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

# I/O Efficient K-Truss Community Search in Massive Graphs

**Yuli Jiang · Xin Huang · Hong Cheng**

the date of receipt and acceptance should be inserted later

**Abstract** Community detection that discovers all densely connected communities in a network has been studied a lot. In this paper, we study *online community search* for query-dependent communities, which is a different but practically useful task. Given a query vertex in a graph, the problem is to find meaningful communities that the vertex belongs to in an online manner. We propose a community model based on the $k$-truss concept, which brings nice structural and computational properties. We design a compact and elegant index structure which supports the efficient search of $k$-truss communities with a linear cost with respect to the community size. We also investigate the $k$-truss community search problem in a dynamic graph setting with frequent insertions and deletions of graph vertices and edges. In addition, to support $k$-truss community search over massive graphs which cannot entirely fit in main memory, we propose I/O-efficient algorithms for query processing under the semi-external model. Extensive experiments on massive real-world networks demonstrate the effectiveness of our $k$-truss community model, the efficiency, and the scalability of our in-memory and semi-external community search algorithms.

**Keywords** $k$-truss; community search; dynamic graphs; semi-external algorithms

Yuli Jiang
The Chinese University of Hong Kong, Hong Kong, China
E-mail: yljiang@se.cuhk.edu.hk

Xin Huang
Hong Kong Baptist University, Hong Kong, China
E-mail: xinhuang@comp.hkbu.edu.hk

Hong Cheng
The Chinese University of Hong Kong, Hong Kong, China
E-mail: hcheng@se.cuhk.edu.hk

## 1 Introduction

Community structure exists in many real-world networks, for example, social networks and biological networks [18]. Community detection, which is to find communities in a network, has been studied a lot in the literature [29, 37]. A different but related problem is *online community search*, which finds communities containing a query vertex in an online manner [31, 7, 32, 8, 17, 3, 19, 36, 1, 11, 20, 38, 42, 43, 13, 44, 12, 18, 41, 23, 24]. These two tasks have different goals: community detection targets all communities in the entire network and usually applies a global criterion to find qualified communities. In contrast, online community search provides *personalized community detection* for a query vertex. As communities for different vertices in a network may have very different characteristics, this user-centered personalized search is more meaningful. Furthermore, as the communities a user participates in represent the social contexts of the user, online community search provides a useful tool for other analytical tasks, such as social circle discovery [28] and social contagion modeling [33]. In this paper, we study modeling and querying of the communities of a query vertex.

Cui et al. [7] have proposed a novel approach for online overlapping community search. A community model was defined as an $\alpha$-adjacency-$\gamma$-quasi-$k$-clique. A $\gamma$-quasi-$k$-clique is a $k$-node graph with at least $\lfloor \gamma \frac{k(k-1)}{2} \rfloor$ edges. Another parameter $\alpha$ is imposed to union two $\gamma$-quasi-$k$-cliques if they share at least $\alpha$ vertices. Given a query vertex $q$, the problem is to find all $\alpha$-adjacency-$\gamma$-quasi-$k$-cliques containing $q$. However, there are several limitations in this community model.

1. $\gamma$ as an average density measure, may not necessarily guarantee a cohesive community structure. Consider the graph in Figure 1 which is a 0.8-quasi-7-clique containing query vertex $q$. However, $q$ is only

connected with one vertex in the community, thus it is not a cohesive community for $q$ obviously.

2. There are three parameters $\alpha, \gamma, k$ in this model, the setting of which may vary significantly for different query vertices. For example, in a research collaboration network, the communities of a famous scholar and a junior scholar can be dramatically different in terms of the community size and density. Thus it is difficult to choose proper values for the three parameters given a query vertex.
3. Finding $\alpha$-adjacency-$\gamma$-quasi-$k$-clique has been proven to be NP-hard [7], which imposes a severe computational bottleneck. The approximate algorithms for clique enumeration and expansion [7] reduce the complexity, but cannot give a theoretical guarantee of the approximation quality.

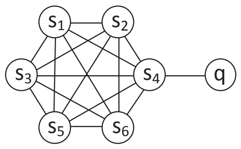


**Fig. 1** A 0.8-quasi-7-clique containing $q$

Considering these limitations, we propose a novel community model based on the $k$-truss concept. Given a graph $G$, the $k$-truss of $G$ is the largest subgraph in which every edge is contained in at least $(k-2)$ triangles within the subgraph [6]. The $k$-truss is a type of cohesive subgraph defined based on triangle which models the stable relationship among three nodes. However, the $k$-truss subgraph may be disconnected, for example, the two shaded regions form the 4-truss subgraph in Figure 2(a) which is obviously disconnected. So the $k$-truss subgraph may not correspond to a meaningful community. On top of the $k$-truss, we impose an *edge connectivity* constraint, that is, any two edges in a community either belong to the same triangle, or are reachable from each other through a series of adjacent triangles. Here two triangles are defined as *adjacent* if they share a common edge. The edge connectivity requirement ensures that a discovered community is connected and cohesive. This defines our novel ***k-truss community model***. To the best of our knowledge, this is the first work that proposes the $k$-truss community. Compared with the $\alpha$-adjacency-$\gamma$-quasi-$k$-clique model, our community model has the following advantages.

1. **Cohesive community**. The $k$-truss community has a cohesive structure according to our analysis in Section 2. For example, the graph in Figure 1 is not a valid $k$-truss community containing $q$ for $k \geq 3$, as the edge $(q, s_4)$ is not in any of the triangles.
2. **Fewer parameters**. Our community model only needs to specify the trussness value $k$. In addition, a $(k+1)$-truss community is contained in a $k$-truss community. Thus by using different $k$ values for community queries, we can get a hierarchical community structure of a query vertex.
3. **Polynomial time algorithm**. There exist polynomial time algorithms for computing the $k$-truss subgraphs [6, 34], which make the $k$-truss community model computationally tractable and efficient.

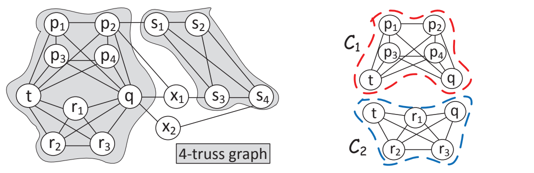
Simply searching $k$-truss community by its definition may incur a large number of wasteful edge accesses as shown in Section 3.2. Thus the key to efficient $k$-truss community query processing is to design an effective index. Towards this goal, we first apply an efficient truss decomposition algorithm [34] on a graph $G$ which computes the $k$-truss subgraphs for all $k$ values. Then we design a novel and elegant index structure, called TCP-Index, to index the pre-computed $k$-truss subgraphs. The TCP-Index preserves the trussness value and the triangle adjacency relationship in a compact tree-shape index, and supports the query of $k$-truss community in *linear time* with respect to the community size, which is optimal. Interestingly, the $k$-truss community model can be generalized using other dense subgraph definitions, such as the $k$-core community and the $k$-edge-connected component community. The TCP-Index and the proposed query algorithm can be extended to handle the above models.

In real-world networks, vertices and edges can be frequently inserted or deleted. Thus we study $k$-truss community search in dynamic graphs in Section 4. We present a theoretical analysis to identify the scope in a graph that is affected by edge insertion/deletion. Specifically, we derive a tight upper bound of the trussness for a newly inserted edge, which allows us to precisely identify the affected region with a light cost. Then we design efficient algorithms to update the trussness value and the TCP-Index in the affected region. The incremental update algorithms effectively support querying $k$-truss community in highly dynamic graphs.

On top of the conference version [17], we further investigate the problem of $k$-truss community search in massive graphs in Section 5, where the entire graph cannot fit in main memory. The motivation is that many big graphs in real world are too large to reside entirely in the main memory. For instance, one web graph crawled by WebBase crawler in 2001 consists of 115.5 million nodes and 1.7 billion edges [2], which is difficult to fit in the main memory of an ordinary machine. In the literature, there exist I/O-efficient algorithms for truss decomposition [34]. However, to the best of our knowledge, *I/O-efficient algorithms for k-truss community search in massive graphs* have not been studied yet. The challenges lie in developing I/O-efficient solutions for TCP-Index construction and online $k$-truss community search. Specifically, the structure of TCP-Index for a graph or even for a neighborhood induced subgraph may not fit in main memory. We develop several novel strategies to tackle the memory capacity issue. We adopt the widely used semi-external model by assuming that all nodes can be stored in memory while the edges are stored on disk. For example, for the graph with 115.5 million nodes and 1.7 billion edges, it only requires 5.5 GB memory for running our semi-external algorithms, which is affordable for most PCs nowadays. For TCP-Index construction, a straightforward approach is to generate the weighted neighborhood graphs and sort the weighted edges by external sorting on disk. However, this approach may incur large I/O costs by frequently moving edges between disk and main memory. To reduce I/O costs, we propose an I/O-efficient algorithm, which does not use external sorting and achieves a low I/O complexity. Moreover, we develop an I/O-efficient algorithm for $k$-truss community search based on two elegant data structures of bitmap and circular queue for reducing I/O costs.

We conduct extensive experimental studies on large real-world networks and have the following findings. First, $k$-truss community search using the TCP-Index is highly efficient in all networks. The query time is from one millisecond for low degree query vertices to a few seconds for high degree query vertices which have large and dense communities. The TCP-Index is very compact and can be constructed very efficiently. Second, the TCP-Index can be updated in milliseconds given an edge insertion/deletion. Thus it is highly efficient to support the $k$-truss community search in dynamic graphs. Third, our semi-external algorithms are particularly efficient on real-world networks in terms of both index construction and query processing. For the WebBase graph with 115.5 million nodes and 1.7 billion edges, our I/O-efficient algorithm takes 11.8 hours for index construction. The query processing takes around one millisecond for low degree query vertices and 0.1 – 1 second for high degree query vertices. Last but not least, we evaluate the quality of the discovered communities on two social networks with ground-truth communities and a scientific collaboration network. The results show that our community model can find cohesive and meaningful communities of a query vertex.

The rest of this paper is organized as follows. We formulate the $k$-truss community search problem in Section 2. We design a novel TCP-Index and an efficient $k$-truss community search algorithm in a static graph in Section 3. We further study how to maintain the TCP-Index for query processing in a dynamic graph in Section 4. We investigate the problem of $k$-truss community search using a semi-external model, and develop



(a) Graph G (b) Two 4-truss communities for q

**Fig. 2** k-truss community example

I/O-efficient algorithms in Section 5. Extensive experimental results on large real-world networks are reported in Section 6. We discuss related work in Section 7 and conclude this paper in Section 8.

## 2 K-Truss Community

### 2.1 Problem Definition

We consider an undirected, unweighted simple graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges. We denote the set of neighbors of a vertex $v$ by $N(v)$, i.e., $N(v) = \{u \in V : (v, u) \in E\}$, and the degree of $v$ by $d(v) = |N(v)|$. We use $d_{max}$ to denote the maximum vertex degree in $G$.

A triangle in $G$ is a cycle of length 3. Let $u, v, w \in V$ be the three vertices on the cycle, and we denote this triangle by $\triangle_{uvw}$. Then the *support* of an edge is defined as follows.

**Definition 1 (Support)** The support of an edge $e(u, v) \in E$ in $G$, denoted by $sup(e, G)$, is defined as $|\{\triangle_{uvw} : w \in V\}|$. When the context is obvious, we replace $sup(e, G)$ by $sup(e)$.

If an edge $e$ is contained in a triangle $\triangle$, we denote it by $e \in \triangle$. Now we give the definitions of *triangle adjacency* and *triangle connectivity* below.

**Definition 2 (Triangle Adjacency)** Given two triangles $\triangle_1, \triangle_2$ in $G$, they are adjacent if $\triangle_1$ and $\triangle_2$ share a common edge, which is denoted by $\triangle_1 \cap \triangle_2 \neq \emptyset$.

**Definition 3 (Triangle Connectivity)** Given two triangles $\triangle_s, \triangle_t$ in $G$, $\triangle_s$ and $\triangle_t$ are triangle connected, if there exist a series of triangles $\triangle_1, \ldots, \triangle_n$ in $G$, where $n \geq 2$, such that $\triangle_1 = \triangle_s$, $\triangle_n = \triangle_t$ and for $1 \leq i < n$, $\triangle_i \cap \triangle_{i+1} \neq \emptyset$.

For the graph $G$ in Figure 2(a), $e(q, p_4)$ is contained in $\triangle_{qp_3p_4}$ and $\triangle_{qp_2p_4}$, thus its support $sup(e(q, p_4)) = 2$. $\triangle_{qp_3p_4}$ and $\triangle_{qp_2p_4}$ are triangle adjacent as they share a common edge $e(q, p_4)$. $\triangle_{tp_3p_4}$ and $\triangle_{qp_2p_4}$ are triangle connected through $\triangle_{qp_3p_4}$ in $G$.

On the basis of definitions of support and triangle connectivity, we define a *k-truss community* as follows.

**Definition 4 (K-Truss Community)** Given a graph $G$ and an integer $k \geq 2$, $G'$ is a $k$-truss community, if $G'$ satisfies the following three conditions:

(1) K-Truss. $G'$ is a subgraph of $G$, denoted as $G' \subseteq G$, such that $\forall e \in E(G')$, $sup(e, G') \geq k - 2$;

(2) Edge Connectivity. $\forall e_1, e_2 \in E(G')$, $\exists \triangle_1, \triangle_2$ in $G'$ such that $e_1 \in \triangle_1$, $e_2 \in \triangle_2$, then either $\triangle_1 = \triangle_2$, or $\triangle_1$ is triangle connected with $\triangle_2$ in $G'$;
(3) Maximal Subgraph. $G'$ is a maximal subgraph satisfying conditions (1) and (2). That is, $\nexists G'' \subseteq G$, such that $G' \subset G''$, and $G''$ satisfies conditions (1) and (2).

Actually, the largest subgraph satisfies condition (1) is exactly the $k$-truss definition used in literature [6,34]. However, the $k$-truss condition itself is insufficient to define a cohesive and meaningful community due to the following two reasons. First, a $k$-truss subgraph can be disconnected, thus does not represent a cohesive community. For example, the two shaded regions in Figure 2(a) form the 4-truss subgraph of $G$, which is obviously disconnected. So this 4-truss subgraph does not correspond to a meaningful community. Second, for a fixed $k$ value, any vertex can belong to at most one $k$-truss subgraph. This cannot deal with a common scenario that a user can participate in multiple communities.

With these considerations, we impose the edge connectivity requirement in condition (2) to ensure the discovered communities are connected and cohesive. The rationale is that, a triangle represents the strong and stable relationship among three vertices. If any two edges in a subgraph are reachable from each other through a series of adjacent triangles, the subgraph must be connected, and have a cohesive structure among all involved vertices. This definition also allows a vertex to participate in multiple communities.

*Example 1* Two 4-truss communities containing vertex $q$ are shown in Figure 2(b) as $C_1$ and $C_2$, respectively. We can verify that every edge in $C_1$ is contained in at least two triangles in $C_1$, any two edges in $C_1$ are reachable through adjacent triangles, and $C_1$ is maximal. Thus $C_1$ is a 4-truss community. These properties also hold for another 4-truss community $C_2$. As the edges in $C_1$ cannot reach the edges in $C_2$ through adjacent triangles, $C_1$ and $C_2$ cannot merge as one large community. This is very reasonable, as there is no connection between the two vertex sets $\{p_1, p_2, p_3, p_4\}$ and $\{r_1, r_2, r_3\}$. In addition, we can see that vertices $q$ and $t$ participate in both communities.

**Problem Definition**. The problem of **k-truss community search** studied in this paper is defined as follows. Given a graph $G(V, E)$, a query vertex $v_q \in V$ and an integer $k \geq 2$, find all $k$-truss communities containing $v_q$. We also study $k$-truss community search in dynamic graphs, where vertices and edges are frequently inserted or deleted. Moreover, we further investigate $k$-truss community search over massive graphs which cannot entirely fit in memory using a semi-external model.

## 2.2 Why K-Truss Community?

Compared with the $\alpha$-adjacency $\gamma$-quasi-$k$-clique [7] community model, $k$-truss community model has three significant advantages: stronger guarantee on cohesive structure, fewer parameters and lower computational cost. These nice properties, which are inherited from the $k$-truss subgraph [6], not only lead to the discovery of more cohesive and meaningful communities, but also enable the design of more efficient, scalable and easier-to-use algorithms for community search. We present these properties in the following.

**Bounded Diameter in K-Truss Community**. The diameter of a $k$-truss community $C$ with $|C|$ vertices is no larger than $\lfloor \frac{2|C|-2}{k} \rfloor$ [6]. This property guarantees that the shortest distance between any two vertices in a community is bounded, which has been considered as an important feature of a good community in [10].

Consider the 4-truss community $C_1$ in Figure 2(b) as an example. The diameter of $C_1$ is 2, which is the same as the diameter upper bound $\lfloor \frac{2\times 6-2}{4} \rfloor = 2$.

**(K-1)-Edge-Connected Graph**. A graph is $(k-1)$-edge-connected if it remains connected whenever fewer than $k-1$ edges are removed [14]. A $k$-truss community is $(k-1)$-edge-connected [6]. This property ensures high connectivity of a community, which has been proposed as a criterion of a good community in [15].

In contrast, the $\gamma$-quasi-$k$-clique is not $(k-1)$-edge-connected for $\gamma < 1$. The 0.8-quasi-7-clique in Figure 1 becomes disconnected when one edge is removed.

**Fewer Parameters**. In $k$-truss community model, we only need to specify the trussness value $k$, which controls or affects the diameter, edge connectivity, and edge support in a community. In contrast, $\alpha$-adjacency $\gamma$-quasi-$k$-clique model uses three parameters, the adjacency parameter $\alpha$, density $\gamma$ and clique size $k$. Although having more parameters may give more leverage on the properties of the community, it is much more difficult to choose proper values for different parameters.

**Polynomial Time Complexity**. There exist polynomial time algorithms [6,34] for computing $k$-truss subgraphs. By applying such algorithms, we can compute $k$-truss subgraphs for all $k$. The pre-computed results enable us to design compact index structures and efficient algorithms for querying $k$-truss communities. In contrast, finding $\gamma$-quasi-$k$-cliques has been proven to be NP-hard [7], which imposes a severe computational bottleneck.

## 2.3 Variations of K-Truss Community

The $k$-truss community can be extended to several interesting variants.

**Densest K-Truss Community**. It is interesting to discover the densest $k$-truss community containing a

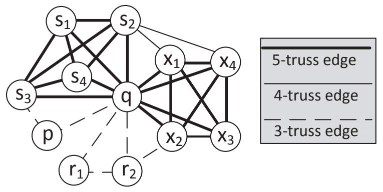


**Fig. 3** An example graph for k-truss community search

user $q$, that is, a $k$-truss community containing $q$ that maximizes the trussness value $k$.

