2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)

# K-Truss Decomposition of Large Networks on a Single Consumer-Grade Machine

Jian Wu, Alison Goshulak, Venkatesh Srinivasan, Alex Thomo
Department of Computer Science, University of Victoria
Victoria, BC, V8P 5C2, Canada
{wujian, agoshula, srinivas, thomo}@uvic.ca

***Abstract*—$k$-truss decomposition of a graph is a method to discover cohesive subgraphs and to study the hierarchical structure among them. The existing algorithms for computing $k$-truss of today's massive networks mainly focus on reducing the runtime using parallel computation on a powerful multi-core server. Our focus, by contrast, is to investigate the feasibility of computing the $k$-truss on a single consumer-grade machine within a reasonable amount of time. We engineer two efficient $k$-truss decomposition algorithms: the edge-peeling algorithm proposed by J. Wang and J. Cheng and the asynchronous $h$-index-updating algorithm proposed by A. E. Sariyuce, C. Seshadhri, and A. Pinar. We reduce their memory usage significantly by optimizing the underlying data structures and by using WebGraph, an efficient framework for graph compression. With our optimized implementation, we show that we can efficiently compute $k$-truss decomposition of large networks (*e.g.*, a graph with 1.2 billion edges) on a single consumer-grade machine.**

***Index Terms*—$k$-truss, $k$-core, graph analytics**

## I. Introduction

Identifying various cohesive subgraphs in a massive network is crucial to the efficient and effective analytics of the network [1]–[4]. $k$-truss is an important kind of cohesive subgraphs of a network that has received growing attention in the recent years [5]–[9]. Motivated by the need to find a structure that is a relaxation of a clique [10] and is efficiently computable, $k$-truss finds applications in social network visual analysis [11], community search [12], maximum clique finding [13], etc. The $k$-truss of a graph is defined as the largest subgraph in which each edge is contained in at least $k-2$ triangles within the subgraph [14]. Given a graph, the $k$-truss decomposition problems aims to find the $k$-trusses of the graph for all $k$.

The definition of $k$-truss is similar to $k$-core [15]–[19], which is defined as the largest subgraph in which every vertex has a minimum degree of $k$ within the subgraph. The $k$-truss focuses on the edges of a graph while the $k$-core focuses on the vertices. We can make an analogy that the edge in $k$-truss is similar to the vertex in $k$-core while the number of triangles for an edge to be contained in is similar to the degree for a vertex. However, the definition of $k$-truss is more rigorous than $k$-core since $k$-truss is based on triangles, which have higher dimensionality than edges that $k$-core is based on.

There are mainly two types of algorithms for efficiently computing the $k$-trusses: the serial algorithm suited for medium-sized graphs and the parallel algorithm suited for large-scale graphs. The serial algorithm is based on the concept of edge peeling proposed by J. Wang and J. Cheng [14]. Their algorithm iteratively eliminates edges at each stage based on their support value until all edges in the graph are removed. In their implementation, a hash table was used to check whether two vertices form an edge or not. The endpoints of each edge were hashed as the keys for the hash table. The hash table can work well for moderate-sized graphs. However, for large graphs, the hash table is expensive to use and designing an optimum hash function is not a trivial problem. The second type of algorithms use more advanced parallelization techniques on high-performance multi-core machines to significantly reduce the runtime [20]–[23]. Memory usage is not the major concern for these parallel programs since they are designed for high-performance machines, which are usually capable of keeping the whole graph as well as the hash table in the main memory. However, the cost for the hardware is high. For algorithms that avoid using the hash table (*e.g.*, [20] uses an array-based alternative), we can still find room for optimization on the data structure design to use the memory more efficiently. In short, both the serial and the parallel algorithms have limitations. For the serial algorithm, the inefficient and cumbersome data structure design (*e.g.*, the use of a hash table) hinders its use for large graphs. For the parallel algorithms, besides the same problem that the serial algorithm suffers, the other problem is the high hardware cost, as the focus of the parallel algorithms is to reduce the runtime on powerful machines.

Our focus in this work is to investigate if it is viable to efficiently and economically compute the $k$-trusses of large networks on a single consumer-grade machine. Therefore, the memory usage by the program is our major concern. We aim to engineer two algorithms: the serial edge-peeling algorithm [14] and the parallel asynchronous $h$-index-updating algorithm [22], with the goal to minimize the memory usage compared to the original implementations, but still with high time efficiency. We target these two algorithms because of their efficiency and relatively small memory footprints. For example, the edge-peeling algorithm optimizes Cohen's very first $k$-truss decomposition algorithm [5] with improved time complexity. The asynchronous $h$-index-updating algorithm has a relatively smaller memory footprint compared to other parallel algorithms. A concrete example would be: for a graph with 41 million vertices and 1.2 billion edges, the $h$-index-updating algorithm needs around 24 GB memory while the

**IEEE/ACM ASONAM 2018, August 28-31, 2018, Barcelona, Spain**
**978-1-5386-6051-5/18/$31.00 © 2018 IEEE**

memory-efficient parallel algorithm in [20] needs around 34 GB memory.

For the $k$-truss problem, or any graph-related problem, keeping the complete graph representation in the main memory is commonly the most memory-consuming component. To significantly reduce the graph's footprint in the main memory, we resort to WebGraph [24], a highly efficient graph compression framework that allows random access to a memory-mapped compressed graph stored on the hard drive. WebGraph also supports thread-safe operations on an immutable graph, facilitating parallel computation. The other memory-consuming component in the $k$-truss program is the use of a hash table to check whether two vertices form an edge or not. The hash table has a total number of entries equal to the number of edges of the graph. The purpose of using a hash table is to achieve constant-time querying in an optimum scenario. However, it is expensive to use for large graphs in practice. Therefore, in our implementation, we avoid using a hash table. We carefully design an array-based structure with a small memory footprint and its corresponding operations to achieve the same functionality that a hash table can provide. With our optimized implementation, we can efficiently compute the $k$-trusses of large networks (up to 1.2 billion edges) on a consumer-grade machine.

