Incremental Influential Community Detection in Large Networks

Klearchos Kosmanos*
School of Informatics
Aristotle University
Thessaloniki, Greece
kleakosm@csd.auth.gr

Panos Kalnis
King Abdullah University of Science and Technology (KAUST)
Thuwal, Saudi Arabia
panos.kalnis@kaust.edu.sa

Apostolos N. Papadopoulos
School of Informatics
Aristotle University
Thessaloniki, Greece
papadopo@csd.auth.gr

ABSTRACT

The concept of network communities has been studied thoroughly in the network science literature since it has many important applications in diverse fields. Recently, the community concept has been combined with the concept of influence. The aim of this combination is to allow for the detection of communities that have also a high degree of influence. To achieve this, there is a need to guarantee that communities are *good* with respect to their structure and also *influential* with respect to attribute values of the nodes participating in the community. In the literature, there are two main directions to attack the problem: *i*) the *online approach*, which computes influential communities in increasing influence value order, and *ii*) the *index-based approach*, which pre-computes influential communities and stores appropriate information in a tree-based index structure. Based on these two directions, we propose a new technique with the following properties: *i*) there is no need to process the graph each time a new query arrives, and *ii*) there is no need to waste computational resources to maintain parts of the index that users are not interested in. This is achieved by starting without any index in memory. Then, using online algorithms, as new queries arrive, we incrementally build parts of the index that help answering similar future queries. Extensive experimental results, on real world graphs, demonstrate the efficiency of our method against existing approaches in most realistic cases.

CCS CONCEPTS

• **Information systems** → **Data mining**; **Social networks**;

KEYWORDS

graph mining, influential community detection, incremental algorithms, performance evaluation

ACM Reference Format:
Klearchos Kosmanos, Panos Kalnis, and Apostolos N. Papadopoulos. 2022. Incremental Influential Community Detection in Large Networks. In *34th International Conference on Scientific and Statistical Database Management (SSDBM 2022), July 6–8, 2022, Copenhagen, Denmark.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3538712.3538724

*Work done and financially supported during internship at the King Abdullah University of Science and Technology (KAUST), Saudi Arabia.

SSDBM 2022, July 6–8, 2022, Copenhagen, Denmark

ACM ISBN 978-1-4503-9667-7/22/07…$15.00
https://doi.org/10.1145/3538712.3538724

1 INTRODUCTION

Community detection is a fundamental problem in network analysis that has attracted much interest during the last decade [12, 17, 30]. Given a graph $G = (V, E)$, where V is the set of nodes and E is the set of edges, the output of a community detection algorithm is a set of communities $C_1, ..., C_s$, where each C_i is a subset of the number of nodes V of the graph. The members of a community form a *densely connected* group of nodes. Community detection in graphs has many similarities with the clustering problem in multi-dimensional datasets. Therefore, depending on the definition of *density*, there are many different alternatives to define communities. For example, it is natural to expect that the number of edges connecting nodes of the same community will be significantly larger than the edges connecting nodes across communities.

Community structures are contained in many real-world networks. As a result, community detection has some important applications [11, 25] such as:

- **Friend recommendation**. Social media platforms maintain a friendship network. The aim is to suggest to a user u some candidate friends. We can recommend to u, persons that exists in his community but they are not yet friends.
- **Event organization**. Social events involve groups of users that are well acquainted. We can recommend same events to users that belong into same communities.
- **Protein complex identification**. In biology, a gene is often regulated by a set of proteins. So, in order to study a gene we may search for proteins that interact with each other with a significant degree, which is actually a community of proteins.
- **Advertisement in e-commence**. Users that belong to the same community tend to share common interest. We can propose advertisements to a user, based on the advertisements that are checked by his community members.
- **Infectious disease control**. If a person gets an infection disease, we should track his community members for a possible transmission. With this way we can limit the spread of the disease. When transition spreads across many persons, the concept of influence may be used to focus on people that belong to communities with the highest influence. These people are responsible for major number of new infections due to their higher influence.

The first priority of a community detection or a community search algorithm is to discover groups of nodes that have a specific degree of *cohesiveness*. However, in many real-world applications nodes are annotated with additional attributes that play a significant role in the formation of the network structure. For example, in a social network graph, the information contained in users' profiles

is essential for friend recommendation or to provide more targeted advertisements. Therefore, it is reasonable to take node attributes into account towards the discovery of more important communities, not only with respect to structure but also with respect to *influence*. This process requires the existence of a meaningful *influence score*.

The introduction of the influence score in each community, suggests that we are dealing with a *two-dimensional problem*, which is defined by two variables r and k. In this setting, r expresses the number of communities in the result, and k defines the *goodness* of the community with respect to connectivity and structure. Based on these two numbers, we are interested in the discovery of the top-r k-influential communities.

The related literature contains two distinct alternatives for the computation of top-r k-influential communities:

- **On-line algorithms**. The main characteristic of any on-line algorithm is that computes every query from scratch, traversing every time the graph. The first on-line algorithm was presented in [21], which computes all k-influential communities in increasing influence value order. So, the last r identified k-influential communities are the results. Then, a Forward and a Backward algorithm was proposed in [6]. Forward is an improvement of [21] whereas Backward detects the k-influential communities in the reverse order, from the most important to the least. Finally an instance-optimal algorithm was presented in [3], which avoids traversing the entire graph for finding just the top-r k-influential communities and outperforms both Backwards and Forward.
- **Index-based algorithms**. An index-based algorithm that efficiently retrieves the top-r k-influential communities was presented in [21]. The general idea is that first all k-influential communities are pre-computed for each possible value of k and stored in a space-efficient tree structure. Then, using this index, it is possible to answer any query for top-r k-influential communities in linear time with respect to the size of the top-r results.

Each previous approach has its own drawback. Algorithms in [21] are global search (they need to traverse the entire graph). Backward [6] tried to implement a local search but with a cost of quadratic time and it is outperformed by Forward when k is large. LocalSearch [3] seems to be the best on-line algorithm (requires linear time proportional to the size of the subgraph it traverses) but still is not competitive to the time that an index can answer a query. On the other hand, the index-based algorithm [21] requires a pre-processing step and needs large amount of main memory; thus is not scalable for very large graphs.

Contributions. In this paper, we propose an efficient approach that can handle multiple queries for top-r k-influential communities. Our approach combines online and index-based techniques in order to eliminate their drawbacks and enhance their advantages. To achieve this, we begin with no index in main memory. Then as queries arrive, we utilize online algorithms to build parts of the index that can help answering similar future queries. This way, we do not waste computational resources to maintain parts of the index that users are not interested in, plus we avoid processing the whole graph every time a new query arrives.

Experimental results shows as our superiority against simple online algorithms in every case/scenario. The time that we save is proportional to the total number of queries (q) that are answered. We notice a faster gain in time when $q >> k$, which is also a more realistic scenario, since queries can be many thousands, but k value is usually some hundreds. Nevertheless, against index-based method, we are not always faster. It depends on especially k and then r values. Bigger values results in a time that can exceed far the time of building the index. Despite, we always need far less amounts of main memory.

Roadmap. The rest of the work is organized as follows. Section 3 presents the necessary background. The proposed approach is presented in detail in Section 4. Performance evaluation results are offered in Section 5 and related work is summarized in Section 2. Finally, Section 6 concludes our work and discusses briefly interesting future research directions.

2 RELATED WORK

In this section, we present briefly research works that are closely related to our work. In particular, we focus on community search, cohesive subgraphs and database cracking.

Community Search. This problem is different from community detection. The concept is that we have a single node or a set of nodes and the goal is to find the community that nodes belong into. Also, the minimum cohesiveness of the community should be given by the user. Different models are proposed based in different concepts such as k-core [1, 25], k-truss [15, 18], edge density [29], edge connectivity [4, 20], α-adjacency γ-quasi-k-clique model [9], spatial-aware community [10], attributed community [16] and random walk [26]. All edges of a k-truss are contained at least in $k-2$ triangles. Edge density is based in various definitions such as the average degree of the nodes. A k-edge connected graph still remains connected when $k-1$ edges are removed from it. A graph with k nodes and at least $\gamma \frac{k(k-1)}{2}$ edges is called a γ-quasi-k-clique. Spatial-aware communities are communities that all nodes are spatially close to each other. In some graphs nodes have properties that are meaningful for the sense of communities. These nodes are associated with attributes and these graphs are called attributed graphs. Note that most of these works do not consider the influence of communities.

