Efficient Star-based Truss Maintenance on Dynamic Graphs

ZITAN SUN, Hong Kong Baptist University, China
XIN HUANG, Hong Kong Baptist University, China
QING LIU, Zhejiang University, China
JIANLIANG XU, Hong Kong Baptist University, China

K-truss is a useful notion of dense subgraphs, which can represent cohesive parts of a graph in a hierarchical way. In practice, in order to enable various truss-based applications to answer queries faster, the edge trussnesses are computed in advance. However, real-world graphs may not always be static and often have edges inserted or removed, leading to costly truss maintenance of recomputing all edge trussnesses. In this paper, we focus on dynamic graphs with *star insertions/deletions*, where a star insertion can represent a newly joined user with friend connections in social networks or a recently published paper with cited references in citation networks. To tackle such star-based truss maintenance, we propose a new structure of AffBall based on the local structure of an inserted/deleted star motif. With AffBall, we make use of the correlation of inserted edges to compute the trussnesses of the inner edges surrounding the star. Then, we analyze the onion layer of k-truss and conduct truss maintenance for the edges beyond the star, which can be efficiently achieved with a time complexity related to the number of the edges that change the onion layer. Moreover, we extend star-based truss maintenance to handle general updates and single-edge insertions/deletions. Extensive experiments on real-world dynamic graphs verify the effectiveness and efficiency of proposed algorithms against state-of-the-art truss maintenance algorithms.

CCS Concepts: • **Computing methodologies**; • **Theory of computation → Data structures and algorithms for data management**; • **Human-centered computing** → Collaborative and social computing; • **Networks** → Network algorithms;

Additional Key Words and Phrases: dynamic graphs, k-truss, incremental algorithms, social networks

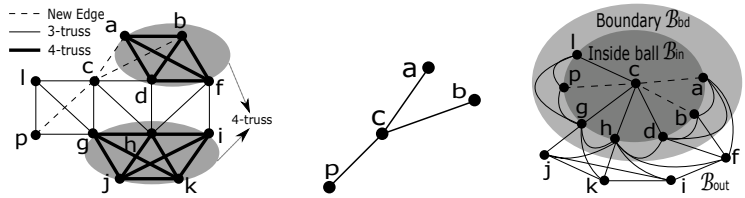
ACM Reference Format:
Zitan Sun, Xin Huang, Qing liu, and Jianliang Xu. 2023. Efficient Star-based Truss Maintenance on Dynamic Graphs. *Proc. ACM Manag. Data* 1, 2, Article 133 (June 2023), 26 pages. https://doi.org/10.1145/3589278

1 INTRODUCTION

Graph is a widely used model to depict entities and their connections in various real-life applications, such as social networks, biological networks, transportation networks, citation networks, and so on [7]. In many graph analytics tasks, dense subgraph identification plays a central role. As a popular notion of dense subgraphs, k-truss is the largest subgraph satisfying that every edge is contained in at least k-2 triangles within this subgraph [7, 12, 37]. Different from those NP-hard computations of dense subgraphs such as cliques [33], quasi-cliques [27], n-clans [26], and n-club [26], k-truss discovery enjoys a polynomial time computation, as well as several nice structural properties, e.g., $(k-1)$-connectivity, a bounded diameter, and a hierarchical structure [7, 12]. The k-truss has many

Authors' addresses: Zitan Sun, zitansun@comp.hkbu.edu.hk, Hong Kong Baptist University, China; Xin Huang, xinhuang@comp.hkbu.edu.hk, Hong Kong Baptist University, China; Qing liu, qingliucs@zju.edu.cn, Zhejiang University, China; Jianliang Xu, xujl@comp.hkbu.edu.hk, Hong Kong Baptist University, China.

2836-6573/2023/6-ART133 $15.00
https://doi.org/10.1145/3589278



(a) Graph G (b) Star motif S_c (c) AffBall $\mathcal{B}$

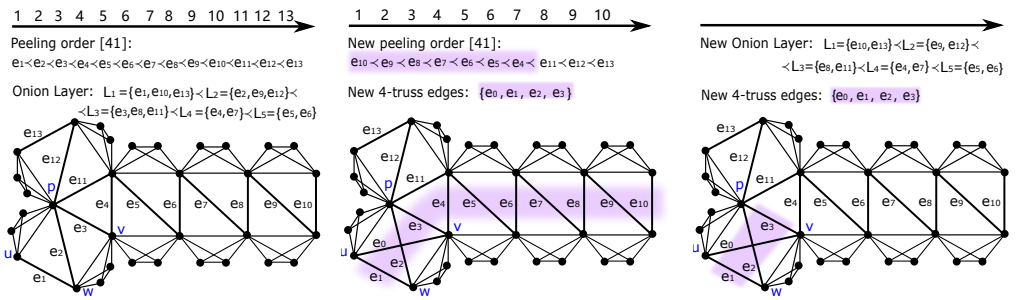
Fig. 1. An example of dynamic graph G (a) with the insertion of a star motif $S_c = \{(c, a), (c, b), (c, p)\}$ (b). The AffBall structure $\mathcal{B} = \mathcal{B}_{in} \cup \mathcal{B}_{bd}$ is shown in Fig. 1(c).

applications, e.g., community search [12, 14, 16, 21, 40] , clique discovery [37], pivotal relationship identification [4, 35, 46], and complex network visualization [43].

In real life, connections evolve. Many real graphs are not static but undergo frequent updates, known as *dynamic graphs*, where nodes and edges may be inserted or deleted. Graph updates can happen anywhere in the graph. However, from a micro node's view point, we can treat all the updates as a series of *star-motif insertions/deletions* (or *star insertions/deletions* for short), where a star motif is represented by a set of edges incident to a common node. For example, Fig. 1(a) and 1(b) exemplify the insertion of a star motif S_c, which consists of three new edges: $S_c = \{(c, a), (c, b), (c, p)\}$ and a common center node c. In fact, such star insertions/deletions exist naturally in dynamic graphs. For instance, in a citation network, the nodes are research papers and the edges are the citation relationships between two papers. A star insertion corresponds to the insertion of a newly published paper, along with its citation relationships. Similarly, a star deletion means the retraction of a paper and its citation records. As another example, in a social network, two user accounts have an edge that indicates their friend relationship. A newly registered account may create a new friend circle by adding a dozen friends, which can be regarded as a star insertion. Likewise, a star deletion corresponds to the deregistration of a user account and also its friend connections.

Truss decomposition is to identify all non-empty k-trusses for different possible values of k in a graph [37]. However, it brings significant challenges to compute all k-trusses in dynamic graphs, which needs to apply truss decomposition algorithms on every updated graph. This is inefficient for dynamic graphs with frequent updates. To tackle it efficiently, existing studies [12, 41] compute the trussnesses of all edges on the original graph into the index, and then develop a new *truss maintenance* algorithm to *partially* update the trussnesses index w.r.t. the graph update. However, most existing truss maintenance algorithms [12, 25, 41] neglect the local correlation of graph updates and treat each update as a single-edge insertion/deletion independently, which limits the update efficiency.

In this paper, we study the problem of truss maintenance over dynamic graphs with star insertions/deletions. We first focus on the case of star insertions, and then extend it to star deletions and general updates. Specifically, the star insertion problem needs to compute the trussnesses of all edges in the new graph $G + S_c$, where G is the original graph and S_c is the graph update of an inserted star motif. Consider the example of the dynamic graph G with a star insertion $S_c = \{(c, a), (c, b), (c, d)\}$ in Fig. 1, the edges $(c, d), (c, g), (c, l), (l, g), (l, p), (p, g)$ change their trussnesses from 3 to 4. Moreover, the new edges $(c, a), (c, b), (c, d)$ have the trussnesses of 4. All these affected edges are connected in the graph. Thus, we may observe that the trussness update has a local property for star insertions. To efficiently address the problem, our key idea is to partition the new graph into two parts: a local subgraph surrounding the star motif S_c and the remaining global



(a) Original graph G (b) Peeling order [41] (c) Our method based onion layers

Fig. 2. A motivating example of truss maintenance using different data structures of edge orders: a peeling order [41] in Fig. 2(b) and our onion layer based order in Fig. 2(c). The graph G in Fig. 2(a) is inserted with a new edge $e_0 = (u, v)$. The regions in purple show that the number of edge orders needs to change by [41] and our star-based method, which are 11 and 4, respectively.

graph. Then, we develop two novel techniques of AffBall and *onion layers* to estimate and refine the trussnesses for the edges in these two parts, respectively.

More specifically, we first propose an AffBall-based technique for trussness estimation in the local neighborhood of the star motif S_c. We define a new subgraph concept of AffBall, whose edges are in the 1-hop neighborhood of S_c. The inside edges of AffBall contain all affected edges whose trussnesses may increase by more than one. Thus, we develop a variant truss decomposition algorithm on this local subgraph AffBall, which computes an estimated trussness for each edge as a lower bound of exact trussness. The difference between the lower bound and the actual trussness is no more than one.

Second, we propose an onion layer based method for trussness refinement in the affected region of the whole graph. After the first phase of trussness estimation in AffBall, all edges' trussnesses are close to exact trussnesses, of which the difference is no more than one. To achieve fast trussness refinement, we present two new concepts of onion layers and onion supports to keep the auxiliary structure information. For a given edge e, the corresponding onion layer and the onion support keep the round number of edge removals and the number of triangles containing e, respectively, when the edge e is deleted from the k-truss peeling process. The number of onion layers helps distinguish the dependent order of edges, similar to the peeling order proposed in [41]. However, the maintenance of our onion layers can be easier and faster than the maintenance of the peeling order [41]. Fig. 2 shows an example of graph G inserted with a new edge $e_0 = (u, v)$. Assume that the peeling order [41] of all 13 edges in 3-truss are numbered from 1 to 13 in Fig. 2(a). A smaller edge number indicates an earlier removal in truss decomposition, e.g., $e_{10} \prec e_{13}$ represents e_{10} should be removed earlier than e_{13}. In the auxiliary structure of onion layers, $L_1 = \{e_1, e_{10}, e_{13}\} \prec L_2 = \{e_2, e_9, e_{12}\}$ indicates that the edges e_1, e_{10}, e_{13} can be removed in the same round and should be removed earlier than $e_2 \in L_2$. After the insertion of e_0, [41] needs to change the order for 11 candidate edges (e_0-e_3 in 4-truss and e_4-e_{10} in 3-truss) as shown in the purple region of Fig. 2(b). On the other hand, our approach only needs to modify the onion layers for 4 candidate edges (e_0-e_3) in the purple region of Fig. 2(c), which is much less than [41]. Leveraging the onion layer based updating rules, it saves unnecessary computations to check the trussnesses of the edges that have unchanged onion layers. Finally, we show that the time complexity of our truss maintenance algorithms is bounded by the number of affected edges with changed onion layers.

In addition, to handle general updates in dynamic graphs, we extend star-based truss maintenance techniques to handle multiple overlapping star insertions. The key idea is to divide the graph update into pairwise disjoint star motifs and insert them in batches. A greedy method is proposed

to generate such disjoint star motifs. To summarize, we make the following contributions in this paper.

- We develop a notion of star motif and formulate the problem of truss maintenance with star insertions. We show the generalization of our problem to handle general updates and single-edge insertions/deletions on dynamic graphs. (Section 3)
- We propose a local structure of AffBall based on an inserted star. We show that all edges whose trussnesses may change by more than one fall into the region of AffBall. We develop a star-based truss maintenance framework to first compute the lower bounds of trussnesses within AffBall and then refine the lower bounds to exact trussnesses across the whole graph. (Section 4)
- We propose two concepts of onion layers and onion supports. Based on them, we develop several useful pruning strategies to reduce candidate edges for recomputing trussnesses. Equipped with onion layer based techniques, our star-based framework can accomplish truss maintenance with a time complexity bounded by the number of edges that change onion layers. Moreover, we extend the star insertion techniques to handle star deletions. In addition, we analyze and compare the complexities of all algorithms against state-of-the-art methods. (Section 5)
- We extend the star-based techniques to handle general graph updates, which uses a greedy method to partition the graph update into multiple batches of pairwise disjoint star motifs and then insert them in batches. (Section 6)
- We conduct extension experiments on nine real-world graphs to demonstrate that our star-based approach runs several orders of magnitude faster than the state-of-the-art approach [41]. Moreover, our star-based approaches are validated to efficiently handle both general graph updates and single-edge insertions/deletions. (Section 7)

We discuss related work in Section 2 and conclude the paper in Section 8.

2 RELATED WORK

Truss mining. Given an undirected graph G, a k-truss is defined as the maximal subgraph $H \subseteq G$ such that each edge of H is contained in at least $k-2$ triangles. Due to its strong structural cohesiveness and high computational efficiency, k-truss is widely used in real applications, such as community search [12, 14, 16, 21, 40] and pivotal relationship identification [4, 35, 46]. In addition, k-truss has been extended to other types of graphs, such as geo-social graphs [5, 34], uncertain graphs [9, 10, 15, 36, 47], bipartite graphs [20], attribute graphs [13, 24, 39], weighted graphs [45], public-private graphs [8], directed graphs [22], multilayer graphs [11], signed graphs [38, 44], and simplicial complexes [28]. For an edge e, the trussness of e is defined as the maximum value of k such that e is contained in k-truss. Given an undirected graph G, the goal of truss decomposition is to compute the trussness for every edge [3, 6, 17, 32, 37]. In this work, we focus on the truss maintenance problem, i.e., to maintain trussnesses when edges are inserted/deleted into/from the graph. A straightforward way for truss maintenance is to perform truss decomposition from scratch, which is inefficient. Therefore, it is necessary to develop efficient algorithms for truss maintenance.

