HKU CS Tech Report TR-2016-01

Noname manuscript No.
(will be inserted by the editor)

Effective and Efficient Attributed Community Search

Yixiang Fang · Reynold Cheng · Yankai Chen · Siqiang Luo · Jiafeng Hu

Received: date / Accepted: date

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 ACQ), 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 index structure and three algorithms based on it. We further propose efficient algorithms for maintaining the index on dynamic graphs. Moreover, we study two problems that are related to the ACQ problem. We evaluate our solutions on six large graphs. Our results show that ACQ is 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.

Y. Fang, R. Cheng, Y. Chen, S. Luo, J. Hu
Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong
E-mail: {yxfang, ckcheng, ykchen, sqluo, jhu}@cs.hku.hk

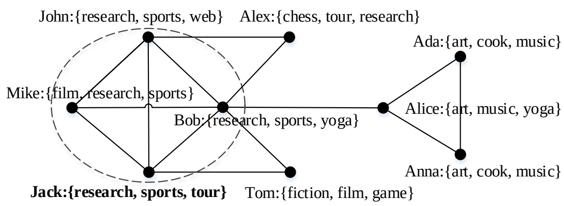


Fig. 1 Attributed graph and AC (circled).

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 [37, 7, 16, 20, 41, 21, 12]. 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 [34, 6, 5, 18, 28, 13]. Particularly, 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 [28, 13, 42, 27, 24, 37, 30, 38]. These solutions are not

Table 1 Classification of works in community retrieval.

Graph Type	Community detection (CD)	Community search (CS)
Non-attributed	[28,13]	[34,6,5,18,22,19]
Attributed	[42,27,24,37,30,38]	**ACQ**

"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 [34,6,5,18,19,3]. 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 of 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`, k=2 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 [34] 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.

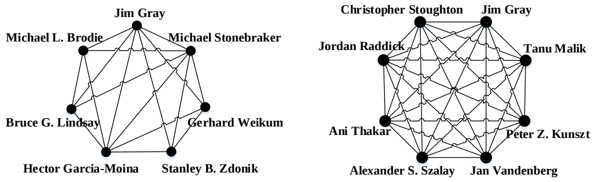
The personalization feature is also useful in marketing. Suppose that Mary, a yoga lover, is a customer of a gym. An ACQ can be issued on a social network, with Mary as the query vertex and S={"yoga"}. Since members of the AC contain the keyword "yoga", they can be the gym's advertising targets. On the 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 [42,30,27,24] 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 [28,13,34,6]. 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.

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 com-

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



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

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

Fig. 2 Two ACs of Jim Gray.

plexity 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. Based on the CL-tree index, we have developed three different ACQ algorithms, and they are able to achieve a superior performance. In practice, graphs are continuously evolving [29,2]. For instance, in the friendship network of Facebook, users may change their profiles, and make new friends or remove friendship. Thus the CL-tree index needs to be updated to reflect the changes in the graph. A straightforward method to handle the update is to rebuild the CL-tree from scratch. However, this can be very computationally expensive, especially when the updates are very frequent. To alleviate this issue, we have developed efficient algorithms to maintain the CL-tree index for dynamic graphs.

In addition, we have proposed two problems that are related to the ACQ problem, which are called *Approximate ACQ problem* (or ACQ-A) and *Multiple-vertex ACQ problem* (or ACQ-M) respectively. ACQ-A is an approximation version of the ACQ query, in which vertices of an AC do not need to exactly share the same keywords as that in ACQ. It relaxes the constraint on sharing common keywords; this could be very useful if vertices of the graph do not have much keyword information. ACQ-M generalizes the ACQ query for supporting multiple query vertices. It takes multiple query vertices as input and returns the ACs containing all of them. This could be very helpful if we want to find the ACs for a group of query vertices. For example, a database workshop organizer may be interested in inviting researchers who have a close relationship with both Jim Gray and Michael Stonebraker. To answer the queries for ACQ-A and ACQ-M, we have also developed efficient algorithms based on the CL-tree index.

We have performed extensive experiments on six large real graph datasets. 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 faster than solutions that do not use the CL-tree. We have also experimentally evaluated the index maintenance algorithms and the results show that they are very efficient. Moreover, we perform the queries of the ACQ-A and ACQ-M problems, and the results show that our index-based algorithms are much faster than the baseline algorithms. In addition, our approaches achieve 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. In Section 7, we discuss how to maintain the CL-tree index for dynamic graphs. We present the query algorithms in Section 6. In Section 8, we introduce two problems related to the ACQ problem and the corresponding query algorithms. Our experimental results are reported in Section 9. We conclude in Section 10.

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 [28,13], 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. [42] considered both links and keywords of vertices to compute the vertices' pairwise similarities, and then clustered the graph. Ruan et al. [30] 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 [27,24], the `Link-PLSA-LDA` and `Topic-Link LDA` models jointly model vertices' content and links based on the `LDA` model. In [37], the attributed graph is clustered based on probabilistic inference. In [31], the topics, interaction types and the social connections are considered for discovering communities. `CESNA` [39] detects overlapping communities by assuming communities "generate" both the link and content. A discriminative approach [40] 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 [34,6,22,5,18] have developed fast algorithms to obtain a community for q. To measure the structure cohesiveness of a community, the *minimum degree* is often used [34,6,22]. Sozio et al. [34] proposed the first algorithm `Global` to find the k-$\widehat{core}$ containing q. Cui et al. [6] 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 [5], k-truss [18] and edge connectivity [17], have also been considered for searching communities. A recent work [22] 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. An earlier version of this paper can be found in [11].

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 [7,20]. Recent work studies the use of a subgraph of G as the query output [21]. 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, these solutions are not designed to find ACs.

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 [34,22,37, 30]. 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.

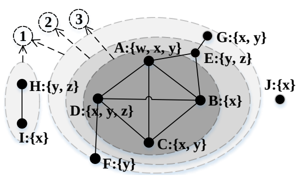
Table 2 Symbols and meanings.

Symbol	Meaning
$G(V,E)$	A 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 $\forall v \in G[S']$, $S' \subseteq W(v)$
$G_k[S']$	The largest connected subgraph of G s.t. $q \in G_k[S']$ and $\forall v \in G_k[S']$, $deg_{G_k[S']} v \geq k$ and $S' \subseteq W(v)$

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 [34,33,4,8,6,22]. This is used in the k-core and the AC. Let us discuss the k-core first.

Definition 1 **(k-core [33,4])** 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 [4], and its connected components, denoted by k-$\widehat{core}$s, are usually the "communities" returned by k-$\widehat{core}$ search algorithms.



(a) graph

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

(b) core number

Fig. 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" [4]: 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 in 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 [4], 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 $q \in G_q$;
• **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 [34]. 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 9) 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]

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

There are other candidates for structure cohesiveness (e.g., k-truss, k-clique) and *keyword cohesiveness* (e.g., Jaccard similarity and string edit distance). An AC can also be defined in different ways. For example, an ACQ user may specify that an AC returned must have vertices that contain a specific set of keywords. An interesting direction is to extend ACQ to support for these criteria, and study their effectiveness.

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 Appendix A. 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 Appendix B.

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.

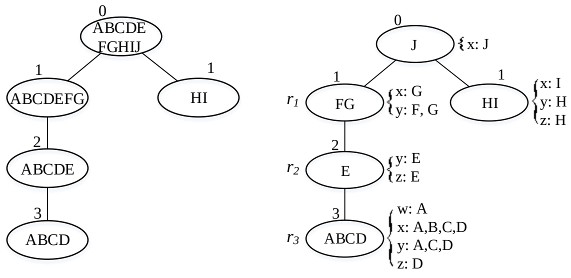
5.1 Index Overview

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.

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 p contains five elements:



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

Fig. 4 An example CL-tree index.

• *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;

• *fatherNode*: the father node of p.

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 [34].

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

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

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

Algorithm 1 illustrates the pseudocodes. We first do k-core decomposition using the linear algorithm [4], 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 Appendix D), 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.

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

Algorithm 2 Index construction: `advanced`

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

Complexity analysis. In Algorithm 2, lines 1-3 can be completed in $O(m)$ (We assume $m \geq n$). 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 $\alpha(n)$, the inverse Ackermann function, is less than 5 for all remotely practical values of n. The keyword inverted lists of all the tree nodes can be computed in $O(n \cdot \widehat{l})$. Therefore, 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 $O(n)$.

