When Structure Meets Keywords: Cohesive Attributed Community Search

Yuanyuan Zhu, Jian He, Junhao Ye
School of Computer Science, Wuhan University
{yyzhu,2017282110267,junhao_ye}@whu.edu.cn

Lu Qin
Centre of AI, University of Technology, Sydney
lu.qin@uts.edu.au

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

Jeffrey Xu Yu
The Chinese University of Hong Kong, Hong Kong, China
yu@se.cuhk.edu.hk

ABSTRACT

As an online, query-dependent variant of the well-known *community detection* problem, *community search* has been studied for years to find communities containing the query vertices. Along with the generation of graphs with rich attribute information, *attributed community search* has attracted increasing interest recently, aiming to select communities where vertices are cohesively connected and share homogeneous attributes. However, existing community models may include cut-edges/vertices and thus cannot well guarantee the strong connectivity required by a cohesive community. In this paper, we propose a new *cohesive attributed community* (CAC) model that can ensure both *structure* cohesiveness and *attribute* cohesiveness of communities. Specifically, for a query with vertex v_q and keyword set S, we aim to find the cohesively connected communities containing v_q with the most shared keywords in S. It is nontrivial as we need to explore all possible subsets of S to verify the existence of structure cohesive communities until we find the communities with the most common keywords. To tackle this problem, we make efforts in two aspects. The first is to reduce the candidate keyword subsets. We achieve this by exploring the anti-monotonicity and neighborhood-constraint properties of our CAC model so that we can filter out the unpromising keyword subsets. The second is to speed up the verification process for each candidate keyword subset. We propose two indexes TIndex and MTIndex to reduce the size of the candidate subgraph before the verification. Moreover, we derive two new properties based on these indexes to reduce the candidate keyword subsets further. We conducted extensive experimental studies on four real-world graphs and validated the effectiveness and efficiency of our approaches.

KEYWORDS

attributed graphs; community search; truss model

CIKM '20, October 19–23, 2020, Virtual Event, Ireland

ACM ISBN 978-1-4503-6859-9/20/10…$15.00
https://doi.org/10.1145/3340531.3412006

ACM Reference Format:
Yuanyuan Zhu, Jian He, Junhao Ye, Lu Qin, Xin Huang, and Jeffrey Xu Yu. 2020. When Structure Meets Keywords: Cohesive Attributed Community Search. In *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM '20), October 19–23, 2020, Virtual Event, Ireland*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3340531.3412006

1 INTRODUCTION

Graph is a powerful model to represent complex structural relationships among objects, which has been widely used in real-world applications. In most of these applications, communities naturally exist within which vertices are densely interconnected. Existing studies on communities mainly fall into two categories: *community detection* [10] [28] [18] to find all the communities in a network, which has been extensively studied for decades; *community search* [13] [17] [24] to find the communities containing the query vertices, which has attracted increasing attention from researchers recently.

Motivation. In this paper, we study *community search* in attributed graphs, to find the *structure* cohesive and *attribute* cohesive communities for a given query, in which vertices are densely connected and share homogeneous attributes. Such attributed communities naturally exist in many real-world graphs with rich attribute information such as social networks, collaboration networks, and biological networks. Finding such communities can help to capture the properties of the targeted vertices. Recently, two models [9] [14] have been proposed to depict the attributed community. [9] finds k-core communities in which vertices have degrees at least k and share the most attributes. However, k-core cannot give any guarantee on the connectivity of a community, which means even deleting one edge may disconnect the community. In other words, cut-edges may exist in the community. Subsequently, [14] finds (k,d)-truss communities with the maximum attribute score based on a higher-order graph motif, triangle, rather than primitive vertices/edges. A (k,d)-truss is a subgraph such that each edge is contained in at least $k-2$ triangles, and the distance from each vertex to the query vertex is no more than d. It it is also a $(k-1)$-core and a $(k-1)$-edge connected subgraph [7], i.e., all vertices have degree at least $k-1$ and remain connected when less than $k-1$ edges are removed. However, this problem is NP-hard, and due to the non-monotonicity and non-submodular properties of the attribute score, the approximate ratio cannot be guaranteed by the heuristic solution in [14]. Moreover, even with the distance constraint d, the structure cohesiveness of the (k,d)-truss may still be

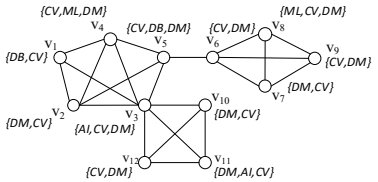


Figure 1: An example of attributed graph

weak as one vertex deletion could also disconnect the community, i.e., cut-vertices may exist in the community no matter how large k is. We illustrate such limitations by the following example.

Example 1. *Consider an example collaboration network in Fig. 1. For a query specified by vertex v_4 and keyword set $S = \{DM, CV, ML\}$, if $k = 3$, subgraph H_1 induced by $\{v_2, v_3, \ldots, v_{12}\}$ with two common attributes DM and CV will be returned in the core based community search [9]. H_1 can be easily disconnected if we delete the cut-edge (v_5, v_6). In the (k,d)-truss community model [14], for $k = 4$ and $d = 2$, subgraph H_2 induced by $\{v_2, v_3, v_4, v_5, v_{10}, v_{11}, v_{12}\}$ with the largest attribute score is returned. However, H_2 can also be disconnected if we remove the cut-vertex v_3. In fact, vertices v_{10}, v_{11}, and v_{12} are not cohesively connected with v_4 due to this cut-vertex.*

From the above example, we can see that the existence of cut-edges/vertices violates the requirements of a cohesive community [3]: vertices are well reachable with short distance and good connectivity. Thus, in this paper, we revisit the attributed community search problem and propose a cut-edge/vertex free *cohesive attributed community* (CAC) model to find both *structure* cohesive and *attribute* cohesive community. We ensure the structure cohesiveness by the triangle connected k-truss model in which any two edges either belong to the same triangle or are reachable from each other through a series of adjacent triangles [13] [1]. Here two triangles are adjacent if they share a common edge. This model can well capture the characteristic of short distance and good connectivity of cohesive communities, because cut-edge/vertex can be successfully avoided (see proof in Section 2) and the diameter is well bounded by $\lfloor \frac{2n-2}{k} \rfloor$ for a k-truss with n vertices [7]. Benefiting from the triangle connectivity constraint, such a community usually has a much smaller n compared with a connected k-truss community in practice. We ensure the attribute cohesiveness by the number of common attributes in the community.

Example 2. *For the example data in Fig. 1, given a query with vertex v_4 and keyword set $\{DM, CV, ML\}$, if $k = 4$, subgraph H_3 induced by $\{v_2, v_3, v_4, v_5\}$ with two common attributes DM and CV will be returned by our CAC model, which includes no cut-edge/vertex. Vertices v_{10}, v_{11}, and v_{12} are excluded from this community as their adjacent edges cannot be triangle connected with vertex v_4.*

Challenges. Finding cohesive attributed communities in large graphs is very challenging. A straightforward solution is to enumerate all the keyword combinations and return the triangle connected k-truss communities with the most shared keywords. However, the number of keyword subsets to enumerate can be as large as $2^l - 1$ for l keywords. It is impractical for large attributed graphs in an online manner, given the fact that $2^l - 1$ can be up to 1 million even for $l = 20$ and the complexity of each triangle connected k-truss community search is $O(m^{1.5})$ where m is the number of edges in the candidate subgraph to verify [13]. Although $O(m^{1.5})$ can be reduced to the output size if an appropriate index is equipped for non-attributed graph (e.g., EquiTruss [1]), it is also impractical to build such index for every keyword subset due to the costly truss decomposition involved in each index construction. It is also impractical to build index for a candidate subgraph induced by a keyword subset and dynamically maintain it for other keyword subsets, because the subgraphs induced by different keyword subsets vary substantially and the maintenance costs even more than reconstruction from scratch for a small portion of change for most real-world datasets [1] [30]. Note that the techniques used community search based on k-core [9] and k-truss with no triangle connectivity constraint [14] also cannot solve our problem due to the inherent problem difference.

Contributions. We tackle the cohesive attributed community search problem by making efforts in two aspects. The first is to reduce the number of verifications of the triangle connected k-truss community by filtering out unpromising keyword subsets, and the second is to accelerate the verification process by reducing the size of the candidate subgraph. To reduce the number of verifications, we explore two properties of our CAC model. The first is the anti-monotonicity property, i.e., if there is no triangle connected k-truss community for a keyword subset, we can avoid the verification of its superset directly. The second is the neighborhood-constraint property so that we can filter out the unpromising keyword subsets directly without the triangle connected k-truss community verification. To accelerate the verification process, we develop a new index TIndex by utilizing the trussness and attribute information. Moreover, based on TIndex, we further derive two new properties, neighborhood-trussness constraint and neighborhood-disjoint constraint, to filter out more keyword subsets. We also propose an improved index MTIndex to further reduce the number of verifications and accelerate the verification process, and develop the incremental and decremental search algorithms based on MTIndex for different kinds of graphs. In summary, the main contributions of this paper are:

- We study the CAC search problem to find triangle connected k-truss communities with the most shared attributes for the first time to capture both the structure and attribute cohesiveness, which is also solvable in polynomial time.
- We explore four properties of the CAC model and develop a new index TIndex to prune unpromising keyword subsets and accelerate the verification process.
- We propose an improved index MTIndex to further reduce the number of verifications and accelerate the verification process, and develop two efficient incremental and decremental search algorithms for different kinds for graphs.
- We conducted extensive experimental studies on real-world datasets, and validated the efficiency and effectiveness of our models and algorithms.