Dense subgraph maintenance. In the literature, many works explore the problem of dense subgraph maintenance, including core maintenance and truss maintenance. Specifically, the core maintenance is to maintain k-cores when the graphs are updated. Typical algorithms include incremental algorithms [19, 30, 31], order-based algorithm [42], and H-index-based algorithm [23]. For the truss maintenance, [12] first studies the problem and proposes algorithms to address single-edge insertions/deletions, which traverse the edges with the same trussness. [25] splits the inserted subgraph into unrelated edges, which ensure that the change of trussness is no more than one.

Moreover, [41] extends the order-based core maintenance algorithm to handle truss maintenance. It is worth mentioning that all existing algorithms for dense subgraph maintenance do not consider the relationship of inserted edges. In our work, we explore the relationship of inserted edges and employ the star motif to design novel truss maintenance algorithms.

3 PRELIMINARIES

In this section, we present definitions and give our problem formulation. Given a graph $G = (V, E)$, a triangle $\triangle_{uvw}$ is defined as a circle of three vertices: u, v, w. For a subgraph $H \subseteq G$ and an edge $(u, v) \in E(H)$, the support $\sup_H((u, v))$ is the number of triangles containing (u, v) in H, i.e., $\sup_H((u, v)) = |N_H(u) \cap N_H(v)|$, where the neighbor set $N_H(u) = \{w|(u, w) \in E(H)\}$. Throughout this paper, we use the notation $'$ for dynamic graphs to represent various concepts in the new graph after the update, e.g., the new graph G', and the new support $\sup'_H(u, v)$, and so on.

Definition 1 (K-Truss [7]). *Given a graph $G = (V, E)$, the subgraph $H \subseteq G$ is the k-truss* if and only if *H is the largest subgraph such that every edge $e \in E(H)$ has the support* $\sup_H(e) \geq k - 2$. *We also use T_k to represent the k-truss for a given integer $k \geq 2$.*

Definition 2 (Trussness). *Given an edge e in graph G, the trussness of e is the largest integer k such that a non-empty k-truss $H \subseteq G$ contains e, denoted as $\tau(e) = \max\{k \in \mathbb{Z}^+ : \exists H \subseteq G, e \in E(H), H \text{ is a } k\text{-truss}\}$.*

Based on the edge trussness, we also define the k-class edges.

Definition 3 (K-Class Edges). *The k-class edges E_k are defined as the set of edges that appear in the k-truss but not in any $(k + 1)$-truss, i.e., $E_k = \{e \in E : \tau(e) = k\}$.*

Consider the example graph G shown in Fig. 1(a) without three dashed edges (a, c), (b, c) and (c, p). The edge (a, b) involves two triangles, $\triangle_{abd}$ and $\triangle_{abf}$, thus the support is $\sup_G((a, b)) = 2$ in G. The subgraph in the gray region is the 4-truss, where every edge has the support of no less than 2. We have the trussness $\tau((a, b)) = 4$. Moreover, for another edge (c, d), the support is $\sup_G((c, d)) = 1$ and the trussness is $\tau((c, d)) = 3$, as the 3-class edge $(c, d) \in E_3$ cannot appear in any 4-truss.

Definition 4 (star motif). *Given an edge set S, if there exists a node c such that every edge $e \in S$ contains c, we call S a star motif, denoted by S_c, where c is the center node.*

Based on the definitions of trussness and star motif, we can formulate the problem of truss maintenance with star insertions as follows.

Problem 1 (Truss Maintenance with Star Insertions). *Given a dynamic graph $G = (V, E)$, the trussness $\tau(e)$ for all edges $e \in E$, and a star motif S to be inserted into G where $S \cap E = \emptyset$, the problem is to compute the new trussness $\tau'(e)$ for each $e \in E'$ in the new graph $G' = (V, E')$, where $E' = E \cup S$.*

Example 1. *Fig. 1(a) shows an example of our problem. The graph G is inserted with a star motif $S = \{(a, c), (b, c), (c, p)\}$, which has three edges and a center node c as shown in Fig. 1(b). The original trussnesses $\tau((c, d)) = 3$ and $\tau((a, b)) = 4$. After the star insertion of S, the new trussnesses are $\tau'((c, d)) = 4$ and $\tau'((a, b)) = 4$ in graph $G'(V, E \cup S)$, as shown in Fig. 3(b).*

Note that we have formulated the problem of truss maintenance with the case of *star insertion* over dynamic graphs. Most techniques introduced in Sections 4 and 5 focus on this star insertion case. Nevertheless, our techniques developed for star insertion can be easily extended to another relevant case of *star deletion*, which will be discussed in Section 5.4. *Single-edge insertion/deletion* is

Algorithm 1 AffBall-based Framework for Star Insertions

Input: Graph $G = (V, E)$, the trussness $\tau(e)$ for all edges $e \in E$, an inserted star motif S_c;
Output: Updated trussness $\tau'(e)$ for all edges $e \in E' = E \cup S_c$;
1: Insert star S_c into G to form an updated graph $G' = (V, E')$ where $E' = E \cup S_c$;
2: Obtain an AffBall $\mathcal{B} = \mathcal{B}_{in} \cup \mathcal{B}_{bd}$ by Def. 5;
3: Compute lower bounds $lowB(e)$ for $e \in \mathcal{B}_{in}$ by applying local truss decomposition on $\mathcal{B}$ in Algorithm 2;
4: Temporarily set the updated trussness as $\tau'(e) \leftarrow lowB(e)$ for $e \in \mathcal{B}_{in}$; // $\tau'(e)$ may further increase by no more than 1.
5: Find the seed candidates $Seed \subseteq \mathcal{B}_{bd}$ for trussness refinement by Algorithm 3;
6: Start from $Seed$ to find more candidate edges to update all $\tau'(e)$ for $e \in E'$, e.g., using an existing truss maintenance [12] or Algorithm 5;
7: **return** Updated $\tau'(e)$ for all edges $e \in E'$;

a special case of star-motif insertion/deletion. Thus, our proposed star-based techniques can also handle the update of single-edge insertions/deletions. For *single-node insertion/deletion*, it can be treated as a star-motif insertion/deletion preceded/followed by the insertion/deletion of an isolated node. Moreover, based on the truss maintenance solutions for star-motif insertions and deletions, we further study a *general truss maintenance* problem to handle arbitrary insertions, which will be introduced in Section 6.

4 AFFBALL

In this section, we introduce a new subgraph structure of AffBall based on an inserted star. Then, we analyze the structural properties of AffBall and propose the rules for updating trussnesses. Finally, we develop an AffBall-based framework for truss maintenance with a star-motif insertion, which first coarsely estimates lower bounds of trussnesses within AffBall region and then finely computes the trussnesses of all edges.

4.1 AffBall Structure

Motivation. According to [12], when a new edge is inserted into a graph, all other edges have their trussnesses changed no more than one. Thanks to this updating rule, it saves lots of computations to avoid checking a large number of candidate edges whose trussnesses are larger than or less than the trussness of the inserted edge. However, when it inserts two or more new edges into a graph, the above rule may not hold any more. To achieve similar updating rules, we develop a new structure AffBall surrounding an inserted star motif S_c to include all edges whose trussnesses may change by more than one. Thus, the outside of AffBall are the edges whose trussnesses may change by no more than one. In the following, we give a formal definition of AffBall.

4.2 AffBall-based Framework for Star Insertions

Definition 5 (AffBall $\mathcal{B}$). *Given a graph $G = (V, E)$ inserted with a star motif S_c where the new edges $E' = E \cup S_c$, an* AffBall *structure consists of inside and boundary edges, denoted as $\mathcal{B} = \mathcal{B}_{in} \cup \mathcal{B}_{bd}$, where the inside edges $\mathcal{B}_{in} = \{(c, v) \in E' | v \in N'(c)\}$ and the boundary $\mathcal{B}_{bd} = \{(u, v) \in E' | u, v \in N'(c)\}$. The remaining edges outside of $\mathcal{B}$ are denoted as $\mathcal{B}_{out} = E' \setminus \mathcal{B}$.*

Consider a graph G inserted with a star motif S_c in Fig. 1. As it inserts 3 new edges $S_c = \{(a, c), (b, c), (c, p)\}$, the AffBall $\mathcal{B}$ has the inside edges as $\mathcal{B}_{in} = \{(c, a), (c, b), (c, d), (c, h), (c, g), (c, p), (c, l)\}$ and the boundary $\mathcal{B}_{bd} = \{(a, b), (a, d), (b, d), (d, h), (h, g), (g, l), (g, p), (l, p)\}$, where every edge

has two endpoints of the center node c's neighbors. Thus, all other edges are outside of AffBall $\mathcal{B}$. Note that if the inserted star motif S has only one edge $S = \{(u, v)\}$, the AffBall $\mathcal{B}$ is degenerated into $\mathcal{B}_{in} = \{(u, v)\}$ and $\mathcal{B}_{bd} = \{(u, w)|w \in N'(u) \cap N'(v)\} \cup \{(v, w)|w \in N'(u) \cap N'(v)\}$.

4.3 AffBall Properties and Updating Rules

Next, we analyze AffBall properties and give the trussness updating rules. An AffBall contains all new edges and also the old edges that are added with more than one new triangles, leading to that they may change their trussnesses by more than one. The inside edge $e \in \mathcal{B}_{in}$ cannot form any triangle with the outside edges $\hat{e} \in \mathcal{B}_{out}$, which means that edges in $\mathcal{B}_{in}$ and edges outside of AffBall are totally separated by the boundary edges $\mathcal{B}_{bd}$. The boundary of AffBall is used as a bridge between $\mathcal{B}_{in}$ and $\mathcal{B}_{out}$ to propagate the change information of supports and trussnesses. The edges in $\mathcal{B}_{bd}$ can form at most one triangle with the edges in $\mathcal{B}_{in}$, meaning that edges on the AffBall boundary can increase their trussnesses by no more than one.

Lemma 1. *Given a graph G inserted with a star motif S_c, the* AffBall *boundary edge $e \in \mathcal{B}_{bd}$ increases the trussness at most by one, i.e., $\forall e \in \mathcal{B}_{bd}$, $\tau'(e) - \tau(e) \leq 1$.*

Proof. Since we insert edges into the graph G, all trussnesses can only increase or remain the same, i.e., $\forall e \in E$, $\tau'(e) \geq \tau(e)$. Next we suppose there exists an edge $e \in \mathcal{B}_{bd}$, $\tau'(e) - \tau(e) \geq 2$. The new trussness $\tau'(e) \geq \tau(e) + 2$ means the $sup_{H'}(e) \geq \tau(e)$, where H' is the $\tau'(e)$-truss after insertion. Since e can form at most one triangle with the edges in $\mathcal{B}_{in}$, which contains all edges in S_c, if we remove S_c again, $sup_H(e) = sup_{H'}(e) - 1 \geq \tau(e) - 1$, where H is the $\tau(e)$-truss after removal. This means $e \in (\tau(e) + 1)$-truss in G, leading to a contradiction. □

Lemma 2. *Given a graph G inserted with a star motif S_c, the* AffBall *outside edge $e \in \mathcal{B}_{out} = E' \setminus \mathcal{B}$ can increase the trussness at most by 1, i.e., $\forall e \in \mathcal{B}_{out}$, $\tau'(e) - \tau(e) \leq 1$.*

Proof. Since edges in $\mathcal{B}_{in}$ cannot form any triangle with the edges not in AffBall, the trussness change of edges in $\mathcal{B}_{in}$ will not affect the edges not in AffBall. According to Lemma 1, edges in $\mathcal{B}_{bd}$ can increase their trussnesses at most by one. Affected by these boundary edges, edges not in AffBall also can increase their trussnesses at most by one. □

For an inside edge $e \in \mathcal{B}_{in}$, we estimate an approximate trussness $lowB(e)$, which is a lower bound close to the real trussness $\tau'(e)$.

