Exploring Truss Maintenance in Fully Dynamic Graphs: A Mixed Structure-Based Approach

Qi Luo, *Student Member, IEEE,* Dongxiao Yu, *Senior Member, IEEE,* Xiuzhen Cheng, *Fellow, IEEE,* Hao Sheng, *Senior Member, IEEE* and Weifeng Lyu, *Member, IEEE*

Abstract—Graphs are widely employed in complex system modeling, VLSI design, and social analysis. Mining cohesive subgraphs is a fundamental problem in graph analysis, while implementing cohesive subgraphs requires analysts to not only ensure cohesiveness but also consider the computational intractability. Among a variety of diverse cohesive structures, k-truss exhibits a perfect trade-off between structure tightness and computational efficiency. In a k-truss, each edge is present in at least $k-2$ triangles. This study aims to contribute to this growing area of truss maintenance in fully dynamic graphs by avoiding expensive re-computation. Specifically, we consider the challenging scenario of batch processing of edge and vertex insertion/deletion and propose efficient algorithms that can maintain the trusses by only searching a very small range of affected edges. Also, our algorithms allow parallel implementations to further improve the efficiency of maintenance. Extensive experiments on both real-world static and temporal graphs illustrate the efficiency and scalability of our algorithms.

Index Terms—Graph analysis; truss maintenance; mixed structure.

1 Introduction

Graphs have been widely adopted to model complex systems, VLSI design, and social networks [1] that are embodied in many current applications [2]. While utilizing massive graphs, it is critical to identify key substructures, known as cohesive subgraphs, to derive useful network information like communities, influential spreaders, autonomous system of Internet and many more. Particularly, cohesive subgraphs can be used to analyze the complex structures of large-scale software systems, since the interior information of software structures can be modeled as hierarchical subgraphs [3]. In Very Large Scale Integration (VLSI) design, graphs are used extensively in the modeling of chipsets and an important task is to find tight areas to minimize connections between circuit elements [4]. The principles of graph theory are utilized in the developments of automated control systems in order to provide fully autonomous control of plant operations in the events of plant failures and changes in scheduling tasks. [5]. To this end, cohesive subgraph computation has received great attentions from both research communities and industry [6], [7].

However, finding the densest subgraph, i.e., a clique, is a NP-hard problem [8]. Hence, most research in recent years has focused on the quantification of sparser cohesive subgraphs such as quasi-clique [9], k-core [10], k-plex [11] and k-truss [12]. Nevertheless, all the aforementioned models except k-core and k-truss suffer from the computational intractability problem. However, k-core, where each vertex has at least k neighbors, may result in incohesive subgraphs

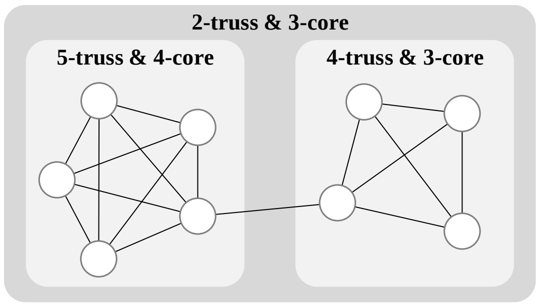


Fig. 1. An example of k-truss and k-core.

[13]. For example, the graph in Fig. 1 is a 3-core one, where each vertex is connected to at least 3 neighbors, but it is not a cohesive graph. Instead, the whole graph is composed of two cohesive subgraphs. On the other hand, k-truss (each edge is present in at least $k-2$ triangles) is a perfect trade-off between structure tightness and computational efficiency, since 1) a k-truss can be computed in $O(m^{1.5})$ time in massive graphs, where m is the number of edges in the graph [14]; 2) a k-truss must be a $(k-1)$-core, but the converse may not be true [12]. k-truss plays an important role in many applications such as community search [15], epidemic spreading [16], social network analysis [17], personalized recommendation [18], influence maximization [19], and Internet routing [20].

Trussness is a metric associated with k-truss and is defined on each edge $e \in G$, where G is a graph; it gives us the maximum k such that e is in a k-truss. After the trussness of each edge $e \in G$ is calculated, all the k-trusses can be obtained. Computing the trussness of each edge in a graph is referred to as *truss decomposition* and is solvable in $O(m^{1.5})$ by adopting an edge peeling algorithm [12] (where m is the number of edges). In recent years, many solutions were

- *Q. Luo, D. Yu (Corresponding Author), and X. Cheng are with the School of Computer Science and Technology, Shandong University, Qingdao, P.R. China. Email: luoqi2018@mail.sdu.edu.cn, {dxyu, xzcheng}@sdu.edu.cn.*
- *H. Sheng and W. Lyu are with School of Computer Science and Engineering, Beihang University, Beijing, P.R. China. Email: {shenghao,lwf}@buaa.edu.cn.*

proposed to improve the efficiency of truss decomposition algorithms [14], [21].

However, all the above mentioned solutions are restricted to static graphs and may not be applicable to real-world graphs that are dynamic, i.e., the graphs are continuously evolving with edge and vertex insertions/deletions. For example, in many social networks (edges are user interactions), users are constantly joining or leaving the network. Similarly, in collaboration networks (edges are author collaborations), new collaborations occur among authors as new papers are published. Although the existing results can compute the trussness of edges in almost linear time, they are inapplicable to real-world scenarios, as real graphs continue to grow endlessly and may contain billions of edges and vertices. This significantly increases the computation requirements due to the need for trussness recomputation for every graph change.

The problem of *truss maintenance* was proposed to address the above-mentioned drawbacks of trussness recomputation. Truss maintenance is a challenging problem, as the insertion/deletion of a single edge may cause all edges' trussness to change. There are two main problems to consider for truss maintenance: 1) determining which edge's trussness will change, and 2) the amount of trussness change after vertex/edge insertions/deletions. Hence, most existing truss maintenance solutions focus on a simple case of single edge [22], [23] or vertex [24] insertion/deletion, as the trussness change of every edge can be easily determined in this special scenario. For the more complex but more practical cases of multiple edge/vertex insertions/deletions, the existing batch processing approaches can only handle multiple edge insertions and fail to accommodate fully dynamic graphs. Though vertex insertion/deletion can be processed using an algorithm adapted for edge insertion/deletion, it is inefficient when the inserted/deleted vertex is connected to many edges. This is because all the affected edges can only be processed one by one using existing algorithms.

In this paper, we study the problem of truss maintenance under the insertion/deletion of a batch of vertices and edges. We propose a structure called *Mixed Structure* which is composed of a set of edges and vertices (as shown in Fig. 2(c)). We prove that the insertion/deletion of a mixed structure only makes each edge change its trussness by at most one. When a mixed structure is inserted/deleted, it only needs to identify the set of edges which can change the trussness, to enable batch processing of inserted/deleted edges and vertices. This allows us to handle the newly inserted/deleted edges in the same manner as existing edges. Furthermore, with the aid of the proposed index called *triangle support*, sufficient conditions can be obtained to determine whether the trussness of an edge will change after the graph changes. Finally, our truss maintenance algorithms admit parallel implementations where the same thread can handle all edges that have the same trussness, to further improve the efficiency of our algorithms. Our contributions are summarized as follows:

1) We propose truss maintenance algorithms in fully dynamic graphs for both insertions and deletions. Our algorithms can process multiple edge and vertex insertions/deletions simultaneously. Compared with the approach in [25], which only processes multiple edge insertions/deletions without considering vertex insertions/deletions, our algorithms can significantly reduce the iterations needed for processing both inserted/deleted edges and vertices, especially for the cases where inserted/deleted vertices are connected to numerous edges.
2) Based on the proposed mixed structure, whose insertion/deletion can be simplified to find the affected edges, our algorithms admit efficient parallel implementations to further reduce the number of iterations needed for truss maintenance.
3) Extensive experiments conducted on the graphs show that our proposed algorithms achieve good efficiency and scalability. Our algorithms are dozens of times faster than the state-of-the-art algorithm, and are dozens to hundreds of times faster than the single vertex and single edge algorithms. The parallel implementations of our algorithms can further reduce the maintenance time with multiple threads.

Roadmap. The rest of this paper is organized as follows. In Section 2 we provide the problem definition. The theoretical basis for the algorithm design is described in Section 3. The truss maintenance algorithms and their parallel implementations are presented in Section 4. The experimental results are illustrated and analyzed in Section 5, and the related research in truss decomposition and truss maintenance is briefly reviewed in Section 6. Finally, in Section 7 we conclude the paper.

2 PROBLEM DEFINITIONS

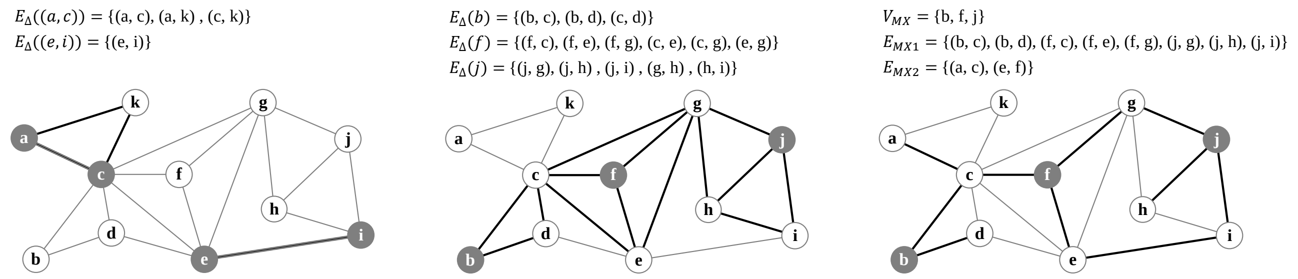
We consider a simple undirected and unweighted graph $G = (V, E)$, where V and E are the vertex set and edge set, respectively. Let $N(v) = \{u \in V : (v, u) \in E\}$ denote the neighbor set of v, and $d(v) = |N(v)|$ is v's degree in G. For a set of vertices $S \subset V$, its induced subgraph is denoted by $G[S]$, where the vertex set is S and the edge set is $\{(u, v)|(u, v) \in E \,\&\, u, v \in S\}$.

