I/O Efficient Max-Truss Computation in Large Static and Dynamic Graphs

Jiaqi Jiang, Qi Zhang, Rong-Hua Li, Qiangqiang Dai, Guoren Wang

Beijing Institute of Technology, Beijing, China;

jiaqijiang@bit.edu.cn; qizhangcs@bit.edu.cn; lironghuabit@126.com; qiangd66@gmail.com; wanggrbit@gmail.com

***Abstract*—Cohesive subgraph mining has received much attention in the area of graph analysis. A k-truss, defined as a subgraph where each edge is associated with at least $k-2$ triangles, serves as a fundamental graph analysis tool. Among all k-trusses, the $k_{\max}$-truss with the maximum k value holds significant importance in various practical applications such as community search and keyword retrieval. Furthermore, it is also closely related to many graph analysis problems, particularly those computational complexity problems parameterized by k. However, real-world graphs often exhibit large-scale characteristics, making it impractical to fully load them into main memory. In this paper, we investigate the problem of finding the $k_{\max}$-truss in external memory settings. To address this problem, we propose an I/O efficient algorithm following a semi-external model, which only allows node information to be loaded into main memory. Our approach leverages greedy strategies and a binary search framework to efficiently find the $k_{\max}$-truss. Subsequently, an elegant data structure is proposed to significantly reduce I/O costs. Furthermore, to address dynamic graph updates, we develop an I/O efficient $k_{\max}$-truss maintenance algorithm based on the local-first update technique. To evaluate the performance of our algorithms, we conduct extensive experiments. The results demonstrate the high efficiency and scalability of our algorithms, which are at least two orders of magnitude faster in runtime and at least one order of magnitude lower in terms of I/O costs compared to the state-of-the-art solutions.**

I. INTRODUCTION

Given a graph $G=(V,E)$, a k-truss is referred to as a subgraph where each edge is associated with at least $k-2$ triangles. The $k_{\max}$-truss, representing the trusses with the maximum k value among all k-trusses [1], encapsulates the central structure of the graph. Obviously, the $k_{\max}$-truss, a densely connected subgraph, retains its profound significance as a critical subgraph within graph G, and it can be applied to practical applications. For instance, in the field of community search, the goal revolves identifying maximal communities with maximum trussness that contain a set of query nodes [2], [3]. Similarly, keyword retrieval aims to find a minimal subgraph with maximum trussness covering the keywords [4].

Moreover, the $k_{\max}$ plays a pivotal role in shaping the complexity analysis for various graph algorithms. It is widely employed as a parameter in fixed-parameter tractable (FPT) graph algorithms [5], where the computational complexity is closely tied to the exponential function of $k_{\max}$. Notably, problems such as the maximum clique problem [6] and clique listing problem [7] are parameterized by $k_{\max}$. Thus, computing the $k_{\max}$ of a graph G can be useful to predict whether such FPT algorithms are tractable in G.

Due to its significance, a fundamental problem is to identify the entire k_{max}-truss within the graph G. However, it is very hard to estimate $k_{\max}$ without computing the entire $k_{\max}$-truss. All k-trusses exhibit a nested hierarchical structure, as depicted in Fig. 1. If the entire $k_{\max}$-truss subgraph is not computed, we assume that it is possible to estimate either subgraph A or B, where subgraph A contains some edges outside the $k_{\max}$-truss, B is a partial subgraph of $k_{\max}$-truss (possibly is not the entire of $k_{\max}$-truss). It is easy to see that subgraph A does not form a $k_{\max}$-truss, as there exist edges with the support less than $k_{\max}-2$. Since subgraph B is possibly not the entire $k_{\max}$-truss, there may be edges with support less than $k_{\max}-2$, which results in subgraph B possibly not being a $k_{\max}$-truss. In addition, it is very difficult to estimate a subgraph B that is exactly a $k_{\max}$-truss. To solve this problem, the existing studies are mainly based on peeling techniques [8]–[10].

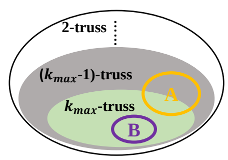


Fig. 1. The hierarchy of all k-trusses

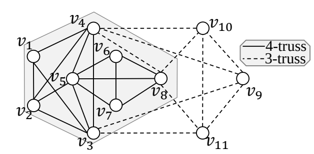


Fig. 2. The running example

These existing algorithms are tailored for in-memory processing to compute $k_{\max}$-truss. However, with the continuous growth in graph sizes, fully loading them into limited memory becomes impractical, making these algorithms incapable of handling large graphs. To address this limitation, Wang *et al.* [11] first proposed an external memory algorithm, called Bottom-Up, by performing complete truss decomposition to obtain the $k_{\max}$-truss. They further enhanced Bottom-Up through the integration of the h-index technique, resulting in the state-of-the-art algorithm Top-Down. Both of these algorithms mainly follow a peeling-based idea, briefly summarized as follows: (1) the input graph is partitioned into multiple local graphs with each local graph loaded into memory for k-truss calculations; (2) the edges connecting these local graphs are reconstructed to form a new graph, and the process returns to (1) iteratively until all edges have been processed.

However, the Top-Down algorithm for $k_{\max}$-truss computation, suffers from three notable drawbacks: (1) It requires too many graph partitions to yield results, and each partition incurs read and write I/O overhead. Moreover, the vertex-based uniform partitioning approach leads to imbalance regarding the size of the graph composed of the set of vertices loaded into memory, potentially exceeding the available memory capacity; (2) The technique employed to compute the upper bound of the k-truss corresponding to each edge is highly time-consuming and results in significant I/O overhead. Additionally, the comparatively loose upper bound can further lead to excessive I/O overhead incurred by Top-Down to obtain the final result; (3) The Top-Down algorithm cannot efficiently maintain the

$k_{\max}$-truss when the networks are dynamically updated. When an edge is inserted or deleted, it must re-compute the $k_{\max}$-truss from scratch.

To address these drawbacks, we present novel I/O efficient algorithms for $k_{\max}$-truss computation, and for the first time, develop the efficient algorithm for $k_{\max}$-truss maintenance under external memory settings. Specifically, we first develop a binary search method and a greedy pruning technique. Then, we design a novel data structure to minimize the read and write costs associated with updating edge information on disk. Subsequently, a novel algorithm for the maintenance of $k_{\max}$-truss is proposed by using local-first update technique. Finally, we conduct extensive experiments to evaluate our proposed algorithms using 171 graphs, and the results demonstrate the efficiency, scalability, and effectiveness of the proposed algorithms. Below, we summarize our contributions as follows.

Novel I/O efficient $k_{\max}$-truss computation algorithms. First, we propose a greedy strategy to identify a local $k'_{\max}$-truss and adopt a binary search framework to reduce I/O overhead. Then, incorporating upper and lower bound techniques to obtain a compact binary search interval which enhances algorithm efficiency. Specifically, the upper bound is based on the k-core technique and the lower bound is based on the theory of average support of the edges. Finally, we efficiently obtain the $k_{\max}$-truss based on the local maximum k-truss using the peeling method.

Novel structure to largely reduce I/O. We combine both disk-based linear-heap and memory-based dynamic-heap into a composite data structure, called LHDH. The disk-based linear-heap, which stores all edges on disk in increasing order of support, enables efficient loading and writing of edges. Since we observe that, some edges need to be frequently updated during the search, we utilize the memory-based dynamic-heap to hold the frequently updated edges in memory. As a result, by utilizing both components, the highly innovative LHDH structure effectively reduces I/O overhead.

The first I/O efficient $k_{\max}$-truss maintenance. We adopt the peeling-based technique to handle an edge insertion or deletion and update the $k_{\max}$-truss accordingly. In our approach, we employ a two-tiered update strategy. Initially, we employ a local-first update technique to update potentially affected areas. As the extent of the affected area surpasses a threshold, we seamlessly transition to a global-second update technique. In the global update phase, we further optimize the process by applying the core pruning technique to filter candidate nodes, subsequently recomputing the $k_{\max}$-truss on this refined set of nodes.

Extensive experiments. We systematically analyze the value of k_{max} of 171 graphs and confirm that the value of $k_{\max}$ for most real-world graphs is much smaller than the degeneracy value, which means it can obtain a tighter bound on the time complexity for the clique listing problem. Besides, we conducted extensive experiments on five medium-sized graphs and five massive-sized graphs to evaluate the performance of our proposed algorithms. The experimental results show that: (1) the proposed SemiLazyUpdate, $k_{\max}$-truss computation method, is two orders of magnitudes faster than the state-of-the-art algorithm Top-Down [11], both I/O costs and memory costs; (2) the $k_{\max}$-truss maintenance method is two orders of magnitudes faster than the method presented in [12] with the ability of processing large-size graphs in less than 10 seconds; (3) the lower bound is closer to the $k_{\max}$, indicating the effectiveness of our greedy strategy. Additionally, we also conduct two case studies to demonstrate the effectiveness of our $k_{\max}$-truss compared to other models.

TABLE I
FREQUENTLY USED NOTATIONS

Notation	Description
$G = (V, E)$	an undirected and unweighted graph
$V(G), E(G)$	the vertex/edge set of G
$N_v(G)$	the set of neighbors of a vertex v in G
$d_{max}(G)$	the maximum vertex degree in G
$\triangle_{uvw}$	a triangle formed by u,v and w
$sup((u, v))$	the support of edge $e = (u, v)$ in G
k_{max}	the maximum trussness of the edges in G
$core(v)$	the coreness of v in G
c_{max}	the maximum coreness of vertex in G
$\S^k_\triangle(G)$	the number of triangles containing edges with support from 0 to k in G
$\mathcal{T}_{edge}(G)$	the storage of all edges in non-decreasing order of support
$E^k_{sup}(G)$	the set of edges with support k in G
$G_{c_{\max}}$,$c_{\max}$-core	the subgraph of G induced by the vertices with maximum coreness
$\triangle^k_{(u,v)}$	$\{\triangle_{uvw} : min\{sup((u, w)), sup((v, w))\} \geq k - 2\}$

II. PRELIMINARIES

Let us consider an undirected and unweighted graph $G = (V, E)$, where $V(G)$ represents the set of vertices and $E(G)$ represents the set of edges. The graph G is composed of $n = |V|$ vertices and $m = |E|$ edges. We define the set of neighbors of a vertex v by $N_v(G)$, i.e., $N_v(G) = \{u \in V : (v, u) \in E\}$, and the degree of v by $d_v(G) = |N_v(G)|$. We use $d_{\max}(G)$ to denote the maximum vertex degree in G. Given a set of nodes $V' \in V$, the subgraph induced by V' is defined as $G(V') = (V', E')$, where $E' = \{(u, v)|(u, v) \in E, u \in V', v \in V'\}$.

Nodes u, v, w form a triangle denoted as $\triangle_{uvw}$ due to their pairwise connections. Furthermore, we denote the total number of triangles in graph G as $\triangle_G$ and the set of all distinct triangles in G as $T(G)$.

Definition 1 (Support): The support of an edge $e = (u, v)$ in G is the number of triangles containing e, defined as $sup(e, G) = |\{\triangle_{uvw} : w \in V\}|$. When the context is clear, we simplify $sup(e, G)$ as $sup(e)$.

Definition 2 (k-Truss [1]): A k-truss $T_k = (V_{T_k}, E_{T_k})$ ($k \geq 2$) is a maximal connected subgraph of G such that for each edge $e \in T_k, sup(e, T_k) \geq k - 2$.

Definition 3 (Trussness): The trussness of an edge $e \in E(G)$ is defined as $\tau(e) = \max\{k : e \in E_{T_k}\}$.

Definition 4 (k-Class): The k-class of G is defined as $\{e : e \in E, \tau(e) = k\}$.

Definition 5 ($k_{\max}$-Truss): The $k_{\max}$-truss of G is defined as $\{e : e \in E, \tau(e) = k_{\max}\}$.

$k_{\max}$ denotes the maximum trussness of the edges in G.

There are very large graphs in the real world, e.g. Facebook's social network [13] can reach 2 billion vertices and 400 billion edges. In-memory graph algorithms cannot handle such large graphs. Therefore, external-memory graph algorithms are needed to handle such large graphs, and we formally define our problem in this paper below.

Problem definition. The objective of this study is to identify and maintain the $k_{\max}$-truss of a graph G. Recognizing that real-world graphs are dynamic, we explore strategies that allow for the maintenance of the $k_{\max}$-truss in dynamic graphs.

Example 1: Let us consider the graph G depicted in Fig. 2. The subgraph enclosed by the shaded region represents the $k_{\max}$-truss with a trussness of 4 for each edge. Conversely, the trussness for the edges outside the shaded region is 3. Hence, it is evident that $k_{\max}$ is 4. When an edge (v_1, v_5) is inserted in G, it can be observed that each edge in the subgraph induced by $\{v_1, v_2, v_3, v_4, v_5\}$ has a trussness of 5. Thus, the subgraph is now a $k_{\max}$-truss with $k_{\max} = 5$.

I/O model. Let M be the size of main memory and B be the disk block size ($B < M$). Files on a disk are organized in blocks and each block size is B bytes. An I/O operation will read/write one block of size B from disk/memory into memory/disk. The I/O cost of an algorithm represents the total number of read and write I/Os. Thus, reading/writing a piece of data of size N from/into disk requires (N/B) I/Os [14].

Currently, the semi-external model is widely adopted to design external-memory algorithms. The semi-external model assumes that the main memory can hold all nodes while cannot store all edges. For instance, previous studies have also used a semi-external model to address k-core related problems in an I/O-efficient manner [15], [16]. The semi-external model enables 128 GB memory of a typical server to efficiently process large graphs with up to 10 billion vertices, encompassing the majority of graphs present in publicly available real-world graph datasets.

