Truss Analytics

Amruth Ayaan Gulawani

February 2025

Contents

1 Introduction

Graphs are essential tools for modeling relationships among diverse entities, where vertices represent nodes, and edges denote the interactions between them. In many real-world applications, mainly social networks, communities emerge as sets of similar entities that are densely connected through specific relationships. These communities are often analyzed and understood through the concept of cohesive subgraphs; dense subgraphs that serve as fundamental building blocks for identifying and characterizing community structures.

Cohesive subgraphs, including models like cliques, quasi-cliques, k-clans, k-clubs, k-plexes, k-cores, and k-trusses, capture the intuition that community members exhibit strong internal connectivity while maintaining limited connections with external entities. The study of cohesive subgraphs is pivotal in community search, a query-driven process that differs from traditional community detection by emphasizing three desirable properties:

1. Cohesive structure for high-quality communities
2. Support for easy querying and personalization
3. Efficient algorithms for real-time query processing

The recent advent of polynomial-time computable cohesive subgraphs, such as k-cores and k-trusses, has revolutionized community search by enabling efficient retrieval and hierarchical analysis of communities based on a single parameter k. This efficiency supports interactive query exploration, allowing users to refine their searches dynamically. Applications of cohesive subgraphs extend to social and information networks, where they facilitate the identification of tightly-knit groups, enhance network analysis, and support tasks like anomaly detection and personalized recommendations. This paper introduces these concepts, focusing on the role of trusses as an advanced cohesive subgraph model, and introduces improvements.

Notation

- G is the graph under discussion. G is assumed undirected and simple.
- V is the vertex set of G.
- E is the edge set of G.
- n is the number of vertices in G.
- $|E|$ is the number of edges in G.
- $d(v)$ denotes the degree (also called valence) of arbitrary vertex v.

2 Previously Proposed Methods

2.1 Cliques

A clique is a subgraph where every vertex is adjacent to each other. Although intuitive, cliques are problematic:

- Rare but omnipresent: Small cliques (say size 3) are too numerous to be useful, while larger cliques are rare, this implies everyone in the large community is tightly knit, whereas in real scenarios the tightly knit communities are often skewed, meaning the degree of nodes are low valued.
- Computational Overhead: Enumerating maximal cliques is NP-hard. Listing all maximal cliques may require exponential time as there exist graphs with exponentially many maximal cliques.

  The Bron–Kerbosch algorithm can be used to list all maximal cliques in worst-case optimal time, and it is also possible to list them in polynomial time per clique.

2.2 n-Cliques

Cliques were generalized into n-cliques, where vertices are at most distance n apart. A 1-clique is a standard clique. However:

- Diffuseness: For $n > 2$, n-cliques become too loosely connected to reflect meaningful cohesion.
- Intractability: Similar to Cliques, Finding n-cliques remains computationally infeasible.
- Dependency: The traversal (information) may pass through non-clique vertex set and this reduces coherence of the community.

2.3 n-Clans and n-Clubs

To address n-clique dependency issue, n-clans (n $\geq$ diameter [1], within the subgraph) and n-clubs (maximal subgraphs of diameter n) were proposed, meaning it differed from n-cliques by eliminating the vertices requiring dependency on non-Clique vertex. Yet:

- Enumeration Overload: These still produce too many subgraphs, similar to Cliques
- Intractability: The computational complexity remains high, limiting scalability.

These can be alternately defined as

- **n-Clans**: An n-clan is an n-clique where the diameter of the induced subgraph $G[V_H]$ is $\leq n$.
- **n-Clubs**: An n-club is a maximal subgraph $H = (V_H, E_H)$ where the diameter of $G[V_H]$ is $\leq n$.

[1] Diameter is the largest distance between two vertices in the graph, where distance between two vertices is defined as the minimum number of that must be traversed to go from one vertex to the other

2.4 k-Plexes

The k-plex is defined as a maximal subgraph where each vertex is adjacent to all but at most k others (a 1-plex is a clique). However:

- Intractability: Like cliques, enumerating k-plexes is computationally expensive.
- Excessive Enumeration: The number of k-plexes can be impractical to fully analyze.

2.5 k-Cores

k-cores are defined as maximal subgraphs[2] where each vertex has degree at least k. K-core of a graph is unique for a chosen K. For instance, 0-core of the graph is the graph itself. While more tractable:

- Overly permissive: k-cores include weakly connected regions, reducing cohesiveness thus lacking specificity.
- Limited cohesion: They do not enforce strong internal connectivity beyond a minimum degree.

2.6 Other Methods

Additional approaches were proposed like LS sets and λ sets compare internal to external connections, but are rare in social networks and although polynomial, they are computationally costly. Clustering methods face similar drawbacks.

