UPPSALA
UNIVERSITET

Stochastic Block Model Methods for Bipartite Graphs

Deepthi Hulithala Venkataramana, Naeim Rashidfarokhi

Supervisor: Davide Vega D'Aurelio

Project in Computational Science: Report

January 2021

PROJECT REPORT

Institutionen för informationsteknologi

Abstract

Many of the phenomena in nature and society can be represented as networks. Networks can be complicated and to understand them, it is a good idea to reduce them into structural blocks. One of the techniques to deduce structural blocks is Stochastic Block Modelling (SBM). As many of the systems, say, drug-target, predator-prey, plant-pollinators can be represented in bipartite networks, we consider deducing structures in bipartite networks.

In this project, we have implemented a standard stochastic block model and a degree-corrected stochastic block model using brute force search algorithm and a heuristic algorithm similar to Kernighan-Lin algorithm. Minimum description length principle is implemented to automate the number of communities to be detected in a network.

We have verified our implementation on three networks and experimented the degree-corrected stochastic block model on seven real world networks.

Acknowledgement

We would like to thank Dr. Davide Vega D'Aurelio, the project supervisor for the constant support, guidance and constructive feedback throughout every stage of the project. The completion of this project would not have been possible without his support.

Deepthi Hulithala Venkataramana
Naeim Rashidfarokhi

Contents

1 Introduction

Many systems or phenomena in nature and society can be represented as a network. A network or graph consists of a known number of nodes with connections represented as edges (see Section 2.1) among the nodes. Networks can be complicated and quite incoherent. So, to understand networks, it is usually a good idea to reduce them into structural blocks. These structural blocks can represent people, animals, organisations or objects that are connected to each other in a similar way. One of the techniques to reduce a network into structural blocks is Stochastic Block Modelling (SBM).

In general, the idea of finding groups or communities of similar nodes in networks is block modeling. Block modeling technique tries to identify groups among nodes in a network with respect to some equivalency of nodes. This quality or similarity of nodes is based on the relations between nodes, i.e. the way nodes are connected to each other by edges. It means that even the absence of an edge between two sample nodes conveys information about a network. Since we are dealing with probabilities of existence or non-existence of all possible edges, we need a generative model. SBM is a generative model (see Section 2.2), which tries to find a probabilistic distribution for edges to explain the nature of a graph. Figure 1 shows this probabilistic distribution in the form of a matrix called stochastic block matrix.

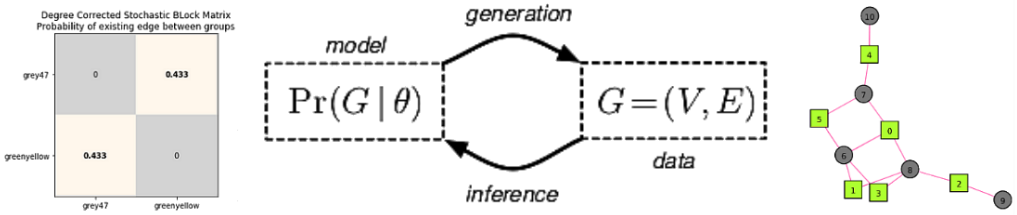


Figure 1: SBM concept: Having a probability matrix, SBM generates an instance of a network corresponding to the matrix (left to right is generation), having a network, SBM can find a probability matrix corresponding to that of the network (right to left is inference).

As seen above, the act of revealing the spirit of a network (real or synthetic) is to find a hidden probability distribution of edges among nodes and SBM uses likelihood estimation (see Section 2.3) to do it. Likelihood is a partition scoring function. To explain that, imagine we divide the nodes of a network into a predefined number of communities. Now, SBM uses an equation to calculate how likely is it to have that community formation. By comparing all likelihood values for different community formations, the highest score yields the most probable community formation. At this step, SBM saves probability values of edges as the result in a stochastic block matrix.

The goal of this study is to apply SBM on bipartite networks which are witnessed frequently in nature and human societies, among others in the field of telecommunications, social science, network biology, medicine, etc. As seen in Figure 2, a bipartite network is a graph with two sets of vertices which are only connected to each other, but not within themselves. For further information see [1].

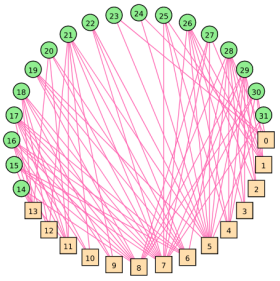


Figure 2: Bipartite network with no edge between similar nodes, pink color to show inter edges and black color to show intra edges.

Some examples of bipartite networks are predator-prey, directors and firms, students and teachers, plants and pollinators, genes and genomes, actors and movies, scientists and papers, drug and target networks, etc, see [8].

2 Concepts and Definitions

This section consists of some required statistical explanations and is provided, mainly from online sources, for the ease of the reader to continue with the project.

2.1 Edge

A graph is composed of two elements, vertices and edges. Each edge has two vertices as endpoints, to which it is attached. Edges may be directed or undirected; undirected edges are also called lines and directed edges are also called arcs or arrows. An edge that connects vertices x and y is sometimes written xy, see [15].

2.2 Generative model

A generative model learns the general rules that govern the appearance of observations in a dataset, in a way to be able to produce a new dataset that has never existed. A generative model must also be probabilistic rather than deterministic. If our model is merely a fixed calculation, such as taking the average value among values in a dataset, it is not generative because the model produces the same output every time. The model must include a stochastic (random) element that influences the individual samples generated by the model. In other words, there is some unknown probabilistic distribution that explains why some observations (here, edges between vertices) are likely to be found in a dataset and why others do not exist. The task to build a generative model is to find this distribution. Later by sampling from this distribution, generation of new observations is possible, see [6].

2.3 Likelihood & Probability

Probability corresponds to finding the chance of something given a sample distribution of the data, while on the other hand, Likelihood refers to finding the best distribution of the data given a particular value of some feature or some situation in the data, see [7].
An example provided at [13] explains the concept by a toss coin. If we know the coin is fair ($p = 0.5$) a typical probability question is: What is the probability of getting two heads in a row?
The answer is: $P(HH) = P(H) \times P(H) = 0.5 \times 0.5 = 0.25$.
A typical statistical question is: Is the coin fair? To answer this we need to ask: To what extent does our sample support our hypothesis that $P(H) = P(T) = 0.5$?
The first point to note is that the direction of the question has reversed. In probability we start with an assumed parameter (P(head)) and estimate the probability of a given sample (two heads in a row).
In statistics we start with the observation (two heads in a row) and make inference about our parameter ($p = P(H) = 1 - P(T) = 1 - q$). So, Probability quantifies expectation of outcome, likelihood quantifies trust in model.
So, the problems connected to these two concepts are inverse to each other. In probability theory we consider some underlying process which has some randomness or uncertainty modeled by random variables, and we figure out what happens. In statistics we observe something that has happened, and try to figure out what underlying process would explain those observations, see [14].

2.4 Probability distribution

The probability distribution for a random variable describes how the probabilities are distributed over the values of the random variable. For a discrete random variable, x, the probability distribution is

defined by a probability mass function, denoted by $f(x)$. This function provides the probability for each value of the random variable. In the development of the probability function for a discrete random variable, two conditions must be satisfied:

- f(x) must be non-negative for each value of the random variable,
- the sum of the probabilities for each value of the random variable must be equal to one, see [3].

2.5 Random variable

A random variable is a numerical description of the outcome of a statistical experiment. A random variable that may assume only a finite number or an infinite sequence of values is said to be discrete; one that may assume any value in some interval on the real number line is said to be continuous. For instance, a random variable representing the number of automobiles sold at a particular dealership on one day would be discrete, while a random variable representing the weight of a person in kilograms (or pounds) would be continuous, see [3].

2.6 Statistical inference

Statistical inference consists uses statistics to draw conclusions about some unknown aspect of a population based on a random sample from that population, see [5].
The practice of statistics falls broadly into two categories (1) descriptive or (2) inferential. When we are just describing or exploring the observed sample data, we are doing descriptive statistics. However, we are often also interested in understanding something that is unobserved in the wider population, this could be the average blood pressure in a population of pregnant women for example, or the true effect of a drug on pregnancy rate, or whether a new treatment performs better or worse than the standard treatment. In these situations we have to recognise that almost always we observe only one sample or do one experiment. If we take another sample or conduct another experiment, then the result would almost certainly vary. This means that there is uncertainty in our result. If we take another sample or conduct another experiment and base our conclusion solely on the observed sample data, we may even end up drawing a different conclusion!.
The purpose of statistical inference is to estimate this sample to sample variation or uncertainty. Understanding how much our results may differ if we did the study again, or how uncertain our findings are, allows us to take this uncertainty into account when drawing conclusions. It allows us to provide a plausible range of values for the true value of something in the population, such as the mean, or size of an effect, and it allows us to make statements about whether our study provides evidence to reject a hypothesis, see [12].

2.7 A stochastic phenomenon

Stochastic means being or having a random variable.The essence of many fields of science have stochastic behaviour such as: biology, chemistry, ecology, neuro-science, physics, as well as technology and engineering fields such as image processing, signal processing, information theory, computer science, cryptography and telecommunications.

2.8 Stochastic block modeling

The stochastic block model is a generative model for random graphs. This model tends to produce graphs containing communities, subsets characterized by being connected with one another with particular edge densities. For example, edges may be more common within communities than between communities. The stochastic block model is important in statistics, machine learning, and network science, where it serves as a useful benchmark for the task of recovering community structure in a graph data, see [18].

2.9 Modularity

Modularity is one of the measure of the structure of networks or graphs. It was designed to measure the strength of division of a network into communities.

3 Methods (used in the project)

3.1 Standard Stochastic Block Model

A null model consists of a network, which is used for comparison, to verify whether the network in question displays some non-trivial features (properties that would not be expected on the basis of chance alone or as a consequence of the constraints), such as community structure in networks.

The null model that we consider in this project is a random network. If the considered network is not random, then there are communities or structure hidden in the network. The stochastic block model method compares our network with our null model to discover the different properties of the network which leads to detecting structures in the network.

Following, we describe the mathematical process to arrive at the formulation of likelihood estimation using probability based on the paper [9] and lecture notes [4].

The Stochastic Block Model is defined by a scalar value and two data structures:

- k is the number of communities,
- $\vec{g}$ is a $n \times 1$ vector which represents the group index of each vertex where n is the number of nodes in the network,
- ω is a $k \times k$ matrix with entries, ω_{rs} in [0, 1] which represents the connectivity probabilities between a vertex of type r and a vertex of type s.

Given a choice of k and an observed network G, we can use the SBM to infer the latent community assignments z and the matrix ω. There are several ways of doing this, the simplest of which is to use maximum likelihood.
We can express the probability $P(G|\omega, g)$ of network G given the parameters as:

$$P(G|\omega, g) = \prod_{i<j} \frac{(\omega_{g_i g_j})^{A_{ij}}}{A_{ij}!}\ exp(-\omega_{g_i g_j}) \times \prod_i \frac{(\omega_{g_i g_i})^{\frac{A_{ii}}{2}}}{(\frac{A_{ii}}{2})!}\ exp(-\frac{1}{2}\omega_{g_i g_i}). \tag{1}$$

The terms in the Eqn. (1) can be simplified as follows:

$$\prod_{i<j}(\omega_{g_i g_j})^{A_{ij}} \prod_i (\omega_{g_i g_i})^{\frac{A_{ii}}{2}} = \prod_{rs} \omega_{rs}^{\frac{m_{rs}}{2}},$$

$$\prod_{i<j} exp(-\omega_{g_i g_j}) \prod_i exp(-\frac{1}{2}\omega_{g_i g_i}) = \prod_{rs} exp(-\frac{1}{2} n_r n_s \omega_{rs}),$$

$$m_{rs} = \sum_{ij} A_{ij} \delta_{g_i, r} \delta_{g_j, s},$$

where,

$$\delta(a, b) = \begin{cases} 1, & \text{if } a = b, \\ 0, & \text{otherwise.} \end{cases}$$

Given that $A_{ij} = A_{ji}$ and $\omega_{rs} = \omega_{sr}$, where r and s are the communities, we can rewrite Eqn. (1) as

$$P(G|\omega, g) = \frac{1}{\prod_{i<j} A_{ij}! \prod_i 2^{\frac{A_{ii}}{2}} (\frac{A_{ii}}{2})!} \times \prod_{rs} \omega_{rs}^{\frac{m_{rs}}{2}}\ exp(-\frac{1}{2} n_r n_s \omega_{rs}), \tag{2}$$

where n_r and n_s are the number of vertices in group r and s respectively.
Our goal is to maximize the likelihood with respect to the parameters ω_{rs} and the community assignments

of the vertices, g. It is simpler to maximize the logarithm of the likelihood instead of the likelihood as it is computationally less expensive. Neglecting constants and terms independent of the parameters ω_{rs}, n_r, n_s, m_{rs} and applying logarithm on Eqn. (2), we get

$$logP(G|\omega, g) = \sum_{rs}(m_{rs}log\omega_{rs} - n_r n_s \omega_{rs}), \tag{3}$$

and we maximize the expression in two stages:

- with respect to the model parameters ω_{rs},
- with respect to the group assignments g.

The maximum likelihood values $\hat{\omega}_{rs}$ of the model parameters are found by simple differentiation of Eqn. (3) with respect to ω_{rs} as

$$\hat{\omega}_{rs} = \frac{m_{rs}}{n_r n_s}. \tag{4}$$

Substituting equation (4) in (3), we get,

$$logP(G|g) = \sum_{rs} m_{rs} log \frac{m_{rs}}{n_r n_s}, \tag{5}$$

Dividing by the total number of edges ($2m$) and vertices (n^2), we get

$$L(G|g) = \sum_{rs} \frac{m_{rs}}{2m} log \frac{\frac{m_{rs}}{2m}}{\frac{n_r n_s}{n^2}}. \tag{6}$$

Let

$$p_K(r,s) = \frac{m_{rs}}{2m},$$

$$p_1(r,s) = \frac{n_r n_s}{n^2},$$

$$L(G|g) = \sum_{rs} p_K(r,s) log \frac{p_K(r,s)}{p_1(r,s)}. \tag{7}$$

which represents the Kullback-Leibler divergence between the probability distributions p_K and p_1.
An approach where one constructs an objective function that measures the difference between an observed quantity and the expected value of the same quantity under an appropriate null model, is a well known strategy in network community detection. One such objective function is the modularity function. The null model chosen is a random network.
The considered objective function, Q can be defined as

$$Q(g) = \sum_{r=1}^{K} [p_K(r,r) - p_{degree}(r,r)], \tag{8}$$

where, $p_{degree}(r,s) = \frac{k_r}{2m}\frac{k_s}{2m}$, and $k_r = \sum_s m_{rs} = \sum_i k_i \delta_{g_i,r}$.

When a network consists of heterogeneous degree distribution i.e. if there is a huge variability between the degrees of the nodes of the network, Standard SBM groups the nodes of the network based on the degrees of the nodes. Two nodes with high degree are more likely to be connected compared to two nodes with lower degree. Hence, the high-degree nodes are in a small community together, with a larger probability of connecting to other larger communities. The likelihood function is maximised if the probability values in the block matrix is close to zero or one. Standard SBM prefers this kind of grouping. Unfortunately, these communities are not similar to the realistic situations. This problem can be solved by defining a stochastic block-model that directly incorporates arbitrary heterogeneous degree distributions.