Cohesive Subgraph Mining. Computation of cohesive subgraphs in a graph is an important concept in social network. There are many definitions of cohesive graphs including k-core [24], k-truss [27], DN-graph [28], maximal clique [7], edge connectivity [5] and locally denset subgraph [23] to name a few. A DN-graph regarding a value λ is a connected graph in which every two connected nodes have in common at least λ neighbors. Maximal clique is a clique of a graph in which there are not remaining nodes in the graph that are connected to each clique's node. Maximal subgraphs are computed based on a threshold value of cohesiveness given by the user. Various studies on k-core and k-truss decomposition satisfy different settings, such as in-memory algorithms [2], external memory algorithms [27] and either MapReduce [8].

Database Cracking. This technique [19] is based on the hypothesis that maintenance of an index is part of query processing and not of updates in database. Each query should not be handled as a request for some data, but as an advice to speed up future queries with similar interest. The unexplored areas of database remain non-indexed until a query becomes interested in these data. All this process brings up the benefits of self-organization.

3 BACKGROUND

In this section, we present some important background information to keep the paper self-contained. In particular, we define the problem formally and describe online as well as index-based approaches.

3.1 Problem Statement

Consider an undirected graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. The degree of each node u in G (denoted as $d(u, G)$) is defined as the number of its neighbors in G. Consider also a vector w that assigns to each node u in V a weight. The weight of a node u (w_u) denotes its influence value in G and it can be its PageRank value, a centrality score, or any other meaningful attribute value. A higher value of the attribute suggests a more influential node.

An induced subgraph of G, denoted by $H = (V_H, E_H)$, is a graph with $V_H \subseteq V$ and $E_H = \{(u, v) | u, v \in V_H, (u, v) \in E\}$. A k-core is an induced subgraph H where every node $u \in H$ has a degree at least k (i.e., $d(u, H) \geq k$). Every graph has a unique maximal k-core H', which is a k-core such that no supergraph H of H' is also a k-core. The core number c_u of a node u in a graph G is the maximal value of k such that u is included in the k-core of G.

In addition, we assume that every node has a distinct weight and that the weight vector is given apriori (i.e., if $i \neq j$ then $w_i \neq w_j$). Given a value τ, V_τ is a subset of V that contains only the nodes with weight no less than τ. We denote by $G[V_\tau]$ the subgraph of G induced by V_τ, that contains all edges of G whose both end-points are in V_τ. For simplicity we use G_τ to denote the subgraph of G induced by vertices of V_τ (i.e. $G_\tau = G[V_\tau]$).

Before we proceed in more details, it is necessary to clearly define the concept of the influence value of an induced subgraph.

Definition 3.1. Given an induced subgraph $H = (V_H, E_H)$ of an undirected graph $G = (V, E)$ the influence value of H, denoted as $f(H)$, is defined as the minimum weight of all nodes in H.

The above definition of the influence value ensures that there will not be any nodes with low weight in a large influence value induced subgraph, ending to a robust to outliers definition.

Despite its influence value, an influential community should also be a cohesive induced subgraph. Various definitions have been studied in the literature [13, 15, 25] like k-core, k-truss, edge connectivity. The most popular of them seems to be k-core due to its simplicity and low-time computability [11]. Below we give the definition of k-influential community, where k defines the cohesiveness of the community.

Definition 3.2. Given an undirected graph $G = (V, E)$ and an integer k, a k- influential community is an induced subgraph $H = (V_H, E_H)$ of G that satisfies the following constraints:

- Connectivity: H is connected
- Cohesiveness: each node u in H has degree at least k.
- Maximal structure: there is no other induced subgraph H' such that (1) H' satisfies connectivity and cohesiveness constraints; (2) H' contains H_k; and (3) H' has the same influence value as H (i.e., $f(H') = f(H_k)$).

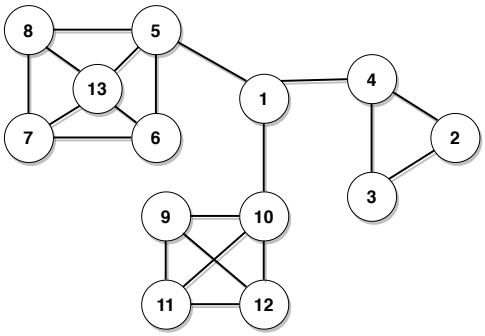


Figure 1: An example graph (numbers denote id and weight).

Example 3.3. Consider the graph given in Figure 1. We can determine that subgraph g_1 induced by nodes $\{9, 10, 11, 12\}$ is a 2-influential community, with influence value 9, as it follows all constrains of definition above. In contrast, the subgraph induced by nodes $\{9, 10, 12\}$ is not a 2-influential community as the maximal structure is not satisfied. This is because $\{9, 10, 12\}$ is contained in the super graph of $\{9, 10, 11, 12\}$ that has also the same influence value. Finally, note that the subgraph induced by nodes $\{10, 11, 12\}$ is a 2-influential community. Despite being a subgraph of g_1, it has a higher influence value (10) than g_1 (9), so again all three constrains are satisfied.

Typically, we are interested in the influential communities whose influence value is larger than the others. In this paper we focus on detecting such communities in a graph. Thus, we aim to find only the top r communities of all communities with a specific cohesiveness value.

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

For example, consider again the graph in Figure 1. We can verify that the top 3-2 influential communities are the subgraphs induced by {10,11,12}, {9,10,11,12} and {7,8,13} with influence value 10,9 and 7 respectively.

3.2 The Online Approach

An on-line solution for detecting the top-r-k-influential communities of a graph was proposed in [3]. In order to understand the approach we need to mention some important lemmas.

Lemma 3.4. *Every k-influential community in $G_{\tau 2}$ is also a k-influential community in $G_{\tau 1}$ if $\tau_1 \leq \tau_2$. It is true that $G_{\tau 1}$ is a supergraph of $G_{\tau 2}$.*

Lemma 3.5. *If g is a k-influential community in $G_{\tau 1}$ and $\tau_1 \leq \tau 2$ and the influence value of g is no smaller than $\tau 2$, then g is also an k-influential community in $G_{\tau 2}$.*

Algorithm 1 Compute $\tau*$

Input: Graph G, number r of influential communities and the cohesiveness value k.
Output: $\tau*$ s.t. $G_{\tau*}$ contains at least r-k- influential communities.

$u \leftarrow (k+r)^{th}$ largest weight node in G
$i \leftarrow 1$
$\tau_i \leftarrow w(u)$
while CountIC(G_{τ_i}) $< k$ and $G_{\tau_i} \neq G$ **do**
  $\tau_{i+1} = argmax_{\tau*}$ s.t. $size(G_{\tau*}) \geq 2 * size(G_{\tau i})$
  $i \leftarrow i + 1$
end while
return $\tau*$

Lemma 3.6. *Let $\tau*$ be the largest value such that $G_{\tau*}$contains at least r-k-influential communities. Then, the set of top-r-k-influential communities in $G_{\tau*}$ is the set of top-r-k-influential communities in G.*

The goal of local search approach is to first detect the smallest possible subgraph of G that contains r-k-influential communities. So we need to find a way to compute $\tau*$ efficiently. In [3] the following strategy has been shown to be the most efficient. We first set τ_1 to be the weight of the $(k+r)^{th}$ largest weight node in G. We do so because the r-k-influential communities contain at least $k+r$ distinct nodes. Then, while G_τ does not contain at least r k-influential communities, we iteratively increase the size of G_τ until it gets doubled. So we exponentially grow the initial $G_{\tau 1}$ until it contains the required number of communities. The pseudo-code to compute $\tau*$ is presented in Algorithm 1.