Graph storage. In this paper, we store G on the disk in a manner similar to previous methods [15] [16]. We organize G by storing its adjacency lists, represented as $\{N_{v_1}(G), N_{v_2}(G), \cdots, N_{v_n}(G)\}$, in a sequential edge file on the disk. The node file stores information about the nodes, including their offsets and degrees. To load the neighbors of a node v_i, we first access the node file for its offset and degree and then load neighbors of v_i from the edge file.

III. $K_{\max}$-TRUSS ALGORITHMS FOR STATIC GRAPHS

In this section, we first review the existing solutions. Current methods predominantly rely on truss decomposition, necessitating the computation of the trussness for each edge in the graph. The state-of-the-art algorithm for truss decomposition under external memory is based on the peeling method, as proposed by [11]. They also proposed an advanced efficient I/O algorithm, named Top-Down, which can compute the $k_{\max}$-truss. However, the Top-Down algorithm is deficient in three aspects as mentioned in the introduction.

To address the above limitations, we introduce a series of novel semi-external algorithms for $k_{\max}$-truss computation. We first propose a basic algorithm, SemiBinary, which uses a binary search approach. Then, we present SemiGreedyCore, which utilizes a combination of k-core pruning and greedy strategies. Finally, we present SemiLazyUpdate, an algorithm utilizing a newly designed data structure, implemented with a lazy update strategy to minimize I/O overhead in updating edge information on the disk.

A. *The* SemiBinary *algorithm*

In this paper, we propose a semi-external algorithm utilizing the binary concept to efficiently identify the $k_{\max}$-truss of a given graph G, thus avoiding the costly process of repeatedly scanning G to generate the subgraph H. Specifically, our approach involves establishing a lower bound lb and an upper bound ub of $k_{\max}$. By conducting a careful binary search within the interval of $[lb, ub]$, our algorithm is capable of rapidly determining the value of $k_{\max}$.

Bounds of $k_{\max}$-truss. The previous lower bound theory presented in [10] states that $k_{\max} \geq \frac{\triangle_G}{m} + 2$, which is an instance of the extension of Nash-Williams' result [17] to trussness. However, this lower bound is too loose. To address this issue, we propose a tighter lower bound, which is further used in the SemiBinary algorithm to improve the efficiency of computing the $k_{\max}$-truss of a graph G.

Let $\triangle^k_{sup}(G)$ represent the set of triangles in G that contain edges with support k, *i.e.*, $\triangle^k_{sup}(G) = \{\triangle_{uvw} : \triangle_{uvw} \in T(G), \exists e \in E(\triangle_{uvw}), sup(e) = k\}$. We use $\S^k_{\triangle}(G)$ to denote the number of triangles in G that contain edges with support from 0 to k, *i.e.*, $\S^k_{\triangle}(G) = \sum_{i=0}^{k} |\triangle^i_{sup}(G)|$. Denote by $E^k_{sup}(G)$ the set of edges with support k in G, *i.e.*, $E^k_{sup}(G) = \{e : e \in E(G), sup(e) = k\}$. We use $\S^k_E(G)$ to denote the number of edges with support from 0 to k, *i.e.*, $\S^k_E(G) = \sum_{i=0}^{k} |E^i_{sup}(G)|$.

Lemma 1 (Lower Bound): Given an undirected graph G containing $k_{\max}$-truss, let $|E^0_{sup}(G)|$ be the total number of edges with support of 0 in G. Then $k_{\max} \geq 3\frac{\triangle_G}{m - |E^0_{sup}(G)|} + 2$. Furthermore, when edges with support less than k are removed from G, $k_{\max} \geq 3\frac{\triangle_G - \S^{k-1}_{\triangle}(G)}{m - \S^{k-1}_E(G)} + 2$.

All missing proofs of the Lemma and algorithmic complexity can be referenced in the full version of this paper [18].

Proof: Assume that the $k_{\max}$-truss is G itself, then $3\triangle_G = \sum_{e \in E} sup(e)$, each edge has a support is $k_{\max} - 2$, it follows that $m(k_{\max} - 2) = 3\triangle_G$. Otherwise, when the $k_{\max}$-truss is not G itself, the support of m edges cannot reach $(k_{\max} - 2)$. Note that edges with support 0 are not considered. We obtain that $(m - |E^0_{sup}(G)|)(k_{\max} - 2) \geq 3\triangle_G$ *i.e.*, $k_{\max} \geq 3\frac{\triangle_G}{m - |E^0_{sup}(G)|} + 2$. In addition to this, when edges with support less than k are removed from G, the number of triangles decreases $\sum_{i=0}^{k-1} |\triangle^i_{sup}(G)|$ and the number of edges decreases $\S^{k-1}_E(G)$, therefore, based on $(m - E^0_{sup}(G))(k_{\max} - 2) \geq 3\triangle_G$, it follows that $(m - \S^{k-1}_E(G))(k_{\max} - 2) \geq (3\triangle_G - 3\S^{k-1}_{\triangle}(G))$. Thus, $k_{\max} \geq 3\frac{\triangle_G - \S^{k-1}_{\triangle}(G)}{m - \S^{k-1}_E(G)} + 2$. □

Lemma 1 implies that $3\frac{\triangle_G}{m - |E^0_{sup}(G)|} + 2$ as a initial lower bound in our search for the $k_{\max}$-truss. Subsequently, when some edges are removed, the lower bound is dynamically adjusted to $3\frac{\triangle_G - \S^{k-1}_{\triangle}(G)}{m - \S^{k-1}_E(G)} + 2$.

Lemma 2 (Upper Bound): Given an undirected graph G containing $k_{\max}$-truss, we use the maximum support among all edges of the graph G as an upper bound, *i.e.*, $ub = max\{sup(e) : e \in E(G)\} + 2$.

Key idea of SemiBinary. The main idea behind SemiBinary is to apply a binary search in $[lb, ub]$ to find the exact value of $k_{\max}$. The algorithm tests if G contains a mid-truss where $mid = \frac{lb+ub}{2}$. If there exists a mid-truss in G, then let lb be $mid + 1$; otherwise, we set ub to $mid - 1$. To determine the presence of a mid-truss in G, we iteratively remove edges

Algorithm 1: SemiBinary

Input: $G = (V, E)$ in the disk
Output: The $k_{\max}$-truss of G

1 Compute $sup(e)$ of each edge in G with a semi-external method [19];
2 $lb \leftarrow 3\frac{\triangle_G}{m-|E^0_{sup}(G)|}+2$; $ub \leftarrow max\{sup(e) : e \in E(G)\} + 2$;
3 Sort all edges of G in non-decreasing order of support and store them in $\mathcal{T}_{edge}(G)$ (merge sort);
4 $pre(i) \leftarrow 0$ for all $0 \leq i \leq (ub + 1)$;
5 ComputePrefix$(E(G), pre, lb, ub)$;
6 **while** $lb \leq ub$ **do**
7 $\quad mid \leftarrow \lfloor (lb + ub)/2 \rfloor$; $lmid \leftarrow mid$;
8 $\quad$ Let H be the subgraph from $pre(mid)$ to $pre(ub + 1)$ in $\mathcal{T}_{edge}(G)$;
9 $\quad$ Compute $sup(e)$ of each edge in H with a semi-external method;
10 $\quad$ Sort all edges of H in non-decreasing order of their support (bin sort);
11 $\quad$ **while** $\exists e = (u, v)$ *of* H s.t. $sup(e) < mid - 2$ **do**
12 $\quad\quad (u, v) = \arg\min_{e \in E(H)} sup(e)$;
13 $\quad\quad$ Load $N_u(H)$ and $N_v(H)$ from disk;
14 $\quad\quad$ **for** $w \in N_u(H) \cap N_v(H)$ **do**
15 $\quad\quad\quad sup((u, w)) \leftarrow sup((u, w)) - 1$;
16 $\quad\quad\quad sup((v, w)) \leftarrow sup((v, w)) - 1$;
17 $\quad\quad\quad$ Reorder (u, w) and (v, w) according to their new support;
18 $\quad\quad$ Remove (u, v) from H;
19 $\quad$ **if** *not all edges in H are removed* **then**
20 $\quad\quad k_{\max} \leftarrow mid$; $lb \leftarrow 3\frac{\triangle_H - \S^{mid-3}_{\triangle}(H)}{|E(H)| - \S^{mid-3}_{E}(H)} + 2$;
21 $\quad\quad$ **if** $lb < mid + 1$ **then** $lb \leftarrow mid + 1$;
22 $\quad\quad mid \leftarrow \lfloor (lb + ub)/2 \rfloor$;
23 $\quad\quad lmid \leftarrow mid$;
24 $\quad\quad$ **goto** line 11;
25 $\quad$ **else**
26 $\quad\quad ub \leftarrow mid - 1$;
27 Output the edges in H whose trussness is $k_{\max}$ as $k_{\max}$-truss;
28 **Procedure** ComputePrefix(E, pre, lb, ub)
29 $cnt(i) \leftarrow 0$ for all $0 \leq i \leq (ub + 1)$;
30 **For** each $e \in E$ **do** $cnt(sup(e)) \leftarrow cnt(sup(e)) + 1$;
31 **For** $i = 1$ to $(ub + 1)$ **do** $pre(i) \leftarrow pre(i - 1) + cnt(i - 1)$;

with support less $mid - 2$ until all remaining edges have the support of at least $mid - 2$. If all edges are removed, then G lacks a mid-truss. Otherwise, the remaining edges form a mid-truss.

Detailed implementation of algorithm. Algorithm 1 shows the pseudo-code of SemiBinary. At the beginning, we compute the support of each edge in G, which makes it easy to get the upper and lower bounds through the by-products. After that, we apply an external memory merge sort algorithm to sort the edges of G in non-decreasing order of support and store them in $\mathcal{T}_{edge}(G)$ (line 1-3). Then we use $pre(i)$ to record the starting position of a batch of edges in $\mathcal{T}_{edge}(G)$ with support i (line 28-31).

Subsequently, the algorithm invokes a binary search procedure to compute the $k_{\max}$-truss. We form a subgraph H from the edges with support not less than $mid - 2$ in $\mathcal{T}_{edge}(G)$. Since the edges in $\mathcal{T}_{edge}(G)$ are kept in order, it is sufficient to read them sequentially. We compute the support of each edge in H, and sort all the edges in non-decreasing order of their support with the bin sort method. The sorted edges are then stored in $\mathcal{A}_{disk}$, similar to how the sorted degree array is kept in [20] (line 8-10). We iteratively delete all edges with support less than $mid - 2$ in the $\mathcal{A}_{disk}$. After removing edge (u, v), the support of edges forming a triangle with (u, v) must be decremented and the position of these edges will be updated in $\mathcal{A}_{disk}$. Instead of physically removing edge (u, v) from H, we simply move the pointer in $\mathcal{A}_{disk}$ to the next edge with the lowest support (line 11-17). If a mid-truss exists in H, it is updated directly on H without having to reselect the edges from $\mathcal{T}_{edge}(G)$ to generate the subgraph (line 18-24). Otherwise, we would have to recompute the subgraph after updating ub to identify the presence of the mid-truss (line 25-26).

Example 2: Consider the graph G in Fig. 2. We observe that $\triangle_G$ is 18 and the max support is 4, thus SemiBinary sets $lb = 4$ and $ub = 6$. In the binary search phase, SemiBinary initializes mid to 5 and scans $\mathcal{T}edge(G)$ in sequence to identify edges with support no less than 3, thereby generating a subgraph H consisting of these edges such as $\{(v_2, v_3), (v_2, v_4), (v_3, v_4), (v_4, v_5), (v_5, v_8)\}$. Subsequently, SemiBinary aims to locate a 5-truss (*i.e.*, $mid = 3$) in subgraph H by removing edges with support smaller than 3 iteratively. There are no remaining edges, so there is no 5-truss. Therefore, SemiBinary updates $ub = 4$ and $mid = 4$ for the next iteration. The algorithm rescans $\mathcal{T}edge(G)$ in sequence to identify edges with support no less than 2, thereby generating a subgraph H. All edges with support less than 2 in the subgraph H are first removed, and then the result is a 4-truss represented by the shaded area. Following this stage, SemiBinary terminates and $k_{\max} = 4$.

Theorem 1: The I/O complexity of Algorithm 1 is $O(max(\frac{|E(G)|d_{\max}(G)}{B}, log_2^{ub}|E(H)|d_{\max}(H)))$, and the CPU time complexity of Algorithm 1 is $O(m^{1.5})$ [11]. Algorithm 1 only requires $O(n)$ memory.

Proof: Only arrays of node-related information are held in memory, hence, its memory overhead is $O(n)$. The I/O overhead consists of three main components, 1) computing the support of all edges, 2) sorting these edges, and 3) deleting edges to update the information of other edges. First, computing the support of each edge in G requires $O(\frac{2|E(G)|}{B} + \sum_{x \in V}(\frac{\sum_{y \in N(x)} N(y)}{B})) \leq O(\frac{|E(G)|(d_{\max}(G))}{B})$ I/Os. Second, sorting edges of G takes $O(\frac{|E(G)|}{B} \log_{\frac{M}{B}} \frac{|E(G)|}{B})$, as it takes $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ I/Os for sorting N numbers using the external sorting algorithm [21]. Third, in the binary search process, it firstly generates a subgraph H saved in the disk, it takes $O(\frac{|E(H)|}{B} \log_{\frac{M}{B}} \frac{|E(H)|}{B})$ I/Os. Then, computing the support of each edge in H requires $O(\frac{|E(H)|(d_{\max}(H))}{B})$ I/Os. Sorting edges in H by support in bin sort method will take $O(|E(H)|)$ I/Os. Finally, it needs to determine whether a mid-truss exists in H. Therefore, in each iteration (line 10-16), it takes $O(\frac{d(x)+d(y)}{B} + |N(x) \cap N(y)|)$ I/Os. In the worst case, it needs to traverse all the edges in H, which causes $O(|E(H)|(\frac{d(x)+d(y)}{B} + |N(x) \cap N(y)|)) \leq O(E(H)d_{\max}(H))$ I/Os. In summary, The dominant I/O overheads arise from computing the support of each edge in the entire graph and removing each edge during the binary search. Since it is not possible to judge the size of the subgraph H generated during the binary search with respect to the size of the entire graph, the total I/O overhead is $O(max(\frac{|E(G)|d_{\max}(G)}{B}, log_2^{ub}|E(H)|d_{\max}(H)))$. Meanwhile, $ub = max\{sup(e) : e \in E(G)\} + 2$. As a result, the CPU time complexity for Algorithm 1 is the time complexity of truss decomposition $O(m^{1.5})$ [11]. □

Remarks. Although the support of some edges is repeatedly computed, our algorithm is designed to be external memory friendly. If an additional external memory-based index table is introduced to eliminate the repeated computation, more I/O overhead will be incurred, which results in lower performance of the algorithm.

