Data Science and Engineering (2021) 6:310–322
https://doi.org/10.1007/s41019-021-00163-3

Efficient Personalized Influential Community Search in Large Networks

Yanping Wu[1] · Jun Zhao[1] · Renjie Sun[1] · Chen Chen[1] · Xiaoyang Wang[1]

Received: 20 January 2021 / Revised: 12 March 2021 / Accepted: 18 April 2021 / Published online: 29 April 2021

Abstract

Community search, which aims to retrieve important communities (i.e., subgraphs) for a given query vertex, has been widely studied in the literature. In the recent, plenty of research is conducted to detect influential communities, where each vertex in the network is associated with an influence value. Nevertheless, there is a paucity of work that can support personalized requirement. In this paper, we propose a new problem, i.e., maximal personalized influential community search. Given a graph G, an integer k and a query vertex u, we aim to obtain the most influential community for u by leveraging the k-core concept. To handle larger networks efficiently, two algorithms, i.e., top-down algorithm and bottom-up algorithm, are developed. In real-life applications, there may be a lot of queries issued. Therefore, an optimal index-based approach is proposed in order to meet the online requirement. In many scenarios, users may want to find multiple communities for a given query. Thus, we further extend the proposed techniques for the top-r case, i.e., retrieving r communities with the largest influence value for a given query. Finally, we conduct extensive experiments on 6 real-world networks to demonstrate the advantage of proposed techniques.

Keywords Influential community · Personalized search · k-core · Top-r

1 Introduction

Retrieving communities and exploring the latent structures in the networks can find many applications in different fields, such as protein complex identification, friend recommendation, event organization, etc. [9, 11]. There are two essential problems in community retrieval, that is, community detection and community search. Generally, given a graph, community detection problem aims to find all or top-r communities from the graph [13, 18], while community search problem is to identify the cohesive communities that contain the given query vertex [7, 20]. In this paper, we focus on the category of community search problem, which is very important for personalized applications. For instance, we can conduct better friend recommendation by identifying the important community that contains the query users. Similarly, we can make better event organization by retrieving the community, which contains the user that we want to invite.

In the literature, lots of research tries to find personalized communities by emphasizing the structure cohesiveness, while, in many cases, we also need to consider the influence of obtained communities. Recently, there are some research that tries to find communities with large influence, e.g., [2, 3, 16]. In [16], Li et al. propose a novel community model called k-influential community, where each vertex is associated with a weight (i.e., influence value) in the graph. A community (i.e., subgraph) is essential when it is cohesive and has large influence value. Efficient algorithms are developed to obtain the top-r communities with the largest influence value. Given the importance of the problem, [2, 3] try to speed up the search from different aspects. Since influence value is user's natural property, by considering the influence value, it can lead us to identify more significant communities.

Nevertheless, the existing works on influential community detection mainly focus on finding all or top-r influential

✉ Xiaoyang Wang
xiaoyangw@zjgsu.edu.cn

Yanping Wu
yanpingw.zjgsu@gmail.com

Jun Zhao
junzhao.zjgsu@gmail.com

Renjie Sun
renjiesun.zjgsu@gmail.com

Chen Chen
chenc@zjgsu.edu.cn

[1] Zhejiang Gongshang University, Hangzhou, China

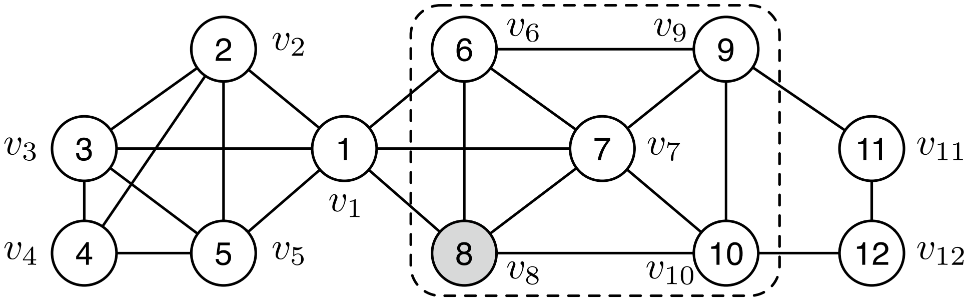


Fig. 1 Running example (the number in the vertex denotes its weight)

communities. The personalized situation is not considered. To fill this gap, in this paper, we propose the maximal personalized influential community (MPIC) search problem. Given a graph G, an integer k and a query vertex q, the MPIC is the community with the largest influence value that contains q, and satisfies the k-core (i.e., the degree of each vertex inside is no less than k), connectivity (i.e., the subgraph is connected) and maximal (i.e., no other supergraph satisfies the previous criteria) constraints. As defined in the previous work [16], the influence value of a community is the minimum weight of all the vertices in the community. Given the graph in Fig. 1, if $k = 3$ and the query vertex is v_8, then the vertices in the dotted line is the corresponding MPIC. Note that, the k-core model is also used in the previous works to measure the cohesiveness of the community [2, 3, 16]. In real applications, users may require to retrieve multiple communities for a given query. The top-r model is widely adopted for this scenario. Therefore, in this paper, we also investigate the top-r case, denoted as TPIC, i.e., finding r communities with the largest influence value, which can facilitate analyzing the significance or global properties for the query vertex. In addition, users can make a trade-off between the community influence and community size.

Challenges The main challenges of the problem lie in the following two aspects. Firstly, the real-world networks, such as social networks, are usually large in size. It is critical for the algorithms to scale for large networks. Secondly, since we investigate the personalized scenario, there may be plenty of queries generated by users in real applications, it is important that the developed algorithms can meet the online requirements.

Contributions To the best of our knowledge, we are the first to investigate the maximal personalized influential community (MPIC) search and top-r personalized influential community (TPIC) search problems. The contributions of this paper are summarized as follows.

- We formally define the MPIC and TPIC search problems.
- For the MPIC search problem, to handle large networks, two algorithms, i.e., top-down algorithm and the bottom-up algorithm, are developed based on different searching orders.
- An optimal index-based method is further proposed for the MPIC search problem in order to meet the online requirements.
- To meet the requirement in real applications, we also extend the proposed techniques to support the top-r scenarios, i.e., TPIC search problem.
- We conduct extensive experiments on 6 real-world networks to evaluate the performance of proposed techniques. As shown, the developed techniques can significantly speed up the search compared with the baseline.

Roadmap The rest of the paper is organized as follows. In Sect. 2, we briefly introduce the related concepts and the problems studied. In Sect. 3, three approaches are proposed to process the maximal personalized influential community search problem. In Sect. 3.4, we conduct the extension to further support the top-r personalized influential community search problem. We demonstrate the efficiency and effectiveness of proposed techniques in Sect. 5. Lastly, we introduce the related works in Sect. 6 and conclude the paper in Sect. 7.

2 Problem Definition

We consider a network $G = (V, E, \omega)$ as an undirected graph, where V and E denote the vertex set and edge set, respectively. Each vertex $u \in V$ is associated with a weight denoted by $\omega(u)$, representing the influence of vertex u. The vertex weight can be its PageRank score or other user defined value. Without loss of generality, we use the same setting as the previous work for vertex weight, where different vertices have different weights [16]. Note, if that is not the case, we use the vertex id to break the tie. We denote the number of vertices by $n = |V|$ and the number of edges by $m = |E|$. A subgraph $S = (V_S, E_S)$ is an induced subgraph of G, if $V_S \subseteq V$ and $E_S = \{(u, v) | u, v \in V_S, (u, v) \in E\}$. Given a subgraph S, the neighbors of $u \in V_S$ is denoted by $N(u, S) = \{v | v \in V_S, (u, v) \in E_S\}$, and $deg(u, S)$ represents the degree of u in S, i.e., $deg(u, S) = |N(u, S)|$. In this paper, we utilize the k-core model to represent the cohesiveness of a community, which is also widely used in the literature [2, 16].