Example 3.7. Consider the graph shown in Figure 1. Assume that we want to find top 4-2- influential communities. We first set τ_1 equal to the weight of the $4+2=6^{th}$ largest weight node. This is node 8 with weight 8. So $\tau_1 = 8$. G_8 contains only 2-2- influential communities. As a result we need to compute the largest τ_2 value such that $size(G_{\tau 2}) \geq 2 * size(G_{\tau 1})$. The size of $G_{\tau 1}$ is 13 as it consists of 6 nodes and 7 edges. We iteratively add the next highest-weight node into $G_{\tau 1}$ until its size gets at least 26. Node 7 is added and increases the size by 3, as it has an edge with 8 and one with 13. Node 6 is added next, increasing the size by 3. The addition of node 5 increases the size by 4. Now we have a total size of 23. We need 3 more. Finally we add nodes 4 and 3 that results to a new size of 26. So $\tau_2 = 3$. G_3 contains 5-2 influential communities. Now we are safe to say that $\tau_2 = \tau* = 3$.

Until now we have shown that we need to count influential communities in a subraph of G, but we have not explained how to achieve this. In this paper we will use the CountIC algorithm presented in [3]. Before we explain the algorithm we first need to give the definition of keynode regarding k-influential community.

Definition 3.8. We call keynode regarding a k value, a node u in a graph G, if there exists a subgraph g of G with influence value equal to u's weight and the minimum degree node of g is at least k. Note that, by definition of influence value, u must be contained in subgraph g.

For example, in Figure 1, node 10 is a keynode regarding $k = 2$ for the subgraph induced by {10,11,12} since the subgraph has an influence value of 10 which is equal to the weight of node 10, and has a minimum degree of at least 2. We will call a node simply a keynode without referring to his k value which can be inferred from the context.

We should mention the next two lemmas regarding a keynode.

Lemma 3.9. *Given a graph G and a value τ, there is at most one k-influential community with influence value τ.*

Lemma 3.10. *There is one-to-one correspondence between k-influential communities and keynodes in G. As a result, the number of keynodes in G is equal to the number of k-influential communities in G.*

Given an influential community g, the node with the minimum weight would be its unique corresponding keynode, denoted by key(g). Note that g may contain keynodes of possible sub-influential communities of g, but it is uniquely identified by the keynode with the minimum weight. For example, {9,10,11,12} is a 2-influential community which keynode is 9. But, it contains 10 which is the keynode for {10,11,12}. So, the unique corresponding keynode of {9,10,11,12} is 9.

The Algorithm. CountIC takes as input a graph g and a value k and returns the number of k-influential communities in g. This is done by finding all the keynodes in g and then return its count. We first need to reduce g in its k-core. [2] presents an efficient way to find the core number of each node in a graph. Another way to compute k-core of g mentioned in [3], is to call procedure Remove for each vertex in g whose degree is less than k. Then we initialize two empty lists, *keys* to store the keynodes and *cvs* that we can ignore for now as it will be used in a further process-to build the

Algorithm 2 CountIC

Input: Graph G and cohesiveness value k .
Output: Number of k-influential communities in G.

$g \leftarrow$the k-core of g
$keys \leftarrow \emptyset$
$cvs \leftarrow \emptyset$
while $g \neq \emptyset$ **do**
  $u \leftarrow$ the minimun weight node in g
  Append u to $keys$
  Remove(u,g,cvs)
end while
return $|keys|$

Procedure Remove(u, g, cvs)
$Q \leftarrow$ empty queue
push u into Q
while $Q \neq \emptyset$ **do**
  Pop v from Q
  for v' $\in$ N(v, G) **do**
    if $d(v', g) = k$ **then**
      Push v' in Q
    end if
  end for
  Remove v from g
  Append v in cvs
end while

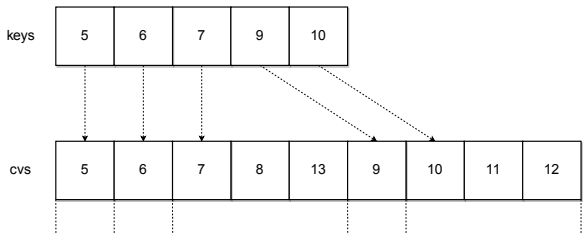


Figure 2: Running example for CountIC 2

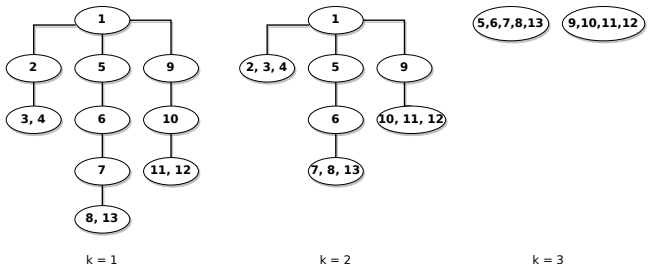


Figure 3: Example of an ICP index.

index. Then, while g is not empty, we iteratively find the minimum weighted node, store it in u, add u to the list of keynodes, and finally remove it from g. Note, that every u will be a keynode as it satisfies the definition of keynode. Furthermore, the process that removes u from g, also computes the k-core of $g \setminus u$. This is necessary as $g \setminus u$ may not be a k-core. To do so, we initialize a queue Q that contains only u. Then we will push every neighbor of u with degree equal to r in Q, and do the same for every node in Q. This process recursively will remove from g every node whose degree becomes less than k as a result of previously deleting nodes from g, resulting to the new k-core. Note that *cvs* will contain all nodes of G, with the order that they are removed from G.

Example 3.11. Lets run algorithm 2 in G_3 with $k = 2$. Remind that G_3 is the subgraph of G that contains all nodes with weight no less than 3. First of all, we reduce G_3 to its 2-core. This action removes nodes 3 and 4, as they both only have a degree of 1. Then we remove the node with the minimum weight. That is node 5. We append to the list of *keys* and call the remove process for this node. None of its neighbors have degree 2, so only node 5 gets removed. Secondly, we delete node 6. Again none of its neighbors gets deleted. Next, node 7 gets removed. The remove process also removes nodes 8 and 13, as they are not par of 2-core anymore. Similarly we delete node 9 and finally node 10, which also deletes nodes 11 and 12. So we have a total of 5 keynodes. That means that there are 5 2-influential communities in G_3. The resulting contents of *keys* and *cvs* are shown in Figure 2.

Until now, we know how to detect a small subgraph of G in order to locally search for influential communities and we also know a way to count their number in the subgraph. The next and final step is to extract the influential communities. This is achieved with the help of our two lists, *keys* and *cvs*. Given these two lists, we create one group of nodes for each keynode in *keys*. The group of each keynode u is denoted by gp(u) and it contains u and all nodes after u and before the next keynode in *cvs*. From the previous example it is true that gp(5)={5}, gp(6)={6}, gp(7)= {7,8,13}, gp(9)={9} and gp(10)={10,11,12}.

In order to extract the final influential communities, we are based on the lemma below. We denote by $IC(u)$, the influential community corresponding to keynode u, as we have already explained that every influential community is corresponding by its unique keynode.

Lemma 3.12. *IC(u) equals the union of gp(u) and IC(u') for each keynode u' such that $w(u') > w(u)$ and there is an edge between a node of gp(u) and a node of gp(u').*

We will not present the algorithm in detail, as it exceeds the needs of this paper. The interested reader is referred to [3] which also describes in detail the required graph organization. In next sessions we will only use the two lists and the groups of every keynode in order to build the index. Then we will be able to extract the influential communities easier by using the index.

3.3 The ICP index

The idea of the index is based on the fact that for each k, the k-influential communities form an inclusion relation. As we have shown in previous examples, an influential community may be contained in another one. Consider $H_k = \{5, 6, 7, 8, 13\}$ which is an influential community with influence value 5. H_k contains {6,7,8,13} and {7,8,13} which are other influential communities.