3.2 Degree Corrected Stochastic Block Model

As the standard SBM does not detect the true communities when the network consists of heterogeneous degree distribution, degree-corrected stochastic block-model is considered where the arbitrary heterogeneous degree distributions is directly incorporated in the model. In the degree-corrected block-model, the probability distribution over undirected network depends not only on the parameters introduced previously but also on a new set of parameters θ_i controlling the expected degrees of vertices i. Considering there are k number of communities, ω_{rs} is a $k \times k$ symmetric matrix of parameters controlling edges between communities r and s, and g_i is the group assignment of vertex i. Let the expected value of the adjacency matrix element A_{ij} be $\theta_i\theta_j\omega_{g_ig_j}$. The probability of the graph G with the given parameters and community assignment can be written as

$$P(G|\theta,\omega,g) = \prod_{i<j} \frac{(\theta_i\theta_j\omega_{g_ig_j})^{A_{ij}}}{A_{ij}!}\ exp(-\theta_i\theta_j\omega_{g_ig_j}) \times \prod_i \frac{(\theta_i^2\omega_{g_ig_i})^{\frac{A_{ii}}{2}}}{(\frac{A_{ii}}{2})!}\ exp(-\frac{1}{2}\theta_i^2\omega_{g_ig_i}). \tag{9}$$

The θ parameters are arbitrary to within a multiplicative constant which is absorbed into the ω parameters. Their normalization can be fixed by imposing the constraint

$$\sum_i \theta_i\delta_{g_i,r} = 1, \tag{10}$$

for all communities r, which makes θ_i equal to the probability that an edge connected to the community to which i belongs lands on i itself. With this constraint, the probability $P(G|\theta,\omega,g)$ can be simplified to the more convenient form

$$P(G|\theta,\omega,g) = \frac{1}{\prod_{i<j} A_{ij}! \prod_i 2^{\frac{A_{ii}}{2}} (\frac{A_{ii}}{2})!} \times \prod_i \theta_i^{k_i} \times \prod_{rs} \omega_{rs}^{\frac{m_{rs}}{2}}\ exp(-\frac{1}{2}\omega_{rs}), \tag{11}$$

where, k_i is the degree of vertex i and $m_{rs} = \sum_{ij} A_{ij}\delta_{g_i,r}\delta_{g_j,s}$.
It is computationally more convenient to maximize the logarithm of the probability than maximizing the probability, thus,

$$logP(G|\theta,\omega,g) = 2\sum_i k_i log\theta_i + \sum_{rs}(m_{rs}log\omega_{rs} - \omega_{rs}). \tag{12}$$

The maximum-likelihood values of the parameters θ_i and ω_{rs} are given by

$$\begin{aligned} \hat{\theta}_i &= \frac{k_i}{k_{g_i}}, \\ \hat{\omega}_{rs} &= m_{rs}, \end{aligned} \tag{13}$$

where k_r is the sum of degrees in community r. The maximum-likelihood parameter estimate has the property of persevering the expected numbers of edges between communities and the expected degree sequence of the network. Then the expected number of edges between communities r and s is

$$\begin{aligned} \sum_{ij} < A_{ij} > &= \sum_j \hat{\theta}_i\hat{\theta}_j\hat{\omega}_{g_ig_j} = \frac{k_i}{k_{g_i}} \sum_j \frac{k_i}{k_{g_i}} m_{g_ig_j}, \\ &= \frac{k_i}{k_{g_i}} \sum_j \sum_r \frac{k_i}{k_{g_i}} m_{g_i,r}\delta_{g_j,r}, \\ &= \frac{k_i}{k_{g_i}} \sum_r m_{g_i,r} = k_i. \end{aligned} \tag{14}$$

Substituting Eqn. (13) in Eqn. (12), we get the maximum of $logP(G|\theta,\omega,g)$ for the degree-corrected block-model.

$$logP(G|\theta,\omega,g) = 2\sum_i k_i log\frac{k_i}{k_{g_i}} + \sum_{rs} m_{rs}log\ m_{rs} - 2m, \tag{15}$$

where, m is the total number of edges in the network.
The first term in the right hand side (R.H.S.) of Eqn. (15) can be simplified as

$$\begin{aligned} 2\sum_i k_i log\frac{k_i}{k_{g_i}} &= 2\sum_i k_i logk_i - 2\sum_i\sum_r k_i\delta_{g_i,r}logk_r, \\ &= 2\sum_i k_i logk_i - \sum_r k_r logk_r - \sum_s k_s logk_s, \\ &= 2\sum_i k_i logk_i - \sum_{rs} m_{rs} logk_r k_s. \end{aligned} \tag{16}$$

Substituting Eqn. (16) in Eqn. (15) and dropping overall constants, we get an unnormalized log-likelihood function as

$$L(G|g) = \sum_{rs} m_{rs} log\frac{m_{rs}}{k_r k_s}. \tag{17}$$

We can rewrite the log-likelihood by adding and multiplying by a constant factor $2m$ as

$$L(G|g) = \sum_{rs} \frac{m_{rs}}{2m} log\frac{\frac{m_{rs}}{2m}}{\frac{k_r}{2m}\frac{k_s}{2m}}, \tag{18}$$

which is the Kullback-Leibler divergence between p_K and p_{degree}.

3.3 Algorithms (used in the project)

In addition to the implementation of log-likelihood equations in standard and degree corrected stochastic block models, searching through different combinations is also needed. For this reason, two different algorithms are implemented namely, brute force search and a local heuristic search. In the end, to find the optimum number of communities for any network, an algorithm based on minimum description length is implemented.

3.3.1 Brute Force Search

Brute force search is based on exhaustive testing of all possibilities. This algorithm is a very general problem solving technique to systematically check if candidates can satisfy a problem's statement. The algorithm is simple to implement and guarantees to find a solution if it exists, see [17].
In the beginning of this project study, when the focus was on understanding the concept of SBM, we implemented Brute force search algorithm for simplicity with two and then n number of communities. Later we switched to a local heuristic search algorithm proposed by Karrer and Newman.

3.3.2 Local heuristic search

As the graph size and specially the number of communities increase, the application of Brute Force Search for community detection moves from inefficient state to impossible. A simple example is a network of 32 nodes with 5 communities where Brute Force Search needs to go through more than two hundred thousands of calculations:

$$\frac{32!}{5! \times (32-5)!} = 201376.$$

In this work, a heuristic algorithm, proposed by Karrer and Newman, see [2], is implemented to overcome this problem. In the algorithm 1 below, *pass* statements can be ignored in coding as they are added for more readability.

Algorithm 1: Local heuristic search

```
for itr in n iterations: do
    divide the network into K random communities;
    set all values in movement-control-dictionary to False;
    for vertex in movement-control-dictionary: do
        if vertex not moved: then
            for key,value in dict.items: do
                if key is alone in its group: then
                    pass;
                else
                    calculate new-mle;
                    if new-mle > old-mle: then
                        replace and hold network partitioning corresponded to new-mle;
                        replace and hold new-mle with old-mle;
                        replace and hold new-vertex-target-group with old-vertex-target-group;
                    else
                        pass;
                    end
                end
            end
            set founded vertex as moved from final new-vertex-target-group value;
            replace original partitioning with the one corresponded to new-mle for another
             investigation;
            save new-mle result for final comparison;
        else
            pass;
        end
    end
    find accumulative MLE peak as iteration result and save it;
end
Among all iterations find the maximum peak as final result;
```

Briefly, in this algorithm, we perform the following steps:

Step.1 We divide the network into some initial set of K communities at random.

Step.2 Then we repeatedly move a vertex from one group to another, selecting at each step the move that will most increase the objective function—or least decrease it if no increase is possible—subject to the restriction that each vertex may be moved only once.

Step.3 After the completion of Step.2, when all vertices have been moved, we inspect the states through which the system passed from start to end of the procedure, select the one with the highest objective score, and use this state as the starting point for a new iteration of the same procedure.

Step.4 During the Step.3, if a complete iteration passes without any increase in the objective function, the algorithm ends.

Step.5 As with many deterministic algorithms, we have found it helpful to run the calculation with several different random initial conditions and take the best result over all runs.

3.3.3 Choosing the number of communities k

In the inference method, to obtain the likely block-model, it is assumed that the number of communities is known in advance. Unfortunately, in general, this quantity is unknown and we infer it from the data. Obtaining the block-model for different number of communities in a network on trial and error basis can be a very tedious and time-consuming task. Hence, to infer the number of communities from the data

in an efficient way, we use the *minimum description length* (MDL) principle, see paper [11]. The MDL principle predicts that the best choice of model which fits a given data is the one which most compresses it, i.e. the best choice minimizes the total amount of information required to describe it.
The standard SBM ensemble consists of networks with N nodes, each belonging to one of the k communities, and the number of edges between nodes of communities r and s is given by the matrix e_{rs}. The degree-corrected SBM further imposes that each node i has a degree given by θ_i, where the set θ_i is an additional parameter set of the model. The network is characterised by its entropy S, which is the measure of the number of possible arrangements the nodes in a network can have. The entrophy, S is equal to $ln\ \Omega$ where, Ω is the total number of network realizations. The likelihood, L is equal to $1/\Omega$. Maximizing $ln\ L$ is equivalent to minimizing the entropy S. Entropy minimization is well-defined, but only as long as the total number of communities, k is known in advance. Otherwise, the optimal value of the entropy, S becomes a strictly decreasing function of k and minimizing the entropy will result in the trivial case where number of communities, k is equal to the number of nodes, N in the network. To minimize the entropy without over-fitting is to consider the total amount of information necessary to describe the data, which includes the entropy as well as the information necessary to describe the model. This quantity is known as the *description length* and for the SBM ensemble, it is given by

$$\Sigma_{s/d} = S_{s/d} + L_{s/d}, \tag{19}$$

where the subscript s/d represents standard SBM/degree-corrected SBM, Σ is the *description length*, L is the information necessary to describe the model and can be defined as,

$$L_s \cong Eh\left(\frac{k(k+1)}{2E}\right) + Nln\ k, \tag{20}$$

where, E and N are the total number of edges and nodes in the network respectively, k is the number of communities and $h(x) = (1+x)ln(1+x)$. The description length of the degree-corrected variant can be defined as

$$L_d = L_s - N\sum_i p_i\ ln\ p_i, \tag{21}$$

where p_i is the fraction of nodes with degree i.
Considering the number of communities, k as 1 is a trivial case. We can define $\Sigma_k \equiv \Sigma_{s/d} - \Sigma_{s/d|k=1}$ as

$$\Sigma_k = Eh\left(\frac{k(k+1)}{2E}\right) + Nln\ k - EI_{s/d}, \tag{22}$$

where,

$$\begin{aligned} I_s &= \sum_{rs} m_{rs} ln\left(\frac{m_{rs}}{w_r w_s}\right), \\ I_d &= \sum_{rs} m_{rs} ln\left(\frac{m_{rs}}{m_r m_s}\right), \end{aligned} \tag{23}$$

where, m_{rs} is equal to $e_{rs}/2E$ and w_r is equal to n_r/N. The MDL principle imposes a natural constraint on the maximum value of k, k_{max} which can be detected given a network size and density. This is obtained by minimizing Σ_k over all possible community structures with a given k, which is achieved by replacing $I_{s/d}$ in Eqn. (22) by its maximum value $ln\ k$,

$$\Sigma_k^{'} = Eh\left(\frac{k(k+1)}{2E}\right) - (E-N)ln\ k. \tag{24}$$

Eqn. (24) is a strictly convex function on k which means there is a global minimum $\Sigma_k^{'}|_{k=k_{max}}$ given uniquely by total number of nodes, N and edges, E.

4 Implementation

This project is done in Python programming for faster development and easier debugging process.

4.1 Brute force search with two communities

At the early stage of the project, the theoretical concepts are implemented to solve a small network of six nodes with two communities. The idea is to implement brute-force search algorithm in combination with standard stochastic block modeling. The implemented code for brute-force search is given in Appendix (7.2.1).

4.2 Brute force search with any number of communities

The next step is to expand the code into any number of communities while brute-force is still the internal search engine to go through all combinations. At this stage, the code is a proper tool for inference process in SBM if computationally possible. The corresponding code is given in Appendix (7.2.2).

4.3 Heuristic search with any number of communities

The final step with respect to SBM implementation is to make the code work for larger networks. Since Brute-force search is a slow and computationally heavy (memory intense) method, the heuristic algorithm introduced in Section (3.3.2), is used instead.

The implementation to make a new combination, after each vertex movement from an original community to a target community, is based on identifying changes in the old combination and updating only that part of the graph. It means that the whole community information for the old combination is preserved and only the changes are updated to form information corresponding to a new combination. Compared with re-calculating all inter and intra edges when one particular vertex shifts from community m to community n, this optimization saves a considerable amount of time. For more details on heuristic search implementation, see Appendix (7.2.3).

4.4 Minimum Description Length (MDL) principle

To automate the number of communities to be detected in the network, we use the MDL principle. See Appendix (7.2.3).

5 Results

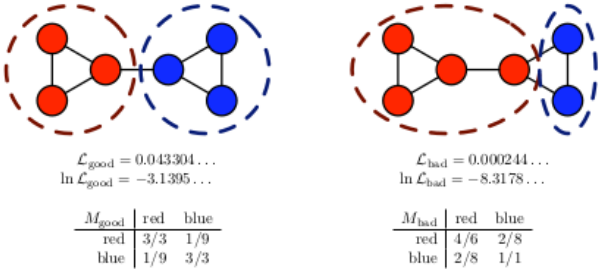
The performance of the code, to implement the theoretical concepts in this report, is tested and presented in this section. The two objectives are accuracy and performance. Section (5.1) explains how the accuracy is achieved while Section (5.2) shows the performance of the code on some selected real networks.

5.1 Verification

Before dealing with any real experiment and possible interpretation of results, it is required to verify the accuracy of the code. For this purpose, three networks are used.

5.1.1 Six-node graph

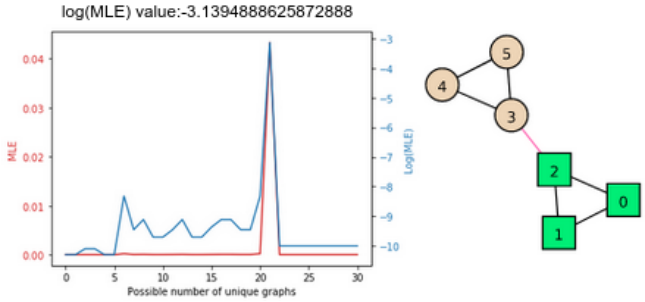
A small network of six nodes, inspired from the lecture notes by Professor Aaron Clauset [4], is selected to verify the code. Due to relatively small size of the graph, which is appropriate for debugging, and the maximum likelihood values provided in the lecture notes, exact comparison is possible.



