Finding Core-periphery Structure in Directed Networks

Finding Core-periphery Structure in Directed Networks

An algorithm for detecting multiple-group core-periphery structure in directed networks

Thesis

to obtain the degree of Master
at Delft University of Technology,
under the supervision of Prof.dr.ir. RE Kooij
to be defended on Tuesday, August 31, 2021 at 13:00

by

Hao Huang

Acknowledgment

Throughout writing this thesis, I have received a great deal of support and assistance.

I would first like to thank my supervisor, Professor Rob Kooij. Your guidance was profound in formulating the research questions and heading in the proper research directions. The insightful feedback during the writing pushed me to refine my thesis to a higher quality that I alone could never achieve. The support and teaching I got from you was invaluable.

I would like to acknowledge Mark van Staalduinen, Managing Director and Founder of CFLW Cyber Strategies. This thesis originated from a cooperation of the TU Delft NAS group and CFLW Cyber Strategies. I want to thank you for offering me this opportunity and professional input on the dark web.

I would also like to thank my schoolmate at TU Delft, who assisted me with their experience on writing and university paperwork. But, more importantly, I want to thank you and all my friends in the United Kindom, the United States, and China who accompanied me in this pandemic. It is this very special time that brought us together. I wish to meet you again one day.

Abstract

The core-periphery structure is a mesoscale topological structure that refers to the presence of a dense core and a sparse periphery. The core-periphery structure has been discovered in financial, biological, and technological networks. Various methods for detecting core-periphery structure have been proposed, exploring different discrete or continuous models and single or multiple core-periphery groups. A method that implements multiple-group detection and edge-direction dependency to core-periphery detection is yet to be researched.

This report proposes an algorithm to extract the core-periphery structure that satisfies multiple-group and edge-direction dependency requirements. This algorithm features a heuristic process and can process a large-scale network in an acceptable amount of time. Details of the theory behind the proposed algorithm are presented. The algorithm is tested on synthetic, random scale-free, and sampled dark web networks to verify the basic and advanced feasibility. Finally, in-depth analysis with the knowledge of core-periphery structure on a large-scale dark web network sample is presented.

CONTENTS

1 Introduction **1**

2 Core-Periphery Structure and Detection Algorithms **3**
2.1 Core-periphery Structure 3
2.2 Multiple-group Detection 5
2.3 Core-Periphery Structure in Directed Graphs 6

3 Methods **7**
3.1 Judging criteria, Group and Partition 7
3.2 Algorithm Workflow 12
3.2.1 Initializing and Pre-processing 13
3.2.2 Label-switching Heuristic Process 13
3.2.3 Outcome of the Algorithm 14
3.2.4 Extra Runs 14
3.3 Performance Analysis and Implementation 14

4 Results **17**
4.1 Synthetic Network 17
4.2 Random Scale-free graphs 18
4.3 Dark Web Dataset 21
4.3.1 Dataset 22
4.3.2 Core-Periphery Analysis 23

5 Discussion **31**

1 Introduction

Graph theory enables us to mathematically study networks that play a role in various aspects of our lives, including physical, biological, financial, information, and technological networks. When studying the networks, the graph can be described as a collection of nodes and edges. Edges link two nodes symmetrically in undirected graphs. But in directed graphs, an edge is a one-way connection from the source node to the target node. Usually, when the direction of edges is introduced into a graph, the complexity of studying it increases considerably.

The core-periphery structure is a fundamental network pattern, typically referring to the presence of a set of densely connected "core" nodes and a "periphery" that is well-connected to the core but sparsely connected to other periphery nodes. Different from communities, the core nodes are more reasonably well-connected to periphery nodes in the network. The core-periphery structure is a mesoscopic feature that lies between the microscopic (local node properties) and the macroscopic (global network properties) level.

The core-periphery structure is a relatively newly proposed concept. It was the first time proposed in the 1970s but not standardized until 2000, and most studies on this topic span from 2010 to now. The core-periphery structure still has great potential in applications. Part of the goal of this project is to experiment with the knowledge of core-periphery structure on real-life networks and see how it can help improve on usual community detection methods.

Many methods for detecting core-periphery structures were developed. The detection and presentation of core-periphery structures vary from different methods. The primary distinction is between discrete and continuous models, which refers to how nodes are determined to belong to the core according to a discrete value or continuous score. Another newly developed idea is multiple-group core-periphery detection. It is beneficial to regard the network as a collection of multiple core-periphery pairs because it presents a global core-periphery structure and regional patterns, thus revealing more de-

tailed information of the network's grouped communities and allowing the study of them separately. This report dedicates the proposed algorithm to detecting discrete-model multiple-group core-periphery structure using an automated partitioning method.

The mass majority of methods for detecting core-periphery structures are developed for undirected networks. Although many of them can be applied to directed networks, usually, they do not provide an edge-direction-dependent definition but rather disregard the edge direction. When it comes to directed graphs, there is no well-established common practice for core-periphery structure detecting. In this report, the primary goal, and at the same time, the biggest challenge, is to make the proposed algorithm edge-direction dependent while maintaining the multiple-group detecting feature as mentioned earlier.

This study is an extended project of my previous work. In cooperation with the TU Delft NAS group, CFLW Cyber Strategies[1] provided a dark web dataset for analysis. The dark web has several unique and challenging topological properties. The proposed algorithm will take those properties into consideration and provide analysis from the core-periphery perspective. Further applications with the knowledge of core-periphery structures are also explored.

The rest of the report is organized as follows: Chapter 2 explains the principle of core-periphery structure and presents precedent algorithms that utilize different methods and parameters to detect the structure. Chapter 3 describes the design of the algorithm in detail of theory and implementation. Chapter 4 reports the result from analysis conducted on various graphs. Finally, chapter 5 concludes the finding and achievements and discusses unresolved questions.

[1]https://cflw.com/

2

CORE-PERIPHERY STRUCTURE AND DETECTION ALGORITHMS

This chapter explains the fundamental principle of core-periphery structure. Two of the alternative methods for detecting core-periphery structures, which this proposed algorithm has taken the example of, are highlighted. One is the multiple-group detecting algorithm developed by Kojaku and Masuda [1]. The other one is the method for detecting core-periphery structures in directed graphs by Andrew Elliott *et al.* [2].

2.1. CORE-PERIPHERY STRUCTURE

Core-periphery structure (also denoted by C-P in this report for the sake of brevity) is a concept that has received attention since the late 1970s. Various notions of the core-periphery structure are proposed in different fields until in 2000, Borgatti and Everett proposed the most widely accepted definition of core-periphery structure in weighted, undirected graphs for both discrete and continuous models [3]. In their definition, a bimodular block model includes a fully connected set of core nodes and a periphery with nodes that are only fully connected to the core node-set. This block model is called an ideal core-periphery structure. The adjacency matrix shows the relationship between core and periphery more clearly. The periphery nodes radiate from the densely connected core. The idealized C-P structure and an approximate one are shown in Figure 2.1 In real-world networks, the ideal structure's requirement is extremely difficult to be satisfied. Usually, a more relaxed criterion of core-periphery structure is applied. The core nodes are densely connected, and the periphery nodes are well connected to the core while loosely connected to each other.

In Borgatti and Everett's algorithm (the BE algorithm for short), to detect a core-periphery pair in a network with N nodes, they defined:

$$Q = \sum_{i=1}^{N} \sum_{j=1}^{N} A_{ij} c_i c_j \tag{2.1}$$

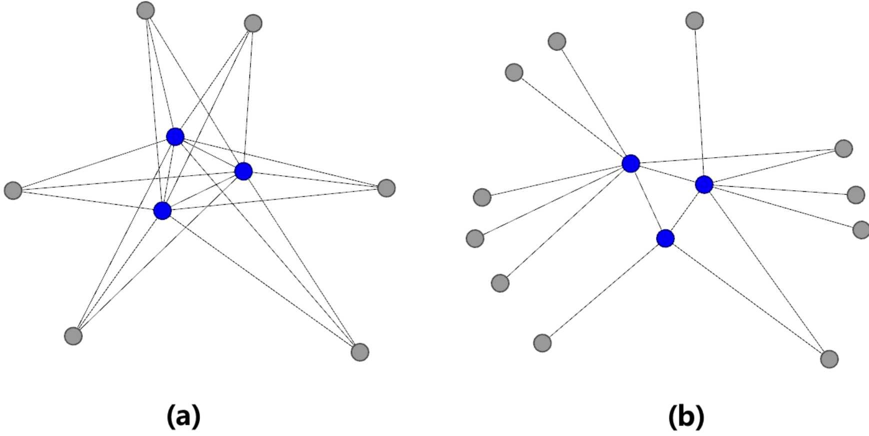


Figure 2.1: Two simple core-periphery structure:
(a) Idealized structure (b) approximate structure

where A_{ij} is the element in the adjacency matrix A of the given network, i.e. $A_{ij} = 1$ if node i and j are adjacent by an edge and $A_{ij} = 0$ otherwise. $\boldsymbol{c}$ is a vector of length equal to the number of nodes in the network containing the C-P partition information. The entries in $\boldsymbol{c}$ is in $\{0, 1\}$. The n-th entry of $\boldsymbol{c}$ represents the partition of the n-th node, i.e. the c_n entry equals to 0 if the node is a periphery node and equals to 1 if it is a core node. The goal is to maximize the value of quality Q by trying different partitions $\boldsymbol{c}$. The motivation behind Eq. 2.1 is that the quality of C-P structure Q only increases when an edge is connecting core nodes. Q is also featured in the proposed algorithm. The typical methods used are approximate procedures such as heuristics, genetic algorithms and simulated annealing.

Borgatti and Everett's definition is a descriptive definition of core-periphery structure, a strictly mathematical definition has not been well proposed yet. Hereafter, various notions and detection algorithms of the core-periphery structure have been developed. Della Rossa *et al.* [4] proposed the method of profiling the core-periphery structure by specifying the behavior of a random walker. This work adopted a continuous score to measure a node's importance in the core-periphery structure. Yang and Leskovec [5] discovered that overlapping communities in community structures can construct the core-periphery structure. Xiao Zhang *et al.* [6] identified core-periphery structure by fitting a stochastic block model to empirical network data using a maximum-likelihood method. Different notations of core-periphery structures are reviewed in [7].

2.2. MULTIPLE-GROUP DETECTION

Kojaku and Masuda [8] argued that the one-block core-periphery model, which the BE algorithm and many other algorithms assume, is merely accounted for by heterogeneous degree distribution. There is a strong tendency for high-degree and low-degree nodes to be core and peripheral nodes, respectively. They proposed the third partition of residual nodes that do not belong to any significant core-periphery pair. On top of that, they developed a multiple-group detection algorithm (the KMER algorithm) [1] that focuses only on the idealized core-periphery structure, assuming the null-model of the given network is an Erdős–Rényi graph. In their study, the periphery is configured to be sparsely connected, which is assumed to be more realistic for real-world networks.

