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

Anonymous Author(s)

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

ACM Reference Format:

Anonymous Author(s). 2019. 2-level Index for Truss Community Query in Large-Scale Graphs. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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. **[A well studied problem, which is called community search ([1, 2, 5, 8, 10, 14, 15, 25]), is to find communities with a query vertex and a specific cohesiveness measure. In this paper we are studying a more generalized problem. Given a set of query vertices and user defined cohesiveness criteria, find communities that both meet cohesiveness criteria and containing all query vertices. We refer to this problem as local community query as it is query-dependent and the runtime does not rely on the size of the graph. Communities in real world networks usually consist of similar vertices and the cohesiveness of a community indicates the similarity among its members, e.g., , users from a highly cohesive community share more common interests. As the community query reveals the community-level relations for a group of vertices, it can be used as a building block for many analytical tasks, such as similarity measurement [26], node classification [7], social recommendation [17], de-anonymization [30].]**

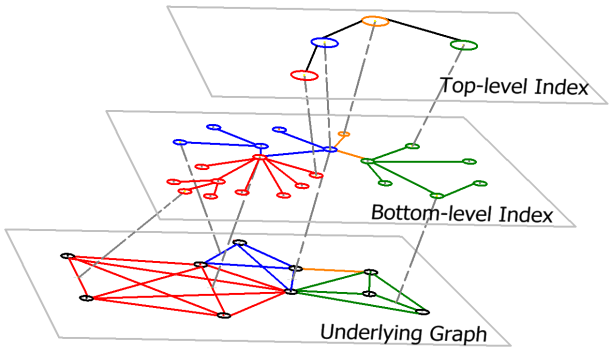


Figure 1: Two layer index structure for k-truss community queries. [The top level index is a super-graph with vertices representing unique k-truss communities and edges representing containment relations between them. The bottom level index is triangle derived graph that translates triangle connectivity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

[The normal procedure of community search involves exhaustively discover all the relevant communities, i.e., , enumerate all the edges in each community, leading to excessive computation time/space if edge-level details of communities are not of interest. There are various types of queries that are useful in real-world applications involving communities that do not require details of communities. For example, if one want to use the cohesiveness of common communities among a set of vertices as a similarity measure, it is not necessary to discover all edges in the common communities. Identifiers of common communities and metadata of each community, e.g., , cohesiveness measure and size, are sufficient. As such, we can generalize the concept of community search to local community queries, which is to identify common communities among a set of query vertices given cohesiveness creteria. Local community queries may or may not search the edge-level details of each relevant community according to application requirements. To get a better idea, consider the example query "Given a set of query vertices, do they belong to same communities?" This type of query does

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

not require edge-level details of communities and is good for applications such as studying common interests among a set of users. We refer to local community queries that do not require edge-level details as community-level queries. It is also possible that an application require the details of exactly which edges belong to relevant communities, such as classic community search queries. We refer to those queries as edge-level queries.]

To study local community queries, we adopt K-truss community as the community model. K-truss as a definition of cohesive subgraph, requires that each edge be contained in at least $(k-2)$ triangles within this subgraph. [8] introduce the model of k-truss community based on triangle connectivity that ensures the connectivity inside the community. **[The bounded diameter property of a K-truss community makes it an excellent choice for discovering cohesive and meaningful communities. The low computation cost of k-truss helps to scale to large-scale graphs.]**

Local community queries can benefit from compact index structures constructed from pre-computed results. Previous works mainly focused on **[the]** community search problem of a single query vertex ([1, 8]). 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**[ing]** unique k-truss communities and edges represent**[ing]** containment relations between them. For the bottom level index, **[w]**e 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 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 answer community-level queries directly. **[Once identifiers of relevant communities are found, they can be handed to the bottom level index to processing inner-community details for edge-level queries.]** For example, given a community search query, we first use the top level index to find target k-truss communities that contain all query vertices and then use the bottom level index to retrieve edges contained in each target k-truss community without expensive triangle enumerations. 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 various kinds of community-level and edge-level k-truss community queries on real-world graphs and demonstrate its performance by comparing with state-of-the-art methods, the TCP index ([8]) and the Equitruss index ([1]). For reproducibility, we make the source code available online.[1]

Our contribution can be summarized as follows.

