Two-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 has been a significant interest 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 problems to a more generalized local community query problem based on a triangle-connected k-truss community model. We classify local community query into two categories, community-level and edge-level query, based on the information required to process a given query. We design a two-level index structure that supports both types of queries with multiple query vertices and arbitrary cohesiveness criteria. We conduct extensive experiments using real-world large-scale graphs and compare with the state-of-the-art methods of k-truss community search. The results show that our method outperforms the state-of-the-art works in the community search problem and community-level problems by a large margin.**

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

I. Introduction

Online social networks have seen rapid growth in recent years and have generated massive data, which has attracted much research effort on developing advanced data mining technologies [3], [6], [10], [11]. Graphs are naturally used to model real-world networks such as social networks. A well-studied graph problem, known as community search [1], [4], [5], [8], is to find communities with a query vertex and a specific cohesiveness measure. In this paper we study a more generalized problem. Given a set of query vertices and user-defined cohesiveness criteria, we want to find community-level relations among all query vertices with or without edge-level details. 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.

The normal procedure of community search involves making an exhaustive discovery of all the relevant communities, i.e., enumerating all the edges in each community, which leads to excessive computation time/space when edge-level details of communities are not pertinent. For example, if one wants to use the cohesiveness of common communities among a set of vertices as a similarity measure, it is not necessary to discover all the edges in common communities. The identifiers of common communities and the 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 are meant to identify common communities among a set of query vertices given the cohesiveness criteria. Depending on application requirements, local community queries may or may not search the edge-level details of each relevant community. To get a better idea, let's consider **[an example application of team formation. We have a number of skilled workers and we want to pick some of these workers to form a team to achieve a certain goal. It is desirable to pick workers that share common interests or are well acquainted so that they can work well together to help achieve the goal. For this type of problem, if we use graph to model the underlying social network and use graph communities to model common interests, all we want to know is whether a set of vertices (workers) belong to some common communities and the cohesiveness of such communities. We don't interest in who else is in those common communities. Similar applications include user similarity (if we use graph community to model similarity among users), tag suggestion (if we use graph communities to model photo tags), etc.]** We refer to local community queries that do not require edge-level details as community-level queries. An application might also require the details regarding exactly which edges belong to relevant communities, such as classic community search queries. We refer to those queries as edge-level queries.

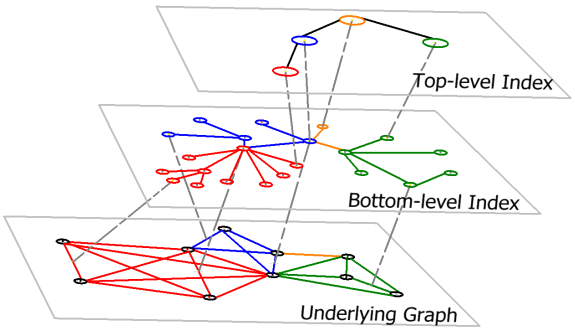

Fig. 1. Two-level index structure for k-truss community queries. The top-level index is a super-graph with the vertices representing unique k-truss communities and the edges representing the containment relations between them. The bottom-level index is a maximum spanning forest of thetriangle-derived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

In this paper, we adopt the k-truss community model based on triangle connectivity introduced by [5]. Previous works were mainly focused on the community search problem of a single query vertex [1], [5]. In this paper, we propose a novel two-level index structure to support both the community-

level and the edge-level local k-truss community queries. An overview of our two-level index is shown in Figure 1. The top-level index is a super-graph whose vertices represent unique k-truss communities and whose edges represent the containment relations between them. For the bottom-level index, we introduce a new type of graph called triangle-derived graph that translates triangle connectivity 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 in a given query, in order to answer community-level queries directly. Once the identifiers of relevant communities are found, they can be handed to the bottom-level index to answer edge-level queries.

