Imperial College London

Imperial College London
Department of Mathematics

# LARGE SCALE NETWORK LEARNING USING DISCRETE CURVATURE

Junrui Wang
CID: 01490622
Supervised by Prof. Anthea Monod & Prof. Emil Saucan

September 2022

Submitted in part fulfilment of the requirements for
the MSc in Statistics of Imperial College London

# Abstract

Extremely large networks comprising millions of nodes are challenging objects to study, the significant computational cost is a specific challenge that we study in this thesis. Specifically, in this work, we build upon previous work by Sigbeku et al. (2021) [SSM22] which develops a Markov chain Monte Carlo random sampler that learns networks by approximating network statistics using the mathematical concept of discrete curvature. Discrete curvature derives from discrete geometry where one main direction of interest is to adapt classical geometric notions to the computational setting for discrete geometric data structures such as networks. In this thesis, we provide an improvement in computational approaches to the proposal of Sigbeku et al. (2021) [SSM22] which now allows for networks of the order of millions of nodes to be randomly sampled and network statistics to be estimated. Further, we propose a novel algorithm that finds the best sample from a large network using discrete curvature.

## Acknowledgements

I would like to thank my supervisors Prof. Anthea Monod & Prof. Emil Saucan who gave me great support and guidance in making this thesis.

## Declaration

*I, Junrui Wang, declare that the work in this thesis is my own work unless otherwise stated.*

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Motivation and Objectives

With the development of modern data analysis techniques and the growing interest in dealing with large amounts of data, the application of machine learning and AI techniques has been widely applied across different kinds of industries and research. However, some data points appear to be on a network setting or closely related to the other points(nodes), such as the destination points connected by paths. Genomics and Medicine are two areas where Network Analysis was widely applied, as it can solve problems in a graphical setting (as shown in Fig 1.1), thereby revealing the inner relationships of large-scale data ([PNEG17]). Moreover, Network analysis provides a profound contribution to the criminal intelligence analysis by quantified assessment ([Spa91]). Therefore Network analysis becomes a very important research area.

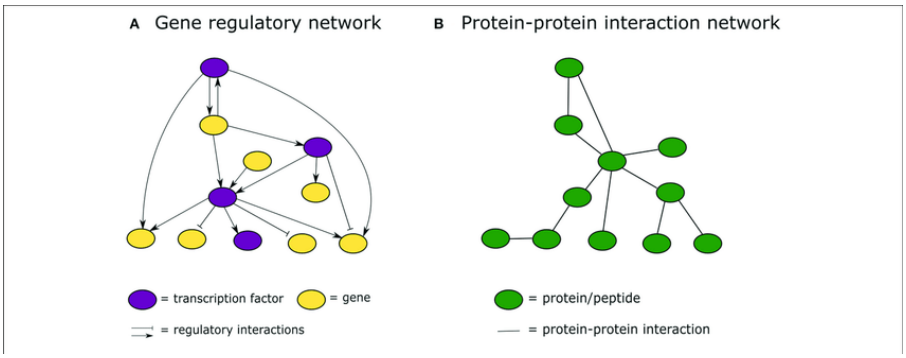


Figure 1.1: Graphical representation of a gene regulatory network & a protein-protein interaction network. (A) Gene regulatory network, the nodes stand for genes or proteins and lines represents the regulatory interactions. (B) Protein-protein interaction network, the nodes represent proteins and the physical protein-protein interactions are represented by the connecting lines ([VVLDCC18]).

In the traditional Network study, *centrality analysis* is applied to examine the importance of a node in the graphical representation setting, where the statistics such as *closeness centrality* and *betweenness centrality* capture the important features of the node. In the Genomics problems such as shown in Fig 1.1, the node represents a gene and the line(edge) might represent the contribution to the same disease, the closeness centrality of a gene would show how close it is to the other genes in the network, and the betweenness centrality measures how much it connects to other groups of genes that contributes to different diseases [PNEG17]. However, the traditional Network study would lead to a high computational cost because we usually use a large network dataset to acquire more accurate analytical results of those statistics and analysis of the entire network becomes almost impossible. Therefore finding a way to focus on the local or a relatively smaller sample size of networks becomes a solution to this problem, and *Network Sampling* is one of the means of constructing smaller-size representations of large networks.

Saucan et al. (2020) [VB20] constructed the theories for edge-based and node-based sampling with discrete curvature for networks and Sigbeku et al.(2021) [SSM22] further adjusted the sampling methods by introducing Markov Process; however, these methods had not been able to be put into a proper model in coding and run on a large-scale network, facing issues such as non-obvious system errors, exceeded RAM, or long-time simulations on a small-scale network (e.g. the Kangaroo Network [Kun]).

In this thesis, we will introduce our python model based on the idea brought forward by Sigbeju et al. (2021). The model is easy and straight to be implemented across different networks and adjusted for different network simulating purposes. Moreover, the model incorporates the existing python packages *Networkx* [HSS08] and developed curvature-based methods and its-relevant criteria. Results showed that a network with millions of nodes could be simulated by our model. Furthermore, we developed an algorithm for finding the best representative sample for the undirected and connected networks.

# Chapter 2

# Background Theory

## 2.1 Basic Definitions

This section will introduce the basic knowledge of network analysis including relevant definitions and notations. These notations will be used throughout this thesis.

**Definition 2.1.** *Network. A network $G = \{V, E\}$ is a collection of nodes and edges, where $V$ denotes the set of nodes and $E$ denotes the set of edges and if an edge in $E$ connects node $i$ to $j$ for $i, j \in V$, then this edge is denoted by $(i, j)$.*

**Remark.** *The network is classified into two categories: directed and undirected network. The network is undirected if the edge connecting node $i$ to $j$ for $i, j \in V$ and $(i, j) \in E$, then $(j, i) \in E$. Alternatively, the network is directed if edge $(i, j) \in E \nRightarrow (j, i) \in E$. In our study, we only focus on* ***UNDIRECTED NETWORK****.*

**Definition 2.2.** *Neighbours. We say a node $j$ is the neighbour of node $i \in V$ if edge $(i, j) \in E$. In words, we also say node $j$ is adjacent to node $j$. Denote the neighbours of node $i$ by $Nb(i)$*

**Remark.** *We denote the edges that start from node $i$ by $e(i)$, i.e. $e(i) = \{(i, j) \in E : j \in V\}$. We also say this $e(i)$ is the set of edges incident to node $i$.*

**Definition 2.3.** *Sample. Given a network $G = V, E$, we say a network $G' = V', E'$ is a sample*

*of the given network $G$ if $V' \subset V$ and $E' \subset E$. We also call this $G'$ as the subgraph of the network $G$.*

**Definition 2.4.** *Adjacency matrix. The adjacency matrix $\mathbf{A} = (A_{ij}) \in \{0,1\}^{|V|\times|V|}$, where*

$$A_{ij} = \begin{cases} 1 & \text{for } (i,j) \in E \\ 0 & \text{otherwise.} \end{cases}$$

**Remark.** *The adjacency matrix for an undirected network is symmetric.*

**Definition 2.5.** *Weight matrix. The weight of an edge $(i,j)$ is a function $w : \mathrm{E} \to \mathrm{R}$. We write $w((i,j))$ as $w_{ij}$. The weight matrix $\mathbf{W} = (w_{ij}) \in \mathrm{R}^{|V|\times|V|}$, where*

$$w_{ij} = \begin{cases} w_{ij} & \text{for } (i,j) \in E \\ 0 & \text{otherwise.} \end{cases}$$