A triangle is a cycle of length 3 in a graph. The triangle formed by $u, v, w \in V$ is denoted by $\triangle_{uvw}$. A triangle that contains vertex u or edge e is denoted by $\triangle_{[u]}$ or $\triangle_{[e]}$, respectively. The *support* of an edge $e(u, v)$ in G is defined as the number of triangles containing e, denoted as $sup_G(e) = |\{\triangle_{uvw} : \triangle_{uvw} \in G \wedge \forall w \in V\}|$. One can see that $sup_G(e) = |N_G(u) \cap N_G(v)|$. The minimum support of all edges in a subgraph $H \subseteq G$ is denoted by $sup_G(H)$. With the above definitions, we can define k-truss as follows:

Definition 1 (k-Truss). A connected subgraph H of G is a k-truss if H is maximal and satisfies the constraint that $sup_H(e) \geq k - 2$ for $\forall e \in H$.

From the above definition, one can see that each edge in H is included in at least $k - 2$ triangles in H. The trussness of a subgraph H is defined as $\tau(H) = \min\{sup_H(e) + 2 : e \in E(H)\}$. The trussness of an edge e is the maximum k such that a k-truss contains e, i.e., $\tau(e) = \max\{\tau(H) : e \in H, H \subset G\}$, which can be computed as follows:.

$$\begin{aligned} &\tau(e) = \max\{k, 2\} \text{ s.t.} \\ &|\{w : \min\{\tau((u, w)), \tau((v, w))\} \geq k\}| \geq k - 2, \qquad (1) \\ &w \in N_G(u) \cap N_G(v). \end{aligned}$$


$E_\Delta((a,c)) = \{(a, c), (a, k), (c, k)\}$
$E_\Delta((e,i)) = \{(e, i)\}$
$E_\Delta(b) = \{(b, c), (b, d), (c, d)\}$
$E_\Delta(f) = \{(f, c), (f, e), (f, g), (c, e), (c, g), (e, g)\}$
$E_\Delta(j) = \{(j, g), (j, h), (j, i), (g, h), (h, i)\}$
$V_{MX} = \{b, f, j\}$
$E_{MX1} = \{(b, c), (b, d), (f, c), (f, e), (f, g), (j, g), (j, h), (j, i)\}$
$E_{MX2} = \{(a, c), (e, f)\}$
a
b
c
d
e
f
g
h
i
j
k

(a) Edge-based triangle edge set (b) Vertex-based triangle edge set (c) Mixed structure

Fig. 2. Examples of triangle edge sets and mixed structures. (a) shows the edge-based triangle edge sets of (a, c) and (e, i), which do not overlap, thus we have $E_{TI} = \{(a, c), (e, i)\}$; (b) shows the vertex-based triangle edge sets of vertex b, f, and j, and we have $V_{TI} = \{b, f, j\}$; (c) shows a mixed structure where V_{MX} is a triangle independent vertex set, E_{MX1} and E_{MX2} are two edge sets satisfying the constraints mentioned in the definition.

The notations of this paper are summarized in Table 1. We drop subscripts whenever possible if it is clear from context.

TABLE 1
Notations and Descriptions

Notations	Descriptions
$G = (V, E)$	graph G with vertex set V and edge set E
H_k^e	the unique k-truss that contains e
$G[S]$	the induced subgraph of $G = (V, E)$ by $S \subseteq V$
$e(u, v)$	an edge with u and v as endpoints
$d_G(v)$	the degree of v in G
$N_G(v)$	the neighbors of v in G
$E_G(v)$	the incident edges of vertex v in G
$\triangle_{wuv}$	a triangle consists of three vertices w, u, v
$\triangle_{[u]}$	a triangle contains a vertex u
$\triangle_{[e]}$	a triangle contains an edge e
$sup_G(e)$	the support of e in G
$\tau_G(e)$	the trussness of e in G
$E_\triangle(e)$	the edge-based triangle edge set of edge e
$E_\triangle(v)$	the vertex-based triangle edge set of vertex v
E_{TI}	a triangle-independent edge set
V_{TI}	a triangle-independent vertex set
H_{MS}	a mixed structure
E_{MS}	all edges in a mixed structure

Problem statement. We consider the problem of *truss maintenance*, which intends to update the trussness of the edges in a fully dynamic graph while avoiding recomputation. More specifically, we focus on the scenarios where vertices and edges are inserted into or deleted from a graph, which are called *incremental* and *decremental* truss maintenance, respectively.

3 Theoretical Basis

In this section, we propose a structure called *Mixed Structure*, which is composed of inserted/deleted edges and vertices. It can be shown that the insertion/deletion of a mixed structure changes each edge's trussness by at most one. By this way, when a mixed structure is inserted/deleted, we only need to identify the set of edges that can change the trussness, enabling batch processing of inserted/deleted edges and vertices.

3.1 Mixed Structure

We first introduce edge-based and vertex-based triangle edge sets. Given a graph $G = (V, E)$, the edge-based triangle edge set of an edge $e \in E$ is the set of all edges of the triangles that contain e, denoted as $E_\triangle(e) = \{e' | e' \in \triangle_{[e]}\}$. $E_{TI} \subset E$ is a *triangle-independent edge set* if it holds that $E_\triangle(e_i) \cap E_\triangle(e_j) = \emptyset$ for any two edges $e_i, e_j \in E_{TI}$. Similarly, the vertex-based triangle edge set of a vertex $v \in V$ is the set of all edges of the triangles that contain v, denoted as $E_\triangle(v) = \{e | e \in \triangle_{[v]}\}$. V_{TI} is a *triangle-independent vertex set* if for any two vertices $v_i, v_j \in V_{TI}$ it holds that $E_\triangle(v_i) \cap E_\triangle(v_j) = \emptyset$. Examples of edge-based and vertex-based triangle edge sets are illustrated in Fig. 2. In the following, we give the definition of *mixed structure*.

Definition 2 (Mixed Structure). Given a graph $G = (V, E)$, a mixed structure $H_{MS} = \{V_{MS}, E_{MS}\}$ of G satisfies the following requirements.

1) V_{MS} is a triangle-independent vertex set of G.
2) $E_{MS} = E_{MS1} \cap E_{MS2}$, where E_{MS1} contains all edges incident to the vertices in V_{MS}, and E_{MS2} is a triangle-independent edge set. Furthermore, for each pair of edges $e_1 \in E_{MS1}$ and $e_2 \in E_{MS2}$, $E_\triangle(e_1) \cap E_\triangle(e_2) = \emptyset$.

In the following, when we refer to inserting/deleting a mixed structure H_{MS}, it is equivalent to inserting/deleting all the edges E_{MS} in the H_{MS}.

3.2 Quantifying Trussness Change

Next, we quantify the trussness change of the edges after inserting/deleting a mixed structure. We first show that the trussness of the edges that are not in the mixed structure can change by at most one. As for the newly inserted edges, the bound of the trussness change is much more complicated, as the trussness change can be much more than one. Here, we propose a concept of *pre-truss* defined on newly inserted edges. With this concept, it can be shown that the trussness of every edge is either its *pre-truss* or *pre-truss+1*. By this way, we can subsequently handle all edges in the same way while identifying the trussness change of the edges.

Lemma 1 (Mixed Structure Deletion). Given a graph G and a mixed structure H_{MS} of G, let $G' = G \setminus E_{MS}$ be the graph obtained after deleting E_{MS} from G. Then the trussness of each edge in G' is decreased by at most one.

Proof: For an edge $e^* \in G'$, assume there is a subgraph $H \in G$ containing e which is a k-truss. By Definition 1, $sup_H(e) \geq k-2$ for each edge e in H. Let $H' = H \setminus E_{MS}$. Without loss of generality, we assume that H' is still connected. Otherwise, we can consider the connected components of H' one by one. For each edge $e \in H'$, it is easy to see that e can lose at most one adjacent triangle due to the deletion of E_{MS} by Definition 2. This means that $sup_{H'}(e) \geq k-3$. In other words, H' is $k-1$-truss. Hence, the trussness of e^* is at least $k-1$, which completes the proof. □

The above lemma implies that the deletion of a mixed structure can make the trussness of each edge decrease by at most one. We next consider the trussness change of the edges when a mixed structure is inserted.

Lemma 2 (Mixed Structure Insertion). Given a graph G and a mixed structure H_{MS} of G, let $G' = G \cup H_{MS}$ be the graph obtained after inserting H_{MS} into G. Then the trussness of each edge in G is increased by at most one.

Proof: We prove this lemma by contradiction. Assume that the trussness of an edge $e^* \in G$ is changed by more than one. Formally, assume that $\tau_G(e) = k$ and $\tau_{G'}(e) = k+x$ where $x > 1$. Let H' be the $(k+x)$-truss containing e in G'. Next we consider the subgraph $H = H' \setminus H_{MS}$. By Lemma 1, H is at least a $(k+x-1)$-truss. Notice that $H \subseteq G$, which means that the trussness of e is at least $k+x-1 > k$. This contradiction completes the proof. □

We next bound the trussness change of newly inserted edges. We first introduce a concept called *pre-truss*, denoted as $\bar{\tau}(e)$, for newly inserted edges in a mixed structure.

Definition 3 ((k, d)-Neighborhood [24]). Given a graph G, the (k, d)-neighborhood of a vertex v, denoted by $G_v^{k,d}$, is the maximal subgraph $H(V_H, E_H) \subseteq G[N(v)]$ such that

1) $\tau_G(e) \geq k, \forall e \in E_H$;
2) $deg_H(u) \geq d, \forall u \in V_H$.

Let $G' = G \cup H_{MS}$, where H_{MS} is a mixed structure of G' with E_{MS1} and E_{MS2} the two corresponding edge sets. The pre-truss of the edges in H is defined as follows.

