Influential Community Discovery in Massive Social Networks Using a Consumer-Grade Machine

by

Shu Chen
B.Eng., Hangzhou Dianzi University, 2014

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

© Shu Chen, 2017
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

Influential Community Discovery in Massive Social Networks Using a Consumer-Grade Machine

by

Shu Chen
B.Eng., Hangzhou Dianzi University, 2014

Supervisory Committee

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

Dr. Kui Wu, Departmental Member
(Department of Computer Science)

Supervisory Committee

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

Dr. Kui Wu, Departmental Member
(Department of Computer Science)

ABSTRACT

Graphs have become very crucial as they can represent a wide variety of systems in different areas. One interesting structure called *community* in graphs has attracted considerable attention from both academia and industry. Community detection is meaningful, but typically hard in arbitrary networks. A lot of research has been done based on structural information, but we would like to find communities which are not only cohesive but also influential or important. A *k-influential community* model based on k-core provided by Li, Qin, Yu, and Mao is helpful to discover these cohesive and important communities. They organize the problem as finding top-r most important communities in a given graph.

In this thesis, our goal is to detect top-r most important communities using efficient and memory-saving algorithms running on a consumer-grade machine. We analyze two existing algorithms, then propose multiple new efficient algorithms for this problem. To test their performance, we conduct extensive experiments on some real-world graph datasets. Experimental results show that our algorithms are able to compute top-r most important communities within a very reasonable amount of time and space in a consumer-grade machine.

Contents

Supervisory Committee **ii**

Abstract **iii**

Table of Contents **iv**

List of Tables **vi**

List of Figures **vii**

List of Algorithms **ix**

Acknowledgements **x**

Dedication **xi**

1 Introduction **1**

1.1 Motivation 1

1.2 Contributions 3

1.3 Agenda 4

2 Related Work **5**

3 Background **8**

3.1 Basics of k-core and k-influential community 8

3.1.1 k-core and k-core decomposition 8

3.1.2 k-influential community 9

3.2 k-core decomposition algorithm 11

3.3 WebGraph 14

4 Initial Algorithms **16**

4.1 Two problems 16
4.2 DFS-based algorithms $C0$ and $NC0$ 16

5 Algorithms for P1 and P2 **21**
5.1 Algorithm C1 for Problem 1 22
5.2 Algorithm C2 for Problem 1 23
5.3 Algorithm NC1 for Problem 2 27
5.4 Algorithm NC2 for Problem 2 29
5.5 Conclusion 32

6 Experimental Results **33**
6.1 Datasets 33
6.2 Equipment 35
6.3 Testing initial algorithms 35
6.3.1 Problem 1 35
6.3.2 Problem 2 36
6.3.3 Conclusion 38
6.4 Testing for Problem 1 38
6.4.1 Experiment 1 - fixed r 38
6.4.2 Experiment 2 - fixed k 41
6.4.3 Conclusion 44
6.5 Testing for Problem 2 44
6.5.1 Experiment 1 - fixed r 44
6.5.2 Experiment 2 - fixed k 48
6.5.3 Conclusion 51

7 Conclusions and Future Work **52**

A Additional Information **54**
A.1 Algorithm $C2$ 54
A.2 Algorithm $NC2$ 59

Bibliography **64**

List of Tables

Table 6.1 Properties of datasets ordered by m. 34
Table 6.2 Ranges of parameters k and r. 34
Table 6.3 Sizes of subgraphs of AstroPh and Slashdot0811 in different k . 36
Table 6.4 Sizes of subgraphs of arabic-2005 in different k 40
Table 6.5 Sizes of subgraphs when $k = 16$ 41
Table 6.6 Details of $NC1$ running on uk-2002 when $r = 40$ 47

List of Figures

Figure 1.1 Communities in a network of network scientist [28] 2

Figure 2.1 Clique example 6
(a) 1-vertex clique 6
(b) 2-vertex clique 6
(c) 3-vertex clique 6
(d) 4-vertex clique 6

Figure 3.1 k-core decomposition for a sample graph 9
Figure 3.2 BZ algorithm applied in a simple graph 12
Figure 3.3 Results of BZ algorithm Core procedure 13
(a) After process node 3 13
(b) After process neighbor 1 of node 0 13
(c) After process neighbor 2 of node 0 13
(d) After process node 4 13
(e) After process node 1 13
(f) After process node 2 13

Figure 4.1 $C_{k,i}$ and $H_{k,i}$, for $k = 2$ and $i \in [1, 4]$. The original graph in (a) are all in black, grayed out vertices and edges are deleted. Weights are the vertex id's. 19
(a) $C_{2,1}$ 19
(b) $H_{2,1}$ 19
(c) $C_{2,2}$ 19
(d) $H_{2,2}$ 19
(e) $C_{2,3}$ 19
(f) $H_{2,3}$ 19
(g) $C_{2,4}$ 19
(h) $H_{2,4}$ 19

Figure 5.1 $C_{k,i}$ for $k = 2$ and $i \in [1, 5]$. The original graph in (a) are all in black, grayed out vertices and edges are deleted. 24
(a) original graph, $C_{2,1}$ 24
(b) after delete $I(1)$, $C_{2,2}$ 24
(c) after delete $I(2)$, $C_{2,3}$ 24
(d) after delete $I(3)$, $C_{2,4}$ 24
(e) after delete $I(4)$, $C_{2,5}$ 24

Figure 6.1 Problem 1 - AstroPh (r=40) 35
Figure 6.2 Problem 1 - Slashdot0811 (r=40) 36
Figure 6.3 Problem 2 - AstroPh (r=40) 37
Figure 6.4 Problem 2 - Slashdot0811 (r=40) 37
Figure 6.5 Problem 1 - Pokec (r=40) 39
Figure 6.6 Problem 1 - LiveJournal1 (r=40) 39
Figure 6.7 Problem 1 - uk-2002 (r=40) 39
Figure 6.8 Problem 1 - arabic-2005 (r=40) 40
Figure 6.9 Problem 1 - Pokec (k=16) 41
Figure 6.10Problem 1 - LiveJournal1 (k=16) 42
Figure 6.11Problem 1 - uk-2002 (k=16) 42
Figure 6.12Problem 1 - arabic-2005 (k=16) 42
Figure 6.13Sizes of top 320 CCI communities 43
Figure 6.14Problem 2 - Pokec (r=40) 45
Figure 6.15Problem 2 - LiveJournal1 (r=40) 45
Figure 6.16Problem 2 - uk-2002 (r=40) 45
Figure 6.17Problem 2 - arabic-2005 (r=40) 46
Figure 6.18Problem 2 - Pokec (k=16) 48
Figure 6.19Problem 2 - LiveJournal1 (k=16) 49
Figure 6.20Problem 2 - uk-2002 (k=16) 49
Figure 6.21Problem 2 - arabic-2005 (k=16) 49
Figure 6.22NC1 running on uk-2002 (k=256) 50

List of Algorithms

1 Batagelj-Zaversnik (BZ) algorithm 11
2 Top-r CCI communities (C0) 17
3 RDelete 18
4 Search Maximally Connected Component (MCC) 18
5 Top-r non-containing CCI communities (NC0) 20
6 Top-r CCI communities (C1) 23
7 Top-r CCI communities (C2) 25
8 RDelete2 25
9 MCC with alive array (MCC2) 26
10 Top-r non-containing CCI communities (NC1) 28
11 Top-r non-containing CCI communities (NC2) 31
12 RDelete3 32

ACKNOWLEDGEMENTS

I would like to thank:

My supervisor, Dr. Alex Thomo, for his support and mentoring throughout the past two years. I am deeply grateful to him for providing patient guidance and insightful comments on this thesis.

My teammates, Diana Popova and Ran Wei, for their ideas, helps, and friendship. This thesis would not be accomplished without their efforts.

My parents, for always being there, supporting me, believing in me, and loving me. I am eternally grateful.

My friends, Fei, Yunlong, Zheng, and others for their sincere friendship, and accompanying me in the low moments. I am so fortunate to make friends with them.

Dr. Kui Wu and Dr. Issa Traore, for serving in my thesis examining committee.

DEDICATION

This thesis is dedicated to my mother.
For being my best friend forever.

Chapter 1

Introduction

Along with the growth of Internet and computer revolution, graphs have become extremely crucial as they can represent a wide variety of systems in different areas. Biology, physics, social sciences, computer science, and other disciplines frequently use graphs to represent entities and their connections. In many graph applications, finding dense, cohesive sub-graphs (also known as communities) is of paramount importance.

1.1 Motivation

Discovering communities in real networks is one of the important tasks in analyzing graphs because community structures are highly correlated with the functionality of the networks in most of the cases [14]. For example, in the graph of websites, communities may represent groups of pages related to similar topics; in the graph of social networks, communities are likely to group users whose number of followers are big. Figure 1.1 displays a graph where each node represents a scientist and colored according to their community membership [28]. Visually, groups of nodes have a high density of links within communities and share the same color.

k-core, a well-known concept in graph theory, has been applied in many community detection problems [40, 8, 1, 35, 13] because it can be computed in polynomial time and used as a subroutine for harder problems [21]. Consider the following scenario. In social networks, it is common that the engagement of users is more likely if their friends are engaged. Similarly, if connected users drop out, then their friends might drop out too. Therefore, it is important for social network providers to calculate the least number of friends a user needs to stay in the network, and determine

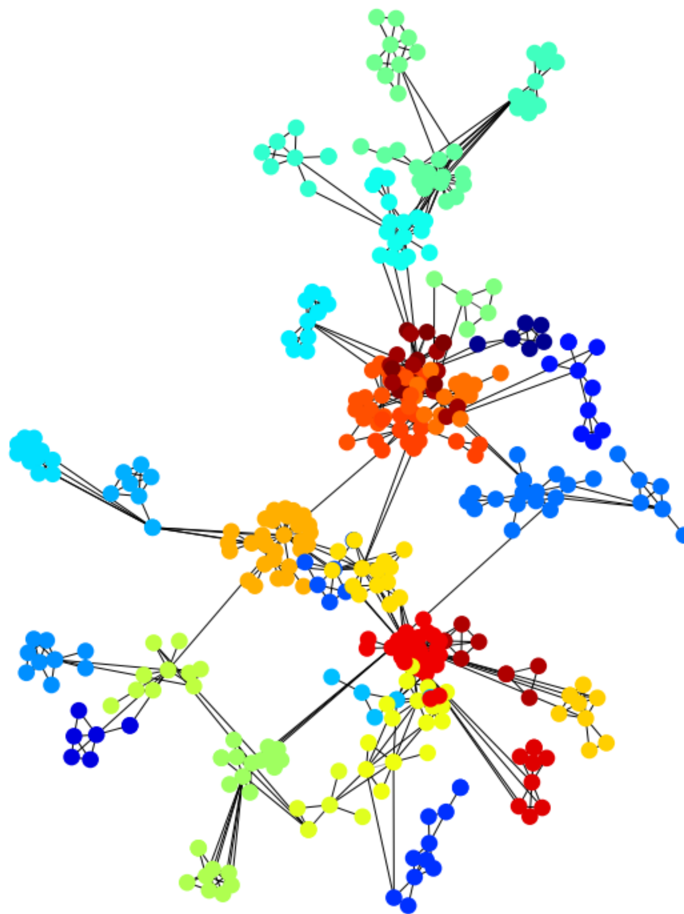


Figure 1.1: Communities in a network of network scientist [28]

the remaining active users after a series of disengagement [21]. This corresponds to k-core of a graph G, which is the largest induced subgraph of G where every vertex has degree at least k.

A lot of research about community detection has been done based on graph structure only. In this thesis, we would like to find the communities which not only have a high density of links, but are also influential or important [9]. For example, in a network of graph-theory researchers, we would like to find well-connected communities which contain persons whose citation rates are high.

Li, Qin, Yu, and Mao introduced a new community model [24] based on k-core, called *k-influential community*, to capture well-connected and influential communities. Given an undirected graph G, they assign each node a weight, which is a numerical value that indicates the influence or importance of the node. The influence of a k-influential community is measured by the minimum weight of nodes in this community.

The k-influential communities are maximally connected, k-core subgraphs of G with the highest influence.

Community detection is meaningful, but typically hard in arbitrary networks [14]. The structure of networks varies a lot in different graphs, and the number or size of communities are unknown. According to the constraints of the k-influential community, discovering them directly is not practical in large real networks. The massive amount of data on large networks significantly increases computation and space complexity.

Although there are many difficulties, Li, Qin, Yu, and Mao in [24] have provided several algorithms with varying performance and space requirements for k-influential community discovery. Based on these algorithms, we would like to know whether there is a faster and more efficient approach for computing k-influential communities.

In summary, calculating *k-influential communities* using fewer resources is the topic we would like to research in this thesis.

1.2 Contributions

The goal of this thesis is to speed-up the computation of k-influential communities in large scale networks, and we would like to achieve this using only a consumer-grade machine.

Even though the algorithms in [24] have greatly improved the performance of discovering k-influential communities, we propose four more efficient algorithms which use space in the order of the compressed version of the graph. As a graph compression framework, we used WebGraph, a highly efficient Java package for reducing the footprint of very large graphs. More precisely, our contributions are as follows.

1. We provide two fast algorithms which require reasonable memory for computing top-r k-influential communities. Compared with the online algorithm in [24], our algorithms are faster by orders of magnitude.

2. We present two fast and memory-saving algorithms for computing top-r non-containing k-influential communities. One of them completely eliminates the computation of maximally connected components, which greatly decreases the running time.

3. We conduct extensive experiments on a selection of real-world network datasets.

The biggest graph we used has about 22 million nodes and 553 million edges. We set testing communities for varying k and r, and the results show that our algorithms are able to compute communities within very reasonable time and space on a consumer-grade machine.

1.3 Agenda

Chapter 1 introduces our motivations and objectives for community detection, and some basic concepts about the k-influential model.

Chapter 2 presents related work on community detection.

Chapter 3 describes background knowledge which supports our algorithms, including detailed concepts of k-core and k-core decomposition, precise definitions of the k-influential model, a fast in-memory k-core decomposition algorithm, and WebGraph, a graph compression framework.

Chapter 4 begins with precise definitions of two problems that we would like to solve in this thesis, then two initial algorithms from [24] for computing k-influential communities and non-containing k-influential communities are introduced.

Chapter 5 proposes two algorithms for the first problem we defined in Chapter 4, and another two algorithms for the second problem. Our ideas and pseudocodes of each algorithm are provided.

Chapter 6 contains experimental methodologies, results, and analyses.

Chapter 7 concludes the thesis and discusses future works.

Appendix displays two complete implementations of our new algorithms.

Chapter 2

Related Work

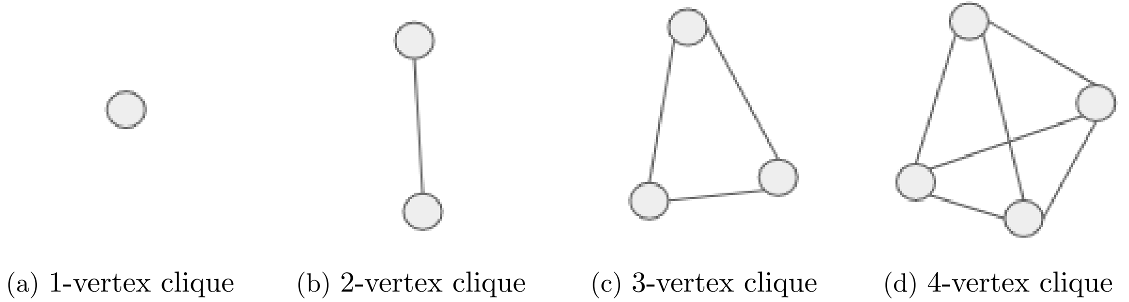
This thesis is inspired by data-mining work on extracting a set of cohesive subgraphs as communities in graphs and networks. Discovering communities is a crucial task to understand physical functions implied by graphs [13, 35].

In recent years, community detection has drawn a large amount of attention and research. In [16], Gregori et. al. developed algorithms to extract a set of k-dense communities from the Internet AS-level topology graph which enables researchers to gain more insight into the structure of the Internet. Koch [22] represented a new method for finding all connected maximal common subgraphs which can be regarded as communities in two graphs. Ground-truth community is defined and detected by Yang et. al. in [39].

Clique is one of the basic concepts in graph theory which is a classic dense subgraph structure. In an undirected graph $G = (V, E)$, a clique is a subset of the vertices such that every two distinct vertices are adjacent. Figure 2.1 shows some examples of clique. Despite the fact that finding a clique of a given size in a graph is an NP-complete problem, a large number of works and algorithms have been developed.

In [11], Cheng et. al. present an external-memory algorithm (EmMCE) for maximal clique enumeration in large graphs. EmMCE proves to be more efficient compared with the conventional in-memory algorithm when the input graph cannot fit in memory. Koch [22] transforms the problem of finding maximal common subgraphs in two graphs into the clique problem.

Nevertheless, the strict definition of the clique is not practical for various applications as it is unlikely that every entity would have a link to every other entity within the subgraph. Some concepts with relaxed definitions have been proposed. n-clique, a generalized concept of clique, was introduced in 1950 [26]. Alba [2] proposed the



(a) 1-vertex clique (b) 2-vertex clique (c) 3-vertex clique (d) 4-vertex clique

Figure 2.1: Clique example

concept of n-club, a maximal subgraph of diameter n, in 1973. k-plex, a structure defined as a graph with n vertices in which each vertex is connected by a path of length 1 to at least $n - k$ of the other vertices, was presented in 1978 [32]. Seidman [31] introduced the concept k-core in 1983. In k-core, each vertex needs to have at least k neighbors.

Compared with most of the other concepts, k-core can be computed and maintained in polynomial time [25, 30], and k-core decomposition, a process of calculating the core number for each node in a graph can be applied to really large graphs [21, 38]. In [10], Cheng et. al. proposed the first external-memory algorithm (EMcore) for core decomposition in massive graphs which are too large for keeping in main memory. Their experimental results show that EMcore is efficient for core decomposition in graphs with up to 52.9 million vertices and 1.65 billion edges.

