

# Induced 6-Cycle Counting in Bipartite Graphs

Jason Niu  
jasonniu@buffalo.edu

## Abstract

Finding subgraphs in a bipartite graph is crucial to understanding its underlying structure. The smallest non-trivial subgraph in a bipartite graph is a 4-cycle, which is also known as a butterfly. Shi and Shun recently used the affordances of parallelization to develop efficient butterfly counting algorithms. However, parallel approaches to counting larger cycles is a relatively unexplored area. In this paper, we propose a parallel algorithm for efficiently counting induced 6-cycles.

## 1 Introduction

Many real-world networks are represented by a bipartite graph. For example, recommendation networks often are represented as a bipartite graph with users on one partition and items on the other [5]. Finding graph motifs that form the building blocks of these networks can reveal the underlying structure within bipartite graphs.

In unipartite graphs, the smallest cycle is a 3-cycle, which is also known as a triangle. However, since bipartite graphs contain no odd cycles, triangles do not exist in bipartite graphs. The smallest cycle in a bipartite graph is a 4-cycle, which is also known as a butterfly. Butterflies are the smallest building blocks for community structures in bipartite graphs.

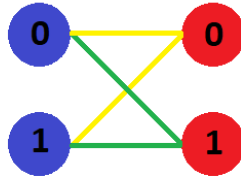


Figure 1: This graph depicts a butterfly. Nodes in  $u$  are in blue while nodes in  $v$  are in red. The butterfly is made of two wedges, which are highlighted in yellow and green.

Counting cycles in bipartite graphs is NP-hard [3]. Sequential algorithms for butterfly counting use the concept of combining wedges (2-paths) to count butterflies (see Figure 1) [1, 6, 8]. When dealing with larger graphs, however, the runtime of sequential algorithms may be problematic. Adapting sequential algorithms for parallelization can significantly reduce the runtime of these algorithms. Shi and Shun [7] recently designed a parallel butterfly counting algorithm which modified Chiba and Nishizeki's wedge retrieval process [1] to enable parallelization.

Given the relevance of bipartite graphs in real-world relationships, it is desirable to find larger cycles within these graphs. Karimi and Banihashemi [4] designed a message-passing algorithm for counting cycles of length  $g$  to  $2g-2$  in bipartite graphs, where  $g$  is the girth of the graph. Dehghan and Banihashemi [2] proposed an algorithm that uses breadth-first search to count cycles of length  $g$  to  $g+4$  in bipartite graphs. However, these sequential algorithms have only been tested on small bipartite graphs.

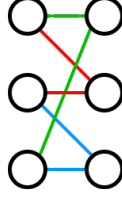


Figure 2: In this figure, a triangle of wedges (colored as red, blue, and green) forms a 6-cycle.

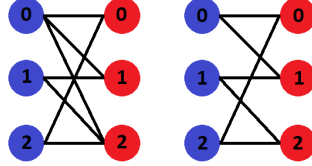


Figure 3: The graph on the left depicts a non-induced 6-cycle. The graph on the right depicts an induced 6-cycle. In the non-induced graph, the removal of the edge from 0 to 2 won't affect the 6-cycle.

Recently, Yang et al. introduced sequential algorithms for counting non-induced 6-cycles in large bipartite networks [9]. Their algorithms are based on the concept that a 6-cycle is a triangle of wedges (see Figure 2) or a pair of 3-paths. In this paper, we propose a parallel algorithm that is also based on the concept that an 6-cycle is a triangle of wedges. Parallelizing algorithms that count larger cycles is extremely valuable due to their increased complexity. This paper presents a framework for counting induced 6-cycles (see Figure 3) in bipartite graphs which uses the affordances of parallelization to maximize efficiency.

## 2 Notation

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 right set, and  $E$  is the set of edges. The neighbors of a node  $v$  is denoted by  $N(v)$ . A **wedge** is a set of three nodes  $u1, u2 \in U$  and  $v \in V$  composed of edges  $(u1, v), (u2, v) \in E$ . We call the nodes  $u1, u2$  **endpoints** and the node  $v$  the **center**. An **induced 6-cycle** is a set of six nodes  $u1, u2, u3 \in U$  and  $v1, v2, v3 \in V$  such that its internal edges exactly form a cycle.

## 3 Algorithm

We introduce a parallel algorithm for counting induced 6-cycles in bipartite graphs. Our algorithm extends the parallel wedge retrieval algorithm proposed by Shi and Shun [7].