**Most Diverse Communities**. It is interesting to know how diverse the social contexts of a user $q$ are, where a community represents a distinct social context. For a set of $k$-truss communities containing $q$ as $\{C_1, \ldots, C_l\}$, the community diversity is measured by the entropy of the community vertex size distribution: $Div(q,k) = -\sum_{1\le i\le l} \frac{|C_i|}{\sum_{1\le j\le l}|C_j|} \log \frac{|C_i|}{\sum_{1\le j\le l}|C_j|}$. By choosing $k$ which maximizes the diversity, i.e., $k = \arg\max_k Div(q,k)$, we can find the most diverse communities.

**Other Community Models**. The community model can be generalized using other dense subgraph definitions, such as the $k$-core community and $k$-edge-connected component community [18,27]. Take $k$-core as an example. A graph $G' \subseteq G$ is a $k$-core community if it satisfies the following three conditions:

1. $G'$ is a $k$-core graph;
2. $G'$ satisfies the triangle-based edge connectivity requirement;
3. $G'$ is a maximal subgraph satisfying the above two conditions.

The techniques for $k$-truss community search proposed in this paper can be extended to solve the above variant forms. Accordingly, the index construction for $k$-truss community (in Algorithm 3) needs to be modified for constructing the TCP-Index for $k$-core community. Specifically, for an edge $(x,y) \in E(G)$, we define $\tau((x,y))$ as the largest $k$ value that $(x,y)$ belongs to a $k$-core community. The core value for every edge can be precomputed using a modified core decomposition method on $G$ in step 1. The rest of Algorithm 3 remains unchanged. For querying $k$-core community using the TCP-Index, the query processing algorithm (in Algorithm 4) remains unchanged.

## 3 Querying K-Truss Community

In this section, we study how to process a $k$-truss community query. We first design a simple $k$-truss index which is then proven to incur unnecessary computational overhead for query processing. Then we design a compact and elegant structure, called *Triangle Connectivity Preserved Index* (TCP-Index), and a highly efficient algorithm to process a $k$-truss community query.

### 3.1 Subgraph and Edge Trussness

We define the trussness of subgraphs and edges as follows.

**Algorithm 1** Truss Decomposition

**Input:** $G = (V, E)$
**Output:** $\tau(e)$ for each $e \in E$

```
1:  $k \leftarrow 2$;
2:  compute $sup(e)$ for each edge $e \in E$;
3:  sort all the edges in ascending order of their support;
4:  while($\exists e$ such that $sup(e) \le (k-2)$)
5:      let $e = (u,v)$ be the edge with the lowest support;
6:      assume, w.l.o.g, $deg(u) \le deg(v)$;
7:      for each $w \in N(u)$ and $(v,w) \in E$ do
8:          $sup((u,w)) \leftarrow sup((u,w)) - 1$;
            $sup((v,w)) \leftarrow sup((v,w)) - 1$;
9:          reorder $(u,w)$ and $(v,w)$ according to their new support;
10:     $\tau(e) \leftarrow k$, remove $e$ from $G$;
11: if(not all edges in $G$ are removed)
12:     $k \leftarrow k+1$;
13:     goto Step 4;
14: return $\{\tau(e) | e \in E\}$;
```

**Definition 5 (Subgraph Trussness)** The trussness of a subgraph $H \subseteq G$ is the minimum support of an edge in $H$, denoted by $\tau(H) = \min\{sup(e,H)+2 : e \in E(H)\}$.

**Definition 6 (Edge Trussness)** The trussness of an edge $e \in E(G)$ is defined as $\tau(e) = \max_{H\subseteq G}\{\tau(H) : e \in E(H)\}$.

Such a subgraph $H$ which defines $\tau(e)$ is denoted as $H^e$. It follows that $\tau(H^e) = \tau(e)$. For an edge $e$ and $2 \le k \le \tau(e)$, we denote the $k$-truss community containing $e$ as $H_k^e$, which is unique in the sense that no other $k$-truss community contains $e$. We use $k_{gmax}$ to denote the maximum trussness of any edge in $G$.

Consider the graph in Figure 3 as an example. The trussness of the edge $e(s_1, s_2)$ is $\tau(e) = 5$, and the subgraph $H^e$ is the 5-clique on the vertex set $\{q, s_1, s_2, s_3, s_4\}$.

### 3.2 A Simple K-Truss Index

We design a simple $k$-truss index and propose an algorithm for $k$-truss community search based on the index.

**K-Truss Index Construction**. We first apply a truss decomposition algorithm [34] on $G$, which computes the trussness of each edge. For the self-completeness of this paper, the truss decomposition algorithm [34] is outlined in Algorithm 1. After the initialization, for each $k$ starting from $k = 2$, the algorithm iteratively removes the lowest support edge $e(u,v)$ if $sup(e) \le k-2$. We assign the trussness of the removed edge as $\tau(e) = k$. Upon the removal of $e$, we also decrement the support of all other edges that form a triangle with $e$, and reorder them according to their new support. This process continues until all edges with support less than or equal to $(k-2)$ are removed. In this way, we compute the trussness of all edges in $G$.

**Algorithm 2** Query Processing Using K-Truss Index

**Input:** $G = (V, E)$, an integer $k$, query vertex $v_q$
**Output:** $k$-truss communities containing $v_q$

```
1: $visited \leftarrow \emptyset$; $l \leftarrow 0$;
2: for $u \in N(v_q)$ do
3:     if $\tau((v_q, u)) \geq k$ and $(v_q, u) \notin visited$
4:         $l \leftarrow l + 1$; $C_l \leftarrow \emptyset$; $Q \leftarrow \emptyset$;
5:         $Q.push((v_q, u))$; $visited \leftarrow visited \cup \{(v_q, u)\}$;
6:         while $Q \neq \emptyset$
7:             $(x, y) \leftarrow Q.pop()$; $C_l \leftarrow C_l \cup \{(x, y)\}$;
8:             for $z \in N(x) \cap N(y)$ do
9:                 if $\tau((x, z)) \geq k$ and $\tau((y, z)) \geq k$
10:                    if $(x, z) \notin visited$
11:                        $Q.push((x, z))$;
12:                        $visited \leftarrow visited \cup \{(x, z)\}$;
13:                    if $(y, z) \notin visited$
14:                        $Q.push((y, z))$;
15:                        $visited \leftarrow visited \cup \{(y, z)\}$;
16: return $\{C_1, \ldots, C_l\}$;
```

For each vertex $v \in V$, we sort its neighbors $N(v)$ in descending order of the edge trussness $\tau(e(u, v))$ for $u \in N(v)$. For each distinct trussness value $k \geq 2$, we mark the position of the first vertex $u$ in the sorted adjacency list where $\tau(e(u, v)) = k$. This supports efficient retrieval of $v$'s incident edges with a certain trussness value. We also use a hash table to keep all edges and their trussness values. This is the simple $k$-truss index.

**Query Processing**. Algorithm 2 outlines the procedure to process a $k$-truss community query based on the simple index. Given an integer $k$ and a query vertex $v_q$, the algorithm checks every incident edge on $v_q$ to search $k$-truss communities. If there exists an unvisited edge $(v_q, u)$ with $\tau((v_q, u)) \geq k$, $(v_q, u)$ is the seed edge to form a new community $C_l$. By definition, all the other edges in $C_l$ can be reached from $(v_q, u)$ through adjacent triangles. So we push $(v_q, u)$ into a queue $Q$ and perform a BFS traversal to search for other edges for expanding $C_l$, i.e., edges which have trussness no less than $k$ and form triangles with edges already in $C_l$ (line 6-15). When $Q$ becomes empty, all edges in $C_l$ have been found. Then the algorithm checks the next unvisited incident edge of $v_q$ for forming a new community $C_{l+1}$. This process iterates until all incident edges of $v_q$ have been processed. Finally, a set of $k$-truss communities containing $v_q$ are returned.

The correctness of Algorithm 2 is apparent since the algorithm essentially computes $k$-truss communities by definition, that is, exploring triangle connected edges with trussness no less than $k$ in a BFS manner. We show the complexity of the simple $k$-truss index construction and query processing by Algorithm 2 as follows.

**Theorem 1** *The construction of the simple $k$-truss index takes $O(\sum_{(u,v)\in E} \min\{d(u), d(v)\})$ time and $O(m)$ space. The index size is $O(m)$. Algorithm 2 takes $O(d_{Amax}|Ans|)$ time to process one query, where $Ans = C_1 \cup \ldots \cup C_l$ is the union of the produced $k$-truss communities, $|Ans|$ is the edge number in $Ans$ and $d_{Amax}$ is the maximum vertex degree in $Ans$.*

The proof can be found in [17].

*Example 2* Suppose we want to query the 4-truss communities containing vertex $q$ in the graph in Figure 3. Algorithm 2 first visits edge $(q, s_1)$ with $\tau((q, s_1)) = 5 \geq 4$, and adds it into $Q$. The algorithm pops $(q, s_1)$ from $Q$ and inserts it into a new community $C_1$. Then the algorithm checks the common neighbors of $q$ and $s_1$ and the edges between them. Consider a common neighbor $s_2$ as an example. As $\tau((q, s_2)) \geq 4$ and $\tau((s_1, s_2)) \geq 4$, both edges $(q, s_2)$ and $(s_1, s_2)$ are then inserted into $C_1$ and also pushed into $Q$ for further expansion. This BFS expansion process continues until $Q$ is empty and the 4-truss community $C_1$ is the induced subgraph by the vertex set $\{q, s_1, s_2, s_3, s_4, x_1, x_2, x_3, x_4\}$.

## 3.3 A Novel TCP-Index

### *3.3.1 Limitations of Simple K-Truss Index*

Algorithm 2 has two drawbacks in its query processing mechanism by using the simple $k$-truss index. Specifically, in line 8-15, for any edge $(x, y)$ that has already been included in $C_l$, the algorithm needs to access adjacent edges $(x, z)$ and $(y, z)$ for each common neighbor $z$ of $x$ and $y$. The following two cases lead to unnecessary and excessive computational overhead.

1. **Unnecessary access of disqualified edges**: If $\tau((x, z)) < k$ or $\tau((y, z)) < k$, $(x, z), (y, z)$ will not be included in $C_l$, thus accessing and checking such disqualified edges become a waste.
2. **Repeated access of qualified edges**: For each edge $(u, v)$ in $C_l$, it is accessed at least $2(k-2)$ times in the BFS traversal, which is a huge waste. This is because $\tau((u, v)) \geq k$, $(u, v)$ is contained in at least $(k-2)$ triangles by definition. For each such triangle denoted as $\triangle_{uvw}$, $(u, v)$ will be accessed twice when we do BFS expansion from the other two edges $(u, w), (v, w)$. It follows that the query time of Algorithm 2 is lower bounded by $\Omega(k|Ans|)$.

Considering these drawbacks, we design a novel Triangle Connectivity Preserved Index (TCP-Index), which avoids computational problems in Algorithm 2. Remarkably, TCP-Index supports $k$-truss community query in $O(|Ans|)$ time, which is essentially optimal. Meanwhile, TCP-Index can be constructed in $O(\sum_{(u,v)\in E} \min\{d(u), d(v)\})$ time and stored in $O(m)$ space, which has exactly the same complexity as simple $k$-truss index.

### *3.3.2 TCP-Index Construction*

We first make some observations from the example in Figure 3.

**Observation 1:** *Consider $\triangle_{pqs_3}$ in which the three edge trussness values are 5, 3, and 3. Then $\triangle_{pqs_3}$ can appear in a 3-truss community, but not in 4- or 5-truss communities, due to the two 3-truss edges. To generalize, a triangle $\triangle_{xyz}$ can appear only in k-truss communities where $k \le \min\{\tau((x,y)), \tau((x,z)), \tau((y,z))\}$.*

**Observation 2:** *Consider a subgraph in Figure 4(a) which is extracted from Figure 3. By definition, vertices $x_1, x_2, x_3$ belong to the same 5-truss community containing q, as each involved edge is 5-truss, and $\triangle_{qx_1x_2}$ and $\triangle_{qx_1x_3}$ are adjacent via edge $(q, x_1)$. Thus we can use a compact representation for vertex q as depicted in solid line in Figure 4(b), which preserves the trussness and adjacency information for community search. Note that there is no need to include edge $(x_2, x_3)$, as the tree-shape structure clearly indicates that $x_2, x_3$ belong to the same 5-truss community by triangle adjacency.*

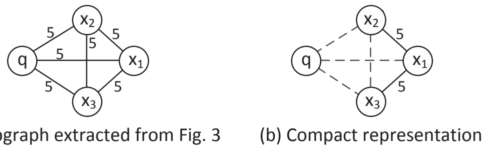

(a) Subgraph extracted from Fig. 3
(b) Compact representation

**Fig. 4** Compact representation of a community with $q$

**Observation 3:** *From Figure 3, we can see two 5-truss communities $\{q, x_1, x_2, x_3, x_4\}, \{q, s_1, s_2, s_3, s_4\}$ involving vertex q are contained in the 4-truss community $\{q, x_1, x_2, x_3, x_4, s_1, s_2, s_3, s_4\}$, which is in turn contained in the 3-truss community, which is the whole graph.*

Based on the above observations, we are ready to construct the TCP-Index using Algorithm 3. For each vertex $x \in V$, we build a graph $G_x$, where $V(G_x) = N(x)$, and $E(G_x) = \{(y,z)|(y,z) \in E(G), y, z \in N(x)\}$. For each edge $(y,z) \in E(G_x)$, we assign a weight $w(y,z) = \min\{\tau((x,y)), \tau((x,z)), \tau((y,z))\}$, which indicates that $\triangle_{xyz}$ can appear only in $k$-truss community where $k \le w(y,z)$ based on Observation 1. The TCP-Index for vertex $x$ is a tree structure, denoted as $\mathcal{T}_x$, which is initialized to be the node set $N(x)$. Then in line 8-12, for each $k$ from the largest weight $k_{max}$ to 2, we iteratively collect the set of edges $S_k \subseteq E(G_k)$ whose weight is $k$. For each $(y,z) \in S_k$, if $y, z$ are still in different components of $\mathcal{T}_x$, we add $(y,z)$ with a weight $w(y,z)$ into $\mathcal{T}_x$. Essentially, $\mathcal{T}_x$ is the maximum spanning forest of $G_x$. The trees $\mathcal{T}_x$ for all $x \in V$ form the TCP-Index of $G$.

*Example 3* Figure 5 shows the TCP-Index for vertex $q$ in the graph in Figure 3. $\mathcal{T}_q$ is initialized to be $N(q)$. Figure 5(a) shows the tree structure when we add edges whose weights are 5. According to Observation 2, when the edges $(x_1, x_2)$ and $(x_1, x_3)$ are added into $\mathcal{T}_q$, the edge $(x_2, x_3)$ will not be added into $\mathcal{T}_q$, as $x_2, x_3$ are already connected in $\mathcal{T}_q$ and we know that $x_2, x_3$ belong to the same 5-truss community by triangle adjacency. This is essential to keep $\mathcal{T}_q$ as a compact forest. The

**Algorithm 3** TCP-Index Construction

**Input:** $G = (V, E)$
**Output:** TCP-Index $\mathcal{T}_x$ for each $x \in V$

```
 1: Perform truss decomposition for $G$;
 2: for $x \in V$ do
 3:     $G_x \leftarrow \{(y,z)|y,z \in N(x), (y,z) \in E\}$;
 4:     for $(y,z) \in E(G_x)$ do
 5:         $w(y,z) \leftarrow \min\{\tau((x,y)), \tau((x,z)), \tau((y,z))\}$;
 6:     $\mathcal{T}_x \leftarrow N(x)$;
 7:     $k_{max} \leftarrow \max\{w(y,z)|(y,z) \in E(G_x)\}$ ;
 8:     for $k \leftarrow k_{max}$ to 2 do
 9:         $S_k \leftarrow \{(y,z)|(y,z) \in E(G_x), w(y,z) = k\}$;
10:         for $(y,z) \in S_k$ do
11:             if $y, z$ are in different components in $\mathcal{T}_x$
12:                 add $(y,z)$ with weight $w(y,z)$ in $\mathcal{T}_x$;
13: return $\{\mathcal{T}_x|x \in V\}$;
```

complete TCP-Index for $q$ is shown in Figure 5(c). According to the community containment relationship in Observation 3, it is sufficient to use a single structure $\mathcal{T}_q$ for all trussness levels from $k_{max}$ to 2.

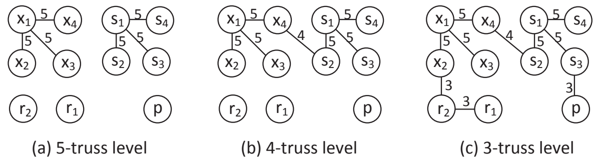


**Fig. 5** TCP-Index construction of vertex $q$

**Theorem 2** *The* TCP-Index *of graph G can be constructed in $O(\sum_{(u,v)\in E} \min\{d(u), d(v)\})$ time and $O(m)$ space by Algorithm 3. The index size is $O(m)$.*

The proof can be found in [17].

*Remark 1* $O(\sum_{(u,v)\in E} \min\{d(u), d(v)\}) \subseteq O(\rho m)$ holds according to [5], where $\rho$ is the arboricity of a graph $G$. $\rho \le \min\{\lceil\sqrt{m}\,\rceil, d_{max}\}$ holds for any graph. Thus the TCP-Index construction time is $O(\sum_{(u,v)\in E} \min\{d(u), d(v)\}) \subseteq O(\rho m) \subseteq O(m^{1.5})$.

### *3.3.3 Query Processing Using TCP-Index*

We first illustrate query processing through an example, before we formally present the algorithm. According to the design of the TCP-Index, if two vertices are connected through a series of edges with weight $\ge k$ in $\mathcal{T}_x$ for $x \in V$, these two vertices belong to the same $k$-truss community via adjacent triangles. Consider $\mathcal{T}_q$ in Figure 5(c). As $x_2, x_3$ are connected through two edges with weight 5, they belong to the same 5-truss community. Thus we first define the *k-level connected vertex set* on a tree $\mathcal{T}_x$ to find all such vertices that belong to a $k$-truss community.

**Definition 7 (K-Level Connected Vertex Set)** For $x \in V$ and $y \in N(x)$, we use $V_k(x,y)$ to denote the set of vertices which are connected with $y$ through edges of weight $\ge k$ in $\mathcal{T}_x$. We regard $y$ also belongs to this set, i.e., $y \in V_k(x,y)$.

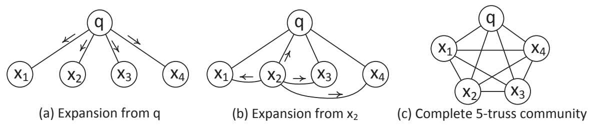


**Fig. 6** 5-truss community query on $q$ using TCP-Index

*Example 4* If we want to query 5-truss communities containing $q$, we first visit an incident edge on $q$, $(q, x_1)$ where $\tau((q, x_1)) = 5$. From $\mathcal{T}_q$ we retrieve the vertex set $V_5(q, x_1) = \{x_1, x_2, x_3, x_4\}$ as they are connected through edges with weight 5. According to Observation 2, these four vertices belong to the same 5-truss community with $q$. As $V_5(q, x_1) \subset N(q)$, we can construct part of the community as shown in Figure 6(a).

