




A D-Truss-Equivalence Based Index for Community Search Over Large Directed Graphs

Wei Ai , Canhao Xie , Tao Meng , Jayi Du , and Keqin Li , *Fellow, IEEE*

Abstract—Community Search (CS) aims to enable online and personalized discovery of communities. Recently, attention to the CS problem in directed graphs (di-graph) needs to be improved despite the extensive study conducted on undirected graphs. Nevertheless, the existing studies are plagued by several shortcomings, e.g., Achieving high-performance CS while ensuring the retrieved community is cohesive is challenging. This paper uses the D-truss model to address the limitations of investigating the CS problem in large di-graphs. We aim to implement millisecond-level D-truss CS in di-graphs by building a summarized graph index. To capture the interconnectedness of edges within D-truss communities, we propose an innovative equivalence relation known as D-truss-equivalence, which allows us to divide the edges in a di-graph into a sequence of super nodes (s-nodes). These s-nodes form the D-truss-equivalence-based index, DEBI, an index structure that preserves the truss properties and ensures efficient space utilization. Using DEBI, CS can be performed without time-consuming access to the original graph. The experiments indicate that our method can achieve millisecond-level D-truss community query while ensuring high community quality. In addition, dynamic maintenance of indexes can also be achieved at a lower cost.

Index Terms—Community search, directed graphs, D-truss, D-truss-equivalence.

I. INTRODUCTION

OVER the past decade, there has been substantial growth in the quantity of intricate data that can be depicted and analyzed through graphs. In real-world network applications, the potential graphs often showcase inherent community structures critical in diverse, interconnected processes. As a result, community detection (CD) [13], [16], [31] has emerged as a highly researched problem in graph management and analytics. The primary goal of community detection is to uncover densely interconnected subgraphs that unveil latent and crucial community structures within graphs. CS [9], [14], [19], [27] is a variant version of the renowned CD that utilizes query nodes for online

personalized community exploration. This method has been widely employed in real-world scenarios involving large-scale graphs, e.g., for any social platform, we can build it into a network where each user and the relationship between them can be described as a vertex and edge. We can recommend friends or push advertisements to a user by searching the community where the user belongs.

Depending on whether the directionality of edges is considered, graphs can be divided into two types: undirected [6], [14], [26] and directed [10], [12], [28]. Correspondingly, the community structure can be partitioned into communities in di-graphs and undirected graphs. The difference between the two is that communities in di-graphs need to consider directional information such as out-degree and in-degree. In contrast, communities in undirected graphs mainly focus on the interconnection relationships between nodes. Extensive research has been dedicated to exploring CS in large graphs, with most studies focusing on undirected graphs. Nevertheless, di-graphs are typical in various fields. However, existing CS methods for undirected graphs cannot be applied to di-graphs. For example, the k -core method cannot identify the out-degree and in-degree of nodes in a di-graph, and the k -truss method cannot distinguish different types of triangles. Some models, e.g., D-core and CF-truss [10], [28] have been proposed to process the discovery of communities in di-graphs. However, the D-core model possesses a notable limitation. For instance, in some graphs, the out-degree and in-degree of different nodes may vary significantly. It may form sparse communities when attempting to include these nodes. CF-truss models the two types of triangles separately, resulting in a vast community. Its practicality for real-world limited.

Huang et al. [22] have investigated CS in di-graphs using a D-truss ((k_c, k_f) -truss) model, which has a solid structure and cohesiveness. In particular, D-truss incorporates cycle and flow triangles as fundamental components. Di-graphs can comprise cycle and flow triangles. The difference between cycle and flow triangles is whether the interior constitutes a loop. Fig. 1 shows two types of triangles. In a D-truss, each edge must constitute cycle (flow) triangles with at least k_c (k_f) vertices. Building upon the D-truss model, they introduced the D-truss CS (DCS) concept and proved its NP-hardness. To address this problem efficiently, they devised two algorithms, e.g., *iLocal* and *iGlobal* [22]. Subsequently, they devised the D-truss index to obtain the maximal D-truss (M-D-truss). Specifically, they present an algorithm capable of computing all the potential D-trusses within a di-graph. The outcomes of the D-truss decomposition were then stored in an index to discover the M-D-truss. Finally,

Manuscript received 4 July 2023; revised 16 April 2024; accepted 22 April 2024. Date of publication 26 April 2024; date of current version 27 September 2024. This work was supported in part by the National Natural Science Foundation of China under Grant 69189338, in part by Excellent Young Scholars of Hunan Province of China under Grant 22B0275, and in part by Changsha Natural Science Foundation under Grant kq2202294. Recommended for acceptance by S. Salihoglu. (Corresponding author: Tao Meng.)

Wei Ai, Canhao Xie, Tao Meng, and Jayi Du are with the College of Computer and Mathematics, Central South University of Forestry and Technology, Changsha, Hunan 410004, China (e-mail: aiwei@hnu.edu.cn; xiecanhao@csuft.edu.cn; mengtao@hnu.edu.cn; dujiayi@csuft.edu.cn).

Keqin Li is with the Department of Computer Science, State University of New York, New Paltz, NY 12561 USA (e-mail: lik@newpaltz.edu).

Our code is available at <https://github.com/XieCanhao04/DEBI>.

Digital Object Identifier 10.1109/TKDE.2024.3393864

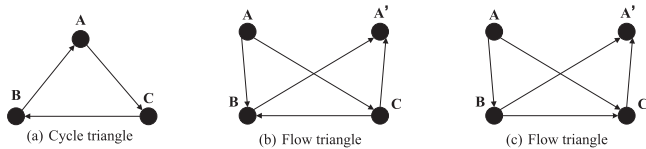


Fig. 1. Different types of triangles in di-graphs. (a) Cycle triangle formed by vertices A, B, and C. (b) Two flow triangles formed by edges $\langle C, B \rangle$, and nodes A, and A'. (c) After changing the direction of edges $\langle B, C \rangle$, we get two completely different flow triangles.

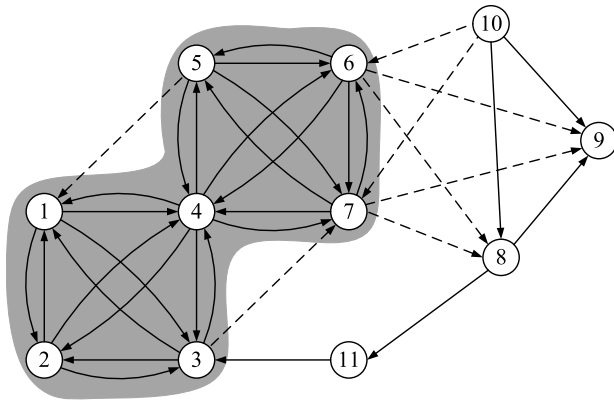


Fig. 2. Given a query vertex $q=4$, two integers $k_c=2$ and $k_f=2$, we can get a $M(2, 2)$ -truss as shown in the gray area. Dashed marks represent the invalid edge access.

find the optimal solution for the maximum D-truss through *Local* and *Global*. However, *iLocal* and *iGlobal* can suffer two significant limitations: (1) Both *Local* and *Global* seek the optimal solution iteratively, which is time-costly. (2) Inefficiency of obtaining M-D-truss in the larger graph. On the one hand, if an edge's skyline trussnesses (ST) do not meet the requirements, accessing and checking this edge will become a waste. On the other hand, for any two vertices connected by a series of edges with their ST dominate (k_c, k_f) , they are contained in the same (k_c, k_f) -truss community (ST is introduced in Section IV). However, during the M-D-truss search by the D-truss index, we need to examine all these edges and their ST. Consider an example in Fig. 2. We can get a $(2, 2)$ -truss with query vertex "4" through the D-truss index, as shown in the gray area in Fig. 2. In this process, the algorithm needs to access and operate on the dashed mark edges and the edges in the gray area. Among them, the dashed mark edges do not satisfy the condition, so access and operation on them are invalid. In addition, the edges in the gray area all have the ST of $(2, 2)$, but the algorithm still needs to visit and operate each edge, which is undoubtedly very time-consuming. We can group these edges with shared characteristics into one class and compress them into a super-node (s-node). In this case, we only need to find the s-nodes that meet the conditions and decompress them when doing a CS. This way, we do not need to do the invalid and repeated access and operation.

To overcome the constraints of previous research, we introduce a novel indexing method for solving the DCS problem. We chose D-truss as the base community model is affected by its

robust and stable structure. We propose a novel definition called D-truss-equivalence to obtain the interconnectedness of edges in D-truss communities. By employing this innovative definition, we have the capability to divide any di-graph into s-nodes that maintain the D-truss information. In D-truss-equivalence, we devise a robust triangle-connectivity constraint: (k_c, k_f) -triangle connectivity to ensure the cohesion of the D-truss communities. Next, we formulate the CS problem using D-truss-equivalence and develop a D-truss-equivalence-based index, DEBI. Subsequently, we present an approach leveraging DEBI to identify D-truss communities containing the query vertex q effectively. CS can conduct directly on DEBI. The DEBI evaluation encompasses theoretical and experimental analyses to gauge its quality and performance. Furthermore, DEBI offers formidable CS capabilities in large-scale graphs. Specifically, our method is 2 to 5 orders of magnitude better than the *iLocal* and *iGlobal*, which also deals with the DCS problem. In addition, Our method can achieve millisecond-level DCS in di-graph.

We encapsulate the main contributions in this paper:

- 1) We present a novel definition termed D-truss equivalence to capture the interconnectedness among edges in D-truss communities. By employing this innovative concept, any di-graph can be divided into s-nodes, facilitating efficient CS.
- 2) We develop and establish DEBI, an index characterized by space efficiency and cost-effectiveness. CS can be conducted directly on DEBI, eliminating the need to access the original graph, representing a theoretically optimal approach. In addition, we designed a dynamic maintenance strategy for the index so that the index can be dynamically updated at a lower cost.
- 3) We conduct comprehensive experiments on real-world networks. The results verify that our method has the best query efficiency, can achieve millisecond-level CS in most cases, and outperforms most baseline methods regarding retrieved community quality.

This paper is structured as follows:

Section II presents a comprehensive review of the related work. Section III illustrates the main ideas of the DCS problem and their theoretical assessment. Section IV introduces a novel concept called D-truss-equivalence, the basis for D-truss-based CS. Section V proposes an index based on D-truss-equivalence and an index-based method for CS. The experimental findings are presented in Section VI, while Section VII offers the concluding remarks for this paper.