Roadmap. Section 2 formulates the problem. Section 3 presents a basic solution. Section 4 gives a new index and new search algorithms. Section 5 presents the improved index and algorithms. Section 6 discusses the experimental results. Section 7 discusses the related work and Section 8 concludes this paper.

2 PROBLEM STATEMENT

Given a set of attributes Σ, a simple undirected attributed graph is represented as $G = (V, E, A)$, where V is the vertex set, $E \subseteq V \times V$

is the edge set, and A is an attribute function which assigns each vertex a set of attributes $A(v) \subseteq \Sigma$. For a graph G, we use $V(G)$ and $E(G)$ to denote vertex set and edge set, and use $|V(G)|$ and $|E(G)|$ to denote vertex number and edge number. For a vertex v in G, let $N_G(v) = \{u \in V | (u,v) \in E\}$ denote its neighborhood and $d_G(v) = |N_G(v)|$ denote its degree. A *triangle* $\triangle_{uvw}$ is a substructure such that $(u,v),(v,w),(u,w) \in E$. The support of an edge $e = (u,v)$ in graph G is the number of triangles containing e, defined as $sup_G(e) = |\{\triangle_{uvw} | w \in V(G)\}|$. In the following, we use $sup(e)$, $N(v)$, and $d(v)$ instead of $sup_G(e)$, $N_G(v)$, and $d_G(v)$ for simplification if the context is clear.

Definition 1. **(k-Truss)** *A connected k-truss in G is a connected subgraph $H \subseteq G$, such that $\forall e \in E(H)$, $sup_H(e) \geq k-2$.*

The trussness of a subgraph $H \subseteq G$ is the minimum support of all the edges in H plus 2, defined as $\tau(H) = \min_{e \in E(H)} sup_H(e) + 2$. The trussness of an edge $e \in E(G)$ is the maximum trussness of subgraphs containing e, i.e., $\tau(e) = \max_{H \subseteq G \wedge e \in E(H)} \tau(H)$. Consider a k-truss $H \subseteq G$. If there exists no supergraph H' of H ($H \subseteq H' \subseteq G$) such that $\tau(H') = \tau(H)$, we call H a *maximal k-truss*.

However, a k-truss can be easily disconnected by a possible cut-vertex, as shown in Example 1. Thus, *triangle connectivity* is further exploited to model the real-world communities [13] [1]. A triangle $\triangle_{uvw}$ is called a *k-triangle*, if the trussness of each constituent edge is no less than k. Given two k-triangles $\triangle_1$ and $\triangle_2$, they are adjacent if they share a common edge, i.e., $\triangle_1 \cap \triangle_2 \neq \emptyset$. $\triangle_s$ and $\triangle_t$ are *k-triangle connected*, denoted as $\triangle_s \overset{k}{\leftrightarrow} \triangle_t$, if there exists a sequence of k-triangles $\triangle_1$,...,$\triangle_n$ ($n \geq 2$) such that $\triangle_s = \triangle_1$, $\triangle_t = \triangle_n$, and for $1 \leq i < n$, $\triangle_i \cap \triangle_{i+1} \neq \emptyset$. Analogously, two edges $e, e' \in E(G)$ are *k-triangle connected*, denoted as $e' \overset{k}{\leftrightarrow} e$ iff (1) e and e' belong to the same k-triangle, or (2) $e \in \triangle_s$, $e' \in \triangle_t$ such that $\triangle_s \overset{k}{\leftrightarrow} \triangle_t$.

Definition 2. **(Triangle Connected k-Truss Community)** *A subgraph $H \subseteq G$ is a triangle connected k-truss community if it satisfies (1) H is a maximal k-truss and (2) $\forall e, e' \in E(H)$, $e \leftrightarrow e'$. We also simplify it as k-truss community if the context is clear.*

Example 3. *Consider the graph in Fig. 1. The support of edge (v_2, v_3) is 3 as it occurs in three triangles $\triangle_{v_2v_3v_1}$, $\triangle_{v_2v_3v_4}$, and $\triangle_{v_2v_3v_5}$. The subgraph H_2 induced by $\{v_2, v_3, v_4, v_5, v_{10}, v_{11}, v_{12}\}$ is a maximal 4-truss, as the minimum support for edges in H_2 is at least 2, i.e., $\tau(H_4) = 2$. Clearly, $\tau(v_2, v_3) = 4$ as there is no subgraph with trussness larger than 4 containing (v_2, v_3). However, H_2 is not a triangle connected 4-truss community, as (v_2, v_3) and (v_3, v_{11}) are not triangle connected. Clearly, the subgraph H_4 induced by $\{v_1, v_2, v_3, v_4, v_5\}$ is a triangle connected 4-truss community.*

Lemma 1. *There is no cut-edge and cut-vertex in the triangle connected k-truss community.*

Proof: Obviously, a triangle connected k-truss community has no cut-edge as it is at least $(k-1)$-edge connected as proved in [7]. Next, we prove that it also has no cut-vertex. Assume that there is a cut-vertex c. Then there exist two neighboring vertices of c, a and b, which are disconnected after deleting c. In other words, there is only one path $a \rightarrow c \rightarrow b$ from a and b, and thus (a,c) and (c,b) are not triangle connected, which contradicts with the definition of triangle connected k-truss community. □.

Algorithm 1: CAC-Basic (G, v_q, S, k)

Input : A graph G, a vertex v_q, a keyword set S, and integer k
Output: All attributed truss communities containing v_q

```
generate F_S = {Ψ_1, Ψ_2, ..., Ψ_h} using S and N(v_q);
R ← ∅; i ← h;
while i ≥ 1 do
    for each S' ∈ Ψ_i do
        find the connected subgraph G[S'] containing v_q from G;
        R ← findKTrussCom (G[S'], v_q, k);
        if R ≠ ∅ then R ← R ∪ R;
    if R = ∅ then i ← i − 1;
    else break;
Output the communities in R;
```

Cohesive Attributed Community (CAC) Search Problem. Given an attributed graph $G = (V, E, A)$, an integer $k \geq 3$, a vertex $v_q \in V$ and a set of keywords S, cohesive attributed community search is to return a set $\mathcal{R}$, such that $\forall H \in \mathcal{R}$ satisfies the following conditions: (1) H is triangle connected k-truss community containing v_q; (2) The size of shared keywords $L(H,S)$ is maximal, where $L(H,S) = \cap_{v \in V(H)}(A(v) \cap S)$ is the set of attributes shared in S by all vertices in H.

Example 4. *Based on the CAC model, subgraph H_3 induced by $\{v_2, v_3, v_4, v_5\}$ is returned with $\tau(H_3) = 4$ and $L(H_3, S) = 2$ for v_4 and $S = \{DM, CV, ML\}$ in Fig. 1. Note that H_4 induced by $\{v_1, v_2, v_3, v_4, v_5\}$ with $\tau(H_4) = 4$ is not returned as a CAC, because $L(H_4, S) = 1 < 2$.*

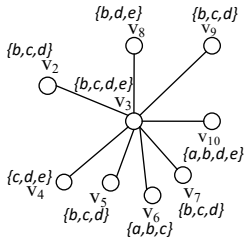
3 BASIC APPROACHES

For cohesive attributed community search, a straightforward method is to enumerate all the keyword subsets to check whether a k-truss community exists and return the communities with the most common attributes. Specifically, given a query vertex v_q and a keyword set S, we enumerate all non-empty subsets of S, $S_1, S_2, \ldots, S_{2^l-1}$. For each subset $S_i \subseteq S$ ($1 \leq i \leq 2^l - 1$), we check whether there exists any k-truss community containing v_q in which all the vertices contain S_i. If such a community exists, we denote it as $H[S_i]$. The k-truss communities with the most common keywords will be returned as CAC. However, repeating k-truss communities search $2^l - 1$ times is impractical to support the online search over large attributed graphs. Thus, in this section, we propose a new method that can reduce the number of keyword subsets to verify. Before getting into the details of the algorithms, we first discuss the anti-monotonicity property of our CAC model.

Property 1. **(Anti-Monotonicity)** *Given a graph G, a vertex v_q and a keyword subset S', if there exists a triangle connected k-truss community $H[S']$ such that each vertex containS', then $\forall S'' \subseteq S'$ there exists a triangle connected k-truss community $H[S''] \supseteq H[S']$.*

This property can be easily derived from the definition of triangle connected k-truss community. Based on this property, if $H[S']$ does not exist, we can stop the examination of all its supersets.