---

**Algorithm 1** PREPROCESSING( $G$ )

---

**Input:**  $G$ : graph**Output:**  $G'$ : ordered graph

```
1:  $w_u \leftarrow 0$ 
2: parallel for each  $x \in U$  do
3:    $n \leftarrow |G(x)|$ 
4:    $w_u \leftarrow w_u + \frac{n*(n-1)}{2}$ 
5:  $w_v \leftarrow 0$ 
6: parallel for each  $x \in V$  do
7:    $n \leftarrow |G(x)|$ 
8:    $w_v \leftarrow w_v + \frac{n*(n-1)}{2}$ 
9: if  $w_u < w_v$  then
10:    $Swap(U, V)$  ▷ swap  $U$  and  $V$ 
11:  $X \leftarrow Sort(U)$  ▷ sort  $U$  in increasing order of degree
12: Let  $x$ 's rank  $R(x)$  be its index in  $X$ 
13: parallel for each  $x \in U \cup V$  do
14:   if  $x \in U$  then
15:      $G'(R(x)) \leftarrow \{y | (x, y) \in E\}$ 
16:   else
17:      $G'(x) \leftarrow Sort(\{R(y) | (x, y) \in E\})$  ▷ sort neighbors' ranks by descending order
18: return  $G'$ 
```

---

We give a preprocessing algorithm (Algorithm 1) that takes as input a bipartite graph, swaps left and right sets depending on their wedge counts, and renames nodes in increasing order of degree. *Preprocessing* swaps  $U$  and  $V$  if the number of wedges whose center is  $\in U$  is less than the number of wedges whose center is  $\in V$ . This helps reduce the number of processed wedges. *Preprocessing* then orders nodes in increasing order of degree, which we found to yield efficient results. Shi and Shun [7] proved that using approximate degree ordering, complement degeneracy ordering, and approximate complement degeneracy ordering also gives work-efficient bounds. By establishing an ordering of nodes, we can avoid traversing the same induced 6-cycle twice.

---

**Algorithm 2** GetWedges( $G$ )

---

**Input:**  $G$ : graph**Output:**  $W$ : list of wedges

```
1: parallel for each  $u1 \in U$  do
2:   parallel for each  $v \in N(u1)$  do
3:     parallel for each  $u2 \in N(v)$  do
4:       if  $u2 > u1$  then
5:          $W((u1, v, u2))$  ▷ add wedge to  $W$ 
6:       else
7:         break
8: return  $W$ 
```

---

We define a wedge retrieval algorithm, *GetWedges* (Algorithm 2), that takes as input a preprocessed graph and outputs a list of wedges. We use  $W$  to denote a parallel container such that  $W(x)$  stores  $x$  in the container.  $W$  should allow for fast access of wedges based on endpoints such that

when accessing all wedges whose smallest endpoint is  $u1 \in U$ , wedges are sorted based on the other endpoint. *GetWedges* is based off of Shi and Shun [7] wedge retrieval algorithm which enables the parallel processing of wedges. For all nodes  $u1 \in U$ , the algorithm retrieves all wedges with endpoints  $u1, u2$  and center  $v$  such that  $u2$  is greater than  $u1$ . Note that Shi and Shun proved how once we have retrieved all wedges with endpoint  $u1$ , there is no need to consider wedges with center  $u1$ . Although this point was originally applied to butterfly counting, the same logic can be applied to induced 6-cycles.

---

**Algorithm 3** *Par6CycleCount*( $G, W$ )

---

**Input:**  $G$ : graph  
**Output:**  $c$ : count of induced 6-cycles

```

1:  $G \leftarrow \text{Preprocessing}(G)$ 
2:  $W \leftarrow \text{GetWedges}(G)$ 
3:  $c \leftarrow 0$ 
4: parallel for each  $u1 \in U$  do
5:   for each  $w1 \in W(u1)$  do  $\triangleright u1$  is smallest endpoint in  $w1$ 
6:      $c' \leftarrow 0$ 
7:      $x, y \leftarrow -1$ 
8:      $skip \leftarrow false$ 
9:      $u1, v1, u2 \leftarrow w1$ 
10:    for each  $w2 \in W(u2)$  do  $\triangleright u2$  is smallest endpoint in  $w2$ 
11:       $u2, v2, u3 \leftarrow w2$ 
12:      if  $skip \cap (x = u3)$  then
13:        continue
14:       $x \leftarrow u3$ 
15:       $skip \leftarrow v1 \in N(u3)$ 
16:      if  $!skip \cap (v2 \notin N(u1))$  then
17:        if  $y \neq u3$  then
18:           $c' \leftarrow 0$ 
19:          for each  $w3 \in W(u1, u3)$  do  $\triangleright u1, u3$  are endpoints of  $w3$ 
20:             $u1, v3, u3 \leftarrow w3$ 
21:            if  $v3 \notin N(u2)$  then
22:               $c' \leftarrow c' + 1$ 
23:             $y \leftarrow u3$ 
24:           $c \leftarrow c + c'$ 
25: return  $c$ 

