Truss-based Community Search over Large Directed Graphs

Qing Liu
Hong Kong Baptist University
qingliu@comp.hkbu.edu.hk

Minjun Zhao
Zhejiang University
minjunzhao@zju.edu.cn

Xin Huang
Hong Kong Baptist University
xinhuang@comp.hkbu.edu.hk

Jianliang Xu
Hong Kong Baptist University
xujl@comp.hkbu.edu.hk

Yunjun Gao
Zhejiang University
gaoyj@zju.edu.cn

ABSTRACT

Community search enables personalized community discovery and has wide applications in large real-world graphs. While community search has been extensively studied for undirected graphs, the problem for directed graphs has received attention only recently. However, existing studies suffer from several drawbacks, e.g., the vertices with varied in-degrees and out-degrees cannot be included in a community at the same time. To address the limitations, in this paper, we systematically study the problem of community search over large directed graphs. We start by presenting a novel community model, called D-truss, based on two distinct types of directed triangles, i.e., flow triangle and cycle triangle. The D-truss model brings nice structural and computational properties and has many advantages in comparison with the existing models. With this new model, we then formulate the D-truss community search problem, which is proved to be NP-hard. In view of its hardness, we propose two efficient 2-approximation algorithms, named *Global* and *Local*, that run in polynomial time yet with quality guarantee. To further improve the efficiency of the algorithms, we devise an indexing method based on D-truss decomposition. Consequently, the D-truss community search can be solved upon the D-truss index without time-consuming accesses to the original graph. Experimental studies on real-world graphs with ground-truth communities validate the quality of the solutions we obtain and the efficiency of the proposed algorithms.

SIGMOD '20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3380587

KEYWORDS

Community search; Directed graph; D-truss

ACM Reference Format:
Qing Liu, Minjun Zhao, Xin Huang, Jianliang Xu, and Yunjun Gao. 2020. Truss-based Community Search over Large Directed Graphs. In *2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3318464.3380587

1 INTRODUCTION

Community search is an important tool for network analysis, which aims to find the densely connected subgraphs containing the query vertices [28]. To date, a lot of research efforts have been devoted to the study of community search over large graphs, of which the majority focus on undirected graphs [11, 15, 17]. However, in reality, directed graphs are ubiquitous, such as social network, Web network, gene regulatory network, and so on [2]. Discovering communities in large directed graphs is of great significance. For example, the users of *Twitter* can be modelled as a directed social network graph, where each vertex represents a user and an edge from vertex i to vertex j means user i follows user j. In this directed graph, the users following a common set of accounts may constitute a community.

To find cohesive communities, many models have been proposed for undirected graphs, e.g., k-core, k-truss, and clique. Among them, the k-truss model has received considerable attention due to its strong structural cohesiveness and high computational efficiency [1, 6, 14]. Specifically, k-truss is defined based on undirected triangles. By definition, every edge in a k-truss is contained in at least k triangles.[1] However, k-truss is unsuitable for directed graphs. This is because undirected graphs have only one type of triangles while directed graphs could contain two types of triangles, i.e., cycle triangle and flow triangle [29]. A cycle (flow) triangle is a cyclic (acyclic) directed graph of three vertices.

[1] In some studies, k-truss is defined that every edge is contained in at least $k-2$ triangles.

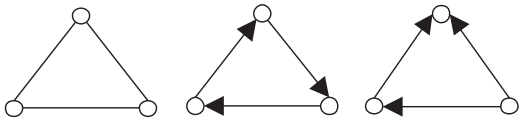


Figure 1: Triangles in Graphs

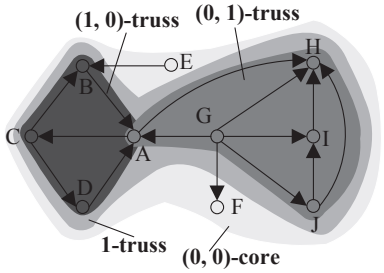


Figure 2: A Directed Graph

Figure 1 shows the three different types of triangles. The cycle triangle and flow triangle can be used to construct different kinds of communities in directed graphs. For example, suppose Figure 2 represents a social network graph. The vertices A, B, C, and D form a community, in which every edge is contained in a cycle triangle. On the other hand, for the community consisting of A, G, H, I, and J, every edge is associated with a flow triangle. Obviously, these two communities exhibit distinct features: the former is a group of strongly connected users, whereas the latter is a group of weakly connected users following a common user (vertex H). However, if we simply employ the k-truss model for this directed graph, it would take these two communities as a single truss community and cannot distinguish them.

Recently, Fang et al. [12] have explored community search over large directed graphs based on a D-core (also called (k, l)-core) model [13]. In particular, given a directed graph G, a query vertex q, two positive integers k and l, it returns a connected subgraph $G' \subseteq G$ such that G' contains q and for every vertex in G', its in-degree and out-degree are no less than k and l, respectively. However, the D-core model may suffer from one significant limitation. For directed graphs, there may exist some special vertices, namely those with high in-degree and low out-degree (e.g., authorities and key opinion leaders) and those with low in-degree and high out-degree (e.g., hubs and followers) [20]. Unfortunately, the D-core model usually returns the communities with only one type of such vertices. If we try to include both types of such vertices, both k and l should be set to be low, which may result in a sparse community. For example, in Figure 2, for the community {A, G, H, I, J}, G's in-degree and H's out-degree are both 0. If we want to retrieve G and H at the same time, we should set $k = l = 0$; in this case, however, the entire graph would be returned as a (0, 0)-core community.

To address the aforementioned limitations of the existing studies, in this paper, we systematically study the problem of community search over large directed graphs. First of all, we present a new community model, called D-truss (also called (k_c, k_f)-truss), for directed graphs. Specifically, D-truss takes cycle triangle and flow triangle as its cornerstone. In a D-truss, every edge should form cycle triangles (flow triangles) with at least k_c (k_f) other vertices. D-truss can overcome the drawbacks of k-truss and D-core, and brings nice structural and computational properties. Back to the example shown in Figure 2, if we want to search (1, 0)-truss with node A as the query vertex, the community {A, B, C, D} will be retrieved; otherwise, if we want to search (0, 1)-truss with node A as the query vertex, the community {A, G, H, I, J} can be found.

Based on the D-truss model, we formally define our problem of D-truss community search (DCS). Specifically, DCS aims to find the subgraph: (1) containing the query vertices; (2) being a D-truss; and (3) having the minimum diameter. It is worth mentioning that we additionally impose the diameter constraint in our problem definition to avoid the so-called "free-rider effect" [31]. Intuitively, community search is user-centered and personalized and, thus, the returned community should be relevant to the given query. However, in some cases, the vertices that are far away from the query vertices and irrelevant to them may be included in the resultant community. Therefore, we use the minimum diameter to eliminate those vertices.

We prove that the DCS problem is NP-hard. Being an NP-hard problem, the D-truss community search is highly challenging. To efficiently solve the problem, we design two polynomial 2-approximation algorithms, i.e., *Global* and *Local*. Note that the approximation is with respect to the minimum diameter. In particular, *Global* works in a top-down manner while *Local* runs in a bottom-up fashion. Moreover, we propose an indexing method to further improve the efficiency of the algorithms. More specifically, we first employ D-truss decomposition to compute all possible D-trusses in the entire graph. Then, we record the skyline trussness (to be defined in Section 6.1) for every edge and compact them in an index. Thus, when searching the D-truss community, the algorithms only need to traverse the index to check the dominance relationship of edges' skyline trussnesses instead of time-consuming accesses to the original graph.

Overall, we make the following contributions in this paper:

- We present a novel community model called D-truss for community search over large directed graphs. We highlight the advantages and interesting properties of D-truss in comparison with the state-of-the-art models.
- We formally define the D-truss community search problem and show that the problem is NP-hard by a reduction to the maximum clique problem.

- We develop two efficient 2-approximation algorithms, i.e., *Global* and *Local*, for our problem. To further improve the efficiency of the algorithms, we propose an indexing method based on D-truss decomposition.
- We conduct extensive experimental studies on real-world graphs with ground-truth communities. Experimental results validate the quality of the solutions we obtain and the efficiency of the proposed algorithms.

Roadmap: The rest of the paper is organized as follows. Section 2 reviews the related work. Section 3 introduces the D-truss model and formally defines the problem of D-truss community search. Section 4 analyzes the hardness and properties of our problem. Section 5 proposes two approximation algorithms, followed by an index-based method proposed in Section 6. Section 7 presents our experimental results. Finally, we conclude this paper in Section 8.

2 RELATED WORK

Community search is first introduced by Sozio and Gionis [28]. Since then, numerous community models have been proposed, such as k-core, k-truss, k-clique, and so on [11, 15, 17]. In particular, k-core-based community search returns the community in which the degree of every vertex is no less than k [3, 8, 28]. It is well known that the k-core community is not guaranteed to be cohesive. To ensure the cohesiveness of the retrieved community, clique has also been considered for community search [34]. However, as the clique model is too restrictive, some relaxed variants have been investigated [7, 30]. More recently, k-truss has been extensively explored [1, 14, 18]. In addition, Wu et al. [31] propose query biased density to reduce the free-rider effect for the returned community. Besides simple graphs, community search has also been investigated for more complex graphs, such as community search over attributed graphs [9, 16], geo-social graphs [4, 10, 35], temporal graphs [25], multi-valued graphs [23], and weighted graphs [22, 24].

The above studies mainly focus on undirected graphs. There are also some recent works targeting on community analysis for directed graphs. In particular, community detection on directed graphs aims to find all the communities in a given directed graph [19, 21, 26, 27, 32, 33]. However, only few work has explored community search over directed graphs, based on core-based models [12]. To the best of our knowledge, so far no work has investigated truss-based community search over directed graphs.

In summary, (i) the techniques of community search over undirected graphs do not capture the directionality of edges and, thus, are not applicable to community search over directed graphs; (ii) the community detection over directed graphs finds all communities from a graph, which is inefficient and cannot be used for the community search as it does not take query vertices into consideration; (iii) as analyzed in Section 1, existing studies of community search on directed graphs suffer from some drawbacks. As such, it is essential to propose more effective models and develop more efficient algorithms for community search over large directed graphs.

3 PROBLEM FORMULATION

Let $G = (V_G, E_G)$ be a directed, simple, and unweighted graph with a set V of vertices and a set E of edges. We call this graph G a digraph for short. Let $n = |V_G|$ and $m = |E_G|$ be the numbers of vertices and edges, respectively. W.l.o.g we assume in this paper that $m \geq n - 1$ [18]. A subgraph $H = (V_H, E_H)$ of G satisfies $V_H \subseteq V_G$ and $E_H = \{\langle u, v\rangle \in E_G : u, v \in V_H\}$. For an edge $\langle u, v\rangle$, we say u is an in-neighbor of v and v is an out-neighbor of u. For a vertex v in H, we denote the set of in-neighbors of v by $\mathsf{N}^+_H(v) = \{u : \langle u, v\rangle \in E_H\}$ and the in-degree of v by $\deg^+_H(v) = |\mathsf{N}^+_H(v)|$. Similarly, we denote the set of out-neighbors of v by $\mathsf{N}^-_H(v) = \{u : \langle v, u\rangle \in E_H\}$, and the out-degree of v by $\deg^-_H(v) = |\mathsf{N}^-_H(v)|$. The degree of vertex v in H is defined as $\deg_H(v) = \deg^+_H(v) + \deg^-_H(v)$. Next, we provide a formal definition of D-truss and formulate the problem of D-truss community search.