The birth of Trusses lies in between the tradeoffs in the form of computational strain and ubiquity provided by cliques, n-cliques, n-clans, n-clubs, and k-plexes and overly promiscuous k-cores, in the form of an alternate generalization of the clique.

[2]Maximal subgraph is a subgraph H of a graph G, and the subgraph H is connected and there is no other vertex that you could add from the vertex set of the subgraph H while also maintaining that connectivity.

3 Trusses

3.1 Introduction to k-Truss

A k-truss is a subgraph where every edge is part of at least k-2 triangles. This ensures the subgraph is densely connected—higher k means tighter cohesion. Alternatively, defined as a non-trivial, single-component subgraph where each edge is part of at least $k-2$ triangles.

To illustrate the structure of a k-truss, consider the following example of a 4-truss, where each edge is part of at least 2 triangles (since $k-2=4-2=2$):

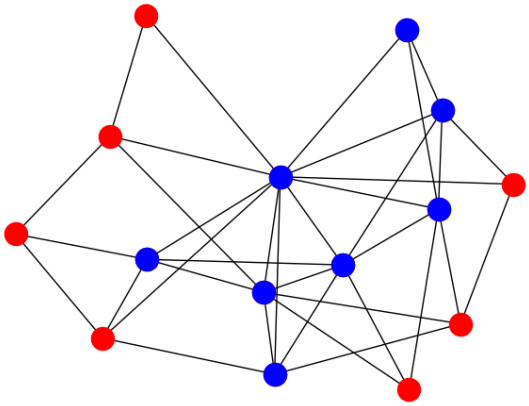

Figure 1: An example of a 4-truss subgraph, where each edge participates in at least two triangle.

3.2 Relaxed Computation

The k-truss can be computed in polynomial time, $O(n|E|)$ for a naive algorithm, can be improved to $O(\sum d^3(v)))$, unlike the intractable clique-based methods. The proof to polynomial convergence is provided by [1]. An interesting observation to be made is to enumerate cliques, the process can be now carried forward by finding the trusses then identifying cliques within the trusses.

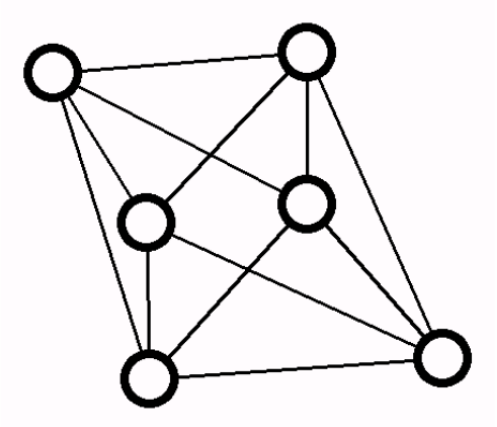

Figure 2: An example of a 4-truss subgraph, but not neccessarily containing a 4-clique

3.3 Balanced Cohesion

Trusses strike a balance between the overly strict clique and the lax k-core. A k-truss is a subgraph of a $(k-1)$-core but enforces tighter connectivity via triangles, reflecting real-world social cohesion.

3.4 Reduced Enumeration

Unlike cliques or k-plexes, which enumerate numerous overlapping subgraphs, trusses consolidate interlocking groups into single entities, simplifying analysis.

Each k-truss has edge connectivity $\geq k-1$, ensuring robustness, and for $k > 2$, a k-truss is a subgraph of a $(k-1)$-truss, enabling hierarchical computation.

3.5 Some Properties of k-truss

1. For $k > 2$, each k-truss of G is the subgraph of a (k–1)-truss of G
2. Each k-truss of G is a subgraph of a (k–1)-core of G
3. Every vertex v in a k-truss has $d(v) \geq k$–1
4. Each k-truss of G has an edge connectivity[3], $k_e \geq k-1$
5. A cutpoint[4] of a k-truss joins k-trusses.
6. If a k-truss contains a cutpoint v, it has $d(v) \geq 2(k-1)$
7. Suppose that a k-truss contains m vertices and has diameter d. Then,
   $m \geq \begin{cases} \frac{d+1}{2}k, & d \text{ is odd} \\ \frac{d}{2}k+1, & d \text{ is even} \end{cases}$

[3]The edge connectivity of a graph is the minimum number of edges whose removal will divide the graph into separate components.

[4]A cutpoint is a vertex whose removal will break the graph into components.

