2-level Index for Truss Community Query in Large-Scale Graphs

Zheng Lu, Yunhe Feng, Qing Cao
Dept. of Electrical Engineering & Computer Science, *University of Tennessee*, Knoxville, USA
{zlu12, yfeng14, cao}@utk.edu

***Abstract*—Recently, there are significant interests in the study of the community search problem in large-scale graphs. K-truss as a community model has drawn increasing attention in the literature. In this work, we extend our scope from the community search problem to more generalized local community query problems based on triangle connected k-truss community model. We classify local community query into two categories, the community-level query and the edge-level query based on the information required to process a community query. We design a 2-level index structure that stores k-truss community relations in the top level index and preserves triangle connectivity between edges in the bottom level index. The proposed index structure not only can process both community-level queries and edge-level queries in optimal time but also can handle both single-vertex queries and multiple-vertex queries. We conduct extensive experiments using real-world large-scale graphs and compare with state-of-the-art methods for k-truss community search. Results show our method can process community-level queries in the range of hundreds of microseconds to less than a second and edge-level queries from a few seconds to hundreds of seconds for highest degree vertices within large communities.**

***Index Terms*—Query-dependent community detection, K-truss, Large-scale graphs**

I. Introduction

Graphs are naturally used to model many real-world networks, e.g., online social networks, biological networks, collaboration and communication networks. A well studied graph problem, which is called community search [1], [2], [5]–[9], [11], is to find communities with a query vertex and a specific cohesiveness measure. In this paper we are studying a related but more generalized problem. Given a set of query vertices and user defined cohesiveness criteria, find communities that both meet cohesiveness criteria and containing all query vertices. We refer to this problem as local community query because it is query-dependent and the runtime does not rely on the size of the graph. As the community query reveals the community-level relations for a group of vertices, it can be used as a building block for many analytical tasks, such as similarity measurement [12], social recommendation [10], de-anonymization [14].

The normal procedure of community search involves exhaustively discover all the relevant communities, i.e., , enumerate all the edges in each community, leading to excessive computation time/space if edge-level details of communities are not of interest. For example, if one want to use the cohesiveness of common communities among a set of vertices as a similarity measure, it is not necessary to discover all edges in common comminities. Identifiers of common communities and cached statistics of each community, e.g., , cohesiveness measure and size, are sufficient. As such, we can generalize the concept of community search to local community queries, which is to identify common communities among a set of query vertices given cohesiveness creteria. Local community queries may or may not search the edge-level details of each relevant community according to application requirements. To get a better idea, consider the example query ”Given a set of query vertices, do they belong to same communities?” This type of query does not require edge-level details of communities and is good for applications such as studying common interests among a set of users. We refer to local community queries that do not require edge-level details as community-level queries. It is also possible that an application require the details of exactly which edges belong to relevant communities, such as classic community search queries. We refer to those queries as edge-level queries.

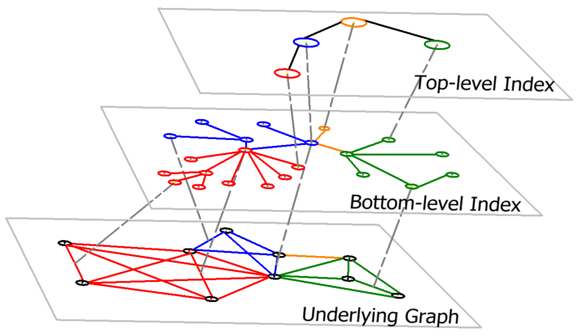

Fig. 1. Two layer index structure for k-truss community queries. The top level index is a super-graph with vertices representing unique k-truss communities and edges representing containment relations between them. The bottom level index is triangle derived graph that translates triangle connectivity in the underlying graph to edge connectivity for fast k-truss community traversal.

To study local community queries, we adopt K-truss community as the community model. K-truss as a definition of cohesive subgraph, requires that each edge be contained in at least $(k-2)$ triangles within this subgraph. [6] introduce the model of k-truss community based on triangle connectivity that ensures the connectivity inside the community. The bounded diameter property of a K-truss community makes it an excellent choice for discovering cohesive and meaningful communities. The low computation cost of k-truss helps to scale to large-scale graphs.

