Noname manuscript No.
(will be inserted by the editor)

Truss Decomposition using Triangle Graphs

Mohsen Rezvani* ·
Mojtaba Rezvani

Received: date / Accepted: date

Abstract Recent studies have shown that social networks exhibit interesting characteristics such as community structures, i.e., vertexes can be clustered into communities that are densely connected together and loosely connected to other vertices. In order to identify communities, several definitions were proposed that can characterize the density of connections among vertices in the networks. Dense triangle cores, also known as k-trusses, are subgraphs in which every edge participates at least $k-2$ triangles (a clique of size 3), exhibiting a high degree of cohesiveness among vertices. There are a number of research works that propose k-truss decomposition algorithms. However, existing in-memory algorithms for computing k-truss are inefficient for handling today's massive networks. In this paper, we propose an efficient, yet scalable algorithm for finding k-trusses in a large-scale network. To this end, we propose a new structure, called triangle graph to speed up the process of finding the k-trusses and prove the correctness and efficiency of our method. We also evaluate the performance of the proposed algorithms through extensive experiments using real-world networks. The results of comprehensive experiments show that the proposed algorithms outperform the state-of-the-art methods by several orders of magnitudes in running time.

Keywords Truss Decomposition · Triangle Graph · Community Detection · Social Networks

Mohsen Rezvani (Corresponding Author)
Faculty of Computer Engineering, Shahrood University of Technology, Shahrood, Iran
E-mail: mrezvani@shahroodut.ac.ir

Mojtaba Rezvani
Australian National University, College of Engineering and Computer Science, Canberra, ACT, 2601, Australia
E-mail: mojtaba.rezvani@anu.edu.au

1 Introduction

In recent years, we have observed an exponential growth in the size of networks in many applications. World Wide Web is expanding rapidly owing to popularity of blogs and web publishing tools, social networks are attracting more people to make the world more connected and provide new opportunities for people to communicate, and bio-informatic researchers are gathering more data about biological networks. In all these networks, looking at the networks and relationships can give us useful information that cannot be easily gained by studying the individuals [1]. Due to the massive size of real-world networks, it has become very difficult to study these networks and gain useful insights about them.

Within these networks, lie some dense subgraphs —a set of nodes that the number of edges between them is greater than the number of edges between these nodes and other nodes. These subgraphs are usually referred to as the cohesive subgraphs and give us an idea about the individuals sharing the same attributes or interests. These subgraphs have long known to be communities [2,3]. Given a graph G, a community is a component of G, a group of vertices which are densely connected internally, and satisfy two properties: 1) *connectivity*, i.e., vertices in the community are connected; and (2) *cohesiveness*, i.e., vertices in the community are intensively linked to each other with respect to a particular goodness metric [3]. For example in a social network, a community can contain people having the same interests in a particular product and having a good chance for targeted advertisement. In a protein network, these communities can contain proteins that have a common function, say a type of cancer, and it provides a new hypothesis for creating new medicine.

Finding good quality communities, i.e., subgraphs that clearly reflect the attributes that we are looking for in the network is very challenging. In recent years, many papers addressed this problem and propose meaningful models of community structure in networks [3]. As a result, some dense subgraph structures were proposed such as k-clique, k-plex, quasi-clique, k-core [4,1], but they are either very time consuming or fail to guarantee a strong connectivity between nodes. Dense triangle cores, also known as k-trusses, are subgraphs where every edge belongs to at least $k-2$ triangles (a clique of size 3) in the subgraph, exhibiting a high degree of cohesiveness among vertices. More formally, a k-truss, $k \in \{3,4,\ldots\}$ for graph $G(V,E)$ is a connected subgraph $G'(V',E')$, $V' \subset V$, $E' \subset E$, such that each edge in G' includes in at least $(k-2)$ triangles [5]. K-trusses were proposed as a structure that can be computed relatively efficiently and guarantee a strong connectivity; k-trusses also help us ranking the nodes in terms of clustering coefficients scores that is proposed to find the affiliation of each individual to a community [5–7].

Despite the very useful applications of k-trusses, computing these subgraphs is a challenging task in large networks. Most papers in this literature have came up with ways to count the number of triangles, and very few have found the communities in the graph efficiently [8]. However, a real social network contains hundreds of millions of nodes which makes it very difficult to

process [9]. Although extensive research was carried out on finding k-trusses, there is still space to improve the efficiency of computing them.

In this paper, we propose a novel algorithm for computing the k-trusses in a graph and conduct experiments in real social networks to evaluate the performance of our proposed algorithm compared with the other methods. We propose a new structure which leverages triangle graphs to speed up the process of finding the k-trusses and we also prove the correctness and efficiency of the proposed method.

There are several major differences between our proposed algorithm and the previous ones:

- This algorithm finds k-trusses based on computing the number of triangles in the graph.
- Our algorithm is capable of being run in parallel environment for finding the common neighbourhood of endpoints of each edge and constructing triangle graph (In the previous studies, this procedure cannot be computed in parallel environment, because the edge removals happen while counting the number of triangles, which leads to conflicts for computing further support numbers).
- Instead of finding k-trusses for every possible values of k, we focus on identifying the k-truss for a given value of k. Experimental results show that this algorithm outperforms the other methods even in the case of finding trusses for every possible value of k. Once we have created the triangle graph, we can quickly find the k-truss for every value of k quickly.
- The two phases of the proposed algorithm can be computed using I/O efficient algorithms that are previously proposed for counting triangles and core decomposition [10].

We provide a comparative evaluation of the performance of our algorithms against state-of-the-art algorithms in truss decomposition using some real datasets. The results show that our method significantly improves the efficiency of the truss decomposition by reducing the processing time compared to the method proposed in [7].

The rest of this paper is organized as follows. Section 2 presents the related work. Section 3 explains some background and basic concepts. Section 4 describes the details of our truss decomposition algorithms. Section 5 describes our experimental results. Finally, the paper is concluded in Section 6.

2 Related Works

In a large-scale networks such as a social network and depending on motivations and application scenarios, there are several structural definitions of communities such as cliques, k-cliques, k-clans, k-plexes, k-core, and quasi-cliques [11].

A clique is a subgraph whose vertex is adjacent to every vertex in it. However, the condition of clique is very strict and it is very unlikely to find a real

community that all of its members are connected to each other. A k-clique means the k nodes connect to each other, whereas a quasi-clique means the number of neighbours of each node is no less than a proportional threshold. An k-clan is a k-clique with diameter no greater than k. A k-plex is a maximal subgraph in which every vertex of the induced subgraph is connected to at least $n - k$ other vertices, where n is the number of vertices in the induced subgraph. A k-core is a subgraph in which the degree of each node is at least k.

The key issue on community with degree-based models is that the members in each community have weak connectivity, i.e., they can be disconnected by removing a small number of edges. Since edge connectivity is a major concern in the formation of close communities, a great deal of effort is devoted to find tightly connected subgraphs that cannot be disconnected by removing only a few number of edges. The k-truss can guarantee a strong edge connectivity in graph since they are $(k + 1)$-edge-connected – won't be disconnected by removing less than $k + 1$ edges.

In community detection research, a large and growing body of literature studied the triangle counting problem in graphs. Some of the papers focused only on counting the number of triangles [12–18] while others focused on finding k-trusses in a graph [5, 7, 19–22]. Our main focus in this paper is finding k-trusses, and in the following, we review the recent works on the k-truss problem.