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

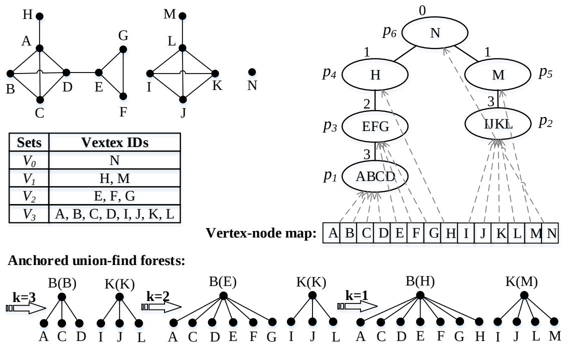


Fig. 5 An index built by `advanced` method.

array, we add p_3 into p_4's child List. When k=0, we generate p_6 and finish the index construction.

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. The decremental algorithm is 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) The core number of a subgraph G' of G, $core_G[G']$, 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 3. 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 Appendix C. 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).

Algorithm 3 Query algorithm: `Inc-S`

1: **function** QUERY(G, $root$, q, k, S)
2: find subtree root nodes $r_k, r_{k+1}, \cdots, r_{core_G[q]}$;
3: initialize l=0, Ψ using S;
4: **while** $true$ **do**
5: $l \leftarrow l+1$; $\Phi_l \leftarrow \emptyset$;
6: **for** each $<S', c> \in \Psi$ **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** $\Phi_l \neq \emptyset$ **then**
13: $\Psi \leftarrow$ GENECAND(Φ_l);
14: **for** each S' in Ψ **do**
15: **if** S' is generated from S_1 and S_2 **then**
16: $c \leftarrow max\{core_G[G_k[S_1]], core_G[G_k[S_2]]\}$;
17: Ψ.update(S', $<S', c>$);
18: **else** break;
19: output the communities of keyword sets in Φ_{l-1};

Algorithm 4 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};

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 3, 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.

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 4 presents the steps of `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).

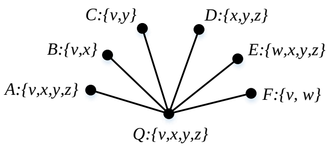
Example 5 Continue the graph and query (q=A, k=1, S=$\{w, x, y\}$) in Example 4. By `Inc-T`, we first find $G_1[\{x\}]$ and $G_1[\{y\}]$, whose vertex sets are $\{A, B, C, D\}$ and $\{A, C, D, E, F, G\}$. Then to find $G_1[\{x, y\}]$, we only need to search it from the subgraph, induced by the vertex 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 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 [14] and FP-Growth [15]) 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 [15].



(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

Fig. 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 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.

Algorithm 5 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;

Based on the above discussions, we design `Dec` as shown in Algorithm 5. We first generate candidate keyword sets using S and 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 Index Maintenance

In practice, the graphs are continuously evolving [29, 2]. Thus keywords and edges of graphs are often frequently updated. Clearly, when the graph is updated, both the CL-tree index and the ACQ query results also need to be updated. A straightforward method for handling the dynamic graph is to rebuild the index from scratch when an update is made. However, this method is very inefficient, especially when the updates are very frequent. To alleviate this issue, in this section we study how to dynamically maintain the CL-tree index efficiently, and propose algorithms for maintaining the CL-tree without rebuilding the CL-tree from scratch.

We first present how to handle keyword update in Section 7.1. Then, we discuss the maintenance of CL-tree for the insertion and deletion of an edge in Sections 7.2 and 7.3.

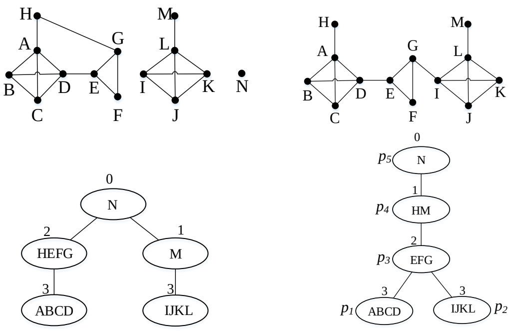
7.1 Keyword Update

The update for keyword update, i.e., inserting or deleting a keyword from a vertex's keyword set, is easy to be handled, since we can simply find the CL-tree node containing the vertex and update its $invertedList$. Recall that in the `advanced` method (Section 5.2.2), we have built a vertex-node map, where each vertex is mapped to a CL-tree node. Note that we can build such a map by traversing the tree if we use `basic`. To insert a new keyword for a vertex v, we can first locate the CL-tree node, p, containing v by the vertex-node map, and then insert the keyword and vertex ID into $p.invertedList$. To remove a keyword of a vertex, we can have a similar process on the CL-tree.

7.2 Edge Insertion

For the update of edge, i.e, inserting (deleting) an edge, it is not straightforward update the CL-tree accordingly. This is because, the insertion (deletion) of a single edge may trigger updates in several CL-tree nodes as well as their structures. We illustrate this by Example 7.

Example 7 Consider the graph in Figure 5. If we insert an edge (H, G) as shown in Figure 7(a), the core number of vertex H increases to 2 and we need to move it down to a node in the lower level. If we insert an edge (G, I), the connectivity of some vertices changes as shown in Figure 7(b) and thus the corresponding subtrees are merged as a new one.



(a) core number (b) connectivity

Fig. 7 The core number and connectivity change.

To maintain the CL-tree for inserting an edge, we propose an algorithm called `insertEdge`. The main idea is that, we first find vertices whose core numbers change, then change their positions in the CL-tree, and merge some subtrees. Let V^+ be the set of vertices whose core numbers increase after inserting an edge (u, v). We summarize the main steps of `insertEdge` as follows.

- **Step 1:** Compute V^+;
- **Step 2:** Move down vertices of V^+;
- **Step 3:** Merge subtrees.

We now elaborate these steps one by one.

Step 1: Compute V^+. Inserting an edge only affects the core numbers of a small number of vertices [23, 32]. We first give a definition, a theorem and a lemma proposed in this paper.

Definition 5 ([23]) Given a graph G and a vertex v, the induced core subgraph of v, denoted as G_v, is a connected subgraph containing v and the core numbers of all vertices in G_v equal to $core_G[v]$.

Notice that, the sets of vertices in G_u (G_v) are actually subsets of vertices in $p_u.vertexSet$ ($p_v.vertexSet$), where p_u, p_v denote the nodes that contain u, v.

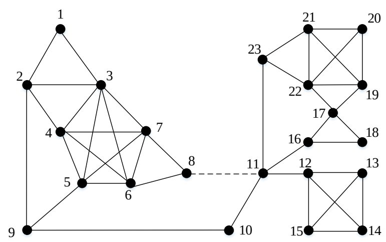
Theorem 1 (k-core update [23]) *Given a graph G and two vertices u and v. After inserting or deleting an edge (u,v) in G, we have that,*

- *If $core_G[u] > core_G[v]$, only the core numbers of vertices in G_v may need to be updated.*
- *If $core_G[u] < core_G[v]$, only the core numbers of vertices in G_u may need to be updated.*
- *If $core_G[u] = core_G[v]$, only the core numbers of vertices in the union of G_u and G_v, i.e., $G_{u \cup v}$ may need to be updated.*

Lemma 5 ([23]) *After inserting (deleting) an edge, the core number of any vertex in G increases (decreases) by at most 1.*

By above theorem and lemma, we can conclude that only a small number of vertices need to change their core numbers. In specific, we can first find node p_u (p_v) and then compute the vertex set V^+ in which vertices's core numbers increase by 1 using the algorithm in [23].

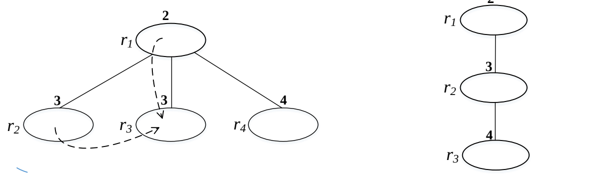
Step 2: Move down vertices of V^+. Let p be the node containing V^+ and c=min$\{core_G[u], core_G[v]\}$). Since the core numbers of vertices in V^+ increase by 1 (from c to c+1), we need move them down to nodes in the lower level. During the moving down process, we may also need to reorganize p's child nodes. Let us illustrate this by Example 8.



(a) the original graph

Tree Nodes	Vertices
r_1	1,2,8,9,10,11,16,17,18,23
r_2	12,13,14,15
r_3	19,20,21,22
r_4	3,4,5,6,7

Tree Nodes	Vertices
r_1	1,2,9,10,16,17,18
r_2	8,11,12,13,14,15,19,20,21,22,23
r_3	3,4,5,6,7



(b) before the edge insertion (c) after the edge insertion

Fig. 8 An example of the tree index update.