Local community queries can benefit from compact index structures constructed from pre-computed results. Previous works mainly focused on the community search problem of a single query vertex [1], [6]. In this paper, we propose a novel 2-level index structure, to support both the community-level and the edge-level k-truss community query. An overview of our 2-level index is shown in Figure 1. The top level index is a super-graph with vertices representing unique k-truss communities and edges representing containment relations between them. For the bottom level index, we introduce a new type of graph called triangle derived graph that translates triangle connectivity in a graph to edge connectivity for fast k-truss community traversal. We can use simple $union$ and $intersection$ operations on the top level index to locate relevant k-truss communities of a given query to answer community-level queries directly. Once identifiers of relevant communities are found, they can be handed to the bottom level index to processing inner-community details for edge-level queries. For example, given a community search query, we first use the top level index to find target k-truss communities that contain all query vertices and then use the bottom level index to retrieve edges contained in each target k-truss community without expensive triangle enumerations. We proved our index and query process to be theoretically optimal and showed its efficiency in practice for various kinds of community-level and edge-level k-truss community queries on real-world graphs and demonstrate its performance by comparing with state-of-the-art methods, the TCP index [6] and the Equitruss index [1]. For reproducibility, we make the source code available online.[1]

Our contribution can be summarized as follows.

- We generalize community search problem into local community query problem. The generalization lies in three aspects. First, we introducing both community-level queries and edge-level queries that provide different level of details for relevant communities. Second, we support multiple query vertices to enable applications that require community relation among query vertices. Third, we incorporate various cohesiveness criteria instead of a single cohesiveness measurement as used in related works.
- We develop a 2-level index structure that can efficiently process both the community-level and the edge-level k-truss community query for a single query vertex or a set of query vertices with any given cohesiveness creteria with a two step process.
- We perform extensive experiments on our 2-level index on large-scale real-world graphs and show that our index structure outperforms state-of-the-art index structures.

The rest of this paper is organized as follows. Section II provides notations and definitions used in this paper. We design a novel 2-level index structure and its query process in Section III-A. The evaluations of our algorithm are in Section IV. We discuss previous works in Section V and conclude our work in Section VI.

[1]https://github.com/DongCiLu/KTruss

II. Preliminaries

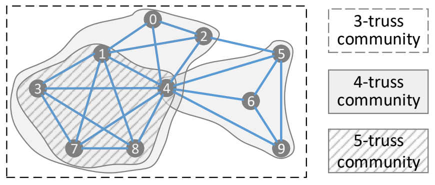

Fig. 2. An example graph with four k-truss communities

In our problem, we consider an undirected, unweighted graph $G = (V, E)$. An example graph is shown in Figure 2. 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 $\triangle_{uvw}$ as a cycle of length 3 with distinct vertices $u, v, w \in V$. Then we define several key concepts 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.

For example, in Figure 2, the edge support for $(0, 2)$ is 2, as it is contained in triangle $(0, 2, 1)$ and triangle $(0, 2, 4)$.

Definition 2 (Trussness): The trussness of a subgraph $G' \in G$ is the minimum support of edges in G' plus 2, denoted by $\tau_{G'} = min\{(s_{e,G'} + 2) : e \in E_{G'}\}$. We then define edge trussness as: $\tau_e = max_{G' \in G}\{\tau_{G'} : e \in E_{G'}\}$.

For example, in Figure 2, subgraph $(1, 3, 7, 8, 4)$ has trussness of 5 as all edges in it have support at least $5 - 2 = 3$. The trussness for edge $(1, 4)$ is 5, as subgraph $(1, 3, 7, 8, 4)$ has trussness of 5.

Definition 3 (K-truss): Given a graph G and $k \geq 2$, $G' \subseteq G$ is a k-truss if $\forall e \in E_{G'}, s_{e,G'} \geq (k - 2)$. G' is a maximal k-truss subgraph if it is not a subgraph of another k-truss subgraph with the same trussness k in G.

Because the K-truss definition does not define the connectivity of the subgraph, we have the following definitions for triangle connectivity.

Definition 4 (Triangle adjacency): $\triangle_1$, $\triangle_2$ are adjacent if they share a common edge, i.e., $\triangle_1 \cap \triangle_2 \neq \emptyset$.

Definition 5 (Triangle connectivity): $\triangle_1$, $\triangle_2$ are triangle connected if they can reach each other through a series of adjacent triangles, i.e., for $1 \leq i < n, \triangle_i \cap \triangle_{i-1} \neq \emptyset$. Two edges e_1, e_2 are triangle connected if $\exists e_1 \in \triangle_1$, $e_2 \in \triangle_2$, $\triangle_1$ and $\triangle_2$ are identical or triangle connected.