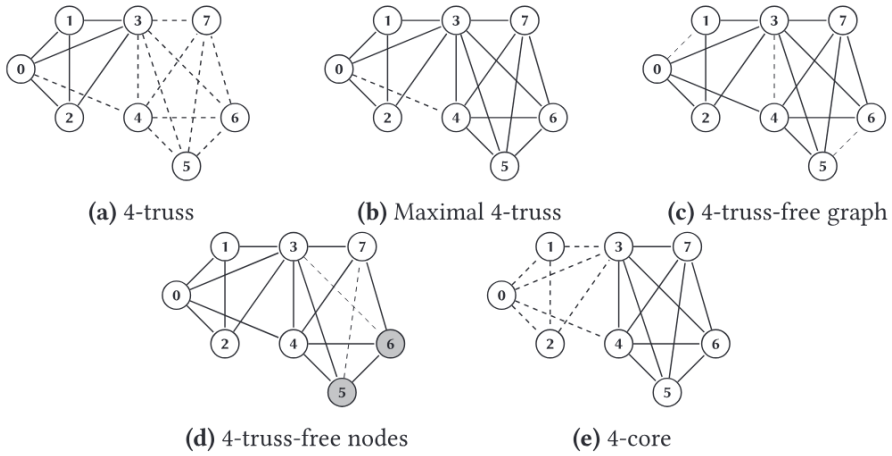

Figure 3: (a) The subgraph induced by the solid edges is a 4-truss, because every edge of the subgraph is contained in at least $4 - 2 = 2$ triangles of the subgraph. (b) The subgraph induced by all edges except the (dashed) edge (0, 4) is the maximal 4-truss of the graph. (c) The graph obtained after deleting the set (0, 1), (3, 4), (5, 6) of (dashed) edges contains no 4-truss. (d) The graph obtained after deleting 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. (e) The subgraph induced by the solid edges is a 4-core.

4 Truss Analytics

Truss analytics addresses three fundamental questions in community detection:

1. k-**Truss Problem**: Who are the members of the group that meet a specific k-requirement?

2. **Max** k-**Truss Problem**: Who are the members in the maximal group that have the closest connection?

3. **Truss Decomposition Problem**: What are the different groups that satisfy varying degrees of relationships?

The identification of k-trusses relies heavily on triangle counting as a fundamental mechanism. This section reviews existing algorithms for truss analytics, starting with modern parallel approaches for the k-Truss Problem, followed by a foundational method, and culminating in an optimized algorithm for the Max k-Truss Problem that leverages binary search to efficiently find the largest k for which the k-truss is non-empty.

4.1 Naive k-Truss Parallel Algorithm

Algorithm 1 Naive k-Truss Parallel Algorithm

procedure NAIVEKTRUSS(G, k)
    $G = (E, V)$ is the input graph with edge set E and vertex set V. k is the given k-Truss value.
    `EdgeDel[ ]` $\leftarrow -1$ ▷ initialize all edges as not deleted
    while there is any edge can be deleted **do do**
        `sup[ ]` $\leftarrow 0$ ▷ initialize the triangle counting array
        for all undeleted edge $e = (u, v) \in E$ **in parallel do** ▷ **CALCULATE # TRIANGLES IN PARALLEL**
            `sup[`e`]` $\leftarrow$ `sup(`e`,` G`)` ▷ calculate sup(e, G) using list intersection or minimum search method
        end for
        for all $e = (u, v) \in E$ **and** e is local **do do** ▷ **REMOVE EDGES THAT CANNOT SUPPORT** $k - 2$ **TRIANGLES**
            if `EdgeDel[`e`]` $== -1$ **and** (`sup[`e`]` $< k - 2$) **then**
                `EdgeDel[`e`]` $\leftarrow k - 1$
            end if
        end for
    end while
    end
    return `EdgeDel`
end procedure

4.2 Optimized k-Truss Parallel Algorithm

The optimized version improves efficiency by reducing redundant triangle counting, focusing only on edges affected by removals. It introduces a set `SetDel` to track edges to be deleted and updates supports incrementally.

Algorithm 2 Optimized k-Truss Parallel Algorithm

procedure OPTKTRUSS(G, k)
    $G = (E, V)$ is the input graph with edge set E and vertex set V. k is the given k-Truss value.
    `EdgeDel[ ]` $\leftarrow -1$ ▷ initialize all edges as not deleted
    `SetDel` $\leftarrow \emptyset$ ▷ initialize the support array of each edge
    for all edge $e \in E$ **in parallel do** ▷ **CALCULATE TRIANGLES IN PARALLEL**
        `sup[`e`]` $\leftarrow$ `sup(`e`,` G`)` ▷ using minimum search method
    end for
    for all edge $e \in E$ **and** e is local **do do** ▷ **REMOVE EDGES THAT CANNOT SUPPORT** $k-2$ **TRIANGLES**
        if `EdgeDel[`e`]` $== -1$ **and** (`sup[`e`]` $< k-2$) **then**
            `EdgeDel[`e`]` $\leftarrow k-1$
            Add e to `SetDel`
        end if
    end for
    while `SetDel` is not empty **do do** ▷ **LOOP (search affected edges)**
        for all $e_i \in$ `SetDel` **and** e_i is local **do do** ▷ **SEARCH AFFECTED EDGES AND UPDATE TRIANGLE SUPPORTS**
            using minimum search method to find e_j and e that can form a triangle with e_i
            reduce the support of e_j and e if they are undeleted
        end for
        for all edge $e \in$ `SetDel` **and** e is local **do do** ▷ **REMOVE EDGES THAT CANNOT SUPPORT** $k-2$ **TRIANGLES**
            if `EdgeDel[`e`]` $== -1$ **and** (`sup[`e`]` $< k-2$) **then**
                `EdgeDel[`e`]` $\leftarrow k-1$
                Add e to `SetDel`
            end if
        end for
        `SetDel.Clear()` ▷ switch the values of the two sets
    end while
    return `EdgeDel`
end procedure