Example 8 Consider a graph in Figure 8(a) and its CL-tree in Figure 8(b). Let us insert an new edge (8, 11). We first get V^+=$\{8, 11, 23\}$ and c=2. Next, we move them down from r_1 to r_3. Besides, we have to merge r_2 into r_3 and place r_4 as r_3's child node, since their connectivity changes after the insertion. The updated CL-tree is depicted in Figure 8(c).

Clearly, moving down vertices of V^+ from p to p's child node (denoted by p') may change the connectivity of p's child nodes. Consider a specific vertex $a \in V^+$ and we initialize two empty sets B_1 and B_2. For each of a's neighbor

b whose $core_G[b]>c$, we first find the node p_b containing b, and then trace it up from p_b along the CL-tree until a child node of p, denoted by o_b. If o_b has a core number of c+1, we put it into B_1; Otherwise, we put it into B_2. Then, after moving down vertices of V^+, nodes in B_1 should be merged into p' and nodes in B_2 will be child nodes of p'.

Algorithm 6 move down vertices: `moveDown`

1: **function** MOVEDOWN(V^+, p)
2: **if** V^+=$\emptyset$ **then return** p;
3: $P \leftarrow \emptyset$;
4: update p using V^+;
5: **for** each $a \in V^+$ **do**
6: **for** each $b \in a$'s neighbor vertices **do**
7: **if** $core_G[b] > c$ and $b \notin V^+$ **then**
8: locate node p_b;
9: run TRACE(p_b), and update P;
10: $p_{max} \leftarrow$ a node of P, which has a core number of c+1 and its $vertexSet$ is the largest among all nodes of P;
11: **if** p_{max} = null **then**
12: create a new node p';
13: update p';
14: add P to $childList$ of p';
15: **else**
16: add V^+ to $p_{max}.vertexSet$;
17: **for** each $p_i \in P$ **do**
18: **if** $p_i.coreNum = c + 1$ **then**
19: merge p_i to p_{max};
20: **else**
21: add p_i to $childList$ of p_{max};
22: $p' \leftarrow p_{max}$;
23: update vertex-node map;
24: **if** $p.vertexSet = \emptyset$ **then**
25: add $\{p.childList - P\}$ to $childList$ of $p.father$;
26: **return** p';

Algorithm 6 presents `moveDown`. If $V^+ \neq \emptyset$, we first initialize a node set P (line 3). Then, we remove V^+ from $p.vertexSet$ and update $p.invertedList$ (line 4). $\forall a \in V^+$, we enumerate a's neighbor b whose $core_G[b]>c$, locate p_b, trace up from p_b to find p_b's ancestor node o_b which is a child node of p, and put o_b into P (lines 5-9). Let the node which has the largest size with core number being c+1 in P be p_{max} (line 10). Next, if p_{max}=$null$, we need to create a new child node of p (lines 11-14); otherwise, we merge and reorganize p's child nodes (lines 15-22). Finally we return node p' (line 26), which will be used later.

Step 3: Merge subtrees. Recall in Figure 7(b), after inserting (G, I), the corresponding subtrees, which correspond to the k-$\widehat{core}$s containing G and I are merged into one subtree. The process of merging subtrees starts from the tree nodes which contain G and I, and ends at their common ancestor node. The merging process is guaranteed by Lemmas 6 and 7.

Lemma 6 *After inserting an edge between two vertices, the maximum numbers of disconnected k-$\widehat{core}$s which need to be merged is two.*

Proof. Suppose that there are at least three disconnected k-$\widehat{core}$s that need to be merged after inserting an edge. We first randomly select three disconnected k-$\widehat{core}$s. Then we can prove by contradiction that one of them is already connected to one of the other two k-$\widehat{core}$s, which means there does not exist the third disconnected k-$\widehat{core}$ that needs to be merged. That completes the proof. □

Lemma 7 *In the process of merging subtrees, the maximum number of nodes which need to be merged in each level is two.*

Proof. It can be proved similarly by contradiction. □

By Lemmas 6 and 7, we conclude that, to merge the subtrees, we can first trace two paths starting from p_u and p_v until their common ancestor in the CL-tree, and then merge the pairs of nodes on the paths, if their core numbers are the same.

Algorithm 7 presents the overall steps of `insertEdge`. Following Theorem 1, we first compute V^+, and invoke `moveDown` to update these nodes in CL-tree (lines 2-16). Next, if p'_u and p'_v belong to two disconnected k-$\widehat{core}$s, we need to merge the subtrees (lines 17-19). In detail, we first trace two paths starting from p'_u and p'_v up until one common ancestor. Then, for each pair of nodes on the paths, if their core numbers are equal, we merge them as a single node. Finally, the tree index is updated. Note that during the above process, the elements of nodes and vertex-node map are also updated.

Algorithm 7 index update algorithm: `insertEdge`

1: **function** INSERTEDGE(p_u,p_v)
2: **if** $p_u.coreNum$=$p_v.coreNum$ **then**
3: compute V_1^+ in $p_u.vertexSet$;
4: $p'_u \leftarrow$ MOVEDOWN(V_1^+,p_u);
5: $p'_v \leftarrow p_v$;
6: **if** $p_u \neq p_v$ **then**
7: compute V_2^+ in $p_v.vertexSet$;
8: $p'_v \leftarrow$ MOVEDOWN(V_2^+,p_v);
9: **else if** $p_u.coreNum < p_v.coreNum$ **then**
10: compute V^+ in $p_u.vertexSet$;
11: $p'_u \leftarrow$ MOVEDOWN(V^+,p_u);
12: $p'_v \leftarrow p_v$;
13: **else**
14: compute V^+ in $p_v.vertexSet$;
15: $p'_v \leftarrow$ MOVEDOWN(V^+,p_v);
16: $p'_u \leftarrow p_u$;
17: **if** p'_u and p'_v are in two disconnected k-$\widehat{core}$s **then**
18: trace two paths starting from p'_u and p'_v up until a common ancestor;
19: merge pairs of nodes with the same core number on the paths;

7.3 Edge Deletion

Similar to the edge insertion, deleting an edge may trigger the updates of CL-tree nodes as well as their structures. We illustrate this by Example 9.

Example 9 Consider the graph in Figure 7. If we delete an edge (H, G) of the graph in Figure 5, the core number of vertex H decreases to 1. Thus we need to create a new node with core number being 1 and then move H up to the new node. If we delete an edge (G, I), the connectivity of some vertices changes as shown in Figure 5 and thus the corresponding subtree has to be split to two new ones.

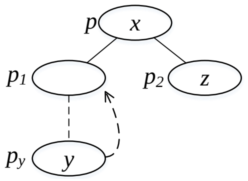
To maintain the CL-tree for deleting an edge, we propose an algorithm called `deleteEdge`. Let V^- be the set of vertices whose core numbers decrease after deleting an edge (u, v). We summarize the main steps of `deleteEdge` as follows.

- **Step 1:** Compute V^-;
- **Step 2:** Split nodes in a path;
- **Step 3:** Move up vertices of V^-.

We now elaborate these steps one by one.

Step 1: Compute V^-. By Lemma 5, the core numbers of vertices in G decrease by at most 1 after deleting an edge. We compute V^- using the algorithm in [23].

Step 2: Split nodes in a path. Similar to edge insertion, the connectivity of vertices may change after deleting an edge. Let p be the node containing vertex u if $core_G[u] \leq core_G[v]$; or the node containing v if $core_G[v] < core_G[u]$. From Example 9, we conclude that after deleting an edge (u, v), we may have to split p and its ancestor nodes. To enable efficient splitting, we first build a *vertex-tree* map for p. In this map, the key is a vertex v_{key}, which is a neighbor of a vertex in p and is in a descendant node of p; the value of v_{key} is a child node of p, whose subtree contains v_{key}. The vertex-tree map can be built simply by traversing the subtree. We illustrate the vertex-tree map via Example 10.



Vertices	Subtree roots
y	p_1
z	p_2

Fig. 9 Illustrating the vertex-tree map.

Example 10 Figure 9 depicts a subtree rooted at p. Suppose p only has one vertex x which has two neighbors y and z. Vertices y and z are in the descendant nodes of p. Then, in the vertex-tree map, there are two keys y and z, and their values are p_1 and p_2 respectively.

Next, we regroup vertices of p using the vertex-tree map. Specifically, we consider each vertex $a \in p.vertexSet$. For each neighbor b of a, if it has $core_G[b] > core_G[a]$, we locate the chid node of p which contains b using the vertex-tree map. As a result, each vertex corresponds to a list of child nodes of p. Then, we partition vertices of p into groups such that:

- For each vertex a in a group g, there is at least another vertex in g that is a neighbor of a or has the same corresponding child node.
- If there are two groups g_1 and g_2, then their corresponding child nodes should be completely different.

