Data Mining and Knowledge Discovery manuscript No.
(will be inserted by the editor)

Fast Truss Community Query in Large-scale Dynamic Graphs

Zheng Lu · Qing Cao

Received: date / Accepted: date

Abstract Recently, there has been significant interest in the study of the community search problem in social and information networks: given one or more query nodes, find densely connected communities containing the query nodes. However, most existing algorithms require linear computational time to the size of the found community for each specific K value. Therefore, state-of-the-art algorithms have limited scalability in large scale graphs, where communities grow to millions of edges.

In this paper, given an undirected graph G and a set of query nodes Q, we study community query using the k-truss based community model. We formulate our problem of finding a connected truss community, as finding a connected k-truss subgraph with all possible k that contains Q. The state-of-art approximation algorithm can achieve this goal with a time complexity of $O(n'm')$ where n' and m' are the size of the result truss community. For queries that only identity and exact size of communities are required, We construct an index structure that can retrieve there information of all connected k-truss communities that contain Q with all possible K values. The algorithm can run in $\sum_{u \in Q} d(u)$, where $d(u)$ is the degree of vertex u. We prove that this is the optimal time complexity for truss community query. Extensive experiments on real-world networks show the effectiveness and efficiency of our algorithms.

Keywords K-truss · dynamic graph · query process

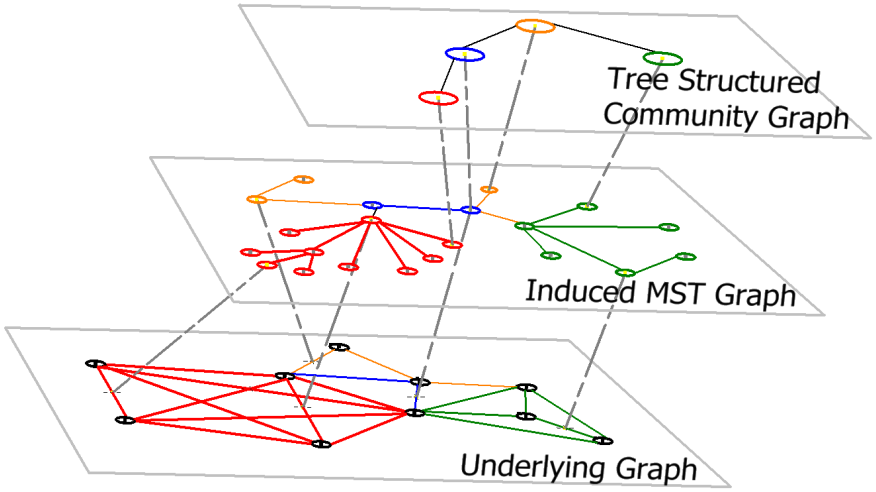

Fig. 1 Two layer index structure for k-truss community queries.

1 Introduction

Community structures naturally exist in many real-world networks such as social, biological, collaboration, and communication networks. The task of community detection is to identify all communities in a network, which is a fundamental and well-studied problem in the literature. Recently, several papers have studied a related but different problem called community search, which is to find the community containing a given set of query nodes. The need for community search naturally arises in many real application scenarios, where one is motivated by the discovery of the communities in which given query nodes participate. Since the communities defined by different nodes in a network may be quite different, community search with query nodes opens up the prospects of user-centered and personalized search, with the potential of the answers being more meaningful to a user Huang et al (2014). As just one example, in a social network, the community formed by a person's high school classmates can be significantly different from the community formed by her family members which in turn can be quite different from the one formed by her colleagues McAuley and Leskovec (2012).

Various community models have been proposed based on different dense subgraph structures such as k-core Sozio and Gionis (2010); Cui et al (2014); Li et al (2015), k-truss Huang et al (2014), quasi-clique Cui et al (2013), weighted

Zheng Lu
Electrical Engineering & Computer Science, University of Tennessee
E-mail: zlu12@vols.utk.edu

Qing Cao
Electrical Engineering & Computer Science, University of Tennessee
E-mail: zlu12@vols.utk.edu

densest subgraph Wu et al (2015), to name a few major examples. Of these, the k-truss as a definition of cohesive subgraph of a graph G, requires that each edge be contained in at least (k - 2) triangles within this subgraph. It is well known that most of real-world social networks are triangle-based, which always have high local clustering coefficient. Triangles are known as the fundamental building blocks of networks Wang and Cheng (2012). In a social network, a triangle indicates two friends have a common friend, which shows a strong and stable relationship among three friends. Intuitively, the more common friends two people have, the stronger their relationship. In a k-truss, each pair of friends is "'endorsed"' by at least (k - 2) common friends. Thus, a k-truss with a large value of k signifies strong inner-connections between members of the subgraph. Huang et al. Huang et al (2014) proposed a community model based on the notion of k-truss as follows. Given one query node q and a parameter k, a k-truss community containing q is a maximal k-truss containing q, in which each edge is "'triangle connected"' with other edges. Triangle connectivity is strictly stronger than connectivity. The k-truss community model works well to find all overlapping communities containing a query node q. We extended this model for the case of multiple query nodes.

As the time complexity of the state-of-art approximation algorithm can calculate the connected truss community containing all query vertices with the largest k with a time complexity of $O(n'm')$ where n' and m' are the size of the result truss community. Although the linear time complexity for retrieve the whole community is optimal, these algorithms have limited scalability in large scale graphs, where communities grow to millions of edges. In many applications, such as query if a set of users are involved in same community and how cohesive is the community, only the infomation, such as the identity, the k and the size, of the community are required rather than the detailed community itself. For these quries, We construct an index structure that can retrieve information of all connected k-truss communities that contain Q with all possible K values. The algorithm can run in $\sum_{u \in Q} d_u$, where d_u is the degree of vertex u. We prove that this is the optimal time complexity for truss community query. Note that if further details of found truss communities are required, our index structure can also retrieve the exact community with linear time complexity to the community size which is also the optimal time complexity. Figure 1 shows an example of our two level index structures: the induced MST graph and the tree-structured community graph.

