Efficient Algorithms for Discovering Importance-Based Communities in Large Web-Scale Networks

by

Ran Wei
B.Sc., Jiangnan University, 2014
A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science

in the Department of Computer Science

Efficient Algorithms for Discovering Importance-Based Communities in Large Web-Scale Networks

by

Ran Wei
B.Sc., Jiangnan University, 2014

Supervisory Committee

Dr. Alex Thomo, Supervisor
(Department of Computer Science)

Dr. Venkatesh Srinivasan, Departmental Member
(Department of Computer Science)

Supervisory Committee

Dr. Alex Thomo, Supervisor
(Department of Computer Science)

Dr. Venkatesh Srinivasan, Departmental Member
(Department of Computer Science)

ABSTRACT

k-core is a notion capturing the cohesiveness of a subgraph in a social network graph. Most of current research works only consider pure network graphs and neglect an important property of the nodes: influence. Li, Qin, Yu, and Mao [PVLDB'15] introduced a novel community model called "k-influential community" which is based on the concept of k-core enhanced with node influence values. In this model, we are interested not only in subgraphs that are well-connected, but also have a high lower-bound on their influence. More precisely, we are interested in finding top r (with respect to influence), k-core communities. We present novel approaches that provide an impressive scalability in solving the problem for graphs of billions of edges using only a consumer-grade machine.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank:

Supervisor Alex Thomo, for his mentoring and support in the past three years as well as his patience and help in guiding me on the research.

My teammates, Shu Chen and Diana Popova, for their effort on the research and help when I met challenges.

My friends for their company, we have so much unforgettable moment together.

My parents for always being there, love me and support me to pursue my dream.

When we love, we always strive to become better than we are. When we strive to become better than we are, everything around us becomes better too.
Paulo Coelho

DEDICATION

To my family, and friends.

Chapter 1

Introduction

In the last decade, community search has been a very popular topic in research and industry because of the many applications that are based on it. In general, people identify those communities by searching for cohesive subgraphs in a graph [16, 22, 26, 40]. A cohesive subgraph means a set of nodes that are well connected with each other. At present, there are several metrics to measure the cohesiveness of a graph, such as k-core, n-cliques, k-trusses, s-plexes, etc. Among these, k-core has its advantages over the other metrics.

k-core of a graph G is defined as the largest induced subgraph of G in which every vertex has degree of at least k. From the perspective of theory, k-core is very popular because it can be used as a subroutine for harder problems, like computing the size of cliques and giving an approximation for the densest subgraph problem[31]. Also, k-core has many applications in industry. It is a powerful method for measuring network structure [49], investigating teamwork within software teams [44], finding structural diversity in social contagion [43], and analyzing complex networks in terms of node hierarchies, self-similarity, and connectivity [4].

Computing k-core can be done in polynomial time; in contrast, computing k-truss is quadratic, whereas n-clique and s-plex are NP-hard. As a result, k-core is a widely used concept in community search for large graphs.

However, computing only densely connected subgraphs ignores an important feature of real-world networks. That is, the community usually has an attribute named influence (also known as importance). For example, we may want to identify the most influential research groups (with dense co-authorship links) in terms of their citations. Another example is in a social network, where we may want to find densely connected communities which have great influence as measured by the quality of their

contributions to some topic.

In many applications, we are interested in finding densely connected and important communities. Fig. 1.1 displays a social network with node influence/importance values. We can see that the nodes are drawn in different sizes. The larger the node, the more important it is. Visually, we can see that bigger nodes are organized into cohesively connected groups (communities). However, there exist smaller nodes that can be cohesively (densely) connected to these groups of bigger (more important) nodes. For an example from co-authorship networks, these smaller nodes can be researchers who are co-authors with prolific authors, but who themselves have a lower stature in the research community with respect to their influence or importance. In our research, we want to get rid of those noise (smaller) nodes to find the most important communities.

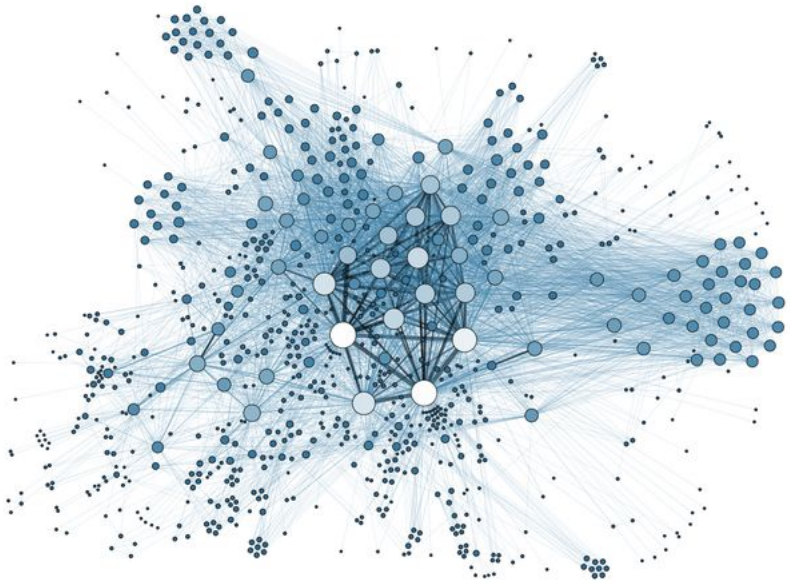


Figure 1.1: Social Network Visualization

In order to find such of communities, Li, Qin, Yu, and Mao [32] proposed a new community model called "k-influential community" based on the concept of k-core. Assume there is an undirected graph $G = (V, E)$. Differently from the traditional k-core, in this model, each node has a weight, which represents the influence or

importance that it has. A k-influential community is a connected, induced subgraph where each node has degree at least k, and where each node has an importance value above some lower bound. On one hand, we use k-core to measure the cohesiveness of the community, on the other hand, we use the node weights to judge the influence of the community. The influential value of a community is defined as the minimum weight of nodes in that community. More specifically, we are interested in finding the top-r (with respect to influence) k-influential communities in a network.

1.1 Motivation

Finding communities in a network is typically hard. Straightforward search for the top-r k-influential communities in a large network is impractical as there may be a huge number of communities that satisfy the degree constraint and so it is hard to find out the top-r communities from them. In spite of this, there are several algorithms that have been used for top-r, k-influential community discovery with varying levels of performance and space requirements.

There are two algorithms presented in [32]. The first one is a straightforward linear time algorithm with respect to the graph size. The idea is to iteratively remove the smallest-weight node from the maximal k-core and then identify communities by running a maximally connected component (MCC) algorithm. The advantage of this method is that it is easy to understand and implement. However, this algorithm only works for moderate size graphs. When the graph is big, the algorithm does not fit our needs because it needs to run the MCC algorithm too many times and that consumes too much time. Another approach is an index-based algorithm which builds a community index that can be queried for communities for any k and r. The advantage of this algorithm is that it is very efficient once the index is built. However to build the index is expensive. Because it is a main-memory-based algorithm, it needs space that is comparable to the (uncompressed) graph size to store the index. When the dataset is big, we are not able to fit the index into main memory.

In [10], we introduce algorithms that speed up the computation of communities by using a consumer grade machine. The algorithms are "forward" in the sense they produce the communities from the least to the more important ones. These forward algorithms recursively peel off the smallest weight node from communities discovered so far. While these algorithms are fast for moderate graphs, because of their forward nature, they do not scale well to very large graphs. The main reason is that there is

a lot of wasted computation for discovering communities of low importance that no one is really interested in.

The goal of this thesis is to come up with a new approach that works in a reverse, backward, way. That is, it discovers the top r communities from the most important to less important ones. This way, we can significantly reduce the computation time of peeling off a huge number of nodes starting from the smallest weight. As a result, when the graph is huge and r relatively small, this algorithm should have the best performance as it only visits a small part of the graph. Also, we introduce efficient data structures, so that we can scale the computation of influential communities to massive graphs with billions of edges. Furthermore, we aim to run the computation using only a consumer grade machine.

1.2 Contributions

There have been many articles related to k-core decomposition. However, there is not too much research related to influential communities for now. In the context of computing importance-based communities in web-scale networks, we make the following contributions.

1. We present backward algorithms to compute top-r k-influential communities efficiently. When the graph is big and r relatively small, these algorithms produce the result by only accessing a small portion of the edges;

2. We modified a data structure, based on the prior research, and now our algorithm is able to compute influential communities of massive graphs with billions of edges using only a single consumer-grade machine.

3. We conduct extensive experiments over several large and very large graphs. The smallest graph is Livejournal. The biggest graph is ClueWeb with about 1 billion nodes and 74 billion edges. The results prove that we are able to compute top-r k-core influential communities for varying combinations of k and r in a large range of values using the backward algorithm. Furthermore, for some specific k and r, using backward algorithm is the fastest algorithm among all of the algorithms presented so far.

1.3 Agenda

Now we give a brief introduction to the topics of this thesis. The main purpose of this thesis is to demonstrate our new approaches to readers and display the extensive experimental result we got. A map of the thesis is as follows:

Chapter 1 introduces the interest of this research topic. It also gives a simple explanation about some concepts and our research.

Chapter 2 states in detail the problem we solve. Additionally, we offer a lot of information about the model we used. At last, we introduce an efficient approach for the computation of k-core, coming with the pseudocode of it and an example.

Chapter 3 explains our new algorithms in detail. There is also a section that describes the data structures we use to scale the computation to process massive graph.

Chapter 4 shows the source of graphs we used in this thesis for experiments. Also it describes the compression technique we used for pre-processing the graph in order to fit it in limited memory.

Chapter 5 presents the extensive experimental results we got. Based on this results, we get a better understanding of the advantages and disadvantages of our algorithms.

Chapter 6 concludes the thesis and makes a summary.

Chapter 7 gives some related work in the area of community search and cohesive subgraph mining.

Chapter 2

Background

In this chapter, we present the problem statement. We give some definitions to make the statements clear in section one. Also, we use a simple example to illustrate how to compute k-core influential communities in section two. In section three we introduce an efficient algorithm for k-core decomposition.

2.1 Basic Definitions

We use undirected graphs to represent networks. Let $G = (V, E)$ be an undirected graph. Correspondingly, V is the set of vertices, and E is the set of edges. In order to illustrate the problem easier, we set n to be the number of vertices and m to be the number of edges.

Let u be a vertex in G, we denote the degree of u by $d_G(u)$, and the set of its neighbors, say $\{v : (u, v) \in E\}$, by $N_G(v)$. We set $d_{\max}(G) = \max\{d_G(u) : u \in V\}$.

Consider $K \subseteq V$ is a subset of vertices of a graph $G = (V, E)$. We have the following definitions.

Definition 1. *Graph $G(K) = (K, E_K)$, where $E_K = \{(u, v) \in E : u, v \in K\}$ is called* the subgraph of G induced by K.

As we said in the previous section, the influential community notion is based on k-core of the graph.

Definition 2. *A k-core of a graph G is an induced subgraph H_k of G, such that, each vertex in H_k has a degree at least k in H_k.*

Similarly, the definition of maximal k-core is:

Definition 3. *The maximal k-core of a graph G is the largest induced k-core subgraph. We denote the maximal k-core of G by $C_k(G)$.*

On the other hand, we have a definition about how to represent the core number for each node:

Definition 4. *The coreness (core number) of a node u is the largest k such that $u \in H_k$ and $u \notin H_{k+1}$*

In general, $C_k(G)$ does not have to be connected, which means it may contain several maximally connected components (MCC's). Also, the maximal k-core of a graph is unique.

2.2 Influential Communities Model

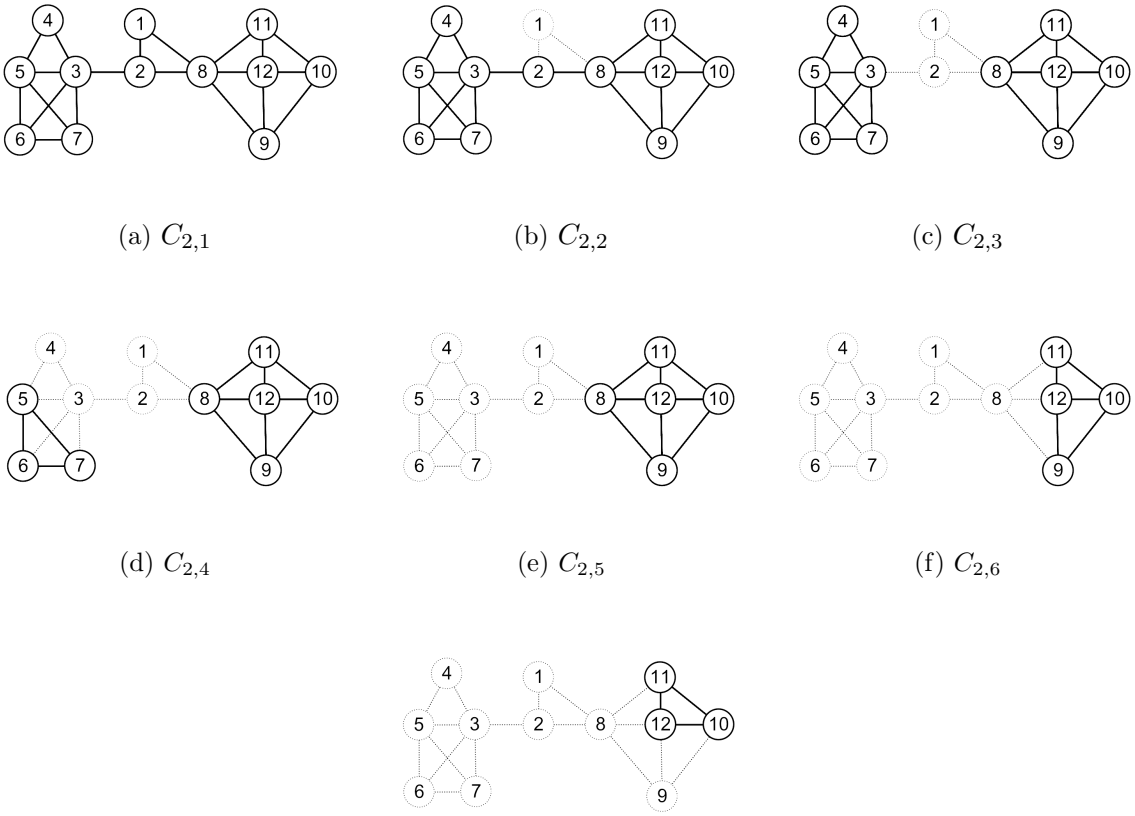
For a start, we would like to recall that the influence value of a community (subgraph) is defined as the smallest weight of the nodes in the community [32]. We choose to use the minimum weight value instead of the average in order to have a guarantee that all the nodes in the community have a weight of a certain level. According to this, if an induced subgraph has a large influence value, it means that each node in it should have large weight. Node weights can stand for different kinds of importance values in practice; such a value can be computed using PageRank, or it can be income, age or salary information of a user in the social network. Here, we use an array w of size n to store the weight value for each node, where n is the number of nodes. Then $w[i]$ is the weight of node i.

