Querying K-Truss Community in Large and Dynamic Graphs

Project Report

Dhruval Patel, Kushal Khatri, Fatma Noor, Shraddha Sharma

PROBLEM

Community detection is used a lot in real-world applications such as social networks, biological networks, etc. One of the different but related problems is online community search which finds communities containing a query vertex in an online manner. In other words, community search provides personalized community detection for a query vertex. For the purpose of this project we will study the modeling and querying the communities of a query vertex and implement a novel solution for it. We suggest a brand-new community model built on the k-truss idea, which has great computational and structural qualities. We create an efficient search of k-truss communities with a linear cost with respect to community size using a small, beautiful index structure. Additionally, we look into the k-truss community search problem in a dynamic graph environment where graph vertices and edges are frequently added and removed. The usefulness and efficiency of our community architecture and search algorithms are demonstrated through extensive trials on huge real-world networks.

PROJECT GOAL AND MOTIVATION

A recent study of related work has proposed a novel approach for online overlapping community search(the α-adjacency-γ-quasi-k-clique model). However, there are some limitations on those models and the authors of this paper have meticulously worked on those limitations and came up with this novel approach of community search model based on the k-truss concept. Given a graph G, the k-truss of G is the largest subgraph in which every edge is contained in at least $(k - 2)$ triangles within the subgraph. The k-truss is a type of cohesive subgraph defined based on triangles which models the stable relationship among three nodes. However, the k-truss subgraph may be disconnected. On top of the k-truss, they impose an edge connectivity constraint, that is, any two edges in a community either belong to the same triangle, or are reachable from each other through a series of adjacent triangles. Here two triangles are defined as adjacent if they share a common edge. The edge connectivity requirement ensures that a discovered community is connected and cohesive. This defines the novel k-truss community model.

Simply looking up the k-truss community using its definition could result in a lot of unnecessary edge accesses. Therefore, creating a successful index is essential for optimal processing of k-truss community queries. To do this, we first run a graph G through an effective truss decomposition algorithm that computes the k-truss subgraphs for all k values. Then, to index the pre-computed k-truss subgraphs, we create the unique and refined TCP-Index structure. The TCP-Index facilitates the query of the k-truss community in linear time with respect to the community size, which is ideal. It keeps the trussness value and the triangle adjacency relationship in a compact tree-shape index.

SURVEY

We compared five different research papers that are related to our project.

- Our first paper was "Bounds and algorithms for K-Truss" which performed k-truss by using truss decomposition algorithm. Then they put some lower and upper bounds to get a critical analysis of the graph G. They have used data structures like arrays and queues. After that they perform fast matrix multiplication to represent the data. These algorithms perform well on dynamic data sets as well.

- The second research paper that we compared was "Parallel Triangle Counting and k-Truss Identification using Graph-centric Methods". In this paper, the performance of the algorithm was measured on CPU and GPU but in project paper F1-Score is used. CoreThenTruss algorithm was added in the process of calculating k-truss, to reduce time complexity which is not done in project paper.

- The third research paper that we compared was "Truss Decomposition in Massive Networks". In this paper the proposed method implements different/modified algorithms for moderate size graphs that can fit in the memory versus two I/O-efficient algorithms to handle massive networks that cannot fit in the memory. The performance for the proposed method was calculated using CPU time whereas our method uses F1-score.

- The fourth paper that was compared is "K-Truss Network Community Detection". The proposed method in our research paper is an extension of this paper's concepts. This paper states that the truss of the subgraph should be atleast (k-1), which is stated as (k-2) in our paper. Our paper states Querying K-Truss Community in Large and Dynamic Graphs and this paper is just for comparison between various Community Detection Algorithms. This paper just uses Truss Decomposition and additionally our paper uses Query Processing Using K-Truss Index, TCP-Index Construction, etc.

