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

Zheng Lu, Yunhe Feng, Qing Cao

*Dept. of Electrical Engineering & Computer Science*, *University of Tennessee*, Knoxville, USA

{zlu12, yfeng14, cao}@utk.edu

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

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

* 1. INTRODUCTION

Online social networks have seen rapid growth in recent years and have generated massive data. The amount of avail-able data has attracted much research effort on developing advanced data mining technologies as fundamentals for important applications and insightful analytical tasks [[3],](#_bookmark30) [[7],](#_bookmark34) [[11],](#_bookmark38) [[12].](#_bookmark39) Graphs are naturally used to model real-world networks such as social networks. A well-studied graph problem, known as community search [[1],](#_bookmark28) [[4]–[6],](#_bookmark33) [[8],](#_bookmark35) is to find communities with a query vertex and a specific cohesiveness measure. In this paper, we study a more generalized problem. Given a set of query vertices and user-defined cohesiveness criteria, we try to find community-level relations among all query vertices with or without edge-level details. We refer to this problem as local community query because it is query-dependent and the runtime does not rely on the size of the graph. 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 [[9],](#_bookmark36) social recommendation [[7],](#_bookmark34) and de-anonymization [[12].](#_bookmark39)

The normal procedure of community search involves making an exhaustive discovery of all the relevant communities, i.e., enumerating all the edges in each community, which leads to excessive computation time/space when edge-level details of communities are not pertinent. For example, if one wants to use the cohesiveness of common communities among a set of vertices as a similarity measure, it is not necessary to discover all the edges in common communities. The identifiers of common communities

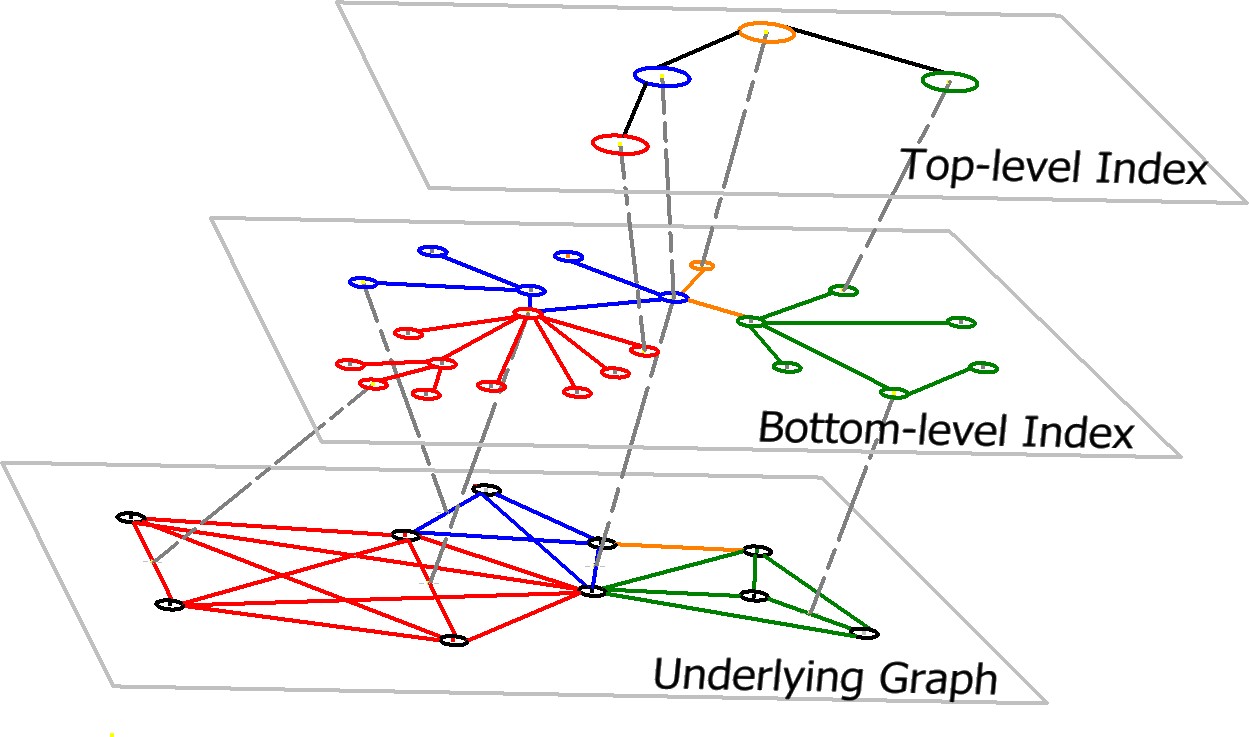


Fig. 1. Two-layer index structure for k-truss community queries. The top-level index is a super-graph with the vertices representing unique k-truss communities and the edges representing the containment relations between them. The bottom-level index is a triangle-derived graph that translates the triangle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

