TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 05/14 pp873–887
DOI: 10.26599/TST.2022.9010019
Volume 28, Number 5, October 2023

Distributed Truss Computation in Dynamic Graphs

Ziwei Mo, Qi Luo*, Dongxiao Yu, Hao Sheng, Jiguo Yu, and Xiuzhen Cheng

Abstract: Large-scale graphs usually exhibit global sparsity with local cohesiveness, and mining the representative cohesive subgraphs is a fundamental problem in graph analysis. The k-truss is one of the most commonly studied cohesive subgraphs, in which each edge is formed in at least $k-2$ triangles. A critical issue in mining a k-truss lies in the computation of the trussness of each edge, which is the maximum value of k that an edge can be in a k-truss. Existing works mostly focus on truss computation in static graphs by sequential models. However, the graphs are constantly changing dynamically in the real world. We study distributed truss computation in dynamic graphs in this paper. In particular, we compute the trussness of edges based on the local nature of the k-truss in a synchronized node-centric distributed model. Iteratively decomposing the trussness of edges by relying only on local topological information is possible with the proposed distributed decomposition algorithm. Moreover, the distributed maintenance algorithm only needs to update a small amount of dynamic information to complete the computation. Extensive experiments have been conducted to show the scalability and efficiency of the proposed algorithm.

Key words: distributed algorithm; dynamic graph; graph mining; cohesive subgraph; k-truss

1 Introduction

As a data structure expressing relationships between entities, graphs have been extensively used in modeling social, communication, and information networks that appear in a variety of applications. In particular, the fundamental task of detecting and searching for cohesive components in large-scale graphs has attracted considerable attention from research to industry because cohesive subgraphs can be a critical reflection of the compact features and key parts of the entire graph[1–3]. The densest subgraph group in the mining graphs is an non-determinstic polynomial (NP)-hard problem; thus, many relaxation versions for the group, such as quasi-clique[4], k-core[5], k-plex[6], k-club[7], and k-truss[8], have been proposed to maintain cohesiveness while improving computational efficiency. A favorable tradeoff between computational efficiencies and cohesiveness, that is, k-truss, which considers the measurement of the closeness of two people in a social network, plays an important role in many applications, such as community search[9, 10], employee training[11], circle detection[12], and social contagion[13].

A k-truss is a cohesive subgraph that requires the containment of each edge in at least $k-2$ triangles. The relevant concept of k-truss is trussness, which is defined on edges as the maximum value of k such that an edge e is in a k-truss but not in a $(k+1)$-truss. The k-truss decomposition problem lies in the computation of the trussness for all edges in graphs. The most common approaches for computing the trussness are based on the peeling-like algorithm[14], which can compute trussness

• Ziwei Mo, Qi Luo, Dongxiao Yu, and Xiuzhen Cheng are with the School of Computer Science and Technology, Shandong University, Qingdao 266200, China. E-mail: hbdong@sdu.edu.cn; luoqi2018@mail.sdu.edu.cn; dxyu@sdu.edu.cn; xzcheng@sdu.edu.cn.
• Hao Sheng is with State Key Laboratory of Software Development Environment, School of Computer Science and Engineering and the Beijing Advanced Innovation Center for Big Data and Brain Computing, Beihang University, Beijing 100191, China. E-mail: shenghao@buaa.edu.cn.
• Jiguo Yu is with the Big Data Institute, Qilu University of Technology (Shandong Academy of Sciences), Jinan 250353, China. E-mail: jiguoyu@sina.com.
∗ To whom correspondence should be addressed.
Manuscript received: 2022-03-14; revised: 2022-05-10; accepted: 2022-06-14

© The author(s) 2023. The articles published in this open access journal are distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/).

in $O(m^{1.5})$ time (where m is the number of edges in the graph). Most algorithms currently adopt the parallel processing mode of dividing graphs or GPU acceleration to speed up the calculation process of trussness. Figure 1 shows an example of a k-truss.

With the increasing scale of graphs in the real world, the limitations of a single machine and memory markedly affect the efficiency of data processing, and data processing in a single machine or memory becomes impossible in numerous scenarios. Another issue that affects the truss computation is the dynamicity of graphs, that is, the graph topology may change over time. For example, edges and nodes are constantly inserted into and removed from the temporal graphs. A graph can have billions of nodes and edges, Thus, recomputing the trussness for all edges may incur a large number of redundant computations when only a small part of the graph is changed. Hence, the problem of k-truss maintenance, that is, effectively updating trussness after the graph changes, has been proposed[15].

However, maintaining trussness in dynamic graphs is challenging. Figure 2 shows that the number and changes affecting the trussness of edges are different for the same number of edges inserted in batches. In Fig. 2b, inserting two edges $(2, 9)$ and $(2, 6)$ into the original graph increases the trussness of two edges from 2 to 3 and that of five edges from 3 to 4, respectively. While in Fig. 2c, two edges (5, 8) and (6, 9) are inserted in the same graph, demonstrating a different change in the trussness of edges in the original graph from that in Fig. 2b. The main difficulties for truss maintenance are primarily from two aspects. First, determining which edge will change the trussness is difficult. This study shows that not only the edges directly connected to the inserted/deleted edge may change the trussness, but also the other edges. Second, determining the changes in trussness of an edge after insertion/deletion of edges is difficult. If the same number of edges are inserted, then the changes in trussness can be different.

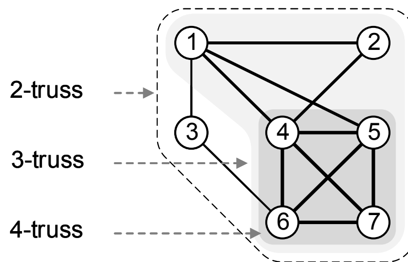


Fig. 1 Example of k-truss.

We propose a distributed truss ecomposition algorithm for trussness decomputation in static graphs and a distributed truss maintenance algorithm for trussness maintenance in dynamic graphs to tackle the above challenges. The current work is inspired by node-centric graph processing models, such as Pregel[16], GraphLab[17], and GPSA[18], which limit algorithms for a single node to manipulate local graph structures. On this basis, we extend the node-centric graph computation model and propose two indexes for each node, which effectively implements the distributed truss decomposition and maintenance algorithm. In our algorithms, each node updates its local data according to the receiving messages from adjacent nodes. The system runs in a synchronous environment until all nodes no longer update data and then the system stops. The distributed truss decomposition algorithm maximizes the local property of the k-truss, and completion of the computation for a graph with 10 million nodes only takes approximately 20 rounds. The distributed truss maintenance algorithm, which is based on decomposition, can also complete the computation quickly by resetting the initial values after the graph change. The following presents the original contributions of this study to truss calculations in several important aspects.

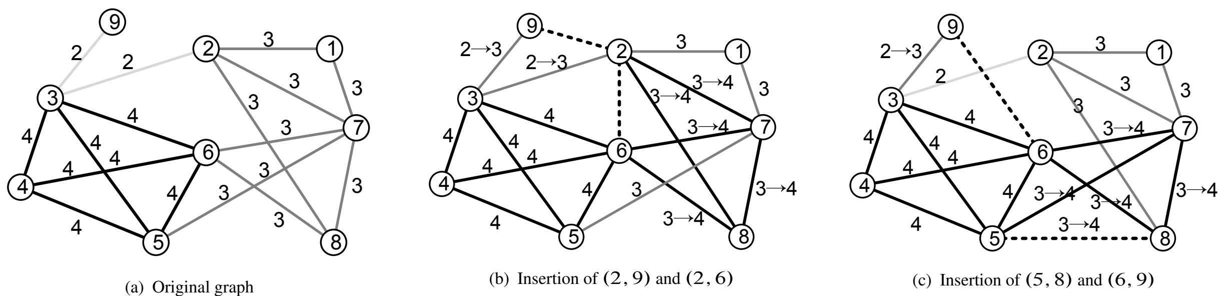


Fig. 2 Example of inserting the same number of edges in the graph but with different trussness changes. In graphs, the values marked on the edges are the trussness of the edges, and the edges with different trussness are distinguished by the thickness of the edges. A large edge trussness is indicated by thick lines in the graphs.

• We study the distributed algorithms for computing trussness. The proposed synchronous distributed model relies on the local property of the k-truss, where nodes only need to focus on neighboring topology and can process large-scale graphs. Based on the distributed model, we propose two indexes, *trussMap* and *adjMap* for each node to record the status. Nodes update the two indexes by exchanging information with neighboring nodes in each round.