The contributions of this work are summarized as follows:

- We engineer two $k$-truss algorithms: the serial edge-peeling algorithm [14] and the parallel asynchronous $h$-index-updating algorithm [22], with optimized data structure design and WebGraph (an efficient framework for graph compression) to achieve both time and memory efficiency.
- We perform a thorough experimental study on different datasets using our optimized algorithms and compare the performance with the original algorithms.
- We investigate the minimum memory requirement for the optimized algorithms to maintain good scalability.
- We show that it is possible to efficiently perform $k$-truss decomposition of large networks on a single consumer-grade machine.

The layout of this work is as follows. Section II introduces the concept of the $k$-truss decomposition. Section III details the two algorithms we engineer and the optimizations we propose. Section IV describes that datasets and experimental setup. Section V presents our experimental results and investigates the minimum memory requirement. Section VI surveys the related works on $k$-truss decomposition. Section VII gives the conclusions of this work.

## II. PRELIMINARIES

For the $k$-truss decomposition problem, we consider undirected, unweighted simple graphs. For a given graph $G$, the vertex set is denoted by $V$ and the edge set is denoted by $E$. Therefore, the number of vertices is $n = |V|$ and the number of edges is $m = |E|$. The set of neighbors of a vertex $u$ is denoted by $neighbor(u)$ such that $neighbor(u) = \{v : (u, v) \in E\}$. The degree of $u$ is defined as $degree(u) = |neighbor(u)|$.

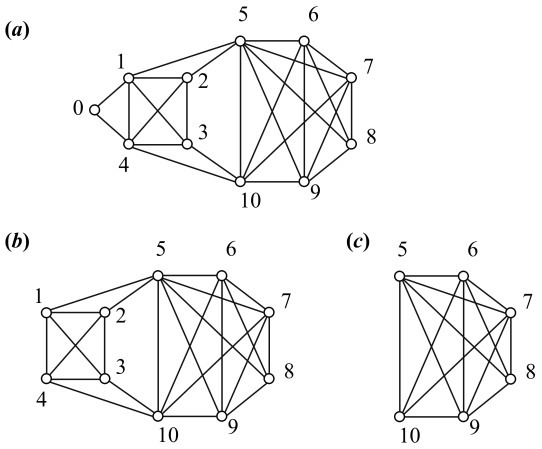


Fig. 1. (a) An undirected, unweighted simple graph $G$; (b) the 4-core of $G$ (no 5-core exists); (c) the 5-truss of $G$ (no 6-truss exists).

Each vertex in the graph is assigned an unique vertex ID from 0 to $n-1$. The order of vertices is based on their vertex IDs. For example, we say $u$ is ordered before $v$ if $u < v$. Based on this ordering, we define a triangle as follows:

*Definition 1:* (*triangle*). A triangle in $G$ is defined as a cycle of three vertices $\{u, v, w \in V\}$, denoted by $\triangle_{uvw}$, such that $u < v < w$ and all three edges exist in $G$ (*i.e.*, $(u, v), (v, w), (u, w) \in E$).

With the notion of triangles, we introduce the definition for support of an edge:

*Definition 2:* (*support*). The support of an edge $e \in E$, denoted by $support(e)$, is defined as the number of triangles in $G$ that contain $e$.

$k$-truss is defined based on the notion of support:

*Definition 3:* ($k$-*truss*). The $k$-truss of $G$, denoted by $T_k$, where $k \geq 2$, is defined as the largest subgraph of $G$, such that every edge $e$ in $T_k$ has $support(e) \geq (k-2)$.

$k$-truss has close connections with the well known concept of $k$-cores. The $k$-core of $G$, denoted by $C_k$, where $k \geq 0$, is defined as the largest subgraph of $G$, such that each vertex $u$ in $C_k$ has $degree(u) \geq k$. Fig. 1(a) shows a simple undirected and unweighted graph $G$. By definition, the 2-truss is simply $G$ itself. Fig. 1(b) shows the 4-core of $G$ in which every vertex has a degree of at least 4. No 5-core of $G$ exists. Fig. 1(c) shows the 5-truss of $G$ in which every edge has a support of at least 3. No 6-truss of $G$ exists. It is interesting to note that the 5-truss also satisfies the requirement of a 4-core by definition. However, it is not true vice versa. This example shows that the $k$-truss can further filter out those marginal vertices and can better represent the core part of a graph than the $k$-core.

We introduce other two important notions related to $k$-truss: trussness and $k$-class.

*Definition 4:* (*trussness*). The trussness of an edge $e$, denoted by $\phi(e)$, is defined as the maximum $k$ such that $e$ belongs to $T_k$ but does not belong to $T_{k+1}$.

The maximum trussness of any edge in $G$ is denoted by $t_{max}$. Based on the trussness, we can define the $k$-class of $G$

as follows:

*Definition 5:* (*k-class*). The $k$-class of $G$ is defined as the set of edges with same trussness of $k$, denoted by $\Phi_k = \{e : e \in E, \phi(e) = k\}$.

