Efficient Community Search in Edge-Attributed Graphs

Ling Li, Yuhai Zhao, Siqiang Luo, Guoren Wang, Zhengkui Wang

Abstract—Given a graph, searching for a community containing a query vertex is a fundamental problem and has found many applications. Most existing community search models are based on non-attributed or vertex-attributed graphs. In many real-world graphs, however, the edges carry the richest information to describe the interactions between vertices; hence, it is important to take the information into account in community search. In this paper, we conduct a pioneer study on the community search on edge-attributed graphs. We proposed the Edge-Attributed Community Search (EACS) problem, which aims to extract a subgraph that contains the given query vertex while its edges have the maximum attribute similarity. We prove that the EACS problem is NP-hard and propose both exact and 2-approximation algorithms to address EACS. Our exact algorithms run up to 2320.34 times faster than the baseline solution. Our approximate algorithms further improve the efficiency by up to 2.93 times. We conducted extensive experiments to demonstrate the efficiency and effectiveness of our algorithms.

Index Terms—Community Search, Edge-Attributed Graphs.

1 Introduction

Vertex attributed graphs [1], [2], [3] or Edge attributed graphs [4], [5], [6], which associate rich text information in their vertices or edges, have been widely adopted in many fields to capture the various types of data relationships. Examples include the co-authorship networks, protein-protein networks, communication networks, and so on [7], [8], [9].

A fundamental topic of attributed graphs is to search a structurally cohesive and attribute-aware community that contains a given query vertex. In the studies [4], [5], [7], [10], [11], [12], [13], the attribute-aware communities are categorized into *vertex-attributed community* and *edge-attributed community*, and edge attributes are observed more prevalent in social and other real-world networks than vertex attributes. Moreover, the edge attributes provide information on network communities' behaviors, i.e., how and why entities are related in a network, while the vertex attributes do not [4], [13]. Thus, the edge-attributed community adds value to a network by carrying more information than the vertex-attributed community. For example, in Figure 1, if the topic of exchanged text messages is considered as the edge attribute, we can clearly see three edge-attributed

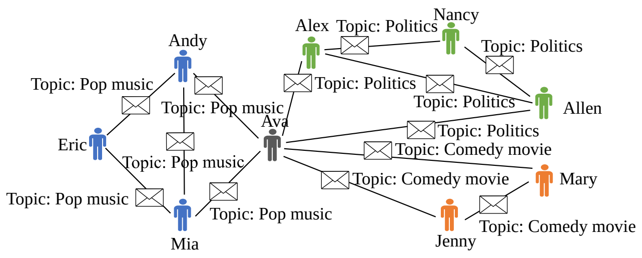


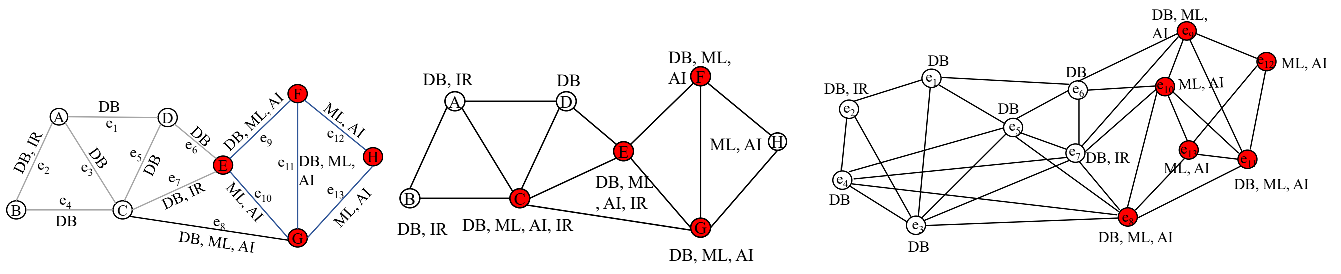
Fig. 1: A social media network with edge attributes

communities of different topics on pop music, politics and comedy movies. It is particularly interesting to note that Ava has edges involved in all the three communities. As described in [4], [13], such overlapping vertices like Ava reflect the fact that a given individual may have different facets to his/her life, which are revealed only in his/her interactions, i.e., edges, with different people. This suggests that rather than the vertex attributes, it may sometimes that the edge attributes provide unparalleled insights which can be leveraged to create interesting communities.

To our surprise, most existing works focus on the vertex-attributed community search but little effort has been paid to the edge-attributed community search. One intuitive solution to address the edge-attributed community search is to move the edge attributes to vertices and apply the existing vertex-attributed algorithms. However, such a naive method may fail in finding qualified edge-attributed communities. For example, in the co-authorship network shown in Figure 2(a), if two authors co-published a paper, there is an edge between them and the topics of this paper are the edge attributes. If the attributes at each edge are shifted to the two end vertices, the graph is transformed to that in Figure 2(b). In the original graph (Figure 2(a)), $\{E, F, G, H\}$ forms a community because all the edges between the vertices share $\{ML, AI\}$ and are cohesively connected. Note that vertex C is excluded from the community because edge (C, E) contains attributes that are significantly different.

- *Ling Li is with the School of Computer Science and Engineering, Northeastern University, China. E-mail:lilingneu@gmail.com.*
- *Yuhai Zhao is with the School of Computer Science and Engineering, Northeastern University, China. E-mail:zhaoyuhai@mail.neu.edu.cn.*
- *Siqiang Luo is with the School of Computer Science and Engineering, Nanyang Technological University, Singapore. E-mail:siqiang.luo@ntu.edu.sg.*
- *Guoren Wang is with the School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China. E-mail: wanggr@bit.edu.cn*
- *Zhengkui Wang is with the InfoComm Technology Cluster, Singapore Institute of Technology, Singapore. E-mail: zhengkui.wang@singaporetech.edu.sg*
- *Corresponding author: Yuhai Zhao.*

Manuscript received April 19, 2005; revised August 26, 2015.



(a) Edge-attributed network (b) Vertex-attributed network (1) (c) Vertex-attributed network (2)

Fig. 2: Illustrating the ineffectiveness of converting edge attributes to vertex attributes.

While (C, G) has similar edge attributes, C is not cohesively connected with $\{E, F, G, H\}$ if (C, G) is added to the community alone. In the transformed graph (Figure 2(b)), however, a different community $\{C, E, F, G\}$ is formed because their vertex attributes all share $\{DB, ML, AI\}$. This effect is due to the lost of edge-vertex attribute mappings. In other words, when given the transformed graph, we cannot recover the original attributed graph because it is unclear how to distribute the vertex attributes to the incident edges. Except for the simple transforming, the edge-attributed network can also be transformed to a vertex attributes network by exploring the line graph [14]. In particular, we see each edge as a vertex. If two edges in the original graph are adjacent, an edge is connected between the corresponding vertices in the line graph. Figure 2(c) is the transformed line graph. Unfortunately, this transformation also leads to information loss and the best community will not be found. For example, in Figure 2(c), the vertices in red color share similar attributes and are cohesively connected. These vertices correspond to $\{e_8, e_9, e_{10}, e_{11}, e_{12}, e_{13}\}$ and form a community $\{C, E, F, G, H\}$ in the original graph. However, the vertex C should be in another community. Another downside of using the line graph is that the graph size is significantly increased, which can be up to $O(n^3)$, where n is the number of vertices in the original graph.

The edge-attributed community search lacks tailored algorithms. To fill this gap, we investigate an attribute-aware community search problem that focuses on *edge attributes* and present the Edge-Attributed Community Search (EACS) problem. EACS extracts a subgraph that is densely connected while all its associated edge attributes have the maximum similarity. In particular, EACS employs the classic k-truss model to capture the structural cohesiveness and controls the homogeneity of edge attributes by minimizing the maximum dissimilarity between a pair of edges within the community. Our experiments in Section 5 show that our community search model can effectively and efficiently extract high-quality communities.

TABLE 1: Differences from existing studies.

Graph Type	**Community Search**	
	with user text input	w/o user text input
Non-attributed	-	[15], [16], [17], [18], [19]
Vertex-attributed	[1], [2], [3], [20]	[21], [22], [23]
Edge-attributed	[5]	**EACS (This paper)**

EACS is different from the existing studies about communities in the graphs. Existing studies mainly fall into two categories: community detection [4], [7], [10], [11], [12], [24], [25], [26] and community search [1], [2], [3], [15], [16], [17], [18], [19], [20], [21], [22], [23]. Community detection aims to find *all* communities in a network. In contrast, the community search problems, including EACS, are different as they find *one* densely connected subgraph that contains the given query vertex. Community search is preferred when the results need to be customized to the user's search and it enjoys much higher computation efficiency. Among the works in community search, only [5] has considered the edge attributes (see Table 1). Unlike our studies, [5] requires users to input keywords to guide the community search. As [21] mentioned, obtaining appropriate query keywords can be non-trivial for users. Therefore, being edge-attribute-centric and text-input free, EACS is different from all the existing studies.

Challenges and contributions. We prove that the proposed EACS problem is NP-hard. Compared with the vertex-attributed community search, the edge-attributed community search is usually more complex since the number of edges is often larger than the number of nodes in real word graphs. When focusing on the edges, the complexity is generally high. Thus, how to develop an edge-attributed community search algorithm with high effectiveness and efficiency is a challenge. A straightforward solution is to enumerate all possible subgraphs that satisfy the minimum truss value k. However, the number of enumerated subgraphs could reach O(2^m), where m is the number of edges. The method is therefore prohibitive.

To efficiently compute EACS, we develop several novel techniques. First, we present an index structure that is based on a new concept called (k, d)-truss. The index structure can effectively reduce the search space. Second, during the search, we develop one candidate reduction rule, two early-search-termination techniques, and one edge selection rule to reduce the number of search branches. Finally, we develop two efficient polynomial 2-approximation algorithms. The main contributions of our work are summarized as follows.

- We propose a novel edge-attributed community search (EACS) problem. We prove that the EACS problem is NP-hard.
- We develop an efficient exact algorithm for the EACS problem with novel techniques, including preprocessing, candidate reduction, early termination, and edges selection. The algorithm produces high-quality communities and runs 2320.34 times faster than the baseline approach.
- We also develop two novel polynomial approximation algorithms on top of the exact algorithm we

TABLE 2: Frequently used notations

Notation	Meaning
G	an undirected edge-attributed graph
$N_G(u)$	the set of neighbors of u in G
$deg_G(u)$	the degree of u in G
$sup_H(u, v)$	the number of triangles in H containing (u, v)
$sup^d_H(u, v)$	the number of d-triangles in H containing (u, v)
$\tau(e)$	the trussness of e in G
$d^*_k(e)$	the score trussness of e in G
$d(e_i, e_j)$	the edge dissimilarity score between two edges
$D(H)$	the maximum edge dissimilarity score among all edges in H
$\bar{d}$	the upper bound of the edge dissimilarity score
$\underline{d}$ and $\underline{d}'$	two lower bound of the edge dissimilarity score

proposed. We show that the algorithms achieve a 2-approximation regarding the metric score defined in EACS. The approximate algorithms can run 2.93 times faster than the advanced exact algorithm.

- We conduct extensive experiments on real networks. The results demonstrate the efficiency and effectiveness of the proposed algorithms.

Outline. We formulate the EACS problem in Section 2. In Section 3, a basic exact method is proposed. In Section 4, we present the advanced exact algorithm and approximation algorithms. We show experimental results in Section 5. We review the related works in Section 6 and conclude the paper in Section 7.

2 EACS

In this section, we first introduce the concepts and models for the Edge-Attributed Community Search (EACS). Then, we show the hardness of EACS.

2.1 Concepts and Models

We focus on an undirected G = (V_G, E_G), where V_G is the set of vertices and E_G is the set of edges. The set of attributes associated with each edge $e{\in}E_G$ is denoted as $attr_e$. Let n = $|V_G|$ and m = $|E_G|$ be the numbers of vertices and edges, respectively. Given vertex $u{\in}V_G$, the neighbor set of u is $N_G(u)$=$\{v|(u, v){\in}E_G\}$ and the degree of u is $deg_G(u)$=$|N_G(u)|$. An edge induced subgraph H=(V_H, E_H) of G satisfies $E_H{\subseteq}E_G$, and for every edge $(u, v){\in}E_H$ we have $u, v{\in}V_H$. A triangle induced on vertices $u, v, w \in V_G$ is denoted as Δ_{uvw}.