For example, in Figure 2, $(1, 4)$ is triangle connected to $(5, 6)$ through a series of adjacent triangles $(1, 2, 4)$, $(2, 4, 5)$ and $(4, 5, 6)$.

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

Definition 6 (K-truss community): A k-truss community is a maximal k-truss subgraph and all its edges are triangle connected.

fig:example shows several examples of k-truss communities. The whole example graph is a 3-truss community as every edge has the support of at least 1 and all edges are triangle connected with each other. Note that there are two separate 4-truss communities in Figure 2. Because the edge support of edge $(2, 5)$ is only 1, it cannot belong to a 4-truss. After excluding edge $(2, 5)$, edges in the two 4-trusses are no longer triangle connected.

III. Design of 2-level Index

In this paper, we aim to solve local k-truss community query problems with a novel 2-level index. We first describe the structure and construction of the 2-level index. Then we show how to use the 2-level index to process local k-truss community queries.

A. Construction of 2-level Index

The index proposed in this paper contains two levels to efficiently process both community-level queries and edge-level queries. The top level index provides information at the community level while the bottom level index offers information at the edge level. The top level is a super-graph, called community graph, whose vertices representing unique k-truss communities and edges representing containment relations between k-truss communities. The bottom level is a maximum spanning forest of a triangle derived graph that preserves the edge level trussness and triangle connectivity inside k-truss communities.

The index is constructed in a bottom-up manner. In the following, we first formally define the triangle derived graph and introduce the algorithm of constructing it from the original graph (Section III-A1). Then we introduce the community graph and show how to use simple graph traversals on the bottom level index, which stores a maximum spanning tree of triangle derived graph, to create the community graph (Section III-A2).

1) Triangle Derived Graph:
The triangle derived graph of an original graph G^o is obtained by associating a vertex with each edge of G^o and connecting two vertices if the corresponding edges of G^o belong to the same triangle. Then we only store a maximum spanning tree of the triangle derived graph, which is enough to save edge level structures of k-truss communities in the original graph. We show the formal definition of the triangle derived graph in Definition 7. We show an example of the triangle derived graph and a maximum spanning forest of it in Figure 3. We outline the maximum spanning forest in Figure 2 with bold lines.

Definition 7 (triangle derived graph): The triangle derived graph G^t is a weighted undirected graph that each edge in the original graph G^o is represented as a vertex in G^t. G^t has an edge e^t connecting vertices $v^t_1, v^t_2 \in G^t$ if and only if their corresponding edges $e^o_1, e^o_2 \in G^o$ belong to the same triangle in G^o. The weight of a vertex in G^t is the trussness of its corresponding edge in G^o. The weight of an edge in G^t is defined as the lowest trussness of edges in the corresponding triangle in G^o. Because of this, the weight of an edge in G^t is always smaller or equal to weights of adjacent vertices.

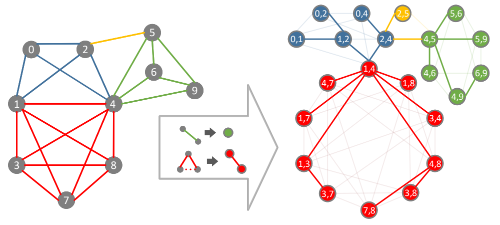

Fig. 3. An example of the triangle derived graph and its maximum spanning tree of the example graph. We show the id of each vertex in the underlying graph on the left. We use a pair of ids of vertices in the underlying graph as the id of a vertex of the triangle derived graph on the right because each vetex in the triangle derived graph represents an edge in the underlying graph.

We use G^m to denote the maximum spanning forest of G^t that has been stored as the bottom level index. To construct G^m, one way is generating the triangle derived graph G^t first and then finding a maximum spanning tree of it. However, this approach is impractical because we need to sort edges in G^t and the number of edges is three times the number of triangles in the original graph G^o, which can be an order of magnitude higher than the number of edges of most real-world graphs. We use two methods to avoid this. First, we find that for real-world graphs, the highest edge trussness is usually small compared to the size of the graph, e.g., only a few thousand for the densest graph in our experiments. So we can use counting sort instead of comparison sort to reduce the time complexity. Second, since edge weight in G^t represents minimum edge trussness in the corresponding triangle, we can sort edges in the original graph G^o to get the sorted order of triangles in G^o which can be translated to sorted order of edges in G^t. These two methods reduce both the time complexity and space complexity of the maximum spanning tree algorithm.