The rest of this paper is organized as follows. In Section 6 we show previous works on community search and detection. Section 2 provides notations and definitions used in this paper. We explain the two layered index structure for truss community search in Section 3. Section 4 discusses index update algorithm for dynamic graphs. The evaluations of our algorithm are in Section 5. We conclude our work in Section 7.

Note 1 add more application Durmaz et al (2017)

2 Preliminaries

In our problem, we consider an undirected, unweighted graph $G = (V, E)$. The number of vertices is denoted as $n = |V|$ and number of edges is denoted as $m = |E|$. If the graph is weighted, we use w_u and w_e to denote the weight of vertex u and edge e. We define the set of neighbors of a vertex v in G as $N_v = u \in V : (v, u) \in E$, and the degree of v as $d_v = |N_v|$. We define a triangle in G as 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 we define several key concepts in this paper as follows.

Definition 1 (Edge support) *The support of an edge $e_{u,v} \in E$ is defined as $s_{e,G} = |\triangle_{uvw} : w \in V|$. We denote it as s_e when the context is clear.*

Definition 2 (Trussness) *The trussness of a subgraph $H \in G$ is the minimum support of edges in H plus 2, denoted by $\tau_H = min\{(s_{e,H} + 2) : e \in E_H\}$. The trussness of an edge e is defined as: $\tau_e = max_{H \in G}\{\tau_H : e \in E_H\}$.*

Definition 3 (K-truss subgraph) *Given a graph G and $k \geq 2$, $H \subseteq G$ is a k-truss if $\forall e \in E_H, s_{e,H} \geq (k-2)$.*

Definition 4 (Maximal k-truss subgraph) *H is a maximal k-truss subgraph if it is not a subgraph of another k-truss subgraph with same trussness k in G.*

We use the same triangle adjacency and triangle connectivity definition as in Huang et al (2014) listed below.

Definition 5 (Triangle adjacency) *$\triangle_1$, $\triangle_2$ are adjacent if they share a common edge, i.e., $\triangle_1 \cap \triangle_2 \neq \emptyset$.*

Definition 6 (Triangle connectivity) *$\triangle_1$, $\triangle_2$ are triangle connected if they can reach each other through a series of adjacent triangles, i.e., for $1 \leq i < n, \triangle_i \cap \triangle_{i+1} \neq \emptyset$.*

Definition 7 (Triangle connected graph) *Two edge e_1, e_2 are triangle connected in a subgraph H if there are two triangle $\triangle_1, \triangle_2$ in H and $e_1 \in \triangle_1, e_2 \in \triangle_2$, either $\triangle_1 = \triangle_2$, or $\triangle_1$ is triangle connected with $\triangle_2$ in H. A graph G is triangle connected if all pairs of edges in G are triangle connected.*

Finally, we define k-truss community based on the definition of k-truss subgraph and triangle connectivity as follows.

Definition 8 (K-truss community) *A k-truss community is a maximal triangle connected k-truss subgraph.*

Figure 2 shows several examples of k-truss communities. The whole example graph is a 3-truss as every edge has support of at least 1. Note that there are 2 separate 4-truss communities in Figure 2 as they are not triangle connected with each other.

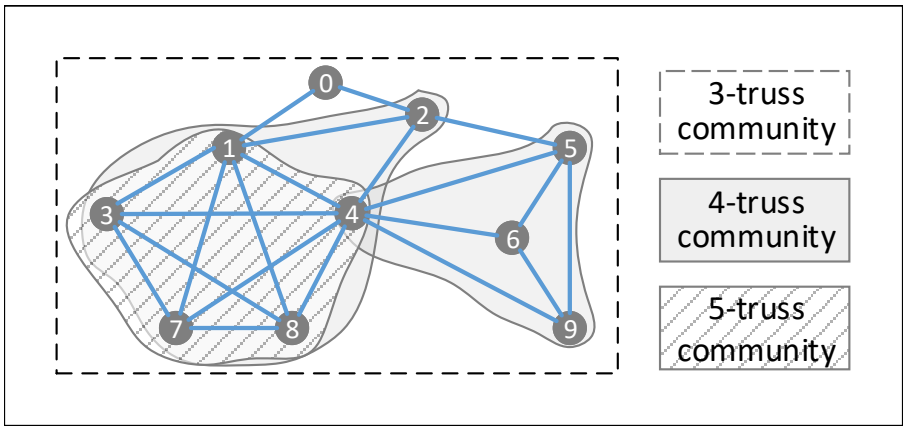

Fig. 2 An example graph for k-truss community

K-truss Community Search The problem of studied in this paper is defined as follows. Given a graph $G(V, E)$, a set of query vertices $Q \in V$, find all truss communities containing Q with maximum k, a specific k or any possible k.

3 Indexed K-truss Community Search

We propose to solve k-truss community search problem using an index based approach. This section describes how to process a k-truss community search query on a static graph, including induced MST graph construction, creating tree-structured community graph, performing various kind of queries on the preprocessed index. In the next section, we describe index update procedure on dynamic graphs.

3.1 Induced MST Graph

We first design an induced MST graph then propose the query algorithm based on it.

Induced MST Graph Construction. We first compute the edge trussness of graph G_o and then construct a new graph G_m, which we called induced MST graph, based on the graph G_o and its edges' trussness. We define the induced MST graph as follows.

Definition 9 (induced MST graph) *The induced MST graph is a weighted maximum spanning forest that each edge e in G_o is represented as a vertex x in G_m. An edge y in G_m represents that the two edges, which are represented by the two adjacent vertices of y, are contained in the same triangle in G_o. The weight of the each vertex in G_m is its represented edge's trussness in G_o. The*

weight of each edge in G_m is the lowest edge trussness of its related triangle's edges in G_o.