M_{good}	red	blue
red	3/3	1/9
blue	1/9	3/3

M_{bad}	red	blue
red	4/6	2/8
blue	2/8	1/1

(a) Likelihood estimation for two combinations,3-3 and 4-2 divided into red and blue communities, provided in Lecture notes, see [4].

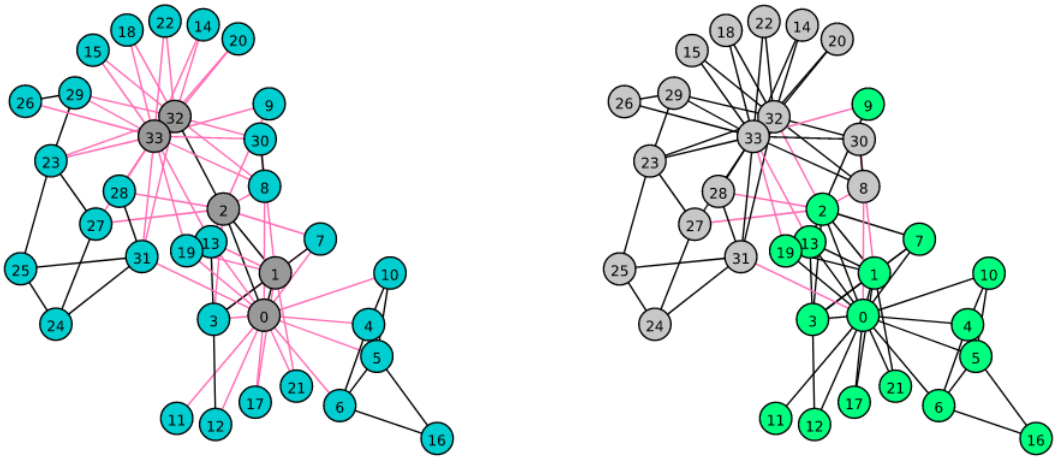


(b) Likelihood estimation for 31 unique combinations and finding the highest score for the symmetric case.

Figure 3: Compassion of likelihood estimations between code results and provided results in the literature.

5.1.2 Karate-club

Zachary's karate club network is one of the very famous network. It consists of 34 nodes which represent the 34 members of a karate club. The edges represent the pairs of members of the club who interact outside the club. A conflict arises between the administrator and instructor, which leads to the split of the club into two groups. Half of the members form a new club with the instructor and the members of the other half find a new instructor or give up the karate-club.
Here, we can witness the communities detected by standard SBM and degree-corrected SBM, where in the standard SBM, high degree nodes are grouped together, whereas the communities detection using degree-corrected SBM is almost similar to the realistic situation.



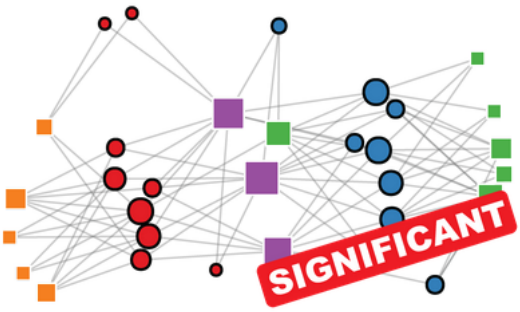
(a) Communities detected using standard SBM.

(b) Communities detected using degree-corrected SBM.

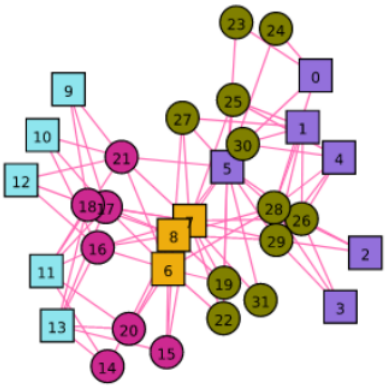
Figure 4: Community detection in Zachary's karate club network with different methods.

5.1.3 Github-Graph

The third case for verification is a bipartite network which is provided in Github [16] and is composed of two groups in one type and three groups in another type. According to bipartite SBM contributors at Github, the same principle is used to find the optimum number of communities. Also community detection is done by SBM for bipartite networks in the Github code but with Markov chain Monte Carlo (MCMC) sampler or the Kernighan-Lin algorithms as the core optimization engines. Despite some similarities, these algorithms are different than the approach taken in this project explained in Section (3.3.2). The comparison of the results are shown in Figure 5.



(a) Bipartite Stochastic Block Model (biSBM) using the MCMC sampler or the Kernighan-Lin algorithm. [16].



(b) Degree Corrected Stochastic Block Model using a local heuristic algorithm.

Figure 5: A bipartite network of 32 nodes with five communities and two types.

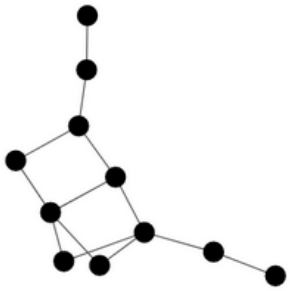
5.2 Experiments

We test the performance of the code to infer and to detect communities on real networks on some graphs from Netzschleuder website, see [10]. The selection criteria are based on graph type, mostly bipartite, and size of graph.
Each experiment starts with a network picture, follows by the results from the code in the following order:

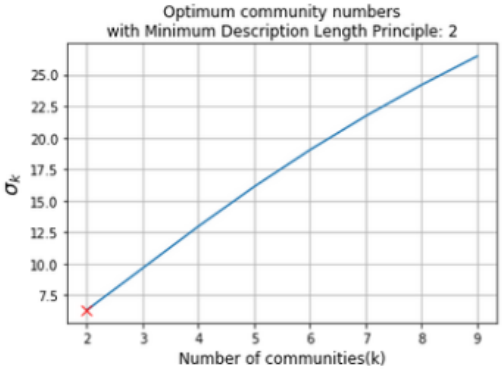
- best number of communities detected by Minimum Description Length principle,
- Adjacency matrix for the network,
- Best Maximum Likelihood estimation with different number of runs (starting from different initial random grouping),
- a representation of communities detected in different colors (for bipartite graphs, shapes represent types while for unipartite graphs shapes are only used to help colors for a better representation of a detected group),
- Stochastic Block Matrix.

5.2.1 South African companies

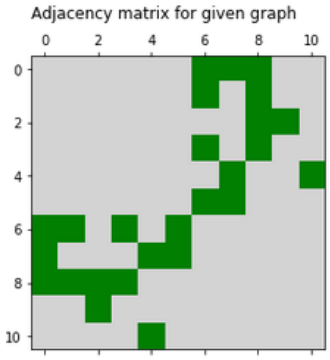
This bipartite network represents the affiliations between a small group of important individuals and five major companies in South African finance, around 1919.



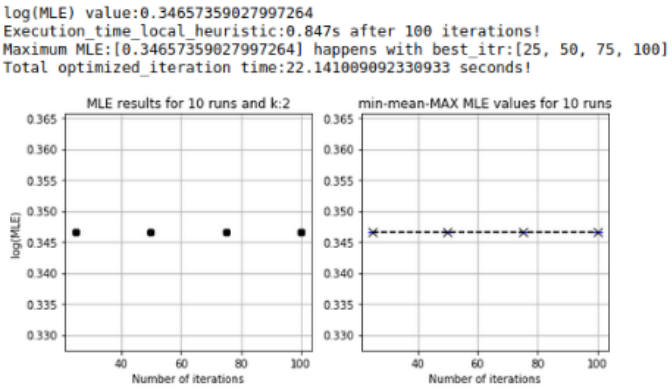
(a) Original network.



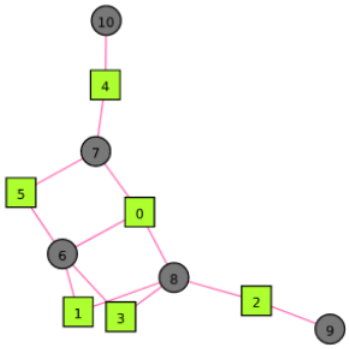
(b) Communities with MDL.



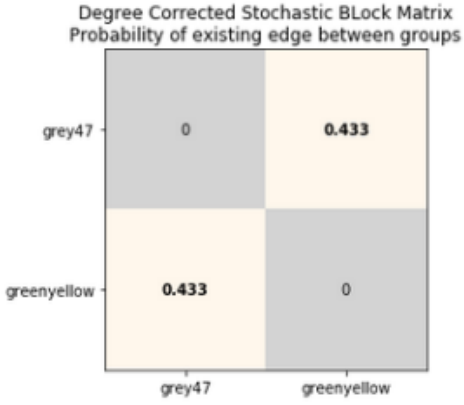
(c) Adjacency matrix.



(d) Best MLE with different runs.



(e) Detected communities.



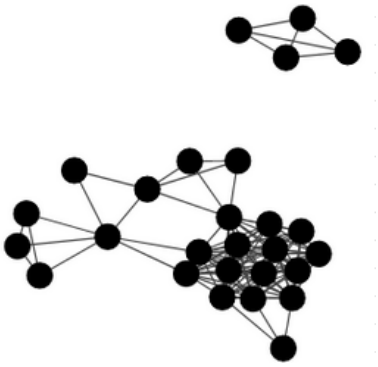
(f) Community information.

Figure 6: South African companies network from original graph to corresponding community information.

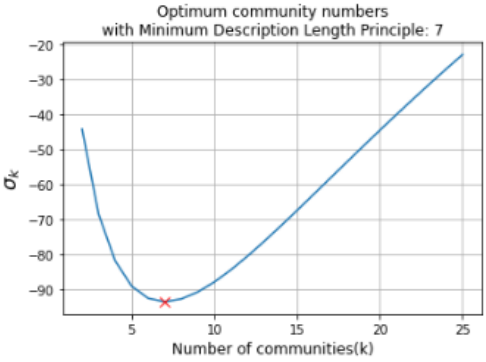
Despite the simplicity of the network, the result shows that there are no connections among individuals which is obvious prior to community detection. However, the probability of being connected to a major company by an important individual is deduced as 40%.

5.2.2 Zebras (2002)

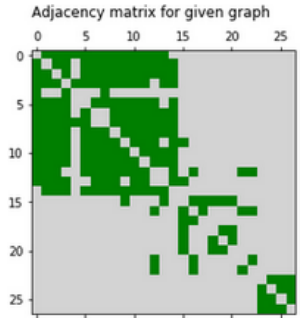
This unipartite graph represents social interactions among a group of wild Grevy's zebras observed in Mpala Ranch in Kenya in 2002.



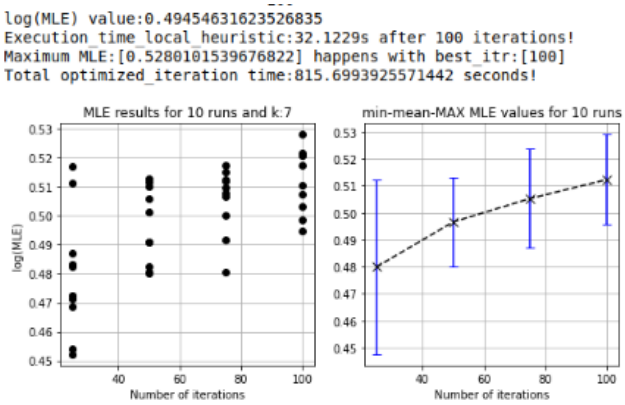
(a) Original network.



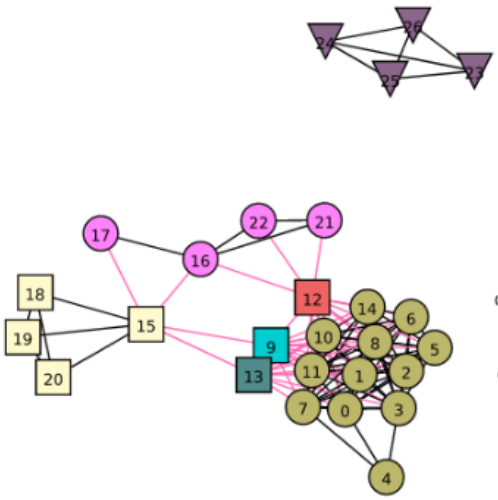
(b) Communities with MDL.



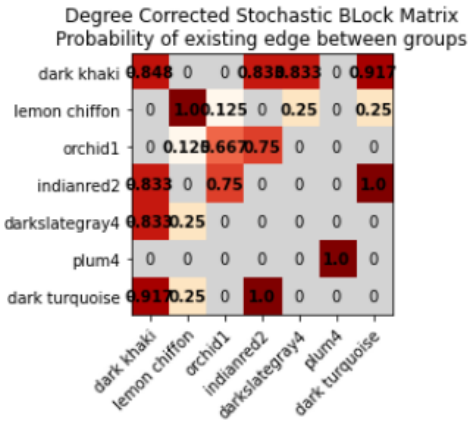
(c) Adjacency matrix.



(d) Best MLE with different runs.



(e) Detected communities.



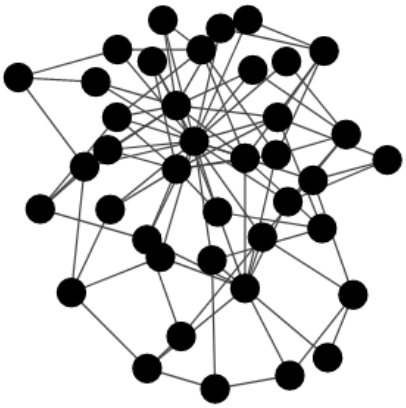
(f) Community information.

Figure 7: Social interactions among a group of wild Grevy's zebras network from original graph to corresponding community information.

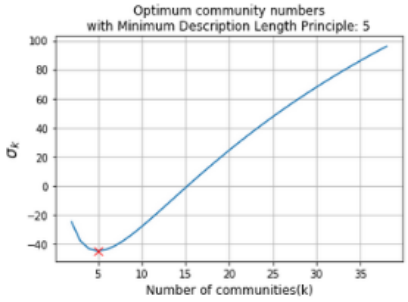
The result shows that despite many connections with other zebras, there are three solo zebras in the herd, zebras with number 9, 12 and 13.

5.2.3 CEO Club Memberships

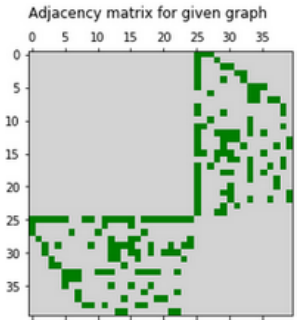
This bipartite graph represents the memberships of chief executive officers and the social organizations (clubs) to which they belong, from the Minneapolis-St. Paul area from 1985.



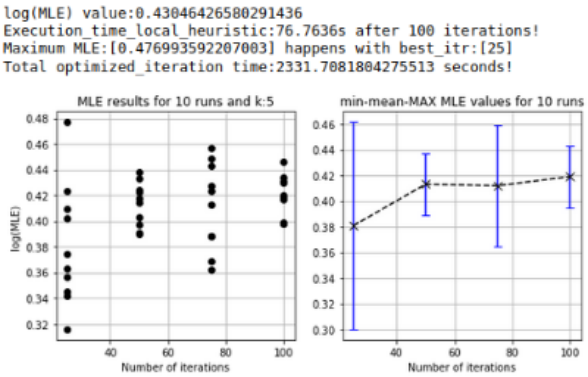
(a) Original network.



