Noname manuscript No.
(will be inserted by the editor)

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

Zheng Lu · Yunhe Feng · Qing Cao

Received: date / Accepted: date

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.

Keywords Query-dependent community detection · K-truss · Large-scale graphs

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

Yunhe Feng
Electrical Engineering & Computer Science, University of Tennessee
E-mail: yfeng14@vols.utk.edu

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

1 Introduction

Graphs are naturally used to model many real-world networks, e.g., online social networks, biological networks, collaboration and communication networks. As community structures are commonly found in real-world networks, community related problems have been widely studied in the literature, such as community detection (Newman and Girvan (2004); Xie et al. (2013)) and community search (Akbas and Zhao (2017); Barbieri et al. (2015); Cui et al. (2014); Huang et al. (2014, 2015); Lee and Lakshmanan (2016); Li et al. (2015); Sozio and Gionis (2010)), and have found a wide range of applications (Durmaz et al. (2017); Yin and Shi (2017); Zong et al. (2015)). Triangles are known as fundamental building blocks of networks. K-truss as a definition of cohesive subgraph based on triangles of a graph, requires that each edge be contained in at least $(k-2)$ triangles within this subgraph. The low computation cost of k-truss makes it suitable to scale to large-scale graphs. The original definition of a k-truss lacks the connectivity constraint so that a k-truss may be an unconnected subgraph. Huang et al. (2014) introduce the model of k-truss community based on triangle connectivity.

Community search, a query-dependent variant of community detection, attracts more attention as it enables targeted community discovery around given seed vertices of interest and is faster to process in the sense that the runtime does not depend on the size of the graph. However, all the relevant communities still have to be exhaustively identified, leading to excessive computation time/space if details of communities are not of interest. There are various types of queries that are useful in real-world applications involving communities that do not require details of communities. We can classify local community queries, such as community search, into two categories according to the level of information required to answer a query, the community-level query and the edge-level query. The community-level query requires only relation information between different communities of interest. For example, "Do query vertices belongs to the same communities?", "What is the level of cohesiveness among all query vertices?". It is possible to process this type of queries by only examining relations between relavant communities to which query vertices belong without diving into inside structures of them. Another type of queries, the edge-level query, requires edge level information to process, for example, the widely studied community search problem or finding boundaries of a target community. One has to know edge level structures inside relevant communities to be able to answer such queries.

Local community queries, such as community search, based on the k-truss community model (Huang et al. (2014)) can benefit from compact index structures constructed from pre-computed results due to the low computational cost of the k-truss community model. Previous works mainly focused on community search problem of a single query vertex (Akbas and Zhao (2017); Huang et al. (2014)). 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 represent unique k-truss communities and edges represent containment relations between k-truss communities. 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 store a maximum spanning forest of the triangle derived graph generated from the

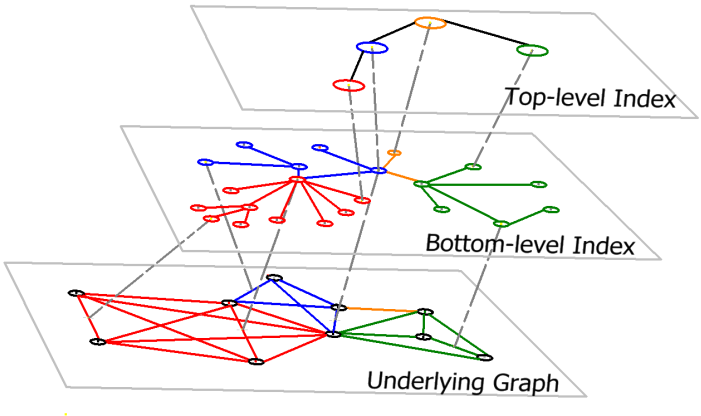


Fig. 1 Two layer index structure for k-truss community queries.

underlying graph that preserves the detailed edge level structure of k-truss communities in the bottom level index. The super-graph forms a forest, and we can use simple *union* and *intersection* operations to locate relevant k-truss communities of a given query. To handle local k-truss community queries, we can use simple *union* and *intersection* operations on the top level index to efficiently locate target communities of a query. These communities can be used to answer community-level queries directly or handed to the bottom level index to processing inner-community details for edge-level queries. The bottom level index is only used for edge-level queries that using edge level information to further process relevant communities provided by the top level index. For example, in a community search query, we can 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. The 2-level index proposed in this paper can efficiently process both single-vertex queries and multiple-vertex queries. We proved our index and query process to be theoretically optimal and showed its efficiency in practice for both 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 (Huang et al. (2014)) and the Equitruss index (Akbas and Zhao (2017)). For reproducibility, we make the source code available online.[1]

Our contribution can be summarized as follows.

- We categorize the local k-truss community queries into the community-level query and the edge-level query based on the information required to answer each type of query.
- We develop a 2-level index structure that can efficiently process both the community-level and the edge-level k-truss community query. The top level index contains a super-graph for locating target communities of a given query. The bottom level index preserves the edge level triangle connectivity for detailed search of inner-community structures.
- We perform extensive experiments on our 2-level index on large-scale real-world graphs and compare it with state-of-the-art index structures. We can process

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

community-level queries in the range of hundreds of microseconds to less than a second. We can process edge-level queries in the range of few seconds to hundreds of seconds for highest degree vertices within large communities.