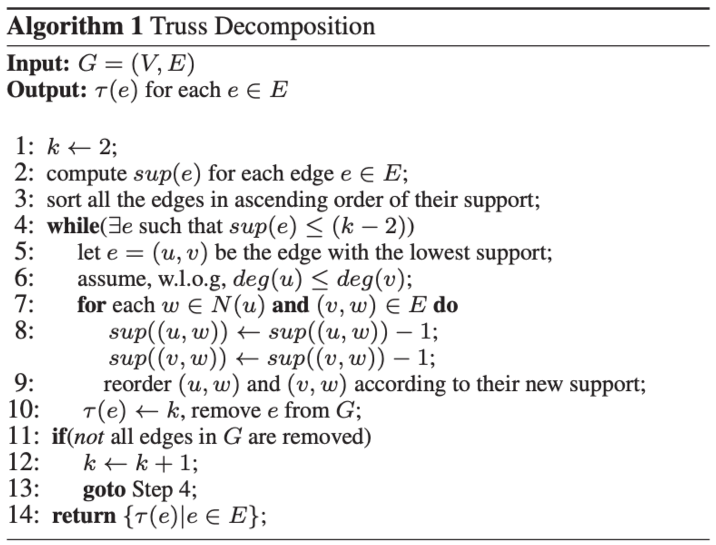
- The last paper that we compared was "Online Search of Overlapping Communities". γ as an average density measure, may not necessarily guarantee a cohesive community structure, while our paper states a novel community model based on the k-truss concept having an edge connectivity constraint on top of it and gives output as different communities which have cohesive structure. There are three parameters α, γ, k in this model, the setting of which may vary significantly for different query vertices, whereas our community model only needs to specify the trussness value k. In addition, a $(k + 1)$-truss community is contained in a k-truss community. Thus by using different k values for community query, we can get a hierarchical community structure of a query vertex. Also, Finding α-adjacency-γ-quasi-k-clique has been proven to be NP-hard, which imposes a severe computational bottle-neck and this algorithm cannot be computed in polynomial time but our algorithm can be computed in polynomial time, which make the k-truss community model computationally tractable and efficient.

DATASETS

For the purpose of evaluation, real world network data is used to measure the effectiveness of the proposed algorithm. The dataset is downloaded from Stanford Network Analysis Project Collection. From this collection, we utilized the data of social networks like Facebook. We also utilized two datasets of peer to peer networks called p2p-Gnutella08 and p2p-Gnutella04. The Facebook dataset consists of friends lists. The data was collected from survey participants using this Facebook app. The data has been anonymized by replacing the Facebook-internal ids for each user with a new value. This dataset contains 4039 nodes and 88234 edges and based on the number of nodes and edges we can tell that this dataset is a densely connected graph. We can validate this from checking the statistics of the graph as there are 1612010 triangles that are connected. The other two datasets that we utilized are from peer to peer network. p2p-Gnutella08 contains 6301 nodes and 20777 edges where there are 2383 triangles that are connected. p2p-Gnutella04 contains 10876 nodes and 39994 edges where there are 934 triangles that are connected. Both of these datasets are not densely connected but not sparse either. We can validate this by the number of triangles as it is relatively low where the computational time is reasonable. For the development purpose we used Visual Studio Code and our programming language is Python in which some of the many libraries that we are utilizing are networkx, pandas, numpy, etc.

ALGORITHMS

1. **Truss Decomposition**: A method to discover cohesive subgraphs and to study the hierarchical structure among them. This algorithm focuses on finding k-truss of the complex static graph in a reduced possible amount of time and with less computation. A k-truss in a graph is a subset of the graph where each edge is supported by at least k additional edges that can form triangles with it. To put in other way, every edge in the truss needs to be a member of k-2 triangles made up of truss-related nodes. According to the definitions of subgraph trussness in paper: The trussness of a subgraph H ⊆ G is the minimum support of an edge in H, denoted by τ (H) = min{sup(e,H) : e ∈ E(H)} The trussness of an edge is e ∈ E(G) is defined as τ (e) = maxH⊆G{τ (H) : e ∈ E(H)}.



Algorithm 1 Truss Decomposition

```
Input: $G = (V, E)$
Output: $\tau(e)$ for each $e \in E$

1:  $k \leftarrow 2$;
2:  compute $sup(e)$ for each edge $e \in E$;
3:  sort all the edges in ascending order of their support;
4:  while($\exists e$ such that $sup(e) \leq (k-2)$)
5:      let $e = (u, v)$ be the edge with the lowest support;
6:      assume, w.l.o.g, $deg(u) \leq deg(v)$;
7:      for each $w \in N(u)$ and $(v, w) \in E$ do
8:          $sup((u, w)) \leftarrow sup((u, w)) - 1$;
            $sup((v, w)) \leftarrow sup((v, w)) - 1$;
9:          reorder $(u, w)$ and $(v, w)$ according to their new support;
10:     $\tau(e) \leftarrow k$, remove $e$ from $G$;
11: if(not all edges in $G$ are removed)
12:     $k \leftarrow k + 1$;
13:     goto Step 4;
14: return $\{\tau(e) | e \in E\}$;
```