Then we can derive a basic apriori solution from bottom to up as follows. First, we divide the keyword subsets of S into l groups, $\Psi_1, \Psi_2, \ldots, \Psi_l$, where each set $S' \in \Psi_i$ is a size-i keyword subset with i keywords. We start examine each $S' \in \Psi_1$ with only one keyword to check whether the triangle connected k-truss community $H[S']$ exists. After we examined Ψ_i, we only examine a size-$(i+1)$



Groups	Keyword subsets
Ψ_1	{b},{c},{d},{e}
Ψ_2	{b, c},{b, d},{c, d},{d, e}
Ψ_3	{b, c, d}

Figure 2: An example of the frequent keyword subsets

keyword subset if its size-i subsets all have k-truss communities. However, in such a process, we still need to check many keyword subsets that do not have a k-truss community. Then, a question naturally arises: can we filter out unpromising keyword subsets before we do the k-truss community search? We answer this question by exploiting the neighborhood constraint property of the k-truss community as follows.

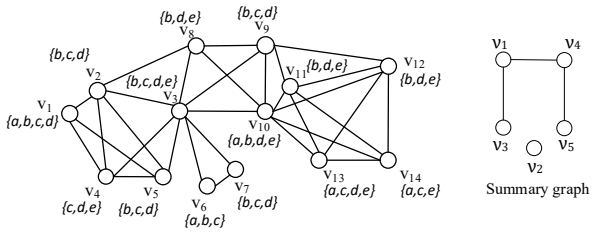
Property 2. **(Neighborhood Constraint)** *For a query node v_q, if a k-truss community $H[S']$ exists for $S' \subseteq A(v_q)$, then v_q has at least $k-1$ neighbors containing S'.*

This property can be derived from the fact that a k-truss community $H[S']$ is also a $(k-1)$-core. Thus, each vertex in $H[S']$ must have degree at least $k-1$. Based on this property, we can generate the candidate keyword subsets from v_q and its neighbors by applying the well studied frequent pattern mining algorithms, e.g., FP-Growth [11], to find the frequent keyword subsets with support at least $k-1$ instead of examining all the keyword subsets.

Example 5. *Consider a query with vertex v_3, keyword set $S = \{b, c, d, e\}$, and trussness $k = 4$. v_3 has 8 neighbors as shown in Fig. 2. We can generate 9 candidate keyword sets by FP-Growth as shown in the table on the right of Fig. 2, which is less than the number of enumerated keyword subsets $2^4 - 1 = 15$.*

Even with the neighborhood constraint, we still need to check many keyword subsets in the bottom-up method until we find a k-truss community with the most common attributes. Another possible way is to examine the keyword subsets from top to down, i.e., start from the keyword subsets with the largest number of keywords, which is usually much faster since the candidate subgraph to verify is much smaller under a larger keyword subset constraint.

The basic top-down method is shown in Algorithm 1. First, we generate the candidate keyword subsets by FP-Growth [11] with support $k-1$ and group them into $h \le l$ groups, denoted as $\mathcal{F}_S = \{\Psi_1, \Psi_2, \ldots, \Psi_h\}$ (line 1). Then we initialize the set of cohesive attributed communities as $\mathcal{R} = \emptyset$ and the size of keyword subset as $i = h$. For each keyword subset $S' \in \Psi_i$, we first find the subgraph $G[S']$ containing S', and then find the triangle connected k-truss communities R containing v_q from $G[S']$ by calling function findKTrussCom (lines 5-6). If the $R \neq \emptyset$, we add it to the result set $\mathcal{R}$ (line 7). If $\mathcal{R}$ is empty after we check all the keyword sets in Ψ_i, we decrease i by 1 and move on to Ψ_{i-1} ; otherwise we terminate the algorithm (lines 8-9). The details of function findKTrussCom can be found in [13] with complexity $O(|E(G[S'])|^{1.5})$. Thus the overall time complexity of CAC-Basic is $O(n_s|E(G'_{max})|^{1.5})$ where n_s is the number of candidate keyword subsets, and G'_{max} is the maximum candidate subgraph.



Super-vertex	Included edges	Attributes
$\nu_1(k=3)$	(2, 8)	b, d
$\nu_2(k=3)$	(3, 6),(3, 7),(6, 7)	b, c, d
$\nu_3(k=4)$	(1, 2),(1, 4),(1, 5),(2, 3),(2, 4), (2, 5),(3, 4),(3, 5),(4, 5)	b, c, d, e
$\nu_4(k=4)$	(3, 8),(3, 9),(3, 10),(8, 9),(8, 10), (9, 10),(9, 11),(9, 12)	b, c, d, e
$\nu_5(k=5)$	(10, 11),(10, 12),(10, 13),(10, 14),(11, 12), (11, 13),(11, 14),(12, 13),(12, 14),(13, 14)	a, b, c, d, e

Figure 3: An example of TIndex

4 ADVANCED APPROACHES

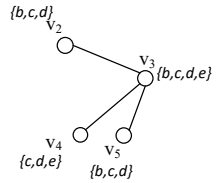
In the CAC-Basic algorithm, for each keyword subset S', we need to find k-truss community from the candidate subgraph $G[S']$ with $O(|E(G[S'])|^{1.5})$ time, which is very time-consuming. In fact, many vertices in the candidate subgraph are not qualified for a k-truss community. Thus, we propose a new TIndex to reduce the size of the candidate subgraphs. By exploiting the property of such index, we can also further reduce the number of frequent keyword subsets.

4.1 A New Index TIndex

Our index is inspired by the extreme case of the anti-monotonicity property, where $S'' \subseteq S'$ is $\emptyset$. If we denote a k-truss community without keyword constraint as H°, then for any triangle connected k-truss community containing $H[S']$, we have $H[S'] \subseteq H^\circ$. Thus, we can actually verify the k-truss community on a smaller candidate subgraph $H^\circ \cap G[S']$ instead of $G[S']$, i.e., finding the k-truss communities H° first and then finding the keyword induced subgraph $G^\circ[S']$ in H°. Thus, we need to compute H° before we do any verification for the keyword subsets.

Directly searching H° from G will need $O(|E(G)|^{1.5})$ time. Fortunately, it can be reduced to the output size with the assistance of EquiTruss [1]. EquiTruss ultlizes *k-truss equivalence* to characterize the strong connection between edges within a k-truss community. Two *k-triangle connected* edges $e, e' \in E(G)$ are *k-truss equivalent* if $\tau(e) = \tau(e') = k$, denoted as $e' \overset{k}{=} e$. The *equivalence class* of an edge $e \in E(G)$ is denoted $C_e = \{e' | e' \overset{k}{=} e, e' \in E(G)\}$. The set of all equivalence classes forms a mutually exclusive and collectively exhaustive partition of $E(G)$ with $k \ge 3$, and it can be considered as the set of super-vertices in a summary graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$. A super-edge $(\nu, \mu) \in \mathcal{E}$ indicates that $\nu, \mu \in \mathcal{V}$ are k-triangle connected ($k = \min\{\tau(\nu), \tau(\mu)\}$), i.e., for any $e \in \nu$, $\exists e' \in \mu$ such that $e \overset{k}{\leftrightarrow} e'$. Such summary graph is indexed as EquiTruss. Based on EquiTruss, for a query vertex v_q and trussness k, we can search H° directly on $\mathcal{G}$. Specifically, we first find the set of super-vertices which contain v_q with trussness at least k, denoted as $\mathcal{V}'$. Then for each super-vertex $\nu \in \mathcal{V}'$, we start the bread-first search by only including super-vertices μ with trussness at least k.

However, in the above process, there may exist super-vertices in which each vertex does not contain any attributes in S and thus will not be included in the final triangle connected k-truss communities for any keyword subset. Thus, we build a new index TIndex to



Group	Keyword sets
Ψ_1^1	$\{c\},\{d\}$
Ψ_2^1	$\{c,d\}$

Group	Keyword sets
Ψ_1^2	$\{b\},\{d\}$
Ψ_2^2	$\{b,d\}$

Figure 4: An example of disjoint frequent keyword subsets

further reduce the size of H°. The main idea is that besides the summary graph, we also keep a set of keyword for each super-vertex to support the early stop. For each edge $(u,v) \in E(G)$, we use $A(u,v) = A(u) \cap A(v)$ to denote the edge attribute which shared by vertices u and v. Then attributes for super-vertex ν are defined as the union of all the edge attributes in ν, i.e., $A(\nu) = \cup_{(u,v)\in\nu} A(u,v)$. Thus we have an attributed summary graph that can keep the truss equivalence information and attribute information. Besides, we also keep two auxiliary structures. One is the vertex inverted list to keep the super-vertex IDs that contain this vertex. The other is the keyword inverted list to keep the set of vertex IDs V_i containing keyword w_i. These structures constitute our new index TIndex.