II. RELATED WORK

Community Detection (CD): Communities are subgraphs in a network where vertices exhibit dense connections, forming cohesive groups. *Community structure* is a general characteristic found in numerous complex networks, particularly in social networks [24]. CD aims to identify community structure within a network [11], [25]. Most traditional techniques are developed based on statistical inference and machine learning, such as spectral clustering and statistical inference, which are suitable

for smaller networks and more straightforward scenarios. However, Better techniques are needed when dealing with graphs and their properties, large-scale networks, and high-dimensional data formed in dynamic environments. Compared with traditional techniques, deep learning can handle the CD problem in high-dimensional data situations [3], [4]. Liu et al. [21] summarized the existing methods from a technical perspective. However, CD must contend with diverse non-euclidean graph data among vertex elements, which cannot be handled well by traditional deep-learning models. Graph neural network [32], as a neural network for processing graph data, can compensate for this drawback. Overall, deep learning techniques provide highly flexible solutions for CD, and their powerful capabilities can tackle complex and diverse network structures. Therefore, deep learning is a new trend for CD.

Community Search: It is committed to solving the problem of discovering the community structure of a particular vertex or a group of vertices, also known as local DC. More specifically, when provided with a node q from a graph G , the goal is to identify all dense subgraphs within G that include vertex q . Currently, the primary CS methods are divided into methods based on k -core, k -truss, and k -clique [6], [14], [26]. In particular, k -core-based CS identifies communities where every vertex has a degree equal to or greater than k [2], [7]. Some indexing methods on k -core, such as the Coreness Centrality [26] method, divide nodes into different core levels. The core degree of a node indicates the highest k core level to which the node belongs. The higher the centrality, the stronger the centrality of the nodes in the network. Ensuring cohesion within k -core communities can be challenging. Ensuring cohesion within k -core communities can be challenging. To address this limitation and obtain cohesive communities, k -clique has also been explored for CS [30]. Because of the excessively limiting characteristics of the clique model, researchers have explored relaxed variants to address this limitation, such as k -truss [1]. The studies above primarily concentrate on analyzing undirected graphs.

Recent studies also concentrate on community analysis for di-graphs, such as D-core, D-truss, and so on [10], [22]. However, none of these approaches have resulted in cohesive communities. Recently, D-truss [22] has been proposed, and modeling communities use cycle and flow triangles [28]. Triangles are fundamental components for measuring robust and enduring community relationships. Each edge in a D-truss community belongs to k_c (k_f) cycle (flow) triangles. As a result, the obtained D-truss communities exhibit a high level of cohesion, supported by strong theoretical guarantees. Moreover, CS has been explored beyond simple graphs, extending to more intricate graph types, such as geo-social graphs [5], temporal graphs [20], weighted graphs [17], multi-valued graphs [18], and attributed graphs [15].

Graph Summarization [23]: It aims to find a set of concise hypergraphs or sparse graphs and clarify the primary structural information or changing trends of the original graphs instead of the original graph for data analysis. An excellent summarized graph covers the main structural features of the original graph, and its nodes are called s-nodes, which represent a set of similar

TABLE I
A PRIMER OF TERMINOLOGIES AND NOTATIONS

Notation	Description
$G_d = (V_{G_d}, E_{G_d})$	A directed, simple graph G_d
$G_s = (V_s, E_s)$	A summarized graph of G_d
$G'_s = (V'_s, E'_s)$	DEBI of G_d
$N_{G_d}(v)$	The neighbors of $v \in V_{G_d}$
$N_{G_d}^+(v)$	The in-neighbors of $v \in V_{G_d}$
$N_{G_d}^-(v)$	The out-neighbors of $v \in V_{G_d}$
$\deg_{G_d}(v)$	The degree of $v \in V_{G_d}$
$\deg_{G_d}^+(v)$	The in-degree of $v \in V_{G_d}$
$\deg_{G_d}^-(v)$	The out-degree of $v \in V_{G_d}$
$\text{csup}_{G_d}(e)$	The cycle-support of $e \in E_{G_d}$
$\text{fsup}_{G_d}(e)$	The flow-support of $e \in E_{G_d}$
Δ_{wvu}	A triangle constituted by vertices w, v, u
$\Delta_{w,v,u}^C$	A cycle-triangle constituted by vertices w, v, u
$\Delta_{w,v,u}^F$	A flow-triangle constituted by vertices w, v, u
$\text{ST}(e) = \{(k_{c1}, k_{f1}), \dots, (k_{cn}, k_{fn})\}$	ST of $e \in E_G$
$\Delta_s \longleftrightarrow \Delta_t$	Δ_s and Δ_t are triangle connected
$e_1 \longleftrightarrow e_2$	e_1 and e_2 are triangle connected
$\Delta_s \xrightarrow{(k_c, k_f)} \Delta_t$	Δ_s and Δ_t are (k_c, k_f) triangle connected
$e_1 \xrightarrow{(k_c, k_f)} e_2$	e_1 and e_2 are (k_c, k_f) triangle connected

vertices in the original graph; its edges are called super-edges (s-edges), which represent the connection between two node sets. In [1], the authors built the summarized graph in the undirected graph, significantly improving the performance of the k -truss CS. However, this method cannot be applied to di-graphs due to the directionality of edges in di-graphs. Cosum [33] transforms RDF (resource description framework) graphs into multi-type graphs and transforms the entity parsing problem into a multi-type graph summary problem.

For different application scenarios, different methods may need to be selected for optimization. To our knowledge, prior research has yet to investigate the interconnectedness among edges within D-truss communities, which motivated the development of DEBI, as introduced in this paper.

III. PRELIMINARIES

This section presents a comprehensive overview of the fundamental concepts associated with the CS problem in di-graphs. Table I summarizes the key terminologies and notations introduced in this paper.

Definition 1. (Cycle-Support): For an edge $e = \langle u, v \rangle \in E_{G_d}$, the cycle-support of e is $|\{w \text{ in } V_{G_d} : \Delta_{u,v,w}^C \text{ in } G\}|$, i.e. $\text{csup}_{G_d}(e)$.

Definition 2. (Flow-Support): For an edge $e = \langle u, v \rangle \in E_{G_d}$, the flow-support of e is $|\{w \text{ in } V_{G_d} : \Delta_{u,v,w}^F \text{ in } G_d\}|$, i.e. $\text{fsup}_{G_d}(e)$.

Specifically, given an edge e in G_d , the cycle support is the number of nodes in G_d that can form a cycle triangle with e , the flow support is the number of nodes in G_d that can form a flow triangle with e . In addition, we standardized the naming rules of a specific triangle to avoid ambiguity and we can intuitively understand the direction of each side inside it through the standardized name of a specific triangle. Given three nodes,

A, B, and C, they can form the only cycle triangle, $\Delta_{A,B,C}^C$, which forms a closed loop inside. That is, the edge direction of the edges inside $\Delta_{A,B,C}^C$ is always fixed. However, A, B, and C can form multiple flow triangles. For example, two different flow triangles can be formed by edges $\{ \langle A, B \rangle, \langle A, C \rangle, \langle B, C \rangle \}$ and $\{ \langle B, A \rangle, \langle A, C \rangle, \langle B, C \rangle \}$, but the vertices involved in forming them are all A, B and C. To avoid ambiguity, we arrange the vertices in the order of decreasing out-degree of each vertex in a particular flow triangle. Consider an example in Fig. 1(b), the out-degrees of nodes A, B, and C are 2, 0, and 1, respectively, so we name this flow triangle $\Delta_{A,C,B}^F$. In the same way, we name the flow triangle $\Delta_{C,B,A}^F$. In addition, after changing the direction of edges $\langle B, C \rangle$, we get two completely different flow triangles $\Delta_{A,B,C}^F$ and $\Delta_{B,C,A}^F$.

Definition 3. (D-truss): Given a subgraph $H_s \subseteq G_d$, H_s is a D-truss, also called (k_c, k_f) -truss, if $\forall e \in E_{H_s}$, $\text{csup}_{H_s}(e) \geq k_c$ and $\text{fsup}_{H_s}(e) \geq k_f$.

Example 2: Consider an example of a di-graph G_d , as displayed in Fig. 2. The edge $\langle 5, 1 \rangle$ constitutes one cycle triangles with vertex "4" in $\Delta_{5,1,4}^C$ and one flow-triangle in $\Delta_{5,1,4}^F$, respectively. Hence, $\text{csup}_{G_d}(\langle 5, 1 \rangle) = 1$ and $\text{fsup}_{G_d}(\langle 5, 1 \rangle) = 1$. As the edge $\langle 11, 3 \rangle$ does not participate in any cycle or flow triangles, $\text{csup}_{G_d}(\langle 11, 3 \rangle) = \text{fsup}_{G_d}(\langle 11, 3 \rangle) = 0$. The flow and cycle-support of all edges in the gray area in the graph is greater than 2. Therefore, the subgraph enclosed within the gray area is a (2, 2)-truss.

For two nodes u and v in G_d , $v \rightarrow u$ is used to indicate that there is a directed path between v and $u \in G_d$. The distance between v and $u \in G_d$, i.e., $\text{dist}_G(u, v)$, corresponds to the length of the shortest directed path between v and $u \in G_d$. $\text{dist}_G(u, v) = +\infty$, if there is not a directed path between v and u .

Definition 4. (Diameter): The diameter of a di-graph G_d is $\max_{u,v \in V_{G_d}} \text{dist}_{G_d}(u, v)$, i.e., $\text{diam}(G_d)$.

Based on above definitions, the problem of DCS has been defined.

Definition 5. (D-Truss CS(DCS)): For a di-graph $G_d(V_{G_d}, E_{G_d})$, a query node $q \in V_{G_d}$, and two integers k_c and k_f , DCS aims to discover the D-truss H'_S with the minimum diameter which q belongs.

The reason for considering the minimum diameter is to avoid the free-rider effect [22] and obtain a strong cohesive community. Therefore, the minimum diameter is used to eliminate these vertices. The DCS problem has been proven to be an NP-hard problem. To solve this, Huang designed two polynomial 2-approximation algorithms, i.e., *Global* and *Local* [22], which found the D-truss community with the shortest query distance in the M-D-truss.

Definition 6. (Query Distance): The bi-directed distance between the query node q and a node set V in G_d represents the distance between them, i.e., $\text{dist}_{G_d}(V, q)$.