- **[We generalize community search problem into local community query problem. The generalization lies in three aspects. First, we introducing both community-level queries and edge-level queries that provide different level of details for relevant communities. Second, we support multiple query vertices to enable applications that require community relation among query vertices. Third, we incorporate various cohesiveness criteria instead of a single cohesiveness measurement as used in related works.]**
- We develop a 2-level index structure that can efficiently process both the community-level and the edge-level k-truss community query for a single query vertex or a set of query vertices with any given cohesiveness creteria. 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 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 3.1. Section 3.2 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

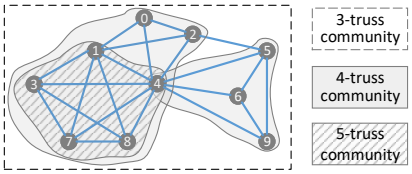


Figure 2: An example graph with four k-truss communities

In our problem, we consider an undirected, unweighted graph $G = (V, E)$. **[An example graph is shown in Figure 2.]** We define the set of neighbors of a vertex v in G as $N_v = u \in V : (v, u) \in E$, and the degree of v as $d_v = |N_v|$. We define a triangle $\triangle_{uvw}$ as a cycle of length 3 with distinct vertices $u, v, w \in V$. Then we define several key concepts as follows.

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

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

Definition 2 (Trussness). *The trussness of a subgraph $G' \in G$ is the minimum support of edges in G' plus 2, denoted by $\tau_{G'} =$*

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

$min\{(s_{e,G'}+2) : e \in E_{G'}\}$. *We then define edge trussness as:* $\tau_e = max_{G' \in G}\{\tau_{G'} : e \in E_{G'}\}$.

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

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

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

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

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

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

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

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

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. **[Because the edge support of edge** $(2,5)$ **is only** 1**, it cannot belong to a** 4**-truss. After excluding edge** $(2,5)$**, edges in the two** 4**-trusses are no longer triangle connected.]**

3 DESIGN OF 2-LEVEL INDEX

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

3.1 Construction of 2-level Index

The index proposed in this paper contains two levels **[to efficiently process both community-level queries and edge-level queries]**. The top level index provides information at the community level while the bottom level index offers information at the edge level. The top level is a super-graph, called community graph, whose vertices represent**[ing]** unique k-truss communities and edges represent**[ing]** containment relations between k-truss communities. **[One can easily locate relevant communities of a given query by using our** *union−intersection* **operation (We will discuss** *union−intersection* **operation in Section 3.2)]**. 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. **[After the relevant communities being retrieved, we can discover their inner community edge-level structures with the bottom level index instead of resorting to expensive triangle enumeration.]** 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 3.1.1). Then we introduce the community graph and show how to use simple graph traversals on **[the bottom level index, which stores a maximum spanning tree of triangle derived graph,]** to create the community graph (Section 3.1.2).

3.1.1 *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 **[defined as]** the lowest trussness of edges in the corresponding triangle in* G^o*. **[Because of this, the weight of an edge in** G^t **is always smaller or equal to weights of adjacent vertices.]***

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.

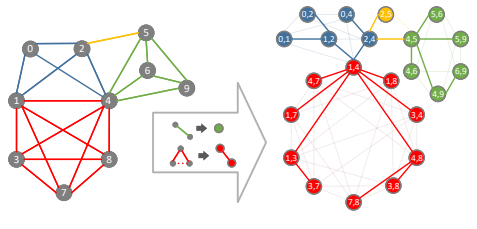


Figure 3: An example of the triangle derived graph and its maximum spanning tree of the example graph. [We show the id of each vertex in the underlying graph on the left. We use a pair of ids of vertices in the underlying graph as the id of a vertex of the triangle derived graph on the right because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

We use G^m to denote the maximum spanning forest of G^t that has been stored as the bottom level index. To construct G^m, one

way is generating the triangle derived graph G^t first and then finding a maximum spanning tree of it. However, this approach is impractical because we need to sort edges in G^t and the number of edges is three times the number of triangles 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, since edge weight in G^t represents minimum edge trussniss in the corresponding triangle, if we first sort edges of G^o from highest trussness to lowest trussness, then we iterate through the sorted list, and for each edge e^o, only list an adjacent triangle when e^o has the lowest trussness in that triangle. We can get the sorted order of edges in G^t with reduced space complexity, from $O(|\triangle^o|$ to $O(|E^o|)$]**. The details of the algorithm are shown in **[A]**lgorithm 1. **[Note that the MAKE-SET operation makes a set contains only a single edge, and the FIND-SET operation returns the SET that contains a given edge.]** The algorithm takes G^o and its edge trussness, which can be computed using the truss decomposition algorithm ([28]), as inputs. **[Since only G^m will be stored in the bottom level index, we will refer to G^m as the triangle derived graph in the following of the paper for similicity.]**

