Efficient Bi-triangle Counting for Large Bipartite Networks

Yixing Yang
The University of New South Wales
Australia
yixing.yang@student.unsw.edu.au

Yixiang Fang*
The University of New South Wales
Australia
yixiang.fang@unsw.edu.au

Maria E. Orlowska
Polish-Japanese Institute of
Information Technology, Poland
omaria@pjwstk.edu.pl

Wenjie Zhang
The University of New South Wales
Australia
wenjie.zhang@unsw.edu.au

Xuemin Lin
The University of New South Wales
Australia
lxue@cse.unsw.edu.au

ABSTRACT

A bipartite network is a network with two disjoint vertex sets and its edges only exist between vertices from different sets. It has received much interest since it can be used to model the relationship between two different sets of objects in many applications (e.g., the relationship between users and items in E-commerce). In this paper, we study the problem of efficient bi-triangle counting for a large bipartite network, where a bi-triangle is a cycle with three vertices from one vertex set and three vertices from another vertex set. Counting bi-triangles has found many real applications such as computing the transitivity coefficient and clustering coefficient for bipartite networks. To enable efficient bi-triangle counting, we first develop a baseline algorithm relying on the observation that each bi-triangle can be considered as the join of three wedges. Then, we propose a more sophisticated algorithm which regards a bi-triangle as the join of two super-wedges, where a wedge is a path with two edges while a super-wedge is a path with three edges. We further optimize the algorithm by ranking vertices according to their degrees. We have performed extensive experiments on both real and synthetic bipartite networks, where the largest one contains more than one billion edges, and the results show that the proposed solutions are up to five orders of magnitude faster than the baseline method.

PVLDB Reference Format:
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.
doi:10.14778/3447689.3447702

1 INTRODUCTION

A bipartite network is a network with two disjoint vertex sets and its edges only exist between vertices from different sets. It has received much interest because in many applications, it can be used to model the relationship between two different sets of objects. For example, in E-commerce, the purchase relationship between users and items can be modeled as a bipartite network, in which users as a whole form one vertex set and items as a whole form another, while each edge means that a user purchases an item. Figure 1 depicts such a network with four users (i.e., $u_1, \cdots, u_4$) and four items (i.e., $l_1, \cdots, l_4$). Moreover, in coding theory [55], bipartite networks are used to represent the interactions between codewords. In graph theory, hypergraphs [15] are usually represented as bipartite networks.

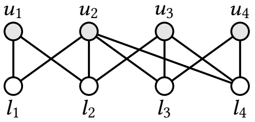
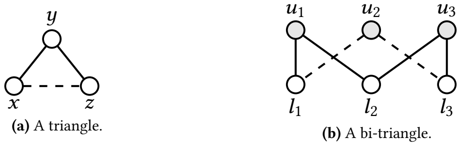


Figure 1: An example of bipartite networks.



(a) A triangle.

(b) A bi-triangle.

Figure 2: Formation of triangles and bi-triangles (Note that dashed lines denote the additional edge and additional wedge in (a) and (b), respectively).

In classic unipartite networks, the triangle is one of the most fundamental structures, which is formed by a wedge closed by an additional edge (see Figure 2(a)), where a wedge is a path with two edges. Triangles play important roles in various network analysis tasks, such as computing clustering coefficient [36], thematic structures analysis [9], social relationships analysis [11, 12, 22, 35, 50], triangle-based community computation [10, 20, 21, 23], and so on. However, when analyzing a bipartite network, these measures and algorithms are not applicable, because there is no triangle in a bipartite graph. To fill this gap, Opsahl et al. proposed the structure of *bi-triangle* [37], which is defined as a 6-cycle, or a cycle with three vertices from one vertex set and three vertices from another vertex set. Essentially, a bi-triangle is formed by two connected wedges closed by an additional wedge, serving as the counterpart of the classic triangle in bipartite networks. For example, Figure 2(b) presents a bi-triangle in the bipartite network of Figure 1.

In this paper, we study the problem of bi-triangle counting, that is to compute the number of bi-triangles in a bipartite network.

*Yixiang Fang is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 14, No. 6 ISSN 2150-8097.
doi:10.14778/3447689.3447702

The importance of bi-triangle counting has been demonstrated by many graph analysis tasks. Below are some typical examples:

(1) Transitivity coefficient: The transitivity coefficient (TC) [47] evaluates how strong the vertices in a network are aggregated. In [37], Opsahl extended this measurement for bipartite networks; that is, given a bipartite network G, with two disjoint sets of vertices U and L, its TC for its vertices set L is defined as $TC(G, L) = \frac{3 \times total\ number\ of\ bi\text{-}triangles}{number\ of\ 4\text{-}paths\ centered\ in\ L}$, where a 4-path is a path with four edges in G (for the other set of vertices U, $TC(G, U)$ is defined similarly). As the number of 4-paths can be efficiently obtained in $O(|V|+\alpha|E|)$ time [7], where V, E, and α (α is always much less than $\sqrt{|E|}$) are the vertex set, edge set, and the arboricity of G respectively, the efficiency of computing the TC largely depends on how efficiently we count bi-triangles. The TC is useful for measuring the quality (cohesiveness) of communities in bipartite networks, which are very useful for various tasks in e-commerce and social networks. We will demonstrate its usefulness in such applications by presenting a case study in Section 7.

(2) Clustering coefficient: The clustering coefficient (CC) [36, 37] measures the ability of a vertex to cluster its neighbors. More precisely, given a bipartite network G, the CC of a vertex $v \in G$ is the ratio of the number of bi-triangles containing v over the number of paths with four edges that center at v, i.e., $CC(G, v) = \frac{number\ of\ bi\text{-}triangles\ containing\ v}{number\ of\ 4\text{-}paths\ centered\ at\ v}$. Clearly, the efficiency of bi-triangle counting has a great effect on the computation of the CC.

(3) Truss computation over bipartite networks: Recently, Yang et al. have proposed a novel k-truss model over heterogeneous information networks (HINs) [57]. When the HIN is simply a bipartite network, the model is then based on bi-triangles, thus a key step of truss computation is to efficiently obtain the number of bi-triangles.

Despite the wide usefulness of bi-triangle counting, little research attention has been paid to this topic yet. We would like to note that in the literature, there is a group of highly related works [41, 43, 51], called butterfly counting on bipartite networks, where a butterfly is a 4-cycle, or a cycle with two vertices from one vertex set and two vertices from another vertex set. However, bi-triangles are inherently different from butterflies, because a butterfly can only capture the relationship between two vertices in each vertex set, rather than three vertices like what the bi-triangle does. More specifically, from the perspective of an important cohesiveness, the bi-triangle based TC [37] measures the cohesiveness of a set of vertices from a single side only (e.g., either users or items in user-item networks), while the butterfly based TC [42] measures the cohesiveness of a whole bipartite network with two sides. Thus, the bi-triangle based TC is more useful when measuring the cohesiveness of vertices from a single side, and we will show this by a case study in Appendix [58]. In addition, some key measures on unipartite networks, such as local clustering coefficient which is the ratio of the number of triangles containing a vertex over the number of "open-triangles" (or paths with two edges) that center at this vertex, cannot be extended for bipartite networks based on butterflies, since an "open-butterfly" (or a path with three edges) does not have a center vertex.

Challenges. Technically, the problem of bi-triangle counting is very challenging because the number of bi-triangles in a bipartite network is often much larger than the size of the network. Let U and L denote the two disjoint sets of vertices in the bipartite network respectively. If the sizes of U and L are n, then the number of bi-triangles could be up to $O(n^6)$, since a bi-triangle is a cycle with six vertices. Besides, our problem is more computationally expensive than many other related counting problems such as butterfly counting [1, 39, 41], due to the same reason. For example, the bipartite network *Trackers*, which contains around 4.0×10^7 vertices and 1.4×10^8 edges, has only 2.01×10^{13} butterflies, but 4.92×10^{28} bi-triangles. Hence, it is desirable to develop efficient solutions for counting bi-triangles on large-scale bipartite networks.

Our contributions. As shown in the literature [1, 39, 41], to count the number of instances of a relatively complicated pattern P in a network, a general fundamental idea is to split P into several small patterns, then count the numbers of these small patterns, and finally derive the total number of P based on the numbers of small patterns. Note that counting the number of P is often cheaper than enumerating P. More specifically, we first derive a formula for counting the number of bi-triangles over three vertices by simply checking how many wedges connecting them. Then, by aggregating the number of bi-triangles for each triplet of vertices, the total number of bi-triangles can be derived. Since this algorithm is based on "*wedge join*", we call it WJ-Count.

Although WJ-Count is straightforward, it is inefficient for large bipartite networks, since it needs to enumerate an extremely large number of vertex triplets, i.e., if each vertex set of the bipartite network contains n vertices, then the total number of triplets could be up to n^3, which greatly limits its efficiency on large bipartite networks, as we will explain in Sections 3 and 7. To alleviate this issue, we propose another counting algorithm based on the key observation that a bi-triangle can be considered as the join of two super-wedges. Here, a super-wedge is a simple path with three edges, or two connected wedges that share an edge. The main advantage of this algorithm over WJ-Count is that it allows us to avoid enumerating the excessively large number of triplets of vertices that inherently narrows WJ-Count's optimization potential. However, joining two super-wedges to generate bi-triangles may result in complicated invalid cases. To solve this challenge, we propose a two-stage counting strategy where the first stage counts both the valid and invalid cases and the second stage counts the invalid cases, respectively. We will discuss the technical details in Section 4. We denote this algorithm by SWJ-Count, since it is based on "*super-wedge join*".

We further optimize SWJ-Count by ranking vertices according to their degrees. The intuition is that vertices with higher degrees tend to be involved in more bi-triangles, which implies that if we perform counting for vertices with higher degrees priorly, we may count more bi-triangles together, allowing us to achieve higher efficiency. Compared with SWJ-Count, the ranking mechanism may result in even complicated invalid cases. To tackle this challenge, we introduce the novel concept of the anchor-vertex, and then by using the anchor-vertex, we can divide the invalid cases into four types such that each type can be counted easily during the process of counting all the valid cases. We denote the optimized algorithm by RSWJ-Count, as it incorporates a ranking mechanism. It is worthwhile mentioning that the three counting algorithms above can be easily parallelized to achieve better efficiency.

In addition, we study two variants of the bi-triangle counting problem, also called local bi-triangle counting, which aim to count the numbers of bi-triangles that contain a specific vertex or edge, respectively. These variants are useful in some real applications, such as computing the local clustering coefficient. We also develop two efficient counting algorithms for solving these two variants.

We have performed extensive experiments on both real and synthetic datasets, such as the efficiency and scalability test. The experimental results show that `RSWJ-Count` and `SWJ-Count` are much faster than the baseline method `WJ-Count`. Particularly, `RSWJ-Count` can process billion-scale networks while `WJ-Count` and `SWJ-Count` can merely handle small and million-scale networks, respectively.

In summary, our principal contributions are as follows:

- We develop a baseline algorithm `WJ-Count` which counts the number of bi-triangles by joining wedges.
- We devise a faster algorithm `SWJ-Count` that represents a bi-triangle as the join result of two super-wedges.
- We further optimize `SWJ-Count` by introducing a ranking mechanism and get an optimized algorithm `RSWJ-Count`.
- We study the problems of local bi-triangle counting, and propose efficient counting algorithms.
- We conduct an extensive experimental evaluation and the results show that our proposed counting algorithms are efficient and `RSWJ-Count` is the fastest one.