Definition 4 (Pre-Truss for Edges in E_{MS1}). The pre-truss of an edge $e(v, w) \in E_{MS1}$ is defined as

$$\bar{\tau}_{G'}(e_{MS1}) = \max\{k, 2\} \text{ s.t. } \{k : w \in G_v^{k,k-2}\}. \tag{2}$$

Definition 5 (Pre-Truss for Edges in E_{MS2}). The pre-truss of an edge $e(u, v) \in E_{MS2}$ is defined as

$$\bar{\tau}_{G'}(e_{MS2}) = \max\{2, k\} \text{ s.t. } |\{w : \min\{\tau_G((u, w)), \tau_G((v, w))\} \geq k\}| \geq k-2. \tag{3}$$

Facilitated with pre-truss, one can get the following result, which indicates that the trussness of the newly inserted edges can be at most *pre-truss+1*.

Lemma 3. Given a graph G and a mixed structure H_{MS} of G‘, with $G' = G \cup H_{MS}$. For any edge $e \in E_{MS}$, it holds that

$$\bar{\tau}_{G'}(e) \leq \tau_{G'}(e) \leq \bar{\tau}_{G'}(e) + 1 \tag{4}$$

Proof: We first prove $\bar{\tau}_{G'}(e) \leq \tau_{G'}(e)$, then show that $\tau_{G'}(e) \leq \bar{\tau}_{G'}(e) + 1$.

1) We prove $\bar{\tau}_{G'}(e) \leq \tau_{G'}(e)$ by considering the following two cases.
Case 1. $e \in E_{MS1}$. Let $e = (v, w)$ and $\hat{k} = \bar{\tau}_{G'}(e)$, where v, w are the endpoints of e. Because $\hat{k} = \bar{\tau}_{G'}(e) = \max_{k\geq 2}\{k : w \in G_v^{k,k-2}\}$, there exists a $\hat{k}$-truss $H_k = (V_H, E_H)$ in G. This means $\forall e' \in H_k$, $sup_{H_k}(e') \geq k-2$. Let $H^* = (V_H \cup \{v\}, E_H \cup \{(w', v)|w' \in G_v^{k,k-2}\})$, which adds vertex v and v's incident edges (w', v) to H. By Definition 3,for each edge $(w', v) \in H^* \setminus H_k$, w' and v have at least $k-2$ common neighbors in H^*, indicating that $sup_{H^*}((v, w')) \geq k-2$, i.e. H^* is at least a k-truss. Hence, $\tau_{G'}(e) \geq \bar{\tau}_{G'}(e)$.
Case 2. $e \in E_{MS2}$. By Definition 5, there are $k-2$ triangles containing e such that $sup_{G'}(e) \geq k-2$. All edges in these triangles, except e, have trussness not smaller than k. In other words, there exists a k-truss containing these edges except e and we denote this k-truss as H_k. Then we obtain that for $e \in H_k \cup \{e\}$, $sup_{G'}(e) \geq k-2$. Hence, $\tau_{G'}(e) \geq k = \bar{\tau}_{G'}(e)$.
2) We next prove $\tau_{G'}(e) \leq \bar{\tau}_{G'}(e) + 1$. We mainly consider the case of $e \in E_{MS1}$, as the case of $e \in E_{MS2}$ can be similarly proved.
Let $e = (v, w)$ and $\hat{k} = \bar{\tau}_{G'}(e) = \max\{k : w \in G_v^{k,k-2}\}$. Assume $\tau_{G'}(e) = \hat{k}+x$, where $x \geq 2$. Then there exists a $(\hat{k}+x)$-truss $H_{\hat{k}+x}$ containing e in G'. We delete v and all its incident edges $E(v)$ from $H_{\hat{k}+x}$, and the remaining graph is denoted as H^*. Then the trussness of each edge in H^* is decreased by at most 1 as per Lemma 1. Hence, for any edge $e* \in H^*$, it satisfies $\tau(e^*) \geq \hat{k}+x-1 \geq \hat{k}+1$, i.e., $\max\{k : w \in G_v^{k,k-2}\} \geq \hat{k}+1$ before v is deleted, which contradicts with the given condition that $\hat{k} = \{k : w \in G_v^{k,k-2}\}$.

□

The above result implies that after inserting a mixed structure, the value of *pre-truss* can be treated as the initial trussness of the newly inserted edges, and the actual trussness of these edges can be at most *pre-truss+1*.

The problem of identifying the edges that change the trussness after inserting/deleting a mixed structure still goes unanswered. In the subsequent subsection, we give the necessary conditions for the edges to change trussness, such that the edges with trussness changes can be identified quickly by traversing as few edges as possible.

3.3 Scoping Edge Traversal

We first introduce a few definitions, then give the necessary conditions for edges with trussness changes.

Definition 6 (k-Triangle). Given a triangle $\triangle_{uvw} \subset G$, if all edges in $\triangle_{uvw}$ have trussness of at least k, i.e., $\min\{\tau((u, v)), \tau((v, w)), \tau((u, w))\} = k$, $\triangle_{uvw}$ is called a k-triangle, denoted as $\triangle_{uvw}^k$.

Let $\triangle_{[e]}^k$ denote a k-triangle containing edge e. For an edge $e(u, v)$, a triangle is called a *support triangle* of e if $\triangle_{uvw}$ satisfies $\min\{\tau((v, w)), \tau((u, w))\} \geq \tau(e)$. Then the *k-triangle support* of edge $e(u,v)$, denoted as $S(e)$, is defined as the number of support triangles of e, i.e.,

$$S(e) = |\{\triangle^{k}_{[e]} : \tau(e) = k, \triangle^{k}_{[e]} \in G\}| \tag{5}$$

With the definition of k-triangle support, one can derive the sufficient conditions to determine whether the trussness of an edge changes after the graph changes.

Lemma 4. Let G' be the graph obtained after inserting a mix structure H_{MS} into graph G, then the trussness of an edge $e \in G'$ would not increase if $S(e) \leq \tau_G(e) - 2$.

Proof: Let $\hat{k} = \tau_G(e)$ and $S(e) = \hat{k} - x$ where $x \geq 3$. We assume that the trussness of e increases after inserting the mix structure. By Lemma 2, we can assume $\tau_{G'}(e) = \hat{k} + 1$. By Equation (1), there are at least $\hat{k} - 1$ support triangles of e in G. This contradicts with the fact that there are $\hat{k} - x$ support triangles of e in G, which completes the proof. □

Lemma 5. Let G' be the graph obtained after deleting a mix structure H_{MS} from graph G, then the trussness of an edge $e \in G'$ would decrease by one if $S(e) < \tau_G(e) - 2$.

Proof: Let $\hat{k} = \tau_G(e)$ and $S(e) = \hat{k} - x$ where $x \geq 3$. Assume $\tau_{G'}(e) = \hat{k}$. This means that e is in at least $\hat{k} - 2$ triangles in which each edge has trussness at least $\hat{k}$. However, e is in at most $\hat{k} - 3$ triangles in which each edge has trussness greater than $\hat{k}$ by $S(e) = \hat{k} - x$. This contradiction completes the proof. □

We next present necessary conditions for determining potential edges whose trussness may change. The necessary conditions can significantly reduce the traversal range. At first, we define *k-triangle connectivity* as follows.

Definition 7 (k-Triangle Connectivity). Given two k-triangles $\triangle^{(s)}$ and $\triangle^{(t)}$ in G, they are k-triangle connected, denoted as $\triangle^{(s)} \overset{\triangle}{\leftrightarrow} \triangle^{(t)}$, if there exists a sequence of $n \geq 2$ k-triangles $\triangle^{(1)}, \cdots, \triangle^{(n)}$ s.t. $\triangle^{(s)} = \triangle^{(1)}$, $\triangle^{(t)} = \triangle^{(n)}$, and for $1 \leq i \leq n$, $\triangle^{(i)} \cap \triangle^{(i+1)} = \{e | e \in E_G\}$ and $\tau(e) = k$. Analogously, we say two edges e,$e' \in E$ are k-triangle connected, denoted as $e \overset{k}{\leftrightarrow} e'$, if and only if (1) e and e' belong to the same k-triangle, or (2) $e \in \triangle^{(s)}$, $e' \in \triangle^{(t)}$, s.t. $\triangle^{(s)} \overset{\triangle}{\leftrightarrow} \triangle^{(t)}$.

Lemma 6 (Insertion Propagation). Let G' be the graph obtained after inserting a mixed structure $H_{MS} = (V_{MS}, E_{MS})$ into the graph $G = (V, E)$. An edge e may increase its trussness only if it satisfies one of the following conditions:

1) e is in a triangle with an edge $\hat{e} \in E_{MS}$ and $\tau_G(e) \leq \bar{\tau}(\hat{e})$;
2) e is k-triangle connected with an edge e' satisfying Condition 1), where $\tau_G(e) = k$.

Proof: We first prove that the edges that do not satisfy the conditions would not increase their trussness after inserting a mixed structure. We assume that $e^* \in G$ does not satisfy the given conditions. Then, one of the following two cases must occur:

Case 1. e^* is in a triangle with an edge $\hat{e} \in E_{MS}$, but $\tau(e^*) > \bar{\tau}(\hat{e})$;

Case 2. e^* is not in a triangle with any edge in E_{MS}, and is not k-triangle connected with any edge $\hat{e}$ that satisfies condition 1.

For *Case 1*, let $\hat{k} = \bar{\tau}(\hat{e})$ and $\tau_G(e^*) = \hat{k} + x$, where $x \geq 1$. By Lemma 2, the trussness of every edge can increase by at most one. We assume that $\tau_{G'}(e^*) = \hat{k} + x + 1$. Then there would be at least $\hat{k} + x - 1$ triangles in which the trussness of every edge is not smaller than $\hat{k} + x + 1$. However, the trussness of $\hat{e}$ can be at most $\hat{k}+1$ by Lemma 3. The contradiction implies that the trussness of e^* cannot increase.