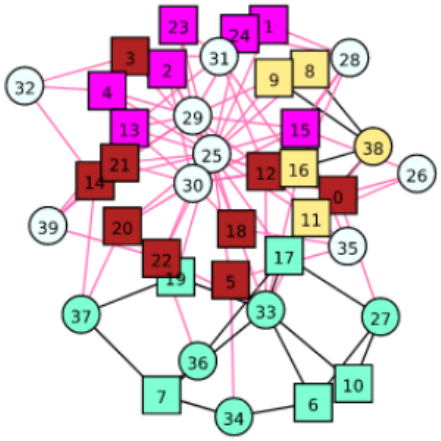
(b) Communities with MDL.



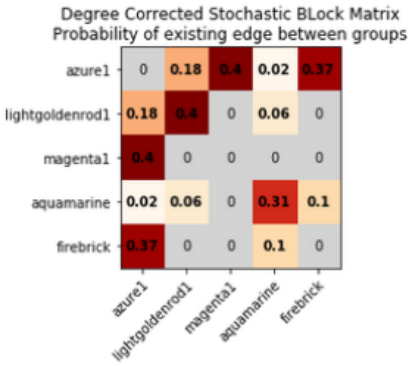
(c) Adjacency matrix.



(d) Best MLE with different runs.



(e) Detected communities.



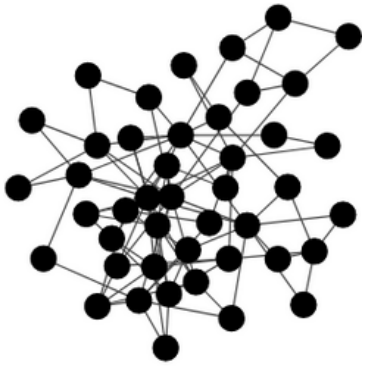
(f) Community information.

Figure 8: CEO and clubs network from the Minneapolis-St. Paul area.

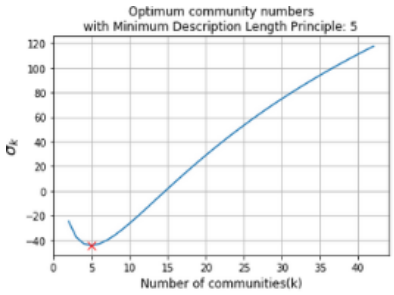
The result shows five detected groups where a group can be a combination of persons, organizations or both. An example is the group of white colored nodes where the members do not have any explicit connections with each other but they behave similarly in the network.

5.2.4 Barnes-Burkett elite affiliations (1962)

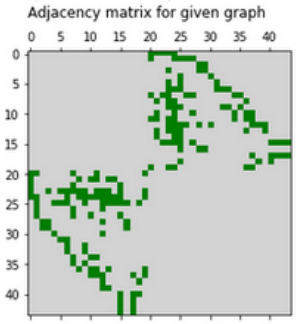
This bipartite graph represents affiliations among elite individuals and the corporate, museum, university boards, or social clubs to which they belonged, from 1962.



(a) Original network.

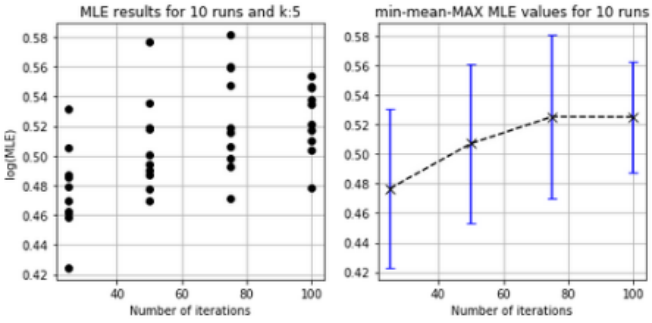


(b) Communities with MDL.

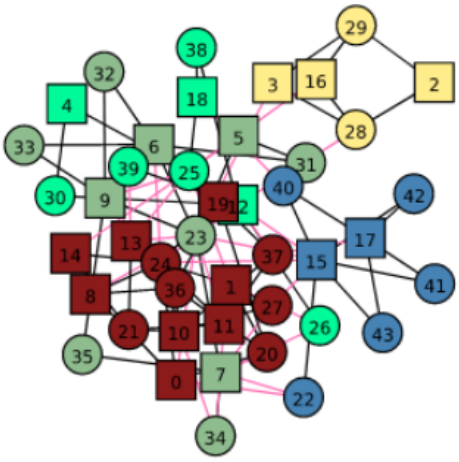


(c) Adjacency matrix.

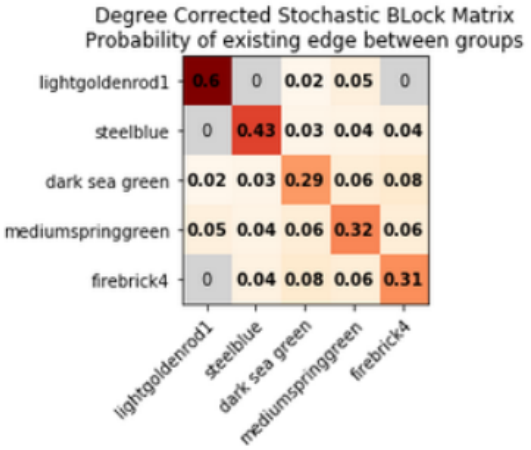
```
log(MLE) value:0.5455372583135634
Execution_time_local_heuristic:102.1285s after 100 iterations!
Maximum MLE:[0.5813705049832544] happens with best_itr:[75]
Total optimized_iteration time:2528.255491256714 seconds!
```



(d) Best MLE with different runs.



(e) Detected communities.



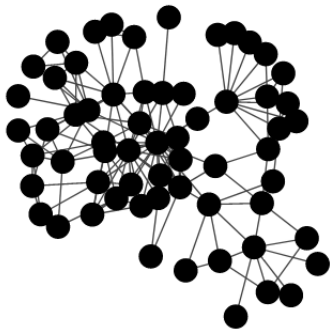
(f) Community information.

Figure 9: Network of affiliations among individuals and organizations.

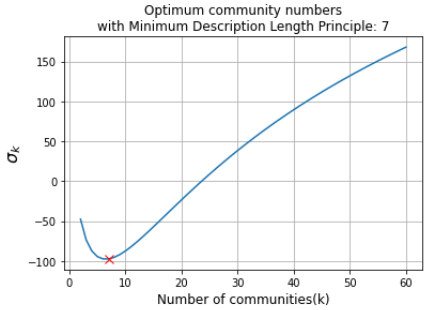
From the matrix, we can observe the most intensive collaborations or internal links among nodes of the group in lightgoldenrod1 color. In contrast with small number of members in lightgoldenrod1 colored group, firebrick4 group has connection with three other groups (except lightgoldenrod1 group).

5.2.5 9-11 terrorist network

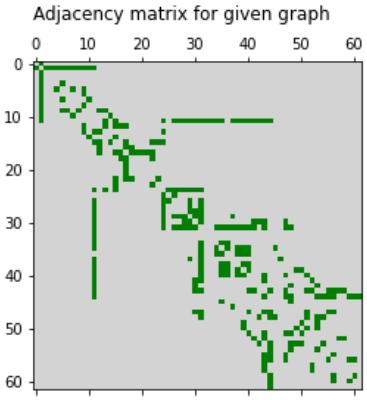
This unipartite network represents the network of individuals and their known social associations, centered around the hijackers that carried out the September 11th, 2001 terrorist attacks.



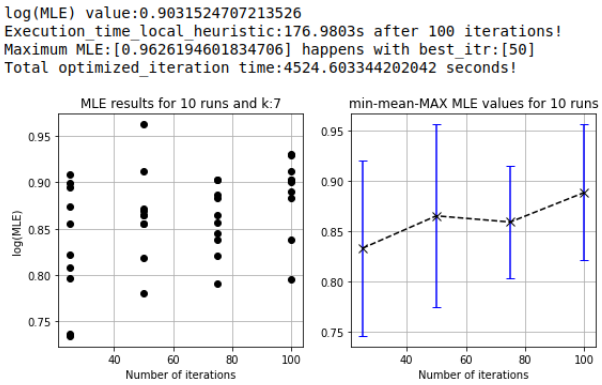
(a) Original network.



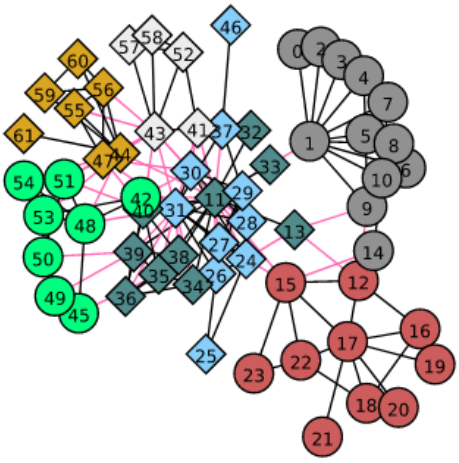
(b) Communities with MDL.



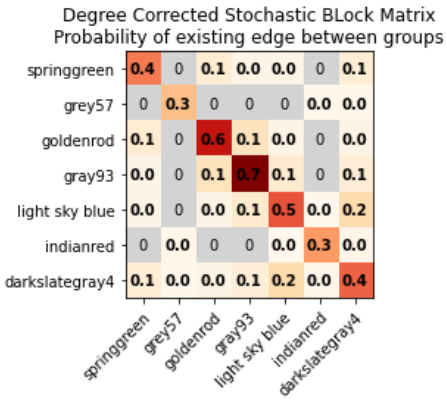
(c) Adjacency matrix.



(d) Best MLE with different runs.



(e) Detected communities.



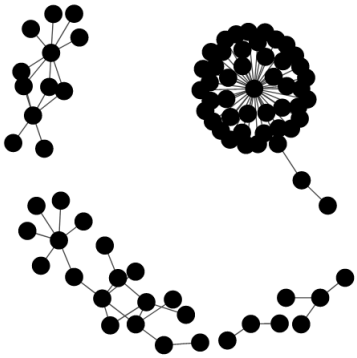
(f) Community information.

Figure 10: Network of individuals and their known social associations around September 11th attack.

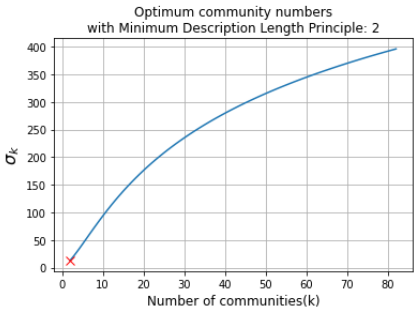
The result shows the internal and external connections of terrorists in seven different groups and the way they organized to execute the attack.

5.2.6 Baseball steroid use (2008)

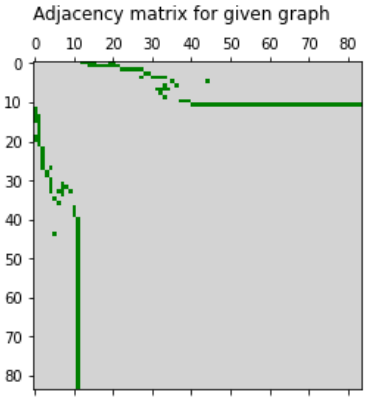
This bipartite network represents steroid use among baseball players. This is a network of players (circular nodes) and their steroid providers (square nodes), see Figure 11 (e).



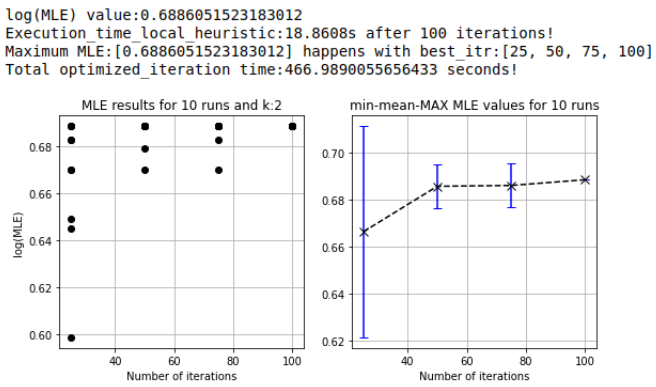
(a) Original network.



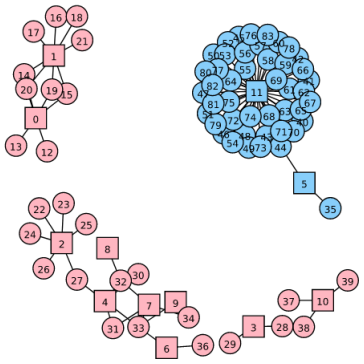
(b) Communities with MDL.



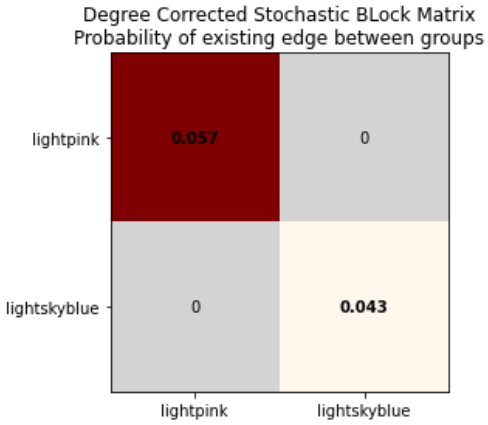
(c) Adjacency matrix.



(d) Best MLE with different runs.



(e) Detected communities.



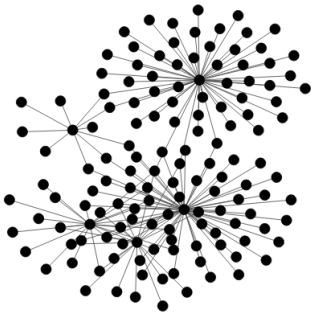
(f) Community information.

Figure 11: A network of baseball players and steroid providers.

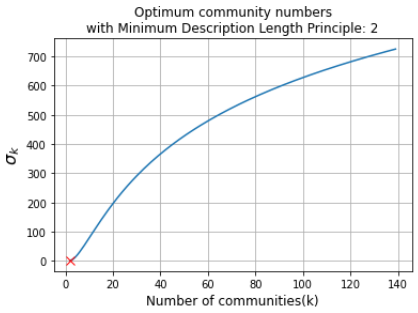
Figure 11 (a) shows some providers are easy to identify, but the result in Figure 11 (e) helps to identify all providers including the smaller ones. We can observe that node number 11, representing a steroid provider, supplies for a large number of players compared to other providers.

5.2.7 American Revolutionary groups (1765-1783)

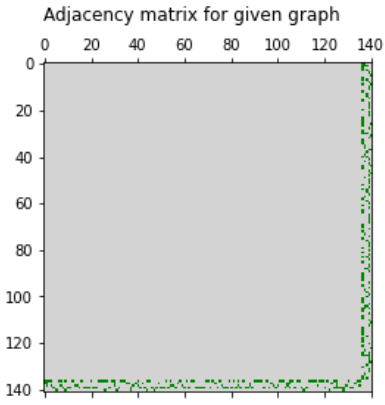
A bipartite network of the memberships of notable people and organizations, from the American Revolution (1765-1783) between users and groups on YouTube, extracted from a larger YouTube network in 2007.



