Influential Community Search in Large Networks

Rong-Hua Li[†], Lu Qin[‡], Jeffrey Xu Yu[*], and Rui Mao[†]

[†] *Guangdong Province Key Laboratory of Popular High Performance Computers, Shenzhen University, China*
[‡]*Centre for QCIS, FEIT, University of Technology, Sydney, Australia*
[*]*The Chinese University of Hong Kong, Hong Kong*
{rhli,yu}@se.cuhk.edu.hk; Lu.Qin@uts.edu.au; mao@szu.edu.cn

ABSTRACT

Community search is a problem of finding densely connected subgraphs that satisfy the query conditions in a network, which has attracted much attention in recent years. However, all the previous studies on community search do not consider the influence of a community. In this paper, we introduce a novel community model called k-influential community based on the concept of k-core, which can capture the influence of a community. Based on the new community model, we propose a linear-time online search algorithm to find the top-r k-influential communities in a network. To further speed up the influential community search algorithm, we devise a linear-space index structure which supports efficient search of the top-r k-influential communities in optimal time. We also propose an efficient algorithm to maintain the index when the network is frequently updated. We conduct extensive experiments on 7 real-world large networks, and the results demonstrate the efficiency and effectiveness of the proposed methods.

1. INTRODUCTION

Many real-world networks, such as social networks and biological networks, contain community structures. Discovering communities in a network is a fundamental problem in network science, which has attracted much attention in recent years [12, 26]. Another related but different problem is community search where the goal is to find the most likely community that contains the query node [22, 10]. The main difference between these two problems is that the community discovery problem is to identify all communities in a network by optimizing some pre-defined criterions [12], while the community search problem is a query-dependent variant of the community discovery problem, which aims to find the community that contains the query node [22].

In all the previous studies on these problems, a community is defined as a densely connected subgraph which ignores another important aspect, namely the influence (or importance) of a community. However, in many application domains, we are interested in finding the most influential communities. Consider the following two scenarios. Assume that Alice is a database researcher. She may want to identify the most influential research groups from the coauthorship network of the database community, so as to be aware of the recent trends of database research from those influential groups. Another frequently encountered example is in online social network domain. Suppose that Bob is an online social network user. He may want to follow the most influential groups in the social network, so as to track the recent activities from those influential groups. Both of these issues need to find the most influential communities from a network.

In this paper, we study, for the first time, the influential community search problem in large networks. To study this issue, we present a new community model called k-influential community based on the well-known concept of k-core [21]. In our definition, we model a network as an undirected graph $G = (V, E)$ where each node in G is associated with a weight, denoting the influence (or importance) of the node. A community is defined as a connected induced subgraph in which each node has degree at least k, where the parameter k measures the cohesiveness of the community. Unlike the traditional definition of k-core [21], our definition of community is not necessary the maximal induced subgraph that satisfies such a degree constraint (i.e., each node has degree at least k). The influence value of a community is defined as the minimum weight of the nodes in that community. An influential community is one that has a large influence value. We call an influential community with parameter k a k-influential community.

The intuition behind our definition is that each node in an influential community should have a large weight, indicating that every member in an influential community is an influential individual. Another possible measure of influence of a community is the average weight of all the nodes. However, this measure has a drawback that it is not robust to the outliers, because by this measure, an influential community with a high average weight may include some low-weight nodes (outliers) which are not influential. Therefore, in this paper, we use the minimum weight to measure the influence value of a community, as it captures the intuitive idea of influential community. In addition, we require that a k-influential community cannot be contained in a super-k-influential community with equivalent influence value. Because if that is the case, the latter will dominate the former on both size and influence value. Based on this novel k-influential community model, the goal of the influential community search problem is to find the top-r k-influential communities with the highest influence value in a network.

Straightforward searching the top-r k-influential communities in a large network is impractical, because there could be a large number of communities that satisfy the degree constraint, and for each community, we need to check whether it is a k-influential community. By an in-depth analysis of the structure of k-influential communities, we discover that all the k-influential communities can be obtained by iteratively deleting the smallest-weight node of the maximal k-core. Based on this finding, we propose a depth-first-search (DFS) based algorithm to search the top-r k-influential communities online. We show that the DFS-based algorithm consumes linear time and space with respect to (w.r.t.) the graph size.

For very large graphs, however, the linear-time DFS-based algorithm is still inefficient. To further accelerate the influential com-

Articles from this volume were invited to present their results at the 41st International Conference on Very Large Data Bases, August 31st - September 4th 2015, Kohala Coast, Hawaii.
Proceedings of the VLDB Endowment, Vol. 8, No. 5
Copyright 2015 VLDB Endowment 2150-8097/15/01.

munity search algorithm, we design a novel index structure, called ICP-Index, to index all the pre-calculated k-influential communities. The ICP-Index preserves all the k-influential communities, and it takes only linear space w.r.t. the graph size. Based on the ICP-Index, the query of the top-r k-influential communities can be computed in linear time w.r.t. the answer size only, thus it is optimal. To construct the ICP-Index, we devise an efficient algorithm that takes $O(\rho m)$ time and $O(m)$ space, where ρ and m denote the arboricity [7] and the number of edges of a graph, respectively. Note that the arboricity of a graph is no larger than $O(\sqrt{m})$ even in the worst case [7], and it has shown to be much smaller than the worst case bound in many real-world sparse graphs [17, 14]. In addition, we also propose an efficient algorithm to incrementally maintain the index when the graph is frequently updated.

We conduct extensive experiments over six web-scale real-world graphs to evaluate the efficiency of the proposed algorithms. The results show that the ICP-Index-based algorithm is several orders of magnitude faster than the DFS-based online search algorithm. The query time of the ICP-Index-based algorithm is from one millisecond for small k and r to a few seconds for large k and r in four large graphs with more than one billion edges. Moreover, the results show that the ICP-Index is compact and can be constructed efficiently. The results also indicate that the proposed index maintenance algorithm is very efficient which is at least four orders of magnitude faster than the baseline algorithm in large graphs. In addition, we also conduct comprehensive case studies on a coauthorship network to evaluate the effectiveness of the k-influential community model. The results demonstrate that using our community model is capable of finding meaningful influential communities in a network, which can not be identified by using the k-truss community model [15].

The rest of this paper is organized as follows. We formulate the influential community search problem in Section 2. The DFS-based algorithm is presented in Section 3. We design a new ICP-Index and propose two index construction algorithms in Section 4. We devise an efficient index update algorithm in Section 5. Extensive experiments are reported in Section 6. We review related work and conclude this paper in Section 7 and Section 8 respectively.

2. PROBLEM STATEMENT

Consider an undirected graph $G = (V, E)$, where V and E denote the node set and edge set respectively. Denote by $n = |V|$ the number of nodes, and by $m = |E|$ the number of edges in G. Let $d(v, G)$ be the degree of a node v in graph G. A graph $H = (V_H, E_H)$ is an induced subgraph of G if $V_H \subseteq V$ and $E_H = \{(u, v)|u, v \in V_H, (u, v) \in E\}$. In this paper, we refer to an induced subgraph H such that each node v in H has degree at least k (i.e., $d(v, H) \geq k$) as a k-core. The maximal k-core H' is a k-core that no super graph H of H' is also a k-core. Note that the maximal k-core of a graph G is unique and can be a disconnected graph. For a node $u \in V$, the core number of u, denoted by c_u, is the maximal k value such that a k-core contains u.

In the literature, the maximal k-core is widely used to represent cohesive communities of a graph [2, 22, 4, 16]. Instead of general cohesive communities, in this work, we seek to find influential communities in a graph. Specifically, in our setting, each node u in G has a weight w_u (such as PageRank or any other user-defined attributes), indicating the influence (or importance) of u. Additionally, we assume without loss of generality that the weight vector $W = (w_1, w_2, \cdots, w_n)$ forms a total order, i.e., for any two nodes v_i and v_j, if $i \neq j$, then $w_i \neq w_j$. Note that if that is not the case, we use the node identity to break ties for any $w_i = w_j$. Before proceeding further, we give the definition of influence value of an induced subgraph as follows.

DEFINITION 1. *Given an undirected graph $G = (V, E)$ and an induced subgraph $H = (V_H, E_H)$ of G, the influence value of H denoted by $f(H)$ is defined as the minimum weight of the nodes in H, i.e., $f(H) = \min_{u \in V_H}\{w_u\}$.*

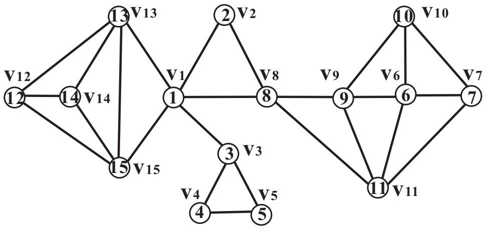


Figure 1: Running example (the numbers denote the weights)

By Definition 1, if the influence value of an induced subgraph H (i.e., $f(H)$) is large, then the weight of every node in H should be a large value, indicating that H is an influential subgraph. Below, we give a brief discussion of why we define f as the minimum weight of the nodes. Regarding the choice of $f(H)$, we need to consider functions that capture the influence (weight) of nodes in H. Moreover, we want the influence value $f(H)$ to be a large value if the induced subgraph H is influential. Except the minimum weight of the nodes in H, one potential definition of $f(H)$ is the average weight of the nodes in H, i.e., $f(H) = \sum_{u \in V_H} w_u/|V_H|$. However, this definition has a drawback that it is not robust to outliers. Specifically, by this definition, an influential subgraph may include low-weight nodes (outliers), albeit it has a high average weight. Therefore, we focus on $f(H)$ that is defined to be the minimum weight of nodes in H, which is robust to such low-weight nodes.

Intuitively, an influential community should not only have a large influence value, but it is also a cohesive induced subgraph. Based on this intuition, we give the definition of k-influential community, where the parameter k controls the cohesiveness of the community.

DEFINITION 2. *Given an undirected graph $G = (V, E)$ and an integer k. A k-influential community is an induced subgraph $H^k = (V_H^k, E_H^k)$ of G that meets all the following constraints.*

- *Connectivity: H^k is connected;*
- *Cohesiveness: each node u in H^k has degree at least k;*
- *Maximal structure: there is no other induced subgraph $\tilde{H}$ such that (1) $\tilde{H}$ satisfies connectivity and cohesiveness constraints, (2) $\tilde{H}$ contains H^k, and (3) $f(\tilde{H}) = f(H^k)$.*

Clearly, the *cohesiveness* constraint indicates that the k-influential community is a k-core. With the *connectivity* and *cohesiveness* constraints, we can ensure that the k-influential community is a connected and cohesive subgraph. And with the *maximal structure* constraint, we can guarantee that any k-influential community cannot be contained in a super-k-influential community with equivalent influence value. Because if that is the case, the latter will dominate the former on both size and influence value. The following example illustrates the definition of k-influential community.