Outline. We formalize the problem definition in Section 2. Section 3 presents our baseline algorithm `WJ-Count`. Section 4 presents `SWJ-Count`. Section 5 presents the best algorithm `RSWJ-Count`. Section 6 presents the problems and solutions of local bi-triangle counting. The experimental results are reported in Section 7. We review the related work in Section 8 and conclude in Section 9.

2 PROBLEM DEFINITION

DEFINITION 1. ***Bipartite network [37]:*** *A bipartite network is an undirected network $G = (V{=}(U \cup L), E)$, where V is the vertex set and E is the edge set s.t. (1) V consists of two disjoint sets, i.e., the set of vertices in the upper layer U and the set of vertices in the lower layer L, where $V = U \cup L$ and $U \cap L = \emptyset$; and (2) each edge $e{=}(u, l) \in E$ connects a vertex $u \in U$ and a vertex $l \in L$, i.e., $E \subseteq U \times L$. (Please note that in this paper, all the bipartite networks are unweighted).*

DEFINITION 2. ***Bi-triangle [37]:*** *Given a bipartite network $G = (V{=}(U \cup L), E)$, a bi-triangle is a 6-cycle, or a cycle with a length of 6 having three vertices in U and three vertices in L. Besides, we use v_1-v_2-v_3-v_4-v_5-v_6-v_1 to denote the bi-triangle composed of vertices $v_1, v_2, \cdots, v_6$ and edges $(v_1, v_2), (v_2, v_3), \cdots, (v_6, v_1)$.*

EXAMPLE 1. In the bipartite network of Figure 1, the upper and lower layer vertex sets are $U = \{u_1, u_2, u_3, u_4\}$ and $L = \{l_1, l_2, l_3, l_4\}$, respectively. Vertices $\{l_1, l_2, l_3, u_1, u_2, u_3\}$ can form a bi-triangle, i.e., l_1-u_1-l_2-u_3-l_3-u_2-l_1. Notice that unlike triangles in unipartite networks, the same set of vertices may be involved in several bi-triangles since they can be linked by different sets of edges. For instance, vertices $\{l_2, l_3, l_4, u_2, u_3, u_4\}$ in Figure 1 participate in two bi-triangles, i.e., l_2-u_2-l_3-u_4-l_4-u_3-l_2 and l_2-u_3-l_3-u_4-l_4-u_2-l_2.

DEFINITION 3. ***Wedge:*** *Given a bipartite network $G = (V{=}(U \cup L), E)$, a vertex triplet (u, v, w) forms a wedge, if $\{u, v, w\} \subseteq V$ and $\{(u, v), (v, w)\} \subseteq E$. We call w the middle-vertex of the wedge.*

Table 1: Notations and meanings.

Notation	Meaning
$G{=}(V{=}(U{\cup}L), E)$	a bipartite network
n, m	the numbers of vertices and edges in G resp.
$e = (u, l)$	an edge between vertices u and l
$w = (x, y, z)$	a wedge with three vertices x, y, and z
$s = (x, y, z, t)$	a super-wedge constituted by edges $(x, y), (y, z), (z, t) \in E$
$N(v)$	the set of neighbors of vertex v
$N_k(v)$	the set of k-hop neighbors of vertex v
$d(v)$	the degree of vertex v
$d_k(v)$	the number of k-hop neighbors of vertex v
$\widehat{d}$	the maximum degree of vertices in G
$\bar{d}$	the average degree of vertices in G

DEFINITION 4. ***Super-wedge:*** *Given a bipartite network $G = (V{=}(U \cup L), E)$, a vertex quadruplet (u, v, w, t) is a super-wedge, if $\{u, v, w, t\} \subseteq V$ and $\{(u, v), (v, w), (w, t)\} \subseteq E$. We call v and w middle-vertices.*

Clearly, a super-wedge can be considered as two connected wedges that share an edge. A bi-triangle can also be considered as the join result of three wedges or two super-wedges.

EXAMPLE 2. In the bipartite network of Figure 1, (l_1, u_1, l_2) forms a wedge and (l_1, u_1, l_2, u_2) is a super-wedge.

PROBLEM 1. ***Bi-triangle counting:*** *Given a bipartite network $G = (V{=}(U \cup L), E)$, return the total number of bi-triangles in G.*

3 A WEDGE BASED ALGORITHM

In this section, we present a baseline bi-triangle counting algorithm, called `WJ-Count`. It borrows the key idea of triangle counting algorithm [17] on the unipartite network: specifically, for each vertex v, it enumerates each vertex pair (x, y) among v's neighbors, and checks whether (v, x, y) form a triangle. To adapt this idea for bi-triangle counting, we enumerate all the vertices in a specific layer (say lower layer) and for each vertex v, we enumerate each vertex pair (x, y) among v's 2-hop neighbors, and then count the number of bi-triangles containing the triplet of v, x, and y. Finally, we aggregate the count on each vertex to get the total count of bi-triangles.

In the following, we first discuss how to count the number of bi-triangles containing three vertices, which is a key step, and then present the overall algorithm.

3.1 Bi-triangle counting for three vertices

We start from the concept of *wj-unit*, which describes the intuition that a bi-triangle can be regarded as three wedges joining together.

DEFINITION 5. ***wj-unit:*** *Given a bipartite network $G = (V{=}(U \cup L), E)$, a wj-unit is a connected subgraph formed by three wedges $w_i = (x_i, y_i, z_i)$ $(1 \leq i \leq 3)$, s.t. $z_1{=}x_2$, $z_2{=}x_3$, and $z_3{=}x_1$.*

It is easy to observe that a bi-triangle must be a wj-unit, but a wj-unit may not correspond to a bi-triangle, because a wj-unit may be acyclic, i.e., its wedges may share the same middle vertex. According to the ways of sharing the middle vertex, we divide the acyclic wj-units into two types:

- **Type-α:** A wj-unit is a type-α acyclic wj-unit, if two of its wedges share the same middle vertex.

- **Type-β:** A wj-unit is a type-β acyclic wj-unit, if all of its wedges share the same middle vertex.

Example 3 illustrates these two types of acyclic wj-units.

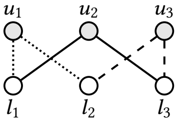
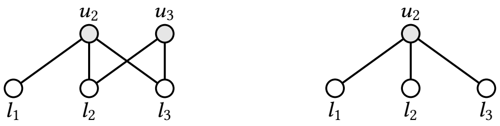


Figure 3: An example of wj-unit.



(a) Type-α. **(b)** Type-β.

Figure 4: Illustrating acyclic wj-units.

Example 3. *In the bipartite network of Figure 1, the three wedges (l_1, u_2, l_2), (l_2, u_3, l_3), and (l_3, u_2, l_1) form a type-α acyclic wj-unit, since (l_1, u_2, l_2), and (l_3, u_2, l_1) share the same middle vertex u_2 (see Figure 4(a)). The three wedges (l_1, u_2, l_2), (l_2, u_2, l_3), and (l_3, u_2, l_1) form a type-β acyclic wj-unit, because they share the same middle vertex u_2 (see Figure 4(b)).*

As a result, a natural idea of counting the number of bi-triangles containing three given vertices is to first obtain all the wj-units by performing wedge join, and then subtract the number of acyclic wj-units, as stated in Theorem 1.

Theorem 1. *Given a bipartite network $G = (V{=}(U \cup L), E)$, and three vertices l_i, l_j, and $l_k \in L$, the number of bi-triangles containing these vertices can be computed by the following formula:*

$$|N(l_i) \cap N(l_j)||N(l_j) \cap N(l_k)||N(l_k) \cap N(l_i)| - \big(|N(l_i) \cap N(l_j)| + |N(l_j) \cap N(l_k)| + |N(l_k) \cap N(l_i)| - 2\big)|N(l_i) \cap N(l_j) \cap N(l_k)|$$

Proof. First, it is obvious that the total number of wj-units containing l_i, l_j, and l_k is $|N(l_i){\cap}N(l_j)||N(l_j){\cap}N(l_k)||N(l_k){\cap}N(l_i)|$. Next, we discuss how to calculate the number of acyclic wj-units. Assume that the three wedges of an acyclic wj-unit that contain l_i, l_j, and l_k are (l_i, x, l_j), (l_j, y, l_k), and (l_k, z, l_i) respectively. In case that the acyclic wj-unit is a type-α acyclic wj-unit, then two of the three middle vertices are the same vertx. W.l.o.g., we assume that the wedges (l_i, x, l_j) and (l_j, y, l_k) share the same middle vertex, i.e., $x{=}y$ which must be in the set $N(l_i) \cap N(l_j) \cap N(l_k)$. This implies that there are $|N(l_i) \cap N(l_j) \cap N(l_k)|$ choices for $x = y$, so for a given $x = y$, the number of choices for z is $|N(l_i) \cap N(l_k)|{-}1$, since $z \in N(l_i) \cap N(l_k)$ and $z \neq x = y$. Thus, in this case, the number of type-α acyclic wj-units is $|N(l_i){\cap}N(l_j){\cap}N(l_k)|\big(|N(l_i){\cap}N(l_k){-}1|\big)$

In case that the acylic wj-unit is a type-β acyclic wj-unit, its three wedges share the same middle vertex, i.e., $x{=}y{=}z$, so the number of type-β acyclic wj-units is $|N(l_i) \cap N(l_j) \cap N(l_k)|$.

Finally, by combining the analysis above, we can derive the formula shown in Theorem 1. □

Clearly, by Theorem 1, we can compute the number of bi-triangles containing any three specific vertices in $O(\widehat{d})$ time, since we only need to collect the sets of common neighbors of each pair of vertices which takes $O(\widehat{d})$ time.

3.2 The overall algorithm of WJ-Count

Based on the wj-unit and Theorem 1, we develop the algorithm, denoted by WJ-Count, as shown in Algorithm 1. First, it initializes a variable $\Lambda{=}0$ (line 1) to keep the number of bi-triangles. Then, for each vertex l in L, it collects l's 2-hop neighbors in a set S (lines 3-7). Note that to avoid recomputation over the same vertex triplet, S only keeps vertices whose ids are larger than l's id (lines 6-7). Next, it enumerates each pair among l's 2-hop neighbors set, and for each such pair (x, y), it counts the number of bi-triangles containing l, x, and y using Theorem 1, and updates Λ accordingly (lines 8-9). Finally, it returns Λ as the result (line 10).

Algorithm 1: WJ-Count

```
   Input: $G = (V{=}(U \cup L), E)$
   Output: The number of bi-triangles in $G$
1  $\Lambda \leftarrow 0$
2  foreach $l \in L$ do
3      $S \leftarrow \emptyset$
4      foreach $x \in N(l)$ do
5          foreach $y \in N(x)$ do
6              if $y$'s id is larger then $l$'s id then
7                  add $y$ into $S$
8      foreach pair $(x, y)$ among $S$ do
9          $\Lambda \leftarrow \Lambda + |N(l) \cap N(x)||N(l) \cap N(y)||N(x) \cap N(y)| - (|N(l) \cap N(x)| + |N(x) \cap N(y)| + |N(y) \cap N(l)| - 2)|N(l) \cap N(x) \cap N(y)|$
10 return $\Lambda$
```

Complexity. The time complexity of WJ-Count is $O\big(\widehat{d} \times \sum_{l \in L} (d_2(l))^2 + \sum_{u \in U} d(u)^2\big)$. The space complexity of WJ-Count is $O\big(m{+}n\big)$. Please see details for the analysis in the Appendix [58].