Definition 1 (*k-core*) Given a graph G and a positive integer k, a subgraph $S \subseteq G$ is the k- core of G, denoted by $C_k(G)$, if S satisfies the following conditions. (*i*) $deg(u, S) \geq k$ for each vertex u in S. (*ii*) S is maximal, i.e., any subgraph $S' \supset S$ is not a k-core.

To compute the k-core of a graph, we can remove the vertex whose degree is less than k recursively. The time complexity of computing k-core is $O(m)$ [26], and the detailed algorithm is shown in Algorithm 1. To identify important communities, we consider both the cohesiveness and the influence of a community. We employ the widely used influence value to measure the influence of a community [2, 16].

```
Algorithm 1: ComputeCore($G$, $k$)
Input  : $G$ : a graph, $k$ : degree constraint
Output : $k$-core of $G$
while exists $u \in G$ with $deg(u, G) < k$ do
    $G = G \setminus u$ ;
return $G$
```

Definition 2 (*Influence value*) Given an induced subgraph S of G, the influence value of S is the minimum weight of the vertex in V_S, denoted as $f(S)$, i.e., $f(S) = \min_{u \in V_S} \omega(u)$.

In the previous works, people usually focus on finding all or top-r influential communities [2, 3, 16], while, as discussed, in real applications, it is also essential to identify the personalized influential communities for different user queries. Given this requirement, we define the personalized influential community as follows.

Definition 3 (*Personalized Influential Community*) Given a graph G, a positive integer k and a query vertex q, a personalized influential community is an induced subgraph S of G, which meets all the following constraints.

- Connectivity: S is connected;
- Cohesiveness: each vertex in S has degree at least k;
- Personalized: query vertex q is contained in S, i.e., $q \in V_S$;
- Maximal: there is no other induced subgraph S' that (*i*) satisfies the first three constraints (i.e., connectivity, cohesiveness and personalized constraints), (*ii*) is a supergraph of S, i.e., $S' \supset S$, and (*iii*) has the same influence value as S, i.e., $f(S) = f(S')$;

Definition 4 (*Maximal Personalized Influential Community MPIC*) Given a graph G, a positive integer k and a query vertex q, the $\underline{m}$ aximal $\underline{p}$ersonalized $\underline{i}$ nfluential $\underline{c}$ ommunity, short as **MPIC**, is the personalized influential community with the largest influence value.

Definition 5 (*Top-r Personalized Influential Community (TPIC)*) Given a graph G, two integers k, r and a query vertex q, the $\underline{t}$ op-r $\underline{p}$ ersonalized $\underline{\text{i}}$ nfluential $\underline{c}$ ommunities, short as **TPIC**, is the r personalized influential communities with the largest influence value.

Problem Statement 1 Given a graph $G = (V, E, \omega)$, a query vertex q and a positive integer k, we aim to develop efficient algorithm to find the MPIC for the query, denoted by **MPIC(**q **,** k**)**.

Problem Statement 2 Given a graph $G = (V, E, \omega)$, a query vertex q, and two positive integers k, r, we aim to develop efficient algorithms to find the TPIC for the query, denoted by **TPIC(**q **,** k **,** r**)**.

Example 1 As shown in Fig. 1, the number in each vertex is the corresponding weight. Suppose $k = 3$ and query vertex is v_8. Then, we can see that the subgraph $S_1 = \{v_6, v_7, v_8, v_9, v_{10}\}$ in the dotted line is the corresponding MPIC with influence value of 6. While the subgraph $S_2 = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}\}$, which satisfies the first four constraints of MPIC with influence value of 1, is not the MPIC, because it is not the one with the largest influence value. For the top-r problem, suppose $r = 2$, then S_1 and S_2 are the two communities returned.

3 Solutions for MPIC Problem

In this section, we first introduce some properties about the personalized influential community. Then, we develop two approaches: top-down method and bottom-up method by verifying the vertices in different orders. Finally, to support efficient online processing and scale for large networks, an index-based method is proposed based on the bottom-up framework.

3.1 Properties of Personalized Influential Community

Lemma 1 *Given a graph G, an integer k and a query vertex q, then the influence value of MPIC(q, k) is at most the weight of q, i.e., $f(\text{MPIC}(q, k)) \leq \omega(q)$.*

Proof MPIC(q, k) must contain q. Based on the definition of influence value, we have

$f(\text{MPIC}(q,k)) = \min_{u \in \text{MPIC}(q,k)} \omega(u) \leq \omega(q)$. Thus, the lemma holds. □

Lemma 2 *Given a graph G and two induced subgraphs S_1 and S_2, we have $V_{S_2} \subset V_{S_1}$ and $V_{S_1} = V_{S_2} \cup \{u\}$.If the weight of u is smaller than the influence value of S_2 (i.e., $\omega(u) < f(S_2)$), then the influence value of S_1 is smaller than that of S_2 (i.e., $f(S_1) < f(S_2)$).*

Proof Based on the definition of influence value, $f(S_1) = \min_{v \in V_{S_1}} \omega(v) \leq \omega(u) < f(S_2)$. Therefore, the lemma holds. □

3.2 Top-Down Algorithm for MPIC

In this section, we present the top-down algorithm which is inspired by the existing influential community detection method [2]. According to Lemma 1, the influence value of the identified MPIC is at most $\omega(q)$. To find the community with the largest influence value, we can first add all the vertices whose weight is no less than $\omega(q)$ and check if we can obtain a community that satisfies the first four constraints of MPIC. If so, we can output the identified community. Otherwise, we can add some vertices with weight smaller than $\omega(q)$ to find the MPIC. The detailed algorithm is shown in Algorithm 2.

```
Algorithm 2: Top-Down Algorithm
   Input  : G : a weighted graph, k : degree constraint,
            q : query vertex
   Output : MPIC for the query
 1 $C_k(G) \leftarrow$ ComputeCore($G$, $k$);
 2 if $q \notin C_k(G)$ then
 3     return error
 4 $C_k(G,q) \leftarrow$ the connected component of $C_k(G)$ that
   contains $q$;
 5 $S \leftarrow$ sort vertices of $C_k(G,q)$ in descending order
   based on vertex weights;
 6 $Q \leftarrow \emptyset$; $i \leftarrow 0$;
 7 while $i < S$.size do
 8     $Q \leftarrow Q \cup \{S[i]\}$;
 9     If $S[i] = q$ then break;
10     $i \leftarrow i + 1$;
11 if $q \in C_k(Q)$ then
12     return the connected component containing $q$ in
       $C_k(Q)$
13 $i \leftarrow i + 1$;
14 while $i < S$.size do
15     $Q \leftarrow Q \cup \{S[i]\}$;
16     if $q \in C_k(Q)$ then
17         return the connected component containing $q$
           in $C_k(Q)$
18     $i \leftarrow i + 1$;
```

