On Breaking Truss-Based Communities

Huiping Chen
King's College London, UK
huiping.chen@kcl.ac.uk

Alessio Conte
Università di Pisa, Italy
alessio.conte@unipi.it

Roberto Grossi
Università di Pisa, Italy
roberto.grossi@unipi.it

Grigorios Loukides
King's College London, UK
grigorios.loukides@kcl.ac.uk

Solon P. Pissis
CWI, The Netherlands
Vrije Universiteit, The Netherlands
solon.pissis@cwi.nl

Michelle Sweering
CWI, The Netherlands
michelle.sweering@cwi.nl

Abstract

A k-truss is a graph such that each edge is contained in at least $k-2$ triangles. This notion has attracted much attention, because it models meaningful cohesive subgraphs of a graph. We introduce the problem of identifying a smallest *edge* subset of a given graph whose removal makes the graph k-truss-free. We also introduce a problem variant where the identified subset contains only edges incident to a given set of nodes and ensures that these nodes are not contained in any k-truss. These problems are directly applicable in communication networks: the identified edges correspond to *vital* network connections; or in social networks: the identified edges can be *hidden* by users or *sanitized* from the output graph. We show that these problems are NP-hard. We thus develop exact exponential-time algorithms to solve them. To process large networks, we also develop heuristics sped up by an efficient data structure for updating the truss decomposition under edge deletions. We complement our heuristics with a lower bound on the size of an optimal solution to rigorously evaluate their effectiveness. Extensive experiments on 10 real-world graphs show that our heuristics are effective (close to the optimal or to the lower bound) and also efficient (up to two orders of magnitude faster than a natural baseline).

CCS Concepts

• **Mathematics of computing** → **Graph algorithms**; • **Information systems** → **Data mining**.

Keywords

graph algorithm, k-truss, community detection

ACM Reference Format:
Huiping Chen, Alessio Conte, Roberto Grossi, Grigorios Loukides, Solon P. Pissis, and Michelle Sweering. 2021. On Breaking Truss-Based Communities. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '21), August 14–18, 2021, Virtual Event, Singapore.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3447548.3467365

KDD '21, August 14–18, 2021, Virtual Event, Singapore.

ACM ISBN 978-1-4503-8332-5/21/08.
https://doi.org/10.1145/3447548.3467365

1 Introduction

Graphs naturally model relationships between entities in a multitude of domains such as social networks, communication networks, or the web. A fundamental data analysis task in these domains is *community detection* (i.e., the identification of cohesive or dense subgraphs of a given graph), which employs different notions of graph community; these include the notions of k-plex, n-clan, n-club, k-core, k-ECC, and k-truss [10, 12]. These notions relax the classic notion of clique [15] (i.e., the ideal situation, where all nodes are pairwise connected), either to capture practical application considerations [10] or to enable more efficient enumeration [6].

Regardless of the community notion, the community structure is a key property of a graph. It is therefore essential to study how such a structure can be maintained or broken [9, 16, 20, 24, 27]. Here we investigate the following general problem.

Community Breaking (CB) problem: Given an undirected graph $G(V, E)$, a set of nodes $U \subseteq V$, and a notion of community, identify a smallest subset E' of E, so that no community in $G' = G(V, E \setminus E')$ contains a node in U.

The CB problem is motivated by the following real-world applications:

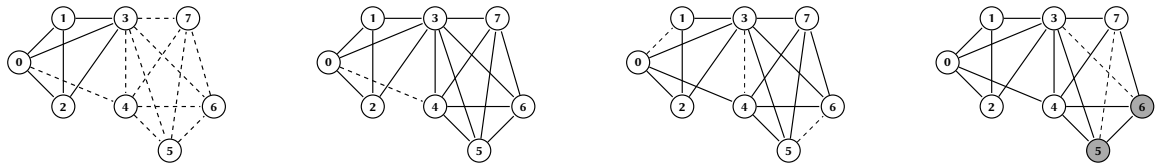
A1. Maintaining communities in social networks [27]. The edges identified in the output of CB correspond to critical edges for maintaining user engagement in communities.

A2. Assessing resilience to attacks or errors in communication networks [16]. The edges identified in the output of CB correspond to vital connections in the network.

A3. Enabling social network users to hide friendships, so that they are not seen as belonging to communities that could lead to their discrimination or unwanted targeted advertisement (e.g., through friend-based profile attribute inference attacks [3]). The edges identified in the output of the CB problem correspond to friendships users could opt to hide [9].

A4. Preventing the detection of confidential communities by sanitizing a graph prior to its dissemination, in the spirit of sanitization works on transaction [21] or sequential data [4]. The edges identified in the output of the CB problem must be removed to hide these communities in the sanitized graph.

Identifying a *small* edge subset is natural yet crucial. For example, in A1 and A2, it allows less costly maintenance of user engagement and network infrastructure improvements, respectively. In A3 and A4, it allows less effort from users and more accurate analysis of the sanitized graph, respectively.



(a) 4**-truss** **(b) Maximal** 4**-truss** **(c)** 4**-truss-free graph** **(d)** 4**-truss-free nodes**

Figure 1: (a) The subgraph induced by the edges $(0,1)$**,** $(0,2)$**,** $(0,3)$**,** $(1,2)$**,** $(1,3)$**,** $(2,3)$ **is a** 4**-truss because every edge of the subgraph is contained in at least** $4-2=2$ **triangles of the subgraph. (b) The subgraph induced by all edges except the (dashed) edge** $(0,4)$ **is the maximal** 4**-truss of the graph. (c) The graph obtained after removing the set** $\{(0,1),(3,4),(5,6)\}$ **of (dashed) edges contains no** 4**-truss. (d) The graph obtained after removing the set** $\{(3,6),(5,7)\}$ **of (dashed) edges is a graph in which the (gray) nodes** 5 **and** 6 **are not contained in any** 4**-truss.**

We focus on the community notion of k-truss [6]. A *k-truss* is a subgraph of a graph such that each edge is contained in at least $k-2$ triangles of the subgraph (see Fig. 1a). A *maximal k-truss* is the largest *k-truss* of the graph (see Fig. 1b). The notion of k-truss has attracted significant interest because k-trusses: (1) are less expensive to enumerate than cliques, k-plexes, n-clans, and n-clubs, as well as more cohesive than k-cores and k-ECCs [10]; and (2) they model meaningful cohesive subgraphs in communication [27], social [13], or collaboration [13] networks thanks to good structural properties such as bounded diameter or strong decomposability [10].

Based on the CB problem, we define two combinatorial optimization problems of practical importance.

MIN-k-TBS (Minimum k-Truss Breaking Set) problem: Given an undirected graph $G(V,E)$ and a parameter k, find a smallest subset E' of E such that $G(V,E\setminus E')$ contains no k-truss. MIN-k-TBS is obtained from the CB problem by considering communities based on the notion of k-truss and $U=V$. MIN-k-TBS addresses applications A1 and A2 above.

Example 1.1. An optimal solution to MIN-k-TBS with $k=4$ on the graph of Fig. 1c is the set of (dashed) edges $E'=\{(0,1),(3,4),(5,6)\}$. This is because removing these edges leads to a graph with no 4-truss and because removing any fewer edges leads to a graph that contains a 4-truss.

MIN-k-CBS (Minimum k-Communities Breaking Set) problem: Given an undirected graph $G(V,E)$, a parameter k, and a set $U\subseteq V$, find a smallest subset E' of E such that the edges in E' are incident to nodes in U and no k-truss in $G(V,E\setminus E')$ contains a node in U. MIN-k-CBS is obtained from the CB problem by considering communities based on the notion of k-truss, having U as input, and further limiting what edges can be removed. It addresses applications A3 and A4 above. In particular, the requirement for edges in E' to be incident to nodes in U is necessary for A3, where users can only hide their own edges. Waiving this requirement for A4 is trivial.

Example 1.2. An optimal solution to MIN-k-CBS with $k=4$ and $U=\{5,6\}$ on the graph of Fig. 1d is the set of edges $E'=\{(3,6),(5,7)\}$. This is because removing these two edges leads to a graph in which neither node 5 nor node 6 belongs to a 4-truss, and because removing any single edge does not prevent both of these nodes from belonging to a 4-truss.

