$). Note that BTJ has the same work as TJ. All notations are denoted in Sections II and VIII.

TABLE III
RUNTIME (IN SECONDS) WHEN USING 1 AND 52 THREADS

	1 thread				52 threads				
Net.	NJ	WJ	TJ	BTJ	NJ	WJ	TJ	BTJ	*RS*
DB	15.5	8.87	7.78	4.88	3.00	2.30	2.72	1.65	1.39
GI	2,818	1,989	1,976	421	78.6	52.3	58.9	9.75	5.37
KI	1,167	782	795	224	31.9	21.0	41.2	6.30	3.34
IM	5,138	2,913	2,027	689	144	77.1	75.4	18.7	4.04
TW	1,615	524	226	81.7	43.6	13.7	9.82	2.47	3.98
ML	–	–	–	43,850	–	–	4,991	890	5.61
RE	–	–	–	85,229	–	–	12,822	1,890	6.78
LJ	–	–	–	–	–	–	30,035	4,682	6.42

"–" denotes >24 hours. *RS* denotes the relative speedup of BTJ over the second best algorithm on 52 threads. All other notations are defined in Sections II and VIII.

also give the statistics of our real-world datasets after PREPROCESSING and the computed work complexities of our algorithms in Table II. For brevity, we use PRE (PREPROCESSING), NJ (NODEJOIN), WJ (WEDGEJOIN), TJ (TRIPLETJOIN), and BTJ (BATCHTRIPLETJOIN). TJ (BTJ) has the best work complexity for most of the datasets.

DBLP (DB) is the graph of authors and their papers [36]. Github (GI) connects users with their projects [37]. IMDB (IM) contains actors and the movies they played in [38]. Kindle (KI) is the network of books and the users who rated those books [39]. Twitter (TW) contains Twitter users and the tags they mentioned in their tweets [40]. Movielens (ML) is a network of users and the movies they rate [41]. Reuters (RE) contains story-word inclusions in Reuters news [42]. LiveJournal (LJ) is the network of users and their group memberships [43].

IX. EXPERIMENTS

In this section, we evaluate our algorithms in Section VI as well as the adaptations in Section V on real-world datasets (see Section VIII).

All experiments are performed on a Linux operating system running on a machine with Intel Xeon Gold processor at 2.1 GHz and 1125 GB DDR4 memory. The processor contains 4 sockets with each having 13 cores for a total of 52 cores. We implemented our algorithms in C++ with Intel TBB 2020.2 [44] and OpenMP 4.5 [45] and compiled using GCC 10.2.0 at the -O3 level. We use the hashmap implementation in [46]. *Our implementation of all the algorithms is available at https://tinyurl.com/par6cycle-code.* We terminated the computation if it took more than 24 hours to finish, denoted by "-" in the results. We also denote the computations that go out of memory by "OOM".

We first consider the strong scaling performance of our algorithms in Section IX-A. Then, we analyze the weak scaling behavior in Section IX-B. Next, we look at the memory usage in Section IX-C. Lastly, we examine the impact of PRE and how different choices translate to improvements in runtime in Section IX-D.

A. Strong Scaling Experiments

Here we provide the strong scaling experiments for all algorithms. Table III shows our runtime experiments on real-world networks using 1 and 52 threads. We also show the speedup of BTJ compared to the second best algorithm in terms of runtime on 52 threads. NJ and WJ are not able to finish computation even with 52 threads on the three largest networks: ML, RE, and LJ. The runtimes for WJ are typically better than NJ as suggested by the numerical calculation of time complexities in Table II. For TJ, the three largest networks timed out on a single thread but were completed on 52 threads. BTJ has the best sequential and parallel runtimes. The only configuration where it could not finish the computation in 24 hours is the sequential run on LJ. On the largest network with $112M$ edges (LJ), BTJ is able to finish the computation in 78 mins by using 52 threads. It is $6.4\times$ faster than the next best algorithm, TJ.

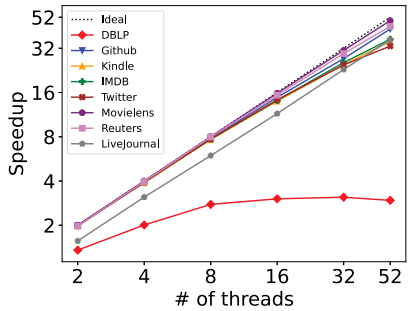

Fig. 4. Speedup of BATCHTRIPLETJOIN for strong scaling experiments on 2, 4, 8, 16, 32, and 52 threads when compared to a single thread. We also show the ideal speedup as a dotted line.