**Remark.** *When $w_{ij} = 1$ for each edge $(i,j) \in E$, the weight matrix is the same as the adjacent matrix, and we call this network as the unweighted network. In our study, we always take $w_{ij} \geq 0$.*

**Definition 2.6.** *Shortest Path. A path in the undirected network is a sequence of nodes $P(v_1, v_n) = (v_1, v_2, \ldots, v_n) \in V \times V \times \cdots \times V$, where $V_i$ is adjacent to $v_{i+1}$ for $1 \leq i < n$. This path $P(v_1, v_n)$ is called a path of length $n$ from $v_1$ to $v_n$. Now assume the path from $v_1$ to $v_n$ exists. Suppose we write $e_{v_i v_j}$ as the edge that connects node $v_i, v_j$, $w$ as the weight function of the edge, $v_1 = i$, $v_n = j$ for some $i, j \in V$. The shortest path from $i$ to $j$ is $P(i,j) = \underset{(v_1,\cdots,v_n)}{\operatorname{argmin}} \sum_{i=1}^{n-1} w\left(e_{v_i v_{t+1}}\right)$.*

**Definition 2.7.** *Distance. Given a shortest path $P(i,j)$, we could have the distance between node $i, j$, which is*

$$dist(i,j) = \min_{(v_1,\cdots,v_n)} \sum_{i=1}^{n-1} w\left(e_{v_i v_{t+1}}\right).$$

**Definition 2.8.** *Connected Network. The network is connected if $\forall i \neq j \in V$, $\exists$ a path $P(i,j)$ that connects $i$ and $j$.*

**Definition 2.9.** *Betweenness centrality (BC). In a connected network, the betweeness centrality*

*of a node $i$ is given by the following mathematical expression:*

$$BC\left(i'\right) = \sum_{i \neq i' \neq j} \frac{\sigma_{ij}\left(i'\right)}{\sigma_{ij}}$$

*, where $\sigma_{ij}$ is the number of the shortest path $P(i,j)$ and $\sigma_{ij}(i')$ is the number of shortest path $P(i,j)$ passing node $i' \in V$.*

**Remark.** *In a connected network, there for every pair of nodes $(i,j)$, there exists at least one shortest path. In an unconnected network, if there is a node $k$ unconnected to any other nodes, we set $BC(k) = 0$. However, it does not affect our study later. This BC statistic is of great importance since it measures how often the node is passed among all shortest paths in a network. Therefore the higher BC values, the more information is captured by this node, and the more important is the node to the entire network.*

**Definition 2.10.** *Closeness Centrality (CC). In a connected network $G = (V, E)$, the closeness centrality of a node $i$ has two mathematical expressions. One is given by Bavelas (1950) [Bav50]:*

$$CC(i) = \frac{1}{\sum_{j \neq i} dist(i,j)}.$$

*Another one normalizes Bavelas' version, given by:*

$$CC(i) = \frac{|V| - 1}{\sum_{j \neq i} dist(i,j)}.$$

**Remark.** *We prefer to use the normalized version CC, as it enables us to compare the CC of nodes in different network sizes.*

**Definition 2.11.** *Degree. The degree of a node $i$ is:*

$$d(i) = \sum_{j=1}^{|V|} A_{ij}$$

*, where $A_{ij}$ is the $ij$th entry of the adjacent matrix $\mathbf{A}$.*

**Definition 2.12.** *Strength. The strength of a node $i$ is:*

$$s(i) = \sum_{j=1}^{|V|} w_{ij}$$

*, where $w_{ij}$ is the $ij$th entry of the weight matrix $\mathbf{W}$.*

## 2.2 Forman-Curvature and Markov Chain Basis

The geometric curvature such as Forman curvature and Haanjes curvature [VB20] captures the importance of nodes in a network. The higher the absolute value of the curvature of a node, the more important the node is to the network. Therefore, using curvature-based sampling methods for networks would optimise the sampling rate of nodes by their importance to the entire network [SSM22].

**Definition 2.13.** *Forman-Curvature for edges. The Forman-Curvature $F$ for edges of a network $G = \{V, E\}$ is a function $F : E \to \mathrm{R}$, defined as:*

$$F((i,j)) = w_{ij}\left\{\frac{w(i)}{w_{ij}} + \frac{w(j)}{w_{ij}} - \sum_{e(i)\sim(i,j)} \frac{w(i)}{\sqrt{w_{ij}w_{e'(i)}}} - \sum_{e(j)\sim(i,j)} \frac{w(j)}{\sqrt{w_{ij}w_{e'(j)}}}\right\} [PW20]$$

*, where $w_{ij}$ represents the weight of edge $(i,j)$, $w(i)$ represents the value assigned to node $i$ ($w(i)$ is called as the 'weight' of the node; in our model, it was chosen to be the degree of the node, i.e. $w(i) = d(i)$), $e(j) \sim (i,j) = \{(i,k) \in e(j) : k \neq j\}$) the set of edges starting from node $i$ excluding the edge $(i,j)$, $e'(i)$ represents an edge starting from node $i$ and this edge is in $e(i) \sim (i,j)$ (the same for $j$).*

**Definition 2.14.** *Forman-Curvature for nodes. For a node $i$, its Forman-Curvature value is a function $F : V \to \mathrm{R}$, defined as:*

$$F(i) = \sum_{(i,k)\in e(i)} F((i,k))[SSM22]$$

**Remark.** *Sometimes we may use the sum of absolute values of Forman-curvature for some*

*specific computational purposes.*

$$F(i) = \sum_{(i,k)\in e(i)} |F((i,k))|$$

**Definition 2.15.** *Markov Process. A discrete-time stochastic process* $X = \{X_n\}_{n\in\mathbb{N}_0}$ *that takes values in a countable state space* $E$ *is a Markov chain (MC) if it satisfies the Markov condition*

$$\mathrm{P}\left(X_n = j \mid X_{n-1} = i, X_{n-2} = x_{n-2}, \ldots, X_0 = x_0\right) = \mathrm{P}\left(X_n = j \mid X_{n-1} = i\right)$$

*for all* $n \in \mathbb{N}$ *and for all* $x_0, \ldots, x_{n-2}$, $i, j \in E$.

**Definition 2.16.** *Time Homogeneous (TH). We say the Markov chain* $\{X_n\}_{n\in\mathbb{N}_0}$ *is time-homogeneous if*

$$\mathrm{P}\left(X_{n+1} = j \mid X_n = i\right) = \mathrm{P}\left(X_1 = j \mid X_0 = i\right)$$

*, where* $\forall n \in \mathbb{N} \cup \{0\}, i, j \in E$.

**Definition 2.17.** *Transition matrix. For a Markov chain* $\{X_n\}_{n\in\mathbb{N}_0}$ *taking values in a finite state space* $E$*, the transition matrix* $x\ \mathbf{P} = (p_{ij})_{i,j\in E}$ *is the* $|E| \times |E|$ *matrix of transition probabilities*

$$p_{ij} = \mathrm{P}\left(X_{n+1} = j \mid X_n = i\right).$$

**Remark.** *In our study, we only consider time-homogeneous Markov Chain (THMC) and write the transition probability that moving to state* $j$ *given the current state* $i$ *as* $(\mathrm{P}\left(X_{n+1} = j \mid X_n = i\right))$ *as* $p_{ij}$.

# Chapter 3

# Model

## 3.1 Motivation

Motivated by the Ricci-curvature sampling method for manifold, the network sampling based on discrete curvatures was studied by Sancan et al.(2020) [VB20] who proposed three types of discrete curvatures: graph Forman-, full Forman- and Haantjes-Ricci curvature. Sancan et al. (2020) emphasized that Ricci curvature behaves as a metric on a metric space, measuring the edge-centrality and analyzing the distribution and behaviour of higher order correlation in networks and this motivated us to further explore and test the application of discrete curvatures, especially the Forman-curvature which has an obvious computational advantage and easy manner in higher dimensional networks [VB20].

