Adaptive Truss Maximization on Large Graphs: A Minimum Cut Approach

Zitan Sun, Xin Huang, Chengzhi Piao
Hong Kong Baptist University
Hong Kong, China
{zitansun, xinhuang, czpiao}@comp.hkbu.edu.hk

Cheng Long
Nanyang Technological University
Singapore
c.long@ntu.edu.sg

Jianliang Xu
Hong Kong Baptist University
Hong Kong, China
xujl@comp.hkbu.edu.hk

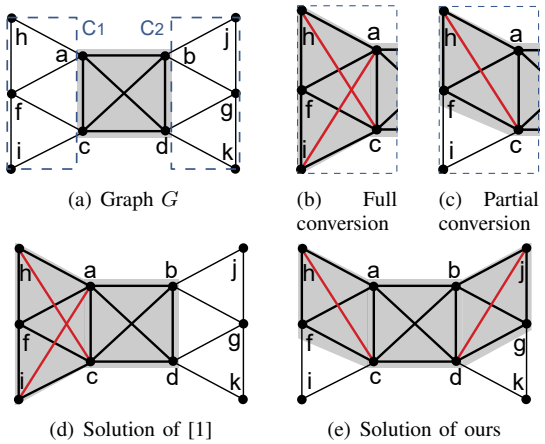
***Abstract*—A cohesive subgraph of k-truss requires that each edge has at least $(k-2)$ triangles, which has wide applications of modeling social communities and complex network visualization. Recently, the study of truss maximization has gained attention, which aims to enlarge k-truss most by inserting b new edges into a graph G. However, existing maximization methods suffer from a stiff strategy of complete truss conversion, that is either *converting the whole* $(k-1)$*-truss component to* k*-truss* or *converting no edge to* k*-truss without using any budget*. To tackle this bottleneck, we develop a novel partial conversion strategy to explore more insertion plans.**

Based on partial conversion strategy, we revisit the problem of truss maximization in this paper and propose adaptive solutions by achieving more new k-truss edges. Specifically, we first decompose all $(k-1)$-truss into a series of disjoint components via the triangle connectivity, where each component's conversion is independent to each other. Then, for each $(k-1)$-truss component, we explore possible insertion plans of *partial conversions*. An intuitive method is to randomly insert a budget no more than b new edges and check the expected profit of new k-truss edges. Obviously, this method is inefficient due to a large search space of edge insertions and many times of expensive k-truss verification. To improve it, we propose a new minimum-cut based approach, which converts a subgraph of $(k-1)$-truss component into a flow graph with weighted edges and finds a key of maximum-flow answer corresponding to a k-truss conversion plan with the minimum budget consumption. Next, we develop a new dynamic programming framework to find the best way to allocate the budget b to all components. We design two fast dynamic programming algorithms and analyze the complexities theoretically. In addition, we explore the case of a large given budget b and extend our techniques to handle the conversion of $(k-h)$-truss into k-truss for $2 \le h \le k-2$. Extensive experiment results demonstrate the superiority of our algorithms against the state-of-the-art methods.

***Index Terms*—k-truss, graph enhancement, minimum cut, maximization, dynamic**

I. Introduction

Graph is a fundamental model to represent various entities and their complex relationships in many real applications. For example, a social network can be modeled as a graph, where the nodes represent users and the edges are their diverse relationships, e.g., friendships, followers/followees, posting/replying comments, and so on. Other real graph applications include the traffic networks, communication networks, financial networks, and biological networks. In the graph theory, k-truss denotes a dense subgraph where every edge is contained in at least $k-2$ triangles in this subgraph. The k-truss structure enjoys several nice properties, such as the high density, strong $(k-1)$-edge connectivity, and polynomial-time computations. The k-truss is widely used to model closely connected community in social networks [2]–[8].



(a) Graph G (b) Full conversion (c) Partial conversion

(d) Solution of [1] (e) Solution of ours

Fig. 1. An example of graph G with the truss number $k=4$ and the budget $b=2$. The edges in grey areas are 4-truss edges. Two components C_1 and C_2 are shown in dashed box in Fig. 1(a). Moreover, Fig. 1(b) and Fig. 1(c) show two plans of *complete conversion* and *partial conversion* in C_1, respectively. The updated graph in Fig. 1(d) is inserted by two new edges $\{(h,c),(a,i)\}$, which achieves 8 new 4-truss edges [1]. Our solution inserts $\{(h,c),(d,j)\}$ to achieve a better answer of 10 new 4-truss edges as shown in Fig. 1(e).

Motivations and applications. In this paper, we revisit and study the problem of k-truss maximization, which aims to enlarge the k-truss of the graph by adding no more than b edges. As a similar concept of k-truss, k-core requires that each vertex has at least k neighbors within this subgraph. A k-truss is also a $(k-1)$-core. As shown in recent studies [1], [9]–[13], [44], truss/core maximizations focus on strengthening the size of k-truss/k-core for connectivity improvement by inserting new edges, which have many real applications including the identification of missing defense links in military networks [1], improving transportation network connectivity [9], boosting the stability of P2P networks [10], [11], and enhancing social group engagement [12], [13]. For instance, the k-truss of flight networks has $(k-1)$-edge connectivity, reflecting even if any less than $(k-1)$ edges are disconnected, the whole k-truss component keeps connected. Thus, a large k-truss of flight

networks ensures the strong connectivity of airline routes. Note that different from the existing truss maximization [1], our work investigates a wide range of budget b for insertion cases, where b ranges from a tinny small one to an extremely large one. Our study provides a comprehensive solution set for real application scenarios, especially when there exists a limited number of budget on edge insertions, e.g., in the economic consideration of coupon promotions by inviting friends to participate in activities on social networks, and also adding new routes for improving connectivity in flight networks.

Challenges. Compared with k-core, k-truss is conceptually more rigorous by requiring the minimum constraint on 3-cliques as triangles, instead of simple node degrees. The task of k-truss maximization is more technically challenging than the k-core maximization [9], which lies on three-fold aspects. First, an incremental number of targeted candidate edges. The truss maximization enlarges k-truss to convert candidate edges by providing more *feasible* triangles by inserting edges. However, the newly inserted edges may still has no enough support of triangles to be retained in k-truss, thus they also become those target candidates to be new k-truss edge. This leads to an incremental number of targeted candidates in search space. Second, the constraint of k-truss is more restrictive than k-core. The inserted edges need to be kept in k-truss for providing feasible support, otherwise they are removed during truss decomposition. Thus, an edge may not be converted into k-truss by adding $2(k-2)$ edges, because newly added edges may not be in k-truss. On the contrary, the solution of core maximization [9] adds a new edge between two candidate $(k-1)$-shell nodes to certainly increase their degrees. Third, the verification of new k-truss edges by invoking truss decomposition is more time costly than that of new k-core nodes by the core decomposition.

The truss maximization problem has been shown to be NP-hard [1]. A recent work of CBTM [1] partitions the $(k-1)$-truss into many smaller components, where each component is a connected subgraph and does not intersect with others. It then calculates the score (the increased size of new k-truss) and the cost (budget consumed) of converting each component into k-truss, respectively. Next, it uses dynamic programming (DP) to find the optimal combination of insertion plans in different components, according to the total budget. This solution has been shown to be effective with a full conversion strategy, but still suffers from two drawbacks as follows. First of all, some components may be too large to be converted within the budget b. Or some edges of a component may need lots of budgets, lowering the average conversion rate. Secondly, the $(k-1)$-truss may be a small part of the graph, leading to a bad performance when the budget b or parameter k is large. This means that it may not find more insertion plans when the $(k-1)$-truss has been fully converted to k-truss.

To address the above challenges and limitations, we propose a novel framework PCFR with several new techniques for truss maximization. Specifically, our PCFR framework consists of three new parts. First, we develop the truss-based partial conversion to explore more insertion plans. Second, we design a new dynamic programming to consider multiple insertion choices, but not binary one any more. Third, we propose a new strategy to convert $(k-h)$-truss into k-truss to handle a large budget b. We illustrate the advantage of our proposed PCFR method against the competitor CBTM [1].

Example 1. *Fig. 1 shows an example of truss maximization in graph G. The whole graph G is the 3-truss, because every edge is contained in at least one triangle. The target is to have more edges become 4-truss, with the budget no more than 2. There are two symmetrical components in G, C_1 (edges $\{(a,h),(f,h),(a,f),(c,f),(c,i),(f,i)\}$) and C_2 (edges $\{(b,j),(g,j),(b,g),(d,g),(d,k),(g,k)\}$). Take C_1 as an example. It can be completely converted to 4-truss by inserting edges $\{(h,c),(a,i)\}$ and gets 8 new 4-truss edges, as shown in Fig. 1 (b). Alternatively, it can be partially converted to 4-truss by inserting one edge (c,h) and gets 5 new 4-truss edges, as shown in Fig. 1 (c). [1] only adopts the complete conversion strategy and gets the result of Fig. 1 (d), where the budget used and the number of new 4-truss edges are $(2,8)$. Our solution considers both complete conversion and partial conversion and gets the result of $(2,10)$, as shown in Fig. 1 (e), which is better than that of [1].*

We first introduce the partial conversion. We intend to address the challenge of how to partially convert a component to k-truss. An immediate idea is to randomly insert edges and record edges that are in k-truss. It is effective on $(k-1)$-truss but not effective on $(k-h)$-truss when $h>1$. Therefore, we convert a component to a directed acyclic graph (DAG) that can show the hierarchical structure of the component. We then convert the DAG to a flow graph and find the minimum cut to get a conversion plan. There is a parameter that can change the structure of the flow graph, so there are many different minimum cuts and many conversion plans with different budgets by adjusting the parameter.

Next, we present new dynamic programming algorithms, which find the best combination of multiple partial conversion choices. We build a table with size of $|C| \times b$, where $|C|$ is the number of components. The value of position (i,j) means the highest score of previous i components within budget j, which is obtained by comparing at most b conversion plans of the ith component, so the time complexity of this algorithm is $O(|C|b^2)$. We also propose an approximate method that sorts all conversion plans first in a decreasing order, whose time complexity can be $O(|C|b)$ when $|C| \gg b$.

