Querying K-Truss Community in Large and Dynamic Graphs

Xin Huang[†], Hong Cheng[†], Lu Qin[‡], Wentao Tian[†], Jeffrey Xu Yu[†]
[†]The Chinese University of Hong Kong, China
[‡]Centre for Quantum Computation and Intelligent Systems, University of Technology, Sydney, Australia
[†]{xhuang,hcheng,wttian,yu}@se.cuhk.edu.hk, [‡]lu.qin@uts.edu.au

ABSTRACT

Community detection which discovers densely connected structures in a network has been studied a lot. In this paper, we study *online community search* which is practically useful but less studied in the literature. Given a query vertex in a graph, the problem is to find meaningful communities that the vertex belongs to in an online manner. We propose a novel community model based on the k-truss concept, which brings nice structural and computational properties. We design a compact and elegant index structure which supports the efficient search of k-truss communities with a linear cost with respect to the community size. In addition, we investigate the k-truss community search problem in a dynamic graph setting with frequent insertions and deletions of graph vertices and edges. Extensive experiments on large real-world networks demonstrate the effectiveness and efficiency of our community model and search algorithms.

Categories and Subject Descriptors

H.2.8 [**DATABASE MANAGEMENT**]: Database Applications—*Data mining*; G.2.2 [**DISCRETE MATHEMATICS**]: Graph Theory—*Graph algorithms*

Keywords

k-truss; community search; dynamic graph

1. INTRODUCTION

Community structure exists in many real-world networks, for example, social networks and biological networks. Community detection, which is to find communities in a network, has been studied a lot in the literature [15, 16, 17, 1, 25]. A different but related problem is *online community search*, which finds communities containing a query vertex in an online manner. These two tasks have different goals: community detection targets all communities in the entire network and usually applies a global criterion to find qualified communities. In contrast, online community search provides *personalized community detection* for a query vertex. As the communities for different vertices in a network may have very different characteristics, this user-centered personalized search is more meaningful. Furthermore, as the communities a user participates in represent the social contexts of the user, online community search provides a useful tool for other analytical tasks, such as social circle discovery [14] and social contagion modeling [20]. In this paper, we study the modeling and querying of the communities of a query vertex.

A recent study by Cui et al. [9] has proposed a novel approach for online overlapping community search. A new community model was defined as an α-adjacency-γ-quasi-k-clique. A γ-quasi-k-clique is a k-node graph with at least $\lfloor \gamma \frac{k(k-1)}{2} \rfloor$ edges. Another parameter α is imposed to union two γ-quasi-k-cliques if they share at least α vertices. Given a query vertex q, the problem is to find all α-adjacency-γ-quasi-k-cliques containing q. However, there are several limitations in this community model.

1. γ as an average density measure, may not necessarily guarantee a cohesive community structure. Consider the graph in Figure 1 which is a 0.8-quasi-7-clique containing query vertex q. However, q is only connected with one vertex in the community, thus it is not a cohesive community for q obviously.
2. There are three parameters α, γ, k in this model, the setting of which may vary significantly for different query vertices. For example, in a research collaboration network, the communities of a famous scholar and a junior scholar can be dramatically different in terms of the community size and density. Thus it is difficult to choose proper values for the three parameters given a query vertex.
3. Finding α-adjacency-γ-quasi-k-clique has been proven to be NP-hard [9], which imposes a severe computational bottleneck. The approximate algorithms for clique enumeration and expansion [9] reduce the complexity, but cannot give a theoretic guarantee of the approximation quality.

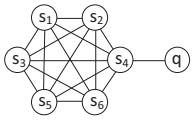


Figure 1: A 0.8-quasi-7-clique containing q

Considering these limitations, we propose a novel community 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 [7]. The k-truss is a type of cohesive subgraph defined based on triangle which models the stable relationship among three nodes. However, the k-truss subgraph may be disconnected, for example, the two shaded regions form the 4-truss subgraph in Figure 2(a) which is obviously disconnected.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.
Copyright 2014 ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2610495.

So the k-truss subgraph may not correspond to a meaningful community. On top of the k-truss, we 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 our novel ***k-truss community model***. To the best of our knowledge, this is the first work that proposes the k-truss community. Compared with the α-adjacency-γ-quasi-k-clique model, our community model has the following advantages.

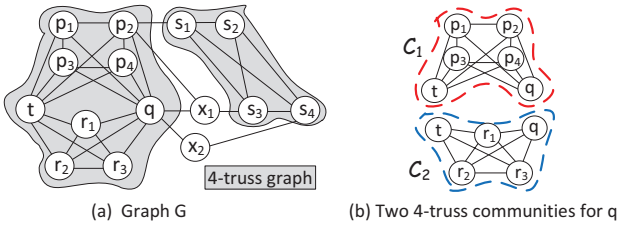
1. **Cohesive community**. The k-truss community has cohesive structure according to our analysis in Section 2. For example, the graph in Figure 1 is not a valid k-truss community containing q for $k \geq 3$, as the edge (q, s_4) is not in any of the triangles.
2. **Fewer parameter**. Our community model 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.
3. **Polynomial time algorithm**. There exist polynomial time algorithms for computing the k-truss subgraphs [7, 21], which make the k-truss community model computationally tractable and efficient.

Simply searching k-truss community by its definition may incur a large number of wasteful edge accesses as shown in Section 3.2. Thus the key to efficient k-truss community query processing is to design an effective index. Towards this goal, we first apply an efficient truss decomposition algorithm [21] on a graph G which computes the k-truss subgraphs for all k values. Then we design a novel and elegant index structure, called TCP-Index, to index the pre-computed k-truss subgraphs. The TCP-Index preserves the trussness value and the triangle adjacency relationship in a compact tree-shape index, and supports the query of k-truss community in *linear time* with respect to the community size, which is optimal.

We further study k-truss community search in dynamic graphs, where graph vertices and edges can be frequently inserted or deleted. We present a theoretical analysis to identify the scope in a graph that is affected by edge insertion/deletion. Specifically, we derive a tight upper bound of the trussness for a newly inserted edge, which allows us to precisely identify the affected region with a light cost. Then we design efficient algorithms to update the trussness value and the TCP-Index in the affected region. The incremental update algorithms effectively support querying k-truss community in highly dynamic graphs.

We conduct extensive experimental studies on large real-world networks and have the following findings. First, the k-truss community search using the TCP-Index is highly efficient in all networks. The query time is from one millisecond for the low degree query vertex to a few seconds for the high degree query vertex which has large and dense communities. The TCP-Index is very compact and can be constructed very efficiently. Second, the TCP-Index can be updated in milliseconds given an edge insertion/deletion. Thus it is highly efficient to support the k-truss community search in dynamic graphs. Last, we evaluate the quality of the discovered communities on two social networks with ground-truth communities and a scientific collaboration network. The results show that our community model can find cohesive and meaningful communities of a query vertex.

The rest of this paper is organized as follows. We formulate the k-truss community search problem in Section 2. We design a novel TCP-Index and an efficient k-truss community search algorithm in a static graph in Section 3. We further study how to maintain the TCP-Index for query processing in a dynamic graph in Section 4. Extensive experimental results on large real-world networks are reported in Section 5. We discuss related work in Section 6 and conclude this paper in Section 7.



(a) Graph G (b) Two 4-truss communities for q

Figure 2: K-truss community example

2. K-TRUSS COMMUNITY

2.1 Problem Definition

We consider an undirected, unweighted simple graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges. We denote the set of neighbors of a vertex v by $N(v)$, i.e., $N(v) = \{u \in V : (v, u) \in E\}$, and the degree of v by $d(v) = |N(v)|$. We use d_{max} to denote the maximum vertex degree in G.

A triangle in G is a cycle of length 3. Let $u, v, w \in V$ be the three vertices on the cycle, and we denote this triangle by $\triangle_{uvw}$. Then the *support* of an edge is defined as follows.

DEFINITION 1 (SUPPORT). *The support of an edge $e(u, v) \in E$ in G, denoted by $sup(e, G)$, is defined as $|\{\triangle_{uvw} : w \in V\}|$. When the context is obvious, we replace $sup(e, G)$ by $sup(e)$.*

If an edge e is contained in a triangle $\triangle$, we denote it by $e \in \triangle$. Now we give the definitions of *triangle adjacency* and *triangle connectivity* below.

DEFINITION 2 (TRIANGLE ADJACENCY). *Given two triangles $\triangle_1, \triangle_2$ in G, they are adjacent if $\triangle_1$ and $\triangle_2$ share a common edge, which is denoted by $\triangle_1 \cap \triangle_2 \neq \emptyset$.*

DEFINITION 3 (TRIANGLE CONNECTIVITY). *Given two triangles $\triangle_s, \triangle_t$ in G, $\triangle_s$ and $\triangle_t$ are triangle connected, if there exist a series of triangles $\triangle_1, \ldots, \triangle_n$ in G, where $n \geq 2$, such that $\triangle_1 = \triangle_s$, $\triangle_n = \triangle_t$ and for $1 \leq i < n$, $\triangle_i \cap \triangle_{i+1} \neq \emptyset$.*

For the graph G in Figure 2(a), $e(q, p_4)$ is contained in $\triangle_{qp_3p_4}$ and $\triangle_{qp_2p_4}$, thus its support $sup(e(q, p_4)) = 2$. $\triangle_{qp_3p_4}$ and $\triangle_{qp_2p_4}$ are triangle adjacent as they share a common edge $e(q, p_4)$. $\triangle_{tp_3p_4}$ and $\triangle_{qp_2p_4}$ are triangle connected through $\triangle_{qp_3p_4}$ in G.

On the basis of the definitions of support and triangle connectivity, we define the *k-truss community* as follows.

DEFINITION 4 (K-TRUSS COMMUNITY). *Given a graph G and an integer $k \geq 2$, G' is a k-truss community, if G' satisfies the following three conditions:*

(1) K-Truss. *G' is a subgraph of G, denoted as $G' \subseteq G$, such that $\forall e \in E(G')$, $sup(e, G') \geq (k-2)$;*

(2) Edge Connectivity. *$\forall e_1, e_2 \in E(G')$, $\exists \triangle_1, \triangle_2$ in G' such that $e_1 \in \triangle_1$, $e_2 \in \triangle_2$, then either $\triangle_1 = \triangle_2$, or $\triangle_1$ is triangle connected with $\triangle_2$ in G';*

(3) Maximal Subgraph. *G' is a maximal subgraph satisfying conditions (1) and (2). That is, $\nexists G'' \subseteq G$, such that $G' \subset G''$, and G'' satisfies conditions (1) and (2).*

Actually the largest subgraph that satisfies condition (1) is exactly the k-truss definition used in the literature [7, 21]. However,

the k-truss condition itself is insufficient to define a cohesive and meaningful community due to the following two reasons. First, a k-truss subgraph can be disconnected, thus does not represent a cohesive community. For example, the two shaded regions in Figure 2(a) form the 4-truss subgraph of G, which is obviously disconnected. So this 4-truss subgraph does not correspond to a meaningful community. Second, for a fixed k value, any vertex can belong to at most one k-truss subgraph. This cannot deal with a common scenario that a user can participate in multiple communities.