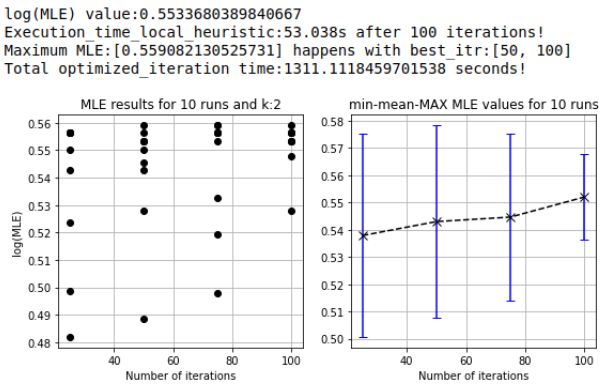
(a) Original network.



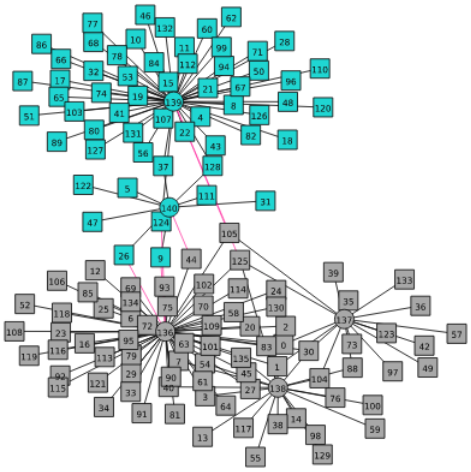
(b) Communities with MDL.



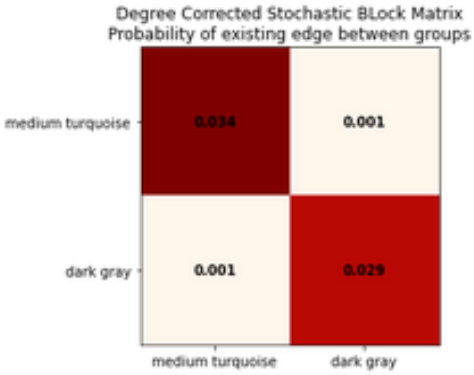
(c) Adjacency matrix.



(d) Best MLE with different runs.



(e) Detected communities.



(f) Community information.

Figure 12: A network of users and groups on YouTube around American Revolution subject.

The result shows how it is possible to identify two communities of youtube users and youtube groups based on their ideas about American revolution.

6 Conclusions

6.1 Summary

The aim of this project is to understand how communities in bipartite networks can be detected by Stochastic Block Model (SBM). For this reason, we study the concept of standard SBM and how it uses likelihood estimation to find the best community formation in a network. However, the standard model cannot detect correct communities in skewed networks. To solve this problem we study an extended model known as degree-corrected SBM to capture any possible community pattern which standard version cannot detect.
Apart from obtaining statistical knowledge and coding experience in this project, we learn two algorithms used to search for the best communities based on likelihood values (most possible results) in a feasible manner and to find an optimum number of communities. These algorithms are based on heuristic search and minimum description length principle respectively.
The code is verified and applied to a number of real networks of large size.

6.2 Limitations

This work does have few limitations. The developed code works only for undirected, non-weighted networks. Moreover the code does not work on graphs neither with self-loops nor multi-edges. These areas are left for further study. For better performance, we suggest to implement other search algorithms such as Louvain or Markov-Chain Monte-Carlo (MCMC) and last but not least, switching from Python to a high performance programming language like C (with possible parallelization).

6.3 Learning lessons

SBM identifies the most probable community formation by finding maximum likelihood value, but it also provides additional information which is likelihood ratio. Between any two combinations for a network, by comparing their likelihood values, we can determine to what extend one of them is more probable than the other one. When the degree distribution is heterogeneous in nature, standard SBM fails to detect the true communities. The degree-corrected model correctly ignores divisions based solely on degree and hence is more sensitive to underlying structure. The performance of the code for different networks but with two different computational units is compared at Table 1.

Table 1: Execution times for some experiments with two different computational units

				CPU1[1]		**CPU2**[2]	
Networks	**Nodes, Edges**	**k**[3]	**I**[4]	**Time(s)**	**MLE**	**Time(s)**	**MLE**
CEOs and social clubs	40, 95	5	125	41.4862	0.476100	79.9764	0.457511
9/11 terrorist attack	62, 152	7	100	180.6786	0.848257	355.0968	0.884689
Youtube groups and users	141, 160	2	125	63.6838	0.559082	135.6801	0.559082
Elite Individuals and organisations	44, 99	5	100	44.1877	0.512380	76.8285	0.526460
Baseball players steroids	84, 84	2	100	18.6116	0.688605	53.0727	0.688605
South African companies	11, 13	2	100	0.3895	0.346573	0.847	0.346573
Zebras	27, 111	7	100	32.1229	0.528010	107.9277	0.528010

[1] Intel i5-8250U CPU @ 1.60GHz
[2] Intel i5-2410M CPU @ 2.30GHz
[3] Number of communities
[4] Number of iterations

Table 1 represents that a considerably longer execution time is required when the number of communities are more for a smaller network compared to the execution time with fewer number of communities for a larger network. Due to the random nature of initial grouping in heuristic algorithm, in the code, the chance of capturing a higher likelihood value increases considerably by M iterations and N runs instead of only $M \times N$ iterations and one run. We observe from the results that 75 to 100 iterations as a minimum base with ten runs as we showed for experiments in the result section.

References

[1] Blelloch, Guy: *Parallel and Sequential Data Structures and Algorithms — Lecture notes.* 2011

[2] Brian Karrer, M. E. J. N.: *Stochastic blockmodels and community structure in networks.* 2010

[3] Britannica: *Random variables and probability distributions.* 2021

[4] Clauset, Aaron: *Lecture 17, Network Analysis and Modeling, CSCI 5352.* 2013

[5] Elsevier: *Statistical Inference.* 2021

[6] Foster, David: *Generative Modeling.* 2020

[7] Gallistel, C. R.: *Bayes for Beginners: Probability and Likelihood.* 2015

[8] Georgios A Pavlopoulos, Athanasia Pavlopoulou Costas Bouyioukos Evripides Markou Pantelis G B.: *Bipartite graphs in systems biology and medicine: a survey of methods and applications.* 2018

[9] Karrer, Brian ; Newman, M. E. J.: Stochastic blockmodels and community structure in networks. In: *Phys. Rev. E* 83 (2011), Jan, S. 016107

[10] Paula Peixoto, Tiago de: *network catalogue, repository and centrifuge.* 2020

[11] Peixoto, Tiago: *Parsimonious Module Inference in Large Networks.* 12 2012

[12] school, Bristol medical: *4 Ideas of statistical inference.* 2021

[13] StackExchange: *What is the difference between "likelihood" and "probability"?* 2010

[14] StackExchange: *What's the difference between probability and statistics?* 2015

[15] TechieDelight: *Terminology and Representations of Graphs.* 2020

[16] Tzu-Chi Yen, Johnson E.: *Heuristic and inference engine for estimating the number of communities in a bipartite network.* 2020

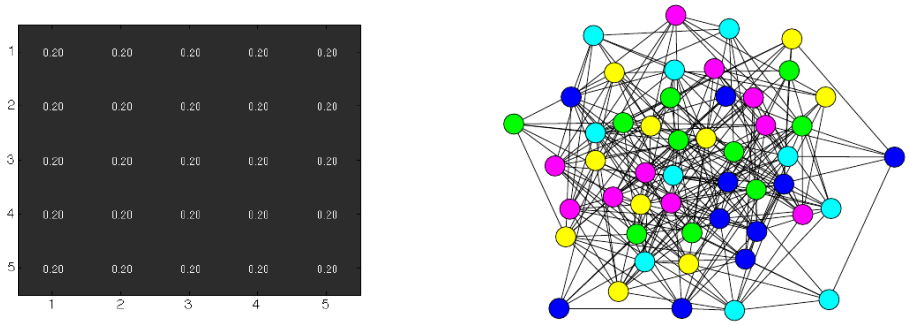
[17] Wikipedia: *Brute-force search.* 2020

[18] Wikipedia: *Stochastic block model.* 2020

7 Appendix

7.1 Graph types

Random graphs
In case of having a constant probability for communication among all communities in a stochastic block matrix, SBM reduces to Erdös-Renyi (ER) random graph G(n,p). G is an instance of graph with n number of vertices and probability p for existing of an edge.
For k equals to 5, Figure 13 is an example of a stochastic block matrix and a corresponding network instance drawn from it.

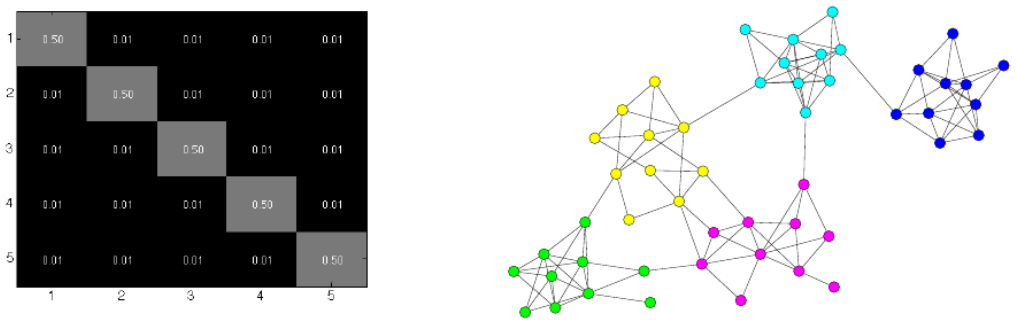


stochastic block matrix random graph

Figure 13: SBM with $\omega_{ij} = const$ reduced to ER and a generate network sample
Source: Lecture notes by Professor Aaron Clauset, University of Colorado, Boulder

In case of having different values for ω_{ij}, SBM still generates Erdos-Renyi random graphs but within each community i, with a specific internal density ω_{ii} but with random bipartite graphs between each pair of communities.

Assortative communities
When vertices tend to connect to vertices that are like them we have an assortative community. In other words, there are more connections (edges) within communities then among them. In mathematical representation, this tendency shows itself by greater probabilities values on the diagonal in compare to off-diagonal values, i.e. $\omega_{ii} > \omega_{ij}$, see Figure 14.



stochastic block matrix assortative communities

Figure 14: SBM with $\omega_{ii} > \omega_{ij}$ for Assortative communities
Source: Lecture notes by Professor Aaron Clauset, University of Colorado, Boulder

This is what we normally expect from communities, which is higher connectivity rate among vertices with some similarity among them. Similar preference exists with modularity function where networks with high modularity have dense connections between the nodes within modules but sparse connections between nodes in different modules.

Disassortative communities
In contrast with assortative communities, when vertices in a network tend to connect to vertices in other groups, we have disassortative communities. It means probability of existing an edge between two unlike vertices is more than two similar ones. This structure shows itself by higher values for off-diagonal values in probability matrix in SBM, see Figure 15.

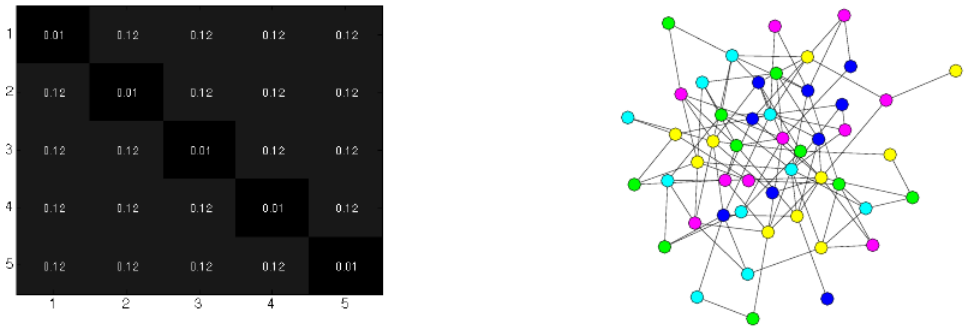


Figure 15: SBM with $\omega_{ii} < \omega_{ij}$ for Disassortative communities
Source: Lecture notes by Professor Aaron Clauset, University of Colorado, Boulder

Disassortative communities are visually similar to Erdös-Renyi (ER) random graphs but with much less or almost zero probability of connection between similar vertices (internal edges).

Core-periphery communities
When the density of connections between communities decreases with the community index, we have a Core-periphery structure. On way of having such a structure is to have exponentially decreasing probability on diagonal elements in probability matrix while still each community of the network connects to all other communities, see Figure 16.

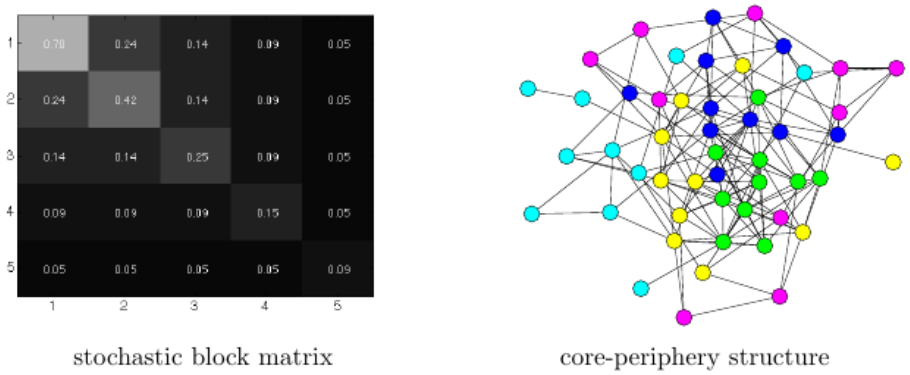


Figure 16: SBM with exponential probability edge distribution for Core-periphery communities
Source: Lecture notes by Professor Aaron Clauset, University of Colorado, Boulder

In Figure 16, the upper left corner probability matrix shows green community connections as the inner core, while the magenta and cyan communities are the outer peripheries of this core-periphery network.

Ordered communities

When in a network, communities are connected to each other according to a hidden sequence, we have an ordered network.

In terms of mathematical representation for this type of network structure, the probability matrix shows similar pattern as in assortative communities in addition to bigger values for the first-off-diagonal components, i.e. $M_{i,i} \approx M_{i,i+1} \approx M_{i,i1}$.

In other works, communities tend to connect to those just above and below themselves in the latent ordering, see Figure 17.