For the $k$-truss decomposition problem, our task is to find the $k$-trusses of $G$ for all $2 \leq k \leq t_{max}$. The $k$-truss can be obtained by taking the union of $k$-classes of $G$ by $T_k = \Phi_k \cup \Phi_{k+1} \cup \cdots \cup \Phi_{t_{max}}$. For the graph shown in Fig. 1(a), the 2-class $\Phi_2$ is an empty set. The 3-class $\Phi_3$ has 6 edges: {(0, 1), (0, 4), (1, 5), (2, 5), (3, 10), (4, 10)}. The 4-class $\Phi_4$ has 6 edges: {(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)}. The 5-class $\Phi_5$ has 14 edges: {(5, 6), (5, 7), (5, 8), (5, 9), (5, 10), (6, 7), (6, 8), (6, 9), (6, 10), (7, 8), (7, 9), (7, 10), (8, 9), (9, 10)}. Therefore, from the $k$-classes above, we can obtain that $T_2$ and $T_3$ is $G$ itself. $T_4$ is $(\Phi_4 \cup \Phi_5)$ and $T_5$ is $\Phi_5$. We can easily verify that for $2 \leq k \leq 5$, each edge $e$ in $T_k$ is contained in at least $(k-2)$ triangles, meaning that $support(e) \geq (k-2)$.

To summarize, if we can compute the trussness for each edge in $G$, we can obtain the $k$-classes of $G$, and then we can obtain the $k$-trusses of $G$ by taking the union of the $k$-classes. Therefore, the $k$-truss decomposition of a graph is equivalent to computing the trussness of each edge in the graph.

## III. ALGORITHMS

We engineer two existing efficient $k$-truss decomposition algorithms: the serial algorithm based on edge peeling [14] and the parallel algorithm based on asynchronous $h$-index updating [22]. Both algorithms start with computing the initial support for each edge since the initial support is the upper bound of the trussness. We describe the initial support computation in III-A and the two $k$-truss decomposition algorithms in detail in III-B and III-C, respectively.

Since the memory usage is our major concern, we surely do not want to load the whole graph into the main memory during computation, especially for large networks. Therefore, we use WebGraph [24], a graph compression framework with high compression ratio that enables random access to the compressed graph, to make the graph's footprint as small as possible. We also design an array-based structure and corresponding operations to achieve the equivalent functionality of a hash table, but with a much smaller footprint. We introduce our memory optimization on the two $k$-truss decomposition algorithms using WebGraph and our carefully engineered data structures in III-D.

### A. Initial Support Computation

The initial support is the upper bound on the trussness of each edge. Its computation is based on triangle enumeration [20]. This algorithm visits each edge in the graph, finds all triangles starting with the edge, and updates the support for all edges contained in those triangles.

Since we define a triangle $\triangle_{uvw}$ based on the ordering of the three vertices ($u < v < w$), to find triangles starting with edge $(u, v)$, we do not need the complete neighbor sets of $u$ and $v$. We define the set of $u$'s neighbors with vertex ID $> u$ to be the upper neighbor set of $u$, denoted by $neighbor^+(u) = \{v : (u, v) \in E, v > u\}$. To find all triangles starting with $(u, v)$, we take the intersection of $neighbor^+(u)$ and $neighbor^+(v)$. $uvw$ forms a triangle $\triangle_{uvw}$ if $w \in \{neighbor^+(u) \cap neighbor^+(v)\}$.

**Algorithm 1** Initial Support Computation

```
1:  function SUPPORTCOMPUTE(G)
2:      for each edge e ∈ E do support[e] ← 0
3:      for each edge e ∈ E in parallel do
4:          u, v ← two endpoints of e
5:          for each w ∈ neighbor+(u) ∩ neighbor+(v) do
6:              e_uw ← get edge ID of (u, w)
7:              e_vw ← get edge ID of (v, w)
8:              atomicAdd(support[e], 1)
9:              atomicAdd(support[e_uw], 1)
10:             atomicAdd(support[e_vw], 1)
11:     return support
```

The initial support computation can be parallelized easily since the procedure of triangle enumeration is independent on different edges. Algorithm 1 summarizes the major steps of the initial support computation. For each edge $e \in E$, step 4 obtains the two endpoints $u$ and $v$ of the edge. Step 5 takes the intersection of $u$ and $v's$ upper neighbor sets. The size of the intersection is the number of triangles starting with edge $(u, v)$. Steps 6 and 7 obtain the edge IDs for $(u, w)$ and $(v, w)$ according to the endpoints. The original program [14] uses a hash table to store the edge IDs. The pair of two endpoints of an edge is used as the key for the hash table. The hash table is also used to check whether two vertices form an edge or not. Although it is convenient to use, for large graphs, the hash table is expensive to use in practice. Instead, we implement these two steps using binary search in the adjacency list of the graph. We use an auxiliary array to label the starting positions of each segment in the adjacency list. Therefore, we can perform binary search in a small range (not in the whole adjacency list), which can be very efficient. Steps 8–9 atomically increment the support of the three edges constituting the triangle $\triangle_{uvw}$ by 1.

### B. Serial Edge Peeling

The serial edge-peeling algorithm uses the output from Algorithm 1 to initialize the support for each edge. Algorithm 2 iteratively removes edges based on their support until all edges in the graph are removed.

Step 3 sorts the edges in ascending order of their support using a linear-time sort (*e.g.*, bin sort) and stores them (edge IDs) in the $sortedEdge$ array. Edges are processed exactly once under the ascending-order configuration. Step 5 obtains the edge $e$ (not removed) with the lowest support from $sortedEdge$. Step 7 ensures that $u$ has a smaller degree than $v$. The removal of edge $(u, v)$ affects the support of all edges that can constitute triangles with $(u, v)$. To find all triangles containing edge $(u, v)$, step 8 iterates $u's$ each neighbor $w$.