3.1 D-truss

A triangle is a subgraph of three vertices connected to each other by three edges [6]. In a digraph, there exist two types of triangles: *cycle triangles* and *flow triangles* [29]. Specifically, each vertex in a cycle triangle formed by three vertices u, v, w, denoted by $\triangle^C_{uvw}$, has in-degree and out-degree of exactly *one*, respectively. A flow triangle, denoted by $\triangle^F_{uvw}$, is a connected graph composed of three vertices having out-degrees equal to *zero*, *one*, and *two*, respectively. Figures 1(b) and 1(c) give examples of a cycle triangle and a flow triangle, respectively. With the cycle and flow triangles, we formally introduce the definitions of cycle support and flow support below.

Definition 3.1. **(Cycle Support).** Given a digraph H and an edge $e = \langle u, v\rangle \in E_H$, the *cycle support* of e in H is defined as the number of vertices that can form cycle triangles with e in H, denoted by $\mathrm{csup}_H(e) = |\{w \in V_H : \triangle^C_{uvw} \text{ in } H\}|$.

Definition 3.2. **(Flow Support).** Given a digraph H and an edge $e = \langle u, v\rangle \in E_H$, the *flow support* of e in H is defined by $\mathrm{fsup}_H(e) = |\{w \in V_H : \triangle^F_{uvw} \text{ in } H\}|$.

On the basis of cycle support and flow support, we define D-truss as follows.

Definition 3.3. **(D-truss).** Given a digraph G and two integers k_c and k_f, a subgraph $H \subseteq G$ is a D-truss, also denoted as (k_c, k_f)-truss, if $\forall e \in E_H$, $\mathrm{csup}_H(e) \geq k_c$ and $\mathrm{fsup}_H(e) \geq k_f$.

A D-truss H is a maximal D-truss if there exists no D-truss $H' \subseteq G$ satisfying $H' \supset H$.

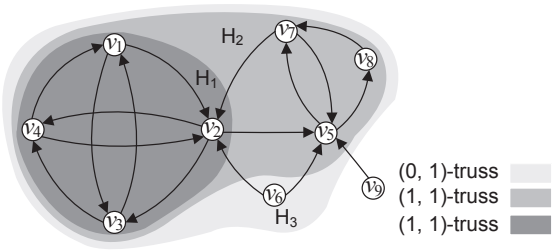


Figure 3: An example of D-truss

Example 1. *Consider an example of digraph G in Figure 3. The edge $\langle v_2, v_5\rangle$ forms one cycle triangle with vertex v_7 in $\triangle^C_{v_2v_5v_7}$ and two flow triangles with v_7 and v_6 in $\triangle^F_{v_2v_5v_7}$ and $\triangle^F_{v_2v_5v_6}$, respectively. Hence, $\mathsf{csup}_G(\langle v_2, v_5\rangle) = 1$ and $\mathsf{fsup}_G(\langle v_2, v_5\rangle) = 2$. As the edge $\langle v_9, v_5\rangle$ is not involved in any cycle triangle or flow triangle, $\mathsf{csup}_G(\langle v_9, v_5\rangle) = \mathsf{fsup}_G(\langle v_9, v_5\rangle) = 0$. There exist several different D-trusses. H_1 and H_2 are (1, 1)-trusses, and H_3 is a (0, 1)-truss. Note that both H_2 and H_3 are maximal D-trusses of G but H_1 is not.*

From the definition, D-truss takes cycle triangle and flow triangle as its cornerstone. This makes sense as both triangles are ubiquitous in digraphs [29]. Besides the social network example mentioned in the Introduction, another example is word network of the Edinburgh Associative Thesaurus. In this network, a directed link from nodes (i.e., words) i to j exists if word j comes to mind when word i is shown as a stimulus. Both the cycle triangle and flow triangle are meaningful. Specifically, the words constituting a cycle triangle have an equal relationship with each other such that each word can recall all other words. On the other hand, the words constituting a flow triangle have a hierarchical relationship such that some words can remind other words but the converse rarely occurs. For example, the words "Sick", "Disease", and "Medicine" can form a cycle triangle while the words "Sick", "Disease", and "Cancer" can constitute a flow triangle.

3.2 D-truss Community Search

Before formally defining the problem of D-truss community search, we first introduce the concepts of connectivity, distance, and diameter in digraph G.

For any two vertices u and v in G, we say that u can reach v (denoted as $u \to v$), if and only if there exists a directed path from u to v in G. If $u \to v$ and $v \to u$ both hold, we say that u and v are strongly connected; if either $u \to v$ or $v \to u$ holds, u and v are weakly connected; if neither $u \to v$ nor $v \to u$ holds, u and v are disconnected. A subgraph H of G is strongly connected if every pair of vertices in H is strongly connected in H. We say that H is weakly connected if every pair of vertices in H is at least weakly connected in H. The distance from u to v in G, denoted by $\mathrm{dist}_G\langle u, v\rangle$, is the length of the shortest directed path from u to v in G. If v is not reachable from u, we say $\mathrm{dist}_G\langle u, v\rangle = +\infty$. In the following, we give a definition of diameter in digraphs [2, 5].

Definition 3.4. **(Diameter).** The diameter of digraph H, denoted by $\mathrm{diam}(H)$, is the maximum distance among all pairs of vertices in H, i.e., $\mathrm{diam}(H) = \max_{u,v\in V_H} \mathrm{dist}_H\langle u, v\rangle$.

Note that if a directed graph is not strongly connected, its diameter is infinity. For example, in Figure 3, the diameter of G is $\mathrm{diam}(G) = +\infty$ as $\mathrm{dist}_G\langle v_5, v_9\rangle = +\infty$. On the basis of the above definitions, we formally define the problem of D-truss community search as follows.

Problem 1. **(D-Truss Community Search).** *Given a digraph $G(V_G, E_G)$, two integers k_c and k_f, and a set of query vertices $Q \subseteq V_G$, the D-truss community search (*DCS*) is to find a weakly connected subgraph $H^* \subseteq G$ satisfying:*

(1) $Q \subseteq V(H^*)$;

(2) H^* is a D-truss ;

(3) $\mathrm{diam}(H^*)$ should be the minimum among all subgraphs satisfying conditions (1) and (2).

In the definition, condition (1) requires that the community contains the query vertex set Q; condition (2) makes sure that the community should be densely connected; condition (3) requires that each vertex in the community should be as close to other vertices as possible, which excludes irrelevant vertices from the answer. All the three conditions together ensure that the returned community is a cohesive subgraph with good quality.

Example 2. *Consider the graph G in Figure 3. Assume that $k_c = 1$, $k_f = 1$, and vertex v_4 is the query vertex. Then, the answer to the DCS problem is the (1, 1)-truss H_1. Although H_2 is also a (1, 1)-truss satisfying the first two constraints of the problem, $\mathrm{diam}(H_2) = 4 > \mathrm{diam}(H_1) = 2$ and vertices v_5, v_7, and v_8 in H_2 have loose connections with v_4. Thus, H_1 is considered better than H_2.*

4 PROBLEM ANALYSIS

In this section, we first analyze the hardness of the DCS problem. Next, we discuss the properties of the D-truss community and show its advantages against the existing models.

4.1 Hardness

To show the hardness of the DCS problem, we first prove another problem, the k-diClique problem defined in Problem 2, is NP-hard. After that, we reduce the k-diClique problem to the D-truss community search problem.

Problem 2. **(k-diClique Problem).** *Given a digraph G, the k-diClique problem is to check whether there exists a k-diClique $H(V_H, E_H)$ of G such that $\forall u, v \in V_H$, $\langle u, v\rangle \in E_H$, and $|V_H| = k$.*

Theorem 4.1. *The k-diClique problem is NP-hard.*

Proof. We consider the decision version of the maximum clique problem, that is, to check whether there exists a k-clique in an undirected graph $\mathcal{G} = (V_{\mathcal{G}}, E_{\mathcal{G}})$ for a given k. We reduce this NP-hard problem to the k-diClique problem.

For the k-diClique problem, we construct a digraph $G(V_G, E_G)$ from the undirected graph $\mathcal{G}$ as follows. Let $V_G = V_{\mathcal{G}}$ and $E_G = \{\langle u, v\rangle, \langle v, u\rangle : (v, u) \in E_G\}$. Obviously, a k-clique $\mathcal{H}$ exists in $\mathcal{G}$ iff there exists a k-diClique H in G where $V_H = V_{\mathcal{H}}$. As such, the k-diClique problem is NP-hard. □

Next, we present the decision version of the D-truss community search (dDCS) problem as follows.

Problem 3. **(dDCS Problem).** *Given a digraph $G = (V_G, E_G)$, a query set Q, three integers k_c, k_f, and d, the dDCS problem aims to check whether there exists a D-truss community H with $\mathrm{diam}(H) = d$ in G.*

Theorem 4.2. *The dDCS problem is NP-hard.*

Proof. We reduce the k-diClique problem to the dDCS problem. Specifically, given an instance of the k-diClique problem defined on a digraph G with parameter k, we construct a new digraph G' for the dDCS problem as follows. First, we copy G to G'. Second, we add a set of dummy vertices V_Q ($|V_Q| \geq 1$) into G', i.e., $V_{G'} = V_G \cup V_Q$. Third, for each vertex $v \in V_Q$ and $u \in V_{G'} - \{v\}$, we add two edges $\langle u, v\rangle$ and $\langle v, u\rangle$ into G', i.e., $E_{G'} = E_G \cup \{\langle u, v\rangle, \langle v, u\rangle : v \in V_Q, u \in V_{G'} - \{v\}\}$. Based on G', we set $Q = V_Q$, $d = 1$, and $k_c = k_f = k + |Q| - 2$ for the dDCS problem. In the following, we show that the instance of the k-diClique problem is a YES-instance iff the corresponding instance of the dDCS problem is a YES-instance.

($\Rightarrow$) : Suppose a k-diClique H exists in G. The subgraph H' induced by $V_H \cup V_Q$ in G' is a strongly connected $(k + |Q| - 2, k + |Q| - 2)$-truss with a diameter of 1. Thus, H' is also a YES-instance of the dDCS problem.

($\Leftarrow$) : Suppose a D-truss H' is a YES-instance of the dDCS problem in digraph G' and H is the subgraph induced by $V_{H'} - V_Q$ in G. First, H' must contain at least $k_c + 2 = k + |Q|$ vertices and thus H must contain at least $k_c + 2 - |Q| = k$ vertices, i.e., $|V_H| \geq k$. Second, as $\mathrm{diam}(H') = 1$, it follows that H is a k'-diClique where $k' \geq k$. Thus, an existing $H'' \subseteq H$ is also a YES-instance of the k-diClique problem. □

4.2 Properties of D-truss Community

This section analyzes the properties of D-truss communities.

