VAC: Vertex-Centric Attributed Community Search

Qing Liu[†1], Yifan Zhu[§2], Minjun Zhao[§3], Xin Huang[†4], Jianliang Xu[†5], Yunjun Gao[§♯6]

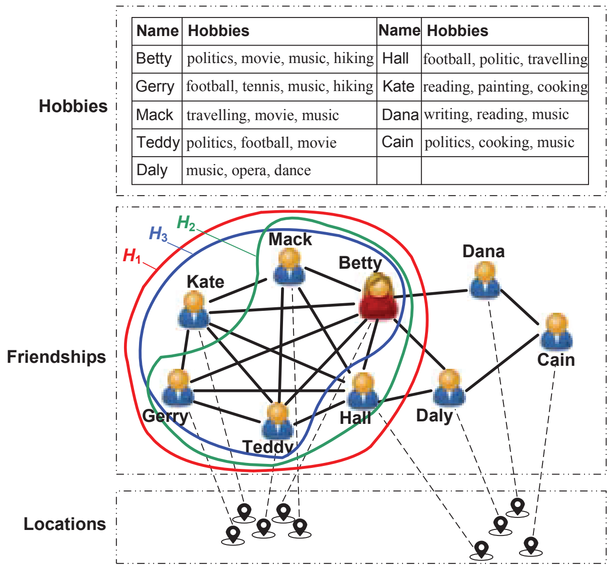
[†]*Department of Computer Science, Hong Kong Baptist University, Hong Kong, China*
[§]*College of Computer Science, Zhejiang University, Hangzhou, China*
[♯]*AlibabaZhejiang University Joint Institute of Frontier Technologies, Hangzhou, China*
{[1]qingliu, [4]xinhuang, [5]xujl}@comp.hkbu.edu.hk, {[2]xtf_z, [3]minjunzhao, [6]gaoyj}@zju.edu.cn

***Abstract*—Attributed community search aims to find the community with strong structure and attribute cohesiveness from attributed graphs. However, existing works suffer from two major limitations: (i) it is not easy to set the conditions on query attributes; (ii) the queries support only a single type of attributes. To make up for these deficiencies, in this paper, we study a novel attributed community search called vertex-centric attributed community (VAC) search. Given an attributed graph and a query vertex set, the VAC search returns the community which is densely connected (ensured by the k-truss model) and has the best attribute score. We show that the problem is NP-hard. To answer the VAC search, we develop both exact and approximate algorithms. Specifically, we develop two exact algorithms. One searches the community in a depth-first manner and the other is in a best-first manner. We also propose a set of heuristic strategies to prune the unqualified search space by exploiting the structure and attribute properties. In addition, to further improve the search efficiency, we propose a 2-approximation algorithm. Comprehensive experimental studies on various real-world attributed graphs demonstrate the effectiveness of the proposed model and the efficiency of the developed algorithms.**

I. Introduction

Community search aims to find the community containing the query vertex set, which is widely explored for personalized community analysis [1]. Community search on both simple and complex networks have been studied, and numerous community search models have been developed, such as k-core [1]–[3], k-truss [4]–[6], and clique [7], [8].

Owing to the rich information present for real-world entities, the vertices of networks are usually associated with some attributes. For example, in Facebook, users can specify hobbies, locations, and other information in their profiles. By combining the graph structure and attribute information, community search in attributed graphs can discover more meaningful communities. Consider the social network shown in Figure 1, where each user is associated with the information of hobbies and location. Assume that we take *Beety* as the query vertex and aim to find a community of 4-truss (i.e., every edge in the community should be contained in at least 2 triangles) containing *Beety*. If we do not consider any other information, the subgraph H_1 indicated by the red circle is the result. If we consider users' hobbies, the community H_2 shown by the green circle is the answer. Compared with H_1, H_2 excludes *Kate* as she does not share any hobby with others. Similarly, we can find the community H_3 by considering the user locations.



Name	Hobbies	Name	Hobbies
Betty	politics, movie, music, hiking	Hall	football, politic, travelling
Gerry	football, tennis, music, hiking	Kate	reading, painting, cooking
Mack	travelling, movie, music	Dana	writing, reading, music
Teddy	politics, football, movie	Cain	politics, cooking, music
Daly	music, opera, dance		

Fig. 1: Motivating Example

There have been a number of studies on the community search over attributed graphs [9]–[16]. However, existing works suffer from the following two limitations.

- First, it is not an easy task for users to specify the appropriate conditions on query attributes. This is because, in general, users are not familiar with the attribute distributions in the whole graph. If the query conditions are too strict, it may return an empty community. For example, in Figure 1, if we require that every user in the community should contain *movie* and *music* in their hobbies, then we cannot find any result. On the other hand, if the query conditions are loose, the returned community may not be attribute cohesive.
- Second, existing works are not flexible. In particular, these works support only a single type of attributes. For example, [13]–[16] support only geographical attributes while [9]–[12] consider only textual attributes. If users have to apply different queries when searching different attributed graphs, it can be an extra burden. It would be desirable that a single query can handle different kinds of attributed graphs.

To make up for these deficiencies, in this paper, we explore a novel problem, called vertex-centric attributed community (VAC) search. As mentioned above, the community in attributed graphs should consider both the structure cohesiveness and attribute cohesiveness. For the structure cohesiveness, we employ the k-truss model as it can find the community with strong cohesiveness while achieving good performance [4]–[6]. Specifically, every edge in a k-truss is involved in at least $k-2$ triangles. With regard to the attribute cohesiveness, we propose a new notation of *attribute score*, which is defined as the maximum attribute score of all pairs of vertices in a subgraph. The rationale is that if a subgraph has strong attribute cohesiveness, every pair of vertices in the subgraph should also be attribute cohesive. Note that we can adopt different functions to compute the attribute score, e.g., Euclidean distance function for geographical attributes and Jaccard distance function for textual attributes, so as to meet different search requirements. Based on the two metrics of cohesiveness, we formally define the VAC problem as follows. Given an attributed graph G, a query vertex set Q, and a parameter k, the VAC problem aims to find the maximal connected subgraph $H \subseteq G$ such that H is a k-truss containing Q and has the minimum attribute score (i.e., the smallest pairwise distance). Compared with existing works, the VAC problem does not need any input on query attributes and can deal with different kinds of attributed graphs.

We prove that the VAC problem is NP-hard. To tackle the VAC problem efficiently, we propose two exact algorithms. The first algorithm is called DFS-based VAC algorithm, which searches the solution in a depth-first manner. In the algorithm, we propose a series of heuristic strategies to improve the performance. For example, we exploit the property of the k-truss to prune the vertices that cannot contribute to the final result, and use the low bounds of subgraphs' attribute scores to terminate the search as early as possible. Moreover, we optimize the search order to achieve better efficiency. The second algorithm is called BFS-based VAC algorithm. In the algorithm, it iteratively finds the subgraphs in descending order of their attribute scores until the subgraph with the minimum attribute score is identified. We also propose a technique to avoid redundant operations. In addition, to improve scalability, we propose an approximate algorithm that iteratively and greedily removes the vertices having the maximal attribute score w.r.t. the query vertex set. We prove that the approximate algorithm can achieve 2-approximation guarantee.

The main contributions of this paper are summarized as follows:

- We propose a novel community model on attributed graphs and study the VAC search problem. We prove that the problem is NP-hard.
- We develop two efficient exact algorithms to solve the VAC problem by integrating a set of pruning strategies.
- We propose a 2-approximation algorithm to significantly accelerate the search process with bounded accuracy.
- We conduct extensive experiments on real networks to demonstrate the performance of the proposed algorithms.

The rest of this paper is organized as follows. Section II reviews related work. Section III formally defines the problem, followed by the problem analysis in Section IV. Sections V and VI present the exact and approximate algorithms, respectively. Section VII reports the extensive experimental results. Finally, Section VIII concludes the paper.

II. Related Work

A. Community Search over Non-Attributed Graphs

Community search has been extensively studied since it was first introduced by Sozio and Gionis [1]. There are several surveys of community search [17]–[19]. In general, existing works can be classified into two categories based on the type of graphs, i.e., simple graphs or complex graphs.

The works of community search on simple graphs focus on devising different community models, such as core-based model [1]–[3], truss-based model [4]–[6], clique-based model [7], [8], [20], [21], edge connectivity component (ECC)-based model [22], [23], and query-biased density model [24].

Community search has also been investigated for complex networks. Fang et al. [25] and Liu et al. [26] study the community search over directed graphs. Chen et al. [27] perform the community search over the profiled graphs in which each vertex has labels arranged in a hierarchical manner. Ebadian and Huang [28] address the community search problem over public-private graphs that consist of one public graph and multiple private graphs.

B. Community Search over Attributed Graphs

Community search over attributed graphs considers both the structure cohesiveness and attribute cohesiveness. According to the query input, existing works of community search over attributed graphs can be classified into two categories. The first category takes both vertices and attributes as query input, and returns the attribute-cohesive community containing the query vertices [9]–[16]. The second category takes only attributes as query input, and returns the community related to the query attributes [29]–[31]. As can be seen, all these existing studies require users to specify a set of attributes as query input, which limits their applications. In contrast, in our proposed community model no attributes need to be specified for a query and the algorithm can find the community with highly similar attributes.

Another related line of work is community detection over attributed graphs (CDA) [32]–[36]. However, they differ from the problem of community search over attributed graphs (CSA) studied in this paper in several aspects [9], [10], [19]. First, the goals are different. CDA usually detects all communities in a graph, while CSA is query-dependent and aims to find the communities containing a set of query vertices/attributes. Second, the criteria of defining communities are different. In CSA, the criteria of defining communities are based on query parameters given by the users, e.g., k-truss. In contrast, CDA often uses the same global criterion (e.g., modularity) to detect communities. Third, the algorithms are different. CSA solutions, usually supported by graph indexes, can search communities very efficiently. In contrast, CSD solutions are often time consuming and unscalable to big graphs, e.g., the clustering-based methods [37] need at least one scan of the entire graph. It is worth mentioning that the seed-centric approaches proposed for CDA (e.g., [38]–[40]) also cannot be employed for our problem due to two main reasons: (1) The community found by the seed-centric approaches may not contain the initial seeds (recall our CSA problem requires that the result community must contain the query vertices). (2) The community model employed in our problem does not fit for the seed-centric approaches due to their non-monotonic properties. In addition, the seed-centric approaches do not provide quality guarantees for the returned (approximate) solution, while our proposed algorithm can achieve 2-approximation to the optimal solution.

