We approve the thesis of Mohammad Adi.

Date of Signature

Dr. Thang N. Bui Associate Professor of Computer Science Chair, Computer Science and Mathematics Programs Thesis Adviser	
Dr. Linda M. Null Associate Professor of Computer Science Computer Science Graduate Coordinator	
Dr. Jeremy J. Blum Associate Professor of Computer Science	
Dr. Sukmoon Chang Associate Professor of Computer Science	
Dr. Omar El Ariss Assistant Professor of Computer Science	

The Pennsylvania State University
The Graduate School

USING ANTS TO FIND COMMUNITIES IN COMPLEX NETWORKS

A Thesis in
Computer Science
by
Mohammad Adi

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

May 2014

The thesis of Mohammad Adi was read and approved* by the following:

Dr. Thang N. Bui
Associate Professor of Computer Science
Chair, Computer Science and Mathematics Programs
Thesis Adviser

Dr. Linda M. Null
Associate Professor of Computer Science
Computer Science Graduate Coordinator

Dr. Jeremy J. Blum
Associate Professor of Computer Science

Dr. Sukmoon Chang
Associate Professor of Computer Science

Dr. Omar El Ariss
Assistant Professor of Computer Science

*Signatures on file in the Graduate School.

Abstract

Many systems arising in different fields can be described as complex networks, a collection of nodes and edges connecting nodes. An interesting property of these complex networks is the presence of communities (or clusters), which represent subsets of nodes within the network such that the number of edges between nodes in the same community is large whereas the number of edges connecting nodes in different communities is small. In this thesis, we give an ant-based algorithm for finding communities in complex networks. We employ artificial ants to traverse the network based on a set of rules in order to discover a "good set" of edges which are likely to connect nodes within a community. Using these edges we construct the communities after which local optimization methods are used to further improve the solution quality. Experimental results on a total of 136 problem instances which include various synthetic and real world complex networks show that the algorithm is very competitive against current state of the art techniques for community detection. In particular, our algorithm is more robust than existing algorithms as it performs well across many different types of networks.

Table of Contents

List of Figures

List of Tables

Acknowledgements

Chapter 1

Introduction

Complex networks are extensively used to model various real-world systems such as social networks (Facebook and Twitter), technological networks (Internet and World Wide Web), biological networks (food webs, protein-protein interaction networks) etc. For example, in social networks the nodes represent people and two nodes are connected by an edge if they are friends with each other. In the World Wide Web nodes represent webpages and an edge represents a hyperlink from one webpage to another. In protein-protein interaction networks nodes represent proteins and edges correspond to protein-protein interactions.

Complex networks exhibit distinctive statistical properties. The first property is that the average distance between nodes in a complex network is short [30]. This property is called the "*small world effect*". The second property is that the degree distribution of the nodes follows a *power-law* [2]. The degree of a node is the number of edges incident to it and the degree distribution of a network is the probability distribution of these degrees over the whole network. The power-law implies that this distribution varies as a power of the degree of a node. That is, the probability distribution function, $P(d)$, of nodes having degree d can be written as $P(d) \approx d^{-\gamma}$, $d > 0$ and $\gamma > 0$, γ is the exponent for the degree distribution. The third property, called *network transitivity*, is that two nodes who are both neighbors of the same third node, have an increased probability of being neighbors of one another [52].

Another property which appears to be common to such complex networks is that of a community structure. While the concept of a community is not strictly defined

in the literature as it can vary with the application domain, one intuitive notion of a community is that it consists of a subset of nodes from the original network such that the number of edges between nodes in the same community is large and the number of edges connecting nodes in different communities is small. Communities in social networks may represent people who share similar interests or backgrounds. For technological networks such as the World Wide Web, communities may represent groups of pages which share a common topic. In biological networks such as protein-protein interaction networks, communities represent known functional modules or protein complexes.

The problem of detecting communities is a very computationally intensive task [16] and as a result finding exact solutions will only work for small systems as the time it would take to analyze large systems would be too long. Therefore, in such cases it is common to use heuristic algorithms which do not return the exact solution but have an added advantage of lower time complexity making the analysis of larger systems feasible.

Recently, the task of finding communities in complex networks has received enormous attention from researchers in different fields such as physics, statistics, computer science, etc. One of the most popular techniques used to detect communities is to model the task as an optimization problem, where the quantity being optimized is called *modularity* [34], which is used to quantify the community structure obtained by an algorithm. Maximizing modularity is one of the most popular techniques for community detection in complex networks. It is further examined in Chapter 2 and we present its drawbacks in Chapter 4. Other techniques for community detection involve using dynamic processes running on a complex network such as random walks, or statistical approaches which employ principles based on information theory such as minimum description length (MDL) principle [45] to find communities.

Ant colony optimization (ACO) algorithms have been previously used to detect communities in complex networks [22, 48, 24]. In ACO, artificial ants are used in sequence to build a solution with later ants using information produced by previous ants. In ACO algorithms for finding communities, artificial ants are used to either

optimize modularity or find small groups of well connected areas in the network which are used as seeds for building communities.

In this thesis, we describe an ant-based optimization (ABO) approach [7], which is different from ACO, for finding communities in complex networks. We disperse a set of ants on the complex network who traverse the network based only on local information. The information produced by the ants is then used to build the first set of communities. Then, local optimization algorithms are employed to improve the solution quality before outputting the final set of communities.

In ACO methods, each ant is used sequentially to construct a solution whereas in the ABO technique used here, a set of ants is used to identify good "areas" in the network, which are edges connecting nodes in the same community, so as to reduce the search space of the problem. Then, construction algorithms are used to build a solution to the problem.

We have run our algorithm and six other community detection algorithms on a total of 136 problem instances out of which 128 are computer generated networks, with different degree distributions and community sizes, whose community structure is known. The remaining 8 are real world networks from different domains whose community structure is generally not known. Experimental results show that our algorithm is very competitive against the other approaches and in particular, it is very robust as it is able to uncover the community structure on networks with varying degree distributions and community sizes.

The rest of this thesis is organized as follows. Chapter 2 provides more detailed information about the problem statement and covers the previous work done on the problem. Our ant-based algorithm is described in Chapter 3. Chapter 4 covers the metrics used to evaluate the community structure produced by an algorithm and Chapter 5 covers the performance of this algorithm on the problem instances and compares it to existing algorithms. The conclusion is given in Chapter 6.

Chapter 2

Preliminaries

2.1 Problem Definition

Complex networks are modeled as graphs whose vertices represent the nodes in the complex network and edges represent the relationship between two nodes. From here on, the complex network under consideration will be represented as a graph $G = (V, E)$, where V represents the vertex set and E the edge set.

Communities are defined to be subsets of vertices such that the the number of edges between vertices in the same community is large and the number of edges between vertices in different communities is small. There are various possible definitions of a community and they are divided into mainly three classes: local, global and based on vertex similarity [16, 51]. Let $S \subseteq V$ and $i \in S$. We define the internal degree and external degree of vertex i with respect to S, denoted by $d_S^{in}(i)$ and $d_S^{out}(i)$, respectively, as follows:

$$d_S^{in}(i) = |\{(i,j) \in E | j \in S\}|, \tag{2.1}$$

$$d_S^{out}(i) = |\{(i,j) \in E | j \notin S\}|. \tag{2.2}$$

The subset S is a *community in the weak sense* [42] if:

$$\sum_{i \in S} d_S^{in}(i) > \sum_{i \in S} d_S^{out}(i). \tag{2.3}$$

That is, the subset S is a community in the weak sense if the sum of the internal degrees of all vertices in S is greater than the sum of the external degrees of all vertices in S. The subset S is a *community in the strong sense* [42], if

$$d_S^{in}(i) > d_S^{out}(i), \quad \forall i \in S. \tag{2.4}$$

That is, the subset S is a community in the strong sense if for each vertex in S, its internal degree is greater than its external degree. This is the definition of a community that we have adopted in this thesis.

The task of finding communities in graphs is usually modeled as an optimization problem. One of the most commonly used techniques is that of maximizing a quantity known as modularity [34]. It is a metric used to quantify the community structure found to determine how good of a structure we have obtained. The idea is that the density of edges connecting vertices in the same community should be higher than the expected density of edges between the same set of vertices if they were connected at random, but with the same degree sequence.

Let $G = (V, E)$ be a graph on n vertices where $V = \{1, \ldots, n\}$. Let $C = \{C_1, \ldots, C_k\}$ be a set of communities in G. We define the modularity of C, denoted by $Q(C)$, as

$$Q(C) = \sum_{i=1}^{k} \left(\frac{e_i}{m} - \left(\frac{D_i}{2m} \right)^2 \right), \tag{2.5}$$

where e_i is the total number of edges inside the ith community and D_i is the sum of the degrees of vertices in the ith community. So the first term represents the fraction of the total edges that are in the ith community and the second term represents the expected value of the fraction of edges if the vertices of the ith community were connected at random but with the same degree sequence.

Modularity is a widely adopted metric to evaluate the community structure obtained on real world networks whose community structure is not known beforehand. High values of modularity indicate strong community structure. Using modularity as

the objective function, the *Community Detection Problem* (CDP) can now be formulated as:

Community Detection Problem:

Input: An undirected graph $G = (V, E)$.

Output: A set of communities $C = \{C_1, \ldots, C_k\}$ which represents the community structure of G such that $\bigcup_{1 \leq i \leq k} C_i = V$ and $C_i \cap C_j = \varnothing$ for $1 \leq i, j \leq k$, $i \neq j$ such that $Q(C)$ is maximum.

Brandes et al. showed that since maximizing modularity is an $\mathcal{NP}$-hard problem [5], it is expected that the true maximum of modularity cannot be found in a reasonable amount of time even for small networks. Over the years, several heuristics have been developed for maximizing modularity. These are discussed in the next section.

It is worth mentioning that while communities can also be hierarchical in nature, i.e., small communities can be nested within larger ones or overlapping, where each node may belong to multiple communities, in this work we only concentrate on finding disjoint communities.

2.2 Previous Work

The seminal paper by Girvan and Newman [18] resulted in a lot of research into the area of community detection from various disciplines. As a result, these days there is a wide variety of community detection algorithms from fields like physics, computer science, statistics, etc. Covering all of them is beyond the scope of this work, for a more thorough review one can consult the comprehensive survey by Fortunato [16].

The methods for detecting communities can be broadly classified into hierarchical methods, modularity-based methods and other optimization methods involving statistics or dynamic processes on the graph.

2.2.1 Hierarchical Methods

Hierarchical community detection methods build a hierarchy of communities by either merging or splitting different communities based on a similarity criterion. The main idea here is to define the similarity criterion between vertices. For example, in data clustering where the points maybe plotted in 2D space, we can use Euclidean distance as a similarity measure. Hierarchical methods can be divided into two types based on the approach they take.

Divisive hierarchical methods start from the complete graph, detect edges that connect different communities based on a certain metric such as edge betweenness [18], and remove them. Edge betweenness of an edge is defined as the number of shortest paths between pairs of vertices that run through that edge. Edges connecting different communities have a high value of edge betweenness and by removing such edges iteratively, we can obtain the communities in the graph. Examples of divise hierarchical approaches can be found in [18, 42, 34].

Agglomerative hierachical methods initially consider each node to be in its own community and then merge communities based on the criterion chosen, until the whole graph is obtained. Examples can be found in [31, 3, 8]. The criteria these algorithms use to merge communities is modularity.

The disadvantage of hierarchical methods is that the result depends upon the similarity criteria used. Also, they return a hierarchy of communities whereas the network under consideration may not have any such hierarchical structure at all.

2.2.2 Modularity-based Methods

Modularity [34], introduced in the previous section, is a metric for evaluating the community structure of a network. Values of modularity approaching 1 (which is the maximum), indicate strong community structure. In practice, modularity usually ranges from 0.3 to 0.7 [34].

Under the assumption that high values of modularity indicate good community structure, the community structure corresponding to the maximum modularity for a

given graph should be the best. This is the reasoning employed by modularity-based methods which try to optimize $Q(C)$ to find communities. These methods are also the most popular methods to be employed for community detection.

The first algorithm to maximize modularity was introduced in [31]. It is an agglomerative hierarchical approach where vertices are merged based on the maximum increase in modularity. Several other greedy techniques have been developed, some of these can be found in [3, 8, 33, 40]. Simulated annealing approaches to maximizing modularity are described in [20, 29]. Extremal optimization for maximizing modularity was used by Duch and Arenas [14]. Genetic algorithms have also been used for maximizing modularity [36, 38, 37].

2.2.3 Other Methods

Various other techniques for community detection using methods based on statistical mechanics, information theory, random walks, etc., have been proposed.

Reichardt and Bornholdt [44] proposed a Potts model approach for community detection. In statistical mechanics, the Potts model is a model of interacting spins on a crystalline lattice. The community structure of the network is interpreted as the spin configuration that minimizes the energy of the spin glass with the spin states being the community indices [44]. Another algorithm based on the Potts model approach is described in [46].

Random walks have also been used to detect communities. The motivation behind this is the idea that a random walker will spend a longer amount of time inside a community due to the high density of edges inside it. These methods are described in [54, 39, 50].

Information theoretic approaches use the idea of describing a graph by using less information than that encoded in its adjacency matrix. The aim is to compress the the amount of information required to describe the flow of information across the graph. The community structure can be used to represent the whole network in a more compact way. The best community structure is the one that maximizes com-

pactness while minimizing information loss [35]. Random walk is used as a proxy for information flow and the minimum description length (MDL) principle [45] can be used to obtain a solution for compressing the information required. The most notable algorithm using this principle, referred to as Infomap, is described in [47].

2.3 Ant Algorithms

So far we have covered what the problem of community detection involves and the type of approaches that have been used to find the community structure in complex networks. To faciliate the understanding of our ant-based approach, a review of ant algorithms is given.

Ant algorithms are a probabilistic technique for solving computational problems using artificial ants. The ants mimic the behavior of an ant colony in nature for foraging food. As they travel, ants lay down a trail of chemical called *pheromone*, which evaporates over time. The higher the pheromone level on a path, the more likely it is to be chosen by the next ant that comes along.

For example, consider a food source and two possible paths to reach it, one shorter than the other. Assume two ants set off on both paths simultaneously. The ant taking the shorter path will return earlier than the other one. Now this ant has covered the trip both ways while the other ant has not yet returned, so the concentration of pheromone on the shorter path will be more. As a result, the next ant will be more likely to choose the shorter path due to its higher level of pheromone. This leads to a further increase of pheromone on that path and eventually all ants will end up taking the shorter path.