Property 1 (*Minimum Degree*). *For each vertex v in a D-truss H, $\deg_H(v) \geq \max\{2k_c, k_f + 1\}$ holds.*

Proof. Each edge of the D-truss H has at least k_c cycle triangles and k_f flow triangles. First, consider the cycle triangles. If a vertex is in a cycle triangle, it must be the head of an edge and the tail of another edge. When the vertex v is the head of an edge e, $\deg^+_H(v) \geq k_c$ since e is contained in k_c different cycle triangles. Similarly, we can derive $\deg^-_H(v) \geq k_c$ when the vertex v is the tail of an edge. Hence, $\deg_H(v) = \deg^+_H(v) + \deg^-_H(v) \geq 2k_c$. Second, consider the flow triangles. If an edge e is contained in k_f different flow triangles and $v \in e$, $\deg_H(v) = \deg^+_H(v) + \deg^-_H(v) \geq k_f + 1$. Therefore, $\deg_H(v) \geq \max\{2k_c, k_f + 1\}$. □

Property 2 (*Strong Connectivity*). *For a D-truss H with $k_c \geq 1$, each pair of vertices in a weakly connected component of H is strongly connected.*

Proof. A digraph G is said strongly connected if any two vertices in G are reachable from each other. Consider two vertices u, v in a weakly connected component of a D-truss. If $\langle u, v\rangle \in E$ or $\langle v, u\rangle \in E$, there must be a cycle triangle containing u and v. Therefore, u and v are reachable from each other. On the other hand, if neither $\langle u, v\rangle \in E$ nor $\langle v, u\rangle \in E$, there must exist an undirected path $\{(v_1, v_2), ..., (v_{n-1}, v_n)\}$ where $v_1 = u$ and $v_n = v$, as H is weakly connected. Since $k_c \geq 1$, for any $i \in [1, n-1]$, the vertices v_i and v_{i+1} are contained in a cycle triangle, meaning that v_i and v_{i+1} are reachable. By transitivity, u and v are also reachable. □

Property 3 (*Partially Hierarchical Structure*). *Given two different maximal D-trusses H_1: (k^1_c, k^1_f)-truss and H_2: (k^2_c, k^2_f)-truss, if $k^1_c \geq k^2_c$ and $k^1_f \geq k^2_f$, it is obvious that $H_1 \subseteq H_2$; if $k^1_c \geq k^2_c$ and $k^1_f \leq k^2_f$, or $k^1_c \leq k^2_c$ and $k^1_f \geq k^2_f$, then $H_1 \not\supseteq H_2$ and $H_1 \not\subseteq H_2$.*

4.3 Comparison with Existing Models

In this section, we compare D-truss with the existing models of community search and highlight its advantages.

Undirected k-Truss Community Model [1, 14, 18], D-core Community Model [12, 13]. As explained in the Introduction, the undirected k-truss community model and the D-core community model suffer from some limitations. Our D-truss community model can address those limitations. In particular, as D-truss is based on both of the cycle triangle and flow triangle, it can distinguish the communities with different structures. However, our D-truss-based community search needs to consider both of the edge's cycle support and flow support, which makes the algorithm design more complex and challenging.

Flow/Cycle Truss [29]. While targeting on the problem of community detection, Takaguchi and Yoshida [29] have proposed *cycle truss* and *flow truss* based on the cycle triangle and flow triangle. The differences between the flow/cycle truss and the D-truss are two-fold. First, the definition of the support is different. The support of an edge e for a flow/cycle truss is the number of triangles e is involved in, whereas e's

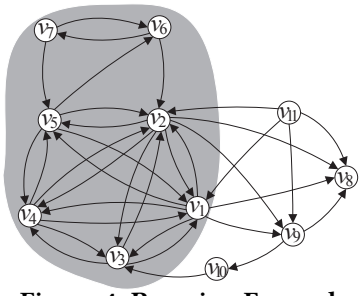


Figure 4: Running Example

support for a D-truss is the number of vertices that can form flow/cycle triangles with e. As the edge e may be involved in multiple triangles with the same vertex, counting the number of vertices can avoid redundance. Second, the flow/cycle truss assumes that a community has only flow triangles or cycle triangles. However, in reality, a community may contain both flow triangles and cycle triangles. For the above two reasons, we believe that the D-truss is more preferable than the flow/cycle truss. In addition, as will be shown by the experimental results in Section 7, the communities returned by our model achieve a higher quality than the other models.

5 ALGORITHMS FOR D-TRUSS COMMUNITY SEARCH

In view of the hardness of the DCS problem, in this section, we propose two efficient 2-approximation algorithms.

5.1 *Global* Algorithm

This section presents the first algorithm, called *Global*. We start by giving a definition of *query distance* in digraphs. Given a digraph G and two vertex sets $R_1, R_2 \subseteq V_G$, the *directed distance* from R_1 to R_2 is defined as $\text{dist}_G\langle R_1, R_2\rangle = \max_{u\in R_1, v\in R_2} \text{dist}_G\langle u, v\rangle$. Furthermore, the *bi-directed distance* between two vertex sets R_1 and R_2 is defined as $\text{dist}_G(R_1, R_2) = \max\{\text{dist}_G\langle R_1, R_2\rangle, \text{dist}_G\langle R_2, R_1\rangle\}$. Based on this, we give the definition of *query distance* as follows.

Definition 5.1. **(Query Distance).** Given a query vertex set Q and a set of vertices X in digraph H, the query distance $\text{dist}_H(X, Q)$ is defined as the *bi-directed distance* between Q and X in H.

For simplicity, we use $\text{dist}_G(v, Q)$ to represent the query distance between vertex v (i.e., $\{v\}$) and Q in digraph G. Moreover, we use $\text{dist}_G(G, Q)$ to represent the query distance between vertices V_G and Q in digraph G. With the above definitions, we present the details of *Global* in Algorithm 1. Specifically, the algorithm first invokes a procedure to find a maximal D-truss containing the query vertex set Q, which is outlined in Algorithm 2. Then, it iteratively removes the farthest vertex v from the query vertex set Q and maintains the remaining graph as a D-truss, until the remaining graph does not contain Q or is no longer a D-truss (lines 3-8). Note that, in each iteration, after removing the farthest vertex, we maintain the D-truss by deleting the edges whose cycle/flow support is smaller than k_c/k_f. Moreover, if there are multiple vertices sharing the same largest query distance, these vertices are all deleted in one iteration. Then, we can obtain a list of candidate D-truss graphs at the end of the iterations. Finally, the D-truss with the minimum query distance is returned as the answer (lines 9-10).

Algorithm 1 *Global* Algorithm

Input: a digraph G; a query vertex set Q; integers k_c and k_f
Output: a connected D-truss H^* with the minimum diameter
1: $i \leftarrow 0$;
2: $G_i \leftarrow$ *FindDTruss*(G, Q, k_c, k_f) using Algorithm 2;
3: **while** $Q \subseteq V_{G_i}$ **do**
4: $\quad v \leftarrow$ *FindFarthestVertex*(G_i, Q);
5: $\quad \text{dist}_{G_i}(G_i, Q) \leftarrow \text{dist}_{G_i}(v, Q)$;
6: $\quad$ Delete vertex v and its incident edges from G_i;
7: $\quad$ Maintain G_i as D-truss by removing vertices/edges;
8: $\quad$ Let the remaining graph as G_{i+1}; $i \leftarrow i + 1$;
9: $H^* \leftarrow \underset{H\in\{G_0, G_1, \ldots, G_{i-1}\}}{\arg\min} \text{dist}_H(H, Q)$;
10: **return** H^*

The procedure of finding the maximal D-truss containing Q is elaborated in Algorithm 2. Recall the minimum degree property of D-truss (Property 1) states that every vertex v of a D-truss H must satisfy $\deg_H(v) \geq \max\{2k_c, k_f + 1\}$. Thus, the algorithm first iteratively deletes all disqualified vertices and their incident edges from the original graph G, where each vertex v has $\deg_G(v) < \max\{2k_c, k_f + 1\}$ (lines 1-12). It then computes the cycle support and flow support for each edge remained in G (lines 13-16). Next, the algorithm iteratively removes the edges whose cycle support is smaller than k_c or flow support is smaller than k_f (lines 17-25). Note that the step of *D-truss maintenance* in Algorithm 1 (line 7) can use the similar operations shown in the lines 13-25 of Algorithm 2. Hence, the details of *D-truss maintenance* are omitted in this paper.

Example 3. *We use the running example in Figure 4 to illustrate the* Global *algorithm. Let vertex v_3 be the query vertex and $k_c = k_f = 1$. The algorithm first finds the maximal (1, 1)-truss D_0 containing v_3, i.e., the subgraph in the grey region. Then, it proceeds to delete vertex v_7, which has the maximum query distance of 4. After deleting vertex v_7 and its incident edges, the subgraph composed by $\{v_1, v_2, v_3, v_4, v_5, v_6\}$ is still a (1, 1)-truss, denoted as D_1. Next, the algorithm continues to delete the vertex having the maximal query distance, i.e., v_6 and v_5, in the following two iterations, and obtain two more (1, 1)-trusses, i.e, D_2: $\{v_1, v_2, v_3, v_4, v_5\}$ and D_3: $\{v_1, v_2, v_3, v_4\}$. Finally, the algorithm returns D_3, which has the minimum query distance of 1. Note that D_3 is also the (1, 1)-truss having the minimum diameter, which is equal to 1 as well.*

5.1.1 Theoretical Analysis. We now analyze the quality approximation and time complexity of the *Global* algorithm.

Algorithm 2 *FindDTruss*

Input: a digraph G; a query vertex set Q; integers k_c and k_f
Output: a maximal D-truss of G

1: Let L_v and L_e be empty queues of vertices and edges respectively;
2: **for** each vertex $v \in V_G$ **do**
3: Compute $\deg_G(v) = \deg_G^+(v) + \deg_G^-(v)$;
4: **if** $\deg_G(v) < \max\{2k_c, k_f + 1\}$ **then**
5: $L_v \leftarrow L_v \cup \{v\}$;
6: **while** $L_v \neq \emptyset$ **do**
7: Pop out a vertex v from L_v;
8: **for** vertex $u \in \mathsf{N}_G^+(v) \cup \mathsf{N}_G^-(v)$ **do**
9: $\deg_G(u) \leftarrow \deg_G(u) - 1$;
10: **if** $\deg_G(u) < \max\{2k_c, k_f + 1\} \wedge u \notin L_v$ **then**
11: $L_v \leftarrow L_v \cup \{u\}$;
12: Remove v and its incident edges from G;
13: **for** each edge $e \in E_G$ **do**
14: Compute $\mathrm{csup}_G(e)$ and $\mathrm{fsup}_G(e)$;
15: **if** ($\mathrm{csup}_G(e) < k_c$ or $\mathrm{fsup}_G(e) < k_f$) **then**
16: $L_e \leftarrow L_e \cup \{e\}$;
17: **while** $L_e \neq \emptyset$ **do**
18: Pop out an edge $\langle u, v\rangle$ from L_e;
19: $\mathsf{N}(u) \leftarrow \mathsf{N}_G^+(u) \cup \mathsf{N}_G^-(u)$; $\mathsf{N}(v) \leftarrow \mathsf{N}_G^+(v) \cup \mathsf{N}_G^-(v)$;
20: **for** each vertex $w \in \mathsf{N}(u) \cap \mathsf{N}(v)$ **do**
21: **for** $e \in \{\langle w, u\rangle, \langle w, v\rangle, \langle u, w\rangle, \langle v, w\rangle\} \cap E_G$ **do**
22: Update $\mathrm{csup}_G(e)$ and $\mathrm{fsup}_G(e)$ accordingly;
23: **if** ($\mathrm{csup}_G(e) < k_c$ or $\mathrm{fsup}_G(e) < k_f$) **then**
24: $L_e \leftarrow L_e \cup \{e\}$;
25: Remove the edge $\langle u, v\rangle$ from G;
26: **return** graph G;

Quality Analysis. Assume that H_g is the D-truss community containing Q returned by *Global* and H^* is an optimal solution to the D-truss community search problem. For $\alpha \geq 1$, we say that *Global* achieves α-approximation iff $\mathrm{diam}(H_g) \leq \alpha \cdot \mathrm{diam}(H^*)$.