The rest of this paper is organized as follows. Section 2 provides notations and definitions used in this paper. We design a novel 2-level index structure in Section 3. Section 4 discusses the query process on the proposed index structure. The evaluations of our algorithm are in Section 5. We discuss previous works in Section 6 and conclude our work in Section 7.

2 Preliminaries

In our problem, we consider an undirected, unweighted graph $G = (V, E)$. 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 $\triangle_{uvw}$ 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.*

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

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 edge 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.*

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.*

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 with each other. Note that there are two separate 4-truss communities in Figure 2 as they are not triangle connected with each other.

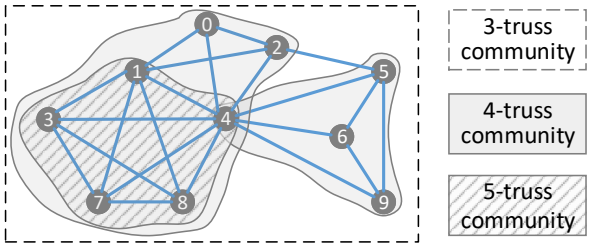


Fig. 2 An example graph with four k-truss communities

3 2-level Index

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

The index proposed in this paper contains two levels. 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 represent unique k-truss communities and edges represent 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. An overview of the index is shown in Figure 1.

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 3.1). Then we introduce the community graph and show how to use simple graph traversals on the triangle derived graph to create the community graph (Section 3.2). Finally, we show the structure we used for the 2-level index to combine the triangle derived graph and the community graph (Section 3.3).

3.1 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 to save edge level structures of k-truss communities in the original graph. By using a maximum spanning forest of the triangle derived graph as the bottom level index, we can use minimum edge enumerations to replace computational expensive triangle enumerations at query time. We show the formal definition of the triangle derived graph in Definition 7.

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 $v_1^t, v_2^t \in G^t$ 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 the trussness of its corresponding edge in G^o. The weight of an edge in G^t is the lowest trussness of edges in the corresponding triangle in G^o.

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.

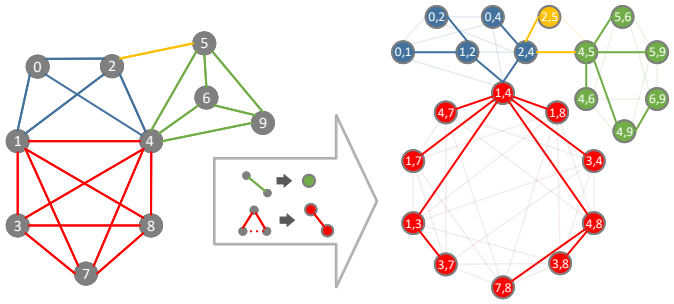


Fig. 3 An example of the triangle derived graph and its maximum spanning tree of the example graph

We have the following theorem of vertex weights and edge weights in the triangle derived graph.

Theorem 1 *In triangle derived graph G^m, for a vertex u and an adjacent edge e, we have $w_u \geq w_e$.*

Proof According to Definition 7, w_u is the trussness of the corresponding edge in G_o while w_e is the lowest trussness of edges in the corresponding triangle in G_o.

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 of 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, as the weight of an edge of G^t is the least trussness of edges in the corresponding triangle of G^o, we can sort edges of G^o to generate edges of G^t in sorted order with reduced space complexity. The details of the algorithm are shown in algorithm 1. The algorithm takes G^o and its edge trussness, which can be computed using the truss decomposition algorithm (Wang and Cheng (2012)), as inputs.

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 $O(|E^o|)$, respectively. Sorting all the edges with

Algorithm 1: Bottom Level Index Construction

Data: $G^o(V^o, E^o)$, edge trussness $\{\tau_e, e \in E^o\}$
Result: The triangle derived graph $G^m(V^m, E^m)$

sorted ← sort edge trussness in decreasing order;
for $e \in E^o$ **do**
  $V^m \leftarrow V^m \bigcup \{e, \tau_e\}$;
  MAKE-SET(e);
end
for $(u, v) \in$ *sorted* **do**
  suppose u is the lower degree end of (u, v);
  for $w \in N_u$ **do**
    if $(v, w) \in E^o$ ***and*** $\tau_{u,v} < \tau_{u,w}$ ***and*** $\tau_{u,v} < \tau_{v,w}$ **then**
      ▷ Compare edge id if trussness of edges are equal to avoid duplication.;
      $\tau_\triangle = min(\tau_{(u,v)}, \tau_{(u,w)}, \tau_{(v,w)})$;
      $E^\triangle \leftarrow \{((u, v), (u, w)), \tau_\triangle\} \bigcup \{((u, v), (v, w)), \tau_\triangle\} \bigcup \{((u, w), (v, w)), \tau_\triangle\}$;
      for $e^\triangle \in E^\triangle$ **do**
        if *FIND-SET*($e^\triangle.v_1$) $\neq$ *FIND-SET*($e^\triangle.v_2$) **then**
          $E^m \leftarrow E^m \bigcup e^\triangle$;
          UNION($e^\triangle.v_1$, $e^\triangle.v_2$);
        end
      end
    end
  end
end
return $G^m(V^m, E^m)$

counting sort costs $O(|E^o| + k_{max})$ time and $O(|E^o| + k_{max})$ space. The time and space complexity for listing all the triangles in G^o is $O(\sum_{(u,v)\in E^o} min\{d_u, d_v\})$ and $O(1)$, respectively. So the time and space complexity for generating the bottom level index when $k_{max} \ll |E^o|$ is $O(\sum_{(u,v)\in E^o} min\{d_u, d_v\})$ 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.