• We propose a top-down distributed truss decomposition algorithm, which maximizes the local properties of the k-truss and can significantly reduce the number of computational rounds. The distributed truss maintenance algorithm requires remarkably few node-aware dynamic changes to maintain the computation. A detailed analysis proves the accuracy and effectiveness of the proposed algorithm.

• We conduct comprehensive experiments on synthetic, real-world, and temporal graphs. The proposed distributed truss decomposition algorithm can compute the trussness of all edges in most graphs using only tens of rounds. In particular, the proposed algorithm is about twice as fast as the baseline in dense graphs. The distributed truss maintenance algorithm also shows good scalability and efficiency.

Roadmap. This paper is organized as follows. Section 2 surveys the works related to k-truss decomposition and maintenance. Section **??** introduces preliminaries regarding conceptions about k-truss and problem definition. Sections 4 and 5 present the distributed truss decomposition algorithm and distributed truss maintenance algorithm, respectively. In Section 6, the experiments and results on synthetic, real-world and temporal graphs are illustrated. Finally, Section 7 concludes this paper and discusses the future works of truss computation.

2 Related Work

As k-truss was firstly presented in Ref. [8], A steady stream of research emerged with the introduction of the k-truss in Ref. [8]. We briefly surveyed studies considering k-truss in this section.

Truss decomposition. In recent years, numerous solutions to truss decomposition, including sequential, distributed, and parallel ones[14, 19–22], have been presented. Cohen[8, 23] found the maximal k-truss in graphs and provided a method for truss decomposition and processing large-scale graphs using streaming graphs on a single machine. Wang and Cheng[14] and Che et al.[24] proposed in-memory truss computation algorithms, and two I/O-efficient algorithms in large-scale graphs. Sariyüce et al.[20, 21] provided "efficient" algorithms to construct the hierarchical structure based on k-truss, and then presented parallel local algorithms for approximated trussness by reducing traversal vertices.

Considerable attention has been provided to distributed and parallel processing techniques for truss decomposition[23, 25–28]. Chen et al.[25] proposed a parallel truss decomposition algorithm based on the MapReduce framework. Shao et al.[28] introduced a novel parallel and efficient truss detection algorithm. Smith et al.[29] presented a parallel algorithm for computing truss decomposition on shared memory systems. Kabir and Madduri[26, 27] proposed a new shared-memory parallel algorithm for truss decomposition of large sparse graphs.

Some works for other types of graphs, such as uncertain graphs[30] and bipartite graphs[31], are depended on special structures. Huang et al.[32] proposed local and global (k,γ)-truss models in probabilistic graphs and developed efficient algorithms for decomposing a probabilistic graph into such maximal (k,γ)-trusses. Esfahani et al.[30] employed a special version of the Central Limit Theorem to obtain the peeling algorithm for truss decomposition of a probabilistic graph that measures very large-scale graphs and offers significant improvements. Zou et al.[31] proposed a novel method for extracting k-truss communities embedded into a bipartite graph. The k-truss decomposition in hypergraphs and attributed graphs[33] is also studied.

Truss maintenance. The authors of this paper believe that minimal attention has been provided to truss maintenance compared with truss decomposition[15, 34–37]. Reference [22] introduced a type of subgraph called Triangle K-Core, which is the prototype of the combination of k-truss and k-core. They also presented the algorithms for computing the Triangle K-Core in static and dynamic graphs. Zhou et al.[36] studied the problem of truss maintenance considering the insertion and deletion of one edge and proposed serial properties of trussness update after changes in one edge changes. The problems of truss-based community search have also been investigeted in Refs. [15, 34]. They proposed the truss-based index via computed k-truss and maintained these indexes in the dynamic graphs. More specifically,

Ref. [15] studied dynamic community search problems with frequent edge insertion and deletion based on k-truss. Zhang and Yu[35] examined the boundedness of truss maintenance problems and argued that the removal and insertion algorithms are boundedness and unbounedness, respectively. Luo et al.[37] proposed a batch processing algorithm based on a triangle disjoint set, which allows up to one change in edge trussness after the graph evolution.

Most of the existing computations on k-truss focus on static graph decomposition, and most are algorithms on a single machine. Moreover, only a small amount of research on the maintenance of k-truss in dynamic graphs has been conducted. This paper focused on k-truss decomposition in static graphs using multicore machines and maintained k-truss in dynamic graphs.

3 Preliminary

We consider an undirected and unweighted graph $G = (V, E)$, where V is the node set, and E is the edge set. Let $n = |V|$ and $m = |E|$ be the number of nodes and the edges, respectively. Denote the neighbors of node u by $N(u)$. Let $EN(e)$ be the adjacency edges of e, which are edges that share a common node with e. Let $EN(v)$ denote the incident edges of v. Denote the triangle with endpoint u, v, and w by $\triangle_{uvw}$. Let H be a node-induced subgraph of G with node set $V(H)$ and edge set $E(H)$.

Definition 1 (support). The support of edge e in G, which is denoted as $sup_G(e)$, is defined as the number of triangles containing e.

The support of $e = (u, v)$ can be calculated by the number of common neighbors of u and v as follows.

$$sup_G(e) = |\{\triangle_{uvw} : \triangle_{uvw} \in G\}| = |N(u) \cap N(v)| \tag{1}$$

Definition 2 (*k*-truss). A k-truss is a maximal connected node-induced subgraph denoted as H, in which the support $sup_H(e)$ of each edge e is no less than $k-2$.

The definition of k-truss indicates that a k-truss is not a subgraph of another k-truss. The trussness of an edge e, which is denoted as $t(e)$, is then defined as the value of maximum k such that e is in a k-truss. Let $H \subset G$ be a k-truss, and then $sup_H(e) \geqslant k-2$ for each $e \in E(H)$. The trussness of $e = (u, v)$ can be computed in accordance with the definition of k-truss using the following equation.

$$t(e) = \arg\max_{k \geqslant 2}\{|\{w|t^w_{\min} \geqslant k\}| \geqslant (k-2)\} \tag{2}$$

where $w \in N(u) \cap N(v)$ and $t^w_{\min} = \min(t(u, w), t(v, w))$.

Definition 3 (triangle connected). Given two edges e_1, e_2 in G, if $e_1 \in \triangle_{[1]}, e_2 \in \triangle_{[t]}$ and a set of triangles $\triangle_{[1]}, \triangle_{[2]}, \ldots, \triangle_{[t]}$ exist, where $t \geqslant 1$ and every two adjacent triangles in the triangle set share a common edge, then e_1 and e_2 are called triangle connected.

If two edges are triangle-connected, we call the set of triangles a triangle path between the two triangles. A k-triangle is a triangle whose trussness of three edges is not less than k, which is denoted as $\triangle^k$.

Definition 4 (*k*-triangle connected). If two edges e_1, e_2 are triangle connected, $t(e_1) = t(e_2) = k$, and the triangles in the triangle path from e_1 to e_2 is k-triangles, then we call that e_1 and e_2 are k-triangle connected.

The notations and descriptions are summarized in Table 1. Subscripts are omitted when the context is clear.

Distributed model. Our model uses the bulk synchronous parallel model for distributed processing, where each iteration considers the computational process to be completed from the perspective of a single node (i.e., user-defined node programs). Each node has two states: active and inactive. Only the active nodes need to participate in the computation in each iteration. The model then uses message passing to communicate between nodes; that is, a node program allows a node to send messages to neighboring nodes. The node updates its state and compute related data based on the received messages.

Problem definition. Given a graph G, we study the truss computation problem based on the following steps: (1) we decompose and calculate the trussness of all edges in G; (2) we maintain and recompute the trussness of all edges in the evolving graph after the insertion/deletion of a batch of edges.

4 Distributed Truss Decomposition

We propose our distributed truss decomposition algorithm in this section based on our distributed model.

Table 1 Notations and descriptions.

Notation	Description
$G = (V, E)$	Graph G with node set V and edge set E
$e = (u, v)$	Edge with u and v as endpoints
$\triangle_{uvw}$	Triangle with three nodes u, v, and w
$N_G(u)$	Set of neighbors of u in G
$EN_G(e)$	Set of neighboring edges of e in G
$EN_G(v)$	Incident edges of v in G
$sup_G(e)$	Support of e in G
$t_G(e)$	Trussness of e in G
$\hat{t}(e)$	Estimate trussness of e

We first introduce two auxiliary indexes, namely *adjMap* and *trussMap*, which are based on the locality property of the k-truss. We then introduce the procedure and analysis of the proposed algorithm, which is based on the two indexes.

The distributed model encounters the following two truss decomposition challenges.