III. Problem Formulation

We consider an undirected, unweighted, attributed graph $G = (V_G, E_G, A_G)$, where V_G is the set of vertices, E_G is the set of edges, and A_G is the set of attributes associated with vertices in G for describing vertex properties. Each vertex $v \in V_G$ is associated with a set of attributes $A(v) \subseteq A_G$. Let $n = |V_G|$ and $m = |E_G|$ be the number of vertices and edges, respectively. We denote the set of neighbors of a vertex v as $N_G(v)$ in G, i.e., $N_G(v) = \{u \in V_G : (u, v) \in E_G\}$. When the context is obvious, we drop the subscript and denote $N_G(v)$ as $N(v)$. For a subgraph $H = (V_H, E_H, A_H)$ of G, $V_H \subseteq V_G$, $E_H \subseteq E_G$, and $A_H \subseteq A_G$ hold. Before formally defining our problem, we first introduce the metrics of structure cohesiveness and attribute cohesiveness employed in our paper.

A. Structure Cohesiveness

In this paper, we employ k-truss [41] to measure the structure cohesiveness of communities since it is well recognized that k-truss has strong structural cohesiveness and high computational efficiency. In particular, k-truss is defined on triangles in a graph. A triangle, denoted as $\triangle_{uvw}$, is a cycle of length 3 comprising three distinct vertices u, v, and w in G. Based on triangles, we present the definitions of edge support and k-truss as follows.

Definition 1. (Edge Support). *The support of an edge $e = (u, v) \in E_G$, denoted as $sup_G(e)$, is the number of triangles containing e, i.e., $sup_G(e) = |\{\triangle_{uvw} : w \in N(u) \cap N(v)\}|$.*

Definition 2. (Connected K-Truss). *Given a graph G and an integer $k \geq 2$, a connected k-truss is a connected subgraph $H \subseteq G$ such that for every edge $e \in E_H$, $sup_H(e) \geq k - 2$.*

Every edge in a connected k-truss H is contained in at least $k - 2$ triangles in H. In addition, a *maximal* connected k-truss H is the largest subgraph of G such that there exists no connected k-truss $H' \supset H$. For example, in Figure 1, H_1, H_2, and H_3 are all 4-trusses as every edge in these three subgraphs is contained in 2 triangles. In addition, since $H_2 \subset H_1$ and $H_3 \subset H_1$, H_1 is a maximal 4-trusses but H_2 and H_3 are not. It is worth mentioning that our community model also can be extended to other dense subgraph models, such as k-core [1]–[3], k-clique [20], k-ECC [22], [23], and k-plex [21].

B. Attribute Cohesiveness

We define the attribute cohesiveness as follows. Given two vertices u, v, we use the attribute score, denoted as $\mathsf{Ascore}(u, v)$, to represent the attribute cohesiveness of u and v in an attributed graph G. For different types of attributes, we can employ different methods to compute the attribute score of two vertices. For the sake of unity, we employ the distance metric to measure the similarity of two vertices, which satisfies the *triangle inequality*, i.e. $\mathsf{Ascore}(u, v) \leq \mathsf{Ascore}(u, w) + \mathsf{Ascore}(w, v)$. For instance, assume that $A(v)$ represents the geo-location attribute of v, we use *Euclidean distance* to compute the attribute score. Specifically, let $A(v)_i$ be the i-th dimension of v, the attribute score of u and v is $\mathsf{Ascore}(u, v) = \sqrt{\sum_{i=1}^{d}(A(u)_i - A(v)_i)^2}$. In contrast, if $A(v)$ represents the textual information of v, the attribute score $\mathsf{Ascore}(u, v)$ can be measured by *Jaccard distance*, i.e., $\mathsf{Ascore}(u, v) = 1 - \frac{|A(u) \cap A(v)|}{|A(u) \cup A(v)|}$. Both Euclidean distance and Jaccard distance satisfy the triangle inequality [42]. When different types of attributes co-exist, we can employ a unified function to combine different distance functions, e.g.: $\mathsf{Ascore}(u, v) = \alpha \cdot \frac{Sdist(u,v)}{Sdist_{max}} + (1 - \alpha) \cdot \frac{Tdist(u,v)}{Tdist_{max}}$, where $Sdist(u, v)$ and $Tdist(u, v)$ compute the spatial distance and textual distance, respectively; $Sdist_{max}$ and $Tdist_{max}$ are the maximal spatial distance and maximal textual distance, respectively, used for normalization; the parameter $0 \leq \alpha \leq 1$ is to balance the spatial proximity and textual relevancy. A nice property of the unified function is that if both $Sdist(u, v)$ and $Tdist(u, v)$ satisfy the triangle inequality, the triangle inequality also holds for $\mathsf{Ascore}(u, v)$. Throughout this paper, we consistently consider that a smaller attribute score is better. Based on the attribute score of two vertices, we define the attribute score for a subgraph as follows.

Definition 3. (Attribute Score). *Given a subgraph $H \subseteq G$, the attribute score of H, denoted as $\mathsf{Ascore}(H)$, is the maximum attribute score of two vertices in H, i.e.,* $\mathsf{Ascore}(H) = \max_{u,v \in V_H} \mathsf{Ascore}(u, v)$.

Take the attributed graph in Figure 1 as an example again, assume that we consider the user's hobbies and employ the Jaccard distance to compute the attribute score. We can observe that Betty and Gerry share two common hobbies. Thus $\mathsf{Ascore}(Betty, Gerry) = 1 - \frac{2}{6} = 0.667$. By calculating the attribute score between every pair of users, we can get $\mathsf{Ascore}(H_1) = 1$ and $\mathsf{Ascore}(H_2) = 0.833$.

C. Problem Formulation

Based on the definitions of structure cohesiveness and attribute cohesiveness, we formulate the problem of vertex-centric attributed community (VAC) search as follows.

Problem 1. (VAC-Problem). *Given an attributed graph $G = (V_G, E_G, A_G)$, a query set of vertices $Q \subseteq V_G$, a parameter k, the problem of vertex-centric attributed community search returns a subgraph $H \subseteq G$ satisfying the following properties:*

- ***Query Participation.*** *$Q \subseteq V_H$;*

- ***Structure Cohesiveness.*** *H is a connected k-truss, i.e., $\forall e \in E_H$, $sup_H(e) \geq k-2$;*
- ***Attribute Cohesiveness.*** *H has the smallest attribute score;*
- ***Maximality.*** *There does not exist another subgraph $H' \supseteq H$ satisfying the above three properties.*

Back to Figure 1, and assume that we take Betty as query vertex and set $k=4$. As $\mathsf{Ascore}(H_2) < \mathsf{Ascore}(H_1)$ and there is no 4-truss subgraph whose attribute score is smaller than H_2, H_2 is the optimal result for the VAC-problem.

IV. Problem Analysis

In this section, we analyze the VAC-problem from two aspects, i.e., the hardness and free rider effect.

A. Hardness

We show that the VAC-problem is NP-hard in this section. To this end, we define the *decision version* of the VAC-problem and prove its decision problem is NP-hard.

Problem 2. (δVAC-Problem). *Given an attributed graph G, a set of query vertices $Q \subseteq V_G$, two parameters k and δ, the δ*VAC-problem *is testing whether there exists a subgraph $H \subseteq G$ as a maximal connected k-truss containing Q such that* $\mathsf{Ascore}(H) \leq \delta$.

Theorem 1. *The δ*VAC-problem *is NP-Hard.*

Proof. We reduce a well-known NP-hard problem of Maximum Clique (decision version) to the δVAC-problem. Given a simple graph $G' = (V_{G'}, E_{G'})$ and a parameter k', the decision version of the Maximum Clique problem is to check whether G' contains a clique of size k'.

To prove the hardness, we construct an instance of attributed graph $G = (V_G, E_G, A_G)$ from G', using the Jaccard distance for attribute score. First, we copy G from G' and then add one dummy vertex v_q into G. Second, we add the edges between v_q and every vertex $u \in V_{G'}$ into G. Thus, $V_G = V_{G'} \cup \{v_q\}$ and $E_G = E_{G'} \cup \{(v_q, u) : u \in V_{G'}\}$. Third, for each vertex $v \in V_G$, we assign the attribute $A(v) = \{e | e = (v,u) \in E_G\}$. For two vertices v, u, the attribute score $\mathsf{Ascore}(u,v)$ is defined as the Jaccard distance between $A(v)$ and $A(u)$, i.e., $\mathsf{Ascore}(u,v) = 1 - \frac{|A(u) \cap A(v)|}{|A(u) \cup A(v)|}$. We set the input parameters of the δVAC-problem on G as $Q = \{v_q\}$, $k = k'+1$, and $\delta = 1 - \frac{1}{2|V_G|}$. In the following, we show that the instance of Maximum Clique decision problem is a Yes-instance iff the corresponding instance of δVAC-problem is a Yes-instance.

($\Rightarrow$) Assume that H is a k'-clique in G'. The corresponding subgraph of G induced by vertices $V_H \cup \{v_q\}$ is denoted as H^*. H^* is a $(k'+1)$-clique and a connected k-truss containing v_q. For every pair of vertices v, u in H^*, $A(u) \cap A(v) = \{(v,u)\}$, indicating $\mathsf{Ascore}(u,v) \leq 1 - \frac{1}{2|V_G|} = \delta$. Thus, there exists an answer is a Yes-instance of the δVAC-problem.