We need edge trussness before constructing bottom-level index, which can be computed using the truss decomposition algorithm [13], as inputs. The time and space complexity for constructing bottom-level index are dominated by the computation of edge trussness of G^o, which are $O(\sum_{(u,v)\in E^o} min\{d_u, d_v\})$ and $O(|E^o|)$, respectively. Since G^m is a maximum spanning forest, the bottom level index takes $O(|V^m|) = O(|E^o|)$ space to store it.

2) Community Graph:
The top level index is a super-graph having vertices representing unique k-truss communities and edges representing containment relations between k-truss communities. We call this index structure the community graph and denote it as G^c. Based on the hierarchical property of k-truss [4], i.e., for $k \geq 2$, each k-truss is the subgraph of a $(k-1)$-truss, we have the formal definition of the community graph in Definition 8. We show an example of the community graph in Figure 4.

Definition 8 (community graph): The community graph G^c is a weighted undirected graph that each k-truss community in the original graph G^o is represented by a vertex in G^c. G^c has an edge e^c connecting vertices $v^c_1, v^c_2 \in G^t$ if and only if

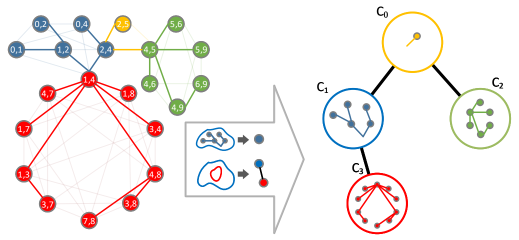

Fig. 4. Construction of the community graph from the triangle derived graph. The graph on the left is a MST of the triangle derived graph. The graph on the right is the community graph having vertices representing unique k-truss communities and edges representing containment relations between k-truss communities. The color of vertices shows the mapping between two graphs.ji

the following two conditions are met for their corresponding k-truss communities $C_1^o, C_2^o \in G^o$:

- C_1^o is a subgraph of C_2^o or the other way around.
- Assume without loss of generality that C_1^o is a subgraph of C_2^o, there is no $C_3^o \in G^o$ such that C_1^o is a subgraph of C_3^o and C_3^o is a subgraph of C_2^o.

G^c only has vertex weights, which are the trussness of the corresponding k-truss communities.

A key property of the community graph is that it is a forest. For the sake of space, we omit the proofs here. This property enable us to easily construct the community graph with a single breath first search (BFS) on the bottom level index G^m. The traversal algorithm create a tree in the community graph G^c of each connected component in G^m. To construct a tree in G^c, the algorithm iteratively processes vertices in G^m using the BFS and map vertices in G^m to vertices in G^c. The map between a vertex $u^m \in G^m$ to a vertex $v^c \in G^c$ means that the edge represented by u^m belongs to the k-truss community represented by v^c. We store this mapping in a lookup table H.

When the traversal algorithm reaches a vertex v_{seed}^m belonging to a new connected component, it first create a new super-vertex in G^c and use it as a start point to build a new tree in G^c. Note that this start point is not necessarily the root of the new tree. After that, for an arbitrary vertex u^m from the same connected component that has been discovered by the BFS, assuming its parent vertex during the search is p^m and p^m has been mapped to v_p^c already, u^m will be mapped to an existing vertex or a new vertex in G^c depending on the relations of the weight of the super-vertex v_p^c and its ancestor in G^c, the weight of the edge (p^m, u^m) and the weight of the vertex u^m. The detailed rule for the mapping can be found in Algorithm 1.

The reason we use weights of the edge (p^m, u^m) between a vertex and its parent vertex as references when mapping a vertex of G^m to G^c is that edges in G^m represents triangle adjacency in the original graph G^o. Since edge weight in G^m represents minimum edge trussniss in the corresponding triangle in G^o and G^m is a maximum spanning tree, the weight of edge (p^m, u^m) limits the highest trussness of a k-truss community u^m's representing edge can belongs to.