In Fig. 4, we plot the speedup of BTJ on 2, 4, 8, 16, 32, and 52 threads. To show the speedup on LJ, we ran BTJ until completion, which took approximately 47.4 hours with a single thread. Comparing 52 threads to a single thread, there is approximately a 3x speedup for DB, 33x speedup for TW, 36x speedup for KI, IM, and LJ, 44x speedup for GI and RE, and a 49x speedup for ML. DB, the network with the smallest induced 6-cycle count, does not scale after 8 threads. The larger networks, such as ML, RE, and LJ, do not have a significant decline in scalability when using 52 threads, suggesting that they would continue scaling for even larger number of threads. GI, ML, and RE achieved the highest speedup due to their large induced 6-cycle counts in relation to graph size (Table I), indicating a dense induced 6-cycle structure. With the speedup numbers consistently over 32x on 52 threads for the networks with the largest induced 6-cycle counts, BTJ exhibits strong scalability.

TABLE IV
RUNTIME (IN SECONDS) OF OUR ALGORITHMS ON 52 DUPLICATES OF THE ORIGINAL DATASET USING 52 THREADS

Alg.	GI	KI	IM	TW	ML
NJ	4,152	1,695	7,534	2,429	–
WJ	2,845	*OOM*	*OOM*	850	*OOM*
TJ	3,264	2,352	3,529	573	–
BTJ	512	357	1,084	129	47,530

"–" denotes >24 hours and "OOM" means the computation run out of memory. All other notations are defined in Sections II and VIII.

TABLE V
MEMORY USED (IN GIGABYTES)

Alg.	DB	GI	KI	IM	TW	ML	RE	LJ
NJ	0.54	1.54	0.69	1.52	0.56	-	-	-
WJ	0.70	0.67	1.10	2.68	0.51	-	-	-
TJ	0.71	1.64	3.47	8.53	1.39	17.1	32.6	125
BTJ	1.64	0.45	1.15	2.36	0.65	3.44	14.1	36.7

"–" indicates that the experiment timed out after 24 hours. All other notations are defined in Sections II and VIII.

TABLE VI
FRACTION OF RUNTIME SPENT ON PRE FOR BTJ ON 52 THREADS

DB	GI	KI	IM	TW	ML	RE	LJ
0.918	0.003	0.055	0.021	0.058	<0.001	0.001	0.002

All notations are denoted in Section VIII.

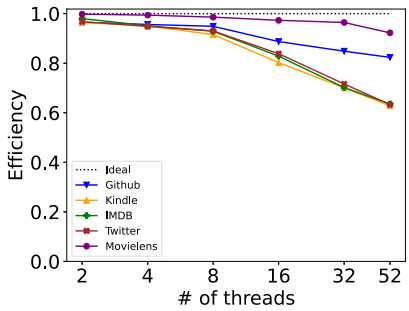

Fig. 5. Efficiency of BATCHTRIPLETJOIN for weak scaling experiments on 2, 4, 8, 16, 32, and 52 threads when compared to a single thread. We also show the ideal efficiency (= 1.0) as a dotted line.

B. Weak Scaling Experiments

Here we provide the weak scaling experiments for all algorithms. For weak scaling, we consider x duplicates of the original network when running on x number of threads. Runtime results on 52 threads for GI, KI, IM, TW, and ML are shown in Table IV. All the algorithms timed out in 24 hours for RE and LJ on 52 threads. Also, DB achieved poor weak scaling results since PRE accounts for over 90% of the time spent (more details in Table VI). BTJ is the only algorithm that could compute the duplicated ML network, which has $520M$ edges, in under 24 hours (13.2 hours). WJ was unable to process the weak scaling experiments on 52 threads for KI, IM, and ML due to prohibitive space complexity. BTJ outperforms the other algorithms significantly in terms of runtime.

We plotted the weak scaling behavior of BTJ on 2, 4, 8, 16, 32, and 52 threads in Fig. 5. We computed the ratio of runtime using x threads on the duplicated network to the sequential runtime on the original network. BTJ performs best on the networks with the highest proportion of induced 6-cycles to graph size, GI and ML. Having approximately 60% efficiency or better even when the graph is 52 times the size of the original, BTJ can be considered to be scalable in terms of weak scaling although there is room for improvement.