($\Leftarrow$) Assume that H is an answer to the δVAC-problem. H^* is the corresponding subgraph of G' induced by the vertices $V_H - \{v_q\}$. For any two vertices $u, v \in V_{H^*}$, $\mathsf{Ascore}(u,v) \leq \delta$, meaning that $(u,v) \in E_{H^*}$. Hence, H^* is a clique. Moreover, H is a connected k-truss, indicating $|V_H| \geq k$ and $|V_{H^*}| \geq k-1 = k'$. Thus, H^* is a k'-clique and a Yes-instance of the Maximum Clique decision problem.

In the same way, we can also prove the instance that we add multiple dummy vertices into G when constructing the attributed graph. The detailed proof is omitted due to space limitations. □

B. Avoiding Free Rider Effect

The free rider effect is an undesirable phenomenon for community search [1]. That is, some cohesive structure, irrelevant to the query vertices, could be included in the community answer. It has been shown that many models suffer from the free rider effect such as the core-based model, quasi-clique based model, and local modularity model [24]. Following [6], we formally define the free rider effect as follows.

Definition 4. (Free Rider Effect). *Given a community goodness metric $f(\cdot)$ (the smaller, the better). Let H and H^* be solutions of community search based on $f(\cdot)$ for the queries $Q \neq \varnothing$ and $Q = \varnothing$, respectively. If $f(H \cup H^*) \leq f(H)$, we say that the community search based on $f(\cdot)$ suffers from free rider effect.*

For $Q = \varnothing$, the VAC-problem finds a maximal query-independent connected k-truss $H^* \subseteq G$ such that $\mathsf{Ascore}(H)$ is minimized. We have the following theorem.

Theorem 2. *Let H and H^* be the discovered communities of the* VAC-problem *with $Q \neq \varnothing$ and $Q = \varnothing$, respectively. If $H \neq H^*$ and $H \cup H^*$ is connected,* $\mathsf{Ascore}(H \cup H^*) > \mathsf{Ascore}(H)$.

Proof. We prove it by contradiction. Assume that $\mathsf{Ascore}(H \cup H^*) \leq \mathsf{Ascore}(H)$. As H and H^* are the discovered communities of the VAC-problem with $Q \neq \varnothing$ and $Q = \varnothing$, both H and H^* are connected k-trusses. If $H \cup H^*$ is connected, it is obvious that $H \cup H^*$ is also a connected k-truss. Moreover, $H \neq H \cup H^*$ due to $H \neq H^*$. In addition, $\mathsf{Ascore}(H \cup H^*) \leq \mathsf{Ascore}(H)$, $H \cup H^*$ could be a subgraph of the result of the VAC-problem with $Q \neq \varnothing$. This contradicts the *maximality* of the VAC-problem, because H is the result of the VAC-problem with $Q \neq \varnothing$ and $H \subset H \cup H^*$. Therefore, the assumption $\mathsf{Ascore}(H \cup H^*) \leq \mathsf{Ascore}(H)$ does not hold, but $\mathsf{Ascore}(H \cup H^*) > \mathsf{Ascore}(H)$ holds. □

Note that, if $H = H^*$ holds, $H = H \cup H^*$ is still the result of the VAC-problem; if $H \cup H^*$ is disconnected, $H \cup H^*$ is not a qualified answer of the VAC-problem. Therefore, according to Theorem 2, we conclude that the VAC-problem can avoid the free rider effect.

V. Exact Algorithms

In this section, we first introduce an overview framework of exact algorithms. Then, we present two exact approaches for the VAC-problem. One approach enumerates all candidate subgraphs in a depth-first-search (DFS) manner; the other enumerates all candidate subgraphs in a best-first-search (BFS)

Algorithm 1 A Framework of Exact Algorithms

```
Input: attributed graph $G$; query vertex set $Q$; integer $k$
Output: a VAC $H^*$ with the smallest attribute score
1: $T_k \leftarrow$ compute the maximal $k$-truss of $G$ [4];
2: $\mathcal{H} \leftarrow$ compute all the connected $k$-trusses containing $Q$ based on $T_k$;
3: Compute $\mathsf{Ascore}(H)$ for each connected $k$-truss $H \in \mathcal{H}$;
4: $H^* \leftarrow \arg\min_{H\in\mathcal{H}} \mathsf{Ascore}(H)$;
5: return $H^*$;
```

manner. Several heuristic strategies are proposed to speed up the search process of both algorithms.

A. An Overview Framework of Exact Algorithms

Let us firstly consider a naive exact algorithm. One straightforward method for the VAC-problem is to enumerate all the k-trusses containing the query set Q. Then, it returns an answer of maximal k-truss with the smallest attribute score among all candidate k-trusses. To find all the k-trusses, we need to enumerate all the possible subgraphs of G, which has a total of $O(2^n)$ subgraphs in worst. In addition, for each subgraph, it takes $O(m^{1.5})$ and $O(n^2 \times |A_G|)$ time to compute the k-truss and attribute score, respectively. As a result, the overall time complexity of this naive algorithm is $O(2^n(m^{1.5}+n^2\times|A_G|))$. Obviously, it is extremely time consuming. Recall that, according to the connectivity constraint, all the connected k-trusses are the subgraphs of the maximal k-truss [6]. By exploiting this property of k-truss, the enumeration of subgraphs only need to start from the maximal k-truss rather than the entire graph G. This reduces the search space substantially.

Algorithm 1 describes our framework of exact algorithms. First, the algorithm computes the maximal k-truss T_k of graph G, which invokes a simple k-truss index-based method [4]. Then, Algorithm 1 enumerates all connected k-trusses H containing Q, and computes the attribute score $\mathsf{Ascore}(H)$ (lines 2-3). Finally, a connected k-truss H^* with the smallest attribute score is returned as the answer (lines 4-5). Assume that m_k and n_k are the number of edges and vertices of T_k, respectively. The time complexity of Algorithm 1 is $O(2^{n_k}({m_k}^{1.5}+{n_k}^2\times|A_G|))$, where $n_k << n$ and $m_k << m$ for large k.

B. DFS-based VAC Algorithm

In this section, we propose an exact algorithm for the VAC-problem based on DFS. We develop several pruning strategies to improve the efficiency of Algorithm 1.

The limitation of Algorithm 1 is that it needs to enumerate all the subgraphs of the maximal k-truss. The enumeration process can be represented as a binary search tree. In the search tree, each non-leaf node has two branches. One branch considers to add a candidate vertex to the subgraph; the other deletes the candidate vertex. Each leaf node represents a subgraph of the maximal connected k-truss containing Q. For example, Figure 2(b) shows a binary search tree for a graph G in Figure 2(a). For the non-leaf node N_1, its left branch adds vertex q to the subgraph; its right branch deletes q. The leaf node N_7 denotes the subgraph of G induced by vertices $\{q, v_1, v_2, v_3, v_4\}$, i.e., the whole maximal k-truss G. Actually, the whole tree has many unnecessary subgraphs as infeasible answers. For instance, our community should contain the query vertex q. Thus, the left branch of N_1 can be safely pruned.

To address the above limitation, we propose a DFS-based VAC algorithm for traversing the binary search tree, which prunes infeasible and unnecessary answers as many as possible. For efficient search, we maintain additional information in every search tree node, including C, M, and H. Specifically, C denotes the set of candidate vertices that may be considered as answers; M is the set of vertices that are selected from C, which should be in the result; and H is the maximal connected k-truss of $C \cup M$. In addition, we dynamically maintain the best answer H^*, which has the smallest attribute score currently. For simplicity, we define two branches of a non-leaf node as *add branch* and *delete branch* below.

- **Add branch:** a vertex $v \in C$ is deleted from C *and* added into M.
- **Delete branch**: a vertex $v \in C$ is deleted from C *but not* added into M.

In the following, we present several heuristic strategies for pruning search space. We develop three kinds of rules including *add branch rules*, *delete branch rules*, and *vertex selection order*.

Add Branch Rules. For the add branch, a vertex v is added to M and deleted from C. Note that H in the add branch does not change as $M + C$ remains unchanged. For simplicity, we use $\mathsf{Ascore}(v + M)$ to represent the attribute score of the induced subgraph of the maximal k-truss by vertices $\{v\} \cup M$. We have the following pruning rules for the add branch.

Rule 1. *If* $\mathsf{Ascore}(v+M) > \mathsf{Ascore}(H^*)$*, the add branch can be pruned.*

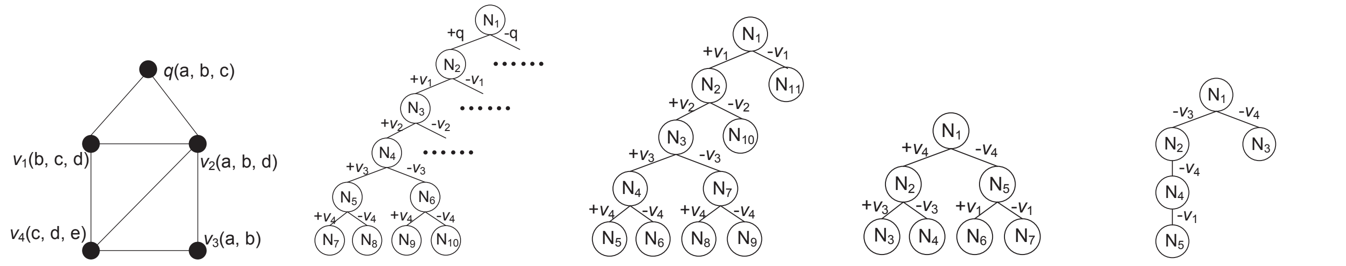
If $\mathsf{Ascore}(v + M) > \mathsf{Ascore}(H^*)$, the subgraph containing $v+M$ has larger attribute score than that of the current optimal k-truss H^*. Thus, $v + M$ cannot become a better answer and the add branch can be pruned.

Rule 2. *If* $\mathsf{Ascore}(v + M) = \mathsf{Ascore}(H^*)$ *and* $|C + M| < |V_{H^*}|$*, the add branch can be pruned.*

If $\mathsf{Ascore}(v + M) = \mathsf{Ascore}(H^*)$, the subgraph composed by $|v + M|$ has the attribute score no smaller than that of the current optimal k-truss H^*. If $|C+M| < |V_{H^*}|$, the cardinality of the subgraph composed by $|v + M|$ is smaller than that of H^*. Recall that the optimal solution of the VAC-problem is with the minimum attribute score and the maximal structure. Thus, the add branch can be pruned.

