2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) | 978-1-6654-9747-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/IPDPSW55747.2022.00060

Families of Butterfly Counting Algorithms for Bipartite Graphs

Jay A. Acosta
Department of Computer Science
The University of Texas at Austin
Austin, USA
jayacosta@utexas.edu

Tze Meng Low
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, USA
lowt@cmu.edu

Devangi N. Parikh
Department of Computer Science
Oden Institute
The University of Texas at Austin
Austin, USA
dnp@cs.utexas.edu

***Abstract*—Butterflies are an important motif found in bipartite graphs that provide a structural way for finding dense regions within the graph. Beyond counting butterflies and enumerating them, other metrics and peeling for bipartite graphs are designed around counting butterfly motifs. The importance of counting butterflies has led to many works on efficient implementations for butterfly counting, given certain situational or hardware constraints. However, most algorithms are based on first counting the building block of the butterfly motif, and from that calculating the total possible number of butterflies in the graph. In this paper, using a linear algebra approach, we show that many provably correct algorithms for counting butterflies can be systematically derived. Moreover, we show how this formulation facilitates butterfly peeling algorithms that find the k-tip and k-wing subgraphs within a bipartite graph.**

***Index Terms*—graph algorithms, bipartite graphs, butterflies, linear algebra**

I. Introduction

Counting and extracting basic graph motifs allows one to study the interactions between vertices within a graph. For unipartite graphs, the triangle (3-clique) is one of the commonly used graph motifs. Triangle counting [8], as well as k-core and k-truss graph peeling algorithms [4] are among the commonly studied algorithms for unipartite graphs.

However, for bipartite graphs, given the unique structure, these triangle-based motifs are not applicable. Consequently, bipartite graphs often use another basic graph motif: the 2×2 biclique or butterfly. Similar to the triangle graph motif, the butterfly can be considered the basic unit of interaction underlying in a bipartite graph. In addition to modeling basic interaction in bipartite graphs, the count of the total number of butterflies can yield information about the clustering coefficient of a graph [15]. Therefore, determining the number of butterflies within a network reveals insights about how the networks components interact both at a lower level and at a larger scale.

Current works related to the butterfly motif focus on providing efficient implementations for counting butterflies under specific scenarios or within hardware constraints. Wang et. al [14] introduced an algorithm that is based on first counting the building block of the butterfly motif and from that calculating the total possible number of butterflies in the graph. Moveover, they present variations of the algorithm that in one case minimize the amount of work space needed and in another reduce the I/O cost. This work forms the basis of other butterfly-based algorithms. The work presented in [10] shows how an approximate count of the butterflies in a bipartite graph can be obtained, while the work in [12] focuses on the parallel implementation of counting butterflies.

For unipartite graphs, it has been shown that by formulating a graph algorithm in terms of linear algebra operations, one can derive a family of algorithms [6]. This can be done by using the FLAME methodology, which is a systematic approach to derive valid loop invariants and hence loop-based algorithms from the specification of the operation [1]. In this paper, we examine the linear algebra formulation to count the butterfly motif in a bipartite graph. Bipartite graphs are special, since the adjacency matrix can be fully defined by a non-square non-symmetric matrix. Moreover, we will show that this linear algebra formulation allows us to derive butterfly peeling algorithms as well.

Contributions. The contributions of this paper are summarized as below:

- A linear algebra formulation to count the exact number of butterflies in a bipartite graph, extract k-tip subgraphs and k-wing subgraphs.
- A family of algorithms for exact butterfly counting in bipartite graphs.
- While implementation performance is not a primary focus of this paper, we present preliminary results of both single-threaded as well as multi-threaded implementations of butterfly counting algorithms.

This paper is organized as follows: In Section II we present the linear algebra formulation of the specification of counting butterflies, while in Section III we derive a family of possible algorithms for counting butterflies. In Section IV, we show how butterfly peeling algorithms can be similarly derived. We show preliminary performance results in Section V, and finally we summarize the conclusions and future work in Section VI.

II. Specification of Butterfly Counting

Let $G = (V_1, V_2, E)$ be a simple, undirected, bipartite graph with vertex sets V_1 and V_2. Let the size of the vertex set V_1 be

978-1-6654-9747-3/22/$31.00 ©2022 IEEE
DOI 10.1109/IPDPSW55747.2022.00060

m and size of the vertex set V_2 be n. In a bipartite graph G, a butterfly is a path $(v_{1i}, v_{2k}, v_{1j}, v_{2p}, v_{1i})$ of length 4, such that vertices $v_{1i}, v_{1j} \in V_1$ and $v_{2k}, v_{2p} \in V_2$. An example of a butterfly is shown in Fig. 1. Some refer to this motif in a bipartite graph as a rectangle [14].

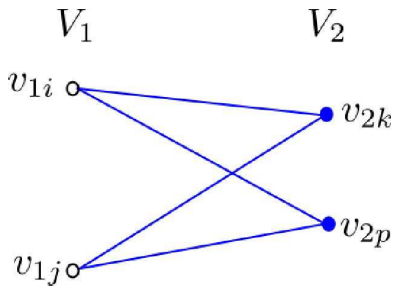

Fig. 1: A bipartite graph G with bipartition V_1 and V_2. The path $(v_{1i}, v_{2k}, v_{1j}, v_{2p}, v_{1i})$ is a butterfly.

A butterfly between two vertices $v_{1i}, v_{1j} \in V_1$ is formed of two distinct paths(v_{1i}, v_{2k}, v_{1j}) and (v_{1i}, v_{2p}, v_{1j}) of length 2. This path of length two that has distinct endpoints is called a wedge. For wedge (v_{1i}, v_{2k}, v_{1j}), vertices v_{1i} and v_{1j} are the endpoints of the wedge, while v_{2k} is called the wedge point.

By counting the distinct wedges between any two vertices in either set V_1 or V_2, we can determine the total number of butterflies in the graph by calculating the possible number of combinations of two distinct wedges that can be made. That is, if there are $n \geq 2$ distinct wedges between distinct endpoints $v_{1i}, v_{1j} \in V_1$, then there are a total of $\binom{n}{2}$ butterflies between v_{1i} and v_{1j}. Therefore, by calculating the number of wedges between each distinct pair of vertices in V_1, we can obtain the total number of butterflies in the given bipartite graph G.

A. Linear Algebraic Butterflies

Let A_G be the adjacency matrix of G. Since G is an undirected bipartite graph, A_G has the following form:

$$A_G = \left(\begin{array}{c|c} 0 & A \\ \hline A^T & 0 \end{array}\right),$$

where A is a $m \times n$ matrix of zeros and ones. Each (i, j) entry of the matrix $B = AA^T$ gives us the number of paths of length 2 between vertices $v_{1i}, v_{1j} \in V_1$. B is a symmetric matrix, and the entries along the diagonal give us the paths of length two that start and end at the same vertex. Therefore, the $beta_{ij}$ entries of strictly upper (or lower) triangular portion of B gives us the number of wedges with distinct end points. By adding the $\binom{\beta_{ij}}{2}$ of each entry of the strictly upper triangular portion of B, we obtain the total number of butterflies in the bipartite graph G.