TABLE VII
ABLATION STUDY FOR THE THREE TECHNIQUES IN PRE

Omission	DB	GI	KI	IM	TW	ML	RE	LJ
None	1.65	**9.75**	**6.30**	18.7	**2.47**	**890**	**1,890**	**4,682**
2-Core	2.39	10.9	8.11	18.6	1,809	943	2,059	–
Node Set Swaps	10.8	10.1	22.9	**6.67**	1,630	–	–	–
Node Reordering	**1.50**	11.7	8.04	20.6	3.86	967	5,189	50,109

Runtime results (in seconds) for BTJ on 52 threads (best in bold). "–" denotes >24 hours. All notations are denoted in Section VIII.

C. Memory Experiments

Here we give the memory usage results. We used the system activity reporter utility (SAR) and run it in the background during the computation. We measure the amount of used space in our machine every second and report the maximum amount used in Table V. Unlike the other algorithms, NODEJOIN does not use any global storage across parallel threads, and thus typically requires the least memory. BTJ has a significantly smaller memory footprint than TRIPLETJOIN thanks to globally storing the sizes of set intersections instead of the entire set. This is also in line with the space complexities of the algorithms.

D. Analyzing the PREPROCESSING

We analyze the impact of the various techniques used in PRE for the best performing algorithm BTJ. As seen in Table II, which lists the size and statistics of the graphs after preprocessing, there is a significant reduction in graph sizes which directly impacts the runtime of the algorithms.

We first look at how much time PRE takes. Table VI shows the proportion of runtime spent on PRE for BTJ on 52 threads. For DB, the dataset with the smallest number of induced 6-cycles, the majority of the time was spent on PRE. Note that DB is the second largest network in terms of node count and has the third highest number of edges, so applying PRE on the nodes and edges takes a significant amount of time compared to traversing the relatively small number of induced 6-cycles. This is the opposite for all the other networks - PRE takes minimal time compared to the rest of the computation. It is even negligible for the three largest networks - ML, RE, and LJ. PRE is the most useful when there is a significant number of induced 6-cycles in the graph, which is true for most networks in our dataset. Handling the networks with low induced 6-cycle density remains a challenge, as exemplified by DB.

Next, we perform an ablation study for the three techniques in PRE. Table VII shows the runtime on 52 threads when a section of PRE is skipped. Applying all the techniques (denoted by "None") achieves the best result for six of the eight networks, including the three largest (ML, RE, and LJ). Certain preprocessing techniques has drastic impacts in some networks, e.g.,

TABLE VIII
RUNTIME (IN SECONDS) FOR DIFFERENT ORDERINGS FOR BTJ ON 52 THREADS (BEST IN **BOLD**)

Net.	None	Degree		Degeneracy		Wedge	
		Dec	Inc	Dec	Inc	Dec	Inc
DB	**1.50**	1.64	1.65	2.91	2.60	1.63	1.65
GI	11.7	10.9	10.7	10.7	10.9	10.5	**9.75**
KI	8.04	7.77	6.87	8.18	7.33	7.66	**6.30**
IM	20.6	21.0	18.8	22.6	20.5	21.0	**18.7**
TW	3.86	7.30	2.58	5.29	4.40	7.33	**2.47**
ML	967	909	932	968	989	**879**	890
RE	5,189	16,406	2,644	3,231	3,097	12,219	**1,890**
LJ	50,109	–	5,085	23,572	17,074	–	**4,682**

We show the decreasing and increasing variants for each. "–" denotes >24 hours. All other notations are defined in Section VIII.

TW benefits heavily from 2-core filtering and node set swapping as 23% of the network is removed during the 2-core filtering and there is a large imbalance between the node sets. With larger networks, the runtime savings from each PRE section is more significant, in particular 2-core filtering and swapping node sets provide drastic gains (see Tables I and II).