EXAMPLE 1. *Consider the graph shown in Fig. 1. Suppose, for instance, that $k = 2$, then by definition the subgraph induced by node set $\{v_{12}, v_{13}, v_{14}, v_{15}\}$ is a 2-influential community with influence value 12, as it meets all the constraints in Definition 2. Note that the subgraph induced by node set $\{v_{12}, v_{14}, v_{15}\}$ is not a 2-influential community. This is because it is contained in a 2-influential community induced by node set $\{v_{12}, v_{13}, v_{14}, v_{15}\}$ whose influence value equals its influence value, thus fail to satisfy the maximal structure constraint.*

In many practical applications, we are typically interested in the most influential communities whose influence values are larger than those of all other influential communities. In this paper, we aim to find such communities in a large graph efficiently. Below, we formulate two influential community search problems.

Problem 1. Given a graph $G = (V, E)$, a weight vector W, and two parameters k and r, the problem is to find the top-r k-influential communities with the highest influence value.

For Problem 1, a k-influential community may be contained in another k-influential community in the top-r results. For example, in Fig. 1, we can easily verify that the top-2 2-influential communities are the subgraphs induced by $\{v_{13}, v_{14}, v_{15}\}$ and $\{v_{12}, v_{13}, v_{14}, v_{15}\}$, respectively. Clearly, in this example, the second 2-influential community contains the first 2-influential community. To avoid the inclusion relationships in the top-r results, in the following, we consider a problem of finding the top-r non-contained k-influential communities.

DEFINITION 3. *Given a graph $G = (V, E)$ and an integer k. A non-contained k-influential community $H^k = (V_H^k, E_H^k)$ is a k-influential community that meets the following constraint.*

- *Non-containment: H^k cannot contain a k-influential community $\bar{H}^k$ such that $f(\bar{H}^k) > f(H^k)$.*

We illustrate Definition 3 in the following example.

EXAMPLE 2. *Let us reconsider the graph shown in Fig. 1. Assume that $k = 2$. By Definition 3, we can see that the subgraphs induced by $\{v_3, v_4, v_5\}$, $\{v_8, v_9, v_{11}\}$ and $\{v_{13}, v_{14}, v_{15}\}$ are non-contained 2-influential communities. However, the subgraph induced by $\{v_{12}, v_{13}, v_{14}, v_{15}\}$ is not a non-contained 2-influential community, because it includes a 2-influential community (the subgraph induced by $\{v_{13}, v_{14}, v_{15}\}$) with a larger influence value.*

Problem 2. Given a graph $G = (V, E)$, a weight vector W, and parameters k and r, the problem is to find the top-r non-contained k-influential communities with the highest influence value.

Note that with the *Non-containment* constraint, there is no inclusion relationship in the top-r non-contained k-influential communities, thus no redundant results are introduced.

Challenges. A k-influential community is different from the maximal k-core in two aspects. First, a k-influential community must be a connected subgraph, whereas the maximal k-core does not impose such a constraint. Second, with the *maximal structure* constraint, a k-influential community H^k requires that there is no super-graph of H^k which is a connected k-core with influence value equivalent to $f(H^k)$. However, the maximal k-core H only requires that there is no super-graph of H which is also a k-core. For a non-contained k-influential community, it further imposes a *non-containment* constraint. Due to these differences, given a graph G, the maximal k-core of G is unique, whereas there are multiple (non-contained) k-influential communities in G. Thus, the methods for computing the maximal k-core cannot be directly used for computing the top-r (non-contained) k-influential communities.

A straightforward method to compute the top-r (non-contained) k-influential communities is first to compute the set of subgraphs that satisfy the *connectivity* and *cohesiveness* constraints. For each subgraph, we further check whether it satisfies the *maximal structure* constraint and the *non-containment* constraint (for non-contained k-influential communities). Finally, the top-r (non-contained) k-influential communities with the highest influence value are returned. Obviously, such a method is inefficient since the number of potential subgraphs that satisfy the *connectivity* and *cohesiveness* constraints can be exponentially large, and for each potential subgraph, we need to check the *maximal structure* constraint, which is costly. In the following sections, we will present several efficient algorithms to tackle these challenges.

3. ONLINE SEARCH ALGORITHM

In this section, we focus on developing online search algorithms for Problem 1, and then discuss how to generalize the proposed algorithms for Problem 2. Before we proceed further, we give three useful lemmas as follows. Due to space limit, all the proofs are deferred to the full version of this paper.

LEMMA 1. *For any graph G, each maximal connected component of the maximal k-core of G is a k-influential community.*

Algorithm 1 The basic algorithm

```
Input:  $G = (V, E)$, $W$, $r$, and $k$
Output: The top-$r$ $k$-influential communities
1: $G_0 \leftarrow G, i \leftarrow 0$;
2: while $G_i$ contains a $k$-core do
3:     Compute the maximal $k$-core $C^k(G_i)$;
4:     Let $H^k(i)$ be the maximal connected component of $C^k(G_i)$ with
       the smallest influence value;
5:     Let $u$ be the smallest-weight node in $H^k(i)$;
6:     Delete $u$;
7:     Let $G_{i+1}$ be a subgraph of $C^k(G_i)$ after deleting $u$;
8:     $i \leftarrow i + 1$;
9: Output $H^k(i-1), \cdots, H^k(i-r)$ if $i \geq r$, otherwise output $H^k(i-1), \cdots, H^k(0)$.
```

LEMMA 2. *For any k-influential community $H^k = (V_H^k, E_H^k)$, if we delete the smallest-weight node in H^k and the resulting subgraph still contains a maximal k-core $C^k = (V_C^k, E_C^k)$, then each maximal connected component of C^k is a k-influential community.*

LEMMA 3. *For any k-influential community $H^k = (V_H^k, E_H^k)$, if we delete the node in H^k with the smallest weight and the resulting subgraph does not contain a k-core, then H^k is a non-contained k-influential community.*

Based on the above lemmas, we are ready to devise efficient algorithms for Problem 1 and Problem 2. Below, we first develop a basic algorithm for our problems, and then propose an optimized algorithm based on depth-first search (DFS), which is much more efficient than the basic one.

3.1 The basic algorithm

The basic idea of our algorithm is described below. First, for a given k, we compute the maximal k-core of the graph G denoted by $C^k(G)$. Then, we iteratively invoke the following procedure until the resulting graph does not contain a k-core. The procedure consists of two steps. Let G_i be the resulting graph in the i-th iteration, and $C^k(G_i)$ be the maximal k-core of G_i. The first step is to delete the smallest-weight node in $C^k(G_{i-1})$, which results in a graph G_i. The second step is to compute the maximal k-core $C^k(G_i)$ of G_i. The detailed description is outlined in Algorithm 1.

Below, we first show that all the $H^k(j)$ $(0 \leq j \leq i-1)$ obtained by Algorithm 1 are k-influential communities. Then, based on this fact, we will prove that Algorithm 1 correctly outputs the top-r k-influential communities in Theorem 2.

THEOREM 1. *Let $\mathcal{H}^k = \{H^k(0), \cdots, H^k(i-1)\}$ be a set including all the $H^k(j)$ $(0 \leq j \leq i-1)$ obtained by Algorithm 1. Then, for $0 \leq j \leq i-1$, $H^k(j)$ is a k-influential community.*

THEOREM 2. *Algorithm 1 correctly finds the top-r k-influential communities.*

According to Theorem 2, we have a corollary as shown below.

COROLLARY 1. *Given a graph G with n nodes. For a given k, the number of k-influential communities in G is bounded by n.*

We analyze the time and space complexity of Algorithm 1 in the following theorem.

THEOREM 3. *The time complexity of Algorithm 1 is $O(N_k m)$ bounded by $O(nm)$, where N_k denotes the number of k-influential communities. The space complexity of Algorithm 1 is $O(n + m)$.*

Note that we can slightly modify Algorithm 1 to obtain the top-r non-contained k-influential communities. Specifically, we only need to add one line behind line 6 in Algorithm 1 to check whether $H^k(i)$ includes a k-core or not. If $H^k(i)$ does not include a k-core, then by Lemma 3, $H^k(i)$ is a non-contained k-influential community, and we mark such a $H^k(i)$ as a candidate result. Finally, in line 9, we only output the top-r results that are marked as candidate results. It is easy to show that the time and space complexity of this algorithm is the same as those of Algorithm 1.

Algorithm 2 The DFS-based algorithm

```
Input:    $G = (V, E)$, $W$, $r$, and $k$
Output:   The top-$r$ $k$-influential communities
1: $i \leftarrow 0$;
2: Compute the maximal $k$-core $C^k(G)$ of $G$;
3: while $C^k(G) \neq \emptyset$ do
4:     Let $H^k(i)$ be the maximal connected component of $C^k(G)$ with
       the smallest influence value;
5:     Let $u$ be the node with the smallest weight in $H^k(i)$;
6:     DFS($u$);
7:     $i \leftarrow i + 1$;
8: Output $H^k(i), \cdots, H^k(i - r + 1)$ if $i \geq r$, otherwise output
   $H^k(i), \cdots, H^k(1)$.

9: Procedure DFS ($u$)
10: for all $v \in N(u, C^k(G))$ do
11:     Delete edge $(u, v)$ from $C^k(G)$;
12:     if $d(v, C^k(G)) < k$ then
13:         DFS($v$);
14: Delete node $u$ from $C^k(G)$;
```

3.2 The DFS-based algorithm

As shown in the previous subsection, Algorithm 1 is very expensive which is clearly impractical for most real-world graphs. Here we present a much more efficient algorithm based on depth-first-search (DFS). The detailed description of the algorithm is shown in Algorithm 2. Similar to Algorithm 1, Algorithm 2 also iteratively computes the k-influential communities one by one. Unlike Algorithm 1, in each iteration, Algorithm 2 does not recompute the maximal k-core. Instead, in each iteration, Algorithm 2 recursively deletes all the nodes that are definitely excluded in the subsequent k-influential communities. In particular, when Algorithm 2 deletes the smallest-weight node in the k-influential community $H^k(i)$ (line 6), the algorithm recursively deletes all the nodes that violate the *cohesiveness* constraint by a DFS procedure (lines 9-14). This is because, when we delete the smallest-weight node u, the degrees of u's neighbor nodes decrease by 1. This may result in that some of u's neighbors violate the *cohesiveness* constraint, thus they cannot be included in the subsequent k-influential communities, and thereby we have to delete them. Similarly, we also need to verify the other hop (e.g., 2-hop, 3-hop, etc.) neighbors whether they satisfy the *cohesiveness* constraint. Clearly, we can use a DFS procedure to identify and delete all those nodes. The correctness of Algorithm 2 is shown in Theorem 4.

THEOREM 4. *Algorithm 2 correctly finds the top-r k-influential communities.*

The complexity of Algorithm 2 is analyzed in Theorem 5.

THEOREM 5. *The time complexity and space complexity of Algorithm 2 are both $O(m + n)$.*

Similarly, we can easily modify Algorithm 2 for Problem 2. In particular, we only need to add one line behind line 6 in Algorithm 2 to check whether all nodes in $H^k(i)$ are deleted or not. If that is the case, $H^k(i)$ is a non-contained k-influential community (by Lemma 3), and we mark such a $H^k(i)$ as a candidate result. Finally, in line 8, we only output the top-r results that are marked. It is easy to show that the time complexity and space complexity of this algorithm are the same as those of Algorithm 2.