Sigbeku et al. (2021) designed a Markov Chain sampling method incorporating 9 Forman-curvature-based samplers (shown in table 3.2) and the applicability of this method was tested. He studied 9 different Forman Samplers [SSM22] over 4 real-world and 2 self-generated networks, showing that the Mean Squared Error for Markov Chain methods with Forman and Uniform samplers all converged, and thereby verifying the applicability of this method theoretically. However, his model constructed in R was not suitable for large-scale networks which might have millions of data. There are 3 main big drawbacks to his model. The first one is the way he stored the data which were stored in a $M \times M$ matrix, where $M$ is the number of nodes

in the network. Rick[Wic] stated that a $M \times M$ matrix requires $M^2 8/10^9$ gigabytes of RAM. If $M = 100,000$, the matrix would take up 2 GB of RAM but a normal computer has RAM 4 or 8 GB, which would cause an error: `Message Error: Unable to allocate sufficient memory`. Secondly, his model had a heavy computational cost. For a small network such as the Kangaroo network with node-size of 17, running 300 samples would take a few hours. Thirdly, there was no clear sign for his simulation to start or end. Due to his heavy computation cost and some non-obvious bugs in his model, sometimes it might take hours but the output was an empty set or the system might produce error alarms after running for a couple of minutes.

Therefore, based on the theories from Saucan et al. (2020) [VB20] and the Markov Chain sampling method from Sigbeku et al.(2021) [SSM22], we constructed a python model for analyzing large-scale networks. In this model, 6 different Forman-curvature-based sampling methods and a uniform sampling method have been added. However, this model could not only be used to generate samples based on the Forman-curvature method; but could also be generalized to test all proposed geometric curvatures, such as Haanjtes curvature. Our python model solves the 3 big issues above and the MCMC sampling algorithm and Best-Sample-Selection algorithm were further developed in our model.

This is a very powerful model, as it has a lot of potential to be adapted to any other Network analysis. The way to adjust the model is very simple and straightforward, as the basic knowledge of python was applied such as `class, function, NumPy, pandas, list, dictionary`. In the next section, we will give a detailed explanation of our model such as what is meant by each function constructed and the theories applied. Then, show the performance of the MC-sample-generating algorithm in our model over 7 different real networks and 2 randomly generated networks; and then introduce our own novel algorithm to find the best sample for a given network. Lastly, we will also explain how to adapt the model and the limitations of its usage.

## 3.2 Model Description

The python code is in the appendix A. Referring to the code, let us explain our model:

`class Networks:`

We construct our model by `class` in python and name our model `Networks`.

`def __init__(self,G):`

Model initialization. The model takes an input `G`. Note that the input `G` need to be a connected graph(network) which is generated by using the package `networkx` [HSSC08]. Each edge in `G` requires at least a keyword `weight` and the value of it should not be `None`, where the spelling of `weight` is restricted and other spellings like `weights` would cause errors in the following steps. In the initialization, we set values:

| Variable Name | Variable Meaning |
|---|---|
| `totNodes` | The total number of nodes in `G`. |
| `node_list` | The list of entire nodes in `G`. |
| `Sample_list` | The list of samples generated by MC-Sampling algorithm. |
| `Sample_Index` | The number of samples generated by MC-Sampling algorithm. |
| `Sample_prob_val` | The dictionary that stores the calculated transition probabilities by some given methods. |

Table 3.1: Model Variables.

**Remark.** *We will explain the reasons to set variables* ***`Sample_list`****,* ***`Sample_Index`*** *and* ***`Sample_prob_val`*** *in the following subsection.*

`def __total_nodes__(self):`

Return the value of total number of nodes in the given network `G`.

`def vec_Forman_edge_curvature_1(self,node_i,node_j):`

Compute the Forman-curvature(FC) value for the edge joining `node_i,j` according to the following adjusted FC equation defined in the previous chapter, and return it.

$$F((i,j)) = w_{ij}\left\{\frac{w(i)+w(j)}{w_{ij}+0.000001} - \sum_{e'(i)\in e(i)\sim(i,j)}\frac{w(i)}{\sqrt{w_{ij}w_{e'(i)}}} - \sum_{e'(j)\in e(j)\sim(i,j)}\frac{w(j)}{\sqrt{w_{ij}w_{e'(j)}}}\right\}$$

Note that to avoid the issue caused by dividing 0, we add 0.000001 to the first denominator.

**Remark.** *The reason why we do not add 0.000001 to the rest two denominators is that our calculating algorithm for the rest two terms avoids this issue, seeing the code. Also, the type of output is scalar.*

`def vec_Forman_node_curvature_1(self,node_i):`

Return the FC-node value for the node $i$ by the equation

$$F(i) = \sum_{(i,k)\in e(i)} F((i,k))[\text{SSM22}].$$

`def vec_Forman_abs_node_curvature_1(self,node_i):`

Return the FC-node value for the node $i$ by the equation

$$F(i) = \sum_{(i,k)\in e(i)} |F((i,k))|.$$

Then let us introduce our two big contributions to the complex network analysis: 1. Refined Markov-Chain Sampling Algorithm; 2. Best-Sample-Selecting Algorithm. Let us describe them one by one.

### 3.2.1 Refined Markov-Chain Sampling Algorithm (RMCS)

Assume we have a network $G = \{V, E\}$ with the total number of nodes $|V| = M$. If we have the Forman-curvature $F : E \to \mathrm{R}$, we can define a scale function $H$ such that $H \circ F : E \to (0, 1)$, so that $H \circ F$ is a probability measure for $E$. Thus, we can write the transition probability $p_{ij} = H \circ F((i, j))$. Referring to Sigbeku (2021)'s proposal, we have 7 different sampling methods.

| Function Name (long) | Formulae |
|---|---|
| Relative Forman Curvature (RFC) | $p_{ij} \propto \lvert F((i,j))\rvert$ |
| RFC Degree Divided | $p_{ij} \propto \frac{\lvert F((i,j))\rvert}{d(j)}$ |
| RFC with CC | $p_{ij} \propto \alpha_j \lvert F((i,j))\rvert$ |
| RFC with CC Degree Divided | $p_{ij} \propto \frac{\alpha_j \lvert F((i,j))\rvert}{d(j)}$ |
| RFC Comms | $p_{ij} \propto \alpha_j \lvert F((i,j))\rvert + (1-\alpha_j)\sum_{e(j)\sim(i,j)} \lvert F((j,k))\rvert$ |
| RFC Degree Divided | $p_{ij} \propto \frac{\alpha_j \lvert F((i,j))\rvert}{d(j)} + (1-\alpha_j)\sum_{e(j)\sim(i,j)} \frac{\lvert F((j,k))\rvert}{d(j)}$ |
| Uniform Sampling | $p_{ij} \propto \frac{1}{d(i)}$ |

Table 3.2: Sigbeku et al. (2021) proposed methods [SSM22].