3.2 Community Graph

The top level index is extracted from k-truss communities and their relations in the original graph G^o for fast locating relevant k-truss communities at query time. It is a super-graph having 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 (Cohen (2008)), 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.

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 $v_1^c, v_2^c \in G^t$ 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.*
- *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.*

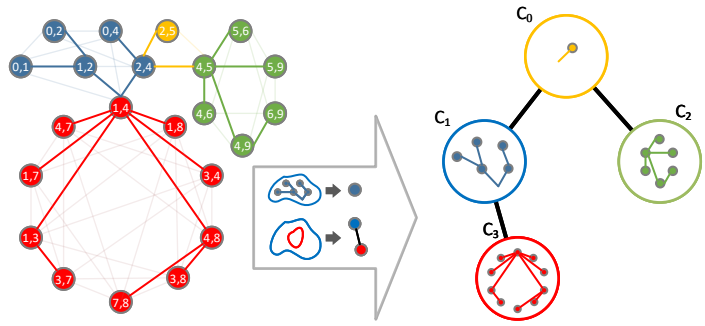


Fig. 4 Overall structure of the 2-level index.

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.

Theorem 2 *The community graph G^c is a forest.*

Proof First, if there is a cycle $v_1^c...v_i^c...v_n^c$ in the community graph G^c and their corresponding k-truss community in the original graph G^o is $C_1^o...C_i^o...C_n^o$, respectively. According to Definition 6 and Definition 8, $C_1^o...C_i^o...C_n^o$ are triangle connected, and it is impossible for an arbitrary pair of community to have same trussness, as it contradicts with the maximal property of k-truss communities. Assume without loss of generality that vertex v_i^c has the largest weight in this cycle, i.e., its corresponding k-truss community C_i^o has the smallest trussness. We denote the two adjacent vertices of it as v_j^c and v_k^c. The corresponding k-truss communities are C_j^o and C_k^o, respectively. By Definition 8, C_j^o and C_k^o are triangle connected. Assume without loss of generality that C_k^o has smaller trussness than C_j^o, we have C_i^o is a subgraph of C_j^o and C_j^o is a subgraph of C_k^o. So the edge between v_i^c and v_k^c contradict with the definition of the community graph.

Second, G^c may have multiple connected components as not all k-truss communities in G^o are triangle connected.

The community graph can be easily constructed with a single BFS traversal on the triangle derived graph G^m. For each connected component in G^m, the algorithm create a tree in the community graph G^c. The algorithm also creates a lookup table H to map each vertex in G^m to the super vertex in G^c to which it belongs. algorithm 2 shows the detailed procedure.

To construct each tree in G^c with a connected component in G^m, the algorithm iteratively processes vertices during the BFS traversal and tracks the parent vertex p^m of each vertex u^m. To process a vertex u^m, the algorithm first compares weights, which are the trussness of corresponding edges in original graph G^o, of u^m to its parent vertex p^m. The algorithm retrieves the super vertex v_p^c using lookup table H, and find an ancestor super vertex with weight, which is trussness of the corresponding k-truss community in the original graph, smaller than the weight of u^m if necessary. Depends on the relations of the weight of the found super vertex,

