HKU CS Tech Report TR-2016-01

Effective Community Search for Large Attributed Graphs

Yixiang Fang, Reynold Cheng, Siqiang Luo, Jiafeng Hu
Department of Computer Science, The University of Hong Kong, Pokfulam Road, Hong Kong
{yxfang, ckcheng, sqluo, jhu}@cs.hku.hk

ABSTRACT

Given a graph G and a vertex $q \in G$, the *community search* query returns a subgraph of G that contains vertices related to q. Communities, which are prevalent in *attributed graphs* such as social networks and knowledge bases, can be used in emerging applications such as product advertisement and setting up of social events. In this paper, we investigate the *attributed community query* (or AC-Q), which returns an *attributed community* (AC) for an *attributed graph*. The AC is a subgraph of G, which satisfies both *structure cohesiveness* (i.e., its vertices are tightly connected) and *keyword cohesiveness* (i.e., its vertices share common keywords). The AC enables a better understanding of how and why a community is formed (e.g., members of an AC have a common interest in music, because they all have the same keyword "music"). An AC can be "personalized"; for example, an ACQ user may specify that an AC returned should be related to some specific keywords like "research" and "sports".

To enable efficient AC search, we develop the CL-tree structure and three algorithms based on it. We evaluate our solutions on four large graphs, namely Flickr, DBLP, Tencent, and DBpedia. Our results show that ACs are more effective and efficient than existing community retrieval approaches. Moreover, an AC contains more precise and personalized information than that of existing community search and detection methods.

1. INTRODUCTION

Due to the recent developments of gigantic social networks (e.g., Flickr, Facebook, and Twitter), the topic of *attributed graphs* has attracted attention from industry and research communities [28, 3, 6, 15, 17, 32, 18]. An attributed graph is essentially a graph associated with text strings or keywords. Figure 1 illustrates an attributed graph, where each vertex represents a social network user, and its keywords describe the interest of that user.

In this paper, we investigate the *attributed community query* (or ACQ). Given an attributed graph G and a vertex $q \in G$, the ACQ returns one or more subgraphs of G known as *attributed communities* (or ACs). An AC is a kind of *community*, which consists of vertices that are closely related [27, 5, 4, 16, 23, 12]. Particularly,

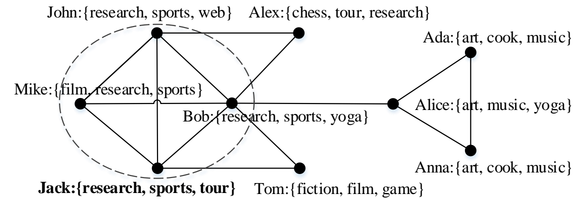


Figure 1: Attributed graph and AC (circled).

Table 1: Classification of works in community retrieval.

Graph Type	Community detection (CD)	Community search (CS)
Non-attributed	[23, 12]	[27, 5, 4, 16, 19]
Attributed	[33, 22, 21, 28, 24]	**ACQ (This paper)**

an AC satisfies *structure cohesiveness* (i.e., its vertices are closely linked to each other) and *keyword cohesiveness* (i.e., its vertices have keywords in common). Figure 1 illustrates an AC (circled), which is a connected subgraph with vertex degree 3; its vertices {`Jack`, `Bob`, `John`, `Mike`} have two keywords (i.e., "research" and "sports") in common.

Prior works. The problems related to retrieving communities from a graph can generally be classified into *community detection* (CD) and *community search* (CS). In general, CD algorithms aim to retrieve all communities for a graph [33, 22, 21, 28, 24]. These solutions are not "query-based", i.e., they are not customized for a query request (e.g., a user-specified query vertex). Moreover, they can take a long time to find all the communities for a large graph, and so they are not suitable for quick or *online* retrieval of communities. To solve these problems, CS solutions have been recently developed [27, 5, 4, 16]. These approaches are query-based, and are able to derive communities in an "online" manner. However, existing CS algorithms assume *non-attributed* graphs, and only use the graph structure information to find communities. The ACQ is a class of CS problem for attributed graphs. As we will show, the use of keyword information can significantly improve the effectiveness of the communities retrieved. Table 1 summarizes some representative existing works in this area.

Features of ACs. We now present more details about ACs.
• **Ease of interpretation.** As demonstrated in Figure 1, an AC contains tightly-connected vertices with similar contexts or backgrounds. Thus, an ACQ user can focus on the common keywords or features of these vertices (e.g., the vertices of the AC in this example contain "research" and "sports", reflecting that all members of this AC like research and sports). We call the set of common

keywords among AC vertices the *AC-label*. In our experiments, the AC-labels facilitate understanding of the vertices that form the AC.

The design of ACs allows it to be used in setting up of social events. For example, if a Twitter member has many keywords about traveling (e.g., he posted a lot of photos about his trips, with keywords), issuing an ACQ with this member as the query vertex may return other members interested in traveling, because their vertices also have keywords related to traveling. A group tour can then be recommended to these members.

• **Personalization.** The user of an ACQ can control the semantics of the AC, by specifying a set of S of keywords. Intuitively, S decides the meaning of the AC based on the user's need. If we let q=`Jack` and S={"research"}, the AC is formed by {`Jack`, `Bob`, `John`, `Mike`, `Alex`}, who are all interested in research. Let us consider another example in the DBLP bibliographical network, where each vertex's attribute is represented by the top-20 frequent keywords in their publications. Let q=`Jim Gray`. If S is the set of keywords {transaction, data, management, system, research}, we obtain the AC in Figure 2(a), which contains six prominent database researchers closely related to Jim. On the other hand, when S is {sloan, digital, sky, survey, SDSS}, the ACQ yields another AC in Figure 2(b), which indicates the seven scientists involved in the SDSS project [1]. Thus, with the use of different keyword sets S, different "personalized" communities can be obtained.

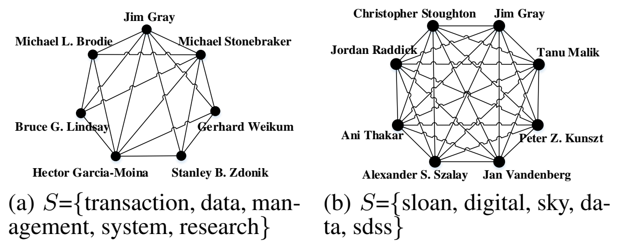
Existing CS algorithms, which do not handle attributed graphs, may not produce the two ACs above. For example, the CS algorithm in [27] returns the community with *all* the 14 vertices shown in Figures 2(a) and (b). The main reasons are: (1) these vertices are heavily linked with Jim; and (2) the keywords are not considered. In contrast, the use of set S in the ACQ places these vertices into two communities, containing vertices that are cohesive in terms of *structure* and *keyword*. This allows a user to focus on the important vertices that are related to S. For example, using the AC of Figure 2(a), a database conference organizer can invite speakers who have a close relationship with Jim.

The personalization feature is also useful in marketing. Suppose that Mary, a yoga lover, is a customer of a gym. An ACQ can be issued on a social network, with Mary as the query vertex and S={"yoga"}. Since members of the AC contain the keyword "yoga", they can be the gym's advertising targets. On the other hand, current CS algorithms may return a community that contains one or more vertices without the keyword "yoga". It is not clear whether the corresponding user of this vertex is interested in yoga.

• **Online evaluation.** Similar to other CS solutions, we have developed efficient ACQ algorithms for large graphs, allowing ACs to be generated quickly upon a query request. On the contrary, existing CD algorithms [33, 24, 22, 21] that generate all communities for a graph are often considered to be offline solutions, since they are often costly and time-consuming, especially on very large graphs.

Technical challenges and our contributions. We face two important questions: (1) What should be a sound definition of an AC? (2) How to evaluate ACQ efficiently? For the first question, we define an AC based on the *minimum degree*, which is one of the most common structure cohesiveness metrics [23, 12, 27, 5]. This measure requires that every vertex in the community has a degree of k or more. We formulate the keyword cohesiveness as maximizing the number of shared keywords in keyword set S. The shared keywords naturally reveal the common features among vertices (e.g., common interest of social network users). We can also use these shared keywords to explain how a community is formed.

[1]URL of the SDSS project: `http://www.sdss.org`.



(a) S={transaction, data, management, system, research}

(b) S={sloan, digital, sky, data, sdss}

Figure 2: Two ACs of Jim Gray.

The second question is not easy to answer, because the attributed graph G to be explored can be very large, and the (structure and keyword) cohesiveness criteria can be complex to handle. A simple way is first to consider all the possible keyword combinations, and then return the subgraphs, which satisfy the minimum degree constraint and have the most shared keywords. This solution, which requires the enumeration of all the subsets of q's keyword set, has a complexity exponential to the size l of q's keyword set. In our experiments, for some queries, l can be up to 30, resulting in the consideration of $2^{30} = 1,073,741,824$ subsets of q. The algorithm is impractical, especially when q's keyword set is large.

We observe the *anti-monotonicity* property, which states that given a set S of keywords, if it appears in every vertex of an AC, then for every subset S' of S, there exists an AC in which every vertex contains S'. We use this intuition to propose better algorithms. We further develop the *CL-tree*, an index that organizes the vertex keyword data in a hierarchical structure. The CL-tree has a space and construction time complexity linear to the size of G. We have developed three different ACQ algorithms based on the CL-tree, and they are able to achieve a superior performance.

We have performed extensive experiments on four large real graph datasets (namely Flickr, DBLP, Tencent, and DBpedia). We found that a large number of common keywords appear across vertices in our graph datasets. In DBLP, for instance, an AC with one common keyword contains over 5,000 vertices on average; an AC with two common keywords contains over 700 vertices. Hence, using shared keywords among vertices as keyword cohesiveness makes sense. We have also studied how to quantify the quality of a community, based on occurrence frequencies of keywords and similarity between the keyword sets of two vertices. We conducted a detailed case study on DBLP. These results confirm the superiority of the AC over the communities returned by existing community detection and community search algorithms, in terms of community quality. The performance of our best algorithm is 2 to 3 order-of-magnitude better than solutions that do not use the CL-tree. Another advantage of our approaches is that they organize and search vertex keywords for ACs effectively, achieving a higher efficiency than existing community search solutions (that do not use vertex keywords in the community search process).

Organization. We review the related work in Section 2, and define the ACQ problem formally in Section 3. Section 4 presents the basic solutions, and Section 5 discusses the CL-tree index. We present the query algorithms in Section 6. Our experimental results are reported in Section 7. We conclude in Section 8.

2. RELATED WORK

Community detection (CD). A large class of studies aim to discover or *detect* all the communities from an entire graph. Table 1 summarises these works. Earlier solutions, such as [23, 12], employ link-based analysis to obtain these communities. However, they do not consider the textual information associated with graphs.

Recent works focus on attributed graphs, and use clustering techniques to identify communities. For instance, Zhou et al. [33] considered both links and keywords of vertices to compute the vertices' pairwise similarities, and then clustered the graph. Ruan et al. [24] proposed a method called `CODICIL`. This solution augments the original graphs by creating new edges based on content similarity, and then uses an effective graph sampling to boost the efficiency of clustering. We will compare ACQ with this method experimentally.

Another common approach is based on topic models. In [22, 21], the `Link-PLSA-LDA` and `Topic-Link LDA` models jointly model vertices' content and links based on the `LDA` model. In [28], the attributed graph is clustered based on probabilistic inference. In [25], the topics, interaction types and the social connections are considered for discovering communities. `CESNA` [30] detects overlapping communities by assuming communities "generate" both the link and content. A discriminative approach [31] has also been considered for community detection. As discussed before, CD algorithms are generally slow, as they often consider the pairwise distance/similarity among vertices. Also, it is not clear how they can be adapted to perform online ACQ. In this paper, we propose online algorithms for finding communities on attributed graphs.

Community search (CS). Another class of solutions aims to obtain communities in an "online" manner, based on a query request. For example, given a vertex q, several existing works [27, 5, 19, 4, 16] have developed fast algorithms to obtain a community for q. To measure the structure cohesiveness of a community, the *minimum degree* is often used [27, 5, 19]. Sozio et al. [27] proposed the first algorithm `Global` to find the k-$\widehat{core}$ containing q. Cui et al. [5] proposed `Local`, which uses local expansion techniques to enhance the performance of `Global`. We will compare these two solutions in our experiments. Other definitions, including k-clique [4] and k-truss [16], have also been considered for searching communities. A recent work [19] finds communities with high influence. These works assume non-attributed graphs, and overlook the rich information of vertices that come with attributed graphs. As we will see, performing CS on attributed graphs is better than on non-attributed graphs.