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 ∈ E^o do
    V^m ← V^m ⋃ {e, τ_e};
    MAKE-SET(e);
end
for (u, v) ∈ sorted do
    suppose u is the lower degree end of (u, v);
    for w ∈ N_u do
        if (v, w) ∈ E^o and τ_{u,v} < τ_{u,w} and τ_{u,v} < τ_{v,w}
         then
            ▷ Compare edge id if trussness of edges are
              equal to avoid duplication.;
            τ_△ = min(τ_(u,v), τ_(u,w), τ_(v,w));
            E^△ ← △_{u,v,w};
            for e^△ ∈ E^△ do
                if FIND-SET(e^△.v_1) ≠ FIND-SET(e^△.v_2) then
                    E^m ← E^m ⋃ e^△;
                    UNION(e^△.v_1, e^△.v_2);
                end
            end
        end
    end
end
return G^m(V^m, E^m)
```

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 [28]. Sorting all the edges with counting sort costs $O(|E^o| + k_{max})$ time and $O(|E^o| + k_{max})$ space**[, where k_{max} is the maximum trussness value]**. The time and space complexity for iterate 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 $k_{max} \ll |E^o|$ are **[dominated by the cost of computation of edge trussness, 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.

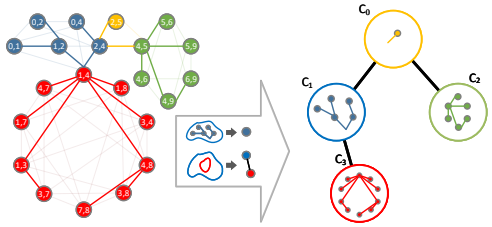


Figure 4: [Construction of the community graph from the triangle derived graph.]

3.1.2 *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**[ing]** unique k-truss communities and edges represent**[ing]** containment relations between k-truss communities. We call this index structure the community graph and denote it as G^c. Based on the hierarchical property of k-truss ([4]), i.e., for $k \geq 2$, each k-truss is the subgraph of a $(k-1)$-truss, we have the formal definition of the community graph in Definition 8.

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.*

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

Theorem 1. *The community graph G^c is a forest.*

A key property of the community graph is that it is a forest (Theorem 1). **[Due to limited space, we show the proof of Theorem 1 in the appendix.] [This property enable us to easily construct the community graph with a single BFS traversal on the bottom level index G^m.]** The traversal algorithm create a tree in the community graph G^c of each connected component

in G^m. **[To construct the new tree in G^c, the algorithm iteratively processes vertices using the BFS traversal and map vertices in G^m to vertices in G^c. The map between a vertex u^m to a vertex v^c means that the edge represented by u^m belongs to the k-truss community represented by v^c and is stored in a lookup table H. When the traversal algorithm reaches a vertex v^m_{seed} belonging to a new connected component, it first create a new super vertex in G^c and use it as a start point to build a new tree in G^c. Note that this start point is not necessarily the root of the new tree. After that, for an arbitrary vertex u^m from the same connected component that has been discovered by the traversal, assuming its parent vertex during traversal is p^m and p^m has been mapped to v^c_p 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^c_p and its ancestor in G^c, the weight of the edge (p^m, u^m) and the weight of the vertex u^m.]**

Theorem 2. *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 $k \leq \tau_{(u^m, v^m)}$.*

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 can be explained by Theorem 2. **[Due to limited space, we show the proof of Theorem 2 in the appendix.] [The intuition is that edges in G^m represents triangle adjacency in the original graph G^o. Since edge weight in G^m represents minimum edge trussniss in the corresponding triangle in G^o and G^m is a maximum spanning tree. The weight of edge (p^m, u^m) limits the highest trussness of a k-truss community u^m's representing edge can belongs to. The procedure of mapping a vertex u^m to a super vertex in G^c can be divided in following two steps. First, start at u^m's parent vertex p^m's corresponding super vertex v^c_p, the algorithm exam ancestors of v^c_p until it finds a super vertex with weight smaller than or equal to the weight of (p^m, u^m) (line 8). This super vertex, which we reuse the symbol v^c_p, represents the community to which u^m can triangle connect. Second, if $\tau_{e^m} = \tau_{u^m} = \tau_{v^c_p}$, we can directly map u^m to v^c_p (line 20). Otherwise, we need to create a new super vertex and either add it as a new child of v^c_p (line 22) or insert it between v^c_p and one of its existing child (line 11-12 and 14-16).]**

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

Algorithm 2: Top Level Index Construction

Data: $G^m(V^m, E^m)$
Result: $G^c(V^c, E^c)$, H