Lemma 3. *For an* AffBall *$\mathcal{B} \subseteq G'$ and each edge $e \in \mathcal{B}_{in}$, we derive a lower bound of updated trussness $\tau'(e)$ as $lowB(e) = \max\{k \in \mathcal{Z}^+ | \exists H \subseteq \mathcal{B}$ such that $\forall e_{in} \in E(H) \cap \mathcal{B}_{in}, \sup_H(e_{in}) \geq k-2$ and $\forall e_{bd} \in E(H) \cap \mathcal{B}_{bd}, \tau(e_{bd}) \geq k\}$. Thus, we have the gap between lower bound $lowB(e)$ and updated trussness $\tau'(e)$ no more than 1, i.e., $\tau'(e) - lowB(e) \leq 1$ holds for $e \in \mathcal{B}_{in}$.*

Proof. First, after a local truss decomposition on $\mathcal{B}$, each edge can get a determined trussness $lowB(e)$. Then, consider swapping the inside and outside of the AffBall, since the edges in $\mathcal{B}_{bd}$ increase their trussnesses at most by 1, according to Lemma 2, edges outside the AffBall can increase their trussnesses at most by 1. Therefore, $\tau'(e) - lowB(e) \leq 1$, $\forall e \in \mathcal{B}_{in}$. □

Leveraging Lemma 3, we can apply the local truss decomposition on subgraph $\mathcal{B}$ that keeps trussnesses unchanged for edges in $\mathcal{B}_{bd}$ and recomputes trussnesses for edges e in $\mathcal{B}_{in}$ as the low bound of $lowB(e)$. Note that all these temporary trussness are close to the exact trussnesses and will be further refined to exact ones. Consider the example of graph G inserted with S_c in Fig. 1. Using a local truss decomposition on $\mathcal{B}$, we can get the lower bounds $lowB((a, c)) = lowB((b, c)) = lowB((c, d)) = 4$ and $lowB((c, p)) = 3$, which are very close to the real trussnesses of 4. Leveraging AffBall in Def. 5, we can replace the graph update source from an inserted star motif S_c to the

Algorithm 2 Compute Lower Bounds in AffBall

Input: AffBall $\mathcal{B}$ for an inserted star motif S_c;
Output: Estimated trussness $lowB(e)$ for $e \in \mathcal{B}_{in}$;
1: Calculate the support $\text{sup}_{\mathcal{B}}(e)$ in AffBall $\mathcal{B}$ for edge $e \in \mathcal{B}_{in}$;
2: Initialize $k = 2$, an empty queue $Q = \emptyset$;
3: **while** $\mathcal{B}_{in} \neq \emptyset$ **or** $Q \neq \emptyset$ **do**
4: **for all** $(c, v) \in \mathcal{B}_{in}$ with $\text{sup}_{\mathcal{B}}((c, v)) \leq k - 2$ **do**
5: $\mathcal{B}_{in} \leftarrow \mathcal{B}_{in} \setminus \{(c, v)\}$, $Q \leftarrow Q \cup \{v\}$;
6: **if** $Q = \emptyset$ **then**
7: **for all** $(v, w) \in \mathcal{B}_{bd}$, $\tau((v, w)) \leq k - 2$ **do**
8: Remove edge (v, w) from $\mathcal{B}$;
9: Decrease $\text{sup}_{\mathcal{B}}((c, v))$ and $\text{sup}_{\mathcal{B}}((c, w))$ by one;
10: **if** $\text{sup}_{\mathcal{B}}((c, v)) \leq k - 2$ **then** $Q \leftarrow Q \cup \{v\}$, $\mathcal{B}_{in} \leftarrow \mathcal{B}_{in} \setminus \{(c, v)\}$;
11: **if** $\text{sup}_{\mathcal{B}}((c, w)) \leq k - 2$ **then** $Q \leftarrow Q \cup \{w\}$, $\mathcal{B}_{in} \leftarrow \mathcal{B}_{in} \setminus \{(c, w)\}$;
12: **if** $Q = \emptyset$ **then** $k \leftarrow k + 1$;
13: **while** $Q \neq \emptyset$ **do**
14: $Q \leftarrow Q \setminus \{v\}$;
15: Assign a lower bound $lowB((c, v)) = \max\{\tau((c, v)), k\}$;
16: **for all** $(v, w) \in \mathcal{B}_{bd}$ **do**
17: Remove (v, w) and update $\text{sup}_{\mathcal{B}}$, Q, $\mathcal{B}_{in}$ as lines 9-11;
18: **return** $lowB(e)$ for $e \in \mathcal{B}_{in}$;

edges $\mathcal{B}_{bd}$. Thus, the former S_c may increase trussnesses by more than one, but the latter $\mathcal{B}_{bd}$ can increase trussnesses by up to one w.r.t. $lowB(e)$ for $e \in \mathcal{B}_{in}$ and $\tau(e)$ for $e \in \mathcal{B}_{bd} \cup \mathcal{B}_{out}$, which can be easily handled even by existing truss maintenance techniques [12].

We present the AffBall-based framework for truss maintenance with a star insertion in Algorithm 1. Here is an overview of the AffBall-based framework, which mainly has two phases. The first phase is to compute the lower bounds for AffBall-inside edges $\mathcal{B}_{in}$, which coarsely estimate the trussnesses (lines 1-4), denoted as Phase I. The second phase is truss maintenance, which finely computes the exact trussnesses for all edges (lines 5-6), denoted as Phase II.

Next, we introduce the details of Algorithm 1. Specifically, the algorithm updates the graph G to G' by inserting S_c (line 1) and identifies an AffBall $\mathcal{B}$ by Definition 5 (line 2). Then, it applies a local decomposition to compute a lower bound $lowB(e)$ for $e \in \mathcal{B}_{in}$ and assigns as the temporary trussness (lines 3-4). The procedure details are outlined in Algorithm 2. Given the original trussness $\tau(e)$ and the temporary trussness $\tau'(e)$, the edges $e \in \mathcal{B}_{in}$ and $e \in \mathcal{B}_{out}$ all meet the requirement of k-truss, and only the edges $e \in \mathcal{B}_{bd}$ may violate the k-truss requirement. The edges in $\mathcal{B}_{bd}$ serve as the seed candidate to find all edges that may increase trussnesses by 1 (line 2). The procedure is shown in Algorithm 3. Finally, it starts from *Seed* to find more candidate edges and refine their trussnesses to exact ones, which can use the existing truss maintenance algorithms [12] (line 6).

Procedure of computing lower bounds in $\mathcal{B}_{in}$. Algorithm 2 shows the details of local truss decomposition. We first calculate the support $\text{sup}_{\mathcal{B}}(e)$ in the subgraph $\mathcal{B}$ for edge $e \in \mathcal{B}_{in}$ (line 1). Then, we start from $k = 2$ and peel edges in $\mathcal{B}_{in}$ (lines 4-5) or edges in $\mathcal{B}_{bd}$ (lines 6-11). The edge removal may further cause other edges removed for violating k-truss requirements (lines 13-17). We set the estimated trussness $lowB$ to the maximum value between the old trussness and the current k value (line 15). This is because all trussnesses increase or remain the same after a star-motif insertion. A higher feasible lower bound can save lots of calculations. When there is no edge to

Algorithm 3 Find Seed Candidates for Trussness Refinement

Input: AffBall $\mathcal{B}$ for an inserted star motif S_c, lower bounds $\tau'(e)$ for $e \in \mathcal{B}_{in}$;
Output: Seed candidates $Seed \subseteq \mathcal{B}_{bd}$;

```
1: Seed ← ∅;
2: for all (u, v) ∈ B_bd do
3:     if min{τ((c, u)), τ((c, v))} < τ(u, v) and min{τ'((c, u)), τ'((c, v))} ≥ τ((u, v)) then
4:         Seed ← Seed ∪{(u, v)};
5: return Seed;
```

remove anymore, we increase k by one (line 12) and repeat the above process, until all edges are removed from $\mathcal{B}$. Note that for an inserted star motif S_c with single edge, we can directly get the estimated trussness $lowB$ by the definition of k-truss, i.e., $lowB(e) = \max\{k | \sup_H(e) \geq k-2\}$ where H is the k-truss.

Procedure of finding seed candidates in $\mathcal{B}_{bd}$. Algorithm 3 shows that, if a triangle $\triangle_{cuv}$ does not exist in the $\tau((u,v))$-truss in original graph G but exists in new $\tau((u,v))$-truss in new graph G', the edge (u,v) may increase its trussness and should be pushed into $Seed$ for trussness refinement.

Although Algorithm 1 works well to invoke an existing truss maintenance algorithms [12], it is not efficient due to a large number of candidate edges to consider in this unbounded algorithm. To further improve the efficiency, we develop a new onion layer based Algorithm 5 in the next section.

5 ONION LAYER BASED TRUSS REFINEMENT

In this section, we introduce two new concepts of onion layers and supports for trussness refinement. Then, we propose an integrated algorithm by combining AffBall and onion layer based techniques for truss maintenance with star insertions. Moreover, we extend insertion techniques to handle star motif deletions and analyze the algorithm complexity.

5.1 Onion Layers

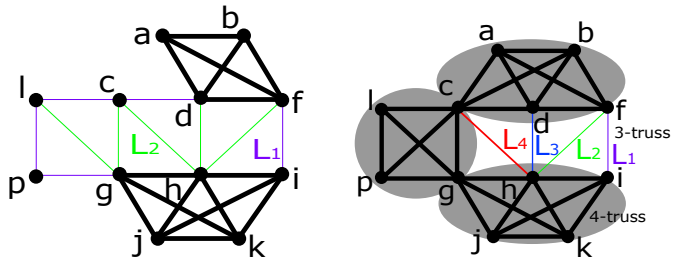
We first define an onion layer as follows.

Definition 6 (Onion Layer). *Given the k-class edges E_k and an integer $l \in \mathbb{Z}^+$, the j-th onion layer L_j is the set of edges satisfying $L_j = \{e \in E_k | \sup_{H_j}(e) \leq k-2\}$ where $H_j = T_k \setminus \bigcup_{i=1}^{j-1} L_i$, T_k is the k-truss, and the initial onion layer is $L_1 = \{e \in E_k : \sup_{T_k}(e) = k-2\}$.*

To mention a specific $k \in \mathbb{Z}^+$ for k-class edges, we denote the j-th onion layer of k-class edges as $L_{k,j}$. When the context is obvious, we call the j-th onion layer as L_j for simplicity throughout this paper. Based on onion layers, we can define the layer number and layer order.

Definition 7 (Layer Number $\mathcal{L}$ and Layer Order $\prec$). *Given a k-class edge $e \in E_k$, the layer number of e is defined as $\mathcal{L}(e) = \{l \in \mathbb{Z}^+ : e \in L_l\}$. The onion layer number $\mathcal{L}(e)$ indicates the number of rounds in which e is removed from k-truss in peeling process. Given two edges e_1 and e_2, the layer order $\prec$ is defined as $e_1 \prec e_2$, if either $\tau(e_1) < \tau(e_2)$ or $\tau(e_1) = \tau(e_2)$, $\mathcal{L}(e_1) < \mathcal{L}(e_2)$ holds.*

We use $e_1 = e_2$ to indicate $\tau(e_1) = \tau(e_2)$ and $\mathcal{L}(e_1) = \mathcal{L}(e_2)$. Moreover, $e_1 \preceq e_2$ holds for either $e_1 = e_2$ or $e_1 \prec e_2$. Note that $\mathcal{L}$ can represent the removal order of edges in the same k-class. The larger the $\mathcal{L}(e)$ is, the later the edge e is removed. For two k-class edges e_1 and e_2, $\mathcal{L}(e_1) = \mathcal{L}(e_2)$ represents that the edges e_1, e_2 are removed at the same round, and the peeling order of these edges does not affect the k-truss results. When comparing the removal order of two edges, it needs to consider both the trussness τ and onion layer number $\mathcal{L}$. Next, we define an onion support below.



(a) The old graph G (b) The new graph G'

Fig. 3. An example of onion layers in 3-class edges.

Definition 8 (Onion Support $\text{SeSup}_{k,l}$). *Given two integers $k \geq 2$ and $l \geq 1$, the onion support of an edge e is defined as the number of triangles lies in at least l-th onion layer of k-truss subgraph, i.e.,* $\text{SeSup}_{k,l}(e) = \sup_{H_l}(e)$, *where* $H_l = T_k \setminus \bigcup_{i=1}^{l-1} L_i$.

When $k = \tau(e)$ and $l = \mathcal{L}(e)$, we omit the subscripts and use $\text{SeSup}(e)$ to represent the number of triangles containing e, when the edge e is deleted in the k-truss decomposition.

Example 2. *Consider a graph G in Fig. 3(a) and $k = 3$. The 3-class edges E_3 have two onion layers L_1 and L_2, which are colored in purple and green, respectively. For the edge (c,d), it has the trussness of 3 and is contained in only one triangle $\triangle_{cdh}$. Thus, (c,d) is removed in the first round of truss decomposition, leading to $(c,d) \in L_1$ and $\mathcal{L}((c,d)) = 1$. As a result, the onion layer $L_1 = \{(c,d),(c,l),(l,p),(g,p),(f,i)\}$. After removing all edges of L_1 in the first round, the onion support $\text{SeSup}_{k,l}((f,h)) = 1 \leq k-2$ for $k = 3$ and $l = 2$, indicating that (f,h) will be removed in the second round and $(f,h) \in L_2$. Thus, we get $L_2 = \{(f,h),(d,h),(c,h),(c,g),(g,l)\}$. For two layer numbers $\mathcal{L}((c,d)) = 1$ and $\mathcal{L}((c,h)) = 2$, we infer that the layer order $(c,d) \prec (c,h)$ holds. When it inserts three edges (a,c), (b,c), (c,p) into G as a new graph G', the four new onion layers of 3-class edges are shown in Fig. 3(b), where $L_1 = \{(i,f)\}$, $L_2 = \{(h,f)\}$, $L_3 = \{(d,h)\}$, and $L_4 = \{(c,h)\}$.*

Computing onion layers and supports. Algorithm 4 shows the procedure of obtaining $\mathcal{L}$ and SeSup in the process of truss decomposition. First of all, the support numbers of all edges are calculated (line 1). The queue Q contains all the edges to be removed (line 2). The edges that are contained in no more than $k-2$ triangles are pushed into Q (lines 4-13) and will be removed one by one (lines 8-15). Once an edge e is pushed into Q, $\tau(e)$ and $\mathcal{L}(e)$ are determined and not changed (lines 5, 13), but $SeSup(e)$ may further decrease due to the removed triangles (line 15). After an edge e is removed from graph, all its neighbor edges should update the support numbers accordingly (lines 10-15). It also checks the edges not in Q and push them into Q if their support have no more than $k-2$ (line 13). Meanwhile, it also update the onion support $SeSup$ for edges already in Q (line 15). When the graph becomes empty by removing all edges E, the algorithm terminates.