B. The SemiGreedyCore algorithm

Note that the upper bound and the lower bound in Algorithm 1 are not tight, resulting in several implications. Firstly, the looser upper bound leads to more iterations in the binary search. Secondly, the looser lower bound leads to a large number of useless computations. To address the aforementioned challenges, we propose two optimizations that result in tighter upper and lower bounds. These refinements contribute to the enhanced performance of the algorithm. We begin with an introduction to the concept of k-core.

Definition 6 (k-Core [22]): A k-core is a maximal subgraph of G, denoted by G_k, such that for $\forall v \in V_{G_k}, d_v(G_k) \geq k$.

Definition 7 (Coreness): The coreness of a vertex $v \in V(G)$ is defined as $core(v) = \max\{k : v \in V(G_k)\}$.

We use the notation $c_{\max}$ to represent the maximum coreness of a vertex in the graph G.

Core-based reduction. In order to improve the efficiency of the algorithm, it is crucial to minimize the number of nodes that are not included in the $k_{\max}$-truss. It is worth noting that a k-truss is a $(k-1)$-core, whereas a $(k-1)$-core is not necessary to be a k-truss. Hence, based on this relationship, we can effectively filter out nodes with lower coreness during the process of searching for the $k_{\max}$-truss. Moreover, the coreness can assist in designing a tighter upper bound, as demonstrated in Lemma 3. Based on the above observations, we first perform a core decomposition to compute the coreness of each node in G in a semi-external manner [15].

Lemma 3 (Upper Bound): Given a graph G, let $ub_{(u,v)}$ be the upper bound of edge (u, v) in G. We have $ub_{(u,v)} = min(core(u), core(v)) + 1$. Thus, the upper bound of the $k_{\max}$-truss in G is that $ub = max\{ub_{(u,v)} : (u, v) \in E(G)\}$.

Proof: If an edge (u, v) is in the $k_{\max}$-truss, $core(u) = k_1$, $core(v) = k_2$ and $\tau_{(u,v)} = k_3$, we have $k_3 \leq min(k_1, k_2) + 1$. Assuming that, on the contrary, we have $k_3 > min(k_1, k_2) + 1$, *i.e.*, $k_3 \geq min(k_1, k_2) + 2$. According to the Definition 2, edge (u, v) has at least $min(k_1, k_2)$ common neighbors, at this point vertex u has at least $min(k_1, k_2) + 1$ neighbours. We have $core(u)' = min(k_1, k_2) + 1 > k_1$, which does not match the original $core(u) = k_1$. Therefore, $k_3 > min(k_1, k_2) + 1$ leading to a contradiction. □

Greedy strategy for $k_{\max}$. Initially, our approach involves identifying the local $k'_{\max}$-truss. For this purpose, we adopt a greedy strategy of selecting nodes with the highest coreness $c_{\max}$, which collectively form the subgraph known as the $G_{c_{\max}}$. We employ the $G_{c_{\max}}$ as a local graph for the extraction of the local $k'_{\max}$-truss. The rationale behind our greedy selection of the $c_{\max}$-core lies in the strong correlation between the $G_{c_{\max}}$ and the $k_{\max}$-truss. The relationship between the $G_{c_{\max}}$ and the $k_{\max}$-truss can be analyzed in two different cases.

Case-1 ($k_{\max}$-truss $\subseteq G_{c_{\max}}$): The $G_{c_{\max}}$ of a graph G contains all nodes and edges of the $k_{\max}$-truss. Moreover, due to the fact that $k_{\max}$-truss is equivalent to a $(k_{\max} - 1)$-core, it follows that $k_{\max} - 1 = c_{\max}$.

Case-2 ($k_{\max}$-truss $\subsetneq G_{c_{\max}}$): The fact that there are nodes and edges in $k_{\max}$-truss that are not in $G_{c_{\max}}$ means that they exist in k-core ($k < c_{\max}$). It can be deduced that $k_{\max} - 1 < c_{\max}$.

In the study by Li et al. [16], it was demonstrated that the $G_{c_{\max}}$ is often smaller in size compared to the original graph, as evidenced by their experimental results. Therefore, we leverage the SemiBinary approach to compute the local $k'_{\max}$-truss on the $G_{c_{\max}}$, which incurs less overhead compared to performing the computation on the original graph. However, it is not guaranteed that the local $k'_{\max}$-truss corresponds to the $k_{\max}$-truss of the original graph. The presented Case-2 demonstrates that $G_{c_{\max}}$ comprises only a subset of vertices from $k_{\max}$-truss, leading to a local $k'_{\max}$ that is smaller than $k_{\max}$. Consequently, vertices that do not satisfy Lemma 4 are eliminated, enabling rapid identification of $k_{\max}$-truss.

Lemma 4: Given an undirected graph G, we have $V_H = \{u \in V | core(u) \geq k'_{\max} - 1\}$, H is a subgraph composed of V_H, thus, $k_{\max}$-truss $\subseteq H$.

Proof: For Case-1, when $k'_{\max} = k_{\max}$, it follows that H is also a $G_{c_{\max}}$, and the $k_{\max}$-truss must be in H. In Case-2, where $k'_{\max} < k_{\max}$, nodes with a coreness less than $k'_{\max} - 1$ are definitely not part of the $k'_{\max}$-truss. Therefore, these nodes must not be part of the $k_{\max}$-truss. In contrast, the $k_{\max}$-truss is in the subgraph H composed of nodes with a coreness of no less than $k'_{\max} - 1$. □

By the greedy strategy and the proof of Lemma 1, we can obtain a tighter lower bound.

Lemma 5 (Lower Bound): Given an undirected graph G, let $k'_{\max}$-truss be the local maximum k-truss in $G_{c_{\max}}$. We have $lb \geq k'_{\max}$.

Detailed implementation of algorithm. Algorithm 2 shows the pseudo-code of SemiGreedyCore. Firstly, we conduct core decomposition using a semi-external method [15] to extract the $G_{c_{\max}}$ from the graph G (line 1-3). Subsequently, we compute the support of each edge in the $G_{c_{\max}}$ and employ an external merge sort algorithm to arrange these edges in non-decreasing order of support. These sorted edges are then stored in $\mathcal{T}edge(G_{c_{\max}})$. Additionally, we use $pre(i)$ to store the starting position of a batch of edges with the same support i in $\mathcal{T}edge(G_{c_{\max}})$. By applying Lemma 1 and Lemma 3, we readily obtain the values of lb and ub (line 4-8).

The local $k'_{\max}$-truss can be found from $G_{c_{\max}}$ in the same way as line 6-26 of Algorithm 1. It is important to note that the local $k'_{\max}$-truss from $G_{c_{\max}}$ may not be the $k_{\max}$-truss of G, but the $k'_{\max}$ is very close to the $k_{\max}$. On this basis, the lower bound lb is updated (line 9-10). Subsequently, we need to select those nodes whose coreness is no less than $lb - 1$. These nodes form the subgraph H'. We also compute the support of each edge in H', then sort all the edges in non-decreasing order of their support with bin sort. The sorted edges are then stored in $\mathcal{A}_{disk}$ (line 11-14). In the end, we iteratively remove the edges in H' with support less than or equal to $lb - 2$. When removing (u, v), we also decrement the support of all other edges that form a triangle with (u, v), and update their new positions in the $\mathcal{A}_{disk}$. This iteration

Algorithm 2: SemiGreedyCore

Input: $G = (V, E)$ in the disk
Output: The $k_{\max}$-truss of G

1 Compute coreness of each node in G [15];
2 $V_{c_{\max}} \leftarrow \{v \in V | core(v) = c_{\max}\}$;
3 Denote by $G_{c_{\max}}$ the subgraph of G induced by $V_{c_{\max}}$;
4 Compute $sup(e)$ of each edge in $G_{c_{\max}}$ with a semi-external method;
5 Sort all edges of $G_{c_{\max}}$ in non-decreasing order of their support and store them in $\mathcal{T}_{edge}(G_{c_{\max}})$ (merge sort);
6 $lb \leftarrow 3\frac{\triangle_{G_{c_{\max}}}}{|E(G_{c_{\max}})|-E^0_{sup}(G_{c_{\max}})} + 2$; $ub \leftarrow c_{max} + 1$;
7 $pre(i) \leftarrow 0$ for all $0 \leq i \leq (ub + 1)$;
8 ComputePrefix $(E(G_{c_{\max}}), pre, lb, ub)$;
9 line 6-26 of Algorithm 1; /* get local $k'_{\max}$-truss from $G_{c_{\max}}$ */;
10 $lb \leftarrow k'_{\max}$;
11 $V_{new} \leftarrow \{v \in V | core(v) \geq lb - 1\}$;
12 Denote by H' the subgraph of G induced by V_{new};
13 Compute $sup(e)$ of each edge in H' with a semi-external method;
14 Sort all edges of H' in non-decreasing order of their support (bin sort);
15 **while** $\exists e = (u, v)$ *of* H' s.t. $sup(e) \leq lb - 2$ **do**
16   $(u, v) = \arg\min_{e \in E(H)} sup(e)$;
17   Load $N_u(H')$ and $N_v(H')$ from disk;
18   **for** $w \in N_u(H') \cap N_v(H')$ **do**
19     $sup((u, w)) \leftarrow sup((u, w)) - 1$;
20     $sup((v, w)) \leftarrow sup((v, w)) - 1$;
21   Reorder (u, w) and (v, w) according to their new support;
22   Remove (u, v) from H';
23 **if** *not all edges in* H' *are removed* **then**
24   $lb \leftarrow lb + 1$;
25   **goto** line 15;
26 $k_{\max} \leftarrow lb$;
27 Output the edges in H' whose trussness is $k_{\max}$ as $k_{\max}$-truss;

Algorithm 3: SemiLazyUpdate

Input: $G = (V, E)$ in the disk
Output: The $k_{\max}$-truss of G

1 lines 1-8 of Algorithm 2;
2 **while** $lb \leq ub$ **do**
3   $mid \leftarrow \lfloor (lb + ub)/2 \rfloor$; $lmid \leftarrow mid$;
4   Let H be the subgraph extracted from $\mathcal{T}_{edge}(G_{c_{\max}})$;
5   Compute $sup(e)$ of each edge in H with a semi-external method;
6   Initialize linear-heap $(lheap)$ and dynamic-heap $(dheap)$;
7   **while** $\exists e = (u, v)$ *of* H s.t. $sup(e) < mid - 2$ **do**
8   DeleteEdgeKernal $(H, lheap, dheap)$;
9   Remove e from H;
10   **if** *not all edges in* H *are removed* **then**
11     $k_{\max} \leftarrow mid$; $lb \leftarrow 3\frac{\triangle_H - \S^{mid-3}_{\triangle}(H)}{|E(H)| - \S^{mid-3}_{E}(H)} + 2$;
12   **if** $lb < mid + 1$ **then** $lb \leftarrow mid + 1$;
13     $mid \leftarrow \lfloor (lb + ub)/2 \rfloor$;
14     $lmid \leftarrow mid$;
15   **goto** line 7;
16   **else**
17     $ub \leftarrow mid - 1$;
18 lines 10-13 of Algorithm 2;
19 Initialize linear-heap $(lheap)$ and dynamic-heap $(dheap)$;
20 **while** $\exists e = (u, v)$ *of* H s.t. $sup(e) \leq lb - 2$ **do**
21   DeleteEdgeKernal $(H, lheap, dheap)$;
22   Remove e from H;
23 **if** *not all edges in* H *are removed* **then**
24   $lb \leftarrow lb + 1$;
25   **goto** line 15;
26 $k_{\max} \leftarrow lb$;
27 Output the edges in H whose trussness is $k_{\max}$ as $k_{\max}$-truss;

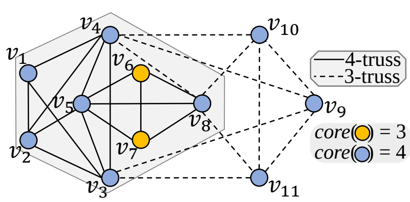


Fig. 3. The running example

continues until all edges in H' with support less than or equal to $(lb-2)$ have been removed. In this way, we find the $k_{\max}$-truss in G (line 15-26).

Example 3: Consider the graph G in Fig. 3. The coreness of the orange node is 3 and the coreness of the blue node is 4. It is easy to see that the subgraph formed by the blue nodes is $G_{c_{\max}}$. $k'_{\max}$-truss (*i.e.*, $k'_{\max} = 4$) computed on $G_{c_{\max}}$ using SemiBinary is the subgraph formed by $\{v_1, v_2, v_3, v_4, v_5\}$ (line 1-9). Meanwhile, lb was updated to $k'_{\max}$ (*i.e.*, $lb = 4$). Next, all nodes with a coreness no less than $lb - 1$ were selected from the G, as shown in the shaded area (line 10-14). Iterate to remove all edges with support less than $lb - 2$. No edges remain and the final $k_{\max}$ is 4. So the $k_{\max}$-truss is the shaded area covered subgraph.