The criteria of judging the quality of detected core-periphery groups is an extension of Eq. 2.1 to the multiple-group scenario. To define a multiple-group C-P structure, the algorithm needs to know whether a node belongs to the core or periphery and to which group the node belongs. A group is defined as a subset of nodes in the network that exhibit a stronger C-P structure than other nodes added. Multiple non-overlapping subsets co-exist in the network and form the best approximation of an idealized multiple-group C-P structure. On top of C-P partition information stored in $\boldsymbol{c}$, another vector $\boldsymbol{g}$ is needed for storing the group partition information. Let G be the number of groups, $\boldsymbol{g}$ is a vector of length N in which the entries are from $\{0, 1, 2, ..., G\}$. The element g_n indicates to which group the n-th node belongs. The generalization of Borgatti and Everett's definition of C-P structure for multiple-group detection is illustrated in Eq. 2.2 and Eq. 2.3.

$$Q = \sum_{i=1}^{N} \sum_{j=1}^{N} A_{ij} B_{ij} \tag{2.2}$$

$$B_{ij}(G, C) = \begin{cases} \delta_{g_i, g_j} & (c_i = 1 \text{ or } c_j = 1 \text{ and } i \neq j) \\ 0 & (\text{otherwise}) \end{cases} \tag{2.3}$$

where δ is Kronecker delta:

$$\delta(g_i, g_j) = \begin{cases} 1 & \text{if } g_i = g_j \\ 0 & \text{if } g_i \neq g_j \end{cases}$$

The adjacency matrix representation of multiple-group C-P structure is shown in Figure 2.2. The calculation using Eq. 2.1 can be interpreted as rearranging the given network's adjacency matrix and overlapping it on the idealized adjacency matrix. When an edge is present in both matrices, the quality of the detected C-P structure will increase. On the other hand, if an edge is absent in either matrix, it will not change the quality value.

The algorithm utilizes a label-switching heuristic process to reach the maximum value of quality. This process is repeated multiple times until the algorithm maximizes the quality value Q. During each iteration, nodes are processed in a newly generated random order. For a node to do "label-switching", several tentative increment values are calculated, referring to the resulting value of switching this node's C-P partition and switching

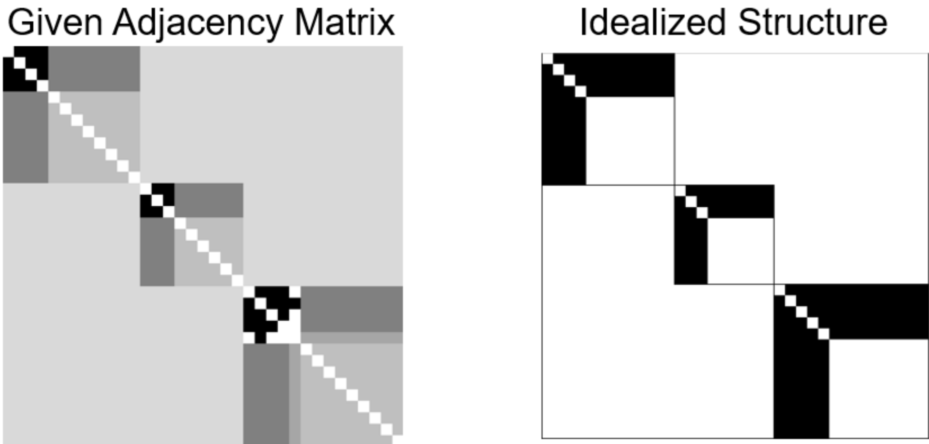


Figure 2.2: Approximate and idealized adjacency matrix
The gray level indicates the density of edges.

it into its neighboring group. Only the operation that yields the highest tentative increment for the quality Q will be adopted. The proposed algorithm will utilize a similar process accompanied by extra data processing to improve performance.

2.3. CORE-PERIPHERY STRUCTURE IN DIRECTED GRAPHS

Many methods for detecting core-periphery structures were developed for undirected networks. Although most of these algorithms can be generalized to directed graphs, they do not also generalize the definition of the discrete C-P structure as edge-direction dependent. Most algorithms either disregard the edge direction or consider the edge in each direction as an independent observation.

Andrew Elliott *et al.* [2] proposed a generalization of the block model, introduced by Borgatti and Everett, to directed networks, in which the definition of both core and periphery are edge-direction dependent. This model is implemented in the proposed algorithm in cooperation with multiple-group detection. Section 3.1 will explain the details of the block model in the work of Andrew Elliott *et al.*.

3

METHODS

In this chapter, the theory and implementation of the proposed algorithm will be discussed. The main goal of the algorithm is to accomplish two significant features: 1. Multiple-group Detection and 2. Edge-direction Dependency. The idea of combining them may seem straightforward, but it is not a trivial task. This goal is achieved through series of data processing, featuring a crucial automated process to maximize the quality of the C-P structure. A criterion extended from the discrete core-periphery structure's original definition is introduced to detect C-P structures and differentiate groups. Every implementation and modification to the mentioned methods will be explained in a temporal order following the data flow in the algorithm.

3.1. JUDGING CRITERIA, GROUP AND PARTITION

Before diving into the data processing, one should understand what notions are used throughout the algorithm.

In undirected networks, the adjacency matrix is symmetric as the edges have no specified direction. An idealized core-periphery structure can be represented in the way shown in Figure 3.1. The idealized structure features a core with full connections among each core node and a periphery that only connects to the core nodes. There should be no connection between any periphery nodes. To detect a C-P structure is to find out the C-P partitioning that resembles the idealized structure as much as possible. In comparison, directed networks have more complicated edge relations. Using the same representa-

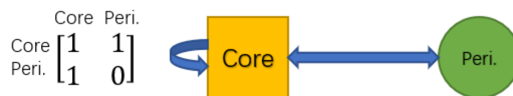
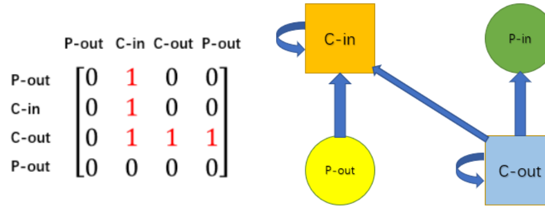


Figure 3.1: Undireted graph representation



$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 3.2: Directed graph representation

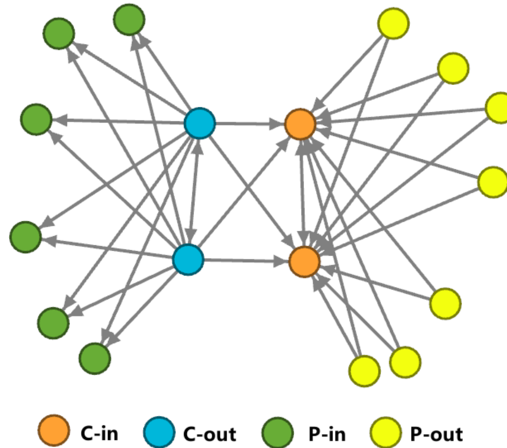


Figure 3.3: Idealized directed C-P structure

tion means disregarding the in and out edge difference, thus potentially erasing some node properties. For example, some major domains in the dark web may have huge differences in the number of in- and out- links. A wiki or directory site can have thousands of outgoing links but only a dozen incoming links. On the other hand, a large market usually has many links pointed towards it but refers to very few other sites. Disregarding the difference between in- and out -degrees will eliminate the possibility of distinguishing a node from other nodes with a similar size but different functions. In a directed graph, it is wise to separate core and periphery sets into a partition that consists of four sets: **Core-in**, **Core-out**, **Periphery-in**, **Periphery-out**. These four sets will be denoted by ***C-in***, ***C-out***, ***P-in***,***P-out*** in this report. Figure 3.3 shows an idealized C-P structure in a directed network. A similar network is tested with the algorithm and presented in Section 4.1.

In the proposed algorithm, the information of detected multiple-group C-P pairs is stored in two arrays: *Group* and *Partition*. For a network with n nodes, the length of both arrays is n. They store the information about which group and which C-P partition a node is categorized into, following its index starting from 0. The entries in *Partition* are from $\{1,2,3,4\}$ representing *C-in*, *C-out*, *P-in* ,and *P-out* respectively. The entries in *Group*

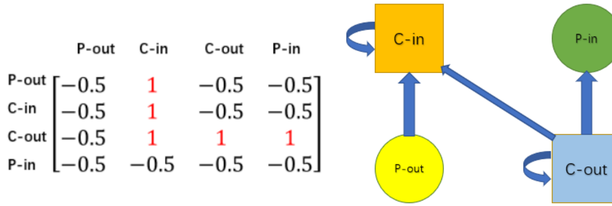


Figure 3.4: Directed graph representation with punishing values

ranges from 0 to the number of nodes according to the configuration. The number of groups can be specified or self-generated through the process. Usually, in the latter case, the indices of groups are its first member's index. In this report, by default, the number of groups is equal to the number of nodes after initialization, which means every group is put into a group with itself as the only member. After the algorithm is complete, the number of groups will be greatly decreased.

With a new partitioning scheme, the idealized adjacency matrix should be modified as shown in Figure 3.2. Note that what is depicted is now a directed network's adjacency matrix. When reading this matrix, it is important to remember that row-labeled nodes represent the source and column-labeled nodes represent the target. This model preserves the direction difference and can be used to calculate the quality value of detected groups of core-periphery structure similarly as Eq. 2.2. As a *C-in* node, it is expected to mostly receive links from the other two "out" partitions and only actively reach out to other *C-in* nodes. Therefore only the *C-in* column equals 1 in the *C-in* row. A *C-out*node is expected to link towards all partitions except *P-out*. Therefore, a *C-out*row link always increases the quality unless it connects to a *P-out* node. For *P-in* nodes, they are not supposed to establish a link towards any other nodes. Therefore, no increment should be gained if any link comes out from *P-in* are found. Finally, for *P-out* nodes, they should not link to other periphery nodes according to the definition of core-periphery structure. Therefore only links towards *C-in* will be counted towards the quality value.

This matrix directly uses the adopted quality value Q as in the BE algorithm and the KMER algorithm. The scoring in the matrix proved to be weak and led to undesirable results because the initialization is random and difficult to have a proper distribution of nodes (in groups and partitions) to kick off the process.

