On Label-Aware Community Search

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

ABSTRACT
Recently, the retrieval of a community from large graph databases has captured a lot of attention. Given a vertex \( q \) of a graph, the community search operation finds a subgraph, or community, which contains vertices closely related to \( q \). Communities are prevalent in social networks, bibliographical graphs, and knowledge bases, and they enable emerging applications like product advertisement and setting up of social events. Existing community search algorithms, which are based mainly on the relationships (or edges) between vertices, overlook the important information carried by the attributes of vertices. We found that communities returned by these algorithms often consist of vertices having keywords about a variety of topics, and this makes it difficult to interpret the meanings behind those communities.

In this paper, we investigate how vertex attributes can be used to generate communities that are easier to analyze. Particularly, we propose the label-aware community (or LAC), which not just requires vertices to be structurally close to each other, but also needs to have common labels. The LAC allows a better understanding of how and why a community is formed (e.g., members of an LAC have a common interest in yoga, because they all have the same label “yoga”). To enable efficient LAC search, we develop a data structure called the CL-tree, and investigate novel query algorithms based on it. We evaluate our solutions on four large graphs, namely Flickr, DBLP, Tencent, and DBpedia. Our experimental results show that LACs can be extracted efficiently. Moreover, they reflect more precise meanings than communities found by existing algorithms.

Categories and Subject Descriptors
H.2.8 [Database Management]: Database Applications—Data mining; G.2.2 [Discrete Mathematics]: Graph Theory—Graph algorithms

1. INTRODUCTION

The retrieval of communities from large graph databases, including social networks, bibliographical graphs, and knowledge bases, has received plenty of research interest. Given a graph \( G \), the community detection problem is about the extraction of subgraph structures whose vertices are closely connected \([10, 8]\). Another related problem, called community search, focuses on retrieving subgraph(s) from \( G \) which have a close relationship with a given query vertex \( q \) \([24, 5, 4, 13]\). Community generation algorithms power a lot of important applications, including friend recommendation, advertising, tag suggestion and setting up of social events. Figure 1 shows a social network, in which an edge with two vertices \( u \) and \( v \) represents a relationship between \( u \) and \( v \). Given that \( q \) is the vertex Jack, and every vertex required for a community has a minimum degree of \( k = 2 \), a possible community \([24]\) of \( q \) is the one that contains six vertices, namely Jack, Mike, John, Bob, Alice, and Tom.

Existing solutions often make use of the graph edge information to produce communities. However, they overlook the rich information associated with vertices, which is commonly found in large graph databases. In Twitter, for instance, a vertex contains a large number of keywords, describing the hobbies, education and check-in places of the user corresponding to the vertex. In Figure 1, Jack has three keywords depicting his interest (e.g., “research”). These keywords, which give information about the vertices involved, can be important to obtaining and explaining a community. Unfortunately, this is often not considered by existing algorithms, which may output communities with vertices that are not semantically close to each other. Let us consider the community of Jack in Figure 1 (circled) again. We can see that the keywords associated with the vertices of this community span a wide range of topics (e.g., research, tour, yoga, and cook). But what is the main theme of this community? What are the members’ common interest? Why do these people constitute to the community of Jack? Notice that Bob and Mike are closely related to Jack, since the keywords...
“research” and “sports” appear in their vertices (e.g., these three persons belong to the same research team and often play basketball together). However, Jack appears to be remote from Alice and Tom, because Jack does not share any keywords with them (e.g., Tom and Jack are alumni but do not have a common keyword; Alice is included into Jack’s community simply because she is linked to Bob). A better candidate for Jack’s community could be the one that only contains Bob, Mike, and himself.

In this paper, we study how vertex keywords can be used to improve the search of communities. We propose the label-aware community (or LAC). Similar to the communities found by traditional community detection/search algorithms, the LAC is a subgraph of $G$ satisfies structure cohesiveness (i.e., the vertices involved in the LAC are closely linked to each other). The LAC also exhibits keyword cohesiveness (i.e., the vertices contained in the LAC have keywords in common). An example LAC for Figure 1 is one that contains Jack, Bob, and Mike, which form a connected subgraph with vertex degree of 2 or more, and have “research” and “sports” keywords in common. Essentially, an LAC consists of vertices with similar contexts or backgrounds, allowing a query user to focus on the common features of these vertices. LACs can be useful in setting up social events. For example, if a social network user has many keywords about traveling (e.g., he posted a lot of photos about his trips, with labels), an LAC search about this user is likely to return people who are also interested in traveling (since their vertices have keywords related to traveling). A tour can then be organized for the people contained in the LAC. Advertisement about traveling products (e.g., cameras and backpacks) can also be recommended to this LAC. We develop the notion of LAC based on the structure cohesiveness metric minimum degree [20, 8, 24, 5], which requires that every vertex in the community has a degree of $k$ or more. We have studied several possible criteria for defining keyword cohesiveness (e.g., that the number of keywords shared by LAC vertices is maximum, or each vertex’s keyword set contains a user-given keyword set). We further investigate the LAC search problem, which aims to find the LAC(s) that contain(s) a given vertex $q$.

Let us further illustrate the LAC search query on the DBLP bibliographical network, with $q = Jim Gray$ and $k = 4$ (i.e., the minimal degree of each LAC vertex is 4). For discussions, we define the label of an LAC to be a set of keywords that appear in all the vertices of the LAC. Figure 2 shows the two LACs found and their respective labels. We can see that the two LACs contain completely different authors (except Jim himself). Moreover, the label of each LAC reflects the backgrounds of the people within that LAC (e.g.,

\[ \text{(a) \{transaction, data, management, system, research\}} \quad \text{(b) \{loan, digital, sky, data, sds\}} \]

Figure 2: Two LACs of Jim Gray (with LAC labels).

In Figure 2(a), the 6 researchers connected to Jim are well known for their work in database systems; in Figure 2(b), the 7 scientists linked to Jim are involved in the SDSS (or Sloan Digital Sky Survey) project. We have also performed search on $q$ using the existing algorithm [24], and found that the community contains all the 14 vertices shown in Figures 2(a) and (b). The main reasons are: (1) these nodes are all heavily linked to Jim; and (2) vertex keywords are not considered in the community search process. It is not easy from this community to understand which researcher has what kind of collaboration with Jim. In contrast, the LAC search places these vertices into two communities, containing vertices that are cohesive in terms of structure and keyword. The LACs found allow a user to focus on the important vertices that constitute a community. For example, using the LAC of Figure 2(a), a database conference organizer can invite speakers who have a close relationship to Jim. As we will explain later, our LAC search solutions use vertex keywords effectively, and are better than existing algorithms [24, 5] that do not use these keywords.

Performing an LAC search is not straightforward, since the graph $G$ to be explored can be very large, and the (structure and keyword) cohesiveness criteria for governing the community search are complex. To tackle this challenge, a straightforward 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. However, this solution, which requires the enumeration of all the subsets of $q$’s keyword set, has a complexity exponential to the size $l$ of $q$’s keyword set. In our experimental evaluation, for some queries, $l$ can have a value of 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 LAC, then for every subset $S'$ of $S$, there exists an LAC in which every vertex contains $S'$. We use this intuition to propose better algorithms. We further develop the CL-tree, an index that organizes the vertex keyword data in a hierarchical structure. The CL-tree has a space and construction time complexity linear to the size of $G$. We have developed three different LAC search algorithms based on the CL-tree, and they are able to achieve a superior performance.