4. INDEX-BASED SEARCH ALGORITHM

Although Algorithm 2 is much more efficient than Algorithm 1, it takes $O(m + n)$ time for each query which is still inefficient for very large graphs. In this section, we present an index-based algorithm whose time complexity is proportional to the size of the top-r results, thus it is optimal. The general idea is that the algorithm first pre-computes all k-influential communities for every k, and then uses a space-efficient structure to index all such k-influential communities in memory. Based on the index, the algorithm outputs the top-r results in optimal time. The challenges of this algorithm are twofold: (1) how to devise a space-efficient structure to store all the k-influential communities, and (2) how to efficiently construct such an index. This is because there could be $O(n)$ k-influential communities for each k (see Corollary 1), and thus there could be $O(k_{\max} n)$ k-influential communities in total, where $k_{\max}$ is the maximal core number of the nodes in G. Obviously, it is impractical to directly store all such k-influential communities for very large graphs. To tackle these challenges, we will present a novel linear-space structure, called ICP-Index (influential-community preserved index), to compress and store all the k-influential communities, and then we propose two algorithms to efficiently construct the ICP-Index.

4.1 The novel ICP-Index

The idea of the ICP-Index is based on the following observation.

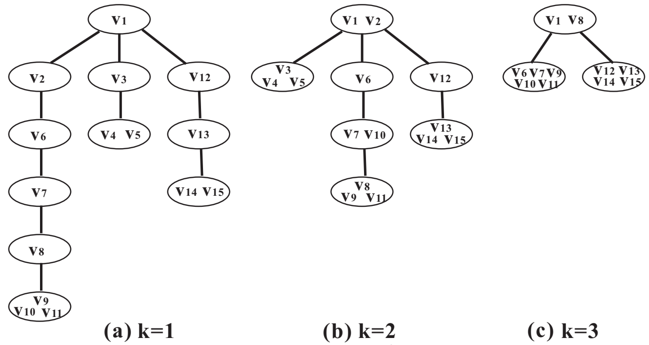
Observation. For each k, the k-influential communities form an inclusion relationship. Based on such an inclusion relationship, all the k-influential communities can be organized by a tree-shape (or a forest-shape) structure.

Recall that Lemma 2 implies an inclusion relationship in the k-influential communities. More specifically, based on Lemma 2, we can see that a k-influential community H^k contains all sub-k-influential communities which are the MCCs of the maximal k-core of $H^k \backslash \{u\}$, where u is the smallest-weight node in H^k. Note that all these sub-k-influential communities are disjoint, because they are different MCCs. Clearly, we can use a two-level tree structure to characterize the inclusion relationships among all these k-influential communities. The parent vertex is H^k, and each child vertex is a MCC of the maximal k-core of $H^k \backslash \{u\}$ which is also a k-influential community. Note that the result of Lemma 2 can be recursively applied in each sub-k-influential community. Thus, we can obtain a tree structure for an *initial* k-influential community, where each vertex of the tree corresponds to a k-influential community. To organize all the k-influential communities of a graph G, we can set the *initial* k-influential communities as the MCCs of the maximal k-core of G. As a consequence, we are able to use a tree (or forest[1]) structure to organize all the k-influential communities, where the root vertex of a tree is a MCC of the maximal k-core of G. Additionally, by Lemma 3, it is easy to see that each leaf vertex in such a tree corresponds to a non-contained k-influential community. To avoid confusion, in the rest of this paper, we use the term vertex to denote a node in a tree.

Compression method. Based on the inclusion relationship between the parent vertex and child vertex in the tree (or forest) structure, we can compress all k-influential communities. Our compression solution is described as follows. For each non-leaf vertex in the tree which corresponds to a k-influential community, we only store the nodes of the k-influential community that are not included in it's sub-k-influential communities (i.e, it's child vertices in the tree). The same idea is recursively applied to all the non-leaf vertices of the tree following a *top-down* manner. For each leaf vertex which corresponds to a non-contained k-influential community, we store all the nodes of that non-contained k-influential community. The following example illustrates the tree organization of all the k-influential communities.

EXAMPLE 3. *Reconsider the graph shown in Fig. 1. Let us consider the case of $k = 2$. Clearly, the entire graph is a connected 2-core, thus it is a 2-influential community. Therefore, the root vertex of the tree corresponds to the entire graph. After deleting the smallest-weight node v_1, we get three 2-influential communities which are the subgraphs induced by the node sets $\{v_3, v_4, v_5\}$,*

[1] If the maximal k-core of G has more than one MCCs, then the index structure is a forest, where each MCC generates a tree.



(a) k=1 **(b) k=2** **(c) k=3**

Figure 2: Tree organization of all the k-influential communities

Algorithm 3 The basic index construction algorithm

```
Input:  $G = (V, E)$ and $W$
Output: The ICP-Index
1:  for $i = 1$ to $k_{\max}$ do
2:      $j \leftarrow 0$; $IT_k \leftarrow \emptyset$
3:      Compute the maximal $k$-core $C^k(G)$ of $G$;
4:      while $C^k(G) \neq \emptyset$ do
5:          Let $H^k(j)$ be the maximal connected component of $C^k(G)$ with
            the smallest influence value;
6:          Let $u$ be the node with the smallest weight in $H^k(j)$;
7:          DFS($u$) {The same DFS procedure as invoked in Algorithm 2};
8:          Let $S_j$ be a set of nodes that are deleted in DFS($u$);
9:          Add a vertex $S_j$ in $IT_i$;
10:         $j \leftarrow j + 1$;
11: return ConstructTree();
```

$\{v_6, \cdots, v_{11}\}$, and $\{v_{12}, \cdots, v_{15}\}$ respectively. Thus, we create three child vertices for the root vertex which corresponds to the three 2-influential communities respectively. Since v_1 and v_2 are not included in these three 2-influential communities, we store them in the root vertex. The same idea is recursively applied in all the three 2-influential communities. For instance, for the 2-influential community induced by $\{v_3, v_4, v_5\}$, we can find that it is a non-contained 2-influential community. By our compression method, we store the nodes $\{v_3, v_4, v_5\}$ in the corresponding tree vertex. For the other child vertices of the root, we have a similar process. Also, similar processes can be used for other k values. Fig. 2 shows the tree organization for all k for the graph shown in Fig. 1.

We refer to the above tree-shape structures for all k from 1 to k_{max} as the ICP-Index. Below, we analyze the space complexity of the ICP-Index in Theorem 6.

THEOREM 6. *The space complexity of the* ICP-Index *is $O(m)$.*

By Theorem 6, the ICP-Index takes linear space w.r.t. the graph size, thus it can be used for very large graphs. Below, we present two algorithms to construct the ICP-Index.

4.2 The basic index construction algorithm

The basic index construction algorithm is to invoke Algorithm 2 $k_{\max}$ times, where $k_{\max}$ is the maximal core number of the nodes in G. Specifically, the basic algorithm consists of two steps. In the first step, the algorithm iteratively calls Algorithm 2 to compute all the tree vertices for each k ($k = 1, \cdots, k_{\max}$). Then, in the second step, the algorithm invokes a tree construction algorithm to build the ICP-Index. The detailed description of the algorithm is outlined in Algorithm 3. Note that in line 8 of Algorithm 3, all the nodes deleted after invoking DFS(u) must be stored in a tree vertex. The reason is that the nodes deleted by DFS(u) are excluded in any sub-k-influential communities of the current k-influential community. Moreover, only these nodes in the current k-influential community are excluded in its sub-k-influential communities. Thus, by our compression method, we need to create a tree vertex that contains all these nodes (line 9 of Algorithm 3). After generating all the tree vertices for all k, the algorithm calls Algorithm 4 to construct the ICP-Index (line 11 of Algorithm 3).

Algorithm 4 ConstructTree()

```
1:  for $i = 1$ to $k_{\max}$ do
2:      Create a signal-vertex tree for each vertex in $IT_i$;
3:  for all node $u$ in $G$ sorted in decreasing order of $w_u$ do
4:      for all $v \in N(u, G)$ s.t. $w_v > w_u$ do
5:          for $i = 1$ to $\min\{c_u, c_v\}$ do
6:              $S_u \leftarrow$ the root node of the tree in $IT_i$ containing $u$;
7:              $S_v \leftarrow$ the root node of the tree in $IT_i$ containing $v$;
8:              if $S_u \neq S_v$ then
9:                  Merge the trees rooted at $S_u$ and $S_v$ in $IT_i$ by adding $S_v$
                    as a child vertex of $S_u$;
10: return $\{IT_1, \cdots, IT_{k_{\max}}\}$;
```

$k = 1$	$\{v_1\}, \{v_2\}, \{v_3\}, \{v_4, v_5\}, \{v_6\}, \{v_7\}, \{v_8\}, \{v_9, v_{10}, v_{11}\}, \{v_{12}\}, \{v_{13}\}, \{v_{14}, v_{15}\}$
$k = 2$	$\{v_1, v_2\}, \{v_3, v_4, v_5\}, \{v_6\}, \{v_7, v_{10}\}, \{v_8, v_9, v_{11}\}, \{v_{12}\}, \{v_{13}, v_{14}, v_{15}\}$
$k = 3$	$\{v_1, v_8\}, \{v_6, v_7, v_9, v_{10}, v_{11}\}, \{v_{12}, v_{13}, v_{14}, v_{15}\}$

Table 1: The tree vertices for all k

Specifically, Algorithm 4 works in a bottom-up manner. To build a tree (or forest) structure for each k ($k = 1, \cdots, k_{\max}$), the algorithm first builds a single-vertex tree for each tree vertex generated in the previous step (lines 1-2 of Algorithm 4). Then, for each k, the final tree (or forest) structure can be obtained by iteratively merging two subtrees (lines 3-9). Here the merge operation between two subtrees T_1 and T_2 is defined as follows. Let r_1 and r_2 be the roots of subtrees T_1 and T_2 respectively. Assume that $f(r_1) < f(r_2)$ where $f(r_i) = \min_{u \in r_i}\{w_u\}$ for $i = 1, 2$. Then, the merge operation between T_1 and T_2 is to set the root of T_2 as a child vertex of the root of T_1. Note that this subtree merge operation can be efficiently implemented by using a union-find structure [9]. Moreover, we find that such a bottom-up tree construction algorithm for all k can be done via traversing the graph once, following a decreasing order of the node weight (lines 3-9 of Algorithm 4). The detailed implementation is depicted in Algorithm 4. We prove the correctness of Algorithm 4 in Theorem 7.

THEOREM 7. *Algorithm 4 correctly creates the tree-shape structures for all k ($k = 1, \cdots, k_{\max}$).*

The correctness of Algorithm 3 can be guaranteed by Theorem 4 and Theorem 7. Below, we give an example to show how Algorithm 3 works.