At this stage, we still miss the edges between the four vertices, for example, $(x_2, x_3), (x_3, x_4)$, etc. This is because $\mathcal{T}_q$ which is a spanning forest does not keep all the edges between the vertices. To fully recover all the edges in the 5-truss community, for each vertex $x_i \in V_5(q, x_1)$, we "reverse" the edge $(q, x_i)$ to $(x_i, q)$, then further expand the community in $x_i$'s neighborhood. Take vertex $x_2$ as an example. We reverse $(q, x_2)$ to $(x_2, q)$ and then query $x_2$'s index $\mathcal{T}_{x_2}$ to get the vertex set $V_5(x_2, q) = \{q, x_1, x_3, x_4\}$, as $x_1, x_3, x_4$ are connected with $q$ in $\mathcal{T}_{x_2}$. Then we can obtain the edges between $x_2$ and every vertex in $V_5(x_2, q)$. After this, the community is shown in Figure 6(b). Similarly, we perform the reverse operation for each vertex $x_1, x_3, x_4$ and get the complete 5-truss community in Figure 6(c). We can observe that in this search process, each edge in a community is accessed exactly twice, for example, accessing $(q, x_2)$ from $\mathcal{T}_q$ and $(x_2, q)$ from $\mathcal{T}_{x_2}$.

Algorithm 4 outlines the procedure of query processing using the TCP-Index. Similar to Algorithm 2, Algorithm 4 computes the $k$-truss communities for a query vertex $v_q$ by expanding from each incident edge on $v_q$ in a BFS manner. If there exists an unvisited edge $(v_q, u)$ with $\tau((v_q, u)) \geq k$, $(v_q, u)$ is the seed edge to form a new community $C_l$ (line 2-4). Then the algorithm performs a BFS traversal using a queue $Q$ in line 5-14. For an unvisited edge $(x, y)$, it searches the vertex set $V_k(x, y)$ from $\mathcal{T}_x$. The procedure of computing $V_k(x, y)$ is listed in line 16-17. For each $z \in V_k(x, y)$, the edge $(x, z)$ is added into $C_l$. Then we perform the reverse operation, i.e., if $(z, x)$ is not visited yet, it is pushed into $Q$ for $z$-centered community expansion using $\mathcal{T}_z$. Note that $(z, x)$ and $(x, z)$ are considered different here. When $Q$ becomes empty, all edges in $C_l$ have been found. The process iterates until all incident edges of $v_q$ have been processed. Finally, a set of $k$-truss communities containing $v_q$ are returned.

We prove the correctness of Algorithm 4 as follows.

**Lemma 1** *Given a query vertex $x \in V$ and an integer $k$, Algorithm 4 correctly computes all $k$-truss communities containing $x$.*

**Algorithm 4** Query Processing Using TCP-Index

**Input:** $G = (V, E)$, an integer $k$, query vertex $v_q$
**Output:** $k$-truss communities containing $v_q$

```
1:  $visited \leftarrow \emptyset$; $l \leftarrow 0$;
2:  for $u \in N(v_q)$ do
3:      if $\tau((v_q, u)) \geq k$ and $(v_q, u) \notin visited$
4:          $l \leftarrow l + 1$; $C_l \leftarrow \emptyset$; $Q \leftarrow \emptyset$;
5:          $Q.push((v_q, u))$;
6:          while $Q \neq \emptyset$
7:              $(x, y) \leftarrow Q.pop()$;
8:              if $(x, y) \notin visited$
9:                  compute $V_k(x, y)$;
10:                 for $z \in V_k(x, y)$ do
11:                     $visited \leftarrow visited \cup \{(x, z)\}$;
12:                     $C_l \leftarrow C_l \cup \{(x, z)\}$;
13:                     if the reverse edge $(z, x) \notin visited$
14:                         $Q.push((z, x))$;
15: return $\{C_1, \cdots, C_l\}$;
16: Procedure compute $V_k(x, y)$
17: return $\{z | z$ is connected with $y$ in $\mathcal{T}_x$ through edges of
        weight $\geq k\}$;
```

The proof can be found in [17].

**Theorem 3** *The time complexity of Algorithm 4 is $O(|Ans|)$, where $Ans = C_1 \cup \ldots \cup C_l$ is the union of the produced $k$-truss communities and $|Ans|$ is the number of edges in $Ans$.*

The proof can be found in [17].

**Complexity Comparison**. By using the TCP-Index and the simple $k$-truss index, each edge in a $k$-truss community is accessed *exactly twice versus at least* $2(k-2)$ *times.* In addition, the TCP-Index successfully avoids the unnecessary access of disqualified edges whose trussness is less than $k$. These are the key reasons to explain the query time difference between Algorithms 4 and 2, i.e., $O(|Ans|)$ versus $O(d_{Amax}|Ans|)$. Meanwhile, the TCP-Index construction has exactly the same time and space complexity as the simple $k$-truss index.

## 4 Querying K-Truss Community in Dynamic Graphs

In this section, we study $k$-truss community search in dynamic graphs where vertices and edges are inserted or deleted. We mainly focus on edge insertion and deletion, because vertex insertion/deletion can be regarded as a sequence of edge insertions/deletions preceded/followed by the insertion/deletion of an isolated vertex.

Consider the insertion of edge $e_0(x, y)$ into $G$ which leads to a set of new triangles $\{\triangle_{xyz} : z \in N(x) \cap N(y)\}$. Due to a new $\triangle_{xyz}$, the support of both edges $(x, z), (y, z)$ increases by 1. This may increase the subgraph trussness, $\tau(H)$, for $H \subseteq G$ which contains $\triangle_{xyz}$ since $\tau(H) = \min\{sup(e, H) : e \in E(H)\}$. It may in turn increase the trussness of those edges contained in

$H$, as $\tau(e) = \max_{H \subseteq G}\{\tau(H) : e \in E(H)\}$, however, such edges may not necessarily be incident on vertices $x$ or $y$. Edge deletion has a similar effect on decreasing edge trussness. To handle the edge trussness update efficiently, the key is to identify the affected region in the graph precisely. Thus, in Section 4.1 we present a theoretical analysis to define the scope that an edge insertion/deletion may affect. We design algorithms for updating the edge trussness and the TCP-Index in Sections 4.2 and 4.3, respectively.

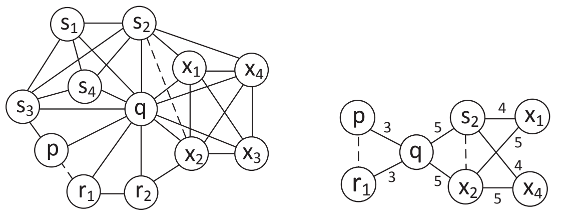
### 4.1 Scope of Affected Edges

We use $\tau(e)$ and $\hat{\tau}(e)$ to represent the trussness of an edge $e$ before and after an edge insertion/deletion respectively. We have the following three properties.

Rule 1: If $e_0$ is inserted into $G$ with $\hat{\tau}(e_0) = l$, then $\forall e \in E(G)$ with $\tau(e) \geq l$, $\hat{\tau}(e) = \tau(e)$ holds.
Rule 2: If $e_0$ is deleted from $G$ with $\tau(e_0) = l$, then $\forall e \in E(G) \setminus \{e_0\}$ with $\tau(e) > l$, $\hat{\tau}(e) = \tau(e)$ holds.
Rule 3: $\forall e \in E(G) \setminus \{e_0\}$, $|\hat{\tau}(e) - \tau(e)| \leq 1$ holds.

The rationale of Rules 1 and 2 is that $e_0$ is not included in a $(l+1)$-truss subgraph. Rule 3 holds since the support of any edge changes by at most 1 with an edge insertion/deletion. In order to apply Rule 1, we need to obtain $\hat{\tau}(e_0)$ first. However, computing $\hat{\tau}(e_0)$ itself is costly. So we resort to an alternative, that is, we estimate an upper bound of $\hat{\tau}(e_0)$, denoted as $\overline{\hat{\tau}(e_0)}$, with a light cost, and apply Rule 1' instead of Rule 1.

Rule 1': If $e_0$ is inserted into $G$, then $\forall e \in E(G)$ with $\tau(e) \geq \overline{\hat{\tau}(e_0)} \geq \hat{\tau}(e_0)$, $\hat{\tau}(e) = \tau(e)$ holds.



(a) $G$ with Edge Insertion (b) Trussness Update

**Fig. 7** An example graph with edge insertion

To estimate $\overline{\hat{\tau}(e_0)}$, we define the *k-level triangles* of an edge.

**Definition 8 (K-Level Triangles)** For an edge $e(u, v)$ and $k \geq 2$, we denote the $k$-level triangles containing $e$ by $\triangle^k_{(u,v)} = \{\triangle_{uvw} : \min\{\tau((u, w)), \tau((v, w))\} \geq k\}$. The number of triangles in $\triangle^k_{(u,v)}$ is denoted by $|\triangle^k_{(u,v)}|$.

Lemma 2 gives the lower and upper bound of $\hat{\tau}(e_0)$.

**Lemma 2** *If an edge $e_0(u, v)$ is inserted into a graph, then $\hat{\tau}(e_0)$ satisfies $k_1 \leq \hat{\tau}(e_0) \leq k_2$ and $k_2 - k_1 \leq 1$, where $k_1 = \max_k\{k : |\triangle^k_{e_0}| \geq k-2\}$, $k_2 = \max_k\{k : |\triangle^{k-1}_{e_0}| \geq k-2\}$.*

The proof can be found in [17].

**Corollary 1** *$\overline{\hat{\tau}(e_0)} = \max_k\{k : |\triangle^{k-1}_{e_0}| \geq k-2\}$ is an upper bound of $\hat{\tau}(e_0)$ .*

The upper bound in Corollary 1 is extremely tight, as it may be larger than the real trussness by at most 1. For example, if we insert edge $(p, r_1)$ into the graph in Figure 7(a), we have $\triangle^3_{(p,r_1)} = \{\triangle_{qpr_1}\}$ and $k_1 = k_2 = 3$ by Lemma 2. So we get $\hat{\tau}((p, r_1)) = 3$. If we insert another edge $(s_2, x_2)$, we have $\triangle^4_{(s_2,x_2)} = \{\triangle_{s_2x_2q}, \triangle_{s_2x_2x_1}, \triangle_{s_2x_2x_4}\}$, $k_1 = 4$, $k_2 = 5$ by Lemma 2.

Due to the insertion/deletion of edge $e_0$, there are two reasons for edge $e \in E(G) \setminus \{e_0\}$ to change trussness, i.e., $\hat{\tau}(e) \neq \tau(e)$: (1) $e$ forms/breaks a triangle with $e_0$ when $e_0$ is inserted/deleted; or (2) the edges of the triangles in which $e$ lies have changed their trussness. We study the insertion case in Lemma 3.

**Lemma 3** *If an edge $e_0$ is inserted into a graph, we first assign $\tau(e_0) = \max_k\{k : |\triangle^k_{e_0}| \geq k-2\}$. Then for $e(x, y) \in E(G) \cup \{e_0\}$ with $\tau(e) = l < \overline{\hat{\tau}(e_0)}$, $e$ may have $\hat{\tau}(e) = l + 1$* ***only in two cases:***
*(1) A new triangle with edges $e, e_0$ and another $e'$ is formed, and $\min\{\tau(e_0), \tau(e')\} \geq l$ holds; or*
*(2) For $e(x, y)$, $\exists z \in N(x) \cap N(y)$, $\min\{\tau((x, z)), \tau((y, z))\} = l$ holds.*

The proof can be found in [17].

According to Lemma 3, we summarize the affected edge trussness due to edge insertion/deletion.

**Scope of Affected Edges**. We define the weight of a triangle by the minimum edge trussness in the triangle. Then the insertion/deletion of an edge $e_0$ may lead to the following trussness update.
(1) **Insertion case**. For $e \in E(G) \cup \{e_0\}$ with $\tau(e) < \overline{\hat{\tau}(e_0)}$, if $e, e_0$ and another $e'$ form a new triangle with weight $\tau(e)$, or $e$ is connected with $e_0$ through a series of adjacent triangles each with weight $\tau(e)$, then $e$ may have $\hat{\tau}(e) = \tau(e) + 1$.
(2) **Deletion case**. For $e \in E(G) \setminus \{e_0\}$ with $\tau(e) \leq \tau(e_0)$, if $e, e_0$ belong to a triangle with weight $\tau(e)$, or $e$ is connected with $e_0$ through a series of adjacent triangles each with weight $\tau(e)$ before deletion, then $e$ may have $\hat{\tau}(e) = \tau(e) - 1$.

### 4.2 Updating Edge Trussness

In this section, we propose an algorithm to update the edge trussness when an edge $e_0$ is inserted. The algorithm to handle edge deletion is similar and thus omitted. From Section 4.1 we know that only edges which have $\tau(e) < \overline{\hat{\tau}(e_0)}$, and are either in the same triangle with $e_0$ or connected to $e_0$ through adjacent triangles at the same trussness level may increase their trussness. Thus we first collect all edges in case (1) of Lemma 3 as candidates, then expand the candidate edges and examine their edge trussness level by level according to case

(2). Finally, we use a variant of truss decomposition algorithm to finalize the trussness update.

The procedure to update edge trussness given an inserted edge $e_0$ is outlined in Algorithm 5. We first insert the edge $e_0$ and compute the range of $\hat{\tau}(e_0)$ as $[k_1, k_2]$ by Lemma 2. Then the algorithm sets $\tau(e_0) = k_1$ and the maximum edge trussness $k_{max} = k_2 - 1$ by Rule 1'. In line 4-9, by case (1) of Lemma 3, we collect every edge which forms a triangle with $e_0$, and has the minimum trussness in the triangle with $\tau(e) = k \leq k_{max}$. These edges are inserted into $L_k$. Then in line 10-30, for each $k$ from $k_{max}$ to 2, the algorithm updates the trussness of edges with $\tau(e) = k$ using three steps, namely, edge expansion (line 11-20), edge eviction (line 21-28), and trussness update (line 29-30). In the edge expansion step, the algorithm expands $L_k$ by finding all edges with trussness $k$ using breadth-first search through adjacent triangles with weight $k$ by case (2) of Lemma 3 (line 14-20). Meanwhile, it computes the number of $k$-level triangles $|\triangle_e^k|$ as $s[e]$ (line 16). In the edge eviction step, the algorithm iteratively evicts edges with $s[e] \leq k-2$ from $L_k$ (line 22) until no such edges exist. After evicting an edge $e$, for each edge $e' \in L_k$ that forms a triangle of weight $k$ with $e$, $s[e']$ is decreased by 1 (line 24-28). In the trussness update step, each edge $e \in L_k$ has $s[e] \geq k-1$ and gets trussness update $\hat{\tau}(e) = k+1$ (line 29-30). Three steps can be further optimized by pushing the edge eviction operation (line 21-28) into the edge expansion step (line 11-20) after each $s[(x,y)]$ is calculated, in order to avoid expanding useless edges in an early stage. The technique is similar to the early node eviction technique for $k$-core update [30]. We omit the detailed discussion on this heuristic.

*Example 5* We update the edge trussness with the insertion of $e_0(s_2, x_2)$ in the graph shown in Figure 7(a). We first compute $k_1 = 4$, $k_2 = 5$, and assign $\tau(e_0) = 4$ and $k_{max} = 4$. Since $\tau(e_0) = k_{max}$ indicates that $e_0$ may increase its trussness, we add $(s_2, x_2)$ into $L_4$. Then the algorithm checks the edges forming new triangles with $e_0$ as shown in Figure 7(b), and adds $(s_2, x_1)$ and $(s_2, x_4)$ into $L_4$. Next the algorithm uses BFS to find the edges connected with those in $L_4$ through adjacent triangles with weight 4. As there is no such edge satisfying this condition, $L_4$ remains unchanged. Meanwhile, for each $e \in L_4$, it computes $|\triangle_e^4|$ as $s[e]$, that is, $s[(s_2, x_2)] = s[(s_2, x_1)] = s[(s_2, x_4)] = 3$ (shown in Figure 7(a)). As the three edges have $s[e] > 2$, the algorithm updates $\hat{\tau}(s_2, x_2) = \hat{\tau}(s_2, x_1) = \hat{\tau}(s_2, x_4) = 5$.

**Algorithm 5** Trussness Update with Edge Insertion

**Input:** $G = (V, E)$, the inserted edge $e_0 = (u, v)$
**Output:** Updated trussness $\hat{\tau}(e)$ for $e \in E(G) \cup \{e_0\}$

```
1:  $G.insert(e_0)$;
2:  compute $[k_1, k_2]$ for $\hat{\tau}(e_0)$ by Lemma 2;
3:  $\tau(e_0) \leftarrow k_1$; $k_{max} \leftarrow k_2 - 1$;
4:  for $k \leftarrow 2$ to $k_{max}$ do $L_k \leftarrow \emptyset$;
5:  for $w \in N(u) \cap N(v)$ do
6:      $k \leftarrow \min\{\tau((w,u)), \tau((w,v))\}$;
7:      if $k \leq k_{max}$ then
8:          if $\tau((w,u)) = k$ then $L_k \leftarrow L_k \cup \{(w,u)\}$;
9:          if $\tau((w,v)) = k$ then $L_k \leftarrow L_k \cup \{(w,v)\}$;
10: for $k \leftarrow k_{max}$ to 2 do
11:     $Q \leftarrow \emptyset$; $Q.push(L_k)$;
12:     while $Q \neq \emptyset$
13:         $(x,y) \leftarrow Q.pop()$; $s[(x,y)] \leftarrow 0$;
14:         for $z \in N(x) \cap N(y)$ do
15:             if $\tau((z,x)) < k$ or $\tau((z,y)) < k$ then continue;
16:             $s[(x,y)] \leftarrow s[(x,y)] + 1$;
17:             if $\tau((z,x)) = k$ and $(z,x) \notin L_k$ then
18:                 $Q.push((z,x))$; $L_k \leftarrow L_k \cup \{(z,x)\}$;
19:             if $\tau((z,y)) = k$ and $(z,y) \notin L_k$ then
20:                 $Q.push((z,y))$; $L_k \leftarrow L_k \cup \{(z,y)\}$;
21:     while $\exists s[(x,y)] \leq k-2$ in $L_k$
22:         $L_k \leftarrow L_k \setminus \{(x,y)\}$;
23:         for $z \in N(x) \cap N(y)$ do
24:             if $\tau((x,z)) < k$ or $\tau((y,z)) < k$ then continue;
25:             if $\tau((x,z)) = k$ and $(x,z) \notin L_k$ then continue;
26:             if $\tau((y,z)) = k$ and $(y,z) \notin L_k$ then continue;
27:             if $(x,z) \in L_k$ then $s[(x,z)] \leftarrow s[(x,z)] - 1$;
28:             if $(y,z) \in L_k$ then $s[(y,z)] \leftarrow s[(y,z)] - 1$;
29:     for $(x,y) \in L_k$ do
30:         $\hat{\tau}((x,y)) \leftarrow k + 1$;
```

## 4.3 Updating TCP-Index

We study how to update the TCP-Index. Recall that, for $x \in V$, the index $\mathcal{T}_x$ is the maximum spanning forest of $x$'s neighborhood graph $G_x$. Thus the key problem is how to update the maximum spanning forest of $G_x$ given an edge insertion/deletion and the consequent edge trussness update.