Third, we develop advanced techniques of $(k-h)$-truss conversion. We extend techniques mentioned above to handle $(k-h)$-truss, so we can get the structure of $(k-h)$-truss and find potential areas for conversion. However, converting edges in $(k-h)$-truss is more difficult than converting edges in $(k-1)$-truss. So our idea is to convert them into a k-clique (k nodes connected to each other), because a k-clique is also a k-truss, which can ensure the successful conversion. The cost for converting $(k-h)$-truss is larger, as h increases. Therefore, we first convert the $(k-1)$-truss, and the superfluous budget

is used to convert $(k-h)$-truss, as h increases. To summarize, we make the following contributions in this paper.

- We propose new methods to partially convert a component to k-truss. The idea is to peel a component into different subgraphs and convert them to many flow graphs with edge weights. For all generated flow graphs, we conduct the minimum cut to find the optimal answers, corresponding to the insertion plans with small budgets. (Section IV)
- We propose a new dynamic programming framework to handle a component with multiple insertion plans for partial conversions. We further optimize the algorithm complexity of our dynamic programming methods by distinguishing important parameters. (Section V)
- We extend the above approaches to handle $(k-h)$-truss conversion for $h \geq 2$, ensuring that our solution can find insertion plans for an extremely large budget b. (Section VI)
- Extensive experiments on nine real-world datasets validate the effectiveness and efficiency of our algorithms. (Section VII)

We discuss related work in Section II and conclude the paper in Section VIII.

II. Related Work

We study related work including the *truss mining*, *dense subgraph maintenance*, and *network structure enhancement*.

Truss mining. We summarize the studies of k-truss mining in terms of two categories. The first one is accelerating the computation of k-truss under various settings; and the second one is generalizing the concept of k-truss on complex graphs. Specifically, many works speed up the computation of k-truss with parallel computing [14], cloud computing [15], GPU [16], FPGA [17]. The concept of k-truss has also been generalized on various graphs, including directed graphs [7], uncertain graphs [18]–[22], signed graphs [23], [24], attribute graphs [25]–[27], dynamic graphs [3], [28]–[31], geo-social graphs [32], [33], bipartite graphs [34], weighted graphs [35], multilayer graphs [36], and simplicial complexes [37].

Dense subgraph maintenance. The task of dense subgraph maintenance aims to update a particular subgraph pattern in dynamic graphs, including k-core maintenance [38]–[42] and k-truss maintenance [3], [29]–[31]. These works consider maintaining dense subgraph structures when the graph changes, such as edges insertion or deletion. However, the graph change is not known in advance and cannot be controlled by their maintenance algorithms. Different from these, our problem of truss maximization can select new edges to be inserted for truss maintenance, which aims at enlarging the k-truss most.

Network structure enhancement. In the literature, there exist several studies on network structure enhancement, such as reachability enhancement, core maximization, and truss maximization by inserting edges or anchoring nodes. The reachability can be enhanced by inserting new edges [43]. [10], [11] enlarge k-core by adding edges to those nodes with low degree. The problem of k-core/k-truss anchoring [12], [44] enlarges the k-core/k-truss by anchoring a few nodes that will not be peeled in the core/truss decomposition. Although the goal of [44] and ours is maximizing k-truss, [44] uses a greedy algorithm to anchor nodes but we adopt DP-based strategies for selecting new edge insertions. The most related works to ours are [1] and [9], which partition $(k-1)$-truss/$(k-h)$-core into small components and convert these components to k-truss/k-core by adding edges. Then, both work [1] [9] use different dynamic programming to find the optimal combination. Our solution follows their framework. However, we proposes two methods that can partially convert a component to k-truss, while [1] only considers the complete conversion. Therefore, we can use small budgets to get a better answer of feasible conversions. In addition, our new dynamic programming framework can support multiple conversion plans for a component. What's more, we propose a new algorithm that can convert $(k-h)$-truss to k-truss when $h > 1$, which is more challenging than converting $(k-h)$-core to k-core [9]. Therefore, we can convert more edges into k-truss when the given budget b is large.

III. Preliminaries

Given a graph $G = (V, E)$ where V is the node set and E is the edge set, we represent $H = (V_H, E_H)$ as a subgraph of G with $V_H \subseteq V$ and $E_H \subseteq E$. For a node $u \in V_H$, $N_H(u)$ is the node set containing all neighbors of node u in H, i.e., $N_H(u) = \{v \in V_H | (u, v) \in E_H\}$. The support number of an edge $(u, v) \in E_H$ is the number of triangles containing this edge in the subgraph H, i.e., $\mathsf{sup}_H((u, v)) = |N_H(u) \cap N_H(v)|$. Based on these concepts, the formal definition of k-truss is given as follows.

Definition 1 (K-Truss [45]). *A subgraph $H = (V_H, E_H)$ is the k-truss if H is the largest subgraph of G such that all edges have support numbers no less than $k-2$ in H, i.e., $\forall (u, v) \in E_H$, $\mathsf{sup}_H((u, v)) \geq k-2$.*

Next, we give a useful definition of trussness, which can be stored into an index to enable fast query of the k-truss.

Definition 2 (Trussness). *The trussness $\tau(e)$ of an edge e in the graph G is defined as the largest number k such that there exists a non-empty k-truss containing e.*

The k-truss has a hierarchical structure, i.e., k-truss $\subseteq (k-1)$-truss. For instance, the edge e must be in $\tau(e)$-truss and $(\tau(e)-1)$-truss. In addition, we use T_k to represent the edge set of k-truss. We denote all edges with the same trussness k as the k-class E_k, i.e., $E_k = \{e \in E | \tau(e) = k\}$. In other words, the edge set of k-truss can be represented as $T_k = \bigcup_{i=k}^{k_{\max}} E_i$, where $k_{\max}$ is the largest number k that there exists a non-empty k-truss in G.

Problem (Truss Maximization). *Given a graph G, a budget $b \in \mathbb{Z}^+$, and the truss number $k \geq 2$, the truss maximization problem is to insert at most b new edges into G such that the edge size of k-truss in the new graph is the largest.*

Algorithm 1 Random

Input: graph G, component c, budget b, repeating times r.
Output: exp-revenue S_c and new edges map P_c.
1: Initialization: $S_c, P_c, Pool \leftarrow \emptyset$;
2: **for all** $(x, y) \in E_c$, $(x, z) \in E$, $(y, z) \notin E$ **do**
3: Add (y, z) to $Pool$;
4: **for all** $i \in [1, r]$ **do**
5: Take an integer b_r at random from $[1, b]$;
6: Take b_r edges C_r at random from $Pool$;
7: Try to insert C_r into G and get the exp-revenue pair (P_r, v_r), where $P_r \subseteq C_r$ is the edges that converted to k-truss and v_r the number of edges in E_c and P_r that converted to k-truss;
8: **if** $v_r > S_c[|P_r|]$ **then**
9: $S_c[|P_r|] \leftarrow v_r$, $P_c[|P_r|] \leftarrow P_r$;
10: Remove useless values in S_c to keep S_c strictly increasing;
11: **return** S_c, P_c;

IV. Interpolation

We first introduce our two-phase framework for solving the truss maximization problem. We introduce Phase-I techniques in this section and Phase-II techniques in the next section.

A. Overview

Our framework includes Phase-I of converting each component to k-truss separately and Phase-II of finding the best budget allocation plan for each component.

Phase-I: Component Interpolation. For each component, we find as many insertion plans as possible, with different budgets used. We use the function S_c to represent the relationship between the budget and the score for the component c, which is strictly increasing. Therefore, this phase can be seen as interpolating S_c as much as possible and with high scores. We propose two algorithms to interpolate S_c. The first one is to randomly insert edges and calculate edges that become k-truss edges (Section IV-B). The second one is to convert a component to a flow graph and conduct minimum cut to find exp-revenue pairs with small budgets (Section IV-C).

Phase-II: Dynamic Programming. According to S_c of all components, we can use DP to find the optimal combination to allocate the budget to each component and obtain the highest total score. We propose two algorithms with different time complexity. Section V-B introduces a dynamic programming algorithm to solve our problem. Section V-D introduces another algorithm that compresses S_c to accelerate the computation. This algorithm has a significant time improvement when there are a large number of components.

B. Random Interpolation

First of all, we give some useful definitions.

Definition 3 (Truss Connectivity)**.** *For two connected edges $e_1 = (a, b)$ and $e_2 = (b, c)$, if they have the same trussness k and the triangle they form exists in the k-truss, i.e., $k = \tau((a, b)) = \tau((b, c)) \leq \tau((a, c))$, e_1 and e_2 are truss connected, donated as $e_1 \underset{k}{\leftrightarrow} e_2$. Moreover, for two arbitrary edges e_1 and e_2, if there exits an edge e such that $e \underset{k}{\leftrightarrow} e_1$ and $e \underset{k}{\leftrightarrow} e_2$, e_1 and e_2 are also truss connected.*

Motivated by the idea of CBTM [1], we take a component c as a subgraph of the $(k-1)$-class in this work. We require that every edge of c is truss connected to each other. In addition, all edges that are truss connected to edges in c are also in c. Thus, the components do not intersect with each other and the conversion of one component does not affect others. All conversions adopted in [1] are complete conversion.

In the following, we propose a novel definition of exp-revenue insertion candidates, which is very useful for our designed partial conversions. The exp-revenue pair of component c is (P, v), corresponding to a pair of an edge set P and a score v, where P is newly inserted edges and v is the revenue of new k-truss edges. In other words, the edges in P are not in the graph. And v is the number of edges in $E_c \cup P$ that newly become k-truss edges after inserting edges in P. Obviously, $1 \leq |P| \leq b$, indicating a possible plan of partial conversions. Since edges not in the k-truss are peeled in the computation of k-truss, they do not contribute to our solution. As a result, we assume all edges in P can successfully become k-truss edges.

Definition 4 (Exp-revenue Insertion Candidates $\mathcal{S}$)**.** *For a component c, S_c represents the relationship between edge size of exp-revenue pairs and the maximum revenue, i.e., $S_c[x] = \max\{v_i||P_i| = x\}$, for all (P_i, v_i), where $P_i \subseteq V \times V - E$, $0 < |P_i| \leq b$. $\mathcal{S}$ is the set including S_c of all components, i.e., $\mathcal{S} = [S_1, S_2, \cdots, S_{|C|}]$, where $|C|$ is the number of components.*

In practice, if two exp-revenue pairs have the same revenue of new k-truss score v, we prefer to choose a set of smaller edge insertions. Therefore, we can remove some values in S_c to keep a discontinuous but strictly increasing order. Note that throughout the following paper, we call the *exp-revenue insertion candidates* $\mathcal{S}$ as the *exp-revenue* for short.