EXAMPLE 4. *Consider the graph shown in Fig. 1. For each k, by invoking the DFS-based algorithm, Algorithm 3 generates all the tree vertices shown in Table 1. Then, the algorithm calls Algorithm 4 to build the tree index. First, for each k ($k = 1, 2, 3$), the Algorithm 4 creates a tree for each vertex. For instance, for $k = 1$, the algorithm generates 11 trees, because in Table 1 (row 1), there are 11 vertices when $k = 1$. Then, the algorithm processes the node v_{15}, as it is the largest-weight node. As can be seen, v_{15} has four neighbor nodes $\{v_1, v_{12}, v_{13}, v_{14}\}$. But the weights of all of them are smaller than w_{15}, thus the algorithm continues to process node v_{14}. Since v_{14} has a neighbor v_{15} whose weight exceeds w_{14}, the algorithm traverses the edge (v_{14}, v_{15}) (line 4 of Algorithm 4). Since the core numbers of v_{14} and v_{15} are 3, these two nodes must be included in the vertices of the trees of $k = 1, 2, 3$. Thus, for each k ($k = 1, 2, 3$), the algorithm first finds the root vertices of the trees including v_{14} and v_{15} respectively (lines 6-7 of Algorithm 4). Since both v_{14} and v_{15} are included in the same vertex for all k, no merge operation will be done. For the remaining nodes, we use the same procedure, and we will obtain the tree-shape structures shown in Fig. 2 when the algorithm terminates.*

We analyze the time complexity of Algorithm 3 and Algorithm 4 as follows.

THEOREM 8. *The time complexities of Algorithm 3 and Algorithm 4 are $O(k_{\max}(m + n))$ and $O(\rho m)$ respectively, where ρ is the arboricity [7] of a graph G.*

Algorithm 5 The new index construction algorithm

```
Input:   $G = (V, E)$
Output:  The ICP-Index
 1: Compute the core number $c_u$ for each node $u \in V(G)$;
 2: for all $u \in V(G)$ do
 3:     $x_u \leftarrow |\{v|v \in N(u, G), c_v >= c_u\}|$; $\tilde{c}_u \leftarrow c_u$;
 4: $IT_i \leftarrow \emptyset$ for $1 \leq i \leq k_{\max}$;
 5: for all $u \in V(G)$ sorted in increasing order of $w_u$ do
 6:     for $i = 1$ to $\tilde{c}_u$ do
 7:         $S_i \leftarrow \{u\}$;
 8:     $k \leftarrow \tilde{c}_u$; $\tilde{c}_u \leftarrow -1$;
 9:     $U \leftarrow \emptyset$;
10:     UpdateCore($u$, $k$, $S$, $U$);
11:     UpdateSupport($U$);
12:     for $i = 1$ to $k$ do
13:         Add a vertex $S_i$ in $IT_i$;
14: return ConstructTree();
```

In addition, it is easy to derive that the space complexity of Algorithm 3 is $O(m + n)$, which is linear w.r.t. the graph size.

4.3 The new index construction algorithm

As shown in previous subsection, the time complexity of the basic index construction algorithm is $O(k_{\max}(m+n))$ which is inefficient when the graph size and $k_{\max}$ are very large. Here we propose a much more efficient algorithm to construct the ICP-Index.

Recall that in Algorithm 3, the most time-consuming step is to generate all the tree vertices for all k. Thus, to reduce the time overhead, we strive to reduce the time cost of the tree vertices generation procedure. Unlike Algorithm 3 which creates all tree vertices following the increasing order of k (i.e., $k = 1, \cdots, k_{\max}$), the key idea of our new algorithm is that it generates all tree vertices following the increasing order of node weights. Specifically, the new algorithm iteratively deletes the nodes following the increasing order of their weights. When the algorithm removes a node u in an iteration, the algorithm will generate the tree vertices containing u for all k. Thus, if all the nodes are deleted, all tree vertices are generated. After creating all tree vertices, the algorithm calls Algorithm 4 to build the ICP-Index. The rationale behind the new algorithm is as follows. We observe in Algorithm 3 that for each k, all the tree vertices are generated based on the increasing order of node weights. Since all the tree generation procedures for $k = 1, \cdots, k_{\max}$ share the same node order, we can simultaneously create all the tree vertices for all k by following this order.

The challenge of the new algorithm is how to correctly create the tree vertices for all k when deleting a node. Note that a node u with core number c_u is included in c_u different vertices in the trees with $k = 1, 2, \cdots, c_u$ respectively. Thus, if u is deleted, the new algorithm must simultaneously creates c_u different tree vertices. Since each tree vertex containing u may also include other nodes, the algorithm also needs to find these nodes and add them into the tree vertex that includes u. Furthermore, after deleting a node, the core numbers of some other nodes may be updated. Therefore, when the algorithm deletes node u, the current core number of u denoted by $\tilde{c}_u$ may not be the original c_u, as it may be updated after a node is deleted. This gives rise to a new challenge to devise such a tree vertices generation algorithm.

To overcome the above challenges, we develop an algorithm that can correctly create the tree vertices for all k when deleting a node. The idea of our algorithm is that when the algorithm deletes a node u in an iteration, it creates $\tilde{c}_u$ (i.e., the current core number of u) tree vertices and dynamically maintains the core numbers of the other nodes after deleting u. By an in-depth analysis of our algorithm, we can show that all the tree vertices containing u that are not created in this iteration have already been generated before deleting u. The detailed description of our algorithm is shown in Algorithm 5.

Algorithm 6 UpdateCore(u, k, S, U)

```
 1: if $\tilde{c}_u \neq -1$ then
 2:     $S_{\tilde{c}_u+1} \leftarrow S_{\tilde{c}_u+1} \bigcup \{u\}$;
 3: $U \leftarrow U \bigcup \{u\}$;
 4: for all $v \in N(u, G)$ $s.t. \tilde{c}_u \leq c_v$ do
 5:     if $\tilde{c}_v = -1$ or $v \in U$ then
 6:         continue;
 7:     if ($\tilde{c}_u = -1$ and $\tilde{c}_v \leq k$) or ($\tilde{c}_u \neq -1$ and $\tilde{c}_v = \tilde{c}_u + 1$) then
 8:         $x_v \leftarrow x_v - 1$;
 9:         if $x_v < \tilde{c}_v$ then
10:             $\tilde{c}_v \leftarrow \tilde{c}_v - 1$;
11:             UpdateCore($v$, $k$, $S$, $U$);
```

Algorithm 7 UpdateSupport(U)

```
1: for all $u \in U$ do
2:     $x_u \leftarrow 0$;
3:     if $\tilde{c}_u = -1$ then
4:         continue;
5:     for all $v \in N(u, G) s.t. \tilde{c}_u \leq c_v$ do
6:         if $\tilde{c}_v \geq \tilde{c}_u$ then
7:             $x_u \leftarrow x_u + 1$;
```

Algorithm 5 iteratively deletes the nodes by following the increasing order of their weights (line 5). In each iteration, the algorithm creates $\tilde{c}_u$ tree vertices when deleting u, where $\tilde{c}_u$ is the updated core number of node u (lines 6-7). Note that in Algorithm 5, the algorithm does not explicitly delete a node. Instead, the algorithm sets the core number of a node to -1, indicating that the node is deleted (line 8). After deleing a node, the algorithm calls Algorithm 6 and Algorithm 7 to dynamically maintain the core numbers of the remaining nodes (lines 10-11). Notice that Algorithm 6 and Algorithm 7 generalize the core maintenance algorithm independently proposed in [16, 20] to handle the case of node deletion[2]. Here we implement this core maintenance algorithm by dynamically updating the *support* of each node u (denoted by x_u), which is defined as the number of neighbors whose updated core numbers are no smaller than $\tilde{c}_u$. When the support of a node u is smaller than it's current core number (i.e., $x_u < \tilde{c}_u$), the core number of u must be updated (lines 9-11 of Algorithm 6). Note that the core numbers of all the remaining nodes decrease by at most 1 after removing a node. In addition, after deleing a node u, the neighbor nodes of u with core numbers larger than $\tilde{c}_u$ may need to update their core number (line 4 of Algorithm 6). Moreover, in the core number maintenance procedure (Algorithm 6), the algorithm also needs to add the nodes whose core numbers are updated into the corresponding tree vertices (line 2 of Algorithm 6). The correctness of Algorithm 5 is shown in Theorem 9.

THEOREM 9. *Algorithm 5 correctly creates the* ICP-Index.

The following example illustrates how Algorithm 5 works.

EXAMPLE 5. *Consider the graph shown in Fig. 1. In the first iteration of Algorithm 5, the algorithm processes node v_1. Since $\tilde{c}_1 = 3$, the algorithm creates three tree vertices that include v_1, which is denoted by $S_1(v_1)$, $S_2(v_1)$, and $S_3(v_1)$ respectively (lines 6-7). Note that here $S_i(v_1)$ $(i = 1, 2, 3)$ denotes a tree vertex that belongs to the tree of $k = i$. Subsequently, the algorithm sets the core number of v_1 to -1, indicating that v_1 is deleted. Then, the algorithm invokes Algorithm 6 to update the core numbers of the remaining nodes. After invoking Algorithm 6, we can find that v_2 is inserted into the tree vertex $S_2(v_1)$, and v_8 is added into the tree vertex $S_3(v_1)$. Moreover, the core numbers of v_2 and v_8 are updated to 1 and 2 respectively. After that, all the tree vertices containing v_1 have been generated, which is consistent with the tree vertices shown in Table 1. In the second iteration, the algorithm continues to deal with node v_2 by following the increasing order of node*

[2] The original core maintenance algorithms independently proposed in [16, 20] mainly focus on edge deletion and insertion.

weights. Since the current core number of v_2 is 1, in this iteration, the algorithm only creates one tree vertex $S_1(v_2)$ that contains v_2 (lines 6-7). Likewise, the algorithm sets the core number of v_2 to -1, denoting that v_2 is removed. Then, the algorithm calls Algorithm 6 to update the core numbers of the remaining nodes. After invoking Algorithm 6, we can see that no node needs to update its core number. Therefore, in this iteration, the algorithm generates only one tree vertex $S_1(v_2)$ that contains only one node v_2. Up to this iteration, all the tree vertices that includes v_2 is created. Other iterations are processed similarly. After processing all nodes, the algorithm correctly generates all tree vertices shown in Table 1. Finally, the algorithm calls Algorithm 4 to construct the ICP-Index.

The time complexity of Algorithm 5 is shown in Theorem 10.

THEOREM 10. *The time complexity of Algorithm 5 is $O(\rho m)$, where ρ is the arboricity of the graph.*

REMARK 1. *According to [7], the arboricity of a graph is never larger than $O(\sqrt{m})$ in the worst case, and it has shown to be very small in many real-world graphs [17, 14]. Thus, the time cost of Algorithm 5 is much lower than the worst case bound, which is also confirmed in our experiments.*

In addition, it is very easy to show that the space complexity of Algorithm 5 is $O(m+n)$.