4.3 Maximal Truss Computation

This subsection presents algorithms for computing the maximal k-truss by reducing the graph through core decomposition and edge removal based on triangle support constraints.

The maximal k-truss of G can be found simply by removing from G those elements that are not part of the maximal k-truss.

Algorithm 3 Reduce Graph to k-Truss

```
1: procedure REDUCEGRAPHTOKTRUSS(G, k)   ▷ remove obvious vertices by taking to k − 1 core
2:     REDUCETOKCORE(G, k − 1); ▷ remove edges not supported by k − 2 edge pairs
3:     REMOVEUNSUPPORTEDEDGES(G, k − 2);   ▷ iteratively remove vertices of insufficient degree
4:     REMOVEISOLATEDVERTICES(G);
5: end procedure
```

Reducing G to its maximal j-core. Vertices are removed if their degree is less than j. As vertices are removed, their neighbors lose degree and must be tested as well.

Algorithm 4 Reduce to j-Core

```
1: procedure REDUCETOKCORE(G, j)                    ▷ initialize degree counter
2:     for all v ∈ V(G) do
3:         D(v) ← 0;
4:     end for
5:     for all (a, b) ∈ G do
6:         D(a) ← D(a) + 1; D(b) ← D(b) + 1;
7:     end for                                      ▷ initialize vertex queue
8:     q ← ∅;
9:     for all v ∈ V(G) do
10:        if D(v) < j then
11:            q ← q ∪ {v};
12:        end if
13:    end for                  ▷ iteratively remove vertices of insufficient degree
14:    while q ≠ ∅ do do
15:        pull v from q;
16:        for all a ∈ N(v) do
17:            D(a) ← D(a) − 1;
18:            if D(a) < j then
19:                q ← q ∪ {a};
20:            end if
21:        end for
22:        remove v from G;
23:    end while
24: end procedure
```

Removes from G any edge that is not supported by at least j edge pairs, that is, any edge that does not complete at least j triangles. Initially, all edges are examined; as edges

are removed, neighboring edges need to be reconsidered. Those edges to be examined are held in a set that is managed in such a way that attempted duplicate entries are rejected.

Algorithm 5 Remove Unsupported Edges

```
1: procedure removeUnsupportedEdges(G, j)      ▷ this will hold edges to remove
2:     q ← ∅; ▷ count triangles C supporting each edge ▷ if count too small, queue edge for removal
3:     for all e = (a, b) ∈ E do do
4:         put members of N(a) in a hash table T;
5:         c ← 0;
6:         for all v ∈ N(b), if v ∈ T then c ← c + 1 do
7:             if c < j then
8:                 q ← q ∪ e; remove e from G;
9:             else
10:                C(e) ← c;
11:            end if
12:        end for
13:    end for        ▷ remove queued edges, perhaps queueing neighboring ones as well
14:    while q ≠ ∅ do do
15:        pull e = (a, b) from q;
16:        put members of N(a) in a hash table T;
17:        I ← ∅;
18:        for all v ∈ N(b), if v ∈ T then I ← I ∪ {v} do
19:            for all e′ joining a or b to an element of I, do do
20:                decrement C(e′);
21:                if C(e′) < j then
22:                    q ← q ∪ e′; remove e′ from G;
23:                end if
24:            end for
25:
26:
```

4.4 Maximal k-Truss Computation with Binary Search

This subsection introduces a parallel algorithm to compute the maximal k-truss using a modified binary search approach to efficiently determine the largest k for which the k-truss is non-empty. The algorithm relies on the Optimized k-Truss Parallel Algorithm (Algorithm 2) as the `KTRUSS` function.