Rule 3. *If* $\mathsf{Ascore}(v + M) = \mathsf{Ascore}(H)$*, the add branch can be pruned.*

If $\mathsf{Ascore}(v + M) = \mathsf{Ascore}(H)$, the attribute score and the maximum size of the subgraph composed by $v + M$ are equal to $\mathsf{Ascore}(H)$ and $|V_H|$, respectively, meaning that we can directly identify the k-truss with the minimal attribute score


q(a, b, c)
v1(b, c, d)
v2(a, b, d)
v4(c, d, e)
v3(a, b)
N1 +q -q N2 +v1 -v1 N3 +v2 -v2 N4 +v3 -v3 N5 N6 +v4 -v4 N7 N8 N9 N10
N1 +v1 -v1 N2 N11 +v2 -v2 N3 N10 +v3 -v3 N4 N7 +v4 -v4 N5 N6 N8 N9
N1 +v4 -v4 N2 N5 +v3 -v3 +v1 -v1 N3 N4 N6 N7
N1 -v3 -v4 N2 N3 -v4 N4 -v1 N5

(a) Toy Graph (b) Binary Search Tree (c) DFS-based Search Tree (d) DFS-based Search Tree (e) BFS-based Search Tree

Fig. 2: Illustrations for Exact Algorithms

and the maximal size in this branch and need not traverse to the leaf nodes. Therefore, the add branch can be pruned.

Delete Branch Rules. For the delete branch, a vertex v is deleted from C but it is not added to M. Correspondingly, as $M + C$ changes, we need to maintain the induced subgraph formed by $M + V - v$ as k-truss. Assume that H and H' are the k-trusses of $M + C$ and $M + C - v$, respectively. Next, we present several pruning rules based on H'.

Rule 4. *If $V_H - V_{H'} \neq \varnothing$, $V_H - V_{H'}$ can be deleted from C.*

The vertex set $V_H - V_{H'}$ denotes the vertices that need to be deleted to maintain $M+C-v$ as k-truss. After deleting v, any vertex that violates the k-truss constraint cannot contribute to the final result. Hence, such vertices can be deleted from C.

Rule 5. *If $(V_H - V_{H'}) \cap M \neq \varnothing$, the delete branch can be pruned.*

Recall that M is the vertex set that should be in the subgraph in this branch and $(V_H - V_{H'})$ is the vertex set that should be deleted. If $(V_H - V_{H'}) \cap M \neq \varnothing$, it contradicts to each other. Thus, the vertex v cannot be deleted and the delete branch can be pruned.

Rule 6. *If $\mathsf{Ascore}(M) = \mathsf{Ascore}(H')$, the delete branch can be pruned.*

The rationale of this rule is the same as that of Rule 3. The only difference is that Rule 6 is invoked when $\mathsf{Ascore}(H')$ varies while for Rule 3 the left part is the trigger.

Vertex Selection Order. We make an observation. At each node in the binary search tree, a vertex is selected from C to expand candidate subgraphs. A straightforward method is to select the vertex randomly. However, we find that different orders of vertex selection yield binary search trees with different cardinalities (i.e., the number of tree nodes), which finally influences the performance of the algorithm. For example, Figures 2(c) and 2(d) shows two search trees with different vertex selection orders. The search tree in Figure 2(c) is larger than the tree in Figure 2(d). Intuitively, the smaller size of binary search, the better performance of the algorithm. Thus, we would like to find a vertex selection order to make the binary search tree as small as possible.

To generate a small binary search tree, disqualified nodes should be deleted as early as possible. According to our pruning rules, if we want to prune the add branch as soon as possible, we should select a vertex to maximize $\mathsf{Ascore}(v + M)$. If we want to prune the delete branch as soon as possible, we should select a vertex to maximize $|V_H - V_{H'}|$. However, to compute $|V_H - V_{H'}|$ for every vertex in C is time consuming. Instead, we select a vertex with the largest degree. The rationale behind it is that the larger degree of a vertex, the higher probability that it may be involved in more triangles. Hence, it will have a higher probability to maximize the $|V_H - V_{H'}|$. Based on the above analysis, we present the vertex selection rule.

Rule 7. *When selecting a vertex from C for subgraph expansion, we (i) first select the vertex having the maximum $\mathsf{Ascore}(v+M)$; and (ii) if there are multiple vertices satisfying (i), we select the vertex with the maximum degree among them.*

Algorithm. The DFS-based VAC algorithm is outlined in Algorithm 2. This algorithm integrates all the pruning strategies in Rules 1-7. Algorithm 2 first computes the maximal connected k-truss containing the query vertex set, calculates its attribute score, initializes H^*, C, and M (lines 1-4). Then, the algorithm invokes the procedure *BST* to find the optimal solution by traversing the binary search tree. In procedure of *BST*, if $C = \varnothing$, it means that we arrive at a leaf node. Thus, the procedure updates the optimal solution and stops (lines 6-7). Otherwise, the procedure *BST* selects a vertex to expand according to Rule 7. First, it adds the selected vertex to M and checks the add branch using Rules 1-3 (lines 10-16). Then, it deletes the selected vertex from C and examines the delete branch using Rules 4-6 (lines 17-23).

Complexity Analysis. Let $|T_{DFS}|$ be the cardinality of the binary search tree of the DFS-based VAC algorithm, $h_{T_{DFS}}$ be the height of binary search tree. The space complexity of the DFS-based VAC algorithm is $O(h_{T_{DFS}} \times m_k)$. It is because the algorithm traverses the tree in a depth first manner. Thus, its space complexity is the height of the search tree times the node capacity, which equals $O(h_{T_{DFS}} \times m_k)$. The time complexity of the DFS-based VAC algorithm is $O(|T_{DFS}| \times m_k^{1.5} + n_k^2 \times |A_G|)$. It first computes the maximal k-truss and its corresponding attribute score, whose time complexity is $O(m_k + n_k^2 \times |A_G|)$. Then, the algorithm invokes the procedure *BST* to find the k-truss with the minimum attribute score, whose performance is determined by the cardinality of the search tree. For tree nodes in the delete branch, the algorithm needs to maintain the k-truss, whose time is $O(m_k^{1.5})$. For tree nodes in the add branch, the algorithm just checks the rules, whose time is $O(1)$. Thus, the time complexity of the

Algorithm 2 DFS-based VAC Algorithm

```
Input: attributed graph $G$; query vertex set $Q$; integer $k$
Output: the $k$-truss $H^*$ with smallest attribute score
 1: $H \leftarrow$ compute the maximal $k$-truss of $G$ [4];
 2: Compute the attribute score $\mathsf{Ascore}(H)$;
 3: $H^* \leftarrow H$; $C \leftarrow V_H - Q$; $M \leftarrow Q$;
 4: BST($C$, $M$, $H$, $H^*$);
 5: return $H^*$

    Procedure BST($C$, $M$, $H$, $H^*$)
 6: if $C = \varnothing$ then
 7:     Update $H^*$;
 8: else
 9:     Select a vertex $v \in C$ according to selection strategy;
    //processing add branch
10:     if $\mathsf{Ascore}(M + v) = \mathsf{Ascore}(H)$ then
11:         Update $H^*$;
12:     else
13:         if $\mathsf{Ascore}(M + v) = \mathsf{Ascore}(H) =$
            $\mathsf{Ascore}(H^*)$ then
14:             Update $H^*$;
15:         else if $\mathsf{Ascore}(M + v) < \mathsf{Ascore}(H)$ then
16:             BST($C - v$, $M + v$, $H$, $H^*$);
    //processing delete branch
17:     $H' \leftarrow H - \{v\}$;
18:     Maintain $H'$ as $k$-truss containing $Q$;
19:     if $H' \neq \varnothing$ and $(\{V_H - V_{H'} - v\} \cap M \neq \varnothing)$ then
20:         if $\mathsf{Ascore}(H') = \mathsf{Ascore}(M)$ then
21:             Update $H^*$;
22:         else
23:             BST($C - V_H + V_{H'}$, $M$, $H'$, $H^*$);
```

TABLE I: Illustration of Algorithm 2 on Graph in Figure 2(a).

Node	M	C	H	H^*
N_1	q	v_1, v_2, v_3, v_4	q, v_1, v_2, v_3, v_4	q, v_1, v_2, v_3, v_4
N_2	q, v_4	v_1, v_2, v_3	q, v_1, v_2, v_3, v_4	q, v_1, v_2, v_3, v_4
N_3	q, v_4, v_3	v_1, v_2	q, v_1, v_2, v_3, v_4	q, v_1, v_2, v_3, v_4
N_4	q, v_4	v_1, v_2	q, v_1, v_2, v_4	q, v_1, v_2, v_4
N_5	q	v_1, v_2	q, v_1, v_2	q, v_1, v_2
N_6	q, v_1	v_2	q, v_1, v_2	q, v_1, v_2
N_7	q	v_2	$\varnothing$	q, v_1, v_2

procedure *BST* is $O(|T_{DFS}| \times m_k^{1.5})$. In total, the DFS-based VAC algorithm takes $O(|T_{DFS}| \times m_k^{1.5} + n_k^2 \times |A_G|)$ time.

Example 1. *We use the graph in Figure 2(a) to illustrate the DFS-based VAC algorithm. Assume that q is the query vertex and $k = 3$. Figure 2(d) depicts the search tree of the DFS-based VAC algorithm and Table I shows the corresponding node contents. First, the root node N_1 is initialized using the maximal 3-truss, i.e., the whole graph. Then, the vertex v_4 is selected from the candidate set for processing as it has the maximum $\mathsf{Ascore}(v + M)$. When adding v_4 to M, we arrive the node N_2. Then, we select the vertex v_3 for processing and get nodes N_3 and N_4, which are all pruned according to Rule 3. Next, the algorithm returns to process the delete branch of node N_1 and gets the node N_5, during which we get the 3-truss composed by q, v_1, v_2. The algorithm continues until all nodes are visited. Finally, the 3-truss composed by q, v_1, v_2 is returned as the final result.*

C. BFS-based VAC Algorithm

This section proposes a BFS-based algorithm for the VAC-problem.