For *Case 2*, suppose that $\hat{k} = \tau_G(e^*)$. We assume the trussness of e^* increases by one, i.e., $\tau_{G'}(e^*) = \hat{k} + 1$. If an edge has its trussness increased, it must hold that either there is a new triangle (increased support) containing it, or in one of the triangles $\triangle^{\hat{k}}_{[e^*]}$, there is an edge e_1 other than e^* whose trussness changes from $\hat{k}$ to $\hat{k} + 1$. Because e^* is not adjacent to newly inserted edges, it must be the latter case. Similarly, if e_1 is also not adjacent to a newly inserted edge, it must have an adjacent edge e_2 whose trussness must change from $\hat{k}$ to $\hat{k} + 1$, and e_1 and e_2 are in the same triangle. Repeat the above procedure, we can find an edge e' such that it is adjacent to a newly inserted edge and its trussness is changed from $\hat{k}$ to $\hat{k} + 1$. This means that e^* and e' are $\hat{k}$-triangle connected, where $\hat{k} = \tau_G(e')$, and e' satisfies Condition 1) by Case 1. This contradicts with our assumption, which completes the proof. □

With a similar analysis, we can get the necessary conditions for the deletion case.

Lemma 7 (Deletion Propagation). Let G' be the graph obtained after deleting a mixed structure $H_{MS} = (V_{MS}, E_{MS})$ from graph $G = (V, E)$. An edge e may decrease the trussness only if e satisfies one of the following conditions:

1) e is in a triangle with an edge $\hat{e} \in E_{MS}$ and $\tau_G(e) \leq \tau(\hat{e})$;
2) e is k-triangle connected with an edge e' satisfying 1), where $\tau_G(e') = k$.

4 Truss Maintenance Algorithms

In this section, we propose truss maintenance algorithms to update the trussness of edges when a batch of edges and vertices are inserted/deleted.

The algorithms employ the theoretical results of the previous section to realize batch processing for insertion/deletion of edges. As discussed before, when a mixed structure is inserted/deleted, the trussness changes by at most one (for those inserted to newly inserted vertices, the change is with respect to pre-truss). In this case, the trussness maintenance problem is simplified into identifying the potential edges whose trussness may change. Furthermore, the necessary conditions identified earlier can greatly reduce the traversal range of the potential edges. With these results, our algorithms first divide the inserted/deleted vertices and edges into multiple mixed structures and then handle one mixed structure at each iteration.

4.1 Incremental Truss Maintenance

The incremental truss maintenance algorithm is presented in Algorithm 1, where we consider inserting a set of vertices Δ_V and a set of edges ΔE into graph G.

The algorithm is executed in iterations. At each iteration, a *mixed structure* $\{V_{MS}, E_{MS1} \cup E_{MS2}\}$ is computed from the remaining uninserted vertices and edges by Definition 2 (Line 1-17). After that, the computed mixed structure is inserted into the graph (Line 18), and then for each inserted edge, the edges whose trussness would change are identified (Line 19-27). During this process, the pre-truss of the inserted edges is first computed (Line 21). Based on the pre-truss values, all candidate edges that form triangles with edges in the mixed structure and whose trussness values are smaller than pre-truss are added into a map called M (Line 23-27). Then the IncrementalTraversal (Algorithm 2) is executed (Line 28). In Algorithm 2, for each value k in M and all edges that are k-triangle connected with edges in $M[k]$, the triangle supports of these edges are computed to determine whether they can increase the trussness by Lemma 4 (Line 2-12). Once an edge is determined not to change the trussness, it would be deleted, and the supports of other potential edges are then updated. Finally, all the edges still in M increase their trussness by one (Line 21-22).

Algorithm 1 Incremental Truss Maintenance

```
Input: $G = (V, E)$, $\{\tau(e) | e \in E\}$, $H = \{\Delta V, \Delta E\}$
Output: $\{\tau(e) | e \in E \cup \Delta E\}$
1:  while $\Delta E \neq \emptyset$ do
2:      $E_\triangle(MS) \leftarrow \emptyset$ ;                    ▷ triangle edges
3:      $V_{MS} \leftarrow \emptyset$ ;                  ▷ vertices in mixed structure
4:      while $\Delta V \neq \emptyset$ do
5:          for $v \in \Delta V$ do
6:              if $E_\triangle(MS) \cap E_\triangle(v) = \emptyset$ then
7:                  $V_{MS}.add(v)$ ;
8:                  $E_\triangle(MS).add(E_\triangle(v))$ ;
9:      $E_{MS1} \leftarrow \bigcup_{v \in V_{MS}} E_G(v)$ ;
10:     $\Delta E \leftarrow \Delta E \setminus E_{MS1}$ ;                    ▷ update $\Delta E$
11:     $E_{MS2} \leftarrow \emptyset$ ;
12:     for $e \in \Delta E$ do
13:         if $E_\triangle(e) \cap E_\triangle(MS) = \emptyset$ then
14:             $E_{MS2}.add(e)$ ;
15:             $E_\triangle(MS).add(E_\triangle(e))$ ;
16:     $\Delta V \leftarrow \Delta V \setminus V_{MS}$ ;                    ▷ update $\Delta V$
17:     $\Delta E \leftarrow \Delta E \setminus E_{MS2}$ ;                   ▷ update $\Delta E$
18:     $G \leftarrow (V \cup V_{MS}, E \cup E_{MS1} \cup E_{MS2})$;
19:     $M \leftarrow \emptyset$ ;          ▷ a map stores same trussness of edges
20:     for $e_0 = (u, v) \in E_{MS1} \cup E_{MS2}$ do
21:         compute pre-$\tau(e_0)$;
22:         $\tau(e_0) \leftarrow$ pre-$\tau(e_0)$ ;
23:         for $w \in N(u) \cap N(v)$ do
24:             $k = \min\{\tau((v, w)), \tau((u, w))\}$;
25:             if $k \leq \tau(e_0)$ then
26:                 if $\tau(e) = k$, $e \in \{(u, w), (v, w)\}$ then
27:                     $M[k].add(e)$ ;
28:     IncrementalTraversal($G, \tau, M$) ;
```

Example. In Fig. 2(c), assume the mixed structure is inserted into the graph. Before insertion, (a, k), (c, k), (h, i), (h, g) are edges whose trussness are equal to 2, (c, d), (c, e), (c, g), (d, e), (e, g) are edges whose trussness are equal to 3. When the mixed structure is inserted, the pre-truss of (e, i) is 2, and the pre-truss of other edges in the mixed structure are 3. After the trussness update by our incremental algorithms,

Algorithm 2 IncrementalTraversal(G, τ, M)

```
1:  for $k = \max(M.keys())$ decrease to $\min(M.keys())$ do
2:      $Q \leftarrow \emptyset$; $Q.push(M[k])$ ;
3:      while $Q \neq \emptyset$ do
4:          $(x, y) \leftarrow Q.pop()$ ;
5:          $S((x, y)) \leftarrow 0$ ;          ▷ number of support triangles
6:          for $z \in N(x) \cup N(y)$ do
7:              if $\min\{\tau((x, z)), \tau((y, z))\} < k$ then continue;
8:              $S((x, y)) \leftarrow S((x, y)) + 1$ ;
9:              if $\tau((z, x)) = k$ and $(z, x) \notin M[k]$ then
10:                 $Q.push((z, x))$ ; $M[k].add((z, x))$ ;
11:             if $\tau((z, y)) = k$ and $(z, y) \notin M[k]$ then
12:                 $Q.push((z, y))$ ; $M[k].add((z, y))$ ;
13:     while $\exists S((x, y)) \leq k - 2$ in $M[k]$ do
14:         $M[k].remove((x, y))$ ;
15:         for $z \in N(x) \cap N(y)$ do
16:             if $\min\{\tau((x, z)), \tau((y, z))\} < k$ then continue ;
17:             if $\tau(e \in \{(z, x), (z, y)\}) = k$ and $e \notin M[k]$ then
18:                 continue ;
19:             if $(e \in \{(z, x), (z, y)\}) \in M[k]$ then
20:                 $S(e) \leftarrow S(e) - 1$ ;
21:     for $e \in M[k]$ do
22:         $\tau(e) \leftarrow k + 1$ ;                  ▷ update trussness
```

the trussness of (c, e), (c, f), (c, g), (e, f), (e, g) and (f, g) increase to 4, the trussness of other edges in the graph are 3.

Analysis. We next analyze the efficiency of our incremental truss maintenance algorithm. We consider Algorithm 1 to insert a vertex set and an edge set $H = \{\Delta V, \Delta E\}$ into graph $G = (V, E)$. Denote by $G_T = G(V \cup T_V, E \cup T_E)$ for any $T_V \subset \Delta V$, $T_E \subset \Delta E$, then we have $G_H = (V \cup \Delta V, E \cup \Delta E)$.

Given a graph G_T, where $T \subseteq H$. Let d^T_{max} be the maximum degree of vertices in G_T. For a vertex $v \in T$, let $V^v_\triangle(T)$ be the set of vertices such that for $\forall v' \in V^v_\triangle(T)$, $E_\triangle(v') \cap E_\triangle(v) \neq \emptyset$. Similarly, we define $E^e_\triangle(T)$ as the set of edges such that for $\forall e' \in E^e_\triangle(T)$, $E_\triangle(e') \cap E_\triangle(e) \neq \emptyset$. Define

$$\mathcal{R}_+ = \max_{T \subseteq H} \max_{v \in \Delta V, e \in \Delta E} \{|V^v_\triangle(T)| + |E^e_\triangle(T)|\}. \tag{6}$$

Let $T \subseteq H$ and $T' \subseteq H \setminus T$. After inserting T into $G_{T'}$, an initial edge is the one that forms a triangle with a newly inserted edge and whose trussness is not larger than the pre-truss of the newly inserted edge. Let $|A^T_{T'}|$ be the initial edges and the edges that are k-triangle connected with the initial edges. Define

$$\mathcal{A}_{max} = \max_{T \subseteq H, T' \subseteq H \setminus T} |A^T_{T'}|. \tag{7}$$

For an edge $e \in G_{T'}$, let $P^T_{T'}(e) = S(e) - \tau_{G_{T'}}(e) + 2$ where $S(e)$ is the k-triangle support of e in $G_{T'}$. Define

$$\mathcal{P}_{max} = \max_{T \subseteq H, T' \subseteq H \setminus T, e \in G_{T'}} P^T_{T'}(e). \tag{8}$$

Based on the above definitions, the following theorem can be obtained.