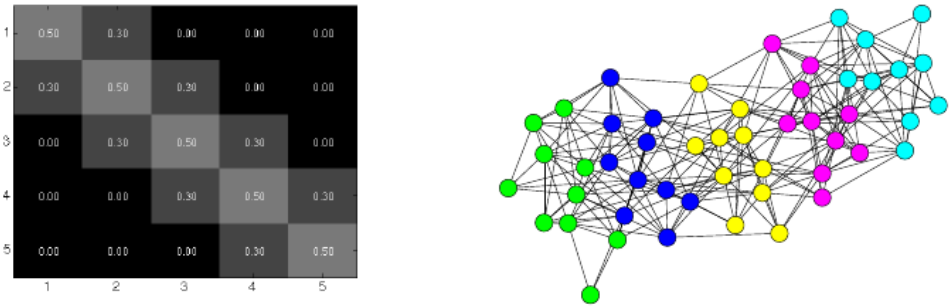


Figure 17: SBM with $M_{i,i} \approx M_{i,i+1} \approx M_{i,i1}$ for Ordered communities

Source: Lecture notes by Professor Aaron Clauset, University of Colorado, Boulder

7.2 Python code

7.2.1 Brute force search

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from itertools import * # to generate combinations

vtx_name6 = np.array((0,1,2,3,4,5))
edge_set6 = np.array([[0,1], [0,2], [1,2], [2,3], [3,4], [3,5], [4,5]])

# https://stackoverflow.com/questions/40709488/all-possibilities-to-split-a-list-into-two-lists
def all_possibilites(vtx_array):
    red = []
    blue = []
    for part1 in product([True,False],repeat=len(vtx_array)):
        red.append([p[1] for p in zip(part1,vtx_array) if p[0]])
        blue.append([p[1] for p in zip(part1,vtx_array) if not p[0]])
    return red, blue


# arrays to keep all possible combinations of two groups
red = []
blue = []

# arrays to keep all Maximum Likihood Estimations
all_mle_values = []
all_mle_values = np.array(all_mle_values)
all_log_mle_values = []
all_log_mle_values = np.array(all_log_mle_values)

red, blue = all_possibilites(vtx_name6)

# removing cases with empty or all points in one groups
red = red[1:-1]
blue = blue[1:-1]
print(type(blue), len(blue), type(red), len(red))
red
```

7.2.2 Any number of communities with brute force search

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from itertools import * # to generate combinations

# k = input("Enter the nbr of groups:")
k = 2 # hard coded for example siz_points
vtx_name6 = np.array((0,1,2,3,4,5))
edge_set6 = np.array([[0,1], [0,2], [1,2], [2,3], [3,4], [3,5], [4,5]])


def MLE_method(red_vtx_name, blue_vtx_name, edge_set):
    n_red = len(red_vtx_name) #red_vtx_name.size
    n_blue = len(blue_vtx_name) #blue_vtx_name.shape[0]
    # print(f'red_vertices:{n_red}, blue_vertices:{n_blue}')

    # total possible edges inside and outside groups
    intra_red_total = n_red * (n_red - 1) / 2
    intra_blue_total = n_blue * (n_blue - 1) / 2
    inter_edges_total = n_red * n_blue # N_ij

    # To solve 0/0 problem in case of 1 point in a group (check with Davide)
    if intra_red_total == 0:
        intra_red_total = 1
    elif intra_blue_total == 0:
        intra_blue_total = 1
```

```
    all_connections = np.isin(edge_set, red_vtx_name)


    intra_red = intra_blue = inter_edges = 0
    for c in all_connections:
        if c[0]==True and c[1]==True:
            intra_red += 1
        elif c[0]==False and c[1]==False:
            intra_blue += 1
        else:
            inter_edges += 1

    groups = ['red', 'blue']
    # stochastic_matrix
    sm = pd.DataFrame(np.zeros((2, 2)), columns = groups)
    sm.index = groups
    sm['red']['red'] = intra_red / intra_red_total
    sm['blue']['blue'] = intra_blue / intra_blue_total
    sm['red']['blue'] = sm['blue']['red'] = inter_edges / inter_edges_total

    # Expected edges
    E = pd.DataFrame(np.zeros((2, 2)), columns= groups)
    E.index = groups
    E['red']['red'] = intra_red
    E['blue']['blue'] = intra_blue
    E['red']['blue'] = E['blue']['red'] = inter_edges

    # Possible edges
    N = pd.DataFrame(np.zeros((2, 2)), columns= groups)
    N.index = groups
    N['red']['red'] = intra_red_total
    N['blue']['blue'] = intra_blue_total
    N['red']['blue'] = N['blue']['red'] = inter_edges_total
    # print(f'N:\n{N}\n')

    iteration = 1
    mle_value = 1 # Equestion 2, Lecture 17 (Aaron Clauset)
    log_mle_value = 0 # Equestion 3, Lecture 17 (Aaron Clauset)

    for i in range(len(groups)):
        for j in range(len(groups)):
            # In an ndirected network, edge ij contributes but edge ji NOT!
            if i <= j:
                if (E.iloc[i][j] == 0): # case with 0 edges between a pair of groups 0^0 = 1! /
                     (sm[i][j])**E[i][j])
                    mle_value *= (1-sm.iloc[i][j])**(N.iloc[i][j]-E.iloc[i][j])
                    log_mle_value += (N.iloc[i][j]-E.iloc[i][j])*np.log(N.iloc[i][j]-E.iloc[i][j]) - /
                         N.iloc[i][j]*np.log(N.iloc[i][j])


                elif (N.iloc[i][j]-E.iloc[i][j] == 0): # case with all edges between a pair of groups
                    mle_value *= ((sm.iloc[i][j])**E.iloc[i][j])
                    log_mle_value += E.iloc[i][j]*np.log(E.iloc[i][j]) - N.iloc[i][j]*np.log(N.iloc[i][j])

                else:
                    mle_value *= /
                         ((sm.iloc[i][j])**E.iloc[i][j])*((1-sm.iloc[i][j])**(N.iloc[i][j]-E.iloc[i][j]))
                    log_mle_value += E.iloc[i][j]*np.log(E.iloc[i][j]) + /
                         (N.iloc[i][j]-E.iloc[i][j])*np.log(N.iloc[i][j]-E.iloc[i][j]) - /
                         N.iloc[i][j]*np.log(N.iloc[i][j])

                iteration += 1
    return mle_value,log_mle_value


def all_possibilites(vtx_array):

    red = []
    blue = []
```

```
    for part1 in product([True,False],repeat=len(vtx_array)):
        red.append([p[1] for p in zip(part1,vtx_array) if p[0]])
        blue.append([p[1] for p in zip(part1,vtx_array) if not p[0]])

    return red, blue


# arrays to keep all possible combinations of two groups
red = []
blue = []

# arrays to keep all Maximum Likihood Estimations
all_mle_values = []
all_mle_values = np.array(all_mle_values)
all_log_mle_values = []
all_log_mle_values = np.array(all_log_mle_values)

red, blue = all_possibilites(vtx_name6)

# removing cases with empty or all points in one groups
red = red[1:-1]
blue = blue[1:-1]
print(type(blue), len(blue), type(red), len(red))


for i in range(len(red)):
    # print(red[i], blue[i])
    x, y = MLE_method(red[i], blue[i], edge_set6)
    all_mle_values = np.append(all_mle_values, x)
    all_log_mle_values = np.append(all_log_mle_values, y)


nbr_graphs = np.array(range(len(all_mle_values)))
fig, ax1 = plt.subplots()
color = 'tab:red'
ax1.set_xlabel('Possible␣number␣of␣graphs')
ax1.set_ylabel('MLE', color=color)
ax1.plot(nbr_graphs, all_mle_values, color=color)
ax1.tick_params(axis='y', labelcolor=color)

 # instantiate a second axes that shares the same x-axis
ax2 = ax1.twinx()
color = 'tab:blue'
ax2.set_ylabel('Log(MLE)', color=color)
ax2.plot(nbr_graphs, all_log_mle_values, color=color)
ax2.tick_params(axis='y', labelcolor=color)

fig.tight_layout()
plt.show()


# Most possible graph
max_idx = list(np.where(all_mle_values == np.amax(all_mle_values)))
# max_idx = np.where(all_log_mle_values == np.amax(all_log_mle_values))
print(f'Graph␣number:{max_idx[0]}')
for i in range(len(max_idx[0])):
    print (f'set{i+1}:␣RED{red[max_idx[0][i]]}␣BLUE{blue[max_idx[0][i]]}')


# test case1
MLE_method(np.array([0,1,2]), np.array([3,4,5]), edge_set6)
```