Based on these, the influential community search problem is to find the top-r, with respect to community influence, maximally connected k-core subgraphs of $C_k(G)$, regarding different k and r. By maximally connected subgraph, we mean that for a k-influential community, say H_k, there should not be a second, induced subgraph, H'_k, that contains H_k and has the same influence value. Because if it happens, H'_k will overshadow H_k in both size and influence value.

The most straightforward method to compute k-influential communities is introduced as follows. Suppose we have calculated the maximal k-core $C_k(G)$ and set $C_{k,1} = C_k(G)$. Let $v_{k,1}$ be the vertex with minimum weight in $C_{k,1}$. Then we define the maximally connected component (MCC) that contains $v_{k,1}$ as the k-influential community with the smallest influence value. We set it as $H_{k,1}$. After this, we remove $v_{k,1}$ and its neighbors whose degree will become smaller than k in $C_{k,1}$ due to

the deletion of $v_{k,1}$. Then we say that the remains of $C_{k,1}$ after the vertices deletion is $C_{k,2}$, which is also a k-core. As we did before, we repeat the previous process to find the smallest weight vertex and its MCC to obtain the second influential community $H_{k,2}$.

In general, we keep "peeling off" the minimum weight vertex $v_{k,i}$ in $C_{k,i}$ until the latter becomes empty. As the MCC of $C_{k,i}$ that contains $v_{k,i}$, $H_{k,i}$ is the i-th influential community. In the end, i is the total number of k-influential communities in this network, and the last r of them are the top r, k-influential communities we want.



(a) $C_{2,1}$ (b) $C_{2,2}$ (c) $C_{2,3}$

(d) $C_{2,4}$ (e) $C_{2,5}$ (f) $C_{2,6}$

(g) $C_{2,7}$

Figure 2.1: Cohesive induced subgraphs for $k = 2$

Example 1 Consider the graph in Fig. 2.1. In order to simplify the example, we set the weight of each vertex to be equal to the vertex id. Suppose we set $k = 2$. The grayed out vertices and edges mean they are deleted during the peeling off process.

When we delete vertex 3 in $C_{2,4}$, vertex 4 is gone as well because its degree becomes 1 after vertex 3 is deleted. The same thing happens when we delete vertex 5 in $C_{2,5}$; both vertex 6 and vertex 7 are deleted recursively. $H_{k,i}$ is the MCC of $C_{k,i}$ containing the vertex of the smallest weight $v_{k,i}$. For example, for $C_{2,6}$ (see Fig. 2.1f), $v_{2,6} = 9$ and $H_{2,6} = \{9, 10, 11, 12\}$.

In the above example, we can see that a k-influential community can be contained in another bigger k-influential community. For example, we can find two 2-influential communities in Fig. 2.1: {8, 9, 10, 11, 12} and {9, 10, 11, 12}. The first community contains the second one. This kind of situation may be undesirable depending on the application. In order to get rid of it, the concept of "non-containing influential communities" is defined. Given a graph $G = (V, E)$ and an integer k, a *non-containing* k-influential community $H_{k,i}$ cannot contain another k-influential community $H'_{k,i}$ that has a higher influence value than $H_{k,i}$.

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

The above process of computing the top-r, k-influential communities is a "forward" process in the sense that it computes the communities in the order from the least important to the most. As such, it "wastes" a lot of cycles to compute communities of low value. On the other hand, a peel-off is not very expensive, so still, a forward approach, as described the above, can work well in practice if the graph is not very big. Our main contribution is backward computation of communities, which produces the communities in the order from the most important to the least. Computing the very top communities first is fast. As we go down and compute less important communities, the computation becomes more expensive. So, the question is: for what values of r and k, and size of the graph, one should choose a backward approach over a forward approach?

2.3 Basic idea of the BZ Algorithm for core decomposition

k-core decomposition is computed by the Batagelj and Zaversnik (BZ) algorithm [6]. We implemented it using the WebGraph API with random access [7]. Before we

introduce our new method, here we will demonstrate the method of efficient k-core decomposition and its importance.

The BZ algorithm is an efficient, $O(m)$ algorithm for determining the coreness of each vertex in a given graph, where m is the number of edges. It computes the cores hierarchy by recursively deleting the vertex with the lowest degree. The deletion is not a physical deletion on the real graph. Instead, a binary array is used to track the status of each vertex logically. The outline is given in Algorithm 1

Algorithm 1 basic idea of k-core decomposition

```
1:  function K-CORE(GRAPH G)
2:      L ← 0, d ← 0, D ← [∅, ...,∅]
3:      for all i ← 0 to n do
4:          d[i] ← d_G[i]
5:          D[d[i]].insert(i)
6:      for all k ← 0 to d_max(G) do
7:          while not D[k].empty() do
8:              i ← D[k].remove()
9:              L[i] ← k
10:             for all j ∈ N_G(i) do
11:                 if d[j] > k then
12:                     D[d[j]].remove(j)
13:                     D[d[j] − 1].insert(j)
14:                     d[j] − −
15:     Return L
```

In total there are three arrays in Algorithm 1. L records the coreness of each vertex in G. d holds the degree of each vertex. At last, D stores different degree values and the set of vertices with that possible degree value.

From Line 2 to Line 5 we initialize the arrays. In the rest of the function, we implement the idea. For example, suppose the vertex with the smallest degree is i and its core value is k. Then k is recorded in L, and i is stored in the first non-empty set of $D[k]$. During the computation, when we delete i, we go through of its neighbors and decrease their degrees by one. So the neighbour j will be moved from $D[d[j]]$ to $D[d[j] - 1]$. In the end, the algorithm returns an array L which stores the coreness of each vertex.

The real challenge in Algorithm 1 is how to implement sets D. The straightforward idea is using a hash set for each $D[k]$. As we know, there are several ways to implement hash set. The most popular hash maps have *expected constant time* (ECT) for lookups

and deletions. However, as [27] found out in practice, we find that ECT is not good enough to deal with large graphs.

In this work, we use the implementation of the BZ algorithm by [27], which used flat array structures to implement the sets D.

Now we know how to compute core number effectively by using BZ algorithm. In the next two chapters, we will demonstrate how to search influential communities with the help of BZ algorithm.

Chapter 3

Backward Algorithm

The key problem is how to calculate the communities effectively. To the best of our knowledge, all the existing algorithms calculate the communities in a forward way, which is starting from the whole graph and doing iterations of getting rid of vertices that do not satisfy our requirements (e.g. weight is small and degree falls below k). In the end, we can get top-r communities that we are looking for. It is a good method when r is relatively large, considering the size of the graph. However, when r is relatively small, like if we want to know the top-20 communities, the forward approach wastes a lot of computation. With the forward algorithms we have to peel off the vertices from the smallest weight and finally get the last 20 communities from it after tons of iterations.

What's more, as we mentioned before, the time complexity of computing the maximal connected component is $O(m)$. As a result, we take too much time in the original algorithm to compute the MCC for each C_k. The prior C_k in the forward approaches is big in size so the cost of computing MCC is considerable. We even observed that in practice for those initial iterations, most of them only delete few nodes. Which means we need to do tons of iterations to peel off all the unnecessary nodes and get the final top r communities when r is relatively small.

We pose the following question: How can we avoid the early peel-off stage and access the vertices in the top r communities as soon as possible?

In this section, we propose a new approach. Differently from a forward approach, instead of peeling off the lowest vertice in each iteration, our algorithm begins in a state where all the vertices are assumed to be "gone" as if they are deleted, and then we "resurrect" the highest-weight vertex in each iteration and check if it can form a k-core community with other vertices that were "resurrected" before.

3.1 Backward Algorithm for P1

Here we give our backward algorithm C3, for calculating top-r k-influential communities in Alg. 2.[1]

This algorithm begins with considering all the vertices being deleted and their core numbers equal to zero (Line 2, 3). We get the vertex with most important weight among the "dead" vertices in each iteration, say v, and make it "alive," then calculate the core values of all the "alive" vertices so far to check whether they need to be updated because of the new "alive" vertex (known as updateCores(v) in the Alg. 1, Line 8). If v happens to have a core number that is greater or equal to k, it proves v is a vertex of the $v_{k,i}$ community. Based on a classical Maximally Connected Algorithm (MCC) starting from v and only using the alive vertices that have a core value greater than or equal to that of v, we can get the community we seek.

Algorithm 2 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
```

In order to show the soundness and completeness of Algorithm 2, we first present

[1] The algorithm is called C3 in order to be compatible with our publication [10]. In the publication, we call C0 the original online algorithm of [32], and C1 and C2 two forward algorithms for P1. The forward algorithms are not discussed here. Detailed descriptions of C1 and C2 can be found in Shu Chen's MSc thesis.

the following lemmas.

Lemma 1. *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 the 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.

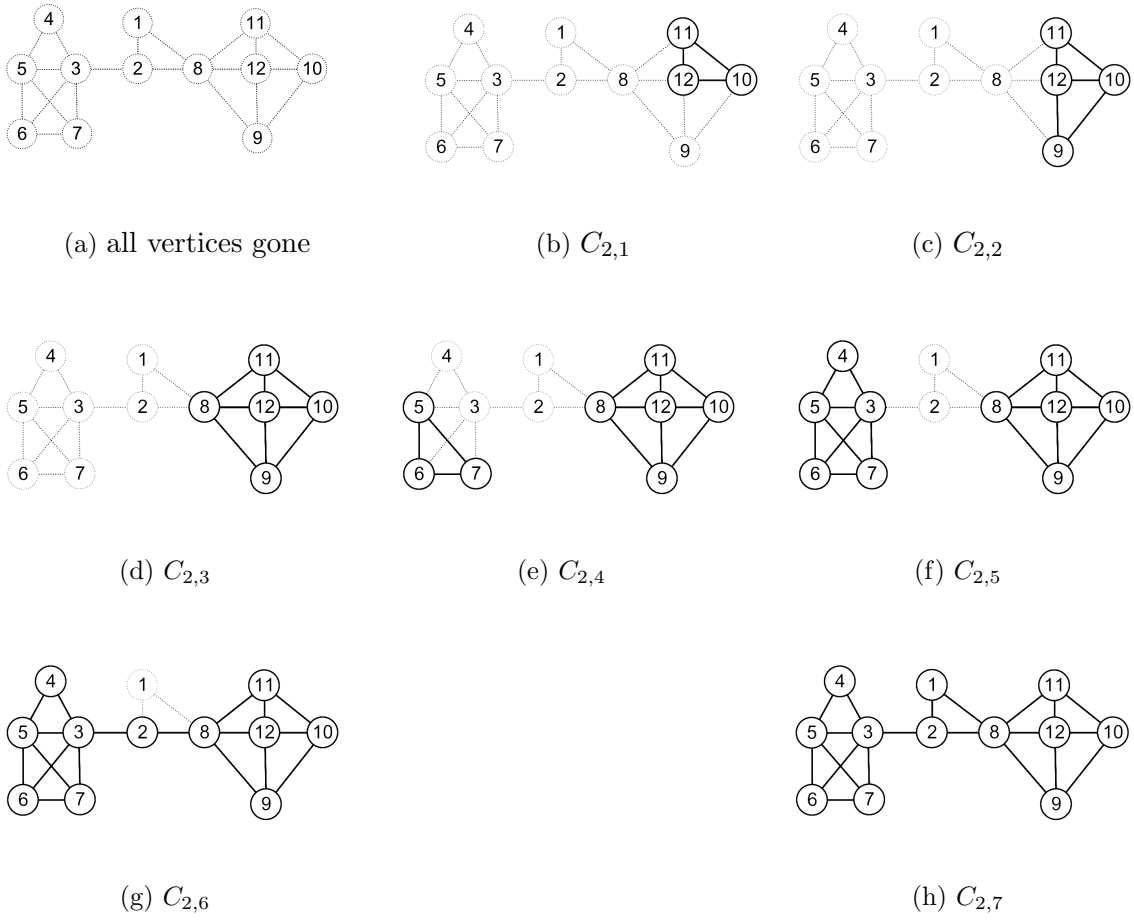
Lemma 2. *Let $i \in [1, p]$. There exists a vertex v, such that $v = v_{k,i}$, and v is in a k-core, which includes v and all the other vertices u with $w[u] > w[v]$.*

Based on lemmas 1 and 2 and the fact that we only produce the top-r results, we can state the following theorem.

Theorem 1. *Algorithm 2 correctly computes all and only the top-r CCI communities.*

Compared to forward algorithms, the advantage of this backward algorithm is that we can find out the top-r communities much faster when r is relatively small. The reason is that the backward algorithm does not need to perform the peel-off computation for most of the low weight vertices. When r is small, a few of high weight vertices are enough to generate all the communities we want. Especially for a large graph, this benefit becomes very obvious. Of course, as r grows, for the same graph the computation time of the backward approach will finally equal to the forward one and then become slower than it. We will present more experimental results to prove that in the next chapter.

Here we still use the graph in Chapter 2 as an example. Previously we introduced how to search influential communities using a forward approach. Now we will illustrate the new backward algorithm to do it. Suppose the weight of each vertex equals to its index and we are looking for communities with $k = 2$. First of all, all the



(a) all vertices gone (b) $C_{2,1}$ (c) $C_{2,2}$

(d) $C_{2,3}$ (e) $C_{2,4}$ (f) $C_{2,5}$

(g) $C_{2,6}$ (h) $C_{2,7}$

Figure 3.1: Backward approach searching cohesive induced subgraphs for $k = 2$

vertices are logically deleted (Fig 3.1a). Vertex 12 is resurrected first because it has the largest weight among all the vertices. Currently, there is only one "alive" vertex so of course we cannot find influential communities. Then we keep doing iterations to resurrect node 11 and node 10. Now node 10 has a coreness that equals 2 which satisfies the IF condition in line 9 of Alg 2. By running the MCC function, {10, 11, 12} is the first community we find using the backward approach. Next, we make node 9 alive and the core number of it is 2 as well, so we execute MCC with node 9 and get a new influential community {9, 10, 11, 12}. Also, after resurrecting node 8, it is included in a new community {8, 9, 10, 11, 12} with the former vertices. In the next iteration, we jump to another vertex that is not directly connected with the previous vertices, that is, node 7, so we continue trying to build another connected component that includes node 7. We keep doing this process until we find a desired number of

communities or all the vertices are "alive" again.

The time complexity of C3 is a little tricky. Although it calls *updateCores* in each iteration, there is a degree condition within it to constraint the update computation be executed only if the resurrected vertex is well connected to the other resurrected vertices. In practice, this mechanism helps reduce the number of update computation significantly. As a result, C3 could be much faster than the forward algorithm for a moderate r.[2] The space complexity of C3 is $O(m)$ because the graph takes $O(m)$ space. In fact, the real space needed is quite manageable because we use the WebGraph framework to reduce the graph size significantly.

3.2 Backward Algorithm for P2

For non-containing communities, we can also construct a backward approach. In [32], they came up with a lemma to identify the non-containing communities easily. To put it simply, for a k-influential community $H_{k,i}$, if we delete all the vertices of it in the next iteration, then we say that $H_{k,i}$ is a non-containing k-influential community.

For the backward approach, we can take advantage of the same idea but have to implement it in a different way. This is because instead of deleting vertices in each iteration, we actually bring dead vertices back, so 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 community.

Our backward algorithm, NC2, is given in Alg. 3.[3] In order to handle massive graphs in a consumer level machine, we need to make the algorithm as efficient as possible. Here 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. 4).

[2] In the experiment chapter we will give more accurate numbers to define the meaning of "moderate".

[3] The algorithm is called NC2 in order to be compatible with our publication [10]. In the publication, we call NC1 the forward algorithm for P2. The forward algorithms are not discussed here. Detailed descriptions of NC1 can be found in Shu Chen's MSc thesis.

Algorithm 3 Top-r non-containing communities (NC2)

Input: G, w, k, r
Output: Top-r non-containing $H_{k,\tau_{j_{\max}-r+1}}$, ..., $H_{k,\tau_{j_{\max}}}$

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

Algorithm 4 MCC with *alive* and *inPC* arrays