Graph keyword search. Given an attributed graph G and a set Q of keywords, graph keyword search solutions output a tree structure, whose nodes are vertices of G, and the union of these vertices' keyword sets is a superset of Q [3, 6, 17]. Recent work studies the use of a subgraph of G as the query output [18]. These works are substantially different from the ACQ problem. First, they do not specify query vertices as required by the ACQ problem. Second, the tree or subgraph produced do not guarantee structure cohesiveness. Third, keyword cohesiveness is not ensured; there is no mechanism that enforces query keywords to be shared among the keyword sets of all query output's vertices. Thus, graph keyword search solutions are not designed to find ACs.

Graph pattern matching (GPM). Given a *pattern* P, the goal of GPM is to extract a set R of subgraphs of G, where for every $r \in R$, r is highly similar to P. Tong et al. [8] studied the use of lines, loops and stars; Fan et al. [9, 10] proposed bounded simulation techniques for GPM queries; in [11], GPM has been studied for finding association rules from graphs. However, there is no detailed study about how to use GPM for community search. To do this, a user has to define P, but this is not trivial: there are many possible topologies for P, and numerous ways of placing keywords on the vertices in P. Moreover, current GPM solutions focus on small patterns that generate small communities, and it is not clear whether they can support large and complex ones. Answering ACQ with GPM can also be expensive, since it involves enumerating many patterns and finding an AC with the largest number of shared keywords. For the ACQ problem, there is no need to specify P.

Table 2: Symbols and meanings.

Symbol	Meaning
$G(V, E)$	An attributed graph with vertex set V and edge set E
$W(v)$	The keyword set of vertex v
$deg_G(v)$	The degree of vertex v in G
$G[S']$	The largest connected subgraph of G s.t. $q \in G[S']$, and for any $v \in G[S']$, $S' \subseteq W(v)$
$G_k[S']$	The largest connected subgraph of G s.t. $q \in G_k[S']$, and for any $v \in G_k[S']$, $deg_{G_k[S']} v \geq k$ and $S' \subseteq W(v)$

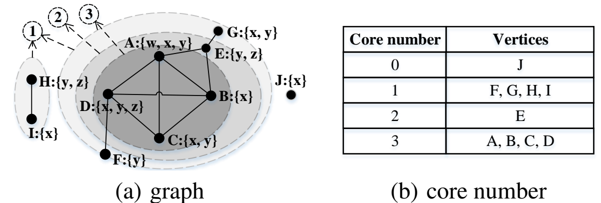
3. THE ACQ PROBLEM

We now discuss the attributed graph model, the k-core, and the AC. In the CS and CD literature, most existing works assume that the underlying graph is undirected [27, 19, 28, 24]. We also suppose that an attributed graph $G(V, E)$ is undirected, with vertex set V and edge set E. Each vertex $v \in V$ is associated with a set of keywords, $W(v)$. Let n and m be the corresponding sizes of V and E. The degree of a vertex v of G is denoted by $deg_G(v)$. Table 2 lists the symbols used in the paper.

A community is often a subgraph of G that satisfies *structure cohesiveness* (i.e., the vertices contained in the community are linked to each other in some way). A common notion of structure cohesiveness is that the *minimum degree* of all the vertices that appear in the community has to be k or more [27, 26, 2, 7, 5, 19]. This is used in the k-core and the AC. Let us discuss the k-core first.

DEFINITION 1 (k-CORE [26, 2]). *Given an integer k ($k \geq 0$), the k-core of G, denoted by H_k, is the largest subgraph of G, such that $\forall v \in H_k$, $deg_{H_k}(v) \geq k$.*

We say that H_k has an order of k. Notice that H_k may not be a connected graph [2], and its connected components, denoted by k-$\widehat{core}$s, are usually the "communities" returned by k-$\widehat{core}$ search algorithms.



Core number	Vertices
0	J
1	F, G, H, I
2	E
3	A, B, C, D

(a) graph (b) core number

Figure 3: Illustrating the k-core and the AC.

EXAMPLE 1. *In Figure 3(a), $\{A, B, C, D\}$ is both a 3-core and a 3-$\widehat{core}$. The 1-core has vertices $\{A, B, C, D, E, F, G, H, I\}$, and is composed of two 1-$\widehat{core}$ components: $\{A, B, C, D, E, F, G\}$ and $\{H, I\}$. The number k in each circle represents the k-$\widehat{core}$ contained in that ellipse.*

Observe that k-*core*s are "nested" [2]: given two positive integers i and j, if $i < j$, then $H_j \subseteq H_i$. In Figure 3(a), H_3 is contained in H_2, which is nested within H_1.

DEFINITION 2 (CORE NUMBER). *Given a vertex $v \in V$, its core number, denoted by $core_G[v]$, is the highest order of a k-core that contains v.*

A list of core numbers and their respective vertices for Example 1 are shown in Figure 3(b). In [2], an $O(m)$ algorithm was proposed to compute the core number of every vertex.

We now formally define the ACQ problem as follows.

PROBLEM 1 (ACQ). *Given a graph $G(V, E)$, a positive integer k, a vertex $q \in V$ and a set of keywords $S \subseteq W(q)$, return a set $\mathcal{G}$ of graphs, such that $\forall G_q \in \mathcal{G}$, the following properties hold:*

- ***Connectivity**. $G_q \subseteq G$ is connected and contains q;*
- ***Structure cohesiveness**. $\forall v \in G_q$, $deg_{G_q}(v) \geq k$;*
- ***Keyword cohesiveness**. The size of $L(G_q, S)$ is maximal, where $L(G_q, S) = \cap_{v \in G_q}(W(v) \cap S)$ is the set of keywords shared in S by all vertices of G_q.*

We call G_q the *attributed community* (or AC) of q, and $L(G_q, S)$ the *AC-label* of G_q. In Problem 1, the first two properties are also specified by the k-$\widehat{core}$ of a given vertex q [27]. The *keyword cohesiveness* (Property 3), which is unique to Problem 1, enables the retrieval of communities whose vertices have common keywords in S. We use S to impose semantics on the AC produced by Problem 1. By default, $S = W(q)$, which means that the AC generated should have keywords common to those associated with q. If $S \subset W(q)$, it means that the ACQ user is interested in forming communities that are related to some (but not all) of the keywords of q. A user interface could be developed to display $W(q)$ to the user, allowing her to include the desired keywords into S. For example, in Figure 3(a), if q=A, k=2 and S=$\{w, x, y\}$, the output of Problem 1 is $\{A, C, D\}$, with AC-label $\{x, y\}$, meaning that these vertices share the keywords x and y.

We require $L(G_q, S)$ to be maximal in Property 3, because we wish the AC(s) returned only contain(s) the most related vertices, in terms of the number of common keywords. Let us use Figure 3(a) to explain why this is important. Using the same query (q=A,k=2,S= $\{w, x, y\}$), without the "maximal" requirement, we can obtain communities such as $\{A, B, E\}$ (which do not share any keywords), $\{A, B, D\}$, or $\{A, B, C\}$ (which share 1 keyword). Note that there does not exist an AC with AC-label being exactly $\{w, x, y\}$. Our experiments (Section 7) show that imposing the "maximal" constraint yields the best result. Thus, we adopt Property 3 in Problem 1. If there is no AC whose vertices share one or more keywords (*i.e.*, $|L(G_q, S)|$=0), we return the subgraph of G that satisfies Properties 1 and 2 only. [2]

There are other candidates for structure cohesiveness (e.g., k-truss, k-clique) and *keyword cohesiveness* (e.g., Jaccard similarity and string edit distance). In our report [29], we studied a variant of keyword cohesiveness, in which an ACQ user may specify that an AC returned must contain vertices having specific keywords. We also examined its approximate version, where given a threshold $\theta \in [0, 1]$, each vertex in an AC has at least $|S| \times \theta$ keywords in S. These variants can be easily supported by our solutions to ACQ. An interesting future work is to compare the use of different structure and keyword cohesiveness definitions in an ACQ.

4. BASIC SOLUTIONS

For ease of presentation, we say that v contains a set S' of keywords, if $S' \subseteq W(v)$. We use $G[S']$ to denote the largest connected subgraph of G, where each vertex contains S' and $q \in G[S]$. We use $G_k[S']$ to denote the largest connected subgraph of $G[S']$, in which every vertex has degree being at least k in $G_k[S']$. We call S' a qualified keyword set for the query vertex q on the graph G, if $G_k[S']$ exists.

Given a query vertex q, a straightforward method to answer ACQ performs three steps. First, all non-empty subsets of S, $S_1, S_2, \cdots, S_{2^l-1}$ (l=$|S|$), are enumerated. Then, for each subset S_i($1 \leq i \leq 2^l$-1), we verify the existence of $G_k[S_i]$ and compute it when it exists (We postpone to discuss the details). Finally, we output the subgraphs having the most shared keywords among all $G_k[S_i]$.

One major drawback of the straightforward method is that we need to compute $2^l - 1$ subgraphs (*i.e.*, $G_k[S_i]$). For large values of l, the computation overhead renders the method impractical, and we do not further consider this method in the paper. To alleviate this issue, we propose the following two-step framework.

4.1 Two-Step Framework

The two-step framework is mainly based on the following *anti-monotonicity* property.

LEMMA 1 (ANTI-MONOTONICITY). *Given a graph G, a vertex $q \in G$ and a set S of keywords, if there exists a subgraph $G_k[S]$, then there exists a subgraph $G_k[S']$ for any subset $S' \subseteq S$.*

All the proofs of lemmas studied in this paper can be found in the full version [29]. The anti-monotonicity property allows us to stop examining all the super sets of S'($S' \subseteq S$), once have verified that $G_k[S']$ does not exist. The basic solution begins with examining the set, Ψ_1, of size-1 candidate keyword sets, *i.e.*, each candidate contains a single keyword of S. It then repeatedly executes the following two key steps, to retrieve the size-2 (size-3, ...) qualified keyword subsets until no qualified keyword sets are found.

- **Verification.** For each candidate S' in Ψ_c (initially c=1), mark S' as a qualified set if $G_k[S']$ exists.
- **Candidate generation.** For any two current size-c qualified keyword sets which only differ in one keyword, union them as a new expanded candidate with size-(c+1), and put it into set Ψ_{c+1}, if all its subsets are qualified, by Lemma 1.

Among the above steps, the key issue is how to compute $G_k[S']$. Since $G_k[S']$ should satisfy the *structure cohesiveness* (*i.e.*, minimum degree at least k) and *keyword cohesiveness* (*i.e.*, every vertex contains keyword set S'). Intuitively, we have two approaches to compute $G_k[S']$: either searching the subgraph satisfying degree constraint first, followed by further refining with keyword constraints (called `basic-g`); or vise versa (called `basic-w`). These two algorithms form our baseline solutions. Their pseudocodes are presented in the appendix of the full version [29].

5. CL-TREE INDEX

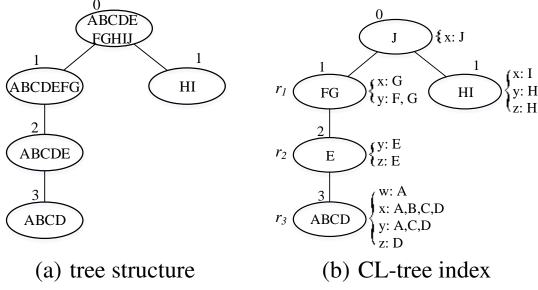
The major limitation of `basic-g` and `basic-w` is that they need to find the k-$\widehat{core}$s and do keyword filtering repeatedly. This makes the community search very inefficient. To achieve higher query efficiency, we propose a novel index, called **CL-tree** (Core Label tree), which organizes both the k-$\widehat{core}$s and keywords into a tree structure. Based on the index, the efficiency of answering ACQ and its variants can be improved significantly. We first introduce the index in Section 5.1, and then propose two index construction methods in Section 5.2.

5.1 Index Overview

The CL-tree index is built based on the key observation that cores are nested. Specifically, a (k+1)-$\widehat{core}$ must be contained in a k-$\widehat{core}$. The rationale behind is, a subgraph has a minimum degree at least $k+1$ implies that it has a minimum degree at least k. Thus, all k-$\widehat{core}$s can be organized into a tree structure[3]. We illustrate this in Example 2.

[2]In practice, the query user can be alerted by the system when there is no sharing among the vertices.

[3]We use "node" to mean "CL-tree node" in this paper.



(a) tree structure (b) CL-tree index

Figure 4: An example CL-tree index.