We have performed extensive experiments on our solutions on four large real graph datasets (namely Flickr, DBLP, Tencent, and DBpedia). We have developed several measures to measure 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 LAC over the communities returned by existing algorithms, in terms of community quality. Particularly, compared to two representative algorithms [24, 5], it is generally easier to understand the meaning behind the LACs; the label of the LAC can tell clearly what the community is about. The performance of our best algorithm is 2 to 3 order-of-magnitude better than solutions that do not use the CL-tree. Another advantage of our approaches is that they organize and search vertex keywords for LACs effectively, achieving a higher efficiency than existing community search solutions (that do not use

\[ 1 \text{URL of the SDSS project: http://www.sdss.org.} \]
vertex keywords in the community search process).

As a summary, we propose the LAC, which is a new definition of the community that considers both structure and keyword cohesiveness. To perform efficient LAC search, we develop the CL-tree and its associated algorithms. We perform experimental evaluation on real graph datasets to validate our approaches. Notice that although our solutions are developed based on the minimum degree metric in this paper, they can potentially be used to address other definitions of communities (e.g., k-truss). Our study can also inspire the development of better community detection algorithms that take vertex keywords into account.

The rest of our paper is organized as follows. In Section 2 we give the problem definition. Section 3 presents the basic solutions, and Section 4 discusses the CL-tree. We present the query algorithms in Section 5. Our experimental results are reported in Section 6. We review the related work in Section 7 and conclude in Section 8.

2. THE LAC SEARCH PROBLEM

We now discuss the graph model, the k-core, and the LAC.

Let \( G(V, E) \) be an undirected graph with vertex set \( V \) and edge set \( E \). Let \( n \) and \( m \) be the corresponding sizes of \( V \) and \( E \). For every \( v \in V \), the degree of \( v \) is denoted by \( \deg(v) \). We let \( W(v) \) be the set of keywords associated with \( v \).

In general, a community is a subgraph of \( G \) that satisfies a structure cohesiveness criterion (i.e., the vertices contained in the community are linked to each other in some way). A well-accepted notion of structure cohesiveness is that the minimum degree of all the vertices that appear in the community has to be \( k \) or more [24, 23, 2, 7, 5, 17]. This is used in the k-core and the LAC. Let us discuss the k-core first.

**Definition 1 (k-core [23, 2]).** Given an integer \( k \geq 0 \), the k-core of \( G \), denoted by \( H_k \), is the largest subgraph of \( G \), such that \( \forall v \in H_k, \deg_{H_k}(v) \geq k \).

We say that \( H_k \) has an order of \( k \). Notice that \( H_k \) may not be a connected graph [2], and its connected components, denoted by \( k\text{-cores} \), are usually the “communities” returned by k-core search algorithms.

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

Observe that k-cores are “nested” [2]; given two positive integers \( i \) and \( j \), if \( i < j \), then \( H_i \subseteq H_j \). In Figure 3(a), \( H_3 \) is contained in \( H_2 \), which is nested within \( H_1 \).

**Definition 2 (Core number).** Given a vertex \( v \in V \), its core number, denoted by \( \text{core}_v \), is the highest order of a k-core that contains \( v \).

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

An efficient solution (called **Global**) for finding a k-core that contains a vertex \( q \in V \) was presented in [24]. Recently, Cui et al. have developed a fast solution called **Local** [5], yielding subgraph(s) of k-core that satisfy structure cohesiveness. In Section 6, we compare our approach with Global and Local.

As discussed in Section 1, two vertices that do not share any keywords may still be placed together in a k-core. In Example 1, \( H \) and \( I \) are included in the 1-core even though their keywords are completely different. To address this issue, we introduce the LAC search problem.

**Problem 1 (LAC search).** Given a graph \( G \), a positive integer \( k \) and a vertex \( q \in V \), return a set \( G_q \) of graphs, such that \( \forall G_q \in G_q, \) the following properties hold:

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

We call \( G_q \) the label-aware community (or LAC) of \( q \), and \( L(G_q) \) the label of \( G_q \). In Problem 1, the first two properties are also specified by the k-core of a given vertex \( q \) [24]. The **keyword cohesiveness** (Property 3), which is unique to Problem 1, enables the retrieval of communities whose vertices have common keywords. In Figure 3(a), if \( q = A \) and \( k = 2 \), the output of Problem 1 is \( \{A, C, D\} \), with label \( \{x, y\} \) (i.e., these vertices share the keywords \( x \) and \( y \)).

The reason \( L(G_q) \) is maximal in Property 3 is that we wish the LAC(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 (\( q = A \) and \( k = 2 \)), 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). Our experiments (Section 6) show that imposing the “maximal” constraint yields the best result. Thus, we adopt Property 3 in Problem 1. For the rare case in which there is no LAC whose vertices share one or more keywords (i.e., \( |L(G_q)| = 0 \)), we return the subgraph of \( G \) that satisfies Properties 1 and 2 only. 2

There are other ways for defining keyword cohesiveness. For example, given a positive integer \( h \), return the LAC(s) whose label(s) contain at least \( h \) keywords. As discussed, this may not be as good as requiring a maximal sharing of keywords (especially when \( h \) is small). This may also be less convenient to the query user, who needs to specify the value of \( h \). Another option is to ask the query user to input a set \( S \) of keywords, and the labels of the LACs returned has to contain \( S \). This enables the user to control over the semantics of the LACs desired. An approximate version of

---

**Figure 3: Illustrating the k-core and the LAC.**

(a) graph

(b) core number
this definition is that given a threshold \( \theta \in [0, 1] \), each vertex in an LAC has at least \( |S| \times \theta \) keywords in \( S \). We found that these variants can be easily supported by our solutions to Problem 1. Their definitions, solutions, and results are in Appendix F. We next focus on algorithms for Problem 1.

3. BASIC SOLUTIONS

For ease of presentation, we say that \( v \) contains a set \( S \) of keywords, if \( S \subseteq W(v) \). We use \( G[S, q] \) to denote the largest connected subgraph of \( G \), where each vertex contains \( S \) and \( q \in G[S, q] \). We use \( G_k[S, q] \) to denote the largest connected subgraph of \( G[S, q] \), in which every vertex has degree being at least \( k \) in \( G_k[S, q] \). We call \( S \) a qualified keyword set for the query vertex \( q \) on the graph \( G \), if \( G_k[S, q] \) exists. If \( S \) is the largest subset of \( W(q) \), then \( G_k[S, q] \) is a target LAC, as it satisfies all the cohesiveness of LAC search.

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

One major drawback of the straightforward method is that we need to compute \( 2^{|W(q)|} - 1 \) subgraphs \((i.e., G_k[S, q])\). 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.

3.1 Two-Step Framework

The two-step framework is 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 \( G_k[S, q] \), then there exists a subgraph \( G_k[S', q] \) for any subset \( S' \subseteq S \).