```
1  for each connected component CC ∈ G^m do
2      v^m_seed ← CC.pop();
3      create_super_vertex(seed, null);
4      for u^m ∈ BFS starting at seed do
5          p^m ← parent of u^m in BFS;
6          e^m ← (u^m, p^m);
7          v^c_p ← H[p^m];
8          while τ_{v^c_p} > τ_{e^m} do v^c_p' ← v^c_p, v^c_p ← v^c_p.parent ;
9          if τ_{v^c_p} < τ_{e^m} then
10             if τ_{e^m} = τ_{u^m} then
11                 v^c_u ← create_super_vertex(u, v^c_p);
12                 v^c_p'.parent ← v^c_u;
13             else
14                 v^c_e ← create_super_vertex(e, v^c_p);
15                 v^c_u ← create_super_vertex(u, v^c_e);
16                 v^c_p'.parent ← v^c_e;
17             end
18         else
19             if τ_{e^m} = τ_{u^m} then
20                 H[u^m] ← v^c_p;
21             else
22                 v^c_u ← create_super_vertex(u, v^c_p);
23             end
24         end
25     end
26 end
27 return G^c(V^c, E^c), H
28 function create_super_vertex (u^m, v^c_p)
29     create v^c_u, H[u^m] ← v^c_u;
30     v^c_u.parent ← v^c_p;
31     V^c ← V^c ⋃ v^c_u;
32     return C_u
33 end
```

3.2 Query on 2-level Index

First, we introducing both community-level queries and edge-level queries that provide different level of details for relevant communities. Second, we support multiple query vertices to enable applications that require community relation among query vertices. Third, we incorporate various cohesiveness criteria instead of a single cohesiveness measurement as used in related works.

We classify k-truss local community queries into two categories according to the level of information required. The community-level query (Section 3.2.1) 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 3.2.2). 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. **[Besides of the two query categories, we also incorporate the ability to add various cohesiveness criteria for queries with our 2-level index.]**

3.2.1 The Community-level Query.

[The 2-level index supports any range of trussness value as cohesiveness criterion for a query. They all support both a single query vertex and a set of query vertices.] Here we listed three most common types of them:

- 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.

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^c ← H[(u^o, v^o)];
21         B ← branch_search(u^c);
22         SS ← SS ⋃ B;
23     end
24     return SS
25 end
```

All three types of community-level k-truss community queries share a similar querying process on the top level index, call**[ed]** *union − intersection* procedure as shown in Algorithm 3. **[Given the 2-level index and a lookup table that maps edges in the original graph G^o to vertices in the community graph G^c as input, the algorithm first iterate through adjacent edges of each query vertex. For each adjacent edge (u^o, v^o), the algorithm starts at u^c, which is the vertex in G^c that is mapped by (u^o, v^o), and collect all its ancestors along the tree branch it belongs in G^c (line 9-16). Then the algorithm takes the *union* of all the tree branches of adjacent edges (line 17-23). The set of super vertices SS represent all the k-truss communities that contains a query vertex. In the next step, the algorithm calculate the *intersection* of the set of super vertices SS of each query vertex to get the set of common super vertices S that represents all the k-truss communities that contains all query vertices (line 3-8).]** Finally, from the set of common super vertices S, 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 time and space complexity for collecting ancestor for a given super vertex isτ_e. To iterate all the adjacent edges of a query vertex to discover the set of super vertex SS takes $\sum_{v \in N_u} \tau_{(u,v)}$ time and space. Finally, the algorithm needs to find the set of super vertex SS for each query vertex to get the set of common super vertex S,] so the total time and space complexity for the *union−intersection* procedure is $\sum_{u \in Q} \sum_{v \in N_u} \tau_{(u,v)}$.

3.2.2 The Edge-level Query.

The edge-level k-truss community query requires information of finest granularity as it needs to explore the inner edge-level structure of a k-truss community. Our bottom level index contains the detailed triangle connectivity information that makes such queries possible. To process a k-truss community query, we first locate the target k-truss communities with the top level index and then compute query results using edge-level details provided by the bottom level index. We show three concrete examples of this type of queries in this section: the k-truss community search (Section **??**), the k-truss community boundary search (Section **??**) and the triangle connected maximin path search (Section **??**).

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. **[A]**lgorithm 4 shows the details.

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

k-truss community boundary search

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 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.

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.

4 IMPLEMENTATION OF 2-LEVEL INDEX

In this section, we show the structure we used for the 2-level index to combine the triangle derived graph and the community graph (Section 4.1).

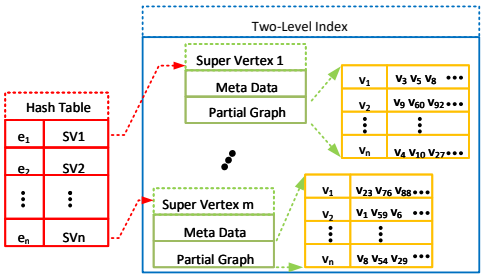


Figure 5: Overall structure of the 2-level index.