• The most important metric for calculating the trussness is the support of edges, which is the initial value of trussness for edges. All neighboring nodes of both endpoints of the edge must be identified to calculate the support of an edge.

• Edges are virtual connections for communication between nodes. Thus, the value of support for each edge cannot exist on a nonexistent entity.

To address the above challenges, we calculate the support value of all neighboring edges by utilizing the node to store local topological information regarding its incident edges and two-hop neighboring nodes. Furthermore, the following property bears the calculation of the trussness applied to the support of the edges. The locality property is formally presented as follows.

Property 1 (Locality[25]). Given a graph $G = (V, E)$, $\forall e \in E$, $t(e) = k$ if and only if the following conditions are met.

(1) An edge subset $E_k \subset EN(e)$ such that $|E_k| = 2(k-2)$, edges in E_k form total $(k-2)$ triangles with e, and the trussness of each edge in E_k is not less than k.

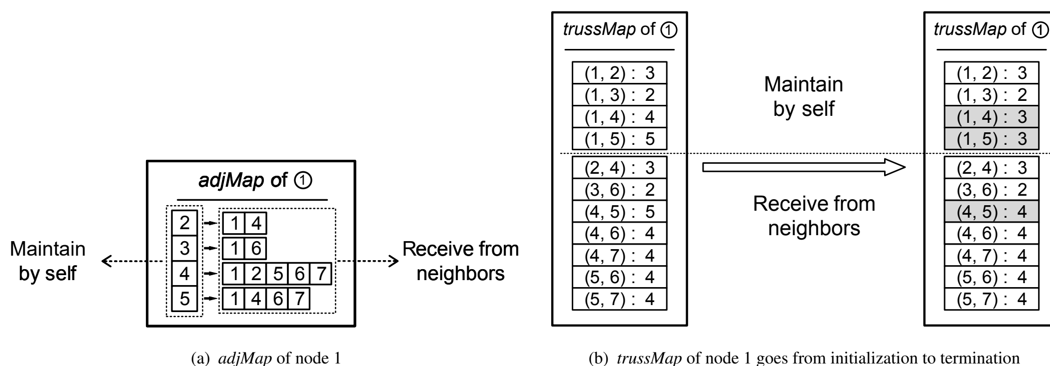
(2) There is not a subset $E_{k+1} \subset EN(e)$ does not exist such that $E_{k+1} = 2(k-1)$, edges in E_{k+1} form total $(k-1)$ triangles with e, and the trussness of each edge in E_{k+1} is not less than $k+1$.

The property describes that if the trussness of an edge e is equal to k, then at least $(k-2)$ triangles contain e and the trussness of each edge in these triangles is at least k. Therefore, the trussness of an edge can be calculated from the trussness of the incident edges of the edge. More specifically, the trussness can be calculated from the trussness of the edges that form triangles with the edge.

We design two indexes according to the locality property to store the neighboring information for each node. For a node u in our distributed system, the first index is *adjMap*, which is a list of key-value pairs used to store 2-hop adjacency nodes of u. The keys of *adjMap* are the neighbor nodes of u, and the value corresponding to each key is the linked list of neighboring nodes of the stored node by the key. The second index is *trussMap*, which stores the estimated trussness of 2-hop incident edges of a node u. The trussness of the incident edges of u is calculated and maintained by u. Figure 3 shows an example of the *adjMap* and the *trussMap* for node 1 in Fig. 2. In Fig. 3a, the keys of *adjMap* are maintained by node 1, and the values are received from neighboring nodes of node 1. Figure 3 shows the change of node 1 updating the *trussMap* after receiving messages from neighbors. The trussMap comprises two parts: one part is the trussness estimates of the incident edges of node 1 (this part is calculated and maintained by node 1 itself), and another part is the trussness estimates of the incident edges of node 1's neighboring nodes (this part is kept by node1 receiving data from the neighboring nodes).

In our distributed model, each node is responsible for computing and updating the estimated trussness of its incident edges by maintaining *trussMap*. Algorithm 1 shows the execution process of node u. The proposed algorithm has two phases: the initialization and the execution phases.

In the initialization phase (lines 1–8), the adjacency nodes of u exchange their *adjMap* values with u to compute *trussMap*. The initially estimated trussness of an edge is the support of the edge plus 2 (line 6). u sends its *trussMap* to the neighboring nodes and sets the



(a) *adjMap* of node 1

(b) *trussMap* of node 1 goes from initialization to termination

Fig. 3 2-hop auxiliary index for node 1 in Fig. 2.

Algorithm 1 Procedure of distributed truss decomposition

```
Input: N(u): the neighbor nodes u
Output: trussMap(e) for e ∈ EN(v)
 1: procedure INITIALIZATION
 2:     adjMap[u] ← N(u);
 3:     send adjMap[u] to v for v ∈ N(u);
 4:     receive adjMap[v] for v ∈ N(u);
 5:     for all v ∈ N(u) do
 6:         trussMap[(u, v)] ← |N(u)| ∩ |N(v)| + 2;
 7:     send trussMap to v ∈ N(u);
 8:     Changed ← False;
 9: procedure EXECUTION
10:     repeat
11:         receive trussMap;
12:         counter = [];
13:         for all w ∈ N(u) ∩ N(v), v ∈ N(u) do
14:             k_m = min{trussMap[(u, w)], trussMap[(v, w)]};
15:             counter.add(k_m);
16:             t = ComputeTruss(counter, trussMap[(v, u)]);
17:             if t < trussMap[(u, v)] then
18:                 trussMap[(u, v)] ← t;
19:                 Changed ← True
20:         if Changed then
21:             send trussMap[(u, ∗)] to v ∈ N(u);
22:     until termination condition
```

Changed flag to false to end the process (lines 7 and 8).

The execution phase is the process of iterative execution of the entire distributed system (lines 9–22). In each round of the execution phase, u receives the trussness of incident edges to neighbor nodes (line 11). Thus, each node can receive the trussness of the 2-hop incident edges. Afterward, u updates the trussness of incident edges through the locality property by the function ComputeTruss (line 16). This function then returns the maximum k for which a value not less than $k-2$ exists in the counter, and the algorithm is shown in Algorithm 2. If the updated estimated trussness t of the incident edges is smaller than the value of the current record, then the value in *trussMap* is updated,

Algorithm 2 ComputeTruss(counter, *k*)

```
 1: for i = 1 to k do
 2:     count[i] ← 0;
 3: for t ∈ counter do
 4:     j ← min{k, t};
 5:     count[j] ← count[j] + 1;
 6: for i = k downto 2 do
 7:     count[i − 1] ← count[i − 1] + count[i];
 8: i ← k;
 9: while i > 2 and count[i] < i − 2 do
10:     i ← i − 1;
11: return i;
```

and the *Changed* flag is set to true (lines 17–19). The updated *trussMap* is sent to the neighboring nodes of u (lines 20 and 21) after all the incident edges of u with their updated trussness are obtained. The steps of the execution phase are repeated until the termination condition is met and the system stops. Finally, the stored values in *trussMap* are the trussness of the edges.

Termination mechanism. Several alternatives for termination conditions of the distributed computing system are avaliable[38]. We use the barrier synchronization mechanism as the termination condition. If none of the nodes in the system updates *trussMap* in a round, that is, no active nodes are avaliable, then the system will stop, and the computation of trussness is complete.

Efficient message strategy. Two strategies that can further reduce the volume of passing messages are avaliable.

(1) The support of an edge is determined by the number of triangles where the edge is contained. Therefore, some edges may not be in any triangle and hence will not participate in any computing. Thus, the trussness of these edges will not be stored in the *trussMap*.

(2) The *trussMap* maintained by each node stores the estimated trussness of all incident edges. In the phase of sending messages in each round, the entire *trussMap* is sent to the neighbor nodes. This approach is substantially inefficient because only some items in *trussMap* may have changed. Theregore, we only need to send the updated items in *trussMap* to the neighboring nodes.

We then analyze the accuracy and efficiency of Algorithm 2.

Lemma 1. In Algorithm 1, the estimated trussness $\hat{t}(e)$ of e is always higher than $t(e)$.

Proof. Assuming $e=(u,v)$ and $t(e)=t$, at least $t-2$ triangles contain e, and the trussness of each edge in these triangles is no less than t according to the definition of k-truss. Assume e' is an edge that forms a triangle with e, $t(e')\geqslant t$, and $\hat{t}(e)<t$ at time t_1. This assumption indicates that the endpoint u or v updates $\hat{t}(e')$, which causes $\hat{t}(e')<t$ at time t_2. By analogy, the endpoints of e' update some estimated trussness of the edges that form the triangles with e' at t_3. The inference leads to an infinite sequence of $t_1>t_2>t_3>\cdots>t_i$. However, this sequence has a loop that yields $t_i=t_1$, which is a contradiction. ■

Lemma 2. Algorithm 1 ensures the convergence of the estimated trussness $\hat{t}(e)$ to $t(e)$ for $\forall e\in E$.