These problems are *intuitively* challenging: there are up to $2^{|E|}$ edge subsets that one may consider; and k-trusses have a hierarchical structure (i.e., a $(k+i)$ truss, for any k and i, is also a k-truss and contains up to $\binom{k+i}{k}$ smaller k-trusses).

Our Contributions and Paper Organization.

1. We define the MIN-k-TBS and MIN-k-CBS problems and show that they are both NP-hard. See Section 3.

2. We show a data structure for maintaining the *truss decomposition* of a graph (i.e., the maximal k-truss for each k) under edge deletions with theoretical guarantees. This is the backbone of our heuristics. We also provide the necessary set of tools for designing our algorithms. See Section 3.

3. We develop exact algorithms for both MIN-k-TBS and MIN-k-CBS. These algorithms are useful for evaluating our heuristics on *small-scale* graphs. See Section 4.

4. We develop three heuristics for MIN-k-TBS; namely, MBH_S, MBH_C, and SNH. The heuristics are based on different theoretical insights and can be trivially adapted to solve MIN-k-CBS. They always return a feasible solution. See Section 5.

5. We show a non-trivial lower bound on the size of an optimal solution to MIN-k-TBS and develop an efficient algorithm to compute it. The lower bound is useful for rigorously evaluating our heuristics on *large-scale* graphs and also for quickly assessing the quality of any solution on a graph of interest before executing any heuristic. This is because an optimal solution always lies in between the lower bound and the solutions produced by heuristics. See Section 6.

6. We perform an extensive experimental evaluation using 10 real-world datasets with up to millions of edges, as well as using 1,000 small-scale synthetic datasets. The evaluation shows that our heuristics produce near-optimal solutions and outperform two natural baselines. See Section 8.

The rest of the paper is structured as follows. In Section 2, we highlight the applicability of our algorithms via analyzing a real social network. We discuss related work in Section 7.

2 Analyzing Real Social Networks

We used a dataset [23] that contains friendship and location information from 114,324 Twitter users who recorded their check-ins in 3,820,891 Foursquare venues. We refer to this dataset as FL (for Friendship-Location). FL can be viewed as a node-labeled graph with 114,324 nodes and 607,327 edges. Each node corresponds to a user, each node label contains the check-ins of the user, and each edge corresponds to a friendship between two users.

FL contains check-ins to venues that may indicate users' sexual orientation, religious beliefs, or gambling habits. Moreover, the k-trusses in FL contain a large number of users who checked-in

to such venues, or are friends with users who checked in to such venues. For example, an 18-truss contained: (1) $115/280 = 41.1\%$ of users who checked-in to `gay bars`; and (2) $7786/11732 = 66.4\%$ of pairs of friends, at least one of whom checked-in to `gay bars`. Similarly, a 24-truss contained: (1) $57/131 = 43.5\%$ of users who checked-in to `casinos` or `strip clubs`; and (2) $2973/4385 = 67.8\%$ of pairs of friends, at least one of whom checked-in to a `casino` or `strip club`.

Some users may not want to be seen as belonging to such k-trusses to avoid being discriminated based on the *homophily* theory: friends are more likely to share attributes [3]. These users can achieve this goal by employing our algorithms for MIN-k-CBS to hide a small number of their friendships (e.g., by setting them to "private"), so that they are not seen as belonging to these k-trusses [9]. In fact, when applied with $k = 18$ and U containing 65 randomly selected nodes among those in the aforementioned 18-truss, our algorithms ensured that the 18-truss contains no nodes in U. Thus, the users corresponding to nodes in U are not seen as belonging to that truss. Specifically, MBH_S, MBH_C, and SNH achieved this by removing only 0.057%, 0.112%, and 0.056% of the edges of the graph, in 1.6, 1.7, and 13.7 seconds, respectively. Similarly, when applied with $k = 24$ and U containing 13 randomly selected nodes among those in the aforementioned 24-truss, MBH_S, MBH_C, and SNH ensured that no 24-truss containing nodes in U exists by removing only 0.012%, 0.016%, and 0.008% of the edges of the graph, in 0.5, 0.6, and 1.2 seconds, respectively. The results highlight the applicability of our methods in realistic settings.

3 Preliminaries and Techniques

We fix an undirected graph $G(V, E)$, with no multiple edges or self-loops. By $N_G(u)$ we denote the set of neighbors of a node $u \in V$ and by $|N_G(u)|$ its degree. A subgraph of G is defined by a set $S \subseteq E$; we use S to represent the set of edges or the subgraph of G induced by S. A triangle in G is a subgraph of three edges $\{e, f, g\}$ connecting three distinct nodes in V.

Given a subgraph S of G, the *support* of an edge e in S is the number of triangles of S that contain e and is denoted by $\sup_S(e)$. For an integer $k \geq 3$, a *k-truss* of G is a subgraph S of G such that every edge e in S has $\sup_S(e) \geq k - 2$. The largest such subgraph (not necessarily connected) is called the *maximal k-truss* of G. A *minimal* k-truss of G is a subgraph S of G such that S is a k-truss, but no proper subset $S' \subset S$ is a k-truss. The trussness of G, denoted by $t(G)$, is the largest k such that there exists a k-truss in G, and the maximal k-truss with $k = t(G)$ is called the *max-truss* of G.

Given an edge e, its *edge trussnness* $t(e)$ is the largest k for which e belongs to a k-truss. The *truss decomposition* of G associates to each e its trussnness $t(e)$; equivalently, it is the set comprised of the maximal k-truss of G, for each k. It can be computed in $O(|E|^{\frac{3}{2}})$ time, e.g., using the algorithm of [7].

The number of triangles of G is denoted by $\mathcal{T}_G$. All triangles of G can be computed in $O(|E|^{\frac{3}{2}})$ time [1]. The *trussness of a triangle* is the minimum among the trussness of its edges.

A *k-Truss Breaking Set* (k-TBS) of G is a set $E' \subseteq E$ such that the graph $G(V, E \setminus E')$ contains no k-truss. MIN-k-TBS, the core problem we introduce, is to find a *smallest k-TBS*:

Problem 3.1 (Minimal k-Truss Breaking Set (MIN-k-TBS))**.** Given a graph $G(V, E)$ and an integer $k \geq 3$, find a smallest k-truss breaking set of G.

Problem 3.2 (Minimal k-Communities Breaking Set (MIN-k-CBS))**.** Given a graph $G(V, E)$, an integer $k \geq 3$, and a set $U \subseteq V$, find a smallest set $E' \subseteq E$ comprised of edges incident to nodes in U, so that no k-truss in the graph $G(V, E \setminus E')$ contains a node in U.

We show the following theorem; the proof is in Appendix A.

Theorem 3.1. *For every $k \geq 3$, MIN-k-TBS is NP-hard.*

Note that MIN-k-TBS is the special case of MIN-k-CBS with $U = V$, i.e., no user wants to belong to any k-truss. Thus, the following corollary holds.

Corollary 3.2. *For every $k \geq 3$, MIN-k-CBS is NP-hard.*

3.1 Combinatorial Properties

We show some combinatorial properties of k-trusses, employed by our algorithms (see Appendix A for the proofs).

Lemma 3.3. *All nodes in a k-truss have degree at least $k - 1$.*

Corollary 3.4. *A k-truss with v nodes satisfies $v \geq k$ and has at least $v \cdot (k - 1)/2$ edges.*

Lemma 3.5. *If a k-truss S contains no k-truss after removing any of its edges, then S is a minimal k-truss.*

Lemma 3.6. *Let S be a k-truss but not a $(k + 1)$-truss. Then there exists an edge e in S such that $S \setminus e$ is not a k-truss.*

Let us remark that although $S \setminus e$ is not a k-truss, it may still *contain* a smaller k-truss. Finally, we recall from [7] a useful property that lets us bound the trussness of a graph by the number of triangles and edges it contains.