Algorithm 6 Max k-Truss Parallel Algorithm

procedure MaxKTruss(G)
    $G = (E, V)$ is the input graph with edges set E and vertices set V
    Let $k_{low} \leftarrow 3$ and set k_{up} based on the proposed analysis method
    return DownwardSearch(G, k_{low}, k_{up})
end procedure
function DownwardSearch(G, k_{low}, k_{up})
    EdgeDel $\leftarrow k$Truss(G, k_{low})
    if (All edges have been deleted) **then**
        return ($k_{low} - 1$, EdgeDel)
    else
        EdgeDel $\leftarrow k$Truss(G, k_{up})
        if (there are undeleted edges in EdgeDel) **then**
            return (k_{up}, EdgeDel)
        else
            $k_{mid} \leftarrow \frac{k_{low}+k_{up}}{2}$
            EdgeDel $\leftarrow k$Truss(G, k_{mid})
            while (All edges have been deleted in EdgeDel) **do**
                $k_{up} \leftarrow k_{mid} - 1$
                $k_{mid} \leftarrow \frac{k_{low}+k_{up}}{2}$
                EdgeDel $\leftarrow k$Truss(G, k_{mid})
            end while
            if ($k_{mid} == k_{up} - 1$) **then**
                return (k_{mid}, EdgeDel)
            else
                $k_{low} \leftarrow k_{mid} + 1$
                return DownwardSearch(G, k_{low}, k_{up})
            end if
        end if
    end if
end function=0

The above algorithm computes a k-truss for a given k, which could be any k, not necessarily the maximal one. To find the maximal k-truss, you would need to run it multiple times with increasing k values until the result is empty, then take the largest k for which the k-truss was nonempty.

5 Case Study

5.1 Understanding k-Truss

Comprehending the core ideas behind k-truss is challenging and it's use-cases can be deceptive. Let us consider a real-life scenario, a group of six candidates, Allison, Bert, Cain, Demetrius, Ephram, and Firoz. Consider the list of relations between each of these friends and the respective graph obtained,

1. Allison [0]: Bert[1], Cain[2], Demetrius[3]
2. Bert [1]: Allison[0], Cain[2], Demetrius[3], Ephram[4]
3. Cain [2]: Allison[0], Bert[1], Demetrius[3], Firoz[5]
4. Demetrius [3]: Allison[0], Bert[1], Cain[2], Ephram[4], Firoz[5]
5. Ephram [4]: Bert[1], Demetrius[3], Firoz[5]
6. Firoz [5]: Cain[2], Demetrius[3], Ephram[4]

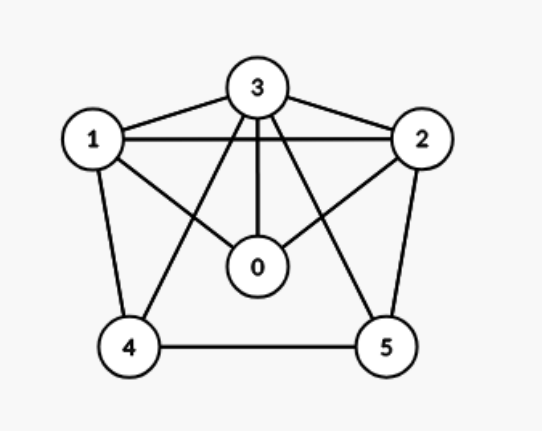

Figure 4: An undirected graph representation of the relationships

To fully understand now the k-truss parallel algorithm and the maximal k-truss algorithm, we shall dissect the relations between each person and deduce some insightful information regarding the candidates.

5.2 Calculating Edge Supports

For the k-truss, we begin by counting the edge's support i.e. we count the number of triangles each edge reinforces,

1. Between Allison and Bert, [0]-[1]: 0-1-2, 0-1-3 $\implies$ Support = 2
2. Between Allison and Cain, [0]-[2]: 0-2-3, 0-1-2 $\implies$ Support = 2
3. Between Allison and Demetrius, [0]-[3]: 0-1-3, 0-2-3 $\implies$ Support = 2
4. Between Bert and Cain, [1]-[2], 0-1-2, 1-2-3 $\implies$ Support = 2

5. Between Bert and Demetrius, [1]-[3], 0-1-3, 1-3-4, 1-2-3 $\implies$ Support = 3
6. Between Bert and Ephram, [1]-[4], 1-3-4 $\implies$ Support = 1
7. Between Cain and Demetrius, [2]-[3], 0-2-3, 2-3-5 , 1-2-3 $\implies$ Support = 3
8. Between Cain and Firoz, [2]-[5], 2-3-5 $\implies$ Support = 1
9. Between Demetrius and Ephram, [3]-[4], 3-4-5, 1-3-4 $\implies$ Support = 2
10. Between Ephram and Firoz, [4]-[5], 3-4-5 $\implies$ Support = 1
11. Between Firoz and Demetrius, [5]-[3], 3-4-5, 2-3-5 $\implies$ Support = 2

5.3 Calculating Trusses

Now we calculate the trusses for different k-values, 2-truss stands as a trivial case ;