We denote G'_m as the graph that is constructed the same way as G_m but with all triangles in G_o as edges, i.e., G_m is the maximum spanning forest of G'_m. We refer to lowest edge trussness of a triangle as the weight of the triangle.

We have the following theorem for vertex weights and edge weights in induced MST graph G_m.

Theorem 1 *In induced MST graph G_m, for each vertex x and each of its adjacent edge y, we have $w_x \geq w_y$.*

Proof According to Definition 9, w_x is the trussness of the represented edge e in G_o while w_y is the lowest trussness of edges in the represented triangle $\triangle$ in G_o. We have $\tau_e \geq \tau_\triangle$, therefore, $w_x \geq w_y$.

Algorithm 1: Induced MST graph construction

Data: $G_o(V_o, E_o)$, edge trussness $\{\tau_e, e \in E_o\}$
Result: $inducedMSTgraphG_m(V_m, E_m)$

$visited \leftarrow \emptyset$;
for $(u, v) \in E_o$ **do**
  suppose u is the lower degree end of (u, v);
  $V_m \leftarrow V_m \bigcup \{(u, v), \tau_{(u,v)}\}$;
  for $w \in N_u$ **do**
    if $(v, w) \in E_o$ ***and*** $\triangle_{uvw} \notin visited$ **then**
      $visited \leftarrow visited \bigcup \triangle_{uvw}$;
      $\tau_{\triangle_{uvw}} = min(\tau_{(u,v)}, \tau_{(u,w)}, \tau_{(v,w)})$;
      $V_m \leftarrow V_m \bigcup \{(u, w), \tau_{(u,w)}\}$;
      $V_m \leftarrow V_m \bigcup \{(v, w), \tau_{(v,w)}\}$;
      $E_m \leftarrow E_m \bigcup \{((u, v), (u, w)), \tau_{\triangle_{uvw}}\}$;
      $E_m \leftarrow E_m \bigcup \{((u, v), (v, w)), \tau_{\triangle_{uvw}}\}$;
      $E_m \leftarrow E_m \bigcup \{((u, w), (v, w)), \tau_{\triangle_{uvw}}\}$;
    end
  end
end
run Kruskal's algorithm on G_m;
return G_m

The truss decomposition algorithm Wang and Cheng (2012) is used to compute trussness of all edges $\{\tau_e, e \in E_o\}$ in G_o. Although it is possible to directly compute k-truss communities based on edge trussness with BFS traversals, such an algorithm suffers from high time complexity for redundant edge access Huang et al (2014). algorithm 1 uses both G_o and edge trussness as inputs to construct the induced MST graph G_m for optimal query time. The algorithm iterates through all edges of G_o and create a vertex in G_m for each edge (u, v) in G_o with weight $\tau_{(u,v)}$. Then for each unvisited neighbor triangle $\triangle_{uvw}$

of edge (u, v), the algorithm creates three edges $((u, v), (u, w))$, $((u, v), (v, w))$ and $((u, w), (v, w))$ in G_m with same weight $\tau_{\triangle_{uvw}} = min(\tau_{(u,v)}, \tau_{(u,w)}, \tau_{(v,w)})$. After that, one can simply run Kruskal's algorithm to get the maximum spanning forest.

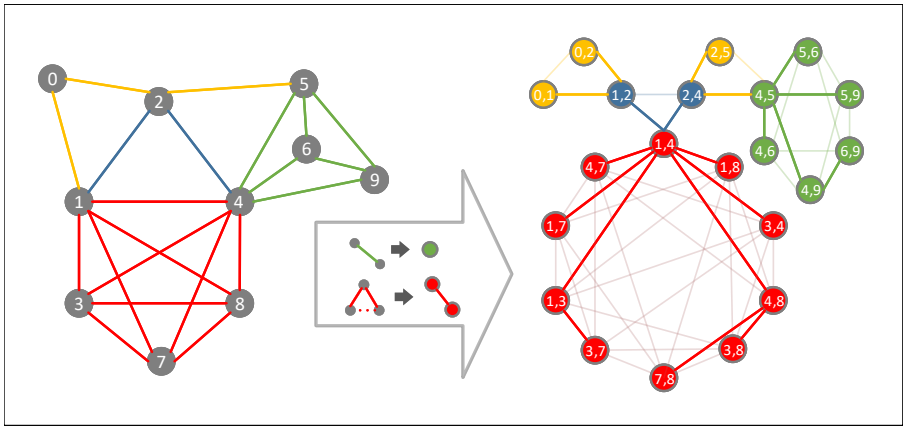

Fig. 3 An example induced MST graph of the example graph in Figure 2

We show an example of induced MST graph in Figure 3. We outline the induced MST graph of the example graph in Figure 2 with bold lines. The rest lines are edges that are generated by algorithm 1 but discarded by Kruskal's algorithm.

Note 2 some possible error of time complexity in Huang et al (2014) The time and space complexity for computation of edge trussness of G_o are $O(\sum_{(u,v)\in E_o} min\{d_u, d_v\})$ and $O(m)$ respectively Huang et al (2014). Listing all the triangles in G_o takes $O(\sum_{(u,v)\in E_o} min\{d_u, d_v\})$ time and $O(\sum_{(u,v)\in E_o} min\{d_u, d_v\})$ space. Finally, running Kruskal's algorithm takes $O(\sum_{(u,v)\in E_o} min\{d_u, d_v\} logm)$ time. As G_m is a maximum spanning forest, so the induced MST graph index takes $O(|V_m|) = O(m)$ space.

Query on Induced MST Graph. To query the k-truss communities of a query vertex q in G_o, the algorithm iterate through adjacent edges of the vertex q. For each neighbor edge (u, q) that is unvisited by the algorithm, it is marked as a seed edge for a new community C_i. Suppose the edge (u, q) in G_o is represented as a vertex x in G_m, the algorithm starts a BFS/DFS from vertex x in G_m and only expands through edges with weight $\geq k$ to find the connected component CC. Then if finds the represented edge e of each vertex $v \in CC$ and adds e to the community C_i. The union of all communities $A = \bigcup C_i$ is all the k-truss communities the vertex q belongs to.