Theorem 1. After inserting a set of vertices ΔV and edges ΔE into G, Algorithm 1 can correctly update the trussness of edges in $O(\mathcal{R}_+ \cdot (d^2_{max} \cdot (|\Delta V| + |\Delta E|) + \mathcal{A}_{max} \cdot (d^2_{max} + \mathcal{P}_{max})))$ time.

Proof: We first prove the correctness of our algorithm. At each iteration of Algorithm 1, all selected edges and vertices satisfy the definition of mixed structure. After inserting a mixed structure, only the edges satisfying the conditions given in Lemma 6 are processed, and only these edges may change their trussness. When an edge e's k-triangle support is not greater than $\tau(e)-2$, the trussness of e does not increase by Lemma 4. Hence, our algorithm can correctly determine whether an edge can increase its trussness. Finally, each remaining edge has enough support triangles to increase its trussness, which ensures our algorithm's correctness.

We next analyze the running time of the algorithm. Because each mixed structure contains at least one vertex in $V^v_\triangle$ or one edge in $E^e_\triangle$, there are at most $\mathcal{R}_+$ iterations. Each iteration takes $O(d^2_{max}\cdot(|\Delta V|+|\Delta E|))$ time to construct a mixed structure, and two loops are executed in Algorithm 2. The first loop takes $O(\mathcal{A}_{max}\cdot d^2_{max})$ time to count the support triangles for edges whose trussness may change, and the second loop takes $O(\mathcal{A}_{max}\cdot\mathcal{P}_{max})$ time to find the edges with the number of support triangles not larger than trussness-2. Combining all together, we get the time complexity of Algorithm 1. □

4.2 Decremental Truss Maintenance

The pseudocode of updating edge trussness after deleting a subgraph from G is outlined in Algorithm 3. The decremental truss maintenance algorithm is similar to Algorithm 1 but without computing the pre-truss. The difference is that when an edge with trussness k has smaller than $k-2$ k-triangles, the trussness of the edge would decrease by one.

Algorithm 3 Decremental Truss Maintenance

```
Input: $G=(V,E)$, $\{\tau(e)|e\in E\}$, $H=\{\Delta V,\Delta E\}$
Output: $\{\tau(e)|e\in E\setminus\Delta E\}$
1: while $\Delta E\neq\emptyset$ do
2:     $E_\triangle(MS)\leftarrow\emptyset$ ;
3:     $V_{MS}\leftarrow\emptyset$ ;
4:     while $\Delta V\neq\emptyset$ do
5:         for $v\in\Delta V$ do
6:             if $E_\triangle(v)\cap E_\triangle(MS)\neq\emptyset$ then
7:                 $V_{MS}.add(v)$ ;
8:                 $E_\triangle(MS).add(E_\triangle(v))$;
9:     $E_{MS1}\leftarrow\bigcup_{v\in V_{MS}}E_G(v)$ ;
10:    $\Delta E\leftarrow\Delta E\setminus E_{MS1}$ ;                ▷ update $\Delta E$
11:    $E_{MS2}\leftarrow\emptyset$ ;
12:    for $e\in\Delta E$ do
13:        if $E_\triangle(e)\cap E_\triangle(MS)=\emptyset$ then
14:            $E_{MS2}.add(e)$ ;
15:            $E_\triangle(MS).add(E_\triangle(e))$ ;
16:    $\Delta V\leftarrow\Delta V\setminus V_{MS}$ ;                ▷ update $\Delta V$
17:    $\Delta E\leftarrow\Delta E\setminus E_{MS2}$ ;               ▷ update $\Delta E$
18:    $G\leftarrow(V\setminus V_{MS},E\setminus(E_{MS1}\cup E_{MS2}))$;
19:    $M\leftarrow\emptyset$ ;        ▷ a map stores same trussness of edges
20:    for $e_0=(u,v)\in E_{MS1}\cup E_{MS2}$ do
21:        for $w\in N(u)\cap N(v)$ do
22:            $k=\min\{\tau((v,w)),\tau((u,w))\}$;
23:            if $k\leq\tau(e_0)$ then
24:                if $\tau(e)=k$, $e\in\{(u,w),(v,w)\}$ then
25:                    $M[k].add(e)$ ;
26:    DecrementalTraversal$(G,\tau,M)$ ;
```

Algorithm 4 DecrementalTraversal(G,τ,M)

```
1: for $k=\min(M.keys())$ decrease to $\max(M.keys())$ do
2:     $Q\leftarrow\emptyset$; $Q.push(M[k])$ ;
3:     while $Q\neq\emptyset$ do
4:         $(x,y)\leftarrow Q.pop()$ ;
5:         $S((x,y))\leftarrow 0$ ;          ▷ number of support triangles
6:         for $z\in N(x)\cup N(y)$ do
7:             if $\min\{\tau((x,z)),\tau((y,z))\}\geq k$ then
8:                 $S((x,y))\leftarrow S((x,y))+1$ ;
9:             if $\tau((z,x))=k$ and $(z,x)\notin M[k]$ then
10:                $Q.push((z,x))$ ; $M[k].add((z,x))$ ;
11:            if $\tau((z,y))=k$ and $(z,y)\notin M[k]$ then
12:                $Q.push((z,y))$ ; $M[k].add((z,y))$ ;
13:    while $\exists S((x,y))<k-2$ in $M[k]$ do
14:        $\tau(x,y)\leftarrow k-1$ ;                    ▷ update trussness
15:        for $z\in N(x)\cap N(y)$ do
16:            if $\min\{\tau((x,z)),\tau((y,z))\}<k$ then continue ;
17:            if $\tau(e\in\{(z,x),(z,y)\})=k$ and $e\notin M[k]$ then
18:                continue ;
19:            if $(e\in\{(z,x),(z,y)\})\in M[k]$ then
20:                $S(e)\leftarrow S(e)-1$ ;
```

Example. In Fig. 2(c), assume the mixed structure is deleted from the graph. Before deletion, the trussness of (c,e), (c,f), (c,g), (e,f), (e,g) and (f,g) are 4, the trussness of other edges in the graph are 3. Update the trussness of the edges in the graph after deleting the mixed structure, the trussness of (c,e), (c,g) and (e,g) decrease to 3, the trussness of (a,k), (c,k), (h,i) and (h,g) decrease to 2.

Analysis. We next analyze the efficiency of the decremental truss maintenance algorithms. We consider Algorithm 3 that deletes a vertex set and an edge set $H=\{\Delta V,\Delta E\}$ from graph $G=(V,E)$. Denoted by $G_T=G(V\setminus T_V,E\setminus T_E)$ for any $T_V\subset\Delta V$, $T_E\subset\Delta E$, then we have $G_H=(V\setminus\Delta V,E\setminus\Delta E)$. Given a graph G_T, where $T\subseteq H$, let d_{max} be the maximum degree of a vertex in G_T. For a vertex $v\in T$, let $V^v_\triangle(T)$ be the set of vertices such that for $\forall v'\in V^v_\triangle(T)$, $E_\triangle(v')\cap E_\triangle(v)\neq\emptyset$. Similarly, we define $E^e_\triangle(T)$ as the set of edges such that for $\forall e'\in E^e_\triangle(T)$, $E_\triangle(e')\cap E_\triangle(e)\neq\emptyset$.

$$\mathcal{R}_-=\max_{T\subseteq H}\max_{v\in\Delta V,e\in\Delta E}\{|V^v_\triangle(T)|+|E^e_\triangle(T)|\}. \tag{9}$$

Let $T\subseteq H$ and $T'\subseteq H\setminus T$. An initial edge is the one that forms a triangle with a deleted edge and whose trussness is not larger than the trussness of the deleted edge. Let $|B^T_{T'}|$ be the initial edges and the edges that are k-triangle connected with the initial edges after deleting T from $G_{T'}$. Define

$$\mathcal{B}_{max}=\max_{T\subseteq H,T'\subseteq H\setminus T}|B^T_{T'}|. \tag{10}$$

For an edge $e\in G_{T'}$, let $Q^T_{T'}(e)=S(e)-\tau_{G_{T'}}(e)+2$ where $S(e)$ is the k-triangle support of e in $G_{T'}$. Let

$$\mathcal{Q}_{max}=\max_{T\subseteq H,T'\subseteq H\setminus T,e\in G_{T'}}Q^T_{T'}(e). \tag{11}$$

Based on the above definitions, the following theorem can be obtained. The analysis is similar to that of Theorem 1, and therefore it is omitted.

Theorem 2. After deleting a set of vertices ΔV and edges ΔE from G, Algorithm 3 can correctly update the trussness of the edges in $O(\mathcal{R}_-\cdot(d^2_{max}\cdot(|\Delta V|+|\Delta E|)+\mathcal{B}_{max}\cdot(d^2_{max}+\mathcal{Q}_{max})))$ time.

4.3 Parallel Implementations

Our truss maintenance algorithms allow for parallel implementations to further improve the performance.

It has been proved in Section 3 that the trussness of every edge can change up to 1 after inserting/deleting a mixed structure. For a specified k, the traversed edges are k-triangle connected, i.e., the traversed edges for different values of k are disjoint.

Thus, we can assign the task of searching edges that may update the trussness to different processes for different k values to take advantage of multicore processors [26]. Ideally, we would use a thread to execute the traversal process for each batch of edges whose pre-truss equal to a particular k. However, it is impossible to execute so many threads in reality, thus we employ the thread pool technique with a fixed number of threads to accomplish our parallel algorithm task. The thread pool allows threads to be reused, thus each thread can continue to perform other tasks without being destroyed after executing one task. As our algorithms spend most of the time traversing edges, this parallelized task partition can significantly improve our algorithms' efficiency.

Algorithm 5 (parallel incremental traversal) and Algorithm 6 (parallel decremental traversal) are the parallel implementations of Algorithm 2 (incremental traversal) and Algorithm 4 (decremental traversal), respectively. The time complexity of the parallel algorithms is the same as sequential algorithms.