Algorithm 1: Top Level Index Construction

```
Data: $G^m(V^m, E^m)$
Result: $G^c(V^c, E^c)$, $H$
for each connected component $CC \in G^m$ do
    $v_{seed}^m \leftarrow CC.pop()$;
    create_super_vertex($seed$, $null$);
    for $u^m \in$ BFS starting at seed do
        $p^m \leftarrow$ parent of $u^m$ in BFS;
        $e^m \leftarrow (u^m, p^m)$;
        $v_p^c \leftarrow H[p^m]$;
        while $\tau_{v_p^c} > \tau_{e^m}$ do  $v_p^{c\prime} \leftarrow v_p^c$,
         $v_p^c \leftarrow v_p^c.parent$ ;
        if $\tau_{v_p^c} < \tau_{e^m}$ then
            if $\tau_{e^m} = \tau_{u^m}$ then
                $v_u^c \leftarrow$ create_super_vertex($u$, $v_p^c$);
                $v_p^{c\prime}.parent \leftarrow v_u^c$;
            else
                $v_e^c \leftarrow$ create_super_vertex($e$, $v_p^c$);
                $v_u^c \leftarrow$ create_super_vertex($u$, $v_e^c$);
                $v_p^{c\prime}.parent \leftarrow v_e^c$;
            end
        else
            if $\tau_{e^m} = \tau_{u^m}$ then
                $H[u^m] \leftarrow v_p^c$;
            else
                $v_u^c \leftarrow$ create_super_vertex($u$, $v_p^c$);
            end
        end
    end
end
return $G^c(V^c, E^c)$, $H$
```

For each vertex of G^m, searching the ancestor super-vertex in G^c of its parent vertex in G^m takes $O(k_{max})$ time, where k_{max} is the highest trussness of k-truss communities in G^o. Since the index construction process is a BFS on a maximum spanning tree with $O(|E^o|)$ vertices, the total construction time is $O(k_{max}|E^o|)$. As each vertex in G^c represents a k-truss community in G^o, and G^c is a forest, the algorithm takes $O(|C^o|)$ space and the index size is $O(|C^o|)$, where $|C^o|$ is number of communities in G^o.

B. Query on 2-level Index

We classify k-truss local community queries into two categories according to the level of information required. The community-level query (Section III-B1) requires only information of relations between k-truss communities and to which k-truss communities query vertices belong. For example, "Do query vertices belongs to same k-truss communities?". This type of queries can be answered solely by the top level index of our 2-level index. Another type of queries, which requires edge level information to process, is called the edge-level query (Section III-B2). For example, the widely studied community search queries. We process this type of queries by first locating the target k-truss communities with the top level index and then diving into the edge-level details with the bottom level index. Besides of the two query categories, we also incorporate the ability to add various cohesiveness criteria for queries with our 2-level index.

1) The Community-level Query:
The 2-level index supports any range of trussness value as cohesiveness criterion for a query. They all support both a single query vertex and a set of query vertices. There are three most common cohesiveness creteria: a specific k value, the maximum k value and any k values. We refer to them as k-truss queries, max-k-truss queries, and any-k-truss queries.

Algorithm 2: $union - intersection$ Algorithm.

```
Data: $G^o(V^o, E^o)$, $G^c(V^c, E^c)$, $H$, $Q$
Result: subgraph $S$ of $G^c$
1  $S \leftarrow \emptyset$;
2  $initialized \leftarrow false$;
3  for $u^o \in Q$ do
4      $SS \leftarrow$ singlev_subgraph($u^o$);
5      if $!init$ then  $S \leftarrow SS$, $init \leftarrow true$ ;
6      else  $S \leftarrow S \cap SS$ ;
7  end
8  return $S$
9  function singlev_subgraph ($u^o$)
10     $SS \leftarrow \emptyset$;
11     for $v^o \in N_u$ do
12         $u^c \leftarrow H[(u^o, v^o)]$;
13         $B \leftarrow$ ancestors of $u^c$ in $G^c$;
14         $SS \leftarrow SS \bigcup B$;
15     end
16     return $SS$
17 end
```

community-level k-truss community queries share a similar querying process on the top level index, called $union - intersection$ procedure. We show the detailed procedure in Algorithm 2. Given the 2-level index and a lookup table H that maps represents edges in the original graph G^o (represented by vertices in G^m) to vertices in the community graph G^c as input, the algorithm first iterate through adjacent edges of each query vertex. For each edge maps to a vertex in G^c, the the algorithm takes the $union$ of it and its ancestors in G^c, to which represent all the communities a query vertex belongs. Then the algorithm take the $intersection$ of the results of all query vertices, to which represents communities that all query vertices belong. Finally, we can easily retrieve communities that have weights of a specified k (k-truss query), calculate communities that have maximum trussness (Max-k-truss query) or return all the vertices (Any-k-truss query).

The time and space complexity for collecting ancestor for a given super-vertex is τ_e. To iterate all the adjacent edges of a query vertex takes $\sum_{v \in N_u} \tau_{(u,v)}$ time and space. Finally, the algorithm needs to find the set of super-vertex for each query vertex to get the set of common super-vertex, so the total time and space complexity for the $union - intersection$ procedure is $\sum_{u \in Q} \sum_{v \in N_u} \tau_{(u,v)}$.