1. Now for 3-truss (Support ($\geq$ 3-2) $\geq$ 1): From all the 11 edges, the edges whose support is $<$ 1 is eliminated and the rest form 3-truss $\implies$ All 11 edges satisfy this and all the candidates from a 3-truss.
2. Similarly for 4-truss (Support ($\geq$ 4-2) $\geq$ 2): From all the 11 edges, the edges whose support is $<$ 2 is eliminated and the rest form 4-truss $\implies$ Except Bert-Ephram, Cain-Firoz and Ephram-Firoz, all other candidate's relation form a 4-truss.
3. Similarly for 5-truss (Support ($\geq$ 5-2) $\geq$ 3): From all the 11 edges, the edges whose support is $<$ 3 is eliminated and the rest form 5-truss $\implies$ Only Bert-Demetrius and Cain-Demetrius form 5-truss
4. Similarly for 6-truss (Support ($\geq$ 6-2) $\geq$ 4): From all the 11 edges, the edges whose support is $<$ 4 is eliminated and the rest form 6-truss $\implies$ No pair of candidates form a 6-truss.

5.4 Physical Interpretation for k-truss

What insights do we draw from the above analysis?

1. 3-truss: Everyone's friendship is reinforced by at least one mutual friend. This is a baseline—it shows the group is connected, but not necessarily super tight.
2. 4-truss: The 4-truss highlights a core group (A,B,C,D,E) where friendships are stronger—each pair shares at least two mutual friends. Firoz is peripheral; his ties aren't as reinforced.
3. 5-truss: The 5-truss pinpoints the tightest clique: Bert, Cain, and Demetrius. Each friendship here is part of multiple trios.

In our hypothetical situation, all 6 candidates form a dense community with 11 friendships and 7 triangles ($\sum sup/3 = 21/3 = 7$), which is a pretty cohesive small group. The higher truss values drive the network's density as they are backed by multiple mutual friends. The highest truss value stands as the tightest group and their removal weakens the entire structure significantly.

Trusses rank people by how embedded they are. Demetrius is a linchpin; Firoz is more detachable.

5.5 Physical Interpretation for maximal k-truss

6 Discovering Cohesive Communities with Relationship Strength

Uncovering dense and cohesive communities in large networks is essential for understanding complex systems, especially when the strength of relationships between entities is a key factor. This exploration aims at identifying top-weighted communities that demonstrate strong interconnectedness, using a model where each connection is part of a specified number of triangular relationships to guarantee robust structural cohesion. Unlike conventional methods that often focus solely on connection patterns, this approach incorporates the strength of relationships, expressed as weights on connections, into the search process. This more accurately reflects real-world networks, such as social interactions where frequent participation signals closeness, collaborative ties where joint work denotes strength, or economic links where transaction volume indicates partnership depth.

To tackle this efficiently, three distinct algorithms are introduced: the Local k-Truss Algorithm (LKA), the Degree-Based Local k-Truss Algorithm (DBLKA), and the Multiple Candidate Local k-Truss Algorithm (MCLKA). Each employs a localized search strategy that examines small network segments rather than the entire structure. LKA processes connections with the highest strength one by one, evaluating whether their immediate surroundings form a cohesive group based on the triangle-based cohesion model. DBLKA builds on this by adding an initial filter, ensuring that only areas with a baseline level of connectivity are further assessed, reducing unnecessary effort. MCLKA further improves the optimization by simultaneously evaluating multiple promising areas, significantly speeding up the discovery of cohesive communities.

These algorithms aim to identify the strongest, most interconnected groups, ranked by the strength of their weakest connection. The process starts by sorting all connections by strength in descending order and gradually building a smaller network subset from the strongest ones. As each connection is added, its local neighborhood is checked for cohesion. Once enough local cohesive groups are detected, the subset is thoroughly analyzed to confirm the top global communities. This localized method stands in stark contrast to traditional approaches that process the entire network, often becoming impractical for massive datasets.

6.1 Mechanics of the Algorithms

The workings of these algorithms are both intuitive and effective. LKA constructs a temporary structure around each strong connection, verifying cohesion by counting triangles and ensuring connectivity. DBLKA improves efficiency with a preliminary check, confirming a minimum connectivity level before proceeding to the triangle-based evaluation, taking advantage of the fact that cohesive groups must meet simpler connectivity criteria. MCLKA groups several strong connections together, assessing their collective cohesion in a single pass, minimizing repetitive tasks, and proving particularly powerful for large networks or high-cohesion demands.

To assess their effectiveness, these algorithms were compared to existing techniques using diverse real-world datasets, ranging from small networks with thousands of entities to expansive ones with millions of connections. These data sets represent systems such as social platforms, communication records, and collaborative environments. Results show that LKA, DBLKA, and especially MCLKA outperform others in speed—often by

orders of magnitude—while precisely identifying the strongest, most cohesive communities. MCLKA stands out when tasked with finding many communities or those with exceptional cohesion, thanks to its streamlined batch verification.