In addition, k-core decomposition has been broadly used for community detection. In [40, 8, 1], they detect a new refined community called k-edge-connected subgraph using k-core decomposition as a foundation. In [35], Sozio and Gionis studied the problem of finding a community in a graph given a set of query nodes, and they provided a global search strategy based on k-core decomposition. They also found that, in the community search problem, the minimum degree is a better measure than other measures, such as average degree and density. Cui et. al. [13] investigate a similar problem of finding the best community containing a given query vertex in its neighborhood, and they proposed a local search method using k-core concept.

However, all these mentioned works do not consider the influence of a community. In this thesis, we focus on k-influential community which is constructed based on k-core by Li et. al. [24]. They provided an online search algorithm and an optimal index-based algorithm for k-influential community detection, and our work in this

thesis uses the online search algorithm of [24] as a foundation.

Based on k-core, a new structure called k-truss which represents the “core” of a k-core is proposed by Cohen [12]. The definition of k-truss is stricter than that of k-core. Namely, it is a subgraph of k-core, where every edge is contained in at least $k-2$ triangles [37]. Community detection based on k-truss has been studied too. In [19], Huang et. al. defined a novel k-truss community model based on the k-truss concept, and provided an algorithm which runs in linear time with respect to the community size. Given a graph and a set of query nodes, the closest truss community search problem is studied in [20].

In next chapter, we introduce comprehensively the concept of k-core, and the k-influential community model which we study in this thesis.

Chapter 3

Background

In this chapter, we present the necessary background information which we use in this thesis. Section 3.1 presents the concepts of k-core and k-influential community, as well as an example of k-core. In Section 3.2, an efficient k-core decomposition algorithm we used, the Batagelj-Zaversnik algorithm, will be introduced. In Section 3.3, we briefly go over a graph compression framework, WebGraph, which is used for decreasing space complexity.

3.1 Basics of k-core and k-influential community

3.1.1 k-core and k-core decomposition

We denote an undirected graph by $G = (V, E)$, where V is the set of vertices, and E is the set of edges. We set $n = |V|$ and $m = |E|$, respectively. Given a vertex v, we denote by $d_G(v)$ the degree of v in G.

The notion of k-core was introduced by Seidman [31] in 1983. Similarly, we have the following definitions following [27].

Definition 1. Given a subgraph C of G induced by subset of nodes S, the degree of node v in C is denoted by $d_C(v)$. C is a k-core if and only if C is a maximal induced subgraph of G such that $\forall v \in C$, $d_C(v) \geq k$.

For a subgraph C of G which is a k-core, we denote it as C_k. We have the following definition of coreness.

Definition 2. A node v in G has coreness or core number k if and only if v belongs to C_k, but not C_{k+1}.

The process of calculating coreness or core number for each node in G is called *k-core decomposition*. Figure 3.1 shows an example of k-core decomposition for a sample graph.

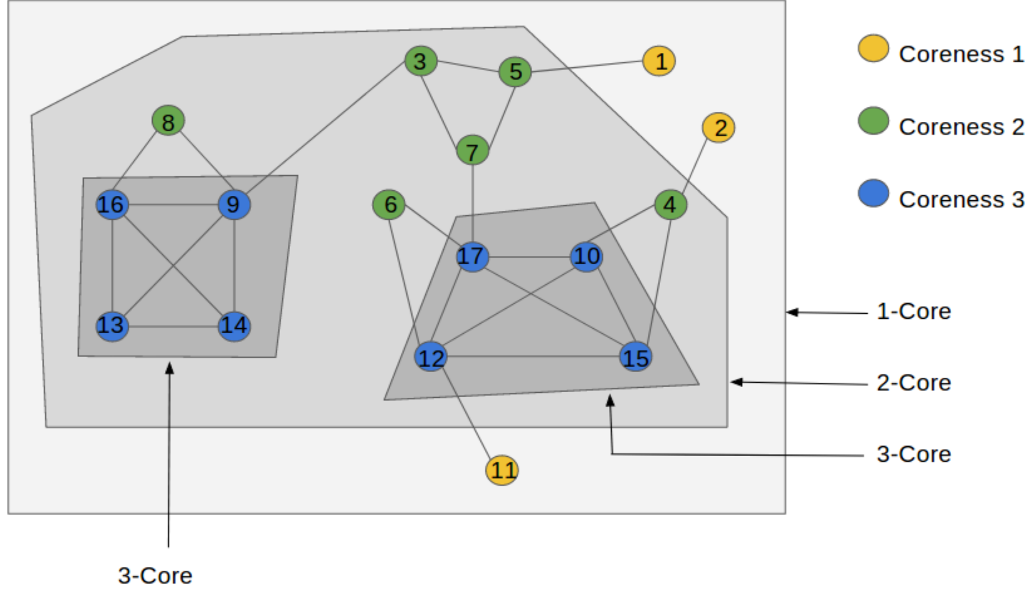


Figure 3.1: k-core decomposition for a sample graph

As described in Figure 3.1, we do not have nodes with coreness 0 in the graph because there is no node without any connection. 1-core is the entire graph G, 2-core is a subgraph which removes the three yellow nodes from G, and 3-core is a subgraph which contains two separated connected components.

Based on definitions and the example, we can get a conclusion that there is at most one k-core in G for every $k = 1, 2, \ldots$. In addition, k-cores are nested, e.g. 3-core $\subseteq$ 2-core in Figure 3.1. Moreover, k-cores $C_k(G)$ are not necessarily connected subgraphs, they can contain multiple maximally connected components.

3.1.2 k-influential community

k-influential community is a model described by Li, Qin, Yu, and Mao in [24] to capture well-connected and influential communities in graphs. It is built based on k-core.

Given a graph, each node can have a weight, a numerical value which indicates the influence or importance of the node. Such a value can be computed using PageRank,

or it can be some attribute of the social network user that the node represents, e.g. age, social status, number of citations, etc.

The influence of a k-influential community is measured by the minimum weight of its nodes. For an induced subgraph H (group of nodes in graph G), there are three constraints for it to be a k-influential community [24].

1. All nodes in H should be connected.

2. Each node in H has degree at least k.

3. H is not contained in other induced subgraphs which satisfy the last two constraints, and have the same influence.

Based on the above constraints, a k-influential community obviously is a k-core cohesive subgraph.

Consider Figure 3.1 for an example. For simplicity of illustration, we set the weights of vertices to be their id's. There are two connected components in the 3-core subgraph $C_3(G)$. We set the left connected component as $C1$ and the right one as $C2$. $C1$ is a 3-influential community with influence 9 because (a) the minimum-weight vertex in $C1$ is vertex 9, (b) it is a connected component where each node has degree at least 3, (c) it is not contained in any other connected subgraph where all nodes have degree 3 or more and influence 9 or more. Similarly, $C2$ is another 3-influential community with influence 10.

Given r and k, the k-influential community problem is to find the top-r (with the highest influences) k-influential communities a graph.

We note that, in the top-r results, a k-influential community can contain another k-influential community of higher influence value. In order to avoid inclusion relationships in top-r results, Li et. al. [24] provide a constraint for checking *non-containing k-influential community*. According to this constraint, a *non-containing* k-influential community H with influence w should not contain some other k-influential community whose influence value is larger than w.

In order to make it more accurate, we call k-influential communities as *connected-cohesive-important(CCI) communities*, and non-containing k-influential communities as *non-containing CCI communities* in following chapters.

Discovering *CCI communities* and *non-containing CCI communities* are two problems we would like to explore in this thesis. To capture such communities, we need to calculate coreness for each node in the graph at first. The next section will describe

an efficient, $O(m)$, algorithm for determining the core decomposition of a given graph [4].

3.2 k-core decomposition algorithm

In the k-core decomposition problem, the goal is to calculate the coreness or core number of each node in a graph. The core number of a node is the highest value of k that the node belongs to a k-core. One property is useful when searching k-core [3]:

> Given a graph G, if we recursively delete all vertices of degree less than k, and their incidental edges, the remaining of graph G is the k-core.

How to rapidly find nodes whose degrees are less than k at each iteration is a challenge in k-core decomposition. Sorting all the remaining nodes in the graph by their degree is a solution. Batagelj-Zaversnik (BZ) algorithm [4], a very efficient algorithm for cores decomposition, is designed based on recursively deleting nodes whose degrees are less than k until the degrees of all remaining nodes are larger than or equal to k, and bin-sort to rapidly find target nodes. The pseudocode of the BZ algorithm is shown in Algorithm 1.

Algorithm 1 Batagelj-Zaversnik (BZ) algorithm

```
Input: A graph G = (V, E), n = |V| and m = |E|
Output: An array deg contains coreness of each node
 1: deg[v] ← compute degree of each node v
 2: bin[i] ← start index of the first node whose degree equals to i in vert
 3: vert[i] ← nodes sorted by degrees in ascending order (bin-sort)
 4: pos[v] ← positions of node v in vert
 5: for i = 0 upto n do
 6:     v = vert[i]
 7:     for all u ∈ neighbors(v) do
 8:         if deg[u] > deg[v] then
 9:             Find the first node w whose degree equals to deg[u] in vert
10:             if u! = w then
11:                 Swap u and w in vert and pos
12:             bin[deg[u]] ← bin[deg[u]] + 1
13:             deg[u] ← deg[u] − 1
14:
15: Output deg
```

The input of the BZ algorithm is a whole graph, and the output is an array or table which contains core number for each node. Given a graph $G = (V, E)$ where V is the set of vertices and E is the set of edges, then set $n = |V|$ and $m = |E|$. It requires four arrays, *deg*, *vert*, *pos*, and *bin*, for coreness calculation. The first array *deg* is initialized to record the degrees of corresponding nodes, and the size of it is n. e.g. $deg[v]$ means the degree of node v where $0 \leq v < n$. The sizes of arrays *vert* and *pos* are both n. The *vert* array contains the set of nodes ascending ordered by their degrees, while positions of nodes in the array *vert* are stored in the *pos*, e.g. if $vert[i] = v$, then $pos[v] = i$. The size of array *bin* equals to the maximum degree of the graph, we denote it as M. The *bin* array is initialized to contain the number of nodes which have the same degree, e.g. if there are 7 nodes with degree 0, and 5 nodes with degree 1, then $bin[0] = 7$ and $bin[1] = 5$. To avoid an additional array, as the algorithm proceeds, the *bin* array records the position of the first node of the corresponding degree in *vert*, e.g. the array $bin[i]$ contains the index of the first node with degree i in the *vert*. Practically, the *vert* array can be divided into multiple blocks of vertices, e.g. block i contains all nodes with degree i, where $0 \leq i \leq M$. If the first 10 nodes in *vert* have degree 0, and the next 9 nodes have degree 1, then $bin[0] = 0$, $bin[1] = 10$, and $bin[2] = 19$.

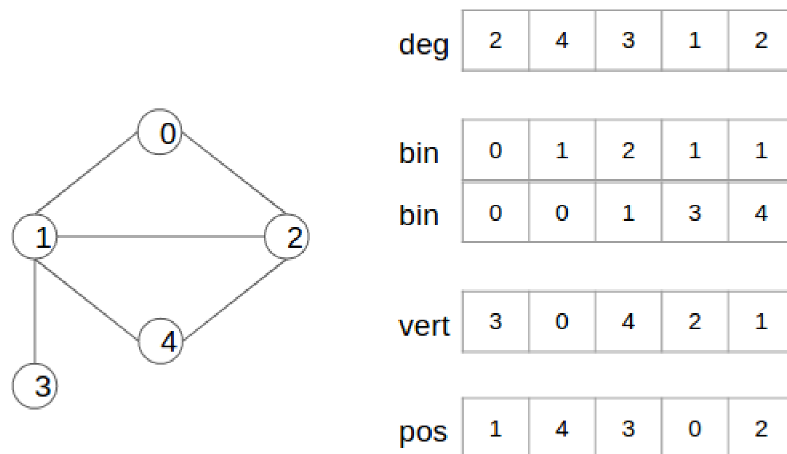
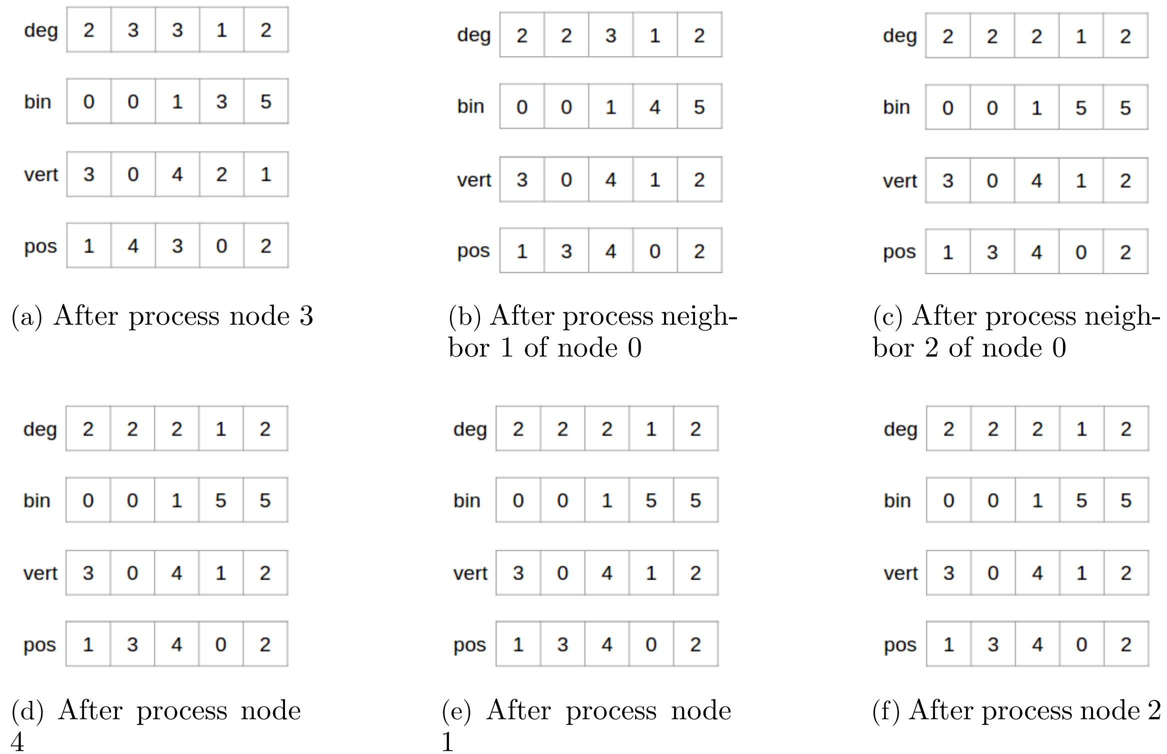


Figure 3.2: BZ algorithm applied in a simple graph

The main process of the BZ algorithm, which we call *Core* procedure, starts from the first node in array *vert* and goes to the end. For the node v of the smallest degree in *vert*, the algorithm decrements the degree of each neighbor u of v if the degree of u is larger than v. Then it moves node u from the current block to the left block, which

can be operated in constant time by swapping u and the first node in the same block. Since positions in $vert$ are changed, it also needs to swap positions in pos. Finally, it increments the start index of the current block in bin, and decrements the degree of u in deg. After processing all nodes in $vert$, deg contains the coreness of each node. To illustrate the above, Figure 3.2 shows an example that applies the BZ algorithm in a simple toy graph.



(a) After process node 3

(b) After process neighbor 1 of node 0

(c) After process neighbor 2 of node 0

(d) After process node 4

(e) After process node 1

(f) After process node 2

Figure 3.3: Results of BZ algorithm Core procedure

Figure 3.2 presents a simple toy graph with 5 nodes, and 4 arrays initialized by BZ algorithm. Index of all arrays starts from 0. As described in the array $vert$, nodes are sorted ascending by their degrees as 3, 0, 4, 2, and 1. Array $pos[i]$ records position of node i in $vert$. The $Core$ procedure starts from the first node in $vert$, 3. A neighbor of 3 is 1, and the degree of 1 is larger than the degree of 3. The first node w which has the same degree as 1 in $vert$ is still 1, so no swap between w and 1. But we still need to increment $bin[deg[1]]$, and decrement $deg[1]$. Results after processing node 3 are shown in Figure 3.3a. Next node is 0 which has two neighbors 1 and 2. For neighbor 1, the degree of 1 is larger than 0, and the first node that has the same degree as 1 is 2 in $vert$, then we swap 1 and 2 in both $vert$ and pos. Moreover, increment

the $bin[deg[1]]$ and decrement the $deg[1]$. For neighbor 2 whose degree is 3, the first node which has the same degree in *vert* is itself, so no swap required. After operate $bin[deg[2]]$ and $deg[2]$, results are displayed in Figure 3.3b and Figure 3.3c. When processing node 4, 1, and 2, we found that all neighbors of these nodes have the same degrees as themselves, therefore, there is no other operation required. Figure 3.3d to 3.3f display results after processing node 4, 1, and 2. According to the output of BZ algorithm, the coreness of node 0, 1, 2, 3, and 4 is 2, 2, 2, 1, and 2, respectively.

The total time complexity of BZ algorithm is shown to be $O(max(m,n))$ [4]. Since $m \geq n-1$ in a connected graph, the time complexity of BZ algorithm is actually $O(m)$ which makes it a very efficient algorithm for k-core decomposition. In this thesis, we use BZ algorithm for coreness computation which is the basis of CCI community detection.

3.3 WebGraph

With the development of World Wide Web, graphs which represent web pages and links become extremely huge. In order to manipulate such very large graphs simply, Paolo and Sebastiano [6, 5] provide a highly efficient graph compression framework, WebGraph.

The WebGraph framework is a suite of codes, algorithms, and tools [6]. The codes which are suitable for storing Web graphs provide a high compression ratio. The algorithm uses lazy techniques that delay decompression until it is actually necessary when accessing a compressed graph. The WebGraph framework is completely documented and packaged a set of jar files. All the information and tools are freely available from the WebGraph home page (http://webgraph.di.unimi.it).