Lemma 5.2. $\mathrm{dist}_{H_g}(H_g, Q) \leq \mathrm{dist}_{H^*}(H^*, Q)$

Proof. Assume that $\{G_0, ..., G_{i-1}\}$ is the list of candidate D-truss graphs obtained by *Global*. We have $G_{i-1} \subseteq ... \subseteq G_1 \subseteq G_0$ and $G_0 \subseteq G$. Obviously, $H_g \in \{G_0, ..., G_{i-1}\}$ and $H^* \subseteq G_0$. We prove the lemma with the help of G_{i-1}. Specifically, we consider two cases of the relationship between G_{i-1} and H^*:

- **(Case I)** $H^* \subseteq G_{i-1}$. Assume that the vertex $v^* \in V_{H^*}$ has the largest query distance in H^*, i.e., $\mathrm{dist}_{H^*}(v^*, Q) = \mathrm{dist}_{H^*}(H^*, Q)$. Obviously, $v^* \in V_{G_{i-1}}$. We show that $\mathrm{dist}_{G_{i-1}}(v^*, Q) = \mathrm{dist}_{G_{i-1}}(G_{i-1}, Q)$. Otherwise, suppose that $\mathrm{dist}_{G_{i-1}}(v^*, Q) < \mathrm{dist}_{G_{i-1}}(G_{i-1}, Q)$; then there must exist another vertex $u \neq v^*$ having $\mathrm{dist}_{G_{i-1}}(u, Q) = \mathrm{dist}_{G_{i-1}}(G_{i-1}, Q)$. Thus, *Global* can obtain a new D-truss by deleting u from G_{i-1}, indicating that G_{i-1} is not the last D-truss obtained by Algorithm 1. This is a contradiction. Thus, $\mathrm{dist}_{G_{i-1}}(v^*, Q) = \mathrm{dist}_{G_{i-1}}(G_{i-1}, Q)$ holds. Therefore, we have $\mathrm{dist}_{H^*}(H^*, Q) = \mathrm{dist}_{H^*}(v^*, Q) \geq \mathrm{dist}_{G_{i-1}}(v^*, Q) = \mathrm{dist}_{G_{i-1}}(G_{i-1}, Q)$. As $\mathrm{dist}_{H_g}(H_g, Q) \leq \mathrm{dist}_{G_{i-1}}(G_{i-1}, Q)$, we have $\mathrm{dist}_{H_g}(H_g, Q) \leq \mathrm{dist}_{H^*}(H^*, Q)$.
- **(Case II)** $H^* \nsubseteq G_{i-1}$. Since $H^* \subseteq G_0$, there exists $j \in [0, i-2]$ such that $H^* \subseteq G_j$ and $H^* \nsubseteq G_{j+1}$. Thus, there must exist a vertex $v^* \in V_{H^*}$ such that $v^* \in G_j$, $v^* \notin G_{j+1}$, and v^* is deleted in line 6 of Algorithm 1. We prove this statement by contradiction. Suppose otherwise no vertex of H^* is deleted in line 6 of Algorithm 1. Then, no vertices/edges of H^* would be deleted in line 7 of Algorithm 1, since H^* itself satisfies the constraints of D-truss. Thus, $H^* \subseteq G_{j+1}$, which is a contradiction. As v^* is deleted in line 6 of Algorithm 1, we have $\mathrm{dist}_{G_j}(v^*, Q) = \mathrm{dist}_{G_j}(G_j, Q)$. Hence, $\mathrm{dist}_{H^*}(H^*, Q) \geq \mathrm{dist}_{H^*}(v^*, Q) \geq \mathrm{dist}_{G_j}(v^*, Q) = \mathrm{dist}_{G_j}(G_j, Q)$. As $\mathrm{dist}_{H_g}(H_g, Q) \leq \mathrm{dist}_{G_j}(G_j, Q)$, we have $\mathrm{dist}_{H_g}(H_g, Q) \leq \mathrm{dist}_{H^*}(H^*, Q)$.

Hence, the proof is completed. □

Lemma 5.3. *Given a digraph G and a query vertex set Q, we have* $\mathrm{dist}_G(G, Q) \leq \mathrm{diam}(G) \leq 2\mathrm{dist}_G(G, Q)$.

Proof. This lemma holds for the triangle inequality of directed distances. □

Theorem 5.4. *Algorithm 1 finds a 2-approximation solution for the D-truss community search problem.*

Proof. According to Lemmas 5.2 and 5.3, $\frac{\mathrm{diam}(H_g)}{2} \leq \mathrm{dist}_{H_g}(H_g, Q) \leq \mathrm{dist}_{H^*}(H^*, Q) \leq \mathrm{diam}(H^*)$. Hence, $\mathrm{diam}(H_g) \leq 2\mathrm{diam}(H^*)$ holds. □

Complexity Analysis. In the following, we analyze the time complexity of Algorithms 1 and 2. Let t be the total number of iterations taken by Algorithm 1 and $m' = |E_{G_0}| \leq m$. Let $\mathsf{N}(v) = \{u : \langle u, v\rangle \in E_G \text{ or } \langle v, u\rangle \in E_G\}$ and $\mathsf{N}^*(v) = \{u \in \mathsf{N}(v) : \deg_G(u) \geq \deg_G(v)\}$.

Lemma 5.5. *For a vertex v in G, $|\mathsf{N}^*(v)| \leq \sqrt{2m}$.*

Proof. For a vertex $u \in V_G$, if $\deg_G(u) \geq \deg_G(v)$, then $\deg_G(u) \geq |\mathsf{N}^*(v)|$. Thus, $\sum_{u \in \mathsf{N}^*(v)} \deg_G(u) \geq |\mathsf{N}^*(v)| \cdot \deg_G(v) \geq |\mathsf{N}^*(v)|^2$. In addition, $\sum_{u \in \mathsf{N}^*(v)} \deg_G(u) \leq \sum_{u \in V_G} \deg_G(u) = 2m$. Combining the above two inequalities, we have $|\mathsf{N}^*(v)|^2 \leq 2m$. Therefore, $|NB^*(v)| \leq \sqrt{2m}$. □

Lemma 5.6. *Algorithm 2 takes $O(m^{1.5})$ time.*

Proof. Algorithm 2 first deletes all the vertices whose degree is less than $\max\{2k_c, k_f + 1\}$. This step takes $O(m + n) = O(m)$ time. Then, it removes all the edges whose cycle support and flow support do not satisfy the requirements of D-truss. For each edge deletion, all triangles involving this edge should be enumerated. This step has a time complexity of $O(\sum_{\langle u,v\rangle \in E_G} \min\{\deg_G(u), \deg_G(v)\}) = O(\sum_{v \in V_G}(\deg_G(v) \cdot |\mathsf{N}^*(v)|)) \subseteq O(\sum_{v \in V_G}(\deg_G(v) \cdot \sqrt{2m})) = O(m^{1.5})$. Overall, Algorithm 2 takes $O(m^{1.5})$ time. □

Theorem 5.7. *Algorithm 1 takes $O(m^{1.5} + t \cdot m'^{1.5})$ time, where $m' = |E_{G_0}|$ and t is the number of iterations incurred.*

Proof. First, Algorithm 1 finds the maximal D-truss of G, which takes $O(m^{1.5})$ time as analyzed in Lemma 5.6. Then, it iteratively deletes the vertex with the largest query distance starting from G_0 with m' edges. In each iteration, Algorithm 1 needs to process two tasks: (i) it computes the query distance using $O(|Q| \cdot m')$ time; (ii) it performs D-truss maintenance using $O(m'^{1.5})$ time. As $|Q| \ll m'$, the time cost of each iteration is $O(m'^{1.5})$. Hence, the time complexity of all iterations is $O(t \cdot m'^{1.5})$. In total, Algorithm 1 takes $O(m^{1.5} + t \cdot m'^{1.5})$ time. □

5.1.2 Optimizations. We propose two optimization techniques of *smart deletion* and *binary search* to accelerate the process of Algorithm 1.

Smart Deletion. We first optimize the *Global* algorithm based on the following observation.

Observation 1. *Consider two sequential D-trusses G_i and G_{i+1} in Algorithm 1, where G_{i+1} is the D-truss obtained by deleting the vertex having the largest query distance from G_i. Obviously, G_{i+1} is not the final answer if* $\text{dist}_{G_{i+1}}(G_{i+1}, Q) > \text{dist}_{G_i}(G_i, Q)$. *The* Global *algorithm may generate many such useless D-trusses.*

Following this observation, a way to improve *Global* is to reduce the generation of useless intermediate D-trusses as much as possible. That is, if it is found that $\text{dist}_{G_{i+1}}(G_{i+1}, Q) > \text{dist}_{G_i}(G_i, Q)$, we continue to delete all the vertices whose query distances are no less than $\text{dist}_{G_i}(G_i, Q)$. After deleting these vertices, we need to maintain the remaining graph as a new D-truss. If the query distance of the new D-truss is not less than $\text{dist}_{G_i}(G_i, Q)$, we repeat the above process until getting a new D-truss whose query distance is less than $\text{dist}_{G_i}(G_i, Q)$ or no such D-truss exists.

Binary Search. Equipped with smart deletion, the algorithm can avoid generation of many useless D-trusses . Consequently, the algorithm can progressively converge to the minimum query distance. However, in the worst case, the algorithm reduces the query distance only by 1 in each iteration, which is not efficient. To address this deficiency, we employ a binary search technique to speed up the process. In this way, the total number of iterations t is bounded by $O(\log d_0)$, where $d_0 = \text{dist}_{G_0}(G_0, Q)$. The following example illustrates how our binary search technique is employed for *Global*.

Example 4. *In Figure 4, assume that the maximal (1, 1)-truss D_0, whose query distance is 4, is already found. We set the low and upper bounds of the query distance as $d_{min} = 1$ and $d_{max} = 4$, respectively. Then, the next query distance for examination is $d = 2.5$. Thus, we delete the vertices whose query distances are larger than 2.5. Correspondingly, vertices v_7 and v_6 are deleted and the (1, 1)-truss D_1 composed by $\{v_1, v_2, v_3, v_4, v_5\}$ is obtained. As the query distance of D_1 is 2, we set $d_{max} = 2$. Then, $d = 1.5$, and we delete vertex v_5, whose query distance is larger than 1.5. Hence, we get the (1, 1)-truss D_2 composed by $\{v_1, v_2, v_3, v_4\}$, which is the final answer.*

Algorithm 3 Local Algorithm

Input: a digraph G; a query vertex set Q; integers k_c and k_f
Output: a D-truss H_l with a small diameter
1: $G_0 \leftarrow$ *FindDTruss*(G, Q, k_c, k_f) using Algorithm 2;
2: $d \leftarrow 1$;
3: **while** $d \leq \text{dist}_{G_0}(G_0, Q)$ **do**
4: Let $V_d = Q \cup \{v \in V_G : \text{dist}_{G_0}(v, Q) \leq d\}$;
5: Let G_d be the induced subgraph of G by vertices V_d;
6: $G_d \leftarrow$ *FindDTruss*(G_d, Q, k_c, k_f);
7: **if** $G_d = \varnothing$ **then**
8: $d \leftarrow d + 1$;
9: **goto** step 3;
10: $S \leftarrow \emptyset$;
11: **for** each vertex $v \in V_{G_d}$ **do**
12: **if** $\text{dist}_{G_d}(v, Q) > d$ **then**
13: $S \leftarrow S \cup \{v\}$;
14: **if** $S = \emptyset$ **then**
15: $H_l \leftarrow G_d$; **Break**;
16: Delete S and their incidents edges E_S from G_d;
17: $G_d \leftarrow$ *DTrussMaintain*(G_d, k_c, k_f, E_S);
18: **goto** step 7;
19: **return** H_l;

5.2 *Local* Algorithm

The *Global* algorithm finds the D-truss community in a top-down manner. In this section, we present another algorithm, called *Local*, which works in a bottom-up manner. The basic idea of *Local* is to find a possible D-truss G_d starting from the minimum query distance of d (i.e., $d = 1$). If such a G_d does not exist, we increase d and find the corresponding D-truss G_{d+1} iteratively. The algorithm terminates by finding the D-truss with $\text{dist}_{G_d}(G_d, Q) = d$, which is returned as the answer.