Thus, ants can be used for finding good paths within a graph. It is this basic idea that is used in ant algorithms for solving computational problems, but there are different variations. The first such approach, called Ant System (AS), was applied to the Traveling Salesman Problem by Marc Dorigo [12]. Here, each ant is used to construct a tour and the pheromone level on all the edges in that tour is updated

based on its length. An ant picks the next destination based on its distance and the pheromone level on that edge. A global update is applied everytime, which evaporates the pheromone on all edges.

Since in AS each ant updates the pheromone globally, the run time can be quite high. Ant Colony System (ACS) was introduced to address this problem [13]. In ACS, a fixed number of ants are positioned on different cities and each ant constructs a tour. Only the iteration best ant, the one with the shortest tour is used to update the pheromone. Ants also employ a local pheromone update in which the pheromone of an edge was reduced as an ant traversed it in order to encourage exploration.

Another variation of AS, the Max-Min Ant System (MMAS), was introduced by Stützle and Hoos [49]. The first change in this model is that the pheromone values are limited to the interval $[\tau_{min}, \tau_{max}]$. Secondly, the global update for each iteration is either done by the iteration best ant or the ant which has the best solution from the beginning. This is used to avoid early convergence of the algorithm. Additionally, the pheromone on each edge is initialized to τ_{max} so as to encourage exploration in the beginning of the algorithm. Apart from this, MMAS used the same structure of AS for edge selection and lack of local pheromone update. Both these variations were an improvement over the original AS.

The techniques mentioned above fall into the category of ant colony optimization (ACO) methods. The approach used in our algorithm falls in to the category of ant-based optimization (ABO) [7]. While in ACO, ants build complete solutions to the problem, in ABO ants are only used to identify good regions of the search space after which construction methods are used to build the final solution [6]. The ants only need local information as they traverse the graph. Choosing the next edge involves the pheromone level and some heurisitic information based on the rules specified for the ants.

To the best of our knowledge, our algorithm is the first ABO method for detecting communities in complex networks. The next chapter describes in detail our ant-based algorithm for finding communities in complex networks.

Chapter 3

Ant-Based Community Detection

In this chapter, we describe our ant-based approach for detecting communities in complex networks. The input to the algorithm is an undirected graph $G = (V, E)$ and the expected output is a (weighted) graph representing the community structure of G. The algorithm is divided into three main phases: *exploration*, *construction* and *optimization*. In the exploration phase, the ants traverse the graph and lay pheromone along the edges as they travel. The construction phase is used to build an initial set of communities based on the pheromone level on the edges after the exploration phase. Finally, the optimization phase is used to improve the solution produced by the construction phase before returning the community structure of G.

3.1 Overview

Our algorithm consists of artificial ants which explore the graph based on a set of rules to discover edges which connect vertices in the same community. Before the exploration phase, an initialization step is used to initialize the pheromone on all edges of G and place an ant on each vertex of G. In the exploration phase, ants traverse the graph based on a fixed set of rules for a number of cycles (or iterations) and lay pheromone along the edges. The objective here is to narrow the search space of the problem by discovering edges connecting vertices in the same community. After the exploration phase, the input to the construction phase is the edges of the graph in decreasing order of their pheromone level which are used to build the first set of com-

munities. This set of communities is used as input to the optimization phase which uses local optimization algorithms to improve the solution quality before outputting the final set of communities. An outline of the algorithm is shown in Figure 3.1.

3.2 Data Structures

Before explaining each phase of the algorithm in detail, the various data structures used in it are mentioned to help facilitate the description. The main data structures used are the following:

- Graph $G = (V, E)$, whose community structure is to be found
- Weighted graph $G_W = (V_W, E_W)$ represents the community structure of G
- Community object
- Ants

The graph consists of a vertex set V and an edge set E and is represented in an adjcacency list format. Each edge is augmented with its pheromone level (phm), the number of times it has been traversed by an ant since the last ($num\_visited$) and its initial pheromone level ($init\_phm$). Each vertex is augmented with information regarding how many vertices are adjacent to it ($neighbors$) and the number of vertices it has in common with each of its neighbors ($common$).

The weighted graph G_W also consists of a vertex sex V_W and edge set E_W. However, each vertex in V_W represents a community and E_W represents the edges between different communities. Hence, G_W can be thought of as a compacted version of the original graph representing its community structure. Each vertex in V_W is augmented with a list of the communities adjacent to it, the sum of the pheromone level of the edges that are within a community ($internal\_phm$), the number of edges that are within that community ($internal\_edges$), the total pheromone of that community

```
AntBasedCommunityDetection(G = (V, E))
Input: G = (V, E), graph whose community structure is to be found
Output: Weighted Graph G*_W, whose vertices represent the community
        structure of G
begin
    // Initialization
    Create the set of ants A, of size |V|
    InitAnts(G, A)
    foreach v ∈ V do
        sort(v.neighbors)
    foreach v ∈ V do
        for i = 0; i < v.degree; i++ do
            u ← v.neighbors[i]
            v.common[i] = set_intersection(v.neighbors, u.neighbors)
    i ← 1
    // Exploration phase
    while i < i_max do
        ExploreGraph(G, A)
        ResetAnts(G, A)
        i ← i + 1
    // Construction phase
    Sort E in decreasing order of pheromone
    G_W ←BuildCommunities(E)
    // Optimization Phase
    G*_W ← LocalOpt(G_W, G)
    return G*_W

InitAnts(G, A)
begin
    Initialize pheromone on all edges of G
    for i = 0; i < |V|; i++ do
        A[i].location ← V[i]
        A[i].tabuList ← ∅
```

Figure 3.1: Ant-Based Community Detection Algorithm

(*total_phm*) which is equal to *internal_phm* plus the pheromone level along each outgoing edge to an adjacent community. Since there may be multiple edges between vertices in different communities in the original graph, they are collapsed into one weighted edge in G_W. Each edge $(A, B) \in E_W$ stores the pheromone level between two communities (*edge_phm*) and the number of edges from the original graph that are between the two communities (*edge_count*).

The Community object consists of the following elements. A vector called *membership* is used to keep track of the community assignment of each vertex $v \in V$. Two vectors are used to maintain the internal degree of each vertex to its community (*in_degree*) and its external degree to the different communities it might be connected to (*out_degrees*). The last element in this object, called *vertex_strength*, which represents the difference between the indegree of a vertex to its community and the total outdegree to vertices outside its community.

The set of ants A has a fixed cardinality, equal to the number of vertices in the graph. Each ant maintains its current location (*location*) which is a vertex in V and a tabu list which stores the most recently visited vertices.

So far we have given an outline of the algorithm and presented its main data structures. The next section describes each phase of the algorithm in detail, starting with the initialization step.

3.3 Initialization

The input to this step is $G = (V, E)$. Here, the algorithm initializes the set of ants and places each ant on a vertex of the graph and sets the tabu list of each ant to be empty. The initial pheromone level of each edge is initialized to 1 so that all edges of the graph are equally likely to be chosen in the beginning of the exploration phase. The minimum pheromone level is also set to 1 as we do not want the pheromone level of an edge to get too low which would prevent it from getting selected in future iterations.

After the above step, the algorithm calculates the number of vertices a vertex has

in common with each of its neighbors. Since each vertex maintains a list of neighbors, the number of common vertices can be computed by a simple set intersection operation. In order to compute the intersection quickly, the list of neighbors for each vertex is sorted. The initialization phase is shown in Figure 3.1.

3.4 Exploration

Once the initialization step is completed, the exploration phase starts where the ants traverse the graph and lay a trail of pheromone along the edges. The exploration is carried out for a fixed number of iterations. The aim here is to discover edges connecting vertices in the same community and mark them with a high level of pheromone. At the end of the exploration phase, the edges with high level of pheromone are selected to build an initial set of communities. An outline of the exploration phase is shown in Figure 3.2.

In each iteration, the ants are moved for a fixed number of steps. In each step all the ants are moved in parallel and this is repeated until a specified number of steps are completed. To increase efficiency, the pheromone level of the edges is updated only after k steps, where k is a fixed constant.

In each step, each ant selects the next vertex to move to depending upon the pheromone level of the edges incident to its current location. Since we have defined communities to be subset of vertices where each vertex has more edges to vertices in its own community, we expect two adjacent vertices to be in the same community if they have more vertices in common from their set of neighbors. Hence, an ant selects an edge with a probability that is proportional to the pheromone level of the edge and the size of the neighborhood overlap with the vertex that the edge leads to. This process of selection, called *proportional selection*, favors edges connecting two vertices which might be in the same community and at the same time an edge connecting two vertices in different communities can also be chosen, but with a smaller probability. When an ant traverses an edge leading to another vertex, the pheromone level of the edge is marked to be updated by incrementing the number of times it has been

traversed (*num_visited*) during that step and the vertex that edge leads to is added to the ant's tabu list to avoid returning to that vertex for a fixed period of time.

We employ a few mechanisms to avoid getting caught in local optima. First, each ant maintains a tabu list which is a fixed length circular queue that stores the most recently visited vertices by the ant. If an ant selects an edge leading to a vertex already present in its tabu list, it attemps to choose another edge. The number of such attempts made by an ant is specified by the paramter *max_tries*. Second, the pheromone level of each edge is evaporated periodically by a certain factor (η). In the beginning, the evaporation rate is high so as to encourage exploration and it is gradually decreased after every iteration to help the ants converge on a good set of edges. Finally, the minimum pheromone level of an edge is set to 1 as we do not want the pheromone level of an edge to become so low that it may not be considered during successive iterations.

We have mentioned previously that the pheromone level of an edge is updated after a fixed number of steps. Each edge keeps track of the number of times it has been traversed since the previous pheromone update. This value is used along with the current pheromone level during the update step. The formula is the same as that used in [6]:

$$e.phm = (1 - \eta)e.phm + e.num\_visited \times e.init\_phm \tag{3.1}$$

where e is the edge in the graph being updated, *phm* is the pheromone level of e, η is the evaporation rate, *num_visited* is the number of times the edge was traversed since the last pheromone update and *init_phm* is the initial pheromone level, in this case $init\_phm = 1, \forall e \in E$.

At the end of each iteration, two operations are performed. First, the pheromone evaporation factor, η, is updated by a constant factor, $\Delta\eta$, as explained above. The initial value of η is set to 0.5 and $\Delta\eta$ is set to 0.95. Second, the ants are reset before the start of the next iteration. About half the ants stay in their current location while the other half are assigned a random vertex as shown in Figure 3.3. The maximum

```
ExploreGraph(G, A)
Input: Graph G and set of ants A
Result: Each ant attempts to change its location
begin
    for s = 1 to max_steps do
        if s mod update_period==0 then
            UpdatePheromone(G)
        foreach a ∈ A do
            num_tries ← 0
            moved ← False
            while not moved and num_tries < max_tries do
                v1 ← a.location
                Select an edge (v1, v2) at random and proportional to the
                phermone level and the size of their neighborhood overlap
                if v2 ∉ a.tabuList then
                    add v2 to a.tabuList
                    a.location ← v2
                    (v1, v2).num_visited++
                    moved ← True
                else
                    num_tries++
```

```
UpdatePheromone(G)
Result: Updates the pheromone level of each edge based on the number of
        times it has been traversed
begin
    foreach e ∈ E do
        e.phm ← (1 − η) × e.phm + e.num_visited × e.init_phm
        e.num_visited ← 0
    if e.phm < min_phm then
        e.phm = min_phm
```

Figure 3.2: Exploration phase

```
ResetAnts(G, A)
begin
    foreach a ∈ A do
        if Random(0, 1) < 0.5 then
            a.location ← Random(0, |V| − 1)
        a.tabuList ← ∅
    // Reduce pheromone evaporation factor η using equation 3.1
```

Figure 3.3: Reset ants

number of steps during each iteration of the exploration phase is set to $\min\{75, \frac{2|V|}{3}\}$.

3.5 Construction

So far we have described the initialization and exploration phase of the algorithm. The next phase of the algorithm is to build an initial set of communities based on the pheromone level of the edges.

At the end of the exploration phase we expect the edges connecting vertices in the same community to have a high level of pheromone. The construction phase utilizes these edges to build the initial set of communities. We sort the edge set in decreasing order of pheromone level and use that as input to the construction phase. To obtain the initial community structure we build the weighted graph G_W. The algorithm reads in the edges $(i, j) \in E$ in sorted order. If both i and j have not been assigned a community, then they are assigned to a new community. If one of i and j is assigned to a community but the other is not, we add it to that community. If both i and j are in separate communities, say A and B, respectively, then we create the weighted edge $(A, B) \in E_W$ if it does not exist and set $edge\_count(A, B)$ to 1, or increment $edge\_count(A, B)$ by 1 if the weighted edge (A, B) is already present in G_W. We also update $edge\_phm(A, B)$ of the weighted edge by the pheromone level of the edge $(i, j) \in E$. The procedure for building the first set of communities is shown in Figure 3.4.

```
BuildCommunities(E)
Input: Edges of G sorted in decreasing order of pheromone
Output: Initial set of communities in G represented by weighted graph G_W
begin
    Initialize weighted graph G_W
    foreach (v_1, v_2) ∈ E in sorted order do
        if both nodes in no community then
            Assign v_1 and v_2 a new community
            Update G_W
        if v_1 is assigned a community but v_2 isn't then
            Add v_2 to v_1's community
            Update G_W
        if v_2 is assigned a community but v_1 isn't then
            Add v_1 to v_2's community
            Update G_W
        if both nodes in separate communities then
            Update G_W
    return G_W
```

Figure 3.4: Build initial set of communities

At the end of this phase we obtain the weighted graph which represents the initial set of communites in G. This community structure is improved in the local optimization phase by attempting to reassign each vertex's community assignment based on its degree distribution, and merging and splitting different communities. This is done until a fixed number of steps have passed without any improvement. We use modularity as the criterion for measuring improvement in successive iterations. The next section describes the local optimization step in further detail.

3.6 Local Optimization

The input to this phase is the initial set of communities obtained at the previous step. Since the construction phase only utilizes the pheromone level of the edges to build the communities and does not use the structure of the graph, it is likely that the community structure obtained is weak. The local optimization step attempts to improve the solution quality by fine tuning the structure of the weighted graph to fit

the definition of a community that we have chosen before outputting the final set of communities in G. An outline of the local optimization phase is shown in Figure 3.5.