7.2.3 Final version with heuristic search

```
get_ipython().run_line_magic('matplotlib', 'inline')
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import itertools as itt # to generate combinations
```

```
import igraph as ig
import random # to uneven split
import copy
import time
from math import comb # to edge finding
from itertools import islice # to uneven split
import statistics

from IPython.core.display import display, HTML
display(HTML("<style>.container␣{␣width:100%␣!important;␣}</style>"))
display(HTML("<font␣color='green'>Done!</font>"))

import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
import scipy.sparse as sparse
import math


# https://stackoverflow.com/questions/3589214/generate-random-numbers-summing-to-a-predefined-value
def uneven_split(chunks, mylist):
    #Return a randomly chosen list of n positive integers summing to total

    length = len(mylist)
    dividers = sorted(random.sample(range(1, length), chunks - 1))

    group_size = [a - b for a, b in zip(dividers + [length], [0] + dividers)]

    # https://stackoverflow.com/questions/38861457/splitting-a-list-into-uneven-groups
    iterator = iter(mylist)
    groups =[list(islice(iterator, 0, i)) for i in group_size]

    groups_dict = {}
    for i in range(len(groups)):
        for j in range(len(groups[i])):
            groups_dict[groups[i][j]] = i

    return groups, groups_dict


# igraph for visualization
def print_my_graph(vtxset, edgeset, groups, save, k, bipartite, graph_size, my_form):

    if (type(groups) is dict):
        # print("DICT CONVERSION!")
        lst = [[] for _ in range(k)]
        for key,value in groups.items():
            lst[value].append(key)
        # print(lst)
        groups = lst

    g = ig.Graph()
    g.add_vertices(list(range(len(vtxset))))
    # print(edgeset)
    g.add_edges(edgeset)

    colors = list((ig.drawing.colors.known_colors).keys())
    shapes = ['rectangle', 'circle', 'triangle-up', 'triangle-down', 'diamond'] #hidden #null

    global my_colors
    my_colors = []
    my_shapes = []

    for i in range(k):
        my_colors.append(random.choice(colors))
        my_shapes.append(random.choice(shapes))

    g.vs["group_nbr"] = 0

    # to find related group for each vertex
    for i in range(len(vtxset)):
```

```
        for j in range(len(groups)):
            if i in groups[j]:
                g.vs[i]["group_nbr"] = j
    # print(f'my_colors:{my_colors}')

    #print(f'edgeset:{edgeset}')
    # https://igraph.org/python/doc/igraph.EdgeSeq-class.html
    my_intra_edges = []
    for i in range(len(groups)):
        temp = g.es.select(_within = groups[i])
        for j in range(len(temp)):
            my_intra_edges.append((temp[j]).tuple)

    #print(f'my_intra_edges:{my_intra_edges}')

    visual_style = {}
    visual_style["vertex_size"] = 20
    visual_style["vertex_label"] = g.vs["name"]
    visual_style["vertex_color"] = [my_colors[group_nbr] for group_nbr in g.vs["group_nbr"]]

    if bipartite:
        print("A bipartite network!")
        g.es["source"] = [e.source for e in g.es]
        # g.es["target"] = [e.target for e in g.es]
        visual_style["vertex_shape"] = [shapes[0] if source in set(g.es["source"]) else shapes[1] for /
            source in list(range(g.vcount()))]
        # [f(x) if condition else g(x) for x in sequence]

    else:
        visual_style["vertex_shape"] = [my_shapes[group_nbr] for group_nbr in g.vs["group_nbr"]]

    inter_ec, intra_ec = 'Hot Pink', 'black'
    visual_style["edge_color"] = [intra_ec if edge in my_intra_edges else inter_ec for edge in /
        g.get_edgelist()]

    visual_style["layout"] = my_form #"kk" # large circular grid_fr
    visual_style["vertex_label_size"] = 9
    visual_style["bbox"] = (graph_size, graph_size)

    if save == True:
        visual_style["vertex_label_size"] = 9
        visual_style["vertex_size"] = 15
        visual_style["bbox"] = (graph_size, graph_size)
        ig.plot(g, "SBM.png", **visual_style);
        print(f'A graph with maximum degree:{g.maxdegree()} at vertex/vertices:{g.vs.select(_degree = /
            g.maxdegree())["name"]} saved.')
    else:
        ig.summary(g)

        print(f"Edge colors:\nInter-edges: {inter_ec}\nIntra-edges: {intra_ec}")
        return (ig.plot(g, **visual_style))


# params: splited vertices as 'groups' wtih type dictionary and igraph object as 'igraph_obj'
def edge_prepping_dict(groups, igraph_obj):
    # print(f'groups:{groups}')
    gp_ids = list(set(groups.values())) # to get a unique set of values which are group numbers!
    k = len(gp_ids)
    # print(f'groups:{groups}, gp_ids:{gp_ids}, k:{k}')

    existing_intra = {} # dictionary to save the number of intra edges of each community
    cardinality_each_group = {} # dictionary to save the number of vertices of each community
    possible_intra = {} # dictionary to save the number of possible intra edges of each community
    degree_each_group = {} # dictionary to save the degree of each community

    #Intra Edges
    for i in range(len(gp_ids)): # i as index is needed for keys in dictionaries (group identification)
        gp = [k for k,v in groups.items() if v == gp_ids[i]] # one group; to get the keys which are /
            vertices with a value as group id groups.items()[i]
        existing_intra[i] = igraph_obj.subgraph(gp).ecount() #subgraph based on vertices
```

```
        #if existing_intra[i] == 0:
            #existing_intra[i] = 1

        # cardinality_each_group[i] = igraph_obj.subgraph(gp).vcount()
        cardinality_each_group[i] = sum(value == gp_ids[i] for value in groups.values())

        possible_intra[i] = comb(cardinality_each_group[i], 2)

        degree_each_group[i] = sum(igraph_obj.degree(gp))

    #Inter edges
    possible_inter = {}
    existing_inter = {}
    total_edges_group_pairs = {} # axilary dictionary
    intra_two_groups = {} # axilary dictionary

    # https://stackoverflow.com/questions/3294889/iterating-over-dictionaries-using-for-loops
    for i in cardinality_each_group.keys():
        for j in cardinality_each_group.keys():
            if i < j:
                possible_inter[(i,j)] = cardinality_each_group[i] * cardinality_each_group[j] # saving /
                     inter edges, key shows group numbers

    # print(f"possible_inter:{possible_inter}")

    for i in range(len(gp_ids)):
        for j in range(len(gp_ids)):
            if i < j:
                gp1 = [k for k,v in groups.items() if v == gp_ids[i]]
                gp2 = [k for k,v in groups.items() if v == gp_ids[j]]
                total_edges_group_pairs[(i,j)] = igraph_obj.subgraph(gp1 + gp2).ecount()
                intra_two_groups[(i,j)] = existing_intra[i] + existing_intra[j]


     # https://www.datacamp.com/community/tutorials/python-dictionary-comprehension
    existing_inter = {key:(total_edges_group_pairs[key] - intra_two_groups[key]) for (key,value) in /
         zip(total_edges_group_pairs, intra_two_groups)}

    # TAKING CARE OF ZERO EXISTING EDGES
    for i in range(len(gp_ids)):
        if possible_intra[i] == 0:
            possible_intra[i] = 1

    return existing_intra, existing_inter, possible_intra, possible_inter, degree_each_group, /
         cardinality_each_group


### Objective-function: maximum probability likelihood
def standard_SBM(intra_exist, inter_exist, intra_possible, inter_possible):

    #conversion of dictionaries to list
    intra_exist = np.array(list(intra_exist.values()))
    inter_exist = np.array(list(inter_exist.values()))
    intra_possible = np.array(list(intra_possible.values()))
    inter_possible = np.array(list(inter_possible.values()))

    k = intra_exist.size # nbr of groups

    # SM = np.zeros([k,k]) # stochastic_matrix
    E = np.zeros([k,k]) # Expected edges as intra_exist for diagonal and inter_exist non_diagonal
    N = np.zeros([k,k]) # Possible edges

    E[np.diag_indices(k)] = intra_exist # diagonal values
    N[np.diag_indices(k)] = intra_possible # diagonal values

    # index production to save inter_exist and inter_exist_total values in correct cells
    itr = 0
    for i in range(k-1):
        for j in range(k):
            if i < j:
```

```
            E[i][j] = E[j][i] = inter_exist[itr]
            N[i][j] = N[j][i] = inter_possible[itr]
            itr +=1

    # print(f'Expected edges inside and between groups:\n{E}')

    ## CALCULATION PART
    log_mle_value = 0 # Equestion 3, Lecture 17 (Aaron Clauset)
    for x in range(k):
        for y in range(k):
            if x <= y:
                if (E[x][y] == 0): # case with 0 edges between a pair of groups 0^0 = 1! /
                     (SM[i][j])**E[i][j])
                    pass
                    #continue

                elif (N[x][y]-E[x][y] == 0): # case with all edges between a pair of groups
                    pass
                    #continue
                else:
                    log_mle_value += E[x][y]*np.log(E[x][y]) + /
                         (N[x][y]-E[x][y])*np.log(N[x][y]-E[x][y])-N[x][y]*np.log(N[x][y])

    # SM = E/N
    return log_mle_value


# def degree_corrected_SMB(intra_exist, inter_exist, gp_degree)

# In[ ]:


def degree_corrected_SMB(intra_exist, inter_exist, gp_degree):

    # print(f'DC_SBM->intra_exist:{intra_exist}')
    intra_exist = np.array(list(intra_exist.values()))
    inter_exist = np.array(list(inter_exist.values()))
    gp_degree = np.array(list(gp_degree.values()))

    k = intra_exist.size # nbr of groups
    # print(f'Number of groups:{k}')
    m = sum(intra_exist) + sum(inter_exist)
    # print(f'All edges in the network:{m}')

    E = np.zeros([k,k]) # As intra_exist for diagonal and inter_exist non_diagonal
    E[np.diag_indices(k)] = intra_exist # diagonal values

    # index production to save inter_exist values in correct cells
    itr = 0
    for i in range(k-1):
        for j in range(k):
            if i < j:
                E[i][j] = E[j][i] = inter_exist[itr]
                itr +=1

    D = np.zeros([k,k]) # D for matrix of total degrees in each group
    D[np.diag_indices(k)] = gp_degree

    ## CALCULATION PART
    m2 = 2 * m
    # print(f'm2:{m2}\n--*--')

    dc_mle = 0 #equation 10 from lecture 17
    for x in range(k):
        for y in range(k):
            if (E[x][y] == 0):
                # print(f'2-> i:{i},j{j}:E=0!')
                continue
            else:
                if x == y:
```

```
                p_k = (E[x][y] * 2) / m2
                dc_mle += p_k * np.log(p_k /((D[x][x]/m2) * (D[y][y]/m2)))

            elif x < y:
                p_k = E[x][y] / m2
                dc_mle += p_k * np.log(p_k /((D[x][x]/m2) * (D[y][y]/m2)))

    return dc_mle


# this function reproduces the graph for the best MLE in a step!
def best_mle_whereabouts(best_change_instep, intra_exist, inter_exist, intra_possible, inter_possible, /
     gp_degree, gp_cardi, k_communities_dict):
    # Six terms needs a copy and an update as below

    key = best_change_instep[0]
    t = best_change_instep[1]
    value = k_communities_dict[key]
    ngbr = graph_obj.neighbors(key)

    gp_degree_temp = gp_degree.copy()
    gp_degree_temp.update({value: gp_degree_temp[value]-graph_obj.degree(key)})
    gp_degree_temp.update({t: gp_degree_temp[t]+graph_obj.degree(key)})

    gp_cardi_temp = gp_cardi.copy()
    gp_cardi_temp.update({value: gp_cardi_temp[value]-1})
    gp_cardi_temp.update({t: gp_cardi_temp[t]+1})

    intra_possible_temp = intra_possible.copy()
    for i in range(len(gp_cardi)):
        intra_possible_temp.update({i: 1 if gp_cardi_temp[i] == 1 else ((gp_cardi_temp[i] * /
             ((gp_cardi_temp[i])-1)) / 2)})

    inter_possible_temp = inter_possible.copy()
    for i in range(len(gp_cardi)):
        for j in range(len(gp_cardi)):
            if i<j:
                inter_possible_temp.update({(i,j):gp_cardi_temp[i] * gp_cardi_temp[j]})

    intra_exist_temp = intra_exist.copy()
    inter_exist_temp = inter_exist.copy()
    for n in ngbr: # n as key for each neighbor
        n_value = k_communities_dict[n] #n_value as group number

        if t == n_value:
            intra_exist_temp.update({t: intra_exist_temp[t]+1})
            inter_exist_temp.update({tuple(sorted((value, n_value))): /
                 inter_exist_temp[tuple(sorted((value, n_value)))]-1})

        elif value == n_value:
            intra_exist_temp.update({value: intra_exist_temp[value]-1})
            inter_exist_temp.update({tuple(sorted((t, n_value))): inter_exist_temp[tuple(sorted((t, /
                 n_value)))]+1})

        else:
            inter_exist_temp.update({tuple(sorted((n_value, t))): inter_exist_temp[tuple(sorted((n_value, /
                 t)))]+1})
            inter_exist_temp.update({tuple(sorted((value, n_value))): /
                 inter_exist_temp[tuple(sorted((value, n_value)))]-1})

    # to update k_communities_dict for next step!
    k_communities_dict_temp = k_communities_dict.copy()
    k_communities_dict_temp.update({key:t})

    return intra_exist_temp, inter_exist_temp, intra_possible_temp, inter_possible_temp, gp_degree_temp, /
         gp_cardi_temp, k_communities_dict_temp


def atomic_group(value, gp_cardi):
    # k, v is in cardinality dict. If 1 exists in values it means corresponding k in cardinality is
```

```
    # an atomic group.

    for k, v in gp_cardi.items():
        if v == 1:
            if k == value: # Now check if the group of our visiting vertex is that atomic group!
                return True
    return False


# param 'itr' is for several-runs with different random initial conditions
def local_heuristic(vtx, edges, k, k_communities_dict, graph_obj, mode, iterations):
    execution_time = time.time()

    # validity check for nbr of groups (k)
    nbr_vertices = len(k_communities_dict.keys())

    if k > nbr_vertices:
        k = nbr_vertices
        print(f'Selected␣number␣of␣groups␣exceeds␣number␣of␣vertices.␣Program␣continues␣with␣{k}␣vertices␣/
             for␣grouping!')

    mode = int(mode)
    SBM = ['Standard␣SBM', 'Degree␣corrected␣SBM']
    if (mode != 0) and (mode != 1):
        print (f"Entered␣mode:{mode}\nSet␣mode␣to:␣0␣->␣{SBM[0]}␣or␣1␣->␣{SBM[1]}")
        return
    else:
        print (f"---\nSelected␣mode␣->␣{[SBM[0]␣if␣mode==0␣else␣SBM[1]]}")

    itr = 0 # counter
    iterations_mle = [] # collection of best communities for iterations
    iterations_community = []

    gp_ids = list(set(k_communities_dict.values())) # to get a unique set of values which are group /
         numbers!
    print(f'gp_ids:{gp_ids}')

    flag = False # to run the algorithm with first randomized-graph and plot it outside the method
    for iteration in range(iterations): # 3th-FOR (most outer for-loop)

        one_move = [False] * len(vtx)
        vertex_watch = dict(zip(vtx, one_move)) # vtx:keys one_move:values changes to True if vtx moves!

        ### Step1: divide the network
        if flag:
            random.shuffle(vtx)
            _, k_communities_dict = uneven_split(k, vtx)
        flag = True

        ### Step2: move one vtx from one community to another
        intra_exist, inter_exist, intra_possible, inter_possible, gp_degree, gp_cardi = /
             edge_prepping_dict(k_communities_dict, graph_obj)

        if mode == 0:
            new_mle = standard_SBM(intra_exist, inter_exist, intra_possible, inter_possible)
        elif mode == 1:
            new_mle = degree_corrected_SMB(intra_exist, inter_exist, gp_degree)


        good_move = () # to save that move which leads to best MLE in each step
        state_mles = [] # to keep the best MLEs of all steps!
        state_mles.append(new_mle)
        state_communities = [] # to keep corresponding communities for best MLEs in state_mles
        state_communities.append(k_communities_dict)

        #print(f'k_communities_dict:{k_communities_dict}')
        for key in vertex_watch.keys(): # 2nd-FOR
            old_mle = None
            for key, value in k_communities_dict.items(): # 1st-FOR
                if (vertex_watch[key] == True): # subject to the restriction that each vertex may be moved /
```

```
            only once
            continue
        if atomic_group(value, gp_cardi):
            continue

        target = copy.deepcopy(gp_ids)
        # print(f'k:{k}, target:{target}, value:{value}')
        target.remove(value)
        ngbr = graph_obj.neighbors(key)
        for t in target:
            move = (key, t)
            # Six terms need a copy and an update as below

            gp_degree_temp = gp_degree.copy()
            gp_degree_temp.update({value: gp_degree_temp[value]-graph_obj.degree(key)})
            gp_degree_temp.update({t: gp_degree_temp[t]+graph_obj.degree(key)})

            gp_cardi_temp = gp_cardi.copy()
            gp_cardi_temp.update({value: gp_cardi_temp[value]-1})
            gp_cardi_temp.update({t: gp_cardi_temp[t]+1})

            intra_possible_temp = intra_possible.copy()
            for i in range(len(gp_ids)):
                intra_possible_temp.update({i: 1 if gp_cardi_temp[i] == 1 else ((gp_cardi_temp[i] /