**Algorithm 2** Serial $K$-Truss Decomposition

```
1:  function k-TRUSS-SERIAL(G, support)
2:      k ← 2
3:      sort all the edges in ascending order of their support
        and store them in sortedEdge array
4:      for each edge e ∈ E such that support[e] ≤ k − 2 do
5:          e ← edge with the lowest support
6:          u, v ← two endpoints of e
7:          if degree[u] > degree[v] then swap u and v
8:          for each w ∈ neighbor(u) do
9:              e_uw ← get edge ID of (u, w)
10:             if (v, w) ∈ E then
11:                 e_vw ← get edge ID of (v, w)
12:                 if support[e_vw] > support[e] then
13:                     support[e_vw] ← support[e_vw] − 1
14:                     reorder sortedEdge
15:                 if support[e_uw] > support[e] then
16:                     support[e_uw] ← support[e_uw] − 1
17:                     reorder sortedEdge
18:         remove e from G.
19:     if not all edges in G are removed then
20:         k ← k + 1
21:         goto step 4
22:     for edge e ∈ E do
23:         trussness[e] ← support[e] + 2
24:     return trussness
```

**Algorithm 3** Parallel $K$-Truss Decomposition

```
1:  function k-TRUSS-PARALLEL(G, support)
2:      for each edge e ∈ E do
3:          h[e] ← support[e], scheduled[e] ← TRUE
4:      updated ← TRUE      ▷ TRUE if any h[e] is updated
5:      while updated do
6:          update ← FALSE
7:          for each edge e ∈ E in parallel do
8:              if scheduled[e] is FALSE then continue
9:              L ← empty list, N ← empty list
10:             for each △ contains e do
11:                 e′, e″ ← the two edges in △ other than e
12:                 N.add(e′), N.add(e″)
13:                 ρ ← min{h[e′], h[e″]}
14:                 L.add(ρ)
15:             H ← h-index of L
16:             if h[e] ≠ H then
17:                 updated ← TRUE
18:                 for each edge e_N in N do
19:                     if H < h[e_N] ≤ h[e] then
20:                         scheduled[e_N] ← TRUE
21:             h[e] ← H
22:             scheduled[e] ← FALSE
23:     for each edge e ∈ E do
24:         trussness[e] ← h[e] + 2
25:     return trussness
```

If $(v, w)$ is an edge, then $u$, $v$, and $w$ form a triangle. Step 10 checks whether $(v, w)$ is an edge or not. We implement this step by binary search in the adjacency list of the graph without a hash table, similar to the operation of obtaining an edge ID. If $(v, w)$ is an edge, then steps 13 and 16 decrement the support of edges $(v, w)$ and $(u, w)$ by 1, respectively. It should be noted that the decrement only applies to edges with support larger than edge $e's$ support. Since the support has been changed, steps 14 and 17 reorder the $sortedEdge$ array to maintain the ascending order with regard to the edge support. Constant-time reordering can be achieved using a method similar to the one used in the $k$-core decomposition [16]. After all triangles containing edge $(u, v)$ are processed, step 18 removes the edge $(u, v)$ from the graph. We do not physically delete the edge from the graph. Instead, we use a bit set to label each edge's state. After removing all edges in the current bin (containing edges with support equal to $k-2$), the program moves to the next bin (incrementing $k$ by 1). The program continues removing edges until all edges in the graph are removed. Step 23 adds 2 to the final support to obtain the trussness for each edge.

### *C. Asynchronous h-index updating*

The edge-peeling method requires processing edges in ascending order of their support, which makes the algorithm inherently sequential since each step depends on the result from the previous step. The asynchronous $h$-index updating algorithm [22] relaxes the "ascending order" requirement and processes edges in a random order, which makes the parallelization possible. The main idea of the algorithm is the iterative $h$-index computation on the support of the edges. $h$-index is a measure to quantify the impact and productivity of researchers by the number of citations of their publications. For a set of real numbers, the $h$-index of the set is defined as the largest number $h$ such that there are at least $h$ elements in the set that are at least $h$. For example, the $h$-index of {2, 2, 3, 3, 3} is 3. The algorithm extends the definition of neighbors and defines an edge's neighbors to be the edges that can form triangles with it. The $h$-index of an edge is fundamentally the same as the support of an edge and is upper bounded by the $h$-index of the edge's neighbor set. The algorithm iteratively updates an edge's $h$-index by computing the $h$-index of its neighbor set until achieving convergence when no updates would happen. The updating scheme is asynchronous, meaning the $h$-index of an edge is updated instantly and the computation of the $h$-index of the neighbor set always uses the up-to-date $h$-index values.

The major steps of the parallel algorithm are summarized in Algorithm 3. Step 3 initializes the $h$-index of each edge by the edge's initial support. We use a boolean indicator $updated$ to check the convergence and to terminate the program. $updated$ stays true if any edge's $h$-index is changed. The program processes each edge in parallel. For each edge $e$, step 10

finds all triangles containing the edge $e$. We use the same implementation as steps 8 and 10 of Algorithm 2, which uses binary search in the adjacency list of the graph without a hash table. Steps 12–14 collects the $h$-indices of $e$'s neighbor edges by choosing the smaller one. Step 15 computes the $h$-index of the neighbor set as the new $h$-index for edge $e$. If the new $h$-index is smaller than $e's$ current $h$-index, it means $e's$ $h$-index will be updated and step 17 sets $updated$ to be true. The algorithm uses a notification mechanism to achieve faster convergence. The $scheduled$ array notifies the program to process the scheduled edges only. Steps 18–20 decide whether a neighbor edge of $e$ is scheduled to be processed in the next iteration. If the neighbor edge's $h$-index is between $e$'s new $h$-index and $e's$ current $h$-index, it means that updating $e's$ $h$-index will affect the upper bound of the neighbor edge's $h$-index. The neighbor edge's $h$-index may have a chance to change. Therefore, it is scheduled to be processed in the next iteration. The program terminates when each edge's $h$-index achieves convergence. Step 24 adds 2 to the final $h$-index to obtain the trussness for each edge.

