Exploring Communities in Large Profiled Graphs

Yankai Chen, Yixiang Fang, Reynold Cheng, *Member, IEEE*, Yun Li, Xiaojun Chen, and Jie Zhang

Abstract—Given a graph G and a vertex $q \in G$, the community search (CS) problem aims to efficiently find a subgraph of G whose vertices are closely related to q. Communities are prevalent in social and biological networks, and can be used in product advertisement and social event recommendation. In this paper, we study *profiled community search* (PCS), where CS is performed on a *profiled graph*. This is a graph in which each vertex has labels arranged in a hierarchical manner. Extensive experiments show that PCS can identify communities with themes that are common to their vertices, and is more effective than existing CS approaches. As a naive solution for PCS is highly expensive, we have also developed a tree index, which facilitates efficient and online solutions for PCS.

Index Terms—Community search, social networks, graph queries, profiled graph

1 Introduction

Given a graph G and a query vertex $q \in G$, the goal of *community search* (CS) is to extract *communities*, or densely connected subgraphs of G that contain q, in an online manner. Communities, which are often found in large graphs such as social media and biological networks, can be used in various applications, such as social event setting, friend recommendation, and research collaboration analysis [1], [2], [3], [4], [5].

In this paper, we investigate the CS problem for a *profiled graph*. This is essentially a kind of *attributed graphs*, where each graph vertex is associated with a set of labels arranged in a hierarchical manner called a *P-tree*. Fig. 1a shows a profiled graph, which is a computer science coll'aboration network; each vertex represents a researcher, and a link between two vertices depicts that the two corresponding researchers have worked together before. Each vertex is associated with a P-tree, which describes the expertise of researchers. Fig. 1c shows the meanings of the terms in each P-tree, following the *ACM Computing Classification System (CCS)*[1], which is partially presented in Fig. 1b. For instance, vertex B denotes a researcher, whose research domain is in computing methodology (*CM*), with specific interest in machine learning (*ML*) and artificial intelligence (*AI*). Profiled graphs are informative and can be found in various graph applications (e.g., knowledge bases, social and collaboration networks). Moreover, the P-trees of profiled graphs systematically organize labels relat'ed to a vertex (e.g., hierarchical and interrelated knowledge in knowledge bases, affiliation, expertise, and locations in social and collaboration networks), reflecting the semantic relationship among them. For example, in a P-tree, label "London" can be a child node of "UK", because London is a UK city.

To our best knowledge, previous CS algorithms are not designed for profiled graphs. Early solutions (e.g., [1], [2], [3]) often only consider graph topology (e.g., a k-core is a community such that each vertex is connected to k or more vertexes). However, they did not consider the use of vertex labels. As pointed out in [4], the communities returned by those solutions are often huge (e.g., a community can easily contain over 1,000 vertices). Moreover, the vertices included in the communities were not quite related. Recent works, such as ACQ [4] and ATC [5], propose to use both graph structure and vertex label information. While these works have been shown to be more effective than CS solutions that do not utilize vertex labels, they did not employ the hierarchical relationship among labels (e.g., P-trees in Fig. 1a). This may lead to suboptimal results. In Fig. 1a, suppose that a renowned expert D wants to organize an academic seminar. Based on the ACQ solution [4], with k=2, only a 2-core is searched (Fig. 2b), whose vertices $\{B, C, D\}$ have several labels (i.e., *r*, *CM*, *ML*, *AI*) in common. However, it fails to return the community in Fig. 2c, whose vertices are also highly similar. For these two communities, the shared labels as well as their relationships in the P-tree are very different. Therefore, both communities can be presented to the organizer for further selection.

In this paper, we study *profiled community search* (PCS), which aims to find *profiled communities*, or PC's, for a profiled graph. In a profiled graph, each vertex is associated with a P-tree. Conceptually, a PC is a group of densely connected vertices, whose P-trees have the largest degree of overlap. This overlapping part is the largest common subtree shared by all the vertices. Fig. 2 illustrates two PC's in the profiled graph of Fig. 1, namely $\{B, C, D\}$ and $\{A, D, E\}$, as well as their largest common subtrees. For example, in Fig. 2c, vertices A, D, and E all possess the subtree with root r and leaf nodes *IS* and *DMS*. Notice that these three vertices also form a 2-core of D, and the common subtree among them is the largest. The common subtree reflects the "theme" of the community. In the PC of Fig. 2b, all the researchers involved share interest in machine learning and artificial intelligence, whereas for Fig. 2c, the researchers are all interested in data management system.

Contributions. As we will explain, a simple solution to solve the PCS problem is extremely expensive. To improve the efficiency of finding PC's (so that they can be used in online applications), we first introduce an *anti-monotonicity* property, which allows the candidates for a PC to be pruned efficiently. We further develop the *CP-tree* index, which systematically organizes the graph vertices and P-trees of a profiled graph. The CP-tree enables the development of two fast PC discovery algorithms. We experimentally evaluate our solutions on two real large profiled graphs and one synthetic profiled graph. Our results show that PC's are better representations of communities, and the CP-tree based algorithms are up to 4 order-of-magnitude faster than basic solution.

Organization. We review the related work in Section 2. Section 3 presents the PCS problem and a basic solution. Section 4 discusses the CP-tree and its related solutions. We report the experimental results in Section 5, and conclude in Section 6.