Example 6. *Consider an example graph G and its summary $\mathcal{G}$ in Fig. 3. The details of the super-vertices are shown in the table at the bottom. Despite sharing the same trussness 3, $(2,8)$ and $\{(3,6),(3,7)(6,7)\}$ belong to two different super-vertices v_1 and v_2 as they are not triangle connected. Super-vertex v_3 represents a triangle connected 4-truss with 9 edges, and its union set of the shared attributes is $\{b,c,d,e\}$.*

Based on TIndex, we can find all the k-truss communities containing v_q in linear time [1]. First of all, we find the set of super-vertices which contains v_q with trussness at least k, denoted as $\mathcal{V}'_k$. Then for each super-vertex $\nu \in \mathcal{V}'_k$, we start the bread-first search by only including super-vertices μ with trussness at least k and $A(\mu) \cap S \neq \emptyset$ to find k-truss communities.

Example 7. *Consider the graph in Fig. 3. For a query with $v_q = v_3$ and $S = \{b,c,d,e\}$, based on the summary graph, we can find communities H_1° (super-vertices v_3) and H_2° (super-vertices v_4 and v_5).*

4.2 TIndex Based Search Algorithms

Based on TIndex, we not only can reduce the size of the candidate subgraph but also can further reduce the number of candidate keyword sets. From the property of k-truss community, we can see that an edge with trussness less than k will never be included in a k-truss community. Thus, we can further derive the following property of k-truss communities.

Property 3. **(Neighborhood-Trussness Constraint)** *For a query node v_q, if a k-truss community $H[S']$ exists for $S' \subseteq A(v_q)$, then v_q has at least $k-1$ neighbors containing S' and any adjacent edge between such neighbor and v_q has trussness at least k.*

Example 8. *Now we reexamine the example in Fig. 2. If $k = 4$, the keywords in v_6 will never contribute to the final CAC result as the trussness of (v_3, v_6) is smaller than 4. Thus, before we generate the candidate keyword subsets by FP-Growth, we can filter out v_6 first.*

As analyzed before, a query vertex v_q might be included in multiple super-vertices with trussness at least k. Based on these super-vertices, we may find multiple k-truss communities as illustrated in Example 7. For two super-vertices with the same trussness k, we have the following lemma.

Lemma 2. *Given a query vertex v_q and an integer k, any two super-vertices v_1, v_2 containing v_q with the same trussness $k' \geq k$ will belong to two different k-truss communities.*

Proof: We prove this by concluding a contradiction based on the definition of the k-truss equivalence. Obviously, there is no super-edge connecting v_1 and v_2. Otherwise, all the edges in these two super-vertices are k'-triangle connected, and these two super-vertices will be combined to one super-vertex according to the definition of the summary graph. Similarly, assume that v_1 and v_2 belong to the same k-truss communities. Then all the edges in these two super-vertices are also k-triangle connected, andv_1 and v_2 will be combined to one super-vertex according to the definition of the summary graph, which leads to a contradiction. □

Based on the above lemma, we can derive the following property for k-truss communities.

Property 4. **(Neighborhood-Disjoint Constraint)** *Given a vertex v_q and keyword set S, if there are k-truss communities $H_1^\circ, H_2^\circ, \ldots, H_m^\circ$ containing v_q, then a k-truss community $H[S']$ containing v_q for $S' \subseteq S$ must be entirely included a community H_i° $(1 \leq i \leq m)$, and v_q has at least $k-1$ neighbors containing S' in H_i°.*

This property can also be derived by concluding a contradiction as follows. Assume that a k-truss community $H[S']$ containing v_q occurs across two communities H_i° and H_j°. Obviously, the two parts contained in H_i° and H_j° must be at least k-triangle connected. Thus, H_i° and H_j° will be triangle connected, and they will belong to the same k-truss community, which contradicts the fact that H_i° and H_j° are two separate triangle connected k-truss communities.

Example 9. *Consider the query vertex v_3 and $S = \{b,c,d,e\}$ in Fig. 3. The 4-truss communities with most keywords will occur either in H_1° (super-vertices v_3) or H_2° (super-vertices v_4 and v_5) but not across them. Thus the two communities with most keywords, H_1 induced by $\{v_1,v_2,v_3,v_4,v_5\}$ with keywords $\{c,d\}$ and H_2 induced by $\{v_3,v_8,v_9,v_{10}\}$ with keywords $\{b,d\}$, are contained in H_1° and H_2°, respectively.*

Based on above properties, we can disjointly generate the frequent keyword subsets for a query v_q by dividing its neighbor set $N_G(v_q)$ into m parts if there are m k-truss communities $H_1^\circ, H_2^\circ, \ldots, H_m^\circ$ containing v_q, and generate the frequent keyword subsets for each part. We illustrate such a process by the following example.

Example 10. *Considethe query vertex v_3 and $S = \{b,c,d,e\}$ in Fig. 3. We can find 4-truss communities H_1° (super-vertices v_3) and H_2° (super-vertices v_4 and v_5). The neighbors of v_3 are only v_2, v_4, v_5 in H_1° and only v_8, v_9, v_{10} in H_2° as shown in Fig. 4. Both of them only have 3 frequent keyword subsets as shown in the two tables on the right. Thus the total number of candidate keyword subsets is largely reduced compared with the number of 9 in Example 5.*

CAC-TIndex **Algorithm**. Based on the above disjoint frequent keyword subset mining strategy, we can derive an improved algorithm CAC-TIndex, as shown in Algorithm 2. First of all, we find all the k-truss communities $H_1^\circ, H_2^\circ, \ldots, H_m^\circ$ containing v_q based on TIndex (line 1). Then we generate the candidate keyword subsets based on S and its neighbors in each community H_j°, denoted as $N_{H_j^\circ}(v_q)$ (lines 2-3). Then we start searching the keyword subsets with the largest number of keywords (line 5), and within each k-truss communities H_i°, we verify the existence of k-truss communities $G_j^\circ[S']$ containing S' (lines 7-12).

Algorithm 2: CAC-TIndex (G, v_q, S, k)

Input : A graph G, a vertex q, a keyword set S and integer k
Output: All the attributed truss communities containing v_q;

find $H_1^\circ, H_2^\circ, \ldots, H_m^\circ$ containing v_q from TIndex;
for $j = 1$ *to* m **do**
  generate $\Psi_1^j, \Psi_2^j, \ldots, \Psi_{h_j}^j$ using S and $N_{H_j^\circ}(v_q)$;
$\mathcal{R} \leftarrow \emptyset$; $i \leftarrow h_j$;
while $i \geq 1$ **do**
  for $j = 1$ *to* m **do**
    for *each* $S' \in \Psi_i^h$ **do**
      find subgraph $G_j^\circ[S']$ containing v_q from H_j°;
      $R \leftarrow$ findKTrussCom($G_j^\circ[S']$, v_q, k);
      if $R \neq \emptyset$ **then** $\mathcal{R} \leftarrow \mathcal{R} \cup R$;
  if $\mathcal{R} = \emptyset$ **then** $i \leftarrow i - 1$;
  else break;
output the communities in $\mathcal{R}$;

Algorithm 3: CAC-MTIndexI (G, v_q, S, k)

Input : A graph G, a vertex q, and an integer k
Output: All the attributed truss communities containing v_q;

generate $\mathcal{F}'_S = \{\Psi'_1, \Psi'_2, \ldots, \Psi'_{h'}\}$ and $\mathcal{T}$;
$\mathcal{R}_1 \leftarrow \cup_{w_i \in \Psi'_1} H[w_i]$; $\mathcal{R}_i \leftarrow \emptyset$ for $1 < i \leq h$; $L_{max} \leftarrow 1$;
$flag[w_i] = true, \forall w_i \in \Psi'_1$; $flag[S'] = false, \forall S' \in \Psi'_j\ (j > 1)$;
push the root and its children from right to left into stack $\mathcal{S}$;
while $\mathcal{S} \neq \emptyset$ **do**
  $node \leftarrow \mathcal{S}.pop()$; $S' \leftarrow node.S'$;
  if $|S'| \geq L_{max}$ *and* $flag[S'] \neq true$ **then**
    $S'' \leftarrow S' \setminus S'[|S'|-1]$;
    while $flag[S''] \neq true$ **do**
      $S'' \leftarrow S'' \setminus S''[|S''|-1]$;
    $G'[S'] \leftarrow H[S''] \cap (\cap_{i=|S''|+1}^{|S'|} H[S'[i]])$;
    $R \leftarrow$ findKTrussCom($G'[S']$, v_q, k);
    if $R \neq \emptyset$ **then**
      $\mathcal{R}_{|S'|} \leftarrow \mathcal{R}_{|S'|} \cup R$; $flag[S'] \leftarrow true$;
      $L_{max} \leftarrow \max\{L_{max}, |S'|\}$;
      for $cnode \in node.children$ *from right to left* **do**
        $\mathcal{S}.push(cnode)$;
Output the communities in $\mathcal{R}_{L_{max}}$;

5 IMPROVED APPROACHES

In the above CAC-TIndex algorithm, we filter out the vertices/edges first by the truss information and then by the attribute information, before we do the k-truss verification for a keyword subset S' on the filtered graph. However, such a filtered graph may still contain vertices/edges that are not included in the k-truss community $H[S']$. Consider a specific case of the anti-monotonicity property where S'' only contains a keyword w_i. A vertex u will never occur in a k-truss community $H[S']$ if it is not included in the k-truss community $H[w_i]$ ($w_i \in S'$). Thus, before finding $H[S']$, we can exclude the vertices that are not in $H[w_i]$ for each $w_i \in S'$. Therefore, in this section, we propose an improved index MTIndex that keeps multiple summary graphs for efficient truss and keyword filtering.