```
1: procedure MCC(G, v, alive, inPC, isNC)
2:     cc ← ∅
3:     MCC-DFS(G, v, alive, cc, inPC, isNC)
4:     return cc
5:
6: procedure MCC-DFS(G, v, alive, cc, inPC, isNC)
7:     cc.add(v)
8:     if inPC[v] = true then
9:         isNC ← false
10:    else
11:        inPC[v] ← true
12:    for all u ∈ N_G(v) do
13:        if cores[u] ≥ k & u ∉ cc then
14:            MCC-DFS(G, u, alive, cc, inPC, isNC)
```

3.3 Core Update Algorithms

The *updateCores* procedure needed by algorithms 2 and 3 comes with its own set of challenges. We have two options: either use an incremental core update algorithm,

such as the one proposed in [33] or recompute the cores using the Batagelj and Zaversnik (BZ) algorithm [6]. We implemented both and compared them. The incremental core update of [33] 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.

3.3.1 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, the BZ algorithm 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. As the notion of deletion in BZ algorithm is very different from the deletion we came up with in the backward algorithm, we have to do the computation in a different way.

First of all, all the vertices are indexed from 0, and there are several arrays that need to be initialized before start 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.

As we said before, maybe BZ algorithm is a perfect choice for k-core decomposition of static graph. However, the graph is keep changing with the new vertex get "resurrected" in the backward algorithm. In order to implement BZ algorithm here I have to make some adjustments:

- *al* is a global array with a dimension of n. Array *al* stores all 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*. For example, $al\_idx[v]$ is the position of v in *al*.

In line 2 of Alg. 5, 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.

ModBZ is invoked by *updateCores* (see Alg. 6). The latter starts by recording the resurrected vertex v in *al* and updating *al_idx* and *n_alive*. Next, it calls *updateDegrees* to update the degrees (in array *degrees*) of alive vertices affected by v's resurrection. *updateDegrees* also updates the value of the maximal degree of alive vertices, *md_alive*, on the fly, so that it is ready for *ModBZ* to use. If the current

degree of v (in the subgraph induced by the alive vertices) happens to be greater or equal to k, we call *ModBZ*.

Algorithm 5 Modified BZ algorithm (ModBZ)

```
1: procedure ModBZ(G)
2:     initialize(vert, pos, bin, 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]−−
17:
18: procedure initialize(vert, pos, bin, G)
19:     for all v ← 1 to n_alive do
20:         cores[al[v]] = degrees[al[v]], bin[cores[al[v]]]++
21:     start ← 0
22:     for all d ← 0 to md_alive do
23:         num ← bin[d], bin[d] ← start
24:         start ← start + num
25:     for all v ← 0 to n_alive do
26:         pos[v] ← bin[cores[al[v]]]
27:         vert[pos[v]] ← v
28:         bin[cores[al[v]]]++
29:     for all d ← md_alive downto 1 do
30:         bin[d] ← bin[d − 1]
31:     bin[0] ← 0
```

Algorithm 6 Core update using *ModBZ*