The proofs of all the 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 \( S_i \), once we have verified that \( G_k[S_i, q] \) does not exist. The basic solution begins with examining the set, \( \Psi_1 \), of size-1 candidate keyword sets, \( i.e., each candidate contains a single keyword of W(q) \). It then repeatedly executes the following two key steps, to retrieve the size-2 (size-3, \ldots) qualified keyword subsets until no qualified keyword sets are found.

- **Verification.** For each candidate \( S \) in \( \Psi_c \) (initially \( c=1 \)), mark \( S \) as a qualified set if \( G_k[S, q] \) 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 \( \Psi_{c+1} \), if all its subsets are qualified, by Lemma 1.

Among the above steps, the key issue is how to compute \( G_k[S, q] \). Since \( G_k[S, q] \) should satisfy the structure cohesiveness \((i.e., minimum degree at least k)\) and keyword cohesiveness \((i.e., every vertex contains keyword set \( S_i \))\).

Intuitively, we have two approaches to compute \( G_k[S, q] \): 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 listed in Appendix B.

4. CL-TREE INDEX

The major limitation of **basic-g** and **basic-w** is that they need to find the \( k \)-cores 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 \( k \)-cores and keywords into a tree structure. Based on the index, the efficiency of LAC search and its variants can be improved significantly, as we will show later. We first introduce the proposed index in Section 4.1, and then propose two methods to build the index in Section 4.2.

4.1 Index Overview

The CL-tree index is built based on the key observation that cores are nested. Specifically, a \((k+1)\)-core must be contained in a \( k \)-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 \( k \)-cores can be organized into a tree structure\(^3\). We illustrate this in Example 2.

![Figure 4: an example CL-tree index.](image)

**Example 2.** Consider the graph in Figure 3(a). All the \( k \)-cores 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 \( k \)-core.

From the tree structure in Figure 4(a), we conclude that, if a \((k+1)\)-core (denoted as \( C_{k+1} \)) is contained in a \( k \)-core (denoted as \( C_k \)), then there is a tree node corresponding to \( C_k \) and its parent node corresponds to \( C_{k-1} \). 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. Thus, 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. Then, the proposed CL-tree index is composed of such a compressed tree structure and additional inverted lists of keywords associated to each tree node. The associated keyword list for a

\(^3\)We use “node” to mean “CL-tree node” in this paper.
tree node (or equally, a $k$-core) stores the appeared keyword-expert vertices of the node, as well as vertices containing them. Such associated keyword lists improve the efficiency for keyword related pruning in LAC search. In summary, each node of the CL-tree contains four elements:

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

Figure 4(b) depicts the CL-tree index for the example graph in Figure 3(a), the elements of each tree node are labeled explicitly. With CL-tree, the following two operations, which act as building blocks for our query algorithms (Section 5), can be performed efficiently:

- **Core-locating**: Given a vertex $q$ and a core number $c$, locate the $k$-core with core number $c$ containing $q$ efficiently, by traversing the CL-tree.
- **Keyword-checking**: Given a $k$-core, find vertices which contain a given keyword set efficiently, by intersecting the inverted lists with the keyword set.

**Remarks.** The CL-tree can also support $k$-core queries on general graphs without keywords. For example, it can be applied to finding $k$-cores in previous community search [24].

**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 $O(l \cdot n)$, where $l$ denotes the average size of $W(v)$ over $V$. Thus, the space cost is linear with the size of the input graph.

### 4.2 Index Construction

To build the CL-tree index, we propose two methods, basic and advanced, as presented in Section 4.2.1 and 4.2.2.

#### 4.2.1 The Basic Method

As $k$-cores 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 $k$-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 illustrates the pseudocodes. We first do $k$-core decomposition using the linear algorithm [2], and obtain an array $coreC[i]$, where $coreC[i]$ is the core number of vertex $i$ in $G$. We denote the maximal core number by $k_{\text{max}}$. Then, we initialize the root node by the core number $k = 0$ and $V$ (line 3). Next, we call the function $\text{BUILDNODE}$ 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 $\text{BUILDNODE}$, we first update $k$ and obtain the vertex set $U[k]$ from $\text{root.vertexSet}$, 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 $k$-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 $\text{BUILDNODE}$ function to build $p_i$’s child nodes recursively (line 16).

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

#### 4.2.2 The Advanced Method

While the basic method is easy to implement, it meets efficiency issues when both the given graph size and its $k_{\text{max}}$ 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 $k_{\text{max}}$ is $n-1$. Therefore, the time complexity of the basic method could be $O(m + l \cdot n)$, 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 of tree node creation and tree edge creation.

1. **Tree node creation.** 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 $c$-cores. A simple algorithm would be, searching connected components for $T_c(0 \leq c \leq k_{\text{max}})$ independently, followed by creating one node for each distinct component. This algorithm apparently costs $O(k_{\text{max}} \cdot m)$ time given that the connected component search for a graph can be completed in 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

**Algorithm 1 Index construction: basic**

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

1. Tree node creation.
2. Tree edge creation.

---

**Algorithm 1 Index construction: advanced**

1. function $\text{BUILDINDEX}(G(V, E))$
2. $coreC[] \leftarrow k$-core decomposition on $G$;
3. $k \leftarrow 0$, root $\leftarrow (k, V)$;
4. $\text{BUILDINDEX}(\text{root}, 0)$;
5. build an inverted list for each tree node;
6. return root;
7. function $\text{BUILDINDEX}(\text{root}, k)$
8. $k \leftarrow k + 1$;
9. if $k \leq k_{\text{max}}$ then
10. obtain $U_k$ from root;
11. compute the connected components for the induced graph on $U_k$;
12. for each connected component $C_i$ do
13. build a tree node $p_i \leftarrow (k, C_i, \text{vertexSet})$;
14. add $p_i$ as a child node of root;
15. remove $C_i$’s vertex set from root.$\text{vertexSet}$;
16. $\text{BUILDINDEX}(p_i, k)$;
To address this issue, we thus propose an auxiliary data structure, called Anchored Union-Find (AUF), more details are in Appendix D), based on the classical union-find forest. We define anchor vertex as follows.

**Definition 3 (Anchor vertex).** Given a connected subgraph \( G' \subseteq G \), the anchor vertex is the vertex with the smallest core number among all the vertices 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 \( k \)-core \( C_p \), its child nodes correspond to the \( k \)-cores, which are contained by \( C_p \) 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 \( k \)-cores dynamically, and they can be used to link nodes with their child nodes. In addition, we maintain a vertex-node map, where the key is a vertex and the value is the tree node contains this vertex, for locating tree nodes. The detailed pseudocodes and analysis are reported in Appendix E.

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

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

5. **QUERY ALGORITHMS**

In this section, we present three query algorithms based on the CL-tree index. Based on how we verify the candidate keyword sets, we divide our algorithms into incremental algorithms (from examining smaller candidate sets to larger ones) and decremental algorithm (from examining larger candidate sets to smaller ones). We propose two incremental algorithms called Inc-S (Incremental Space-efficient) and Inc-T (Incremental Time efficient), to trade off between the memory consumption and the computational overhead. We also design a decremental algorithm called Dec (Decremental). Our interesting finding is that, while Dec seems not intuitive, it ranks as the most efficient one. Inc-S and Inc-T are presented in Section 5.1. Dec is introduced in Section 5.2.