Very closely related to our work is in-memory truss decomposition algorithms [5, 7]. Cohen's paper [5] who was the first to introduce the k-truss concept as a cohesive subgraph and proposed an $O(\sum_{v \in V} d^2(v))$ time algorithm, where $d(v)$ is the degree of a vertex v. The work employed a hash dictionary every time for finding the common neighborhood of two endpoints of each edge and remove that edge if the support number of that edge is less that k. Cohen also proposed another algorithm [20] for calculating the k-trusses based on map-reduce technique. This approach although feasible on large graphs, repetitively calculates the support value of all the edges in the partition and makes the algorithm time consuming. Wang et. al [7] modified the first algorithm proposed by Cohen, and proposed a more efficient algorithm that performs better while computing the common neighborhood of the graph and propose two I/O efficient algorithms for computing k-trusses for every possible k until the graph is empty. Since for larger values of k, the k-trusses are only subset of smaller ones, some of these outputs are not useful in many applications, so this algorithm is performing the BottomUp and TopDown operations for some values of k that are not useful. They also claimed to improve the truss decomposition algorithm by proposing an $O(|E|^{3/2})$ algorithm, where $|E|$ is the number of edges in the graph, but there is no convincing proof of the claimed time complexity. Recently, Habib et. al [23] proposed an algorithm to discover the top-r weighted k-truss communities, which are k-truss community with the highest weight. Jiang et. al [24] presented a compact index structure to provide an efficient search of k-truss communities with a linear cost with respect to the community size. They also proposed an I/O-efficient algorithm

for query processing under the semi-external model. In this paper, we compare the performance of our approaches against the method proposed in [7].

Some of the papers presented in the literature proposed I/O efficient algorithms for finding the k-truss in graphs that do not fit in the RAM [6,19], and some suggested parallel algorithms for very large graphs [20–22,25,26]. One of the first papers proposing a parallel algorithm for truss decomposition is [6] which was one of the finalists of the 2017 GraphChallenge [27]. Conte et. al [28,25] proposed two parallel algorithms, EXTRUSS and HYBTRUSS, for truss decomposition on massive networks which has been finalist of the 2018 GraphChallenge for k-truss decomposition. At the same round of GraphChallenge, the distributed algorithm proposed by Pearce and Sanders [9] has been champion. Almasri et. al [26] presented a scalable multi-GPU algorithm for truss decomposition in which each GPU handles a different k value. Although these methods are feasible, the base framework that they follow has a time complexity of $O(\sum_{v \in V} d^2(v))$ and there is room to improve them.

3 Preliminaries

Let $G = (V, E)$ be a simple undirected graph, where V is the set of vertices and E is the set of edges. Let $N_G(v)$ be the set of vertices that are adjacent to vertex $v \in V$ in G. The degree of a vertex v is defined as the number of edges incident to it, i.e. $d_G(v) = |N_G(v)|$. A graph $G = (V, E)$ is a *k-core* if the degree of each vertex in it is at least k. A k-core component of G is a maximal induced subgraphs of G in which the degree of each vertex is at least k. A *triangle* Δ_{uvw} in G consists of three edges $\{(u, v), (v, w), (u, w)\} \subseteq E$. We use $\Delta_{uvw} \in G$ to indicate the existence of a triangle in G that connect vertices u, v and w to each other. Given two edges (u, v) and (v, w), we use the *exclusive or* operation on two edges to refer to the third edge that forms a triangle with them, i.e. $(u, v) \oplus (v, w) = (u, w)$. Since the edges are not directed in our graph, we make no difference between (u, v), (v, u) or $\{u, v\}$ in this paper.

For every edge $e = (u, v) \in E$ in graph G, its *support*, represented by $sup_G(e)$, is defined as the number of triangles in G containing e that can be computed by counting the common neighbours of its endpoint vertices u and v, i.e. $|N_G(u) \cap N_G(v)|$. A high support value of an edge indicates the closeness of its two vertices to each other in a network. Similarly, a small value of support of an edge indicates a weak connection between two vertices or between two groups of vertices. Recent studies employed the support value of edges to identify weak connections and find communities by removing those weak links of a given network. One of the interesting community structures based on the support is defined as follows,

Definition 1 *A graph $G = (V, E)$ is a* k-truss *if the support of each of its edges is at least k.*

Definition 2 *Given a graph $G = (V, E)$,* k-truss components *of G is the set of maximal subgraph(s) of G, in which the support of each edge is at least k.*

The *k-truss component decomposition problem* in a graph G is the problem of finding the set of k-truss components of G. A k-truss has several density characteristics such as minimum degree. The following lemma highlights the relationship between k-truss and $k+2$-core, which has a minimum degree of $k+1$ among degrees.

Lemma 1 *Each k-truss component in G is also a $(k+1)$-core. But a $(k+1)$-core is not necessarily a k-truss*[1].

In the following, we review the baseline algorithm for k-truss detection in networks and then we introduce two novel algorithms in Section 4 that use an auxiliary data structure to efficiency identify k-truss components in large-scale networks.

3.1 Baseline algorithm for k-truss detection

Before proceeding, we prove the following lemma, which provides us with an algorithm for k-truss decomposition of a given graph.

Lemma 2 *Given a graph $G = (V, E)$ and an edge $e \in E$ with $sup_G(e) < k$, the k-truss of G is the same as k-truss of $G_1 = G \setminus \{e\}$.*

Proof Since the support of e is strictly less than k, it does not belong to any k-truss component of G. So, all the edges that are in k-truss components of G, are still in k-truss components of G_1. On the other hand, since there is no new edges in G_1, there cannot be an edge in k-truss of G_1 which is not in k-truss of G.

The following lemma states the k-truss components of a graph G do not share any edge or vertex.

Lemma 3 *k-truss components in a graph $G = (V, E)$ are disjoint*[2].

Corollary 4 *Given a graph $G = (V, E)$, let $G_0 = G$ and $G_i = G \setminus \bigcup_{j=1}^{j \leq i} \{e_j\}$ where e_j is an edge whose support is less than k in G_{j-1}. The k-truss components of G are identical to the k-truss components of G_i.*

This corollary suggests an algorithm for finding the k-truss of a given graph. In each iteration of this algorithm, one edge from G_{i-1} whose support is strictly less than k is removed in G_i, until there is no such an edge. Finally, the graph that remains is the k-truss.

The baseline algorithm for k-truss detection iteratively removes those edges whose support is no larger than k, until the support of all edges in the residual graph is no less than k. Let i be the first iteration of the algorithm, initially set to $i = 1$, and let G_i be the residual graph in iteration i, where $G_0 = G$. Let S_i be the set of edges in G_0 whose support is no larger than k. The algorithm iteratively removes the set of edges S_i from graph G_i, and forms the graph G_{i+1} and the set of edges S_{i+1}, i.e. $G_{i+1} = G_i \setminus S_i$. The algorithm continues finding G_i and S_i, until S_i is empty and G_i is a k-truss.

[1] The proof can be found in [7].

[2] The proof follows immediately from the properties of k-truss presented in [7,5,1]

4 Efficient k-truss detection

In this section, we propose two algorithms to efficiently decompose a large graph into its k-truss components. The main idea of our approach is to construct an efficient auxiliary graph from the triangles of the original graph G. After that, we employ the new graph to find the k-truss of G in linear time. In the rest of this section, we propose two types of graph constructed from the triangles of a graph G. We first introduce the Triangle Graph in Section 4.1, we then prove the properties of this graph. After that, we present another version of the graph, called *Bipartite Triangle Graph* in Section 4.6. We show that the proposed decomposition algorithm can be applied on both graphs, while the first one is more compact and efficient.

4.1 Definition of Triangle Graph

In this section we define the notion of *triangle graphs* and devise a new algorithm that finds the k-trusses using the triangle graph. We use this novel data structure to find the k-truss of G in linear time, in terms of the size of the triangle graph. To this end, we repeatedly remove those vertices from the triangle graph are corresponding to the removal edges in the original graph. We then show that employing the triangle graph improves the complexity of our approach for the k-truss decomposition problem. We also prove the correctness of our algorithm and show that its time and space complexity is $O(|E|^{1.5})$.