Example 2. *Given graph G in Fig. 1 (a), there are two components C_1 and C_2 in 3-class. An exp-revenue pair (P, v) of C_1 can be $(\{(c, h)\}, 5)$, which means 5 edges becoming 4-truss edges when inserting (c, h) into G. There are two exp-revenue pairs that can be considered, such as $(\{(c, h), (a, i)\}, 8)$ and $(\{(c, h)\}, 5)$. Thus, S_{C_1} becomes $\{1 : 5, 2 : 8\}$. There are two symmetrical components in 3-class. Therefore, the exp-revenue $\mathcal{S} = [S_{C_1}, S_{C_2}] = [\{1 : 5, 2 : 8\}, \{1 : 5, 2 : 8\}]$.*

Next, we introduce our random algorithm to find insertion plans for a component. Our general idea is randomly inserting at most b edges into the component c and obtaining score $S_c[x]$, where x is the number of inserted edges successfully in the k-truss, $0 < x \leq b$.

Algorithm 1 shows the process of random algorithm. We use r to represent the number of random times in each component. First of all, we find edges that can form triangles with edges in the component and save them as the candidature edge set $Pool$ (lines 2-3), and new edges are chosen from it. Then, we repeat r times (line 4) and in each time, we randomly choose a budget $b_r \in [1, b]$ (line 5). We randomly choose b_r edges from $Pool$ and form new edge set C_r (line 6). We try to insert these edges into G and get the result v_r and P_r (line 7).

The number v_r is the number of edges in the component c that successfully become k-truss edges. P_r is the edge set in C_r that successfully become k-truss edges. Note that the real budget used is $|P_r|$, rather than b_r, where $|P_r| \leq b_r$. Not all edges in C_r can be successfully transformed into k-truss edges. Edges cannot be in k-truss are peeled in the computation of k-truss, thus not inserting them do not affect the result. The result is saved into S_c and P_c (line 9). Finally, we remove some budgets in S_c and P_c, such that S_c is strictly increasing (line 10). Because we prefer to choose an exp-revenue pair that has the same or higher score with a smaller budget.

This method is extremely effective when converting $(k-1)$-truss to k-truss and b is small. However, this method involves lots of insertions, which may have an expensive cost when the graph is large. What's more, this method can hardly find exp-revenue pairs when converting $(k-h)$-truss to k-truss, where $h > 1$. Therefore, we propose another method based on the analysis of graph structure.

C. Interpolation by Minimum Cut

For each component, we want to get as many scores as possible with small budgets. In the process of truss decomposition, due to the collapse of some key edges, many edges may be peeled and cannot become a k-truss. If we can anchor these edges, there would be lots of edges becoming the k-truss and the budget used is much smaller than that of the complete conversion of components.

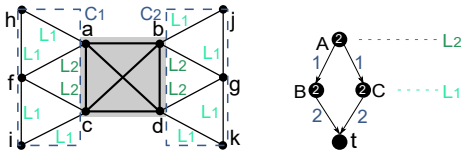
Therefore, our method has the following steps: 1. transform the component to a directed acyclic graph (DAG), which can show its structure; 2. construct a flow graph according to the DAG; 3. conduct the minimum cut to find the most valuable part of the component; and 4. convert these edges to k-truss. We have a parameter that can change the structure of the flow graph in step 2, hence the steps 2-4 can be repeated many times and different exp-revenue pairs are obtained.

Step 1: DAG Construction. First of all, let's introduce the concept of onion layer.

Definition 5 (Onion Layer L)**.** *The onion layer of an edge e is defined as the rounds in which the edge was removed, i.e., $L(e) = \max\{0, l \in N : \sup_H(e) + 2 > k\} + 1$, where $k = \tau(e)$, H is a subgraph that $E_H = T_k - \{e' | \tau(e') = k, L(e') < l\}$. Given edges e_1 and e_2, $e_1 \succ e_2$ means either $\tau(e_1) > \tau(e_2)$ or $\tau(e_1) = \tau(e_2)$, $L(e_1) > L(e_2)$. And $e_1 \succeq e_2$ means either $e_1 \succ e_2$ or $\tau(e_1) = \tau(e_2)$, $L(e_1) = L(e_2)$.*

The onion layer shows the peeling order of edges in the graph. For two edges e_1, e_2, if $e_1 \succ e_2$, it means that e_1 is peeled later than e_2. If we anchor e_2 by inserting some edges, e_1 may also become the k-truss.

In order to further utilize the properties of onion layer, we propose the definition of onion layer connectivity and Block B. Given a triangle $\triangle_{xyz}$, edges (x, y) and (y, z) are onion layer connected if $k = \tau((x, y)) = \tau((y, z))$, $l = L((x, y)) = L((y, z))$ and $(x, z) \succeq (x, y)$, donated as $(x, y) \underset{k,l}{\leftrightarrow} (y, z)$. Moreover, if e_1 e_2 are onion layer connected and e_2 e_3 are also onion layer connected, we define e_1 e_3 to be onion layer connected. We divide a component into many Blocks B and edges in each B are onion layer connected and B is the maximum. Here we use $L(B)$ to represent the onion layer of B. Since we merge edges with the same onion layer into B, two connected Blocks are in different onion layers. If we treat each Block as a vertex and add one directed link between two connected Blocks (from high-level onion layer to low-level onion layer), we can convert a component into a DAG. For two Blocks B_1 and B_2, $L(B_1) > L(B_2)$, $\langle B_1, B_2 \rangle$ represents the directed link from B_1 to B_2 in the DAG. We use $W(B_1, B_2)$ to represent the weight of link $\langle B_1, B_2 \rangle$ in the DAG. Let $Q \subseteq B_1$ be the edge set that edges in Q are connected to B_2. For example, if $e_1 \in B_1$, $e_2 \in B_2$, e_1, e_2 and e_3 are in the same triangle, $e_3 \succeq e_2$, we say $e_1 \in Q$. We define the weight of the link $\langle B_1, B_2 \rangle$ as the size of Q, i.e., $W(B_1, B_2) = |Q|$. The weight of link $\langle B_1, B_2 \rangle$ is also the capacity of the link, which represents the difficulty of converting B_1 into the k-truss without converting B_2. We add a new sink vertex t into the DAG and add a new link between B_i and t if B_i has no out-degree ($d_i = \sum_{j=1}^{|B|} W(B_i, B_j) = 0$, $|B|$ is the number of Blocks), and $W(B_i, t)$ is the size of B_i. By constructing the DAG, we transform the truss maximization problem to a cut problem, i.e., divide the Blocks into two parts. Blocks in the part containing the sink vertex t are discarded and other blocks are converted to k-truss. The real cost (the number of inserted edges) may differ from estimates (the cut), but this method can suggest the part that is easy to convert.



(a) Graph G (b) DAG on C_1

Fig. 2. The graph G and the DAG constructed by the component C_1 of 3-class in G. Green numbers shows the onion layers of edges in 3-class.

Example 3. *Fig. 2 (b) shows an example of a DAG constructed from the component C_1 of 3-class in Fig. 2 (a). The onion layers of edges in 3-class in Fig. 2 (a) are labeled in green numbers. Each vertex in Fig. 2 (b) represents a Block and the white number in it represents the number of edges in this Block. For example, Block $A = \{(a, f), (c, f)\}$ contains two edges (a, f) and (c, f), because they are onion layer connected, and Blocks $B = \{(a, h), (f, h)\}$, $C = \{(c, i), (f, i)\}$. The blue numbers are the weight of links between blocks, which shows the cost of converting a Block. For instance, there is one edge (a, f) in A being connected to edges in B in triangle, thus $W(A, B) = 1$. $W(B, t) = 2$ because Block B has no out-degree and we connect it to the sink vertex t and the weight is the number of edges in B. If we want to convert edges in all blocks to k-truss edges, we can simply cut $\langle B, t \rangle$ and $\langle C, t \rangle$, which needs the cost of 4 and can convert 6 edges.*

We can find the minimum cut of the DAG, but it only has one exp-revenue pair. Our framework can find higher score with more exp-revenue pairs. We need an algorithm to find more valid cut plans from the DAG. Next, we construct many flow graphs, according to the DAG, and then find the minimum

cut to get more exp-revenue pairs.

Step 2: Construction of Flow Graphs. Inspired by [46] that employs a parameter g to generate subgraphs with varying densities, we propose a novel graph construction technique that incorporates multiple min-cuts controlled by the parameter g. First, we introduce a new source vertex s to the DAG in Step 1 and connect it to all Blocks B_i by adding links $\langle s, B_i\rangle$. The weight of each $\langle s, B_i\rangle$ is set as the sum of all link weights in the DAG, denoted as q. Next, we add links $\langle B_i, t\rangle$ from each Block B_i to the target vertex t. The weight $W(B_i, t)$ of each $\langle B_i, t\rangle$ is determined by the expression $\max\{0, g - w_1 L(B_i) - w_2|B_i| - d_i\}$, where g, w_1, and w_2 are three parameters. Importantly, we ensure that $W(B_i, t)$ remains non-negative. With these steps, we successfully construct the flow graph. The parameters w_1 and w_2 are positive values that play a crucial role in adjusting the weight of the onion layer and the size of the Blocks, respectively. Generally, Blocks with a higher onion layer or larger size are more likely to be selected rather than to be discarded. On the other hand, the parameter g is a non-negative number that acts as a gate in our graph construction. Specifically, we add an link $\langle s, B_i\rangle$ from the source vertex s to each Block B_i, and a link $\langle B_i, t\rangle$ from each Block B_i to the target vertex t. While the capacity from s to B_i is fixed at q, the capacity from B_i to t is controlled by the parameter g, taking into account the values of w_1 and w_2. By adjusting the parameter g, we can effectively control the maximum flow to the sink vertex t, which is also referred to as the minimum cut. This provides us with a mechanism to regulate the flow and optimize graph construction.

Step 3: The Minimum Cut of Flow Graphs. A partition of the flow graph into two sets, S and T, such that $s \in S$ and $t \in T$, determines a s-t cut. The capacity of the cut $c(S, T)$ is the sum of weights of links between S and T. We anchor Blocks in S, i.e., we choose this part of the component and convert them to the k-truss. A minimum capacity cut can provide a conversion exp-revenue pair with a small budget. We use $h_{w_1,w_2}(g)$ to represent the score obtained by a minimum cut with parameter g, i.e., set the value of g and construct a flow graph, then conduct a s-t cut and $h_{w_1,w_2}(g) = \sum_{B_i \in S} |B_i|$.