Note that $\binom{\beta_{ij}}{2} = \frac{\beta_{ij}(\beta_{ij}-1)}{2}$. Applying $\binom{\beta_{ij}}{2}$ with all elements on the matrix B allows us to define:

$$C = \frac{1}{2} B \circ (B - J),$$

where J is a $m \times m$ matrix of ones, and $\circ$ is Hadamard matrix multiplication, otherwise known as element-wise multiplication. Each entry of the strictly upper triangular part of C is the number of butterflies between the i-th and j-th vertex of the V_1 vertex set. Therefore, the total number of butterflies in G is given by

$$\bowtie_G = \sum_{i<j} \gamma_{ij}, \tag{1}$$

where γ_{ij} is the (i, j)-th entry of C, and $\bowtie_G$ is the sum of all $\gamma_{i,j}$ where $i < j$. Now,

$$\sum_{ij} \gamma_{ij} = \sum_{i<j} \gamma_{ij} + \sum_{i>j} \gamma_{ij} + \sum_{i=j} \gamma_{ij}.$$

Since B is symmetric, C is also symmetric. Therefore, $\sum_{i<j} \gamma_{ij} = \sum_{i>j} \gamma_{ij}$. Moreover, $\sum_{i=j} \gamma_{ij} = \Gamma(C)$, where $\Gamma(X)$ is the trace of the matrix X. Thus, (1) becomes

$$\bowtie_G = \frac{1}{2} \sum_{ij} \gamma_{ij} - \frac{1}{2} \Gamma(C). \tag{2}$$

Substituting the definition of C in (2), we get

$$\begin{aligned} \bowtie_G &= \frac{1}{4} \sum_{ij} (B \circ (B - J))_{ij} - \frac{1}{4} \Gamma(B \circ (B - J)) \\ &= \frac{1}{4} \sum_{ij} (B \circ (B - J))_{ij} - \frac{1}{4} \Gamma(B \circ B - B). \end{aligned}$$

Now, relying on the properties

$$\sum_{ij} (X \circ Y) = \Gamma(XY^T) = \Gamma(YX^T), \tag{3}$$

[13] and

$$\Gamma(X + Y) = \Gamma(X) + \Gamma(Y),$$

$\bowtie_G$ becomes

$$\begin{aligned} \bowtie_G &= \frac{1}{4}\Gamma(BB^T) - \frac{1}{4}\Gamma(JB^T) - \frac{1}{4}\Gamma(B \circ B) + \frac{1}{4}\Gamma(B) \\ &= \frac{1}{4}\Gamma(BB^T) - \frac{1}{4}\Gamma(B \circ B) \\ &\quad - \left(\frac{1}{4}\Gamma(JB^T) - \frac{1}{4}\Gamma(B)\right). \end{aligned} \tag{4}$$

Recall that each entry of $B = AA^T$ is the number of paths of length 2 starting at vertex $v_{1i}, v_{1j} \in V_1$. Therefore, the first term of (4), $\Gamma(BB^T)$ is the total number of paths of length 4 that start and end at vertex v_{1i}. While a butterfly is a path of length 4 that starts and end at vertex v_{1i}, not all paths of length 4 that start and end at the same vertex are butterflies. Therefore, to count the total number of butterflies, we have to remove the paths of length 4 that are not butterflies. In the following paragraphs, we will explain how the remaining terms in (4) account for the paths that need to be removed.

A path (v_{1i}, v_{2i}, v_{1i}) that has length 2, and starts and ends at vertex v_{1i} is called a line. A diagonal entry of B is the number of lines starting and ending at vertex v_{1i}. Hence, the second term in (4), $\Gamma(B \circ B)$, represents the total number of paths of length 4 that are composed of 2 lines.

The remaining group of paths of length 4 that are not butterflies is of the following form $(v_{1i}, v_{2k}, v_{1j}, v_{2k}, v_{1i})$, a wedge and then the same wedged traced back. Using the same strategy to formulate (2), the total number of wedges is given by

$$\mathcal{W} = \frac{1}{2}\sum_{ij}\beta_{ij} - \frac{1}{2}\Gamma\left(B\right), \tag{5}$$

where β_{ij} is the (i,j)-th entry of B. Rewriting B as $J \circ B^T$, and using (3), $\sum_{ij}\beta_{ij}$ can be written as $\Gamma(JB^T)$. Therefore, the total number of wedges given by (5) can be rewritten as

$$\mathcal{W} = \frac{1}{2}\Gamma(JB^T) - \frac{1}{2}\Gamma\left(B\right), \tag{6}$$

To count the total number of paths of length 4 that are formed by repeating the same wedge twice, we need the total number of wedges given by (6). However, a wedge (v_{1i}, v_{2k}, v_{1j}) creates the following paths of length 4– $(v_{1i}, v_{2k}, v_{1j}, v_{2k}, v_{1i})$ and $(v_{1j}, v_{2k}, v_{1i}, v_{2k}, v_{1k})$. Therefore, to account for the number of paths of length 4 that are formed by repeating the same wedge twice (6) needs to be divided by 2. This accounts for the last two terms in (4).

The count of butterflies given by (4) at first glance may make it seem that we may be end up require more computations, since we first count all paths of length 4 and then remove the paths of length 4 that are not butterflies. However, in Section III, we will see that this is not the case.

Rewriting (4) in terms of the adjacency matrix A, we get

$$\begin{aligned}\bowtie_G =& \frac{1}{4}\Gamma(AA^TAA^T) - \frac{1}{4}\Gamma(AA^T \circ AA^T)\\ &- \left(\frac{1}{4}\Gamma(JAA^T) - \frac{1}{4}\Gamma(AA^T)\right),\end{aligned} \tag{7}$$

since $B = B^T$ and therefore $\Gamma(JB^T) = \Gamma(JB)$.

III. Deriving a Family of Butterfly Counting Algorithms

The core of our approach for identifying loop-based algorithms to count butterflies is to identify loop invariants that track the number of butterflies of different categories that have been counted at the start and end of every iteration of the loop. In this section, we will consider two families of algorithms for counting butterflies where the first vertex in the butterfly is in either V_1 or V_2.

A. Partitioning Set V_2

Consider a partition of the bipartite graph G such that vertex set V_2 is split into two parts as shown in Fig. 2. The blue solid vertices are in one partition while the red vertices indicated with a cross are in another partition. The vertices of V_1 are indicated using an open circle with a black outline. This partitioning creates three categories of butterflies (recall a butterfly between the two vertices in V_1 is formed by combining two distinct wedges):

1) The set of butterflies, $\bowtie_L$, where both distinct wedge points in V_2 are in the L partition. These are the butterflies indicated with the solid blue line between the black open circle vertices, and the blue solid vertices in Fig. 2.
2) The set of butterflies, $\bowtie_{LR}$, where one of the two distinct wedge points V_2 is in the L partition, while the other is in the R partition.
3) The set of butterflies, $\bowtie_R$, where both distinct wedge points in V_2 are in the R partition. These are the butterflies indicated with the dashed red line between the black open circle vertices in V_1, and the red vertices with a cross.