We measure the structural cohesiveness of a community using the connected k-truss. Given a graph G and an integer k, the connected k-truss of G is the connected subgraph H of G such that for each $(u, v){\in}E_H$, $sup_H(u, v) \geq k-2$, where $sup_H(u, v)$=$|\{w|w{\in}V_H, \Delta_{uvw}{\in}H\}|$. The *trussness* of an edge $e{\in}E_G$, denoted by $\tau(e)$, is the largest k such that a k-truss contains e. We choose the k truss-based model instead of the k-core-based model [17] because a k-truss is an edge-induced subgraph and it can be more naturally integrated into the edge-attributed community search.

In the following, we will introduce how to capture similar actions among cohesively interaction vertices. Note that most community search approaches focus on finding vertex-induced communities by exploring the links and vertex attributes, while we are interested in edge-induced communities with similar edge attributes. Hence, we need to define a function to capture similar edges with information encoded. In this paper, we mainly focus on the edge attributes that are in form of the text document. Similar definitions can be used for other formats.

Definition 1. Edge Dissimilarity Score d**.** Given two edges e_i and e_j, the edge dissimilarity score is $d(e_i, e_j)$=1-$\frac{|attr_{e_i} \cap attr_{e_j}|}{|attr_{e_i} \cup attr_{e_j}|}$, where $attr_{e_i}{\cap}attr_{e_j}$ is the set of common words from both $attr_{e_i}$ and $attr_{e_j}$, and $attr_{e_i}{\cup}attr_{e_j}$ is the union of the words from both $attr_{e_i}$ and $attr_{e_j}$.

Definition 2. Graph Dissimilarity Score D**.** Given an undireacted graph G, the dissimilarity score of this graph is $D(G)$=max $_{e_i,e_j \in E_G} d(e_i, e_j)$.

Intuitively, an interesting community should be densely connected, and all the edges carry sufficiently similar information. Hence, we define the following community search problem concerning the two factors.

Definition 3 (EACS). Given a graph G, a query vertex q, and a parameter k, the edge-attributed community search (EACS) aims to find the maximum subgraph H of G such that it satisfies the following conditions:

1) *k-truss constraint*. H is a connected k-truss and contains q.
2) *Edge-attributed constraint*. $D(H)$ is minimized among all subgraphs of G satisfying the above truss constraint.

In the problem definition, the k-truss constraint ensures that the subgraph is densely connected. The edge-attributed constraint guarantees that every pair of edges is sufficiently similar. We call a subgraph of H that satisfies all two conditions an optimal solution. A connected k-truss subgraph containing q is denoted as a possible subgraph, which means the optimal solution may be contained in this graph. Note that, the optimal solutions are not unique. So we aim to find the maximum subgraph which has the maximum number of edges.

Example 1. Consider the edge-attributed graph shown in Figure 2(a). Given q=E, k = 3, vertices E, F, G, H compose a connected 3-truss and the subgraph induced by them has the minimum graph edge dissimilarity score. According to the problem definition, they composed an edge-attributed community. However, using the existing vertex attributes solution [21] on two transformed vertex attributes graphs, we will obtain {C, E, F, G} or {C, E, F, G, H}. The first one misses vertex H and includes C while the other one includes C. We can see that the existing vertex attributes solution can not capture edge attributes cohesiveness well on transformed vertex attributes networks.

2.2 Hardness of EACS

Theorem 1. The EACS problem is NP-hard for $k{\geq}4$.

Proof: We prove that the decision version of the EACS problem is NP-hard by reducing an instance of the k'-clique problem that is NP-hard. The decision EACS problem is as follows: given a graph $G(V, E)$, a query vertex q, a parameter k and a score d, determining whether G has a subgraph H such that H is a k-truss containing q and $D(H){\leq}d$.

Given an instance of the k'-clique problem with $G(V,E)$ and an integer k', which aims to determine whether G has a clique with k' vertices. We reduce it into a decision EACS problem as follows. We add a dummy vertex q, and an edge between q and every vertex of V. Then the resulting graph G'=(V', E'), where V'=$V\cup\{q\}$, E'=$E\cup\{(q,u)|u\in V\}$. For each 4-clique C_i^4 in G', we add C_i^4 as edge attributes for each edge $e\in C_i^4$. The query of decision EACS problem is as follows: given query vertex q, the parameter k and d=1-$\frac{1}{\binom{|E'|}{6}}$. For all $(k-2)$-cliques C_i^{k-2} in G', we have $D(C_i^{k-2})\leq d$, which means the $(k-2)$-clique which contain q is the solution of decision EACS problem and $V(C_i^{k-2})\backslash q$ is a is a (k-3)-clique in G. For a solution H in G', due to $D(H)\leq$1-$\frac{1}{\binom{|E'|}{6}}$, we have each pair of edges (e_i, e_j) from $E(H)$ is at least in a same 4-clique, which means H is a clique in G'. Otherwise, there must be a vertex u and a vertex v in H, and vertex u and vertex v have no edge. Then the edges composed of the neighbors of u and u and the edges composed of the neighbors of v and v can not be in at least a same 4-clique, which means existing at least two edges, i.e., e_i and e_j, the $d(e_i,e_j)$=1-0>1-$\frac{1}{\binom{|E'|}{6}}$. Then we have $D(H)$>d=1-$\frac{1}{\binom{|E'|}{6}}$. It is a contradiction.

It is concluded that $C\subseteq V$ is a clique in G if and only if $C\cup q$ is a solution to the decision EACS problem in G'. Since finding cliques in a graph is NP-hard and the reduction process can be done in polynomial time, the decision EACS problem is NP-hard. □

3 Basic solution

We first present a basic solution (BasExact) to EACS. The main idea is based on the observation that a solution to EACS must be contained in a connected k-truss. Therefore, BasExact enumerates the maximal connected k-truss containing q as the candidate supergraph of the final solution, and then *refines* it by *shrinking the supergraph* to get the final solution with the minimum graph dissimilarity score.

The pseudo-code is given in Algorithm 1. It first computes the maximal connected k-truss containing q by the existing k-truss computation algorithm [27](Line 1). We denote the k-truss by H_k and the solution for EACS must be contained in H_k because of the truss cohesiveness constraint of EACS. In the meanwhile, as for the query vertex q, there must be one edge that is adjacent to q being contained in the solution to EACS. Therefore, Algorithm 1 explores all edges that are adjacent to q in H_k to get the initial partial solution $M\cup C$, where M maintains the chosen edges for the solution and C maintains the candidate edges (Lines 3-6) of the solution. The H_k is separated into M and $C(=E(H_k)\backslash M)$ to be further refined via the *NaiveShrink* procedure. In the while loop (Lines 7-13), Algorithm 1 iteratively shrinks the subgraphs and updates the current best solution until no subgraphs can be further shrunk. As M must be contained in the current partial solution, if D(M) is already larger than the current best score d^*, no better solutions exist in $M\cup C$. Also, if D(M) is equal to d^* but the number of edges of $M\cup C$ is smaller than that of the current best subgraph *opt*, there is no better solution in $M\cup C$ either (Lines 9-10). Hence, we can safely skip checking the subgraphs in $M\cup C$ in these two cases. In other cases, Algorithm 1 updates the current optimal solution *opt* and d^* (Lines 11-12) and searches for a solution within $M\cup C$ by invoking the procedure *NaiveShrink* (Line 13). The procedure *NaiveShrink* (Line 15) enumerates a possible k-truss with a smaller graph edge dissimilarity score. To enumerate all possible subgraphs, *NaiveShrink* partitions the search space into ($M\cup e$, $C\backslash e$) and (M, $C\backslash e$) for a randomly selected edge (Lines 17-21), where the former includes e in the subgraph, and the latter excludes e from the subgraph. Note that when removing an edge from C, the possible subgraph must contain M and is a connected k-truss (Line 19). In the while loop of Algorithm 1, the whole process corresponds to a binary enumeration tree.

Algorithm 1: BasExact

```
Input: Graph $G$, a query vertex $q$, parameter $k$.
Output: A $k$-truss $opt$ with minimum graph
        dissimilairy score.
1  Let $H_k$ be the connected $k$-truss of $G$ containing $q$;
2  $queue$={($M$=∅, $C$=∅)}, $d^*$=1, $opt$=∅;
3  for each $e_i$ adjacent to $q$ and $e_i\in E(H_k)$ do
4      $M\leftarrow e_i$;
5      $C\leftarrow E(H_k)\backslash e_i$;
6      $queue$.insert(($M$, $C$));
7  while $queue\neq\emptyset$ do
8      ($M$, $C$)=$queue$.pop();
9      if D($M$)>$d^*$ or D($M$)=$d^*$ ∧ $|M\cup C|\leq|opt|$ then
10         continue;
11     else
12         update $opt \leftarrow M\cup C$, $d^* \leftarrow$D($M\cup C$);
13     NaiveShrink($M$, $C$, $queue$);
14 return $opt$;
15 Procedure NaiveShrink($M$, $C$, $queue$)
16     if $C\neq\emptyset$ then
17         $e\leftarrow$select an edge from $C$;
18         $queue$.insert(($M\cup e$, $C\backslash e$));
19         $H\leftarrow$get connected $k$-truss containing $M$ from
           $M\cup C\backslash e$;
20         if $H\neq\emptyset$ and $q\subseteq V(H)$ then
21             $queue$.insert(($M$, $E(H)\backslash M$));
```

Complexity Analysis. The main part of Algorithm 1 is to enumerate all k-truss containing query vertices, which costs O($2^{|E(H_k)|}$) in the worst case. Another part is to maintain connected k-truss in the *NaiveShrink* search procedure when an edge is removed (Line 13). This can be solved in O($\sum_{(u,v)\in M\cup C}\min\{deg_{M\cup C}(u),deg_{M\cup C}(v)\}$)=O($|M\cup C|^{1.5}$). In the while loop, Algorithm 1 computes the edge dissimilarity score for each graph $M\cup C$, and it costs O($|M\cup C|^2$) time (Line 14). As $|M\cup C|$ is smaller than $|E(H_k)|$, the complexity of Algorithm 1 is O($2^{|E(H_k)|}(|E(H_k)^2|)$).

Remarks. The cost of Algorithm 1 can be high in processing large graphs, and there is still room for improving the efficiency. First of all, Algorithm 1 does not adopt any preprocessing approach to efficiently reduce the search space, which results in generating many redundant intermediate subgraphs. Secondly, in the search process, *NaiveShrink* does

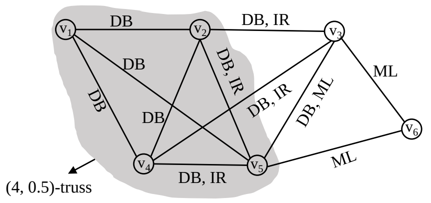


Fig. 3: An example of (k, d)-truss.

not adopt any candidate set pruning rules. However, given the current optimal d^*, many edges of C will not be part of the optimal solution. Also, a simple lower bound for D(M) is used, which can not terminate the unpromising search earlier. Finally, in the *NaiveShrink* search procedure, some subgraphs can be searched multiple times. The reason is that, in each enumeration, *NaiveShrink* partitions the search space into ($M\cup e$, $C\setminus e$) and (M, $C\setminus e$). The subgraph generated by $M\cup e$ and $C\setminus e$ is the same as the subgraph generated by M and C. This subgraph will be computed about the dissimilarity score again. Hence, in the next section, we present more advanced solutions for EACS.

4 Advanced Solutions

In this section, we present advanced solutions for EACS. We first introduce a novel concept (k, d)-truss, which is tailored to our advanced solutions (see Section 4.1). Then, we present an advanced exact solution (AdvExact) equipped with a number of interesting techniques based on the (k, d)-truss (see Section 4.2). Finally, based on the advanced exact solutions, we propose two 2-approximation algorithms to further speed up the computation of EACS (see Section 4.3).

4.1 (k, d)-truss

The (k, d)-truss is an extension of the k-truss by integrating the *triangle edge support* d. Given a triangle Δ_{uvw} of an undirected edge-attributed graph $G(V, E)$ and a parameter d, if D(Δ_{uvw})$\leq d$, Δ_{uvw} is called a d-Triangle (Δ^d_{uvw}) of G. We first give the definition of d-Triangle edge support.

Definition 4. d-Triangle Edge Support. Given a parameter d, the d-Triangle edge support of an edge (u, v) in G, denoted by $Sup^d_G(u, v)$, is the number of d-Triangles in G containing (u, v), i.e., $Sup^d_G(u, v)$=$|\{w|\ \Delta^d_{uvw}\in G\}|$.

Based on the definition of the d-Triangle edge support, we define (k, d)-truss as follows.