```
1: procedure updateCores(G, v)
2:     al[n_alive] ← v, al_idx[v] = n_alive, n_alive++
3:     updateDegrees(v)
4:     if degrees[v] ≥ k then
5:         ModBZ()
6:
7: procedure updateDegrees(G, v)
8:     for all alive u ∈ N_G(v) do
9:         degrees[v]++, degrees[u]++
10:        md_alive ← max{md_alive, degrees[v], degrees[u]}
```

We experimented extensively with the core update algorithm of [25] and the modified BZ algorithm described above. We found that recomputing the core numbers by the modified BZ algorithm is faster. The reason is that the core update algorithm of [25] updates cores after the insertion of an edge, so, in our case, upon the resurrection of a vertex, there are many edges inserted, and the algorithm is triggered many times.

3.4 Flat Arrays Implementation

In the last section, we introduced a Modified BZ algorithm to compute the k-core update. In that implementation, in order to achieve high performance, everything (e.g. vertices deletion and neighbors update) is implemented by flat arrays rather than hashmap. As argued in previous research [27], if we use hash-based structures, no matter which JAVA library it comes from, we cannot have a good enough performance for massive graphs. The reason is because hashmaps have *expected* constant time (ECT) for lookup and deletions, which is not good enough when processing very big graphs.

We came across a similar problem in our research. When we did experiments with the forward algorithm, we found we had a memory problem on the massive graph Clueweb. However, this is beyond our expectation because after compressing this massive graph with Webgraph, we were able to load it into memory and use the Webgraph random access API. The reason is we used inefficient data structures in the first version of the forward algorithm. For example, in the beginning, we created

a custom object named NodeInfluence to store the node information. It contained the node index and its influence value.

Although a custom object of the Java queue type works well for small to medium size graph, it turned out not to be a good choice for massive graphs. The reason is the memory manager has problem with allocating billions of custom object NodeInfluence. It crashes by saying not enough memory even though the memory space is much larger than the graph after storing it with Webgraph.

In contrast to the above methods, our solution is to use a simple but quite effective data structure, flat array. We initialize a flat array of nodes sorted by their weight, *rank*, to replace the priority queue. Now we can keep the order of nodes instead of storing the weight of each node so that we can get rid of NodeInfluence (e.g. $rank[0]$ stores the node index with the most important weight). Similarly, we replace ArrayDeque I by a flat array as well. Also we create a new array called *Ioffsets* to record the index boundaries of the vertices deleted in the same iteration of I. For example, vertices deleted in iteration 0 are stored in $I[0], \ldots, I[Ioffsets[0] - 1]$, and vertices deleted in iteration 1 are stored in $I[Ioffsets[0]], \ldots, I[Ioffsets[1] - 1]$, and so on.

For example, see Fig. 2.1 and Table 3.1. In Fig. 2.1, we show the process of peeling of nodes and get 2-influential communities. In the first iteration, we peeled off node 1, so we store 1 at array $I[0]$, along with that, as there is only one node is deleted in this iteration, so $Ioffsets[0] = 1$. In the second iteration, we only deleted node 2, so $I[1] = 2$. Now the size of I is 2 so $Ioffsets[1] = 2$. Similar, in the third iteration we deleted two nodes 3 and 4, so we added them into I. Currently, the boundary of I comes to 4, which means the next cell in *Ioffsets* is $Ioffsets[2] = 4$.

Now the memory manager will have a much easier work to allocate big chunks for the arrays rather than many small objects in a queue. Now even though the space complexity is still $O(m)$ (same with the priority queue), the memory is allocated in a one bigger chunk and the memory manager will be happy. We compared the performance before and after this structure modification, and now we can load massive graph into memory to push our research to a bigger scale.

index	rank	I	Ioffsets
0	1	1	1
1	2	2	2
2	3	3	4
3	4	4	7
4	5	5	8
5	6	6	9
6	7	7	12
7	8	8	
8	9	9	
9	10	10	
10	11	11	
11	12	12	

Table 3.1: Flat arrays rank, I, and Ioffsets in the end after the peeling off nodes process of the graph in Fig. 2.1 is done. $I[0]$ stores the node that is deleted in the first iteration, $I[1]$ stores the node that is deleted in the second iteration, $I[Ioffsets[1]], \ldots, I[Ioffsets[2]-1]$ store the nodes that are deleted in the third iteration, and so on.

Chapter 4

Graph Information and Preprocessing

In this chapter, we will introduce the network graphs we used for experiments. Additionally, we will present a graph compression framework, WebGraph [7], which can reduce the size of massive graphs so that they can fit in main memory.

4.1 Graph Information

We use eight web-scale network graphs for our research (see Table 4.1). They can be downloaded from `http://law.di.unimi.it/datasets.php`. From the size perspective, they can be divided into four sub-groups. AstroPhysics and LiveJournal can be considered as medium size graph. UK2002, Arabic2005 and UK2005 are large graphs. Webbase2010 and Twitter2010 are very-large. In the end, ClueWeb is a really massive dataset. We use it in order to test the improvement after we modified the data structure to solve the memory issue. According to the requirements of our research, all of the graphs have to be undirected, that is in graph G, for each edge(u, v), there must be an edge (v, u) correspondingly. Additionally, self-loop edges (for example, (v, v)) are not included in our experiments. The numbers showed in Table 4.1 match the data after the above cleaning.

As the original data does not have the concept of weight, we also create our own weight array. We tried to use PageRank to generate weights for the nodes, however, we found the resulting communities were highly correlated with the PageRank value. As such, in order to make our result more objective, we decided to write a program

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	955.2 M	37,372 M	75,611,696	4,244

Table 4.1: Datasets ordered by m. The two last columns give the maximum degree and maximum core number, respectively.

to assign random weights to the vertices of each graph. The idea is to initialize an array *rank* of size n, where $rank[i]$, for $i \geq 0$, stores the index of $i+1$-th important vertex (e.g. rank[0] stores the id of the most important vertex). Then we shuffle the array randomly, and after that, we write the value of shuffled array (vertices index) to a file ordered by the weight. In order to speed up the reading process, we convert the weight file from text format to binary format.

4.2 WebGraph

One of our main goals is running computation on a single machine, which means we need to keep the graph footprint as small as possible. At first, we use a simple text file to store the graph edge pairs. Then we realize that it is impossible to fit large graph into memory. For example, the edge pair file for Arabic is 21.5GB. However, a single consumer-level machine only has a memory of about 8GB to 12GB in general. Let alone we have more graphs larger than Arabic. In order to achieve our goal, we take the advantage of Webgraph, a highly efficient and well-maintained graph compression framework to preprocess the data. Other works that efficiently use Webgraph are [27, 39, 42].

Webgraph is able to compress the graph so that the compressed file can be an order of magnitude lower than the original one. For example, the file for Arabic is only about 150MB after compression. Additionally, Webgraph provides a series of API which helps us make life easier. *load*() will load the graph from disk to memory. For example, *numNodes*() is a method that returns the number of nodes of the graph.

Chapter 5

Experiments