### *4.3.1 Updating TCP-Index With Edge Insertion*

**Index Update with Edge Insertion**. With an inserted edge $e_0(u, v)$, we first consider how to update $\mathcal{T}_u$ and $\mathcal{T}_v$. Take vertex $u$ as an example. $G_u$ now includes a new vertex $v$ and a set of new edges $\{(v,w) | w \in N(u) \cap N(v)\}$. Then we update the maximum spanning forest $\mathcal{T}_u$ with the new vertex and edges in $G_u$. The time cost is $O(|N(u)| + |N(u) \cap N(v)|) \subseteq O(|N(u)|)$.

For example, consider the insertion of $(s_2, x_2)$ in Figure 7(a). The index $\mathcal{T}_{x_2}$ before insertion is shown in Figure 8(a). Now vertex $s_2$ and three edges $(s_2, q)$, $(s_2, x_1)$, $(s_2, x_4)$ all with weight 5 are inserted into $G_{x_2}$. So the updated $\mathcal{T}_{x_2}$ is shown in Figure 8(b).

Now we consider how to update $\mathcal{T}_w$ for $w \in N(u) \cap N(v)$. $G_w$ includes a new edge $(u, v)$. We first compute the weight $w(u,v) = \min\{\hat{\tau}((u,v)), \hat{\tau}((u,w)), \hat{\tau}((v,w))\}$. If $u, v$ are in different components of $\mathcal{T}_w$, we add the edge $(u, v)$ with weight $w(u, v)$ into $\mathcal{T}_w$; otherwise, we can find a unique path $P$ between $u$ and $v$ in $\mathcal{T}_w$. Then we compute the minimum weight on $P$ as $w^* = \min_{e\in P} w(e)$. If $w^* \geq w(u,v)$, $\mathcal{T}_w$ remains unchanged; if $w^* < w(u,v)$, the corresponding edge is replaced by $(u,v)$ in $\mathcal{T}_w$. The time cost is $O(|N(w)|)$.

Continue with the above example. With the insertion of $(s_2, x_2)$, their common neighbor $q$'s index before insertion is shown in Figure 8(e). There is a path $(s_2, x_4, x_1, x_2)$ with a minimum weight 4. So we replace $(s_2, x_4)$ by $(s_2, x_2)$ with weight 5 in Figure 8(f).

**Index Update with Trussness Increase**. Besides the above cases, we also need to consider the index maintenance for an existing edge $e(x,y)$ which has trussness update $\hat{\tau}(e) = \tau(e) + 1$ due to the insertion of $e_0$. We discuss the update of $\mathcal{T}_x$ (and similarly $\mathcal{T}_y$). For every triangle $\triangle_{xyz}$, we denote $w_{xyz} = \min\{\tau((x,y)), \tau((x,z)), \tau((y,z))\}$, and $\hat{w}_{xyz} = \min\{\hat{\tau}((x,y)), \hat{\tau}((x,z)), \hat{\tau}((y,z))\}$. If $\hat{w}_{xyz} = w_{xyz}$, $\mathcal{T}_x$ remains unchanged. Otherwise, for $\hat{w}_{xyz} = w_{xyz} + 1$, if $(y,z) \in \mathcal{T}_x$, we update the edge weight $w(y,z) = \hat{w}_{xyz}$ in $\mathcal{T}_x$; if $(y,z) \notin \mathcal{T}_x$, we find the unique path $P$ between $y$ and $z$ in $\mathcal{T}_x$ and update $P$ with the new weight $w(y,z)$ in a similar process as described above for updating $\mathcal{T}_w$. For the neighbor vertex $z \in N(x) \cap N(y)$, the index $\mathcal{T}_z$ can be updated similarly.

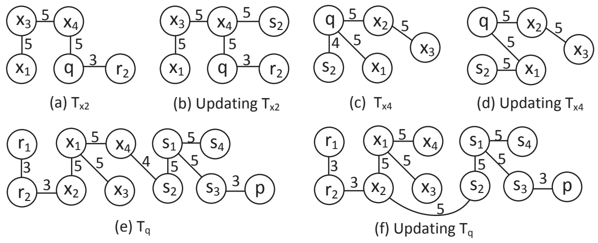


**Fig. 8** The updating steps of TCP-Index

Continue with the above example. With the insertion of $(s_2, x_2)$, we have $\hat{\tau}((s_2, x_4)) = \tau((s_2, x_4)) + 1$, and $\hat{w}_{s_2 x_4 x_1} = 5$. The index $\mathcal{T}_{x_4}$ before insertion is shown in Figure 8(c). We use $(s_2, x_1)$ with weight 5 to replace $(s_2, q)$ with weight 4 in Figure 8(d).

#### *4.3.2 Updating TCP-Index With Edge Deletion*

**Index Update with Edge Deletion**. With a deleted edge $e_0(u,v)$, we first consider how to update $\mathcal{T}_u$ and $\mathcal{T}_v$. Take vertex $u$ as an example. We first delete vertex $v$ and edges $\{(v,w) | (v,w) \in \mathcal{T}_u\}$ from $\mathcal{T}_u$. Then we update the maximum spanning forest $\mathcal{T}_u$ with the available edges in $E(G_u)$.

Now we consider how to update $\mathcal{T}_w$ for $w \in N(u) \cap N(v)$. If $(u,v) \in \mathcal{T}_w$, we try to find an edge $(u', v') \in E(G_w)$ where $u' \in V_2(w,u), v' \in V_2(w,v)$ with the maximum weight to replace $(u,v)$ in $\mathcal{T}_w$. If no such replacement edge exists, we simply remove $(u,v)$. On the other hand, if $(u,v) \notin \mathcal{T}_w$, $\mathcal{T}_w$ remains unchanged.

**Index Update with Trussness Decrease**. Besides the above cases, we also need to consider the index maintenance for an existing edge $e(x,y)$ which has trussness update $\hat{\tau}(e) = \tau(e) - 1$ due to the edge deletion. We discuss the update of $\mathcal{T}_x$ (and similarly $\mathcal{T}_y$). For every $\triangle_{xyz}$, we denote $w_{xyz} = \min\{\tau((x,y)), \tau((x,z)), \tau((y,z))\}$, and $\hat{w}_{xyz} = \min\{\hat{\tau}((x,y)), \hat{\tau}((x,z)), \hat{\tau}((y,z))\}$. If $\hat{w}_{xyz} = w_{xyz}$, $\mathcal{T}_x$ remains unchanged. Otherwise, for $\hat{w}_{xyz} = w_{xyz} - 1$, if $(y,z) \notin \mathcal{T}_x$, $\mathcal{T}_x$ remains unchanged; if $(y,z) \in \mathcal{T}_x$, we try to find an edge $(y', z') \in E(G_x)$ where $y' \in V_{w_{xyz}}(x,y), z' \in V_{w_{xyz}}(x,z)$ and $w(y', z') = w_{xyz}$ to replace $(y,z)$ in $\mathcal{T}_x$. If no such replacement edge exists, we just update $w(y,z) = \hat{w}_{xyz}$. For the neighbor vertex $z \in N(x) \cap N(y)$, the index $\mathcal{T}_z$ can be updated in a similar way.

## 5 I/O-Efficient Algorithms for K-Truss Community Search

In this section, we study $k$-truss community search in massive graphs which cannot fit entirely in main memory. In this case, Algorithm 3 fails to construct TCP-Index for the graph in memory. Even worse, TCP-Index may not be held in memory, as its size complexity is $O(m)$. As a consequence, TCP-Index-based query processing for $k$-truss community search by Algorithm 4 does not work either. To address the limitations of these in-memory algorithms on very large graphs, we develop I/O-efficient solutions for TCP-Index construction and query processing under a semi-external model.

In the following, we first give an overview of our solution in Section 5.1. In Section 5.2, we present I/O-efficient algorithms for TCP-Index construction. We first introduce a straightforward indexing method extended from Algorithm 3, which incurs a lot of I/O operations. To improve the efficiency, we design various novel strategies to reduce I/O costs for TCP-Index construction. We analyze the correctness and I/O complexity of our algorithms. Based on TCP-Index on disk, we propose an I/O-efficient algorithm for querying $k$-truss community in Section 5.3.

### 5.1 Solution Overview

We mainly consider how to reduce the number of basic operations of reading/writing blocks from disk/memory into memory/disk, under the semi-external model as follows.

**Semi-External Model.** We adopt a widely-used semi-external model in our solution. We denote the size of main memory by $M$ and the disk block size by $B$. An I/O operation will read/write one block of size $B$ from disk/memory into memory/disk. The semi-external model assumes that all graph nodes can be stored in the main memory while the edges cannot, i.e., $M \geq c \times |V(G)|$ where $c$ is a small constant. This assumption practically holds as the node size is far smaller than the edge size in most real social networks and web graphs. For example, for the WebBase graph with 115.5 million nodes and 1.7 billion edges, it only requires 5.5 GB memory for running our semi-external algorithms, which is affordable for most PCs nowadays.

**I/O-Efficient TCP-Index Construction.** We consider the TCP-Index construction for a vertex $x \in V$. Recall that our in-memory TCP-Index construction in Algorithm 3 mainly consists of two steps:

1. Finding $G_x$. We first generate a weighted neighborhood graph $G_x$. We find all edges $(y, z)$ in $G_x$ for $y, z \in N(x)$ and compute their weights. Note that the size of $G_x$ may exceed the memory capacity $M$.
2. Constructing $\mathcal{T}_x$. The TCP-Index $\mathcal{T}_x$ is built as the maximum spanning forest of $G_x$. We start building $\mathcal{T}_x$ from isolated vertices in $N(x)$, and insert edges of $G_x$ one by one in the decreasing order of edge weight to link different components into one.

Thus the key problem is how to find $G_x$ and sort the edges in $E(G_x)$ when $|E(G_x)| \geq M$.

Section 5.2.1 first presents a straightforward solution to generating $G_x$. It writes the new found edges of $G_x$ into disk immediately when the edges and their edge weights are obtained. When the whole structure of $G_x$ is on disk, we apply an external sorting algorithm on all edges in $E(G_x)$ based on their weights. Finally, we construct $\mathcal{T}_x$ by one scan of the sorted edges on disk. This method is simple, but needs a large number of I/Os.

To reduce the I/O cost of external sorting, we apply the idea of merge sort to construct TCP-Index in Section 5.2.2. The key idea is to divide $G_x$ into multiple partitions, where each partition has $O(M)$ edges sorted in memory and then written to disk. To build $\mathcal{T}_x$, we read blocks with the largest edge weights from all partitions into memory, using the idea of merge sort. To further improve the I/O efficiency, in Section 5.2.3, we develop a sort-free method for TCP-Index construction, which does not need to perform external sorting.

**I/O-Efficient K-Truss Community Search.** Note that the constructed TCP-Index for all vertices takes $O(m)$ space, which cannot fit into memory. Moreover, in the process of community search, the visited edge list and queue $Q$ may also exceed the memory size $M$. To address these issues, we use two data structures of bitmap and circular queue, which realize the functions of visited edge list and the sequential visiting order in an I/O-efficient way. The bitmap uses one bit to record an edge as visited or not, which saves the space cost. The purpose of circular queue is to avoid frequently allocating and releasing memory, as it is easy to write to and load from disk.

**Algorithm 6** External-Sort TCP-Index Construction

```
Input: $G = (V, E)$
Output: TCP-Index $\mathcal{T}_x$ for each $x \in V$

1: Apply I/O-efficient truss decomposition for $G$ [34];
2: for $x \in V$ do
3:     Load the block containing $N(x)$ from disk;
4:     for $y \in N(x)$ do
5:         Load the block containing $N(y)$ from disk;
6:         for $z \in N(x) \cap N(y)$ do
7:             $w(y, z) \leftarrow \min\{\tau((x, y)), \tau((x, z)), \tau((y, z))\}$;
8:             Store $< (y, z), w(y, z) >$ in disk;
9:     Sort all edges $(y, z)$ in the decreasing order of $w(y, z)$ where $y, z \in N(x)$;
10:    $\mathcal{T}_x \leftarrow N(x)$;
11:    $k_{max} \leftarrow \max\{w(y, z)|y, z \in N(x)\}$;
12:    for $k \leftarrow k_{max}$ to 2 do
13:        Load edges $(y, z)$ with $w(y, z) = k$ from disk;
14:        $S_k \leftarrow \{(y, z)|y, z \in N(x), w(y, z) = k\}$;
15:        for $(y, z) \in S_k$ do
16:            if $y, z$ are in different components in $\mathcal{T}_x$
17:                add $(y, z)$ with weight $w(y, z)$ in $\mathcal{T}_x$;
18:    Store $\mathcal{T}_x$ in disk;
19: return $\{\mathcal{T}_x | x \in V\}$;
```

## 5.2 TCP-Index Construction in a Semi-External Model

### *5.2.1 External-Sort based TCP-Index Construction*

In this section, we first present a basic method to construct TCP-Index in Algorithm 6 which consists of two steps. The first step is to construct and output the weighted graph $G_x$ to disk iteratively. The second step is then to apply an external-sorting algorithm to sort the edges of $G_x$ in decreasing order of weights for constructing $\mathcal{T}_x$.

Algorithm 6 first applies an I/O-efficient truss decomposition for $G$ [34] and obtains the trussness of all edges in $G$ (line 1). Then, for each vertex $x \in V$, it builds the TCP-Index $\mathcal{T}_x$ (line 2-19). Specifically, we construct the weighted graph $G_x$ (line 3-9). For each vertex $y \in N(x)$, we load the blocks containing the adjacent list $N(y)$ from disk (line 4-5). Then, we compute all triangles $\triangle_{xyz}$ where $z \in N(x) \cap N(y)$, and store the edge $(y, z)$ with the weight $w(y, z)$ in disk (line 6-8). After obtaining $G_x$, we use an external-sorting algorithm on the edges of $G_x$ in the decreasing order of edge weights (line 9). Next, we construct $\mathcal{T}_x$ as the maximum spanning forest of $G_x$ (line 10-18). We build $\mathcal{T}_x$ from isolated vertices in $N(x)$, and load from disk sorted edges as many as the memory allows (line 10-13). We then insert them one by one into $\mathcal{T}_x$ to connect different components (line 14-17), which is the same as Algorithm 3. Finally, $\mathcal{T}_x$ for each vertex $x \in V$ is built and stored on disk (line 18-19).

**Algorithm 7** Merge-Sort TCP-Index Construction

```
Input: $G = (V, E)$
Output: TCP-Index $\mathcal{T}_x$ for each $x \in V$

1: Apply I/O-efficient truss decomposition for $G$ [34];
2: Let $r \leftarrow 0$;
3: for $x \in V$ do
4:     Load the block containing $N(x)$ from disk;
5:     $G_x \leftarrow \emptyset$;
6:     for $y \in N(x)$ do
7:         Load the block containing $N(y)$ from disk;
8:         for $z \in N(x) \cap N(y)$ do
9:             $w(y,z) \leftarrow \min\{\tau((x,y)), \tau((x,z)), \tau((y,z))\}$;
10:            $G_x \leftarrow G_x \cup \{< (y,z), w(y,z) >\}$;
11:            if there is no enough memory to store new edges
12:                $r \leftarrow r + 1$;
13:                Sort $E(G_x)$ in decreasing order of weights;
14:                Store $G_x$ as partition $P_r$ on disk;
15:                $G_x \leftarrow \emptyset$;
16:    The partitions $\{P_1, P_2, \dots, P_r\}$ keep $G_x$ on disk;
17:    while $r \geq M/B$ do
18:        merge the partitions to reduce $r$;
19:    Load $\frac{M}{Br}$ blocks with the largest weights into memory from each partition $P_i$ where $1 \leq i \leq r$, and mark them as visited;
20:    $\mathcal{T}_x \leftarrow N(x)$;
21:    while $\exists$ one unvisited block in $P_i$ do
22:        $(y,z) \leftarrow$ edge with the largest weight in memory;
23:        $S_k \leftarrow \{(y,z) | y, z \in N(x), w(y,z) = k\}$;
24:        for $(y,z) \in S_k$ do
25:            if $y, z$ are in different components in $\mathcal{T}_x$
26:                add $(y,z)$ with weight $w(y,z)$ in $\mathcal{T}_x$;
27:        if all edges of a block in $P_i$ have been traversed
28:            Load a block with the largest weights from $P_i$ into memory, and mark it as visited;
29:    Store $\mathcal{T}_x$ in disk;
30: return $\{\mathcal{T}_x | x \in V\}$;
```

### *5.2.2 Merge-Sort based TCP-Index Construction*

Algorithm 6 wastes a lot of I/O costs on many repeated reading/writing operations of generating, sorting, and scanning $G_x$. To reduce the I/O costs, we do not need to sort all edges of $G_x$. Instead, we create multiple edge partitions of $G_x$, each having $O(M)$ edges, which are sorted in memory and then written to disk. We then read blocks from all partitions into memory and use the idea of merge sort to build TCP-Index. The method is outlined in Algorithm 7.

Algorithm 7 consists of two phases: generating $G_x$ (line 1-15) and building $\mathcal{T}_x$ from $G_x$ (line 17-30). To generate $G_x$ for a vertex $x$, it performs the following operations iteratively. $G_x$ is initialized empty and grows with a new edge $(y, z)$ where $y \in N(x)$ and $z \in N(x) \cap N(y)$. Once the number of edges in $G_x$ reaches the memory capacity (line 11), the algorithm sorts all edges of $G_x$ in decreasing order of weights in memory (line 13). It then stores all edges on disk and sets $G_x$ to be empty again (line 14-15). Thus, $G_x$ is divided into several partitions $\{P_1, P_2, \dots, P_r\}$. For each partition $P_i$, edges are sorted in order (line 16). To build $\mathcal{T}_x$ from $G_x$, Algorithm 7 adopts the idea of merge sort over all partitions. If the number of partitions $r \geq M/B$, it performs several rounds of merge to reduce $r$ until $r < M/B$, which ensures that at least one block from each partition of $\{P_1, P_2, \dots, P_r\}$ can be read into memory (line 17-18). It loads $\frac{M}{Br}$ blocks with the largest weights into memory from each partition $P_i$, and marks them as visited (line 19). Then, it builds $\mathcal{T}_x$ from isolated nodes by inserting edges with the largest weights, and expands it to a maximum spanning forest (line 21-28). If all edges of a block from partition $P_i$ have been visited in memory, we load the next unvisited block of $P_i$ (line 27-28). Finally, the algorithm generates the TCP-Index for each vertex $x \in V$ (line 30).