In the *iGlobal* algorithm, the initial step involves utilizing the D-truss index to retrieve the M-D-truss encompassing the query vertex q . Subsequently, it progressively eliminates the vertex v farthest from the query vertex q through iterative steps. It preserves the remaining graph as a D-truss if it contains

Algorithm 1: D-Truss Decomposition.

Input: a di-graph $G_d = (V_{G_d}, E_{G_d})$

Output: ST of each edge in G_d

```

1: for each edge  $e \in E_{G_d}$  do
2:   Calculate  $\text{csup}_{G_d}(e)$  and  $\text{fsup}_{G_d}(e)$ ;
3:    $k_c \leftarrow 0$ ;  $k_f \leftarrow 0$ ;  $D_{(0,0)} \leftarrow G_d$ ;
4:    $L_e \leftarrow \emptyset$ ;
5:   while  $|G_d| \neq 0$  do
6:     for each edge  $e = \langle u, v \rangle \in E_{G_d}$  do
7:       if  $\text{csup}_{G_d}(e) \leq k_c$  then
8:          $L_e \leftarrow L_e.append(e)$ ;
9:       while  $|L_e| \neq 0$  do
10:         $e = \langle u, v \rangle = L_e.pop$ ;
11:         $G_d.remove(e)$ ;
12:        for each  $x \in N(u) \cup N(v)$  do
13:          for  $e_N \in \{ \langle x, v \rangle, \langle x, u \rangle, \langle v, x \rangle, \langle u, x \rangle \}$  do
14:            Recalculate  $\text{csup}_{G_d}(e_N)$  and  $\text{fsup}_{G_d}(e_N)$ ;
15:            if  $\text{csup}_{G_d}(e_N) \leq k_c$  then
16:               $L_e \leftarrow L_e.append(e_N)$ ;
17:           $k_c \leftarrow k_c + 1$ ;
18:           $D_{(k_c,0)} \leftarrow G$ ;
19:           $k_{cmax} \leftarrow k_c$ ;
20:        for  $k_c \leftarrow 0$  to  $k_{cmax}$  do
21:           $H_s \leftarrow D_{(k_c,0)}$ ;
22:          while  $|D_{(k_c,0)}| \neq 0$  do
23:             $e$  is the edge with the  $\min\{\text{fsup}_H(e) | e \in H_s\}$ ;
24:             $k_f \leftarrow \text{fsup}_{H_s}(e)$ ;
25:            Calculate  $\_Skyline\_Trussness(e, k_c, k_f)$ ;
26:            Delete  $\_Edge(e, k_c, k_f, H_s)$ ;
27:        Return  $\{ST(e) | e \in E_{G_d}\}$ ;
Procedure Calculate  $\_Skyline\_Trussness(e, k_c, k_f)$ 
28: for each trussness  $(k_{c0}, k_{f0}) \in ST(e)$  do
29:   if  $(k_{c0}, k_{f0}) \prec (k_c, k_f)$  then
30:      $ST(e).remove((k_{c0}, k_{f0}))$ ;
31:   if  $(k_c, k_f) \prec (k_{c0}, k_{f0})$  then
32:     Return;
33:  $ST(e) \leftarrow ST(e) \cup \{(k_c, k_f)\}$ ;
Procedure Delete  $\_Edge(e, k_c, k_f, H_s)$ 
34: Delete  $e = \langle u, v \rangle$  from  $H_s$ ;
35: for each incident edge  $e_h$  of  $u$  or  $v$  do
36:   Recalculate  $\text{csup}_{G_d}(e_h)$  and  $\text{fsup}_{G_d}(e_h)$ ;
37:   if  $\text{csup}_{H_s}(e_h) < k_c$  or  $\text{fsup}_{H_s}(e_h) < k_f$  then
38:     Calculate  $\_Skyline\_Trussness(e, k_c, k_f)$ ;
39:   Delete  $\_Edge(e, k_c, k_f, H_s)$ ;

```

q and qualifies as a D-truss. The output is determined by selecting the D-truss with the minimum query distance. Algorithm *iGlobal* and *iLocal* takes $O(\min\{k_{cmax}, k_{fmax}\} \cdot |E_{G_d}|^{1.5} + t \cdot |E_{G_m}|^{1.5})$ and $O(\min\{k_{cmax}, k_{fmax}\} \cdot |E_{G_d}|^{1.5} + \delta \cdot |E_{G_m}|^{1.5})$ time, respectively, where E_{G_m} is the M-D-truss which q belongs, t is the number of iterations incurred, and $\delta = \min\{\text{dist}_{G_m}(G_m, q), n\}$. Therefore, it is incapable of efficient CS in large-scale graphs with *iGlobal* and *iLocal* because of its high time complexity (TC).

IV. D-TRUSS-EQUIVALENCE

In order to systematically overcome the limitations of current approaches, we introduce a novel concept called D-truss-equivalence. This notion aims to capture the intrinsic relationship among edges that exhibit strong connectivity within a D-truss community. Consequently, a D-truss-equivalence-based index, DEBI, can theoretically be constructed to achieve high-performance CS with sufficient quality assurance. Before presenting our work, we need a step to decompose the di-graph G_d into a series of D-trusses in advance.

Definition 7. (Edge Trussness): For an edge $e \in E_{G_d}$, (k_c, k_f) is a trussness of e , if e belong to a (k_c, k_f) -truss, i.e., $(k_c, k_f) \in T(e)$.

Definition 8. (Trussness Dominance): For two trussnesses (k_c^1, k_f^1) and (k_c^2, k_f^2) of an edge e , trussness (k_c^1, k_f^1) dominates trussness (k_c^2, k_f^2) , denoted as $(k_c^2, k_f^2) \prec (k_c^1, k_f^1)$, if: $k_c^1 > k_c^2$ and $k_f^1 \geq k_f^2$; or (2) $k_c^1 \geq k_c^2$ and $k_f^1 > k_f^2$.

Note that, if there are two trussnesses (k_c^1, k_f^1) and (k_c^2, k_f^2) of e , $k_c^1 \geq k_c^2$ and $k_f^1 \geq k_f^2$, we denoted it as $(k_c^2, k_f^2) \preceq (k_c^1, k_f^1)$. The core idea of Trussness Dominance is similar to Coreness Centrality, which divides edges or nodes into different levels. The higher the level, the stronger the centrality of the edge or node. The key difference is that the two dimensions are different. One considers nodes, and the other considers edges.

Definition 9. (Skyline Trussness): For the trussnesses $T(e) = \{(k_c^1, k_f^1), (k_c^2, k_f^2), \dots, (k_c^n, k_f^n)\}$, the ST of edge e refer to the trussnesses values that are not dominated by others, i.e., $ST(e) = (k_c^i, k_f^i) \in T(e)$: not exists $(k_c^j, k_f^j) \in T(e)$, s.t., $(k_c^i, k_f^i) \prec (k_c^j, k_f^j)$.

Algorithm 1 can be used to compute the ST of edges in G_d . First, in the initialization process (Lines 1-2), the algorithm employs the triangle enumeration method to compute the cycle-support and flow-support of edges in G_d . Second, with k_f set to 0, k_{cmax} (Lines 5-18). Within the computation of a specific $D(k_c, 0)$, the algorithm iteratively eliminates edges with a cycle-support value lower than k_c (Lines 10-17). Third, for all D-trusses $D(k_c, 0)$ with k_c varying from 0 to k_{cmax} , the algorithm identifies the potential ST for each edge within $D(k_c, 0)$ (Lines 20-27). Finally, the algorithm outputs the ST of each edge in E_{G_d} . The TC of Algorithm 1 is $O(\min\{k_{cmax}, k_{fmax}\} \cdot |E_G|^{1.5})$ and its space complexity (SC) is $O(\min\{k_{cmax}, k_{fmax}\} \cdot |E_G|)$.

Example 2: We utilize Algorithm 1 to compute ST for all edges in the di-graph G_d . The computed results are shown in Fig. 3, where edges with different ST are depicted with different line types.

We note edge $\langle '1', '2' \rangle$ and edge $\langle '6', '7' \rangle$ connected through a series of edges with skyline trussness equal (2, 2) belong to the same maximal (2, 2)-truss. However, vertices "1" and "2" are not closely connected to vertices "6", and "7", and it is unreasonable to consider them as members of the same community. Therefore, we define a vital triangle-connectivity constraint: (k_c, k_f) -triangle connectivity, to get the cohesive communities, as follows.

Definition 10. ((k_c, k_f) -triangle): For a triangle $\Delta_{u,w,v} \subseteq G_d$, For each constituent edge e of $\Delta_{u,w,v}$, if there is a $T \in ST(e)$, $T \succeq (k_c, k_f)$, Δ_{uvw} is defined as a (k_c, k_f) -triangle.

Algorithm 2: Summarized Graph Construction.

Input: $G_d = (V_{G_d}, E_{G_d})$
Output: $G_s = (V_s, E_s)$ of G_d

- 1: D-Truss Decomposition(G_d);
- 2: $st \leftarrow \emptyset$;
- 3: **for** each $e \in E_{G_d}$ **do**
- 4: **if** $\exists T(e) \in ST(e) = d$ **then**
- 5: $\phi_d \leftarrow \phi_d \cup e$; $st \leftarrow st \cup \{d\}$;
- 6: **for** each $d \in st$ **do**
- 7: **for** each $e \in \phi_d$ **do**
- 8: $\phi_{d.e}.L_{id} \leftarrow \emptyset$; $\phi_{d.e}.visited \leftarrow \text{False}$;
- 9: $Id \leftarrow 0$;
- 10: **for** each d in st (d is not dominate by others) **do**
- 11: **while** $\exists e \in \phi_d$ **do**
- 12: $e.visited \leftarrow \text{True}$; $L \leftarrow \emptyset$;
- 13: $Id \leftarrow Id + 1$;
- 14: Create a s-node ν with $\nu.id \leftarrow Id$;
- 15: $V_s \leftarrow V_s \cup \{\nu\}$;
- 16: $L.append(e)$;
- 17: **while** $|L| \neq 0$ **do**
- 18: $e\langle u, v \rangle \leftarrow L.pop()$;
- 19: **if** $d = (0, 0)$ **then**
- 20: **for** each $id \in e.L_{id}$ **do**
- 21: $E_s \leftarrow E_s \cup \{\langle \mu, \nu \rangle\} // \mu.id = id$;
- 22: **foreach** incident edge e_h of e **do**
- 23: **if** $ST(e_h) \cap d = (0, 0)$ **then**
- 24: ProcessEdge1(e_h);
- 25: **else**
- 26: ProcessEdge2(e_h);
- 27: $\phi_d \leftarrow \phi_d - e$;
- 28: **if** $d \neq (0, 0)$ **then**
- 29: **for** each $id \in e.L_{id}$ **do**
- 30: $E_s \leftarrow E_s \cup \{\langle \mu, \nu \rangle\}$;
- 31: **for** each $x \in N(u) \cap N(v)$ **do**
- 32: **for** $\{(e_1, e_2) | e_1 \in \{\langle x, v \rangle, \langle v, x \rangle\} \cap E_{G_d}, e_2 \in \{\langle x, u \rangle, \langle u, x \rangle\} \cap E_{G_d}\}$ **do**
- 33: **if** $d \in ST(e_1) \cap ST(e_2)$ **then**
- 34: ProcessEdge1(e_1);
- 35: ProcessEdge1(e_2);
- 36: **if** $\exists \tau \in ST(e_1) \not\preceq d$ and $\exists \tau \in ST(e_2) \not\preceq d$ **then**
- 37: ProcessEdge2(e_1);
- 38: ProcessEdge2(e_2);
- 39: $\phi_d \leftarrow \phi_d - e$; $ST(e) \leftarrow ST(e) - d$
- 40: **if** $|ST(e)| = 0$ **then**
- 41: $E_{G_d} \leftarrow E_{G_d} - e$
- 42: $st \leftarrow st - d$;
- 43: **Return** $G_s = (V_s, E_s)$