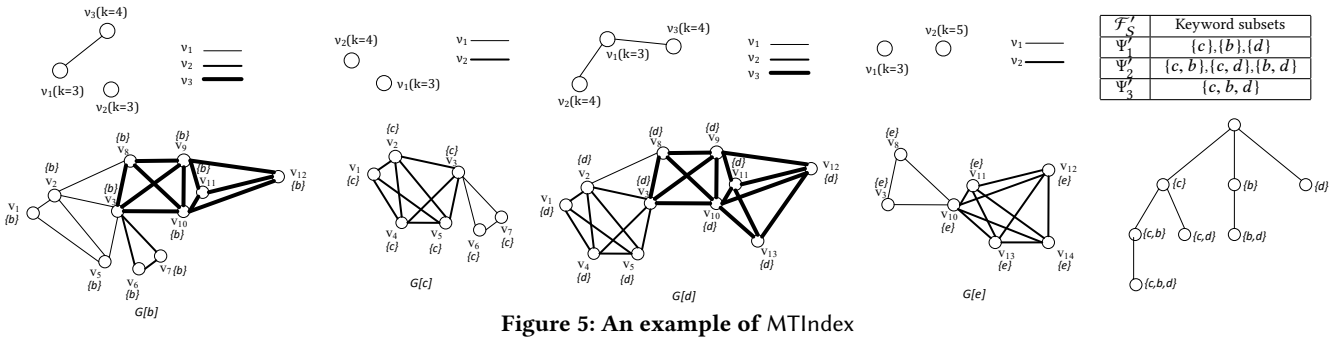
5.1 An Improved Index MTIndex

We build MTIndex to index the attribute and trussness information as follows. For each keyword w_i, we extract all the vertices containing w_i, V_i, based on which the induced subgraph is denoted as $G[w_i]$. Then we build a summary graph $\mathcal{G}_i = (\mathcal{V}_i, \mathcal{E}_i)$ for each $G[w_i]$ with $k \geq 3$. Note that, in this index, we only need to store $\mathcal{G}_i$ rather than $G[w_i]$, which is usually much smaller and more space-efficient than $G[w_i]$ for real-world networks. Such index is usually much smaller than TIndex and the original graph on most of the real-world datasets (see details in Section 7).

Filtered and ordered frequent keyword subsets. Suppose that we have mined the frequent keyword subsets $\mathcal{F}_S = \{\Psi_1, \Psi_2, \ldots, \Psi_h\}$. We can do this either by mining the frequent patterns directly from the neighborhood of v_q or by first mining the frequent patterns from the partial neighborhood in each k-truss community based on TIndex disjointly and then union them together. Clearly, the latter generates fewer keyword subsets than the former, but it needs the assistance of index TIndex. Here, we use the result generated by the former to save the index space. After generating $\mathcal{F}_S$, we can obtain the *filtered and ordered frequent keyword subsets* as follows. First, for each $w_i \in \Psi_1$, we check the existence of k-truss community $H[w_i]$ containing v_q based on the summary graph $\mathcal{G}_i$. If $H[w_i]$ does not exist, we filter out w_i in Ψ_1 and all the keyword subsets containing w_i in Ψ_j for $j > 1$. Thus, we can obtain a new Ψ'_j for each $j \geq 1$. For each keyword subset containing only one keyword in Ψ'_1, we sort them based the size of their k-truss communities, i.e., $w_i \prec w_j$ if $|E(H[w_i])| < |E(H[w_j])|$. Then for each Ψ'_j ($j > 1$), the keywords in $S' \in \Psi'_j$ are also sorted based on this order, i.e., $S'[i] \prec S'[i+1]$ for $1 \leq i < |S'|$ where $S'[i]$ be the i-th element in S'. For two subsets $S', S'' \in \Psi'_j$, we say $S' \prec S''$ if $S'[k] = S''[k]$ for $1 \leq k < i < j$, and $S'[i] \prec S''[i]$. Thus, we can obtain the *filtered and ordered frequent keyword subsets* $\mathcal{F}'_S = \{\Psi'_1, \Psi'_2, \ldots, \Psi'_{h'}\}$.

Ordered tree. Based on $\mathcal{F}'_S$, we can build an ordered tree $\mathcal{T}$ as follows. The root of this tree in the zero layer is a null node, and we store each keyword subset in Ψ'_1 in each of its child node in the first layer. Similarly, we store each keyword subset in Ψ'_2 in the each node in the second layer. Suppose the keyword set stored in node u in the j-th layer is $S' \in \Psi'_j$. Any keyword set $S'' \in \Psi'_{j+1}$ with $S'[i] = S''[i]$ for $1 \leq i \leq j$ will be stored in the child node of node u. The order of these child nodes is also determined by the order of keyword subsets in Ψ'_{j+1}.

Example 11. *For the example graph G in Fig. 3, the induced subgraph for each keyword and their summary graph is shown in Fig. 5. In $G[b]$, edges in light lines are 3-triangle connected with trussness 3, and thus are contained in super-vertex v_1. Despite having the same trussness, (v_3, v_6), (v_3, v_7), and (v_6, v_7) in medium lines are contained by another super-vertex v_2 as they are not 3-triangle connected with edges in v_1. The remaining edges in bold lines are grouped as super-vertex v_3 with trussness 4. v_1 and v_3 are connected as they are 3-triangle connected. Given a query with v_3, $S = \{b, c, d, e\}$ and $k = 4$, $\mathcal{F}'_S$ is shown in the table on the right. All the frequent keyword subsets containing e have been deleted, since there is no 4-truss community containing v_3 in the summary of $G[e]$. The keyword subsets in Ψ'_1 are sorted as c, b, d as $|E(H[c])| = 12, |E(H[b])| = 20, |E(H[d])| = 24$. The subsets Ψ'_2 and the keywords in each subset are also sorted based on this order. For example, $\{c, b\} \prec \{c, d\}$ because $b \prec d$. Based on $\mathcal{F}'_S$, we build the ordered tree in the bottom right of Fig. 5.*



$\mathcal{F}'_S$	Keyword subsets
Ψ'_1	{c},{b},{d}
Ψ'_2	{c, b},{c, d},{b, d}
Ψ'_3	{c, b, d}

Figure 5: An example of MTIndex

Algorithm 4: CAC-MTIndexD (G, v_q, S, k)

Input : A graph G, a vertex v_q, a keyword set S, and integer k
Output: All the attributed truss communities containing v_q

1 generate $\mathcal{F}'_S = \{\Psi'_1, \Psi'_2, \ldots, \Psi'_{h'}\}$;
2 $\mathcal{R} \leftarrow \emptyset$; $i \leftarrow h'$;
3 **while** $i \geq 1$ **do**
4 **for** *each* $S' \in \Psi'_i$ **do**
5 $G'[S'] \leftarrow \cap_{j=1}^{i} H[S'[j]]$;
6 $R \leftarrow$ findKTrussCom($G'[S']$, v_q, k);
7 **if** $R \neq \emptyset$ **then** $\mathcal{R} \leftarrow \mathcal{R} \cup R$;
8 **if** $\mathcal{R} = \emptyset$ **then** $i \leftarrow i - 1$;
9 **else** break;
10 Output the communities in $\mathcal{R}$;

5.2 MTIndex Based Search Algorithms

Then we discuss how to search the filtered and ordered frequent keyword set $\mathcal{F}'_S$ and the ordered tree $\mathcal{T}$ to find the triangle connected k-truss communities with the most common keywords as early as possible. In fact, it depends on the keyword distribution of the graph. If the vertices share very few keywords, we can start from the size-1 keyword subsets; otherwise, we start from the keyword subsets with the largest number of keywords. For a new query, we can simply evaluate the keyword distribution by checking the existence of triangle connected k-truss communities for $\Psi'_{\lfloor h'/2 \rfloor}$. If such communities exists, we start from Ψ'_1; otherwise, we start from $\Psi'_{h'}$. Based on this intuition, we give the incremental and decremental search algorithms as follows.