- Finding Support
    - Triangle counting can be performed by iterating over the edges of the graph.
    - For each edge(u, v) check if nodes u and v have a common neighbor w; if so, nodes u,v,w form a triangle.
    - The common neighbors of nodes u and v can be determined by intersecting the edge lists of u and v. We sort the list because traversing on a sorted list will reduce time complexity.
- Finding subgraphs with the help of truss value

    - The value of k starts from 2, we start removing the lowest support edge by checking a condition i.e. if support(e) is less than equal to k-2.
    - For k=4, the edges with support < 4 - 2 = 2 should be removed and then supports for remained edges are updated.
    - Finally, we return a dictionary containing keys as truss number and values as list of edges having that particular truss number.

2. **Query Processing Using K-Truss Index**: The procedure to process a k-truss community query based on the simple index. This algorithm is an extension of the first algorithm. So, we first create a straightforward k-truss index and then suggest a k-truss community search algorithm based on the index.

Algorithm 2 Query Processing Using K-Truss Index

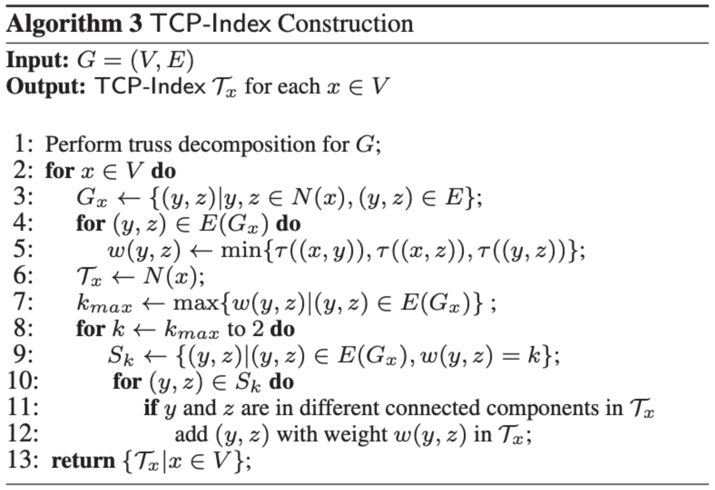
Input: $G = (V, E)$, an integer k, query vertex v_q
Output: k-truss communities containing v_q

```
$visited \leftarrow \emptyset; l \leftarrow 0;$
for $u \in N(v_q)$ do
    if $\tau((v_q, u)) \geq k$ and $(v_q, u) \notin visited$
        $l \leftarrow l + 1; C_l \leftarrow \emptyset; Q \leftarrow \emptyset;$
        $Q.push((v_q, u)); visited \leftarrow visited \cup \{(v_q, u)\};$
        while $Q \neq \emptyset$
            $(x, y) \leftarrow Q.pop(); C_l \leftarrow C_l \cup \{(x, y)\};$
            for $z \in N(x) \cap N(y)$ do
                if $\tau((x, z)) \geq k$ and $\tau((y, z)) \geq k$
                    if $(x, z) \notin visited$
                        $Q.push((x, z)); visited \leftarrow visited \cup \{(x, z)\};$
                    if $(y, z) \notin visited$
                        $Q.push((y, z)); visited \leftarrow visited \cup \{(y, z)\};$
return $\{C_1, \ldots, C_l\};$
```

- Given: A graph g, an integer value k and a query vertex q
- Main goal: Return a list of communities
- Check k value. If k = 2 then return whole graph as one community, or continue
- Compute Truss Decomposition (Algorithm 1)
- Iterate over the neighbors of the query vertex q and check if the edge trussness is greater than or equal to k
    - If two nodes satisfy the k value requirement, then the value of k-truss index is incremented, and we add that edge into the visited list.
- Add the unique edge to the community and a new stack Q
- Iterate over the stack Q and pop the first edge in it and check the edge's nodes if they form a triangle
    - Check the trussness of both edges
    - If trussness is greater than or equal to k for both edges, then we add it to the visited and Q
- Append this one community along with the list of other communities and return it after the iteration ends