We did many experiments on the graphs we introduce in Chapter 4 using our new algorithms to check the performance. At first, in order to implement the backward algorithm, we evaluated the performance of both updateCore and modified BZ algorithm and got rid of the approach that is significantly slower than another. Furthermore, we kept running extensive experiments on the backward algorithm against several graphs to compare its performance with forward algorithms. Especially, we did extensive experiments on ClueWeb, a massive graph, with forward and backward algorithm to see the scalability after we applied flat arrays to overcome memory issues. We implemented algorithms for computing top r, k-influential containing communities and top r, k-influential non-containing communities respectively. The values of r and k we tried are displayed in Table 5.1 All algorithms are implemented in JAVA. Except for the experiments on ClueWeb, other experiments are conducted on a MacBook Pro running OS X Yosemite with a 3.4GHz Intel Core i7 (4-core) processor, 16GB RAM. For ClueWeb, we choose a machine with larger memory because the compressed graph by WebGraph framework is about 20GB. CLueWeb was tested on a machine running Ubuntu Server 14.04.3 LTS with 2.10 GHz Intel(R) Xeon(R) E5-2620 v2 (6-core) CPU and 64GB RAM. For your information, both machines belong to a consumer grade because their prices are about $3K.

Param.	**Range**
k	2, 4, 8, 16, 32, 64, 128, 256, 512
r	10, 20, 40, 80, 160, 320

Table 5.1: Parameters k and r, and their ranges.

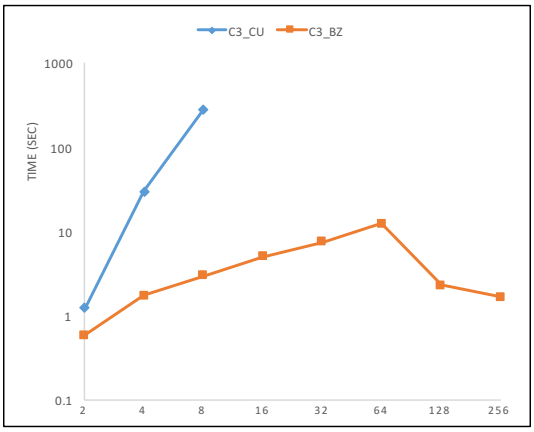
5.1 Test Core Update Algorithms

We implemented two approaches for the *updateCores* in the backward algorithm. One is proposed in [25], in our paper it is represented by $C3\_CU$ and $NC2\_CU$. Another one is using modified Batagelj and Zaversnik (BZ) algorithm to re-compute core numbers, $C3\_BZ$ for containing communities and $NC2\_BZ$ for non-containing communities. We test their performance on some graphs. Here we pick two graphs to show the result: LiveJournal and UK-2002. UK-2002 is the largest graph that we can get several results for varying k and r combinations on $C3\_CU$ and $NC2\_CU$ within one hour.

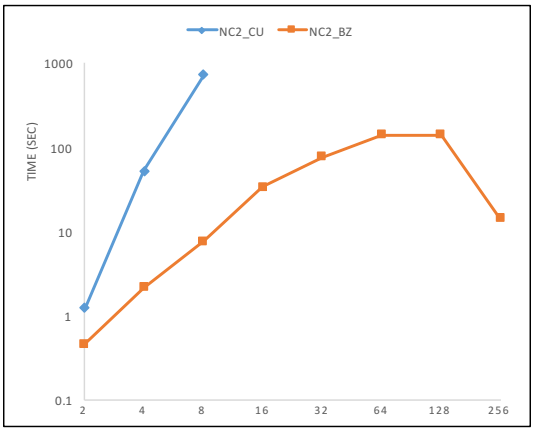
From Fig. 5.1, we can find $C3\_CU$ and $NC2\_CU$ are slower than $C3\_BZ$ and $NC2\_BZ$, we can not even get all the result points for UK-2002 and LiveJournal with CU algorithms. It demonstrates that when it comes to node update, in spite of we can treat it as a bunch of edge insertions, CU still has problems with dealing with billions of edge updates in a short time. Maybe it is a good approach to maintain a dynamic graph without the drastic change in the long term. However, our requirement is recomputing core numbers as soon as possible with thousands of edges updated in each iteration. The modified BZ can do this faster.

Another interesting thing we can see from the result is that although UK-2002 is bigger than LiveJournal, the computation time of modified BZ does not show an obvious increase. For the CU algorithm we can even get the result for more possible k and r combinations when graph gets bigger. This is because when the graph contains more nodes, we are more often able to resurrect a vertex that has degree greater than or equal to the k we set. Therefore, the re-computation of core numbers is not wasted. These resurrected nodes exist in the influential communities. On the other hand, when the graph is small, it is more difficult to find a vertex satisfying our requirements. It means we have to visit more vertices in the graph, which results in a longer computation time. We will explain this in detail in the next section.

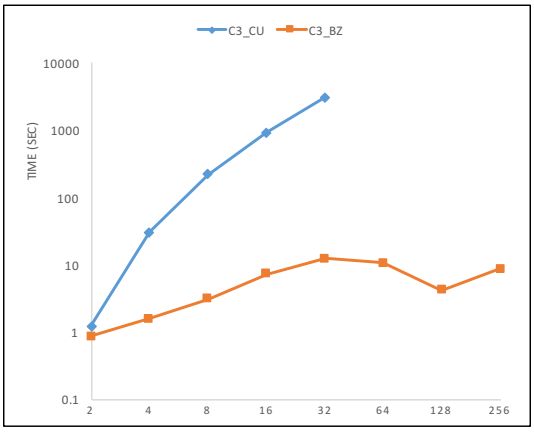
After this analysis, we realized that modified BZ is a better choice for our algorithm. So in the further experiments, we used the modified BZ to conduct the updateCores procedure in the backward algorithm and tested it against other graphs. What's more, our idea is proved in this comparison that backward algorithm is suitable for large size graph.



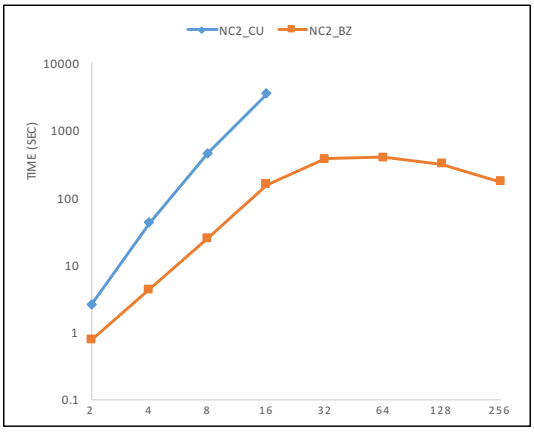
(a) LiveJ Cont



(b) LiveJ NC



(c) UK-2002 CONT



(d) UK-2002 NC

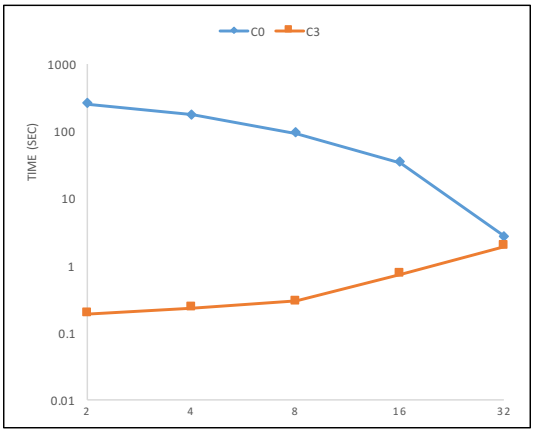
Figure 5.1: Incremental and recomputing algorithms performance on LiveJournal and UK2002 when varying k ($r = 10$)

5.2 Comparing with the Original Algorithms

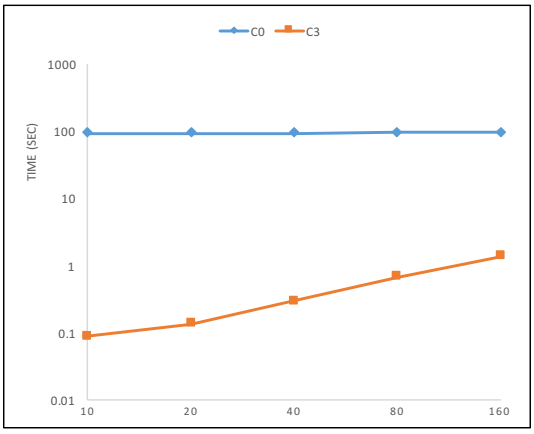
We start by comparing the backward algorithms with the original algorithms proposed in [32]. $C0$ is the original one for containing top r, k-influential communities and $NC0$ is for non-containing top r, k-influential communities. On the other hand, we have $C3$ and $NC2$ for containing top r, k-influential communities and non-containing top r, k-influential communities respectively. Because of the limitation of memory, we were only able to get the result for some k and r for $C0$ and $NC0$ on AstroPh and Livejournal.

From Fig. 5.2, we can see the difference between our algorithms and the original ones. For P1, the backward approach outperforms the original algorithm on both AstroPh and LiveJournal by orders of magnitudes in most cases regardless of k and r. As we expected, when k gets bigger, the backward algorithms runs longer to compute the communities. The reason is that few of the resurrected nodes have such a big k, especially for a graph like AstroPh with thousands of edges and nodes and the maximal degree is 56. So the backward algorithm has to perform many iterations of core re-computation. When k becomes considerable for graphs (e.g. 32 for AstroPh and 256 for LiveJournal), the gap between the backward approach and original is small. We have a similar situation with growing r in the backward algorithm, as we have to visit more vertices in the graph, and so the longer it takes to re-compute the core numbers. On the other hand, with the increase of k, the run time for original algorithms decrease as the size of C_k becomes smaller. Of course, it is faster to search communities in a smaller size graph. We vary r and find it barely has an influence on the time. This is because for the original algorithms, the dominant cost is peeling off nodes and computing the MCCs. As a result, the time difference of outputting varying top r results can be ignored. For P2, we can find that $NC0$ is even faster than $NC2$ with large k on AstroPh. The reason is similar to the one of P1. However, with the same parameter r, non-containing communities search must run more iterations than containing communities search. In the end, it visits more nodes in the graph and wastes more core re-computations.

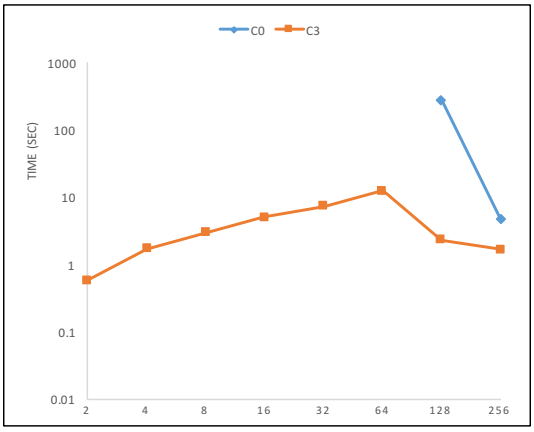
In some cases, the speed advantage of the backward algorithm is not clear. The original algorithm even beat backward when $k = 32$ on P2 for AstroPh. From the memory perspective, we still think backward is much better because original algorithms are only able to generate the result for AstroPh and LiveJournal when $k = 128$ and $k = 256$ in our consumer grade machine. We can use backward algorithms to handle massive graph. As the graph becomes bigger, the advantage of backward approaches gets more clear. This is not a surprise; according to our analysis in the last chapter, backward algorithms are not suitable for small size graphs with large k or r.



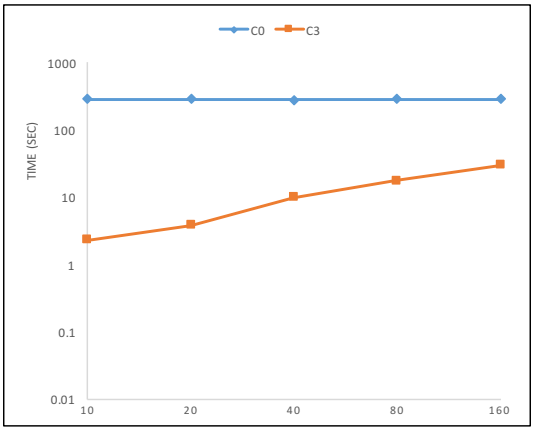
(a) AstroPh Cont vary k



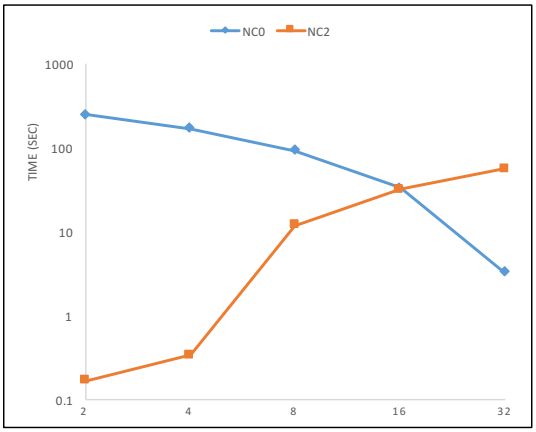
(b) AstroPh Cont vary r



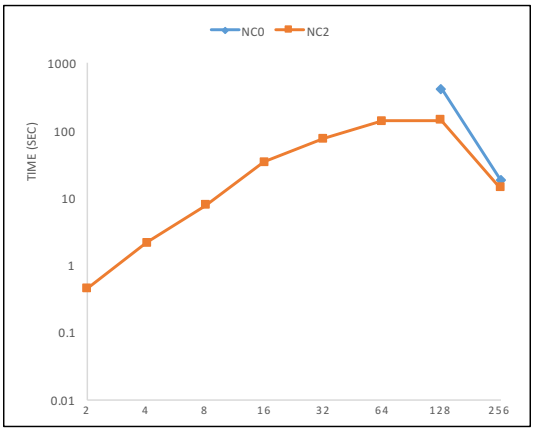
(c) LiveJ Cont vary k



(d) LiveJ Cont vary r



(e) AstroPh NC vary k



(f) LiveJ NC vary k

Figure 5.2: Original and proposed algorithms on AstroPh. and LiveJ. when varying k ($r = 40$), varying r ($k = 8$) for containing communities and varying k ($r = 10$) for non-containing communities.

5.3 Comparing with the Forward Algorithms

Here we conduct more experiments to compare the performance between backward algorithms with one forward algorithms. The forward algorithms are the most efficient ones so far for computing influential communities. We change the parameters (k and r) and try several combinations of them, which indeed helps us explore further on the advantage as well as some limitations of backward algorithms. We will describe them for both P1 and P2 separately.

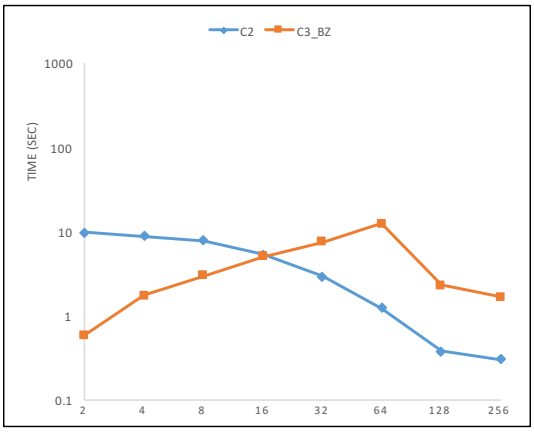
For P1, we first fixed the parameter r to 10 and 40, then changed the value of k gradually to see the trend. According to Fig. 5.3, we can see that $C3\_BZ$ beats $C2$ when k and r are relatively small. For example, when k is smaller than 16 on LiveJournal, we can see that $C3\_BZ$ is faster, and for a larger graph than LiveJournal like Arabic, as long as the value of k is smaller than 256, $C3\_BZ$ always outperforms $C2$. In general, as opposed to the forward algorithm, the computing time of backward algorithm rises as k and r increase. The reason is explained in the last subsection where we compared backward algorithms with the original ones. Both $C2$ and $C0$ are forward algorithms and they are based on the similar idea of peeling vertices from smallest weight vertex iteratively. Because $C2$ is way faster than $C0$, now we are able to compare the backward approach with the forward approach on larger graphs to provide reliable results. On one hand, we can see that when r is small ($r = 10$), we can see $C3\_BZ$ to be considerably faster than $C2$ (even an order of magnitude). Eventually, the curves will meet, and $C2$ becomes the winner. On the other hand, when $r = 40$, the shift happens at smaller k value. It suggests a relation between k and r. That is, for a specific graph, when the value of k multiple r is lower than a boundary, the backward approach is a better choice for the communities search.