Limitations. Although WJ-Count is intuitive, it has two major limitations: (1) It suffers from a serious issue of recomputation, since a pair of vertices x and y may be contained in many vertex triplets, leading to heavy recomputation of $|N(x) \cap N(y)|$. For example, in the bipartite network of Figure 1, the pair of vertices l_1 and l_2 is in a triplet of l_1, l_2, and l_3, and another triplet of l_1, l_2, and l_4, so $|N(l_1) \cap N(l_2)|$ will be computed twice. (2) The number of enumerated vertex triplets is too large. For example, the Discogs network, which contains 2,025,596 vertices and 5,320,276 edges, has $8.19{\times}10^{15}$ triplets that WJ-Count will enumerate.

To tackle these limitations, a simple idea is to project the bipartite network into a unipartite network, say G_L, such that its vertex set is L and there is an edge between any pair of vertices if they are contained in at least one wedge in G. Then, for each edge $(x, y) \in G_L$, we assign it a weight denoting the number of wedges connecting x and y. By doing this, the recomputation issue can be avoided since we can look up the edges' weights in G_L to get the size of common neighbors set. Meanwhile, the number of triplets enumerated can be reduced, since for any bi-triangle, its three vertices in the lower layer L must form a triangle in G_L, making us only need to enumerate triplets whose vertices form triangles in G_L. However, this idea is still problematic because G_L is dense and large, which implies that (1) although the number of triplets enumerated is reduced, the number of triangles in G_L is still very large; and (2) G_L cannot be kept in the main memory even for moderate-size networks (e.g., the Discogs network in Table 2 has 5,302,276 edges, but its projected network has 19,534,205,828 edges). Hence, it is desirable to develop more efficient counting algorithms with different paradigms.

4 A SUPER-WEDGE BASED ALGORITHM

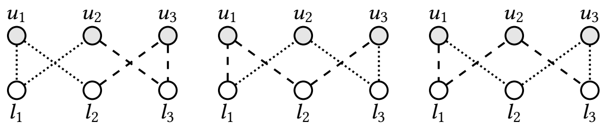
To avoid the limitations of `WJ-Count`, in this section we propose a more efficient algorithm, called `SWJ-Count`, which counts bi-triangles based on the join of super-wedges.

4.1 The main idea of `SWJ-Count`

We begin with a novel concept of swj-unit.

Definition 6. ***swj-unit:*** *Given a bipartite network $G = (V{=}(U \cup L), E)$, an swj-unit is a connected subgraph formed by two super-wedges $s_i{=}(x_i, y_i, z_i, t_i)$ $(1{\le}i{\le}2)$, which satisfy $x_1{=}x_2$ and $t_1{=}t_2$.*

As shown in Figure 5, an swj-unit can be formed by three different pairs of super-wedges. For example, in the bipartite network of Figure 1, super-wedges $s_1{=}(l_2, u_1, l_1, u_2)$ and $s_2{=}(l_2, u_3, l_3, u_2)$ form an swj-unit (see Figure 5(a)).



(a)

(b)

(c)

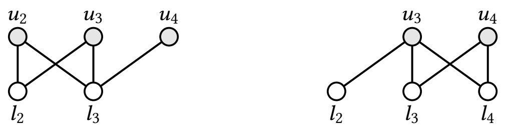
Figure 5: An swj-unit with three different pairs of super-wedges (each super-wedge is in a type of dashed lines).

Similar to wj-unit, a bi-triangle must be an swj-unit, but an swj-unit may not correspond to a bi-triangle, as its super-wedges may share the same middle vertex, i.e., $y_1{=}y_2$ or $z_1{=}z_2$, and we call it an acyclic swj-unit. Note that we ignore the case of sharing two middle vertices (i.e., $y_1{=}y_2$ and $z_1{=}z_2$), since s_1 is identical to s_2 in this case. There are two types of acyclic swj-units, both of which can be considered as a butterfly with an additional edge:

- **Type-L:** The two super-wedges of the swj-unit share a middle vertex in the lower layer.
- **Type-U:** The two super-wedges of the swj-unit share a middle vertex in the upper layer.

Example 4. *In the bipartite network of Figrue 1, super-wedges $s_1{=}(l_2, u_2, l_3, u_4)$ and $s_2{=}(l_2, u_3, l_3, u_4)$ form a type-L acyclic swj-unit (see Figure 6(a)), while super-wedges $s_2{=}(l_2, u_3, l_3, u_4)$ and $s_3{=}(l_2, u_3, l_4, u_4)$ form a type-U acyclic swj-unit (see Figure 6(b)).*

A straightforward idea to count bi-triangles using swj-units, similar to `WJ-Count`, works as follows: (1) for two vertices l and u connected by a super-wedge, we count the number of swj-units and acyclic swj-units containing them respectively, so the number of bi-triangles containing them can be derived quickly; and (2) we do (1) for all the pairs of vertices that are connected by super-wedges to get the total count of bi-triangles.



(a) Type-L.

(b) Type-U.

Figure 6: Illustration for acyclic swj-units.

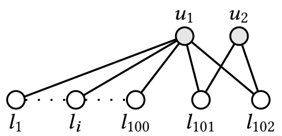


Figure 7: A bipartite network ($1 \le i \le 100$).

Although the approach above is simple, it may involve much recomputation when computing. For example, on the bipartite network in Figure 7, we will show that the number of butterflies containing u_1 and u_2 can be recomputed at least 100 times. First, when computing the number of the type-U acyclic swj-units containing vertices l_1 and u_2, it needs to computes the number of butterflies containing the u_1 and u_2, as each such butterfly can form a type-U acyclic swj-units with l_1; Similarly, when computing the number of type-U acyclic swj-units containing vertices l_i and u_2 for all i = 2, 3, ..., 100, the number of butterflies containing the u_1 and u_2 will be computed repeatedly. Hence, the butterfly counting over vertices u_1 and u_2 will be repeated at least 100 times. Clearly, the number of butterflies over a vertex pair may be recomputed for $\widehat{d}$ times and there are up to n^2 vertex pairs that are linked by wedges, so there could be up to $O(\widehat{d} \times n^2)$ times of recomputation.

To avoid the recomputation above, we propose another approach: instead of conducting counting for each vertex pair that is connected by a super-wedge, we propose a two-stage strategy where the first stage globally counts the numbers of swj-units, and the second stage globally counts acyclic swj-units on the entire network, respectively; and then derive the total number of bi-triangles by subtracting the latter one from the first one. The intuitions behind are that: (1) the number of swj-units containing two vertices that are connected by a super-wedge can be obtained easily; and (2) the number of acyclic swj-units over two vertices that are connected by a wedge can be obtained efficiently by Lemma 1.

Lemma 1. *Given a bipartite network $G = (V{=}(U \cup L), E)$, for any two vertices $l_i, l_j \in L$, the number of type-L acyclic swj-units containing them is $\binom{|N(l_i) \cap N(l_j)|}{2}(d(l_i){+}d(l_j){-}4)$; for any two vertices $u_i, u_j \in U$, the number of type-U acyclic swj-units containing them is $\binom{|N(u_i) \cap N(u_j)|}{2}\,(d(u_i) + d(u_j) - 4)$.*

Proof. We only prove the case of type-L, since the proof for the case of type-U is similar. For l_i and l_j, there are $\binom{|N(l_i \cap N l_j)|}{2}$ butterflies containing them [43]. For each butterfly, there are $d(l_i) -$ 2 edges adjacent to l_i that can form acyclic swj-units with the butterfly, and $d(l_j) - 2$ edges adjacent to l_j that can form acyclic

Algorithm 2: `SWJ-Count`

```
   Input: $G = (V{=}(U \cup L), E)$
   Output: The number of bi-triangles in $G$
1  $\Lambda \leftarrow 0, \overline{\Lambda} \leftarrow 0$
2  foreach $l \in L$ do
3      $S \leftarrow \emptyset, H \leftarrow \emptyset$
4      foreach $u \in N(l)$ do
5          foreach $x \in N(u)/\{l\}$ do
6              if $x \notin S.keySet()$ then $S.insert((x, 0))$
7              $S[x] \leftarrow S[x] + 1$
8      foreach $x \in S.keySet()$ do
9          $\overline{\Lambda} \leftarrow \overline{\Lambda} + \binom{S[x]}{2} \times (d(x) - 2)$
10         foreach $y \in N(x)$ do
11             if $y \notin H.keySet()$ then $H.insert((y, 0))$
12             $H[y] \leftarrow H[y] + S[x]$
13     foreach $y \in H.keySet()$ do
14         if $y \in N(l)$ then $H[y] \leftarrow H[y]-d(y)+1$
15         $\Lambda \leftarrow \Lambda + \binom{H[y]}{2}$
16 foreach $u \in U$ do
17     $T \leftarrow \emptyset$
18     foreach $l \in N(u)$ do
19         foreach $t \in N(l)/\{u\}$ do
20             if $t \notin T.keySet()$ then $T.insert((t, 0))$
21             $T[t] \leftarrow T[t] + 1$
22     foreach $t \in T.keySet()$ do
23         $\overline{\Lambda} \leftarrow \overline{\Lambda} + \binom{T[t]}{2} \times (d(t) - 2)$
24     return $\frac{\Lambda-(\overline{\Lambda})}{3}$
```

swj-units with the butterfly. Hence, there are $\binom{|N(l_i) \cap N(l_j)|}{2}(d(l_i) + d(l_j) - 4)$ acyclic type-L swj-units containing l_i and l_j. □

Following the intuitions above, we have Theorem 2.

Theorem 2. *Given a bipartite network $G = (V{=}(U \cup L), E)$, if there are Λ swj-units in G and $\overline{\Lambda}$ acyclic swj-units in G, then the number of bi-triangles is $\frac{(\Lambda-\overline{\Lambda})}{3}$.*

Proof. The theorem holds obviously, since each bi-triangle can be represented by three swj-units (see Figure 5). □

4.2 The overall algorithm of `SWJ-Count`

Based on the discussions above, we propose `SWJ-Count`, whose key steps are as follows: First, for each vertex $l \in L$, we enumerate l's-2 hop neighbors and count the number of type-L acyclic swj-units containing l and each of its 2-hop neighbors. Based on l's 2-hop neighbors, we further enumerate l's 3-hop neighbors and count the number of swj-units containing l and each of its 3-hop neighbors. Then, we count the number of type-U acyclic swj-units over the upper layer in a similar manner. Finally, we get the number of bi-triangles, by subtracting the number of acyclic swj-units from the total number of swj-units.