### D. Memory Optimization

The original implementations for Algorithms 2 and 3 keep the entire graph ($i.e.$, the adjacency list representation) in the main memory and use a hash table for checking the existence of an edge and querying for the edge ID, which is expensive and inefficient for large graphs. To eliminate the necessity of keeping the entire graph in the main memory, we use the WebGraph framework. In this section, we detail our data structure design to avoid using the hash table but still can achieve the equivalent functionality without any performance degradation.

Fig. 2 shows our optimized data structures for $k$-truss decomposition. We use a small simple graph as an example. For the example graph, $n = 6$ and $m = 9$. The edges of the graph can be stored in $edgeHead$ and $edgeTail$ arrays both with size $m$. Vertices in $edgeHead$ and $edgeTail$ are sorted in ascending order. $edgeID$ in the original programs is a hash table with $m$ entries. We show that we can remove the redundant $edgeID$ and $edgeHead$ without affecting any computation.

We introduce an *offset* array with size $n$ in our implementation. The *offset* array summarizes the $edgeHead$ information by recording the start position of each vertex in $edgeHead$. For example, vertex 0 starts at position 0, vertex 1 starts at position 1, and vertex 2 starts at position 3 in $edgeHead$. Therefore, *offset*$[0] = 0$, *offset*$[1] = 1$, and *offset*$[2] = 3$, respectively. The hash table has two functionalities: given two vertices $u$ and $v$, check if $(u, v)$ is an edge; given an edge $(u, v)$, get the edge ID. We show that only by $edgeTail$ and *offset*, we can still achieve the two functionalities. For checking if $(u, v)$ is an edge, we assert that $u < v$. Then we perform binary search in $edgeTail$ for $v$ in the range of [*offset*$[u]$, *offset*$[u + 1]$), which is $neighbor^+(u)$. If we can find $v$ in this range, then $(u, v)$ is an edge. For example, if we want to check if (2, 5) is an edge, we can search for 5 in range [3, 6) in $edgeTail$. The other functionality is to get the edge ID given the two endpoints. The process is similar since the binary search would return the target index.

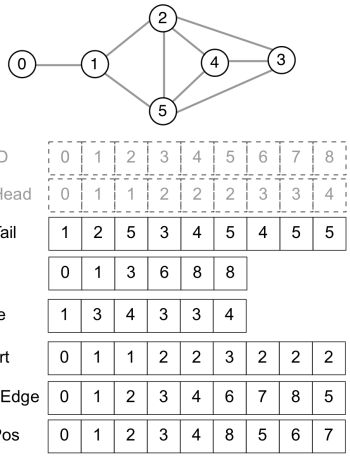


Fig. 2. Optimized data structures for $k$-truss decomposition.

$sortedEdge$ is used to store edges in ascending order of their support. $edgePos$ is used to store the index of an edge in $sortedEdge$. For example, $edgePos[5] = 8$, meaning that edge 5 is at position 8 in $sortedEdge$ ($sortedEdge[8] = 5$). For Step 10 in Algorithm 2 and Step 11 in Algorithm 3, we still need the full neighbor set of a vertex. Since we cannot get the complete neighbor set from $edgeTail$, we therefore use WebGraph for random access to the compressed graph to retrieve the complete neighbor set of a vertex. Since Algorithm 3 processes edges in a random order, there is no need to keep $sortedEdge$ and $edgePos$. However, the $scheduled$ needs another $m$-sized array.

By eliminating $edgeID$ and $edgeHead$, we optimize the memory usage down to $(4m + 2n)$ for Algorithm 2 and $(3m+2n)$ for Algorithm 3, compared to the original programs' $(6m + n)$ and $(5m + n)$ memory usage (no *offset* needed). Since m is usually very large, the memory reduction would be significant for large graphs. For example, for a graph with 41 million vertices and 1.2 billion edges, the original programs for Algorithm 2 and 3 would consume at least 29.0 GB and 24.1 GB memory, if we assume the unit is an integer. Our optimized programs only need 19.5 GB and 14.7 GB memory, respectively. It should be noted that an $m$-entry hash table usually has a larger memory footprint than an $m$-length array. Therefore, in practice, the memory reduction should be larger than the theoretical calculation.

WebGraph plays an important role in reducing the memory usage. Without WebGraph, we have to keep the complete graph ($i.e.$, the adjacency list representation) in the memory to retrieve the neighbor set of a vertex. By the WebGraph's API, we can efficiently get access to the compressed graph stored on the hard drive. Therefore, WebGraph eliminates the necessity of maintaining the complete graph in the main memory, allowing us to remove the $edgeHead$ array.

## IV. EXPERIMENTAL METHODOLOGY

TABLE I
SUMMARY OF DATASETS

| Graph | $\|V\|$ | $\|E\|$ | $t_{max}$ |
|---|---|---|---|
| $cnr$-2000 | 325 557 | 2 738 969 | 84 |
| $dblp$-2011 | 986 324 | 3 353 618 | 119 |
| $ljournal$-2008 | 5 363 260 | 49 514 271 | 414 |
| $arabic$-2005 | 22 744 080 | 553 903 073 | 3248 |
| $uk$-2005 | 39 459 925 | 783 027 125 | 589 |
| $twitter$-2010 | 41 652 230 | 1 202 513 046 | 1998 |

We implement the initial support computation, the serial edge peeling, and the asynchronous $h$-index updating algorithms in Java. Experiments are conducted on a machine with Intel quad-core i7, 2.6 GHz CPU, 32 GB RAM, and 500 GB SSD hard drive, running Ubuntu 17.10. The cost for this machine is less than 1,500 Canadian dollars, thus qualifying as a consumer-grade machine.

