Efficient Computation of Importance-Based Communities in Web-Scale Networks Using a Single Machine

Shu Chen, Ran Wei, Diana Popova, Alex Thomo
University of Victoria, British Columbia, Canada
{shuc12,ranwei,dpopova,thomo}@uvic.ca

ABSTRACT

Finding decompositions of a graph into a family of communities is crucial to understanding its underlying structure. Algorithms for finding communities in networks often rely only on structural information and search for cohesive subsets of nodes. In practice however, we would like to find communities that are not only cohesive, but also influential or important. In order to capture such communities, Li, Qin, Yu, and Mao introduced a novel community model called "k-influential community" based on the concept of k-core, with numerical values representing "influence" assigned to the nodes. They formulate the problem of finding the top-r most important communities as finding r connected k-core subgraphs ordered by the lower-bound of their importance. In this paper, our goal is to scale-up the computation of top-r, k-core communities to web-scale graphs of tens of billions of edges. We feature several fast new algorithms for this problem. With our implementations, we show that we can efficiently handle massive networks using a single consumer-level machine within a reasonable amount of time.

1. INTRODUCTION

One of the most important tasks in analyzing graphs is finding communities of nodes that have close ties with each other [11, 14, 18, 32]. Communities are usually conceived as subgraphs with a high density of links within the subgraph and a comparatively lower density of links with the rest of the graph. Discovering communities is of great importance in sociology, biology, computer science, and other disciplines where systems are often represented as graphs [12].

While most of the works on community detection focus on graph structure only, in practice, we would often like to find the communities that are not only well-connected internally, but also important or influential. For example, we would like to discover well-connected communities of *prolific* celebrities, *highly-cited* researchers, *outspoken* individuals, *authoritative* financial analysts, etc. To capture such communities, Li, Qin, Yu, and Mao introduced in [23] a novel community model called "k-influential community" based on the concept of k-core, with values of "influence" or "importance" assigned to the nodes. They formulate the problem of finding the top-r most important communities as finding r connected k-core subgraphs ordered by the lower-bound of their nodes' importance. This model embeds the node importance/influence into the very process of community detection. In this paper, we focus on this community model. Our goal is to scale the computation of influential communities to massive graphs with billions of edges.

Finding communities in a network is typically hard [12]. Straightforward search for the top-r, k-core communities in a large network is impractical because there could be a large number of communities that satisfy the core constraint, and for each community, we need to check its importance. Despite these difficulties, several algorithms for top-r, k-core community detection have been developed in [23] with varying levels of performance and space requirements.

Algorithms in [23]. There are two main algorithms presented in [23]: direct and index-based. The direct algorithm computes the communities based on the core decomposition of the graph for a given k. It progressively removes nodes of least importance and then runs a maximally connected component (MCC) algorithm to discover the next community. The direct algorithm can be used for graphs of moderate size; when the graph is large, the algorithm does not scale. This happens because of the large number of MCC runs.

The index-based algorithm first builds an index. The index can then be used to quickly extract top-r, k-core communities for any r and k. The proposed procedures for building the index are quite efficient; however, the index is a main memory structure and needs space comparable to the size of the original graph. As a result, the index-based solution does not scale for very large graphs. For example, we were unable to construct the index for Clueweb, a dataset of 74 billion edges, using 64GB of memory. To the best of our knowledge, scaling the computation of influential communities to massive graphs of such a scale is an open problem.

Algorithms in this paper. Our goal is to significantly speed-up the direct computation of communities and scale-up to massive graphs with tens of billions of edges. Furthermore, we would like to achieve this using only a consumer-grade machine (in the trend of [19, 22]). In order to make the graph footprint as small as possible, we used Webgraph, a highly efficient, and actively maintained graph compression framework [4].

We propose efficient algorithms that require space in the order of the *compressed* version of the graph. This is an order of magnitude lower than the memory required by the index-based solution of [23], which needs space in the order of the *uncompressed* version of the graph.

Our algorithms can be classified into "forward" and "backward." The forward algorithms recursively peel off the k-core subgraph of the graph starting from the least important nodes. Along the way, we maintain proper bookkeeping information so that MCC runs are minimized or eliminated. In contrast, the backward algorithms process the graph starting from the most important nodes first. While the forward

algorithms compute communities from the least to the most important, the backward algorithms compute them in the reverse order, from the most important to the least. More specifically, our contributions are as follows.

1. We present two fast, forward algorithms for computing top-r, k-core communities. We minimize the number of MCC computations and achieve orders of magnitude speed-up compared to the direct algorithm of [23].
2. We present an even faster forward algorithm for computing top-r, k-core disjoint communities, which completely eliminates MCC computations.
3. We present backward algorithms for fast computing of the most important communities. When the graph is big and r relatively small, these algorithms perform best and produce the result by only accessing a small portion of the graph.
4. We present extensive experiments on large and very large graphs. Our biggest graph is Clueweb with about 1 billion nodes and 74 billion edges. Our results show that we are able to compute communities for every combination of k and r in a large range of values using the forward algorithms. We can do this faster for a good number of k and r combinations using the backward algorithms.

2. PRELIMINARIES

We represent networks using undirected graphs. We denote an undirected graph by $G = (V, E)$, where V is the set of vertices, and E is the set of edges. We set n and m to be $|V|$ and $|E|$, respectively. Given a vertex v, we denote the set of its neighbors, $\{u : (u, v) \in E\}$, by $N_G(v)$.

We focus on "cohesive" subgraphs based on the concept of k-core. We say that a subgraph H of G is *k-core* if each vertex in H has degree at least k in H. The *maximal k-core* of G is the largest such subgraph. We denote the maximal k-core of G by $C_k(G)$. In general, $C_k(G)$ is not necessarily connected, i.e. it can contain several maximally connected components (MCC's). For $1 \leq i < j$, if $C_i(G) \neq \emptyset$, then $C_i(G) \supset C_j(G)$.

2.1 Importance-based Communities

Consider a graph $G = (V, E)$; we follow the community model of [23], which is based on the notion of k-core cohesive subgraphs. Furthermore, an importance weight array w of size n is given, such that $w[v]$ is the weight of $v \in V$. These weights can represent centrality scores, h-index, wealth, social status, age, etc. In practice, it is easy to derive such weights from attributes attached to nodes in real networks or compute them using auxiliary information (e.g. p- or h-index). Also, a strict total order is assumed on the weights of array w; this can be easily achieved by breaking ties based on the lexicographical order of vertex ids. Given a subgraph $H = (V_H, E_H)$ of G, the weight of H is defined to be the lower-bound of its vertex weights, i.e. $W(H) = \min\{w[v] : v \in V_H\}$. The idea of the influential community model of [23] is to extract cohesive subgraphs of high importance weight.

More precisely, given k and r, the *influential community problem* is to discover the top-r (w.r.t. weight) maximally connected, k-core subgraphs of $C_k(G)$.

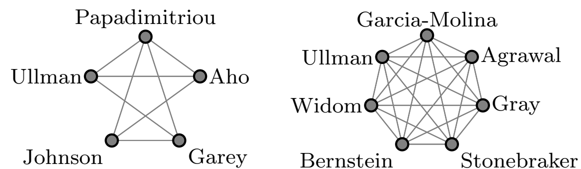


Figure 1: Arnet: [left] top-1, $k = 3$, [right] top-1, $k = 6$

As a real example, using the authorship network from ArnetMiner (http://arnetminer.org) and weighing authors by their p-index ([?]), we obtain for $k = 3$ and $k = 6$ the top-1 communities shown in Fig. 1. The influence of the people in those communities is undisputed, with the first community having a higher p-index lower-bound than the second. In general, k can serve as tradeoff between cohesiveness and importance. The higher the k, the higher the cohesiveness, but the lower the importance of computed communities can be.[1]

Such communities can be discovered by "peeling off" $C_k(G)$ as follows. Let us set $C_{k,1} = C_k(G)$, and let $v_{k,1}$ be the minimum weight vertex in $C_{k,1}$. We define the 1st influential community, $H_{k,1}$, as the maximally connected component (MCC) of $C_{k,1}$ containing $v_{k,1}$. Then, we peel off $v_{k,1}$ by recursively deleting it and its neighbors whose degree in $C_{k,1}$ falls below k during deletions. Let $C_{k,2}$ be what remains of $C_{k,1}$ after (recursively) peeling off $v_{k,1}$. Clearly, $C_{k,2}$ is k-core. We repeat the same process now on $C_{k,2}$ and obtain the 2nd influential community $H_{k,2}$.