4.4 Query processing algorithm

Based on the ICP-Index, the query processing algorithm is straightforward. For Problem 1, to compute the top-r k-influential communities, the algorithm first finds the tree corresponding to k from the ICP-Index, and then outputs the nodes in the top-r subtrees with the highest weights (the weight of a subtree is the minimum weight of nodes in its root vertex). This is because in our ICP-Index, the nodes included in a subtree of the tree corresponding to k exactly form a k-influential community. Similarly, for Problem 2, the algorithm outputs nodes in the top-r leaf vertices with the highest weights in the tree corresponding to k, as the nodes in each leaf vertex form a non-contained k-influential community. The time complexity of the query processing algorithm for both Problem 1 and Problem 2 is linear w.r.t. the answer size[3], thus it is optimal.

5. UPDATE IN DYNAMIC NETWORKS

Many real-world networks are frequently updated. Clearly, when the network is updated, both the ICP-Index and the top-r results also need to be updated. The challenge is that a single edge insertion or deletion may trigger updates in a number of tree vertices of the ICP-Index. This can be an expensive operation because the corresponding tree vertices need to be recomputed. For example, consider a graph shown in Fig. 1. After inserting an edge (v_{10}, v_{11}), the tree vertex $\{v_9, v_{10}, v_{11}\}$ in the tree of $k=1$ (See Table 1) needs to be split into two tree vertices $\{v_9\}$ and $\{v_{10}, v_{11}\}$. In the tree of $k=2$, the two tree vertices $\{v_7, v_{10}\}$ and $\{v_8, v_9, v_{11}\}$ are updated by three tree vertices which are $\{v_7\}$, $\{v_8\}$, and $\{v_9, v_{10}, v_{11}\}$. In the tree of $k=3$, no update is needed. To overcome this challenge, we will propose an efficient algorithm for dynamically maintaining the tree vertices of the ICP-Index when the network is updated. Note that we can also efficiently answer the query by using the tree vertices only (without the tree structure). Specifically, we can first find the top-r tree vertices, and then only search the neighbors of the nodes in the tree vertices to construct the answer (i.e., the tree structure is implicitly constructed online). It is easy to show that the time complexity of this algorithm is the same as the time complexity of the previous tree-based algorithm to construct the top-r results (include edges). Therefore, in this paper, we mainly focus on updating the tree vertices. Below, we consider two types of updates: edge insertion and edge deletion.

Before we proceed further, we define some useful and frequently used notations. Let $r_{\max}$ be the maximal r in the queries posed by the users. For example, we can assume $r_{\max} = 100,000$, because users typically are not interested in the results beyond top-$100,000$. For convenience, we refer to the tree of $k=i$ in the ICP-Index as tree-i. Let $\tilde{r}_u$ be the rank of u in the sorted list of nodes with the increasing order by weights. For simplicity, we assume that the rank of a node is based on the property of the node itself, which is independent of edge updates. For each tree-i ($i = 1, \cdots, k_{\max}$), we assign a timestamp for every tree vertex when it is generated by Algorithm 2. Here the timestamp is an integer ranging from 1 to n_i, where n_i denotes the number of vertices in tree-i. Note that by definition, a tree vertex with a large timestamp implies that the tree vertex has a large influence value. Denote by $R_u^{(i)}$ the timestamp of the tree vertex that contains node u in tree-i. For convenience, we also refer to $R_u^{(i)}$ as the timestamp of node u in tree-i when the definition is clear. Let $\tilde{r}_{\max}^{(i)}$ be the rank of the smallest weight node in the tree vertex with timestamp $n_i - r_{\max} + 1$. For example, reconsider the graph shown in Fig. 1. We can see that $\tilde{r}_{v_9} = 9$. In tree-1, $R_{v_9}^{(1)} = 8$, because v_9 is included in the tree vertex $\{v_9, v_{10}, v_{11}\}$ whose timestamp is 8 (See Table 1). Assume that $r_{\max} = 4$. Then, $\tilde{r}_{\max}^{(1)} = 9$, because in tree-1, the tree vertex with timestamp $n_1 - r_{\max} + 1$ (equals 8) is $\{v_9, v_{10}, v_{11}\}$, where the rank of the smallest weight node (v_9) is 9.

5.1 Handling Edge Insertion

Here we consider the case of inserting an edge (u, v). The straightforward method is to re-compute all tree vertices using Algorithm 5 when the graph is updated. Clearly, this method is inefficient for large graphs. Below, we first present two basic updating rules, and then propose a minimum tree re-computation method to further reduce the computational cost for edge insertion.

The basic updating rules: we give two basic updating rules below.
Rule 1: let $c_{\min} = \min\{c_u, c_v\}$ (i.e., the minimum core number of u and v). Then, after inserting (u, v), every tree-i for $i > c_{\min}+1$ will not be updated. This is because when inserting an edge, the core numbers of the nodes increase by at most one [16]. As a result, each i-influential community for $i > c_{\min}+1$ does not change, and thus every tree-i remains unchanged.
Rule 2 (Lazy update): the key idea of the lazy update rule is that we only maintain the tree vertices when they affect the top-r results for $r \le r_{\max}$. Formally, we have the following lemma.

LEMMA 4. *For each tree-i ($i = 1, \cdots, k_{\max}$), if $\tilde{r}_u < \tilde{r}_{\max}^{(i)}$ or $\tilde{r}_v < \tilde{r}_{\max}^{(i)}$, the tree vertices in the top-r results for $r \le r_{\max}$ keep unchanged when the graph is updated by inserting or deleting an edge (u, v).*

Based on the above lemma, when inserting an edge (u, v), we first check the conditions $\tilde{r}_u < \tilde{r}_{\max}^{(i)}$ and $\tilde{r}_v < \tilde{r}_{\max}^{(i)}$. If one of them holds, we do not perform any update operation for the tree vertices in tree-i, because their updates do not affect the top-r results for $r \le r_{\max}$.

The minimum tree re-computation method: Besides the basic updating rules, here we present a method which can achieve minimum tree re-computation when an edge is inserted. The method, as verified in our experiments, can largely reduce the computational cost for edge insertion even after **Rule 1** and **Rule 2** are applied. Recall that after inserting an edge (u, v), all tree-i with $i > c_{\min}+1$ do not change (by **Rule 1**), thus we only need to update all tree-i with $i = 1, \cdots, c_{\min}+1$. Specifically, we consider two cases: (1) all tree-i with $i = 1, \cdots, c_{\min}$, and (2) tree-($c_{\min}+1$).

For case (1), we let l_w^i be the number of w's neighbors whose timestamps are no less than w after inserting (u, v), i.e., $l_w^i = |\{x | x \in N(w, G) \wedge R_x^{(i)} \ge R_w^{(i)}\}|$. By this definition, l_w^i denotes the degree of w in the i-core after deleting all nodes whose timestamps are smaller than w. We assume without loss of generality

[3] Suppose that each answer only contains the set of nodes in each community; Otherwise, we simply compute the induced subgraph by the nodes in the answer.

that $R_u^{(i)} \leq R_v^{(i)}$ in tree-i. Let $IT_i[R_u^{(i)}]$ be the tree vertex containing u and $\bar{u}$ be the smallest weight node in $IT_i[R_u^{(i)}]$. After inserting (u, v), for each tree-i with $i = 1, \cdots, c_{\min}$ (case (1)), we study whether $IT_i[R_u^{(i)}]$ needs to be updated. To this end, we recover the procedure of generating the tree vertex $IT_i[R_u^{(i)}]$. In particular, we perform a similar DFS procedure as Algorithm 2 to recursively delete the nodes in $IT_i[R_u^{(i)}]$. Unlike Algorithm 2, here we use l_w^i as the *degree* of node w, and the DFS procedure only traverses the nodes in $IT_i[R_u^{(i)}]$ and their neighbors as well. Similar to Algorithm 2, the DFS procedure initially traverses node $\bar{u}$. When a neighbor node of w for $w \in IT_i[R_u^{(i)}]$ is deleted, l_w^i decreases by 1, and when l_w^i is smaller than i, w is deleted. If node u is deleted when the DFS procedure terminates, the tree vertex $IT_i[R_u^{(i)}]$ does not need to be updated, and thereby all tree vertices keep unchanged. The reason is as follows. First, the insertion of edge (u, v) does not affect the tree vertices with timestamps smaller than $R_u^{(i)}$. Second, if u is deleted, all other nodes in $IT_i[R_u^{(i)}]$ must be deleted (by the definition of tree vertex), and thus the tree vertex $IT_i[R_u^{(i)}]$ does not change. Third, if u is deleted, all u's outgoing edges are also deleted, and thus inserting the edge (u, v) does not affect the tree vertices with timestamps larger than $R_u^{(i)}$. On the other hand, if node u fails to be removed by the DFS procedure, then we re-compute all the tree vertices for tree-i. Below, we give a sufficient and necessary condition for updating the tree vertices in tree-i.

LEMMA 5. *For each tree-i with $i = 1, \cdots, c_{\min}$, the tree vertices in tree-i need to be updated after inserting (u, v), if and only if u is not deleted by the DFS procedure.*

By Lemma 5, a sufficient and necessary condition for updating the tree vertices in tree-i is that u is not deleted by the DFS procedure. Thus, our algorithm, which is based on such a sufficient and necessary condition, is optimal in the sense that the number of tree re-computations by our algorithm is minimum.

For case (2) (tree-$(c_{\min} + 1)$), if u or v's core number is updated, we use **Rule 2** to update the tree vertices in tree-$(c_{\min} + 1)$. Otherwise, no update is needed.

The algorithm for handling edge insertion is depicted in Algorithm 8, which integrates both the basic updating rules and the minimum tree re-computation method. In lines 4-5 and lines 9-10, we use **Rule 2** for updating. In lines 6-7, we apply the minimum tree re-computation method to update the tree vertices. In the main loop (line 3), we use **Rule 1** for updating. In lines 11-17, the procedure IsRecompute is used to determine whether u (assume $R_u^{(i)} \leq R_v^{(i)}$) is deleted by the DFS procedure (InsertionDFS, lines 18-24) or not. Note that in the InsertionDFS procedure, we set $l_u^k = -1$ to denote that u is deleted. The correctness of Algorithm 8 can be guaranteed by Lemma 4 and Lemma 5. The time complexity for checking the tree re-computation conditions in Algorithm 8 (line 6) is $O(\sum_{i=1}^{c_{\min}} \sum_{u \in IT[R_u^{(i)}]} d_u)$. In the experiments, we will show that our algorithm is at least four orders of magnitude faster than the straightforward re-computation based algorithm in large graphs.