Theorem 2 *The union of all communities* $\bigcup C_i$ *found by algorithm 2 is the union of all the k-truss communities containing query vertex* q.

Proof According to Definition 9, a vertex x in induced MST graph G_m with weight $w_x \leq k$ means the represented edge e in G_o has trussness $\tau_e \leq k$ and thus can be included in a k-truss community. An edge (x, y) in induced MST graph G_m with weight $w_{(x,y)} \leq k$ means the represented triangle $\triangle$ in G_o has all three edges with trussness higher or equal to k and thus the triangle is included in a k-truss community containing all three edges of it. Adjacent edges in G_m means adjacent triangles in G_o and connected components in G_m means triangle connected components in G_o. So, BFS/DFS search starts with a seed vertex x with weight constraint will find the maximal connected component including x which representing the k-truss community that e belongs to in G_o (x represents e in G_m). Therefore, performing such BFS/DFS searches on each edge of the query vertex will find all the k-truss communities that the query vertex belongs to.

Algorithm 2: Query on induced MST graph

Data: $G_o(V_o, E_o)$, $G_m(V_m, E_m)$, an integer k, a query vertex q
Result: a union of all k-truss communities $\bigcup C_i$ containing q

```
1  i ← 0, visited ← ∅;
2  for u ∈ N_q do
3      if (u, q) ∉ visited then
4          find representing vertex x of (u, q) in G_m;
5          CC ← connected component containing x with edges of weight ≥ k;
6          C_i ← ∅;
7          for v ∈ CC do
8              find represented e of v in G_o;
9              visited ← visited ⋃ e;
10             C_i ← C_i ⋃ e;
11         end
12         i ← i + 1;
13     end
14 end
15 return ⋃ C_i
```

Since the query process is performing a BFS on a maximum spanning forest, each query takes $O(|A|)$ time and $O(|A|)$ space, where $|A|$ is the number of edges in A. Although such time complexity is already optimal if the detailed communities are required. We propose a new index structure that can be constructed upon the induced MST graph to further reduce the time complexity if details of k-truss communities are not required.

3.2 Tree-structured Community Graph

We first show how to construct the tree-structured community graph based on induced MST graph. Then we design an algorithm to efficiently query the tree-structured community graph.

Tree-structured Community Graph Construction. A key observation in Cohen (2008) is that, for $k \geq 2$, each k-truss of G_o is the subgraph of a (k-1)-truss of G_o. With this observation, for k-truss communities, we have the following theorem.

Theorem 3 *A k-truss community C_k is the subgraph of a l-truss community C_l, if C_k and C_l are triangle connected and $l < k$. If k-truss community C_k is the subgraph of both l_1-truss community C_{l_1} and l_2-truss community C_{l_2}, then $l_1 \neq l_2$.*

Proof For the first part, since $l < k$, if edges in C_k are triangle connected through triangles with trussness of k, then they are also triangle connected through triangles with trussness of l.

Note 3 do we call it k-truss or l_1-truss. For the second part, suppose $l_1 = l_2$, then edges in C_{l_1} and C_{l_2} are triangle connected through C_k. So $C_{l_1} \bigcup C_{l_2}$ meets the definition of k-truss community (Definition 8) and becomes a larger k-truss community. This contradicts with C_{l_1} and C_{l_2} are k-truss communities themselves, i.e., they are maximal k-truss.

According to Theorem 3, we can build another tree-structured index upon our existing induced MST graph to further facilitate KTruss computation. In this new tree-structured index, we use vertices to represent k-truss communities, i.e., we assign each k-truss community an unique ID and a representing vertex in the new index. If one k-truss community is the subgraqh of another k-truss community, we assign an edge to connect the representing vertices. Each vertex can have a list associated with it including the status of the related k-truss community, such as the trussness of the community, the size of the community, etc. We call this new index the tree-structured community graph and denote it as G_t. For each vetx of G_t, we also have meta data of the represented k-truss communities, e.g., the trussness, the size, etc., stored with it. These meta-data can be gathered very easily through the index construction process. For the ease of query, we build a hash table h that for each edge e in G_o (vertex x in G_m), we record the ID of the k-truss community that includes it with highest order k. We denote such a k-truss community as C^{max}. We have the following theorem for G_t.

Theorem 4 *The tree-structured community graph G_t is a forest.*

Proof First, according to Theorem 3, there is only one ancestor for each k-truss community for . Also, there is no inter level edges according to the definition of maximal KTruss. So, if the graph contains a loop, then a KTruss may contains more than 1 ancestors.

Second, G_t can be disconnected as not all k-truss communities are triangle connected with each other.

algorithm 3 shows the procedure to build the tree-structured community graph G_t. The algorithm uses BFS to traverse the induced MST graphG_m. For