CAC-MTIndexI **Algorithm**. The incremental cohesive attributed community search is shown in Algorithm 3. First, we obtain $\mathcal{F}'_S = \{\Psi'_1, \ldots, \Psi'_{h'}\}$ and $\mathcal{T}$ (line 1). Then, we initialize $\mathcal{R}_i$ to store the k-truss communities with i shared keywords, L_{max} to store the largest number of shared keyword found currently, $flag[S']$ to indicate whether $H[S']$ has been obtained, and stack $\mathcal{S}$ to store the nodes that have been visited. Next, we start the depth first search on $\mathcal{T}$. At each step, we pop out the node in the stack $\mathcal{S}$ and get the keyword subset S' stored in this node (line 6). Here we consider to examine the existence of the k-truss community only if $|S'| \geq L_{max}$. Let $G'[S']$ denote the candidate subgraph before the k-truss verification. To compute $G'[S']$, one straight forward method is to compute the intersection of all the $H[w']$ for $w' \in S'$. Here, we give an more efficient method to obtain a smaller $G'[S']$ by utilizing the maximum subset $S'' \subseteq S'$ where $H[S'']$ exists. We can obtain the subset $S'' \subseteq S'$ by gradually deleting the last keyword in S' (lines 9-10) until $flag[S''] = true$. Then we can obtain $G'[S']$ by $H[S''] \cap (\cap_{i=|S''|+1}^{|S'|} H[S'[i]])$ in line 11, and extract the k-truss community from $G'[S']$ in line 12. If R is not empty, we add it to $\mathcal{R}'_{|S'|}$ and update $flag[S']$ (line 14). Moreover, we update L_{max} if needed, and push the children of this node into stack $\mathcal{S}$ (lines 15-17). Such process repeats until the stack is empty.

Table 1: Datasets used in our experiments

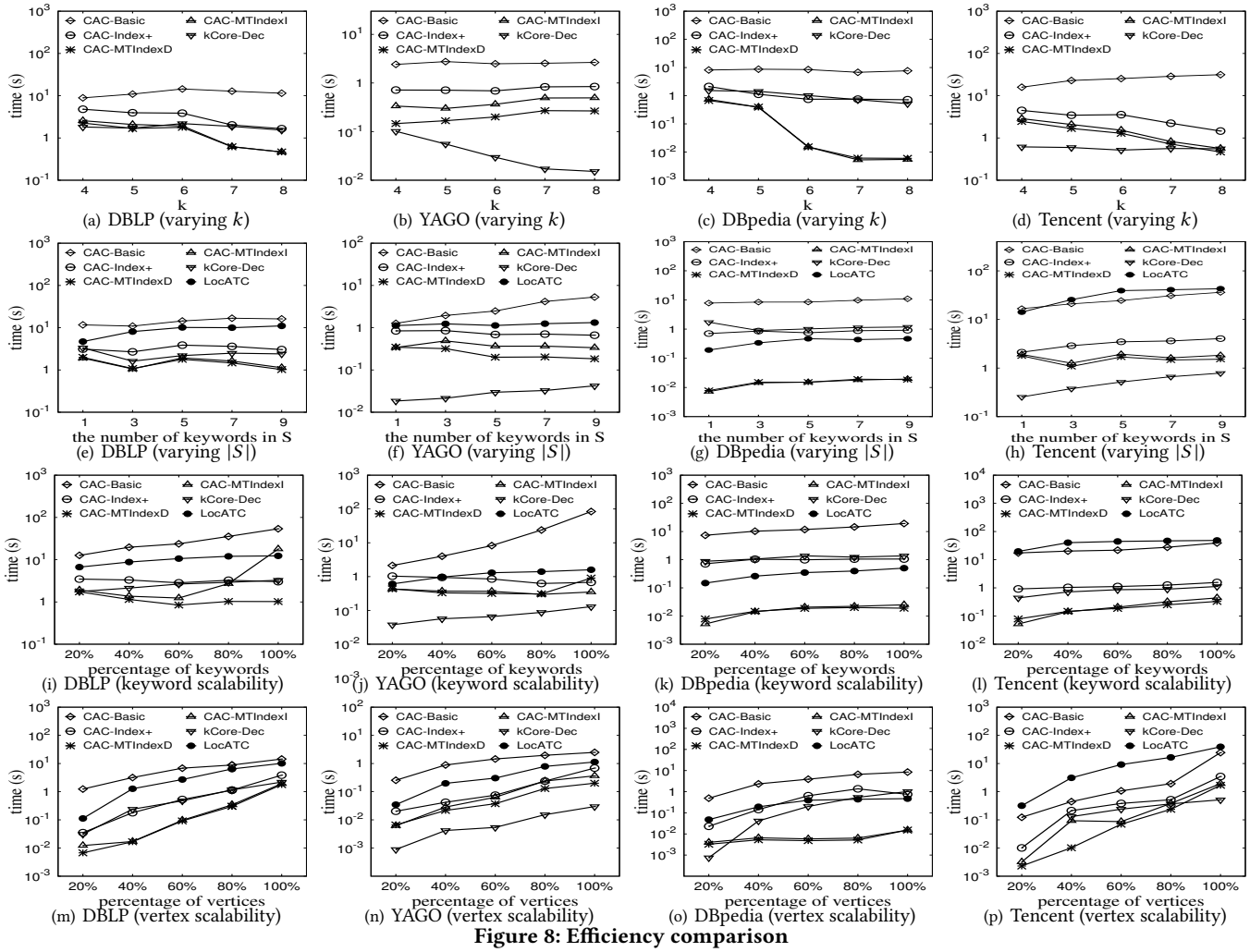
Dataset	Vertices	Edges	k_{max}	$\widehat{d}$	$\widehat{l}$	Size
DBLP	2,000,979	9,925,613	287	9.92	14.02	283.40MB
YAGO	2,637,144	5,226,311	680.4	3.96	9.58	199.64MB
DBpedia	5,897,742	17,642,447	403.2	5.98	4.26	401.41MB
Tencent	2,320,895	50,133,369	405	43.2	6.96	866.91MB

CAC-MTIndexD **Algorithm**. The decremental search is shown in Algorithm 4, which starts from the frequent keyword subset S' in $\Psi'_{h'}$ with the largest number of keywords. First, we obtain the filtered and ordered frequent keyword set $\mathcal{F}'_S = \{\Psi'_1, \ldots, \Psi'_{h'}\}$ (line 1). Then, we check the frequent keyword subsets from $\Psi'_{h'}$ to Ψ'_1 until we find the k-truss communities. Note that for each Ψ'_i, we check the keyword subsets based on their orders, and we will stop the whole process if we find the k-truss communities.

6 EXPERIMENTS

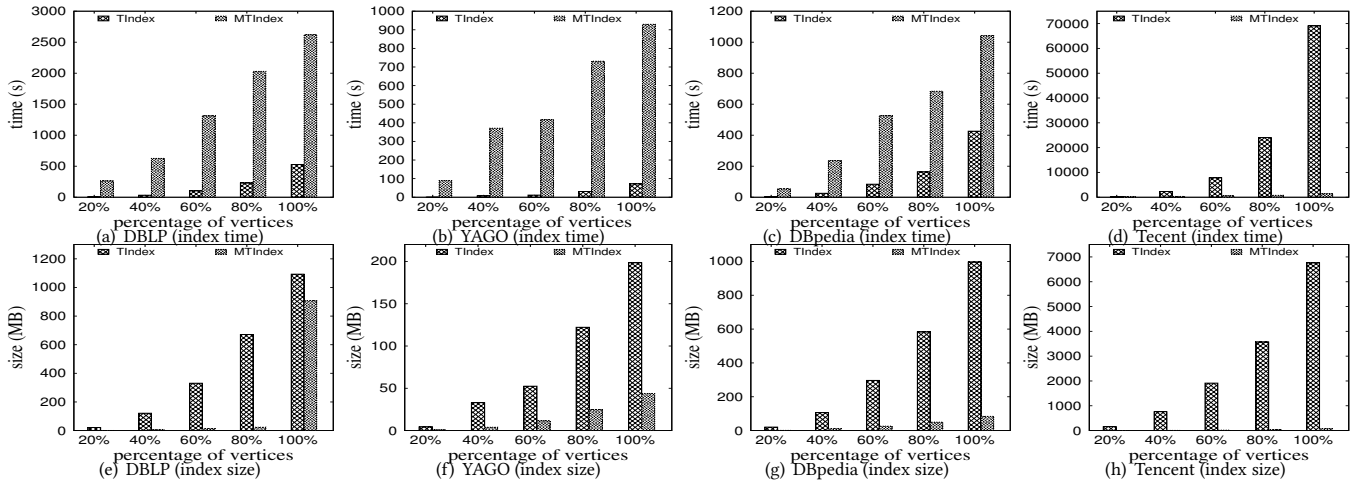
To our best knowledge, there is no existing work on cohesive attributed community search based on the triangle connected k-truss with the maximization of the shared attributes defined in this paper. We compare our methods with two closely related works [9] [14] to asses the effectiveness and efficiency of our algorithms. Here, we choose the most efficient algorithm kCore-Dec in [9] and locATC in [14] as baselines, and implemented our algorithms CAC-Basic, CAC-TIndex, CAC-MTIndexI, and CAC-MTIndexD. We conducted the experiments on four real-world datasets widely used in previous works [9] [14]: DBLP[1] with 2 million vertices and 9.9 million edges, YAGO[2] with 2.64 million and 5.23 million edges, DBpedia[3] with 5.90 million vertices and 17.6 million edges, and Tencent[4] with 2.3 million vertices and 50.1 million edges. Details of these datasets are shown in Table 1 where $\widehat{d}$ and $\widehat{l}$ are the average degree and keyword number. For each dataset, we randomly select 300 query vertices with trussness at least 6, which ensures that each query is included a k-truss community. The input keyword set S is set to the set of attributes contained by the query vertex. All the algorithms are implemented in C++, and all the experiments were conducted on a Linux server with Intel Xeon CPU 2.60GHz and 128GB memory.