4.1 2-level IndexStructure

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 [A]lgorithm 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 [1], 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.

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 ([8]) and the Equitruss index ([1]) 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 [T]able 1. To simplify our experiments, we treat them as undirected,

[2] www.cloudlab.us

Table 1: Datasets

Dataset	Type	$\|V_{wcc}\|$	$\|E_{wcc}\|$	$\|\triangle_{wcc}\|$	k_{max}
Wiki	Comm.	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	Collab.	1.1M	56.3M	4.9B	2209

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

Table 2: Comparison of Index Construction

Graph	[Decomp.]	Index Time (Sec.)			Index Size (MB)		
Name	[Time (Sec.)]	TCP	Equi	Our	TCP	Equi	Our
Wiki	[239]	139	63	83	58	25	32
Baidu	[742]	494	269	350	306	179	237
Skitter	[366]	167	151	139	139	240	193
Sinaweibo	[10728]	11048	5724	6871	2744	1390	1810
Livejournal	[2201]	1313	795	1020	1129	585	844
Orkut	[9028]	7659	3609	5059	3302	1722	2479
Bio	[11239]	13964	6223	8874	393	177	289
Hollywood	[14002]	16620	4154	10182	1929	813	1276

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 [T]able 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 [T]able 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 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.

[3] snap.stanford.edu
[4] networkrepository.com

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 [8] 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 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). 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

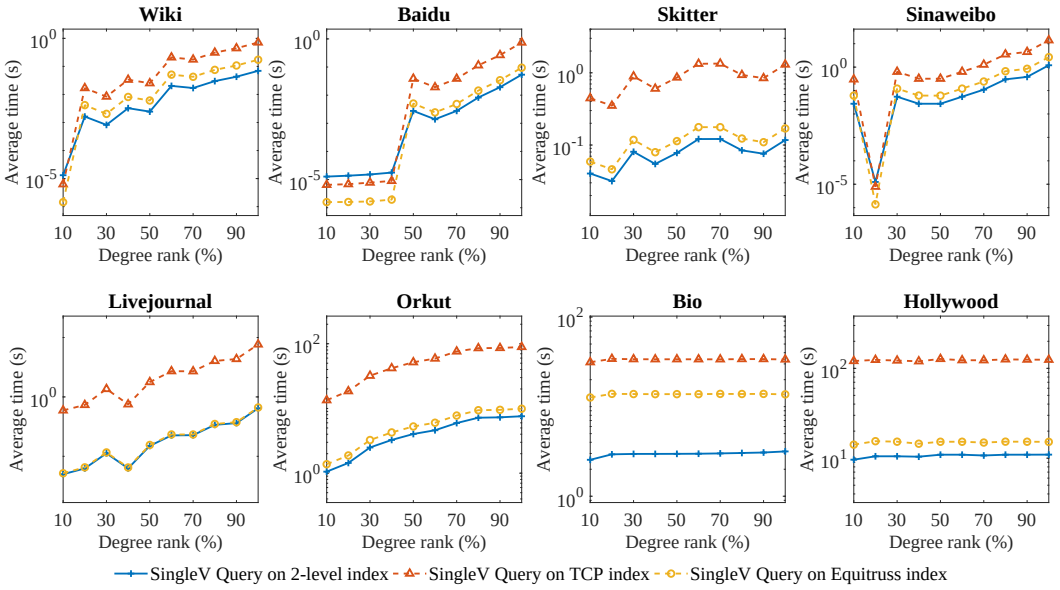
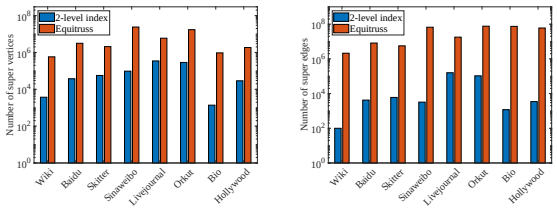


Figure 6: Comparison of single vertex k-truss community search of the 2-level index, the TCP index and the Equitruss index.



(a) Number of vertices in super-graphs. (b) Number of edges in super-graphs.

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

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.