After this set of experiments, we fixed another parameter k to 16 and 128. From Fig. 5.4, we find out that when r gets bigger and bigger, the computing time of $C2$ is pretty stable as the size of MCC does not change much with r. However, the backward approach needs to visit a larger part of the graph in order to get a sufficient number of influential communities. As such, the cost of re-computing core numbers leads the query time getting much longer. Especially when k is very large, the backward algorithm may never be faster than the forward in no scenario. For Twitter, we are not even able to get result for all the points within one hour.

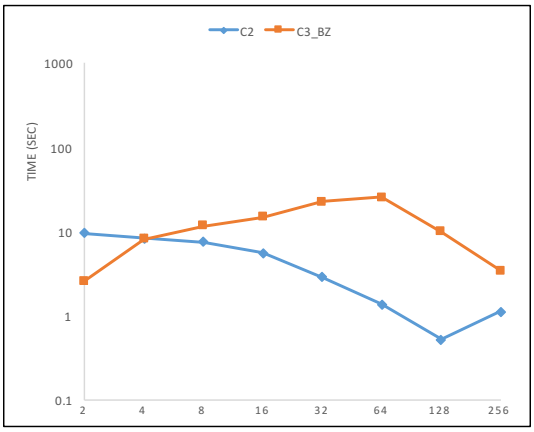
We also noticed that in the experiments against LiveJournal, UK-2002 and UK-2005 there are unexpected fluctuations when r is fixed with varying k. For example,

in Fig. 5.3c and Fig. 5.3d, the runtime for $C2$ goes down at $k = 256$ and then increase somehow at $k = 512$. Also, the runtime for $C3\_BZ$ decreases after $k = 32$ and then goes up at $k = 512$. The reason for the suddenly increase at $k = 512$ is computation time is also influenced by the graph structure. When $k = 256$, all the communities are constructed from a same big clique by few vertices elimination or addition. However, when $k = 512$, these communities come from different cliques, which means the loss of locality causes heavy computation task for our algorithms. Also, we can find a clue for the reason of decreased runtime after a point for $C3\_BZ$ in Table 4.1. We find LiveJournal, UK-2002 and UK-2005 have smaller kmax than other graphs we choose. Especially for UK-2005, although it has millions more nodes and edges than Arabic, the kmax is only about 1/6 of the kmax of Arabic. Recall the degree conditions we have in $updateCores$ to only be looking for updates if the resurrected vertex is well connected to the other resurrected vertices, so when k becomes large, it helps significantly reducing the number of updates. Because $kmax$ for Arabic and Twitter is quite larger than 512, so we do not see the decrease in our experiments for them.

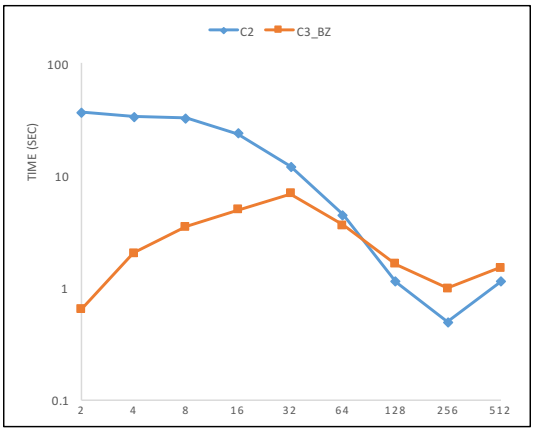
Additionally, we did experiments to query the non-containing communities with both of these approaches. Fig. 5.5 shows the result when k is varied for $r = 10$ and Fig. 5.6 shows the performance when r is varied for $k = 16$. From the result we can see the growing of k has more negative influence on $NC2\_BZ$. The backward approach only outperforms the forward one when k is very small. When k or r is large, we are even not able to get any result for Twitter due to the heavy dynamic computation of core number updating beyond our memory limitation. From this group of experiments we can conclude that the backward algorithm is not good for non-containing communities search compared to the forward algorithm. The reason is many re-computations of core numbers are wasted either because the resurrected vertex also exists in another community which means it will not lead us to a non-containing community, or it is not even a community. Finding containing ones has already been difficult for large k and r. So the disadvantage of backward method shows much more clear on P2.



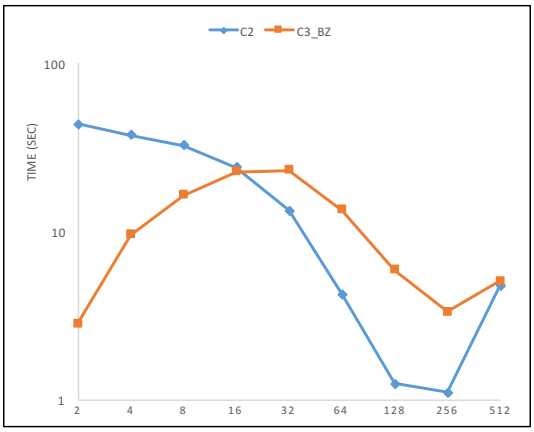
(a) LiveJournal r=10



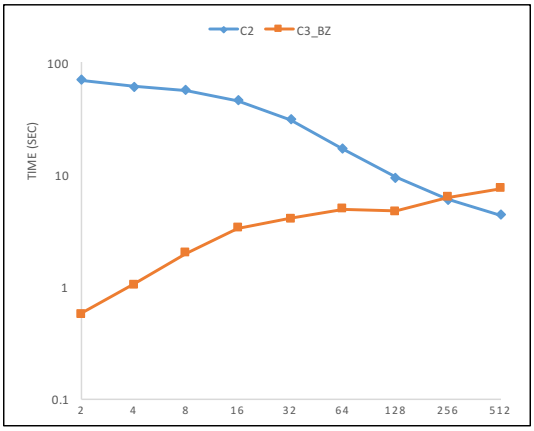
(b) LiveJournal r=40



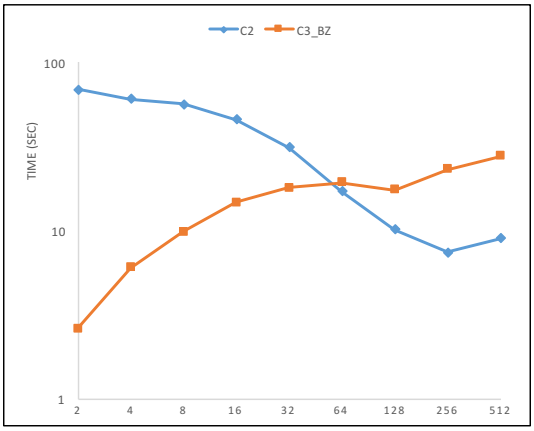
(c) UK-2002 r=10



(d) UK-2002 r=40

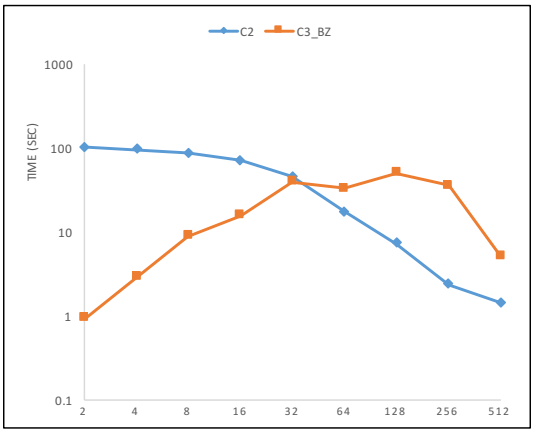


(e) Arabic r=10

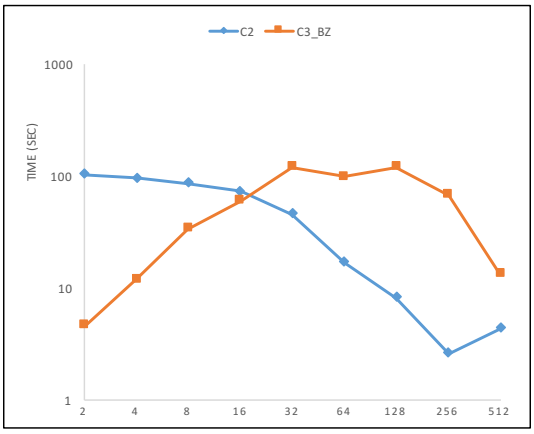


(f) Arabic r=40

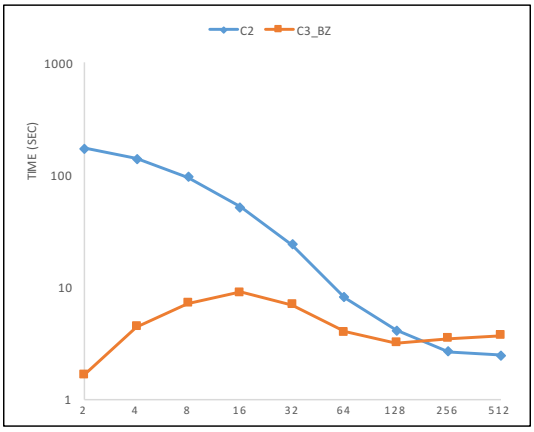
Figure 5.3: Performance of backward algorithm and forward algorithm on searching containing communities when varying k ($r = 10$ and $r = 40$)



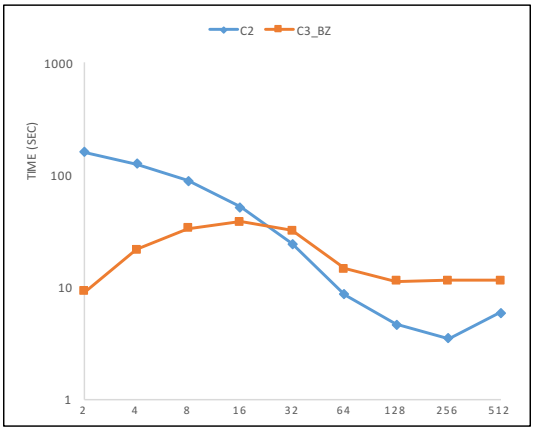
(g) UK-2005 r=10



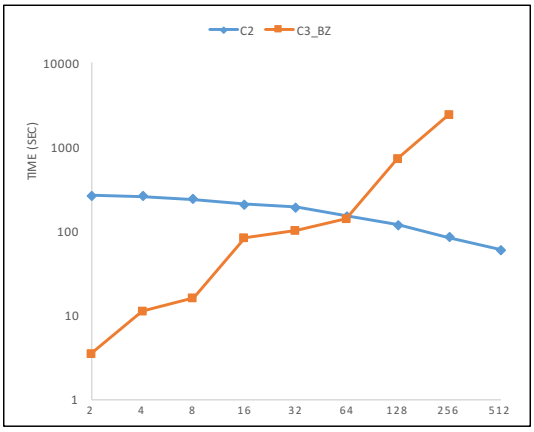
(h) UK-2005 r=40



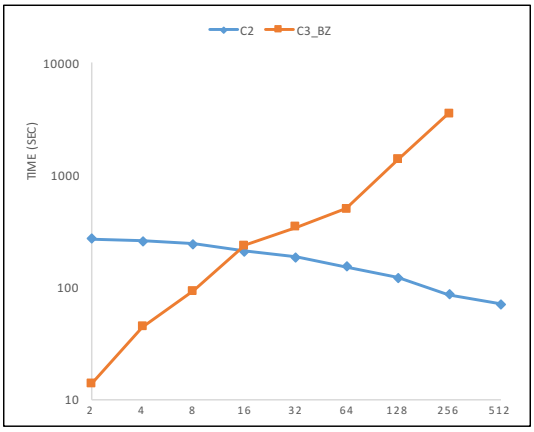
(i) WebBase r=10



(j) WebBase r=40

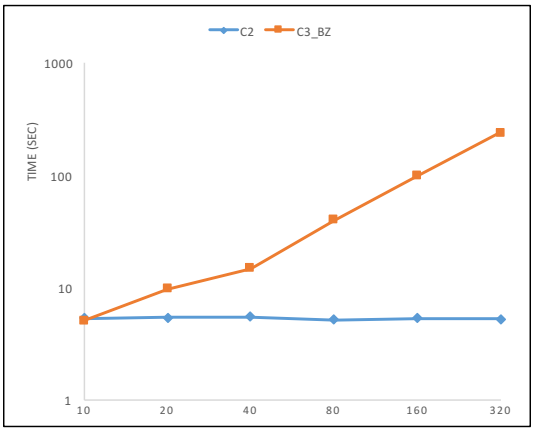


(k) Twitter r=10

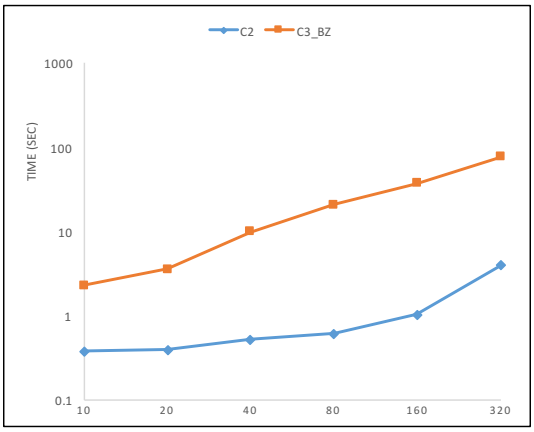


(l) Twitter r=40

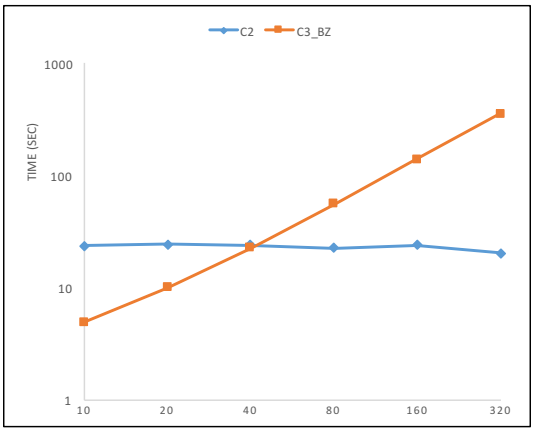
Figure 5.3: Performance of backward algorithm and forward algorithm on searching containing communities when varying k ($r = 10$ and $r = 40$)



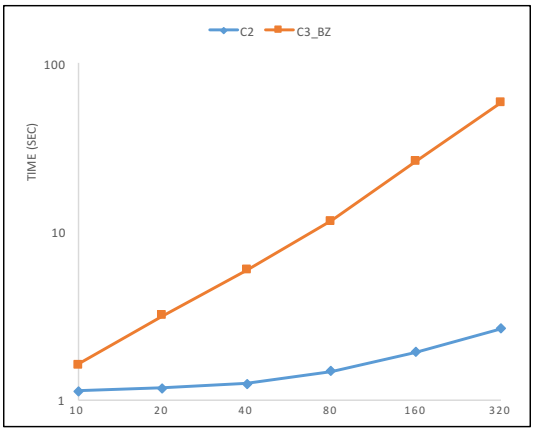
(a) LiveJournal k=16



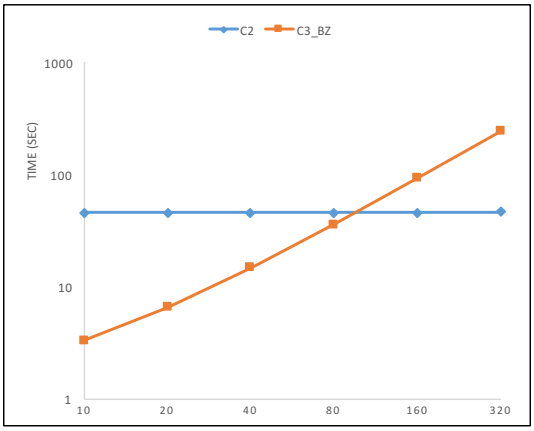
(b) LiveJournal k=128



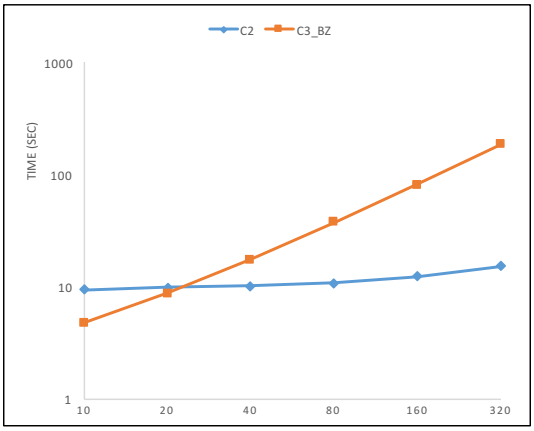
(c) UK-2002 k=16



(d) UK-2002 k=128

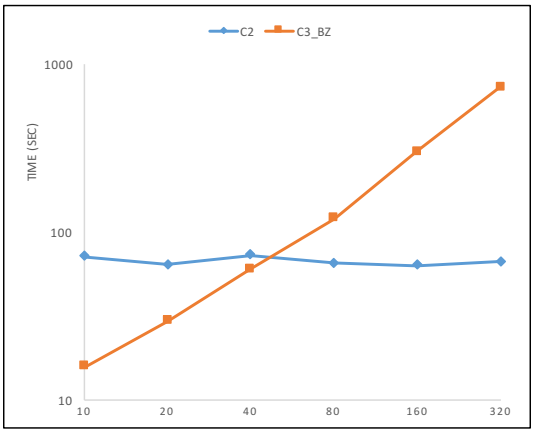


(e) Arabic k=16

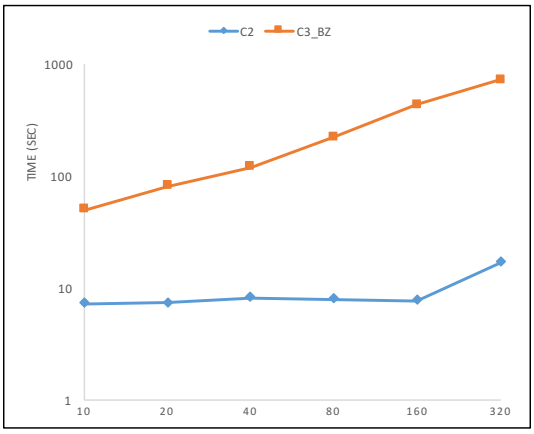


(f) Arabic k=128

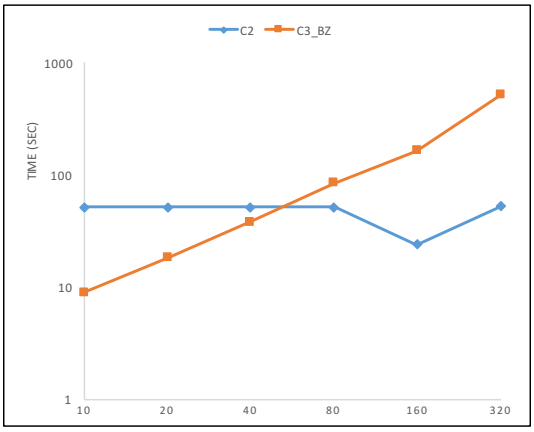
Figure 5.4: Performance of backward algorithm and forward algorithm on searching containing communities when varying r ($k = 16$ and $k = 128$)



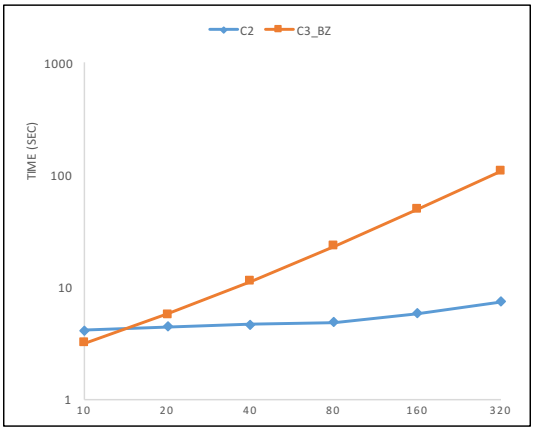
(g) UK-2005 k=16



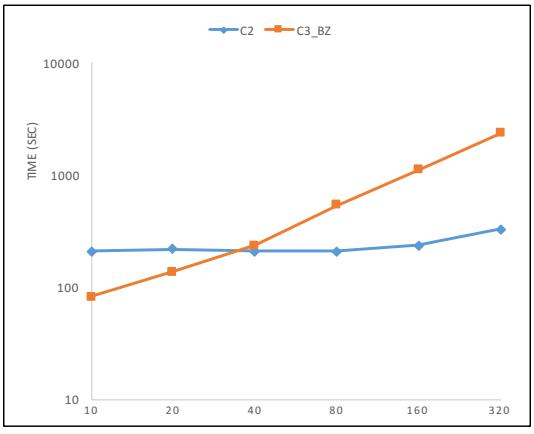
(h) UK-2005 k=128



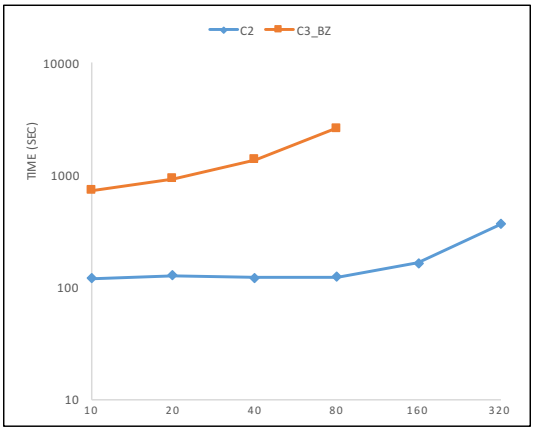
(i) WebBase k=16



(j) WebBase k=128

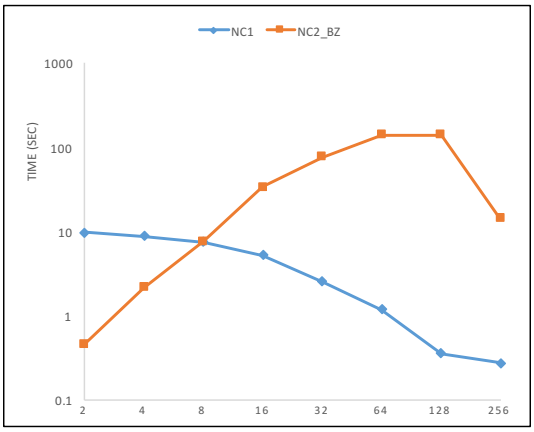


(k) Twitter k=16

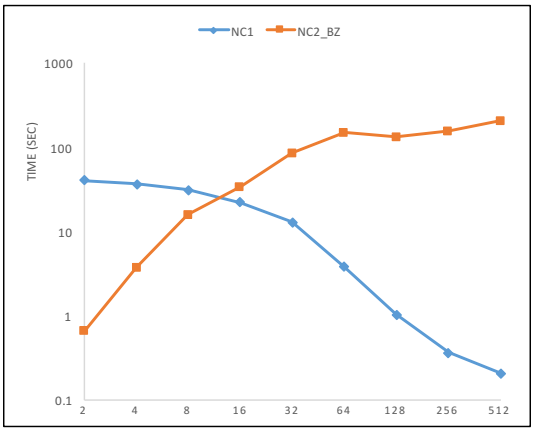


(l) Twitter k=128

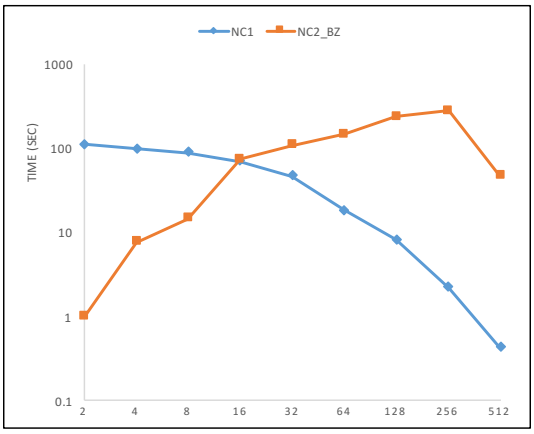
Figure 5.4: Performance of backward algorithm and forward algorithm on searching containing communities when varying r ($k = 16$ and $k = 128$)



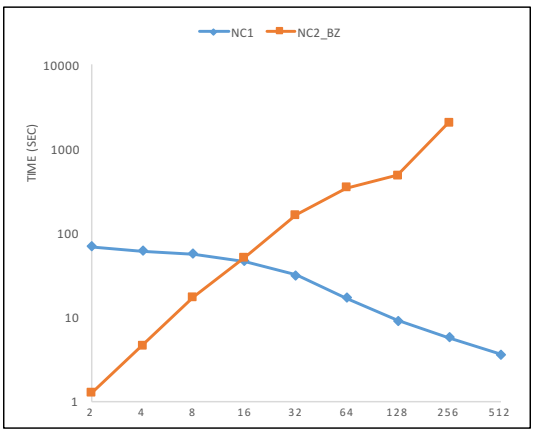
(a) LiveJournal r=10



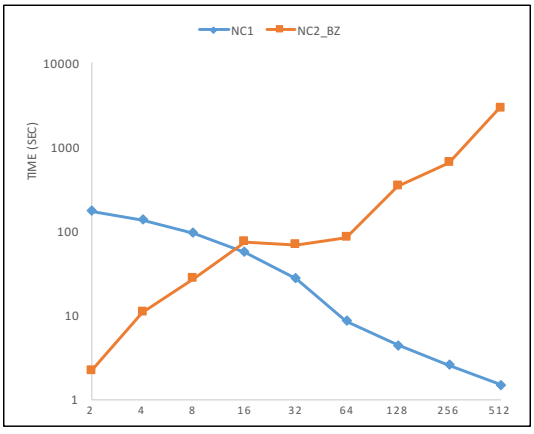
(b) UK-2002 r=10



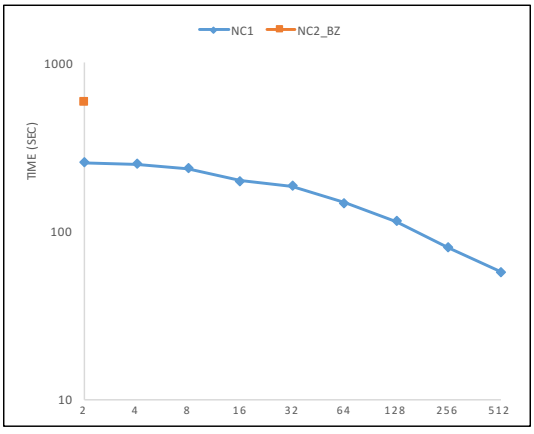
(c) UK-2005 r=10



(d) Arabic r=10

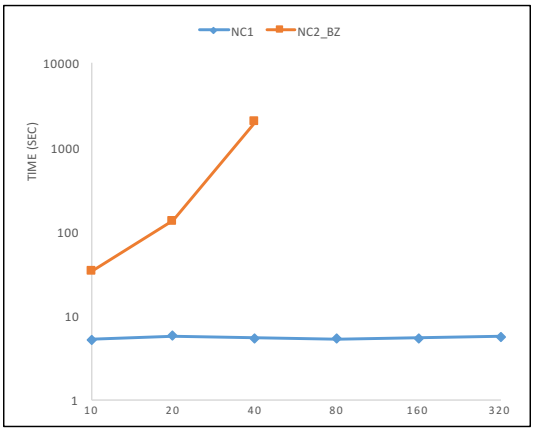


(e) WebBase r=10

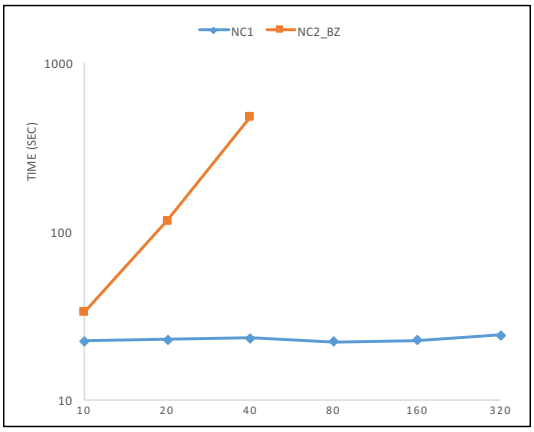


(f) Twitter r=10

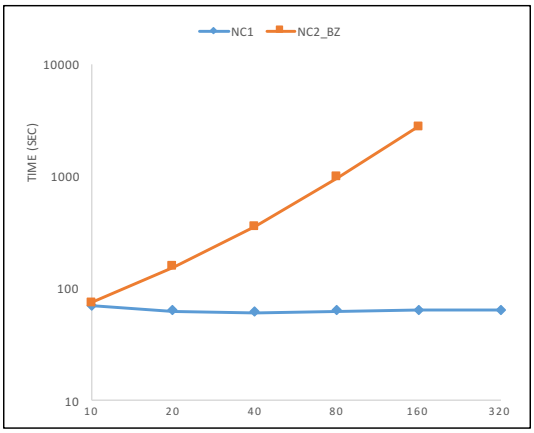
Figure 5.5: Performance of backward algorithm and forward algorithm on searching non-containing communities when varying k ($r = 10$)



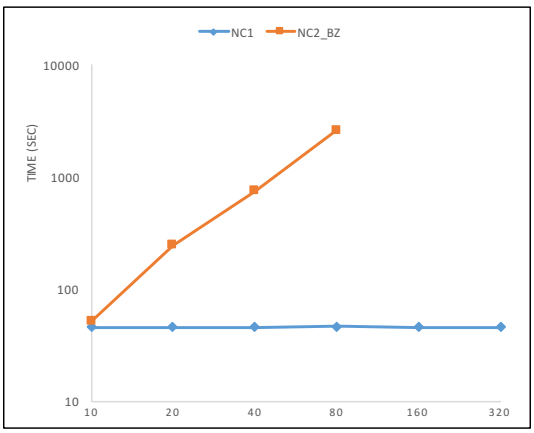
(a) LiveJournal k=16



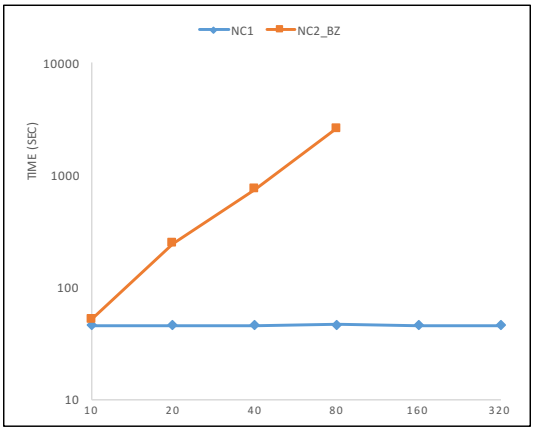
(b) UK-2002 k=16



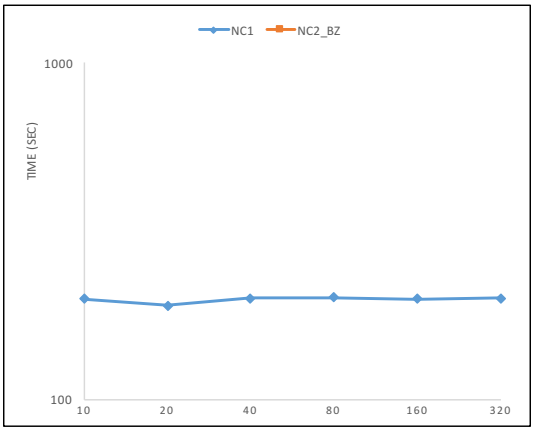
(c) UK-2005 k=16



(d) Arabic k=16



(e) WebBase k=16



(f) Twitter k=16

Figure 5.6: Performance of backward algorithm and forward algorithm on searching non-containing communities when varying r ($k = 16$)

5.4 Testing On ClueWeb

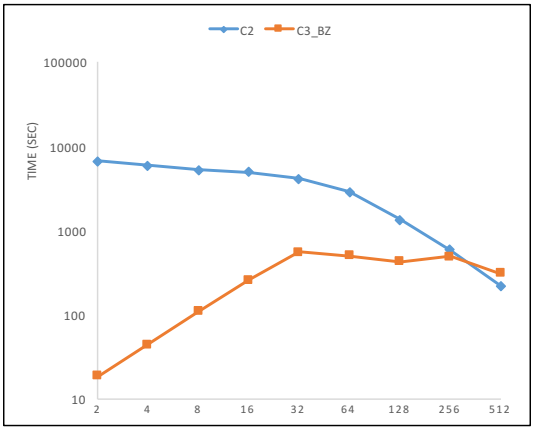
Furthermore, another improvement in this paper is we modified the data structure to fit massive graph, ClueWeb with billions of edges and nodes, in our consumer level machine. So we also did many experiments on ClueWeb to test backward algorithms as well as scale forward algorithms to consume a higher magnitude graph.

Fig. 5.7 shows the results for computing containing communities of ClueWeb with $C2$ and $C3\_BZ$ for varying k. It proves that when the graph gets bigger, backward approach becomes faster than forward. Especially when k and r are pretty small (when $r = 10$ and $k = 2, 4, 8, 16, 32$), the computation time of $C3\_BZ$ has orders of magnitude better performance than $C2$. From Fig. 5.8 we can see that when $k = 2$, $C3\_BZ$ is still way faster than $C2$ with a large r of 320.

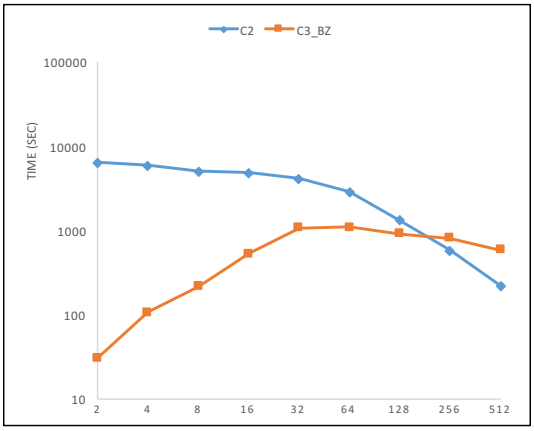
On the other hand, Fig. 5.7 shows again the conclusion we got before that when r grows bigger and $C3\_BZ$ has to get further on the graph to get wanted communities, the advantage of the backward algorithm on small k would not be too impressive and finally was beaten by the forward algorithm. Fig. 5.8 further presents the different influences of growing r on two different algorithms. $C2$ is very steady as the size of MCC does not change with r. But $C3\_BZ$ is very sensitive to the increasing r when k is unchanged.

For P2, the good result from observation is we get more numbers because of the memory increases. We can see that backward approach beats forward approach when r and k are relatively small. However, we still notice some points are missing in the chart which means we were unable to get result for these r and k. Comparing Fig. 5.8 with Fig. 5.10, we find that when $k = 2$, the performance gap of backward approaches between P1 and P2 is not obvious. However, when $k = 64$ or $k = 128$, we can hardly get results for any r by $NC2\_BZ$. It suggests that using backward algorithms on P2 for big k is not a smart choice for ClueWeb.