5.2 Onion Layer Incremental Maintenance

Truss updating rules. Next, we introduce the truss updating rules based on onion layers $\mathcal{L}(e)$ and onion support $\text{SeSup}(e)$.

Lemma 4. *Consider an edge e in graph G, and two integers $k = \tau(e)$, $l = \mathcal{L}(e)$. If $l = 1$, $\text{SeSup}(e) = k-2$ holds; If $l > 1$, $\text{SeSup}(e) \in [0, k-2]$ and $\text{SeSup}_{k,l-1}(e) > k-2$ hold.*

Proof. When $l = 1$, no edge in the k-truss has been removed, so $\text{SeSup}(e) = \sup_{T_k}(e) \geq k-2$. And e is removed in the first round, so $\text{SeSup}(e) \leq k-2$. Therefore, $\text{SeSup}(e) = k-2$ when $\mathcal{L}(e) = 1$. Next, when $l \neq 1$, $\text{SeSup}(e)$ cannot be negative and $\text{SeSup}(e) \leq k-2$, otherwise e will

Algorithm 4 Compute Onion Layers and Supports

Input: Graph $G = (V, E)$;
Output: Trussness $\tau(e)$, onion layer number $\mathcal{L}(e)$, onion support $SeSup(e)$ for all edges $e \in E$;

```
1: Calculate the support sup(e) for e ∈ E;
2: Initialization: k ← 2, l ← 1, a queue Q ← ∅;
3: while E ≠ ∅ do
4:    for all e ∈ E, sup(e) ≤ k − 2 do
5:       Q ← Q ∪ {e}, τ(e) ← k, L(e) ← l, SeSup(e) ← sup(e); // Batch assignment of edges in
         the same layer
6:    if Q = ∅ then k ← k + 1, l ← 1;
7:    while Q ≠ ∅ do
8:       Pop an edge (u, v) from Q; Remove (u, v) from E;
9:       Assign the current layer number: l ← L((u, v));
10:      for all e ∈ {(u, w), (v, w) ∈ E|w ∈ N(u) ∩ N(v)} do
11:         if e ∉ Q then
12:            Decrease the support sup(e) by one;
13:            if sup(e) ≤ k − 2 then τ(e) ← k, L(e) ← l + 1, SeSup(e) ← sup(e), Q ← Q ∪ {e};
14:         else if l < L(e) then
15:            Decrease SeSup(e) by one;
16: return τ(e), L(e) and SeSup(e) for e ∈ E;
```

be in $l+1$ layer. So $SeSup(e) \in [0, k-2]$. Since e is in l layer, so in $l-1$ layer, $\sup_{H_{l-1}}(e) > k-2$, otherwise, e would be in $l-1$ layer. □

The key rule of our updating approach is to ensure the exact update of all edges' SeSup following Lemma 4, as a star insertion brings the change of triangles in support calculations. Once the onion support of all edges satisfies Lemma 4, $\mathcal{L}$ and τ are also updated, the truss maintenance process ends. In the following, we study how to efficiently maintain SeSup by pruning candidate edges.

Lemma 5. *Given an edge (u, v), if $k = \tau((u, v))$ and $l = \mathcal{L}((u, v)) > 1$, there exists a neighbor edge in $l-1$ layer, i.e., $\exists e' \in \{(u, w), (v, w)|w \in N(u) \cap N(v)\}$, $\tau(e') = k$ and $\mathcal{L}(e') = l - 1$.*

Proof. According to Lemma 4, if $\mathcal{L}((u, v)) = l$, $\sup_{H_{l-1}}((u, v)) > k - 2$. If e' does not exist, $\sup_{H_l}((u, v)) = \sup_{H_{l-1}}((u, v)) > k-2$. Then, $\mathcal{L}((u, v))$ should be $l+1$, which leads to a contradiction. □

The above lemma shows the local property of onion layer, i.e., $\mathcal{L}(e)$ can be affected by the change of onion layer of its neighbors. The following lemma shows the condition when an edge increases its onion layer number.

Lemma 6. *Given an edge e, $k = \tau(e)$ and $l = \mathcal{L}(e)$, if $\mathsf{SeSup}'_{k,l}(e) > k - 2$, then the new onion layer number $\mathcal{L}'(e) > \mathcal{L}(e)$ holds.*

Proof. According to Lemma 4, $\mathsf{SeSup}_{k,l}(e) \leq k-2$ and $\mathsf{SeSup}_{k,l-1}(e) > k-2$. If new $\mathsf{SeSup}'_{k,l}(e) > k - 2$, the edge e will not be peeled in l layer, so new $\mathcal{L}'(e) > \mathcal{L}(e)$. □

Lemma 6 offers a very useful rule of maintaining the onion layers in BFS manner. Unlike trussness that increases the support number cannot guarantees the trussness increment, an edge e must increase its onion layer when $\mathsf{SeSup}(e) > k - 2$. Once the onion layers of the graph is maintained, the trussness of the graph is also updated. Therefore, we can design a truss maintenance algorithm

Algorithm 5 Onion Layer based Truss Refinement (Insertion)

Input: Graph $G = (V, E)$, a star motif S_c, trussness $\tau(e)$, onion layer $\mathcal{L}(e)$ and support $SeSup(e)$ for all edges e;
Output: New $\tau'(e)$, $\mathcal{L}'(e)$ and $SeSup'(e)$ for all edges e;

1: Find seed edges: $Seed \leftarrow \{e \in \mathcal{B}_{bd} : e$ violdates Lemma 4$\}$;
2: Initialize empty queues Q_{seed}, Q_{k+1}, $Q_k \leftarrow \emptyset$;
3: **while** $Seed \neq \emptyset$ **do**
4: Assign $k \leftarrow \max\{\tau(e)|e \in Seed\}$;
5: **for all** $e \in Seed$, $\tau(e) = k$ **do**
6: $Q_{seed} \leftarrow Q_{seed} \cup \{e\}$, $Seed \leftarrow Seed \setminus \{e\}$;
7: **while** $Q_{seed} \neq \emptyset$ **do**
8: Find $(u, v) \in Q_{seed}$ with the smallest $l = \mathcal{L}((u, v))$;
9: **if** $\exists e \in Q_k$ with $\mathcal{L}(e) < l$ **then**
10: $Q_k \leftarrow Q_k \setminus \{e\}$;
11: **for all** $e' \in Q_{seed} \cup Q_{k+1}$ forming a triangle with e **do**
12: Update $SeSup(e')$;
13: **if** $\text{SeSup}(e') \leq k - 2$ **then**
14: Assign $\mathcal{L}'((u, w)) \leftarrow l$;
15: Delete e' from Q_{seed} or Q_{k+1};
16: **continue**;
17: Assign $\mathcal{L}'((u, v)) \leftarrow +\infty$, $Q_{seed} \leftarrow Q_{seed} \setminus \{(u, v)\}$;
18: **for all** $w \in N(u) \cap N(v), \min(\tau((u, w)), \tau((v, w))) = k, (u, v) \preceq (u, w), (u, v) \preceq (v, w)$ **do**
19: **if** $(u, v) \prec (u, w)$ **and** $(u, v) \prec (v, w)$ **then**
20: **if** $(u, w) \prec' (u, v)$ **and** $(u, w) \prec' (v, w)$ **then**
21: Increase the support $SeSup'((u, w))$ by one;
22: **if** $\text{SeSup}'((u, w)) > k - 2$ **then**
23: $Q_{seed} \leftarrow Q_{seed} \cup \{(u, w)\}$;
24: **else if** $\tau(u, w) = k$ **and** $\mathcal{L}(u, w) = l$ **then**
25: $Q_k \leftarrow Q_k \cup \{(u, w)\}$;
26: Operate edge (v, w) similar for as line 19-25;
27: $Q_{k+1} \leftarrow Q_{k+1} \cup \{(u, v)\}$;
28: **if** $Q_k \neq \emptyset$ **then**
29: Remove edges in Q_k and update neighbor edges' $SeSup$ and $\mathcal{L}$ values in Q_{seed}, Q_{k+1} as lines 11-15;
30: **for all** $e \in Q_{k+1}$ **do**
31: Assign $\tau'(e) \leftarrow k + 1$, $\mathcal{L}'(e) \leftarrow 1$;
32: Update $\mathcal{L}'$ and $SeSup'$ of Q_{k+1} in $(k + 1)$-truss as lines 5-29;
33: **return** $\tau'(e)$, $\mathcal{L}'(e)$ and $SeSup'(e)$ for all edges e;

bounded by the size of edges that change onion layers. Moreover, according to Lemma 6, we infer that for two edges e, $\hat{e}$ and $e \prec \hat{e}$, then $\hat{e}$ may be affected when e has an increased layer number $\mathcal{L}(e)$. However, for other edges if $e' \preceq e$, the increased layer number $\mathcal{L}(e)$ has no effect on e'.

The algorithm. We first define the change of edges in terms of trusssness and onion layer number as follows.

Definition 9 (T_CHG and L_CHG). *Given a graph $G = (V, E)$ inserted with a star motif S_c, the set of edges with trussness changed is defined as* $\mathsf{T\_CHG} = \{e|\tau(e) \neq \tau'(e), e \in E\} \cup S_c$, *and the set of edge with layer changed is defined as* $\mathsf{L\_CHG} = \{e|\tau(e) = \tau'(e), \mathcal{L}(e) \neq \mathcal{L}'(e), e \in E\} \cup \mathsf{T\_CHG}$.

We first consider one simple method to update onion layers. According to Lemma 6, given a seed candidate edge e in $\mathcal{B}_{bd}$, we put all neighbor edges e' that $e \prec e'$, $\tau(e') = \tau(e)$ into *Seed* and so on. Then, we conduct a truss decomposition for edges in *Seed* to calculate new onion layers and trussnesses, whcih finishes the update to admit Lemma 4. However, this method are not efficient, as the size of *Seed* may be very large and not bounded by |L_CHG|, i.e., there may be useless edges in *Seed* with no changed onion layer numbers.

To tackle the above limitation, we propose our onion layer based maintenance algorithm, which is bounded by a polynomial time complexity w.r.t. |L_CHG| to handle each edge just once. The key idea is to remove edges that cannot change their onion layer numbers in time. Generally speaking, we need three edge sets: Q_{seed}, Q_{k+1}, Q_k, where Q_{seed} keeps the seed edges not meeting Lemma 4 that increase the support of their neighbors, Q_{k+1} keeps the edges from Q_{seed} that may changes the turssness from k to $k+1$, and Q_k keeps the visited k-truss edges that cannot become a $(k+1)$-truss edge. Specifically, we consider to update the k-class edges for a particular k. We first identify all seed edges in $\mathcal{B}_{bd}$ violating Lemma 4 as Q_{seed}. We handle the edges in Q_{seed} in increasing order of old layer numbers $\mathcal{L}(e)$. Assume that the minimum $\mathcal{L}(e)$ in Q_{seed} is l. The process of handling an edge $e \in Q_{seed}$ with $l = \mathcal{L}(e)$ is as follows. We first set $\mathcal{L}'(e) = +\infty$ and update its neighbor edges' SeSup; Next, we will put its neighbor edges e_b having $\mathcal{L}(e_b) > l$ and $\mathsf{SeSup}(e_b) > k-2$ into Q_{seed} and also the edges $\hat{e}$ whose $\mathcal{L}(\hat{e}) = l$ into Q_k; Then, the handled edge e is pushed into Q_{k+1}. Before we handle edges in Q_{seed}, we first remove edges e whose $\mathcal{L}(e) < l$ in Q_k, which will affect SeSup for edges in Q_{seed} and Q_{k+1}. For those affected edges e, if they meet Lemma 4 again, we update their $\mathcal{L}'(e) = l$ and they finish updating and are removed from Q_{seed} or Q_{k+1}. If we remove all edges in Q_{seed} and there is no edge in Q_k, all edges $e \in Q_{k+1}$ should increase their trussness $\tau(e)$ by one. After adjusting onion layer numbers $\mathcal{L}$ in the new $(k+1)$-class, the algorithm terminates.

Algorithm 5 outlines the details of onion layer based trussness refinement. First, it creates three empty edge sets Q_{seed}, Q_{k+1}, Q_k in line 2. The edges not meeting Lemma 4 are pushed into Q_{seed} (line 6, 23). Edges handled (checking all neighbors, lines 17-27) are pushed into Q_{k+1} (line 27). Adjacent edges that are in current layer l are pushed into Q_k (line 25). Before handling edges in Q_{seed}, the edges e with $\mathcal{L}(e) < l$ are removed from Q_k (line 9), which further affect *SeSup* for edges in Q_{seed} and Q_{k+1}. Finally, after removing all edges in Q_{seed} and Q_k, edges remained in Q_{k+1} should increase their τ (line 31). We also need to adjust onion layers in $(k+1)$-truss accordingly, which only runs once as the trussness only increase by at most by one for a star insertion (line 32).