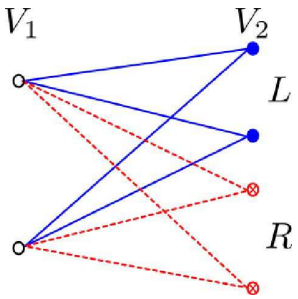

Fig. 2: The partitioning of the vertex set V_2 of Bipartite graph G. The blue solid vertices are in partition L while the red vertices indicated with a cross are in partition R. The vertices of V_1 are indicated using an open circle with a black outline.

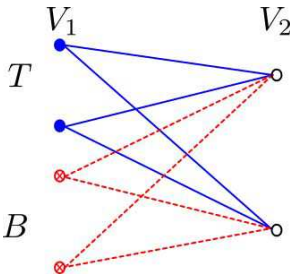

Fig. 3: The partitioning of the vertex set V_1 of Bipartite graph G. The blue solid vertices are in partition T while the red vertices indicated with a cross are in partition B. The vertices of V_2 are indicated using an open circle with a black outline.

These categories of butterflies are all disjoint and therefore the total number of butterflies in the graph G can be computed by adding up the number of butterflies in each category:

$$\bowtie_G = \bowtie_L + \bowtie_{LR} + \bowtie_R. \tag{8}$$

The partitioning in shown in Fig. 2 can be represented by partitioning the adjacency matrix A as follows.

$$A \rightarrow \left(\begin{array}{c|c} A_L & A_R \end{array}\right).$$

Substituting this partitioning into (7), and using the fact that trace is invariant to rotation of its operands, we get

$$\begin{aligned}\bowtie_G =& \frac{1}{4}\Gamma(A_LA_L^TA_LA_L^T) + \frac{1}{4}\Gamma(A_RA_R^TA_RA_R^T)\\ &+ \frac{1}{2}\Gamma(A_LA_L^TA_RA_R^T)\\ &- \frac{1}{4}\Gamma(A_LA_L^T \circ A_LA_L^T) - \frac{1}{4}\Gamma(A_RA_R^T \circ A_RA_R^T)\\ &- \frac{1}{2}\Gamma(A_LA_L^T \circ A_RA_R^T)\\ &- \frac{1}{4}\Gamma(JA_LA_L^T) - \frac{1}{4}\Gamma(JA_RA_R^T)\\ &+ \frac{1}{4}\Gamma(A_LA_L^T) + \frac{1}{4}\Gamma(A_RA_R^T).\end{aligned} \tag{9}$$

Loop Invariant 1
Butterflies in Category 1 counted.

$$\bowtie_G = \bowtie_L$$

Loop Invariant 2
Butterflies in Category 1 and 2 counted.

$$\bowtie_G = \bowtie_L + \bowtie_{LR}$$

Loop Invariant 3
Butterflies in Category 2 and 3 counted.

$$\bowtie_G = \bowtie_R + \bowtie_{LR}$$

Loop Invariant 4
Butterflies in Category 3 counted.

$$\bowtie_G = \bowtie_R$$

Fig. 4: Valid loop invariants that will lead to algorithms that compute the total number of butterflies when V_2 is partitioned into two parts L and R.

Comparing (8) and (9), and using the fact that the trace is a linear mapping, we recognize that

$$\begin{aligned}
\bowtie_L &= \frac{1}{4}\Gamma\left(A_L A_L^T A_L A_L^T - A_L A_L^T \circ A_L A_L^T - J A_L A_L^T + A_L A_L^T\right), \\
\bowtie_{LR} &= \frac{1}{2}\Gamma(A_L A_L^T A_R A_R^T - A_L A_L^T \circ A_R A_R^T), \\
\bowtie_R &= \frac{1}{4}\Gamma\left(A_R A_R^T A_R A_R^T - A_R A_R^T \circ A_R A_R^T - J A_R A_R^T + A_R A_R^T\right).
\end{aligned} \tag{10}$$

Therefore (8) and (9) both represent the postcondition (or specification) of a butterfly counting algorithm.

To derive algorithms using the FLAME methodology, we must first identify valid loop invariants from the postcondition. A loop invariant is an assertion that must be true at the start and end of every iteration of the loop. Moreover, the loop invariant must be true right before and after the loop. A meaningful valid loop invariant indicates partial progress towards the post condition. To derive the loop invariant, we can use either (8) or (9) representation of the post condition. As to what constitutes a valid loop invariant please refer to [9]. From (8), we get four valid loop invariants listed in Figure 4. Using (10), these loop invariants can also be rewritten in terms of A, which we have left out for brevity.

B. Partitioning Set V_1

Symmetrically, we can consider a different partition of the bipartite graph G, where the set V_1 is split into two parts as shown in Fig. 3. This leads to the following three categories of butterflies:

4) The set of butterflies, $\bowtie_T$, where both distinct wedge points in V_1 are in the T partition.
5) The set of butterflies, $\bowtie_{TB}$, where one of the two distinct wedge points V_1 is in the T partition, while the other is in the B partition.
6) The set of butterflies, $\bowtie_B$, where both distinct wedge points in V_1 are in the B partition.

Loop Invariant 5
Butterflies in Category 4 counted.

$$\bowtie_G = \bowtie_T$$

Loop Invariant 6
Butterflies in Category 4 and 5 counted.

$$\bowtie_G = \bowtie_T + \bowtie_{TB}$$

Loop Invariant 7
Butterflies in Category 5 and 6 counted.

$$\bowtie_G = \bowtie_{TB} + \bowtie_B$$

Loop Invariant 8
Butterflies in Category 6 counted.

$$\bowtie_G = \bowtie_B$$

Fig. 5: Valid loop invariants that will lead to algorithms that compute the total number of butterflies when V_1 is partitioned into two parts T and B.

As before, the total number of butterflies in G, can be written as

$$\bowtie_G = \bowtie_T + \bowtie_{TB} + \bowtie_B. \tag{11}$$

The partitioning in shown in Fig. 3 can be represented by partitioning the adjacency matrix A as follows.

$$A \rightarrow \left(\frac{A_T}{A_B}\right)$$

Repeating the same process as described in the previous subsection, we get

$$\begin{aligned}
\bowtie_T &= \frac{1}{4}\Gamma\left(A_T A_T^T A_T A_T^T - A_T A_T^T \circ A_T A_T^T - J A_T A_T^T + A_T A_T^T\right), \\
\bowtie_{TB} &= \frac{1}{2}\Gamma(A_T A_T^T A_B A_B^T - A_T A_T^T \circ A_B A_B^T), \\
\bowtie_B &= \frac{1}{4}\Gamma\left(A_B A_B^T A_B A_B^T - A_B A_B^T \circ A_B A_B^T - J A_B A_B^T + A_B A_B^T\right).
\end{aligned} \tag{12}$$

and another set of loop invariants listed in Fig. 5.