5.1 **The Incremental Algorithms**

While the high-level idea of incremental algorithms resembles the basic solutions (see Section 3), Inc-S and Inc-T advance them with the exploitation of the CL-tree. Specifically, they can always verify the existence of \( G_k[S, q] \) 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 by Inc-S and Inc-T.

5.1.1 **Inc-S Algorithm**

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

**Definition 4 (Subgraph core number).** Given a subgraph \( G' \) (\( G' \subseteq G \)), its core number, \( \text{core}_{G'}(G') \), is defined as \( \min \{ \text{core}_{G'}(v) \mid v \in G' \} \).

Inc-S follows the two-step framework (verification and candidate generation) introduced in Section 3. With the CL-tree and the definition of subgraph core number, we improve the verification step as follows.

- **Core-based verification.** For each newly generated size-\( (c-1) \) candidate keyword set \( S \) that is expanded from size-\( c \) sets \( S_1 \) and \( S_2 \), mark \( S \) as a qualified set if \( G_k[S, q] \) exists within a \( k \)-core of core number \( \max \{ \text{core}_{G_k[S_1, q]}, \text{core}_{G_k[S_2, q]} \} \).
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 \( G_k[S, q] \) in a smaller \( k\)-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 \( G_k[S_1, q] \) and \( G_k[S_2, q] \) 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 \( G_k[S, q] \) exists, then it must appear in an \( k\)-core with core number at least

\[
\max\{\text{core}_C[G_k[S_1, q]], \text{core}_C[G_k[S_2, q]]\}. \tag{1}
\]

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

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

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

We present \( \text{Inc-S} \) in Algorithm 2. The input is a CL-tree rooted at \( \text{root} \), a query vertex \( q \) and an integer \( k \) to confine the minimum degree. We apply \( \text{core-locating} \) on the CL-tree to locate the internal nodes whose corresponding \( \text{k-cores} \) contain \( q \) (line 2). Note that their core numbers are in the range of \( [k, \text{core}_C[q]] \), as required by the structure cohesiveness. Then, we set \( l=0 \), indicating the sizes of current keyword sets, and initialize a set \( \Psi \) of \( \{r \} \) at node \( \langle S, G \rangle \). In the first while loop (lines 4-18), for each \( \langle S, c \rangle \) pair, we first perform \( \text{keyword-checking} \) to find \( G[S, q] \) using the keyword inverted lists of the subtree rooted at node \( r_c \). If we cannot ensure that \( G[S, q] \) does not contain a \( k\)-core by Lemma 3, we then find \( G_k[S, q] \) from \( G[S, q] \) (lines 8-9). If \( G_k[S, q] \) exists, we put \( S \) with its core number into the set \( \Phi_l \) (lines 10-11). Next, if \( \Phi_l \) is nonempty, we generate new candidates by calling \( \text{GECAND}(\Phi_l) \), which is detailed in Appendix C. For each candidate \( S \) in \( \Psi \), we compute its core number using Lemma 2 and update it as a pair in \( \Psi \) (lines 12-17); otherwise, we stop (line 18). Finally, we output the communities of the latest verified keyword sets as the target LACs (line 19).

**Algorithm 2 Query algorithm: \( \text{Inc-S} \)

1. function \( \text{QUERY}(G, \text{root}, q, k) \)
2. \( \text{find subtree root nodes} r_2, r_{k+1}, \ldots, r_{\text{core}_C[q]} \)
3. \( \text{initialize} l=0, \Psi \) using \( W(q) \)
4. \( \text{while true do} \)
5. \( l \leftarrow l + 1; \Phi_l \leftarrow \emptyset \)
6. \( \text{for each} \langle S, c \rangle \in \Psi \) do
7. \( \text{find} G[S, q] \) under the root \( r_c \)
8. \( \text{if} G[S, q] \) is not pruned by Lemma 3 then
9. \( \text{find} G_k[S, q] \) from \( G[S, q] \)
10. \( \text{if} G_k[S, q] \) exists then
11. \( \Phi_l \text{add} \langle S, \text{core}_C[G_k[S, q]] \rangle \)
12. \( \text{if} \Phi_l \neq \emptyset \) then
13. \( \Psi \leftarrow \text{GECAND}(\Phi_l) \)
14. \( \text{for each} S \in \Psi \) do
15. \( \text{if} S \) is generated from \( S_1 \) and \( S_2 \) then
16. \( c \leftarrow \max\{\text{core}_C[G_k[S_1, q]], \text{core}_C[G_k[S_2, q]]\} \)
17. \( \Psi \text{update}(S, \langle c, S \rangle) \)
18. \( \text{else break} \)
19. \( \text{output the communities of keyword sets in} \Phi_{l-1} \)

For \( S_2 \), we do not need to consider the keyword constraint again when finding \( G_k[S, q] \).

Based on Lemma 4, we introduce a new algorithm \( \text{Inc-T} \). Different from \( \text{Inc-S} \), \( \text{Inc-T} \) maintains \( G_k[S, q] \) rather than \( \text{core}_C[G_k[S, q]] \) for each qualified keyword set \( S \). As we will demonstrate later, \( \text{Inc-T} \) is more effective for shrinking the subgraphs containing the LACs, and thus more efficient. As a trade-off for better efficiency, \( \text{Inc-T} \) consumes more memory as it needs to store a list of subgraph \( G_k[S, q] \) in memory.

Algorithm 3 presents \( \text{Inc-T} \). We first apply \( \text{core-locating} \) to find the \( k\)-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 \( \Psi \) of \( \{<S,G>\} \) pairs, where \( S \) is a set containing a keyword from \( W(q) \) and \( G \) is the initial \( k\)-core (line 3). In the first while loop (lines 4-18), for each \( <S,c> \) pair, we first perform \( \text{keyword-checking} \) to find \( G[S, q] \) using the keyword inverted lists of the subtree rooted at node \( r_c \). If we cannot ensure that \( G[S, q] \) does not contain a \( k\)-core by Lemma 3, we then find \( G_k[S, q] \) from \( G[S, q] \) (lines 8-9). If \( G_k[S, q] \) exists, we put \( S \) with its core number into the set \( \Phi_l \) (lines 10-11). Next, if \( \Phi_l \) is nonempty, we generate new candidates by calling \( \text{GECAND}(\Phi_l) \), which is detailed in Appendix C. For each candidate \( S \) in \( \Psi \), we compute its core number using Lemma 2 and update it as a pair in \( \Psi \) (lines 12-17); otherwise, we stop (line 18). Finally, we output the communities of the latest verified keyword sets as the target LACs (line 19).