The local optimization phase is divided into four different steps. First is the reassignment step where the algorithm attempts to change each vertex's community assignment based on the indegree of a vertex to its community and the outdegree to vertices in different communities that the vertex may be connected to. The second step involves merging communities in the weighted graph based on how well connected different communities are. The third step perturbs the community assignments of the vertices in G based on each vertex's degree distribution in order to obtain a different configuration. The aim of the perturbation step is to avoid getting stuck in local optima.

The first three steps are performed continuously until a fixed number of iterations pass without any improvement. Then, the algorithm goes to the last step which involves breaking up different communities to obtain a different structure. The next sections describe each step in further detail.

3.6.1 Reassignment

Since the construction phase does not utilize the structural properties of the graph, it is possible that the communities built do not fit the strong definition of a community. The reassignment step attempts to correct this by changing each vertex's community assignment to the community it has more number of edges to. This is repeated until the solution quality does not improve for a fixed number of iterations.

Each vertex $v \in V$ keeps track of the indegree to vertices inside its community and outdegree to vertices in other communities. Let the total outdegree of a vertex be equal to the number of edges incident on that vertex which lead to vertices outside its community. We sort V based on the decreasing value of the total outdegree of the vertices. Then for each vertex v in the sorted order, we find the community to which it has the maximum outdegree. Let this community be denoted by C. If v's outdegree to C is greater than the indegree to its current community, we change the community assignment of v to C.

```
LocalOpt(G_W, G)
Input: Graph G, current weighted graph G_W
Output: Final set of communities in G represented by G*_W
begin
    i, j, k ← 0 // Initialize counters
    Initialize best weighted graph G*_W
    while i < max_decrease do
        while j < max_decrease do
            while k < 5 do
                ReassignCommunities(G_W)
                G_W ←RebuildGraph(E)
                if modularity improves then
                    G*_W ← G_W
                else
                    k++
            k ← 0 // Reset k
            // Merge best solution at this stage
            G_W ← G*_W
            MergeCommunities(G_W)
            if modularity improves then
                G*_W ← G_W
            Perturb(G, G_W)
            if modularity improves then
                G*_W ← G_W
            else
                j++
        j ← 0 // Reset j
        // Split the communities in G*_W
        G_W ← G*_W
        SplitCommunities(G_W)
        if modularity improves then
            G*_W ← G_W
        else
            i++
    return G*_W
```

Figure 3.5: Local Optimization

After one round of reassigning is complete, we rebuild the weighted graph representing the current community assignments of the vertices in G. If there is an improvement in modularity, then we update the best weighted graph. As mentioned above, the reassignment step is carried out continuously until there is no improvement in modularity for 5 successive iterations. Even after this, the communities may not satisfy the strong community condition. At this point, the algorithm proceeds to the merging step. The reassignment and rebuilding step is shown in Figure 3.6.

3.6.2 Merging

The input to this step is the weighted graph G_W having the highest modularity from the reassignment step. The current community structure of G_W may be weak as it is possible for a community to be broken up into subcommunities which could be well connected between each other. By merging these subcommunities it is possible to improve the community structure.

Let A be a community. We define the following measures:

- $internal\_phm(A)$ denotes the total pheromone level of all edges inside community A.
- $edge\_phm(A, B)$ denotes the pheromone level of the weighted edge $(A, B) \in E_W$ connecting communities A and B.
- Since G_W is represented as an adjacency list, let $N(i)$ denote the ith neighbor of community A in its adjacency list. Let $total\_phm$ denote the sum of $internal\_phm$ and the pheromone along each weighted edge incident to community A. It can be written as

$$total\_phm(A) = internal\_phm(A) + \sum_{i=1}^{d_A} edge\_phm(A, N(i)), \quad (A, N(i)) \in E_W, \tag{3.2}$$

where d_A is the degree of community A. $total\_phm$ is used to determine the

`ReassignCommunities(`G_W`)`
Input: Weighted Graph G_W
Result: Reassign vertices to a different community based on their degree
begin
    Sort V based on decreasing order of total outdegree
    foreach $v \in V$ *in sorted order* **do**
        Find community (C) to which v's out-degree is maximum
        if $v.out\_degree[C] > v.in\_degree$ **then**
            $membership[v] \leftarrow C$

`RebuildGraph(`E`)`
Input: Edges of the original graph E
Output: A new weighted graph representing the current community assignment
begin
    Initialize G'_W
    `// At this stage all vertices in` G `have been assigned a community`
    foreach $(v_1, v_2) \in E$ **do**
        if v_1 *and* v_2 *are in the same community* **then**
            Add v_1 and v_2 to the same community in G'_W
            Updated G'_W
        else
            Add v_1 and v_2 to their respective communities
            Update G'_W
    return G'_W

Figure 3.6: Reassign Communities and Rebuild Weighted Graph

connectivity of a community to its neighbors. If a community is well connected to another one, the fraction of *total_phm* along the edge connecting the two communities should be high. The measures which indicate how well connected a community is are defined below.

- $strength(A) = \frac{internal\_phm(A)}{total\_phm(A)}$, denotes the strength of community A. It is used to determine how well connected community A is.

- Let $strength(A, B)$ denote how well connected two communities A and B are by the weighted edge $(A, B) \in E_W$. It can be written as

$$strength(A, B) = \frac{edge\_phm(A, B)}{total\_phm(A)} + \frac{edge\_phm(A, B)}{total\_phm(B)}. \tag{3.3}$$

Since ants are used to discover intracommunity edges in the exploration phase, we expect the edges between vertices in the same community to have a high level of pheromone. As a result, the weighted edge $(A, B) \in E_W$ between two well connected communities, A and B, in G_W should also have a high level of pheromone ($edge\_phm(A, B)$) as compared to the pheromone level of the weighted edge (A, B) if A and B were weakly connected.

We consider community A to be well connected on its own, if $strength(A)$ is above a certain threshold as specified by the parameter *community_strength*. If A is well connected to another community, B, the pheromone level of the weighted edge $(A, B) \in E_W$ should be high and merging A and B should improve the community structure.

The main idea while merging communities is to find the edges in E_W which are between two well connected communities. Based on the above observation that two well connected communities in G_W should have a high level of pheromone along the weighted edge connecting them, $strength(A, B)$ should be high for such edges.

In the merging step, the algorithm calculates $strength(A, B)$ for each weighted edge $(A, B) \in E_W$ and sorts E_W in decreasing order of $strength(A, B)$. Then for each weighted edge (A, B) in the sorted order, we merge communities A and B if either

```
MergeCommunities(G_W)
Input: Best weighted graph after the reassignment step
Result: Merge communities in G_W
begin
    Calculate strength(A, B) for each weighted edge (A, B) ∈ E_W
    Sort E_W based on decreasing order of strength(A, B)
    foreach (A, B) ∈ E_W in sorted order do
        // Each vertex in G_W is a community
        if (internal_phm(A)/total_phm(A)) > community_strength) and
        (internal_phm(B)/total_phm(B) > community_strength) then
            Do not merge A and B
        if (edge_phm(A, B)/total_phm(A) > (internal_phm(A)/total_phm(A))
        then
            Merge A and B
        if (edge_phm(A, B)/total_phm(B) > (internal_phm(B)/total_phm(B))
        then
            Merge A and B
```

Figure 3.7: Merging Communities

$strength(A)$ or $strength(B)$ is below the threshold value. The merging step is shown in Figure 3.7.

At the end of the merging step the algorithm calculates the new modularity of G_W and updates the best weighted graph if merging the communities increases the modularity. After merging, we proceed to the third step in the optimization phase.

3.6.3 Perturbation

So far the local optimization algorithms reassign the communities for vertices based on their degree distribution and merge different communities based on the criteria specified in the previous section. The next step perturbs the community structure of the weighted graph in order to obtain a different configuration to avoid getting stuck in local optima.

The input to the perturbation step is the current best weighted graph. Here the algorithm attempts to create a different community structure by changing the community assignment of those vertices which are on the fringes of their respective

communities. We consider a vertex to be on the fringe of its current community if the difference between the vertex's indegree to its community and its total outdegree to vertices in different communities is small, in the range [0, 2]. By perturbing the community assignments of such vertices, it is possible to obtain a different configuration without changing the current community structure drastically.

Each vertex $v \in V$ maintains the difference between its indegree relative to the community that it belongs to, and total outdegree in an element called $vertex\_strength$ (see Section 3.2). At this step, we also maintain a tabu list for each vertex which is a fixed length circular queue storing the most recently assigned communities for each vertex during the perturbation step. This tabu list should not be confused with the tabu list maintained by the ants during the exploration phase.

The perturbation is performed as follows. For each vertex $v \in V$, we check if $vertex\_strength[v]$ lies in between 0 and 2. If yes, then we find the community C to which v has the maximum outdegree. Let the outdegree of v to C be denoted by $d_C^{out}(v)$. If C is present in v's tabu list then we consider the next vertex in V, otherwise we carry out the following steps. If $d_C^{out}(v) > 2 \times vertex\_strength[v]$, then we change the community assignment of v to C and add C to v's tabu list to avoid choosing this community for a fixed period of time. Since it is possible that v's total outdegree could be split between multiple communities, we only perturb its community assignment if $d_C^{out}(v)$ is greater than $2 \times vertex\_strength[v]$. The perturbation step is shown in Figure 3.8.

3.6.4 Splitting Communities

If the perturbation step results in an increase in modularity then the best weighted graph is updated. At the end of the perturbation step we go back to the reassignment step if the number of iterations without improvement is less than the parameter $max\_decrease$. However, if this threshold is exceeded the algorithm goes to the next step in the local optimization phase which involves splitting the communities.

The input to this step is the best weighted graph obtained after the first three

```
Perturb(G, G_W)
Input: Original graph G, weighted graph G_W
Result: Perturb vertex communities and rebuild resulting weighted graph
begin
    foreach v ∈ V do
        if vertex_strength[v] is between 0 and 2 then
            Find community (C) to which v has the highest out-degree
            (d_c^out(v)))
            if C ∈ tabu list OR d_c^out(v) < 2 × vertex_strength[v] then
                // Skip
            else
                Add C to tabu list
                membership[v] ← C // Change community
    G_W ←RebuildGraph(E)
```

Figure 3.8: Perturbation

steps in the local optimization phase have run for a number of iterations until there is no improvement. In order to break up a community, the algorithm needs a starting point or a seed for the new community. We decided to use a 4-clique, which is a fully connected subgraph of 4 vertices as the seed.

For each community in the best weighted graph, we try to build a 4-clique using the vertices of the original graph G in the community. A greedy approach is used to build the clique: starting with the vertex whose degree is highest in the current community, we add its neighbor with the highest degree to the current potential clique and repeat until we have added four vertices to the potential clique or we cannot chose any other vertex at which point we restart building the clique using the vertex with the next highest degree in that community. If the potential clique of size 4 is actually a clique, then we use this clique as the seed for the new community. Let the new community be denoted by C.

For all the remaining vertices in the current community under consideration which are not included in C, we compute their outdegree to C and indegree to the current community as these values change due to the removal of the 4-clique. If the outdegree of a vertex v in the current community to C is higher than the indegree of v to

its current community, the community assignment of v is changed to C and the in and out degrees of all its adjacent vertices are updated to reflect this change. This way, groups of vertices in each community which are well connected to the clique are assigned a new community.

After this procedure is repeated for all communities in the weighted graph, we recompute the modularity to check for improvement. If the modularity improves, then we update the best solution obtained so far. Otherwise we go back to the reassignment step. As mentioned for other steps in the local optimization, the splitting of communities is attempted until the modularity does not improve for a number of iterations. Once the threshold is exceeded, we terminate the algorithm and return the best weighted graph G^*_W.

3.7 Parameters

The previous sections covered the various steps involved in the ant based algorithm for finding communities. During the description we mentioned several parameters which are used in the implementation and this section provides a list of them. The various parameters in the algorithm are mentioned in Table 3.1. These parameters are not for a single type of graph but have been used for all graphs on which the algorithm is tested. Some of these parameters are adopted from [6].

The parameter *community_strength* is used to determine whether a community in the weighted graph is well connected on its own during the merging step. Its value is fixed based on the edge density (as a percentage), $\delta(G)$, of the graph whose community structure is to be found. It is defined as follows

$$\delta(G) = \frac{2|E|}{|V||V-1|} \times 100 \tag{3.4}$$

where $|E|$ is the number of edges in G and $|V|$ is the number of vertices in G. The value of $\delta(G)$ depends on the graph under consideration. For complete graphs, where every pair of vertices is connected by an edge, the value is 100. Since complex networks are

Table 3.1: Parameters in the algorithm

Parameter	**Value**	**Comments**
i_{max}	25	Maximum number of iterations during the exploration phase
max_steps	$\min\{\frac{2\|V\|}{3}, 75\}$	Maximum number of steps in each iteration
η	0.5	Pheromone evaporation rate
$\Delta\eta$	0.95	Pheromone update constant
$update\_period$	$max\_steps/3$	Number of cycles between pheromone update
$LIST\_SIZE$	2	Tabu list size
$max\_decrease$	3	Number of steps without improvement
$community\_strength$	0.25, 0.35 or 0.80	Threshold for a community
max_tries	2	Number of attempts to move made by an ant in each step

usually sparse, the value of $\delta(G)$ is usually much lower. *community_strength* is then defined as follows

$$community\_strength = \begin{cases} 0.8, & \text{if } \delta(G) < 0.1 \\ 0.35, & \text{if } 0.1 \leq \delta(G) \leq 1 \\ 0.25, & \text{if } \delta(G) > 1 \end{cases} \tag{3.5}$$

If G is very sparse ($\delta(G) < 0.1$), then *community_strength* is set to a high value. Since intracommunity edges have a high level of pheromone, a lower threshold will always be crossed since G has a very low edge density. This will prevent communities from merging, leading to a large number of small communities (of sizes 2 or 3).

The value of $\delta(G)$ is calculated when the algorithm reads the graph G, so the value of *community_strength* is set during runtime making the algorithm self-adaptive.

We tried several values for the maximum number of iterations, i_{max}, ranging from 25 to 100. The running time of the algorithm is directly affected by this parameter and we noticed that a value of 25 and above for i_{max} did not improve the results obtained. Hence this parameter is fixed to 25.