In this thesis, we implemented BZ algorithm and our new algorithms using the WebGraph API for random access. Two classes of WebGraph are used in our programs:

1. *it.unimi.dsi.webgraph.ImmutableGraph*
2. *it.unimi.dsi.webgraph.NodeIterator*

Class *ImmutableGraph* is a simple class representing an immutable graph which is a graph that is computed once for all, then stored and accessed repeatedly. Function *ImmutableGraph.load()* is called to load graphs for random access. Then we call *ImmutableGraph.successorArray(u)* and *ImmutableGraph.outdegree(v)* to get neighbors

of node u and the outdegree of node v, respectively. An object of class *NodeIterator* is created by *ImmutableGraph.nodeIterator()*, which returns a node iterator for scanning the graph sequentially, starting from the first node.

Chapter 4

Initial Algorithms

In this chapter, two problems (P1 and P2) which we set to solve in this thesis are precisely defined at first. Then we comprehensively describe two algorithms from [24], $C0$ and $NC0$, for computing CCI and non-containing CCI communities, respectively, i.e. $C0$ solves problem P1, and $NC0$ solves P2.

4.1 Two problems

According to the previous chapter, we define two top-r CCI community problems (**P1** and **P2**):

1. Given an undirected graph G, and two positive integers k and r, how to compute the top-r CCI communities?
2. Given an undirected graph G, and two positive integers k and r, how to compute the top-r non-containing CCI communities?

4.2 DFS-based algorithms $C0$ and $NC0$

Depth-First-Search (DFS) is an algorithm for traversing (or searching) tree or graph data structures. The most common use of DFS algorithm is to find connected components in a graph. According to discussions in Section 3.1.2, given k and r, the CCI community and non-containing CCI community problem is to discover top-r (with the highest weights) maximally connected, k-core communities of graphs. In the following, we use the DFS-based algorithm to search CCI communities in a simple graph.

Recall that we use undirected graphs to represent networks. Consider an undirected graph $G = (V, E)$, where V is the set of vertices, and E is the set of edges. n and m are the number of vertices and edges, respectively. Furthermore, each node u in G has a weight w_u indicating the influence or importance of u. We assume a strict total order on weights. In case of ties, we use the lexicographical order of vertex ids to break the ties. The influence or importance of a CCI community is defined to be the lower-bound of its vertices' weights.

Since a CCI community actually is a k-core or a connected component of k-core in a graph, we denote the maximal k-core of G by $C_k(G)$, and CCI communities of G by $H_k(G)$. Similar to k-core decomposition, the DFS-based algorithm in [24] discovers CCI communities $H_k(G)$ by "peeling off" the maximal k-core $C_k(G)$. We call the DFS-based algorithm for Problem 1 as **C0** in this thesis, and the pseudocode of $C0$ is given in Algorithm 2.

Algorithm 2 Top-r CCI communities (C0)

Input: G, w, k, r

Output: $H_{k,1}$, ..., $H_{k,r}$

1: $C_k(G) \leftarrow$ compute the maximal k-core of G
2: $cache \leftarrow \emptyset$
3: $i \leftarrow 1$
4: $C_{k,i} \leftarrow C_k(G)$
5: **while** $C_{k,i} \neq \emptyset$ **do**
6: Let v be the minimum-weight vertex in $C_{k,i}$
7: $H_{k,i} \leftarrow MCC(C_{k,i}, v)$
8: **if** $cache.size() = r$ **then**
9: $cache.deleteFirst()$
10: $cache.addLast(H_{k,i})$
11: $C_{k,i+1} \leftarrow RDelete(C_{k,i}, v)$
12: $i \leftarrow i + 1$
13:
14: Output $cache$

Algorithm 3 RDelete

1: **procedure** RDELETE(C, v)
2: **for all** $u \in N_C(v)$ **do**
3: Delete edge (u, v) from C
4: **if** $d_C(u) < k$ **then**
5: *RDelete*(C, u)
6:
7: Delete v from C

Algorithm 4 Search Maximally Connected Component (MCC)

1: **procedure** MCC(C, v)
2: $cc \leftarrow \emptyset$
3: *MCC-DFS*(C, v, cc)
4: **return** cc
5:
6: **procedure** MCC-DFS(C, v, cc)
7: $cc.add(v)$
8: **for all** $u \in N_C(v)$ **do**
9: **if** $u \notin cc$ **then**
10: *MCC-DFS*(C, u, cc)

To illustrate the algorithm $C0$, Figure 4.1 displays procedures that process a toy graph G to find CCI communities. In order to make it simple, we set weights of vertices to be their ids. Gray nodes and edges indicate they are removed during procedures in Figure 4.1. Since there are multiple iterations to peel off the whole graph, we denote $C_{k,i}$ as a k-core subgraph before the *ith* iteration, and $H_{k,i}$ as a CCI community found in $C_{k,i}$ in the *ith* iteration.

Given $k = 2$ and $r = 5$, the goal is to find top-5 (with the highest influences) CCI communities in the maximal 2-core subgraph of G. In $C0$, *cache* is a list with size r used to store found CCI communities, and i mentions the number of iteration.

We compute the maximal 2-core $C_2(G)$ of G at first, then we set $C_{2,1} = C_2(G)$ because all vertices have corenesses that larger than 2. $C_{2,1}$ is the 2-core subgraph before the first iteration, and v_1 is the minimum weight vertex in $C_{2,1}$. Through the procedure *Search Maximally Connected Component (MCC)* which is given in

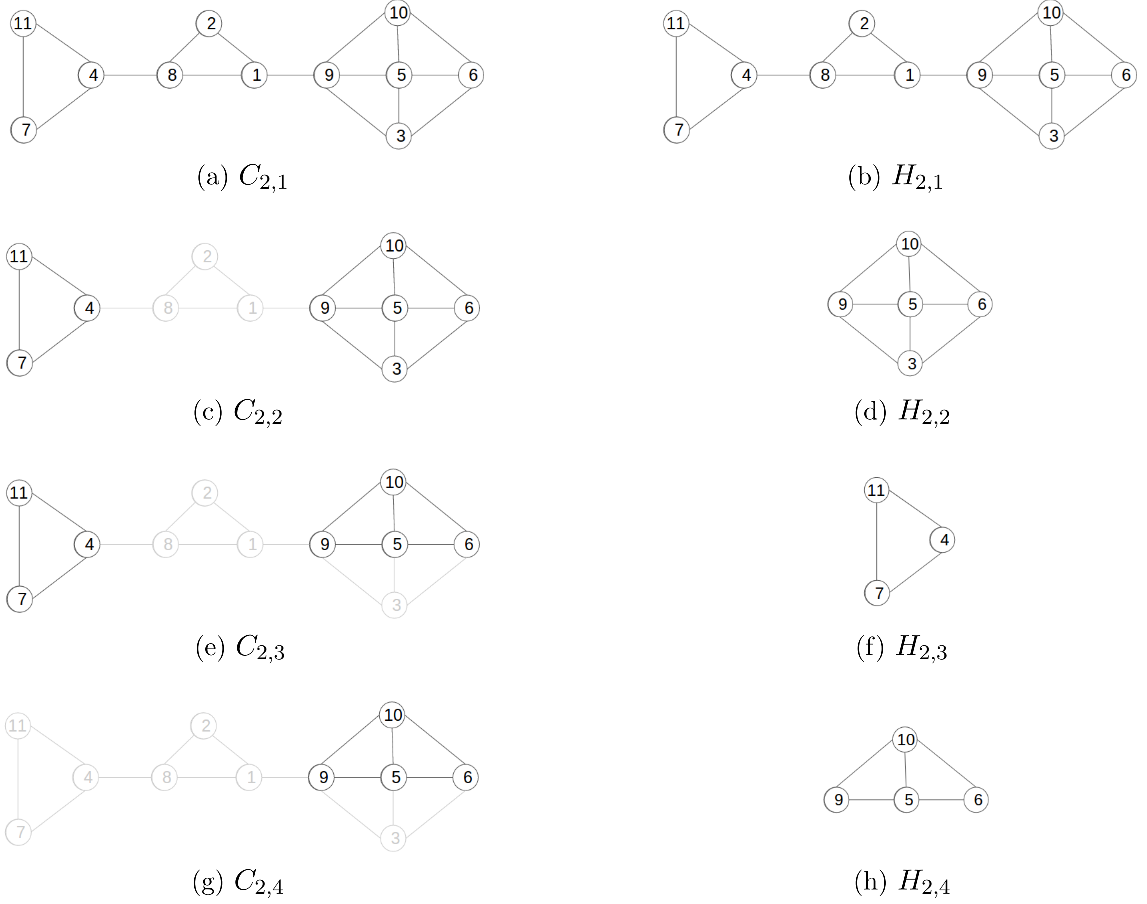


Figure 4.1: $C_{k,i}$ and $H_{k,i}$, for $k = 2$ and $i \in [1, 4]$. The original graph in (a) are all in black, grayed out vertices and edges are deleted. Weights are the vertex id's.

Algorithm 4, $H_{2,1}$ is the MCC of $C_{2,1}$ containing v_1. Precisely, $H_{2,1}$ is the first CCI community with influence "1" we found. Since *cache* is empty, $H_{2,1}$ is saved directly. Next, we call procedure *RDelete* to remove v_1, and recursively remove neighbors of v_1 whose degrees in $C_{2,1}$ are lower than 2. In Figure 4.1c, we can see that vertex 2 and 8 are deleted when peeling off vertex 1. After recursively "peeling off" v_1, we define $C_{2,2}$ as remains of $C_{2,1}$, and increment i for next iteration. We repeat this process until $C_{2,i}$ becomes empty for certain i. Finally, we get four CCI communities: $H_{2,1}$ with influence "1" in Figure 4.1b, $H_{2,2}$ with influence "3" in Figure 4.1d, $H_{2,3}$ with influence "4" in Figure 4.1f, and $H_{2,4}$ with influence "5" in Figure 4.1h. $H_{2,4}$ is the top-1 CCI community with the highest importance.

Since DFS procedure in *RDelete* recursively deletes all nodes whose degrees are less than k, it is clear that CCI communities $H_{k,i}$, MCC of $C_{k,i}$, are all k-core.

In the example Figure 4.1, there are four CCI communities: $H_{2,1} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$, $H_{2,2} = \{3, 5, 6, 9, 10\}$, $H_{2,3} = \{4, 7, 11\}$, and $H_{2,4} = \{5, 6, 9, 10\}$. $H_{2,3}$ and $H_{2,4}$ do not contain any other communities, thus they are non-containing CCI communities. To find non-containing CCI communities, one step should be added after procedure *RDelete* to check whether all vertices in $H_{k,i}$ are deleted or not [24]. If yes, then it is a non-containing CCI communities, and we store it in *cache*. We call this algorithm for Problem 2 as **NC0**, and it is given in Algorithm 5.

Algorithm 5 Top-r non-containing CCI communities (NC0)

Input: G, w, k, r

Output: $H_{k,1}, \ldots, H_{k,r}$

1: $C_k(G) \leftarrow$ compute the maximal k-core of G
2: $cache \leftarrow \emptyset$
3: $i \leftarrow 1$
4: $C_{k,i} \leftarrow C_k(G)$
5: **while** $C_{k,i} \neq \emptyset$ **do**
6:     Let v be the minimum-weight vertex in $C_{k,i}$
7:     $flag \leftarrow false$
8:     $H_{k,i} \leftarrow MCC(C_{k,i}, v)$
9:     $C_{k,i+1} \leftarrow RDelete(C_{k,i}, v)$
10:     $flag \leftarrow$ Check whether vertices in $H_{k,i}$ are all deleted in $C_{k,i+1}$
11:     **if** $flag = true$ **then**
12:         **if** $cache.size() = r$ **then**
13:             $cache.deleteFirst()$
14:         $cache.addLast(H_{k,i})$
15:     $i \leftarrow i + 1$
16:
17: Output *cache*

In next chapter, we will discuss bottlenecks of $C0$ and $NC0$, the time complexity and space complexity, then provide multiple new algorithms to more efficiently solve two problems (P1 and P2) using fewer resources on a consumer-grade machine.

Chapter 5

Algorithms for P1 and P2

In the preceding chapter, algorithm $C0$ and $NC0$ are introduced for searching top-r CCI communities and non-containing CCI communities respectively. Since $C0$ and $NC0$ follow the same logic, they have the same time complexity and space complexity. We will only thoroughly analyze the algorithm $C0$ in below.

There are two bottlenecks in $C0$. The first one is the large times of MCC computations, which compute from the first minimum-weight vertex to the end. The time complexity of computing an MCC is $O(m)$, but computing for each minimum-weight vertex costs time $O(m \cdot n)$. The time complexity for procedure *RDelete* is $O(m)$ which can be absorbed by MCC computation. Therefore, the time complexity of algorithm $C0$ is $O(m \cdot n)$, which is impractical for big graphs running on a consumer-grade machine.

The second bottleneck is memory usage. Since algorithm $C0$ loads a whole graph in memory, and stores vertices of top-r communities, the space complexity is $O(m + n \cdot r)$. When r is small, we assume that $n \cdot r$ can be absorbed by m. Nevertheless, when r is a large number, $n \cdot r$ becomes much bigger than m which may exceed the available memory in a consumer-grade machine.

In order to break the bottlenecks, we proposed two algorithms for Problem 1 and two algorithms for Problem 2 in this chapter. Our algorithms are inspired from the algorithm $C0$ and $NC0$ described in Section 4.1, as well as a k-core decomposition algorithm, the BZ algorithm, shown in Section 3.2. They outperform $C0$ and $NC0$ by orders of magnitude.

In Section 5.1, we describe an algorithm $C1$ for Problem 1, then a faster algorithm $C2$ for the same problem is provided in Section 5.2. In Section 5.3, an algorithm $NC1$ for Problem 2 is introduced, then an improved algorithm $NC2$ for the same problem

is presented in Section 5.4.

5.1 Algorithm C1 for Problem 1

There are two reasons that MCC computations are quite expensive procedures. First, the *RDelete* procedure does not remove a few vertices at most of the early iterations. We observed such situations in some real-world graph datasets. Second, because of the first reason, finding MCC in early $C_{k,i}$ takes a lot of time since they are quite big in size.

However, *RDelete*, a recursive deletion procedure, is inexpensive to compute, especially for early iterations. It deletes the minimum-weight vertex v from $C_{k,i}$, then recursively deletes all v's neighbors whose degrees are less than k, until there are no more vertices with degrees less than k. The total time of *RDelete* costs $O(m)$ which is no more than traversing a whole graph. Thus, in order to reduce MCC computations, we observe that we only need to compute MCC for the last r iterations because only those iterations produce results, top-r CCI communities, we want.

How to get the total number of iterations beforehand is another question we pose. And we notice that running *RDelete* procedure two times is able to solve this problem. In the first run, we can calculate the total number of iterations, then compute MCCs only in the second run. A new algorithm **C1** based on the above logic is described in Algorithm 6.

Input of $C1$ includes a graph G, weights for each vertex, k and r. At first, we call BZ algorithm for computing coreness for each node in graph G, then remove vertices whose coreness are less than k, and remains of the graph forms the original $C_k(G)$. C, a duplication of $C_k(G)$, represents $C_{k,i}$ after *RDelete* in each iteration. Counter i is used to calculate how many iterations are required. In the first while loop (lines 4-7), it picks the minimum-weight vertex v from the current C, then recursively remove v and its neighbors by calling *RDelete*. Each iteration is recorded by incrementing counter i. After the first run, we recover C using the original $C_k(G)$. In the second loop (lines 11-17), another counter j is used to accumulate the times of iterations. Only when $j > i - r$, which means the last r iterations, we call the MCC procedure and output results directly. Procedure *RDelete* is called the second time to recursively delete vertices from the graph. Totally, we run MCC for only r times.

Based on previous discussions, we can state that

Algorithm 6 Top-r CCI communities (C1)

Input: G, w, k, r
Output: $H_{k,1}$, ..., $H_{k,r}$

1: $C_k(G) \leftarrow$ compute the maximal k-core of G
2: $i \leftarrow 1$
3: $C \leftarrow C_k(G)$
4: **while** $C \neq \emptyset$ **do**
5: Let v be the minimum-weight vertex in C
6: $RDelete(C,v)$
7: $i \leftarrow i + 1$
8:
9: $C \leftarrow C_k(G)$
10: $j \leftarrow 1$
11: **while** $C \neq \emptyset$ **do**
12: Let v be the minimum-weight vertex in C
13: **if** $j > i - r$ **then**
14: $H \leftarrow MCC(C,v)$
15: Output H
16: $RDelete(C,v)$
17: $j \leftarrow j + 1$

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

The time complexity of $C1$ is $O(m \cdot r)$ because MCC computation only runs r times, and time of calling two *RDelete* procedures is absorbed by MCC computation. In fact, since the last r MCC computations operate on quite small subgraphs which produced after $i - r$ "peeling off" procedures, $O(m \cdot r)$ is much smaller than $O(m \cdot n)$, the time complexity of $C0$.

For space complexity, we do not store r communities anymore, because only last r communities are computed, and output directly. Therefore, the space complexity of $C1$ is $O(m)$.

In next section, we will provide a faster algorithm which reduces the time complexity by a factor of (about) 2 compared with $C1$.