EXAMPLE 2. *Consider the graph in Figure 3(a). All the k-$\widehat{core}$s can be organized into a tree as shown in Figure 4(a). The height of the tree is 4. For each tree node, we attach the core number and vertex set of its corresponding k-$\widehat{core}$.*

From the tree structure in Figure 4(a), we conclude that, if a (k+1)-$\widehat{core}$ (denoted as $\mathcal{C}_{k+1}$) is contained in a k-$\widehat{core}$ (denoted as $\mathcal{C}_k$), then there is a tree node corresponding to $\mathcal{C}_{k+1}$ and its parent node corresponds to $\mathcal{C}_k$. Besides, the height of the tree is at most $k_{max} + 1$, where k_{max} is the maximum core number.

The tree structure in Figure 4(a) can be stored compactly, as shown in Figure 4(b). The key observation is that, for any internal node p in the tree, the vertex sets of its child nodes are the subsets of p's vertex set, because of the inclusion relationship. To save space cost, we can remove the redundant vertices that are shared by p's child nodes from p's vertex set. After such removal, we obtain a compressed tree, where each graph vertex appears only once. This structure constitutes the CL-tree index, the nodes of which are further augmented by inverted lists (Figure 4(b)). For each keyword e that appears in a CL-tree node, a list of IDs of vertices whose keyword sets contain e is stored. For example, in node r_3, the inverted list of keyword y contains $\{A, C, D\}$. As discussed later, given a keyword set T, these inverted lists allow efficient retrieval of vertices whose keyword sets contain T. To summarize, each CL-tree node contains four elements:

- *coreNum*: the core number of the k-$\widehat{core}$;
- *vertexSet*: a set of graph vertices;
- *invertedList*: a list of $< key, value >$ pairs, where the key is a keyword contained by vertices in $vertexSet$ and the $value$ is the list of vertices in $vertexSet$ containing key;
- *childList*: a list of child nodes.

Figure 4(b) depicts the CL-tree index for the example graph in Figure 3(a), the elements of each tree node are labeled explicitly. Using the CL-tree, the following two key operations used by our query algorithms (Section 6), can be performed efficiently.

- **Core-locating.** Given a vertex q and a core number c, find the k-$\widehat{core}$ with core number c containing q, by traversing the CL-tree.
- **Keyword-checking.** Given a k-$\widehat{core}$, find vertices which contain a given keyword set, by intersecting the inverted lists of keywords contained in the keyword set.

Remarks. The CL-tree can also support k-$\widehat{core}$ queries on general graphs without keywords. For example, it can be applied to finding k-$\widehat{core}$ in previous community search methods [27].

Space cost. Since each graph vertex appears only once and each keyword only needs constant space cost, the space cost of keeping such an index is $O(\widehat{l} \cdot n)$, where $\widehat{l}$ denotes the average size of $W(v)$ over V. Thus, the space cost is linear to the size of G.

5.2 Index Construction

To build the CL-tree index, we propose two methods, `basic` and `advanced`, as presented in Section 5.2.1 and 5.2.2.

5.2.1 The Basic Method

As k-$\widehat{core}$s of a graph are nested naturally, it is straightforward to build the CL-tree recursively in a top-down manner. Specifically, we first generate the root node for 0-core, which is exactly the entire graph. Then, for each k-$\widehat{core}$ of 1-core, we generate a child node for the root node. After that, we only remain vertices with core numbers being 0 in the root node. Then for each child node, we can generate its child nodes in the similar way. This procedure is executed recursively until all the nodes are well built.

Algorithm 1 Index construction: `basic`

1: **function** BUILDINDEX($G(V, E)$)
2: $core_G[\,] \leftarrow k$-core decomposition on G;
3: $k \leftarrow$0, $root \leftarrow (k, V)$;
4: BUILDNODE($root$, 0);
5: build an inverted list for each tree node;
6: **return** $root$;
7: **function** BUILDNODE($root$, k)
8: $k \leftarrow k + 1$;
9: **if** $k \leq k_{max}$ **then**
10: obtain U_k from $root$;
11: compute the connected components for the induced graph on U_k;
12: **for** each connected component C_i **do**
13: build a tree node $p_i \leftarrow (k, C_i.vertexSet)$;
14: add p_i into $root.childList$;
15: remove C_i's vertex set from $root.vertexSet$;
16: BUILDNODE(p_i, k);

Algorithm 1 illustrates the pseudocodes. We first do k-core decomposition using the linear algorithm [2], and obtain an array $core_G[\,]$(line 2), where $core_G[i]$ denotes the core number of vertex i in G. We denote the maximal core number by k_{max}. Then, we initialize the root node by the core number k=0 and V (line 3). Next, we call the function BUILDNODE to build its child nodes (line 4). Finally, we build an inverted list for each tree node and obtain a well built CL-tree (lines 5-6).

In BUILDNODE, we first update k and obtain the vertex set U_k from $root.vertexSet$, which is a set of vertices with core numbers being at least k. Then we find all the connected components from the subgraph induced by U_k (lines 8-11). Since each connected component C_i corresponds to a k-$\widehat{core}$, we build a tree node p_i with core number k and the vertex set of C_i, and then link it as a child of $root$ (lines 12-14). We also update $root$'s vertex set by removing vertices (line 15), which are shared by C_i. Finally, we call the BUILDNODE function to build p_i's child nodes recursively until all the tree nodes are created (line 16).

Complexity analysis. The k-core decomposition can be done in $O(m)$. The inverted lists of each node can be built in $O(\widehat{l} \cdot n)$. In function BUILDNODE, we need to compute the connected components with a given vertex set, which costs $O(m)$ in the worst case. Since the recursive depth is k_{max}, the total time cost is $O(m \cdot k_{\max} + \widehat{l} \cdot n)$. Similarly, the space complexity is $O(m + \widehat{l} \cdot n)$.

5.2.2 The Advanced Method

While the `basic` method is easy to implement, it meets efficiency issues when both the given graph size and its k_{max} value are large. For instance, when given a clique graph with n vertices (*i.e.*, edges exist between every pair of nodes), the value of k_{max} is n–1. Therefore, the time complexity of the `basic` method could be $O((m + \widehat{l}) \cdot n)$, which may lead to low efficiency for large-scale

graphs. To enable more efficient index construction, we propose the `advanced` method, whose time and space complexities are almost linear with the size of the input graph.

The `advanced` method builds the CL-tree level by level in a bottom-up manner. Specifically, the tree nodes corresponding to larger core numbers are created prior to those with smaller core numbers. For ease of presentation, we divide the discussion into two main steps: creating tree nodes and creating tree edges.

1. Creating tree nodes. We observe that, if we acquire the vertices with core numbers at least c and denote the induced subgraph on the vertices as T_c, then the connected components of T_c have one-to-one correspondence to the c-$\widehat{core}$s. A simple algorithm would be, searching connected components for $T_c(0 \leq c \leq k_{max})$ independently, followed by creating one node for each distinct component. This algorithm apparently costs $O(k_{max} \cdot m)$ time, as computing connected components takes linear time.

However, we can do better if we can incrementally update the connected components in a level by level manner (*i.e.*, maintain the connected components of T_{c+1} from those of T_c). We note that, such a node creation process is feasible by exploiting the classical *union-find forest* [1]. Generally speaking, the union-find forest enables efficient maintenance of connected components of a graph when edges are incrementally added. Using union-find forest to maintain connected components follows a process of edge examination. Initially, each vertex is regarded as a connected component. Then, edges are examined one by one. During the examine process, two components are merged together when encounters an edge connecting them. To achieve an efficient merge of components, the vertices in the component form a tree. The tree root acts as the representative vertex of the component. As such, merging two components is essentially linking two root vertices together. To guarantee the CL-tree nodes are formed in a bottom-up manner, we assign an examine priority to each edge. The priority is defined by the larger value of the two core numbers corresponding to the two end vertices of an edge. The edges associated to vertices with larger core numbers are examined first.

2. Creating tree edges. Tree edges are also inserted during the graph edge examination process. In particular, when we examine a vertex v with a set, B, of its neighbors, whose core numbers are larger than $core_G[v]$, we require that the tree node containing v should link to the tree node containing the vertex, whose core number is the smallest among all the vertices in B. Nevertheless, the classical union-find forest is not able to maintain such information. To address this issue, we thus propose an auxiliary data structure, called **Anchored Union-Find** (details of AUF are in [29]), based on the classical union-find forest. We first define *anchor vertex*.

DEFINITION 3 (ANCHOR VERTEX). *Given a connected subgraph $G' \subseteq G$, the anchor vertex is the vertex with core number being $min\{core_G[v] | v \in G'\}$.*

The AUF is an extension of union-find forest, in which each tree has an anchor vertex, and it is attached to the root node. In CL-tree, for any node p with corresponding k-$\widehat{core}$ $\mathcal{C}_k$, its child nodes correspond to the k-$\widehat{core}$s, which are contained by $\mathcal{C}_k$ and have core numbers being the most close to the core number of node p. This implies that, when building the CL-tree in a bottom-up manner, we can maintain the anchor vertices for the k-$\widehat{core}$s dynamically, and they can be used to link nodes with their child nodes. In addition, we maintain a vertex-node map, where the key is a vertex and the value is the tree node contains this vertex, for locating tree nodes. The pseudocodes and analysis are reported in the full version [29].

Complexity analysis. With our proposed AUF, we can reduce the complexity of CL-tree construction to $O(m \cdot \alpha(n))$, where $\alpha(n)$, the inverse Ackermann function, is less than 5 for all remotely practical values of n [1].

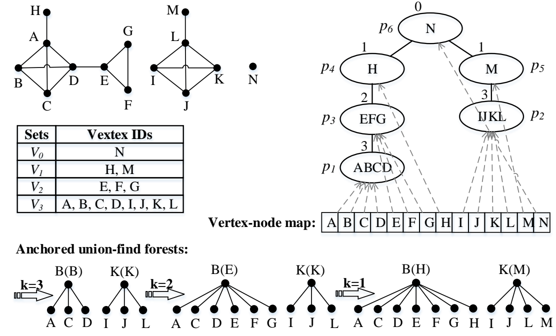


Figure 5: An index built by `advanced` method.

EXAMPLE 3. *Figure 5 depicts an example graph with 14 vertices $A, \cdots, N$. V_i denotes the set of vertices whose core numbers are i. When k=3, we first generate two leaf nodes p_1 and p_2, then update the AUF, where roots' anchor vertices are in the round brackets. When k=2, we first generate node p_3, then link it to p_1, and then update the AUF forest. When k=1, we first generate nodes p_4 and p_5. Specifically, to find the child nodes of p_4, we first find its neighbor A, then find A's parent B using current AUF forest. Since the anchor vertex of B is E and E points to p_3 in the inverted array, we add p_3 into p_4's child List. When k=0, we generate p_6 and finish the index construction.*

Index maintenance. We now briefly discuss an *incremental* version of the CL-tree construction algorithm, which can handle the changes of keywords and graph edges, without rebuilding the CL-tree from scratch. To insert (or remove) a keyword of a vertex, we just need to update the $invertedList$ of the CL-tree node containing the vertex. To insert (or remove) a graph edge, we borrow the results from [20], which discusses how to maintain a k-core. A more detailed discussion can be found in our report [29]. We plan to investigate this issue more extensively in the future.

6. QUERY ALGORITHMS

In this section, we present three query algorithms based on the CL-tree index. Based on how we verify the candidate keyword sets, we divide our algorithms into incremental algorithms (from examining smaller candidate sets to larger ones) and decremental algorithm (from examining larger candidate sets to smaller ones). We propose two incremental algorithms called `Inc-S` (Incremental Space efficient) and `Inc-T` (Incremental Time efficient), to trade off between the memory consumption and the computational overhead. We also design a decremental algorithm called `Dec` (Decremental). Our interesting finding is that, while `Dec` seems not intuitive, it ranks as the most efficient one. `Inc-S` and `Inc-T` are presented in Section 6.1. `Dec` is introduced in Section 6.2.

6.1 The Incremental Algorithms

While the high-level idea of incremental algorithms resembles the basic solutions (see Section 4), `Inc-S` and `Inc-T` advance them with the exploitation of the CL-tree. Specifically, they can always verify the existence of $G_k[S']$ within a subgraph of G instead of the entire graph G. More interestingly, the subgraph for such verifications shrinks when the candidate set S' expands. Therefore, a large sum of redundant computation is cut off during the verification. We present `Inc-S` and `Inc-T` in Sections 6.1.1 and 6.1.2.

6.1.1 Inc-S Algorithm

We first introduce a new concept, called **subgraph core number**, which is geared to the main idea of `Inc-S`.

DEFINITION 4 (SUBGRAPH CORE NUMBER). *Given a subgraph G' of G, its core number, $core_G[G']$, is defined as $min\{core_G[v] | v \in G'\}$.*

`Inc-S` follows the two-step framework (*verification* and *candidate generation*) introduced in Section 4. With the CL-tree, we improve the verification step as follows.