Lastly, we check the impact of different node ranking choices. In line 3 of PRE, we perform increasing wedge ordering on U. To test its performance compared to other ordering schemes, we conduct experiments using degree, degeneracy, and wedge orderings. Degree ordering ranks the nodes based on their degrees. Degeneracy ordering is an ordering of vertices given by repeatedly finding and removing vertices of smallest degree, also known as ordering by core numbers. Wedge ordering uses the number of 2-paths from each node. Table VIII shows the runtime with 52 threads on the decreasing and increasing versions of each of those ordering schemes. We have also included the runtime when no ordering is implemented, denoted as "None", to measure the impact of node ordering on speed. The increasing versions of each ordering scheme typically outperform the decreasing versions. In the increasing versions, the amount of work (i.e., number of induced 6-cycles) is more evenly distributed across parallel threads, preventing the runtime of one thread to dominate over the others. For example, in increasing degree ordering, the highest degree node is likely to participate in more induced 6-cycles compared to a lower degree node. Since we process induced 6-cycles based on the node with minimum id and higher degree nodes are assigned a higher id, we process a lower proportion of induced 6-cycles for higher degree nodes in their parallel threads (and vice versa). Comparing individual ordering schemes, increasing wedge ordering outperforms the other ordering schemes in six of eight networks, including the two largest networks (RE and LJ), which is why we consider it as the default ordering in PRE.

X. BUTTERFLY VERSUS INDUCED 6-CYCLE

Here we present case studies on the Amazon ratings-Kindle items dataset, Amazon-Kindle (AK in short) [47], the Steam-Games network (SG in short) [48], [49], [50], and the Yelp reviews-businesses dataset, Yelp-Business (YB in short) [51]. AK contains 5.7 M Kindle store product reviews spanning May 1996 - Oct 2018, SG contains 7 M reviews from the Steam video game platform, and YB contains 6.7 M business reviews for 150 K businesses. For our case studies, we use Algorithm 6, FINDTRIPLETS, along with a maximal butterfly and minimal induced 6-cycle variant.

Fig. 6. Amazon-Kindle's Kindle item projections. Red: 430 K butterflies and 0 induced 6-cycles; Green: 260 butterflies and 2.5 K induced 6-cycles.

A. Amazon-Kindle *And* Steam-Games

For AK, Fig. 6 shows anecdotal examples from the projection graphs containing Amazon Kindle products. For the induced 6-cycle projection, each pair of items are in very few butterflies but the three items as a whole are in many induced 6-cycles. The inverse is true for butterfly projection - we find three products where each pair of items participate in many butterflies but as a group participate in very few induced 6-cycles. In Fig. 6, the butterfly and the induced 6-cycle projections correspond to 430 K (0) and 260 (2.5 K) butterflies (induced 6-cycles), respectively. Both projections contain the illustrated version of *Life on the Mississippi*.

In the induced 6-cycle projection for AK, the *Life on the Mississippi* book is commonly paired with Mark Twain's famous *The Adventures of Tom Sawyer* novel or a Benjamin Franklin autobiography. *Life on the Mississippi* is a real-life memoir about Mark Twain's personal experience along the Mississippi river and *The Adventures of Tom Sawyer* is a fictional work about a boy growing up in the 1840 s. Fans of Mark Twain would be interested in both Mark Twain's books while history enthusiasts would prefer the nonfiction autobiographies of Mark Twain and Benjamin Franklin. With *The Adventures of Tom Sawyer* having more than twice the number of reviews than the other two novels, many Benjamin Franklin autobiography readers have read this classic novel without an interest in Mark Twain. After all, Benjamin Franklin is known as a polymath while Mark Twain is mainly known as a writer.

The butterfly projection for AK provides a different perspective: it contains three different versions of the *Life on the Mississippi* book. This is an aspect of Amazon's review system where users who rated one of these versions automatically reviewed all three. As such, there exists no induced 6-cycles with these three items as they form a large bi-clique.

If we disregard any duplicate edges, butterflies can relate to a series of items. While AK features many induced 6-cycle dominant clusters, SG contains several butterfly dominant relationships. In SG, there exists a clique-like relationship within the Blackwell and Space Pilgrim video game franchises (Fig. 7). With 490 induced 6-cycles and 36 K butterflies, there is a strong link between the first three games in the Blackwell video game series. Episodes 2, 3, and 4 of the Space Pilgrim series also

Fig. 7. `Steam-Games`'s Steam game projections. Top: 36 K butterflies and 490 induced 6-cycles; Bottom: 6 K butterflies and 102 induced 6-cycles.

Fig. 8. `Yelp-Business`'s business projection with 70 K induced 6-cycles and 2.6 K butterflies.

corresponds to just 102 induced 6-cycles compared to 6 K butterflies.