To resolve this problem, an enhanced scoring matrix is implemented. Now the presence of an unexpected edge will punish the quality value as presented in Figure 3.4. The punishing strength for edges connecting any C-P partitions can be configured to adjust the result. In the final version of the algorithm, the punishing value is universally set to -0.5. The choice of -0.5 is the result of a series of testing on the punishing values. All values in $[-0.9, -0.1]$ proved to be feasible while the values around -0.5 results in more core nodes

detected than values near -0.1. A punishing value near -0.9 on the other hand makes the detection result unstable. The difference between runs using punishing value -0.9 varies greatly in both C-P and group partitions. The universal punishing value of -1, as suggested by Andrew Elliott *et al.* in Ref. [2] returned too many core nodes. The comparison among different punishing values are illustrated in Figure 3.6. This settings of values in Figure 3.4 will be the default settings and applied during the testing in Chapter 4. Figure 3.5 shows the legend used in Figure 3.6, 3.7 and some other figures in Chapter 4. These networks are 200-node scale-free networks generated by networkX, using the same settings as explained in Section 4.2, and visualized with Cytoscape [9].

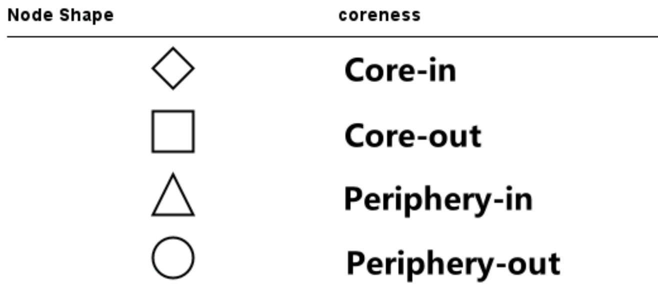


Figure 3.5: Network Visualization Legend

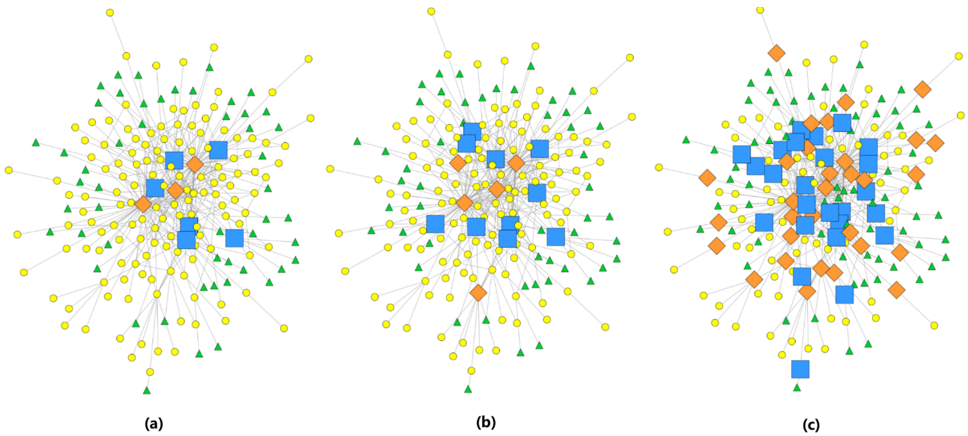


Figure 3.6: Different punishing values and results
(a)punishing by -0.1 (b) punishing by -0.4 (c) punishing by -1

For the multiple-group detection part, in the early stage of testing, an edge is disregarded when it is found connecting two different groups, and a new array *group_members* is generated for every group. Every group's members are registered in this array. The algorithm needs extra resources to store the information of each group, making the computa-

tion costly and coding difficult. The proposed algorithm introduced another punishing value for inter-group edges. The proposed algorithm now considers all groups as a whole and deducts the quality value when an edge connects two different groups regardless of the C-P partition of the two nodes this link is connecting. This change is made assuming that a group of core-periphery structures should be internally well connected but more loosely connected to external groups. The default setting of the inter-group punishing value is -0.2. This choice is made after testing. As the inter-group punishing value increases, the algorithm becomes unstable detects more groups and core nodes, as shown in Figure 3.7.

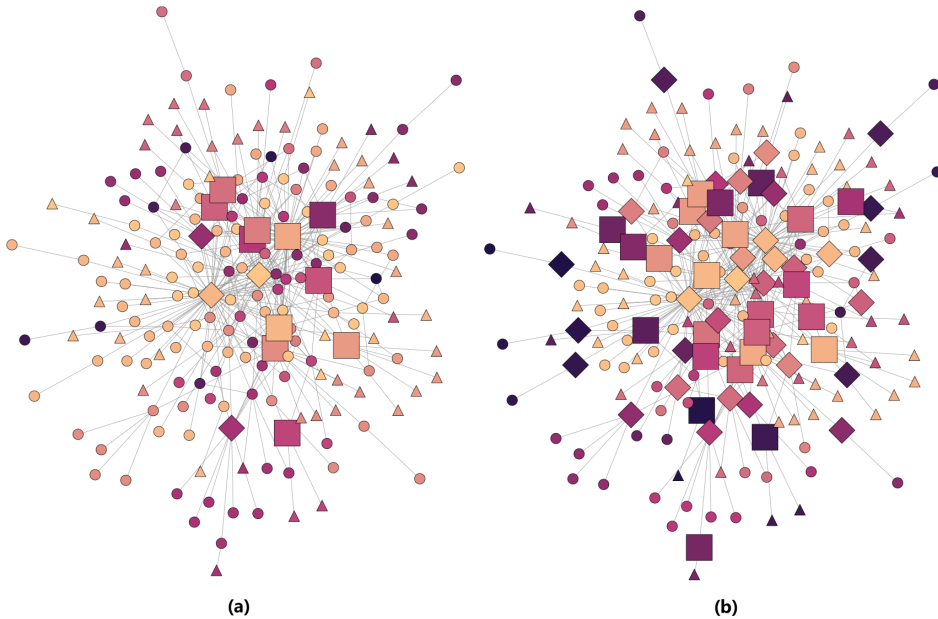


Figure 3.7: Different inter-group punishing values and results
(a)inter-group edges punished by -0.2 (a)inter-group punished by -0.5
Nodes are colored by groups

In this project, the quality value Q is normalized by P, the maximum number of possible edges in the idealized C-P structure. Essentially in Eq. 2.1 and Eq. 2.2, calculating the quality value Q is counting the edges present in both the given network's approximate adjacency matrix and the corresponding idealized adjacency matrix. *Group* and *Partition* determine the number of edges in both matrices. In directed networks, P can be calculated by:

$$P = N_1 \times (N_1 - 1) + N_2 \times (N_2 - 1 + N_1 + N_3) + N_4 \times N_1 \tag{3.1}$$

where N_1, N_2, N_3, and N_4 refer to the number of nodes in the current tentative C-P partition. $N_1 \times (N_1 - 1)$ represents the number of links within $C - in$ nodes; $N_2 \times (N_2 - 1 +$

$N_1 + N_3$) represents the number of outgoing links from $C - out$ to other three node-sets; $N_4 \times N_1$ represents the number of outgoing links from $P - out$ to $C - in$. The normalized quality value $\mathbf{Q} = \frac{Q}{P}$ ranges from -1 to 1. Before the algorithm run completes, it is possible for $\mathbf{Q}$ to decrease to a negative value because of the punishing matrix.

3.2. ALGORITHM WORKFLOW

In this section, the algorithm workflow will be presented in the same order as the data flow through the processes. The coding is done in Python. The graph is generated with a python package networkX for processing complex networks [10]. Most calculation and data manipulations in Python are done with Numpy [11] with the assistance of a JIT compiler Numba [12]. The pseudocode of the complete algorithm is shown in Algorithm 1. The variable G is the network object generated from the dataset by networkX, and *adjS* is a 1-D array of a compressed edge list. Note that the variable *adjS* is an edge list but used the starting letters of "adjacency". This variable name is used because an adjacency matrix of the given network is originally used but replaced by an edge list to improve calculation speed in the final version. This change is explained in Section 3.3

Algorithm 1: Algorithm for finding multiple-group core-periphery structure

```
1  findCP (G, adjS);
   Input  : networkX graph G, the compressed edge list matrix adjS
   Output: Group, Partition
2  detect trivial nodes, determine trivial nodes' C-P partition
3  initialize Group and Partition
4  Quality = negative infinity;
5  while True do
6  |   increment = 0;
7  |   shuffle list of nodes Node_Order;
8  |   for i in Node_Order do
9  |   |   compute tentative values for changing Group_i and Partition_i
10 |   |   get highest tentative quality newQ;
11 |   |   if newQ > Quality then
12 |   |   |   adapt the change that leads to newQ
13 |   |   |   increment = newQ - Quality
14 |   |   |   Quality = newQ
15 |   |   end
16 |   end
17 |   if increment == 0 then
18 |   |   break;
19 |   end
20 end
21 return Group, Partition
```

3.2.1. INITIALIZING AND PRE-PROCESSING

After a network is fed into the algorithm, the information will be reformed to maximize the algorithm's speed and simplify the calculation. Other than general information that can be retrieved later by passing the graph G generated by networkX, the network is stored as a compressed edge list. This matrix will be inquired repetitively in a later procedure.

All nodes with only one edge connected are regarded as trivial nodes, either in or out. All trivial nodes will be exempted from the label-switching heuristic process (explained in Section 3.2.2), put into *P-in* or *P-out* partition according to its in- or out- degrees, and remain unchanged till the end. This procedure will rule out a considerable portion of nodes in real-life networks and save the time and space required for the algorithm to complete.

All nodes except for trivial nodes will be assigned a random partition from four possibilities. If not specified, they will be assigned to a group indexed by its node index. On initialization, the graph will have several groups equal to the number of nodes.

3.2.2. LABEL-SWITCHING HEURISTIC PROCESS

The information of *Group* and *Partition*, stored in two arrays, and the edge list are passed into this phase. $Group_i$ and $Partition_i$ stores the i-th node's group and C-P partition information. Inspired by KMER, the algorithm will calculate multiple tentative values for a node and adopt the operation on either $Group_i$ or $Partition_i$ that yields the highest increment. This process is called Label-switching Heuristics.
The new random order for processing nodes will be drawn at the start of the label-switching heuristic process. In each iteration, every node except the trivial nodes will be scanned in the newly generated order. The randomly generated order can cause differences among multiple runs on the same dataset. However, the randomness of this process is also a good feature of the algorithm. If a significant core-periphery structure is present in the network, results from different runs should be structurally identical.

Two sets of possible operations are considered: 1. Switching the node to the other three node-sets, which it is currently not in, and 2. Switching the node into its neighboring groups that it is currently not in. When calculating a tentative value, the edge list will be inquired. All edges associated with these nodes will be considered and contribute to the tentative increment based on the scoring matrix. If more than one positive increment is returned, only the change that yields the highest increment will be applied into *Group* or *Partition*.