Observation. We begin with an important observation. Recall that the VAC-problem is to find the k-truss with the minimum attribute score. Given two k-trusses T_1 and T_2 satisfying $T_2 \subseteq T_1$ and $\mathsf{Ascore}(T_1) > \mathsf{Ascore}(T_2)$, to find T_2 from T_1, from the attribute score's perspective, we must break the relationship of vertices in T_1 having the maximum attribute score. In other words, let vertices u, $v \in T_1$ satisfy $\mathsf{Ascore}(u, v) = \mathsf{Ascore}(T_1)$, to get the k-truss T_2, we must delete at least one of u and v from T_1. Otherwise, we cannot get a k-truss with an attribute score smaller than T_1.

Based on the above observation, we propose the BFS-based VAC algorithm, which searches the answer in a best first manner. The basic idea is that for each candidate k-truss containing Q, we first find the vertex pair (u, v) having the maximal attribute score. Then, we split it into two smaller k-trusses containing Q by deleting the vertices u and v, respectively. The process continues until there is no k-truss containing Q. For the BFS-based VAC algorithm, we want to highlight two points. First, when deleting the vertex contributing to the maximal attribute score, we only delete the vertex which does not belong to the query vertex set. This is obvious as the query vertex set must be contained in the k-truss according to the definition. Second, when we delete the vertex and generate the new smaller k-truss, we will get at most 2 new k-trusses for each original k-truss. As the deletion goes on, there will be numerous k-trusses, which is inefficient. To this end, we employ the following lemma to keep the number of k-trusses as small as possible.

Lemma 1. *Let $\mathcal{H} = \{H_1, H_2, ...\}$ be the set of k-trusses generated by iteratively deleting the vertices having the maximum attribute score. For H_i, $H_j \in \mathcal{H}$, if $H_i \subseteq H_j$, H_i can be safely pruned.*

Proof. We prove the correctness from two following instances:

(i) H_i does not contain k-truss with a smaller attribute score. If $\mathsf{Ascore}(H_i) = \mathsf{Ascore}(H_j)$, H_i can be pruned as $|H_i| < |H_j|$. If $\mathsf{Ascore}(H_i) < \mathsf{Ascore}(H_j)$, we can find another H_j' by iteratively deleting the vertex with the maximal attribute score from H_j such that $\mathsf{Ascore}(H_i) = \mathsf{Ascore}(H_j')$ and $H_i \subseteq H_j'$. Thus, H_i can also be pruned as $|H_i| \leq |H_j'|$.

(ii) H_i contains k-truss H_i' with $\mathsf{Ascore}(H_i') < \mathsf{Ascore}(H_i)$. We can also find a k-truss H_j' by iteratively deleting the vertex with the maximal attribute score from H_j such that $\mathsf{Ascore}(H_j') = \mathsf{Ascore}(H_i')$ and $H_i' \subseteq H_j'$. Obviously, H_j' is a better answer than H_i'. □

Algorithm. Algorithm 3 describes the details of the BFS-based VAC algorithm using Lemma 1 for pruning. The algorithm first computes the maximal k-truss containing the query vertex set and uses it for initialization (lines 1-4). Here, $\mathcal{H}$ is used to store the k-trusses generated by iteratively deleting the pair of vertices having the maximum attribute score. In each iteration, the algorithm invokes the procedure *Delete* on every k-truss in $\mathcal{H}$ to find the k-truss with a smaller attribute score by deleting the vertices having the maximum attribute score respectively (lines 6-10). After all k-trusses are processed, the algorithm applies Lemma 1 to prune the k-truss (lines 11-13). The procedure *Delete* deletes the vertex and maintains the k-truss

Algorithm 3 BFS-based VAC Algorithm

```
Input: attributed graph $G$; query vertex set $Q$; integer $k$
Output: the $k$-truss $H^*$ with smallest attribute score
 1: $H \leftarrow$ compute the maximal $k$-truss of $G$ [4];
 2: Compute the attribute score of $H$;
 3: Push $H$ into $\mathcal{H}$;
 4: $H^* \leftarrow H$;
 5: while $\mathcal{H} \neq \varnothing$ do
 6:     for each $k$-truss $H \in \mathcal{H}$ do
 7:         Let $u$, $v \in V_H$ be the vertex satisfying
            $\mathsf{Ascore}(u, v) = \mathsf{Ascore}(H)$;
 8:         Delete($u$, $H$, $H^*$, $\mathcal{H}$, $Q$);
 9:         Delete($v$, $H$, $H^*$, $\mathcal{H}$, $Q$);
10:         $\mathcal{H} \leftarrow \mathcal{H} - H$;
11:     for $\forall$ $H'$, $H \in \mathcal{H}$ do
12:         if $H' \subseteq H$ then
13:             $\mathcal{H} \leftarrow \mathcal{H} - H'$ ;
14: return $H^*$

    Procedure Delete($v$, $H$, $H^*$, $\mathcal{H}$, $Q$)
15: if $v \notin Q$ then
16:     $H \leftarrow H - v$;
17:     Maintain $H$ as $k$-truss;
18:     if $H \neq \varnothing$ and $Q \subseteq H$ then
19:         if ($\mathsf{Ascore}(H) = \mathsf{Ascore}(H^*)$ and $|V_{H^*}| < |V_H|$)
            or ($\mathsf{Ascore}(H) < \mathsf{Ascore}(H^*)$) then
20:             $H^* \leftarrow H$;
21:         if $\mathsf{Ascore}(Q) \neq \mathsf{Ascore}(H)$ then
22:             Push $H$ into $\mathcal{H}$;
```

after deletion (lines 16-17). Then, the procedure uses the new k-truss to update H^*(lines 19-20). If $\mathsf{Ascore}(Q) = \mathsf{Ascore}(H)$, the new k-truss cannot contain any k-truss with a smaller attribute score, so it is pruned. Otherwise, the new k-truss is inserted into $\mathcal{H}$ for further examination (lines 21-22).

Complexity Analysis. Let $|\mathcal{H}|_{max}$ be the maximum size of $\mathcal{H}$, $|T_{BFS}|$ be the cardinality of the search tree of the BFS-based VAC algorithm. The space and time complexities of the BFS-based VAC algorithm are $O(|\mathcal{H}|_{max} \times m_k)$ and $O(|T_{BFS}| \times m_k^{1.5} + n_k^2 \times |A_G|)$, respectively. The BFS-based VAC algorithm first computes the maximal k-truss and its attribute score, whose time complexity is the same as that of the DFS-based VAC algorithm. Then, for each candidate k-truss in $\mathcal{H}$, the BFS-based VAC algorithm deletes the pair of vertices having the maximum attribute score respectively and generates two new k-trusses. Thus, the space complexity of the BFS-based VAC algorithm is determined by the maximum size of $\mathcal{H}$ and the space complexity is $O(|\mathcal{H}|_{max} \times m_k)$. As each deletion needs to maintain the k-truss, whose time is $O(m_k^{1.5})$. Thus, the time complexity of the BFS-based VAC algorithm is determined by the number of k-trusses enumerated, which is equal to the cardinality of the search tree of the BFS-based VAC algorithm. Hence, it is $O(|T_{BFS}| \times m_k^{1.5} + n_k^2 \times |A_G|)$.

Example 2. *We also illustrate the BFS-based VAC algorithm using the graph in Figure 2(a). Figure 2(e) shows its search tree. N_1 is initialized with the maximal 3-truss. Then, we select the vertex pair with the maximum attribute score, i.e., (v_3, v_4). After deleting v_3 and v_4 respectively, we get two 3-trusses $N_2 = \{q, v_1, v_2, v_4\}$ and $N_3 = \{q, v_1, v_2\}$. Then, the two 3-trusses are used to update H^*, i.e., $H^* = \{q, v_1, v_2\}$. As $N_3 \subset N_2$, the node N_3 is pruned. Next, the node N_2 is processed. The algorithm continues until no 3-truss exists. Finally, the 3-truss composed by $\{q, v_1, v_2\}$ is returned as the optimal solution, which is consistent with the result of the DFS-based VAC algorithm.*

D. Extension to Other Community Models

In this section, we discuss how to extend the two exact algorithms to other dense subgraph-based community models, e.g., k-core, k-edge-connected component, and k-clique. Take k-core as an example. Recall that both the exact and approximate algorithms proposed in our work follow a filter-and-refinement framework, i.e., they first find an upper bound (i.e., maximal k-truss) of the result and then refine it to obtain the final result. We can slightly modify these two phases to make our algorithms work for the k-core model. More specifically, for the filter phase, we should compute the maximal connected k-core containing the query vertices; for the refinement phase, the updating algorithm should maintain the remaining graph as a connected k-core after removal of vertices and edges.

VI. Approximate Algorithm

In this section, we first analyze the non-approximability of the VAC-$\mathsf{problem}$. Then, we introduce an approximate algorithm for the VAC-$\mathsf{problem}$. We also give a detailed analysis of approximation and complexity for the proposed algorithm.

A. Non-Approximability

We start with a definition of approximation ratio. Assume that H^* and H are the results of exact and approximate algorithms for the VAC-$\mathsf{problem}$, respectively. For $\alpha \geq 1$, we say that an approximate algorithm achieves α-approximation to the optimal answer H^* or within an approximate factor of α, if and only if, (1) H contains Q; (2) H is a connected k-truss; and (3) $\mathsf{Ascore}(H) \leq \alpha \cdot \mathsf{Ascore}(H^*)$. In other words, the approximation ratio α is defined on the attribute score of communities.

We next prove the non-approximability. We establish this result of non-approximability through a reduction from NP-hard problems. Again, we reduce the Maximum Clique decision problem to the approximation of the VAC-$\mathsf{problem}$. Recall that, to prove the hardness of the VAC-$\mathsf{problem}$, we construct an attributed graph G from a simple graph G' in Theorem 1. Without loss of generality, assume that $|V_G| = n$ and $n \geq 3$.