Algorithm 3: Tree-structured community graph Construction

Data: $G_m(V_m, E_m)$
Result: $G_t(V_t, E_t)$, h

$Q \leftarrow \emptyset$, $parent \leftarrow \emptyset$;
while $V_m \neq \emptyset$ **do**
  $seed \leftarrow$ an unvisited vertex in V_m, $Q \leftarrow Q \bigcup seed$;
  while $Q \neq \emptyset$ **do**
    $x = Q.pop()$;
    for $z \in N_x$ **do**
      $Q \leftarrow Q \bigcup z$, $parent[z] \leftarrow x$;
    end
    if $x \in parent$ **then**
      $y \leftarrow parent[x]$, $C_a \leftarrow C_y^{max}$;
      while $\tau_{C_a} > w_{(x,y)}$ **do**
        $C_c \leftarrow C_a$, $C_a \leftarrow$ parent of C_a in G_t;
        if $C_a = \emptyset$ **then**
          $\tau_{C_a} \leftarrow -1$ ▷ Reach the top of the tree.;
        end
      end
      if $\tau_{C_a} < w_{(x,y)}$ **then**
        if $w_{(x,y)} = w_x$ **then**
          create C_x^{max}, $h[x] \leftarrow C_x^{max}$;
          $C_x^{max}.parent \leftarrow C_a$, $C_c.parent \leftarrow C_x^{max}$;
        else
          create C_x^{max}, $h[x] \leftarrow C_x^{max}$;
          create $C_{(x,y)}$, $C_{(x,y)}.parent \leftarrow C_a$;
          $C_c.parent \leftarrow C_{(x,y)}$;
          $C_x^{max}.parent \leftarrow C_{(x,y)}$;
        end
      else
        if $w_{(x,y)} = w_x$ **then**
          $h[x] \leftarrow C_a$;
        else
          create C_x^{max}, $h[x] \leftarrow C_x^{max}$;
          $C_x^{max}.parent \leftarrow C_a$;
        end
      end
    else
      create C_x^{max}, $h[x] \leftarrow C_x^{max}$;
      $V_t \leftarrow V_t \bigcup C_x^{max}$;
    end
    remove x from V_m;
  end
end
return $G_t(V_t, E_t)$, h

each vertex x, if it does not have a parent vertex in the BFS traversal, then the algorithm uses it as a seed vertex to create a new index tree. Otherwise it is combined to the same index tree $T \in G_t$ as its parent vertex y. According to Theorem 1, we have the following equation.

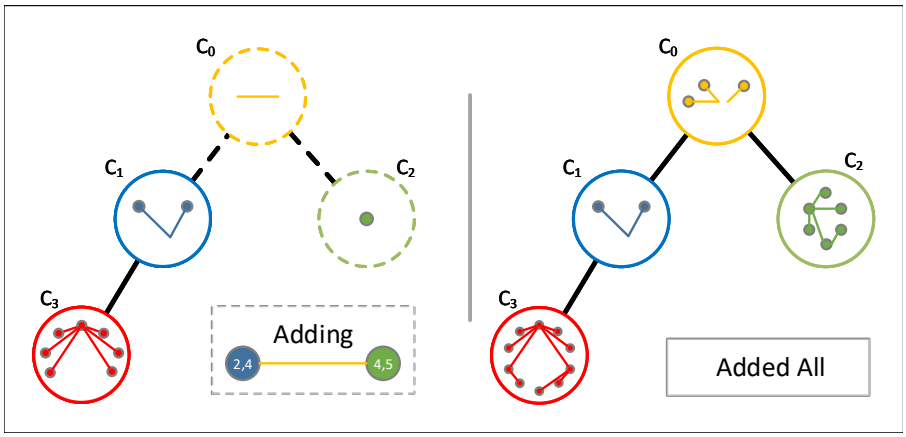

Fig. 4 An example tree-structured community graph of the induced MST graph in Figure 3

$$w_x \geq w_{(x,y)}, w_y \geq w_{(x,y)} \tag{1}$$

An example is shown in Figure 4

Theorem 5 *For a vertex x and its neighbor vertex y in induced MST graph G_m, if their representing edges in G_o are contained in the same k-truss community with trussness of k, then $k \leq w_{(x,y)}$.*

Proof Since G_m is the maximum spanning forest, it has the cycle property, i.e., for any cycle in G'_m, if the weight of an edge in the cycle is smaller than the individual weights of all the other edges in the cycle, then this edge cannot belong to a maximum spanning forest. So there is no path in G'_m between x and y that has all edges with weight $> w_{(x,y)}$. Suppose x and y representing e_x and e_y in G_o, this means that e_x is not triangle connected to e_y through edges in G_o with trussness $> w_{(x,y)}$. Therefore, it is not possible for e_x and e_y to exist in the same k-truss community with $k > w_{(x,y)}$.

Having a parent y in the BFS search only means that the vertex x can be combined to the current index tree T. We still have a problem to solve: On which part of T should the algorithm add the vertex x? According to Theorem 5 and Equation 1, the algorithm needs to backtrack T from C_y^{max} to find an ancestor vertex C_a that meets $\tau_{C_a} \leq w_{(x,y)}$ and use it as the merge point of x. We refer to the index vertices $C_y^{max}, ..., C_i, ..., C_a$ as the backtrack branch for vertex x in T and denote it as B.

Once the algorithm has found C_a, it needs to check the relations of τ_{C_a}, $w_{(x,y)}$ and w_x to decide how to merge vertex x to T. Note that they follow $\tau_{C_a} \leq w_{(x,y)} \leq w_x$, so we have 4 cases shown in algorithm 3. As long as $\tau_{C_a} \neq w_x$, we create a new index vertex C_x^{max} with trussness $\tau_{C_x^{max}} = w_x$. If $\tau_{C_a} < w_{(x,y)} < w_x$, we also create a new index vertex $C_{(x,y)}$ with trussness

$\tau_{C_{(x,y)}} = w_{(x,y)}$. Then we adjust the tree structure of T with new index vertices. Finally, we update the hash table to record in which index vertex x is.

For each vertex of G_m, the backtrack procedure takes $O(k_{max})$ time, where k_{max} is the highest trussness of any k-truss community in G_o. Since the index construction process is a BFS on a maximum spanning tree, the tree-structured community graph construction algorithm takes $O(k_{max}m)$ time. As each vertex in G_t represents a k-truss community in G_o, and G_t is a forest. The algorithm takes $O(m)$ space and the index size is also $O(m)$ space. Although in practice, the size of G_t is much smaller than $O(m)$.

Query on Tree-structured Community Graph. Tree-structured community graph supports three basic types of k-truss community queries of a single query vertex q as listed below.

- K-truss query: Given a vertex q and an integer k, find the k-truss community that contains q.
- Max-k-truss query: Given a vertex q, find the k-truss community with highest possible trussness that contains q.
- Any-k-truss query: Given a vertex q, find all the k-truss communities that contains q.