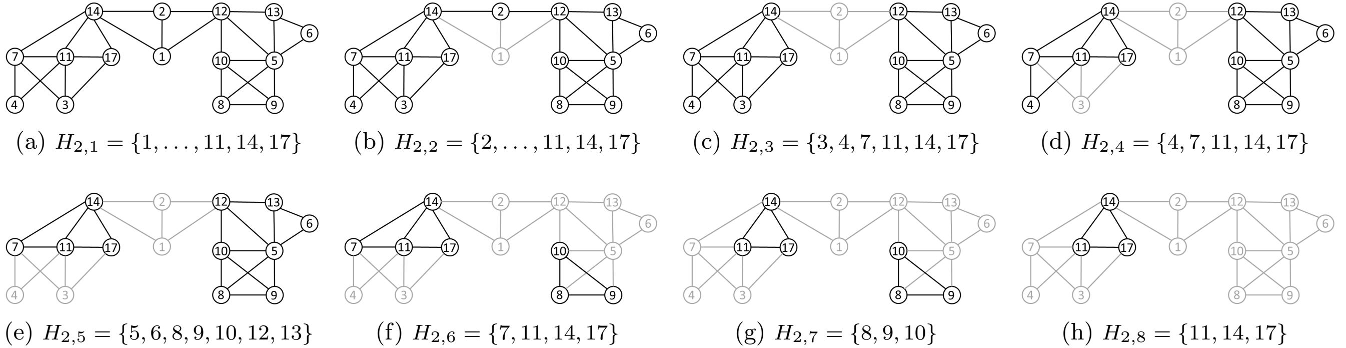
In general, let $C_{k,i}$ be what remains of $C_{k,i-1}$, for $i \geq 2$, after peeling off the minimum weight vertex in $C_{k,i-1}$. Let $v_{k,i}$ be the minimum weight vertex in $C_{k,i}$. We define the ith influential community, $H_{k,i}$, as the MCC of $C_{k,i}$ containing $v_{k,i}$. We continue like this until $C_{k,i}$ becomes empty for some i. For each $i \geq 2$, such that $C_{k,i} \neq \emptyset$, we have $W(H_{k,i}) > W(H_{k,i-1})$. We output the last (top) r communities.

To illustrate the above, see Fig. 2. For simplicity, the weights of vertices are set to be their ids. We see $C_{k,i}$'s in black, for $k = 2$ and $i \in [1, 8]$, for the graph in Fig. 2a. The grayed out vertices and edges are deleted during peel offs (recursive deletes). For example, when we delete vertex 5 in $C_{2,5}$ (Fig. 2e), vertices 6, 12, and 13 are recursively deleted as well (grayed-out in Fig. 2f). $H_{k,i}$ is the MCC of $C_{k,i}$ containing the vertex of the smallest weight $v_{k,i}$. For example, for $C_{2,7}$ (Fig. 2g), $v_{2,7} = 8$ and $H_{2,7} = \{8, 9, 10\}$.

Subgraphs $H_{k,i}$, being MCC's of $C_{k,i}$, are all k-core. We call them *connected-cohesive-important* (CCI) communities. It can be verified that either $H_{k,i} \supset H_{k,j}$, for $1 \leq i < j$, or $H_{k,i} \cap H_{k,j} = \emptyset$. The first case happens when $v_{k,i}$ is in the same MCC in $C_{k,i}$ as $v_{k,j}$, whereas the second happens when they are not. There are possibly several chains of such $\supset$ containments. For Fig. 2, we have two such chains, $H_{2,1} \supset H_{2,2} \supset H_{2,3} \supset H_{2,4} \supset H_{2,6} \supset H_{2,8}$, and $H_{2,1} \supset H_{2,2} \supset H_{2,5} \supset H_{2,7}$. The last CCI community in each chain does not contain any other CCI community. We call them *non-containing* CCI communities. They are interesting because they represent the nuclei of bigger communities.

We define two top-r CCI community problems. Specifically, given a graph G, and two positive integers k and r, the first problem (**P1**) is to compute the top-r CCI communities, and the second problem (**P2**) is to compute the top-r *non-containing* CCI communities.

[1]See [23] for more case studies showing the benefits of the proposed model as well as its advantages over other models.



(a) $H_{2,1} = \{1, \dots, 11, 14, 17\}$ (b) $H_{2,2} = \{2, \dots, 11, 14, 17\}$ (c) $H_{2,3} = \{3, 4, 7, 11, 14, 17\}$ (d) $H_{2,4} = \{4, 7, 11, 14, 17\}$

(e) $H_{2,5} = \{5, 6, 8, 9, 10, 12, 13\}$ (f) $H_{2,6} = \{7, 11, 14, 17\}$ (g) $H_{2,7} = \{8, 9, 10\}$ (h) $H_{2,8} = \{11, 14, 17\}$

Figure 2: $C_{k,i}$'s, for $k = 2$ and $i \in [1, 8]$, for the graph in (a) are in black. The grayed out vertices and edges are deleted. Weights are the vertex id's. $H_{k,i}$'s are the MCC's containing the vertex of the smallest weight.

The peel-off procedure we described earlier is in fact the direct algorithm proposed in [23].[2] We call this algorithm *C-original*, and it solves problem **P1**. We discuss solutions for **P2** later in the paper. The bottleneck of C-original is the very large number of MCC computations it executes starting from each minimum-weight $v_{k,i}$. Along the way, we need to keep a cache of the last r CCI communities thus discovered. This is because the communities are generated in reverse order of their importance. In the next section, we present new algorithms for **P1** and **P2** that drastically reduce the number of MCC computations or completely eliminate them.

3. FORWARD ALGORITHMS FOR P1

Let us first analyze algorithm C-original described in the previous section. Since the complexity of *MCC* is $O(m)$ and we compute it for each min-weight vertex $v_{k,i}$, the complexity of C-original is $O(m \cdot n)$, which is impractical for big graphs. Regarding space complexity, observe that we need to remember each time the last r communities computed so far. Since we only store the vertices of these communities, the space complexity is $O(m + n \cdot r)$. For small r (say not more than 30), we can say that the second term $n \cdot r$ is absorbed by the first, m. However, for larger r's, $n \cdot r$ becomes eventually bigger than m. Therefore, the algorithm has also a memory bottleneck. In the following we describe our proposed algorithms for P1. They outperform C-original by orders of magnitude.

Our First Approach. What takes most of the time in C-original is computing MCC's for each $C_{k,i}$. The early $C_{k,i}$ are especially expensive as they are quite big in size. Furthermore, many $C_{k,i}$ are just *slightly* smaller than the previous one, $C_{k,i-1}$.

We observed in practice that for most of the early iterations, the peel-off process does not remove more than few vertices each time. We can observe this fact even in the small example we presented in Fig. 2 (see the grayed out vertices in (2a)-(2e)). Therefore, many of the *MCC* computations are performed on almost identical graphs.