3. **TCP-Index Construction**: Triangle Connectivity Preserved Index (TCP-Index) is a compact and elegant structure, which uses an efficient algorithm to process a k-truss community query. Using the straightforward k-truss index as its query processing mechanism, Algorithm 2 has two shortcomings. In particular, it requires the algorithm to access neighboring edges (x, z) and (y, z) for each common neighbor z of x and y for every edge (x, y) that has previously been included in

Cl. The below two scenarios explains the significant and unneeded computing overhead. Unnecessary access of disqualified edges: if τ((x,z))<k or τ((y,z)) < k, (x,z), (y,z) will not be included in Cl, thus accessing and checking such disqualified edges is a waste of time and computation. Repeated access of qualified edges: For each edge (u, v) in Cl, it is accessed at least 2*(k − 2) times in the BFS traversal, which is also a waste of time and computation. This is because τ ((u, v)) ≥ k, (u, v) is present in at least (k − 2) triangles by definition. For each such triangle denoted as uvw, (u, v) will be accessed twice when we do BFS expansion from the other two edges (u, w), (v, w). It follows that the query time of Algorithm 2 is lower bounded by Ω(k|Ans|). Considering these two drawbacks, the authors of this paper, design a novel Triangle Connectivity Preserved Index, or TCP-Index in short, which avoids the computational problems in Algorithm 2. Remarkably, the TCP-Index supports the k-truss community query in O(|Ans|) time, which is essentially optimal. Meanwhile, the TCP-Index can be constructed in O(Σ(u,v)∈E min{d(u), d(v)}) time and stored in O(m) space, which has exactly the same complexity as the simple k-truss index.



Algorithm 3 TCP-Index Construction

Input: $G = (V, E)$
Output: TCP-Index $\mathcal{T}_x$ for each $x \in V$

```
1:  Perform truss decomposition for $G$;
2:  for $x \in V$ do
3:      $G_x \leftarrow \{(y, z) | y, z \in N(x), (y, z) \in E\}$;
4:      for $(y, z) \in E(G_x)$ do
5:          $w(y, z) \leftarrow \min\{\tau((x, y)), \tau((x, z)), \tau((y, z))\}$;
6:      $\mathcal{T}_x \leftarrow N(x)$;
7:      $k_{max} \leftarrow \max\{w(y, z) | (y, z) \in E(G_x)\}$ ;
8:      for $k \leftarrow k_{max}$ to 2 do
9:          $S_k \leftarrow \{(y, z) | (y, z) \in E(G_x), w(y, z) = k\}$;
10:         for $(y, z) \in S_k$ do
11:             if $y$ and $z$ are in different connected components in $\mathcal{T}_x$
12:                 add $(y, z)$ with weight $w(y, z)$ in $\mathcal{T}_x$;
13: return $\{\mathcal{T}_x | x \in V\}$;
```

- To overcome the Unnecessary access of disqualified edges and repeated access of qualified edges, we Introduce TCP_Index
- To initialize The TCP-Index, for each vertex x ∈ V , we build a subgraph Gx, where Vertices in Gx = Neighbors of x, and Edges of x and all its neighbors.
- For each edge present in all the triangles of this subgraph we assign a weight which is the minimum of all the trussness values among the edges present in a particular triangle.
- Then the TCP-Index(Tx) is a list containing all the neighbors of node x.
- K_max is assigned to be the maximum weight value of subgraph Gx.
- We iterate this k_max from range k_max to 2 and if the weight of any edge = this k_max value then we push that particular edge to a list Sx.
- The above steps are repeated until all the nodes in Gx are Traversed.
- Essentially, Tx is the maximum spanning forest of Gx. The trees Tx for all x ∈ V form the TCP-Index of graph G.
- And finally, we return the TCP-Index Tx

4. **Query Processing Using TCP-Index**: According to the TCP-index, if any vertices are connected through edges with the same number of weights, these vertices should belong to the same k-truss community via adjacent triangles. Definition (k-level connected vertex set): For x ∈ V and y ∈

N(x), we use Vk(x, y) to denote the set of vertices which are connected with y through edges of weight ≥ k in Tx. We regard y also belongs to this set, i.e., y ∈ Vk (x, y).