Algorithm 2 shows `SWJ-Count`. First, it initializes two variables Λ and $\overline{\Lambda}$ (line 1), where Λ and $\overline{\Lambda}$ keep the numbers of swj-units and acyclic swj-units respectively. Then, for each vertex $l \in L$ (line 2), it initializes two maps S and H (line 3), where S's keys are l's 2-hop neighbors, values are numbers of wedges (2-paths) connecting l and its 2-hop neighbors; and H has a similar function for l's 3-hop neighbors. Next, it enumerates l's neighbors' neighbors sets to get S (lines 4-7). After that, the for-loop (lines 8-12) continues to enumerate l's 3-hop neighbors and builds H. Meanwhile, $\overline{\Lambda}$ is updated by adding the number of type-L acyclic swj-units that containing l and x (line 9). Then, the for-loop (lines 13-15) computes the number of swj-units adjacent to l and updates the variable Λ. As the values in H may be larger than the number of super-wedges connecting l and y (e.g., sequences like l, y, x, y is contained), it subtracts the number of self-intersected 3-paths (line 14). Later, the for-loop (lines 16-23) computes the number of type-U acyclic swj-units and updates $\overline{\Lambda}$. Finally, it returns $\frac{\Lambda-\overline{\Lambda}}{3}$ (line 24).

Example 5. *For the bipartite network in Figure 1,* `SWJ -Count` *starts from l_1 and enumerates its 2- and 3-hop neighbors to build the 2- and 3-hop neighbor maps S=[(l_2, 2), (l_3, 1), (l_4, 1)] and H=[(u_1, 2), (u_2, 4), (u_3, 4), (u_4, 2)], respectively. Meanwhile, $\overline{\Lambda}$ is updated as $\binom{2}{2} \times 1 + \binom{1}{2} \times 1 + \binom{1}{2} \times 1 = 1$, and Λ is updated as 7. Then, it continues to process l_2, l_3 and l_4 in the same way, and $\overline{\Lambda}$ and Λ are updated to 11 and 34 respectively. Finally, it enumerates vertices of U to get the number of type-U acyclic swj-units, and returns 3.*

Complexity. The time compleixty of `SWJ-Count` is $O\big(\sum_{l \in L}\big(d(l)^2{+}d_3(l){+}\sum_{x \in N_2(l)} d(x)\big){+}\sum_{u \in U} d(u)^2\big)$. The space complexity of `SWJ-Count` is $O(m+n)$. Please see details for the analysis in Lemma 7 in the Appendix [58].

`SWJ-Count` v.s. `WJ-Count`. It may not be easy to theoretically judge which one runs faster. Instead, we compare them in terms of the operations regarding a vertex $l \in L$. In the beginning, they take the same cost to build l's 2-hop neighbors (lines 2-7 in Algorithm 1 and lines 2-7 in Algorithm 2). Next, `WJ-Count` enumerates all the $\binom{d_2(l)}{2} \approx \frac{(d_2(l))^2}{2}$ vertex pairs among S (lines 8-9 in Algorithm 1), while `SWJ-Count` enumerates all l's 3-hop neighbors to build H (lines 8-15 in Algorithm 2), which has $\sum_{x \in N_2(l)} d(x) \approx d_2(l)\bar{d}$ operations (here $\bar{d}$ denotes the average degree). In practice, since for most $l \in L$, $\frac{(d_2(l))^2}{2} \gg d_2(l)\bar{d}$, `SWJ-Count` runs much faster.

5 A RANKED SUPER-WEDGE BASED ALGORITHM

In this section, we further optimize `SWJ-Count` by introducing a ranking mechanism. Recall that `SWJ-Count` processes all the super-wedges from vertices in the layer L, but each super-wedge can be processed from any of its end vertices in either L or U. We have the intuition that vertices with higher degrees always have more adjacent super-wedges. Hence, processing super-wedges from vertices with higher degrees may speed up the process by sharing some computation. We illustrate this by the network in Figure 8. (1) If we process all the super-wedges from vertices in L, then for each vertex l, to find the super-wedges ending at l, we need to sequentially enumerate the neighbor sets of l, l's 1-hop neighbors, and l's 2-hop neighbors, which take $d(l)$, $\sum_{x \in N(l)} d(x)$, and $\sum_{x \in N_2(l)} d(x)$ operations respectively, so processing vertices from l_1 to l_{102} takes $\sum_{h=1}^{102}\big(d(l_h) + \sum_{x \in N(l_h) \cup N_2(l_h)} d(x)\big)$=41,200 operations to find all the super-wedges. (2) Similarly, if we process all the super-wedges from vertices in U, we also need 41,200 operations. (3) If we process super-wedges in the left part of G (with vertices u_1, u_2, l_1, $\cdots$, l_{100}) from vertices in U, and then process super-wedges in the right part of G (with vertices l_{101}, l_{102}, u_3, $\cdots$, u_{102}) from vertices in L, then we only need $\sum_{v \in \{u_1, u_2, l_{201}, l_{202}\}} \big(d(v) + \sum_{x \in N(v) \cup N_2(v)} d(x)\big)$=2,000 operations. Clearly, by ranking vertices according to their degrees, the process can benefit from searching super-wedges from vertices with higher degrees; moreover, the ranking mechanism can allow to us to complete counting by processing only a small fraction of swj-units, which are termed as *rswj-units as explained in the following.*

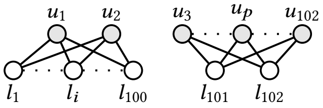


Figure 8: A bipartite network.

5.1 The main idea of RSWJ-Count

We begin with some key concepts in RSWJ-Count.

Definition 7. ***Vertex rank:*** *Given a bipartite network G = (V=($U \cup L$), E), a vertex rank σ over G is a bijective mapping $\sigma : V \longrightarrow \{1, 2, \cdots, |V|\}$.*

In this section, we focus on vertex rank according to degrees, denoted as DegRank. More specific, a rank σ is DegRank if σ satisfies that if $u, v \in V$ s.t. $d(u) < d(v)$, then $\sigma(u) > \sigma(v)$.

We use notations $\Phi_k(r, v)$=$\{u|u \in N_k(v) \land \sigma(u)<r\}$ and $\Psi_k(r, v)$=$\{u|u \in N_k(v) \land \sigma(u)>r\}$ to describe the subsets of v's k-hop neighbors with ranks smaller and larger than r, respectively. Besides, we use $\phi_k(r, v)$ and $\psi_k(r, v)$ to denote $|\Phi_k(r, v)|$ and $|\Psi_k(r, v)|$ respectively. When k=1, subscripts are omitted.

Definition 8. ***r-super-wedge and r-wedge:*** *Given a bipartite network G = (V=($U \cup L$), E) and a rank σ, we say a super-wedge (x, y, z, t) is an r-super-wedge if $\sigma(x)$=min$\{\sigma(x), \sigma(y), \sigma(z), \sigma(t)\}$. Similarly, we say a wedge (x, y, z) is an r-wedge if $\sigma(x)$=min$\{\sigma(x), \sigma(y), \sigma(z)\}$.*

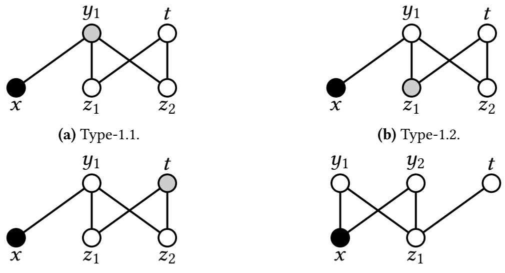
Based on the above definitions, we introduce the concept of rswj-unit and Theorem 3.

Definition 9. ***rswj-unit:*** *In a bipartite network G with rank σ, an swj-unit is an rswj-unit if it is composed of two r-super-wedges.*

Theorem 3. *Given a bipartite network G, each bi-triangle of G can be represented as a unique rswj-unit.*

Proof. Figure 5 shows that each bi-triangle can be represented by three different swj-units composed of three pairs of super-wedges respectively. However, if we focus on the vertex v whose $\sigma(v)$ is the smallest and search super-wedges starting from v in the bi-triangle, then we can only get one pair of super-wedges. It is easy to see that this pair of super-wedges are all r-super-wedges. □

Similar to the swj-unit and wj-unit, some rswj-units cannot form bi-triangles, which are also called acyclic rswj-units. Specifically, let r_1=(x, y_1, z_1, t) and r_2=(x, y_2, z_2, t) be two r-super-wedges. Then, the types of acyclic rswj-units they can form are as follows:



(a) Type-1.1. **(b)** Type-1.2. **(c)** Type-1.3. **(d)** Type-2.

Figure 9: Illustration for acyclic rswj-units (start-vertices are in black; anchor-vertices are in gray).

- **Type-1:** they form a type-1 acyclic rswj-unit, if they share the first middle vertex, i.e., y_1=y_2, denoted by y. For better illustration of algorithms, we introduce a concept of *anchor-vertex*, based on which we further divide this type into three sub-types. Let S=$\{y, z_1, z_2, t\}$. Then, the anchor-vertex a is

$$a \leftarrow \underset{x \in S}{\arg\min}\{\sigma(x)\}. \quad (1)$$

  The three sub-types are:
  - **Type-1.1:** If a is y (see Figure 9(a)).
  - **Type-1.1:** If a is z_1 or z_2 (see Figure 9(b)).
  - **Type-1.3:** If a is t (see Figure 9(c)).
- **Type-2:** they form a type-2 acyclic rswj-unit, if they share the second middle vertex, i.e., z_1=z_2 (see Figure 9(d)). We call x is the anchor-vertex of the type-2 acyclic rswj-unit.

Next, Lemmas 2-4 illustrate how to count the number of acyclic rswj-units in each type for any two vertices in a bipartite network G = (V=($U \cup L$), E).

Lemma 2. *Given a bipartite network and its two vertices v_1 and v_2 with $\sigma(v_1) < \sigma(v_2)$, the number of type-1.1 acyclic rswj-units, which contain v_1 and v_2 where v_1 is the anchor-vertex, is $\binom{|\Pi(v_1,\, v_2)|}{2}\phi(\sigma(v_1), v_1)$, where $\Pi(v_1, v_2)$ denotes the set $\Psi(\sigma(v_1), v_1) \cap N(v_2)$.*

Proof. It is easy to see that there are $\binom{|\Pi(v_1,\, v_2)|}{2}$ butterflies containing v_1 and v_2. Also, for each such butterfly, there are $\phi(\sigma(v_1), v_1)$ edges which can form type-1.1 acyclic rswj-units with the butterfly; hence, the lemma holds. □

Lemma 3. *Given a bipartite network and its two vertices v_1 and v_2 with $\sigma(v_1) < \sigma(v_2)$, the number of type-1.2 acyclic rswj-units, which contain v_1 and v_2 where v_1 is the anchor-vertex, is ($|\Pi(v_1, v_2)|$ − 1)× $\sum\limits_{u \in \Pi(v_1,\, v_2)} \phi(\sigma(v_1), u)$.*

Proof. For each $u \in \Pi(v_1, v_2)$, there are $|\Pi(v_1, v_2) - 1|$ vertices which can form a butterfly with u, v_1 and v_2; besides, u has $\phi(\sigma(v_1), u)$ adjacent edges which can form type-1.2 acyclic rsjw-units with the butterfly. Hence, the number of acyclic rswj-units can be calculated as ($|\Pi(v_1, v_2)|$ − 1)× $\sum\limits_{u \in \Pi(v_1,\, v_2)} \phi(\sigma(v_1), u)$. □

Lemma 4. *Given a bipartite network and its two vertices v_1 and v_2 with $\sigma(v_1) < \sigma(v_2)$, the number of type-1.3 and type-2 acyclic* *rswj-units, which contain v_1 and v_2 where v_1 is the anchor-vertex, is* $\binom{|\Pi(v_1, v_2)|}{2}\big(d(v_2) - 2\big)$.