**Algorithm 3 Query algorithm: \( \text{Inc-T} \)

1. function \( \text{QUERY}(G, \text{root}, q, k) \)
2. \( \text{find k-core containing} q \)
3. \( \text{initialize} l=0, \Psi \) using \( W(q) \)
4. \( \text{while true do} \)
5. \( l \leftarrow l + 1; \Phi_l \leftarrow \emptyset \)
6. \( \text{for each} \langle S, G \rangle \in \Psi \) do
7. \( \text{find} G[S, q] \) from \( G \)
8. \( \text{if} G[S, q] \) is not pruned by Lemma 3 then
9. \( \text{find} G_k[S, q] \) from \( G[S, q] \)
10. \( \text{if} G_k[S, q] \) exists then
11. \( \Phi_l \text{add} \langle S, \text{core}_C[G_k[S, q]] \rangle \)
12. \( \text{if} \Phi_l \neq \emptyset \) then
13. \( \Psi \leftarrow \text{GECAND}(\Phi_l) \)
14. \( \text{for each} S \in \Psi \) do
15. \( \text{if} S \) is generated from \( S_1 \) and \( S_2 \) then
16. \( \hat{G} \leftarrow G_k[S_1, q] \cap G_k[S_2, q] \)
17. \( \Psi \text{update}(S, \langle \hat{G}, \hat{S} \rangle) \)
18. \( \text{else break} \)
19. \( \text{output the communities of keyword sets in} \Phi_{l-1} \)

**Example 5.** Continue the graph and query \( (q=A, k=1) \)
in Example 4. By Algorithm 3, we first find $G_1\{x, A\}$ and $G_1\{y, A\}$, whose vertex sets are $\{A, B, C, D\}$ and $\{A, C, D, E, F, G\}$. Then to find $G_1\{x, y, A\}$, we only need to search it from the subgraph, induced by the vertex set $\{A, C, D\}$.

5.2 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. We now introduce them as follows.

1. **Generation of candidate keyword sets.** Dec exploits the key observation that, if $S (S \subseteq W(q))$ is a qualified keyword set, then there are at least $k$ of $q$’s neighbors containing $S$. This is because every vertex in $G_q[S, q]$ must have 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 its neighbor vertices. For each vertex $v$, we only select the keywords, which are contained by 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 $deg_c(q)$ itemsets. Then we apply the well studied frequent pattern mining algorithms (e.g., Apriori [10] and FP-Growth [11]) 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 $k$ vertices, we set the minimum support as $k$. In this paper, we use the well-known FP-Growth algorithm [11, 9].

![Figure 6: An example of candidate generation.](image)

**Example 6.** Consider a query vertex $Q$ with $6$ neighbors in Figure 6(a), where the selected keywords of each vertex are listed in the braces. By FP-Growth, $8$ candidate keyword sets ($k=3$) are generated, as shown in Figure 6(b). $\Psi_1$ denotes the set of keyword sets with sizes being $i$.

2. **Verification of candidate keyword sets.** As candidates can be obtained using $q$ and $q$’s neighbors directly, we can verify them either incrementally as that in Inc-3, or in a decremental manner (larger candidate keyword sets first and smaller candidate keyword sets later). In this paper, we choose the latter manner. The rationale behind is that, for any two keyword sets $S_1 \subseteq S_2$, the number of vertices containing $S_2$ is usually smaller than that of $S_1$, which implies $S_2$ can be verified more efficiently than $S_1$.

Based on the above discussions, we design Dec as shown in Algorithm 4. We first generate candidate keyword sets using $q$ 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 $k$-core contains $q$ (line 3). Next, we traverse the subtree rooted at $r_k$ and find vertices which share keywords with $q$ (line 4). $R_i$ denotes 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 $\hat{R}$ 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 set $S \in \Psi_1$, we first try to find $G_q[S, q]$ in the subgraph induced on $\hat{R}$; find $G_q[S, q]$ from $G_q[S, q]$; if $G_q[S, q]$ exists then find $G_q[S, q]$ and $G_q[S, q]$ into $S$ if it exists (lines 8-12). Finally, if we have found at least one qualified community, we stop at the end of this loop and output $Q$ as the target LACs; otherwise, we examine smaller candidate keyword sets in next loop.

![Algorithm 4: Query algorithm: Dec](image)

6. **EXPERIMENTS**

We now present the results. Section 6.1 discusses the setup. We discuss the results in Sections 6.2 and 6.3.

6.1 Setup

We consider four real datasets. For Flickr\(^4\) [25], 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 1 shows the number of vertices and edges, the $k_{max}$ value, a vertex’s average degree $\bar{d}$, and its keyword set size $\hat{d}$.

To perform LAC search, we set the default value of $k$ to 6. For each dataset, we randomly select 300 query vertices with core numbers of 6 or more, which ensures that there is a $k$-core containing each query vertex. Each data point is the average result for these 300 queries.

We implement all the algorithms in Java, and run the experiments on a machine having a quad-core Intel i7-3770

\(^4\)https://www.flickr.com/
\(^5\)http://dblp.uni-trier.de/xml/
\(^6\)http://www.kddcup2012.org/c/kddcup2012-track1
\(^7\)http://dbpedia.org/datasets
3.40GHz processor, 32GB of memory, and a 1TB hard disk, with Ubuntu installed. We present the effectiveness and performance results of our approaches in Sections 6.2 and 6.3.

### 6.2 Results on Effectiveness

We now study the effectiveness of the communities found by 3 community search solutions: LAC (implemented by Dec, our fastest solution), Global [24], and Local [5], both discussed in Section 2. We then discuss a case study.

#### 6.2.1 Keyword Effectiveness

We consider two aspects of a community: structure and keyword cohesiveness. As discussed in Section 2, LAC, Global, and Local all satisfy structure cohesiveness. We thus focus on the keyword cohesiveness of these solutions, and compare them with respect to two measures, namely CMF and CPJ. Let \( C(q) = \{C_1, C_2, \ldots, C_L\} \) be the set of \( L \) communities returned by an algorithm for a query vertex \( q \in V \).

- **Community member frequency (CMF):** This is inspired by the classical document frequency measure [22]. 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 \( f_{i,k} \) be the number of vertices of \( C_i \) whose keyword sets contain the \( k \)-th keyword of \( W(q) \). Then, \( \frac{f_{i,k}}{W(q)} \) 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 \( C(q) \):

\[
CMF(C(q)) = \frac{1}{L \cdot |W(q)|} \sum_{i=1}^{L} \sum_{k=1}^{W(q)} \frac{f_{i,k}}{|C_i|}
\]  

(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 \( C_j \) 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 \( C(q) \):

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

(4)

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

Figure 7 shows the CMF and CPJ values for the four datasets. We can see that the keyword cohesiveness of LAC is superior to both Global and Local, because LAC considers vertex keywords, while Global and Local do not.

**Effect of common keywords.** We next examine the impact of the length of an LAC label (i.e., the number of keywords shared by all the vertices of the LAC) on keyword cohesiveness. We collect LACs containing one to five keywords, and then group the LACs according to their label lengths. The average CMF and CPJ value of each group is shown in Figure 8. For all the datasets, when the label lengths increase, both CMF and CPJ value rises. This justifies the use of the maximal label length as the criterion of returning an LAC in Problem 1.