Peel-offs are performed by a recursive delete procedure, *RDelete*, which takes as parameters a k-core subgraph C and a vertex v. It deletes v from C, then recursively deletes all v's neighbors whose degree becomes less than k, and continues so until there are no more vertices with degree less than k. In the end, what remains of C is either a k-core subgraph, or an empty graph. *RDelete* is inexpensive to compute, especially for the early iterations. The total time spend on all *RDelete* calls together is $O(m)$, i.e. not more than just traversing the graph. So, the bottleneck is in fact in the *MCC* computations.

We pose the following question: How can we reduce *MCC* computations? Towards this goal, we observe that we only need to run *MCC* for the last r iterations because it is only those iterations that produce the top-r results.

The problem is though that we do not know beforehand how many iterations there will be in total. However, this can be found by running twice the logic of vertex removals. The new algorithm, C1, is given in Alg. 1. The first run of vertex removals (lines 1-5) does not call *MCC* at all. It selects the min-weight vertex $v_{k,i}$ (variable v) from the current $C_{k,i}$ (variable C), then peels off $v_{k,i}$ by calling *RDelete*, and records the iteration by incrementing variable i. The purpose of this part is to find out how many iterations are needed. The final value of i will be the total number of iterations. The second run of vertex removals (lines 6-13) starts anew and knowing i, only calls *MCC* in the last r iterations. C is reinitialized before the second run. Now, we use j instead of i, and only call *MCC* when $j > i - r$. In total, we run *MCC* for only r times, which happens during the last r iterations. Based on the above discussion, we can verify the following conclusion.

THEOREM 1. *Algorithm C1 correctly computes all the top-r CCI communities of a given graph G.*

Since we run *MCC* only r times, we have that

THEOREM 2. *The time complexity of C1 is $O(m \cdot r)$.*

This is much smaller than $O(m \cdot n)$ for practical values of r. Also, $O(m \cdot r)$ is only a lose upper bound for C1 because the last r iterations operate on very small graphs obtained after deleting most of the vertices in the first $i - r$ iterations. Therefore the *MCC* computations of the last r iterations cost significantly less than $O(m)$ in practice.

In our experiments, we observe C1 to be orders of magnitude faster than C-original.

Regarding space complexity, observe that we do not need anymore to remember the last r communities computed so far. This is because we only compute the very last r communities, which not only are the smallest r communities, but can also be printed (or saved) out right away. Therefore we state the following theorem.

[2] Algorithm 2 in [23]

THEOREM 3. *The space complexity of C1 is $O(m)$.*

In practical terms, we only need space to store a compressed version of the graph. Modern compression frameworks reduce the footprint of real graphs near an order of magnitude.

Algorithm 1 Top-r CCI communities (C1)

Input: G, w, k, r
Output: $H_{k,p-r+1}$, $\ldots$, $H_{k,p}$
1: $C \leftarrow C_k(G)$, $i \leftarrow 1$
2: **while** $C \neq \emptyset$ **do**
3: Let v be the minimum-weight vertex in C
4: $RDelete(C,v)$
5: $i \leftarrow i + 1$
6: $C \leftarrow C_k(G)$, $j \leftarrow 1$
7: **while** $C \neq \emptyset$ **do**
8: Let v be the minimum-weight vertex in C
9: **if** $j > i - r$ **then**
10: $H \leftarrow MCC(C,v)$
11: Output H
12: $RDelete(C,v)$
13: $j \leftarrow j + 1$

Next, we present a better algorithm which reduces the time by a constant factor of (about) 2.

Our Second Approach. The next question we pose is: How to avoid running the second **while** loop and cut the running time in (about) half?

Towards this goal, we introduce a hash-based structure, which we call *iteration-delete-history* and denote by I. This is a hash-table indexed by i, the iteration number. For each i, we store in $I(i)$ a list of deleted vertices in iteration i.

For an illustration, consider Fig. 2. For this example, we have 8 iterations, and $I(1) = \{1\}$, $I(2) = \{2\}$, $I(3) = \{3\}$, $I(4) = \{4\}$, $I(5) = \{5, 6, 12, 13\}$, $I(6) = \{7\}$, $I(7) = \{8, 9, 10\}$, $I(8) = \{11, 14, 17\}$.

The algorithm is given in Alg. 2. Structure I is populated during the run of the **while** loop. More specifically, it is populated in a modified *RDelete* procedure. The modified *RDelete*, which we call *RDelete2*, takes two extra parameters, I and i, and has one extra operation, the insertion of v to $I(i)$. Since the procedure is recursive, all the deleted vertices in iteration i will be inserted into $I(i)$. We implemented I as a flat array of dimension n accompanied by another array storing the positions of bucket boundaries. Since the buckets of I are filled out in order of increasing i, each operation on I takes constant time.

We do not execute any *MCC* computation in the **while** loop of Alg. 2. Once the **while** loop completes, we start running the necessary r *MCC* computations in the subsequent **for** loop. However, since the vertices are deleted at this point, we need each time to make some vertices *alive* again. The **for** loop goes downwards starting from the max iteration number, i, and ending in $i-r+1$. First, we make "alive" the vertices deleted in the last iteration of the **while** loop, then the vertices deleted in the second last iteration, and so on. We record the vertices that become alive in an array called *alive*. Each time we make a set of vertices alive, we run an *MCC* computation. The *MCC* computation works on the original graph, G, consulting array *alive* as it performs a DFS. Only the alive vertices are considered for computing the maximally connected component. It can be verified that algorithm C2 produces the same result as C1, just in reverse order, i.e. $H_{k,p}$, $\ldots$, $H_{k,p-r+1}$. Therefore, we can state the following theorem.

THEOREM 4. *Algorithm C2 correctly computes all the top-r CCI communities of a given graph G.*

The asymptotic time complexity of C2 is the same as that of C1; however, in terms of constant factors, C2 is about twice faster than C1. For the space complexity, observe that structure I takes $O(n)$ space, which is absorbed by $O(m)$ needed to hold the graph (typically true for a compressed graph as well). Therefore, we state the following theorem.

THEOREM 5. *The space complexity of C2 is $O(m)$.*

Algorithm 2 Top-r CCI communities (C2)

Input: G, w, k, r
Output: $H_{k,p}$, $\ldots$, $H_{k,p-r+1}$
1: $C \leftarrow C_k(G)$, $i \leftarrow 1$, $I \leftarrow \emptyset$
2: **while** $C \neq \emptyset$ **do**
3: Let v be the minimum-weight vertex in C
4: $I(i) \leftarrow \emptyset$
5: $RDelete2(C,v,I,i)$
6: $i \leftarrow i + 1$
7: $alive \leftarrow \mathbf{0}$
8: **for** $j = i$ **downto** $i - r + 1$ **do**
9: **for all** $v \in I(j)$ **do**
10: $alive[v] \leftarrow 1$
11: $v \leftarrow I(i).first()$
12: $H \leftarrow MCC(G, v, alive)$
13: Output H

4. FORWARD ALGORITHMS FOR P2

In [23], the computation of *non-containing* (NC) communities is done by modifying C-original to check each time whether upon calling *RDelete* in an iteration i all the vertices of $H_{k,i-1}$ (of the previous iteration) are deleted. In such a case, it can be concluded that $H_{k,i-1}$ is a non-containing community. We call this algorithm NC-original; it still calls a MCC procedure to calculate $H_{k,i-1}$. As such, the performance of NC-original is similar to C-original. For big graphs, both of them are not practical.

Here we propose another algorithm that completely eliminates the need to run MCC computations. For this, we observe that all the information we need for non-containing communities is in the iteration-delete-structure, I, that we maintain. Towards this goal, we give the following definition and then a lemma.