With these considerations, we impose the edge connectivity requirement in condition (2) to ensure the discovered communities are connected and cohesive. The rationale is that, a triangle represents the strong and stable relationship among three vertices. If any two edges in a subgraph are reachable from each other through a series of adjacent triangles, the subgraph must be connected, and have a cohesive structure among all involved vertices. This definition also allows a vertex to participate in multiple communities.

Example 1: Two 4-truss communities containing vertex q are shown in Figure 2(b) as C_1 and C_2, respectively. We can verify that every edge in C_1 is contained in at least two triangles in C_1, any two edges in C_1 are reachable through adjacent triangles, and C_1 is maximal. Thus C_1 is a 4-truss community. These properties also hold for another 4-truss community C_2. As the edges in C_1 cannot reach the edges in C_2 through adjacent triangles, C_1 and C_2 cannot merge as one large community. This is very reasonable, as there is no connection between the two vertex sets $\{p_1, p_2, p_3, p_4\}$ and $\{r_1, r_2, r_3\}$. In addition, we can see that vertices q and t participate in both communities. □

Problem Definition. The problem of **k-truss community search** studied in this paper is defined as follows. Given a graph $G(V, E)$, a query vertex $v_q \in V$ and an integer $k \geq 2$, find all k-truss communities containing v_q. In addition, we study k-truss community search in dynamic graphs, where vertices and edges are frequently inserted or deleted.

2.2 Why K-Truss Community?

Compared with the latest α-adjacency γ-quasi-k-clique [9] community model, the k-truss community model has three significant advantages: stronger guarantee on cohesive structure, fewer parameters and lower computational cost. These nice properties, which are inherited from the k-truss subgraph [7], not only lead to the discovery of more cohesive and meaningful communities, but also enable the design of more efficient, scalable and easier-to-use algorithms for community search. We present these properties in the following.

Bounded Diameter in K-Truss Community. The diameter of a k-truss community C with $|C|$ vertices is no larger than $\lfloor \frac{2|C|-2}{k} \rfloor$ [7]. This property guarantees that the shortest distance between any two vertices in a community is bounded, which has been considered as an important feature of a good community in [10].

Consider the 4-truss community C_1 in Figure 2(b) as an example. The diameter of C_1 is 2, which is the same as the diameter upper bound $\lfloor \frac{2\times 6-2}{4} \rfloor = 2$.

(K-1)-Edge-Connected Graph. A graph is $(k-1)$-edge-connected if it remains connected whenever fewer than $k-1$ edges are removed [11]. A k-truss community is $(k-1)$-edge-connected [7]. This property ensures the high connectivity of a community, which has been proposed as a criterion of a good community in [13].

In contrast, the γ-quasi-k-clique is not $(k-1)$-edge-connected for $\gamma < 1$. The 0.8-quasi-7-clique in Figure 1 becomes disconnected when one edge is removed.

Fewer Parameter. In the k-truss community model, we only need to specify the trussness value k, which controls or affects the diameter, the edge connectivity, and the edge support in a community. In contrast, the α-adjacency γ-quasi-k-clique model uses three parameters, the adjacency parameter α, the density γ and the clique size k. Although having more parameters may give more leverage on the properties of the community, it is much more difficult to choose proper values for different parameters.

Polynomial Time Complexity. There exist polynomial time algorithms [7, 21] for computing k-truss subgraphs. By applying such algorithm, we can compute the k-truss subgraphs for all k. The pre-computed results enable us to design compact index structures and efficient algorithms for querying k-truss communities. In contrast, finding γ-quasi-k-cliques has been proven to be NP-hard [9], which imposes a severe computational bottleneck.

2.3 Variations of K-Truss Community

The k-truss community can be extended to several interesting variants.

Densest K-Truss Community. It is interesting to discover the densest k-truss community containing a user q, that is, a k-truss community containing q that maximizes the trussness value k.

Most Diverse Communities. It is interesting to know how diverse the social contexts of a user q are, where a community represents a distinct social context. For a set of k-truss communities containing q as $\{C_1, \ldots, C_l\}$, the community diversity is measured by the entropy of the community vertex size distribution: $Div(q,k) = -\sum_{1\leq i\leq l} \frac{|C_i|}{\sum_{1\leq j\leq l}|C_j|} \log \frac{|C_i|}{\sum_{1\leq j\leq l}|C_j|}$. By choosing k which maximizes the diversity, i.e., $k = \arg\max_k Div(q,k)$, we can find the most diverse communities.

Other Community Models. The community model can be generalized using other dense subgraph definitions, such as the k-core community, and k-edge-connected component community.

The proposed techniques for k-truss community search in this paper can be extended to solve the above variant forms.

3. QUERYING K-TRUSS COMMUNITY

In this section, we study how to process a k-truss community query on a graph. We first design a simple k-truss index which is then proven to incur unnecessary computational overhead for query processing. Then we design a compact and elegant structure, called *Triangle Connectivity Preserved Index* (TCP-Index), and a highly efficient algorithm to process a k-truss community query.

3.1 Subgraph and Edge Trussness

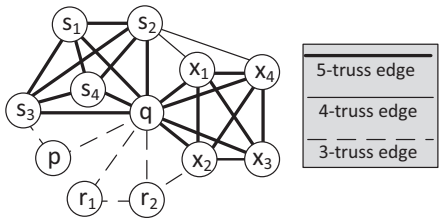


Figure 3: An example graph for k-truss community search

We define the trussness of a subgraph and an edge as follows.

DEFINITION 5 (SUBGRAPH TRUSSNESS). *The trussness of a subgraph $H \subseteq G$ is the minimum support of an edge in H, denoted by $\tau(H) = \min\{sup(e,H) : e \in E(H)\}$.*

DEFINITION 6 (EDGE TRUSSNESS). *The trussness of an edge $e \in E(G)$ is defined as $\tau(e) = \max_{H\subseteq G}\{\tau(H) : e \in E(H)\}$.*

Algorithm 1 Truss Decomposition

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

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

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

```
1: visited ← ∅; l ← 0;
2: for u ∈ N(v_q) do
3:     if τ((v_q, u)) ≥ k and (v_q, u) ∉ visited
4:         l ← l + 1; C_l ← ∅; Q ← ∅;
5:         Q.push((v_q, u)); visited ← visited ∪ {(v_q, u)};
6:         while Q ≠ ∅
7:             (x, y) ← Q.pop(); C_l ← C_l ∪ {(x, y)};
8:             for z ∈ N(x) ∩ N(y) do
9:                 if τ((x, z)) ≥ k and τ((y, z)) ≥ k
10:                    if (x, z) ∉ visited
11:                        Q.push((x, z)); visited ← visited ∪ {(x, z)};
12:                    if (y, z) ∉ visited
13:                        Q.push((y, z)); visited ← visited ∪ {(y, z)};
14: return {C_1, ..., C_l};
```

Such a subgraph H which defines $\tau(e)$ is denoted as H^e. It follows that $\tau(H^e) = \tau(e)$. For an edge e and $2 \le k \le \tau(e)$, we denote the k-truss community containing e as H_k^e, which is unique in the sense that no other k-truss community contains e. We use k_{gmax} to denote the maximum trussness of any edge in G.

Consider the graph in Figure 3 as an example. The trussness of the edge $e(s_1, s_2)$ is $\tau(e) = 5$, and the subgraph H^e is the 5-clique on the vertex set $\{q, s_1, s_2, s_3, s_4\}$.

3.2 A Simple K-Truss Index

We first design a simple k-truss index and propose an algorithm for k-truss community search based on the index.

K-Truss Index Construction. We first apply a truss decomposition algorithm [21] on the graph G, which computes the trussness of each edge. For the self-completeness of this paper, the truss decomposition algorithm [21] is outlined in Algorithm 1. After the initialization, for each k starting from $k = 2$, the algorithm iteratively removes a lowest support edge $e(u, v)$ if $sup(e) \le k - 2$. We assign the trussness of the removed edge as $\tau(e) = k$. Upon the removal of e, we also decrement the support of all other edges that form a triangle with e, and reorder them according to their new support. This process continues until all edges with support less than or equal to $(k - 2)$ are removed. In this way, we compute the trussness of all edges in G.

For each vertex $v \in V$, we sort its neighbors $N(v)$ in descending order of the edge trussness $\tau(e(u, v))$ for $u \in N(v)$. For each distinct trussness value $k \ge 2$, we mark the position of the first vertex u in the sorted adjacency list where $\tau(e(u, v)) = k$. This supports efficient retrieval of v's incident edges with a certain trussness value. We also use a hashtable to keep all the edges and their trussness values. This is the simple k-truss index.

Query Processing. Algorithm 2 outlines the procedure to process a k-truss community query based on the simple index. Given an integer k and a query vertex v_q, the algorithm checks every incident edge on v_q to search k-truss communities. If there exists an unvisited edge (v_q, u) with $\tau((v_q, u)) \ge k$, (v_q, u) is the seed edge to form a new community C_l. By definition, all the other edges in C_l can be reached from (v_q, u) through adjacent triangles. So we push (v_q, u) into a queue Q and perform a BFS traversal to search for other edges for expanding C_l, i.e., edges which have trussness no less than k and form triangles with edges already in C_l (line 6-13). When Q becomes empty, all edges in C_l have been found. Then the algorithm checks the next unvisited incident edge of v_q for forming a new community C_{l+1}. This process iterates until all incident edges of v_q have been processed. Finally a set of k-truss communities containing v_q are returned.

The correctness of Algorithm 2 is apparent since the algorithm essentially computes k-truss communities by definition, that is, exploring triangle connected edges with trussness no less than k in a BFS manner. We show the complexity of the simple k-truss index construction and query processing by Algorithm 2 as follows.