Algorithm 3 shows the details of the *Local* algorithm. It first computes the maximal D-truss using Algorithm 2 (line 1). It then starts with $d = 1$ (line 2), and iteratively finds the D-truss G_d with $\text{dist}_{G_d}(G_d, Q) = d$ (line 2). Specifically, it collects all vertices whose query distances are no greater than d into $V_d = Q \cup \{v \in V_G : \text{dist}_{G_0}(v, Q) \leq d\}$ (line 4). Afterwards, it constructs G_d as the induced subgraph of G by V_d (line 5). Next, the algorithm examines whether G_d contains a D-truss D_{G_d} whose query distance is d, i.e., $\text{dist}_{G_d}(G_d, Q) = d$ (lines 6-18). If such a D-truss G_d exists (lines 14-15), it is returned as the final result of H_l (line 19). Otherwise, the algorithm increases d by 1 for a new iteration.

Example 5. *Consider our running example in Figure 4. Assume that the maximal (1, 1)-truss D_0 is already found.*

We start with $d = 1$ and get the sub-digraph composed by $\{v_1, v_2, v_3, v_4\}$, since these four vertices' query distances are equal to 1. As this sub-digraph is exactly a (1,1)-truss, it is returned as the answer.

In the following, we analyze the quality approximation and time complexity of the *Local* algorithm.

Lemma 5.8. *For the community H_l returned by Algorithm 3, $\text{dist}_{H_l}(H_l, Q) \leq \text{dist}_{H^*}(H^*, Q)$ holds.*

Proof. Suppose that $\text{dist}_{H_l}(H_l, Q) > \text{dist}_{H^*}(H^*, Q)$. Thus, there exists another D-truss G_d satisfying $\text{dist}_{G_d}(G_d, Q) < \text{dist}_{H_l}(H_l, Q)$. However, according to the principle of Algorithm 3, H_l has the minimum query distance, which is a contradiction. □

Theorem 5.9. *Algorithm 3 returns a 2-approximation solution for the D-truss community search problem.*

Proof. Similar to the proof of Theorem 5.4, Theorem 5.9 can be proved by Lemmas 5.3 and 5.8. □

Let $m' = |E_{G_0}|$ and $\delta = \min\{\text{dist}_{G_0}(G_0, Q), n\}$, where G_0 is the maximal D-truss of G containing Q.

Theorem 5.10. *Algorithm 3 takes $O(m^{1.5} + \delta \cdot m'^{1.5})$ time.*

Proof. The algorithm first finds the maximal D-truss in $O(m^{1.5})$ time, as shown in Theorem 5.6. Then, it examines whether there exists a D-truss G_d with $\text{dist}_{G_d}(G_d, Q) = d$. In each iteration, it takes $O(m'^{1.5})$ time to find the D-truss. In total, the algorithm takes $O(m^{1.5} + \delta \cdot m'^{1.5})$ time. □

6 INDEX-BASED ALGORITHMS

Both *Global* and *Local* need to retrieve the maximal D-truss. However, computing the maximal D-truss online from scratch is very inefficient for large digraphs. Actually, all the possible maximal D-trusses of a given digraph are determined, which can be precomputed and stored in an index for speeding up our algorithms. Motivated by this, in this section, we discuss how to employ D-truss decomposition to accelerate our algorithms. Specifically, we first present the details of D-truss decomposition, i.e., to compute all the possible D-trusses for a given digraph. Then, we develop an index to store the results of D-truss decomposition. Finally, we describe how to retrieve the maximal D-truss from the index.

6.1 D-truss Decomposition

In this section, we present an algorithm for D-truss decomposition. First, we give some definitions. For a given edge $e = \langle u, v \rangle \in E_G$, if e presents in a D-truss $D_{(k_c, k_f)}$, we say that (k_c, k_f) is a trussness of edge e, denoted as $\mathsf{T}(e) = \{(k_c, k_f)\}$. Note that an edge may belong to multiple D-trusses. Hence, an edge may have multiple trussnesses.

Definition 6.1. **(Trussness Dominance).** Given 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 and only if the following conditions hold: (1) $k_c^1 \geq k_c^2$ and $k_f^1 > k_f^2$; or (2) $k_c^1 > k_c^2$ and $k_f^1 \geq k_f^2$.

Definition 6.2. **(Skyline Trussness).** Given an edge e and all its trussnesses $\mathsf{T}(e) = \{(k_c^1, k_f^1), (k_c^2, k_f^2), ..., (k_c^n, k_f^n)\}$, the skyline trussness of e, denoted as $\mathsf{ST}(e)$, contains the trussnesses that are not dominated by others. Formally, $\mathsf{ST}(e) = \{(k_c^i, k_f^i) \in T(e) : \nexists (k_c^j, k_f^j) \in T(e), \text{s.t.}, (k_c^i, k_f^i) \prec (k_c^j, k_f^j)\}$.

The problem of D-truss decomposition is to compute the skyline trussness for every edge in digraph G. Algorithm 4 outlines the D-truss decomposition algorithm. It first computes the cycle support and flow support of every edge in G (lines 1-2). Then, assuming $k_f = 0$, the algorithm computes the $(k_c, 0)$-truss, denoted by $D_{(k_c,0)}$, for k_c from 0 to k_{cmax} (lines 5-20). When computing a specific $D_{(k_c,0)}$, it iteratively removes the edges whose cycle support is less than k_c (lines 7-18). Next, for all D-trusses $D_{(k_c,0)}$ for k_c from 0 to k_{cmax}, the algorithm finds the candidate skyline trussness for every edge in $D_{(k_c,0)}$ (lines 21-28). Specifically, for each D-truss $D_{(k_c,0)}$, the algorithm first sorts all the edges in ascending order of their flow support (line 23). Then, an edge e with the minimum flow support (denoted by k_f) is examined (lines 25-26). The (k_c, k_f) is considered as the candidate skyline trussness of e and checked by the procedure *Compute_SkyTruss* (line 27). After examination, the edge $e = \langle u, v \rangle$ with the minimum flow support is deleted from the current D-truss by employing the procedure *DeleteEdge* (line 28). Note that if the removal of one edge affects other edges' support, the procedure will directly compute the skyline trussness of the affected edges. The process continues until the current D-truss becomes empty (line 24).

Next, we describe the details of the two procedures, i.e., *Compute_SkyTruss* and *DeleteEdge*, as shown in Algorithm 5. Specifically, for the procedure *Compute_SkyTruss*, when checking a candidate skyline trussness of edge e, if it dominates some existing skyline trussness in $\mathsf{ST}(e)$, the existing one is deleted from $\mathsf{ST}(e)$ (lines 1-3). Otherwise, if the candidate skyline trussness is not dominated by any existing skyline trussness, it is added to $\mathsf{ST}(e)$ (lines 4-6). For the procedure *DeleteEdge*, if an edge $e = \langle u, v \rangle$ is deleted, it updates the cycle support and flow support of the edges incident to u or v (lines 8-9). If the cycle support (or flow support) of these edges does not satisfy the constraints of a (k_c, k_f)-truss, these edges are deleted by recursively calling the procedure of *DeleteEdge* and their corresponding skyline trussnesses are computed (lines 10-12).

Example 6. *We use Figure 4 to illustrate the D-truss decomposition. Assume that the (0,0)-truss has been examined and*

Algorithm 4 D-truss Decomposition

Input: a digraph $G = (V_G, E_G)$
Output: skyline trussness for every edge in G

for each edge $e \in E_G$ **do**
  Compute $\text{csup}_G(e)$ and $\text{fsup}_G(e)$;
$k_c \leftarrow 0$; $k_f \leftarrow 0$; $D_{(0,0)} \leftarrow G$;
Let L_e be an empty queue;
while $G \neq \emptyset$ **do**
  $k_c \leftarrow k_c + 1$;
  for each edge $e = \langle u, v \rangle \in E_G$ **do**
    if $\text{csup}_G(e) < k_c$ **then**
      $L_e \leftarrow L_e \cup \{e\}$;
  while $L_e \neq \emptyset$ **do**
    Pop out an edge $e = \langle u, v \rangle$ from L_e;
    Delete e from G;
    $\mathsf{N}(u) \leftarrow \mathsf{N}^+_G(u) \cup \mathsf{N}^-_G(u)$; $\mathsf{N}(v) \leftarrow \mathsf{N}^+_G(v) \cup \mathsf{N}^-_G(v)$;
    for each vertex $w \in \mathsf{N}(u) \cap \mathsf{N}(v)$ **do**
      for $e' \in \{\langle w, u \rangle, \langle w, v \rangle, \langle u, w \rangle, \langle v, w \rangle\} \cap E_G$ **do**
        Update $\text{csup}_G(e')$ and $\text{fsup}_G(e')$ accordingly;
        if $(\text{csup}_G(e') < k_c)$ **then**
          $L_e \leftarrow L_e \cup \{e\}$;
  $D_{(k_c,0)} \leftarrow G$;
Let the maximum flow truss as $k_{cmax} \leftarrow k_c$;
for $k_c \leftarrow 0$ to k_{cmax} **do**
  Let the graph $H \leftarrow D_{(k_c,0)}$;
  Sort all edges $e \in E_H$ in ascending order of $\text{fsup}_H(e)$;
  while $D_{(k_c,0)} \neq \varnothing$ **do**
    let e be an edge with the minimum $\text{fsup}_H(e)$ in H;
    $k_f \leftarrow \text{fsup}_H(e)$;
    Compute_SkyTruss(e, k_c, k_f);
    DeleteEdge(e, k_c, k_f, H);

Algorithm 5 Procedures

Procedure Compute_SkyTruss(e, k_c, k_f)
for each trussness $(k'_c, k'_f) \in \mathsf{ST}(e)$ **do**
  if $(k'_c, k'_f) \prec (k_c, k_f)$ **then**
    Delete (k'_c, k'_f) from $\mathsf{ST}(e)$;
  if $(k_c, k_f) \prec (k'_c, k'_f)$ **then**
    Return; //(k_c, k_f) is dominated in $\mathsf{ST}(e)$.
$\mathsf{ST}(e) \leftarrow \mathsf{ST}(e) \cup \{(k_c, k_f)\}$;