```

---

We now describe the induced 6-cycle counting algorithm, which is given as *Par6CycleCount* in Algorithm 3. Given a bipartite graph  $G$ , this algorithm applies  $G$  to the preprocessing and wedge retrieval algorithms described in Algorithms 1 and 2, respectively. We use  $W(x)$  to denote the set of wedges where  $x$  is the smallest of the two endpoints and  $W(x, y)$  to denote the set of wedges whose endpoints are  $x$  and  $y$ . *Par6CycleCount* finds triangles of wedges  $\{u1, v1, u2\}$ ,  $\{u2, v2, u3\}$ , and  $\{u1, v3, u3\}$  such that  $u1 < u2 < u3$ . For every triangle of wedges, there are a maximum of three checks for inducedness (see Figure 4). Inducedness checks are skipped if a similar set of wedges was processed previously (see Figure 5).

*Proof of correctness.* We will prove Algorithm 3, *Par6CycleCount*, is correct by contradiction.

Suppose there exists a set of nodes that is inaccurately counted as an induced 6-cycle. Since lines

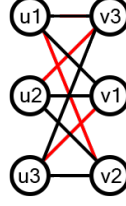


Figure 4: In this figure, the red edges prevent inducedness for the 6-cycle colored in black. Each inducedness check (lines 15, 16, and 21) in Algorithm 3, *Par6CycleCount*, corresponds to a red edge.

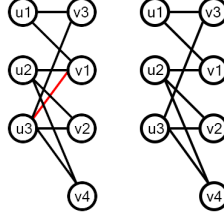


Figure 5: The speedups implemented in lines 12 (left) and 17 (right) of *Par6CycleCount*. The figure on the left shows only one inducedness check (the red edge) is needed for wedges  $\{u2, v2, u3\}$  and  $\{u2, v4, u3\}$ . The figure on the right shows that the count of 6-cycles for wedge  $\{u2, v2, u3\}$  is the same as wedge  $\{u2, v4, u3\}$ .

Therefore, we only need to do calculations for one wedge.

15, 16, and 21 check for inducedness given a set of 6 nodes, every node set counted must be an induced 6-cycle, contradicting our claim.

Suppose there exists an induced 6-cycle that is counted twice. In Algorithm 2, *GetWedges*, the ordering of node traversal in line 4 prevents a wedge from being traversed multiple times, causing each wedge to be stored only once in line 5. There also exists a specific ordering of wedge traversal for *Par6CycleCount* in lines 5, 10, and 19. For every induced 6-cycle, the wedge which contains the smallest two endpoints is traversed first (line 5). Then, the wedge containing the largest two endpoints is traversed second (line 10). Finally, the wedge which contains the smallest and the largest endpoint is traversed last (line 19). This prevents an induced 6-cycle from being traversed multiple times, contradicting our claim of duplicity in induced 6-cycle counting.

Suppose there exists an induced 6-cycle  $x = \{u1, u2, u3, v1, v2, v3\}$  that isn't counted. Then  $x$  contains three wedges  $a = \{u1, v1, u2\}$ ,  $b = \{u2, v2, u3\}$ , and  $c = \{u1, v3, u3\}$ , which the algorithm processes in that order (lines 5, 10, and 19). Therefore, the algorithm counts the triangle of wedges  $a$ ,  $b$ , and  $c$  as an induced 6-cycle, contradicting our claim.

Since all possible cases lead to contradiction, *Par6CycleCount* returns the correct induced 6-cycle count.  $\square$

## 4 Results

In this section, we present the runtime of our algorithm on real-world datasets, whose main characteristics are shown in Table 1.

All experiments were performed on a Linux operating system running on a machine with Intel Xeon Gold processor at 2.1 GHz. We ran our algorithm on 4 sockets with each having 13 cores for a total of 52 cores. We implemented our algorithm in C++ and compiled using gcc 10.2.0 at the -O3 level. The runtime, in seconds, is shown in Table 2. The runtime shown is the average runtime for 3 different runs of *Par6CycleCount* with the specified number of cores.