**Remark.** *Sigbeku et al. (2021) also proposed another 3 methods: Relative Forman Curvature with Haantjes, Relative Forman Curvature with Haantjes Degree Divided, and Relative Forman Curvature with Assortativity Coefficient, but their formula repeated the formula listed above, so we just ignore them.*

*Moreover, the python functions* `vec_RFC_...`, `vec_uniform` *(Appendix A) calculated the unscaled $p_{ij}$ (Right side of $\sim$ in the second column of table 3.2) for corresponding method, where the name of the python functions follows this pattern*

`vec_` + `"(Initials of) parts of method's name"`

*. Because these functions all return an FC-adjusted value for the edge $(i,j)$ (in other words, we try to analyze the FC-adjusted value pairs by pairs, vectors by vectors), we suggest adding a prefix* `vec_` *to distinguish from the matrix-construction method.*

Then, choosing an initial node $X_0 = k$ for $k \in V$, we could generate a Markov Chain (MC) $\{X_i\}_{i=0}^{M'}$, where $M'$ is the number of the steps, $X_i$ takes value in $V$. The transition probabilities $p_{ij}$ is calculated by:

1. Calculate the absolute FC-values for all edges $(i,k) \in e(i)$;

2. Scale the FC-value for the edge $(i,k)$ by dividing the sum of all FC-values for the edge in $e(i)$. The mathematical expression is:

$$p_{ij} = \frac{|F((i,j))|}{\sum_{(i,k)\in e(i)} |F((i,k))|}.$$

**Remark.** *1. The denominator is exact my suand ggested definition for FC-value for a node. (see the remark for Definition 2.14).*

*2. The transition probabilities for a node $i$ is $(p_{ij})_{j\in V}$. 3. In the python code, the function* `to_vec_prob` *calculates the $p_{ij}$.*

Say $\{x_i\}_{i=1}^{M'}$ is a realization of the MC, then the set $V' = \{i : i \in \{x_i\}_{i=1}^{M'}\} \subset V$. Taking a subset $E'$ of $E$, where $E'$ only takes the edges connecting the nodes in $V'$, we have a sample $G' = \{V', G'\}$ of the original network $G$.

However, in the MC process $\{X_i\}_{i=0}^{M'}$, for some $i, j = 0, 1, 2, ..., M'$, $X_i, X_j$ might takes the same value $i$ in $E$, i.e the node $i$ appears more than once in this MC process and the total number of nodes in the sample (denote it by $ttn$) is less than $M'$. Thus, if we want to get a specific total number of nodes in a sample, we could keep running the MC process until we get $ttn = M'$. Note that we expect $M' < M$, so we usually take $M' = int(r * M)$ for some $r$ s.t. $0 < r < 1$.

Therefore, we have two scenarios: 1. Run $M'$ iterations and then stop; 2. Keep running until we have $ttn = M'$. `for` loop could be applied to Scenario 1 and `while` loop could be applied to Scenario 2. But running within a `for` loop is faster than a `while` loop if the iterations are the same (details seeing from `https://www.l3harrisgeospatial.com/Learn/Blogs/Blog-Details/ArtMID/10198/ArticleID/15332/What-Type-of-Loop-Should-I-Use`). Thus, we prefer Scenario 1 than 2, but there is an option to choose which scenario to use.

However, during practice, we find calculating transition probabilities is very time-consuming (In a large-scale network–youtube-link network[Kun], calculating the transition probabilities for Node 5517 takes about 1 min.), so we further refined the algorithm: given a sampling method (Table 3.2), we store the transition probability for a node $i$ in the dictionary variable `Sample_prob_val` (Table 3.1) with key equal to the name of the sampling method so that if we need the transition probability for node $i$ using the same method again, we do not need to re-calculate but directly have it from `Sample_prob_val`.

`def MCMC_Sample_generator(self, Initial_state, vec_func = None, alpha_j = None, proportion = 0.5, iters = False):`

This is the function to apply the RMCS algorithm in python.
`Initial_state`: the initial position for the MC to start. Default `vec_func = None` controls the sampling method. The Default sampling method for the MC-Sampling algorithm is uniform sampling. It can be changed to other methods by setting `vec_func`)
Default `iters = False`.`iters` controls which scenario to use. The default setting is using Scenario 1; if `iters = True`, the MC-Sampling algorithm is set to Scenario 2.
Default `alpha_j = None`. `alpha_j` is for some methods like `vec_RFC_cc` which requires the input value `alpha_j`. The default setting is for the method without the input `alpha_j`.
Default `proportion = 0.5`. `proportion` controls the number of nodes of the sample to generate and this number is not bigger than $0.5M$ ($M$ the total number of nodes of the original network G). Output: one sample (generated) of the given network $G$.
Note: every time running this function, the generated sample would be saved in the variable `Sample_list` and `Sample_Index` + 1 (see table 3.1).

To specify, we describe the basic MC mathematical logic in the following algorithm table but for the complicated code-achievement, please refer to Appendix A.

**Remark.** *1. For each RMCS simulation, it is better to randomly choose an initial position for it. 2. In python code, the function* ***`random_node_generator(self,size = 1,replace = True)`*** *helps to randomly choose an initial position ('randomly choose' means 'choose a node by uniform distribution'). Multiple initials could be chosen at once by changing* ***`size = N`****, where $N$ is the number of samples.*

### 3.2.2 Best-Sample-Selecting Algorithm (BSS)

Since the RMCS algorithm just generates a sample but does not guarantee the sample generated is the one possessing the main topological features of the original network, we designed an algorithm by using the RMCS.

**Algorithm 1:** Refined MC-Sampling Algorithm Scenario 1

**Data:** $G = \{V, E\}$.
Input: Initial position $i_0 \in V$; proportion of nodes $r$, $0 < r < 1$.
Set: $x_0 = i_0$ , $M' = \texttt{int}(rM)$.
**for** $i = 1, 2, ..., M'$ **do**
  **if** $(p_{i_0 j})_{j \in e(i_0)}$ *has not been calculated* **then**
    Calculate $(p_{i_0 j})_{j \in Nb(i)}$ by

$$p_{i_0 j} = \frac{|F((i_0, j))|}{\sum_{(i_0,k) \in e(i_0)} |F((i_0, k))|}, \forall j \in Nb(i)$$

  **end**
  Choose $x_i$ by $X_i = x_i \sim Multinominal(n = Nb(i), p = (p_{i_0 j})_{j \in e(i_0)})$.
  Set $i_0 = x_i$.
**end**
Output: $(x_i)_{i=0}^{M'}$.

**Algorithm 2:** Refined MC-Sampling Algorithm Scenario 2

**Data:** $G = \{V, E\}$.
Input: Initial position $i_0 \in V$; proportion of nodes $r$, $0 < r < 1$.
Set: $x_0 = i_0$ , $M' = \texttt{int}(rM)$, $\texttt{sample} = [i_0]$.
**while** *length of sample* $< M'$ **do**
  **if** $(p_{i_0 j})_{j \in e(i_0)}$ *has not been calculated* **then**
    Calculate $(p_{i_0 j})_{j \in Nb(i)}$ by

$$p_{i_0 j} = \frac{|F((i_0, j))|}{\sum_{(i_0,k) \in e(i_0)} |F((i_0, k))|}, \forall j \in Nb(i)$$

  **end**
  Choose $x_1$ by $X_1 = x_1 \sim Multinominal(n = Nb(i), p = (p_{i_0 j})_{j \in e(i_0)})$.
  Set $i_0 = x_1$.
  `sample.append(`$x_1$`)`
**end**
Output: $(x_i)_{i=0}^{M'}$.

**Algorithm 3:** Best-Sample-Selecting Algorithm

**Data:** $G = \{V, E\}, |V| = M$.
Input: $0 < \beta < 1$; the number of samples $N$.
Generate $N$ samples by the RMCS algorithm.
Calculate the total number of nodes for each sample, denoted by $M'_i, i = 1, 2, ..., N$.
Set $M''_N = \min_{i=1,2,...,N}(M'_i)$.
Select the nodes that occurred at least $\beta N$ times among the generated samples, i.e.

$$\texttt{selected\_nodes} = \{i \in V : \text{Occurrence of node } i > \beta N\}.$$

Output: `selected_nodes`.