Algorithm 4 Query Processing Using TCP-Index

Input: $G = (V, E)$, an integer k, query vertex v_q
Output: k-truss communities containing v_q

```
1:  $visited \leftarrow \emptyset$; $l \leftarrow 0$;
2:  for $u \in N(v_q)$ do
3:      if $\tau((v_q, u)) \geq k$ and $(v_q, u) \notin visited$
4:          $l \leftarrow l + 1$; $C_l \leftarrow \emptyset$; $Q \leftarrow \emptyset$;
5:          $Q.push((v_q, u))$;
6:          while $Q \neq \emptyset$
7:              $(x, y) \leftarrow Q.pop()$;
8:              if $(x, y) \notin visited$
9:                  compute $V_k(x, y)$;
10:                 for $z \in V_k(x, y)$ do
11:                     $visited \leftarrow visited \cup \{(x, z)\}$; $C_l \leftarrow C_l \cup \{(x, z)\}$;
12:                     if the reversed edge $(z, x) \notin visited$
13:                         $Q.push((z, x))$;
14: return $\{C_1, \cdots, C_l\}$;

15: Procedure compute $V_k(x, y)$
16: return $\{z | z$ is connected with $y$ in $\mathcal{T}_x$ through edges of weight $\geq k\}$;
```

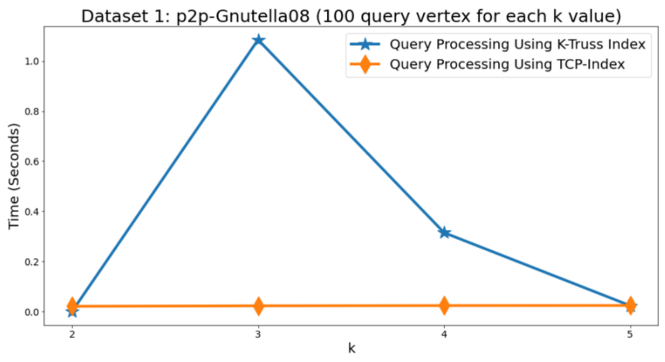
- The goal is to get a actual community for a query vertex q
- We utilize TCP-index(output of algo-3)
- If q is our query node, we will take the list of neighbors of q
- Then each incident edge with respect to q ,we compare τ((q,u)) ≥ k and push the edge in Q
- Then the algorithm per- forms a BFS traversal using a queue Q .
- For an unvisited edge (x, y), it searches the vertex set Vk(x, y) from Tx
- For each z ∈ Vk(x,y), the edge (x,z) is added into Cl.
- Then we perform the reverse operation, i.e., if (z, x) is not visited yet, it is pushed into Q for z-centered community expansion using Tz. Note that (z, x) and (x, z) are considered different here.
- When Q becomes empty, all edges in Cl have been found. The process iterates until all incident edges of q have been processed. Finally, a set of k-truss communities containing q are returned.

EVALUATION

- **Test Dataset**: While writing the code we have created a demo graph to understand the logic and also to validate whether our code is returning the output as expected in the paper. In this graph we added 12 nodes and 26 edges. The runtime to compute all algorithms is less than a second.

- **Dataset #1**: p2p-Gnutella08

  We ran this dataset on our personal device as it contains 6301 nodes and 20777 edges forming 2383 triangles among all the nodes and edges. For this dataset, the highest possible k value that we achieved is 5. We ran algorithm 2 and algorithm 4 with randomly chosen 100 query vertices for every k value. We also calculated runtime for each randomly generated query vertex and calculated its average. Then we plotted these results as shown in the figure below.

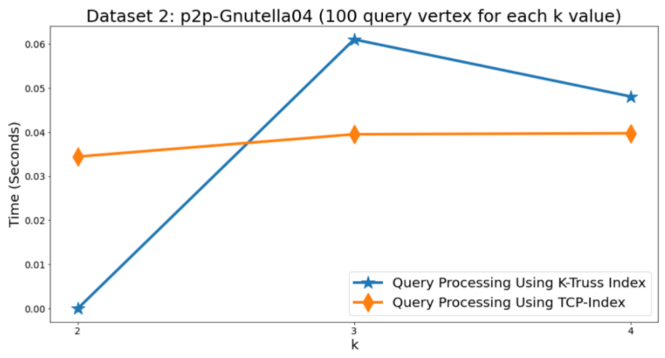


From the above plot, we can see that Query Processing using TCP-Index algorithm is more efficient than K-Truss Index algorithm. As for each k value, the average to find the community for any query vertex q is very less using TCP-Index than K-Truss Index. The average time values for each k value is as following.