Lemma 3.13. *For any k -influential community H_k, if we delete the minimum weight node and the resulting subgraph still contains a maximal k-core C_k, then each maximal connected component of C_k is a k- influential community.*

Based on the lemma above it is true that a k-influential community contains all sub-k- influential communities which are the MCCs of the maximal k-core of $H_k \setminus u$. So we can construct a two level tree structure to organize this inclusion relationship as follows: the parent vertex is H_k and each MCC of the maximal k-core of $H \setminus u$ will be a child of the H_k. Every child also will be a k -influential community. Note that we can recursively use the above result to every child of the initial influential community, resulting in a tree structure.

So, all k-influential communities can be organized as a tree-shape structure, where the root of the tree will represent the influential community with the lowest influence value and every child will be a different influential community with higher influence value. As a result the top-r-k- influential communities will be close to leaves.

It is not effective to store in every vertex of a k tree the whole influential community but we can use the inclusion relationship to avoid this problem. 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 do not exist in its children influential communities.

Example 3.14. Consider the graph in Figure 1. All nodes of G form a 2-core. Assume that we are interested for all 2-influential communities in G. The whole graph is a 2-influential community with the smallest possible influence value of 1. If we remove node

1, three MCCs are created, that are also 2- influential communities. That is {2,3,4}, {5,6,7,8,13} and {9,10,11,12}. So far we should have for 2-tree a root vertex containing only node 1 with 3 children, one for each MCC. Root contains only node 1 as it is the only node that does not appear in any sub-2-influential community. Then we remove node 2, which also removes nodes 3 and 4. So the vertex {2,3,4} will be a leaf as it does not have any children. Then the removal of node 5 creates one MCC that contains {6,7,8,13}, so vertex {5,6,7,8,13} now splits into a parent node {5} with a single child {6,7,8,13}. We continue the process with the same logic resulting to the 2-tree shown in Figure 3.

When we want to extract a k-influential community from a vertex v of the k-tree we have to return as a result, not only the nodes contained in this vertex, but in addition all nodes that exist in all vertices of the sub-tree with root the initial vertex v. Note for $k = 3$ we do not have a tree, but a forest structure, as the 3-core of our graph consist of two MCCs. Figure 3 shows the whole ICP index structure.

The algorithm to build the ICP-index of a graph G is presented in detail in [21] . We will not present these algorithms in this paper as our goal is not to build the whole ICP-index, but based on that algorithm we will only build the most bottom parts of each tree that users are interested in.

4 THE PROPOSED APPROACH

Consider a scenario that we have a large-scale graph G and some users that query for k-influential communities over G. We expect that some queries will be very similar to each other. For example, a user may ask for the top-10 10-influential communities and then another user may ask for the top-5 10-influential communities. In this case, we have already computed the second query as it is contained in the answer of the first one. Therefore, we should avoid re-computing the top-5 10-influential communities query. Assume further that a third query is asking for the top-20 10-influential communities. We should find a way to use the previous results of top-10 10-influential communities in order to compute the top-20 10-influential communities. Thus, we aim to determine the best strategy to answer these queries given an input graph.

A simple approach is to use an on-line algorithm, but in this case we do not use any relation that exists between different queries and plus we have to process all the graph every time a new query occurs. Then we can think about building the ICP-index. Using the index we will be able to answer every query in time linear to the size of communities in the query, so it is optimal [21]. We just need to build the whole index once. Also, we do not care for any relationship between the queries. The problem with this implementation is that for large-scale graphs most times we need a large amount of main memory. Also, we have to pre-computed all k-influential communities, even if users are interested in finding top-100 or the top-1000 only. Or users may not be interested in all possible k-influential communities, but only for specific values of k. In that case, building the ICP-index will make unnecessary computations in order to compute some k-trees, plus it will lead to excessive main memory usage.

In this section we propose a method that starts with no index in memory and then based on users queries it uses online algorithms

Algorithm 3 Query

Input: Graph G, number r of influential communities and the cohesiveness value k.
Output: Top r-k-influential communities in G.

```
if k-tree exists in ICP then
    if k-tree contains less than r nodes then
        Call on-line algorithm to extend the k-tree of ICP
    end if
else
    Call online algorithm to create the k-tree of ICP
end if
Use ICP to extract top r-k-influential communities
```

Algorithm 4 ConstructTree

Input: k value of cohesiveness, cvs and $keys$ lists.
Output: part of k-tree in ICP index.

```
Keep nodes of cvs and keys that they do not exist in any vertex of k-tree
Create a signal-vertex tree in k-tree for each gp(u) of keys
for node u in cvs with reversed order do
    for v ∈ N(u, G) s.t.w_v > w_u do
        Su ← the root node of the tree containing u in k-tree
        Sv ← the root node of the tree containing v in k-tree
        if Su ≠ Sv then
            Merge the trees rooted at Su and Sv in k-tree by adding
            Sv as a child vertex of Su
        end if
    end for
end for
```

to answer them, and furthermore, builds partially the k-tree of the ICP index. This will help future queries, as it makes use of any possible relation between different queries (queries with same k value). Furthermore it uses far less memory, especially when the graph is large and contains many communities. This happens because, even if we have a single query for every possible k value, still we will not build the whole tree, but only the bottom levels of it, since we are interested only for the top influential communities.

The Algorithm. Given a new query that asks for top-r-k-influential communities, we first look if the k tree of ICP index already exists. If this is the case we look if the k-tree contains at least r influential communities. If this is true, then we are able to answer the query using the index, without making any process in the graph. Else, if only t influential communities are in the k-tree ($t < r$) we need to extend the index in order to contain at least r influential communities. This is achieved by calling the online algorithm and then creating the vertices of k-tree that do not already exist. Finally, if k-tree does not exist in the index, we need to run the online algorithm and create the k-tree.

Furthermore, we need to explain how to build the k-tree of the ICP index, after calling the online algorithm. We will make use of the cvs list. Remember that using cvs and each keynode u in $keys$ we can retrieve the gp(u) which is a vertex of the k-tree. First we create a single tree vertex for each gp(u). Thus we only have to

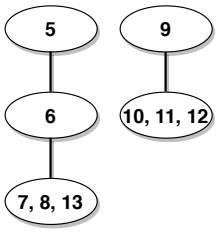


Figure 4: Index status after executing the first query.

create pointers between every gp(u), in order to create the tree structure. We will edit the algorithm presented in [21], since we only want to partly create the k-tree. Note that nodes in *cvs* will be in increasing weight order because each time we delete the node in G with the minimum weight. The algorithm takes each node u in *cvs* and then for each neighbor v with weight more than $w(u)$ finds the root vertex of $gp(u)$ and $gp(v)$ denoted by Su and Sv. Then, if these two vertices do not have the same root, we merge them by adding Sv as a child vertex of Su.

Example 4.1. Suppose we ask for the top 4-2-influential communities. Initially, we do not have any part of the index available so the online algorithm is called. We observe in Figure 2 the final contents of *keys* and *cvs*. We note all groups for every keynode u in *keys*. In order to build the index, we first create a single vertex tree for each group. Then, we traverse *cvs* in reverse order. Node 12 does not have any neighbor with larger weight. Node 11 has one neighbor with larger weight, that is node 12. But both nodes have the same single tree vertex so no merge operation is performed. When we reach node 9 we see that it has node 12 as neighbor. These two vertices have different roots, so we merge them by setting vertex {10,11,12} as a child of vertex {9}. Now, the other two neighbors of 9, that are 10 and 11, have the same root tree vertex (that is vertex {9}), because of the previous merge. The process continues with the same logic. So at vertex {6}, we need to merge it with {7,8,13} and then vertex {5} needs to be merged with vertex {6}. The state of the index after the first query is shown in Figure 4.