[1] http://dblp.uni-trier.de/xml/
[2] https://www.mpi-inf.mpg.de/yago
[3] http://dbpedia.com/
[4] http://www.kddcup2012.org/c/kddcup2012-track1



(a) DBLP (varying k) (b) YAGO (varying k) (c) DBpedia (varying k) (d) Tencent (varying k)
(e) DBLP (varying $|S|$) (f) YAGO (varying $|S|$) (g) DBpedia (varying $|S|$) (h) Tencent (varying $|S|$)
(i) DBLP (keyword scalability) (j) YAGO (keyword scalability) (k) DBpedia (keyword scalability) (l) Tencent (keyword scalability)
(m) DBLP (vertex scalability) (n) YAGO (vertex scalability) (o) DBpedia (vertex scalability) (p) Tencent (vertex scalability)

Figure 8: Efficiency comparison

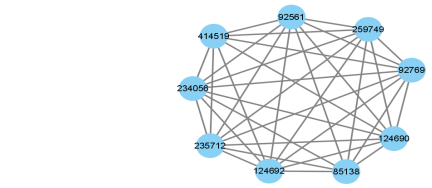


(a) DBLP (index time) (b) YAGO (index time) (c) DBpedia (index time) (d) Tecent (index time)
(e) DBLP (index size) (f) YAGO (index size) (g) DBpedia (index size) (h) Tencent (index size)

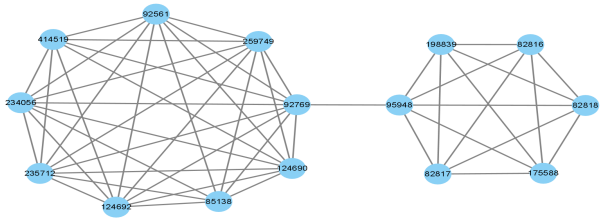
Figure 9: Time and size of index construction

6.1 Effectiveness Evaluation

We evaluate the effectiveness of our CAC model by comparing it with the core-based attributed community search kCore-Dec [9] and the truss-based community search without the consideration of attributes (denoted as NCTruss) since the locATC [14] only return the community with the largest k. Here, we only show the result of CAC-Basic as our algorithms all generate the same optimal result.

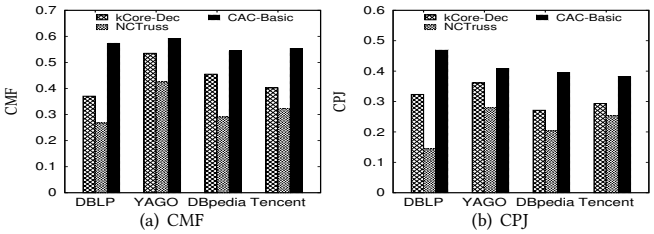


(a) 6-truss community with $L(H, S) = 3$



(b) 5-core community with $L(H, S) = 3$

Figure 6: A case study on DBLP (v_q= 4141451, S = {application, architecture, base, bring, center})



(a) CMF (b) CPJ

Figure 7: Attribute cohesiveness

Structure cohesiveness. Obviously, our CAC model has the most strong constraint on structure as it is also a k-truss and a $(k-1)$-core. Since NCTruss is also based on triangle connected k-truss community, we only compare our algorithm and kCore-Dec by a case study example in Fig. 6, where v_q = 414519 is randomly selected from the DBLP dataset with S = {application, architecture, base, bring, center}. Fig. 6(a) shows the 6-truss communities found with three shared keywords. Fig. 6(b) shows the 5-core communities found by kCore-Dec, which has the same number of shared keywords but a smaller trussness 2.

Attribute cohesiveness. We evaluate the attribute cohesiveness of the returned communities by two measures, community member frequency (CMF) and community pairwise Jaccard (CPJ) [9] [14]. CMF evaluates the keyword occurrence frequencies in the communities, and CPJ evaluates the average Jaccard similarity between the keyword sets of all the vertex pairs in the communities. Fig. 7 (a) and 7 (b) show the CMF and CPJ for the evaluated algorithms, respectively. CAC-Basic achieves better CMF and CPJ values than kCore-Dec and NCTruss. NCTruss performs the worst as expected because it mainly focuses on the structure cohesiveness but not the attribute information.

6.2 Efficiency Evaluation

Varying k. We evaluate the algorithms by varying k from 4 to 8. Fig. 8(a)-(d) shows that CAC-MTIndexI and CAC-MTIndexD perform the best on all the datasets. The running time decreases when k increases except YAGO, because the trussness of YAGO is much higher compared with other datasets. Thus, the increase of k will not drastically reduce the size of the community, and verify a community with higher trussness will cost more time. Generally, CAC-MTIndexI and CAC-MTIndexD are faster than CAC-Basic by 1-3 orders of magnitude, and also generally faster than kCore-Dec except on YAGO and Tencent. It is because the vertices in DBLP and DBPedia are more homogenous in terms of attributes, and large cores are involved kCore-Dec. locATC is not reported because it returns the communities with the largest k instead of the fixed k.

Varying $|S|$. We randomly select 1, 3, 5, 7, and 9 keywords for the query node to evaluate the scalability of the algorithms on $|S|$, Fig. 8(e)-(h) shows that CAC-MTIndexI and CAC-MTIndexD perform the best on all the datasets. CAC-MTIndexD is much faster than CAC-MTIndexI on YAGO, because the vertices in YAGO usually share many common keywords. Generally, CAC-MTIndexI and CAC-MTIndexD are faster than CAC-Basic by 1-3 orders of magnitude, and faster than locATC by 1-2 orders of magnitude. They also outperform kCore-Dec on most of the datasets except YAGO and Tencent as analyzed before. The improvement becomes more significant when the number of keywords increases in most cases.

Keyword scalability. We evaluate the scalability of the algorithms by varying the number of keywords. Specifically, we randomly select 20%, 40%, 60%, 80%, and 100% keywords of each vertex. Fig. 8(i)-(l) shows that CAC-Basic needs more time as the keyword number increases, because more frequent keyword subsets are checked. Such increase is not significant for MTIndex, CAC-TIndex, CAC-MTIndexI, and CAC-MTIndexD as they filtered out many unpromising frequent keyword subsets. Generally, CAC-MTIndexI and CAC-MTIndexD are faster than CAC-Basic by 1-3 orders of magnitude, and outperform locATC and kCore-Dec on most datasets.

Vertex scalability. We also evaluate the scalability of the algorithms by randomly selecting 20%, 40%, 60%, 80% and 100% vertices in each dataset. Fig. 8(m)-(p) shows that the running time increases as the vertex number increases. Generally, CAC-MTIndexI and CAC-MTIndexD are faster than CAC-Basic by 1-3 orders of magnitude, and outperform locATC and kCore-Dec on most datasets.

Index construction. To evaluate the scalability of the index, we randomly select 20%, 40%, 60%, 80%, and 100% vertices for each data set and show the construction time and size of TIndex and MTIndex in Fig. 9. For DBLP, YAGO, and DBpedia, TIndex can be built efficiently while MTIndex cost more time, as TIndex needs only one truss decomposition while MTIndex needs multiple truss decompositions. However, for Tencent, MTIndex cost less time than TIndex. The underlying reason is that MTIndex is constructed on a much smaller $G[w_i]$ due to the high diversity of the keywords. The index space of MTIndex is smaller than that of TIndex, and is even much smaller than the original graph size for YAGO, and DBpedia, and Tecnent due to the high diversity of the keywords. The improvement on DBLP is less significant because each attributes in DBLP are usually contained in a large number of vertices.

7 RELATED WORK

Our work is related to *community detection* (CD) with/without attributes and *community search* (CS) with/without attributes.

Community Detection. (a) *Non-attributed CD*. Community detection aims to identify all the communities in the entire network, which has been extensively studied in the literature. Most of the earlier studies can be found in surveys [10] and [28]. In recent years, various models on dense subgraphs were also studied for community detection in the decomposition manner, such as core decomposition [6] [16], truss decomposition [7] [25] [21], and k-edge/vertex connectivity component decomposition [4] [26]. Besides, some algorithms have been proposed for core/truss maintenance in dynamic graphs [30]. (b) *Attributed CD*. Attributed community detection is to find all densely connected communities with homogeneous attributes [32] [20]. [32] considers both links and keywords of vertices to compute the pairwise similarities between vertices, and then clusters the vertices to obtain the communities. [20] proposed *CODICIL*, which creates new edges based on content similarity, and then uses graph sampling to boost the efficiency of clustering. [29] defines a new model based on meta-paths to deal with heterogenous information networks. A survey of clustering attributed graphs can be found in [2]. The CD algorithms discussed above are generally inefficient for the online community search problem.