Max-k-truss query is naturally supported by simply looking up the hash table h and comparing trussness of $h[x_e]$ for each neighbor edge. We show the queries process algorithms for k-truss query and any-k-truss query in algorithm 4. A common operation used in both query algorithms is what we called backtrack branch search, which is defined in Definition 10 below. We can see that if a specific k is provided, the backtrack branch search will stop once the trussness falls below k. On the other hand, if no k is provided, a value of 0 is used and the search will reach the root of the tree.

Definition 10 (Backtrack branch search) *Given a vertex $C_0 \in G_t$ and an integer k, the backtrack branch search returns a list of vertices $C_0, ..., C_i, ...$ that C_{i+1} is the parent vertex of C_i in G_t and any vertex C_i meets $\tau_{C_i} \geq k$. We refer to the searching results $C_0, ..., C_i, ...$ as backtrack branch and denote it as B.*

Tree-structured community graph also supports all three types of queries when the input is a set of query vertices Q. The query process algorithms simply takes intersections of the query results of each individual query vertex for k-truss queries and any-k-truss queries. For max-k-truss queries, the query process algorithm needs to calculate the least common ancestors in G_t of the results of each individual query vertex.

For single vertex queries, the time complexity is $O(d_q)$ for max-k-truss queries and $O(\sum_{e \in N_q} \tau_{h[x_e]})$ for k-truss and any-k-truss queries. The space complexity is $O(1)$ for max-k-truss queries, $O(d_q)$ for k-truss queries and $\sum_{e \in N_q} \tau_{h[x_e]}$ for any-k-truss queries. For multiple vertices max-k-ktruss queries,

Algorithm 4: Query on Tree-structured community graph

Data: $G_o(V_o, E_o)$, $G_t(V_t, E_t)$, the hash table h, a query vertex q or a set of query vertices Q, [an integer k]
Result: a set of k-truss community IDs R

```
1  function branch_search (C ∈ G_t, G_t, [k = 0])
2      B ← ∅;
3      while C ≠ ∅ and τ_C ≥ k do
4          B ← B ⋃ C;
5          C ← C.parent;
6      end
7      return B
8  end
9  function query_k (q, G_o, G_t, k)
10     R ← ∅;
11     for e ∈ N_q do
12         B ← branch_search (h[x_e], G_t, k);
13         if τ_{B[−1]} = k then
14             R ← R ⋃ B[−1];                    ▷ B[−1] is the last element in B
15         end
16     end
17     return R
18 end
19 function query_anyk (q, G_o, G_t)
20     R ← ∅;
21     for e ∈ N_q do
22         B ← branch_search (h[x_e], G_t);
23         R ← R ⋃ B;
24     end
25     return R
26 end
```

since the least common ancestor computation takes $O(H)$[1] time, where H is the height of the tree. The query time is $O(\sum_{q\in Q}(\max_{e\in N_q} \tau_{h[x_e]} + d_q))$ and the space is $O(\max_{q\in Q} \max_{e\in N_q} \tau_{h[x_e]})$. Multiple vertices k-truss queries take $O(\sum_{q\in Q} \sum_{e\in N_q} \tau_{h[x_e]})$ time and $O(\max_{q\in Q} d_q)$ space. For multiple vertices any-k-truss queries, the time and space complexity is $O(\sum_{q\in Q} \sum_{e\in N_q} \tau_{h[x_e]})$ and $O(\max_{q\in Q} \sum_{e\in N_q} \tau_{h[x_e]})$ respectively.

4 Dynamic updates

This section describes update procedures of both induced MST graph and tree-structured community graph in dynamic graphs. We focus on edge insertion/deletion as vertex insert/deletion can be represented by inserting/deleting an isolated vertex and several following edge insertions/deletions.

The scope of affected edges when a new edge is inserted/deleted has been well studied in Huang et al (2014). When an edge e_0 has been added/removed

[1] $O(H)$ is for simple online algorithm, off-line algorithms can achieve time complexity of $O(1)$ Bender and Farach-Colton (2000).

from G_o, the affected edges, i.e., with trussness changes, are either directly forming new triangles of weight $\geq \tau_{e_0}$ or are triangle connected to the new edge e_0 with all triangles of weight $\geq \tau_{e_0}$. With this rule, the authors of Huang et al (2014) also developed an efficient algorithm to calculate edge trussness updates.

We can then proceed to update the induced MST graph G_m once edge trussness in G_o have been updated. For the insertion of an edge e_0, a new vertex x_0 is added to G_m^{prime} with several new adjacent edges and updated weight of some vertices and edges. All we need to do is maintain the maximum spanning tree G_m with these changes in G_m^{prime}. As maintaining Minimum spanning tree is a well studied area with highly efficient algorithms Cattaneo et al (2002), we omit the details here.

Note 4 are there problems if we dont update vertex and edges in order? Once we have updated G_m, we can proceed to update the tree-structured community graph G_t. There are two types of changes for the update of G_m. One is vertex based that includes adding the new vertex x_0 and updating weights of other affected vertices. For a vertex x that have updated weight, according to Theorem 1, the algorithm needs to create a new child vertex of current vertex $h[x]$ in G_t based on the new weight. Note that this update is incomplete in the sense that we haven't taken into account the weight changes of the adjacent edges.