5.2 Handling Edge Deletion

Consider the case of deleting an edge (u, v). Similarly, we have two basic updating rules. First, for **Rule 1**, each tree-i with $i > c_{\min}$ ($c_{\min} = \min\{c_u, c_v\}$) will not be updated after deleting an edge (u, v), because all i-influential communities for $i > c_{\min}$ remain unchanged after removing (u, v). Second, for **Rule 2**, we can also use Lemma 4 to handle the edge deletion case. To further improve the efficiency, we also propose a minimum tree re-computation method. For each tree-i with $i = 1, \cdots, c_{\min}$, we let l_w^i be the number of w's neighbors whose timestamps are no less than w after deleting (u, v), i.e., $l_w^i = |\{x | x \in N(w, G) \wedge R_x^{(i)} \geq R_w^{(i)}\}|$. Below, we give a sufficient and necessary condition for updating the tree vertices.

Algorithm 8 EdgeInsertion(u, v)

```
Input:  $G = (V, E)$, and edge $(u, v)$
Output: The updated tree vertices
 1: Updated core numbers for all nodes;
 2: $c_{\min} \leftarrow \min\{c_u, c_v\}$;
 3: for $i = 1$ to $c_{\min}$ do
 4:     if $\tilde{r}_u < \tilde{r}_{\max}^{(i)}$ or $\tilde{r}_v < \tilde{r}_{\max}^{(i)}$ then
 5:         Continue;
 6:     if IsRecompute($u$, $v$, $i$) then
 7:         Recompute all tree vertices for tree $i$;
 8: if $u$ or $v$'s core number is updated then
 9:     if $\tilde{r}_u \geq \tilde{r}_{\max}^{(c_{\min}+1)}$ and $\tilde{r}_v \geq \tilde{r}_{\max}^{(c_{\min}+1)}$ then
10:         Recompute all tree vertices for tree $c_{\min} + 1$;

11: Procedure bool IsRecompute ($u$, $v$, $k$)
12: $R_{\min}^{(k)} \leftarrow \min\{R_u^{(k)}, R_v^{(k)}\}$, $\bar{w} \leftarrow R_u^{(k)} < R_v^{(k)} ? u : v$;
13: for all $w \in IT_k[R_{\min}^{(k)}]$ do
14:     $l_w^k \leftarrow |\{x | x \in N(w, G) \wedge R_x^{(k)} \geq R_w^{(k)}\}|$;
15: Let $\bar{u}$ be the smallest weight node in $IT_k[R_{\min}^{(k)}]$;
16: InsertionDFS($\bar{u}$, $k$, $IT_k[R_{\min}^{(k)}]$);
17: return $(l_{\bar{w}}^k \neq -1)$;

18: Procedure InsertionDFS ($u$, $k$, $IT_k[R_{\min}^{(k)}]$)
19: $l_u^k \leftarrow -1$;
20: for all $v \in N(u, G)$ do
21:     if $v \notin IT_k[R_{\min}^{(k)}]$ or $l_v^k = -1$ then
22:         Continue;
        $l_v^k \leftarrow l_v^k - 1$;
23:     if $l_v^k < k$ then
24:         InsertionDFS ($v$, $k$, $IT_k[R_{\min}^{(k)}]$);
```

Algorithm 9 EdgeDeletion(u, v)

```
Input:  $G = (V, E)$, and edge $(u, v)$
Output: The updated tree vertices
 1: Updated core numbers for all nodes;
 2: $c_{\min} \leftarrow \min\{c_u, c_v\}$;
 3: for $i = 1$ to $c_{\min}$ do
 4:     if $\tilde{r}_u < \tilde{r}_{\max}^{(i)}$ or $\tilde{r}_v < \tilde{r}_{\max}^{(i)}$ then
 5:         Continue;
 6:     Compute $l_u^i$ and $l_v^i$;
 7:     if $l_u^i < i$ or $l_v^i < i$ then
 8:         Recompute all tree vertices for tree $i$;
```

LEMMA 6. *For each tree-i with $i = 1, \cdots, c_{\min}$, the tree vertices in tree-i need to be updated after deleing (u, v), if and only if $l_u^i < i$ or $l_v^i < i$.*

Based on Lemma 6, we can use l_u^i and l_v^i to determine whether the tree vertices in tree-i need to be updated. The algorithm for handling edge deletion is outlined in Algorithm 9, which integrates both the basic updating rules and the minimum tree re-computation method. In lines 4-5, we use **Rule 2** for updating, and in lines 6-8, we use the minimum tree re-computation method to update the tree vertices. In the main loop (line 3), we use **Rule 1** for updating. The time complexity for checking all re-computation conditions in Algorithm 9 (lines 6-7) is $O(d_u + d_v)$. In addition, it is worth mentioning that both Algorithm 8 and Algorithm 9 do not increase the space complexity for top-r k-influential communities search.

6. PERFORMANCE STUDIES

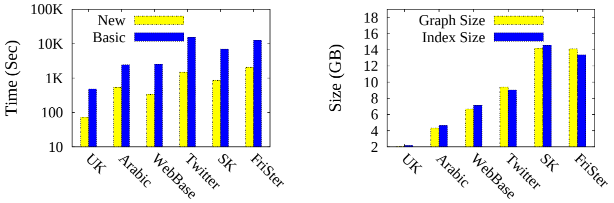
We conduct extensive experiments to evaluate the proposed algorithms. To construct the ICP-Index, we implement both the basic (Algorithm 3) and the new (Algorithm 5) algorithms, denoted

Dataset	n	m	$d_{\max}$	$k_{\max}$
UK	18,520,486	298,113,762	194,955	943
Arabic	22,744,080	639,999,458	575,628	3,247
WebBase	118,142,155	1,019,903,190	816,127	1,506
Twitter	41,652,230	1,468,365,182	2,997,487	2,488
SK	50,636,154	1,949,412,601	8,563,816	4,510
FriSter	65,608,366	1,806,067,135	5,214	304

Table 2: Datasets

Parameter	Range	Default value
k	2, 4, 8, 16, 32, 64, 128, 256	32
r	5, 10, 20, 40, 80, 160, 320	40
n	20%, 40%, 60%, 80%, 100%	100%
m	20%, 40%, 60%, 80%, 100%	100%

Table 3: Parameters



(a) Index-construction time (b) Index size

Figure 3: Index testing

by Basic and New respectively. For query processing, we implement four algorithms, named Online-All, Online-NCT, Index-All, and Index-NCT, respectively. Online-All and Online-NCT are the DFS-based online search algorithms (Algorithm 2) which are used to compute the top-r k-influential communities and the top-r non-contained k-influential communities respectively; Similarly, Index-All and Index-NCT are the ICP-Index based algorithms used to compute the top-r k-influential communities and the top-r non-contained k-influential communities respectively. Note that we do not implement the basic online search algorithm (Algorithm 1), as it is impractical for many real-world graphs. All algorithms are implemented in C++. All experiments are conducted on a computer with 3.46GHz Intel Xeon X5690 (6-core) CPU and 96GB memory running Red Hat Enterprise Linux 6.4 (64-bit). In all experiments, both the graph and the ICP-Index are resident in main memory.

Datasets. We use six web-scale real-world graphs in our experiments. The detailed statistics of our datasets are shown in Table 2. The first five datasets in Table 2 are downloaded from (`http://law.di.unimi.it/datasets.php`), and the FriSter dataset is downloaded from (`http://snap.stanford.edu`). Among the six graphs, UK, Arabic, WebBase, and SK are web graphs, and Twitter and FriSter are social networks.

Parameters. In all the experiments, without otherwise specified, we use the PageRank score of node u to denote its weight, as PageRank is a widely-used model to measure the influence (or importance) of the nodes. For each dataset, we vary 4 parameters: r (denoting the parameter of top-r), k (denoting the parameter of k-influential community), the percentage of nodes n, and the percentage of edges m. The range of the parameters and their default values are shown in Table 3. When varying m (or n) for scalability testing, we extract subgraphs of 20%, 40% 60%, 80% and 100% edges (or nodes) of the original graph with a default value of 100%. When varying a certain parameter, the values for all the other parameters are set to their default values.

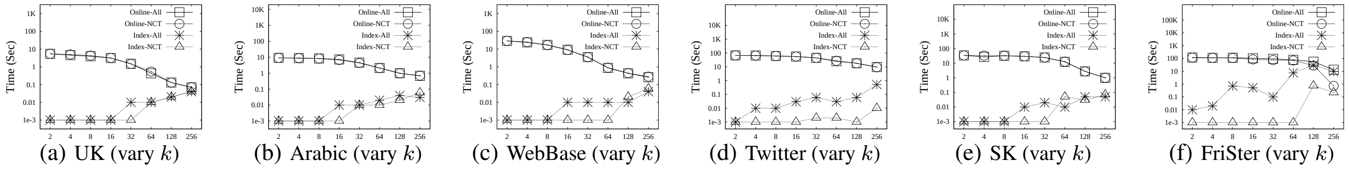
Exp-1: Index Construction. We build the ICP-Index for six graphs using both Basic and New. The index-construction time is shown in Fig. 3(a). New is 5 to 10 times faster than Basic in all datasets. Moreover, we can see that New is very efficient which takes only 1,477 seconds (<25 minutes) in the Twitter dataset (more than 1 billion edges and 41 million nodes). This is because New can avoid computing influential communities for all k values one by one, which saves much computational cost. The result is also consistent with the theoretical analysis shown in Theorem 8 and Theorem 10. We further compare the size of the ICP-Index with the size of the original graph. The results are depicted in Fig. 3(b). Over all the datasets, the sizes of ICP-Index are almost the same as the size of the original graph. This result confirms the theoretical analysis shown in Theorem 6.

Exp-2: Query Processing (Vary k). We vary k from 2 to 256 and evaluate the query processing time for the four proposed algorithms by fixing $r=40$. The results are reported in Fig. 4. In all datasets, when k increases, the processing time of Online-All and Online-NCT decreases. This is because when k increases, the size of the maximal k-core decreases, and the time complexity of Online-All and Online-NCT is dominated by traversing the maximal k-core. Instead, when k increases, the processing time of both Index-All and Index-NCT increases. This is because when k increases, the size of the top-r results increases, and thus it takes more time to calculate the top-r results for both Index-All and Index-NCT. When k is small, Index-All and Index-NCT is several orders of magnitude faster than Online-All and Online-NCT, respectively. When k is large, the advantages of Index-All and Index-NCT are not significant. The reason is that, when k increases, the time cost for traversing the k-core decreases, while the time spent on outputting the top-r results increases. For instance, in UK, when the core number increases to 256, the time overhead for outputting the top-r results dominates the whole query processing time for all algorithms. Thus, the processing time of all the algorithms are similar.

Exp-3: Query Processing (Vary r). We vary the parameter r from 5 to 320 and evaluate the query processing time of the four algorithms by fixing $k=16$. The results are shown in Fig. 5. Over all datasets, we can see that the processing time of all the algorithms increases with increasing r. For Online-All and Online-NCT, the processing time increases very slowly. This is because for both Online-All and Online-NCT, the dominant cost is spent on traversing the maximal k-core other than outputting the top-r results. For Index-All and Index-NCT, when r is small, the processing time increases slowly. However, when r is large, the processing time of Index-All increases while the processing time of Index-NCT still keeps stable. The reason is that when r increases, the size of the r-th answer in the top-r results for the Index-All algorithm tends to increase. Thus, when r is large, a large number of redundant subgraphs are outputted in the top-r results. For Index-NCT, when r increases, the size of the r-th answer in the top-r results does not significantly increase, thus the processing time of Index-NCT keeps stable. For example, in the FriSter dataset, when r increases to 320, the processing time of Index-All approaches the processing time of Online-NCT and Online-All, indicating that a large number of redundant subgraphs are computed in Index-All. However, in this case, Index-NCT is still very efficient, which is four orders of magnitude faster than Index-All.