Theorem 2: Let l be the number of iterations of core decomposition [15]. The I/O complexity of Algorithm 2 is $O(max(log_2^{ub}|E(G_{c_{\max}})|d_{\max}(G_{c_{\max}}), |E(H')|d_{\max}(H'))$, and the CPU time complexity of Algorithm 2 is $O(|E_{H'}|^{1.5})$. Algorithm 2 requires $O(n)$ memory.

Proof: Algorithm 2 has the same space complexity as Algorithm 1. I/O overhead mainly consists of three components: 1) Core decomposition. 2) Discovering local $k'_{\max}$-truss from $G_{c_{\max}}$. 3) Finding $k_{\max}$-truss in the subgraph H' generated after updating the lower bound with $k'_{\max}$. Firstly, it performs core decomposition, which takes $O(\frac{l \times (m+n)}{B})$ I/Os. Secondly, the local $k'_{\max}$-truss is discovered within $G_{c_{\max}}$, and the I/O overhead is computed in a similar manner as in Algorithm 1, with a complexity of $O(log_2^{ub}|E(G_{c_{\max}})|d_{\max}(G_{c_{\max}}))$ I/Os. Thirdly, based on the local $k'_{\max}$, it constructs the subgraph H', then iteratively remove the unsatisfied edges. This process will take $O(\frac{|E(H')|(1+d_{\max}(H'))}{B} + \frac{|E(H')|}{B}\log_{\frac{M}{B}}\frac{|E(H')|}{B} + |E(H')|(\frac{d_{\max}(H')}{B} + d_{\max}(H')))) \leq O(|E(H')|(d_{\max}(H'))))$ I/Os. Since the $G_{c_{\max}} \subseteq H'$, the I/O overhead associated with $G_{c_{\max}}$ has a log_2^{ub} factor. Therefore, it is difficult to determine whether the I/O overhead incurred at $G_{c_{\max}}$ or H' is greater. As a result, the I/O complexity of Algorithm 2 is $O(max(log_2^{ub}|E(G_{c_{\max}})|d_{\max}(G_{c_{\max}}), |E(H')|d_{\max}(H'))$. The largest subgraph for performing triangle enumeration is H', thus the CPU time complexity of Algorithm 2 is $O(|E(H')|^{1.5})$ [11]. □

Note that Algorithm 2 is pruned by the k-core based technique, which can significantly reduce the useless nodes. Consequently, Algorithm 2 requires triangle listing not in G but in subgraph $G_{c_{\max}}$ compared to Algorithm 1, which greatly reduces I/O overhead. Besides, the tighter lower bound and upper bound also contribute significantly to performance improvement.

C. *The* SemiLazyUpdate *algorithm*

Algorithms analysis. We notice that the deletion of edge is a frequent operation in both Algorithm 2 and Algorithm 1. The tight inter-connections among edges via triangles cause a ripple effect when an edge is deleted. The support of the two edges forming a triangle with the deleted edge needs to be updated, resulting in changes to their positions in $\mathcal{A}_{disk}$. For example, consider a scenario where an edge (u, v) has

Algorithm 4: DeleteEdgeKernal

Input: $G = (V, E)$, $lheap$ and $dheap$

1 $(u, v) = \arg\min_{e \in E(G)} sup(e)$;
2 Load $N_u(G)$ and $N_v(G)$ from disk;
3 **for** $w \in N_u \cap N_v$ **do**
4   **if** $\exists (u, w) \notin dheap$ **then**
5   **if** $sup((u, w)) > sup((u, v))$ **then**
6   Take out (u, w) from $lheap$;
7   Put (u, w) into $dheap$;
8       $sup((u, w)) \leftarrow sup((u, w)) - 1$ in $dheap$;
9   **else**
10   **if** $dheap.getSup((u, w)) \neq sup((u, v))$ **then**
11       $sup((u, w)) \leftarrow sup((u, w)) - 1$ in $dheap$;
12   Adjust position of (u, w) in $dheap$;
13   Replace (u, w) with (v, w) and repeat line 4-12.
14 **if** $dheap.size() > capacity$ **then**
15   **for** $i = 1$ *to capacity* **do**
16     $(u, v) \leftarrow dheap.top()$; $dheap.pop()$;
17   Insert (u, v) into the position of $sup((u, v))$ in $lheap$;
18 **while** $dheap.size() > 0$ *and lowest sup of* $lheap \geq sup(dheap.top())$ **do**
19   $(u, v) \leftarrow dheap.top()$; $dheap.pop()$;
20   Insert (u, v) into front of $lheap$;

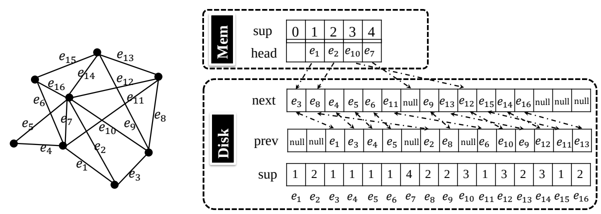


Fig. 4. Demostration of linear-heap on the left graph

higher support compared to some adjacent edges, which have lower support. In the traditional approach, deleting adjacent edges requires frequent access to (u, v), updating its support and position on disk, and incurring intolerable I/O overhead.

To address this issue and reduce the costly overhead of edge deletion, we have devised an efficient data structure based on a combination of disk and memory.

The I/O-optimal structure: LHDH. The implementation of this structure involves two components: 1) a disk-based linear-heap, inspired by [23], which stores all edges in non-decreasing order of support, and 2) a memory-based dynamic-heap structure that handles frequently updated edges. By combining these two structures, we achieve lazy updates in the edge deletion process. This approach eliminates the need for triggering an I/O overhead for each update. Next, we will describe these two structures in detail.

linear-heap. Edges are efficiently stored in a linear-heap, arranged in increasing order of support, as illustrated on the left side of Fig. 5. To optimize the loading and writing process, edges with the same support are linked together in a doubly-linked format, as shown in Fig. 4. Given that the maximum support of edges is less than n, it becomes feasible to retain the information of the head node in a doubly-linked table in memory. This arrangement enables effective access to individual edges and efficiently writes it back to disk.

dynamic-heap. The dynamic heap is based on a min-heap. The frequently updated edges in this heap are ordered by support, with lower support edges placed at the top and higher support edges at the bottom. When an edge is updated, its support is decreased by one, and its position is dynamically

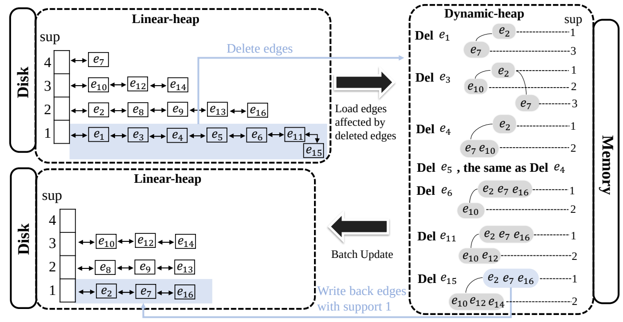


Fig. 5. The LHDH structure

adjusted upwards in the heap based on the heap ordering. As a result, the edge with the smallest support is consistently popped from the top of the heap. In addition, there is a limit to the capacity of dynamic-heap to hold edges, and if this capacity is exceeded, we write a certain number of edges at the top of the dynamic-heap back into linear-heap.

Detailed implementation of algorithm. Our lazy update algorithm is shown in Algorithm 3. The process of applying the data structure is outlined in Algorithm 4. Initially, all edges are stored in the linear heap, and the algorithm iteratively deletes the edge with the smallest support from the linear heap. During edge deletion, the dynamic heap structure is utilized to retain edges with high support that are frequently updated, allowing multiple updates to be performed in memory and reducing the overhead of writing back to disk multiple times.

Specifically, to remove an edge (u, v) from the $lheap$, a linear heap, we also need to update the neighboring edges that form a triangle with this edge. If the edge (u, w) with higher support is not present in the $dheap$, a dynamic heap, it is removed from the $lheap$ and placed into the $dheap$, and its support is updated (line 4-8). Otherwise, it is directly updated in the $dheap$. In the case where $sup(u, w) = sup(u, v)$, this means that the edge (u, w) is about to be deleted, pending a batch write back to $lheap$ (line 10-13). It is worth noting that when the number of edges present in the $dheap$ exceeds the capacity limit, the first $capacity$ edges are continually removed from the top of the $dheap$, and based on the support of these edges, these edges are written to their corresponding locations in the $lheap$ (line 14-17). Finally, if the support of the top edge of the $dheap$ is no greater than the smallest support in the $lheap$, then these edges need to be written back to the $lheap$ from the $dheap$ (line 18-20).

Example 4: Consider the graph G in Fig. 4, the process of deleting edges is shown in Fig. 5. We aim to delete edges located in $lheap$ with minimum support (*i.e.*, $sup = 1$). Firstly, delete e_1, e_2 and e_7 are added to $dheap$ with their support updated. Next, the algorithm deletes e_3, and since $sup(e_2) = sup(e_3)$, only e_{10} is added to $dheap$ and its support is set as $sup(e_{10}) - 1$. When we delete e_4, e_7 is already in $dheap$ and does not need to be loaded from $lheap$. Instead, we update the support of e_7 and adjust its position upwards in $dheap$. Finally, these edges with support 1 in $dheap$ are written back to $lheap$. It is worth noting that since e_{10}, e_{12}, e_{14} are still in $dheap$, their positions in $lheap$ have not been updated.

Let $Cost$ be the number of I/O operations triggered when updating an edge that is not in $dheap$.

Theorem 3: The memory costs of Algorithm 3 are characterized by $O(n + capacity)$. The I/O complexity of 3 is $O(max(log_2^{ub}|E(G_{c_{\max}})|Cost, |E(H')|Cost))$, and the CPU time complexity of 3 is $O(|E(H')|^{1.5})$ [11].

Proof: Algorithm 3 has a space complexity of $O(n + capacity)$ due to the allocation of memory space for the $dheap$. The I/O complexity is optimized compared to Algorithm 2. It has been observed that $Cost$ is significantly smaller than $|N(x) \cap N(y)|$, indicating that $0 < Cost \ll |N(x) \cap N(y)|$. The CPU time complexity is equivalent to Algorithm 2. □

Remarks. Given the hierarchical structure nature of k-truss, our algorithms designed for $k_{\max}$-truss computation may also be used to compute models based on hierarchical structure. One such example is the k-(r, s)-nucleus decomposition, representing a generalization of truss decomposition. The k-(r, s)-nucleus of G is defined such that each r-clique is contained in at least k s-cliques, where $1 \leq r < s$ [24]. By extending the techniques we proposed in this paper, the problem of computing the maximum k-(r, s)-nucleus in large graphs under the semi-external model also can be efficiently solved. This indicates that our proposed techniques provide an important reference for solving similar problems.

Discussions. We observe that Zhang *et al.* [25] implemented the $k_{\max}$-truss computation on GPU. However, our proposed techniques are very different from those proposed in [25]. The detailed discussion is presented below.

First, [25] proposed k-clique technique to early terminate the computation. In contrast, we propose a k-core based technique to obtain a tight lower bound for pruning nodes of graph G. The advantages are that k-core has a lower computational cost, and the tight lower bound has a stronger pruning ability. Besides, [25] proposed an adaptive step size pruning technique, the number of iterations of this algorithm is linearly related to $k_{\max}$. However, we propose a binary search approach where the number of iterations is linear with $\log k_{max}$, thus greatly reducing the I/O overhead, compared to the approach in [25].

Second, The lower bounds we propose are mainly used to prune the size of the subgraphs that need to be loaded into memory. The high performance of [25] mainly depends on being an in-memory computation algorithm, which allows to access the global graph information efficiently, and thus it is not necessary to use the lower bounds in [25]. However, in our external memory algorithm, tighter lower bounds are proposed to determine whether the edges are to be loaded into the main memory, which greatly reduces the I/O consumption.

IV. $k_{\max}$-TRUSS MAINTENANCE IN DYNAMIC GRAPHS

In this section, we propose a novel approach to dynamically maintain the $k_{\max}$-truss in a semi-external setting. In this problem, we only have information about the edges in the $k_{\max}$-truss, and no information is provided for the other edges. When an edge is added or removed from the graph, we need to recompute the $k_{\max}$-truss, which results in substantial I/O overhead. To address this challenge, we introduce a new technique for maintaining the $k_{\max}$-truss efficiently.

Before introducing our algorithm, we present the conclusions, on which these efficient algorithms are designed.

Algorithm 5: Deletion

```
1  Delete (u, v) from k_max-truss;
2  Update d_u and d_v;
3  if u ∈ k_max-truss and v ∈ k_max-truss then
4      Queue ← ∅;
5      Load N_u(k_max-truss) and N_v(k_max-truss);
6      for w ∈ N_u(k_max-truss) ∩ N_v(k_max-truss) do
7          sup((u, w)) ← sup((u, w)) − 1;
8          if sup((u, w)) < k_max − 2 then
9              Queue ← (u, w);
10         Repeat lines 6-8 for (v, w);
11     while Queue ≠ ∅ do
12         (x, y) ← Queue.pop();
13         Load N_x(k_max-truss) and N_y(k_max-truss);
14         for z ∈ N_x(k_max-truss) ∩ N_y(k_max-truss) do
15             if sup((x, z)) = k_max − 2 and (x, z) not be visited then
16                 Queue ← (x, z);
17             Repeat lines 15-16 for (y, z);
18         sup((u, v)) ← sup((u, v)) − 1;
19         Remove (u, v) from k_max-truss;
20     if k_max-truss = ∅ then
21         k_max ← k_max − 1;
22         Update coreness of each node in G [15];
23         V_new ← {v ∈ V | core(v) ≥ k_max − 1};
24         Denote by H the subgraph induced by V_new;
25         Compute sup(e) of each edge in H with a semi-external method;
26         Repeat lines 19-26 of Algorithm 3 to get new k_max-truss;
```

Lemma 6: If an edge is inserted into (deleted from) graph G, the trussness for any $e \in E(G)$ may increase (decrease) by at most 1 [26].