Definition 5. (k, d)-truss. Given an undirected edge-attributed graph G, a score d and an integer $k\geq 3$, a subgraph H is a (k, d)-truss of G if $\forall$ $(u, v)\in E(H)$, $Sup^d_H(u, v)\geq k$-2.

A (k, d)-truss H is *maximal* if there does not exist a (k, d)-truss $H'\subseteq G$ such that $H\subset H'$.

Example 2. Figure 3 shows an example of an edge-attributed graph. The edge (v_1, v_4) forms 0-Triangle with vertex v_2 and forms 0.5-Triangle with vertex v_5. Hence, $Sup^{0.5}_G(v_1, v_2)$=2. Since edges (v_1, v_2), (v_1, v_4), (v_1, v_5), (v_2, v_4), (v_2, v_5), (v_4, v_5) are all contained by at least two 0.5-Triangles, they compose of a $(4, 0.5)$-truss. Note that this subgraph is also a maximal $(4, 0.5)$-truss.

Lemma 1. The maximal connected (k, d)-truss of G is unique.

Proof: We assume by contradiction that there exist two different maximal connected (k, d)-trusses, H_1 and H_2, $E(H_1)\neq E(H_2)$ and $E(H_1)\cap E(H_2)\neq\emptyset$. As $E(H_1)\cap E(H_2)\neq\emptyset$, they can form a larger connected (k, d)-truss $H_1 \cup H_2$. The contradiction ensues. □

To obtain the (k, d)-truss, we can iteratively remove the edges whose d-Triangle support is less than k-2. However, if the graph is large, many edges will need to be removed, leading to low efficiency. To quickly locate the maximum possible subgraph (k, d)-truss, we propose an index, called the KdTruss-Index. The main idea of this index is to maintain the (k, d)-truss for every possible k and d. Then, given any k and d, the index can support (k, d)-truss computation in optimal time. For ease of presenting the KdTruss-Index, we first give the following lemma and definition.

Lemma 2. Given an edge-attributed graph G, a maximal (k, d)-truss must be contained in a maximal (k', d')-truss if $k\geq k'$, $d\leq d'$.

Proof: We assume that the maximal (k, d)-truss and the maximal (k', d')-truss are H and H' respectively. $\forall e\in E(H)$, $Sup^d_G\geq k-2$. As $d'\geq d$, we have $Sup^{d'}_G\geq k-2$. So H is a (k, d')-truss. Due to $k'\leq k$, H must be a (k', d')-truss. As the maximal (k', d')-truss is unique, we have H must be a subgraph of H'. □

Based on Lemma 2, for an edge $e \in E$ and the minimum value d_{min}, if e is contained in a maximal (k, d_{min})-truss, e must belong to a maximal (k, d)-truss for $d\geq d_{min}$. Hence, we can construct an index by sorting all edges according to the minimum d value of edges for a given k. Next, we give the formal definition for this minimum d value, which is called *score trussness*.

Definition 6. Score Trussness. Given an edge-attributed graph G and a positive integer $k \geq 3$, $\forall$ $e\in E$, the score trussness of e, denoted by $d^*_k(e)$, is the minimum d such that e is contained in a (k, d)-truss.

The KdTruss-Index can be regarded as a list $\mathcal{L}$, where $\mathcal{L}[k]$ stores the sorted edges in descending order of the score trussness of each edge in k-truss. Based on this structure, we can quickly locate the (k, d)-truss. In the following, we introduce the construction of this index.

Note that if an edge only has k-2 d-Triangles in the k-truss, its score trussness must be at most d. Hence, the main idea of building KdTruss-Index is to iteratively peel an edge with the maximum d such that $Sup^d_k(e)$=k-2 in the k-truss. The pseudo-code of building KdTruss-Index is shown in Algorithm 2. It first performs the truss decomposition algorithm [27] on G and finds the maximum k_{max} such that there exists a k_{max}-truss (Line 1). Then in Lines 3-17, for each k from 3 to k_{max}, the algorithm iteratively processes $\mathcal{L}[k]$. In particular, it first computes the maximum d_{max} by exploring each edge e with $Sup^d_{H_k}(e)$=k (Line 6). Then, the algorithm iteratively removes the edges whose d_{max}-Triangle support is less than k-2 (Lines 7-17). Once an edge is removed from H_k, this edge and corresponding d_{max} is added into $\mathcal{L}[k]$ (Line 17). Note that in the while loop (Lines 10-17), when an edge (u, v) is removed, it needs to compute the d-Triangle support for edges that are composed of a triangle with (u, v), i.e., (u, w), (v, w) $\in\Delta_{u,v,w}$. To avoid recomputing the d-Triangles for corresponding edges, we

Algorithm 2: KdTruss-Index

```
Input: Graph $G$.
Output: The KdTruss-Index $\mathcal{L}$.
1  Perform truss decomposition on $G$ and get the maximum trussness $k_{max}$;
2  $queue \leftarrow \emptyset$;
3  for $k \leftarrow 3$ to $k_{max}$ do
4      $H_k \leftarrow k$-truss of $G$;
5      while $E(H_k) \neq \emptyset$ do
6          $d_{max} \leftarrow$ maximum $d$ that $Sup^{d}_{H_k}(e)=k$, $\forall e \in E(H_k)$;
7          for $e \in E(H_k)$ do
8              if $Sup^{d_{max}}_{H_k}(e) \leq k$-2 then
9                  $queue \leftarrow queue \cup \{e\}$;
10         while $queue \neq \emptyset$ do
11             Pop out an edge $(u, v)$ from $queue$;
12             for $w \in N_{H_k}(u) \cap N_{H_k}(v)$ do
13                 for $e \in \{(u, w), (v, w)\}$ do
14                     if $Sup^{d_{max}}_{H_k \setminus (u,v)}(e) \leq k$-2 then
15                         $queue \leftarrow queue \cup \{e\}$;
16             Remove $e$ from $H_k$;
17             $\mathcal{L}[k]$.add(($d_{max}$, $e$));
18 return $\mathcal{L}$;
```

can maintain an array $Array$ that keeps the score $D(\Delta_{u,v,w})$ in $Array[e]$ if e is in this triangle. When an edge e is required to update the d-Triangles support, we remove the corresponding score from $Array[e]$. Then, the updated d-Triangles support of the edge is the number of scores that is less than d in $Array[e]$. Since $Array$ keeps the edges that are not removed. If we have obtained (k, d_1)-truss, to compute the (k, d_2)-truss with $d_2 \leq d_1$, we only need to compute from (k, d_1)-truss.

Complexity Analysis. Algorithm 2 first performs the truss decomposition algorithm, which costs O($E(G)^{1.5}$) time. Then, it iteratively removes edges, where all triangles involving any of the removed edges should be enumerated. This step costs

$$O\left(\sum_{(u,v)\in E(H_k)} \min\{deg_{H_k}(u), deg_{H_k}(v)\}\right) = O\left(|E(H_k)|^{1.5}\right)$$

Overall, Algorithm 2 costs

$$O\left(E(G)^{1.5} + \sum_{k=3}^{k_{max}} |E(H_k)|^{1.5}\right)$$

Index-based (*k*, *d*)-truss Finding. To obtain the maximal connected (k, d)-truss, we can first determine the edge which first appears with score trussness of d in $\mathcal{L}[k]$. Then, all edges behind this edge in $\mathcal{L}[k]$ belong to the maximal (k, d)-truss. Therefore, we can find the maximal connected subgraph from these edges.

4.2 The Advanced Exact Solution

In this section, we introduce an advanced exact solution (AdvExact). The main idea is to prune the graph and reduce the search branches. The skeleton of the algorithm is shown in Algorithm 3. It first applies a (k, d)-truss based pruning

Algorithm 3: AdvExact

```
Input: Graph $G$, a query vertex $q$, parameter $k$.
Output: A $k$-truss opt with minimum graph dissimilairy score.
1  Conduct graph pruning based on $(k, d)$-truss;
2  Algorithm 1 Lines 2-13 and replace NaiveShrink by AdvancedShrink (see Algorithm 7);
3  return opt;
```

on the input graph (Line 1). Then, in the pruned graph, the procedures are the same as in Algorithm 1 (Line 2), except that the *NaiveShrink* is replaced by *AdvancedShrink*. We note that the (k, d)-truss based pruning and the new shrink procedure *AdvancedShrink* is where the novelty of Algorithm 3 exists.

The (*k*, *d*)-truss pruning (Algorithm 3 Line 1). Let d^* be the dissimilarity score of the optimal H^*, we have the following lemma.

Lemma 3. Given an undirected edge-attributed graph G and d^*, the optimal solution H^* must be contained in the maximal connected (k, d^*)-truss of G that contains q.

Proof: Based on the problem definition, the edge dissimilarity score of each pair of edges must be smaller than d^*, so each triangle in H^* must be a d^*-Triangle. As H^* is a k-truss, each edge is contained by at least k d^*-Triangles, indicating that H^* is a (k, d^*)-truss. Because the optimal solution H^* must be a connected subgraph containing q and the maximal connected (k, d^*)-truss containing q is unique, H^* must be contained in the maximal (k, d^*)-truss of G that contains q. □

Based on Lemma 3, a (k, d)-truss can be used to prune irrelevant edges before enumerating subgraphs. Although the (k, d^*)-truss-based pruning can be effective, it requires knowing d^*. However, finding the exact d^* is NP-hard as it in turn needs to compute the optimal solution of EACS. To address this challenge, we give an upper bound $\bar{d}$ of d^* and make use of the property that the maximal connected (k, d^*)-truss must be contained in the maximal $(k, \bar{d})$-truss (by Lemma 2). We further show that computing $\bar{d}$ can be finished in polynomial time. The key idea is to find a possible connected k-truss H that contains the query vertex q and has a small graph dissimilarity score $D(H)$. Then we treat $D(H)$ as the upper bound of d^*.

For this purpose, we propose an algorithm, GreedyUpperBound to find a possible k-truss that contains q. The pseudo-code is shown in Algorithm 4. Algorithm 4 regards the edges which are adjacent to q as a center. Then, the algorithm iteratively removes the edges that have the maximum edge dissimilarity scores with respect to the chosen e (Lines 5-12). In this step, once some edges are removed, it needs to maintain the connected k-truss (Line 9). In particular, we first iteratively remove the edges that have not been contained in k-2 triangles to obtain the k-truss and then discover the connected subgraphs that contain q. Then it updates the current best score d (Lines 10-11). Until $q \not\subseteq V(H)$, the while loop terminates. After all edges that are adjacent to q have been processed, it returns $\bar{d}$=$D(H')$ as the upper bound of d^*. The upper bound holds because based on Algorithm 3, $\bar{d}$ is always updated by the connected k-truss containing q.

Algorithm 4: GreedyUpperBound

```
Input: Graph $G$, integer $k$, a query vertex $q$.
Output: an upper bound $\bar{d}$.
1  Let $H_k$ be the $k$-truss of $G$ containing $q$;
2  $d \leftarrow 1$, $H' = \emptyset$;
3  for each $e_q$ is adjcent to $q$ do
4      $H = H_k$;
5      while $q \subseteq V(H)$ do
6          $d_{max} \leftarrow \arg\max_{e_i \in E(H)} d(e_i, e_q)$;
7          $S \leftarrow \{e_i | d(e_i, e_q) \geq d_{max}, e_i \in E(H)\}$;
8          Delete $S$ from $E(H)$;
9          $H'' \leftarrow$ connected $k$-truss containing $q$ induced
           by $E(H)$;
10         if $H'' \neq \emptyset$ and $d_{max} < d$ then
11             $d = d_{max}$; $H' = H''$;
12         $H = H''$;
13 $\bar{d} = D(H')$;
14 return $\bar{d}$;
```

Complexity Analysis. Algorithm 4 first computes a k-truss of G, which costs O($|E(G)|^{1.5}$) time. Then, for each edge adjacent to q, it iteratively deletes the edges with the largest edge dissimilarity score from H_k. In this process, computing the dissimilarity sore requires O($|E(H_k)|$) time, and maintaining a connected k-truss needs O($|E(H_k)|^{1.5}$) time. Assuming that l is the maximum number of iterations in the while loop, Algorithm 4 takes O($|E(G)|^{1.5}$+$deg(q)(l|E(H_k)|^{1.5})$) time.

Note that Algorithm 4 needs to process all edges that are adjacent to the query vertex q. However, there exists redundant computation. Let us consider Lines 4 in Algorithm 4. Let e_1 and e_2 be the edges that are adjacent to the query vertex q. If we first explore e_1 and obtain an upper bound $\bar{d}$. When processing e_2, we still need to iteratively remove edges from the k-truss H_k. Indeed, by exploring the obtained $\bar{d}$, we can process e_2 in a smaller subgraph. Next, we introduce a Lemma, which promises further pruning without affecting the final $\bar{d}$.