Proof. In any round during the entire process, $\hat{t}(e)$ will only decrease and will be no less than $t(e)$ by Lemma 1. Therefore, we only need to prove that $\hat{t}(e)$ will eventually be equal to $t(e)$ for any edge e. We set $\hat{t}(e) = sup(e)+2$ in the initialization phase and provide proof by induction on the values of $t(e)$.

- $t(e) = 2$. In this case, $sup(e) = 0$, that is, e is not in any triangles. $\hat{t}(e) = t(e)$ in the beginning of the distributed system.
- $t(e) = 3$. Assume that $\hat{t}(e) \geqslant 4$ always holds. Therefore, each edge in at least two triangles has trussness no less than 4. Therefore, a subgraph that contains these edges is avaliable, and the support of each edge is larger than 2. The subgraph is a 4-truss, which is a contradiction.
- Induction step: suppose there is an edge e such that $t(e) = k \geqslant 4$ and $\hat{t}(e) \geqslant k+1$ forever constantly. Then, $l \geqslant k-2$ triangles contain e and for each $e' \in l \backslash \{e\}$ such that $t(e') \geqslant k$. According to Lemma 1, $\hat{t}(e') \geqslant k+1$ by $\hat{t}(e) \geqslant k+1$.

If $l = k-2$, then $\hat{t}(e)$ will eventually decrease to $t(e)$ according to Property 1.

If $l \geqslant k-1$, then e is in at least $k-1$ triangles and $e' \in l \setminus \{e\}$ such that $\hat{t}(e') \geqslant k+1$. Therefore, $t(e) = k+1$ according to the definition of k-truss, which is a contradiction. ■

Theorem 1. The number of rounds of Algorithm 1 is bound by $2|E| + \sum_{e \in E}(sup(e) - t(e))$.

Proof. As long as this distributed system is running, at least one edge will perform the update operation of estimated trussness. In the worst case, only one edge of the estimated trussness is updated in each round, and this edge only changes by 1. For an edge e, the quantity by which its estimated trussness decreases from its initial value to its final value is $(sup(e) + 2 - t(e))$. This quantity represents the update error of e. Thus, the execution time is bound by the sum of update errors of all edges, which is $\sum_{e \in E}(sup(e) + 2 - t(e)) = 2|E| + \sum_{e \in E}(sup(e) - t(e))$. ■

From the above lemmas and theorem, we derive the following Theorems regearding the time and information complexities of the distributed truss decomposition algorithm.

Theorem 2. Given a graph G, the time complexity of Algorithm 1 to compute the trussness of edges is $O(m - t_{\min})$, where m is the number of edges in G and $t_{\min}$ is the minimum trussness of the graph.

Proof. As the system is running, at least one edge whose estimated trussness will be updated in each round exists. This condition indicates that at least one minimum trussness must be determined in each round. After the trussness of all edges has been determined, the system requires one round of initialization and one round of deterministic termination. Hence, the number of rounds of the algorithm is no larger than $m - t_{\min} + 2$, and the time complexity is $O(m - t_{\min})$. ■

Theorem 3. Given a graph G, the message complexity of Algorithm 1 to compute the trussness of edges is $O(\sum_{e \in E} sup(e))$.

Proof. In the worst case, one of the endpoints of an edge e receives at most $|sup(e) + 2 - t(e)|$ messages from the neighboring nodes. In whatever graph, the minimum value of trussness of edges is 2. Therefore, the number of messages of the algorithm is bound by $\sum_{e \in E}(sup(e) + sup(e')) = 2\sum_{e \in E}(sup(e))$, and the message complexity is $O(\sum_{e \in E} sup(e))$. ■

5 Distributed Truss Maintenance

We consider the truss maintenance problem in dynamic graphs in this section. In dynamic graphs, insertion and deletion of edges lead to changes in the graph structure and trussness values of the edges. The problem to be addressed in this section lies in the maintenance of the trussness values of edges in dynamic graphs. We first consider the insertion and deletion of one edge and the case of multiple edge insertion and deletion.

As described in the previous section, the basic idea of the distributed truss decomposition algorithm is to decrease the estimated trussness of each edge from $sup(e) + 2$ to $t(e)$. Similarly, in a dynamic graph, if we initially provide a suitable value of estimated trussness to the newly inserted edges and the edges whose trussness may be affected, then computing the trussness of all edges will be efficient for nodes.

A new graph G' will be generated after the insertion or deletion of edges from G. The trussness of $e \in G'$ (except inserted edges) may change due to two reasons: (1) the formation or breakage of new triangles containing e; (2) the increase or decrease of trussness of edges that form triangles with e.

According to the aforementioned reasons, the main idea of distributed truss maintenance is to recalculate the trussness of those edges that may change, thus saving a considerable amount of computational resources. We

will first introduce how trussness will be updated after one edge is inserted or deleted. We will then discuss the insertion and deletion of multiple edges. The deletion and insertion of nodes can be regarded as the deletion and insertion of edges in iteration. Thus, we will omit the analysis process.

5.1 One edge dynamic

We first introduce the properties related to the trussness update of the insertion/deletion of an edge. The distributed algorithms for truss maintenance are then presented.

At most, one triangle is formed for destroyed after the insertion/deletion of an edge. The support of an edge is the number of triangles contained in that edge, and the trussness of an edge is related to the minimum support of the surrounding edges. The insertion or deletion of an edge may cause an increase or decrease in the trussness of other edges by at most 1, which has been proven in Ref. [15]. A new graph G' is formally generated after inserting e_0 in G. The increase in the trussness of edge e in G' may be performed in two ways: (1) e, e_0, and another edge form a new triangle, which increases the support of e; (2) the trussness of some edges of the triangle in which e is located increases by one. After deleting e_0 in G, the trussness of edge e in the newly generated graph G' may decrease in two ways: (1) e and e_0 belong to the same triangle in G, and the deletion of e_0 reduces the support of e; (2) the trussness of some edges in the triangle where e is located decreases by one.

For the trussness of the inserted edge $e_0 = (u, v)$, we set the bounds for e_0 based on the trussness of the edges before the insertion of e_0. Assume $t_{\mathrm{LB}}(e_0)$ is the lower bound of $t(e_0)$ and $t_{\mathrm{UB}}(e_0)$ is the upper bound of $t(e)$. According to the definition of k-truss,

$$t_{\mathrm{LB}}(e_0) = \arg\max_{k \geqslant 2}\{|\{w|t^{w}_{\min} \geqslant k\}| \geqslant (k-2)\} \quad (3)$$

where $w \in N_G(u) \cap N_G(v)$. The trussness of edges forming triangles with e_0 will increase by at most 1. Therefore, we can deduce that

$$t_{\mathrm{UB}}(e_0) - t_{\mathrm{LB}}(e_0) \leqslant 1 \quad (4)$$

The above can be formally summarized as the following property[37].

Property. If inserting edge $e_0 = (u, v)$ to $G = (V, E)$, then $e_1, e_2 \in E$ may increase trussness by 1, where $t(e_1) = k < t_{\mathrm{UB}}(e_0)$ and e_2 is k-triangle connected with e_1. If deleting edge e_0 from $G = (V, E)$, then $e_1, e_2 \in E \setminus e_0$ may decrease trussness by 1, where $t(e_1) = k \leqslant t(e_0)$ and e_2 is k-triangle connected with e_1.

An example of inserting edges is shown in Fig. 2, which demonstrates the formation of new triangles after edge insertion. The trussness of part of the edges may increase due to the new triangles.

After an edge $e_0 = (u, v)$ is inserted, the two endpoints u and v initially update their *adjMap* and send the updated *adjMap* to their neighbors. Then, the two nodes both calculate the upper bound of e_0 (i.e., $t_{\mathrm{UB}}(e_0)$). An edge e will be marked if e forms a triangle with e_0 and $t(e) < t_{\mathrm{UB}}(e_0)$, and these edges will also be marked if they are k-triangle connected with e. All marked edges are potential edges whose trussness may increase by 1. Therefore, we increase the trussness estimates in the *trussMap* of these potential edges by 1, and perform the same process as Algorithm 1 for the system until the convergence of the trussness estimates of all edges to the final stable values.