### *5.2.3 Sort-Free TCP-Index Construction*

To further improve the I/O efficiency, we propose an optimal algorithm for TCP-Index construction in Algorithm 8. This method does not need to store the whole structure of $G_x$ on disk. Alternatively, it dynamically generates partial structure of $G_x$ and updates the maximum spanning forest $\mathcal{T}_x$ based on the current structure of $G_x$. The generation of $G_x$ and update of $\mathcal{T}_x$ can be done in memory without any I/O cost.

The key idea of Algorithm 8 is based on the principle that a maximum spanning forest $\mathcal{T}_x$ of $G_x$ is identical to the maximum spanning forest of $\mathcal{T}^* \cup \{G_x - G^*\}$ where $\mathcal{T}^*$ is a maximum spanning forest of $G^*$. This indicates that we can generate a partial structure of $G_x$, build the maximum spanning forest for it, and then release the partial structure in memory, and continue to generate other parts of $G_x$. The correctness of this principle is proved in Theorem 4. Algorithm 8 generates partial structure of $G_x$ whose size is limited by $M$ and updates $\mathcal{T}_x$ at the same time (line 10-13). After that, the graph structure of $G_x$ is discarded, which makes room in memory to further generate other parts of $G_x$. An in-memory procedure findMSF is developed to compute the maximum spanning forest $\mathcal{T}_x$ from any graph structure $G^*$, which is similar to Algorithm 3.

*Example 6* Figure 9 shows an example of generating neighborhood graph $G_x$ and finding maximum spanning forest (MSF) simultaneously. The edges of $G_x$ are identified one by one. When the size of $G_x$ reaches the memory size $M$ as Figure 9(a) shows, we find the MSF $\mathcal{T}_1$ of graph $G_1$ in Figure 9(b). Only edges in $\mathcal{T}_1$ are kept in memory, and other edges of $G_1$ are discarded. Then we continue to identify edges of $G_x$ and add them into $\mathcal{T}_1$ until the size reaches $M$ again or all edges of $G_x$ are visited. Assume all edges in $G_x$ have been identified and the graph now is shown in Figure 9(c). We can verify that the MSF $\mathcal{T}_2$ in Figure 9(d) built by Algorithm 8

**Algorithm 8** Sort-Free TCP-Index Construction

```
Input: $G = (V, E)$
Output: TCP-Index $\mathcal{T}_x$ for each $x \in V$

 1: Apply I/O-efficient truss decomposition for $G$ [34];
 2: for $x \in V$ do
 3:     Load the block containing $N(x)$ from disk;
 4:     $G_x(V, E) \leftarrow G(N(x), \emptyset)$ ;
 5:     for $y \in N(x)$ do
 6:         Load the block containing $N(y)$ from disk;
 7:         for $z \in N(x) \cap N(y)$ do
 8:             $w(y, z) \leftarrow \min\{\tau((x, y)), \tau((x, z)), \tau((y, z))\}$;
 9:             $E(G_x) = E(G_x) \cup \{(y, z)\}$;
10:             if there is no enough memory to store new edges
11:                 $G_x \leftarrow$ findMSF ($G_x$);
12:     $\mathcal{T}_x \leftarrow$ findMSF ($G_x$);
13:     Store $\mathcal{T}_x$ in disk;
14: return $\{\mathcal{T}_x | x \in V\}$;
15:
16: Procedure findMSF (Graph $G^*$)
17:     Sort $E(G^*)$ in the decreasing order of weights;
18:     $\mathcal{T}^* \leftarrow V(G^*)$;
19:     $k_{max} \leftarrow \max\{w(y, z) | y, z \in G^*\}$;
20:     for $k \leftarrow k_{max}$ to 2 do
21:         $S_k \leftarrow \{(y, z) | y, z \in E(G^*), w(y, z) = k\}$;
22:         for $(y, z) \in S_k$ do
23:             if $y, z$ are in different components in $\mathcal{T}^*$
24:                 add $(y, z)$ with weight $w(y, z)$ in $\mathcal{T}^*$;
25:     return $\mathcal{T}^*$;
```

is also the MSF of the whole neighborhood graph $G_x$. In this process we build the TCP-Index of a vertex in memory.

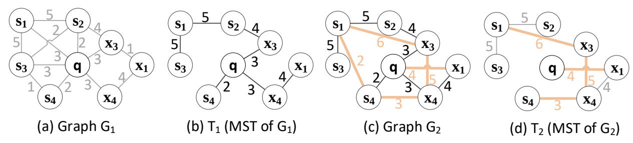


**Fig. 9** An example graph to finding MSF

**Correctness Analysis.** To prove the correctness of Algorithm 8, we need to show that given a vertex $x$, the output $\mathcal{T}_x$ by Algorithm 8 is exactly the MSF of $G_x$. The core idea of our proof is shown as follows. When the condition that there is no enough memory to store new edges (line 10) holds for the first time, we denote the partial structure of $G_x$ as $g_1$ and its corresponding MSF as $\mathcal{T}_1$ (line 11). Algorithm 8 then assigns $\mathcal{T}_1$ as the new structure of $G_x$, denoted as $g_2$, i.e., $g_2 = \mathcal{T}_1$ (line 11), and continues reading the remaining edges of $G_x$ from disk to memory, until the size of $g_2$ reaches the memory limit. We denote the corresponding MSF of $g_2$ as $\mathcal{T}_2$. Algorithm 8 repeats the above process until all edges of $G_x$ are read from disk to memory. We denote the last partial graph of $G_x$ in memory as $g_r$ and its corresponding tree as $\mathcal{T}_r$ where $r \geq 1$. Clearly, the whole graph is $G_x = g_1 \cup g_2 \cup \ldots \cup g_r$, then we prove that $\mathcal{T}_r$ is the MSF of $G_x$ in Theorem 4.

In the following, we introduce some useful notations in our theorem and lemmas. Given a graph $G(V, E)$, the induced subgraph of $G$ by a vertex set $S \subseteq V$ is denoted by $G[S] = (S, E_S)$ where $E_S = \{(u, v) \in E : u, v \in S\}$. The weight of a subgraph $H \subseteq G$ is defined as $w(H) = \sum_{e \in E(H)} w(e)$. In addition, we use $G \cup \{e^*\}$ to represent a new graph that is formed by $G$ with an edge insertion of $e^*$.

**Lemma 4** *Consider a graph $G(V, E)$ and a new edge $e^* = (x, y) \notin E$. If $\mathcal{T}$ is the MSF of $G$, then the MSF of $\mathcal{T} \cup \{e^*\}$ is also the MSF of $G \cup \{e^*\}$.*

*Proof* Let $H_1 = \mathcal{T} \cup \{e^*\}$ and $H_2 = G \cup \{e^*\}$. To prove that the MSF of $H_1$ is also the MSF of $H_2$, it is equivalent to prove $w(\mathcal{T}_1) = w(\mathcal{T}_2)$, where $\mathcal{T}_1$ and $\mathcal{T}_2$ are the MSF of $H_1$ and $H_2$ respectively.

First, we prove $w(\mathcal{T}_1) \leq w(\mathcal{T}_2)$. For $\mathcal{T} \subseteq G$, we have $H_1 \subseteq H_2$. In addition, $\mathcal{T}_2$ is the MSF of $H_2$. Obviously, $w(\mathcal{T}_1) \leq w(\mathcal{T}_2)$.

Second, we prove $w(\mathcal{T}_1) \geq w(\mathcal{T}_2)$. This inequality holds based on the two following cases.

**Case 1:** $e^* \notin \mathcal{T}_2$. We have that $\mathcal{T}_2$ is the MSF of $G$, as $G \subseteq H_2$. Since $\mathcal{T}$ is the MSF of $G$, $w(\mathcal{T}_2) = w(\mathcal{T})$ holds. Moreover, $\mathcal{T}_1$ is the MSF of $H_1 \supset \mathcal{T}$. Thus, $w(\mathcal{T}_1) \geq w(\mathcal{T}) = w(\mathcal{T}_2)$ holds.

**Case 2:** $e^* \in \mathcal{T}_2$. Assume that we delete $e^* = (x, y)$ from $\mathcal{T}_2$ and split the whole forest into two disjoint forests $\mathcal{T}_x$ and $\mathcal{T}_y$. Let $X, Y$ be the vertex set of $\mathcal{T}_x$ and $\mathcal{T}_y$ respectively, and $X \cap Y = \emptyset$. We have two induced subgraphs $G[X] \subseteq G$ and $G[Y] \subseteq G$, and their corresponding MSFs as $\hat{\mathcal{T}_x}$, $\hat{\mathcal{T}_y}$. Obviously, $w(\hat{\mathcal{T}_x}) \geq w(\mathcal{T}_x)$ and $w(\hat{\mathcal{T}_y}) \geq w(\mathcal{T}_y)$. If $\mathcal{T}$ contains an edge $e' = (x', y')$ where $x' \in X$ and $y' \in Y$, we have $w(\mathcal{T}) - w(e') \geq w(\hat{\mathcal{T}_x}) + w(\hat{\mathcal{T}_y}) \geq w(\mathcal{T}_x) + w(\mathcal{T}_y)$, because $\mathcal{T}$ is the MSF of $G$ and $G \supseteq G[X] \cup G[Y]$; otherwise, $\mathcal{T}$ contains no such edge $e'$, then $w(\mathcal{T}) = w(\hat{\mathcal{T}_x}) + w(\hat{\mathcal{T}_y}) \geq w(\mathcal{T}_x) + w(\mathcal{T}_y)$. Overall, $w(\mathcal{T}) \geq w(\mathcal{T}_x) + w(\mathcal{T}_y)$ holds. Now, $\mathcal{T}_1$ is the MSF of $H_1 = \mathcal{T} \cup \{e^*\}$, indicating $w(\mathcal{T}_1) \geq w(\mathcal{T}) - w(e') + \max\{w(e'), w(e^*)\} \geq w(\mathcal{T}_x) + w(\mathcal{T}_y) + \max\{w(e'), w(e^*)\} \geq w(\mathcal{T}_x) + w(\mathcal{T}_y) + w(e^*) = w(\mathcal{T}_2)$. Note that, $w(e') = 0$ if $e'$ does not exist in $\mathcal{T}$. As a result, $w(\mathcal{T}_1) \geq w(\mathcal{T}_2)$ follows from this.

Finally, $w(\mathcal{T}_1) = w(\mathcal{T}_2)$ holds, due to $w(\mathcal{T}_1) \leq w(\mathcal{T}_2)$ and $w(\mathcal{T}_1) \geq w(\mathcal{T}_2)$.

**Lemma 5** *Given two graphs $G_a$ and $G_b$, $\mathcal{T}_a$ is a MSF of $G_a$, and $\mathcal{T}_b$ is a MSF of $G_b$. If $\mathcal{T}_a \subseteq G_b$, then $\mathcal{T}_b$ is a MSF of $G_a \cup G_b$.*

*Proof* For $\mathcal{T}_a \subseteq G_b$, let $G_b = \mathcal{T}_a \cup \{e_1\} \cup \{e_2\} \cup \ldots \cup \{e_l\}$ where $l$ is a positive integer. Since $\mathcal{T}_a$ is a MSF of $G_a$, we have $\mathcal{T}_a \subseteq G_a$, and $G_a \cup G_b = G_a \cup \mathcal{T}_a \cup \{e_1\} \cup \{e_2\} \cup \ldots \cup \{e_l\} = G_a \cup \{e_1\} \cup \{e_2\} \cup \ldots \cup \{e_l\}$. Now, we prove that the MSF of $\mathcal{T}_a \cup \{e_1\} \cup \{e_2\} \cup \ldots \cup \{e_l\}$ is the MSF of $G_a \cup \{e_1\} \cup \{e_2\} \cup \ldots \cup \{e_l\}$.

We prove it using induction. Specifically, we show that $\mathcal{T}_i$ is $\mathsf{MSF}$ of $G_i$, where $0 \leq i \leq l$. Let $\mathcal{T}_0 = \mathcal{T}_a$ and $G_0 = G_a$. Obviously, $\mathcal{T}_i$ is the $\mathsf{MSF}$ of $G_i$ for $i = 0$. For $0 \leq i \leq l$, let $\mathcal{T}_{i+1}$ be the $\mathsf{MSF}$ of $\mathcal{T}_i \cup \{e_i\}$ and $G_{i+1} = G_i \cup \{e_i\}$.

Next, we show that for $1 \leq i \leq l$, given $\mathcal{T}_i$ as the $\mathsf{MSF}$ of $G_i$ and $\mathcal{T}_{i+1}$ as the $\mathsf{MSF}$ of $\mathcal{T}_i \cup \{e_i\}$, $\mathcal{T}_{i+1}$ is also the $\mathsf{MSF}$ of $G_i \cup \{e_i\} = G_{i+1}$. This rule clearly holds by Lemma 4. As a result, $\mathcal{T}_{i+1}$ is always the $\mathsf{MSF}$ of $G_{i+1}$, where $0 \leq i \leq l$.

Finally, for $i = l$, $G_{l+1} = G_a \cup \{e_1\} \cup \{e_2\} \cup \ldots \cup \{e_l\}$, and $\mathcal{T}_{l+1}$ is the $\mathsf{MSF}$ of $G_{l+1} = G_a \cup G_b$. Moreover, $\mathcal{T}_{l+1}$ is a subgraph of $\mathcal{T}_a \cup \{e_1\} \cup \{e_2\} \cup \ldots \cup \{e_l\} = G_b$, and $G_b \subseteq G_a \cup G_b = G_{l+1}$, thus $\mathcal{T}_{l+1}$ is also the $\mathsf{MSF}$ of $\mathcal{T}_a \cup \{e_1\} \cup \{e_2\} \cup \ldots \cup \{e_l\} = G_b$, which completes the proof.

**Theorem 4** *Given $r$ graphs $g_1$, $g_2$, $\ldots$, $g_r$ and their corresponding $\mathsf{MSFs}$ $\mathcal{T}_1$, $\mathcal{T}_2$, $\ldots$, $\mathcal{T}_r$, let $G_i = g_1 \cup \ldots \cup g_i$ and $\mathcal{T}_i \subseteq g_{i+1}$ for $1 \leq i \leq r-1$. Then, it holds that $\mathcal{T}_r$ is the $\mathsf{MSF}$ of $G_r = g_1 \cup \ldots \cup g_r$.*

*Proof* We apply the induction to prove that $\mathcal{T}_i$ is the $\mathsf{MSF}$ of $G_i$ for $1 \leq i \leq r$. First, for $i = 1$, $\mathcal{T}_1$ obviously is the $\mathsf{MSF}$ of $G_1 = g_1$. Next, for $1 \leq i \leq r-1$, assume that $\mathcal{T}_i$ is the $\mathsf{MSF}$ of $G_i$. By Lemma 5, we have $\mathcal{T}_i \subseteq g_{i+1}$, then $\mathcal{T}_{i+1}$ is a $\mathsf{MSF}$ of $G_i \cup g_{i+1} = G_{i+1}$. Finally, for $i = r-1$, we prove that $\mathcal{T}_r$ is a $\mathsf{MSF}$ of $G_r = g_1 \cup \ldots \cup g_r$.

Based on Theorem 4, we finally prove the correctness of Algorithm 8.

### *5.2.4 Complexity Analysis*

We analyze the complexity of the three $\mathsf{TCP\text{-}Index}$ construction algorithms. The memory consists of $\frac{M}{B}$ blocks. For vertex $x$, the edge size of its neighborhood induced subgraph $G_x$ is $|E(G_x)| \leq \sum_{y \in N(x)} \min\{d(x), d(y)\}$. The storage of $G_x$ takes $O(\frac{|E(G_x)|}{B})$ blocks. Note that we do not take the complexity of truss decomposition [34] into account. We have the following I/O complexity for the three algorithms.

**Theorem 5** *Algorithm 6 takes $O(\frac{\sum_{(x,y)\in E}(d(x)+d(y))}{B} + \sum_{x\in V} \frac{|E(G_x)|}{B} \log_{\frac{M}{B}} \frac{|E(G_x)|}{B})$ I/Os, where $G_x$ is the induced subgraph of $G$ by $N(x)$.*

*Proof* Algorithm 6 constructs $\mathsf{TCP\text{-}Index}$ for all vertices in graph $G$. Consider the $\mathsf{TCP\text{-}Index}$ construction in $G_x$ for vertex $x$. The algorithm consists of three steps: load adjacent lists, store and reload graph $G_x$, and sort edges of $G_x$ in disk.

First, the step of loading all adjacent lists $N(y)$ for $y \in N(x)$ takes $O(\frac{\sum_{y\in N(x)} |N(y)|}{B})$ I/Os. Second, $G_x$ occupies $O(\frac{|E(G_x)|}{B})$ blocks on disk. Moving $G_x$ between memory and disk takes $O(\frac{|E(G_x)|}{B})$ I/Os. Third, sorting all edges of $G_x$ takes $O(\frac{|E(G_x)|}{B} \log_{\frac{M}{B}} \frac{|E(G_x)|}{B})$ I/Os, as it takes $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ I/Os for sorting $N$ numbers using the external sorting algorithm [22].

Adding up these three steps, Algorithm 6 takes $O(\sum_{x\in V}(\frac{\sum_{y\in N(x)}|N(y)|}{B} + \frac{|E(G_x)|}{B} + \frac{|E(G_x)|}{B} \log_{\frac{M}{B}} \frac{|E(G_x)|}{B})) \subseteq O(\frac{\sum_{(x,y)\in E}(d(x)+d(y))}{B} + \sum_{x\in V} \frac{|E(G_x)|}{B} \log_{\frac{M}{B}} \frac{|E(G_x)|}{B})$ I/Os.

**Theorem 6** *Algorithm 7 takes $O(\frac{\sum_{(x,y)\in E}(d(x)+d(y))}{B} + \sum_{x\in V} \frac{|E(G_x)|}{B} \log_{\frac{M}{B}} \frac{|E(G_x)|}{M})$ I/Os, where $G_x$ is the induced subgraph of $G$ by $N(x)$.*

*Proof* Algorithm 7 has two steps: generating $G_x$ and moving $G_x$ between memory and disk. It takes $O(\frac{\sum_{(x,y)\in E}(d(x)+d(y))}{B})$ I/Os for computing all $G_x$ in graph $G$. Next, we consider the cost of moving $G_x$ between memory and disk. For a given vertex $x$, $G_x$ occupies $O(\frac{|E(G_x)|}{B})$ blocks. All edges of $G_x$ are partitioned into $r$ partitions $P_1, \ldots, P_r$ where $r = \lceil \frac{|E(G_x)|}{M} \rceil$. The memory has $\frac{M}{B}$ blocks. We consider two following cases. (1) $\frac{M}{B} > r$, Algorithm 7 skips the merging process of external sorting (line 17-18 of Algorithm 7); or (2) $\frac{M}{B} \leq r$. It takes $O(\log_{\frac{M}{B}} \frac{|E(G_x)|}{M})$ rounds to merge partitions into a small number of sorted edge lists such that the number of partitions $r < \frac{M}{B}$ holds. Each round of merging process needs a full scan of the whole graph $G_x$ using $O(\frac{|E(G_x)|}{B})$ I/Os. Overall, it takes $O(\sum_{x\in V} \frac{|E(G_x)|}{B} \log_{\frac{M}{B}} \frac{|E(G_x)|}{M})$ I/Os for moving $G_x$ between memory and disk to identify $T_x$. As a result, the I/O complexity of Algorithm 7 is $O(\frac{\sum_{(x,y)\in E}(d(x)+d(y))}{B} + \sum_{x\in V} \frac{|E(G_x)|}{B} \log_{\frac{M}{B}} \frac{|E(G_x)|}{M})$.