Algorithm 2: 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;
            end
        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);
            end
        end
    end
end
return G^c(V^c, E^c), H
function create_super_vertex (u^m, v^c_p)
    create v^c_u, H[u^m] ← v^c_u;
    v^c_u.parent ← v^c_p;
    V^c ← V^c ⋃ v^c_u;
    return C_u
end
```

the weight of the edge (p^m, u^m) and the weight of vertex u^m, the algorithm decides whether to create a new super vertex in G^c or assign u^m to an existing super vertex.

The reason we use the weight of the edge between a vertex and its parent vertex as the reference when adding a vertex of G^m to G^c can be explained by Theorem 3.

Theorem 3 *For a vertex u^m and its neighbor vertex v^m in the triangle derived graph G^m, if their representing edges in G^o belong to the same k-truss community with the trussness of k, then $k \leq \tau_{(u^m,v^m)}$.*

Proof Since G^m is the maximum spanning forest, it has the cycle property, i.e., for any cycle in the triangle derived graph G^t, 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^t between u^m and v^m that has all edges with weight higher than $\tau_{(u^m,v^m)}$. Let e^o_u and e^o_v be

their corresponding edges in original graph G_o, then e_u^o is not triangle connected to e_v^o in G^o with edges that have trussness greater than $\tau_{(u^m,v^m)}$. Therefore, it is not possible for e_u^o and e_v^o to exist in the same k-truss community with trussness greater than $\tau_{(u^m,v^m)}$.

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 community graph construction algorithm takes $O(k_{max}|E^o|)$ time. As each vertex in G^c represents a k-truss community in G^o, and G^c is a forest. The algorithm takes $O(|E^o|)$ space and the index size is $O(|E^o|)$.

3.3 2-level IndexStructure

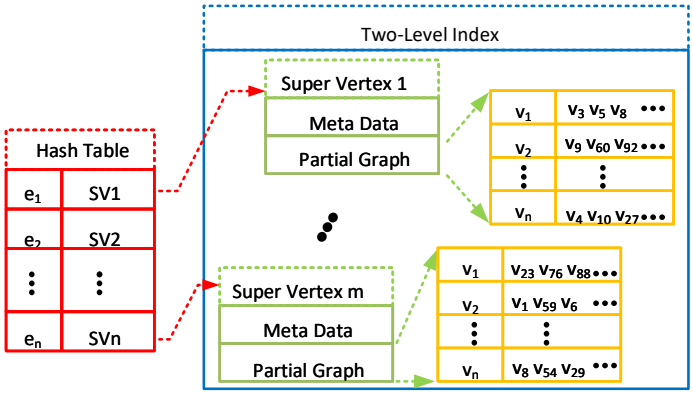


Fig. 5 Overall structure of the 2-level index.

We combine the triangle derived graph and the community graph to form the 2-level index and arrange it in a structure that can provide information at different granularity. Figure 5 shows an overview of the index structure. On the top level, we use an array to store the community graph. Each entry represents a single super vertex. Super edges are expressed by parent and children pointers. Within each entry, it also stores meta-data of the corresponding k-truss community, e.g., the trussness, the community size, etc, for fast information retrieval. This meta-data can be gathered as a byproduct of index construction process. Finally, each entry contains a pointer of the data structure that stores part of the bottom level index.

The bottom level index, which is the triangle derived graph, is stored as several separated adjacent lists. Each adjacent list contains edges of a single k-truss community that does not belong to any of its children k-truss communities. These adjacent lists can be easily generated as byproducts when constructing community graph using algorithm 2.

Finally, we have a hash table that maps each edge in the original graph to an entry in the top level of the 2-level index. One can also follow the similar fashion used in Akbas and Zhao (2017), which maps each vertex in the original graph to multiple entries in the top level index so that the original graph is not required during query time.

4 Query on 2-level Index

We classify k-truss community related queries into two categories according to the level of information required. The community-level query (Section 4.1) 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?", "To which k-truss communities query vertices belong have the highest trussness?". 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 4.2). 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.

4.1 The Community-level Query

The 2-level indexsupports three basic types of community-level k-truss community queries of a single or a set of query vertices listed below:

- K-truss query: Given a set of vertices Q and an integer k, find all the k-truss communities that contain Q with trussness of k.
- Max-k-truss query: Given a set of vertices Q, find k-truss communities that contain Q with the highest possible trussness.
- Any-k-truss query: Given a set of vertices Q, find all the k-truss communities that contain Q.

All three types of community-level k-truss community queries share a similar querying process on the top level index of the 2-level index, call $union-intersection$ procedure. It first takes the *union* subgraphs of the community graph that their corresponding k-truss communities in the original graph contain edges of a single query vertex; then it calculates the *intersection* of subgraphs of each query vertex. Finally, from the *intersection* of subgraphs of the community graph, we can retrieve vertices that have weights of a specified k (k-truss query), calculate vertices that have maximum weights (Max-k-truss query) or return all the vertices (Any-k-truss query). The corresponding k-truss communities in the original graph are used to answer the query. algorithm 3 shows the detailed steps of the $union-intersection$ procedure.

The *branch_search* in the algorithm 3 is to find all the truss communities that to which an edge in the original graph belongs. The time and space complexity for this step isτ_e. The *singlev_subgraph* iterates all the adjacent edges of a query vertex to discover the subgraph in G^c that contains it. The time and space complexity is $\sum_{v \in N_u} \tau_{(u,v)}$. Finally, the algorithm needs to find subgraph for each query vertex, so the total time and space complexity is $\sum_{u \in Q} \sum_{v \in N_u} \tau_{(u,v)}$.

Algorithm 3: *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 ← ∅;
2  initialized ← false;
3  for u^o ∈ Q do
4  |  SS ← singlev_subgraph(u^o);
5  |  if !initialized then S ← SS, initialized ← true ;
6  |  else S ← S ∩ SS ;
7  end
8  return S
9  function branch_search (u^s)
10 |  B ← ∅;
11 |  while u^c ≠ null do
12 |  |  B ← B ⋃ u^s;
13 |  |  u^c ← u^c.parent;
14 |  end
15 |  return B
16 end
17 function singlev_subgraph (u^o)
18 |  SS ← ∅;
19 |  for v^o ∈ N_u do
20 |  |  u^s ← H[(u^o, v^o)];
21 |  |  B ← branch_search(u^c);
22 |  |  SS ← SS ⋃ B;
23 |  end
24 |  return SS
25 end
```

4.2 The Edge-level Query

The edge-level k-truss community query requires information of finest granularity as it needs to explore the inner 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 show three concrete examples of this type of queries in this section: the k-truss community search (Section 4.2.1), the k-truss community boundary search (Section 4.2.2) and the triangle connected maximin path search (Section 4.2.3).

4.2.1 K-truss community search

The k-truss community search 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. algorithm 4 shows the details.

Algorithm 4: K-truss Community Search Query

Data: $G^o(V^o, E^o)$, $G^m(V^m, E^m)$, $G^c(V^c, E^c)$, H, Q
Result: all k-truss communities C containing Q

```
S ← union − intersect(G^o, G^c, H, Q);
for v^c ∈ S do
    C_i ← ∅;
    for v^m ∈ v^c.adj_list.keys do
        find corresponding e^o of v^m; C_i ← C_i ⋃ e^o;
    end
end
return ⋃ C_i
```

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 once, so the time and space complexity for the search are $\sum_{u \in Q} \sum_{v \in N_u} \tau_{(u,v)} + |\bigcup C_i|$.