In Algorithm 2, we first compute the k-core of G, denoted by $C_k(G)$. Since the MPIC must be inside $C_k(G)$, if q does not belong to the k-core, error code is returned in Line 3, which means we cannot find a MPIC containing q. Otherwise, due to the connectivity constraint, we only need to keep the connected component that contains q. Then we sort the survived vertices in descending order by their weights and store them in S (Lines 4–5). We load the query q and the vertices ranked above q into Q (Lines 7–10). If the k-core $C_k(Q)$ of Q contains q, then we return the connected component containing q, which can be obtained by conducting a BFS from q (Lines 11–12). Otherwise, we add the remaining vertices in S one by one to Q until the k-core of Q contains the query vertex q, and the connected component is returned (Lines 14–18).

```
Algorithm 3: Bottom-Up Algorithm
Input  : G : a weighted graph, k : degree constraint,
         q : query vertex
Output : MPIC for the query
$C_k(G) \leftarrow$ ComputeCore($G$, $k$);
if $q \notin C_k(G)$ then
    return error
$C_k(G,q) \leftarrow$ the connected component of $C_k(G)$ that
contains $q$;
$S \leftarrow$ sort vertices of $C_k(G,q)$ in ascending order based
on vertex weights;
while $S \neq \emptyset$ do
    $D \leftarrow \emptyset$;
    $u \leftarrow S.front()$;
    if Delete$(u, q, S, D) = 1$ then
        return $S \cup D$

Procedure Delete$(u, q, S, D)$;
initialize a queue $R = \{u\}$;
while $R \neq \emptyset$ do
    $w \leftarrow R.pop()$;
    if $w = q$ then
        return 1
    for each $v \in N(w, S)$ do
        $deg(v, S) \leftarrow deg(v, S) - 1$;
        if $deg(v, S) < k$ then
            $R.push(v)$;
    remove $w$ from $S$;
    $D \leftarrow D \cup \{w\}$;
for each connected component $S'$ in $S$ do
    if $q \notin S'$ then
        remove $S'$ from $S$;
return 0
```

Example 2 Consider the graph in Fig. 1. Suppose $k = 3$ and the query vertex is v_8. Following the top-down algorithm, we first compute the k-core of G. Thus, v_{11} and v_{12} are deleted, because they violate the degree constraint. Then, we add v_8 and the vertices ranked higher (i.e., $\{v_{10}, v_9\}$) than v_8 into Q. However, they cannot form a 3-core. Then, we insert vertex one by one into Q. Until v_6 is added, there is a 3-core containing query v_8, i.e., $\{v_{10}, v_9, v_8, v_7, v_6\}$, which is the MPIC returned.

3.3 Bottom-Up Algorithm for MPIC

In the top-down algorithm, we first add all the vertices ranked higher than q into Q. After that, by adding each vertex into Q, we need to invoke the k-core computation procedure. Even though the time complexity of k-core computation is $O(m)$, in the worst case, we need to repeat the process n times, which can be time-consuming. Ideally, we can add more vertices into Q for each iteration. However, in order to guarantee the correctness of the algorithm, it is difficult to determine the appropriate number of vertices to be added. If too many vertices are added, we may need a lot of computations to shrink the result. Otherwise, we still need to compute the k-core plenty of times. To reduce the computation cost, in this section, the bottom-up method is proposed, which can avoid computing the k-core repeatedly.

According to Lemma 2, for a given induced subgraph, we can increase its influence value by removing the vertex with the smallest weight. Intuitively, since we aim to find the MPIC, we can iteratively remove the vertices with the smallest weight and keep tracking the other constraints of MPIC, until the MPIC is found. Different from the top-down approach, in the bottom-up method, we visit the vertices in ascending order and remove the unpromising vertices iteratively. The detailed algorithm is shown in Algorithm 3.

For the algorithm, the first three steps are exactly the same as the top-down method (Lines 1–4). Then, we sort the survived vertices in ascending order by the weight of vertex and store them in S (Line 5). Then, we try to remove the vertex with the current smallest weight one by one until the query vertex q is met (Lines 6–10). For each vertex u processed, we invoke the Delete procedure, which details are shown in Lines 11–26. For each processed vertex u, we need to ensure the remained subgraph satisfies the k-core constraint. After deleting a vertex, it may cause its neighbors to have less than k neighbors. Then we remove these vertices as well (Lines 17–20). We put the vertices that violate the degree constraint into R and process them iteratively. When $w = q$ (Line 15), it means either (*i*) the input vertex u of Delete procedure is q, or (*ii*) $deg(q, S)$ becomes less than k because of the deletion u. In this case, the remained subgraph S and D (i.e., $S \cup D$) form the MPIC. This is because, when we remove the input vertex u, it will cause the remained subgraph does not contain q or q violates the degree constraint. The reason that we keep tracking the deleted vertices D for each Delete procedure is for case when *ii* situation happens. Since the identified community should satisfy the connectivity constraint, we can safely remove the connected components in S that do not contain q (Lines 23–25).

Example 3 Consider the graph in Fig. 1. Suppose $k = 3$ and the query vertex is v_8. Following the bottom-up approach, v_{11} and v_{12} are firstly deleted due to the k-core computation. After deleting the vertex v_1 with the smallest weight, the remained graph are separated into two connected components. Therefore, we can safely remove the connected component $\{v_2, v_3, v_4, v_5\}$ from S since it does not contain the query vertex. Then, we process v_6. As we can see, when processing v_6 in the Delete procedure, it will result in v_8 violating the degree constraint. Then, we can stop and output $\{v_6, v_7, v_8, v_9, v_{10}\}$ as the result.

3.4 Index-Based Algorithm for MPIC

In the bottom-up approach, we invoke the k-core computation at the beginning of the algorithm and the total cost of checking degree constraint in Delete only takes $O(m)$ time, which avoids lots of computations compared to the top-down method. However, the bottom-up approach still has some limitations. (*i*) When deleting the vertices, it still costs a lot for processing very large graphs. (*ii*) For real applications, different users may have different requirements and there may exist a large amount of queries. Therefore, it is hard for it to meet the online requirement.

```
Algorithm 4: Index Construction
Input  : $G$ : a weighted graph
Output : constructed index
for $k$ from 1 to $k_{max}$ do
    $C_k(G) \leftarrow$ ComputeCore($G$, $k$);
    $rn \leftarrow$ empty root node of $T_k$;
    for each connected component $S$ in $C_k(G)$ do
        BuildNode($k, rn, S$);

Procedure BuildNode($k, rn, S$);
$u \leftarrow$ the vertex with the smallest weight in $S$;
$R \leftarrow \{u\}$; $D \leftarrow \emptyset$;
while $R$ is not empty do
    $w \leftarrow R.pop()$;
    for each $v \in N(w, S)$ do
        $deg(v, S) \leftarrow deg(v, S) - 1$;
        if $deg(v, S) < k$ then
            $R.push(v)$;
    remove $w$ from $S$;
    $D \leftarrow D \cup \{w\}$;
construct an intermediate node $crn$ containing $D$;
append $crn$ to the parent node $rn$;
for each connected component $S'$ in $S$ do
    BuildNode($k, crn, S'$);
```

Motivated by the requirements, in this section, we propose an index-based algorithm by leveraging the bottom-up framework. In the bottom-up method, for a given k, we try to delete the vertex u with the smallest weight in each iteration by Delete procedure. Then, we can obtain the MPIC for certain vertices, such as the vertex u and the vertices removed when processing u. If we process the vertices in order, we can obtain the MPICs for all the vertices. Therefore, we can build a tree structure index according to the processing order. Let $k_{\max}$ be the largest core number, i.e., the largest k value for any k-core. If we build a tree index for each k value, then we can answer any given query efficiently. In Algorithm 4, we present the details of how to index the visited vertices effectively. Then, we show how to answer a query by utilizing the index.