Essentially, each group corresponds to a k-$\widehat{core}$. Since deleting an edge in a k-$\widehat{core}$ can result in at most two k-$\widehat{core}$, vertices in p can be partitioned into at most two groups. After regrouping, we can split the nodes in the path from p to all its ancestor nodes as follows:

- If there is only one group, we do not split p; otherwise we split p into two nodes, each of which contains a group of vertices and links to a set of child nodes that its vertices correspond to.
- If p remains unsplit, and each child node of p is still linked to p, we stop; otherwise, we perform these two steps for p's father node.

Clearly, the splitting process is recursively performed on p's ancestor nodes and thus we split nodes in a path. We give Example 11 to illustrate the process.

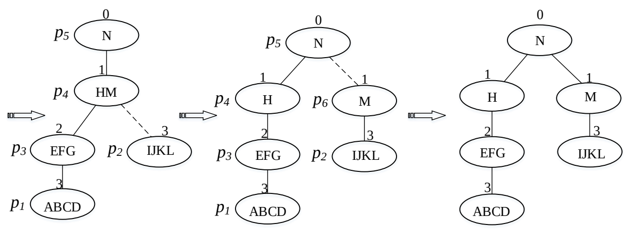


Fig. 10 The process of splitting nodes in a path.

Example 11 Consider the graph in Figure 7(b). We show the splitting process for deleting (G, I) in Figure 10. We first locate node p_3 containing G, regroup vertices of p_3 and find that p_2 cannot be linked as a child node p_3. So we link p_2 to p_3's father node p_4 and perform splitting on p_4. After regrouping vertices in p_4, we split it to two nodes because p_2 and p_3 are respectively shared by vertices H and M. Now since each child node of p_5 is still linked to it, we stop the splitting process.

Step 3: Move up vertices of V^-. After computing V^- and splitting some nodes in CL-tree, we move up vertices of V^- to an upper level. We denote the algorithm of performing moving up by `moveUp`.

We outline `moveUp` in Algorithm 8. We firs update p by removing V^- from its vertex set and updating its inverted list (line 3). Then, if the core number of $p.father$ is c–1, we add V^- to it; otherwise, we create a new node and add it to the tree (lines 4-9). Next, we collect the remaining vertices of p in set, regroup them and split nodes and update the vertex-node map (lines 10-12). We also need to update $childList$ and $invertedList$ of $p.father$ (lines 14-15). If there exists child nodes that are unconsidered in line 11, we re-link them to $p.father$ because these nodes are connected to vertices of V^- (line 16).

Algorithm 8 move up vertices: `moveUp`

1: **function** MOVEUP(V^-, p)
2:     **if** V^-=∅ **then return** ;
3:     update p;
4:     **if** $(p.father).coreNum$ = c–1 **then**
5:         add V^- to $p.father$;
6:     **else**
7:         create a new node $newFather$;
8:         add V^- to $newFather$;
9:         add $newFather$ to the tree;
10:     $set \leftarrow p.vertexSet$;
11:     $P \leftarrow$ regroup vertices of set and split node;
12:     update vertex-node map;
13:     link each $p_i \in P$ to $p.father$;
14:     update $invertedList$ of $p.father$;
15:     $P' \leftarrow$ child nodes unconsidered in above step;
16:     **if** $P' \neq \emptyset$ **then** re-link each $p \in P'$ to $p.father$;

We present the overall steps of `deleteEdge` in Algorithm 9. Similar to edge insertion, we have three cases to be handled separately. In these three cases, we first compute V^- (lines 3,7,11). Then we split the nodes and get the updated node(s) p'_v or p'_u or both p'_u and p'_v (lines 4,8,12). Next we apply `moveUp` to move up vertices of V^- (lines 5,9,13). In $p_u.coreNum$ = $p_v.coreNum$ case, if vertices of V^- belong to two disconnected $k-\widehat{core}$s, we should separate V^- to two sets and invoke `moveUp` accordingly (lines 14-16).

Algorithm 9 index algorithm: `deleteEdge`

1: **function** DELETEEDGE(p_u, p_v)
2:     **if** $p_u.coreNum > p_v.coreNum$ **then**
3:         compute V^- in $p_v.vertexSet$;
4:         $p'_v \leftarrow$ split p and its ancestor nodes;
5:         MOVEUP(V^-, p'_v);
6:     **else if** $p_u.coreNum < p_v.coreNum$ **then**
7:         compute V^- in $p_u.vertexSet$;
8:         $p'_u \leftarrow$ split p and its ancestor nodes;
9:         MOVEUP(V^-, p'_u);
10:     **else**
11:         compute V^- in $p_u.vertexSet$;
12:         $p'_u, p'_v \leftarrow$ split p and its ancestor nodes;
13:         **if** $p'_u = p'_v$ **then** MOVEUP(V^-, p'_u);
14:         **else**
15:             $V_u{}^-, V_v{}^- \leftarrow$ separate V^-;
16:             MOVEUP($V_u{}^-$,p'_u); MOVEUP($V_v{}^-$,p'_v);

8 The ACQ-A and ACQ-M Problems

In this section, we introduce two problems related to the ACQ problem, namely *Approximate ACQ problem* (or ACQ-A) and *Multiple-vertex ACQ problem* (or ACQ-M). We also develop the query algorithms based on the CL-tree.

8.1 The ACQ-A Problem

We first present an approximation version of the ACQ query, denoted by Problem 2. In Problem 2, vertices of an AC do not need to exactly share the same keywords in S; instead, they just need to share a predefined percentage of keywords in S. Thus, the keyword cohesiveness is relaxed. This could be useful for graphs if the keyword information of vertices is weak.

Problem 2 (ACQ-A) 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:

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

We illustrate problem 2 using Example 12.

Example 12 In Figure 3(a), let q=A and k=2. If S=$\{x, y\}$, θ=50%, ACQ-A will return the subgraph induced by the vertex set $\{A, B, C, D, E\}$ as the target AC.

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 detailed algorithms are presented in Appendix E.

We also propose an efficient query algorithm `SWT`, based on the CL-tree index. Algorithm 10 presents `SWT`. 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 at least $|S| \times \theta$ keywords by applying `keyword-checking`. Next, we find $G[S]$ from the subgraph induced by vertices in V' (line 3), and find $G_k[S]$(line 4). Finally, we output $G_k[S]$ as the target AC, if it exists (line 5).

8.2 The ACQ-M Problem

The ACQ-M problem generalizes the ACQ problem for supporting a set Q of vertices, and it finds the ACs containing all the vertices in Q. We give its definition as follows.

Algorithm 10 Query algorithm: `SWT`

function QUERY(G, $root$, q, k, S)
  find the node r_k from the CL-tree index;
  traverse the subtree rooted at r_k and collect a set V' of vertices containing at least $|S| \times \theta$ keywords by intersecting the inverted lists;
  find $G[S]$ from the subgraph induced by V';
  find $G_k[S]$ from $G[S]$;
  output $G_k[S]$ as the target AC.

Problem 3 Given a graph G, a positive integer k, a vertex set $Q \subseteq V$, and a predefined keyword set S, return a subgraph G_Q, the following properties hold:

• **Connectivity**. $G_Q \subseteq G$ is connected and G_Q contains all the vertices of 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 illustrate Problem 3 via Example 13.

Example 13 In Figure 3(a), let Q=$\{A, C\}$ and k=2. If S=$\{w, x, y, z\}$, then ACQ-M returns the subgraph induced by the vertex set $\{A, C, D\}$ as the target AC, whose shared keyword set is$\{x, y\}$.

To answer the query in Problem 3, we can first find a set S' of intersected keywords, which are contained by S and every vertex in Q. Then, we randomly take a vertex $q \in Q$ as the query vertex. Finally, we can find the target ACs by any of previous ACQ algorithms. Following the above idea, we extend `basic-g` and `basic-w` and obtain two basic algorithms, i.e., `basic-g-v2` and `basic-w-v2`. We also extend `Dec` and get an index based algorithm `MDec`. Algorithm 11 presents `MDec`. Note that we do not extend Inc-S and Inc-T, as they are generally slower than `Dec`, which has been demonstrated by the earlier version of this paper [11].

Algorithm 11 Query algorithm: `MDec`

function QUERY(G, $root$, Q, k, S)
  $S' = (\bigcap_{i=0}^{|Q|-1} W(q_i)) \cap S$;
  $q \leftarrow$ randomly select a vertex from Q;
  run `Dec` with q, k, and S';
  output target ACs which contain Q;

9 Experiments

We now present the experimental results. Section 9.1 discusses the setup. We discuss the effectiveness and efficiency results in Sections 9.2 and 9.3 respectively.