THEOREM 1. *The construction of the simple k-truss index takes $O(\sum_{(u,v)\in E} \min\{d(u), d(v)\})$ time and $O(m)$ space. The index size is $O(m)$. Algorithm 2 takes $O(d_{Amax}|Ans|)$ time to process one query, where $Ans = C_1 \cup \ldots \cup C_l$ is the union of the produced k-truss communities, $|Ans|$ is the edge number in Ans and d_{Amax} is the maximum vertex degree in Ans.*

PROOF. The truss decomposition algorithm (Algorithm 1) takes $O(\sum_{(u,v)\in E} \min\{d(u), d(v)\})$ time and $O(m)$ space for computing the trussness of all edges. Sorting the adjacency lists of all vertices in G takes $O(m)$ time and $O(m)$ space. Building an edge hashtable costs $O(m)$ time and $O(m)$ space. Thus, the construction of the k-truss index takes $O(\sum_{(u,v)\in E} \min\{d(u), d(v)\})$ time and $O(m)$ space. The index size is $O(m)$.

In k-truss community search, for each edge (u, v) in the generated communities, Algorithm 2 accesses the common neighbors of u and v, i.e., $N(u) \cap N(v)$ (line 7-9), whose size is bounded by d_{Amax}. Thus the query time complexity is $O(d_{Amax}|Ans|)$. □

Example 2: Suppose we want to query the 4-truss communities containing vertex q in the graph in Figure 3. Algorithm 2 first visits edge (q, s_1) with $\tau((q, s_1)) = 5 \ge 4$, and adds it into Q. The algorithm pops (q, s_1) from Q and inserts it into a new community C_1. Then the algorithm checks the common neighbors of q and s_1 and the edges between them. Consider a common neighbor s_2 as an example. As $\tau((q, s_2)) \ge 4$ and $\tau((s_1, s_2)) \ge 4$, both edges (q, s_2) and (s_1, s_2) are then inserted into C_1 and also pushed into Q for further expansion. This BFS expansion process continues until Q is empty and the 4-truss community C_1 is the induced subgraph by the vertex set $\{q, s_1, s_2, s_3, s_4, x_1, x_2, x_3, x_4\}$. □

3.3 A Novel TCP-Index

3.3.1 Limitations of Simple K-Truss Index

Algorithm 2 has two drawbacks in its query processing mechanism by using the simple k-truss index. Specifically, in line 8-13, for any edge (x, y) that has already been included in C_l, the algorithm needs to access adjacent edges (x, z) and (y, z) for each

common neighbor z of x and y. The following two cases lead to unnecessary and excessive computational overhead.

1. **Unnecessary access of disqualified edges**: If $\tau((x,z)) < k$ or $\tau((y,z)) < k$, $(x,z),(y,z)$ will not be included in C_l, thus accessing and checking such disqualified edges *become a waste*.
2. **Repeated access of qualified edges**: For each edge (u,v) in C_l, it is accessed at least $2(k-2)$ times in the BFS traversal, which is a huge waste. This is because $\tau((u,v)) \geq k$, (u,v) is contained in at least $(k-2)$ triangles by definition. For each such triangle denoted as $\triangle_{uvw}$, (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 $\Omega(k|Ans|)$.

Considering these two drawbacks, we 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 $O(\sum_{(u,v)\in E}\min\{d(u),d(v)\})$ time and stored in $O(m)$ space, which has exactly the same complexity as the simple k-truss index.

3.3.2 TCP-Index Construction

We first make some observations from the example in Figure 3.

Observation 1: *Consider $\triangle_{pqs_3}$ in which the three edge trussness values are 5, 3, and 3. Then $\triangle_{pqs_3}$ can appear in a 3-truss community, but not in 4- or 5-truss communities, due to the two 3-truss edges. To generalize, a triangle $\triangle_{xyz}$ can appear only in k-truss communities where $k \leq \min\{\tau((x,y)),\tau((x,z)),\tau((y,z))\}$.*

Observation 2: *Consider a subgraph in Figure 4(a) which is extracted from Figure 3. By definition, vertices x_1,x_2,x_3 belong to the same 5-truss community containing q, as each involved edge is 5-truss, and $\triangle_{qx_1x_2}$ and $\triangle_{qx_1x_3}$ are adjacent via edge (q,x_1). Thus we can use a compact representation for vertex q as depicted in solid line in Figure 4(b), which preserves the trussness and adjacency information for community search. Note that there is no need to include edge (x_2,x_3), as the tree-shape structure clearly indicates that x_2,x_3 belong to the same 5-truss community by triangle adjacency.*

(a) Subgraph extracted from Fig. 3 (b) Compact representation

Figure 4: Compact representation of a community with q

Observation 3: *From Figure 3, we can see the two 5-truss communities $\{q,x_1,x_2,x_3,x_4\},\{q,s_1,s_2,s_3,s_4\}$ involving vertex q are contained in the 4-truss community $\{q,x_1,x_2,x_3,x_4,s_1,s_2,s_3,s_4\}$, which is in turn contained in the 3-truss community, which is the whole graph.*

Based on the above observations, we are ready to construct the TCP-Index using Algorithm 3. For each vertex $x \in V$, we build a graph G_x, where $V(G_x) = N(x)$, and $E(G_x) = \{(y,z)|(y,z) \in E(G), y,z \in N(x)\}$. For each edge $(y,z) \in E(G_x)$, we assign a weight $w(y,z) = \min\{\tau((x,y)),\tau((x,z)),\tau((y,z))\}$, which indicates that $\triangle_{xyz}$ can appear only in k-truss community where $k \leq w(y,z)$ based on Observation 1. The TCP-Index for vertex x is a tree structure, denoted as $\mathcal{T}_x$, which is initialized to be the node set $N(x)$. Then in line 8-12, for each k from the largest weight k_{max} to 2, we iteratively collect the set of edges $S_k \subseteq E(G_k)$ whose weight is k. For each $(y,z) \in S_k$, if y,z are still in different components of $\mathcal{T}_x$, we add (y,z) with a weight $w(y,z)$ into $\mathcal{T}_x$. Essentially, $\mathcal{T}_x$ is the maximum spanning forest of G_x. The trees $\mathcal{T}_x$ for all $x \in V$ form the TCP-Index of graph G.

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 ∈ V do
3:     G_x ← {(y,z)|y,z ∈ N(x),(y,z) ∈ E};
4:     for (y,z) ∈ E(G_x) do
5:         w(y,z) ← min{τ((x,y)),τ((x,z)),τ((y,z))};
6:     T_x ← N(x);
7:     k_max ← max{w(y,z)|(y,z) ∈ E(G_x)} ;
8:     for k ← k_max to 2 do
9:         S_k ← {(y,z)|(y,z) ∈ E(G_x), w(y,z) = k};
10:        for (y,z) ∈ S_k do
11:            if y and z are in different connected components in T_x
12:                add (y,z) with weight w(y,z) in T_x;
13: return {T_x|x ∈ V};
```

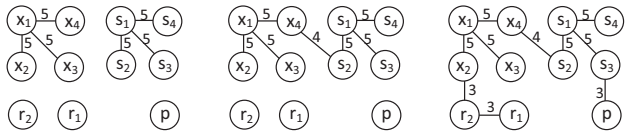


Figure 5: TCP-Index construction of vertex q

Example 3: Figure 5 shows the TCP-Index for vertex q in the graph in Figure 3. $\mathcal{T}_q$ is initialized to be $N(q)$. Figure 5(a) shows the tree structure when we add edges whose weights are 5. According to Observation 2, when the edges (x_1,x_2) and (x_1,x_3) are added into $\mathcal{T}_q$, the edge (x_2,x_3) will not be added into $\mathcal{T}_q$, as x_2,x_3 are already connected in $\mathcal{T}_q$ and we know that x_2,x_3 belong to the same 5-truss community by triangle adjacency. This is essential to keep $\mathcal{T}_q$ as a compact forest. The complete TCP-Index for q is shown in Figure 5(c). According to the community containment relationship in Observation 3, it is sufficient to use a single structure $\mathcal{T}_q$ for all trussness levels from k_{max} to 2. □

THEOREM 2. *The TCP-Index of graph G can be constructed in $O(\sum_{(u,v)\in E}\min\{d(u),d(v)\})$ time and $O(m)$ space by Algorithm 3. The index size is $O(m)$.*

PROOF. The first step costs $O(\sum_{(u,v)\in E}\min\{d(u),d(v)\})$ time. For a vertex $x \in V$, it takes $O(\sum_{y\in N(x)}\min\{d(x),d(y)\})$ time to list all triangles containing x to obtain G_x in line 3. The edge number $|E(G_x)|$ is bounded by $O(\sum_{y\in N(x)}\min\{d(x),d(y)\})$, thus $\mathcal{T}_x$ can be computed in $O(\sum_{y\in N(x)}\min\{d(x),d(y)\})$ time by Kruskal's algorithm. For all vertices in V, it takes $O(\sum_{x\in V}\sum_{y\in N(x)}\min\{d(x),d(y)\})$ time in total to build the TCP-Index. Thus the time cost of Algorithm 3 is $O(\sum_{(u,v)\in E}\min\{d(u),d(v)\})$.

For a vertex $x \in V$, G_x, as a subgraph of G, takes $O(m)$ space, which can be released after obtaining $\mathcal{T}_x$. $\mathcal{T}_x$, as a spanning forest on the vertex set $N(x)$, takes $O(|N(x)|)$ space. Thus the TCP-Index size for all vertices is $O(\sum_{x\in V}|N(x)|) = O(m)$. □

REMARK 1. *According to [6], $O(\sum_{(u,v)\in E}\min\{d(u),d(v)\}) \subseteq O(\rho m)$ where ρ is the arboricity of a graph G. $\rho \leq \min\{\lceil\sqrt{m}\rceil, d_{max}\}$ holds for any graph. Thus the TCP-Index construction time is $O(\sum_{(u,v)\in E}\min\{d(u),d(v)\}) \subseteq O(\rho m) \subseteq O(m^{1.5})$.*

3.3.3 Query Processing Using TCP-Index

We first illustrate query processing through an example, before we formally present the algorithm. According to the design of the TCP-Index, if two vertices are connected through a series of edges with weight $\geq k$ in $\mathcal{T}_x$ for $x \in V$, these two vertices belong to the same k-truss community via adjacent triangles. Consider $\mathcal{T}_q$ in Figure 5(c). As x_2, x_3 are connected through two edges with weight 5, they belong to the same 5-truss community. Thus we first define the *k-level connected vertex set* on a tree $\mathcal{T}_x$ to find all such vertices that belong to a k-truss community.

DEFINITION 7 (K-LEVEL CONNECTED VERTEX SET). *For $x \in V$ and $y \in N(x)$, we use $V_k(x, y)$ to denote the set of vertices which are connected with y through edges of weight $\geq k$ in $\mathcal{T}_x$. We regard y also belongs to this set, i.e., $y \in V_k(x, y)$.*

Example 4: If we want to query 5-truss communities containing q, we first visit an incident edge on q, (q, x_1) where $\tau((q, x_1)) = 5$. From $\mathcal{T}_q$ we retrieve the vertex set $V_5(q, x_1) = \{x_1, x_2, x_3, x_4\}$ as they are connected through edges with weight 5. According to Observation 2, these four vertices belong to the same 5-truss community with q. As $V_5(q, x_1) \subset N(q)$, we can construct part of the community as shown in Figure 6(a).

At this stage, we still miss the edges between the four vertices, for example, $(x_2, x_3), (x_3, x_4)$, etc. This is because $\mathcal{T}_q$ which is a spanning forest does not keep all the edges between the vertices. To fully recover all the edges in the 5-truss community, for each vertex $x_i \in V_5(q, x_1)$, we "reverse" the edge (q, x_i) to (x_i, q), then further expand the community in x_i's neighborhood. Take vertex x_2 as an example. We reverse (q, x_2) to (x_2, q) and then query x_2's index $\mathcal{T}_{x_2}$ to get the vertex set $V_5(x_2, q) = \{q, x_1, x_3, x_4\}$, as x_1, x_3, x_4 are connected with q in $\mathcal{T}_{x_2}$. Then we can obtain the edges between x_2 and every vertex in $V_5(x_2, q)$. After this, the community is shown in Figure 6(b). Similarly, we perform the reverse operation for each vertex x_1, x_3, x_4 and get the complete 5-truss community in Figure 6(c). We can observe that in this search process, each edge in a community is accessed exactly twice, for example, accessing (q, x_2) from $\mathcal{T}_q$ and (x_2, q) from $\mathcal{T}_{x_2}$. □

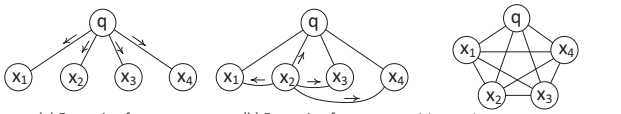


Figure 6: 5-truss community query on q using TCP-Index

Algorithm 4 outlines the procedure of query processing using the TCP-Index. Similar to Algorithm 2, Algorithm 4 computes the k-truss communities for a query vertex v_q by expanding from each incident edge on v_q in a BFS manner. If there exists an unvisited edge (v_q, u) with $\tau((v_q, u)) \geq k$, (v_q, u) is the seed edge to form a new community C_l (line 2-4). Then the algorithm performs a BFS traversal using a queue Q in line 5-13. For an unvisited edge (x, y), it searches the vertex set $V_k(x, y)$ from $\mathcal{T}_x$. The procedure of computing $V_k(x, y)$ is listed in line 15-16. For each $z \in V_k(x, y)$, the edge (x, z) is added into C_l. 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 $\mathcal{T}_z$. Note that (z, x) and (x, z) are considered different here. When Q becomes empty, all edges in C_l have been found. The process iterates until all incident edges of v_q have been processed. Finally a set of k-truss communities containing v_q are returned.

We prove the correctness of Algorithm 4 as follows.

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 ← ∅; l ← 0;
2: for u ∈ N(v_q) do
3:     if τ((v_q, u)) ≥ k and (v_q, u) ∉ visited
4:         l ← l + 1; C_l ← ∅; Q ← ∅;
5:         Q.push((v_q, u));
6:         while Q ≠ ∅
7:             (x, y) ← Q.pop();
8:             if (x, y) ∉ visited
9:                 compute V_k(x, y);
10:                for z ∈ V_k(x, y) do
11:                    visited ← visited∪{(x, z)}; C_l ← C_l∪{(x, z)};
12:                    if the reversed edge (z, x) ∉ visited
13:                        Q.push((z, x));
14: return {C_1, ··· , C_l};
15: Procedure compute V_k(x, y)
16: return {z|z is connected with y in T_x through edges of weight ≥ k};
```