and the cached statistics of each community, e.g. cohesiveness, measure, and size, are sufficient. As such, we can generalize the concept of community search to local community queries, which are meant to identify common communities among a set of query vertices given the cohesiveness criteria. Depending on application requirements, local community queries may or may not search the edge-level details of each relevant community. To get a better idea, let us consider an example query: “Given a set of query vertices, do they belong to same communities?” This type of query does not require the 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 like community-level queries. An application might also possibly require the details regarding exactly which edges belong to relevant communities, such as classic community search queries. We refer to those queries as edge-level queries.

In this paper, we adopt k-truss community as the com- munity model. K-truss, as a definition of a cohesive subgraph, requires each edge to be contained in at least *k* 2 triangles.

−

1. introduces the model k-truss community based on triangular connectivity, which ensures that the community remains connected. 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 in the scaling of large graphs.

Local community queries can benefit from compact index

structures constructed from pre-computed results. Previous works were mainly focused on the community search problem of a single query vertex [[1],](#_bookmark28) [[5].](#_bookmark32) In this paper, we propose a novel two-level index structure to support both the community-level and the edge-level local k-truss community queries. An overview of our two-level index is shown in [Figure 1.](#_bookmark0) The top-level index is a super-graph whose vertices represent unique k-truss communities and whose edges represent the containment relations between them. For the bottom-level index, we introduce a new type of graph called triangle-derived graph that translates triangle connectivity to edge connectivity for fast k-truss community traversal. We can use simple *union* and *intersection* operations on the top-level index to locate relevant k-truss communities in a given query, in order to answer community-level queries directly. Once the identifiers of relevant communities are found, they can be handed to the bottom-level index to answer edge-level queries. 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 the edges contained in each target k-truss community without expensive triangle enumerations.

Our index also supports queries with arbitrary cohesiveness criteria. Previous works required a specific cohesiveness measurement, e.g., a specific *k*, to process a query. However, in reality such criteria may not be available. For example, one might not successfully find whether a set of query vertices belongs to same communities or which is the most cohesive community that contains all the query vertices. It will be quite hard to come up with a proper *k* value before processing a given query.

We prove our index and query process to be theoretically optimal, show its practical efficiency for various types of community-level and edge-level local k-truss community queries on real-world graphs, and demonstrate its performance by comparing with state-of-the-art methods, the TCP index [[5]](#_bookmark32), and the Equitruss index [[1].](#_bookmark28) For reproducibility, we make the source code available online.[1](#_bookmark1)

Our contribution can be summarized as follows:

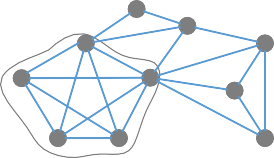
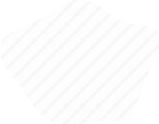
* + We generalize the community search problem into the local community query problem. The generalization comprises three aspects. First, we introduce both community-level queries and edge-level queries that provide different levels of information for relevant communities. Second, we support multiple query vertices to enable applications that require community relations among query vertices. Third, we incorporate various cohesiveness criteria instead of a single cohesiveness measurement, which is used in related works.
  + We develop a two-level index structure that can efficiently process both the community-level and the edge-level k-truss community query for a single query vertex or a set of query vertices with any given cohesiveness criteria, using an efficient two-step process.

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

* + We perform extensive experiments on our 2-level index for large-scale real-world graphs and show that our index structure outperforms state-of-the-art index structures.

The rest of this paper is organized as follows. Section [II](#_bookmark2) provides notations and definitions. We design the index structure and its query process in Section [III.](#_bookmark4) The evaluations are in Section [IV.](#_bookmark16) We discuss previous works in Section [V](#_bookmark26) and conclude our work in Section [VI.](#_bookmark27)

* 1. PRELIMINARIES



9

7 8

6

3 4

5

1

2

0

4-truss

community

3-truss

community

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



5-truss

community

In our problem, we consider an undirected, unweighted graph *tt* = (*V, E*). An example graph is shown in [Figure 2.](#_bookmark3) We define the set of neighbors of a vertex *v* in *tt* as *Nv* = *u V*: (*v, u*) *E*, and the degree of *v* as *dv* = *Nv*. Then, we define several key concepts as follows.

∈ ∈ | |

*Definition 1 (Edge support)*:The support of an edge *eu,v E* is defined as *se,G* = *uvw w V*.

|O ∈ |

∈

For example, in [Figure 2,](#_bookmark3) 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 *tt* is the minimum support of edges in *tt*j plus 2, denoted by *τGt* = *min*{(*se,Gt* + 2): *e* ∈ *EGt* }. We then define edge trussness as follows: *τe* = *maxGt* ∈*G τGt* : *e EGt* .