Index construction In Algorithm 4, we build a tree index for each k value from 1 to $k_{\max}$ (Lines 1–5). In each iteration, we first compute the corresponding k-core, and for each connected component, we construct the indexed tree nodes by invoking BuildNode procedure. The details of BuildNode procedure are shown in Lines 6–20. The BuildNode procedure is very similar to the Delete procedure in Algorithm 3. It starts by processing the vertex with the smallest weight (Line 7). When we process a vertex u, it will cause some other vertices violating the degree constraints (Lines 11–14) and we add them and u into D (Line 16). According to the bottom-up method, it means the vertices in D belong to the same MPIC. Then, we construct an intermediate node *crn* that contains the vertices in D, and append it to its parent node *rn* (Lines 17–18). Then, we recursively call the BuildNode to process each connected component S' of the remained subgraph S (Lines 19–20). After processing each k, the index is constructed. Based on the construction process, we can see that the MPIC of a vertex consists of its belonged intermediate node and its children nodes in the index.

Example 4 Figure 2 shows the constructed index for the graph in Fig. 1 when $k = 1, 2, 3$. The *ER* node is the empty root node. It is for the case when the computed k-core in Line 2 of Algorithm 4 is not connected. For $k = 1$, the constructed index is shown in Fig. 2a. We first process v_1 which will result in 2 connected components. Then, we remove v_2 and v_3 and create two intermediate nodes for them, since the removal of them does not make other vertices violate the degree constraint. When deleting v_4, the degree of v_5 becomes less than 1. Then, we construct an intermediate node that contains v_4 and v_5. We conduct similar procedure for the other connected component, and the constructed index is shown in the right branch. Similar procedure is conducted for $k = 2, 3$, where the corresponding index are shown in Fig. 2b, c.

Query processing As we can see, for a given query, the MPIC consists of the intermediate node that contains the query vertex and all its children nodes in the corresponding k index. If we maintain $k_{\max}$ pointers for each vertex to its corresponding intermediate nodes, we can efficiently locate the vertex's intermediate node for a given k and traverse the index to return the result. For a given query, if we cannot find its intermediate node in the index, it means it does not has a MPIC for the query.

Example 5 Consider the graph in Fig. 1. The constructed index is shown in Fig. 2 for $k = 1, 2, 3$. Given the query vertex v_8, the MPIC is the vertices in the dotted line for $k = 1, 2, 3$, respectively.

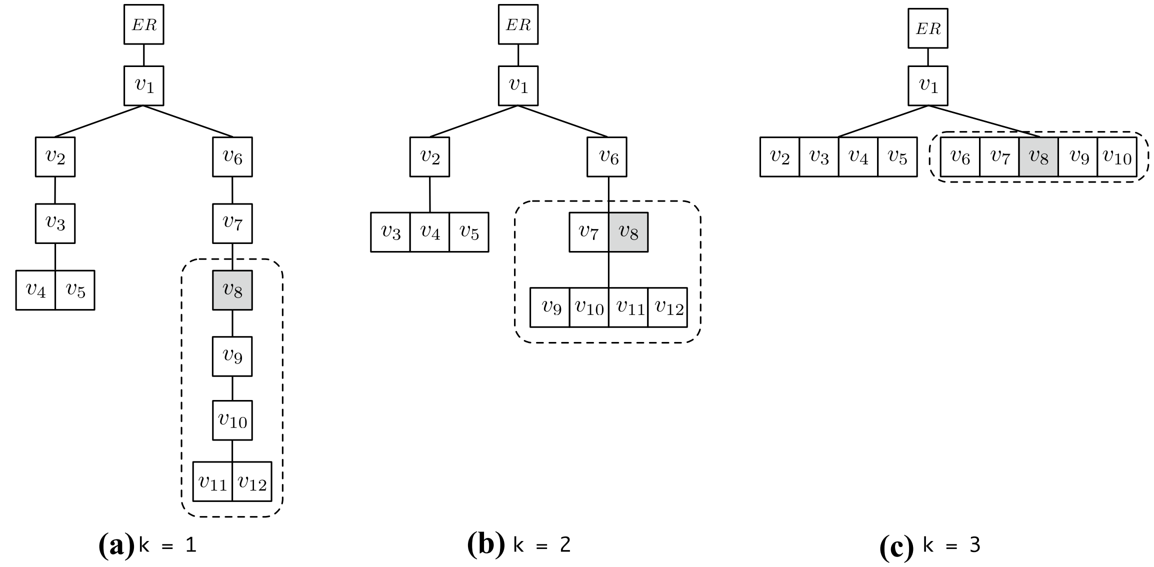
Discussion If we do not need to retrieve the specific vertices in MPIC, the index can answer the query in $O(1)$ time by just returning the pointer for the intermediate node. Otherwise, we need to traverse from the intermediate node to obtain all the vertices. In this paper, we use the second case in the experiments, since the first two algorithms will obtain all the vertices in MPIC. For a given query, we need to at least access all the output data once. Therefore, the index-based method is optimal.

4 Solutions for TPIC Problem

In real applications, top-r queries are widely adopted to provide users with more choices. In this section, we investigate the top-r personalized influential community (TPIC) problem, and extend the proposed solutions for MPIC problem to support the top-r cases. Note that, the MPIC problem is a special case of TPIC when r equals 1.

Algorithm 5: Top-Down-R Algorithm

```
Input  : $G$ : a weighted graph, $k$ : degree constraint,
         $r$ : number constraint, $q$ : query vertex
Output : TPIC for the query
$C_k(G) \leftarrow$ ComputeCore($G$, $k$);
if $q \notin C_k(G)$ then
    return error
$C_k(G, q) \leftarrow$ the connected component of $C_k(G)$ that
contains $q$;
$S \leftarrow$ sort vertices of $C_k(G, q)$ in descending order
based on vertex weights;
initialize an empty queue $C$;
$Q \leftarrow \emptyset$; $i \leftarrow 0$; $j \leftarrow 0$;
while $i < S.\text{size}$ do
    $Q \leftarrow Q \cup \{S[i]\}$;
    if $S[i] = q$ then
        break;
    $i \leftarrow i + 1$;
while $i < S.\text{size}$ and $j < r$ do
    if $q \in C_k(Q)$ then
        Let $C_k^j(Q)$ be the connected component
        containing $q$ in $C_k(Q)$;
        if $|C_k^j(Q)| > |C.rear|$ then
            $C.push(C_k^j(Q))$;
            $j \leftarrow j + 1$;
    $Q \leftarrow Q \cup \{S[i]\}$;
    $i \leftarrow i + 1$;
if $j < r - 1$ then
    return error
return $C$
```



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

Fig. 2 Example for Index Construction

4.1 Top-Down Algorithm for TPIC

For the top-down-based searching framework, in order to support top-*r* queries, we need to conduct further exploration to find *r* results. The algorithm details are shown in Algorithm 5.

The first four steps are the same as Algorithm 2. We initialize an empty queue *C* to store *r* personalized influential communities in Line 6 and an integer *j* to record the number of communities identified. Then, we add all the vertices whose weight is no less than $\omega(q)$ into *Q* in Lines 8–12. The main difference is the process of checking and obtaining top-*r* personalized influential communities in Lines 13–23. When the *k*-core $C_k(Q)$ of *Q* contains *q*, we use $C_k^j(Q)$ to store the connected component containing *q* in $C_k(Q)$ (Lines 14–15). Based on the formal definition of the queue, the first community is inserted from one end called the rear, and the removal of the existing community takes place from the other end called as front. We use *C*.rear to denote the latest community inserted in *C*. If $|C_k^j(Q)| > |C.rear|$, it means the current community is not the same as the community obtained in the previous step. Then, we push it into *C* (Lines 16–18). Next, we iteratively add the remaining vertices from *S* into *Q* to obtain the connected components containing *q* (Line 19). The iteration is terminated until we find *r* communities or all the vertices have been visited. Finally, we return top-*r* result set *C* if *r* communities have been retrieved in the process. Otherwise, we return an error code.