**Theorem 7** *Algorithm 8 takes $O(\frac{\sum_{(x,y)\in E}(d(x)+d(y))}{B})$ I/Os.*

*Proof* Algorithm 8 only needs to load adjacent lists from disk to memory. Other operations of $\mathsf{TCP\text{-}Index}$ construction are performed in memory. Thus, Algorithm 8 takes $O(\frac{\sum_{(x,y)\in E}(d(x)+d(y))}{B})$ I/Os.

**Space Complexity Analysis.** All three algorithms take $O(\min\{M, m\})$ in-memory space and $O(\frac{m}{B})$ blocks on disk. For constructing $\mathsf{TCP\text{-}Index}$ for each graph $G_x$, it only keeps all edges of $G_x$ using $O(m)$ disk space. In addition, graph $G$ and $\mathsf{TCP\text{-}Index}$ of $G$ both take $O(m)$ space. Overall, the consumption of disk space by all three algorithms is $O(\frac{m}{B})$ blocks.

In summary, Table 1 shows the comparison of the three algorithms in terms of three key operations, I/O complexity, and disk space complexity. All three methods need to generate $G_x$ by loading adjacent lists into memory. $\mathsf{Free\text{-}TCP}$ performs other operations in memory. Both $\mathsf{Merge\text{-}TCP}$ and $\mathsf{External\text{-}TCP}$ need to read/ write disk operations by moving $G_x$ between memory and disk for constructing $T_x$. $\mathsf{External\text{-}TCP}$ applies the external sorting algorithm on $G_x$, while $\mathsf{Merge\text{-}TCP}$ does not need this step.

**Table 1** A comparison of three algorithms in terms of three key operations, I/O complexity, and disk space complexity.

| Algorithms | External-TCP | Merge-TCP | Free-TCP |
|---|---|---|---|
| Generating $G_x$ | ✓ | ✓ | ✓ |
| Moving $G_x$ between memory/disk | ✓ | ✓ | |
| Sorting $G_x$ | ✓ | | |
| I/O Complexity | $O(\frac{\sum_{(x,y)\in E}(d(x)+d(y))}{B} + \sum_{x\in V}\frac{\lvert E(G_x)\rvert}{B}\log_{\frac{M}{B}}\frac{\lvert E(G_x)\rvert}{B})$ | $O(\frac{\sum_{(x,y)\in E}(d(x)+d(y))}{B} + \sum_{x\in V}\frac{\lvert E(G_x)\rvert}{B}\log_{\frac{M}{B}}\frac{\lvert E(G_x)\rvert}{M})$ | $O(\frac{\sum_{(x,y)\in E}(d(x)+d(y))}{B})$ |
| Disk Space Complexity | $O(\frac{m}{B})$ | $O(\frac{m}{B})$ | $O(\frac{m}{B})$ |

**Algorithm 9** I/O-efficient K-Truss Community Search

**Input:** $G = (V, E)$, an integer $k$, query vertex $v_q$
**Output:** $k$-truss communities containing $v_q$

```
 1: $visited \leftarrow \emptyset$; $l \leftarrow 0$;
 2: for $u \in N(v_q)$ do
 3:     if $\tau((v_q, u)) \geq k$ and $(v_q, u) \notin visited$
 4:         $l \leftarrow l + 1$; $C_l \leftarrow \emptyset$; $Q \leftarrow \emptyset$;
 5:         $Q.push((v_q, u))$;
 6:         while $Q \neq \emptyset$ or $Qfile \neq \emptyset$
 7:             if $Q \neq \emptyset$
 8:                 $(x, y) \leftarrow Q.pop()$;
 9:             else
10:                 read edges into $Q$ from $Qfile$;
11:                 $(x, y) \leftarrow Q.pop()$;
12:             if $(x, y) \notin visited$
13:                 load $T_x$ from disk;
14:                 compute $V_k(x, y)$;
15:                 for $z \in V_k(x, y)$ do
16:                     $visited \leftarrow visited \cup \{(x, z)\}$;
17:                     output $(x, z)$ into the community file;
18:                     if the reverse edge $(z, x) \notin visited$
19:                         if $Q$ is full
20:                             write $Q$ into $Qfile$;
21:                         $Q.push((z, x))$;
22: return;
23: Procedure compute $V_k(x, y)$
24: return $\{z | z$ is connected with $y$ in $\mathcal{T}_x$ through edges of
    weight $\geq k\}$;
```

### 5.3 I/O-efficient K-Truss Community Search

In this section, we propose an I/O-efficient algorithm for $k$-truss community search in Algorithm 9. Based on the TCP-Index on disk, we consider how to efficiently compute all $k$-truss communities containing a query vertex $v_q$. Recall that the in-memory Algorithm 4 starts from an edge $(v_q, u)$ with $\tau((v_q, u)) \geq k$, and expands $(v_q, u)$ to a $k$-truss community by finding other community edges based on TCP-Index. During the process of $k$-truss community search, each edge $(x, y)$ is accessed by queue $Q$ exactly twice in the form: edge $(x, y)$ and reverse edge $(y, x)$. However, the size of community may exceed the memory capacity, indicating that we need to implement a new data structure of queue $Q$. Moreover, to check whether an edge $(x, y)$ has been visited or not, Algorithm 4 uses a hash table, which may also fail to hold large communities by the limited memory. To address these issues, we use two data structures of bitmap and circular queue in the semi-external model.

**Bitmap.** We use a bitmap to implement the function of hash table, which uses 1 bit to mark whether an edge has been visited or not. Specifically, w.l.o.g., given an edge $(x, y) \in E$ and $x < y$, we assign an integer for edge $(x, y)$, denoted as $\delta(x, y) \in [1, m]$. Another integer $\delta(y, x) = \delta(x, y) + m$ is assigned to edge $(y, x)$, denoted as $\delta(y, x) \in [m + 1, 2m]$. Thus, for edge $(x, y)$, we use the $\delta(x, y)$-th position of the bitmap to mark whether it has been visited or not. If $|E| = 1,000,000,000$, it only needs 119.3 MB to implement the bitmap. Moreover, for sparse graphs with $|E| \leq 32|V|$, the bitmap can be kept in main memory, since an integer consists of at least 32 bits.

For some massive graphs, the bitmap may not fit into memory. We develop two simple techniques to reduce I/O cost. (1) When the gap between one edge's position and its reverse edge's position, $m$, is very large, line 18 in Algorithm 9 may cause disk access every time. We may simply skip the checking for the reverse edge and the result will not change, for line 12 will check again. As a side effect, the size of the queue will increase. But as the queue is accessed sequentially, it will cause much fewer I/Os compared with the bitmap. (2) In line 12, the access of bitmap is random and depends on the sequence in the queue. As changing the order of edges in the queue will not change the query result, we can sort the edges in the queue to access the bitmap more sequentially and reduce I/O cost.

**Circular Queue.** We use a circular queue $Q$ to implement the queue used in Algorithm 4. Specifically, we can store the edges of $Q$ on disk as $Qfile$ when needed, and read the blocks of $Qfile$ sequentially. The circular queue $Q$ maintains a fixed number of edges in memory. If $Q$ is not full, we push an edge into $Q$; otherwise, we write edges of $Q$ into disk. If $Q$ is empty, we read edges from disk as many as possible. As we read the edges sequentially, this operation costs few I/Os.

Algorithm 9 presents our I/O-efficient query processing method. We initialize the bitmap to 0 (line 1). If there is an edge$(v_q, u)$ on query node with trussness larger than or equal to $k$, we search the community expanded from this edge in a BFS manner using the circular queue $Q$ and its corresponding file $Qfile$ in disk (line 2-5). When $Q$ or $Qfile$ is not empty, we process the edge $(x, y)$ popped from $Q$ (line 6-11). If the edge $(x, y)$ is unvisited, we compute all nodes that can be connected to $y$ through edges whose trussness is larger than or equal to $k$ in $\mathcal{T}_x$ (line 12-14). For each node $z \in V_k(x, y)$, we mark edge $(x, z)$ as visited in the bitmap and output this edge to the disk as part of the searched community. Then we calculate the reverse edge $(z, x)$ through the bitmap definition (i.e., $\delta(z, x) = \delta(x, z) \pm m$) and check whether it is unvisited (line 18). If it is unvisited and $Q$ is full, we write all edges in $Q$ to disk (line 19-20), and then push it into $Q$ (line 21).

## 6 Experiments

We evaluate the efficiency and effectiveness of our proposed algorithms on real-world networks. All algorithms are implemented in C++. We evaluate in-memory algorithms and I/O-efficient algorithms on different computing environments respectively. All the experiments of in-memory algorithms are conducted on Windows with 2.67GHz six-core CPU and 100GB main memory. To evaluate the semi-external algorithms, we test them on a machine with 3.4GHz four-core CPU and a manually set small memory capacity, so the tested massive graphs cannot fit into memory.

**Data sets**. We use 7 publicly available real-world networks to evaluate the algorithms. The network statistics are shown in Table 2. Except for Wise[1] (a microblogging network from WISE 2012 Challenge), UK2002[2] (a web graph from a 2002 crawl of the .uk domain) and WebBase[2] (a web graph from the 2001 crawl performed by the WebBase crawler), all the other data sets are downloaded from the Stanford Network Analysis Project[3]. All networks are treated as undirected in the experiments.

### 6.1 In-Memory K-Truss Community Search Performance

#### *6.1.1 Query Processing*

We evaluate and compare the performance of the two in-memory $k$-truss community query algorithms: Algorithm 2 that uses the simple $k$-truss index and Algorithm 4 that uses the TCP-Index.

In the first experiment, we select query vertices with different degrees to test the query processing time. For each network, we sort the vertices in descending order of their degrees and partition them into 10 equal-sized buckets. We randomly select 100 vertices from each bucket for query. The average query processing time for each degree group is reported in Figure 10. We fix $k = 10$ for all networks except for WikiTalk and Wise which use $k = 4$, because the edges in these two networks have smaller trussness. As we can see, for the high degree query vertices which usually have larger and denser $k$-truss communities, the TCP-Index based method is two orders of magnitude faster than the $k$-truss index based method; whereas for the low degree query vertices which have smaller and sparser $k$-truss communities for the same $k$, the query time of the two methods is very close and is around a few milliseconds. This shows the superiority of the TCP-Index based query processing, especially for high degree query vertices.

In the second experiment, we vary the parameter $k$ to test the query time for $k$-truss community search. For each network, we randomly generate two test sets: a set of 100 high degree query vertices (degree in top 30%) and another set of 100 low degree query vertices (degree in the remaining 70%). We denote the two query methods on the high/low degree test sets as Truss-H/Truss-L and TCP-H/TCP-L, respectively. Figure 11 shows the average query processing time of each method when we vary the parameter $k$. As we can see, for the high degree query vertices, the TCP-Index based method is two to three orders of magnitude faster than the $k$-truss based method for all $k$ values; while for the low degree query vertices, the TCP-Index based method is still one to two orders of magnitude faster in most networks especially when $k$ is small. Finally, we can see that the query processing time decreases when $k$ increases, because the discovered communities become smaller when $k$ increases. This experiment again demonstrates the advantage of the TCP-Index based query processing and conforms with the time complexity analysis of the two query methods.

**Table 2** Network statistics (K = $10^3$ and M = $10^6$)

| Network | $\lvert V_G \rvert$ | $\lvert E_G \rvert$ | $d_{max}$ | $k_{gmax}$ |
|---|---|---|---|---|
| WikiTalk | 2.4**M** | 5**M** | 100029 | 53 |
| Flickr | 80**K** | 11.8**M** | 5706 | 308 |
| LiveJournal | 4.8**M** | 69**M** | 20333 | 362 |
| Orkut | 3.1**M** | 117.2**M** | 33313 | 78 |
| Wise | 58.6**M** | 265.1**M** | 278489 | 80 |
| UK2002 | 18.6**M** | 298.1**M** | 194955 | 944 |
| WebBase | 115.5**M** | 1709.6**M** | 816127 | 1507 |

[1] http://www.wise2012.cs.ucy.ac.cy/challenge.html
[2] http://law.di.unimi.it/datasets.php
[3] http://snap.stanford.edu/data/index.html

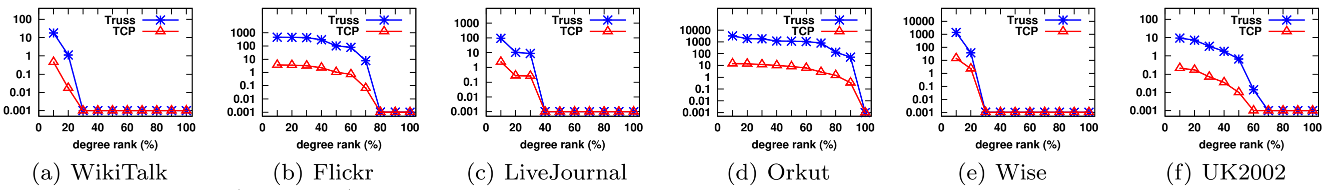


**Fig. 10** Query time (in seconds) of different algorithms for query vertices in different degree percentile groups

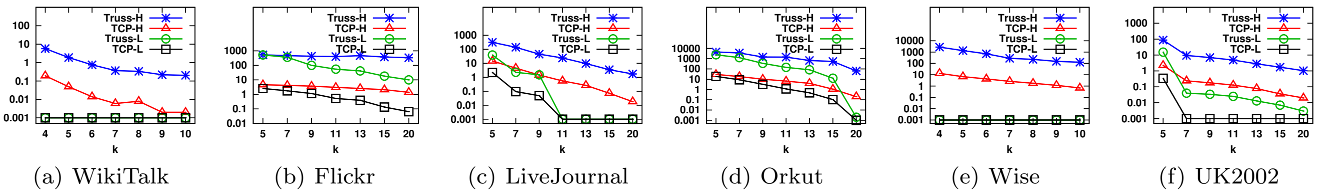


**Fig. 11** Query time (in seconds) of different algorithms for different $k$

*6.1.2 Index Construction*

In this experiment, we compare the two indexing schemes: the simple $k$-truss index and the TCP-Index in terms of index size and index construction time in Table 3. Note that the index is maintained in memory for both schemes. The reported index time includes the truss decomposition time and index construction time.

We can observe that the size of the TCP-Index is about 3 times that of the $k$-truss index, and 4.3 times of the original graph size. This confirms that both indexing schemes have $O(m)$ space complexity and are very compact. In addition, the index construction is very efficient for both schemes. The index time of the TCP-Index is around 1.4–3.4 times that of the $k$-truss index. The longest TCP-Index time is 1.9 hours on Wise.

*6.1.3 Scalability Test*

To evaluate the scalability of our proposed methods, we generate a series of power-law graphs using the Python-Web Graph Generator[4] [25], which implements a threaded variant of the RMAT algorithm. We vary $|V|$ from 1,000 to 10,000,000, and $|E| = 20|V|$. We select 100 vertices of the highest degree from each graph as the query nodes and set the trussness parameter $k = 4$.

Figure 12(a) shows the index construction time of the $k$-truss index and TCP-Index. Both methods scale very well with the vertex number. The construction time of TCP-Index is 2 times that of the $k$-truss index. Figure 12(b) reports the average query processing time. TCP-Index takes around 10 milliseconds to process one query in all networks. It is more than one order of magnitude faster than the $k$-truss method in query processing.

*6.1.4 Updating TCP-Index in Dynamic Graphs*

In this experiment, we evaluate the performance of incremental update of the TCP-Index when the input network is updated. For each network, we randomly insert/delete 1000 edges, and update the edge trussness

[4] `http://pywebgraph.sourceforge.net/`

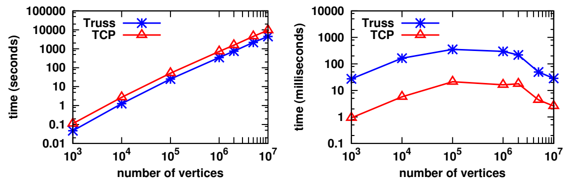


**Fig. 12** Scalability test

and the TCP-Index after each edge insertion/deletion. The average update time, including the edge trussness update time and the index update time, is reported in Table 4. In addition, we report the batch update time for the 1000 edge insertions/deletions. All the experiments are repeated for 20 times, and the average performance is reported. For comparison, we also report the time for constructing the TCP-Index from scratch when the network is updated with an edge insertion/deletion.

The results in Table 4 show that the update time per edge insertion ranges from 0.2 to 16.1 milliseconds. The batch update for 1000 edge insertions can achieve further performance improvement, compared with the instant update which handles the inserted edges one by one. Thus, handling edge insertion is highly efficient.

For the deletion case, the update time per edge deletion ranges from 3.9 to 38.8 milliseconds. The batch update for 1000 edge deletions also achieves further performance improvement. Compared with the insertion case, handling edge deletion is a little more costly, as it has a larger search space for finding a replacement edge in the TCP-Index.

We can see that the incremental update approach is several orders of magnitude faster than constructing the TCP-Index from scratch when a network is updated. This demonstrates the superiority of our proposed incremental update algorithms.

## 6.2 I/O-Efficient K-Truss Community Search Performance

In this experiment, we evaluate the efficiency of semi-external algorithms. We choose four big graphs: Orkut,

**Table 3** Comparison of index size (in Megabytes) and index construction time (wall-clock time in seconds)

| Network | Graph Size | Index Size | | Index Time | |
|---|---|---|---|---|---|
| | | K-Truss | TCP-Index | K-Truss | TCP-Index |
| WikiTalk | 80 | 118 | 296 | 41 | 138 |
| Flickr | 90 | 135 | 485 | 690 | 1326 |
| LiveJournal | 672 | 1003 | 3174 | 1176 | 1686 |
| Orkut | 1792 | 2662 | 8714 | 2291 | 3342 |
| Wise | 4209 | 5960 | 11049 | 3078 | 6997 |
| UK2002 | 4055 | 5980 | 21238 | 1374 | 2860 |

**Table 4** TCP-Index update time (wall-clock time in milliseconds)

| Network | Insertion Per Edge | Insertion 1000 Edges | Deletion Per Edge | Deletion 1000 Edges | Computing from Scratch |
|---|---|---|---|---|---|
| WikiTalk | 0.2 | 125 | 3.9 | 2509 | 138000 |
| Flickr | 10.2 | 6344 | 58 | 33763 | 1326000 |
| LiveJournal | 0.7 | 693 | 3.9 | 1891 | 1686000 |
| Orkut | 16.1 | 17190 | 29.6 | 21351 | 3342000 |
| Wise | 7.8 | 3902 | 38.8 | 31282 | 6997000 |
| UK2002 | 3.9 | 4065 | 12.2 | 12326 | 2860000 |

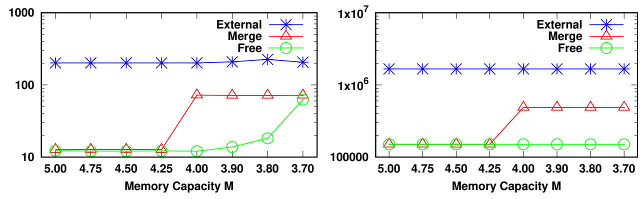
Wise, UK2002, and WebBase for testing. Under the semi-external model assumption, we set the memory size $M$ as a few times of the vertex size of a graph, but cannot hold all graph edges. We test the I/O efficiency of TCP-Index construction and query processing.