C. Derivation of the algorithm corresponding to Loop Invariant 1.

In this section we use the FLAME methodology to systematically derive the algorithm hand-in-hand with the proof of correctness. We follow the 8 steps listed in the FLAME worksheet [1].

Step 1: Define the pre-condition and post-condition.

We start the algorithm with no butterflies having been counted. Therefore the precondition

$$P_{\text{pre}} : \bowtie_G = 0.$$

Once the algorithm is done, all the butterflies in G must be counted. Using (7), the post-condition is given by

$$P_{\text{post}} : \bowtie_G = \frac{1}{4}\Gamma(AA^TAA^T) - \frac{1}{4}\Gamma(AA^T \circ AA^T) - \left(\frac{1}{4}\Gamma(JAA^T) - \frac{1}{4}\Gamma(AA^T)\right),$$

Step 2: Determine the loop invariant.

In the previous subsections, we identified 8 possible loop invariants. We will proceed with the remaining steps using Loop Invariant 2, which is listed in Fig. 4. Using (10), the loop invariant 2 can be written as

$$P_{\text{inv}} : \bowtie_G = \frac{1}{4}\Gamma\left(A_LA_L^TA_LA_L^T - A_LA_L^T \circ A_LA_L^T - JA_LA_L^T + A_LA_L^T\right) + \frac{1}{2}\Gamma(A_LA_L^TA_RA_R^T - A_LA_L^T \circ A_RA_R^T).$$

Step 3: Determine the loop guard.

Once the loop completes, all the butterflies must be counted. For this to be true, all the vertices in V_2 must be in the L partition. Therefore, the loop must continue to execute while the columns of A_L are less than the columns of A. The loop guard can be written as $n(A_L) < n(A)$, where $n(A_L)$, and $n(A)$ indicates the columns of matrix A_L and A respectively.

Step 4: Determine the initialization.

When we start the loop, the precondition must imply the loop invariant. For this to hold, there must be no vertices in A_L implying that no butterflies has been counted. We must initialize A_L to have 0 columns.

Step 5: Making progress.

In the loop, we must make progress towards counting the butterflies. To make progress we must pick an arbitrary vertex in the R partition, and consider the butterflies in Category 1, and 2. Since the vertex we consider can be any vertex, it is convenient to consider the vertex represented by the first column in the A_R partition.

Therefore, the adjacency matrix A can be repartitioned as follows:

$$\left(\begin{array}{c|c} A_L & A_R \end{array}\right) \leftarrow \left(\begin{array}{c|cc} A_0 & a_1 & A_2 \end{array}\right), \tag{13}$$

where a_1 has 1 column.

Once we count the butterflies associated with a_1, we must move that vertex into the L partition at the bottom of the loop. This means the adjacency matrix is updated as follows:

$$\left(\begin{array}{c|c} A_L & A_R \end{array}\right) \leftarrow \left(\begin{array}{cc|c} A_0 & a_1 & A_2 \end{array}\right). \tag{14}$$

Step 6: State before the update. In (13), we have identified which vertex will be moved to the L partition. Substituting these new submatrices into the loop invariant and recognizing that the trace is invariant to rotation, we obtain the loop invariant in terms of these new partitions.

$$\begin{aligned}\bowtie_G =& \frac{1}{4}\Gamma(A_0A_0^TA_0A_0^T - A_0A_0^T \circ A_0A_0^T - JA_0^TA_0 + A_0A_0^T) \\ &+ \frac{1}{2}\Gamma\left(A_0A_0^Ta_1a_1^T + A_0A_0^TA_2A_2^T \right. \\ &\left. - A_0A_0^T \circ a_1a_1^T - A_0A_0^T \circ A_2A_2^T\right)\end{aligned} \tag{15}$$

Step 7: State after the update. Similarly, substituting (13) in the loop invariant, we get the state of the loop invariant after the update is done.

$$\begin{aligned}\bowtie_G =& \frac{1}{4}\Gamma(A_0A_0^TA_0A_0^T - A_0A_0^T \circ A_0A_0^T - JA_0^TA_0 + A_0A_0^T) \\ &+ \frac{1}{2}\Gamma\left(A_0A_0^Ta_1a_1^T + A_0A_0^TA_2A_2^T \right. \\ &\left. - A_0A_0^T \circ a_1a_1^T - A_0A_0^T \circ A_2A_2^T\right) \\ &+ \frac{1}{4}\Gamma(a_1a_1^Ta_1a_1^T - a_1a_1^T \circ a_1a_1^T - Ja_1a_1^T + a_1a_1^T) \\ &+ \frac{1}{2}\Gamma(a_1a_1^TA_2A_2^T) - \frac{1}{2}\Gamma(a_1a_1^T \circ A_2A_2^T)\end{aligned} \tag{16}$$

Step 8: Determine the update. Comparing (15) and (16), we can obtain the update statement that must be performed to maintain the loop invariant:

$$\begin{aligned}\bowtie_G :=& \bowtie_G + \frac{1}{4}\Gamma(a_1a_1^Ta_1a_1^T - a_1a_1^T \circ a_1a_1^T - Ja_1a_1^T + a_1a_1^T) \\ &+ \frac{1}{2}\Gamma(a_1a_1^TA_2A_2^T) - \frac{1}{2}\Gamma(a_1a_1^T \circ A_2A_2^T)\end{aligned} \tag{17}$$

Since, a_1 represents the neighborhood of vertex $v_{2k} \in V_2$, the term $\Gamma(a_1a_1^Ta_1a_1^T - a_1a_1^T \circ a_1a_1^T - Ja_1a_1^T + a_1a_1^T)$ would represent the butterflies with only $v_{2k} \in V_2$ as a wedge point. Since, it is not possible to create a butterfly with only one vertex in V_2, this term becomes zero. In next paragraph we explain how.

Since, a_1 represents the neighborhood of vertex $v_{2k} \in V_2$, the (i,j)-th entry of $a_1a_1^T$ represents the number of paths of length 2 that start at vertex $v_{1j} \in V_1$, and pass through $v_{2k} \in V_2$ and end at vertex $v_{1j} \in V_1$. Therefore, the (i,j)-th entry of $a_1a_1^Ta_1a_1^T$ represents the number of paths of length 4 that start at vertex $v_{1j} \in V_1$ and end at vertex $v_{1j} \in V_1$, and and pass through $v_{2k} \in V_2$ twice. Similarly, the (i,j)-th entry of $(a_1a_1^T \circ a_1a_1^T)$ represents a line of length 4 going through v_{2k} twice. $Ja_1a_1^T - a_1a_1^T$ represents the number of wedges that have have v_{2k} as a wedge point. Therefore, taking the trace of this quantity results in 0.