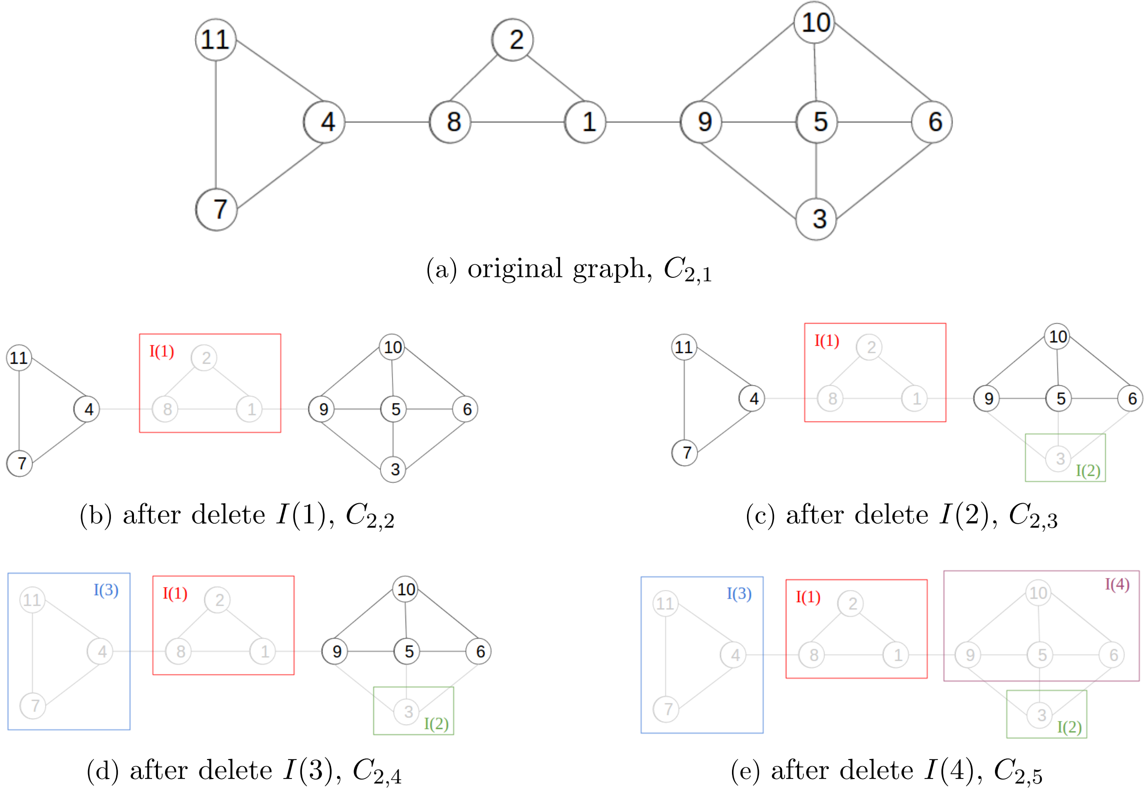
5.2 Algorithm C2 for Problem 1

Compared with $C0$, $C1$ greatly decreases the time complexity and space complexity. However, we still run two while loops on a whole graph in $C1$. How to avoid the

second while loop is the next question we want to figure out.

During our experiments, we found that vertices removed in *RDelete* can be stored in certain data structures to avoid the second ”peeling off” procedure. The data structure we used here is a hash-based structure, which we call *iteration-delete-history* and denote by I. In Java codes, we use data structure *ArrayDeque* to represent I.

Each element $I(i)$ in I is a list of deleted vertices in the corresponding iteration i. Take Figure 5.1 as an example for illustration.



(a) original graph, $C_{2,1}$

(b) after delete $I(1)$, $C_{2,2}$

(c) after delete $I(2)$, $C_{2,3}$

(d) after delete $I(3)$, $C_{2,4}$

(e) after delete $I(4)$, $C_{2,5}$

Figure 5.1: $C_{k,i}$ for $k = 2$ and $i \in [1, 5]$. The original graph in (a) are all in black, grayed out vertices and edges are deleted.

Given $k = 2$ and weights equals to nodes' id, the original graph G is shown in Figure 5.1a. Figure 5.1b to 5.1e are 2-core subgraphs of G after recursively delete nodes with the minimum weight and its neighbors. There are 4 iterations to peel off the whole graph, nodes deleted in these iterations are: $I(1) = \{1, 2, 8\}$, $I(2) = \{3\}$, $I(3) = \{4, 7, 11\}$, and $I(4) = \{5, 6, 9, 10\}$.

Algorithm **C2** based on ideas of such hash-based structure is described in Algorithm 7.

Algorithm 7 Top-r CCI communities (C2)

Input: G, w, k, r

Output: $H_{k,r}$, ..., $H_{k,1}$

1: $C_k(G) \leftarrow$ compute the maximal k-core of G
2: $C \leftarrow C_k(G)$
3: $i \leftarrow 1$
4: $I \leftarrow \emptyset$
5: **while** $C \neq \emptyset$ **do**
6: Let v be the minimum-weight vertex in C
7: $I(i) \leftarrow \emptyset$
8: *RDelete2*(C,v,I,i)
9: $i \leftarrow i + 1$
10:
11: $alive \leftarrow \mathbf{0}$
12: **for** $j = i$ **downto** $i - r + 1$ **do**
13: **for all** $v \in I(j)$ **do**
14: $alive[v] \leftarrow 1$
15: $v \leftarrow I(j).first()$
16: $H \leftarrow$ *MCC2*$(C_k(G), v, alive)$
17: Output H

Algorithm 8 RDelete2

1: **procedure** RDelete2(C, v, I, i)
2: **for all** $u \in N_C(v)$ **do**
3: Delete edge (u, v) from C
4: **if** $d_C(u) < k$ **then**
5: *RDelete2*(C, u, I, i)
6:
7: Delete v from C
8: $I(i).add(v)$

Algorithm 9 MCC with alive array (MCC2)

```
1: procedure MCC2(C_k(G), v, alive)
2:     cc ← ∅
3:     MCC-DFS2(C_k(G), v, alive, cc)
4:     return cc
5:
6: procedure MCC-DFS2(C_k(G), v, alive, cc)
7:     cc.add(v)
8:     for all u ∈ N_{C_k(G)}(v) do
9:         if alive[u] = true  &  u ∉ cc then
10:            MCC-DFS2(C_k(G), u, alive, cc)
```

Same as $C1$, input of $C2$ includes a graph G, weights for each vertex, k and r. The BZ algorithm is used to compute core number for each vertex in G, and $C_k(G)$ represents the maximal k-core subgraph of G. C means remained subgraphs after $RDelete2$ procedures, and counter i calculates how many iterations are required to delete the whole graph. Furthermore, a data structure saved lists of deleted nodes, I, is initialized to empty. In the first while loop, we choose the minimum-weight vertex in C, and initialize an empty list of the corresponding iteration in I, which we call $I(i)$. Then $RDelete2$, a modified $RDelete$ procedure, which takes two extra parameters, I and i, is called. In $RDelete2$, we add all deleted vertices during recursively deletion to list $I(i)$.

A flat array "alive" is used to represent deletion states of all nodes in G, it is initialized to all 0 which means all nodes are deleted. A for loop goes down starting from the total iteration number, i, to $i-r+1$, so there are r iterations in the for loop. In each iteration, we poll the last list of deleted nodes from I, and set these vertices to 1 in "alive", which means they have not been deleted. Then $MCC2$ is called based on the maximal k-core $C_k(G)$ of G, and the consulting array $alive$. Only those vertices have not been removed are considering for computing maximally connected components. Once we find an MCC, output it directly.

Compared with $C1$, results of $C2$ are in reverse order, i.e. $H_{k,r}, ..., H_{k,1}$. Based on previous discussions, we can state that

Theorem 2. Algorithm C2 correctly computes all the top-r CCI communities of a given graph G.

The time complexity of $C2$ is the same as $C1$, $O(m \cdot r)$. However, in terms of constant factors, $C2$ is almost twice faster than $C1$. Because we only run $RDelete2$ procedure one time in $C2$, and array operations, such as insertion and poll, only take constant time.

For space complexity, structure I takes $O(n)$ space, and the graph takes $O(m)$ space. Since n is always much smaller than m in undirected graphs, $O(n)$ can be absorbed by $O(m)$. Therefore, the space complexity of Algorithm $C2$ is $O(m)$.

5.3 Algorithm NC1 for Problem 2

Similar to $C0$, $NC0$ also has bottlenecks in time complexity and memory usage. The time complexity of $NC0$ is $O(m \cdot n)$, and the space complexity is $O(m + n \cdot r)$. It is clear that $NC0$ is impractical for big graphs running on a consumer-grade machine as well.

As described in $NC0$, the key to finding non-containing CCI communities is to check whether all vertices in MCC of vertex v are all deleted after recursively deleting vertex v and its neighbors in the $RDelete$ procedure. Then, we posed a question that how to easily get lists of deleted nodes in advance. Based on previous experience, we use the same iteration-delete-history structure as $C2$ to store deleted nodes in each iteration.

In addition, how to easily and accurately check whether vertices in MCC are equal to vertices deleted in the $RDelete$ procedure is another question we posed. Since both procedures $RDelete$ and MCC use the depth-first-search algorithm based on the same vertex in a graph, the minimum-weight vertex, to find connected components, their results should be the same. However, the $RDelete$ procedure takes vertices' degrees as a filter condition, then a smaller connected component is found compared with the connected component found in procedure MCC. Thus, we can directly compare the number of vertices in the MCC and the list of deleted nodes, instead of comparing each node, to save time.

Based on above discussions, algorithm **NC1** is given in Algorithm 10.

Input of $NC1$ includes a graph G, weights for each vertex, k and r. The BZ algorithm is used to compute core number for each vertex in G, and $C_k(G)$ represents the maximal k-core subgraph of G. Same as $C1$ and $C2$, C represents remained subgraphs after $RDelete2$ procedure, and a counter i calculates the number of total iterations required. A hash-based structure I, which saved lists of deleted nodes, is

Algorithm 10 Top-r non-containing CCI communities (NC1)

Input: G, w, k, r
Output: $H_{k,r}$, ..., $H_{k,1}$
1: $C_k(G) \leftarrow$ compute the maximal k-core of G
2: $C \leftarrow C_k(G)$
3: $i \leftarrow 1$
4: $I \leftarrow \emptyset$
5: **while** $C \neq \emptyset$ **do**
6: Let v be the minimum-weight vertex in C
7: $I(i) \leftarrow \emptyset$
8: *RDelete2*(C,v,I,i)
9: $i \leftarrow i + 1$
10:
11: $j \leftarrow 0$
12: $alive \leftarrow \mathbf{0}$
13: **while** $I \neq \emptyset$ & $j < r$ **do**
14: $I_k \leftarrow I.poll()$
15: **for all** $v \in I_k$ **do**
16: $alive[v] \leftarrow 1$
17: $v \leftarrow I_k.first()$
18: $H \leftarrow MCC2(C_k(G), v, alive)$
19: **if** $H.size() = I_k.size()$ **then**
20: Output H
21: $j \leftarrow j + 1$

initialized to empty at first. In the first while loop, we choose the minimum-weight vertex in C, and initialize an empty list of the corresponding iteration i in I which called $I(i)$. Then in $RDelete2$, we add all deleted vertices to list $I(i)$ during recursion.

Another counter j is used to count how many non-containing CCI communities have been found. Same as $C2$, a flat array "alive", which represents deletion states of nodes, is initialized to all 0. e.g. $alive[v] = 0$ means node v is deleted. The second while loop takes $I \neq \emptyset$ and $j < r$ as conditions because it is possible that we are not able to find enough r non-containing CCI communities even go through the whole graph. In each iteration, we poll the last list of deleted nodes from I, then assign to I_k. Vertices in I_k are set to 1 in array *alive*. Then $MCC2$ is called based on the maximal k-core $C_k(G)$ of graph G, and the consulting array *alive*. Only those vertices have not been deleted, e.g. $alive[v] = 1$, are considering for computing the maximally connected component. When the number of vertices in the MCC equals to the number of vertices in I_k, all nodes in the current MCC of minimum-weight vertex

v have been deleted after $RDelete2$ procedure started with v. Then the current MCC is a non-containing CCI communities, and we output it directly.

Since MCC computation starts from the last $C_{k,i}(G)$, results of $NC1$ are in reverse order, e.g. $H_{k,r}, ..., H_{k,1}$. Based on previous discussions, we can state that

Theorem 3. Algorithm NC1 correctly computes all the top-r non-containing CCI communities of a given graph G.

The time complexity of $NC1$ highly depends on the input graph. Numbers of non-containing CCI communities in the input graph is a factor which influences times of running MCC computation. Because the second loop in $NC1$ keeps computing MCC until it finds r non-containing CCI communities or the whole graph has been searched. Therefore, the best time complexity of $NC1$ is $O(m \cdot r)$, while the worst time complexity is $O(m \cdot n)$. Commonly, we take the worst time complexity $O(m \cdot n)$ as the time complexity of $NC1$.

For space complexity, $NC1$ takes the same space as $C2$, $O(m)$. Although structure I and graph G take $O(n)$ and $O(m)$ space respectively, $O(n)$ can be absorbed by $O(m)$ because n is much smaller than m in undirected graphs.

5.4 Algorithm NC2 for Problem 2

Although $NC1$ takes less space complexity than $NC0$, it still has the same time complexity as $NC0$ which is impractical for big graphs running on a consumer-grade machine.

When we check deleted nodes in structure I, we observe that all information need for non-containing CCI communities has been stored in I in fact. Therefore, we propose an idea which use I to find non-containing CCI communities instead of the MCC procedure.

The number of alive neighbors of a vertex v is a condition which can check whether there is a non-containing CCI community. Given a vertex v, we define the current degree is the number of alive neighbors of v. A flat array d is used to save current degrees of nodes. When a node v is alive, $d(v)$ stores the current degree of v. When a node v is deleted, $d(v)$ is not updated anymore and stores the degree when deletion happens.

In some iteration i, when each node v in $I(i)$ has current degree equals 0, $d(v) = 0$, then all neighbors of vertex v have been removed. Nodes in this $I(i)$ form a

last standing community in a community containment chain, therefore it is a non-containing CCI community.

Based on previous discussions, we can state that if for each $v \in I(i)$, $d(v) = 0$, then $I(i)$ is a non-containing CCI community.

Take Figure 5.1 as an example for illustration. There are 4 iterations to peel off the whole graph, nodes deleted in these iterations are: $I(1) = \{1, 2, 8\}$, $I(2) = \{3\}$, $I(3) = \{4, 7, 11\}$, and $I(4) = \{5, 6, 9, 10\}$. Since node 4 and 9 still exist in $C_{2,2}$, $d(1)$ and $d(8)$ are not equal to 0. $I(1)$ is not a non-containing CCI community. In $C_{2,3}$, which after deleting $I(2)$, $d(3) = 3$ because node 5, 6 and 9 exist, $I(2)$ is not a non-containing CCI community either. In $C_{2,4}$, node 4, 7 and 11 are deleted in $I(3)$, and all of their neighbors are deleted as well. $I(3)$ is a non-containing CCI community. Then deleting $I(4)$, $C_{2,5}$ becomes a empty graph. Since degrees of all deleted nodes in $I(4)$ are 0, $I(4)$ is also a non-containing CCI community.

Our algorithm **NC2**, which eliminates MCC computation, is given in Algorithm 11.

Input of $NC2$ includes a graph G, weights for each vertex, k and r. At first, we call BZ algorithm to compute coreness for each node in graph G, then remove vertices whose core numbers are less than k, and remains of the graph forms the original maximal k-core $C_k(G)$. C, a duplication of $C_k(G)$, represents $C_{k,i}$ after $RDelete3$ procedure in each iteration i.

For all vertices in C, we calculate the current degree of each node and store them in an array d. In while loop, the minimum-weight vertex is chosen and passed to $RDelete3$ procedure. In $RDelete3$, the array d of current degrees is updated during recursive deletion, and deleted nodes are added to the $I(i)$. There is a flag $isNC$ which used to check whether current degrees of all nodes in a $I(i)$ are 0, the default value of $isNC$ is true. If there is any node has current degree greater than 0, $isNC$ is set to false. When $isNC$ is true, we keep $I(i)$, and increment i before jump to next iteration. But when flag $isNC$ is false, which means the corresponding $I(i)$ is not a non-containing CCI community, we set current $I(i)$ to empty, then jump to next iteration without i increment.

After the while loop, all non-containing CCI communities have been saved in I, and a for loop is used to output top-r results. But the total number of non-containing CCI communities may not larger than or equal to r sometimes, a conditional break is added to check whether I is empty. The for loop goes from 1 up to r, we check whether current I is empty at first, then poll the last element in I, and output it

Algorithm 11 Top-r non-containing CCI communities (NC2)

Input: G, w, k, r
Output: $H_{k,r}$, $\ldots$, $H_{k,1}$

```
1:  C_k(G) ← compute the maximal k-core of G
2:  C ← C_k(G)
3:  for all vertex v of C do
4:      d[v] = d_C(v)
5:
6:  i ← 1
7:  I ← ∅
8:  while C ≠ ∅ do
9:      Let v be the minimum-weight vertex in C
10:     I(i) ← ∅
11:     RDelete3(C, v, I, i)
12:     isNC ← true
13:     for all v ∈ I(i) do
14:         if d[v] > 0 then
15:             isNC ← false
16:     if isNC = false then
17:         I(i) ← ∅
18:         continue
19:     i ← i + 1
20:
21: for j = 1  upto  r do
22:     if I is empty then
23:         break
24:     H ← I.pollLast()
25:     Output H
```

directly.

Results of $NC2$ are in reverse order, i.e. $H_{k,r}, ..., H_{k,1}$. Based on the above reasoning, we can state that

Theorem 4. Algorithm NC2 correctly computes all the top-r non-containing CCI communities of a given graph G.

Since r is much smaller than m, time of r iterations in for loop can be regarded as a constant value. The time complexity of $NC2$ is $O(m)$ because it only iterates the graph one time, and no MCC computation anymore. Although $NC2$ uses data structure I which takes $O(n)$, it can be absorbed by the space, $O(m)$, taken by graph G. Therefore, the space complexity of $NC2$ is $O(m)$ too.

Algorithm 12 RDelete3

```
1: procedure RDELETE3(C, v, I, i)
2:     Mark v
3:     for all u ∈ N_C(v) do
4:         d[u] ← d[u] − 1;
5:         if u is not marked & d[u] < k then
6:             RDelete3(C, u, I, i)
7:
8:     Delete v from C
9:     I(i).add(v)
```

5.5 Conclusion

In this chapter, we introduce two new algorithms for Problem 1 and two new algorithms for Problem 2. Given an undirected graph $G = (V, E)$ where $n = |V|$ and $m = |E|$, k, and r, details of algorithms are follow:

1. Algorithm $C1$ for Problem 1 decreases the times of maximally connected components computation. The time complexity of $C1$ is $O(m \cdot r)$, and the space complexity is $O(m)$.