Procedure ProcessEdge1(e):

- 44: **if** $\phi_{d.e}.visited = \text{False}$ **then**
- 45: $\phi_{d.e}.visited = \text{True}$;
- 46: $L.append(e)$;

Procedure ProcessEdge2(e):

- 47: **if** $Id \notin \phi_{d.e}.L_{id}$ **then**
- 48: $(e).L_{id} \leftarrow \phi_{d.e}.L_{id} \cup \{Id\}$

Definition 11. ((k_c, k_f) -triangle connectivity): For two (k_c, k_f) -triangles Δ_1 and Δ_2 in G_d , they are (k_c, k_f) -triangle connected, i.e., $\Delta_1 \xleftrightarrow{(k_c, k_f)} \Delta_2$, if Δ_1 and Δ_2 can be connected through a series of (k_c, k_f) -triangles, and for every edge e_c common to two adjacent triangles, $\exists T(e_c) \in ST(e_c)$, $T(e_c) = (k_c, k_f)$.

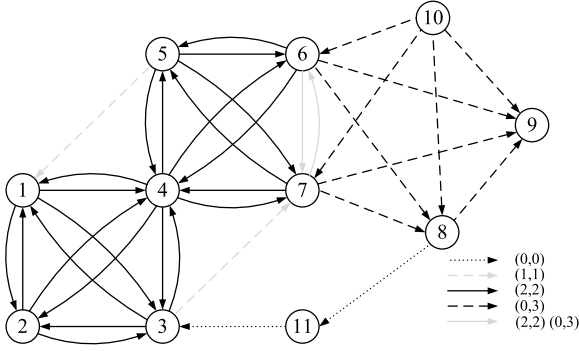


Fig. 3. The ST of edges in G_d . The trussness of edge e represents the number of flow triangles k_c and the number of cycle triangles k_f that can be formed with e in G_d . Edges with different ST are illustrated in different line types.

Example 3: For the di-graph G_d depicted in Fig. 3, and two $(0, 3)$ -triangles $\Delta_{7,8,9}^F$ and $\Delta_{10,6,9}^F$. They are $(0, 3)$ -triangle connected as there is a $(0, 3)$ -triangles $\Delta_{6,7,9}^F$, such that $\Delta_{10,6,9}^F \cap \Delta_{6,7,9}^F = \{(7, 9)\}$, $\Delta_{6,7,9}^F \cap \Delta_{10,6,9}^F = \{(6, 9)\}$, and for all these edges, there exists a $T(e) \in \text{ST}(e)$, $T(e) = (0, 3)$. However, the two $(2, 2)$ -triangles $\Delta_{4,5,6}^F$ and $\Delta_{1,2,3}^F$ are not $(2, 2)$ -triangle connected, Because neither $\Delta_{1,4,5}^C$ nor $\Delta_{1,4,5}^F$ is a $(2, 2)$ -triangle.

Similarly, given two edges $e_1, e_2 \in E_{G_d}$, they are (k_c, k_f) -triangle connected, i.e., $e_1 \xleftrightarrow{(k_c, k_f)} e_2$, if e_1 and e_2 contained in a same (k_c, k_f) -triangle, or $e_1 \in \Delta_1, e_2 \in \Delta_2$, s.t $\Delta_1 \xleftrightarrow{(k_c, k_f)} \Delta_2$. Based on the above prior knowledge, we proposed a new relation called D-truss equivalence, defined on the set of edges E_{G_d} , as follows:

Definition 12. (*D-truss-equivalence (Also called (k_c, k_f) -truss-equivalence*): Given two edges $e_1, e_2 \in E_{G_d}$, they are (k_c, k_f) -truss equivalent ($k_c \neq 0$ or $k_f \neq 0$), denoted as $e_1 \xleftrightarrow{(k_c, k_f)} e_2$, if and only if $(k_c, k_f) \in \text{ST}(e_1) \cap \text{ST}(e_2)$ and $e_1 \xleftrightarrow{(k_c, k_f)} e_2$.

It is important to note that any two edges in E_{G_d} with $\text{ST}(0, 0)$ are (k_c, k_f) -truss equivalent if they are connected through edges with $\text{ST}(0, 0)$.

Definition 13. (*D-Truss-Equivalence-Based CS*): For a di-graph $G_d(V_{G_d}, E_{G_d})$, a query node $q \in V_{G_d}$, and two integers k_c and k_f , the D-Truss-Equivalence-Based CS is to discovery all subgraphs $H_s \subseteq G_d$ satisfying:

- 1) $q \in V_{G_d}$;
- 2) H_s is a (k_c, k_f) -truss;
- 3) Every edge in H_s are (k_c, k_f) -triangle connective.

V. D-TRUSS EQUIVALENCE BASED INDEX

Based on D-truss-equivalence, we devise and construct a graph-structured index, DEBI, that supports CS with theoretically optimal performance.

A. Index Design and Construction

Utilizing the concept of D-truss equivalence, we partition all edges of di-graph G_d into a series of equivalence classes. Furthermore, we have developed a D-truss-equivalence-based index, DEBI. Our main idea is to use D-truss equivalence to

di-graph G_d to build a summarized graph that preserves the ST information of edges and the adjacency information of vertices in di-graph G_d . Afterward, the weighted index, DEBI, is obtained by building a maximum-spanning tree.

First, we build the summarized graph $G_s = (V_s, E_s)$ to preserve the ST information of the di-graph G_d . V_s represents a set of s-nodes, while E_s represents a set of s-edges. Each s-node $\nu \in V_s$ corresponds to a distinct equivalence class C_e . The s-edges, i.e., $(\mu, \nu) \in E_s$, capture the connections between s-nodes, representing the relationships between D-truss communities.

Example 4: The DEBI of the di-graph G_d (Fig. 2) is presented in Fig. 4(c). It contains six s-nodes, which means 6 D-truss equivalence classes. For example, the s-node ν_5 represents a $(3, 0)$ -truss community of 11 edges. These edges exhibit $(3, 0)$ -triangle connectivity and share the same ST value of $(3, 0)$. In addition, eight s-edges in DEBI depict the triangle connectivity between s-nodes, representing the relationships between D-truss communities.

We construct the summarized graph for a di-graph G_d based on D-truss equivalence in Algorithm 2. First, we invokes Algorithm 1 to get the ST for edges in E_{G_d} (Line 1). Then, we reassign edges to a distinct set Φ_d based on their ST, and we record all ST values in the variable st . (Lines 2-5). For an edge $e \in \Phi_d$, we maintain two auxiliary data structures: one is visited, used to indicate whether the edge e an edge e has been processed during index building. It is initially set to False (Line 8). The other is L_{id} , which is a collection of s-node tokens, with each token associated with an s-node that has been explored in the past, denoted as μ , where $T(\mu)$ does not dominate d , i.e., $d \not\prec T(\mu)$, μ is connected to the current s-node ν , $T(\nu) = d$. The set $\Phi_{d.e.L_{id}}$ is initialized as an empty set. (Line 8). When a value d is present in st and is not dominated by others, the algorithm then examines edges in Φ_d (Lines 10-39). When an edge $e \in \Phi_d$ is selected, a new s-node ν will be set to represent the equivalence class of e (Lines 11-15). When $d = (0, 0)$, by exploring the incident edges of edge e , we identify all edges that share D-truss equivalence with e and append them to the s-node ν (Lines 23-27). When $d \neq (0, 0)$, the algorithm explores the incident (k_c, k_f) -triangles of edge e to identify all edges that share D-truss equivalence with e . Subsequently, these edges are added to the s-node ν (Lines 31-39). Throughout the exploration process, we also check for the existence of an s-node μ in $\Phi_{d.e.L_{id}}$ that meets the conditions $T(\nu) \not\prec T(\mu)$ and μ is triangle-connected to ν through edge e . Upon finding such an s-node μ , a new s-edge (μ, ν) is established in the index (Lines 29-30). Given any incident triangle of e , and the other two edges of the triangle e_1 and e_2 , if exist a τ in $\text{ST}(e_1)$, $\tau \not\prec d$ and exist a τ in $\text{ST}(e_2)$, $\tau \not\prec d$, the identifier of the current s-node ν will be associated with e_1 and e_2 because ν is triangle-connected to the s-node μ , which e_1 and e_2 belong. Moreover, a s-edge (μ, ν) will be created when e_h is visited (Lines 45-46). Once e and all its incident triangles have been examined, e is deleted from Φ_k (Line 27 or 39) to ensure that each edge e in Φ_d belongs to at most one s-node.