Theorem 3.7 ([7]). *Given a graph G with m edges and $\mathcal{T}_G$ triangles, its trussness $t(G)$ is at least $\frac{\mathcal{T}_G}{m} + 2$.*

3.2 Triangles Update

`Triangles Update` is a simple data structure to maintain the triangles of G under edge deletions. We construct a perfect hash table H in which the set of keys is the set of edges in G that are contained in at least one triangle. The value $H[e]$ for key (edge) e is the linked list of triangles $\{e, f, g\}$ containing e, for all f, g. Triangle $\{e, f, g\}$ exists also in $H[f]$ and $H[g]$. Element $\{e, f, g\}$ in $H[e]$ has two pointers to the elements in $H[f]$ and $H[g]$ representing $\{e, f, g\}$. The size of this data structure is in $O(\mathcal{T}_G)$. Constructing it takes $O(|E|^{\frac{3}{2}})$ time; the time to compute all triangles. Upon deleting edge e, we can:

1. Find the list of triangles $\{e, f, g\}$, for all f, g, in time linear in the size of this list. This is precisely $H[e]$.

2. Remove elements $\{e, f, g\}$, for all f, g, from $H[e]$, $H[f]$, and $H[g]$ in time linear in the size of the elements we remove. We do this by traversing $H[e]$ and updating the pointers.

3.3 Truss Decomposition Update

A practical algorithm for updating the truss decomposition of a graph under edge insertions was proposed in [13]. The authors mention that a similar algorithm for deletion could be derived, however, no theoretical guarantee is given for either. Here we present a new data structure under edge deletions with theoretical guarantees. In particular, our data structure maintains the trussness $t(e)$ of all edges $e \in E$ under edge deletions. To update $t(e)$, it also maintains the number of triangles of different trussness that e belongs to. Indeed, if e belongs to at least $k-2$ triangles of trussness at least k, then e belongs to a k-truss. We also analyze the construction time, the size, and the update time for our data structure.

The deletion of e breaks all triangles hinging on e, and we set $t(e)$ to zero. In addition, when the trussness of an edge decreases, the trussness of any edge that shares a triangle with that edge may decrease too. Hence, the effect of deleting a single edge can propagate through the graph.

Based on these observations, we design `Truss Update`, a data structure for efficiently maintaining the truss decomposition under edge deletions. Our data structure requires:

1. The trussness $t(G)$ of the input graph.

2. An array T_{cur} of size $|E|$ containing the trussness $t_{\text{cur}}(e)$ of each edge e of the current graph.

3. The set $\text{TRI}(e)$ of all triangles containing e, for each edge e of the current graph, implemented by `Triangles Update`.

4. A 2d array T of size $|E| \times t(G)$, whose element $T[e][i]$ corresponds to the number of triangles of trussness i in $\text{TRI}(e)$, for every edge e in the current graph and every $i \in [3, t(G)]$.

5. A stack L of edges whose trussness is to be updated.

When e is deleted, we assign $T[e][i]$ to 0, for each $i \in [3, t(G)]$. This is because e is no longer contained in any triangles. We also push e to L for its trussness to be updated and propagated. Then, while L is nonempty, we repeatedly pop an edge f from L and perform the following update operations:

O_1: Update the trussness $t_{\text{cur}}(f)$ of edge f, if f is in fewer than $t_{\text{cur}}(f)-2$ triangles of trussness $t_{\text{cur}}(f)$.

O_2: For each triangle $\{f, g, h\} \in \text{TRI}(f)$ whose trussness changes, update its entries in T and push g and h into L.

Complexity Analysis. The data structure occupies $O(|E| \cdot t(G) + \mathcal{T}_G)$ space. Constructing it takes $O(|E|^{\frac{3}{2}} + |E| \cdot t(G)) = O(|E|^{\frac{3}{2}})$ time. For the updates, observe that deleting an edge can only decrease the trussness of any other edge by at most 1. Say we delete edge e that had trussness x. Operation O_1 is implemented in $O(x)$ time for the deleted edge e, and $O(1)$ time for all subsequent edges (as their trussness can only decrease by 1). Operation O_2 takes $O(\sup_G(f))$ time for each edge f whose trussness decreases, thanks to `Triangles Update`. As we only consider each edge once on each deletion (and thus each triangle at most three times), the worst-case time for a single deletion is $O(\mathcal{T}_G)$. The advantage is the *amortized* time complexity of updating. We only add edges to L whose trussness is decreased. When we delete all edges, we decrease their trussness by at most $t(G)$ and hence update the trussness of each triangle at most $3t(G)$ times. Therefore deleting all edges takes only $O(t(G) \cdot \mathcal{T}_G)$ time. Thus, the cost of updating our data structure is significantly lower than recomputing the truss decomposition of the graph *after every deletion* in $O(|E|^{\frac{3}{2}})$ time per each deletion.

4 Exact Algorithm

An exact algorithm for MIN-k-TBS can be designed based on the following fact: a graph has a k-truss if and only if it has a minimal k-truss. As a minimal k-truss is broken by removing any of its edges (see Lemma 3.5), we observe that an edge subset of G is a feasible solution to MIN-k-TBS if and only if it intersects *all* minimal k-trusses of G; and it is an optimal solution when it is one of the smallest among such subsets.

The above observation draws a connection between MIN-k-TBS and the *hypergraph transversal problem* [17], which seeks to find a minimum transversal of a hypergraph (i.e., a smallest set of nodes of a hypergraph that intersects all its hyperedges). Specifically, let H be the hypergraph whose nodes are in one-to-one correspondence to the edges of the graph G; the hyperedges of H are all and only the minimal k-trusses of G. Clearly, an optimal solution to MIN-k-TBS is a *minimum transversal* of H.

A minimum hypergraph transversal can be found by the algorithm in [17]. However, before finding such a transversal, one needs to construct H, which in turn requires listing all minimal k-trusses of G. For this task, we devised the MTL (Minimal k-Truss Listing) algorithm, presented below.

Thus, our exact algorithm for MIN-k-TBS first executes MTL on G to construct H, and then finds a minimum transversal of H, which is an optimal solution to MIN-k-TBS.

MTL Algorithm. Listing all minimal k-trusses of G is significantly harder than computing the truss decomposition of G, since minimal k-trusses can be exponential in number. For example, every k-clique is also a minimal k-truss.

To address this task, we therefore base our MTL algorithm on the classic *binary partition* method and equip it with pruning criteria to prevent unnecessary recursive branches and save computation time. The completeness of our algorithm easily follows from the fact that the binary partition method fully explores the space of possible solutions.

As can be seen in Algorithm 1 below, MTL uses a function `List-min` that is applied recursively to extend a partial solution SOL and backtracks when no extension is possible. The set X keeps tracks of the elements that were tried already and should not be added to SOL to prevent duplication.

`List-min` halts whenever SOL *contains* a k-truss (Line 4), as SOL surely does not need further extension to be a minimal k-truss, and it outputs SOL only if SOL *is* a minimal k-truss (Line 5). Thanks to Corollary 3.4, we can also avoid the check in Line 4 entirely when the number of edges in SOL is too small to possibly create a k-truss. The check in Line 5 is performed by computing the trussness of SOL, after the removal of each single edge; a task made more efficient by `Truss Update`.

`List-min` also employs the following pruning conditions:

First, SOL is extended with edges connected to it, so that it remains connected. Indeed, a minimal k-truss is necessarily connected; otherwise its connected components would be smaller k-trusses. Furthermore, the edges that are used to extend SOL are incident to nodes of minimum degree available (Line 8). This does

Algorithm 1: MTL

```
Input  : A graph G = (V, E) and an integer k ≥ 3.
Output : All minimal k-trusses of G.
1  List-min (G, ∅, ∅, k)
2  Function List-min(G, sol, x, k)
3      if Prune (G,sol,x,k) then return
4      if sol contains a k-truss then            /* using Corollary 3.4 */
5          if sol is a minimal k-truss then output sol
6          return
7      else
8          v ← node of minimum degree in G \ x, among those incident to
             sol and with at least one edge in E \ (sol ∪ x)
9          foreach edge e = {v, w} in E \ (sol ∪ x) do
10             List-min (G, sol ∪ {e}, x, k)
11             x ← x ∪ {e}
12 Function Prune(G, sol, x, k)
13     foreach v incident to sol do
14         if |N_{G\x}(v)| < k − 1 then return true
15     foreach e ∈ sol do
16         if |TRI_{G\x}(e)| < k − 2 then return true
17         if t_{G\x}(e) < k then return true
18     return false
```

not affect the correctness of the binary partition method; it is a heuristic choice that increases the effectiveness of the Prune function described below, by generating graphs with nodes of small degree whenever these edges are added to the excluded set x.