In this chapter we have described an ant-based algorithm for finding communities in complex networks in detail. The following chapter describes the methods used for generating synthetic graphs with known community structure in order to test community detection algorithms. It also describes the metric used for evaluating the results

obtained by an algorithm against this known structure and presents more information about modularity, which is the quality metric used to evaluate the community structure obtained on real world networks whose community structure is not known.

Chapter 4

Benchmark Graphs

In order to test a community detection algorithm, it is necessary to compare its performance against other existing methods. Since community detection algorithms usually output some community structure for any input graph, it is also necessary to evaluate how good that structure is. In the field of community detection, while the intuitive idea of a community is considered the same, there are currently a wide range of methods available which employ different techniques to solve this problem, and as a result they tend to produce different outputs for the same network. Due to this it is necessary to use benchmark graphs whose community structure is known, to be able to test different community detection algorithms.

We have mentioned earlier that the metric adopted to evaluate the community structure for real world graphs is modularity as we don't know the real community structure of such networks. Since the community structure for synthetic graphs is known, we can use a different metric to evaluate community structure found by an algorithm by comparing it against the known community structure of the synthetic graph to determine how similar they are.

Before describing the metric for synthetic graphs, this chapter discusses two benchmarks for generating synthetic graphs with known community structure. The last section discusses the drawbacks of modularity.

4.1 Synthetic Graphs

One of the most commonly used method for generating graphs with known community structure is the planted ℓ-partition model [9]. This model partitions a graph with $n = g \cdot \ell$ vertices into ℓ communities with g vertices each. A vertex is linked to others in its own community with probability p_{in} and to those outside its community with probability p_{out}. If $p_{in} > p_{out}$ then the number of intracommunity edges are higher and the graph has a community structure.

4.1.1 Girvan-Newman Benchmark

The Girvan-Newman benchmark [18] is a special case of the planted ℓ-partition model where $n = 128$, $\ell = 4$ and the average degree of each vertex is 16. This was the first benchmark suggested for testing community detection algorithms so it was quickly adopted to test community detection algorithms .

Tests on this benchmark were performed by increasing the outdegree of each vertex in each instance. Since the degree of each vertex is fixed to 16, when the outdegree is set to 8 each vertex has as many connections to nodes in its own community as to those outside it and as a result the community structure is very fuzzy. Due to this most algorithms begin to fail at this value for the outdegree.

Even though this benchmark became very popular, it is evident that its structure is too simple. In particular, it does not possess the properties attributed to complex networks, such as power-law degree distributions, heterogeneous community sizes etc. The need for benchmarks having these properties became necessary. The LFR Benchmark, which is described next, was proposed to address this issue.

4.1.2 LFR Benchmark

The LFR benchmark [27] is an improvement over the Girvan-Newman benchmark as it takes into account the structure of complex networks and thus is more representative of networks found in real life. It is also a special case of the planted ℓ-partition model where group sizes and node degrees vary according to a power law. This poses a much

harder test for community detection algorithms.

This benchmark possesses a variety of parameters. We can specify the number of nodes in the graph, their average and maximum degrees, maximum and minimum community sizes. The benchmark assumes that degree distributions and community sizes follow a power-law whose exponents are γ and β respectively. γ is usually set in the range [2, 3] whereas β is in the range [1, 2]. As mentioned in Chapter 1, the degree distribution of a graph is the probability distribution of the degrees of all vertices in the graph. Power-law implies that the degree distribution varies as a power of the degree of the vertices. If $P(d)$ is the probability distribution function of vertices having degree d, then: $P(d) \approx d^{-\gamma}$, $d > 0$ and $\gamma > 0$. The community size distribution is defined similarly. The most important parameter here is called the mixing parameter, denoted by μ, which specifies what fraction of its edges a vertex shares with vertices outside its community. If $d(i)$ is the degree of the ith vertex in the synthetic graph with a certain μ, the internal degree of vertex i relative to its community is $(1 - \mu)d(i)$ and its external degree is $\mu d(i)$. As a result, the mixing parameter denotes how well connected the communities in the synthetic graph are between each other.

We can generate different instances by varying μ to make the communities fuzzy and harder to detect. The technique to generate the graphs is fast, of the order $O(m)$, where m is the number of edges in the graph. The LFR benchmark is known to pose a harder test for community detection algorithms and is the benchmark we have adopted for testing our ant based algorithm as well. In the next section we describe metrics to evaluate the community structure obtained by different algorithms for synthetic graphs and real world networks.

4.2 Community Structure Evaluation

4.2.1 Normalized Mutual Information

Since we already know the community structure for synthetic graphs, we can use that information for comparing how similar the community structure obtained by an algorithm is to the planted communities in the graph.

For synthetic graphs the most widely adopted quality metric is the Normalized Mutual Information (NMI), as described in [11]. Here we define a *confusion matrix* N, where the rows correspond to the "planted" communities and the columns correspond to the communities found by an algorithm. N_{ij} represents the number of nodes in the ith planted community that appear in the jth found community. Let $\mathcal{P} = \{A_1, \ldots, A_k\}$ denote the set of planted communities in the graph and $\mathcal{F} = \{B_1, \ldots, B_l\}$ denote the set of communities found by an algorithm. Let n denote the number of vertices in the synthetic graph. The NMI, denoted by $I(\mathcal{P}, \mathcal{F})$, based on information theory is defined as follows:

$$I(\mathcal{P}, \mathcal{F}) = \frac{-2 \sum_{i=1}^{k} \sum_{j=1}^{l} N_{ij} \log \left(\frac{nN_{ij}}{|A_i||B_j|} \right)}{\sum_{i=1}^{k} |A_i| \log \left(\frac{|A_i|}{n} \right) + \sum_{j=1}^{l} |B_j| \log \left(\frac{|B_j|}{n} \right)} \tag{4.1}$$

where $|A_i|$ denotes the cardinality of the ith planted community and $|B_j|$ denotes the cardinality of the jth found community.

If the found set of communities is identical to the planted one then $I(\mathcal{P}, \mathcal{F}) = 1$, which is its maximum value. If the found set of communities is totally independent of the planted one then, $I(\mathcal{P}, \mathcal{F}) = 0$.

4.2.2 Modularity

As described in Chapter 2, modularity is a widely adopted quality metric for evaluating the community structure obtained on real world networks. We cannot calculate the NMI on such networks as we do not have prior information about the real community structure of such networks.

Modularity is calculated by computing the fraction of edges that fall within a community as compared to the fraction if the vertices in a community were connected randomly keeping the same degree sequence. It was assumed that high modularity structures correspond to a good community structure which is the motivation behind modularity optimization algorithms.

Despite the huge popularity of modularity optimization methods due to their speed and thus allowing the opportunity to analyze very large networks, its properties have been recently investigated which brought forward a few drawbacks.

Fortunato and Barthélemy [17] showed that modularity suffers from a resolution limit. They found that modularity maximization favors community structures where several subcommunities are aggregated into one community. Modularity fails to indentify communities smaller than a certain scale which depends on the size of the network and the connectedness of the communities. This contradicts the notion of a community being a local measure instead of a global one. To this extent, they compared the communities found using modularity optimization by simulated annealing [20] and then reapplied the method on each community returned by the algorithm. They found that most communities themselves had a clear community structure with high modularity values. As a result, the final number of communities obtained were much more than the ones reported by modularity maximization.

Good et al. [19] further examined the performance of modularity maximization and apart from the resolution limit they found two other drawbacks. First, there are an exponential number of structurally diverse alternate community structures whose modularity is very close to the maximum, this is called the degeneracy problem. This explains the good performance of modularity maximization methods as they are able to discover a high ranking community structure depending on the implementation and explains the reason why different algorithms can have widely varying outputs for the same network. Second, the maximum modularity Q_{max} depends upon the size of the network and the number of communities it contains. This means that the maximum of modularity is dependent upon the topology of the network under consideration.

In this chapter we have covered the techniques used for generating various syn-

thetic graphs and methods to evaluate the output of community detection algorithms. The next chapter presents the experimental results of the ant based algorithm on the LFR benchmark and several real world networks and compares its performance against six well known community detection algorithms.

Chapter 5

Experimental Results

In this chapter we describe the dataset used for testing our algorithm and compare its performance against six other well known community detection algorithms.

5.1 Dataset

Our ant-based community detection algorithm (ABCD) was run on a total of 136 problem instances comprising of 128 synthetic networks generated using the LFR benchmark and 8 real world networks.

For the LFR benchmark, the following parameters were fixed for all graphs: the average degree is 20 and the maximum degree is 50. We generated graphs with 1000 and 5000 vertices and for a given size, two different ranges for the community sizes are specified: "small" communities whose sizes are in between 10 and 50 vertices and "large" communities whose sizes are in between 20 and 100 vertices. For a given type of graph, the mixing parameter is varied from 0.1 to 0.8, in increments of 0.1. Hence, for each type we have 8 different graphs. We generated 4 sets of graphs with different exponents for the degree distribution (γ) and community size distribution (β). γ is usually in the range [2, 3] whereas β is in the range [1, 2]. These values are the most often used in the literature. To be able to test the algorithm across a wide range of graphs, we set the combination (γ, β) to the following values: (2,1), (2,2), (3,1) and (3,2).

Hence, for a combination of γ and β, we have 4 different types of graphs, with

1000 and 5000 vertices, having small and big communities. For each type, the mixing parameter is varied from 0.1 to 0.8 giving a total of 32 instances for a given combination, for a total 128 instances for all 4 types of graph. The parameters chosen here were adopted from [26].

We have further tested the algorithm on 8 real world networks which have been widely used previously. These graphs range from 30 to 10000 vertices. The following are the real world networks used, their sizes are shown in Table 5.1:

- Zachary's Karate Club [53]: a social network of friendships between 34 members of a karate club at a US university.
- Dolphin Social Network [28]: a social network of frequent associations between 62 dolphins in a community living off Doubtful Sound, New Zealand.
- Les Miserables [25]: This is a coapperance network of characters in the novel Les Miserables. The vertices represent characters and two vertices are joined by an edge if they interacted in atleast one of the chapters in the book. This network has weighted edges.
- Books about US politics [1]: A network of books about US politics published around the time of the 2004 presidential election and sold by Amazon. Edges between books represent frequent copurchasing of books by the same buyers. The network was compiled by V. Krebs and is unpublished.
- American College Football: A network of American football games between Division I colleges during Fall 2000 [18]. The original graph had two issues which were corrected in [15].
- Email URV [21]: a network of e-mail interchanges between members of the Univeristy Rovira i Virgili (Tarragona), Spain.
- Power Grid [52]: a network representing the topology of the Western States Power Grid of the United States.

Table 5.1: Network Sizes

Network	$\|V\|$	$\|E\|$
Karate Club	34	78
Dolphin Social Network	62	159
Les Miserables	72	254
Political Books	105	441
College Football	115	613
Email URV	1133	5451
Power Grid	4941	6594
PGP	10680	24316

- PGP Network [4]: giant component of the network of users of the Pretty-Good-Privacy (PGP) algorithm for secure information interchange.

We have also compared the performance of our algorithm on the dataset with six other community detection algorithms:

- Leading Eigenvector [32]: This algorithm maximizes modularity by using a spectral graph theory approach. It optimizes a modularity matrix instead of focusing on the eigenvectors of the Laplacian matrix as used by other spectral approaches for graph partitioning.

- Fast Greedy [8]: This is an agglomerative hierarchical method (see section 2.2) which maximizes modularity in a greedy manner. Each vertex is assigned the community to which the gain in modularity is maximum.

- Louvain [3]: This is one of the most popular modularity maximization technique which is an extension of Fast Greedy. It uses a two phase agglomerative hierarchical method where the first phase is similar to that used by Fast Greedy. During the second phase, it builds a new graph who vertices represent the communities found in the first phase and the process of phase one is repeated.

- Infomap [47]: It is an information theoretic approach for finding communities as described in Chapter 2. The algorithm compresses the information of a dynamic process on the graph, a random walk. The optimal compression is achieved by

optimizing a function, which is the minimum description length (MDL) [45] of the random walk. The community structure is represented by a two-level description based on Huffman coding [23]. One level is used to distinguish the communities in the network and the other to distinguish nodes within a community.

- Walktrap [39]: This is a hierarchical agglomerative method that uses random walks to find communities in complex networks. The idea here is that a random walker will tend to stay in the same community. The number of steps taken by a random walker is set to 4, which is the default value in the implementation.

- Label Propagation [43]: This is a heuristic algorithm that uses network structure alone for finding communities. It works by labeling the vertices with unique labels and updating those labels by majority voting with the neighbors of each vertex. In the end, vertices having the same label are assigned a community.

5.2 Setup

Our algorithm is implemented in C++ and compiled with the flag -std=c++0x. The tests were run on a 3.4 GHz Core i5-3570 machine with 32 GB of RAM running Ubuntu 12.04. We have used the "igraph" library [10] , which is a software package in R [41] for complex networks that comes built in with the above mentioned algorithms. For synthetic networks the community structure is evaluated by computing the normalized mutual information (NMI) and for real world networks we report the modularity obtained. We have run our algorithm, along with Infomap and Label Propagation 50 times on each instance. The remaining 4 algorithms return the same community structure on every run.

5.3 Results

For all the instances in the LFR Benchmark we have shown graphs that show the change in NMI as the mixing parameter is increased. Note that each point in the plot corresponds to the best result obtained over 50 runs. For the real world networks we report the modularity and the number of communities obtained by all the algorithms. Additionally, for each instance, we have reported the best, mean, standard deviation and the coefficient of variance (CV) of ABCD, Infomap and Label Propagation in the appendix. We define the coefficient of variation as the ratio (as a percentage) of the standard deviation and mean of the results over 50 runs for each instance.

5.3.1 LFR Benchmark

In this section we compare the results obtained by ABCD against the best known results from the six other algorithms. It should be noted that the best known values were not obtained by one algorithm alone. Based on previous work in the literature, Infomap and Louvain have been indentified to be among the best performing algorithms on the LFR benchmark [26].

Overall, on the LFR Benchmark our algorithm matches the previously best known values, in 71.9% of the instances. ABCD finds better than the best-known values in 10.1% of the instances, and the second best values in 16.4% of the instances. For the remaining 1.6% of the instances ABCD finds the third best values. There are 4 instances for which we could not obtain the data for the Leading Eigenvector algorithm as igraph reported an error stating the maximum number of iterations had exceeded for those instances.