Lemma 4. Given a k-truss H_k containing q and $\bar{d}$, if there exists a smaller $\bar{d}'$, then the k-truss H_k' containing q with D(H_k')$\leq\bar{d}'$ must be contained in the maximal (k, $\bar{d}$)-truss.

As the (k, $\bar{d}$)-truss must be the subgraph of the maximal k-truss. By Lemma 4, in Line 4 of Algorithm 4, we can use KdTruss-Index to directly get the (k, $\bar{d}$)-truss instead of k-truss to refine the search space.

AdvancedShrink (Algorithm 3 Line 2). The (k,d)-truss based pruning is applied before search. Here, we introduce the *AdvancedShrink* procedure with several techniques that are used to efficiently generate candidate subgraphs that contain the final solution.

Observation 1. *a partial solution $M\cup C$, the current best score d that is the recorded optimal graph edge dissimilarity score till now, the optimal solutions with smaller scores must be contained in the connected k-truss induced by edges S, where S=$\{e_i|d(e_i, e_j)\leq d, \forall\ e_i\in C, e_j\in M\}$.*

By this observation, when certain edges are removed, the remaining graph should always be a connected k-truss while containing M. To satisfy this constraint, the disqualified edges will be further removed, giving us the following rule for generating smaller candidate subgraphs that could contain the final solution.

Candidate Reduction Rule. *Given a partial solution $M\cup C$ and the current best score d, we can remove all edges from $C\backslash S$ and any edge e from S if the trussness of e $\tau_{M\cup C\backslash S}(e)<k$-2.*

Note that once applying the candidate reduction rule on C, if C becomes empty or there exists no connected k-truss containing M, the search of the branch can be terminated immediately. Here, we introduce two lower bounds of d^*, which could be used to terminate the search early. Given a partial solution, if the current best score d^* is less than the lower bounds, we do not need to enumerate the possible subgraphs of this partial solution. The key idea of computing the lower bounds is to use the current partial solution $M\cup C$ to compute the subgraph k-truss H that contains M while D(H) is the smallest. However, directly computing the lower bound in this way is equal to finding the optimal solution in $M\cup C$, which is NP-hard. Instead, we find relaxed lower bounds. Let $d^*(M\cup C)$ be the optimal solution. We first introduce a lower bound by mainly considering the edges in M (Definition 7), followed by another lower bound considering the whole $M\cup C$ (Definition 8).

Definition 7. Lower Bound $\underline{d}$. Given partial solution M, the (k, d)-truss based lower bound of $d^*(M\cup C)$, denoted by $\underline{d}(M)$, is the smallest value d such that there is a (k, d)-truss containing M.

Lemma 5 shows the correctness of the lower bound $\underline{d}$.

Lemma 5. Given partial solution $M\cup C$ and integer k, $\underline{d}(M)\leq d^*(M\cup C)$.

Proof: Based on Lemma 3, M is contained in (k, $d^*(M\cup C)$)-truss. As $\underline{d}(M)$ is the smallest d such that (k, d)-truss contains M, we have $\underline{d}(M)\leq d^*(M\cup C)$. □

Note that the aforementioned lower bound only considers the edges in M. In certain cases, this lower bound may not be tight. Here, we design another lower bound by exploring both M and C. The computation of a lower bound could be converted to another problem, i.e., how to add edges into M from C to let M become a connected k-truss with the smallest dissimilarity score. Note that addressing this problem exactly can incur high complexity; hence, we use the greedy method to obtain a lower bound as follows.

Definition 8. Lower Bound $\underline{d}'$. Given partial solution $M\cup C$, the greedy based lower bound of $d^*(M\cup C)$, denoted by $\underline{d}'(M\cup C)$, is the smallest value d such that $M\cup\{e|$D$(M\cup e)\leq d$, $e\in C\}$ is composed a connected k-truss containing M.

Lemma 6. Given partial solution $M\cup C$ and integer k, $\underline{d}'(M\cup C)\leq d^*(M\cup C)$.

Proof: As $\underline{d}'(M\cup C)$ is the minimum d that M could be expanded to a connected k-truss, we have $\underline{d}'(M\cup C)\leq d^*(M\cup C)$. Otherwise, $d^*(M\cup C)$ will be the minimum d that M could be expanded to a connected k-truss. □

Lemma 7. Given partial solution $M\cup C$ and integer k, if $e\in M$ and $Sup_{M\cup C}(e)$ = $k-2$, $\{e'$ | $e'\in C$, s.t. $\exists e''\in M\cup C$, e', e'', and e compose a triangle$\}$ must exist in the optimal k-truss generated by current solution.

Algorithm 5: LowerBound

```
Input: The chosen edges set $M$, candidate $C$, integer $k$, KdTruss-Index $\mathcal{L}$.
Output: a lower bound $\underline{d}^*$.
1  $M'$=$M$∪{$e'$|$e'$∈$C$, $s.t.$∃$e$∈$M$, $e''$∈$C$, $e'$, $e''$, and $e$ compose a triangle}, $C'$=$C \setminus M'$;
2  $count$=0; $\underline{d}(M)$=0;
3  for $e$∈$\mathcal{L}[k]$ do
4      if $e$∈$M$ then
5          $count$++;
6      if $count$=$|M|$ then
7          $\underline{d}(M)$=$d_k^*(e)$;
8  $\underline{d}'(M'∪C')$=$\underline{d}(M)$, $S$←$M'$;
9  while True do
10     for $e$∈$C'$ do
11         if D($M'$∪{$e$})≤$\underline{d}'(M'∪C')$ then
12             $S$=$S$∪{$e$};
13     if $S$ composed a connected $k$-truss containing $M'$ then
14         break;
15     else
16         $d$←arg $\min_{e∈C'\setminus S∧D(M'∪e)>\underline{d}'(M'∪C')}$ D($M'$∪$e$);
17         $\underline{d}'(M'∪C')$ = $d$;
18 return $\max(\underline{d}(M), \underline{d}'(M'∪C'))$
```

Proof: As the edges in M must exist, to satisfy the k-truss constraint, if an edge e with $Sup_{M∪C}(e) = k-2$, the edges that with e compose triangles must exist. □

Based on Lemma 5, Lemma 6 and Lemma 7, we obtain our final lower bound $\underline{d}^*$ by considering the maximum value between $\underline{d}(M')$ and $\underline{d}'(M'∪C')$, where M'=M∪{e'|e'∈C, $s.t.$∃e∈M, e''∈C, e', e'', and e compose a triangle}, C'=$C \setminus M'$. We can efficiently obtain $\underline{d}(M')$ by determining edges in $\mathcal{L}[k]$ of the KdTruss-Index. To compute the lower bound $\underline{d}'(M'∪C')$, we need to increase $\underline{d}'(M'∪C')$ from a small value (e.g., 0) to a possible d_{max} so that M' and {e|D(M'∪e)≤d_{max}, e∈C'} can compose a connected k-truss. However, when the initial value is too small, it will lead to the redundant computation. Interestingly, we can use the $\underline{d}(M)$ to be the initial value of $\underline{d}'(M'∪C')$ instead of increasing d from 0. The pseudo-code of discovering the lower bound $\underline{d}^*$ is shown in Algorithm 5. It first explores the KdTruss-Index to find $\underline{d}(M')$ (Lines 2-7). As all edge in $\mathcal{L}[k]$ is sorted in descending order according to score trussness, it iteratively determines the edges in $\mathcal{L}[k]$ until all the edges in M' are found (Lines 3-7). Then the score trussness of the final found edge is $\underline{d}(M')$ (Line 7). Next, it evaluates $\underline{d}'(M'∪C')$ (Lines 8-17). It initializes $\underline{d}'(M'∪C')$ as $\underline{d}(M)$ (Line 8). Then in each iteration of the while loop, for each edge in C', it adds all edges whose edge dissimilarity scores (to M') are less than d (Lines 10-12). If the edges in S compose a connected k-truss containing M', the $\underline{d}'(M'∪C')$ is found (Lines 13-14). Otherwise, a larger value $\underline{d}'(M'∪C')$ is obtained to merge more edges into S (Lines 15-17).

Complexity Analysis. As Algorithm 5 needs to explore all edges in $\mathcal{L}[k]$ in the worst case (Lines 2-7), the time complexity is O($|\mathcal{L}[k]||M'|$). Let ℓ be the number of iterations in the while loop (Lines 8-17). As in each iteration, each edge is checked whether it could be added into S, which costs O($|C'|$). Additionally, verifying the connected k-truss costs O($|S|^{1.5}$). Because in the worst case, S=C', the overall time complexity of all iterations is O($\ell|C'|^{1.5}$). Thus, the total time complexity is O($|\mathcal{L}[k]||M'|$+$\ell|C'|^{1.5}$).

Proper Edge Selection. Note that if we could obtain a candidate subgraph with a small dissimilarity score early, then many branches that will not generate the candidate subgraph with a smaller dissimilarity score can be skipped early. The key idea of achieving this goal is how to select edges to partition the search space. Recall the *NaiveShrink* function in Algorithm 1, it randomly selects an edge e and partitions the search space into two parts, i.e., (M∪{e}, C\{e}) and (M, C\{e}). However, this partitioning approach will lead to redundant computation and will explore many irrelevant branches. We observe that if an edge e with the maximum D(M∪{e}) is added into M, the generated branches will be early terminated with high probability. Thus, we can select the edges with the maximum D(M∪{e}) to generate branches.

However, the above simple edge selection method still receptively generates the same subgraphs. Note that during the search, we always keep the reduced subgraph as a connected k-truss. Removing an edge from M∪C will lead to the removal of other edges and could get a candidate graph with a small dissimilarity score as fast as possible. Thus, we could enumerate candidate subgraphs by removing edges instead of generating a subgraph that is not changed, i.e. (M∪{e}, C\{e}). To achieve this goal, we first introduce the following definition and lemma.

Definition 9. Exclusion Set. Given partial solution M∪C and a score d, the exclusion set, denoted by S_d(M∪C), is the edge set s.t. ∀ e_i, e_j∈S_d(M∪C), we have $d(e_i, e_j)$>d and S_d(M∪C)⊆C.

Lemma 8. Given partial solution M∪C, the current best score *opt* and S_{opt}(M∪C), the optimal solution can only be from (M∪{e_i}, C\$S_{opt}$($M$∪$C$)) and ($M$, C\$S_{opt}$($M$∪$C$)), where e_i∈S_{opt}(M∪C).

Proof: During the search, given a partial solution M∪C, we always want to find a candidate subgraph with a smaller score compared to the current best score *opt*. As all pairs of edges in S_{opt}(M∪C) have scores that are larger than *opt*, we can safely remove other edges from S_{opt}(M∪C) once an edge from S_{opt}(M∪C) is chosen to join M. □

Based on Lemma 8, we could generate multi branches, and each subgraph induced by these branches is must smaller than the original partial solution M∪C. One issue is when D(M∪C)=*opt*, S_{opt}(M∪C) will be empty. To solve this problem, we propose another set S'_{opt}(M∪C) s.t. ∀ e_i, e_j∈S'_{opt}(M∪C), $d(e_i, e_j)$=*opt*, S'_{opt}(M∪C)⊆C.

Although by exploring the exclusion set, we could reach a smaller subgraph as soon as possible, obtaining the exclusion set is still a challenge. By definition, finding an exclusion set is equal to discovering a clique in a graph G_C based on C, where each edge in C is seen as a vertex in G_C and if $d(e_i, e_j)$>d, an edge exists in G_C. In addition, we also need to consider selecting the edge e with the maximum D(M∪e). To efficiently find a promising exclusion set, we

Algorithm 6: ExclusionSet

```
Input: Chosen edges $M$, candidate edges $C$, a score $d$.
Output: Exclusion set $S_d(M\cup C)$.
1 $S\leftarrow$Sort all edges of $C$ in descending order of D($M\cup\{e\}$), $e\in C$;
2 for $e\in S$ do
3     $S_d(M\cup C)=\{e\}$;
4     for $e_i\in S$ do
5         if $e_i\notin S_d(M\cup C)\wedge d(e_i,e_j)>d,\forall e_j\in S_d(M\cup C)$ then
6             $S_d(M\cup C)=S_d(M\cup C)\cup\{e_i\}$;
7     if $|S_d(M\cup C)|\neq 1$ then
8         return $S_d(M\cup C)$;
```

could first sort all edges of C in descending order. Next, we iteratively add an edge e with $d(e,e_i)>d$, $\forall e_i\in S_d(M\cup C)$.