B. `Yelp-Business`

Induced 6-cycles, as with butterflies, can find the proximity of businesses to each other. Groups of businesses in `YB` which have high butterfly or induced 6-cycle counts are typically located close to each other. The majority of customers would rather visit businesses close to their area than those far away.

Fig. 8 contains three New Orleans, LA restaurants which are in 70262 induced 6-cycles and 2633 butterflies. Their locations form an 'L' shape where the the seafood locations (*Deanie's Seafood* and *The Original Pierre Maspero's*) are the endpoints and the bar (*The Carousel Bar & Lounge*) is in the middle. The comparable seafood locations are the farthest distance away, reducing the number of intersecting customers and therefore its butterfly count. For three nodes to have a high induced 6-cycle and low butterfly count, each pair of nodes should participate in a similar number of butterflies as the other two pairs. Since the two seafood locations share a strongly overlapping customer base, increasing the distance between them causes its butterfly count to be comparable with the pairs containing the distinctive bar.

Groups of three businesses which share many induced 6-cycles but few butterflies often follow a trend in their relative positions. Those with comparable characteristics tend to form a triangle in their placements stretched away from high population areas. Distance becomes a greater factor than preference the more similar businesses are to each other, often causing customers to disregard the farthest location. Therefore, many customers only consider the two closest businesses but not the third, creating induced 6-cycles. While butterflies can signal how close businesses are to each other with its clique-like projections, induced 6-cycles can provide an intuition for the relative direction between businesses. As a future work, one can predict the relative distances and perspectives between businesses based on their butterfly and induced 6-cycle counts.

XI. Conclusion and Future Work

We introduced efficient and scalable parallel algorithms to count induced 6-cycles in bipartite networks. To the best of our knowledge, this is the first inquiry in induced 6-cycle counting. Experiments on real-world bipartite networks show that our best algorithm, BatchTripletJoin, is highly parallelizable in relation to the number of processors and enables practical computation for large networks with up to half a billion edges; on the 52 times scaled `Movielens` network with a total of $520M$ edges, BatchTripletJoin finishes the computation in 13.2 hours by using 52 threads.

Although BatchTripletJoin exhibits strong performance, it is unable to compute some large networks in under 24 hours with 52 threads, such as the 52 times scaled `Reuters` and `LiveJournal` networks (3B-5B edges). It also shows poor scalability when the network has relatively few induced 6-cycles, as in the `DBLP` network. As a future work, we will investigate scaling our algorithm to larger networks with billions of edges along with handling the networks with low induced 6-cycle counts. One interesting question in this context is how quickly one can terminate the computation if the graph has no induced 6-cycles.

Our framework can also easily be extended to larger bipartite motifs, such as induced 8-cycles, by updating our batch scheme from node triplets to node quadruples. It would be interesting to see potential applications regarding these larger structures. Dynamic bipartite graphs are another avenue for innovation as many real-world networks change over time. We will investigate fast approaches to updating the induced 6-cycle count with each outgoing or incoming connection.

References

[1] S. P. Borgatti and M. G. Everett, "Network analysis of 2-mode data," *Social Netw.*, vol. 19, no. 3, pp. 243–269, 1997.
[2] M. Latapy, C. Magnien, and N. D. Vecchio, "Basic notions for the analysis of large two-mode networks," *Social Netw.*, vol. 30, no. 1, pp. 31–48, 2008.
[3] D. Gibson, R. Kumar, and A. Tomkins, "Discovering large dense subgraphs in massive graphs," in *Proc. 31st Int. Conf. Very Large Data Bases*, 2005, pp. 721–732.
[4] I. Dhillon, "Co-clustering documents and words using bipartite spectral graph partitioning," in *Proc. 7th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2001, pp. 269–274.
[5] D. C. Fain and J. O. Pedersen, "Sponsored search: A brief history," *Bull. Amer. Soc. Inf. Sci. Technol.*, vol. 32, no. 2, pp. 12–13, 2006.
[6] X. Li and H. Chen, "Recommendation as link prediction in bipartite graphs: A graph kernel-based machine learning approach," *Decis. Support Syst.*, vol. 54, no. 2, pp. 880–890, 2013.
[7] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, no. 6684, pp. 440–442, 1998.
[8] M. E. Newman, "Scientific collaboration networks. I. network construction and fundamental results," *Phys. Rev. E*, vol. 64, no. 1, 2001, Art. no. 016131.