Proof. It is easy to see that there are $\binom{|\Pi(v_1, v_2)|}{2}$ butterflies containing v_1 and v_2. Besides, for each such butterfly, there are $\phi(\sigma(v_1), v_2)$ edges which can form type-1.3 acyclic rswj-units with the butterfly and $\psi(\sigma(v_1), v_2)$-2 edges which can form type-2 acyclic rswj-units with the butterfly; hence, there are $\binom{|\Pi(v_1, v_2)|}{2}(d(v_2) - 2)$ type-1.3 and type-2 acyclic rswj-units. □

Besides, the following definition and the lemma gives us a pruning technique based on the (2, 2)-core [30, 52].

Definition 10. *Given a bipartite network $G = (V{=}(U \cup L), E)$ and two integers α and β, the (α, β)-core [30] of G is the largest subgraph of G such that any of its vertex in U has a degree at least α, and any of its vertex in L has a degree at least β.*

Lemma 5. *Given a bipartite network $G = (V{=}(U \cup L), E)$, all the bi-triangles of G must be contained in its (2, 2)-core.*

Theorem 4. *Given a bipartite network $G = (V{=}(U \cup L), E)$, if there are Λ rswj-units and $\overline{\Lambda}$ acyclic rswj-units, then there are $(\Lambda - \overline{\Lambda})$ bi-triangles.*

5.2 The overall algorithm of RSWJ-Count

Algorithm 3 summarizes the key steps of RSWJ-Count. In the outset, as a preprocessing step, it computes the (2, 2)-core by recursively removing vertices that are not in it, and builds the DegRank on the (2, 2)-core (line 1). Then it initializes two variables Λ and $\overline{\Lambda}$ for keeping the numbers of rswj-units and acyclic rswj-units (line 2), respectively. Next, the for-loop (lines 3-19) enumerates each vertex $v \in V$ to obtain the number of rswj-units and acyclic rswj-units adjacent to it. Specifically, it first initializes three maps S, H, and T, where S and H are used to keep v's 2- and 3-hop neighbors respectively, and T keeps the information that will be used in calculating the number of type-1.2 acyclic rswj-units. Then, it builds maps S and T (lines 5-9). Note that only vertices whose rank values are larger than v's rank value are visited (lines 5-6), which further indicates that $S[y]$ is the number of r-wedges connecting v and y, and $T[y]$ is $\sum_{x \in \Pi(v,y)} \phi(\sigma(v), x)$ (which is a key part of calculating the type-1.2 acyclic rswj-units). After that, the for-loop (lines 10-16) enumerates neighbors set of each vertex in S to build H, where for each of v's 3-hop neighbor z, $H[z]$ is the number of r-super-wedges connecting v and z. Meanwhile, the numbers of type-1.1, type-1.2, type-1.3, and type-2 acyclic rswj-units are calculated by Lemmas 2–4 and summarized in $\overline{\Lambda}$ (lines 14-16). Later, it calculates the number Λ of rswj-units using H (lines 17-19). Note that line 18 removes self-intersected 3-paths. Finally, it subtracts $\overline{\Lambda}$ from Λ to get the total count by Theorem 4 (line 21).

Complexity. The time complexity of RSWJ-Count is $O\big(m + n + \sum_{v \in V} d(v)\big(\phi(v) + \phi_2(v)\big)\big)$, where $\Phi_k(v)$ is the abbreviation of $\Phi_k(\sigma(v), v)$ and $\phi_k(v)$, $\Psi_k(v)$, $\psi_k(v)$ have similar meanings. The space cost of RSWJ-Count is $O(m{+}n)$. Please see details for the analysis in Lemma 8 in the Appendix [58].

Algorithm 3: RSWJ-Count

```
   Input: $G = (V{=}(U \cup L), E)$;
   Output: The number of bi-triangles in $G$;
1  compute the (2, 2)-core and build the DegRank on it
2  $\Lambda \leftarrow 0, \overline{\Lambda} \leftarrow 0$
3  foreach $v \in V$ do
4      $S \leftarrow \emptyset, H \leftarrow \emptyset, T \leftarrow \emptyset$
5      foreach $x \in \Psi(\sigma(v), v)$ do
6          foreach $y \in \Psi(\sigma(v), x)$ do
7              if $y \notin S.keySet()$ then
8                  $S.insert((y, 0)), T.insert((y, 0))$
9              $S[y] \leftarrow S[y] + 1, T[y] \leftarrow T[y] + \phi(\sigma(v), x)$
10     foreach $y \in S.keySet()$ do
11         foreach $z \in \Psi(\phi(v), y)$ do
12             if $z \notin H.keySet()$ then $H.insert((z, 0))$
13             $H[z] \leftarrow H[z] + S[y]$
14         $\overline{\Lambda} \leftarrow \overline{\Lambda} + \binom{S[y]}{2} \times \phi(\sigma(v), v)$        /* Lemma 2 */
15         $\overline{\Lambda} \leftarrow \overline{\Lambda} + (S[y] - 1) \times T[y]$        /* Lemma 3 */
16         $\overline{\Lambda} \leftarrow \overline{\Lambda} + \binom{S[y]}{2} \times (d(y) - 2)$        /* Lemma 4 */
17     foreach $z \in H.keySet()$ do
18         if $z \in N(v)$ then $H[z] \leftarrow H[z] - \psi(\sigma(v), z)$
19         $\Lambda \leftarrow \overline{\Lambda} + \binom{H[z]}{2}$
20 return $(\Lambda - \overline{\Lambda})$
```

6 LOCAL BI-TRIANGLE COUNTING

In this section, we study an extension, called local bi-triangle counting, which aims to count the numbers of bi-triangles that contain a specific vertex v_0 or edge e_0. This problem is useful in many real applications, such as local clustering coefficient computation in bipartite networks [37].

To solve the local counting problem, we adapt our ideas for global counting. Generally, we have three options, based on representing bi-triangles as (1) wj-units, (2) swj-units, or (3) rswj-units. First, option (1) is not a good choice, because it may result in enumerating a large number of vertex triplets. Next, among options (2) and (3), it seems that (3) should be the better one, because in the case of global counting, RSWJ-Count (adopting the idea of (3)) outperforms SWJ-Count (adopting the idea of (2)). However, in fact, in the case of local counting, option (3) is the worse choice. The main reason is that a bi-triangle containing v_0 may be composed of two r-super-wedges whose end vertex is not v_0, but v_0's 1, 2, 3-hop neighbors. Hence, to count these bi-triangles, we have to find the r-super-wedges composing them, which requires us to first enumerate v_0's 1, 2, 3-hop neighbors. However, the enumeration of v_0's 1, 2, 3-hop neighbors already includes the process of all the super-wedges starting from v_0. This indicates that representing bi-triangles as swj-units and processing these swj-units from v_0 is a better choice. In other words, (2) is better than (3).

6.1 Bi-triangle counting for a query vertex

To count the number of bi-triangles containing v_0, we present an algorithm, denoted by V-LCount, which follows the idea of

SWJ-Count, but has a key difference that it only counts the swj-units and acyclic swj-units containing v_0.

Algorithm 4: V-LCount

```
   Input: $G = (V=(U \cup L), E)$ and a vertex $v_0 \in V$
   Output: The number of bi-triangles containing $v_0$;
1  $\Lambda \leftarrow 0, \overline{\Lambda} \leftarrow 0, S \leftarrow \emptyset, H \leftarrow \emptyset$
2  foreach $y \in N_2(v_0)/\{v_0\}$ do $S[y] \leftarrow S[y] + 1$
3  foreach $y \in S.keySet()$ do
4      $\overline{\Lambda} \leftarrow \overline{\Lambda} + \binom{S[y]}{2} \times (d(y) - 2)$
5      foreach $z \in N(y)$ do $H[z] \leftarrow H[z] + S[y]$
6  foreach $z \in H.keySet()$ do
7      if $z \in N(l)$ then $H[z] \leftarrow H[z] - d(z) + 1$
8      $\Lambda \leftarrow \Lambda + \binom{H[z]}{2}$
9  foreach $x \in N(v_0)$ do
10     $T \leftarrow \emptyset$
11     foreach $z \in N_2(x)/\{v_0\} \wedge z.id < y.id$ do $T[z] \leftarrow T[z] + 1$
12     foreach $z \in T.keySet()$ do $\overline{\Lambda} \leftarrow \overline{\Lambda} + \binom{T[z]}{2}$
13 return $(\Lambda - \overline{\Lambda})$
```

Algorithm 4 shows V-LCount. First, it initializes two variables (i.e., Λ and $\overline{\Lambda}$) and two maps (i.e., S and T), which have similar meanings with those in SWJ-Count. Then, the for-loop (line 2) enumerates v's 2-hop neighbors to build S. After that, the for-loop (lines 3-5) enumerates v's 3-hop neighbors to build H; meanwhile, the number of acyclic swj-units are computed and updated in $\overline{\Lambda}$ (line 4). Then the for-loop (lines 6-8) enumerates v's 3-hop neighbors in H and updates Λ, where line 8 ensures that self-intersected paths are not included. Lastly, it counts the acyclic swj-units that are not counted in line 8, updates $\overline{\Lambda}$ (lines 9-12), and returns the result accordingly (line 13).

Complexity. It is easy to observe that V-LCount has the time complexity: $O\big(d_3(v_0) \times \sum_{x \in N(v_0)} d(x) + \sum_{y \in N_2(v_0)} d(y)\big)$, and the space complexity: $O\big(m + n\big)$.

6.2 Bi-triangle counting for a query edge

In this section, we present an algorithm E-LCount for counting the number of bi-triangles containing a query edge e_0=(u, l). It also follows the idea of SWJ-Count and only counts the swj-units and acyclic swj-units containing e_0.

Algorithm 5 presents the pseudocodes. Initially, it creates variables (i.e., Λ and $\overline{\Lambda}$) and maps (i.e., S, H, and T), which have similar meanings with those in V-LCount. Then, we build T and S by using two for-loops respectively (lines 2-5). Next, it enumerates the neighbor sets of vertices in S to build H (lines 6-8), and computes the number of acyclic swj-units (line 7). Finally, it enumerates vertices in the intersection of H and T to compute the number of swj-units (lines 9-11) and returns the overall count (line 12).

Complexity. It is easy to observe that E-LCount has the time complexity of $O\big(\sum_{x \in N(u) \cup N(l)} d(x) + \sum_{y \in N^2(u)} d(y)\big)$, and the space complexity of $O\big(m + n\big)$.