A. Edge Deletion

In Lemma 6, after the deletion of an edge, we can conclude that the trussness of any edge $e \in E(k_{\max}\text{-}truss)$ will decrease by at most 1. This implies that the old trussness of edges in the $k_{\max}$-truss serves as upper bounds for their new trussness.

Lemma 7: The $k_{\max}$-truss of G may be updated when deleting (u, v) only if both u and v are part of the $k_{\max}$-truss.

Proof: If edge (u, v) is not in $k_{\max}$-truss, it must not be in $k_{\max}$-class, so the deletion of (u, v) will have no effect on $k_{\max}$-truss. □

By Lemma 7, $k_{\max}$-truss may be updated only if both u and v are contained in $k_{\max}$-truss.

Let $com_{(u,v)}$ be the common neighbors of u and v, *i.e.*, $com_{(u,v)} = \{w : w \in N_u(G) \cap N_v(G)\}$. Let $E_{com_{(u,v)}}$ be the set of edges between $com_{(u,v)}$ and (u, v), *i.e.*, $E_{com_{(u,v)}}$ = $\{(x, y) : x \in com_{(u,v)}, y \in \{u, v\}\}$.

Lemma 8: After deleting (u, v), $k_{\max}$-truss will be updated only if there is at least one edge $e \in E_{com_{(u,v)}}$ with support less than $k_{\max} - 2$.

Proof: If the support of an edge $(x, y) \in E_{com_{(u,v)}}$ is less than $k_{\max} - 2$, thus (x, y) will be deleted from the original $k_{\max}$-truss, resulting in an update of the $k_{\max}$-truss. □

Detailed implementation of algorithm. Our main algorithm is outlined in Algorithm 5. We need to update the degree of the vertex first when deleting an edge (u, v). According to Lemma 7, we only need to consider the case where both nodes u and v are in the $k_{\max}$-truss (lines 1-3). We initialize a queue Q (line 4) to store edges whose deletion of (u, v) leads to neighboring edges with support less than $k_{\max} - 2$, causing changes in the original $k_{\max}$-truss (lines 4-10). Subsequently, we employ the peeling method to iteratively remove edges from the queue Q while collecting edges with new support less than $k_{\max}-2$ in a breadth-first search manner (lines 11-19). If not all edges in $k_{\max}$-truss are deleted, some edges with support no less than $k_{\max}-2$ still form $k_{\max}$-truss; otherwise, $k_{\max}$-truss vanishes. In the latter case, we need to recompute $(k_{\max}-1)$-truss, as an edge deletion can only decrease the maximum trussness by 1 (Lemma 6). Additionally, due to the deletion of edges, the core values of the nodes in graph G may change. Thus, based on Lemma 4, we utilize the core values to prune out useless nodes and then invoke Algorithm SemiLazyUpdate to find $(k_{\max}-1)$-truss on the refined subgraph (lines 20-26).

Example 5: Consider the graph G in Fig. 2. Suppose that we delete an edge (v_2, v_5), and we have $k_{\max} = 4$ with the $k_{\max}$-truss being the subgraph covered with the shaded region. First, the algorithm computes $E_{com_{(v_2,v_5)}}$,*i.e.*, $\{(v_2, v_3), (v_2, v_4), (v_3, v_5), (v_4, v_5)\}$. Since the deletion of (v_2, v_5), $sup((v_3, v_5))$ and $sup((v_4, v_5))$ both become 1. These two edge no longer belong to $k_{\max}$-truss, which is then composed of the subgraph formed by the nodes $\{v_1, v_2, v_3, v_4\}$ and the subgraph formed by $\{v_5, v_6, v_7, v_8\}$.

B. Edge Insertion

Here we discuss the case of edge insertion. A new edge (u, v) inserted into graph G results in an increase in the trussness of any $e \in E(G)$ by at most 1, according to Lemma 6, which may result in edges with trussness $k_{\max}-1$ becoming part of $k_{\max}$-truss. As a result, Lemma 7 does not apply to the case of edge insertion.

We define the *k-level triangles* of an edge in $k_{\max}$-truss.

Definition 8 (k-level triangles): Let (u, v) be an edge and $k \geq 2$. We define the set of *k-level* triangles containing (u, v) as $\triangle^k_{(u,v)} = \{\triangle_{uvw} : min\{sup((u, w)), sup((v, w))\} \geq k - 2\}$. The number of triangles in this set is denoted by $|\triangle^k_{(u,v)}|$.

Lemma 9: For an edge (u, v) to be inserted, $k_{\max}$-truss will be updated if (1) edge $(u, v) \in E(k_{\max}\text{-truss})$. Or (2) u and v are not both in $k_{\max}$-truss, and $min\{sup((u, v)) + 2, min(core(u), core(v)) + 1\} \geq k_{\max}$.

Proof: First, we consider the case (1). If $|\triangle^{k_{\max}+1}_{(u,v)}| < k_{\max} - 1$, (u, v) is added to the $k_{\max}$-truss, but $k_{\max}$ does not become $k_{\max}+1$. If $|\triangle^{k_{\max}+1}_{(u,v)}| \geq k_{\max}-1$, this may lead to the generation of $(k_{\max}+1)$-truss from $k_{\max}$-truss.

Second, we consider the case (2). Even if $(u, v) \notin E(k_{\max}\text{-truss})$, the insertion of edge (u, v) into $(k_{\max}-1)$-truss may result in some edges becoming part of $k_{\max}$-truss. Thus based on Lemma 3, the insertion of (u, v) may affect $k_{\max}$-truss if the new upper bound of (u, v) is no less than $k_{\max}$. □

Detailed implementation of algorithm. Our algorithm for edge insertion is shown in Algorithm 6. By Lemma 9, we need to consider two cases to maintain $k_{\max}$-truss. First, when (u, v) is inserted into the $k_{\max}$-truss, we increase the support of all edges that form a triangle with (u, v) and compute $|\triangle^{k_{\max}+1}_{(u,v)}|$. If $|\triangle^{k_{\max}+1}_{(u,v)}| < k_{\max}-1$, it indicates that the insertion of (u, v) does not affect the trussness of other edges (line 4-13). Otherwise, it is possible to form a new $(k_{\max}+1)$-truss. Next, we assume the existence of $(k_{\max}+1)$-truss, remove the edges with support of $k_{\max}-2$, and finally see if any edges with a support of $k_{\max}-1$ still exist. This is achieved by iterating through the vertices of edges with support no less than $k_{\max}-1$ and adding the edges of their neighbors with a support of $k_{\max}-2$ to the candidate set $QueueE$ (line 14). Subsequently, we employ the peeling method to iteratively remove edges from the candidate set while collecting edges with new support $k_{\max}-2$ in a breadth-first search manner (lines 15-25). If there are edges with support greater than $k_{\max}-2$ at the end, they form a $(k_{\max}+1)$-truss; otherwise, those edges whose support has been updated regain their original support (lines 26-29).

Algorithm 6: Insertion

1 Insert (u, v) into $k_{\max}$-truss;
2 Update d_u and d_v;
3 **if** $u \in k_{\max}$-*truss and* $v \in k_{\max}$-*truss* **then**
4 $\quad QueueV \leftarrow \emptyset$;
5 $\quad$ Load $N_u(k_{\max}$-truss) and $N_v(k_{\max}$-truss);
6 $\quad$ **for** $w \in N_u(k_{\max}$-*truss*$) \cap N_v(k_{\max}$-*truss*$)$ **do**
7 $\quad\quad sup((u, w)) \leftarrow sup((u, w)) + 1$;
8 $\quad\quad$ **if** $sup((u, w)) > k_{\max} - 2$ **then**
9 $\quad\quad\quad$ **if** $u \notin QueueV$ **then** $QueueV \leftarrow u$;
10 $\quad\quad\quad$ **if** $w \notin QueueV$ **then** $QueueV \leftarrow w$;
11 $\quad\quad$ Repeat lines 9-10 for (v, w);
12 $\quad$ **if** $|\triangle^{k_{\max}+1}_{(u,v)}| < k_{\max} - 1$ **then** Continue;
13 $\quad QueueE \leftarrow \emptyset$; $\mathcal{S} \leftarrow \emptyset$;
14 $\quad$ ColCandidate ($k_{\max}$-truss, $QueueV$, $QueueE$);
15 $\quad$ **while** $QueueE \neq \emptyset$ **do**
16 $\quad\quad (u, v) \leftarrow QueueE.top()$; $QueueE.pop()$;
17 $\quad\quad$ Load $N_u(k_{\max}$-truss) and $N_v(k_{\max}$-truss) from disk;
18 $\quad\quad$ **for** $w \in N_u(k_{\max}$-*truss*$) \cap N_v(k_{\max}$-*truss*$)$ **do**
19 $\quad\quad\quad$ **if** $sup((u, w)) \leq k_{\max} - 2$ and $sup((v, w)) \leq k_{\max} - 2$ **then** Continue;
20 $\quad\quad\quad$ **if** $sup((u, w)) > k_{\max} - 2$ **then**
21 $\quad\quad\quad\quad$ **if** $(u, w) \notin \mathcal{S}$ **then** $\mathcal{S} \leftarrow \{u, w, sup((u, w))\}$;
22 $\quad\quad\quad\quad sup((u, w)) \leftarrow sup((u, w)) - 1$;
23 $\quad\quad\quad\quad$ **if** $sup((u, w)) = k_{\max}$ *and* (u, w) *not be visited* **then**
24 $\quad\quad\quad\quad\quad QueueE \leftarrow (u, w)$
25 $\quad\quad\quad$ Repeat lines 20-24 for (v, w);
26 $\quad$ **if** $\exists e = (u, v)$ *of* $k_{\max}$-*truss* s.t. $sup(e) > k_{\max} - 2$ **then**
27 $\quad\quad k_{\max} \leftarrow k_{\max} + 1$;
28 $\quad$ **else**
29 $\quad\quad$ **for** $\{u, v, s\} \in \mathcal{S}$ **do** $sup((u, v)) \leftarrow s$;
30 **else**
31 $\quad$ Update coreness of each node in G [15];
32 $\quad$ **if** $min\{sup((u, v)) + 2, min(core(u), core(v)) + 1\} \geq k_{\max}$ **then**
33 $\quad\quad$ Repeat lines 24-26 of Algorithm 5;

Second, if there exists at least a node that is not in $k_{\max}$-truss. As shown in case (2) in Lemma 9, for edges that satisfy the condition, they will be extended to $k_{\max}$-truss. In this case, the $k_{\max}$-truss will not contain the $(k_{\max}+1)$-truss. To address this situation, we initially employ the core pruning technique (Lemma 4) to eliminate nodes unlikely to be part of the $(k_{\max}+1)$-truss. Finally, we identify the $(k_{\max}+1)$-truss, *i.e.*, the new $k_{\max}$-truss, in the refined subgraph. (line 31-33).

Example 6: Consider the graph G in Fig. 2. Suppose that we insert an edge (v_1, v_5). First, the algorithm computes $E_{com_{(v_1,v_5)}}$. All these edge supports in $E_{com_{(v_1,v_5)}}$ increase by 1. Then, we find the candidate set, *i.e.*, $\{(v_5, v_6), (v_5, v_7), (v_5, v_8)\}$ and the $(k_{\max}+1)$-truss that is assumed to exist, i.e. the subgraph between these nodes $\{v_1, v_2, v_3, v_4, v_5\}$. Removing edges from the candidate set does not affect the support of edges in hypothetical $(k_{\max}+1)$-truss. Finally, this hypothetical $(k_{\max}+1)$-truss

Algorithm 7: ColCandidate

```
Input: k_max-truss, QueueV and QueueE
while QueueV ≠ ∅ do
    u ← QueueV.top(); QueueV.pop();
    Load N_u(k_max-truss);
    for w ∈ N_u(k_max-truss) do
        if sup((u, w)) > k_max − 2 then
            if w ∉ QueueV and w not be visited then QueueV ← w;
        else
            If (u, w) ∉ QueueE and (w, u) ∉ QueueE then
                QueueE ← (u, w);
```

becomes the real $(k_{\max}+1)$-truss.

V. Experiments

A. Experimental setup

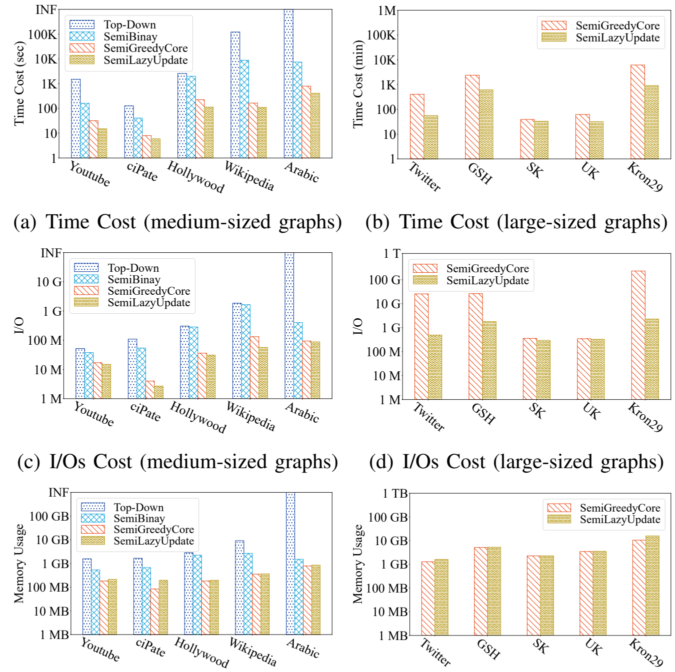
Different algorithms. For the computation of k_{max}-truss, we implement the proposed external memory algorithms, namely, SemiBinary, SemiGreedyCore and SemiLazyUpdate. To facilitate comparison, we also implement the state-of-the-art external memory algorithm Top-Down [27]. For k_{max}-truss maintenance, we implement the proposed Deletion and Insertion. To our knowledge, there is no external memory algorithm specifically designed to directly maintain the k_{max}-truss for dynamic graphs. In our experiments, we use external memory algorithms initially tailored for maintaining all k-trusses [12], as baselines. These baselines are identified as YLJ-Deletion and YLJ-Insertion for edge deletion and insertion respectively, and we implement them as the source codes are not publicly available.