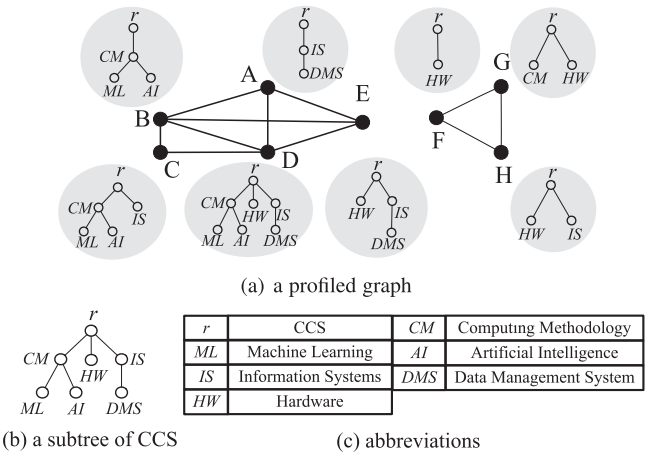
2 Related Work

In the literature, there are two kinds of work related to the retrieval of communities, namely *community detection* (CD) and *community search* (CS).

1. ACM CCS: http://www.acm.org/publications/class-2012

- *Y. Chen, Y. Fang, and R. Cheng are with the Department of Computer Science, The University of Hong Kong, Hong Kong. E-mail: {ykchen, yxfang, ckcheng}@cs.hku.hk.*
- *Y. Li is with the Department of Computer Science and Technology, Nanjing University, Nanjing, Jiangsu 210008, China. E-mail: liycser@gmail.com.*
- *X. Chen is with the College of Computer Science and Software, Shenzhen University, Shenzhen, Guangdong 518060, China. E-mail: xjchen@szu.edu.cn.*
- *J. Zhang is with the School of Computer Science and Engineering, Nanyang Technological University, Singapore 639798. E-mail: zhangj@ntu.edu.sg.*

Manuscript received 21 Mar. 2018; revised 13 Nov. 2018; accepted 15 Nov. 2018. Date of publication 22 Nov. 2018; date of current version 3 July 2019.
(Corresponding author: Yixiang Fang.)
Recommended for acceptance by L. B. Holder.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TKDE.2018.2882837



(a) a profiled graph

(b) a subtree of CCS

r	CCS	*CM*	Computing Methodology
ML	Machine Learning	*AI*	Artificial Intelligence
IS	Information Systems	*DMS*	Data Management System
HW	Hardware		

(c) abbreviations

Fig. 1. A profiled graph, a subtree of CCS, and meanings of terms.

Community Detection (CD) aims to obtain all the communities from a given graph. Recent works [6], [7] use clustering techniques or topic models to obtain communities from attributed graphs. However, these studies often assume that the attribute of the vertex is a set of keywords, and do not consider the hierarchical relationship among them. In addition, CD solutions are typically time consuming, and they may not be suitable for online applications that require fast retrieval of communities. It is also interesting to examine how our PCS solutions can be extended to support CD.

Community Search (CS) returns communities for a given vertex in a fast and online manner. Most existing CS solutions [1], [2], [3], [16], [8], [9] only consider graph topologies, but not the labels associated with the vertices. Recent CS solutions, such as ACQ [4], [10], [17] and ATC [5], make use of both vertex labels and graph structure to find communities. However, they are not designed for profiled graphs, and do not consider the hierarchical relationship among vertex labels. We have performed detailed experiments on real datasets (Section 5). We show that our algorithms yield better communities than state-of-the-art CS solutions.

3 Problem Definition and Basic Solution

In this section, we first formally introduce the PCS problem, and then give a basic solution to the PCS problem.

3.1 The PCS Problem

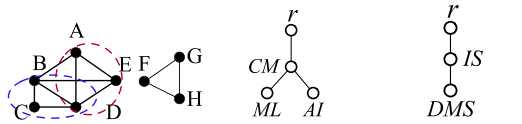
A profiled graph $G(V, E)$ is an undirected graph with vertex set V and edge set E. Each vertex $v \in V$ is associated with a *profiled tree* (P-tree) to describe v's hierarchical attributes.

Definition 1 (P-tree). *The P-tree of vertex q, denoted by $T(q)$=$(V_{T(q)}, E_{T(q)})$, is a rooted ordered tree, where $V_{T(q)}$ is the set of attribute labels and $E_{T(q)}$ is the set of edges between labels. A P-tree satisfies following constraints: (1) There is only one root node $r \in V_{T(q)}$; (2) $\forall (x, y) \in E_{T(q)}$, it is directed and y is the child attribute label of x; and (3) $\forall y \in V_{T(q)}$ and $y \neq r$, there is one and only one $x \in V_{T(q)}$, s.t. $(x, y) \in E_{T(q)}$.*

In practice, labels in the upper levels of the P-tree are more semantically general than those in lower levels. All edges in $E_{T(q)}$ preserve the semantic relationships among labels in $V_{T(q)}$.

Definition 2 (Induced Rooted Subtree). *Given two P-trees S=(V_S, E_S) and T=(V_T, E_T), S is the induced rooted subtree of T, denoted by $S \subseteq T$, if $V_S \subseteq V_T$ and $E_S \subseteq E_T$.*

Essentially, an induced rooted subtree defines an inclusion relationship between two P-trees. Unless otherwise specified, we use "subtree" to mean "induced rooted subtree". We call the unified P-tree of all vertices' P-trees a *Global P-tree* (GP-tree), which usually corresponds to a taxonomy system in practice.



(a) two PC's (b) {B, C, D} (c) {A, D, E}

Fig. 2. Illustrating profiled community search (PCS).