- **Core-based verification.** For each newly generated size-(c+1) candidate keyword set S' expanded from size-c sets S_1 and S_2, mark S' as a qualified set if $G_k[S']$ exists *in a subgraph of core number* $max\{core_G[G_k[S_1]], core_G[G_k[S_2]]\}$.

The core-based verification guarantees that, with the expansion of the candidate keyword sets, the verification becomes faster as it only needs to examine the existence of $G_k[S']$ in a smaller k-$\widehat{core}$ (Recall that cores with large core numbers are nested in the cores with small core numbers). The correctness of such shrunk verification range is guaranteed by the following lemma.

LEMMA 2. *Given two subgraphs $G_k[S_1]$ and $G_k[S_2]$ of a graph G, for a new keyword set S' generated from S_1 and S_2 (i.e., $S' = S_1 \cup S_2$), if $G_k[S']$ exists, then it must appear in a k-$\widehat{core}$ with core number at least*

$$max\{core_G[G_k[S_1]], core_G[G_k[S_2]]\}. \quad (1)$$

The verification process can be further accelerated by checking the numbers of vertices and edges, as indicated by Lemma 3.

LEMMA 3. *Given a connected graph $G(V, E)$ with n=$|V|$ and m=$|E|$, if $m - n < \frac{k^2-k}{2} - 1$, there is no k-$\widehat{core}$ in G.*

This lemma implies that, for a connected subgraph G', whose edge and vertex numbers are m and n, if $m - n < \frac{k^2-k}{2} - 1$, then we cannot find $G_k[S']$ from G'.

We present `Inc-S` in Algorithm 2. The input is a CL-tree rooted at $root$, a query vertex q, a positive integer k and a keyword set S. We apply `core-locating` on the CL-tree to locate the internal nodes whose corresponding k-$\widehat{core}$s contain q (line 2). Note that their core numbers are in the range of $[k, core_G[q]]$, as required by the structure cohesiveness. Then, we set l=0, indicating the sizes of current keyword sets, and initialize a set Ψ of $< S', c >$ pairs, where S' is a set containing a keyword from S and c is the initial core number k (line 3). Note that we skip those keywords, which are in S, but not in $W(q)$. In the while loop (lines 4-18), for each $< S', c >$ pair, we first perform `keyword-checking` to find $G[S']$ using the keyword inverted lists of the subtree rooted at node r_c. If we cannot ensure that $G[S']$ does not contain a k-$\widehat{core}$ by Lemma 3, we then find $G_k[S']$ from $G[S']$ (lines 8-9). If $G_k[S']$ exists, we put S' with its core number into the set Φ_l (lines 10-11). Next, if Φ_l is nonempty, we generate new candidates by calling GENECAND(Φ_l), which is detailed in the full version [29]. For each candidate S' in Ψ, we compute the core number using Lemma 2 and update it as a pair in Ψ (lines 12-17); otherwise, we stop (line 18). Finally, we output the communities of the latest verified keyword sets (line 19).

EXAMPLE 4. *Consider the graph in Figure 3(a) and its index in Figure 4(b). Let q=A, k=1 and S=$\{w, x, y\}$. By Algorithm 2, we first find 3 root nodes r_1, r_2 and r_3. In the first while loop, we find 2 qualified keyword sets $\{x\}and\{y\}$ with core numbers being 3 and 1. By Lemma 2, we only need to verify the new candidate keyword set $\{x, y\}$ under node r_3.*

Algorithm 2 Query algorithm: `Inc-S`

```
1: function QUERY(G, root, q, k, S)
2:     find subtree root nodes r_k, r_{k+1}, ··· , r_{core_G[q]};
3:     initialize l=0, Ψ using S;
4:     while true do
5:         l ← l + 1; Φ_l ← ∅;
6:         for each < S', c > ∈ Ψ do
7:             find G[S'] under the root r_c;
8:             if G[S'] is not pruned by Lemma 3 then
9:                 find G_k[S'] from G[S'];
10:                if G_k[S'] exists then
11:                    Φ_l.add(< S', core_G[G_k[S']] >);
12:        if Φ_l ≠ ∅ then
13:            Ψ ← GENECAND(Φ_l);
14:            for each S' in Ψ do
15:                if S' is generated from S_1 and S_2 then
16:                    c ← max{core_G[G_k[S_1]], core_G[G_k[S_2]]};
17:                    Ψ.update(S', < S', c >);
18:        else break;
19:    output the communities of keyword sets in Φ_{l−1};
```

6.1.2 Inc-T Algorithm

We begin with a lemma which inspires the design of `Inc-T`.

LEMMA 4. *Given two keyword sets S_1 and S_2, if $G_k[S_1]$ and $G_k[S_2]$ exist, we have*

$$G_k[S_1 \cup S_2] \subseteq G_k[S_1] \cap G_k[S_2]. \quad (2)$$

This lemma implies, if S' is generated from S_1 and S_2, we can find $G_k[S']$ from $G_k[S_1] \cap G_k[S_2]$ directly. Since every vertex in $G_k[S_1] \cap G_k[S_2]$ contains both S_1 and S_2, we do not need to consider the keyword constraint again when finding $G_k[S']$.

Based on Lemma 4, we introduce a new algorithm `Inc-T`. Different from `Inc-S`, `Inc-T` maintains $G_k[S']$ rather than $core_G[G_k[S']]$ for each qualified keyword set S'. As we will demonstrate later, `Inc-T` is more effective for shrinking the subgraphs containing the ACs, and thus more efficient. As a trade-off for better efficiency, `Inc-T` consumes more memory as it needs to store a list of subgraph $G_k[S']$ in memory.

Algorithm 3 presents `Inc-T`. We first apply `core-locating` to find the k-$\widehat{core}$ containing q from the CL-tree (line 2). Then, we set l=0, indicating the sizes of current keyword sets, and initialize a set Ψ of $< S', \widehat{G} >$ pairs, where S' is a set containing a keyword from S and $\widehat{G}$ is the k-$\widehat{core}$. The while loop (lines 4-18) is similar with that of `Inc-S`. The main differences are that: (1) for each qualified keyword set S', `Inc-T` keeps $G_k[S']$ in memory (line 11); and (2) for each candidate keyword set S' generated from S_1 and S_2, `Inc-T` finds $G_k[S']$ from $G_k[S_1] \cap G_k[S_2]$ directly without further keyword verification (lines 6-9, 16).

EXAMPLE 5. *Continue the graph and query (q=A, k=1, S=$\{w, x, y\}$) in Example 4. By `Inc-T`, we first find $G_1[\{x\}]$ and $G_1[\{y\}]$, whose vertex sets are $\{A, B, C, D\}$ and $\{A, C, D, E, F, G\}$. Then to find $G_1[\{x, y\}]$, we only need to search it from the subgraph, induced by the vertex set $\{A, C, D\}$.*

6.2 The Decremental Algorithm

The decremental algorithm, denoted by `Dec`, differs from previous query algorithms not only on the generation of candidate keyword sets, but also on the verification of candidate keyword sets.

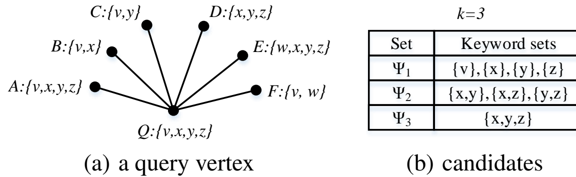
1. Generation of candidate keyword sets. `Dec` exploits the key observation that, if S' ($S' \subseteq S$) is a qualified keyword set, then there are at least k of q's neighbors containing set S'. This

Algorithm 3 Query algorithm: `Inc-T`

1: **function** QUERY(G, $root$, q, k, S)
2: find the k-$\widehat{core}$, which contains q;
3: initialize l=0, Ψ using S;
4: **while** $true$ **do**
5: $l \leftarrow l + 1$; $\Phi_l \leftarrow \emptyset$;
6: **for** each $< S', \widehat{G} > \in \Psi$ **do**
7: find $G[S']$ from $\widehat{G}$;
8: **if** $G[S']$ is not pruned by Lemma 3 **then**
9: find $G_k[S']$ from $G[S']$;
10: **if** $G_k[S']$ exists **then**
11: Φ_l.add($< S', G_k[S'] >$);
12: **if** $\Phi_l \neq \emptyset$ **then**
13: $\Psi \leftarrow$ GENECAND(Φ_l);
14: **for** each $S' \in \Psi$ **do**
15: **if** S' is generated from S_1 and S_2 **then**
16: $\widehat{G} \leftarrow G_k[S_1] \cap G_k[S_2]$;
17: Ψ_l.update(S',$< S', \widehat{G} >$);
18: **else** break;
19: output the communities of keyword sets in Φ_{l-1};

is because every vertex in $G_k[S']$ must has degree at least k. This observation implies, we can generate all the candidate keyword sets directly by using the query vertex q and q's neighbors, without touching other vertices.

Specifically, we consider q and q's neighbor vertices. For each vertex v, we only select the keywords, which are contained by S and at least k of its neighbors. Then we use these selected keywords to form an itemset, in which each item is a keyword. After this step, we obtain a list of itemsets. Then we apply the well studied frequent pattern mining algorithms (e.g., Apriori [13] and FP-Growth [14]) to find the frequent keyword combinations, each of which is a candidate keyword set. Since our goal is to generate keyword combinations shared by at least k neighbors, we set the minimum support as k. In this paper, we use the well-known FP-Growth algorithm [14].



(a) a query vertex

k=3

Set	Keyword sets
Ψ_1	{v},{x},{y},{z}
Ψ_2	{x,y},{x,z},{y,z}
Ψ_3	{x,y,z}

(b) candidates

Figure 6: An example of candidate generation.

EXAMPLE 6. *Consider a query vertex Q (k=3, S=$\{v, x, y, z\}$) with 6 neighbors in Figure 6(a), where the selected keywords of each vertex are listed in the curly braces. By FP-Growth, 8 candidate keyword sets) are generated, as shown in Figure 6(b). Ψ_i denotes the set of keyword sets with sizes being i.*

2. Verification of candidate keyword sets. As candidates can be obtained using S and q's neighbors directly, we can verify them either incrementally as that in `Inc-S`, or in a decremental manner (larger candidate keyword sets first and smaller candidate keyword sets later). In this paper, we choose the latter manner. The rationale behind is that, for any two keyword sets $S_1 \subseteq S_2$, the number of vertices containing S_2 is usually smaller than that of S_1, which implies S_2 can be verified more efficiently than S_1.

Based on the above discussions, we design `Dec` as shown in Algorithm 4. We first generate candidate keyword sets using S and

Algorithm 4 Query algorithm: `Dec`

1: **function** QUERY(G, $root$, q, k, S)
2: generate $\Psi_1, \Psi_2, \cdots, \Psi_h$ using S and q's neighbors;
3: find the subtree root node r_k;
4: create $R_1, R_2, \cdots, R_{h'}$ by using subtree rooted at r_k;
5: $l \leftarrow h$; $Q \leftarrow \emptyset$;
6: $\widehat{R} \leftarrow R_h \cup \cdots \cup R_{h'}$;
7: **while** $l \geq 1$ **do**
8: **for** each $S' \in \Psi_l$ **do**
9: find $G[S']$ from the subgraph induced on $\widehat{R}$;
10: find $G_k[S']$ from $G[S']$;
11: **if** $G_k[S']$ exists **then** Q.add($G_k[S']$);
12: **if** Q=$\emptyset$ **then**
13: $l \leftarrow l - 1$;
14: $\widehat{R} \leftarrow \widehat{R} \cup R_l$;
15: **else** break;
16: output communities in Q;

q's neighbors by FP-Growth algorithm (line 2). Then, we apply `core-locating` to find the root (with core number k) of the subtree from CL-tree, whose corresponding k-$\widehat{core}$ contains q (line 3). Next, we traverse the subtree rooted at r_k and find vertices which share keywords with q (line 4). R_i denote the sets of vertices sharing i keywords with q. Then, we initialize l as h (line 5), as we verify keyword sets with the largest size h first. We maintain a set $\widehat{R}$ dynamically, which contains vertices sharing at least l keywords with q (line 6). In the while loop, we examine candidate keyword sets in the decremental manner. For each candidate $S' \in \Psi_l$, we first try to find $G[S']$, then find $G_k[S']$, and put $G_k[S']$ into Q if it exists (lines 8-11). Finally, if we have found at least one qualified community, we stop at the end of this loop and output Q; otherwise, we examine smaller candidate keyword sets in next loop.

7. EXPERIMENTS

We now present the experimental results. Section 7.1 discusses the setup. We discuss the results in Sections 7.2 and 7.3.