∈

{ ∈ }

For example, in [Figure 2,](#_bookmark3) subgraph (1*,* 3*,* 7*,* 8*,* 4) has a truss-

ness of 5, as all edges in it have support at least 5 2 = 3. The trussness of edge (1*,* 4) is also 5.

−

*Definition 3 (k-truss)*:Given a graph *tt* and *k* 2, *tt*j *tt* is a k-truss if *e EGt, se,Gt* (*k* 2). *tt*j is a maximal k-truss subgraph if it is not a subgraph of another k-truss subgraph with the same trussness *k* in *tt*.

∀ ∈ ≥ −

≥ ⊆

Because the K-truss definition does not define the connectivity, we use the definition of triangle connectivity.

*Definition 4 (Triangle adjacency):* 1, 2 are adjacent if they share a common edge, i.e., 1 2 .

O ∩ O ∅

O O

*Definition 5 (Triangle connectivity)*:1, 2 are triangle connected if they can reach each other through a series of adjacent triangles, i.e., for 1 *i < n, i i*−1 = . Two

O O

∃ ∈ O ∈ O

≤ O ∩ O ƒ ∅

edges *e*1, *e*2 are triangle connected if *e*1 1, *e*2 2,

1 and 2 are identical or triangle connected.

O O

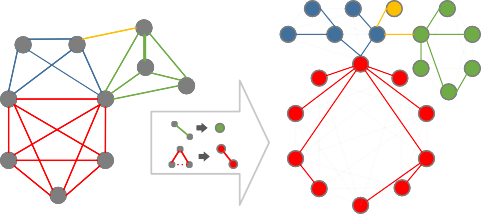
For example, in [Figure 2,](#_bookmark3) (1*,* 4) is triangle connected to

(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 with all its edges being triangle connected.



0,,2 0,,4 2,,5

5,,6

5

0,,1

1,,2

2,,4

4,,5

5,,9

0

2

6

1,,4

4,,6

6,,9

4,,7

1,,8

9

4,,9

1

4

1,,7

3,,4

3

8

1,,3

4,,8

3,,7

3,,8

7

7,,8

[Figure 2](#_bookmark3) shows several examples of k-truss communities. The whole example graph is a three-truss community as every edge has the support of at least one and all edges are triangle connected. Note that there are two separate four-truss communities. Because (2, 5) has the support of 1, it cannot belong to a four-truss. After excluding edge (2, 5), edges in the two four-trusses are no longer triangle connected.

* 1. DESIGN OF TWO-LEVEL INDEX

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

1. *Construction of two-level Index*

The index proposed in this paper contains two levels to efficiently process both community-level queries and edge-level queries. The top level is a super-graph, called community graph, whose vertices represent unique k-truss communities and edges represent the containment relations between k-truss communities. The bottom level is a maximum spanning forest of a triangle-derived graph that preserves the edge level trussness and triangle connectivity inside the k-truss communities. The index is constructed in a bottom-up manner. In the following, we first formally define the triangle-derived graph and introduce the algorithm to construct the same from the original graph (Section [III-A1).](#_bookmark5) Then, we introduce the community graph and show how to use simple graph traversals on the bottom-level index to create the community graph (Section [III-A2).](#_bookmark8)

* 1. *MST of Triangle-Derived Graph:*

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

*ttm*; one way to do this is by generating the triangle-derived graph *ttt* first and then finding its maximum spanning tree. However, this approach is impractical because we need to sort the edges in *ttt*, which can have an order of magnitude larger than the original graph *tt*. We use two methods to avoid the same. First, we find that for real-world graphs, the highest edge trussness is usually small compared to the size of the graph, e.g. only a few thousand for the densest graph in our experiments. Hence, we can use counting sort instead of comparison sort to reduce the time complexity. Second, since edge weight in *ttt* represents the minimum edge trussness in the corresponding triangle in *tto*, we can sort edges in the original graph *tto* to get the sorted order of the triangles in *tto*, which can be translated into the sorted order of edges in *ttt*. These two methods reduce both the time and space complexity of the maximum spanning tree algorithm.