Now, recognizing that the trace is invariant under rotation, the update statement can be simplified as

$$\bowtie_G := \frac{1}{2}a_1^TA_2A_2^Ta_1 - \frac{1}{2}\Gamma(a_1a_1^T \circ A_2A_2^T) + \bowtie_G \tag{18}$$

Within the new update statement, the terms also coincide to different structures of wedges. The term $\frac{1}{2}a_1^TA_2A_2^Ta_1$ represents paths of length 4 with v_{2k} and another distinct vertex in the L partition of V_2. Along with butterflies, this term also includes paths of length 4 that are formed with 2 lines, which is captured by the $\frac{1}{2}\Gamma(a_1a_1^T \circ A_2A_2^T)$. By carefully implementing this update, the computation of the subtraction term can be avoided.

D. Deriving the remaining algorithms

Following the steps in the previous section, algorithms corresponding to the invariants listed in Fig. 4 and Fig. 5 can be derived. For completeness, these algorithms have been listed in Fig. 6 and Fig. 7.

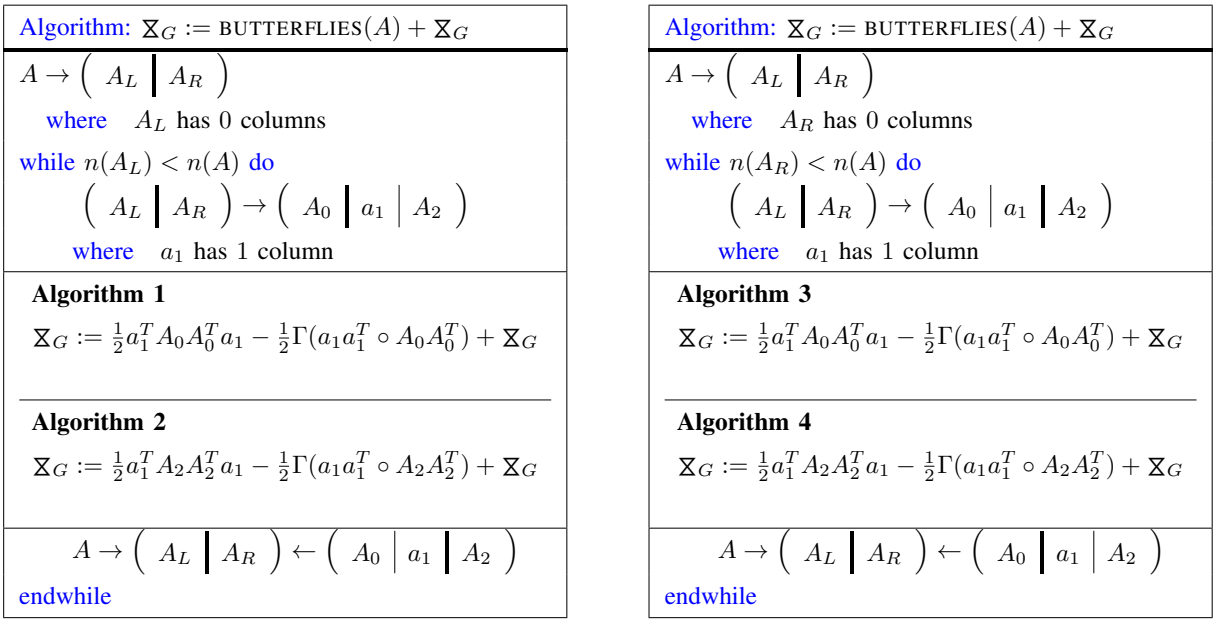

Fig. 6: The resulting butterfly counting algorithms when vertex set V_2 is partitioned. The algorithms on the left represent algorithms obtained when the vertex set V_2 is traversed from the L to R partitioning, while the algorithms on the right represent algorithms when traversing from R to L.

IV. Beyond Butterfly Counting

Beyond counting the number of butterflies in a bipartite graphs, the k-tip and k-wings decomposition for bipartite graphs are often of interest. These two decompositions are analogous to the the k-core and k-truss statistics for unipartite graphs. In this section, we discuss how the above formulation for butterfly counting can be adapted to obtain the k-tip and k-wings decomposition.

A. Definitions of k-tips and k-wings

Given a bipartite graph $G = (V_1, V_2, E)$, a maximal subgraph H induced from G is k-tip, if every vertex in H is part of at least k butterflies. Similarly, if every edge of a maximal subgraph H induced from G is part of at least k butterflies, then H is known as a k-wings. The $k-$tip and k-wings decomposition of a bipartite graph was first introduced in [11].

B. Formulation of k-tip.

For the k-tip decomposition of G, we are interested in the subgraph of G where each vertex contributes to at least k butterflies. To obtain a k-tip subgraph, we must first compute the number of butterflies each vertex contributes, then remove the vertices that contribute to fewer than k butterflies. When the vertices that do not contribute to at least k butterflies are removed, it is possible that the number of butterflies at the remaining vertices become less than k. Hence, we must repeat this process till no vertices are removed. The resulting subgraph is the corresponding k-tip decomposition of G. Since this is an iterative process, we rename the adjacency matrix of G to A_0.

From (7), one recognizes that the diagonal elements of $\frac{1}{4}\left(A_0 A_0^T A_0 A_0^T - A_0 A_0^T \circ A_0 A_0^T - J A_0 A_0^T + A_0 A_0^T\right)$ gives us the number of butterflies at each vertex in V_1. Therefore,

$$s = \frac{1}{4}\text{DIAG}(A_0 A_0^T A_0 A_0^T - A_0 A_0^T \circ A_0 A^T - J A_0 A_0^T + A_0 A_0^T), \quad (19)$$

where $\text{DIAG}(X)$ creates a vector of the diagonal elements of X, gives us the number of butterflies at each vertex in V_1. Hence,

$$m = s \geq k, \quad (20)$$

creates a mask for the vertices in V_1, and $m^T A$ gives us the mask for the vertex set V_2. Now,

$$M = m m^T A_0 \quad (21)$$

$$A_1 = A_0 \circ M. \quad (22)$$

This process of (19)–(22) is repeated till no more vertices have been removed. At this stage, the subgraph H described by the adjacency matrix A_i is the k-tip decomposition of G.