Definition 3 *Given a graph $G = (V, E)$, the* triangle graph $G' = (V', E')$ *of G is a graph that has one vertex for every edge in G, and between any two vertices (u, v) and (u, w) of G' there is an edge if they form a triangle Δ_{uvw} by a third edge (v, w) in G, i.e.,*

$$V' = E$$

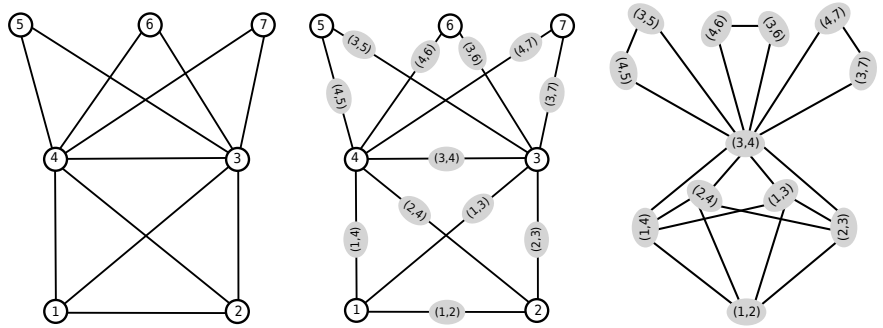
$$E' = \{((u, v), (v, w)), ((u, v), (u, w)), ((v, w), (u, w)) : \forall \Delta_{uvw} \in G\}$$

The following example demonstrates how the triangle graph of a graph looks like.

Example 1 *Fig. 1 shows an example of a graph and its triangle graph.*

4.2 Properties of Triangle Graph

The proposed triangle graph has several characteristics. One of the essential characteristics of the triangle graph G' of a graph G is that for every triangle in G, there are three edges in its triangle graph G'. In the following the exact relationship between the number of triangles formed by each edge in G and the degree of its corresponding vertex in G' is described.



(a) Original graph G.

(b) Original graph G with a label for each edge.

(c) The triangle graph corresponding to the original graph G.

Fig. 1: An example of a graph and its corresponding triangle graph. In order to make a triangle graph from graph G, shown in (a), we first add a simple label on each edge of the graph, as shown in (b). After that, we make a new graph G', as shown in (c) which is a triangle graph corresponding to G.

Property 5 *Given a graph $G = (V, E)$, the degree of every vertex in the triangle graph $G' = (V', E')$ of the graph G is exactly two times the support of the corresponding edge in G, i.e.,*

$$deg_{G'}((u, v)) = 2 \times sup_G((u, v)). \tag{1}$$

As a result, the following lemma states the relationship between a k-truss graph and its k-core triangle graph.

Lemma 6 *A graph $G = (V, E)$ is k-truss if and only if its triangle graph is a $2k$-core.*

Proof We first prove that if a graph $G = (V, E)$ is k-truss, then its triangle graph is a $2k$-core. According to Definition 1, since G is k-truss, the support of each of its edges is at least k. Thus, the degree of every vertex in the triangle graph G' corresponding to G is at least $2 \times k$, according to Property 5. Therefore, the triangle graph G' is a $2k$-core graph.

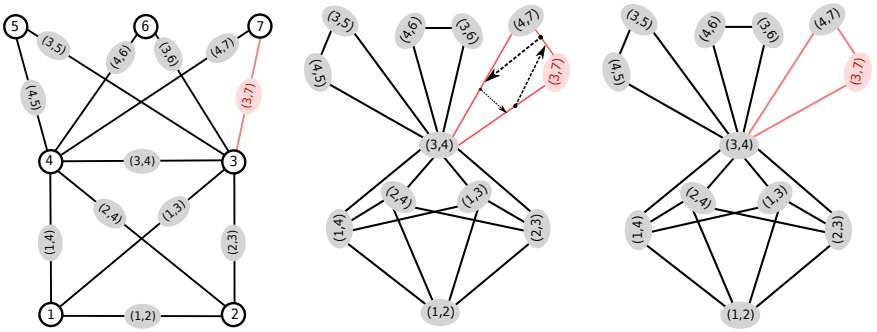
Now, we prove that if the triangle graph is a $2k$-core, then the original graph is k-truss. Since the triangle graph G' is a $2k$-core, the degree of every vertex in G' is at least $2 \times k$. Thus, every edge in the original graph G has the support of at least k. Therefore, the original graph G is k-truss.

4.3 Vertex Removal Algorithm

In spite of the close relationship between the set of vertices in G' and the set of edges in G, the removal of an edge in G is not equivalent to the removal of the corresponding vertex in G'. More specifically, the Property 5 will not be

maintained between the new triangle graph and the new graph. The reason is that after removal of a vertex (u, v) from G', there may still exist some edges between other vertices that correspond to triangles that were formed in G by the edge (u, v) and they were collapsed upon the removal of this edge. The following example illustrates the comparison between the removal of an edge in G and the removal of the corresponding vertex in G'.

Example 2 *Here is a comparison between the removal of an edge in G and the removal of the corresponding vertex in G'. Let assume the original graph G shown in Fig. 2a and we want to remove the edge $(3, 7)$ from G. By removing the corresponding edge in G', we reach to the graph shown in Fig. 2b. However, such removal does not maintain Property 5, as shown in Fig. 2b. In order to maintain the property, we must remove both edges in G' generated from edge $(3, 7)$ in G. Fig. 2c shows the resulted graph after removing the three corresponding edges from G'.*



(a) Original graph G.

(b) Removing the corresponding vertex from G'.

(c) Removing the edge properly from G'.

Fig. 2: Removal of an edge in G is not necessarily equivalent to removing the corresponding vertex in G'. Since edge $(3, 7)$ participates in a triangle $\Delta_{3,4,7}$ in G, the removal of this edge from G is equivalent of removing three edges in the triangle graph G' as follows: $((3.4), (3, 7))$, $((3, 7), (4, 7))$ and $((3, 4), (4, 7))$, as well as vertex $(3, 7)$.

Therefore, we propose Algorithm 1 for vertex removal operation in G' which manages to maintain the Property 5. As one can see in this algorithm, in order to remove a vertex p from triangle graph G', we influence such removal to all the neighbours of v. More specificity, we have to remove all the edges initiated or targeted to v. Removing every single edges form the triangle graph G' needs to remove all the edges in the same triangle with such an edge. It is to be noted that we use vector color in this algorithm to keep the status of every node in the triangle graph. Moreover, the initial color for all vertex in the graph is set to *white*. For example, when we remove edge $((3, 7), (4, 7))$ from graph shown in Fig. 2b, we must remove both edges $((4, 7), (3, 4))$ and $((3, 7), (3, 4))$, as they

all three were in a same triangle in the original graph shown in Fig. 2a. As shown in Algorithm 1, we can find the pointers to two of these three edges ((p, v)) by traversing the adjacency list of the removal vertex (p). However, finding the pointer to the third edge is not easily accessible from the removal vertex. In order to address this challenge, we use an extra pointer in the data structure of each edge in the triangle graph which points to the next edge generated in this graph because of existing in a similar triangle in the original graph. For example in graph shown in Fig. 2b, edge $((3, 7), (4, 7))$ has a pointer to edge $((4, 7), (3, 4))$, edge $((4, 7), (3, 4))$ has a pointer to edge $((3, 4), (3, 7))$, and $((3, 4), (3, 7))$ has a pointer to edge $((3, 7), (4, 7))$. Such pointers are shown using dashed lines in this figure. These pointers help us to find all the edges of a single triangle required to be removed using an algorithm with a time complexity in $O(1)$. We refer to these pointers by the nextLinkInTriangle function at line 7 in Algorithm 1.