Suppose then that someone asks for top 6-2-influential communities. We only have 5 influential communities stored, so we have to extend the index. The previous $\tau*$ was equal to 3. The new $\tau*$ will be 1 (since we double the size of G_3), so we search the entire graph. Note that we do not expect this to happen in larger graphs as they contain more communities. The *cvs* will now contain {1,2,3,4} and then all the nodes we saw in the previous example {5,6…}. We only create single tree vertices for 1 and {2,3,4} as the other vertices do already exist. Then in order to expand the index we do not use all nodes in *cvs* but only nodes previous than node 5. That is because this is the node that we stopped building the index at the previous query. □

Complexity Analysis. Index requires $O(\rho \cdot m)$ time to be created, where ρ is the arboricity of the graph. Then, a top r-k-influential community query may be answered in time proportional to the size of top-r results. So, any query can be answered in $O(r_{max})$ where r_{max} denotes the maximum size of communities contained in all queries. Thus, if q queries occur, total time needed will not be more that $O(q \cdot r_{max} + \rho m)$. This time is independent from the k-value that users are interested in. Even if we know that queries contain only some k values of maximum k, we cannot change the order of the complexity. Furthermore, since communities are computed with increasing influence value, we are unable to save time if we are only interested for the top of communities. This happens because, top r influential communities are computed last.

Our approach needs same time to answer queries if index already exists. So we have a total cost of $O(q \cdot r_{max})$, just using the index for q queries. Now, we only have to calculate the time of building the index. In worst case every possible k-tree will be created. In this case at least a single query will exist for each possible k value. Based on local search approach, the online algorithm traverses only the subgraph that contains the required number of communities. So, the time complexity is $O(size(G_{\tau*}))$. Typically in most cases it is true that $O(size(G_{\tau*})) << O(m)$ and previous experiments [3] show that local approach traverses lees than 1% of whole graph. However, in worst case a single query for each k, will ask for very high r value, or all k-influential communities. Then, $O(size(G_{\tau*}))$ will be at most equal or close enough to $O(size(G))$. So, index will be created in time $O(k \cdot size(G))$ which is $O(k \cdot m)$, since $size(G) = m + n$ but, it is true that m is never less than $n - 1$ (for undirected connected graphs), so we can assume that the order complexity of $O(size(G)) = O(2m) = O(m)$. Therefore, if q queries occur, total time needed using our method will not be more that $O(q \cdot r_{max} + k \cdot m)$.

As a result, when we have not any limits for k and r values and whole index will be constructed over time, our method will be up to k_{max}/ρ times slower than directly building the index from the beginning. Note, that the arboricity ρ of a graph is proven never to be more than $m^{1/3}$ and in real word graphs it is a very small number [14, 22].

Online algorithm cannot answer every query in time $O(r_{max})$, as it does not use any index and it has to process the graph every time a query occurs. If we consider again a worst case scenario, in which some queries need to traverse the whole graph (because k or/and r values are very high), then the algorithm needs $O(q \cdot m)$ time to answer q queries.

5 PERFORMANCE EVALUATION

In this section, we present performance evaluation results regarding the comparison of the different algorithmic techniques. More specifically, we have conducted an extensive performance evaluation to assess the efficiency to answer a specific number of queries in a graph. The queries are randomly generated. We have used three different methods:

(1) Firstly, we answer the queries using only the best online algorithm.
(2) Then, we build the ICP index in order to answer the same queries.
(3) Finally we have used the technique proposed in this paper.

5.1 Set-up and Datasets

Every test was repeated five times for five different random sets of queries. We vary three query parameters for each graph. The number of queries (q) made by users, the maximum number (r) of communities that a user asks for and also the maximum cohesiveness value (k) of the communities. PageRank algorithm is used to calculate the weight of each node u, as a very common method for this purpose. We have implemented all methods in Python version 3.8.2.

All experiments were conducted on a multi-core machine equipped with an Intel Xeon E5-2680 @2.80 GHz CPU and 756 GB main memory running Linux Ubuntu 20.04.1 LTS.

Table 1: Datasets and characteristics.

Dataset	#nodes	#edges	*max-d*	*max-k*
Youtube	1,134,890	2,987,624	28,754	51
Wiki	1,791,489	28,511,807	238,342	99
Live-Journal	3,997,962	34,681,189	14,815	353
Orkut	3,072,627	117,185,083	33,313	253
Friendster	65,608,366	1,806,067,135	5,241	304

We have used five real-world graphs in our experiments downloaded from the Stanford Network Analysis Platform (SNAP, http://snap.stanford.edu/). The most important characteristics of the datasets are given in Table 1. The columns *max-d* and *max-k* denote the maximum node degree and the maximum cohesiveness value respectively. In each graph every node has a distinct weight value.

5.2 Experimental Results

In the sequel, we report experimental results based on different values of the most important parameters k, q and r.

5.2.1 Impact of the maximum cohesiveness value (k). . In this series of experiments, we vary the maximum cohesiveness value k of the communities that are asked by users. So $k = 10$ means that queries include at maximum 10-influential communities and $k = 30$ includes all previous possible queries plus communities with cohesiveness up to 30. We make 100 queries that can ask up to top 100 communities. The results are show in Figure 5. We observe that the time that the index needs it is actually the time of its construction. Then, as it can handle queries for every k, it does not make any difference in time as we increase k. On the other hand, this is not true for online algorithms, since they require more time to answer queries with higher values of k. This happens because they need to traverse a larger part of the graph in order to include nodes that exist in the k-core and thus, they are candidates to form a k-influential community. We note also that our method is better than online algorithm in every range of k. This is reasonable, as some queries will use the index that has been built due to previous queries.

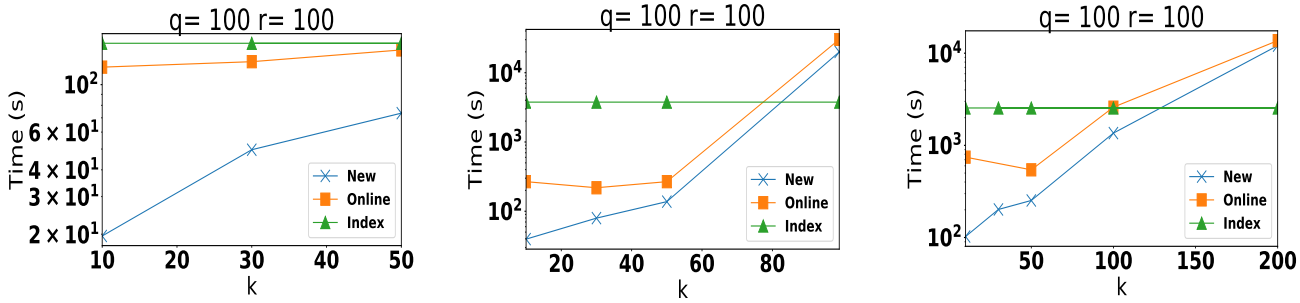
We should mention also that when $q >> k$ (for example when $k = 10$) we observe a bigger difference in times, because the first ten trees of the index will be available after first 10-20 queries. Therefore, the next 80 queries will use or extend the index, leading to savings of computational time. On the other hand, the online algorithm does not have any benefits from the first 10-20 queries, since every query is handled with exactly the same way. In case where $q \simeq k$ we observe a slight benefit from our method since some queries (but not many) may ask for the same value of k.

When $k >> q$ we can see that the two algorithms tend to have the same processing time (e.g., for $k = 200$). In this case, only few queries will ask for same k value. One interesting observation here, is that for some graphs (YouTube, Orkut) our method never exceeds index's time, even for the maximum value of k. Instead, in other graphs (Wiki, LiveJournal), we observe that there is a value of k for which the index-based approach shows better performance. This behavior is attributed to the fact that denser graphs tend to contain more cohesive communities. Therefore, when we ask for high cohesive influential communities we still need to traverse a small part of graph. Instead, if only a few such communities exist (e.g., 3-15) then a single query asking for them will end up traversing the whole graph. As a result, we may far exceed index's total time.