We need edge trussness before constructing bottom-level index, which can be computed using the truss decomposition algorithm [[10]](#_bookmark37) as inputs. The time and space complexities for constructing the bottom-level index are dominated by the computation of edge trussness of *tto*, which are

The triangle-derived graph of an original graph *tto*is obtained by associating a vertex with each edge of *tt*and connecting

*O*(Σ

(*u,v*)∈*Eo*

*min*{*du*

*, dv*}) and *O*(|*Eo*|), respectively. Since

two vertices if the corresponding edges of *tto* belong to the same triangle. Then, we only store a maximum spanning tree of the triangle-derived graph, which is enough to store the edge-level structures of k-truss communities in the original graph. We show the formal definition of a triangle-derived graph in *[Definition 7](#_bookmark7)*[.](#_bookmark7) We show an example of the triangle-derived graph and its maximum spanning forest in *[Figure 3](#_bookmark6)*[.](#_bookmark6) We outline the maximum spanning forest with bold lines.

*Definition 7 (triangle-derived graph):* The triangle-derived graph *ttt* is a weighted, undirected graph where each edge in the original graph *tto* is represented as a vertex in *ttt*. *ttt* has an edge *et* which connects vertices vt , vt ttt if and only if their corresponding edges *eo, eo tto* belong to the same triangle in *tto*. The weight of a vertex in *ttt* is defined as the trussness of its corresponding edge in *tto*. The weight of an edge in *ttt* is defined as the lowest trussness of edges in the corresponding triangle in *tto*. Thus, the weight of an edge in *ttt* is always smaller or equal to the weights of adjacent vertices.

1

2

1

2

∈

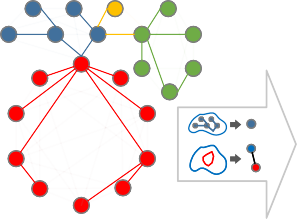
∈

We use *ttm* to denote the maximum spanning forest of *ttt*

that has been stored as the bottom-level index. To construct

*ttm* is a maximum spanning forest, the bottom-level index

takes *O*(|*V m*|) = *O*(|*Eo*|) space to store it.



0,,2 0,,4 2,,5

5,,6

0,,1 1,,2

2,,4

4,,5 5,,9

1,,4

4,,6 6,,9

4,,7

1,,8

4,,9

1,,7

3,,4

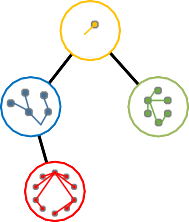
1,,3

4,,8

3,,7

3,,8

7,,8



C0

C1

C2

C3

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

* 1. *Community Graph:*

The top-level index is a super-graph whose vertices represent unique k-truss communities and edges represent containment relations between k-truss communities. We call

this index structure the community graph and denote it as *ttc*. Based on the hierarchical property of k-truss [[3],](#_bookmark30) i.e., for *k* 2, each *k*-truss being the subgraph of a (*k* 1)-truss, we have the formal definition of the community graph in *[Definition 8](#_bookmark10)*[.](#_bookmark10) We show an example of the community graph in [Figure 4.](#_bookmark9)

≥ −

**Algorithm 1:** Top Level Index Construction

**Data:** *ttm*(*V m, Em*)

**Result:** *ttc*(*V c, Ec*), *H*

**1 for** *each connected component CC* ∈ *ttm* **do**

*v*

*Definition 8 (community graph)*:The community graph *ttc* **2**

*m*

*seed*

← *CC.pop*();

is a weighted undirected graph that represents each k-truss community **3**

in the original graph *tto* by a vertex in *ttc*. *ttc* **4**

has an edge *ec* in connecting vertices *vc, vc* ∈ *ttt* if and only if **5**

1 2

create super vertex(*seed*, *null*);

**for** *um* ∈ *BFS starting at seed* **do**

*pm* ← parent of *um* in BFS;

**6** *em* ← (*um, pm*);

the following two conditions are met for their corresponding

* 1. russ communities *Co, Co* ∈ *tto*: **7**

*vc* ← *H*[*pm*];

1 2 *p*

*c* j *c*

*Co* is a subgraph of *Co* or the other way around.

**8 while** *τvc > τem* **do** *vp*

← *vp*,

* + - 1 2 *c p*

We assume without loss of generality that *Co* is a subgraph

1

*p*

*vp* ← *vc.parent* ;

*o o o o*

**9 if** *τvc < τem* **then**

of *C*2, there is no *C*3 ∈ *tt* and *C*1 is a subgraph *p*

of *Co* and *Co* is a subgraph of *Co*.