LEMMA 1. *Given a query vertex $x \in V$ and an integer k, Algorithm 4 correctly computes all k-truss communities containing x.*

PROOF. First, for an edge (y, z) in $\mathcal{T}_x$, according to the definition $w(y, z) = \min\{\tau((x, y)), \tau((x, z)), \tau((y, z))\}$, if $w(y, z) \geq k$, $\triangle_{xyz}$ is included in a k-truss community of x.

Second, for two adjacent edges $(y, z_1), (y, z_2)$ in $\mathcal{T}_x$, it means $\triangle_{xyz_1}, \triangle_{xyz_2}$ are adjacent via edge (x, y).

Third, $V_k(x, y)$ contains the set of vertices which are connected with y through edges of weight $\geq k$ in $\mathcal{T}_x$. Based on the above two points, it leads to the discovery of all the triangles with weight $\geq k$ that can reach edge (x, y) in x's neighborhood. These connected triangles appear in the same k-truss community containing x.

Last, for an edge (x, z) where $z \in V_k(x, y)$, the same operation on its reverse edge (z, x) will further expand the k-truss community in z's neighborhood via $\mathcal{T}_z$. Thus the k-truss community is expanded via adjacent triangles in a BFS manner.

Based on the above points, we have established the correctness of Algorithm 4. □

THEOREM 3. *The time complexity of Algorithm 4 is $O(|Ans|)$, where $Ans = C_1 \cup \ldots \cup C_l$ is the union of the produced k-truss communities and $|Ans|$ is the number of edges in Ans.*

PROOF. Each edge (x, y) in the generated communities is accessed exactly twice: accessing (x, y) from $\mathcal{T}_x$ and (y, x) from $\mathcal{T}_y$. Thus the time complexity of Algorithm 4 is $O(|Ans|)$. □

Complexity Comparison. By using the TCP-Index and the simple k-truss index, each edge in a k-truss community is accessed *exactly twice versus at least* $2(k-2)$ *times*. In addition, the TCP-Index successfully avoids the unnecessary access of disqualified edges whose trussness is less than k. These are the key reasons to explain the query time difference between Algorithms 4 and 2, i.e., $O(|Ans|)$ versus $O(d_{Amax}|Ans|)$. Meanwhile, the TCP-Index construction has exactly the same time and space complexity as the simple k-truss index.

4. QUERYING K-TRUSS COMMUNITY IN DYNAMIC GRAPHS

In this section, we study k-truss community search in dynamic graphs where vertices and edges are inserted or deleted. We mainly focus on edge insertion and deletion, because vertex insertion/deletion

can be regarded as a sequence of edge insertions/deletions preceded/followed by the insertion/deletion of an isolated vertex.

Consider the insertion of edge $e_0(x, y)$ into G which leads to a set of new triangles $\{\triangle_{xyz} : z \in N(x) \cap N(y)\}$. Due to a new $\triangle_{xyz}$, the support of both edges $(x, z), (y, z)$ increases by 1. This may increase the subgraph trussness, $\tau(H)$, for $H \subseteq G$ which contains $\triangle_{xyz}$ since $\tau(H) = \min\{sup(e, H) : e \in E(H)\}$. It may in turn increase the trussness of those edges contained in H, as $\tau(e) = \max_{H \subseteq G}\{\tau(H) : e \in E(H)\}$, however, such edges may not necessarily be incident on vertices x or y. Edge deletion has a similar effect on decreasing edge trussness. To handle the edge trussness update efficiently, the key is to identify the affected region in the graph precisely. Thus, in Section 4.1 we present a theoretical analysis to define the scope that an edge insertion/deletion may affect. We design algorithms for updating the edge trussness and the TCP-Index in Sections 4.2 and 4.3, respectively.

4.1 Scope of Affected Edges

We use $\tau(e)$ and $\hat{\tau}(e)$ to represent the trussness of an edge e before and after an edge insertion/deletion respectively. We have the following three properties.

Rule 1: If e_0 is inserted into G with $\hat{\tau}(e_0) = l$, then $\forall e \in E(G)$ with $\tau(e) \geq l$, $\hat{\tau}(e) = \tau(e)$ holds.

Rule 2: If e_0 is deleted from G with $\tau(e_0) = l$, then $\forall e \in E(G) \setminus \{e_0\}$ with $\tau(e) > l$, $\hat{\tau}(e) = \tau(e)$ holds.

Rule 3: $\forall e \in E(G) \setminus \{e_0\}$, $|\hat{\tau}(e) - \tau(e)| \leq 1$ holds.

The rationale of Rules 1 and 2 is that e_0 is not included in a $(l + 1)$-truss subgraph. Rule 3 holds since the support of any edge changes by at most 1 with an edge insertion/deletion. These three properties can be rigorously proved, but the proof is omitted due to the space limit. In order to apply Rule 1, we need to obtain $\hat{\tau}(e_0)$ first. However, computing $\hat{\tau}(e_0)$ itself is costly. So we resort to an alternative, that is, we estimate an upper bound of $\hat{\tau}(e_0)$, denoted as $\overline{\hat{\tau}(e_0)}$, with a light cost, and apply Rule 1' instead of Rule 1.

Rule 1': If e_0 is inserted into G, then $\forall e \in E(G)$ with $\tau(e) \geq \overline{\hat{\tau}(e_0)} \geq \hat{\tau}(e_0)$, $\hat{\tau}(e) = \tau(e)$ holds.

To estimate $\overline{\hat{\tau}(e_0)}$, we define the *k-level triangles* of an edge.

DEFINITION 8 (K-LEVEL TRIANGLES). *For an edge $e(u, v)$ and $k \geq 2$, we denote the k-level triangles containing e by $\triangle^k_{(u,v)} = \{\triangle_{uvw} : \min\{\tau((u, w)), \tau((v, w))\} \geq k\}$. The number of triangles in $\triangle^k_{(u,v)}$ is denoted by $|\triangle^k_{(u,v)}|$.*

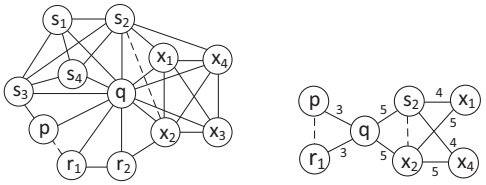
Lemma 2 gives the lower and upper bound of $\hat{\tau}(e_0)$.