Procedure DeleteEdge(e, k_c, k_f, H)
Delete $e = \langle u, v \rangle$ from H;
for each edge e' incident to u or v **do**
  Update $\text{csup}_H(e')$ and $\text{fsup}_H(e')$ accordingly;
  if $\text{csup}_H(e') < k_c$ or $\text{fsup}_H(e') < k_f$ **then**
    Compute_SkyTruss(e', k_c, k_f);
    DeleteEdge(e', k_c, k_f, H);

the edge $\langle v_7, v_5 \rangle$'s current skyline trussness is (0, 1). Next, we examine the (1, 0)-truss (i.e., the grey area in Figure 4). According to the flow support in the (1, 0)-truss, the edge $\langle v_7, v_5 \rangle$ (with the minimum flow support being 1) is deleted. Then (1, 1) is checked with the existing skyline trussness of edge $\langle v_7, v_5 \rangle$. As (1, 1) dominates (0, 1), (1, 1) is inserted into the skyline trussness set of edge $\langle v_7, v_5 \rangle$ while (0, 1) is deleted. Afterwards, the edge $\langle v_7, v_5 \rangle$ is deleted and the cycle support and flow support of

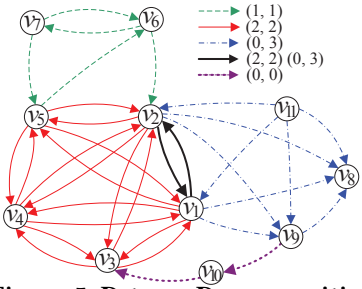


Figure 5: D-truss Decomposition

edges $\langle v_7, v_6 \rangle$, $\langle v_6, v_7 \rangle$, and $\langle v_6, v_5 \rangle$ are updated, after which the edges $\langle v_7, v_6 \rangle$ and $\langle v_6, v_7 \rangle$ are also deleted. The (1, 0)-truss is continuously examined until all edges are deleted. Finally, we can find the skyline trussnesses for all the edges, as shown in Figure 5.

6.2 Index-based Maximal D-truss Finding

We first introduce an index to store the skyline trussnesses of all the edges in G. Then, we present an index-based method to find the maximal D-truss.

D-Truss Index. The D-Truss index is constructed alike the adjacency list. For every vertex $v \in V_G$, we keep both its in-neighbors $\mathsf{N}^+(v)$ and out-neighbors $\mathsf{N}^-(v)$, as well as the skyline trussnesses of its incident edges.

Index-based Maximal D-truss Finding. The basic idea is to use the query vertex set Q as seeds and find all the edges of the maximal D-truss using a breath-first search. Algorithm 6 gives the details. Specifically, it initializes a queue L_v with the query vertex set Q and mark them as visited (lines 1-2). Then, it pops out a vertex v from L_v and examines all the edges incident to v (lines 4-12). If an edge e has not been visited, we check the dominance relationship between the skyline trussness of e and the given (k_c, k_f). If there exists a skyline trussness of e dominating or equal to (k_c, k_f) (denoted as $(k_c, k_f) \preceq (k^i_c, k^i_f)$ in line 9), it implies that e belongs to the maximal D-truss and is added to G_0 (lines 9-12). Moreover, the algorithm adds the unvisited edge point to L_v. To find the remaining edges for the maximal D-truss, the algorithm continues to visit all edges incident to $v \in L_v$ until L_v becomes empty.

Example 7. *Consider our running example. We start from the query vertex v_3. The edges incident to v_3 are first examined. According to Figure 5, as all the edges incident to v_3, except the edge $\langle v_{10}, v_3 \rangle$, dominate (1, 1) and all v_3's neighbors are not visited, these edges are added to the result. Correspondingly, v_3's in- and out-neighbors v_1, v_2, and v_4 are inserted into the queue. Then, the vertex v_1 is popped from the queue and examined. As v_3 is visited, there is no need to check the edges $\langle v_1, v_3 \rangle$ and $\langle v_3, v_1 \rangle$. In addition, the skyline trussnesses of edges $\langle v_1, v_8 \rangle$, $\langle v_1, v_9 \rangle$, $\langle v_{11}, v_1 \rangle$ do not dominate (1, 1). These three edges are not added to the result and thus the vertices v_8,*

Algorithm 6 Index-based Maximal D-truss Finding

Input: a digraph G; a query vertex set Q; integers k_c and k_f
Output: The maximal D-truss of G

```
1: Queue: L_v ← Q;
2: Mark all vertices of Q as visited;
3: Let G_0 be an empty graph;
4: while L_v ≠ ∅ do
5:     Pop out a vertex v from L_v;
6:     for each unvisited edge e incident to v do
7:         Mark e as visited;
8:         for each skyline trussness (k_c^i, k_f^i) ∈ ST(e) do
9:             if (k_c, k_f) ⪯ (k_c^i, k_f^i) then
10:                Add edge e into G_0;
11:                L_v ← L_v ∪ {u};
               // u is another end point of edge e
12:                Mark u as visited; Break;
13: return G_0;
```

v_9, v_{11} are not added to the queue. The algorithm continues until the queue becomes empty. Finally, we get the D-truss consisting of vertices $\{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$, which is the same as the result obtained by Algorithm 2.

Complexity Analysis. We analyze the complexity of Algorithms 4 and 6, as well as the memory size of the D-truss index. Let k_{cmax} and k_{fmax} be the maximum cycle/flow truss of a directed graph G, i.e., $k_{cmax} = \max_{k_c}\{H \subseteq G : H \text{ is } (k_c, 0)\text{-truss}\}$ and $k_{fmax} = \max_{k_f}\{H \subseteq G : H \text{ is } (0, k_f)\text{-truss}\}$. Algorithm 4 can be easily implemented in taking $O(\min\{k_{cmax}, k_{fmax}\} \cdot m^{1.5})$ time for D-truss decomposition and D-truss index construction. The space complexity of Algorithm 4 and the memory size of the D-truss index both take $O(\min\{k_{cmax}, k_{fmax}\} \cdot m)$ space. This is because the largest cardinality of a skyline trussness set $\mathsf{ST}(e)$ will not exceed $O(\min\{k_{cmax}, k_{fmax}\})$ for any edge e in G. In addition, Algorithm 6 takes $O(\min\{k_{cmax}, k_{fmax}\} \cdot m)$ time for finding the maximal D-truss, which is much faster than Algorithm 2.

7 EMPIRICAL EVALUATION

In this section, we conduct extensive experiments to evaluate the effectiveness and efficiency of our proposed model and algorithms. All experiments are conducted on a Linux Server with 2.10 GHz six-core CPU and 188 GB memory running Ubuntu 16.04.6. The algorithms are implemented in C++.

Datasets. We use six real-world datasets of directed networks in the experiments. Table 1 summarizes the statistics of these networks.The EAT network is retrieved from the Pajek datasets[2] and all other networks are obtained from the Stanford Network Analysis Project.[3] Note that among them, two networks of Email and Wiki have ground-truth communities.

[2] http://vlado.fmf.uni-lj.si/pub/networks/data/
[3] http://snap.stanford.edu/

Table 1: Dataset statistics.

Dataset	$\lvert V_G\rvert$	$\lvert E_G\rvert$	d_{max}	k_{cmax}	k_{fmax}
Email	1.0**K**	25.6**K**	544	14	21
EAT	23.1**K**	685**K**	1,106	3	8
Twitter	81.3**K**	1.8**M**	3,758	161	199
BerkStan	685**K**	7.6**M**	84,290	41	80
Wiki	1.8**M**	28.5**M**	238,607	36	37
Pokec	1.6**M**	30.6**M**	20,518	18	27

Queries and Parameters. We evaluate the performance of all algorithms using different queries by varying the parameters, including the query size $|Q|$, the degree rank, parameters k_c and k_f, # vertices $|V_G|$. For each network, we sort all vertices in descending order of their degrees. A vertex is said to be with degree rank of X%, if it belongs to the highest X% degree in the network. For each experiment, we run 200 queries for community search and report the average results.

7.1 Quality Evaluation

Exp-1: Quality Comparisons on Ground-truth Communities. We assess the quality of community models using two directed graphs with ground-truth communities, i.e., Email and Wiki. Specifically, Email is a communication network between the members of a large European research institution. There are 42 ground-truth communities, which are respectively formed by the individuals of 42 departments. Wiki is a web graph of Wikipedia hyperlinks. Each webpage belongs to one or more categories. The webpages fall into the same category form a ground-truth community.

We compare our model with five other state-of-the-art community models, i.e., D-core [12], CF-truss [29], flow-truss [29], cycle-truss [29], and k-truss [14]. Note that CF-truss [29] is implemented by combining the results of flow-truss and cycle-truss. To evaluate the quality of discovered communities, we use five metrics, including recall, precision, F1-score $= 2 \cdot \frac{recall \times precision}{recall + precision}$, community member similarities CMS^{in} and CMS^{out} [12]. Both CMS^{in} and CMS^{out} are metrics of computing the similarity among the members in a community H, defined as follows:

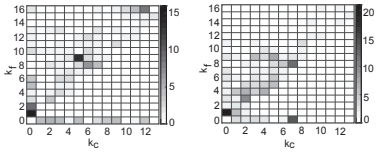
$$\text{CMS}^{\text{in}}(H) = \frac{1}{|V_H|^2} \sum_{u \in V_H} \sum_{v \in V_H} \frac{|\mathsf{N}^+_H(u) \cap \mathsf{N}^+_H(v)|}{|\mathsf{N}^+_H(u) \cup \mathsf{N}^+_H(v)|}$$

$$\text{CMS}^{\text{out}}(H) = \frac{1}{|V_H|^2} \sum_{u \in V_H} \sum_{v \in V_H} \frac{|\mathsf{N}^-_H(u) \cap \mathsf{N}^-_H(v)|}{|\mathsf{N}^-_H(u) \cup \mathsf{N}^-_H(v)|}$$

We test two kinds of queries, i.e., single query vertex ($|Q| = 1$) and multiple query vertices ($|Q| > 1$). For each multiple-vertex query, we randomly select 2-10 vertices that appear in a unique ground-truth community. To ensure a fair comparison, we compare the best community with the highest F1-score found by each community model.

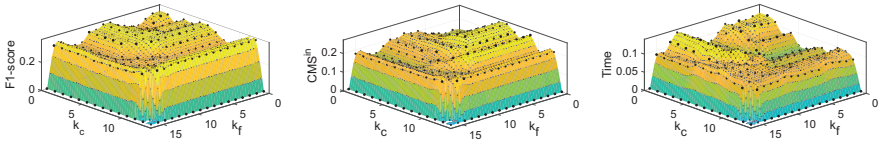
Table 2: Quality comparison of different community search models on *Email* with ground-truth communities.