Following the steps in Section III, we can derive the loop invariants as well as algorithms for peeling k-tips. For illustration, we present one algorithm of the many that can be derived. We recognize that by partitioning the adjacency matrix A_0 as follows:

$$A_0 \rightarrow \left(\begin{array}{c} A_T \\ \hline A_B \end{array} \right),$$

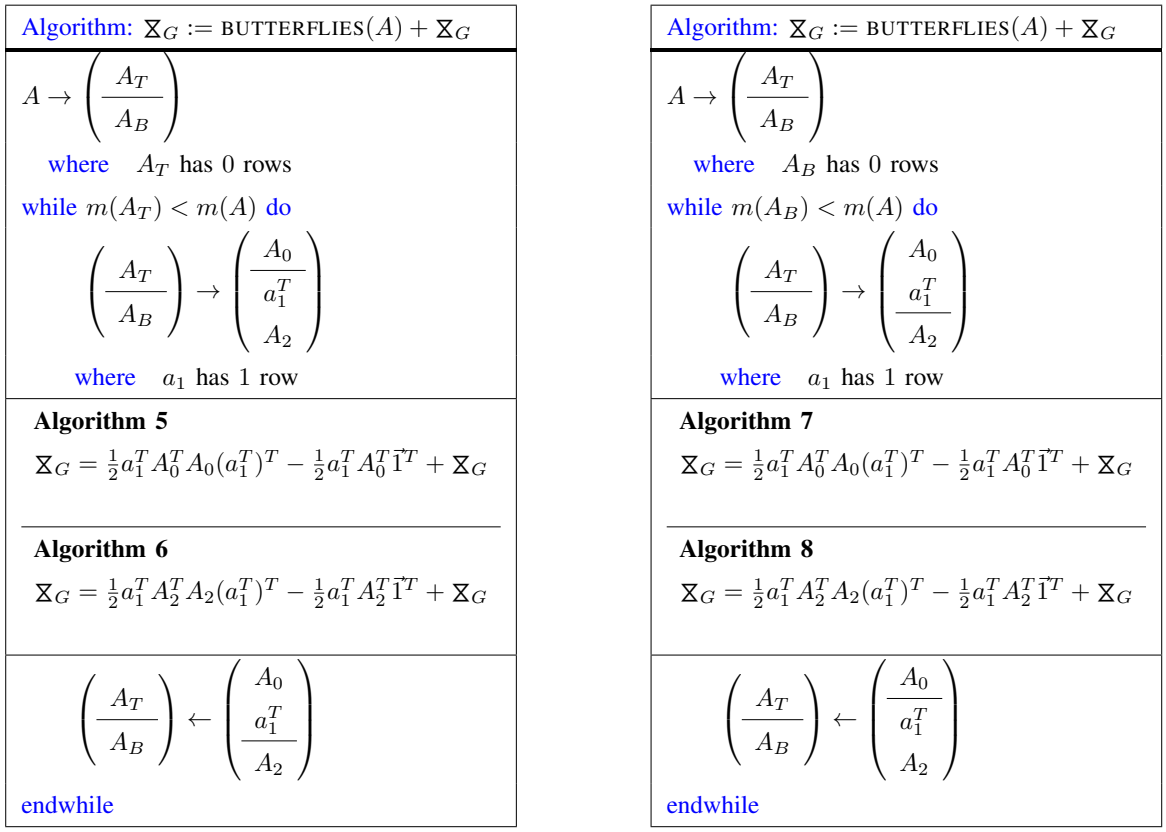

Fig. 7: The resulting butterfly counting algorithms when vertex set V_1 is partitioned. The algorithms on the left represent algorithms obtained when the vertex set V_1 is traversed from the T to B partitioning, while the algorithms on the right represent algorithms when traversing from B to T.

the s must also be partitioned in a similar fashion so that the dimensions of the matrix operations conform. Such a partitioning allows us to derive a "look-ahead" algorithm in which the s_T is fully computed, and s_B is partially updated. Since s_T is fully computed, we can compute the mask m for V_1 using (20) as soon as we finish computing a part of s vector. Once the mask is computed, we can compute the new adjacency matrix A_1 using (22). This process is repeated till no vertices are removed. Fig. 8 illustrates one such iteration of the computation of (19)–(20).

C. Formulation of k-wing.

Every edge of the k-wing decomposition of G is part of at least k butterflies. Similar to the k-truss decomposition of unipartite graphs [7], to obtain the k-wings of G is a two step process. First the number of butterflies each edge contributes to (or support of each edge) is computed, and then the edges with insufficient butterflies are removed. This process is repeated till no edges are removed, or all the edges of the graph have been removed. Similar to the k-tip formulation, this is an integrative process, so we rename the adjacency matrix of G to A_0

Computing the support of each edge in G:
Here, we first focus on finding the support of an arbitrary edge $e = (u, v)$ where $u \in V_1$ and $v \in V_2$. To find the number of butterflies that contain the edge e, we must first find the number of wedges that contain edge e. Without the loss of generality, we will consider the wedges that start and end in V_1. In other words, we will consider wedges where v is the wedge point. If we can find the number of neighbors of v that are distinct from u, we can compute the number of wedges that contain edge e. Suppose the neighborhood of v is given by $N(v)$. Therefore, the number of wedges that contain e is $\|N(v)\| - 1$, where $\|X\|$ is the cardinality of set X.

Of these $\|N(v)\| - 1$ wedges with end vertices u and w, where $w \in V_1 - \{u\}$ and $w \in N(v) - \{u\}$, we need to find the number of distinct wedges with the u and w as end points that do not have v as the wedge point. The number of these new wedges is given by $\|N(u) \cap N(w)\| - 1$. These new wedges, together with the $\|N(v)\| - 1$ wedges that have v as a wedge point will form unique butterflies containing the edge (u, v).

Hence, by combining the two sets of wedges described above, the number of butterflies containing the edge (u, v) is given by

Algorithm: $[s, m] := \text{KTIP\_UNB\_VAR1}(A_0)$

$$A_0 \rightarrow \left(\frac{A_{0_T}}{A_{0_B}}\right), \; s \rightarrow \left(\frac{s_T}{s_B}\right), \; m \rightarrow \left(\frac{m_T}{m_B}\right)$$

where A_{0_T}, s_T, m_T has 0 rows.

while $m(A_{0_T}) < m(A_0)$ do

$$\left(\frac{A_{0_T}}{A_{0_B}}\right) \rightarrow \left(\frac{A_{0_0}}{a_{0_1}^T}{A_{0_2}}\right), \left(\frac{s_T}{s_B}\right) \rightarrow \left(\frac{s_0}{\sigma_1}{s_2}\right), \left(\frac{m_T}{m_B}\right) \rightarrow \left(\frac{m_0}{\mu_1}{m_2}\right)$$

where $a_{0_1}^T$ has 1 row, σ_1, μ_1 is a scalar

$\sigma_1 := \frac{1}{4}(a_1^T A_2^T A_2 (a_1^T)^T - \vec{1}^T (a_1^T)^T A_2 + \sigma_1$

$s_2 := \frac{1}{4}(A_2 (a_1^T)^T a_1^T A_2^T - \vec{1} a_1^T A_2$

$\mu_1 := \sigma_1 \geq k$

$$\left(\frac{A_{0_T}}{A_{0_B}}\right) \leftarrow \left(\frac{A_{0_0}}{a_1^T}{A_2}\right), \left(\frac{s_T}{s_B}\right) \leftarrow \left(\frac{s_0}{\sigma_1}{s_2}\right), \left(\frac{m_T}{m_B}\right) \leftarrow \left(\frac{m_0}{\mu_1}{m_2}\right)$$

endwhile

Fig. 8: The resulting algorithm for k-tip peeling is derived for when the adjacency matrix is partitioned into a top and bottom partitioning. Here, while computing the number of butterflies each vertex is a part of, the mask can be simultaneously computed.