Example 3. *The onion layer of the 3-class is shown in Fig. 2(a)(c). After inserting $e_0 = (u, v)$, we get a degenerate $\mathcal{B}$ that $\mathcal{B}_{in} = \{e_0\}$, $\mathcal{B}_{bd} = \{(u, p), e_3, e_1, (w, v)\}$. Since it is a degenerate $\mathcal{B}$ and we can directly get $\tau(e_0) = 3$, $\mathcal{L}(e_0) = 2$,* $\mathsf{SeSup}(e_0) = 1$ *by counting its neighbors. Next, we update* SeSup *for edges in $\mathcal{B}_{bd}$ and we get* $\mathsf{SeSup}'(e_1) = 2$. *Since* $\mathsf{SeSup}'(e_1) = 2 > k-2 = 3-2 = 1$, *we get $Seed = \{e_1\}$. In the maintenance process (Algorithm 5), we first put e_1 into Q_{seed} (line 6). Then e_1 is popped out from Q_{seed} (line 17) and we set $\mathcal{L}'(e_1) = +\infty$. The increase of $\mathcal{L}'(e_1)$ will increase* $\mathsf{SeSup}(e_0)$ *and* $\mathsf{SeSup}(e_2)$ *(line 21). Again,* $\mathsf{SeSup}'(e_0) = 2 = \mathsf{SeSup}'(e_2) > k-2$, *these two edges are put into Q_{seed} (line 23). After visiting all neighbors of e_1, it is put into Q_{k+1} (line 27). In the next round, e_0 is popped out from Q_{seed} and after visiting all its neighbors, the states change to the following: $Q_{seed} = \{e_2, e_3\}$, $Q_{k+1} = \{e_1, e_0\}$,* $\mathsf{SeSup}'(e_3) = 2$. *Since $\mathcal{L}(e_2) = 2 < \mathcal{L}(e_3) = 3$, e_2 is popped out from Q_{seed}, and the states change to the following: $Q_{seed} = \{e_3\}$, $Q_{k+1} = \{e_1, e_0, e_2\}$,* $\mathsf{SeSup}'(e_3) = 3$. *After handling e_3, the states change to the following: $Q_{seed} = \{\}$, $Q_{k+1} = \{e_1, e_0, e_2, e_3\}$, $Q_k = \{e_{11}\}$. Then Q_{seed} becomes empty, so we end the BFS process. After removing e_{11} in Q_k,* $\mathsf{SeSup}'(e_3)$ *changes from*

Algorithm 6 Compute Lower Bounds and Onion Layers/Supports

Input: AffBall $\mathcal{B}$ for an inserted star motif S_c, trussness $\tau(e)$, $\mathcal{L}(e)$ and $SeSup(e)$ for $e \in \mathcal{B}$;
Output: Estimated $lowB(e)$, $\mathcal{L}'(e)$ and $SeSup'(e)$ for $e \in \mathcal{B}_{in}$;
1: Calculate the support $\text{sup}_{\mathcal{B}}(e)$ in AffBall $\mathcal{B}$ for edge $e \in \mathcal{B}_{in}$;
2: Initialize $l \leftarrow 1$, $k \leftarrow 2$, an empty queue $Q \leftarrow \emptyset$;
3: **while** $\mathcal{B}_{in} \neq \emptyset$ **or** $Q \neq \emptyset$ **do**
4: **for all** $(c, v) \in \mathcal{B}_{in}$ with $\text{sup}_{\mathcal{B}}((c, v)) \leq k - 2$ **do**
5: Assign $lowB((c, v)) \leftarrow k$, $\mathcal{L}'((c, v)) \leftarrow l$, $SeSup'((c, v)) \leftarrow \text{sup}_{\mathcal{B}}((c, v))$; $Q \leftarrow Q \cup \{v\}$;
6: **if** $Q = \emptyset$ **then**
7: **for all** $(v, w) \in \mathcal{B}_{bd}$, $\tau((v, w)) \leq k - 2$, $\mathcal{L}((v, w)) \leq l$ **do**
8: Remove edge (v, w) from $\mathcal{B}$;
9: Decrease $\text{sup}_{\mathcal{B}}((c, v))$ and $\text{sup}_{\mathcal{B}}((c, w))$ by one;
10: **if** $\text{sup}_{\mathcal{B}}((c, v)) \leq k - 2$ **then** $Q \leftarrow Q \cup \{v\}$, update $lowB$, $\mathcal{L}'$, $SeSup'$ as line 5;
11: **if** $\text{sup}_{\mathcal{B}}((c, w)) \leq k - 2$ **then** $Q \leftarrow Q \cup \{w\}$, update $lowB$, $\mathcal{L}'$, $SeSup'$ as line 5;
12: Increase l by one;
13: **if** $Q = \emptyset$ **and** $\forall (v, w) \in \mathcal{B}_{bd}$, $\tau((v, w)) > k - 2$ **then**
14: Assign $k \leftarrow k + 1$, $l \leftarrow 1$;
15: **while** $Q \neq \emptyset$ **do**
16: **if** $\exists v \in Q$, $\mathcal{L}'((c, v)) \leq l$ **then**
17: $Q \leftarrow Q \setminus \{v\}$, $\mathcal{B}_{in} \leftarrow \mathcal{B}_{in} \setminus \{(c, v)\}$;
18: **for all** $(v, w) \in \mathcal{B}_{bd}$ **do**
19: Remove (v, w) and update $\text{sup}_{\mathcal{B}}$, Q, $\mathcal{B}_{in}$ as lines 9-11;
20: **else if** $\exists (v, w) \in \mathcal{B}_{bd}$, $\tau((v, w)) \leq k - 2$, $\mathcal{L}((v, w)) \leq l$ **then** Break;
21: **else then** Increase l by one;
22: **return** $lowB(e)$, $\mathcal{L}'(e)$ and $SeSup'(e)$ for $e \in \mathcal{B}_{in}$;

3 *to* 2*, which is still larger than* $k - 2$*, so we keep* e_3 *in* Q_{k+1}*. Finally, we add these* 4 *edges into the* 4*-truss and after updating their* $\mathcal{L}$ *and* SeSup *in the* 4*-truss, the algorithm ends.*

5.3 Our Complete Star Insertion Approach

We present a complete approach for star insertion using all the above techniques. Recall that our AffBall based framework in Algorithm 1 has two phases of truss estimation and truss refinement. For the truss estimation in Algorithm 2, it only computes the lower bounds of $e \in \mathcal{B}_{in}$, but the onion layers and supports are not computed in $\mathcal{B}_{in}$. To dismiss it, we revise Algorithm 2 to Algorithm 6.

Computing lower bounds, onion layers, and onion supports. Algorithm 6 shows how to compute $\mathcal{L}$ and $SeSup$ in AffBall, which combines the technique of Algorithm 2 and Algorithm 4 in one process of local truss decomposition. We have two global variables k and l and a queue Q (line 2), where k is the current trussness and l is the current onion layer and Q will hold nodes that are not in $(k + 1)$-truss; Similar to truss decomposition, we will peel edges in $\mathcal{B}_{in}$ as k increases (lines 3-21). For a specific k, we will peel edges as l increases (lines 15-21). After all edges in $\mathcal{B}_{in}$ are removed, we get lower bounds of τ', $\mathcal{L}'$ and $SeSup'$ for edges in $\mathcal{B}_{in}$ (line 5, 17).

Finally, our complete method for star insertion is presented in Algorithm 1, which uses Algorithm 6 (line 1 of Algo. 1) for truss estimation in AffBall and Algorithm 5 (lines 5-6 of Algo. 1) for truss refinement. Note that Algorithm 5 uses a smaller set of seed candidates instead of the *Seed* in Algorithm 3.

Complexity analysis. We analyze the algorithm complexity.

Table 1. A comparison of time complexity and boundedness of different truss maintenance algorithms.

		XH [12]	**NodePP** [8]	**Order** [41]	**Ours**
Insertion	Time complexity	$O(\sum_{e^+ \in S_c} f(\|\|H_{e^+}\|\|_1))$	$O(f(\|\|H^*\|\|_1))$	$O(f(\|\|AFF^{\preceq}\|\|_1))$	$O(f(\|\|\mathsf{L\_CHG}\|\|_1))$
	Boundedness	×	×	✓	✓
Deletion	Time complexity	$O(f(\|\|\mathsf{T\_CHG}\|\|_1))$	-	$O(f(\|\|AFF^{\preceq}\|\|_1))$	$O(f(\|\|\mathsf{L\_CHG}\|\|_1))$
	Boundedness	✓	-	✓	✓

Theorem 1. *Our star insertion method for truss maintenance in Algorithm 1, equipped with Algorithm 6 for truss estimation and Algorithm 5 for truss refinement, takes $O(|\mathcal{N}_L| \log |\mathcal{N}_L| + T_{\triangle}(\mathcal{N}_L))$ time in $O(|E|)$ space, where $\mathcal{N}_L$ denotes the edge set of the 1-hop neighborhood of* L_CHG *and $T_{\triangle}(\mathcal{N}_L))$ is the time taken to list triangles containing e for all edges $e \in \mathcal{N}_L$.*

Proof. The two main steps of the algorithm are Algorithm 6 and Algorithm 5. First of all, we prove that Algorithm 5 is bounded by L_CHG. Following [29], the maintenance algorithm on a graph is bounded if the algorithm will only visit the h-hop neighborhood of edges whose state variables will be modified after maintenance. Obviously, only state of edges in L_CHG changed after applying our onion layer maintenance Algorithm 5. So next, we will show that Algorithm 5 will only visit the 1-hop neighborhood of L_CHG. Before handling edge e with $l = \mathcal{L}(e)$ in Q_{seed} in Algorithm 5, edges e' in Q_k with $\mathcal{L}(e') < l$ are removed. If e is still in Q_{seed}, e must increase its $\mathcal{L}$. Therefore, edges in Q_{k+1} are in L_CHG, and edges used to be in Q_{seed} and Q_k are in the 1-hop neighborhood of L_CHG. The maximun size of Q_{seed} and Q_k is $\mathcal{N}_L$, and for each edge e in Q_{seed} and Q_k, it takes $O(\log |\mathcal{N}_L|)$ to find the minimum $\mathcal{L}$ and $O(T_{\triangle}(\{e\}))$ to list triangles containing e, so the time complexity of Algorithm 5 is$O(|\mathcal{N}_L| \log |\mathcal{N}_L| + T_{\triangle}(\mathcal{N}_L))$. Secondly, we prove that Algorithm 6 is also bounded by L_CHG. This is because Algorithm 6 will only visit the 1-hop neighborhood of the center node c. Algorithm 6 needs to list triangles containing edges in $\mathcal{B}_{in}$ and order edges in $\mathcal{B}_{bd}$, so it takes $O(|\mathcal{B}_{bd}| \log |\mathcal{B}_{bd}| + T_{\triangle}(\mathcal{B}_{in})) = (|\mathcal{N}_L| \log |\mathcal{N}_L| + T_{\triangle}(\mathcal{N}_L))$ time. Finally, our algorithm keeps state variables for all edges in the graph, thus the space complexity is $O(|E|)$. In conclusion, the star insertion algorithm is bounded by L_CHG and takes $O(|\mathcal{N}_L| \log |\mathcal{N}_L| + T_{\triangle}(\mathcal{N}_L))$ time in $O(|E|)$ space. □

5.4 Handle Star Deletions

In this section, we discuss how to extend our star-based insertion techniques to handle star deletions.

For a graph $G(V, E)$ with a star motif S_c deleted, we maintain the trussnesses of all edges e in new graph $G'(V, E \setminus S_c)$ using the following solution. We adopt an AffBall-based insertion framework in Algorithm 1 to handle a star deletion similarly, which also has two steps of *Phase I* and *Phase II*. Specifically, in Phase I, we first remove S_c from graph G to generate new graph G'. We then invoke a local truss decomposition similar as Algorithm 6 on AffBall to get the corresponding lower bounds, onion layers, and onion supports for edges in $\mathcal{B}_{in}$. For now, all edges in $\mathcal{B}_{in} \cup \mathcal{B}_{out}$ satisfy the maintenance rule of Lemma 4. We find the seed candidates of edges in $\mathcal{B}_{bd}$ that violates Lemma 4, which may have decreased trussnesses in G'. Next, in Phase II, we recalculate the edge trussnesses in Algorithm 7, which is similar as the insertion one in Algorithm 5. We show the details of *Phase I* and *Phase II* to handle star deletions as follows.

Phase I: AffBall **deletion based truss estimation**. We still use Algorithm 6 to compute the lower bounds $lowB(e)$, $\mathcal{L}'(e)$ and $SeSup'(e)$ for $e \in \mathcal{B}_{in}$. The reason that Algorithm 6 can be applied to both star insertion and deletion is that we simply use the local truss decomposition to obtain feasible values of $\tau'(e)$, $\mathcal{L}'(e)$ and $SeSup'(e)$ such that edges in $\mathcal{B}_{in}$ satisfy Lemma 4 again.

Phase II: onion layer based truss refinement. Similar with the star insertion case in Lemma 6, we have the following updating rules for star deletions.