Model	Precision		Recall		F1-Score		CMS^{in}		CMS^{out}		Time (sec)	
	$\|Q\| = 1$	$\|Q\| > 1$	$\|Q\| = 1$	$\|Q\| > 1$	$\|Q\| = 1$	$\|Q\| > 1$	$\|Q\| = 1$	$\|Q\| > 1$	$\|Q\| = 1$	$\|Q\| > 1$	$\|Q\| = 1$	$\|Q\| > 1$
D-truss	**0.5111**	**0.2539**	0.3166	0.6848	**0.3417**	**0.2859**	**0.3132**	**0.1493**	**0.3002**	**0.1352**	0.0136	0.1283
CF-truss	0.0736	0.0561	0.6792	0.7904	0.1234	0.0974	0.1445	0.1095	0.1325	0.0967	0.3076	0.2734
Cycle-truss	0.0657	0.0516	0.6921	0.7944	0.1130	0.0914	0.1213	0.1005	0.1101	0.0885	0.3206	0.2908
Flow-truss	0.0707	0.0539	0.6864	0.7947	0.1199	0.0945	0.1396	0.1071	0.1268	0.0937	0.2979	0.2655
D-core	0.0573	0.0475	**0.7586**	0.8213	0.1039	0.0873	0.1097	0.1000	0.0921	0.0832	**0.0009**	**0.0008**
k-truss	0.0644	0.0534	0.7518	**0.8589**	0.1137	0.0969	0.1163	0.0977	0.0949	0.0771	0.0574	0.0622



(a) $|Q| = 1$ (b) $|Q| > 1$

Figure 6: Optimal (k_c, k_f) Settings



(a) F1-Score (b) CMS^{in} (c) Time (sec)

Figure 7: Effects of k_c and k_f on Community Quality

Table 2 shows the results for the Email dataset. Similar performance trends are observed for the Wiki dataset, which is omitted due to space limitations. Clearly, our D-truss model significantly outperforms all other competitive models, in terms of precision, F1-score, CSM^{in}, and CSM^{out}. The high scores of precision and F1-score imply that the communities found by our model are closer to the ground truth. The reason behind is two-fold. First, D-truss requires that every edge is contained in at least k_c cycle triangles and k_f flow triangles, ensuring that the vertices in a D-truss are densely connected. Second, we impose the minimum-diameter constraint in our problem definition, which effectively excludes the irrelevant vertices from the result. In addition, as pointed out by [19], a cohesive community usually contains highly similar vertices. The highest scores of CSM^{in} and CSM^{out} also verify that the communities returned by D-truss have the strongest cohesive structure compared with the other models. However, the recall of our model is not as good as the other models. This is because the other models usually return communities with a very large size, which are likely to cover most of the vertices in the ground-truth communities. In terms of the running time performance, D-truss performs better than CF-truss, cycle-truss, and flow-truss, by using the proposed index.

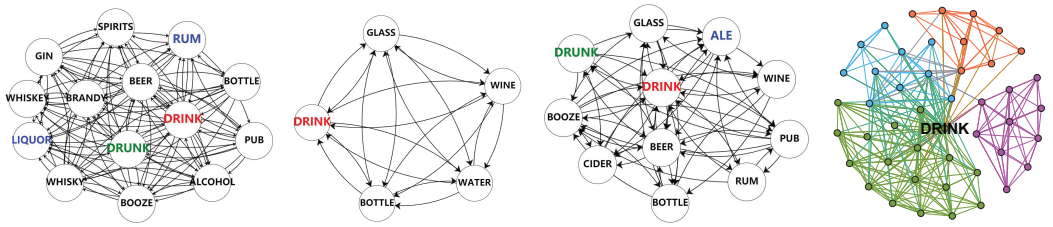
Figure 6 shows the optimal (k_c, k_f) distribution of the D-truss community using a heatmap. The optimal parameter settings for different query vertices are quite different. We find that no optimal parameters settings belong to the case of $k_c > k_f > 0$, meaning that in the Email dataset, the communities with mixed relationships tend to be dominated by hierarchical relationships.

We also evaluate the effects of k_c and k_f on the quality of retrieved communities. As shown in Figure 7(a), the F1-score remains relatively stable when k_c and k_f are varied, which is ensured by the minimum-diameter constraint in our problem definition. As for the CMS^{in} and CMS^{out} measures, they exhibit very similar trends. Only the CMS^{in} results are reported, in the interest of space. We can observe from Figure 7(b) that the medium values of k_c and k_f, in the range of 3–8, are more probable to yield higher CMS results. As for the total running time (Figure 7(c)), it generally decreases with the growth of k_c and k_f. This is because smaller values of k_c and k_f lead to larger maximal D-trusses and, hence, it takes the algorithm more time in diameter refining.

Exp-2: Case Studies on EAT. In the next set of experiments, we conduct case studies on the EAT dataset. EAT is a word association network, where the nodes represent English words. A directed link from nodes i to j indicates the associative relationship between the two: word j comes to mind when word i is shown as a stimulus.

We perform the D-truss community search on EAT using three queries: $Query1 = \{Q = \text{“}Drink\text{”}, k_c = 0, k_f = k_f^{max} = 7\}$, $Query2 = \{Q = \text{“}Drink\text{”}, k_c = k_c^{max} = 3, k_f = 0\}$, $Query3 = \{Q = \text{“}Drink\text{”}, k_c = \lceil k_c^{max}/2 \rceil = 2, k_f = \lceil k_f^{max}/2 \rceil = 4\}$. Here, $Query1$ and $Query2$ aim to find the community that is more flow-triangle-biased and more cycle-triangle-biased, respectively, while $Query3$ tries to retrieve the community with a mixed structure. For comparison, we also perform the D-core community search and CF-truss community search using “Drink” as the query vertex. Figure 8 shows the results of four communities by different models and queries. We make the following observations:

- First, the first three communities returned by our model are all related to “Drink”, including the containers for drinking (e.g., “Glass”, “Bottle”), the beverages (e.g., “Beer”, “Wine”), etc. In addition, they do not contain each other and have different semantics. Specifically, the (0,7)-truss shown in Figure 8(a) contains more liquor drinks that tend to have a hierarchical relationship with “Drink”. The (3,0)-truss shown in Figure 8(b) represents the words that can be recalled by each other, indicating a more equal relationship. In contrast, the



(a) (0,7)-truss (b) (3,0)-truss (c) (2,4)-truss (d) CF-truss

Figure 8: Case Studies on EAT

(2,4)-truss shown in Figure 8(c) contains the words having both hierarchical and equal relationships.

- Second, our model can identify the special vertices, e.g., those with high in-degree and low out-degree (i.e., the blue vertices) and those with low in-degree and high out-degree (i.e., the green vertices). For example, in Figure 8(a), the out-degree and in-degree of "Liquor" are 1 and 13, respectively, while the out-degree and in-degree of "Drunk" are 11 and 4, respectively. However, if we want to return the community containing both "Liquor" and "Drunk" using the D-core model, we should find the (1, 4)-core in the graph, whose result contains 6,980 vertices and 221,685 edges, and that is not quite meaningful.
- Third, both the CF-truss and D-core models cannot find communities effectively. For CF-truss, we set the same parameters as *Query*3. Figure 8(d) shows the corresponding result, which contains 46 vertices and 324 edges. For D-core, its results are extremely large. Even the minimum D-core community has 2,205 vertices and 61,451 edges. Clearly, these two models are less useful for real-world queries.

Discussion. Based on the results of quality evaluation, we offer two guidelines for the setting of k_c and k_f. First, the user can freely set the values of k_c and k_f for finding desired communities. In general, a large k_c (k_f) value will find a community that tends to have equal (hierarchical) relationships among its members. Second, if the user does not know how to set k_c and k_f but want to find a high-quality community, the setting of these two parameters can be left to the database server. More specifically, we suggest two methods:

- Max(CMS): Observing that the F1-scores and the CMS similarities follow a similar performance trend, the first method selects the setting that yields the highest CMS similarity (the average of CMS^{in} and CMS^{out});
- Max(k_c+k_f): The second method selects the setting that maximizes the sum of k_c and k_f so that the density of the returned community will be as high as possible. We use the CMS measure to break ties.

Both methods can work without knowing the ground-truth communities. Note that the possible values that k_c and k_f can be set are bounded by the skyline trussness of the query vertices' incident edges. While Max(CMS) needs to enumerate all the possible values, Max(k_c+k_f) can start testing from the largest possible sum until a feasible community is found. Table 3 shows that Max(CMS) can achieve a quality performance close to the ideal strategy that knows the ground truth and optimizes the F1-score. In contrast, Max(k_c+k_f)'s F1-score is about 30% lower than Max(CMS), but it is 4X faster.

Table 3: Comparison of parameter setting methods (*Email*, $|Q| = 1$).

Method	F1-Score	CMS^{in}	CMS^{out}	Latency (sec)
Ideal	0.3417	0.3132	0.3002	–
Max(CMS)	0.3241	0.3178	0.3046	6.06
Max(k_c+k_f)	0.2111	0.1753	0.1617	1.53

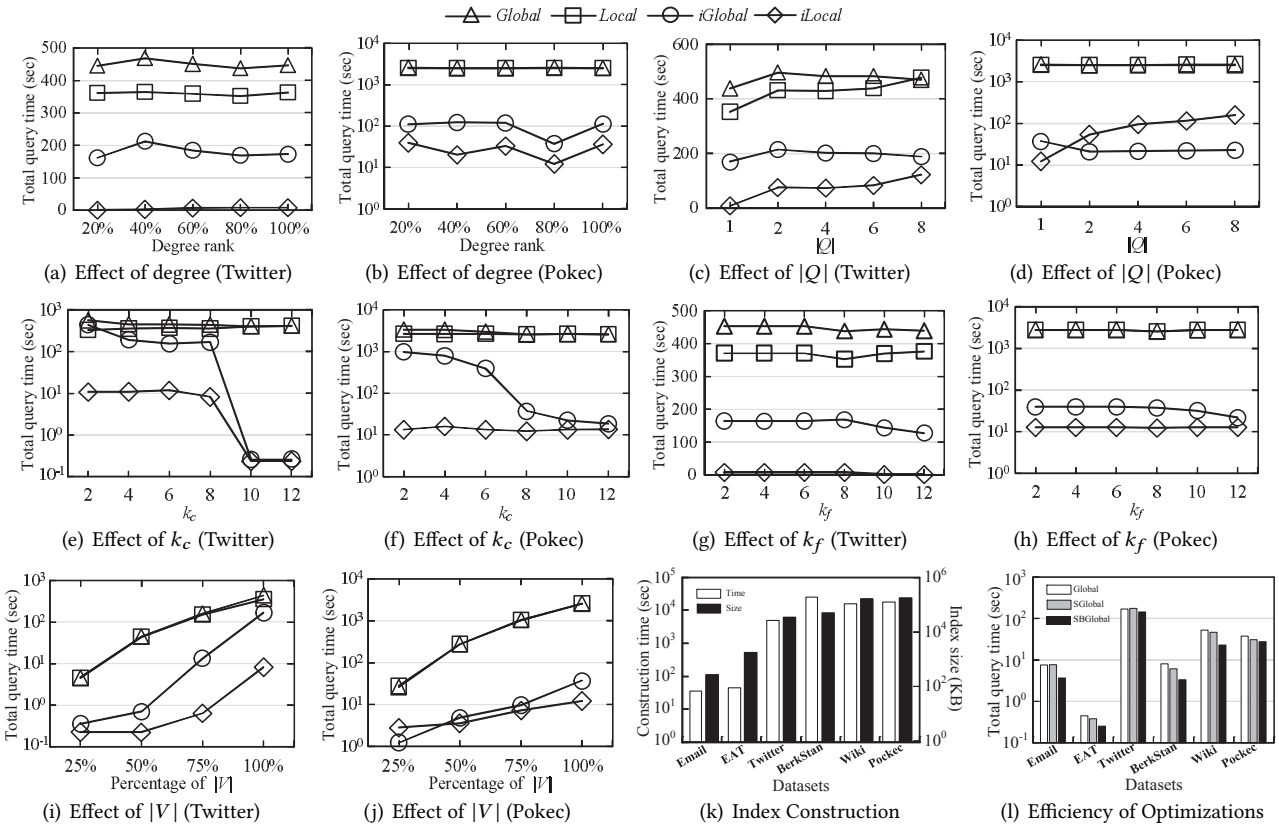
7.2 Efficiency Evaluation

In this section, we evaluate the efficiency of our proposed algorithms, including *Global*, *Local*, index-based *Global* (denoted as *iGlobal*), and index-based *Local* (denoted as *iLocal*), using the Twitter and Pokec datasets.