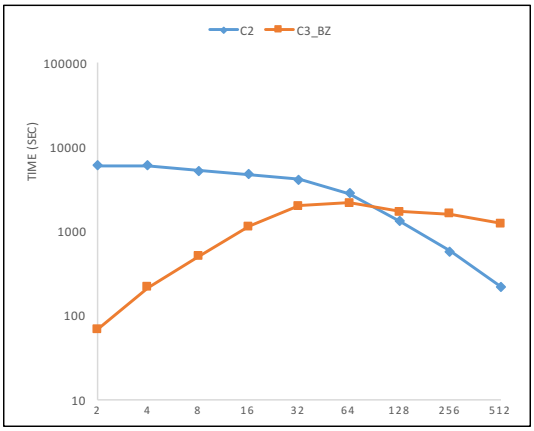
According to the experiments, we conclude that the backward algorithm for P1 is very efficient for small r and k regardless the size of graph. For example, when $k = 2$ and $r = 10$, $C3\_BZ$ always gets result within 60 sec from LiveJournal to ClueWeb.



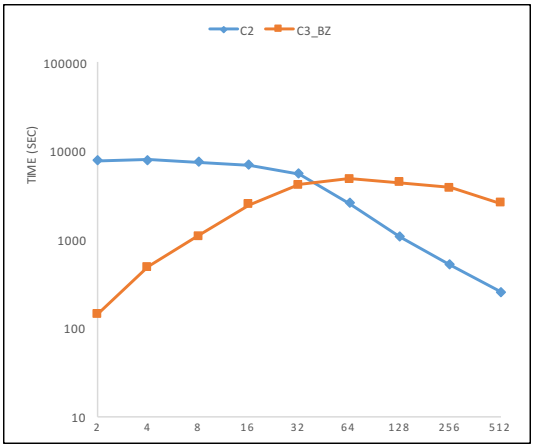
(a) r=10



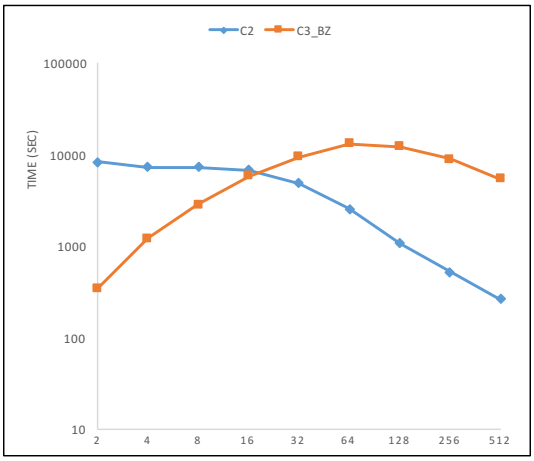
(b) r=20



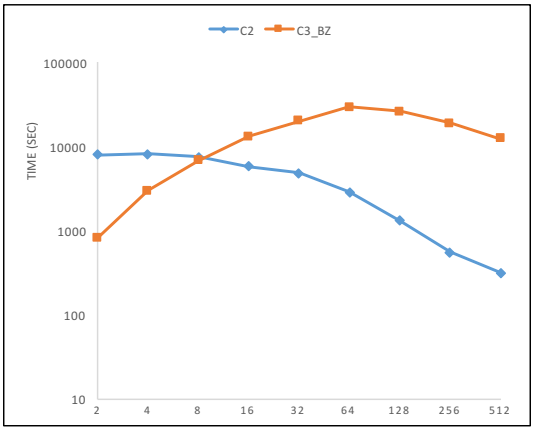
(c) r=40



(d) r=80

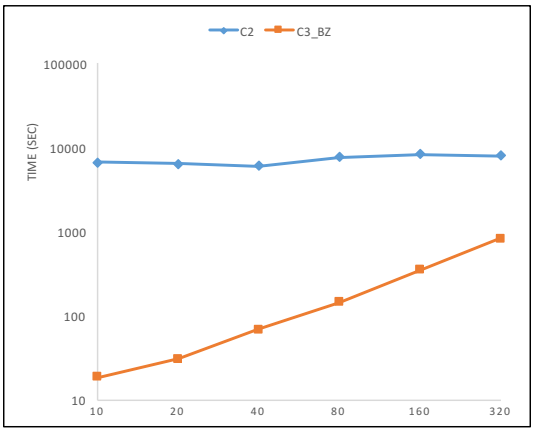


(e) r=160

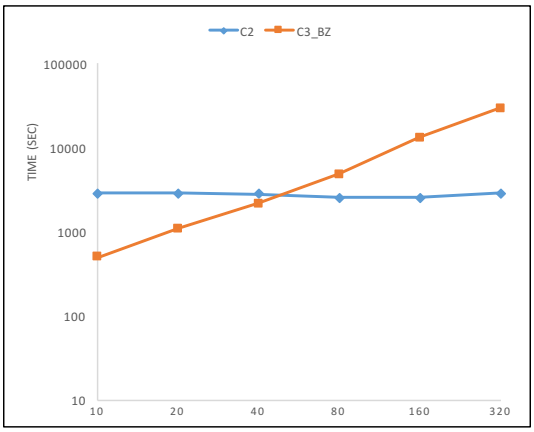


(f) r=320

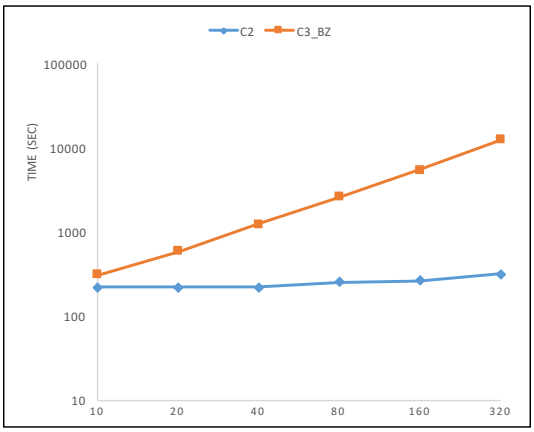
Figure 5.7: Performance of backward algorithm and forward algorithm on searching containing communities for ClueWeb when varying k



(a) k=2

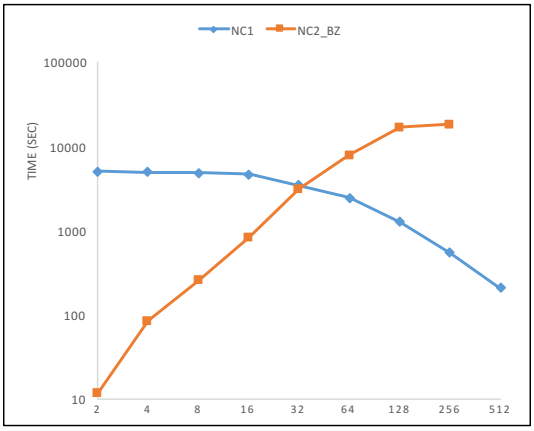


(b) k=64

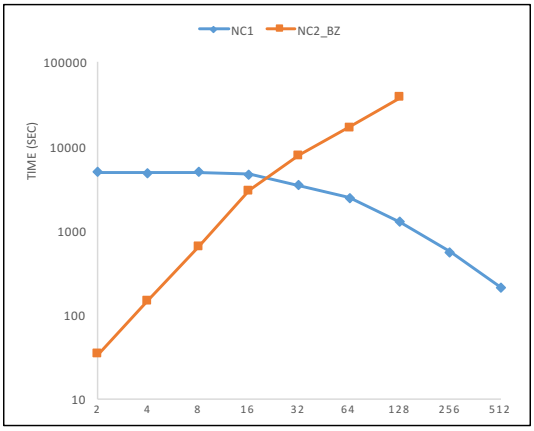


(c) k=128

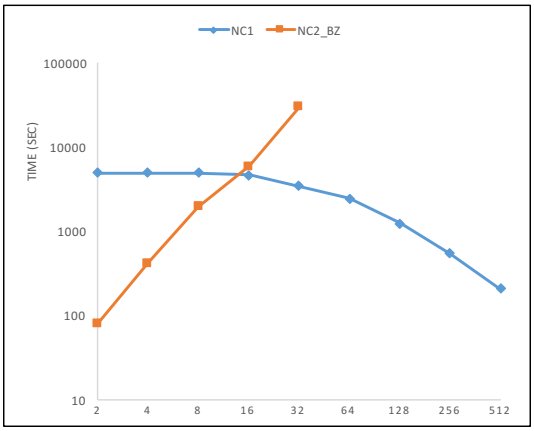
Figure 5.8: Performance of backward algorithm and forward algorithm on searching containing communities for ClueWeb when varying r



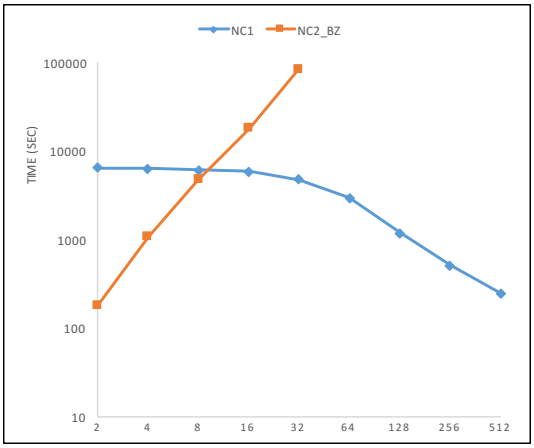
(a) r=10



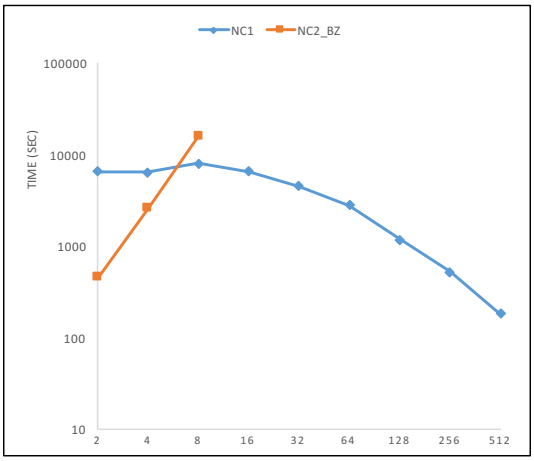
(b) r=20



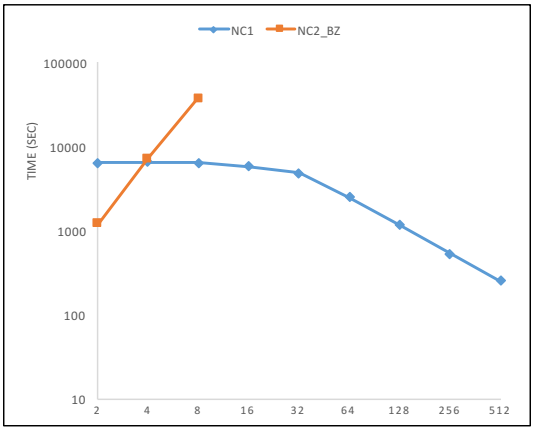
(c) r=40



(d) r=80

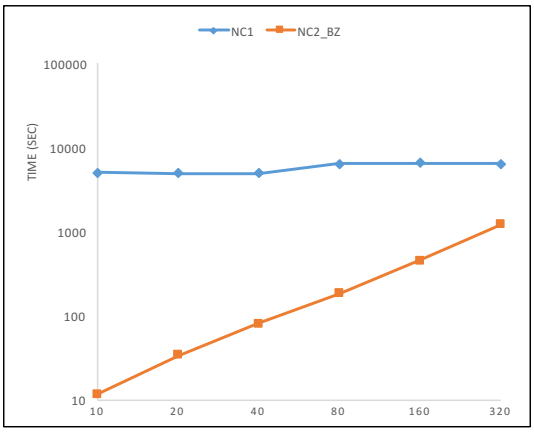


(e) r=160

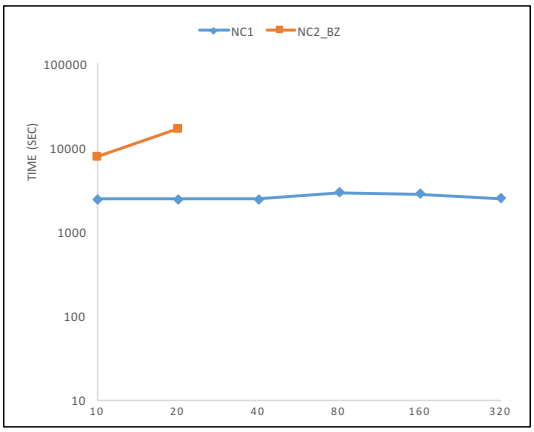


(f) r=320

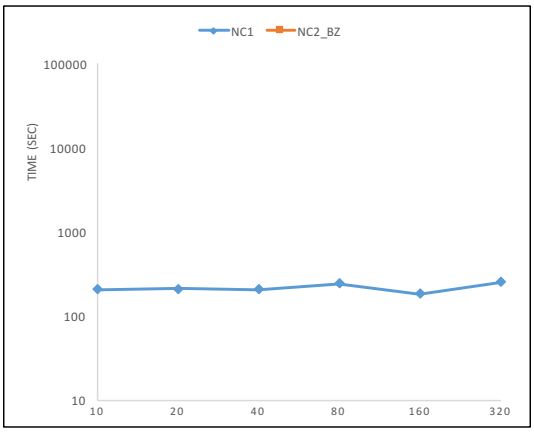
Figure 5.9: Performance of backward algorithm and forward algorithm on searching non-containing communities for ClueWeb when varying k



(a) k=2



(b) k=64



(c) k=128

Figure 5.10: Performance of backward algorithm and forward algorithm on searching non-containing communities for ClueWeb when varying r

Chapter 6

Conclusions and Future Work

This thesis presented a new approach to computing the top r, k influential communities. According to extensive experiments, we proved we can get results on massive real network graphs with billions of edges on a consumer level machine. Moreover, we do see the advantages of our approach in many cases compared to existing methods.

First, unlike the traditional forward algorithms, we proposed a new method to compute the influential communities in reverse. That is, instead of deleting nodes with the smallest weights in an intact graph, we started with resurrecting nodes with large weights and slowly reconstruct the graph. We implemented two different approaches to overcome a challenge with it and selected the better one to conduct the main testing in our research. Also, we modified the data structure of an outstanding forward algorithm, where we replaced the hash based structure to flat arrays to solve the memory issues. By doing this, we scaled the forward algorithms to compute influential communities of a massive graph with a consumer level machine.

We did experiments for both containing communities and non-containing communities on seven network graphs to evaluate the performance of backward algorithms. One of them is a massive graph with billions of edges. According to comparisons with the original algorithm, we concluded that the backward algorithms need less computation time than original ones. In particular, we can get results for large graphs which is not possible for the original algorithms. We also compared performance with the forward algorithms on varying k and r. The result shows that the backward algorithms beat forward ones for moderate k and r (In general, by moderate we mean the value of r is not larger than 64 or the value of k is not larger than 16 for a graph like Arabic), and they have better performance for searching containing communities than non-containing communities. Additionally, with the modification of data struc-

ture, now we are able to conduct experiments for a massive graph like ClueWeb with billions of edges and nodes with both forward and backward algorithms. We can get results for all the forward approaches and most of the cases for backward approaches. By doing experiments on this massive graph, we conclude that backward algorithms have better performance on large graphs.

In conclusion, we can say that the backward algorithms are a good choice for the case when the graph is massive like ClueWeb and/or r and k are moderate.

This work could be extended in several directions. First it is important to consider edge-labeled graphs, which is very common in social networks and graph-structured data (cf. [21, 19, 20]). For example, the edges on a network such as LinkedIn could be labeled by "co-worker", "colleague", "attended-same-university", etc. Then k-core based communities can be restricted to those defined by edges carrying only certain kinds of labels. Second, we are interested in extending this work to probabilistic graphs [8, 24], where the influence that an edge can have is given by a probability. Third, we would like to explore applications of the communities we compute in link and trust prediction [30, 46] as well as enhancing the quality of recommendation systems [14, 17, 28]. Fourth, we would like to investigate the role that the top-r, k-core communities can play in biological networks [23], in learning the news in social networks [34], and in clearing contamination from a network [38]. Finally, on the scalability side, we would like to devise distributed algorithms in the spirit of [37, 41], for computing top-r, k-core communities using clusters of many machines.

Chapter 7

Related Work

The community search in large networks has been extensively studied in both theory and industry area. A challenging task is how to effectively extract cohesive induced subgraphs in analyzing graphs. Cohesive subgraph is an important concept in social network analysis. At the very beginning, clique structure is defined to measure the cohesiveness of a network. Many research groups have worked on extracting cliques following different requirements. In [12, 13], Cheng et. al. proposed a series of external-memory algorithms for finding and enumerating maximal clique. They extracted communities by moving parts of the graph into the main memory and then combining the results so that they can solve the problem of memory management when the sizes of graphs become too large. However, the strict definition of clique maybe too strong for many applications. There are many different relaxed definitions of cohesive graphs in the literature, such like k-truss [15] and k-core [36], etc. Interestingly, many equivalent concepts of k-truss were defined in different papers. For example, Saito et. al. defined a k-truss community model in [35], which is followed by Gregori in [22]. On the other hand, in [47], another term k-truss k-mutual-friend subgraph is defined by the authors.

The concept of k-core is first introduced by Seidman in [36] and many studies are based on this definition. Batagelj and Zaversnik proposed an $O(m+n)$ algorithm for k-core decomposition in [6] and it is implemented in this paper. Later, in [11] Cheng et. al. invented a new algorithm for the disk-resident graphs using a top-to-down method to overcome the random access limitation. Li et. al. came up with a new approach to update core numbers for the dynamic graphs effectively in [33], which is also implemented in this paper. k-core decomposition has been extensively used in community search. Sozio et. al. [40] studied the community search problem of

finding the maximal connected k-core that containing the query node. In that paper, they proposed a linear time algorithm to solve it. Later, Cui et. al. introduced a more efficient method to solve the same problem in [16]. Some researchers make use of it to extract refined communities (cf. [1, 9, 48]), such as maximal k-edge connected subgraphs.

Besides community search, k-core decomposition has also been widely used for many applications, including visualization of large networks (cf. [3, 5]), analyzing the biological network structure in [2], and identifying influential spreader in networks [29]. In addition to community search, there is another research problem called community discovery which aims to discover all the communities in a network. This topic is studied in the literature extensively as well. For example, in [18], Fortunato gave a thorough exposition from the definition of the main elements of the problem to the techniques designed so far. Also, Xie et. al. reviewed the algorithms for detecting overlapping communities as well as proposed a framework for evaluating the algorithms' ability to detect overlapping communities in [45].