We perform experiments on the following six datasets: a small crawl of the National Research Council of Italy domain from 2000 ($cnr$-2000); a network modeling the collaboration of scientists using the bibliography service DBLP in 2011 ($dblp$-2011); a snapshot of the LiveJournal social network from 2008 ($ljournal$-2008); a crawl in 2005 of websites potentially containing pages written in Arabic ($arabic$-2005); a crawl of the .uk domain in 2005 ($uk$-2005); a crawl of the Twitter social network from 2010 ($twitter$-2010). All datasets are obtained from http://law.di.unimi.it/datasets.php. Characteristics of the datasets are summarized in Table 1. For both the serial and the parallel $k$-truss decomposition programs, we allocate 4 GB memory for $cnr$-2000, $dblp$-2011, and $ljournal$-2008, 16 GB for $arabic$-2005 and $uk$-2005. For $twitter$-2010, we allocate 22 GB memory for the serial program and 16 GB memory for the parallel program.

Directed graphs are converted to undirected graphs by taking the union with the directed graph's transposed graph. Self-loops in the graph are removed to ensure the simple graph prerequisite.

## V. RESULTS AND ANALYSIS

We present our experimental results on the runtime, maximum trussness, and trussness distributions for different datasets in II. We investigate two schemes to further reduce the memory usage and evaluate the performance of the two schemes in V-B.

### A. Performance Results

Fig. 3 shows the runtime of the initial support computation, optimized serial, and optimized parallel $k$-truss decomposition for different datasets. For small datasets such as $cnr$-2000 and $dblp$-2011, all algorithms can finish in 30 $s$. For the medium-sized $ljournal$-2008, all algorithms can finish in 15 $min$. However, for large datasets such as $arabic$-2005, $uk$-2005, and $twitter$-2010, the runtime increases significantly. Especially for $twitter$-2010, the initial support computation takes 9 $h$, the serial $k$-truss decomposition takes 50 $h$, and the parallel $k$-truss decomposition takes 203 $h$. Given the large size of the dataset and the limited computation ability of our machine, the runtime for $twitter$-2010 is still acceptable. We also notice that in most cases, the parallel algorithm even runs slower than the serial algorithm. Since the parallel algorithm processes edges in a random order, it requires many iterations to achieve convergence. By contrast, the serial algorithm processes edges in ascending order of their support. Therefore, it can achieve convergence with only one iteration since it processes each edge for only once. The performance of the parallel algorithm is limited by the CPU of our machine. The parallel algorithm would eventually outperform the serial algorithm on a multi-core server ($e.g.$, 24-core).

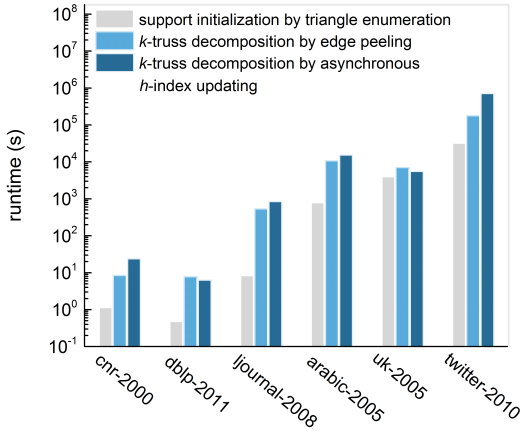


Fig. 3. Runtime of initial support computation, optimized serial, and optimized parallel $k$-truss decomposition for different datasets.

TABLE II
RUNTIME OF ALGORITHM 2 WITH DIFFERENT IMPLEMENTATIONS

| Runtime (s) | $cnr$-2000 | $dblp$-2011 | $ljournal$-2008 |
|---|---|---|---|
| HashMap | 1371 | 47 | OutOfMemoryError |
| Optimized | 10 | 9 | 555 |
| Speedup ratio | 137.1 | 5.2 | – |

We implement Algorithm 2 (serial edge peeling) using HashMap in Java standard library[1] as the baseline to compare the performance with our optimized implementation using array-based data structures and WebGraph. Table II shows the runtime comparison. The HashMap implementation takes 1371 $s$ for $cnr$-2000 and 47 $s$ for $dblp$-2011 while our optimized implementation only takes 10 $s$ for $cnr$-2000 and 9 $s$ for $dblp$-2011, showing a significant speedup. It should be noted that it is the use of array-based structures that contributes to the speedup. The HashMap implementation encounters "OutOfMemoryError" when computing $ljournal$-2008 with the same memory allocation as our optimized implementation (4 GB). The same error occurs when computing larger graphs such as $arabic$-2005, $uk$-2005, or $twitter$-2010

[1]https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

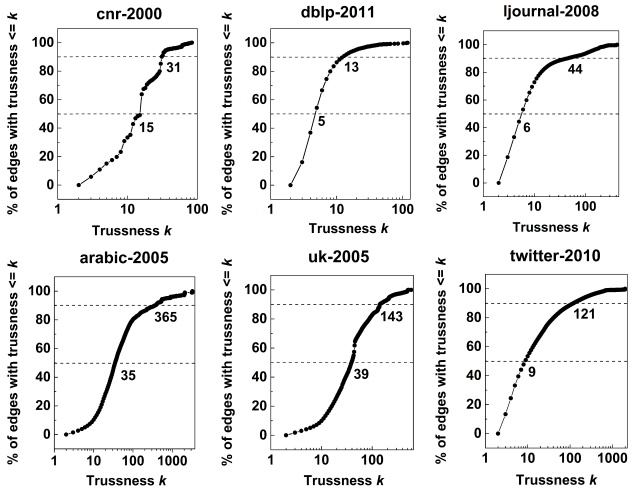


Fig. 4. Trussness distributions for different datasets.

using the HashMap implementation under the same memory allocation condition. Results show the HashMap implementation is neither time nor memory efficient compared to our optimized implementation. We attribute the inefficiency to the increased lookup time in the hash map caused by the potential key collisions. Algorithm 3 (asynchronous $h$-index updating) would suffer the same problem if using the HashMap implementation. Therefore, we did not specifically implement Algorithm 3 using HashMap for performance comparison with our optimized implementation.