4.2.2 k-truss community boundary search

The boundary of a k-truss community C_i contains edges in the community that have triangle adjacent neighbor edges belong to other k-truss communities $C_j, C_j \notin C_i$. To find the boundary of a k-truss community, as vertices in the community graph G^c have subgraphs of the triangle derived graph G^m stored in it, we can first gather the subgraph S of G^o that contains children vertices of the corresponding vertex v^c of the queried k-truss community C using the top level index. Then we iterate through all the vertices of the subgraph of G^m that stored in S and select vertices that have neighbors stored in other G^c vertices u^c that $u^c \notin S$. Finally, the collected vertices of G^m can be mapped back to edges of the original graph G^o. Since the search exams all the edges in the queried community, the algorithm's time complexity is $O(|C|)$, and the space complexity is $O(|B|)$, where B denotes the returned boundary. For the sake of space, we omitted the detailed algorithm here.

4.2.3 Triangle connected maximin path search

Given two query vertices, a triangle connected maximin path is a path connecting two queries vertices that has all the edges are triangle connected and maximizes the minimum edge trussness. To find such a path, the algorithm first perform a max-k-truss community-level query to find communities containing both query vertices that have the highest trussness, denoted by $C_{max\tau}$. Then in one of the target communities, the algorithm starts a breadth first search that amid to find a path connecting any pair of edges of the source and target vertices. Due to the property of a maximum spanning tree, the found path is a maximin path of edge trussness. Such a path is guaranteed to exist as long as a max-k-truss community containing both the source and target vertices can be found because edges belonged to the same k-truss community is triangle connected. The algorithm performs a BFS after a max-k-truss info query, so the time complexity is $\sum_{v \in N_{src}} \tau_{(src,v)} + \sum_{v \in N_{dst}} \tau_{(dst,v)} + \min_{C \in C_{max\tau}} |C|$ and space complexity is $O(|P|)$, where P denotes the returned path. For the sake of space, we omitted the detailed algorithm here.

Table 1 Datasets

Dataset	Type	$\|V_{wcc}\|$	$\|E_{wcc}\|$	$\|\triangle_{wcc}\|$	k_{max}
Wiki	Communication	2.4M	4.7M	9.2M	53
Baidu	Web	2.1M	17.0M	25.2M	31
Skitter	Internet	1.7M	11.1M	28.8M	68
Sinaweibo	Social	58.7M	261.3M	213.0M	80
Livejournal	Social	4.8M	42.8M	285.7M	362
Orkut	Social	3.1M	117.2M	627.6M	78
Bio	biological	42.9K	14.5M	3.6B	799
Hollywood	Collaboration	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.

5 Evaluations

In this section, we evaluate our proposed index structure for various types of k-truss community related queries on real-world networks. We first compare the 2-level index with state-of-the-art solutions, the TCP index (Huang et al. (2014)) and the Equitruss index (Akbas and Zhao (2017)) for index construction (Section 5.1) and single vertex k-truss community search (Section 5.2.1). Then we show the effectiveness of our index for all three types of community-level k-truss community queries and their corresponding k-truss community search with single and multiple query vertices (Section 5.2.2 & Section 5.2.3). Finally, we analyze results of edge-level k-truss community queries (Section 5.3). All experiments are implemented in C++ and are run on a Cloudlab[2] c8220 server with 2.2GHz CPUs and 256GB memory.

Datasets

We use 8 real-world graphs of different types as shown in the table 1. 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[3] and Network Repository[4].

5.1 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 2. We exclude 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 2 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 stores the

[2] www.cloudlab.us

[3] snap.stanford.edu

[4] networkrepository.com

Table 2 Comparison of Index Construction

Graph	Graph	Index Time (Sec.)			Index Size (MB)		
Name	Size (MB)	TCP	Equi	Our	TCP	Equi	Our
Wiki	57.5	138.6	62.8	83.3	58.4	24.9	32.4
Baidu	224.5	493.9	268.5	350.3	305.8	179.0	237.3
Skitter	149.1	166.9	151.1	138.5	138.6	239.7	192.7
Sinaweibo	4049.9	11047.6	5724.3	6870.7	2743.8	1390.0	1810.4
Livejournal	627.6	1312.9	794.9	1020.0	1128.8	585.3	844.1
Orkut	1769.8	7659.4	3609.1	5058.7	3301.6	1721.8	2479.0
Bio	165.7	13963.9	6222.7	8873.6	393.1	176.6	289.4
Hollywood	791.7	16619.7	4154.4	10181.7	1928.7	812.5	1276.3

edges alongside vertices, which 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.

5.2 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 Huang et al. (2014) 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.

5.2.1 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 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.