Lemma 1. *$h_{w_1,w_2}(g)$ is non-negative and decreases with g.*

Proof. $h_{w_1,w_2}(g)$ is the sum of Block size in S. Therefore, it is non-negative. When g increases, for each Block, the inflow capacity remains the same, and the outflow capacity becomes larger, thus the inflow is more likely to be saturated, and the Block is more likely to be partitioned to T. Consequently, the number of blocks in S decreases and $h_{w_1,w_2}(g)$ decreases. □

The maximum value of $h_{w_1,w_2}(g)$ is the number of edges in the component, i.e., all Blocks locate in S, which can be achieved by setting $g = 0$ (no flow to t). The minimum value of $h_{w_1,w_2}(g)$ is 0, i.e., all Blocks are in T, which can be achieved by setting $g = 2q + w_1 L_{\max} + w_2 B_{\max}$ ($W(B_i, t) > W(s, B_i)$), where $L_{\max}$ is the largest onion layer in the component and $B_{\max}$ is the largest Block size. Since $h_{w_1,w_2}(g)$ decreases with g according to Lemma 1, we can binary search g in $[0, 2q + w_1 L_{\max} + w_2 B_{\max}]$ to obtain different cut plans. For each cut plan, we only convert Blocks in S to k-truss. Therefore, a component can have multiple conversion exp-revenue pairs.

Finally, we study how to completely convert edges in S to be in the k-truss.

Step 4: Complete Conversion of a Subgraph. The conversion strategy in [1] has the following drawbacks:

1) If unstable edges (support less than $k-2$) cannot be converted to stable edges (support no less than $k-2$), they are removed and the algorithm need to restart, which may cause loss of scores and long running time.

2) Even unstable edges has been converted to stable edges, they may still cannot be in the k-truss. Because new inserted edges may be connected to other components and when other components cannot be converted to k-truss, this component are also affected.

For the drawback 1, we propose an improved method. For an unstable edge e, we first try to find a new edge that can increase the support number of e, such that e can become a stable edge. If e cannot become the stable edge by just inserting one edge, we use Clique Strategy, i.e., find other $(k-2)$ nodes and convert these k nodes to the k-clique. K-clique is a subgraph composed of k nodes where each two nodes are connected. A k-clique is the smallest k-truss. Thus if we want to convert an edge to be the k-truss, we can find k nodes that contains two endpoints of this unstable edge and add edges between any two nodes. The number of inserted edges may be $O(k^2)$, which is a large number when k is not small. Therefore, we also adopt Greedy Strategy, i.e., insert edges one by one to cover the most edges that are unstable. Finally, we choose an exp-revenue pair with lower budget between these two strategies. The method may cost lots of budget, but it is acceptable because we need to make sure that the edges can be successfully converted to k-truss edges.

To address drawback 2, we propose component-based support number.

Definition 6 (CSup)**.** *Given a graph G, a component c and an integer k, the component based support number of an edge $\hat{e} \in E_c$ is defined as the support number in the subgraph of k-truss and c, i.e., $CSup(\hat{e}) = \mathsf{sup}_H(\hat{e})$, where $E_H = \{e | \tau(e) \geq k$ or $e \in E_c\}$.*

$CSup$ requires the edges involved in the calculation of support number to be in the same component. Otherwise, if other components are not chosen, these edges are still peeled.

Algorithm 2 shows how to convert edges S to k-truss edges. We first compute $CSup$ for edges in S in the subgraph H, where H consists of the k-truss of the graph and S (line 2). If an edge e has $CSup(e) < k-2$, we mark it as an unstable edge (line 3). In line 4, we find all edges that are not in the graph and can form a triangle with edges in S. Some of these edges may increase the $CSup$ of more than one edge in S. Therefore, we greedily select an edge that covers the most

Algorithm 2 Complete Conversion

Input: graph G, the number k, the target edge set S.
Output: the candidate edge set P.
1: Get subgraph H that includes edges in k-truss and S;
2: Compute $CSup$ for edges in S;
3: Label $e \in S$ whose $CSup(e) < k-2$ as unstable edges;
4: Greedily find new edges to make unstable edges stable ($CSup(e) \geq k-2$) and put into P;
5: **while** $\exists e \in S$ with $CSup(e) < k-2$ **do**
6: Use Clique Strategy and Greedy Strategy to make e stable. Choose one with the lowest budget and save it to P;
7: **return** P;

edges that are unstable each time. The selected edges also need to be stable edges and are put into P. However, some edges in S may still be unstable (line 5). For this case, we have two strategies, as mentioned above. We choose the one of the two strategies with the fewest number of inserted edges and save the result in P (line 6).

Example 4. *Fig. 3(a) shows an example of DAG. Fig. 3(b)-(f) show* 5 *different flow graphs constructed from the DAG in Fig. 3(a), with different parameters. The sum of link weights in Fig. 3(a) is* $q = 60$*, thus we add a source vertex* s *and links* $\langle s, B_i \rangle$ *to all Blocks with the capacity* 60*, as shown in Fig. 3(b)-(f) red links. The label on each link is the current flow and the capacity of this link. We also add links* $\langle B_i, t \rangle$ *from each Block to the sink vertex* t *(black links in Fig. 3(b)-(f)). For Fig. 3(b)-(e),* $w_1 = 1$*,* $w_2 = 1$*, thus* $g_{\max} = 2q + w_1 L_{\max} + w_2 B_{\max} = 2 \times 60 + 1 \times 2 + 1 \times 2 = 124$ *and* $g \in [0, 124]$*. We set different* g *for Fig. 3(b)-(e), i.e.,* 0*,* 124*,* 62*,* 55*, and get different scores, i.e.,* 5*,* 0*,* 2*,* 3*, respectively. Specifically, in Fig. 3(e) with* $g = 55$*, for Block* d*, its onion layer is* 1 *and its size is* 1*, thus* $W_1(d,t) = g - w_1 L(B_i) - w_2 |B_i| - d_i = 55 - 1 - 1 - 0 = 53$*. The original weight in Fig. 3(a) is* $W_2(d,t) = 5$*. Therefore, we sum them up and* $W(d,t) = 58$ *in Fig. 3(e). We can find the minimum cut, as the red dash line shown in Fig. 3(e), which costs a budget of* 20 *and achieves the score of* 3 *in Fig. 3(a). Similarly, we can find the minimum cut in Fig. 3(f), which costs a budget of* 25 *and achieves the score of* 4 *in Fig. 3(a). Combining results in Fig. 3(b)-(f)), we have pairs of cost and score in Fig. 3(a) as* $\{(0,0), (15,2), (20,3), (25,4), (30,5)\}$.

V. Multiple budget assignments framework

In previous section, we compute many exp-revenue pairs of a component with different budgets. However, it is difficult to determine the best budget for the component. Therefore, our idea is to take all exp-revenue pairs into consider and let dynamic programming process decide which exp-revenue pair to use. It is worth mentioning that our new strategy provides a framework for solving this problem. For each component, we can use various algorithms to obtain different exp-revenue pairs and apply them to our framework.

Next, we introduce our new problem of multiple budgets assignment.

Algorithm 3 Sequential DP

Input: exp-revenue $\mathcal{S} = [S_1, S_2, \cdots, S_{|C|}]$, budget b.
Output: the dynamic programming table DP.
1: Initialization: $DP_{i,j} \leftarrow 0$ for $i \in [0, |C|]$ and $j \in [0, b]$;
2: **for all** $i \in [1, |C|]$ **do**
3: **for all** $j \in [1, b]$ **do**
4: **for all** $u \in [0, j]$ **do**
5: **if** $DP_{i-1,u} + S_i[j-u] > DP_{i,j}$ **then**
6: $DP_{i,j} \leftarrow DP_{i-1,u} + S_i[j-u]$;
7: **return** DP;

A. The problem of multiple budgets assignment

Assume that there are components C, labeled as $1, \cdots, |C|$. For each component $c \in [1, |C|]$, we have the exp-revenue $S_c[:]$, and if the budget used in this component is x and $S_c[x]$ is the score obtained. The total budget for all components is b and we wants to get the highest total score. We assume that all $S_c[:]$, $c \in [1, |C|]$ are known. We also assume that the budget allocation of one component does not affect that of other components. Here is the multiple budgets assignment problem.

Problem 1. *Given the total budget* b *and a series of exp-revenue* $S_c[x_c] \geq 0$*,* $x_c \in \mathbb{N}$*,* $1 \leq c \leq |C|$*, the problem is to find the best allocation plan* $\mathbf{x} = [x_1, \cdots, x_{|C|}]$ *to maximize total score, i.e.,*

$$\mathbf{x} = \arg\max_{\mathbf{x}} \sum_{c=1}^{|C|} S_c[x_c] \quad \textit{subject to} \sum_{c=1}^{|C|} x_c \leq b. \tag{1}$$

The dynamic programming solution mentioned in [1] is to solve the 0-1 backpack problem, which is no longer applicable to this problem. Therefore, we propose a new dynamic programming solution.

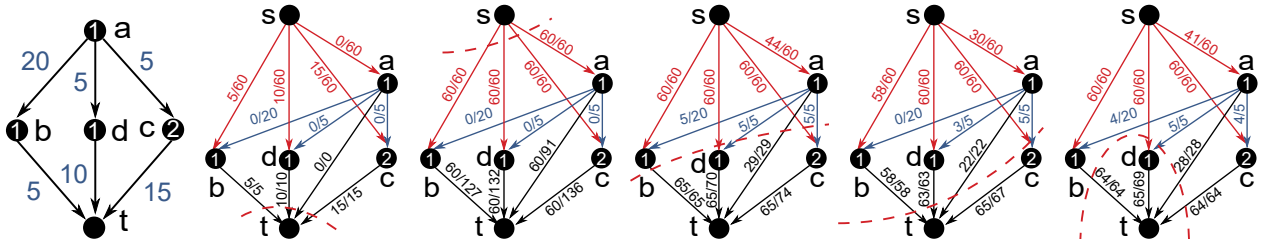
B. Dynamic programming framework of sequential access

First of all, we introduce a table DP with size of $(|C|+1) \times (b+1)$, where the first row and the first column are set to be 0, i.e., $DP_{0,[1:b]} = 0$ and $DP_{[1:|C|],0} = 0$, and other values are defined as follow.

$$DP_{i,j} = \max\{DP_{i-1,u} + S_i[j-u] | u \in [0, j]\}, \quad \text{where } i \in [1, |C|], j \in [1, b] \tag{2}$$

As shown in Equation 2, $DP_{i,j}$ represents the maximum total score only considering the previous i components, with budget used no more than j. For $0 \leq u \leq j$, $S_i[j-u]$ represents the score of component i with the budget $j-u$. Therefore, the value of $DP_{i,j}$ is the largest item by combining the current component i and the results of previous $(i-1)$ components.