$$\begin{aligned}
&\sum_{w \in N(v) - \{u\}} (\|N(u) \cap N(w)\| - 1) \\
&= \left(\sum_{w \in N(v) - \{u\}} \|N(u) \cap N(w)\| \right) - (\|N(v)\| - 1) \\
&= \left(\sum_{w \in N(v)} \|N(u) \cap N(w)\| - \|N(u) \cap N(u)\| \right) - (\|N(v)\| - 1) \\
&= \sum_{w \in N(v)} \|N(u) \cap N(w)\| - \|N(u)\| - \|N(v)\| + 1, \qquad (23)
\end{aligned}$$

where $\sum_{w \in N(v)} \|N(u) \cap N(w)\|$ represents the number of wedges between vertices u and all the neighbors w of v in V_1.

We now discuss how to turn the above insight into a linear algebraic expression. Let e_i be a column vector of all zeros, and the i^{th} position is set to 1. To find the neighbors of a vertex $u \in V_1$, we need to perform $e_u^T A_0$, while to find the neighbor of vertex $v \in V_2$, the computation that must be performed is $A e_v$. Furthermore, we know that the neighborhood of v can be decomposed as

$$N(v) = \sum_{w \in N(v)} e_w.$$

The number of wedges between $u, w \in V_1$ is given by $e_u^T A_0 A_0^T e_w$. Furthermore, the sum of all wedges where $w \in N(v)$ is given by

$$\begin{aligned}
&e_u^T A_0 (A_0^T e_{w_0} + A_0^T e_{w_1} + ...A_0^T e_{w_{N(v)-1}}) \\
=&e_u^T A_0 A_0^T N(v) = e_u^T A_0 A_0^T A_0 e_v.
\end{aligned}$$

Substituting the above and the definition of neighborhoods into (23), the number of butterflies involving the original edge (u, v) is

$$e_u^T A_0 A_0^T A_0 e_v - e_u^T A_0 A_0^T e_u - e_v^T A_0^T A_0 e_v + 1. \qquad (24)$$

The next step towards generalizing (24) is to consider the support of all the edges connected to u. Depending on which neighbor of u we are considering, (24) remains the same but the vector e_v changes. To generalize this, instead of considering the support of all edges connected to u, we can assume that u is connected to all vertices vertices $v \in V_2$, compute the support, and then mask out the edges that do not exist between u and the vertices in V_2. Thus, we get the following equation that gives us the support of u at each vertex in V_2:

$$(e_u^T A_0 A_0^T A_0 + \vec{1}_{\|V_2\|}^T (1 - e_u^T A_0 A_0^T e_u) - diag(A_0^T A_0)^T) \circ e_u^T A_0,$$

where $\vec{1}_s$ is a s-dimensional vector of all ones.

Finally, since we have to compute this for all vertices in V_1, we can create a matrix S_w

$$\begin{aligned}
S_w = \Big(&A_0 A_0^T A_0 - diag(A_0 A_0^T) \vec{1}_{\|V_2\|}^T \\
&- \vec{1}_{\|V_1\|} diag(A_0^T A_0)^T + J \Big) \circ A_0, \qquad (25)
\end{aligned}$$

where S_w is a $m \times n$ matrix, and the (i, j)-th entry of S_w represents the support of that edge, or in other words, the number of butterflies that consist of that edge.

Removing the edges with insufficient support:
Now that we have the support of each edge given by (25), we can eliminate the edges with insufficient support. This can be done by creating a mask M, and then applying the mask to the adjacency matrix A_0.

$$M = S_w \geq k \qquad (26)$$

$$A_1 = A_0 \circ M. \qquad (27)$$

As with k-tips, here we iterate through computing (25)–(27) till we cannot remove any edges, or all edges have been removed. The resulting adjacency matrix at this stage defines the k-wing subgraph. Following similar steps as shown in Section III, algorithms for peeling k-wings can be derived.

V. Validation

Although the primary contribution of this paper is focused upon the derivation of correct linear algebra algorithms and extensions to other graph algorithms, it is important to show how the properties of these algorithms can affect performance. Similarly, preliminary results will also provide insight into how future graph algorithms can exploit these properties to yield better performance.

For reference, the results of the C implementation of these graph algorithms were obtained on an Intel(R) Core(TM) i7-8750H CPU with a total of 6 cores. The dataset used to obtain these results is from the KONECT [5] database. Statistics provided by the Fig. 9 is also obtained from the KONECT database.

Dataset Name	$\|V_1\|$	$\|V_2\|$	$\|E\|$	$\bowtie_G$
`arXiv cond-mat`	16,726	22,015	58,595	70,549
`Producers`	48,833	138,844	207,268	266,983
`Record Labels`	168,337	18,421	233,286	1,086,886
`Occupations`	127,577	101,730	250,945	24,509,245
`GitHub`	56,519	120,867	440,237	50,894,505

Fig. 9: The following are table contains the name listed under the KONECT dataset, as well as the sizes of the vertex sets V_1 and V_2, the number of edges $|E|$, and the number of butterflies $\bowtie_G$.

To begin, each invariant was run using the datasets shown in Fig. 9. For storage, invariants 1-4 stored the graph in memory in CSC format, while invariants 5-8 stored in CSR format. This is because, for invariants 1-4, each iteration of the loop exposes a column of the matrix, making CSC the favorable format to access adjacent column elements. Similarly, invariants 5-8 expose a row of the matrix at each iteration, making CSR favorable for accessing adjacent row elements.

After implementing both the CSR and CSC formats, we utilized the butterfly count statistic within the KONECT dataset as shown in Fig. 9 to check the correctness of each implementation. To note, the KONECT database labels the butterfly count as the square count. However, since squares are isomorphic to the butterfly structure present in bipartite graphs, counting butterflies will be equivalent to counting squares for bipartite graphs.

Here we consider two properties of the graph—the partition size, which is the size of the bipartitions V_1 and V_2 and edge sparsity, which is the relative sparsity of the adjacency matrix. From Fig. 10 and Fig. 11, performance from each of the invariants is affected by these two graph properties.

The datasets `Record Labels` and `Occupations`, the partition sizes are such that $|V_1| < |V_2|$, while for the others have $|V_1| > |V_2|$. Reflecting upon results shown in Fig. 10 and Fig. 11, invariants 1-4 tended to perform better than invariants 5-8 on datasets when $|V_1| < |V_2|$, while invariants 5-8 achieved better performance on the other datasets. Recall that invariants 1-4 partition the vertex set V_2, while invariants 5-8 partition the vertex set V_1. Therefore, based on the dataset, an algorithm should be picked that partitions the smaller of the two vertex sets.