Second, List-min uses the Prune function to detect recursive branches that surely cannot extend sol to a k-truss. Prune checks three properties, ordered from the most efficiently computable to the most powerful in terms of pruning power:

1. There is any node v incident to sol with degree $< k-1$ in $G \setminus$ x (as v cannot gain enough neighbors to be in a k-truss).

2. There is an edge $e \in$ sol contained in less than $k-2$ triangles in $G \setminus$ x, for any edge in sol.

3. There is an edge $e \in$ sol with trussness less than k in $G \setminus$ x, for any edge in sol.

If any of these properties holds in $G \setminus$ x, sol cannot be extended to a k-truss, since elements of x cannot be added to sol. In this case, Prune returns true; otherwise, it returns false. Furthermore, Prune can utilize Truss Update for finding the edge trussness in the current subgraph (Line 17), without explicitly computing the truss decomposition of the subgraph.

Modifications for MIN-k-CBS. The following trivial modifications are needed to exactly solve MIN-k-CBS: (1) Algorithm 1 is modified to produce only minimal k-trusses containing nodes in U. (2) The hypergraph H is constructed using the minimal k-trusses output by the modified Algorithm 1. (3) The algorithm in [17] is modified to output a minimum hypergraph transversal containing only edges incident to nodes in U. This is precisely an optimal solution to MIN-k-CBS.

5 Heuristic Algorithms

We describe three heuristic algorithms, which are based on the theoretical insights presented in Section 3.

5.1 Max-Truss Breaking Heuristics

Any k'-truss with $k' > k$ is also a k-truss, by definition. Thus, we need to remove edges that break every k'-truss for all $k' > k$ as well, to obtain a solution to MIN-k-TBS. On the other hand, if a k-truss is not a k'-truss for any $k' > k$, then we can remove a single edge to break it, as implied by Lemma 3.6.

Algorithm 2: MBH_S

```
Input  : A graph G = (V, E) and an integer k ≥ 3
Output : A feasible solution E' to MIN-k-TBS
1  G' ← G and Ẽ ← E
2  while t(G') ≥ k do
3      Let T be the max-truss of G'
4      Select an arbitrary edge e' in T with support t(G') − 2 in T
5      Compute the set TRI(T, e') of triangles in T containing e'
6      Select an edge e such that {e, f, g} ∈ TRI(T, e') and e has maximum
         support in T
7      Ẽ ← Ẽ \ {e} and G' ← G(V, Ẽ)
8  return E' ← E \ Ẽ
```

The max-truss T of G satisfies the condition of Lemma 3.6 with $k = t(G)$. Thus, there exists *a single edge* e whose removal breaks T. The process can be repeated until the trussness $t(G')$ of the residual graph G' falls below k, at which point we obtain a feasible solution to MIN-k-TBS. Our Max-Truss Breaking Heuristics are based on this idea. Specifically, Lemma 3.6 confirms the existence of an edge e among the edges that form triangles with an edge e' with support $t(G') - 2$ in the max-truss (see the proof of Lemma 3.6). However, Lemma 3.6 does not specify how such e may be selected. We thus explore two strategies to select an edge e for the current graph G':

1. We select as e the edge with maximum support in the max-truss T of G', breaking ties arbitrarily. This way to select e intuitively preserves the graph size (i.e., reduces the total number of deleted edges), because the removal of e breaks a large number of triangles which no longer appear in the max-truss of G' in subsequent iterations. We refer to this heuristic as MBH_S, where S stands for size preservation. Algorithm 2 implements this idea.

2. We denote by $\text{TRI}_{\geq k}(G', e)$ (respectively, $\text{TRI}_{<k}(G', e)$) the set of all triangles in G' of trussness at least k (respectively, below k) containing edge e. We select as e an edge from T with largest ratio $\frac{|\text{TRI}_{\geq k}(G', e)|}{|\text{TRI}_{<k}(G', e)|}$, breaking ties arbitrarily. This strategy deletes edges that, on one hand, are in many triangles which inevitably have to be broken, and on the other, are in few triangles which do not. The former triangles are those with trussness at least k, as their existence would imply a k-truss. The latter triangles are those with trussness below k. Note that, by preserving the latter triangles, this strategy helps maintaining the global clustering coefficient of the graph[1]. We refer to the heuristic employing this strategy as MBH_C, where C stands for cluster coefficient preservation. The MBH_C pseudocode is the same as that of MBH_S (Algorithm 2) except for Line 6 which is replaced by:

Select an edge e: $\{e, f, g\} \in \text{TRI}(T, e')$ and e has maximum $\frac{|\text{TRI}_{\geq k}(G', e)|}{|\text{TRI}_{<k}(G', e)|}$.

As there may be many edges e' with support $t(G') - 2$ in the max-truss (Line 4), a variation of MBH_S or MBH_C can select as e an edge that forms a triangle with any of those edges e', in addition to satisfying the criteria of strategy 1 or 2.

Complexity Analysis. In each iteration, MBH_S computes the trussness $t(G')$ of G', the max-truss T of G', and the support of each edge

[1]The global clustering coefficient quantifies the tendency of the nodes of a graph G to cluster together [18] and is defined as 3 times the ratio between number of triangles in G and number of all triplets (triangles and wedges) in G.

e in T. These computations take $O(|E|^{\frac{3}{2}})$ time. All triangles in T containing edge $e' = (u, v)$ are computed in time $O(|V|) = O(|E|)$. Selecting an edge e (Line 6) is then performed by traversing $\text{TRI}(T, e')$ in $O(|E|)$ time. Since MBH_S performs $r \leq |E|$ iterations, where r is the total number of removed edges, the total time is $O(|E|^{\frac{3}{2}} r)$ in the worst case. The only difference in MBH_C is in the computation of Line 6 that computes the ratio $\frac{|\text{TRI}_{\geq k}(G',e)|}{|\text{TRI}_{<k}(G',e)|}$, for each e that forms a triangle in $\text{TRI}(T, e')$. This computation takes time $O(|E|^{\frac{3}{2}})$ per iteration of the while loop. Thus, MBH_C also takes $O(|E|^{\frac{3}{2}} r)$ time in the worst case. Plugging in the `Truss Update` and `Triangles Update` data structures speeds up the computation and yields the *improved* worst-case time of $O(|E|^{\frac{3}{2}} + |E|r + t(G)\mathcal{T}_G)$ for both heuristics (see Appendix B).

Modifications for MIN-k-CBS. We modify Line 6 in Algorithm 2 to consider only edges incident to nodes in U, as required by MIN-k-CBS. If no such edge can be selected, we remove an edge that is incident to a node in U and has a maximum support in the maximal k-truss for MBH_S (or maximum ratio $\frac{|\text{TRI}_{\geq k}(G',e)|}{|\text{TRI}_{<k}(G',e)|}$ for MBH_C), among all edges that are incident to nodes in U, breaking ties arbitrarily. This guarantees that no node in U belongs to a k-truss in $G(V, E \setminus E')$.

5.2 "Save the Neighbors" Heuristic

A straightforward way to construct a feasible solution to MIN-k-TBS is to iteratively remove an edge from the max-truss of the graph G, until G has trussness below k. However, this heuristic may delete an unnecessarily large number of edges hinging on the *same* triangles, when these triangles have large trussness (i.e., their edges have trussness much larger than k).

The main idea of our Save the Neighbors Heuristic (SNH) is to reduce the number of deleted edges by limiting the unnecessary subsequent deletion of neighboring edges of an edge e that is selected for deletion. Let M be the *maximal* k-truss of the current graph G', and consider a candidate triangle $\{e, f, g\}$ hinging on e inside M. While we want a large number of triangles of M to contain e, that is a large set $\text{TRI}_{\geq k}(M, e)$, at the same time we want to limit the propagation to the triangles in M that also contain f and g, that is we want small sets $\text{TRI}_{\geq k}(M, f)$ and $\text{TRI}_{\geq k}(M, g)$. For this, we employ the utility function $\Gamma_k(M, e, f, g)$, defined as follows:

$$\left(\frac{|\text{TRI}_{\geq k}(M,e)|}{\max(|\text{TRI}_{\geq k}(M,f)| - k + 2, 1)} + \frac{|\text{TRI}_{\geq k}(M,e)|}{\max(|\text{TRI}_{\geq k}(M,g)| - k + 2, 1)}\right).$$

This idea aims at breaking the necessary amount of triangles in M by removing few of its edges. Note that M is the maximal k-truss of G' (not the max-truss as in the Max-Truss Breaking Heuristics). Algorithm 3 describes SNH.