Complexity Analysis: In Algorithm 2. The D-truss decomposition takes $O(\min\{k_{cmax}, k_{fmax}\} \cdot E_{G_d}^{1.5})$ time. For lines 12-37, considering an edge $e \in E_{G_d}$, and e belongs to n sets

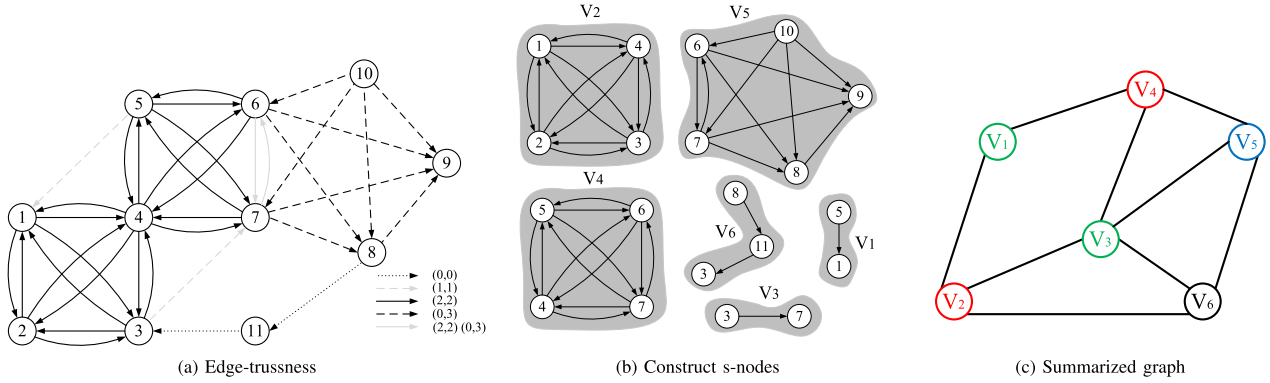


Fig. 4. The process of build the summarized graph. (a) Compute ST for edges $e \in E_{G_d}$ and divide di-graph G_d into equivalence classes. (b) Each s-node represents an equivalence class. (c) Construct the summarized graph of G_d . Each s-edge illustrates the connections relationship between s-nodes.

Algorithm 3: DEBI Construction.

Input: $G_s = (V_s, E_s)$
Output: $G'_s = (V'_s, E'_s)$

- 1: $W \leftarrow \emptyset$;
- 2: **for each** $\langle \mu, \nu \rangle \in E_s$ **do**
- 3: $w(\langle \mu, \nu \rangle) = (\min\{k_c(T(\mu)), k_c(T(\nu))\},$
 $\min\{k_f(T(\mu)), k_f(T(\nu))\})$;
- 4: $W \leftarrow W \cup w(\langle \mu, \nu \rangle)$;
- 5: $G_s = V_s$;
- 6: **for each** $w \in W$ (w is not dominated by any others)**do**
- 7: $S_w \leftarrow \{\langle \mu, \nu \rangle \mid \langle \mu, \nu \rangle \in E_s, w(\langle \mu, \nu \rangle) = w\}$;
- 8: **for each** $\langle \mu, \nu \rangle \in S_w$ **do**
- 9: **if** μ and ν are in different connected components
 in G'_s **then**
- 10: Add $\langle \mu, \nu \rangle$ with weight $w(\langle \mu, \nu \rangle)$ in G'_s ;
- 11: **Return** G'_s ;

of ϕ_d (where $n = |\text{ST}(e)|$). For each occurrence of edge e in ϕ_d , we identify all equivalent edges by examining all event triangles associated with e , after which e is removed from ϕ_d . Consequently, each $e \in E_{G_d}$ is scrutinized n times. The procedures ProcessEdge1 and ProcessEdge2 each take $O(1)$ time. Therefore, the TC of Algorithm 2 is $O(\min k_{cmax}, k_{fmax} \cdot |E_{G_d}|^{1.5} + n \cdot |E_{G_d}|)$. Additionally, since each $e \in E_{G_d}$ can be part of n s-nodes, the SC of Algorithm 2 is $n \cdot |E_{G_d}|$.

Given a summarized graph G_s , we construct the DEBI in Algorithm 3. For each s-edge $e(\mu, \nu) \in G_s$, we allocate a weight $w(\langle \mu, \nu \rangle) = (\min\{k_c(T(\mu)), k_c(T(\nu))\}, \min\{k_f(T(\mu)), k_f(T(\nu))\})$, and W is a list to record all weight values (Lines 2-4). The index is a tree structure, initialized as the s-nodes set V_s (Lines 5). For any $w \in W$, if others do not dominate w , we collect all s-edges in E_s with weight equal to w into S_w (Lines 6-7). For each $\langle \mu, \nu \rangle \in S_w$, we add $\langle \mu, \nu \rangle$ with a weight $w(\langle \mu, \nu \rangle)$ into G'_s , if μ, ν are not connected in G'_s , (Lines 8-10).

Complexity Analysis. In Algorithm 3. Each s-edge $\langle \mu, \nu \rangle$ is examined only once throughout the DEBI build process. So the TC of Algorithm 3 is $O(|E_s|)$. Furthermore, the SC of Algorithm 4 is $n \cdot |E_{G_d}|$.

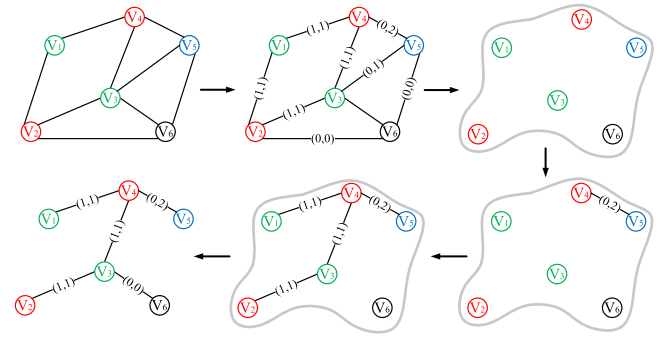


Fig. 5. Index Construction. First, allocate a weight for each s-edge; second, initialize the index as the s-nodes set; finally, construct the index by building a maximum-spanning tree.

Example 5: Fig. 5 shows the index construction process based on the summarized graph G_s in Fig. 4(c). First, we allocate a weight w to each s-edge in G_s . G'_s is initialized to be V_s . Second, we add the weighted s-edges to G'_s in turn. When the edges $\langle \nu_4, \nu_1 \rangle$, $\langle \nu_3, \nu_2 \rangle$ and $\langle \nu_4, \nu_3 \rangle$ are added into G'_s , the edge $\langle \nu_1, \nu_2 \rangle$ will not be added into G'_s , as $\langle \nu_1, \nu_2 \rangle$ are already connected in G'_s .

B. Index Based CS

After DEBI is developed from G_d , CS can be directly performed on DEBI without unnecessary access to G_d . The DEBI-based CS is detailed in Algorithm 4. The algorithm's input is the DEBI of di-graph G_d , a query node q , and two integers k_c and k_f . First, we find the s-nodes to which the query vertex q belongs from the index G'_s . We use a list $L = \nu_1, \dots, \nu_l$ to preserve this information. Meanwhile, we preserve an auxiliary data structure: visited, which indicates whether the s-node e has been processed and is initialized to False (Lines 1 - 2). And then, for each s-node $\nu \in L$ with $T(\nu) \succeq (k_c, k_f)$, we traverse G'_s in a BFS fashion (Lines 4 - 15). For each incident s-node μ , if μ has not been processed and $T(\mu) \succeq (k_c, k_f)$, the algorithm includes the edges in μ into the D-truss community C_l (Line 11).

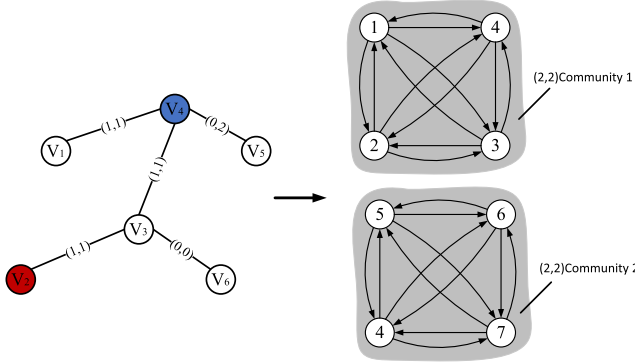


Fig. 6. Index based CS. Given two integers $k_c=2$ and $k_f=2$, and the query node v_4 . The algorithm could find two (2, 2)-communities.

Algorithm 4: DEBI-Based CS.

Input: $G'_s=(V'_s, E'_s)$, a query node q , integers k_c and k_f

Output: all D-truss communities containing q

```

1: for each  $\nu \in V'_s$  do
2:    $\nu.visited \leftarrow \text{False}$ ;
3:    $l \leftarrow 0$ ;
4:   for each  $\nu \in H(q)$  do
5:     if  $\nu.visited = \text{False}$  and  $T(\nu) \succeq (k_c, k_f)$  then
6:        $l \leftarrow l + 1, C_l \leftarrow \emptyset$ ;
7:        $\nu.visited = \text{True}$ ;
8:        $Q \leftarrow \emptyset, Q.append(\nu)$ ;
9:       while  $|Q| \neq 0$  do
10:         $\nu \leftarrow Q.pop()$ ;
11:         $C_l \leftarrow C_l \cup \{e | e \in \nu\}$ ;
12:        for each  $\mu \in N(\nu)$  do
13:          if  $w(\langle \mu, \nu \rangle) \succeq (k_c, k_f)$  and  $\mu.visited = \text{False}$ 
then
14:             $\mu.visited \leftarrow \text{True}$ ;
15:             $Q.append(\mu)$ ;
16:   Return  $\{C_1, \dots, C_l\}$ 

```

Subsequently, a series of D-truss communities which q belongs is obtained.

Complexity Analysis: In Algorithm 4, each edge in C_i is examined one time during the decomposition process. Consequently, the TC of Algorithm 4 is $O(|\cup_{i=1}^l C_i|)$.

Example 6: For the di-graph G_d in Fig. 4(a), given the query node v_4 , and two integers $k_c = 2$ and $k_f = 2$. As shown in Fig. 6, the algorithm first finds the s-nodes from DEBI (in Fig. 5) where v_4 is located, which are ν_2 and ν_4 . Starting from ν_2 , $T(\nu_2) = (2, 2)$, and the algorithm includes all edges in ν_2 into C_1 . However, ν_2 's neighboring s-node ν_3 is disqualified because $T(\nu_3) = (1, 1)$. Then, the algorithm starts from the second s-node ν_4 . As $T(\nu_4) = (2, 2)$, the algorithm includes all edges in ν_2 into C_2 , ν_4 's neighboring s-nodes ν_1 and ν_5 are disused because $T(\nu_1) = T(\nu_3) = (1, 1)$, $T(\nu_5) = (0, 3)$. Finally, we will get two communities, C_1 and C_2 .