Algorithm 1 Removing a vertex from a triangle graph.

Input: A triangle graph $G' = (V', E')$ and a vertex $p \in V'$
{T}he initial color for all vertex in the graph is set to *white*
Output: Remove vertex p from the triangle graph G'

```
 1: for each v ∈ N_{G'}(p), where (p, v) ∈ E' do
 2:     /* remove vertex v */
 3:     decrement the degree of v
 4:     if color[v]! = black then
 5:         color[v] ← gray
 6:     /* remove the triangle next link of edge (p, v) */
 7:     (v, w) ← nextLinkInTriangle(p, v)
 8:     if color[(v, w)] == white then
 9:         /* remove vertex w */
10:         decrement the degree of w
11:         if color[w]! = black then
12:             color[w] ← gray
13:         color[(v, w)] ← black
14:     color[(p, v)] ← black
15: color[p] ← black
```

Now we prove that our procedure for removing a vertex in Algorithm 1 maintains the relationship between G and G' presented in Property 5. To this end, we use notation G_i that was explained in Corollary 4, to represent an equivalence between iterations of our algorithm with the one explained in that Corollary. In order to prove the correctness of the algorithm, we show that at iteration i of our algorithm, the degree of every non-black vertex $(u, v) \in V'$ is in degree of $2 \times sup_{G_i}(u, v)$ and we use these degrees to determine which nodes to enqueue.

Lemma 7 *[Loop Invariant] After each iteration i of our algorithm, the degree of every non-black vertex $(u, v) \in V'$ is $2 \times sup_{G_i}(u, v)$, where G_i is the graph G after removal of every edge whose corresponding node in G' is black.*

Proof Suppose S_i is the set of all the nodes whose color is black after iteration i. In the beginning of the algorithm, $S_0 = \emptyset$ and the degree of each node is $deg(u, v) = 2 \times sup_G(u, v)$.

Assume after i-th iteration, the loop invariant holds. We prove that after $(i + 1)$-th iteration, the loop invariant still holds.

Suppose $(i+1)$-th iteration, the vertex $(u, v) \in V'$ is dequeued and $S_{i+1} = S_i \cup \{(u, v)\}$;

According to the initial construction of the triangle graph, (u, v) is only connected to the vertices in V' whose corresponding edge form a triangle with (u, v) in G. During $(i + 1)$-th iteration, we visit every neighbour of (u, v) in the form of either (u, w) or (v, w) and change their degree if their support is changed in $G_{i+1} = G_i \backslash (u, v)$. When we visit a non-black node $(u, w) \in V'$ as a neighbour of (u, v), three cases happen:

1. colour[(v, w)]=white: In this case, (v, w) is neither added to the queue (grey) nor removed from G (black) in G_i. In fact, this implies that three edges (u, v), (u, w) and (v, w) are all present in G_i and form a triangle $\Delta_{uvw} \in G_i$. Thus after removing (u, v), the triangle Δ_{uvw} is collapsed and the support of (u, w) in G_{i+1} is decreased by one and therefore, the algorithm decreases the degree of (u, w) by 2 and invariant holds.
2. colour[(v, w)]=gray: In this situation, the degree of (v, w) is strictly less than $2k$ and it is added to queue for checking its neighbours. The edge (v, w) is not removed from G_i yet, and the triangle Δ_{uvw} is in G_i. Therefore, after removal of (u, v), the support of (u, w) will be decreased by 1, and accordingly, the degree of its vertex in G' is decreased by 2 in our algorithm.
3. colour[(v, w)]=black: This means that the edge (v, w) is removed from G_i in the previous iterations and the triangle Δ_{uvw} does not exist in G_i. In this case, to maintain the invariance, the algorithm does not change the degree of node (u, w) in G'.

In our proposed algorithm, all of the above conditions are considered and after the $(i + 1)$-th iteration, the lemma still holds. Thus, we can claim that after the last iteration of the algorithm, the lemma still holds.

Lemma 8 *The proposed vertex removal operation maintains the Property 5 between G' and G, i.e.,*

$$\forall (u, v) \in E \quad deg_{G'}((u, v)) = 2 \times sup_G((u, v)). \tag{2}$$

Proof The proof is very similar to the proof of Lemma 7.

Lemma 9 *Given a graph G and a positive integer k, after running our algorithm, the colour of any node in G' is white if and only if the corresponding edge is in the k-truss of G.*

Proof We first prove that if the colour of a vertex (u, v) is black in G', then it is not in the k-truss, and then we proceed by proving that if an edge is not in the k-truss of G, its corresponding node becomes black in our algorithm.

As the Lemma 7 implies, at each iteration i of the algorithm, the degree of a node in G' is equal to two times support of the corresponding edge in G_i. Each node will be enqueued to Q if its degree is strictly less than $2k$, which implies that the corresponding edge forms less than k triangles in G_i, and thus is not in the k-truss of G. Therefore, if the colour of a vertex (u, v) is black in G', then the corresponding edge is not in the k-truss of G_i or G.

Next, we prove that if an edge is not in the k-truss, the colour of the corresponding node in G' will be changed to gray and consequently to black.

Initially, the degree of every node (u, v) in G' is $2 \times sup_G(u, v)$. In steps 8-12 of the algorithm, if the degree of any node is strictly less than $2k$ we enqueue that node and change its colour to gray. Suppose that at the i-th iteration, we enqueue every node whose corresponding edge has a support less than k in G_i and change its colour to gray. we prove that in the $(i + 1)$-th iteration, we do not miss any such node. Presume there exist an edge (u, v) whose support is less than k in G_{i+1} and we do not enqueue it in $(i + 1)$-th iteration, only two cases happen:

1. The colour of this node is not white; In this case, this node has been enqueued in the previous iterations.
2. The colour of this node is white; Since this node has not been enqueued in this iteration, its degree must have been at least $2k$. Since we claimed that the support of this node is less than k, thus, we face a contradiction with Lemma 7 that states $deg_{G'}[(u, v)] = 2 \times sup_{G_{i+1}}(u, v)$.

Therefore, if an edge is not in the k-truss, its colour will be changed to grey and consequently to black.

So far, we proved the necessary and sufficient conditions for our algorithm to be correct.

Theorem 10 *The proposed algorithm finds the k-truss of a graph G.*

Proof This theorem follows from Lemma 9. In each iteration of the algorithm, one vertex is dequeued from Q and any vertex cannot be in Q twice (due to its colour change); Thus, the algorithm terminates after a finite number of iterations which is less than $|E'|$.

4.4 Algorithm Overview

Our approach consists of three phases: 1- Removing all the nodes whose degree is less than $k + 1$ from G, 2- Constructing the triangle graph G', 3- Colouring the nodes of G' whose corresponding edge in G has a support strictly less than k, iteratively.

Let $G = (V, E)$ be an undirected simple graph. Since the time complexity of computing the $(k+1)$-core is $O(|E|)$, and each k-truss is necessarily a $(k+1)$-core, first we find all $(k + 1)$-cores in the graph to remove a large portion of nodes and edges that are not in the k-truss.

It is obvious that the second phase, constructing the triangle graph, is the bottle-neck of our algorithm, thus, we use an efficient triangle listing algorithm proposed by Schank et al. [29] to construct the triangle graph. More specifically, for every triangle that this algorithm finds in G, we add three edges to G' that connect the nodes whose corresponding edges are in that triangle.