9.1 Setup

We consider six real datasets. The first four datasets (Flickr, DBLP, Tencent, and DBpedia) are static graphs. For *Flickr* [4] [35], 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 numbers of vertices and edges, k_{max} value, a vertex's average degree $\widehat{d}$, and its keyword set size $\widehat{l}$.

The remaining two dynamic datasets, i.e., *DFlickr* and *Youtube* [25,26], are dynamic evolving graphs, which contain the snapshots of graphs as the time goes on. Note that these two datasets do not have keywords. Both DFlickr and Youtube datasets are about the user friendship networks on Flickr and Youtube websites respectively. Each vertex denotes a user and each edge denotes a friendship between two users. DFlickr contains edges which are inserted and deleted during the evolving process; while in Youtube, there are only inserted edges as the time goes on. In Table 3, the initial numbers of vertices and edges in the first day of each dataset are reported. In the next 100 days, for *DFlickr*, 10,301,741 edges were inserted and 2,211,272 edges were deleted; for *Youtube*, 13,954,071 edges were inserted.

Table 3 Datasets used in our experiments.

Dataset	Vertices	Edges	k_{max}	$\widehat{d}$	$\widehat{l}$
Flickr	581,099	4,972,274	152	17.1	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	17.7	15.0
DFlickr	2,585,569	22,838,277	600	17.6	—
Youtube	1,881,147	4,571,023	55	4.9	—

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

[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`

select 300 query vertices with core numbers of 6 or more, which ensures that there is a k-core containing each query vertex. In all the following figures about efficiency, we report the average time cost of these queries.

To evaluate the index maintenance algorithms, we consider all the six datasets. For the first four datasets, we randomly select 1,000 vertices and for each of them, we randomly insert and delete one keyword to evaluate the perform keyword update. Meanwhile, we randomly insert and delete five groups of edges, each of which has 100 edges, and their core numbers vary from 5 to 25. For each of the remaining datasets (DFlickr and Youtube), we first take the snapshots in 100 consecutive days, then divide them into five groups, each of which are in a period of 20 consecutive days, and finally we randomly select 200 records from each group as test edges.

We implement all the algorithms in Java, and run experiments on a machine having a quad-core Intel i7-3770 3.40GHz processor, and 32GB of memory, with Ubuntu installed. We present the effectiveness and efficiency results in Sections 9.2 and 9.3.

9.2 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.

9.2.1 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 $C(q)$:

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

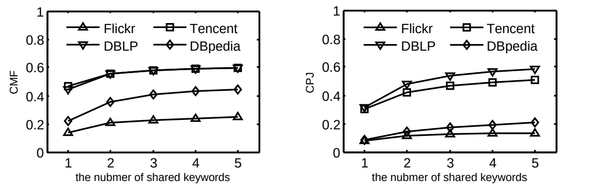
Notice that $CMF(C(q))$ ranges from 0 to 1. The higher its value, the more cohesive is a community.

• **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 $C(q)$:

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

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 11. 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.

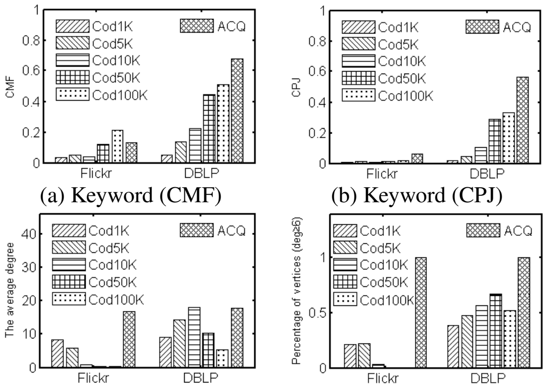


(a) CMF (b) CPJ

Fig. 11 AC-label length.

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` [30] 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 [27,40]; 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 [30]. We first run these algorithms offline to obtain all the communities. Given a query vertex q, they return communities containing q as the results.

We consider both keyword and structure for evaluating community quality. (1) *Keyword:* Figures 12(a) and (b) show



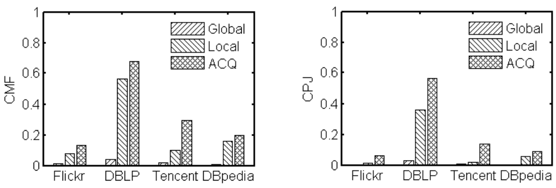
(a) Keyword (CMF) (b) Keyword (CPJ)

(c) Structure (Avg. degree) (d) Structure (degree ≥ 6)

Fig. 12 Comparing with community detection method.

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 12(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 [34] and Local [6]. Both of them use the metric minimum degree, we thus focus on the keyword cohesiveness. Figure 13 shows the CMF and CPJ values on 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.



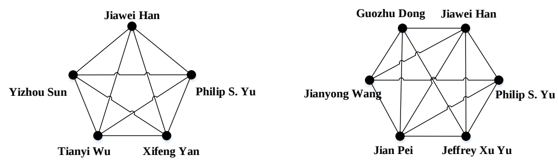
(a) CMF (b) CPJ

Fig. 13 Comparing with community search methods.

9.2.2 A Case Study

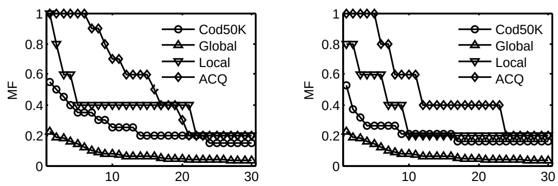
We next perform a case study on the DBLP dataset, in which we consider two renowned researchers in database and data mining: Jim Gray and Jiawei Han. We use $k = 4$ here. We use Cod50K to represent CODICIL for further analysis. We mainly consider the input query keyword set S, keywords and sizes of communities.

1. Effect of S. Figure 14 shows two ACs of Jiawei (AC-labels are shown in the captions), where the query keyword 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 14(a)) and pattern mining (Figure 14(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 keyword set S, and we thus do not show the results.



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

Fig. 14 Jiawei Han's ACs.



(a) Jim Gray (b) Jiawei Han

Fig. 15 Frequency distribution of keywords.

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

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 15. 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 Jim and Jiawei's communities in Tables 5 and 6. We can see that, words "sloan", "digital", "sky", "survey", and "sdss" reflect that the community is likely about the SDSS project led by Jim. The top-6 keywords of Jiawei's AC are related to heterogenous networks. In contrast, the keywords of Global and Local tend to be less related to the query keyword set, and thus they cannot be used to characterize the communities specifically related to Jiawei. Note that the top-6 keywords of Global are the same for both Jim and Jiawei, as they are in the same k-$\widehat{core}$. The overall results show that, ACQ performs better than other methods.

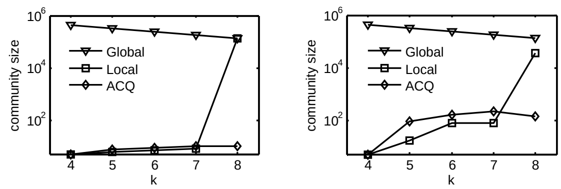
Table 5 Top-6 keywords (Jim Gray).

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, heterog

3. Effect of k on community size. We vary the value of k and report the average size of communities in Figure 16. 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.



(a) Jim Gray (b) Jiawei Han

Fig. 16 Community size.

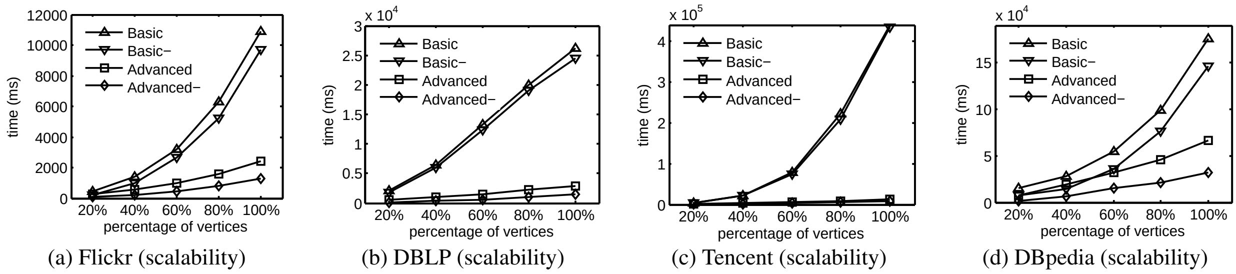
9.3 Results on Efficiency

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 17(a)-17(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 mins, 2 mins, 1 day, and 3+ days (we stop it after runing 3 days) to cluster the vertices of Flickr, DBLP, Tencent and DBpedia offline respectively.