**10 if** *τem* = *τum* **then**

3 3 2

*c*

**11** *vc* ← create super vertex(*u*, *vc*);

*tt* only has vertex weights, which represent the trussness of the corresponding k-truss communities. **12**

*v* j*.parent* ← *v* ;

A key property of the community graph is that it is a forest. **13**

*u p*

*c c*

*p u*

# else

**14** *vc* ← create super vertex(*e*, *vc*);

For the sake of space, we omit the proofs here. This property

*e*

*p*

**15** *vc* ← create super vertex(*u*, *vc*);

enable us to easily construct the community graph with a single breath first search (BFS) on the bottom-level index **16** *ttm*. The traversal algorithm creates a tree in the community **17** graph *ttc* of each connected component in *ttm*. To construct **18** a tree in *ttc*, the algorithm iteratively processes vertices in **19** *ttm* using the BFS and map vertices in *ttm* tovertices in **20** *ttc*. The map between a vertex *um ttm* to a vertex *vc ttc* **21** means that the k-truss community represented by *vc* has the highest **22** trussness among all k-truss communities that contains the edge **23** represented by *um*. We store this mapping in a lookup table **24** *H*. For example, in our example graph in [Figure 2,](#_bookmark3) edge **25**

∈ ∈

*v* j*.parent* ← *v* ;

*u e*

*c c*

*p e*

# end else

**if** *τem* = *τum* **then**

*m c*

*H*[*u* ] ← *v* ;

*p*

**else**

*c c*

*v* ← create super vertex(*u*, *v* );

*u p*

# end end

**end**

1*,* 4 belongs to three k-truss communities denoted as *C*0*, C*1*,* and *C*3 in [Figure 4.](#_bookmark9) Since *C*3 has the highest trussness of 5, edge 1*,* 4 is mapped to *C*3.

**26 end**

**27 return** *ttc*(*V c, Ec*)*, H*

When the traversal algorithm reaches a vertex *vm* belong-

*seed*

ing to a new connected component, it first creates a new super-vertex in *ttc* and uses it as a starting point to build a new tree in *ttc*. Note that this starting point is not necessarily the root of the new tree. Thereafter, for an arbitrary vertex *um* from the same connected component that has been discovered by the BFS, assuming its parent vertex during the search is *pm* (whichhas been mapped to *vc* already), *um* will be mapped on an existing vertex or a new vertex in *ttc* depending on the relations of the weight of the super-vertex *vc* and its ancestor in *ttc*, the weight of the edge (*pm, um*), and the weight of the vertex *um*. The detailed rule for the mapping can be found in Algorithm [1.](#_bookmark11)

*p*

*p*

The reason we use weights of the edge (*pm, um*) between a vertex and its parent vertex as references when mapping a vertex of *ttm* to *ttc*,is that edges in *ttm* represents triangle adjacency in the original graph *tto*. Since edge weight in *ttm* represents minimum edge trussness in the corresponding triangle in *tto* and *ttm* is a maximum spanning tree, the weight of edge (*pm, um*) limits the highest trussness of a k-truss community, to which the *um*’s representing edge can belong.

For each vertex of *ttm*, searching the ancestor super-vertex (in *ttc*)of its parent vertex in *ttm* takes *O*(*kmax*) time, where

*kmax* is the highest trussness of k-truss communities in *tto*. Since, the index construction process is a BFS on a maximum spanning tree with *O*(*Eo*) vertices, the total construction time is *O*(*kmax Eo*). As each vertex in *ttc* represents a k-truss community in *tto* and *ttc* is a forest, the algorithm takes *O*( *Co* ) space and the index size is *O*(*Co*), where *Co* is the number of communities in *tto*.

1. *Query on Two-level Index*

| |

| |

| | | | | |

We classify k-truss local community queries into two categories according to the level of information required. The community-level query (Section [III-B1)](#_bookmark12) only requires the information on the relations between k-truss communities and to which k-truss communities the query vertices belong. For example, “Do query vertices belongs to same k-truss communities?” These queries can be answered solely by the top-level index. Another type of queries, which requires edge-level information to process, is called the edge-level query (Section [III-B2).](#_bookmark15) The widely studied community search query is the simplest form of such queries. We process this type of queries by first locating the target k-truss communities with the top-level index and then diving into the edge-level details with the bottom-level

index. Besides the two query categories, we also incorporate the ability to add various cohesiveness criteria for the queries with our two-level index.

* 1. *The Community-level Query:*

The query process for community-level local k-truss community query is called the *union intersection* procedure. We show the detailed procedure in Algorithm [2.](#_bookmark17) Given the two-level index and a lookup table *H* that maps and represents edges in the original graph *tto* (indicated by vertices in *ttm*) to vertices in the community graph *ttc* as input, the algorithm first iterates through the adjacent edges of each query vertex. For each edge maps to a vertex in *ttc*, the algorithm takes its *union* and its ancestors, discovered by the search towards the root of the tree, in *ttc*. This signifies all the communities to which a query vertex belongs. Then, we take the *intersection* of the results of all query vertices which represents the communities to which all query vertices belong.