Definition 3 (Maximal Common Subtree). *Given a profiled graph G, the maximal common subtree of G, denoted by $\mathcal{M}(G)$, holds the properties: (1) $\forall v \in G$, $\mathcal{M}(G) \subseteq T(v)$; (2) there exists no other common subtree $\mathcal{M}'(G)$ such that $\mathcal{M}(G) \subseteq \mathcal{M}'(G)$.*

The common subtree depicts the common hierarchical part among all P-trees in a subgraph. To further adequately depict the common profile shared by all vertices, we define the maximal structure $\mathcal{M}(G)$, which can sufficiently find commonalities among diverse interests of users. As a result, we can maximize vertices' common profiles, including the topology and semantics of users' profiles. Next, we formally introduce the PCS problem.

Problem 1 (PCS). Given a profiled graph $G(V, E)$, a positive integer k, and a query node $q \in G$, find a set $\mathcal{G}$ of graphs, such that $\forall G_q \in \mathcal{G}$, the following properties hold:

- *Connectivity.* $G_q \subseteq G$ is connected and contains q;
- *Structure cohesiveness.* $\forall v \in G_q$, $deg_{G_q}(v) \geq k$, where $deg_{G_q}(v)$ denotes the degree of v in G_q;
- *Profile cohesiveness.* There exists no other $G'_q \subseteq G$ satisfying the above two constraints, such that $\mathcal{M}(G_q) \subseteq \mathcal{M}(G'_q)$.
- *Maximal structure.* There exists no other G'_q satisfying the above properties, such that $G_q \subset G'_q$ and $\mathcal{M}(G_q) = \mathcal{M}(G'_q)$;

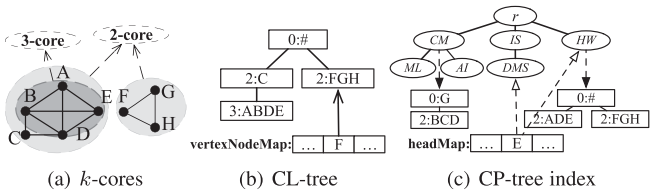
Essentially, a profiled community (PC) is a subgraph of G, in which vertices are closely related in both structure and semantics. In Problem 1, the first two properties and last property ensure the structure cohesiveness, as shown in the literature [8], [10]. The unique property *profile cohesiveness* captures the maximal shared profile among all the vertices of G_q. Moreover, since the shared subtree $\mathcal{M}(G_q)$ shows the common hierarchical attribute, it can well explain the semantic theme of the community.

3.2 A Basic Solution

Since vertices in the PC's share a common subtree of the query vertex q, a straightforward method it that we can enumerate all the subtrees of q's P-tree and find the corresponding PC's. However, as illustrated in Lemma 1, the search space may be exponentially large and computation overhead renders it impractical. To alleviate this issue, we iteratively perform the following two steps. All proof of lemmas can be found in our supplemental materials.

Lemma 1. *The maximum number of subtrees of a P-tree with x nodes is $2^{x-1} + 1$.*

Step 1: Candidate Subtree Generation. To generate the candidate subtrees, the key problem is how to avoid redundancies of the subtree enumeration. In [11], Asai et al. introduced a tree pattern enumeration strategy, and it is based on the following two concepts: (1) *Rightmost leaf* is the last P-tree node according to the depth-first traversal order. (2) *Rightmost path* is defined as a path from the root node to the rightmost leaf. Given a tree T', a new subtree T can only be generated by adding a new node t to T' such that the following hold: (1) t's parent node is on the rightmost path of T'; (2) t



(a) k-cores (b) CL-tree (c) CP-tree index

Fig. 3. k-cores, CL-tree, and CP-tree index.

is the rightmost leaf of T. As shown in [11], this generation strategy guarantees that all the subtrees of the P-tree will be enumerated without repetition. Thus, we follow this strategy to generate the candidate subtrees.

Step 2: Community Verification. After a candidate subtree T has been generated, we verify the existence of the corresponding community. We use $G_k[T]$ to represent the largest connected subgraph of G containing q where each vertex has at least k neighbors and contains the subtree T. We say that, T is *feasible*, if $G_k[T]$ exists. The verification step is mainly based on the following lemma.

Lemma 2 (Anti-Monotonicity). *Given a subtree T, if $G_k[T] \neq \emptyset$, then $\forall T' \subseteq T, G_k[T'] \neq \emptyset$.*

By Lemma 2, we can conclude that, if $G_k[T]$ is infeasible, then we can stop generating subtrees from T. The `basic` method begins with generating a subtree from the root node. Then, it iteratively performs the two steps above to retrieve all the feasible $G_k[T]$s, until no larger subtrees can be generated. Pseudocodes of `basic` can be found in the supplemental materials.

Complexity Analysis. Let m be the number of edges in G. In worst case all edges are traversed to compute the $G_k[T]$ and all the subtrees are verified. As a result, `basic` completes in $O(2^{|T(q)|} \cdot m)$ time where $|T(q)|$ denotes the number of nodes of $T(q)$. In practice, the value of $2^{|T(q)|}$ could be exponentially large and this makes `basic` impractical. To alleviate this issue, we propose more efficient index-based solutions in next section.

4 Index-Based Solutions

We first introduce some preliminaries and the proposed *CP-tree* index, and then discuss the index-based query algorithms.