Complexity Analysis. In each iteration, SNH computes the trussness $t(G')$ of G' and M (the maximal k-truss of G'). These computations take $O(|E|^{\frac{3}{2}})$ time. Computing all triangles (of trussness at least k) in M takes $O(|E|^{\frac{3}{2}})$ time. Based on these and the utility function $\Gamma_k(M, e, f, g)$, SNH then evaluates the formula in Line 6. The evaluation of the formula over all edges in M takes $O(|E| + \mathcal{T}_G)$ time because each triangle contains $3 = O(1)$ edges (thus it is evaluated three times in the sum of Line 6) and there are $\mathcal{T}_G$ triangles. Since SNH performs $r \leq |E|$ iterations, where r is the total number of removed edges, and $\mathcal{T}_G = O(|E|^{\frac{3}{2}})$, SNH takes $O(|E|^{\frac{3}{2}} r)$ time in the worst case. Plugging in the `Truss Update` and `Triangles Update` data structures constructed on M speeds up the computation and yields the worst-case time of $O(|E|^{\frac{3}{2}} + |E|r + t(G)\mathcal{T}_G + \mathcal{T}_G \cdot r)$. This bound is larger than the improved bound of MBH_S and MBH_C by an additive term $\mathcal{T}_G \cdot r$, because SNH considers up to $\mathcal{T}_G$ triangles in each of the r iterations it performs.

Algorithm 3: SNH

Input : A graph $G = (V, E)$ and an integer $k \geq 3$
Output: A feasible solution E' to MIN-k-TBS

1 $G' \leftarrow G$ and $\tilde{E} \leftarrow E$
2 **while** $t(G') \geq k$ **do**
3 max $\leftarrow -\infty$
4 Let M be the maximal k-truss of G'
5 **for** *each* $e \in M$ **do**
6 score $\leftarrow \sum_{\{e,f,g\} \in \text{TRI}_{\geq k}(M,e)} \Gamma_k(M, e, f, g)$
7 **if** score > max **then**
8 max $\leftarrow$ score; selected $\leftarrow e$
9 $\tilde{E} \leftarrow \tilde{E} \setminus \{\text{selected}\}$ and $G' \leftarrow G(V, \tilde{E})$
10 **return** $E' \leftarrow E \setminus \tilde{E}$

Modifications for MIN-k-CBS. The only modification is that the selected edge e must also be incident to a node in U.

6 Lower Bound on the Size of OPT

Let OPT be an optimal solution to MIN-k-TBS. Due to the exponential time complexity of our exact algorithm (Section 4), computing OPT is a heavy task even for small graphs with few hundreds of nodes. We design an algorithm for computing a lower bound on |OPT|, the size of OPT.

Our main idea is to use cliques as a "proxy" for trusses. Since a k-clique is a k-truss, we must at the very least make the input graph G free from k-cliques to solve MIN-k-TBS.

A first idea is to apply Turan's theorem [5]: a graph with n nodes and *no* clique of size k or more cannot have more than $\frac{k-2}{k-1}\frac{n^2}{2}$ edges, thus it must be "missing" at least $\binom{n}{2} - \frac{k-2}{k-1}\frac{n^2}{2}$ edges. Turan's theorem is unlikely to be useful if applied directly to G, but it will always give us a positive lower bound of edges to remove, if applied to a clique of size *at least* k.

We thus devise an algorithm, called LB (for Lower Bound), which works in three phases:

1. Computes an *edge clique partition* of G, defined below, to obtain a collection of edge-wise disjoint cliques.

2. Applies the best available lower bound on each clique.

3. Outputs a lower bound on |OPT|, by summing the bounds of the cliques. This is possible because the cliques are all edge-wise disjoint.

Although LB does not provide a tight lower bound, it provides a bound that is close to |OPT| (see Section 8) and hence serves as a good reference point for evaluating the effectiveness of our heuristics. Below we detail the phases of LB.

Computing an Edge Clique Partition. An edge clique partition (ECP hereafter) of a graph G is a collection of cliques of G such that any two cliques do not share edges (they may share a single node), and each edge of G is contained in one of the cliques. A trivial ECP is given by the set of edges of G, but to get a good lower

bound, we want an ECP with few large cliques rather than many small ones. While minimizing the number of cliques is famously NP-complete [15], the authors of [8] recently introduced a fast and flexible framework for the related *edge clique cover* problem (where cliques are allowed to overlap): one algorithm from this framework, called "pivoting" (see Table 2 in [8]), is aimed precisely at finding covers with large cliques. We take this algorithm, and adapt it to our needs by simply deleting each clique from G, as soon as it is found. Since the deleted edges in this clique cannot be placed in other cliques by the algorithm, we obtain an ECP.

Lower Bounding the Number of Edges to Remove from Each Clique. Turan's theorem, as mentioned above, immediately provides a lower bound. This is, however, far from tight, as graphs without large cliques may still have high trussness (e.g., a complete 3-partite graph has no 4-cliques but can have trussness up to $n/3 + 2$). To get a finer bound, we can employ Theorem 3.7, which implies a graph with m edges and T triangles has trussness at least $\frac{T}{m} + 2$. We combine this with known lower bounds on triangles from [11, 19] and [5, Corollary 6.1.8], for a graph with n nodes and m edges, which are synthesized below:

1. If $m \leq n^2/4$, $T \geq 0$ [5].

2. If $n^2/4 \leq m \leq n^2/3$, $T \geq \frac{9mn - 2n^2 - 2(n^3-3m)^{3/2}}{27}$ [11].

3. If $n^2/4 \leq m \leq \lfloor n^2/4 \rfloor + \lfloor n/2 \rfloor$, $T \geq (m - \lfloor n^2/4 \rfloor)\lfloor n^2/2 \rfloor$ [19].

4. If $m \geq n^2/3$, a lower bound for T is obtained by building the piece-wise linear function interpolating the points given by integer $y = 2, 3, \ldots$ in $m = (y-1)n^2/2y$, $T = (4m - n^2)m/3n$, and computing the interpolated value of T corresponding to the specific required m [5].

Given a graph with n nodes and m edges, we use the above formulas to get a lower bound on the number T of triangles (if more than one applies, we take the largest). Then, Theorem 3.7 implies the trussness of this graph is at least $\frac{T}{m} + 2$: We find the *highest* number $m_{\max}$ of edges (and the relative T) for which $\frac{T}{m_{\max}} + 2 < k$; this means that a graph of n and trussness $< k$ must have *no more than* $m_{\max}$ edges. If we have a clique with n nodes and $\binom{n}{2}$ edges, this means we must remove at least $\binom{n}{2} - m_{\max}$ edges from it.

Given a clique of size at least k from the ECP, we use as lower bound the maximum of the lower bound computed by Turan's theorem and that computed by Theorem 3.7.

Computing the Lower Bound on |OPT|. As the cliques are all edge-wise disjoint, we sum the bounds obtained in the previous phase, and output the sum as a lower bound on |OPT|.

Complexity Analysis. The time complexity of LB is dominated by the time of the "pivoting" algorithm in [8]. A straightforward analysis of the latter algorithm yields an $O(q\Delta^2|E|)$ time bound, where q, Δ and $|E|$ is the size of the largest clique, the highest degree, and the number of edges in G, respectively.

7 Related Work

The notion of k-truss [6] has been the focus of many works which aim at detecting a maximal k-truss for each k (e.g., [7]), or a k-truss containing certain nodes and/or attributes (e.g., [13, 14]). There is also a considerable amount of work on extending the notion of k-truss to capture application-specific requirements (e.g., [9, 14, 22]).

Several recent works studied how to modify the community structure of a graph based on the concept of k-core [16, 24–26] or k-truss [24, 27]. All these works consider fixed-budget problems, where the goal is to modify the maximal k-core or k-truss of a graph by adding or deleting a *fixed* number of edges or nodes, according to some criterion relevant to the maximal k-core or k-truss of the input graph. Importantly and unlike these works, we consider problems that are not specific to the maximal k-truss but rather consider *all* k-trusses (or all those containing pre-specified nodes). This task is inherently more difficult due to the hierarchical structure of k-trusses. Furthermore, we consider problems seeking to find a global-optimum solution and not a fixed-budget solution.