- k=2
  - K-Truss: 1.4400482177734375e-06 seconds
  - TCP-Index: 0.020675482749938964 seconds
- k=3
  - K-Truss: 1.0849417114257813 seconds
  - TCP-Index: 0.02264768123626709 seconds
- k=4
  - K-Truss: 1.0849417114257813 seconds
  - TCP-Index: 0.023553359508514404 seconds
- k=5
  - K-Truss: 0.023295941352844237 seconds
  - TCP-Index: 0.023920440673828126 seconds

- **Dataset #2**: p2p-Gnutella04

We ran this dataset on our personal device as it contains 10876 nodes and 39994 edges forming 934 triangles among all the nodes and edges. For this dataset, the highest possible k value that we achieved is 4. We ran algorithm 2 and algorithm 4 with randomly chosen 100 query vertices for every k value. We also calculated runtime for each randomly generated query vertex and calculated its average. Then we plotted these results as shown in the figure below.



From the above plot, we can see that Query Processing using TCP-Index algorithm is more efficient than K-Truss Index algorithm. As for each k value, the average time to find the community for any query vertex q is very less using TCP-Index than K-Truss Index. The average time values for each k value are as following.

- k=2
  - K-Truss: 1.4281272888183594e-06 seconds
  - TCP-Index: 0.03443689823150635 seconds
- k=3
  - K-Truss: 0.06101433038711548 seconds
  - TCP-Index: 0.039503679275512696 seconds
- k=4
  - K-Truss: 0.0480766487121582 seconds
  - TCP-Index: 0.03972599983215332 seconds

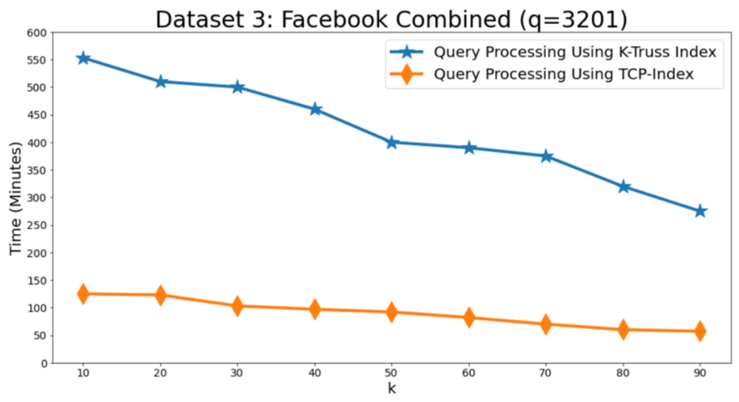
- **Dataset #3**: Facebook combined

We ran this dataset on a GSU server as it contains 4039 nodes and 88234 edges forming 1612010 triangles among all the nodes and edges which tells us that the graph is densely connected. We ran this dataset on all algorithms with different query vertex and k values for algorithm 2 and algorithm 4. Following is the runtime that our computer took to run the algorithms based on the input query and k value.

- k=3 and q=5: 57 hours and 17 mins
  - Algorithm1: 21 hours and 17 mins
  - Algorithm2: 8 hours and 30 mins
  - Algorithm3: 25 hours and 21 mins
  - Algorithm4: 2 hours and 9 mins

- k=4 and q=269: 55 hours and 37 mins
  - Algorithm1: 20 hours and 20 mins
  - Algorithm2: 7 hours and 17 mins
  - Algorithm3: 24 hours and 21 mins
  - Algorithm4: 3 hours and 39 mins

The highest possible k value of this dataset is 97. We ran the algorithm 2 and algorithm 4 with q=3201. Following plot represents the runtime for both algorithms where k value is represented on x-axis and time in minutes is represented on y-axis. As the k value increases, the runtime decreases for the algorithm 2 but for algorithm 4, the runtime decreases very slowly. Also, algorithm 2 is four times computationally expensive than algorithm 4 in terms of highly dense dataset like this.



There is one downside with this dataset, it is a very dense graph causing high computational time and therefore we ran it on the server and not on our personal computers.