Algorithm 5: E-LCount

```
   Input: $G = (V=(U \cup L), E)$ and an edge $e_0$=$(u, l) \in E$
   Output: The number of bi-triangles containing $e_0$;
1  $\Lambda \leftarrow 0, \overline{\Lambda} \leftarrow 0, S \leftarrow \emptyset, T \leftarrow \emptyset, H \leftarrow \emptyset$
2  foreach $t \in N(u)/\{l\}$ do
3      foreach $w \in N(t)/\{u\}$ do $T[w] \leftarrow T[w] + 1$
4  foreach $x \in N(l)/\{u\}$ do
5      foreach $y \in N(x)/\{l\}$ do $S[y] \leftarrow S[y] + 1$
6  foreach $y \in S.keySet()$ do
7      if $y \in N(l)$ then $\overline{\Lambda} \leftarrow \overline{\Lambda} + S[y] \times (d(y) - 2)$
8      foreach $z \in N(y)/\{l\}$ do $H[z] \leftarrow H[z] + S[y]$
9  foreach $z \in H.keySet() \cap T.keySet()$ do
10     if $z \in N(l)$ then $H[z] \leftarrow H[z] - d(z) + 1$
11     $\Lambda \leftarrow \Lambda + H[z] \times T[z]$
12 return $(\Lambda - \overline{\Lambda})$
```

7 EXPERIMENTS

In this section, we present the results of empirical studies. We first discuss the setup in Section 7.1 and then present the experimental results in Section 7.2.

7.1 Setup

Datasets. We evaluate the algorithms on 11 real networks (see Table 2), including affiliation networks (e.g., Actor-Movie, Discogs and LiveJournal), bibliographic networks (e.g., DBLP and Dewiki), social networks (e.g., Twitter, Youtube, and Movielens), text networks (e.g., Reuters and Gottron), and hyperlink networks (e.g., Trackers). All these datasets are downloaded from KONECT[1]. Please note that all the networks are undirected, and we randomly choose one side as the upper layer and the other one as the lower layer. Besides, we include one synthetic dataset, namely Syn, which is generated according to the bipartite network model [4], where the vertices in the lower and upper layers follow the Power-law degree distribution and the Poisson degree distribution, respectively. Table 2 reports the statistics of these datasets.

Environment. All the experiments are conducted on a machine with a Linux system, an Intel Xeon E-2288G 3.7GHz CPU, and 64GB RAM. Note that we manually terminate running an algorithm if it cannot finish within 48 hours and mark it as +∞ in Figures 10-17.

7.2 Experimental results

1. Efficiency of bi-triangle counting algorithms. Figure 10 reports the efficiency of algorithms WJ-Count, SWJ-Count, and RSWJ-Count, on all the datasets. We can see that SWJ-Count consistently outperforms WJ-Count, since it represents bi-triangles as swj-units rather than wj-units, which avoids the enumeration of the excessively large number of vertex triplets as WJ-Count does. Besides, the fastest algorithm is RSWJ-Count, which is up to five orders of magnitude faster than the baseline algorithm WJ-Count. The reason is that it not only avoids the enumeration of vertex triplets, but also takes the advantage of the DegRank and the pruning technique of exploiting the (2, 2)-core (see Lemma 5).

[1] http://konect.uni-koblenz.de/networks

Table 2: Datasets used in the experiments.

Dataset	Id	Type	$\|U\|$	$\|L\|$	$\|E\|$	#wedges	#super-wedges	#r-super-wedges	#bi-triangles
DBPedia	D1	bibliographic	18, 422	168, 338	233, 286	1.45×10^{8}	3.23×10^{10}	1.94×10^{7}	3.62×10^{8}
Youtube	D2	social	30, 088	94, 239	293, 360	7.02×10^{7}	1.25×10^{9}	1.26×10^{8}	2.02×10^{10}
Actor-Movie	D3	affiliation	383, 641	127, 824	1, 470, 404	3.95×10^{7}	1.22×10^{9}	2.09×10^{8}	1.42×10^{9}
Twitter	D4	social	530, 419	175, 215	4, 664, 605	1.01×10^{9}	5.51×10^{10}	2.32×10^{9}	1.61×10^{12}
Discogs	D5	affiliation	1, 754, 824	270, 772	5, 302, 276	6.61×10^{9}	1.76×10^{12}	3.49×10^{10}	1.92×10^{14}
Movielens	D6	social	10, 678	69, 879	10, 000, 054	4.01×10^{10}	2.52×10^{13}	5.72×10^{12}	1.60×10^{18}
Dekiwi	D7	bibliographic	3, 119, 968	425, 842	26, 011, 353	9.54×10^{10}	2.80×10^{13}	3.81×10^{12}	2.08×10^{17}
Reuters	D8	text	781, 266	283, 912	60, 569, 726	2.04×10^{11}	3.26×10^{14}	6.71×10^{13}	2.46×10^{19}
Gottron	D9	text	556, 078	1, 173, 226	83, 629, 405	3.59×10^{11}	1.30×10^{19}	1.54×10^{14}	4.15×10^{20}
Livejournal	D10	affiliation	7, 489, 074	3, 201, 204	112, 307, 385	6.93×10^{10}	3.81×10^{13}	3.58×10^{12}	6.12×10^{18}
Trackers	D11	hyperlink	27, 665, 731	12, 756, 245	140, 613, 762	1.42×10^{17}	4.45×10^{20}	9.14×10^{17}	4.92×10^{28}
Syn	D12	synthetic	5, 000, 000	5, 000, 000	1, 100, 266, 304	6.98×10^{13}	2.25×10^{23}	5.16×10^{19}	6.55×10^{24}

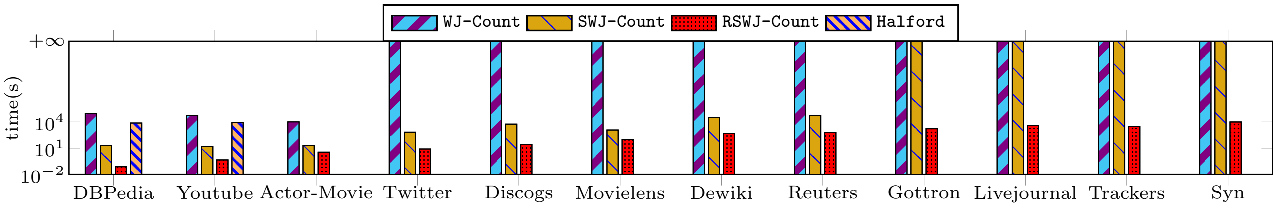
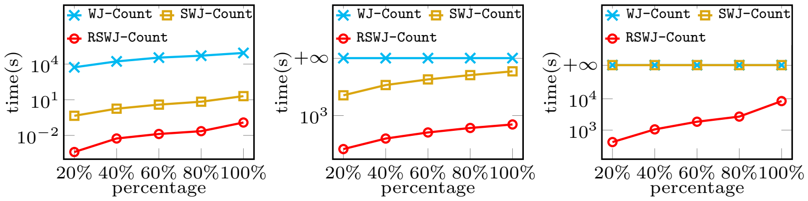


Figure 10: Efficiency of bi-triangle counting on all the 12 datasets.



(a) Youtube (b) Reuters (c) Syn

Figure 11: Scalability of bi-triangle counting algorithms.

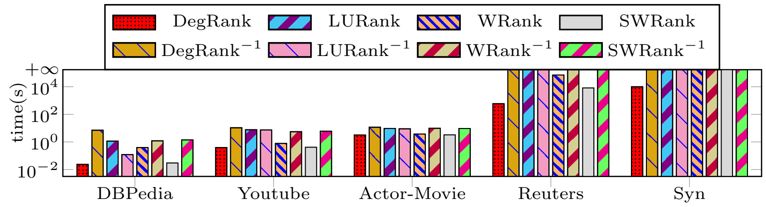
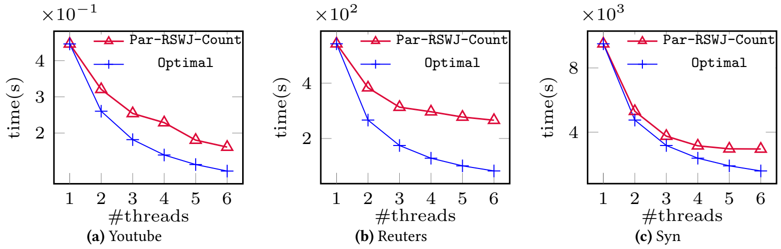


Figure 12: Comparison of different ranks in RSWJ-Count.



(a) Youtube (b) Reuters (c) Syn

Figure 14: Parallelizability of RSWJ-Count.

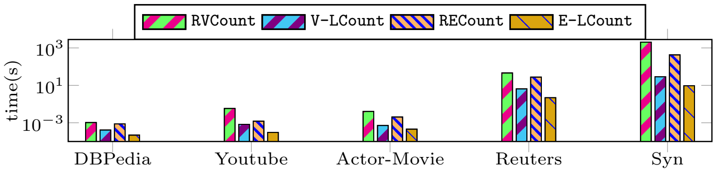
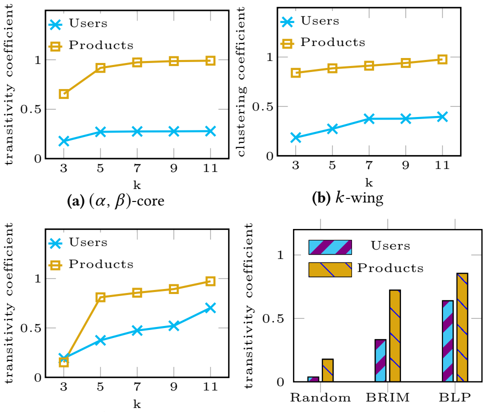
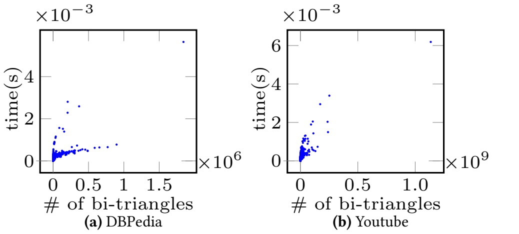


Figure 15: Efficiency of local bi-triangle counting algorithms.



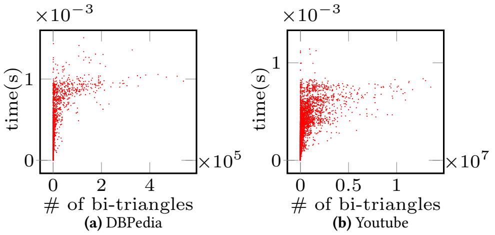
(a) (α, β)-core (b) k-wing (c) k-ctruss (d) Clustering algorithms

Figure 13: A case study.



(a) DBPedia (b) Youtube

Figure 16: Query time cost distribution of V-LCount.



(a) DBPedia (b) Youtube

Figure 17: Query time cost distribution of E-LCount.

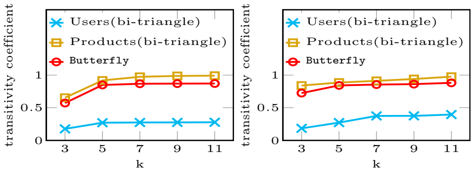
More specifically, WJ-Count can only finish within 48 hours on the three smallest datasets, i.e., DBPedia, Youtube, and Actor-Movie, because the number of vertex triplets needed to be enumerated grows significantly regarding the size of the dataset. In addition, SWJ-Count can process networks with millions of edges (e.g., Twitter, Discogs, Movielens, Dekiwi, and Reuters), while RSWJ-Count is able to process all the networks including the billion-scale network Syn, because these datasets contain many higher degree vertices, which cannot be well handled by SWJ-Count. In addition, we also analyse the relationship between the number of key factors and the running time of the algorithms. As the space limitation, we put this part in the Appendix [58].