Bibliography

[1] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Linear-time enumeration of maximal k-edge-connected subgraphs in large networks by random contraction. In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*, pages 909–918. ACM, 2013.

[2] Md. Altaf-Ul-Amine, Kensaku Nishikata, Toshihiro Korna, Teppei Miyasato, Yoko Shinbo, Md. Arifuzzaman, Chieko Wada, Maki Maeda, Taku Oshima, Hirotada Mori, and Shigehiko Kanaya. Prediction of protein functions based on k-cores of protein-protein interaction networks and amino acid sequences. *Genome Informatics*, 14:498–499, 2003.

[3] J. I. Alvarez-hamelin, Luca Dall'asta, Alain Barrat, and Alessandro Vespignani. Large scale networks fingerprinting and visualization using the k-core decomposition. In Y. Weiss, B. Schlkopf, and J. Platt, editors, *Advances in Neural Information Processing Systems 18*, pages 41–50. MIT Press, Cambridge, MA, 2005.

[4] J. Ignacio Alvarez-Hamelin, Luca Dall'Asta, Alain Barrat, and Alessandro Vespignani. k-core decomposition: a tool for the analysis of large scale internet graphs. *CoRR*, abs/cs/0511007, 2005.

[5] Vladimir Batagelj, Andrej Mrvar, and Matjaž Zaveršnik. *Partitioning Approach to Visualization of Large Graphs*, pages 90–97. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

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

[7] Paolo Boldi and Sebastiano Vigna. The webgraph framework i: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*, pages 595–602. ACM, 2004.

[8] Francesco Bonchi, Francesco Gullo, Andreas Kaltenbrunner, and Yana Volkovich. Core decomposition of uncertain graphs. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 1316–1325. ACM, 2014.

[9] Lijun Chang, Jeffrey Xu Yu, Lu Qin, Xuemin Lin, Chengfei Liu, and Weifa Liang. Efficiently computing k-edge connected components via graph decomposition. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 205–216. ACM, 2013.

[10] Shu Chen, Ran Wei, Diana Popova, and Alex Thomo. Efficient computation of importance based communities in web-scale networks using a single machine. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, CIKM '16, pages 1553–1562, New York, NY, USA, 2016. ACM.

[11] James Cheng, Yiping Ke, Shumo Chu, and M Tamer Özsu. Efficient core decomposition in massive networks. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 51–62. IEEE, 2011.

[12] James Cheng, Yiping Ke, Ada Wai-Chee Fu, Jeffrey Xu Yu, and Linhong Zhu. Finding maximal cliques in massive networks. *ACM Transactions on Database Systems (TODS)*, 36(4):21, 2011.

[13] James Cheng, Linhong Zhu, Yiping Ke, and Shumo Chu. Fast algorithms for maximal clique enumeration with limited memory. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1240–1248. ACM, 2012.

[14] Maria Chowdhury, Alex Thomo, and William W Wadge. Trust-based infinitesimals for enhanced collaborative filtering. In *COMAD*, 2009.

[15] Jonathan Cohen. Trusses: Cohesive subgraphs for social network analysis. *National Security Agency Technical Report*, 2008.

[16] Wanyun Cui, Yanghua Xiao, Haixun Wang, and Wei Wang. Local search of communities in large graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 991–1002. ACM, 2014.

[17] Sahar Ebrahimi, Norha M Villegas, Hausi A Müller, and Alex Thomo. Smarterdeals: a context-aware deal recommendation system based on the smartercontext engine. In *Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research*, pages 116–130. IBM Corp., 2012.

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

[19] Gösta Grahne and Alex Thomo. Query containment and rewriting using views for regular path queries under constraints. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 111–122. ACM, 2003.

[20] Gösta Grahne and Alex Thomo. Query answering and containment for regular path queries under distortions. In *FoIKS*, volume 4, pages 98–115. Springer, 2004.

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

[22] Enrico Gregori, Luciano Lenzini, and Chiara Orsini. k-dense communities in the internet as-level topology graph. *Computer Networks*, 57(1):213–227, 2013.

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

[24] Nasrin Hassanlou, Maryam Shoaran, and Alex Thomo. Probabilistic graph summarization. In *Proceedings of the 14th International Conference on Web-Age Information Management*, WAIM'13, pages 545–556. Springer-Verlag, 2013.

[25] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. Querying k-truss community in large and dynamic graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1311–1322. ACM, 2014.

[26] Xin Huang, Laks VS Lakshmanan, Jeffrey Xu Yu, and Hong Cheng. Approximate closest community search in networks. *Proceedings of the VLDB Endowment*, 9(4):276–287, 2015.

[27] Wissam Khaouid, Marina Barsky, Venkatesh Srinivasan, and Alex Thomo. K-core decomposition of large networks on a single pc. *Proceedings of the VLDB Endowment*, 9(1):13–23, 2015.

[28] Maryam Khezrzadeh, Alex Thomo, and William W Wadge. Harnessing the power of favorites lists for recommendation systems. In *Proceedings of the third ACM conference on Recommender systems*, pages 289–292. ACM, 2009.

[29] Maksim Kitsak, Lazaros K. Gallos, Shlomo Havlin, Fredrik Liljeros, Lev Muchnik, H. Eugene Stanley, and Hernn A. Makse. Identification of influential spreaders in complex networks. *Nature Physics*, 6(11):888893, 2010.

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

[31] Victor E. Lee, Ning Ruan, Ruoming Jin, and Charu Aggarwal. *A Survey of Algorithms for Dense Subgraph Discovery*, pages 303–336. Springer US, Boston, MA, 2010.

[32] Rong-Hua Li, Lu Qin, Jeffrey Xu Yu, and Rui Mao. Influential community search in large networks. *Proceedings of the VLDB Endowment*, 8(5):509–520, 2015.

[33] Rong-Hua Li, Jeffrey Xu Yu, and Rui Mao. Efficient core maintenance in large dynamic graphs. *Knowledge and Data Engineering, IEEE Transactions on*, 26(10):2453–2465, 2014.

[34] Krishnan Rajagopalan, Venkatesh Srinivasan, and Alex Thomo. A model for learning the news in social networks. *Annals of Mathematics and Artificial Intelligence*, 73(1):125–138, 2015.

[35] Kazumi Saito, Takeshi Yamada, and Kazuhiro Kazama. Extracting communities from complex networks by the k-dense method. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 91(11):3304–3311, 2008.

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

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

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

[39] Michael Simpson, Venkatesh Srinivasan, and Alex Thomo. Efficient computation of feedback arc set at web-scale. *Proceedings of the VLDB Endowment*, 10(3):133–144, 2016.

[40] Mauro Sozio and Aristides Gionis. 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*, pages 939–948. ACM, 2010.

[41] Dan C. Stefanescu, Alex Thomo, and Lida Thomo. Distributed evaluation of generalized path queries. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, SAC '05, pages 610–616. ACM, 2005.

[42] Babak Tootoonchi, Venkatesh Srinivasan, and Alex Thomo. Efficient implementation of anchored 2-core algorithm. In *ASONAM*, 2017.

[43] Johan Ugander, Lars Backstrom, Cameron Marlow, and Jon Kleinberg. Structural diversity in social contagion. *Proceedings of the National Academy of Sciences of the United States of America*, 109(16):5962–5966, 2012.

[44] T. Wolf, A. Schrter, D. Damian, L. D. Panjer, and T. H. D. Nguyen. Mining task-based social networks to explore collaboration in software teams. *IEEE Software*, 26(1):58–66, Jan 2009.

[45] Jierui Xie, Stephen Kelley, and Boleslaw K Szymanski. Overlapping community detection in networks: The state-of-the-art and comparative study. *ACM Computing Surveys (csur)*, 45(4):43, 2013.

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

[47] Feng Zhao and Anthony K. H. Tung. Large scale cohesive subgraphs discovery for social network visual analysis. In *Proceedings of the 39th international conference on Very Large Data Bases*, PVLDB'13, pages 85–96. VLDB Endowment, 2013.

[48] Rui Zhou, Chengfei Liu, Jeffrey Xu Yu, Weifa Liang, Baichen Chen, and Jianxin Li. Finding maximal k-edge-connected subgraphs from a large graph. In *Proceedings of the 15th International Conference on Extending Database Technology*, pages 480–491. ACM, 2012.

[49] M. ngeles Serrano, Marin Bogu, Alessandro Vespignani, and Peter J. Bickel. Extracting the multiscale backbone of complex weighted networks. *Proceedings of the National Academy of Sciences of the United States of America*, 106(16):6483–6488, 2009.