Algorithm 6: Bottom-Up-R Algorithm

```
Input  : $G$ : a weighted graph, $k$ : degree constraint,
         $r$ : number constraint, $q$ : query vertex
Output : TPIC for the query
$C_k(G) \leftarrow$ ComputeCore($G$, $k$);
if $q \notin C_k(G)$ then
    return error
$C_k(G,q) \leftarrow$ the connected component of $C_k(G)$ that
contains $q$;
$S \leftarrow$ sort vertices of $C_k(G,q)$ in ascending order based
on vertex weights;
initialize an empty queue $C$;
$C.push(S)$;
while $S \neq \emptyset$ do
    $D \leftarrow \emptyset$;
    $u \leftarrow S.front()$;
    if Delete$(u, q, S, D) = 1$ then
        if $|C| = r$ then
            $C.pop()$;
            $C.push(S \cup D)$;
            return $C$
        else
            return error
    else
        if $|C| = r$ then
            $C.pop()$;
        $C.push(S)$;
```

4.2 Bottom-Up Algorithm for TPIC Problem

In the top-down-based searching framework, we need to invoke the k-core computation procedure many times. Even though we can compute the k-core in linear time, it still means a lot of computations. Therefore, we extend the bottom-up search framework to reduce the cost. The detailed algorithm is shown in Algorithm 6.

In the algorithm, we initialize an empty queue C to store top-r communities in Line 6. We add the connected component of $C_k(G)$ that contains q into C in Line 7, which is the personalized influential community with the smallest influence value. Then, we iteratively invoke the Delete procedure, and keep maintaining the current best r results. The details of Delete procedure are shown in Lines 11–26 of Algorithm 3. Particularly, if $\text{DELETE}(u, q, S, D) = 1$, it means we reach the query vertex and no more vertices should be removed. If there are already r communities in C, we first pop the community with minimum influence value from C, then we push the community currently found in C. An error code is returned, if we cannot find r results.

4.3 Index-Based Algorithm for TPIC

To meet the online requirement, we extend the index-based algorithm to solve the TPIC problem. Note that the index structure is the same as that for the MPIC problem. To computer the top-r communities for a given query, we first locate the corresponding index tree and the intermediated node that contains the query vertex. Then, we iteratively find its $r - 1$ parent nodes. Finally, we return all the vertices in the r subtree rooted at the identified intermediated node and parent nodes as the results. The correctness can be easily verified based on the definition of the personalized influential community and index structure.

Example 6 Reconsider the graph in Fig. 1. The constructed index is the same as that for MPIC problem, i.e., Fig. 2. Given the query vertex v_8, $k = 2$ and $r = 2$, we first locate the tree node where v_8 is located in Fig. 2b and return the vertices set in the subtree rooted at this tree node as top-1 result, i.e., $\{v_7, v_8, v_9, v_{11}, v_{12}\}$, and then we return the vertices set in the subtree rooted at the parent node of this tree node as top-2, i.e., $\{v_6, v_7, v_8, v_9, v_{11}, v_{12}\}$.

Discussion In this paper, we investigate the case, where each vertex has a different weight. It is also the most widely adopted settings in the literature. However, our method can also be easily extended to support the case, where vertices have same weight. Specifically, in the top-down algorithm, when we process some vertices with the same weight, we add them together into the candidate set and then check the result. Similarly, in the bottom-up algorithm, when there are many vertices with the current smallest influence value, we need to delete them together and then do the remaining procedure as the original algorithm. The index-based algorithm is based on the bottom-up algorithm. Hence, different from the original index construction algorithm that we iteratively delete the vertex with the smallest weight, we only need to modify it by removing all the vertices with the same smallest weight at the same time and putting them with their followers into the same tree node.

5 Experiments

In this section, we conduct extensive experiments on real-world networks to evaluate the performance of proposed techniques.

5.1 Experiment Setup

Algorithms Since there is no previous work for the proposed problem, we conduct experiments with the proposed algorithms.

- **Top-down**. Algorithm proposed in Section 3.2 for the MPIC problem.
- **Bottom-up**. Algorithm proposed in Section 3.3 for the MPIC problem.
- **Index-based**. Algorithm proposed in Section 3.4 for the MPIC problem.
- **Top-down-r**. Algorithm proposed in Section 4.1 for the TPIC problem.
- **Bottom-up-r**. Algorithm proposed in Section 4.2 for the TPIC problem.
- **Index-based-r**. Algorithm proposed in Section 4.3 for the TPIC problem.

The top-down and top-down-r algorithms serve as the baseline methods for the corresponding problems.

Datasets We evaluate the algorithms on 6 real-world datasets, i.e., Email, Brightkite, Gowalla, YouTube, Wiki and Livejournal. Table 1 shows the statistic details of the datasets. The datasets are downloaded from the Stanford Network Analysis Platform,[1] which are public available. Similar as previous work, we use the PageRank value to serve as the vertex weight [16].

Parameter and workload To evaluate the performance of proposed techniques, we vary the weight of query vertex and k. To generate the query vertices, we sort the

[1] http://snap.stanford.edu.

vertices according to the weight and divide them into 5 buckets. For each bucket, we randomly select 200 vertices as query vertices. For k, we vary k from 5 to 25 with **10** as the default value. For r, we vary r from 5 to 25 with **10** as the default value. For each setting, we run the algorithms 10 times and report the average response time.

All algorithms are implemented in C++ with GNU GCC 7.4.0. Experiments are conducted on a PC with Intel Xeon 3.2GHz CPU and 32GB RAM using Ubuntu 18.04 (64-bit).

Table 1 Statistics of datasets

Dataset	# Vertices	#Edges	$d_{\max}$	$k_{\max}$
Email	36,692	183,831	1367	43
Brightkite	58,228	214,078	1134	52
Gowalla	196,591	950,327	14,730	51
YouTube	1,134,890	2,987,624	28,754	51
Wiki	1,791,488	13,846,826	16,063	72
Livejournal	3,997,962	34,681,189	14,815	360

5.2 Experiment Results of Index Construction

We first present the index construction time for all datasets, the results are shown in Fig. 3. As we can observe, the index construction phase is very efficient. It only takes 0.290 seconds for Brightkite dataset. For the largest network Livejournal, which has more than 34 million edges, it only takes 325.656 seconds for constructing the index.

5.3 Experiment Results for MPIC Problem

Results of varying query vertex weight By varying the query vertex weight, we conduct the experiments on all the datasets. The response time is shown in Fig. 4, where k is set as the default value. As observed, the bottom-up method is much faster than the top-down method, since the top-down method may compute the k-core many times. Among all, the index-based method runs fastest, due to the novel index structure proposed. For the top-down method, the response time increases when the weight increases. This is because, for query vertex with larger weight, it may compute the k-core more times when adding vertices one by one.

Results of varying k We conduct the experiments on all the datasets by varying the query parameter k. The results of response time are shown in Fig. 5, where similar trend can be observed. The bottom-up and index-based methods are significantly faster than the top-down method, and the index-based method is the fastest one for all cases. With the increase of k, the response time of top-down method decreases. This is because, for larger k, the identified MPIC tends to be smaller.

5.4 Experiment Results for TPIC Problem

Results of varying query vertex weight In Fig. 6, we report the experiment results of TPIC on all the datasets by vary the vertex weight, where r and k are set as the default value. As we can observe, index-based-r is much faster than top-down-r and bottom-up-r. In the two largest datasets, i.e., Wiki and Livejournal, the index-based-r method achieves up to 5 and 6 orders of magnitudes speedup compared with the top-down-r method, respectively. As the weight increases, the processing time of Top-down-r increases significantly, because it will invoke more k-core computation procedure, while the bottom-up-r and index-based-r methods are not sensitive to the weight of query vertex.