Algorithm 7 Onion Layer based Truss Refinement (Deletion)

Input: Graph $G = (V, E)$, a deleted star S_c, trussness $\tau(e)$, onion layer $\mathcal{L}(e)$ and support $SeSup(e)$ for all edges $e \in E$;
Output: New $\tau'(e)$, $\mathcal{L}'(e)$ and $SeSup'(e)$ for all edges $e \in E$;
1: Find seed edges: $Seed \leftarrow \{e \in \mathcal{B}_{bd} : e$ violates Lemma 4$\}$;
2: Initialize empty queues $Q_{seed} \leftarrow \emptyset$;
3: **while** $Seed \neq \emptyset$ **do**
4: Assign $k \leftarrow \min\{\tau(e) | e \in Seed\}$;
5: **for all** $e \in Seed$, $\tau(e) = k$ **do**
6: $Q_{seed} \leftarrow Q_{seed} \cup \{e\}$, $Seed \leftarrow Seed \setminus \{e\}$;
7: **while** $Q_{seed} \neq \emptyset$ **do**
8: Find (u, v) in Q_{seed} with the largest $l = \mathcal{L}((u, v))$, $Q_{seed} \leftarrow Q_{seed} \cup \{(u, v)\}$;
9: Recalculate $\tau'((u, v))$, $\mathcal{L}'((u, v))$ and $SeSup'((u, v))$ according to neighbors;
10: **for all** $w \in N(u) \cap N(v)$ **do**
11: **if** (u, w) not satisfying Lemma 4 **then**
12: $Q_{seed} \leftarrow Q_{seed} \cup \{(u, w)\}$;
13: Operate edge (v, w) similarly in lines 11-12;
14: **return** τ, $\mathcal{L}$ and $SeSup$;

Lemma 7. *Given an edge e, $k = \tau(e)$ and $l = \mathcal{L}(e)$, if new $\mathrm{SeSup}'_{k,l-1}(e) \leq k - 2$, new onion layer numbers $\mathcal{L}'(e) < \mathcal{L}(e)$.*

Proof. According to Lemma 4, $\mathrm{SeSup}_{k,l}(e) \leq k-2$ and $\mathrm{SeSup}_{k,l-1}(e) > k-2$. If new $\mathrm{SeSup}'_{k,l-1}(e) \leq k - 2$, the edge e will be peeled in $l - 1$ layer, so new $\mathcal{L}'(e) < \mathcal{L}(e)$. □

Based on Lemma 7, we propose Algorithm 7 for truss refinement. It maintains the onion layers of all edges in decreasing order of their original onion layers in graph G'. Similar to star insertions in Algorithm 5, we put all edges in $Seed$ into Q_{seed} (line 6), update τ', $\mathcal{L}'$ and $SeSup'$ for edges in Q_{seed} (line 9) and put their neighbors that not satisfy Lemma 4 into Q_{seed} (line 12). The algorithm stops when all edges in Q_{seed} are removed.

Complexity analysis. We analyze the complexity of star deletions.

Theorem 2. *Under our star-based truss maintenance framework in Algorithm 1, the star deletion method using Algorithm 7 for truss refinement totally takes $O(|\mathcal{N}_L| \log |\mathcal{N}_L| + T_{\triangle}(\mathcal{N}_L))$ time in $O(|E|)$ space.*

Proof. Edges in Q_{seed} are all not satisfying Lemma 4, so they will change $\mathcal{L}$ and are in L_CHG. For each edge in Q_{seed}, we will check all its neighbors, which are in the 1-hop neighborhood of L_CHG. And we need to order edges in Q_{seed} according to their $\mathcal{L}$ and list triangles containing edges in Q_{seed}. So the total time complexity is $O(|\mathcal{N}_L| \log |\mathcal{N}_L| + T_{\triangle}(\mathcal{N}_L))$. Again, we keep state variables for all edges in the graph, so the space complexity is $O(|E|)$. Therefore, Algorithm 7 is also bounded by L_CHG. □

5.5 Complexity Analysis and Comparison

In this section, we analyze different truss maintenance algorithms XH [12], NodePP [8], Order [41], and also our approach for star insertions/deletions, in terms of time complexity and boundedness, as shown in Table 1. We adopt a comparison approach of complexity and boundedness for graph incremental algorithms following [29, 41]. We use the notation f to represent some polynomial

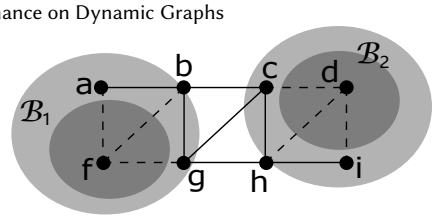


Fig. 4. An example of inserting two disjoint star motifs S_f and S_d with 6 new edges.

functions and $||H||_1$ to represent the size of 1-hop neighborhood of a given edge set H, i.e., $||H||_1 = |\{(u,w) \in E | \forall (u,v) \in H, w \in N(u)\}|$.

We first consider the case of star insertion S_c. We analyze all algorithms as follows.

- We analyze XH [12]. Following [41], for a single-edge insertion $e^+ = (u,v)$, we infer the insertion time complexity of XH [12] is $O(f(||H_{e^+}||_1))$, where $H_{e^+} = \{e^+\} \cup \bigcup_{k=2}^{K_{XH}} T_k$, T_k is the set of k-class edges connected with e^+, and the local maximum trussness $K_{XH} = \max\{\min\{\tau((u,w)), \tau((v,w))\} | w \in N(u) \cap N(v)\}$. To perform a star insertion, it needs to apply XH [12] multiple times to insert new edges $e^+ \in S_c$ one by one, leading to a total time complexity of $O(\sum_{e^+ \in S_c} f(||H_{e^+}||_1))$.
- Next, we analyze NodePP [8]. It has two steps, first removing all incident edges to node c and then inserting these edges and S_c back to the new graph $G'(V', E')$, which invokes similar maintenance techniques with [12]. Thus, the edge set $H_{pp} = \{(u,v) \in E' | u, v \in N'(c) \cup \{c\}\}$ and the local maximum trussness $K_{PP} = \max\{\tau(e) | e \in H_{pp}\}$. The candidate edges are $H^* = H_{pp} \cup \bigcup_{k=2}^{K_{PP}} T_k$, where T_k is the set of k-class edges connected with edges in H_{pp}. As a result, the time complexity of NodePP [8] is $O(f(||H^*||_1))$.
- Then, we analyze Order [41]. It utilizes the truss decomposition order $\preceq$ to maintain trussness. After maintenance, it returns new trussnesses, as well as a new order $\preceq'$ of edges. Consequently, the number of affected edges is $|AFF^{\preceq}|$ [41], where $AFF^{\preceq} = \mathsf{T\_CHG} \cup \{e \in E | \exists e' \in E \setminus AFF^{\preceq}, e \preceq e', e' \preceq' e\}$. The star insertion time complexity of Order is $O(f(||AFF^{\preceq}||_1))$ [41].
- According to Theorem 1, our insertion time complexity is $O(f(||\mathsf{L\_CHG}||_1))$, where $\mathsf{L\_CHG} = \mathsf{T\_CHG} \cup \{e \in E | \mathcal{L}(e) \neq \mathcal{L}'(e)\}$.

In summary, both our method and Order [41] are bounded, with regard to $f(||\mathsf{L\_CHG}||_1)$ and $f(||AFF^{\preceq}||_1)$, respectively. However, XH [12] and NodePP [8] are unbounded [41], where $f(||H_{e^+}||_1)$ and $f(||H^*||_1)$ can be extremely large. Moreover, $f(||\mathsf{L\_CHG}||_1)$ is much less than $\sum_{e^+ \in S_c} f(||H_{e^+}||_1)$ and $f(||H^*||_1)$, and also smaller than $f(||AFF^{\preceq}||_1)$ in practice as validated by our efficiency experiments. Thus, our algorithm for star insertions are more efficient than other competitors.

Next, we consider the case of star deletion S_c. All three algorithms XH [12], Order [41], and our method are bounded with regard to $f(||\mathsf{T\_CHG}||_1)$, $f(||AFF^{\preceq}||_1)$ and $f(||\mathsf{L\_CHG}||_1)$, respectively. They are all competitively efficient, but Order [41] and our method take additional efforts to maintain peeling order and onion layers, respectively.

6 HANDEL GENERAL UPDATES

In this section, we discuss how to handle general graph updates where inserted/deleted edges may randomly appear anywhere in the graph. We focus on the insertion case here, and the deletion case can be similarly handled. For general graph insertions, we can convert inserted edges into multiple star motifs. A straightforward way to handle multiple star insertions is to insert them one by one using our developed algorithms. But this method may repeatedly check some edges, which results in redundancy. Thus, in this section, we propose a more efficient method to handle multiple star motifs $\mathbb{S} = \{S_{c_1}, \cdots, S_{c_h}\}$ for an integer $h \geq 1$.

Algorithm 8 Handle Multiple Star Insertions

Input: Graph $G = (V, E)$, trussness $\tau(e)$, onion layer $\mathcal{L}(e)$ and $SeSup(e)$ for $e \in E$, multiple inserted stars $\mathbb{S} = \{S_{c_1}, \cdots, S_{c_h}\}$ for $h \geq 1$;
Output: New $\tau'(e)$, $\mathcal{L}'(e)$ and $SeSup'(e)$ for all edges e;
1: **while** $\mathbb{S} \neq \emptyset$ **do**
2: $\mathbb{B} =$ **GreedySplitStars**$(G, \mathbb{S})$;
3: Initialize empty set $Seed \leftarrow \emptyset$;
4: **for all** $\mathcal{B} \in \mathbb{B}$ **do**
5: Local trussness decomposition of $\mathcal{B}$ by Algorithm 6;
6: Temporarily set $\tau'(e) \leftarrow lowB(e)$ for $e \in \mathcal{B}_{in}$;
7: Add edges not satisfying Lemma 4 in $\mathcal{B}_{bd}$ to $Seed$;
8: Update trussness $\tau'(e)$ by Algorithm 5 starting from $Seed$;
9: **return** $\tau'(e)$, $\mathcal{L}'(e)$ and $SeSup'(e)$ for all edges e;

Procedure **GreedySplitStars** (G = (V, E), $\mathbb{S}$)
10: Initialize empty set $\mathbb{B} \leftarrow \emptyset$, $\hat{E} \leftarrow E$;
11: **for all** $S_{c_i} \in \mathbb{S}$, in descending order of size $|S_{c_i}|$ **do**
12: Get $\mathcal{B}$ according to S_{c_i};
13: **if** $\mathcal{B} \cap \mathcal{B}_j = \emptyset, \forall \mathcal{B}_j \in \mathbb{B}$ **then**
14: $\mathbb{B} \leftarrow \mathbb{B} \cup \{\mathcal{B}\}$; $\mathbb{S} \leftarrow \mathbb{S} \setminus \{S_{c_i}\}$; $\hat{E} \leftarrow \hat{E} \cup S_{c_i}$;
15: **return** $\mathbb{B}$;

Recall that in Phase II of our proposed algorithm, we only need to maintain trussnesses and onion layers by checking the edges in $\mathcal{B}_{bd}$ since the other edges satisfy the k-truss constraint. As there are h AffBalls $\mathcal{B}_1, \cdots, \mathcal{B}_h$ for h inserted stars in $\mathbb{S}$, we can treat them as one AffBall $\mathcal{B}$, where $\mathcal{B}_{in} = \cup(\mathcal{B}_i)_{in}$ and $\mathcal{B}_{bd} = \cup(\mathcal{B}_i)_{bd}$. In the other words, we can conduct local truss decomposition for each AffBall $\mathcal{B}_i$ where $1 \leq i \leq h$, and then maintain trussnesses and onion layers for the whole graph only once. According to the property of $\mathcal{B}_{bd}$, the edges in $\mathcal{B}_{bd}$ satisfy that the change of its trussness is no more than one. However, if we insert several star motifs at the same time, the change of trussnesses of the edges in $\mathcal{B}_{bd}$ may be more than one. It is because edges in the $\mathcal{B}_{bd}$ may be included in more than one new triangle. Therefore, to ensure that the trussnesses of the edges in $\mathcal{B}_{bd}$ do not change by more than one, the star motifs should be divided into different batches, where star motifs are *pairwise disjoint* in each batch, i.e., $\cap\mathcal{B}_i = \emptyset$. In other words, two stars S_{c_i} and S_{c_j} are pairwise disjoint *if and only if* they have no common edge in the AffBall regions of S_{c_i} and S_{c_j}. Each time, we insert one batch of star motifs into the graph and then maintain trussnesses and onion layers.

Handle multiple star insertions. Algorithm 8 shows how to maintain trussness τ and onion layer $\mathcal{L}$ when multiple star motifs $\mathbb{S}$ are inserted into the graph G. As different star motifs may overlap, we first employ a greedy algorithm *GreedySplitStars* to find pairwise disjoint star motifs $\mathbb{B}$ (line 2). Specifically, we always choose an unvisited star motif S_i of the largest size (line 11) and find the corresponding $\mathcal{B}$. If $\mathcal{B}$ does not overlap with other $\mathcal{B}_j \in \mathbb{B}$, we add it to $\mathbb{B}$. We repeat this process until all star motifs have been visited. Then, for each $\mathcal{B} \in \mathbb{B}$ (line 4), we use Algorithm 6 to adjust trussness τ and onion layer $\mathcal{L}$ of edges in $\mathcal{B}_{in}$. Then, we get the precise trussness τ and onion layer $\mathcal{L}$ of the whole graph by employing Algorithm 5 (line 8). Algorithm 8 terminates until all star motifs are inserted.

Example 4. *Figure 4 shows an example of inserting two disjoint star motifs into the graph.* $(\mathcal{B}_1)_{in}$ *contains three new edges* (a, f), (b, f) *and* (f, g). $(\mathcal{B}_1)_{bd} = \{(a, b), (b, g)\}$. $(\mathcal{B}_2)_{in}$ *contains three new*

Table 2. Network statistics

Network	$\|V\|$	$\|E\|$	d_{max}	k_{max}
Deezer	41,773	125,826	112	7
Amazon	334,863	925,872	549	7
DBLP	684,911	2,284,991	611	115
Skitter	1,696,414	11,095,298	35,455	68
Patents	6,009,554	16,518,947	793	36
Pokec	1,632,803	22,301,964	14,854	29
LJ	4,847,571	42,851,237	20,333	352
Orkut	3,072,441	117,185,083	33,313	73
Wise	58,655,849	261,321,033	278,489	80

edges (c,d), (d,h) *and* (d,i). $(\mathcal{B}_2)_{bd} = \{(c,h),(h,i)\}$. *Since* $\mathcal{B}_1$ *and* $\mathcal{B}_2$ *are disjoint, we conduct the local truss decomposition on* $\mathcal{B}_1$ *and* $\mathcal{B}_2$*, respectively, and then maintain the trussnesses of edges outside* $\mathcal{B}$.