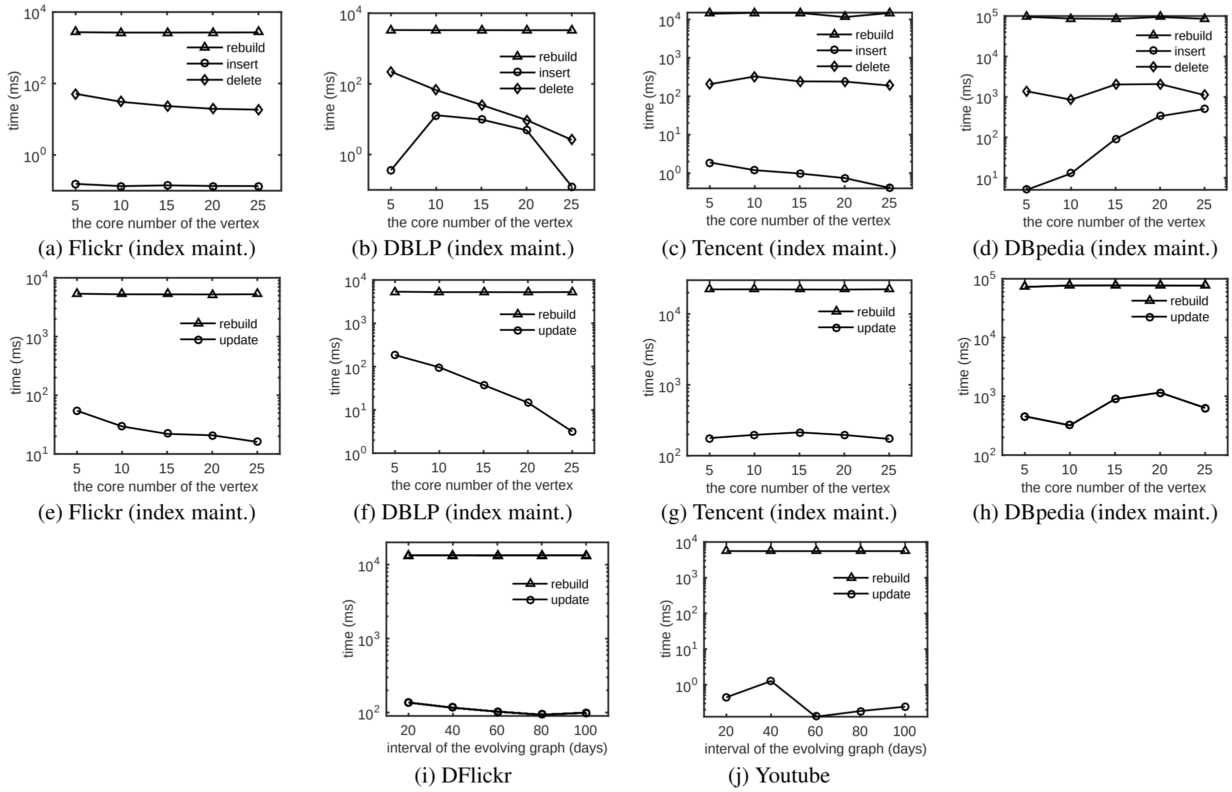
2. Index Maintenance. We first evaluate the performance of keyword update and the results show that the keyword update is very fast. For example, we compute the average time of 1,000 keyword insertion and deletion on Flickr dataset, and the results show that they are over 10^6 and 10^5 times faster than rebuilding the index respectively. Similar results are obtained on other three datasets.

Next, we show the performance of edge update on four static datasets in Figures 18(a)-18(h) by varying k. In Figures 18(a)-18(b), we report the efficiency by separately performing edge insertion and deletion. Clearly, insertEdge is 10^2 to 10^5 times faster than rebuilding the index, and deleteEdge is also around 10^2 times faster than rebuilding index. The main reason is that, inserting or deleting one edge only affects a small proportion of CL-tree nodes and their connectivity. In other words, most of the nodes remain unaffected. Moreover, the algorithm deleteEdge is slower than insertEdge. This is because, splitting tree nodes generally involves more computational cost than merging



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

Fig. 17 Efficiency results of index construction.



(a) Flickr (index maint.) (b) DBLP (index maint.) (c) Tencent (index maint.) (d) DBpedia (index maint.)

(e) Flickr (index maint.) (f) DBLP (index maint.) (g) Tencent (index maint.) (h) DBpedia (index maint.)

(i) DFlickr (j) Youtube

Fig. 18 Efficiency results of index maintenance.

tree nodes. In addition, we put all the insertion and deletion edges together, and report the efficiency by performing insertion and deletion for these edge with a random order. We report the results in Figures 18(c)-18(d), where "update" denotes our algorithms including both `insertEdge` and `deleteEdge`. We can see that, the index update algorithm is still much faster than rebuilding the index.

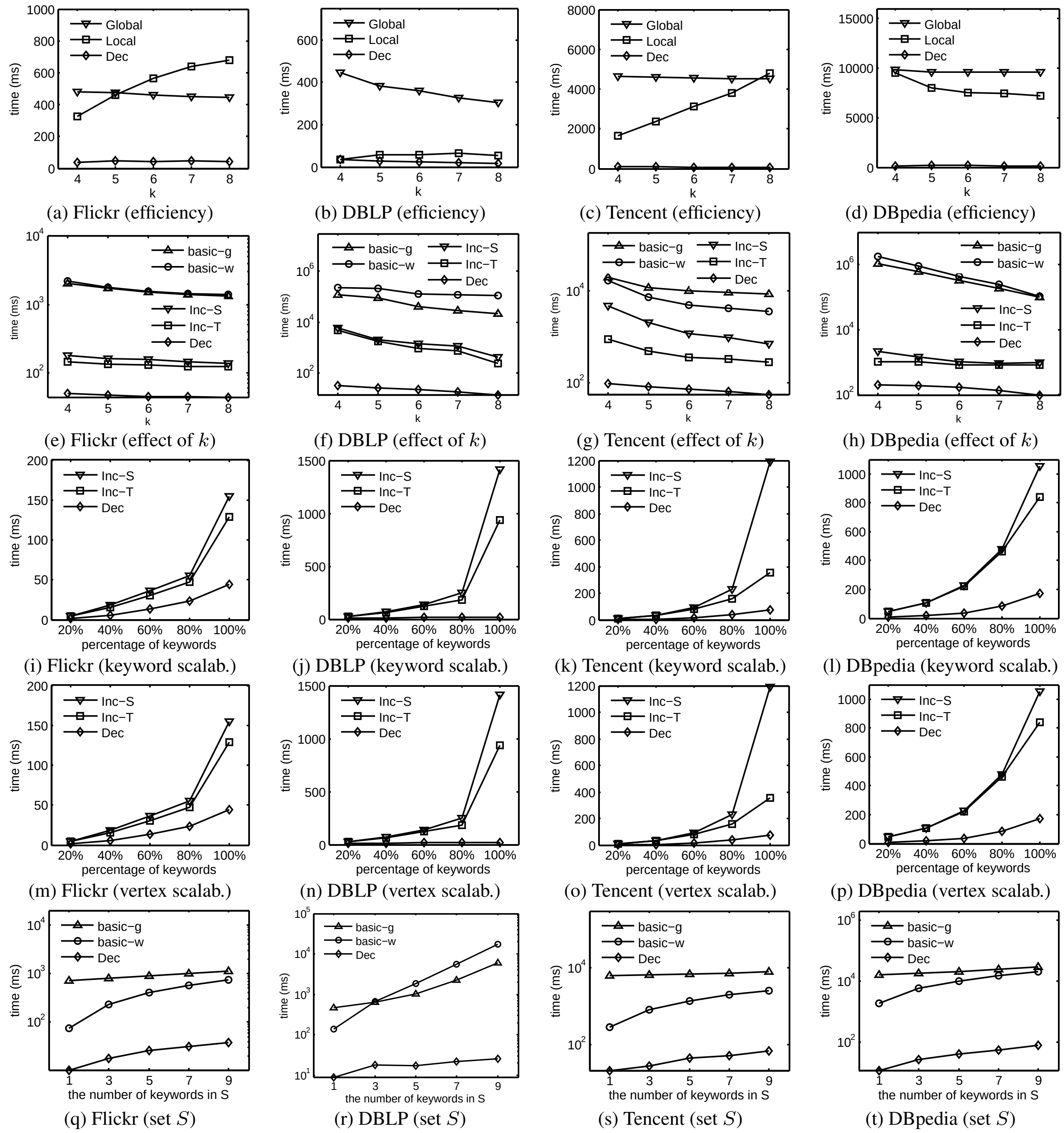
The results on real dynamic graphs (DFlickr and Youtube datasets) are shown in Figures 18(i)-18(j). It is obvious to observe that, the results on real dynamic graphs are similar to those on static graphs, and our proposed algorithms are at least two orders of magnitude faster than rebuilding the CL-tree from scratch. In summary, our proposed algorithms are efficient for maintaining the index for dynamic graphs.