−

The two-level index supports any range of trussness value as

a cohesiveness criterion for a query. There are three most common cohesiveness criteria: a specific *k* value, the maximum *k* value, and any other *k* values. We refer to them as k-truss queries, max-k-truss queries, and any-k-truss queries. With the results of the *union intersection* procedure, we can easily retrieve communities that have weights of a specified *k* (k-truss query), identify how many communities have maximum trussness (Max-k-truss query), or return all the vertices (Any-k-truss query).

−

**Algorithm 2:** *union* − *intersection* Algorithm.

**Data:** *tto*(*Vo, Eo*), *ttc*(*Vc, Ec*), *H*, *Q*

**Result:** Subgraph *S* of *ttc*

**1** *S*;

← ∅

**2** *initialized false*;

←

**3 for** *uo Q* **do**

∈

**4** *SS* single subgraph(*uo*);

←

**5 if** !*init* **then** *S SS*, *init true*;

← ←

**6 else** *S S SS*;

← ∩

**7 end**

**8 return** *S*

**9 function** *singlev subgraph (uo)*

**10** *SS* ← ∅;

**11 for** *vo* ∈ *Nu* **do**

**12** *uc* ← *H*[(*uo, vo*)];

**13** *B* ← ancesStors of *uc* in *ttc*;

*SS* ← *SS*

*B*;

**14**

TABLE I DATASETS

Dataset Type *|Vwcc| |Ewcc| |Owcc| kmax*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Skitter  Sinaweibo | Internet 1.7M 11.1M 28.8M 68  Social 58.7M 261.3M 213.0M 80 | | | | |
| Orkut | Social | 3.1M | 117.2M | 627.6M | 78 |
| Bio | biological | 42.9K | 14.5M | 3.6B | 799 |
| Hollywood | Collab. | 1.1M | 56.3M | 4.9B | 2209 |

Number of vertices, edges, triangles and the maximum trussness (*kmax*) in the largest WCC. Sorted by the number of triangles.

TABLE II

COMPARISON OF INDEX CONSTRUCTION

Graph Decomp. Index Time (Sec.) Index Size (MB)

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

* 1. *The Edge-level Query:*

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

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

−

Note that the two-level index is capable of processing more complex edge-level queries besides community search. Due to limited space, we cannot undertake further discussion on this issue here.

The tiΣme andΣspace complexity of *union*−*intersection* algo-

rithm is

*v*∈*Nu τ*(*u,v*). Each edge in the target commu-

*u*∈*Q*

**15 end**

**16 return** *SS*

nities will only be accessed exactly once, so, the time and space

compleSxity for the search are *u*∈*Q v*∈*Nu τ*(*u,v*) + |*Ci*|,

Σ Σ S

**17 end**

The time and space complexity for the collecting ancestor of

where *Ci* is the union of target communities.

* 1. EVALUATIONS

a given super-vertex is *τe* because *ttc* is a forest. To reiterate, all the adjacent edges of a query vertex takes *v Nu τ*(*u,v*) time and space. Finally, the algorithm needs to find the set of super-vertices for each query vertex to get a set of common super-vertices, so the total time and space complexity for the *union*−*intersection* procedure is *u*∈*Q v*∈*Nu τ*(*u,v*).

Σ ∈

Σ Σ

In this section, we evaluate our proposed index structure for various types of local k-truss community queries on real-world networks. We compare the two-level index with state-of-the-art solutions, the TCP index [[5]](#_bookmark32), and the Equitruss index

[[1]](#_bookmark28) for index construction (Section [IV](#_bookmark20)-A), single vertex k-truss community search (Section [IV-B1)](#_bookmark23), and multiple-vertex local

100

Average time (s)

**Skitter**

100

Average time (s)

**Sinaweibo**

**Orkut**

**Bio**

102

Average time (s)

**Hollywood**

10 30 50 70 90

Degree rank (%)

10 30 50 70 90

Degree rank (%)

100

10 30 50 70 90

Average time (s)

Degree rank (%)

100

10 30 50 70 90

Average time (s)

Degree rank (%)

10 30 50 70 90

Degree rank (%)

SingleV Query on 2-level index SingleV Query on TCP index SingleV Query on Equitruss index

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

100

Average time (s)

**Skitter**

100

Average time (s)

**Sinaweibo**

100

Average time (s)

**Orkut**

100

Average time (s)

**Bio**

100

Average time (s)

**Hollywood**

10 30 50 70 90

Degree rank (%)

10 30 50 70 90

Degree rank (%)

10 30 50 70 90

Degree rank (%)

10 30 50 70 90

Degree rank (%)

10 30 50 70 90

Degree rank (%)

MultipleV Community-level Query on 2-level index MultipleV Community-level Query on Equitruss index

Fig. 6. Comparison of multiple vertex k-truss community-level query of the 2-level index and the Equitruss index.

k-truss community query (Section [IV-B2), respectively.](#_bookmark24) All experiments are implemented in C++ and run on a Linux server with 2.2 GHz CPUs and 256 GB memory.