This process will be repeated until no positive tentative increment is returned for any processed node in an iteration. Upon finishing the run, the two arrays *Group* and *Partition* will be storing all changes accumulated in this run and be returned.

3.2.3. OUTCOME OF THE ALGORITHM

The outcome of a completed algorithm run is two arrays *Group* and *Partition*, and entries are the nodes' group and C-P partition in the order of their indices. They store the information about the group and the C-P partition a node belongs to. *Group* and *Partition* will participate in extra runs if required.

3.2.4. EXTRA RUNS

3

The result of a completed run is manually checked. If the results of group partitioning are not optimal, i.e., cannot provide clear aid for efficient analysis, extra runs of the algorithm can be applied to a subgraph that only contains core nodes. In this section and later part of the report, layer-n refers to the result after the n-th run of the algorithm. For example, layer-2 is the second run of the algorithm, i.e., the first "extra run".

Upon finishing the extra run, all nodes in the same group in the previous layer with a node in the current layer will be grouped into the current layer's group. When a desirable result is achieved, the algorithm casts every high layer *Group* result to its previous layer indices until it reaches its layer-1 indices.

3.3. PERFORMANCE ANALYSIS AND IMPLEMENTATION

In the final version, the number of calculations performed is based on the network's number of edges and nodes. For an iteration to complete, the algorithm will scan through n nodes and calculate $3+k$ tentative values where k is the number of neighboring groups of the scanned node. Each calculation requires the algorithm to check both nodes' core/periphery partition connected by all m edges in the network. The time complexity of this step is:

$$\mathcal{O}(mn)$$

Originally, the calculations are performed by scanning through the adjacency matrix and check if the edge is present. The quality score is adjusted according to the two nodes it is connecting. One of the main goals is to apply the algorithm on a large-scale dataset, which is presented in Section 4.3.2. A "compressed edge list" is implemented in the form of a 1-D array to boost processing speed. The elements in this array are constructed by: *element* = *(source index)* × *n* + *(target index)* where n is the total number of nodes . Both the index of source and the index of the target are arranged in ascending order. With this format of compressed edge list, the algorithm can save the space that stored zeros in the normal adjacency matrix and skip all unnecessary queries. This single data manipulation improved the speed on a 10000 edge dataset by more than 100 times.

Various tools are implemented to boost the performance further. Python is known for slow scientific calculation compared to C++. A JIT compiler for Python, Numba is used to translate Python code to faster C-like machine code. Multi-processing is implemented at the node scanning phase. The algorithm takes all neighboring groups and computes tentative increments for them simultaneously.

With the assistance of Numba, the performance of the algorithm is improved by more

than 100 times. Furthermore, with a 10000-edge, 7789-node network as the benchmark, using Numba reduced the time for completing the run from around 2000 seconds to under 30 seconds[1].

[1] Computer specification: Intel i7-9700K @ 3.60GHz, 16GB RAM @ 3200MHz, RTX2070 Super

4

RESULTS

This section showcases the algorithm's results on synthetic networks with planted C-P structure, randomly generated networks, and two real-life networks sampled from the dark web to prove the algorithm's feasibility on different types of networks. Before the test results, the result from a synthetic directed network is provided to verify the algorithm's functionality. The topological properties of these graphs will be presented and analyzed with the assistance of core-periphery structural information. Node-specific observations will also be made on the dark web datasets. Unless specified, the results presented are generated with the default settings shown in Figure 4.1 mentioned in the previous chapter. The indices of nodes are assigned by their order of appearance in the dataset. Its first member's index assigns the indices of groups.

	P-out	C-in	C-out	P-in
P-out	-0.5	1	-0.5	-0.5
C-in	-0.5	1	-0.5	-0.5
C-out	-0.5	1	1	1
P-in	-0.5	-0.5	-0.5	-0.5

Inter-group edges: -0.2

Figure 4.1: Default settings of the punishing matrix

4.1. SYNTHETIC NETWORK

Before presenting the test result on more complex and large-scale networks, the algorithm is verified by a directed network with planted C-P structure. The graph is shown in Figure 4.2. The five nodes in the center are planted core nodes with much higher in- or out- degrees compared to nodes in the periphery. The core nodes are also fully con-

nected, while the periphery nodes only connect one or two core nodes. Only very few edges are connecting two periphery nodes. The figure shows that even with some edges that differ from the idealized C-P structure, all nodes are categorized correctly into both group and C-P partition as intended.

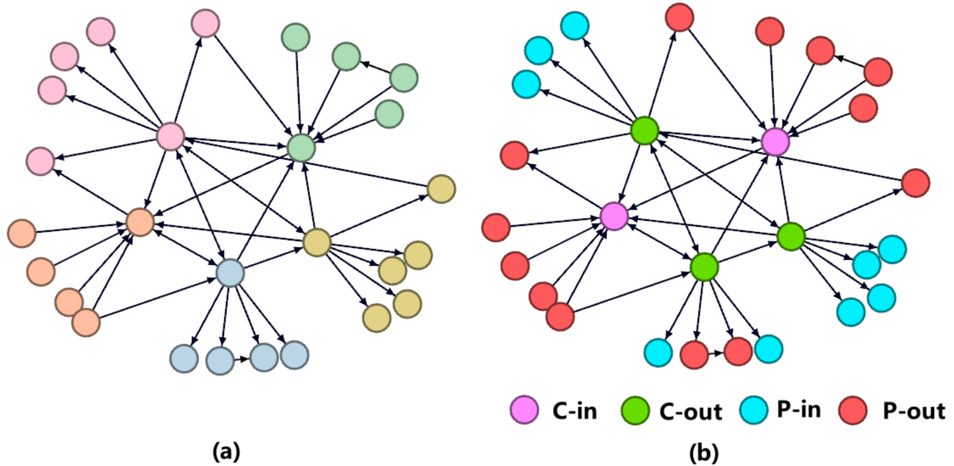


Figure 4.2: A directed network with planted C-P structure
(a) Nodes colored by group (b) Nodes colored by C-P partition

4.2. RANDOM SCALE-FREE GRAPHS

The algorithm processes three random directed scale-free networks with increasing size to verify the basic functionalities. The probability for adding a new node connected to an existing node chosen randomly according to the in-degree distribution, adding an edge between two existing nodes (one chosen according to the in-degree distribution and the other chosen randomly according to the out-degree distribution), and adding a new node connected to an existing node chosen randomly according to the out-degree distribution, are 0.2, 0.6, 0.2 correspondingly. The networks are generated by networkX using parameters: $alpha = 0.2, beta = 0.6, gamma = 0.2, seed = 123$. Self-loops and multi-edges are removed before being fed into the algorithm. The number of nodes in each network is 200, 400, and 1800 respectively. The visualization tool used is Cytoscape [9], the layout is the Prefuse Force Directed layout. The network visualization legend is shown in Figure 3.5. The periphery nodes also have a smaller size to avoid visual clutter. For conciseness, visualizations in this section use colored nodes by their group's index and modified nodes' shapes according to its C-P partition as shown in Figure 3.5. This node-shape mapping persists through every extra run layer. In this section, a new partition is introduced as residual nodes. The residual nodes are the product of extra runs. They are periphery nodes from the last layer but do not fit in any merged groups in the new layer. Such nodes can be determined as not participating in any significant C-P structure and denoted as residual nodes. In the visualization, residual nodes are colored dark and have a V-shape marker.

200-NODE NETWORK

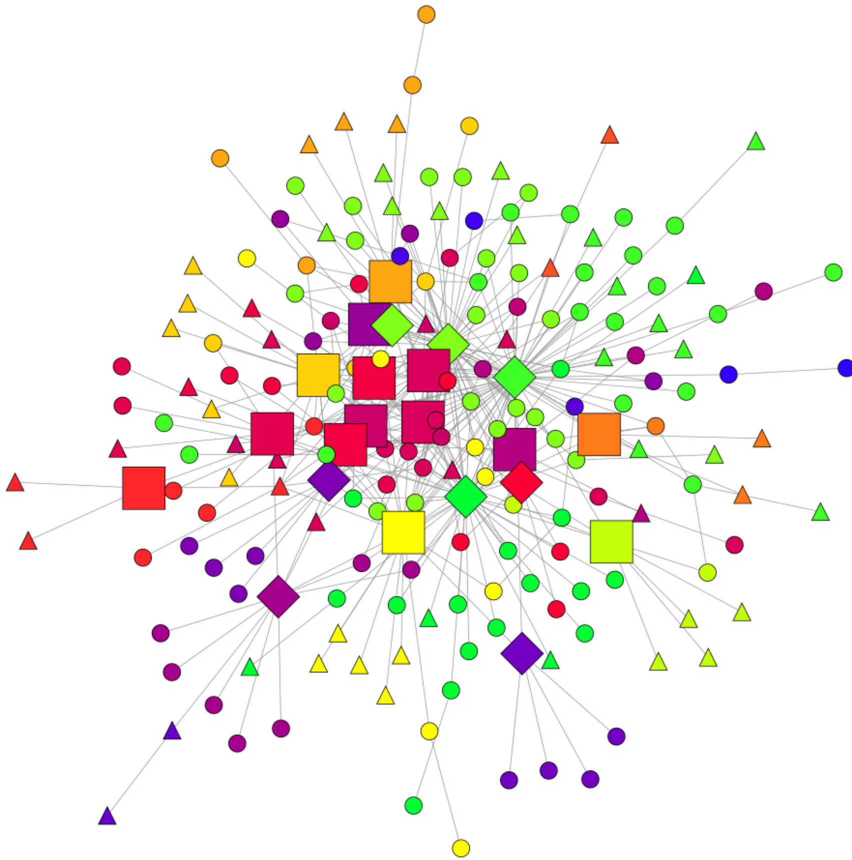


Figure 4.3: 200-node scale-free network

Figure 4.3 shows the result of core-periphery detection. After removing self-loops and multi-edges, the average out-degree of this graph is 1.78. The power-law exponent for out-degree distribution is $\alpha_{out} = 2.37$ and for in-degree distribution $\alpha_{in} = 2.50$. Among 200 nodes, 22 nodes(8 $C-in$ and 14 $C-out$) are categorized as core nodes. There are two groups containing 2 $C-out$ nodes and one group containing 2 $C-in$ nodes. All other core nodes are partitioned along with their attached periphery nodes. The core has much stronger connections than the periphery.