4.1 k-core and CL-Tree

***k*-core**. In line with existing CS [4], [8], we use k-core to satisfy the constraints of minimum degree and maximal structure of a PC. Given an integer k ($k \geq 0$), the k-core of G, denoted by G_k, is the largest subgraph of G, such that $\forall v \in G_k$, $deg_{G_k}(v) \geq k$. Since G_k may be disconnected, we use k-$\widehat{core}$s to denote one of its connected components. An important property of k-core is the "nested" property: given two integer i and j, j-$\widehat{core} \subseteq i$-$\widehat{core}$ if $i < j$. In Fig. 3a, the 0-core represents the whole graph, and 3-core is nested in 2-core. Computing all the k-cores of a graph G, known as core decomposition, can be completed by an $O(m)$ algorithm [12], where m is the number of edges in G.

CL-Tree. Since k-cores are nested, all the k-cores of a graph can be organized into a tree structure, called *CL-tree* [4]. In this paper, we adopt it, but skip the labels on the tree. The CL-tree of the graph in Fig. 3a is shown in Fig. 3b. Clearly, vertices in each CL-tree node and other vertices in all its descendant nodes represent a k-$\widehat{core}$. For example, vertex C and other vertices $\{A, B, D, E\}$ in its child node compose a 2-$\widehat{core}$. Since each vertex appears only once, the space cost of CL-tree is $O(n)$ where n is the number of vertices in G. In addition, we maintain a map *vertexNodeMap*, where the key is the vertex and the value is the node of the corresponding CL-tree node, and it allows us to locate the k-$\widehat{core}$ containing any query vertex efficiently.

4.2 CP-tree Index

Index Overview. We build the Core Profiled tree (CP-tree) index by considering both the P-tree structure and k-cores. We depict an example CP-tree in Fig. 3c using the profiled graph in Fig. 1a. Each CP-tree node corresponds to a label and stores the k-cores sharing this label. To summarize, each node p consists of following four elements: (1) *label*: the attribute label; (2) *parentNode*: the parent node of p; (3) *childList*: a list of child CP-tree nodes of p; and (4) *vertexNodeMap*: a map that stores the CL-tree. In addition, we maintain a map *headMap*, where the key is a vertex v, and the value is a list of CP-tree nodes, each of which corresponds to a leaf node of v's P-tree. Main advantages of CP-tree are listed below.

- *Restore P-trees.* By utilizing the *headMap*, each vertex's P-tree can be restored by traversing the leaf nodes up to the root node.
- *Locating* ***k*-$\widehat{core}$.** Given an integer k, a query vertex q and a CP-tree node t, using *vertexNodeMap*, we design a function $get(k, q, t)$ to get the k-$\widehat{core}$ containing q where each vertex contains the label $t.label$ in constant time cost.
- *Query efficiency.* As discussed above, the label information of each vertex's P-tree can be efficiently accessed using the *headMap*.

Index Construction. We incrementally create CP-tree nodes and then link them up to build the CP-tree index. For each vertex v, we read $T(v)$ and create new CP-tree nodes (lines 2-5). For each CP-tree node t, we add v in t for later CL-tree construction (lines 6, 9). If P-tree node x is a leaf node, we update $headMap$ (line 7). Then we link up all CP-tree nodes following the GP-tree structure. If GP-tree is unknown, we can simultaneously unify it while reading P-trees(line 10). Finally, $\mathcal{I}$ is returned (line 11).

Algorithm 1. CP-Tree Index Construction

```
1:  function buildIndex(G(V, E))
2:    for each v ∈ V do
3:      for each x ∈ T(v) do
4:        t ← a CP-tree node in I such that t.label = x.label;
5:        if t = null then create a CP-tree node t and add it in I;
6:        add v in t;
7:        if x is the leaf node of T(v) then headMap.put(v, t);
8:    for each t ∈ I do
9:      Build CL-tree for the subgraph of t;
10:     link to its parent and child nodes;
11:   return I;
```

Complexity Analysis. Obviously, lines 2-7 take the linear time. The time complexity of building a CL-tree is $O(m \cdot \alpha(n))$ [4], [10] where m is the number of edges in G and $\alpha(n)$, the inverse Ackermann function, is less than 5 for large value of n. Thus the time complexity of building CP-tree is $O(|P| \cdot m \cdot \alpha(n))$, and it is linear to the size of G. The space cost of CP-tree is $O(|P| \cdot n)$ where $|P|$ denotes the number of labels in G. The space cost of the *headMap* is $O(\hat{l} \cdot n)$ where $\hat{l}$ denotes the average number of leaf nodes in each vertex's P-tree and $\hat{l} < |P|$. Therefore, the total space complexity is $O(|P| \cdot n)$ which is linear to the size of G.

4.3 Index-Based Query Algorithms

Now we present our index-based query solutions. The first one follows the framework of `basic`, and it incrementally generates and verifies the subtrees of P-tree (from smaller subtrees to larger ones). Thus we call it `incre`. The advanced methods borrows some ideas from MARGIN [13], the algorithm of mining *maximal frequent subgraphs*. As we will explain later, advanced methods can find all PC's by examining a small fraction of subtrees, resulting in high efficiency. In addition, their time complexities are $O(2^{|T(q)|} \cdot m)$, because in the worst case all the subtrees are verified. However, as we will

show in Section 5.3, in practice they are much more efficient than such worse-case time complexities.

4.3.1 The Method `incre`

We begin with an interesting lemma, which greatly accelerates the verification step.