2) The Edge-level Query:
The edge-level k-truss community query requires information of finest granularity as it needs to explore the inner edge-level structure of a k-truss community. Our bottom level index contains the detailed triangle connectivity information that makes such queries possible. To process a k-truss community query, we first locate the target k-truss communities with the top level index and then compute query results using edge-level details provided by the bottom level index.

We use k-truss community search as a concrete example as it is the simplest form of the edge-level k-truss community query. First, the $union - intersection$ algorithm is performed to get target communities of the query. Then for each community in target communities, we collect edges contained in it by gathering the vertex list of subgraphs of G^m stored alongside G^c vertices. Finally, edges of the original graph G^o can be retrieved by converting their corresponding vertices in G^m.

The $union - intersect$ algorithm takes $\sum_{u \in Q} \sum_{v \in N_u} \tau_{(u,v)}$ time and space. Each edge in the target communities will only be accessed exactly once, so the time and space complexity for the search are $\sum_{u \in Q} \sum_{v \in N_u} \tau_{(u,v)} + |\bigcup C_i|$, where $\bigcup C_i$ is the union of target communities.

TABLE I
DATASETS

Dataset	Type	$\|V_{wcc}\|$	$\|E_{wcc}\|$	$\|\triangle_{wcc}\|$	k_{max}
Skitter	Internet	1.7M	11.1M	28.8M	68
Sinaweibo	Social	58.7M	261.3M	213.0M	80
Orkut	Social	3.1M	117.2M	627.6M	78
Bio	biological	42.9K	14.5M	3.6B	799
Hollywood	Collab.	1.1M	56.3M	4.9B	2209

Datasets with the number of vertices, edges, triangles and the maximum trussness (k_{max}) in the largest weakly connected components without self edges. Sorted by the number of triangles.

TABLE II
COMPARISON OF INDEX CONSTRUCTION

Graph	Decomp.	Index Time (Sec.)			Index Size (MB)		
Name	Time (Sec.)	TCP	Equi	Our	TCP	Equi	Our
Skitter	151	167	151	139	139	240	193
Sinaweibo	10169	11048	5724	6871	2744	1390	1810
Orkut	8731	7659	3609	5059	3302	1722	2479
Bio	13496	13964	6223	8874	393	177	289
Hollywood	12619	16620	4154	10182	1929	813	1276

IV. EVALUATIONS

In this section, we evaluate our proposed index structure for various types of local k-truss community queries on real-world networks. We compare the 2-level index with state-of-the-art solutions, the TCP index [6] and the Equitruss index [1] for index construction (Section IV-A), single vertex k-truss community search (Section IV-B1) and multiple vertex k-truss

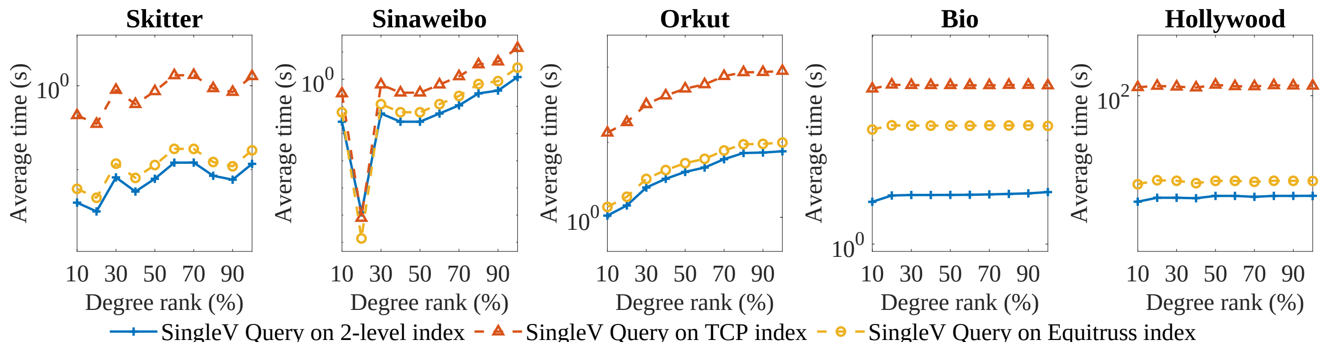

Fig. 5. Comparison of single vertex k-truss community search of the 2-level index, the TCP index and the Equitruss index.