**Remark.** *We could control the number of selected nodes by values of $N$ and $\beta$, thereby affecting the accuracy of the selection. Since, if the node is uniformly sampled, then the probability of a node $i$ that is selected in a sample is at least $M''_N/M$. Then the expected number of occurrences of node $i$ is $N * M''_N/M$. If node $i$ is important to the network, then its absolute Forman-curvature value is high, then its occurrence is expected to be bigger than $N * M''_N/M$; otherwise, its occurrence is expected to be smaller than $N * M''_N/M$. The more samples we get (the larger $N$), the more accurate we could get.*

*Thus, we usually choose $\beta = M''_N/M$, but the accuracy of selecting a relatively bigger $\beta$ would be affected by changing $\beta$.*

```
def Best_sampler(self, samples = None, vec_fun, alpha_fac = None, N=100):
```

This is the function to apply the BSS algorithm in python code.
`samples`: whether to run the RMCS algorithm within this function. Its Default value is `None`. If `sample` is `None`, then the RMCS algorithm was implemented first to generate samples. If not, it should take the values of the nodes of samples generated by the RMCS algorithm and the RMCS algorithm would not be run within the function.
`vec_fun`: the sampling method to apply when `sample` is `None`. It is currently only designed for methods without the `alpha_j` argument.
`alpha_fac`: adjust the $\beta$ values. Its Default value is `None`. If `alpha_fac` is `None`, then $\beta = \beta$. If `alpha_fac` $= a$ for some numeric value $a$, then $\beta = a * \beta$. Otherwise, the system produces warning errors.
`N`: `N` represents the number of samples to run, when `sample` is `None`. Its Default value is 100.

## 3.3 Model Analysis

Our aims of the Model-Analysis section are two:

- Test the speed of generating a sample by RMCS Algorithm with Forman-Curvature based-methods;

- Test the accuracy of the BSS Algorithm.

### 3.3.1 Analysis for RMCS Algorithm

The RMCS Algorithm was tested on 6 real-world networks, the animal social interaction networks Kangaroo and Dolphin, sports interaction American football, the character relational network Les Miserables (Lesmis), the European road network Eurorad, the social network of Youtube users and their connections Youtube_link [Kun], and 2 self-generated data $G_{1,2}$ (listed in the table below 3.3). The sampling method 'Relative Forman Curvature' (RFC) was chosen for the test among the Forman-curvature-based methods, because the sampling speed of the Forman-curvature-based methods mainly depends on the calculation speed of $|F((i,j))|$ (see table 3.2). Then, we showed the efficiency of our sampling methods by comparing the RFC and uniform sampling methods. Lastly, our results were compared with the results from Sigbeku et al. (2021) [SSM22].

| Networks Name | $(\|V\|, \|E\|)$ |
|---|---|
| Dolphins (dp) | $(62, 158)$ |
| Lesmis | $(77, 258)$ |
| Kangaroo | $(17, 91)$ |
| Dimacs10-football (ft) | $(115, 612)$ |
| Subelj_euroroad (er) | $(1171, 1417)$ |
| Youtube_link | $(1138499, 4942297)$ |
| $G_1$ | $(300, \binom{300}{2} = 44850)$ |
| $G_2$ | $(1000, \binom{1000}{2} = 499500)$ |

Table 3.3: Test Networks with their corresponding number of nodes $|V|$ and the number of edges $|E|$. $G_{1,2}$ were randomly generated with nodes pair-wisely connected and weights $w_{ij} \sim Multinominal(0, 1, 2, ..., 10,\ p = 1/11)$.

We do the analysis in 2 parts: 1. Based on the relatively small size networks: the first 5 networks in Table 3.3; 2. Based on the rest 3 large networks.

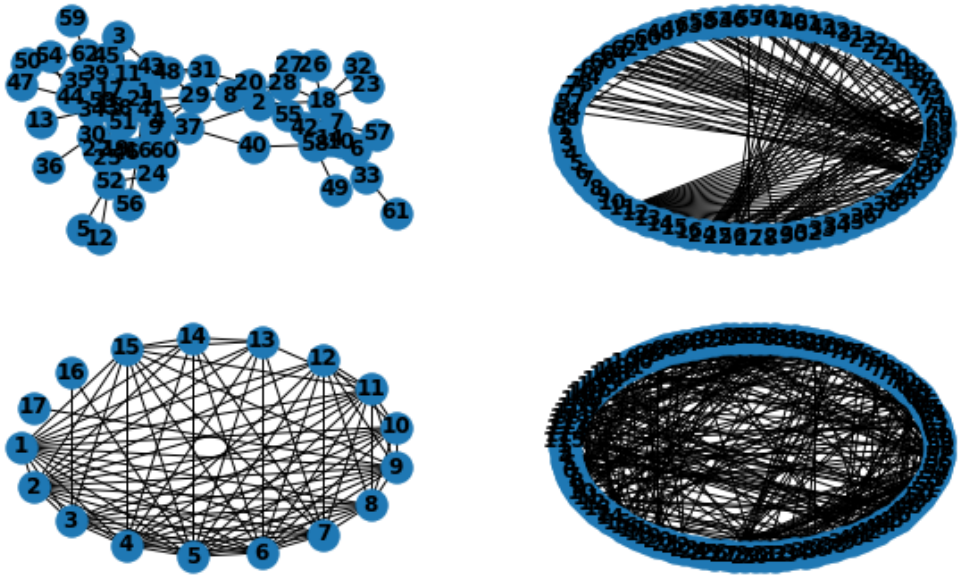

Figure 3.1: Networks graph for Dolphins, Lesmis, Kangaroo, ft (reading order: from Left to right, top to bottom). Due to the drawing limitation, networks with node number> 1000 were not drawn.

**Part 1**

Take the proportion of nodes in a sample to be 80% and use RMCS-Scenario-1. We calculate the average speed for generating one sample as follows:

Say $N$ is the number of samples, $T$ is the total time taken to generate $N$ samples, then the average time taken $\bar{t}$ is:

$$\bar{t} = \frac{T}{N}.$$

Take $N = 1000$ for the first 5 networks in Table 3.3, we can see from the table 3.4 that $\bar{t} < 0.12s$ for the regular-size networks.

**Part 2**

Since the size of the rest networks was big and the nodes of $G_{1,2}$ were pair-wisely connected, we take $N = 5$ and compare the performance between the uniform and RFC sampling methods.

| Networks Name | $\bar{t}$ (to 3 decimal) |
|---|---|
| Dolphins (dp) | 0.012s |
| Lesmis | 0.053s |
| Kangaroo | 0.020s |
| American football (ft) | 0.101s |
| Euroroad (er) | 0.117s |

Table 3.4: Average time taken by the RFC method for the regular-size networks.

For the youtube-link network: With proportion = 0.01, uniform sampling methods took about 40s on average; with proportion = 0.1, uniform sampling took about 923s (15 mins). With proportion = 0.01,. RFC sampling took about 236 mins (4hs or so).

For $G_1$: with proportion = 0.1, RFC sampling took about 27s.
For $G_2$: with proportion = 0.1. RFC sampling took about 22mins 26s while uniform sampling took about 0.3s.

From the above analysis, we can draw comments on the speed of the model:

- Time taken by Uniform sampling showed that the RMCS algorithm in our model is very fast. Compared with the model from Sigbeku et al. (2021) [SSM22] on lesmis network, keeping the rest arguments same, his model took about 10s by RFC methods, while ours was only 0.053s, which was about 200 times faster.