Exp-3: Varying the degree rank of query vertices. We sort the vertices in descending order of their degrees, and randomly select the query vertices from top 20%, 40%, 60%, 80%, 100% vertices in the testing graphs. As shown in Figures 9(a) and 9(b), all the algorithms are generally stable w.r.t. the varied degree of query vertices, indicating that the degree rank hardly affects the query time. This is because D-truss is mainly determined by the skyline trussness of the query vertices' incident edges, but not their degrees. *iLocal* outperforms the other three algorithms; *iGlobal* is the second best. In addition, *Local* performs better than *Global* on the Twitter dataset. On the other hand, *Local* and *Global* achieve a similar performance on the Pokec dataset, due to the expensive operation of finding the maximal D-truss in a larger graph.

Exp-4: Varying the query size $|Q|$. Figures 9(c) and 9(d) show the results when we vary $|Q|$ from 1 to 8. When $|Q|$ is increased, most of the algorithms incur slightly longer query time. Again, the index-based algorithms of *iLocal* and *iGlobal* consistently perform well.

Exp-5: Varying the parameters k_c **and** k_f. Figures 9(e) and 9(f) report the results by varying k_c from 2 to 12 on the Twitter and Pokec datasets, respectively. Figure 9(g) and 9(h) show the results by varying k_f from 2 to 12. When k_c or k_f is increased, the query time of all the four algorithms is reduced.



(a) Effect of degree (Twitter) (b) Effect of degree (Pokec) (c) Effect of $|Q|$ (Twitter) (d) Effect of $|Q|$ (Pokec)

(e) Effect of k_c (Twitter) (f) Effect of k_c (Pokec) (g) Effect of k_f (Twitter) (h) Effect of k_f (Pokec)

(i) Effect of $|V|$ (Twitter) (j) Effect of $|V|$ (Pokec) (k) Index Construction (l) Efficiency of Optimizations

Figure 9: Efficiency Evaluation of D-truss Community Search Algorithms

This is because when k_c or k_f grows, the maximal D-truss gets denser and closer, from which less time is needed to compute the final answer.

Exp-6: Scalability testing. We evaluate the scalability of our algorithms using a series of graphs with different fractions of vertices extracted from each dataset. Figures 9(i) and 9(j) show the results. As expected, all our algorithms scale very well to the graph size. For the (larger) Pokec dataset, the query time of the index-based algorithms, *iLocal* and *iGlobal*, increases more slowly than the non-index-based algorithms. This is because our D-truss index helps save time in finding the maximal D-truss in a large graph.

Exp-7: Index construction costs. We show the index construction time and index size in Figure 9(k), respectively. For all the datasets, the D-truss index can be built within 7 hours, and the index size is within 1 GB. We believe such indexing overheads are acceptable for community search applications.

Exp-8: Efficiency of optimizations. Finally, we evaluate the efficiency improvements of the two optimizations proposed in Section 5.1.2, i.e., smart deletion and binary search. Figure 9(l) reports the results. Note that *Global* denotes the basic algorithm without any optimizations; *SGlobal* integrates the smart deletion; *SBGlobal* is the algorithm with the two optimizations. The two optimizations work very well. For example, for the BerkStan dataset, the performance improvements are 23.9% and 46.3%, respectively, and they together improve the performance of the basic algorithm by about 60%.

8 CONCLUSIONS

This paper studies truss-based community search over large directed graphs. First, we devise a new community model called D-truss for community search over directed graphs. With the D-truss model, we formally define the problem of D-truss community search over directed graphs. Given a directed graph G and a query vertex set Q, the problem is to find the D-truss containing Q with the minimum diameter. As this problem is NP-hard, we propose two efficient polynomial algorithms with 2-approximation guarantee. An indexing method based on the D-truss decomposition results is further developed to expedite the algorithms. Extensive experimental results on large real-world networks with ground-truth communities confirm the effectiveness and efficiency of our proposed model and algorithms.

Acknowledgments. This work is supported by Research Grants Council of Hong Kong under GRF Projects 12200819, 12200917, CRF Project C6030-18GF, the NSFC under Grant No. 61702435 & 61972338, the National Key R&D Program of China under Grant No. 2018YFB1004003, the NSFC-Zhejiang Joint Fund under Grant No. U1609217, and the ZJU-Hikvision Joint Project.

REFERENCES

[1] Esra Akbas and Peixiang Zhao. 2017. Truss-based Community Search: a Truss-equivalence Based Indexing Approach. *PVLDB* 10, 11 (2017), 1298–1309.

[2] Jørgen Bang-Jensen and Gregory Z Gutin. 2008. *Digraphs: theory, algorithms and applications*. Springer Science & Business Media.

[3] Nicola Barbieri, Francesco Bonchi, Edoardo Galimberti, and Francesco Gullo. 2015. Efficient and effective community search. *Data Min. Knowl. Discov.* 29, 5 (2015), 1406–1433.

[4] Lu Chen, Chengfei Liu, Rui Zhou, Jianxin Li, Xiaochun Yang, and Bin Wang. 2018. Maximum Co-located Community Search in Large Scale Social Networks. *PVLDB* 11, 10 (2018), 1233–1246.

[5] Fan Chung. 2006. The diameter and Laplacian eigenvalues of directed graphs. *The Electronic Journal of Combinatorics* 13, 1 (2006), 4.

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

[7] Wanyun Cui, Yanghua Xiao, Haixun Wang, Yiqi Lu, and Wei Wang. 2013. Online search of overlapping communities. In *SIGMOD*. 277–288.

[8] Wanyun Cui, Yanghua Xiao, Haixun Wang, and Wei Wang. 2014. Local search of communities in large graphs. In *SIGMOD*. 991–1002.

[9] Yixiang Fang, Reynold Cheng, Yankai Chen, Siqiang Luo, and Jiafeng Hu. 2017. Effective and efficient attributed community search. *VLDB J.* 26, 6 (2017), 803–828.

[10] Yixiang Fang, Reynold Cheng, Xiaodong Li, Siqiang Luo, and Jiafeng Hu. 2017. Effective Community Search over Large Spatial Graphs. *PVLDB* 10, 6 (2017), 709–720.

[11] Yixiang Fang, Xin Huang, Lu Qin, Ying Zhang, Wenjie Zhang, Reynold Cheng, and Xuemin Lin. 2020. A survey of community search over big graphs. *VLDB J.* 29, 1 (2020), 353–392.

[12] Yixiang Fang, Zhongran Wang, Reynold Cheng, Hongzhi Wang, and Jiafeng Hu. 2019. Effective and Efficient Community Search Over Large Directed Graphs. *IEEE Trans. Knowl. Data Eng.* 31, 11 (2019), 2093–2107.

[13] Christos Giatsidis, Dimitrios M. Thilikos, and Michalis Vazirgiannis. 2011. D-cores: Measuring Collaboration of Directed Graphs Based on Degeneracy. In *ICDM*. 201–210.

[14] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-truss community in large and dynamic graphs. In *SIGMOD*. 1311–1322.

[15] Xin Huang, Laks VS Lakshmanan, and Jianliang Xu. 2019. *Community Search over Big Graphs*. Morgan & Claypool Publishers.

[16] Xin Huang and Laks V. S. Lakshmanan. 2017. Attribute-Driven Community Search. *PVLDB* 10, 9 (2017), 949–960.

[17] Xin Huang, Laks V. S. Lakshmanan, and Jianliang Xu. 2017. Community Search over Big Graphs: Models, Algorithms, and Opportunities. In *ICDE*. 1451–1454.

[18] Xin Huang, Laks V. S. Lakshmanan, Jeffrey Xu Yu, and Hong Cheng. 2015. Approximate Closest Community Search in Networks. *PVLDB* 9, 4 (2015), 276–287.

[19] Youngdo Kim, Seung-Woo Son, and Hawoong Jeong. 2010. Finding communities in directed networks. *Physical Review E* 81, 1 (2010), 016103.

[20] Jon M. Kleinberg. 1999. Authoritative Sources in a Hyperlinked Environment. *J. ACM* 46, 5 (1999), 604–632.

[21] Elizabeth A Leicht and Mark EJ Newman. 2008. Community structure in directed networks. *Physical review letters* 100, 11 (2008), 118703.

[22] Jianxin Li, Xinjue Wang, Ke Deng, Xiaochun Yang, Timos Sellis, and Jeffrey Xu Yu. 2017. Most Influential Community Search over Large Social Networks. In *ICDE*. 871–882.

[23] Rong-Hua Li, Lu Qin, Fanghua Ye, Jeffrey Xu Yu, Xiaokui Xiao, Nong Xiao, and Zibin Zheng. 2018. Skyline Community Search in Multi-valued Networks. In *SIGMOD*. 457–472.

[24] Rong-Hua Li, Lu Qin, Jeffrey Xu Yu, and Rui Mao. 2015. Influential Community Search in Large Networks. *PVLDB* 8, 5 (2015), 509–520.

[25] Rong-Hua Li, Jiao Su, Lu Qin, Jeffrey Xu Yu, and Qiangqiang Dai. 2018. Persistent Community Search in Temporal Networks. In *ICDE*. 797–808.

[26] Fragkiskos D Malliaros and Michalis Vazirgiannis. 2013. Clustering and community detection in directed networks: A survey. *Physics Reports* 533, 4 (2013), 95–142.

[27] Marina Meila and William Pentney. 2007. Clustering by weighted cuts in directed graphs. In *SDM*. 135–144.

[28] Mauro Sozio and Aristides Gionis. 2010. The community-search problem and how to plan a successful cocktail party. In *SIGKDD*. 939–948.

[29] Taro Takaguchi and Yuichi Yoshida. 2016. Cycle and flow trusses in directed networks. *CoRR* abs/1603.03519 (2016). http://arxiv.org/abs/1603.03519

[30] Charalampos E. Tsourakakis, Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Maria A. Tsiarli. 2013. Denser than the densest subgraph: extracting optimal quasi-cliques with quality guarantees. In *SIGKDD*. 104–112.

[31] Yubao Wu, Ruoming Jin, Jing Li, and Xiang Zhang. 2015. Robust Local Community Detection: On Free Rider Effect and Its Elimination. *PVLDB* 8, 7 (2015), 798–809.

[32] Jaewon Yang, Julian J. McAuley, and Jure Leskovec. 2014. Detecting cohesive and 2-mode communities indirected and undirected networks. In *WSDM*. 323–332.

[33] Tianbao Yang, Yun Chi, Shenghuo Zhu, Yihong Gong, and Rong Jin. 2010. Directed Network Community Detection: A Popularity and Productivity Link Model. In *SDM*. 742–753.

[34] Long Yuan, Lu Qin, Wenjie Zhang, Lijun Chang, and Jianye Yang. 2018. Index-Based Densest Clique Percolation Community Search in Networks. *IEEE Trans. Knowl. Data Eng.* 30, 5 (2018), 922–935.

[35] Qijun Zhu, Haibo Hu, Cheng Xu, Jianliang Xu, and Wang-Chien Lee. 2017. Geo-social group queries with minimum acquaintance constraints. *VLDB J.* 26, 5 (2017), 709–727.