8 Experimental Evaluation

We experimentally evaluate our heuristics, by comparing them to our exact algorithm and the lower bound, as well as to two natural baselines, in terms of effectiveness and efficiency. We focus on the MIN-k-TBS problem. (Recall that in Section 2, we showed results for MIN-k-CBS using a real dataset.)

Experimental Datasets and Setup. We used 10 real-world datasets (see Table 1 for their characteristics). The first 5 datasets are available from http://networkrepository.com; FL from https://sites.google.com/site/yangdingqi/home/foursquare-dataset; and all other datasets from https://snap.stanford.edu/. We also used 1,000 synthetic datasets with 30 nodes and 84 edges each, generated using the Albert-Barabasi model.

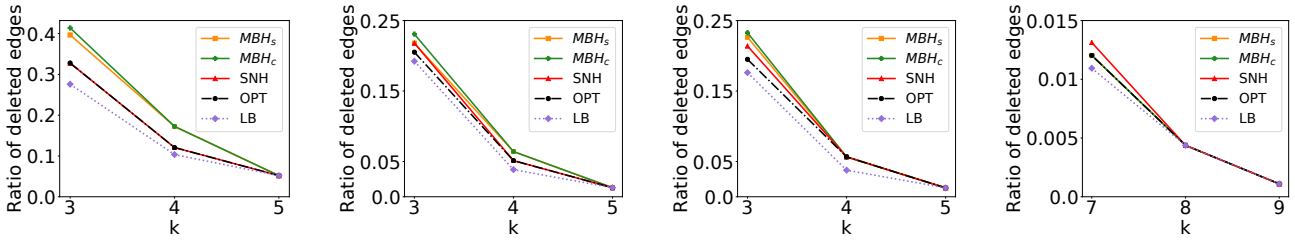
Dataset	Domain	# Edges	# Nodes	$t(G)$	Max degree	Avg degree
TRIBES	Social	58	16	5	10	7
KARATE	Social	78	34	5	17	4
DOLPHINS	Social	159	62	5	12	5
NETSCIENCE	Collab.	914	379	9	34	4
JAZZ	Collab.	2,724	198	30	100	27
WIKI	Web	100,761	8,298	23	1,065	24
EPINIONS	Social	405,739	75,888	33	3,044	10
FL	Social	607,327	114,324	30	1,755	10
DBLP	Collab.	1,871,070	511,163	115	576	11
AMAZON	E-comm.	2,439,436	410,236	11	2,760	7

Table 1: Characteristics of real datasets.

We compared our heuristics to two natural baselines:

ATk (for All Trussness $\geq k$): It removes all edges of trussness at least k. Clearly, ATk finds a feasible solution, since it suffices to remove all edges identified by ATk to solve MIN-k-TBS, but it does not consider the impact of an edge deletion on the trussness of other edges. Thus, we compared against ATk to show how many edges are "saved" by our heuristics. ATk is very fast. It takes $O(|E|^{\frac{3}{2}})$ time, as it only computes the trussness decomposition of G once.

GTk (for Greedy Trussness $\geq k$): GTk is the baseline that motivated SNH (see Section 5.2). That is, GTk iteratively removes the edge with the highest trussness, breaking ties arbitrarily, until there is no k-truss in the graph. GTk requires $O(|E|^{\frac{3}{2}}r)$ time to delete r edges, since it computes the truss decomposition after every iteration. Thus, it is expected to be much slower than ATk. However, GTk identifies substantially fewer edges to remove than ATk, because it considers the impact of removing an edge to the trussness of other edges. We compared against GTk to show the effectiveness of our heuristics and the efficiency impact of our data structures.



(a) TRIBES (b) KARATE (c) DOLPHINS (d) NETSCIENCE

Figure 2: Ratio of deleted edges, for varying k, on small graphs. Note that SNH essentially coincides with OPT (optimal solution) and that SNH is close to our LB (lower bound). Further note that LB is not far from OPT.

	MBH_S	MBH_C	SNH
min	1	1	1
mean	1.07	1.064	1.043
median	1.062	1.059	1.055
max	1.267	1.278	1.25
st. dev.	0.054	0.052	0.044

(a) $k = 3$

	MBH_S	MBH_C	SNH
min	1	1	1
mean	1.15	1.251	1.018
median	1	1	1
max	2	3	1.5
st. dev.	0.265	0.361	0.085

(b) $k = 4$

Table 2: Statistics for the ratio between the number of removed edges by a heuristic and by the exact algorithm (i.e., in an optimal solution) on 1,000 synthetic graphs.

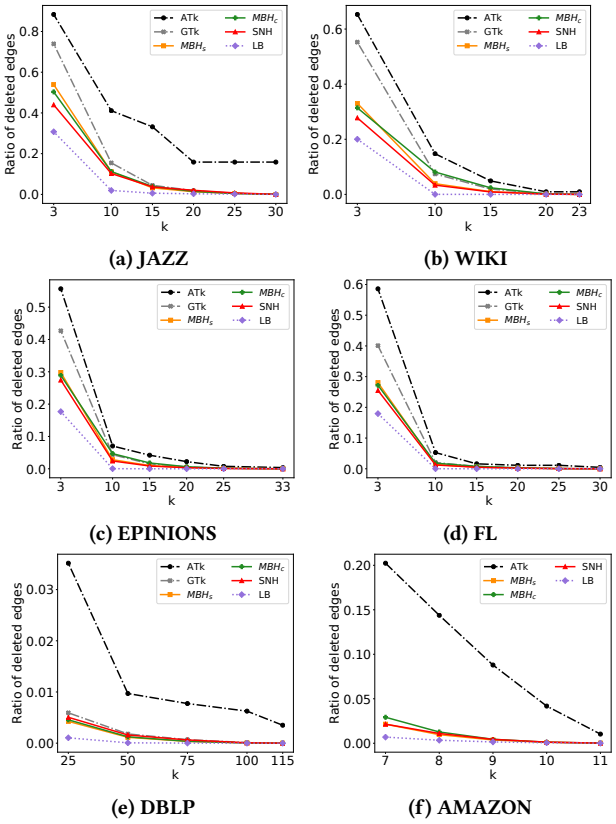
We also compared our heuristics to the exact algorithm denoted by OPT (see Section 4) and the lower bound algorithm denoted by LB (see Section 6) to rigorously assess the effectiveness/efficiency trade-offs offered by our heuristics.

To measure effectiveness, we used: (1) the ratio of deleted edges $\frac{|E'|}{|E|}$; and (2) the relative error RE $= \frac{C_G - C_{G'}}{C_G}$, where C_G (respectively, $C_{G'}$) is the global clustering co-efficient of G (respectively, G') [18].

We implemented all evaluated methods and executed them on an Intel Xeon @ 2.60GHz with 128GB RAM. We omit the results of the variations of MBH discussed in Section 5.1 as they performed similarly to MBH_S and MBH_C but were much slower. We also omit the versions of our heuristics that do not employ `Truss Update`, as they were more than one order of magnitude slower. In our implementations, we used the algorithm of [7] to compute the truss decomposition. Our code is available at https://bitbucket.org/breakingtruss/kdd2021.

Effectiveness on Small Graphs. We show that our heuristics find near-optimal solutions (close to OPT), and also that the lower bound computed by our LB algorithm is not far from OPT. This can be seen in Fig. 2 and Table 2, which show statistics for the number of deleted edges, for real and synthetic graphs, respectively. On synthetic graphs, our heuristics removed at most 7% more edges than the optimal on average (see Table 2a). On real graphs, the results are similar. SNH is the best-performing heuristic, which shows the effectiveness of its strategy for avoiding unnecessary edge deletion. MBH_S and MBH_C also performed very well, with the former being able to delete fewer edges, as it considers solely the support of edges in the max-truss. As expected by its design that considers triangles of all trussness values, MBH_C outperformed the other heuristics in terms of RE (see Appendix C).

Effectiveness on Large Graphs. We show that our heuristics are fairly close to the lower bound, which implies that they are even closer to the optimal solution. Also, our heuristics substantially outperform both baselines, particularly for small k values (see Fig. 3). Again, SNH outperformed MBH_S and MBH_C, with MBH_S being slightly better than MBH_C as before. As expected, our exact algorithm, which has an exponential time complexity, did not terminate in 24 hours in these experiments, and so its results are omitted.