Reason of performance difference. The main reason that the 2-level index is faster than the TCP index is the avoidance of the expensive BFS search during

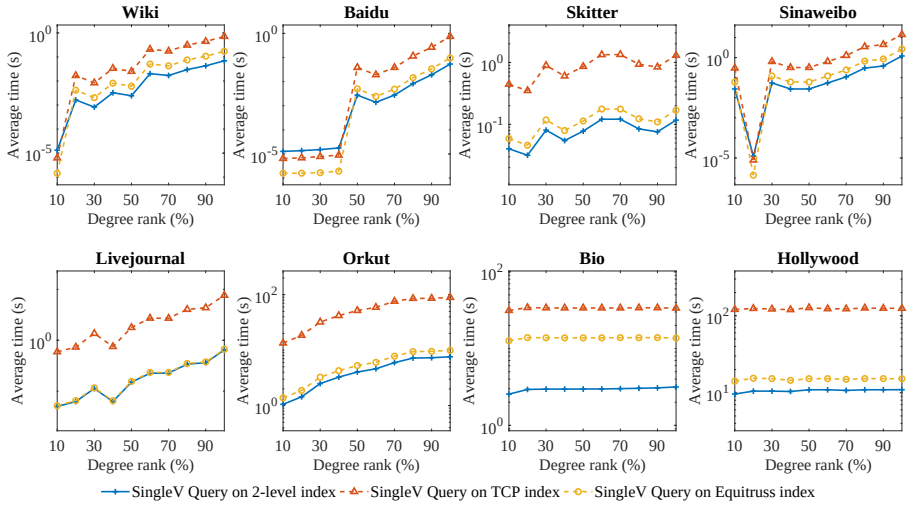
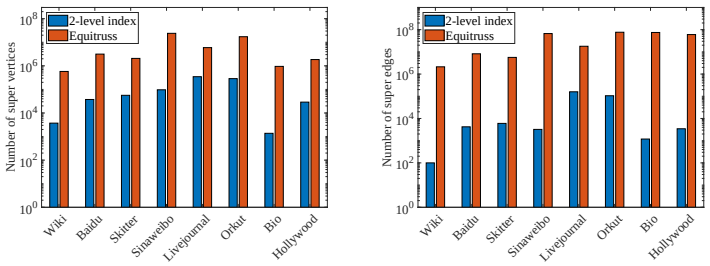


Fig. 6 Comparison of single vertex k-truss community search of the 2-level index, the TCP index and the Equitruss index.



(a) Number of vertices in super-graphs.

(b) Number of edges in super-graphs.

Fig. 7 Super-graph size comparison of the 2-level index and the Equitruss index.

query time. The reason for performance differences of the 2-level index and the Equitruss index on various graphs is less obvious given that they both search for target communities on a super-graph of the original graph and then collect edges belonging to the target community. The difference lies in the fact that vertices and edges in the super-graph of the two indices represent different subgraphs and their relations of the original graph. Each vertex in the 2-level index represents a single k-truss community while vertices in the Equitruss index only represents a fraction of a k-truss community. So one vertice in 2-level index may be split into several vertices in the Equitruss index.

We show the super-graph sizes of the 2-level index and the Equitruss index in Figure 7. We can see that the size of the super-graph of the Equitruss index is an order of magnitude larger than the super-graph of the 2-level index. The Equitruss index is slow while finding target communities due to the larger super-graph size. However, edge lists of a k-truss community can be more effectively retrieved as it

is already stored in each super vertex. For the 2-level index, target communities are easier to identify, however, one need to iterate through the adjacent lists stored in super vertices to retrieve edges in the community.

5.2.2 The community-level query vs. the edge-level query (community search).

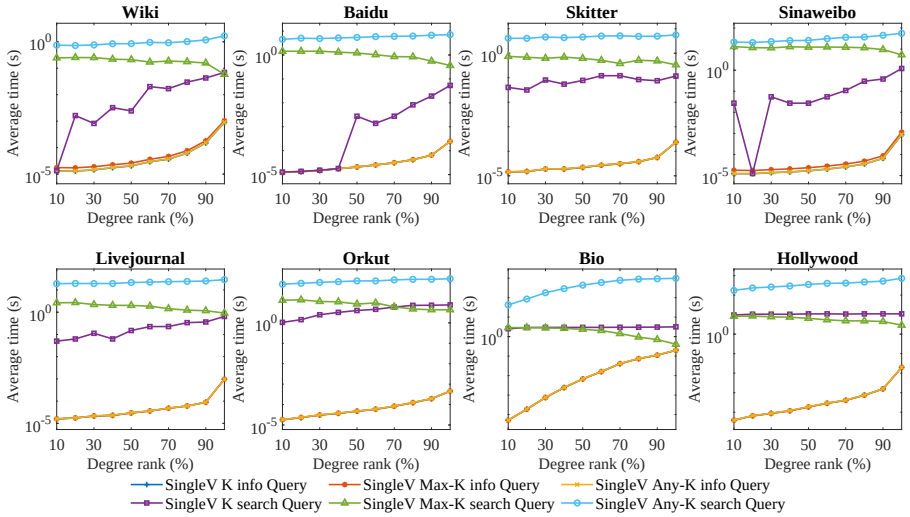


Fig. 8 Three types (k-truss, max-k-truss, any-k-truss) of single-vertex community-level k-truss community query *vs*. community search.

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 Figure 8 and Figure 9, 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.