#### 6.2.2 A Case Study

We next perform a case study on the DBLP dataset, in which we consider the communities of two renowned researchers in database and data mining: Jim Gray and Jiawei Han. We use \( k = 4 \) here. We have discussed the LACs of Jim (Figure 2) in Section 1. Figure 9 shows two LACs of Jiawei, whose LAC labels contain four or more keywords. (The LAC labels are shown in the captions.) These two groups of Jiawei’s collaborators are involved in graph analysis (Figure 9(a)) and pattern mining (Figure 9(b)). Although these researchers all have close co-author relationship with Jiawei, the use of vertex keywords enables the identification of communities with different research themes. Similar observations can be made for Jim’s case. However, these researchers are placed into a single \( k \)-core. We next give a more detailed analysis in terms of their keywords and community sizes.

**Figure 7: Keyword cohesiveness.**

**Figure 8: LAC label length.**

**Figure 9: Jiawei Han’s LACs.**

1. **Keyword analysis.** Table 2 shows the CMF and CPJ values of Jim and Jiawei’s communities. Observe that LAC is more effective than Global and Local.

We further analyze the frequency distribution of keywords in their communities. Specifically, given a keyword \( w_b \), we define the member frequency (MF) of \( w_b \) as: \( MF(w_b, C(q)) = \)
As shown in Table 3, the $k$-core returned by Global has over 139K distinct keywords, about 2,300 times more than that returned by LAC (less than 60 keywords). While the semantics of the $k$-core can be difficult to understand, the small number of distinct keywords of LAC makes it easier to understand why the community is so formed. We further report the keywords with the 6 highest MF values in the communities in Tables 4 and 5. For Jim’s LAC, the words “sloan,“digital,”sky,”data,”sdss” reflect that the community is likely about the SDSS project led by Jim. The top-6 keywords of Jiawei’s LAC are related to heterogeneous networks. We also observe that the top-6 keywords of Global are the same for both Jim and Jiawei, as they are in the same 4-core. Thus, they cannot be used to characterize the communities specifically related to Jim and Jiawei.

2. Community size analysis. We have also studied the average size of communities under different values of $k$. As we can see in Figure 11, 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 resorts to Global, and returns the same community as Global [5]. The size of an LAC is relatively insensitive to the change of $k$. In this experiment, the LAC has around 139,881 community members. The number of distinct keywords of LAC communities is also the fewest, as shown in Table 3.

6.3 Results on Efficiency

For the graph in each dataset, we randomly select 20%, 40%, 60% and 80% of its vertices, and obtain four subgraphs induced by these vertex sets. Also, for each vertex in each dataset, we randomly select 20%, 40%, 60% and 80% of its keywords, and obtain four sub-keyword sets. The experimental results of index construction and LAC search are reported in Figures 12 and 13 respectively.

1. index construction scalability. Figures 12(a)-(d) compare the efficiency of the Basic and Advanced methods. Since their main difference is the part of building the tree structure, we also compare the parts of only building the tree without keywords. We denote them by Basic- and Advanced- respectively. Advanced performs consistently faster than Basic. Also, Advanced scales better than Basic. When the size of subgraph increases, the performance gap between Advanced and Basic becomes larger. Similar results can be observed between Advanced- and Basic-.

2. comparison with Global and Local. Figures 13(a)-(d) compare our best algorithm Dec with Global and Local. Similar with the results reported in [5], Local performs faster than Global for most cases. Dec is the most efficient one on all the datasets, because it uses the CL-tree index.

3. effect of $k$. Figures 13(e)-(h) compare the query efficiency under different $k$ values. A lower $k$ value implies that the returned subgraph tends to be larger. Hence, more time cost is needed for all algorithms. For all the datasets, basic-g performs faster than basic-w, but they are slower than the index based algorithms. Inc-T performs better than Inc-S, and Dec performs the best. Besides, the performance gaps decrease as the values of $k$ increase.

4. query scalability w.r.t. keyword. Figures 13(i)-(l) compare the queries’ scalability w.r.t. the number of keywords. Note that for each dataset, we only vary the percentage of keywords for each vertex, and all the vertices are considered. Dec scales the best. Besides, their running time increases as more keywords are involved.

5. query scalability w.r.t. vertex. Figures 13(m)-(p) compare the queries’ scalability w.r.t. the number of vertices. Note that for each dataset, we only vary the percentage of vertices, and all the keywords of each vertex are considered. Again, Dec scales the best.

6. analysis of Dec. To analyze why Dec performs the best, in Figures 13(q)-(t), we plot the average number of vertices, that share different numbers of keywords with the query vertices. Note that for each dataset, we vary the percentage of vertices from 20% to 100% (all the keywords of each vertex are kept). Recall that, Dec verifies candidate keyword sets in a decremental manner, i.e., larger keyword sets are examined before smaller ones. We can see that,

### Table 2: Keyword cohesiveness.

<table>
<thead>
<tr>
<th>Researcher</th>
<th>Global</th>
<th>Local</th>
<th>LAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jim Gray</td>
<td>0.404</td>
<td>0.608</td>
<td>0.296</td>
</tr>
<tr>
<td>Jiawei Han</td>
<td>0.366</td>
<td>0.250</td>
<td>0.600</td>
</tr>
</tbody>
</table>

### Table 3: # distinct keywords of communities.

<table>
<thead>
<tr>
<th>Researcher</th>
<th>Global</th>
<th>Local</th>
<th>LAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jim Gray</td>
<td>139,881</td>
<td>60</td>
<td>44</td>
</tr>
<tr>
<td>Jiawei Han</td>
<td>139,881</td>
<td>58</td>
<td>54</td>
</tr>
</tbody>
</table>

### Table 4: Top-6 keywords (Jim Gray).

<table>
<thead>
<tr>
<th>Algo.</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>use, system, model, network, analysis, data</td>
</tr>
<tr>
<td>Local</td>
<td>database, system, multipetabyte, data, lsst, story</td>
</tr>
<tr>
<td>LAC</td>
<td>Sloan, digital, sky, data, sdss, server</td>
</tr>
</tbody>
</table>

### Table 5: Top-6 keywords (Jiawei Han).

<table>
<thead>
<tr>
<th>Algo.</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>use, system, model, network, analysis, data</td>
</tr>
<tr>
<td>Local</td>
<td>scalable, topical, text, phrase, corpus, mine</td>
</tr>
<tr>
<td>LAC</td>
<td>mine, analysis, data, information, network, heterogen</td>
</tr>
</tbody>
</table>

![Figure 10: Frequency distribution of keywords.](image-url)
Figure 12: Efficiency results of index construction.

Figure 13: Efficiency results of LAC search.
the number of vertices decreases very fast as the number of shared keywords increases. So if we verify larger keyword sets first, we can find the LACs from the subgraph induced by a smaller number of vertices. Hence, Dec performs fast.

7. RELATED WORK

Community detection and search. Community detection aims to discover all the communities from a graph. To related topic community search, finds communities for particular vertices [24, 5, 17, 4, 13]. To measure the structure cohesiveness of a community, the minimum degree is often used [24, 5, 17]. As discussed before, Sozio et al. [24] proposed the first algorithm Global to find the k-core containing q. Cui et al. [5] proposed Local, which uses local expansion techniques to enhance the performance of Global.