400-NODE NETWORK

The results from the 400-Node scale-free network is shown in Figure 4.4. After removing self-loops and multi-edges, the average out-degree of this graph is 1.82. The power-law exponent for out-degree distribution is $\alpha_{out} = 2.47$ and for in-degree distribution $\alpha_{in} = 2.57$. As the number of nodes doubles, the number of core nodes increased from 15 to 37. The nodes are colored by their group index with a continuous mapping. Groups

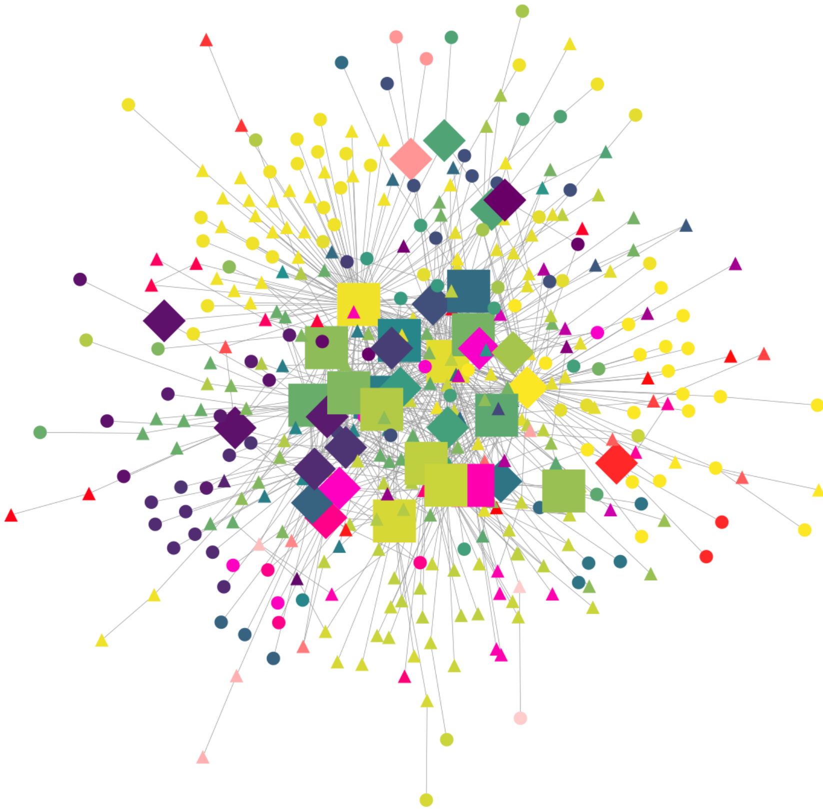


Figure 4.4: 400-node scale-free network

with similar color are likely to be merged in extra runs.

Figure 4.5 shows the result from layer-2. There are 7 $C-in$ nodes but 0 $C-out$ nodes. All 7 $C-in$ nodes are in the same group, forming a strong core-periphery structure with their inherited periphery nodes from last layer.

1800-NODE NETWORK

The algorithm detected 22 C-in and 15 $C-out$ nodes. After removing self-loops and multi-edges, the average out-degree of this graph is 1.98. The power-law exponent for out-degree distribution is $\alpha_{out} = 2.35$ and for in-degree distribution $\alpha_{in} = 2.62$. With a much bigger jump in the number of nodes, the number of core nodes did not grow proportionally but only up to 104 (62 $C-in$ and 42 $C-out$). At this stage, the number of groups and core nodes are already difficult for efficient analysis. Extra runs should be applied to improve the crude visualization. The result from layer-2, as shown in Figure 4.6 greatly reduced the number of core nodes to 16. All 16 core nodes are in the same group. It proved that this network has a very dense core. Layer-3, shown in Figure 4.7 further re-

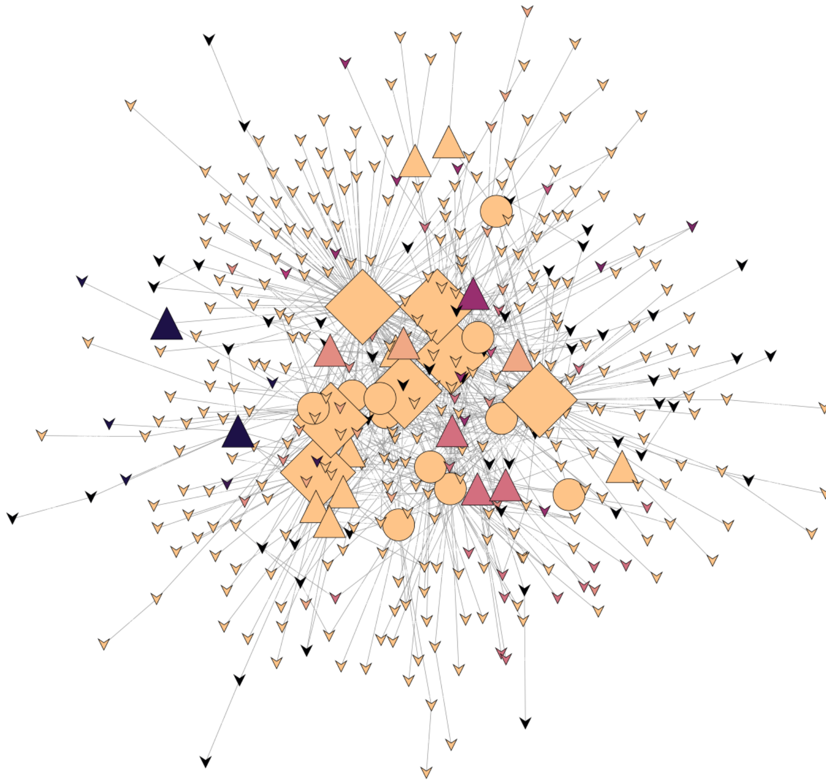


Figure 4.5: 400-node scale-free network on layer-2
Nodes colored by groups

duced the number of core nodes to 3, highlighting the three most important core nodes in the network. Most periphery nodes from layer-1 and layer-2 are carried into layer-3 as non-residual nodes. They are colored yellow, the same as the biggest group.

REMARKS

The algorithm proves to work as intended. Nodes are correctly categorized according to their in/out-degree and relation to other nodes. However, although the connection among core nodes is not weak, the algorithm still struggles to merge them. This is most likely caused by the default settings' assumption that groups should be separable from each other, represented by weak connections between groups regardless of nodes' C/P partition. A workaround is to use extra runs on higher layer subgraphs. This method is demonstrated in Section 4.3.2.

4.3. DARK WEB DATASET

In-depth analysis is conducted on the dataset provided by CFLW Cyber Strategies. This section will first introduce the dataset format and general information about the sam-

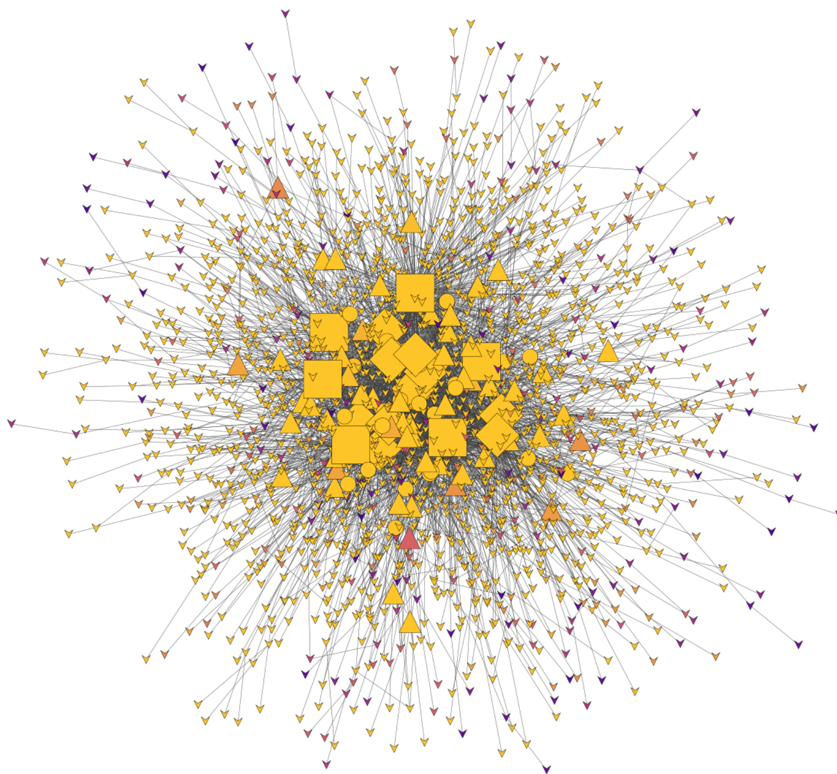


Figure 4.6: 1800-node scale-free network on layer-2
Nodes colored by groups

pled network, then presents the analysis done on the dataset with the knowledge of multiple-group core-periphery structure.

Because of the limit of Cytoscape, it was not easy to smoothly process the graph with such an enormous size. In this section, the visualization and analysis are done with the assistance of Gephi. Therefore some features are missing. Expressly, the nodes' shape is limited to be circular.

4.3.1. DATASET

The dataset is a collection of 100,001 randomly sampled relations in the dark web. Each relation is a directed link from Domain #1 to Domain #2. For each relation, both domain's information is presented as follows: domain index, domain address, domain title, and timestamp. In this report, the temporal properties are not studied. The first five relations are shown in Table 4.1. The network is connected and contains 49294 unique domains in the dark web as nodes and 100,001 directed relations as edges. The average degree (both in- and out- included) is 2.029. The network diameter is 22, and the average path

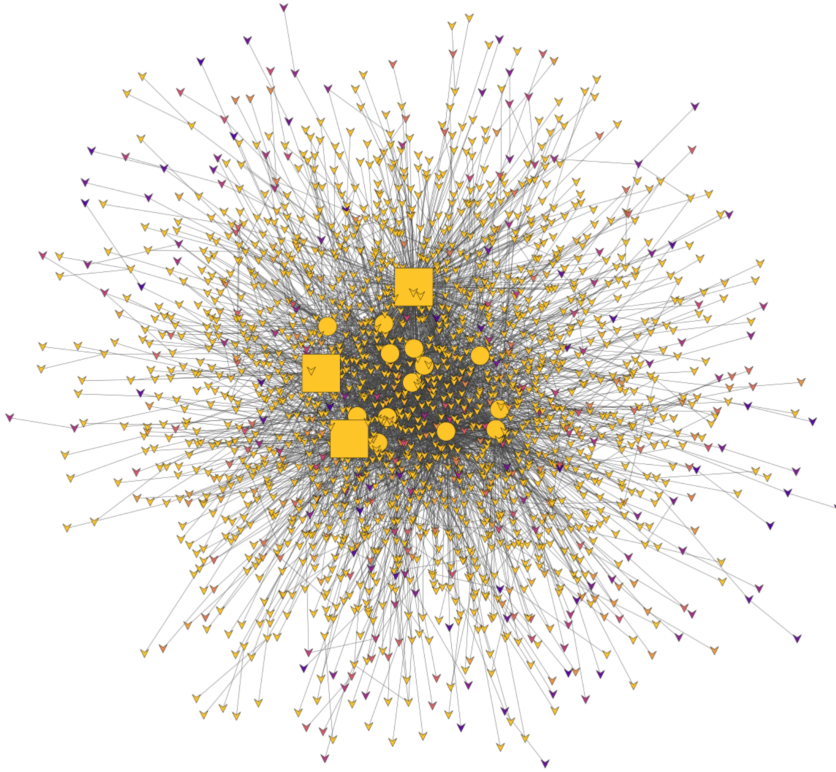


Figure 4.7: 1800-node scale-free network on layer-3
Nodes colored by groups

length is 9.39, which is double the average path length of the real dark web [13], [14]. The sampled relations are more spread to ensure randomness than a real-word dark web, i.e., the difference between nodes with the highest degrees is smaller. For example, node 22 has the highest degree of 946. Following 22, node 46 and 278 each has 906 and 811 links. Compared to another dark web dataset that values the width of sampling, which crawled 10000 links in total on domains exhaustively, a dominant node has 5500 edges while the second biggest one only has 650.