Algorithm 5: Index Update.

Input: The affected edge set $E_{G_d}^*$, The affected super node set V_s^*

Output: The updated G_s^* in DEBI

```

1:  $T(e) \leftarrow \text{D-truss Decomposition}(E_{G_d}^*)$ 
2: if Insert a edge then
3:   for each  $e \in E_{G_d}^*$  do
4:     if  $\tau \in T(e)$  not dominate by T in  $ST(e)$  then
5:        $ST^*(e) \leftarrow ST(e) \cup \tau$ ;
6: if Delete a edge then
7:    $ST^*(e) \leftarrow ST(e)$ ;
8:  $G_s^* \leftarrow \emptyset$ ;
9: for each  $e \in E_{G_d}^*$  do
10:  if any T in  $ST^*(e)$  but not in  $ST(e)$  and  $G_s^*$  then
11:    create a s-node  $\nu, \nu \leftarrow \nu \cup e$ ;
12:    delete  $e$  from the s-node in  $V_s^*$ ;
13:     $G_s^* \leftarrow G_s^* \cup T: \nu$ ;
14:  if any T in  $ST^*(e)$  and  $G_s^*$  then
15:    delete  $e$  from the s-node in  $V_s^*$ ;
16:     $\nu$  of T:  $\nu \leftarrow \nu \cup e$ ;
17: Return  $\{G_s^*\}$ 

```

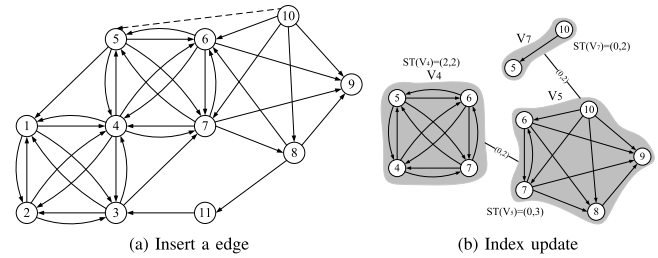


Fig. 7. An Example of index update when inserting an edge. (a) Insert a edge $\langle 10, 5 \rangle$; (b) Index update.

C. Dynamic Maintenance of DEBI

This section will discuss the index maintenance strategy when the original graph changes and propose the corresponding index update algorithm, as the real graph is not static. We consider two cases, i.e., inserting or deleting edges in the original graph.

D-truss Decomposition. When inserting/deleting an edge $e^* = \langle u, v \rangle$ in G_d , a series of triangles $\{\Delta_{w,v,u}^F : w \in N(u) \cap N(v)\}$ are created/destroyed. Therefore, we only need to calculate the STes of edges in the induced subgraph G_d^{*} containing the nodes $\{u, v, w\}$. We employ $ST(e)$ and $ST^*(e)$ to represent the ST values of edge $e \in G_d$ before and after the insertion or deletion of an edge, respectively.

Index Update. After updating the ST information of the original graph, we first find the affected supernodes. Then, separate the changed edges of the skyline truss from the original super nodes, form new super nodes, and generate corresponding super edges (the inserted edge should also be included in the new super nodes when inserting a new edge). Finally, we merge the newly formed super node and other super nodes (k_c, k_f)-triangle-connected to it.

TABLE II
DATASETS

Datasets	$ V_G $	$ E_G $	d_{max}	k_{cmax}	k_{fmax}
Networks with ground-truth communities					
Email	1.0K	25.6K	544	14	21
Social networks					
ETA	23.1K	685K	1106	3	8
Slashdot	77.4K	905.5K	5048	33	33
Twitter	81.3K	1.8M	3758	161	199
Pokec	1.6M	30.6M	20,518	18	27

Example 7: For the di-graph G_d in Fig. 7(a). Inserting a new edge $\langle 10, 5 \rangle$ into G_d , we can get the induced subgraph composed of nodes 10 and 5 and their common neighbor nodes as $G_d^* = \{\langle 5, 6 \rangle, \langle 5, 7 \rangle, \langle 6, 5 \rangle, \langle 6, 7 \rangle, \langle 7, 5 \rangle, \langle 7, 6 \rangle, \langle 10, 6 \rangle, \langle 10, 7 \rangle, \langle 10, 5 \rangle\}$. First, we calculate the trussnesses of each edge in G_d^* and the $ST(\langle 10, 5 \rangle) = (0, 2)$, however, the new trussnesses of edges $\langle 5, 6 \rangle, \langle 5, 7 \rangle, \langle 6, 5 \rangle, \langle 6, 7 \rangle, \langle 7, 5 \rangle, \langle 7, 6 \rangle, \langle 10, 6 \rangle$ and $\langle 10, 7 \rangle$ are dominated by existing ST, thus the ST of these edges are not updated. Then, we find the affected super nodes V_4 and V_5 . Since the skylines of the edges in V_4 and V_5 have not been updated, a super node consisting of edges $\langle 10, 5 \rangle$ will be generated. Finally, new super edges are generated based on the connectivity relationships between super nodes, and the final updated index is generated through the index simplification method.

In Algorithm 5, we show the critical steps of updating the index of the original graph after inserting or deleting edges. First, we use the Algorithm 1 to calculate the trussnesses of edges in induced subgraph G_d^* composed of the affected edge set $E_{G_d}^*$ (Line 1). Then, we update the ST of the edge in G_d accordingly according to the operation of inserting or deleting the edge (Lines 2-7). Finally, separate the changed edges of the skyline truss from the original super nodes and form new super nodes (Lines 8-17).

Complexity Analysis: In Algorithm 5. The D-truss decomposition takes $O(\min\{k_{cmax}, k_{fmax}\} \cdot |E_{G_d}^*|^{1.5})$ time. During the index update operation, each edge in $E_{G_d}^*$ is accessed only once. Thus, the TC of Algorithm 5 is $O(\min\{k_{cmax}, k_{fmax}\} \cdot |E_{G_d}^*|^{1.5}) + n \cdot |E_{G_d}^*|$.

VI. EXPERIMENT

This section presents our experimental studies on CS in real-world networks. We executed general experiments to verify the performance and quality of our method. All experiments were performed on a Windows Server with a 2.50 GHz six-core CPU and 32 GB memory. The algorithms were implemented in Python.

A. Datasets

In our experiments, We employed four real-world datasets comprising directed networks. The statistical details for these networks are presented in Table II, including a network Email with ground-truth communities and four social networks, EAT, Slashdot, Twitter, and Pokec. The EAT network is obtained from

TABLE III
THE SIZE OF THE ORIGINAL GRAPH, THE TIME AND SPACE REQUIRED FOR THE DEBI CONSTRUCT, AND THE MEMORY SIZE OF THE PROGRAM

Graph	Size	Index size	Time	ECR	$ V_s' $	MS
Email	0.19	0.16	1.2	0.0563	1.44K	32
ETA	6.92	5.24	472.3	0.0012	0.82K	124
Slashdot	10.49	8.75	1034.7	0.0022	1.99K	198
Twitter	43.5	35.7	3247.5	0.0142	25.5K	233
Pokec	404.3	323.6	42793.7	0.0019	58.14K	2.7K

Pajek, while all other networks are retrieved from the Stanford Network Analysis Project.

Email represents a communication network consists of 42 ground-truth communities. EAT is a collection of word association norms that provides information on the frequency of word associations based on data collected from subjects. Slashdot is a renowned technology news website. The network comprises connections representing friend and foe relationships among the users of Slashdot. Twitter contains the association relationship between Twitter users. Pokec is an online social network that connects over 1.6 million people in Slovakia.

B. Index Construction Evaluation

We commence our experiments by DEBI construction, a process conducted offline before CS. Once constructed, these indexes are stored in the main memory, facilitating efficient CS in large di-graphs. We focus on five key evaluation metrics:

- 1) The time needed for constructing the index (s);
- 2) The size of the index (MB);
- 3) The edge compression ratio (ECR): $|E_s'| / |E_{G_d}|$;
- 4) Number of super nodes $|V_s'|$ in DEBI;
- 5) The running memory size (MS) of the program (MB).

From Table III, we can find that DEBI can be built in a short time. For larger graphs such as Pokec, the D-truss-equivalence-based index can still be built in half a day, and the size of DEBI is consistently smaller than the original graph sizes, because the edges of the original graph are all compressed in the corresponding s-nodes, there is no redundant information in DEBI. In addition, on all data sets, the program's running memory is kept within a small range. Consequently, DEBI can be built efficiently with little space costs in large graphs.

C. Index Maintenance

In this part of the experiments, we test the performance of index maintenance when inserting or deleting edges in the original graph. For each graph, we randomly insert or delete $m = 0.002|E_{G_d}|$ edges and update the index. The experiments were conducted 20 times, and the final result was determined by taking the average time. The time reported in Fig. 8 is the average time of these m edges updates.

As shown in Fig. 8, the index update time is much shorter than the index construction time. This is because when we insert or delete an edge in the original graph, the affected edge is limited to a relatively small induced subgraph, and we just need to update the ST of the edges in the induced subgraph, and then reconstruct the affected super nodes.

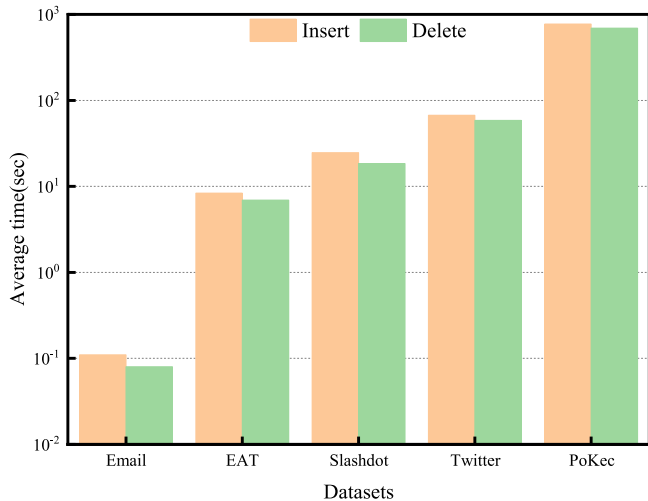


Fig. 8. The performance of dynamic maintenance of DEBI.