In addition, we include a representative matrix based counting algorithm–Halford's algorithm for comparison. As there are many large matrices used in this algorithm, it can only process small datasets due to the memory limitation. For DBPedia and Youtube, we can see that Halford's algorithm is much slower than SWJ-Count and RSWJ-Count as it involves much matrix computation.

2. Scalability of bi-triangle counting algorithms. In this experiment, we test the scalability of the three counting algorithms. Specifically, for each real dataset, we randomly select 20%, 40%, 60%, 80%, and 100% of its edges and obtain five subgraphs induced by these edges, respectively. For the synthetic datasets, we generate smaller graphs where the numbers of the edges are 20%, 40%, 60%, and 80% of the Syn dataset, keeping other parameters the same. Then, we perform the bi-triangles counting over these small graphs using the three counting algorithms and depict the results in Figure 11. For lack of space, we only show the results on three datasets. We can see that generally, all algorithms scale well.

3. Comparison of different ranks in RSWJ-Count. To compare the efficiency of different ranks in RSWJ-Count, we experimentally test eight ranks, i.e., DegRank, DegRank^{-1}, LURank, LURank^{-1}, WRank, WRank^{-1}, SWRank, and SWRank^{-1}, where DegRank is used in RSWJ-Count, LURank puts vertices of L before vertice of U like what SWJ-Count does, WRank ranks vertices according to the numbers of wedges adjacent to them decreasingly, and SWRank ranks vertices according to the numbers of super-wedges adjacent to them decreasingly. Note that DegRank^{-1} denotes the reversed rank of DegRank and this notation has similar meanings for others. In the experiment, we run RSWJ-Count by only replacing its DegRank by each of the above ranks. Figure 12 depicts the results. Clearly, DegRank has the best performance, which confirms our observation in Section 5.

4. Investigation of transitivity coefficient (TC). In this experiment, we conduct a case study of the bi-triangle-based TC [37]. To do this, we collect a dataset of Amazon commerce networks which contains two types of vertices, i.e., users and products, and each edge represents a user giving a 5-stars rating for a product. Then, we run three classic bipartite network community detection algorithms, which are (α, β)*-core* [30], *k-wing* [53] and *k-Ctruss* [57], on it by varying k from 3 to 11, and use the TC [37] (See definition in Section 1) to measure the found communities. The results are shown in Figures 13(a)-(c). Clearly, as the k increases, the TC values for all the models increase continuously. Moreover, we include two additional classic bipartite network clustering algorithms which are *BRIM* [3] and *BLP* [31]. As reported in [31], *BLP* outperforms *BRIM*. We separately use *BRIM* and *BLP* to obtain clusters and compute the average TC over these clusters. Also, we randomly divide the network into several clusters and compute the average TC over these randomly generated clusters. The results are reported in Figure 13(d). We can see that the clusters found by *BLP* have the largest TC values while the random subgraphs have the lowest ones. Hence, the TC can well measure the quality of the communities or clusters.



(a) (α, β)-core

(b) k-wing

Figure 18: Comparing butterfly based TC and bi-triangle based TC.

In addition, we study the difference between the butterfly based TC [42] and bi-triangle based TC [37], where the butterfly based TC is defined as the ratio between the number of 3-paths and the number of butterflies. Generally, the bi-triangles based TC measures the cohesiveness for a set of vertices from a single layer only (e.g., either users or products in user-product networks), while butterfly based TC measures the cohesiveness of a whole bipartite network with two layers. In Figure 18, we show the values of butterfly based TC and the bi-triangle based TC on the user-product network. Clearly, the product vertices always have higher bi-triangle based TC. This is because a user may only buy a few products, but a product is often bought by many users, making products with higher cohesiveness. However, this difference between the two vertex layers cannot be captured by the butterfly based TC.

5. Parallelizability of RSWJ-Count. As our algorithms process each vertex independently, it is natural to parallelize them to achieve higher efficiency. Since WJ-Count and SWJ-Count inherently have higher time complexities, we focus on parallelizing RSWJ-Count. In specific, we parallelize the for-loop (lines 3-19) of Algorithm 3 by multiple threads such that different vertices are processed by different threads. We denote the parallelized version by Par-RSWJ-Count, which adopts the dynamic schedule, i.e., each vertex that has not been executed will be assigned to an idle thread. We run Par-RSWJ-Count by varying the number of threads and report the time cost in Figure 14. Further, we depict the line of the optimal parallelization which means the running time will be reduced to its $\frac{1}{k}$ when k cores are used. Generally, we can see that as the number of threads increases, the running time decreases continuously. However, we can also observe that Par-RSWJ-Count is still far from optimal. This is because we use dynamic scheduling, which is far from optimal [16]. Another reason is that we only parallelize the outer-loop; hence, the algorithm is not not fully parallized.

6. Efficiency of local bi-triangle counting algorithms. In this experiment, we evaluate local counting algorithms V-LCount and E-LCount which are presented in Section 6. Additionally, we include two more variants RVCount and RECount for counting bi-triangles contianing a specific vertex or edge. These two variants share the same idea of V-LCount and E-LCount, while the only difference is that the bi-triangles are regarded as rswj-units instead of swj-units. For each dataset we randomly generate 10,000 query vertices and 10,000 query edges, then run RVCount, V-LCount, RECount, and E-LCount for all of these query vertices and edges respectively, and finally report the average query response time for five datasets in Figure 15. Clearly, V-LCount and E-LCount are faster than RVCount and RECount, which verifies our intuition mentioned in Section 6. Also, V-LCount is slower than E-LCount, because a vertex is often involved in more bi-triangles than an edge.

In addition, we show how the numbers of bi-triangles containing the query vertex and query edge affect the efficiency. Specifically, we record the time cost for each query vertex or edge, and depict the time cost distribution of V-LCount and E-LCount on two datasets in Figures 16 and 17, respectively, where the horizontal and vertical axes denote the numbers of bi-triangles containing the query vertex/query edge and the time cost, respectively. Generally, vertices and edges that are contained in more bi-triangles take more time cost, since they tend to be linked with more super-wedges and swj-units, which results in more computational cost. However, there are some outliers that cost much computation time but are not contained in many bi-triangles. This is because these vertices and edges are contained in many acyclic swj-units but fewer bi-triangles.

8 RELATED WORKS

The structure of the bi-triangle was first introduced by Opsahl et al. [37]. Despite the wide usefulness of bi-triangles, no one has systematically studied the problem of bi-triangle counting yet. In the following, we review several groups of related works.

• *Butterfly counting in bipartite networks.* The butterfly is one of the most well-known motifs in bipartite networks. Conceptually, a butterfly is a complete 2×2 biclique. The problem of butterfly counting was first introduced by [2], since it is used in many bipartite network analysis methods [29, 34]. Vahid et al. [43] proposed both exact and randomized algorithms for counting butterflies. Recently, Wang et al. [51] have developed an improved algorithm based on the vertex priority, which is the state-of-the-art algorithm. Similar to Wang's algorithm, RSWJ-Count also incorporates a vertex priority mechanism, i.e., the deg-rank. However, counting bi-triangles meets more challenging invalid cases (i.e., acyclic rswj-units) which do not appear in counting butterflies. To tackle this issue, we propose the novel concept of the anchor-vertex, and then based on it, we can divide the invalid cases into four types such that each type can be counted easily during the process of counting all the valid cases. Besides, based on the butterfly, some novel cohesive subgraph models have been developed on bipartite networks. In [59], Zou et al. proposed the bitruss model and a bitruss decomposition algorithm. Later, Wang et al. developed an improved bitruss decomposition algorithm [53]. In [44], Ahmet and Ali proposed two cohesive subgraph models (i.e., k-tip and k-wing) using butterflies.

• *Motif counting on unipartite networks.* A motif (a.k.a graphlet or higher-order structure)is a small connected subgraph. The topic of motif counting on unipartite networks has received much attention [33, 40], since it has found various real applications (e.g., protein function prediction in biological networks and spam detection in email networks). Generally, existing solutions can be classified into two groups [40], namely network-centric methods [24, 25, 32, 38, 39, 45, 48, 54] and subgraph-centric methods [5, 6, 13, 14, 18], where network-centric methods assess the frequencies of all the motifs with k vertices in a graph while subgraph-centric methods only count the frequency of a specific subgraph. In the first group, both exact and approximation algorithms are extensively studied. For instance, Pinar et al. [39] proposed a counting framework which cuts motifs into smaller ones and gets the large counts using the counts of smaller motifs; Madhav et al. [25] and Wang et al. [54] proposed a randomized algorithm for counting motifs with four and over five vertices respectively. Since these methods count frequencies of a set of motifs, they cannot be directly used for our bi-triangle counting. For the second group, Floderus et al. [13] proposed algorithms for detecting and counting small motifs; Bressan et al. [6] proposed complexity bounds for counting a specifically sized subgraph. However, these algorithms often have high time complexities. Moreover, they are developed based on unipartite networks and not customized for a cycle with six vertices. Hence, we do not consider these algorithms in our experiments.

• *Subgraph enumeration on unipartite networks.* This topic, which aims to list all the subgraphs of a large network that are isomorphic to a query subgraph, is fundamental in network analysis. In [49], Ullmann proposed the first subgraph matching algorithm. Later, VF2 [8] and QuickSI [46] were proposed by considering the order of vertices, while Turbo$_{\text{ISO}}$ [19] was developed using the concept of neighborhood equivalence class to accelerate the computation. There are also many parallel algorithms, such as DualSim [26] and TwinTwigJoin [27]. DualSim enumerates subgraphs in parallel on a single machine, while TwinTwigJoin works in a distributed platform using MapReduce. Detailed surveys can be found in [28, 56].

9 CONCLUSION

In this paper, we examine the problem of bi-triangle counting over large bipartite networks, which has found many real applications such as computing transitivity coefficient and clustering coefficient. To count bi-triangles, we first propose an intuitive algorithm WJ-Count which represents the bi-triangle as a wj-unit, or the join of three wedges. We then propose a more efficient algorithm SWJ-Count, which adopts the concept of super-wedge and represents each bi-triangle as an swj-unit. Moreover, we optimize SWJ-Count by incorporating the DegRank over vertices and representing each bi-triangle as an rswj-unit. Besides, we develop two efficient local counting algorithms V-LCount and E-LCount for counting bi-triangles containing a given vertex or edge respectively. The experimental results on both real and synthetic large datasets show that our algorithms are efficient, and the best one is up to five orders of magnitude faster than the baseline algorithm.

In the future, we will investigate efficient parallel and distributed counting algorithms. Another interesting research direction is to develop efficient bi-triangle counting algorithms by exploiting modern hardwares (e.g., GPU). In addition, we will investigate some other related meaningful motifs over bipartite networks from the perspectives of counting and enumeration.

ACKNOWLEDGMENTS

The work was supported by the Australian Research Council Funding of DP200101338, DP180103096, and DP200101116.

REFERENCES

[1] Nesreen K. Ahmed, Jennifer Neville, Ryan A. Rossi, and Nick G. Duffield. 2015. Efficient Graphlet Counting for Large Networks. In *ICDM 2015, Atlantic City, NJ, USA, November 14-17, 2015*. 1–10.