The pseudo-code of finding a promising exclusion set is shown in Algorithm 6. It first sorts all edges (Line 1). Then it selects an edge from S to get the exclusion set by iteratively determining whether an edge could be added to the current exclusion set (Lines 2-8). If not added, it selects another edge to obtain the exclusion set. If the size of the exclusion set is not equal to 1, it terminates (Lines 7-8). Note that $S'_d(M\cup C)$ could be computed in a similar fashion; we omit the details due to space limitations.

Complexity Analysis. Algorithm 6 first sorts all edges in C, which needs O($|C|^2|M|$). Then for each edge in S, it costs O($|C||S_d(M\cup C)|$) time to add edges into $S_d(M\cup C)$. In the worst cases, Algorithm 6 will explore all edges in C. So the total time complexity is O($|C|^2|M|+|C|^2|S_d(M\cup C)|$).

Advanced Shrink Method. Equipped with all the techniques mentioned earlier, we develop an advanced shrink algorithm to enumerate all candidate subgraphs. The pseudo-code is shown in Algorithm 7. It first applies the candidate reduction rule (Line 1). If the candidate set C becomes empty, there will be no candidate subgraph enumerated (Lines 2-3). Note that if C is changed, it needs to maintain the k-truss connectivity (Line 5). If q does not belong to $V(H)$, this search branch can be terminated (Lines 6-7). Then, it updates the current optimal solution as M and new C may compose a new candidate subgraph (Lines 8-9). Next, we could use the lower bound $\underline{d}^*$ to determine whether this branch could be terminated (Lines 10-11). If one of the lower bounds is larger than the current best score opt, this branch is terminated. We can also terminate this branch when the lower bound $\underline{d}^*$ is equal to D($M\cup C$) because in this situation there will be no subgraph with a smaller score compared to the current partial solution. Then, we find the exclusion set to generate branches (Lines 12-16). As each pair of edges in S does not coexist in the subgraphs with a smaller score, we could directly remove S from C if $e\in S$ is added into M and generate branches (Lines 17-20). Finally, we remove S from C to generate a possible branch (Lines 21-23). Note that the number of branches produced by this algorithm can be more than 2. All such generated candidate subgraphs are shrunk at least one edge compared to the partial solution $M\cup C$. Thus, we could reach the optimal solution sooner.

Algorithm 7: AdvancedShrink

```
Input: Chosen edge $M$, candidate $C$, $queue$.
1 $C\leftarrow$candidate edge reduction;
2 if $C=\emptyset$ then
3     return;
4 if $C$ is changed then
5     $H\leftarrow$get connected $k$-truss containing $M$ from $M\cup C$;
6     if $q\not\subseteq V(H)$ then
7         return;
8     if D($H$)$<d^*$ or D($H$)$=d^*\wedge|E(H)|>|opt|$ then
9         update $opt=H$, $d^*$=D($H$);
10 if $\underline{d}^*>d^*$ or $\underline{d}^*$=D($M\cup C$) then
11     return;
12 $S\leftarrow S_{d*}(M\cup C)$;
13 if $S=\emptyset$ and $D(M\cup C)=d^*$ then
14     $S\leftarrow S'_{d^*}(M\cup C)$;
15 if $S=\emptyset$ then
16     $S\leftarrow$select an edge from $C$;
17 for $e\in S$ do
18     $H\leftarrow$get connected $k$-truss containing $M\cup\{e\}$ from $M\cup\{e\}\cup(C\backslash S)$;
19     if $H\neq\emptyset$ and $q\subseteq V(H)$ then
20         $queue$.insert($M\cup\{e\}$, $E(H)\backslash(M\cup\{e\})$);
21 $H\leftarrow$get connected $k$-truss containing $M$ from $M\cup(C\backslash S)$;
22 if $H\neq\emptyset$ and $q\subseteq V(H)$ then
23     $queue$.insert($M$, $E(H)\backslash M$);
```

Complexity Analysis. Algorithm 7 first finds the removable edges from the current solution, which needs O($|M\cup C|^{1.5}+|M||C|$) time. Then it computes the dissimilarity score to update the current best solution, which costs O($|M\cup C|^2$). Computing the lower bound needs O($|\mathcal{L}[k]||M|+\ell|C|^{1.5}$) time and computing the exclusion set needs O($|C|^2|M|+|C|^2|S_d(M\cup C)|$) time. Finally to obtain the branches, Algorithm 7 costs O(($|S|+1)|M\cup C|^{1.5}$)) time. Thus, the total time complexity of Algorithm 7 is O($|M\cup C|^2+|\mathcal{L}[k]||M'|+\ell|C'|^{1.5}+|C|^2|M|+|C|^2|S_d(M\cup C)|+(|S|+1)|M\cup C|^{1.5}$)).

Let T be the set of all branches that are enumerated in the Algorithm 3. Algorithm 3 costs $\sum_{(M_i,C_i)\in T}|M_i\cup C_i|^2+|\mathcal{L}[k]||M'_i|+\ell_i|C'_i|^{1.5}+|C_i|^2|M_i|+|C_i|^2|S_d(M_i\cup C_i)|+(|S_i|+1)|M_i\cup C_i|^{1.5})$ time.

4.3 The 2-Approximation Solution

As the EACS problem is NP-hard, the exact algorithm proposed has exponential time in the worst case unless P=NP. Here, we present two 2-approximation algorithms that can run in polynomial times. They are inspired by Algorithm 4 and Algorithm 5 (Lines 8-17). In Algorithm 4 GreedyUpperBound, we could treat the subgraph H' with D(H')=$\bar{d}$ as the final result. Note that Algorithm 4 not only provides an upper bound of the optimal score but also achieves 2-approximation of the optimal score (See Lemma 2). Furthermore, we note that we only need H'

as the returning approximate result, and hence Line 13 is omitted.

Theorem 2. Algorithm 4 discovers a 2-approximation solution for the EACS problem.

Proof: Let H^* be the optimal solution, and H be the community found by Algorithm 4. As the edge dissimilarity score satisfies the property of triangular inequality, i.e., $d(e_i, e_j) \le d(e_i, e_l) + d(e_j, e_l)$. Thus, we have

$$D(H) \le 2\arg \min_{e_{q_j} \in E_H(q)} (\arg \max_{e_i \in E(H)} d(e_i, e_{q_j}))$$

where $E_H(q)$ contain edges that are adjacent to the query vertex q and $E_H(q) \subseteq E(H)$. Furthermore, we have

$$\arg \min_{e_{q_j} \in E(q)} (\arg \max_{e_i \in E(H)} d(e_i, e_{q_j})) \le \arg \max_{e_{i'} \in E(H^*), e_{q_{j'}} \in E_{H^*}(q)} d(e_{i'}, e_{q_{j'}})$$

Otherwise, H can continue shrinking in Algorithm 4.

Due to $\arg\max_{e_{i'} \in E(H^*), e_{q_{j'}} \in E_{H^*}(q)} d(e_{i'}, e_{q_{j'}}) \le \mathrm{D}(H^*)$, we have

$$D(H) \le 2\arg \min_{e_{q_j} \in E_H(q)} (\arg \max_{e_i \in E(H)} d(e_i, e_{q_j})) \le 2\arg \max_{e_{i'} \in E(H^*), e_{q_{j'}} \in E_{H^*}(q)} d(e_{i'}, e_{q_{j'}}) \le 2D(H^*)$$

□

Algorithm 4 finds the community by global greedy peeling edges. When the dissimilarity score among edges is diverse and the final result has a small dissimilarity score, the number of iterations l will be large, leading to low efficiency. Besides, if the initial k-truss is large, it will incur a high cost in verifying the k-truss. To address this issue, we present another algorithm, called *LocalExpand*. Like Algorithm 5 (Lines 8-16), this algorithm finds the community by iteratively adding edges. The pseudo-code of LocalExpand is shown in Algorithm 8. For each chosen edge e_q, we iteratively increase d' to include more edges that could induce a connected k-truss (Lines 6-16). We also use (k, d) to prune irrelevant edges (Line 4). We also note that *LocalExpand* could also be used to find an upper bound for the optimal solution, i.e., $\bar{d}$=D(H).

Theorem 3. Algorithm 8 discovers a 2-approximation solution for the EACS problem.

Proof: The proof is similar to the proof of Theorem 2. □

Complexity Analysis. Let H be the returned subgraph. In each iteration of the while loop, Algorithm 8 costs at most O($| E(H) |^{1.5}$) time to determines whether there is a connected k-truss that contains q. Let l' be the number of iterations in the while loop. Algorithm 8 costs O($|E(G)|^{1.5}$+$deg(q)(l'|E(H)|^{1.5})$) time.

Comparing Two Approximation Algorithms. Algorithm 4 iteratively removes edges from a global range, while Algorithm 8 finds the community by local expanding. When the dissimilarity score among edges is diverse and the k-truss is large, Algorithm 8 may be preferable. We will empirically evaluate these two algorithms in the experiment section.

Algorithm 8: LocalExpand

```
Input: Graph $G$, integer $k$, a query vertex $q$.
Output: A connected $k$-truss $H$ with small D($H$).
1  Let $H_k$ be the $k$-truss of $G$ containing $q$;
2  $d$=1, $H \leftarrow \emptyset$;
3  for each $e_q$ is adjacent to $q$ do
4      $S=\emptyset$; $H_k \leftarrow$($k$, 2$d$)-truss;
5      $d' \leftarrow \arg\min_{e \in E(H_k) \setminus S} d(e, e_q)$;
6      while true do
7          if $d < d'$ then
8              break;
9          $S = \{e | d(e, e_q) \le d, \forall e \in E(H_k)\}$;
10         $H' \leftarrow$ connected $k$-truss containing $q$ induced by $S$;
11         if $H' \neq \emptyset$ then
12             if $d' < d$ or $d' = d$ and $| E(H) | < | E(H') |$ then
13                 $H = H'$; $d = d'$;
14             break;
15         else
16             $d' \leftarrow \arg\min_{e \in E(H_k) \setminus S} d(e, e_q)$;
17 return $H$;
```

5 Experiments

In this section, we evaluate the proposed edge-attributed community models and algorithms.

5.1 Experimental Setup

Datasets. We use seven real-life edge-attributed graphs in our experiments. ACM and DBLP are co-authorship networks, where each edge indicates the co-authorship between two authors. The edge attributes are about the topic of the corresponding papers. We generate 3 graphs from three years for ACM and DBLP respectively. For the edge attribute, we get the stemmed words and remove the stop words and meaningless keywords. According to the method [28], we generate the ground-truth communities based on the journals or conferences in which the authors publish their papers. We consider the connected authors that have published papers in the same journal or conference as a community. The IMDB is a movie cooperative network. We consider the words in the title and the genres to which the movie belongs to as the edge attributes. And we generate ground-truth communities by considering the actors in the same movie. We also generate three synthetic datasets, including Facebook, Amazon, YouTube. Amazon and YouTube can be found in SNAP (http://snap.stanford.edu/data/). The Facebook dataset is from [29]. These datasets have ground truth communities, but they have no edge attributes. To generate edge attributes, we first generate 1000 edge attributes. Then we randomly select 7 attributes and assign each of these attributes to edges in each ground-truth community. To model noise in the data, we randomly select a random integer in [1, 7] attributes for each edge that is not in the ground-truth community. Table 3 shows the statistics of the ten edge-attributed graphs.

TABLE 3: Dataset statistics.

Data Set	|V|	|E|	|Attributes|	k_{max}
ACM-1	57,152	110,991	8491	22
ACM-2	164,667	373,465	12,927	54
ACM-3	190,667	435,452	13,961	51
DBLP-1	341,625	934,159	18,097	57
DBLP-2	358,859	1,013,306	18,520	64
DBLP-3	376,291	1,097,535	18,562	101
IMDB	396,672	1,773,103	91,060	29
Facebook	3,622	72,964	1000	96
Amazon	334,863	925,872	1000	7
YouTube	1,134,890	2,987,624	1000	19

Algorithms. We compare the closest competitors, ETruss and ATruss [21], which search communities in graphs with vertex attributes. To apply ETruss and ATruss, we convert the edge-attributed graphs to vertex-attributed graphs using two methods mentioned in Section 1, which are respectively referred to as ETruss(Vertex), ATruss(Vertex) and ETruss(Line), ATruss(Line). We also compare our algorithms with the KTruss [27] that only considers the graph structures. As ACM-1, ACM-2, ACM-3, DBLP-1, DBLP-2, and DBLP-3 are related to the heterogeneous graph, we also compare the methods that are proposed for community search in heterogeneous graphs, including FastBCore [30], BatchECore [30], BatchVCore [30]. We evaluate various versions of our proposed algorithms, and we summarize all the algorithms as follows.