Lemma 3. *Given a CP-tree index $\mathcal{I}$, a subtree T' and a new subtree T which is generated from T' by adding a new P-tree node. We have $G_k[T] \subseteq G_k[T'] \cap \mathcal{I}.get(k, q, T \backslash T')$, where $T \backslash T'$ denotes the new added node.*

As `incre` searches the communities in the subgraph which are found in former iteration, the query efficiency is improved. We present `incre` in Algorithm 2.

Algorithm 2. `incre` Query Algorithm

```
1: function QUERY(I, q, k)
2:   restore T(q) using I.headMap;
3:   G ← ∅, Ψ ← GENERATESUBTREE(∅, T(q));
4:   while Ψ ≠ ∅ do
5:     T' ← Ψ.pop(); flag ← true;
6:     Φ ← GENERATESUBTREE(T', T(q));
7:     for each T ∈ Φ do
8:       compute G_k[T] from G_k[T'] ∩ I.get(k, q, T\T');
9:       if G_k[T] ≠ ∅ then
10:        flag ← false; Ψ.push(T);
11:    if flag = true and T' is maximal then G = G ∪ G_k[T'];
12:  return G;
```

We first use *headMap* to locate the leaf nodes of $T(q)$ and then restore $T(q)$ (line 2). We initialize Ψ by using $T(q)$ (line 3). In the iteration, for current subtree T', we generate new subtrees. For each new subtree T, we verify the existence of $G_k[T]$ using the index (lines 4-8). If $G_k[T]$ exists, we add T in Φ (lines 9-10); otherwise if no subtree can be generated from T' or all subtrees generated from T' are infeasible, we add $G_k[T']$ in $\mathcal{G}$ if T' is maximal (line 11). Finally, all PC's are returned (line 12).

4.3.2 The `advanced` Methods

The method `incre` follows the Apriori-based method, which explores all possible subtrees by traversing the search space from smaller subtrees to larger ones; while, as demonstrated in the supplemental materials, the maximal feasible subtrees often lie in the middle of the search space, which implies that most of the exploration may be avoided. Based on this observation, we adapt MARGIN [13] to tackle PCS.

MARGIN. It does not perform a bottom-up (or top-down) traversal of the search space; instead, it narrows the search space by examining only subgraphs that lie on the border of frequent and infrequent subgraphs. It firstly finds an initial pair of graphs (CR, R) where R is frequent and CR is not. CR is the subgraph of R and they differ by exactly one edge. (CR, R) is called a `cut` and from this `cut`, MARGIN expands to new adjacent subgraphs by adding or deleting an edge and finds all other `cuts`. MARGIN defines this function as `expandCut` and proves that `expandCut` is able to find all maximal frequent subgraphs. Inspired by MARGIN, we design the following functions.

1. Function `expandPtree`. This function is adapted from `expandCut` [13] and the main modifications are as follows. Pseudocodes can be found in the supplemental materials.

- We dynamically obtain child subgraphs and parent sugraphs, which are called *child subtrees* and *parent subtrees* in our case, using the *parentNodes* and *childLists* of CP-tree nodes, instead of pre-computing all subtrees in the search space as MARGIN does.
- We define a pair of P-trees (IF, F) as a cut, where IF is the child subtree of F and F is feasible while IF is not;
- We dynamically verify whether a feasible subtree is maximal.
- We develop a function `verifyPTree` to verify the feasibility.

Lemma 4. *Given a P-tree pair (IF, F), `expandPtree` can find all feasible subtrees for a PCS query.*

2. Function `verifyPtree` . Given a subtree T, T_{child} and T_{parent} denote a child and the parent subtree of T. Let l denote the number of T_{parent}'s leaf nodes and t_{n_i} represent the ith leaf node of T_{parent}. Derived from Lemma 3, we have

- $G_k[T_{child}] \subseteq G_k[T] \cap \mathcal{I}.get(k, q, T_{child} \setminus T)$.
- $G_k[T_{parent}] \subseteq \bigcap_{i=1}^{l} \mathcal{I}.get(k, q, t_{n_i})$.

Since all P-trees are subtrees of the GP-tree, if a P-tree has the attribute t, then t's parent attribute t' is also included. Thus, $\mathcal{I}.get(k, q, t) \subseteq \mathcal{I}.get(k, q, t')$. For a special subtree T_i (a path from leaf node t_{n_i} to root node r), we can finally get $G_k[T_i] = \mathcal{I}.get(k, q, t_{n_i})$. Note that T_{parent} can be seen as several paths and thus we get $G_k[T_{parent}] \subseteq \bigcap_{i=1}^{l} \mathcal{I}.get(k, q, t_{n_i})$.

Based on CP-tree, `verifyPtree` can efficiently verify subtrees. Next we discuss three methods to find the initial cut.

3. Function `find-I`. We can adapt `incre` to find the initial cut. Once we find a subtree which is feasible while its child subtree is not, then we can regard them as an initial cut.

4. Function `find-D`. We can decrementally generate subtrees from larger subtrees to smaller ones. In each step, for an infeasible subtree T, we remove one of T's leaf nodes and verify the feasibility of the new subtrees. Once there is a new feasible subtree, we treat T and this new subtree as the initial cut.

5. Function `find-P`. We can find the initial cut by directly verifying subtrees instead of the node one by one. Intuitively, P-tree can be divided into several paths (from leaf nodes to the root). According to Lemma 2, these paths can be further verified by checking the corresponding leaf nodes. We call it find initial cut by path (`find-P`). Due to the page limit, we present the pseudocodes of `find-P` in Algorithm 3.