In the third phase of this algorithm, we iteratively change the colour of the nodes in G' whose corresponding edge in G has a support less than k. Initially, every node $(u, v) \in V'$ is white. Once we find a node (u, v) whose degree is less than $2k$, we enqueue that node and changes its colour to gray. In each iteration, one node whose colour is gray is picked from queue and its colour is changed to black and the degree of its neighbours is updated.

In each iteration, one node $(u, v) \in V'$ whose degree is strictly less than $2k$ (and the colour is gray) is dequeued and its colour is changed to black. When we change the colour of a node $(u, v) \in V'$ to black, we check any two neighbours $(u, w), (v, w) \in N_{G'}((u, v))$ to see whether one of them is black or not. If none of them were black, we decrease their degree by two and add them to the queue if their colour is white (haven't been in the queue before). We repeat this set of instructions until the queue contains no more node and the degree of any node in the resulting graph G' is at least $2k$. Algorithm 2 shows the details of this iterative procedure.

Algorithm 2 k-truss component decomposition of G

Input: $G = (V, E), k$
Output: k-truss components of G
1: /* Construct the triangle graph */
2: $G' = (V', E') \leftarrow T(G)$;
3: **for** each $(u, v) \in V'$ **do**
4: $deg[(u, v)] \leftarrow$ degree of (u, v) in G';
5: **while** $\exists (u, v) \in V'$ with $deg[(u, v)] < 2k$ **do**
6: Perform the vertex removal operation on (u, v) in G';
7: **for** each $(x, y) \in N_{G'}((u, v))$ **do**
8: $\deg[(x, y)] \leftarrow \deg[(x, y)] - 2$

We also argue that the proposed algorithm is capable of being run in parallel for finding the common neighbourhood of endpoints of each edge and constructing the triangle graph. Specifically, the algorithm consists of a parallelisable operation for counting the number of triangles that include an edge e. Counting of triangles for edge e is performed m times, which stands for the number of edges in a graph. Counting of triangles for each edge can be performed independent of other edges in parallel, without questioning the integrity of the algorithm.

4.5 Algorithm Analysis

In the following subsection we analyse the worst case time complexity of our proposed algorithm.

Lemma 11 *Given a graph $G = (V, E)$, its number of triangles is $O(|E|^{1.5})$.*

Proof For each node v, let us denote by $A(v)$ the set $\{u \in N(v), d_G(u) \geq d_G(v)\}$, where $d_G(v)$ is the degree of node v in G; this set contains only neighbours of v whose degree is no less than degree of v itself.

Without loss of generality, let's consider each triangle Δ_{uvw} such that $d_G(u) \leq d_G(v) \leq d_G(w)$; the set of triangles is $T = \{\Delta_{uvw} | \forall (u, v), (v, w), (u, w) \in E, d_G(u) \leq d_G(v) \leq d_G(w)\}$. One may then discover Δ_{uvw} by checking that w is in $A(u) \cap A(v)$. Thus, we can claim that the set of triangles is a subset of $T' = \{\Delta_{uvw} | \forall (u, v) \in E, w \in A(u) \cap A(v)\}$. Since $w \in A(u) \cap A(v)$ implies two edges (u, w) and (v, w).

To prove that the size of T' is $O(|E|^{1.5})$, we show that for every edge $(u, v) \in E$, the size of $A(u)$, $A(v)$ and accordingly their intersection are at most $2\sqrt{|E|}$. Suppose there exist $u \in V$ such that $|A(u)| \geq 2\sqrt{|E|} + 1$. Since the degree of all of them is at least equal to the degree of u, the sum of their degrees become $\sum_{v \in A(u)} d_G(v) \leq (2\sqrt{|E|} + 1) \times (2\sqrt{|E|} + 1) > 2|E|$, which is impossible.

So far, we reached a contradiction while assuming $|A(u)| > 2\sqrt{|E|}$ for any vertex $u \in V$, thus for every edge $(u, v) \in E$ we can find at most $2\sqrt{|E|}$ common neighbours in $A(u) \cap A(v)$ and we can conclude the following:

$$|T'| \in O(|E|^{1.5}) \Rightarrow |T| \in O(|E|^{1.5})$$

Lemma 12 *Given a graph $G = (V, E)$, its triangle graph $G' = (V', E')$ has $O(|E|^{1.5})$ number of edges.*

Proof For every triangle $\Delta_{uvw} \in G$ there are three edges in G'. Suppose T is the set of triangles in G, thus,

$$|E'| = 3|T| \Rightarrow |E'| \in O(|E|^{1.5})$$

Theorem 13 *Constructing the triangle graph $G' = (V', E')$ of a given graph $G = (V, E)$ takes $O(|E|^{1.5})$ time and space.*

Proof We prove this theorem by proposing an algorithm for constructing the triangle graph of a given graph G. The algorithm is very similar to the triangle listing algorithm proposed in [29]. The only difference between triangle graph construction and triangle listing of [29] is that here, for every triangle found by the algorithm, we add a set of edges to the triangle graph.

Algorithm 3 Algorithm for constructing triangle graph

Input: $G = (V, E)$
Output: $G' = (V', E')$ s.t., G' is the triangle graph of G
1: $G' = (V' = E, E' = \emptyset)$
2: number the vertices with an injective function $\rho()$
such that $d_G(u) > d_G(v)$ implies $\rho(u) < \rho(v)$ for all u and v
3: let A be an array of n arrays initially empty
4: **for** each vertex v taken in increasing order of $\rho()$ **do**
5: **for** each $u \in N_G(v)$ with $\rho(u) > \rho(v)$ **do**
6: **for** each w in $A[u] \cap A[v]$ **do**
7: add edges $E' \leftarrow E' \cup \{((u,v),(v,w)),((u,v),(u,w)),((v,w),(u,w))]\}$
8: add v to $A[u]$

The time complexity of this algorithm exactly follows from the time complexity of triangle listing in [29].

The worst case space complexity of this algorithm is equal to the amount of space need to store the nodes and edges in G' which is:

$$O(|V'| + |E'|) = O(|E| + |E|^{1.5}) = O(|E|^{1.5})$$

Theorem 14 *The overall time and space complexity of our proposed algorithm is $O(|E|^{1.5})$ using adjacency list graph representation.*

Proof The algorithm consists of three main stages: 1- $(k+1)$-core decomposition, 2- Triangle graph construction, 3- Node colouring loop. The first stage of the algorithm is simply done in $O(|E|)$. The second stage is the construction of the triangle graph and as the Theorem 13 implies, the time complexity of this stage is $O(|E|^{1.5})$. The third stage of the algorithm iterates over all the edges in the graph at most once (to update the degrees), thus the time complexity of this stage is $O(|V'| + |E'|)$. Therefore, the overall time complexity of the algorithm is:

$$O(|E| + |E|^{1.5} + |E|^{1.5}) = O(|E|^{1.5})$$

4.6 Bipartite Triangle Graph

Now, we introduce another efficient data structure for storing the triangles of a graph G, called the *bipartite triangle graph*.

Definition 4 *Given a graph $G = (V, E)$, the* bipartite triangle graph $G' = (V', E')$ *of G is a bipartite graph that has two vertex for every edge in G, one in each part of the graph. For example, let assume $(u, v) \in G$ then we have $(u, v) \in V'$ and $(u, v)' \in V'$. For every triangle $\Delta_{uvw} \in G$, we have six corresponding nodes in G' including (u, v), $(u, v)'$, (v, w), $(v, w)'$, (u, w) and $(u, w)'$. For every triangle $\Delta_{uvw} \in G$, we have also six corresponding edges in G', two edges from each node in G' to other related nodes in the other part of the graph. For example, for node $(u, v) \in V'$, we have two edges*

$((u,v),(v,w)') \in E'$ *and* $((u,v),(u,w)') \in E'$. *In summary, we can define the bipartite triangle graph as follows:*

$$V' = \{(v,u),(v,u)' : \forall (v,u) \in E\}$$

$$E' = \{((u,v),(v,w)'),((u,v),(u,w)'),((v,w),(u,v)'),((v,w),(u,w)'),$$
$$((u,w),(u,v)'),((u,w),(u,w)') : \forall \Delta_{uvw} \in G\}$$

The following example demonstrates how the bipartite triangle graph of a graph looks like.