DEFINITION 1. *Given a vertex v, the* current degree *is the number of alive neighbors of v.*

We record current degrees in an array d. While a vertex v is alive, $d[v]$ will contain the current degree of v. When v is deleted, $d[v]$ is not updated anymore, i.e. for deleted vertices, d will remember their degree at the time (iteration) of their deletion.

Now, if in some iteration i, we have that for each $v \in I(i)$, $d[v] = 0$, then all the vertices neighboring some vertex in $I(i)$ are gone (already deleted), i.e. the set of vertices in $I(i)$ was the last standing community in a community containment chain. Based on this reasoning we have the following lemma.

LEMMA 1.

1. *For each non-containing $H_{k,i}$, $H_{k,i} = I(i)$.*
2. *Let $i \geq 1$. If for each $v \in I(i)$, $d[v] = 0$, then $I(i)$ is a non-containing CCI.*

Proof. (1) can be verified from the definitions. For (2), suppose $I(i)$ is not a non-containing CCI, i.e. we have that $I(i) \subset H_{k,i}$, and this is a strict containment. Since $H_{k,i}$ is a connected component, there exist at least one edge between some $v \in I(i)$ and some $u \in H_{k,i} \setminus I(i)$. We have that the weight of any vertex in $H_{k,i} \setminus I(i)$ is greater than the weight of any vertex in $I(i)$. Therefore, u is not yet deleted in iteration i. Hence $d[v] \geq 1$, which is a contradiction. □

Our algorithm, NC1, is given in Alg. 3. We also modify *RDelete2* to properly update array d during deletions. We call the modified procedure, *RDelete3*.

NC1 completely eliminates *MCC* computations. It starts by initializing C to $C_k(G)$, and the current vertex degrees to their degrees in C. In the **while** loop, after populating $I(i)$ via *RDelete3*, we check to see whether all the vertices in $I(i)$ have a degree of zero (lines 10-12). If true, then $I(i)$ is a non-containing community. Based on the above reasoning and Lemma 1, we state the following theorem.

THEOREM 6. *Algorithm NC1 correctly computes all the top-r non-containing CCI communities of a given graph G.*

NC1 only iterates once over the graph and the only structure it uses is I. Therefore, we have the following theorem.

THEOREM 7. *The time and space complexity of algorithm NC1 is $O(m)$.*

Algorithm 3 Top-r non-containing communities (NC1)

```
Input: G, w, k, r
Output: Top-r non-containing H_{k,j_max-r+1}, ..., H_{k,j_max}
 1: C ← C_k(G)
 2: for all vertex v of C do
 3:     d[v] = d_C(v)
 4: i ← 1, I ← ∅, j ← 1
 5: while C ≠ ∅ do
 6:     Let v be the minimum-weight vertex in C
 7:     I(i) ← ∅
 8:     RDelete3(C, v, I, i)
 9:     isNC ← true
10:     for all v ∈ I(i) do
11:         if d[v] > 0 then
12:             isNC ← false
13:     if isNC = true then
14:         H ← I(i)
15:         Output H
16:         j ← j + 1
17:         if j > r then
18:             break
19:     i ← i + 1
```

5. BACKWARD ALGORITHMS

So far the algorithms we presented were forward; they were peeling off the graph from the lowest weight vertices to the highest. Such an approach is reasonable when r (in top-r) is big. However, imagine what happens when r is moderate, say we want to see the top-50 communities quickly. With the forward approach, we would need to start working our way up from the smallest weight vertices, and only at the end of the computation be able to see the top communities.

The approach we propose in this section is backward. It starts with a state where all the vertices are initially considered "deleted". Then, in each iteration, we "resurrect" a deleted vertex v of the highest-weight (among the deleted vertices) and see whether v and the other resurrected vertices before v are able to form a k-core. The benefit of this idea is that we can produce top-r communities for moderate r quickly without need to ever process the majority of low weight vertices. For moderate values of r the time needed is better than for the forward approaches as only a small part of the graph is accessed. This is especially pronounced for big graphs. As r grows, the time taken by the two approaches starts converging. Eventually, for some r, the backward approach will take more time than the forward one.

We give the algorithm for the backward computation of CCI communities (C3) in Alg. 4. We start by making all the vertices "deleted". Then we resurrect vertices in order of their importance starting from the most important vertex. Each time, we update the core values of vertices made alive so far. For this we call the *updateCores* procedure, which detects whether the core numbers of the alive vertices have the potential to be updated, and if so, it updates them. Often there is no need to update cores because the vertex just resurrected does not have sufficient connections with the vertices already resurrected. If the vertex just resurrected, say v, happens to have a core value that is greater or equal to k, then we conclude that v is one of the $v_{k,i}$ vertices, and as such, we compute *MCC* starting from v and using only the alive vertices having a core number larger or equal to v.

Algorithm 4 Top-r CCI communities (C3)

```
Input: G, w, k, r
Output: H_{k,p}, ..., H_{k,p-r+1}
 1: for all v ∈ V do
 2:     alive[v] ← false
 3:     cores[v] ← 0
 4: i ← 1
 5: for j = n downto 1 do
 6:     Let v be the maximum-weight deleted vertex in V
 7:     alive[v] ← true
 8:     updateCores()
 9:     if cores[v] ≥ k then
10:         H ← MCC(G, v, cores)
11:         Output H
12:         i ← i + 1
13:         if i > r then
14:             break
```

To show the soundness and completeness of Algorithm 4, we first present the following lemmas.

LEMMA 2. *Let v be the maximum-weight deleted vertex in V that is resurrected in a given iteration. Let i be the greatest index, such that $v \in C_{k,i}$. Suppose that v belongs to a k-core of alive vertices. Then, v is the minimum weight vertex in $C_{k,i}$, i.e. $v = v_{k,i}$.*

Proof. Suppose vertex $v_{k,i}$ is alive and $w[v_{k,i}] < w[v]$. If we recursively-delete $v_{k,i}$ from $C_{k,i}$, we will be left with $C_{k,i+1}$. Vertex v does not belong in $C_{k,i+1}$ because of the premises. Therefore, v has to be deleted when we recursively delete

$v_{k,i}$. This is a contradiction, because v belongs to a k-core of alive vertices, all of which have weight greater than $v_{k,i}$. □

Now, we can show that we do not miss any $v_{k,i}$ in the backward direction. We give the following lemma, whose proof follows directly from the definitions and so we omit it.

LEMMA 3. *Let $i \in [1, p]$. There exists a vertex v, such that $v = v_{k,i}$, and v is in a k-core, which includes v and all the other vertices u with $w[u] > w[v]$.*

Based on lemmas 2 and 3 and the fact that we only produce the top-r results, we can state the following theorem.

THEOREM 8. *Algorithm 4 correctly computes all and only the top-r CCI communities.*

The time complexity of C3 is quadratic from a worst case perspective. This is because we call *updateCores* for each resurrection. However, we have degree conditions in *updateCores* to only look for updates if the resurrected vertex is well connected to the other resurrected vertices, thus reducing the number of updates significantly. In practice, C3 can be much faster than C2 for moderate r and big graphs.

NC communities. For non-containing communities, we can also construct a backward approach. To this end, recall the forward approach of [23] for non-containing communities. In order to determine whether $H_{k,i-1}$ is non-containing, they check if all the vertices of $H_{k,i-1}$ are deleted in the next iteration i. For the backward approach, we will use the same idea, but in a different way. In a nutshell, when we resurrect a vertex v, and it happens to be a min-weight vertex, we compute the corresponding community, say H; then we check to see whether any element of H participates in any community discovered earlier. If not, H is non-containing.