Algorithm 5: Zipper (Combine two branches)

Data: An inserted/updated edge $(x, y) \in G_m$
Result: None

$C_x \leftarrow h[x]$, $C_y \leftarrow h[y]$;
$k \leftarrow w_{(x,y)}$;
while $\tau_{C_x} > k$ **do** $C_x \leftarrow C_x.parent$;
while $\tau_{C_y} > k$ **do** $C_y \leftarrow C_y.parent$;
$C \leftarrow \emptyset$ ▷ Assume $\tau_\emptyset = -1$.;
while $C_x \neq \emptyset$ ***or*** $C_y \neq \emptyset$ **do**
  if $\tau_{C_x} > \tau_{C_y}$ **then**
    if $C \neq \emptyset$ **then** $C.parent \leftarrow C_x$;
    $C \leftarrow C_x$, $C_x \leftarrow C_x.parent$;
  else if $\tau_{C_x} < \tau_{C_y}$ **then**
    if $C \neq \emptyset$ **then** $C.parent \leftarrow C_y$;
    $C \leftarrow C_y$, $C_y \leftarrow C_y.parent$;
  else
    $C_c \leftarrow$ combine (C_x, C_y);
    if $C \neq \emptyset$ **then** $C.parent \leftarrow C_c$;
    $C \leftarrow C_c$, $C_x \leftarrow C_x.parent$, $C_y \leftarrow C_y.parent$;
  end
end

Another type of change is that a new edge is added to G_m or an edge has updated weight. This happens when a new triangle is formed in G_o or the

lowest trussness of edges in a triangle has changed. For a new edge (x, y) in G_m, if x and y belongs to different connected components in G_m, then we need to combine the two branches of trees to which $h[x]$ and $h[y]$ belongs. We call this procedure zipper as shown in algorithm 5. In this way, K-Truss communities with trussness lower than the weight of the edge are combined together and the rest of high trussness communities become two separate children branch. An example is shown in Figure 5.

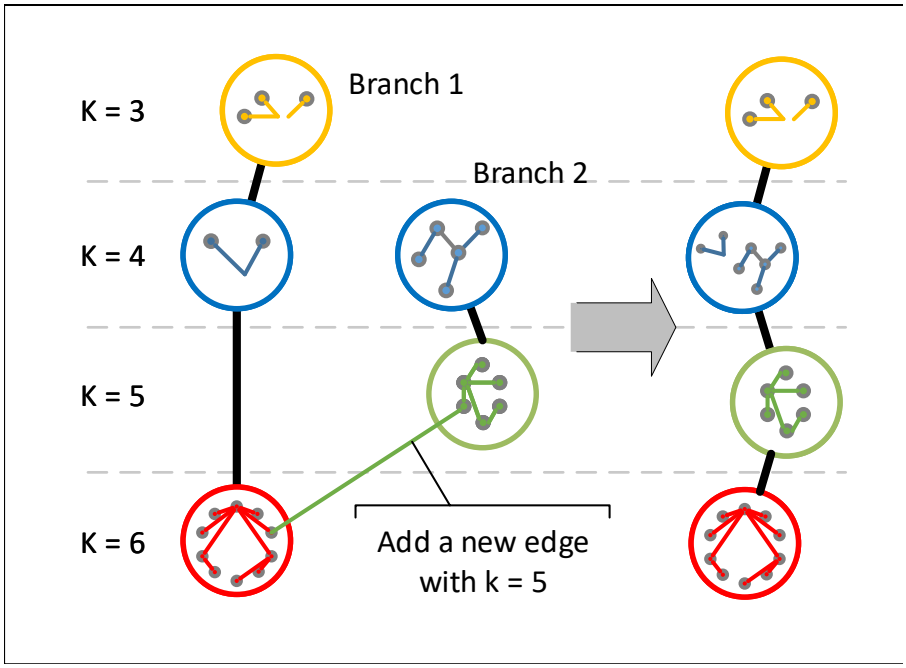

Fig. 5 An example or zipper operation when adding a new edge.

Note 5 add a theorem here? However, if a new edge is inserted in G_m without join two disconnected parts, another edge with lower weight in G_m must have been deleted to maintain the minimum spanning tree. In this case, if the two ends of the new edge $h[x]$ and $h[y]$ is different, we can still use zipper procedure to combine these two branches. The only difference is that we need to apply the zipper procedure from the least common ancestor of $h[x]$ and $h[y]$. Same goes for an edge with updated weight, as it can be viewed as deleting the edge with old weight and inserting a new edge with updated weight.

The update procedure for edge deletion in G_o follows a similar process. It involves a vertex removal and vertex weight decreasing as well as several edge deletion and edge weight decreasing in G_m. Similarly, in G_t, we use a procedure to 'unzip' the branch of a deleted edge into two separate branches or a partially separated branch.

Table 1 Datasets

Dataset	Type	$\|V_{wcc}\|$	$\|E_{wcc}\|$	$\overline{\sigma}$
Wiki	Communication	2.4M	4.7M	3.9
Skitter	Internet	1.7M	11.1M	5.07
Livejournal	Social	4.8M	43.4M	5.6
Hollywood	Collaboration	1.1M	56.3M	3.83
Orkut	Social	3M	117M	4.21
Sinaweibo	Social	58.7M	261.3M	4.15
Webuk	Web	39.3M	796.4M	7.45
Friendster	Social	65M	1.8B	5.03

Datasets with the number of vertices and edges in the largest weakly connected components, and the average shortest distance $\overline{\sigma}$ of 100,000 vertex pairs.

When using the zipper procedure to combine two branches (or divide one branch), we may have performance issues when updating the hashtable h. As one vertex change in G_t may involve several entries to be updated in h. So how do we solve this problem....

Note 6 add time complexity analysis

5 Evaluations

In this section, we show the results of experimental evaluation of truss community query on our index structure.

5.1 Datasets

We evaluate our algorithm on 8 graphs from different disciplines as shown in table 1. All graphs are complex networks that have power-law degree distributions and relatively small diameters. To simplify our experiments, we treat them as undirected, un-weighted graphs and only use the largest weakly connected component of each graph. All datasets are collected from Snap Leskovec and Krevl (2014).

5.2 Experiment settings

We evaluate our algorithms, we use a Cloudlab Ricci et al (2014) c8220 server with two 10-core 2.2GHz E5-2660 processors and 256GB memory. We base our algorithm on Snap Leskovec and Krevl (2014). All algorithms are implemented in C++.

5.3 Query time

We first compare the query time of truss community search based on our index structure with the state-of-art related work.