Datasets. We collect 168 real-world networks of various types, along with 3 synthetic graphs, all of which are undirected. The detailed statistics of these networks are summarized in TABLE II. Among these datasets, the synthetic graphs Kron29 is generated by Graph500 kronecker (https://graph500.org/). The remaining networks are sourced from the Koblenz Network Collection (http://konect.unikoblenz.de/), the Stanford Network Collection (http://snap.stanford.edu/data/), the Web Graph Collection (http://webgraph.di.unimi.it/). For brevity, we use the GSH to represent the gsh-2015-host dataset.

Experimental settings. All algorithms are implemented in C++, and compiled using the g++ compiler with O3 optimization. Our experiments are conducted on a PC with an Intel Xeon Gold 5218R CPU @2.10GHz, 96GB of DDR4 RAM, and 7200 RPM SATA III 1TB SSD disk, running the Linux operating system. The block size is determined by the operating system, which fixed it at 4k bytes. The running time of an algorithm is measured by the time elapsed during the program's execution. For the input graph G, it is converted into a binary adjacency list form and stored on disk using the standard external-memory sorting algorithm. We set *capacity* to the number of vertices in G. Unless explicitly stated otherwise, we employ the symbol "INF" to indicate that the algorithm cannot terminate within 48 hours.

B. Performance studies

In this subsection, we select 5 medium-sized graphs and 5 large-sized graphs to evaluate the efficiency of our algorithms.



(a) Time Cost (medium-sized graphs) (b) Time Cost (large-sized graphs)
(c) I/Os Cost (medium-sized graphs) (d) I/Os Cost (large-sized graphs)
(e) Mem Cost (medium-sized graphs) (f) Mem Cost (large-sized graphs)

Fig. 6. Results of various algorithms for $k_{\max}$-truss computation

Exp-1: k_{max}-truss computation. Fig. 6 presents the results regarding running time, I/O cost, and memory overhead for different algorithms for k_{max}-truss computation. Note that, for large-sized graphs, both the Top-Down and SemiBinary algorithms surpass the time constraints, and thus their results are not included in Fig. 6. For the dataset Arabic, the Top-Down algorithm also exceeds the time constraints, and we label its runtime, I/O cost, and memory overhead as "INF" in Fig. 6.

As depicted in Fig. 6 (a-b), the proposed three novel algorithms show superior performance when compared to the existing Top-Down algorithm. Notably, SemiLazyUpdate, our optimal algorithm, remains at least two orders of magnitude faster than Top-Down on most graphs. In comparison to SemiGreedyCore, SemiLazyUpdate significantly enhances efficiency, particularly on large-scale graphs. For instance, for the dataset Kron29 with nearly 6 billion edges, SemiLazyUpdate computes the k_{max}-truss in a mere 1,012 minutes. In contrast, SemiGreedyCore requires 6,166 minutes, rendering it nearly six times slower than SemiLazyUpdate.

In terms of I/O cost, Fig. 6 (c-d) reveal that Top-Down has the highest I/O consumption among all algorithms, while SemiLazyUpdate exhibits the lowest. SemiBinary incurs higher I/O costs compared to SemiGreedyCore. These findings align with the experimental observations regarding time costs, as the runtime of external memory algorithms is notably influenced by the associated I/O costs. The rationale behind the reduced I/O overhead of SemiGreedyCore compared to SemiBinary lies in its effective utilization of the greedy strategy, allowing it to prune vertices that are definitely not included in the k_{max}-truss. Note that, as shown in Fig. 6 (d), the I/O overhead of SemiLazyUpdate on large-scale graphs is at least an order of magnitude less than that of SemiGreedyCore on Twitter, GSH, and Kron29. These results underscore the significant benefits of the combination

TABLE II

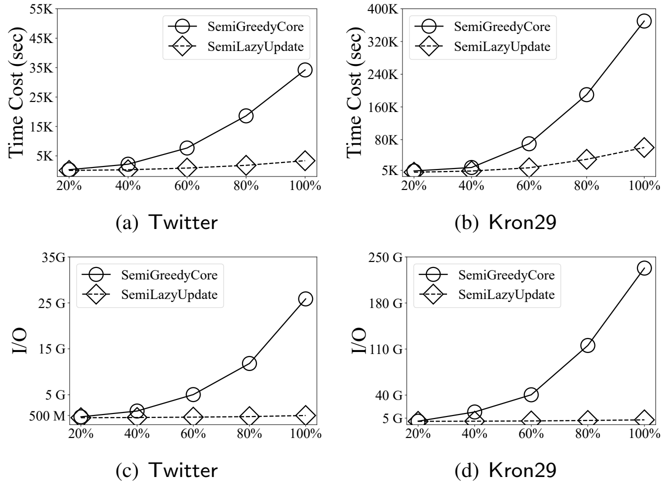
NETWORKS STATISTICS AND THE $k_{\max}$ RESULTS (1K=10^3,1M=10^6,AND 1G=10^9)

Networks	Name	$\|V\|$	$\|E\|$	$k_{\max}$	δ	Name	$\|V\|$	$\|E\|$	$k_{\max}$	δ	Name	$\|V\|$	$\|E\|$	$k_{\max}$	δ
Biological	Diseasome	0.5K	1.2K	11	10	G-Worm	3.5K	6.5K	7	10	CE-GN	2.2K	53.7K	23	48
	ecoMangwet	0.1K	1.4K	13	23	G-FisYeast	2.0K	12.6K	16	34	DR-CX	3.3K	84.9K	89	95
	Yeast	1.5K	1.9K	6	5	G-Fruitfly	7.3K	24.9K	7	12	HS-CX	4.4K	108.8K	90	98
	Celegans	0.5K	2.0K	9	10	G-Human	9.4K	31.2K	13	12	G-Yeast	6.0K	156.9K	36	64
	ecoFoodweb	0.1K	2.1K	11	24	SC-GT	1.7K	34.0K	49	60	CE-CX	15.2K	246.0K	75	78
	ecoFlorida	0.1K	2.1K	5	15	SC-CC	2.2K	34.9K	69	68	HuGene2	14.0K	9.0M	1683	1902
	G-Plant	1.7K	3.1K	10	12	HS-LC	4.2K	39.5K	59	66	HuGene1	21.9K	12.3M	1793	2047
	DM-HT	3.0K	4.7K	4	11	CE-PG	1.9K	47.8K	55	80	MoGene	43.1K	14.5M	799	1045
Collaboration	caHepPh	11.2K	117.6K	239	238	caAstroPh	17.9K	197.0K	57	56	caIMDB	896.3K	3.8M	3	23
	caGrQc	4.2K	13.4K	44	43	caCiteseer	227.3K	814.1K	87	86	caDBLP	540.5K	15.2M	337	336
	caCondMat	21.4K	91.3K	26	25	caMath	391.5K	873.8K	25	24	**Hollywood**	**1.1M**	**113.8M**	**2209**	**2208**
Citation	ctDBLP	12.6K	49.6K	9	12	ctCiteseer	384.1K	1.7M	13	15	ctHepPh	28.1K	3.1M	411	410
	ctCora	23.2K	89.2K	11	13	ctHepTh	22.9K	2.4M	562	561	**ctPatent**	**3.8M**	**16.5M**	**36**	**64**
Online contact	emDNC	0.9K	10.4K	75	74	emEU	32.4K	54.4K	13	22	comEnron	87.0K	297.5K	36	53
	comFBwal	45.8K	183.4K	10	16	dbpedia − team	365K	780K	3	9	emEuAll	265.0K	364.5K	20	37
	comUc	1.9K	13.8K	7	20	comDIGG	30.4K	85.2K	5	9	emEnLarge	33.7K	180.8K	22	43
Infrastructure	Euro	1.2K	1.4K	3	2	US1	129.2K	165.4K	3	3	Italy	6.7M	7.0M	3	3
	USAir97	0.3K	2.1K	22	26	PA	1.1M	1.5M	4	3	Britain	7.7M	8.2M	3	3
	Power	4.9K	6.6K	6	5	Belgium	1.4M	1.5M	3	3	Germany	11.5M	12.4M	3	3
	Openflights	2.9K	15.7K	23	28	Netherlands	2.2M	2.4M	3	3	Asia	12.0M	12.7M	4	3
	Luxembourg	114.6K	119.7K	3	2	CA	2.0M	2.8M	4	3	US2	23.9M	28.9M	4	3
Social	FbFood	620	2.1K	10	11	WikiElec	7.1K	100.8K	23	53	LiveMocha	104.1K	2.2M	27	92
	Weibo	58.7M	261.3M	80	193	GemsecRO	41.8K	125.8K	7	7	Buzznet	101.2K	2.8M	59	153
	BlogCata	88.8K	2.1M	101	221	fbMedia	27.9K	206.0K	31	31	fbSport	13.9K	86.8K	29	31
	Epinions	26.6K	100.1K	18	32	Brightkite	58.2K	214.1K	43	52	FourSq	639.0K	3.2M	38	63
	Hamster	2.4K	16.6K	25	24	GemsecHU	47.5K	222.9K	12	11	Themarker	69.4K	1.6M	51	164
	fbTvshow	3.9K	17.2K	57	56	Douban	154.9K	327.2K	11	15	Lastfm	1.2M	4.5M	23	70
	Twitter	**41.6M**	**1.4G**	**1998**	**2488**	Slashdot1	77.4K	469.2K	35	54	wikiTalk	2.4M	4.7M	53	131
	Livejournal	4.0M	27.9M	214	213	GemsecHR	54.6K	498.2K	13	21	Caster	149.7K	5.4M	207	419
	Gplus	23.6K	39.2K	7	12	Slashdot2	82.2K	504.2K	36	55	DIGG	770.8K	5.9M	73	236
	Advogato	5.2K	39.4K	19	25	Academia	190.2K	788.3K	11	19	Flixster	2.5M	7.9M	47	68
	fbPoli	5.9K	41.7K	26	31	fbArtist	50.5K	819.1K	23	69	Dogster	426.8K	8.5M	93	248
	Anybeat	12.6K	49.1K	25	33	TwiFollows	465.0K	833.5K	6	30	twiHiggs	456.6K	12.5M	72	125
	fbCom	14.1K	52.1K	21	20	Delicious	426.4K	908.3K	10	22	Flickr	1.7M	15.6M	153	309
	fbPubFig	11.6K	67.0K	25	42	Gowalla	196.6K	950.3K	29	51	Pokec	1.6M	22.3M	29	47
	fbGovern	7.1K	89.4K	30	46	**Youtube**	**3.2M**	**9M**	**33**	**88**	Orkut	3.0M	106.3M	75	230
Hyperlink	Polblogs	0.6K	2.3K	10	12	WikiIS	69.4K	907.4K	378	379	Wiki	1.9M	4.5M	31	66
	EPA	4.3K	8.9K	4	6	WikiFY	65.6K	921.6K	156	155	WikiTH	266.9K	4.6M	391	390
	Webbase	16.1K	25.6K	33	32	Notre	325.7K	1.1M	155	155	WikiLT	268.2K	5.1M	263	268
	WikiChInter	1.9M	9.0M	33	120	WikiIA	24.0K	1.2M	530	534	BerkStan	685.2K	6.6M	201	201
	Spam	4.8K	37.4K	23	35	WikiAF	72.3K	1.5M	364	363	IT	509.3K	7.2M	432	431
	Indochina	11.4K	47.6K	50	49	IkArabic	163.6K	1.7M	102	101	WikiEO	413.0K	8.2M	689	688
	WikiPedia	**13.5M**	**437M**	**1101**	**1135**	WikiAST	83.3K	2.0M	91	107	WikiCh	1.9M	9.0M	33	120
	Google	1.3K	2.8K	18	17	Stanford	281.9K	2.0M	62	71	UK2	129.6K	11.7M	500	499
	WikiVote	889	2.9K	7	9	BaiduRe	415.6K	2.4M	95	228	Hudong	2.0M	14.4M	267	266
	WikiNN	215.9K	2.9M	246	250	Italycnr	325.6K	2.7M	84	83	Baidu	2.1M	17.0M	31	78
	WikiYO	41.2K	696.4K	477	476	WikiLV	190.0K	2.9M	382	384	**UK**	**105M**	**3.3G**	**5705**	**5704**
	WikiCKB	60.7K	802.1K	342	373	WikiLA	181.2K	3.0M	255	266	**GSH**	**68.6M**	**1.8G**	**9923**	**9955**
	WikiSW	58.8K	877.0K	156	263	GoogleDir	875.7K	4.3M	44	44	**SK**	**50.6M**	**1.9G**	**4511**	**4510**
Technological	Routers	2.1K	6.6K	16	15	WHOIS	7.5K	56.9K	71	88	RLCaida	190.9K	607.6K	19	32
	PGP	10.7K	24.3K	27	31	Internet	40.2K	85.1K	17	23	Skitter	1.7M	11.1M	68	111
	Caida	26.5K	53.4K	16	22	P2P	62.6K	147.9K	4	6	IP	2.3M	21.6M	4	253
Software	Jung	6.1K	50.3K	17	65	JDK	6.4K	53.7K	17	65	Linux	30.8K	213.2K	10	23
Lexical	EAT	23.1K	297.1K	9	34	Bible	1.8K	9.1K	11	15	Yahoo	653.3K	2.9M	3	29
Miscellaneous	**Arabic**	**22.7M**	**639.9M**	**3248**	**3247**	misFlickr	105.9K	2.3M	574	573	misDBpedia	4.0M	12.6M	18	20
	misTwin	14.3K	20.6K	27	26	misAmazon	403.4K	2.4M	11	10	misActor	382.2K	15.0M	294	365
Synthetic	**Kron29**	**536.8M**	**5.9G**	**1976**	**3987**	CL-1000000	910K	2.7M	4	12	geo1k-40k	1K	40K	34	47

of linear-heap and dynamic-heap in greatly reducing I/O consumption, thereby enhancing the efficiency of k_{max}-truss computation.

