Induced 6-Cycle Counting in Bipartite Graphs

March 2, 2021

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 parallization 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.

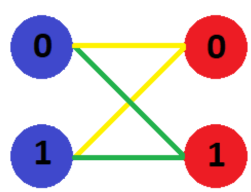

Figure 1: This graph depicts a butterfly. Nodes in u are in blue while nodes in v are in red. The butterfly is made of two wedges, which are highlighted in yellow and green.

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 parallization 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 parallization.

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

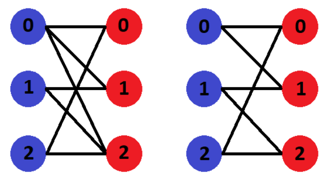

Figure 2: The graph on the left depicts a non-induced 6-cycle. The graph on the right depicts an induced 6-cycle. Nodes in u are in blue while nodes in v are in red. In the non-induced graph, the removal of the edge from u0 to v2 won't affect the 6-cycle.

bipartite graphs. However, these sequential algorithms have only been tested on small bipartite graphs. Parallizing 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 2) in bipartite graphs which uses the affordances of parallization 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)*. The ranking of a node *v* is denoted by *R(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: *R*: ranking of nodes

$X \leftarrow Sort(U \cup V)$ ▷ sort vertices in decreasing order of degree
Let x's rank $R(x)$ be its index in X
parallel for each $x \in X$ **do**
    $N(x) \leftarrow Sort(y|(x, y) \in E)$ ▷ sort neighbors by decreasing order of rank
end for
return R

We give a preprocessing algorithm (Algorithm 1) that takes as input a bipartite graph and returns a ranking of nodes in decreasing order of degree. Preprocessing also sorts neighbors by decreasing order of rank. 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*, *R*)

Input: *G*: graph, *R*: ranking of nodes
Output: *W*: list of wedges

initialize W
parallel for each $u1 \in U \cup V$ **do**
    parallel for each $v \in N(u1)$ **do**
        if $R(v) > R(u1)$ **then**
            parallel for each $u2 \in N(v)$ **do**
                if $R(u2) > R(u1)$ **then**
                    $W((u1, u2, v))$
                else
                    break
                end if
            end for
        else
            break
        end if
    end for
end for
return W

We define a wedge retrieval algorithm, GetWedges (Algorithm 2), that takes as input a preprocessed graph and its ranking *R*. We use *W* to denote a parallel unordered container such that *W(x)* stores *x* in the container. GetWedges is based off of Shi and Shun [7] wedge retrieval algorithm which enables the parallel processing of wedges. For all nodes *u1* in *G*, the algorithm retrieves all wedges with endpoints *u1, u2* and center *v* such that *u2* and *v* both have rank greater than *u1*.

Algorithm 3 BFSCount(*G*, *R*, *w*)

Input: *G*: graph, *R*: ranking of nodes, and *w*: wedge
Output: *c*: count of induced 6-cycles

```
c ← 0
u1, u2, v ← w
For any node u ∈ G, d(u) = ∞
d(u2) ← 0
Let Q ← queue
Q.enqueue(u2)
while Q is not empty do
    x = Q.dequeue
    parallel for each y ∈ N(x) do
        if d(y) = ∞ then
            d(y) ← d(x) + 1
            if d(y) = 1 then
                if R(y) > R(v) and y ∉ N(u1) then
                    Q.enqueue(y)
                end if
            else if d(y) = 2 then
                if y ∉ N(v) then
                    Q.enqueue(y)
                end if
            else if d(y) = 3 then
                if R(y) > R(v) and y ∉ N(u2) then
                    Q.enqueue(y)
                end if
            else
                if y = u1 then
                    c ← c + 1
                    d(y) ← ∞
                end if
            end if
        end if
    end for
end while
return c
```

BFSCount (Algorithm 3) is a modified version of the traditional breadth-first search algorithm. The algorithm takes as input a wedge *w = (u1, u2, v)* and returns the number of induced 6-cycles with nodes *u1, u2, u3* $\in$ *U* and *v, v2, v3* $\in$ *V* such that *v2* and *v3* both have rank greater than *v*.

Algorithm 4 Par6CycleCount(*G*)

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

$R \leftarrow Preprocessing(G)$
$W \leftarrow GetWedges(G, R)$
$c \leftarrow$0
parallel for each $w \in W$ **do**
    $c \leftarrow c + BFSCount(G, R, w)$
end for
return c

We now describe the full induced 6-cycle counting algorithm, which is given as Par6CycleCount in algorithm 4. Given a bipartite graph G, this algorithm applies G to the preprocessing and wedge retrieval algorithms described in algorithms 1 and 2, respectively. The summation of applying the modified BFS algorithm in algorithm 3 to each retrieved wedge is then returned as the total induced 6-cycle count.

The code used for this report is available at https://github.com/Deerjason/par6cycle. Parallization is achieved through the use of Intel Threading Building Blocks due to its multicore capabilities. Currently, parallel butterfly counting is implemented in the develop branch in butterflyCount.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.