- ETruss(Vertex): the exact method in [21] by shifting edge attributes to vertices.
- ATruss(Vertex): the approximation method in [21] by shifting edge attributes to vertices.
- ETruss(Line): the exact method in [21] by converting the input edge-attributed graph to its vertex-attributed line graphs.
- ATruss(Line): the approximation method in [21] by converting the input edge-attributed graph to its vertex-attributed line graphs.
- KTruss: the model in [27] only consider the link structure.
- FastBCore, BatchECore, BatchVCore: the basic $(k, \mathcal{P})$-core model, edge-disjoint $(k, \mathcal{P})$-core model, the vertex-disjoint $(k, \mathcal{P})$-core model in [30].
- BasExact: our proposed Algorithm 1 (Section 3).
- AdvExact-G: our proposed advanced solution in Section 4.
- AdvExact-L: our proposed advanced solution in Section 4 and the upper bound computation is replaced by Algorithm 8
- AGlobal: our proposed approximation GlobalShrink algorithm (Section 4).
- ALocal: our proposed approximation LocalExpand algorithm (Section 4).

We also compare various version of AdvExact-L algorithm without employing certain pruning techniques, as follows.

- AdvExact-L-NoP: the AdvExact-L without (k, d)-truss pruning.
- AdvExact-L-NoC: the AdvExact-L without candidate reduction.
- AdvExact-L-NoL: the AdvExact-L without lower bounds.
- AdvExact-L-NoES: the AdvExact-L without edge selection rule.
- AdvExact-L-NoLem7: the AdvExact-L without using the Lemma 7.

Queries and Parameters. We evaluate our model and algorithms by varying the parameter k and its default value is 4. We randomly selected 100 queries from ground-truth communities and each query exists in at least one 4-truss. Note that we set the time limit as 1000s.

Evaluation Environment. We implement all the algorithms in Java and run experiments on a machine having an Intel(R) Xeon(R) 1.90GHz CPU and 24GB of memory.

5.2 Model Evaluation

Metrics. To evaluate the effectiveness of our model and our algorithms, we use the F1-score. In particular, let H be the found community and H' be the ground-truth community, then $F1(H, H') = \frac{2precision(H,H')recall(H,H')}{precision(H,H')+recall(H,H')}$, where $precision(H, H') = \frac{|V(H) \cap V(H')|}{|V(H)|}$, $recall(H, H') = \frac{|V(H) \cap V(H')|}{|V(H')|}$. Here, we compare ten algorithms: KTruss, ETruss(Vertex), ATruss(Vertex), ETruss(Line) ATruss(Line), FastBCore, BatchECore, BatchVCore, ALocal, and AdvExact-L.

Evaluation. Figure 4 reports the F1-scores on all datasets with various k's. Note that on all DBLP datasets, IMDB, Amazon and YouTube, the line graph is out of memory; hence, the results of ETruss(Line) and ATruss(Line) are not reported. The ETruss(Vertex) and ETruss(Line) have lower F1-scores compared with ATruss(Vertex) and ATruss(Line). This is because most query tasks are not completed in 1000s. FastBCore, BatchECore, and BatchVCore are the methods that are proposed for community search in the heterogeneous graph. We use the author-paper-author as the meta-path to search the community containing the given author. Our algorithms ALocal and AdvExact-L achieve the highest F1-scores in most cases when k is small. Besides, we also see that the results of the ALocal are similar to AdvExact-L, which shows that our approximation algorithm could obtain communities with quality similar to the exact algorithms. With the increase of k, the F1-scores of all algorithms decrease. This is because when k is large, many edges that do not meet the requirements of k-truss are deleted, which leads to only part of ground-truth communities can be found. Figure 5 shows the average community size returned by all algorithms on all datasets with various k's. Note that when the size is 0, the result is not reported. On YouTube, ETruss(Vertex) can not complete most query tasks in 1000s. Thus, the results of ETruss(Vertex) are lower than the results of other algorithms. Since when converting the graph to the line graph, the graph becomes very dense, and the ETruss(Line) and ATruss(Line) always return the community with many vertices. We can see that our proposed algorithms always return the community with a small size. This demonstrates that our method can filter more vertices. In Figure 6, we show the ratio of queries that return no empty communities. We can see that the ratio decreases with the increase of k. This is because when k is large, some queries may not exist in one k-truss.

Case Study. Figure 7 shows one result of querying the ACM-2 and Facebook. The red nodes are the query nodes. The

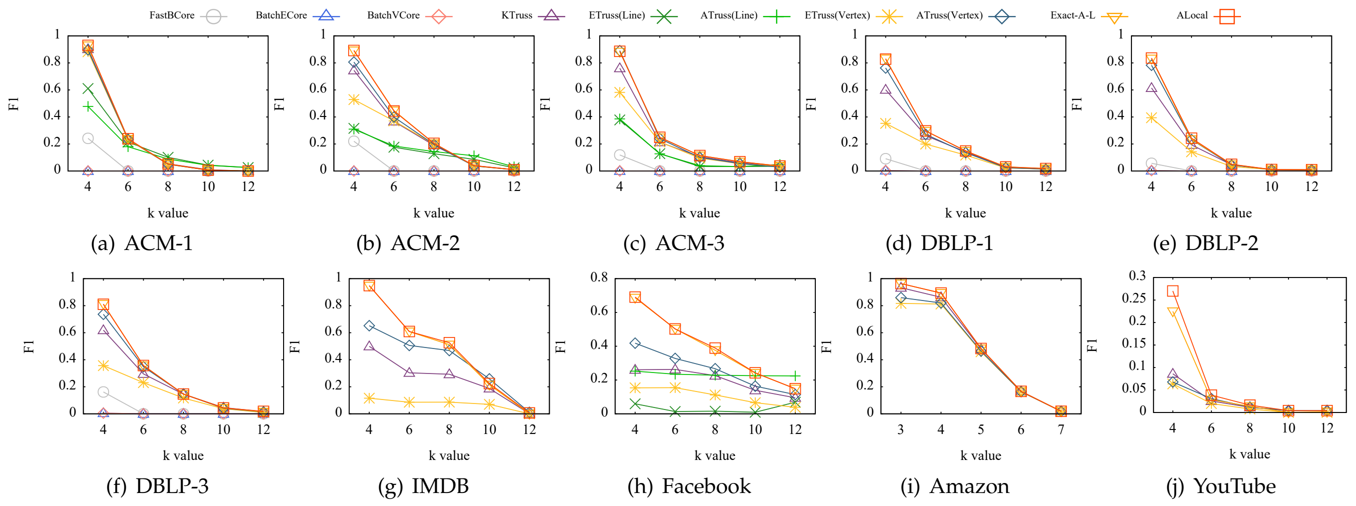


Fig. 4: F1-score with varying k.

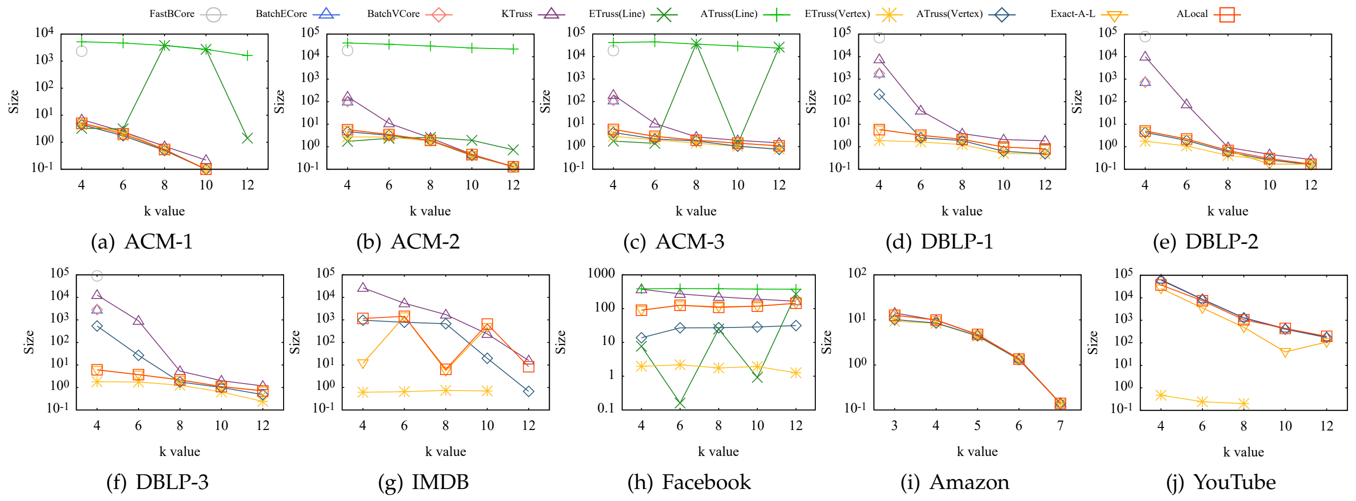


Fig. 5: Average community size evaluation with varying k.

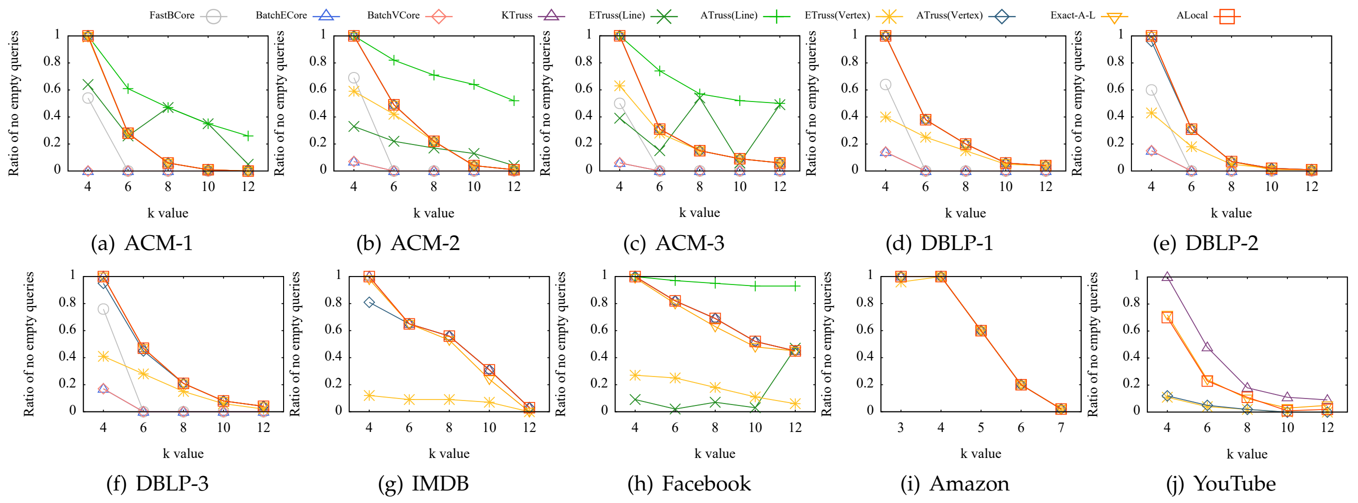
blue nodes are the nodes that exist in the ground-truth community (Figure 7 (a) and (d)), the community returned by ATruss(Vertex) (Figure 7 (b) and (e)), and the community returned by ALocal (Figure 7 (c) and (f)), respectively. Since KTruss and ATruss(Line) return a community with too many nodes to show, we do not give the case studies for the two algorithms. We can see that the ATruss(Vertex) could not find all the nodes in ground-truth communities, while our method can discover the community that is similar to the ground-truth community. This is because when converting the edge attributes to the node attributes, the ATruss(Vertex) cannot capture real relationships between attributes and edges.

5.3 Efficiency Evaluation

In this section, we evaluate the running times of our proposed algorithms. We also investigate the index construction time and memory cost.