Algorithm 3 presents the new dynamic programming process. We need to compute $|C| \times b$ values of the DP table (lines 2-6). For each value $DP_{i,j}$, we iterate over all $DP_{i-1,u}$, $u < j$ (line 4), and find the largest one (line 5). Finally, we return the calculated DP table (line 7).



(a) A DAG (b) $h_{1,1}(0) = 5$ (c) $h_{1,1}(124) = 0$ (d) $h_{1,1}(62) = 2$ (e) $h_{1,1}(55) = 3$ (f) $h_{1,10}(70) = 4$

Fig. 3. An example of DAG (a) and its flow graphs (b)-(f) obtained by changing parameters g, w_1 and w_2. For (b)-(e), $w_1 = 1$ and $w_2 = 1$. For (f), $w_1 = 1$ and $w_2 = 10$. The minimum cut is represented by a red dashed line.

C. Time complexity of Algorithm 3

The external two-layer loops are executed $|C| \times b$ times (lines 2-6). The line 4 repeats at most $b+1$ times. Thus, the time complexity of Algorithm 3 is $O(|C|b^2)$.

However, $O(|C|b^2)$ may be a very large number when $|C|$ is large and b is not very small. In Algorithm 3, every exp-revenue pair of a component is considered. In fact, if an exp-revenue pair has a low conversion ratio, it can be ignored. Therefore, we propose another dynamic programming framework to solve the same problem, but has a different time complexity.

D. Dynamic programming framework with sorted input

Next, we introduce another version of DP. We intend to build a DP table with a size of $(\min\{|C|, b\} + 1) \times (b + 1)$, where the first row and the first column are set to be 0, i.e., $DP_{0,[1:b]} = 0$ and $DP_{[1:\min\{|C|,b\}],0} = 0$, and other values are defined as follow.

$$\begin{aligned} DP_{i,j} = \max\{&DP_{i,j-1}, DP_{i-1,j}, \\ &\{DP_{i-1,u} + S_c[j-u]| \\ &\mathbf{x} = Sol_{i-1,u}, \mathbf{x}[c] = 0, \text{ for } u < j, c \le |C|\}, \\ &\{DP_{i,u} - S_c[\mathbf{x}[c]] + S_c[j-u+\mathbf{x}[c]]| \\ &\mathbf{x} = Sol_{i-1,u}, \mathbf{x}[c] > 0, \text{ for } u < j, c \le |C|\}\}, \\ &\text{where } i \in [1, \min\{|C|, b\}], j \in [1, b] \end{aligned} \tag{3}$$

As shown in Equation 3, $DP_{i,j}$ is defined differently from the one in Equation 2. It represents the maximum total score that the number of chosen components and the used budget do not exceed i and j, respectively. Its value is chosen from the largest of the four terms. If there are more optional components or more budget, the score should be higher, thus the value of $DP_{i,j}$ should not be less than $DP_{i,j-1}$ and $DP_{i-1,j}$. We also need $Sol_{i,j}$ to save the best allocation plan for $DP_{i,j}$. As for the third term $\{DP_{i-1,u} + S_c[j-u]\}$, $\forall u < j, \forall c \le n$, $S_c[j-u]$ represents the score got by choosing component c with the budget $j-u$. If component c has not been chosen by $DP_{i-1,u}$ ($\mathbf{x}[c] = 0$), we can try to add it and compare the score with other values. In the forth term $\{DP_{i,u} - S_c[\mathbf{x}[c]] + S_c[j-u+\mathbf{x}[c]]\}$, we increase the budget without the increase of the number of components. If component c has been chosen by $DP_{i,u}$ and the budget used is $\mathbf{x}[c]$, we can try to use a lager budget $j-u+\mathbf{x}[c]$ and compare the score with other values. The advantage of this method is that we build a $\min\{|C|, b\} \times b$ table, rather than a $|C| \times b$ table. When $b \ll |C|$, this method runs much faster than Algorithm 3.

Algorithm 4 Sorted DP

Input: exp-revenue $\mathcal{S}$, budget b.
Output: the table DP and solutions Sol.

1: Initialization: $DP_{i,j} \leftarrow 0$, $Sol_{i,j} \leftarrow \emptyset$, for $i \in [0, \min\{|C|, b\}]$, $j \in [0, b]$;
2: Building b maximum heaps M with $\mathcal{S}$;
3: **for all** $i \in [1, \min\{|C|, b\}]$ **do**
4: **for all** $j \in [1, b]$ **do**
5: **if** $DP_{i,j-1} > DP_{i,j}$ **then**
6: $DP_{i,j} \leftarrow DP_{i,j-1}$;
7: $Sol_{i,j} \leftarrow Sol_{i,j-1}$;
8: **if** $DP_{i-1,j} > DP_{i,j}$ **then**
9: $DP_{i,j} \leftarrow DP_{i-1,j}$;
10: $Sol_{i,j} \leftarrow Sol_{i-1,j}$;
11: **for all** $u \in [1, j-1]$ **do**
12: Find a component c in $M[j-u]$ with the largest score and $c \notin Sol_{i-1,u}$;
13: **if** $DP_{i-1,u} + S_c[j-u] > DP_{i,j}$ **then**
14: $DP_{i,j} \leftarrow DP_{i-1,u} + S_c[j-u]$;
15: $Sol_{i,j} \leftarrow Sol_{i-1,u}$;
16: $Sol_{i,j}[c] \leftarrow j-u$;
17: **for all** $c \in Sol_{i,j}$, $b_t = Sol_{i,j}[c] > 0$ **do**
18: **for all** all larger budget $b_u > b_c$ that $b_u \in S_c$ **do**
19: **if** $DP_{i,j} - S_c[b_c] + S_c[b_u] > DP_{i,j+b_u-b_c}$ **then**
20: $DP_{i,j+b_u-b_c} \leftarrow DP_{i,j} - S_c[b_c] + S_c[b_u]$;
21: $Sol_{i,j+b_u-b_c} \leftarrow Sol_{i,j}$;
22: $Sol_{i,j+b_u-b_c}[c] \leftarrow b_u$;
23: **return** DP, Sol;

Algorithm 4 presents the details of new dynamic programming algorithm. We need to compute $\min\{|C|, b\} \times b$ values of the DP table (lines 3-22). First of all, we compress $\mathcal{S}$ into M for an easy of usage (line 2). M is a vector of size of b, where $M[i]$ is a maximum heap that sorts components from large to small, according to their scores, for $i \in [1, b]$. In other words, we first group components by desired budget, and then sort each group by their scores. As a result, we can quickly find the best option for a fixed budget. For each value $DP_{i,j}$, we first directly compare it with previous values $DP_{i,j-1}$ (line 5) and $DP_{i-1,j}$ (line 8) and take the maximum value. Next, we iterate over all $DP_{i-1,u}$, $u < j$ (line 11), and try a component c that has the largest score when the budget is $j-u$ and has not been considered. For term 4 in Equation 3, for the sake of efficiency, we do not calculate it through other tables, but use the result of $DP_{i,j}$ to calculate other tables. We iterate over all components c used in $DP_{i,j}$ (line 17), and try all budgets

TABLE I
DP TABLE BUILD BY ALGORITHM 3

b	0	1	2	3	4	5
	0	0	0	0	0	0
A	0	3	3	3	3	3
B	0	3	5	7	7	7
C	0	4	7	9	11	12

TABLE II
DP TABLE BUILD BY ALGORITHM 4

b	0	1	2	3	4	5
	0	0	0	0	0	0
1	0	4 $\{C:1\}$	5 $\{C:2\}$	6 $\{C:3\}$	6 $\{C:3\}$	6 $\{C:3\}$
2	0	4 $\{C:1\}$	7 $\{A:1, C:1\}$	8 $\{A:1, C:2\}$	9 $\{A:1, C:3\}$	10 $\{B:2, C:3\}$
3	0	4 $\{C:1\}$	7 $\{A:1, C:1\}$	9 $\{A:1, B:1, C:1\}$	11 $\{A:1, B:2, C:1\}$	12 $\{A:1, B:2, C:2\}$

b_u that are larger than the current used budget b_c, and update the value of $DP_{i,j+b_u-b_c}$. Finally, we return the calculated DP table (line 23). Since we need to consider whether a component has been used, we also need the set Sol to record the selected components.

E. Time complexity of Algorithm 4

Building heaps M takes $O(|C|b)$. The external two-layer loops are executed $\min\{|C|, b\} \times b$ times (lines 3-22). Terms 1 and 2 are executed once respectively. Line 11 repeats j times and line 12 checks at most $i+1$ times (the worst case is that the first i components in the heap have been selected) and the size of the heap is at most $|C|$, thus term 3 takes $O(b \min\{|C|, b\} \log(|C|))$. For each component c, we can just save conversion exp-revenue pairs whose budgets are no larger than b. Line 17 repeats i times and line 18 repeats at most $|S_c| \leq b$ times, thus term 4 is executed $b \times \min\{|C|, b\}$ times. Therefore, the time complexity of Algorithm 4 is $O(|C|b + b^2(\min\{b, |C|\})^2 \log(|C|))$.

This algorithm compresses S_c and builds a smaller DP table to accelerate the computation, which has a significant time improvement when there are a large number of components $|C|$. When $b \ll |C|$, the time complexity of Algorithm 4 is close to $O(|C|b)$, much faster than that of Algorithm 3, which is $O(|C|b^2)$. However, it performs bad when b is very large and obtains suboptimal results in some cases. Fortunately, in practical applications, $b \ll |C|$, and the result is very close to the optimal solution. In addition, our solution requires a compromise between efficiency and score. Therefore, we use both frameworks together to achieve the best performance. When $b > |C|$, we use Algorithm 3, otherwise we use Algorithm 4.