Regarding edge sparsity, the graph algorithms tended to have better performance with sparse graphs than dense graphs. This pattern can be observed when comparing the `GitHub` dataset with the `Producers` dataset. Both the `GitHub` and `Producers` dataset have roughly the same number of vertices and equivalent partition sizes. However, the `Producers` dataset has about half the number of nodes as the `GitHub` dataset. As a result, some instances of the algorithm performance yielded a slow down as much as two times for the same invariant.

In addition, it is worth observing that invariants 2 and 4 for the left-right partitioning and invariants 5 and 7 for the top-bottom partitioning tended to perform better than invariants 1 and 3 and invariants 6 and 8 respectively. An important connection between these algorithms is to notice that invariants 2 and 7 can both be considered "look-ahead" algorithms because these algorithms use parts of the adjacency matrix that will be exposed in future iterations.

VI. Conclusion and Future Work

In this paper, we derived a family of proven correct algorithms for computing various graph metrics related to counting butterflies motifs in a bipartite graph. Fundamental to our approach is the identification of multiple loop invariants, a formal methods concept that is used to prove the correctness of loop based algorithms, from a single linear algebraic specification for counting butterflies. Each loop invariant is then used to derive a different algorithm for counting butterflies. We also show that the specification for counting butterflies can easily be updated to specify and derive loop invariants for other metrics associated with butterfly counting such as k-tip and k-wing.

While the multitude of algorithms for different metrics for bipartitie graph is the focus of this work, we nonetheless show preliminary sequential and parallel performance numbers to demonstrate that the approach yields algorithms that can be easily parallelized. In follow-on work, we believe that optimizations such as sorting by vertex degrees [3], [12], and fine-grain parallelization [2] can be applied to the algorithms presented in this work to get even better performance. This is something we look forward to pursuing.

VII. Acknowledgements

We thank members of the Science of High-Performance Computing (SHPC) group for their encouragement and feedback. This research was sponsored in part by the National Science Foundation (Award CSSI-2003921). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

Dataset	Inv. 1	Inv. 2	Inv. 3	Inv. 4	Inv. 5	Inv. 6	Inv. 7	Inv. 8
arXiv cond-mat	2.926	1.793	2.896	1.742	1.828	1.465	1.862	1.470
Producers	73.485	71.140	74.192	71.140	18.349	11.190	18.703	10.742
Record Labels	4.865	2.966	4.913	3.004	106.314	98.978	105.978	98.619
Occupations	26.190	17.695	26.092	17.571	92.398	96.111	94.003	97.914
GitHub	104.069	81.841	103.197	81.899	49.493	40.111	48.505	40.290

Fig. 10: Timing data for each unblocked implementation of each invariant given a dataset. All times displayed in the chart are in seconds.

Dataset Name	Inv. 1	Inv. 2	Inv. 3	Inv. 4	Inv. 5	Inv. 6	Inv. 7	Inv. 8
arXiv cond-mat	0.525	0.430	0.573	0.393	0.309	0.355	0.293	0.292
Producers	16.269	15.069	16.276	15.753	3.207	4.512	3.174	4.470
Record Labels	0.732	1.923	0.808	1.910	23.802	24.774	22.389	25.145
Occupations	5.235	8.126	5.673	8.082	18.223	20.215	18.854	19.894
GitHub	18.473	32.111	17.072	30.799	6.646	15.772	6.431	15.479

Fig. 11: Timing data for each parallel (6 threads) implementation of each invariant given a dataset. All times displayed in the chart are in seconds.

REFERENCES

[1] BIENTINESI, P., GUNNELS, J. A., MYERS, M. E., QUINTANA-ORTÍ, E. S., AND VAN DE GEIJN, R. A. The Science of Deriving Dense Linear Algebra Algorithms. *ACM Trans. Math. Softw. 31*, 1 (Mar 2005), 1–26.

[2] BLANCO, M., LOW, T. M., AND KIM, K. Exploration of Fine-Grained Parallelism for Load Balancing Eager K-truss on GPU and CPU. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)* (2019), pp. 1–7.

[3] BLANCO, M. P., MCMILLAN, S., AND LOW, T. M. Towards an Objective Metric for the Performance of Exact Triangle Count. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)* (2020), pp. 1–7.

[4] COHEN, J. Trusses: Cohesive Subgraphs for Social Network Analysis. *National security agency technical report 16*, 3.1 (2008).

[5] KUNEGIS, J. KONECT – The Koblenz Network Collection. In *Proc. Int. Conf. on World Wide Web Companion* (2013), pp. 1343–1350.

[6] LEE, M., AND LOW, T. M. A Family of Provably Correct Algorithms for Exact Triangle Counting. In *Proceedings of the First International Workshop on Software Correctness for HPC Applications* (New York, NY, USA, 2017), Correctness'17, Association for Computing Machinery, p. 14–20.

[7] LOW, T. M., SPAMPINATO, D. G., KUTULURU, A., SRIDHAR, U., POPOVICI, D. T., FRANCHETTI, F., AND MCMILLAN, S. Linear Algebraic Formulation of Edge-centric K-truss Algorithms with Adjacency Matrices. In *2018 IEEE High Performance extreme Computing Conference (HPEC)* (2018), pp. 1–7.

[8] LUCE, R. D., AND PERRY, A. D. A Method of Matrix Analysis of Group Structure. *Psychometrika 14*, 2 (Jun 1949), 95–116.

[9] MYERS, M. E., AND VAN DE GEIJN, R. A. LAFF-on Programming for Correctness. https://www.edx.org/course/laff-programming-correctness-utaustinx-ut-p4c-14-01x, 2017. Massive Open Online Course on edX.

[10] SANEI-MEHRI, S.-V., SARIYUCE, A. E., AND TIRTHAPURA, S. Butterfly Counting in Bipartite Networks. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (New York, NY, USA, 2018), KDD '18, Association for Computing Machinery, p. 2150–2159.

[11] SARIYÜCE, A. E., AND PINAR, A. Peeling Bipartite Networks for Dense Subgraph Discovery. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining* (New York, NY, USA, 2018), WSDM '18, Association for Computing Machinery, p. 504–512.

[12] SHI, J., AND SHUN, J. Parallel Algorithms for Butterfly Computations. *CoRR abs/1907.08607* (2019).

[13] STYAN, G. P. Hadamard Products and Multivariate Statistical Analysis. *Linear Algebra and its Applications 6* (1973), 217–240.

[14] WANG, J., FU, A. W.-C., AND CHENG, J. Rectangle Counting in Large Bipartite Graphs. In *2014 IEEE International Congress on Big Data* (2014), pp. 17–24.

[15] WANG, K., LIN, X., QIN, L., ZHANG, W., AND ZHANG, Y. Vertex Priority Based Butterfly Counting for Large-Scale Bipartite Networks. *Proc. VLDB Endow. 12*, 10 (jun 2019), 1139–1152.