(a) JAZZ (b) WIKI (c) EPINIONS (d) FL (e) DBLP (f) AMAZON

Figure 3: Ratio of deleted edges, for varying k, on large graphs. SNH is close to our LB (lower bound). Thus, SNH is even closer to OPT (optimal solution), which is not computable on large graphs. GTk results are omitted from Fig. 3f, as it did not terminate in 24 hours.

We also show that MBH_C preserves the global clustering coefficient better than the other heuristics and baselines in Table 3. These results suggest that, when performed carefully, edge deletion does not substantially affect the clustering structure of the graph. This is useful when one wants the output graph to be published for analysis (e.g., in A4 in Introduction).

Efficiency. We show that our heuristics are one to two orders of magnitude faster than GTk (Fig. 4), despite outperforming it in

k	ATk	GTk	MBH_S	MBH_C	SNH
30	8.87	0.10	**0.08**	0.09	0.22
25	8.87	1.17	1.57	**0.8**	1.73
20	8.87	4.14	4.14	**2.43**	4.56
15	27	8.55	8.12	**6.19**	9.95
10	40.01	24.02	24.67	**23.37**	28.25
5	73.44	59.74	64.12	**63.71**	66.33

(a) JAZZ

k	ATk	GTk	MBH_S	MBH_C	SNH
23	7.6	0.044	0.034	**0.002**	0.003
20	7.6	1.47	1.37	**0.66**	0.83
15	28.65	10.93	11.83	**7.44**	10.7
10	55.22	29.38	32.142	**23.66**	32.4
5	81.11	65.98	70.22	**64.88**	70.68

(b) WIKI

k	ATk	GTk	MBH_S	MBH_C	SNH
33	6.517	0.040	0.006	**0.003**	0.008
25	9.07	3.08	2.61	**1.99**	3.20
20	23.78	7.90	7.19	**6.06**	8.31
15	35.90	17.28	17.00	**14.67**	17.90
10	49.74	34.07	34.33	**30.79**	35.09
5	81.85	68.57	68.96	**64.02**	67.95

(c) EPINIONS

k	ATk	GTk	MBH_S	MBH_C	SNH
30	6.922	0.025	0.011	**0.006**	0.02
25	16.37	1.81	1.41	**1.19**	1.87
20	16.37	6.04	5.29	**4.53**	6.26
15	20.36	11.90	11.38	**10.12**	11.83
10	36.94	21.62	21.35	**19.57**	21.78
5	71.37	54.80	54.80	**51.72**	54.45

(d) FL

k	ATk	GTk	MBH_S	MBH_C	SNH
115	4.65	0.0024	0.0023	**0.0022**	0.0023
100	7.82	0.48	0.17	**0.16**	0.31
75	9.18	2.6	1.36	**1.35**	2.35
50	10.97	5.61	**3.84**	4.03	5.37
25	20.53	11.89	**10**	10.3	12.11

(e) DBLP

k	ATk	MBH_S	MBH_C	SNH
11	2.3	0.13	**0.08**	0.1
10	9.94	0.83	**0.52**	0.83
9	21.23	2.75	**1.81**	2.87
8	34.38	6.83	**4.74**	7.12
7	47	13.7	**10.23**	13.81

(f) AMAZON

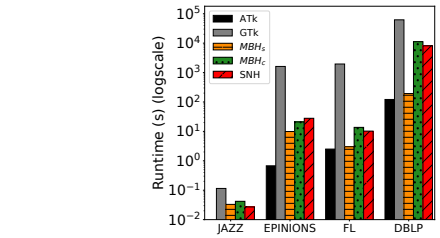
Table 3: RE% in terms of global clustering coefficient, for varying k on large graphs. The best-performing method is in bold. As expected by its design, MBH_C is the clear winner. The results for $k = 3$ are omitted, as RE = 100% for all methods by definition; GTk results are omitted from Table 3f, as it did not terminate in 24 hours.

terms of quality (Fig. 3). For example, we were unable to run GTk on AMAZON within 24 hours. The reason is that our heuristics employ `Truss Update` and `Triangles Update`, instead of the expensive truss decomposition procedure employed by GTk. MBH_S is faster than MBH_C, since MBH_C also considers triangles with trussness below k, as well as than SNH, since the Γ_k function considers all triangles (of trussness at least k) in the maximal k-truss. As expected, ATk is the fastest method, because it does not need to recompute the trussness of edges after edge removal; recall that it is by far the worst in terms of effectiveness (Fig. 3). Of note, LB took less than 10 seconds in any case, thus providing a quick assessment tool for the user, as noted in Introduction.

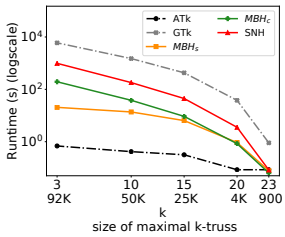
Acknowledgments H. Chen was supported by CSC scholarship. M. Sweering was supported by the Netherlands Organisation for Scientific Research (NWO) through Gravitation-grant NETWORKS-024.002.003. A. Conte and R. Grossi were partially supported by MIUR, Grant 20174LF3T8 AHeAD.

References

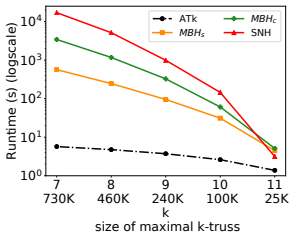
[1] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.
[2] N. R. Aravind, R. B. Sandeep, and N. Sivadasan. Dichotomy results on the hardness of H-free edge modification problems. *SIAM Journal on Discrete Mathematics*, 31(1):542–561, 2017.
[3] G. Beigi and H. Liu. A survey on privacy in social media: Identification, mitigation, and applications. *ACM/IMS Trans. Data Sci.*, 1(1), 2020.
[4] G. Bernardini, H. Chen, A. Conte, R. Grossi, G. Loukides, N. Pisanti, S. P. Pissis, G. Rosone, and M. Sweering. Combinatorial algorithms for string sanitization. *ACM Trans. Knowl. Discov. Data*, 15(1), 2020.
[5] B. Bollobás. *Extremal graph theory*. Courier Corporation, 2004.
[6] J. Cohen. Trusses: Cohesive subgraphs for social network analysis. *Nat. Secur. Agency Tech. Rep.*, 16:3–29, 2008.



(a) All datasets with a 25-truss



(b) WIKI



(c) AMAZON

Figure 4: (a) Runtime when $k = 25$, for all datasets with a 25-truss. (b,c) Runtime for varying k. GTk results are omitted from Fig. 4c, as it did not terminate in 24 hours.