CONCLUSION

Algorithms 1 and 3 have the highest computational time. Since the dataset #1 and dataset #2 has a high number of nodes and edges, the computational time was higher, but it was possible to run on our personal computer. Dataset #3 is very dense therefore it is not possible to run on our personal device therefore we ran it on a GSU server, however, the computational time was still high approximating 56 hours depending on the query. One challenge was finding the dataset that isn't too sparse or dense because if the dataset is too sparse then there might not exist any communities whereas if the dataset is too dense then there might be a lot of cliques which will increase the computational time and we can validate this from our dataset #3 which averaged 56 hours on a server.

CONTRIBUTION

- **Shraddha**: The main contribution for Shraddha was to write a script for Algorithm 1 which is Truss Decomposition. The biggest challenge for her was to understand the pseudocode. Since running the algorithm on a bigger dataset was time consuming and hard to evaluate the output, Shraddha created a small dataset which was used for testing only. This small dataset contains a reasonable number of nodes and edges which is easy to visualize and manually get the output to verify whether the algorithm is running correctly or not.

- **Dhruval**: The main goal for Dhruval was to write a script for Algorithm 2 which is Query Processing Using K-Truss Index. One big challenge here was that a bigger dataset was required to accurately get the communities output and the dataset that Shraddha created for testing purposes was not suitable for this algorithm mainly because there weren't a lot of communities that can get detected. Therefore, running this function on a bigger dataset was very important therefore an additional task for Dhruval was to find a GSU server where we can run the code. But in the meantime, we used our personal laptop although it was computationally very time consuming.

- **Kushal**: Kushal was assigned to write a script for Algorithm 3 which is TCP-Index Construction. This algorithm utilized Algorithm 1 which was written by Shraddha therefore Kushal had to verify whether the output of Truss Decomposition was correct or not. For this algorithm, it was possible to run the script on the small dataset that Shraddha created so Kushal tested his code by manually finding the output from the graph and comparing it with the script's output. Furthermore, these scripts were written in a notebook so Kushal converted the notebook into a script and added a few lines to output the result for each algorithm into a text file. This script was used for running on a GSU server.

- **Fatma**: The main task of Fatma was to write a script for Algorithm 4 which is Query Processing Using TCP-Index which utilizes the output of Algorithm 3 therefore Fatma had to verify whether Kushal's script was generating the correct output or not. The biggest challenge for Fatma was to understand the K-Level Connected Vertex Set definition. Additional work for Fatma was to find a bigger dataset from a reputable source such as SNAP (Stanford Network Analysis Project). There was a lot of graph dataset available on this platform but Fatma's duty was to find the dataset that is best suitable for our concept of K-Truss.

- ***Group work***: Although everyone in the group had individual tasks and contributions, we did work together at a few steps in this project. Shraddha and Dhruval worked together for writing their scripts and Kushal and Fatma worked together for writing theirs. Scripts had few bugs such as incorrect variable format causing errors in other algorithms. Also, it was a better choice to manually verify the results together to prevent any errors in our outputs. We also worked on writing the documentation together as per everyone's contribution.

SOURCE CODE

https://github.com/dhruval1219/Network-Science-Term-Project

REFERENCES

[1] Huang, X. et al. Querying K-truss community in large and dynamic graphs: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, ACM Conferences.

[2] Stanford Large Network Dataset Collection (SNAP) Datasets

[3] J.Cheng, Y.Ke, A.W.-C. Fu, J.X.Yu, and L.Zhu. Finding maximal cliques in massive networks by h*-graph.

[4] J.Wangand, J.Cheng. Truss decomposition in massive networks. PVLDB, 5(9):812–823, 2012

[5] J.Xie, S.Kelley, and B.K.Szymanski. Overlapping community detection in networks: The state-of-the-art and comparative study. ACM Comput. Surv., 45(4):43, 2013.

[6] Burkhardt, P., Faber, V., & Harris, D. G. Bounds and algorithms for k-truss.
[7] Voegele, C., Lu, Y.-S., Pai, S., & Pingali, K. Parallel Triangle counting and K-truss identification using Graph-Centric Methods.

[8] Wang, J., & Cheng, J. Truss decomposition in Massive Networks.

[9] McCulloh, I., & Savas, O. k-Truss Network Community Detection.

[10] Cui, W., Xiao, Y., Wang, H., Lu, Y., & Wang, W. Online search of Overlapping Communities.