Algorithm 5 Top-r non-containing communities (NC2)

Input: G, w, k, r
Output: Top-r non-containing $H_{k,j_{\max}-r+1}, \ldots, H_{k,j_{\max}}$
1: **for all** $v \in V$ **do**
2: $\quad alive[v] \leftarrow false$
3: $\quad inPC[v] \leftarrow false$
4: $\quad cores[v] \leftarrow 0$
5: $i \leftarrow 1$
6: **for** $j = n$ **downto** 1 **do**
7: $\quad$ Let v be the maximum-weight deleted vertex in V
8: $\quad alive[v] \leftarrow true$
9: $\quad updateCores()$
10: $\quad$ **if** $cores[v] \geq k$ **then**
11: $\quad\quad isNC \leftarrow true$
12: $\quad\quad H \leftarrow MCC(G, v, cores, isNC)$
13: $\quad\quad$ **if** $isNC = true$ **then**
14: $\quad\quad\quad$ Output H
15: $\quad\quad\quad i \leftarrow i + 1$
16: $\quad\quad\quad$ **if** $i > r$ **then**
17: $\quad\quad\quad\quad$ **break**

Our backward algorithm, NC2, is given in Alg. 5. In order to achieve maximum efficiency (which is crucial, especially for big graphs), we opt for a boolean array, *inPC* (**in**-a-**P**reviously-discovered-**C**ommunity), which records the vertices that participate in some community discovered earlier. Using a boolean array makes the complexity of checking whether a vertex participates in a previously discovered community constant. We handle the population of *inPC* and the membership check of vertices in it in a modified MCC procedure (see Alg. 6).

Algorithm 6 MCC with *alive* and *inPC* arrays

1: **procedure** MCC($G, v, alive, inPC, isNC$)
2: $\quad cc \leftarrow \emptyset$
3: $\quad$ *MCC-DFS*($G, v, alive, cc, inPC, isNC$)
4: $\quad$ **return** cc
5: **procedure** MCC-DFS($G, v, alive, cc, inPC, isNC$)
6: $\quad cc.add(v)$
7: $\quad$ **if** $inPC[v] = true$ **then**
8: $\quad\quad isNC \leftarrow false$
9: $\quad$ **else**
10: $\quad\quad inPC[v] \leftarrow true$
11: $\quad$ **for all** $u \in N_G(v)$ **do**
12: $\quad\quad$ **if** $cores[u] \geq k$ & $u \notin cc$ **then**
13: $\quad\quad\quad$ *MCC-DFS*($G, u, alive, cc, inPC, isNC$)

5.0.1 Core Update upon Vertex Resurrection

The *updateCores* procedure needed by algorithms 4 and 5 comes with its own set of challenges. We have two options: either use an incremental core update algorithm, such as the one proposed in [24] or recompute the cores using the Batagelj and Zaversnik (BZ) algorithm [3]. We implemented both and compared them. The incremental core update of [24] considers the addition of each edge separately. Hence, the addition of a vertex triggers a sequence of core updates, one for each edge coming from the added vertex. In our case, we have many vertex resurrections, and it turned out that re-computing the cores using the BZ algorithm was faster (see Section 6).

Modified BZ Algorithm. In order to use the BZ algorithm, we need to properly adapt it so that it remains fast in spite of changing graph parameters (which is the case as we incrementally resurrect vertices). In the following, we give some details about the BZ algorithm and then describe our adaptations.

At a high level, BZ computes the core decomposition by recursively deleting the vertex with the lowest degree. The deletions are not physically done on the graph; an array is used to capture (logical) deletions. The notion of "deleted vertices" in effect of core computations is different from that considered at the start of the backward algorithms, and as such, it is recorded and handled differently.

To achieve high performance, everything needs to be implemented as flat arrays so that each logical deletion costs (precisely) constant time. As shown in [19], using hash-based structures makes the algorithm take orders of magnitude longer to complete.

Vertices are assumed to be numbered sequentially starting from 0. There are several arrays needed for the modified BZ algorithm (ModBZ, Alg. 7). They are as follows.

Array *degrees* records the degree of each vertex considering only alive vertices. This array is global and with a dimension of n, where n is the number of all vertices, alive or not. Array *cores* records at any given time for any alive vertex v the degree of v considering only the alive, and not-yet-deleted by BZ, vertices. In the end, *cores* will contain the core numbers of each vertex considering only alive vertices. We make this array global and with a dimension of n. Array *vert* contains the alive vertices in ascending order of their degrees. We make this array local and with a dimension of *n_alive*, where *n_alive* is the number of alive vertices. Array *pos* contains the indices of the vertices in *vert*, i.e. *pos*[v] is

the position of v in *vert*. We make this array local and with a dimension of *n_alive*. Array *bin* stores the index boundaries of the vertex blocks having the same degree in *vert*. We make it local and with a dimension of *m_alive*, which is the greatest degree in the graph induced by the alive vertices.

In addition to the above arrays, we will need two new arrays for *ModBZ*, *al* and *al_idx*. We make them global with a dimension of n. Array *al* stores the alive vertices. When a vertex v is resurrected, we store v in $al[n_alive]$ and increment *n_alive*. Array *al_idx* contains the indices of the vertices in *al*, i.e. $al_idx[v]$ is the position of v in *al*.

In line 2 of Alg. 7, arrays *vert*, *pos*, and *bin* are initialized. The main algorithm is in lines 3–16. The top for-loop runs for each vertex, 0 to *n_alive*, scanning array *vert*. We obtain a vertex id from *vert*, translate it to an id, v, in the normal $[0, n]$ range, and check whether it is alive. We only continue the computation if v is alive. Since *vert* contains the alive vertices in ascending order of their degrees, and v is the not-yet deleted vertex of the lowest degree, the coreness of v is its current degree considering only the alive, and not-yet-deleted by ModBZ, vertices, i.e. $cores[v]$. Now v is logically deleted. For this, we process each neighbor u of v with $cores[u] > cores[v]$ (see line 8). Vertex u needs to have its current degree, $cores[u]$, decremented (see line 16). However before that, u needs to be moved to the block on the left in *vert* since its degree will be one less. This is achieved in constant time (see lines 9-15). These operations are made possible by the existence of array *al_idx*, which translates vertex ids to the $[0, n_alive]$ range needed by the local arrays. Specifically, u is swapped with the first vertex, w, in the same block in *vert*. Also, the positions of u and w are swapped in *pos*. Then, the block index in *bin* is updated incrementing it by one (line 16), thus losing the first element of the block, u, which becomes the last element of the previous block.

Algorithm 7 Modified BZ algorithm (ModBZ)

```
 1: procedure ModBZ(G)
 2:     initialize(vert, pos, bin, cores, G)
 3:     for all i ← 0 to n_alive do
 4:         v ← al[vert[i]]
 5:         if v not alive then
 6:             continue
 7:         for all alive u ∈ N_G(v) do
 8:             if cores[u] > cores[v] then
 9:                 du ← cores[u], pu ← pos[al_idx[u]]
10:                 pw ← bin[du], w ← al[vert[pw]]
11:                 if u ≠ w then
12:                     pos[al_idx[u]] ← pw
13:                     vert[pu] ← al_idx[w]
14:                     pos[al_idx[w]] ← pu
15:                     vert[pw] ← al_idx[u]
16:                 bin[du]++, cores[u]−−
```

6. EXPERIMENTAL RESULTS

We performed our analysis by extensive experiments on several real-world graphs. The experiments were divided into two parts: first, we evaluated the performance of different algorithms and eliminated from consideration those that were slower than the others by a large degree; and second, we conducted a broad testing of the remaining algorithms, with the goal of finding the best solutions for extracting the most important communities.