[7] 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*, 8:139096–139109, 2020.
[8] A. Conte, R. Grossi, and A. Marino. Large-scale clique cover of real-world networks. *Information and Computation*, 270:104464, 2020.
[9] S. Ebadian and X. Huang. Fast algorithm for k-truss discovery on public-private graphs. In *IJCAI*, pages 2258–2264, 2019.
[10] Y. Fang, X. Huang, L. Qin, Y. Zhang, W. Zhang, R. Cheng, and X. Lin. A survey of community search over big graphs. *VLDB J.*, 29(1):353–392, 2020.
[11] D. C. Fisher. Lower bounds on the number of triangles in a graph. *Journal of graph theory*, 13(4):505–512, 1989.
[12] S. Fortunato. Community detection in graphs. *Physics Reports*, 486(3):75–174, 2010.
[13] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu. Querying k-truss community in large and dynamic graphs. In *SIGMOD*, page 1311–1322, 2014.
[14] X. Huang and L. V. S. Lakshmanan. Attribute-driven community search. *Proc. VLDB Endow.*, 10(9):949–960, 2017.
[15] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.
[16] R. Laishram, A. E. Sariyüce, T. Eliassi-Rad, A. Pinar, and S. Soundarajan. Measuring and improving the core resilience of networks. In *WWW*, page 609–618, 2018.
[17] K. Murakami and T. Uno. Efficient algorithms for dualizing large-scale hypergraphs. *Discrete Applied Mathematics*, 170:83 – 94, 2014.
[18] M. E. J. Newman, S. H. Strogatz, and D. J. Watts. Random graphs with arbitrary degree distributions and their applications. *Physical review E*, 64(2):026118, 2001.
[19] V. S. Nikiforov and N. G. Khadzhiivanov. Solution of the problem of p. erdos on the number of triangles in graphs with n vertices and [n2/4]+ l edges. *CR Acad. Bulgare Sci*, 34(969-970):2, 1981.
[20] F. Rousseau, J. Casas-Roma, and M. Vazirgiannis. Community-preserving anonymization of graphs. *Knowl. Inf. Syst.*, 54(2):315–343, 2018.
[21] Y. Saygin, V. S. Verykios, and C. Clifton. Using unknowns to prevent discovery of association rules. *SIGMOD Rec.*, 30(4):45–54, 2001.
[22] C. Wang and J. Zhu. Forbidden nodes aware community search. *AAAI*, 33(01):758–765, 2019.
[23] D. Yang, B. Qu, J. Yang, and P. Cudre-Mauroux. Revisiting user mobility and social relationships in lbsns: A hypergraph embedding approach. In *WWW*, page 2147–2157, 2019.
[24] F. Zhang, C. Li, Y. Zhang, L. Qin, and W. Zhang. Finding critical users in social communities: The collapsed core and truss problems. *IEEE Transactions on Knowledge and Data Engineering*, 32(1):78–91, 2020.
[25] Z. Zhou, F. Zhang, X. Lin, W. Zhang, and C. Chen. K-core maximization: An edge addition approach. In *IJCAI*, pages 4867–4873, 7 2019.
[26] W. Zhu, C. Chen, X. Wang, and X. Lin. K-core minimization: An edge manipulation approach. In *CIKM*, page 1667–1670, 2018.
[27] W. Zhu, M. Zhang, C. Chen, X. Wang, F. Zhang, and X. Lin. Pivotal relationship identification: The k-truss minimization problem. In *IJCAI*, pages 4874–4880, 2019.

A Omitted Proofs

Proof of Theorem 3.1. It is NP-hard to find a smallest set of edges to delete to make G triangle-free, which is exactly the MIN-3-TBS problem. Assuming the Exponential Time Hypothesis (ETH), we cannot even solve this problem in $2^{o(|E'|)} \cdot n^{O(1)}$ time [2], where $n = |V|$.

We will now prove that MIN-k-TBS is also NP-hard for $k > 3$ using a reduction from MIN-3-TBS.

Recall that $G = (V, E)$ is the graph for which we want to solve MIN-3-TBS. Let T be the set of triangles in G. We consider a new graph $G_k = (V_k, E_k)$, which is constructed as follows. For each triangle $t \in T$, let $S_t := t \cup [k-3] \times \{t\}$ denote the nodes of t and $k-3$ new nodes. The new graph consists of the union of the cliques $\binom{S_t}{2}$ over all triangles $t \in T$. Formally,

$$G_k := \left(V \cup [k-3] \times T, \bigcup_{t \in T} \{\{u, v\} \mid u, v \in S_t, u \neq v\} \right).$$

We will now show that solving MIN-3-TBS for G is equivalent to solving MIN-k-TBS for G_k.

Suppose $G' = (V, E \setminus E')$ does not contain any triangles. Then for each triangle $t \in T$, there must be an edge $\{t_1, t_2\} \subseteq t \cap E'$. Note that for all $i \in [k-3] \times \{t\}$ the edges (i, t_1) and (i, t_2) are contained in at most $k-3$ triangles in $(V_k, E_k \setminus E')$. Therefore their trussness is below k. Therefore no edge $e \in \binom{S_t}{2} \setminus \binom{t}{2}$ can be in $k-2$ triangles of trussness k. It follows that the k-trusses in $(V_k, E_k \setminus E')$ are the k-trusses in $(V_k, E \setminus E')$. However G' and hence $(V_k, E \setminus E')$ are triangle-free. Therefore $(V_k, E_k \setminus E')$ does not contain any k-trusses.

Suppose $G'_k = (V_k, E_k \setminus E')$ does not contain any k-trusses. Let $f : E_k \to E$ be any function such that

- $f(e) = e$ for all $e \in E$, and
- $f(e) \subseteq t$ for all $t \in T$ and $e \in \binom{S_t}{2} \setminus \binom{t}{2}$.

For each $t \in T$, the induced subgraph $G'_k[S_t]$ is not a k-truss. Hence it is not a k-clique and there must be an edge $e_t \in E' \cap \binom{S_t}{2}$. Since $f(e_t) \subseteq t$, the triangle t does not appear in $(V, E \setminus f(E'))$. Therefore $(V, E \setminus f(E'))$ is triangle-free.

It follows that solving MIN-3-TBS for G is equivalent to solving MIN-k-TBS for G_k. Therefore, the problem MIN-k-TBS is NP-hard.

Proof of Lemma 3.3. An edge e of the k-truss has, by definition, support at least $k-2$ in the k-truss. Thus, e is adjacent to at least $k-2$ other edges of the k-truss on each of its endpoints, and each endpoint is incident to at least $k-2$ edges plus e itself.

Proof of Corollary 3.4. By Lemma 3.3 each node in the k-truss has degree at least $k-1$. As each edge e has support at least $k-2$ in the k-truss, by definition, the k-truss contains at least k nodes (2 incident to e, and $k-2$ as the third node of each triangle). The claim is completed by the so-called "hand-shaking lemma": a graph has as many edges as the sum of degrees of its nodes divided by 2.

Proof of Lemma 3.5. Every subgraph of S is contained in at least one graph obtained by removing an edge of S. If no such graph contains a k-truss, then no subgraph of S contains a k-truss, which implies that S is a minimal k-truss.

Proof of Lemma 3.6. Let e' be an edge of minimum support in S, which must be exactly $k-2$ (or S would be a $(k+1)$-truss); and let e be one of the edges forming a triangle in S with e'. The support of e' in $S \setminus e$ is $k-3$ and the claim follows.

Proof of Theorem 3.7. The proof of Theorem 1 in [7] shows that $t_G \geq \frac{T_S}{m_S} + 2$ for any subgraph S of G with T_S triangles and m_S edges. The claim holds as G is a subgraph of itself.

B Improvements to MBH$_S$ and MBH$_C$

We plug in the `Truss Update` and `Triangles Update` data structures, which both can be constructed in $O(|E|^{3/2})$ time, into MBH$_S$ and MBH$_C$. Since `Truss Update` provides the trussness of each edge in $O(1)$ time, Lines 3 and 4 take $O(|E|)$ time. Since `Triangles Update` provides $O(1)$-time access to the list of triangles containing an edge e', Lines 5 and 6 take $O(\sup_T(e'))$ time. The update of G' is handled by the maintenance of `Truss Update` and `Triangles Update`, which amortizes to $O(t(G)\mathcal{T}_G + \mathcal{T}_G) = O(t(G)\mathcal{T}_G)$ time across *all* edge deletions. This gives the *improved* time bound of $O(|E|^{3/2} + |E|r + t(G)\mathcal{T}_G)$. For MBH$_C$, the only difference is that we consider all triangles in G' in Line 6. Since we delete a triangle after considering it, the cost amortizes to $O(\mathcal{T}_G)$. This gives the same *improved* time bound of $O(|E|^{3/2} + |E|r + t(G)\mathcal{T}_G)$.

C Additional Experimental Results

k	MBH$_S$	MBH$_C$	SNH
5	12.061	**6.811**	8.418
4	**30.541**	35.327	33.276

(a) TRIBES

k	MBH$_S$	MBH$_C$	SNH
5	6.878	6.878	**6.407**
4	25.216	**21.597**	36.861

(b) KARATE

k	MBH$_S$	MBH$_C$	SNH
5	4.364	**4.159**	5.767
4	28.759	**25.167**	28.669

(c) DOLPHINS

k	MBH$_S$	MBH$_C$	SNH
9	0.716	**0.138**	0.138
8	1.873	**1.156**	2.626
7	3.945	**3.148**	4.65

(d) NETSCIENCE

Table 4: RE% in terms of global clustering coefficient, for varying k on small graphs. The best-performing method is in bold. MBH$_C$ is the clear winner. Results for $k = 3$ are omitted, as $RE = 100\%$ for all methods by definition.