Example 5. *Assume there are three components* $\{A, B, C\}$ *and budget* $b = 5$*. The exp-revenues are* $S_A = [3]$ *(only one budget),* $S_B = [2, 4]$*,* $S_C = [4, 5, 6]$ *(budget can be* 1*,* 2*, or* 3*). Table I and Table II show the DP table built by Algorithm 3 and Algorithm 4, respectively. For example, in Table I,* $DP[2][2] = 5$*, because* $DP[1][1] + S_B[1] = 5$ *is the maximum. As another example, in Table II,* $DP[2][3] = 8$*, because it chooses* $S_A[1]$ *and* $S_C[2]$*. The best allocation plan is* $\mathbf{x} = [1, 2, 2]$ *and the total score is* 12*. If we use the method in [1] to solve this problem, it assumes that for a component, either do not choose it, or convert it totally, i.e.,* $S_A = [3]$*,* $S_B = [4]$*,* $S_C = [6]$ *and* $x_A \in \{0, 1\}$*,* $x_B \in \{0, 2\}$ *and* $x_C \in \{0, 3\}$*. Therefore, the best solution using the method in [1] is* $\mathbf{x} = [0, 2, 3]$ *and the total score is* 10*, which is smaller than the result of our new method as* 12*.*

TABLE III
COMPLEXITY COMPARISON OF DIFFERENT ALGORITHMS

	Component Conversion	Budget Assignment
[1]	Full conversion $O(m_c d_c^2)$	Binary DP $O(\|C\|b)$
Ours	Random conversion in Algorithm 1 $O(m_c(d_c + r\rho))$	Sequential DP in Algorithm 3 $O(\|C\|b^2)$
	Min-cut conversion in Section IV-C $O(tm_c(n_c^2 + d_c^2))$	Sorted DP in Algorithm 4 $O(\|C\|b + b^2(\min\{b, \|C\|\})^2 \log(\|C\|))$

F. The Summary of Complexity Comparison

Our methods and the existing method [1] both have two key steps: component conversion and budget assignment. [1] adopts full conversion strategy and uses binary DP to solve the 0-1 backpack problem for budget assignment. Let us consider a candidate $(k-1)$-light component c. We denote the number of nodes and edges as $n_c = |V(C)|$ and $m_c = |E(C)|$, respectively. The maximum node degree is d_c in component c. First, we analyze the time complexity of component conversion on c. In the worst case, all edges in component c need to be converted by the additional support of new edges. For each candidate $e = (u, v)$, it needs the number of potential new edges that form triangles directly with e, which is bounded by $O(|N(v)| + |N(u)|) = O(d_c)$. Thus, we have a total of potential new edges $O(m_c d_c)$. The time cost of computing the support for all edges takes $O(d_c)$. Thus, the time complexity of full conversion is $O(m_c {d_c}^2)$. On the other hand, our random-based partial conversion in Algorithm 1 first finds all potential new edges in $O(m_c d_c)$ time, then randomly inserts no more than b edges, and checks the feasibility of k-truss by truss maintenance algorithm in $O(\rho m_c)$, where ρ is the arboricity of component c with $\rho \leq \min\{d_c, \sqrt{m_c}\}$. The above process needs to be repeated in r times for finding good answers. Thus, the time complexity is $O(m_c(d_c + r\rho))$. In addition, our min-cut based partial conversion invokes the minimum cut algorithm for finding candidates in $O(m_c n_c^2)$ time and conducting the conversion in $O(m_c d_c^2)$ time. The whole process repeating t times has a time complexity of $O(tm_c(n_c^2 + d_c^2))$ Next, we analyze the DP-based techniques for budget assignment. Assume that we have a total of $|C|$ different components C and a budget b. The binary 0-1 DP [1] builds a $|C| \times b$ DP table, thus the time complexity is $O(|C|b)$. The time complexities of Sequential and Sorted DP algorithms are presented in Section V, which cost more than binary DP for achieving high-quality of combined answers.

VI. EXTENSION TO HANDLE $(k-h)$-TRUSS

Previous sections show how to convert $(k-1)$-class to k-truss. However, in some cases where $(k-1)$-class is empty or

Algorithm 5 General Framework

Input: graph $G = (V, E)$, the number k, the budget b.
Output: the edge set A to be inserted.
1: Compute trussness $\tau(e)$ for edges $e \in E$;
2: $k' \leftarrow k-1$, $A \leftarrow \emptyset$;
3: **while** $k' > 2$ **do**
4: Partition k'-truss to components;
5: Compute exp-revenue $\mathcal{S}$ for components by Algorithm 1 and techniques in Section IV-C;
6: Using Algorithm 3 or Algorithm 4 to find at most $b-|A|$ new edges U;
7: Insert U to G and update $\tau(e)$ for $e \in E$;
8: $A \leftarrow A \cup U$;
9: **if** $|A| \geq b$ **then**
10: break;
11: $k' \leftarrow k'-1$;
12: **return** A;

b is large, $(k-1)$-class has been completely converted to k-truss. This shows a budget surplus. Therefore, we also convert $(k-h)$-truss to k-truss when $h > 1$ and b is enough. First of all, we extend the definition of components of $(k-1)$-truss to $(k-h)$-truss.

Definition 7 (General Component)**.** *A general component c is a connected subgraph where every edge e has $k-h \leq \tau(e) < k$.*

We also extend the concept of onion layer as follows.

Definition 8 (General Onion Layer L_G)**.** *The general onion layer of an edge e in a general component c, i.e., $e \in E_c$, is defined as the number of rounds in which the edge was removed in the computation of k-truss, i.e., $L_G(e) = \max\{0, l \in N : \sup_H(e)+2 \geq k\}+1$, where $H = T_k \cup E_c - \{e' | L_G(e') < l\}$.*

The general onion layer shows the structure of edges in the $(k-h)$-truss but not in the k-truss. Obviously, when we anchor all edges in the first general onion layer, i.e., $L_G(e) = 1$, the whole component becomes the k-truss. Given the definitions of general components and general onion layer, we can naturally extend techniques in previous sections to $(k-h)$-truss.

Next, we present the general framework of converting $(k-h)$-truss to k-truss. Algorithm 5 shows how to convert edges to k-truss edges with budget b. We need three inputs: the graph G, the number k and the budget b. First of all, we compute the trussnesses of edges in the graph (line 1). We start from $(k-1)$-truss (line 2) and if the budget is enough, we decrease k' to convert other trusses (line 11). For each k'-truss, we first divide it to many general components (line 4) and build exp-revenue using techniques in Section IV. Then, we use DP in Section V to find the optimal combinations of components and their budgets. The result is U and its size is no more than $b-|A|$. We insert these edges U into G and update trussnesses and save U to A. If all budgets are spent, the algorithm ends. Otherwise, the algorithm continues to convert $(k'-1)$-truss (line 11).

VII. Experiments

In this section, we evaluate our algorithms with other baseline algorithms. The experiments are conducted on a Linux Server with AMD EPYC 7742 (2.25 GHz, 2S/64C) and 2T main memory. All algorithms are implemented in C++.[1]

Datasets. We use nine real-world datasets as shown in Table IV. The dataset Syracuse56 is from [48]. All other datasets can be downloaded from SNAP [49]. Facebook, Syracuse56, Brightkite, Gowalla and LiveJournal are friendship networks. Enron is an email communication network. The dataset Twitter is crawled from Twitter. The dataset Stanford is the Stanford web graph. Wiki-Talk is a Wikipedia talk graph. All directed graphs are converted to undirected graphs.

Competitors. We compare our algorithm PCFR with three baseline algorithms, RD [1], GTM [1], and CBTM [1]. PCFR contains all techniques proposed in this work. It uses random method (Algorithm 1) to partially convert components in $(k-1)$-truss to k-truss and uses the flow method (Section IV-C) to partially convert components in $(k-h)$-truss to k-truss, where h is sequentially $1, 2, \cdots, k-2$. Then it uses dynamic programming (Algorithm 3 when $b > |C|$, otherwise Algorithm 4) to find the optimal combination. For each component, Algorithm 1 tests 10 times ($r = 10$). The minimum-cut based method in Section IV-C sets w_1 to 1, and tests w_2 twice (1 or 10). For each case, it tests the value of g 10 times ($t = 10$). CBTM completely converts components in $(k-1)$-truss to k-truss (each component has only one exp-revenue pair) and uses binary dynamic programming to find the optimal combination. RD and GTM both first find a candidate edge set C where edges are not in the graph and have support no less than $k-2$, the insertion of which can increase the support of edges in $(k-1)$-truss. RD randomly chooses b edges from C as the result and directly inserts them into the graph. GTM is a per-edge insertion greedy method and uses candidate pruning techniques in [1]. We also modify PCFR to test the effectiveness of our techniques. Algorithm PCF removes the random method, Algorithm PCR removes the flow method, compared with PCFR.