Our index also supports queries with arbitrary cohesiveness criteria. Previous works required a specific cohesiveness measurement, e.g., a specific k, to process a query. However, in reality such criteria may not be available. For example, **[if we want to know the common interests among a set of users, what is a good guess of cohesiveness measure, /ie k in k-truss community model, to start with? More practical queries are searching for all communities that contain all the query vertices regardless of their cohesiveness, or searching for communities that contain all the query vertices with highest possible cohesiveness measure.]**

We prove our index and query process to be theoretically optimal, show its practical efficiency for various types of community-level and edge-level local k-truss community queries on real-world graphs and demonstrate its performance by comparing with state-of-the-art methods, the TCP index [5] and the Equitruss index [1]. For reproducibility, we make the source code available online (https://github.com/DongCiLu/KTruss).

Our contribution can be summarized as follows.

- We generalize the community search problem into the local community query problem. The generalization comprises three aspects. First, we introduce both community-level and edge-level queries that provide different level of information for relevant communities. **[For applications that only require community-level information such as team formation or tag suggestion, processing community-level instead of edge-level can avoid a large amount of redundant computation.]** 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. **[Especially, we support queries with non-specific cohesiveness criteria, such as maximum possible cohesiveness or arbitrary cohesiveness, which is quite useful in applications.]**
- We develop a two-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 criteria, using an efficient two step process. **[We proved our two-level index is theoretical optimal for both community-level and edge-level queries.]**
- We perform extensive experiments on our two-level index for large-scale real-world graphs and compare our index structure with the state-of-the-art index structures, Equitruss and TCP-Index. **[Experimental results show that two-level index outperforms the state-of-the-art solutions on community search query and is an order of magnitude faster on community-level query.]**

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

II. Preliminaries

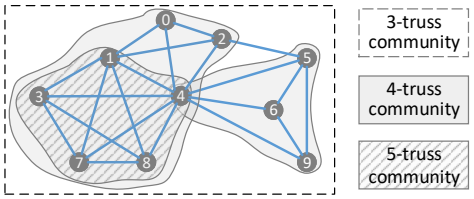

Fig. 2. An example graph with four k-truss communities

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 follow definitions of basic concepts used in [5] and list them below.]**

Definition 1 (Edge support): The support of an edge $e_{u,v} \in E$ is defined as $s_{e,G} = |\triangle_{uvw} : w \in V|$.

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 follows: $\tau_e = max_{G' \in G}\{\tau_{G'} : e \in E_{G'}\}$.

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 of edge $(1, 4)$ is also 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, we use the definition of 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 $(4, 5, 6)$.

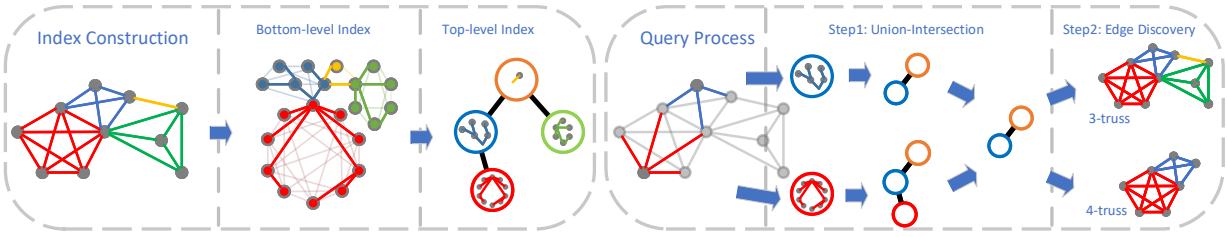

Fig. 3. Index construction and Query process on two-level index

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 with all its edges being triangle connected.

Figure 2 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. Note that there are two separate 4-truss communities. Because $(2, 5)$ has the support of 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 Two-level Index

In this paper, we aim to solve local k-truss community query problems with a novel two-level index. We first describe the structure and construction of the two-level index. Then we show how to process queries with it. **[We show the flow chart of index construction and query process in Figure 3.]**

A. Construction of Two-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 is a super-graph, called community graph, whose vertices represent unique k-truss communities and edges represent the 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 the 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 to construct 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 to create the community graph (Section III-A2).