Results of varying k In Fig. 7, we report the experiment results of TPIC on all the datasets by vary k, where r is set as the default value. Similar trend can be found. Index-based-r is the fastest method among the three approaches. In the largest dataset, i.e., Livejournal, the Index-based-r method can achieve up to 6 orders of magnitudes speedup compared with the top-down-r method. As k increases, the processing time decreases, since larger k will lead to smaller communities. With the increase of k, the response time of top-down-r method decreases. This is because, for larger k, the identified communities tend to be smaller. As shown, the bottom-up-r and index-based-r methods can scale well for the parameter k.

Results of varying r To further evaluate the techniques proposed, in Fig. 8, we report the experiment results of TPIC by varying r, and the other parameters are set as the default value. As we can see, index-based-r still dominate the other two methods. When r increases, the processing time increases for all the methods. This is because, for lager r, we need to retrieve more results.

5.5 Summary

As demonstrated in the experiments, for both problems, i.e., MPIC and TPIC, the bottom-up and index-based methods are significantly faster than the top-down method, and they can scale well for different query parameters. Especially for the index-based method, it usually can achieve orders of magnitudes speedup, and is efficient enough for most online requirements. As the index-based approach requires additional space to store the index structure, users can select the appropriate strategies when facing the real scenarios.

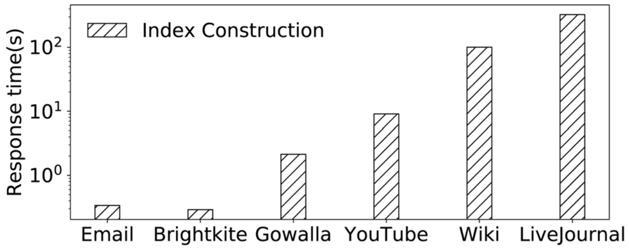


Fig. 3 Index construction time

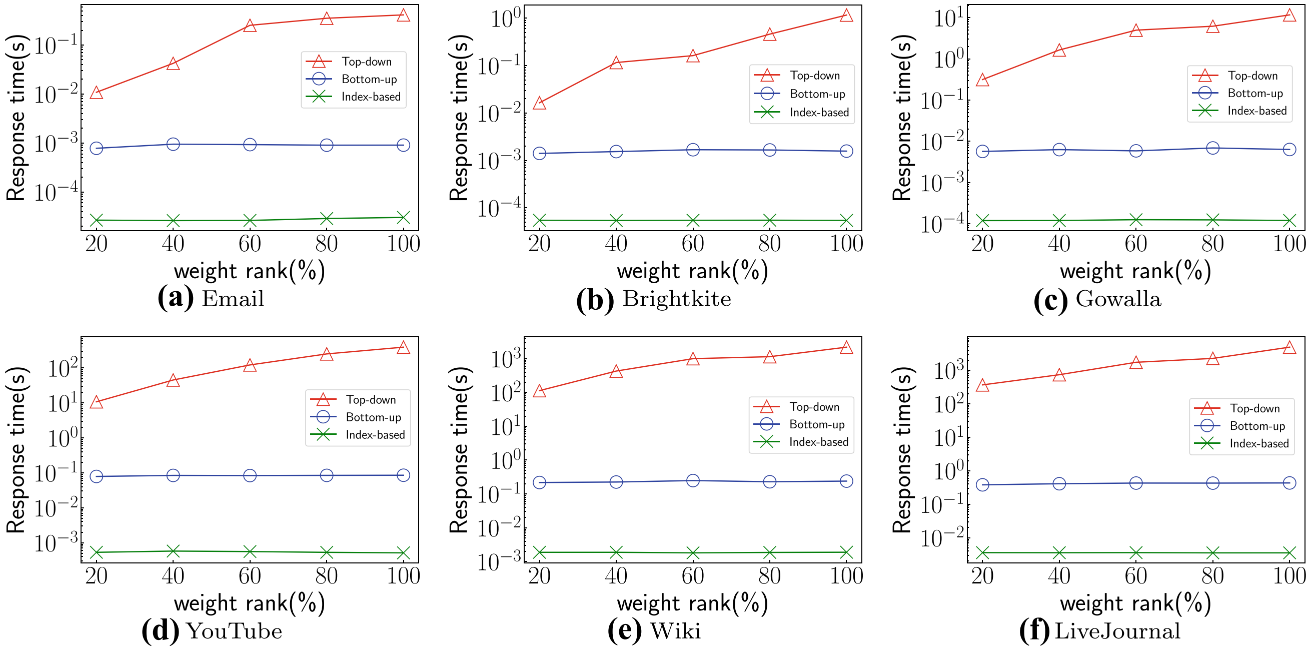
6 Related Work

We present the related work from the following three aspects, i.e., cohesive subgraph mining, community search and influential community detection.

Cohesive subgraph mining Cohesive subgraph mining is a very important tool for graph analysis and can find many applications in different fields [5, 17, 19]. In the literature, different models are proposed to measure the cohesiveness of a subgraph, such as k-core [21, 26], k-truss [25, 27], clique [6, 22, 24], etc. There are also some works that try to identify cohesive subgraph on special graphs, such as identifying k-core and k-truss over uncertain graphs [12, 23].

Community search For cohesive subgraph mining, people usually focus on finding all or high ranked cohesive subgraphs. Given a graph G and query vertices, the community search problem aims to identify a densely connected subgraph that contains the query vertices [9]. To measure the cohesiveness of a community, different models are used. In [7, 20], authors use the minimum degree to serve as the metric, which is similar to the k-core constraint. [20] proposes a global search framework to identify the community. Cui et al. [7] develop a local search method to avoid visiting too many vertices. Huang et al. [10] leverage the k-truss model and propose the triangle-connected k-truss community problem. It designs a triangle connectivity-preserving index to efficiently search the k-truss communities. There is lots of research for other kinds of graphs, e.g., attribute graphs and profile graphs [4, 8]. [9] presents a comprehensive survey of recent advanced methods for community search problems.

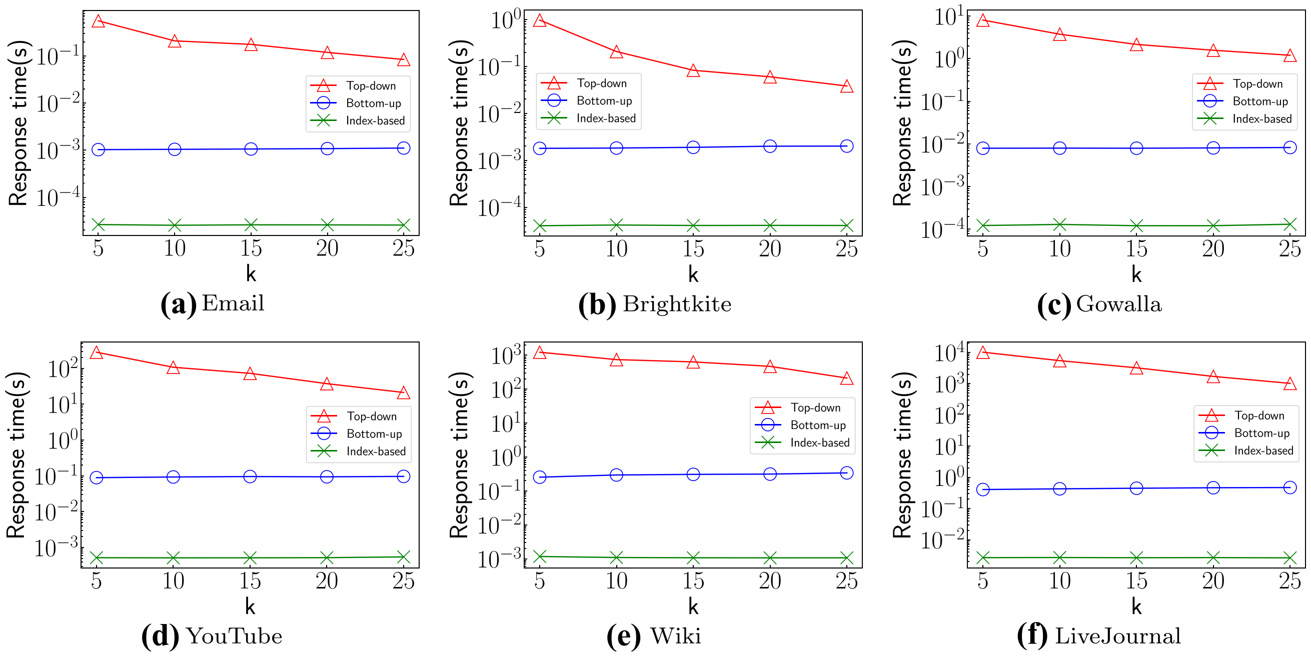
Influential community detection In traditional community detection/search problems, the influence value of a community has been neglected. In [16], Li et al. present a novel community model called k-influential community. Given a graph G, each vertex is associated with a weight, i.e., influence value. It aims to find the top-r k-influential communities, where the cohesiveness is measured based on the k-core model. In [3], Chen et al. propose the backward searching technique to enable early termination. Recently, Bi et al. [2] develop a local search method, which can overcome the deficiency of accessing the whole graph. Li et al. [15] present an I/O-efficient algorithm to compute the top-r influential communities. In [14], authors further investigate the case when each user is associated with multiple weights. However, as observed, these works aim to identify the influential communities for the whole network, while the personalized case has not been considered.