2. Algorithm $C2$ for Problem 1 takes a hash-based structure to save deleted nodes. The time complexity of $C2$ is $O(m \cdot r)$, but $C2$ actually twice faster than $C1$. The space complexity of $C2$ is $O(m)$.

3. Algorithm $NC1$ for Problem 2 takes the same hash-based structure as $C2$. The best time complexity of $NC1$ is $O(m \cdot r)$, while the worst time complexity is $O(m \cdot n)$. The space complexity of $NC1$ is $O(m)$.

4. Algorithm $NC2$ for Problem 2 completely eliminates maximally connected components computation. The time complexity of $NC2$ is $O(m)$, and the space complexity is $O(m)$ too.

Chapter 6

Experimental Results

In this chapter, we performed experimental analysis by evaluating our four algorithms provided in the preceding chapter on several real-world graph datasets. There are two parts of experiments: first, we compared the performance of initial algorithms from [24], $C0$ and $NC0$, with our algorithms for Problem 1 and 2 in two small graph datasets; second, we conducted a broad testing of our four algorithms to find the best solutions for solving Problem 1 and 2 we proposed in Chapter 4.

Section 6.1 introduces multiple real-world graph datasets we used in tests. In section 6.2, details of equipment and codes implement are presented. Then we conducted the first part of experiments: compare the initial algorithms $C0$ and $NC0$ with our four new algorithms in Section 6.3. Section 6.4 compares algorithm $C1$ and $C2$ for the Problem 1. Finally, $NC1$ and $NC2$ for the Problem 2 are compared in Section 6.5.

6.1 Datasets

We perform the experiments on the following 6 graph datasets:

- Astro Physics collaboration network (AstroPh)
- Slashdot social network obtained in Novermber 2008 (Slashdot0811)
- Pokec online social network (Pokec)
- LiveJournal online social network (LiveJournal1)
- Results of web crawl on pages using .uk domain in 2002 (uk-2002)

- Results of web crawl on pages written in Arabic in 2005 (arabic-2005)

The first four datasets are available for download from the Standford Network Analysis Platform (http://snap.stanford.edu/data/index.html). The last two datasets, uk-2002 and arabic-2005, can be obtained from the Laboratory of Web Algorithmics (http://law.di.unimi.it/datasets.php). Properties of 6 datasets, namely the number of vertices and edges, the maximum degree and core number, and the average core number, are displayed in Table 6.1.

Dataset	n	m	$d_{\max}$	$k_{\max}$	k_{avg}
AstroPh	18,771	198,050	504	56	13
Slashdot0811	77,360	469,180	2539	54	6
Pokec	1,632,803	22,301,964	14,854	47	14
LiveJournal1	4,846,609	42,851,237	20,333	372	9
uk-2002	18,483,186	261,787,258	194,955	943	16
arabic-2005	22,743,881	553,903,073	575,628	3,247	28

Table 6.1: Properties of datasets ordered by m.

All datasets need a preprocessing. First, directed graphs are converted to undirected graphs by removing self-loops (e.g. (v, v)), adding inverse edges (u,v) for each edge (v,u) if it did not exist, and deleting duplicated edges. Second, we compress graphs of txt format to WebGraph format. Third, we assigned random weights (Java double type), ranging from 0 to 100, to the vertices of each graph.

Each graph contains four files, namely .graph, .offsets, .properties, and weight, after preprocessing. Our programs take these four files and two parameters, k and r, as input, and the timeout was set to 1 hour for each algorithm to run. Ranges of k and r are given in Table 6.2.

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

Table 6.2: Ranges of parameters k and r.

6.2 Equipment

All of our experiments are conducted on a consumer-grade laptop with 2.40 GHz Intel Core i7 (4 cores) CPU, and 16 GB RAM, running Ubuntu 16.04 LTS.

We implemented all the algorithms in Java, and use the Java version ”OpenJDK 1.8.0”. In experiments, we allow each Java program to use a maximum of 8 GB heap size and 1 GB stack size.

6.3 Testing initial algorithms

In the first part of our experiment, we compare the initial algorithm $C0$ with our counterparts, $C1$ and $C2$ for Problem 1. For Problem 2, we compare the initial algorithm $NC0$ with improved algorithms $NC1$ and $NC2$.

According to our analysis of $C0$ and $NC0$, they are not practical for big graphs. So we use two smallest datasets, AstroPh and Slashdot0811, for the first part of experiments.

Parameter r is set to 40, and k is ranging from 2, 4, 8, 16, to 32 when testing two problems.

6.3.1 Problem 1

Figure 6.1 and 6.2 show results of the $C0$, $C1$ and $C2$ comparison.

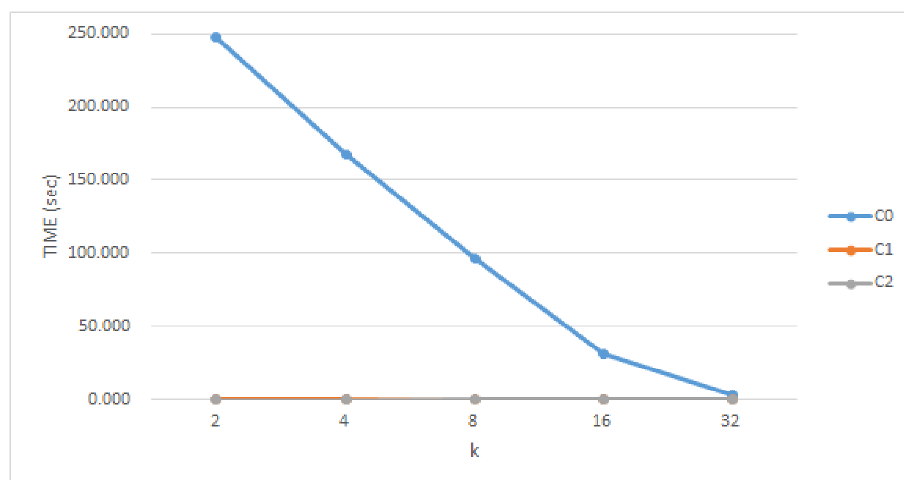


Figure 6.1: Problem 1 - AstroPh (r=40)

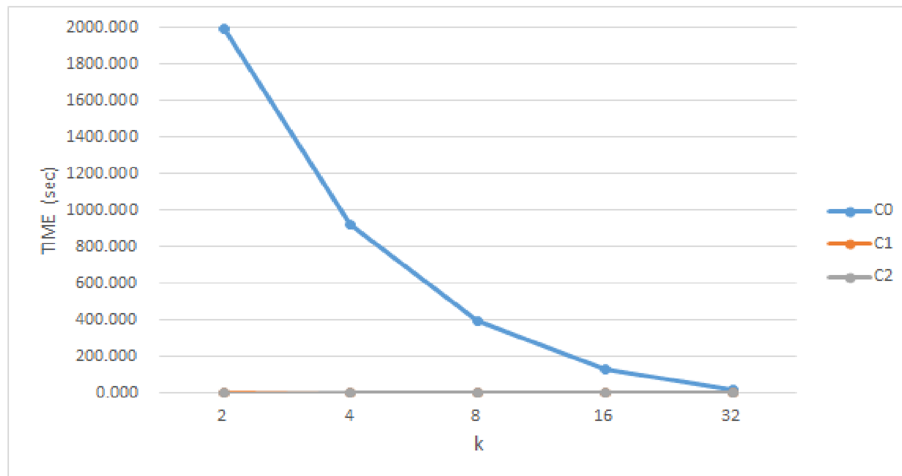


Figure 6.2: Problem 1 - Slashdot0811 (r=40)

Sizes of subgraphs of two datasets when k is varied are displayed in Table 6.3.

k	**AstroPh -** n	**Slashdot0811 -** n
2	17,439	47,760
4	13,741	29,026
8	9,425	17,200
16	5,664	8,977
32	1,926	3,188

Table 6.3: Sizes of subgraphs of AstroPh and Slashdot0811 in different k

Charts clearly show that $C1$ and $C2$ outperform $C0$ by orders of magnitude on both two datasets. Although subgraphs are relatively small when k is large, $C0$ still takes much longer time compared with $C1$ and $C2$, e.g. 16 seconds in $C0$, 0.275 seconds in $C1$, and 0.262 seconds in $C2$ when $k = 32$ on dataset Slashdot0811.

6.3.2 Problem 2

Figure 6.3 and 6.4 show results of the $NC0$, $NC1$ and $NC2$ comparison.

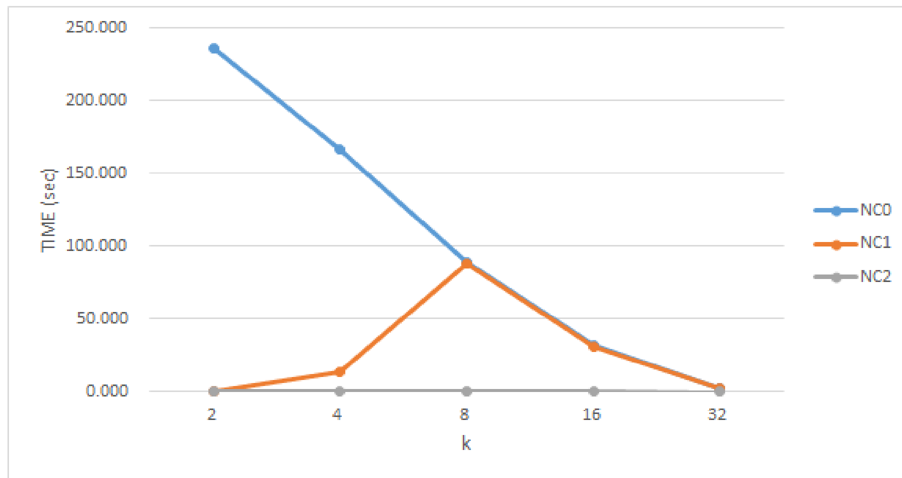


Figure 6.3: Problem 2 - AstroPh (r=40)

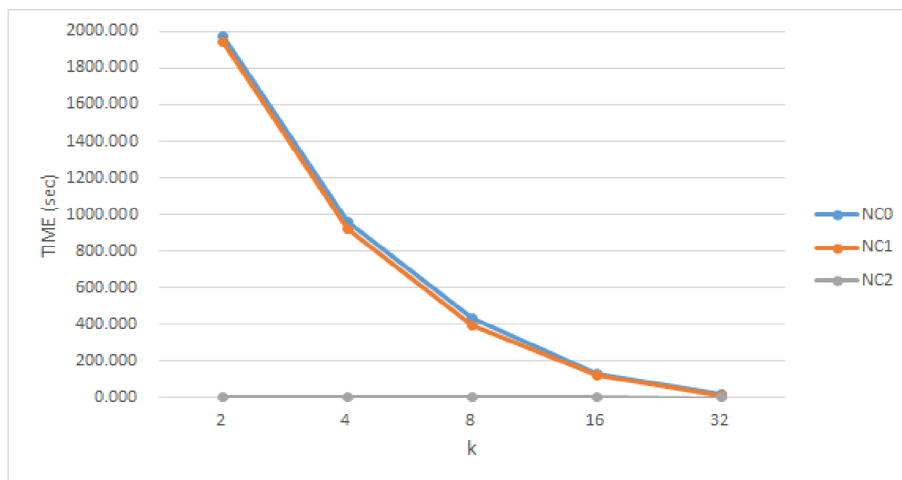


Figure 6.4: Problem 2 - Slashdot0811 (r=40)

Charts show that $NC2$ outperforms $NC0$ by orders of magnitude, and this is what we expected.

$NC1$ performs quite good in some k but has the same performance as $NC0$ most of time. Based on our analysis in Section 5.3, the performance of $NC1$ is not stable, it highly depends on the structure of input graphs. When there are not enough non-containing CCI communities, which is common on small graphs, $NC1$ goes through the whole graph and calculates MCC for each minimum-weight vertex. Then $NC1$ takes the worst time complexity $O(m \cdot n)$ which is the same as $NC0$.

According to results of running AstroPh dataset, we only found 4, 2, and 1, instead of 40, non-containing CCI communities when k equals 8, 16, and 32. It

implies that $NC1$ goes through the whole graph but still cannot find enough non-containing CCI communities. Same reason in dataset Slashdot0811, there are only 1 non-containing CCI communities when k equals 4, 8, 16, and 32, and 12 non-containing CCI communities when $k = 2$. Nevertheless, $NC1$ almost takes the same time as $NC2$ when $k = 2$ in AstroPh, which is expected because it found 40 non-containing CCI communities in the $k = 2$ subgraph.

6.3.3 Conclusion

Since $C0$ and $NC0$ are much slower than $C1$, $C2$, and $NC2$ even in small graphs, we eliminate $C0$ and $NC0$ from further testing on large-scale graphs.

For $NC1$, although its performance is not stable on small graphs with fewer non-containing CCI communities, it shows competitive results when graphs containing adequate non-containing CCI communities, such as $k = 2, 4$ in AstroPh. Based on our previous research, large-scale graphs commonly contain plenty of non-containing CCI communities, so we still test $NC1$ on further testing.

6.4 Testing for Problem 1

As described in Chapter 5, algorithms $C1$ and $C2$ are designed for Problem 1. Next, we present test results and analysis on datasets Pokec, LiveJournal1, uk-2002, and arabic-2005. Since AstroPh and Slashdot0811 are relatively small, we do not show the results of them.

Since there are two flexible parameters, k and r, our experiments are divided into two groups: first, we set r to a fixed value, and choose k from ranges mentioned in Table 6.2; second, fix k, and r is picked from ranges mentioned in Table 6.2.

6.4.1 Experiment 1 - fixed r

Figure 6.5, 6.6, 6.7, and 6.8 show results for computating top-r CCI communities when $r = 40$ and k is varied on four datasets.

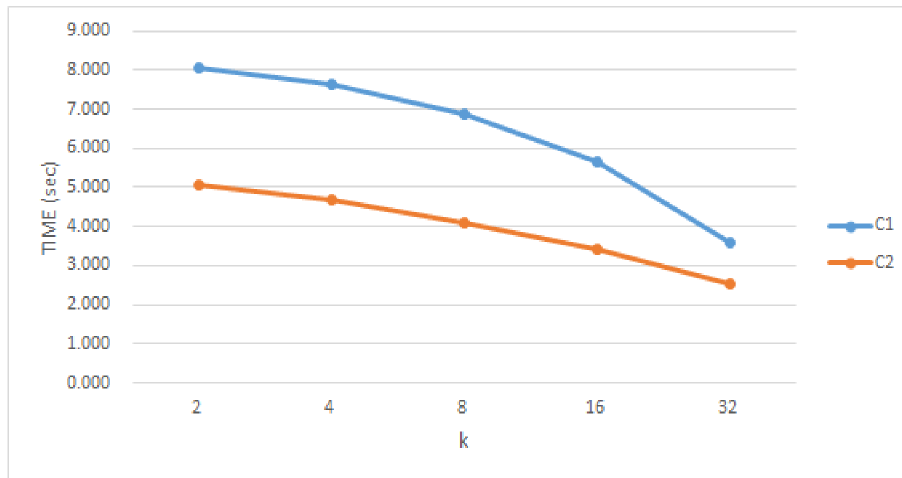


Figure 6.5: Problem 1 - Pokec (r=40)

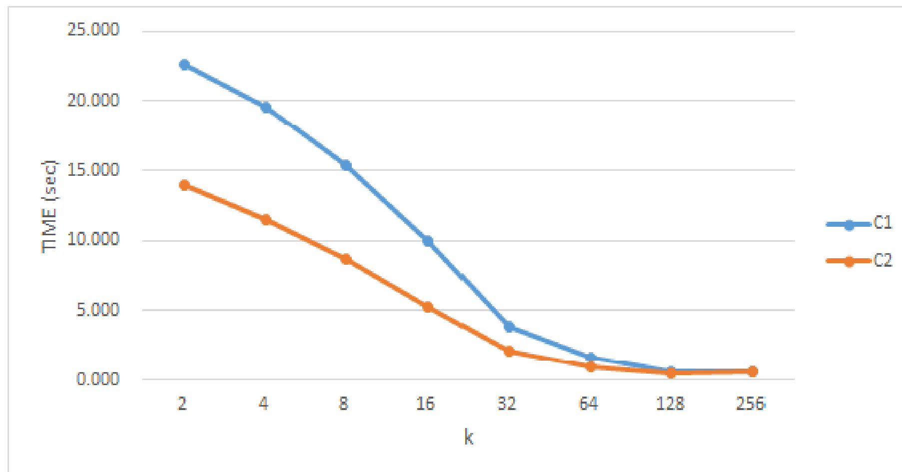


Figure 6.6: Problem 1 - LiveJournal1 (r=40)

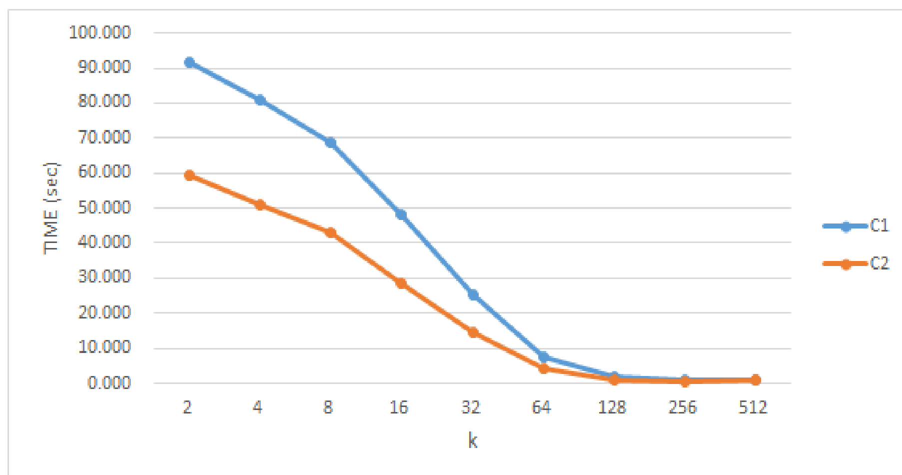


Figure 6.7: Problem 1 - uk-2002 (r=40)

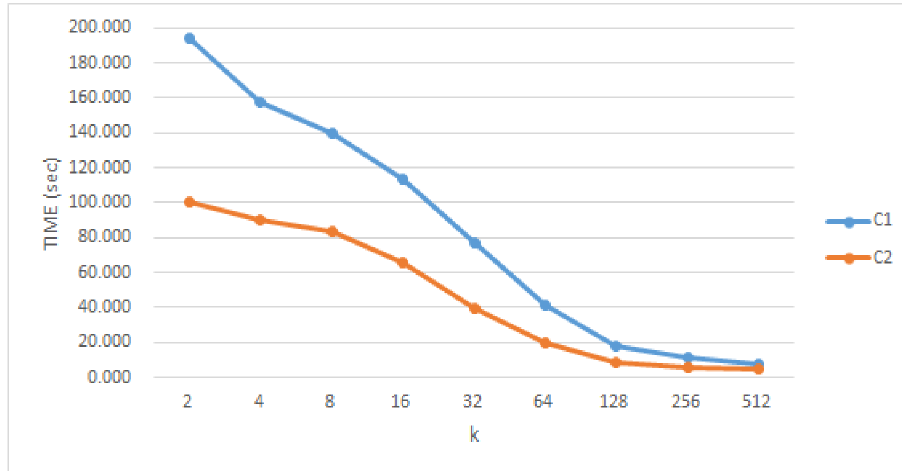


Figure 6.8: Problem 1 - arabic-2005 (r=40)

It is clear that $C2$ outperforms $C1$ in all datasets with most of k, given $r = 40$. This is expected because although the time complexity of $C1$ and $C2$ are both $O(m{\cdot}r)$, the $C2$ actually only pass over the graph one time while $C1$ does two times. However, when k is big, which indicate relatively small subgraphs, $C1$ and $C2$ cost almost the same time, e.g. 1.027 seconds in $C1$, and 0.912 seconds in $C2$ when $k = 512$ on dataset uk-2002. The reason is that when a subgraph is small, pass over the subgraph two times or one time does not take a big difference, then we see similar running time between $C1$ and $C2$ on subgraphs with large k.

k	n
2	20,193,090
4	17,383,395
8	14,939,880
16	11,376,436
32	5,825,037
64	1,829,190
128	459,938
256	194,957
512	79,363

Table 6.4: Sizes of subgraphs of arabic-2005 in different k

Charts also show that the runtime of $C1$ and $C2$ both decrease when k increases. The reason is that a subgraph C_k becomes smaller when k gets bigger, iterations

running on small graphs take less time. As an example, Table 6.4 shows the sizes of subgraphs of dataset arabic-2005 in different k. Based on Figure 6.8 and Table 6.4, it is clear that the runtime of $C1$ and $C2$ is positively correlated with the sizes of subgraphs.