Example 3 *Fig. 3 shows an example of a graph and its bipartite triangle graph.*

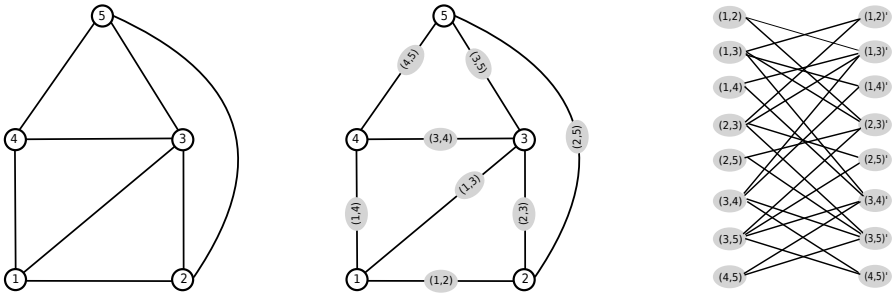


Fig. 3: An example of a graph and its bipartite triangle graph.

It is to be noted that the way we use the bipartite graph for the truss decomposition is very similar to the previous triangle graph. Moreover, similar properties can be proved similarly for the bipartite triangle graph and we avoid repeating the proofs. It is very clear that the size of the bipartite triangle graph is larger than the previous triangle graph and we will evaluate the performance of these two triangle graph in more details in experimental section.

5 Experimental Results

In this section, we first describe the experimental environment settings and then detail the steps taken to evaluate the efficiency of our approach. In this section we present the initial efficiency results of our algorithms in terms of the running time and memory usage. We compared the running time and memory usage of our approach with the in-memory algorithm proposed by Wang [7] using several real world datasets.

5.1 Experimental Environment

The main objective of the experiments is to investigate the efficiency of our algorithms for the truss decomposition. We compare the performance of our approach against the in-memory algorithm proposed in [7], and we call it *Wang*. We call our methods *BipartiteTriangle* and *SimpleTriangle*, where denote our basic approach using the bipartite triangle graph and the more efficient triangle graph, respectively.

All the experiments were conducted on an HP PC with 3.30GHz Intel Core i5-2500 processor with 16Gb RAM running an Ubuntu Desktop 18.04 LTS. We implemented our algorithms in C++. All the source codes are publicly available on GitHub[3]. Also we employed the published source code of the Wang approach for our comparative experiments.

In this section, we evaluate the performance of our approach for truss decomposition using eight real-world datasets. All of these datasets are from the Stanford Network Analysis Project (SNAP) [30]. The *Amazon* dataset was collected by crawling Amazon website which is based on *Customers Who Bought This Item Also Bought* feature of the Amazon website. *DBLP* presents a co-authorship network where two authors are connected if they publish at least one paper together. *LiveJ* was collected from LiveJournal, a free on-line blogging community where users declare friendship each other. In the *Youtube* social network, users form friendship each other and they can create groups which other users can join [31]. *Facebook* dataset was collected from survey participants using a Facebook app and it includes node features (profiles), circles, and ego networks [32]. The *EUCore* network was generated using email data from a large European research institution [33]. *Enron* was generated from the email communication within of around half million emails. The *Brightkite* and *Gowalla* networks were collected using two location-based social networking websites where users shared their locations by checking-in [34].

Table 1 represents some statistics of the datasets that includes the number of vertices, number of edges, disk usage size, maximum and median degrees, and the number of triangles for each network. As one can see in this table, most of the datasets provide a rather small median degree, which because of the heavily tail of power-low distribution in these networks [7].

5.2 Size of Triangle Graphs

In Section 4, we proposed two efficient algorithms, BipartiteTriangle and SimpleTriangle for truss decomposition. An important difference between these two methods is the size of their triangle graphs. Thus, in this section we aim to evaluate the size of these two triangle graphs. As mentioned, the truss decomposition procedure in both algorithms contains three main steps: 1) core

[3] We made all the source codes publicly available on GitHub under link: `https://github.com/mrezvani110/ktruss.`

Table 1: Statistics of datasets used in the experiments ($K = 10^3$, $M = 10^6$ and $G = 10^9$): the number of vertices and edges ($|V_G|$ and $|E_G|$), disk usage size in bytes, maximum and median degree (d_{max} and d_{med}), and the number of triangles (T_G).

Dataset	$\|V_G\|$	$\|E_G\|$	$size$	d_{max}	d_{med}	T_G
Amazon	335K	926K	12M	168	3	667K
DBLP	317K	1.0M	13M	306	2	2.2M
Youtube	1.1M	3.0M	37M	28576	2	3.1M
Facebook	4.0K	88.0K	835K	1043	11	1.6M
EUCore	1.0K	25.5K	214K	334	19	105K
Enron	36.7K	184K	4.0M	1383	3	727K
Brightkite	58.2K	214K	4.8M	1134	2	495K
Gowalla	197K	950K	12M	14730	9	2.3M

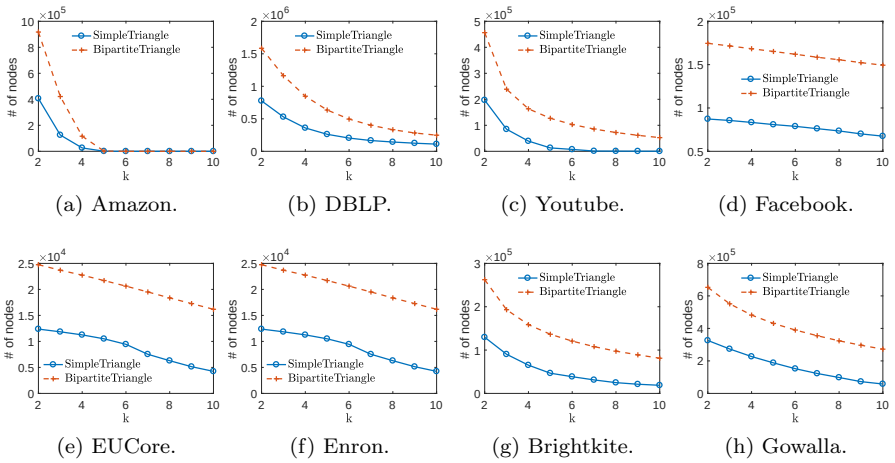
decomposition over the original graph, 2) constructing the triangle graph either bipartite or simple triangle graph, 3) core decomposition over the triangle graph generated in the previous step. It is clear that the first step is identical for both triangle graphs as it runs over the original input graph. For the second step, each algorithm generates its own triangle graph. As we explained in Section 4, the size of the simple triangle graph is exactly half of the bipartite graph for both the number of nodes and the number of edges in the graphs.

Fig. 4 and Fig. 5 show the number nodes and edges for the triangle graphs after the third step of the truss decomposition procedure, respectively. As one can see in the figures, the simple triangle graph consumes even less memory for maintaining the triangle graph after the third step of the procedure. In fact, the bipartite triangle graph needs around three orders of magnitude larger memory usage than the simple triangle graph. This is because the fact that the simple triangle graph is really more compact comparing to the bipartite triangle graph and the final core decomposition step in the algorithm can almost filter out ineffective nodes and edges from the graph for truss decomposition. It is to be noted that for some networks, such as Amazon and DBLP, the difference between the size of two triangle graphs reduces as the value of k increases. This is because the fact that the number of nodes and edges in these triangle graphs exponentially decrease by increasing the value of k in truss decomposition.