Figures 5.1 to 5.4 show the performance of all algorithms as the mixing parameter is varied from 0.1 to 0.8 for all four combinations of (γ, β). ABCD performs consistently well across most of the instances though the NMI drops off after $\mu > 0.6$ as for these instances each vertex has 30% or lesser of its outgoing edges to vertices in its own community and the community structure of the graph is very fuzzy.

Tables 5.2 to 5.17 show the best NMI values obtained by each algorithm on all

the LFR benchmark instances. Numbers in bold indicate the highest NMI values obtained. On the instances where ABCD performs better the best-known values, the improvement in NMI ranges from 0.005 to 0.2063. On the instances where ABCD performs second best, the difference in NMI from the best known values range from 0.0053 to 0.2281.

The tables in the appendix show the mean, best, standard deviation and coefficient of variation (CV) of ABCD, Infomap and Label Propagation. On instances with 1000 vertices having small communities, the CV of ABCD varies from 0 to 13.76, the CV of Infomap varies from 0 to 134.69 whereas the CV of Label Propagation varies from 0.05 to 190.26. On graphs with 1000 vertices having big communities, the CV of ABCD varies from 0 to 24.45, the CV of Infomap varies from 0 to 123.72 and that of Label Propgation varies from 0 to 109.81.

For the instances with 5000 vertices having small communities, the CV of ABCD varies from 0 to 6.39 whereas the CV of Infomap is in the range 0 to 1.35 and that of Label Propagation varies from 0 to 118.72. Finally, on the instances with 5000 vertices having big communities, the CV of ABCD ranges from 0 to 30.14, the CV of Infomap ranges from 0 to 56.81 and the CV of Label Propagation varies from 0 to 20.64.

5.3.2 Real World Networks

On the real world networks, ABCD finds the community structure with the highest modularity value for two instances, the Email URV network and the network on political books. The increase in modularity is 0.014 and 0.0034 respectively. ABCD found the second highest modularity on four of the networks with difference from the best known ranging from 0.0009 to 0.0273. On the remaining two instances, ABCD found the third highest modularity values with the difference from the best known being 0.0068 and 0.025.

On the college football network where the number of communities is known to be 11, ABCD found the correct community structure with a modularity of 0.6032. However, the Louvain method found the community structure with a modularity of 0.6046

having 10 communities. For the Les Miserables network, ABCD found the community structure with a modularity of 0.5487 whereas the Louvain method found the highest modularity value of 0.5555. The difference in modularity can be attributed to the network being weighted and ABCD does not take edge weights into account whereas the Louvain method does.

Table 5.18 shows the modularity values for all 7 algorithms and the number of communities found by each one, denoted by N_C. It can be seen that N_C values for all algorithms on the smaller networks are very close to each other. The difference in N_C for the various algorithms becomes significant as the networks get larger with modularity maximization approaches favoring larger community sizes. This is consistent with the resolution limit of modularity as mentioned in the previous section.

5.3.3 Discussion

This section summarizes the performance of our algorithm against Infomap and the Louvain method both of which have been identified to be amongst the best performing algorithms on the LFR benchmark [26].

ABCD vs. Infomap

Out of the 64 instances of the LFR benchmark consisting of 5000 vertices, ABCD obtains the same value as Infomap on 48 instances, gets higher values on 3 instances and gets lower values than Infomap on the remaining 13 instances. The CV of ABCD for these instances varies from 0 to 30.41 whereas that of Infomap varies from 0 to 56.81.

On the remaining 64 instances of the LFR benchmark consisting of 1000 vertices, ABCD obtains the same value as Infomap on 44 instances, gets higher values than Infomap on 14 instances and does worse than Infomap on the remaining 6 instances. However, ABCD obtains a higher mean than Infomap on 4 out of these 6 instances. Overall, on these instances, the CV of ABCD varies from 0 to 24.45 whereas the CV of Infomap varies from 0 to 134.69.

On the real world networks, ABCD obtains a higher modularity value than Infomap on 6 out of the 8 instances and gets a lower modularity value on the remaining 2 instances.

ABCD vs. Louvain

On the synthetic graphs with 1000 vertices, ABCD obtains the same value as Louvain on 18 instances and performs better on the remaining 46 instances.

On the synthetic graphs with 5000 vertices, ABCD performs better than Louvain on all 64 instances. This can be explained due to the resolution limit of modularity which becomes worse on larger networks. The number of communities found by Louvain is lesser than the number of planted communities due to modularity maximization merging communities.

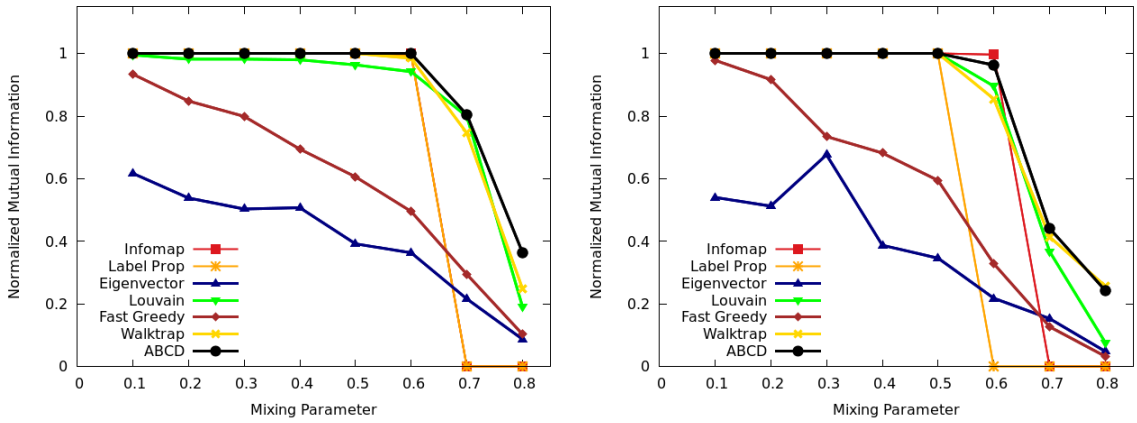
On the real world networks, ABCD gets a higher modularity than Louvain on 3 instances whereas it gets a lower modularity on the remaining 5. The difference in the number of communities found for the larger networks is significant as the Louvain method finds the community structure with a higher modularity but a smaller number of communities due to the resolution limit imposed by modularity maximization. While ABCD uses modularity as the criterion to measure improvement, it does not suffer from the resolution limit and is still able to find a community structure with a high modularity value.

5.3.4 Summary

Based on the above observations, it can be seen that ABCD is robust as it is able to consistently perform well across a wide range of graphs. While Infomap has a good performance on the LFR benchmark, especially on the instances with 5000 vertices, ABCD performs well against Infomap on those instances and outperforms Infomap on the instances with 1000 vertices. In addition, on the real world networks, ABCD obtains a higher modularity value on 6 out of the 8 instances. The difference in the number of communities found on the larger instances is significant. For example,

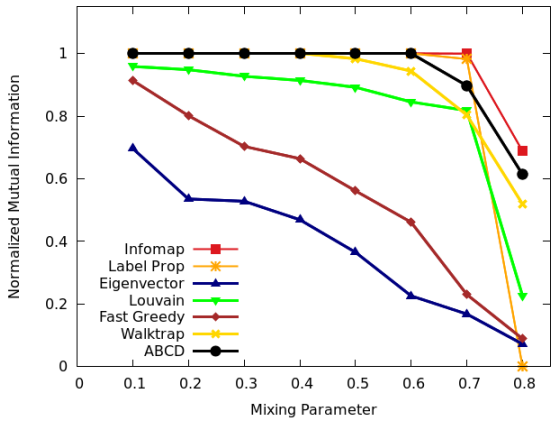
on the PGP network, Infomap finds 1074 communities with a modularity of 0.8034. However, a large number of the communities found are very small (having sizes 2 or 3). ABCD on the other hand finds 408 communities with a modularity of 0.8561.

ABCD performs better than Louvain on the LFR benchmark in general. On the real world networks, Louvain obtains a higher modularity on 5 instances. The difference in the number of communities found here is also significant. Since Louvain explicitly maximizes modularity, it suffers from the resolution limit of modularity and always returns communities having larger sizes which result in the higher modularity value. While our algorithm uses modularity to measure improvement, we do not explicitly maximize modularity alone and hence ABCD does not suffer from the resolution limit imposed by modularity. This is evident from the results. For example, ABCD finds the highest modularity value on the Email URV network (0.5529) with 24 communities whereas Louvain finds the second highest modularity value (0.5426) with 11 communities. Infomap on the other hand finds 63 communities with a modularity of 0.5319.