1) MST of 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 store the edge-level structures of k-truss communities in the original graph. We show the formal definition of a triangle-derived graph in Definition 7. We show an example of the triangle-derived graph and its maximum spanning forest in Figure 4. We outline the maximum spanning forest with bold lines.

Definition 7 (triangle-derived graph): The triangle-derived graph G^t is a weighted undirected graph where each edge in the original graph G^o is represented as a vertex in G^t. G^t has an edge e^t which connect vertices $v_1^t, v_2^t \in G^t$ if and only if their corresponding edges $e_1^o, e_2^o \in G^o$ belong to the same triangle in G^o. The weight of a vertex in G^t is defined as 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. Thus, the weight of an edge in G^t is always smaller or equal to the weights of adjacent vertices.

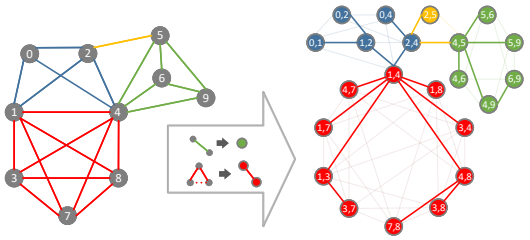

Fig. 4. An example of the triangle-derived graph and its maximum spanning tree of the example graph. We show the id of each vertex in the underlying graph on the left. We use a pair of ids of vertices in the underlying graph as the id of a vertex of the triangle-derived graph on the right.

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 to do this is by 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 the edges in G^t, which can be an order of magnitude larger than the origianl graph G^o. 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. Hence, we can use counting sort instead of comparison sort to reduce the time complexity. Second, since edge weight in G^t represents the minimum edge trussness in the corresponding triangle in G^o, we can sort edges in the original graph G^o to get the sorted order of the triangles in G^o which can be translated to sorted order of edges in G^t. These two methods reduce both the time and space complexities of the maximum spanning tree algorithm.

We need edge trussness before constructing bottom-level index, which can be computed using the truss decomposition algorithm [9], as inputs. The time and space complexities for constructing the 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.

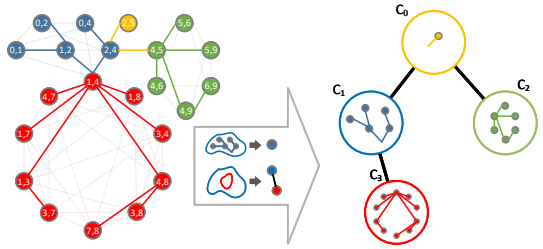

Fig. 5. Construction of the community graph from the triangle-derived graph. The graph on the left is an MST of the triangle-derived graph. The graph on the right is the community graph having vertices that represent unique k-truss communities and edges that represent the containment relations between k-truss communities. The color of vertices shows the mapping between graphs.

2) Community Graph:

The top-level index is a super-graph whose vertices represent unique k-truss communities and edges represent 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 [3], 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 5.

Definition 8 (Community graph): The community graph G^c is a weighted undirected graph that represents each k-truss community in the original graph G^o by a vertex. G^c has an edge e^c connecting vertices $v_1^c, v_2^c \in G^t$ if and only if 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.
- We 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 represent 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 creates 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 the k-truss community represented by v^c has the highest trussness among all k-truss communities that contains the edge represented by u^m. We store this mapping in a lookup table H. For example, in our example graph in Figure 2, edge $1, 4$ belongs to three k-truss communities denoted as C_0, C_1, and C_3 in Figure 5. Since C_3 has the highest trussness of 5, edge $1, 4$ is mapped to C_3.

When the traversal algorithm reaches a vertex v_{seed}^m belonging to a new connected component, it first creates a new super-vertex in G^c and uses it as a starting point to build a new tree in G^c. Note that this starting point is not necessarily the root of the new tree. Thereafter, 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 (which 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 trussness 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, to which u^m's representing edge can belongs.