5.3 Running Time of Triangle Graphs

In this experiment, we evaluate the processing time of the main steps of the truss decomposition procedure for both bipartite and simple triangle graphs. Since the first step is identical for these two graphs, we only evaluate the elapsed time of the second and third steps in the procedure.

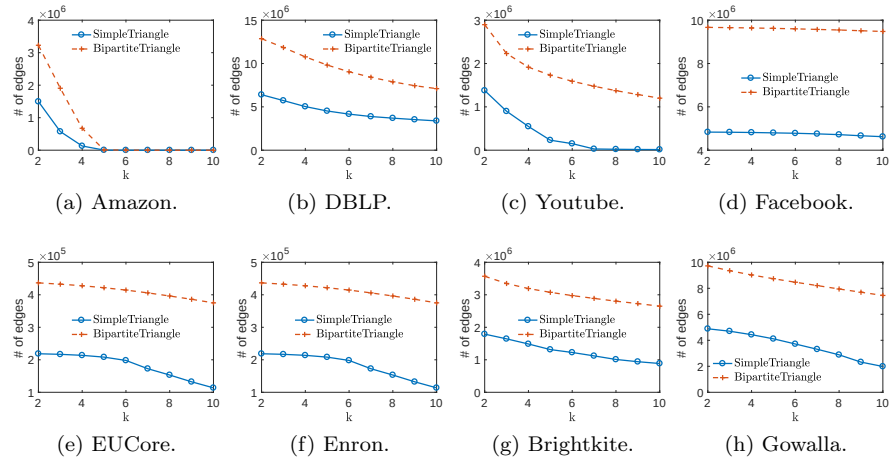
Fig. 6 illustrates the running time of the second step of truss decomposition (triangle graph generation) for both triangle graph and various networks. The results show that the generating the simple triangle graph is apparently much



(a) Amazon. (b) DBLP. (c) Youtube. (d) Facebook.

(e) EUCore. (f) Enron. (g) Brightkite. (h) Gowalla.

Fig. 4: The number nodes for the triangle graphs after the third step of the truss decomposition procedure.

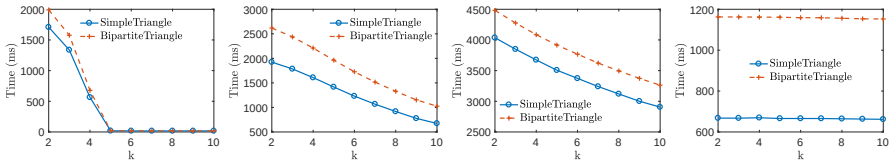


(a) Amazon. (b) DBLP. (c) Youtube. (d) Facebook.

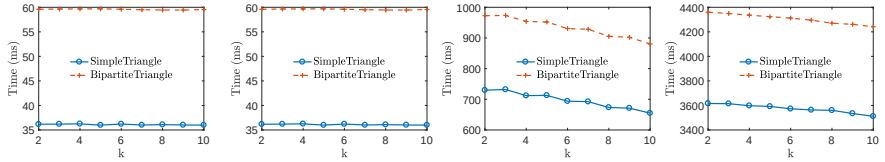
(e) EUCore. (f) Enron. (g) Brightkite. (h) Gowalla.

Fig. 5: The number edges for the triangle graphs after the third step of the truss decomposition procedure.

faster than the bipartite triangle graph. This can be explained by the fact that the simple triangle graphs are smaller in both the number of nodes and edge, as reported in the results of the previous experiments. More precisely, since the simple triangle graph is smaller than the bipartite one for both the number of nodes and edges, the graph generation algorithm is run much faster on the simple triangle graph.

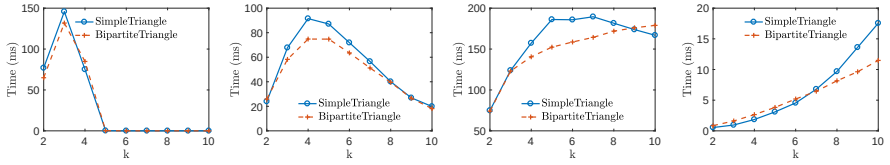


(a) Amazon. (b) DBLP. (c) Youtube. (d) Facebook.

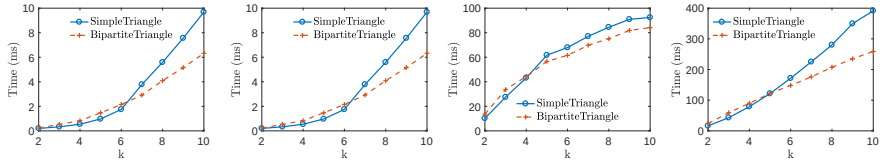


(e) EUCore. (f) Enron. (g) Brightkite. (h) Gowalla.

Fig. 6: The running time of the second step of the truss decomposition procedure.



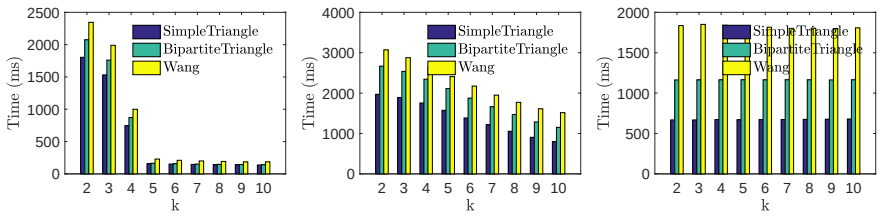
(a) Amazon. (b) DBLP. (c) Youtube. (d) Facebook.



(e) EUCore. (f) Enron. (g) Brightkite. (h) Gowalla.

Fig. 7: The running time of the third step of the truss decomposition procedure.

Fig. 7 illustrates the running time of the third step of truss decomposition (core decomposition over the triangle graph) for both triangle graph and various networks. The results show that the running time for almost all of the networks increases as the value of k increases. Moreover, both triangle graph provides approximately similar performance for the third step of the truss decomposition, even that for some values of k, the bipartite graph runs slightly faster than the simple triangle graph.



(a) Amazon. (b) DBLP. (c) Facebook.

Fig. 8: The total running time of the truss decomposition algorithms.

5.4 Performance of Truss Decomposition

In this section, we compare the performance of our both truss decomposition algorithms against the Wang method proposed in [7]. To this end, we measure both the total processing time and the peak memory usage of the algorithms over the networks presented in the datasets.

Fig. 8 shows the total running time of the truss decomposition for both triangle graphs and the Wang approach. It is to be noted that we exclude the reading time of the input graph for all three methods as it is not in fact a part of the truss decomposition procedure. Unfortunately, we were not able to obtain the result on all datasets for the Wang approach because it did not finish after waiting a long time. In fact, we only able to obtain the results for Wang on three datasets, Amazon, DBLP, and Facebook, as shown in Fig. 8. As one can see in the results, both proposed triangle graphs take less time for running the truss decomposition on the various values of k. It is to be noted that the dramatic decrease of the running time in Amazon for k larger than 4 is because the fact that there is no k-truss in this graph for such values of k, and the reported running times for these values are only for the first step of the procedure which is the first core decomposition. The results thus verify that our proposed triangle graphs are more feasible for truss decomposition in massive networks.