Theorem 3. *Given an attributed graph G in Theorem 1 and a number $k \geq 2$, the k-clique $H \subseteq G$, $\mathsf{Ascore}(H) \leq \frac{2n-4}{2n-3}$.*

Proof. For $v, u \in V_H$, if there is an edge between u and v, then $\mathsf{Ascore}(u, v) = 1 - \frac{A(u) \cap A(v)}{A(u) \cup A(v)} = 1 - \frac{1}{deg_G(u) + deg_G(v) - 1} \leq 1 - \frac{1}{2n-3} = \frac{2n-4}{2n-3}$. Therefore, $\mathsf{Ascore}(H) = \max_{u,v \in V_H} \mathsf{Ascore}(u, v) \leq \frac{2n-4}{2n-3}$. □

Theorem 4. *Unless $P = NP$, for any $\varepsilon > 0$, the* VAC-$\mathsf{problem}$ *cannot be approximated in polynomial time within a factor $(\frac{2n-3}{2n-4} - \varepsilon)$.*

Proof. Assume that there exists an approximate algorithm $\mathbb{A}$ which returns a polynomial-time solution H with an approximation factor $(\frac{2n-3}{2n-4} - \varepsilon)$ with respect to the optimal solution H^*. Then, we have $\mathsf{Ascore}(H) \leq (\frac{2n-3}{2n-4} - \varepsilon) \cdot \mathsf{Ascore}(H^*)$.

Next, we employ this approximate solution to exactly solve the Maximum Clique decision problem as follows. If $\mathsf{Ascore}(H) < 1$, the optimal solution H^* has $\mathsf{Ascore}(H^*) < \mathsf{Ascore}(H) < 1$, meaning that every pair of vertices in the simple graph G' has an edge. Thus, G' has a clique of size $k-|Q|$. If $\mathsf{Ascore}(H) = 1$, we have $(\frac{2n-3}{2n-4})\cdot\mathsf{Ascore}(H^*) > (\frac{2n-3}{2n-4}-\varepsilon)\cdot \mathsf{Ascore}(H^*) \geq \mathsf{Ascore}(H) = 1$. Thus, $\mathsf{Ascore}(H^*) > \frac{2n-4}{2n-3}$, which is in contradiction to Theorem 3. Hence, G cannot contain a k-truss. Combining the above two instances, we can claim that G contains a clique of size k if and only if the approximate algorithm $\mathbb{A}$ can return a solution H with $\mathsf{Ascore}(H) < 1$. However, the Maximum Clique decision problem cannot be solved in polynomial time unless $P = NP$. Therefore, the theorem is proved. □

Therefore, we prove that the VAC-problem cannot be approximated within a factor better than $\frac{2n-3}{2n-4}$. The prospects for an effective α-approximate algorithm is very challenging, especially for a small value of α. In view of this result, we propose a 2-approximation algorithm in the next section.

B. 2-Approximation Algorithm

Due to the non-approximability within a factor of $\frac{2n-3}{2n-4}$ where $1 < \frac{2n-3}{2n-4} < 2$, we present an approximate algorithm that computes the solution in a greedy manner. We start by computing the query attribute scores. For a vertex $v \in V_G$, the query attribute score of v, denoted as $\mathsf{Ascore}(v, Q)$, is the maximum attribute score of v and $q \in Q$, i.e., $\mathsf{Ascore}(v, Q) = \max_{q\in Q} \mathsf{Ascore}(v, q)$.

The basic idea of the approximate algorithm is as follows. It first computes the maximal k-truss containing the query vertex set. Then, it iteratively deletes the vertices with the largest query attribute score. Meanwhile, it maintains the remaining graph as a connected k-truss containing Q. Algorithm 4 outlines the details of the 2-approximation algorithm. Specifically, it first computes the maximal connected k-truss H containing Q for a given parameter k from G (line 1). For each vertex v in H, Algorithm 4 computes the query attribute score $\mathsf{Ascore}(v, Q)$ (line 2). It then selects one vertex v with the maximum query attribute score (line 5). Next, the algorithm deletes v and its incident edges (line 6) and maintains a connected k-truss, by iteratively removing the disconnected nodes and edges whose support is less than k (line 7). The algorithm repeats the above step until there exists no connected k-truss containing Q. Finally, the last feasible connected k-truss H^* is returned as the answer (line 8). In addition, the approximate algorithm can be easily extended to other community models with only two modifications, just like the exact algorithms discussed in Section V-D.

Example 3. *We again use the graph in Figure 2(a) to illustrate the approximate VAC algorithm. First, H^* is initialized by the whole graph. Then, v_4 is selected for deleting as $\mathsf{Ascore}(v_4, q)$ is maximum. After deleting v_4, we get a new k-truss composed by $\{q, v_1, v_2\}$, which is used to update H^*. Next, v_1 is selected for deletion. After deleting v_1, there does not exist k-truss. Thus, the algorithm stops and the k-truss composed by $\{q, v_1, v_2\}$ is returned, which is the same as the results of exact algorithms.*

Algorithm 4 Approximate VAC Algorithm

```
Input: attributed graph $G$; query vertex set $Q$; integer $k$
Output: vertex-centric attributed community $H^*$
1: $H \leftarrow$ compute the maximal $k$-truss of $G$ [4];
2: Computing $\mathsf{Ascore}(v, Q)$ for every vertex $v \in V_H$;
3: while $Q \subseteq V_H$ do
4:     $H^* \leftarrow H$;
5:     Let $v$ be a vertex with the maximum $\mathsf{Ascore}(v, Q)$;
6:     Delete vertex $v$ and its incident edges from $H$;
7:     Maintain $H$ as a connected $k$-truss containing $Q$ by
         removing the disqualified vertices/edges from $H$;
8: return $H^*$
```

C. Approximation and Complexity Analysis

We show that Algorithm 4 can achieve 2-approximation to the optimal solution. Let H and H^* be the solutions returned by Algorithm 4 and exact algorithms, respectively.

Lemma 2. $\mathsf{Ascore}(H, Q) \leq \mathsf{Ascore}(H^*, Q)$*, where* $\mathsf{Ascore}(H, Q) = \max_{v\in V_H} \mathsf{Ascore}(v, Q)$.

Proof. As shown in Algorithm 4, both H and H^* are the subgraphs of the maximal connected k-truss. In the following, we prove the lemma by considering two possible relationships between H and H^*.

(1) $H^* \subseteq H$. We can find a vertex $v \in V_H$, which has the maximum query attribute score in both H and H^*. This is because if $v \in H$ but $v \notin H^*$, we can get a new k-truss after deleting v from H, which is in contradiction to the approximate algorithm. As $H^* \subseteq H$, $\mathsf{Ascore}(v, Q) = \mathsf{Ascore}(H^*, Q)$. Thus, $\mathsf{Ascore}(H, Q) = \mathsf{Ascore}(H^*, Q)$.

(2) $H \nsubseteq H^*$. Let H_M be the maximal k-truss. Then, we have the following two relationships: (i) $H \subseteq ... \subseteq H_{i+1} \subseteq H_i \subseteq ... \subseteq H_M$, where H_i is the intermediate k-truss generated by the approximate algorithm; (ii) $H^* \subseteq H_M$. We can find a H_i such that $H^* \subseteq H_i$ and $H^* \nsubseteq H_{i+1}$. Also, let the vertex $v \in V_{H_i}$ satisfying that v has the maximum query attribute score. We can get that v also belongs to H^*. This is because if v is not contained in H_E, $H^* \subseteq H_{i+1}$. It contradicts our assumption. Hence, $\mathsf{Ascore}(H^*, Q) = \mathsf{Ascore}(H_i, Q) \geq \mathsf{Ascore}(H_{i+1}, Q) \geq \mathsf{Ascore}(H, Q)$.

Combining above two instances, we prove Lemma 2. □

Theorem 5. *Algorithm 4 returns a 2-approximation solution for the* VAC-problem.

Proof. Assume that $\mathsf{Ascore}(H^*) = \mathsf{Ascore}(v_1, v_2)$ and $\mathsf{Ascore}(H^*, Q) = \mathsf{Ascore}(v_3, Q)$, where $v_1, v_2, v_3 \in V_{H^*}$. Since we employ the distance metric to compute the attribute score, it satisfies the triangle inequality. In other words, $\mathsf{Ascore}(H^*) = \mathsf{Ascore}(v_1, v_2) \leq \mathsf{Ascore}(v_1, Q) + \mathsf{Ascore}(v_2, Q) \leq 2\mathsf{Ascore}(v_3, Q) = 2\mathsf{Ascore}(H^*, Q)$, which also holds for H. Combining with Lemma 2, we can infer that $\frac{\mathsf{Ascore}(H)}{2} \leq \mathsf{Ascore}(H, Q) \leq \mathsf{Ascore}(H^*, Q) \leq \mathsf{Ascore}(H^*)$. As a result, $\mathsf{Ascore}(H) \leq 2\mathsf{Ascore}(H^*)$. □

TABLE II: Network statistics ($\mathbf{K}$ = 10^3 and $\mathbf{M}$ = 10^6)

Dataset	$\|V\|$	$\|E\|$	d_{max}	k_{max}	$\|A\|$	$\|A(v)\|$
Facebook	1.9**K**	8.9**K**	416	96	576	12
DBLP	41.3**K**	210.3**K**	152	35	29	2
Brightkite	58.2**K**	214.1**K**	567	43	116**K**	2
Gowalla	196.6**K**	950.3**K**	7365	29	393**K**	2
Tencent	1.3**M**	9.6**M**	126870	39	255**K**	7
Poket	1.6**M**	22.3**M**	7427	29	232**K**	4

Complexity Analysis. Let l be the number of iterations in Algorithm 4. The space and time complexities of Algorithm 4 are $O(m_k)$ and $O(l \times m_k^{1.5})$, respectively. As Algorithm 4 only keeps a single k-truss of G, the space complexity is $O(m_k)$. For each iteration, Algorithm 4 needs to maintain a connected k-truss. The overall time complexity of is $O(l \times m_k^{1.5})$.

VII. Experiments

In this section, we experimentally evaluate our proposed 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++.