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 ∈ G^m do
    v^m_seed ← CC.pop();
    create_super_vertex(seed, null);
    for u^m ∈ BFS starting at seed do
        p^m ← parent of u^m in BFS;
        e^m ← (u^m, p^m);
        v^c_p ← H[p^m];
        while τ_{v^c_p} > τ_{e^m} do v^{c'}_p ← v^c_p,
          v^c_p ← v^c_p.parent ;
        if τ_{v^c_p} < τ_{e^m} then
            if τ_{e^m} = τ_{u^m} then
                v^c_u ← create_super_vertex(u, v^c_p);
                v^{c'}_p.parent ← v^c_u;
            else
                v^c_e ← create_super_vertex(e, v^c_p);
                v^c_u ← create_super_vertex(u, v^c_e);
                v^{c'}_p.parent ← v^c_e;
        else
            if τ_{e^m} = τ_{u^m} then
                H[u^m] ← v^c_p;
            else
                v^c_u ← create_super_vertex(u, v^c_p);
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 the number of communities in G^o.

B. Query on Two-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) only requires the information on the relations between k-truss communities and to which k-truss communities the query vertices belong. For example, "Do query vertices belongs to same k-truss communities?". These queries can be answered solely by the top-level index. Another type of queries, which requires edge-level information to process, is called the edge-level query (Section III-B2). The widely studied community search query is the simplest form of such 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 the two query categories, we also incorporate the ability to add various cohesiveness criteria for the queries with our two-level index. We prove our two-level index is theoretical optimal for both community-level and edge-level queries.

1) The Community-level Query:
The query process for community-level local k-truss community query is called the $union - intersection$ procedure. We show the detailed procedure in Algorithm 2. Given the two-level index and a lookup table H that maps 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 iterates through the adjacent edges of each query vertex. For each edge maps to a vertex in G^c, the the algorithm takes the $union$ of itself and its ancestors, discovered by search towards the root of the tree, in G^c. This represents all the communities to which a query vertex belongs. Then, we take the $intersection$ of the results of all query vertices which represents the communities to which all query vertices belong.

The two-level index supports any range of trussness value as a cohesiveness criterion for a query. 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. With the results of the $union-intersection$ procedure, we can easily retrieve communities that have weights of a specified k (k-truss query), find 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 the ancestors of a given super-vertex is τ_e because G^c is a forest. 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-vertices for each query vertex to get a set of common super-vertices, 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 local k-truss community query requires information regarding the 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.

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

```
S ← ∅;
initialized ← false;
for u^o ∈ Q do
    SS ← singlev_subgraph(u^o);
    if !init then S ← SS, init ← true ;
    else S ← S ∩ SS ;
return S
function singlev_subgraph (u^o)
    SS ← ∅;
    for v^o ∈ N_u do
        u^c ← H[(u^o, v^o)];
        B ← ancestors of u^c in G^c;
        SS ← SS ⋃ B;
    return SS
```

TABLE I
DATASETS

Dataset	Type	$\lvert V_{wcc}\rvert$	$\lvert E_{wcc}\rvert$	$\lvert \triangle_{wcc}\rvert$	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

Number of vertices, edges, triangles and the maximum trussness (k_{max}) in the largest WCC.

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

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

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 two-level index with the state-of-the-art solutions: the TCP index [5] and the Equitruss index [1] for index construction (Section IV-A), single vertex k-truss community search (Section IV-B1) and multiple-vertex local k-truss community query (Section IV-B2), respectively.