7 EXPERIMENTS

In this section, we conduct experiments to evaluate our proposed algorithms. The experiments are conducted on a Linux Server with Xeon E5-2630 v4 (2.2 GHz) and 256GB main memory. All algorithms are implemented in C++.[1]

Datasets. We employ nine real-world networks in experiments. Specifically, Deezer is friendship networks of streaming music service users from 3 European countries. DBLP is a collaboration network [1]. Amazon is an Amazon product network. Skitter is an internet topology network. Patents is a U.S. citation network, which includes all citations of patents granted between 1975 and 1999. Pokec, LJ (LiveJournal) and Orkut are online social networks. Wise is a large micro-blogging graph [2]. Except DBLP and Wise, all other networks are downloaded from SNAP [18]. Table 2 summarizes the statistics of all networks.

Competitors. We evaluate four algorithms in experiments.

- Star: is our truss maintenance algorithm for star insertions and deletions.
  Star incorporates two phases of techniques: Phase I is to preprocess AffBall for trussness estimation in Section 4; and Phase II is to use onion layer based method to refine all edges' trussnesses in Section 5. Star handles general updates in Section 6.
- XH [12]: is the trussness maintenance algorithm for single-edge insertion/deletion. For multiple edge insertions/deletions, we insert/delete edges one by one.
- NodePP [8]: is a k-truss discovery algorithm for public-private graphs. Note that NodePP is extened to only support edge insertions in experiments.
- Order [41]: is an order-based truss maintenance algorithm, which is the state-of-the-art truss maintenance algorithm to handle edge insertions/deletions in batch.

It is worth mentioning that [25] splits the inserted subgraph into unrelated edges and inserts them in several batches, which degenerates into single-edge insertion like XH [12]. Thus, we omit the method of [25] in our experiments. By default, we generate 100 star motifs for insertions/deletions on each network. For each star motif, we randomly choose a center node $c \in V$ and select its neighbors with a probability of 0.5 to form the star motif. We delete a star motif from the network to test the truss maintenance time for edge deletions, and then insert the star motif back to test the truss maintenance time for edge insertions. We repeat each experiment 100 times and report the average results.

[1] https://github.com/jinrdfh/TrussMaintenance

Table 3. Efficiency and indexing evaluation of all truss maintenance algorithms. The best results are in bold.

	Star Insertions (ms)				Star Deletions (ms)		
Networks	XH	NodePP	Order	Star	XH	Order	Star
Deezer	0.814	4.88	1.17	**0.026**	0.278	**0.006**	0.026
Amazon	6.00	38.2	8.49	**0.038**	1.70	**0.013**	0.049
DBLP	10.9	271	25.2	**0.048**	10.0	**0.05**	**0.05**
Skitter	3490	31900	110	**1.13**	34.7	0.84	**0.096**
Patents	84.9	430	130	**0.13**	15.2	0.26	**0.076**
Pokec	1.27×10^4	3.17×10^5	180	**0.43**	25.9	0.96	**0.21**
LJ	2.65×10^4	2.07×10^5	310	**0.51**	34.8	**0.43**	0.51
Orkut	4850	9.97×10^5	200	**6.35**	72.8	**1.00**	3.56
Wise	1.34×10^6	1.43×10^7	2050	**0.49**	330	0.10	**0.027**

	Index Size (MB)				Indexing Time (seconds)			
Networks	XH	NodePP	Order	Star	XH	NodePP	Order	Star
Deezer	**1.7**	**1.7**	2.5	2.5	**0.07**	**0.07**	**0.07**	**0.07**
Amazon	**14**	**14**	18	18	0.97	0.97	**0.69**	1.05
DBLP	**34**	**34**	44	44	2.64	2.64	**1.93**	2.50
Skitter	**165**	**165**	212	212	58	58	56	**53**
Patents	**284**	**284**	316	316	20	20	**16**	21
Pokec	**338**	**338**	426	426	45	45	**42**	46
LJ	**551**	**551**	662	662	81	81	**74**	76
Orkut	**1165**	**1165**	1387	1387	314	314	323	**311**
Wise	**4931**	**4931**	4985	4985	2820	2820	**2369**	2460

Exp-I: Efficiency evaluation. We first evaluate the efficiency of our proposed algorithm and competitors through inserting/deleting star motifs. Table 3 reports the empirical results, including truss maintenance time for edge insertions/deletions, index size, and indexing time. We have the following observations.

For star insertions, Star achieves the best performance over all networks; Order is worse than Star; NodePP has the worst performance. It is because (1) for Star, the technique of AffBall efficiently handles edges whose trussnesses change more than one and the technique of onion layer successfully reduces the number of candidate edges; (2) Order needs to maintain the peeling order in truss decomposition ; (3) NodePP first deletes the center node and then inserts it back, which incurs a huge amount of calculations. Overall, Star is orders of magnitude faster than the state-of-the-art algorithm Order on all networks. In particular, Star can achieve about 4, 000**x** speedup on the largest dataset Wise. For the star deletion, Star performs best on some datasets, e.g., Skitter, Patents, and Wise, and Order performs best on other datasets. It is because, for Order and Star, the trussness maintenance problem for edge deletions is bounded by $AFF^{\leq}$ and L_CHG, respectively. L_CHG could be bigger or smaller than $AFF^{\leq}$. Thus, Star and Order perform best on different networks. On the whole, comparing the star insertion with the star deletion in Table 3, we can observe that edge insertions run far more slower than edge deletions for Order. However, Star not only improves edge insertions significantly, but also has comparable performance for edge deletions compared with Order, demonstrating the efficiency of Star.

For the index of all algorithms, XH and NodePP have the same index size and indexing time since they both store the trussnesses of all edges, which can be computed by truss decomposition. Moreover, Star and Order have the similar index sizes, which are larger than that of XH and NodePP. The reason behind is that, except the trussnesses of all edges, Star stores onion layers and SeSup

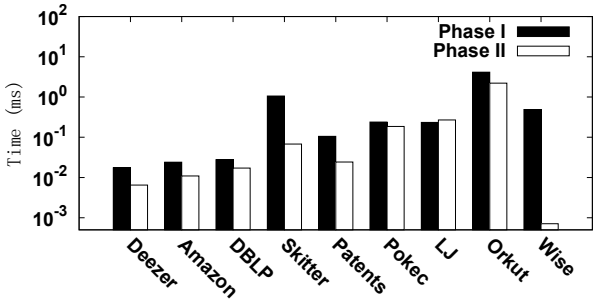
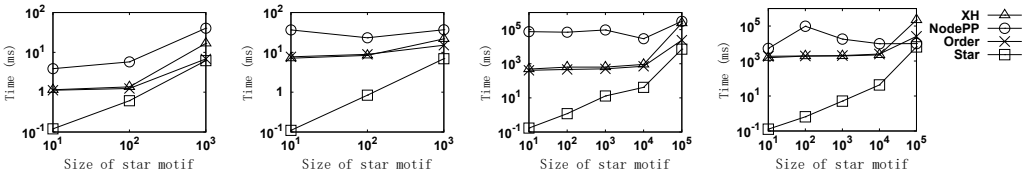


Fig. 5. Evaluation of two phases of Star

Table 4. The evaluation of the number of examined edges for Star

	Deezer	Amazon	DBLP	Skitter	Patents	Pokec	LJ	Orkut	Wise
$\|E\|$	1.3×10^5	9.3×10^5	2.3×10^6	1.1×10^7	1.7×10^7	2.2×10^7	4.2×10^7	1.2×10^8	2.6×10^8
$\|\mathcal{B}_{in}\|$	32.01	33.44	25.39	481.71	19.14	139.65	110.88	138.87	1562.08
|L_CHG|	34.42	56.0	81.99	2997.35	13.42	309.85	888.8	848.54	3093.87
|T_CHG|	19.74	37.14	71.01	903.54	5.64	92.72	296.53	298.82	1015.83



(a) Deezer (b) Amazon (c) Orkut (d) Wise

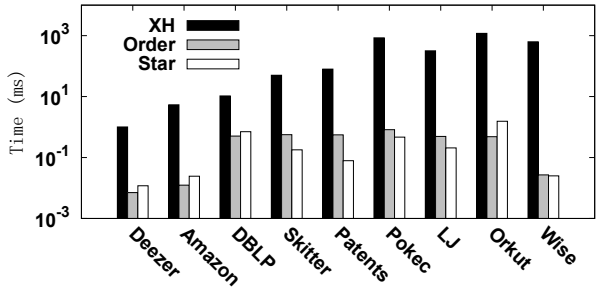
Fig. 6. The effect of star motif size (Insertion)

for all edges, and Order stores supports for all edges. Therefore, Star and Order have larger indexes. As all these auxiliary structure information can be obtained from truss decomposition, the four algorithms have similar indexing times.

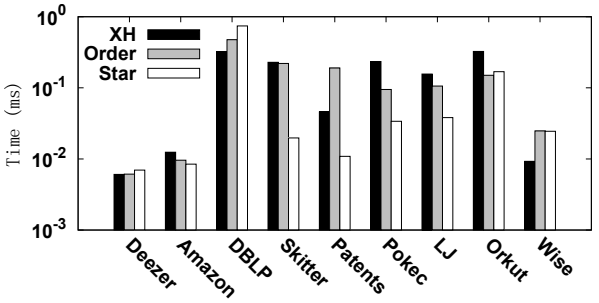
Exp-II: Evaluation of two phases of Star. This experiment tests the efficiency of two phases of Star. Fig. 5 shows the empirical results over all networks. We can observe that Phase I takes more time than Phase II over all datasets except LJ. Specifically, Phase I employs local truss decomposition to get the lower bound trussnesses of edges in AffBall, which recalculates trussnesses in AffBall. Phase II uses onion layers to maintain trussnesses of edges. In most networks, the number of edges with changed onion layers is small. Thus, Phase II takes less time than Phase I. For LJ, onion layers change a lot, leading to more maintenance time of Phase II.

Exp-III: Evaluation of the number of examined edges in Star. We explore the number of examined edges during the truss maintenance by Star. Table 4 shows the results. Note that the four parameters $|E|$, $|\mathcal{B}_{in}|$, |L_CHG|, and |T_CHG| denote the number of edges in graph G, $|\mathcal{B}_{in}|$, changed onion layers, and changed trussnesses, respectively. $\mathcal{B}_{in}$ and L_CHG are related to the Phase I and Phase II of Star, respectively. We can observe that both $|\mathcal{B}_{in}|$ and |L_CHG| are much smaller than $|E|$, ensuring the efficiency of Star.

Exp-IV: The effect of star motif size. In this experiment, we study the effect of star motif size on algorithms. To this end, we vary the star motif size from 10 to 1000 for two small datasets Deezer and Amazon. For two large datasets Orkut and Wise, the star motif size varies from 10 to 100, 000, which is larger than the maximum degree in Orkut. The maintenance time of star insertions is



(a) Single-edge insertion



(b) Single-edge deletion

Fig. 7. Evaluation of single-edge insertion/deletion

shown in Fig. 6. We can observe that except NodePP in Fig. 6, the running times of all algorithms increase with the growth of the inserted star motif size. It is because the larger the inserted star motif, the more edges are affected, resulting in more maintenance time. In addition, when the size of the inserted star motif increases, the performance of Star deteriorates faster than other algorithms. It is because the Phase I of Star conducts local truss decomposition on AffBall, whose running time is relevant to the size of inserted stars. When the size of inserted stars increases, more edges outside AffBall change their trussnesses, which increases the running time of Phase II of Star. Nevertheless, Star still has the best performance over all datasets.

Exp-V: Evaluation of single-edge insertion/deletion. This experiment evaluates the efficiency of Star by inserting/deleting a single edge into/from the network. In the experiment, we randomly choose an edge to delete from the graph and then insert it back. We totally choose 100 different edges and report the average time, which is shown in Fig. 7. For the single-edge insertion, as shown in Fig. 7(a), Star outperforms Order on the networks of Skitter, Patents, Pokec, LJ, and Wise. On the other four networks, Order has better performance. This is because, for a single edge, Star does not need to conduct local truss decomposition (i.e., Phase I), and only employs the onion layer technique of Phase II for truss maintenance. Thus, for some networks such as Amazon and DBLP, the efficiency of Star is not as good as the case of star insertions, which is the goal of this paper. Nevertheless, the performance of Star is still similar to that of Order for the single-edge insertion. In Fig. 7(b) of the single-edge deletion, three algorithms have similar performance of maintenance time, as all these three algorithms are bounded.