7.1 Setup

We consider four real datasets. For *Flickr* [4], a vertex represents a user, and an edge denotes a "follow" relationship between two users. For each vertex, we use the 30 most frequent tags of its associated photos as its keywords. For *DBLP* [5], a vertex denotes an author, and an edge is a co-authorship relationship between two authors. For each author, we use the 20 most frequent keywords from the titles of her publications as her keywords. In the *Tencent* graph provided by the KDD contest 2012 [6], a vertex is a person, an organization, or a microblog group. Each edge denotes the friendship between two users. The keyword set of each vertex is extracted from a user's profile. For the *DBpedia* [7], each vertex is an entity, and each edge is the relationship between two entities. The keywords of each entity are extracted by the Stanford Analyzer and Lemmatizer. Table 3 shows the number of vertices and edges, the k_{max} value, a vertex's average degree $\widehat{d}$, and its keyword set size $\widehat{l}$.

To evaluate ACQs, we set the default value of k to 6. The input keyword set S is set to the whole set of keywords contained by the query vertex. For each dataset, we randomly select 300 query vertices with core numbers of 6 or more, which ensures that there is a

[4] `https://www.flickr.com/`
[5] `http://dblp.uni-trier.de/xml/`
[6] `http://www.kddcup2012.org/c/kddcup2012-track1`
[7] `http://dbpedia.org/datasets`

Table 3: Datasets used in our experiments.

Dataset	Vertices	Edges	k_{max}	$\widehat{d}$	$\widehat{l}$
Flickr	581,099	9,944,548	152	17.11	9.90
DBLP	977,288	3,432,273	118	7.02	11.8
Tencent	2,320,895	50,133,369	405	43.2	6.96
DBpedia	8,099,955	71,527,515	95	17.66	15.03

k-core containing each query vertex. Each data point is the average result for these 300 queries. We implement all the algorithms in Java, and run experiments on a machine having a quad-core Intel i7-3770 processor, and 32GB of memory, with Ubuntu installed.

7.2 Results on Effectiveness

We now study the effectiveness of ACQ, and compare it with existing CD and CS methods. We then discuss a case study.

7.2.1 ACQ Effectiveness

We first define two measures, namely CMF and CPJ, for evaluating the keyword cohesiveness of the communities. Let $C(q)$={$C_1, C_2, \cdots, C_{\mathcal{L}}$} be the set of $\mathcal{L}$ communities returned by an algorithm for a query vertex $q \in V$ (Note that S=$W(q)$).

• **Community member frequency (CMF)**: this is inspired by the classical document frequency measure. Consider a keyword x of q's keyword set $W(q)$. If x appears in most of the vertices (or members) of a community C_i, then we regard C_i to be highly cohesive. The CMF uses the occurrence frequencies of q's keywords in C_i to determine the degree of cohesiveness. Let $f_{i,h}$ be the number of vertices of C_i whose keyword sets contain the h-th keyword of $W(q)$. Then, $\frac{f_{i,h}}{|C_i|}$ is the relative occurrence frequency of this keyword in C_i. The CMF is the average of this value over all keywords in $W(q)$, and all communities in $C(q)$:

$$CMF(C(q)) = \frac{1}{\mathcal{L} \cdot |W(q)|} \sum_{i=1}^{\mathcal{L}} \sum_{h=1}^{|W(q)|} \frac{f_{i,h}}{|C_i|} \quad (3)$$

Notice that $CMF(C(q))$ ranges from 0 to 1. The higher its value, the more cohesive is a community.

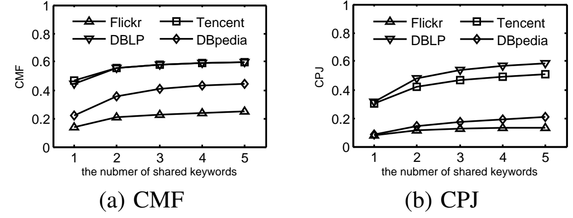
• **Community pair-wise Jaccard (CPJ)**: this is based on the similarity between the keyword sets of any pair of vertices of community C_i. We adopt the Jaccard similarity, which is commonly used in the IR literature. Let $C_{i,j}$ be the j-th vertex of C_i. The CPJ is then the average similarity over all pairs of vertices of C_i, and all communities of $C(q)$:

$$CPJ(C(q)) = \frac{1}{\mathcal{L}} \sum_{i=1}^{\mathcal{L}} \left[\frac{1}{|C_i|^2} \sum_{j=1}^{|C_i|} \sum_{k=1}^{|C_i|} \frac{|W(C_{i,j}) \cap W(C_{i,k})|}{|W(C_{i,j}) \cup W(C_{i,k})|} \right] \quad (4)$$

The $CPJ(C(q))$ value has a range of 0 and 1. A higher value of $CPJ(C(q))$ implies better cohesiveness.

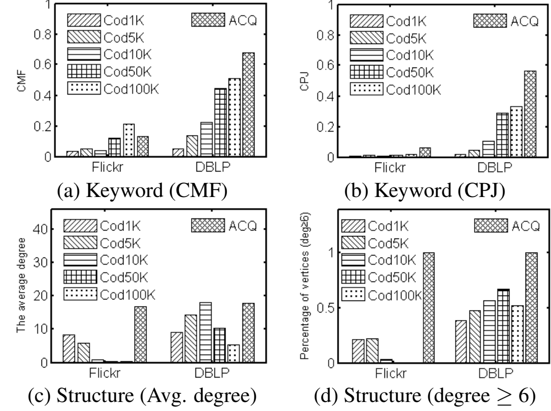
1. Effect of common keywords. We examine the impact of the AC-label length (i.e., the number of keywords shared by all the vertices of the AC) on keyword cohesiveness. We collect ACs containing one to five keywords, and then group the ACs according to their AC-label lengths. The average CMF and CPJ value of each group is shown in Figure 7. For all the datasets, when the AC-label lengths increase, both CMJ and CPJ value rises. This justifies the use of the maximal AC-label length as the criterion of returning an AC in our ACQ Problem.

2. Comparison with existing CD methods As mentioned ahead, the existing CD methods for attributed graph can be adapted for community search. We choose to adapt CODICIL [24] for comparison. The main reasons are: (1) it has been tested on the ever reported largest attributed graph (vertex number:3.6M); (2) it identifies communities of comparable or superior quality than those of many existing methods like [22, 31]; and (3) it runs faster than many existing methods. Since CODICIL needs users to specify the number of clusters expected, we set the numbers as 1K, 5K, 10K, 50K and 100K. The corresponding adapted algorithms are named as Cod1K, $\cdots$, Cod100K respectively. Other parameter settings are the same as those in [24]. We first run these algorithms offline to obtain all the communities. Given a query vertex q, they return communities containing q as the results.



(a) CMF (b) CPJ

Figure 7: AC-label length.



(a) Keyword (CMF) (b) Keyword (CPJ)

(c) Structure (Avg. degree) (d) Structure (degree $\geq$ 6)

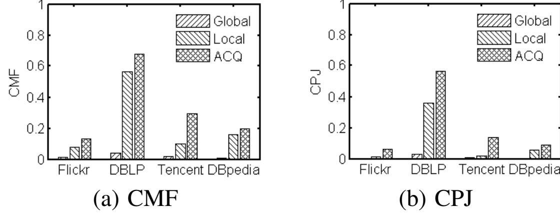
Figure 8: Comparing with community detection method.

We consider both keyword and structure for evaluating community quality. (1) *Keyword:* Figures 8(a) and (b) show that ACQ (implemented by Dec) always performs the best, in terms of CMF and CPJ. (2) *Structure:* As CODICIL has no guarantee on vertices' minimum degrees, it is unfair to compare them using this metric. We intuitively compare their structure cohesiveness by reporting the average degree of the vertices in the communities and the percentage of vertices having degrees of 6 or more. When the number of clusters in CODICIL is too large or too small, the structure cohesiveness becomes weak, as shown in Figures 8(c) and (d). ACQ always performs better than CODICIL, even when its number of cluster is well set (e.g., Cod10K and Cod50K on DBLP dataset).

3. Comparison with existing CS methods. The existing methods mainly focus on non-attributed graphs. We implement two state-of-the-art methods: Global [27] and Local [5]. Both of them use the metric minimum degree, we thus focus on the keyword cohesiveness. Figure 9 shows the CMF and CPJ values for the four datasets. We can see that the keyword cohesiveness of ACQ is superior to both Global and Local, because ACQ considers vertex keywords, while Global and Local do not.

7.2.2 A Case Study

We next perform a case study on the DBLP dataset, in which we consider two renowned researchers in database and data mining: Jim Gray and Jiawei Han. We use $k = 4$ here. We use Cod50K to

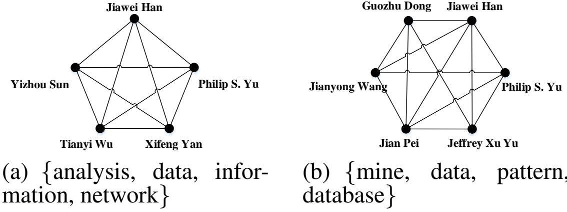


(a) CMF (b) CPJ

Figure 9: Comparing with community search methods.

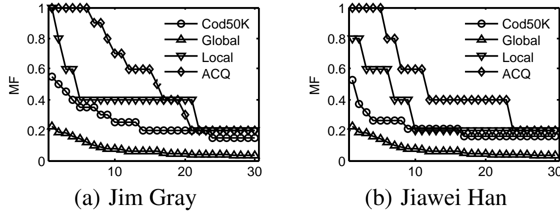
represent `CODICIL` for further analysis. We mainly consider the input query keyword set S, keywords and sizes of communities.

1. Effect of S. Figure 10 shows two ACs of Jiawei (The AC-labels are shown in the captions), where the set S are set as {analysis, mine, data, information, network} and {mine, data, pattern, database} respectively. These two groups of Jiawei's collaborators are involved in graph analysis (Figure 10(a)) and pattern mining (Figure 10(b)). Although these researchers all have close co-author relationship with Jiawei, the use of the input keyword set S enables the identification of communities with different research themes. For Jim, we can obtain similar results as discussed in Section 1 (Figure 2). While for `CODICIL`, it is not clear how to consider the input keyword set S, and we thus do not show the results.



(a) {analysis, data, information, network} (b) {mine, data, pattern, database}

Figure 10: Jiawei Han's ACs.



(a) Jim Gray (b) Jiawei Han

Figure 11: Frequency distribution of keywords.

2. Keyword analysis. We analyze the frequency distribution of keywords in their communities. Specifically, given a keyword w_h, we define the member frequency (MF) of w_h as: $MF(w_h, C(q)) = \frac{1}{\mathcal{L}}\sum_{i=1}^{\mathcal{L}} \frac{f_{i,h}}{|C_i|}$. The MF measures the occurrence of a keyword in $C(q)$. For each C_q generated by an algorithm, we select 30 keywords with the highest MF values. We report the MF of each keyword in descending order of their MF values in Figure 11. We see that `ACQ` has the highest MF values for the top 20 keywords. Thus, the keywords associated with the communities generated by `ACQ` tend to repeat among the community members.

The number of distinct keywords of `ACQ` communities is also the fewest, as shown in Table 4. For example, the k-$\widehat{core}$ returned by `Global` has over $139K$ distinct keywords, about 2,300 times more than that returned by `ACQ` (less than 60 keywords). While the semantics of the k-$\widehat{core}$ can be difficult to understand, the small number of distinct keywords of AC makes it easier to understand why the community is so formed. We further report the keywords with the 6 highest MF values in the communities in Tables 5 and 6. We can see that, the top-6 keywords are highly related to their input query keyword sets. Note that the top-6 keywords of `Global` are the same for both Jim and Jiawei, as they are in the same k-$\widehat{core}$. Thus, they cannot be used to characterize the communities specifically related to Jim and Jiawei. The overall results show that, `ACQ` performs better than other methods.

Table 4: # distinct keywords of communities.

Researcher	`Cod50K`	`Global`	`Local`	`ACQ`
Jim Gray	134	139,881	60	44
Jiawei Han	140	139,881	58	54

3. Effect of k on community size. We vary the value of k and report the average size of communities in Figure 12. We can see that the communities returned by `Global` are extremely large (more than 10^5), which can make them difficult for a query user to analyze. The community size of `Local` increases sharply when k=8. In this situation, `Local` returns the same community as `Global`. The size of an AC is relatively insensitive to the change of k, as AC contains around a hundred vertices for a wide range of values of k.