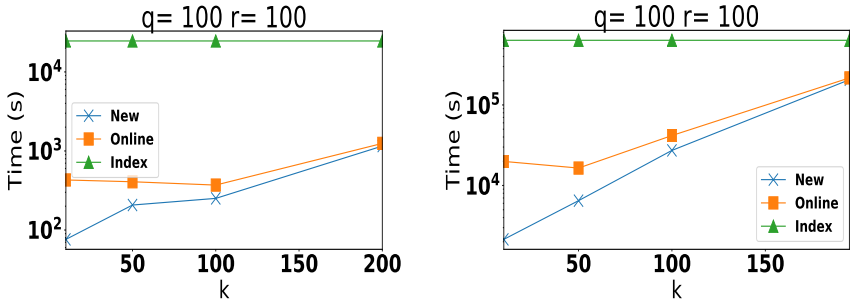
5.2.2 Impact of the number of queries (q). . We vary the number of queries (q) that users make for each graph. We compare the online algorithm with our approach. We fix maximum $k = 50$ and $r = 100$. Searching for low cohesiveness communities makes online algorithms run faster since they need to search in a smaller part of the graph. So fixing a large k just wastes time from our experiments. We want to show how the two different approaches scale as q increases. We do not compare the index method since once we have the index we can answer any query in time linear to the size of the results and that is independent from any previous queries. The results are presented in Figure 6. We can see that for online algorithm the time increases linear to the number of queries, since we do not get any benefit from previous queries. This is not true for our method. The time increases linear until we have a single query for each possible k value. Then the index is used every time (or very frequently) that the time we need is just using the index to extract the top-r-k-influential communities. Clearly an upper bound in time is created as lots of queries occur. We have a logarithmic grow of time.

5.2.3 Impact of the number of communities (r). . In this series of experiments, we vary the number r of communities asked by users. We test how the different algorithms scale when as the value of r increases. We select values from 100 up to 2,000. Note that we expect that the time needed for large values of r will always be at least as the index's time. This happens because if we ask for all or most communities that exist in a graph, we will have to traverse the entire graph. As a result, online algorithms loose their advantage of local search. Therefore, if we are sure that our queries include many thousands and highly cohesive communities, the index is the best solution, provided that we can offer large amounts of main memory. We are interested in studying how quickly the runtime increases as we increase r.

The corresponding results are shown in Figure 7. Larger graphs tend to contain more communities, so we do not notice exponential growth of time, as soon as we keep r relatively low. However,

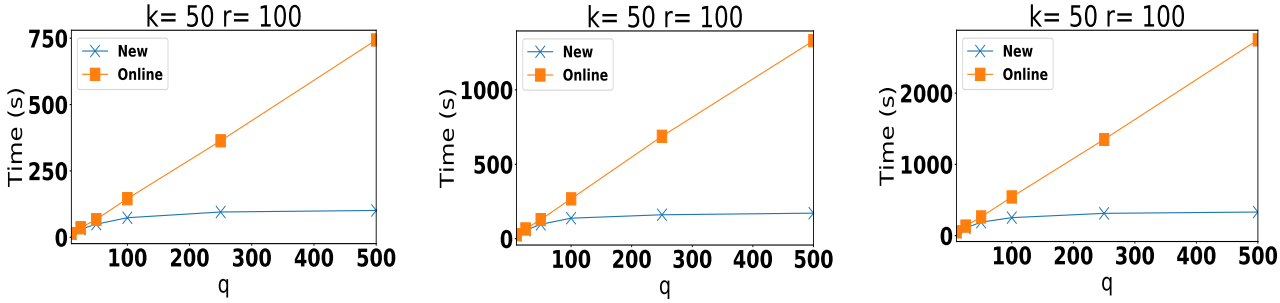


(a) YouTube (b) Wiki (c) LiveJournal

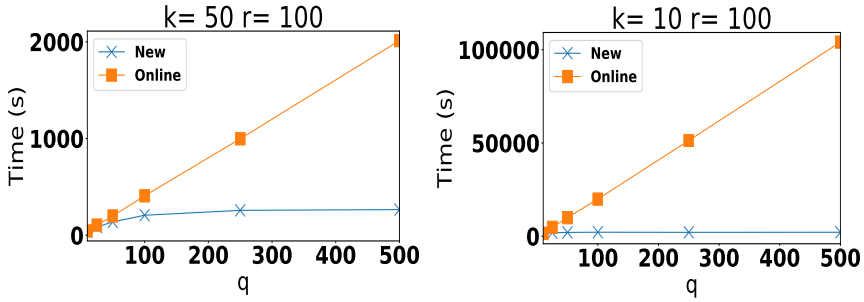


(d) Orkut (e) Friendster

Figure 5: Runtime performance for different values of k.

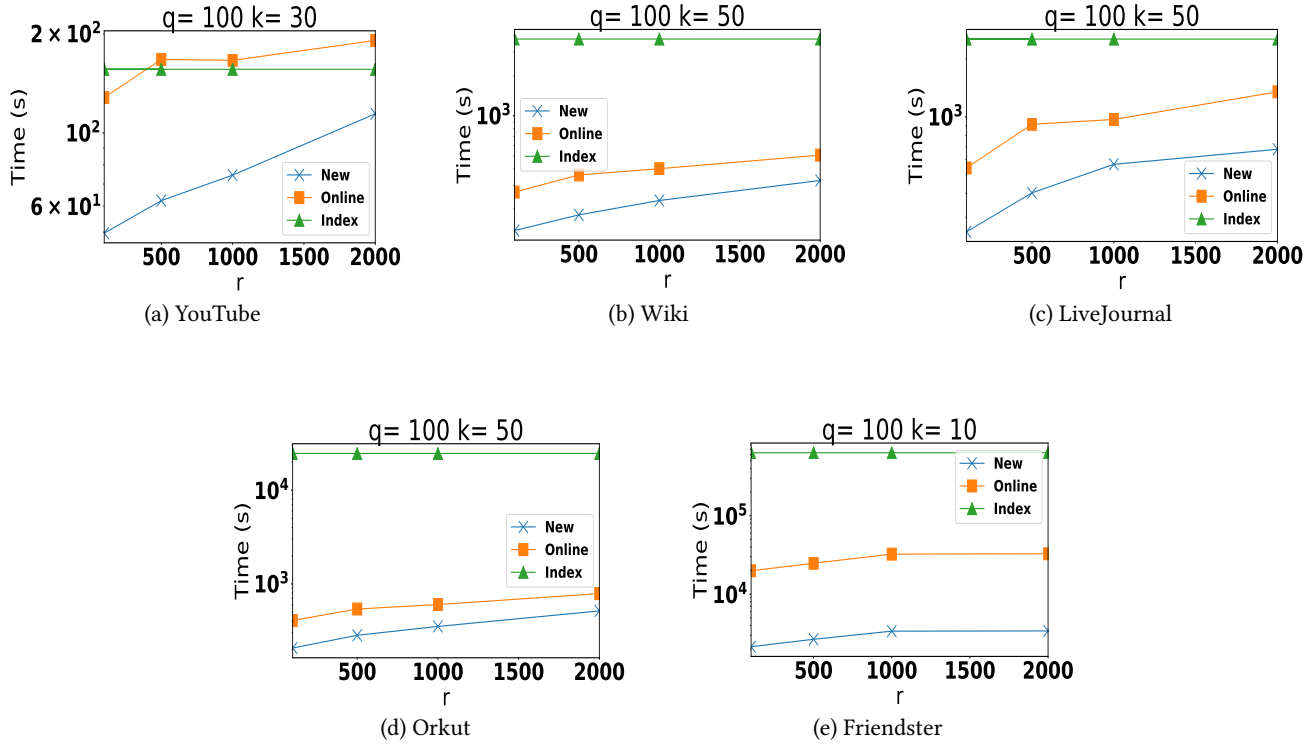


(a) YouTube (b) Wiki (c) LiveJournal



(d) Orkut (e) Friendster

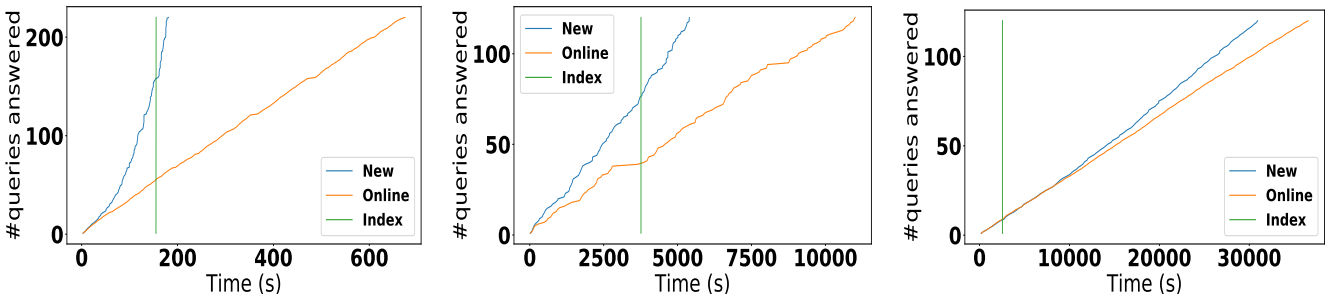
Figure 6: Runtime performance for different values of q.