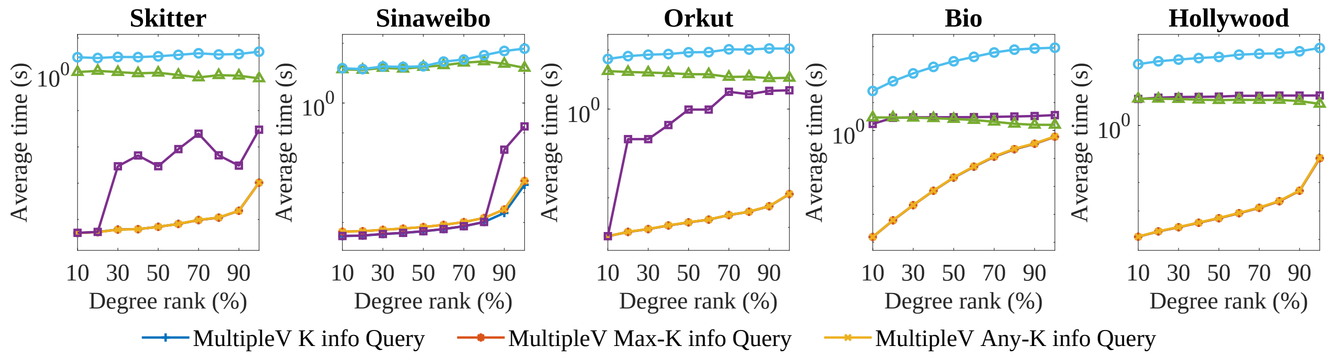

Fig. 6. Three types (k-truss, max-k-truss, any-k-truss) multiple-vertex (3) community-level k-truss community query $vs.$ community search.

community search. All experiments are implemented in C++ and are run on a Linux server with 2.2GHz CPUs and 256GB memory.

We use 5 real-world graphs of different types as shown in the Table I. To simplify our experiments, we treat them as undirected, un-weighted graphs and only use the largest weakly connected component of each graph. We also removed all the self edges in each graph. All datasets are publicly available from Stanford Network Analysis Project[2] and Network Repository[3].

A. *Index construction*

We show in this section the index size and index construction time of the 2-level index compared to the TCP index and the Equitruss index in Table II. We separate the truss decomposition time for all three methods so that the index construction time only shows how long it takes to generate a certain index with edge trussness provided. We can see in Table II that the 2-level index has comparable construction time to the Equitruss index and both are faster than the TCP index. The index size of the 2-level index is smaller than the TCP index as there are no repeating edges stored in the index. However, the Equitruss has the smallest index size since it only stores edge list of the original graph while the 2-level index also preserves the triangle connectivity inside k-truss communities. Note that if only community-level k-truss community queries are processed, the algorithm only needs to retrieve the top level index which has a much smaller size.

[2]snap.stanford.edu

[3]networkrepository.com

B. *Query performance*

In this section, we evaluate the query time of various query types to show the effectiveness of the 2-level index. As k-truss community query time heavily relies on the degree of query vertices, we use a similar procedure as used by [6] to partition vertices to be used in the experiments. Because only vertices with a degree of $k+1$ can appear in a k-truss community with trussness of k. If we partition vertices uniformly by their degree and the graph has highly screwed degree distribution, then vertices in most of the partitions would have a too low degree to appear in a k-truss community. To show the performance of different algorithms on mining community structures in this kind of graphs, we fix the trussness of k-truss community search queries at 10 and discard vertices with degree less than 20. Then we uniformly partition the rest of vertices according to their degrees into 10 categories and at each category, we randomly select 100 sets of query vertices.

1) Single vertex k-truss community search.: We first evaluate the single vertex k-truss community search performance and compare the query time with the TCP index and the

Equitruss index. The results are shown in Figure 5. The 2-level index achieves best average query time for all graphs. It has an order of magnitude speedup compared to the TCP index for all graphs and 5% to 400% speedup compared to the Equitruss index for all graphs. However, the speed up is linear as all three indices have the same time complexity to handle single vertex k-truss community search queries. Note that very low average query time (around 10^{-5} second) means there is no vertex belonging to any k-truss community in that degree rank. The main reason that the 2-level index is faster than the TCP index is the avoidance of the expensive BFS search during query time. The reason for performance differences of the 2-level index and the Equitruss index on various graphs lies in that the super-graph size of 2-level index is much smaller thus easier to locate target communities.