4. GPM Results. We have examined communities produced by Graph Pattern Matching (GPM) algorithms on the DBLP dataset. We consider a "`Star-`a" pattern, which contains a vertices linked to the query vertex q. A keyword set S is randomly drawn from Jim Gray's keyword sets, i.e., $W(q)$. Each vertex of the pattern is associated with S. We vary the size of S; for each $|S|$, we obtain 100 different sets. We then execute the `Match` algorithm [9] with its default settings. Table 7 shows the fraction of GPM queries returning non-empty communities. When $|S| \geq 3$, only a small fraction of `Star-`a patterns yields non-empty subgraphs. These subgraphs vary, depending on the topology and keyword sets associated with the patterns. Hence, if GPM is used to find communities, the user has to be careful in choosing the right pattern.

Table 5: Top-6 keywords (Jim Gray).

Algo.	Keywords
`Cod50K`	server, archive, sloan, digital, database
`Global`	use, system, model, network, analysis, data
`Local`	database, system, multipetabyte, data, lsst, story
`ACQ`	sloan, digital, sky, data, sdss, server

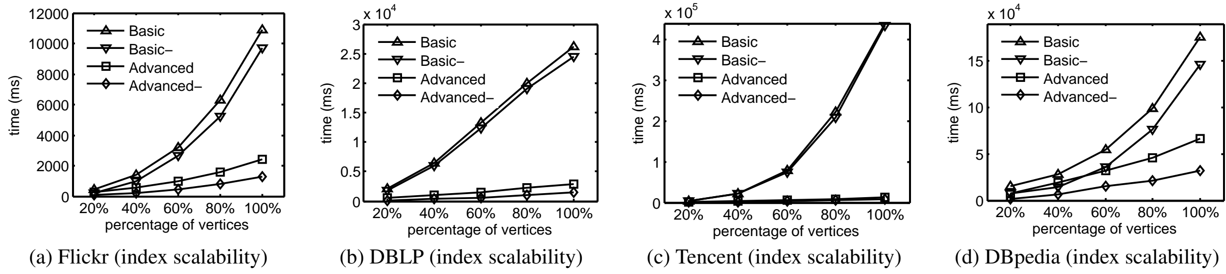
Table 6: Top-6 keywords (Jiawei Han).

Algo.	Keywords
`Cod50K`	information, mine, data, cube, text, network
`Global`	use, system, model, network, analysis, data
`Local`	scalable, topical, text, phrase, corpus, mine
`ACQ`	mine, analysis, data, information, network, heterogen

7.3 Results on Efficiency

For each dataset, we randomly select 20%, 40%, 60% and 80% of its vertices, and obtain four subgraphs induced by these vertex sets. For each vertex, we randomly select 20%, 40%, 60% and 80% of its keywords, and obtain four keyword sets.

1. Index construction. Figures 13(a)-13(d) compare the efficiency of `Basic` and `Advanced`. We study their main parts, which build the tree without considering keywords. We denote them by `Basic-` and `Advanced-`. Notice that `Advanced` performs consistently faster, and scales better, than `Basic`. When the subgraph size increases, the performance gap between `Advanced` and `Basic` is enlarged. Similar results can be observed between `Advanced-` and `Basic-`. In addition, we also run the CD method `CODICIL`, which takes 32 min, 2 min, 1 day, and 3+ days (we stop



(a) Flickr (index scalability) (b) DBLP (index scalability) (c) Tencent (index scalability) (d) DBpedia (index scalability)

Figure 13: Efficiency results of index construction.

(a) Flickr (efficiency) (b) DBLP (efficiency) (c) Tencent (efficiency) (d) DBpedia (efficiency)

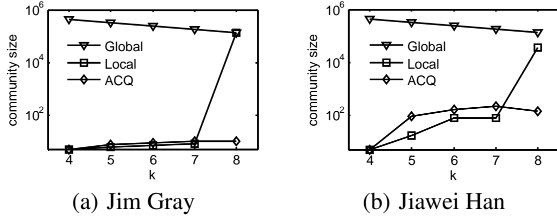
(e) Flickr (effect of k) (f) DBLP (effect of k) (g) Tencent (effect of k) (h) DBpedia (effect of k)

(i) Flickr (keyword scalability) (j) DBLP (keyword scalability) (k) Tencent (keyword scalability) (l) DBpedia (keyword scalability)

(m) Flickr (vertex scalability) (n) DBLP (vertex scalability) (o) Tencent (vertex scalability) (p) DBpedia (vertex scalability)

(q) Flickr (set S) (r) DBLP (set S) (s) Tencent (set S) (t) DBpedia (set S)

Figure 14: Efficiency results of community search.



(a) Jim Gray (b) Jiawei Han

Figure 12: Community size.

Table 7: % GPM queries that return at least one subgraph.

Researcher	$\lvert S\rvert$	Star-6	Star-8	Star-10
Jim Gray	1	93%	93%	93%
	2	68%	60%	56%
	3	21%	16%	14%
	4	5.0%	3.0%	2.0%
	5	0.0%	0.0%	0.0%

it after 3 days) to cluster the vertices of Flickr, DBLP, Tencent and DBpedia offline respectively.

2. Efficiency of CS methods. Figures 14(a)-14(d) compares our best algorithm Dec with existing CS methods. We see that Local performs faster than Global for most cases. Also, Dec, which uses the CL-tree index, is the fastest.

3. Effect of k. Figures 14(e)-14(h) compare the query efficiency under different k. A lower k renders a larger subgraph, so as the time costs, for all the algorithms. Note that basic-g performs faster than basic-w, but are slower than index-based algorithms. Inc-T performs better than Inc-S, and Dec performs the best. The performance gaps decrease as k increases.

4. ACQ scalability w.r.t. keyword. Figures 14(i)-14(l) examines scalability over the fraction of keywords for each vertex. All the vertices are considered. The running times of the algorithms increase as more keywords are involved. Dec performs the best.

Next, we present the results about DBpedia, the largest dataset used in our experiments. Results for other datasets are similar, and are reported in our report [29] due to space constraints.

5. ACQ scalability w.r.t. vertex. Figures 14(m) and 14(p) reports the scalability over different fraction of vertices. All the keywords of each vertex are considered. Again, Dec scales the best.

6. Effect of size of S. For each query vertex, we randomly select 1, 3, 5, 7 and 9 keywords to form the query keyword set S. As Dec performs better than Inc-S and Inc-T, we mainly compare Dec with the baseline solutions. Figures 14(q)- 14(t) show that the cost of all algorithms increase with the $|S|$. Also, Dec is 1 to 3 order-of-magnitude faster than basic-g and basic-w.

7. Effect of invertedList. We implement Inc-S$\star$ and Inc-T$\star$, which are respective variants of Inc-S and Inc-T, without the invertedList on each CL-tree node. Figure 15 shows the efficiency of Inc-S (Inc-T) and Inc-S$\star$ (Inc-T$\star$). We see that Inc-S (Inc-T) is 1 to 2 order of magnitude faster than Inc-S$\star$ (Inc-T$\star$). The keyword-checking operation, which uses the invertedList, is frequently performed. The use of invertedList thus improves the performance of our algorithms significantly.

8. Non-attributed graphs. We also test Dec and Local on non-attributed graphs, by ignoring the keywords of the graph dataset. The experimental results are shown in Figure 16. For Flickr, Tencent and DBpedia, Dec is consistently faster than Local. In Dec, cores are organized into the CL-tree structure. Because the height of the CL-tree is not very high (lower than 405 for all datasets), the core-locating operation can be done quickly. For DBLP, Dec is also faster than Local, except when k=4. In DBLP, a paper often has few (around 3 to 5) co-authors. Since an author may be closely related to a few co-authors, finding a 4-$\widehat{core}$ in Local can be done efficiently through local expansion. From these experiments, we conclude that Dec can also be efficiently executed on non-attributed graphs.

8. CONCLUSIONS

An AC is a community that exhibits structure and keyword cohesiveness. To facilitate ACQ evaluation, we develop the CL-tree index and its query algorithms. Our experimental results show that ACs are easier to interpret than those of existing community detection/search methods, and they can be "personalized". Our solutions are also faster than existing community search algorithms. In the future, we will study how graph pattern matching techniques can be used to solve the ACQ problem. We also plan to extend our solutions to support directed and dynamic graphs. We will investigate algorithms for maintaining the CL-index. We will study the use of other measures of structure cohesiveness (e.g., k-truss, k-clique) and keyword cohesiveness (e.g., Jaccard similarity and string edit distance) in the ACQ definition.

9. REFERENCES

[1] https://en.wikipedia.org/wiki/Disjoint-set_data_structure.

[2] V. Batagelj and M. Zaversnik. An o(m) algorithm for cores decomposition of networks. *arXiv*, 2003.

[3] G. Bhalotia et al. Keyword searching and browsing in databases using banks. In *ICDE*, 2002.

[4] W. Cui et al. Online search of overlapping communities. In *SIGMOD*, 2013.

[5] W. Cui et al. Local search of communities in large graphs. In *SIGMOD*, 2014.

[6] B. Ding et al. Finding top-k min-cost connected trees in databases. In *ICDE*, 2007.

[7] S. N. Dorogovtsev et al. K-core organization of complex networks. *Physical review letters*, 2006.

[8] T. et al. Fast best-effort pattern matching in large attributed graphs. In *KDD*, 2007.

[9] W. Fan et al. Graph pattern matching: from intractable to polynomial time. *PVLDB*, 2010.

[10] W. Fan et al. Incremental graph pattern matching. In *SIGMOD*, 2011.

[11] W. Fan et al. Association rules with graph patterns. *PVLDB*, 8(12):1502–1513, 2015.

[12] S. Fortunato. Community detection in graphs. *Physics Reports*, 486(3):75–174, 2010.

[13] J. Han, M. Kamber, and J. Pei. *Data mining: concepts and techniques*. Elsevier, 2011.

[14] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD*, 2000.

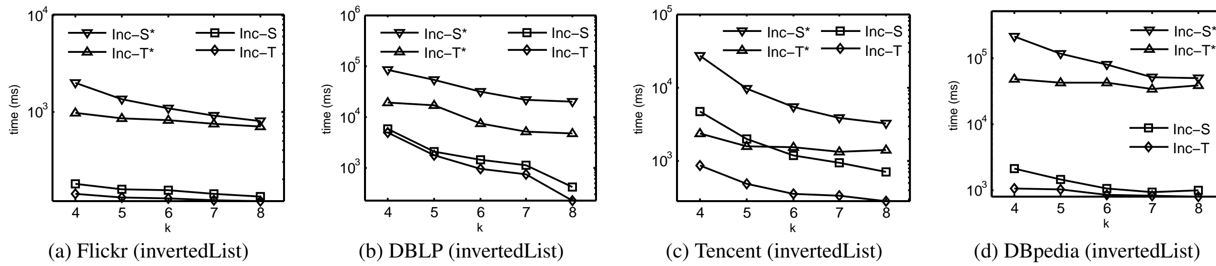
[15] H. He et al. Blinks: ranked keyword searches on graphs. In *SIGMOD*, 2007.

[16] X. Huang et al. Querying k-truss community in large and dynamic graphs. In *SIGMOD*, 2014.

[17] V. Kacholia et al. Bidirectional expansion for keyword search on graph databases. In *VLDB*, 2005.

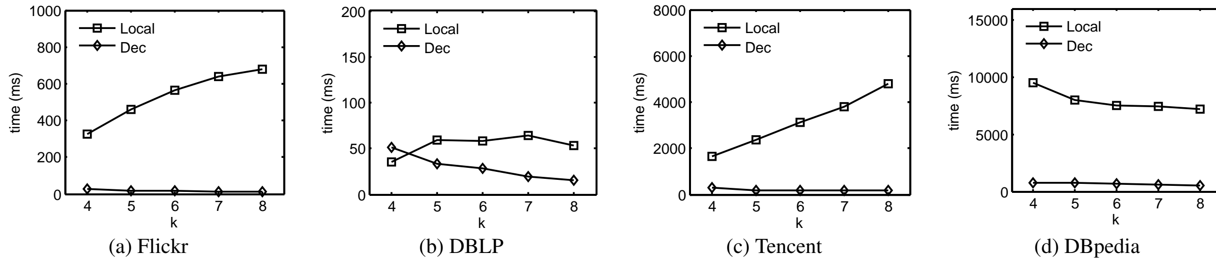
[18] M. Kargar and A. An. Keyword search in graphs: Finding r-cliques. *PVLDB*, 4(10):681–692, 2011.

[19] R.-H. Li, L. Qin, J. X. Yu, and R. Mao. Influential community search in large networks. In *PVLDB*, 2015.



(a) Flickr (invertedList) (b) DBLP (invertedList) (c) Tencent (invertedList) (d) DBpedia (invertedList)

Figure 15: Effect of invertedList.