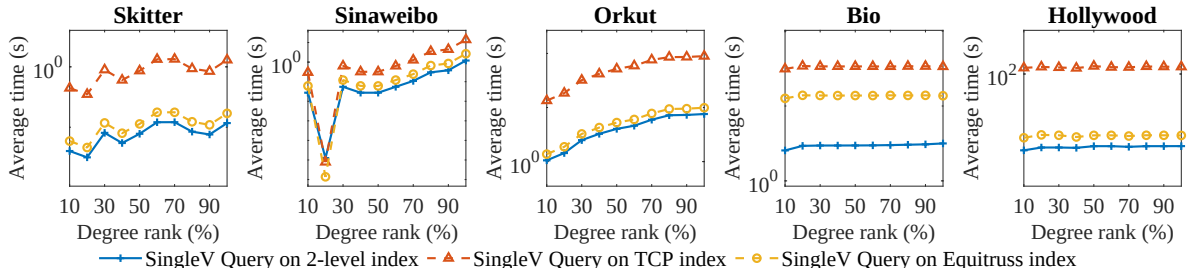

Fig. 6. Comparison of single vertex k-truss community search of the two-level index, the TCP index and the Equitruss index.

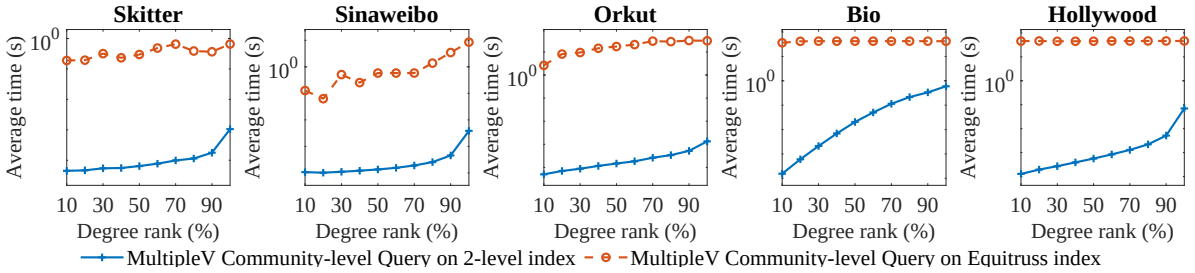

Fig. 7. Comparison of multiple vertex k-truss community-level query of the two-level index and the Equitruss index.

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

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

We use five real-world graphs of different types shown in 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. All datasets are publicly available from the Stanford Network Analysis Project (snap.stanford.edu) and the Network Repository (networkrepository.com).

A. Index construction

In this section, we show the index size and index construction time of the two-level index compared to the TCP index and the Equitruss index in Table II. We can see in Table II that the two-level index has a construction time that is comparable with the Equitruss index and that both are faster than the TCP index. The index size of the two-level index is smaller than the TCP index. However, the Equitruss has the smallest index size since it only stores edge list of the original graph while the two-level index also preserves the triangle connectivity inside k-truss communities.

B. Query performance

In this section, we evaluate the query time of k-truss community queries to show the effectiveness of the two-level index. We perform 10-truss community queries and discard vertices with degrees less than 20 as they normally do not belong to any 10-truss community. We uniformly partition the rest of vertices according to their degrees into 10 buckets and randomly select 100 sets of query vertices for each bucket.

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 6. The two-level index achieves the best average query time for all graphs. It has an order of magnitude speedup compared to the TCP index for all graphs, along with 5% to 400% speedup compared to the Equitruss index for all graphs. The speed up is linear as all three indices have the same time complexity to handle single vertex k-truss community search queries. The main reason why the two-level index is faster than the TCP index is the avoidance of the expensive BFS search during query time. The reason behind the performance differences of the two-level index and the Equitruss index on various graphs lies in the fact that the super-graph size of the two-level index is much smaller, making it easier to locate target communities.

2) Multiple-vertex k-truss community query.: We first perform community-level multiple-vertex k-truss queries with both the two-level index and the Equitruss index. We are not able to perform the same experiment on the TCP index as there is no easy modification that would enable it to support multiple-vertex queries. We can see in Figure 7 that the two-level index outperforms the Equitruss index by several orders of magnitude. This clearly shows the difference between the two indices; the Equitruss index has a larger super-graph and is slower for community-level queries, where only the super-graph is required.