Our LAC search performs better than them experimentally. Other definitions, including k-clique [4], k-truss [13], and edge-surplus [26], have also been considered for searching communities. A recent work [17] finds communities with high influence. Some other works [16, 18] are about uncovering communities from seed member sets. These works overlook the rich information of vertices, which is considered in the LAC search process.

Graph keyword search. Given a graph G and a set Q of keywords, graph keyword search solutions output a tree structure, whose nodes are vertices of G, and the union of these vertices’ keyword sets is a superset of Q [3, 6, 12, 14, 27]. Recent work studies the use of a subgraph of G as the query output [19, 21, 15]. These works are substantially different from the LAC search problem. First, they do not specify query vertices as required by the LAC search. Second, the tree or subgraph produced do not guarantee structure cohesiveness. Third, keyword cohesiveness is not ensured; there is no mechanism that enforces query keywords to be shared among the keyword sets of all query output’s vertices. Thus, graph keyword search solutions are not designed to solve the LAC search problem.

8. CONCLUSION

We examine the LAC search problem, which finds communities that exhibit both structure and keyword cohesiveness. To enable efficient LAC search, we develop the CL-tree index and its query algorithms. Our experimental results show that LACs are easier to interpret than k-cores. Our LAC solutions are also faster than existing k-core search algorithms. In the future, we will investigate keyword cohesiveness in other community definitions (e.g., k-truss), as well as community detection problems.

9. REFERENCES

APPENDIX

A. PROOFS OF LEMMAS

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

**Proof.** Based on the definition of $G[S, q]$, each vertex of $G[S, q]$ contains $q$, Consider a new keyword set $S' \subseteq S$. We can easily conclude that, each vertex of $G[S', q]$ contains $S'$ as well. Also, note that $q \in G[S, q]$. These two properties imply that there exists one subgraph of $G$, namely $G[S, q]$, 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., $G[S', q]$).

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

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

**Lemma 2.** Given two subgraphs $G[S_1, q]$ and $G[S_2, q]$ 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 $G[S, q]$ exists, then it must appear in a $k$-core with core number at least

$$\max \{core_{G[S_1, q]}, core_{G[S_2, q]}\}.$$  

**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 $G[S, q] \subseteq G[S_1, q]$. With such a containment relationship, it follows that $\min \{core_{G[S, q]} \} v \in G[S_1, q] \leq \min \{core_{G[S, q]} \} v \in G[S, q]$. Hence, the core number of $G[S, q]$ is at least the core number of $G[S_1, q]$. Formally, $core_{G[S, q]} \subseteq core_{G[S_1, q]}$. For the same reason, $core_{G[S_2, q]} \subseteq core_{G[S, q]}$. It directly follows the lemma.

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

**Proof.** From Definition 1, we can easily conclude that, for any specific $k$, a $k$-core has at least $k+1$ vertices. Since each vertex in a specific $k$-core has at least $k$ edges, the minimum number of edges in a $k$-core is $\frac{(k+1)k}{2}$. Consider a connected graph, which contains a $k$-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 $k$-core. The total number of edges is

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

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

**Lemma 4.** Given two keyword sets $S_1$ and $S_2$, if $G[S_1, q]$ and $G[S_2, q]$ exist, we have

$$G[S_1 \cup S_2, q] \subseteq G[S_1, q] \cap G[S_2, q].$$

**Proof.** Based on Proposition 1 and $S_1 \subseteq S_1 \cup S_2$, we have $G[S_1 \cup S_2, q] \subseteq G[S_1, q]$. For the same reason we have $G[S_1 \cup S_2, q] \subseteq G[S_2, q]$. It directly follows the lemma.

B. DETAILS OF BASIC SOLUTIONS

We present the pseudocodes of basic-g and basic-w in Algorithms 5 and 6.

The input of basic-g is a graph $G$, a query vertex $q$ and an integer $k$. It first generates a set $\Psi$, of candidate keyword sets, each of which contains a single keyword of $W(q)$ (line 2). Then, it finds the $k$-core, $C_k$, containing $q$ from the graph $G$. In the while loop (lines 4-14), it first initializes an empty set $\Phi$ (line 5), which is used to collect all the qualified keyword sets. Then for each candidate keyword set $S_i \in \Psi$, it finds $G[S_i, q]$ from $G$ by checking the keyword constraint. After that, it finds $G[S_i, q]$ from $G[S_i, q]$ (lines 7-8), and put it into $\Phi$ if $G[S_i, q]$ exists (lines 9-10). After checking all the candidate keyword sets in $\Psi$, if there are at least one qualified keyword sets in $\Phi$, it generates a new set $\Psi$ of larger candidate keyword sets by calling GENE&CAND($\Phi$) (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 LACs.

**Algorithm 5** Basic solution: basic-g

1: function Query($G$, $q$, $k$)
2:  init $\Psi$ using $W(q)$;
3:  find the $k$-core, $C_k$, containing $q$ from $G$;
4:  while true do
5:    $\Phi \leftarrow \emptyset$;
6:    for each $S_i \in \Psi$ do
7:      find $G[S_i, q]$ from $C_k$;
8:      find $G[S_i, q]$ from $G[S_i, q]$;
9:      if $G[S_i, q]$ exists then
10:        $\Phi$ add($S_i$);
11:      if $\Phi \neq \emptyset$ then
12:        $\Psi \leftarrow \text{GENECAND($\Phi$)}$;
13:      else
14:        break;
15:  output the communities of keyword sets in $\Phi$;
16: end while;
17: output the communities of keyword sets in $\Phi$;

Algorithm 6 presents the pseudocodes of basic-w. It follows the main steps of basic-g, except that for each candidate keyword set $S_i$, it finds $G[S_i, q]$ from $G$ directly, rather than from $C_k$.

**Algorithm 6** Basic solution: basic-w

1: function Query($G$, $q$, $k$)
2:  init $\Psi$ using $W(q)$;
3:  while true do
4:    $\Phi \leftarrow \emptyset$;
5:    for each $S_i \in \Psi$ do
6:      find $G[S_i, q]$ from $G$;
7:      find $G[S_i, q]$ from $G[S_i, q]$;
8:      if $G[S_i, q]$ exists then
9:        $\Phi$ add($S_i$);
10:      if $\Phi \neq \emptyset$ then
11:        $\Psi \leftarrow \text{GENECAND($\Phi$)}$;
12:      else
13:        break;
14:  output the communities of keyword sets in $\Phi$;
C. CANDIDATE GENERATION

Given a set $\Phi$ of qualified keyword sets, Algorithm 7 generates new candidate keyword sets incrementally by linking each pair of keyword sets. We first initialize $\Psi$ as an empty set (line 2). Then for each pair, $S_i$ and $S_j$, of keyword sets in $\Phi$, we sort their keywords. If they differ only at the last keyword, then we generate a new keyword set $S_i \cup S_j$, by a union operation (lines 3-6). According to Lemma 1, if any subset of $S$ appears in $\Phi$, we prune $S$; otherwise, we regard it as a candidate and add it into $\Psi$ (lines 7-8). Finally, we return $\Psi$ as the generated candidate keyword sets (line 9).