Similarly, when sizes of datasets increase from 1 million nodes (Pokec) to 20 million nodes (arabic-2005), the runtime also increases from 5 seconds to almost 100 seconds using $C2$ when $k = 2$.

6.4.2 Experiment 2 - fixed k

Figure 6.9, 6.10, 6.11, and 6.12 show results for computating top-r CCI communities when $k = 16$ and r is varied on four datasets: Pokec, LiveJournal1, uk-2002, and arabic-2005.

When k is fixed, all subgraphs in varied r of a certain dataset are consistent. Table 6.5 shows the sizes of each subgraph.

Dataset	n
Pokec	639,563
LiveJournal1	853,108
uk-2002	6,629,011
arabic-2005	11,376,436

Table 6.5: Sizes of subgraphs when $k = 16$

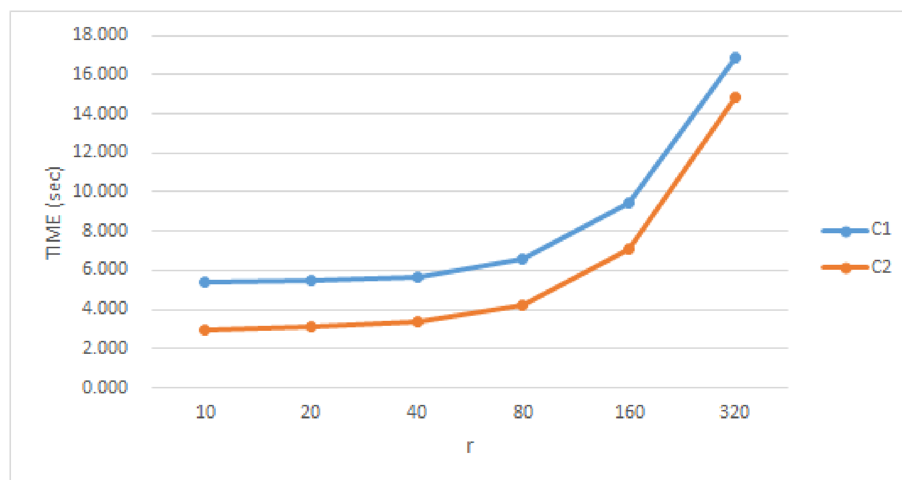


Figure 6.9: Problem 1 - Pokec (k=16)

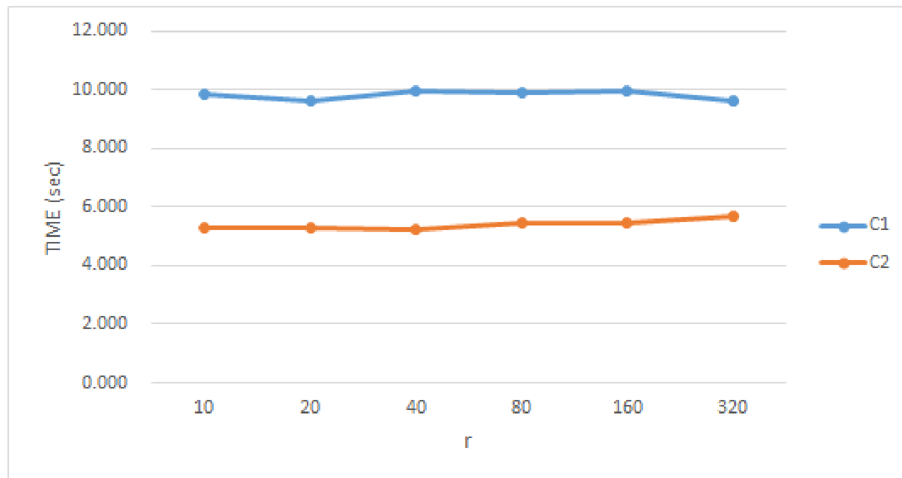


Figure 6.10: Problem 1 - LiveJournal1 (k=16)

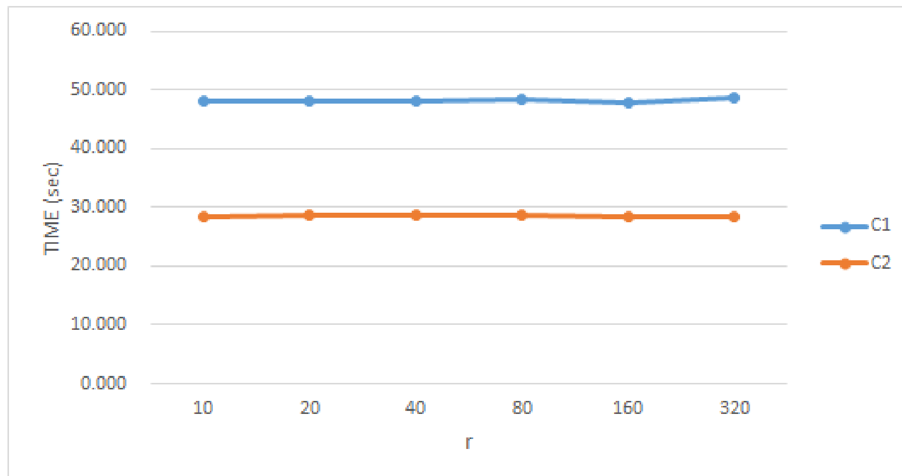


Figure 6.11: Problem 1 - uk-2002 (k=16)

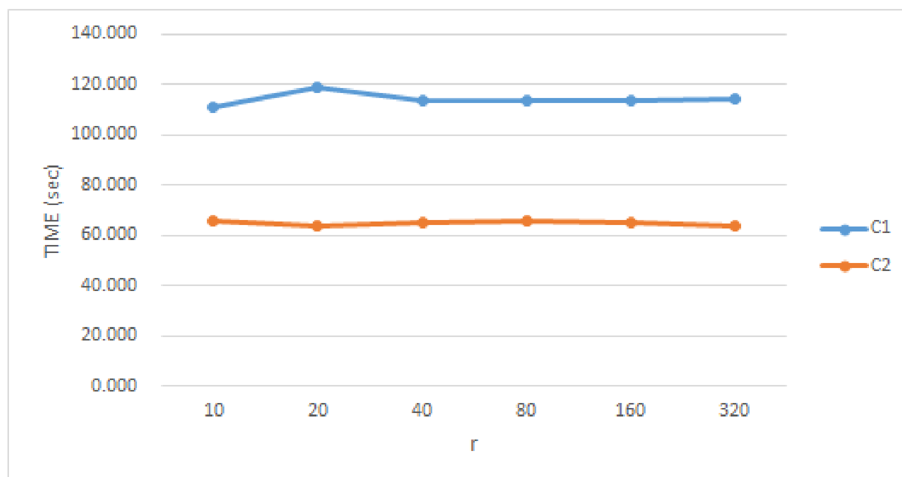


Figure 6.12: Problem 1 - arabic-2005 (k=16)

The charts clearly show that $C2$ outperforms $C1$ for all varied r on all four datasets when $k = 16$. More precisely, we found that the runtime of $C1$ generally takes almost double time of $C2$. This is expected according to our analysis because $C1$ goes through the whole graph two times while $C2$ only makes one pass over.

The runtime of $C1$ and $C2$ in varied r also depends on the graph structure. In dataset LiveJournal1, uk-2002, and arabic-2005, the runtime of $C1$ and $C2$ are quite stable when r increases from 10 to 320. However, the runtime of both two algorithms greatly increases in dataset Pokec when r becomes larger. To find out the reason, we conducted a further analysis to explain this interesting result. We conducted multiple tests to check sizes of top 320 CCI communities found in datasets Pokec, LiveJournal1, and uk-2002. Results are given in Figure 6.13.

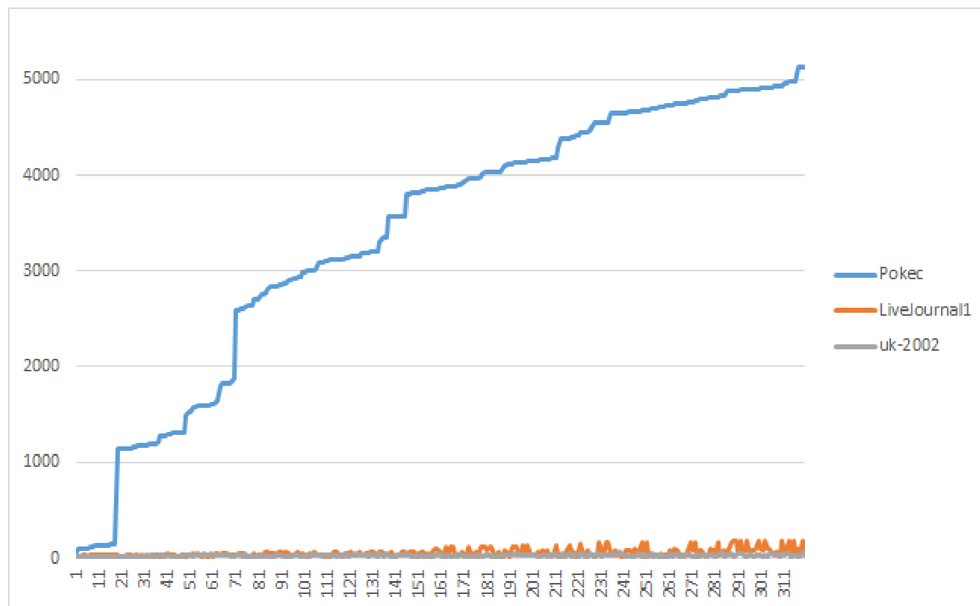


Figure 6.13: Sizes of top 320 CCI communities

As described in the chart, sizes of top 320 CCI communities found in LiveJournal1 and uk-2002 are quite stable around 20 to 100 vertices per CCI communities. Nevertheless, sizes of top 320 CCI communities found in Pokec increased greatly from 70 to 5000 vertices, which indicates the time of MCC computation spending on Pokec should be much longer than that on other graphs. According to Table 6.1, the maximum core number of dataset Pokec is only 47 while the average core number is relatively high, 14. Compared with other datasets, network Pokec shows high cohesiveness, which generally produces large sizes of communities.

6.4.3 Conclusion

Based on above experiments with varied k and r, we get some conclusions:

1. Runtime of $C1$ and $C2$ increases along with sizes of datasets.
2. When a subgraph is relatively small (k is big), performance of $C1$ and $C2$ are similar.
3. When a subgraph is relatively big (k is small), $C2$ outperforms $C1$ almost two times.
4. Generally runtime of $C1$ and $C2$ are not sensitive to varied r in the same graph. But when a graph has high cohesiveness, runtime of $C1$ and $C2$ increases when r becomes larger.

6.5 Testing for Problem 2

For Problem 2, we compare algorithms $NC1$ and $NC2$ in this section. We display test results and analysis on four datasets: Slashdot0811, Pokec, LiveJournal1, uk-2002, and arabic-2005, in below.

Same as testing for Problem 1, there are two flexible parameters, k and r. Therefore, our experiments are divided into two groups: first, we set r to a fixed value, and choose k from ranges mentioned in Table 6.2; second, fix k, and r is picked from ranges mentioned in Table 6.2.

6.5.1 Experiment 1 - fixed r

Figure 6.14, 6.15, 6.16, and 6.17 show results for computing top-r CCI communities when $r = 40$ and k ranges from 2 to 512 on four datasets. Missing points in charts means the runtime overs the time limit (1 hour).

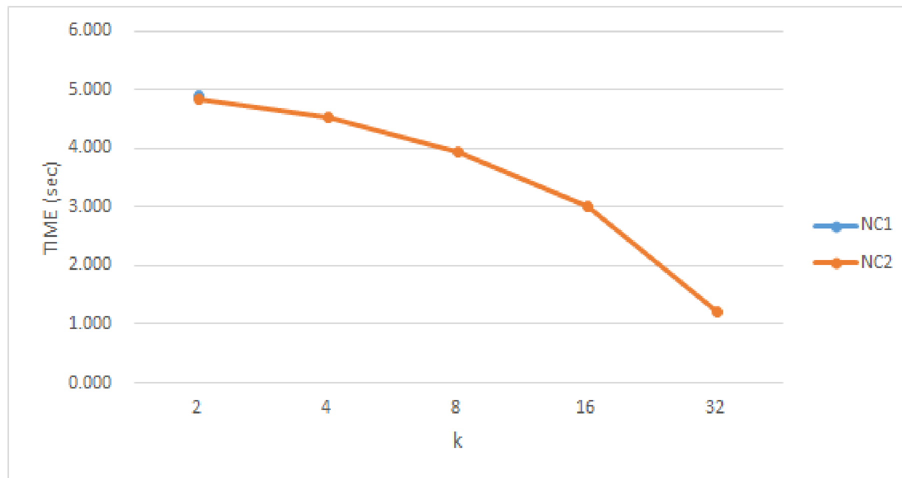


Figure 6.14: Problem 2 - Pokec (r=40)

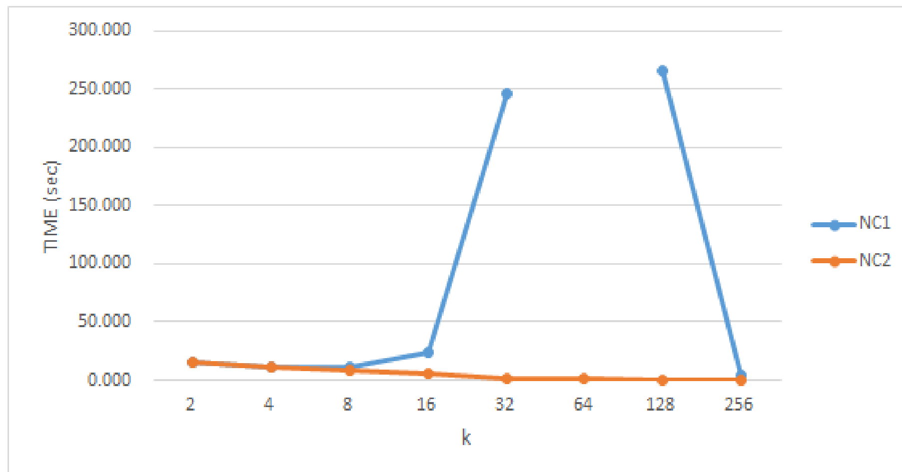


Figure 6.15: Problem 2 - LiveJournal1 (r=40)

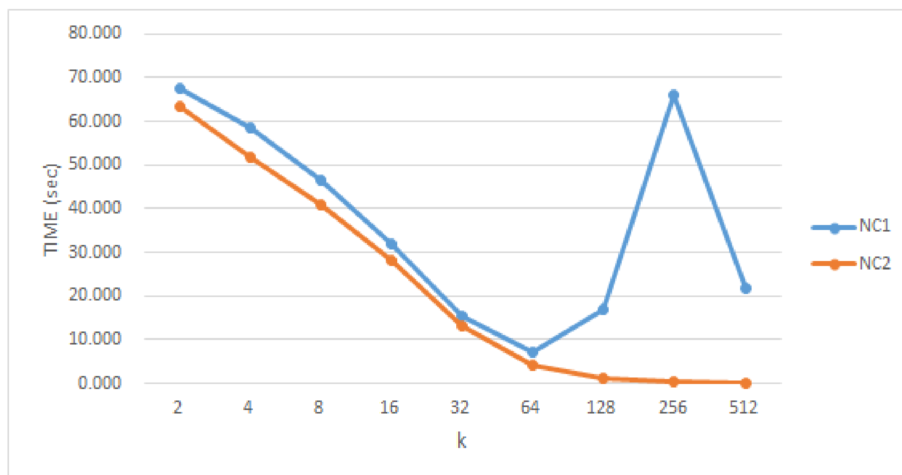


Figure 6.16: Problem 2 - uk-2002 (r=40)

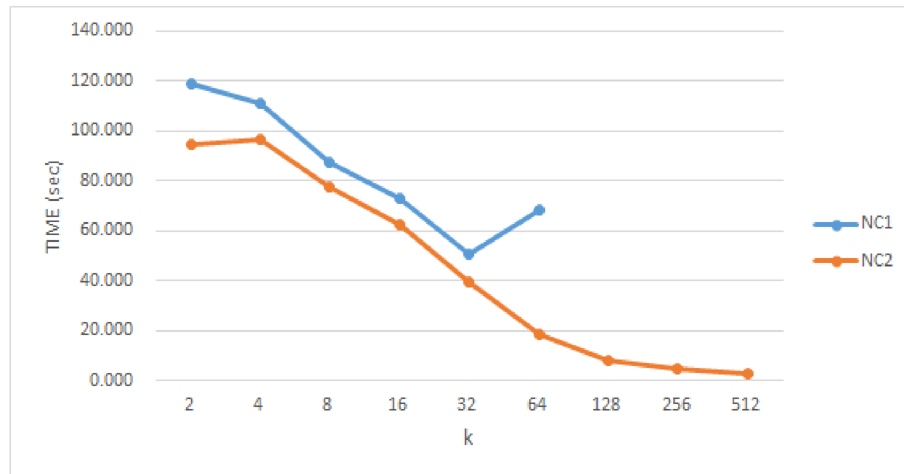


Figure 6.17: Problem 2 - arabic-2005 (r=40)

Charts clearly show that $NC2$ outperforms $NC1$ for all k on all tested datasets. Although points look like overlapped when k equals 2 and 4 in LiveJournal1, $NC2$ still faster than $NC1$ within 2 seconds. Moreover, the runtime of $NC2$ decreases quickly when k rises. This is expected because the time complexity of $NC2$ is $O(m)$, and sizes of subgraphs with varied k become smaller when k rises.