Community Search. (a) *Non-attributed CS*. Community search aims to find communities containing a given set of query vertices. Various models have been proposed to measure the cohesiveness of the community, based on notations such as random-walk [23], query biased edge density [27], clique and quasi-clique [17], k-core [8] [22], k-truss [13] [15] [1], and k-edge connected component [12], to name a few major examples. Here, we elaborate on truss based community search related to this paper, which aims to find truss communities containing a given set of query vertices. [13] constructs TCP-Index to support the efficient search of all the k-truss communities containing a given query vertex. [1] builds a more compact index EquiTruss to accelerate the computation of k-truss communities. To avoid the free-rider effect, [15] proposed an approximate method to find truss with the maximum trussness and minimum diameter. (b) *Attributed CS*. Attributed community search began to attract increasing attention from researchers recently. [9] aims to obtain k-core communities in which vertices share the most attributes. [14] finds k-truss community with the maximum attribute score, and provides heuristic algorithms to solve this NP-hard problem approximately. Recently, a variant has been proposed to deal with multiple types of attributes [19]. As stated before, these methods are inherently different from the model studied in this paper. Besides, [33] [5] [31] studied the keyword-centric community search problem where no query vertices are specified, which is orthogonal to the problem studied in this paper.

8 CONCLUSIONS

In this paper, we studied the cohesive attributed communities search problem based on a newly proposed cut-edge/vertex free *cohesive attributed community* (CAC) model to find both *structure* cohesive and *attribute* cohesive community. We explored the anti-monotonicity property and neighborhood constraint of our CAC model to reduce the number of keyword subsets, and also developed two indexes TIndex and MTIndex to filter out more keyword subsets and reduce the size of the candidate subgraph. Extensive experimental studies on four real-world datasets validated the effectiveness and efficiency of our approaches.

Acknowledgments. The work was supported in part by grants of Natural Science Foundation 61972291 and 61702435, Natural Science Foundation of Hubei Province 2018CFB519, Fundamental Research Funds for the Central Universities 2042019kf0224, ARC FT200100787, the Research Grant Council of the Hong Kong, China No. 14203618 and No. 1420291. Yuanyuan Zhu is the corresponding author.

REFERENCES

[1] Esra Akbas and Peixiang Zhao. 2017. Truss-based Community Search: a Truss-equivalence Based Indexing Approach. *PVLDB* 10, 11 (2017), 1298–1309.
[2] Cécile Bothorel, Juan David Cruz, Matteo Magnani, and Barbora Micenková. 2015. Clustering attributed graphs: models, measures and methods. *Network Science* 3, 3 (2015), 408–444.
[3] Ulrik Brandes and Thomas Erlebach (Eds.). 2005. *Network Analysis: Methodological Foundations*. LNCS, Vol. 3418. Springer.
[4] Lijun Chang, Jeffrey Xu Yu, Lu Qin, Xuemin Lin, Chengfei Liu, and Weifa Liang. 2013. Efficiently computing k-edge connected components via graph decomposition. In *SIGMOD*. 205–216.
[5] Lu Chen, Chengfei Liu, Kewen Liao, Jianxin Li, and Rui Zhou. 2019. Contextual Community Search Over Large Social Networks. In *ICDE*. 88–99.
[6] James Cheng, Yiping Ke, Shumo Chu, and M. Tamer Özsu. 2011. Efficient core decomposition in massive networks. In *ICDE*. 51–62.
[7] Jonathan Cohen. 2008. Trusses: Cohesive subgraphs for social network analysis. *National Security Agency Technical Report* 16 (2008).
[8] Wanyun Cui, Yanghua Xiao, Haixun Wang, and Wei Wang. 2014. Local search of communities in large graphs. In *SIGMOD*. 991–1002.
[9] Yixiang Fang, Reynold Cheng, Siqiang Luo, and Jiafeng Hu. 2016. Effective Community Search for Large Attributed Graphs. *PVLDB* 9, 12 (2016), 1233–1244.
[10] Santo Fortunato. 2010. Community detection in graphs. *Physics reports* 486, 3 (2010), 75–174.
[11] Jiawei Han, Jian Pei, and Yiwen Yin. 2000. Mining Frequent Patterns without Candidate Generation. In *SIGMOD*. 1–12.
[12] Jiafeng Hu, Xiaowei Wu, Reynold Cheng, Siqiang Luo, and Yixiang Fang. 2016. Querying Minimal Steiner Maximum-Connected Subgraphs in Large Graphs. In *CIKM*. 1241–1250.
[13] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-truss community in large and dynamic graphs. In *SIGMOD*. 1311–1322.
[14] Xin Huang and Laks V. S. Lakshmanan. 2017. Attribute-Driven Community Search. *PVLDB* 10, 9 (2017), 949–960.
[15] Xin Huang, Laks V. S. Lakshmanan, Jeffrey Xu Yu, and Hong Cheng. 2015. Approximate Closest Community Search in Networks. *PVLDB* 9, 4 (2015), 276–287.
[16] Wissam Khaouid, Marina Barsky, S. Venkatesh, and Alex Thomo. 2015. K-Core Decomposition of Large Networks on a Single PC. *PVLDB* 9, 1 (2015), 13–23.
[17] Pei Lee and Laks V. S. Lakshmanan. 2016. Query-Driven Maximum Quasi-Clique Search. In *ICDM*. 522–530.
[18] Zhenjun Li, Yunting Lu, WeiPeng Zhang, RongHua Li, Jun Guo, Xin Huang, and Rui Mao. 2018. Discovering Hierarchical Subgraphs of K-Core-Truss. *Data Science and Engineering* 3, 2 (2018), 136–149.
[19] Qing Liu, Yifan Zhu, Minjun Zhao, Xin Huang, Jianliang Xu, and Yunjun Gao. 2020. VAC: Vertex-Centric Attributed Community Search. In *ICDE*. 937âĂŞ948.
[20] Yiye Ruan, David Fuhry, and Srinivasan Parthasarathy. 2013. Efficient community detection in large networks using content and links. In *WWW*. 1089–1098.
[21] Ahmet Erdem Sariyüce and Ali Pinar. 2016. Fast Hierarchy Construction for Dense Subgraphs. *PVLDB* 10, 3 (2016), 97–108.
[22] Mauro Sozio and Aristides Gionis. 2010. The community-search problem and how to plan a successful cocktail party. In *SIGKDD*. 939–948.
[23] Hanghang Tong and Christos Faloutsos. 2006. Center-piece subgraphs: problem definition and fast solutions. In *SIGKDD*. 404–413.
[24] Chaokun Wang and Junchao Zhu. 2019. Forbidden Nodes Aware Community Search. In *AAAI*. AAAI Press, 758–765.
[25] Jia Wang and James Cheng. 2012. Truss Decomposition in Massive Networks. *PVLDB* 5, 9 (2012), 812–823.
[26] Dong Wen, Lu Qin, Ying Zhang, Lijun Chang, and Ling Chen. 2019. Enumerating k-Vertex Connected Components in Large Graphs. In *ICDE*. 52–63.
[27] Yubao Wu, Ruoming Jin, Jing Li, and Xiang Zhang. 2015. Robust Local Community Detection: On Free Rider Effect and Its Elimination. *PVLDB* 8, 7 (2015), 798–809.
[28] Jierui Xie, Stephen Kelley, and Boleslaw K. Szymanski. 2013. Overlapping community detection in networks: The state-of-the-art and comparative study. *ACM Comput. Surv.* 45, 4 (2013), 43:1–43:35.
[29] Yixing Yang, Yixiang Fang, Xuemin Lin, and Wenjie Zhang. 2020. Effective and efficient truss computation over large heterogeneous information networks. In *ICDE*. 901âĂŞ912.
[30] Yikai Zhang and Jeffrey Xu Yu. 2019. Unboundedness and Efficiency of Truss Maintenance in Evolving Graphs. In *SIGMOD*. 1024–1041.
[31] Zhiwei Zhang, Xin Huang, Jianliang Xu, Byron Choi, and Zechao Shang. 2019. Keyword-Centric Community Search. In *ICDE*. 422–433.
[32] Yang Zhou, Hong Cheng, and Jeffrey Xu Yu. 2009. Graph Clustering Based on Structural/Attribute Similarities. *PVLDB* 2, 1 (2009), 718–729.
[33] Yuanyuan Zhu, Qian Zhang, Lu Qin, Lijun Chang, and Jeffrey Xu Yu. 2018. Querying Cohesive Subgraphs by Keywords. In *ICDE*. IEEE Computer Society, 1324–1327.