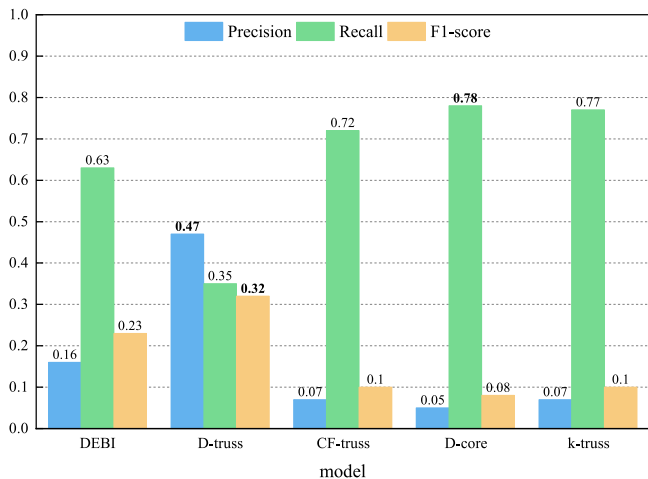


Fig. 9. Comparison of the quality of communities retrieved by different CS methods on Email.

D. Quality Evaluation

Exp-1. Quality Comparative analyses on Email: In this part of the experiment, we use a dataset Email with ground-truth communities to assess the quality of various CS methods. For each method, we run and obtain the average results of 200 CS examples, selected from 100 nodes whose degree ranks are in the top 30 percentile and bottom 70 percentile, respectively.

We contrast our method with four other CS methods, e.g., D-truss, k -truss, CF-truss and D-core. For the D-truss, we run $iLocal$ and $iGlobal$ and take the average result; for CF-truss, we run the cycle and flow truss and take the average result. We employ three metrics: Precision (P), Recall (R), and F1-score to assess the quality of the identified communities. F1-score is defined as follows:

$$F1 - Score = \frac{2 * P * R}{P + R} \quad (1)$$

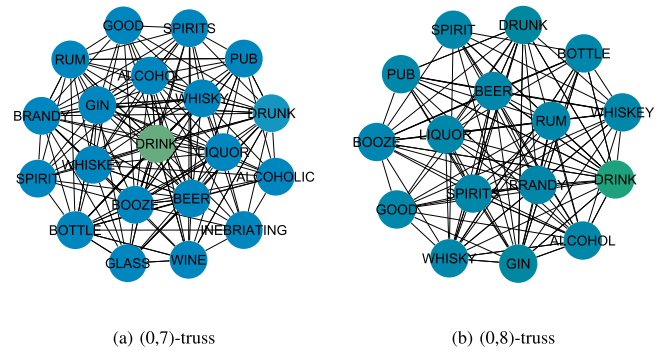


Fig. 10. Case study on EAT. (a) A (0, 7)-truss community of “DRINK” in EAT. (b) A (0, 8)-truss community of “DRINK” in EAT.

Fig. 9 reports the results of the community quality assessment on Email. Our method significantly outperforms most competitive models regarding precision and F1-score. Specifically, The precision of our method is 2.29, 3.2, and 2.29 times that of the CF-truss, D-core, and k -truss methods, respectively, and the F1-score is 2.3, 2.88, and 2.3 times that of the CF-truss, D-core and k -truss methods respectively. This means we can better guarantee that the community members returned by our method belong to the same community. Our method divides communities offline by establishing equivalence classes with triangular connectivity constraints, which can guarantee the quality of retrieved communities and achieve high-performance CS. As for the recall rate, the recall of DEBI (ours) is 0.63, which is only 0.15 lower than the best baseline method D-core because models like CF-truss and D-core retrieve a vast community, which can encompass most of the nodes within a ground-truth community. This also means that most of the members returned by these methods do not belong to the target community we want to retrieve. If these results are used in downstream tasks, it will cause a huge waste of resources. In the DCS problem definition, we need to discover the D-truss with the minimum diameter as the answer, which causes our method to be unable to obtain a high recall rate. It is worth noting that recall is not the only indicator of community quality. Judging from comprehensive indicators such as F1-score, precision, and recall, our method is better than most baseline methods regarding community quality.

Exp-2. Case analysis in EAT: We will conduct case analyses on the EAT dataset in this experiment segment. EAT represents a word association network, where the vertices correspond to English words. The directed connection from vertex a to b represents the relevancy of a and b vertices. When a word a is given, we will think of another word b .

We conduct CS on EAT using two query instances: $Q1 = \{Q = \text{“DRINK”}, k_c = 0, k_f = 7\}$, $Q = \{Q = \text{“DRINK”}, k_c = 0, k_f = 8\}$. The (0, 7)-truss community and (0, 8)-truss community, which contain “Drink,” are shown in Fig. 10(a) and (b), respectively. We can find that when k_f is set to 8, some members, such as “INEBRIATING” and “WINE,” were removed from the community. It means these members are not as closely tied to “DRINK” as those in (0, 8)-truss

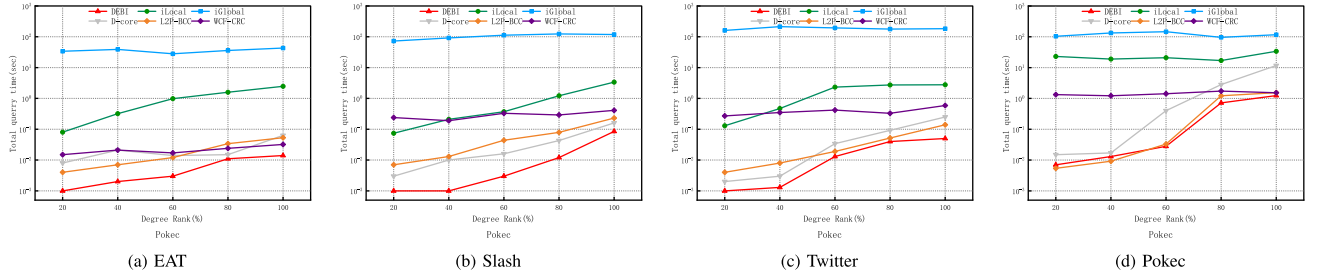
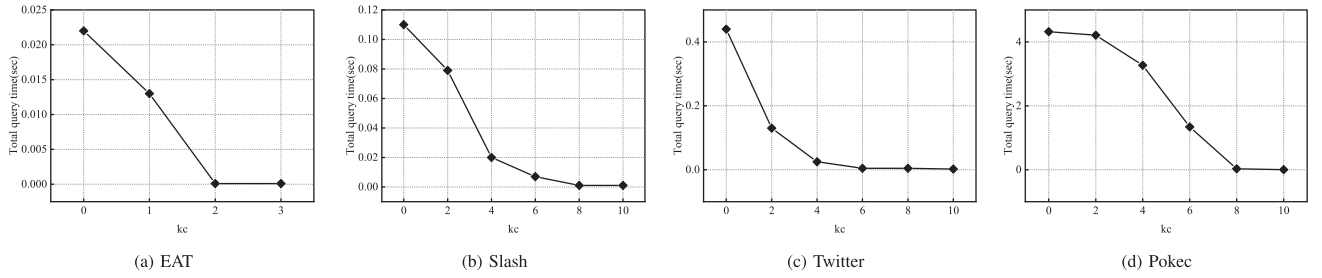
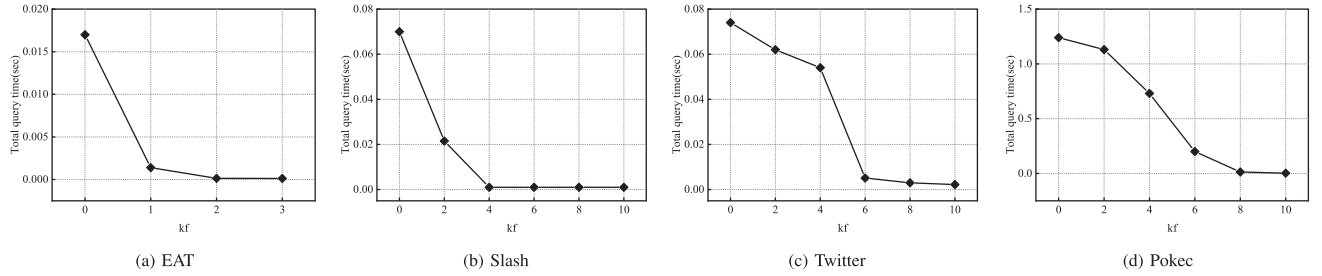


Fig. 11. CS performance under various node-degree.

Fig. 12. CS performance under various values of k_c .Fig. 13. CS performance under various values of k_f .

community. Consequently, by adjusting the parameters k_c and k_f , we can obtain a range of communities with varying densities and relevance to the query node.

E. Performance Evaluation

Exp-3. Changing the Degree of Query Vertices: In networks, nodes of different degrees often participate in communities characterized by different levels of cohesion. In this segment of the experiments, we evaluated the performance of various methods, e.g., L^2P-BCC [8], $iGlobal$, $iLocal$, $WCF-CRC$ [29] and $D-core$ on four datasets, e.g., Pokec, Twitter, Slash, and EAT networks, by adjusting the degree of nodes. First, we sort the vertices for each graph according to decreasing degrees and divide them into five equal groups. For example, the first group contains the top 20 percent of nodes by degree. Afterward, We randomly picked 100 vertices from each group for CS and reported their average running time.

As shown in Fig. 11, on all four datasets, our method is 2 to 5 orders of magnitude better than the $iLocal$ and $iGlobal$

methods, which also deal with the DCS problem. At the same time, our method performs best on Slashdot, EAT, and Twitter datasets. L^2P-BCC performs best on the height node in the Pokec dataset. Moreover, it can be observed that the time needed for $WCF-CRC$, $iLocal$, $iGlobal$, and $DEBI$ to perform CS remains nearly constant when the degree of the query node is altered. This finding suggests that the degree of the query node has minimal influence on these methods. When query vertices are selected from low-degree percentile groups, the running time for the algorithm $D-core$ and L^2P-BCC gradually increases. This is because the area the returned community covers increases as the degree decreases. In smaller graphs like EAT and Slash, $DEBI$ and $iLocal$ performed well, followed by $iGlobal$. However, in a larger graph like Pokec, the performance of $iLocal$ drops significantly due to finding the largest D-truss using the D-truss index in a larger graph taking too much time. For our method, $DEBI$ could maintain excellent and stable performance on datasets of any size.

Exp-4. Varying k_c and k_f : We investigate the time for CS query across distinct datasets by altering the parameters k_c or

k_f in this experiment. The objective is to assess how different values of k_c or k_f influence the query time of our method.