Algorithm 3. Find the Initial Cut: `Find-P`

```
1: function FIND-P(I, S, q, k)
2:   IF ← ∅; F ← find a leaf node t ∈ S s.t. I.get(k, q, t) ≠ ∅;
3:   if F ≠ ∅ then
4:     for each t ∈ S do
5:       computing G_k[F ∪ t] from G_k[F] ∩ I.get(k, q, t);
6:       if G_k[F ∪ t] ≠ ∅ then F = F ∪ t;
7:       else
8:         path ← trace a path from t to r in I;
9:         find t', t'_parent on path s.t. G_k[t']=∅, G_k[t'_parent] ≠ ∅;
10:        IF = F ∪ t'_parent; F = F ∪ t';
11:        Break;
12:  else
13:    for each t ∈ S do S.replace(t, t.parent);
14:    FIND-P(I, S, q, k);
15:  complete subtrees IF, F;
16:  return (IF, F);
```

S denotes a P-tree node set. Initially, it consists of all leaf nodes of $T(q)$. If there does not exist a feasible node in S, we trace up to verify their parent nodes (lines 13-14). Next, we iteratively check the nodes in S. If we find a node t and $G_k[F \cup t]$ exists, we update F (lines 5-6). Let t'_{parent} denote the parent node of t'. If we find a node t that $G_k[F \cup t]$ does not exist, we trace up to find the

TABLE 1
Datasets Used in Our Experiments

Dataset	Vertices	Edges	$\widehat{d}$	$\widehat{P}$	| GP-tree |
ACMDL	107,656	717,958	13.34	11.54	1,908
Flickr	581,099	4,972,274	17.11	26.63	1,908
PubMed	716,459	4,742,606	13.22	27.10	10,132

"boundary" where $G_k[t'_{parent}]$ exists while $G_k[t']$ does not and thus we find an initial pair (lines 8-11). Note that at now stage, IF, F may not be complete subtrees. Thus for the nodes in IF and F, we need to include all their ancestor nodes and then return (IF, F) as a cut (lines 15-16).

Algorithm 4. Advanced Method

1: **function** QUERY($\mathcal{I}, q, k$)
2: $\mathcal{G} \leftarrow \emptyset$;
3: $(IF, F) \leftarrow$ FIND($\mathcal{I}, S, q, k$);
4: EXPANDPTREE($IF, F, \mathcal{G}$);
5: **return** $\mathcal{G}$;

Algorithm 4 gives the overall `advanced` methods. Notice that, there are three functions, i.e., `find-I`, `find-D`, and `find-P`, of finding the initial cut, so we have three variants of `advanced`, denoted by `adv-I`, `adv-D` and `adv-P` respectively.

5 Experiments

5.1 Setup

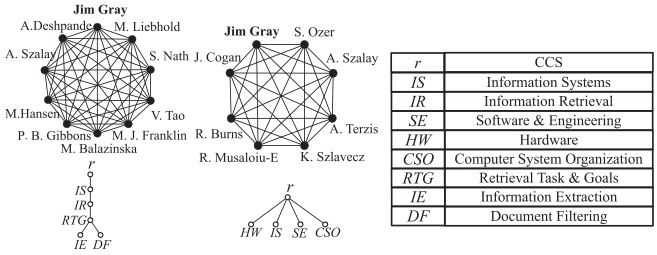
We consider two real datasets (ACMDL and PubMed) and one synthetic dataset (Flickr). *ACMDL*[2] and *PubMed*[3] are the co-authorship networks of researchers in computer science and biomedical areas respectively. Each vertex of them represents an author, and an edge is a co-authorship between two authors. For each author, her papers have been categorized by a hierarchical subject classification system (*ACM CCS* or *Medical Subject Headings (MeSH)*[4]), so we build the P-tree by unifying the categorization information of all her papers. For *Flickr*[5] [14], each vertex represents a user and each edge denotes a "follow" relationship between two users. For each user, we use a hash function and map the associated textual content to subjects of CCS to synthesize a P-tree. By doing this, the same textual contents could be mapped for constructing the same nodes in P-trees. Table 1 shows the statistics of the datasets, including the numbers of vertices and edges, vertices' average degree $\widehat{d}$, the average number of labels in P-trees $\widehat{P}$, and the average number of labels in the GP-tree.

To evaluate PCS queries, in line with [4], we set the default value of k to 6. For each dataset, we randomly select 100 query vertices from the 6-core. We implement all the algorithms in Java, and run experiments on a machine having an eight-core Intel 3.40 GHz processor, and 16GB of memory, with Ubuntu installed.

5.2 PCS Effectiveness

As mentioned before, the existing CS methods mainly focus on non-attributed graphs. A recent work ACQ [4], [10] investigates CS on attributed graphs. In ACQ, each vertex in the attributed graph is associated with a set of keywords. Communities retrieved by ACQ should satisfy the structure cohesiveness (k-core constraint) and "keyword cohesiveness" [4], [10], i.e., the number of common keywords shared by all vertices in communities should be maximum.

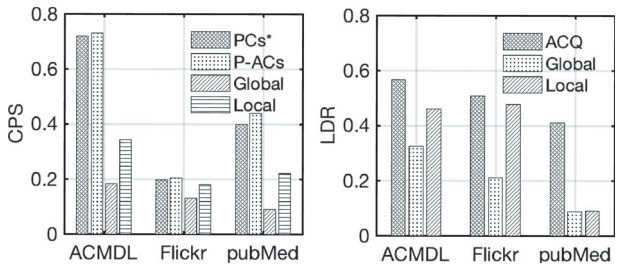
2. https://dl.acm.org/
3. https://www.nlm.nih.gov
4. https://meshb.nlm.nih.gov/
5. https://www.flickr.com/



r	CCS
IS	Information Systems
IR	Information Retrieval
SE	Software & Engineering
HW	Hardware
CSO	Computer System Organization
RTG	Retrieval Task & Goals
IE	Information Extraction
DF	Document Filtering

(a) PC1 (b) PC2 (c) abbreviations

Fig. 4. Two PC's of Jim Gray.