Fig. 4 shows the trussness distributions for different datasets. The maximum trussness for different datasets are summarized in Table 1. The distributions show that for all datasets, edges with high trussness are only a small percentage of the total edges. The majority of edges have low trussness. For example, in $cnr$-2000 with $t_{max} = 84$, 50% of edges have trussness less than or equal to 15 and 90% of edges have trussness less than or equal to 31.

### *B. Schemes to Further Reduce Memory Usage*

In this section, we investigate two schemes to further reduce the memory usage and evaluate their performance. One is to remove the $edgeTail$ array. The other is to remove the $edgePos$ array. Both schemes, described in detail, below can reduce the memory usage from $(4m + 2n)$ to $(3m + 2n)$ for Algorithm 2 and from $(3m+2n)$ to $(2m+2n)$ for Algorithm 3. However, as we will show, both schemes would result in a significant performance drop for the runtime and scalability. This suggests that the data structures proposed in III-C have the minimum memory requirement for Algorithms 2 and 3 in order to have good performance, and these are $(4m + 2n)$ for Algorithm 2 and $(3m + 2n)$ for Algorithm 3.

The first scheme is to remove $edgeTail$ array shown in Fig. 2. We can use Webgraph's API to export $edgeTail$ to a new compressed graph in which each vertex only has neighbors with larger vertex ID. We implement Algorithm 2 using this scheme. Results show that removing $edgeTail$, the runtime for the $k$-truss decomposition of $cnr$-2000 is significantly increased from 10 $s$ to 1735 $s$. We do not attempt to run this implementation on other larger graphs. Removing $edgeTail$ would affect both procedures of checking the existence of an edge and getting the edge ID since both procedures depend on the binary search in the neighbor sets contained in $edgeTail$. Without $edgeTail$ array, each binary search operation requires random access to the compressed graph (including the decompression from the compressed graph and disk I/O), which is very time consuming. Therefore, removing $edgeTail$ is not of practical use, which also applies to Algorithm 3.

The other scheme is to remove the $edgePos$ array shown in Fig. 2. $edgePos$ is used to store the position of an edge in $sortedEdge$ array (see the explanation in III-C). If we remove $edgePos$, we have to search for a given edge in $sortedEdge$ since we lose the convenience of getting the edge position directly from $edgePos$. Since the binary search operation requires a sorted array, we have to maintain $sortedEdge$ sorted with regard to the edge ID in each support segment. Each time when the support of an edge is decreased, we have to move this edge to the corresponding support segment in $sortedEdge$ and insert the edge to the right position in order to maintain the support segment sorted. The insertion operation involves moving a large amount of elements in $sortedEdge$, which would increase the runtime. We implement Algorithm 2 using this scheme. Results show that the $k$-truss decomposition of $cnr$-2000 takes 145 $s$, much faster than the first scheme (1735 $s$) but still 14$\times$ slower than our optimized implementation (10 $s$). For larger graphs such as $ljournal$-2008, the performance becomes worse as the truss decomposition takes 37496 $s$ (67 $\times$ slower than the optimized implementation of 555 $s$), showing poor scalability.

## VI. RELATED WORK

Cohen [5] first introduced the $k$-truss concept as a relaxation of a clique for social network analysis. He proposed a serial edge-peeling algorithm requiring $O(\sum_{u \in G} degree(u)^2)$ time. J. Wang et al. [14] optimized the edge-peeling algorithm by introducing an extra step (step 7 in Algorithm 2). They proved that the time complexity could be reduced to $O(m^{1.5})$ by adding this step. Algorithm 2 is based on Wang's optimized algorithm.

Thereafter, several parallel $k$-truss algorithms were proposed to parallelize the serial algorithm. H. Kabir and K. Madduri [20] proposed a shared-memory parallel program using a level-synchronous parallelization on Wang's serial algorithm [14] for $k$-truss decomposition. The parallelization is similar to ParK [25], a parallel algorithm for $k$-core decomposition. In their implementation, they did not use a hash table to maintain edges in the graph. Instead, they used array-based structures to achieve memory efficiency and safe concurrent updates. A. E. Sariyuce et al. [22] proposed the parallel nucleus decomposition, a generic graph decomposition framework that generalizes $k$-core and $k$-truss decomposition and can use higher-order structures such as cliques [10] to

discover cohesive subgraphs. They generalized the iterative $h$-index computation for any parallel nucleus decomposition. Algorithm 3 is based on this generic algorithm. C. Voegele et al. [23] proposed a parallel graph-centric $k$-truss decomposition algorithm using the relation between a $k$-truss and a $k$-core. $k$-truss is closely related to $k$-core. For example, a $k$-truss is always a $(k-1)$-core; however, it is not true vice versa [5]. It means that a $k$-truss is a subgraph of a $(k-1)$-core. The algorithm first computes the $(k-1)$-core, which would eliminate a large number of nodes and the edges connected to them. Then the algorithm computes the $k$-truss on the resultant graph, which is faster than the direct computation from the original graph. Most recently, S. Smith et al. [21] proposed a shared-memory parallel $k$-truss decomposition algorithm based on the edge peeling. They split the edge peeling process into multiple bulk-synchronous substeps. They showed their program's ability to efficiently decompose a graph with 1.2 billion edges.

All the parallel algorithms mentioned above are designed for high-performance computing systems. The memory consumption is high for these programs. For example, the program in [20] consumes $(7m + 2n)$ memory and is claimed to be memory efficient since the implementation does not use a hash table to maintain edges of the graph. For a graph with 41 million vertices and 1.2 billion edges, the memory consumption is 34 GB in theory, assuming the unit is a 4-byte integer. By contrast, our optimized serial and parallel algorithms only need 22 GB and 16 GB memory in practice for the same-sized graph, showing higher memory efficiency.