6.2 Performance and Future Directions

Traditional methods struggle with large networks, bogged down by excessive memory demands or poor scalability. In contrast, LKA, DBLKA, and MCLKA adapt fluidly to increasing network size and complexity, maintaining efficiency as the number of desired communities or cohesion requirements grows. Their strength lies in concentrating on the most relevant network portions, guided by relationship strength, rather than processing extraneous data.

This work highlights the value of integrating relationship strength into community discovery and the advantage of localized searches over exhaustive analyses. Potential improvements include adjusting the number of local areas examined to match the true number of communities more closely, further enhancing speed. Future possibilities include incorporating entity traits to find uniform, cohesive groups or extending the algorithms to handle multiple types of connection strengths for richer insights into complex systems.

Ultimately, this exploration advances the discovery of meaningful communities in vast networks, balancing structural integrity with relationship significance. LKA, DBLKA, and MCLKA offer practical, scalable solutions that surpass traditional methods, making them highly relevant for domains where both connection volume and quality matter, such as social dynamics, teamwork studies, or economic modeling.

Input: $Graph\ G(V, E, W), k, and\ r$
Output: r communities
1: assume $Graph\ G(V, E, W)$ is presorted w.r.t. edge weights in descending order;
2: **while** ($\exists e_{(u,v)} \in E_G$ and $num\_local\_Communities$<r) **do**
3: add $e_{(u,v)}$ into Y;
4: **if** ($e_{(u,v)}$ is not pruned due to property 0.1) **then**
5: $X_{e(u,v)} \leftarrow Build\_Temp\_Graph(X_{e(u,v)})$;
6: **end if**
7: $H \leftarrow TrussDecomposition(X_{(u,v)}, k)$; ▷ make k-truss of X as in Algorithm 1
8: **if** (H is not empty) **then**
9: $num\_local\_communities+ = 1$;
10: **end if**
11: **end while**
12: $Enumerating\_Global\_Communities(Y)$

Figure 5: LOCAL k-TRUSS ALGORITHM (LKA)

Input: $Graph\ G(V, E, W)$, k, and r.
Output: *top-r weighted k-truss communities.*
1: assume $GraphG(V, E, W)$ is presorted w.r.t. edge weights in descending order;
2: **while** ($\exists e_{(u,v)} \in E_G$ and $num\_local\_Communities$<r) **do**
3: add $e_{(u,v)}$ into Y;
4: **if** ($e_{(u,v)}$is not pruned due to property 0.1) **then**
5: $X_{\text{e(u,v)}} \leftarrow Build\_Temp\_Graph(X_{e(u,v)})$;
6: **end if**
7: **if** ($X_{\text{e(u,v)}}$) is $(k-1)-core$) **then**
8: $num\_vertices \leftarrow count\_vertices \in X$;
9: **if** ($num\_vertices$==k) **then**
10: $num\_local\_communities+ = 1$
11: **end if**
12: **if** ($num\_vertices > k$) **then**
13: $H \leftarrow TRussDecomposition(X_{(u,v)}, k)$; ▷ make k-truss of X as in Algorithm 1
14: **if** (H is not empty) **then**
15: $num\_local\_communities+ = 1$;
16: **end if**
17: **end if**
18: **end if**
19: **end while**
20: $Enumerating\_Global\_Communities(Y)$

Figure 6: Degree Based LOCAL k-TRUSS ALGORITHM (DBLKA)

Input: $Graph\ G(V, E, W)$, k, and r.
Output: *top-r weighted k-truss communities.*
Input: $Graph\ G(V, E, W)$, k, and r.
Output: *top-r weighted k-truss communities.*
1: assume $GraphG(V, E, W)$ is presorted w.r.t. edge weights in descending order;
2: **while** ($\exists e_{(u,v)} \in E_G$ and $num\_local\_Communities < r$) **do**
3: add $e_{(u,v)}$ into Y;
4: **if** ($e_{(u,v)}$ is not pruned due to property 0.1) **then**
5: $List\_key-edges \leftarrow e_{(u,v)}$;
6: add w into affected_vertices : $\forall w \in \{nb(u) \cap nb(v)\}$ and $deg(w) \geq k-1$;
7: **end if**
8: **if** ($len(List\_key-edges) < r$) **then**
9: go to step3;
10: **end if**
11: **if** ($len(List\_key-edges) == r$) **then**
12: $Z \leftarrow Build\_Temp\_Graph(affected\_vertices)$
13: **end if**
14: $H \leftarrow TrussDecomposition(Z, k)$; ▷ make k-truss of X as in Algorithm 1
15: **if** (H is not empty) **then**
16: **while** (**do**$\exists e \in List\_key-edges$ and $e \in H$)
17: $num\_local\_Communities$ += 1;
18: **end while**
19: empty $List\_key-edges$
20: **end if**
21: **end while**
22: $Enumerating\_Global\_Communities(Y)$