(a) Flickr (b) DBLP (c) Tencent (d) DBpedia

Figure 16: `Dec` **vs** `Local`**.**

[20] R.-H. Li, J. X. Yu, and R. Mao. Efficient core maintenance in large dynamic graphs. *TKDE*, 2014.

[21] Y. Liu et al. Topic-link lda: joint models of topic and author community. In *ICML*, 2009.

[22] R. M. Nallapati et al. Joint latent topic models for text and citations. In *KDD*, 2008.

[23] M. Newman et al. Finding and evaluating community structure in networks. *Physical review E*, 2004.

[24] Y. Ruan et al. Efficient community detection in large networks using content and links. In *WWW*, 2013.

[25] M. Sachan et al. Using content and interactions for discovering communities in social networks. In *WWW*, 2012.

[26] S. B. Seidman. Network structure and minimum degree. *Social networks*, 5(3):269–287, 1983.

[27] M. Sozio et al. The community-search problem and how to plan a successful cocktail party. In *KDD*, 2010.

[28] Z. Xu et al. A model-based approach to attributed graph clustering. In *SIGMOD*, 2012.

[29] Y. Fang et al. Effective community search for large attributed graphs (technical report). `http://www.cs.hku.hk/research/techreps/document/TR-2016-01.pdf`.

[30] J. Yang et al. Community detection in networks with node attributes. In *ICDM*, 2013.

[31] T. Yang et al. Combining link and content for community detection: a discriminative approach. In *KDD*, 2009.

[32] J. X. Yu et al. Keyword search in databases. *Synthesis Lectures on Data Management*, 2009.

[33] Y. Zhou et al. Graph clustering based on structural/attribute similarities. *VLDB*, 2009.

APPENDIX

A. PROOFS OF LEMMAS

LEMMA 1 (ANTI-MONOTONICITY). *Given a graph G, a vertex $q \in G$ and a set S of keywords, if there exists a subgraph $G_k[S]$, then there exists a subgraph $G_k[S']$ for any subset $S' \subseteq S$.*

PROOF. Based on the definition of $G_k[S]$, each vertex of $G_k[S]$ contains S. Consider a new keyword set $S' \subseteq S$. We can easily conclude that, each vertex of $G_k[S]$ contains S' as well. Also, note that $q \in G_k[S]$. These two properties imply that there exists one subgraph of G, namely $G_k[S]$, with core number at least k, such that it contains q and every vertex of it contains keyword set S'. It follows that there exists such a subgraph with maximal size (*i.e.*, $G_k[S']$). □

PROPOSITION 1. *For any keyword set S, and vertex q, if $G_k[S]$ exists, then $G_k[S] \subseteq G_k[S']$ for any subset $S' \subseteq S$.*

PROOF. Since $G_k[S]$ contains vertex q and every vertex in $G_k[S]$ contains S' (due to $S' \subseteq S$), then $G_k[S] \cup G_k[S']$ also contains vertex q and every vertex in it contains S'. In addition, the core numbers of $G_k[S]$ and $G_k[S']$ are at least k, it follows that the core number of $G_k[S] \cup G_k[S']$ is at least k. Based on the definition of $G_k[S']$, we have $G_k[S] \cup G_k[S'] \subseteq G_k[S']$. It follows that $G_k[S] \subseteq G_k[S']$. □

LEMMA 2. *Given two subgraphs $G_k[S_1]$ and $G_k[S_2]$ of a graph G, for a new keyword set S' generated from S_1 and S_2 (i.e., $S' = S_1 \cup S_2$), if $G_k[S']$ exists, then it must appear in a k-$\widehat{core}$ with core number at least*

$$max\{core_G[G_k[S_1]], core_G[G_k[S_2]]\}. \tag{3}$$

PROOF. Since S' is generated from S_1 and S_2, then $S_1 \subseteq S'$ and $S_2 \subseteq S'$. Based on Proposition 1, we have $G_k[S'] \subseteq G_k[S_1]$. With such a containment relationship, it follows that $min\{core_G[v]| v \in G_k[S_1]\} \leq min\{core_G[v]|v \in G_k[S']\}$. Hence, the core number of $G_k[S']$ is at least the core number of $G_k[S_1]$. Formally, $core_G[G_k[S_1]] \leq core_G[G_k[S']]$. For the same reason, $core_G[G_k[S_2]] \leq core_G[G_k[S']]$. It directly follows the lemma. □

LEMMA 3. *Given a connected graph $G(V, E)$ with n=$|V|$ and m=$|E|$, if $m - n < \frac{k^2-k}{2} - 1$, there is no k-$\widehat{core}$ in G.*

PROOF. From Definition 1, we can easily conclude that, for any specific k, a k-$\widehat{core}$ has at least k+1 vertices. Since each vertex in a specific k-$\widehat{core}$ has at least k edges, the minimum number of edges in a k-$\widehat{core}$ is $\frac{(k+1)k}{2}$.

Consider a connected graph, which contains a k-$\widehat{core}$ and has the minimum number of edges, where the k-core contains only $k+1$ vertices and all the rest $n-(k+1)$ vertices are connected with this k-$\widehat{core}$. The total number of edges is

$$\frac{(k+1)k}{2} + [n - (k+1)] = m \tag{4}$$

By simple transformation, we can conclude that, if $m - n < \frac{k^2-k}{2} - 1$, there is no k-$\widehat{core}$ in G. □

LEMMA 4. *Given two keyword sets S_1 and S_2, if $G_k[S_1]$ and $G_k[S_2]$ exist, we have*

$$G_k[S_1 \cup S_2] \subseteq G_k[S_1] \cap G_k[S_2]. \tag{5}$$

PROOF. Based on Proposition 1 and $S_1 \subseteq S_1 \cup S_2$, we have $G_k[S_1 \cup S_2] \subseteq G_k[S_1]$. For the same reason we have $G_k[S_1 \cup S_2] \subseteq G_k[S_2]$. It directly follows the lemma. □

B. DETAILS OF BASIC SOLUTIONS

Algorithms 5 and 6 present `basic-g` and `basic-w` respectively. The input of `basic-g` is a graph G, a query vertex q, an integer k and a set S. It first generates a set, Ψ, of candidate keyword sets, each of which contains a single keyword of S (line 2). Then, it finds the k-$\widehat{core}$, $\mathcal{C}_k$, containing q from the graph G. In the while loop (lines 4-14), it first initializes an empty set Φ (line 5), which is used to collect all the qualified keyword sets. Then for each candidate keyword set $S' \in \Psi$, it finds $G[S']$ from $\mathcal{C}_k$ by considering the keyword constraint. After that, it finds $G_k[S']$ from $G[S']$ (lines 7-8), and put it into Φ if $G_k[S']$ exists (lines 9-10). After checking all the candidate keyword sets in Ψ, if there are at least one qualified keyword sets in Φ, it generates a new set Ψ of larger candidate keyword sets by calling GENECAND(Φ) (see Appendix C) and continues to checking longer candidate keyword sets in next loop; otherwise, it stops and outputs all the communities of the latest verified keyword sets as the target ACs.

Algorithm 5 Basic solution: `basic-g`

```
 1: function QUERY(G, q, k, S)
 2:     init Ψ using S;
 3:     find the k-core, C_k, containing q from G;
 4:     while true do
 5:         Φ ← ∅;
 6:         for each S' ∈ Ψ do
 7:             find G[S'] from C_k;
 8:             find G_k[S'] from G[S'];
 9:             if G_k[S'] exists then
10:                 Φ.add(S');
11:         if Φ ≠ ∅ then
12:             Ψ ← GENECAND(Φ);
13:         else
14:             break;
15:     output the communities of keyword sets in Φ;
```

Algorithm 6 presents the pseudocodes of `basic-w`. It follows the main steps of `basic-g`, except that for each candidate keyword set S', it finds $G[S']$ from G directly, rather than from $\mathcal{C}_k$.

Algorithm 6 Basic solution: `basic-w`

```
 1: function QUERY(G, q, k, S)
 2:     init Ψ using S;
 3:     while true do
 4:         Φ ← ∅;
 5:         for each S' ∈ Ψ do
 6:             find G[S'] from G;
 7:             find G_k[S'] from G[S'];
 8:             if G_k[S'] exists then
 9:                 Φ.add(S');
10:         if Φ ≠ ∅ then
11:             Ψ ← GENECAND(Φ);
12:         else
13:             break;
14:     output the communities of keyword sets in Φ;
```

C. CANDIDATE GENERATION

Given a set Φ of qualified keyword sets, Algorithm 7 generates new candidate keyword sets incrementally by linking each pair of keyword sets. We first initialize Ψ as an empty set (line 2). Then for each pair, S_i and S_j, of keyword sets in Φ, we sort their keywords. If they differ only at the last keyword, then we generate a new keyword set S'=$S_i \cup S_j$, by a union operation (lines 3-6). According to Lemma 1, if any subset of S' does not appear in Φ, we

prune S'; otherwise, we regard it as a candidate and add it into Ψ (lines 7-8). Finally, we return Ψ as the results (line 9).

Algorithm 7 Generate candidate keyword sets

```
1: function GENECAND(Φ)
2:     Ψ ← ∅;
3:     for each S_i ∈ Φ do
4:         for each S_j ∈ Φ do
5:             if S_i and S_j differs at the last keyword then
6:                 initialize S'=S_i ∪ S_j;
7:                 if S' cannot be pruned by Lemma 1 then
8:                     Ψ.add(S');
9:     return Ψ;
```

D. ANCHORED UNION-FIND

Algorithm 8 presents the four functions of the anchored union-find (AUF) data structure.

Algorithm 8 Functions on the AUF data structure

```
1: function MAKESET(x)
2:     x.parent ← x;
3:     x.rank ← 0;
4:     x.anchor ← x;
5: function FIND(x)
6:     if x.parent=x then
7:         x.parent ← FIND(x.parent);
8:     return x.parent;
9: function UNION(x, y)
10:    xRoot ← FIND(x);
11:    yRoot ← FIND(y);
12:    if xRoot=yRoot then return ;
13:    if xRoot.rank < yRoot.rank then
14:        xRoot.parent ← yRoot;
15:    else if xRoot.rank > yRoot.rank then
16:        yRoot.parent ← xRoot;
17:    else
18:        yRoot.parent ← xRoot;
19:        xRoot.rank ← xRoot.rank + 1;
20: function UPDATEANCHOR(x, core_G[ ], y)
21:    xRoot ← FIND(x);
22:    if core_G[xRoot.anchor] > core_G[y] then
23:        xRoot.anchor ← y;
```

The functions FIND and UNION are exactly the same as that of the classical union-find data structure [1]. For function MAKESET, the only change made on the classical MAKESET is that, it adds a line of code for initializing $x.anchor$ as x (line 4). The function UPDATEANCHOR is used to update the anchor vertex of x's representative vertex. It first finds x's representative vertex by calling FIND (line 21). Then, if the core number of x' representative vertex is larger than that of the current input vertex y, it updates the anchor vertex of x's representative vertex as y.

Complexity analysis. The time complexities of functions FIND and UNION are $O(\alpha(n))$ [1], where $\alpha(n)$ is less than 5 for al-l practical values of n. In function MAKESET, since initializing $x.anchor$ can be done in $O(1)$, the time complexity of MAKESET is still $O(1)$. In function UPDATEANCHOR, as FIND can be completed in $O(\alpha(n))$ and updating anchor can be completed in $O(1)$, the total time cost of function UPDATEANCHOR is $O(\alpha(n))$.