We use five real-world graphs of different types, shown in Table [I.](#_bookmark13) To simplify our experiments, we treat them as undirected, un-weighted graphs and only use the largest weakly connected component of each graph. All datasets are publicly available from the Stanford Network Analysis Project[2](#_bookmark21) and Network Repository[3](#_bookmark22).

1. *Index construction*

In this section, we show the index size and index construction time of the two-level index compared to the TCP index and the Equitruss index in Table [II.](#_bookmark14) We separate the truss decomposition time for all three methods. We can see in Table [II](#_bookmark14) that the two-level index has a construction time that is comparable with the Equitruss index, and that both are faster than the TCP index. The index size of the two-level index is smaller than the TCP index. However, the Equitruss has the smallest index size since it only stores edge list of the original graph while the two-level index also preserves the triangle connectivity inside k-truss communities.

1. *Query performance*

In this section, we evaluate the query time of k-truss community queries to show the effectiveness of the two-level index. We perform 10-truss community queries and discard vertices with degrees less than 20, as they normally do not

2snoap.stanford.edu

3networkrepository.com

belong to any 10-truss community. Because the query time heavily replies on the degree of query vertices, we uniformly partition the rest of vertices according to their degrees into 10 buckets. We randomly select 100 sets of query vertices for each degree bucket.

* 1. *Single-vertex k-truss community search*:We first evalu-

ate the single-vertex k-truss community search performance and compare the query time with the TCP index and the Equitruss index. The results are shown in [Figure 5.](#_bookmark18) The two-level index achieves the best average query time for all graphs. It has an order of magnitude speedup compared to the TCP index for all graphs, along with 5% to 400% speedup compared to the Equitruss index for all graphs. The speed up is linear, as all three indices have the same time complexity to handle single vertex k-truss community search queries. Notably, very low average query time (around 10−5 seconds) means that there is no vertex belonging to any k-truss community in that degree rank.

The main reason why the two-level index is faster than the TCP index is the avoidance of the expensive BFS search during query time. The reason behind the performance differences of the two-level index and the Equitruss index on various graphs lies in the fact that the super-graph size of the two-level index is much smaller, making it easier to locate target communities.

* 1. *Multiple-vertex k-truss community query:* We first per-

form community-level multiple-vertex k-truss queries with both the two-level index and the Equitruss index. We are not able to perform the same experiment on the TCP index as there is no easy modification that would enable it to support multiple-vertex queries. We can see in [Figure 6](#_bookmark19) that the two-level index outperforms the Equitruss index by several orders of magnitude. This clearly shows the difference between the two indices; the Equitruss index has

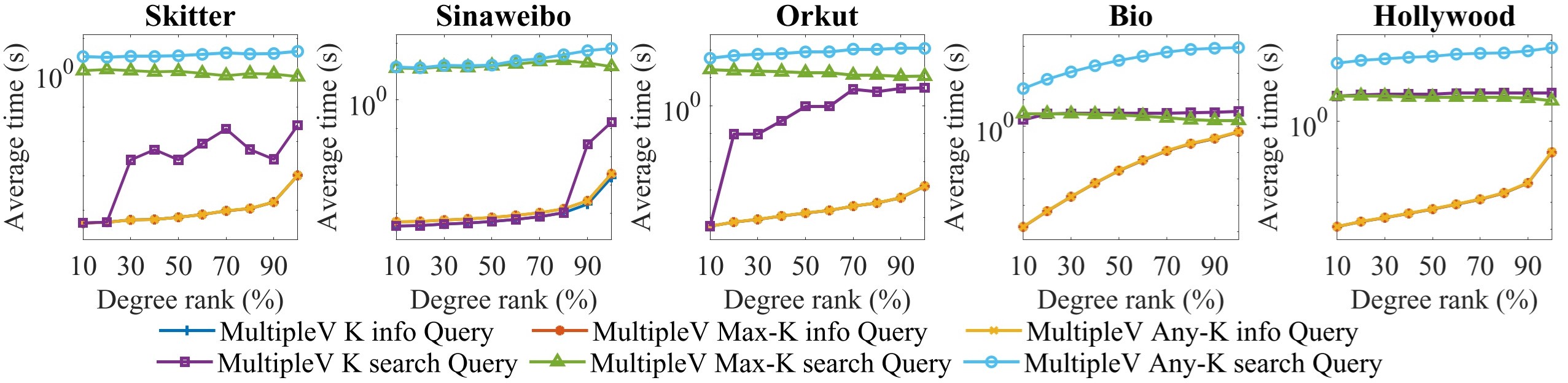


Fig. 7. Three types (k-truss, max-k-truss, any-k-truss) of multiple-vertex (3) community-level k-truss community queries vs.community search.

a larger super-graph and is slower for community-level queries, where only the super-graph is required. The performance difference shows that our two-level index is quite suitable for community-level queries.