We then perform both the community-level and the edge-level multiple-vertex queries with all three basic types of

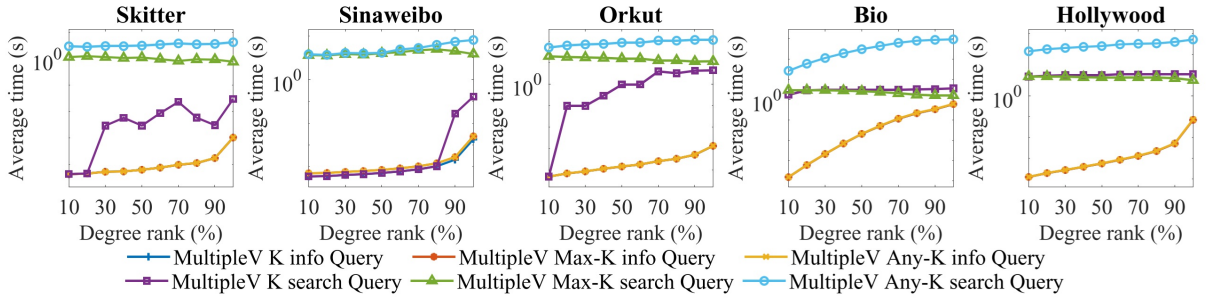

Fig. 8. Three types (k-truss, max-k-truss, any-k-truss) multiple-vertex (3) community-level k-truss community queries $vs.$ community search.

cohesiveness criteria, i.e., k-truss, max-k-truss and any-k-truss. We show the results in Figure 8. The average time for community-level queries spans from 3.4 x 10^{-5} second to 0.06 seconds. Typically, the average time for community search queries is much higher than community-level queries (since one needs to access edge-level information), ranging from 0.03 to 91.9 seconds depending on the size of the target communities. Among all the three types of cohesiveness criteria, any-k-truss edge-level queries usually take the longest query time.

V. Related Works

Our work falls in the category of cohesive subgraph mining [2], [4], [8] such as community detection and community search. It is closely related to an inspiring work [5], which introduce the model of k-truss community based on triangle connectivity. **[The k-truss concept is first introduced by the work [3]. However, the original definition of a k-truss lacks the connectivity constraint so that a k-truss may be a unconnected subgraph. The notion of triangle connected k-truss communities is also referred to as $k-(2,3)$ nucleus in [7] where they propose an approach based on the disjoint-set forest to speed up the process of nucleus decomposition.]** Due to expensive triangle enumeration, triangle-connected k-truss communities have slow computation efficiency, especially for vertices belonging 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 [5], where each vertex holds its maximum spanning forest based on the edge trussness of their ego-network.

The Equitruss index [1] uses 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 uses 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 the triangle connectivity. Our two-level index contains a more compact super-graph in which a vertex contains a k-truss community and edges represent the k-truss community containment relations. **[The difference in the size of the super-graph leads to different performance for various local k-truss community queries, especially for the community-level queries.]** We have used both TCP index and Equitruss index as comparisons in our evaluation.

VI. Conclusion

[In this work, we designed an two level index structure to solve local community query problem, which is a generalized version of the community search problem. We showed that various types of local community queries can be efficiently processed on our index with a two step procedure, and our procedure is theoretically optimal. We conducted extensive experiments of both the community-level and the edge-level local k-truss community queries on real-world graphs. Results show that our index structure outperforms the state-of-the-art method, Equitruss and TCP-Index.]

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] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] A. E. Sariyüce, C. Seshadhri, A. Pinar, and Ü. V. Çatalyürek. Nucleus decompositions for identifying hierarchy of dense subgraphs. *ACM Transactions on the Web (TWEB)*, 11(3):16, 2017.

[8] 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.

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

[10] J. Wang, C. Jiang, S. Guan, L. Xu, and Y. Ren. Big data driven similarity based u-model for online social networks. In *GLOBECOM 2017-2017 IEEE Global Communications Conference*, pages 1–6. IEEE, 2017.

[11] 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.