(a) CPS (b) LDR

Fig. 5. Comparing PCS with CS methods.

We compare PCS with ACQ. To run ACQ queries, we set each vertex's attribute as a set of keywords, which are the keywords in its P-tree. In the following, we first present a case study, and then show the quality and diversity of communities.

- *A Case Study*: We perform a case study on the ACMDL dataset and consider a renowned researcher: Jim Gray. We set k = 4 here. We present Jim's two PC's, i.e., PC1 and PC2, with different research areas in Fig. 4. Notice that ACQ only finds one community PC1 shown in Fig. 4a. This is because, ACQ maximizes the number of shared keywords, so PC2 shown in Fig. 4b, which has five shared keywords, cannot be returned. In addition, all shared keywords of PC1 are organized in a tree with few branches, which implies that the semantics of keywords are highly overlapped with each other. In contrast, the shared subtree of PC2 has multiple branches, so the semantics of keywords are very different and diversified. Hence, PCS are more effective than ACQ for extracting communities from profiled graphs.
- *Community Pairwise Similarity (CPS)*: We compare PCS with three classic CS methods using "minimum degree" definition: ACQ [4], `Global` [1] and `Local` [9]. We use Tree Edit Distance (TED) to compute the similarity between the P-trees of any pair of vertices in community G_l. Let T_i be the P-tree of the i-th vertex in G_l. The CPS is then the average similarity over all pairs of G_l's vertices, and all communities of $\mathcal{G}$:

$$CPS(\mathcal{G}) = 1 - \sum_{l=1}^{|\mathcal{G}|} \left[\frac{1}{|G_l|^2} \sum_{j=1}^{|G_l|} \sum_{i=1}^{|G_l|} \frac{TED(T_i, T_j)}{|T_i \cup T_j|} \right], \quad (1)$$

The CPS($\mathcal{G}$) value has a range of 0 and 1. The higher the value is, the more cohesive the community is. As shown in Fig 5a, PCs* denotes the communities that only PCS can search. P-ACs represents those returned by both of PCS and ACQ. P-ACs have the most P-tree nodes (i.e., keywords in ACQ definition) in common, and the fewest vertices. Thus they have the highest CPS values. Note that PCs* have a close CPS value with P-ACs which implies that these unique PC's are also of highly quaility.

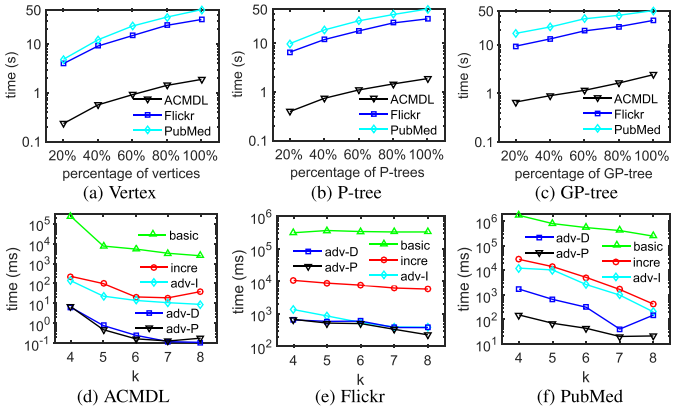


Fig. 6. Efficiency evaluation on ACMDL, Flickr, and PubMed datasets.

- *Level-diversity ratio (LDR)*: To further measure the quality of PC's, we define a metric, called *level-diversity ratio* (LDR), to measure the diversity of attributes level by level in the shared subtrees. F denotes the method that we use here to compare with PCS. Given a query vertex q, we use $\mathcal{T}(F, q, j)$ to represent the maximal common P-trees of j-th community returned by the method F. $\mathcal{L}$ is the number of levels in P-tree $T(q)$. $L_i(T)$ is the number of unique labels in the i-th level of P-tree T. $\mathcal{H}$ and $\mathcal{J}$ denote the numbers of communities returned by the method F and PCS respectively. A lower LDR value implies that the method F is less diverse than PCS.

$$LDR(q, F) = \frac{1}{\mathcal{L}} \sum_{i=1}^{\mathcal{L}} \frac{\sum_{h=1}^{\mathcal{H}} L_i\left[\mathcal{T}(F, q, h)\right]}{\sum_{j=1}^{\mathcal{J}} L_i\left[\mathcal{T}(PCS, q, j)\right]}. \quad (2)$$

Intuitively, LDR reflects the proportion of unique labels in each level. The experimental results are depicted in Fig. 5b, which shows that communities returned by ACQ can only cover 40 to 60 percent labels of PC's in each level. This implies that PC's found by PCS have higher diversity than those of ACQ, because PCS focuses on maximizing the common structure of P-trees, rather than the number of common keywords. As a result, all communities with the semantically maximal properties can be found, and the communities are of high diversity.

We evaluate the accuracy by using the ground-truth communities. The F1-scores demonstrate that, compared with other methods, PCS can stably extract communities with high accuracy over three real networks. For details, please refer to the supplemental materials.