We classify the endpoints of edges that may update trussness into two categories. (1) The nodes are the endpoints of the inserted edges, denoted by promoter node set; (2) the nodes are the endpoints of those edges that are k-triangle connected to the inserted edges, denoted by spread node set. Each node in the promoter node set is a promoter node, and each node in the spread node set is a spread node.

The promoter nodes are responsible for computing the upper bound trussness of inserting edges and recording edges that may be affected. Meanwhile, the spread nodes are responsible for sending messages to the nodes where the requirements are met. We first describe the distributed algorithm for inserting an edge, and then briefly discuss the algorithm for deleting one edge. Figure 4 shows an example of promoter nodes and spread nodes. After inserting edge $(2, 6)$ in the graph, nodes 2 and 6 become promoter nodes, and the initial trussness of the newly inserted edge $(2, 6)$ is 3. Neighboring nodes connected to nodes 2 and 6 by trussness not larger than 3 become spread nodes.

Distributed truss maintenance with one-edge

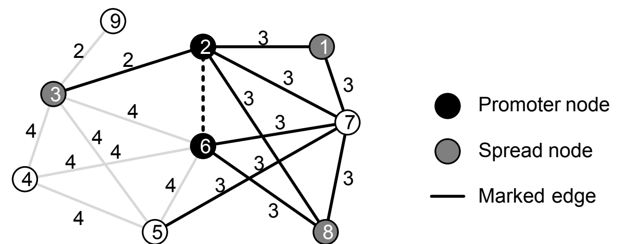


Fig. 4 An example of promoter nodes and spread nodes.

insertion. In our algorithm, each node maintains two auxiliary sets: updated edge set (UES) is a set that stores the neighboring edges of the node, and the trussness of these neighboring edges may change; updated node set (UNS) is a set of nodes that stores the nodes adjacent to this node and the *trussMap* of these nodes may change.

For the promoter node, the edges stored in its *UES* are all the edges whose trussness is lower than the upper bound of the inserted edge's trussness; its *UNS* stores another endpoint of those edges in its *UES*. The edges stored in *UES* are those whose trussness may increase by 1. We reinitialize the trussness estimates of these edges to their current trussness plus 1. All nodes in the system perform the same process as in Algorithm 1 after the reinitialization is completed. The initialization pseudocodes of the promoter and spread nodes are shown in Algorithms 3 and 4, respectively.

For a promoter node u, the procedure is divided into two parts, prepare phase (lines 1–11 in Algorithm 3) and propose phase (lines 12–16 in Algorithm 3). Due to the insertion of a new edge e_0, the *adjMap* of u is first updated, and then compute the upper bound of e_0 (lines 2–5 in Algorithm 3). Each edge that forms a triangle with e_0 is then traversed: if the edge satisfies the condition for possible updates, then the edge is added to UES, and another endpoint of the edge is added to UNS (lines 6–11 in Algorithm 3). In the Propose phase, the trussness of each edge in UES plus one, and each node in UNS spreads the *Change* signal (lines 12–16 in Algorithm 3).

For a spread node p, the procedure is still divided into two parts. In prepare phase, the edge whose trussness may change (lines 3 and 4 in Algorithm 4) is first determined. Each edge that forms a triangle with the potential edge and has the same trussness as the potential edge is then added to the UES (lines 5–11 in Algorithm 4). The propose phase of the spread node is the same as the promoter node, and the difference lies in the additional trussness of edges by 1 that no longer changes.

Algorithm 3 Re-initialize stage: promoter node *u*

```
Input: adjMap, trussMap, e0 = (u, v)
Output: adjMap, trussMap
 1: procedure PREPARE
 2:     adjMap[u].add(v);
 3:     trussMap[(u, v)] ← t_UB(e0);
 4:     send adjMap[u] to v ∈ N(u);
 5:     receive adjMap[v] for v ∈ N(u);
 6:     UES ← ∅;
 7:     UNS ← ∅;
 8:     for all w ∈ N(u) ∩ N(v) do
 9:         if trussMap[(u, w)] < t_UB(e0) then
10:             UES.add((u, w));
11:             UNS.add(w);
12: procedure PROPOSE
13:     for all e ∈ UES do
14:         trussMap[e] ← trussMap[e] + 1;
15:     for all w ∈ UNS do
16:         Send Change to v ∈ N(u);
```

Algorithm 4 Re-initialize stage: spread node *p*

```
Input: adjMap, trussMap
Output: trussMap
 1: procedure PREPARE
 2:     receive adjMap;
 3:     receive Change from o;
 4:     k̄ ← trussMap[(p, o)];
 5:     for all w ∈ N(p) ∩ N(o) do
 6:         if min{trussMap[(p, w)], trussMap[(o, w)]} = k̄
            then
 7:             UNS.add(w);
 8:             if trussMap[(p, w)] = k̄ then
 9:                 UES.add((p, w));
10:             if trussMap[(o, w)] = k̄ then
11:                 UES.add((o, w));
12: procedure PROPOSE
13:     for all e ∈ UES do
14:         trussMap[e] ← trussMap[e] + 1;
15:     for all w ∈ UNS do
16:         Send Change to v ∈ N(p);
```

Analysis. The reinitialization stage ends when no node receives *UES*. The time complexity of the reinitialization stage depends on the diameter of the induced subgraph generated by *UNS*. The induced graph generated by *UNS* is denoted as H, which is a k-truss. The diameter of a connected k-truss with n nodes is not greater than $[\frac{2n-2}{k}]$[8]. As shown in Ref. [9], for a graph $G(V, E)$ and a node set $Q \subset V$, we have $dist_G(G, Q) \leqslant diam(G) \leqslant 2dist_G(G, Q)$, where $dist_G(u, v)$ is the length of the shortest path between u and v in G and $diam(G) = \max_{u,v \in G}\{dist_G(u, v)\}$ is the diameter of graph G. The time complexity of reinitialization stage is $O(diam(H))$. Algorithm 1 will be terminated for two rounds of the execution phase in most cases (one round ensures that all nodes will not update *trussMap*) after resetting the estimated trussness of edges.

Distributed truss maintenance with one-edge deletion. The algorithm for a single edge deletion is simpler than the insertion case. After an edge is deleted, the two endpoints of this edge first update their *adjMap* and remove the nonexistent items from the *trussMap* due

to the change in the graph structure. The execution phase of Algorithm 1 is then performed, and the estimated trussness of edges will converge to the real trussness. In this case, only nodes in *UNS* will participate in the calculation, and only a few rounds will be needed in most cases.

We do not need to label nodes in UNS simliar to the insertion scenario due to the absence of potential edges to determine. Hence, the distributed truss maintenance algorithm for edge deletion is efficient and fast.

5.2 Dynamics of multiple edges

In this section, we study the truss maintenance after the insertion/deletion of multiple edges. The main challenges of maintaining trussness in the case of multiple edges dynamics remain as follows: determining which edges change in trussness and by how much. Algorithms for the case of a single edge dynamic can be used iteratively when multiple edges are inserted/deleted. However, we can further optimize the algorithms when inserting/removing multiple edges for efficient computation.

Insertion of multiple edges. Algorithm 5 shows the procedure of truss maintenance after the insertion of multiple edges. Suppose the set of inserted edges is ΔE. For each edge $e_0 \in \Delta E$, Algorithms 3 and 4 are first executed after e_0 is inserted (lines 1–5). After all edges in ΔE are inserted, the execution phase of Algorithm 1 is executed until termination (line 6).

Deletion of multiple edges. Algorithm 5 shows the procedure of truss maintenance after the deletion of multiple edges. Suppose the set of deleted edges is $\Delta E \subset E$. The *adjMap* and *trussMap* of all nodes are directly updated after deleting all the edges in ΔE. That is, each node deletes the nonexistent neighbors, and then sends the adjacency list to the remaining neighbors. Afterward, each node deletes the item whose key does not exist in the *trussMap* and then sends the *trussMap* of incident edges to the neighbors. Finally, nodes perform the execution phase of Algorithm 1 until termination. In the case of deleting numerous edges, the endpoints of each deleted edge can recalculate the value of $sup(e) + 2$ of the incident edges. An endpoint u updates $trussMap[e] = \min\{sup(e) + 2, trussMap[e]\}$ if e is an incident edge of u. Given a large number of deleted edges and broken triangles, the value of $sup(e) + 2$ for $e \in G \backslash \Delta E$ may even be smaller than $t(e)$. Storing the smaller values in *trussMap* enables the rapid termination of the algorithm.