Exp-I: Efficiency evaluation. In this experiment, we compare our algorithm PCFR with the baseline algorithms, RD, GTM and CBTM. We set $k = 20$ for five small datasets, and $k = 40$ for four large datasets (Twitter, Stanford, Wiki-Talk, LiveJournal), respectively. This parameter setting follows the configuration in [1]. We set the budget $b = 200$ and report the score (the increased edge size of new k-truss). We also report the running time of algorithms. "-" means the algorithm cannot finish in 24 hours. As shown in Table IV, our algorithm PCFR achieves the highest score than all baselines on all datasets, because PCFR considers multiple exp-revenue pairs for a component, which also include the complete conversion plan by CBTM. On the other hand, for the same reason, PCFR takes a longer time, but it is worth the extra time. Especially on dataset Stanford, PCFR takes the same budget but achieves

[1] https://github.com/MaxTruss/MaxTruss

TABLE IV
EFFICIENCY EVALUATION

Network	$\|V\|$	$\|E\|$	d_{max}	k_{max}	$\|C\|$	Score				Running Time (in seconds)			
						RD	GTM	CBTM	PCFR	RD	GTM	CBTM	PCFR
Facebook	4,039	88,234	1,045	97	100	873	1278	1821	**3635**	3.51	620.91	**3.30**	24.86
Enron	36,692	183,831	1,383	22	9	3204	2209	3858	**5165**	**23.11**	5037.56	27.26	382.90
Brightkite	58,228	214,078	1,134	43	55	852	611	989	**1468**	5.98	689.66	8.93	**5.81**
Syracuse56	13,654	543,982	1,340	59	354	1277	7234	7261	**7482**	131.62	79192	**106.83**	219.57
Gowalla	196,591	950,327	14,730	29	110	1519	3868	4439	**4656**	**86.39**	48295	113.84	193.84
Twitter	81,306	1,768,149	3,383	82	82	4843	4139	7017	**13583**	**131.38**	21841	185.96	1071.62
Stanford	281,903	2,312,497	38,625	62	655	3024	5007	4200	**13111**	112.90	2967.99	**77.68**	1023.20
Wiki-Talk	2,394,385	5,021,410	100,029	53	115	1459	-	5400	**5464**	**760.01**	-	826.19	6581.90
LiveJournal	3,997,962	34,681,189	14,815	352	433	1010	-	16412	**20139**	**128.08**	-	1041.07	227.07

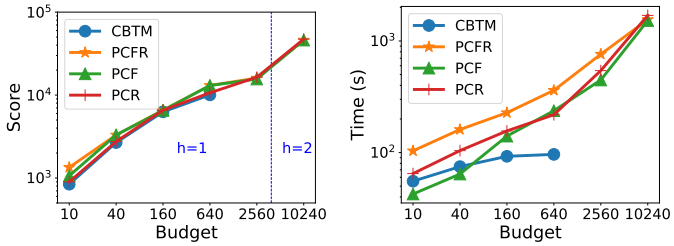


Fig. 4. The score (left) and running time (right) of our algorithms and CBTM by varying b on Syracuse56, $k = 20$.

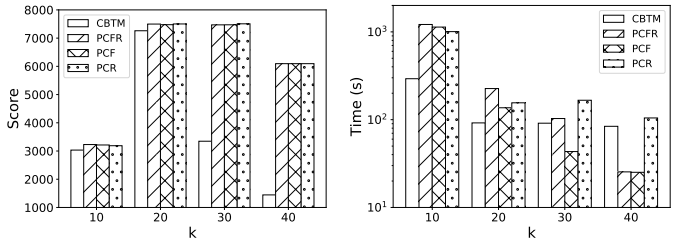
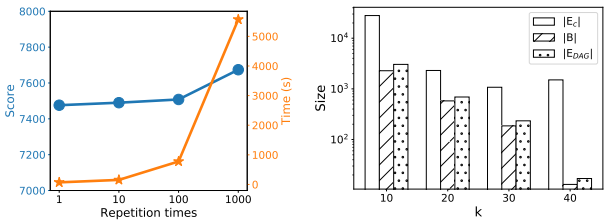


Fig. 5. The score (left) and running time (right) of our algorithms and CBTM by varying k on dataset Syracuse56, $b = 200$.



(a) Repeating test (b) DAG size

Fig. 6. Performance of PCR by varying r, with $k = 20$ and $b = 200$ in subfigure (a) and the size of the largest component $|E_c|$, the number of nodes $|B|$ and edges $|E_{DAG}|$ in the corresponding DAG by varying k in subfigure (b) on dataset Syracuse56.

more than 3 times of score than that of CBTM. For some datasets, like Enron and Stanford, PCFR runs much slower than CBTM, because PCFR also finds solutions in $(k-h)$-truss, where $h > 1$, while CBTM only finds solutions in $(k-1)$-truss. For dataset LiveJournal, PCFR runs much faster than CBTM, which shows the efficiency of Algorithm 4. RD runs fast on many datasets, but has very low score. GTM runs the slowest, because it involves too many truss maintenance operations.

Exp-II: Parameter evaluation. In this experiment, we test the efficiency of our algorithms by varying the total budget b, the target trussness k, and the repeating times r.

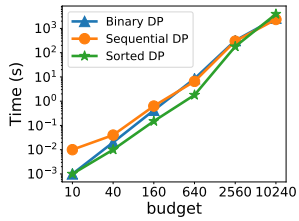
First of all, we test our algorithms by varying the budget b. Fig. 4 reports the score and running time obtained by our algorithms and the baseline algorithm CBTM, when $k = 20$ on dataset Syracuse56. The blue numbers represent the value of h, reflecting the $(k-h)$-class our algorithms handled. When $b \leq 3929$, all algorithms convert edges in $(k-1)$-class and when $b > 3929$, our algorithms continue to convert edges in $(k-2)$-class. CBTM cannot handle the case when b is extremely large ($b = 2560$ and 10240), thus we do not plot these two points. If we only look at the results of PCFR, we can find that the conversion rate (score/b) decreases as the budget b increases. This is because we prefer to prioritize edges with high conversion rate. Comparing PCFR with CBTM, we can find that when b is very small or very large, our algorithm can achieve better performance. This is because when b is very small, our partial conversion plans can get higher score. When b is very large, our algorithm can find more candidate edges from $(k-h)$-truss. However, when b is close to the number that can completely convert all $(k-1)$-truss to k-truss, our algorithm and CBTM have similar performance. In most cases, PCFR runs slower than other algorithms. Because PCFR uses both techniques in PCF and PCR. PCFR considers more cases than CBTM, but the running time is much less than the complexities expected ($O(|C|b^2)$ for PCFR and $O(|C|b)$ for CBTM). PCR takes more time than PCF in most cases, because PCR uses a randomized algorithm that needs to maintain trussnesses in every attempt.

Next, we test our algorithms by varying k, when $b = 200$ on dataset Syracuse56, as shown in Fig. 5. The score has no apparent relationship to k, because the structures of k-class may differ from each other. But the size of the k-truss decreases as k increases. As a result, the running time roughly decreases with the increase of k. Our algorithms also perform much better than CBTM when k is large. The reasons are twofold. First, the size of $(k-1)$-truss is small when k is large and our algorithms continue to convert $(k-h)$-truss. Second, there are many components that CBTM cannot convert since it is difficult to find edges with sufficient supports when k is large. PCF runs faster than PCR when k increases, because PCR involves many truss maintenance operations, which change more states of edges when k is larger. PCFR runs faster than PCR when $k = 40$, because PCR conducts random insertions on $(k-h)$-truss, while PCFR only conducts random insertions on $(k-1)$-truss. The scores of PCFR and PCR are almost the same, but PCFR runs much faster than PCR, which shows the limitation of random algorithms on $(k-h)$-truss for $h > 1$.

We also test our randomization algorithm PCR with different repeating times. As shown in Fig. 6 (a), as r increases, the score increases slowly, but the running time increases quickly.

TABLE V
THE SCORE OF BINARY (0-1) DP [1], SEQUENTIAL DP (ALGORITHM 3) AND SORTED DP (ALGORITHM 4) BY VARYING b ON GOWALLA, $k = 10$

b	Binary DP	Sequential DP	Sorted DP
10	495	**650**	**650**
40	1526	**2009**	**2009**
160	4217	**5084**	**5084**
640	11904	**13065**	**13065**
2560	28966	**32638**	32627
10240	43562	**43565**	**43565**



(a) Gowalla

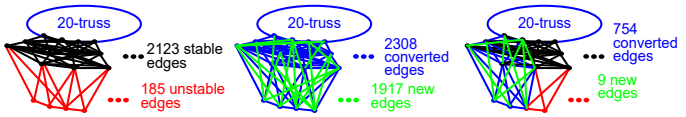
Fig. 7. Running time of Binary DP, Sequential DP in Algorithm 3 and Sorted DP in Algorithm 4 by varying b on Gowalla, $k = 10$, where $|C| = 3727$.

Consequently, in our experiments, we only set $r = 10$ for efficiency.

Exp-III: DAG size evaluation. We report the size of a component and the size of the transformed DAG in our algorithms, as shown in Fig. 6 (b). We select the largest component of k-class with different k on the dataset Syracuse56. $|E_c|$ is the number of edges in the component. $|B|$ and $|E_{DAG}|$ represent the number of vertices and links in the DAG, respectively. In our algorithms, we merge connected edges in a component with the same onion layers into a Block, which is the vertex of the DAG, and two connected Blocks have a link in the DAG. The result shows that the size of the DAG is much smaller than that of the component. Accordingly, our algorithms can efficiently handle large graphs. It also shows that the size of DAG is smaller when k increases, because k-truss is more cohesive and more edges are in the same onion layers.

Exp-IV: Efficiency and quality evaluation of three algorithms: Binary DP, Sequential DP, and Sorted DP.

We compare two DP algorithms proposed in Section V, as well as a binary version (Binary DP) modified from our Sequential DP. For each component, Binary DP either chooses it with the largest budget or does not choose it. We set $k = 10$ and vary b on dataset Gowalla, where the number of components $|C|$ is 3727. Fig. 7 reports the running time of three DP algorithms. When $b \leq 640 < |C|$, Sorted DP performs better than Sequential DP. When $b = 10240 \geq |C|$, Sequential DP runs faster. Thus, our algorithm PCFR uses Sequential DP to find the best combination for $b \geq |C|$ and then uses Sorted DP for $b < |C|$. Binary DP and Sequential DP have similar performance when b is not very small ($b > 10$), because in practice, there are very few budget options for each component to choose from. In addition, we also report the quality of three DP algorithms, as shown in Table V. Although Sorted DP obtains suboptimal results, the score difference is very small, i.e., 11 when $b = 2560$. Sequential DP and Sorted DP outperform Binary DP in most case for a small b. When the total budget is very large, i.e., $b = 10240$, three algorithms achieve similar scores, because all components tend to be completely converted.



(a) Original graph (b) Full conversion (c) Partial conversion by our PCFR method

Fig. 8. Case study on social network Syracuse56. Here, $k = 20$. Black edges represent the stable edges, red edges represent the unstable edges, green edges represent the new inserted edges, and blue edges represent edges converted to 20-truss edges.

Exp-V: Case study on social network Syracuse56. In this experiment, we investigate the effectiveness of our algorithm PCFR on Syracuse56, where nodes represent users and edges represent their friendships on a social network Facebook. We focus on a large candidate component for k-truss conversion where $k = 20$. Fig. 8(a) shows the candidate component with 2,308 edges, of which 185 edges are unstable. To fully convert all 2,308 edges into 20-truss, the complete conversion method needs to insert 1,917 new edges to provide sufficient supports for these unstable edges as shown in Fig. 8(b). In this way, it achieves the conversion ratio of $\frac{2308+1917}{1917} = 2.2$, which is pretty small. On the other hand, our partial conversion algorithm only inserts 9 new edges, which converts 754 candidate edges to k-truss edges as shown in Fig. 8(c). Our method achieves a significant conversion ratio of $\frac{754+9}{9} = 84.8$, which is much larger than 2.2 achieved by the complete conversion. This shows practical usefulness of our proposed method in real applications of truss maximization under a limited number of budgets, especially in the economic consideration of coupon promotions by inviting friends to participate in activities on social networks, and also adding new routes for connectivity extension in flight networks.