We then perform both the community-level and the edge-level multiple-vertex queries with all three basic types of cohesiveness criteria, i.e., k-truss, max-k-truss, and any-k-truss. We show the results in [Figure 7.](#_bookmark25) The average time for

community-level queries spans from 3*.*4 ×10−5 seconds to 0*.*06 seconds. Typically, the average time for community search queries is much higher than community-level queries (since one needs to access edge-level information), ranging from 0*.*03–91*.*9 seconds depending on the size of the target communities. Among all the three types of cohesiveness criteria, any-k-truss edge-level queries usually take the highest query time as it needs to process the most k-truss communities.

* 1. RELATED WORKS

Our work falls in the category of cohesive subgraph mining [[2],](#_bookmark29) [[4],](#_bookmark31) [[6],](#_bookmark33) [[8],](#_bookmark35) such as community detection and community search. It is closely related to an inspiring work [[5]](#_bookmark32) which introduced the model of k-truss community based on triangle connectivity. However, triangle-connected k-truss communities have slow computation efficiency, especially for vertices belonging to large k-truss communities. To speed up the community search based on this model, an index structure called the TCP index is proposed in [[5],](#_bookmark32) where each vertex holds its maximum spanning forest based on the edge trussness of their ego-network. Later, the Equitruss index [[1]](#_bookmark28) is proposed, which uses a super-graph based on truss-equivalence as an index to speed up the single vertex k-truss community search. The Equitruss index is similar to our index structure in the sense that it also uses a super-graph for the index. However, the vertex in the super-graph of the Equitruss index is a subgraph of a k-truss community, while an edge represents the triangle connectivity. Our two-level index contains a more compact super-graph, in which a vertex contains a k-truss community and the edges represent the k-truss community containment relations. We have used both TCP index and Equitruss index as comparisons in our evaluation.

* 1. CONCLUSION

In this work, we categorize local k-truss community queries into the community-level query and the edge-level query. We designed 2-level index that stores the community graph in the top level index for locating relevant communities and the triangle derived graph in the bottom level index to preserve the triangle connectivity at the edge level inside each k- truss community. We showed experimentally that our index structure outperformed state-of-the-art methods.

REFERENCES

1. E. Akbas and P. Zhao. Truss-based community search: A truss- equivalence based indexing approach. *Proceedings of the VLDB Endowment*, 10(11):1298–1309, 2017.
2. D. Bera, F. Esposito, and M. Pendyala. Maximal labelled-clique and click-biclique problems for networked community detection. In *2018 IEEE Global Communications Conference (GLOBECOM)*, pp 1–6. IEEE, 2018.
3. J. Cohen. Trusses: Cohesive subgraphs for social network analysis.

*National Security Agency Technical Report*, 16, 2008.

1. W. Cui, Y. Xiao, H. Wang, and W. Wang. Local search of communities in large graphs. In *Proceedings of the 2014 ACM SIGMOD international* *conference on Management of data*, pp. 991–1002. ACM, 2014.
2. X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu. Querying k-truss community in large and dynamic graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 1311–1322. ACM, 2014.
3. R.-H. Li, L. Qin, J. X. Yu, and R. Mao. Influential community search in large networks. *Proceedings of the VLDB Endowment*, 8(5): 509–520, 2015.
4. J. Liu, Q. Lian, L. Fu, and X. Wang. Who to connect to? Joint recommendations in cross-layer social networks. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pp. 1295–1303. IEEE, 2018.
5. M. Sozio and A. Gionis. The community-search problem and how to plan a successful cocktail party. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data* *mining*, pages 939–948. ACM, 2010.
6. A. Tsitsulin, D. Mottin, P. Karras, and E. Mu¨ller. Verse: Versatile graph embeddings from similarity measures. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*, pages 539–548. International World Wide Web Conferences Steering Committee, 2018.
7. J. Wang and J. Cheng. Truss decomposition in massive networks.

*Proceedings of the VLDB Endowment*, 5(9):812–823, 2012.

1. J. Wang, C. Jiang, S. Guan, L. Xu, and Y. Ren. Big data driven similarity-based u-model for online social networks. In *GLOBECOM 2017-2017* *IEEE Global Communications Conference*, pp. 1–6. IEEE, 2017.
2. X. Wu, Z. Hu, X. Fu, L. Fu, X. Wang, and S. Lu. Social network de- anonymization with overlapping communities: Analysis, algorithm and experiments. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pp. 1151–1159. IEEE, 2018.