In terms of memory usage, as evident from Fig. 6 (e-f), our algorithms exhibit lower memory consumption compared to Top-Down. SemiGreedyCore, in particular, requires the least amount of memory as expected. The memory overhead of SemiBinary is greater than that of SemiGreedyCore because it contains numerous unpromising nodes that must not be in k_{max}-truss, resulting in increased memory usage. Furthermore, SemiLazyUpdate consumes slightly more memory than SemiGreedyCore but still less than SemiBinary. This is due to the dynamic-heap structure incorporated in SemiLazyUpdate, which stores frequently updated edges in memory, leading to a marginal increase in memory overhead. These results confirm our theoretical results in Section III-C. Additionally, the SemiLazyUpdate algorithm requires less than 16 GB of memory to effectively process the largest dataset Kron29. These results highlight the efficiency and potential applicability of SemiLazyUpdate for large-scale graph analysis with limited memory resources.

Exp-2: Scalability testing for k_{max}-truss computation. We randomly select 20%-80% of the vertices from each dataset to generate four subgraphs for testing the scalability of SemiGreedyCore and SemiLazyUpdate. Due to the space limits, we present results specifically for the Twitter and Kron29 datasets with different scales of $|V|$. Fig. 7 shows the time costs and I/O costs of SemiGreedyCore and SemiLazyUpdate with varying $|V|$ on the Twitter and Kron29. The results also show a consistent trend on graphs with different orders of magnitude in the number of vertices. Specifically, with an increase in the number of vertices (i.e.



(a) Twitter (b) Kron29

(c) Twitter (d) Kron29

Fig. 7. Scalability of the k_{max}-truss computation algorithms (vary $|V|$)

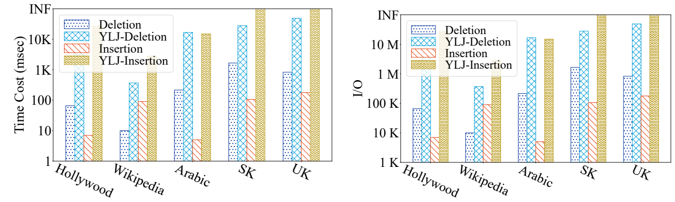
TABLE III
THE RESULTS OF THE GRAPH REDUCED BY SemiGreedyCore ($1K=10^3$, $1M=10^6$, AND $1G=10^9$)

Dataset			$G_{c_{max}}$			G
Name	$\|V\|$	$\|E\|$	$\|E(G_{c_{max}})\|$	*per*	k'_{max}	k_{max}
Youtube	3.2M	9M	29,001	**0.31%**	32	33
ctPate	3.8M	16.5M	3,951	**0.02%**	36	38
Hollywood	1.1M	113.8M	2,438,736	2.14%	**2209**	**2209**
Wikipedia	13.5M	437M	643,181	**0.15%**	**1101**	**1101**
Arabic	22.7M	639.9M	5,273,128	**0.82%**	**3248**	**3248**
Twitter	41.6M	1.4G	4,585,552	**0.31%**	1994	1998
GSH	68.6M	1.8G	52,570,705	2.92%	9921	9923
SK	50.6M	1.9G	10,185,835	**0.52%**	**4511**	**4511**
UK	105M	3.3G	16,270,660	**0.49%**	**5705**	**5705**
Kron29	536.8M	5.9G	72,011,126	**1.22%**	**1978**	**1978**

$|V|$), the running time and I/O costs across all algorithms also exhibit an upward trend. Again, the SemiLazyUpdate algorithm outperforms the SemiGreedyCore in all parameter settings, particularly showcasing a tenfold improvement on the Twitter dataset. Significantly, the time and I/O overhead incurred by SemiLazyUpdate show a steady and linear increase with the expansion of the vertex set, whereas SemiGreedyCore demonstrates a substantially steeper increase in both time and I/O overhead. This suggests that SemiLazyUpdate is highly scalable and efficiently handles large-scale graphs. Regarding memory usage, which scales linearly with the number of vertices, we do not present the results due to space limits.

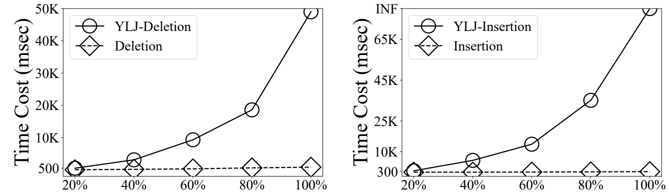
Exp-3: Pruning performance of SemiGreedyCore. TABLE III provides a detailed characterization of $G_{c_{max}}$ by performing SemiGreedyCore on the original graph G. In TABLE III, $|E(G_{c_{max}})|$ represents the number of edges in $G_{c_{max}}$, *per* signifies the percentage of edges in $G_{c_{max}}$ relative to the total number of edges in G. Additionally, k'_{max} (k_{max}) denotes the maximum trussness in $G_{c_{max}}$ (G). As can be seen from TABLE III, on most other datasets, SemiGreedyCore retains less than 2% of the remaining edges from the original graph. In particular, k'_{max} in $G_{c_{max}}$ closely aligns with k_{max} in the original graph G, with a difference of no more than 4 on all datasets. These results confirm the efficiency of our graph reduction approach in handling large real-world networks.

Exp-4: k_{max}-truss maintenance. In this experiment, we redefine the term "INF" when the runtime exceeds the 100K milliseconds time limit, consequently designating its I/O cost as "INF". We randomly insert (delete) 1000 edges for each dataset and invoke the YLJ-Insertion and Insertion



(a) Average Time (b) Average I/Os

Fig. 8. Results of k_{max}-truss maintenance



(a) Average Update Time On UK (b) Average Update Time On UK

Fig. 9. Scalability of k_{max}-truss maintenance (vary $|V|$)

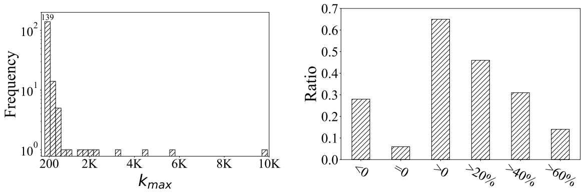
(YLJ-Deletion and Deletion) algorithms to maintain the k_{max}-truss. The results, including the average processing time and I/O costs, for three medium-sized graphs and two large-sized graphs are illustrated in Fig. 8. Similar results are observed for the other datasets as well.

Across all datasets, our algorithms, Insertion and Deletion, consistently outperform YLJ-Insertion and YLJ-Deletion by at least one order of magnitude in handling both edge insertion and deletion. For example, on the Hollywood dataset, YLJ-Insertion requires 27,008 ms to maintain the k_{max}-truss for an edge insertion, whereas Insertion only takes 7 ms. For an edge deletion, the runtime of YLJ-Deletion is 6,670 ms, while Deletion completes the maintenance of k_{max}-truss in just 63 ms, showcasing a performance advantage of two orders of magnitude over the former. The limitation of YLJ-Insertion and YLJ-Deletion lies in their dependence on a breadth-first search within the k_{max}-truss to identify edges with a trussness value of k_{max}, forming a candidate set for potential updates. These results demonstrate that our algorithms outperform YLJ-Insertion and YLJ-Deletion in terms of time and I/O efficiency for dynamically updated graphs.

Exp-5: Scalability testing for k_{max}-truss maintenance. we make use of the dataset UK to test the scalability of the proposed k_{max}-truss maintenance algorithms. As shown in Fig. 9 (a), for an edge deletion, as $|V|$ increases from 20% to 100%, the Deletion exhibits a stable increase in processing time, whereas the YLJ-Deletion experiences a sharp rise. From the Fig. 9 (b), for an edge insertion, Insertion and YLJ-Insertion still show the same trend as an edge deletion. Once again, we can observe that the runtime of our algorithms is significantly lower than those of the state-of-the-art algorithms. Therefore, these results indicate that our algorithms show better scalability for k_{max}-truss maintenance.

C. k_{max} of Real-World Networks

Exp-6: The distribution of k_{max}. We conduct an extensive evaluation of the k_{max} values for a total of 168 real-world graphs, with the results presented in TABLE II. The distribution of k_{max} for these 168 graphs is also depicted in Fig. 10. Generally, the majority of the graphs exhibit

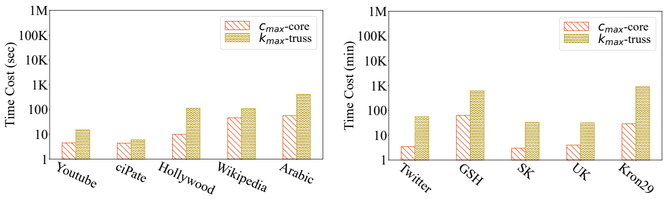


(a) All networks (b) Ratio of $\frac{c_{max}-k_{max}}{c_{max}}$

Fig. 10. Results of k_{max} and c_{max}

TABLE IV
COMPARISON OF RESULTING SIZES

Name	$\lvert E(c_{\max}\text{-core})\rvert$	$\lvert E(k_{\max}\text{-truss})\rvert$	Name	$\lvert E(c_{\max}\text{-core})\rvert$	$\lvert E(k_{\max}\text{-truss})\rvert$
Youtube	29,001	8,250	Twitter	4,585,552	3,993,811
ctPate	3,951	2,075	GSH	52,570,705	49,923,540
Hollywood	2,438,736	2,438,736	SK	10,185,835	10,176,815
Wikipedia	643,181	643,181	UK	16,270,660	16,270,660
Arabic	5,273,128	5,273,128	Kron29	72,011,126	6,137,945



(a) Time Cost (medium-sized graphs) (b) Time Cost (large-sized graphs)

Fig. 11. Running time of different models

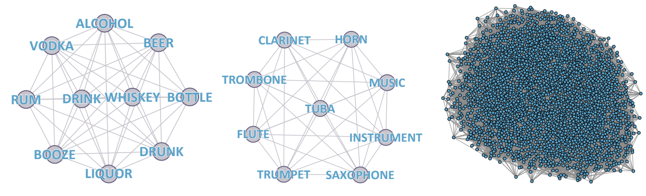
relatively small $k_{\max}$ values. As seen from Fig. 10 (a), it is evident that 139 networks have a k_{max} value smaller than 200, confirming that many real-world networks indeed have a small k_{max} value. However, certain large and cohesive graphs, particularly social networks and hyperlink networks, may have significantly larger $k_{\max}$ values. For instance, the social network Twitter has a $k_{\max}$ value of 2,488, while the web graph GSH has a strikingly high $k_{\max}$ value of 9,955.

Exp-7: Comparison between k_{max} and degeneracy. Degeneracy [28] is a crucial metric for measuring the sparsity of a graph, denoted by c_{max}. Here we compare k_{max} with c_{max} by calculating $\frac{c_{max}-k_{max}}{c_{max}}$ for 168 real-world graphs in TABLE II, and the results are shown in Fig. 10 (b). As observed, c_{max} is greater than k_{max} in 65% of the real-world graphs. Additionally, on 28% of the more cohesive real-world graphs, we find that $k_{max} = c_{max} + 1$ in the worst-case scenario. Notably, a significant proportion of real-world graphs exhibit a power-law distribution, particularly evident in social network. In approximately 90% of such graphs, the k_{max} are less than the c_{max}. Moreover, both k_{max} and c_{max} are commonly employed as complexity bounds for identical graph algorithms, as mentioned earlier. Given that k_{max} is significantly smaller than c_{max} in most graphs, expressing the complexity bound in terms of k_{max} yields a more precise and stringent estimation.

Exp-8: Comparison between $k_{\max}$-truss and other models.

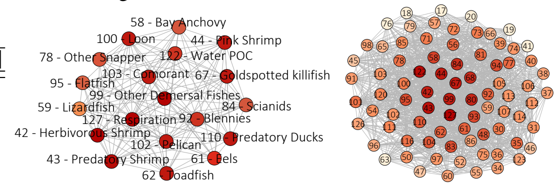
In community search, k-core, k-truss, and k-clique models are commonly employed [29]. However, accurately computing the maximum clique is NP-hard [30], and we fail to obtain the maximum clique in 3,000 minutes on Youtube. Consequently, due to the inefficiency of maximum clique computation, we choose the $c_{\max}$-core as the comparison model.

Fig. 11 shows the runtime of $c_{\max}$-core and $k_{\max}$-truss model on 10 graphs. The running time of $c_{\max}$-core is lower than that of $k_{\max}$-truss, which is mainly because the I/O complexity of $c_{\max}$-core computation is $O(\log_2 h(d)\times\tau(\tilde{n}+\tilde{m})/2)$



(a) $k_{\max}$-truss (b) maximum clique (c) $c_{\max}$-core

Fig. 12. The 9-truss, 9-clique and 13-core on WordNet



(a) $k_{\max}$-truss (b) $c_{\max}$-core

Fig. 13. 11-truss and 23-core on FoodWeb

[16], while the I/O complexity of $k_{\max}$-truss computation is $O(max(log_2^{ub}|E(G_{c_{\max}})|Cost, |E(H')|Cost))$, presented in Theorem 3. While $k_{\max}$-truss exhibits slightly lower computational efficiency compared to $c_{\max}$-core, the resulting size of $k_{\max}$-truss is smaller than $c_{\max}$-core. For instance, on the Kron29, the number of edges for $k_{\max}$-truss is an order of magnitude less than that of $c_{\max}$-core. The experimental results show that $k_{\max}$-truss is more capable of capturing the key information of the graph compared to $c_{\max}$-core, which can be used to enhance the information mined by $c_{\max}$-core.