[9] G. Robins and M. Alexander, "Small worlds among interlocking directors: Network structure and distance in bipartite graphs," *Comput. Math. Org. Theory*, vol. 10, no. 1, pp. 69–94, 2004.

[10] A. E. Sariyüce and A. Pinar, "Peeling bipartite networks for dense subgraph discovery," in *Proc. 11th ACM Int. Conf. Web Search Data Mining*, 2018, pp. 504–512.

[11] A. R. Benson, R. Abebe, M. T. Schaub, A. Jadbabaie, and J. Kleinberg, "Simplicial closure and higher-order link prediction," in *Proc. Nat. Acad. Sci.*, vol. 115, no. 48, pp. E11221–E11230, 2018.

[12] C. Seshadhri, A. Pinar, and T. G. Kolda, "Triadic measures on graphs: The power of wedge sampling," in *Proc. SIAM Int. Conf. Data Mining*, SIAM, 2013, pp. 10–18.

[13] N. K. Ahmed, J. Neville, R. A. Rossi, N. G. Duffield, and T. L. Willke, "Graphlet decomposition: Framework, algorithms, and applications," *Knowl. Inf. Syst.*, vol. 50, no. 3, pp. 689–722, 2017.

[14] J. Wang, A. W.-C. Fu, and J. Cheng, "Rectangle counting in large bipartite graphs," in *Proc. IEEE Int. Cong. Big Data*, 2014, pp. 17–24.

[15] S.-V. Sanei-Mehri, A. E. Sariyuce, and S. Tirthapura, "Butterfly counting in bipartite networks," in *Proc. 24th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2018, pp. 2150–2159.

[16] S. G. Aksoy, T. G. Kolda, and A. Pinar, "Measuring and modeling bipartite graphs with community structure," *J. Complex Netw.*, vol. 5, no. 4, pp. 581–603, 2017.

[17] R. Kannan et al., "Receipt: Refine coarse-grained independent tasks for parallel tip decomposition of bipartite graphs," in *Proc. VLDB Endowment*, vol. 14, pp. 404–417, 2020.

[18] Q. Zhu, J. Zheng, H. Yang, C. Chen, X. Wang, and Y. Zhang, "Hurricane in bipartite graphs: The lethal nodes of butterflies," in *Proc. 32nd Int. Conf. Sci. Statist. Database Manage.*, 2020, pp. 1–4.

[19] N. Chiba and T. Nishizeki, "Arboricity and subgraph listing algorithms," *SIAM J. Comput.*, vol. 14, no. 1, pp. 210–223, 1985.

[20] J. Shi and J. Shun, "Parallel algorithms for butterfly computations," in *Proc. Symp. Algorithmic Princ. Comput. Syst.*, SIAM, 2020, pp. 16–30.

[21] S.-V. Sanei-Mehri, Y. Zhang, A. E. Sariyüce, and S. Tirthapura, "Fleet: Butterfly estimation from a bipartite graph stream," in *Proc. 28th ACM Int. Conf. Inf. Knowl. Manage.*, 2019, pp. 1201–1210.

[22] T. Opsahl, "Triadic closure in two-mode networks: Redefining the global and local clustering coefficients," *Social Netw.*, vol. 35, no. 2, pp. 159–167, 2013.

[23] Y. Yang, Y. Fang, M. E. Orlowska, W. Zhang, and X. Lin, "Efficient bi-triangle counting for large bipartite networks," in *Proc. VLDB Endowment*, vol. 14, no. 6, pp. 984–996, 2021.

[24] N. Pržulj, D. G. Corneil, and I. Jurisica, "Modeling interactome: Scale-free or geometric?," *Bioinformatics*, vol. 20, no. 18, pp. 3508–3515, 2004.

[25] N. Pržulj, "Biological network comparison using graphlet degree distribution," *Bioinformatics*, vol. 23, no. 2, pp. e177–e183, 2007.

[26] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, "Network motifs: Simple building blocks of complex networks," *Science*, vol. 298, no. 5594, pp. 824–827, 2002.

[27] E. Martorana, G. Micale, A. Ferro, and A. Pulvirenti, "Establish the expected number of induced motifs on unlabeled graphs through analytical models," *Appl. Netw. Sci.*, vol. 5, no. 1, 2020, Art. no. 58.