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. We randomly select 1000 query vertices from various degree buckets and perform the boundary search for the k-truss community with highest trussness

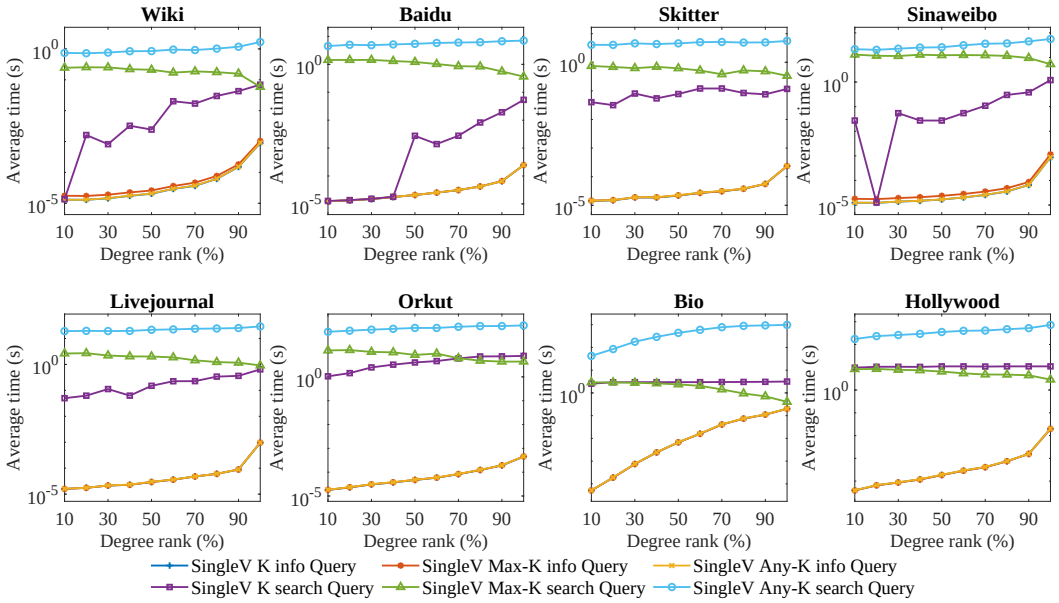


Figure 8: Three types (k-truss, max-k-truss, any-k-truss) of single-vertex community-level k-truss community query *vs*. community search.

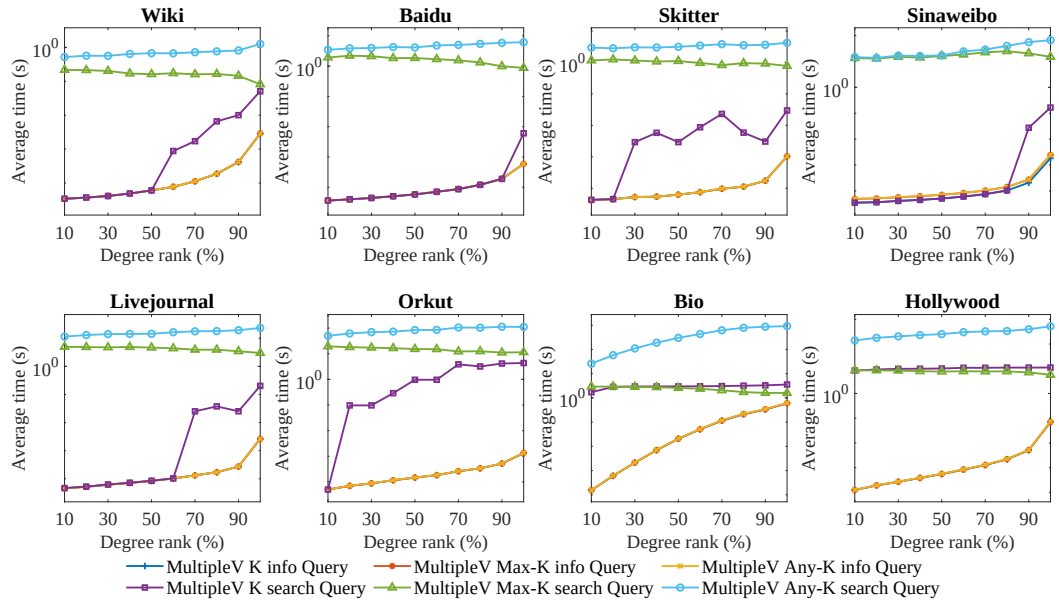


Figure 9: Three types (k-truss, max-k-truss, any-k-truss) multiple-vertex (3) community-level k-truss community query *vs*. community search.

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.

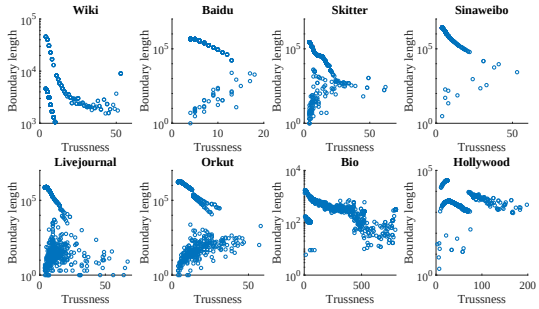


Figure 10: Randomly sampled boundary length for k-truss communities with different trussness.