4.3.2. CORE-PERIPHERY ANALYSIS

The run-time for detecting the multiple-group core-periphery structure in this network is around 60 minutes (3816 seconds). In comparison, to process 10% of the network (10000 edges, 11028 nodes), the algorithm only spent 31 seconds.

Among 49294 nodes, only 0.73% are categorized as core nodes (0.41% of Core-in and 0.32% Core-out). 13.33% of the nodes are Periphery-out. An enormous portion of 85.94% nodes is Periphery-in. The type of domains is very distinguishable by checking its C/P

ID Domain # 1	Domain Address # 1	Domain title # 1	ID Domain # 2	Domain Address # 2	Domain title # 2
151183	http://t63y6g....onion	Onion Links	920	http://4yjes6....onion	login | Drugmarket
298607	http://c3b24j....onion	Darknet Tor - Wiki Tor	19279	http://vrm4fr....onion	GlobaLeaks
865087	http://urcjjn....onion	Scam List of Tor	54435	http://galaxy....onion	CC Galaxy is A...
336225	http://kyoyvb. . . .onion	Scam List of Tor	59417	http://bitcar....onion	BITCARDS - Home
336294	http://3bpfr2....onion	Onion Link List...	41637	http://r26lia....onion	The CC Buddies...

Table 4.1: First Five relations in the dataset

partition. C-in and $P - in$ nodes are usually market or service domains. $C - out$ and $P - out$ nodes are usually wiki/directory link list sites. The first run detected 360 core

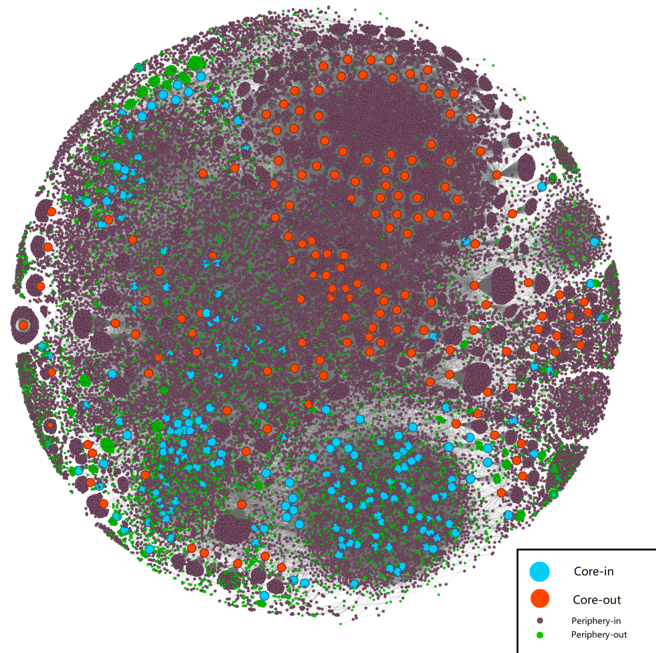


Figure 4.8: Core-periphery structure View colored by core/periphery partition

nodes and 7328 groups, with only 32 groups each having more than 0.5% percent of the network's population(from 1.76% to 2.21%). The largest group is #22, with node #22 itself, having the highest degree, in- and out- added up to 946, in this network. One property of the dark web is that many large-degree nodes have most of their descendants only connected to themselves. Secondly, the core nodes are mostly disconnected from each other. In this network, big $C - out$ nodes occupy the majority of links but rarely connect. By definition, core nodes should strongly connect with other core nodes, while periphery nodes should only connect to the cores. To make different groups more distinguishable, the algorithm is configured to discourage links (punish inter-group links regardless of C-P partition). It is difficult for the algorithm to partition multiple core nodes in a single group through one run because of these two properties. Most of the formed groups

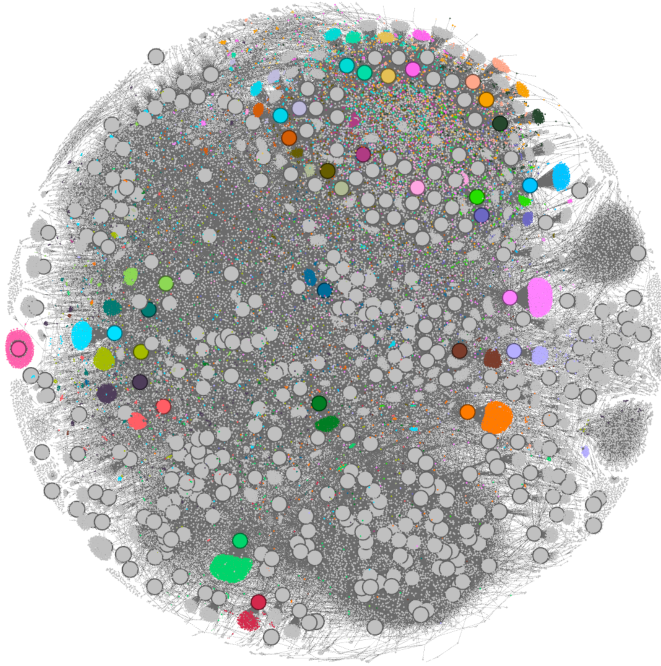


Figure 4.9: Top 32 groups

consist of only one core node and its descendants in a large number. The desired result should be multiple core nodes partitioned together in a large group. Looking at Figure 4.8, a few sets of core nodes located closely together in this layout are observable: (1): A set of Core-in nodes on the top-left sector, each having a large number of trivial $P-out$ descendants. (2): A set of Core-out nodes on the top-right sector, each having a large number of trivial $P-in$ descendants. Also, a large set of $P-in$ nodes, surrounded by Core-out nodes, are connected to multiple Core-out nodes. (3): A set of Core-in nodes on the bottom-right sector, sharing a large set of mixed-type periphery nodes.

A subgraph is extracted from the network by only selecting the core nodes, there are 360 such nodes, and 71 of them are isolated. Both the number of groups and the number of core nodes are still too large for efficient analysis after only one algorithm run. To improve the result, there are two options. One is by tuning the punishing matrix (decrease the punishing value of inter-group edges connecting core nodes) to encourage core nodes to group together. However, this option is expected to have a slight effect because of the dominant number of trivial nodes and very sparse connections among core nodes. The second option is to perform an extra run on the networks but only include core nodes. From this small core-node-only network, a higher layer of core-periphery structure can be extracted. Lesser core nodes will be categorized as periphery nodes in additional runs and re-partitioned into a higher level C-P group along with all its periphery nodes. Additional runs can be performed multiple times until the core node-set

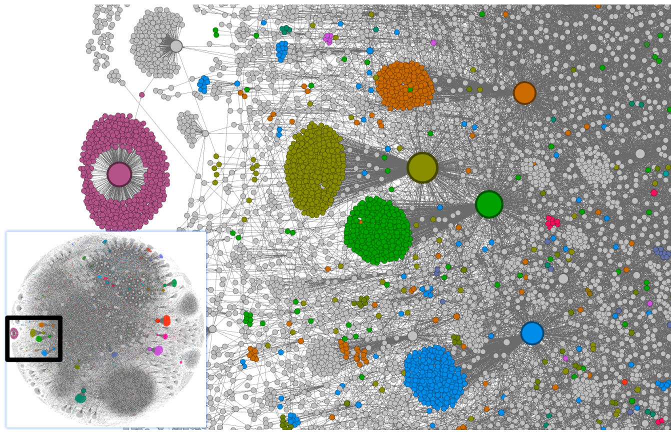


Figure 4.10: Several groups of core-periphery nodes

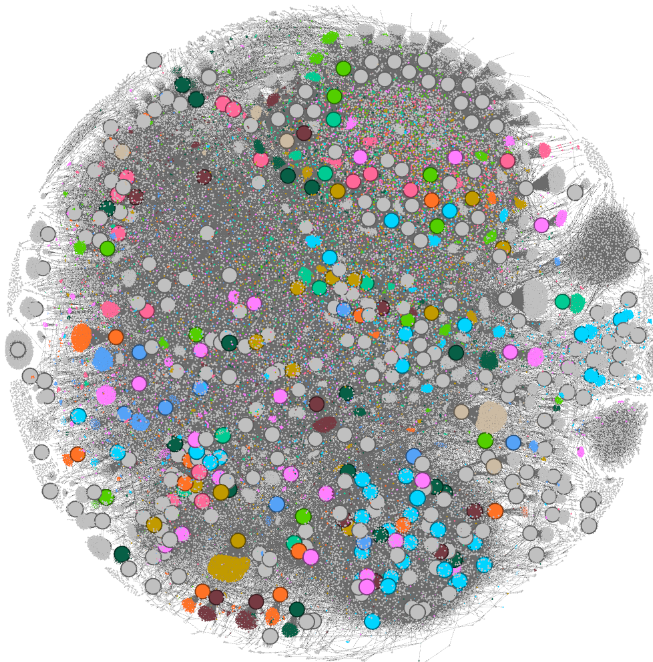


Figure 4.11: 11 largest groups after 4 extra runs

is refined to a small size, highlighting only the most important core nodes. The algorithm can accumulate a score through multiple runs each time a node remains in the core node-set. Eventually, the most important core nodes will have the highest score.