Efficiency on Different Algorithms for EACS. We compare the efficiency of AdvExact-L, ALocal, KTruss, ETruss(Vertex), ATruss(Vertex), ETruss(Line), ATruss(Line), FastBCore, BatchECore, and BatchVCore in Figure 8. As on all DBLP datasets, IMDB, Amazon and YouTube, ETruss(Line) and ATruss(Line) run out of memory, their results are not reported. In Figure 8, we can see that AdvExact-L, ALocal and KTruss always outperform ETruss or ATruss algorithms. Our algorithms are faster than ETruss or ATruss because we incorporate delicate (k,d)-truss structures and pruning techniques. Since KTruss, FastBCore, BatchECore, and BatchVCore do not consider the edge similarity, their efficiency is high. The running times of AdvExact-L and ALocal decrease with k because a larger k leads to a smaller search space. But for Exact-A-L, in certain cases, even if the search space is narrowed, the computation of the different pruning techniques will also lead to the search time increasing. Thus, the search time may increase when k increases (e.g., in Figure 8 (e) from k=8 to k=12). Considering that AdvExact-L and ALocal extract higher-quality communities compared with KTruss, overall our algorithms provide better efficacy.

Efficiency of Different Techniques. We compare the efficiency of AdvExact-L, AdvExact-L-NoP, AdvExact-L-NoC, AdvExact-L-NoL, AdvExact-L-NoES, AdvExact-L-NoLem7, Exact-B on different datasets to evaluate different techniques in Figure 9. Note that Exact-B and AdvExact-L-NoP run out



(a) ACM-1 (b) ACM-2 (c) ACM-3 (d) DBLP-1 (e) DBLP-2
(f) DBLP-3 (g) IMDB (h) Facebook (i) Amazon (j) YouTube

Fig. 6: Ratio of no empty queries with varying k.

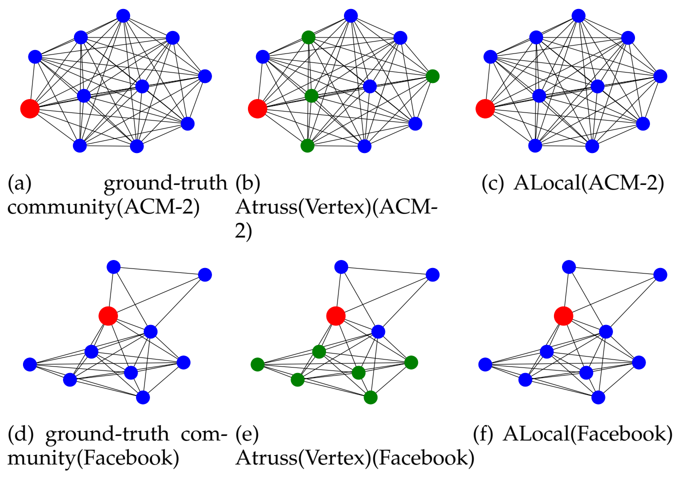

(a) ground-truth community(ACM-2) (b) Atruss(Vertex)(ACM-2) (c) ALocal(ACM-2)
(d) ground-truth community(Facebook) (e) Atruss(Vertex)(Facebook) (f) ALocal(Facebook)

Fig. 7: Case study.

of memory on YouTube, their results are not reported. We observe that AdvExact-L significantly outperforms Exact-B. Besides, different techniques lead to various effects. In most cases, when removing any technique, the running time increases. It is worth noting that the (k, d)-truss pruning rule has the greatest impact on efficiency. Note that if the (k, d)-truss pruning is not used, the candidate reduction and edge selection rules need much time to find the corresponding set, which will lead to the AdvExact-L-NoP being slower than the Exact-B.

Efficiency of Exact Algorithms and Approximation Algorithms. Figure 10 reports the running time of our proposed two approximation algorithms and our advanced exact solutions based on different upper-bound computations. In most cases, the ALocal runs faster than AGlobal, AdvExact-L is faster than AdvExact-G. This demonstrates the effectiveness of the local expansion in Algorithm 8.

Index Construction Time and Index Size. We show the index construction time and index size in Figure 11. With the increase in the graph size, the running time and memory usage also increased. However, the index construction for all datasets can be finished within 1 hour and the index size is within 250M, demonstrating that the indexing scheme is practically affordable.

5.4 Discussion

EACS employs the classic k-truss model to capture the structural cohesiveness. The k-truss model relaxes the clique model and enhances the k-core model. Besides, it captures the cohesiveness in communities by utilizing the triangle, which usually implies strong relationships in communities [19]. Our model also has these nice properties of the k-truss model. On the other hand, EACS model is somewhat affected by the limitations inherent to the k-truss model. For example, if k is set to large, to satisfy the cohesiveness constraint, many weak edges will be removed such that some qualified vertices may be thus removed from the community. In our model, k is set small. Moreover, to resist some unqualified vertices from mixing into the community due to the small k, we exploit the similarity of edge attributes to get a refined community by removing the edges that are not similar to each other. Intuitively, the community with similar edge attributes is generally firm. Thus, our proposed model could alleviate the influence of weak edge removal since it considers the cohesiveness constraint and similarity of edge attributes at the same time. We have done experiments on a wide range of datasets, which have different numbers of edges (3622∼1134890), vertices (72964∼2987624), edge attributes (1000∼91060), and k_{max} values (7∼101). From Figure 4 in Section 5.2, we have a general observation across all datasets that when k is small (i.e., $k = 3$ or 4), the query returns an effective community with a higher $F1$ score, while the effectiveness of the communities decreases as k increases. Therefore, we recommend that users set k to 3 or 4 when using our model.

6 Related Work

Community detection. Community detection has been widely studied in the last decade. Most works often mainly by exploring link structure to find cohesively interaction groups. However, in the real networks, there exists much information that could be used to help capture meaningful communities, such as vertex attributes [10], [11], [12], [31], timestamp [32], [33], direction [34] etc. Qi et al. [4] first propose exploiting the edge content to find communities with similar edge content. And they show that edge content could greatly improve the effectiveness of community

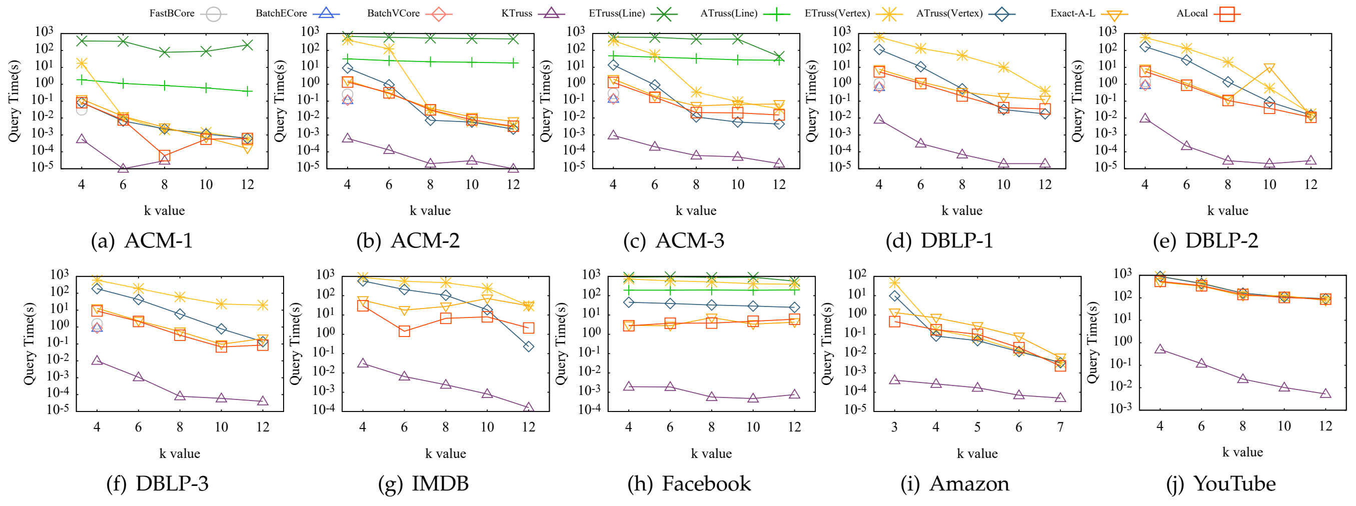


Fig. 8: Efficiency of different algorithms for EACS.

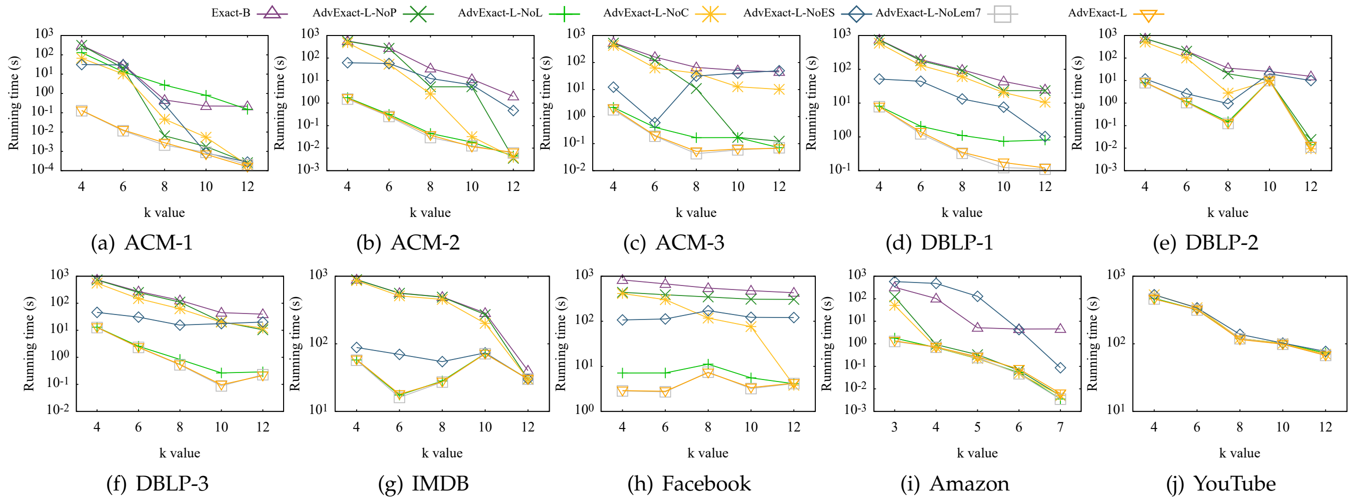


Fig. 9: Efficiency of different techniques.

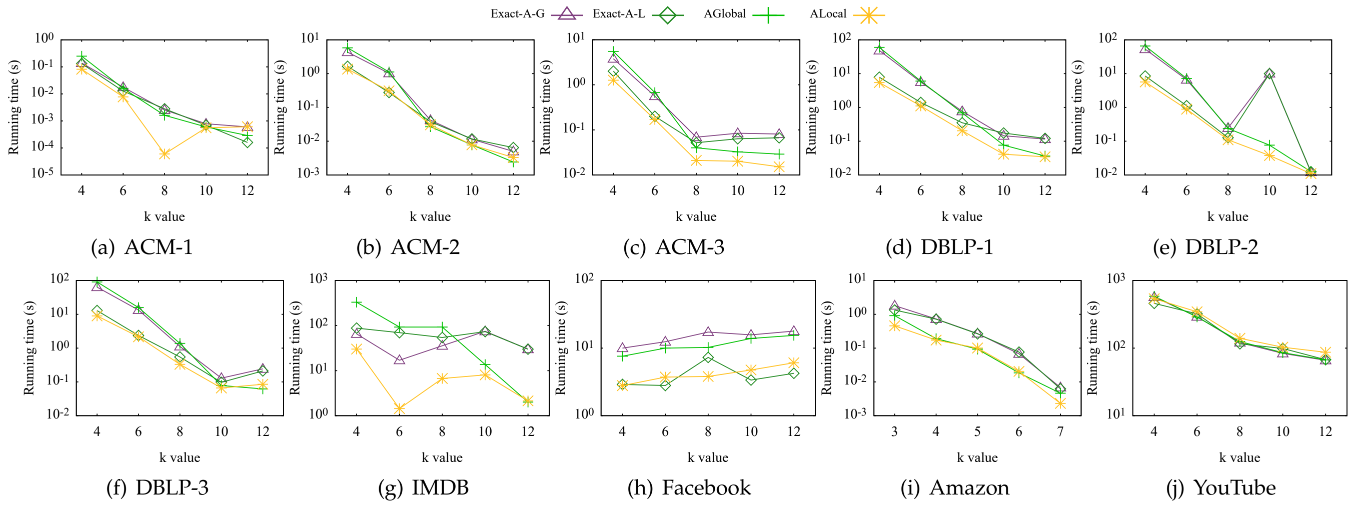


Fig. 10: Efficiency of exact algorithms and approximation algorithms.