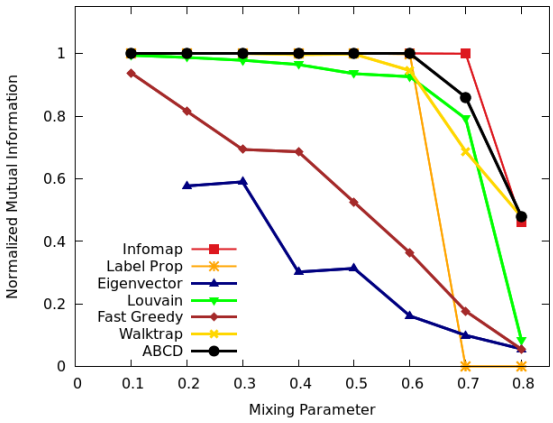


(a) NMI: 1000-Small

(b) NMI: 1000-Big

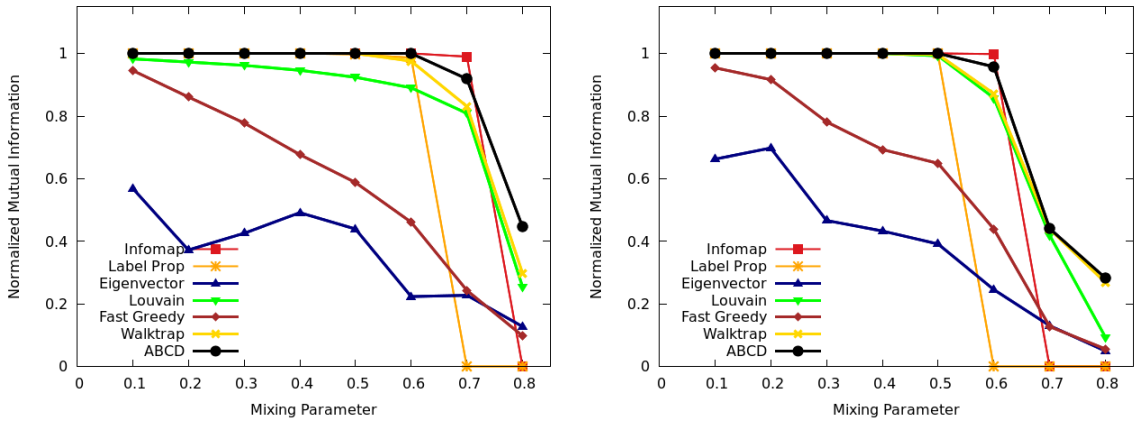


(c) NMI: 5000-Small



(d) NMI: 5000-Big

Figure 5.1: NMI Results on LFR Benchmark with $(\gamma, \beta) = (2, 1)$



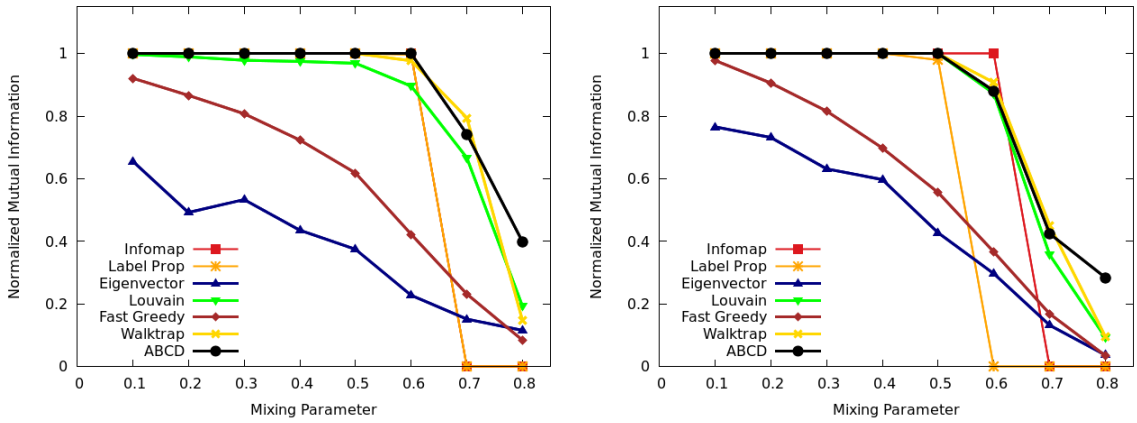
(a) NMI: 1000-Small

(b) NMI: 1000-Big

(c) NMI: 5000-Small

(d) NMI: 5000-Big

Figure 5.2: NMI Results on LFR Benchmark with $(\gamma, \beta) = (2, 2)$



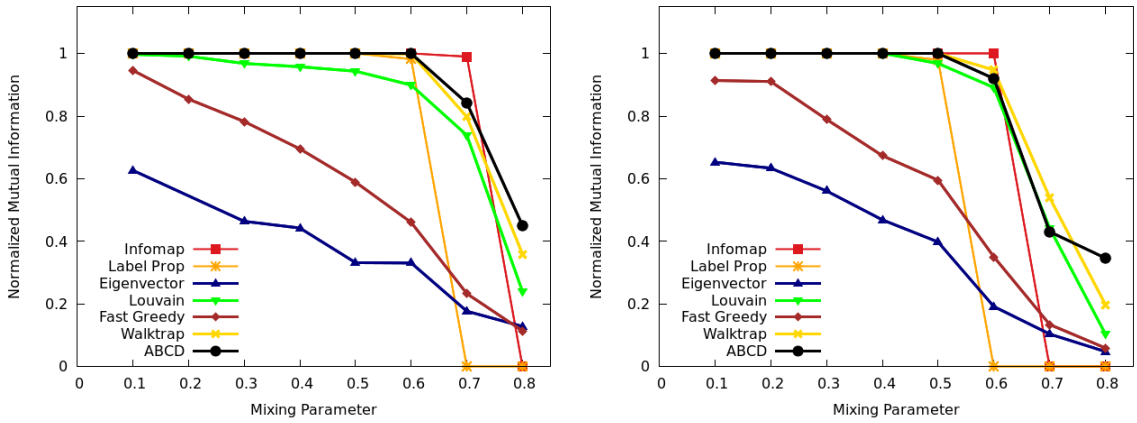
(a) NMI: 1000-Small

(b) NMI: 1000-Big

(c) NMI: 5000-Small

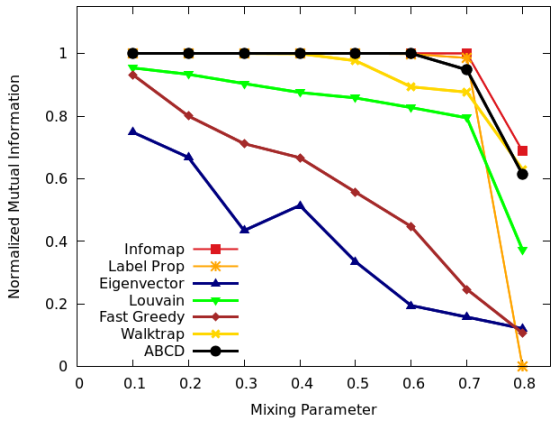
(d) NMI: 5000-Big

Figure 5.3: NMI Results on LFR Benchmark with $(\gamma, \beta) = (3, 1)$

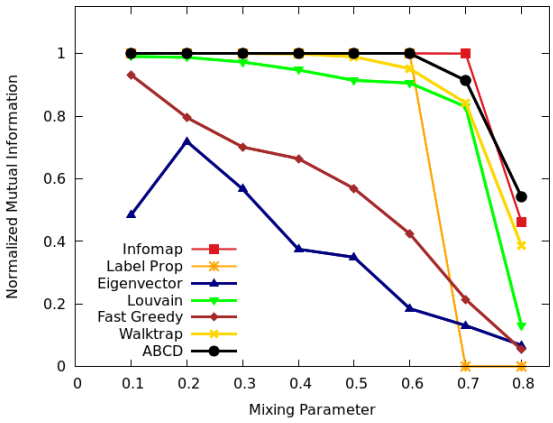


(a) NMI: 1000-Small

(b) NMI: 1000-Big



(c) NMI: 5000-Small



(d) NMI: 5000-Big

Figure 5.4: NMI Results on LFR Benchmark with $(\gamma, \beta) = (3, 2)$

Table 5.2: Best NMI Values: 1000-Small($\gamma = 2, \beta = 1$)

μ	ABCD	Infomap	Label Prop	Eigenvector	Louvain	FG	Walktrap
0.1	1	1	1	0.6175	0.9954	0.9341	1
0.2	1	1	1	0.5388	0.9821	0.8483	1
0.3	1	1	1	0.5036	0.9823	0.7992	1
0.4	1	1	1	0.5078	0.9800	0.6947	1
0.5	1	1	1	0.3923	0.9636	0.6063	1
0.6	1	1	0.9927	0.3639	0.9420	0.4957	0.9847
0.7	**0.8044**	0	0	0.2166	0.7990	0.2949	0.7457
0.8	**0.4362**	0	0	0.0878	0.1911	0.1035	0.2486

Table 5.3: Best NMI Values: 1000-Big($\gamma = 2, \beta = 1$)

μ	ABCD	Infomap	Label Prop	Eigenvector	Louvain	FG	Walktrap
0.1	1	1	1	0.5406	1	0.9784	1
0.2	1	1	1	0.5128	1	0.9168	1
0.3	1	1	1	0.6766	1	0.7351	1
0.4	1	1	1	0.3879	1	0.6824	1
0.5	1	1	1	0.3465	1	0.5946	1
0.6	0.9630	**0.9967**	0	0.2186	0.8956	0.3286	0.8549
0.7	**0.4414**	0	0	0.1535	0.3674	0.1274	0.4143
0.8	0.2426	0	0	0.0487	0.0766	0.0328	**0.2572**

Table 5.4: Best NMI Values: 5000-Small($\gamma = 2, \beta = 1$)

μ	ABCD	Infomap	Label Prop	Eigenvector	Louvain	FG	Walktrap
0.1	1	1	1	0.6968	0.9586	0.9139	1
0.2	1	1	1	0.5358	0.9483	0.8015	1
0.3	1	1	1	0.5284	0.9272	0.7035	1
0.4	1	1	1	0.4692	0.9141	0.6642	1
0.5	1	1	1	0.3655	0.8923	0.5614	0.9834
0.6	1	1	1	0.2262	0.8453	0.4612	0.9439
0.7	0.8973	**0.9997**	0.9823	0.1682	0.8179	0.2305	0.8033
0.8	0.6138	**0.6883**	0	0.0724	0.2263	0.0892	0.5191

Table 5.5: Best NMI Values: 5000-Big($\gamma = 2, \beta = 1$)

μ	ABCD	Infomap	Label Prop	Eigenvector	Louvain	FG	Walktrap
0.1	1	1	1	-	0.9943	0.9382	1
0.2	1	1	1	0.5769	0.9874	0.8159	1
0.3	1	1	1	0.5903	0.9785	0.6937	1
0.4	1	1	1	0.3027	0.9645	0.6864	0.9967
0.5	1	1	1	0.3142	0.9356	0.5253	0.9979
0.6	1	1	1	0.1623	0.9259	0.3637	0.9457
0.7	0.8602	**0.9996**	0	0.1004	0.7919	0.1775	0.6875
0.8	**0.4798**	0.4632	0	0.0567	0.0846	0.0558	0.4784

Table 5.6: Best NMI Values: 1000-Small($\gamma = 2, \beta = 2$)

μ	ABCD	Infomap	Label Prop	Eigenvector	Louvain	FG	Walktrap
0.1	1	1	1	0.5694	0.9828	0.9457	1
0.2	1	1	1	0.3726	0.9726	0.8611	1
0.3	1	1	1	0.4261	0.9621	0.7783	1
0.4	1	1	1	0.4918	0.9464	0.6773	1
0.5	1	1	0.9971	0.4397	0.9241	0.5882	1
0.6	1	1	0.9867	0.2242	0.8906	0.4614	0.9756
0.7	0.9189	**0.99**	0	0.2282	0.8090	0.2437	0.8314
0.8	**0.4463**	0	0	0.1283	0.2552	0.0986	0.2979

Table 5.7: Best NMI Values: 1000-Big($\gamma = 2, \beta = 2$)

μ	ABCD	Infomap	Label Prop	Eigenvector	Louvain	FG	Walktrap
0.1	1	1	1	0.6624	1	0.9543	1
0.2	1	1	1	0.6983	1	0.9168	1
0.3	1	1	1	0.4669	1	0.7818	1
0.4	1	1	1	0.4334	1	0.6934	1
0.5	1	1	1	0.3918	0.9929	0.6493	0.9966
0.6	0.9577	**0.9983**	0	0.2461	0.8578	0.4397	0.8716
0.7	**0.4410**	0	0	0.1310	0.4194	0.1280	0.4394
0.8	**0.2831**	0	0	0.0502	0.0942	0.0568	0.2685

Table 5.8: Best NMI Values: 5000-Small($\gamma = 2, \beta = 2$)

μ	ABCD	Infomap	Label Prop	Eigenvector	Louvain	FG	Walktrap
0.1	1	1	1	0.6939	0.9395	0.8986	1
0.2	1	1	1	0.5956	0.9142	0.8033	1
0.3	1	1	1	-	0.9049	0.7057	1
0.4	1	1	1	0.3768	0.8922	0.6602	0.9992
0.5	1	1	1	0.2985	0.8645	0.6289	1
0.6	1	1	0.9965	0.2972	0.8280	0.5099	0.9691
0.7	0.9586	**1**	0.9853	0.1652	0.7951	0.2990	0.8392
0.8	0.6845	**0.8241**	0	0.0907	0.3507	0.1136	0.6516

Table 5.9: Best NMI Values: 5000-Big($\gamma = 2, \beta = 2$)

μ	ABCD	Infomap	Label Prop	Eigenvector	Louvain	FG	Walktrap
0.1	1	1	1	0.6824	0.9849	0.9263	1
0.2	1	1	1	0.6046	0.9786	0.7894	1
0.3	1	1	1	0.6532	0.9629	0.7406	1
0.4	1	1	1	0.3687	0.9366	0.6267	1
0.5	1	1	1	0.2745	0.9221	0.5718	0.9991
0.6	1	1	0.9991	0.1989	0.8957	0.4292	0.9821
0.7	0.9199	**0.9991**	0	0.09	0.7635	0.1774	0.6876
0.8	0.5040	**0.5093**	0	0.0654	0.1249	0.0589	0.4925

Table 5.10: Best NMI Values: 1000-Small($\gamma = 3, \beta = 1$)

μ	ABCD	Infomap	Label Prop	Eigenvector	Louvain	FG	Walktrap
0.1	1	1	1	0.6561	0.9969	0.9211	1
0.2	1	1	1	0.4924	0.9894	0.8664	1
0.3	1	1	1	0.5336	0.9784	0.8078	1
0.4	1	1	1	0.4358	0.9748	0.7237	1
0.5	1	1	1	0.3749	0.9689	0.6186	1
0.6	1	1	0.9965	0.2281	0.8957	0.4229	0.9768
0.7	0.7426	0	0	0.1520	0.6669	0.2320	**0.7945**
0.8	**0.3989**	0	0	0.1161	0.1926	0.0848	0.1468

Table 5.11: Best NMI Values: 1000-Big($\gamma = 3, \beta = 1$)

μ	ABCD	Infomap	Label Prop	Eigenvector	Louvain	FG	Walktrap
0.1	1	1	1	0.7657	1	0.9779	1
0.2	1	1	1	0.7320	1	0.9058	1
0.3	1	1	1	0.6318	1	0.8162	1
0.4	1	1	1	0.5976	1	0.6983	1
0.5	1	1	0.9794	0.4278	1	0.5562	1
0.6	0.8806	**1**	0	0.2970	0.8719	0.3661	0.9072
0.7	0.4246	0	0	0.1324	0.3572	0.1689	**0.4514**
0.8	**0.2844**	0	0	0.0373	0.0929	0.0353	0.0956

Table 5.12: Best NMI Values: 5000-Small($\gamma = 3, \beta = 1$)

μ	ABCD	Infomap	Label Prop	Eigenvector	Louvain	FG	Walktrap
0.1	1	1	1	0.7694	0.9685	0.9251	1
0.2	1	1	1	0.6384	0.9533	0.7837	1
0.3	1	1	1	-	0.9302	0.7003	1
0.4	1	1	1	0.3684	0.9099	0.6429	0.9989
0.5	1	1	1	0.4449	0.8923	0.5614	0.9834
0.6	1	1	0.9979	0.1958	0.8395	0.4588	0.9257
0.7	0.8973	**0.9998**	0.9823	0.1272	0.8130	0.2293	0.8359
0.8	0.6138	**0.6883**	0	0.0899	0.2471	0.0806	0.5128

Table 5.13: Best NMI Values: 5000-Big($\gamma = 3, \beta = 1$)

μ	ABCD	Infomap	Label Prop	Eigenvector	Louvain	FG	Walktrap
0.1	1	1	1	0.7004	0.9971	0.9202	1
0.2	1	1	1	0.5930	0.9929	0.7987	1
0.3	1	1	1	0.3619	0.9837	0.7125	1
0.4	1	1	1	0.3778	0.9698	0.6902	1
0.5	1	1	1	0.2606	0.9439	0.6459	1
0.6	1	1	0.9979	0.1539	0.9049	0.3815	0.9745
0.7	0.7717	**0.9998**	0	0.0963	0.7484	0.1567	0.7697
0.8	**0.4769**	0.4632	0	0.0757	0.0640	0.0419	0.4274

Table 5.14: Best NMI Values: 1000-Small($\gamma = 3, \beta = 2$)

μ	ABCD	Infomap	Label Prop	Eigenvector	Louvain	FG	Walktrap
0.1	1	1	1	0.6270	0.9974	0.9462	1
0.2	1	1	1	-	0.9914	0.8546	1
0.3	1	1	1	0.4644	0.9683	0.7828	1
0.4	1	1	1	0.4429	0.9577	0.6956	1
0.5	1	1	1	0.3325	0.9435	0.5894	1
0.6	1	1	0.9821	0.3313	0.8992	0.4608	0.9988
0.7	0.8418	**0.99**	0	0.1772	0.7388	0.2332	0.7974
0.8	**0.4505**	0	0	0.1287	0.2398	0.1144	0.3591

Table 5.15: Best NMI Values: 1000-Big($\gamma = 3, \beta = 2$)

μ	ABCD	Infomap	Label Prop	Eigenvector	Louvain	FG	Walktrap
0.1	1	1	1	0.6531	1	0.9139	1
0.2	1	1	1	0.6338	1	0.9105	1
0.3	1	1	1	0.5613	1	0.7897	1
0.4	1	1	1	0.4689	1	0.6738	1
0.5	1	1	0.9794	0.3983	0.9683	0.5956	1
0.6	0.92	**1**	0	0.1921	0.8913	0.3509	0.9472
0.7	0.4310	0	0	0.1044	0.4422	0.1347	**0.5398**
0.8	**0.3467**	0	0	0.0484	0.1044	0.0598	0.1982

Table 5.16: Best NMI Values: 5000-Small($\gamma = 3, \beta = 2$)

μ	ABCD	Infomap	Label Prop	Eigenvector	Louvain	FG	Walktrap
0.1	1	1	1	0.7487	0.9541	0.9324	1
0.2	1	1	1	0.6684	0.9335	0.8004	1
0.3	1	1	1	0.4347	0.9039	0.7121	1
0.4	1	1	1	0.5151	0.8755	0.6672	0.9992
0.5	1	1	0.9995	0.3354	0.8582	0.5573	0.9771
0.6	1	1	0.9984	0.1954	0.8275	0.4476	0.8936
0.7	0.9480	**1**	0.9859	0.1587	0.7946	0.2474	0.8766
0.8	0.6138	**0.6883**	0	0.1216	0.3731	0.1078	0.6301

Table 5.17: Best NMI Values: 5000-Big($\gamma = 3, \beta = 2$)