Dataset	n	m	$d_{\max}$	$k_{\max}$
AstroPhysics	133.2 K	396 K	504	56
LiveJournal	4.8 M	43 M	20,333	372
UK2002	18.4 M	262 M	194,955	943
Arabic2005	22.7 M	554 M	575,628	3,247
UK2005	39.4 M	783 M	1,776,858	588
Webbase2010	115.6 M	855 M	816,127	1,506
Twitter2010	41.7 M	2,405 M	2,997,487	2,488
Clueweb	978.4 M	74,744 M	75,611,696	4,244

Table 1: Datasets ordered by m. The two last columns give the maximum degree and maximum core number.

We implemented all the algorithms in Java and used Webgraph [4] as a graph compression framework. We chose Webgraph because of excellent compression ratios it achieves and also because it is actively maintained and continuously improved (http://webgraph.di.unimi.it). Webgraph only decompresses on the fly the part of the graph needed to access. The decompression is very fast; we did not observe any noticeable delay compared to an uncompressed hash map graph representation (when the latter could fit in memory).

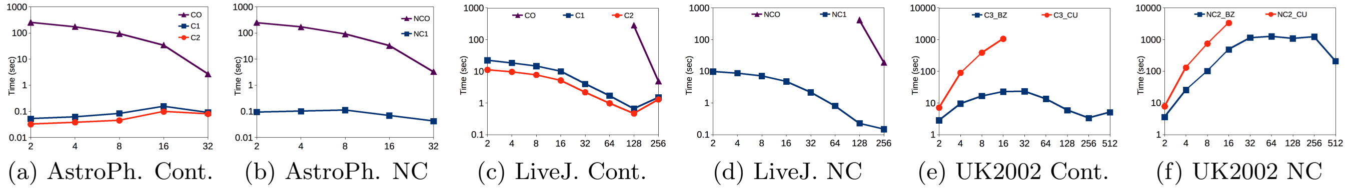
Datasets. The graphs we used were obtained from http://law.di.unimi.it/datasets.php. They vary from medium to massive sizes (see Table 1). Each directed graph was converted to an undirected one by adding for each edge (u, v), its inverse (v, u), if it did not exist; also, self-loops (e.g. (v, v)) were eliminated. The edge numbers in Table 1 refer to this version of the graphs. We assigned random weights to the vertices of each graph. For all the datasets, but Clueweb, a timeout of one hour was set for each algorithm to run. For Clueweb, the timeout was set to 24 hours (albeit we did not need more than a few hours for most of the algorithm runs).

Equipment. The results for the first seven datasets in Table 1 (from AstroPhysics to Twitter2010) were obtained on a consumer-grade laptop: processor 3.4GHz Intel Core i7 (4-core), 16GB RAM, running OS X Yosemite.

Clueweb could not be tested on our laptop because the compressed graph uses about 20GB of space and would not fit into memory.Clueweb was tested on a machine with 2.10GHz Intel(R) Xeon(R) E5-2620 v2 (6-core) CPU and 64GB RAM, running Ubuntu Server 14.04.3 LTS. Note that the processor speed is lower than that of our laptop, but the memory is larger. Despite its big memory, the machine had a price of about $3K, qualifying it as consumer-grade.

Testing Original Algorithms. First, we start by comparing the direct algorithms of [23], C-original (CO) and NC-original (NCO), with our counterparts, C1, C2, and NC1, respectively. We were only able to obtain results for CO and NCO using the first two (moderate) datasets, AstroPhysics and LiveJournal. Fig. 3 shows the results of the comparison: (a), (b) are for AstroPhysics, and (c), (d) are for LiveJournal. C1 and C2 outperform CO, and NC1 outperforms NCO, in both cases by orders of magnitude. For LiveJournal, CO and NCO were only able to produce results for $k = 128$ and $k = 256$ (for these values, C_k was small enough for them to handle). Thus, we eliminate CO and NCO from further testing; they could not produce results on large-scale graphs within a reasonable amount of time.

Testing Core Updates. The *updateCores* procedure used in the backward algorithms (Section 5) was implemented using two different approaches: the incremental core update



(a) AstroPh. Cont. (b) AstroPh. NC (c) LiveJ. Cont. (d) LiveJ. NC (e) UK2002 Cont. (f) UK2002 NC

Figure 3: Original and proposed algorithms on AstroPh. and LiveJ. when varying k ($r = 40$), 3a-3d. BZ versus CU 3e-3f.

Name	Description	Problem
C1	Alg. 1	1
C2	Alg. 2	1
C3_BZ	Alg. 4, Alg. 7	1
NC1	Alg. 3	2
NC2_BZ	Alg. 5, Alg. 7	2

Table 2: Proposed Algorithms

(CU) algorithm proposed by Li, Yu, and Mao in [24], and the modified BZ algorithm, ModBZ (Alg. 7). The first approach was implemented in C3_CU and NC2_CU, and the second in C3_BZ and NC2_BZ. These algorithms were tested extensively on several smaller and medium size graphs. Here, we are presenting the results on UK 2002, which was the largest dataset that we could have the computation completed (at least for several k values) for C3_CU and NC2_CU. Fig. 3 (e) and (f) show that C3_CU and NC2_CU were slower than C3_BZ and NC2_BZ, respectively. After this analysis, we eliminated C3_CU and NC2_CU from further consideration; they could not be feasibly used for community extraction on large-scale graphs in a reasonable amount of time.

Main Testing. The bulk of testing was done for the algorithms in Table 2. We start by presenting test results and analysis on LiveJournal, UK 2002, Arabic 2005, UK 2005, Webbase 2010, and Twitter 2010. We omit results on AstroPhysics as this dataset is relatively small. The results on the largest dataset, Clueweb, are presented separately.

Problem 1: Computing containing communities. Figures 4 and 5 show results for computing containing communities when k and r are varied, respectively.

Analysis of results. (1) The charts clearly show that C2 outperforms C1 for all k and r, on all tested datasets. This is expected because C2 makes only one pass over the graph as opposed to two that C1 does (see Alg. 1 and 2).
(2) The run times of C1 and C2 decrease in general as the k-core subgraph, C_k, becomes smaller with the increase of k (Fig. 4).
(3) The runtime of C1 and C2 depends also on the graph structure. For an example, let us consider Fig. 4b (UK 2002). The run times for both C1 and C2 go down for $k = 256$, but then suddenly go up for $k = 512$. The C_k sizes for $k = 256$ and $k = 512$ are 18,179 and 2,951 nodes, respectively. So, to retrieve the same number of communities from a much smaller graph takes longer! To find out why, we conducted a further analysis to explain this surprising result. We conducted multiple tests with different values of k, up to and including $k_{\max}$, which is 943 for UK 2002. We analyzed the nodes that the CCIs were constructed from. The tests show that (a) all CCIs for $k = 256$ are retrieved from the same big ($k = 943$) clique by eliminating just one, least important, node after another. This takes very little time, and the runtime for $k = 256$ becomes quite low as we can observe in the figure; (b) for $k = 512$, there were not enough important nodes in a similar clique, so some CCIs were retrieved from a different cluster with high cohesiveness and relatively high importance. This "switch" to a different area of the graph takes more time than staying steady (due to loss of locality) and the run time of algorithms became somewhat larger. It must be noted however that the effect described above is reflected only on the charts for the smaller graphs, with run times of 1 to 2 seconds. For bigger graphs, the fluctuations in importance (weights) distribution and their interplay with the graph structure does not influence the running time noticeably. This happens because the overhead of switching to a different part of C_k is absorbed by the time needed for the rest of the computation.