3. Efficiency of CS methods. Figures 19(a)-19(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.

4. Effect of k. Figures 19(e)-19(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.

5. ACQ scalability w.r.t. keyword. Figures 19(i)-19(l) examine 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.

6. ACQ scalability w.r.t. vertex. Figures 19(m)-19(p) report the scalability over different fraction of vertices. Al-



(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 scalab.) (j) DBLP (keyword scalab.) (k) Tencent (keyword scalab.) (l) DBpedia (keyword scalab.)

(m) Flickr (vertex scalab.) (n) DBLP (vertex scalab.) (o) Tencent (vertex scalab.) (p) DBpedia (vertex scalab.)

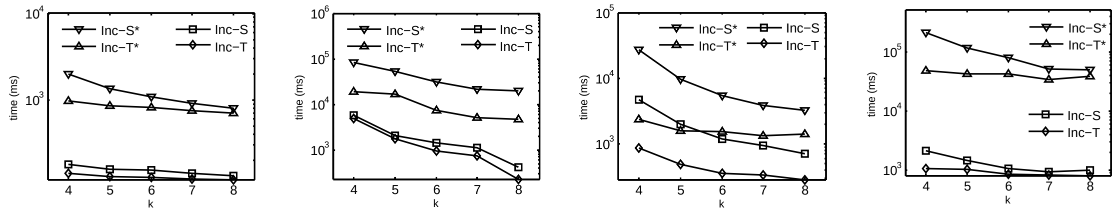
(q) Flickr (set S) (r) DBLP (set S) (s) Tencent (set S) (t) DBpedia (set S)

Fig. 19 Efficiency results of community search.

l the keywords of each vertex are considered. Again, `Dec` scales the best.

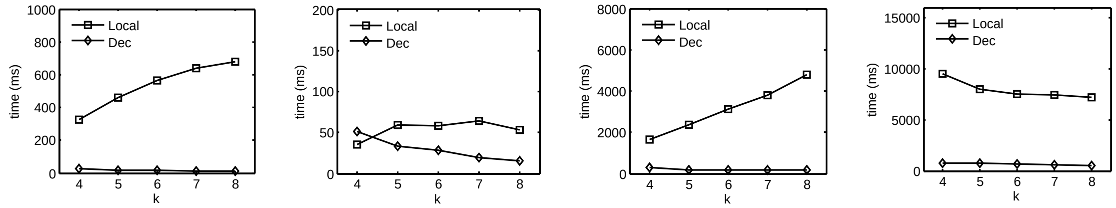
7. Effect of size of S. For each query vertex, we randomly select 1, 3, 5, 7 and 9 keywords to form the query keyword set S. As `Dec` performs better than `Inc-S` and `Inc-T`, we mainly compare `Dec` with the baseline solutions. Figures 19(q)-19(t) show that the cost of all algorithms increase with the $|S|$. Also, `Dec` is 1 to 3 order-of-magnitude faster than `basic-g` and `basic-w`.

8. Effect of invertedList. To test the importance of invertedList, we have implemented `Inc-S`$\star$ and `Inc-T`$\star$, which are respective variants of `Inc-S` and `Inc-T`, but without the invertedList structure at each CL-tree node. Figure 20 shows the results. We see that `Inc-S` (`Inc-T`) is 1 to 2 order of magnitude faster than `Inc-S`$\star$ (`Inc-T`$\star$) on all the four datasets in our experiments. The reason is that the keyword-checking operation mentioned in the above example is frequently performed in the ACQ search process. Thus, the invertedList, which improves the performance of this operation, allows the ACQ search to be conducted more efficiently.



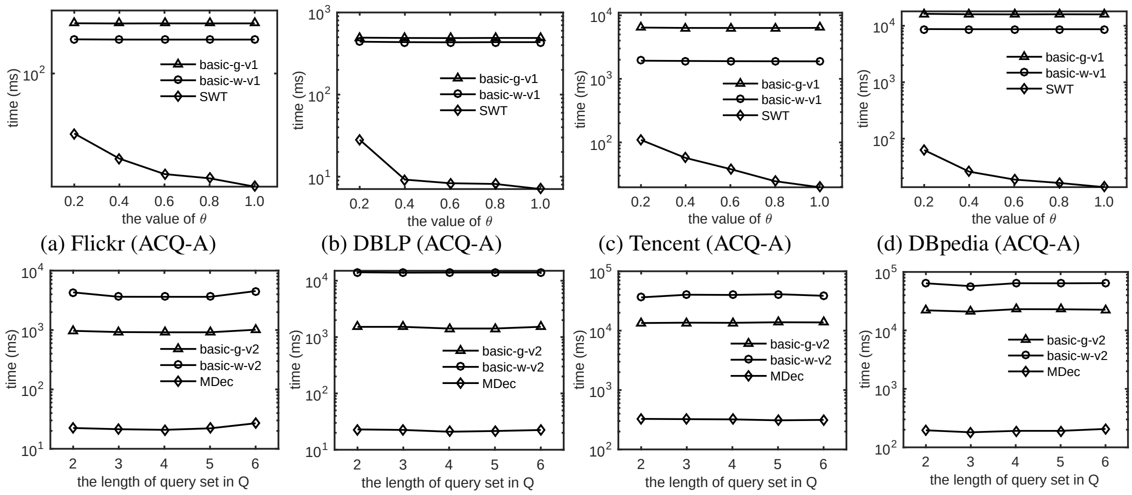
(a) Flickr (b) DBLP (c) Tencent (d) DBpedia

Fig. 20 Effect of InvertedList for `Inc-S` and `Inc-T`.



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

Fig. 21 Results on non-attributed graphs.



(a) Flickr (ACQ-A) (b) DBLP (ACQ-A) (c) Tencent (ACQ-A) (d) DBpedia (ACQ-A)

(e) Flickr (ACQ-M) (f) DBLP (ACQ-M) (g) Tencent (ACQ-M) (h) DBpedia (ACQ-M)

Fig. 22 Efficiency results of ACQ-A and ACQ-M.

9. Non-attributed graphs. We have tested `Dec` and `Local` on non-attributed graphs. This is done by running these algorithms on our datasets, without using any of their associated keyword sets. As shown in Figure 21, 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 this dataset, 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.

10. Effect of θ in ACQ-A. 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 ACQ-A using `basic-g-v1`, `basic-w-v1` and `SWT`. Figures 22(a)-22(d) show their efficiency results. We can see that `SWT` outperforms the basic solutions consistently, since it is based on the CL-tree index.

11. Effect of $|Q|$ in ACQ-M. We randomly select five groups of query sets by varying the size of Q from 2 to 6. Each group has 200 query sets. We run `basic-g-v2`, `basic-w-v2` and `MDec` with these five groups of query sets, and report efficiency in Figures 22(e)-22(h). We can observe that, similar to the results of single query vertices, `MDec` is at least two orders of magnitude faster than the baseline solutions which do not use the CL-tree index.

10 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. We further propose index maintenance algorithms for dynamic graphs. Moreover, we formally define ACQ-A and ACQ-M problems and propose efficient query algorithms based on the CL-tree index. Our experimental results on several dataseats show that ACs are easier to interpret than those of existing community detection/search methods, and they can be "personalized". In addition, our solutions are also faster than existing community search algorithms.

In the future, 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. We will also investigate how the directions of edges will affect the formation of an AC. We will examine how graph pattern matching techniques [36, 9, 10] can be extended to find ACs effectively from large attributed graphs. An interesting research direction is to study how to automatically generate a meaningful graph pattern that well reflects a real community, and how to use these patterns to find ACs efficiently.