Algorithm 5 ParallelIncrementalTraversal(G, τ, M)

```
1:  for each $k$ in $M.keys()$ in parallel do
2:      $Q.push(M[k])$;                                  ▷ a stack stores edges
3:      $V \leftarrow \emptyset$;                        ▷ visited edges
4:      $TS \leftarrow \emptyset$;                       ▷ triangle support
5:      while $Q \neq \emptyset$ do
6:          $(x, y) \leftarrow Q.pop()$;
7:          if $V.contains((x, y))$ then continue;
8:          $V.add((x, y))$;
9:          for each $z \in N(x) \cap N(y)$ do
10:             $t \leftarrow \min\{\tau((x, z)), \tau((y, z))\}$;
11:             if $(t > k)$ or $(\tau(e') = k$ and $TS(e') > k - 2$ for $e' \in \{(x, z), (y, z)\})$ then
12:                 if $TS((x, y)) = null$ then
13:                     $TS((x, y)) = 1$;
14:                 else
15:                     $TS((x, y)) \leftarrow S((x, y)) + 1$ ;
16:                 if $\tau((x, z)) = k$ and $!V.contains((x, z))$ then
17:                     $Q.push((x, z))$;
18:                 if $\tau((y, z)) = k$ and $!V.contains((y, z))$ then
19:                     $Q.push((y, z))$;
20:     while $\exists TS((x, y)) \leq k - 2 \in E_k$ do
21:         $E_k.remove((x, y))$;
22:         for $z \in N(x) \cup N(y)$ do
23:             if $\min\{\tau((x, z)), \tau((y, z))\} < k$ then continue ;
24:             if $\tau(e \in \{(z, x), (z, y)\}) = k$ and $e \notin E_k$ then
25:                 continue ;
26:             if $(x, z) \in E_k$ then
27:                 $TS((x, z)) \leftarrow TS((x, z)) - 1$ ;
28:             if $(y, z) \in E_k$ then
29:                 $TS((y, z)) \leftarrow TS((y, z)) - 1$ ;
30:     for $e \in E_k$ do
31:         $\tau(e) \leftarrow k + 1$ ;                  ▷ update trussness
```

Algorithm 6 ParallelDecrementalTraversal($G, \{\tau(e) | e \in E \cup E_{MS}\}, M$)

```
1:  for each $k$ in $M.keys()$ in parallel do
2:      $Q.push(M[k])$;                                  ▷ a stack stores edges
3:      $V \leftarrow \emptyset$;                        ▷ visited edges
4:      $S \leftarrow \emptyset$;                        ▷ triangle support of edges
5:      while $Q \neq \emptyset$ do
6:          $(x, y) \leftarrow Q.pop()$;
7:          if $V.contains((x, y))$ then continue;
8:          $V.add((x, y))$;
9:          for each $z \in N(x) \cap N(y)$ do
10:             if $\min\{\tau((x, z)), \tau((y, z))\} \geq k$ then
11:                 if $S((x, y)) = null$ then
12:                     $S((x, y)) = 1$;
13:                 else
14:                     $S((x, y)) \leftarrow S((x, y)) + 1$ ;
15:                 if $\tau((x, z)) = k$ and $!V.contains((x, z))$ then
16:                     $Q.push((x, z))$;
17:                 if $\tau((y, z)) = k$ and $!V.contains((y, z))$ then
18:                     $Q.push((y, z))$;
19:     $E_k \leftarrow S.keys()$;                       ▷ candidate edges
20:     while $\exists S((x, y)) < k - 2$ in $E_k$ do
21:         $E_k.remove((x, y))$ ;
22:         $\tau(e) \leftarrow k - 1$ ;                 ▷ update trussness
23:         for $z \in N(x) \cup N(y)$ do
24:             if $\min\{\tau((x, z)), \tau((y, z))\} < k$ then continue ;
25:             if $\tau(e \in \{(z, x), (z, y)\}) = k$ and $e \notin E_k$ then
26:                 continue ;
27:             if $(x, z) \in E_k$ then
28:                 $S((x, z)) \leftarrow S((x, z)) - 1$ ;
29:             if $(y, z) \in E_k$ then
30:                 $S((y, z)) \leftarrow S((y, z)) - 1$ ;
```

5 Experiments

We conduct extensive experiments to evaluate the performances of our algorithms. The evaluations are carried out on 6 real-world graphs and 3 temporal graphs. We first report the performance of our algorithms under different proportions of vertex and edge updates. Then, we test the parallelism of our parallel implementations under various number of threads. To derive the factors that affect our algorithms, we analyze their performance by inserting/deleting vertices with different degrees. Finally, we compare our algorithms with those given in [22], [24], [25], to evaluate the performance improvement of our algorithms over the previous ones.

All programs are implemented in Java and compiled with JDK 8. The evaluations are performed on a machine with Intel Xeon CPU E5-2620 @2.1GHz with 120 GB memory. The reported results are the average over 10 runs.

Datasets. The datasets are publicly available in the Stanford Network Analysis Project (SNAP)[1]. *EmailEnron* is an email communication network covering all the email communications within a dataset of around half a million emails. *Gowalla* is a location-based social networking website where users share their locations by checking-in. *EmailEuAll* is a network generated using the email data from a large European research institution. *Amazon* is a network that is collected by crawling the Amazon website. *YouTube* is a video-sharing website that includes a social network.

1. http://snap.stanford.edu/data

WikiTalk is a Wikipedia talk communication network, which is a free encyclopedia written collaboratively by volunteers around the world. The statistics of these graphs are reported in Table 2, where $|\triangle|$ denotes the number of triangles, and $Truss_{max}$ denotes the maximum trussness of the edges in a graph.

The details of the temporal graphs are provided in Table 3. In these temporal graphs, each edge has a timestamp. The *WikiTalk* temporal network represents edits made by Wikipedia registered users in each other's talk page. *SuperUser* and *Overflow* are temporal networks of interactions on the Super User and Stack overflow websites, respectively. All graphs are treated as undirected in our experiments.

TABLE 2
Statistic of Real-world Graphs

Dataset	|V|	|E|	$\|\triangle\|$	$Truss_{max}$
EmailEnron	36K	183K	727K	423
Gowalla	196K	950K	2273K	1300
EmailEuAll	265K	420K	267K	723
Amazon	334K	952K	667K	164
YouTube	1135K	2988K	3056K	4037
WikiTalk	2394K	5021K	9203K	1634

TABLE 3
Statistic of Temporal Graphs

Dataset	|V|	Static Edges	Temporal Edges
SuperUser	197K	1.44M	924.9K
WikiTalk	1.14K	7.8M	3.3M
Overflow	2.6M	63.5M	36.2M

Performance Evaluation. For each graph, we randomly choose $p_i = i\%$ vertices and $p_j = j\%$ edges of the original graph to construct various mixed structures for our experimental study. More specifically, we employ all the edges incident to the p_i vertices and the p_j edges as well as the p_i vertices to form a set from which we use a simple greedy approach to select one maximal mixed structure for each iteration – the mixed structure is removed from the set before selecting the one for the next iteration. We consider $i, j = 1, 2, 3, 4, 5$ but here only report the results for $i = j$ as the trends are the same. Fig. 3 shows the processing time for edge and vertex insertion and deletion. It can be seen that when more vertices and edges are updated, the total time increases sub-linearly. This is because when more updates happen, more vertices and edges can be selected into the mixed structure processed for each iteration. Therefore, our algorithms are more suitable for processing a large number of updates.

We next show the performance of our algorithms in temporal networks. For each graph, we choose five time points $T1$, $T2$, $T3$, $T4$, $T5$, where $T1 < T2 < T3 < T4 < T5$, and edges inserted/deleted before Ti, $(1 \leq i \leq 5)$ are regarded as the updated edges. We show the average processing time per edge in Fig. 4 for edge insertions and deletions. It can be seen that in both cases, as the number of inserted/deleted edges increases, the processing time per edge decreases.

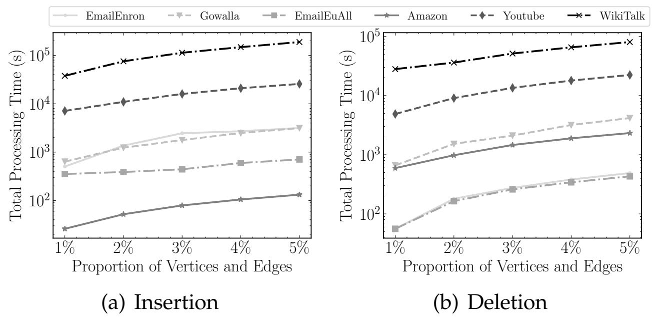


Fig. 3. The processing time of our truss maintenance algorithms at different scales to update vertices and edges.

Hence, our algorithms are suitable for real-world scenarios.

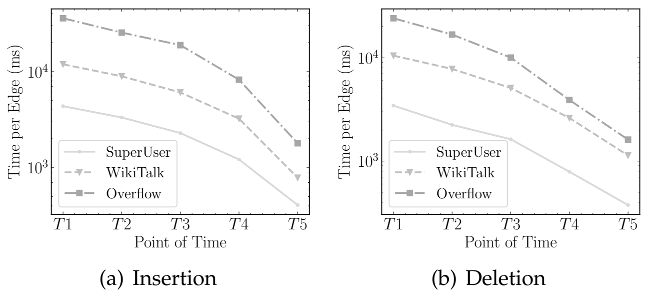


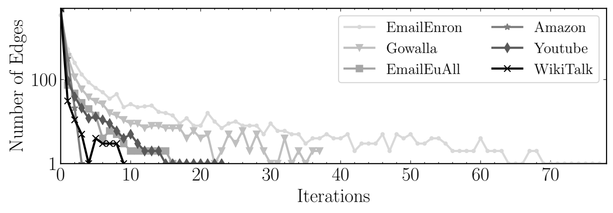
Fig. 4. Performance in temporal graphs.