### *6.2.1 Index Construction*

**TCP-Index construction for a graph.** We test and compare three semi-external algorithms for TCP-Index construction: External-TCP, Merge-TCP, and Free-TCP. Table 5 reports the memory capacity, I/O cost, and running time of all three methods. We can observe that External-TCP is much slower than the other two methods. It takes more than 48 hours for External-TCP to construct TCP-Index on Wise, UK2002 and WebBase, thus it is terminated. Free-TCP performs the best in terms of both I/O cost and running time. Merge-TCP achieves good performances close to Free-TCP. This is because only a few vertices have large neighborhood induced subgraphs, whose sizes exceed the memory capacity. The difference between Merge-TCP and Free-TCP lies in computing TCP-Index for such vertices with large degree. For vertices with small degree, TCP-Index construction can be completed in memory for both Merge-TCP and Free-TCP.

**TCP-Index construction for a vertex with the largest neighborhood subgraph.** To further compare these three methods, we select one vertex with the largest neighborhood subgraph, and evaluate TCP-Index construction for this vertex. The I/O cost and running time are reported in Table 6. Free-TCP achieves the best performance with the smallest number of I/O costs and running time. On the other hand, External-TCP performs the worst. Compared with External-TCP, Merge-TCP reduces I/O cost and running time almost by half on Orkut, Wise and UK2002, and achieves substantially better efficiency on WebBase. Free-TCP and Merge-TCP achieve similar performance on WebBase, since the size of the neighborhood subgraph $G_x$ does not exceed the memory capacity.

**Vary memory capacity $M$.** In this experiment, we vary the memory capacity $M$ to compare the three TCP-Index construction methods on WebBase. We report the I/O cost and running time in Figure 13. As we can see, when the memory capacity decreases, External-TCP performs stably in terms of running time and I/O cost, because External-TCP always needs to store all edges of $G_x$ on disk. For Merge-TCP, when $G_x$ can fit into the memory ($M \geq 4.25$ GB), the running time and I/Os become the lowest. When the size of $G_x$ exceeds the memory capacity ($M < 4.25$ GB), Merge-TCP takes much higher running time and I/Os, because Merge-TCP needs to store edges on disk and reload them into memory to build TCP-Index. Free-TCP performs the best on all different $M$ values in terms of running time and I/Os. The I/O cost of Free-TCP remains stable for all different $M$ values. The running time of Free-TCP increases slightly when the memory capacity becomes small.



(a) Running Time (b) I/Os

**Fig. 13** Running time (in seconds) and I/Os of External-TCP, Merge-TCP, and Free-TCP versus the memory capacity $M$ (in GB) on WebBase.

### *6.2.2 Query Processing*

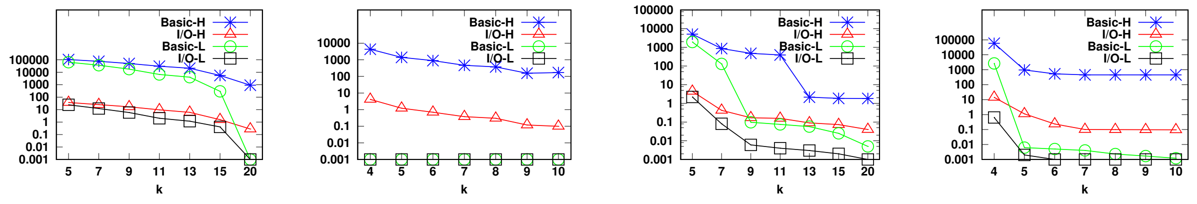
In this experiment, we evaluate the performance of semi-external query processing algorithms for $k$-truss community search. We compare two approaches. The first

**Table 5** Comparison of TCP-Index construction by three methods External-TCP, Merge-TCP, and Free-TCP, in terms of I/O costs and running time (in seconds). Here, $\mathbf{M} = 10^6$.

| Network | Memory Capacity | I/O Cost | | | Index Construction Time | | |
|---|---|---|---|---|---|---|---|
| | | External-TCP | Merge-TCP | Free-TCP | External-TCP | Merge-TCP | Free-TCP |
| Orkut | 150**MB** | 663M | 491**M** | 487**M** | 127057.8 | 6083.0 | 5892.8 |
| UK2002 | 750**MB** | – | 1262**M** | 1252**M** | – | 9382.5 | 8747.0 |
| Wise | 2000**MB** | – | 4680**M** | 4211**M** | – | 127971.8 | 76852.6 |
| WebBase | 5000**MB** | – | 7629**M** | 7583**M** | – | 44430.3 | 42732.9 |

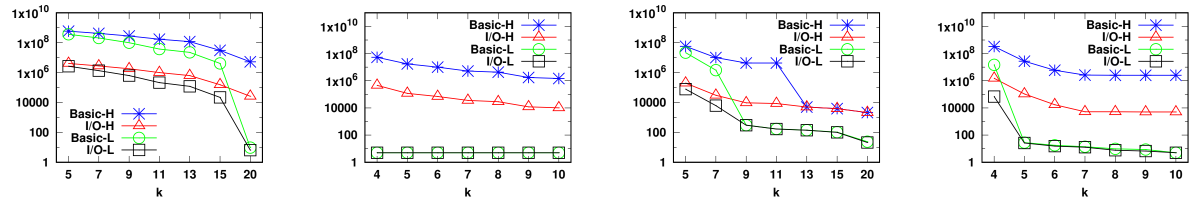
**Table 6** For one vertex with large degree, we test and compare TCP-Index construction for this vertex by three methods External-TCP, Merge-TCP, and Free-TCP, in terms of I/O cost and running time (in seconds). Here, $\mathbf{K} = 10^3$ and $\mathbf{M} = 10^6$.

| Network | Memory Capacity | I/O Cost | | | Index Construction Time | | |
|---|---|---|---|---|---|---|---|
| | | External-TCP | Merge-TCP | Free-TCP | External-TCP | Merge-TCP | Free-TCP |
| Orkut | 150**MB** | 200K | 86K | 53K | 20.113 | 13.727 | 1.456 |
| UK2002 | 750**MB** | 740K | 204K | 51K | 91.949 | 47.681 | 4.756 |
| Wise | 2000**MB** | 1484K | 915K | 752K | 112.522 | 63.707 | 18.535 |
| WebBase | 5000**MB** | 1666K | 150K | 150K | 201.967 | 12.691 | 12.304 |



(a) Orkut (b) Wise (c) UK2002 (d) WebBase

**Fig. 14** Query time (in seconds) of query processing algorithms for different $k$

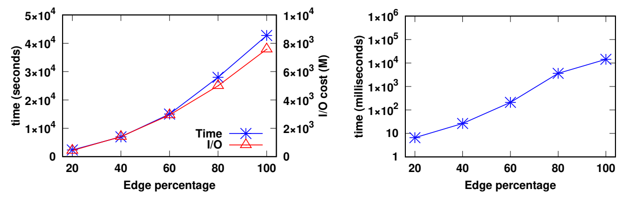


(a) Orkut (b) Wise (c) UK2002 (d) WebBase

**Fig. 15** I/O cost of query processing algorithms for different $k$

one is a variant of in-memory query processing algorithm (i.e., Algorithm 4), denoted by *basic*, which directly stores and accesses data on disk whenever the memory is used up. The second approach is our I/O-efficient algorithm in Algorithm 9, denoted by *I/O*. We select query nodes of high degree (degree in top 30%) and low degree (degree in the remaining 70%). To evaluate the effectiveness of Algorithm 9, we vary the parameter $k$ to find $k$-truss community for each query vertex. We report the average query time and the average number of I/Os respectively.

The results of query time by *basic* and *I/O* are reported in Figure 14. Compared with *basic*, our I/O-efficient algorithm performs two to four orders of magnitude faster for query nodes of high degree and low degree in most cases. The results show that the query processing of $k$-truss community search can be done in milliseconds to several seconds by our *I/O* algorithm. In addition, the results of I/O cost are reported in Figure 15. Similarly, our *I/O* method takes much fewer I/Os than *basic* and achieves two orders of magnitude of I/O cost saving in most cases.



(a) Index construction (b) Query processing

**Fig. 16** Efficiency test on WebBase with edge insertions

### *6.2.3 Performance Comparison between the Original Graph and Updated Graph*

In this experiment, we evaluate the effectiveness and efficiency of our I/O-efficient method Free-TCP between the original graph and the updated graph after edge insertions. This experiment is conducted on the largest graph WebBase. We simulate edge insertions by creating five subgraphs of WebBase which include 20%, 40%, 60%, 80%, and 100% edges respectively. For each subgraph, we use the same 100 high-degree query vertices (degree in top 30% in the complete WebBase graph) and set the trussness parameter $k = 4$ for finding 4-truss communities.

**Table 7** The graph statistics of five subgraphs of WebBase and the community quality measures. #Community denotes the total number of communities from 100 query vertices. Here M = $10^6$.

| Edge Percentage | $\lvert E_G \rvert$ | $d_{max}$ | $k_{gmax}$ | Discovered communities | |
|---|---|---|---|---|---|
| | | | | #Community | Clustering coefficient |
| 20% | 342**M** | 163514 | 44 | 12 | 0.095 |
| 40% | 684**M** | 326952 | 202 | 57 | 0.303 |
| 60% | 1026**M** | 489987 | 486 | 84 | 0.484 |
| 80% | 1368**M** | 652771 | 903 | 91 | 0.627 |
| 100% | 1710**M** | 816127 | 1507 | 95 | 0.767 |

To evaluate quality of the discovered communities, we calculate the average clustering coefficient. For a vertex $v_i$, its local clustering coefficient $c_i$ in a community $C$ is calculated as $c_i = \frac{2\{e_{jk}:v_j,v_k\in N_i, e_{jk}\in E(C)\}}{d_i(d_i-1)}$, where $N_i$ denotes the neighbors of vertex $v_i$ in $C$, $d_i$ is the degree of vertex $v_i$ in $C$, and $E(C)$ is the edge set of $C$. The clustering coefficient of a community $C$ is defined as the average local clustering coefficient of each vertex in $C$, i.e., $c = \frac{1}{|C|}\sum_{i=1}^{|C|} c_i$, where $|C|$ denotes the number of vertices in the community $C$.

Table 7 reports the statistics of the five subgraphs of WebBase in terms of the number of edge $|E_G|$, the maximum degree $d_{max}$, and the largest trussness $k_{gmax}$. We also report the number of discovered communities and the average clustering coefficient of the communities. We can see that as the graph includes more edges, there are more query vertices containing in 4-truss communities and the clustering coefficient of communities also becomes larger.

Figure 16(a) shows the running time and I/O cost of index construction by Free-TCP. As we can see, Free-TCP scales very well with the increased number of edges in terms of both running time and I/O cost. Figure 16(b) reports the average query time of our I/O-efficient community search method in Algorithm 9. It takes more time for community search on larger graphs as there are more discovered communities and the discovered communities have more edges too.

### *6.2.4 K-Truss Community Search Comparison: EquiTruss and Free-TCP*

EquiTruss proposed by Akbas and Zhao [1] is the latest $k$-truss community search method using the truss equivalence techniques. Due to the same truss-based community definition, EquiTruss finds the same communities as our method. In this experiment, we evaluate and compare the index construction and query processing time between EquiTruss and Free-TCP. We obtain the executable program of EquiTruss [1] from the authors. We randomly select 100 high degree query vertices (degree in top 30%) and set $k = 5$ for querying 5-truss communities.

Table 8 reports the memory usage, index construction time, query preprocessing time, and query time on four big graphs. We can observe that the memory consumption of EquiTruss is more than two orders of magnitude larger than that of Free-TCP. As a semi-external method, Free-TCP can construct TCP-Index in a very small memory capacity, which is even smaller than the graph size on disk. On the other hand, EquiTruss [1] requires a large memory space for index construction, and it fails to construct index on the largest graph WebBase on the server with 384GB memory capacity. In terms of index construction time, EquiTruss runs faster than Free-TCP on Orkut and Wise, but is slower than Free-TCP on UK2002.

Next we compare the query efficiency of Free-TCP and EquiTruss. In the implementation of EquiTruss, we observe that EquiTruss needs a one-off preprocessing step before answering queries as follows: it reads the entire graph and its constructed index into memory and then maps the index to the original graph edges. This preprocessing step consumes a long time. On the other hand, our TCP-Index does not need such a preprocessing step. The query time of the two methods is close. Free-TCP is faster than EquiTruss on Orkut and Wise, but is slightly slower on UK2002.

## 6.3 Comparison with Other Community Search Models

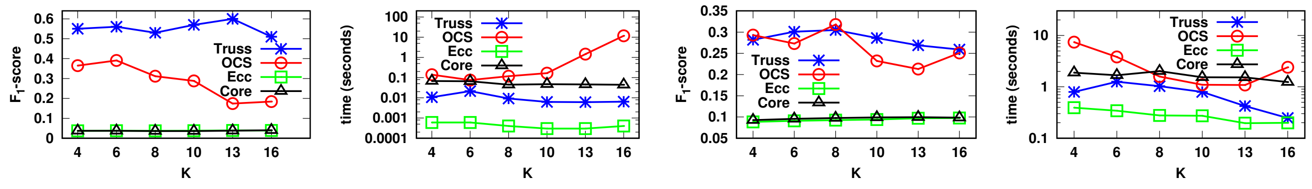
In this experiment, we compare the $k$-truss community model (the in-memory version) with three other community search models: quasi-clique (OCS) [7], $k$-core [8], and $k$-ECC [3]. We compare the efficiency and effectiveness of these four models on networks with and without ground-truth communities, as well as perform a case study on the DBLP collaboration network.

### *6.3.1 Comparison on Effectiveness and Efficiency*

**On networks with ground-truth communities.** We conduct the quality and efficiency evaluations on two social networks: Facebook and Twitter from the Stanford Network Analysis Project, both of which contain ground-truth communities for individual nodes. The Facebook network contains 4,039 vertices, 88,234 edges, and 10 query vertices with ground-truth communities in their neighborhood. The Twitter network contains 81,306 vertices, 1,768,149 edges, and 973 query vertices with ground-truth communities in their neighborhood. For a query, we denote the discovered com-

**Table 8** Comparison of EquiTruss [1] and the I/O-efficient algorithm Free-TCP, in terms of memory cost and running time (in seconds). Here, 1**GB** = 1024**MB**.

| Network | Graph Size | Memory Cost | | Index Construction Time | | Query Preprocess Time | | Query Time | |
|---|---|---|---|---|---|---|---|---|---|
| | | Free-TCP | EquiTruss | Free-TCP | EquiTruss | Free-TCP | EquiTruss | Free-TCP | EquiTruss |
| Orkut | 1.7**GB** | 150**MB** | 100.2**GB** | 5892.8 | 4299.9 | nil | 1210.9 | 119.7 | 120.4 |
| UK2002 | 4.9**GB** | 750**MB** | 133.3**GB** | 8747.0 | 10872.6 | nil | 1117.2 | 4.2 | 3.2 |
| Wise | 4.6**GB** | 2.0**GB** | 165.9**GB** | 76854.6 | 6839.1 | nil | 1943.2 | 4.3 | 29.4 |
| WebBase | 29.8**GB** | 5.0**GB** | - | 42732.9 | - | nil | - | 1.1 | - |



(a) Facebook $F_1$ score (b) Facebook query time (c) Twitter $F_1$ score (d) Twitter query time

**Fig. 17** Quality and efficiency comparison of four community models on networks with ground-truth communities

munities as $\mathcal{C} = \{C_1, \ldots, C_i\}$, and the ground-truth communities as $\overline{\mathcal{C}} = \{\overline{C}_1, \ldots, \overline{C}_j\}$. We use the $F_1$ score to measure the alignment between a discovered community $C$ and a ground-truth community $\overline{C}$. Since we do not know the correspondence between communities in $\mathcal{C}$ and $\overline{\mathcal{C}}$, we compute the optimal match via linear assignment [28] by $\max_{f:\mathcal{C}\to\overline{\mathcal{C}}} \frac{1}{|f|} \sum_{C \in dom(f)} F_1(C, f(C))$, where $f$ is a (partial) correspondence between $\mathcal{C}$ and $\overline{\mathcal{C}}$.

We compare the $F_1$ score of the four community search models using the executable programs provided by the respective authors. For $k$-truss, $k$-ECC and $k$-core models, we vary the parameter $k$; for the quasi-clique model, we follow the experimental setting in [7] to use the $(k-1,1)$-OCS model ($\alpha = k-1$, $\gamma = 1$) and vary the clique size $k$. Figures 17(a) and (c) show the $F_1$ score of the detected communities by the four methods versus their respective parameter $k$ on Facebook and Twitter. Although the parameter $k$ has different meanings in different models, the results still show the obvious differences in the $F_1$ score between our method and the others over a broad range of parameter values. On both networks, we observe that our $k$-truss model has a very stable performance when we vary the trussness $k$. The $(k-1,1)$-OCS model is consistently worse than our method. The $k$-ECC and $k$-core models have a very low $F_1$ score. The reason is that these two models only find one community for a query. However, there are more than one ground-truth community for a query and the discovered community only corresponds to one of the ground-truth communities. For the quasi-clique model, we also tried other $\alpha$ and $\gamma$ values. However, when we set $\alpha < k-1$ or $\gamma < 1$, the program cannot output all communities within the time limit set in the executable program due to the expensive quasi-clique enumeration.

Figures 17(b) and (d) report the average query time of these four models on Facebook and Twitter respectively. Our method is more efficient than the $(k-1,1)$-OCS and $k$-core models, but slower than $k$-ECC.

**On networks without ground-truth communities.** We also compare the above four community search models on networks without ground-truth communities. For a fair comparison, we set the parameter $k$ as the largest value such that communities can be found for a query vertex in each community search model respectively. We test these four community models on four networks: WikiTalk, Flickr, LiveJournal and a web graph Google with 875,713 vertices and 5,105,039 edges (downloaded from Stanford Network Analysis Project). We randomly select 100 high degree vertices (degree in top 30%) as the query vertices.