VIII. CONCLUSION

In this study, we focus on the truss maximization problem, which involves identifying no more than b new edges to expand the existing k-truss. Our approach involves dividing the $(k-h)$-truss into separate components that do not affect each other. Then we propose a minimum-cut based approach to partially convert each component to k-truss, providing our algorithms with a wider range of edge insertion plans. To handle multiple budget assignments, we propose a framework that enables the identification of the optimal combination of these conversion options. Our algorithm, PCFR, demonstrates the ability to effectively enlarge the k-truss across various budget variations. We validate the effectiveness and efficiency of our algorithms through extensive experiments.

ACKNOWLEDGMENT

The work is supported by Hong Kong RGC Grant Nos. 22200320, 12200021, 12201923, C2004-21GF, and 12202221. Xin Huang is the corresponding author.

REFERENCES

[1] X. Sun, X. Huang, Z. Sun, and D. Jin, "Budget-constrained truss maximization over large graphs: A component-based approach," in *CIKM*, pp. 1754–1763, 2021.

[2] J. Wang and J. Cheng, "Truss decomposition in massive networks," *Proc. VLDB Endow.*, pp. 812–823, 2012.

[3] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu, "Querying k-truss community in large and dynamic graphs," in *SIGMOD*, pp. 1311–1322, 2014.

[4] C. Zong, P. Gong, X. Zhang, T. Qiu, A. Zhang, and M. Wang, "Efficient size-constrained (k, d)-truss community search," in *ADMA*, pp. 405–420, 2023.

[5] X. Liu and Z. Zou, "Common-truss-based community search on multi-layer graphs," in *ADMA*, pp. 277–291, 2023.

[6] W. M. A. Habib, H. M. O. Mokhtar, and M. E. El-Sharkawi, "Discovering top-weighted k-truss communities in large graphs," *J. Big Data*, p. 36, 2022.

[7] Q. Liu, M. Zhao, X. Huang, J. Xu, and Y. Gao, "Truss-based community search over large directed graphs," in *SIGMOD*, pp. 2183–2197, 2020.

[8] E. Akbas and P. Zhao, "Truss-based community search: a truss-equivalence based indexing approach," *Proc. VLDB Endow.*, pp. 1298–1309, 2017.

[9] X. Sun, X. Huang, and D. Jin, "Fast algorithms for core maximization on large graphs," *Proc. VLDB Endow.*, vol. 15, no. 7, pp. 1350–1362, 2022.

[10] R. Chitnis and N. Talmon, "Can we create large k-cores by adding few edges?," in *Computer Science - Theory and Applications - 13th International Computer Science Symposium, CSR*, pp. 78–89, 2018.

[11] Z. Zhou, F. Zhang, X. Lin, W. Zhang, and C. Chen, "K-core maximization: An edge addition approach," in *IJCAI*, pp. 4867–4873, 2019.

[12] K. Bhawalkar, J. M. Kleinberg, K. Lewi, T. Roughgarden, and A. Sharma, "Preventing unraveling in social networks: The anchored k-core problem," *SIAM J. Discret. Math.*, pp. 1452–1475, 2015.

[13] Q. Linghu, F. Zhang, X. Lin, W. Zhang, and Y. Zhang, "Global reinforcement of social networks: The anchored coreness problem," in *SIGMOD*, pp. 2211–2226, 2020.

[14] H. Kabir and K. Madduri, "Shared-memory graph truss decomposition," in *HiPC*, pp. 13–22, 2017.

[15] P. Chen, C. Chou, and M. Chen, "Distributed algorithms for k-truss decomposition," in *International Conference on Big Data*, pp. 471–480, 2014.

[16] R. Wang, L. Yu, Q. Wang, J. Xin, and L. Zheng, "Productive high-performance k-truss decomposition on GPU using linear algebra," in *HPEC*, pp. 1–7, 2021.

[17] S. Huang, M. El-Hadedy, C. Hao, Q. Li, V. S. Mailthody, K. Date, J. Xiong, D. Chen, R. Nagi, and W. Hwu, "Triangle counting and truss decomposition using FPGA," in *HPEC*, pp. 1–7, 2018.

[18] X. Huang, W. Lu, and L. V. S. Lakshmanan, "Truss decomposition of probabilistic graphs: Semantics and algorithms," in *SIGMOD*, pp. 77–90, 2016.

[19] Z. Zou and R. Zhu, "Truss decomposition of uncertain graphs," *Knowledge and Information Systems*, vol. 50, no. 1, pp. 197–230, 2017.

[20] F. Esfahani, J. Wu, V. Srinivasan, A. Thomo, and K. Wu, "Fast truss decomposition in large-scale probabilistic graphs," in *EDBT*, pp. 722–725, 2019.

[21] Z. Sun, X. Huang, J. Xu, and F. Bonchi, "Efficient probabilistic truss indexing on uncertain graphs," in *WWW*, pp. 354–366, 2021.

[22] F. Esfahani, M. Daneshmand, V. Srinivasan, A. Thomo, and K. Wu, "Truss decomposition on large probabilistic networks using h-index," in *SSDBM*, pp. 145–156, 2021.

[23] Y. Wu, R. Sun, C. Chen, X. Wang, and Q. Zhu, "Maximum signed (k, r)-truss identification in signed networks," in *CIKM*, pp. 3337–3340, 2020.

[24] J. Zhao, R. Sun, Q. Zhu, X. Wang, and C. Chen, "Community identification in signed networks: A k-truss based model," in *CIKM*, pp. 2321–2324, 2020.

[25] X. Huang and L. V. Lakshmanan, "Attribute-driven community search," *Proc. VLDB Endow.*, vol. 10, no. 9, pp. 949–960, 2017.

[26] Q. Liu, Y. Zhu, M. Zhao, X. Huang, J. Xu, and Y. Gao, "VAC: vertex-centric attributed community search," in *ICDE*, pp. 937–948, 2020.

[27] X. Xie, M. Song, C. Liu, J. Zhang, and J. Li, "Effective influential community search on attributed graph," *Neurocomputing*, pp. 111–125, 2021.

[28] S. Ebadian and X. Huang, "Fast algorithm for k-truss discovery on public-private graphs," in *IJCAI*, pp. 2258–2264, 2019.

[29] Q. Luo, D. Yu, X. Cheng, Z. Cai, J. Yu, and W. Lv, "Batch processing for truss maintenance in large dynamic graphs," *IEEE Trans. Comput.*, pp. 1435–1446, 2020.

[30] Y. Zhang and J. X. Yu, "Unboundedness and efficiency of truss maintenance in evolving graphs," in *SIGMOD*, pp. 1024–1041, 2019.

[31] Z. Sun, X. Huang, Q. Liu, and J. Xu, "Efficient star-based truss maintenance on dynamic graphs," *Proc. ACM Manag. Data*, pp. 133:1–133:26, 2023.

[32] L. Chen, C. Liu, R. Zhou, J. Li, X. Yang, and B. Wang, "Maximum co-located community search in large scale social networks," *Proc. VLDB Endow.*, vol. 11, no. 10, pp. 1233–1246, 2018.

[33] R. Sun, Y. Wu, and X. Wang, "Diversified top-r community search in geo-social network: A k-truss based model," in *EDBT*, pp. 2:445–2:448, 2022.

[34] Y. Li, T. Kuboyama, and H. Sakamoto, "Truss decomposition for extracting communities in bipartite graph," in *IMMM*, pp. 76–80, 2013.

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

[36] H. Huang, Q. Linghu, F. Zhang, D. Ouyang, and S. Yang, "Truss decomposition on multilayer graphs," in *Big Data*, pp. 5912–5915, 2021.

[37] G. Preti, G. D. F. Morales, and F. Bonchi, "Strud: Truss decomposition of simplicial complexes," in *WWW* (J. Leskovec, M. Grobelnik, M. Najork, J. Tang, and L. Zia, eds.), pp. 3408–3418, 2021.

[38] R. Li, J. X. Yu, and R. Mao, "Efficient core maintenance in large dynamic graphs," *IEEE Trans. Knowl. Data Eng.*, pp. 2453–2465, 2014.

[39] A. E. Sariyüce, B. Gedik, G. Jacques-Silva, K. Wu, and Ü. V. Çatalyürek, "Streaming algorithms for k-core decomposition," *Proc. VLDB Endow.*, pp. 433–444, 2013.

[40] A. E. Sariyüce, B. Gedik, G. Jacques-Silva, K. Wu, and Ü. V. Çatalyürek, "Incremental k-core decomposition: algorithms and evaluation," *VLDB J.*, pp. 425–447, 2016.

[41] Y. Zhang, J. X. Yu, Y. Zhang, and L. Qin, "A fast order-based approach for core maintenance," in *ICDE*, pp. 337–348, 2017.

[42] Q. Liu, X. Zhu, X. Huang, and J. Xu, "Local algorithms for distance-generalized core decomposition over large dynamic graphs," *Proc. VLDB Endow.*, pp. 1531–1543, 2021.

[43] F. Corò, G. D'Angelo, and C. M. Pinotti, "Adding edges for maximizing weighted reachability," *Algorithms*, p. 68, 2020.

[44] F. Zhang, Y. Zhang, L. Qin, W. Zhang, and X. Lin, "Efficiently reinforcing social networks over user engagement and tie strength," in *ICDE*, pp. 557–568, 2018.

[45] J. Cohen, "Trusses: Cohesive subgraphs for social network analysis," tech. rep., National Security Agency, 2008.

[46] A. V. Goldberg, "Finding a maximum density subgraph," 1984.

[47] M. Bussieck, H. Hassler, G. J. Woeginger, and U. T. Zimmermann, "Fast algorithms for the maximum convolution problem," *Operations research letters*, pp. 133–141, 1994.

[48] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI* (B. Bonet and S. Koenig, eds.), pp. 4292–4293, 2015.

[49] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection." http://snap.stanford.edu/data, June 2014.