Algorithm 5 Re-initialize stage of multiple edges dynamic

```
Input: adjMap, trussMap, ΔE
1: for e0 ∈ ΔE do
2:     for all u ∈ e0 do
3:         Algorithm 3;
4: for all spread nodes do
5:     Algorithm 4;
6: execution phase of Algorithm 1;
```

Analysis. The analysis of previous sections indicate that the time complexity of single edge insertion is $O(diam(H))$. Thus the time complexity for multiple edges insertion is $O(diam(G)\cdot|\Delta E|)$, where $diam(G)$ is the diameter of G.

6 Experiment

In this section, we conduct our algorithms on the real-world and synthetic graphs to evaluate the performance. We first introduce the datasets and then evaluate the efficiency of truss decomposition and maintenance.

All the programs are compiled with Java 8 and run on a Linux machine with an Intel Xeon 3.4 GHz CPU and 120 GB RAM. All experiments are run ten times and the average values are reported.

6.1 Datasets

Four static graphs, four temporal graphs, and four synthetic graphs have been adopted. The static and temporal graphs can be downloaded from Stanford Network Analysis Project (SNAP)[†].

For the static graphs, ca-HepPh (CH) is a collaboration network of Arxiv High Energy Physics; com-DBLP (CD) is a DBLP collaboration network and com-YouTube (CY) is YouTube online social network, and both are ground-truth communities; roadNet-CA (RC) is a road network of California.

Temporal graphs are networks where edges have timestamps. An edge (u, v, t) means that node u connects to node v at time t. email-Eu-core (EU) is a network of E-mails between users at a research institution; math-overflow (MO) and Ask-Ubuntu (AU) are networks of comments, questions, and answers on Math Overflow and Ask Ubuntu; superuser (SU) is a temporal network of interactions on the stack exchange web site Superuser[‡]. The statistics of the datasets are shown in Table 2.

Synthetic graphs are generated by the SNAP system by

† http://pan.baidu.com/s/1mgBTFOO

‡ https://superuser.com

Table 2 Statistics of datasets. $|V|$ is the number of nodes, $|E|$ is the number of edges, $|\Delta|$ is the number of triangles of the graph, and deg_{avg} is the average degree.

Name		$\lvert V\rvert$	$\lvert E\rvert$	$\lvert\Delta\rvert$	deg_{avg}
Static graphs	CH	1.2×10^4	1.18×10^5	3.358×10^6	19.73
	CD	3.17×10^5	1.049×10^6	2.224×10^6	6.62
	CY	1.134×10^6	2.987×10^6	3.056×10^6	5.26
	RA	1.965×10^6	2.766×10^6	1.20×10^5	2.81
Temporal graphs	EU	9×10^2	3.32×10^5	1.05×10^5	32.58
	MO	2.4×10^4	5.06×10^5	1.403×10^6	15.13
	AU	1.59×10^5	9.64×10^5	6.80×10^5	5.72
	SU	1.94×10^5	1.443×10^6	1.543×10^6	7.36
Synthetic graphs	ER	1.31×10^5	3.35×10^5	1.7×10^4	5.11
	WS	1.31×10^5	3.93×10^5	4.9×10^4	6.00
	BA	1.31×10^5	7.86×10^5	7×10^3	11.99
	HK	1.31×10^5	7.86×10^5	3.57×10^5	11.99

applying different models. All synthetic graphs have the same node size. Erdos-Renyi (ER)[39], Watts-Strogatz (WS)[40], Barabasi-Albert (BA)[41], and Holme and Kim (HK)[42] are four models for random graph generation.

6.2 Truss decomposition

We selected GPTruss[25] as the baseline to demonstrate the performance of the proposed distributed truss decomposition algorithm, namely DisTruss, in comparative experiments. GPTruss converts the original graph to a line graph (i.e., nodes to edges and edges to nodes) and then computes it as a distributed system with the nodes as independent computational units.

Figure 5 shows the execution time of the two

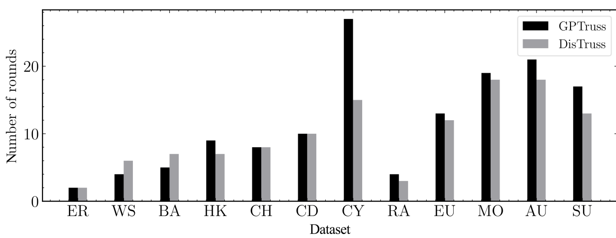


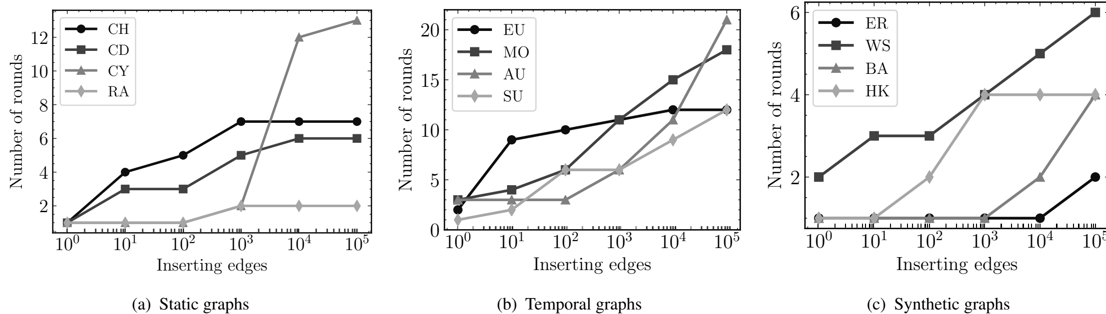
Fig. 5 Proposed method is compared with other methods in truss decomposition.

algorithms on all graphs. The results show that most graphs require only a few rounds to complete the distributed truss decomposition, and only a few dense and large-scale graphs require dozens of rounds, such as CY in static graphs and all temporal graphs. In all graphs of the experiment, the truss decomposition of DisTruss is more efficient compared with GPTruss. In particular, DisTruss is significantly more efficient than GPTruss in some graphs with a large number of edges (i.e., dense graphs). This condition is due to GPTruss, which needs to convert the graph into a line graph, thus generating additional nodes and complicating its computation. In sparse graphs, GPTruss and DisTruss require almost the same number of rounds because the number of nodes in a sparse graph is similar to the number of edges.

6.3 Truss maintenance

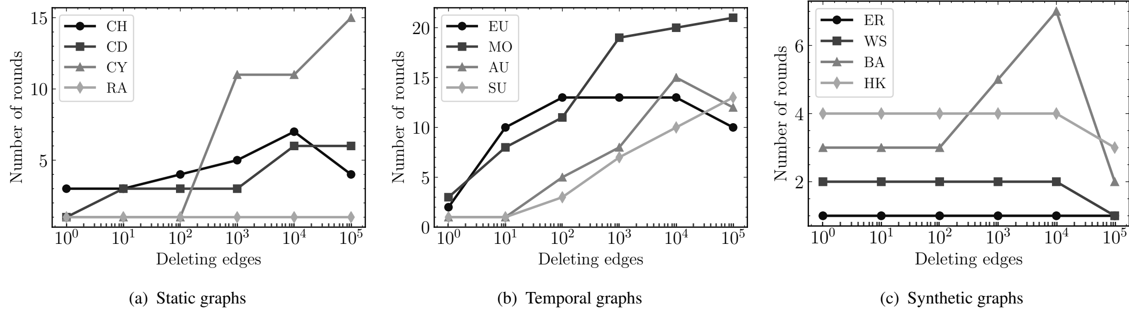
We introduce detailed experiments on distributed truss maintenance to evaluate the efficiency of the proposed distributed truss maintenance algorithms. We randomly chose 10^i edges as the changed edges for insertion and deletion, where $i = 0, 1, 2, 3, 4, 5$ in each graph. We first evaluate the time complexity of the proposed algorithms in all datasets for insertion and deletion of edges. The message complexity of the algorithms on all datasets is then evaluated. Finally, we demonstrate the number of nodes that participates in the computation process in the dynamic setting.

Time efficiency. Figures 6 and 7 shows the execution time of proposed algorithms in each graph after the insertion and deletion of edges. In all graphs, the number of rounds increases as the number of inserted edges rises. The running time is almost the same as the decomposition algorithm when the inserting edges occupy the majority of the original graph. Thus, when the number of inserting edges is too large, truss maintenance has lost its superiority over the decomposition algorithm. However, the maintenance algorithm is still efficient when the number of inserted edges is smaller than the



(a) Static graphs (b) Temporal graphs (c) Synthetic graphs

Fig. 6 Number of rounds after inserting edges in datasets.