Exp-VI: The efficiency evaluation of handling multiple star insertions. In this experiment, we test the efficiency of our Star algorithm to handle general updates for multiple overlapping star insertions. We vary the number of star motifs from 10 to 100 and keep an overlapping rate of star

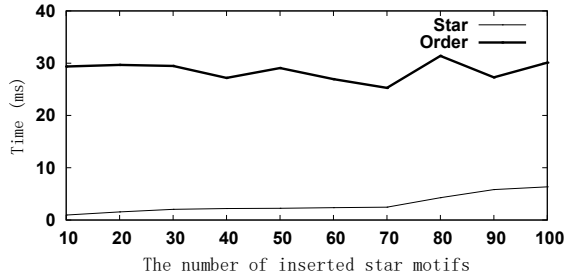
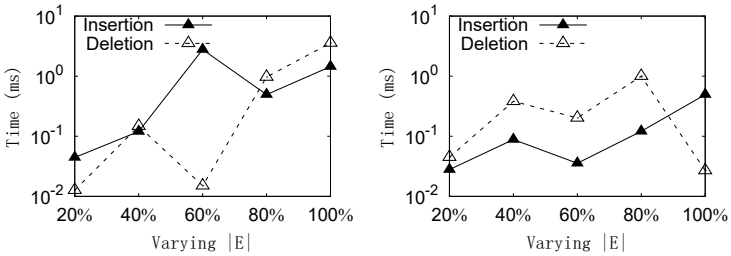


Fig. 8. The efficiency of handling multiple overlapping star motifs on DBLP



(a) Orkut (b) Wise

Fig. 9. Scalability evaluation

insertions as 30%. Fig. 8 shows the maintenance time of Star and Order on the network DBLP. It is obvious that when the number of inserted star motifs increases, the performance of both Star and Order deteriorates. The reason behind is that the more star motifs are inserted, the more edges are affected, leading to more maintenance time. Although our algorithm Star needs to insert pairwise disjoint star motifs in multiple batches, it runs substantially faster than the existing method Order.

Exp-VII: Scalability evaluation. We vary the graph size by randomly selecting 20%, 40%, 60%, 80%, and 100% edges from the graph and evaluate the scalability of Star. Fig. 9 reports the maintenance time for the networks Orkut and Wise. As expected, when the graph size increases, both star insertions and deletions take more time. It is because the larger the graph, the more edges are examined by Star, resulting in longer maintenance time.

Exp-VIII: Case study on a patent citation network. We conduct a case study for truss maintenance algorithms on graph Patents. Specifically, Patents is the U.S. patents citation graph, which are granted from 1975 to 1999. Each patent has a timestamp representing the publication time. Patents contains about 1900 different timestamps. In experiments, we use graph induced by the patent of first 1800 timestamps (i.e., from 1975 to 1997) as original graph G and insert the patents of the next 100 timestamps (i.e., from 1997 to 1999) into G. Thus, we perform 100 rounds of edge insertions in total. In each round, we have $2,524$ star insertions with $27,211$ edges on average, which has an overlapping rate of 11%. Fig. 10 shows the maintenance time of Star, Order, and XH. We can observe that Star consistently outperforms Order and XH for all timestamps, demonstrating the efficiency of our star-based trussness maintenance algorithms to handle real general updates.

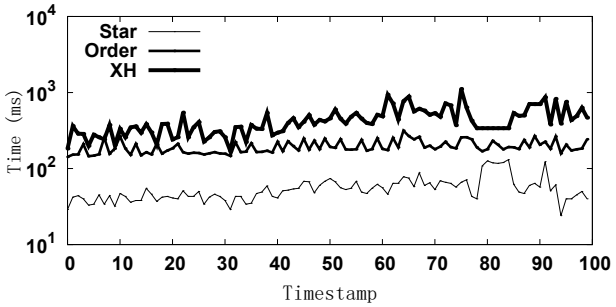


Fig. 10. Case study on Patents citation network

8 CONCLUSION

This paper studies the problem of truss maintenance on dynamic graphs where edges with a common node are inserted into or removed from the graph. We propose an AffBall-based truss maintenance framework for star insertions/deletions. It makes use of two techniques: AffBall-based local truss estimation and onion layers based truss refinement, which are efficient and bounded by the number of affected edges with onion layer changed. Our proposed techniques can be applied to handle general graph updates. Experiments demonstrate our star-based solutions run orders of magnitude faster than state-of-the-art algorithms in handling star insertions on large datasets. This work opens up several interesting questions, e.g., how to maintain trusses on timestamped edges in temporal graphs where the edges are updated with different timestamps.

ACKNOWLEDGMENTS

The work is supported by Hong Kong RGC Grant Nos. 22200320, 12200021, C2004-21GF, 12201520, and GDNSF 2019B1515130001. Dr Simon Wang has helped improve the linguistic presentation of this manuscript. Xin Huang is the corresponding author.

REFERENCES

[1] http://dblp.uni-trier.de.
[2] http://www.wise2012.cs.ucy.ac.cy/challenge.html.
[3] Y. Che, Z. Lai, S. Sun, Y. Wang, and Q. Luo. 2020. Accelerating Truss Decomposition on Heterogeneous Processors. *PVLDB* 13, 10 (2020), 1751–1764.
[4] Chen Chen, Mengqi Zhang, Renjie Sun, Xiaoyang Wang, Weijie Zhu, and Xun Wang. 2022. Locating pivotal connections: the K-Truss minimization and maximization problems. *WWW* (2022), 899–926.
[5] Lu Chen, Chengfei Liu, Rui Zhou, Jianxin Li, Xiaochun Yang, and Bin Wang. 2018. Maximum Co-located Community Search in Large Scale Social Networks. *Proc. VLDB Endow.* 11, 10 (2018), 1233–1246.
[6] Pei-Ling Chen, C.K. Chou, and Ming-Syan Chen. 2014. Distributed algorithms for k-truss decomposition. In *International Conference on Big Data*. 471–480.
[7] J. Cohen. 2008. *Trusses: Cohesive Subgraphs for Social Network Analysis.* Technical Report. National Security Agency.
[8] Soroush Ebadian and Xin Huang. 2019. Fast algorithm for K-truss discovery on public-private graphs. In *IJCAI*. 2258–2264.
[9] Fatemeh Esfahani, Mahsa Daneshmand, Venkatesh Srinivasan, Alex Thomo, and Kui Wu. 2021. Truss Decomposition on Large Probabilistic Networks using H-Index. In *SSDBM*. 145–156.
[10] Fatemeh Esfahani, Jian Wu, V. Srinivasan, A. Thomo, and K. Wu. 2019. Fast Truss Decomposition in Large-scale Probabilistic Graphs. In *EDBT*. 722–725.
[11] Hongxuan Huang, Qingyuan Linghu, Fan Zhang, Dian Ouyang, and Shiyu Yang. 2021. Truss Decomposition on Multilayer Graphs. In *Big Data*. 5912–5915.
[12] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-truss community in large and dynamic graphs. In *SIGMOD*. 1311–1322.
[13] Xin Huang and Laks VS Lakshmanan. 2017. Attribute-driven community search. *PVLDB* 10, 9 (2017), 949–960.

[14] Xin Huang, Laks V. S. Lakshmanan, Jeffrey Xu Yu, and Hong Cheng. 2015. Approximate Closest Community Search in Networks. *Proc. VLDB Endow.* 9, 4 (2015), 276–287.

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

[16] Yuli Jiang, Xin Huang, and Hong Cheng. 2021. I/O efficient k-truss community search in massive graphs. *VLDB J.* 30, 5 (2021), 713–738.

[17] H. Kabir and K. Madduri. 2017. Shared-Memory Graph Truss Decomposition. In *HiPC*. 13–22.

[18] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[19] Rong-Hua Li, Jeffrey Xu Yu, and Rui Mao. 2014. Efficient Core Maintenance in Large Dynamic Graphs. *IEEE Trans. Knowl. Data Eng.* 26, 10 (2014), 2453–2465.

[20] Yanting Li, Tetsuji Kuboyama, and Hiroshi Sakamoto. 2013. Truss Decomposition for Extracting Communities in Bipartite Graph. In *IMMM*. 76–80.

[21] Boge Liu, Fan Zhang, Wenjie Zhang, Xuemin Lin, and Ying Zhang. 2021. Efficient Community Search with Size Constraint. In *ICDE*. 97–108.

[22] Qing Liu, Minjun Zhao, Xin Huang, Jianliang Xu, and Yunjun Gao. 2020. Truss-based Community Search over Large Directed Graphs. In *SIGMOD*. 2183–2197.

[23] Qing Liu, Xuliang Zhu, Xin Huang, and Jianliang Xu. 2021. Local Algorithms for Distance-generalized Core Decomposition over Large Dynamic Graphs. *VLDB* (2021), 1531–1543.

[24] Qing Liu, Yifan Zhu, Minjun Zhao, Xin Huang, Jianliang Xu, and Yunjun Gao. 2020. VAC: Vertex-Centric Attributed Community Search. In *ICDE*. 937–948.

[25] Qi Luo, Dongxiao Yu, Xiuzhen Cheng, Zhipeng Cai, Jiguo Yu, and Weifeng Lv. 2020. Batch Processing for Truss Maintenance in Large Dynamic Graphs. *IEEE Trans. Comput.* (2020), 1435–1446.

[26] Robert J Mokken. 1979. Cliques, clubs and clans. *Quality & Quantity* 13, 2 (1979), 161–173.

[27] Jian Pei, Daxin Jiang, and Aidong Zhang. 2005. Mining Cross-Graph Quasi-Cliques in Gene Expression and Protein Interaction Data. In *ICDE*. 353–354.

[28] Giulia Preti, Gianmarco De Francisci Morales, and Francesco Bonchi. 2021. STruD: Truss Decomposition of Simplicial Complexes. In *WWW*, Jure Leskovec, Marko Grobelnik, Marc Najork, Jie Tang, and Leila Zia (Eds.). 3408–3418.

[29] G. Ramalingam and Thomas W. Reps. 1996. On the Computational Complexity of Dynamic Graph Problems. *Theor. Comput. Sci.* (1996), 233–277.

[30] Ahmet Erdem Sariyüce, Bugra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V. Çatalyürek. 2013. Streaming Algorithms for k-core Decomposition. *VLDB* (2013), 433–444.

[31] Ahmet Erdem Sariyüce, Bugra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V. Çatalyürek. 2016. Incremental k-core decomposition: algorithms and evaluation. *VLDB J.* (2016), 425–447.

[32] Ahmet Erdem Sariyüce, C. Seshadhri, and Ali Pinar. 2018. Local Algorithms for Hierarchical Dense Subgraph Discovery. *Proc. VLDB Endow.* 12, 1 (2018), 43–56.

[33] Stephen B Seidman and Brian L Foster. 1978. A graph-theoretic generalization of the clique concept*. *Journal of Mathematical sociology* 6, 1 (1978), 139–154.

[34] Renjie Sun, Yanping Wu, and Xiaoyang Wang. 2022. Diversified Top-r Community Search in Geo-Social Network: A K-Truss Based Model. In *EDBT*. 2:445–2:448.

[35] Xin Sun, Xin Huang, Zitan Sun, and Di Jin. 2021. Budget-constrained Truss Maximization over Large Graphs: A Component-based Approach. In *CIKM '21*. 1754–1763.

[36] Zitan Sun, Xin Huang, Jianliang Xu, and Francesco Bonchi. 2021. Efficient Probabilistic Truss Indexing on Uncertain Graphs. In *WWW '21*. 354–366.

[37] Jia Wang and James Cheng. 2012. Truss Decomposition in Massive Networks. *PVLDB* 5, 9 (2012), 812–823.

[38] Yanping Wu, Renjie Sun, Chen Chen, Xiaoyang Wang, and Qiuyu Zhu. 2020. Maximum Signed (k, r)-Truss Identification in Signed Networks. In *CIKM*. 3337–3340.

[39] Xiaoqin Xie, Mingjie Song, Chiming Liu, Jiaming Zhang, and Jiahui Li. 2021. Effective influential community search on attributed graph. *Neurocomputing* (2021), 111–125.

[40] ZhiBang Yang, Xiaoxue Li, Xu Zhang, Wensheng Luo, and Kenli Li. 2022. K-truss community most favorites query based on top-t. *World Wide Web* (2022), 949–969.

[41] Yikai Zhang and Jeffrey Xu Yu. 2019. Unboundedness and Efficiency of Truss Maintenance in Evolving Graphs. In *SIGMOD*. 1024–1041.

[42] Yikai Zhang, Jeffrey Xu Yu, Ying Zhang, and Lu Qin. 2017. A Fast Order-Based Approach for Core Maintenance. In *ICDE 2017*. 337–348.

[43] Feng Zhao and Anthony KH Tung. 2012. Large scale cohesive subgraphs discovery for social network visual analysis. *PVLDB* 6, 2 (2012), 85–96.

[44] Jun Zhao, Renjie Sun, Qiuyu Zhu, Xiaoyang Wang, and Chen Chen. 2020. Community Identification in Signed Networks: A K-Truss Based Model. In *CIKM*. 2321–2324.
[45] Zibin Zheng, Fanghua Ye, Rong-Hua Li, Guohui Ling, and Tan Jin. 2017. Finding weighted k-truss communities in large networks. *Inf. Sci.* 417 (2017), 344–360.
[46] Weijie Zhu, Mengqi Zhang, Chen Chen, Xiaoyang Wang, Fan Zhang, and Xuemin Lin. 2019. Pivotal Relationship Identification: The K-Truss Minimization Problem. In *IJCAI*. 4874–4880.
[47] Zhaonian Zou and Rong Zhu. 2017. Truss decomposition of uncertain graphs. *Knowledge and Information Systems* 50, 1 (2017), 197–230.

Received October 2022; revised January 2023; accepted February 2023