To explore the relevant factors affecting our algorithms, we randomly choose $5K$ vertices and $5K$ edges as the updated vertices and edges. We then sort the original graph vertices by their degree and divide them into two categories (low degree and high degree). Fig. 5 (a) shows the number of edges in the mixed structure of each iteration while Fig. 5 (b) and Fig. 5 (c) illustrate the cases where the updated vertices are selected from the low degree and high degree categories, respectively. The processing time for different categories is illustrated in Fig. 6.

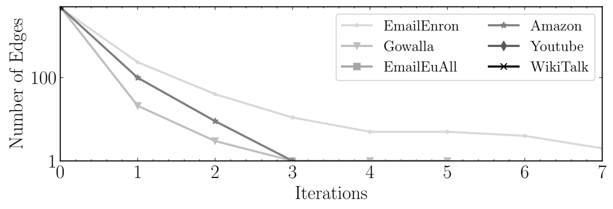
It can be seen that when the vertices have a smaller degree, more edges can be handled simultaneously using our algorithms during the first few iterations. For the Amazon and WikiTalk datasets, all the edges can be handled quickly (within one iteration). For vertices from the high degree category, it takes more iterations to handle the updates. From Fig. 6, it is evident that for all datasets, the processing time increases as the degree of the inserted vertices gets larger.

To evaluate our algorithms' parallelism, we vary the number of parallel threads in our parallel algorithms. For each graph, the number of updated vertices is $5k$, so is that of the edges, and the size of the thread pool ranges from 2 to 32. We report the results in Fig. 7. It can be seen that as the size of the thread pool grows, the total processing time decreases nearly linearly, but the speedup slows down and almost becomes constant. This is because, even though the processing time is small, the cost to maintain the synchronization among threads is high, which greatly affects the performance.

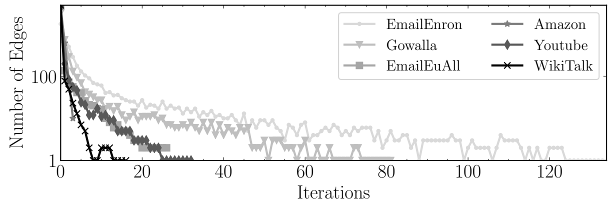
Our sequential truss maintenance and parallel truss maintenance algorithms are denoted as MSTruss and PM-



(a) Average degree vertices



(b) Low degree vertices



(c) High degree vertices

Fig. 5. The number of edges in the mixed structures at iterations.

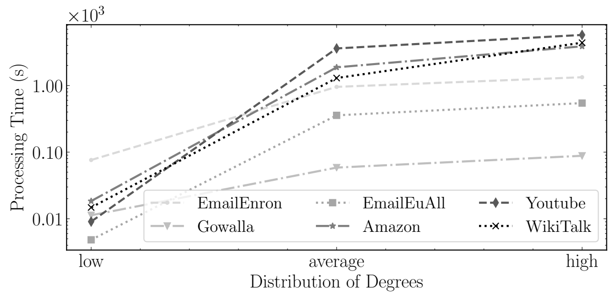
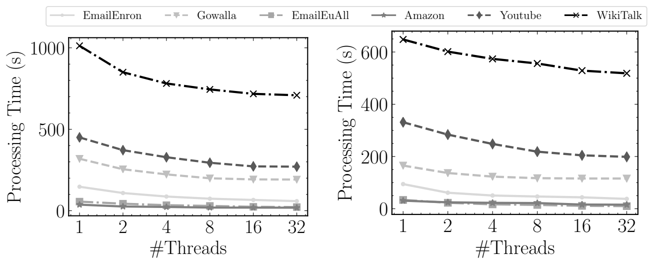


Fig. 6. The processing time under different degree distributions.

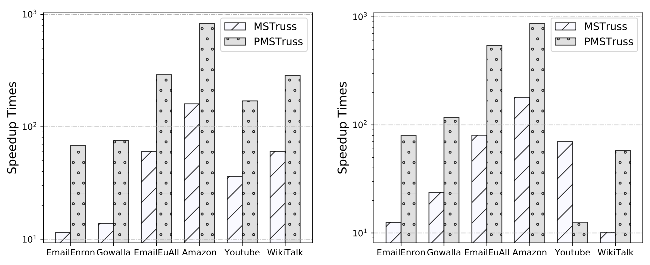
STruss, respectively. The algorithms in [24] for single vertex insertion/deletion are denoted as PP-truss, and the algorithms in [22] as TCP-Index, where PP-truss is implemented for vertex updates, while TCP-Index is for edge updates. We randomly select 10^4 nodes and 10^4 edges to compare the speedup rates of our two algorithms with those of PP&TCP-truss, where PMSTruss uses 32 threads.



(a) Insertion (b) Deletion

Fig. 7. The processing time of our parallel truss maintenance algorithms with different number of threads.

In Fig. 8, the x and y axes represent the datasets and the speedup rates, respectively. It can be seen that for all datasets, MSTruss can reduce the processing time by tens to hundreds of times, and PMSTruss can further improve the update efficiency. Furthermore, one can see that the acceleration is more significant in sparse graphs. In dense graphs, MSTruss needs more iterations to handle the same number of vertex and edge updates, and it also needs more time to construct a mixed structure.



(a) Insertion (b) Deletion

Fig. 8. The speedup times of our algorithms compared to PP&TCP-truss.

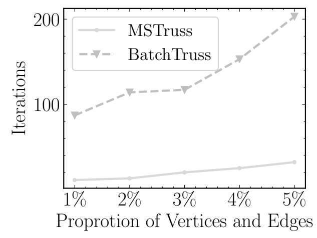
Furthermore, we compare the efficiency of MSTruss and BatchTruss [25], with both being batch processing approaches for truss maintenance. For the increment/decrement case, a TDS (Triangle Disjoint Set) is computed at each iteration of BatchTruss, and an MS (Mixed Structure) is computed at each iteration of MSTruss. We randomly choose $p_i\%$ vertices and $p_j\%$ edges as inserted/deleted ones. The other experiment settings are the same as before. Firstly, we compare the number of iterations taken by both algorithms with the same vertex and edge insertions/deletions. We illustrate the comparison results using the deletion case. The results are presented in Fig. 9. It can be seen that the number of iterations of BatchTruss is 12 times of that of the MSTruss on each graph, and the gap gets bigger as the number of deleted vertices and edges increases. This demonstrates that the mixed structure can contain much more edges compared to TDS.

We also compare the speedup times for both algorithms and the results are illustrated in Fig. 10. It can be seen that for all datasets, our algorithms can reduce the processing time by 2-14 times and 2-45 times in the insertion and deletion scenarios, respectively. When the number of inserted/deleted edges and vertices gets larger, the speedup gets more significant. Hence, our algorithms are more suitable for handling large amounts of updates.

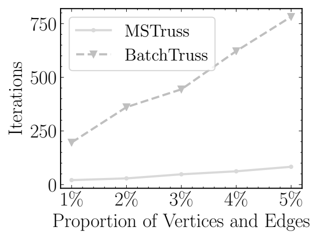
The experiment results show that our algorithms exhibit good efficiency and scalability. Our algorithms can significantly speed up the truss maintenance procedure compared to those single vertex and single edge algorithms. Additionally, they are suitable for handling a large number of vertex and edge insertions/deletions in large-scale graphs, which is essential in real-world scenarios.

6 Related Work

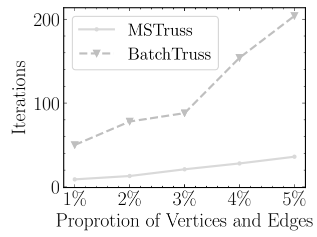
We provide a brief review on the existing algorithms for truss decomposition and truss maintenance.



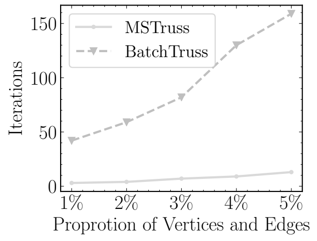
(a) EmailEnron



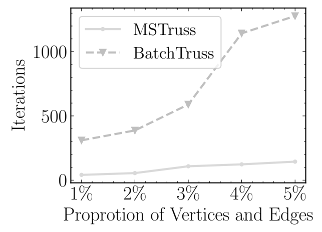
(b) Gowalla



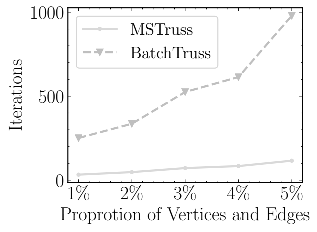
(c) EmailEuAll



(d) Amazon

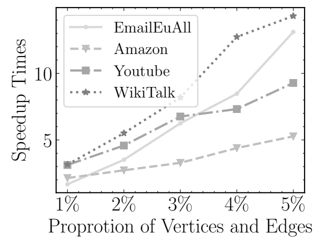


(e) Youtube

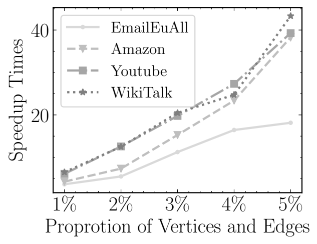


(f) WikiTalk

Fig. 9. The iterations of BatchTruss and MSTruss.



(a) Insertion



(b) Deletion

Fig. 10. The speedup times of our algorithms compared to BatchTruss.

Truss decomposition. The k-truss and truss decomposition were first presented in [12], which is a solution to find all k-trusses in a graph. Truss decomposition has been studied in various settings, including in-memory algorithms [27], external-memory algorithms [14], approximation algorithms [28] and distributed algorithms [29]. k-truss was also studied in specific graphs such as probabilistic graphs [30], bipartite graphs [31], and weighted graphs [32].

Truss maintenance. Zhou et al. [33] studied how to maintain trusses in evolving graphs. Huang et al. [22] proposed a tree-shape structure called TCP-Index, which supports efficient search of k-truss communities with a linear cost in dynamic graphs. Akbas et al. [23] presented an index structure *EquiTruss* based on truss-equivalence for searching EquiTruss-based communities. Ebadian et al. [24] developed an algorithm in public-private graphs, which can update k-truss with one vertex insertion. Zhang et al. [34] studied the boundedness problem of truss maintenance. Luo et al. [25] proposed a batch truss maintenance algorithm by presenting an edge structure called triangle disjoint set.