(a) YouTube (b) Wiki (c) LiveJournal

(d) Orkut (e) Friendster

Figure 7: Runtime performance for different values of r.



(a) YouTube (b) Wiki (c) LiveJournal

Figure 8: Number of queries completed vs. the required runtime (in sec.) to build the whole index.

YouTube is an example of exponential time growth, since it only contains about 5,000 30-influential communities and we ask up to 2,000 communities. So, after first 500-1,000 communities, a significantly larger part of the graph is traversed to answer next queries.

5.2.4 Impact of incrementally building the full index. . In this experiment we try to simulate a different scenario of random queries. To do so, we change the distribution of parameters k and r. We use the Gaussian distribution for the cohesiveness value k, as it is more possible that users will be interested for communities with medium cohesiveness. Queries with very high or very low k value will still be existing, but will occur with a lower probability. Furthermore, we use the power-law distribution for the number r that queries contain, as most users are interested only for some communities and queries that ask for all or many thousands communities will be rare. So, queries contain the whole range of k and r parameters for each graph. The plots present the number of queries that are answered over time. Note that as we do not restrict the value of r with an upper bound, the maximum number of k-influential communities will be asked (for each k) over time. This will result into finally building the whole index. As we make this process incrementally, it is expected that it will take more time than the straightforward building of the index. However, this is not a usual case, as top-r k-influential community detection problems is not about detecting all possible communities, but the value of r is rather small.

Previous studies (online and index based) usually vary r up to 320 in most experiments and up to 1,000 in one case [3, 21]. In our previous experiments we varied r up to 2,000 and we have

shown the superiority of our approach in most cases. Results in Figure 8 show that that we can far exceed index method time, as we expected. Nevertheless, we aim to see how many queries can be answered before we exceed index's contraction time. Since it is a time consuming experiment, we present results only for our three medium-sized graphs. We observe that for YouTube we are able to answer about 150 queries before we reach index's time. For Wiki this number is reduced almost in half (about 80 queries) and for LiveJournal we are not able to answer more than 20 queries. The results depend on the properties of each graph and the time that first tough queries arrive. Once again, it is evident that the online algorithm does not scale well as the number of queries increases.

5.2.5 Index size. . In the next series of experiments, we compare the total size of the index that is constructed for the two different methods, i.e., the index-based and the proposed one. We do not have to compare the online method, since it does not use any index at all. We execute 100 queries that ask for up to top 2,000 k-influential communities for each possible value of k. A large value of r (equal to 2,000) ensures that parts of the index created by our method have a sufficient size. However, we expect that the total size will be even less in a more realistic scenario.

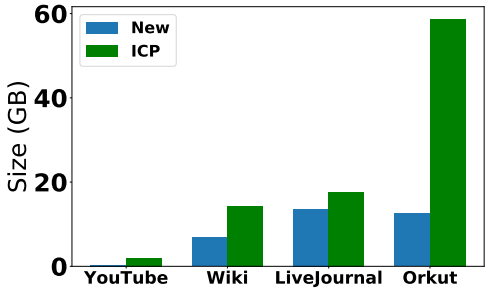


Figure 9: Index size for different methods and graphs.

The performance results presented in Figure 9 demonstrate that the proposed technique uses less main memory than ICP. This is reasonable and expected, as we incrementally build the index based on users' interests – avoiding a significant part of it. A significant advantage of this method is that it shows great scalability in very large graphs. Please note that this is one of the most significant problems of index-based method – it is impractical for very large graphs, especially for a client-based machine [6]. Therefore, building the index incrementally overcomes this challenge – as long as we assume that users will never ask for all influential communities. Thus, memory usage is another major factor that we need to consider even if incrementally building the index costs more in time.

5.3 Discussion

The above experiments clearly prove that when we are interested in querying for top-r k-influential communities in a graph, the proposed method outperforms the online algorithm in all cases. This happens because the online algorithm does not make use of any relations between different queries. Instead, our approach (sometimes) will make use of the index that builds even if $q < k$, saving computational time in contrast to the online algorithm.

Another important observation for the online algorithm is that it does not scale well as the number of queries increases. The time keeps increasing linearly and this is not acceptable for a strategy that aims to answer multiple queries. Instead, the proposed approach will scale after a satisfied number of queries, since the index will be used to extract the influential communities. On the other hand, the problem of scalability as queries arrive does not exist for the index based method and in some cases, especially when the value of k is large, building the whole index requires less time. This is true when queries contain k and r values such that online algorithm needs to traverse almost the whole graph in order to provide the answer. Therefore, based on the maximum value k that users ask, we should decide between incrementally building the index or building it at once. Note that in very big graphs we will be unable to build the whole index, since this would require large amounts of main memory.

An upcoming question is if we can determine the values of k and r towards deciding if the index would be preferred against the proposed approach. We are not able to easily answer this question, as maximum k-core, density of graphs and other properties seem to affect the performance of the different algorithms studied. Therefore, a cost model would be necessary, to be able to estimate the computational cost as a function of k and r. In addition, another important aspect of the cost is the main memory footprint required. We plan to investigate these issues in the near future, since the existence of a cost model will enable the use of the available algorithms in a more efficient manner and also will allow the implementation of incremental influential community detection algorithms inside data analytics engines.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we study incremental processing of top-r k-influential community detection. In this setting, r denotes the number of communities and k denotes the degree of cohesiveness that communities must satisfy. In general there are two different directions to attack the problem: *i*) the *online approach*, which computes influential communities in increasing influence value order, and *ii*) the *index-based approach*, which pre-computes influential communities and stores appropriate information in a tree-based index structure.

In this work, we propose a combination of online and index-based approaches in order to design a novel technique that preserves only the advantages of each method and incrementally detects influential communities in response to users' queries. Therefore, we study how online algorithms can be used to build the index partially in time linear with respect to the size of the subgraph that contains the required number of influential communities. Finally, we offer extensive performance evaluation results, based on real-life networks, demonstrating the superiority of the proposed approach. In all interesting cases, the proposed approach shows excellent runtime and scalability performance.

There are several interesting future research directions that may extend the techniques presented in this paper such as:

i) the exploitation of parallel/distributed systems towards more efficient processing,
ii) the use of other criteria for quantifying the influence of a community and
iii) the adaptation of the proposed techniques for the dynamic case, where the underlying network evolves in time.

ACKNOWLEDGMENTS

This work was supported financially by King Abdullah University of Science and Technology (KAUST), Saudi Arabia. In particular, Klearchos Kosmanos (the first author) was a Research Intern in KAUST during the period June 2020 - August 2020 working in the topic of Incremental Influential Community Search.