2) The community-level query vs. the edge-level query (community search).: We perform all three basic types, i.e., k-truss, max-k-truss and any-k-truss, of community-level k-truss community queries and perform community search queries on the targeting communities found by community-level queries. We show both single-query-vertex cases and multiple-query-vertex (3 vertices) cases in **??** and Figure 6, respectively.

We can see in both figures that our index is very effective for both community-level queries and community search queries. The average time for community-level queries spans from $1.22x10^{-5}$ second to 0.62 second. The average time for community search queries is typically much higher than community-level queries since it needs to access edge level information, ranging from $2.20x10^{-5}$ to 979.81 seconds depending on the size of target communities. The multi-hundred average run time comes from searching all truss communities that contain a query vertex (any-k-truss query) with a very high degree in the densest graph (bio). The fast query time of community-level queries makes it an excellent candidate for applications that require community-relation information such as whether a set of vertices belong to the same k-truss communities without digging into the details of any k-truss community.

V. Related Works

Our work falls in the category of cohesive subgraph mining [3], [5], [9], [11] such as community detection and community search. It's most related to the inspiring work [6], which introduce the model of k-truss community based on triangle connectivity. 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. To speed up the community search based on this model, an index structure called the TCP index is proposed in [6] that each vertex holds their maximum spanning forest based on edge trussness of their ego-network. Later, the Equitruss index [1] is proposed that use a super-graph based on truss-equivalence as an index to speed up the single vertex k-truss community search. The Equitruss index is similar to our index structure in the sense that it also use a super-graph for the index. However, the vertex in the super-graph of the Equitruss index is a subgraph of a k-truss community while an edge represents triangle connectivity. Our 2-level index contains a more compact super-graph that a vertex contains a k-truss community and edges represent k-truss community containment relations. We have used both TCP index and Equitruss index as comparisons in our evaluation.

VI. Conclusion

In this work, we use information required to process a community query to divide local k-truss community queries into two categories, the community-level query and the edge-level query. We designed 2-level index that stores the community graph in the top level index for locating relevant communities and the triangle derived graph in the bottom level index to preserve the triangle connectivity at the edge level inside each k-truss community. We showed experimentally that our index structure outperformed state-of-the-art methods for processing both community-level queries and edge-level queries with a single query vertex or multiple query vertices.

References

[1] E. Akbas and P. Zhao. Truss-based community search: a truss-equivalence based indexing approach. *Proceedings of the VLDB Endowment*, 10(11):1298–1309, 2017.

[2] N. Barbieri, F. Bonchi, E. Galimberti, and F. Gullo. Efficient and effective community search. *Data Mining and Knowledge Discovery*, 29(5):1406–1433, 2015.

[3] D. Bera, F. Esposito, and M. Pendyala. Maximal labelled-clique and click-biclique problems for networked community detection. In *2018 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2018.

[4] J. Cohen. Trusses: Cohesive subgraphs for social network analysis. *National Security Agency Technical Report*, 16, 2008.

[5] W. Cui, Y. Xiao, H. Wang, and W. Wang. Local search of communities in large graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 991–1002. ACM, 2014.

[6] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu. Querying k-truss community in large and dynamic graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1311–1322. ACM, 2014.

[7] X. Huang, L. V. Lakshmanan, J. X. Yu, and H. Cheng. Approximate closest community search in networks. *Proceedings of the VLDB Endowment*, 9(4):276–287, 2015.

[8] P. Lee and L. V. Lakshmanan. Query-driven maximum quasi-clique search. In *Proceedings of the 2016 SIAM International Conference on Data Mining*, pages 522–530. SIAM, 2016.

[9] R.-H. Li, L. Qin, J. X. Yu, and R. Mao. Influential community search in large networks. *Proceedings of the VLDB Endowment*, 8(5):509–520, 2015.

[10] J. Liu, Q. Lian, L. Fu, and X. Wang. Who to connect to? joint recommendations in cross-layer social networks. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 1295–1303. IEEE, 2018.

[11] M. Sozio and A. Gionis. 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*, pages 939–948. ACM, 2010.

[12] A. Tsitsulin, D. Mottin, P. Karras, and E. Müller. Verse: Versatile graph embeddings from similarity measures. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*, pages 539–548. International World Wide Web Conferences Steering Committee, 2018.

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

[14] X. Wu, Z. Hu, X. Fu, L. Fu, X. Wang, and S. Lu. Social network de-anonymization with overlapping communities: Analysis, algorithm and experiments. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 1151–1159. IEEE, 2018.