## VII. CONCLUSIONS

To conclude, we engineered two $k$-truss decomposition algorithms to achieve smaller memory usage and better performance. We showed that the use of a hash table suggested by the original programs is both time and memory consuming in the practical implementation. We optimized the data structure design by using array-based structures. We also designed corresponding operations on the array-based structures to achieve the same functionality as a hash table but with a much smaller memory footprint and better performance. With the help of WebGraph, we managed to significantly reduce the memory usage. We performed a thorough experimental study on different datasets using our optimized implementations. We showed that it is viable to compute the $k$-truss decomposition of large networks (up to 1.2 billion edges) on a consumer-grade machine within a reasonable amount of time.

In addition, we investigated two schemes to further reduce the memory usage. Results show that any further memory reduction would cause significant degradation on the runtime and scalability of the $k$-truss decomposition programs, which proves that our carefully engineered data structures are indeed the optimum design to be both time and memory efficient.

## REFERENCES


[1] G. Zhao and J. Yuan, "Discovering thematic patterns in videos via cohesive sub-graph mining," in *IEEE International Conference on Data Mining*, 2011, pp. 1260–1265.

[2] Y. Shao, L. Chen, and B. Cui, "Efficient cohesive subgraphs detection in parallel," in *Proceedings of the ACM International Conference on Management of Data*, 2014, pp. 613–624.

[3] N. Wang, S. Parthasarathy, K. L. Tan, and A. K. Tung, "Csv: visualizing and mining cohesive subgraphs," in *Proceedings of the ACM International Conference on Management of Data*, 2008, pp. 445–458.

[4] F. Moser, R. Colak, A. Rafiey, and M. Ester, "Mining cohesive patterns from graphs with feature vectors," in *Proceedings of the SIAM International Conference on Data Mining*, 2009, pp. 593–604

[5] J. Cohen, "Trusses: Cohesive subgraphs for social network analysis," *National Security Agency Technical Report*, vol. 16, 2008.

[6] 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 ACM International Conference on Management of Data*, 2014, pp. 1311–1322.

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

[8] X. Huang, W. Lu, and L. V. Lakshmanan, "Truss decomposition of probabilistic graphs: Semantics and algorithms," in *Proceedings of the ACM International Conference on Management of Data*, 2016, pp. 77–90.

[9] A. M. Katunka, C. Yan, K. B. Serge, and Z. Zhang, "K-Truss Based Top-Communities Search in Large Graphs," in *International Conference on Advanced Cloud and Big Data*, 2017, pp. 244–249.

[10] C. Bron and J. Kerbosch, "Algorithm 457: finding all cliques of an undirected graph," *Communications of the ACM*, vol. 16, no. 9, pp.575–577, 1973.

[11] F. Zhao and A. K. Tung, "Large scale cohesive subgraphs discovery for social network visual analysis," in *Proceedings of the VLDB Endowment*, 2012, vol. 6, no. 2, pp. 85–96.

[12] X. Huang, L. V. Lakshmanan, J. X. Yu, and H. Cheng, "Approximate closest community search in networks," in *Proceedings of the VLDB Endowment*, 2005, vol. 9, no. 4, pp.276–287.

[13] A. Verma, A. Buchanan, and S. Butenko, "Solving the maximum clique and vertex coloring problems on very large sparse networks," in textitINFORMS Journal on Computing, vol. 27, no. 1, pp.164–177, 2015.

[14] J. Wang and J. Cheng, "Truss decomposition in massive networks," in *Proceedings of the VLDB Endowment*, 2012, vol. 5, no. 9, pp. 812-823.

[15] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo, "K-core decomposition of large networks on a single PC," in *Proceedings of the VLDB Endowment*, 2015, vol. 9, no. 1, pp.13–23.

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

[17] X. Hu, F. Liu, V. Srinivasan, and A. Thomo, "k-core Decomposition on Giraph and GraphChi," in *International Conference on Intelligent Networking and Collaborative Systems*, 2017, pp. 274–284.

[18] B. Tootoonchi, V. Srinivasan, and A. Thomo, "Efficient implementation of anchored 2-core algorithm," in *Proceedings of the IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, 2017, pp. 1009–1016.

[19] H. Zhang, H. Zhao, W. Cai, M. Zhao, and G. Luo, "Visualization and cognition of large-scale software structure using the k-core analysis," in *IEEE International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, 2008, pp. 954–957.

[20] H. Kabir and K. Madduri, "Shared-memory graph truss decomposition," *arXiv preprint arXiv:1707.02000*, 2017.

[21] S. Smith, X. Liu, N. K., Ahmed, A. S. Tom, F. Petrini and G. Karypis, "Truss decomposition on shared-memory parallel systems," in *IEEE High Performance Extreme Computing Conference*, 2017.

[22] A. E. Sariyuce, C. Seshadhri, and A. Pinar, "Parallel local algorithms for core, truss, and nucleus decompositions," *arXiv preprint arXiv:1704.00386*, 2017.

[23] C. Voegele, Y. S. Lu, S. Pai and K. Pingali, "Parallel triangle counting and k-truss identification using graph-centric methods," in *IEEE High Performance Extreme Computing Conference*, 2017.

[24] P. Boldi and S. Vigna, "The webgraph framework I: compression techniques," in *Proceedings of the ACM International Conference on World Wide Web*, 2004, pp. 595–602.

[25] N. S. Dasari, R. Desh, and M. Zubair,2014, October. "ParK: An efficient algorithm for k-core decomposition on multicore processors," in *IEEE International Conference on Big Data*, 2014, pp. 9–16.