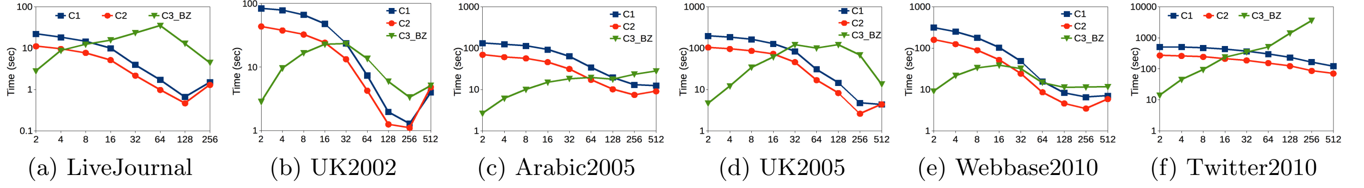
Let us now focus on the performance of the C3_BZ algorithm. We make the following observations: (1) For small r's and k's, C3_BZ is a good choice; this is because we only need to perform few core re-computations (small r) and most of these re-computations are not wasted, i.e. we are able to find new members of k-core quite often when we resurrect vertices (small k). (2) The bigger the graph, the better the chance that using C3_BZ will give better performance for small to moderate r's and k's; this will be significantly more pronounced for Clueweb (presented later in this section).

Finally, let us consider the performance of C1, C2, and C3_BZ as r varies in Fig. 5. We see that the runtime of C1 and C2 is pretty much constant. This happens because most of their time is spent on peeling off the graph until no vertex remains; this is the same regardless of r. The value of r determines the number of MCC runs in C1 and C2; but these MCC runs become negligible in C1 and C2 as they are only performed in the end, after most of the vertices of the graph have been deleted. C3_BZ is sensitive to r. As expected, the greater the value of r, the longer C3_BZ takes to complete.

Problem 2: Computing non-containing communities. Figure 6 shows results for non-containing communities; (a)-(c) show the performance of NC1 for varying k on different datasets, and (d)-(f) show the performance of NC1 for varying r on the same datasets. We see that the runtime of NC1 quickly goes down as k increases, (a)-(c). Also, NC1 continues to not be sensitive to r, (d)-(f). The performance of NC2_BZ (backward approach) was not competitive for these six datasets (not shown in the interest of figure clarity). Nevertheless, for very large graphs, such as Clueweb, as we show later, NC2_BZ is more useful.

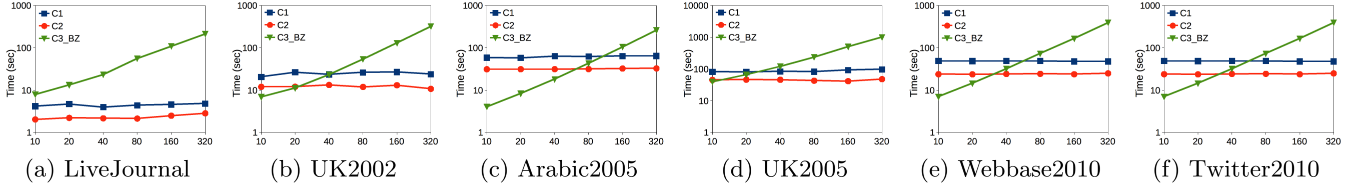
Experiments on Clueweb. A special case for testing the proposed algorithms was using the Clueweb dataset. The results are presented in Fig. 7 through Fig. 9.

Fig. 7 and Fig. 8 show in detail the performance of the algorithms when varying k. It is clear that the backward approaches implemented in C3_BZ and NC2_BZ are to be



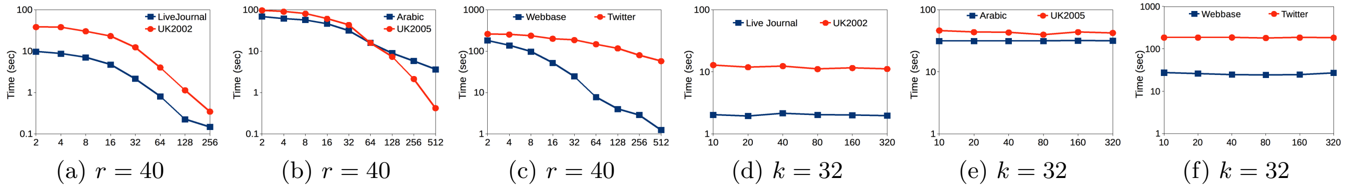
(a) LiveJournal (b) UK2002 (c) Arabic2005 (d) UK2005 (e) Webbase2010 (f) Twitter2010

Figure 4: Containing Communities: Performance when varying k ($r = 40$).



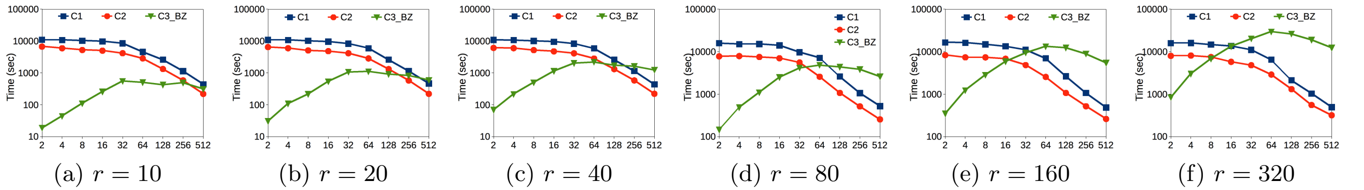
(a) LiveJournal (b) UK2002 (c) Arabic2005 (d) UK2005 (e) Webbase2010 (f) Twitter2010

Figure 5: Containing Communities: Performance when varying r ($k = 32$).



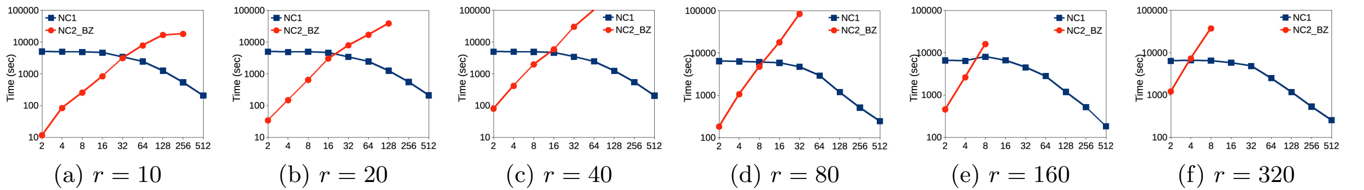
(a) $r = 40$ (b) $r = 40$ (c) $r = 40$ (d) $k = 32$ (e) $k = 32$ (f) $k = 32$

Figure 6: Non-Containing Communities: Performance when varying k and r.



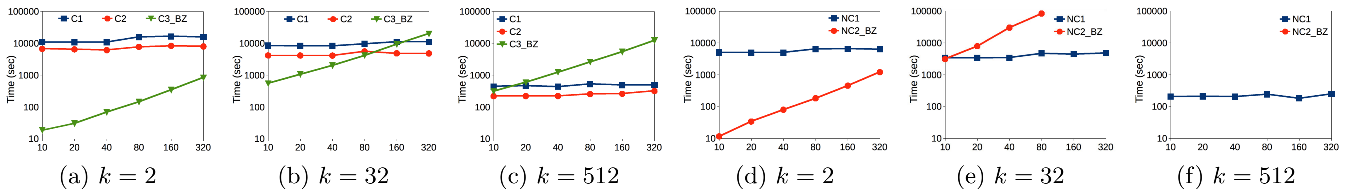
(a) $r = 10$ (b) $r = 20$ (c) $r = 40$ (d) $r = 80$ (e) $r = 160$ (f) $r = 320$

Figure 7: Clueweb: Containing Communities. Performance when varying k.



(a) $r = 10$ (b) $r = 20$ (c) $r = 40$ (d) $r = 80$ (e) $r = 160$ (f) $r = 320$

Figure 8: Clueweb: Non-Containing Communities. Performance when varying k.