References

1. https://en.wikipedia.org/wiki/Disjoint-set_data_structure.
2. B. Bahmani, R. Kumar, M. Mahdian, and E. Upfal. Pagerank on an evolving graph. In *KDD*, pages 24–32, 2012.
3. N. Barbieri, F. Bonchi, E. Galimberti, and F. Gullo. Efficient and effective community search. *Data Mining and Knowledge Discovery*, 29(5):1406–1433, 2015.
4. V. Batagelj and M. Zaversnik. An o(m) algorithm for cores decomposition of networks. *arXiv*, 2003.
5. W. Cui, Y. Xiao, H. Wang, Y. Lu, and W. Wang. Online search of overlapping communities. In *SIGMOD*, pages 277–288, 2013.
6. W. Cui, Y. Xiao, H. Wang, and W. Wang. Local search of communities in large graphs. In *SIGMOD*, pages 991–1002, 2014.
7. B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding top-k min-cost connected trees in databases. In *ICDE*, 2007.
8. S. N. Dorogovtsev et al. K-core organization of complex networks. *Physical review letters*, 2006.
9. W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph pattern matching: from intractable to polynomial time. *PVLDB*, 2010.
10. W. Fan, X. Wang, Y. Wu, and J. Xu. Association rules with graph patterns. *PVLDB*, 8(12):1502–1513, 2015.
11. Y. Fang, R. Cheng, S. Luo, and J. Hu. Effective community search for large attributed graphs. *PVLDB*, 9(12):1233–1244, 2016.
12. Y. Fang, H. Zhang, Y. Ye, and X. Li. Detecting hot topics from twitter: A multiview approach. *Journal of Information Science*, 40(5):578–593, 2014.
13. S. Fortunato. Community detection in graphs. *Physics Reports*, 486(3):75–174, 2010.
14. J. Han, M. Kamber, and J. Pei. *Data mining: concepts and techniques*. Elsevier, 2011.
15. J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD*, 2000.
16. H. He, H. Wang, J. Yang, and P. S. Yu. Blinks: ranked keyword searches on graphs. In *SIGMOD*, 2007.
17. J. Hu, X. Wu, R. Cheng, S. Luo, and Y. Fang. Querying minimal steiner maximum-connected subgraphs in large graphs. In *CIKM*, pages 1241–1250, 2016.
18. X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu. Querying k-truss community in large and dynamic graphs. In *SIGMOD*, 2014.
19. X. Huang, L. V. Lakshmanan, J. X. Yu, and H. Cheng. Approximate closest community search in networks. *Proceedings of the VLDB Endowment*, 9(4):276–287, 2015.
20. V. Kacholia et al. Bidirectional expansion for keyword search on graph databases. In *VLDB*, 2005.
21. M. Kargar and A. An. Keyword search in graphs: Finding r-cliques. *PVLDB*, 4(10):681–692, 2011.
22. R.-H. Li, L. Qin, J. X. Yu, and R. Mao. Influential community search in large networks. In *PVLDB*, 2015.
23. R.-H. Li, J. X. Yu, and R. Mao. Efficient core maintenance in large dynamic graphs. *TKDE*, 2014.
24. Y. Liu, A. Niculescu-Mizil, and W. Gryc. Topic-link lda: joint models of topic and author community. In *ICML*, 2009.
25. A. Mislove. *Online Social Networks: Measurement, Analysis, and Applications to Distributed Information Systems*. PhD thesis, Rice University, Department of Computer Science, May 2009.
26. A. Mislove, H. S. Koppula, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Growth of the flickr social network. In *Proceedings of the 1st ACM SIGCOMM Workshop on Social Networks (WOSN'08)*, August 2008.
27. R. M. Nallapati, A. Ahmed, E. P. Xing, and W. W. Cohen. Joint latent topic models for text and citations. In *KDD*, 2008.
28. M. Newman et al. Finding and evaluating community structure in networks. *Physical review E*, 2004.
29. C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng. On querying historical evolving graph sequences. *VLDB*, 4(11):726–737, 2011.
30. Y. Ruan, D. Fuhry, and S. Parthasarathy. Efficient community detection in large networks using content and links. In *WWW*, 2013.
31. M. Sachan et al. Using content and interactions for discovering communities in social networks. In *WWW*, 2012.
32. A. E. Sarıyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek. Incremental k-core decomposition: algorithms and evaluation. *The VLDB Journal*, 25(3):425–447, 2016.
33. S. B. Seidman. Network structure and minimum degree. *Social networks*, 5(3):269–287, 1983.
34. M. Sozio and A. Gionis. The community-search problem and how to plan a successful cocktail party. In *KDD*, 2010.
35. B. Thomee et al. The new data and new challenges in multimedia research. *arXiv:1503.01817*, 2015.
36. H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad. Fast best-effort pattern matching in large attributed graphs. In *KDD*, 2007.
37. Z. Xu, Y. Ke, Y. Wang, H. Cheng, and J. Cheng. A model-based approach to attributed graph clustering. In *SIGMOD*, 2012.
38. J. Yang, J. McAuley, and J. Leskovec. Community detection in networks with node attributes. In *ICDM*, pages 1151–1156, 2013.
39. J. Yang, J. McAuley, and J. Leskovec. Community detection in networks with node attributes. In *ICDM*, 2013.
40. T. Yang, R. Jin, Y. Chi, and S. Zhu. Combining link and content for community detection: a discriminative approach. In *KDD*, 2009.
41. J. X. Yu, L. Qin, and L. Chang. Keyword search in databases. *Synthesis Lectures on Data Management*, 2009.
42. Y. Zhou, H. Cheng, and J. X. Yu. Graph clustering based on structural/attribute similarities. *VLDB*, 2009.

A Proofs of Lemmas

Lemma 4 (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 5 *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 6 *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 7 *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 Basic Solutions for ACQ

Algorithms 12 and 13 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 12 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 13 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 13 Basic solution: `basic-w`

```
function QUERY(G, q, k, S)
    init Ψ using S;
    while true do
        Φ ← ∅;
        for each S' ∈ Ψ do
            find G[S'] from G;
            find G_k[S'] from G[S'];
            if G_k[S'] exists then
                Φ.add(S');
        if Φ ≠ ∅ then
            Ψ ← GENECAND(Φ);
        else
            break;
    output the communities of keyword sets in Φ;
```

C Candidate Generation

Given a set Φ of qualified keyword sets, Algorithm 14 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 14 Generate candidate keyword sets

1: **function** GENECAND(Φ)
2: $\Psi \leftarrow \emptyset$;
3: **for** each $S_i \in \Phi$ **do**
4: **for** each $S_j \in \Phi$ **do**
5: **if** S_i and S_j differs at the last keyword **then**
6: initialize S'=$S_i \cup S_j$;
7: **if** S' cannot be pruned by Lemma 1 **then**
8: Ψ.add(S');
9: **return** Ψ;

D Anchored Union-find

Algorithm 15 presents the four functions of the anchored union-find (AUF) data structure.

Algorithm 15 Functions on the AUF data structure

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

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 initializes $x.anchor$ as x (line 2). 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 20). 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 all practical values of n. In function MAKESET, 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 Basic Algorithms for ACQ-A and ACQ-M

1. ACQ-A. We present `basic-g-v1` and `basic-w-v1` in Algorithms 16 and 17 respectively.

Algorithm 16 Query algorithm: `basic-g-v1`

function QUERY(G, q, k, S)
find the k-$\widehat{core}$, $\mathcal{C}_k$, containing q from G;
collect a set V' of vertices containing at least $|S| \times \theta$ keywords from $\mathcal{C}_k$;
find $G[S]$ from the subgraph induced by V';
find $G_k[S]$ from $G[S]$;
output $G_k[S]$ as the target AC.

Algorithm 17 Query algorithm: `basic-w-v1`

function QUERY(G, q, k, S)
collect a set V' of vertices containing at least $|S| \times \theta$ keywords from G;
find $G[S]$ from the subgraph induced by V';
find $G_k[S]$ from $G[S]$;
output $G_k[S]$ as the target AC.

2. ACQ-M. Algorithms `basic-g-v2`, `basic-w-v2` are extended from `basic-g` and `basic-w`. Detailed pseudocodes are attached in Algorithms 18 and 19 respectively.

Algorithm 18 Query algorithm: `basic-g-v2`

function QUERY(G, Q, k, S)
S' = $(\bigcap_{i=0}^{|Q|-1} W(q_i)) \cap S$;
$q \leftarrow$ randomly select a vertex from Q;
run `basic-g` with q, k, and S';
output target ACs which contain Q;

Algorithm 19 Query algorithm: `basic-w-v2`

function QUERY(G, Q, k, S)
S = $(\bigcap_{i=0}^{|Q|-1} W(q_i)) \cap S$;
$q \leftarrow$ randomly select a vertex from Q;
run `basic-w` with q, k, and S';
output target ACs which contain Q;