D. Case Study

Exp-9: Word association network. Here we conduct a case study on a word association network, referred to as WordNet [31]. Each node represents a word, and edges between words indicate strong semantic relationships. Our goal is to identify a dense subgraph that contains the largest number of words with similar meanings, offering a precise depiction of the scenarios associated with these semantically related words. To achieve this, we compute the $k_{\max}$-truss, along with the $c_{\max}$-core and maximum clique for comparison purposes.

We begin by showing the $k_{\max}$-truss, which is a 9-truss containing 10 words, depicted in Fig. 12 (a). This 9-truss identifies semantically related words associated with "ALCOHOL ", effectively capturing the essence of the complete context. Conversely, as illustrated in Fig. 12 (b), the 9-clique achieves a similar outcome by uncovering words associated with "MUSIC ". However, due to the strict definition of the k-clique [32], which makes this model not noise-resistant. For instance, in Fig. 12 (a), "BOTTLE" and "DRINK" are not edge-connected, yet they belong to the same scene. This semantic relationship can be effectively captured by the $k_{\max}$-truss. In contrast, the k-clique fails to achieve this outcome. Lastly, in Fig. 12 (c), the 13-core delves into a larger graph with loosely connected vertices, which curtails its ability to provide a precise characterization of the given scenario. These results highlight that the $k_{\max}$-truss provides a more comprehensive representation of semantically related words in specific contexts.

Exp-10: Florida bay food web. We conduct a case study on FoodWeb from SNAP [33] to identify important species that are involved in the food chain that completes the carbon cycle in an ecosystem. Each node represents a taxon (similar to a species), and edges represent the transfer of carbon information between species. Fig. 13 (a) and Fig. 13 (b) show the community search results obtained by applying the $k_{\max}$-truss model and the $c_{\max}$-core model, respectively. Notably, the degree of a node in the FoodWeb directly correlates with its involvement in multiple food chains. This leads us to establish a noteworthy correlation: a higher node degree indicates greater significance of a species within the ecosystem, visually represented by a darker color.

The inspection of Fig. 13 (b) reveals a total of 80 identified species, among which those concentrated in the central region exhibit heightened importance. Conversely, the community outcome in Fig. 13 (a) is predisposed to capturing pivotal species compared $c_{\max}$-core as shown in Fig. 13 (b). Thus, based on this comparative analysis, it can be inferred that the $k_{\max}$-truss model presents a superiority over the $c_{\max}$-core model in terms of effectiveness in revealing key species within the ecosystem's food web structure.

VI. Related Work

K-Truss Decomposition. Identifying cohesive subgraphs is a crucial task in social network analysis, particularly in the context of k-truss [1]. Numerous studies have been conducted to investigate k-truss decomposition. The earliest algorithm for truss decomposition is proposed by Cohen [1]. After that, several different algorithms have been proposed to compute k-trusses [2], [8]–[11], [34], [35]. The above algorithms are in-memory algorithms that are slow in handling large real-world graphs, except that [11] is an I/O efficient algorithm. The most relevant study to our work is [11], where the Top-Down algorithm is implemented to compute $k_{\max}$-truss. Nonetheless, experimental results demonstrate that our approach outperforms the Top-Down algorithm.

Truss Maintenance. In real-world scenarios, graphs are subject to continuous changes over time. Many works have focused on developing efficient incremental algorithms. For truss maintenance, Zhou *et al.* [26] focused on dynamically maintaining maximal trusses in evolving networks. Ebadian *et al.* [36] presented a novel hybrid strategy for updating k-truss in public-private graphs. Zhang *et al.* [37] proposed an efficient truss maintenance algorithm on dynamic graphs based on the truss decomposition order. Luo *et al.* [38] proposed a batch truss maintenance algorithm by presenting an edge structure called a triangle disjoint set. In addition to the in-memory algorithm, Jiang *et al.* [12] proposed an I/O efficient algorithm to maintain the k-truss community in the case of dynamic graphs. Given that the prior research did not address the specialized maintenance of the $k_{\max}$-truss, we stand as the first to implement I/O efficient maintenance of the $k_{\max}$-truss.

I/O-Efficient Graph Algorithm. I/O-efficient graph algorithms have been an active research area in recent years. There have been several proposals for I/O-efficient graph algorithms for a variety of graph problems, such as core decomposition [39] [15], triangle enumeration problem [40], truss decomposition [11], ECC graph decomposition problem [41], strong connected components computation [42], [43], diversified top-k clique search problem [44], c-Approximate Nearest Neighbor Search in High-dimensional Space [45].

VII. Conclusions

In this paper, we address the problem of computing the $k_{\max}$-truss on massive graphs that cannot be fully accommodated in the main memory. We propose an I/O efficient algorithm with a memory usage of $O(n)$ and explore two optimization strategies to further reduce the I/O and CPU costs. As real-world graphs are subject to dynamic changes, we also develop an I/O-efficient $k_{\max}$-truss maintenance algorithm tailored for dynamic graphs. Through a comprehensive evaluation involving 168 real-world graphs and 3 synthetic graphs, the $k_{\max}$ tend to be notably smaller than the degeneracy. Experimental results demonstrate that our algorithms can be implemented two orders of magnitude faster than the state-of-the-art approaches in terms of both $k_{\max}$-truss computation and maintenance.

Acknowledgement. This work was partially supported by (I) the National Key Research and Development Program of China 2020AAA0108503, and (II) NSFC Grants U2241211, 62072034. Rong-Hua Li is the corresponding author of this paper.

References

[1] J. Cohen, "Trusses: Cohesive subgraphs for social network analysis," *National security agency technical report*, vol. 16, no. 3.1, 2008.

[2] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu, "Querying k-truss community in large and dynamic graphs," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014, pp. 1311–1322.

[3] E. Akbas and P. Zhao, "Truss-based community search: a truss-equivalence based indexing approach," *Proceedings of the VLDB Endowment*, vol. 10, no. 11, pp. 1298–1309, 2017.

[4] Y. Zhu, Q. Zhang, L. Qin, L. Chang, and J. X. Yu, "Querying cohesive subgraphs by keywords," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 2018, pp. 1324–1327.

[5] R. G. Downey and M. R. Fellows, *Parameterized complexity*. Springer Science & Business Media, 2012.

[6] A. Buchanan, J. L. Walteros, S. Butenko, and P. M. Pardalos, "Solving maximum clique in sparse graphs: an o (nm+ n2^ d/4) o (nm+ n 2 d/4) algorithm for d d-degenerate graphs," *Optimization Letters*, vol. 8, pp. 1611–1617, 2014.

[7] L. Gianinazzi, M. Besta, Y. Schaffner, and T. Hoefler, "Parallel algorithms for finding large cliques in sparse graphs," in *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, 2021, pp. 243–253.

[8] H. Kabir and K. Madduri, "Parallel k-truss decomposition on multicore systems," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2017, pp. 1–7.

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

[10] A. Conte, D. De Sensi, R. Grossi, A. Marino, and L. Versari, "Truly scalable k-truss and max-truss algorithms for community detection in graphs," *IEEE Access*, vol. 8, pp. 139 096–139 109, 2020.

[11] J. Wang and J. Cheng, "Truss decomposition in massive networks," *Proc. VLDB Endow.*, vol. 5, no. 9, pp. 812–823, 2012. [Online]. Available: http://vldb.org/pvldb/vol5/p812_jiawang_vldb2012.pdf

[12] Y. Jiang, X. Huang, and H. Cheng, "I/o efficient k-truss community search in massive graphs," *The VLDB Journal*, vol. 30, pp. 713–738, 2021.

[13] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1804–1815, 2015.

[14] A. Aggarwal and S. Vitter, Jeffrey, "The input/output complexity of sorting and related problems," *Communications of the ACM*, vol. 31, no. 9, pp. 1116–1127, 1988.

[15] D. Wen, L. Qin, Y. Zhang, X. Lin, and J. X. Yu, "I/o efficient core graph decomposition at web scale," in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 2016, pp. 133–144.

[16] R.-H. Li, Q. Song, X. Xiao, L. Qin, G. Wang, J. X. Yu, and R. Mao, "I/o-efficient algorithms for degeneracy computation on massive networks," *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 7, pp. 3335–3348, 2020.

[17] C. S. J. Nash-Williams, "Edge-disjoint spanning trees of finite graphs," *Journal of the London Mathematical Society*, vol. 1, no. 1, pp. 445–450, 1961.

[18] J. Jiang, Q. Zhang, R.-H. Li, Q. Dai, and G. Wang, "I/o efficient max-truss computation in large static and dynamic graphs," 2023. [Online]. Available: https://github.com/jiangjiaqi6/MaxTruss/blob/master/main-icde-fullverion.pdf

[19] B. Menegola, "An external memory algorithm for listing triangles," 2010.

[20] V. Batagelj and M. Zaversnik, "An o (m) algorithm for cores decomposition of networks," *arXiv preprint cs/0310049*, 2003.

[21] F.-C. Leu, Y.-T. Tsai, and C. Y. Tang, "An efficient external sorting algorithm," *Information processing letters*, vol. 75, no. 4, pp. 159–163, 2000.

[22] S. B. Seidman, "Network structure and minimum degree," *Social networks*, vol. 5, no. 3, pp. 269–287, 1983.

[23] L. Chang and L. Qin, *Cohesive subgraph computation over large sparse graphs: algorithms, data structures, and programming techniques*. Springer, 2018.

[24] A. E. Sariyuce, C. Seshadhri, A. Pinar, and U. V. Catalyurek, "Finding the hierarchy of dense subgraphs using nucleus decompositions," in *Proceedings of the 24th International Conference on World Wide Web*, 2015, pp. 927–937.

[25] Z. Zhang, L. Ren, Y. You, K. Zhang, and J. Huang, "Dense subgraph mining kmax-truss optimization algorithm," in *2nd International Conference on Artificial Intelligence, Automation, and High-Performance Computing (AIAHPC 2022)*, vol. 12348. SPIE, 2022, pp. 541–546.

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

[27] J. Wang and J. Cheng, "Truss decomposition in massive networks," *arXiv preprint arXiv:1205.6693*, 2012.

[28] D. R. Lick and A. T. White, "k-degenerate graphs," *Canadian Journal of Mathematics*, vol. 22, no. 5, pp. 1082–1096, 1970.

[29] Y. Fang, X. Huang, L. Qin, Y. Zhang, W. Zhang, R. Cheng, and X. Lin, "A survey of community search over big graphs," *The VLDB Journal*, vol. 29, pp. 353–392, 2020.

[30] R. M. Karp, *Reducibility among combinatorial problems*. Springer, 2010.

[31] X. Huang, H. Cheng, R.-H. Li, L. Qin, and J. X. Yu, "Top-k structural diversity search in large networks," *The VLDB Journal*, vol. 24, no. 3, pp. 319–343, 2015.

[32] S. Wasserman and K. Faust, "Social network analysis: Methods and applications," 1994.

[33] R. E. Ulanowicz and D. L. DeAngelis, "Network analysis of trophic dynamics in south florida ecosystems," *US Geological Survey Program on the South Florida Ecosystem*, p. 114, 1999.

[34] P.-L. Chen, C.-K. Chou, and M.-S. Chen, "Distributed algorithms for k-truss decomposition," in *2014 IEEE International Conference on Big Data (Big Data)*. IEEE, 2014, pp. 471–480.

[35] S. Smith, X. Liu, N. K. Ahmed, A. S. Tom, F. Petrini, and G. Karypis, "Truss decomposition on shared-memory parallel systems," in *2017 IEEE high performance extreme computing conference (HPEC)*. IEEE, 2017, pp. 1–6.

[36] S. Ebadian and X. Huang, "Fast algorithm for k-truss discovery on public-private graphs," *arXiv preprint arXiv:1906.00140*, 2019.

[37] Y. Zhang and J. X. Yu, "Unboundedness and efficiency of truss maintenance in evolving graphs," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1024–1041.

[38] Q. Luo, D. Yu, X. Cheng, Z. Cai, J. Yu, and W. Lv, "Batch processing for truss maintenance in large dynamic graphs," *IEEE Transactions on Computational Social Systems*, vol. 7, no. 6, pp. 1435–1446, 2020.

[39] J. Cheng, Y. Ke, S. Chu, and M. T. Özsu, "Efficient core decomposition in massive networks," in *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 2011, pp. 51–62.

[40] X. Hu, Y. Tao, and C.-W. Chung, "Massive graph triangulation," in *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*, 2013, pp. 325–336.

[41] L. Yuan, L. Qin, X. Lin, L. Chang, and W. Zhang, "I/o efficient ecc graph decomposition via graph reduction," *The VLDB Journal*, vol. 26, no. 2, pp. 275–300, 2017.

[42] Z. Zhang, J. X. Yu, L. Qin, L. Chang, and X. Lin, "I/o efficient: computing sccs in massive graphs," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013, pp. 181–192.

[43] X. Wan and H. Wang, "Efficient semi-external scc computation," *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[44] L. Yuan, L. Qin, X. Lin, L. Chang, and W. Zhang, "Diversified top-k clique search," *The VLDB Journal*, vol. 25, pp. 171–196, 2016.

[45] W. Liu, H. Wang, Y. Zhang, W. Wang, and L. Qin, "I-lsh: I/o efficient c-approximate nearest neighbor search in high-dimensional space," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 1670–1673.