LEMMA 2. *If an edge $e_0(u, v)$ is inserted into a graph, then $\hat{\tau}(e_0)$ satisfies $k_1 \leq \hat{\tau}(e_0) \leq k_2$ and $k_2 - k_1 \leq 1$, where $k_1 = \max_k\{k : |\triangle^k_{e_0}| \geq k - 2\}$, $k_2 = \max_k\{k : |\triangle^{k-1}_{e_0}| \geq k - 2\}$.*

PROOF. We first prove $\hat{\tau}(e_0) \geq k_1$. Since $k_1 = \max_k\{k : |\triangle^k_{e_0}| \geq k - 2\}$, we can find triangles $\triangle_{uvw_1}, \ldots, \triangle_{uvw_{k_1-2}}$ in which all edges have trussness no less than k_1 except for e_0. Each of such edges $(u, w_1), (v, w_1), \ldots, (u, w_{k_1-2}), (v, w_{k_1-2})$ corresponds to a unique k_1-truss community as $H^{(u,w_1)}_{k_1}$, $H^{(v,w_1)}_{k_1}, \ldots,$ $H^{(u,w_{k_1-2})}_{k_1}$, $H^{(v,w_{k_1-2})}_{k_1}$. Then we union all these k_1-truss communities with edge e_0 to form a connected subgraph H which becomes a seed of k_1-truss community by definition. Thus, for $e_0 \in H$, $\hat{\tau}(e_0) \geq k_1$ is true.

Next we prove $\hat{\tau}(e_0) \leq k_2$ by contradiction. Suppose to the contrary that $\hat{\tau}(e_0) = l > k_2$, then there exists an l-truss community $H^{e_0}_l$ in which edges have trussness no less than l, and have $l - 2$ triangles containing e_0 by definition. Since the trussness of all edges except e_0 increases by at most 1 after insertion by Rule 3, we have $|\triangle^{l-1}_{e_0}| \geq l - 2$ before insertion and then derive $k_2 \geq l$ by the definition of k_2, which is a contradiction.

Finally, we prove $k_2 - k_1 \leq 1$. By the definition of k_2, we have $|\triangle^{k_2-1}_{e_0}| \geq k_2 - 2 \geq k_2 - 3$, and derive $k_1 \geq k_2 - 1$ by the definition of k_1. Thus $k_2 - k_1 \leq 1$ is true. □



(a) G with Edge Insertion (b) Trussness Update

Figure 7: An example graph with edge insertion

COROLLARY 1. $\overline{\hat{\tau}(e_0)} = \max_k\{k : |\triangle^{k-1}_{e_0}| \geq k - 2\}$ *is an upper bound of* $\hat{\tau}(e_0)$.

The upper bound in Corollary 1 is extremely tight, as it may be larger than the real trussness by at most 1. For example, if we insert edge (p, r_1) into the graph in Figure 7(a), we have $\triangle^3_{(p,r_1)} = \{\triangle_{qpr_1}\}$ and $k_1 = k_2 = 3$ by Lemma 2. So we get $\hat{\tau}((p, r_1)) = 3$. If we insert another edge (s_2, x_2), we have $\triangle^4_{(s_2,x_2)} = \{\triangle_{s_2x_2q}$, $\triangle_{s_2x_2x_1}, \triangle_{s_2x_2x_4}\}$, $k_1 = 4$, $k_2 = 5$ by Lemma 2.

Due to the insertion/deletion of edge e_0, there are two reasons for edge $e \in E(G) \setminus \{e_0\}$ to change trussness, i.e., $\hat{\tau}(e) \neq \tau(e)$: (1) e forms/breaks a triangle with e_0 when e_0 is inserted/deleted; or (2) the edges of the triangles in which e lies have changed their trussness. We study the insertion case in Lemma 3.

LEMMA 3. *If an edge e_0 is inserted into a graph, we first assign $\tau(e_0) = \max_k\{k : |\triangle^k_{e_0}| \geq k - 2\}$. Then for $e(x, y) \in E(G) \cup \{e_0\}$ with $\tau(e) = l < \overline{\hat{\tau}(e_0)}$, e may have $\hat{\tau}(e) = l + 1$ **only in two cases**:*

(1) A new triangle with edges e, e_0 and another e' is formed, and $\min\{\tau(e_0), \tau(e')\} \geq l$ holds; or

(2) For $e(x, y)$, $\exists z \in N(x) \cap N(y)$, $\min\{\tau((x, z)), \tau((y, z))\} = l$ holds.

PROOF. An edge may increase the trussness by at most 1 with an insertion by Rule 3. For $e(x, y)$, assume $\tau(e) = l$, $\hat{\tau}(e) = l+1$. This implies $|\triangle^{l+1}_{(x,y)}| < l + 1$ before insertion, and $|\hat{\triangle}^{l+1}_{(x,y)}| \geq l + 1$ after insertion, where $\hat{\triangle}^{l+1}_{(x,y)} = \{\triangle_{xyz} : \min\{\hat{\tau}((x, z)), \hat{\tau}((y, z))\} \geq l + 1\}$. Then we prove cases (1) and (2) may lead to the increased trussness of e.

For case (1), the trussness of e_0 or e' may increase due to the edge insertion, thus it is possible to have $\min\{\hat{\tau}(e_0), \hat{\tau}(e')\} \geq l + 1$, and $|\hat{\triangle}^{l+1}_{(x,y)}| \geq l + 1$.

For case (2), as $\min\{\tau((x, z)), \tau((y, z))\} = l$, we know $\triangle_{xyz} \notin \triangle^{l+1}_{(x,y)}$. With an edge insertion, it is possible to have $\min\{\hat{\tau}((x, z)), \hat{\tau}((y, z))\} \geq l + 1$, and $|\hat{\triangle}^{l+1}_{(x,y)}| \geq l + 1$.

Next consider any edge $e(x, y) \in E(G) \cup \{e_0\}$ with $\tau(e) = l$, if it does not satisfy case (1), there is no new triangle containing e formed to account for trussness increase. Thus it is impossible to have $\hat{\tau}(e) = l + 1$. If it does not satisfy case (2), then the existing triangle $\triangle_{xyz}$ is with either $\min\{\tau((x, z)), \tau((y, z))\} < l$ or $\min\{\tau((x, z)), \tau((y, z))\} \geq l + 1$. After insertion we respectively have $\min\{\hat{\tau}((x, z)), \hat{\tau}((y, z))\} < l + 1$ or $\min\{\hat{\tau}((x, z)), \hat{\tau}((y, z))\} \geq l + 1$. For either situation, we can derive $|\hat{\triangle}^{l+1}_{(x,y)}| = |\triangle^{l+1}_{(x,y)}| < l + 1$. Thus if e satisfies neither case (1) nor (2), it is impossible to have the trussness increase as $\hat{\tau}(e) = l + 1$. □

According to Lemma 3, we summarize the affected edge trussness due to edge insertion/deletion.

Scope of Affected Edges. We define the weight of a triangle by the minimum edge trussness in the triangle. Then the insertion/deletion of an edge e_0 may lead to the following trussness update.

(1) **Insertion case**. For $e \in E(G) \cup \{e_0\}$ with $\tau(e) < \overline{\hat{\tau}(e_0)}$, if e, e_0 and another e' form a new triangle with weight $\tau(e)$, or e is connected with e_0 through a series of adjacent triangles each with weight $\tau(e)$, then e may have $\hat{\tau}(e) = \tau(e) + 1$.

(2) **Deletion case**. For $e \in E(G) \setminus \{e_0\}$ with $\tau(e) \leq \tau(e_0)$, if e, e_0 belong to a triangle with weight $\tau(e)$, or e is connected with e_0 through a series of adjacent triangles each with weight $\tau(e)$ before deletion, then e may have $\hat{\tau}(e) = \tau(e) - 1$.

4.2 Updating Edge Trussness

In this section, we propose an algorithm to update the edge trussness when an edge e_0 is inserted. The algorithm to handle edge deletion is similar and thus omitted. From Section 4.1 we know that only edges which have $\tau(e) < \overline{\hat{\tau}(e_0)}$, and are either in the same triangle with e_0 or connected to e_0 through adjacent triangles at the same trussness level may increase their trussness. Thus we first collect all edges in case (1) of Lemma 3 as candidates, then expand the candidate edges and examine their edge trussness level by level according to case (2). Finally, we use a variant of truss decomposition algorithm to finalize the trussness update.

The procedure to update edge trussness given an inserted edge e_0 is outlined in Algorithm 5. We first insert the edge e_0 and compute the range of $\hat{\tau}(e_0)$ as $[k_1, k_2]$ by Lemma 2. Then the algorithm sets $\tau(e_0) = k_1$ and the maximum edge trussness $k_{max} = k_2 - 1$ by Rule 1'. In line 4-9, by case (1) of Lemma 3, we collect every edge which forms a triangle with e_0, and has the minimum trussness in the triangle with $\tau(e) = k \leq k_{max}$. These edges are inserted into L_k. Then in line 10-30, for each k from k_{max} to 2, the algorithm updates the trussness of edges with $\tau(e) = k$ using three steps, namely, edge expansion (line 11-20), edge eviction (line 21-28), and trussness update (line 29-30). In the edge expansion step, the algorithm expands L_k by finding all edges with trussness k using breadth-first search through adjacent triangles with weight k by case (2) of Lemma 3 (line 14-20). Meanwhile, it computes the number of k-level triangles $|\triangle_e^k|$ as $s[e]$ (line 16). In the edge eviction step, the algorithm iteratively evicts edges with $s[e] \leq k - 2$ from L_k (line 22) until no such edges exist. After evicting an edge e, for each edge $e' \in L_k$ that forms a triangle of weight k with e, $s[e']$ is decreased by 1 (line 24-28). In the trussness update step, each edge $e \in L_k$ has $s[e] \geq k - 1$ and gets trussness update $\hat{\tau}(e) = k + 1$ (line 29-30). The three steps can be further optimized by pushing the edge eviction operation (line 21-28) into the edge expansion step (line 11-20) after each $s[(x, y)]$ is calculated, in order to avoid expanding useless edges in an early stage. The technique is similar to the early node eviction technique proposed in [18] for k-core update. Thus, we omit the detailed discussion on this heuristic due to space limitation.