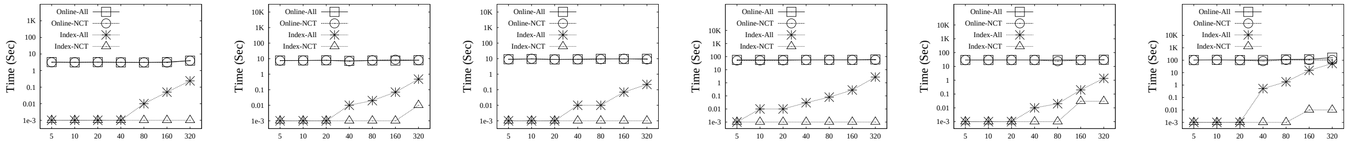
Exp-4: Scalability for Indexing. We vary the number of edges (m) and nodes (n) in Twitter, SK, and FriSter datasets to study the scalability of the index construction algorithms: Basic and New. The results are reported in Fig. 6. As can be seen, both Basic and New scale near linearly in most datasets. Moreover, we can see that New is around one order of magnitude faster than Basic, which is consistent with the previous observations. In addition, we also report the scalability results for index size in Fig. 7. We can see that the index size is nearly the same as the graph size over all testing cases, which confirms the theoretical analysis shown in Section 4.

Exp-5: Scalability for Query Processing. We vary the number of edges (m) and nodes (n) in Twitter, SK, and FriSter datasets to evaluate the scalability of the proposed query processing algorithms. Fig. 8 depicts the results. As desired, the query processing time for the online search algorithms (Online-All and Online-NCT) increases with increasing graph size. However, for the index-based algorithms (Index-All and Index-NCT), the query processing time



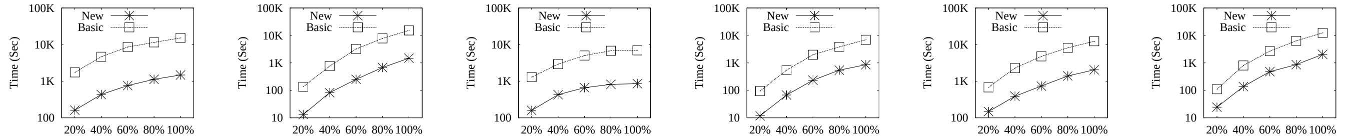
(a) UK (vary k) (b) Arabic (vary k) (c) WebBase (vary k) (d) Twitter (vary k) (e) SK (vary k) (f) FriSter (vary k)

Figure 4: Query processing testing (Vary k)



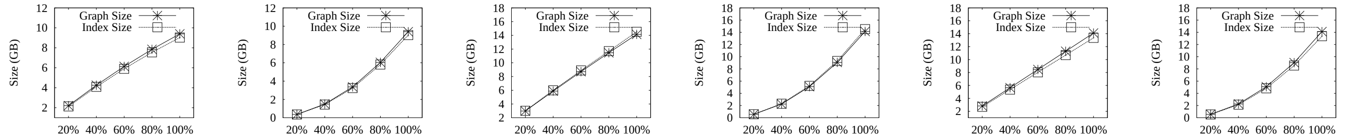
(a) UK (vary r) (b) Arabic (vary r) (c) WebBase (vary r) (d) Twitter (vary r) (e) SK (vary r) (f) FriSter (vary r)

Figure 5: Query processing testing (Vary r)



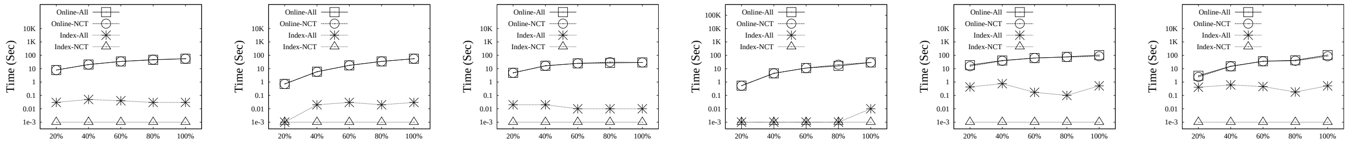
(a) Twitter (vary m) (b) Twitter (vary n) (c) SK (vary m) (d) SK (vary n) (e) FriSter (vary m) (f) FriSter (vary n)

Figure 6: Scalability testing (Index time)



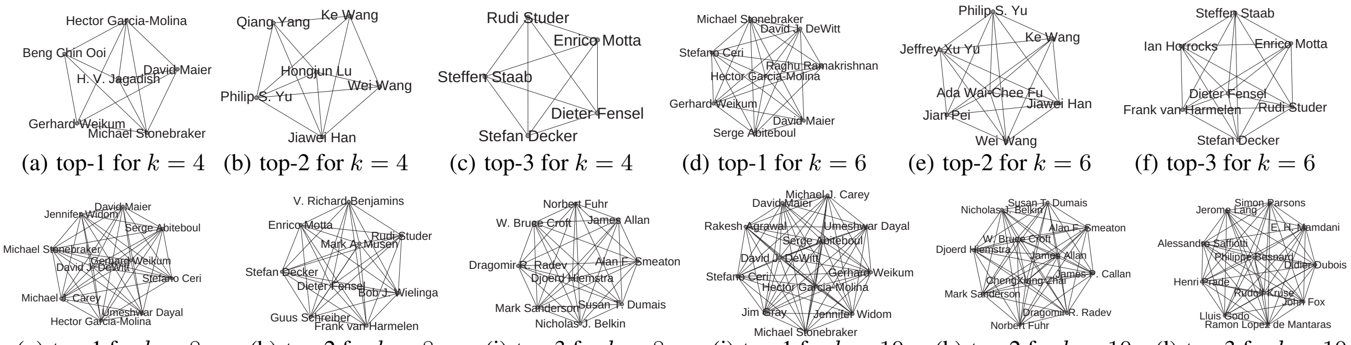
(a) Twitter (vary m) (b) Twitter (vary n) (c) SK (vary m) (d) SK (vary n) (e) FriSter (vary m) (f) FriSter (vary n)

Figure 7: Scalability testing (Index size)



(a) Twitter (vary m) (b) Twitter (vary n) (c) SK (vary m) (d) SK (vary n) (e) FriSter (vary m) (f) FriSter (vary n)

Figure 8: Scalability testing (Query processing time)



(a) top-1 for $k = 4$ (b) top-2 for $k = 4$ (c) top-3 for $k = 4$ (d) top-1 for $k = 6$ (e) top-2 for $k = 6$ (f) top-3 for $k = 6$

(g) top-1 for $k = 8$ (h) top-2 for $k = 8$ (i) top-3 for $k = 8$ (j) top-1 for $k = 10$ (k) top-2 for $k = 10$ (l) top-3 for $k = 10$

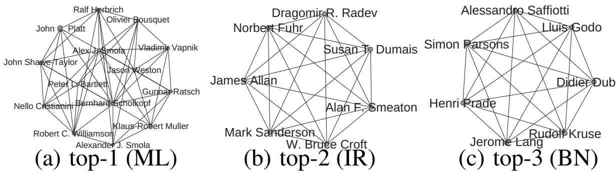
Figure 9: Case study: results for different k and r.

does not significantly increase when the graph size increases. The reason is that the processing time of Index-All and Index-NCT are mainly dependent on the size of the top-r communities, and the size of the top-r communities is not largely affected by the size of the graph. As a result, in all testing cases, Index-All and Index-NCT are at least one order of magnitude faster than Online-All and Online-NCT, respectively.

Exp-6: Dynamic Update. In this experiment, we evaluate the efficiency of the proposed index updating algorithms. We compare three algorithms which are Ba, Ne, and Recompute. Ba is the algorithm using two basic updating rules; Ne is the algorithm using both two basic updating rules and the minimum tree recomputation method (Algorithm 8 and Algorithm 9); Recompute is the straightforward updating algorithm which uses Algorithm 5

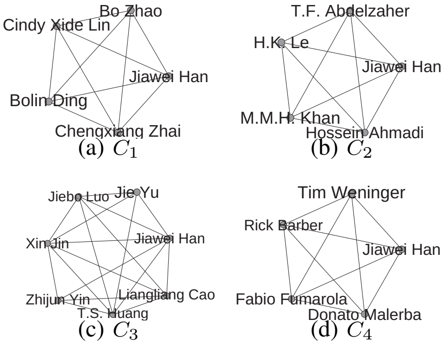
Dataset	Ins (Ba)	Del (Ba)	Ins (Ne)	Del (Ne)	Recompute
UK	2.460	2.188	0.148	0.107	67.27
Arabic	9.658	9.483	0.798	0.466	518.36
WebBase	0.522	0.483	0.201	0.175	331.74
Twitter	66.500	64.947	0.035	0.001	1211.39
SK	2.936	2.940	0.507	0.298	897.41
FriSter	6.074	6.076	0.203	0.001	1919.56

Table 4: Update Time Per Edge (in Seconds)



(a) top-1 (ML) (b) top-2 (IR) (c) top-3 (BN)

Figure 10: Top-3 results using labels for weights ($k = 6$)



(a) C_1 (b) C_2 (c) C_3 (d) C_4

Figure 11: Four truss communities containing "Jiawei Han"

to re-compute all tree vertices when the graph is updated by an edge insertion/deletion. In all testings, we set $r_{\max} = 100,000$. For each dataset, we randomly delete 1K edges, and update the index after every deletion, and then we insert the same 1K edges and update the index after every insertion. The average update time per edge insertion/deletion is reported in Table 4. From Table 4, we can make the following observations. Compared to Recompute, Ba can significantly reduce the cost of maintaining the tree vertices. For example, in WebBase, Ba only takes 0.5 seconds to maintain all the tree vertices for either insertion or deletion, while Recompute requires more than 330 seconds. However, only applying the basic updating rules may be still inefficient. For example, in Twitter, Ba needs more than 60 seconds for each edge insertion/deletion which is inefficient. Ne, however, can significantly cut the updating time of Ba by applying the minimum tree re-computation method. For instance, in the Twitter dataset, by using Ne, the updating time for an edge insertion/deletion is reduced from 66.5/64.9 seconds to 0.035/0.001 seconds. For Ne, handling edge deletion is more efficient than handling edge insertion, because checking the re-computation condition for edge insertion needs to invoke a DFS procedure (see Algorithm 8). In general, we can see that the updating time of Ne is several orders of magnitude faster than the straightforward re-computation based method (Recompute) over all datasets, which confirms the theoretical analysis in Section 5.