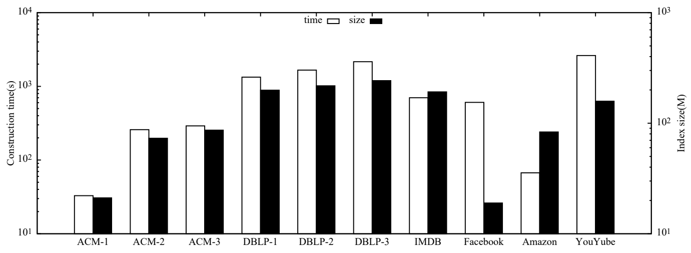


Fig. 11: Evaluating index construction.

detection. After that, Wang et al. [7] developed methods to discover topically meaningful communities in the networks with node and edge content. More recently, Amburg et al. [6] find the community with categorical edge labels. In general, community detection always finds all communities by exploring a global criterion; hence, it is inefficient to adapt the methods above mentioned for our EACS problem.

Community search. Community search aims to discover a densely connected group that contains query vertices, which has attracted much attention in recent years. To capture the cohesive structure, different models have been developed, including k-core [15], [17], k-truss [16], [18], k-clique [35], density modularity [36], etc. However, all these works only focus on the link structure of the community. Recently, some works have been developed for more complex networks such as location-based networks [37], [38], [39], heterogeneous networks [30], [40], temporal networks [41]. The most similar problem to EACS is the community search in vertex-attributed networks, such as [1], [21], [22], [42]. The difference between our work and [1], [42] is that they require to take *both* vertices and attributes as query input and find the community containing query vertices and similar *vertex attributes* within it. There are also works that only take attributes [22] or vertices [21] as input, and in our experiments, we have systematically compared with [21]. There is one work [43] that does not take attributes as query input, but their work is based on k-core and they need two user-specified parameters, k and r, to be set, while our model only requires one parameter k. Fewer parameters can reduce the burden on the users. Lastly, we note that Kang et al. [5] proposed community search in edge-attributed networks, but they require users to input appropriate query keywords, which is not an easy task for users.

7 Conclusion

In this paper, we focused on the edge-attributed community search problem (EACS) that allows finding a community containing query vertex with cohesive structure and homogeneous edge attributes. We proved that this problem is NP-hard. We first proposed a basic exact solution by enumerating all possible k-truss. To quickly find the communities, several nontrivial techniques are proposed, including (k, d)-truss pruning, candidate reduction, the lower bounds-based early termination, and the edge selection-based search space partition. In addition, we also proposed two efficient approximation algorithms with an approximation ratio of 2. Extensive experiments demonstrated the effectiveness and efficiency of our model and algorithms.

References

[1] X. Huang and L. V. Lakshmanan, "Attribute-driven community search," *Proceedings of the VLDB Endowment*, vol. 10, no. 9, pp. 949–960, 2017.

[2] C. Wang, H. Wang, H. Chen, and D. Li, "Attributed community search based on effective scoring function and elastic greedy method," *Information Sciences*, vol. 562, pp. 78–93, 2021.

[3] Y. Zhu, J. He, J. Ye, L. Qin, X. Huang, and J. X. Yu, "When structure meets keywords: Cohesive attributed community search," in *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, 2020, pp. 1913–1922.

[4] G.-J. Qi, C. C. Aggarwal, and T. Huang, "Community detection with edge content in social media networks," in *2012 IEEE 28th International conference on data engineering*. IEEE, 2012, pp. 534–545.

[5] Q. Kang, Y. Kang, and H. Kong, "Edge-attributed community search for large graphs," in *Proceedings of the 2nd International Conference on Big Data Research*, 2018, pp. 114–118.

[6] I. Amburg, N. Veldt, and A. Benson, "Clustering in graphs and hypergraphs with categorical edge labels," in *Proceedings of The Web Conference 2020*, 2020, pp. 706–717.

[7] C.-D. Wang, J.-H. Lai, and S. Y. Philip, "Neiwalk: Community discovery in dynamic content-based networks," *IEEE transactions on knowledge and data engineering*, vol. 26, no. 7, pp. 1734–1748, 2013.

[8] A. L. Hu and K. C. Chan, "Utilizing both topological and attribute information for protein complex identification in ppi networks," *IEEE/ACM transactions on computational biology and bioinformatics*, vol. 10, no. 3, pp. 780–792, 2013.

[9] I. Alsmadi and I. Alhami, "Clustering and classification of email contents," *Journal of King Saud University-Computer and Information Sciences*, vol. 27, no. 1, pp. 46–57, 2015.

[10] C. Zhe, A. Sun, and X. Xiao, "Community detection on large complex attribute network," in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, 2019, pp. 2041–2049.

[11] M. Qin and K. Lei, "Dual-channel hybrid community detection in attributed networks," *Information Sciences*, vol. 551, pp. 146–167, 2021.

[12] Z. Chen, A. Sun, and X. Xiao, "Incremental community detection on large complex attributed network," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 15, no. 6, pp. 1–20, 2021.

[13] D. Jin, X. Wang, D. He, J. Dang, and W. Zhang, "Robust detection of link communities with summary description in social networks," *IEEE Transactions on Knowledge and Data Engineering*, vol. 33, no. 6, pp. 2737–2749, 2019.

[14] T. Yoshida, "Weighted line graphs for overlapping community discovery," *Social Network Analysis and Mining*, vol. 3, no. 4, pp. 1001–1013, 2013.

[15] M. Sozio and A. Gionis, "The community-search problem and how to plan a successful cocktail party," in *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2010, pp. 939–948.

[16] X. Huang, L. V. Lakshmanan, J. X. Yu, and H. Cheng, "Approximate closest community search in networks," *arXiv preprint arXiv:1505.05956*, 2015.

[17] K. Yao and L. Chang, "Efficient size-bounded community search over large networks." *Proc. VLDB Endow.*, vol. 14, no. 8, pp. 1441–1453, 2021.

[18] E. Akbas and P. Zhao, "Truss-based community search: a truss-equivalence based indexing approach," *Proceedings of the VLDB Endowment*, vol. 10, no. 11, pp. 1298–1309, 2017.

[19] B. Liu, F. Zhang, W. Zhang, X. Lin, and Y. Zhang, "Efficient community search with size constraint," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 97–108.

[20] Y. Fang, R. Cheng, S. Luo, and J. Hu, "Effective community search for large attributed graphs," *Proceedings of the VLDB Endowment*, vol. 9, no. 12, pp. 1233–1244, 2016.

[21] Q. Liu, Y. Zhu, M. Zhao, X. Huang, J. Xu, and Y. Gao, "Vac: vertex-centric attributed community search," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 937–948.

[22] L. Chen, C. Liu, K. Liao, J. Li, and R. Zhou, "Contextual community search over large social networks," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 88–99.

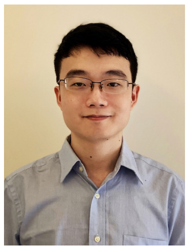
[23] Z. Zhang, X. Huang, J. Xu, B. Choi, and Z. Shang, "Keyword-centric community search," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 422–433.
[24] K. Kloster and D. F. Gleich, "Heat kernel based community detection," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014, pp. 1386–1395.
[25] J. Shao, Z. Han, Q. Yang, and T. Zhou, "Community detection based on distance dynamics," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2015, pp. 1075–1084.
[26] L. Prokhorenkova and A. Tikhonov, "Community detection through likelihood optimization: in search of a sound model," in *The World Wide Web Conference*, 2019, pp. 1498–1508.
[27] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu, "Querying k-truss community in large and dynamic graphs," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014, pp. 1311–1322.
[28] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, 2012, pp. 1–8.
[29] Y. Zhang, Y. Xiong, Y. Ye, T. Liu, W. Wang, Y. Zhu, and P. S. Yu, "Seal: Learning heuristics for community detection with generative adversarial networks," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 1103–1113.
[30] Y. Fang, Y. Yang, W. Zhang, X. Lin, and X. Cao, "Effective and efficient community search over large heterogeneous information networks," *Proceedings of the VLDB Endowment*, vol. 13, no. 6, pp. 854–867, 2020.
[31] Y. Ruan, D. Fuhry, and S. Parthasarathy, "Efficient community detection in large networks using content and links," in *Proceedings of the 22nd international conference on World Wide Web*, 2013, pp. 1089–1098.
[32] A. Gorovits, E. Gujral, E. E. Papalexakis, and P. Bogdanov, "Larc: Learning activity-regularized overlapping communities across time," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, pp. 1465–1474.
[33] D. Zhuang, M. J. Chang, and M. Li, "Dynamo: Dynamic community detection by incrementally maximizing modularity," *IEEE Transactions on Knowledge and Data Engineering*, 2019.
[34] Y. Kim, S.-W. Son, and H. Jeong, "Finding communities in directed networks," *Physical Review E*, vol. 81, no. 1, p. 016103, 2010.
[35] L. Yuan, L. Qin, W. Zhang, L. Chang, and J. Yang, "Index-based densest clique percolation community search in networks," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 5, pp. 922–935, 2017.
[36] J. Kim, S. Luo, G. Cong, and W. Yu, "DMCS : Density modularity based community search," in *SIGMOD*, 2022, pp. 889–903.
[37] A. Al-Baghdadi and X. Lian, "Topic-based community search over spatial-social networks," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2104–2117, 2020.
[38] J. Luo, X. Cao, X. Xie, Q. Qu, Z. Xu, and C. S. Jensen, "Efficient attribute-constrained co-located community search," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 1201–1212.
[39] Y. Fang, Z. Wang, R. Cheng, X. Li, S. Luo, J. Hu, and X. Chen, "On spatial-aware community search," *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 4, pp. 783–798, 2018.
[40] X. Jian, Y. Wang, and L. Chen, "Effective and efficient relational community detection and search in large dynamic heterogeneous information networks," *Proceedings of the VLDB Endowment*, vol. 13, no. 10, pp. 1723–1736, 2020.
[41] R.-H. Li, J. Su, L. Qin, J. X. Yu, and Q. Dai, "Persistent community search in temporal networks," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 2018, pp. 797–808.
[42] Y. Fang, R. Cheng, Y. Chen, S. Luo, and J. Hu, "Effective and efficient attributed community search," *The VLDB Journal*, vol. 26, no. 6, pp. 803–828, 2017.
[43] F. Zhang, Y. Zhang, L. Qin, W. Zhang, and X. Lin, "When engagement meets similarity: Efficient (k, r)-core computation on social networks," in *Proc. VLDB Endow*, 2017, pp. 10(10): 998–1009.

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China (62032013), NTU SUG, Singapore MOE Tier 1 (RS05/21, RG18/21) and Tier 2 (T2EP20122-0008) fundings, Innovative Talents of Higher Education in Liaoning Province (No. LR2020076), the basic Research Operating Funds for National Defence Major Incubation Projects (No. N2116017), Chinese Scholarship Council.

Ling Li is currently working toward a Ph.D. degree in the School of Computer Science and Engineering at Northeastern University, China. Her current research interests include graph algorithms, data mining, and big data.

Yuhai Zhao received the B.S. degree in computer science and technology and Ph.D. degree in computer software and theory from Northeastern University, Shenyang, China, in 1999 and 2007, respectively. He is currently a Professor at the Department of Computer Science, Northeastern University. He is a Senior Member of China Computer Federation (CCF). His research interests include data mining, database, machine learning, and bioinformatics.



Siqiang Luo is currently an assistant professor at the School of Computer Science and Engineering, Nanyang Technological University. He was a postdoc at Harvard University from 2019 to 2020. He received his Ph.D. degree in computer science from the University of Hong Kong in 2019. He received his master's and bachelor's degrees from Fudan University in 2013 and 2010 respectively. His research interest includes data management and data mining.

Guoren Wang (Member, IEEE) received the BSc, MSc, and Ph.D. degrees in computer science from Northeastern University, China, in 1988, 1991, and 1996, respectively. He is currently a professor at the School of Computer Science and Technology, Beijing Institute of Technology, China. His research interests include XML data management, query processing and optimization, bioinformatics, highdimensional indexing, parallel database systems, and cloud data management. He has published more than 350 research papers.

Zhengkui Wang (Member, IEEE) received the BSc, MSc, and Ph.D. degrees in computer science from, Heilongjiang University of Science and Technology, Harbin Institute of Technology, and National University of Singapore in 2006, 2008, and 2013, respectively. He is currently an associate professor at the InfoComm Technology cluster, Singapore Institute of Technology, Singapore. His research interests include data mining, text analytics, imaging, machine learning, and graph neural networks.