μ	ABCD	Infomap	Label Prop	Eigenvector	Louvain	FG	Walktrap
0.1	1	1	1	0.4841	0.9904	0.9319	1
0.2	1	1	1	0.7197	0.9879	0.7955	1
0.3	1	1	1	0.5674	0.9727	0.7015	1
0.4	1	1	1	0.3752	0.9472	0.6644	0.9985
0.5	1	1	1	0.3494	0.9143	0.5689	0.9898
0.6	1	1	1	0.1854	0.9056	0.4241	0.9515
0.7	0.9137	**0.9998**	0	0.1316	0.8299	0.2149	0.8420
0.8	**0.5422**	0.4632	0	0.0685	0.1312	0.0546	0.3863

Table 5.18: Best Modularity Values

Network	ABCD		Infomap		Label Prop		Eigenvector		Louvain		CNM		Walktrap	
	Modularity	N_C	Modularity	N_C	Modularity	N_C	Modularity	N_C	Modularity	N_C	Modularity	N_C	Modularity	N_C
Karate Club	0.4151	4	0.402	4	0.402	4	0.3934	4	**0.4188**	4	0.3807	3	0.3532	5
Dolphin Social Network	0.5268	4	**0.5277**	5	**0.5277**	4	0.4911	5	0.5185	5	0.4955	4	0.4888	4
Les Miserables	0.5487	8	0.5513	8	0.5462	6	0.5322	8	**0.5555**	6	0.5006	5	0.5214	8
Political Books	**0.5262**	4	0.5228	6	0.5228	4	0.4672	4	0.5205	4	0.5019	4	0.5069	4
College Football	0.6032	11	0.6005	11	0.6005	11	0.4926	8	**0.6046**	10	0.5497	6	0.6029	10
Email URV	**0.5529**	24	0.5319	63	0.5254	20	0.4888	7	0.5426	11	0.4942	16	0.5307	49
Power Grid	0.9112	109	0.8193	480	0.8156	488	0.8252	35	**0.9362**	40	0.9335	41	0.8309	364
PGP	0.8561	408	0.8034	1074	0.8170	939	0.6798	25	**0.8834**	99	0.8525	205	0.7894	1574

Appendix A

Performance Tables of ABCD

Table A.1: NMI: 1000-Small($\gamma = 2, \beta = 1$)

μ	ABCD				InfoMap				Label Prop			
	Mean	Best	σ	CV	Mean	Best	σ	CV	Mean	Best	σ	CV
0.1	0.9998	1	0.0004	0.04	1	1	0	0	0.9999	1	0.0005	0.05
0.2	0.9998	1	0.0004	0.04	1	1	0	0	0.9993	1	0.0016	0.15
0.3	0.9999	1	0.0003	0.03	1	1	0	0	0.9972	1	0.0043	0.43
0.4	0.9998	1	0.0006	0.06	1	1	0	0	0.9919	1	0.0098	0.98
0.5	0.9997	1	0.0013	0.13	1	1	0	0	0.9847	1	0.0111	1.13
0.6	0.9997	1	0.0042	0.42	1	1	0	0	0.4263	0.9927	0.4675	109.66
0.7	0.7379	0.8044	0.0383	5.19	0	0	0	-	0	0	0	-
0.8	0.3640	0.4362	0.0376	10.33	0	0	0	-	0	0	0	-

Table A.2: NMI: 1000-Big($\gamma = 2, \beta = 1$)

μ	ABCD				InfoMap				Label Prop			
	Mean	Best	σ	CV	Mean	Best	σ	CV	Mean	Best	σ	CV
0.1	1	1	0	0	1	1	0	0	0.9993	1	0.0035	0.35
0.2	1	1	0	0	1	1	0	0	0.9998	1	0.0017	0.17
0.3	1	1	0	0	1	1	0	0	0.9987	1	0.0043	0.42
0.4	0.9998	1	0.0013	0.13	1	1	0	0	0.9897	1	0.0122	1.23
0.5	0.9821	1	0.0212	2.16	1	1	0	0	0.4794	1	0.4521	94.29
0.6	0.7665	0.9630	0.0957	12.48	0.6778	0.9967	0.4697	69.29	0	0	0	-
0.7	0.3834	0.4414	0.0271	7.07	0	0	0	-	0	0	0	-
0.8	0.1827	0.2426	0.0289	15.82	0	0	0	-	0	0	0	-

Table A.3: NMI: 5000-Small($\gamma = 2, \beta = 1$)

μ	ABCD				InfoMap				Label Prop			
	Mean	Best	σ	CV	Mean	Best	σ	CV	Mean	Best	σ	CV
0.1	1	1	0	0	1	1	0	0	0.9999	1	0.0002	0.02
0.2	1	1	0	0	1	1	0	0	0.9997	1	0.0004	0.04
0.3	1	1	0	0	1	1	0	0	0.9994	1	0.0006	0.06
0.4	0.9999	1	0.0001	0.01	1	1	0	0	0.9989	1	0.001	0.09
0.5	0.9998	1	0.0002	0.02	1	1	0	0	0.9976	1	0.0014	0.14
0.6	0.9994	1	0.0005	0.05	1	1	0	0	0.9940	1	0.0023	0.23
0.7	0.8795	0.8973	0.0104	1.18	0.9997	0.9997	0	0	0.4037	0.9823	0.4793	118.72
0.8	0.5815	0.6138	0.0194	3.33	0.6677	0.6883	0.0083	1.24	0	0	0	-

Table A.4: NMI: 5000-Big($\gamma = 2, \beta = 1$)

μ	ABCD				InfoMap				Label Prop			
	Mean	Best	σ	CV	Mean	Best	σ	CV	Mean	Best	σ	CV
0.1	1	1	0	0	1	1	0	0	1	1	0	0
0.2	1	1	0	0	1	1	0	0	0.9998	1	0.0005	0.05
0.3	0.9999	1	0.0002	0.02	1	1	0	0	0.9998	1	0.0006	0.06
0.4	0.9998	1	0.0004	0.04	1	1	0	0	0.9993	1	0.0009	0.09
0.5	0.9995	1	0.0008	0.08	1	1	0	0	0.9981	1	0.0019	0.19
0.6	0.9983	1	0.0019	0.19	1	1	0	0	0.9472	0.9979	0.1955	20.64
0.7	0.8013	0.8602	0.034	4.25	0.9991	0.9996	0.0003	0.03	0	0	0	-
0.8	0.4447	0.4798	0.0188	4.22	0.3404	0.4632	0.1934	56.81	0	0	0	-

Table A.5: NMI: 1000-Small($\gamma = 2, \beta = 2$)

μ	ABCD				InfoMap				Label Prop			
	Mean	Best	σ	CV	Mean	Best	σ	CV	Mean	Best	σ	CV
0.1	0.9993	1	0.0009	0.09	1	1	0	0	0.9998	1	0.0008	0.08
0.2	0.9993	1	0.0007	0.07	1	1	0	0	0.9994	1	0.0015	0.15
0.3	1	1	0	0	1	1	0	0	0.9972	1	0.0034	0.34
0.4	1	1	0.0002	0.02	1	1	0	0	0.9932	1	0.0052	0.52
0.5	0.9999	1	0.0005	0.05	1	1	0	0	0.9835	0.9971	0.0123	1.25
0.6	0.9965	1	0.0030	0.30	1	1	0	0	0.6793	0.9867	0.4286	63.09
0.7	0.8057	0.9189	0.0545	6.76	0.5951	0.99	0.4908	82.48	0	0	0	-
0.8	0.3864	0.4463	0.0387	10.02	0	0	0	-	0	0	0	-

Table A.6: NMI: 1000-Big($\gamma = 2, \beta = 2$)

μ	ABCD				InfoMap				Label Prop			
	Mean	Best	σ	CV	Mean	Best	σ	CV	Mean	Best	σ	CV
0.1	1	1	0	0	1	1	0	0	1	1	0	0
0.2	1	1	0	0	1	1	0	0	0.9993	1	0.0027	0.27
0.3	0.9996	1	0.0012	0.12	1	1	0	0	0.9985	1	0.0042	0.42
0.4	0.9998	1	0.0012	0.12	1	1	0	0	0.99	1	0.0132	1.32
0.5	0.9989	1	0.0032	0.32	1	1	0	0	0.88	1	0.2314	94.29
0.6	0.8125	0.9577	0.0805	9.91	0.9774	0.9983	0.1411	14.43	0	0	0	-
0.7	0.3828	0.4410	0.0253	6.62	0	0	0	-	0	0	0	-
0.8	0.2219	0.2831	0.0542	24.45	0	0	0	-	0	0	0	-

Table A.7: NMI: 5000-Small($\gamma = 2, \beta = 2$)

μ	ABCD				InfoMap				Label Prop			
	Mean	Best	σ	CV	Mean	Best	σ	CV	Mean	Best	σ	CV
0.1	1	1	0	0	1	1	0	0	0.9998	1	0.0003	0.03
0.2	1	1	0	0	1	1	0	0	0.9995	1	0.0004	0.04
0.3	1	1	0	0	1	1	0	0	0.9993	1	0.0006	0.06
0.4	0.9994	1	0.0005	0.05	1	1	0	0	0.9986	1	0.001	0.09
0.5	0.9998	1	0.0003	0.03	1	1	0	0	0.9971	0.9997	0.0011	0.11
0.6	0.9996	1	0.0004	0.04	1	1	0	0	0.9929	0.9965	0.002	0.2
0.7	0.9456	0.9586	0.0046	0.49	1	1	0	0	0.8728	0.9853	0.294	33.69
0.8	0.6624	0.6845	0.0141	2.13	0.7999	0.8241	0.0098	1.22	0	0	0	-

Table A.8: NMI: 5000-Big($\gamma = 2, \beta = 2$)

μ	ABCD				InfoMap				Label Prop			
	Mean	Best	σ	CV	Mean	Best	σ	CV	Mean	Best	σ	CV
0.1	1	1	0	0	1	1	0	0	0.9999	1	0.0002	0.02
0.2	1	1	0	0	1	1	0	0	0.9999	1	0.0004	0.04
0.3	0.9999	1	0.0003	0.03	1	1	0	0	0.9997	1	0.0005	0.05
0.4	0.9998	1	0.0004	0.04	1	1	0	0	0.9992	1	0.0008	0.08
0.5	0.9996	1	0.0005	0.05	1	1	0	0	0.9979	1	0.0017	0.17
0.6	0.9986	1	0.0015	0.15	1	1	0	0	0.9894	0.9991	0.0098	0.99
0.7	0.8483	0.9199	0.0336	3.96	0.9982	0.9991	0.0004	0.04	0	0	0	-
0.8	0.4705	0.5040	0.0195	4.14	0.4511	0.5093	0.1364	29.84	0	0	0	-

Table A.9: NMI: 1000-Small($\gamma = 3, \beta = 1$)

μ	ABCD				InfoMap				Label Prop			
	Mean	Best	σ	CV	Mean	Best	σ	CV	Mean	Best	σ	CV
0.1	1	1	0.0002	0.02	1	1	0	0	0.9999	1	0.0007	0.07
0.2	0.9998	1	0.0004	0.04	1	1	0	0	0.9995	1	0.0017	0.17
0.3	0.9997	1	0.0005	0.05	1	1	0	0	0.9971	1	0.0029	0.29
0.4	1	1	0	0	1	1	0	0	0.9958	1	0.0059	0.59
0.5	0.9991	1	0.0020	0.20	1	1	0	0	0.9824	1	0.0148	1.51
0.6	0.9925	1	0.0071	0.71	1	1	0	0	0.2065	0.9665	0.3928	190.26
0.7	0.6465	0.7426	0.0433	6.70	0	0	0	-	0	0	0	-
0.8	0.3192	0.3989	0.0439	13.76	0	0	0	-	0	0	0	-

Table A.10: NMI: 1000-Big($\gamma = 3, \beta = 1$)

μ	ABCD				InfoMap				Label Prop			
	Mean	Best	σ	CV	Mean	Best	σ	CV	Mean	Best	σ	CV
0.1	1	1	0	0	1	1	0	0	0.9996	1	0.002	0.2
0.2	1	1	0	0	1	1	0	0	1	1	0	0
0.3	0.9997	1	0.0015	0.15	1	1	0	0	0.9968	1	0.0076	0.76
0.4	0.9997	1	0.0016	0.16	1	1	0	0	0.9912	1	0.0139	1.40
0.5	0.9755	1	0.0242	2.48	1	1	0	0	0.4103	0.9794	0.4506	109.81
0.6	0.55	0.8806	0.0834	15.16	0.4	1	0.4949	123.72	0	0	0	-
0.7	0.3709	0.4246	0.0298	8.04	0	0	0	-	0	0	0	-
0.8	0.2355	0.2844	0.0354	15.02	0	0	0	-	0	0	0	-

Table A.11: NMI: 5000-Small($\gamma = 3, \beta = 1$)

μ	ABCD				InfoMap				Label Prop			
	Mean	Best	σ	CV	Mean	Best	σ	CV	Mean	Best	σ	CV
0.1	1	1	0	0	1	1	0	0	0.9999	1	0.0003	0.03
0.2	1	1	0	0	1	1	0	0	0.9997	1	0.0004	0.04
0.3	1	1	0	0	1	1	0	0	0.9995	1	0.0004	0.04
0.4	1	1	0	0	1	1	0	0	0.9988	1	0.0007	0.07
0.5	0.9998	1	0.0003	0.03	1	1	0	0	0.9975	1	0.0015	0.15
0.6	0.9995	1	0.0005	0.05	1	1	0	0	0.9946	0.9979	0.0016	0.16
0.7	0.7374	0.9466	0.0472	6.39	0.9998	0.9998	0	0	0.4037	0.9823	0.4793	118.72
0.8	0.5943	0.6425	0.0160	2.68	0.6685	0.6842	0.0085	1.27	0	0	0	-

Table A.12: NMI: 5000-Big($\gamma = 3, \beta = 1$)

μ	ABCD				InfoMap				Label Prop			
	Mean	Best	σ	CV	Mean	Best	σ	CV	Mean	Best	σ	CV
0.1	1	1	0	0	1	1	0	0	1	1	0.0001	0.01
0.2	1	1	0.0002	0.02	1	1	0	0	0.9999	1	0.0005	0.05
0.3	1	1	0.0002	0.02	1	1	0	0	0.9998	1	0.0006	0.06
0.4	0.9997	1	0.0006	0.06	1	1	0	0	0.9993	1	0.0012	0.12
0.5	0.9997	1	0.0006	0.06	1	1	0	0	0.9978	1	0.0019	0.19
0.6	0.9942	1	0.0038	0.38	1	1	0	0	0.9855	0.9979	0.1955	20.64
0.7	0.6840	0.7717	0.0308	4.51	0.9998	0.9998	0	0	0	0	0	-
0.8	0.4363	0.4769	0.0284	6.50	0.3793	0.4412	0.1279	33.71	0	0	0	-