(a) Email **(b)** Brightkite **(c)** Gowalla

(d) YouTube **(e)** Wiki **(f)** LiveJournal

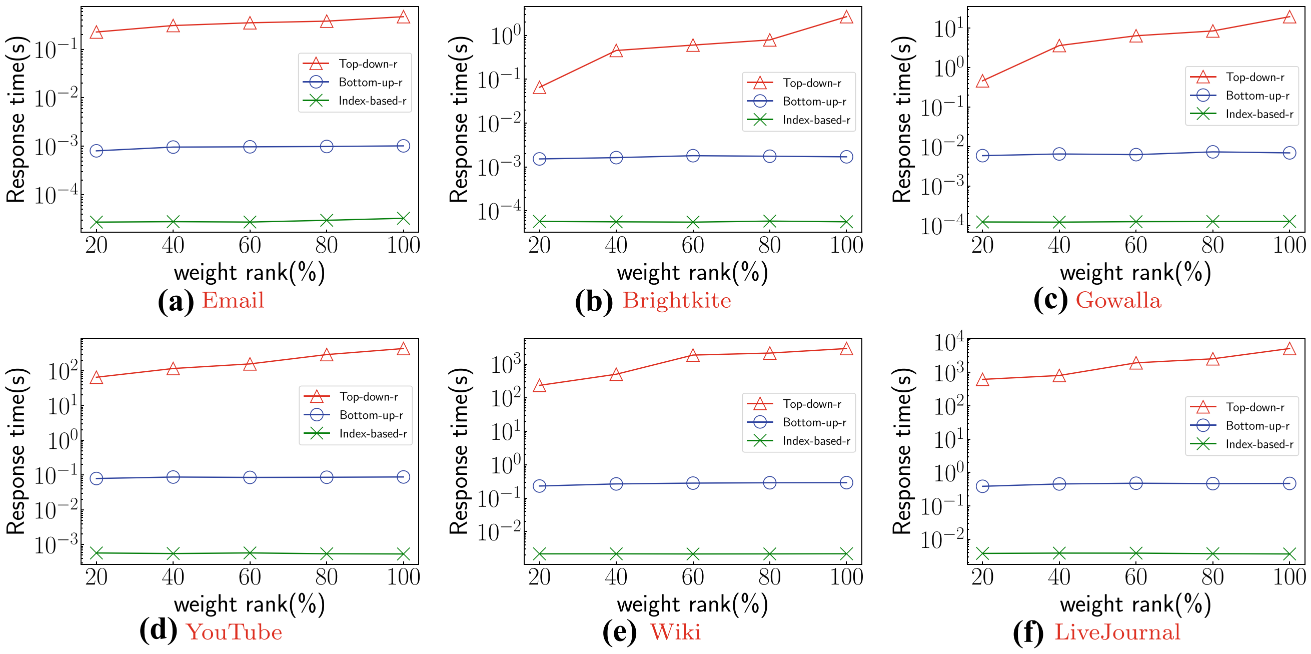
Fig. 4 Experiment results of MPIC by varying query vertex weight



(a) Email (b) Brightkite (c) Gowalla

(d) YouTube (e) Wiki (f) LiveJournal

Fig. 5 Experiment results of MPIC by varying *k*



(a) Email (b) Brightkite (c) Gowalla

(d) YouTube (e) Wiki (f) LiveJournal

Fig. 6 Experiment results of TPIC by varying query vertex Weight

7 Conclusion

In this paper, we investigate the maximal personalized influential community search and top-*r* personalized influential community search problems, which is an important tool for many applications, such as personalized friend recommendation and social network advertisement. For the maximal personalized influential community search problem, in order to scale for large networks, two algorithms, i.e., top-down algorithm and bottom-up