[2] Sinan Aksoy, Tamara G. Kolda, and Ali Pinar. 2017. Measuring and modeling bipartite graphs with community structure. *J. Complex Networks* 5, 4 (2017).

[3] Michael J. Barber. 2007. Modularity and community detection in bipartite networks. *Phys. Rev. E* 76 (2007).

[4] Etienne Birmelé. 2009. A scale-free graph model based on bipartite graphs. *Discrete Applied Mathematics* 157, 10 (2009), 2267 – 2284. Networks in Computational Biology.

[5] Andreas Björklund, Petteri Kaski, and Lukasz Kowalik. 2017. Counting Thin Subgraphs via Packings Faster than Meet-in-the-Middle Time. *ACM Trans. Algorithms* 13, 4 (2017), 48:1–48:26.

[6] Marco Bressan. 2019. Faster Subgraph Counting in Sparse Graphs. In *IPEC*.

[7] Norishige Chiba and Takao Nishizeki. 1985. Arboricity and Subgraph Listing Algorithms. *SIAM J. Comput.* 14 (1985), 210–223.

[8] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE TPAMI* 26, 10 (2004).

[9] Jean-Pierre Eckmann and Elisha Moses. 2002. Curvature of co-links uncovers hidden thematic layers in the World Wide Web. *PNAS* (2002).

[10] Yixiang Fang, Xin Huang, Lu Qin, Ying Zhang, Wenjie Zhang, Reynold Cheng, and Xuemin Lin. 2020. A survey of community search over big graphs. *The VLDB Journal* (2020), 353–392.

[11] Yixiang Fang, Yixing Yang, Wenjie Zhang, Xuemin Lin, and Xin Cao. 2020. Effective and Efficient Community Search over Large Heterogeneous Information Networks. *Proc. VLDB Endow.* 13, 6 (2020), 854–867.

[12] Yixiang Fang, Kaiqiang Yu, Reynold Cheng, Laks V.S. Lakshmanan, and Xuemin Lin. 2019. Efficient algorithms for densest subgraph discovery. *PVLDB* 12, 11 (2019), 1719–1732.

[13] Peter Floderus, Mirosław Kowaluk, Andrzej Lingas, and Eva-Marta Lundell. 2013. Detecting and Counting Small Pattern Graphs. In *Algorithms and Computation*. Springer Berlin Heidelberg, 547–557.

[14] J. Flum and M. Grohe. 2002. The parameterized complexity of counting problems. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings*. 538–547.

[15] P. Frankl and V. Rödl. 1985. Near Perfect Coverings in Graphs and Hypergraphs. *European Journal of Combinatorics* 6, 4 (1985), 317 – 326.

[16] R. L. Graham. 1966. Bounds for certain multiprocessing anomalies. *The Bell System Technical Journal* 45, 9 (1966), 1563–1581.

[17] Oded Green, Pavan Yalamanchili, and Lluís-Miquel Munguía. [n.d.]. Fast triangle counting on the GPU. In *IA3 2014*. 1–8.

[18] Thomas R. Halford and Keith M. Chugg. 2006. An algorithm for counting short cycles in bipartite graphs. *IEEE Trans. Inf. Theory* 52, 1 (2006), 287–292.

[19] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turbo$_{\text{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. 337–348.

[20] Jiafeng Hu, Reynold Cheng, Kevin Chen-Chuan Chang, Aravind Sankar, Yixiang Fang, and Brian YH Lam. 2019. Discovering maximal motif cliques in large heterogeneous information networks. In *ICDE*. IEEE, 746–757.

[21] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-truss community in large and dynamic graphs. In *SIGMOD*.

[22] Xin Huang and Laks V. S. Lakshmanan. 2017. Attribute-Driven Community Search. *Proc. VLDB Endow.* 10, 9 (2017), 949–960. https://doi.org/10.14778/3099622.3099626

[23] Xin Huang, Wei Lu, and Laks V. S. Lakshmanan. 2016. Truss Decomposition of Probabilistic Graphs: Semantics and Algorithms. In *SIGMOD*. 77–90.

[24] Madhav Jha, C. Seshadhri, and Ali Pinar. 2013. A space efficient streaming algorithm for triangle counting using the birthday paradox. In *SIGKDD*.

[25] Madhav Jha, C. Seshadhri, and Ali Pinar. 2015. Path Sampling: A Fast and Provable Method for Estimating 4-Vertex Subgraph Counts. In *WWW*. 495–505.

[26] Hyeonji Kim, Juneyoung Lee, Sourav S. Bhowmick, Wook-Shin Han, Jeong-Hoon Lee, Seongyun Ko, and Moath H. A. Jarrah. 2016. DUALSIM: Parallel Subgraph Enumeration in a Massive Graph on a Single Machine. In *SIGMOD*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.).

[27] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. 2017. Scalable subgraph enumeration in MapReduce: a cost-oriented approach. *VLDB J.* 26, 3 (2017).

[28] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, et al. 2019. Distributed subgraph matching on timely dataflow. *PVLDB* (2019).

[29] Pedro Lind, Marta C. Gonzalez, and Hans Herrmann. 2005. Cycles and clustering in bipartite networks. *PRE* (2005).

[30] Boge Liu, Long Yuan, Xuemin Lin, Lu Qin, Wenjie Zhang, and Jingren Zhou. 2019. Efficient (a, &bgr;)-core Computation: an Index-based Approach. In *WWW*, Ling Liu, Ryen W. White, Amin Mantrach, Fabrizio Silvestri, Julian J. McAuley, Ricardo Baeza-Yates, and Leila Zia (Eds.).

[31] X. Liu and T. Murata. 2009. Community Detection in Large-Scale Bipartite Networks. In *2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology*.

[32] Chenhao Ma, Reynold Cheng, Laks V.S. Lakshmanan, Tobias Grubenmann, Yixiang Fang, and Xiaodong Li. 2019. LINC: a motif counting algorithm for uncertain graphs. *PVLDB* 13, 2 (2019), 155–168.

[33] Ali Masoudi-Nejad, Falk Schreiber, and Zahra Razaghi Moghadam Kashani. 2012. Building blocks of biological networks : a review on major network motif discovery algorithms. *IET Systems Biology* 6, 5 (2012), 164–174.

[34] Nathalie Del Vecchio Matthieu Latapy, Clemence Magnien. 2008. Basic notions for the analysis of large two-mode networks. *Social Networks* 30, 1 (2008).

[35] M. E. J. Newman. 2003. The structure and function of complex networks. *SIAM REVIEW* 45, 2 (06 2003), 167–256.

[36] M. E. J. Newman. 2005. Approximating Clustering Coefficient and Transitivity. *Journal of Graph Algorithms and Applications* 9, 2 (2005), 265–275.

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

[38] Roger Pearce. 2017. Triangle counting for scale-free graphs at scale in distributed memory. In *HPEC*. 1–4.

[39] Ali Pinar, C. Seshadhri, and Vaidyanathan Vishal. 2017. ESCAPE: Efficiently Counting All 5-Vertex Subgraphs. In *WWW*. 1431–1440.

[40] Pedro Manuel Pinto Ribeiro, Pedro Paredes, Miguel E.P. Silva, David Oliveira Aparício, and Fernando M. A. Silva. 2019. A Survey on Subgraph Counting: Concepts, Algorithms and Applications to Network Motifs and Graphlets. *ArXiv* abs/1910.13011 (2019).

[41] Garry Robins and Malcolm Alexander. 2004. Small Worlds Among Interlocking Directors: Network Structure and Distance in Bipartite Graphs. *Comput. Math. Organ. Theory* 10, 1 (2004), 69–94.

[42] G. Robins and M. Alexander. 2004. Small Worlds Among Interlocking Directors: Network Structure and Distance in Bipartite Graphs. *Computational and Mathematical Organization Theory* 10 (2004), 69–94.

[43] Seyed-Vahid Sanei-Mehri, Ahmet Erdem Sariyüce, and Srikanta Tirthapura. 2018. Butterfly Counting in Bipartite Networks. In *KDD*. 2150–2159.

[44] Ahmet Erdem Sariyüce and Ali Pinar. 2018. Peeling Bipartite Networks for Dense Subgraph Discovery. In *WSDM*. 504–512.

[45] C. Seshadhri, Ali Pinar, and Tamara G. Kolda. 2012. Fast Triangle Counting through Wedge Sampling. *CoRR* abs/1202.5230 (2012).

[46] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endow.* 1, 1 (2008), 364–375.

[47] Katherine Faust Stanley Wasserman. [n.d.]. *Social Network Analysis: Methods and Applications.*

[48] C. E. Tsourakakis, P. Drineas, E. Michelakis, I. Koutis, and C. Faloutsos. 2011. Spectral counting of triangles via element-wise sparsification and triangle-based link recommendation. *SNAM* 1, 2 (2011), 75–81.

[49] J. R. Ullmann. 1976. An algorithm for subgraph isomorphism. *JOURNAL OF THE ACM* 28, 1 (1976), 31–42.

[50] Kai Wang, Xin Cao, Xuemin Lin, Wenjie Zhang, and Lu Qin. 2018. Efficient computing of radius-bounded k-cores. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 233–244.

[51] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2019. Vertex Priority Based Butterfly Counting for Large-scale Bipartite Networks. *Proc. VLDB Endow.* 12, 10 (2019), 1139–1152.

[52] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2020. Efficient bitruss decomposition for large-scale bipartite graphs. In *ICDE*. IEEE, 661–672.

[53] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2020. Efficient Bitruss Decomposition for Large-scale Bipartite Graphs. In *ICDE 2020*. 661–672.

[54] Pinghui Wang, Junzhou Zhao, Xiangliang Zhang, Zhenguo Li, Jiefeng Cheng, John C. S. Lui, Don Towsley, Jing Tao, and Xiaohong Guan. 2018. MOSS-5: A Fast Method of Approximating Counts of 5-Node Graphlets in Large Graphs. *IEEE Trans. Knowl. Data Eng.* 30, 1 (2018), 73–86.

[55] Taotao Wang and Soung Chang Liew. 2014. Joint Channel Estimation and Channel Decoding in Physical-Layer Network Coding Systems: An EM-BP Factor Graph Framework. *IEEE Trans. Wireless Communications* 13, 4 (2014), 2229–2245.

[56] Junchi Yan, Xu-Cheng Yin, Weiyao Lin, Cheng Deng, Hongyuan Zha, and Xiaokang Yang. 2016. A short survey of recent advances in graph matching. In *ICMR*. 167–174.

[57] Yixing Yang, Yixiang Fang, Xuemin Lin, and Wenjie Zhang. 2020. Effective and Efficient Truss Computation over Large Heterogeneous Information Networks. In *ICDE*. 901–912.

[58] Yixing Yang, Yixiang Fang, Maria E. Orlowska, Wenjie Zhang, Xuemin Lin. [n.d.]. Efficient Bi-triangle Counting for Large Bipartite Networks (Technical report). http://www.cse.unsw.edu.au/~z5151364/technique_report.pdf.

[59] Zhaonian Zou. 2016. Bitruss decomposition of bipartite graphs. In *International Conference on Database Systems for Advanced Applications*. Springer, 218–233.