- For a large-scale network with a size of millions, it is better to process it on GPU than CPU.

- We also tried multiprocessing in Windows 11 system, but the time taken by multi-processing is much slower than using a comprehension method (or for loop) in python. For multi-processing, generating $N = 10$ samples of the Dolphins network took about 6.3s, while comprehension took about 0.2s.

- The reason for the difference in time-elapsing between RFC and uniform method is due to the complexity of Forman-Curvature. Suppose node $i$ has $n_i$ neighbours, i.e. $|e(i)| = n_i$, where node $j$ is one of them and has $n_j$ neighbours, the number of necessary iterations to calculate the FC values is $n_i + n_j$; to calculate the transition probabilities $(p_{ij})_{j\in e(i)}$

of node $i$, it requires another $n_i$ iterations. Together, the RFC-method requires at least $O(n_i(n_i + \min_{j \in e(i)} n_j))$ iterations, but uniform sampling only need $n_i$ iterations. This caused calculating speed of the RFC-method is $O(n_i(n_i + \min_{j \in e(i)} n_j))$ times slower than the uniform sampling for every node $i$ .

### 3.3.2 Analysis for BSS Algorithm

**Default settings were applied unless otherwise stated.*

Sigbeku et al. (2021) [SSM22] found that RFC_dd sampling method was the most efficient among the Forman Samplers in terms of the MSE convergence to the global and local network properties, but RFC was the least efficient. Therefore, we tested both two methods in our python model.

Take one example from the samples of each network in Part 1, we shall see Fig 3.6.

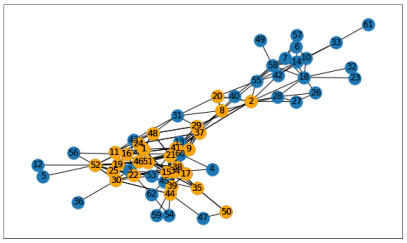

Figure 3.2: Dolphins Sample.

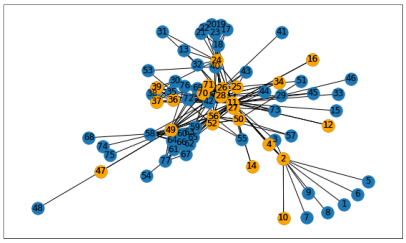

Figure 3.3: Lesmis Sample.

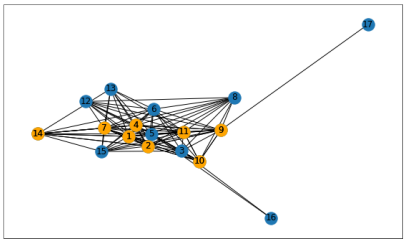

Figure 3.4: Kangaroo Sample.

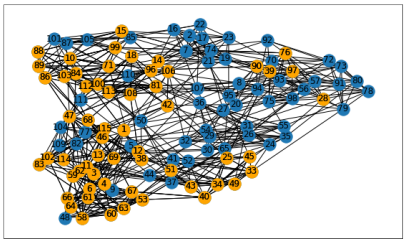

Figure 3.5: Football Sample.

Figure 3.6: Samples generated by RMCS Algorithm with RFC method.

But from Figures 3.6, we found that even though the RMSC algorithm captures the main clusters of the network, it misses the bottleneck which is important for betweenness approximation. Thus, the BSS algorithm helps to solve this issue. By generating 4222, 2002, 2002, and 1000 samples for network

Dolphins, Lesmis, Euroroad and Football by RFC-method, respectively, we got the best samples. Then, generate 10, 1000, and 4222 samples for Dolphins by RFC_dd-method and get the best sample for it. Here we draw the best samples by the BSS algorithm for network Dolphins, Lesmis and Football in the following.

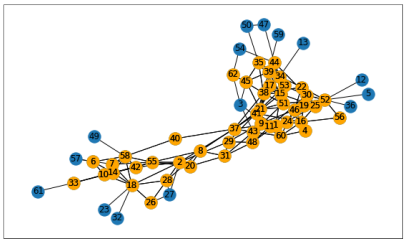

Figure 3.7: Dolphins Best Sample by RFC-method.

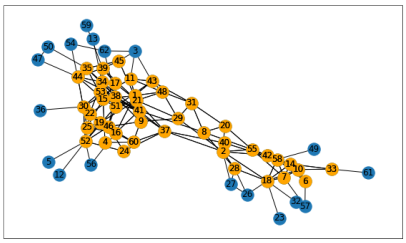

Figure 3.8: Dolphins Best Sample by by Relative Forman Curvature Degree Divided(RFC_dd) method with $N = 4222$.

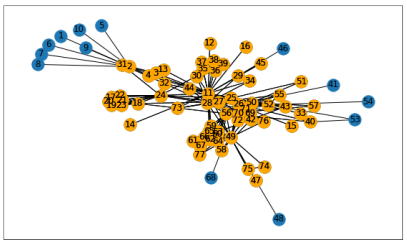

Figure 3.9: Lesmis Best Sample by RFC method.

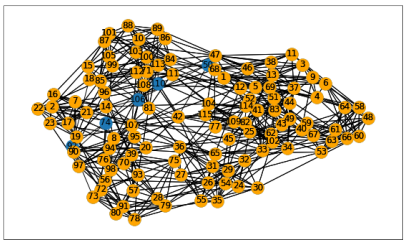

Figure 3.10: Footbal Best Sample by RFC method.

From Fig 3.7, 3.8, 3.9, 3.10, they looked like to capture all the important features of networks. Then we calculate the difference in percentage in terms of betweenness and closeness centrality, defined as:

$$D_k = 100 \times \frac{|k(S) - k(G)|}{k(G)}, k = (b, c)$$

, where $b$ represents betweenness centrality, $c$ represents closeness centrality, $D_k$ represents the difference in percentage for the corresponding centrality, $G$ represents the original network, $S$ represents the best sample, $k(.)$ represents the corresponding centrality of the network.

| Networks with Method | $(D_b, D_c)/\%$ (to 2 decimal) | Time taken/s (to 1 decimal) |
|---|---|---|
| Dp with RFC_dd ,$N = 10$ | (34.92, 7.14) | 0.2s |
| Dp with RFC_dd, $N = 1000$ | (10.47, 20.09) | 0.2s |
| Dp with RFC_dd, $N = 4222$ | (8.76, 19.12) | 0.6s |
| Dp with RFC | (5.20, 16.44) | 0.3s |
| Lesmis with RFC | (4.44, 12.05) | 0.2s |
| Ft with RFC | (5.05, 0.23) | 0.8s |
| Euroroad with RFC | (5.021, 3.28) | 66.4s |

Table 3.5: Percentage difference and time taken for the corresponding networks and methods.

Thus, referring to Table 3.5 and the BSS algorithm, we might conclude that :

- FC Sampling methods had some differences but it does not affect too much about $D_k, k = b, c$. Even RFC is better than RFC_dd in our cases as its $D_k, k = b, c$ were smaller.
- Our BSS algorithm is fast enough and accurate enough to be applied as a proper algorithm for generating a sample even for a network with a large number of nodes.
- We may adjust the accuracy by adjusting $\beta$.
- It is not good enough to generate just one sample by RMCS algorithm, since the proportion of nodes of a sample is hard to be decided, but this sample shall give an initial guess for the best sample.
- The more samples generated, the more confident in the best sample chosen by the BSS algorithm.
- In python-code, the function `betweenness_dif_percent` and `closeness_dif_percent` is defined to use and the function `sub_graph` is for generating a subgraph in the model which used the in-build function `subgraph` from `networkx` [HSSC08].
- An interesting thing to notice is that for the Dolphins network, with the number of samples $N = 4222$, we found that even though Sigbeku et al. (2021) [SSM22] showed that the method RFC_dd would converge faster to global and local network properties in his model, the calculating time was faster for RFC and the percentage difference was also smaller for RFC for both statistics $D_{b,c}$.