Figs. 12(a) and 13(a) show the outcomes of varying k_c and k_f from 0 to 3 on EAT. Figs. 12(b), (c), (d) and 13(b), (c), (d) showcase the effects of adjusting k_c and k_f from 0 to 10 on the Slash, Twitter and Pokec, respectively. As k_c or k_f increases, the runtime of our method decreases across all four datasets. This occurrence arises as the augmentation of k_c or k_f decreases the number of nodes and edges encompassed by the identified communities. The TC of our algorithm is predominantly influenced by the size of the (k_c, k_f) -truss communities.

VII. CONCLUSION

This paper explores truss-based CS in large di-graphs. First, we introduce a novel equivalence relation known as D-truss equivalence. We then construct a summarized graph of a di-graph G_d using D-truss equivalence, thereby preserving the D-truss information inherent in G_d . Afterward, we simplify the summarized graph by the maximum spanning tree idea to develop the D-truss-equivalence-based index, DEBI. CS can be productively run directly on DEBI without requiring access to the original graph. We conducted comprehensive experiments on large di-graphs that conclusively demonstrated that the method in this paper outperforms the existing techniques significantly in performance. In future research, we plan to optimize community constraints further to obtain higher recall and precision while ensuring efficient community queries.

ACKNOWLEDGMENT

The authors deepest gratitude goes to the anonymous reviewers and AE for their careful work and thoughtful suggestions that have helped improve this article substantially.

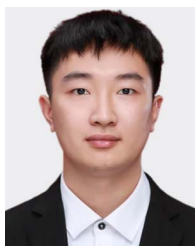
REFERENCES

- [1] E. Akbas and P. Zhao, "Truss-based community search: A truss-equivalence based indexing approach," in *Proc. VLDB Endowment*, vol. 10, no. 11, pp. 1298–1309, 2017.
- [2] N. Barbieri, F. Bonchi, E. Galimberti, and F. Gullo, "Efficient and effective community search," *Data Mining Knowl. Discov.*, vol. 29, pp. 1406–1433, 2015.
- [3] C. Chen, K. Li, Y. Li, and X. Zou, "ReGNN: A redundancy-eliminated graph neural networks accelerator," in *Proc. IEEE Int. Symp. High-Perform. Comput. Architecture*, 2022, pp. 429–443.
- [4] C. Chen, K. Li, X. Zou, and Y. Li, "DyGNN: Algorithm and architecture support of dynamic pruning for graph neural networks," in *Proc. 58th ACM/IEEE Des. Automat. Conf.*, 2021, pp. 1201–1206.
- [5] L. Chen, C. Liu, R. Zhou, J. Li, X. Yang, and B. Wang, "Maximum co-located community search in large scale social networks," in *Proc. VLDB Endowment*, vol. 11, no. 10, pp. 1233–1246, 2018.
- [6] W. Cui, Y. Xiao, H. Wang, Y. Lu, and W. Wang, "Online search of overlapping communities," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 277–288.
- [7] W. Cui, Y. Xiao, H. Wang, and W. Wang, "Local search of communities in large graphs," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 991–1002.
- [8] Z. Dong, X. Huang, G. Yuan, H. Zhu, and H. Xiong, "Butterfly-core community search over labeled graphs," 2021, *arXiv:2105.08628*.
- [9] Y. Fang et al., "A survey of community search over big graphs," *VLDB J.*, vol. 29, pp. 353–392, 2020.
- [10] Y. Fang, Z. Wang, R. Cheng, H. Wang, and J. Hu, "Effective and efficient community search over large directed graphs," *IEEE Trans. Knowl. Data Eng.*, vol. 31, no. 11, pp. 2093–2107, Nov. 2019.

- [11] S. Fortunato, "Community detection in graphs," *Phys. Rep.*, vol. 486, no. 3–5, pp. 75–174, 2010.
- [12] C. Giatsidis, D. M. Thilikos, and M. Vazirgiannis, "D-cores: Measuring collaboration of directed graphs based on degeneracy," *Knowl. Inf. Syst.*, vol. 35, pp. 311–343, 2013.
- [13] D. Hric, R. K. Darst, and S. Fortunato, "Community detection in networks: Structural communities versus ground truth," *Phys. Rev. E*, vol. 90, no. 6, 2014, Art. no. 062805.
- [14] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu, "Querying k-truss community in large and dynamic graphs," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 1311–1322.
- [15] X. Huang and L. V. S. Lakshmanan, "Attribute-driven community search," in *Proc. VLDB Endowment*, vol. 10, no. 9, pp. 949–960, 2017.
- [16] J. Leskovec, K. J. Lang, and M. Mahoney, "Empirical comparison of algorithms for network community detection," in *Proc. 19th Int. Conf. World Wide Web*, 2010, pp. 631–640.
- [17] J. Li, X. Wang, K. Deng, X. Yang, T. Sellis, and J. X. Yu, "Most influential community search over large social networks," in *Proc. IEEE 33rd Int. Conf. Data Eng.*, 2017, pp. 871–882.
- [18] R.-H. Li et al., "Skyline community search in multi-valued networks," in *Proc. Int. Conf. Manage. Data*, 2018, pp. 457–472.
- [19] R.-H. Li, L. Qin, J. X. Yu, and R. Mao, "Influential community search in large networks," in *Proc. VLDB Endowment*, vol. 8, no. 5, pp. 509–520, 2015.
- [20] R.-H. Li, J. Su, L. Qin, J. X. Yu, and Q. Dai, "Persistent community search in temporal networks," in *Proc. IEEE 34th Int. Conf. Data Eng.*, 2018, pp. 797–808.
- [21] F. Liu et al., "Deep learning for community detection: Progress, challenges and opportunities," 2020, *arXiv:2005.08225*.
- [22] Q. Liu, M. Zhao, X. Huang, J. Xu, and Y. Gao, "Truss-based community search over large directed graphs," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 2183–2197.
- [23] Y. Liu, T. Safavi, A. Dighe, and D. Koutra, "Graph summarization methods and applications: A survey," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 1–34, 2018.
- [24] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Phys. Rev. E*, vol. 69, no. 2, 2004, Art. no. 026113.
- [25] S. Parthasarathy, Y. Ruan, and V. Satuluri, "Community discovery in social networks: Applications, methods and emerging trends," in *Social Network Data Analytics*. Berlin, Germany: Springer, 2011, pp. 79–113.
- [26] A. E. Sariyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek, "Incremental k-core decomposition: Algorithms and evaluation," *VLDB J.*, vol. 25, pp. 425–447, 2016.
- [27] M. Sozio and A. Gionis, "The community-search problem and how to plan a successful cocktail party," in *Proc. 16th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2010, pp. 939–948.
- [28] T. Takaguchi and Y. Yoshida, "Cycle and flow trusses in directed networks," *Roy. Soc. Open Sci.*, vol. 3, no. 11, 2016, Art. no. 160270.
- [29] Y. Tang, J. Li, N. Al H. Haldar, Z. Guan, J. Xu, and C. Liu, "Reliable community search in dynamic networks," 2022, *arXiv:2202.01525*.
- [30] C. Tsourakakis, "The k-clique densest subgraph problem," in *Proc. 24th Int. Conf. World Wide Web*, 2015, pp. 1122–1132.
- [31] J. Xie, S. Kelley, and B. K. Szymanski, "Overlapping community detection in networks: The state-of-the-art and comparative study," *ACM Comput. Surv.*, vol. 45, no. 4, pp. 1–35, 2013.
- [32] J. Zhou et al., "Graph neural networks: A review of methods and applications," *AI Open*, vol. 1, pp. 57–81, 2020.
- [33] L. Zhu, M. Ghasemi-Gol, P. Szekely, A. Galstyan, and C. A. Knoblock, "Unsupervised entity resolution on multi-type graphs," in *Proc. 15th Int. Semantic Web Conf.*, Springer, 2016, pp. 649–667.



Wei Ai received the PhD degree from the College of Computer Science and Electronic Engineering, Hunan University, Changsha, China. She is currently an assistant professor with the Central South University of Forest and Technology, China. Her research interests include data mining, Big Data, cloud computing, and parallel computing.



Canhao Xie is currently working toward the graduate degree with the College of Computer Information and Engineering, Central South University of Forestry and Technology, Changsha, China. His research interests include complex network analysis and community query optimization.



Tao Meng received the PhD degree from the College of Computer Science and Electronic Engineering, Hunan University, Changsha, China. He is currently an assistant professor with the Central South University of Forest and Technology, China. His research interests include data mining, network analysis, and deep learning.



Jayi Du received the BSc, MSc, and PhD degrees in computer science from Hunan University, China, in 2004, 2010, and 2015, respectively. He is currently an assistant professor with the Central South University of Forest and Technology, China. His research interests include modeling and scheduling for parallel and distributed computing systems, embedded system computing, cloud computing, parallel system reliability, and parallel algorithms.



Keqin Li (Fellow, IEEE) received the BS degree in computer science from Tsinghua University, in 1985, and the PhD degree in computer science from the University of Houston, in 1990. He is a SUNY distinguished professor with the State University of New York and a National distinguished professor with Hunan University (China). He has authored or co-authored more than 990 journal articles, book chapters, and refereed conference papers. He received several best paper awards from international conferences including PDPTA-1996, NAECON-1997, IPDPS-2000, ISPA-2016, NPC-2019, ISPA-2019, and CPSCCom-2022. He holds nearly 75 patents announced or authorized by the Chinese National Intellectual Property Administration. He is among the world's top five most influential scientists in parallel and distributed computing in terms of single-year and career-long impacts based on a composite indicator of the Scopus citation database. He was a 2017 recipient of the Albert Nelson Marquis Lifetime Achievement Award for being listed in Marquis Who's Who in Science and Engineering, Who's Who in America, Who's Who in the World, and Who's Who in American Education for over 20 consecutive years. He received the Distinguished Alumnus Award from the Computer Science Department at the University of Houston in 2018. He received the IEEE TCCLD Research Impact Award from the IEEE CS Technical Committee on Cloud Computing in 2022 and the IEEE TCSVC Research Innovation Award from the IEEE CS Technical Community on Services Computing in 2023. He won the IEEE Region 1 Technological Innovation Award (Academic) in 2023. He is a member of the SUNY Distinguished Academy. He is an AAAS fellow, an AAlA fellow, and an ACIS founding fellow. He is an academician member of the International Artificial Intelligence Industry Alliance. He is a member of Academia Europaea (Academician of the Academy of Europe).