After four extra runs, the algorithm successfully partitioned smaller groups together. There are now 11 groups with more than 2% of the population (from 2.21% to 4.05%). The former largest group #22 is now merged into group #38, the new largest group. Five groups stood out: #38 (4.05%), #256 (3.85%), #752 (3.37%), #3758 (2.95%), and #212 (2.88%). Group #38 has fewer core nodes than others but has the core node #22 with the largest degree 946. Node #38 is a $C-out$ node titled "DeepLink Onion Directory". Group #256 contains a large portion of node-set (3) mentioned above. These $P-in$ nodes are related together by their mutual $P-out$ neighbors, Wiki/Directory sites with very low degrees in this sampled network. A few $C-out$ nodes are also present. They are connected to the $C-in$ cluster by three $C-out$ hub nodes #205, #215, and #1063. All three nodes are link list sites. There is always at least one hub node in each group. They are node #415 "The Deep Searches" in group #256, node #974 "CB3ROB Tactical Data Services - TOR Darknet site listing" in group #3758, and node #496 "DarkNet 2020 - Wiki Links Tor" in group #212.

Only four nodes survived through 3 extra runs and remained in layer-4 as core nodes: #256 ($C-in$, group #256), #752 ($C-in$, group #752), #2417 ($C-in$, group #752), and #212 ($C-out$, group #362). According to the information provided by the core-periphery structure, #212 is the one with the most value to investigate. The egocentric network of node #212 is shown in Figure 4.13. Node #212 is highlighted because it is the only $C-out$ that survived four rounds of filtering and stayed in the same core/periphery partition as in layer 1. This egocentric network contains 15 core nodes from layer-1 (7 $C-in$ and 8 $C-out$) including the largest $C-out$ node #22. Node #212 is titled "CB3ROB Tactical Data Services - TOR Darknet site listing" which appears to be a darknet service directory site. Node #212 connects to 7 different layer-4 groups including the 4 out of 5 biggest groups, serving as the most important central hub.

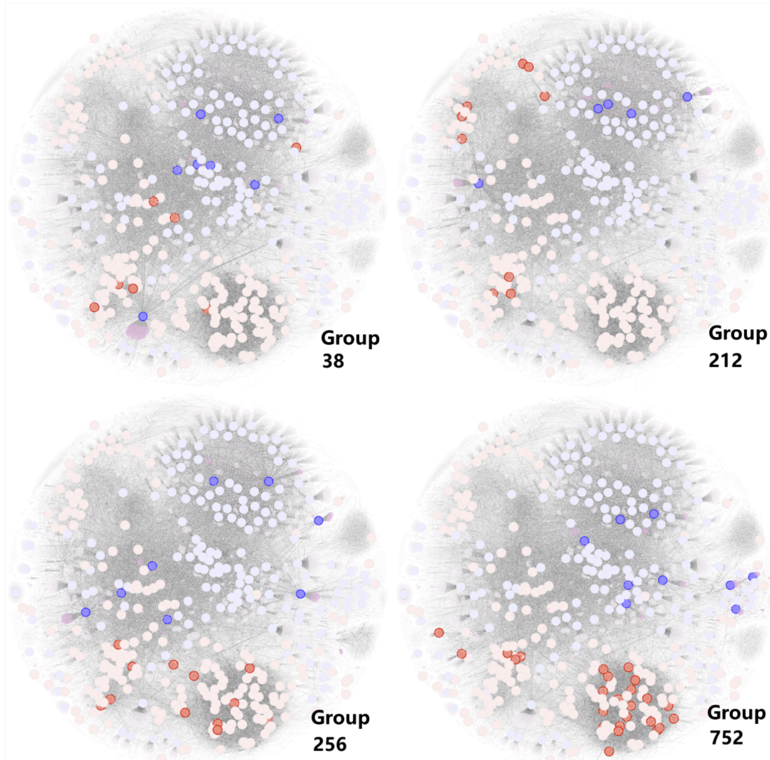


Figure 4.12: Four major groups after 4 extra runs; Red:C-in Blue:C-out

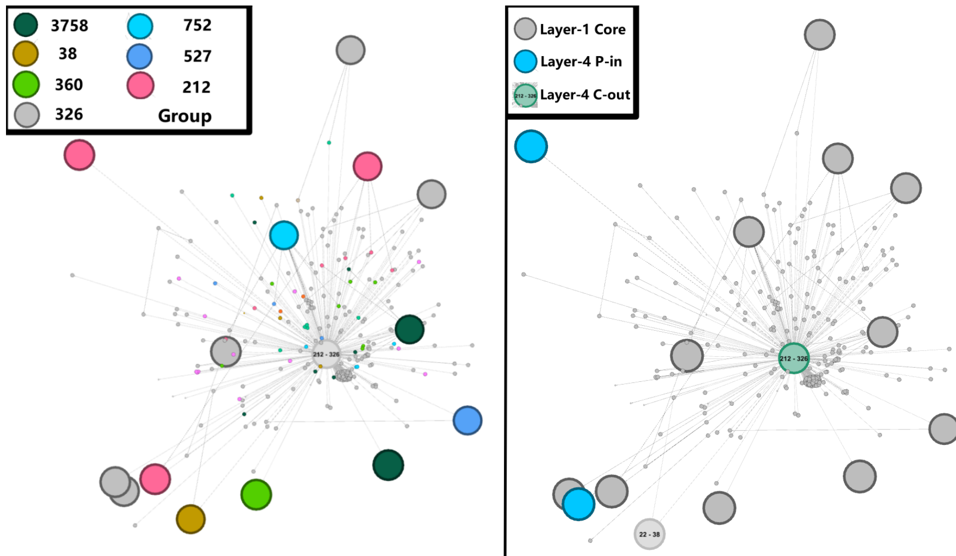


Figure 4.13: Egocentric network of node #212;
Left: Layer-4 group view Right: Layer-4 C/P view

5

Discussion

Discussion and Future Work

The tests on different networks showed satisfying results, especially on the 100000 edges dark web sample network. However, some flaws in my approach do exist. The core/periphery partitioning proves to be accurate with all service/market and link list sites categorized into $C - in$ and $C - out$, respectively. The in and out partitions enable us to differentiate nodes' function, which is an excellent addition to the core-periphery structure analysis. However, the group partitioning does not return desired results on its own. Through only one run, one group almost always only includes one core node and its trivial peripheries. The result can be enhanced by doing extra runs, but finding better punishing values, especially for inter-group edges, is a more beneficial approach. Right now, the algorithm uses a universal punishing value for all inter-group edges. A way to improve this is to check if such edges connect core nodes. If so, a lighter or zero punishment could be adapted to encourage different groups to merge through links between their core nodes. Due to the limit of time and lack of experience, I could not fully explore this path. In continued work, people could follow this direction.

On a brighter note, this algorithm's final version became around 10000 times faster than its first functionally complete version. This accomplishment is the combined results of using pre-processing, multi-processing, data manipulation, and compiling with Numba. Points for improvement still exist, and I have already noticed two. The level of parallelism is relatively low, and more powerful tools like graphic card computation (CUDA) are not utilized. Additionally, the final version of the algorithm still scans through every node present in the network. A possible improvement is only to scan the edges that involve the label-switched node. However, this increases code complexity significantly and is not easily compatible with the compiling tool Numba, which single-handedly improves the performance by a factor of 100. Further increasing the speed has the potential to enable analysis on much larger scale networks.

This project was originated from a summer project I participated in, organized by the TU

Delft NAS Group and CFLW Cyber Strategies, in the summer of 2020. In the beginning, I only used existing python libraries to do core-periphery detection on a smaller dataset representing an undirected network. However, I am happy that I can eventually accomplish the goal of implementing multiple-group core-periphery detection into directed graphs.

CONCLUSION

This thesis proposed an algorithm that can detect multiple groups of core-periphery node pairs in directed graphs. This algorithm utilizes an extended version of the core-periphery structure's definition in the form of a "punishing matrix" that can make the detection edge-direction dependent. To divide nodes into different groups, a inter-group punishing value is added to the process. Through a label-switching heuristic process, the algorithm updates each node's core/periphery or group partitions in multiple iterations, eventually reaching the maximum quality of the multiple-group core-periphery structure. A series of data manipulation and tools are used to improve the performance. The algorithm can process large-scale networks within an acceptable amount of time. The results can be used to aid network analysis. From the perspective of core-periphery structure, the information extracted can uncover important nodes that other methods may not highlight. This method for detecting multiple-group core-periphery structures in directed graphs is highly configurable and can enable a more advanced approach in network analysis.

Bibliography

[1] Sadamori Kojaku and Naoki Masuda. "Finding multiple core-periphery pairs in networks". In: *Physical Review E* 96.5 (Nov. 2017). ISSN: 2470-0053. DOI: 10.1103/physreve.96.052313. URL: http://dx.doi.org/10.1103/PhysRevE.96.052313.

[2] Andrew Elliott et al. "Core-Periphery Structure in Directed Networks". In: *CoRR* abs/1912.00984 (2019). arXiv: 1912.00984. URL: http://arxiv.org/abs/1912.00984.

[3] Stephen Borgatti and Martin Everett. "Models of Core/Periphery Structures". In: *Social Networks* 21 (Nov. 1999), pp. 375–395. DOI: 10.1016/S0378-8733(99)00019-2.

[4] F. D. Rossa, F. Dercole, and C. Piccardi. "Profiling core-periphery network structure by random walkers". In: *Scientific Reports* 3 (2013).

[5] Jaewon Yang and Jure Leskovec. "Structure and Overlaps of Ground-Truth Communities in Networks". In: *ACM Trans. Intell. Syst. Technol.* 5.2 (Apr. 2014). ISSN: 2157-6904. DOI: 10.1145/2594454. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/2594454.

[6] Xiao Zhang, Travis Martin, and M. Newman. "Identification of core-periphery structure in networks". In: *Physical review. E, Statistical, nonlinear, and soft matter physics* 91 (Sept. 2014). DOI: 10.1103/PhysRevE.91.032803.

[7] P. Csermely et al. "Structure and dynamics of core/periphery networks". In: *Journal of Complex Networks* 1.2 (Oct. 2013), pp. 93–123. ISSN: 2051-1329. DOI: 10.1093/comnet/cnt016. URL: http://dx.doi.org/10.1093/comnet/cnt016.

[8] Sadamori Kojaku and Naoki Masuda. "Core-periphery structure requires something else in the network". In: *New Journal of Physics* 20 (Oct. 2017). DOI: 10.1088/1367-2630/aab547.

[9] Paul Shannon et al. "Cytoscape: a software environment for integrated models of biomolecular interaction networks". In: *Genome research* 13.11 (2003), pp. 2498–2504.

[10] Aric Hagberg, Pieter Swart, and Daniel S Chult. *Exploring network structure, dynamics, and function using NetworkX*. Tech. rep. Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[11] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[12] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. "Numba: A LLVM-Based Python JIT Compiler". In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. LLVM '15. Austin, Texas: Association for Computing Machinery, 2015. ISBN: 9781450340052. DOI: 10.1145/2833157.2833162. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/2833157.2833162.