Example 5: We update the edge trussness with the insertion of $e_0(s_2, x_2)$ in the graph shown in Figure 7(a). We first compute $k_1 = 4$, $k_2 = 5$, and assign $\tau(e_0) = 4$ and $k_{max} = 4$. Since $\tau(e_0) = k_{max}$ which indicates that e_0 may increase its trussness, we add (s_2, x_2) into L_4. Then the algorithm checks the edges forming new triangles with e_0 as shown in Figure 7(b), and adds (s_2, x_1) and (s_2, x_4) into L_4. Next the algorithm uses BFS to find the edges connected with those in L_4 through adjacent triangles with weight 4. As there is no such edge satisfying this condition, L_4 remains unchanged. Meanwhile, for each $e \in L_4$, it computes $|\triangle_e^4|$ as $s[e]$, that is, $s[(s_2, x_2)] = s[(s_2, x_1)] = s[(s_2, x_4)] = 3$ (shown in Figure 7(a)). As the three edges have $s[e] > 2$, the algorithm updates $\hat{\tau}(s_2, x_2) = \hat{\tau}(s_2, x_1) = \hat{\tau}(s_2, x_4) = 5$. □

Algorithm 5 Trussness Update with Edge Insertion

Input: $G = (V, E)$, the inserted edge $e_0 = (u, v)$
Output: Updated trussness $\hat{\tau}(e)$ for $e \in E(G) \cup \{e_0\}$

```
1: G.insert(e0);
2: compute [k1, k2] for τ̂(e0) by Lemma 2;
3: τ(e0) ← k1; kmax ← k2 − 1;
4: for k ← 2 to kmax do Lk ← ∅;
5: for w ∈ N(u) ∩ N(v) do
6:     k ← min{τ((w, u)), τ((w, v))};
7:     if k ≤ kmax then
8:         if τ((w, u)) = k then Lk ← Lk ∪ {(w, u)};
9:         if τ((w, v)) = k then Lk ← Lk ∪ {(w, v)};
10: for k ← kmax to 2 do
11:     Q ← ∅; Q.push(Lk);
12:     while Q ≠ ∅
13:         (x, y) ← Q.pop(); s[(x, y)] ← 0;
14:         for z ∈ N(x) ∩ N(y) do
15:             if τ((z, x)) < k or τ((z, y)) < k then continue;
16:             s[(x, y)] ← s[(x, y)] + 1;
17:             if τ((z, x)) = k and (z, x) ∉ Lk then
18:                 Q.push((z, x)); Lk ← Lk ∪ {(z, x)};
19:             if τ((z, y)) = k and (z, y) ∉ Lk then
20:                 Q.push((z, y)); Lk ← Lk ∪ {(z, y)};
21:     while ∃s[(x, y)] ≤ k − 2 in Lk
22:         Lk ← Lk \ {(x, y)};
23:         for z ∈ N(x) ∩ N(y) do
24:             if τ((x, z)) < k or τ((y, z)) < k then continue;
25:             if τ((x, z)) = k and (x, z) ∉ Lk then continue;
26:             if τ((y, z)) = k and (y, z) ∉ Lk then continue;
27:             if (x, z) ∈ Lk then s[(x, z)] ← s[(x, z)] − 1;
28:             if (y, z) ∈ Lk then s[(y, z)] ← s[(y, z)] − 1;
29:     for (x, y) ∈ Lk do
30:         τ̂((x, y)) ← k + 1;
```

4.3 Updating TCP-Index

We study updating the TCP-Index. Recall that, for $x \in V$, the index $\mathcal{T}_x$ is the maximum spanning forest of x's neighborhood graph G_x, where $V(G_x) = N(x)$, and $E(G_x) = \{(y, z) | (y, z) \in E(G), y, z \in N(x)\}$. Thus the key problem is how to update the maximum spanning forest of G_x given an edge insertion/deletion and the consequent edge trussness update.

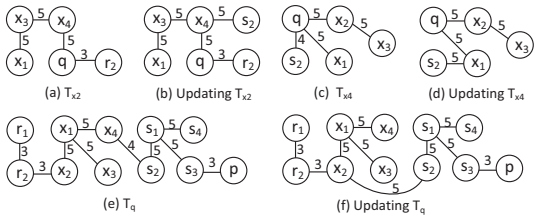


Figure 8: Updating TCP-Index

4.3.1 Updating TCP-Index With Edge Insertion

Index Update with Edge Insertion. With an inserted edge $e_0(u, v)$, we first consider how to update $\mathcal{T}_u$ and $\mathcal{T}_v$. Take vertex u as an example. G_u now includes a new vertex v and a set of new edges $\{(v, w) | w \in N(u) \cap N(v)\}$. Then we update the maximum spanning forest $\mathcal{T}_u$ with the new vertex and edges in G_u. The time cost is $O(|N(u)| + |N(u) \cap N(v)|) \subseteq O(|N(u)|)$.

For example, consider the insertion of (s_2, x_2) in Figure 7(a). The index $\mathcal{T}_{x_2}$ before insertion is shown in Figure 8(a). Now vertex

s_2 and three edges $(s_2, q), (s_2, x_1), (s_2, x_4)$ all with weight 5 are inserted into G_{x_2}. So the updated $\mathcal{T}_{x_2}$ is shown in Figure 8(b).

Now we consider how to update $\mathcal{T}_w$ for $w \in N(u) \cap N(v)$. G_w includes a new edge (u, v). We first compute the weight $w(u, v) = \min\{\hat{\tau}((u, v)), \hat{\tau}((u, w)), \hat{\tau}((v, w))\}$. If u, v are in different components of $\mathcal{T}_w$, we add the edge (u, v) with weight $w(u, v)$ into $\mathcal{T}_w$; otherwise, we can find a unique path P between u and v in $\mathcal{T}_w$. Then we compute the minimum weight on P as $w^* = \min_{e \in P} w(e)$. If $w^* \geq w(u, v)$, $\mathcal{T}_w$ remains unchanged; if $w^* < w(u, v)$, the corresponding edge is replaced by (u, v) in $\mathcal{T}_w$. The time cost is $O(|N(w)|)$.

Continue with the above example. With the insertion of (s_2, x_2), their common neighbor q's index before insertion is shown in Figure 8(e). There is a path (s_2, x_4, x_1, x_2) with a minimum weight 4. So we replace (s_2, x_4) by (s_2, x_2) with weight 5 in Figure 8(f).

Index Update with Trussness Increase. Besides the above cases, we also need to consider the index maintenance for an existing edge $e(x, y)$ which has trussness update $\hat{\tau}(e) = \tau(e) + 1$ due to the insertion of e_0. We discuss the update of $\mathcal{T}_x$ (and similarly $\mathcal{T}_y$). For every triangle $\triangle_{xyz}$, we denote $w_{xyz} = \min\{\tau((x, y)), \tau((x, z)), \tau((y, z))\}$, and $\hat{w}_{xyz} = \min\{\hat{\tau}((x, y)), \hat{\tau}((x, z)), \hat{\tau}((y, z))\}$. If $\hat{w}_{xyz} = w_{xyz}$, $\mathcal{T}_x$ remains unchanged. Otherwise, for $\hat{w}_{xyz} = w_{xyz} + 1$, if $(y, z) \in \mathcal{T}_x$, we update the edge weight $w(y, z) = \hat{w}_{xyz}$ in $\mathcal{T}_x$; if $(y, z) \notin \mathcal{T}_x$, we find the unique path P between y and z in $\mathcal{T}_x$ and update P with the new weight $w(y, z)$ in a similar process as described above for updating $\mathcal{T}_w$. For the neighbor vertex $z \in N(x) \cap N(y)$, the index $\mathcal{T}_z$ can be updated similarly.

Continue with the above example. With the insertion of (s_2, x_2), we have $\hat{\tau}((s_2, x_4)) = \tau((s_2, x_4)) + 1$, and $\hat{w}_{s_2x_4x_1} = 5$. The index $\mathcal{T}_{x_4}$ before insertion is shown in Figure 8(c). We use (s_2, x_1) with weight 5 to replace (s_2, q) with weight 4 in Figure 8(d).

4.3.2 Updating TCP-Index With Edge Deletion

Index Update with Edge Deletion. With a deleted edge $e_0(u, v)$, we first consider how to update $\mathcal{T}_u$ and $\mathcal{T}_v$. Take vertex u as an example. We first delete vertex v and edges $\{(v, w)|(v, w) \in \mathcal{T}_u\}$ from $\mathcal{T}_u$. Then we update the maximum spanning forest $\mathcal{T}_u$ with the available edges in $E(G_u)$.

Now we consider how to update $\mathcal{T}_w$ for $w \in N(u) \cap N(v)$. If $(u, v) \in \mathcal{T}_w$, we try to find an edge $(u', v') \in E(G_w)$ where $u' \in V_2(w, u), v' \in V_2(w, v)$ with the maximum weight to replace (u, v) in $\mathcal{T}_w$. If no such replacement edge exists, we simply remove (u, v). On the other hand, if $(u, v) \notin \mathcal{T}_w$, $\mathcal{T}_w$ remains unchanged.

Index Update with Trussness Decrease. Besides the above cases, we also need to consider the index maintenance for an existing edge $e(x, y)$ which has trussness update $\hat{\tau}(e) = \tau(e) - 1$ due to the edge deletion. We discuss the update of $\mathcal{T}_x$ (and similarly $\mathcal{T}_y$). For every $\triangle_{xyz}$, we denote $w_{xyz} = \min\{\tau((x, y)), \tau((x, z)), \tau((y, z))\}$, and $\hat{w}_{xyz} = \min\{\hat{\tau}((x, y)), \hat{\tau}((x, z)), \hat{\tau}((y, z))\}$. If $\hat{w}_{xyz} = w_{xyz}$, $\mathcal{T}_x$ remains unchanged. Otherwise, for $\hat{w}_{xyz} = w_{xyz} - 1$, if $(y, z) \notin \mathcal{T}_x$, $\mathcal{T}_x$ remains unchanged; if $(y, z) \in \mathcal{T}_x$, we try to find an edge $(y', z') \in E(G_x)$ where $y' \in V_{w_{xyz}}(x, y), z' \in V_{w_{xyz}}(x, z)$ and $w(y', z') = w_{xyz}$ to replace (y, z) in $\mathcal{T}_x$. If no such replacement edge exists, we just update $w(y, z) = \hat{w}_{xyz}$. For the neighbor vertex $z \in N(x) \cap N(y)$, the index $\mathcal{T}_z$ can be updated in a similar way.

5. EXPERIMENTS

We evaluate the efficiency and effectiveness of our proposed algorithms on real-world networks. All algorithms are implemented in C++ and all the experiments are conducted on Windows with 2.67GHz six-core CPU and 100GB main memory.

Datasets. We use 6 publicly available real-world networks to evaluate the algorithms. The network statistics are shown in Table 1. Except for Wise[1] (a micro-blogging network from WISE 2012 Challenge) and UK2002[2] (a web graph from a 2002 crawl of the .uk domain), all the other datasets are downloaded from the Stanford Network Analysis Project[3]. All networks are treated as undirected in the experiments.

Table 1: Network statistics (K $= 10^3$ and M $= 10^6$)

Network	$\|V_G\|$	$\|E_G\|$	d_{max}	k_{gmax}
WikiTalk	2.4**M**	5**M**	100029	53
Flickr	80**K**	11.8**M**	5706	308
LiveJournal	4.8**M**	69**M**	20333	362
Orkut	3.1**M**	117.2**M**	33313	78
Wise	58.6**M**	265.1**M**	278489	80
UK2002	18.6**M**	298.1**M**	194955	944

5.1 Query Processing

We evaluate and compare the performance of the two k-truss community query algorithms: Algorithm 2 that uses the simple k-truss index and Algorithm 4 that uses the TCP-Index.