Table 2 shows the peak memory usage of the truss decomposition for both triangle graphs and the Wang approach. As one can see in the results, both of our triangle graphs consume more memory for truss decomposition procedure comparing to the Wang method. This is because the fact that we have to make one extra graph, the triangle graph to speed up the procedure. In other words, we sacrifice the memory usage to achieve a better processing time. Moreover, as we expected the Bipartite triangle graph consume more memory than the Simple triangle graph. This can be explained by the results obtained in our experiment for the size of the triangle graphs in the previous section.

Table 2: The peak memory usage (bytes) of the truss decomposition algorithms ($M = 10^6$ and $G = 10^9$).

	Amazon	DBPL	Facebook
		$k = 2$	
Wang	307M	751M	467M
Bipartite	751M	2.08G	1.38G
Simple	448M	1.12G	700M
		$k = 3$	
Wang	271M	716M	467M
Bipartite	649M	1.97G	1.38G
Simple	396M	1.06G	700M
		$k = 4$	
Wang	194M	677M	467M
Bipartite	381M	1.83G	1.38G
Simple	262M	994M	700M

6 Conclusions

In this paper, we proposed two novel and efficient algorithms for the truss decomposition problem. To this end, we introduced two triangle graphs for representing the triangles of a graph, and we used these triangle graphs to efficiently detect the k-trusses in the graph. The experimental results over some real-world datasets showed that our method significantly improves the efficiency of the truss decomposition by reducing the processing time compared to the state of the art. We plan to extend our algorithms to propose a distributed approach for truss decomposition in massive graphs. We also plan to investigate the effectiveness of our triangle graphs for the truss community search problem in a dynamic graph setting with frequent insertions and deletions of graph vertices and edges.

7 Compliance with Ethical Standards:

– **Funding:** This research has not been funded by any academic or industrial grant.
– **Conflict of Interest:** Author A, *Mohsen Rezvani*, declares that he has no conflict of interest. Author B, *Mojtaba Rezvani*, declares that he has no conflict of interest.
– **Ethical approval:** This article does not contain any studies with human participants or animals performed by any of the authors.

Acknowledgements The authors would like to acknowledge the financial support of the Shahrood University of Technology for this research under project No: 14016.

References

1. Fragkiskos D Malliaros, Christos Giatsidis, Apostolos N Papadopoulos, and Michalis Vazirgiannis. The core decomposition of networks: Theory, algorithms and applications. *The VLDB Journal*, 29(1):61–92, 2020.
2. Rong-Hua Li, Lu Qin, Jeffrey Xu Yu, and Rui Mao. Influential community search in large networks. *Proceedings of the VLDB Endowment*, 8(5):509–520, 2015.
3. Yixiang Fang, Xin Huang, Lu Qin, Ying Zhang, Wenjie Zhang, Reynold Cheng, and Xuemin Lin. A survey of community search over big graphs. *The VLDB Journal*, 29(1):353–392, 2020.
4. Yi-Xiu Kong, Gui-Yuan Shi, Rui-Jie Wu, and Yi-Cheng Zhang. k-core: Theories and applications. *Physics Reports*, 2019.
5. Jonathan Cohen. Trusses: Cohesive subgraphs for social network analysis. *National Security Agency Technical Report*, 2008.
6. Shaden Smith, Xing Liu, Nesreen K Ahmed, Ancy Sarah Tom, Fabrizio Petrini, and George Karypis. Truss decomposition on shared-memory parallel systems. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2017.
7. Jia Wang and James Cheng. Truss decomposition in massive networks. In *Proc. of VLDB*, pages 812–823, 2012.
8. Sitao Huang, Mohamed El-Hadedy, Cong Hao, Qin Li, Vikram S Mailthody, Ketan Date, Jinjun Xiong, Deming Chen, Rakesh Nagi, and Wen-mei Hwu. Triangle counting and truss decomposition using fpga. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018.
9. Roger Pearce and Geoffrey Sanders. K-truss decomposition for scale-free graphs at scale in distributed memory. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2018.
10. James Cheng, Yiping Ke, Shumo Chu, and M Tamer Ozsu. Efficient core decomposition in massive networks. In *Proc. of ICDE*, pages 51–62. IEEE, 2011.
11. R.A. Hanneman and M. Riddle. *Introduction to Social Network Methods*. University of California, 2005.
12. Shaikh Arifuzzaman, Maleq Khan, and Madhav Marathe. Patric: a parallel algorithm for counting triangles in massive networks. In *Proc. of CIKM*, pages 529–538. ACM, 2013.
13. Shumo Chu and James Cheng. Triangle listing in massive networks and its applications. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 672–680, 2011.
14. Xiaocheng Hu, Yufei Tao, and Chin-Wan Chung. Massive graph triangulation. In *Proc. of SIGMOD*, pages 325–336. ACM, 2013.
15. Ha-Myung Park and Chin-Wan Chung. An efficient mapreduce algorithm for counting triangles in a very large graph. In *Proc. of CIKM*, pages 539–548. ACM, 2013.
16. Kanat Tangwongsan, A Pavan, and Srikanta Tirthapura. Parallel triangle counting in massive streaming graphs. In *Proc. of CIKM*, pages 781–786. ACM, 2013.
17. Sayan Ghosh and Mahantesh Halappanavar. Tric: Distributed-memory triangle counting by exploiting the graph structure. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2020.
18. Santosh Pandey, Xiaoye Sherry Li, Aydin Buluc, Jiejun Xu, and Hang Liu. H-INDEX: Hash-indexing for parallel triangle counting on gpus. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2019.
19. Feng Zhao and Anthony K. H. Tung. Large scale cohesive subgraphs discovery for social network visual analysis. In *Proc. of VLDB*, pages 85–96, 2012.
20. Jonathan Cohen. Graph twiddling in a mapreduce world. *Computing in Science & Engineering*, 11(4):29–41, 2009.

21. Louise Quick, Paul Wilkinson, and David Hardcastle. Using pregel-like large scale graph processing frameworks for social network analysis. In *Proc. of ASONAM*, pages 457–463, 2012.
22. Yingxia Shao, Lei Chen, and Bin Cui. Efficient cohesive subgraphs detection in parallel network analysis. In *Proc. of SIGMOD*, 2014.
23. Wafaa M. A. Habib, Hoda M. O. Mokhtar, and Mohamed E. El-Sharkawi. Weight-based k-truss community search via edge attachment. *IEEE Access*, 8:148841–148852, 2020.
24. Yuli Jiang, Xin Huang, and Hong Cheng. I/O efficient k-truss community search in massive graphs. *The VLDB Journal*, pages 1–26, 2021.
25. Alessio Conte, Daniele De Sensi, Roberto Grossi, Andrea Marino, and Luca Versari. Discovering k-trusses in large-scale networks. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–6, 2018.
26. Mohammad Almasri, Omer Anjum, Carl Pearson, Zaid Qureshi, Vikram S. Mailthody, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. Update on k-truss decomposition on gpu. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2019.
27. MIT/Amazon/IEEE. GraphChallenge. `https://graphchallenge.mit.edu/`. [Online; accessed 11-August-2020].
28. Alessio Conte, Daniele De Sensi, Roberto Grossi, Andrea Marino, and Luca Versari. Truly scalable k-truss and max-truss algorithms for community detection in graphs. *IEEE Access*, 8:139096–139109, 2020.
29. Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *Proc. of WEA*, pages 606–609, 2005.
30. Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.
31. Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.
32. Jure Leskovec and Julian J Mcauley. Learning to discover social circles in ego networks. In *Advances in neural information processing systems*, pages 539–547, 2012.
33. Hao Yin, Austin R Benson, Jure Leskovec, and David F Gleich. Local higher-order graph clustering. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 555–564, 2017.
34. E Cho, SA Myers, and J Leskovec. Friendship and mobility: Friendship and mobility: User movement in location-based social networksfriendship and mobility. In *User Movement in Location-Based Social Networks ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2011.