5.3 Results of Efficiency Evaluation

In this section, we show the efficiency results of index construction and PCS queries. More results are in the supplemental materials.

1. Index Construction. For each dataset, we randomly select 20, 40, 60 and 80 percent of its vertices (or vertices' P-trees, or GP-tree) to obtain four sub-datasets respectively. Figs. 6a and 6b show the scalability of the CP-tree index construction method. Fig. 6c examines the scalability over different fractions of the GP-tree. We observe that, the time cost of the index construction is linear to the size of profiled graphs, which confirms our analysis before.

2. Query efficiency. We vary the value of k and show the query efficiency of different algorithms in Figs. 6d and 6f. The method `incre` is 100 times faster than the `basic` method, but slower than the method `adv-I`. Further, `adv-D` and `adv-P` are 10 times faster than `incre`. The reason is that, compared with `incre`, the advanced methods narrow the search space by verifying a smaller fraction of subtrees. Also, the efficiency gap in finding an initial cut results in the sightly different performance of the advanced methods. Thus, the index-based methods run fast and `adv-P` stably scales the best. Note that three advanced methods perform similarly on Flickr. This is because the initial cut results are in the middle of the search space. Thus they have similar performance even though they search from different directions.

6 Conclusions and Future Directions

In this paper, we study the PCS problem, which aims to find communities from profiled graphs in an online manner. We develop an index and some query algorithms. Our experimental results demonstrate the effectiveness of PCS and the efficiency of our solutions. In the future, we will study other structure (e.g., k-truss) and profile cohesiveness measures in the PCS definition.

Acknowledgments

Reynold Cheng, Yixiang Fang, and Yankai Chen were supported by the Research Grants Council of Hong Kong (RGC Projects HKU 106150091, 17229116, 17205115) and the University of Hong Kong (Projects 104004572, 102009508, 104004129). Xiaojun Chen was supported by NSFC under Grant no. 61773268. Jie Zhang was supported by the MOE AcRF Tier 1 funding (M4011894.020) and the Telenor-NTU Joint R&D funding. The authors thank the editors and reviewers for their insightful comments.

References

[1] M. Sozio and A. Gionis, "The community-search problem and how to plan a successful cocktail party," in *Proc. 16th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2010, pp. 939–948.
[2] W. Cui, Y. Xiao, H. Wang, Y. Lu, and W. Wang, "Online search of overlapping communities," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2013, pp. 277–288.
[3] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu, "Querying k-truss community in large and dynamic graphs," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2014, pp. 1311–1322.
[4] Y. Fang, R. Cheng, S. Luo, and J. Hu, "Effective community search for large attributed graphs," *Proc. VLDB Endowment*, vol. 9, no. 12, pp. 1233–1244, 2016.
[5] X. Huang and L. V. Lakshmanan, "Attribute-driven community search," *Proc. VLDB Endowment*, vol. 10, no. 9, pp. 949–960, 2017.
[6] Y. Ruan, D. Fuhry, and S. Parthasarathy, "Efficient community detection in large networks using content and links," in *Proc. 22nd Int. Conf. World Wide Web*, 2013, pp. 1089–1098.
[7] Z. Xu, Y. Ke, Y. Wang, H. Cheng, and J. Cheng, "A model-based approach to attributed graph clustering," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2012, pp. 505–516.
[8] R.-H. Li, L. Qin, J. X. Yu, and R. Mao, "Influential community search in large networks," *Proc. VLDB Endowment*, vol. 8, no. 5, pp. 509–520, 2015.
[9] W. Cui, Y. Xiao, H. Wang, and W. Wang, "Local search of communities in large graphs," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2014, pp. 991–1002.
[10] Y. Fang, R. Cheng, Y. Chen, S. Luo, and J. Hu, "Effective and efficient attributed community search," *VLDB J.*, vol. 26, pp. 1–26, 2017.
[11] T. Asai, K. Abe, S. Kawasoe, H. Sakamoto, H. Arimura, and S. Arikawa, "Efficient substructure discovery from large semi-structured data," *IEICE Trans. Inf. Syst.*, vol. 87, no. 12, pp. 2754–2763, 2004.
[12] V. Batagelj and M. Zaversnik, "An O(m) algorithm for cores decomposition of networks," *CoRR*, vol. cs.DS/0310049, 2003, http://arxiv.org/abs/cs.DS/0310049
[13] L. T. Thomas, S. R. Valluri, and K. Karlapalem, "Margin: Maximal frequent subgraph mining," *ACM Trans. Knowl. Discovery Data*, vol. 4, no. 3, 2010, Art. no. 10.
[14] B. Thomee, D. A. Shamma, G. Friedland, B. Elizalde, K. Ni, D. Poland, D. Borth, and L.-J. Li, "The new data and new challenges in multimedia research," *CoRR*, vol. abs/1503.01817, 2015, http://arxiv.org/abs/1503.01817
[15] J. Leskovec and J. J. Mcauley, "Learning to discover social circles in ego networks," in *Proc. 25th Int. Conf. Neural Inf. Process. Syst.*, 2012, pp. 539–547.
[16] Y. Fang, Z. Wang, R. Cheng, H. Wang, and J. Hu, "Effective and efficient community search over large directed graphs," *IEEE Trans. Knowledge Data Eng.*, 2018.
[17] Y. Fang, et al., "C-explorer: Browsing communities in large graphs," *PVLDB*, vol. 10, no. 12, pp. 1885–1888, 2017.