                    * ((gp_cardi_temp[i])-1)) / 2)})

            inter_possible_temp = inter_possible.copy()
            for i in range(len(gp_ids)):
                for j in range(len(gp_ids)):
                    if i<j:
                        inter_possible_temp.update({(i,j):gp_cardi_temp[i] * gp_cardi_temp[j]})

            intra_exist_temp = intra_exist.copy()
            inter_exist_temp = inter_exist.copy()

            for n in ngbr: # n as key for each neighbor
                n_value = k_communities_dict[n] #n_value as group number

                if t == n_value:
                    intra_exist_temp.update({t: intra_exist_temp[t]+1})
                    inter_exist_temp.update({tuple(sorted((value, n_value))): /
                        (inter_exist_temp[tuple(sorted((value, n_value)))])-1})

                elif value == n_value:
                    intra_exist_temp.update({value: intra_exist_temp[value]-1})
                    inter_exist_temp.update({tuple(sorted((t, n_value))): /
                        inter_exist_temp[tuple(sorted((t, n_value)))]+1})

                else:
                    inter_exist_temp.update({tuple(sorted((n_value, t))): /
                        inter_exist_temp[tuple(sorted((n_value, t)))]+1})
                    inter_exist_temp.update({tuple(sorted((value, n_value))): /
                        inter_exist_temp[tuple(sorted((value, n_value)))]-1})

            if mode == 0:
                new_mle = standard_SBM(intra_exist_temp, inter_exist_temp, intra_possible_temp, /
                    inter_possible_temp)
            if mode == 1:
                new_mle = degree_corrected_SMB(intra_exist_temp, inter_exist_temp, gp_degree_temp)

            if (old_mle is None) or (new_mle > old_mle):
                # print(f'new_mle:{new_mle}, old_mle:{old_mle}')
                old_mle = new_mle
                good_move = move
                # print(f'good_move:{good_move}')

    if (old_mle is not None):)
        vertex_watch.update({good_move[0]: True})
        intra_exist, inter_exist, intra_possible, inter_possible, gp_degree, gp_cardi, /
            k_communities_dict = best_mle_whereabouts(good_move, intra_exist, inter_exist, /
```

```
                    intra_possible, inter_possible, gp_degree, gp_cardi, k_communities_dict)
                state_communities.append(k_communities_dict)
                state_mles.append(old_mle)

        state_mles_np = np.array(state_mles)
        state_communities_np = np.array(state_communities)

        # mle_accumulated_array = np.add.accumulate(state_mles_np)
        if mode == 0:
            mle_accumulated_array = state_mles_np[:-1]/state_mles_np[1:]
            index = np.argmax(mle_accumulated_array) + 1
        if mode == 1:
            mle_accumulated_array = np.diff(state_mles_np)
            index = np.argmax(mle_accumulated_array) + 1

        peak_community = state_communities_np[index] # one structure with best MLE
        peak_community_mle = state_mles_np[index] # one number as the best MLE in one iteration

        iterations_mle.append(peak_community_mle)
        iterations_community.append(peak_community)

        itr += 1
        if itr % 25 == 0:
            print(f"{itr}", end="␣")
        else:
            print("*", end="␣")

    iterations_mle_np = np.array(iterations_mle)
    iterations_community_np = np.array(iterations_community)
    result_index = np.argmax(iterations_mle_np)
    MLE = iterations_mle_np[result_index]
    COMMUNITY = iterations_community_np[result_index]

    print(f'\nlog(MLE)␣value:{MLE}')
    print(f"Execution_time_local_heuristic:{round((time.time()␣-␣execution_time),␣4)}s␣after␣{itr}␣/
         iterations!")
    return COMMUNITY, MLE

def build_graph(case, name, size):

    if case == 99:
        graph_obj = ig.Graph.Read_GML(name) # replace your file name to read it!
        vtx = list(range(graph_obj.vcount()))
        edges = graph_obj.get_edgelist()
        graph_size = (0, 0, size, size)
        color = 'black' # Honeydew Linen
        label = 10
        node_size = 16

    else:
        vtx = dict_graph_vtx[case]
        edges = dict_graph_edges[case]
        graph_obj = ig.Graph()
        graph_obj.add_vertices(vtx)
        graph_obj.add_edges(edges)
        graph_size = (0, 0, size, size)
        color = 'black' #'lightblue'
        label = 10
        node_size = 14

    # https://en.wikipedia.org/wiki/X11_color_names
    ig.summary(graph_obj)
    return graph_obj, vtx, edges, graph_size, node_size, label, color


def stochastic_BM(k, i_graph, result):
    print(f"Number␣of␣groups:{k}")

    lst = [[] for _ in range(k)]
    for key,value in result.items():
```

```
        lst[value].append(key)

    existing_intra, existing_inter, possible_intra, possible_inter, degree_each_group, /
         cardinality_each_group = edge_prepping_dict(result, i_graph)
    print(f'Degree␣of␣groups:{degree_each_group}')
    print(f'Cardinality␣of␣groups:{cardinality_each_group}')

    existing_intra = np.array(list(existing_intra.values()))
    existing_inter = np.array(list(existing_inter.values()))
    possible_intra = np.array(list(possible_intra.values()))
    possible_inter = np.array(list(possible_inter.values()))
    # ------------------------------------------------ #
    M = np.zeros([k,k]) # stochastic_matrix
    E = np.zeros([k,k]) # Expected edges as intra_exist for diagonal and inter_exist non_diagonal
    N = np.zeros([k,k]) # Possible edges

    E[np.diag_indices(k)] = existing_intra # diagonal values
    N[np.diag_indices(k)] = possible_intra # diagonal values

    # index production to save inter_exist and inter_exist_total values in correct cells
    itr = 0
    for i in range(k-1):
        for j in range(k):
            if i < j:
                E[i][j] = E[j][i] = existing_inter[itr]
                N[i][j] = N[j][i] = possible_inter[itr]
                itr +=1
    M = E/N
    return M


def print_M(stochastic_matrix, mode, gp_colors):

    k = len(stochastic_matrix)
    fig, ax = plt.subplots()
    # https://matplotlib.org/3.3.3/tutorials/colors/colormaps.html

    stochastic_matrix = np.ma.masked_where(stochastic_matrix==0, stochastic_matrix)
    cmap = plt.cm.OrRd
    cmap.set_bad(color='lightgray')
    im = ax.imshow(stochastic_matrix, cmap = cmap) # 'binary' 'copper'

    ax.set_xticks(np.arange(k))
    ax.set_yticks(np.arange(k))

    ax.set_xticklabels(gp_colors)
    ax.set_yticklabels(gp_colors)

    if k > 3:
        plt.setp(ax.get_xticklabels(), rotation=45, ha="right", rotation_mode="anchor")

    for i in range(k):
        for j in range(k):
            if stochastic_matrix[i, j] != 'masked':
                value = round(stochastic_matrix[i, j],2)
                text = ax.text(j, i, value, ha="center", va="center", color="k", fontsize=10, weight='bold')
            else:
                value = 0
                text = ax.text(j, i, value, ha="center", va="center", color="k", fontsize=10) # /
                     weight='bold'

    if mode == 0:
        ax.set_title("Standard␣Stochastic␣BLock␣Matrix\nProbability␣of␣existing␣edge␣between␣groups")
    elif mode == 1:
        ax.set_title("Degree␣Corrected␣Stochastic␣BLock␣Matrix\nProbability␣of␣existing␣edge␣between␣/
             groups")
    # cb = colorbar(label='a label')
    fig.tight_layout()
    plt.show()
```

```
# Input data!
k = int(input("Enter the nbr of groups:"))
case = int(input("Enter node numbers to run (6, 11, 32, 35 or 99 for gml file):"))

edges6 = np.array([[0,1], [0,2], [1,2], [2,3], [3,4], [3,5], [4,5]])
edges11 = np.array([[0,1], [0,2], [1,2], [10,3], [3,4], [3,5], [4,5],[6,7], [7,3], [6,4], [6,5], [0,9], /
    [3,8], [7,8], [0,3], [4,2], [1,9]])
edges32 = /
    np.array([[0,23],[0,24],[0,28],[0,30],[1,25],[1,26],[1,27],[1,28],[1,29],[1,30],[2,26],[2,28],[2,29]
,[3,26],[3,28],[3,29],[4,25],[4,26],[4,28],[4,29],[4,30],[5,19],[5,21],[5,22],[5,23],[5,24],[5,25],[5,26],[5,27]
,[5,28],[5,29],[5,31],[6,15],[6,16],[6,17],[6,19],[6,20],[6,21],[6,22],[6,28],[7,15],[7,16],[7,17],[7,18],[7,19]
,[7,20],[7,21],[7,22],[7,25],[7,26],[7,27],[7,29],[7,30],[7,31],[8,14],[8,15],[8,16],[8,17],[8,18]
,[8,19],[8,27],[8,28],[8,29],[8,30],[9,17],[9,18],[9,21],[10,16],[10,17],[10,21],[11,14],[11,16],[11,17],[11,18]
,[11,20],[11, 21],[12, 16],[12,18],[12,21],[13,14],[13,15],[13,16],[13,17],[13,18],[13,20]])
edges35 = np.array([[2, 1],[3, 1],[3, 2],[4, 1], [4, 2], [4, 3],[5, 1],[6, 1],[7, 1], [7, 5], [7, 6],[8, /
    1],[8, 2], [8, 3], [8, 4],[9, 1],[9, 3],[10, 3],[11, 1],[11, 5], [11, 6],[12, 1],[13, 1], [13, /
    4],[14, 1], [14, 2], [14, 3], [14, 4],[17, 6], [17, 7],[18, 1], [18, 2],[20, 1], [20, 2],[22, 1], /
    [22, 2],[26, 24], [26, 25],[28, 3], [28, 24], [28, 25],[29, 3],[30, 24], [30, 27],[31, 2], [31, 9], /
    [32, 1], [32, 25], [32, 26], [32, 29],[33, 3], [33, 9], [33, 15], [33, 16], [33, 19], [33, 21], /
    [33, 23], [33, 24], [33, 30], [33, 31], [33, 32],[34, 9], [34, 10], [34, 14], [34, 15], [34, 16], /
    [34, 19], [34, 20], [34, 21], [34, 23], [34, 24], [34, 27], [34, 28], [34, 29], [34, 30], [34, 31], /
    [34, 32], [34, 33]])

dict_graph_vtx = {6:[0,1,2,3,4,5], 11:list(range(11)), 32:list(range(32)), 35:list(range(35))}
dict_graph_edges = {6: edges6, 11: edges11, 32: edges32, 35: edges35}

name = "datasets/ceo_club.gml"
bipartite = True # True or False <- FOR VERTEX TYPE REPRESENTATION

my_graph_size = 250
my_form ='kk' #'large' #'tree'
#check_extra >>>>
graph_obj, vtx, edges, graph_size, node_size, label, color = build_graph(case, name, my_graph_size)
ig.plot(graph_obj, bbox=graph_size, vertex_color = color, vertex_size = node_size, layout = my_form, /
    vertex_label_size = label)


# Initial randomization and plot
random.shuffle(vtx)
k_communities, k_communities_dict = uneven_split(k, vtx)
print_my_graph(vtx, edges, k_communities, False, k, bipartite, my_graph_size, my_form)


result_SE, _ = local_heuristic(vtx, edges, k, k_communities_dict, graph_obj, 0, 100)
print_my_graph(vtx, edges, result_SE, False, k, bipartite, my_graph_size, my_form)

M_SE = stochastic_BM(k, graph_obj, result_SE)
print_M(M_SE, 0, my_colors)

result_DC, _ = local_heuristic(vtx, edges, k, k_communities_dict, graph_obj, 1, 100)
M_DC = stochastic_BM(k, graph_obj, result_DC)
print_my_graph(vtx, edges, result_DC, False, k, bipartite, my_graph_size, my_form)

print_M(M_DC, 1, my_colors)

#Adjacency matrix
adj_mtx = np.array(list(graph_obj.get_adjacency()))
cmap = ListedColormap(['lightgray', 'g'])
plt.matshow(adj_mtx, cmap=cmap)
plt.title('Adjacency matrix for given graph', loc='left', y=1.1)
plt.show()


clustering_coeff = graph_obj.transitivity_undirected()
print(f'Clustering_coeffecient: {clustering_coeff}')

assort_deg = graph_obj.assortativity_degree()
print(f'Degree assortativity: {assort_deg}')
```

```
def strictly_convex_k(graph):
    #E = total number of edges; N = total number of vertices
    E = graph.ecount()
    N = graph.vcount()
    print(f'E:{E},␣N:{N}')

    sigma_k_values = {}
    for k in range(2, N-1): # k is the number of communities
        m = k*k + k
        x = (m)/(2*E)
        h = (1+x)*(np.log(1+x)) - (x)*(np.log(x))
        sigma_k = E*h - (E-N)*(np.log(k))
        sigma_k_values.update({k:sigma_k})

    #print(f'sigma_k_values:{sigma_k_values}')

    optimal_k = min(sigma_k_values.keys(), key=(lambda k: sigma_k_values[k]))
    print(f'optimal_k:␣{optimal_k}')

    data = sorted(sigma_k_values.items()) # sorted by key, return a list of tuples
    x, y = zip(*data) # unpack a list of pairs into two tuples
    plt.grid()
    plt.plot(x, y)
    plt.ylabel(r'$\sigma_k$', size=16)
    plt.xlabel('Number␣of␣communities(k)', size=12)
    plt.title(f'Optimum␣community␣numbers\nwith␣Minimum␣Description␣Length␣Principle:␣{optimal_k}')
    plt.plot(optimal_k, sigma_k_values[optimal_k], 'rx', markersize=8)
    plt.show()

    return optimal_k


optimal_k = strictly_convex_k(graph_obj)


# A function to find necessary iterations for a given constant grouping (k)
def find_best_iteration(k, vtx, edges, graph_obj, mode, default, max_itr, interval):
    itrs = list(range(default, max_itr + interval, interval))
    print(f"Iterations␣to␣test:{itrs}")

    mle_itr_dict = {}
    for itr in itrs:
        _, k_communities_dict = uneven_split(k, vtx)
        _, result = local_heuristic(vtx, edges, k, k_communities_dict, graph_obj, mode, itr)
        mle_itr_dict.update({itr: result})

    max_mle = max(mle_itr_dict.values())
    best_itr = [k for k, v in mle_itr_dict.items() if v == max_mle]
    print(f'LOOK␣->␣max_mle:{max_mle}␣happens␣with␣best_itr:{best_itr}')

    mles = np.array(list(mle_itr_dict.values()))
    plt.plot(itrs, mles, '-bo')
    plt.ylabel('log(MLE)')
    plt.xlabel('Number␣of␣iterations')
    plt.title(f'Best␣MLE␣result␣with␣k={k}␣and␣{best_itr[0]}␣iterations')
    plt.title(f'MLE␣results␣for␣different␣iterations␣for␣k:{k}')
    plt.grid()

    return best_itr


# A comparative function; example -> 1000 iterations or 10 simulations with 100 iterations!? 100x10 vs /
    1000
def use_iterations_wisely(vtx, edges, k, graph_obj, mode, base_itr, nbr_simulations):
    print(f"Comparative␣check␣for␣{base_itr}x{nbr_simulations}␣simulations␣vs␣/
        1x{base_itr*nbr_simulations}␣simulation!")

    # CASE 1: nbr_simulations such as 10 with 'base_itr' such as 100
    mle_itr_dict = {}
    for i in range(nbr_simulations):
```

```
        _, k_communities_dict = uneven_split(k, vtx)
        _, result = local_heuristic(vtx, edges, k, k_communities_dict, graph_obj, mode, base_itr)
        mle_itr_dict.update({(i+1): result})

    mle_itr_np = np.array(list(mle_itr_dict.values()))
    plt.plot(list(range(nbr_simulations)), mle_itr_np, '-go')
    plt.title(f'Results for k={k}, {base_itr}x{nbr_simulations} simulations vs /
         1x{base_itr*nbr_simulations} simulation')
    plt.ylabel('log(MLE)')
    plt.xlabel('Simulation numbers')
    plt.grid()

    index_M = np.argmax(mle_itr_np)
    mle_A_M = mle_itr_np[index_M]
    index_m = np.argmin(mle_itr_np)
    mle_A_m = mle_itr_np[index_m]

    # CASE 2: one simulation with 'itrs' iterations such as 1000
    _, k_communities_dict = uneven_split(k, vtx)
    _, mle_B = local_heuristic(vtx, edges, k, k_communities_dict, graph_obj, mode, /
         base_itr*nbr_simulations)
    plt.plot(nbr_simulations-1, mle_B, 'rx', markersize=10)
    print("-------")
    print(f"RESULT:\nWith 1 simulation and {base_itr*nbr_simulations} of iterations:{mle_B}\nWith /
         {nbr_simulations} simulations and {base_itr} of /
         iterations:\nMax:{mle_A_M}\nmin:{mle_A_m}\nDifference:{mle_A_M-mle_A_m}")

    return 'Insight!'


# A function to find necessary iterations and number of runs
def optimized_iteration(k, vtx, edges, graph_obj, mode, default, max_itr, interval, runs):
    itrs = list(range(default, max_itr + interval, interval))
    print(f"Iterations to test:{itrs}")

    mle_itr_dict = {}
    comm_itr_dict = {}
    for itr in itrs:
        mle_list = []
        comm_list = []
        for run in range(runs):
            _, k_communities_dict = uneven_split(k, vtx)
            community, mle = local_heuristic(vtx, edges, k, k_communities_dict, graph_obj, mode, itr)
            mle_list.append(mle)
            comm_list.append(community)
        mle_itr_dict.update({itr: mle_list})
        comm_itr_dict.update({itr: comm_list})

    max_mle_dict = {}
    mean_values = []
    error_range = []
    for key, value in mle_itr_dict.items():
        minimum = min(value)
        maximum = max(value)
        mean = statistics.mean(value)

        max_mle_dict.update({key: [maximum]})
        error_range.append((maximum-minimum)/2)
        mean_values.append(mean)


    max_mle = max(max_mle_dict.values())
    best_key = [k for k, v in max_mle_dict.items() if v == max_mle]
    print(f'Maximum MLE:{max_mle} happens with best_itr:{best_key}')

    target_comm = mle_itr_dict[best_key[0]]
    max_index = target_comm.index(max_mle)

    f = plt.figure(figsize=(9,4))
    mles = np.array(list(mle_itr_dict.values()))
```

```
    ax = f.add_subplot(121)
    ax.plot(itrs, mles, 'o', color='black')
    plt.ylabel('log(MLE)')
    plt.xlabel('Number␣of␣iterations')
    plt.title(f'Best␣MLE␣result␣with␣k={k}␣and␣{best_key[0]}␣iterations')
    plt.title(f'MLE␣results␣for␣{runs}␣runs␣and␣k:{k}')
    plt.grid()

    ax2 = f.add_subplot(122)
    ax2.errorbar(itrs, mean_values, yerr=error_range, ecolor='blue', fmt='kx', markersize= 8, /
        linestyle='--', capsize=4) # uplims=True,lolims=True
    plt.xlabel('Number␣of␣iterations')
    plt.title(f'min-mean-MAX␣MLE␣values␣for␣{runs}␣runs')
    plt.grid()

    return (comm_itr_dict[best_key[0]])[max_index]


start_time = time.time()
result_DC2 = optimized_iteration(optimal_k, vtx, edges, graph_obj, 1, 25, 100, 25, 10)
print(f'Total␣optimized_iteration␣time:{time.time()␣-␣start_time}␣seconds!')


print_my_graph(vtx, edges, result_DC2, False, optimal_k, bipartite, my_graph_size, my_form)


M_DC2 = stochastic_BM(optimal_k, graph_obj, result_DC2)
print_M(M_DC2, 1, my_colors)
```