REFERENCES

[1] Nicola Barbieri, Francesco Bonchi, Edoardo Galimberti, and Francesco Gullo. 2015. Efficient and effective community search. *Data Min. Knowl. Discov.* 29, 5 (2015), 1406–1433. https://doi.org/10.1007/s10618-015-0422-1

[2] Vladimir Batagelj and Matjaz Zaversnik. 2003. An O(m) Algorithm for Cores Decomposition of Networks. *CoRR* cs.DS/0310049 (2003). http://arxiv.org/abs/cs/0310049

[3] Fei Bi, Lijun Chang, Xuemin Lin, and Wenjie Zhang. 2018. An Optimal and Progressive Approach to Online Search of Top-K Influential Communities. *Proc. VLDB Endow.* 11, 9 (2018), 1056–1068. https://doi.org/10.14778/3213880.3213881

[4] Lijun Chang, Xuemin Lin, Lu Qin, Jeffrey Xu Yu, and Wenjie Zhang. 2015. Index-based Optimal Algorithms for Computing Steiner Components with Maximum Connectivity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 459–474. https://doi.org/10.1145/2723372.2746486

[5] Lijun Chang, Jeffrey Xu Yu, Lu Qin, Xuemin Lin, Chengfei Liu, and Weifa Liang. 2013. Efficiently computing k-edge connected components via graph decomposition. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, 205–216. https://doi.org/10.1145/2463676.2465323

[6] Shu Chen, Ran Wei, Diana Popova, and Alex Thomo. 2016. Efficient Computation of Importance Based Communities in Web-Scale Networks Using a Single Machine. In *Proceedings of the 25th ACM International Conference on Information and Knowledge Management, CIKM 2016, Indianapolis, IN, USA, October 24-28, 2016*, Snehasis Mukhopadhyay, ChengXiang Zhai, Elisa Bertino, Fabio Crestani, Javed Mostafa, Jie Tang, Luo Si, Xiaofang Zhou, Yi Chang, Yunyao Li, and Parikshit Sondhi (Eds.). ACM, 1553–1562. https://doi.org/10.1145/2983323.2983836

[7] James Cheng, Linhong Zhu, Yiping Ke, and Shumo Chu. 2012. Fast algorithms for maximal clique enumeration with limited memory. In *The 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '12, Beijing, China, August 12-16, 2012*, Qiang Yang, Deepak Agarwal, and Jian Pei (Eds.). ACM, 1240–1248. https://doi.org/10.1145/2339530.2339724

[8] Jonathan Cohen. 2009. Graph Twiddling in a MapReduce World. *Comput. Sci. Eng.* 11, 4 (2009), 29–41. https://doi.org/10.1109/MCSE.2009.120

[9] Wanyun Cui, Yanghua Xiao, Haixun Wang, Yiqi Lu, and Wei Wang. 2013. Online search of overlapping communities. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, 277–288. https://doi.org/10.1145/2463676.2463722

[10] Yixiang Fang, Reynold Cheng, Xiaodong Li, Siqiang Luo, and Jiafeng Hu. 2017. Effective Community Search over Large Spatial Graphs. *Proc. VLDB Endow.* 10, 6 (2017), 709–720. https://doi.org/10.14778/3055330.3055337

[11] Yixiang Fang, Xin Huang, Lu Qin, Ying Zhang, Wenjie Zhang, Reynold Cheng, and Xuemin Lin. 2020. A survey of community search over big graphs. *VLDB J.* 29, 1 (2020), 353–392. https://doi.org/10.1007/s00778-019-00556-x

[12] Santo Fortunato. 2010. Community detection in graphs. *Physics Reports* 486, 3-5 (2010), 75 – 174. https://doi.org/DOI:10.1016/j.physrep.2009.11.002

[13] Jiafeng Hu, Xiaowei Wu, Reynold Cheng, Siqiang Luo, and Yixiang Fang. 2017. On Minimal Steiner Maximum-Connected Subgraph Queries. *IEEE Trans. Knowl. Data Eng.* 29, 11 (2017), 2455–2469. https://doi.org/10.1109/TKDE.2017.2730873

[14] Xiaocheng Hu, Yufei Tao, and Chin-Wan Chung. 2013. Massive graph triangulation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, 325–336. https://doi.org/10.1145/2463676.2463704

[15] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-truss community in large and dynamic graphs. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 1311–1322. https://doi.org/10.1145/2588555.2610495

[16] Xin Huang and Laks V. S. Lakshmanan. 2017. Attribute-Driven Community Search. *Proc. VLDB Endow.* 10, 9 (2017), 949–960. https://doi.org/10.14778/3099622.3099626

[17] Xin Huang, Laks V. S. Lakshmanan, and Jianliang Xu. 2017. Community Search over Big Graphs: Models, Algorithms, and Opportunities. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*. IEEE Computer Society, 1451–1454. https://doi.org/10.1109/ICDE.2017.211

[18] Xin Huang, Laks V. S. Lakshmanan, Jeffrey Xu Yu, and Hong Cheng. 2015. Approximate Closest Community Search in Networks. *Proc. VLDB Endow.* 9, 4 (2015), 276–287. https://doi.org/10.14778/2856318.2856323

[19] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*. www.cidrdb.org, 68–78. http://cidrdb.org/cidr2007/papers/cidr07p07.pdf

[20] Hu Jiafeng, Wu Xiaowei, Cheng Reynold, Luo Siqiang, and Fang Yixiang. 2017. On Minimal Steiner Maximum-Connected Subgraph Queries. *tkde* 29, 11 (2017), 2455–2469. https://doi.org/10.1109/TKDE.2017.2730873

[21] Rong-Hua Li, Lu Qin, Jeffrey Xu Yu, and Rui Mao. 2015. Influential Community Search in Large Networks. *Proc. VLDB Endow.* 8, 5 (2015), 509–520. https://doi.org/10.14778/2735479.2735484

[22] Min Chih Lin, Francisco J. Soulignac, and Jayme Luiz Szwarcfiter. 2012. Arboricity, h-index, and dynamic algorithms. *Theor. Comput. Sci.* 426 (2012), 75–90. https://doi.org/10.1016/j.tcs.2011.12.006

[23] Lu Qin, Rong-Hua Li, Lijun Chang, and Chengqi Zhang. 2015. Locally Densest Subgraph Discovery. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, NSW, Australia, August 10-13, 2015*, Longbing Cao, Chengqi Zhang, Thorsten Joachims, Geoffrey I. Webb, Dragos D. Margineantu, and Graham Williams (Eds.). ACM, 965–974. https://doi.org/10.1145/2783258.2783299

[24] Ahmet Erdem Sariyüce and Ali Pinar. 2016. Fast Hierarchy Construction for Dense Subgraphs. *Proc. VLDB Endow.* 10, 3 (2016), 97–108. https://doi.org/10.14778/3021924.3021927

[25] Mauro Sozio and Aristides Gionis. 2010. The community-search problem and how to plan a successful cocktail party. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, July 25-28, 2010*, Bharat Rao, Balaji Krishnapuram, Andrew Tomkins, and Qiang Yang (Eds.). ACM, 939–948. https://doi.org/10.1145/1835804.1835923

[26] Hanghang Tong and Christos Faloutsos. 2006. Center-piece subgraphs: problem definition and fast solutions. In *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, August 20-23, 2006*, Tina Eliassi-Rad, Lyle H. Ungar, Mark Craven, and Dimitrios Gunopulos (Eds.). ACM, 404–413. https://doi.org/10.1145/1150402.1150448

[27] Jia Wang and James Cheng. 2012. Truss Decomposition in Massive Networks. *Proc. VLDB Endow.* 5, 9 (2012), 812–823. https://doi.org/10.14778/2311906.2311909

[28] Nan Wang, Jingbo Zhang, Kian-Lee Tan, and Anthony K. H. Tung. 2010. On Triangulation-based Dense Neighborhood Graphs Discovery. *Proc. VLDB Endow.* 4, 2 (2010), 58–68. https://doi.org/10.14778/1921071.1921073

[29] Yubao Wu, Ruoming Jin, Jing Li, and Xiang Zhang. 2015. Robust Local Community Detection: On Free Rider Effect and Its Elimination. *Proc. VLDB Endow.* 8, 7 (2015), 798–809. https://doi.org/10.14778/2752939.2752948

[30] Jierui Xie, Stephen Kelley, and Boleslaw K. Szymanski. 2013. Overlapping community detection in networks: The state-of-the-art and comparative study. *ACM Comput. Surv.* 45, 4 (2013), 43:1–43:35. https://doi.org/10.1145/2501654.2501657