In terms of runtime in relation to the number of cores, Github, Kindle, and IMDB had significant

Table 1: Bipartite networks

Real-world datasets	U	V	E	# induced 6-cycles
Database	11,193	8,915	30,716	$5.02 \times 10^4$
CondMat	16,727	22,016	58,595	$1.92 \times 10^4$
Marvel	6,487	12,942	96,662	$1.70 \times 10^9$
DBLP	4,000,150	1,425,813	8,649,016	$5.10 \times 10^7$
Github	56,555	123,345	440,237	$1.37 \times 10^{11}$
Kindle	1,406,890	430,530	3,205,467	$5.20 \times 10^9$
IMDB	1,232,031	419,661	5,596,667	$2.01 \times 10^{10}$

Table 2: Runtime (seconds)

Real-world datasets	# cores						
	1	2	4	8	16	32	52
Database	0.034	0.03	0.019	0.019	0.015	0.021	0.026
CondMat	0.039	0.039	0.033	0.026	0.024	0.028	0.03
Marvel	4.034	2.034	1.055	0.592	0.34	0.231	0.236
DBLP	11.054	8.041	5.692	4.06	3.01	2.815	2.79
Github	662.976	334.258	168.153	90.041	46.005	26.653	19.619
Kindle	1191.062	605.218	309.911	168.394	86.568	50.424	35.445
IMDB	1222.31	616.419	317.124	170.187	87.783	50.025	35.404

speedups of about 34x when comparing 52 cores to a single core. There also isn't a significant decline in expected improvement when using 52 cores for these graphs, suggesting that additional cores may be used to significantly speedup the algorithm. The smaller graphs, CondMat and Database, have this decline at around 8 cores. Since our implementation of the algorithm parallelizes based on the size of the node sets, larger graphs typically have a higher ideal number of cores compared to smaller graphs in terms of expected runtime improvement over cost. In *Par6CycleCount*, lines 12 and 17 avoid duplicate processing if the current 6-cycle is similar to a previous 6-cycle (see Figure 5). Marvel's large induced 6-cycle count compared to its graph size indicates a dense induced 6-cycle structure, which results in significant speedups from this duplicate processing avoidance. DBLP has a low induced 6-cycle count compared to its graph size, which indicates a sparse induced 6-cycle structure. Since fewer induced 6-cycles are processed, DBLP has a faster runtime than Github, Kindle, and IMDB even though it is a larger graph.

## 5 Final Remarks

The code used for this report is available at <https://github.com/Deerjason/par6cycle>. Parallelization is achieved through the use of Intel Threading Building Blocks due to its multi-core capabilities and strong tasking support. Parallel induced 6-cycle counting is implemented in *6CycleCount.cpp*.

## References

- [1] Norishige Chiba and Takao Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on computing*, 14(1):210–223, 1985.
- [2] Ali Dehghan and Amir H Banihashemi. Counting short cycles in bipartite graphs: A fast technique/algorithm and a hardness result. *IEEE Transactions on Communications*, 68(3):1378–1390, 2019.

- [3] Jörg Flum and Martin Grohe. The parameterized complexity of counting problems. *SIAM Journal on Computing*, 33(4):892–922, 2004.
- [4] Mehdi Karimi and Amir H Banihashemi. Message-passing algorithms for counting short cycles in a graph. *IEEE transactions on communications*, 61(2):485–495, 2012.
- [5] Xin Li and Hsinchun Chen. Recommendation as link prediction in bipartite graphs: A graph kernel-based machine learning approach. *Decision Support Systems*, 54(2):880–890, 2013.
- [6] Seyed-Vahid Sanei-Mehri, Ahmet Erdem Sariyuce, and Srikanta Tirthapura. Butterfly counting in bipartite networks. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2150–2159, 2018.
- [7] Jessica Shi and Julian Shun. Parallel algorithms for butterfly computations. *arXiv preprint arXiv:1907.08607*, 2019.
- [8] Jia Wang, Ada Wai-Chee Fu, and James Cheng. Rectangle counting in large bipartite graphs. In *2014 IEEE International Congress on Big Data*, pages 17–24. IEEE, 2014.
- [9] Yixing Yang, Yixiang Fang, Maria E Orlowska, Wenjie Zhang, and Xuemin Lin. Efficient bi-triangle counting for large bipartite networks. *PVLDB*, 14(6):984–996, 2021.