Figure 18 reports the results of the four methods. In terms of the index construction time, our $k$-truss model is slower than the $k$-core model but faster than $k$-ECC as shown in Figure 18(a). Note that the quasi-clique method OCS does not need to build any index. Figure 18(b) reports the query time. Our $k$-truss model is slower than $k$-ECC, but is faster than $k$-core and OCS. Note that OCS cannot finish the execution on LiveJournal in 5 days and output no community.

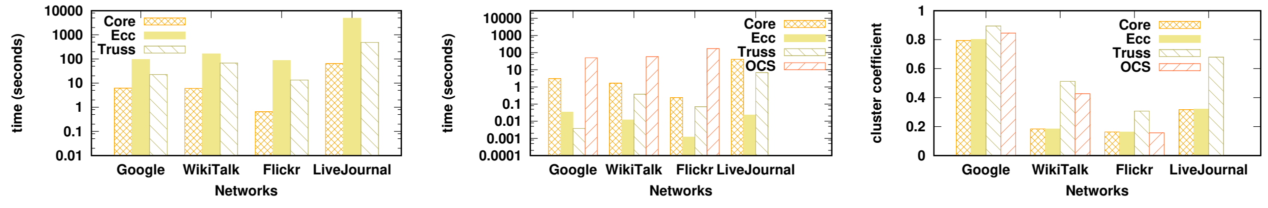
To evaluate the result quality, we report the average clustering coefficient of the discovered communities by all models in Figure 18(c). Our $k$-truss model always finds the communities with the highest cluster coefficient. Overall, our $k$-truss model achieves a good balance of fast index construction, query processing and higher community quality.

### 6.3.2 Case Study on DBLP

We build a collaboration network from the DBLP data set[5] for case study. A vertex represents an author and an edge is added between two authors if they have co-authored 3 times or more. The network contains 234,879 vertices and 541,814 edges.

We query the 5-truss communities containing 'Jiawei Han' which are shown in Figure 19. For OCS,

[5] `http://dblp.uni-trier.de/xml/`



(a) Index construction time (b) Query time (c) Clustering coefficient

**Fig. 18** Quality and efficiency comparison of four community models on networks without ground-truth communities

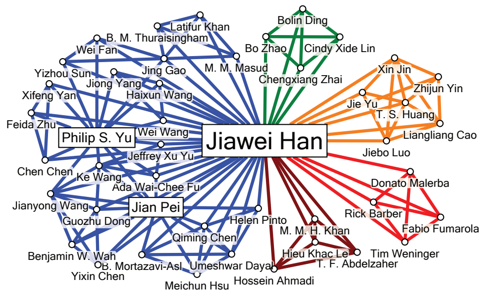


**Fig. 19** Five 5-truss communities containing 'Jiawei Han'

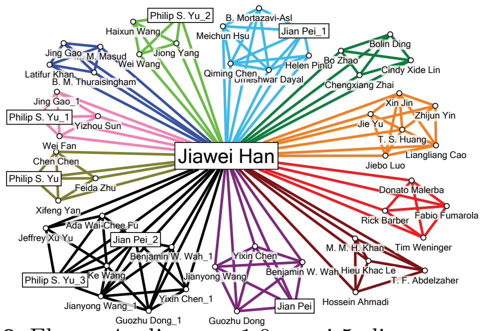


**Fig. 20** Eleven 4-adjacency-1.0-quasi-5-clique communities containing 'Jiawei Han'

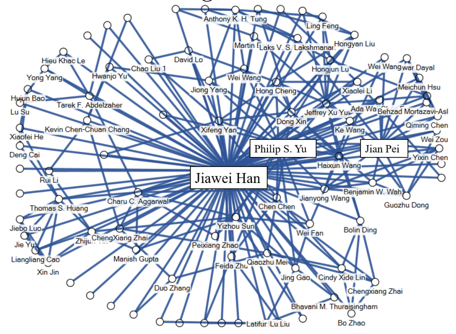


**Fig. 21** The 5-core community containing 'Jiawei Han'

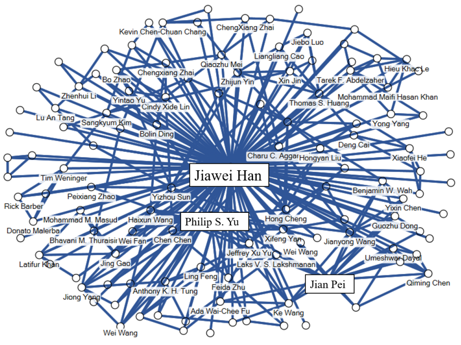


**Fig. 22** The 5-ECC community containing 'Jiawei Han'

we follow the case study setting in [7] which uses the $(k-1,1)$-OCS model to query 'Jiawei Han' by setting $k=5, \alpha=4, \gamma=1$ and produces communities at a similar scale as shown in Figure 20. We duplicate some authors who participate in more than one community in Figure 20, e.g., 'Jian Pei', 'Jian Pei_1' and 'Jian Pei_2', for a better visualization effect. We also query the 5-core and 5-ECC in Figures 21 and 22 respectively. Note that we only plot the one-hop neighbors of 'Jiawei Han' in the discovered communities by all four methods. We have the following observations.

(1) Our model generates 5 communities containing 'Jiawei Han', among which the 4 smaller ones are also found by the $(k-1,1)$-OCS model and depicted using the same color in Figure 20.

(2) The largest 5-truss community depicted in blue in Figure 19, however, is decomposed into 7 smaller communities by the $(k-1,1)$-OCS model in Figure 20. This phenomenon can be explained by the different mechanisms of the two community models. The $(k-1,1)$-OCS model tends to find the small, clique-based "*paper communities*", in which all the involved scholars appear in the same paper. For example, a paper community is formed by 'Jiawei Han', 'Philip S. Yu', 'Chen Chen', 'Xifeng Yan', and 'Feida Zhu'. In contrast, such small paper communities can be merged into a larger dense one by the triangle adjacency condition in our $k$-truss model. For example, two small paper communities can be merged if they share a common edge as ('Jiawei Han', 'Philip S. Yu') and form a 5-truss graph after being merged. In addition, we observe a community containing 'Guozhu Dong' and 5 other authors (depicted in purple) in Figure 20 is completely subsumed by another bigger community (depicted in black) in the same figure. Such duplicate output, which is not desired, may be explained by the approximate heuristics for clique enumeration and expansion in [7].

(3) A less restrictive community criterion can be realized by tuning $\alpha$ and $\gamma$ in [7]. But in our experiment, if we set $\alpha < k-1$ or $\gamma < 1$, it cannot output all communities within the time limit set in the executable program due to the expensive quasi-clique enumeration.

(4) With the same parameter of $k$, the core-based and ECC-based models find much larger communities compared with the truss-based model. In Figure 19, there are 45 vertices in the one-hop 5-truss communities when querying 'Jiawei Han'. In contrast there are 90 vertices and 119 vertices in the one-hop 5-core

and one-hop 5-ECC communities in Figures 21 and 22. In addition, the $k$-truss and OCS modelscan find more than one community for a query, but the $k$-core and $k$-ECC models can only find one community for a query.

## 7 Related Work

The related work to our study includes community search, $k$-truss mining, and I/O-efficient graph processing.

**Community Search**. Community search aims at finding query-dependent dense subgraphs over graphs. A comprehensive survey on community search can be found in [12,18]. There are numerous community search models based on different dense subgraph patterns, including $k$-core [31,8,11,42,13,44,41], $k$-truss [17,19,1,20,43,23,24], $k$-clique [32,7,38], $k$-edge connectivity component ($k$-ECC) [3,16], and query-biased densest subgraph [36]. Sozio and Gionis [31] first studied the problem of community search and proposed a $k$-core based model for online community search. Cui et al. [7] studied the problem of online search of overlapping communities for a query vertex by designing a new $\alpha$-adjacency $\gamma$-quasi-$k$-clique model. Wu et al. [36] proposed a query-biased densest subgraph based community search model to avoid the free-rider effect. Huang et al. [19] extended the $k$-truss community model [17] to find the closest $k$-truss community with the smallest diameter, which can avoid the free-rider effect. Various latest studies of community search conducted investigations on more complex graphs, such as attributed graphs [11,20,41,24], weighted graphs [42,43], and directed graphs [13,23]. Different from the above studies, to the best of our knowledge, our newly proposed I/O-efficient algorithms are the first work to address the problem of community search over massive graphs in a semi-external setting, where the entire graph cannot fit into the memory.

The most similar study to our work is [1]. Akbas and Zhao studied the same problem of $k$-truss community search and proposed a space-efficient and truss-preserving index called EquiTruss [1]. EquiTruss is a summarized graph built upon the $k$-truss equivalence classes, where each supernode represents an equivalence class of edges and each superedge represents that two end-point supernodes of components are triangle connected. For a given query, it needs to map the supernode in EquiTruss index to all the edges of the original graph first, which is time consuming. However, our query processing algorithms leverage TCP-Index to reconstruct the community edges only during the search process. Moreover, the total number of visited edges in the search process is proportional to the query community size but not the graph size, which makes our TCP-Index algorithms efficient for large graphs. A detailed experimental comparison between EquiTruss and our methods is presented in Section 6.2.4.

**K-Truss Mining**. $k$-truss [6] is an important motif based on triangles in a graph, requiring that each edge is contained in at least $k-2$ triangles within the $k$-truss. $k$-truss decomposition [34] finds the $k$-trusses for all values of $k$ in a graph. $k$-truss mining has also been studied on various kinds of graphs, including directed graphs [23], probabilistic graphs [21], public-private networks [9], and dynamic graphs [17,40]. The problem of truss maintenance is to compute the trussness of all edges when a graph updates with vertex/edge insertions/deletions. Zhang and Parthasarathy [39] designed an incremental algorithm for updating triangle $k$-core (equivalent to $k$-truss). Recently, Zhang and Yu [40] haven shown the truss maintenance is bounded for edge deletions but is unbounded for edge insertions. Based on the truss decomposition order, they proposed an efficient truss maintenance algorithm on dynamic graphs.

**I/O-Efficient Graph Processing**. Many I/O-efficient algorithms have been proposed to solve various graph analysis problems. I/O-efficient algorithms have been proposed for core decomposition [4] and truss decomposition [34]. Wen et al. [35] proposed a more efficient semi-external algorithm for core decomposition and core maintenance over dynamic graphs. More I/O-efficient algorithms can be found in the survey [26].

## 8 Conclusions

In this paper, we study the online community search problem in a network. We propose a novel $k$-truss community model based on the $k$-truss concept which is shown to have cohesive and hierarchical community structure. To support the efficient search of $k$-truss community, we design a novel and compact tree-shape index, called the TCP-Index, which preserves the edge trussness and the triangle adjacency relationship, and supports community search in linear time with respect to the community size. We further study the $k$-truss community search in dynamic graphs and propose efficient incremental algorithms to update the index. Moreover, we propose I/O-efficient algorithms to construct TCP-Index and query $k$-truss communities for massive graphs where the entire graph cannot fit into the main memory. We conduct extensive experiments on large real-world networks, and the results demonstrate the effectiveness and efficiency of the proposed algorithms.

**Acknowledgements** The work was supported by grants from the Research Grant Council of the Hong Kong Special Administrative Region, China [Project No.: CUHK 14205617], [Project No.: CUHK 14205618], and [Project No.: HKBU 22200320], and NSFC Grant Nos. U1936205 and 61702435.

## References

1. Akbas, E., Zhao, P.: Truss-based community search: a truss-equivalence based indexing approach. PVLDB **10**(11), 1298–1309 (2017)
2. Boldi, P., Vigna, S.: The WebGraph framework I: Compression techniques. In: WWW, pp. 595–601 (2004)
3. Chang, L., Lin, X., Qin, L., Yu, J.X., Zhang, W.: Index-based optimal algorithms for computing steiner components with maximum connectivity. In: SIGMOD, pp. 459–474 (2015)
4. Cheng, J., Ke, Y., Chu, S., Özsu, M.T.: Efficient core decomposition in massive networks. In: ICDE, pp. 51–62 (2011)
5. Chiba, N., Nishizeki, T.: Arboricity and subgraph listing algorithms. SIAM J. Comput. **14**(1), 210–223 (1985)
6. Cohen, J.: Trusses: Cohesive subgraphs for social network analysis. Tech. rep., National security agency technical report (2008)
7. Cui, W., Xiao, Y., Wang, H., Lu, Y., Wang, W.: Online search of overlapping communities. In: SIGMOD, pp. 277–288 (2013)
8. Cui, W., Xiao, Y., Wang, H., Wang, W.: Local search of communities in large graphs. In: SIGMOD, pp. 991–1002 (2014)
9. Ebadian, S., Huang, X.: Fast algorithm for k-truss discovery on public-private graphs. In: IJCAI, pp. 2258–2264 (2019)
10. Edachery, J., Sen, A., Brandenburg, F.J.: Graph clustering using distance-k cliques. In: International Symposium on Graph Drawing, pp. 98–106 (1999)
11. Fang, Y., Cheng, R., Chen, Y., Luo, S., Hu, J.: Effective and efficient attributed community search. VLDB J. **26**(6), 803–828 (2017)
12. Fang, Y., Huang, X., Qin, L., Zhang, Y., Zhang, W., Cheng, R., Lin, X.: A survey of community search over big graphs. VLDB J. **29**(1), 353–392 (2020)
13. Fang, Y., Wang, Z., Cheng, R., Wang, H., Hu, J.: Effective and efficient community search over large directed graphs. IEEE Trans. Knowl. Data Eng. **31**(11), 2093–2107 (2018)
14. Hartmanis, J.: Computers and intractability: a guide to the theory of np-completeness. Siam Review **24**(1), 90 (1982)
15. Hartuv, E., Shamir, R.: A clustering algorithm based on graph connectivity. Inf. Process. Lett. **76**(4–6), 175–181 (2000)
16. Hu, J., Wu, X., Cheng, R., Luo, S., Fang, Y.: On minimal steiner maximum-connected subgraph queries. IEEE Trans. Knowl. Data Eng. **29**(11), 2455–2469 (2017)
17. Huang, X., Cheng, H., Qin, L., Tian, W., Yu, J.X.: Querying k-truss community in large and dynamic graphs. In: SIGMOD, pp. 1311–1322 (2014)
18. Huang, X., Lakshmanan, L.V., Xu, J.: Community Search over Big Graphs. Morgan & Claypool Publishers (2019)
19. Huang, X., Lakshmanan, L.V., Yu, J.X., Cheng, H.: Approximate closest community search in networks. PVLDB **9**(4), 276–287 (2015)
20. Huang, X., Lakshmanan, L.V.S.: Attribute-driven community search. PVLDB **10**(9), 949–960 (2017)
21. Huang, X., Lu, W., Lakshmanan, L.V.S.: Truss decomposition of probabilistic graphs: Semantics and algorithms. In: SIGMOD, pp. 77–90 (2016)
22. Leu, F., Tsai, Y., Tang, C.Y.: An efficient external sorting algorithm. Inf. Process. Lett. **75**(4), 159–163 (2000)
23. Liu, Q., Zhao, M., Huang, X., Xu, J., Gao, Y.: Truss-based community search over large directed graphs. In: SIGMOD, pp. 2183–2197 (2020)
24. Liu, Q., Zhu, Y., Zhao, M., Huang, X., Xu, J., Gao, Y.: Vac: Vertex-centric attributed community search. In: ICDE, pp. 937–948 (2020)
25. Macropol, K., Singh, A.: Scalable discovery of best clusters on large graphs. PVLDB **3**(1-2), 693–702 (2010)
26. Maheshwari, A., Zeh, N.: A survey of techniques for designing i/o-efficient algorithms. In: Algorithms for Memory Hierarchies, pp. 36–61 (2002)
27. Malliaros, F., Giatsidis, C., Papadopoulos, A., Vazirgiannis, M.: The core decomposition of networks: Theory, algorithms and applications. VLDB J. **29**(1), 61–92 (2020)
28. McAuley, J.J., Leskovec, J.: Learning to discover social circles in ego networks. In: NIPS, pp. 548–556 (2012)
29. Newman, M.E., Girvan, M.: Finding and evaluating community structure in networks. Physical review E **69**(2), 026113 (2004)
30. Sarıyüce, A.E., Gedik, B., Jacques-Silva, G., Wu, K.L., Çatalyürek, Ü.V.: Streaming algorithms for k-core decomposition. PVLDB **6**(6), 433–444 (2013)
31. Sozio, M., Gionis, A.: The community-search problem and how to plan a successful cocktail party. In: KDD, pp. 939–948 (2010)
32. Tsourakakis, C., Bonchi, F., Gionis, A., Gullo, F., Tsiarli, M.: Denser than the densest subgraph: extracting optimal quasi-cliques with quality guarantees. In: KDD, pp. 104–112 (2013)
33. Ugander, J., Backstrom, L., Marlow, C., Kleinberg, J.: Structural diversity in social contagion. Proc. Natl. Acad. Sci. **109**(16), 5962–5966 (2012)
34. Wang, J., Cheng, J.: Truss decomposition in massive networks. PVLDB **5**(9), 812–823 (2012)
35. Wen, D., Qin, L., Zhang, Y., Lin, X., Yu, J.X.: I/o efficient core graph decomposition: Application to degeneracy ordering. IEEE Trans. Knowl. Data Eng. **31**(1), 75–90 (2019)
36. Wu, Y., Jin, R., Li, J., Zhang, X.: Robust local community detection: on free rider effect and its elimination. PVLDB **8**(7), 798–809 (2015)
37. Xie, J., Kelley, S., Szymanski, B.K.: Overlapping community detection in networks: The state-of-the-art and comparative study. ACM Comput. Surv. **45**(4), 43 (2013)
38. Yuan, L., Qin, L., Zhang, W., Chang, L., Yang, J.: Index-based densest clique percolation community search in networks. IEEE Trans. Knowl. Data Eng. **30**(5), 922–935 (2017)
39. Zhang, Y., Parthasarathy, S.: Extracting analyzing and visualizing triangle k-core motifs within networks. In: ICDE, pp. 1049–1060 (2012)
40. Zhang, Y., Yu, J.X.: Unboundedness and efficiency of truss maintenance in evolving graphs. In: SIGMOD, pp. 1024–1041 (2019)
41. Zhang, Z., Huang, X., Xu, J., Choi, B., Shang, Z.: Keyword-centric community search. In: ICDE, pp. 422–433 (2019)
42. Zheng, D., Liu, J., Li, R.H., Aslay, C., Chen, Y.C., Huang, X.: Querying intimate-core groups in weighted graphs. In: IEEE ICSC, pp. 156–163 (2017)
43. Zheng, Z., Ye, F., Li, R.H., Ling, G., Jin, T.: Finding weighted k-truss communities in large networks. Inf. Sci. **417**, 344–360 (2017)
44. Zhu, R., Zou, Z., Li, J.: Diversified coherent core search on multi-layer graphs. In: ICDE, pp. 701–712 (2018)