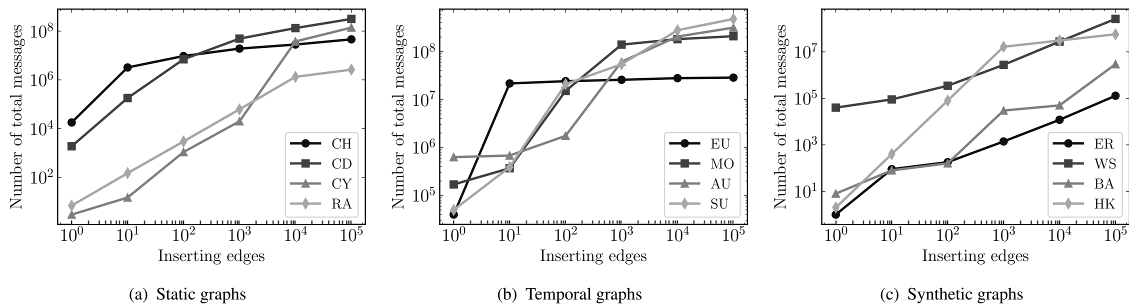
(a) Static graphs (b) Temporal graphs (c) Synthetic graphs

Fig. 7 Number of rounds after deleting edges in datasets.

half edges of the original graphs. For dense graphs, such as ca-HepPh and email-Eu-core, the average degree of their nodes is substantially high. Therefore, the number of active nodes is significantly larger than other graphs after the insertion of edges. Thus, they require the computation of additional rounds after the topology of graphs is changed. Unlike the insertion of edges, the graph becomes sparse during the removal of numerous edges. Thus, only a few rounds are needed to complete the computation.

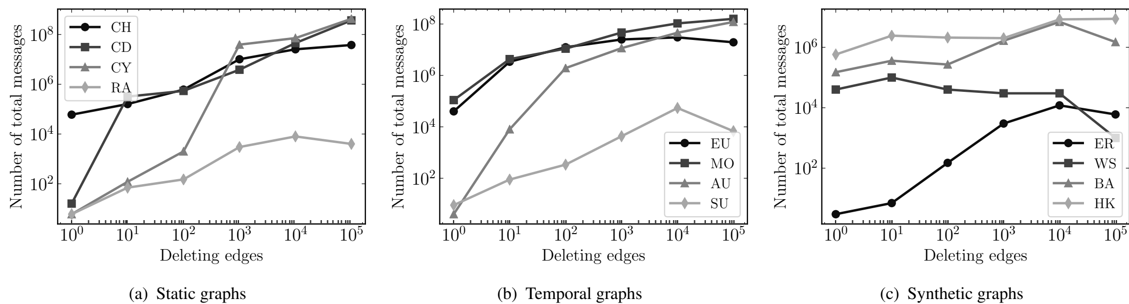
Message efficiency. Figures 8 and 9 show the number of messages of the proposed truss maintenance algorithms in all datasets after the insertion and deletion of edges,respectively. The number of messages reflects the range of nodes affected by the inserted edges. Additional messages are required when the range of affected nodes is large. Furthermore, additional messages must be exchanged under dense graphs. In static graphs, the CH is the densest graph according to the average degree in Table 2. The size of CH is substantially small considering nodes and edges, thus, a large average density leads to a remarkably large number of triangles. The number of edges with the same trussness is also large, which leads to the update of additional messages to maintain the trussness when dynamic edge changes in the same size occur. This condition also shows that the number of messages required for the trussness update is substantial if the trussness change is large or the set of edges to be propagated is wide.

Resource efficiency. Figure 10 shows the activated nodes of the proposed truss maintenance algorithms under different numbers of changed edges. The number of nodes involved in the calculation generally increases



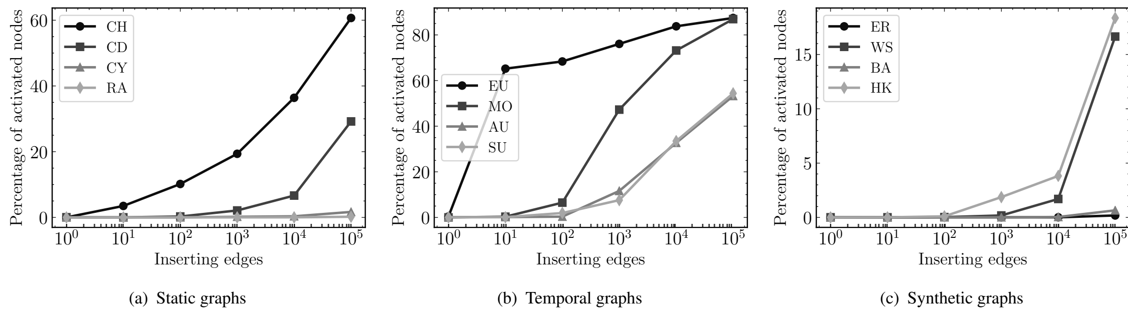
(a) Static graphs (b) Temporal graphs (c) Synthetic graphs

Fig. 8 Number of messages after inserting edges in datasets.



(a) Static graphs (b) Temporal graphs (c) Synthetic graphs

Fig. 9 Number of messages after deleting edges in datasets.



(a) Static graphs (b) Temporal graphs (c) Synthetic graphs

Fig. 10 Percentage of activated nodes after inserting edges in datasets.

as the number of inserted edges rises. Compared with the node size of the entire graph, the dynamic graph case has only a small percentage of active nodes. Therefore, the dynamic maintenance algorithms save more computational resources compared with the static decomposition algorithm. In very dense graphs, such as EU, with an average of 32, the proposed algorithm involves a large number of nodes in the computation.

7 Conclusion

We studied a fundamental graph analysis problem, that is, the computation of k-trusses. Previous approaches focus on static graphs or sequential computational models, but the proposed algorithms can handle multiple dynamically changing edges simultaneously. We propose distributed truss decomposition and maintenance algorithms by designing a synchronized node-centric distributed model. In particular, the proposed algorithms require only the local structure information of the graph and can be efficiently implemented in a distributed environment. Extensive experiments demonstrate the excellent properties of the proposed distributed algorithms, and this new understanding should help improve the decomposition and maintenance of the truss under the impact of other computing platforms. Therefore, future work therefore may consider larger amounts of batch processing while evaluating their efficiency on remarkably large-scale graphs.

Acknowledgment

This work was supported in part by the National Key Research and Development Program of China (No. 2020YFB1005900), in part by National Natural Science Foundation of China (No. 62122042), and in part by Shandong University Multidisciplinary Research and Innovation Team of Young Scholars (No. 2020QNQT017).

References

[1] Q. Luo, D. Yu, Z. Cai, X. Lin, and X. Cheng, Hypercore maintenance in dynamic hypergraphs, in *Proc. 37th IEEE Int. Conf. on Data Engineering*, Chania, Greece, 2021, pp. 2051–2056.

[2] Q. Luo, D. Yu, F. Li, Z. Dou, Z. Cai, J. Yu, and X. Cheng, Distributed core decomposition in probabilistic graphs, in *Proc. 8th Int. Conf. on Computational Data and Social Networks*, Ho Chi Minh City, Vietnam, 2019, pp. 16–32.

[3] D. Yu, N. Wang, Q. Luo, F. Li, J. Yu, X. Cheng, and Z. Cai, Fast core maintenance in dynamic graphs, *IEEE Trans. Comput. Soc. Syst.*, vol. 9, no. 3, pp. 710–723, 2022.

[4] J. Abello, M. G. C. Resende, and S. Sudarsky, Massive quasi-clique detection, in *Proc. 5th Latin American Symp. on LATIN 2002: Theoretical Informatics*, Cancun, Mexico, 2002, pp. 598–612.

[5] S. B. Seidman, Network structure and minimum degree, *Soc. Netw.*, vol. 5, no. 3, pp. 269–287, 1983.

[6] S. B. Seidman and B. L. Foster, A graph-theoretic generalization of the clique concept, *J. Math. Sociol.*, vol. 6, no. 1, pp. 139–154, 1978.

[7] R. J. Mokken, Cliques, clubs and clans, *Qual. Quant.*, vol. 13, no. 2, pp. 161–173, 1979.

[8] J. Cohen, *Trusses: Cohesive Subgraphs for Social Network Analysis.*, Tech. Rep., National Security Agency, Middleburg, VA, USA, 2008.