6.1 Case studies

We use a co-authorship network extracted from ArnetMiner (`http://arnetminer.org`) for case studies. The dataset consists of authors in different research areas including database, data mining, semantic web, machine learning, information retrieval, Bayesian network, and so on. The graph contains 5411 nodes and 17,477 edges. Each author (node) is associated with a label, denoting the research area of that author. Based on this dataset, we conduct three various case studies to evaluate the effectiveness of the k-influential community model.

Results for different k and r. In this case study, we use the number of publications to denote the weight of an author. We vary k from 4 to 10, and generate the top-3 non-contained k-influential communities for each k value. The results are depicted in Fig. 9. As can be seen, for a certain k, the top results of the non-contained k-influential communities tend to cover high influential researchers in different research areas. For example, when $k = 4$, the top-1 result includes high-influential researchers in database area, the top-2 result contains high-influential researchers in data mining area, and the top-3 result consists of high-influential researchers in semantic web area. The researchers in each community are highly connected with each other, and each of them plays an leading role in the specific research area. These results indicate that the k-influential community model is indeed capable of capturing both influence and cohesiveness of a community.

In addition, we can see that the parameter k can balance the tradeoff between influence and cohesiveness of a community. In general, the influence value of a community decreases with increasing k. For instance, comparing Fig. 9(a) with Fig. 9(d), when k increases from 4 to 6, some high influential researchers such as "H. V. Jagadish" and "Beng Chin Ooi" leave the community, while some other researchers are added into the community, forming a more cohesive but relatively lower influential community. The reason is that when k increases, the *cohesiveness* constraint in the k-influential community model becomes more strict, which may exclude some high influential nodes from the community, and thus may reduce the influence of the community. For a practical recommendation, if the user wants to find a high influential community, a small k is preferred, while if the user aims at finding a high cohesive but relatively low influential community, a large k is preferred.

Using labels for weights. In this case study, we use the labels for weights to study the effectiveness of the k-influential community model. Specifically, we first give different weights for different labels. Then, we rank the nodes based on the weights, and break ties based on the number of publications. Fig. 10 reports the results for $k = 6$ given that the weights of different labels are ranked as "Machine Learning (ML)" > "Information Retrieval (IR)" > "Bayesian Network (BN)", and so on. Similar results can also be observed for different k values (e.g., $k = 8$) and different weighting methods. From Fig. 10, we can see that the top-3 results are consistent with our weighting method (the top-1 result is a "Machine Learning" community, the top-2 result is a "Information Retrieval" community, and the top-3 result is a "Bayesian Network" community). These results suggest that the k-influential community model can also capture user-specified definition of influence. In practice, the users can define the influence based on their preferences, and our proposed methods can be applied to identify the influential communities based on user-defined influence.

Comparison with truss community. Here we compare the proposed community model with the truss community model [15], which is successfully applied to find query-dependent cohesive communities in a large network. For a fair comparison, we compare the k-influential community with the $k + 1$ truss community. This is because a $k+1$ truss is a k-core [24], and our k-influential community is based on k-core. Below, we consider the case when $k = 4$. Similar conclusions can also be made for other k values. Fig. 11 depicts four 5-truss communities containing "Jiawei Han". From Fig. 11, we can see that the 5-truss communities mainly contains professor Jiawei Han's students or research fellows. However, in our 4-influential community model, professor Jiawei Han's community (see Fig. 9(b)) includes many other influential researchers in data mining area who have a co-author relationship with "Jiawei Han". The reason is that the k-truss community only captures the cohesiveness of a community, while our k-influential community not only captures the cohesiveness, but it also considers the influence of a community. Therefore, in practice, if the users wants to find the influential communities in a network, our community model is much better than the k-truss community model.

7. RELATED WORK

Community search and discovery. Sozio et al. [22] studied the community search problem in social networks where the goal is to find the maximal connected k-core with maximal k value that contains the query nodes. In [22], the authors proposed a linear-time algorithm to solve the community search problem. Recently, Cui et al. [11] proposed a more efficient local search algorithm for the same problem. Except the maximal k-core-based model, Cui et al. [10] proposed an α-adjacency γ-quasi-k-clique model to study the overlap community search problem. More recently, Huang et al. [15] studied the community search problem based on a k-truss community model. In addition, another related but different problem is community discovery, which is to discover all the communities in a network. This issue is extensively studied in the literature. Two surveys on this topic can be found in [12, 26]. All the mentioned work do not consider the influence of a community. In this paper, we study the influential community search problem, and our goal is to find the most influential communities in a network.

Cohesive subgraph mining. Cohesive subgraph is an important concept in social network analysis. There are many different definitions of cohesive graphs in the literature, which consists of maximal clique [5, 6], k-core [21, 4, 16], k-truss [8, 24], DN-graph [25], maximal k-edge connected subgraph [29, 3, 1], and so on. Due to a large number of applications, the cohesive subgraph mining problem has attracted much attention in recent years. For example, James et al. proposed a series of external-memory algorithms for finding and enumerating maximal clique [5, 6], and for k-core [4] and k-truss [24] decomposition in massive graphs. Interestingly, many equivalent concepts of k-truss were independently proposed in different papers. For instance, in [19], Saito and Yamada termed the k-truss k-dense community, and this term was also followed in [13]; In [23], k-truss is termed k-brace; In [27], Zhang and Parthasarathy termed the k-truss triangle k-core, and in [28], Zhao and Tung termed the k-truss k-mutual-friend subgraph. DN-graph was proposed in [25] which is closely related k-truss. Unlike k-truss, the problem of mining the DN-graphs is NP-hard. The maximal k-edge connected subgraph (MkCS), also called structural cohesion in sociology [18], is typically more cohesive than k-core and k-truss. Recently, several efficient algorithms were proposed to compute the MkCS. For instance, in [29], Zhou et al. proposed several pruning techniques to speed up the MkCS mining algorithm. In [3], Chang et al. presented a linear-time algorithm based on a graph decomposition framework. In [1], Akiba et al. proposed a linear-time randomized algorithm for the same problem based on a random edge contraction technique.

8. CONCLUSION

We study a problem of finding the top-r influential communities in a network. We propose a new community model called k-influential community to capture the influence of a community. To find the top-r k-influential communities efficiently, we propose a linear-time online search algorithm and an optimal index-based algorithm. Our index structure only takes linear space, and can be constructed efficiently. We also develop an efficient algorithm to maintain the index when the graph is frequently updated. Finally, extensive experiments on 7 large real-world networks demonstrate the efficiency and effectiveness of our algorithms.

Acknowledgements. The work was supported in part by (i) NSFC Grants (61402292, 61170076, U1301252, 61033009) and Natural Science Foundation of SZU (grant no. 201438); (ii) ARC DE140100999; (iii) Research Grants Council of the Hong Kong SAR, China, 14209314 and 418512; (iv) China 863 (no. 2012AA010239) and Guangdong Key Laboratory Project (2012A061400024).

9. REFERENCES

[1] T. Akiba, Y. Iwata, and Y. Yoshida. Linear-time enumeration of maximal k-edge-connected subgraphs in large networks by random contraction. In *CIKM*, 2013.

[2] V. Batagelj and M. Zaversnik. An O(m) algorithm for cores decomposition of networks. *CoRR*, cs.DS/0310049, 2003.

[3] L. Chang, J. X. Yu, L. Qin, X. Lin, C. Liu, and W. Liang. Efficiently computing k-edge connected components via graph decomposition. In *SIGMOD*, 2013.

[4] J. Cheng, Y. Ke, S. Chu, and M. T. Özsu. Efficient core decomposition in massive networks. In *ICDE*, 2011.

[5] J. Cheng, Y. Ke, A. W.-C. Fu, J. X. Yu, and L. Zhu. Finding maximal cliques in massive networks. *ACM Trans. Database Syst.*, 36(4):21, 2011.

[6] J. Cheng, L. Zhu, Y. Ke, and S. Chu. Fast algorithms for maximal clique enumeration with limited memory. In *KDD*, 2012.

[7] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, 14(1):210–223, 1985.

[8] J. Cohen. Trusses: Cohesive subgraphs for social network analysis. *Technique report*, 2005.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[10] W. Cui, Y. Xiao, H. Wang, Y. Lu, and W. Wang. Online search of overlapping communities. In *SIGMOD*, 2013.

[11] W. Cui, Y. Xiao, H. Wang, and W. Wang. Local search of communities in large graphs. In *SIGMOD*, 2014.

[12] S. Fortunato. Community detection in graphs. *Physics Reports*, 486(3-5):75–174, 2010.

[13] E. Gregori, L. Lenzini, and C. Orsini. k-dense communities in the internet as-level topology graph. *Computer Networks*, 57(1):213–227, 2013.

[14] X. Hu, Y. Tao, and C.-W. Chung. Massive graph triangulation. In *SIGMOD*, 2013.

[15] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu. Querying k-truss community in large and dynamic graphs. *SIGMOD*, 2014.

[16] R. Li, J. X. Yu, and R. Mao. Efficient core maintenance in large dynamic graphs. *IEEE Trans. Knowl. Data Eng.*, 26(10):2453–2465, 2014.

[17] M. C. Lin, F. J. Soulignac, and J. L. Szwarcfiter. Arboricity, h-index, and dynamic algorithms. *Theor. Comput. Sci.*, 426:75–90, 2012.

[18] J. Moody and D. R. White. Structural cohesion and embeddedness: A hierarchical concept of social groups. *American Sociological Review*, 68:103–127, 2003.

[19] K. Saito and T. Yamada. Extracting communities from complex networks by the k-dense method. In *ICDM Workshops*, 2006.

[20] A. E. Sariyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek. Streaming algorithms for k-core decomposition. *PVLDB*, 6(6):433–444, 2013.

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

[22] M. Sozio and A. Gionis. The community-search problem and how to plan a successful cocktail party. In *KDD*, 2010.

[23] J. Ugander, L. Backstrom, C. Marlow, and J. Kleinberg. Structural diversity in social contagion. *PNAS*, 2011.

[24] J. Wang and J. Cheng. Truss decomposition in massive networks. *PVLDB*, 5(9):812–823, 2012.

[25] N. Wang, J. Zhang, K.-L. Tan, and A. K. H. Tung. On triangulation-based dense neighborhood graphs discovery. *PVLDB*, 4(2):58–68, 2010.

[26] J. Xie, S. Kelley, and B. K. Szymanski. Overlapping community detection in networks: The state-of-the-art and comparative study. *ACM Comput. Surv.*, 45(4):43, 2013.

[27] Y. Zhang and S. Parthasarathy. Extracting, analyzing and visualizing triangle k-core motifs within networks. In *ICDE*, 2012.

[28] F. Zhao and A. K. H. Tung. Large scale cohesive subgraphs discovery for social network visual analysis. *PVLDB*, 6(2):85–96, 2012.

[29] R. Zhou, C. Liu, J. X. Yu, W. Liang, B. Chen, and J. Li. Finding maximal k-edge-connected subgraphs from a large graph. In *EDBT*, 2012.