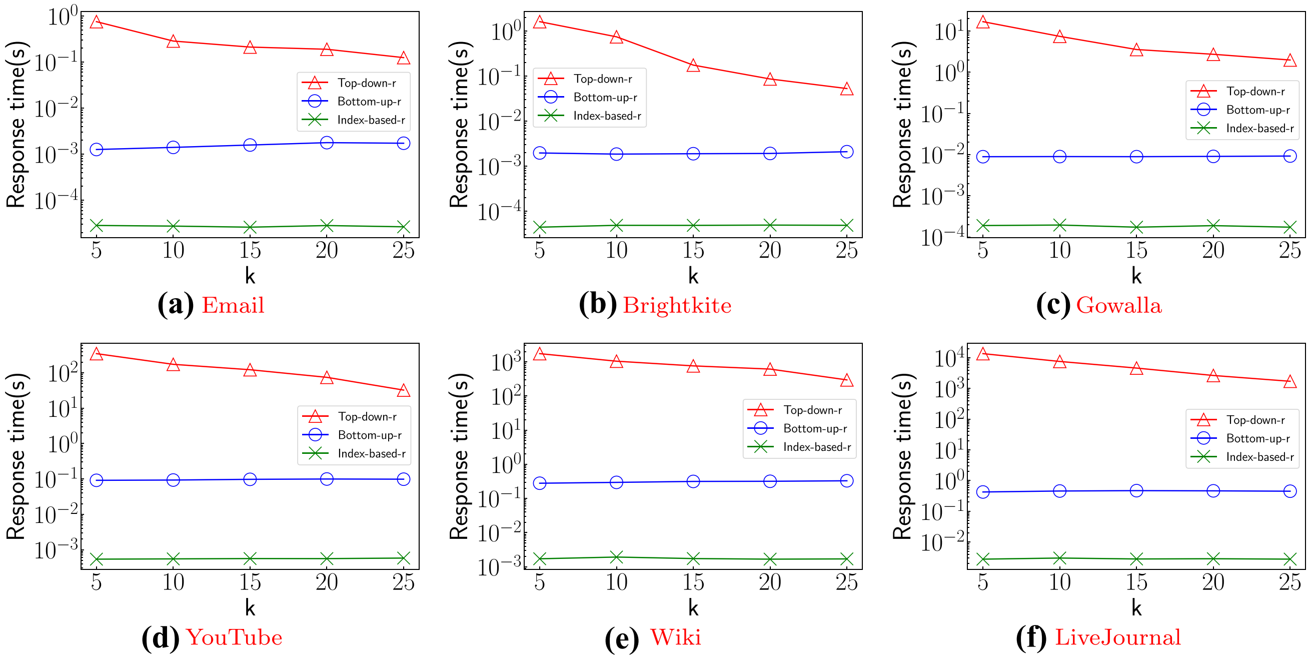


Fig. 7 Experiment results of TPIC by varying *k*

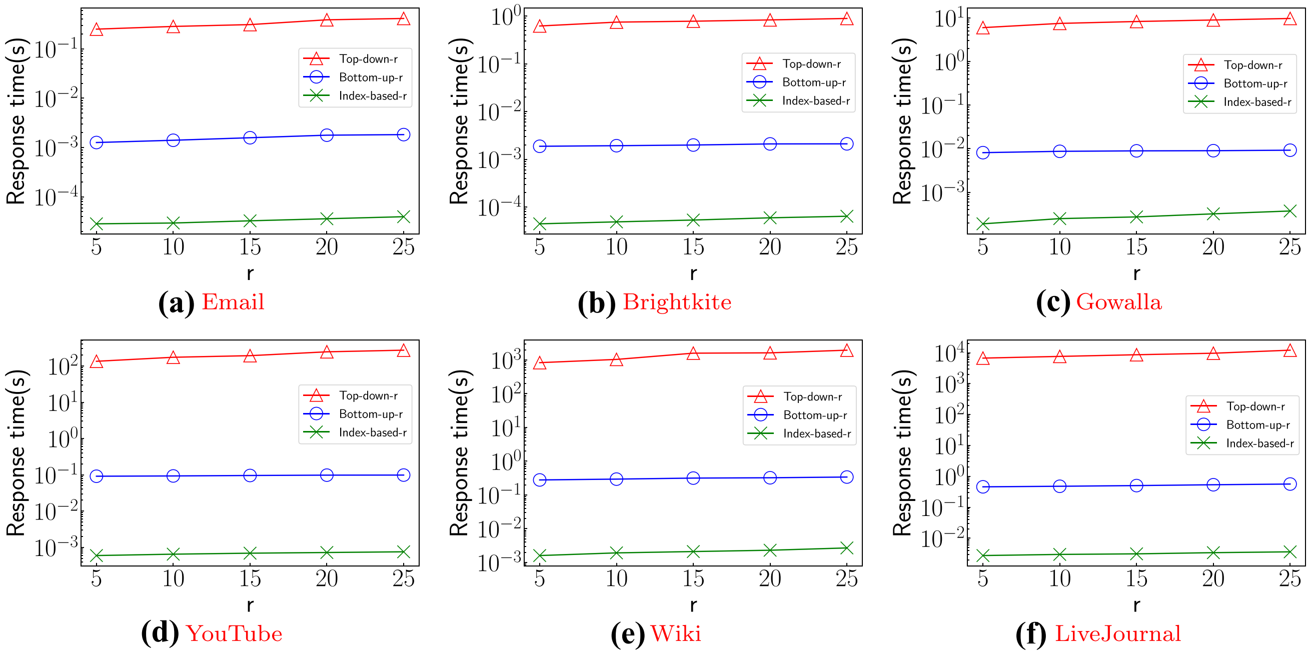


Fig. 8 Experiment results of TPIC by varying *r*

algorithm, are developed based on different vertex accessing orders. To fulfill the requirement of online searching, an index-based method is proposed. In addition, we further extend the searching frameworks to support top-*r* case when multiple communities are needed by the users. Finally, comprehensive experiments are conducted to verify the advantage of developed techniques on 6 real world datasets.

Acknowledgements The preliminary version of this article has been published in APWeb-WAIM 2020 [1].

References

1. –: Efficient personalized influential community search in large networks. In: Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) joint international conference on web and big data, pp 86–101 (2020)
2. Bi F, Chang L, Lin X, Zhang W (2018) An optimal and progressive approach to online search of top-k influential communities. VLDB
3. Chen S, Wei R, Popova D, Thomo A (2016) Efficient computation of importance based communities in web-scale networks using a single machine. In: CIKM
4. Chen Y, Fang Y, Cheng R, Li Y, Chen X, Zhang J (2018) Exploring communities in large profiled graphs. TKDE
5. Cheng J, Ke Y, Chu S, Özsu MT (2011) Efficient core decomposition in massive networks. In: ICDE
6. Cheng J, Ke Y, Fu AWC, Yu JX, Zhu L (2011) Finding maximal cliques in massive networks. TODS
7. Cui W, Xiao Y, Wang H, Wang W (2014) Local search of communities in large graphs. In: SIGMOD
8. Fang Y, Cheng R, Luo S, Hu J (2016) Effective community search for large attributed graphs. VLDB
9. Fang Y, Huang X, Qin L, Zhang Y, Zhang W, Cheng R, Lin X (2019) A survey of community search over big graphs. VLDB J
10. Huang X, Cheng H, Qin L, Tian W, Yu JX (2014) Querying k-truss community in large and dynamic graphs. In: SIGMOD
11. Huang X, Lakshmanan LV, Xu J (2017) Community search over big graphs: models, algorithms, and opportunities. In: ICDE
12. Huang, X., Lu, W., Lakshmanan, L.V.: Truss decomposition of probabilistic graphs: Semantics and algorithms. In: SIGMOD (2016)
13. Khan BS, Niazi MA (2017) Network community detection: a review and visual survey. arXiv
14. Li R, Qin L, Ye F, Yu JX, Xiao X, Xiao N, Zheng Z (2018) Skyline community search in multi-valued networks. In: SIGMOD
15. Li R, Qin L, Yu JX, Mao R (2017) Finding influential communities in massive networks. VLDB J
16. Li RH, Qin L, Yu JX, Mao R (2015) Influential community search in large networks. VLDB
17. Liu B, Yuan L, Lin X, Qin L, Zhang W, Zhou J (2020) Efficient (α, β)-core computation in bipartite graphs. VLDB J 29(5):1075–1099
18. Parthasarathy S, Ruan Y, Satuluri V (2011) Community discovery in social networks: applications, methods and emerging trends. In: Social network data analytics
19. Sariyüce AE, Pinar A (2016) Fast hierarchy construction for dense subgraphs. VLDB
20. Sozio M, Gionis A (2010) The community-search problem and how to plan a successful cocktail party. In: KDD
21. Sun R, Chen C, Wang X, Zhang Y, Wang X (2020) Stable community detection in signed social networks. TKDE
22. Sun R, Zhu Q, Chen C, Wang X, Zhang Y, Wang X (2020) Discovering cliques in signed networks based on balance theory. In: DASFAA, pp 666–674
23. Yang B, Wen D, Qin L, Zhang Y, Chang L, Li R (2019) Index-based optimal algorithm for computing k-cores in large uncertain graphs. In: ICDE
24. Yuan L, Qin L, Zhang W, Chang L, Yang J (2018) Index-based densest clique percolation community search in networks. IEEE Trans Knowl Data Eng 30(5):922–935
25. Zhao J, Sun R, Zhu Q, Wang X, Chen C (2020) Community identification in signed networks: a k-truss based model. In: CIKM, pp 2321–2324
26. Zhu W, Chen C, Wang X, Lin X (2018) K-core minimization: an edge manipulation approach. In: CIKM
27. Zhu W, Zhang M, Chen C, Wang X, Zhang F, Lin X (2019) Pivotal relationship identification: the k-truss minimization problem. In: Kraus S (ed) IJCAI