Figure 7: Multiple candidates local k-truss algorithm(MCLKA)

7 Enhanced Cohesiveness for Public and Private Truss Communities

The k-truss model excels in identifying cohesive subgraphs with polynomial-time efficiency, but its requirement of at least $k-2$ triangles per edge can misrepresent practical community structures in diverse network contexts. A query-driven approach refines this by introducing a new cohesiveness metric that improves the applicability of the model in both public and private graph settings, complementing the developments in weighted k-truss methodologies.

The problem involves finding a connected subgraph H that contains a query vertex q on a graph $G = (V, E)$, satisfying k-truss properties while maximizing cohesiveness. This is achieved through the average trussness per vertex, defined as

$$T(H) = \frac{\sum_{e \in E(H)} t(e)}{|V(H)|},$$

where $t(e)$ represents the number of triangles containing the edge e, and $|V(H)|$ is the number of vertices in H. This vertex-centric measure, which requires $T(H) \geq k-2$, ensures balanced internal connectivity compared to the edge-centric k-truss, avoiding the inclusion of loosely connected regions or the exclusion of smaller cohesive groups.

An algorithm improves the standard calculation of k trusses with a two-phase process: (1) building an initial k truss containing q by identifying edges with at least $k-2$ triangles, and (2) refining it by pruning vertices with a low contribution of trussness, calculated as

$$t(v, H) = \frac{\sum_{e \in E(v,H)} t(e)}{deg_H(v)},$$

where $E(v, H)$ are edges incident to v in H, and $deg_H(v)$ is the degree of v in H. Optimization techniques, such as pruning based on upper-bound trussness estimates and early termination when connectivity and cohesiveness are met, reduce computational overhead while ensuring H remains a connected, cohesive community around q.

This approach distinguishes between public and private graph contexts. In public graphs, like open social or collaboration networks, it identifies widely accessible cohesive communities, such as groups of interconnected users or collaborators around a queried node. In private graph settings, such as restricted subgraphs or hidden cliques within larger networks, it uncovers tightly knit communities with limited external visibility, despite the graph being analyzable as a whole. This flexibility makes the method suitable for diverse applications, from public team formation to private group detection.

Experimental results on networks ranging from approximately 317,000 edges to 1.8 billion edges demonstrate significant improvements. The algorithm achieves runtimes up to 30% faster than baseline k-truss methods, reducing processing time from around 10 seconds to 7 seconds for $k = 10$ on a mid-sized network. It also produces communities with higher cohesiveness, with $T(H)$ values reaching approximately 8.5 compared to 6.2 for baselines on a large social network. Scalability is evident in handling billion-edge graphs in roughly 100 seconds for $k = 5$, compared to over 150 seconds for traditional approaches. These outcomes highlight its superiority over k-cores, which are overly permissive, and edge-connectivity models, which incur higher computational costs, reinforcing the k-truss framework's role in social network analysis for applications like recommendation systems.

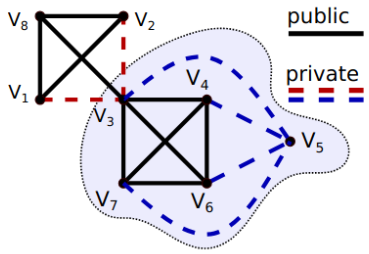

Figure 8: An example of public-private graph. Black solid edges are public. Blue dashed edges are private to v_5, and red dashed edges are private to v_3. The gray area is a 5-truss in personalized pp-graph of v_5 as g_{v_5}

8 Conclusion

This paper has demonstrated the transformative potential of the k-truss model in addressing the challenges of community detection within large-scale graphs, offering a computationally efficient and cohesive alternative to traditional methods such as cliques and k-cores. Through the development of innovative algorithms—including the Naive and Optimized k-Truss Parallel Algorithms, the binary search-based Max k -Truss Algorithm and the localized LKA, DBLKA, and MCLKA approaches—the study provides scalable solutions that balance structural integrity with relationship significance. The introduction of a refined cohesiveness metric for public and private graph contexts further enhances the model's applicability, as evidenced by significant runtime and cohesiveness improvements across diverse datasets. Future research may explore integrating entity attributes or multidimensional connection strengths, promising even richer insights into complex network systems. Ultimately, this work lays a robust foundation for advancing truss-based analytics in domains such as social network analysis, anomaly detection, and personalized recommendations.

References

[1] Jonathan Cohen; Trusses: Cohesive subgraphs for social network analysis. National Security Agency, Technical Report 16.3.1 (2008)