A. Experimental Settings

Datasets. We use six real-world attributed graphs, which are often applied to evaluate the methods of attributed community analysis [10], [11], [33], in the experiments. Table II summarizes the statistics of the six attributed graphs. Facebook, DBLP, Pokec, and Tencent[1] are textual attributed graphs, in which we adopt Jaccard distance to calculate attribute scores. Specifically, Facebook, Pokec, and Tencent are social networks. The attributes of a vertex are the user's features extracted from the profiles. Note that Facebook contains the ground-truth communities. DBLP is a co-authorship graph built from the DBLP digital library.[2] The vertex attributes are the conferences or journals, such as VLDB, SIGMOD, ICDE, where the author has published at least one paper. Brightkite and Gowalla are two location attributed graphs. For these two networks, we employ Euclidean distance to compute the attribute score. Note that Facebook, Brightkite, Gowalla, and Pokec are all downloaded from SNAP.[3] For the sake of space limit, we only report our results on Facebook and Gowalla. Similar results can be found on other datasets.

Queries and Parameters. We evaluate our model and algorithms by varying various parameter settings, including the query size $|Q|$, the parameter k, and the graph cardinality $|V_G|$. In each experiment, we run 200 queries for attributed community search on all datasets and report the average results. For each query, we randomly select a set of query vertices Q from a graph.

B. Model Evaluation on Ground-truth Communities

Algorithms and Metrics. To evaluate the effectiveness of the vertex-centric community search on attributed graphs, we compare our exact algorithm in Algorithm 3 (ETruss) and approximate algorithm in Algorithm 4 (ATruss) with existing truss-based community search methods, i.e., attribute-driven community search (ATC) [11] and k-truss community search (KCS) [4]. Moreover, we extend our exact and approximate algorithms to the k-core-based community model, denoted by ECore and ACore, respectively. To measure the quality of communities, we use the quality metrics of precision, community structure similarity (CSS), and community attribute similarity (CAS). In particular, assume that H is the community discovered by algorithms and $\widehat{H}$ is the ground-truth community, then

$$precision(H) = \frac{|V_H \cap V_{\widehat{H}}|}{|V_H|}$$

$$CSS(H) = \frac{1}{|V_H|^2} \sum_{u \in V_H} \sum_{v \in V_H} \frac{N(u) \cap N(v)}{N(u) \cup N(v)}$$

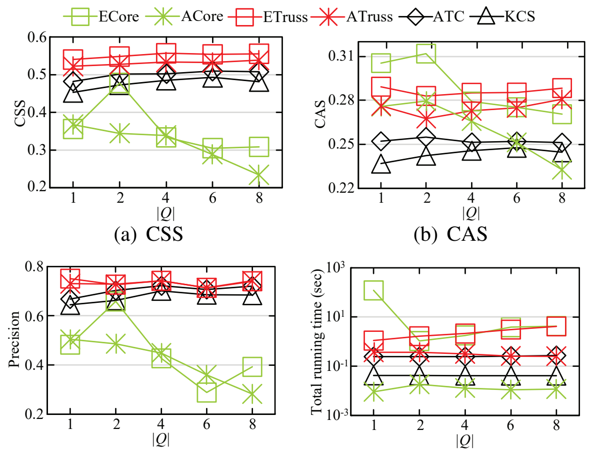
$$CAS(H) = \frac{1}{|V_H|^2} \sum_{u \in V_H} \sum_{v \in V_H} \frac{A(u) \cap A(v)}{A(u) \cup A(v)}$$

In addition, we report the running time for all algorithms.

Exp-1: The effect of $|Q|$ on the effectiveness and efficiency. We vary the number of query vertices $|Q|$ from 1 to 8 to evaluate the effectiveness and efficiency of all methods. Figures 3(a)-(d) show the results of all methods on the Facebook dataset. Figures 3(a)-(b) show the results of CSS and CAS. As expected, ETruss outperforms ATruss. Both of them perform better than both ACT and KCS in all cases tested, meaning that the communities returned by our model are more cohesive than those of ACT and KCS in terms of both topology structure and community attributes. Note that ECore and ACore achieve the worst CSS performance among all methods. Figure 3(c) shows the precision results. The ETruss and ATruss methods again perform better than the others, indicating that the communities returned by our model are more similar to the ground-truth. Figure 3(d) shows the runtime results. The two exact algorithms ETruss and ECore take the longest time. This is because they need to explore the whole search space of k-truss/k-core enumerations. Among all methods, ACore takes the least time due to the easy computation of k-core.

Exp-2: The effect of k on the effectiveness and efficiency. Next, we evaluate the effectiveness and efficiency of all methods by varying parameter k from 8 to 20. Figures 4(a)-(d) report the results on the Facebook dataset. Again, our methods ETruss and ATruss have the best performance among all in terms of CSS, CAS, and precision. When k increases, we find that both CSS and CAS decrease for the methods ETruss, ATruss, ECore, and ACore. This is because the maximal k-truss/k-core found in the first step of the algorithms becomes smaller with the increased k. As a result, those vertices that contribute to high cohesiveness are missing from the answers. Thus, CSS and CAS decrease. It is interesting to note that the trends of precision are not stable in both Figures 3(c) and 4(c). This is because k takes different optimal values for different query vertices.

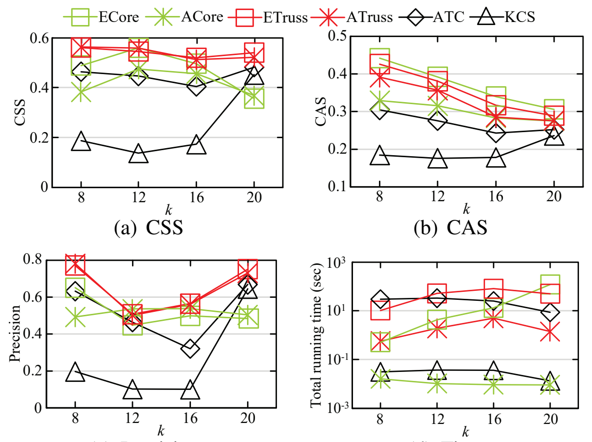
[1]http://www.kddcup2012.org/c/kddcup2012-track1
[2]https://dblp.uni-trier.de/xml/
[3]http://snap.stanford.edu/



(a) CSS (b) CAS

(c) Precision (d) Runtime

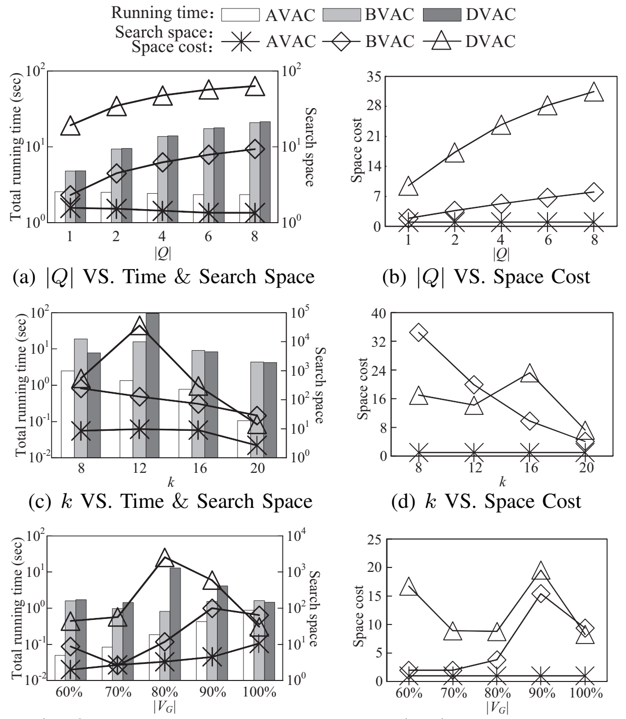
Fig. 3: Model Evaluation on Facebook for $|Q|$



(a) CSS (b) CAS

(c) Precision (d) Time

Fig. 4: Model Evaluation on Facebook for k



(a) $|Q|$ VS. Time & Search Space (b) $|Q|$ VS. Space Cost

(c) k VS. Time & Search Space (d) k VS. Space Cost

(e) $|V_G|$ VS. Time & Search Space (f) $|V_G|$ VS. Space Cost

Fig. 5: Efficiency Evaluation of VAC Algorithms on Gowalla

C. Efficiency Evaluation

Algorithms and Metrics. In this section, we evaluate the efficiency of our proposed algorithms, i.e., DFS-based VAC algorithm in Algorithm 2 (denoted by DVAC), BFS-based VAC algorithm in Algorithm 3 (denoted by BVAC), and approximate VAC algorithm in Algorithm 4 (denoted by AVAC). Note that the method in Algorithm 1 cannot complete the community search task within a week, which is not reported here. We report three metrics for each algorithm, i.e., the running time, the size of search space, and the space cost. For the size of search space, we calculate (i) the cardinality of the search tree for DVAC and BVAC and (ii) the number of *while* iterations for AVAC. For the space cost, we report (i) the height of search tree for DVAC, (ii) the maximum size of heap $\mathcal{H}$ for BVAC, and (iii) the number of stored k-trusses for AVAC.

Exp-3: The effect of $|Q|$ on efficiency. We first evaluate the efficiency by varying the number of query vertices $|Q|$ from 1 to 8. Figures 5(a) and 5(b) show the results. As expected, the performance of AVAC is better than that of DVAC and BVAC. Although DVAC has larger search space than BVAC, DVAC and BVAC almost have the same running time. This is because in the search tree, BVAC needs to invoke the k-truss maintenance operation for every node while DVAC only needs to apply the k-truss maintenance operation for the delete branch. Although DVAC has larger search space than BVAC, BVAC invokes less k-truss maintenance operations, leading to less running time. When the number of query vertices $|Q|$ grows, the performance of DVAC and BVAC degrades while the performance of AVAC becomes a little bit better. The reason behind is that with the increased $|Q|$, the maximal k-truss returned in the first step becomes smaller and the returned community has larger attribute score. If the maximal k-truss becomes small, the iterations for AVAC also decrease. If the returned community has a larger attribute score, the pruning ability of DVAC and BVAC becomes weaker, leading to degradation of DVAC and BVAC and more space cost.