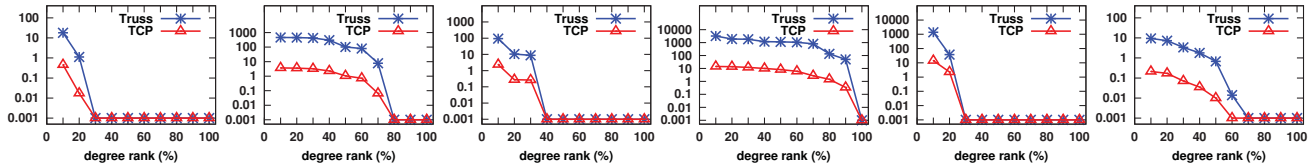
In the first experiment, we select query vertices with different degrees to test the query processing time. For each network, we sort the vertices in descending order of their degrees and partition them into 10 equal-sized buckets. We randomly select 100 vertices from each bucket for query. The average query processing time for each degree group is reported in Figure 9. We fix $k = 10$ for all networks except for WikiTalk and Wise which use $k = 4$, because the edges in these two networks have smaller trussness. As we can see, for the high degree query vertices which usually have larger and denser k-truss communities, the TCP-Index based method is two orders of magnitude faster than the k-truss index based method; whereas for the low degree query vertices which have smaller and sparser k-truss communities for the same k, the query time of the two methods is very close and is around a few milliseconds. This shows the superiority of the TCP-Index based query processing, especially for high degree query vertices.

In the second experiment, we vary the parameter k to test the query time for k-truss community search. For each network, we randomly generate two test sets: a set of 100 high degree query vertices (degree in top 30%) and another set of 100 low degree query vertices (degree in the remaining 70%). We denote the two query methods on the high/low degree test sets as Truss-H/Truss-L and TCP-H/TCP-L, respectively. Figure 10 shows the average query processing time of each method when we vary the parameter k. As we can see, for the high degree query vertices, the TCP-Index based method is two to three orders of magnitude faster than the k-truss based method for all k values; while for the low degree query vertices, the TCP-Index based method is still one to two orders of magnitude faster in most networks especially when k is small. Finally we can see that the query processing time decreases when k increases, because the discovered communities become smaller when k increases. This experiment again demonstrates the advantage of the TCP-Index based query processing and conforms with the time complexity analysis of the two query methods.

5.2 Index Construction

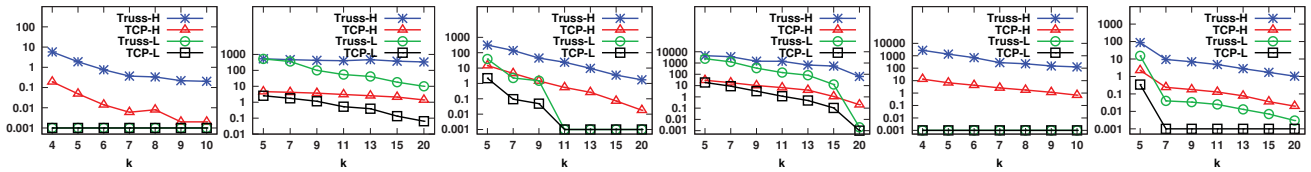
In this experiment, we compare the two indexing schemes: the simple k-truss index and the TCP-Index in terms of index size and index construction time in Table 2. Note that the index is main-

[1] http://www.wise2012.cs.ucy.ac.cy/challenge.html
[2] http://law.di.unimi.it/datasets.php
[3] snap.stanford.edu



(a) WikiTalk (b) Flickr (c) LiveJournal (d) Orkut (e) Wise (f) UK2002

Figure 9: Query time (in seconds) of different algorithms for query vertices in different degree percentile groups



(a) WikiTalk (b) Flickr (c) LiveJournal (d) Orkut (e) Wise (f) UK2002

Figure 10: Query time (in seconds) of different algorithms for different k

tained in memory for both schemes. The reported index time includes the truss decomposition time and index construction time.

We can observe that the size of the TCP-Index is about 3 times that of the k-truss index, and 4.3 times of the original graph size. This confirms that both indexing schemes have $O(m)$ space complexity and are very compact. In addition, the index construction is very efficient for both schemes. The index time of the TCP-Index is around 1.4–3.4 times that of the k-truss index. The longest TCP-Index time is 1.9 hours on Wise. The TCP-Index costs more time and space than the k-truss index for building the maximum spanning forest structure.

Table 2: Comparison of index size (in Megabytes) and index construction time (wall-clock time in seconds)

Network	Graph Size	Index Size		Index Time	
		K-Truss	TCP-Index	K-Truss	TCP-Index
WikiTalk	80	118	296	41	138
Flickr	90	135	485	690	1326
LiveJournal	672	1003	3174	1176	1686
Orkut	1792	2662	8714	2291	3342
Wise	4209	5960	11049	3078	6997
UK2002	4055	5980	21238	1374	2860

5.3 Updating TCP-Index in Dynamic Graphs

In this experiment, we evaluate the performance of incremental update of the TCP-Index when the input network is updated. For each network, we randomly insert/delete 1K edges, and update the edge trussness and the TCP-Index after each edge insertion/deletion. The average update time, including the edge trussness update time and the index update time, is reported in Table 3. In addition, we report the batch update time for the 1K edge insertions/deletions. All the experiments are repeated for 20 times, and the average performance is reported. For comparison, we also report the time for constructing the TCP-Index from scratch when the network is updated with an edge insertion/deletion.

Table 3: TCP-Index update time (wall-clock time in milliseconds)

Network	Insertion Per Edge	Insertion 1K Edges	Deletion Per Edge	Deletion 1K Edges	Computing from scratch
WikiTalk	0.2	125	3.9	2509	138000
Flickr	10.2	6344	58	33763	1326000
LiveJournal	0.7	693	3.9	1891	1686000
Orkut	16.1	17190	29.6	21351	3342000
Wise	7.8	3902	38.8	31282	6997000
UK2002	3.9	4065	12.2	12326	2860000

The results in Table 3 show that the update time per edge insertion ranges from 0.2 to 16.1 milliseconds. The batch update for 1K edge insertions can achieve further performance improvement, compared with the instant update which handles the inserted edges one by one. Thus, handling edge insertion is highly efficient.

For the deletion case, the update time per edge deletion ranges from 3.9 to 38.8 milliseconds. The batch update for 1K edge deletions also achieves further performance improvement. Compared with the insertion case, handling edge deletion is a little more costly, as it has a larger search space for finding a replacement edge in the TCP-Index.

We can see that the incremental update approach is several orders of magnitude faster than constructing the TCP-Index from scratch when a network is updated. This demonstrates the superiority of our proposed incremental update algorithms.

In addition, we also compare our edge trussness update method (Algorithm 5) with the update method for triangle k-core [26], which has the same definition with k-truss. The average update time upon an edge insertion/deletion is reported in Table 4. Our method takes 15 milliseconds or less for updating trussness in all networks. It is two to three orders of magnitude faster than the update method in [26]. The poor efficiency of [26] is due to its mechanism which updates the trussness by inserting/deleting triangles one by one for an edge insertion/deletion. Processing k-core triangles is very costly especially in dense graphs such as Flickr and Orkut.

Table 4: Edge trussness update time (wall-clock time in milliseconds)

Network	Insertion Per Edge		Deletion Per Edge	
	Triangle k-core	Our method	Triangle k-core	Our method
WikiTalk	0.21	0.07	40.43	0.19
Flickr	16700	5.99	17224	6.11
LiveJournal	1.75	0.54	60.45	0.63
Orkut	2453	15.13	179.2	1.67
Wise	5.59	3.21	18.33	1.32
UK2002	68.2	3.14	1445	2.76

5.4 Quality Evaluation

5.4.1 *Social Networks with Ground-truth Communities*

To evaluate the effectiveness of the k-truss community model, we use two social networks: Facebook and Twitter from the Stan-

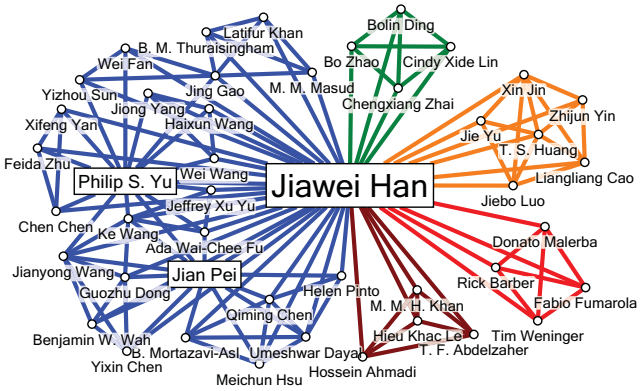


Figure 12: Five 5-truss communities containing Jiawei Han

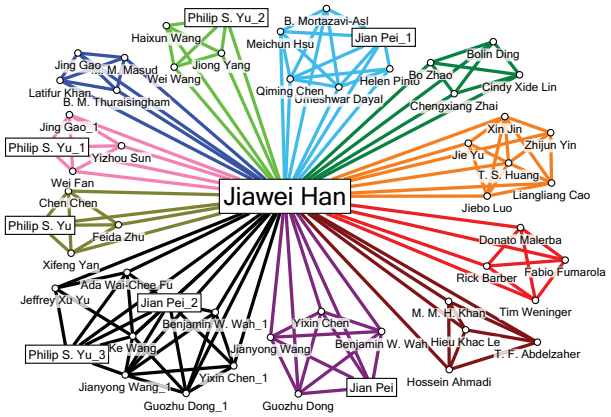


Figure 13: Eleven 4-adjacency-1.0-quasi-5-clique communities containing Jiawei Han

ford Network Analysis Project[3], both of which contain ground-truth communities for individual nodes. The Facebook network contains 4,039 vertices, 88,234 edges, and 10 query vertices with ground-truth communities in their neighborhood. The Twitter network contains 81,306 vertices, 1,768,149 edges, and 973 query vertices with ground-truth communities in their neighborhood. For a query, we denote the discovered communities as $\mathcal{C} = \{C_1, \ldots, C_i\}$, and the ground-truth communities as $\overline{\mathcal{C}} = \{\overline{C}_1, \ldots, \overline{C}_j\}$. We use the F_1 score to measure the alignment between a discovered community C and a ground-truth community $\overline{C}$. Since we do not know the correspondence between communities in $\mathcal{C}$ and $\overline{\mathcal{C}}$, we compute the optimal match via linear assignment [14] by maximizing

$$\max_{f:\mathcal{C}\to\overline{\mathcal{C}}} \frac{1}{|f|} \sum_{C\in dom(f)} F_1(C, f(C)), \qquad (1)$$

where f is a (partial) correspondence between $\mathcal{C}$ and $\overline{\mathcal{C}}$.