5.4 Index construction time and size

We show in this section that our index achieves the similar construction time and size as previous work.

5.5 Index update time

The index update time upon graph changes, i.e. edge deletion, is shown in ...

6 Related Works

Our work is most related to the inspiring work Huang et al (2014) which introduce the notion of k-truss community based on triangle connectivity. An index structure call TCP is proposed in Huang et al (2014) that each vertex holds their maximum spanning forest based on edge trussness of their ego-network. Triangle connected k-truss communities mitigate the "free-rider" issue but at the cost of slow computation efficiency especially for vertices belongs to large k-truss communities so that they are not able to meet requirements of queries in some cases. We have use this work as a comparison in section section 5.

The notion of triangle connected k-truss community is also referred to as $k-(2,3)$ nucleus in Sariyüce and Pinar (2016) where they propose an approach based on disjoint-set forest to speed up the process of nuclues decomposition. We are more emphasize on the fast query process rather speed up the truss decomposition. Also the tree-structured community graph is easier to maintain for dynamic graphs.

Our work falls in the category of cohesive subgraph mining Koujaku et al (2016); Sozio and Gionis (2010); Cui et al (2014); Li et al (2015); Cui et al (2013); McAuley and Leskovec (2012), such as clique Bron and Kerbosch (1973); Rossi et al (2014), k-core Cheng et al (2011); Shin et al (2016), k-truss Huang et al (2014); Wang and Cheng (2012); Cohen (2008); Huang et al (2015, 2016) and quasi-clique Tsourakakis et al (2013). An α-adjacency γ-quasi-k-clique model is introduced by Cui et al (2014) for online searching of overlapping communities. ρ-dense core is a pseudo clique recently introduced by Koujaku et al (2016) that is able to deliver optimal solution for graph partition problems. The pattern of k-core structures is studied by Shin et al (2016) for applications such as finding anomalies in real world graphs, approximate degeneracy of large-scale graphs and so on. K-truss decomposition is also studied in Huang et al (2016) for probabilistic graphs.

7 Conclusion

In this paper, we describe ...

References

Bender MA, Farach-Colton M (2000) The lca problem revisited. In: LATIN 2000: Theoretical Informatics, Springer

Bron C, Kerbosch J (1973) Algorithm 457: finding all cliques of an undirected graph. Communications of the ACM 16(9):575–577

Cattaneo G, Faruolo P, Petrillo UF, Italiano GF (2002) Maintaining dynamic minimum spanning trees: An experimental study. In: Workshop on Algorithm Engineering and Experimentation, Springer, pp 111–125

Cheng J, Ke Y, Chu S, Özsu MT (2011) Efficient core decomposition in massive networks. In: Data Engineering (ICDE), 2011 IEEE 27th International Conference on, IEEE, pp 51–62

Cohen J (2008) Trusses: Cohesive subgraphs for social network analysis. National Security Agency Technical Report 16

Cui W, Xiao Y, Wang H, Lu Y, Wang W (2013) Online search of overlapping communities. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, ACM, pp 277–288

Cui W, Xiao Y, Wang H, Wang W (2014) Local search of communities in large graphs. In: Proceedings of the 2014 ACM SIGMOD international conference on Management of data, ACM, pp 991–1002

Durmaz A, Henderson TA, Brubaker D, Bebek G (2017) Frequent subgraph mining of personalized signaling pathway networks groups patients with frequently dysregulated disease pathways and predicts prognosis. In: PSB

Huang X, Cheng H, Qin L, Tian W, Yu JX (2014) Querying k-truss community in large and dynamic graphs. In: Proceedings of the 2014 ACM SIGMOD international conference on Management of data, ACM, pp 1311–1322

Huang X, Lakshmanan LV, Yu JX, Cheng H (2015) Approximate closest community search in networks. Proceedings of the VLDB Endowment 9(4):276–287

Huang X, Lu W, Lakshmanan LV (2016) Truss decomposition of probabilistic graphs: Semantics and algorithms. In: Proceedings of the 2016 International Conference on Management of Data, ACM, pp 77–90

Koujaku S, Takigawa I, Kudo M, Imai H (2016) Dense core model for cohesive subgraph discovery. Social Networks 44:143–152

Leskovec J, Krevl A (2014) SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data

Li RH, Qin L, Yu JX, Mao R (2015) Influential community search in large networks. Proceedings of the VLDB Endowment 8(5):509–520

McAuley JJ, Leskovec J (2012) Learning to discover social circles in ego networks. In: NIPS, vol 2012, pp 548–56

Ricci R, Eide E, The CloudLab Team (2014) Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. USENIX ;login: 39(6), URL `https://www.usenix.org/publications/login/dec14/ricci`

Rossi RA, Gleich DF, Gebremedhin AH, Patwary MMA (2014) Fast maximum clique algorithms for large graphs. In: Proceedings of the 23rd International Conference on World Wide Web, ACM, pp 365–366

Sariyüce AE, Pinar A (2016) Fast hierarchy construction for dense subgraphs. Proceedings of the VLDB Endowment 10(3):97–108

Shin K, Eliassi-Rad T, Faloutsos C (2016) Corescope: Graph mining using k-core analysis-patterns, anomalies and algorithms. In: Data Mining (ICDM), 2016 IEEE 16th International Conference on, IEEE, pp 469–478

Sozio M, Gionis A (2010) The community-search problem and how to plan a successful cocktail party. In: Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, pp 939–948

Tsourakakis C, Bonchi F, Gionis A, Gullo F, Tsiarli M (2013) Denser than the densest subgraph: extracting optimal quasi-cliques with quality guarantees. In: Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, pp 104–112

Wang J, Cheng J (2012) Truss decomposition in massive networks. Proceedings of the VLDB Endowment 5(9):812–823

Wu Y, Jin R, Li J, Zhang X (2015) Robust local community detection: on free rider effect and its elimination. Proceedings of the VLDB Endowment 8(7):798–809