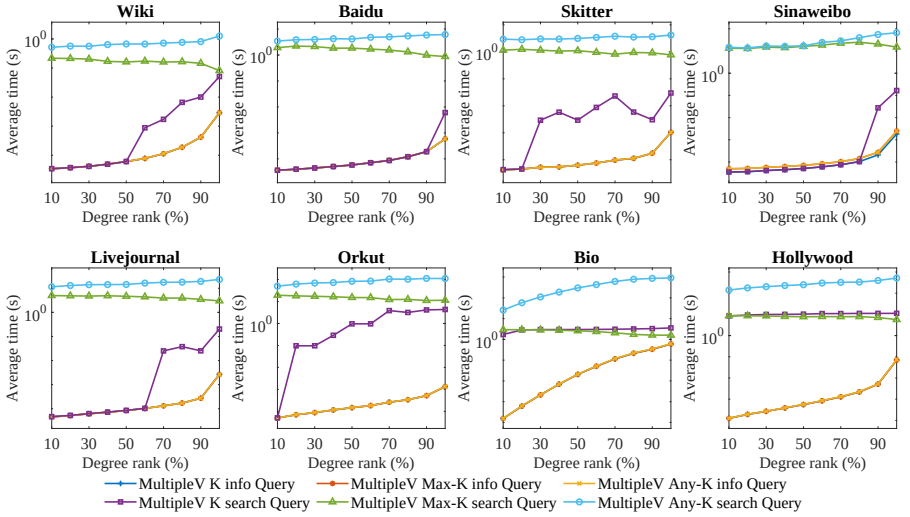


Fig. 9 Three types (k-truss, max-k-truss, any-k-truss) multiple-vertex (3) community-level k-truss community query *vs.* community search.

5.2.3 K-truss query vs. max-k-truss query vs. any-k-truss query.

We can also see in Figure 8 and Figure 9 any-k-truss community search queries always have the highest average run time because it searches all the possible truss communities to which the query vertex/vertices belong. We can also see that k-truss community search queries usually have much smaller average run time than max-k-truss community search queries. It is because that many k-truss queries fail to find a truss community as the query vertex/vertices do not belong to any truss community with the specified k, which is 10 in our experiments. However, this problem is less severe for max-k-truss queries. Max-k-truss queries can always find a target community as long as the query vertex/vertices belong to any truss community. In most cases, max-k-truss queries can provide more useful information for applications that do not have much knowledge of the community structure in the underlying graph.

Another interesting trend is that as the degree of query vertex increases, the average run time for k-truss and any-k-truss community search queries increases, the average run time for max-k-truss queries decreases. The trend is caused by different reasons for three types of query. For k-truss queries, the average query time increases as the query vertex degree increases because that it is more likely to find a target k-truss community with the specified k, which is 10 in our experiments. For max-k-truss queries, the average query time decreases as the query vertex degree increases because that target truss communities have higher trussness and smaller size. For any-k-truss queries, because the k-truss communities have hierarchical structures, more truss communities will be discovered by a query so that the average query time increases when the query vertex degree increases.

5.3 Edge-level Query Analysis

5.3.1 K-truss community boundary search.

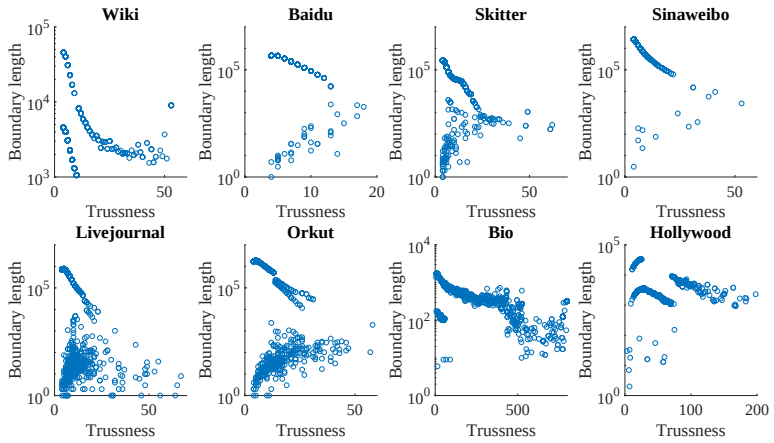


Fig. 10 Randomly sampled boundary length for k-truss communities with different trussness.

We randomly select 1000 query vertices from various degree buckets and perform the boundary search for the k-truss community with highest trussness that contains each query vertex and the trussness of the community and their boundary length in Figure 10. We can see that in many graphs there is a huge k-truss community of size several magnitude larger than other smaller k-truss communities. This community usually have a hierarchical structure, i.e., larger k-truss communities with low trussness contain smaller k-truss communities with high trussness. Figure 10 also shows that the upper bound of the boundary length of k-truss communities decreases as the trussness increases. The main reason for this is that sizes of high trussness k-truss communities are usually smaller than sizes of low trussness k-truss communities. However, the lower bound of the boundary length of k-truss communities increases as the trussness increases. The reason is that there are many small-size k-truss communities which are triangle connected to very few other k-truss communities, i.e., they are like isolated islands of the graph and many of them haven't formed a hierarchical structure.

5.3.2 Triangle connected maximin path search.

We randomly select 1000 pair of vertices from various degree buckets and show the average query time for the triangle connected maximin path search with them in Figure 11. The figure clearly shows that as the degree of vertex increases, the query time decreases. The reason is that for a pair of vertices with high degree, it is more likely that they belong to the same k-truss community with higher trussness and smaller size. So there is no surprise that the query vertices are closer to each other and a triangle connected maximim path between them tends to be shorter. We

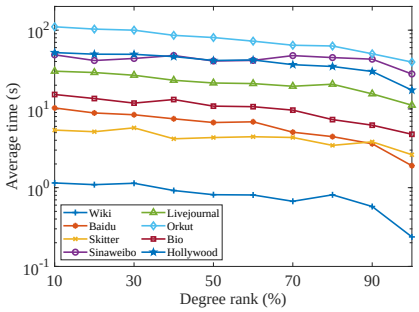


Fig. 11 Average query time for triangle connected maximim path search.

notice that the triangle connected maximin path search have much higher average query times for the same graph than k-truss community search because it needs to run a BFS traversal inside a target community which is very time-consuming.