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.

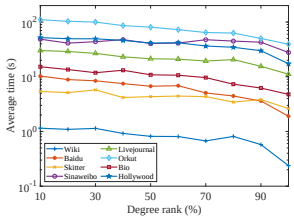


Figure 11: Average query time for triangle connected maximim path search.

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 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 ([8]), 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 [8] that each vertex holds their maximum spanning forest based on edge trussness of their ego-network. Later, the Equitruss index ([1]) is proposed that use a super-graph based on truss-equivalence as an index to speed up the single vertex k-truss community search. The Equitruss index is similar to our index structure in the sense that it also use a super-graph for the index. However, the vertex in the super-graph of the Equitruss index is a subgraph of a k-truss community while an edge represents triangle connectivity. Our 2-level index contains a more compact super-graph that a vertex contains a k-truss community and edges represent k-truss community containment relations. We have used both TCP index and Equitruss index as comparisons in our evaluation.

Our work falls in the category of cohesive subgraph mining ([5, 13, 15, 18, 25]) such as community detection and community search. Many previous works have studied this problem based on various cohesive subgraph models, such as clique ([3, 19]), k-core ([2, 16, 24]), k-truss ([4, 8, 10, 11, 28, 32]), k-plex ([29]) and quasi-clique ([14, 27]). The k-truss concept is first introduced by the work [4]. However, the original definition of a k-truss lacks the connectivity constraint so that a k-truss may be a unconnected subgraph. [8] 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 [20, 21] 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 [11, 33] for probabilistic graphs.

An α-adjacency γ-quasi-k-clique model is introduced by [5] for online searching of overlapping communities. ρ-dense core is a pseudo clique recently introduced by [13] that can deliver the optimal solution for graph partition problems. The pattern of k-core structures is studied by [24] for applications such as finding anomalies in real world graphs, approximate degeneracy of large-scale graphs and so on. [15] introduce a novel community model called k-influential community based on the concept of k-core, which can capture the influence of a community. [31] systematically study the existing goodness metrics and provide theoretical explanations on why they may cause the free rider effect. [10] 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 ([6, 9, 22, 23]), 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. [12] 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

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

[2] Nicola Barbieri, Francesco Bonchi, Edoardo Galimberti, and Francesco Gullo. 2015. Efficient and effective community search. *Data Mining and Knowledge Discovery* 29, 5 (2015), 1406–1433.

[3] Coen Bron and Joep Kerbosch. 1973. Algorithm 457: finding all cliques of an undirected graph. *Commun. ACM* 16, 9 (1973), 575–577.

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

[5] Wanyun Cui, Yanghua Xiao, Haixun Wang, and Wei Wang. 2014. Local search of communities in large graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 991–1002.

[6] Yixiang Fang, Reynold Cheng, Siqiang Luo, and Jiafeng Hu. 2016. Effective community search for large attributed graphs. *Proceedings of the VLDB Endowment* 9, 12 (2016), 1233–1244.

[7] Keith Henderson, Brian Gallagher, Lei Li, Leman Akoglu, Tina Eliassi-Rad, Hanghang Tong, and Christos Faloutsos. 2011. It's who you know: graph mining using recursive structural features. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 663–671.

[8] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-truss community in large and dynamic graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 1311–1322.

[9] Xin Huang and Laks VS Lakshmanan. 2017. Attribute-driven community search. *Proceedings of the VLDB Endowment* 10, 9 (2017), 949–960.

[10] Xin Huang, Laks VS Lakshmanan, Jeffrey Xu Yu, and Hong Cheng. 2015. Approximate closest community search in networks. *Proceedings of the VLDB Endowment* 9, 4 (2015), 276–287.

[11] Xin Huang, Wei Lu, and Laks VS Lakshmanan. 2016. Truss decomposition of probabilistic graphs: Semantics and algorithms. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 77–90.

[12] Roberto Interdonato, Andrea Tagarelli, Dino Ienco, Arnaud Sallaberry, and Pascal Poncelet. 2017. Local community detection in multilayer networks. *Data Mining and Knowledge Discovery* 31, 5 (2017), 1444–1479.

[13] Sadamori Koujaku, Ichigaku Takigawa, Mineichi Kudo, and Hideyuki Imai. 2016. Dense core model for cohesive subgraph discovery. *Social Networks* 44 (2016), 143–152.