[9] X. Huang, L. V. S. Lakshmanan, J. X. Yu, and H. Cheng, Approximate closest community search in networks, *Proc. VLDB Endow.*, vol. 9, no. 4, pp. 276–287, 2015.

[10] M. Sozio and A. Gionis, The community-search problem and how to plan a successful cocktail party, in *Proc. 16th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, Washington, DC, USA, 2010, pp. 939–948.

[11] J. Zhang, P. S. Yu, and Y. Lv, Enterprise employee training via project team formation, in *Proc. 10th ACM Int. Conf. on Web Search and Data Mining*, Cambridge, UK, 2017, pp. 3–12.

[12] T. Chakraborty, S. Patranabis, P. Goyal, and A. Mukherjee, On the formation of circles in co-authorship networks, in *Proc. 21th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, Sydney, Australia, 2015, pp. 109–118.

[13] J. Ugander, L. Backstrom, C. Marlow, and J. M. Kleinberg,

Structural diversity in social contagion, *Proc. Natl. Acad. Sci. USA*, vol. 109, no. 16, pp. 5962–5966, 2012.

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

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

[16] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, Pregel: A system for large-scale graph processing, in *Proc. 2010 ACM SIGMOD Int. Conf. on Management of Data*, Indianapolis, IN, USA, 2010, pp. 135–146.

[17] Y. C. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, Distributed graphlab: A framework for machine learning and data mining in the cloud, *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, 2012.

[18] J. Sun, D. Zhou, H. Chen, C. Chang, Z. Chen, W. Li, and L. He, GPSA: A graph processing system with actors. in *Proc. 44th Int. Conf. on Parallel Processing*, Beijing, China, 2015, pp. 709–718.

[19] S. Chu and J. Cheng, Triangle listing in massive networks and its applications, in *Proc. 17th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, San Diego, CA, USA, 2011, pp. 672–680.

[20] A. E. Sariyüce and A. Pinar, Fast hierarchy construction for dense subgraphs, *roceedings VLDB Endowment*, vol. 10, no. 3, pp. 97–108, 2016.

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

[22] Y. Zhang and S. Parthasarathy, Extracting analyzing and visualizing triangle K-core motifs within networks, in *Proc. 2012 IEEE 28th Int. Conf. on Data Engineering*, Arlington, VA, USA, 2012, pp. 1049–1060.

[23] J. Cohen, Graph twiddling in a mapreduce world, *Comput. Sci. Eng.*, vol. 11, no. 4, pp. 29–41, 2009.

[24] 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.

[25] P. L. Chen, C. K. Chou, and M. S. Chen, Distributed algorithms for k-truss decomposition, in *Proc. 2014 IEEE Int. Conf. on Big Data*, Washington, DC, USA, 2014, pp. 471–480.

[26] H. Kabir and K. Madduri, Parallel k-truss decomposition on multicore systems. in *Proc. 2017 IEEE High Performance Extreme Computing Conf.*, Waltham, MA, USA, 2017, pp. 1–7.

[27] H. Kabir and K. Madduri, Shared-memory graph truss decomposition, in *Proc. 2017 IEEE 24th Int. Conf. on High Performance Computing*, Jaipur, India, 2017, pp. 13–22.

[28] Y. Shao, L. Chen, and B. Cui, Efficient cohesive subgraphs detection in parallel, in *Proc. 2014 ACM SIGMOD Int. Conf. on Management of Data*, Snowbird, UT, USA, 2014, pp. 613–624.

[29] S. Smith, X. Liu, N. K. Ahmed, A. S. Tom, F. Petrini, and G. Karypis, Truss decomposition on shared-memory parallel systems, in *Proc. 2017 IEEE High Performance Extreme Computing Conf.*, Waltham, MA, USA, 2017, pp. 1–6.

[30] F. Esfahani, J. Wu, V. Srinivasan, A. Thomo, and K. Wu, Fast truss decomposition in large-scale probabilistic graphs, In *Proc. 22nd Int. Conf. on Extending Database Technology*, Lisbon, Portugal, 2019, pp. 722–725.

[31] Z. Zou, Bitruss decomposition of bipartite graphs, in *Proc. 21st Int. Conf. on Database Systems for Advanced Applications*, Dallas, TX, USA, 2016, pp. 218–233.

[32] X. Huang, W. Lu, and L. V. S. Lakshmanan, Truss decomposition of probabilistic graphs: Semantics and algorithms, in *Proc. 2016 Int. Conf. on Management of Data*, San Francisco, CA, USA, 2016, pp. 77–90.

[33] H. Sun, Y. Zhang, X. Jia, P. Wang, R. Huang, J. Huang, L. He, and Z. Sun, A truss-based approach for densest homogeneous subgraph mining in node-attributed graphs, *Comput. Intell.*, vol. 37, no. 2, pp. 995–1010, 2021.

[34] E. Akbas and P. Zhao, Truss-based community search: A truss-equivalence based indexing approach, *Proc. VLDB Endow.*, vol. 10, no. 11, pp. 1298–1309, 2017.

[35] Y. Zhang and J. X. Yu, Unboundedness and efficiency of truss maintenance in evolving graphs, in *Proc. 2019 Int. Conf. on Management of Data*, Amsterdam, The Netherlands, 2019, pp. 1024–1041.

[36] R. Zhou, C. Liu, J. X. Yu, W. Liang, and Y. Zhang, Efficient truss maintenance in evolving networks, arXiv preprint arXiv: 1402.2807, 2014.

[37] Q. Luo, D. Yu, X. Cheng, Z. Cai, J. Yu, and W. Lv, Batch processing for truss maintenance in large dynamic graphs, *IEEE Trans. Comput. Soc. Syst.*, vol. 7, no. 6, pp. 1435–1446, 2020.

[38] A. Montresor, F. De Pellegrini, and D. Miorandi, Distributed k-core decomposition, *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 2, pp. 288–300, 2013.

[39] T. G. Kolda, A. Pinar, T. D. Plantenga, and C. Seshadhri, A scalable generative graph model with community structure, *SIAM J. Sci. Comput.*, vol. 36, no. 5, pp. C424–C452, 2014.

[40] D. J. Watts and S. H. Strogatz, Collective dynamics of ‘small-world’ networks, *Nature*, vol. 393, no. 6684, pp. 440–442, 1998.

[41] A. L. Barabási and R. Albert, Emergence of scaling in random networks, *Science*, vol. 286, no. 5439, pp. 509–512, 1999.

[42] P. Holme and B. J. Kim, Growing scale-free networks with tunable clustering, *Phys. Rev. E*, vol. 65, no. 2, p. 026107, 2002.

Ziwei Mo is currently pursuing the PhD degree in School of Computer Science and Technology, Shandong University, China. Her research interests include distributed computing and data mining.

Qi Luo received the BS and MS degrees in computer science from Northeastern University, China in 2015, and Shandong University, China, in 2018, respectively. He is currently pursuing the PhD degree with the School of Computer Science and Technology, Shandong University. His research interests include graph analysis and distributed computing.

Dongxiao Yu received the BS degree in 2006 from Shandong University, China and the PhD degree in 2014 from the University of Hong Kong, China. He became an associate professor in the School of Computer Science and Technology, Huazhong University of Science and Technology, in 2016. 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.

Hao Sheng received the BS and PhD degrees from Beihang University, Beijing, China, in 2003 and 2009, respectively. He is currently an associate professor with the School of Computer Science and Engineering, Beihang University. He is working on computer vision, pattern recognition, and machine learning.

Jiguo Yu received the PhD degree from Shandong University, China, in 2004. He became a full professor with the School of Computer Science, Qufu Normal University, China in 2007. He is currently a full professor with the Qilu University of Technology (Shandong Academy of Sciences), China. His main research interests include privacy-aware computing, wireless networking, distributed algorithms, blockchain, and graph theory.

Xiuzhen Cheng received the MS and PhD degrees in computer science from the University of Minnesota Twin Cities, MN, USA in 2000 and 2002, respectively. She is a professor in the School of Computer Science and Technology, Shandong University, China. She has published more than 170 peer-reviewed papers. Her current research interests include cyber physical systems, wireless and mobile computing, sensor networking, wireless and mobile security, and algorithm design and analysis. She has served on the editorial boards of several technical journals and the technical program committees of various professional conferences/workshops. She also has chaired several international conferences. She worked as a professor with the Department of Computer Science, the George Washington University, Washington, DC, USA, from 2013 to 2017. She worked as a program director for the US National Science Foundation (NSF) from April to October in 2006 (full time), and from April 2008 to May 2010 (part time). She received the NSF CAREER Award in 2004. She is a fellow of IEEE and a member of ACM.