For $NC1$, we can state that the performance is not stable. in Pokec, $NC1$ runs for more than 1 hour for all k, except for $k = 2$, for which it has the same time as $NC2$. In LiveJournal1, $NC1$ runs much longer than $NC2$ when k varies from 32 to 128, while similar results between $NC1$ and $NC2$ can be observed for other k. In uk-2002, $NC1$ takes almost the same time as $NC2$ for $k = 2$ to $k = 64$, but the runtime of $NC1$ is much longer than that of $NC2$ when k is larger than 128. Similarly as in uk-2002, $NC1$ runs over the time limit when k varies from 128 to 512 in arabic-2005. According to our analysis in Section 5.3, NC1 highly depends on the total number of non-containing CCI communities contained in a subgraph, and how many iterations are needed to find them.

We define a percentage *MCC Computation Ratio* which represents how easy it is to find non-containing CCI communities in a given graph using algorithm $NC1$. It is a quotient using the time needed for MCC computation divided by the number of iterations peeling off a graph. When the value of MCC Computation Ratio is closer to 0%, it indicates that it is quite easy to find non-containing CCI communities in this graph.

Definition 3. Given a graph G, $NC1$ calculates number of iterations required to

peel off the whole graph which we denote as i, and times of MCC computation called which denoted as mt.

$$MCC\ Computation\ Ratio = \frac{mt}{i}$$

We take dataset uk-2002 for detailed analysis of $NC1$. Table 6.6 shows results. Specifically, column "total iteration" records i, number of iterations needed to peel off a whole graph, in Algorithm $NC1$, column "MCC Computation" is the times of MCC computation called, column "Non-containing" means how many non-containing CCI communities found in the current subgraph, and the last column "Ratio" is *MCC Computation Ratio*.

k	Total Iterations	MCC Computation	Non-containing	Ratio
2	12,407,794	46	40	0.0004%
4	9,121,360	88	40	0.001%
8	5,985,709	133	40	0.002%
16	2,848,157	276	40	0.010%
32	823,767	771	40	0.094%
64	157,538	1,385	40	0.879%
128	26,494	3,073	40	11.599%
256	5,065	5,065	35	100.00%
512	902	902	4	100.00%

Table 6.6: Details of $NC1$ running on uk-2002 when $r = 40$

As described in the table, all ratios are smaller than 1% from $k = 2$ to $k = 64$, which means times of MCC computation are relatively small compared with numbers of total iterations. Non-containing CCI communities are quite easy to detected in this range, we can say that the time complexity of $NC1$ where k from 2 to 64 in uk-2002 is close to $O(m \cdot r)$. However, ratios become larger, even up to 100%, from $k = 128$ to $k = 512$. It is obvious that $NC1$ computes MCC for each minimum-weight vertex when $k = 256$ and $k = 512$ because times of MCC computation is the same as numbers of total iterations peeled the graph. The time complexity of such conditions is the worst case, $O(m \cdot n)$. For $k = 128$, the ratio of 12% indicates that the time complexity should stay between the best case and the worst case.

Take Figure 6.16 for checking above discussions. First, runtimes of $NC1$ are quite

fast when k from 2 to 64, because their time complexity are all close to $O(m \cdot r)$, and m decreases. Second, since the subgraph of $k = 512$ is smaller than that of $k = 256$, and runtimes of $NC1$ on two subgraphs are both $O(m \cdot n)$, we can see that $NC1$ runs faster in $k = 512$ compared with $k = 256$. Third, the runtime of $k = 128$ is slower than that of $k = 64$, but faster than that of $k = 256$, this is expected because the time complexity of $k = 128$ stays on the middle of $O(m \cdot r)$ and $O(m \cdot n)$ based on its *MCC Computation Ratio*.

Therefore, we can state that given r, $NC1$ is sensitive to graph structures. Details are as follow:

1. Runtime of $NC1$ is positively correlated with the size of graph.
2. Runtime of $NC1$ is positively correlated with the *MCC Computation Ratio*.

6.5.2 Experiment 2 - fixed k

In this section, we compare performances of $NC1$ and $NC2$ when r is varied and k is fixed. Figure 6.18, 6.19, 6.20, and 6.21 show results for computating top-r non-containing CCI communities when $k = 16$ and r ranges from 10 to 320 on four datasets.

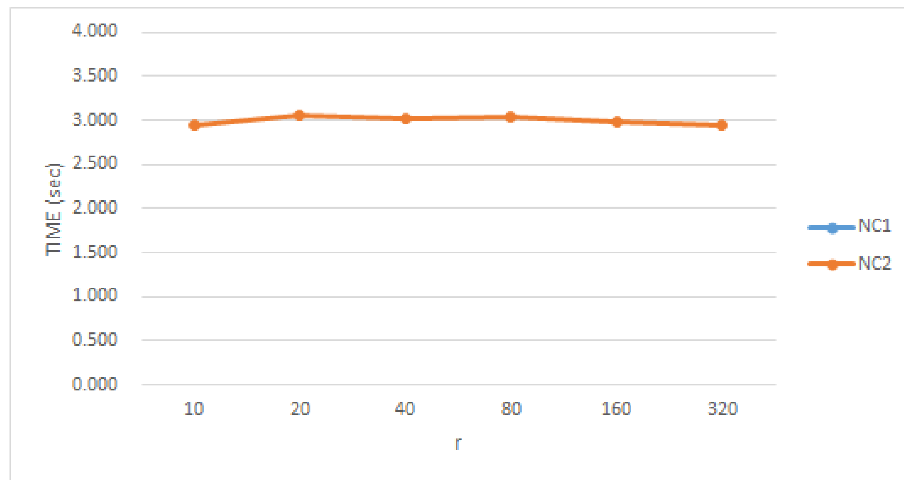


Figure 6.18: Problem 2 - Pokec (k=16)

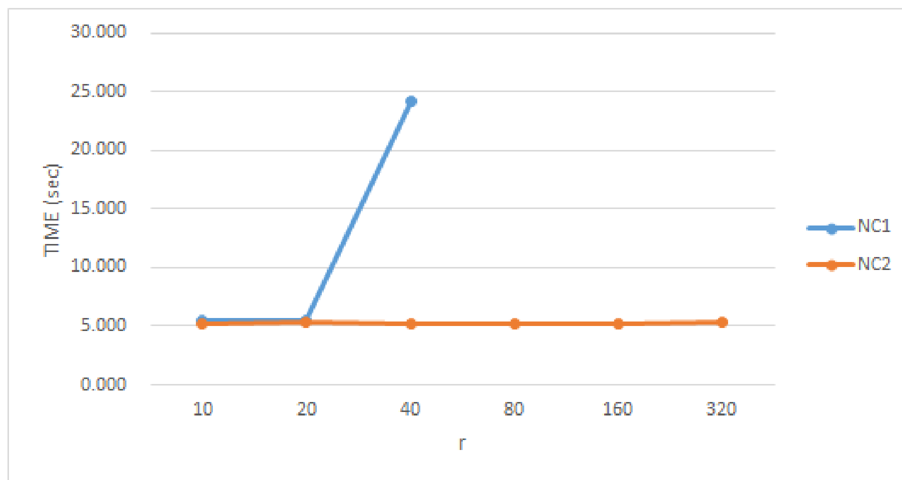


Figure 6.19: Problem 2 - LiveJournal1 (k=16)

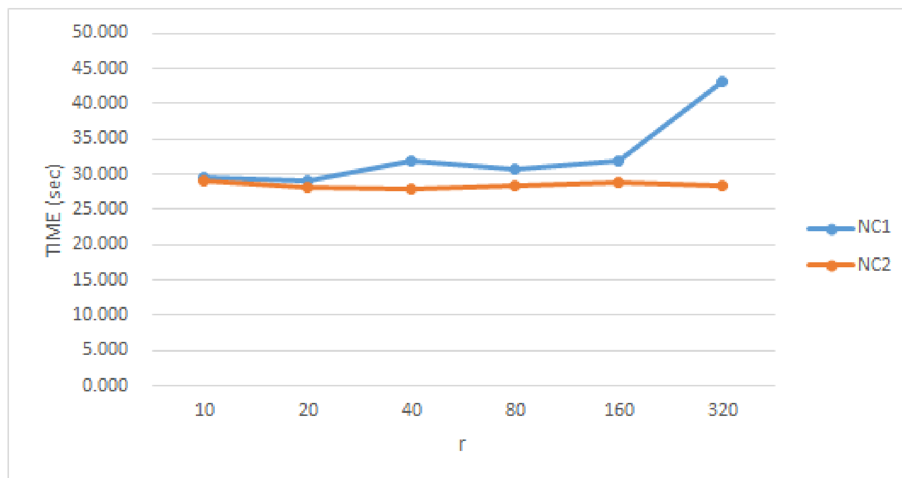


Figure 6.20: Problem 2 - uk-2002 (k=16)

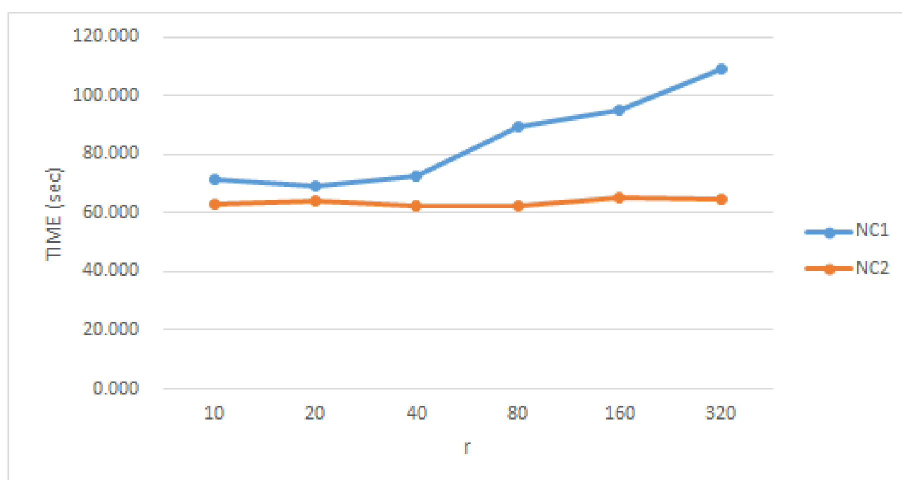


Figure 6.21: Problem 2 - arabic-2005 (k=16)

Based on charts, we can state that $NC2$ is not sensitive to r when k is fixed. In addition, $NC2$ outperforms $NC1$ for all k and r, on all tested datasets. This is expected because the time complexity of $NC2$ is $O(m)$, only depends on sizes of graphs, but the best time complexity of $NC1$ takes $O(m \cdot r)$.

For $NC1$, it can not get results on all r, on all datasets. In Pokec, a relatively high cohesiveness graph, none of r is able to get results. In general, runtimes of $NC1$ increase when r becomes larger. This is expected because the time complexity of $NC1$ stays between $O(m \cdot r)$ and $O(m \cdot n)$, m are all the same when k is fixed in a graph, but times of MCC computation become bigger in order to find more non-containing CCI communities. As in LiveJournal1, we are able to get results from $r = 10$ to $r = 40$, but $NC1$ runs over the time limit once r becomes larger than 40.

Further analysis of $NC1$ was conducted to explain the relationship between runtime and varied r.

We test $NC1$ running on uk-2002 when $k = 256$, and results are given in Figure 6.22.

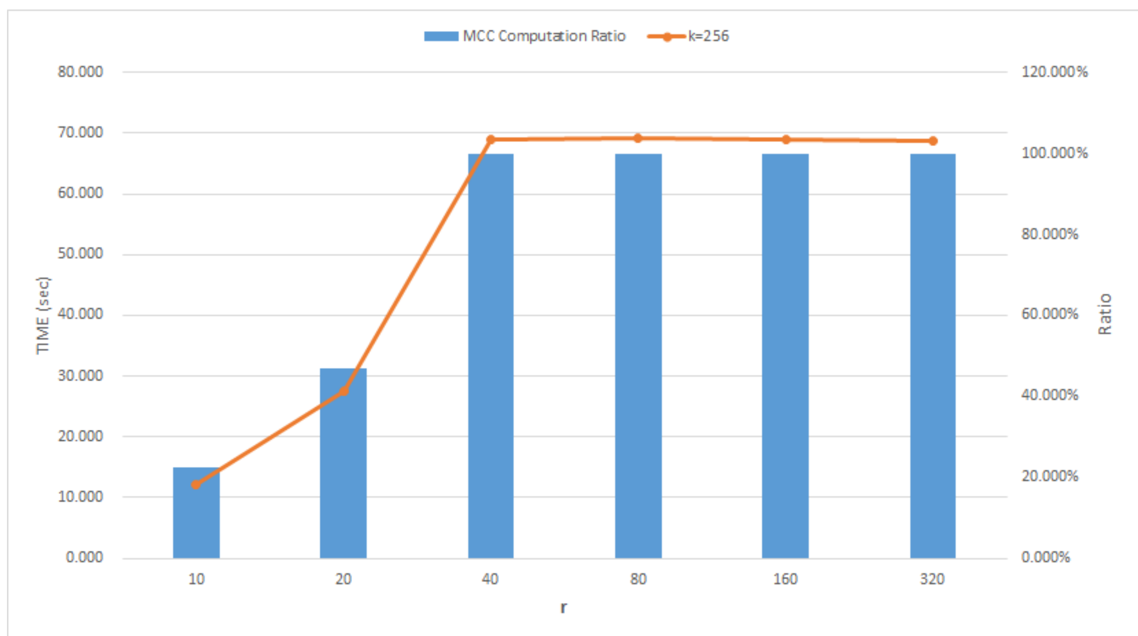


Figure 6.22: NC1 running on uk-2002 (k=256)

In Figure 6.22, runtimes of $NC1$ increase from 12 seconds to 28 seconds, then stay on around 70 seconds from $r = 10$ to $r = 320$. At the same time, *MCC Computation Ratio* increases from 22% to 47%, then holds on 100%. It is interesting that runtimes and *MCC Computation Ratio* have the same pattern. This is expected because $NC1$

calculates MCC for more vertices when *MCC Computation Ratio* becomes larger. We can state that the runtime for 100% *MCC Computation Ratio* includes the MCC computation for each minimum-weight vertex, and takes the worst time complexity of $NC1$, $O(m \cdot n)$. Since m and n do not change when k is fixed, runtimes of $NC1$ are the same or very close when r from 40 to 320 in the subgraph of uk-2002 where $k = 256$.

Based on above discussions, it is clear that given a k, or a subgraph, the runtime of $NC1$ is positively correlated with the *MCC Computation Ratio*.

6.5.3 Conclusion

Based on above experiments with varied k and r, we get some conclusions:

1. $NC2$ outperforms $NC1$ on all combinations of varied k and r.
2. Runtime of $NC2$ grows along with increment of sizes of graphs.
3. Runtime of $NC2$ is not sensitive to varied r given a fixed k.
4. Runtime of $NC1$ is positively correlated with the sizes of graphs and the *MCC Computation Ratio*.

Chapter 7

Conclusions and Future Work

This thesis introduces an existing fast in-memory algorithm, Batagelj-Zaversnik (BZ) algorithm, for computing k-core decomposition, a highly efficient graph compression framework WebGraph at first. Then the *k-influential community* model is proposed, and two initial algorithms (**C0** and **NC0**) for detecting such models are presented.

Based on the *k-influential community* model, also called *CCI community*, we define two top-r CCI community problems which we would like to solve in this thesis:

1. Given an undirected graph G, and two positive integers k and r, how to compute the top-r CCI communities?

2. Given an undirected graph G, and two positive integers k and r, how to compute the top-r non-containing CCI communities?

For the first problem, we propose two algorithms (**C1** and **C2**), which are faster than $C0$ by orders of magnitude, and take space in the order of the compressed version of the graph. For the second problem, we also propose two algorithms (**NC1** and **NC2**), which require very reasonable memory compared with $NC0$. For $NC2$, it is faster than $NC0$ by orders of magnitude. Given an undirected graph $G = (V, E)$ where $n = |V|$ and $m = |E|$, k, and r, details of algorithms are as follow:

1. Algorithm $C1$ decreases the times of maximally connected components computation. The time complexity of $C1$ is $O(m \cdot r)$, and the space complexity is $O(m)$.