[14] Pei Lee and Laks VS Lakshmanan. 2016. Query-Driven Maximum Quasi-Clique Search. In *Proceedings of the 2016 SIAM International Conference on Data Mining*. SIAM, 522–530.

[15] Rong-Hua Li, Lu Qin, Jeffrey Xu Yu, and Rui Mao. 2015. Influential community search in large networks. *Proceedings of the VLDB Endowment* 8, 5 (2015), 509–520.

[16] Zhen-jun Li, Wei-Peng Zhang, Rong-Hua Li, Jun Guo, Xin Huang, and Rui Mao. 2017. Discovering Hierarchical Subgraphs of K-Core-Truss. In *International Conference on Web Information Systems Engineering*. Springer, 441–456.

[17] Jiaqi Liu, Qi Lian, Luoyi Fu, and Xinbing Wang. 2018. Who to Connect to? Joint Recommendations in Cross-layer Social Networks. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 1295–1303.

[18] Julian J McAuley and Jure Leskovec. 2012. Learning to Discover Social Circles in Ego Networks.. In *NIPS*, Vol. 2012. 548–56.

[19] Ryan A Rossi, David F Gleich, Assefaw H Gebremedhin, and Md Mostofa Ali Patwary. 2014. Fast maximum clique algorithms for large graphs. In *Proceedings of the 23rd International Conference on World Wide Web*. ACM, 365–366.

[20] Ahmet Erdem Sariyüce and Ali Pinar. 2016. Fast hierarchy construction for dense subgraphs. *Proceedings of the VLDB Endowment* 10, 3 (2016), 97–108.

[21] Ahmet Erdem Sariyüce, C Seshadhri, Ali Pinar, and Ümit V Çatalyürek. 2017. Nucleus decompositions for identifying hierarchy of dense subgraphs. *ACM Transactions on the Web (TWEB)* 11, 3 (2017), 16.

[22] Jingwen Shang, Chaokun Wang, Changping Wang, Gaoyang Guo, and Jun Qian. 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, 65–66.

[23] Jingwen Shang, Chaokun Wang, Changping Wang, Gaoyang Guo, and Jun Qian. 2017. An attribute-based community search method with graph refining. *The Journal of Supercomputing* (2017), 1–28.

[24] Kijung Shin, Tina Eliassi-Rad, and Christos Faloutsos. 2016. CoreScope: Graph Mining Using k-Core Analysis—Patterns, Anomalies and Algorithms. In *Data Mining (ICDM), 2016 IEEE 16th International Conference on*. IEEE, 469–478.

[25] Mauro Sozio and Aristides Gionis. 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, 939–948.

[26] Anton Tsitsulin, Davide Mottin, Panagiotis Karras, and Emmanuel Müller. 2018. Verse: Versatile graph embeddings from similarity measures. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 539–548.

[27] Charalampos Tsourakakis, Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Maria Tsiarli. 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, 104–112.

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

[29] Yue Wang, Jian Xun, Zhenhua Yang, and Jia Li. 2017. Query Optimal k-Plex Based Community in Graphs. *Data Science and Engineering* (2017), 1–17.

[30] Xinyu Wu, Zhongzhao Hu, Xinzhe Fu, Luoyi Fu, Xinbing Wang, and Songwu Lu. 2018. Social network de-anonymization with overlapping communities: Analysis, algorithm and experiments. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 1151–1159.

[31] Yubao Wu, Ruoming Jin, Jing Li, and Xiang Zhang. 2015. Robust local community detection: on free rider effect and its elimination. *Proceedings of the VLDB Endowment* 8, 7 (2015), 798–809.

[32] Zibin Zheng, Fanghua Ye, Rong-Hua Li, Guohui Ling, and Tan Jin. 2017. Finding weighted k-truss communities in large networks. *Information Sciences* 417 (2017), 344–360.

[33] Zhaonian Zou and Rong Zhu. 2017. Truss decomposition of uncertain graphs. *Knowledge and Information Systems* 50, 1 (2017), 197–230.

A PROOF OF THEOREM 1

Proof. First, if there is a cycle $v_1^c \ldots v_i^c \ldots v_n^c$ in the community graph G^c and their corresponding k-truss community in the original graph G^o is $C_1^o \ldots C_i^o \ldots C_n^o$, respectively. According to Definition 6 and Definition 8, $C_1^o \ldots C_i^o \ldots C_n^o$ are triangle connected, and it is impossible for an arbitrary pair of communit**[ies]** 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. □

B PROOF OF THEOREM 2

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_u^o and e_v^o 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 $\tau_{(u^m,v^m)}$. □