[13] Jennifer Xu and Hsinchun Chen. “The Topology of Dark Networks”. In: *Commun. ACM* 51.10 (Oct. 2008), pp. 58–65. ISSN: 0001-0782. DOI: 10.1145/1400181.1400198. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/1400181.1400198.

[14] Manlio Domenico and Alex Arenas. “Modeling Structure and Resilience of the Dark Network”. In: *Physical Review E* 95 (Dec. 2016). DOI: 10.1103/PhysRevE.95.022313.

Appendix

The main body of the proposed algorithm is provided in this Appendix.
Also included is code for the generation of the used scale-free networks and or importing data from the dark web dataset. The code can be used after uncommenting it. For code of extra runs and saved files please refer to the repository: https://github.com/lhawx0/multiDiCP
Details of included files are provided in the repository.

Listing 1: Insert code directly in your document

```
from concurrent.futures import ThreadPoolExecutor
import networkx as nx
import matplotlib.pyplot as plt
import math
import csv
import numpy as np
import pandas as pd
from numba import njit

@njit
def calculate_Q_reverse(adjS,partition_old,group_old,target,par_new,group_new):
    """Calculate the quality value of given network, done by going through all the edges

    Args:
        adjS (numpy.array): the compressed edge list in 1-d array form
        partition_old (numpy.array): The old partition array
        group_old (numpy.array): The old group array
        target (integer): the index of scanned node
        par_new (integer): the tentative new partition of target
        group_new (integer): the tentative new group of target

    Returns:
        [float]: the quality value Q
    """
    partition = np.copy(partition_old)
    group = np.copy(group_old)
    partition[target],group[target] = par_new,group_new
    hit = 0
    num_Cin = np.count_nonzero(partition == 1 )
    num_Cout = np.count_nonzero(partition == 2 )
    num_Pin = np.count_nonzero(partition == 3 )
    num_Pout = np.count_nonzero(partition == 4)
    number_nodes = len(partition)

    for sidx in adjS:
        i, j = divmod(sidx,number_nodes)
        if group[i] == group[j]:
            if partition[i] == 1:
                if partition[j] == 3 or partition[j] == 4:
```

```
                    hit -= 0.5
                elif partition[j] == 1: hit += 1
                elif partition[j] == 2: hit -= 0.5
            elif partition[i] == 2:
                if partition[j] == 2: hit -= 0.5
                else: hit += 1
            elif partition[i] == 3:
                hit -= 0.5
            elif partition[i] == 4:
                if partition[j] == 1: hit += 1
                else: hit -= 0.5

        else:
            hit -= 0.2

    max_hit = num_Cin*(num_Cin - 1) + num_Cout *(num_Cout - 1 + num_Cin +num_Pin) + num_Pout*num_
    if max_hit == 0:
        Q = - math.inf
        return Q
    else:
        Q = hit/max_hit

    return Q

def find_CP_M4(adjS,G):
    """find the multiple-group core-periphery structure in a directed network

    Args:
        adjS (numpy.array): the compressed edge list in 1-d array form
        G (netxorkX network object): the imported directed network


    Returns:
        group,partition (numpy.array, numpy.array): the multiple-group core-periphery structure in
    """

    number_of_nodes = G.number_of_nodes()
    remain = []
    remove = []
    for node,degree in dict(G.degree()).items():
        if degree >=2:
            remain.append(node)
        elif degree<2:
            remove.append(node)

    remain = np.array(remain)
    remove = np.array(remove)
    belongs = {}
    for node in remove:
        for neib in nx.all_neighbors(G,node):
            belongs[node] = neib

    owns = {}
    owning = {}
    for node,belon in belongs.items():
        if belon in owns:
            owns[belon].append(node)
```

```
            owning[belon] += 1
        else:
            owns[belon] = [node]
            owning[belon] = 1

owns_list = [np.array(owns[i]) for i in owns]

max_len = 0
for li in owns_list:
    if len(li) > max_len:
        max_len = len(li)

for li in owns_list:
    if len(li) < max_len:
        zero = np.zeros(max_len - len(li))
        li = np.concatenate((li,zero))

owns_list = np.array(owns_list)
trivial_partition = {}
signi_partition = {}
for node in remove:
    if G.in_degree(node) == 1:
        trivial_partition[node] = 3
    elif G.out_degree(node) == 1:
        trivial_partition[node] = 4
    elif nx.all_neighbors(G,node) == 0:
        pass
    else:
        raise ValueError("removed node degree is not 1")

for node in remain:
    ins = G.in_degree(node)
    outs = G.out_degree(node)

    if ins > outs:
        signi_partition[node] = 1
    elif ins < outs:
        signi_partition[node] = 2
    else:
        signi_partition[node] = np.random.randint(1,5)


partition = np.zeros(number_of_nodes)
for idx in range(number_of_nodes):
    if idx in trivial_partition:
        partition[idx] = trivial_partition[idx]
    elif idx in signi_partition:
        partition[idx] = signi_partition[idx]
    else:
        raise ValueError("idx does not exist")

group = np.arange(number_of_nodes)

for owner,owned in owns.items():
    group[owned] = owner

order = np.array(remain)
```

```
print("run_started ...")
Q = - math.inf
iters = 0
last_incre = 1
while(True):
    iters += 1
    np.random.shuffle(order)
    incre = 0
    for i in order:
        new_Q = Q
        diff = 0
        par_old = partition[i]
        par_new = par_old
        if par_old !=1:
            par_Q = calculate_Q_reverse(adjS, partition, group, i, 1, group[i])
            if par_Q> new_Q:
                par_new = 1
                new_Q = par_Q
        if par_old !=2:
            par_Q = calculate_Q_reverse(adjS, partition, group, i, 2, group[i])
            if par_Q > new_Q:
                par_new = 2
                new_Q = par_Q
        if par_old !=3:
            par_Q = calculate_Q_reverse(adjS, partition, group, i, 3, group[i])
            if par_Q > new_Q:
                par_new = 3
                new_Q = par_Q
        if par_old !=4:
            par_Q = calculate_Q_reverse(adjS, partition, group, i, 4, group[i])
            if par_Q > new_Q:
                par_new = 4
                new_Q = par_Q
        nbs = [e for e in nx.all_neighbors(G, i)]
        to_groups = set(np.take(group, nbs))
        base_Q = Q
        switch_to = -1

        def wrapperfunc(target_group):
            return target_group, calculate_Q_reverse(adjS, partition, group, i, partition[i], targe
        processes = []
        with ThreadPoolExecutor() as ex:
            processes.append(ex.map(wrapperfunc, to_groups))

        for results in processes[0]:
            temp_Q = results[1]
            to = results[0]
            if temp_Q >= base_Q:
                base_Q = temp_Q
                switch_to = to

        if base_Q > new_Q and switch_to != -1:
            # the result of group-switching is better
            diff = base_Q - Q
            Q = base_Q
            group[i] = switch_to
            if i in owns:
```

```
                group[owns[i]] = switch_to
            elif new_Q > base_Q and par_new != par_old:
                # the result of partition switching is better
                diff = new_Q - Q
                Q = new_Q
                partition[i] = par_new
                # update hit/max

            incre += diff

        print("still running: " + str(iters))
        if (incre <= 10**-10 and last_incre <= 10**-10  and iters >= 4):
            print("iters:"+str(iters))
            print("incre:"+str(incre))
            break
        last_incre = incre

    print("Final Q:" + str(Q))
    return partition,group

def main():

    ##########################################################################################
    ##   Importing the dataset and generating the networkX object G
    # df = pd.read_excel("DATAEXTRACT.xlsx", skiprows = 0, dtype='int64')
    # e_list = df.values.tolist()
    # map_list = {}
    # index = 0
    # for pairs in e_list:
    #     if pairs[0] not in map_list:
    #         map_list[pairs[0]] = index
    #         index += 1

    #     if pairs[1] not in map_list:
    #         map_list[pairs[1]] = index
    #         index += 1

    # for pairs in e_list:
    #     pairs[0] = map_list[pairs[0]]
    #     pairs[1] = map_list[pairs[1]]

    # G = nx.DiGraph()
    # G.add_edges_from(e_list)

    ##########################################################################################


    ##########################################################################################
    #  Generating random scale-free network; the first argument: of number of nodes.
    F = nx.scale_free_graph(200,alpha=0.2,beta=0.6,gamma=0.2,seed = 123)
    G = nx.empty_graph(F.number_of_nodes(),create_using=nx.DiGraph())
    ed_list = []
    for e in F.edges():
        if e not in ed_list and e[0] != e[1]:
            ed_list.append(e)
    G.add_edges_from(ed_list)
```

```
####################################################################################################

####################################################################################################
## plotting the network with matplotlib, not used
# pos = nx.layout.spring_layout(G)

# node_sizes = [3 + 10 * i for i in range(len(G))]
# M = G.number_of_edges()
# edge_colors = range(2, M + 2)

# nodes = nx.draw_networkx_nodes(G, pos, node_size=node_sizes, node_color="blue")
# edges = nx.draw_networkx_edges(
#       G,
#       pos,
#       node_size=node_sizes,
#       arrowstyle="->",
#       arrowsize=10,
#       edge_color=edge_colors,
#       edge_cmap=plt.cm.Blues,
#       width=2,
# )
# labels = nx.draw_networkx_labels(G,pos)

# pc = mpl.collections.PatchCollection(edges, cmap=plt.cm.Blues)
# pc.set_array(edge_colors)
# plt.colorbar(pc)

# ax = plt.gca()
# ax.set_axis_off()
# plt.show()
####################################################################################################


####################################################################################################
## generating the compressd edge list
num_nodes = G.number_of_nodes()
num_edges = G.number_of_edges()
print(num_nodes)
print(num_edges)
print(num_edges/num_nodes)
adjS = np.empty(num_edges,dtype=np.uint32)
idx = 0
for node in G.nodes():
    for neib in sorted(G.neighbors(node)):
        adjS[idx] = num_nodes * node + neib
        idx += 1

# start the run
part,group= find_CP_M4(adjS,G)

#compact index of groups
group_map = {}
group_map_re = {}
idx = 0
for g in group:
    if g not in group_map:
```

```
            group_map[g] = idx
            group_map_re[idx] = g
            idx += 1
    for i in range(len(group)):
        group[i] = group_map[group[i]]

    # save result for extra runs and visualization
    nx.write_gml(G, "FINAL200_4.gml")
    with open('FINAL200_4.csv', mode='w', newline='') as export_file:
        export_writer = csv.writer(export_file, delimiter=',', quotechar='"', quoting=csv.QUOTE_MINIMAL)
        export_writer.writerow(['Id','coreness','group'])
        for node in G.nodes:
            export_writer.writerow([node, part[node], group[node]])

    print(part)
    print(group)

if __name__ == "__main__":
    main()
```