6 Related Works

Our work is most related to the inspiring work (Huang et al. (2014)), 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 Huang et al. (2014) that each vertex holds their maximum spanning forest based on edge trussness of their ego-network. Later, the Equitruss index (Akbas and Zhao (2017)) 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.

Our work falls in the category of cohesive subgraph mining (Cui et al. (2014); Koujaku et al. (2016); Li et al. (2015); McAuley and Leskovec (2012); Sozio and Gionis (2010)) such as community detection and community search. Many previous works have studied this problem based on various cohesive subgraph models, such as clique (Bron and Kerbosch (1973); Rossi et al. (2014)), k-core (Barbieri et al. (2015); Li et al. (2017); Shin et al. (2016)), k-truss (Cohen (2008); Huang et al. (2014, 2015, 2016); Wang and Cheng (2012); Zheng et al. (2017)), k-plex (Wang et al. (2017)) and quasi-clique (Lee and Lakshmanan (2016); Tsourakakis et al. (2013)). The k-truss concept is first introduced by the work Cohen (2008). However, the original definition of a k-truss lacks the connectivity constraint so that a k-truss may be a unconnected subgraph. Huang et al. (2014) introduce the model of k-truss community based on triangle connectivity. The notion of triangle

connected k-truss communities is also referred to as $k-(2,3)$ nucleus in Sariyüce and Pinar (2016); Sariyüce et al. (2017) where they propose an approach based on the disjoint-set forest to speed up the process of nucleus decomposition. K-truss decomposition is also studied in Huang et al. (2016); Zou and Zhu (2017) for probabilistic graphs.

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 can deliver the 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. Li et al. (2015) introduce a novel community model called k-influential community based on the concept of k-core, which can capture the influence of a community. Wu et al. (2015) systematically study the existing goodness metrics and provide theoretical explanations on why they may cause the free rider effect. Huang et al. (2015) try to address the "free rider" issue by finding communities that meet cohesive criteria with the minimum diameter. There are also many works studied community related problems on attribute graph (Fang et al. (2016); Huang and Lakshmanan (2017); Shang et al. (2016, 2017)), e.g., finding communities that satisfy both structure cohesiveness, i.e., its vertices are tightly connected, and keyword cohesiveness, i.e., its vertices share common keywords. Interdonato et al. (2017) design a framework for local community detection in multilayer networks.

7 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 proved the effectiveness of our index structure theoretically and experimentally for processing both community-level queries and edge-level queries with a single query vertex or multiple query vertices. We compared with state-of-the-art methods for single-vertex k-truss community search and showed that our method has the best performance.

References

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

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

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

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

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

Fang Y, Cheng R, Luo S, Hu J (2016) Effective community search for large attributed graphs. Proceedings of the VLDB Endowment 9(12):1233–1244

Huang X, Lakshmanan LV (2017) Attribute-driven community search. Proceedings of the VLDB Endowment 10(9):949–960

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

Interdonato R, Tagarelli A, Ienco D, Sallaberry A, Poncelet P (2017) Local community detection in multilayer networks. Data Mining and Knowledge Discovery 31(5):1444–1479

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

Lee P, Lakshmanan LV (2016) Query-driven maximum quasi-clique search. In: Proceedings of the 2016 SIAM International Conference on Data Mining, SIAM, pp 522–530

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

Li Zj, Zhang WP, Li RH, Guo J, Huang X, Mao R (2017) Discovering hierarchical subgraphs of k-core-truss. In: International Conference on Web Information Systems Engineering, Springer, pp 441–456

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

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

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

Sariyüce AE, Seshadhri C, Pinar A, Çatalyürek ÜV (2017) Nucleus decompositions for identifying hierarchy of dense subgraphs. ACM Transactions on the Web (TWEB) 11(3):16

Shang J, Wang C, Wang C, Guo G, Qian J (2016) Agar: an attribute-based graph refining method for community search. In: Proceedings of the Sixth International Conference on Emerging Databases: Technologies, Applications, and Theory, ACM, pp 65–66

Shang J, Wang C, Wang C, Guo G, Qian J (2017) An attribute-based community search method with graph refining. The Journal of Supercomputing pp 1–28

Shin K, Eliassi-Rad T, Faloutsos C (2016) Corescope: Graph mining using k-core analysispatterns, 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

Wang Y, Xun J, Yang Z, Li J (2017) Query optimal k-plex based community in graphs. Data Science and Engineering pp 1–17

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

Xie J, Kelley S, Szymanski BK (2013) Overlapping community detection in networks: The state-of-the-art and comparative study. Acm computing surveys (csur) 45(4):43

Yin Z, Shi X (2017) Taming near repeat calculation for crime analysis via cohesive subgraph computing. arXiv preprint arXiv:170507746

Zheng Z, Ye F, Li RH, Ling G, Jin T (2017) Finding weighted k-truss communities in large networks. Information Sciences 417:344–360

Zong B, Xiao X, Li Z, Wu Z, Qian Z, Yan X, Singh AK, Jiang G (2015) Behavior query discovery in system-generated temporal graphs. Proceedings of the VLDB Endowment 9(4):240–251

Zou Z, Zhu R (2017) Truss decomposition of uncertain graphs. Knowledge and Information Systems 50(1):197–230