7 Conclusion

In this paper, we presented efficient batch processing algorithms for truss maintenance in fully dynamic networks. The proposed algorithms can handle inserted/deleted edges and vertices simultaneously, improving over existing approaches that only consider either edge or vertex updates of graphs. Our algorithms are based on the proposed mixed structure and the concept of pre-truss. The mixed structure helps quantify the edges' trussness change in the original graph, while the pre-truss can help get an initial trussness of the inserted edges, such that the trussness updates of those edges can be handled using the same approach as other edges. More importantly, our algorithms admit parallel implementations, which can further significantly improve truss maintenance efficiency. Extensive experiments on both real-world static and temporal graphs show that our algorithms are superior over previous ones, and the results demonstrate that our algorithms yield good efficiency and scalability.

Acknowledgments

This work was supported in part by the National Key Research and Development Program of China under Grant 2020YFB1005900 and in part by NSFC under Grant 62122042.

References

[1] Z. Cai, Z. He, X. Guan, and Y. Li, "Collective data-sanitization for preventing sensitive information inference attacks in social networks," *IEEE Trans. Dependable Secur. Comput.*, vol. 15, no. 4, pp. 577–590, 2018.
[2] T. Wei, C. Wang, and C. W. Chen, "Modularized morphing of deep convolutional neural networks: A graph approach," *IEEE Transactions on Computers*, vol. 70, no. 2, pp. 305–315, 2021.
[3] Z. Haohua, D. Yufu, Q. Weiyi, F. Wenjiang, and Z. Liqing, "Mining topology characteristics and evolution of large-scale software from core-shell structure," in *2009 Ninth International Conference on Hybrid Intelligent Systems*, vol. 2, 2009, pp. 45–48.
[4] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: applications in vlsi domain," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 7, no. 1, pp. 69–79, 1999.
[5] M. Nelson and P. E. Jordan, "Automatic reconfiguration of a ship's power system using graph theory principles," *IEEE Transactions on Industry Applications*, vol. 51, no. 3, pp. 2651–2656, 2015.
[6] M. Fyrbiak, S. Wallat, S. Reinhard, N. Bissantz, and C. Paar, "Graph similarity and its applications to hardware security," *IEEE Transactions on Computers*, vol. 69, no. 4, pp. 505–519, 2020.
[7] F. Sheng, Q. Cao, and J. Yao, "Exploiting buffered updates for fast streaming graph analysis," *IEEE Transactions on Computers*, vol. 70, no. 2, pp. 255–269, 2021.
[8] A. Das, M. Svendsen, and S. Tirthapura, "Incremental maintenance of maximal cliques in a dynamic graph," *VLDB J.*, vol. 28, no. 3, pp. 351–375, 2019.
[9] J. Abello, M. G. C. Resende, and S. Sudarsky, "Massive quasi-clique detection," in *LATIN*, ser. Lecture Notes in Computer Science, vol. 2286. Springer, 2002, pp. 598–612.
[10] S. B. Seidman, "Network structure and minimum degree," *Social Networks*, vol. 5, no. 3, pp. 269–287, 1983.
[11] S. B. Seidman and B. L. Foster, "A graph-theoretic generalization of the clique concept*," *Journal of Mathematical Sociology*, vol. 6, no. 1, pp. 139–154, 1978.

[12] J. Cohen, "Trusses: Cohesive subgraphs for social network analysis," in *National Security Agency technical report*, vol. 16, 2008, pp. 3–29.

[13] F. Zhao and A. K. H. Tung, "Large scale cohesive subgraphs discovery for social network visual analysis," *Proc. VLDB Endow.*, vol. 6, no. 2, pp. 85–96, 2012.

[14] J. Wang and J. Cheng, "Truss decomposition in massive networks," *PVLDB*, vol. 5, no. 9, pp. 812–823, 2012.

[15] M. Sozio and A. Gionis, "The community search problem and how to plan a successful cocktail party," in *KDD*. ACM, 2010, pp. 939–948.

[16] M. G. Rossi, F. D. Malliaros, and M. Vazirgiannis, "Spread it good, spread it fast: Identification of influential nodes in social networks," in *Proceedings of the 24th International Conference on World Wide Web Companion, WWW*. ACM, 2015, pp. 101–102.

[17] W. Zhu, M. Zhang, C. Chen, X. Wang, F. Zhang, and X. Lin, "Pivotal relationship identification: The k-truss minimization problem," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI*, 2019, pp. 4874–4880.

[18] A. E. Sariyüce and A. Pinar, "Fast hierarchy construction for dense subgraphs," *PVLDB*, vol. 10, no. 3, pp. 97–108, 2016.

[19] A. Caliò, A. Tagarelli, and F. Bonchi, "Cores matter? an analysis of graph decomposition effects on influence maximization problems," in *WebSci '20*, 2020, pp. 184–193.

[20] G. Ausiello, D. Firmani, and L. Laura, "Real-time anomalies detection and analysis of network structure, with application to the autonomous system network," in *International Wireless Communications and Mobile Computing Conference*. IEEE, 2011, pp. 1575–1579.

[21] Y. Che, Z. Lai, S. Sun, Y. Wang, and Q. Luo, "Accelerating truss decomposition on heterogeneous processors," *Proc. VLDB Endow.*, vol. 13, no. 10, pp. 1751–1764, 2020.

[22] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu, "Querying k-truss community in large and dynamic graphs," in *International Conference on Management of Data, SIGMOD*, Snowbird, UT, USA, 2014, pp. 1311–1322.

[23] E. Akbas and P. Zhao, "Truss-based community search: a truss-equivalence based indexing approach," *PVLDB*, vol. 10, no. 11, pp. 1298–1309, 2017.

[24] S. Ebadian and X. Huang, "Fast algorithm for k-truss discovery on public-private graphs," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI*, S. Kraus, Ed., 2019, pp. 2258–2264.

[25] Q. Luo, D. Yu, X. Cheng, Z. Cai, J. Yu, and W. Lv, "Batch processing for truss maintenance in large dynamic graphs," *IEEE Transactions on Computational Social Systems*, pp. 1–12, 2020.

[26] M. A. N. Al-hayanni, A. Rafiev, F. Xia, R. Shafik, A. Romanovsky, and A. Yakovlev, "Parma: Parallelization-aware run-time management for energy-efficient many-core systems," *IEEE Transactions on Computers*, vol. 69, no. 10, pp. 1507–1518, 2020.

[27] A. E. Sariyüce, C. Seshadhri, and A. Pinar, "Local algorithms for hierarchical dense subgraph discovery," *PVLDB*, vol. 12, no. 1, pp. 43–56, 2018.

[28] A. Conte, R. Grossi, A. Marino, and L. Versari, "Efficient estimation of graph trussness," *CoRR*, vol. abs/2010.00967, 2020.

[29] P. Chen, C. Chou, and M. Chen, "Distributed algorithms for k-truss decomposition," in *2014 IEEE International Conference on Big Data, Big Data 2014, Washington, DC, USA, October 27-30, 2014*, 2014, pp. 471–480.

[30] X. Huang, W. Lu, and L. V. Lakshmanan, "Truss Decomposition of Probabilistic Graphs: Semantics and Algorithms," in *SIGMOD*, 2016, pp. 77–90.

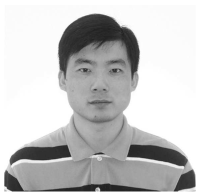
[31] Y. Li, T. Kuboyama, and H. Sakamoto, "Truss decomposition for extracting communities in bipartite graph," *International Conference on Advances in Information Mining and Management*, pp. 76–80, 2013.

[32] Z. Zheng, F. Ye, R. Li, G. Ling, and T. Jin, "Finding weighted k-truss communities in large networks," *Inf. Sci.*, vol. 417, pp. 344–360, 2017.

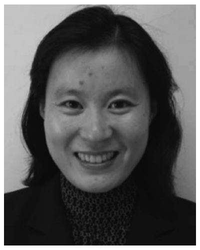
[33] R. Zhou, C. Liu, J. X. Yu, W. Liang, and Y. Zhang, "Efficient truss maintenance in evolving networks," *CoRR*, vol. abs/1402.2807, 2014.

[34] Y. Zhang and J. X. Yu, "Unboundedness and Efficiency of Truss Maintenance in Evolving Graphs," in *SIGMOD*, 2019, pp. 1024–1041.

Qi Luo received the B.S. and M.S. degrees in computer science from Northeastern University at Qinhuangdao, China, in 2015, and Shandong University, China, in 2018, respectively. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Technology, Shandong University. His research interests include graph analysis and distributed computing.



Dongxiao Yu received the B.S. degree in 2006 from the School of Mathematics, Shandong University and the Ph.D degree in 2014 from the Department of Computer Science, The University of Hong Kong. He is currently a professor in the School of Computer Science and Technology, Shandong University. His research interests include wireless networks, distributed computing and graph algorithms.



Xiuzhen Cheng received her M.S. and Ph.D. degrees in computer science from the University of Minnesota Twin Cities in 2000 and 2002, respectively. She is a professor in the School of Computer Science and Technology, Shandong University, Qingdao, China. Her current research interests include cyber physical systems, wireless and mobile computing, sensor networking, wireless and mobile security, and algorithm design and analysis.

Hao Sheng received the B.S. and Ph.D. degrees from the School of Computer Science and Engineering, Beihang University, Beijing, China, in 2003 and 2009, respectively. He is working on computer vision, pattern recognition, and machine learning.

Weifeng Lyu received the Ph.D. degree in computer science from Beihang University, Beijing, China, in 1998. He is a Professor, the Dean of the School of Computer Science and Engineering, and the Vice Director of the State Key Laboratory of Software Development Environment, Beihang University. His research interests include intelligent transportation and data analysis.