[28] C. Seshadhri and S. Tirthapura, "Scalable subgraph counting: The methods behind the madness," in *Proc. Web Conf.*, 2019, pp. 1317–1318.

[29] A. Dehghan and A. H. Banihashemi, "Counting short cycles in bipartite graphs: A fast technique/algorithm and a hardness result," *IEEE Trans. Commun.*, vol. 68, no. 3, pp. 1378–1390, Mar. 2020.

[30] J. Niu, J. Zola, and A. E. Sariyüce, "Counting induced 6-cycles in bipartite graphs," in *Proc. 51st Int. Conf. Parallel Process.*, 2022, pp. 1–10.

[31] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. Cambridge, MA, USA: MIT Press, 2022.

[32] M. Karimi and A. H. Banihashemi, "Message-passing algorithms for counting short cycles in a graph," *IEEE Trans. Commun.*, vol. 61, no. 2, pp. 485–495, Feb. 2013.

[33] V. Batagelj and M. Zaversnik, "An O(m) algorithm for cores decomposition of networks," 2003, *arXiv:cs/0310049*.

[34] J. Kunegis, "KONECT: The koblenz network collection," in *Proc. 22nd Int. Conf. World Wide Web*, 2013, pp. 1343–1350.

[35] E. T. Aaron Clauset and M. Sainz, "The colorado index of complex networks," 2016. [Online]. Available: https://icon.colorado.edu/

[36] M. Ley, "The dblp computer science bibliography: Evolution, research issues, perspectives," in *Proc. Int. Symp. String Process. Inf. Retrieval*, Springer, 2002, pp. 1–10.

[37] S. Chacon, "The 2009 github contest," 2009. [Online]. Available: https://github.com/blog/466-the-2009-github-contest

[38] "IMDB," 2016. [Online]. Available: https://www.imdb.com/interfaces/

[39] R. He and J. McAuley, "Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering," in *Proc. 25th Int. Conf. World Wide Web*, 2016, pp. 507–517.

[40] M. De Choudhury, Y.-R. Lin, H. Sundaram, K. S. Candan, L. Xie, and A. Kelliher, "How does the data sampling strategy impact the discovery of information diffusion in social media?," in *Proc. 4th Int. AAAI Conf. Weblogs Social Media*, 2010, pp. 34–41.

[41] F. M. Harper and J. A. Konstan, "The movielens datasets: History and context," *ACM Trans. Interactive Intell. Syst.*, vol. 5, no. 4, pp. 1–19, 2015.

[42] D. D. Lewis, Y. Yang, T. Russell-Rose, and F. Li, "RCV1: A new benchmark collection for text categorization research," *J. Mach. Learn. Res.*, vol. 5, pp. 361–397, 2004.

[43] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *Proc. 7th ACM SIGCOMM Conf. Internet Meas.*, 2007, pp. 29–42.

[44] C. Pheatt, "Intel threading building blocks," *J. Comput. Sci. Colleges*, vol. 23, no. 4, pp. 298–298, 2008.

[45] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel Programming in OpenMP*. San Mateo, CA, USA: Morgan Kaufmann, 2001.

[46] G. Popovitch, "Parallel-hashmap," 2022. [Online]. Available: https://github.com/greg7mdp/parallel-hashmap

[47] J. Ni, J. Li, and J. McAuley, "Justifying recommendations using distantly-labeled reviews and fine-grained aspects," in *Proc. Conf. Empirical Methods Natural Lang. Process. 9th Int. Joint Conf. Natural Lang. Process.*, 2019, pp. 188–197.

[48] W.-C. Kang and J. McAuley, "Self-attentive sequential recommendation," in *Proc. IEEE Int. Conf. Data Mining*, 2018, pp. 197–206.

[49] M. Wan and J. McAuley, "Item recommendation on monotonic behavior chains," in *Proc. 12th ACM Conf. Recommender Syst.*, 2018, pp. 86–94.

[50] A. Pathak, K. Gupta, and J. McAuley, "Generating and personalizing bundle recommendations on steam," in *Proc. 40th Int. ACM SIGIR Conf. Res. Develop. Inf. Retrieval*, 2017, pp. 1073–1076.

[51] "Yelp open dataset," 2023, [Online]. Available: https://www.yelp.com/dataset/

[52] "Center for computational research, University at Buffalo," 2021, [Online]. Available: http://hdl.handle.net/10477/79221