## 3.4 Model Adjustment Guidelines

To use the model, please download it via the GitHub link `https://github.com/JericRui/Network_Analysis_MSc` and save the file on the local computer and in the python working directory. Suppose the file name saved is `ImportedNetworks.py`, please import the model via the code:

```
from ImportedNetworks import Networks
```

Then, use all the things inside the model would be used.

It would be interesting to add some extra curvature-based methods to our model and the model could be adjusted according to the following:

- Write the wanted function as `vec_FunctionName(self, node_i, node_j, possible_arguments to be added)`.
- If there were other possible arguments, please remember to add the possible arguments in the functions `to_vec_prob`, `MCMC_Sample_generator`. For example, change function `to_vec_prob` by `def to_vec_prob(self, node_i, vec_func, alpha_j = None, possible_arguments):`.

# Chapter 4

# Conclusion

## 4.1 Summary of Thesis Achievements

In summary, we made two main contributions to the Complex Network Analysis:

- Refined the Markov Chain Sampling algorithm by geometric tools–Forman Curvature and enhanced the sampling-generating speed by 200 times compared with the current MC-Sampling method. It brings the Markov Chain theory into practice and makes the algorithm suitable for real-world network analysis.
- Designed a new algorithm to generate the best sample of a given network and the algorithm is fast, flexible and accurate enough to be applied to all possible undirected, completely connected networks.

In addition, the following contribution is that we constructed a python model, which could be therefore used as a package in python to simulate more networks and is flexible enough to be adjusted for all other possible network simulations.

One thing worth being notified of is that it is better to have a super-computer or GPU to perform RMCS and BSS simulations for a large-scale network. The common computer with 8 or 12 cores could easily be used to deal with a network with less than a hundred thousand of edges but a bit time-consuming to deal with a network with edges more than that.

## 4.2 Applications

The BSS algorithm provides us with a way to generate the best sample of a given network, thus it could be extended to many other areas related to the complex network. For a people-relationship network, we may predict the most important people to a person on his/her social networking map. For the brain-neuro study, we can classify the most related neurons among a neuro network with a large number of neurons. For military battles, we could simulate the most possible locations of enemies according to the network map of the enemies' route of march. But currently, these are just at a hypothetical stage; it would be interesting to extend our model into many other fields in the future.

## 4.3 Future Work

We would like to suggest the following works in the future:

- Model update. There are several parts to improve. 1. In python, if a function `func` uses another function, the input arguments of `func` could be written as `func(..., *arg)` so that we do not need to change the functions in the ways stated in the section 3.4. But due to time limitations, the brute definition constructions were applied. 2. The variable `Sample_list`, `Sample_Index` were not related to the sampling method. To save the generated samples according to their corresponding sampling method is a good way to mimic the way we did for the variable `Sample_prob_val`. The ultimate goal is to develop our model into a formal python package; possibly collaborating with Networkx [HSSC08] would be even better.

- New curvatures. There are many other possible geometric tools to use, such as Haantjes curvature [VB20]. It is suggested to just put it into our model and test its performance.

- Apply the model in a practical field. Such as biology, and social network, there are a lot of network-related fields to be applied. The best sample could therefore be generated and test the model performance.