Table A.13: NMI: 1000-Small($\gamma = 3, \beta = 2$)

μ	ABCD				InfoMap				Label Prop			
	Mean	Best	σ	CV	Mean	Best	σ	CV	Mean	Best	σ	CV
0.1	0.9998	1	0.0005	0.05	1	1	0	0	0.9998	1	0.001	0.09
0.2	0.9998	1	0.0006	0.06	1	1	0	0	0.9996	1	0.0014	0.14
0.3	0.9999	1	0.0003	0.03	1	1	0	0	0.9998	1	0.0036	0.36
0.4	1	1	0.0003	0.03	1	1	0	0	0.9951	1	0.0045	0.45
0.5	0.9997	1	0.0010	0.10	1	1	0	0	0.986	1	0.014	1.41
0.6	0.9968	1	0.0050	0.50	1	1	0	0	0.7218	0.9821	0.4103	56.85
0.7	0.7304	0.8418	0.0540	7.39	0.359	0.99	0.4836	134.69	0	0	0	-
0.8	0.3653	0.4505	0.0456	12.72	0	0	0	-	0	0	0	-

Table A.14: NMI: 1000-Big($\gamma = 3, \beta = 2$)

μ	ABCD				InfoMap				Label Prop			
	Mean	Best	σ	CV	Mean	Best	σ	CV	Mean	Best	σ	CV
0.1	1	1	0	0	1	1	0	0	1	1	0	0
0.2	1	1	0	0	1	1	0	0	1	1	0	0
0.3	0.9997	1	0.0015	0.15	1	1	0	0	0.9989	1	0.0031	0.31
0.4	0.9998	1	0.0011	0.11	1	1	0	0	0.9886	1	0.019	1.92
0.5	0.9897	1	0.0190	1.92	1	1	0	0	0.8825	1	0.2291	25.96
0.6	0.7130	0.92	0.0939	13.17	0.98	1	0.1414	14.43	0	0	0	-
0.7	0.3742	0.4310	0.0278	7.44	0	0	0	-	0	0	0	-
0.8	0.2749	0.3467	0.0391	14.23	0	0	0	-	0	0	0	-

Table A.15: NMI: 5000-Small($\tau_1 = 3, \tau_2 = 2$)

μ	ABCD				InfoMap				Label Prop			
	Mean	Best	σ	CV	Mean	Best	σ	CV	Mean	Best	σ	CV
0.1	1	1	0	0	1	1	0	0	0.9999	1	0.0003	0.03
0.2	1	1	0	0	1	1	0	0	0.9997	1	0.0004	0.04
0.3	1	1	0	0	1	1	0	0	0.9993	1	0.0006	0.06
0.4	1	1	0.0001	0.01	1	1	0	0	0.9986	1	0.0008	0.08
0.5	0.9999	1	0.0002	0.02	1	1	0	0	0.9977	0.9995	0.0012	0.12
0.6	0.9997	1	0.0004	0.04	1	1	0	0	0.9934	0.9984	0.0022	0.22
0.7	0.9364	0.9480	0.0064	0.69	1	1	0	0	0.7779	0.9859	0.3986	50.52
0.8	0.6551	0.6942	0.0155	2.37	0.7961	0.8170	0.0108	1.35	0	0	0	-

Table A.16: NMI: 5000-Big($\gamma = 3, \beta = 2$)

μ	ABCD				InfoMap				Label Prop			
	Mean	Best	σ	CV	Mean	Best	σ	CV	Mean	Best	σ	CV
0.1	1	1	0	0	1	1	0	0	0.9999	1	0.0002	0.02
0.2	1	1	0.0001	0.01	1	1	0	0	0.9999	1	0.0004	0.04
0.3	1	1	0	0	1	1	0	0	0.9998	1	0.0005	0.05
0.4	0.9999	1	0.0004	0.04	1	1	0	0	0.9996	1	0.0007	0.07
0.5	0.9998	1	0.0004	0.04	1	1	0	0	0.9983	1	0.0013	0.13
0.6	0.9975	1	0.0024	0.24	1	1	0	0	0.9897	1	0.0057	0.57
0.7	0.8559	0.9137	0.0296	3.46	0.9998	0.9998	0	0	0	0	0	-
0.8	0.4495	0.5422	0.1355	30.14	0.499	0.5145	0.0072	1.45	0	0	0	-

Table A.17: Real World Networks

Network	ABCD				InfoMap				Label Prop			
	Mean	Best	σ	CV	Mean	Best	σ	CV	Mean	Best	σ	CV
Karate	0.3947	0.4151	0.0132	3.34	0.402	0.402	0	0	0.3556	0.402	0.0844	23.74
Dolphins	0.5081	0.5267	0.0224	4.44	0.5247	0.5277	0.0032	0.61	0.4796	0.5277	0.0545	11.36
Les Miserables	0.5193	0.5487	0.0324	6.25	0.5471	0.5513	0.0019	0.36	0.4921	0.5462	0.0780	15.86
Political Books	0.5120	0.5262	0.0183	3.57	0.5228	0.5228	0	0	0.4891	0.5228	0.0159	3.24
Football	0.6	0.6032	0.0054	0.89	0.6005	0.6005	0	0	0.5921	0.6005	0.0137	2.32
Email URV	0.504	0.5529	0.024	4.77	0.5228	0.5319	0.0036	0.69	0.2568	0.5254	0.1962	76.38
Power Grid	0.9030	0.9112	0.0031	0.34	0.8166	0.8193	0.0012	0.15	0.8034	0.8156	0.0042	0.53
PGP	0.8504	0.8561	0.0026	0.31	0.8019	0.8034	0.0008	0.62	0.8039	0.8170	0.0046	0.57

References

[1] Books about US Politics. http://www.orgnet.com/.

[2] Albert-Laszlo Barabasi and Reka Albert. Emergence of scaling in random networks. *Science*, 286(5439), pp. 509–512, 1999.

[3] Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, 2008.

[4] Marian Boguna, Romualdo Pastor-Satorras, Albert Diaz-Guilera, and Alex Arenas. Models of social networks based on social distance attachment. *Physical Review E*, 70(5):056122, Nov 2004.

[5] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Gorke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. On modularity clustering. *IEEE Transactions on Knowledge and Data Engineering*, 20(2), pp. 172–188, 2008.

[6] Thang N. Bui, Xianghua Deng, and Catherine M. Zrncic. An improved ant-based algorithm for the degree-constrained minimum spanning tree problem. *IEEE Transactions on Evolutionary Computation*, 16(2), pp. 266–278, 2012.

[7] Thang N. Bui, ThanhVu Nguyen, and Joseph R. Rizzo, Jr. Parallel shared memory strategies for ant-based optimization algorithms. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pp. 1–8, New York, NY, USA, 2009.

[8] Aaron Clauset, Mark Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical Review E*, 70(6):06111, Dec 2004.

[9] Anne Condon and Richard M. Karp. Algorithms for graph partitioning on the planted partition model. *Random Structures & Algorithms*, 18(2), pp. 116–140, 2001.

[10] Gabor Csardi and Tamas Nepusz. The igraph software package for complex network research. *InterJournal*, Complex Systems, pp. 1695, 2006.

[11] Leon Danon, Albert Daz-guilera, and Jordi Duch. Comparing community structure identification. *Journal of Statistical Mechanics: Theory and Experiment*, 2005(09):P09008, 2005.

[12] Marco Dorigo. *Optimization, Learning and Natural Algorithms*. Ph.D. thesis, Politecnico di Milano, Italy, 1992.

[13] Marco Dorigo and Luca M. Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1), pp. 53–66, 1997.

[14] Jordi Duch and Alex Arenas. Community detection in complex networks using extremal optimization. *Physical Review E*, 72:027104, 2005.

[15] Tim S. Evans. Clique graphs and overlapping communities. *Journal of Statistical Mechanics: Theory and Experiment*, 2010(12), pp. 12037, 2010.

[16] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(35), pp. 75–174, 2010.

[17] Santo Fortunato and Marc Barthlemy. Resolution limit in community detection. *Proceedings of the National Academy of Sciences*, 104(1), pp. 36–41, 2007.

[18] Michelle Girvan and Mark Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12), pp. 7821–7826, 2002.

[19] Benjamin H. Good, Yves-Alexandre de Montjoye, and Aaron Clauset. Performance of modularity maximization in practical contexts. *Physical Review E*, 81:046106, Apr 2010.

[20] Roger Guimera and Luis A. Nunes Amaral. Functional cartography of complex metabolic networks. *Nature*, 433(7028), pp. 895–900, Feb 2005.

[21] Roger Guimera, Leon Danon, Albert Diaz-Guilera, Francesc Giralt, and Alex Arenas. Self-similar community structure in a network of human interactions. *Physical Review E*, 68:065103, Dec 2003.

[22] Dongxiao He, Jie Liu, Dayou Liu, Di Jin, and Zhengxue Jia. Ant colony optimization for community detection in large-scale complex networks. *Seventh International Conference on Natural Computation (ICNC)*, Volume 2, pp. 1151–1155, Jul 2011.

[23] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9), pp. 1098–1101, Sept 1952.

[24] Di Jin, Dayou Liu, Bo Yang, Carlos Baquero, and Dongxiao He. Ant colony optimization with markov random walk for community detection in graphs. *Proceedings of the 15th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining - Volume Part II*, PAKDD'11, pp. 123–134, Berlin, Heidelberg, Springer-Verlag. 2011.

[25] Donald E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing*. ACM, New York, NY, USA, 1993.

[26] Andrea Lancichinetti and Santo Fortunato. Community detection algorithms: A comparative analysis. *Physical Review E*, 80(5):056117, Nov 2009.

[27] Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. Benchmark graphs for testing community detection algorithms. *Physical Review E*, 78:046110, Oct 2008.

[28] David Lusseau, Karsten Schneider, OliverJ. Boisseau, Patti Haase, Elisabeth Slooten, and SteveM. Dawson. The bottlenose dolphin community of doubtful sound features a large proportion of long-lasting associations. *Behavioral Ecology and Sociobiology*, 54(4), pp. 396–405, 2003.

[29] Claire P. Massen and Jonathan P. K. Doye. Identifying communities within energy landscapes. *Physical Review E*, 71:046101, Apr 2005.

[30] Stanley Milgram. The Small World Problem. *Psychology Today*, 2, pp. 60–67, 1967.

[31] Mark Newman. Fast algorithm for detecting community structure in networks. *Physical Review E*, 69, Sept 2003.

[32] Mark Newman. Finding community structure in networks using the eigenvectors of matrices. *Physical Review E*, 74(3):036104, Sept 2006.

[33] Mark Newman. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences*, 103(23), pp. 8577–8582, 2006.

[34] Mark Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69(2):026113, Feb 2004.

[35] Günce Keziban Orman, Vincent Labatut, and Hocine Cherifi. Comparative evaluation of community detection algorithms: a topological approach. *Journal of Statistical Mechanics: Theory and Experiment*, 2012(08):P08001, 2012.

[36] Clara Pizzuti. GA-Net: A genetic algorithm for community detection in social networks. *Parallel Problem Solving from Nature PPSN X*, volume 5199 of *Lecture Notes in Computer Science*, pp. 1081–1090. Springer Berlin Heidelberg, 2008.

[37] Clara Pizzuti. Boosting the detection of modular community structure with genetic algorithms and local search. *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pp. 226–231, New York, NY, USA, 2012. ACM.

[38] Clara Pizzuti. A multiobjective genetic algorithm to find communities in complex networks. *IEEE Transactions on Evolutionary Computation*, 16(3), pp. 418–430, 2012.

[39] Pascal Pons and Matthieu Latapy. Computing communities in large networks using random walks. *Computer and Information Sciences - ISCIS 2005*, Volume 3733 of *Lecture Notes in Computer Science*, pp. 284–293. Springer Berlin Heidelberg, 2005.

[40] Josep M. Pujol, Javier Béjar, and Jordi Delgado. Clustering algorithm for determining community structure in large networks. *Physical Review E*, 74:016107, Jul 2006.

[41] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2011.

[42] Filippo Radicchi, Claudio Castellano, Federico Cecconi, Vittorio Loreto, and Domenico Parisi. Defining and identifying communities in networks. *Proceedings of the National Academy of Sciences of the United States of America*, 101(9), pp. 2658–2663, 2004.

[43] Usha Nandini Raghavan, Reka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3):036106, Sept 2007.

[44] Jörg Reichardt and Stefan Bornholdt. Detecting fuzzy community structures in complex networks with a potts model. *Physical Review Letters*, 93:218701, Nov 2004.

[45] Jorma Rissanen. Modeling by shortest data description. *Automatica*, 14(5), pp. 465–471, 1978.

[46] Peter Ronhovde and Zohar Nussinov. Local resolution-limit-free potts model for community detection. *Physical Review E*, 81:046114, Apr 2010.

[47] Martin Rosvall and Carl T. Bergstrom. Maps of random walks on complex networks reveal community structure. *Proceedings of the National Academy of Sciences*, 105(4), pp. 1118–1123, 2008.

[48] S. Sadi, S. Oguducu, and A.S. Uyar. An efficient community detection method using parallel clique-finding ants. *IEEE Congress on Evolutionary Computation (CEC)*, pp. 1–7, 2010.

[49] Thomas Sttzle and Holger H. Hoos. Maxmin ant system. *Future Generation Computer Systems*, 16(8), pp. 889–914, 2000.

[50] Stijn van Dongen. *Graph Clustering by Flow Simulation*. Ph.D. thesis, University of Utrecht, 2000.

[51] Stanley Wasserman and Katherine Faust. *Social Network Analysis: Methods and Applications*. Cambridge University Press, 1994.

[52] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, (393), pp. 440–442, 1998.

[53] Wayne W. Zachary. An information flow model for conflict and fission in small groups. *Journal of Anthropological Research*, 33(4), pp. 452–473, 1977.

[54] Haijun Zhou. Network landscape from a brownian particle's perspective. *Physical Review E*, 67:041908, Apr 2003.