\[
\text{Algorithm 7 Generate candidate keyword sets}
\]

\begin{algorithm}
\begin{algorithmic}[1]
\Function{GENECAND}{$\Phi$}
\State $\Psi \leftarrow \emptyset$
\For{each $S_i \in \Phi$}
\For{each $S_j \in \Phi$}
\If{$S_i$ and $S_j$ differ at the last keyword}
\State initialize $S = S_i \cup S_j$
\EndIf
\EndFor
\State if $S$ cannot be pruned by Lemma 1 then
\State $\Psi$.add($S$)
\EndFor
\EndFunction
\State return $\Psi$
\end{algorithmic}
\end{algorithm}

D. ANCHORED UNION-FIND

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

\[
\text{Algorithm 8 Functions on the AUF data structure}
\]

\begin{algorithm}
\begin{algorithmic}[1]
\Function{MAKESET}{$x$}
\State $x.parent \leftarrow x$
\State $x.rank \leftarrow 0$
\State $x.anchor \leftarrow x$
\EndFunction
\Function{FIND}{$x$}
\If{$x.parent = x$}
\State $x.parent \leftarrow \text{FIND}(x.parent)$
\EndIf
\State return $x.parent$
\EndFunction
\Function{UNION}{$x$, $y$}
\State $xRoot \leftarrow \text{FIND}(x)$
\State $yRoot \leftarrow \text{FIND}(y)$
\If{$xRoot = yRoot$} return \EndIf
\If{$xRoot.rank < yRoot.rank$}
\State $xRoot.parent \leftarrow yRoot$
\EndIf
\ElseIf{$xRoot.rank > yRoot.rank$}
\State $yRoot.parent \leftarrow xRoot$
\EndIf
\State $yRoot.parent \leftarrow xRoot$
\State $xRoot.rank \leftarrow xRoot.rank + 1$
\EndFunction
\Function{UPDATEANCHOR}{$x$, $coreC[i]$, $y$}
\State $xRoot \leftarrow \text{FIND}(x)$
\If{$coreC[xRoot.anchor] > coreC[y]$}
\State $xRoot.anchor \leftarrow y$
\EndIf
\EndFunction
\end{algorithmic}
\end{algorithm}

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

E. DETAILS OF THE ADVANCED METHOD

Algorithm 9 presents the advanced method. Similar with basic method, we first conduct $k$-decomposition (line 2). Then, for each 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 $p$, for each component (lines 10-11). For each vertex $v \in \{C_i - V'\}$, we add a pair $(v, p_v)$ to the map (lines 12-13). Then for each of $v$’s neighbor, $u$, if its core number is at least $coreC[v]$, we link $u$ and $v$ together in the AUF by a $\text{union}$ operation (lines 14-16), and find $p_v$’s child nodes using the anchor of the AUF tree (lines 17-21). After vertex $v$ has been added into the CL-tree, we update the anchor (lines 22-24). Then we move to the upper level in next loop (line 25). After the while loop, we build the root node of the CL-tree (line 26). Finally, we build the inverted list for each tree node and obtain the built index (lines 27-28).

Complexity analysis. In Algorithm 9, lines 1-3 can be completed in $O(m \cdot n)$ (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 $\alpha(n)$ is less than $5$ for all practical values of $n$. In function $\text{makeSet}$, since initializing $x.anchor$ can be done in $O(1)$, the time complexity of $\text{makeSet}$ is still $O(1)$. In function $\text{updateAnchor}$, as $\text{FIND}$ can be completed in $O(\alpha(n))$ and updating anchor can be completed in $O(1)$, the total time cost of function $\text{updateAnchor}$ is $O(\alpha(n))$. 

the inverse Ackermann function. The keyword inverted lists of all the tree nodes can be computed in $O(n \cdot \log n)$. Therefore, the CL-tree can be built in $O(n \cdot \log n \cdot \log \log n)$. The space cost is $O(n \cdot \log n \cdot \log \log n)$, as maintaining an AUP takes $O(n)$.

**F. VARIANTS OF LAC SEARCH**

In what follows, we formally define three typical variants in Section F.1. Then we extend previous algorithms to answer the variants in Section F.2, and present the experimental results finally in Section F.3.

**F.1 Variant Definitions**

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

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

In Variant 2, the keyword cohesiveness is relaxed. This can be applied for cases when the keyword information is weak.

**Variant 3.** Given a graph $G$, a positive integer $k$, a vertex $q \in V$ and a predefined keyword set $S$, and a threshold $\theta \in [0,1]$, return a subgraph $G_q$, the following properties hold:

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

Basically, Variant 3 considers the constraints introduced in Variants 1 and 2. We illustrate all the variants in Example 7.

**Example 7.** Consider Figure 3(a). Let $q=A$ and $k=2$.

For Variant 1, if the predefined keyword set is $\{x\}$, then the vertex set $\{A, B, C, D\}$ forms the target LAC. For Variant 2, if the predefined keyword set is $\{x, y\}$ and the threshold is 50%, then the vertex set $\{A, B, C, D, E\}$ forms the target LAC. For Variant 3, if the predefined keyword set is $\{x, y\}$, then the vertex set $\{A, C, D\}$ forms the target LAC.

**F.2 Algorithms of Variants**

**1. Variant 1.** In line with Problem 1, we first introduce the basic solutions without index, which are extended naturally from basic-g and basic-w, and are denoted by basic-g-v1 and basic-w-v1 respectively. Their details pseudocodes are presented in Algorithms 10 and 11.
Variant 1, we can only obtain a community for $S_1$ (see Figure 15(a)), and no communities for $S_2$. Note that the captions indicate the shared keywords of the communities. While for Variant 3, we can obtain two communities (see Figures 15(a) and 15(b)). This implies that, for Variant 3, although we cannot find a community with label being exactly $S_2$, we still can find a community with label being \{cube, information\}, which shares a large portion of the input query keywords.

3. effect of $\theta$ in Variant 2. For each query vertex, we randomly select 10 keywords to form set $S$, set $\theta$ as 0.2, 0.4, 0.6, 0.8 and 1.0, and answer the query of Variant 2 using basic-g-v2, basic-w-v2 and SWT. Figures 14(e)-14(h) show their efficiency. Similar with Variant 1, we can see that SWT performs the best, as it uses the CL-tree index.

4. effect of $|S|$ in Variant 3. Similar with Variant 1, we consider query vertices having at least 9 keywords. For each of them, we randomly select 3, 5, 7 and 9 keywords to form the query keyword sets, and answer the query of Variant 3 using basic-g-v3, basic-w-v3 and Dec-v3. Figures 14(i)-14(l) show their efficiency. Clearly, Dec-v3 performs 1 to 3 order-of-magnitude faster than basic-g-v3 and basic-w-v3 on all the datasets.

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

Figure 15: Jiawei Han’s communities on Variants 1 and 3.

2. effect of $|S|$ in Variant 1. We consider query vertices having at least 9 keywords. For each of them, we randomly select 1, 3, 5, 7 and 9 keywords to form the query keyword sets, and answer the query of Variant 1 using basic-g-v1, basic-w-v1 and SWT. Figures 14(a)-14(d) show their efficiency. We can see that SWT outperforms the basic solutions consistently. Also, the performance gap increases between SWT and basic solutions as $|S|$ increases. This is because it uses the CL-tree index.