# Appendix A

# Python Model Code

This is the python-code for our Network model. If there is anything unclear, it is helpful to visit `https://github.com/JericRui/Network_Analysis_MSc` to find the entire code. To test the model, please import this model into the local environment and follow the instructions in the chapter **Model**, section 3.4.

```
import numpy as np
import time
import networkx as nx
class Networks:
    def __init__(self,G):
        self.G = G
        self.totNodes = len(list(G.nodes))
        self.node_list = list(set(G.nodes))
        self.Sample_list = []
        self.Sample_Index = 0
        self.Sample_prob_val= dict()
    def __total_nodes__(self):
        return self.totNodes

    def vec_Forman_edge_curvature_1(self,node_i,node_j):
    #incident_sum of node i
        wt_ij = self.G[node_i][node_j]['weight']
        incident_sum_11 = 0
        for nbr_k in self.G[node_i]:
            if nbr_k != node_j:
                wt_k = self.G[node_i][nbr_k]['weight']
                if  wt_ij*wt_k != 0:
                    incident_sum_11 += self.G.degree(node_i)/np.sqrt((wt_ij*wt_k))
```

```
        #incident sum of node_j
        incident_sum_12 = 0
        for nbr_k in self.G[node_j]:
            if nbr_k != node_i:
                wt_k = self.G[node_j][nbr_k]['weight']
                if  wt_ij*wt_k != 0:
                    incident_sum_12 += self.G.degree(node_j)/np.sqrt((wt_ij*wt_k))
        return wt_ij*((self.G.degree(node_i)+self.G.degree(node_j))/(wt_ij+0.000001)
        - incident_sum_11-incident_sum_12 )

    def vec_Forman_node_curvature_1(self,node_i):
        tmp = [self.vec_Forman_edge_curvature_1(node_i,node_j) for node_j in
        list(self.G[node_i].keys())]
        return sum(tmp)

    def vec_Forman_abs_node_curvature_1(self,node_i):
        tmp = [np.abs(self.vec_Forman_edge_curvature_1(node_i,node_j)) for node_j in
        list(self.G[node_i].keys())]
        return sum(tmp)


    def vec_rfc(self,node_i,node_j):
        # relative_forman_curvature
        unscaled_p = np.abs(self.vec_Forman_edge_curvature_1(node_i,node_j))
        return unscaled_p

    def vec_rfc_dd(self,node_i,node_j):
        # relative_forman_curvature_Degree_Divided
        unscaled_p = np.abs(self.vec_Forman_edge_curvature_1(node_i,node_j))/self.G.degree(node_i)
        return unscaled_p

    def vec_rfc_cc(self,node_i,node_j,alpha_j):
        # relative_fc_with_cc
        unscaled_p = alpha_j *np.abs(self.vec_Forman_edge_curvature_1(node_i,node_j))
        return unscaled_p

    def vec_rfc_cc_dd(self,node_i,node_j,alpha_j):
        # relative_fc_with_cc_Degree_Divided/Haantjes/Assortativity Coefficient
        unscaled_p = alpha_j *np.abs(self.vec_Forman_edge_curvature_1(node_i,node_j))
        /self.G.degree(node_i)
        return unscaled_p
    def vec_rfc_comms(self,node_i,node_j,alpha_j):
        # relative_forman_curvature_comms

        unscaled_p = alpha_j *np.abs(self.vec_Forman_edge_curvature_1(node_i,node_j))
        + (1-alpha_j)*(self.vec_Forman_abs_node_curvature_1(node_j)-
        self.vec_Forman_edge_curvature_1(node_i,node_j))
        return unscaled_p
```

```
72
73      def vec_rfc_comms_dd(self,node_i,node_j,alpha_j):
74          # relative_forman_curvature_comms
75          unscaled_p = alpha_j *np.abs(self.vec_Forman_edge_curvature_1(node_i,node_j))
76          /self.G.degree(node_i)
77          + (1-alpha_j)*(self.vec_Forman_abs_node_curvature_1(node_j)-
78          self.vec_Forman_edge_curvature_1(node_i,node_j))/self.G.degree(node_j)
79          return unscaled_p
80
81      def vec_uniform(self,node_i,Node_j):
82          return 1
83
84      def random_node_generator(self,size = 1,replace = True):
85          return np.random.choice(self.node_list, size = size, replace=replace)
86
87      def vec_scale(self,vec):
88          if type(vec).__module__ == np.__name__ :
89              tt = vec.sum()
90              return vec/tt
91          else:
92              tt = sum(vec)
93              return [i/tt for i in vec]
94
95      def to_vec_prob(self, node_i, vec_func, alpha_j = None):
96          if not(alpha_j is None):
97              tmp = np.array([vec_func(node_i,node_j,alpha_j) for node_j in
98              list(self.G[node_i].keys())])
99              return self.vec_scale(tmp)
100         else:
101             tmp = [vec_func(node_i,node_j) for node_j in list(self.G[node_i].keys())]
102             return self.vec_scale(tmp)
103
104
105     def MCMC_Sample_generator(self, Initial_state, vec_func = None ,alpha_j = None,
106     proportion = 0.5, iters = False):
107         if vec_func is None:
108             vec_func = self.vec_uniform
109             #Default values is rfc_dd
110         proportion_of_nodes = np.floor(proportion * self.totNodes) # Size of subgraph
111         print('Session Start....')
112         t_start = time.perf_counter()
113         Current_state = Initial_state
114         self.Sample_list.append([Current_state])
115         self.Sample_prob_val[f'{vec_func.__name__}'] = {Current_state:None}
116         # iteration = 0
117         if iters:
118             while (len(self.Sample_list[self.Sample_Index]) <= proportion_of_nodes):
119
```

```
                #check whether saved or not
                if self.Sample_prob_val[f'{vec_func.__name__}'].get(Current_state) is None:
                    probs = self.to_vec_prob(node_i=Current_state,
                    vec_func = vec_func,alpha_j = alpha_j)
                    self.Sample_prob_val[f'{vec_func.__name__}'][Current_state] = probs
                probs =  self.Sample_prob_val[f'{vec_func.__name__}'][Current_state]
                change = np.random.choice( self.G[Current_state], replace= False, p = probs)
                self.Sample_list[self.Sample_Index].append(change)
                self.Sample_list[self.Sample_Index] = list(set(self.Sample_list[self.Sample_Index]))
                Current_state = change

        else:
            for _ in range(int(proportion_of_nodes)) :
                if self.Sample_prob_val[f'{vec_func.__name__}'].get(Current_state)  is None:
                    probs = self.to_vec_prob(node_i=Current_state,
                    vec_func = vec_func,alpha_j = alpha_j)
                    self.Sample_prob_val[f'{vec_func.__name__}'][Current_state] = probs
                probs =  self.Sample_prob_val[f'{vec_func.__name__}'][Current_state]
                change = np.random.choice( self.G[Current_state], replace= False, p =probs)
                self.Sample_list[self.Sample_Index].append(change)
                self.Sample_list[self.Sample_Index] = list(set(self.Sample_list[self.Sample_Index]))
                Current_state = change
        t_end  = time.perf_counter()
        print(f'Session end...{t_end-t_start}')
        self.Sample_Index +=1
        return self.Sample_list[-1]

    def sub_graph(self, nodes):
        #nodes of the subgraph
        sub_G = self.G.subgraph(nodes)
        return sub_G

    def closeness_criteria(self, sub_G):
        # return values
        GS = nx.closeness_centrality(self.G)
        GS_mean = np.mean(list(GS.values()))

        GS_sub = nx.closeness_centrality(sub_G)
        GS_sub_mean = np.mean(list(GS_sub.values()))
        return (GS_mean - GS_sub_mean)**2

    def betweenness_criteria(self,sub_G):
        # return values
        GS = nx.betweenness_centrality(self.G, weight = 'weight')
        GS_mean = np.mean(list(GS.values()))

        GS_sub = nx.betweenness_centrality(sub_G, weight = 'weight')
        GS_sub_mean = np.mean(list(GS_sub.values()))
```

```
168         return (GS_mean - GS_sub_mean)**2
169
170     def betweenness_dif_percent(self,sub_G):
171         # return value
172         GS = nx.betweenness_centrality(self.G, weight = 'weight')
173         GS_mean = np.mean(list(GS.values()))
174
175         GS_sub = nx.betweenness_centrality(sub_G, weight = 'weight')
176         GS_sub_mean = np.mean(list(GS_sub.values()))
177         return abs(GS_mean - GS_sub_mean)/GS_mean *100
178
179     def closeness_dif_percent(self, sub_G):
180         # return values
181         GS = nx.closeness_centrality(self.G)
182         GS_mean = np.mean(list(GS.values()))
183
184         GS_sub = nx.closeness_centrality(sub_G)
185         GS_sub_mean = np.mean(list(GS_sub.values()))
186         return abs(GS_mean - GS_sub_mean)/GS_mean *100
187
188     def Best_sampler(self, samples = None, vec_fun, alpha_fac = None, N=100):
189
190         if samples is None:
191             rd_inits = self.random_node_generator(N)
192             samples = [self.MCMC_Sample_generator(Initial_state=state, proportion=0.8,
193             vec_func= vec_fun) for state in rd_inits]
194
195         sample_node_ls = []
196         for val in samples:
197             sample_node_ls+=val
198         sample_node_ls = list(set(sample_node_ls))
199         node_occurace = {val:len([1 for sample in samples if val in sample])
200         for val in sample_node_ls}
201         alpha = min([len(val)/self.totNodes  for val in samples])
202         if not alpha_fac is None:
203             alpha *= alpha_fac
204         best_sample = [val for val in node_occurace.keys() if node_occurace[val] > alpha*N]
205         return best_sample
```

# Bibliography


[Bav50] Alex Bavelas. Communication Patterns in Task-Oriented Groups. *Acoustical Society of America Journal*, 22(6):725, January 1950.

[HSS08] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[HSSC08] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[Kun] Jérôme Kunegis. Koblenz network collection project. Available at `http://konect.cc/networks/` (2013/05/12).

[PNEG17] Benedict Paten, Adam Novak, Jordan Eizenga, and Erik Garrison. Genome graphs and the evolution of genome inference. *Genome research*, 27, 03 2017.

[PW20] Donald B. Percival and Andrew T. Walden. *References*, page 643–660. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 2020.

[Spa91] Malcolm K. Sparrow. The application of network analysis to criminal intelligence: An assessment of the prospects. *Social Networks*, 13(3):251–274, 1991.

[SSM22] John Sigbeku, Emil Saucan, and Anthea Monod. Curved markov chain monte carlo for network learning. In Rosa Maria Benito, Chantal Cherifi, Hocine Cherifi, Esteban Moro, Luis M. Rocha, and Marta Sales-Pardo, editors, *Complex Networks & Their Applications X*, pages 461–473, Cham, 2022. Springer International Publishing.

[VB20] Emil Saucan Vladislav Barkanass, Jürgen Jost. Geometric sampling of networks. *Social Networks*, 1:1–50, 2020.

[VVLDCC18] Katy Vandereyken, Jelle Van Leene, Barbara De Coninck, and Bruno Cammue. Hub protein controversy: Taking a closer look at plant stress response hubs. *Frontiers in Plant Science*, 9:694, 06 2018.

[Wic] Rick Wicklin. Statistical programming with sas/iml software. Statistical programming in SAS with an emphasis on SAS/ML program. Availabel at `https://blogs.xsas.com/content/iml/2014/04/28/how-much-ram-do-i-need-to-store-that-matrix.html` (2014).