We compare the F_1 score of our k-truss community model and the α-adjacency-γ-quasi-k-clique model [9] using the executable program provided by the authors with a time limit of 60 seconds for a query. For our model, we vary the trussness parameter k; for the quasi-clique model, we follow the experimental setting in [9] to use the $(k-1,1)$-OCS model ($\alpha = k-1$, $\gamma = 1$) and vary the clique size k. Figure 11(a) and (c) show the F_1 score of the detected communities by both methods versus their respective parameter k on Facebook and Twitter. Although the parameter k has different meanings in the two models, the results still show a comprehensive performance comparison of our model and [9] over a broad range of parameter values. On both networks, we observe that our k-truss model has a very stable performance when we vary the trussness k. The $(k-1,1)$-OCS model is consistently worse than our method. For the quasi-clique model, we also tried other α and γ values. However, when we set $\alpha < k-1$ or $\gamma < 1$, the program cannot output all communities within the time limit set in the executable program due to the expensive quasi-clique enumeration.

Figure 11(b) and (d) report the average query time of our method and [9]. Our method is clearly more efficient than the $(k-1,1)$-OCS model.

5.4.2 *Case Study on DBLP*

We build a collaboration network from the DBLP data set[4] for case study. A vertex represents an author and an edge is added between two authors if they have co-authored 3 times or more. The network contains 234,879 vertices and 541,814 edges.

We query the 5-truss community containing 'Jiawei Han' which is shown in Figure 12. For comparison, we follow the case study in [9] which uses the $(k-1,1)$-OCS model to query 'Jiawei Han' by setting $k = 5, \alpha = 4, \gamma = 1$, which produces communities at a similar scale as shown in Figure 13. Note that we duplicate some authors who participate in more than one community in Figure 13, e.g., 'Jian Pei', 'Jian Pei_1' and 'Jian Pei_2', for a better visualization effect. We have the following observations.

(1) Our model generates 5 communities containing 'Jiawei Han', among which the 4 smaller ones are also found by the $(k-1,1)$-OCS model and depicted using the same color in Figure 13.

(2) The largest 5-truss community depicted in blue in Figure 12, however, is decomposed into 7 smaller communities by the $(k-1,1)$-OCS model in Figure 13. This phenomenon can be explained by the different mechanisms of the two community models. The $(k-1,1)$-OCS model tends to find the small, clique-based "*paper communities*", in which all the involved scholars appear in the same paper. For example, a paper community is formed by 'Jiawei Han', 'Philip S. Yu', 'Chen Chen', 'Xifeng Yan', and 'Feida Zhu'. In contrast, such small paper communities can be merged into a larger dense one by the triangle adjacency condition in our k-truss model. For example, two small paper communities can be merged if they share a common edge as ('Jiawei Han', 'Philip S. Yu') and form a 5-truss graph after being merged.

(3) A less restrictive community criterion can be realized by tuning α and γ in [9]. But in our experiment, if we set $\alpha < k-1$ or $\gamma < 1$, it cannot output all communities within the time limit set in the executable program due to the expensive quasi-clique enumeration.

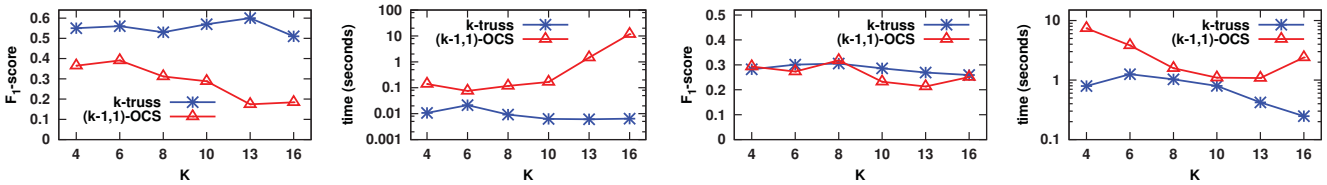
(4) Finally, we observe a community containing 'Guozhu Dong' and 5 other authors (depicted in purple) in Figure 13 is completely subsumed by another bigger community (depicted in black) in the same figure. Such duplicate output, which is not desired, may be explained by the approximate heuristics for clique enumeration and expansion in [9].

6. RELATED WORK

The related work to our study include community search and detection, and dense subgraph mining.

Community Search and Detection. Cui et al. [9] have recently studied the problem of online search of overlapping communities for a query vertex by designing a new α-adjacency γ-quasi-k-clique model. It is the most related work to ours and has been discussed in detail in Section 2 and compared empirically in Section 5. A different but related problem is community detection, which identifies communities in the entire network. There are two major cat-

[4] http://dblp.uni-trier.de/xml/



(a) Facebook F_1 score (b) Facebook query time (c) Twitter F_1 score (d) Twitter query time

Figure 11: Quality evaluation on social networks

egories: non-overlapping community detection [15, 17] and overlapping community detection [16, 1, 25].

Dense Subgraph Mining. There are different definitions of dense subgraph patterns, including clique [3, 5, 22, 24], quasi-clique [19], k-core [2, 4], k-truss [7, 21, 26], dense neighborhood graph [23], dense bipartite subgraph [12], etc.

Clique and quasi-clique enumeration methods include the classical algorithm [3], the external-memory H^*-graph algorithm [5], redundancy-aware clique enumeration [22], maximum clique computation in MapReduce [24], and optimal quasi-clique mining [19].

Core decomposition and truss decomposition have been studied in various settings, including in-memory algorithms [2, 7, 26], external-memory algorithms [4, 21], and MapReduce algorithm [8]. Sarıyüce et al. [18] proposed the first incremental k-core decomposition algorithms for graph stream. Zhang and Parthasarathy [26] designed an incremental algorithm for updating triangle k-core (equivalent to k-truss) with edge insertions/deletions. In our work, our focus is to update the TCP-Index in a dynamic setting, which relies on the incremental update of the edge trussness. We have also compared with [26] experimentally in updating edge trussness, and our updating method is two to three orders of magnitude faster.

Wang et al. [23] defined a dense neighborhood graph based on the common neighbors. Gibson et al. [12] studied mining dense bipartite subgraphs.

7. CONCLUSIONS

In this paper, we study the online community search problem in a network. We propose a novel k-truss community model based on the k-truss concept which is shown to have cohesive and hierarchical community structure. To support the efficient search of k-truss community, we design a novel and compact tree-shape index, called the TCP-Index, which preserves the edge trussness and the triangle adjacency relationship, and supports community search in linear time with respect to the community size. We further study the k-truss community search in dynamic graphs and propose efficient incremental algorithms to update the index. We conduct extensive experiments on large real-world networks, and the results demonstrate the effectiveness and efficiency of the proposed algorithms.

8. ACKNOWLEDGMENTS

The work is supported by the Hong Kong Research Grants Council (RGC) General Research Fund (GRF) Project No. CUHK 411211, 411310, 418512. Lu Qin is supported by ARC DE140100999.

9. REFERENCES

[1] Y.-Y. Ahn, J. P. Bagrow, and S. Lehmann. Link communities reveal multiscale complexity in networks. *Nature*, 466(7307):761–764, 2010.

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

[3] C. Bron and J. Kerbosch. Finding all cliques of an undirected graph (algorithm 457). *Commun. ACM*, 16(9):575–576, 1973.

[4] J. Cheng, Y. Ke, S. Chu, and M. T. Özsu. Efficient core decomposition in massive networks. In *ICDE*, pages 51–62, 2011.

[5] J. Cheng, Y. Ke, A. W.-C. Fu, J. X. Yu, and L. Zhu. Finding maximal cliques in massive networks by h*-graph. In *SIGMOD*, pages 447–458, 2010.

[6] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, 14(1):210–223, 1985.

[7] J. Cohen. Trusses: Cohesive subgraphs for social network analysis. Technical report, National Security Agency, 2008.

[8] J. Cohen. Graph twiddling in a mapreduce world. *Computing in Science and Engineering*, 11(4):29–41, 2009.

[9] W. Cui, Y. Xiao, H. Wang, Y. Lu, and W. Wang. Online search of overlapping communities. In *SIGMOD*, pages 277–288, 2013.

[10] J. Edachery, A. Sen, and F. J. Brandenburg. Graph clustering using distance-k cliques. In *Proceedings of the 7th International Symposium on Graph Drawing*, pages 98–106, 1999.

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[12] D. Gibson, R. Kumar, and A. Tomkins. Discovering large dense subgraphs in massive graphs. In *VLDB*, pages 721–732, 2005.

[13] E. Hartuv and R. Shamir. A clustering algorithm based on graph connectivity. *Information Processing Letters*, 76(4–6):175–181, 2000.

[14] J. J. McAuley and J. Leskovec. Learning to discover social circles in ego networks. In *NIPS*, pages 548–556, 2012.

[15] M. E. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113, 2004.

[16] G. Palla, I. Derényi, I. Farkas, and T. Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *Nature*, 435(7043):814–818, 2005.

[17] M. Rosvall and C. T. Bergstrom. Maps of random walks on complex networks reveal community structure. *Proceedings of the National Academy of Sciences*, 105(4):1118–1123, 2008.

[18] A. E. Sarıyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek. Streaming algorithms for k-core decomposition. *PVLDB*, 6(6):433–444, 2013.

[19] C. E. Tsourakakis, F. Bonchi, A. Gionis, F. Gullo, and M. A. Tsiarli. Denser than the densest subgraph: Extracting optimal quasi-cliques with quality guarantees. In *KDD*, pages 104–112, 2013.

[20] J. Ugander, L. Backstrom, C. Marlow, and J. Kleinberg. Structural diversity in social contagion. *Proceedings of the National Academy of Sciences*, 109(16):5962–5966, 2012.

[21] J. Wang and J. Cheng. Truss decomposition in massive networks. *PVLDB*, 5(9):812–823, 2012.

[22] J. Wang, J. Cheng, and A. W.-C. Fu. Redundancy-aware maximal cliques. In *KDD*, pages 122–130, 2013.

[23] N. Wang, J. Zhang, K.-L. Tan, and A. K. H. Tung. On triangulation-based dense neighborhood graphs discovery. *PVLDB*, 4(2):58–68, 2010.

[24] J. Xiang, C. Guo, and A. Aboulnaga. Scalable maximum clique computation using mapreduce. In *ICDE*, pages 74–85, 2013.

[25] 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.

[26] Y. Zhang and S. Parthasarathy. Extracting analyzing and visualizing triangle k-core motifs within networks. In *ICDE*, pages 1049–1060, 2012.