E. DETAILS OF THE ADVANCED METHOD

Algorithm 9 presents the `advanced` method. Similar with `basic` method, we first conduct k-decomposition (line 2). Then, for each

Algorithm 9 Index construction: `advanced`

```
1: function BUILDINDEX(G(V, E))
2:     core_G[ ] ← k-core decomposition on G;
3:     for each v ∈ V do MAKESET(v);
4:     put vertices into sets V_0, V_1, ··· , V_{k_max};
5:     k ← k_max, map ← ∅;
6:     while k ≥ 0 do
7:         V' ← ∅;
8:         for each v ∈ V_k do V'.add(FIND(v));
9:         compute connected components for V_k ∪ V';
10:        for each component with vertex set C_i do
11:            create a node p_i using (k, {C_i − V'});
12:            for each v ∈ {C_i − V'} do
13:                map.add(v, p_i);
14:                for each u ∈ v's neighbor vertices do
15:                    if core_G[u] ≥ core_G[v] then
16:                        UNION(u, v);
17:                    if core_G[u] > core_G[v] then
18:                        uRoot ←FIND(u);
19:                        uAnchor ← uRoot.anchor;
20:                        p' ← map.get(uAnchor);
21:                        add p' to p's child List;
22:                vRoot ←FIND(v);
23:                if core_G[vRoot.anchor] > core_G[v] then
24:                    UPDATEANCHOR(vRoot, core_G[ ], v);
25:        k ← k − 1;
26:    build the root node root;
27:    build an inverted list for each tree node;
28:    return root.
```

vertex, we initialize an AUF tree node (line 3). We group all the vertices into sets (line 4), where set V_k contains vertices with core numbers being exactly k (line 5). Next, we initialize k as k_{max} and the vertex-node map map, where the key is a vertex and the value is a CL-tree node whose vertex set contains this vertex. In the while loop (lines 6-25), we first find the set V' of the representatives for vertices in V_k, then compute the connected components for vertex set $V_k \cup V'$ (lines 7-9). Next, we create a node p_i for each component (lines 10-11). For each vertex $v \in \{C_i - V'\}$, we add a pair (v, p_i) to the map (lines 12-13). Then for each of v's neighbor, u, if its core number is at least $core_G[v]$, we link u and v together in the AUF by a UNION operation (lines 14-16), and find p_i's child nodes using the anchor of the AUF tree (lines 17-21). After vertex v has been added into the CL-tree, we update the anchor (lines 22-24). Then we move to the upper level in next loop (line 25). After the while loop, we build the root node of the CL-tree (line 26). Finally, we build the inverted list for each tree node and obtain the built index (lines 27-28).

Complexity analysis. In Algorithm 9, lines 1-3 can be completed in $O(m)$ (We assume m¿n). In the while loop, the number of operations on each vertex and its neighbors are constant, and each can be done in $O(\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function. The keyword inverted lists of all the tree nodes can be computed in $O(n \cdot \widehat{l})$. Therefor, the CL-tree can be built in $O(m \cdot \alpha(n) + n \cdot \widehat{l})$. The space cost is $O(m + n \cdot \widehat{l})$, as maintaining an AUF takes $O(n)$.

F. INDEX MAINTENANCE

For CL-tree maintenance, we note that on our largest graph DB-pedia ($|V|$=8M, $|E|$=72M), the CL-tree can be constructed in 30 seconds. A simple solution is to rebuild the CL-tree for batches of graph updates periodically (say, every 15 minutes).

We now sketch an *incremental* solution which updates the CL-tree without rebuilding it. We consider two actions that affect the

CL-tree: (1) insertion or deletion of a keyword on a vertex's keyword set; and (2) insertion or removal of an edge on the graph G.

1. Keyword update. In the `Advanced` index construction method (Sec 5.2.2), we have built a vertex-node map, where each vertex is mapped to a CL-tree node. To insert a new keyword for a vertex v, we can first use the map to find the CL-tree node, which contains v, and then insert the keyword and vertex ID into the associated invertedList. To remove a keyword of a vertex, we can first find the CL-tree node, then look for the invertedList containing the keyword, and remove the vertex ID and/or keyword from the list. Note that only nodes that contain v need to be considered. Since the CL-tree is compressed, only the invertedList of a single CL-tree node containing v needs to be updated.

2. Edge update. The insertion or removal of an edge affects the core number of vertices in G. Notice that the structural cohesiveness of an AC follows the k-core definition. Therefore, it is possible to adopt the techniques that have been developed for updating k-cores. Here we consider [20], which presents an efficient k-core maintenance algorithm. As discussed in [20], given that a new edge (u, v) is inserted to G, only an vertex whose core number is c or $c + 1$ will be affected, where c is the minimum of the core numbers of u and v in G (i.e., $c = \min\{core_G[u], core_G[v]\}$). Based on this result, a possible CL-tree maintenance solution involves (1) finding the CL-tree nodes p_u and p_v containing u and v; (2) traversing the subtrees rooted at p_u and p_v and stopping at CL-tree nodes with core number $c + 2$ or more; (3) for each CL-tree node visited, update its links to child nodes. These steps only touches a small fraction of the CL-tree. The case of deletion can be done in a similar manner. We plan to study this solution in more detail, and conduct extensive experimental evaluation.

G. VARIANTS OF ACQ PROBLEM

In this section, we formally define two typical variants in Section G.1. Then we present the algorithms for these variants in Section G.2, and show the experimental results finally in Section G.3.

G.1 Variant Definitions

VARIANT 1. *Given a graph G, a positive integer k, a vertex $q \in V$ and a predefined keyword set S, return a subgraph G_q, the following properties hold:*

1. ***Connectivity**. $G_q \subseteq G$ is connected and contains q;*
2. ***Structure cohesiveness**. $\forall v \in G_q$, $deg_{G_q}(v) \geq k$;*
3. ***Keyword cohesiveness**. $\forall v \in G_q$, $S \subseteq W(v)$.*

Variant 1 can be applied to applications, which need communities having specific keyword constraints.

VARIANT 2. *Given a graph G, a positive integer k, a vertex $q \in V$, a predefined keyword set S, and a threshold $\theta \in [0,1]$, return a subgraph G_q, the following properties hold:*

1. ***Connectivity**. $G_q \subseteq G$ is connected and contains q;*
2. ***Structure cohesiveness**. $\forall v \in G_q$, $deg_{G_q}(v) \geq k$;*
3. ***Keyword cohesiveness**. $\forall v \in G_q$, it has at least $|S| \times \theta$ keywords in S.*

In Variant 2, the keyword cohesiveness is relaxed. This can be applied for cases when the keyword information is weak. We illustrate all the variants in Example 7.

EXAMPLE 7. *Consider Figure 3(a). Let q=A and k=2. For Variant 1, if the predefined keyword set is $\{x\}$, then the vertex set $\{A, B, C, D\}$ forms the target AC. For Variant 2, if the predefined keyword set is $\{x, y\}$ and the threshold is 50%, then the vertex set $\{A, B, C, D, E\}$ forms the target AC.*

G.2 Algorithms of Variants

1. Variant 1. In line with Problem 1, we first introduce the basic solutions without index, which are extended naturally from `basic-g` and `basic-w`, and are denoted by `basic-g-v1` and `basic-w-v1` respectively. Their details pseudocodes are presented in Algorithms 10 and 11.

Algorithm 10 Query algorithm: `basic-g-v1`

```
1: function QUERY(G, q, k, S)
2:     find the k-ĉore, C_k, containing q from G;
3:     collect a set V' of vertices containing S from C_k;
4:     find G[S] from the subgraph induced by V';
5:     find G_k[S] from G[S];
6:     output G_k[S] as the target AC.
```

Algorithm 11 Query algorithm: `basic-w-v1`

```
1: function QUERY(G, q, k, S)
2:     collect a set V' of vertices containing S from G;
3:     find G[S] from the subgraph induced by V';
4:     find G_k[S] from G[S];
5:     output G_k[S] as the target AC.
```

Algorithm 12 presents the index based algorithm, `SW`, for Variant 1. We first apply `core-locating` to find node r_k, whose corresponding k-$\widehat{core}$ contains q, from CL-tree (line 1). Then we traverse the subtree rooted at r_k, and collect a set V' of vertices containing S by applying `keyword-checking`. Next, we find $G[S]$ from the subgraph induced by vertices in V' (line 3), and find $G_k[S]$ from $G[S]$ (line 4). Finally, we output $G_k[S]$ as the target AC, if it exists (line 5).

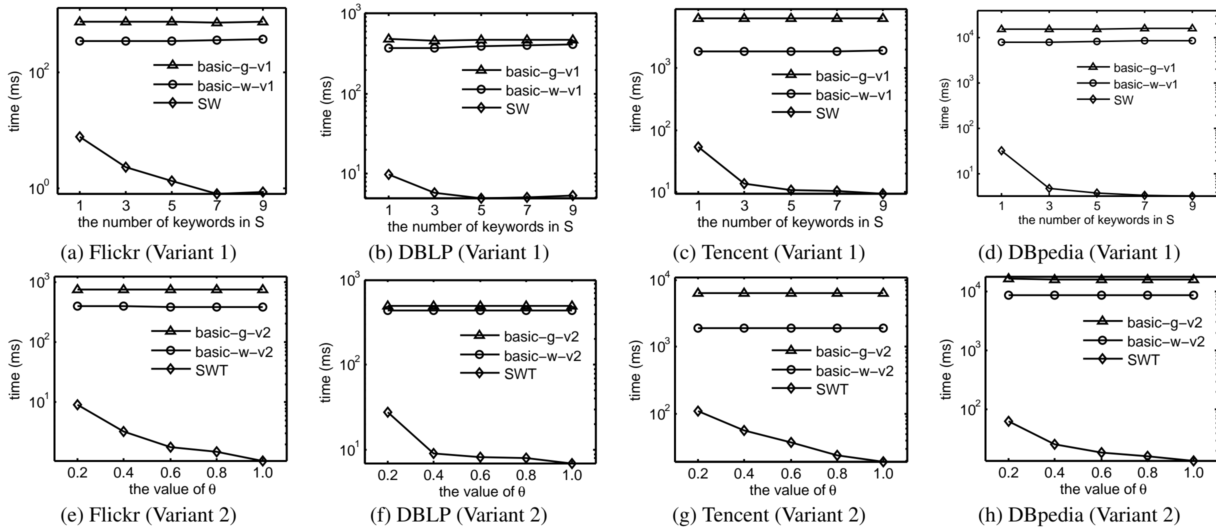
Algorithm 12 Query algorithm: `SW`

```
1: function QUERY(G, root, q, k, S)
2:     find the node r_k from the CL-tree index;
3:     traverse the subtree rooted at r_k and collect a set V' of
       vertices containing S by intersecting the inverted lists;
4:     find G[S] from the subgraph induced by V';
5:     find G_k[S] from G[S];
6:     output G_k[S] as the target AC.
```

2. Variant 2. We adapt the three algorithms of Variant 1 for answering the query of Variant 2. We denote the adapted algorithms by `basic-g-v2`, `basic-w-v2` and `SWT` (search by keywords with threshold constraint) respectively. The main adaptation is that, for the lines of codes on collecting vertices containing S, *i.e.*, line 3 in Algorithm 10, line 2 in Algorithm 11, and line 3 in Algorithm 12, we relax the constraint such that each collected vertex only needs to share at least $|S| \times \theta$ keywords in S. Then we find the target AC from the subgraph induced by these vertices.

G.3 Experiments on Variants

1. Case study of Variants 1 and 2. We search Jiawei Han's communities with explicit AC-label constraints in Variants 1 and 2. We consider two keyword sets, *i.e.*, S_1={stream, classification} and S_2={cube, information, cluster}. For Variant 1, we can only obtain a community for S_1 (see Figure 18(a)), and no communities for S_2. Note that the captions indicate the shared keywords of the

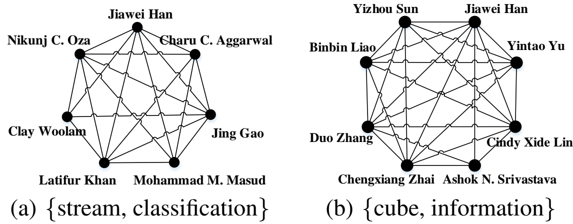


(a) Flickr (Variant 1) (b) DBLP (Variant 1) (c) Tencent (Variant 1) (d) DBpedia (Variant 1)

(e) Flickr (Variant 2) (f) DBLP (Variant 2) (g) Tencent (Variant 2) (h) DBpedia (Variant 2)

Figure 17: Efficiency results of Variants of ACQ problem.

communities. While for Variant 2 (θ=0.6), we can obtain two communities (see Figures 18(a) and 18(b)). This implies that, for Variant 2, although we cannot find a community with AC-label being exactly S_2, we still can find a community, in which each member contains at least 60% percentage of keywords of the input query keyword set.



(a) {stream, classification} (b) {cube, information}

Figure 18: Jiawei Han's communities on Variants 1 and 2.

2. Effect of $|S|$ in Variant 1. We consider query vertices having at least 9 keywords. For each of them, we randomly select 1, 3, 5, 7 and 9 keywords to form the query keyword sets, and answer the query of Variant 1 using `basic-g-v1`, `basic-w-v1` and `SW`. Figures 17(a)-17(d) show their efficiency. We can see that `SW` outperforms the basic solutions consistently. Also, the performance gap increases between `SW` and basic solutions as $|S|$ increases. This is because it uses the CL-tree index.

3. Effect of θ in Variant 2. For each query vertex, we randomly select 10 keywords to form set S, set θ as 0.2, 0.4 0.6, 0.8 and 1.0, and answer the query of Variant 2 using `basic-g-v2`, `basic-w-v2` and `SWT`. Figures 17(e)-17(h) show their efficiency. Similar with Variant 1, we can see that `SWT` performs the best, as it uses the CL-tree index.