(a) $k = 2$ (b) $k = 32$ (c) $k = 512$ (d) $k = 2$ (e) $k = 32$ (f) $k = 512$

Figure 9: Clueweb: Performance when varying r. (a), (b), and (c) - Cont. Communities; (d), (e), and (f) - NC Communities.

used for extracting the top communities for several combinations of k and r (e.g. $k = 2, \ldots, 256$ and $r = 10$ for C3_BZ, or $k = 2, \ldots, 32$ and $r = 10$ for NC2_BZ among others). In several such cases, C3_BZ and NC2_BZ proved to have orders of magnitude better performance than the forward algorithms. Clueweb testing confirmed the trend noticeable before: the bigger the graph, the more beneficial it is to use C3_BZ and NC2_BZ. With k and r increasing, these algorithms keep the better performance on large graphs longer than on the smaller graphs.

On the other hand, if the application is to extract many communities, the forward algorithms are recommended.

In a nutshell, the results show that we are able to compute containing and non-containing communities for every

combination of k and r on Clueweb using the forward algorithms. We can do that faster for a good number of k and r combinations using the backward algorithms. Being able to scale to Clueweb is a significant contribution because this dataset is an order of magnitude bigger than the second large dataset we consider, Twitter 2010, as well as the datasets considered in [23].

7. RELATED WORK

Effectively extracting a set of cohesive subgraphs as communities is an important task in analyzing graphs (cf. [11, 32, 18]). Over the years, community computation has been extensively researched from a theoretical and practical point of view (cf. [20, 14, 36]).

The classic dense subgraph structure is a clique. A large number of works have dealt with extracting cliques according to different requirements; to name a few: [8, 9, 20]. However, the strict clique definition may be too strong for various applications. Several relaxed definitions have been proposed, such as: s-clique [25], s-club [2], k-plex [29], k-core [28], and others. In contrast to most of the other notions, k-core can be computed and maintained in polynomial time and there exists algorithms that scale to large graphs (cf. [7, 19, 35] and [24, 27])

k-core decomposition has been extensively used for detecting communities. For example, k-core decomposition is used by Zhou et. al. [38], Chang et. al. [6], and Akiba et. al. [1] as a foundation for extracting refined communities, such as maximal k-edge connected subgraphs. The k-core-based influential community framework we focus on in this paper is introduced by Li et. al. [23]. Sozio et al. [32] and Cui et. al. [11] compute maximal k-core communities containing given query nodes. A refinement of k-core is k-truss ([34]): the subgraph of k-core in which every edge is supported by at least $k-2$ triangles. Community models based on k-truss are proposed by Huang et al. [17] and [18].

8. CONCLUSIONS

We presented fast forward and backward algorithms for computing top-r, k-core containing and non-containing communities. While the forward algorithms compute communities from the least to the most important, the backward algorithms compute them in the reverse order, from the most important to the least. Our algorithms scale to very large graphs. The largest graph we tested was Clueweb with about 1 billion nodes and 74 billion edges. We were able to compute top-r, k-core communities for every combination of k and r in a wide range of values using the forward algorithms. We could compute top communities faster for a good number of k and r combinations using the backward algorithms. Despite the massive size of the graphs considered, our computations were of small footprint; we produced all the results using relatively inexpensive machines.

As future work, we would like to explore the usefuleness of top-r, k-core communities in trust prediction [21], in clearing a contamination from a network [26, 31], in identifying community formation in biological networks [15], and in devising network-based collaborative filtering algorithms [10, 37]. Also, of interest is extending the notion of top-r, k-core communities to probabilistic graphs [5, 16] and to edge-labeled graphs [13, 30]. Finally, given the enormous size of network graphs today, distributed algorithms in the spirit of [33, 30] need to be devised for discovering top-r, k-core communities.

9. REFERENCES

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

[2] R. D. Alba. A graph-theoretic definition of a sociometric clique. *Journal of Mathematical Sociology*, 3(1):113–126, 1973.

[3] V. Batagelj and M. Zaversnik. An o (m) algorithm for cores decomposition of networks. *arXiv preprint cs/0310049*, 2003.

[4] P. Boldi and S. Vigna. The webgraph framework i: compression techniques. In *WWW*, 2004.

[5] F. Bonchi, F. Gullo, A. Kaltenbrunner, and Y. Volkovich. Core decomposition of uncertain graphs. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1316–1325. ACM, 2014.

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

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

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

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

[10] M. Chowdhury, A. Thomo, and W. W. Wadge. Trust-based infinitesimals for enhanced collaborative filtering. In *COMAD*, 2009.

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

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

[13] G. Grahne and A. Thomo. Algebraic rewritings for optimizing regular path queries. *Theoretical Computer Science*, 296(3):453–471, 2003.

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

[15] T. Gutiérrez-Bunster, U. Stege, A. Thomo, and J. Taylor. How do biological networks differ from social networks?(an experimental study). In *Advances in Social Networks Analysis and Mining (ASONAM), 2014 IEEE/ACM International Conference on*, pages 744–751. IEEE, 2014.

[16] N. Hassanlou, M. Shoaran, and A. Thomo. Probabilistic graph summarization. In *International Conference on Web-Age Information Management*, pages 545–556. Springer, 2013.

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

[18] X. Huang, L. V. Lakshmanan, J. X. Yu, and

H. Cheng. Approximate closest community search in networks. *PVLDB*, 9(4), 2015.

[19] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo. K-core decomposition of large networks on a single pc. *PVLDB*, 9(1):13–23, 2015.

[20] I. Koch. Enumerating all connected maximal common subgraphs in two graphs. *Theoretical Computer Science*, 250(1):1–30, 2001.

[21] N. Korovaiko and A. Thomo. Trust prediction from user-item ratings. *Social Network Analysis and Mining*, 3(3):749–759, 2013.

[22] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[23] R.-H. Li, L. Qin, J. X. Yu, and R. Mao. Influential community search in large networks. *PVLDB*, 8(5):509–520, 2015.

[24] R.-H. Li, J. X. Yu, and R. Mao. Efficient core maintenance in large dynamic graphs. *TKDE*, 26(10):2453–2465, 2014.

[25] R. D. Luce. Connectivity and generalized cliques in sociometric group structure. *Psychometrika*, 15(2):169–190, 1950.

[26] K. Rajagopalan, V. Srinivasan, and A. Thomo. A model for learning the news in social networks. *Annals of Mathematics and Artificial Intelligence*, 73(1-2):125–138, 2015.

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

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

[29] S. B. Seidman and B. L. Foster. A graph-theoretic generalization of the clique concept. *Journal of Mathematical sociology*, 6(1):139–154, 1978.

[30] M. Shoaran and A. Thomo. Fault-tolerant computation of distributed regular path queries. *Theoretical Computer Science*, 410(1):62–77, 2009.

[31] M. Simpson, V. Srinivasan, and A. Thomo. Clearing contamination in large networks. *IEEE Transactions on Knowledge and Data Engineering*, 28(6):1435–1448, 2016.

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

[33] D. C. Stefanescu, A. Thomo, and L. Thomo. Distributed evaluation of generalized path queries. In *Proceedings of the 2005 ACM symposium on Applied computing*, pages 610–616. ACM, 2005.

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

[35] D. Wen, L. Qin, Y. Zhang, X. Lin, and J. X. Yu. I/O efficient core graph decomposition at web scale. *CoRR*, abs/1511.00367, 2015.

[36] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. *Knowl. and Inf. Syst.*, 42(1):181–213, 2015.

[37] N. Yazdanfar and A. Thomo. Link recommender: Collaborative-filtering for recommending urls to twitter users. *Procedia Computer Science*, 19:412–419, 2013.

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