2. Algorithm $C2$ takes a hash-based structure to save deleted nodes. The time complexity of $C2$ is $O(m \cdot r)$, but $C2$ actually twice faster than $C1$. The space complexity of $C2$ is $O(m)$.

3. Algorithm $NC1$ takes the same hash-based structure as $C2$. The time complexity of $NC1$ is $O(m \cdot n)$. The space complexity of $NC1$ is $O(m)$.

4. Algorithm $NC2$ completely eliminates maximally connected components computation. The time complexity of $NC2$ is $O(m)$, and the space complexity is $O(m)$ too.

Finally, we present extensive experiments on a selection of real-world network datasets. The biggest graph we used is arabic-2005 which has about 22 million nodes and 553 million edges. Our results show that algorithms $C1$, $C2$, and $NC2$ are able to compute communities of Problem 1 and 2 for all combination of k and r in all datasets, on a consumer-grade machine, within very reasonable time. In addition, $C2$ and $NC2$ perform best for Problem 1 and Problem 2 respectively. We provide full implementations of $C2$ and $NC2$ in appendix.

What we have done in this thesis provides the basis for future work in several directions. First, it is interesting to apply top-r, k-core communities to probabilistic graphs [7, 18] and edge-labeled graphs [15, 33]. Second, with the development of the internet, the size of network graphs become considerable. Inspired from [33, 36], distributed algorithms for computing top-r, k-core communities should be devised. Third, we would like to explore the usefulness of top-r, k-core communities in identifying community formation in biological networks [17], in trust prediction [23], in learning the news in social networks [29], and in clearing a contamination from a network [34].

Appendix A

Additional Information

A.1 Algorithm $C2$

```
import it.unimi.dsi.webgraph.ImmutableGraph;
import it.unimi.dsi.webgraph.NodeIterator;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.*;

public class C2 {
   double[] gw;
   int[] deg; // degrees
   int k; // given k
   BitSet gone;
   ImmutableGraph G;
   ArrayDeque<ArrayDeque<Integer>> deletedNodes;
   long E = 0;

   public C2(String G_basename, String nodeweighsfilename) throws
       Exception {
      G = ImmutableGraph.load(G_basename);
      gone = new BitSet(G.numNodes());
      gw = new double[G.numNodes()];
      System.out.println("Number of nodes in undirected graph: " +
          G.numNodes());

      BufferedReader reader = new BufferedReader(new
          FileReader(nodeweighsfilename));
      String line;
      while ((line = reader.readLine()) != null) {
         String[] strA = line.split("\t");
         Integer node = Integer.parseInt(strA[0]);
         Double weight = Double.parseDouble(strA[1]);
         if(node < G.numNodes())
            gw[node] = weight;
      }
```

```
        reader.close();
        System.out.println("Reading weight finish!");
    }

    // Computes the core decomposition, implements the Batagelz and
        Zaversnik algorithm
    public int[] KCoreCompute() {

        int n = G.numNodes();
        int md = maxDegree(G);
        int[] vert = new int[n];
        int[] pos = new int[n];
        int[] deg = new int[n];
        int[] bin = new int[md + 1]; // md+1 because we can have zero degree

        for (int d = 0; d <= md; d++)
            bin[d] = 0;
        for (int v = 0; v < n; v++) {
            deg[v] = G.outdegree(v);
            E += deg[v];
            bin[deg[v]]++;
        }

        int start = 0;
        for (int d = 0; d <= md; d++) {
            int num = bin[d];
            bin[d] = start;
            start += num;
        }

        // bin-sort vertices by degree
        for (int v = 0; v < n; v++) {
            pos[v] = bin[deg[v]];
            vert[pos[v]] = v;
            bin[deg[v]]++;
        }
        // recover bin[]
        for (int d = md; d >= 1; d--)
            bin[d] = bin[d - 1];
        bin[0] = 0; // 1 in original

        // main algorithm
        for (int i = 0; i < n; i++) {
            int v = vert[i]; // smallest degree vertex
            int[] N_v = G.successorArray(v);
            int v_deg = G.outdegree(v);
            for (int ii = 0; ii < v_deg; ii++) {
                int u = N_v[ii];

                if (deg[u] > deg[v]) {
                    int du = deg[u];
                    int pu = pos[u];
                    int pw = bin[du];
                    int w = vert[pw];
                    if (u != w) {
                        pos[u] = pw;
                        vert[pu] = w;
```

```
                pos[w] = pu;
                vert[pw] = u;
            }
            bin[du]++;
            deg[u]--;
        }
    }
}
return deg;
}

public int maxDegree(ImmutableGraph G) {
    Iterator<Integer> degIter = G.outdegrees();
    int maxDegree = -1;
    while (degIter.hasNext()) {
        Integer deg = degIter.next();
        if (deg > maxDegree)
            maxDegree = deg;
    }
    return maxDegree;
}

void topInfluentialCommunitiesAlg2(Integer r, Integer k) throws
     IOException {
    this.k = k;
    System.out.println("r = " + r + " k = " + k);

    // Compute core decomposition
    this.deg = KCoreCompute();
    System.out.println("KCoreCompute finish!");

    // Create a priority queue for the nodes of Ck
    PriorityQueue<NodeInfluence> pqueue = new
        PriorityQueue<NodeInfluence>(10, new NodeInfluenceComparator());

    // Populate the priority queue
    NodeIterator vi = G.nodeIterator();
    while (vi.hasNext()) {
        int u = vi.next();
        if (deg[u] >= k)
            pqueue.add(new NodeInfluence(u, gw[u]));
        else
            gone.set(u);
    }

    System.out.println("Ck_size " + pqueue.size());

    // Reuse the deg array to keep for each live vertex its number of
        live neighbors
    vi = G.nodeIterator();
    while (vi.hasNext()) {
        int u = vi.next();
        deg[u] = d_Ck(u);
    }

    long startTime = System.currentTimeMillis();
    long estimatedTime;
```

```
        deletedNodes = new ArrayDeque<ArrayDeque<Integer>>();
        ArrayDeque<Integer> DN;

        while (!pqueue.isEmpty()) {
            int u = -1;
            do {
                if (pqueue.isEmpty())
                    break;
                u = pqueue.poll().node;
            } while (gone.get(u));

            if (u == -1 || gone.get(u))
                break;

            DN = new ArrayDeque<Integer>();
            DFS(u, DN);
            deletedNodes.addLast(DN);

            if(deletedNodes.size() > r)
                deletedNodes.removeFirst();
        }

        while (!deletedNodes.isEmpty()) {
            DN = deletedNodes.pollLast();
            int DNsize = DN.size();
            for(int n = 0; n < DNsize; n++){
                gone.clear(DN.pollLast());
            }
            int u = DN.pollFirst();
            Set<Integer> cc = getConnectedComponentCk(u);

            System.out.println("cc.size() is " + cc.size() + ", influence is "
                 + gw[u]);
            System.out.println(cc);
        }

        estimatedTime = System.currentTimeMillis() - startTime;
        System.out.println("Time for core part = " + estimatedTime / 1000.0);
    }

    void DFS(Integer u, ArrayDeque<Integer> DN) {
        gone.set(u);
        DN.add(u);
        int[] u_neighbors = G.successorArray(u);
        int u_deg = G.outdegree(u);
        for (int i = 0; i < u_deg; i++) {
            int v = u_neighbors[i];
            deg[v]--;
            if (!gone.get(v) && deg[v] < k)
                DFS(v, DN);
        }
    }

    int d_Ck(int v) {
        int d = 0;
        int[] v_neighbors = G.successorArray(v);
```

```
   int v_deg = G.outdegree(v);
   for (int i = 0; i < v_deg; i++) {
      int w = v_neighbors[i];
      if (!gone.get(w))
         d++;
   }

   return d;
}

Set<Integer> N_Ck(Integer u) {
   Set<Integer> N_u_Ck = new HashSet<Integer>();
   int[] u_neighbors = G.successorArray(u);
   int u_deg = G.outdegree(u);
   for (int i = 0; i < u_deg; i++) {
      int v = u_neighbors[i];
      if (!gone.get(v))
         N_u_Ck.add(v);
   }

   return N_u_Ck;
}

Set<Integer> getConnectedComponentCk(Integer u) {
   Set<Integer> cc = new HashSet<Integer>();
   ConnectedComponentsCkDFS(u, cc);
   return cc;
}

void ConnectedComponentsCkDFS(Integer u, Set<Integer> cc) {
   cc.add(u);
   Set<Integer> N_u_Ck = N_Ck(u);
   for (Integer v : N_u_Ck)
      if (!cc.contains(v)) //Avoid loops
         ConnectedComponentsCkDFS(v, cc);
}

class NodeInfluence {
   Integer node;
   Double influence;

   NodeInfluence(Integer node, Double influence) {
      this.node = node;
      this.influence = influence;
   }
}

public class NodeInfluenceComparator implements
    Comparator<NodeInfluence> {
   public int compare(NodeInfluence x, NodeInfluence y) {
      if (x.influence < y.influence)
         return -1;
      if (x.influence > y.influence)
         return 1;
      return 0;
   }
}
```

```
  public static void main(String[] args) throws Exception {
    C2 kc = new C2(args[0], args[1]);
    kc.topInfluentialCommunitiesAlg2(Integer.valueOf(args[2]),
        Integer.valueOf(args[3]));

    System.out.println("C2 is done");
  }
}
```

A.2 Algorithm $NC2$

```
import it.unimi.dsi.webgraph.ImmutableGraph;
import it.unimi.dsi.webgraph.NodeIterator;

import java.io.BufferedReader;
import java.io.FileReader;
import java.util.*;

public class NC2 {
  double[] gw;
  int[] deg; // degrees
  int k; // given k
  BitSet gone;
  ImmutableGraph G;
  ArrayDeque<ArrayDeque<Integer>> deletedNodes;
  long E = 0;

  public NC2(String G_basename, String nodeweighsfilename) throws
       Exception {
    G = ImmutableGraph.load(G_basename);
    gone = new BitSet(G.numNodes());
    gw = new double[G.numNodes()];
    System.out.println("Number of nodes in undirected graph: " +
        G.numNodes());

    BufferedReader reader = new BufferedReader(new FileReader(
          nodeweighsfilename));
    String line;

    while ((line = reader.readLine()) != null) {
      String[] strA = line.split("\t");
      Integer node = Integer.parseInt(strA[0]);
      Double weight = Double.parseDouble(strA[1]);
      if (node < G.numNodes())
        gw[node] = weight;
    }
    reader.close();
    System.out.println("Reading weight finish!");
  }

  // Computes the core decomposition, implements the Batagelz and
       Zaversnik algorithm
```

```
public int[] KCoreCompute() {

   int n = G.numNodes();
   int md = maxDegree(G);
   int[] vert = new int[n];
   int[] pos = new int[n];
   int[] deg = new int[n];
   int[] bin = new int[md + 1]; // md+1 because we can have zero degree

   for (int d = 0; d <= md; d++)
      bin[d] = 0;
   for (int v = 0; v < n; v++) {
      deg[v] = G.outdegree(v);
      E += deg[v];
      bin[deg[v]]++;
   }

   int start = 0;
   for (int d = 0; d <= md; d++) {
      int num = bin[d];
      bin[d] = start;
      start += num;
   }

   // bin-sort vertices by degree
   for (int v = 0; v < n; v++) {
      pos[v] = bin[deg[v]];
      vert[pos[v]] = v;
      bin[deg[v]]++;
   }
   // recover bin[]
   for (int d = md; d >= 1; d--)
      bin[d] = bin[d - 1];
   bin[0] = 0; // 1 in original

   // main algorithm
   for (int i = 0; i < n; i++) {
      int v = vert[i]; // smallest degree vertex
      int[] N_v = G.successorArray(v);
      int v_deg = G.outdegree(v);
      for (int ii = 0; ii < v_deg; ii++) {
         int u = N_v[ii];

         if (deg[u] > deg[v]) {
            int du = deg[u];
            int pu = pos[u];
            int pw = bin[du];
            int w = vert[pw];
            if (u != w) {
               pos[u] = pw;
               vert[pu] = w;
               pos[w] = pu;
               vert[pw] = u;
            }
            bin[du]++;
            deg[u]--;
         }
```

```
      }
   }
   return deg;
}

public int maxDegree(ImmutableGraph G) {
   Iterator<Integer> degIter = G.outdegrees();
   int maxDegree = -1;
   while (degIter.hasNext()) {
      Integer deg = degIter.next();
      if (deg > maxDegree)
         maxDegree = deg;
   }
   return maxDegree;
}

void topInfluentialCommunitiesAlg2(Integer r, Integer k) {
   this.k = k;
   System.out.println("r = " + r + " k = " + k);

   // Compute core decomposition
   this.deg = KCoreCompute();
   System.out.println("KCoreCompute finish!");

   // Create a priority queue for the nodes of Ck
   PriorityQueue<NodeInfluence> pqueue = new
       PriorityQueue<NodeInfluence>(10, new NodeInfluenceComparator());

   // Populate the priority queue
   NodeIterator vi = G.nodeIterator();
   while (vi.hasNext()) {
      int u = vi.next();
      if (deg[u] >= k)
         pqueue.add(new NodeInfluence(u, gw[u]));
      else
         gone.set(u);
   }

   System.out.println("Ck_size " + pqueue.size());

   // Reuse the deg array to keep for each live vertex its number of
       live neighbors
   vi = G.nodeIterator();
   while (vi.hasNext()) {
      int u = vi.next();
      deg[u] = d_Ck(u);
   }

   // Continue with the algorithm
   long startTime = System.currentTimeMillis();
   long estimatedTime;
   ArrayDeque<Integer> DN;
   deletedNodes = new ArrayDeque<ArrayDeque<Integer>>();

   while (!pqueue.isEmpty()) {
      int u = -1;
      do {
```

```
            if (pqueue.isEmpty())
               break;
            u = pqueue.poll().node;
         } while (gone.get(u));

         if (u == -1 || gone.get(u))
            break;

         DN = new ArrayDeque<Integer>();
         DFS(u, DN);

         boolean isNC = true;
         Iterator<Integer> i = DN.iterator();
         while (i.hasNext()) {
            if (deg[(int) i.next()] != 0) {
               isNC = false;
               break;
            }
         }
         if (isNC) {
            deletedNodes.addLast(DN);
         }
      }

      for (int i = 0; i < r; i++) {
         if (deletedNodes.isEmpty())
            break;
         ArrayDeque<Integer> cc = deletedNodes.pollLast();
         System.out.println("cc.size() is " + cc.size());
         System.out.println(cc);
      }
      estimatedTime = System.currentTimeMillis() - startTime;
      System.out.println("Time for core part = " + estimatedTime / 1000.0);
   }

   void DFS(Integer u, ArrayDeque<Integer> DN) {
      gone.set(u);
      DN.add(u);
      int[] u_neighbors = G.successorArray(u);
      int u_deg = G.outdegree(u);
      for (int i = 0; i < u_deg; i++) {
         int v = u_neighbors[i];
         deg[v]--;
         if (!gone.get(v) && deg[v] < k)
            DFS(v, DN);
      }
   }

   int d_Ck(int v) {
      int d = 0;

      int[] v_neighbors = G.successorArray(v);
      int v_deg = G.outdegree(v);
      for (int i = 0; i < v_deg; i++) {
         int w = v_neighbors[i];
         if (!gone.get(w))
            d++;
```

```
        }

        return d;
    }

    Set<Integer> N_Ck(Integer u) {
        Set<Integer> N_u_Ck = new HashSet<Integer>();

        int[] u_neighbors = G.successorArray(u);
        int u_deg = G.outdegree(u);
        for (int i = 0; i < u_deg; i++) {
            int v = u_neighbors[i];
            if (!gone.get(v))
                N_u_Ck.add(v);
        }

        return N_u_Ck;
    }

    class NodeInfluence {
        Integer node;
        Double influence;

        NodeInfluence(Integer node, Double influence) {
            this.node = node;
            this.influence = influence;
        }
    }

    public class NodeInfluenceComparator implements
         Comparator<NodeInfluence> {
        public int compare(NodeInfluence x, NodeInfluence y) {
            if (x.influence < y.influence)
                return -1;
            if (x.influence > y.influence)
                return 1;
            return 0;
        }
    }

    public static void main(String[] args) throws Exception {
        NC2 kc = new NC2(args[0], args[1]);
        kc.topInfluentialCommunitiesAlg2(Integer.valueOf(args[2]),
            Integer.valueOf(args[3]));

        System.out.println("NC2 is done");
    }
}
```

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

[3] Vladimir Batagelj and Andrej Mrvar. Pajek-program for large network analysis. *Connections*, 21(2):47–57, 1998.

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

[5] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. ACM Press, 2011.

[6] 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.

[7] 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.

[8] 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.

[9] 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. ACM, 2016.

[10] 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.

[11] 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.

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

[13] 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.

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

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

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

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

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

[25] 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.

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

[27] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. Distributed k-core decomposition. *Parallel and Distributed Systems, IEEE Transactions on*, 24(2):288–300, 2013.

[28] M. A. Porter, J.-P. Onnela, and P. J. Mucha. Communities in networks. *ArXiv e-prints*, February 2009.

[29] 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.

[30] Ahmet Erdem Saríyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V Çatalyürek. Streaming algorithms for k-core decomposition. *Proceedings of the VLDB Endowment*, 6(6):433–444, 2013.

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

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

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

[34] 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.

[35] Mauro Sozio and Aristides Gionis. The community-search problem and how to plan a successful cocktail party. *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 939–948, 2010.

[36] 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.

[37] Jia Wang and James Cheng. Truss decomposition in massive networks. *Proceedings of the VLDB Endowment*, 5(9):812–823, 2012.

[38] Dong Wen, Lu Qin, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. I/O efficient core graph decomposition at web scale. *CoRR*, abs/1511.00367, 2015.

[39] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

[40] 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.