Exp-4: The effect of k on efficiency. Figures 5(c) and 5(d) report the results when the parameter k is varied from 8 to 20. With the increasing of k, all three algorithms achieve better performance. This is because when k grows, we will find a smaller maximal k-truss in the first step, resulting in smaller search space. Thus, all algorithms require less time to find the optimal results. Correspondingly, the space cost of DVAC and BVAC also becomes less. As AVAC only needs to keep one k-truss, its space cost remains unchanged. In addition, there are some exceptions, e.g., the search space of DAVA is very large under $k = 12$. This is due to some outliers in the corresponding results.

Exp-5: The effect of graph cardinality $|V_G|$ on efficiency. Finally, we evaluate the scalability of our algorithms with different fractions of vertices extracted from the original dataset. The results are plotted in Figures 5(e) and 5(f). There are some fluctuations for DVAC and BVAC while AVAC has an upward trend. This is because if the cardinality grows, the maximal k-truss becomes larger. Thus, AVAC requires more time. For DVAC and BVAC, the attribute score of optimal community influences the pruning ability of algorithms a lot. When the cardinality changes, the attribute score of the optimal community can becomes larger or smaller, which is uncertain. Thus, both DVAC and BVAC fluctuate.

VIII. Conclusions

In this paper, we study a new attributed community search problem called vertex-centric community search, which can handle different types of attributes. To answer the VAC problem, we propose two exact algorithms with depth first search and best first search, respectively. Due to the hardness of the problem, we also develop an approximate algorithm, which can achieve 2-approximation to the optimal solution. Extensive experimental evaluation validates the effectiveness and efficiency of the proposed model and algorithms.

Acknowledgments. This work is supported by Research Grants Council of Hong Kong under GRF Projects 12201018, 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. Qing Liu and Yifan Zhu have contributed equally to this work. Yunjun Gao is the corresponding author.

References

[1] M. Sozio and A. Gionis, "The community-search problem and how to plan a successful cocktail party," in *SIGKDD*, 2010, pp. 939–948.

[2] W. Cui, Y. Xiao, H. Wang, and W. Wang, "Local search of communities in large graphs," in *SIGMOD*, 2014, pp. 991–1002.

[3] N. Barbieri, F. Bonchi, E. Galimberti, and F. Gullo, "Efficient and effective community search," *Data Min. Knowl. Discov.*, vol. 29, no. 5, pp. 1406–1433, 2015.

[4] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu, "Querying k-truss community in large and dynamic graphs," in *SIGMOD*, 2014, pp. 1311–1322.

[5] E. Akbas and P. Zhao, "Truss-based community search: a truss-equivalence based indexing approach," *PVLDB*, vol. 10, no. 11, pp. 1298–1309, 2017.

[6] X. Huang, L. V. S. Lakshmanan, J. X. Yu, and H. Cheng, "Approximate closest community search in networks," *PVLDB*, vol. 9, no. 4, pp. 276–287, 2015.

[7] C. E. Tsourakakis, F. Bonchi, A. Gionis, F. Gullo, and M. A. Tsiarli, "Denser than the densest subgraph: extracting optimal quasi-cliques with quality guarantees," in *SIGKDD*, 2013, pp. 104–112.

[8] W. Cui, Y. Xiao, H. Wang, Y. Lu, and W. Wang, "Online search of overlapping communities," in *SIGMOD*, 2013, pp. 277–288.

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

[10] Y. Fang, R. Cheng, Y. Chen, S. Luo, and J. Hu, "Effective and efficient attributed community search," *VLDB J.*, vol. 26, no. 6, pp. 803–828, 2017.

[11] X. Huang and L. V. S. Lakshmanan, "Attribute-driven community search," *PVLDB*, vol. 10, no. 9, pp. 949–960, 2017.

[12] Z. Wang, Y. Yuan, G. Wang, H. Qin, and Y. Ma, "An effective method for community search in large directed attributed graphs," in *MSN*, 2017, pp. 237–251.

[13] Q. Zhu, H. Hu, C. Xu, J. Xu, and W. Lee, "Geo-social group queries with minimum acquaintance constraints," *VLDB J.*, vol. 26, no. 5, pp. 709–727, 2017.

[14] Y. Fang, R. Cheng, X. Li, S. Luo, and J. Hu, "Effective community search over large spatial graphs," *PVLDB*, vol. 10, no. 6, pp. 709–720, 2017.

[15] L. Chen, C. Liu, R. Zhou, J. Li, X. Yang, and B. Wang, "Maximum co-located community search in large scale social networks," *PVLDB*, vol. 11, no. 10, pp. 1233–1246, 2018.

[16] Y. Fang, Z. Wang, R. Cheng, X. Li, S. Luo, J. Hu, and X. Chen, "On spatial-aware community search," *IEEE Trans. Knowl. Data Eng.*, vol. 31, no. 4, pp. 783–798, 2019.

[17] X. Huang, L. V. S. Lakshmanan, and J. Xu, "Community search over big graphs: Models, algorithms, and opportunities," in *ICDE*, 2017, pp. 1451–1454.

[18] X. Huang, L. V. Lakshmanan, and J. Xu, *Community Search over Big Graphs*. Morgan & Claypool Publishers, 2019.

[19] Y. Fang, X. Huang, L. Qin, Y. Zhang, W. Zhang, R. Cheng, and X. Lin, "A survey of community search over big graphs," *VLDB J.*, vol. 29, no. 1, pp. 353–392, 2020.

[20] L. Yuan, L. Qin, W. Zhang, L. Chang, and J. Yang, "Index-based densest clique percolation community search in networks," *IEEE Trans. Knowl. Data Eng.*, vol. 30, no. 5, pp. 922–935, 2018.

[21] Y. Wang, X. Jian, Z. Yang, and J. Li, "Query optimal k-plex based community in graphs," *Data Science and Engineering*, vol. 2, no. 4, pp. 257–273, 2017.

[22] L. Chang, X. Lin, L. Qin, J. X. Yu, and W. Zhang, "Index-based optimal algorithms for computing steiner components with maximum connectivity," in *SIGMOD*, 2015, pp. 459–474.

[23] J. Hu, X. Wu, R. Cheng, S. Luo, and Y. Fang, "On minimal steiner maximum-connected subgraph queries," *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 11, pp. 2455–2469, 2017.

[24] Y. Wu, R. Jin, J. Li, and X. Zhang, "Robust local community detection: On free rider effect and its elimination," *PVLDB*, vol. 8, no. 7, pp. 798–809, 2015.

[25] Y. Fang, Z. Wang, R. Cheng, H. Wang, and J. Hu, "Effective and efficient community search over large directed graphs," *IEEE Trans. Knowl. Data Eng.*, vol. 31, no. 11, pp. 2093–2107, 2019.

[26] Q. Liu, M. Zhao, X. Huang, J. Xu, and Y. Gao, "Truss-based community search over large directed graphs," To appear in SIGMOD 2020.

[27] Y. Chen, Y. Fang, R. Cheng, Y. Li, X. Chen, and J. Zhang, "Exploring communities in large profiled graphs," *IEEE Trans. Knowl. Data Eng.*, vol. 31, no. 8, pp. 1624–1629, 2019.

[28] S. Ebadian and X. Huang, "Fast algorithm for k-truss discovery on public-private graphs," in *IJCAI*, 2019, pp. 2258–2264.

[29] Y. Zhu, Q. Zhang, L. Qin, L. Chang, and J. X. Yu, "Querying cohesive subgraphs by keywords," in *ICDE*, 2018, pp. 1324–1327.

[30] L. Chen, C. Liu, K. Liao, J. Li, and R. Zhou, "Contextual community search over large social networks," in *ICDE*, 2019, pp. 88–99.

[31] Z. Zhang, X. Huang, J. Xu, B. Choi, and Z. Shang, "Keyword-centric community search," in *ICDE*, 2019, pp. 422–433.

[32] J. Yang, J. J. McAuley, and J. Leskovec, "Community detection in networks with node attributes," in *ICDM*, 2013, pp. 1151–1156.

[33] F. Zhang, Y. Zhang, L. Qin, W. Zhang, and X. Lin, "When engagement meets similarity: Efficient (k, r)-core computation on social networks," *PVLDB*, vol. 10, no. 10, pp. 998–1009, 2017.

[34] Y. Li, C. Sha, X. Huang, and Y. Zhang, "Community detection in attributed graphs: An embedding approach," in *AAAI*, 2018, pp. 338–345.

[35] C. Zhe, A. Sun, and X. Xiao, "Community detection on large complex attribute network," in *SIGKDD*, 2019, pp. 2041–2049.

[36] D. Jin, Z. Liu, W. Li, D. He, and W. Zhang, "Graph convolutional networks meet markov random fields: Semi-supervised community detection in attribute networks," in *AAAI*, 2019, pp. 152–159.

[37] C. Bothorel, J. D. Cruz, M. Magnani, and B. Micenková, "Clustering attributed graphs: Models, measures and methods," *Network Science*, vol. 3, no. 3, pp. 408–444, 2015.

[38] S. Pool, F. Bonchi, and M. van Leeuwen, "Description-driven community detection," *ACM TIST*, vol. 5, no. 2, pp. 28:1–28:28, 2014.

[39] R. Kanawati, "Seed-centric approaches for community detection in complex networks," in *International Conference on Social Computing and Social Media*, 2014, pp. 197–208.

[40] H. Sun, H. Du, J. Huang, Z. Sun, L. He, X. Jia, and Z. Zhao, "Detecting semantic-based communities in node-attributed graphs," *Computational Intelligence*, vol. 34, no. 4, pp. 1199–1222, 2018.

[41] J. Cohen, "Trusses: Cohesive subgraphs for social network analysis," *National security agency technical report*, vol. 16, pp. 3–1, 2008.

[42] S. Kosub, "A note on the triangle inequality for the jaccard distance," *Pattern Recognition Letters*, vol. 120, pp. 36–38, 2019.